Locks and Barriers

Reading Chapter 3 of Andrews.

- Critical sections and locks
  - Abstract solution
  - Hardware Solution
  - Software Solutions
    - After-you
    - Safe-Sluice
    - Peterson's algorithm
- Barriers
  - Abstract solution
  - Various solutions
  - Application

Critical Sections

So how can we implement atomicity?

Suppose a concurrent program has a number of *unconditional* atomic actions $\langle S_0 \rangle$, $\langle S_1 \rangle$,..., $\langle S_n \rangle$.

And, no variable written in any of these actions is read anywhere outside any of the actions.

Then we can implement each of these actions as follows

$$\langle S_i \rangle \longrightarrow \mathsf{CSEnter}\ S_i\ \mathsf{CSExit}$$

S_i is called a *critical section*.

We need to ensure that, as a global invariant,

At most one thread has executed CSEnter *more often than it has executed* CSExit.

To simplify, we will assume each thread has one critical section.

```
init
process CS[i = 1 to n] {
        while (...) {
                noncritical section
                CSEnter
                critical section
                CSExit }
}
```

Assume any thread entering its critical section will eventually leave it.

Mutual Exclusion: At most one thread is in its critical section at time.

Absence of Deadlock: If two or more threads are trying to enter, one will succeed.

Absence of Unnecessary Delay: A thread gets to enter CS without unnecessary delay.

Eventual Entry: A thread trying to enter CS will eventually succeed.

Specification of the problem

Let's use a variable to count the number of threads that are executing their critical section.

---

int $\#in := 0;$
# desired global invariant: $0 \leq \#in \leq 1$
process CS[$i = 1$ **to** n] {
    while (**true**) {
        noncritical section
        $\langle \#in := \#in + 1; \rangle$
        critical section
        $\langle \#in := \#in - 1; \rangle$ } }

Both assignments interfere with the desired global invariant.

A correct algorithm (but a failed proof)

We can eliminate the interference from $\langle \#in := \#in + 1; \rangle$ using the techinque of "Strengthen the precondition via conditional synchronization."

---

int $\#in := 0;$
# desired global invariant: $0 \leq \#in \leq 1$
process CS[$i = 1$ **to** n] {
    while (**true**) {
        noncritical section
        $\langle \mathbf{await}(\#in = 0)\ \#in := \#in + 1; \rangle$
        critical section
        ## $\#in = 1$
        $\langle \#in := \#in - 1; \rangle$ } }

The problen is that $\langle \#in := \#in - 1; \rangle$ in one thread interferes with the assertion $\#in = 1$ in all other threads.

We *could* use the conditional synchronization trick again, but does it make sense that a thread should wait to leave the critical section?

No. The algorithm is correct. The proof needs fixing.

A better way to count

Because of mutual exclusion, no other thread can be at the problematic assertion. Can we strengthen its precondition to show that?

We count the threads in the critical section with a boolean array *in*. Now define $\#$ to be the population count function

$$\#in \;=\; \sum_{i \in \{1,..,n\}} toInt(in[i])$$

where $toInt(false) \;=\; 0$ and $toInt(true) = 1$

---

bool $in[1 : n] := ([n]false);$
# global invariant: $0 \leq \#in \leq 1$
process CS[$i = 1$ **to** n] {
    while (**true**) {
        noncritical section
        $\langle$**await**$(\#in = 0)$ $in[i] := true;\rangle$
        critical section
        ## $in[i]$ — by disjoint variables
        $\langle in[i] := false;\rangle$ } }

Data refining the coarse-grained solution.

Introduce a boolean variable *lock* defined by the global invariant

$$lock = (\#in = 1)$$

Demote the *in* array to a ghost variable.

---

bool $lock := \mathbf{false}$;
bool $in[1 : n] := ([n]false)$; # ghost var
# global invariant: $0 \leq \#in \leq 1 \quad \wedge \quad lock = (\#in = 1)$
process CS[$i = 1$ **to** n] {
    while (**true**) {
        noncritical section
        $\langle \mathbf{await}(\neg lock)\ in[i] := true;\ lock := true; \rangle$
        critical section
        ## $in[i]$ — by disjoint variables
        $\langle in[i] := false;\ lock := false; \rangle$ } }

Now we are back to our earlier correct algorithm, but with the improvement of using a boolean counter. (And of having a correct proof.)

