Lecturer: Nir Shavit

Handout 2

1 The Mutual Exclusion Problem

A *system* consists of n processes $\{1, \ldots, n\}$. Each *process* is a state machine, possibly with an infinite number of states. We consider an interleaving model of concurrency, where executions are modeled as sequences of *steps*. Each step is performed by a single process. A process i performs either a write step, $x := v$, or a read step, $v := x$, where v is a local variable and x is a shared register, or performs some local computation, and moves to a new state, from which one or more subsequent steps are enabled. *Process states* consist of the values of local variables and of the control (program counter). A *system state* consists of the states of the processes and registers. For convenience, we assume that each process can take at least one step from any system state.

We assume that each process can be described by a program that consists of two distinguished sections: a *non_critical_section* (sometimes called the *remainder_section*) and a *critical section*. Each process alternates between executing its non-critical and critical section as follows (Figure ??):

Process$_i$:
repeat forever
non_critical_section$_i$
critical_section$_i$
end repeat

Figure 1: Typical process

Our goal will be to implement algorithms that solve the mutual exclusion problem, by adding source code sections in each process immediately prior to and after it's *critical_section* (labeled *trying* and *exit* in Figure ??).

Formally we define a partial ordering '$\rightarrow$' on blocks of code of any two processes. Let $A_i^{[k]}$ denote the k-th running of the code section A in Process$_i$[1].

Definition Let $A_i^{[k]} \rightarrow B_j^{[k']}$ iff the first operation of the execution of $B_j^{[k']}$ is totally preceded by the last operation of the execution of $A_i^{[k]}$. Furthermore, let $A_i^{[k]} \not\rightarrow B_j^{[k']}$ denote $\neg(A_i^{[k]} \rightarrow B_j^{[k']})$.

A process P_i will typically enter and exit it's critical section a number of times during an execution of the above code section. The critical and non-critical section executions can

[1] At times, the superscripts will be dropped when context is clear.

be denoted by $CS_i^{[k]}$ and $NCS_i^{[k]}$ respectively, and we can write:

$$NCS_i^{[1]} \to CS_i^{[1]} \to NCS_i^{[2]} \to CS_i^{[2]} \to \ldots$$

A solution to the mutual exclusion problem will be designed to have the following properties (the first being essential):

Mutual Exclusion Property For every two processes P_i, P_j and any two integers k, k', $CS_i^{[k]} \to CS_j^{[k']}$ or $CS_j^{[k']} \to CS_i^{[k]}$.

Deadlock Freedom Property If process P_i never completes a *trying* section, then there is in infinite sequence of critical sections being executed by other processes.

Lockout Freedom Property Every *trying* section execution will eventually terminate (thus a process wishing to enter its critical section will do so within a finite time).

Note that lockout freedom implies deadlock freedom.

2 Solutions for Two Processes

We begin with two algorithms which do not meet all the desired properties.

2.1 Algorithm 1

Algorithm 1 is depicted in Figure **??**. [2]

Let $write_i(A := B)$ denote the execution of an operation of P_i in which variable A is assigned value B (similarly for $read_i(A)$. In addition, when a specific value B must be read within a given context, we shall write $read_i(A = B)$. For example, in Figure **??** $write_1(Q_1 := true)$ is line 1 of P_1, and $read_2(Q_1)$ is the part of line 2 of P_2 which reads the value of Q_1.

Lemma .1 *Algorithm 1 satisfies the mutual exclusion property.*

Proof: Assume by way of contradiction that the converse holds. Thus there exist integers k and k' s.t. $CS_1^{[k]} \not\to CS_2^{[k']}$ and $CS_2^{[k']} \not\to CS_1^{[k]}$. Consider the last execution of the trying section by each process prior to its k-th (respectively k'-th) execution of the critical section. From the code of the algorithm we have:

$$write_1(Q_1 := true) \to read_1(Q_2 = false) \to CS_1 \quad (1)$$
$$write_2(Q_2 := true) \to read_2(Q_1 = false) \to CS_2 \quad (2)$$
$$read_1(Q_2 = false) \to write_2(Q_2 := true) \quad (3)$$

Equation **??** holds since otherwise we could not have $Q_2 = false$.

