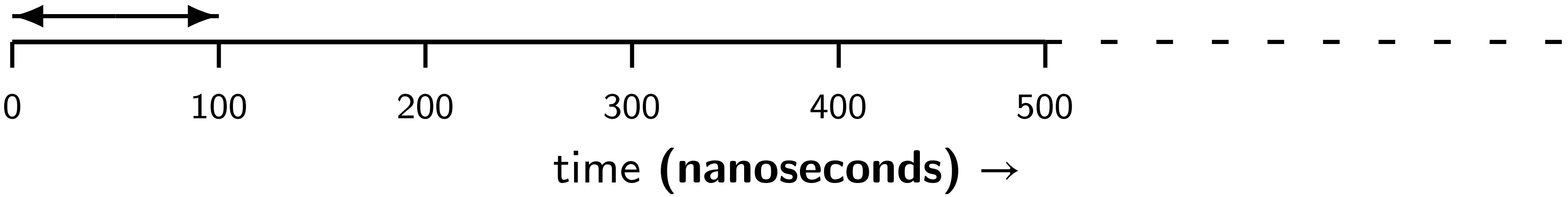
M. Ben-Ari

Principles of Concurrent and Distributed Programming

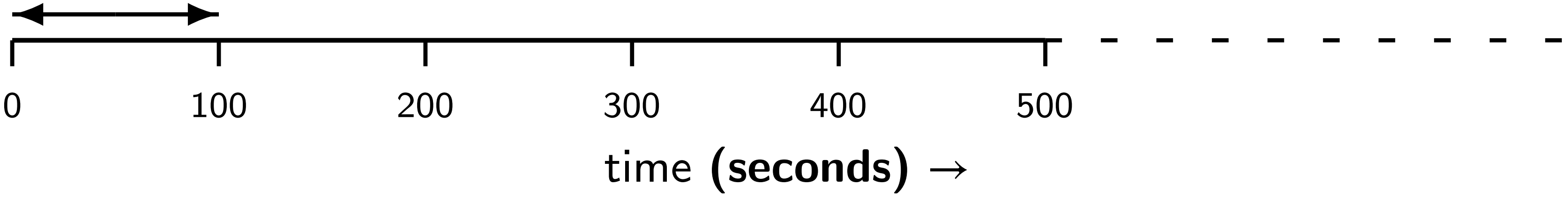
Second Edition

Addison-Wesley, 2006

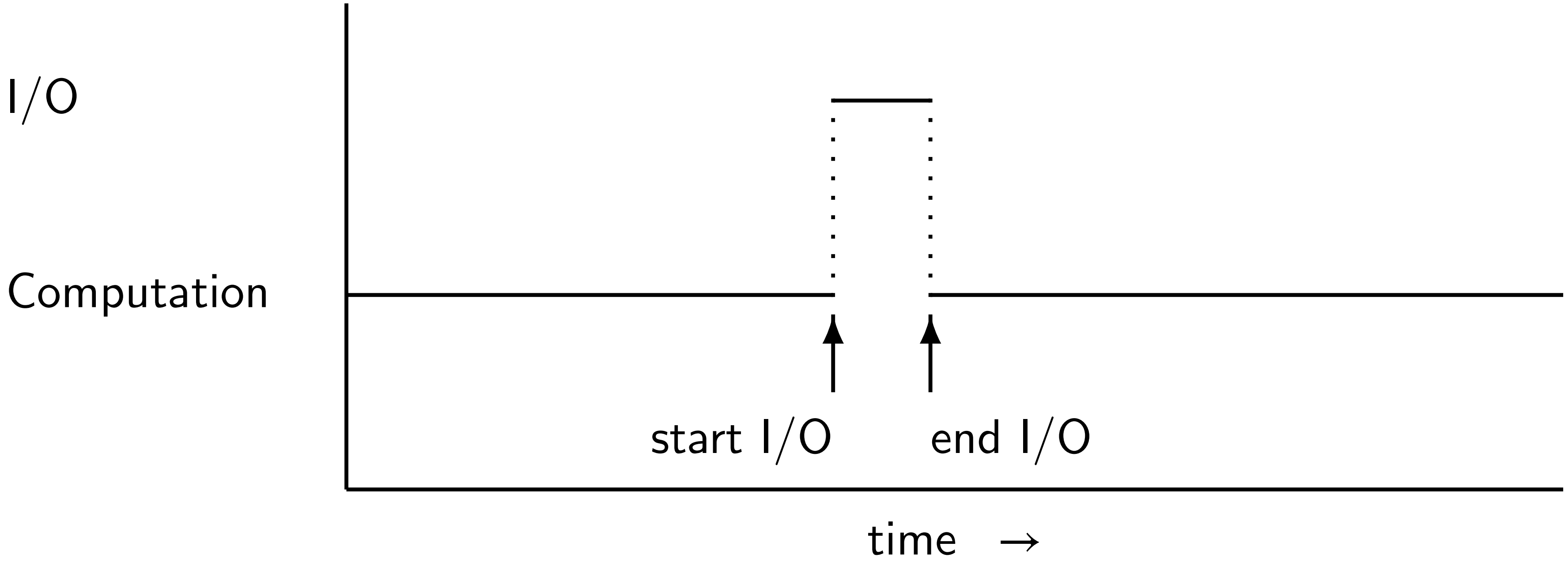
Computer Time



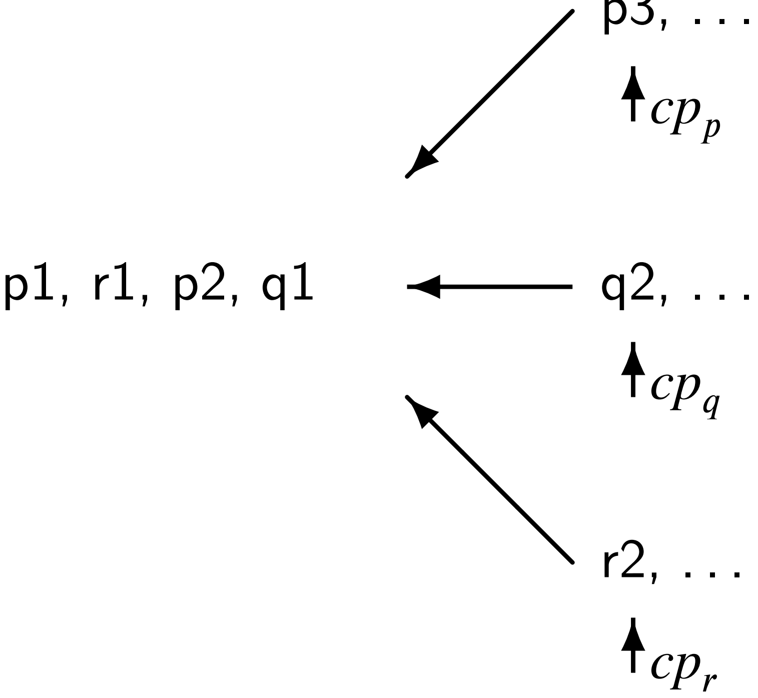
Human Time



Concurrency in an Operating System



Interleaving as Choosing Among Processes



p3, . . .

$\uparrow cp_p$

p1, r1, p2, q1

q2, . . .

