Concepts of Concurrent Computation
Spring 2015
Lecture 3: Synchronisation Algorithms

Sebastian Nanz
Chris Poskitt

Chair of Software Engineering
ETH zürich

It's easy to make mistakes

- concurrent threads often share resources
- we want to avoid race conditions
- can be avoided through locks, but:
  => *non-trivial*
  => *may introduce deadlock, starvation, ...*

Solutions, problems, and more solutions

- many early attempts to solve the problem of exclusive resource access

  => *many proposed solutions still had deficiencies*

- we will study some of these classical synchronisation algorithms

  => *learn from their shortcomings to better understand the problem*

Today's lecture

- define the mutual exclusion problem
  => *common framework for evaluating solutions to the problem of common resource access*

- consider some solutions to the problem and their properties

- apply techniques for proving properties of such solutions

Next on the agenda

1. mutual exclusion problem
2. towards a solution
3. Peterson’s algorithm
4. Bakery algorithm

Mutual exclusion

- mutual exclusion is a form of synchronisation used to avoid the simultaneous use of a shared resource

- the part of a program that accesses a shared resource is called a critical section

Mutual exclusion

- mutual exclusion is a form of synchronisation used to avoid the simultaneous use of a shared resource

- the part of a program that accesses a shared resource is called a critical section

```
while true loop
    entry protocol
    critical section
    exit protocol
    non-critical section
end
```

Mutual exclusion

- mutual exclusion is a form of synchronisation used to avoid the simultaneous use of a shared resource

- the part of a program that accesses a shared resource is called a critical section

```
while true loop
    entry protocol
    critical section
    exit protocol
    non-critical section
end
```

mutual exclusion problem concerns getting these right

Mutual exclusion problem

- given *n* processes of the form:

```
while true loop
    entry protocol
    critical section
    exit protocol
    non-critical section
end
```

- design entry and exit protocols to ensure:

1. mutual exclusion

2. freedom from deadlock

3. freedom from starvation

Mutual exclusion problem

- given *n* processes of the form:

```
while true loop
    entry protocol
    critical section
    exit protocol
    non-critical section
end
```

- design entry and exit protocols to ensure:

1. mutual exclusion

   at most one process ever in its critical section

2. freedom from deadlock

3. freedom from starvation

Mutual exclusion problem

- given *n* processes of the form:

```
while true loop
    entry protocol
    critical section
    exit protocol
    non-critical section
end
```

- design entry and exit protocols to ensure:

1. mutual exclusion

   at most one process ever in its critical section

2. freedom from deadlock

   if more than one process attempting to enter their critical sections, one will eventually succeed

3. freedom from starvation

Mutual exclusion problem

- given *n* processes of the form:

```
while true loop
    entry protocol
    critical section
    exit protocol
    non-critical section
end
```

- design entry and exit protocols to ensure:

1. mutual exclusion

   at most one process ever in its critical section

2. freedom from deadlock

   if more than one process attempting to enter their critical sections, one will eventually succeed

3. freedom from starvation

   if a process is trying to enter its critical section, it will eventually succeed

Assumptions and considerations

- processes communicate only via atomic (i.e. indivisible) steps
- assume that if a process enters its critical section, it will eventually exit from it
- a process could terminate (or loop forever) in its non-critical section
- shared resources will not be accessed outside of these processes

Locks

- synchronisation mechanisms based on the ideas of entry- and exit-protocols are called locks

- typically implemented as a pair of functions:

```
lock
  do
    entry protocol
  end
```

```
unlock
  do
    exit protocol
  end
```

Next on the agenda

1. mutual exclusion problem ✓

2. towards a solution

3. Peterson's algorithm

4. Bakery algorithm

Towards a solution

- the mutual exclusion problem is deceptively tricky, and took a while to become well-understood

- many incorrect solutions published in the 1960s
  => *we will work along a series of failing attempts*
  until establishing a solution

- first, restrict ourselves to two processes (n = *2*)

Brief aside: busy waiting

- we will use the following pseudocode:

```
await b
```

which is equivalent to:

```
while not b loop end
```

Brief aside: busy waiting

- we will use the following pseudocode:

```
await b
```

which is equivalent to:

```
while not b loop end
```

busy waiting

Brief aside: busy waiting

- we will use the following pseudocode:

```
await b
```

which is equivalent to:

```
while not b loop end
```

busy waiting

inefficient in multitasking systems
...but makes sense if waiting times shorter than context switching

Solution attempt no. 1

- idea: use two variables enter1 and enter2; if enter*i* is true, it means that process P*i* intends to enter its critical section

`enter1 := false` `enter2 := false`	
P1	P2

Solution attempt no. 1

- idea: use two variables enter1 and enter2; if enter*i* is true, it means that process P*i* intends to enter its critical section

enter1 := **false** enter2 := **false**			
P1		P2	
	while true loop		
1	**await not** enter2	1	
2	enter1 := **true**	2	
3	critical section	3	
4	enter1 := **false**	4	
5	non-critical section	5	
	end		

Solution attempt no. 1

- idea: use two variables enter1 and enter2; if enter*i* is true, it means that process P*i* intends to enter its critical section

```
enter1 := false
enter2 := false
```

P1

```
while true loop
    await not enter2
    enter1 := true
    critical section
    enter1 := false
    non-critical section
end
```

P2

```
while true loop
    await not enter1
    enter2 := true
    critical section
    enter2 := false
    non-critical section
end
```

Solution attempt no. 1

- idea: use two variables enter1 and enter2; if enter*i* is true, it means that process P*i* intends to enter its critical section

Shared:

```
enter1 := false
enter2 := false
```

P1

```
while true loop
    await not enter2
    enter1 := true
    critical section
    enter1 := false
    non-critical section
end
```

P2

```
while true loop
    await not enter1
    enter2 := true
    critical section
    enter2 := false
    non-critical section
end
```

incorrect: does not enforce mutual exclusion

Solution attempt no. 1 is incorrect

- the two processes can end up in their critical sections at the same time:

P2	1	**await not** enter1
P1	1	**await not** enter2
P1	2	enter1 := **true**
P2	2	enter2 := **true**
P2	3	critical section
P1	3	critical section

Solution attempt no. 1 is <u>incorrect</u>

- the two processes can end up in their critical sections at the same time:

P2	1	**await not** enter1
P1	1	**await not** enter2
P1	2	enter1 := **true**
P2	2	enter2 := **true**
P2	3	critical section
P1	3	critical section

the "awaits" guard the critical sections!
perhaps set enter1 and enter2 before?

