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:

P_0: LD R, x		; load register R from x
P_0: INC R		; increment register R
	P_1: LD R, x	; load register R from x
	P_1: INC R	; increment register R
P_0: ST R,x		; store register R to x
	P_1: 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

```
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

```
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.

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.

```
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

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.

We give a second proof of mutual exclusion due to Dijkstra. This proof does not reason on the sequence of events; it uses an assertional proof. For the purposes of this 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.

Consider the predicate $H(0)$ defined as

$$H(0) \equiv wantCS[0] \wedge [(turn = 1) \vee ((turn = 0) \wedge trying[1])]$$

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

$$H(1) \equiv wantCS[1] \wedge [(turn = 0) \vee ((turn = 1) \wedge trying[0])]$$

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 $(turn = 1) \wedge trying[0]$ true. So, the only case left is falsification of $(turn = 1) \wedge trying[0]$. Since $wantCS[1]$ is also true, we look at falsification of $(turn = 1) \wedge trying[0] \wedge wantCS[1]$. P_0 can falsify this only by setting $trying[0]$ to false (i.e., by acquiring the permission to enter the critical section). But, $(turn = 1) \wedge (wantCS[1])$ 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 $\neg trying[0] \wedge H(0) \wedge \neg trying[1] \wedge H(1)$, which implies $(turn = 0) \wedge (turn = 1)$, 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

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

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 Filter Algorithm

We now extend Peterson's algorithm for more than two processes. The first difficulty that we face is that in Peterson's algorithm, a process would set the *turn* variable to that of the other process before checking the entry condition. Now that there are more than two processes, it is not clear how to set the *turn* variable. Therefore, instead of the *turn* variable, we use the variable *last* which stores the pid of the last process that wrote to that variable. Now a process can use *last* to wait whenever there is a conflict. In a system of two processes, one of them will wait and the other one can enter the critical section. If there are $N > 2$ processes, then one of them will wait and the remaining can get through. Note that we have managed to reduce the number of contending processes from N to $N-1$. By applying this idea $N-1$ times, we can reduce the number of active processes from N to 1.

Figure 2.7 shows the Filter Algorithm. We have $N-1$ gates numbered $1 \ldots N-1$. For each thread i, variable $gate[i]$ stores the gate that thread is trying to enter. Initially, $gate[i]$ is 0 for each i. For each gate k, we use $last[k]$ to store the last process that tries to enter the gate (i.e. writes on the variable $last[k]$). In `requestCS` method, P_i goes through a gate k as follows. P_i checks whether there is any process P_j which is at that gate or a higher numbered gate and P_i is the last one to arrive at that gate. Under this condition, P_i waits by continually rechecking the condition. It will exit from the *while* loop only if $gate[j]$ becomes lower than $gate[i]$ when P_j exits the critical section or the variable $last[k]$ is changed.

Since this algorithm is a minor generalization of Peterson's algorithm, we leave its proof of correctness as an exercise.

2.4 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 2.8 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

```
import java.util.Arrays;

