Chapter 2

Mutual Exclusion Problem

2.1 Introduction

When processes share data, it is important to synchronize their access to the data so that updates are not lost as a result of concurrent accesses and the data are not corrupted. This can be seen from the following example. Assume that the initial value of a shared variable x is 0 and that there are two processes, P_0 and P_1 such that each one of them increments x by the following statement in some high-level programming language:

$$x = x + 1$$

It is natural for the programmer to assume that the final value of x is 2 after both the processes have executed. However, this may not happen if the programmer does not ensure that $x = x + 1$ is executed atomically. The statement $x = x + 1$ may compile into the machine-level code of the form

```
LD R, x    ; load register R from x
INC R      ; increment register R
ST R, x    ; store register R to x
```

Now the execution of P_0 and P_1 may get interleaved as follows:

```
P0: LD R, x                ; load register R from x
P0: INC R                  ; increment register R
                P1: LD R, x ; load register R from x
                P1: INC R   ; increment register R
P0: ST R,x                 ; store register R to x
                P1: ST R,x  ; store register R to x
```

Thus both processes load the value 0 into their registers and finally store 1 into x resulting in the "lost update" problem.

To avoid this problem, the statement $x = x + 1$ should be executed `atomically`. A section of the code that needs to be executed atomically is also called a *critical region* or a *critical section*. The problem of ensuring that a critical section is executed atomically is called the *mutual exclusion problem*. This is one of the most fundamental problems in concurrent computing and we will study it in detail.

The mutual exclusion problem can be abstracted as follows. We are required to implement the interface shown in Figure 2.1. A process that wants to enter the critical section (CS) makes a call to `requestCS` with its own identifier as the argument. The process or the thread that makes this call returns from this method only when it has the exclusive access to the critical section. When the process has finished accessing the critical section, it makes a call to the method `releaseCS`.

```
public interface Lock {
    public void requestCS(int pid); //may block
    public void releaseCS(int pid);
}
```

Figure 2.1: Interface for accessing the critical section

The entry protocol given by the method `requestCS` and the exit protocol given by the method `releaseCS` should be such that the mutual exclusion is not violated.

To test the Lock, we use the program shown in Figure 2.2. This program tests the Bakery algorithm that will be presented later. The user of the program may test a different algorithm for a lock implementation by invoking the constructor of that lock implementation. The program launches N threads as specified by `arg[0]`. Each thread is an object of the class `MyThread`. Let us now look at the class `MyThread`. This class has two methods, `nonCriticalSection` and `CriticalSection`, and it overrides the `run` method of the `Thread` class as follows. Each thread repeatedly enters the critical section. After exiting from the critical section it spends an undetermined amount of time in the noncritical section of the code. In our example, we simply use a random number to sleep in the critical and the noncritical sections.

Let us now look at some possible protocols, one may attempt, to solve the mutual exclusion problem. For simplicity we first assume that there are only two processes, P_0 and P_1.

2.2 Peterson's Algorithm

Our first attempt would be to use a shared boolean variable `openDoor` initialized to `true`. The entry protocol would be to wait for `openDoor` to be true. If it is true, then a process can enter the critical section after setting it to `false`. On exit, the process resets it to `true`. This algorithm is shown in Figure 2.3.

This attempt does not work because the testing of `openDoor` and setting it to `false` is not done atomically. Conceivably, one process might check for the `openDoor` and go past the `while` statement in Figure 2.3. However, before that process could set `openDoor` to `false`, the other process starts executing. The other process now checks for the value of `openDoor` and also gets out of busy wait. Both the processes now can set `openDoor` to false and enter the critical section. Thus, mutual exclusion is violated.

In the attempt described above, the shared variable did not record who set the `openDoor` to false. One may try to fix this problem by keeping two shared variables, `wantCS[0]` and `wantCS[1]`, as shown in Figure 2.4. Every process P_i first sets its own `wantCS` bit to true at line 3 and then waits until the `wantCS` for the other process is false at line 4. We have used $1 - i$ to get the process identifier of the other process when there are only two processes - P_0 and P_1. To release the critical section, P_i simply resets its `wantCS` bit to false. Unfortunately, this attempt also does not work. Both processes could set their `wantCS` to true and then indefinitely loop, waiting for the other process to set its `wantCS` false.

