

## Chapter 8

# Lock Managers and Mutual Exclusion

The mutual exclusion, or mutex, problem involves  $N$  concurrent processes, each executing code that has “critical sections”. The problem is to specify “entry” and “exit” procedures to surround each critical section such that (1) at most one process is inside a critical section at any time, and (2) any process wanting to enter the critical section eventually does so. A “classical” mutex solution has no processes other than the ones that may access a critical section. There are also mutex solutions which make use of “arbiter” processes.

The mutual exclusion problem was formulated in 19XX by XX (Dijkstra?), assuming only read-write atomicity and no busy waiting (hence no arbiter). Dekker presented the first correct two process solution in 19XX and the first  $N$  process solution in 19XX (ten years later?). Since then, simpler solutions were developed. The simplest two-process solution is probably the Peterson algorithm. The simplest  $N$ -process solution is probably the Bakery algorithm.

Below, we describe a generic classical mutex solution and outline the obvious way to transform the solution to a lock manager. We then do the same for a generic arbiter-based mutex solution.

To reason about these systems, we need to refer to the processes contained in them, for which we need access to the unique abstract ids that the processes get when they are created. In the programs that follow, the type  $\Pi id$  denotes process ids and the variables  $\pi id[0], \dots, \pi id[N - 1]$  hold the ids of the processes ( $\pi id[i]$  is updated when thread  $i$  is created).

### 8.1 Lock managers from classical mutex solutions

A classical mutex solution has the following structure:

```
system-program mutex-csol(int N) {    // generic classical mutex solution
  atomicity-assumptions{ mutex_aa }  // atomicity assumptions of solution

  init() {
    mutex_vars;                       // mutex variables (used only in mutex_entry and mutex_exit)
    other_variables;                  // non-mutex variables (used in NCS, CS)
    for i := 0 to N-1 do {
       $\Pi id$   $\pi id[i]$  := StartThread(code(i));
    }
  }

  function code(int i) {
    forever do {
      NCS: “noncritical section”;
      mutex_entry(i); // entry code; involves only mutex variables; blockable
      CS: “critical section”
      mutex_exit(i); // exit code; involves only mutex variables; no blocking
    }
  }
}
```

```

} // end code(i)

fairness-assumptions{ mutex_fa } // fairness assumptions of solution
} // end mutex-csol

```

The constructs `mutex_vars`, `mutex_entry(i)` and `mutex_exit(i)` are such that the above system satisfies the following safety and progress assertions:

- $\square [\forall 0..N-1 i, j, i \neq j :: \neg(\pi id[i] \text{ in CS} \wedge \pi id[j] \text{ in CS})]$
- $[\forall 0..N-1 i :: \pi id[i] \text{ at CS} \rightsquigarrow \pi id[i] \text{ at mutex\_exit}(i)]$   
 $\Rightarrow [\forall 0..N-1 i :: \pi id[i] \text{ at mutex\_entry}(i) \rightsquigarrow \pi id[i] \text{ at CS}]$

Given the above classical mutex solution for  $N$  processes, here is one way to obtain a lock manager that offers `LockService(N)`:

```

system-program LockMgr-mutex-csol(int N) {
  atomicity-assumptions{ mutex_aa } // atomicity assumptions of mutex solution

  init() {
    mutex_vars; // mutex variables
  }

  function void reqHandler(int i) {
    mutex_entry(i);
    LockService.Acq(i)
  }

  xc-event void Req(int i) {
    maps LockService.Req(i)
    ec 0 ≤ i < N
    ac  $\Pi id \pi id[i] := \text{StartThread}(\text{reqHandler}(i))$ 
  }

  xc-event void Rel(int i) {
    maps LockService.Rel(i)
    ec 0 ≤ i < N
    ac mutex_exit(i)
  }

  fairness-assumptions { mutex_fa }

} // end LockMgr-mutex-csol

```

## 8.2 Lock managers from arbiter-based mutex solutions

An arbiter-based  $N$ -process mutex solution has the following structure:

```

system-program mutex-asol(int N) {
  atomicity-assumptions{ mutex_aa } // atomicity assumptions of solution

  init() {
    mutex_vars; // mutex variables (used only in mutex_entry and mutex_exit)
    other_vars; // non-mutex variables (used in NCS, CS)
    Pid  $\pi \text{arbiter} := \text{StartThread}(\text{MutexArbiter}()); // arbiter process
  }$ 
```

```

    for i := 0 to N-1 do {
        Pid  $\pi$ id[i] := StartThread(code(i)); // contending processes
    }
}

function MutexArbiter() {
    forever do {
        wait for a process i to signal hungry;
        for some hungry process i do { signal okToEat } ;
        wait for process i to signal notEating ;
    }
}

function code(int i) {
    forever do {
        NCS: noncritical section ;
        signal hungry ; // entry code
        wait for okToEat ;
        CS: <critical section> ;
        signal notEating // exit code
    }
} // end code(i)

fairness-assumptions{ mutex_fa } // fairness assumptions of solution
} // end mutex-asol

```

This system satisfies the same safety and progress assertions as with the classical solution. Note that signalling is done via shared variables, and occurs only between a contending process and the arbiter process.