$\uparrow cp_q$

r2, . . .

$\uparrow cp_r$

Possible Interleavings

p1→q1→p2→q2,

p1→q1→q2→p2,

p1→p2→q1→q2,

q1→p1→q2→p2,

q1→p1→p2→q2,

q1→q2→p1→p2.

Algorithm 2.1: Trivial concurrent program

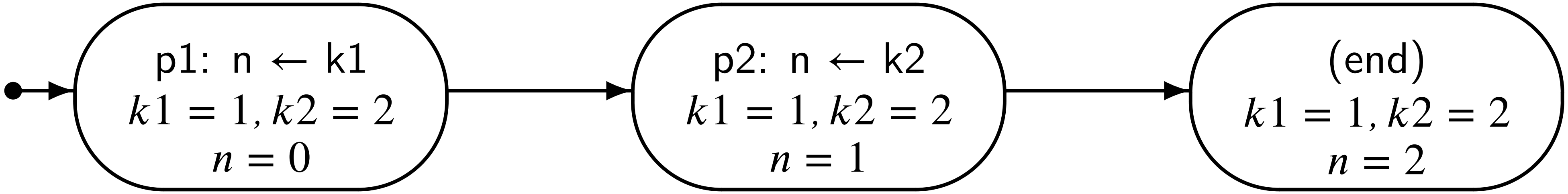
integer n ← 0

p	**q**
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

Algorithm 2.2: Trivial sequential program

```
integer n ← 0
    integer k1 ← 1
    integer k2 ← 2
p1: n ← k1
p2: n ← k2
```