Solution attempt no. 2

```
enter1 := false
enter2 := false
```

P1

```
while true loop
    enter1 := true
    await not enter2
    critical section
    enter1 := false
    non-critical section
end
```

P2

```
while true loop
    enter2 := true
    await not enter1
    critical section
    enter2 := false
    non-critical section
end
```

Solution attempt no. 2

enter1 := **false**	
enter2 := **false**	
P1	P2
while true loop	**while true loop**
enter1 := **true**	enter2 := **true**
await not enter2	**await not** enter1
critical section	critical section
enter1 := **false**	enter2 := **false**
non-critical section	non-critical section
end	**end**

mutual exclusion?

Solution attempt no. 2

```
enter1 := false
enter2 := false
```

P1

```
while true loop
    enter1 := true
    await not enter2
    critical section
    enter1 := false
    non-critical section
end
```

P2

```
while true loop
    enter2 := true
    await not enter1
    critical section
    enter2 := false
    non-critical section
end
```

mutual exclusion?

Solution attempt no. 2

`enter1 := **false**` `enter2 := **false**`			
P1		P2	
	while true loop		**while true loop**
1	enter1 := **true**	1	enter2 := **true**
2	**await not** enter2	2	**await not** enter1
3	critical section	3	critical section
4	enter1 := **false**	4	enter2 := **false**
5	non-critical section	5	non-critical section
	end		**end**

mutual exclusion?  *freedom from deadlock?*

Solution attempt no. 2

```
enter1 := false
enter2 := false
```

P1	P2
while true loop	**while true loop**
enter1 := **true**	enter2 := **true**
await not enter2	**await not** enter1
critical section	critical section
enter1 := **false**	enter2 := **false**
non-critical section	non-critical section
end	**end**

mutual exclusion? ✓

freedom from deadlock? ✗

Solution attempt no. 2 is incorrect

- the two processes can deadlock:

P1	1	enter1 := **true**
P2	1	enter2 := **true**
P2	2	**await not** enter1
P1	2	**await not** enter2

Solution attempt no. 3

- try something different!

  namely, a single variable turn that has value *i* if it's P*i*'s turn to enter the critical section

```
turn := 1 or turn := 2
```

P1

```
while true loop
    await turn = 1
    critical section
    turn := 2
    non-critical section
end
```

P2

```
while true loop
    await turn = 2
    critical section
    turn := 1
    non-critical section
end
```

mutual exclusion?

freedom from deadlock?

Solution attempt no. 3

- try something different!

  namely, a single variable turn that has value *i* if it's P*i*'s turn to enter the critical section

```
turn := 1 or turn := 2
```

P1	P2

while true loop
    await turn = 1
    critical section
    turn := 2
    non-critical section
end
	while true loop
	    await turn = 2
	    critical section
	    turn := 1
	    non-critical section
	end

mutual exclusion? ✓ *freedom from deadlock?* ✓

Is attempt no. 3 really correct?

