The Critical Section Problem

Algorithm 3.1: Critical section problem	
global variables	
p	**q**
local variables	local variables
loop forever	loop forever
non-critical section	non-critical section
preprotocol	preprotocol
critical section	critical section
postprotocol	postprotocol

Any solution to the critical section (CS) problem must satisfy three requirements:

- Mutual exclusion (ME)
- Freedom from deadlock
- Freedom from starvation

Mutual exclusion: The critical section statements must not be interleaved.

Mutual exclusion: The critical section statements must not be interleaved.

Deadlock free: If some processes are trying to enter their CS's, then one must eventually succeed.

Mutual exclusion: The critical section statements must not be interleaved.

Deadlock free: If some processes are trying to enter their CS's, then one must eventually succeed.

$$(\exists p \mid \text{tryingToEnterCS}(p) : \text{entersCS}(p))$$

Mutual exclusion: The critical section statements must not be interleaved.

Deadlock free: If some processes are trying to enter their CS's, then one must eventually succeed.

$$(\exists p \mid \text{tryingToEnterCS}(p) : \text{entersCS}(p))$$

Starvation free: If any process tries to enter its CS, then that process must succeed.

Mutual exclusion: The critical section statements must not be interleaved.

Deadlock free: If <u>some</u> processes are trying to enter their CS's, then <u>one</u> must eventually succeed.

$$(\exists p \mid \text{tryingToEnterCS}(p) : \text{entersCS}(p))$$

Starvation free: If <u>any</u> process tries to enter its CS, then <u>that</u> process must succeed.

$$(\forall p \mid \text{tryingToEnterCS}(p) : \text{entersCS}(p))$$

Mutual exclusion:
A safety property. Always no interleaving in CS.

Mutual exclusion:
A safety property. Always no interleaving in CS.

Deadlock free:
A liveness property. Eventually one of several process must enter CS.

Mutual exclusion:
A safety property. Always no interleaving in CS.

Deadlock free:
A liveness property. Eventually one of several process must enter CS.

Starvation free:
A liveness property. Eventually a particular process must enter CS.

Deadlock vs Starvation

Starvation-free is a stronger requirement than deadlock-free.

Deadlock vs Starvation

Starvation-free is a stronger requirement than deadlock-free.

$$(\forall p \mid \text{tryingToEnterCS}(p) : \text{entersCS}(p))$$

implies

$$(\exists p \mid \text{tryingToEnterCS}(p) : \text{entersCS}(p))$$

Deadlock vs Starvation

Starvation-free is a stronger requirement than deadlock-free.

$$(\forall p \mid \text{tryingToEnterCS}(p) : \text{entersCS}(p))$$

implies

$$(\exists p \mid \text{tryingToEnterCS}(p) : \text{entersCS}(p))$$

(9.20.2) $(\exists x \mid : R) \Rightarrow ((\forall x \mid R : P) \Rightarrow (\exists x \mid R : P))$

General analysis assumptions

Once a process starts executing statements in its CS, it must eventually terminate (leave its CS).

The non-critical sections need not terminate.

No variables in the protocols are outside the protocols and vice versa.

The operating system scheduler is weakly fair.

First attempt

The preprotocol is a single atomic “await” statement.

The postprotocol is a single atomic assignment statement.

The processes take turns accessing their critical sections.

Algorithm 3.2: First attempt			
integer turn ← 1			
	p		**q**
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	await turn = 1	q2:	await turn = 2
p3:	critical section	q3:	critical section
p4:	turn ← 2	q4:	turn ← 1

Variable turn does not appear in the non-critical section or the critical section.

Algorithm 3.2: First attempt		
integer turn ← 1		
p		**q**
	loop forever	loop forever
p1:	non-critical section	q1: non-critical section
p2:	await turn = 1	q2: await turn = 2
p3:	critical section	q3: critical section
p4:	turn ← 2	q4: turn ← 1

Spin lock

A technique for implementing the await statement with a loop.

await turn = 1

is implemented as

```
while (turn != 1) ;  // Do nothing
```

Spin lock

How many critical references are in the spin lock?

```
while (turn != 1) ;  // Do nothing
```

Spin lock

How many critical references are in the spin lock?

```
while (turn != 1) ;  // Do nothing
```

One!

Spin lock

How many critical references are in the spin lock?

```
while (turn != 1) ;  // Do nothing
```

One!

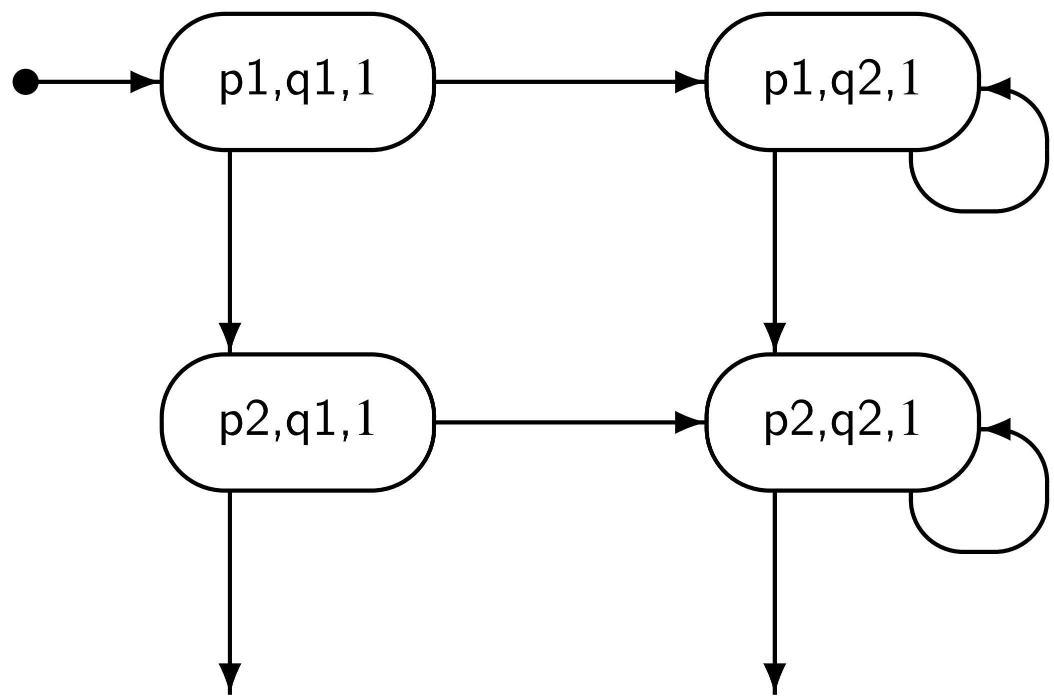
So, as long as the other statements in our solution have at most one critical reference, the program satisfies LCR. We can analyze it as if all the statements are atomic.