State Diagram for a Sequential Program

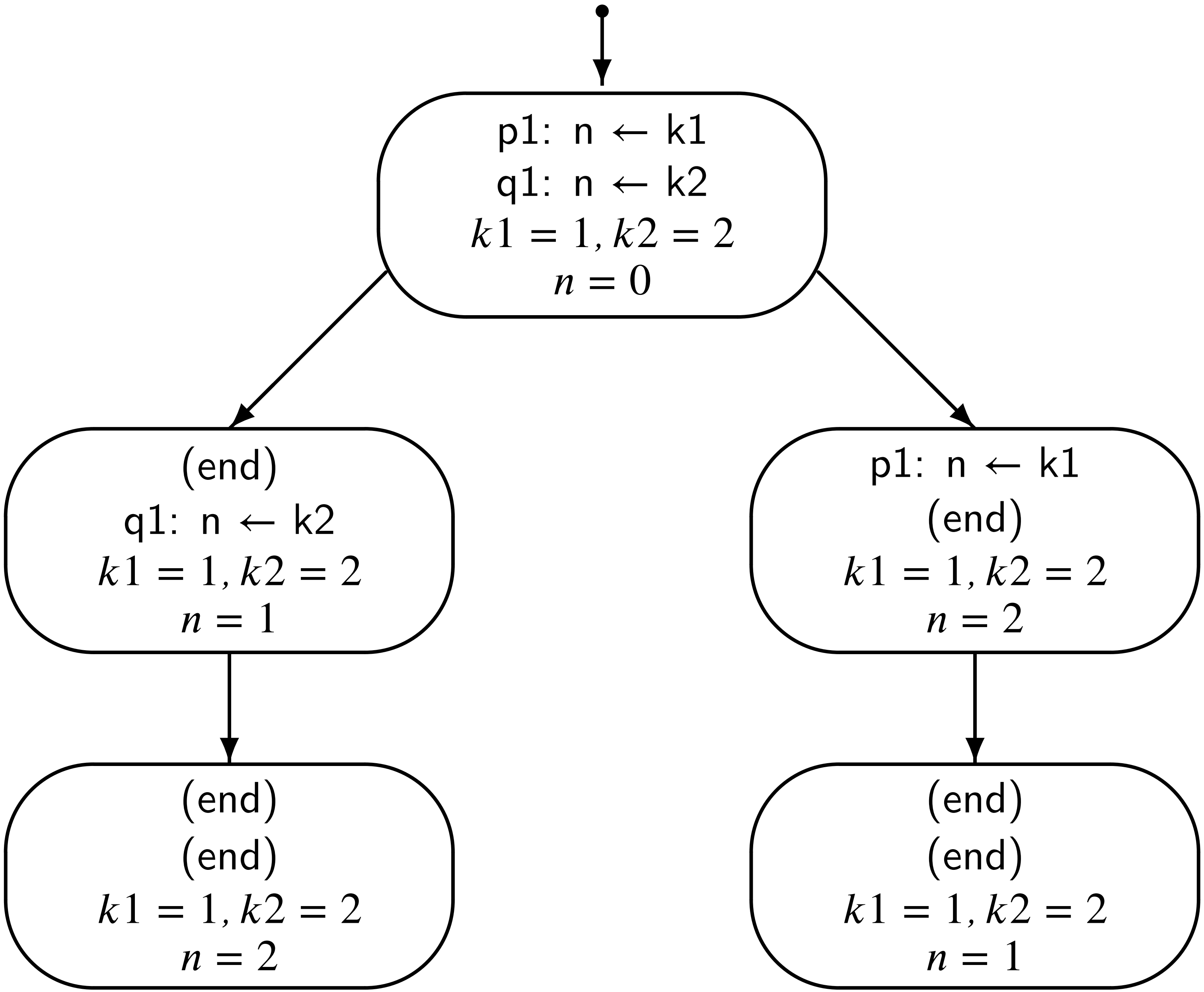


p1: n ← k1
$k1 = 1, k2 = 2$
$n = 0$

p2: n ← k2
$k1 = 1, k2 = 2$
$n = 1$

(end)
$k1 = 1, k2 = 2$
$n = 2$

State Diagram for a Concurrent Program



p1: n ← k1
q1: n ← k2
$k1 = 1, k2 = 2$
$n = 0$

(end)
q1: n ← k2
$k1 = 1, k2 = 2$
$n = 1$