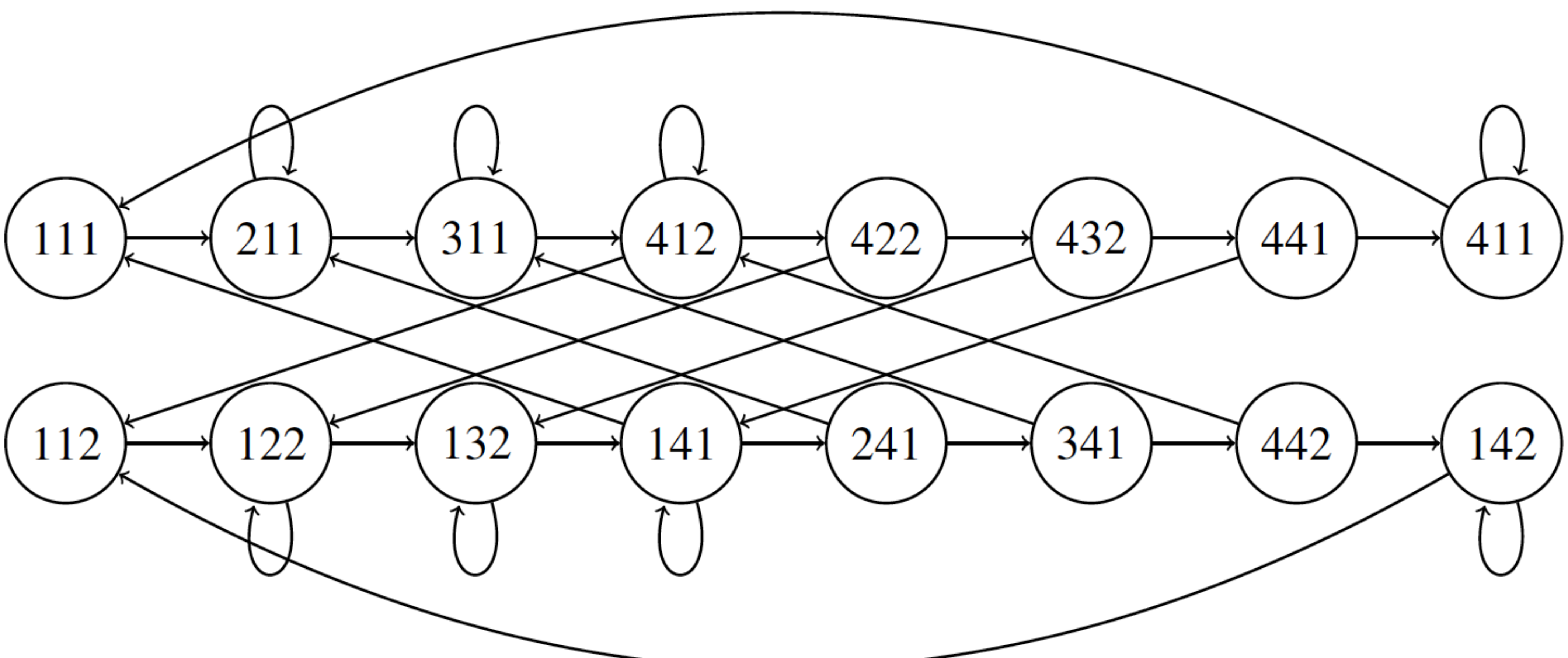
- let's try to prove it

- draw the related transition system; states are labeled with triples (i, j, k): program pointer values P1 ▻ i and P2 ▻ j, and value of the variable turn = k.

Is attempt no. 3 really correct?

- let's try to prove it

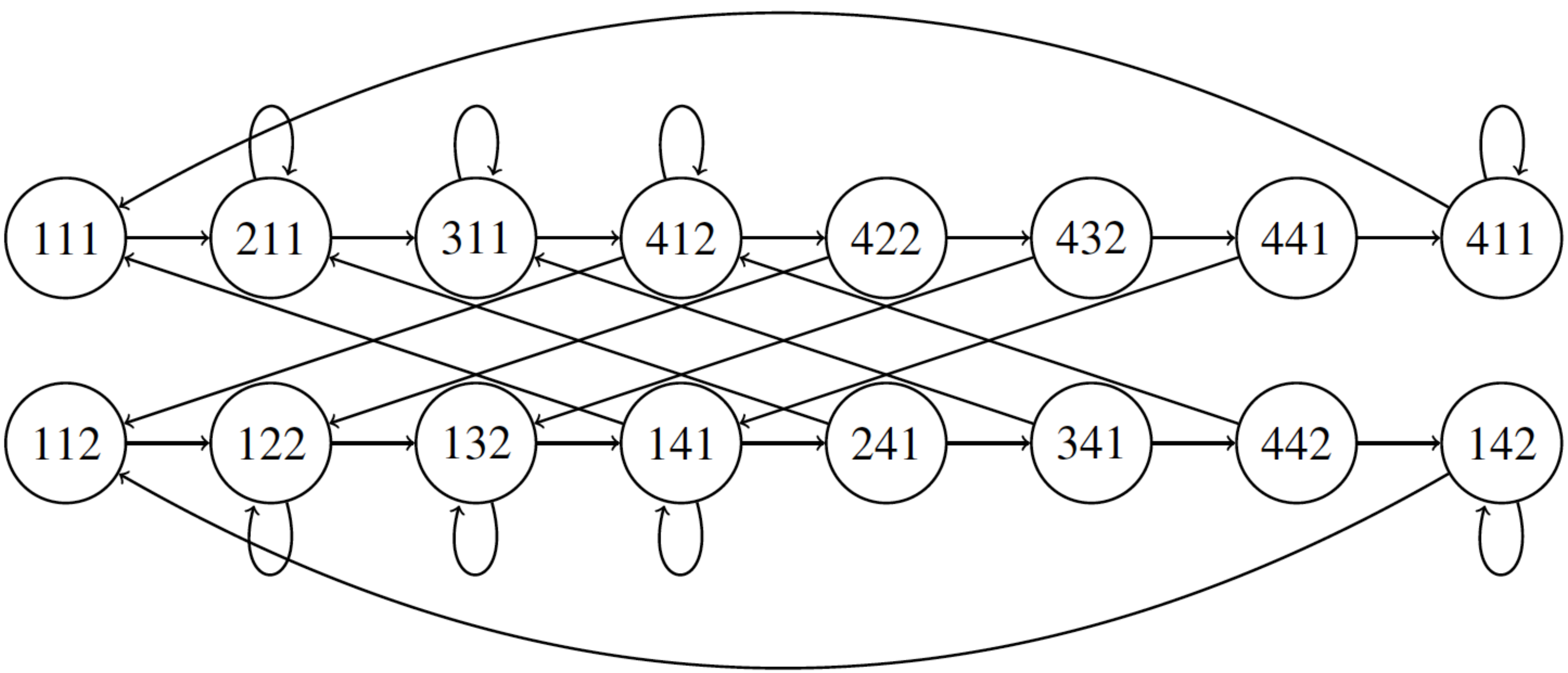
- draw the related transition system; states are labeled with triples (i, j, k): program pointer values P1 ▻ i and P2 ▻ j, and value of the variable turn = k.



Is attempt no. 3 really correct?