[2]We omit the **repeat forever** loop for clarity.

$$write_1(Q_1 := true) \rightarrow \quad read_1(Q_2 = false) \rightarrow write_2(Q_2 := true) \rightarrow read_2(Q_1 = false) \tag{4}$$

From Equations ??, ?? and ?? by transitivity we have Equation ??. But this means that $write_1(Q_1 := true) \rightarrow read_2(Q_1 = false)$ without an intervening $write_1(Q_1 := false)$, a contradiction. ■ v Note that Algorithm 1 exhibits deadlock if the process executions are interleaved, since if both $write_1(Q1 := true)$ and $write_2(Q2 := true)$ are executed before $read_1(Q_2)$ and $read_2(Q_1)$, both processes will never complete their *trying* sections. On the other hand, if one of the processes is run before the other, no deadlock occurs.

2.2 Algorithm 2

Lemma .2 *Algorithm 2 satisfies the mutual exclusion property.*

Proof: Assume by way of contradiction that the converse holds. Thus there exist integers k and k' s.t. $CS_1^{[k]} \not\rightarrow CS_2^{[k']}$ and $CS_2^{[k']} \not\rightarrow CS_1^{[k]}$. Consider as before the last execution of the trying section by each process prior to its k-th (respectively k'-th) execution of the critical section. From the algorithm we have:

$$write_1(TURN := 1) \rightarrow read_1(TURN = 2) \rightarrow CS_1 \tag{5}$$

$$write_2(TURN := 2) \rightarrow read_2(TURN = 1) \rightarrow CS_2 \tag{6}$$

Since there must be a write of the value 2 after $write_1(TURN := 1)$ and before $read_1(TURN = 2)$ of ??, and since this is the last such write, we have

$$write_1(TURN := 1) \rightarrow write_2(TURN := 2) \rightarrow read_1(TURN = 2) \tag{7}$$

Since the value of 1 was overwritten and no additional write of 1 is performed, we have a contradiction to ??, since $read_2(TURN = 2)$. ■

Unlike in Algorithm 1, in Algorithm 2 deadlock occurs if one of the processes runs before the other. For instance, if P_1 is run first it will wait indefinitely for $write_2(TURN = 2)$. A concurrent execution will succeed.

2.3 Algorithm 3

We shall now combine the strategies of the above algorithms to construct one which is entirely deadlock-free.

We now show sketches of an operational proof of the algorithms correctness.

Lemma .3 *Algorithm 3 satisfies the mutual exclusion property.*

Proof: Again assume by way of contradiction that the converse holds. We shall confine ourselves to the last executions of the trying sections as before. From the code we have:

$$write_1(Q_1 := true) \rightarrow \tag{8}$$
$$write_1(TURN := 1) \rightarrow read_1(Q2) \rightarrow read_1(TURN) \rightarrow CS_1$$
$$write_2(Q_2 := true) \rightarrow \tag{9}$$
$$write_2(TURN := 2) \rightarrow read_2(Q1) \rightarrow read_2(TURN) \rightarrow CS_2$$

Since $TURN$ is a shared variable, if two concurrent writes are applied to it only one of the two written values will be readable. Thus we can assume without loss of generality that P_1 was the last process to write to $TURN$:

$$write_2(TURN := 2) \rightarrow write_1(TURN := 1) \tag{10}$$

The following Equation **??** is a subset of **??**: because P_1 has entered it's critical section we must have $Q_2 = false$ (from the **repeat until** line).

$$write_1(TURN := 1) \rightarrow read_1(Q_2 = false) \tag{11}$$

By transitivity, **??**, **??** and **??** we construct equation **??**.

$$write_2(Q_2 := true) \rightarrow \tag{12}$$
$$write_2(TURN := 2) \rightarrow write_1(TURN := 1) \rightarrow read_1(Q_2 = false)$$

Therefore we have $write_2(Q_2 := true) \rightarrow read_1(Q_2 = false)$, a contradiction because no other $write_i(Q_2)$ was performed before the critical section executions.

