Deriving a Scalable Algorithm for Mutual Exclusion*

Yih-Kuen Tsay

Department of Information Management
National Taiwan University
Taipei 106, TAIWAN
tsay@im.ntu.edu.tw

Abstract. This paper details the design of a scalable algorithm for the mutual exclusion problem. Starting by inserting a redundant assignment into Peterson's algorithm for two processes, we derive another algorithm that uses only local spins, i.e., a process busy-waits only on locally accessible shared variables. The new two-process algorithm is then adapted to serve as a building block of the complete tournament-like algorithm; the adaptation is such that the entire algorithm still uses only local spins, which is crucial for scalability. We consider the simplicity of the algorithm and its derivation to be the main contributions of this paper.

Keywords: local spins, mutual exclusion, read/write atomicity, refinement, scalability.

1 Introduction

New solutions seem never to stop emerging for the mutual exclusion problem, despite its relatively old age.[1] A new solution is proposed either to meet newly conceived criteria or to improve over previous ones with respect to certain tangible or less tangible measures. Simplicity is one of the less tangible measures. This paper presents a solution to the mutual exclusion problem which we believe to be simpler than previous ones that meet the same criteria.

To recapitulate the problem, each of a set of N processes from time to time may wish to execute a special segment of code called its critical section; the execution time of the critical section is assumed to be finite. Two segments of code—the *trying* protocol and the *exit* protocol are needed respectively before and after the critical section of each process such that the following two requirements are satisfied: (a) mutual exclusion—no two processes are in their critical sections at the same time and (b) fairness (starvation-freedom)—a process that is trying to enter the critical section, i.e., executing the trying protocol, will eventually enter the critical section.

* This work was supported in part by research awards from the National Science Council, Taiwan (R.O.C.) and College of Management, National Taiwan University.

[1] The mutual exclusion problem has come into existence since 1962, according to Dijkstra [3].

In [10], Yang and Anderson proposed the first (and, to our knowledge, the only previous) scalable solution that assumes only read/write atomicity. Their solution is scalable mainly due to the facts that

1. a process uses only local spins, i.e., it busy-waits only on locally accessible shared variables, and
2. the number of remote read/write operations for each access to the critical section is $O(\log N)$.

It is assumed that a shared variable can be made locally accessible to at most one process. Also, busy-waiting on a remote shared variable implies an unbounded number of remote read operations.

Scalable solutions that assume stronger atomic operations such as fetch&add and compare&swap had been proposed earlier [1, 7]. However, such solutions require special hardware support and may fail to meet the criteria of fairness and portability (across different hardware architectures). Moreover, despite its use of less powerful read/write operations, Yang and Anderson's solution is nearly as efficient as these earlier solutions.

In this paper, we propose a new scalable solution that also assumes only read/write atomicity. The proposed solution is scalable due to the same reasons as described previously for Yang and Anderson's. One distinct feature of our work is that we derive our algorithm through a sequence of refinement steps (rather than give the final solution and then prove its correctness). We justify the refinement steps informally, though a more formal treatment is possible.

The overall structure of our algorithm is based on the idea of a tournament. All tournament-like algorithms, including [9, 4, 6, 10], require a suitable mutual exclusion algorithm for two processes. Near two decades ago, Peterson [8] proposed an elegant solution to the two-process mutual exclusion problem. He argued that his algorithm does not appear out of nowhere, but is derived from combining two simpler algorithms, each solving part of the problem. Peterson's idea inspired us to derive a new two-process algorithm that uses only local spins and has essentially the same simple structure as his algorithm does. We adapt the new two-process algorithm to serve as a building block of our tournament-like algorithm; the adaptation is such that the entire algorithm still uses only local spins.