- solution attempt 3 satisfies mutual exclusion

  proof. Mutual exclusion expressed as LTL formula:
  G ¬(P1▻2 ∧ P2▻2)

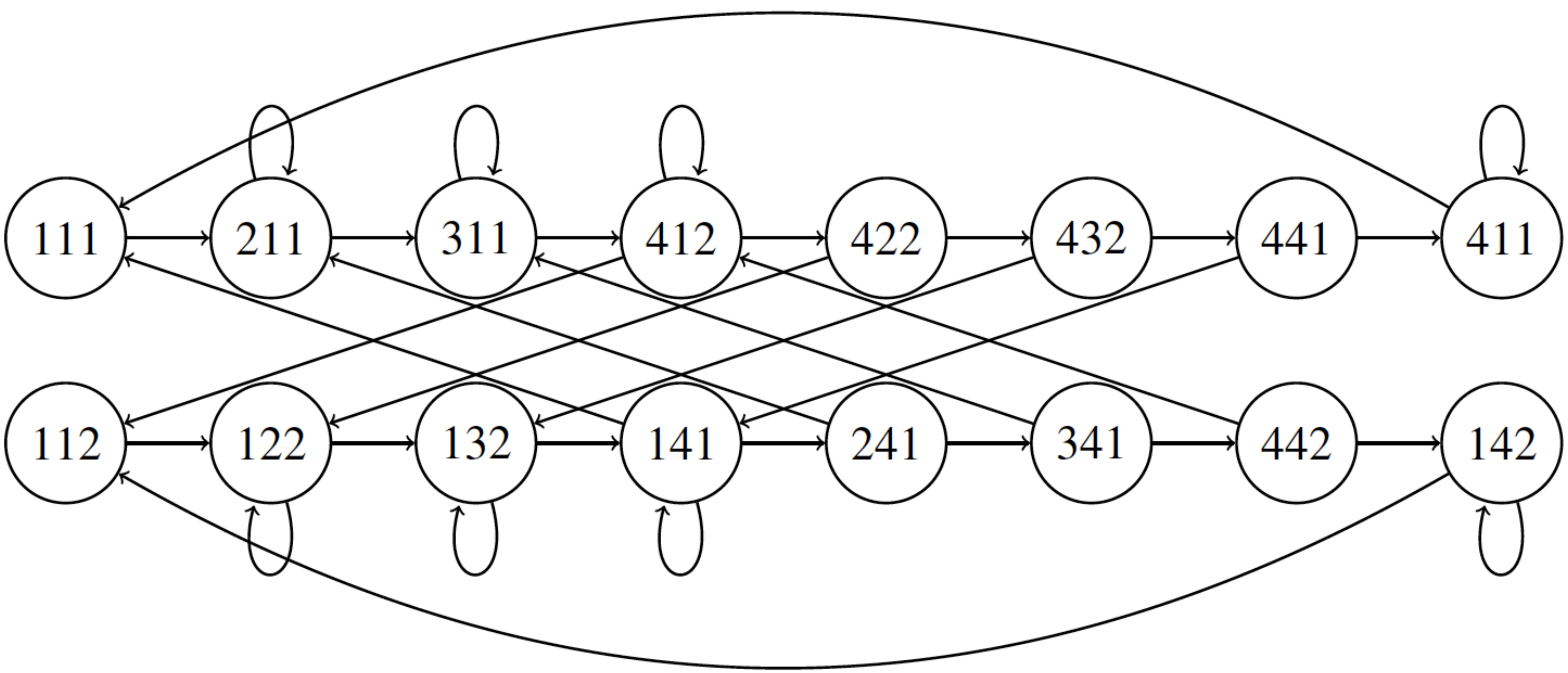


Is attempt no. 3 really correct?

- solution attempt 3 satisfies mutual exclusion

  proof. Mutual exclusion expressed as LTL formula:
  G ¬(P1▻2 ∧ P2▻2)

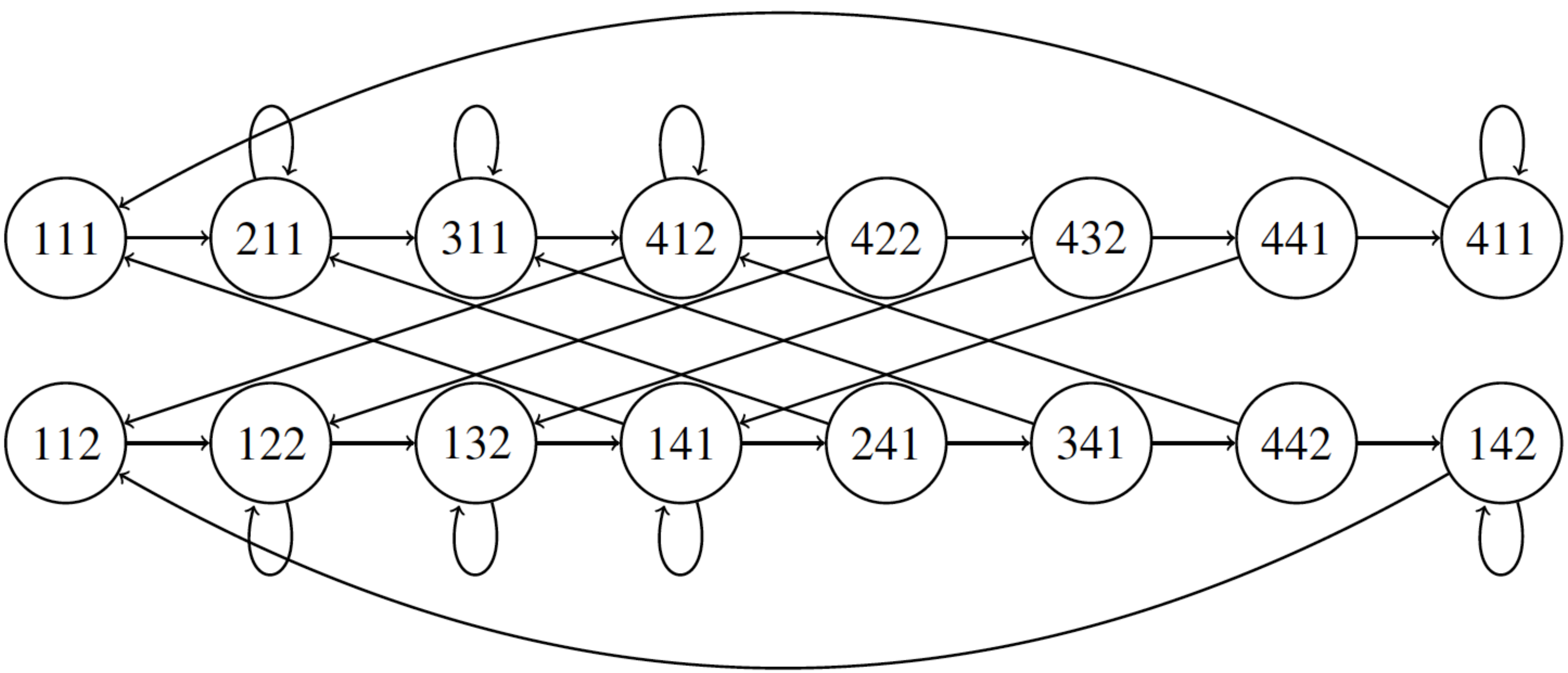
Easy to see that this formula holds, as there are no states of the form (2, 2, k).