p1: n ← k1
(end)
$k1 = 1, k2 = 2$
$n = 2$

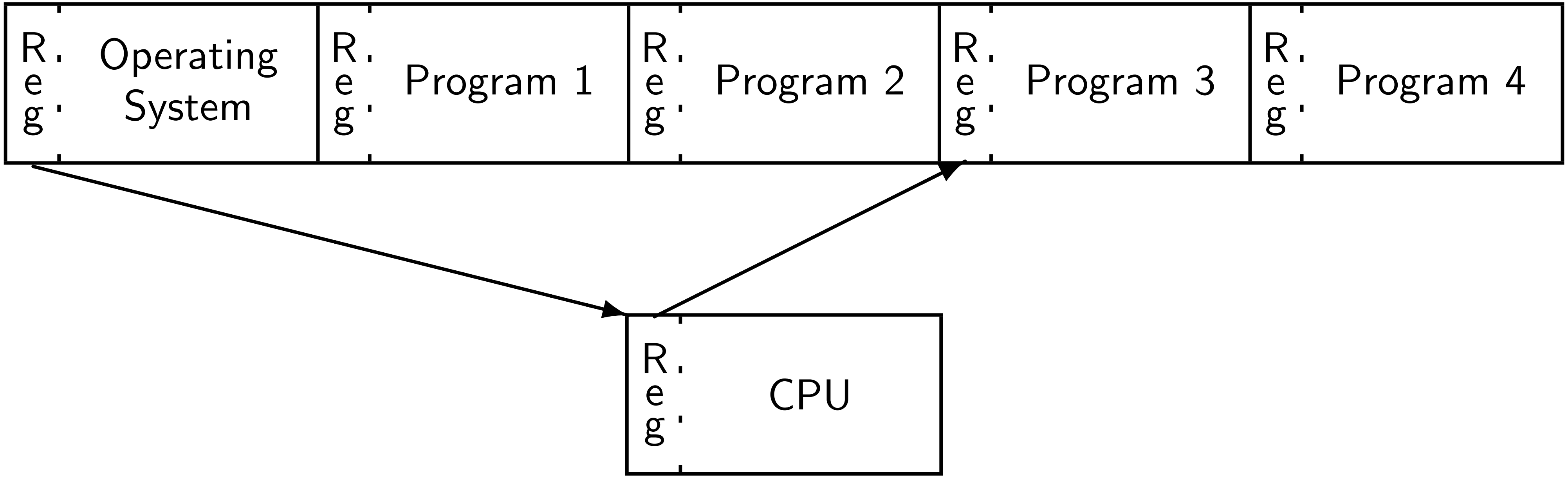
(end)
(end)
$k1 = 1, k2 = 2$
$n = 2$

(end)
(end)
$k1 = 1, k2 = 2$
$n = 1$

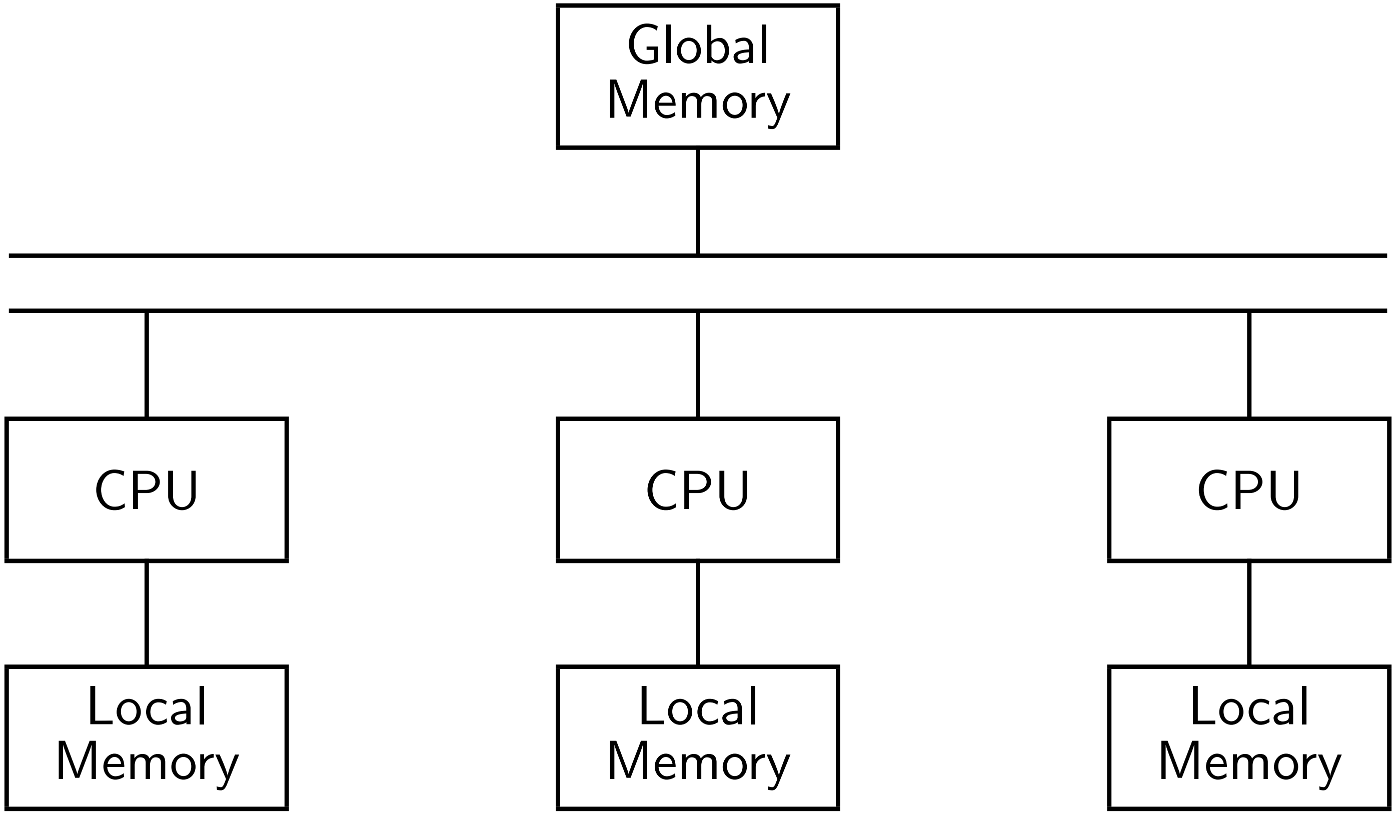
Scenario for a Concurrent Program

Process p	Process q	n	k1	k2
p1: n←k1	q1: n←k2	0	1	2
(end)	**q1: n←k2**	1	1	2
(end)	(end)	2	1	2