Another criterion closely related to scalability is adaptiveness. Scalability emphasizes the performance of an algorithm when the number of contending processes is large, while adaptiveness emphasizes the performance when the contention is low. Lamport [5] proposed an algorithm in which only a constant number of operations is required for entering the critical section in the absence of contention. His algorithm is followed by a few more algorithms, in particular Choy and Singh [2], that exhibit better adaptiveness. An existing idea for combining an adaptive algorithm with our scalable algorithm will be discussed.

2 The Two-Process Algorithm

In this section, we derive the basic algorithm for two processes that uses only local spins. The derivation starts by inserting a redundant assignment into the exit protocols of Peterson's two-process algorithm. The redundancy is then exploited to eliminate busy-waitings on remote shared variables.

2.1 Peterson's Algorithm

The celebrated Peterson's two-process algorithm is recreated below.

```
/* shared variables and initial conditions */
shared var Q: array [0..1] of boolean;
           TURN: 0..1;
initially Q[0..1] = false;
```

```
/* process P0 */                    /* process P1 */
while true do                       while true do
    remainder;                          remainder;
    Q[0] := true;                       Q[1] := true;
    TURN := 0;                          TURN := 1;
    await ¬Q[1] ∨ TURN ≠ 0;             await ¬Q[0] ∨ TURN ≠ 1;
    critical section;                   critical section;
    Q[0] := false                       Q[1] := false
od                                  od
```

Peterson pointed out in his paper that the two-process algorithm is derived by combining two simple (but incorrect) algorithms, which we refer to as Primitives A and B^-, respectively.

Primitive A:

```
...                     ...
Q[0] := true;           Q[1] := true;
await ¬Q[1];            await ¬Q[0];
critical section;       critical section;
Q[0] := false;          Q[1] := false;
...                     ...
```

Primitive B^-:

```
...                     ...
TURN := 0;              TURN := 1;
await TURN ≠ 0;         await TURN ≠ 1;
critical section;       critical section;
...                     ...
```

Both primitives satisfy the mutual exclusion requirement, but both may deadlock. However, the deadlocks are caused by very different reasons. In Primitive A, deadlock occurs when both processes are trying (and execute the assignments in their trying protocols at about the same time); while, in Primitive B$^-$, deadlock occurs when only one process is trying. A "disjunctive" combination of the two primitives avoids the deadlocks, while preserving mutual exclusion.

2.2 Peterson's Algorithm with Redundancy

There are no exit protocols in Primitive B$^-$, as they would become redundant in Peterson's algorithm. In Primitive B$^-$, one of the two processes will be blocked in the **await** statement even after the other has left its critical section. This suggests that the left-out exit protocols are "$TURN := 0$" for P_0 and "$TURN := 1$" for P_1. Let us put them back to recover the "original" primitive, referred to as Primitive B.

Primitive B:

...	...
$TURN := 0$;	$TURN := 1$;
await $TURN \neq 0$;	**await** $TURN \neq 1$;
critical section;	critical section;
$TURN := 0$;	$TURN := 1$;
...	...

Combining Primitives A and B in the same way as Peterson did, we obtain a variant of Peterson's algorithm with redundancy in the exit protocols, shown below.

...	...
$Q[0] := true$;	$Q[1] := true$;
$TURN := 0$;	$TURN := 1$;
await $\neg Q[1] \vee TURN \neq 0$;	**await** $\neg Q[0] \vee TURN \neq 1$;
critical section;	critical section;
$Q[0] := false$;	$Q[1] := false$;
$TURN := 0$;	$TURN := 1$;
...	...

