A Family of 2-process Mutual Exclusion Algorithms

Notes on UNITY: 13-90

Jayadev Misra*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

2/8/90

1 Introduction

In Peterson [1981], Peterson suggested a mutual exclusion algorithm. The algorithm, though simple in structure, seems quite involved when a formal proof is attempted. We suggest how Peterson's algorithm and some similar algorithms may be derived by starting with a high-level algorithm and refining it. The refinement consists of implementing a complex shared data structure by elementary data structures.

2 Mutual Exclusion Using a Shared Queue

The mutual exclusion problem is easily solved if there is a shared queue that the processes can access in an exclusive manner (it seems paradoxical to solve the mutual exclusion problem using a facility that already implements mutual exclusion in the access to the data structure; we will simplify matters in the next section). A process attempting to enter its critical section appends its identity to the back of the queue; the process at the head of the queue enters its critical section and upon completion of its critical section removes its identity from the queue. The correctness of this solution is easy to see: The process executing its critical section is at the head of the queue and hence, at most one process can execute its critical section at any time. Furthermore, a process attempting to execute its critical section will do so eventually if every process completes its critical section in finite time. This is because (1) the process at the head of the queue enters its critical section, completes it and then removes the head item of the queue, (2) from (1), every item in the queue eventually becomes the head item and hence every process whose id is in the queue eventually enters the critical section, (3) a process attempting to enter its critical section appends its id to the queue and hence, from (2), it enters its critical section eventually.

In the program, given below, q is the shared queue and append, head, tail, empty have their usual meanings. The algorithm is described for two processes u, v; we write their ids as "u" and "v", respectively. We have employed Dijkstra's guarded command notation, Dijkstra [1976], where

*This work was partially supported by ONR Contract 26-0679-4200 and by Texas Advanced Research Program grant 003658–065.

if $b \rightarrow$ skip **fi**

amounts to a wait as long as b is false, and completes only after b has been (detected to be) true; if b becomes true only intermittently, there is no guarantee about the completion of this statement.

initially q = empty

process u

loop
  noncritical section;
  q := append(q,"u");
  if head(q) = "u" $\rightarrow$ skip **fi**;
  critical section;
  q := tail(q)
end

process v

loop
  noncritical section;
  q := append(q,"v");
  if head(q) = "v" $\rightarrow$ skip **fi**
  critial section;
  q := tail(q)
end

3 Implementing the Shared Queue for 2 Processes: Peterson's Algorithm

The queue of Section 2 takes on five possible values when it is shared between two processes: empty, "u", "v", "u" "v", "v" "u" (here "u" "v" represents the queue that has "u" as the head item followed by "v"). Therefore, we need at least three boolean variables to represent the queue. Let boolean variables u, v be true when the corresponding id is in the queue. Then it remains to distinguish between the two queue values, "u" "v" and "v" "u". We introduce a boolean variable *turn* that is true when q = "v" "u" and false when q = "u" "v"; the value of *turn* in other cases is irrelevant. The operations on the queue can now be written in terms of the operations on u, v, *turn*, as follows.

q = empty	is	u, v := false, false
q := append (q, "u")	is	u, *turn* := true, true
head(q) = u	is	$\neg v \ \vee \ \neg turn$
q := tail(q) {in process u}	is	u := false

To see the transformation for append, note that q becomes "u" or "v" "u" as a result of appending "u" to it. In the first case, *turn*'s value is irrelevant and in the second case, *turn* has to be set true; therefore, we set *turn* true in both cases in addition to setting u to true. The test, head(q) = u, given that u is in q is equivalent to, v is not in q or q = "u" "v", i.e., $\neg v \ \vee \ \neg turn$. With similar transformations in process v we obtain the following program.

initially u, v = false, false

process u

loop
  noncritical section;
  u,*turn* := true, true;

process v

loop
  noncritical section;
  v,*turn* := true, false;

if $\neg v \ \vee \ \neg turn \rightarrow$ skip **fi**;	**if** $\neg u \vee turn \rightarrow$ skip **fi**;
critical section;	critial section;
$u \ :=$ false	$v \ :=$ false
end	**end**

From Multiple Assignments to Single Assignments

The algorithm given above is (almost) Peterson's 2-process mutual exclusion algorithm. The remaining step is to decouple the assignments to u,*turn* in process u (and similarly v,*turn* in process v). We show that it is safe to replace

u,*turn* := true, true

by

u := true; *turn* := true

We employ a theory developed in Feijen [1987] to justify this replacement. The guard in any **if** command can be strengthened without affecting the safety of the computation, because strengthening a guard merely disallows certain execution sequences. We strengthen the guard in process v to

$\neg w \vee turn$

where w is a new boolean variable satisfying $\neg w \ \Rightarrow \ \neg u$, i.e., $u \ \Rightarrow \ w$. In order to satisfy $u \ \Rightarrow \ w$, we have w := true immediately preceding u,*turn* := true, true in process u. Also, it is safe to set w to false whenever u is set false.