Multitasking System



Multiprocessor Computer



Global Memory

CPU

CPU

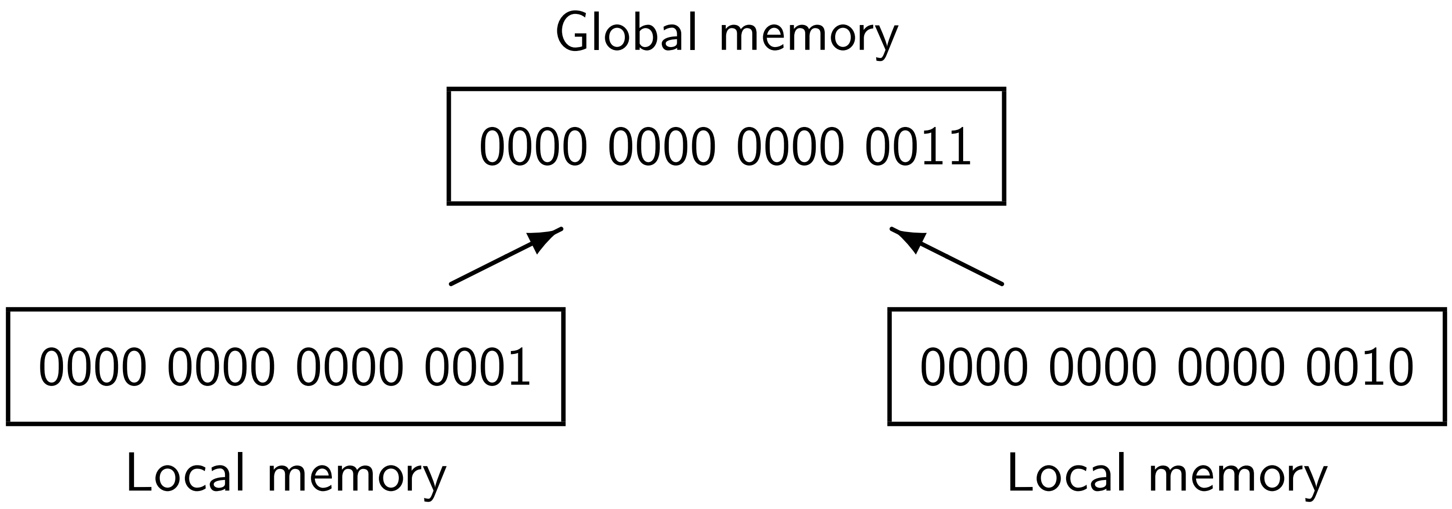
CPU

Local Memory

Local Memory

Local Memory

Inconsistency Caused by Overlapped Execution



Global memory

0000 0000 0000 0011

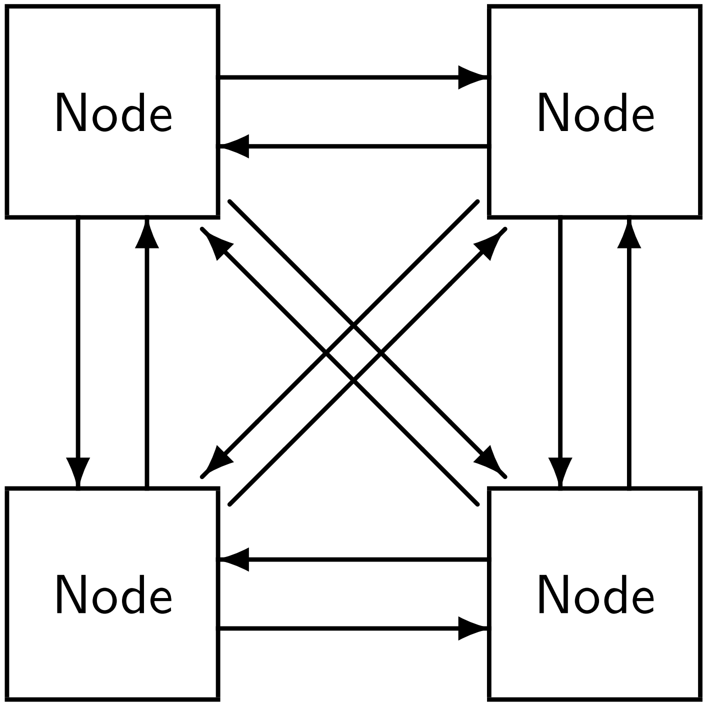
0000 0000 0000 0001

Local memory

0000 0000 0000 0010

Local memory

Distributed Systems Architecture

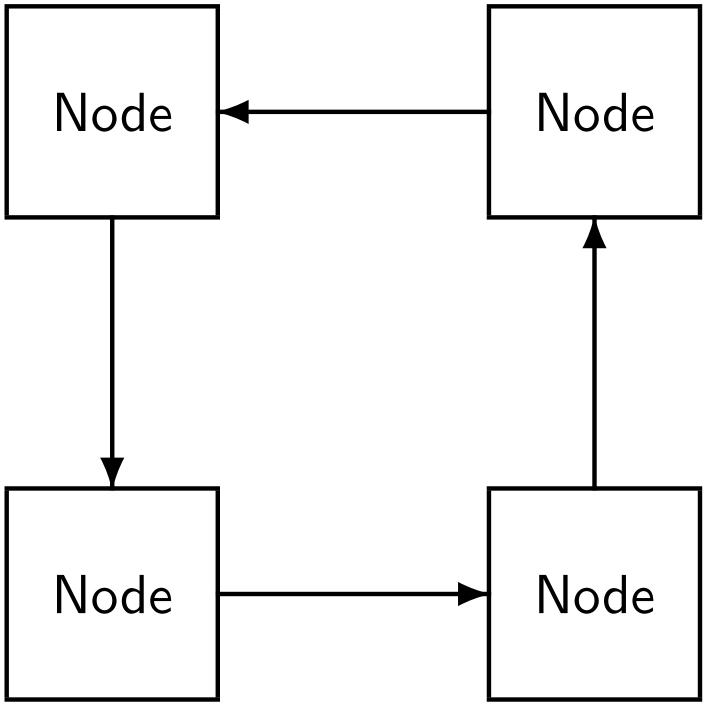


Node

Node

Node

Node



Node

Node

Node

Node

Algorithm 2.3: Atomic assignment statements

integer n ← 0

p	**q**
p1: n ← n + 1	q1: n ← n + 1

Scenario for Atomic Assignment Statements

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	**q1: n←n+1**	1
(end)	(end)	2

Process p	Process q	n
p1: n←n+1	**q1: n←n+1**	0
p1: n←n+1	(end)	1
(end)	(end)	2

Algorithm 2.4: Assignment statements with one global reference

```
integer n ← 0
```

p

```
     integer temp
p1:  temp ← n
p2:  n ← temp + 1
```

q

```
     integer temp
q1:  temp ← n
q2:  n ← temp + 1
```

Correct Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	q1: temp←n	0	0	?
(end)	**q1: temp←n**	1	0	?
(end)	**q2: n←temp+1**	1	0	1
(end)	(end)	2	0	1

Incorrect Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	**q1: temp←n**	0	0	?
p2: n←temp+1	q2: n←temp+1	0	0	0
(end)	**q2: n←temp+1**	1	0	0
(end)	(end)	1	0	0

