CRITICAL SECTION PROBLEM

Dr. Emmanuel S. Pilli

Critical Section Problem

- Each of *N* processes is executing in an infinite loop, a sequence of statements that can be divided into two subsequences:
  - *critical section*
  - *non critical section*
- Three correctness specifications are required of any solution:
  - *Mutual Exclusion*
  - *Freedom from deadlock*
  - *Freedom from starvation*

Critical Section Problem

- Mutual Exclusion
  - Statements from the critical sections of two or more processes must not be interleaved
- Freedom from deadlock
  - If ***some*** processes are trying to enter their critical sections, then ***one*** of them must eventually succeed
- Freedom from starvation
  - If ***any*** process tries to enter its critical section, then ***that*** process must eventually succeed

Critical Section Problem

- A *synchronization* mechanism must be provided to ensure that the correctness requirements are met.
- Synchronization mechanism consists of additional statements that are placed before and after the critical section
- The statements placed before critical section are called *preprotocol* and those after it are called *postprotocol*

Critical Section Problem

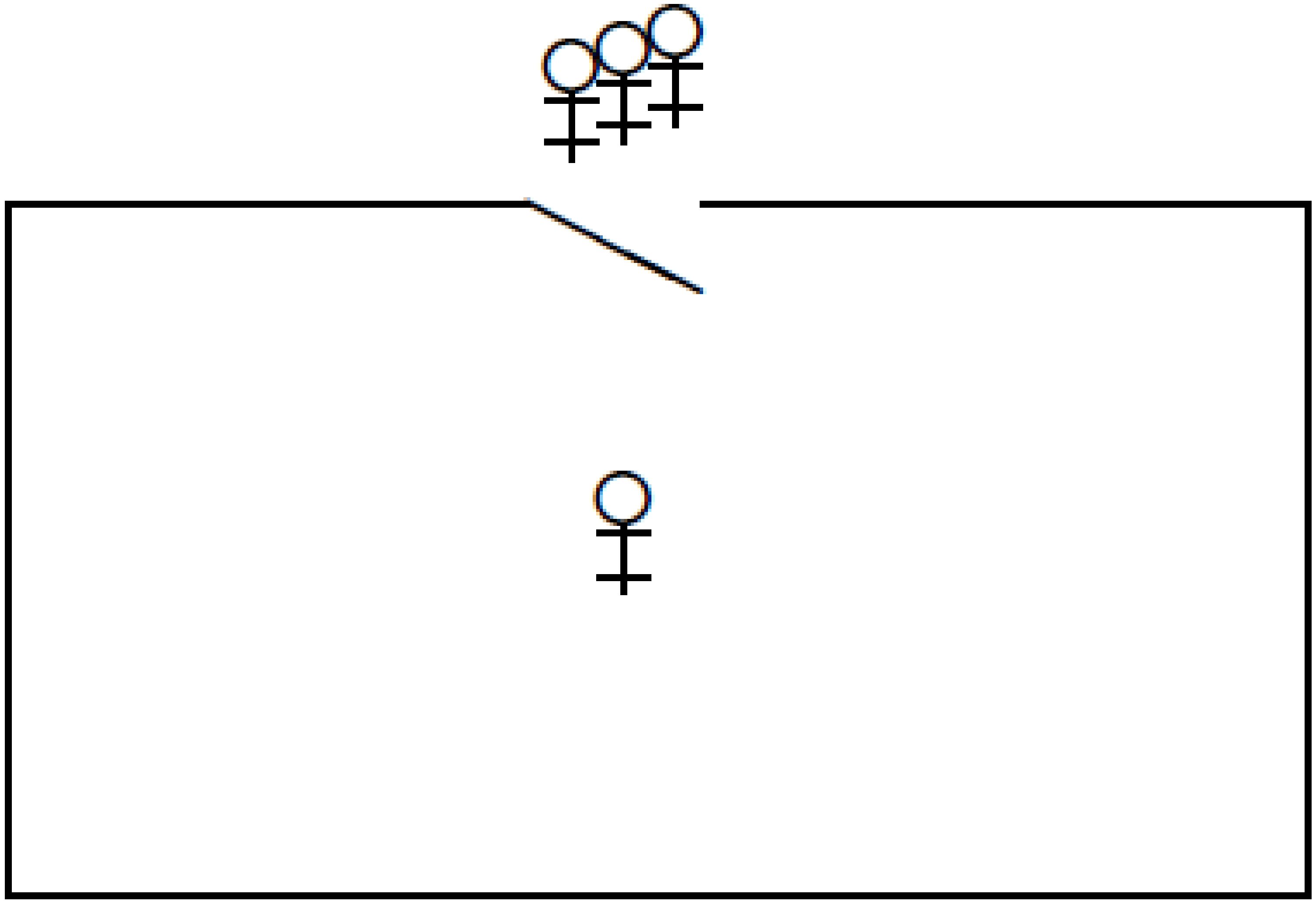
Algorithm 3.1: Critical section problem	
global variables	
p	**q**

local variables
loop forever
    non-critical section
    preprotocol
    critical section
    postprotocol
	local variables
	loop forever
	    non-critical section
	    preprotocol
	    critical section
	    postprotocol

