Lesson 3

Critical Section Problem

Ch 3 [BenA 06]

Critical Section Problem

Solutions without HW Support

State Diagrams for Algorithms

Busy-Wait Solutions with HW Support

Mutual Exclusion Real World Example

Fig. Pesutuvan varaus

- How to reserve a laundry room?
  - Housing corporation with many tenants
- Reliable
  - No one else can reserve, once one reservation for given time slot is done
  - One can not remove other's reservations
- Reservation method
  - One can make decision independently (without discussing with others) on whether laundry room is available or not
  - One can have reservation for at most one time slot at a time
- People not needing the laundry room are not bothered
- One should not leave reservation on when moving out
- One should not lose reservation tokens/keys

mutual exclusion, i.e., mutex

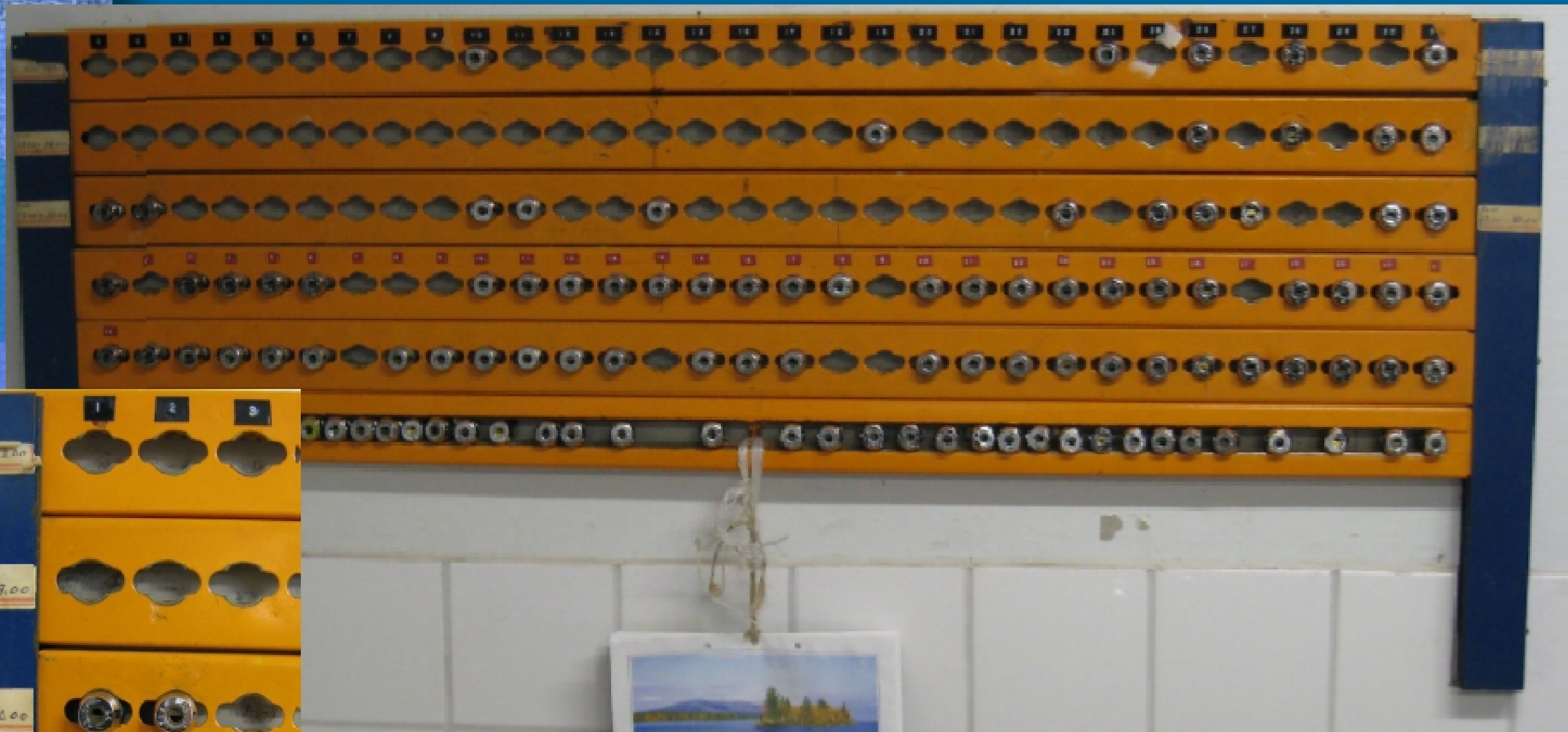
non-preemptive

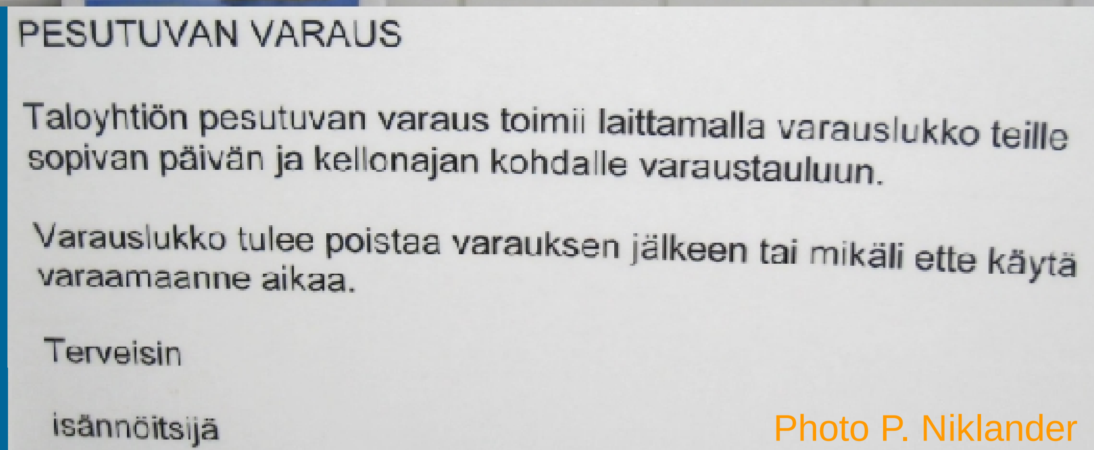
keskeytettämätön

distributed/centralized

no simultaneous resource possession

recovery?





PESUTUVAN VARAUS

Taloyhtiön pesutuvan varaus toimii laittamalla varauslukko teille sopivan päivän ja kellonajan kohdalle varaustauluun.

Varauslukko tulee poistaa varauksen jälkeen tai mikäli ette käytä varaamaanne aikaa.

Terveisin

isännöitsijä

Photo P. Niklander

Concurrent Indivisible Operations

- Echo

```
char out, in;  // globals
procedure echo {
  input (in, keyboard);
  out = in;
  output (out, display);
}
```

Process P1	Process P2
…	…
input (in,..);	…
	input(in,..);
out = in;	out = in;