Class exercise

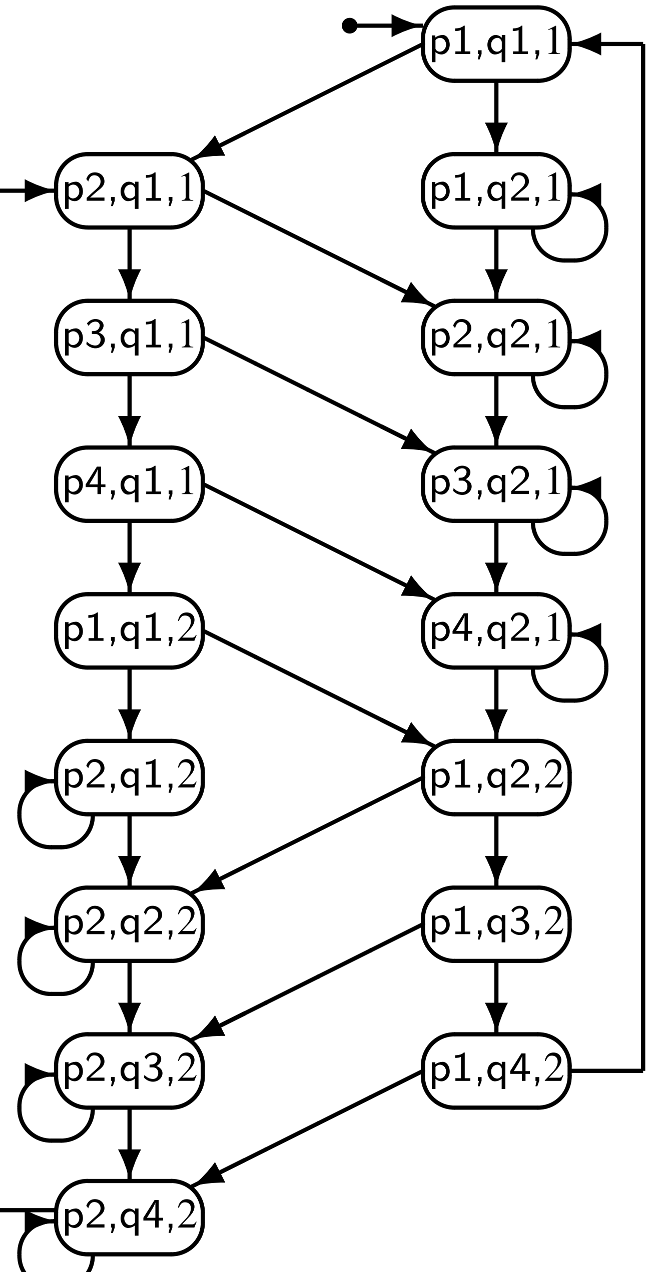
Construct the first part of the state transition diagram from (p1, q1, 1) to (p2, q2, 1).
(Label each transition with the process that executes.)

Algorithm 3.2: First attempt	
integer turn ← 1	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await turn = 1	q2: await turn = 2
p3: critical section	q3: critical section
p4: turn ← 2	q4: turn ← 1

First States of the State Diagram



State Diagram for the First Attempt



Analysis of mutual exclusion

Do either of the states (p3, q3, 1) or (p3, q3, 2) appear in the state transition diagram?

Analysis of mutual exclusion

Do either of the states (p3, q3, 1) or (p3, q3, 2) appear in the state transition diagram?

No!

Analysis of mutual exclusion

Do either of the states (p3, q3, 1) or (p3, q3, 2) appear in the state transition diagram?

No!

Conclusion: We have ME.

Problem

There are too many states to examine.

Problem

There are too many states to examine.

Solution

Omit statements p1 and p3, as they do not matter in the analysis anyway.

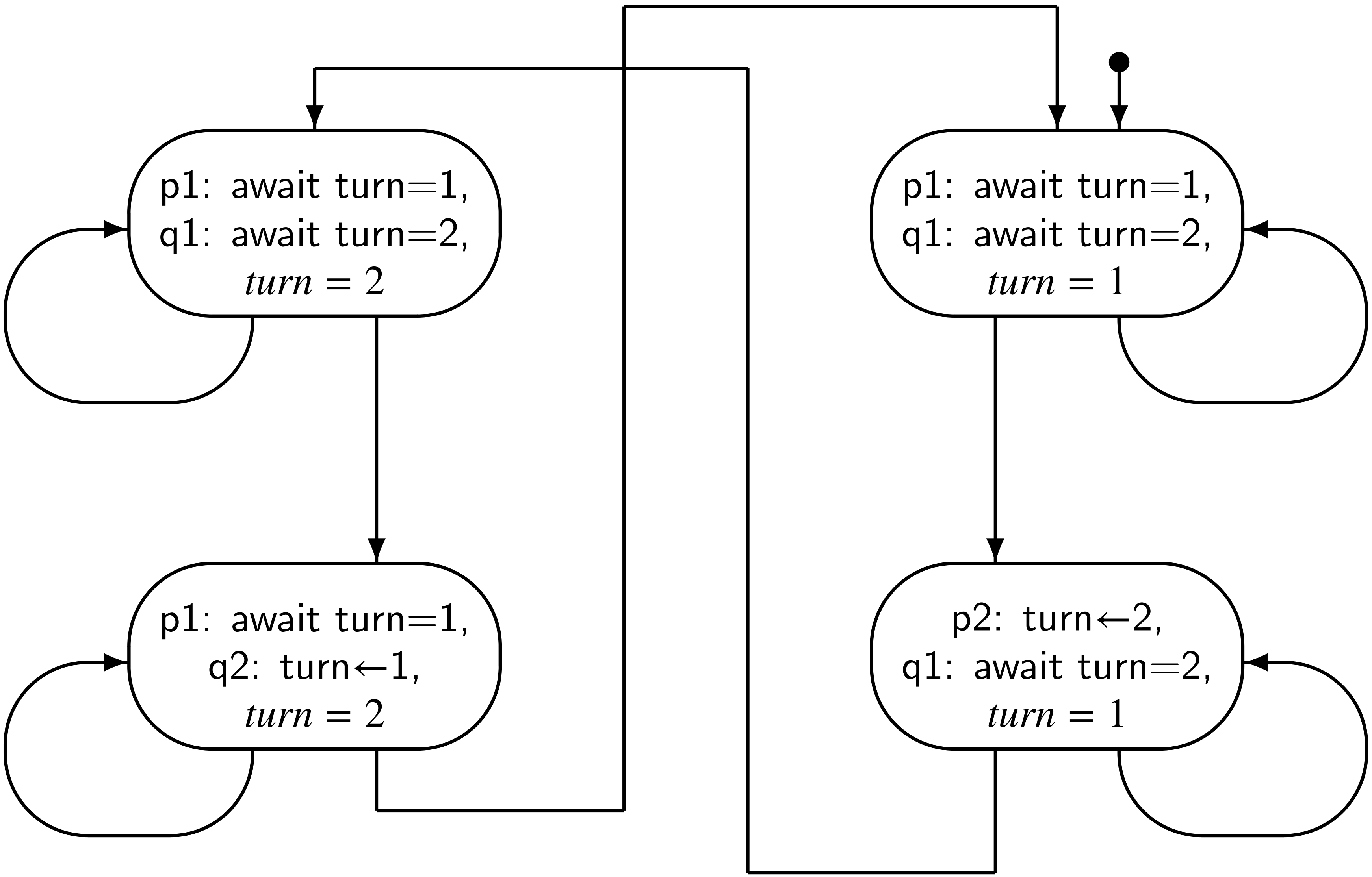
Algorithm 3.5: First attempt (abbreviated)	
integer turn ← 1	
p	**q**
loop forever	loop forever
p1: await turn = 1	q1: await turn = 2
p2: turn ← 2	q2: turn ← 1

Class exercise

Construct the state transition diagram.

Algorithm 3.5: First attempt (abbreviated)	
integer turn ← 1	
p	**q**
loop forever p1: await turn = 1 p2: turn ← 2	loop forever q1: await turn = 2 q2: turn ← 1

State Diagram for the Abbreviated First Attempt



Analysis of mutual exclusion

Analysis of mutual exclusion

Do either of the states (p2, q2, 1) or (p2, q2, 2) appear in the state transition diagram?

Analysis of mutual exclusion

Do either of the states (p2, q2, 1) or (p2, q2, 2) appear in the state transition diagram?

No!

Analysis of mutual exclusion

Do either of the states (p2, q2, 1) or (p2, q2, 2) appear in the state transition diagram?

No!

Conclusion: We have ME.

Analysis of deadlock

Deadlock free: If some try to enter, one must succeed.

Question: In what state are p and q both trying to enter?

Analysis of deadlock

Deadlock free: If some try to enter, one must succeed.

Question: In what state are p and q both trying to enter?

