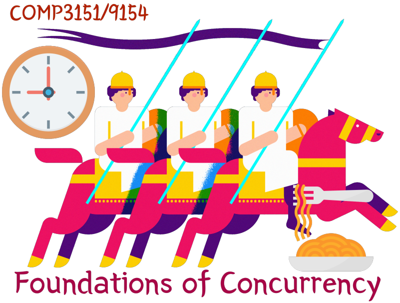

Some More Critical Section Solutions

Johannes Åman Pohjola
CSE, UNSW
Term 2 2022

Where we are at

We've discussed the critical section problem, the four properties of critical section solutions, and some solutions for two processes.

Where we are at

We've discussed the critical section problem, the four properties of critical section solutions, and some solutions for two processes.

In this lecture, we will see some of the classic critical section solutions for n processes.

More liveness desiderata:

- **Eventual Entry** (or *starvation-freedom*) Once a process enters its pre-protocol, it will eventually be able to execute its critical section.

More liveness desiderata:

- **Eventual Entry** (or *starvation-freedom*) Once a process enters its pre-protocol, it will eventually be able to execute its critical section.
- **Bounded waiting** Once a process enters its pre-protocol, it can be *bypassed* by other processes at most $f(n)$ times for some f. (n is the number of processes)

More liveness desiderata:

- **Eventual Entry** (or *starvation-freedom*) Once a process enters its pre-protocol, it will eventually be able to execute its critical section.
- **Bounded waiting** Once a process enters its pre-protocol, it can be *bypassed* by other processes at most $f(n)$ times for some f. (n is the number of processes)
- **Linear waiting** No process can enter its critical section twice while another process is in its pre-protocol.

More liveness desiderata:

- **Eventual Entry** (or *starvation-freedom*) Once a process enters its pre-protocol, it will eventually be able to execute its critical section.
- **Bounded waiting** Once a process enters its pre-protocol, it can be *bypassed* by other processes at most $f(n)$ times for some f. (n is the number of processes)
- **Linear waiting** No process can enter its critical section twice while another process is in its pre-protocol.

Question

Which of the above are linear temporal properties?

From 2 to n Processes

In the 5th attempt of lecture 2 (a.k.a. Dekker's Algorithm) we used a shared variable `turn` to remember whose turn it would be to enter the CS in case of contention.
This turns out to be simple for 2 processes but complex for n.

Tie-Breaker (Peterson's) Algorithm for 2 Processes

Algorithm 1.1: Peterson's algorithm		
boolean wantp ← false, wantq ← false integer last ← 1		
p		**q**
forever do		**forever do**
p1:	*non-critical section*	q1: *non-critical section*
p2:	wantp ← true	q2: wantq ← true
p3:	last ← 1	q3: last ← 2
p4:	**await** wantq = false or last ≠ 1	q4: **await** wantp = false or last ≠ 2
p5:	*critical section*	q5: *critical section*
p6:	wantp ← false	q6: wantq ← false

Tie-Breaker Code for n Processes

Algorithm 1.2: Peterson's algorithm (*n* processes, process *i*)

```
            integer array in[1..n] ← [0,. . . ,0]
            integer array last[1..n] ← [0,. . . ,0]

    forever do
p1:     non-critical section
        for all j ∈ {1..n − 1}
p2:         in[i] ← j
p3:         last[j] ← i
            for all processes k ≠ i
p4:             await in[k] < j or last[j] ≠ i
p5:     critical section
p6:     in[i] ← 0
```

Properties of the Tie-Breaker Algorithm

Do we satisfy:

- Eventual entry?
- Bounded waiting?
- Linear waiting?

Properties of the Tie-Breaker Algorithm

Do we satisfy:

- Eventual entry?
- Bounded waiting?
- Linear waiting?

Literature review

In "Some Myths about Famous Mutual Exclusion Algorithms" by Alagarsamy (2003), it is pointed out that the n-process variant does not ensure bounded waiting. We can use Promela to check that eventual entry holds (assuming weak fairness, fixing a small n), and that linear wait fails.