…	output (out,..);
output(out,..);	

  – What if *out* and/or *in* local variables?

- Data base update
  – Name, id, address, salary, annual salary, …
- How/when/by whom to define granularity for indivisible operations?

Executing Many Processes Concurrently

- One CPU
  - Execute one process until
    - It requests a service that takes time to do
    - Some interrupt occurs and operating system gives execution turn to somebody else
      - E.g., time slice interrupt

aikaviipalekeskeytys

  - Another process may still run concurrently in GPU or some other I/O controller
- Many CPU's
  - Execute many processes always concurrently
  - Execution turn for one process may end any time (request service, or interrupt occurs)

Critical Section Problem

- Critical section (CS)
  - Code segment that only one process may be executing at a time
  - May also be set of code segments, and only one process may be executing at a time any code segment in that set
  - Not necessarily an atomic operation
    - Other processes may be scheduled, but they can not execute in (this) critical section
- Critical Section Problem (Mutex Problem)
  - How to guarantee that only one process at a time is executing critical section?

Discuss

Critical Section (CS) Solution

- Mutex (mutually exclusive code) solved
- No deadlock: someone will succeed
- No starvation (and no unnecessary delay)
  - Everyone succeeds eventually
- *Protocol* does not use common variables with CS actual work
  - Can use it's own local or shared variables

poissulkemisong. ratk.