Answer: In states (p1, q1, 1) and (p1, q1, 2).

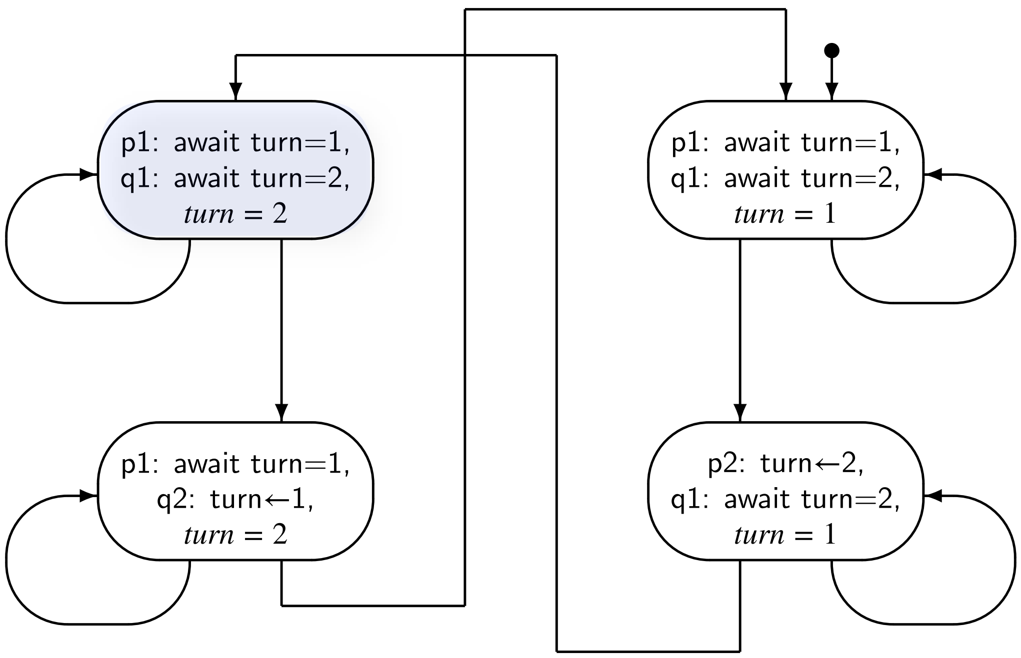
Analysis of deadlock

Deadlock free: If some try to enter, one must succeed.

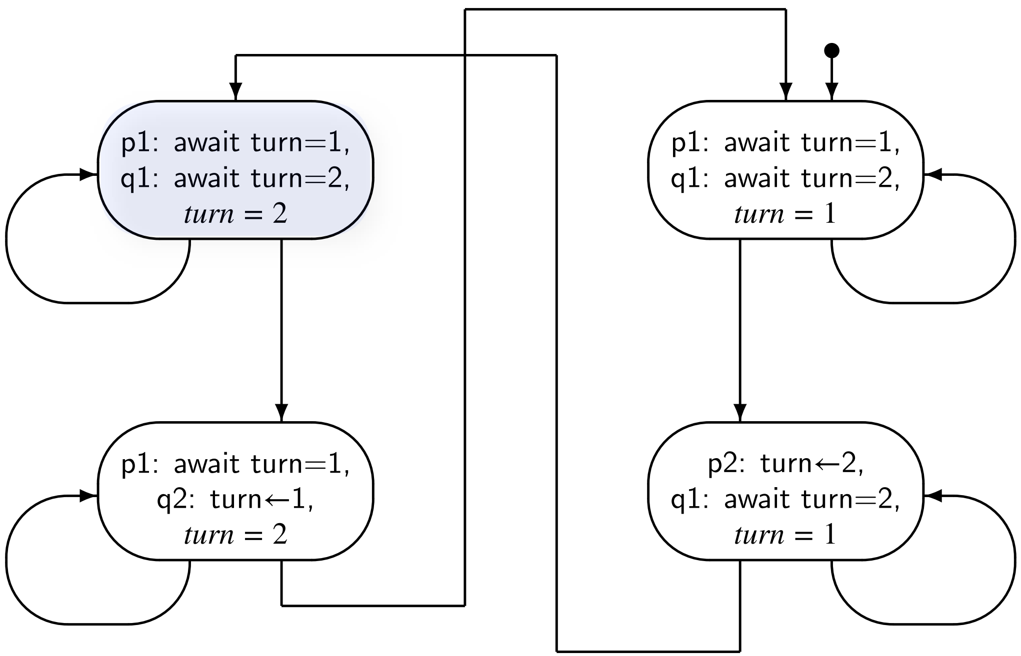
Question: In what state are p and q both trying to enter?

Answer: In states (p1, q1, 1) and (p1, q1, 2).

Analysis: Deduce what must happen from one of these states, say (p1, q1, 2).

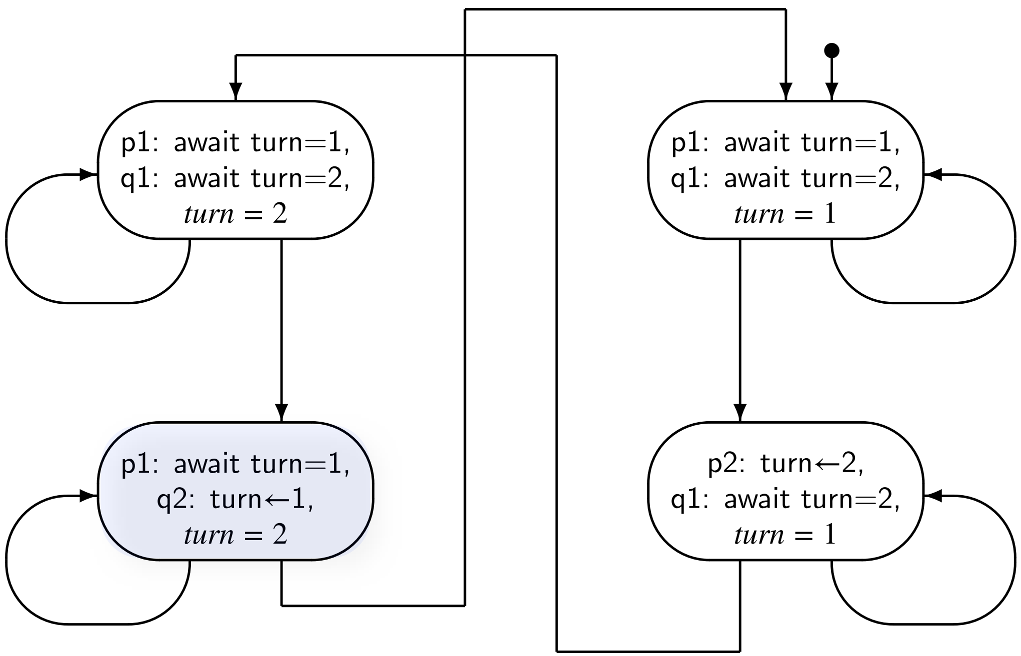

p1: await turn=1,
q1: await turn=2,
turn = 2
p1: await turn=1,
q1: await turn=2,
turn = 1
p1: await turn=1,
q2: turn←1,
turn = 2
p2: turn←2,
q1: await turn=2,
turn = 1