class PetersonN implements Lock {
    int N;
    int[] gate;
    int[] last;
    public PetersonN(int numProc) {
        N = numProc;
        gate = new int[N]; //We only use gate[1]..gate[N-1]; gate[0] is unused
        Arrays.fill(gate, 0);
        last = new int[N];
        Arrays.fill(last, 0);
    }
    public void requestCS(int i) {
        for (int k = 1; k < N; k++) {
            gate[i] = k;
            last[k] = i;
            for (int j = 0; j < N; j++) {
                while ((j != i) &&  // there is some other process
                       (gate[j] >= k) &&  // that is ahead or at the same level
                       (last[k] == i)) // and I am the last to update last[k]
                {};// busy wait
            }
        }
    }
    public void releaseCS(int i) {
        gate[i] = 0;
    }
}
```

Figure 2.7: PetersonN.java

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 2.8: 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.

2.5 Lower Bound on the Number of Shared Memory Locations

In this section, we show that any algorithm that solves the mutual exclusion problem for n processes must use at least n memory locations. The key idea in showing the lower bound is that the system never gets into an *inconsistent* state in which a thread is not able to determine by reading shared locations whether the critical section is empty or not.

Consider a system with two processes, P and Q. Suppose that there is a protocol that uses a single shared location A to coordinate access to the critical section. It is clear that a thread must write to A before entering the critical section otherwise the other thread would not be able to distinguish this state from the state in which the critical section is empty. Now, we consider an adversarial schedule as follows. Suppose that P is about to write to A before entering the CS. We now let process Q execute its protocol, possibly writes on the location A, and enter the CS. We now resume process P which writes on location A overwriting anything that Q may have written. At this point, the system is in an inconsistent state because even though Q is in the CS, it is indistinguishable from the state in which the CS is available.

It is instructive to extend this argument for three processes P, Q and R using two shared locations A and B. By previous argument any correct protocol for two processes must use both the locations. By letting P run three times, we know that there exists an execution in which P writes on some location, say A, twice. We first run P till it is ready to write to A for the first time. Then, we run Q till it is ready to write in a separate location, say B, for the first time before entering the CS. If Q does not write at any location other than A and enters CS, P can overwrite what Q wrote and also enter the CS. At this point Q is about to write on B and P is about to write on A. We now run P again. Since P overwrites on A and nothing has been written on B; it cannot tell whether Q has taken any step so far. We let P enter the CS and then request CS again till it is about to write on A again. At this point both A and B are in state consistent with no process in the CS. Next, we let R run and enter the CS. Then, we run P and Q for one step thereby overwriting any change that R may have done. One of them must be able to enter the CS to keep the algorithm deadlock-free. We have a violation of mutual exclusion in that state because R is already in the CS.

2.6 Fischer's Algorithm

Our lower bound result assumed that processes are asynchronous. We now give an algorithm that uses timing assumptions to provide mutual exclusion with a single shared variable *turn*. The variable *turn* is either -1 signifying that the critical section is available or has the identifier of the process that has the right to enter the critical section. Whenever any process P_i finds that *turn* is -1, it must set *turn* to i in at most c time units. It must then wait for at least *delta* units of time before checking the variable *turn* again. The algorithm requires *delta* to be greater than c. If *turn* is still set to i, then it can enter the critical section.

```
public class Fischer implements Lock {
        int N;
        int turn;
        int delta;

        public Fischer (int numProc) {
                N = numProc;
                turn = -1;
                delta = 5;
        }
        public void requestCS(int i) {
                while (true) {
                        while (turn != -1) {}; // wait for the door to open
                        turn = i; // write my id on turn
                        try { // Assume that delta is bigger than time to update turn
                                Thread.sleep(delta);
                        }
                        catch (InterruptedException e){};
                        if (turn == i) return;
                }
        }
        public void releaseCS(int i) {
                turn = -1;
        }
}
```

Figure 2.9: Fischer's mutual exclusion algorithm

We first show mutual exclusion. Suppose that both P_i and P_j are in the CS. Suppose that *turn* is i. This means that *turn* must have been j when P_j entered and then later *turn* was set to i. But P_i can set *turn* to i only within c time units of P_j setting *turn* to j. However, P_j found *turn* to be j even after $d \geq c$ time units. Hence, both P_i and P_j cannot be in CS.

We leave the proof of deadlock-freedom as an exercise.

2.7 A Fast Mutex Algorithm

We now present an algorithm due to Lamport that allows fast accesses to critical section in absence of contention. The algorithm uses an idea called *splitter* that is of independent interest.

2.7.1 Splitter

A *splitter* is a method that splits processes into three disjoint groups: *Left*, *Right*, and *Down*. We can visualize a splitter as a box such that processes enter from the top and either move to the left, the right or go down which explains the names of the groups. The key property a splitter satisfies is that at most one process goes in the down direction and not all processes go in the left or the right direction.

The algorithm for the splitter is shown in Fig. 2.10.

```
Pi::
 var
     door: {open, closed} initially open
     last : pid initially -1;