Now, variable u is an auxiliary variable—it does not appear in any test nor in assignment to any other variable. So, variable u can be removed from the program. In the resulting program variables w and *turn* are assigned in the following order,

w := true; *turn* := true

Everywhere else, w has assumed the role of u. Renaming w to u completes the transformation in process u. After applying a similar transformation in process v, we obtain Peterson's algorithm, below.

initially $u, v =$ false, false

process u	process v
loop	**loop**
noncritical section;	noncritical section;
u := true; *turn* := true;	v := true; *turn* := false;
if $\neg v \ \vee \ \neg turn \rightarrow$ skip **fi**;	**if** $\neg u \vee turn \rightarrow$ skip **fi**;
critical section;	critial section;
$u \ :=$ false	$v \ :=$ false
end	**end**

Note: The given transformation preserves the safety property, namely the property of mutual exclusion. We have not shown that progress properties—absence of starvation—are preserved. We do so in Section 5 for another algorithm derived from the high-level algorithm of Section 2.

4 A New 2-process Mutual Exclusion Algorithm

We suggest a different implementation of the shared queue of Section 2. As in Section 3, let u, v be true whenever the corresponding process id is in the queue. We introduce a boolean variable p that is true when the queue has v as its head item (i.e., the queue is "v" or "v" "u"), and false when the queue has u as its head item; value of p is immaterial if the queue is empty.

Note that p has different values when $q =$ "v" "u" and when $q =$ "u" "v". Thus, all five possible queue values are distinguished by u, v, p. Also, note that p is "more defined" than *turn* of Section 3: When the queue has two elements, the two variable values match; when the queue has one element, *turn*'s value is irrelevant though p's value is determined; for empty queue both variable values are irrelevant.

The tests and assignments in process u are transformed as follows.

q := append(q, "u")	is	p, u := v,true
head(q) = u	is	$\neg p$
q := tail(q)	is	p, u := true, false

The first transformation can be seen by noting that appending "u" results in a queue that is either "u" or "v" "u". In either case, u is to be set true. In the first case, p is to be set true and in the second case p is to be set false; in either case it acquires the value of v. The other two transformations are easily justified.

The resulting program is (after similar transformations on process v)

initially u, v = false, false

process u

loop
  noncritical section;
  p, u := v,true;
  if $\neg p \rightarrow$ skip **fi**;
  critical section;
  p, u := true, false
end

process v

loop
  noncritical section;
  p, v := $\neg u$,true;
  if $p \rightarrow$ skip **fi**;
  critial section;
  p, v := false, false
end

The above program is simple enough that mutual exclusion and the absence of deadlock can be established from its text, directly. First note that $u \wedge \neg p$ is preserved by every statement in process v (and similarly, $v \wedge p$ in process u). Whenever process u is in its critical section $u \wedge \neg p$ holds; whenever process v is in its critical section $v \wedge p$ holds. Hence, mutual exclusion is guaranteed. Process u waits if p holds and v waits if $\neg p$ holds; therefore the two processes are never deadlocked. We prove absence of starvation in the next section.

The program given here has the advantage over Peterson's that exactly one boolean variable has to be checked in the guard of the **if** statement. Unfortunately, the program requires assignments of the form $p := v$ and $p := \neg u$, naming shared variables on both sides of an assignment.