Critical Section Problem

- The protocol may require local or global variables
- Critical section must progress
  - Once a process starts to execute the statements in a critical section, it must eventually finish execution of those statements
- The Non-Critical section need not progress
  - If the control pointer of a process is in its non-critical section, the process may terminate or enter an infinite loop

Critical Section Problem



First Attempt

- await turn = 1 waits until the condition turn = 1 becomes true
- This can be implemented by a busy-wait loop

Algorithm 3.2: First attempt			
integer turn $\leftarrow$ 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 $\leftarrow$ 2	q4:	turn $\leftarrow$ 1

Correctness

Algorithm 3.3: History in a sequential algorithm
integer a ← 1, b ← 2
p1: Millions of statements
p2: a ← (a+b)*5
p3: . . .

Algorithm 3.4: History in a concurrent algorithm	
integer a ← 1, b ← 2	
p	**q**
p1: Millions of statements	q1: Millions of statements
p2: a ← (a+b)*5	q2: b ← (a+b)*5
p3: . . .	q3: . . .

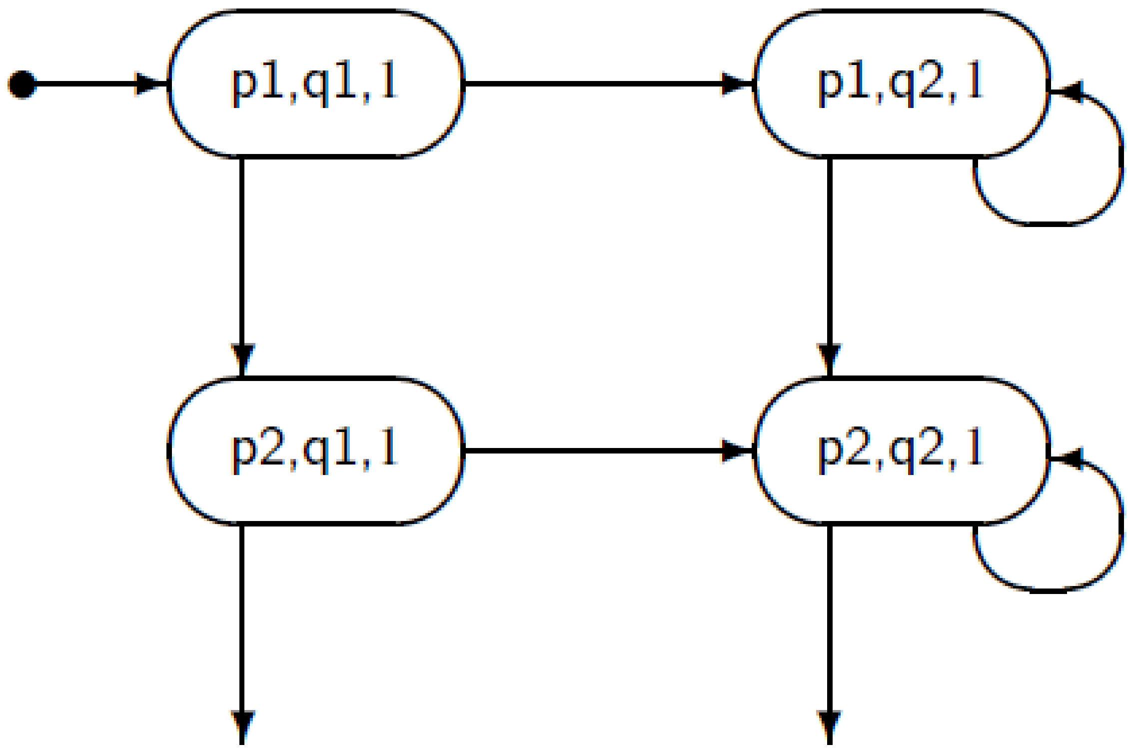
Correctness

- Sequential
  - $s_i = (p2, 10, 20)$ and $s_{i+1} = (p3, 150, 20)$

- Concurrent
  - $s_i = (p2, q2, 10, 20)$
  - $s^p_{i+1} = (p3, q2, 150, 20)$ or $s^q_{i+1} = (p2, q3, 10, 150)$

- The set of reachable states are the only states that can appear in any computation
- To check correctness, it is only necessary to examine the set of reachable states and the transitions among them