2.3 First Refinement: Exploiting the Redundancy

The redundancy in the exit protocols provides opportunities for optimizing the trying protocols (with respect to the number of remote operations). Thanks to the assignments to $TURN$ in the exit protocols, the testings of $\neg Q[0]$ and $\neg Q[1]$ in the **await** statements need to be done only once using conditional statements and can be separated from the repeated testings of $TURN \neq 0$ and $TURN \neq 1$. The resulting algorithm is shown in Figure 1.

```
...                                   ...
Q[0] := true;                         Q[1] := true;
TURN := 0;                            TURN := 1;
if Q[1] then                          if Q[0] then
   await TURN ≠ 0                        await TURN ≠ 1
fi;                                   fi;
critical section;                     critical section;
Q[0] := false;                        Q[1] := false;
TURN := 0;                            TURN := 1;
...                                   ...
```

Fig. 1. A two-process algorithm obtained from the first refinement

Remark: In the algorithm above, if P_0 finds $Q[1]$ to be *false* after setting $Q[0]$ to *true*, then it could immediately enter the critical section without setting $TURN$ to 0 and subsequently executing the conditional statement; also, P_0 could set $TURN$ to 0 in the exit protocol only if it finds Q_1 to be *true*. An analogous observation applies to P_1. The algorithm could be further refined as below to make clearer the use of $TURN$. However, to prevent the additional testings of Q from distracting our focus on the elimination of remote busy-waitings, we postpone this refinement till a later stage.

```
...                                   ...
Q[0] := true;                         Q[1] := true;
if Q[1] then                          if Q[0] then
   TURN := 0;                            TURN := 1;
   if Q[1] then                          if Q[0] then
      await TURN ≠ 0                        await TURN ≠ 1
   fi                                    fi
fi;                                   fi;
critical section;                     critical section;
Q[0] := false;                        Q[1] := false;
if Q[1] then                          if Q[0] then
   TURN := 0                             TURN := 1
fi;                                   fi;
...                                   ...
```

2.4 Second and Third Refinements: Localizing Busy-Waitings

We still have one remote busy-waiting in each process after the first refinement. Since the same variable $TURN$ is involved, the only way to go is representing $TURN$ by two separate variables. As the second refinement step, we introduce two new boolean variables $T[0]$ and $T[1]$ to replace $TURN$ and stipulate that $T[0] = T[1]$ if and only if $TURN = 0$. The assignment $TURN := 0$ in P_0 becomes a sequence of two assignments $t := T[1]$ and $T[0] := t$ (t is a private variable), via which P_0 attempts to make $T[0]$ and $T[1]$ equal; while, the statement **await**

$TURN \neq 0$ becomes **await** $T[1] \neq t$, where t can be shown to equal $T[0]$ in the statement. An analogous change is made for P_1 so that P_1 attempts to make $T[0]$ and $T[1]$ unequal and t can be shown to equal $\overline{T[1]}$ (the binary complement of $T[1]$) in the **await** statement. The algorithm becomes as follows; $\bar{t}$ denotes the binary complement of t.

/* t is a private variable */	/* t is a private variable */
...	...
$Q[0] := true$;	$Q[1] := true$;
$t := T[1]$;	$t := T[0]$;
$T[0] := t$;	$T[1] := \bar{t}$;
if $Q[1]$ **then**	**if** $Q[0]$ **then**
await $T[1] \neq t$	**await** $T[0] \neq t$
fi;	**fi**;
critical section;	critical section;
$Q[0] := false$;	$Q[1] := false$;
$t := T[1]$;	$t := T[0]$;
$T[0] := t$;	$T[1] := \bar{t}$;
...	...

If we would allocate $T[0]$ and $T[1]$ in such a way that $T[0]$ is locally accessible to P_1 and $T[1]$ is locally accessible to P_0, then we would have successfully eliminated all remote busy-waitings. However, to make an indexed variable more suggestive of its location, we swap $T[0]$ and $T[1]$ in the third refinement step. The resulting algorithm is shown in Figure 2.