But! How can we implement the first await-statement atomically?

A Hardware Solution

Modern CPUs offer instructions to aid mutual exclusion. One such is Test-and-set.

Test-and-set atomically reads and writes a variable as follows.

TS r_i r_j is $\langle r_i := M[r_j];\ M[r_j] := true; \rangle$

On a uniprocessor turn off all interrupts.

On a multi-processor, cooperation of bus&caches is needed.

Refine the above course grained solution with:

```
bool lock := false;
process CS[i = 1 to n] {
    while (true) {
        noncritical section
        do { r1 := &lock ;
             TS r0 r1
        } while( r0 );
        critical section
        lock := false; } }
```

However, this does not ensure eventual entry given weak-fairness.

“Software” Solutions

One can also enforce mutual exclusion using only store and fetch instructions to communicate.

- Peterson’s tie-breaker algorithm — see next few slides
- The Bakery algorithm — see Andrews’s book
- Dekker’s algorithm — see exercise 3.1 in Andrews’s book

The after-you algorithm

The *after-you* algorithm for mutual exclusion is

---

init: int $t := 0$; # turn indicator ## invariant $t \in \{0, 1\}$

P_0 :	P_1 :
$t := 1;$	$t := 0;$
$\langle$await($t = 0$)$\rangle$	$\langle$await($t = 1$)$\rangle$
$\{t = 0\}$	$\{t = 1\}$
CS_0	CS_1

There is no interference since $\{t = 0\}t := 0\{t = 0\}$

This enforces mutual exclusion. Since we can't have both $t = 0$ and $t = 1$ at the same time.

It is deadlock free, since t must be either 0 or 1.

But it does cause unnecessary delay since P_1 can not enter until P_0 sets t to 1.

It does not ensure eventual entry either.

In short *after-you* is a good start, but not a suitable algorithm.

Safe-Sluice

The *safe-sluice* algorithm is:

---

init: bool $r[2] := [false, false]$ # Request flags

P_0 :	P_1 :
$r[0] := true;$	$r[1] := true;$
$\langle$await($\neg r[1]$)$\rangle$	$\langle$await($\neg r[0]$)$\rangle$
$\{A_0\}$	$\{A_1\}$
CS_0	CS_1
$r[0] := false;$	$r[1] := false;$

Safe-sluice enforces mutual exclusion.

But it can lead to deadlock. Consider if both set their request flags before either executes the await.

We will fix this problem later.

First let's prove that mutual exclusion is enforced.

We would like to find predicates A_0 and A_1 such that

- A_0 is true when P_0 is in its critical section.
- A_1 is true when P_1 is in its critical section.

- Both can't be true at once: $\neg(A_0 \wedge A_1)$

We might try

$$A_0 \triangleq r[0] \wedge \neg r[1] \qquad A_1 \triangleq r[1] \wedge \neg r[0]$$

A_0 is true as soon as P_0 enters, but the assignment $r[1] := true$ interferes with this A_0.

A second try

We introduce a "ghost variable" t into the safe-sluice.

The purpose of t is to facilitate reasoning and so we don't need to include it in the implementation.

When $r[0]$ and $r[1]$, then $t = i$ indicates that P_i was first to set its request flag..

init: bool $r[2] := [false, false]$ # Request flags
init: int t := 0 # ghost variable ## invariant $t \in \{0, 1\}$

P_0 :	P_1 :
$\langle r[0] := true; \mathbf{t} := \mathbf{1};\rangle$	$\langle r[1] := true; \mathbf{t} := \mathbf{0};\rangle$
$\langle$await($\neg r[1]$)$\rangle$	$\langle$await($\neg r[0]$)$\rangle$
$\{A_0\}$	$\{A_1\}$
CS_0	CS_1
$r[0] := false;$	$r[1] := false;$

Now consider $A_0 \triangleq r[0] \wedge (\neg r[1] \vee t = 0)$

There is no interference since

$$\{A_0\}\ \langle r[1] := true; t := 0\rangle\ \{A_0\}$$

And we have

$$\begin{aligned} & A_0 \wedge A_1 \\ = \ & r[0] \wedge (\neg r[1] \vee t = 0) \wedge r[1] \wedge (\neg r[0] \vee t = 1) \\ = \ & r[0] \wedge r[1] \wedge t = 0 \wedge t = 1 \\ = \ & false \end{aligned}$$

Eliminating deadlock

If both threads get stuck in the safe sluice, then why not use t to decide which should be unstuck?