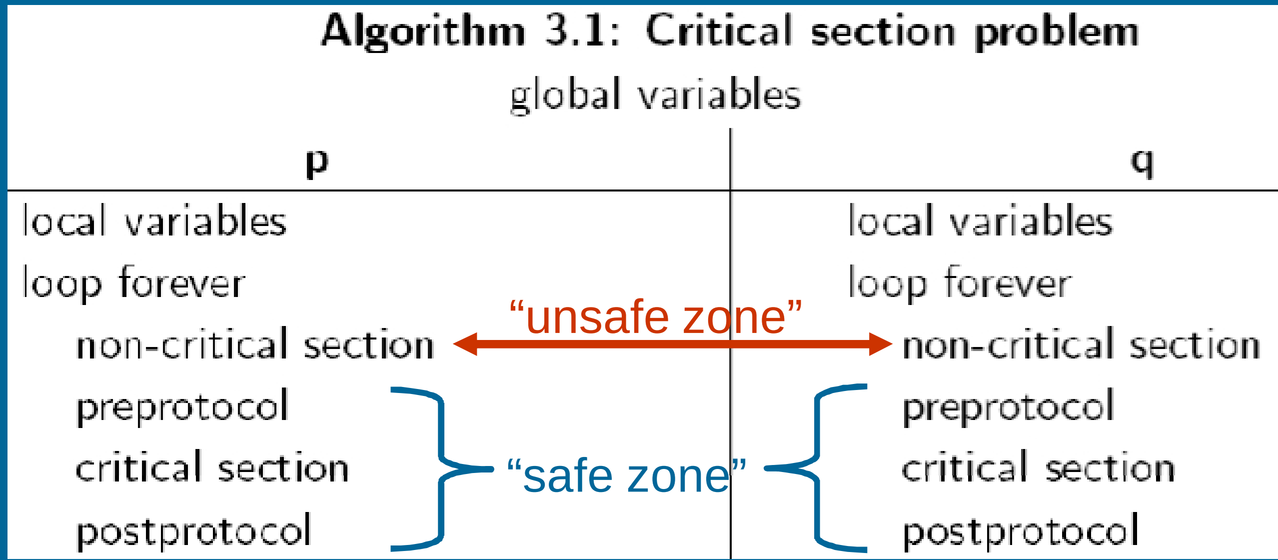
ei lukkiutumista

ei nälkiintymistä

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

Critical Section Assumptions



- Preprotocol and postprotocol have no common local/global variables with critical/non-critical sections
  - They do not disturb/affect each other
- Non-critical section may stall or terminate
  - Can not assume it to complete
- Critical section will complete (will not terminate or die)
  - Postprotocol eventually executed once critical section is entered
- Process will not terminate in preprotocol or postprotocol (!!!)
  - Process may terminate (die) only in non-critical section

Critical Section Solution

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

- How to prove correct or incorrect?
  - Mutex? (functional correct, one at a time in CS)
  - No deadlock? (eventually someone from many will get in)
  - No starvation? (eventually any specific one will get in)

“await condition” statement

- Pseudo language construct

“await until my turn”

- Implement <u>somehow</u> *waiting until given condition becomes true*
  - Use clever algorithms
    - Dekker, Peterson, …
  - Use hardware (HW) help – special instructions & data?
    - Interrupts, lock variables with busy wait loops, …
  - Use operating system (OS) – suspend process?
    - Semaphores, barrier operations, busy waits loops, …
    - Implemented using HW (or those clever algorithms)
  - Use programming language utilities?
    - Semaphores, monitor condition variables, barrier operations, protected object *when* statements, …
    - Implemented using OS
- Specifics discussed more later on

Correctness Proofs

- Prove incorrect
  - Come up with <u>one</u> scenario that does not work
    - Two processes execute in sync?
    - Some other unlikely scenario?

often non-trivial

- Prove correct
  - Heuristics: “I did not come up with any proofs (counterexample) for incorrectness and I am smart”
    - ⇨ I can not prove incorrectness
    - ⇨ It must be correct…

“easy”, unreliable

  - State diagrams
    - Describe algorithm with states:

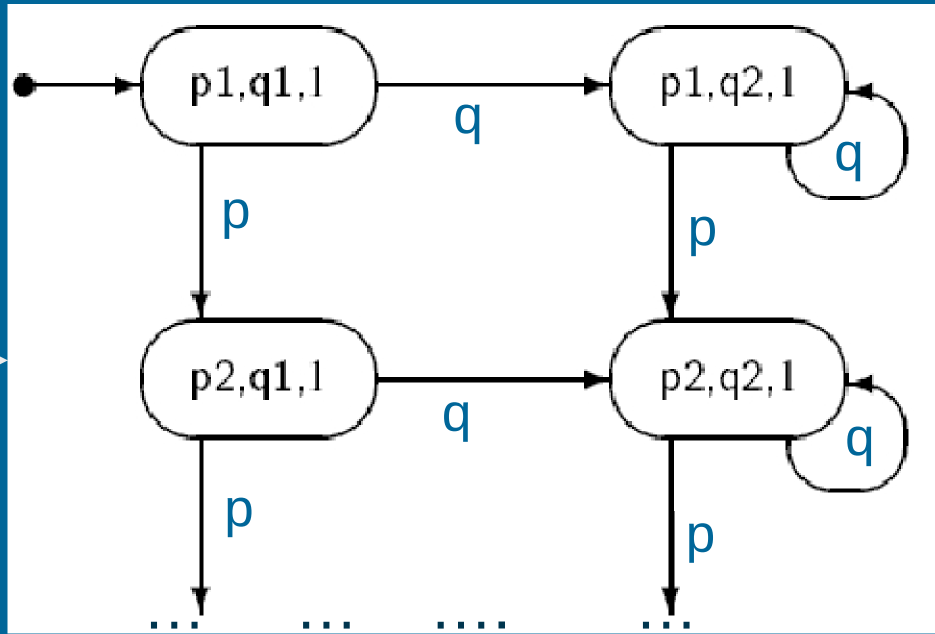
      { relevant control pointer (cp) values,
      relevant local/global variable values }
    - Analyze state diagrams to prove correctness

difficult, reliable

State Diagram for Alg. 3.2

Algorithm 3.2

- State {p_i, q_i, turn}
  - Control pointer p_i
  - Control pointer q_i
  - Global variable turn
  - 1st four states →
- Mutex ok
  - State {p3, q3, turn} not accessible in state diagram?
- No deadlock?
  - When many processes try concurrently, one will succeed
- No starvation?
  - Whenever any (one) process tries, it will eventually succeed