Is attempt no. 3 <u>really</u> correct?

- solution attempt 3 is free of deadlock

  proof. Deadlock freedom expressed as LTL formula:
  G ((P1 ▻ 1 ∧ P2 ▻ 1) -> **F** (P1 ▻ 2 ∨ P2 ▻ 2))

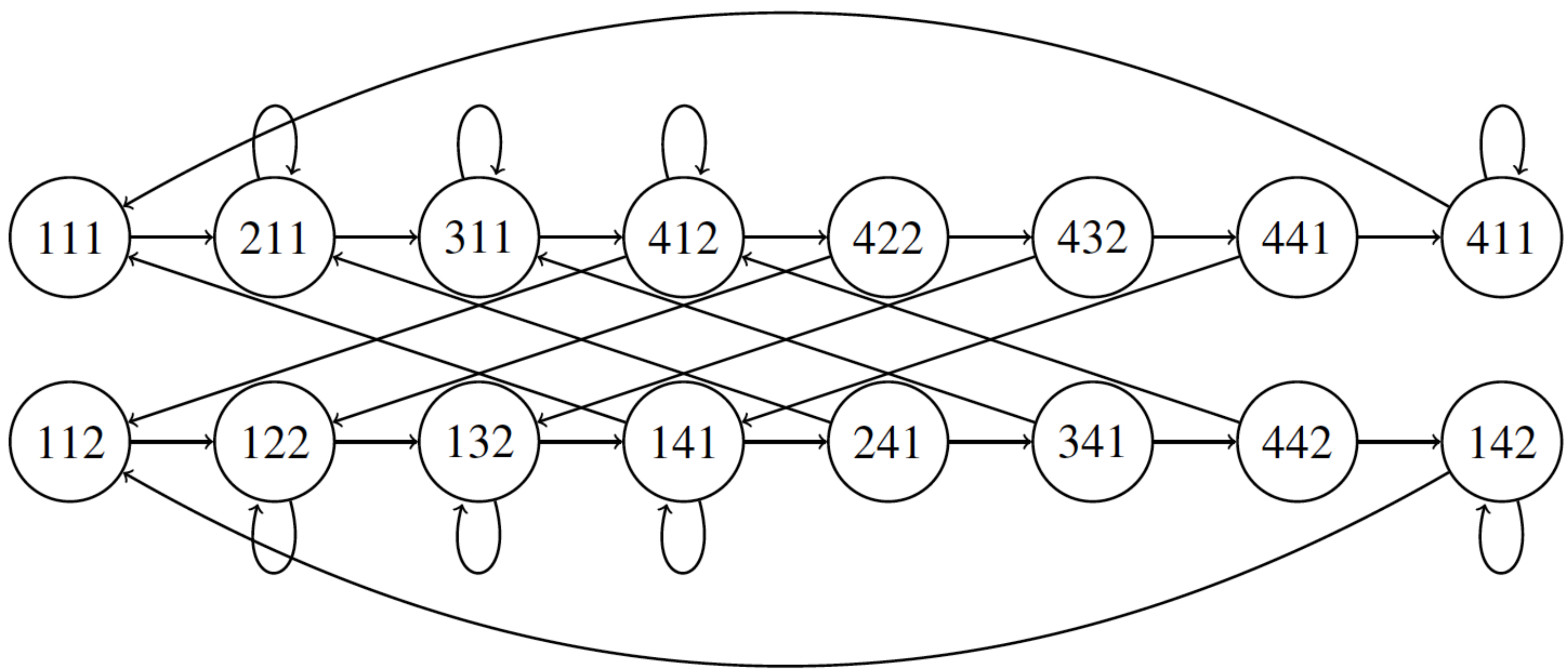


Is attempt no. 3 really correct?

- solution attempt 3 is free of deadlock

  proof. Deadlock freedom expressed as LTL formula:
  G ((P1 ⊳ 1 ∧ P2 ⊳ 1) -> **F** (P1 ⊳ 2 ∨ P2 ⊳ 2))

We have to examine the states (1, 1, 1) and (1, 1, 2); in both cases, one of the processes is able to enter its critical section.



Is attempt no. 3 really correct?

- finally, what about freedom from starvation?

  Expressed as LTL formula (for *i* = *1,2*):