■

Lemma .4 *Algorithm 3 satisfies the lockout freedom property.*

Proof: Assume by way of contradiction that the claim does not hold. Without loss of generality P_1 never finishes it's *trying* section and is constantly executing the **repeat until** line reading $TURN = 1$ and $Q_2 = true$. Thus, P_2 must be in its trying section forever or repeatedly entering and exiting it, going through the critical section. It cannot repeatedly enter and exit its *trying* section since once such transition sets $TURN = 2$ forever. Thus, it must be in its trying section forever, i.e. executing the **repeat until** line. But since $Q_1 = true$, it must be reading $TURN = 2$, a contradiction. ■

Corollary .5 *Algorithm 3 satisfies the deadlock freedom property.*

3 Solutions for N Processes

Two generalizations of the two process algorithm into an n process algorithm are possible. Below we present one of them, the *filter algorithm*.

The idea behind the *filter algorithm* is that a process is forced to pass $n-1$ levels of "exclusion" in order to enter it's critical section. Each level j will have a separate $TURN[j]$

variable which will allow us to let all processes but one to proceed to the next level. Thus, loosing one process on every level, only one of a collection of concurrently trying processes will eventually enter the lowest level, i.e. the critical section. Note that the $Q1$ and $Q2$ variables in the two process code are replaced by a $Q[1..n]$ vector, each $Q[i]$ corresponding to process P_i.

Definition Process P_i is said to be at level 0 if $Q[i] = 0$, and at *level* j for $j > 0$, if it last completed the iteration of the **for** loop with $Q[i] \geq j$

Lemma .6 *For every $j = 0, \ldots, n-1$ there are at most $n - j$ processes at level j.*

Proof: By induction on j.

Induction Base: $j = 0$ is trivial.

Assume there are at most $n - j + 1$ processes at level $j - 1$. *Induction Step:* Assume by way of contradiction that there are at some point in the execution $n - j + 1$ or more at level j. Let us look at their last iteration of the loop prior to entering level j. By the induction hypothesis and given that any process at level j is also at level $j - 1$, there are exactly $n - j + 1$ processes at level j. Let P_i be the last process among them to write $TURN[j] := i$ while having $Q[i] = j$ (the write is performed while at level $j - 1$). For each P_k at level j where $k \neq i$:

$$write_k(TURN[j] := k) \rightarrow write_i(TURN[j] := i) \tag{13}$$

From the algorithm we have:

$$write_k(Q[k] := j) \rightarrow write_k(TURN[j] := k) \tag{14}$$

$$write_i(TURN[j] := i) \rightarrow read_i(Q[k]) \rightarrow read_i(TURN[j]) \tag{15}$$

From the code it follows that for any k, while P_k is at level $j - 1$, $Q[k]$ does not decrease. From Equations **??**, **??**, and **??** by transitivity it follows that for any P_k at level j,

$$write_k(Q[k] := j) \rightarrow read_i(Q[k]) \tag{16}$$

Since there are $n - j + 1$ of them (we include $k = i$ in our count), it must be that P_i read $TURN[j] \neq i$. However, P_i is the process to last write to $TURN[j]$, so $read_i(TURN[j] = i)$ in Equation **??**, a contradiction. ■

Being in the critical section is equivalent to being at level $n - 1$ and so we have:

Corollary .7 *Algorithm 4 satisfies the mutual exclusion property.*

Lemma .8 *If any process P_i is at level j and never gets into level $j + 1$, then some other processes will infinitely often enter end exit level $j + 1$.*

Proof: We'll use inverted induction on j, *i.e* induction on the the levels $n - 1 - j$, where the level for $j = 0$ is $n - 1$, the critical section.

Induction base: As the base of our induction, assume that a process is stuck at level $n - 2$, that is, it repeatedly fails the test loop. If no other process is forever in or forever entering level $n - 2$, then P_i will eventually read $(\forall k \neq i, Q[k] < n - 2)$ and therefore will eventually enter level $n-1$. If some other process P_k is forever in level $n-2$ (there is no lower level except the critical section), then since P_i always reads $TURN[n-2]=i$, it must be that P_k will eventually read $TURN[n-2] \neq k$ and enters level $n-1$. It follows that there must be some process that is forever entering and exiting the critical section.