Lemma 1. *The following are true for the algorithm in Figure 2: for P_0, t equals $T[1]$ when (the control of) P_0 is at Statement 5; and, for P_1, t equals $\overline{T[0]}$ when P_1 is at Statement 5.*

Proof. For P_0, since only P_0 may change the value of $T[1]$, Statement 3 makes $T[1]$ equal to t, and Statement 4 does not change t or $T[1]$, it follows that t equals $T[1]$ when P_0 is at Statement 5. An analogous argument applies to P_1. □

Theorem 1. *The algorithm in Figure 2 solves the mutual exclusion problem of two processes. And, the number of remote operations for each access to the critical section is $O(1)$.*

Proof. We prove only one interesting and nontrivial case that, when both processes find $Q[0]$ and $Q[1]$ to be *true*, it is impossible for the processes to pass the **await** statements and end up in the critical section simultaneously. Assume without loss of generality that P_0 passes the **await** statement first. From the preceding lemma, $T[0] \neq T[1]$ when P_0 passes the **await** statement. We need to show that P_1 cannot subsequently find $T[1] \neq t$, i.e., $T[0] = T[1]$ according to the preceding lemma, and pass the **await** statement before P_0 leaves the critical section. We observe that P_1 must have changed the value of $T[0]$ after P_0 reads $T[0]$ for the last time in Statement 2 and before P_0 enters the critical

/* shared variables and initial conditions */
shared var Q, T: **array** $[0..1]$ **of boolean**;
initially $Q[0..1] = \mathit{false}$;

/* process P_0 */	/* process P_1 */
private var t: **boolean**;	**private var** t: **boolean**;
while *true* **do**	**while** *true* **do**
0: remainder;	0: remainder;
1: $Q[0] := \mathit{true}$;	1: $Q[1] := \mathit{true}$;
2: $t := T[0]$;	2: $t := T[1]$;
3: $T[1] := t$;	3: $T[0] := \bar{t}$;
4: **if** $Q[1]$ **then**	4: **if** $Q[0]$ **then**
5: **await** $T[0] \neq t$	5: **await** $T[1] \neq t$
fi;	**fi**;
6: critical section;	6: critical section;
7: $Q[0] := \mathit{false}$;	7: $Q[1] := \mathit{false}$;
8: $t := T[0]$;	8: $t := T[1]$;
9: $T[1] := t$	9: $T[0] := \bar{t}$
od	**od**

Fig. 2. A two-process algorithm using only local spins

section; otherwise, P_0 cannot possibly find $T[0] \neq T[1]$ after the execution of Statements 2 and 3, which makes $T[0]$ and $T[1]$ equal. Consider the following three representative scenarios (the private variable t is indexed by the identity of its owner process to avoid confusion) :

1: $Q[0] := \mathit{true}$;	
	...
2: $t_0 := T[0]$;	
	...
	1: $Q[1] := \mathit{true}$;
	2: $t_1 := T[1]$;
	3: $T[0] := \overline{t_1}$;
3: $T[1] := t_0$;	
4,5: ...	

In this scenario, P_1 will find $T[0] \neq T[1]$ as P_0 did and be blocked, since the values of $T[0]$ and $T[1]$ have not been changed after P_0 last found $T[0] \neq T[1]$ and passed the **await** statement.

1: $Q[0] := true$;	
	...
2: $t_0 := T[0]$;	
	...
	8: $t_1 := T[1]$;
	9: $T[0] := \overline{t_1}$;
	...
	1: $Q[1] := true$;
	2: $t_1 := T[1]$;
3: $T[1] := t_0$;	
4,5: ...	
	3: $T[0] := \overline{t_1}$;

We observe that, in the above scenario, the execution of Statement 2 by P_1 does not change the value of t_1 and the execution of Statement 3 by P_1 does not change the value of $T[0]$. Like in the first scenario, P_1 will find $T[0] \neq T[1]$ as P_0 did and be blocked, since the values of $T[0]$ and $T[1]$ have not been changed after P_0 last found $T[0] \neq T[1]$ and passed the **await** statement.