$$(p1, q1, 2)$$

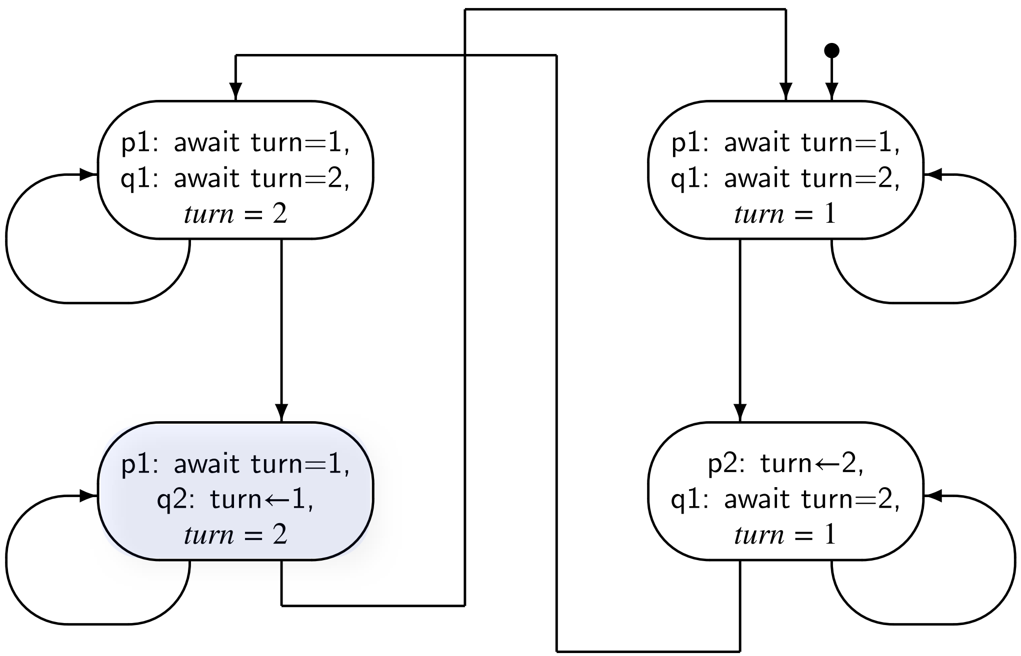


$$(p1, q1, 2)$$

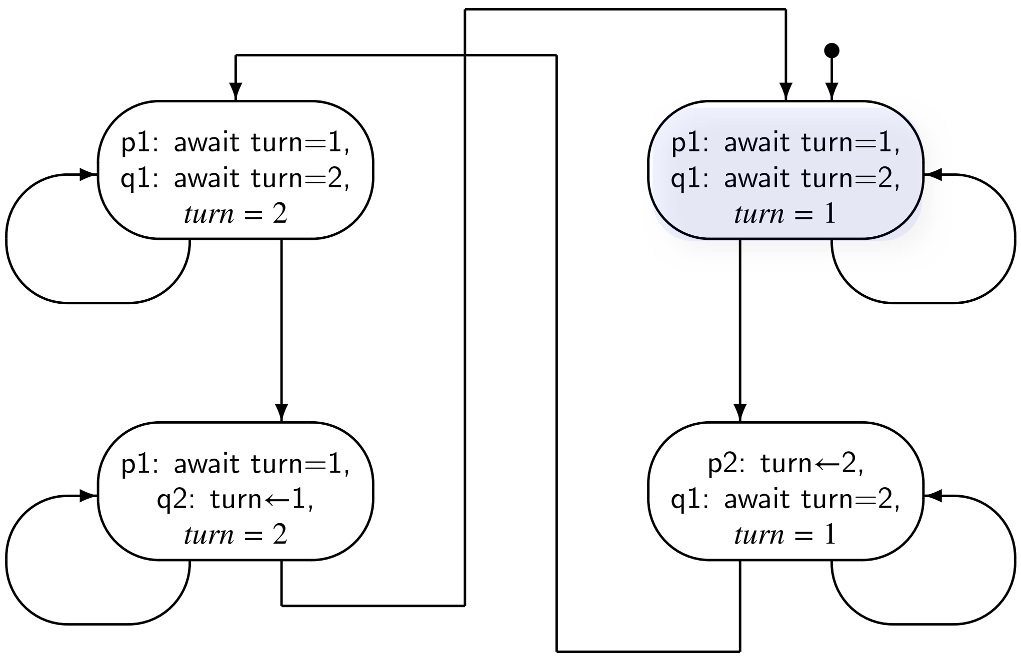
$$\Rightarrow \quad \langle q \text{ selected by weak fairness} \rangle$$



$$
\begin{aligned}
& (p1, q1, 2) \\
\Rightarrow \quad & \langle q \text{ selected by weak fairness} \rangle \\
& (p1, q2, 2)
\end{aligned}
$$



$$
\begin{aligned}
& (p1, q1, 2) \\
\Rightarrow \quad & \langle q \text{ selected by weak fairness} \rangle \\
& (p1, q2, 2) \\
\Rightarrow \quad & \langle q \text{ must complete CS, selected by weak fairness} \rangle
\end{aligned}
$$



$$
\begin{aligned}
& (p1, q1, 2) \\
\Rightarrow \quad & \langle q \text{ selected by weak fairness} \rangle \\
& (p1, q2, 2) \\
\Rightarrow \quad & \langle q \text{ must complete CS, selected by weak fairness} \rangle \\
& (p1, q1, 1)
\end{aligned}
$$

Analysis of deadlock

Analysis starting with state (p1, q1, 1) is similar.

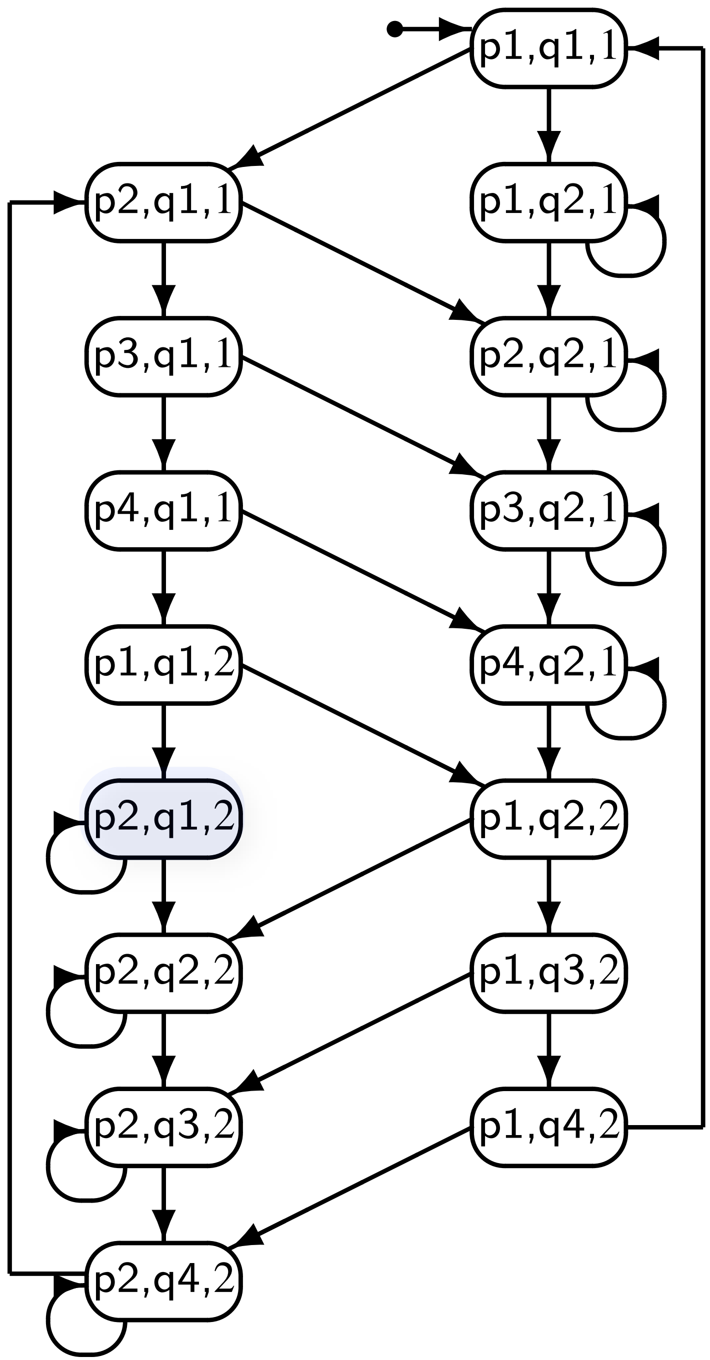
If <u>some</u> try to enter, <u>one</u> must succeed.