 last := i;
 if (door == closed)
     return Left;
 else
     door := closed;
     if (last == i) return Down;
     else return Right;
 end
```

Figure 2.10: Splitter Algorithm

A splitter consists of two variables: *door* and *last*. The door is initially open and if any process finishes executing splitter the door gets closed. The variable *last* records the last process that executed the statement *last := i*.

Each process P_i first records its pid in the variable *last*. It then checks if the door is closed. All processes that find the door closed are put in the group *Left*. We claim

Lemma 2.1 *There are at most $n-1$ processes that return* Left.

Proof: Initially, the door is open. At least one process must find the door to be open because every process checks the door to be open before closing it. Since at least one process finds the door open, it follows that $|Left| \leq n-1$.

■

Process P_i that find the door open checks if the last variable contains its pid. If this is the case, then the process goes in the *Down* direction. Otherwise, it goes in the *Right* direction.

We now have the following claim.

Lemma 2.2 *At most one process can return* Down.

Proof: Suppose that P_i be the first process that finds the door to be open and *last* equal to i (and then later returns *Down*). We have the following order of events: P_i wrote *last* variable, P_i closed the door, P_i read *last* variable as i. During this interval, no process P_j modified the last variable. Any process that modifies *last* after this interval will find the door closed and therefore cannot return *Down*. Consider any process P_j that modifies *last* before this interval. If P_j checks *last* before the interval, then P_i is not the first process then finds last as itself. If P_j checks *last* after P_i has written the variable last, then it cannot find itself as the last process since its pid was overwritten by P_i.

■

Lemma 2.3 *There are at most $n-1$ processes that return* Right*.*

Proof: Consider the last process that wrote its index in *last*. If it finds the door closed, then that process goes left. If it finds the door open then it goes down.

■

Note that the above code does not use any synchronization. In addition, the code does not have any loop.

2.7.2 Lamport's Fast Mutex Algorithm

Lamport's fast mutex algorithm shown in Fig. 2.11 uses two shared registers X and Y that every process can read and write. A process P_i can acquire the critical section either using a *fast* path when it finds $X = i$ or using a *slow* path when it finds $Y = i$. It also uses n shared single-writer-multiple-reader registers $flag[i]$. The variable $flag[i]$ is set to value *up* if P_i is actively contending for mutual exclusion using the fast path. The shared variable X plays the role of the variable *last* in the splitter algorithm. The variable Y plays the role of door in the splitter code. When Y is -1, the door is open. A process P_i closes the door by updating Y with i.

Processes that are in the group *Left* of the splitter, simply retry. Before retrying, they lower their flag and wait for the door to be open (i.e. Y to be -1). A process that is in the group *Down* of the splitter succeeds in entering the critical section. Note that at most process may succeed using this route. Processes that are in the group *Right* of the splitter first wait for all flags to go down. This can happen only if no process returned *Down*, or if the process that returned *Down* releases the critical section. Now consider the last process in the group *Right* to update Y. That process will find its pid in Y and can enter the critical section. All other processes wait for the door to be open again and then retry.

Theorem 2.4 *Fast Mutex algorithm in Fig. 2.11 satisfies Mutex.*

Proof: Suppose P_i is in the critical section. This means that Y is not -1 and P_i exited either with $X = i$ or $Y = i$.

Any process that finds $Y \neq -1$ gets stuck till Y becomes -1 so we can focus on processes that found Y equal to -1.

Case 1: P_i entered with $X = i$

Its flag stays up and thus other processes stay blocked.

Case 2: P_i entered with $Y = i$

Consider any P_j which read $Y == -1$. X is not equal to j otherwise P_j would have entered CS and P_i would have been blocked

Since $Y = i$, P_j would get blocked waiting for Y to become -1.

■

Theorem 2.5 *Fast Mutex algorithm in Fig. 2.11 satisfies deadlock-freedom.*

Proof:

Consider processes that found the door open, i.e., Y to be -1. Let Q be the set of processes that are stuck that found the door open. If any one of them succeeded in "last-to-write-X" we are done; otherwise, the last process that wrote Y can enter the CS.

■

```
var
    X, Y: int initially -1;
    flag: array[1..n] of {down, up};

acquire(int i)
{
 while (true)
    flag[i] := up;
    X := i;
    if (Y != -1) { // splitter's left
        flag[i] := down;
        waitUntil(Y == -1)
        continue;
    }
    else {
        Y := i;
        if (X == i) // success with splitter
            return; // fast path
        else {// splitter's right
            flag[i] := down;
            forall j:
                waitUntil(flag[j] == down);
            if (Y == i) return; // slow path
            else {
                waitUntil(Y == -1);
                continue;
            }
        }
 }
}

 release(int i)
 {
    Y := -1;
    flag[i] := down;
 }
```

Figure 2.11: Lamport's Fast Mutex Algorithm

2.8 Locks with Get-and-Set Operation

Although locks for mutual exclusion can be built using simple read and write instructions, any such algorithm requires as many memory locations as the number of threads. By using instructions with higher atomicity, it is much easier to build locks. For example, the `getAndSet` operation (also called `testAndSet`) allows us to build a lock as shown in Fig. 2.12.