Algorithm 1.3: Simplified bakery algorithm (two processes)	
integer np $\leftarrow$ 0, nq $\leftarrow$ 0	
p	**q**
forever do	**forever do**
p1: *non-critical section*	q1: *non-critical section*
p2: np $\leftarrow$ nq + 1	q2: nq $\leftarrow$ np + 1
p3: **await** nq = 0 or np $\leq$ nq	q3: **await** np = 0 or nq < np
p4: *critical section*	q4: *critical section*
p5: np $\leftarrow$ 0	q5: nq $\leftarrow$ 0

Note the asymmetry here! Why do we need it?

What if we don't have atomicity for each statement?

Mutual Exclusion

The following are invariants

$$np = 0 \Leftrightarrow P@p1..2 \tag{1}$$

$$nq = 0 \Leftrightarrow Q@q1..2 \tag{2}$$

$$P@p4 \Rightarrow nq = 0 \vee np \leq nq \tag{3}$$

$$Q@q4 \Rightarrow np = 0 \vee nq < np \tag{4}$$

and hence also $\neg(P@p4 \wedge Q@q4)$.

Mutual Exclusion

The following are invariants

$$np = 0 \Leftrightarrow P@p1..2 \quad (1)$$

$$nq = 0 \Leftrightarrow Q@q1..2 \quad (2)$$

$$P@p4 \Rightarrow nq = 0 \lor np \leq nq \quad (3)$$

$$Q@q4 \Rightarrow np = 0 \lor nq < np \quad (4)$$

and hence also $\neg(P@p4 \land Q@q4)$.

Other Safety Properties

Deadlock freedom: The disjunction $nq = 0 \vee np \leq nq \vee np = 0 \vee nq < np$ of the conditions on the **await** statements at p3/q3 is equivalent to $\top$. Hence it is not possible for both processes to be blocked there.

Other Safety Properties

Deadlock freedom: The disjunction $nq = 0 \vee np \leq nq \vee np = 0 \vee nq < np$ of the conditions on the **await** statements at p3/q3 is equivalent to $\top$. Hence it is not possible for both processes to be blocked there.

Absence of unnecessary delay: Even if one process prefers to stay in its non-critical section, no deadlock will occur by the first two invariants (1) and (2).

Eventual Entry

For p to fail to reach its CS despite wanting to, it needs to be stuck at p3 where it will evaluate the condition infinitely often by weak fairness. To remain stuck, each of these evaluations must yield false. In LTL:

$$\Box\Diamond\neg(nq = 0 \vee np \leq nq)$$

which implies

$$\Box\Diamond nq \neq 0 \text{ , and} \tag{5}$$

$$\Box\Diamond nq < np \text{ .} \tag{6}$$

Eventual Entry

For p to fail to reach its CS despite wanting to, it needs to be stuck at p3 where it will evaluate the condition infinitely often by weak fairness. To remain stuck, each of these evaluations must yield false. In LTL:

$$\Box\Diamond\neg(nq = 0 \lor np \leq nq)$$

which implies

$$\Box\Diamond nq \neq 0 \quad \text{, and} \tag{5}$$

$$\Box\Diamond nq < np \quad . \tag{6}$$

Because there is no deadlock, (5) implies that process q goes through infinitely many iterations of the main loop without getting lost in the non-critical section.

Eventual Entry

For p to fail to reach its CS despite wanting to, it needs to be stuck at p3 where it will evaluate the condition infinitely often by weak fairness. To remain stuck, each of these evaluations must yield false. In LTL:

$$\Box\Diamond\neg(nq = 0 \vee np \leq nq)$$

which implies

$$\Box\Diamond nq \neq 0 \text{ , and} \tag{5}$$

$$\Box\Diamond nq < np \text{ .} \tag{6}$$

Because there is no deadlock, (5) implies that process q goes through infinitely many iterations of the main loop without getting lost in the non-critical section. But then it must set nq to the constant $np + 1$. From then onwards it is no longer possible to fail the test $(nq = 0 \vee np \leq nq)$, contradiction.