If $t = i$ then P_i gets to go.

We change the wait condition. Note A_i is still a postcondition of the awaits.

t is no longer a mere ghost variable.

---

init: bool $r[2] := [false, false]$ # Request flags
init: int $t := 0$ # Turn indicator ## invariant $t \in \{0, 1\}$

P_0 :	P_1 :
$\langle r[0] := true; t := 1; \rangle$	$\langle r[1] := true; t := 0; \rangle$
$\langle$await($\neg r[1] \vee \mathbf{t = 0}$)$\rangle$	$\langle$await($\neg r[0] \vee \mathbf{t = 1}$)$\rangle$
$\{A_0\}$	$\{A_1\}$
CS_0	CS_1
$r[0] := false$;	$r[1] := false$;

Since t is either 0 or 1, one or the other can always proceed.

Splitting the atomic assignment

Peterson's key insight is that the atomic pair of assignments can be split (if you split it the right way) and the algorithm still works. This gives us Peterson's tie breaker algorithm:

---

init: bool $r[2] := [false, false]$ # Request flags
init: int $t := 0$ # Turn indicator ## invariant $t \in \{0, 1\}$

P_0 :	P_1 :
$\mathbf{r[0]} := \mathbf{true}$	$\mathbf{r[1]} := \mathbf{true}$
$\mathbf{t} := \mathbf{1}$;	$\mathbf{t} := \mathbf{0}$;
$\langle$await($\neg r[1] \vee t = 0$)$\rangle$	$\langle$await($\neg r[0] \vee t = 1$)$\rangle$
$\{B_0\}$	$\{B_1\}$
CS_0	CS_1
$r[0] := false$;	$r[1] := false$;

Let's prove mutual exclusion.

We need B_0 and B_1 such that $\neg(B_0 \wedge B_1)$.

We can't use $B_0 = A_0$ because of interference from $r[1] := true$.

We have

$$\{A_0\}\ r[1] := \mathit{true}; t := 0;\ \{A_0\}$$

but not

$$\{A_0\}\ r[1] := \mathit{true}\ \{A_0\}$$

There is a nasty spot between these two statements.

So we introduce a ghost variable $n[i]$ to indicate when P_i is in the nasty spot.

This gives:

init: bool $r[2] := [false, false]$ # Request flags
init: int $t := 0$ # Turn indicator ## invariant $t \in \{0, 1\}$
init bool n[2] := [false, false] # Nasty spot

P_0 :	P_1 :
$\langle r[0] := true; \mathbf{n[0]} := \mathbf{true};\rangle$	$\langle r[1] := true; \mathbf{n[1]} := \mathbf{true};\rangle$
$\langle t := 1; \mathbf{n[0]} := \mathbf{false};\rangle$	$\langle t := 0; \mathbf{n[1]} := \mathbf{false};\rangle$
$\langle$await($\neg r[1] \vee t = 0$)$\rangle$	$\langle$await($\neg r[0] \vee t = 1$)$\rangle$
$\{B_0\}$	$\{B_1\}$
CS_0	CS_1
$r[0] := false$;	$r[1] := false$;

Now we define

$$B_0 \triangleq r[0] \wedge \neg n[0] \wedge (\neg r[1] \vee t = 0 \vee n[1])$$
$$B_1 \triangleq r[1] \wedge \neg n[1] \wedge (\neg r[0] \vee t = 1 \vee n[0])$$

Is B_i true following the wait? Why?

Check interference freedom

$$\{B_0\}\ \langle r[1] := true; n[1] := true; \rangle\ \{B_0\}$$
$$\{B_0\}\ \langle t := 0; n[1] := false; \rangle\ \{B_0\}$$
$$\{B_0\}\ r[1] := false;\ \{B_0\}$$

Check mutual exclusion

$$\begin{aligned} & B_0 \wedge B_1 \\ = & \left(\begin{array}{c} r[0] \wedge \neg n[0] \wedge (\neg r[1] \vee t = 0 \vee n[1]) \\ \wedge\ r[1] \wedge \neg n[1] \wedge (\neg r[0] \vee t = 1 \vee n[0]) \end{array} \right) \\ = & \ r[0] \wedge \neg n[0] \wedge r[1] \wedge \neg n[1] \wedge t = 0 \wedge t = 1 \\ = & \ false \end{aligned}$$

Using spin loops

Even though the AMO property is not satisfied by

$$\neg r[1] \vee t = 0$$

because the operator is an “or” we can implement the await statements with spin loops.