```
import java.util.concurrent.atomic.*;

public class GetAndSet implements MyLock {
    AtomicBoolean isOccupied = new AtomicBoolean(false);
    public void lock() {
        while (isOccupied.getAndSet(true)) {
            Thread.yield();
           // skip();
        }
    }
    public void unlock() {
        isOccupied.set(false);
    }
}
```

Figure 2.12: Building Locks Using GetAndSet

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.

Most modern machines provide the instruction `compareAndSet` which takes as argument an expected value and a new value. It atomically sets the current value to the new value if the current value equals the expected value. It also returns true if it succeeded in setting the current value; otherwise, it returns false. The reader is invited to design a mutual exclusion protocol using `compareAndSet`.

We now consider an alternative implementation of locks using getAndSet operation. In this implementation, a thread first checks if the lock is available using the `get` operation. It calls the `getAndSet` operation only when it finds the critical section available. If it succeeds in `getAndSet`, then it enters the critical section; otherwise, it goes back to *spinning* on the get operation. The implementation called GetAndGetAndSet (or testAndTestAndSet) is shown in Fig. 2.13.

```
import java.util.concurrent.atomic.*;

public class GetAndGetAndSet implements MyLock {
    AtomicBoolean isOccupied = new AtomicBoolean(false);
    public void lock() {
        while (true) {
           while (isOccupied.get()) {
           }
           if (!isOccupied.getAndSet(true)) return;
        }
    }
    public void unlock() {
        isOccupied.set(false);
    }
}
```

Figure 2.13: Building Locks Using GetAndGetAndSet

Although the implementations in Fig. 2.12 and Fig. 2.13 are functionally equivalent, the second implementation usually results in faster accesses to the critical section on current multiprocessors. Can you guess why?

The answer to the above question is based on the current architectures that use a shared bus and a local cache with each core. Since an access to the shared memory via bus is much slower compared to an access to the local cache, each core checks for a data item in its cache before issuing a memory request. Any data item that is found in the local cache is termed as a *cache hit* and can be served locally. Otherwise, we get a *cache miss* and the item must be served from the main memory or cache of some other core. Caches improve the performance of the program but require that the system ensures coherence and consistency of cache. In particular, an instruction such as getAndSet requires that all other cores should invalidate their local copies of the data item on which getAndSet is called. When cores spin using getAndSet instruction, they repeatedly access the bus resulting in high contention and a slow down of the system. In the second implementation, threads spin on the variable `isOccupied` using get. If the memory location corresponding to `isOccupied` is in cache, the thread only reads cached value and therefore avoids the use of the shared data bus.

Even though the idea of getAndGetAndSet reduces contention of the bus, it still suffers from high contention whenever a thread exits the critical section. Suppose that a large number of threads were spinning on their *cached* copies of `isOccupied`. Now suppose that the thread that had the lock leaves the critical section. When it updates `isOccupied`, the cached copies of all spinning threads get invalidated. All these threads now get that `isOccupied` is false and try to set it to true using `getAndSet`. Only, one of them succeeds but all of them end up contributing to the contention on the bus. An idea that is useful in reducing the contention is called *backoff*. Whenever a thread finds that it failed in getAndSet after a successful get, instead of continuing to get the lock, it backs off for a certain random period of time. The *exponential* backoff doubles the maximum period of time a thread may have to wait after any unsuccessful trial. The resulting implementation is shown in Fig. 2.14.

```java
import java.util.concurrent.atomic.*;

public class MutexWithBackOff{
    AtomicBoolean isOccupied = new AtomicBoolean(false);
    public void lock() {
        while (true) {
            while (isOccupied.get()) {
            }
            if (!isOccupied.getAndSet(true)) return;
            else {
                int timeToSleep = calculateDuration();
                Thread.sleep(timeToSleep);
            }
        }
    }
    public void unlock() {
        isOccupied.set(false);
    }
}
```

Figure 2.14: Using Backoff during Lock Acquisition

Another implementation that is sometimes used for building locks is based on getting a ticket number similar to Bakery algorithm. This implementation is also not scalable since it results in high contention for currentTicket.