1: $Q[0] := true$;	
	...
2: $t_0 := T[0]$;	
	...
	8: $t_1 := T[1]$;
	9: $T[0] := \overline{t_1}$;
	...
	1: $Q[1] := true$;
3: $T[1] := t_0$;	
4,5: ...	
	2: $t_1 := T[1]$;
	3: $T[0] := \overline{t_1}$;

In this scenario, the execution of Statements 2 and 3 by P_1 makes $T[0]$ and $T[1]$ unequal. P_1 will be blocked until later P_0 changes the value of $T[1]$ in the exit protocol. □

2.5 Fourth Refinement: Using "*TURN*" Only If Necessary

Having eliminated all remote busy-waitings, we now take the refinement step indicated earlier in the end of Section 2.3. Though *TURN* has been refined into a pair of boolean variables, the refinement step still applies. This completes our derivation of the two-process algorithm; its final design is shown in Figure 3.

Theorem 2. *The algorithm in Figure 3 solves the mutual exclusion problem of two processes. And, the number of remote operations for each access to the critical section is* $O(1)$.

```
/* shared variables and initial conditions */
shared var Q, T: array [0..1] of boolean;
initially Q[0..1] = false;
```

```
/* process P0 */
private var t: boolean;
while true do
   remainder;
   Q[0] := true;
   if Q[1] then
      t := T[0];
      T[1] := t;
      if Q[1] then
         await T[0] ≠ t
      fi
   fi;
   critical section;
   Q[0] := false;
   if Q[1] then
      t := T[0];
      T[1] := t
   fi
od
```

```
/* process P1 */
private var t: boolean;
while true do
   remainder;
   Q[1] := true;
   if Q[0] then
      t := T[1];
      T[0] := t̄;
      if Q[0] then
         await T[1] ≠ t
      fi
   fi;
   critical section;
   Q[1] := false;
   if Q[0] then
      t := T[1];
      T[0] := t̄
   fi
od
```

Fig. 3. Final design of the two-process algorithm

3 Adapting the Two-Process Algorithm

The two-process algorithm derived in the previous section provides a mechanism of arbitration between two *predetermined* processes. However, in a tournament-like algorithm for N processes, an arbitration may need to be made between two processes whose identities can be determined only during the execution time. To take into account this dynamic aspect, we consider the following variant of the mutual exclusion problem.

Definition 1. The dynamic two-process mutual exclusion problem*: The set of processes are divided into two groups, represented respectively by two disjoint sets ID_0 and ID_1 of non-negative integers. The terms trying protocol, critical section, and exit protocol as well as the two correctness requirements are as defined before. However, we assume that the processes are* well-behaved *in the sense that at most one process from each group can be* active *at any time; a process is said to be active if its control is between the beginning of the trying protocol and the end of the exit protocol.*

We shall adapt the two-process algorithm to solve this problem. The main difficulty is that, since which processes will become active cannot be predetermined, a fixed allocation of $T[0]$ and $T[1]$ will ruin the virtue of "local spins only". The idea is to let the processes supply their own copies of $T[0]$ or $T[1]$. But, this causes a new problem for a process to identify its opponent, i.e., to tell which copy of $T[0]$ or $T[1]$ (depending on its role) it should access. To resolve the problem, we change the type of $Q[0]$ and $Q[1]$ from boolean to the set $\{-1\} \cup ID_0 \cup ID_1$ so that they can be used not only as flags but also for recording the identities of processes. To try to enter its critical section, a process sets the appropriate entry of Q to its identity, which is reset to -1 by the process after leaving the critical section. Testing the truth value of $Q[0]$ or $Q[1]$ in the original algorithm is replaced by testing that of $opp \neq -1$, where opp records the value of $Q[0]$ or $Q[1]$.

