Lecture 4: Mutex with only atomic reads and writes (Impractical, but help understand concurrent programs)

K. V. S. Prasad

TDA384/DIT391 Principles of Concurrent Programming
Chalmers Univ. and Univ. of Gothenburg

12 September 2019

Hardware atomic actions

Concurrency means and *communicating processes*. Communicating what?

- *shared resources* (including data)
- *Synchronisation* (timing signals)

In the 1950's, I/O devices were run in parallel with the CPU, and the *mutex* access to the shared buffer was managed by *timing*.

But this is delicate. A robust solution would use explicit synchronisation:

1 atomic⟨if resource free then grab resource⟩;
  //*atomic* prevents other processes from stealing the resource
  //between the if test and the then action.

2 release resource

Now CPU instructions are typically atomic: they execute fully or not at all.
How do we make larger sections of code (like line 1 above) atomic ?

Mutex with only atomic reads and writes (Impractical, but help understand concurrent programs)

In the 1960's, hardware instructions like *test-and-set* were introduced

- to create such larger atomic sections of code
- and to do this in software via primitives like *locks* and *semaphores*

But some curious questions bothered people:

- Do we really need packaged instructions like *test-and-set*?
- Could (atomic) read and write be enough?

Surprisingly, the answer to the second question is *yes*!

Today, we see one such solution, Peterson's algorithm, after first looking at simpler attempts.

- Such algorithms are not practical solutions to the CS problem
- But they are excellent to help understand concurrent programs

CS problem for two processes, with one *turn*

Reminder: we require that the program satisfy

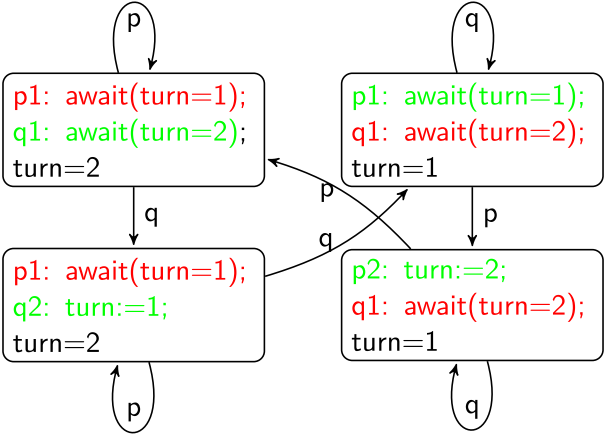
- *mutex property*: if *p* is at p2 (abbr. "p2"), then $\neg$ q2
- *deadlock free*: p1 $\wedge$ q1 $\rightarrow$ p and q will not both be stuck waiting
  (i.e., p or q will progress to CS)
- *starvation free*: p1 $\rightarrow$ p will progress to CS

int *turn* := 1			
process p		process q	
while *true* {		while *true* {	
	//NCS;		//NCS;
p1:	*await(turn=1)*;	q1:	*await(turn=2)*;
	//CS;		//CS;
p2:	*turn:=2*;	q2:	*turn:=1*;
	};		};

await(turn=2) is a *busy loop* of atomic *while (turn!=2);*

//while *turn* is not 2, do nothing

State diagram, of CS program with one *turn*



- No state has p2 and q2. So mutex.
- No state has both processes in red (blocked). Also, every state has an exit arrow. So no deadlock.
- Both p1 and q1 can also loop in the NCS before starting await. So if p1 is blocked, and q1 is looping in its NCS, then p is starved.

Detour through the Sandro/Carlo slides 20 - 30 for naive and Peterson's algorithm, and slides 52-53 for test-and-set
Then back here.

CS problem with *swap*

Suppose *swap(x,y)* atomically interchanges the values of *x* and *y*.

int c := 1	
process p	process q
int l := 0;	int l := 0;
while *true* {//NCS;	while *true* {//NCS;
p1: while (l=0) {	q1: while (l=0) {
p2: *swap(c, l)*;	q2: *swap(c, l)*;
};//CS;	}; //CS;
p3: *swap(c,l)*;};	q2: *swap(c,l)*;};

- Suppose we refer to the *l* of *p* as *lp* and to the *l* of *q* as *lq*. Invariant: Exactly one of *c, lp, lq* is 1; the others are 0. Therefore mutex. The process in its CS has its l=1.
- One process has to get the token, and won't give it back until after the CS. So no deadlock.
- Can starve if *p* only swaps when *c*=0, but would have to be very unlucky. Fair but consistently badly synched scheduler.

The bakery algorithm

int np := 0; int nq := 0	
process p	process q
while *true* {NCS; p1: $np := nq+1$; p2: await (nq=0 or $np \leq nq$); p3: CS; np:=0;};	while *true* {NCS; q1: $nq := np+1$; q2: await (np=0 or $nq < np$); q3: CS; nq:=0;};

Note the asymmetry in p2 and q2.

Invariants: np=0 iff p1, and p3 $\rightarrow$ C, where C=(nq=0) or ($np \leq nq$).

Also, nq=0 iff q1, and q3 $\rightarrow$ D, where D=(np=0) or ($np < nq$).

- The p1 and q1 invariants are trivial.
- The second is true at init.
  Suppose $\neg$p3 and $\neg$C. If now p3 becomes true, it must be by executing p2, so C will become true too.
  Suppose p3 and C. Can we reach p3 and $\neg$C? Then only *q* can act, at q2 or q5, and both make C true.

Mutex for the bakery algorithm

We had

Invariants: np=0 iff p1, and p3 $\rightarrow$ C, where C=(nq=0) or ($np \leq nq$).
Also, nq=0 iff q1, and q3 $\rightarrow$ D, where D=(np=0) or ($nq < np$).

Can p3$\wedge$q3 be true?

If p3$\wedge$q3, then from the p1 and q1 invariants, np$\neq 0 \wedge$nq$\neq 0$.

Then from the p3 and q3 invariants, p3$\wedge$q3$\rightarrow$($np \leq nq$)$\wedge$($np < nq$).

From this contradiction, it follows that p3$\wedge$q3 cannot be true. Mutex.