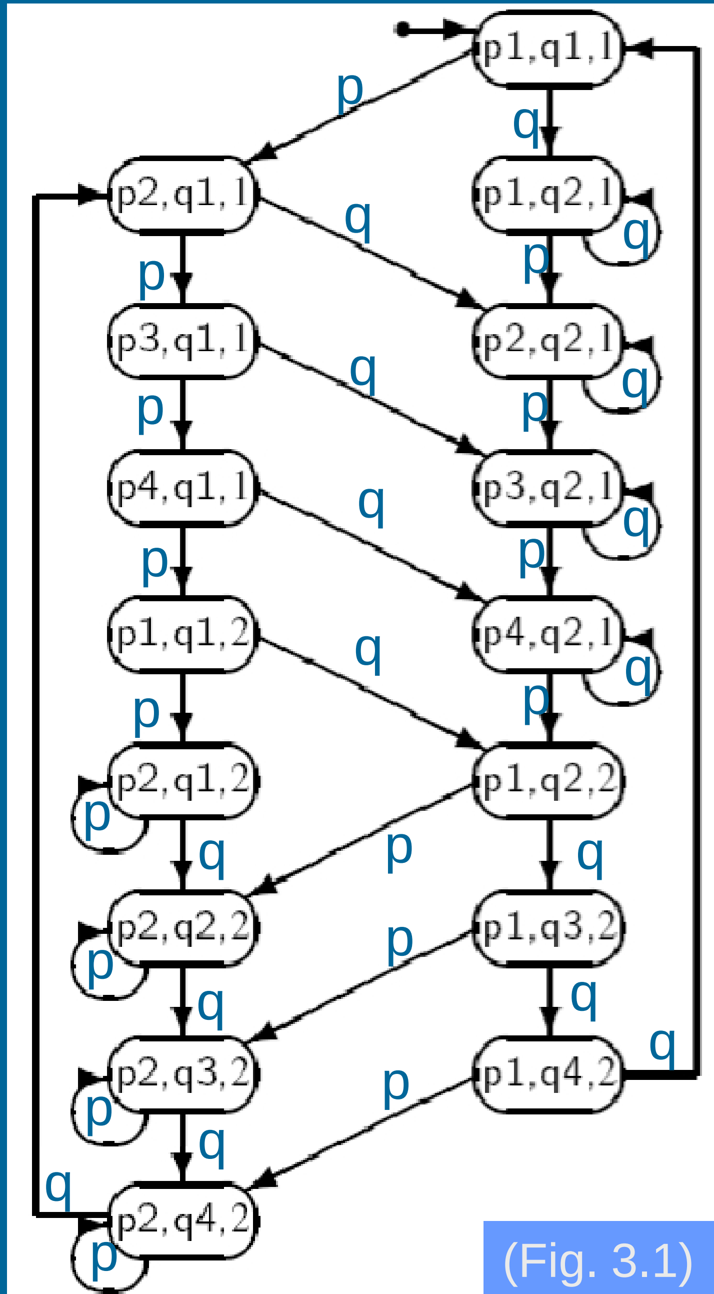


How to prove it?

State Diagram for Algorithm 3.2

Algorithm 3.2

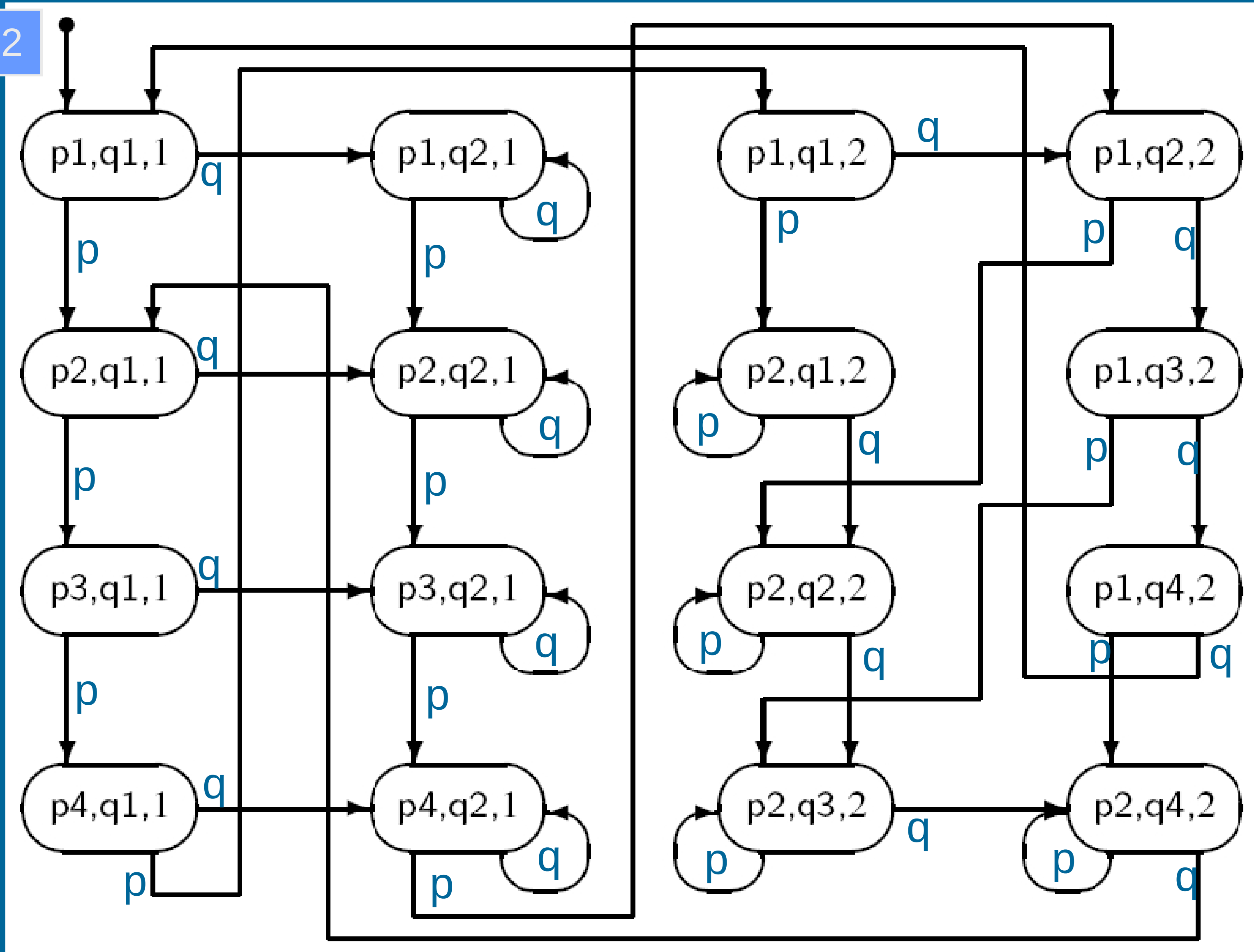
- Create complete diagram with all accessible states
- No states
  - {p3, q3, 1}
  - {p3, p3, 2}
- I.e., mutex secured proof!
- Problem:
  - Too many states?
  - Difficult to create
  - Difficult to analyze