Moreover, it is possible for a process in the trying protocol to see a different opponent when it checks the value of $Q[0]$ or $Q[1]$ for the second time. This occurs, for example, when a process p_i, $i \in ID_0$, first reads and checks $Q[1]$ to find that p_j, $j \in ID_1$, is its opponent. Subsequently p_j leaves the critical section and finishes executing the exit protocol. Another process $p_{j'}$, $j' \in ID_1$, may then start to execute the trying protocol, setting $Q[1]$ to j'. At this moment, p_i will find that $Q[1]$ has been changed. We claim that p_i may immediately enter the critical section. This is the case because $p_{j'}$ sees p_i as its opponent and will make $T[i]$ and $T[j']$ unequal and be blocked at the **await** statement until later p_i sees $p_{j'}$ as its opponent in the exit protocol and make $T[i]$ and $T[j']$ equal.

Figure 4 shows the adapted two-process algorithm for process p_i to compete as P_0 and process p_j to compete as P_1.

Theorem 3. *The algorithm in Figure 4 solves the dynamic two-process mutual exclusion problem. And, the number of remote operations for each access to the critical section is $O(1)$.*

4 The N-Process Algorithm

In this section, we utilize the adapted two-process algorithm as a building block to obtain a tournament-like algorithm for N processes. Without loss of generality, we assume that N (≥ 2) is a power of 2 (dummy processes may be added if N is not so) and the processes will be referred to as p_0, p_1, ..., and p_{N-1}.

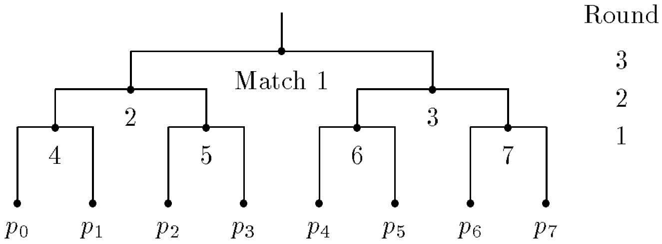
Consider a tournament tree of N leaves which represent from left to right the N processes p_0, p_1, ..., and p_{N-1}. To enter the critical section, a process (leaf) has to "win" (in the ascending order) the $\log N$ rounds of matches represented by its ancestors. There are totally $N-1$ matches in the tournament tree, which we refer to as Matches 1 through $N-1$ going from top to bottom and left to right. Below is a tournament tree for 8 processes:

```
/* shared variables and initial conditions */
shared var Q: array [0..1] of {−1} ∪ ID0 ∪ ID1;
           T: array [ID0 ∪ ID1] of boolean;
initially Q[0..1] = −1;
```

```
/* p_i, i ∈ ID0, competing as P0 */
private var opp: {−1} ∪ ID0 ∪ ID1;
            opp': {−1} ∪ ID0 ∪ ID1;
            t: boolean;
while true do
   remainder;
   Q[0] := i;
   opp := Q[1];
   if opp ≠ −1 then
      t := T[i];
      T[opp] := t;
      opp' := Q[1];
      if opp' = opp then
         await T[i] ≠ t
      fi
   fi;
   critical section;
   Q[0] := −1;
   opp := Q[1];
   if opp ≠ −1 then
      t := T[i];
      T[opp] := t
   fi
od
```

```
/* p_j, j ∈ ID1, competing as P1 */
private var opp: {−1} ∪ ID0 ∪ ID1;
            opp': {−1} ∪ ID0 ∪ ID1;
            t: boolean;
while true do
   remainder;
   Q[1] := j;
   opp := Q[0];
   if opp ≠ −1 then
      t := T[j];
      T[opp] := t̄;
      opp' := Q[0];
      if opp' = opp then
         await T[j] ≠ t
      fi
   fi;
   critical section;
   Q[1] := −1;
   opp := Q[0];
   if opp ≠ −1 then
      t := T[j];
      T[opp] := t̄
   fi
od
```

Fig. 4. The adapted two-process algorithm for the dynamic two-process mutual exclusion problem



Note that the matches that a process has to win are predetermined, but its opponents in the matches, except in the first round, are not. To enter the critical section, for instance, p_4 has to win Matches 6, 3, and 1 (in this order). The opponent of p_4 in Match 3 may either be p_6 or p_7.