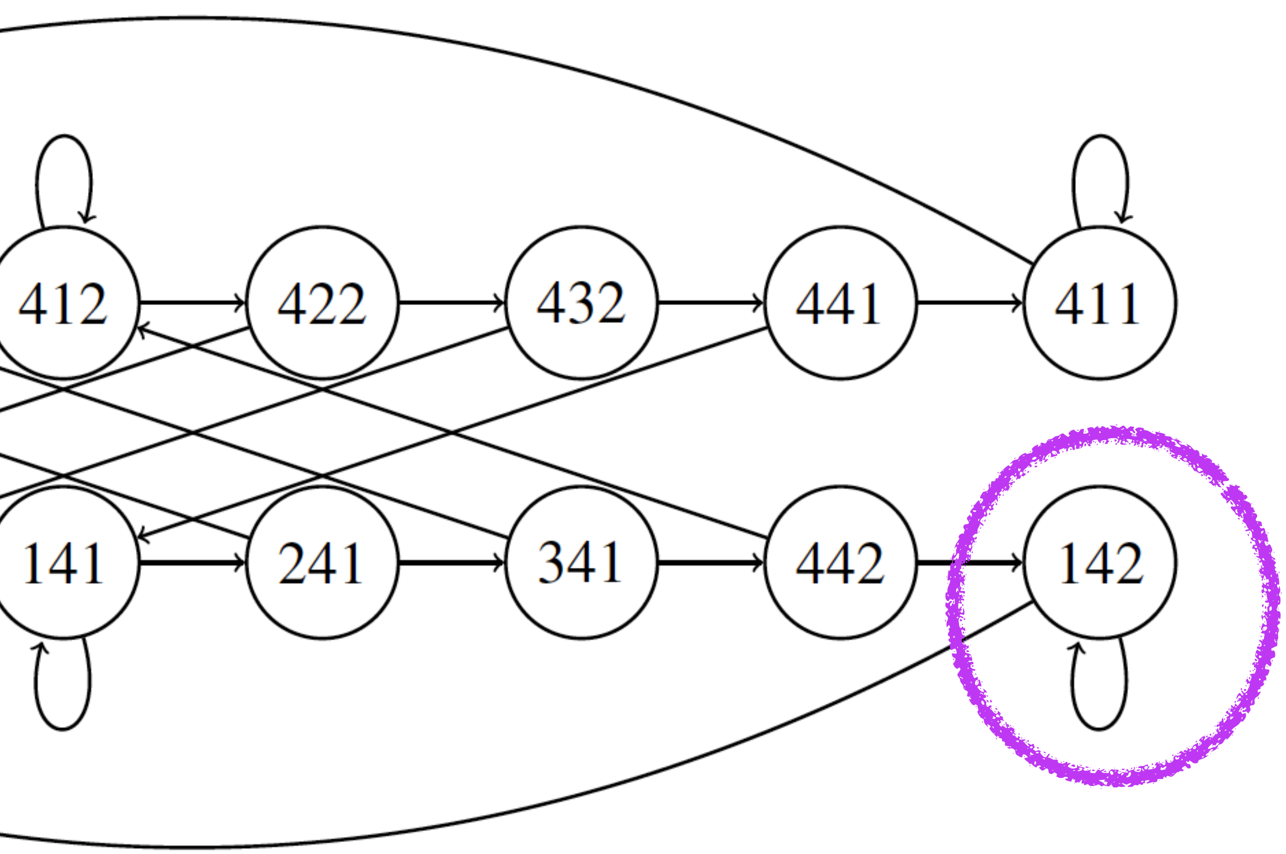
  G (Pi▻1 -> **F** (Pi▻2))

Is attempt no. 3 really correct?

- finally, what about freedom from starvation? ✗

  Expressed as LTL formula (for *i* = *1,2*):

  G (Pi▻1 -> **F** (Pi▻2))



what if P2 terminates in its non-critical section? Then P1 will starve!

Next on the agenda

1. mutual exclusion problem ✓
2. towards a solution ✓
3. Peterson's algorithm
4. Bakery algorithm

Peterson's algorithm (two processes)

- combine attempts no. 2 and 3; if both processes have set their enter-flag to true, then the value of turn decides who may enter the critical section

```
enter1 := false
enter2 := false
turn := 1 or turn := 2
```

P1

```
while true loop
    enter1 := true
    turn := 2
    await not enter2 or turn = 1
    critical section
    enter1 := false
    non-critical section
end
```

P2

```
while true loop
    enter2 := true
    turn := 1
    await not enter1 or turn = 2
    critical section
    enter2 := false
    non-critical section
end
```

Peterson's algorithm satisfies mutual exclusion

- assume that both P1 and P2 are in their critical section and that P1 entered before P2
- when P1 entered the critical section we have enter1 = true, and P2 must thus have seen turn = 2 upon entering its critical section
- P2 could not have executed line 2 after P1 entered, as this sets turn = 1 and would have excluded P2, as P1 does not change turn while being in the critical section
- however, P2 could not have executed line 2 before P1 entered either because then P1 would have seen enter2 = true and turn = 1, although P2 should have seen turn = 2
- => contradiction!

Peterson's algorithm is starvation free

- assume P1 is forced to wait in the entry protocol forever
- P2 can eventually do only one of three actions:

  (1) be in its non-critical section: then enter2 is false, thus allowing P1 to enter.

  (2) wait forever in its entry protocol: impossible because turn cannot be both 1 and 2

  (3) repeatedly cycle through its code: then P2 will set turn to 1 at some point and never change it back

Peterson's algorithm for *n* processes

```
enter[1] := 0; ...; enter[n] := 0
turn[1] := 0; ...; turn[n – 1] := 0
```