(Fig. 3.1)

Alternate Layout for Full State Diagram

Alg. 3.2

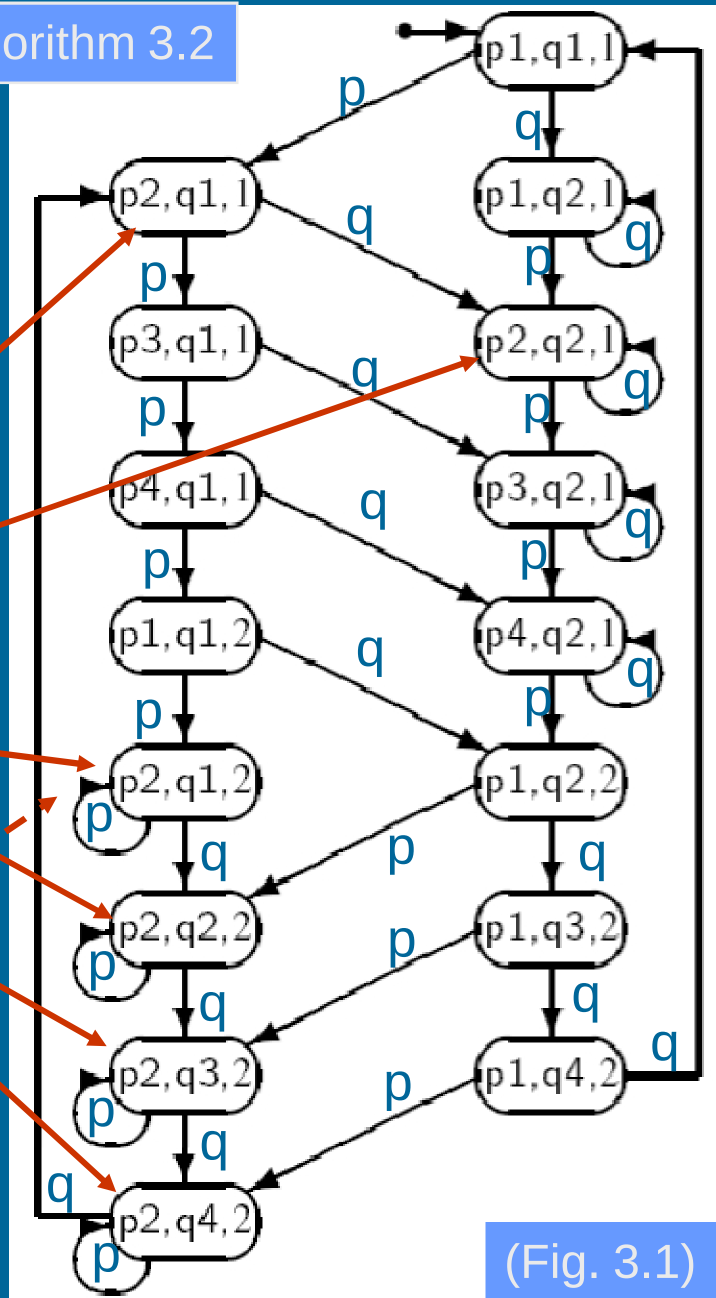


Correctness (3)

Algorithm 3.2

- Mutex?
  - Ok, no state {p3, q3, ??}
- No deadlock?
  - many try, one can always get in? (into a state with p3 or q3)
  - {**p2**, q1, 1}: P can get in
  - {**p2**, q2, 1}: P can get in
  - {**p2**, q1 tai q2, 2}:
    - Q can get in
  - {**p2**, q3 tai q4, 2}:
    - P can get in eventually
  - {pi, q2, ?} similarly. *q.e.d.*
- No starvation?
  - One tries, it will eventually get in?
  - {p2, q1, 2}
    - Q dies (ok to die in q1), P will starve! **Not good!**

All states with p2



(Fig. 3.1)

Reduced Algorithm for Easier Analysis

Proven erroneous!

