Mutual Exclusion Algorithms Correctness Proofs

November 6, 2011

1 Peterson’s 2-process algorithm

Lemma 1 The algorithm satisfies mutual exclusion.

Proof: We say that a process enters the CS when its evaluation of the condition of line 3 returns true. Assume the claim is false and consider the first time both processes are in the CS. Without loss of generality, assume that \( p_0 \) entered the CS first. Then \( p_0 \) executed the following sequence of events.

\[
\begin{align*}
p_0: b[0] := \text{true} & \implies p_0: \text{turn} := 0 \\
& \implies p_0: \text{await}(b[1] = \text{false} \text{ or turn} = 1) \\
& \implies p_0: \text{CS}.
\end{align*}
\]

Two possibilities exist. If the condition \( b[1] = \text{false} \) was satisfied, then, in order to enter the CS, \( p_1 \) has to execute the following events after that:

\[
\begin{align*}
p_1: b[1] := \text{true} & \implies p_1: \text{turn} := 1 \\
& \implies p_1: \text{await}(b[0] = \text{false} \text{ or turn} = 0)
\end{align*}
\]

But since \( p_0 \) is in the CS, \( b[0] = 1 \) holds. Thus \( p_1 \) must wait until \( p_0 \) exits the CS.
The second possibility is that, when \( p_0 \) enters the CS, both \( b[1]=\text{true} \) and \( \text{turn}=1 \) hold. It follows that \( p_1 \) executes line 2 after \( p_0 \) does. Consequently, when \( p_1 \) later evaluates the condition of line 3, \((\text{turn}=1) \land (b[0] := \text{true})\) holds until after \( p_0 \) exits the CS and \( p_1 \) must wait. This is a contradiction.  

**Lemma 2** The algorithm satisfies deadlock-freedom.

**Proof:** If only one process is in its entry/exit section, it will succeed in entering the critical section because the flag of the other process is down. If both processes try to enter the CS, then it cannot be that both wait forever in line 3: as \( \text{turn} \) is either 0 or 1, one of them will enter the CS.

**Lemma 3** The algorithm satisfies starvation-freedom.

**Proof:** Assume not. Then one process, WLOG \( p_0 \), waits forever in line 3, while \( p_1 \) does one of the following.

1. Stays forever in the remainder section.

2. Waits forever in line 3.

3. Continuously enters and exits the critical section.

In case 1, \( p_0 \) succeeds in entering the critical section because \( \text{flag}[1]=0 \) holds. From Lemma 2, case 2 cannot happen. As for case 3, once \( p_0 \) waits in line 3 and \( p_1 \) enters and then exits the critical section, \( \text{turn}=1 \) and \( b[0]=\text{true} \) hold until \( p_0 \) enters the CS. Thus \( p_1 \) cannot enter the CS and \( p_0 \) eventually does.

2 Tournament tree based on Peterson’s 2-process algorithm

**Lemma 4** The algorithm satisfies mutual exclusion.
Proof: Assume otherwise to obtain a contradiction. Then there is an execution $E$ in which two processes are in their critical sections at the same time. Thus there are tree nodes $n$ such that, during $E$, two (or more) processes are in the critical section of the 2-process algorithm of $n$. Let $n'$ be the node in which this happens for the first time and let $t$ denote that time. Thus, in time $t$ there are two winning processes in Peterson’s 2-process algorithm of node $n'$. Moreover, from our selection of $n'$, these two processes are the only processes participating in the 2-process algorithm at $n'$, one of them has id 0 at $n$ and the other has id 1 at $n'$. This contradicts the correctness of Peterson’s algorithm.

Lemma 5 The algorithm satisfies starvation freedom.