Algorithm 2.5: Stop the loop A

```
integer n ← 0
boolean flag ← false
```

p

```
p1:  while flag = false
p2:      n ← 1 − n
```

q

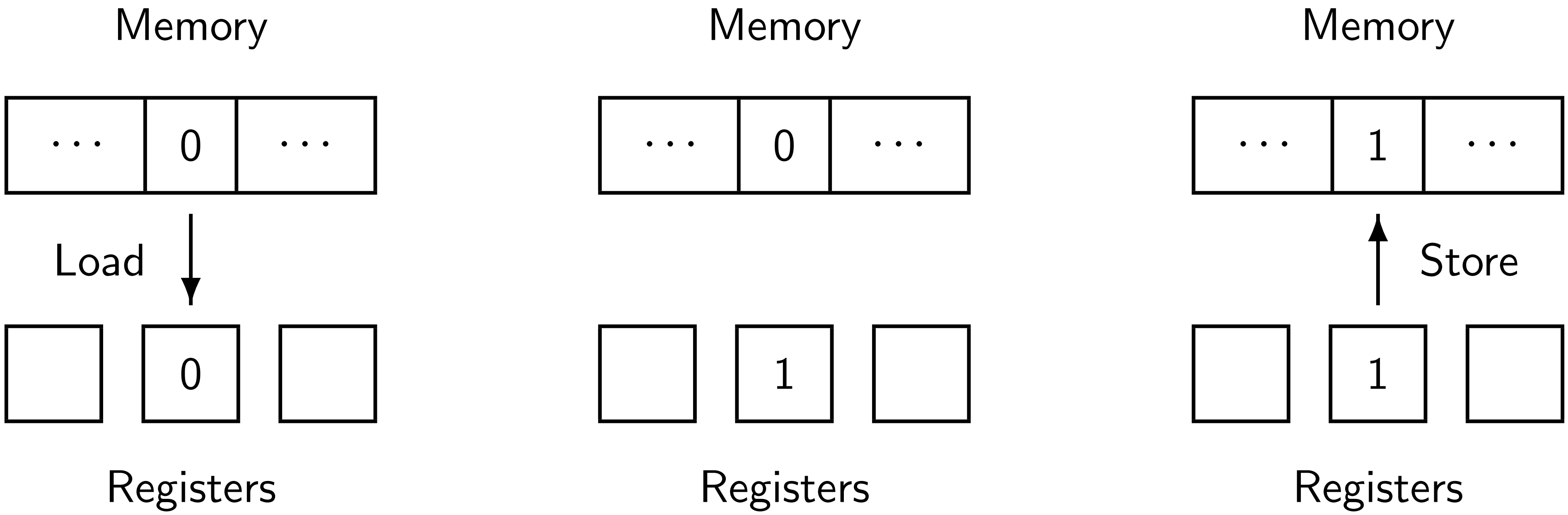
```
q1:  flag ← true
q2:
```

Algorithm 2.6: Assignment statement for a register machine

integer n ← 0

p	**q**
p1: load R1,n	q1: load R1,n
p2: add R1,#1	q2: add R1,#1
p3: store R1,n	q3: store R1,n

Register Machine



Scenario for a Register Machine

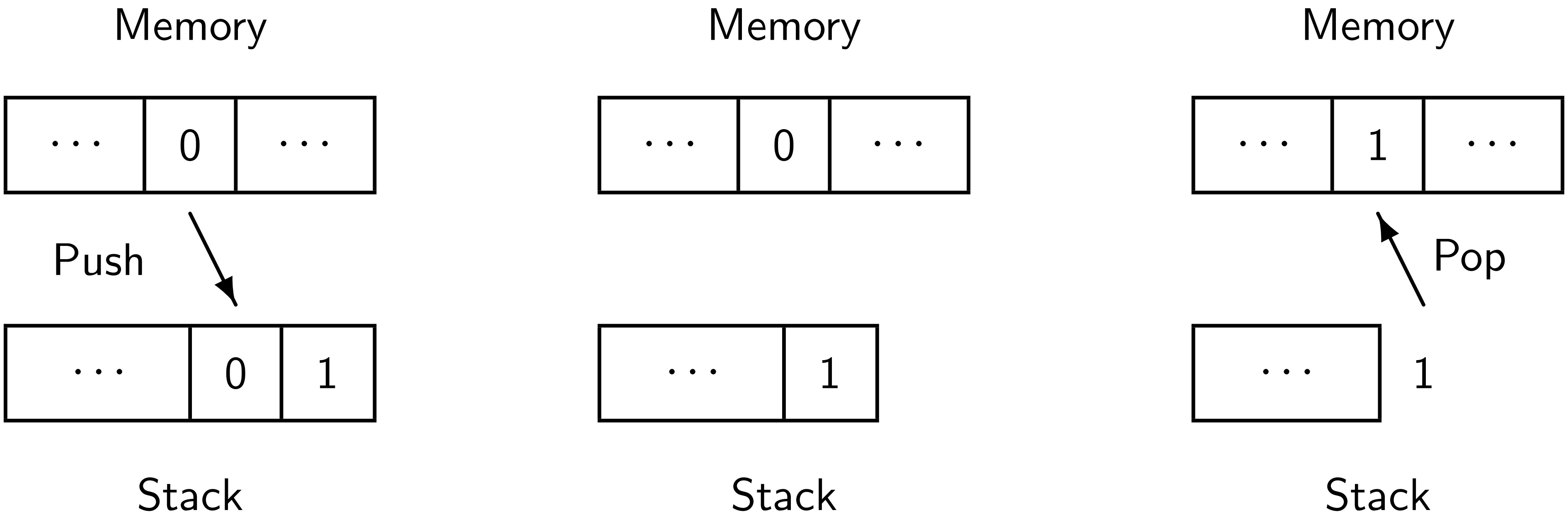
Process p	Process q	n	p.R1	q.R1
p1: load R1,n	q1: load R1,n	0	?	?
p2: add R1,#1	**q1: load R1,n**	0	0	?
p2: add R1,#1	q2: add R1,#1	0	0	0
p3: store R1,n	**q2: add R1,#1**	0	1	0
p3: store R1,n	q3: store R1,n	0	1	1
(end)	**q3: store R1,n**	1	1	1
(end)	(end)	1	1	1

Algorithm 2.7: Assignment statement for a stack machine

integer n ← 0

p	q
p1: push n	q1: push n
p2: push #1	q2: push #1
p3: add	q3: add
p4: pop n	q4: pop n

Stack Machine



Memory

Memory

Memory

···	0	···

···	0	···

···	1	···

Push

Pop

···	0	1

···	1

···

1

Stack

Stack

Stack

Algorithm 2.8: Volatile variables

integer n ← 0

	p		**q**
	integer local1, local2		integer local
p1:	n ← *some expression*	q1:	local ← n + 6
p2:	*computation not using n*	q2:	
p3:	local1 ← (n + 5) ∗ 7	q3:	
p4:	local2 ← n + 5	q4:	
p5:	n ← local1 * local2	q5:	

Algorithm 2.9: Concurrent counting algorithm

```
integer n ← 0
```

p

```
    integer temp
p1: do 10 times
p2:     temp ← n
p3:     n ← temp + 1
```

q

```
    integer temp
q1: do 10 times
q2:     temp ← n
q3:     n ← temp + 1
```

Concurrent Program in Pascal

```
program count;
var n: integer := 0;

procedure p;
var temp, i : integer;
begin
  for i := 1 to 10 do
    begin
    temp := n;
    n := temp + 1
    end
end;
```