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`.

This protocol does guarantee *mutual exclusion*. It also guarantees that if both processes are trying to

```
import java.util.Random;
public class MyThread extends Thread {
    int myId;
    Lock lock;
    Random r = new Random();
    public MyThread(int id, Lock lock) {
        myId = id;
        this.lock = lock;
    }
    void nonCriticalSection() {
        System.out.println(myId + " is not in CS");
        Util.mySleep(r.nextInt(1000));
    }
    void CriticalSection() {
        System.out.println(myId + " is in CS *****");
        // critical section code
        Util.mySleep(r.nextInt(1000));
    }
    public void run() {
        while (true) {
            lock.requestCS(myId);
            CriticalSection();
            lock.releaseCS(myId);
            nonCriticalSection();
        }
    }
    public static void main(String[] args) throws Exception {
        MyThread t[];
        int N = Integer.parseInt(args[0]);
        t = new MyThread[N];
        Lock lock = new Bakery(N);//or any other mutex algorithm
        for (int i = 0; i < N; i++) {
            t[i] = new MyThread(i, lock);
            t[i].start();
        }
    }
}
```

Figure 2.2: A program to test mutual exclusion

```
class Attempt1 implements Lock {
    boolean openDoor = true;
    public void requestCS(int i) {
        while (!openDoor) ; // busy wait
        openDoor = false;
    }
    public void releaseCS(int i) {
        openDoor = true;
    }
}
```

Figure 2.3: An attempt that violates mutual exclusion

```
1  class Attempt2 implements Lock {
2      boolean wantCS[] = {false, false};
3      public void requestCS(int i) { // entry protocol
4          wantCS[i] = true;    //declare intent
5          while (wantCS[1 - i]) ; // busy wait
6      }
7      public void releaseCS(int i) {
8          wantCS[i] = false;
9      }
10 }
```

Figure 2.4: An attempt that can deadlock

```
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

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:

```
(wantCS[j] && (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.

Intuitively, Peterson's algorithm uses the order of updates to `turn` to resolve the contention. If both processes are interested in the critical section, then the process that updated `turn` last, loses and is required to wait.

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.

```
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

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 prove that mutual exclusion is satisfied by Peterson's algorithm by the method of contradiction. Suppose, if possible, both processes P_0 and P_1 are in the critical section for some execution. Each of the processes P_i must have set the variable *turn* to $1 - i$. Without loss of generality, assume that P_1 was the last process to set the variable *turn*. This means that the value of *turn* was 0 when P_1 checked the entry condition for the critical section. Since P_1 entered the critical section in spite of *turn* being 0, it must have read $wantCS[0]$ to be false. Therefore, we have the following sequence of events:
P_0 sets *turn* to 1, P_1 sets turn to 0, P_1 reads $wantCS[0]$ as false. However, P_0 sets the *turn* variable to 1 after setting $wantCS[0]$ to true. Since there are no other writes to $wantCS[0]$, P_1 reading it as false gives us the desired 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

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

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

A crucial disadvantage of Peterson's algorithm is that it uses shared variables that may be written by multiple writers. Specifically, the correctness of Peterson's algorithm depends on the fact that concurrent writes to the *last* variables result in a valid value.

We now describe Lamport's bakery algorithm, which overcomes this disadvantage. The algorithm is similar to that used by bakeries in serving customers. Each customer who arrives at the bakery receives a number. The server serves the customer with the smallest number. In a concurrent system, it is difficult

to ensure that every process gets a unique number. So in case of a tie, we use process ids to choose the smaller process.

The algorithm shown in Figure 2.7 requires a process P_i to go through two main steps before it can enter the critical section. In the first step (lines 15–21), it is required to choose a number. To do that, it reads the numbers of all other processes and chooses its number as one bigger than the maximum number it read. We will call this step the *doorway*. In the second step the process P_i checks if it can enter the critical section as follows. For every other process P_j, process P_i first checks whether P_j is currently in the doorway at line 25. If P_j is in the doorway, then P_i waits for P_j to get out of the doorway. At lines 26–29, P_i waits for the $number[j]$ to be 0 or $(number[i], i) < (number[j], j)$. When P_i is successful in verifying this condition for all other processes, it can enter the critical section.