$2 \rightarrow n$

Algorithm 1.4: Simplified bakery algorithm (N processes)
integer array[1..n] number ← [0,...,0]
loop forever
p1: non-critical section
p2: number[i] ← max(number) + 1
p3: for all *other* processes j
p4: await (number[j] = 0) or (number[i] ≪ number[j])
p5: critical section
p6: number[i] ← 0

once again relying on atomicity of non-LCR lines of Ben-Ari pseudo-code; ≪ breaks ties using PIDs:

$$a[i] \ll a[j] \quad \Leftrightarrow \quad (a[i] < a[j]) \vee (a[i] = a[j] \wedge i < j)$$

An Implementable Algorithm

Algorithm 1.5: Lamport's bakery algorithm	
	boolean array[1..n] choosing $\leftarrow$ [false,. . . ,false]
	integer array[1..n] number $\leftarrow$ [0,. . . ,0]
	forever do
p1:	*non-critical section*
p2:	choosing[i] $\leftarrow$ true
p3:	number[i] $\leftarrow 1 +$ max(number)
p4:	choosing[i] $\leftarrow$ false
p5:	**for** all *other* processes j
p6:	**await** choosing[j] $=$ false
p7:	**await** (number[j] $=$ 0) or (number[i] $\ll$ number[j])
p8:	*critical section*
p9:	number[i] $\leftarrow$ 0

Properties of Lamport's bakery algorithm

"The algorithm has the remarkable property that if a read and a write operation to a single memory location occur simultaneously, then only the write operation must be performed correctly. The read may return any arbitrary value!"

Lamport, 1974 (CACM)

Cons:

$\mathcal{O}(n)$ pre-protocol; unbounded ticket numbers

Assertion 1:

If $P_k @ p1..2 \wedge P_i @ p5..9$ and k then reaches $p5..9$ while i is still there, then $\text{number}[i] < \text{number}[k]$

Assertion 2:

$$P_i @ p8..9 \wedge P_k @ p5..9 \wedge i \neq k \Rightarrow (\text{number}[i], i) \ll (\text{number}[k], k)$$

When contention is low...

access to the CS should be fast, that is, consist of a fixed number of steps (aka $\mathcal{O}(1)$) with no **awaits**.

Almost correct fast solution

Algorithm 1.6: Fast algorithm for two processes (outline)	
integer gate1 ← 0, gate2 ← 0	
p	**q**
forever do	**forever do**
non-critical section	*non-critical section*
p1: gate1 ← p	q1: gate1 ← q
p2: **if** gate2 ≠ 0 **goto** p1	q2: **if** gate2 ≠ 0 **goto** q1
p3: gate2 ← p	q3: gate2 ← q
p4: **if** gate1 ≠ p	q4: **if** gate1 ≠ q
p5: **if** gate2 ≠ p **goto** p1	q5: **if** gate2 ≠ q **goto** q1
critical section	*critical section*
p6: gate2 ← 0	q6: gate2 ← 0

Invariants

$$P@p5 \wedge \text{gate2} = p \Rightarrow \neg(Q@q3 \vee Q@q4 \vee Q@q6) \quad (7)$$

$$Q@q5 \wedge \text{gate2} = q \Rightarrow \neg(P@p3 \vee P@p4 \vee P@p6) \quad (8)$$

$$P@p4 \wedge \text{gate1} = p \Rightarrow \text{gate2} \neq 0 \quad (9)$$

$$P@p6 \Rightarrow \text{gate2} \neq 0 \wedge \neg Q@q6 \wedge (Q@q3 \vee Q@q4 \Rightarrow \text{gate1} \neq q) \quad (10)$$

$$Q@q4 \wedge \text{gate1} = q \Rightarrow \text{gate2} \neq 0 \quad (11)$$

$$Q@q6 \Rightarrow \text{gate2} \neq 0 \wedge \neg P@p6 \wedge (P@p3 \vee P@p4 \Rightarrow \text{gate1} \neq p) \quad (12)$$

Mutual exclusion follows from invariants (10) and (12).

Invariants

$$P@p5 \wedge \text{gate2} = p \Rightarrow \neg(Q@q3 \vee Q@q4 \vee Q@q6) \quad (7)$$

$$Q@q5 \wedge \text{gate2} = q \Rightarrow \neg(P@p3 \vee P@p4 \vee P@p6) \quad (8)$$

$$P@p4 \wedge \text{gate1} = p \Rightarrow \text{gate2} \neq 0 \quad (9)$$

$$P@p6 \Rightarrow \text{gate2} \neq 0 \wedge \neg Q@q6 \wedge (Q@q3 \vee Q@q4 \Rightarrow \text{gate1} \neq q) \quad (10)$$

$$Q@q4 \wedge \text{gate1} = q \Rightarrow \text{gate2} \neq 0 \quad (11)$$

$$Q@q6 \Rightarrow \text{gate2} \neq 0 \wedge \neg P@p6 \wedge (P@p3 \vee P@p4 \Rightarrow \text{gate1} \neq p) \quad (12)$$

Mutual exclusion follows from invariants (10) and (12).