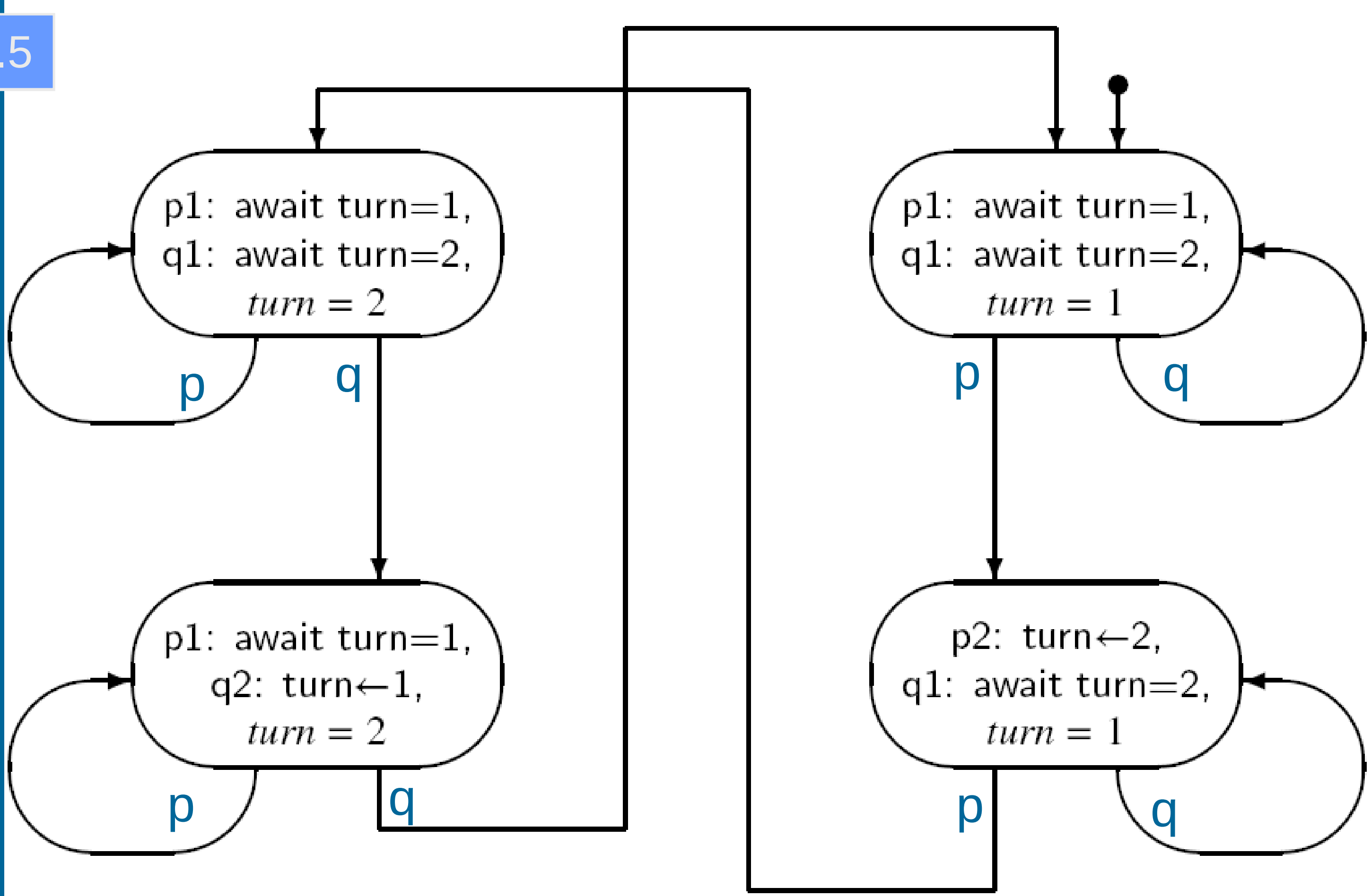
Algorithm 3.2: First attempt			
integer turn ← 1			
p		**q**	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	await turn = 1	q2:	await turn = 2
p3:	critical section	q3:	critical section
p4:	turn ← 2	q4:	turn ← 1

- Reduce algorithm to reduce number of states of state diagrams: leave irrelevant code out
  - Nothing relevant (for mutex) left out?