Remark: The tournament here is very different from a real tournament; the analogy is only for an informal explanation of how the algorithm works. In a match here, we allow a loser to become a new winner when the original winner returns at a later time (after executing the critical section) and "turns" the loser into a new winner. A player may also win the same match many times in a row (or many instances of the same match) if its potential opponent never shows up.

4.1 Trying and Exit Protocols for a Match

Our plan is to use an instance of the adapted two-process algorithm for determining the winner of a particular match. We stipulate that the process from the left branch of a match plays the role of P_0, while the process from the right plays P_1. For instance, p_4 should play P_0 and p_5 should play P_1 in match 6, whose winner then should play P_0 in Match 3. Note that the role of a process in each round of match can be predetermined. Below is the adapted two-process algorithm instantiated for process p_i to play Match m in Round r as P_0 and process p_j to play as P_1:

/* shared variables involved and their initial conditions */
shared var $Q[m,0], Q[m,1]$: $-1..(N-1)$;
            $T[r,i], T[r,j]$: **boolean**;
initially $Q[m,0..1] = -1$;

...	...
/* entering Round r */	/* entering Round r */
$Q[m,0] := i$;	$Q[m,1] := j$;
$opp := Q[m,1]$;	$opp := Q[m,0]$;
if $opp \neq -1$ **then**	**if** $opp \neq -1$ **then**
$t := T[r,i]$;	$t := T[r,j]$;
$T[r,opp] := t$;	$T[r,opp] := \bar{t}$;
$opp' := Q[m,1]$;	$opp' := Q[m,0]$;
if $opp' = opp$ **then**	**if** $opp' = opp$ **then**
await $T[r,i] \neq t$	**await** $T[r,j] \neq t$
fi	**fi**
fi;	**fi**;
...	...
/* next round or critical section */	/* next round or critical section */
...	...
$Q[m,0] := -1$;	$Q[m,1] := -1$;
$opp := Q[m,1]$;	$opp := Q[m,0]$;
if $opp \neq -1$ **then**	**if** $opp \neq -1$ **then**
$t := T[r,i]$;	$t := T[r,j]$;
$T[r,opp] := t$	$T[r,opp] := \bar{t}$
fi;	**fi**;
/* leaving Round r */	/* leaving Round r */
...	...

Parameterizing i (the identity of a process), m, r, and x (the role of a process), we obtain the final trying and exit protocols for a match, shown below.

binary_trying(i, m, r, x):
 $Q[m, x] := i$;
 $opp := Q[m, \overline{x}]$;
 if $opp \neq -1$ **then**
  $t := T[r, i]$;
  $T[r, opp] := \overline{x} \cdot t + x \cdot \overline{t}$;
  $opp' := Q[m, \overline{x}]$;
  if $opp' = opp$ **then**
   await $T[r, i] \neq t$
  fi
 fi;

binary_exit(i, m, r, x):
 $Q[m, x] := -1$;
 $opp := Q[m, \overline{x}]$;
 if $opp \neq -1$ **then**
  $t := T[r, i]$;
  $T[r, opp] := \overline{x} \cdot t + x \cdot \overline{t}$
 fi;

4.2 The Complete Algorithm

The derivation is near completion. What remains to be done is appropriate allocation of the shared variables.