Concurrent Program in Pascal

```
procedure q;
var temp, i : integer;
begin
  for i := 1 to 10 do
    begin
    temp := n;
    n := temp + 1
    end
end;

begin
  cobegin p; q coend;
  writeln('The value of n is ', n)
end.
```

Concurrent Program in C

```
int n = 0;

void p() {
  int temp, i;
  for (i = 0; i < 10; i++) {
    temp = n;
    n = temp + 1;
  }
}
```

Concurrent Program in C

```
void q() {
  int temp, i;
  for (i = 0; i < 10; i++) {
    temp = n;
    n = temp + 1;
  }
}

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

Concurrent Program in Ada

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Count is
   N: Integer := 0;
   pragma Volatile(N);

   task type Count_Task;
   task body Count_Task is
      Temp: Integer;
   begin
      for I in 1..10 loop
         Temp := N;
         N := Temp + 1;
      end loop;
   end Count_Task;
```

Concurrent Program in Ada

```
begin
    declare
        P, Q: Count_Task;
    begin
        null ;
    end;
    Put_Line("The value of N is " & Integer’Image(N));
end Count;
```

Concurrent Program in Java

```
class Count extends Thread {
    static volatile int n = 0;

    public void run() {
      int temp;
      for (int i = 0; i < 10; i++) {
        temp = n;
        n = temp + 1;
      }
    }
```

Concurrent Program in Java

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

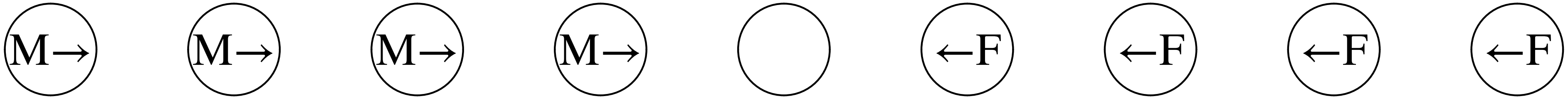
Concurrent Program in Promela

```
#include "for.h"
#define TIMES 10
byte    n = 0;

proctype P() {
    byte temp;
    for  (i ,1, TIMES)
        temp = n;
        n = temp + 1
    rof  (i )
}
```

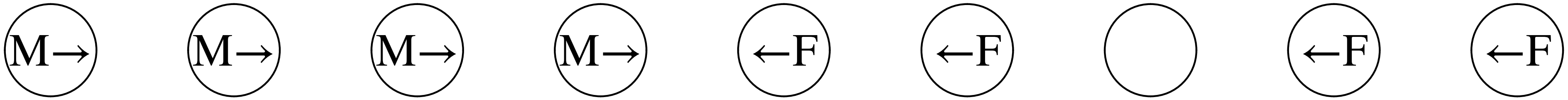
Concurrent Program in Promela

```
init {
    atomic {
        run P();
        run P()
    }
    ( _nr_pr == 1);
    printf ("MSC: The value is %d\n", n)
}
```