Problem: (7) and (8) aren't actually invariants of this algorithm.

Algorithm 1.7: Fast algorithm for two processes	
integer gate1 ← 0, gate2 ← 0 boolean wantp ← false, wantq ← false	
p	**q**
p1: gate1 ← p wantp ← true	q1: gate1 ← q wantq ← true
p2: **if** gate2 ≠ 0 wantp ← false **goto** p1	q2: **if** gate2 ≠ 0 wantq ← false **goto** q1
p3: gate2 ← p	q3: gate2 ← q
p4: **if** gate1 ≠ p wantp ← false **await** wantq = false	q4: **if** gate1 ≠ q wantq ← false **await** wantp = false
p5: **if** gate2 ≠ p **goto** p1 **else** wantp ← true	q5: **if** gate2 ≠ q **goto** q1 **else** wantq ← true
critical section	*critical section*
p6: gate2 ← 0 wantp ← false	q6: gate2 ← 0 wantq ← false

Mutex review

None of the mutual exclusion algorithms presented so far scores full marks.

Selected problems:

- have a $\mathcal{O}(n^2)$ pre-protocol (Peterson)
- rely on special instruction (e.g. xc, ts, etc.)
- use unbounded ticket numbers (e.g. bakery)
- sacrifice eventual entry (e.g. fast)

Szymanski's Algorithm

- has none of these problems,
- enforces *linear wait*,
- requires at most $4p - \lceil \frac{p}{n} \rceil$ writes for p CS entries by n competing processes, and
- can be made immune to process failures and restarts as well as read errors occurring during writes.

How does he do it?

Idea

"The prologue is modeled after a waiting room with two doors. [...] All processes requesting entry to the CS at roughly the same time gather first in the waiting room. Then, when there are no more processes requesting entry, waiting processes move to the end of the prologue. From there, one by one, they enter their CS. Any other process requesting entry to its CS at that time has to wait in the initial part of the prologue (before the waiting room)." Szymanski, 1988, in ICCS

Phases of the pre-protocol

1. announce intention to enter CS
2. enter waiting room through door 1; wait there for other processes
3. last to enter the waiting room closes door 1
4. in the order of PIDs, leave waiting room through door 2 to enter CS

Shared variables

Each process `i` exclusively writes a variable called `flag`, which is read by all the other processes. It assumes one of five values:

- **0** denoting that `i` is in its non-CS,
- **1** declares `i`'s intention to enter the CS
- **2** shows that `i` waits for other processes to enter the waiting room
- **3** denotes that `i` has just entered the waiting room
- **4** indicates that `i` left the waiting room

Algorithm 1.8: Szymanski's algorithm (*n* processes, process *i*)

```
              integer array flag[1..n] ← [0,. . . ,0]

     forever do
p1:      non-critical section
p2:      flag[i]:=1
p3:      await ∀j. flag[j] <3
p4:      flag[i]:=3
p5:      if ∃j. flag[j] = 1 then
p6:          flag[i]:=2
p7:          await ∃j. flag[j]=4
p8:      flag[i]:=4
p9:      await ∀j<i. flag[j] <2
p10:     critical section
p11:     await ∀j>i. flag[j] <2 or flag[j] >3
p12:     flag[i]:=0
```

How to implement the atomic tests

The atomic tests can be implemented by loops. The order of the tests is crucial for the mutual exclusion property. But which order? Szymanski's original paper is unclear on the matter.

See Promela Code samples (and your homework ;).

How to prove mutual exclusion

This is reasonably hard. So hard indeed that even Turing Award winners (Manna and Pnueli) published about solving the problem (with non-atomic tests), using the "one big invariant" method.
See the de Roever book pp.157–164 for a proof using the Owicki-Gries method on (parameterized) transition diagrams (with atomic tests).
What is hard about the proof? Finding the assertions.

What now?

- You should be making progress on Assignment 0 (due Monday) and Homework 2 (due Friday).
- You can (soon) find Promela code on the website for most of the algos discussed today.
- New questions about critical sections will be up soon, due Friday next week.