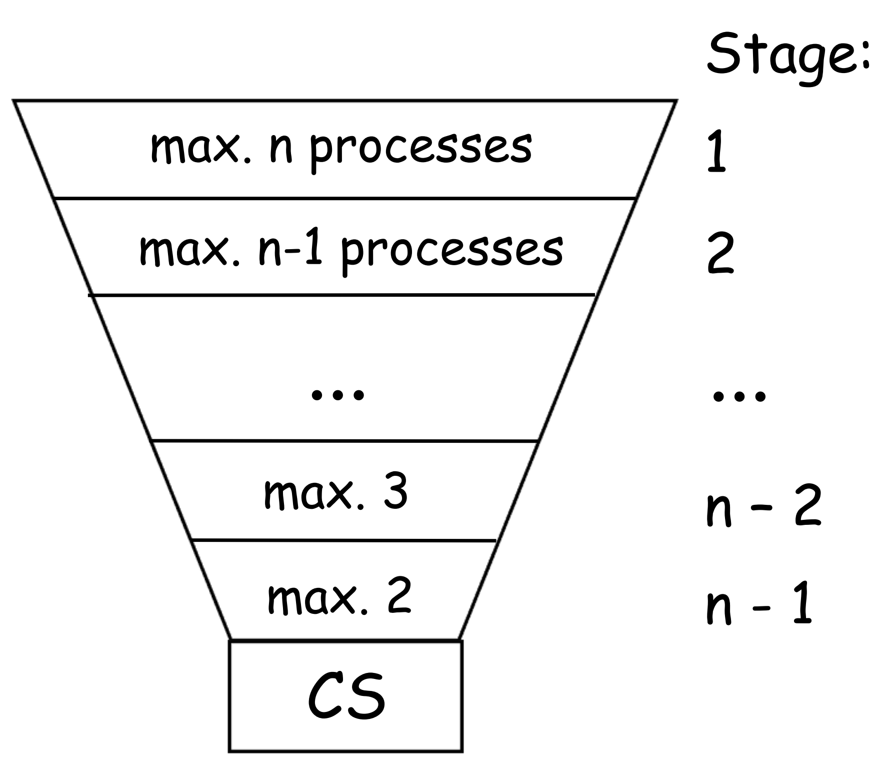
P_i

```
for j = 1 to n – 1 do
    enter[i] := j
    turn[j] := i
    await (for all k != i : enter[k] < j) or turn[j] != i
end
critical section
enter[i] := 0
non-critical section
```

direct generalisation!

Peterson's algorithm for *n* processes

- every process has to go through n – 1 stages to reach the critical section: variable j indicates the stage
- enter[i]: stage the process P_i is currently in
- turn[j]: which process entered stage j last
- waiting: P_i waits if there are still processes at higher stages, or if there are processes at the same stage unless P_i is no longer the last process to have entered this stage
- idea for mutual exclusion proof:
  at most n – j processes can have passed stage j =>
  at most n – (n - 1) = 1 processes can be in the critical section



Next on the agenda

1. mutual exclusion problem ✓
2. towards a solution ✓
3. Peterson's algorithm ✓
4. Bakery algorithm

Freedom

- freedom from starvation still allows that processes may enter their critical sections before a certain, already waiting process is allowed access

- we study an algorithm that has very strong fairness guarantees

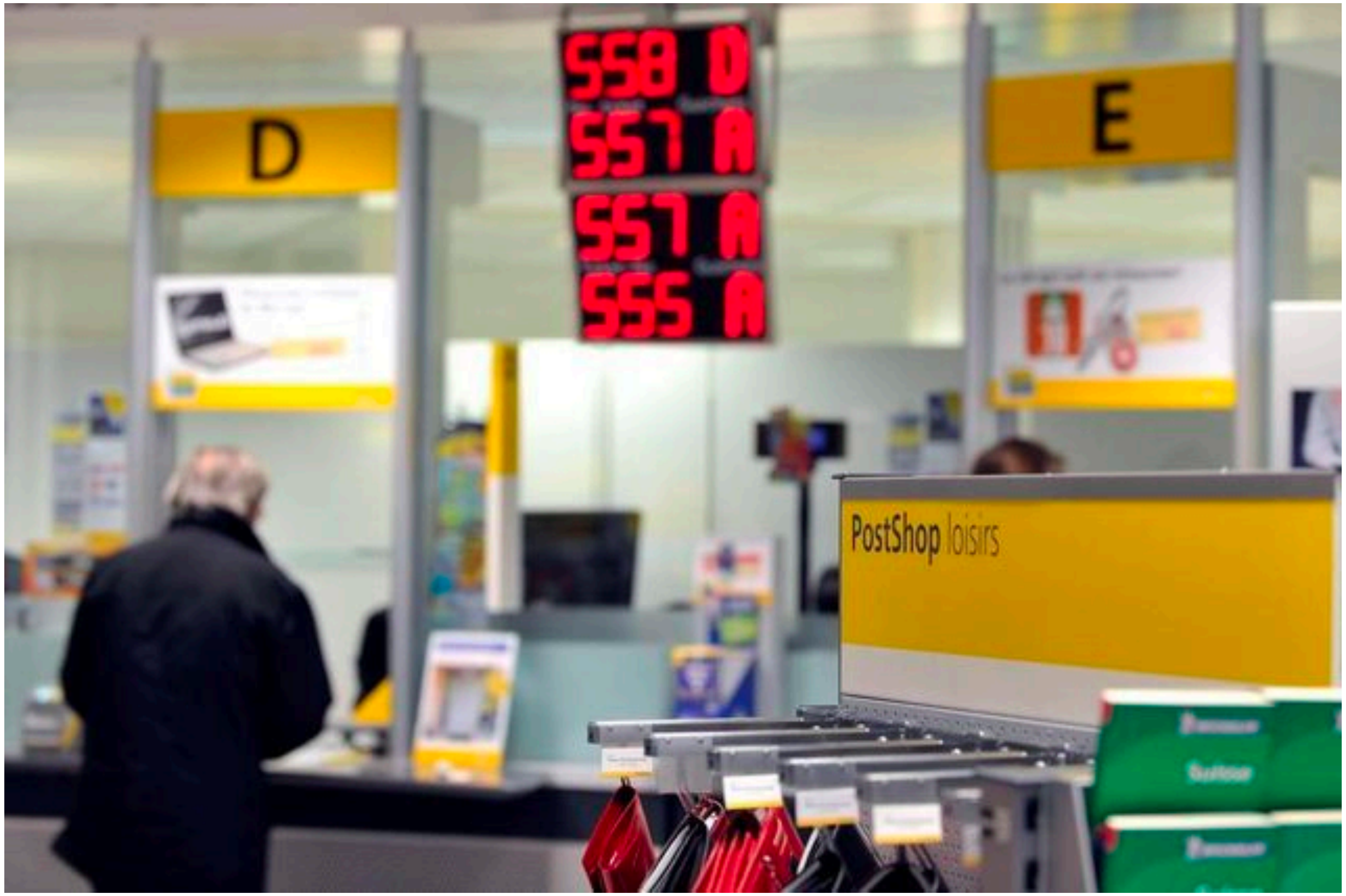
Short aside: bounded waiting

- bounded waiting: if a process is trying to enter its critical section, then there is a bound on the number of times any other process can enter its critical section before the given process does so
- r-bounded waiting: If a process tries to enter its critical section then it will be able to enter before any other process is able to enter its critical section r + 1 times
- first-come-first-served: 0-bounded waiting