```java
import java.util.concurrent.atomic.*;

public class TicketMutex {
    AtomicInteger nextTicket = new AtomicInteger(0);
    AtomicInteger currentTicket = new AtomicInteger(0);
    public void lock() {
        int myticket = nextTicket.getAndIncrement();
        while (myticket != currentTicket.get()) {
            // skip();
        }
    }
    public void unlock() {
        int temp = currentTicket.getAndIncrement();
    }
}
```

Figure 2.15: Using Tickets for Mutex

2.9 Queue Locks

Our previous approaches to solve mutual exclusion require threads to spin on the shared memory location `isOccupied`. As we have seen earlier, any update to this variable results in multiple threads getting cache invalidation. Even when threads backoff, we have the issue of deciding the duration of time to back off. If we do not back off for a sufficient period, the bus contention is still there. On the other hand, if the period for the backoff is large, then the threads may be sleeping even when the critical section is not occupied. In this section, we present alternate methods to avoid spinning on the same memory location. All three methods maintain a queue of threads waiting to enter the critical section. Anderson's lock uses a fixed size array, CLH lock uses an implicit linked list and MCS lock uses an explicit linked list for the queue. One of the key challenges in designing these algorithms is that we cannot use locks to update the queue.

2.9.1 Anderson's Lock

Anderson's lock uses a circular array `Available` of size n which is at least as big as the number of threads that may be contending for the critical section. The array is circular so that the index i in the array is always a value in the range $0..n-1$ and is incremented modulo n. Different threads waiting for the critical section spin on the different slots in this array thus avoiding the problem of multiple threads spinning on the same variable. An atomic integer `tailSlot` (initialized to 0) is maintained which points the next available slot in the array. Any thread that wants to lock, reads the value of the `tailSlot` in its local variable `mySlot` and advances it in one atomic operation using `getAndIncrement()`. It then spins on `Available[mySlot]` until the slot becomes available. Whenever a thread finds that the entry for its slot is true, it can enter the critical section. The algorithm maintains the invariant that distinct processes have distinct slots and also that there is at most one entry in `Available` that is true. To unlock, the thread sets its own slot as false and the entry in the next slot to be true. Whenever a thread sets the next slot to be true, any thread that was spinning on that slot can then enter the critical section. Note that in Anderson lock, only one thread is affected when a thread leaves the critical section. Other threads continue to spin on other slots which are cached and thus result only in accesses of local caches.

Note that the above description assumes that each slot is big enough so that adjacent slots do not share a cache line. Hence even though we just need a single bit to store `Available[i]`, it is important to keep it big enough by padding to avoid the problem of *false sharing*. Also note that since Anderson's lock assigns slots to threads in the FCFS manner, it guarantees fairness and therefore freedom from starvation.

A problem with Anderson lock is that it requires a separate array of size n for each lock. Hence, a system that uses m locks shared among n threads will use up $O(nm)$ space.

```
// pseudo-code for Anderson Lock
public class AndersonLock {
    AtomicInteger tailSlot = new AtomicInteger(0);
    boolean [] Available;
    ThreadLocal<Integer> mySlot ;//initialize to 0