```
 1  class Bakery implements Lock {
 2      int N;
 3      boolean[] choosing; // inside doorway
 4      int[] number;
 5      public Bakery(int numProc) {
 6          N = numProc;
 7          choosing = new boolean[N];
 8          number = new int[N];
 9          for (int j = 0; j < N; j++) {
10              choosing[j] = false;
11              number[j] = 0;
12          }
13      }
14      public void requestCS(int i) {
15          // step 1: doorway: choose a number
16          choosing[i] = true;
17          for (int j = 0; j < N; j++)
18              if (number[j] > number[i])
19                  number[i] = number[j];
20          number[i]++;
21          choosing[i] = false;
22
23          // step 2: check if my number is the smallest
24          for (int j = 0; j < N; j++) {
25              while (choosing[j]) ; // process j in doorway
26              while ((number[j] != 0) &&
27                      ((number[j] < number[i]) ||
28                      ((number[j] == number[i]) && j < i)))
29                  ; // busy wait
30          }
31      }
32      public void releaseCS(int i) { // exit protocol
33          number[i] = 0;
34      }
35  }
```

Figure 2.7: Lamport's bakery algorithm

We first prove the assertion:

(A1) If a process P_i is in critical section and some other process P_k has already chosen its number, then $(number[i], i) < (number[k], k)$.

Let t be the time when P_i read the value of $choosing[k]$ to be $false$. If P_k had chosen its number before t, then P_i must read P_k's number correctly. Since P_i managed to get out of the kth iteration of the *for* loop, $((number[i], i) < (number[k], k))$ at that iteration. If P_k had chosen its number after t, then P_k must have read the latest value of $number[i]$ and is guaranteed to have $number[k] > number[i]$. If $((number[i], i) < (number[k], k))$ at the kth iteration, this will continue to hold because $number[i]$ does not change and $number[k]$ can only increase.

We now claim the assertion:

(A2) If a process P_i is in critical section, then $(number[i] > 0)$.

(A2) is true because it is clear from the program text that the value of any number is at least 0 and a process executes increment operation on its number at line 20 before entering the critical section.

Showing that the bakery algorithm satisfies mutual exclusion is now trivial. If two processes P_i and P_k are in critical section, then from (A2) we know that both of their numbers are nonzero. From (A1) it follows that $(number[i], i) < (number[k], k)$ and vice versa, which is a contradiction.

The bakery algorithm also satisfies starvation freedom because any process that is waiting to enter the critical section will eventually have the smallest nonzero number. This process will then succeed in entering the critical section.

It can be shown that the bakery algorithm does not make any assumptions on *atomicity* of any read or write operation. Note that the bakery algorithm does not use any variable that can be written by more than one process. Process P_i writes only on variables $number[i]$ and $choose[i]$.

There are two main disadvantages of the bakery algorithm: (1) it requires $O(N)$ work by each process to obtain the lock even if there is no contention, and (2) it requires each process to use timestamps that are unbounded in size.

2.4 Hardware Solutions

As we have seen, pure software solutions to mutual exclusion can be quite complex and expensive. However, mutual exclusion can be provided quite easily with the help of hardware. We discuss some techniques below.

2.4.1 Disabling Interrupts

In a single-CPU system, a process may disable all the interrupts before entering the critical section. This means that the process cannot be context-switched (because context switching occurs when the currently running thread receives a clock interrupt when its current timeslice is over). On exiting the critical section, the process enables interrupts. Although this method can work for a single-CPU machine, it has many undesirable features. First, it is infeasible for a multiple-CPU system in which even if interrupts are disabled in one CPU, another CPU may execute. Disabling interrupts of all CPUs is very expensive. Also, many system facilities such as clock registers are maintained using hardware interrupts. If interrupts are disabled, then these registers may not show correct values. Disabling interrupts can also lead to problems if the user process has a bug such as an infinite loop inside the critical section.

2.4.2 Instructions with Higher Atomicity

Most machines provide instructions with a higher level of atomicity than *read* or *write*. The `testAndSet` instruction provided by some machines does both read and write in one atomic instruction. This instruction

reads and returns the old value of a memory location while replacing it with a new value. We can abstract the instruction as a `testAndSet` method on an object of the class `TestAndSet` as shown in Figure 2.8.