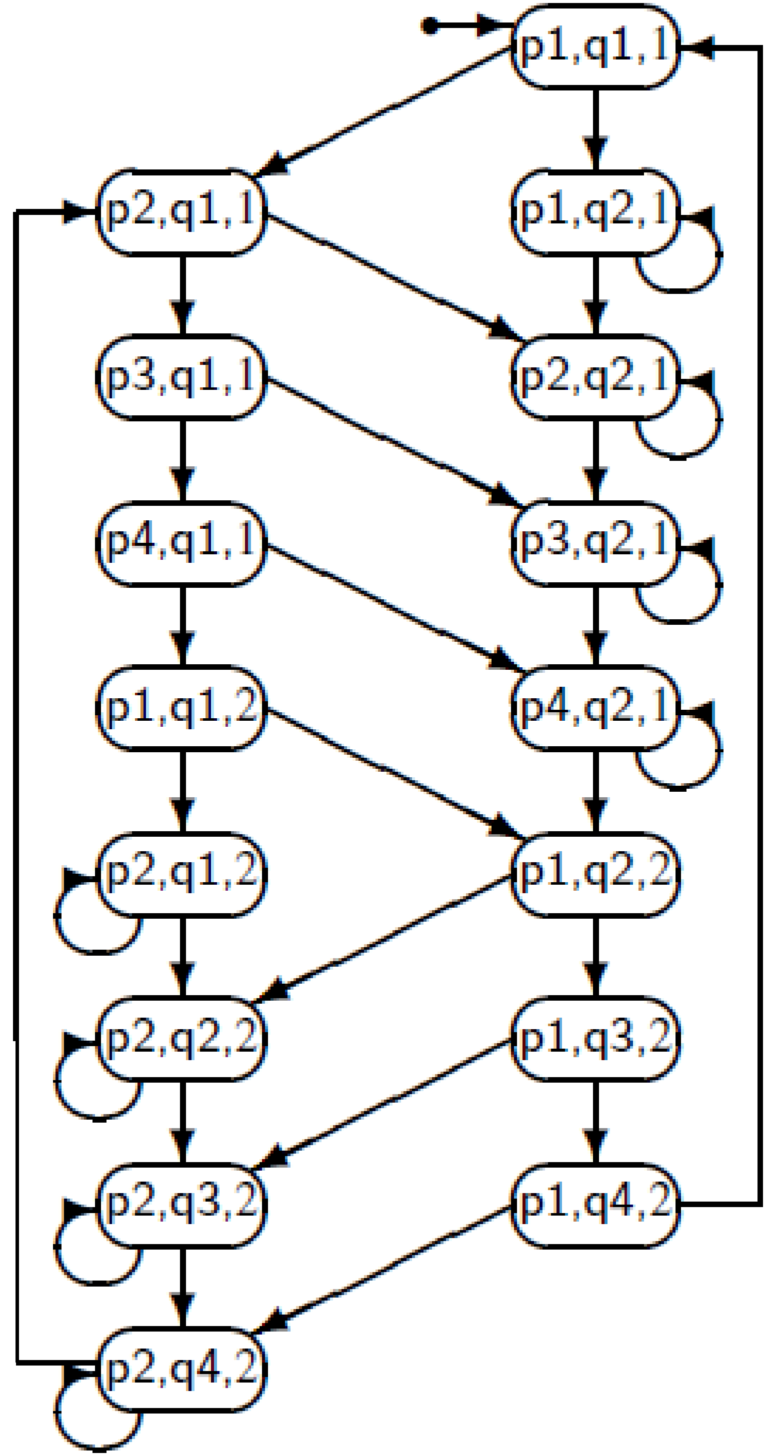
State Diagram – First steps



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

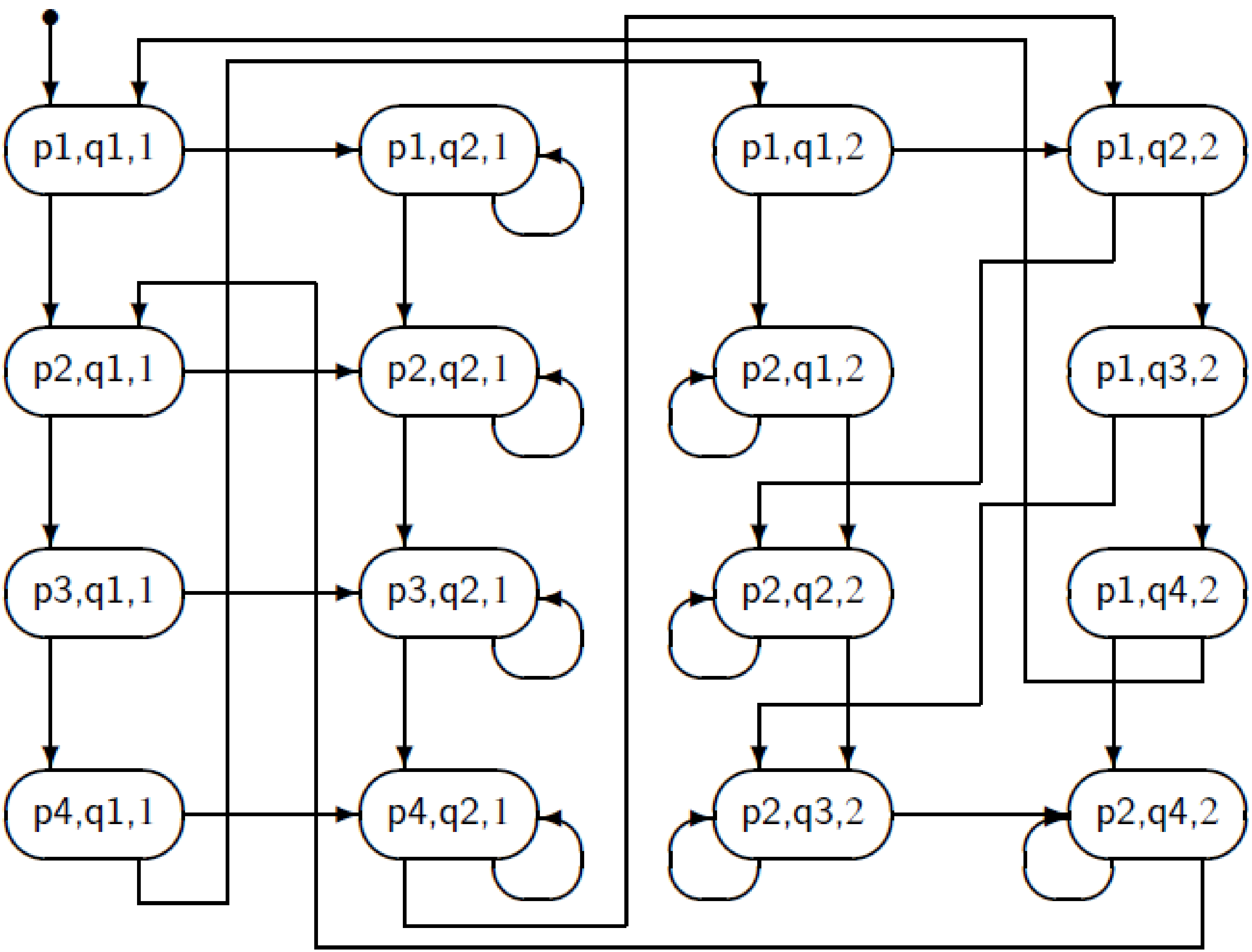
Sixteen steps