From Multiple Assignments to Single Assignments

We suggest a particular order in which each of the assignments in the multiple assignment statements can be executed. The resulting program is shown below.

initially u, v = false, false

process u

loop
  noncritical section;
  u := true; p := v;
  if $\neg p \rightarrow$ skip **fi**;
  critical section;
  u := false; p := true
end

process v

loop
  noncritical section;
  v := true; p := $\neg u$;
  if $p \rightarrow$ skip **fi**;
  critial section;
  v := false; p := false
end

Note: Switching the order of the two assignments after either critical section makes the program incorrect. To see this, suppose: process v is in its critical section and process u assigns true to u; process v completes its critical section and assigns false to p; process u assigns true to p since v is still true; process v assigns false to v and then remains in its noncritical section forever. Now p will remain true forever and hence, process u will never enter its critical section. □

The following is an informal proof of starvation freedom. We show that once u is in the queue, it eventually becomes the head of the queue and enters the critical section. Whenever u is *true*, the queue configuration is one of vu, uv or u. In the first case, the condition $(v \wedge p)$ holds. This cannot be falsified by process u and hence process v enters its critical section. Eventually u becomes the head of the queue and then $(u \wedge \neg p)$ holds. In the last case, $(u \wedge \neg p)$ holds. This condition cannot be falsified by process v and hence eventually process u enters its critical section.

5 A Formal Proof of the Algorithm of Section 4

The proof given in the last section is sufficiently confusing and unreliable to warrant a formal proof. Fortunately, the formal proof is quite succinct. First, we rewrite the program of Section 5 introducing two explicit program counters—m for process u and n for process v—that take on integer values in the range 0 to 4.

initially u, v, m, n = false, false, 0, 0

process u

loop
  noncritical section;
  u, m := true, 1; p, m := $v, 2$;
  if $\neg p \rightarrow$ skip **fi**;
  {enter critical section}m := 3;
  critical section;
  u, m := false, 4; p, m := true, 0
end

process v

loop
  noncritical section;
  v, n := true, 1; p, n := $\neg u, 2$;
  if $p \rightarrow$ skip **fi**;
  {enter critical section}n := 3;
  critial section;
  v, n := false, 4; p, n := false, 0
end

Next, we translate this program to a UNITY program (Chandy and Misra [1988]); the translation is mostly automatic. The reason for working within the UNITY notation is that a number of useful proof rules can now be employed. We do not represent the noncritical sections explicitly—zero values for m, n denote that the corresponding process is in its noncritical section. Also, we don't represent the critical sections explicitly: $m = 3$ if process

u is in its critical section (similarly for process v). We postulate that u completes its noncritical section if some, externally determined, predicate $u.h$ becomes true (similarly, we have $v.h$ for v). We assume that every critical section is eventually completed, i.e., m is set 4 some time after it becomes 3 (similarly for n).

Program 2-mutex
initially u, v, m, n = false, false, 0, 0
assign

$$
\begin{array}{lll}
\{\text{process } u\text{'s program}\} & & \\
u, m := \text{true}, 1 & \text{if} & u.h \ \wedge \ m = 0 \\
[\!] \ p, m := v, 2 & \text{if} & m = 1 \\
[\!] \ m := 3 & \text{if} & \neg p \ \wedge \ m = 2 \\
[\!] \ u, m := \text{false}, 4 & \text{if} & m = 3 \\
[\!] \ p, m := \text{true}, 0 & \text{if} & m = 4 \\
 & & \\
[\!] \ \{\text{process } v\text{'s program}\} & & \\
v, n := \text{true}, 1 & \text{if} & v.h \ \wedge \ n = 0 \\
[\!] \ p, n := \neg u, 2 & \text{if} & n = 1 \\
[\!] \ n := 3 & \text{if} & p \ \wedge \ n = 2 \\
[\!] \ v, n := \text{false}, 4 & \text{if} & n = 3 \\
[\!] \ p, n := \text{false}, 0 & \text{if} & n = 4
\end{array}
$$

end

Proofs of Safety Property: Mutual Exclusion

We show that m, n cannot both be 3 (i.e., both processes cannot be in their critical sections) simultaneously. The following invariants are proven by showing that they hold initially and that every statement preserves the truth of each of these predicates.

I1. $[1 \leq m \leq 3 \ \equiv \ u] \ \wedge \ [3 \leq m \leq 4 \ \Rightarrow \ \neg p]$
I2. $[1 \leq n \leq 3 \ \equiv \ v] \ \wedge \ [3 \leq n \leq 4 \ \Rightarrow \ p]$

Given I1 $\wedge$ I2,
$m = 3 \ \wedge \ n = 3$
$\Rightarrow \{m = 3 \ \Rightarrow \ \neg p \ , \ n = 3 \ \Rightarrow \ p\}$
false

Proof of Progress Property: Absence of Starvation

The following facts can be proven from the program text. Proofs F0, F1, F2 are direct; the latter two (F1, F2) are *ensures* properties. Property F3 follows from $m = 3$ *ensures* $m = 4$ and $m = 4$ *ensures* p.