Each match m needs a copy of $Q[m, 0..1]$; there are $N - 1$ matches in total. Each process p_i needs to supply a copy of $T[r, i]$ for each round r; there are $\log N$ rounds. The array T should be allocated in such a way that the entries $[1..\log N, i]$ are locally accessible to p_i. How Q should be allocated is not critical. The complete algorithm is shown in Figure 5. In the algorithm, $match(i, r)$ gives the number of the match that p_i should play in Round r, while $role(i, r)$ tells its role. The value of $match(i, r)$ can be computed by $2^{\log N - r} + \lfloor \frac{i}{2^r} \rfloor$ and the value of $role(i, r)$ can be computed by $\lfloor \frac{i}{2^{r-1}} \rfloor \bmod 2$

Theorem 4. *The algorithm in Figure 5 solves the mutual exclusion problem of N processes. And, the number of remote operations for each access to the critical section is $O(\log N)$.*

Proof. We associate with each match an instance of the dynamic two-process mutual exclusion problem (Definition 1) as follows. The processes that may become the players of the match constitute the set of processes in the instance. Those coming from the left branch (according to the tournament tree) of the match form the group ID_0, while those from the right branch form the group ID_1; the two groups are apparently disjoint. The critical section of a process in

```
/* shared variables and initial conditions */
shared var Q: array [1..(N − 1), 0..1] of −1..(N − 1);
           T: array [1..log N, 0..(N − 1)] of boolean;
initially Q[1..(N − 1), 0..1] = −1;

/* process p_i, 0 ≤ i ≤ (N − 1) */
private var opp, opp': −1..(N − 1);
            t: boolean;
while true do
    remainder;
    binary_trying(i, match(i, 1), r, role(i, 1));
    binary_trying(i, match(i, 2), r, role(i, 2));
    ...
    binary_trying(i, match(i, log N), r, role(i, log N));
    critical section;
    binary_exit(i, match(i, log N), r, role(i, log N));
    ...
    binary_exit(i, match(i, 2), r, role(i, 2));
    binary_exit(i, match(i, 1), r, role(i, 1))
od
```

Fig. 5. The N-process algorithm

the problem instance consists of the code between the end of the trying protocol for the match and the beginning of the exit protocol for the same match.

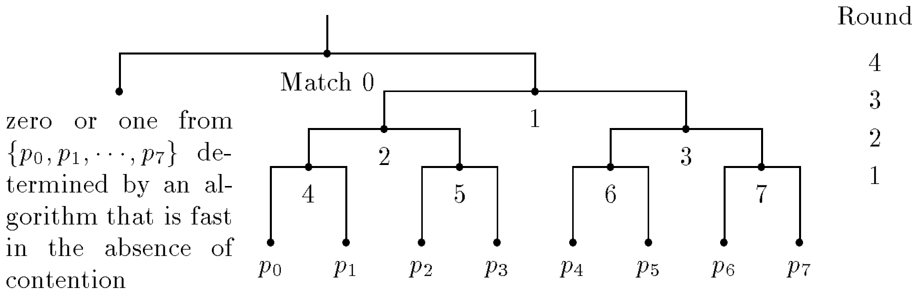
In every problem instance associated with a match of the first round, the processes are well-behaved, since there is exactly one process in each of the two groups of the problem instance. It is easy to see by induction (from bottom to top of the tournament tree) that, in every instance of the dynamic two-process mutual exclusion problem associated with a match, the processes are well-behaved.

In the problem instance associated with Match 1, the execution time of the critical section, which is identical to the critical section as defined in the N-process mutual exclusion problem, is finite by assumption. Again, it is easy to see by induction (from top to bottom of the tournament tree) that, in every instance of the dynamic two-process mutual exclusion problem associated with a match, the execution time of the critical section is finite.

The theorem then follows from Theorem 3. □

5 Concluding Remarks

In our algorithm, a trying process needs to go through $\log N$ rounds of matches even if there are actually no other processes trying to enter their critical sections. The algorithm can be made fast in the absence of contention, like Yang and Anderson's tournament algorithm [10]. Below is their idea in picture:



References

1. T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
2. M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.
3. E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of ACM*, 8:569, 1965.
4. J.L.W. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.
5. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
6. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
7. J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
8. G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.
9. G.L. Peterson and M.J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 91–97, 1977.
10. J.-H. Yang and J.H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.