- (p3, q3,1) or (p3, q3,2) do not occur – mutual exclusion property holds



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

Alternate Layout for First Attempt

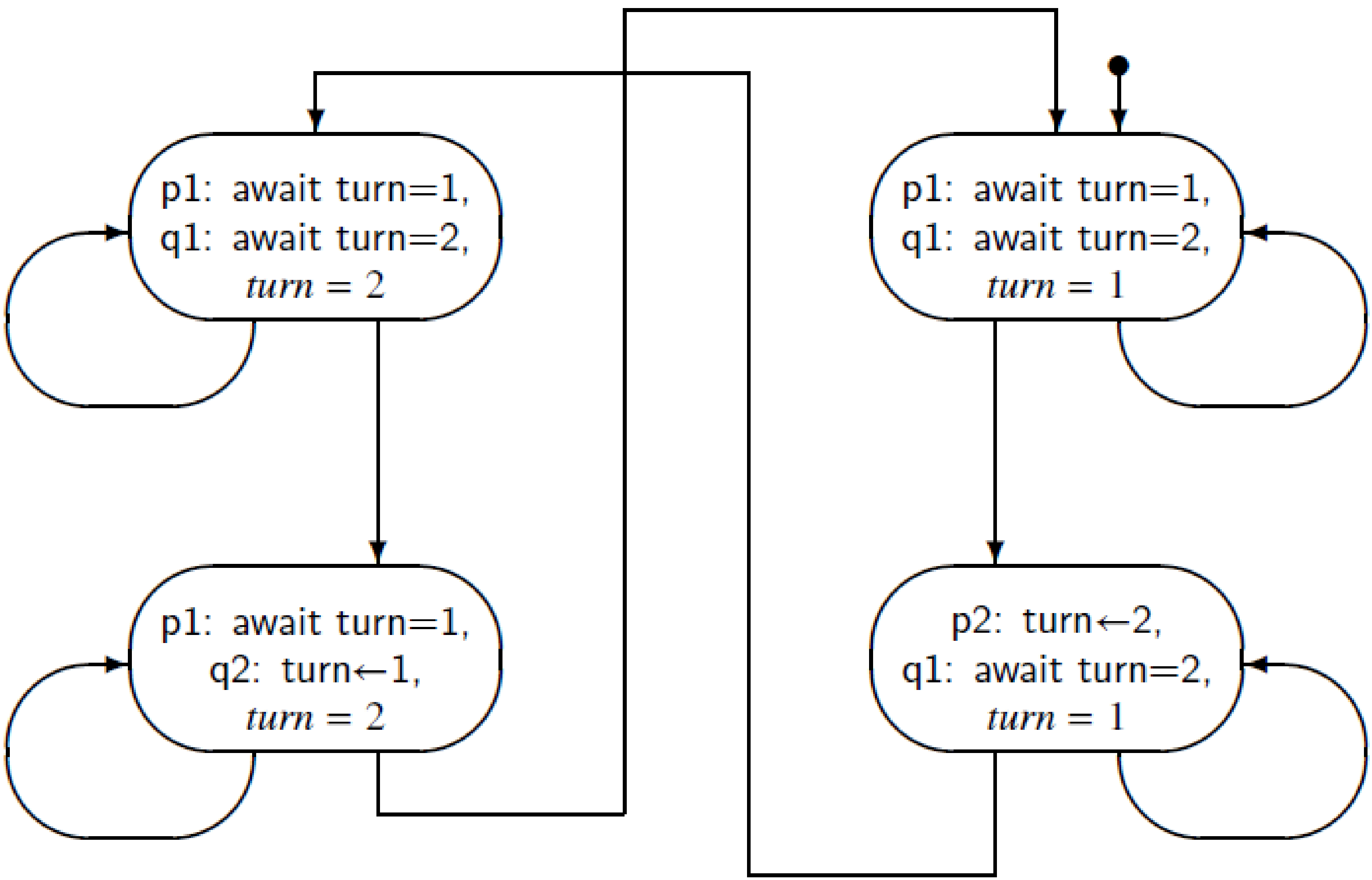


First Attempt (Abbreviated)