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

State Diagram for Reduced Algorithm

Alg. 3.5

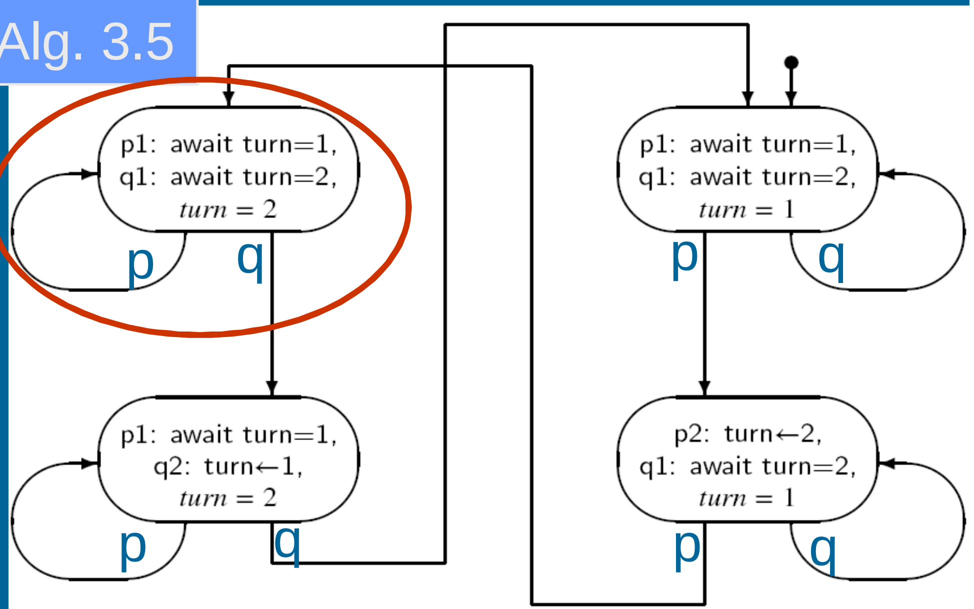


(Fig. 3.2)

- Much fewer states!

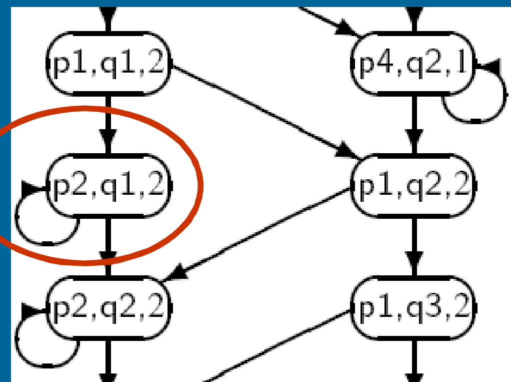
Correctness of Reduced Algorithm (2)

Alg. 3.5



- Mutex? OK
  - No state {p2, q2, turn}
- No deadlock: Some are trying, one may get in? OK
  - Top left (p & q trying): q will get in
  - Bottom left (p trying): q will eventually execute (assumption!)
  - Top & bottom right: mirror situation
- No starvation?
  - Tricky, reduced too much!
    - NCS combined with await
  - Look at original diagram
    - Problem if Q dies in NCS

Not OK



should be OK to die in NCS, but not OK to die in protocol

Critical Section Solution #2

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

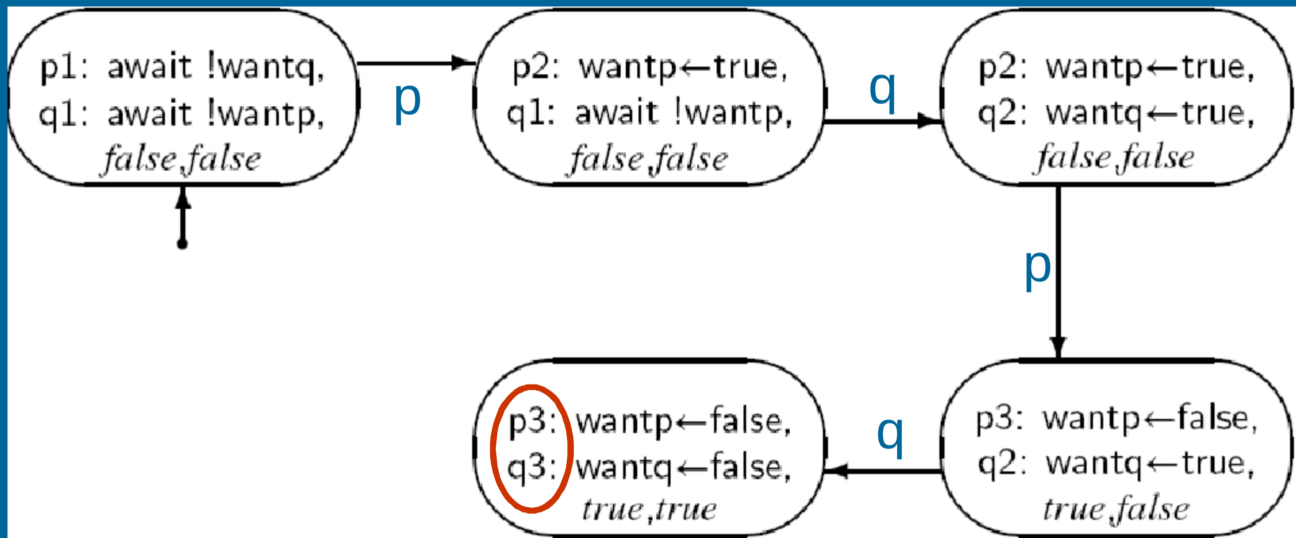
- Each have their own global variable *wantp* and *wantq*
  - True when process is in critical section
- Process dies in NCS?
  - Starvation problem ok, because it’s *want*-variable is false
- Mutex? Deadlock?