    public AndersonLock(int n) { // constructor
     // all Available false except Available[0]
    }
    public void lock() {
      mySlot.set(tailSlot.getAndIncrement() % n);
      spinUntil(Available[mySlot]);
    }
    public void unlock() {
        Available[mySlot.get()] = false;
        Available[(mySlot.get()+1) % n] = true;
    }
}
```

Figure 2.16: Anderson Lock for Mutex

2.9.2 CLH Queue Lock

We can reduce the memory consumption in Anderson's algorithm by using a dynamic linked list instead of a fixed size array. The idea of CLH Lock (shown in Fig. 2.17) is due to Travis Craig, Erik Hagersten and Anders Landin. For each lock, we maintain a linked list. Any thread that wants to get that lock inserts a node in that linked list. Just as we had a shared variable `tailSlot` in Anderson's algorithm, we maintain a shared variable `tailNode` that points to the last node inserted in the linked list. Suppose a thread wants to insert `myNode` in the linked list. This insertion in the linked list must be atomic; therefore, the thread uses `pred = tailNode.getAndSet(myNode)` to insert its node in the linked list as well as get the pointer to the previous node in the variable `pred`. Each node has a field `locked` which is initially set to true when the node is inserted in the linked list. A thread spins on the field `locked` of the predecessor node `pred`. Whenever a thread releases the critical section, it sets the `locked` field to false. Any thread that was spinning on that field can now enter the critical section.

It is important to note that if the thread that exits the critical section wants to enter the critical section again, it cannot reuse its own node because the next thread may still be spinning on that node's field. However, it can reuse the node of its predecessor.

Also note that we do not maintain an explicit pointer in each node. The linked list is virtual. A thread that is spinning has the variable `pred` that points to the predecessor node in the linked list. The node itself has a single field: `locked`.

2.9.3 MCS Queue Lock

In many architectures, each core may have local memory such that access to its local memory is fast but access to the local memory of another core is slower. In such architectures, CLH algorithm results in threads spinning on remote locations. We now show a method due to John Mellor-Crummey and Michael Scott called MCS lock that avoids remote spinning, i.e. spinning on memory of a different core.

```
import java.util.concurrent.atomic.*;
public class CLHLock implements MyLock {
    class Node {
        boolean locked;
    }
    AtomicReference<Node> tailNode;
    ThreadLocal<Node> myNode;
    ThreadLocal<Node> pred;

    public CLHLock() {
        tailNode = new AtomicReference<Node>(new Node());
        tailNode.get().locked = false;
        myNode = new ThreadLocal<Node> () {
            protected Node initialValue() {
                return new Node();
            }
        };
        pred = new ThreadLocal<Node> ();

    }
    public void lock() {
        myNode.get().locked = true;
        pred.set(tailNode.getAndSet(myNode.get()));
        while(pred.get().locked) { Thread.yield(); };
    }
    public void unlock() {
        myNode.get().locked = false;
        myNode.set(pred.get()); // reusing predecessor node for future use
    }
}
```

Figure 2.17: CLH Lock for Mutex

In MCS lock (shown in Fig. 2.18), we maintain a queue based on an explicit linked list. Any thread that wants to access the critical section inserts its node in the linked list at the tail. Each node has a field `locked` which is initialized to true when the node is inserted in the linked list. A thread spins on that field waiting for it to become false. It is the responsibility of the thread that exits the critical section to set this field to false.

When a thread p exits the critical section, it first checks if there is any node linked next to its node. If there is such a node, then it makes the `locked` field false and removes its node from the linked list by making its next pointer null. The tricky case is when it finds that there is no node linked next to its node. There can be two reasons for this. First, there is no thread that has inserted its node in the linked list using `tailNode` yet. In this case, `tailNode` is still pointing to its node. In this case, the thread must simply set the tailNode to null. The second case is that a thread q has called tailNode.getAndSet but not yet set the next pointer of p's node to q's node. In this case, the thread p waits until its next field is not null. It can then set the locked field for that node to be false as before.

```java
import java.util.concurrent.atomic.*;

public class MCSLock implements MyLock {
    class QNode {
        boolean locked;
        QNode next;
        QNode() {
            locked = true;
            next = null;
        }
    }
    AtomicReference<QNode> tailNode = new AtomicReference<QNode>(null);
    ThreadLocal<QNode> myNode;

    public MCSLock() {
        myNode = new ThreadLocal<QNode> () {
            protected QNode initialValue() {
                return new QNode();
            }
        };
    }
    public void lock() {
        QNode pred = tailNode.getAndSet(myNode.get());
        if (pred != null){
            myNode.get().locked = true;
            pred.next = myNode.get();
            while (myNode.get().locked){ Thread.yield(); };
        }
    }
    public void unlock() {
        if (myNode.get().next == null) {
            if (tailNode.compareAndSet(myNode.get(), null)) return;
            while (myNode.get().next == null) { Thread.yield(); };
        }
        myNode.get().next.locked = false;
        myNode.get().next = null;
    }
}
```

Figure 2.18: MCS Lock for Mutex

2.10 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 2.19: 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. Prove the correctness of the Filter algorithm in Figure 2.7. (Hint: Show that at each level, the algorithm guarantees that at least one processes loses in the competition.)

2.5. Consider the software protocol shown in Figure 2.19 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?

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

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

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

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

2.11 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].