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

Algorithm 3.2: First attempt	
integer turn $\leftarrow$ 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 $\leftarrow$ 2	q4: turn $\leftarrow$ 1

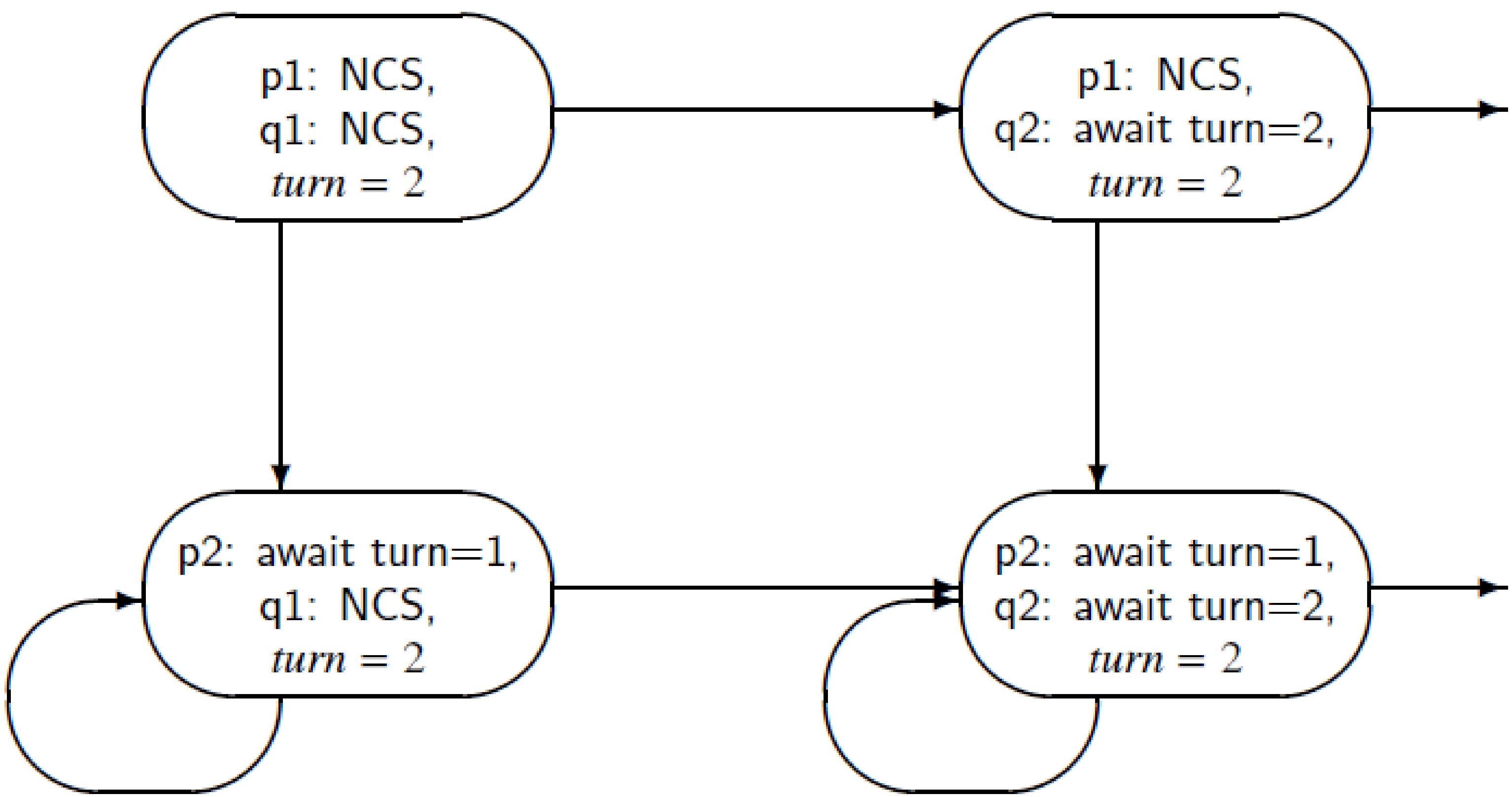
State Diagram – First Attempt (Abbv)



Correctness of First Attempt

- Proof of Mutual exclusion is immediate from the state diagram
- Proof of freedom from deadlock
  - If ***some*** processes are trying to enter their critical sections, then ***one*** of them must eventually succeed
- Proof of freedom from starvation
  - There is always some process holding the permission resource, so some process can always enter the CS ensuring there is no deadlock
  - If the process holding the permission resource remains indefinitely in its NCS, other process will never receive the resource and will never enter CS

State Diagram – Fragment



Second Attempt

- First attempt – both processes set and tested a single variable.
- If one process dies, other is blocked
- Each process is now given its own variable
- want***i*** is true from step where process **i** wants to enter its critical section until it leaves
- await statements ensure that a process does not enter its CS while another process has its flag set

Second Attempt

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

Second Attempt

- If a process halts in its critical section, the value of its variable want will remain false and the other process will always succeed in immediately entering the critical section
- Solves the problem of *starvation*
- But as we move ahead, we see that *mutual exclusion* property is not satisfied.