Assume that the claim holds for level $n-j-1$. Let us prove it for a process at level $l=n-j-2$.

Induction step: If there's another process stuck at a higher level $l' \geq l+1$, then by the induction hypothesis some third process will be infinitely entering and exiting l' and therefore will infinitely enter l. If no other process is (1) forever in or (2) forever entering level, l then P_i will eventually read $\forall k \neq iQ[k] < l$ and, therefore, get into level $l+1$. If some other process P_k is in level l, it cannot be forever at level l since P_i always reads $TURN[l]=i$, P_k must eventually read $TURN[l] \neq k$, and, therefore, P_k will enter level $l+1$. Thus it must be that case (2) holds, and some process is infinitely often entering level l. ■

Corollary .9 *Algorithm 4 is deadlock free.*

Proof: Assume that some process P_i never completes its trying section. That means that P_i is stuck at some level l. According to Lemma **??** some other processes will infinitely enter level $l+1$. In order to pass twice through level j, a process must pass through level $n-1$ and therefore some other processes must infinitely often enter the critical section. ■

Lemma .10 *Algorithm 4 is lockout free.*

Proof: If any process P_i is stuck at level j, then by Lemma **??** some other processes P_k will infinitely often enter level $j+1$. After the second time P_k enters level j, P_i will read $TURN \neq i$ and enter level $j+1$. Therefore P_i will eventually enter the critical section. ■

4 Fairness

The lockout freedom property of the Mutual Exclusion algorithms we have seen, guarantees that each one of the trying processes will enter its critical section. However, it does not impose an order determining which will enter its the critical section first. The ideal form of fairness would be that if a process P_i starts its trying section before process P_j does, it will enter the critical section before it.

$$start(trying_i^{[k]}) \rightarrow start(trying_j^{[k']}) \Rightarrow \mathrm{CS}_i^{[k]} \rightarrow \mathrm{CS}_j^{[k']}$$

However this kind of fairness seems impossible to achieve even given atomicity of operations, since any given process cannot determine the order among trying section executions. A process can in any one step only read or write. Hence, a process cannot in the same step mark the start its trying section (requires writing) and check if other process have already started their trying sections (requires reading).

The compromise is thus to split the trying section into two parts:

1. *doorway* section which is *wait-free*, that is, its execution consists of a bounded number of steps.

2. *waiting* section which may take an unbounded number of steps.

We can now define a measure of fairness:

Definition *r-bounded waiting property*: For any two processes P_i, P_j and for any k, k':

If $doorway_i^{[k]} \rightarrow doorway_j^{[k']}$ then $\mathrm{CS}_i^{[k]} \rightarrow \mathrm{CS}_j^{[k'+r]}$

That is if process p_i finished it's doorway before another process p_k, process p_k will enter it's critical section at most r times before p_i. The strong form of fairness known as *first-come-first-serve* is thus equivalent to *0-bounded waiting.*

5 The Bakery Algorithm

The algorithm in Figure **??** satisfies *first-come-first-serve.* It is a simplification of the famous *bakery algorithm* of Lamport.

x_i - indicates whether p_i wants to enter it's critical section. l_i - label variable. Determines p_i's order among trying processes. The order among labels is a lexicographical order on pairs (l_i, i) of integers. Note that l_i grows indefinitely. It is easy to check that as the execution progresses for each process the label l_i written is monotonically increasing, and for all processes the labels written are monotonically nondecreasing (it is possible that at a given moment some processes have the same label).

Lemma .11 *Algorithm 5 is deadlock free.*

Proof: Algorithm 5 is deadlock free, since for all waiting processes there's always one process with a smaller (l_i, i) pair than the others. The lexicographical order is unique, because each process has a unique number i. ■

Lemma .12 *Algorithm 5 satisfies the first-come-first-served property.*

Proof: (brief sketch) It satisfies *first-come-first-serve*, since for any two processes P_i and P_j if $labeling_i \rightarrow labeling_j$ it is the case that

$$write_i(l_i) \rightarrow read_j(l_i) \rightarrow write_j(l_j := l_i + 1)$$

and thus $l_i < l_j$. Thus, if a process P_i finished it's *doorway* before another, then its label is smaller. The process with higher label will fail as long as $x_i = true$ since labels only grow. ■

Note that deadlock freedom and *first-come-first-serve* implies lockout freedom.