Short aside: bounded waiting

- starvation-freedom $\Rightarrow$ deadlock-freedom
- starvation-freedom $\not\Rightarrow$ bounded waiting
- bounded waiting $\not\Rightarrow$ starvation-freedom
- bounded waiting + deadlock-freedom
  $\Rightarrow$ starvation-freedom

Bakery algorithm



Bakery algorithm: first attempt

- idea: ticket systems for customers, at any turn the customer with the lowest number will be served
- number[i]: ticket number drawn by a process Pi
- waiting: until Pi has the lowest number currently drawn

Bakery algorithm: first attempt

- idea: ticket systems for customers, at any turn the customer with the lowest number will be served
- number[i]: ticket number drawn by a process Pi
- waiting: until Pi has the lowest number currently drawn

```
number[1] := 0; ...; number[n] := 0
```

P_i

```
number[i] := 1 + max(number[1], ..., number[n])
for all j != i do
    await number[j] = 0 or number[i] < number[j]
end
critical section
number[i] := 0
non-critical section
```

Bakery algorithm: first attempt

- idea: ticket systems for customers, at any turn the customer with the <u>lowest number</u> will be served
- number[i]: ticket number drawn by a process Pi
- waiting: until Pi has the lowest number currently drawn

```
number[1] := 0; ...; number[n] := 0
```

P_i

```
1   number[i] := 1 + max(number[1], ..., number[n])
2   for all j != i do
3       await number[j] = 0 or number[i] < number[j]
    end
4   critical section
5   number[i] := 0
6   non-critical section
```

problem?

Bakery algorithm: first attempt

- idea: ticket systems for customers, at any turn the customer with the lowest number will be served
- number[i]: ticket number drawn by a process Pi
- waiting: until Pi has the lowest number currently drawn

```
number[1] := 0; ...; number[n] := 0
```

P_i

```
1  number[i] := 1 + max(number[1], ..., number[n])
2  for all j != i do
3      await number[j] = 0 or number[i] < number[j]
   end
4  critical section
5  number[i] := 0
6  non-critical section
```

problem?

atomic? if not, deadlock possible

A fix?

- replace the comparison number[i] < number[j] by (number[i], i) < (number[j], j)
- the "less than" relation is defined in this case as

$$(a, b) < (c, d) \quad \text{if} \quad (a < c) \text{ or } ((a = c) \text{ and } (b < d))$$

- **idea:** if two ticket numbers turn out to be the same, the process with the lower identifier gets precedence

A fix? Unfortunately not.

- unfortunately, with the "fix" we no longer have mutual exclusion:
- P1 and P2 both compute the current maximum as 0
- P2 assigns itself ticket number 1 (number[2] := 1) and proceeds into critical section
- P1 assigns itself ticket number 1 (number[1] := 1) and proceeds into critical section, because

  (number[1], 1) < (number[2], 2)

(Correct) Bakery algorithm

- indicate with a flag if a process is currently calculating its ticket number

```
number[1] := 0; ...; number[n] := 0
choosing[1] := false, ..., choosing[n] := false
```

P_i

```
choosing[i] := true
number[i] := 1 + max(number[1], ..., number[n])
choosing[i] := false
for all j != i do
    await choosing[j] = false
    await number[j] = 0 or (number[i], i) < (number[j], j)
end
critical section
number[i] := 0
non-critical section
```

doorway (lines choosing[i] := true … choosing[i] := false)

bakery (for all j != i do … end)

Some properties

- lemma 1. If processes Pi and Pk are in the bakery and Pi entered the bakery before Pk entered the doorway, then number[i] < number[k].

- lemma 2. If process Pi is in its critical section and process Pk is in the bakery then (number[i], i) < (number[k], k).

Correctness of the Bakery algorithm

- the Bakery algorithm satisfies mutual exclusion

  proof. Follows from Lemma 2.

- the Bakery algorithm is deadlock-free

  proof. Some waiting process P_i has the minimum value of (number[i], i) among all the processes in the bakery. This process must eventually complete the for loop and enter the critical section.

- the Bakery algorithm is first-come-first-served

  proof. Follows from Lemmas 1 and 2.

Considerations

- drawback: values of the ticket numbers can grow unboundedly
  => *two processes could alternatingly draw ticket numbers until the maximum size of an integer on the system is reached*

- size and number of shared memory locations is important
  => *Peterson's algorithm: 2n-1 registers (bounded by n)*
  => *Bakery algorithm: 2n registers (unbounded in size)*
  => *general lower bound: mutual exclusion problem for n processes satisfying mutual exclusion and global progress needs to use n shared one-bit registers*

- algorithms assume memory access is atomic: may not be the case
  => *Bakery algorithm can help: each memory location written by only a single process*
  => *NB: later lecture will consider more complex atomic primitives*

Next on the agenda

1. mutual exclusion problem ✓
2. towards a solution ✓
3. Peterson's algorithm ✓
4. Bakery algorithm ✓

Summary

- mutual exclusion problem is deceptively tricky
- can be solved via locks, but must take care to avoid introducing deadlock, starvation, unfairness
- classical solutions: Peterson's algorithm, Bakery algorithm
- *coming weeks:* more modern synchronisation mechanisms to solve mutual exclusion