Conclusion: Deadlock-free.

Analysis of starvation

Starvation free: If <u>any</u> tries to enter, <u>it</u> must succeed.

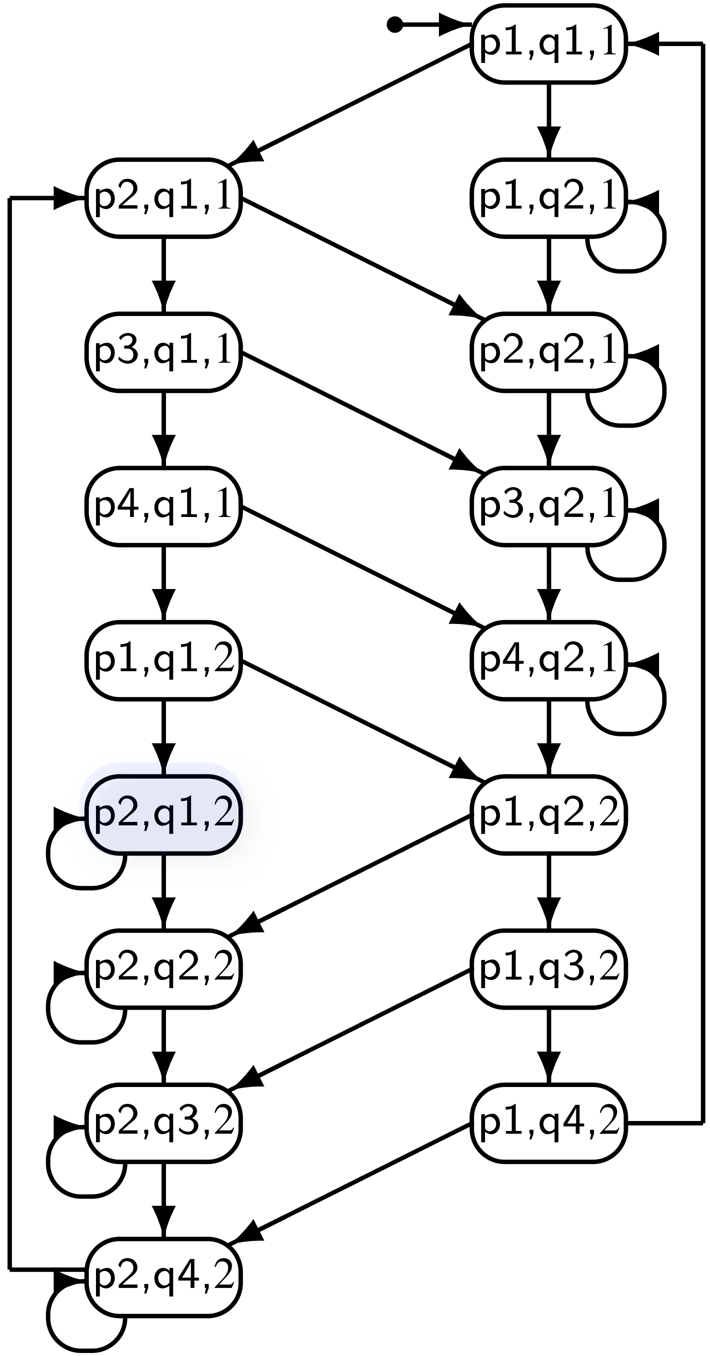
Analysis: See state (p2, q1, 2) in the non-abbreviated state diagram.



Algorithm 3.2: First attempt

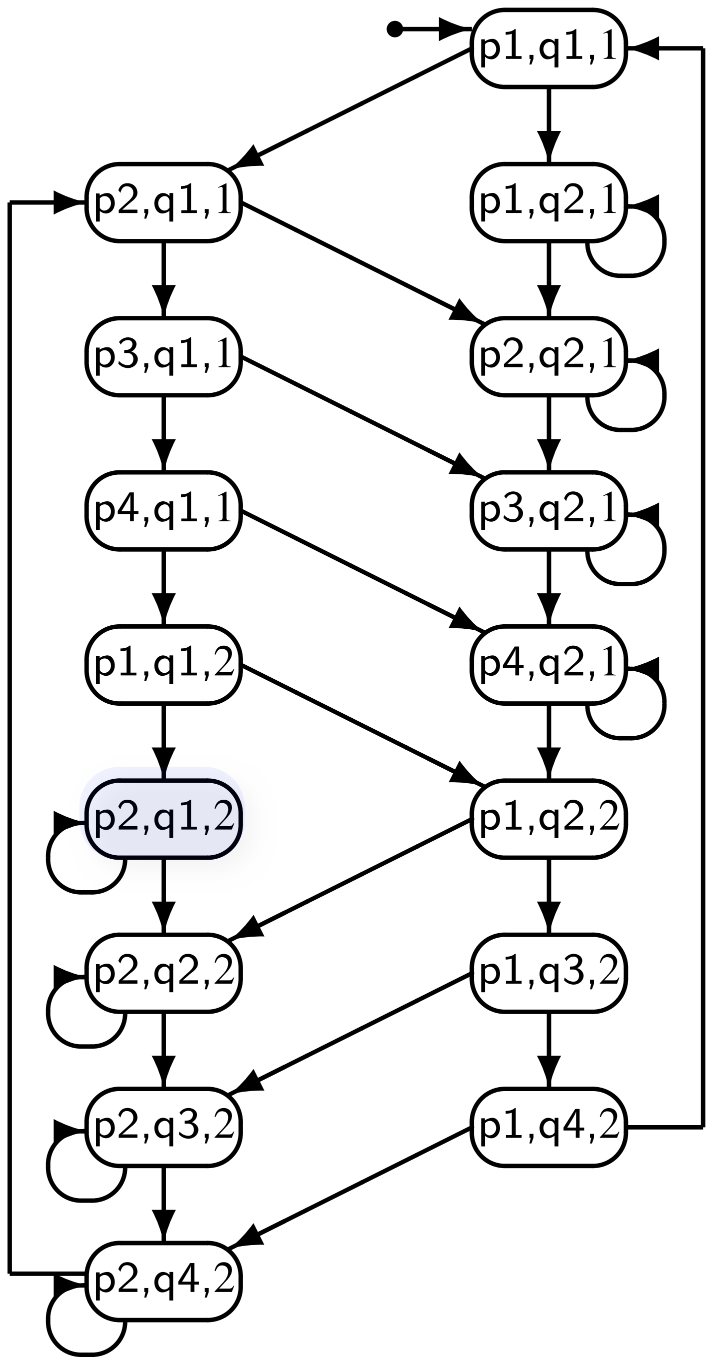
integer turn ← 1

	p		q
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	await turn = 1	q2:	await turn = 2
p3:	critical section	q3:	critical section
p4:	turn ← 2	q4:	turn ← 1



Algorithm 3.2: First attempt			
integer turn ← 1			
p		**q**	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	await turn = 1	q2:	await turn = 2
p3:	critical section	q3:	critical section
p4:	turn ← 2	q4:	turn ← 1