Proof: Assume otherwise. Then there are processes that wait forever in line 7. Among these processes, let $p$ be a process that waits on a node $n$ with maximum level. It must be that $p$ waits for some process $q$ which is either at that level or a higher one. $q$ cannot be stuck on the same node, because the $\text{turn}$ variable of $n$ is favorable for it. From the maximality of $n$, it cannot be stuck on a higher level. Thus, $q$ eventually enters its CS and performs the exit code for node $n$. From the proof of Lemma 4, no process can bypass $p$ from its sub-tree. Thus, after $q$ performs the exit code $p$ can enter its critical section.

3 Fast mutual exclusion algorithm

Lemma 6 Lamport’s fast mutex algorithm satisfies mutual exclusion

Proof: We say that a process enters the critical section in the fast path if it enters from line 8. We say that a process enters the critical section in the slow path if it enters from line 11. The proof proceeds by proving the following 3 claims.

Claim 1 Consider the first time in which two processes are in the CS. Then it cannot be that the two of them entered through the fast path.
**Proof:** To obtain a contradiction, assume that $p_i, p_j$ are simultaneously at the CS and both entered from the fast path. Consider the last time they write to the fast lock in line 2 before entering the CS. Without loss of generality, assume that $p_i$ is the first to write to the fast lock in line 2. Then it must be that $p_i$ enters the CS from the fast path before $p_j$ executes line 2. We thus have:

\[
p_i : \text{fast-lock} := i \implies p_i : \text{slow-lock} := i
\]

\[
\implies p_i : \text{fast-lock} = i
\]

\[
\implies p_j : \text{fast-lock} := j
\]

\[
\implies p_j : \text{slow-lock} <> 0.
\]

So $p_j$ fails to enter through the fast path. This is a contradiction.

\[
\]

**Claim 2** Consider the first time in which two processes are in the CS. Then it cannot be that the two of them entered through the slow path.

**Proof:** To obtain a contradiction, assume that $p_i, p_j$ are simultaneously at the CS and both entered from the slow path. Assume WLOG that $p_i$ enters first from the slow path. Process $p_i$ has set $\text{slow-lock} := i$ in line 7 and then read $\text{slow-lock} = i$ in line 11. Note that, after $p_i$ finishes the loop of line 10, $\text{slow-lock}$ cannot change until after some process exits the CS. Thus process $p_j$ cannot enter the CS through the slow path before $p_i$ exits the CS.

\[
\]

**Claim 3** Consider the first time in which two processes are in the CS. Then it cannot be that one of them came through the slow path and the other came through the fast path.

**Proof:** Assume a process coming from the fast path, $p_i$, enters the CS first, followed by a process coming from the slow path, $p_j$. Then $p_i$ performs the following sequence of events:

\[
p_i : \text{want}[i] := \text{true} \implies p_i : \text{fast-lock} := i \implies p_i : \text{slow-lock} = 0
\]

\[
\implies p_i : \text{slow-lock} := i \implies p_i : \text{fast-lock} = i.
\]
Thus when \( p_i \) enters the CS, all other processes are either in the remainder section or before line 7 (and will remain there until \( p_i \) exits the CS), or are in lines 8-12. All of these latter processes will eventually evaluate the condition of line 12 as true and will jump to line 1, since \( p_i \) resets its flag only after it zeros slow-lock.

Assume then that \( p_i \) enters the CS first through the slow path, followed by \( p_j \) that enters through the fast path. Then \( p_i \) performs the following sequence of events:

\[
p_i: \text{fast-lock} := i \quad \Rightarrow \quad p_i: \text{slow-lock} := i \quad \Rightarrow \quad p_i: \text{want}[j] = false
\]

\[
\Rightarrow \quad p_i: \text{slow-lock} = i.
\]

Let \( t \) be the time when \( p_i \) finds that \( \text{want}[j] = false \). In time \( t \), \( p_j \) is not in its entry or exit code. Moreover, in time \( t \) slow-lock is non-zero. Thus \( p_j \) cannot enter through the fast path until \( p_i \) performs line 15.

The progress proofs for Lamport’s algorithm are left as an exercise.