```
public class TestAndSet {
    int myValue = -1;
    public synchronized int testAndSet(int newValue) {
        int oldValue = myValue;
        myValue = newValue;
        return oldValue;
    }
}
```

Figure 2.8: TestAndSet hardware instruction

If the `testAndSet` instruction is available, then one can develop a very simple protocol for mutual exclusion as shown in Figure 2.9.

```
class HWMutex implements Lock {
    TestAndSet lockFlag;
    public void requestCS(int i) { // entry protocol
        while (lockFlag.testAndSet(1) == 1) ;
    }
    public void releaseCS(int i) { // exit protocol
        lockFlag.testAndSet(0);
    }
}
```

Figure 2.9: Mutual exclusion using TestAndSet

This algorithm satisfies the mutual exclusion and progress property. However, it does not satisfy starvation freedom. Developing such a protocol is left as an exercise.

Sometimes machines provide the instruction `swap`, which can swap two memory locations in one atomic step. Its semantics is shown in Figure 2.10. The reader is invited to design a mutual exclusion protocol using `swap`.

```
public class Synch{
    public static synchronized void swap(Boolean m1, Boolean m2){
        Boolean temp = m1;
        m1 = m2;
        m2 = temp;
     }
}
```

Figure 2.10: Semantics of `swap` operation

2.5 Problems

```
class Dekker implements Lock {
    boolean wantCS[] = {false, false};
    int turn = 1;
    public void requestCS(int i) { // entry protocol
        int j = 1 - i;
        wantCS[i] = true;
        while (wantCS[j]) {
            if (turn == j) {
                wantCS[i] = false;
                while (turn == j) ;// busy wait
                wantCS[i] = true;
            }
        }
    }
    public void releaseCS(int i) { // exit protocol
        turn = 1 - i;
        wantCS[i] = false;
    }
}
```

Figure 2.11: Dekker.java

2.1. Show that any of the following modifications to Peterson's algorithm makes it incorrect:

(a) A process in Peterson's algorithm sets the *turn* variable to itself instead of setting it to the other process.

(b) A process sets the *turn* variable before setting the *wantCS* variable.

2.2. Show that Peterson's algorithm also guarantees freedom from starvation.

2.3. Show that the bakery algorithm does not work in absence of *choosing* variables.

2.4. Consider the software protocol shown in Figure 2.11 for mutual exclusion between two processes. Does this protocol satisfy (a) mutual exclusion, and (b) livelock freedom (both processes trying to enter the critical section and none of them succeeding)? Does it satisfy starvation freedom?

2.5. Modify the bakery algorithm to solve k-mutual exclusion problem, in which at most k processes can be in the critical section concurrently.

2.6. Give a mutual exclusion algorithm that uses atomic swap instruction.

2.7. Give a mutual exclusion algorithm that uses TestAndSet instruction and is free from starvation.

*2.8. Give a mutual exclusion algorithm on N processes that requires $O(1)$ time in absence of contention.

2.6 Bibliographic Remarks

The mutual exclusion problem was first introduced by Dijkstra [Dij65a]. Dekker developed the algorithm for mutual exclusion for two processes. Dijkstra [Dij65b] gave the first solution to the problem for N processes. The bakery algorithm is due to Lamport [Lam74], and Peterson's algorithm is taken from a paper by Peterson [Pet81].