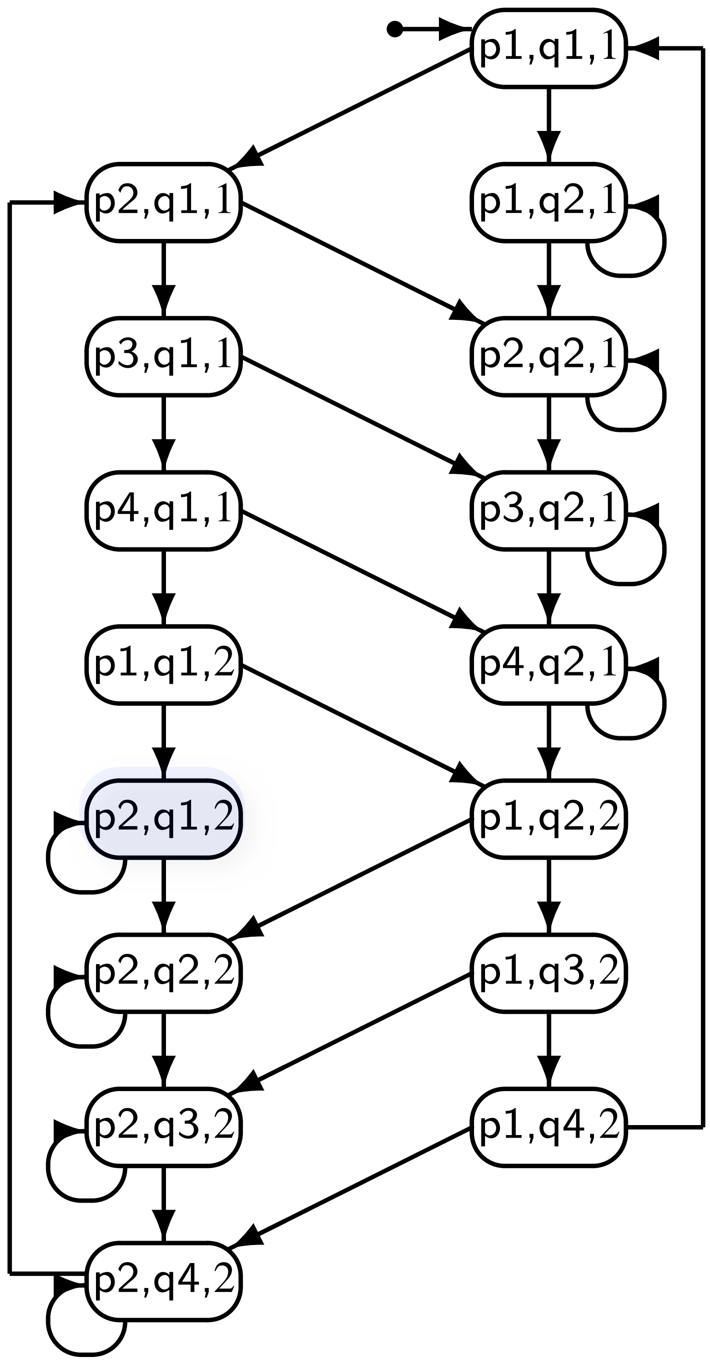
p is trying to enter CS.



Algorithm 3.2: First attempt			
integer turn ← 1			
p		**q**	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	await turn = 1	q2:	await turn = 2
p3:	critical section	q3:	critical section
p4:	turn ← 2	q4:	turn ← 1

p is trying to enter CS.

q is in non-CS.



Algorithm 3.2: First attempt

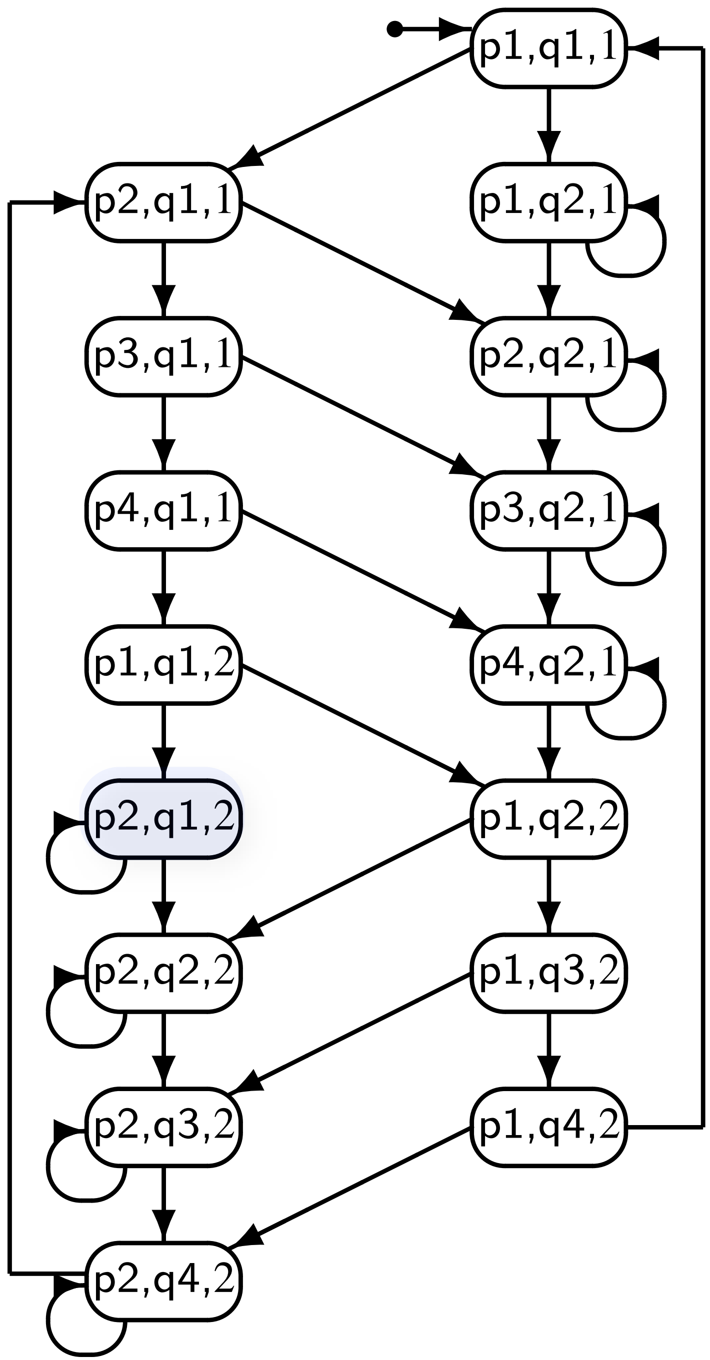
integer turn ← 1

p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await turn = 1	q2: await turn = 2
p3: critical section	q3: critical section
p4: turn ← 2	q4: turn ← 1

p is trying to enter CS.

q is in non-CS.

q need not make progress.



Algorithm 3.2: First attempt			
integer turn ← 1			
p		**q**	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	await turn = 1	q2:	await turn = 2
p3:	critical section	q3:	critical section
p4:	turn ← 2	q4:	turn ← 1

p is trying to enter CS.

q is in non-CS.

q need not make progress.

So, p is starved.

Analysis of starvation

Conclusion: The first attempt is not starvation-free.

Demo `alg-3-2.cm`

Demo as written. ⇒ It appears to work. We have ME.

Change one process to loop 5 times. ⇒ We have starvation.

alg-3-2.cm

```
int n = 0;
int turn = 1;

void r() {
  int temp, i;
  for (i = 0; i < 10; i++) {
    // non-critical section
    cout << "r.i = " << i << endl;
    // preprotocol
    while (turn != 1)
      ;
    // critical section
    temp = n;
    n = temp + 1;
    // postprotocol
    turn = 2;
  }
}
```

alg-3-2.cm, continued

```
void q() {
  int temp, i;
  for (i = 0; i < 10; i++) {
    // non-critical section
    cout << "q.i = " << i << endl;
    // preprotocol
    while (turn != 2)
      ;
    // critical section
    temp = n;
    n = temp + 1;
    // postprotocol
    turn = 1;
  }
}

void main() {
  cobegin { r(); q(); }
  cout << "The value of n is " << n << "\n";
}
```

Demo `Alg0302.java`

Each process has its own processor ID initialized in the constructor.