Lemma .13 *Algorithm 5 satisfies mutual exclusion.*

Proof:(brief sketch) Assume by way of contradiction that the converse holds. Choose two processes P_i and P_j that are concurrently in the critical section. Let P_j be such that: $(l_j, j) > (l_i, i)$. When P_j last executed the test before executing the critical section, it must have read $x_i = false$ or $(l_j, j) < (l_i, i)$. However, since labels of a given process P_i are monotonically increasing, it must be that it read $x_i = false$. Thus,

$$labeling_j \rightarrow read_j(x_i) \rightarrow write_i(x_i := true) \rightarrow labeling_i$$

Since we assumed that $(l_j, j) > (l_i, i)$ that is a contradiction to Lemma **??**. ■

References

[1] L.Lamport, “The Mutual Exclusion Problem”, Part I and II, JACM 33, No. 2, April 1986, pp. 313-370.

[2] G.L.Peterson, “Myths about the mutual exclusion problem”, IPL 12, No. 3, pp. 115-116, Jun. 1981.

[3] E.W.Dijkstra, “Solution of a problem in concurrent programming control”, CACM 8,9, pp.569, Sep. 1965.

[4] L.Lamport, “A new solution of Dijkstra concurrent programming problem”, CACM 17, No. 8, pp.453-455, Aug. 1974.

Process$_i$:

repeat forever

$\quad\quad$ *non_critical_section*$_i$

$\quad\quad$ *trying*$_i$

$\quad\quad$ *critical_section*$_i$

$\quad\quad$ *exit*$_i$

end repeat

Figure 2: Typical solution process

P_1	P_2
Initially $Q_1 = \mathit{false}$	Initially $Q_2 = \mathit{false}$
P_1:	P_2:
$\quad Q_1 := \mathit{true}$;	$\quad Q_2 := \mathit{true}$;
$\quad$ **repeat until** $\neg Q_2$;	$\quad$ **repeat until** $\neg Q_1$;
$\quad CS_1$	$\quad CS_2$
$\quad Q_1 := \mathit{false}$;	$\quad Q_2 := \mathit{false}$;

Figure 3: Algorithm 1

P_1	P_2
P_1:	P_2:
$\quad TURN := \mathit{1}$;	$\quad TURN := \mathit{2}$;
$\quad$ **repeat until** $TURN = 2$;	$\quad$ **repeat until** $TURN = 1$;
$\quad CS_1$	$\quad CS_2$

Figure 4: Algorithm 2

P_1	P_2
Initially $Q_1 = \mathit{false}$	Initially $Q_2 = \mathit{false}$
P_1:	P_2:
$\quad Q_1 := \mathit{true}$;	$\quad Q_2 := \mathit{true}$;
$\quad TURN := \mathit{1}$;	$\quad TURN := \mathit{2}$;
$\quad$ **repeat until** $\neg Q_2$ **or** $TURN = \mathit{2}$;	$\quad$ **repeat until** $\neg Q_1$ **or** $TURN = \mathit{1}$;
$\quad CS_1$	$\quad CS_2$
$\quad Q_1 := \mathit{false}$;	$\quad Q_2 := \mathit{false}$;

Figure 5: Algorithm 3

```
Initially Q[1..n] = 0
P_i:
   for j := 1 to n − 1 do
       Q[i] := j;
       TURN[j] := i;
       repeat until (∀k ≠ i Q[k] < j) or TURN[j] ≠ i;
   od
   CS_i
   Q_i := 0;
```

Figure 6: Algorithm 4

```
P_i :
   x_i := true; (* start doorway*)
   l_i := max(l_1, ..., l_n) + 1; (*end doorway*)
   repeat until (∀k)(¬x_k or (l_i, i) < (l_k, k));
   CS_i
   x_i := false;
```

Figure 7: Bakery Algorithm