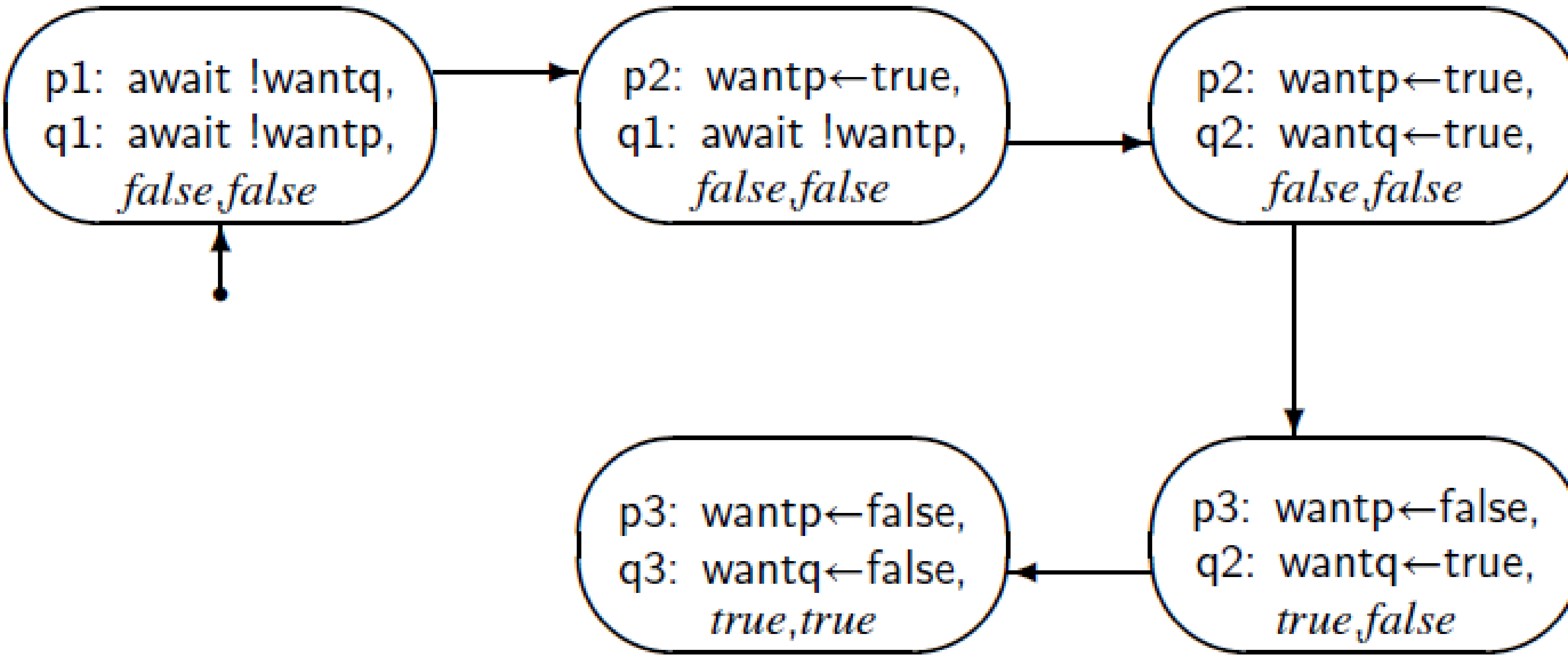
Second Attempt (Abbreviated)

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

Tabular Form

Process p	Process q	wantp	wantq
p1: await wantq=false	q1: await wantp=false	*false*	*false*
p2: wantp←true	**q1: await wantp=false**	*false*	*false*
p2: wantp←true	q2: wantq←true	*false*	*false*
p3: wantp←false	**q3: wantq←true**	*true*	*false*
p3: wantp←false	q3: wantq←false	*true*	*true*

State Diagram – Fragment



Second Attempt

- To prove that mutual exclusion holds, it must be checked that no forbidden state appears in any scenario
- If mutual exclusion does in fact hold, we need to construct the full state diagram for the algorithm, because every path in the diagram is a scenario
- Every state must be examined to make sure it is not a forbidden state
- We can stop construction if a forbidden state is encountered.