F0. $m = 2$ *unless* $m = 3$
F1. $m = 1 \ \mapsto \ p = v \ \wedge \ m = 2$
F2. $\neg p \ \wedge \ m = 2 \ \mapsto \ m = 3$
F3. $m = 3 \ \mapsto \ p$

In the following proof we use three rules in addition to reflexivity, transitivity and disjunction on *leads-to*.

- Implication: The rhs of a *leads-to* property can be weakened.

• PSP: An *unless* property and *leads-to* property can be used to derive a *leads-to* property as follows.

$$\frac{\begin{array}{c} p \ \textit{unless} \ q \\ r \ \mapsto \ b \end{array}}{p \ \wedge \ r \ \mapsto \ (p \ \wedge \ b) \ \vee \ q}$$

• Cancellation: If the rhs of a *leads-to* is of the form $q \ \vee \ r$, and we have $r \ \mapsto \ b$, then the rhs may be replaced by $q \ \vee \ b$.

F4. $u \ \mapsto \ p$

	$\neg p \ \wedge \ m = 2$	$\mapsto \ p$	, transitivity on (F2,F3)
	$p \ \wedge \ m = 2$	$\mapsto \ p$	, $p \ \wedge \ m = 2 \ \Rightarrow \ p$
(1)	$m = 2$	$\mapsto \ p$	, disjunction on the above two
	$m = 1$	$\mapsto \ m = 2$	, weakening the rhs of F1
(2)	$m = 1$	$\mapsto \ p$	, transitivity on the above two
	$1 \leq m \leq 3$	$\mapsto \ p$	, disjunction on (2,1,F3)
	u	$\mapsto \ p$	, using I1 : $1 \leq m \leq 3 \ \equiv \ u$.

F5. $v \ \mapsto \ \neg p$: Proof similar to F4 (switch the roles of u and v, m and n, p and $\neg p$)

F6. (Absence of starvation) $m = 1 \ \mapsto \ m = 3$

$m = 1$	$\mapsto \ p = v \ \wedge \ m = 2$	
		, rewriting F1
$m = 1$	$\mapsto \ (p \ \wedge \ v \ \wedge \ m = 2) \ \vee \ (\neg p \ \wedge \ \neg v \ \wedge \ m = 2)$	
		, rewriting rhs of above
$m = 1$	$\mapsto \ (v \ \wedge \ m = 2) \ \vee \ (\neg p \ \wedge \ m = 2)$	
		, weakening rhs of above
$v \ \wedge \ m = 2$	$\mapsto \ (\neg p \ \wedge \ m = 2) \ \vee \ m = 3$	
		, PSP on F0, F5
$m = 1$	$\mapsto \ (\neg p \ \wedge \ m = 2) \ \vee \ m = 3$	
		, from the above two using cancellation
$m = 1$	$\mapsto \ m = 3$	
		, from the above and F2 using cancellation

6 Conclusion

Our derivation gave us two different algorithms for 2-process mutual exclusion. It is conceivable that other interesting implementations of the shared queue exist. Observe that *turn* of Section 3 is used to distinguish between two configurations of the queue; its traditional interpretation as a priority—*turn* is true means process v has priority—is entirely incidental. Also, our development of the algorithm using a queue establishes other facts—such as, a process may overtake a waiting process at most once, which follows from the FIFO nature of the queue—that are quite difficult to prove from the refined version.

For 3 processes, the number of possible queue configurations is 16. A tantalizing possibility is that four shared bits suffice for 3-process mutual exclusion. I am skeptical that such an algorithm, that moreover retains the symmetry among the processes, exists.

Acknowledgment

I am grateful to Edgar Knapp for his careful reading and critical suggestions. The informal proof of absence of starvation at the end of Section 4 is due to J. R. Rao.

7 References

1. Chandy, K. M., and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

2. Dijkstra, E. W., *A Discipline of Programming*, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

3. Feijen, W. H. J., "A Little Program Transformation," *WF8*, Department of Computer Sciences, The University of Texas at Austin, 1987.

4. Peterson, G. L., "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, **12**:3, June 1981, pp. 115–116.