One way to look at this is that you can write the spin loop in this form

---

loop {
    exit when $\neg r[1]$
    exit when $t = 0$ }
$\{\neg r[1] \vee t = 0\}$

Clearly either $\neg r[1]$ or $t = 0$ will be true upon exit from the loop.

More conventionally we can write a spin loop

---

while($r[1] \wedge t \neq 0$) /*spin*/ ;

Aside: Consider a statement $\langle \mathsf{await}(a \wedge b) \rangle$ where a and b are both written by other threads, is it safe to implement it as

---

while($\neg a \vee \neg b$) /*spin*/ ; ?

A full proof outline

Throughout this presentation I've been a bit lax.

Really one ought to create a full proof outline and check all interference conditions

init: bool $r[2] := [false, false]$ # Request flags
init: int $t := 0$ # Turn indicator ## invariant $t \in \{0, 1\}$
init bool $n[2] := [false, false]$ # Nasty spot

P_0 :	P_1 :
$\{true\}$	$\{true\}$
$\langle r[0] := true; n[0] := true; \rangle$	$\langle r[1] := true; n[1] := true; \rangle$
$\{C_0 : r[0] \wedge n[0]\}$	$\{C_1 : r[1] \wedge n[1]\}$
$\langle t := 1; n[0] := false; \rangle$	$\langle t := 0; n[1] := false; \rangle$
$\{Inv_0 : r[0] \wedge \neg n[0]\}$	$\{Inv_1 : r[1] \wedge \neg n[1]\}$
$\langle$await($\neg r[1] \vee t = 0$)$\rangle$	$\langle$await($\neg r[0] \vee t = 1$)$\rangle$
$\{B_0\}$	$\{B_1\}$
CS_0	CS_1
$r[0] := false;$	$r[1] := false;$

We need to prove 9 triples to show P_1 does not interfere with P_0's assertions.

$$\{C_0\}\ \langle r[1] := \mathit{true}; n[1] := \mathit{true};\rangle\ \{C_0\}$$
$$\{C_0\}\ \langle t := 0; n[1] := \mathit{false};\rangle\ \{C_0\}$$
$$\{C_0\}\ r[1] := \mathit{false}\ \{C_0\}$$
$$\{Inv_0\}\ \langle r[1] := \mathit{true}; n[1] := \mathit{true};\rangle\ \{Inv_0\}$$
$$\{Inv_0\}\ \langle t := 0; n[1] := \mathit{false};\rangle\ \{Inv_0\}$$
$$\{Inv_0\}\ r[1] := \mathit{false}\ \{Inv_0\}$$
$$\{B_0\}\ \langle r[1] := \mathit{true}; n[1] := \mathit{true};\rangle\ \{B_0\}$$
$$\{B_0\}\ \langle t := 0; n[1] := \mathit{false};\rangle\ \{B_0\}$$
$$\{B_0\}\ r[1] := \mathit{false}\ \{B_0\}$$

Since C_0 and D_0 do not refer to variables that are modified by P_1 the first 6 are trivial by *disjoint variables*.

The not quite Peterson's algorithm

Peterson's paper claims that his algorithm is a simple combination of safe-sluice and after-you.

But it is perhaps not as simple as he said.

If you split the atomic assignment $\langle r[0] := \mathit{true}; t := 1; \rangle$ the other way to get

$$t := 1; r[0] := \mathit{true};$$

then the algorithm does not enforce mutual exclusion.

Exercise: Find a sequence of actions that lets both threads into their critical sections.

Exercise: Suppose we attempt proof of this algorithm with the same strategy and the same B_i. Find the interference:

$$\{B_0\}\ \langle t := 0; n[1] := \mathit{true}; \rangle\ \{B_0\}$$
$$\{B_0\}\ \langle r[1] := \mathit{true}; n[1] := \mathit{false}; \rangle\ \{B_0\}$$
$$\{B_0\}\ r[1] := \mathit{false};\ \{B_0\}$$

Exercise: Extend the proof to the N-way Peterson's algorithm.

The slides leading up to this one are based on work by Peterson (of course), Dijkstra, Feijn and Bijlsma, and van de Snepsheut.