Alg0302.java

```
class Alg0302 extends Thread {
   static volatile int n = 0;
   static volatile int turn = 1;
   int processID;

   Alg0302(int pID) {
      processID = pID;
   }
```

Alg0302.java, continued

```
public void run() {
   int temp, delay;
   for (int i = 0; i < 10; i++) {
      try {
         // non-critical section
         System.out.println("p" + processID + ".i = " + i);
         // preprotocol
         while (turn != processID)
            ;
         // critical section
         delay = (int) (100 * Math.random());
         Thread.sleep(delay);
         temp = n;
         delay = (int) (100 * Math.random());
         Thread.sleep(delay);
         n = temp + 1;
         // postprotocol
         turn = (processID == 1) ? 2 : 1;
      } catch (InterruptedException e) {
      }
   }
}
```

Alg0302.java, continued

```
    public static void main(String[] args) {
        Alg0302 p1 = new Alg0302(1);
        Alg0302 p2 = new Alg0302(2);
        p1.start();
        p2.start();
        try {
            p1.join();
            p2.join();
        } catch (InterruptedException e) {
        }
        System.out.println("The value of n is " + n);
    }
}
```

Second attempt

p announces its intent to enter its critical section by setting wantp to true.

q waits until p does not want to enter before q announces q’s intent to enter q’s CS.

When p exits its CS, p sets wantp to false, as p no longer wants to enter.

Algorithm 3.6: Second attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await wantq = false	q2: await wantp = false
p3: wantp ← true	q3: wantq ← true
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Analysis of starvation

Suppose p is stuck at p1, that is, not making progress, with wantp and wantq both false.

Algorithm 3.6: Second attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section ←	q1: non-critical section
p2: await wantq = false	q2: await wantp = false
p3: wantp ← true	q3: wantq ← true
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Analysis of starvation

The following scenario is still possible:
q1, q2, q3, q4, q5, q1, q2, q3, q4, q5, q1, q2, q3, ...

Analysis of starvation

The following scenario is still possible:
q1, q2, q3, q4, q5, q1, q2, q3, q4, q5, q1, q2, q3, ...

Conclusion: Second attempt is starvation-free.

Analysis of mutual exclusion

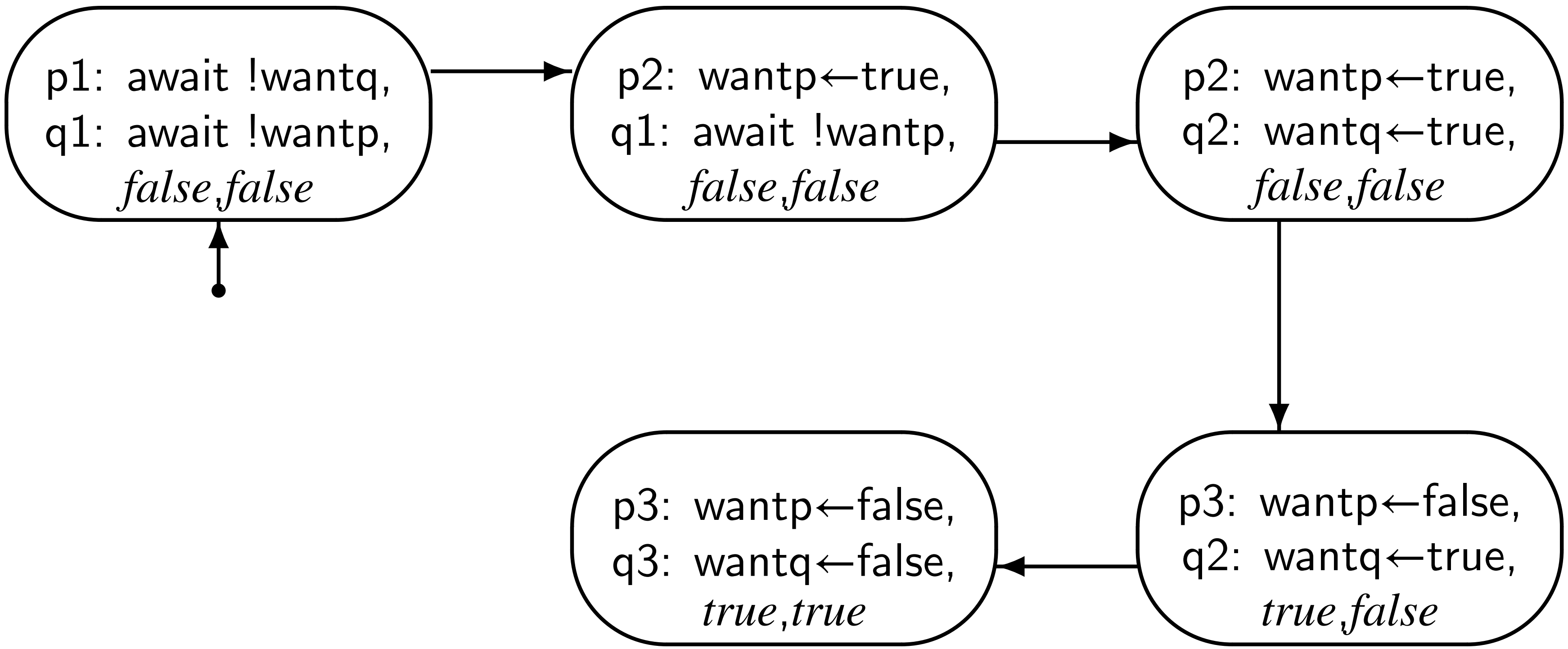
Consider the abbreviated algorithm.

Algorithm 3.7: Second attempt (abbreviated)	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: await wantq = false	q1: await wantp = false
p2: wantp ← true	q2: wantq ← true
p3: wantp ← false	q3: wantq ← false

Analysis of mutual exclusion

Class exercise: Starting at state (p1, q1, F, F), show state transitions that get to state (p3, q3, _, _).

Fragment of the State Diagram for the Second Attempt



Analysis of mutual exclusion

Conclusion: Second attempt does not enforce ME.

Third attempt

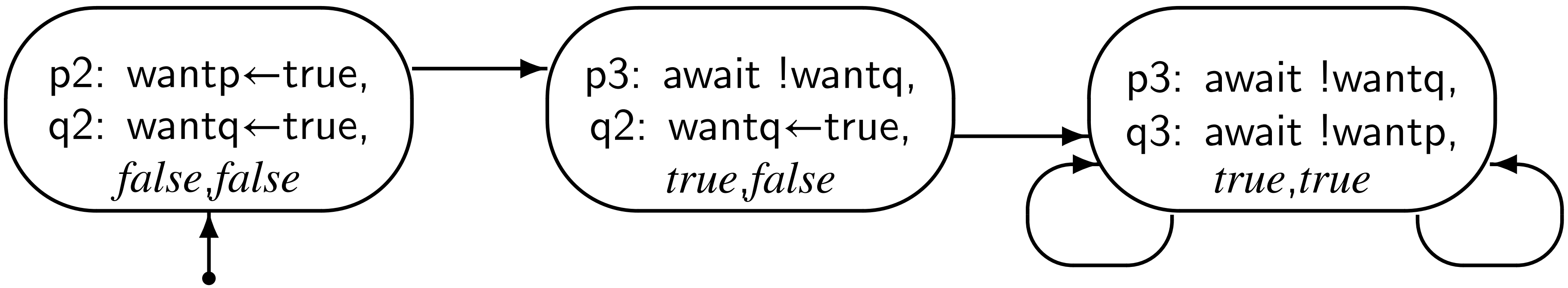
Switch the order of p2 and p3 from the second attempt to get mutual exclusion.

Algorithm 3.8: Third attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Analysis of deadlock

Class exercise: Starting at state (p1, q1, F, F), show state transitions that get to state (p3, q3, T, T) with no possibility of progress.

