5.3 Peterson's Algorithm

The shortcoming of Lamport's Bakery Algorithm is that the variable NUM is not bounded. However, Peterson's algortihm is bounded and still provides Mutual Exclusion and No Starvation. We start with a simplified version of Peterson's Algorithm for 2 processors, shown as Algorithm 3.

Algorithm 3 Peterson's Algorithm With Starvation

Code for processor p_0

Entry Section

1 $want_0 := 1$;

2 **wait until** $(want_1 = 0)$;

Critical Section

Exit Section

3 $want_0 := 0$;

Remainder Section

Code for processor p_1

Entry Section

1 $want_1 := 0$;

2 **wait until** $(want_0 = 0)$;

3 $want_1 := 1$;

4 **if** $(want_0 = 1)$ **then goto 1**;

Critical Section

Exit Section

5 $want_1 := 0$;

Remainder Section

Note that this algorithm is not symmetric. Whenever p_0 wants to enter the critical section, and sets its *want* bit to 1, p_1 has to give up trying to enter the critical section, and reset its *want* bit. Therefore. p_0 always has priority while processor p_1 never has priority to enter the critical section, thus causing starvation of p_1. See the starvation senario in Figure 5.3. First, p_0 sets $want_0$ to 1 in line 1, passes the wait step in line 2 by reading $want_1 = 0$ and enters critical section. At the same time, p_1 waits at line 2 on reading $want_0 = 1$. After p_0 exits critical section, and sets $want_0 = 0$, it immediately re-enters the entry section and sets $want_0$ to be 1 again. At this point, p_1 reads $want_0 = 1$ again, and continues to wait at line 2. This process can repeat, so, in this scenario, p_1 will be starved.

However, the algorithm is bounded and provides both mutual exclusion and no deadlock. We would like to remark that if the entry sections of p_0 and p_1 are symmetric, then the process could be deadlocked as both processes can start at the same time.

For example, assume that both p_0 and p_1 both operate the "Code for processor p_0" above. Suppose that they enter their entry sections at the same time. Now p_0 sets $want_0 = 1$ and p_1 sets $want_1 = 1$ simultaneously. Now in line 2 of the Entry section, both p_0 and p_1 loop until the *want* variable of the other processor is 0. However, this event will never occur as neither reaches their exit section to set their *want* variables to 0. So the processors are deadlocked.

Similarly, assume that both p_0 and p_1 both operate the "Code for processor p_1" above. Suppose that

Figure 10: Starvation for processor p_1

they enter their entry sections at the same time. Now p_0 sets $want_0 = 0$ and p_1 sets $want_1 = 0$ simultaneously. Now in line 2 of the entry section, both p_0 and p_1 pass through the wait loop, and set their *want* variables to 1 simultaneously in line 3. Now, in line 4, they both read the *want* variable of the other processor to be 1 and go back to line 1 simultaneously. This sequence of events is repeated infinitely often, and again the processors are deadlocked.

We can therefore modify the algorithm to instead alternate the priority of the processors, using a priority bit that decides who has priority to enter the critical section. The modified algorithm is symmetric and bounded, and provides mutual exclusion and no starvation. The algorithm, which uses three shared bits (the two want bits and the priority bit), is described as Algorithm 4. We will prove the mutual exclusion and no starvation properties of this 2-processor algorithm.

Lemma 5.6. *If p_i is in lines 4–7 (including the critical section), then $want_i = 1$.*

Proof. We only need to look at p_i's code, since only p_i modifies $want_i$. In line 3, p_i sets $want_i = 1$ and never sets $want_i$ again in lines 4–7. □

Theorem 5.7. *This algorithm provides Mutual Exclusion.*

Proof. Suppose for contradiction, p_0 and p_1 are in critical section together at time t. This scenario is illustrated in Figure 11.

Let w_0 be the last time p_0 executes line 3 (and sets $want_0$ to 1) before it is in critical section at time t. Similarly, let w_1 be the last time p_1 executes line 3 (and sets $want_1$ to 1) before it is in critical section at time t. Also, let r_0 (and r_1) be the last time p_0 (and p_1, respectively) executes line 5 or 6, reading $want_1 = 0$ (and $want_0 = 0$, respectively) before entering critical section.