Attempt #2 Reduced

pro-to-col

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

- No mutex! {p3, q3, ?} reachable
  - Problem: p2 should be part of critical section (but is not!)



Critical Section Solution #3

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

- Avoid previous problem, mutex ok
- Deadlock possible: {p3, q3, wantp=true, wantq=true}
- Problem: cyclic wait possible, both insist their turn next
  - No preemption

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

- Avoid deadlock by giving away your turn if needed
- Mutex ok: P in p6 only if !wantq (⇨ Q is not in q6)
- Deadlock (livelock) possible:
  {p3, q3, …}→{p4, q4, …}→{p5, q5, …}
  - Unlikely but possible!
  - **Livelock**: both executing all the time, not waiting suspended
    - Neither one advances

elolukko

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

- Combine 1st and 4th attempt
- 3 global (mutex ctr) variables: shared *turn*, semi-private *want*'s
  - only one process writes to *wantp* or *wantq* (= semi-private)
- *turn* gives you the right to insist, i.e., priority
  - Used only when both want CS 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

Proof

- Mutex ok: P in p8 only if !wantq (⇨ Q can not be in q8)
- No deadlock, because P or Q can continue to CS from {p3, q3, ..}
- No starvation, because
  - If in {p6, …}, then eventually {p6, q9, …} and {..., q10, ...}
  - Next time {p3, …} or {p4, …} will lead to {p8, …}

Proven correct!

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

- mutex with **no HW-support needed, need only shared memory**
- Bad: complex, many instructions
  - Must execute each instruction at a time, in this order
    - Will not work, if compiler optimizes code too much!
  - In simple systems, can do better with HW support
    - Special machine instructions to help with this problem

Discuss

Mutex with HW Support

- Specific machine instructions for this purpose
  - Suitable for many situations
  - Not suitable for all situations
- Interrupt disable/enable instructions
- Test-and-set instructions
  - Other similar instructions
- Specific memory areas
  - Reserved for concurrency control solutions
  - Lock variables (for test-and-set) in their own cache?
    - Different cache protocol for lock variables?
    - Busy-wait without memory bus use?

Disable
-- Critical Section --
Enable

Lock (L)
-- Critical Section --
Unlock (L)

Disable Interrupts

- Environment
  - All (competing) processes on run on the same processor (core?)
  - Not for multiprocessor systems
    - Disabling interrupts does it only for the processor executing that instruction
- Disable/enable interrupts
  - Prevent process switching during critical sections
    - Good for only very short time
    - Prevents also (other) operating system work (in that processor) while in CS

Disable
Enable

Can not execute this, if not running…

Disable
-- CS --
Enable

Disable
-- CS --
Enable

Test-and-set Lock Variables

Lukkomuuttujat

- Environment
  - All processes with shared memory
  - Should have multiple processors
  - Not very good for uniprocessor systems (or synchronizing processes running on the same processor)
    - Wait (**busy-wait**) while holding the processor!
- Test-and-set *machine instruction*
  - Indivisibly read old value and write new value (complex mem-op)

```
Test-and-set (common, local)
        local ← common   ; read old state
        common ← 1       ; mark reserved
```

shared   local

```
Test-and-set (shLock, locked);
while (locked)
    Test-and-set (shLock, locked);
-- CS --
shLock = 0;
```

```
Test-and-set (shLock, locked);
while (locked)
    Test-and-set (shLock, locked);
-- CS --
shLock = 0;
```

Other Machine Instructions for Synchronization Problem Busy-Wait Solutions

- Test-and-set

```
Test-and-set (common, local)
        local ← common    ; read state
        common ← 1        ; mark reserved
```

- Exchange

```
Exchange (common, local)
        local ↔ common  ; swap values
```

- Fetch-and-add

```
Fetch-and-add (common, local, x)
        local ← common          ; read state
        common ← common+x ; add x
```

- Compare-and-swap

```
int Compare-and-swap (common, old, new)
        return_val ← common
        if (common == old)
                common ← new
```

Use all in busy-wait loops

"read-modify-write" memory bus transaction (local in HW register)

"read-after-write" memory bus transaction may also be used

Lock variables and busy wait

- Need shared memory
- Use processor while waiting
  - Waste of a processor?
  - Not so smart with just one processor
    - Busy waits suspended when *time slice* ends (i.e., when OS time slice interrupt occurs)
  - Should wait only a very short time
    - Unless plenty of processors
  - Real fast resume when wait ends
    - Good property in some environments

Summary

- Critical section (CS)
- Critical Section Problem
- Solutions without HW Support
- State Diagrams for Algorithms
- Busy-Wait Solutions with HW Support