Fragment of the State Diagram Showing Deadlock



Analysis of deadlock

Conclusion: Third attempt is not deadlock-free.

Fourth attempt

p announces intent to enter by setting wantp to true.

In a loop, checks if q wants to enter. If so, they are wanting to enter at the same time.

In the body, p sets wantp to false and then back to true, allowing interleaving between them. p is temporarily relinquishing its attempt to enter if at first unsuccessful.

Algorithm 3.9: Fourth attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: while wantq	q3: while wantp
p4: wantp ← false	q4: wantq ← false
p5: wantp ← true	q5: wantq ← true
p6: critical section	q6: critical section
p7: wantp ← false	q7: wantq ← false

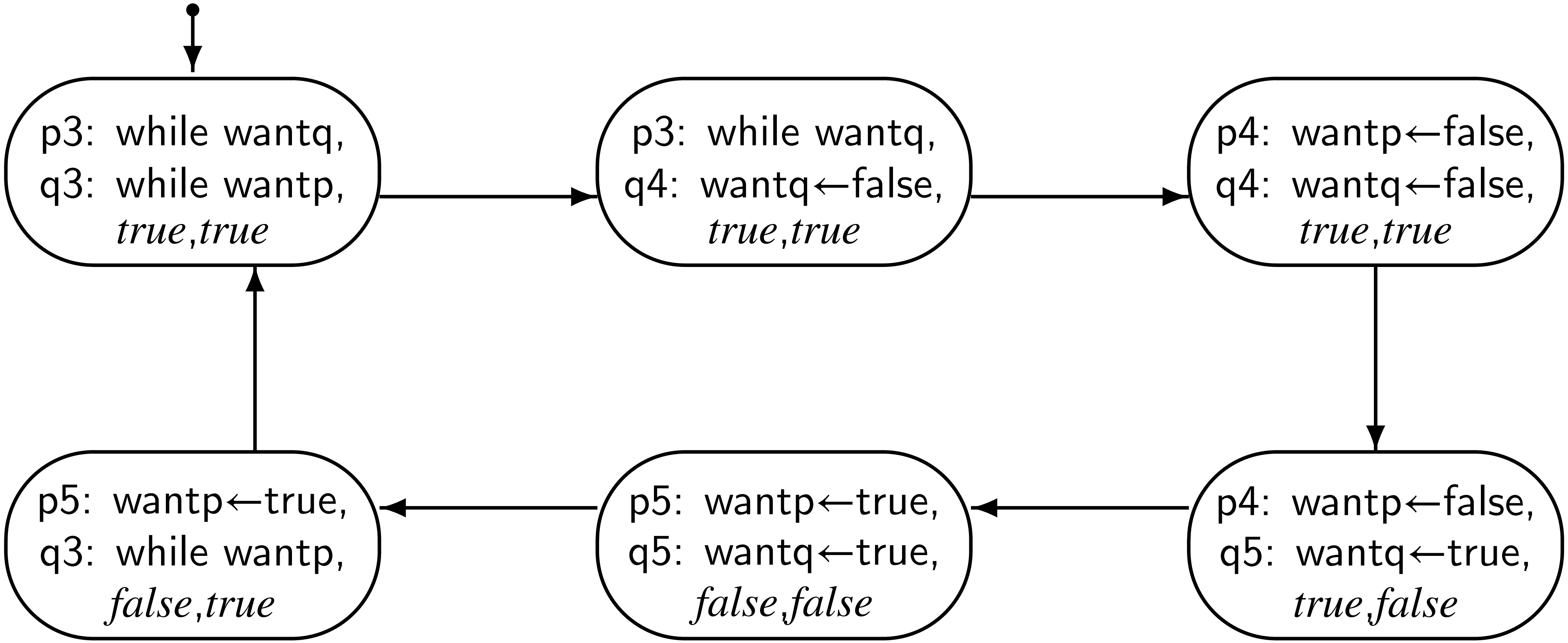
Fourth attempt

Mutual exclusion: Yes (Proof omitted.)

Deadlock-free: Yes (Proof omitted.)

Starvation-free: No
There is a perfect interleaving that starves both.

Cycle in the State Diagram for the Fourth Attempt



Dekker's algorithm

A combination of the first and fourth attempts.

The turn variable means whose turn it is to insist on entering if they both want to enter at the same time.

Algorithm 3.10: Dekker's algorithm

boolean wantp ← false, wantq ← false
integer turn ← 1

p		q	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	while wantq	q3:	while wantp
p4:	if turn = 2	q4:	if turn = 1
p5:	wantp ← false	q5:	wantq ← false
p6:	await turn = 1	q6:	await turn = 2
p7:	wantp ← true	q7:	wantq ← true
p8:	critical section	q8:	critical section
p9:	turn ← 2	q9:	turn ← 1
p10:	wantp ← false	q10:	wantq ← false

Dekker's algorithm

In process p, if

- wantq = true
- turn = 2

then q will enter its CS.

Proof of correctness is in Chapter 4.

Test-and-set statements

If high-level programming languages had atomic test-and-set statements, the critical section problem would be trivial.

```
test-and-set (common, local) {
   local ← common
   common ← 1
}
```

The test-and-set is guaranteed atomic, i.e., no interleaving between its two internal statements.

CS algorithm with test-and-set

Initialize common to 0.

Preprotocol: Repeatedly test-and-set until local is 0. If common is initially 0, local will be set to 0 and common to 1 in one atomic operation, and process will enter CS.

Postprotocol: Set common to 0, so the next process will be able to enter its CS.

Algorithm 3.11: Critical section problem with test-and-set	
integer common ← 0	
p	**q**
integer local1	integer local2
loop forever	loop forever
p1: non-critical section	q1: non-critical section
repeat	repeat
p2: test-and-set(	q2: test-and-set(
common, local1)	common, local2)
p3: until local1 = 0	q3: until local2 = 0
p4: critical section	q4: critical section
p5: common ← 0	q5: common ← 0

Exchange statements

If high-level programming languages had atomic exchange statements, the critical section problem would be trivial.

```
exchange (a, b) {
   integer temp
   temp ← a
   a ← b
   b ← temp
}
```

The exchange is guaranteed atomic, i.e., no interleaving between its three internal statements.

CS algorithm with exchange

Initialize common to 1 and local to 0.

Preprotocol: Repeatedly exchange until local is 1. If common is initially 1, local will be set to 1 and common to 0 in one atomic operation, and process will enter CS.

Postprotocol: Exchange common and local back again, so the next process will be able to enter its CS.

Algorithm 3.12: Critical section problem with exchange	
integer common ← 1	
p	**q**
integer local1 ← 0	integer local2 ← 0
loop forever	loop forever
p1: non-critical section	q1: non-critical section
repeat	repeat
p2: exchange(common, local1)	q2: exchange(common, local2)
p3: until local1 = 1	q3: until local2 = 1
p4: critical section	q4: critical section
p5: exchange(common, local1)	q5: exchange(common, local2)

Test-and-set at the machine level - Intel

INSTRUCTION SET REFERENCE

BTS—Bit Test and Set

Opcode	Instruction	Description
0F AB	BTS *r/m16,r16*	Store selected bit in CF flag and set
0F AB	BTS *r/m32,r32*	Store selected bit in CF flag and set
0F BA /5 *ib*	BTS *r/m16,imm8*	Store selected bit in CF flag and set
0F BA /5 *ib*	BTS *r/m32,imm8*	Store selected bit in CF flag and set

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

CF ← Bit(BitBase, BitOffset)
Bit(BitBase, BitOffset) ← 1;

Exchange at the machine level - ARM

4.35 SWP - Swap

Syntax:

SWP{<cond>} <Rd>, <Rm>, [<Rn>]

RTL:

if(cond)
    temp ← [Rn]
    [Rn] ← Rm
    Rd ← temp

Flags updated:

None

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	0	0	Rn				Rd				SBZ				1	0	0	1	Rm			