Algorithm 4 Modified Peterson's Algorithm Without Starvation

Code for processor p_0

Entry Section

1 $want_0 := 1;$
2 **wait until** $(want_1 = 0)$ **or** $(priority = 0);$
3 $want_0 := 1;$
4 **if** $(priority = 1)$
5 **then if** $(want_1 = 1)$ **then goto** *line* 1 **end**;
6 **else wait until** $(want_1 = 0);$

Critical Section

Exit Section

7 $priority := 1;$
8 $want_0 := 0;$

Remainder Section

Code for processor p_1

Entry Section

1 $want_1 := 0;$
2 **wait until** $(want_0 = 0)$ **or** $(priority = 1);$
3 $want_1 := 1;$
4 **if** $(priority = 0)$
5 **then if** $(want_0 = 1)$ **then goto** *line* 1 **end**;
6 **else wait until** $(want_0 = 0);$

Critical Section

Exit Section

7 $priority := 0;$
8 $want_1 := 0;$

Remainder Section

Clearly, $w_0 < r_0 < t$ and $w_1 < r_1 < t$. By Lemma 5.6, $want_0 = 1$ between time w_0 and t. Since p_1 reads $want_0 = 0$ at time r_1, it follows that $r_1 < w_0$. Also, by Lemma 5.6, $want_1 = 1$ between time w_1 and t. Since p_0 reads $want_1 = 0$ at time r_0, it follows that $r_0 < w_1$.

Now $w_0 < r_0$ and $r_0 < w_1$ imply $w_0 < w_1$. However, $w_1 < r_1$ and $r_1 < w_0$ imply $w_1 < w_0$, a contradiction. ☐

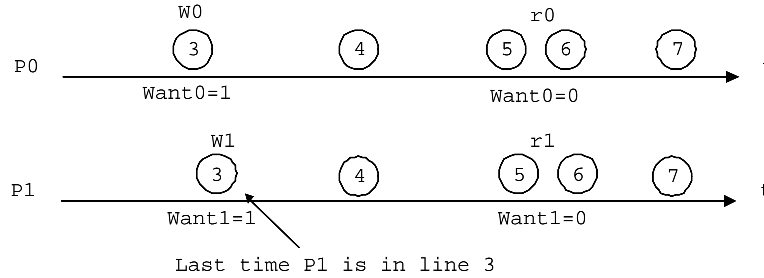

Figure 11: Illustration for the Proof of Theorem 5.7

Theorem 5.8. *This algorithm provides no deadlock.*

Proof. Suppose, for contradiction, after some time t, some processor is forever in entry section, but no processor is in critical section.

We consider 2 cases:

(1) Both processors are in entry section forever starting at time $s \geq t$. After some time $s' \geq s$, each processor is either

(a)looping in line 2, or

(b)cycling through lines 1–5, or

(c)looping in line 6.

Without loss of generality, assume $priority = 0$ at time s. Note that $priority$ cannot change after time t since the only time it is modified is in line 7 and no processor is in line 7 after time s.

Consider processor p_0. Since $priority = 0$, p_0 cannot loop in line 2; *i.e.*, Case (a) is impossible. For the same reason, the **if** clause in line 4 is not satisfied, so p_0 will not execute line 5; *i.e.*, Case (b) is impossible. So, p_0 passes lines 2–4 and loops in line 6 with $want_0 = 1$; *i.e.*, Case (c) must hold.

Consider p_1. After time s', p_0 is looping in line 6 with $want_0 = 1$, so p_1 will never pass line 2; *i.e.*, Case (b) is impossible. Since $priority = 0$, p_1 must have been the last processor to set the $priority$ bit, when it last exited the critical section. So, when p_1 enters the entry section this time, the **if** clause in line 4 is always satisfied, so p_1 will never execute line 6; *i.e.*, Case (c) is impossible. Therefor p_1 will loop in line 2 after time s', *i.e.*, Case (a) must hold.

However, starting at time s', when p_1 loops in line 2, $want_1 = 0$. Now, since p_0 loops in line 6, p_0 will read $want_0$ continuously. So, eventually p_0 will read $want_1 = 0$, and enter critical section. Hence there is a contradiction, and the algorithm provides no deadlock.

(2) Only one processor is in entry section forever. Without loss of generality, let it be p_0. So, at all times after some time $s > t$, p_0 is in entry section while p_1 is in remainder section. So, after time s, $want_1 = 0$ and p_0 cannot loop in line 2, lines 1–5, or line 6. So p_0 enters critical section. Hence there is a contradiction, and the algorithm provides no deadlock.

□