Third Attempt

- Second attempt – variables want are intended to indicate when a process is in its critical section
- Once a process has completed its await, it cannot be prevented from entering its CS
- The state reached after await but before assignment to want is effectively part of CS, but value of want does not indicate this

Third Attempt

- Recognizes that the await statement should be part of the critical section by moving the assignment to want before the await
- The construction of the state diagram shows that mutual exclusion is not violated
- However, the algorithm can *deadlock* as shown

Third Attempt

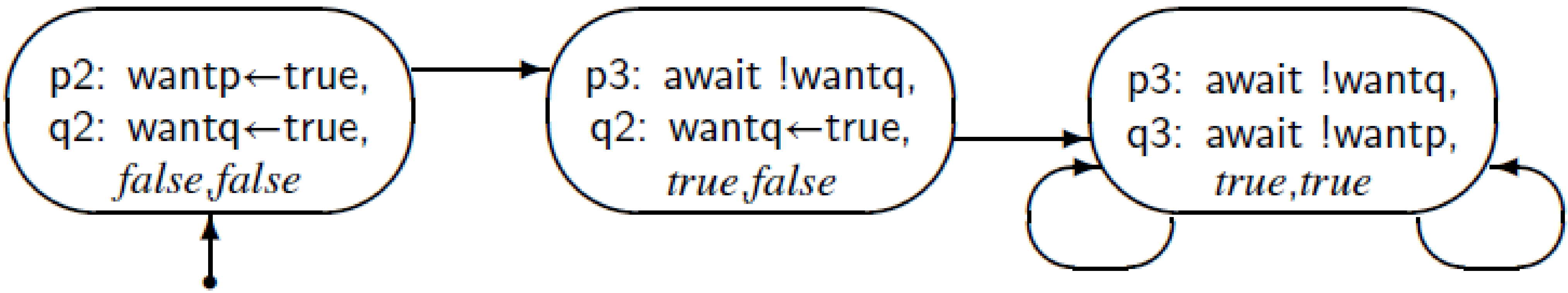
Algorithm 3.8: Third attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq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

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

Tabular Form

Process p	Process q	wantp	wantq
p1: non-critical section	q1: non-critical section	*false*	*false*
p2: wantp←true	**q1: non-critical section**	*false*	*false*
p2: wantp←true	q2: wantq←true	*false*	*false*
p3: await wantq=false	**q2: wantq←true**	*true*	*false*
p3: await wantq=false	q3: await wantp=false	*true*	*true*

State Diagram – Fragment



Deadlock and Livelock

- Term deadlock is usually used with a frozen computation where nothing whatsoever is being computed
- A scenario where several processes are actively executing statements, but nothing useful gets done is called livelock

Livelock

- Attempting to use resource preemption approaches to preventing deadlock may cause livelock: Processes don’t deadlock, but fail to make progress either
- A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result.
- As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work.
- *Deadlock:* "Me first, Me first” *Livelock:* "You first, You first"

Livelock

- Wayne and Larry want to listen to a CD on CD player
- Larry has the CD but wants the CD player
- Wayne has the CD player but wants the CD
- Wayne steals (i.e. preempts) the CD from Larry, meanwhile Larry steals the CD player from Wayne
- Now, Wayne has the CD and Larry has the CD player
- Wayne steals the CD player from Larry, meanwhile Larry steals the CD from Wayne
- Now, Wayne has the CD player and Larry has the CD
- . . .
- **Livelock**

Deadlock vs Livelock

- **Deadlock** is a condition in which a task waits indefinitely for conditions that can never be satisfied
  - task claims exclusive control over shared resources
  - task holds resources while waiting for other resources to be released
  - tasks cannot be forced to relinquish resources
  - a circular waiting condition exists
- **Livelock** conditions can arise when two or more tasks depend on and use the some resource causing a circular dependency condition where those tasks continue running forever, thus blocking all lower priority level tasks from running

Deadlock vs Livelock

- Deadlock
  - It happens when a process waits for another one who is using some needed resource to finish with it, while the other process also wait for the first process to release some other resource.
- Livelock
  - A Livelock looks like a deadlock in the sense that two (or more) processes are blocking each others. But with the livelock, each process is waiting “actively”, trying to resolve the problem on its own (like reverting back its work and retry).

Deadlock vs Livelock

- A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

Fourth Attempt

- Third attempt – when a process sets a variable want to be true, not only does it indicate its intension to enter its critical section, but also insists on its right to do so
- Deadlock occurs when both processes simultaneously insist on entering their CS
- Requires a process to give up its intention to enter the CS if it discovers that it is contending with the other process