Given the above arbiter-based mutex solution, a lock manager can be obtained simply by wrapping the signalling in event calls.

```

system-program LockMgr-mutex-asol(int N) {
    atomicity-assumptions{ mutex_aa }

    init() {
        mutex_vars; // mutex variables
        Pid  $\pi$ arbiter := StartThread(MutexArbiter()); // start arbiter
    }

    function void MutexArbiter() {
        forever do {
            wait for a process to signal hungry ;
            for some hungry process i do { signal okToEat(i) ; LockService.Acq(i) } ;
            wait for process i to signal notEating ;
        }
    }

    xc-event void Req(int i) {
        maps LockService.Req(i)
        ec 0 ≤ i < N
        ac signal hungry(i)
    }
}

```

```
xc-event void Rel(int i) {
  maps LockService.Rel(i)
  ec  $0 \leq i < N$ 
  ac signal notEating(i)
}

fairness-assumptions { mutex_fa }

} // end LockMgr-mutex-asol
```

### 8.3 Concluding Remarks

TBD

### 8.4 Exercises

- 8.1 Assume you have a classical mutex solution and an assertional proof of it. Construct an assertional proof that the corresponding lock manager satisfies the lock service.
- 8.2 Repeat problem 8.1 for an arbiter-based mutex solution.
- 8.3 Obtain a N-process mutual exclusion solution by repeatedly applying a two-process mutual exclusion solution. Do not use any other synchronization primitives (such as test-and-sets, disabling/enabling interrupts, semaphores, etc.). (Hint: consider a binary tree with N leaves. If you want, assume that N is a power of 2.)

## Chapter 9

# Bakery Algorithm

### 9.1 Introduction

The bakery algorithm is one of the simplest known solutions to the  $N$ -process mutual exclusion problem on a platform that provides read-write atomicity only. The basic idea is that each non-thinking process has a variable that indicates the position of that process in a hypothetical queue of all the non-thinking processes. Each process in this queue scans the variables of the other processes, and enters the critical section only upon determining that it is at the head of the queue.

But the resulting algorithm is still not easy to understand. So below we first look at a simplified version of the bakery algorithm, one that uses a coarser grain of atomicity, namely, atomic read-modify-write (where the read is over  $N$  processes), and prove that it works. Then we look at the original version and prove that it works.

### 9.2 Simplified bakery algorithm

We consider a system of processes  $0, 1, \dots, N-1$ , which use the simplified bakery algorithm to achieve mutually exclusive access to their critical sections. Each process  $i$  has an integer variable  $\text{num}[i]$ , initially 0, that is readable by all processes but writeable by process  $i$  only.

When process  $i$  is thinking,  $\text{num}[i]$  equals zero. When process  $i$  becomes hungry, it sets  $\text{num}[i]$  to a value higher than the  $\text{num}$  of every other process; this operation is assumed to be atomic in this simplified algorithm. Then process  $i$  scans the other processes in order. For each process  $j$ , process  $i$  waits until  $\text{num}[j]$  is either zero or greater than  $\text{num}[i]$ . After going past every process, process  $i$  enters the critical section. Upon leaving the critical section, process  $i$  zeroes  $\text{num}[i]$ .

```
system-program simplified-Bakery(N) {
  init() {
    int[0..N-1] num := 0 ; // num[i]=0 iff process i is thinking
    for i := 0 to N-1 do {
      Pid  $\pi$ id[i] := StartThread(code(i));
    }
  }
}

function code(int i) {
  forever do {
    NCS: noncritical section;
    // entry code
    a1: num[i] := max(num[0], num[1], ..., num[N-1]) + 1 ; // a1 is atomic
    for p := 0 to N-1 do { // p is local to (process executing) entry(i)
      a2: while num[p]≠0 and num[p]<num[i] do no-op ;
    }
  }
}
```

```

    // end entry code
    CS: critical section
    // exit code
    b1: num[i] := 0 ;
    // end exit code
  }
} // end code(i)

fairness-assumptions{ [∀ 0..N-1 i :: Wfair(πid[i])]
}

```

The desired correctness properties are the following safety and progress assertions:

- $\square [\forall 0..N-1 i, j, i \neq j :: Z(i, j)]$ , where  
 $Z(i, j) : \neg(u[i] \text{ on CS} \wedge u[j] \text{ on CS})$
- $[\forall 0..N-1 i :: P(i)] \Rightarrow [\forall 0..N-1 i :: Q(i)]$ , where  
 $P(i) : u[i] \text{ on CS} \rightsquigarrow u[i] \text{ at b1}$   
 $Q(i) : u[i] \text{ at a1} \rightsquigarrow u[i] \text{ at CS}$

### 9.3 Operational proof of simplified algorithm

We prove that the simplified bakery algorithm satisfies the desired assertions. Below, for a hungry process  $i$  and any process  $j$ , we say “ $i$  passes  $j$ ” to mean that process  $i$  executes `a2` (while loop test) with  $i.p = j$  and the condition false (i.e.,  $\text{num}[j] = 0$  or  $\text{num}[j] > \text{num}[i]$ ). We say “ $i$  passed  $j$ ” to mean that  $i$  passes  $j$  at some point after  $i$  last became hungry.

**Safety** Suppose process  $i$  starts eating at  $t_0$ . Consider any other process  $j$ ; we need to show that  $j$  is not eating at  $t_0$ . At some time  $t_1$  ( $< t_0$ ),  $i$  passes  $j$ . So  $\text{num}[j] = 0$  or  $\text{num}[j] > \text{num}[i]$  holds at  $t_1$ .

- Suppose  $\text{num}[j] = 0$  holds at  $t_1$ . Then process  $j$  is thinking at  $t_1$ , i.e.,  $j$  is in `NCS` or `a1`. If  $j$  becomes hungry at some time  $t_2$  before  $t_0$ , it sets  $\text{num}[j]$  to a value higher than  $\text{num}[i]$ . So  $j$  cannot pass  $i$  during the interval  $(t_2, t_1)$ . So  $j$  cannot be in `CS` at time  $t_0$ .
- Suppose  $\text{num}[j] \neq 0$  and  $\text{num}[j] > \text{num}[i]$  hold at  $t_1$ . Then it suffices to show that  $j$  has not yet passed  $i$ . Let process  $j$  have chosen its `num` at time  $t_2$ . Let process  $i$  have chosen its `num` at time  $t_3$ . Because  $\text{num}[j]$  is higher than  $\text{num}[i]$ ,  $t_2$  must be in the interval  $(t_3, t_1)$  (obvious?). Thus at  $t_2$ ,  $j$  has not passed  $i$ . Thus  $j$  has not passed  $i$  at  $t_0$ .

**Progress** Suppose process  $i$  becomes hungry at time  $t_1$ ; we need to show that it eventually eats. Because the `max` operation is atomic, no two processes with non-zero `nums` can have their `nums` equal. Thus all processes with non-zero `nums` are totally ordered by `num`. Furthermore, this “queue” of processes is first-in-first-out. A process joins the queue when its `num` becomes non-zero. But because this can only happen via the `max` operation, the process would join at the tail of the queue. The process at the head of the queue has the smallest non-zero `num`. So it never gets stuck in the while loop test for any process. So it eventually eats. Also, only the process at the head can leave because the only way to leave is to eat and only the head process can eat (from the safety property).

### 9.4 Homework 2, problem 2

You are asked to obtain an assertional proof of the simplified algorithm. Specifically,

- a. For the safety part, supply a collection of predicates whose conjunction, say  $X$ , is such that (1)  $X$  satisfies the invariance rule (i.e.,  $X$  holds initially and is preserved by every event), and (2)  $X$  implies  $[\forall i, j, i \neq j :: Z(i, j)]$ .

The following may be helpful:

- The notion “ $i$  passed  $j$ ” may be formalized by the following predicate:

$$\text{passed}(i, j) : (i \text{ on } a2 \wedge \pi id[i].p > j) \vee \pi id[i] \text{ on } CS, b1$$

- Assume that the index variable  $p$  in the for loop of the entry code is incremented atomically when the last statement of the loop is executed, i.e., just when  $a2$  is executed with the test failing. There is no loss of generality here because  $p$  is local to the process executing the loop.
- One would expect some like the following predicates to be invariant:

$$A_1 : \pi set = \{u[i] :: 0 \leq i < N\}$$

$$A_2(i, j) : \text{passed}(i, j) \Rightarrow$$

$$(a) \pi id[j] \text{ on } \{NCS, a1\} \vee$$

$$(b) (\pi id[j] \text{ on } a2 \wedge \neg \text{passed}(j, i) \wedge (\text{num}[j] > \text{num}[i]))$$

- b. For the progress part, come up with a lower-bounded function  $F(i)$  such that the following hold:

As long as process  $i$  is in  $a2$ , there is an weak-fair event whose execution decreases  $F(i)$  and no other event execution increases  $F(i)$ .