Fourth Attempt

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

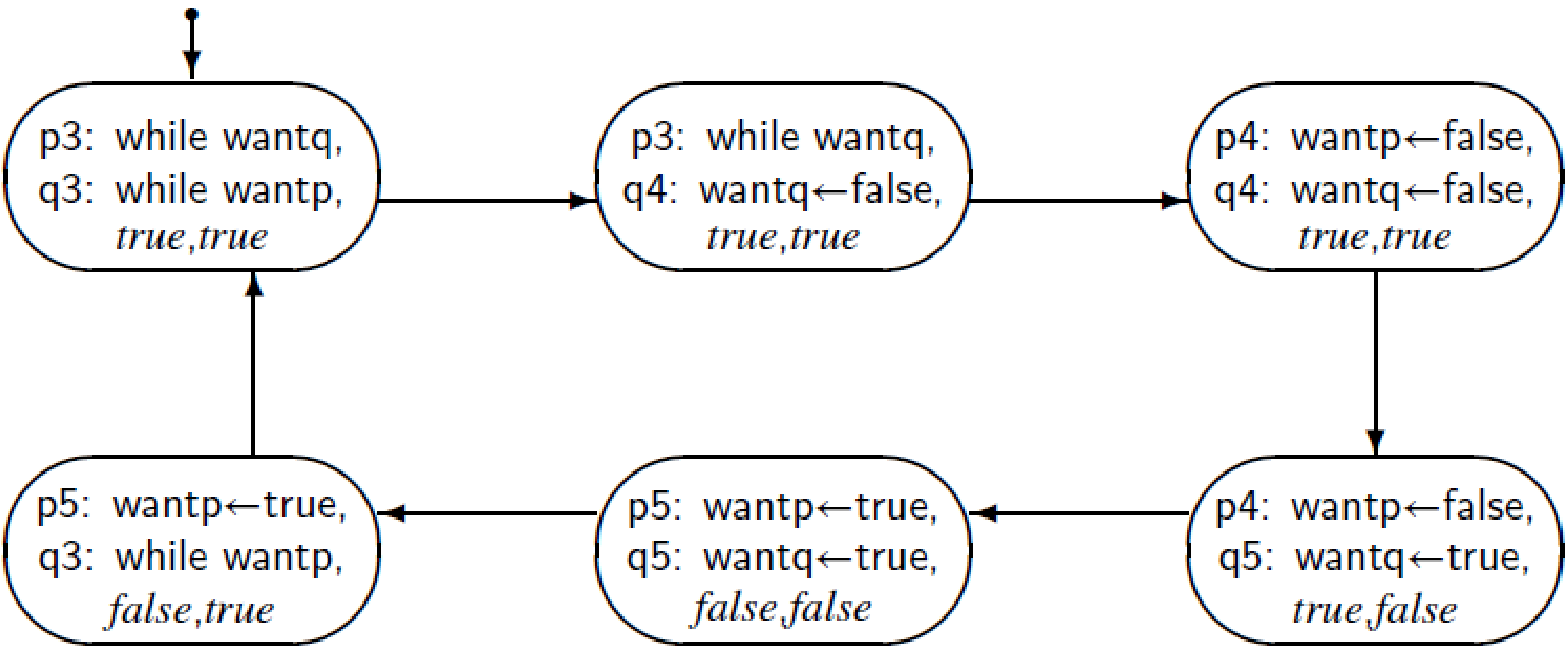
Fourth Attempt

- Statements p4 and p5 will be meaningless in a sequential algorithm, but useful in concurrent algorithm
- Arbitrary interleaving from and between the two processes, second process may execute an arbitrary number of statements between the two assignment statements to wantp
- When process p relinquishes the attempt to enter the critical section by resetting wantp to false, process q may now execute the wait statement and succeed in entering the CS

Fourth Attempt

- A state diagram will show that the mutual exclusion property holds and that there is no deadlock
- Scenario for starvation exists as shown in figure in next slide
- If interleaving is perfect, where the execution of a statement of process q is always followed by the execution of an equivalently numbered statement of process p, both the processes are starved

State Diagram – Fragment



Dekker’s Algorithm

- Combination of first and fourth attempts
- First Attempt – explicitly passed the right to enter the Critical Section
- This caused processes to be closely coupled and prevented correct behavior in absence of contention
- Fourth Attempt – each process had its own variable which prevented problems in absence of contention
- Both processes however insist on entering CS in the presence of contention

Dekker's Algorithm

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

- Similar to fourth attempt
- *Right to insist on entering* rather than *Right to enter* is explicitly passed between the processes
- The individual variables ensure mutual exclusion

- Dekker’s algorithm is correct: it satisfies the mutual exclusion property
- Its free from starvation and deadlock

Complex Atomic Statements

- It is difficult to solve critical section problem by just load and store statements
- The difficulty disappears if an atomic statement can both load and store
- An atomic statement is defined as the execution of a few following statements with no possible interleaving between them
- test-and-set, exchange, fetch-and-add, compare-and-swap are examples used to solve critical section problems

Complex Atomic Statements

- Check correctness of first two
  - test-and-set
  - exchange

- Solve the critical section problem and verify the correctness also for the remaining two
  - fetch-and-add
  - compare-and-swap

Complex Atomic Statements

test-and-set (common, local) is

local $\leftarrow$ common

common $\leftarrow$ 1

Algorithm 3.11: Critical section problem with test-and-set	
integer common $\leftarrow$ 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 $\leftarrow$ 0	q5: common $\leftarrow$ 0

Complex Atomic Statements

exchange (a, b) is
  integer temp
  temp ← a
  a ← b
  b ← temp

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)

Complex Atomic Statements

fetch_and_add(common, local, x) is

local ← common

common ← common + x

compare_and_swap (common, old, new) is

integer temp

temp ← common

if common = old

  common ← new

return temp