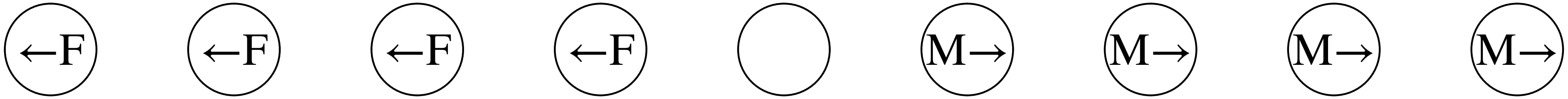
Frog Puzzle



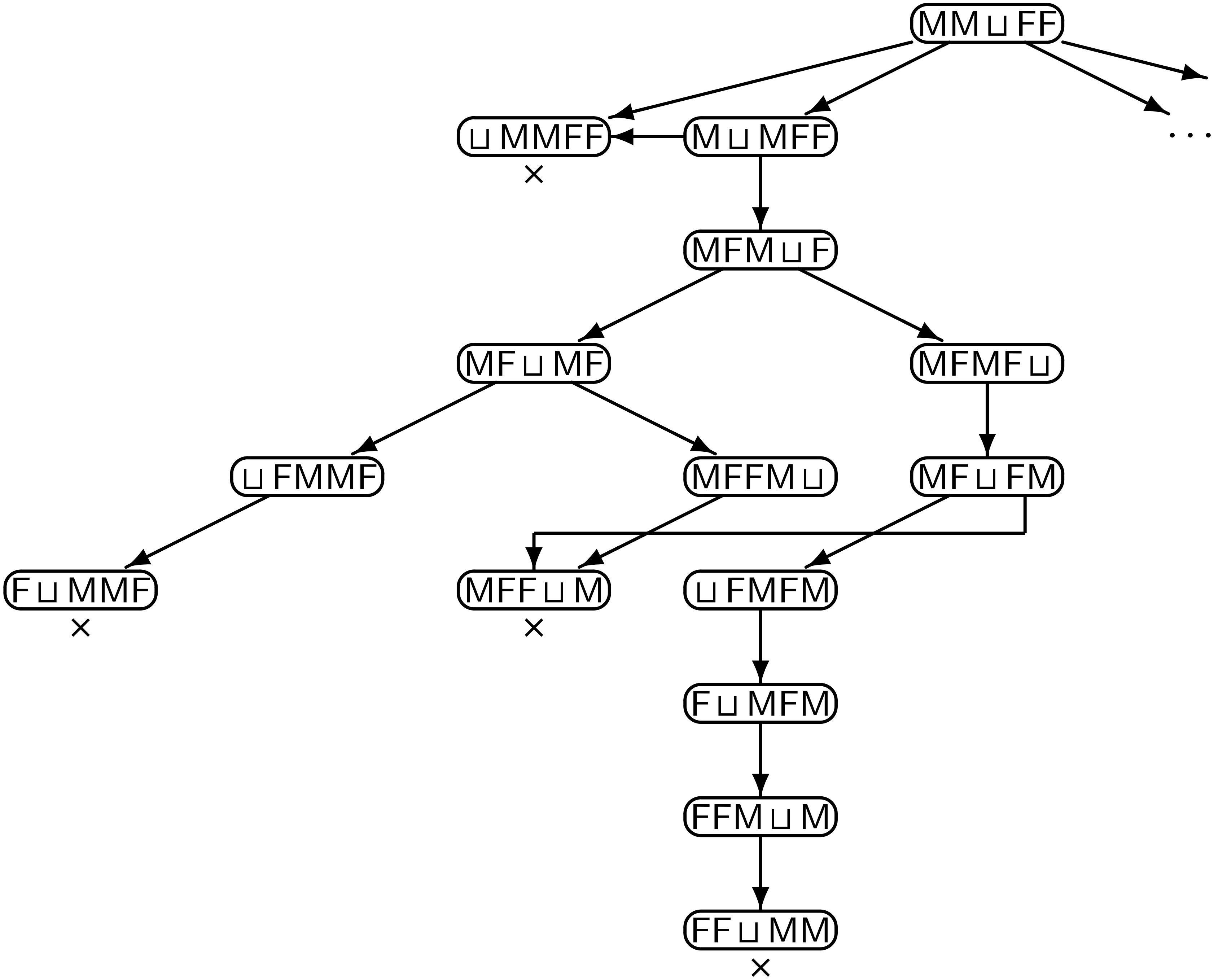
One Step of the Frog Puzzle



Final State of the Frog Puzzle


←F ←F ←F ←F M→ M→ M→ M→

(Partial) State Diagram for the Frog Puzzle



Algorithm 2.10: Incrementing and decrementing

integer n ← 0

p

```
     integer temp
p1: do K times
p2:     temp ← n
p3:     n ← temp + 1
```

q

```
     integer temp
q1: do K times
q2:     temp ← n
q3:     n ← temp − 1
```

Algorithm 2.11: Zero A

boolean found

p

```
     integer i ← 0
p1: found ← false
p2: while not found
p3:     i ← i + 1
p4:     found ← f(i) = 0
```

q

```
     integer j ← 1
q1: found ← false
q2: while not found
q3:     j ← j − 1
q4:     found ← f(j) = 0
```

Algorithm 2.12: Zero B

boolean found ← false

p

```
    integer i ← 0
p1: while not found
p2:     i ← i + 1
p3:     found ← f(i) = 0
```

q

```
    integer j ← 1
q1: while not found
q2:     j ← j − 1
q3:     found ← f(j) = 0
```

Algorithm 2.13: Zero C

boolean found ← false

p

```
integer i ← 0
while not found
    i ← i + 1
    if f(i) = 0
        found ← true
```

q

```
integer j ← 1
while not found
    j ← j − 1
    if f(j) = 0
        found ← true
```

Algorithm 2.14: Zero D

```
boolean found ← false
integer turn ← 1
```

p

```
     integer i ← 0
p1:  while not found
p2:      await turn = 1
             turn ← 2
p3:      i ← i + 1
p4:      if f(i) = 0
p5:          found ← true
```

q

```
     integer j ← 1
q1:  while not found
q2:      await turn = 2
             turn ← 1
q3:      j ← j − 1
q4:      if f(j) = 0
q5:          found ← true
```

Algorithm 2.15: Zero E

```
boolean found ← false
integer turn ← 1
```

p

```
     integer i ← 0
p1:  while not found
p2:     await turn = 1
           turn ← 2
p3:     i ← i + 1
p4:     if f(i) = 0
p5:        found ← true
p6:  turn ← 2
```

q

```
     integer j ← 1
q1:  while not found
q2:     await turn = 2
           turn ← 1
q3:     j ← j − 1
q4:     if f(j) = 0
q5:        found ← true
q6:  turn ← 1
```

Algorithm 2.16: Concurrent algorithm A

```
                integer array [1..10] C ← ten *distinct* initial values
                integer array [1..10] D
      integer myNumber, count
p1:   myNumber ← C[i]
p2:   count ← number of elements of C less than myNumber
p3:   D[count + 1] ← myNumber
```

Algorithm 2.17: Concurrent algorithm B

```
integer $n \leftarrow 0$
```

p

```
p1: while $n < 2$
p2:     write(n)
```

q

```
q1: $n \leftarrow n + 1$
q2: $n \leftarrow n + 1$
```

Algorithm 2.18: Concurrent algorithm C

integer n ← 1

p

```
p1: while n < 1
p2:     n ← n + 1
```

q

```
q1: while n >= 0
q2:     n ← n − 1
```

Algorithm 2.19: Stop the loop B

```
integer n ← 0
boolean flag ← false
```

p

```
p1:  while flag = false
p2:      n ← 1 − n
p3:
```

q

```
q1:  while flag = false
q2:      if n = 0
q3:          flag ← true
```

Algorithm 2.20: Stop the loop C

```
integer n ← 0
boolean flag ← false
```

p

```
p1: while flag = false
p2:     n ← 1 − n
```

q

```
q1: while n = 0 // Do nothing
q2: flag ← true
```

Algorithm 2.21: Welfare crook problem

```
integer array[0..N] a, b, c ← . . . (as required)
integer i ← 0, j ← 0, k ← 0

    loop
p1:     if condition-1
p2:         i ← i + 1
p3:     else if condition-2
p4:         j ← j + 1
p5:     else if condition-3
p6:         k ← k + 1
        else exit loop
```

Algorithm 3.1: Critical section problem

global variables

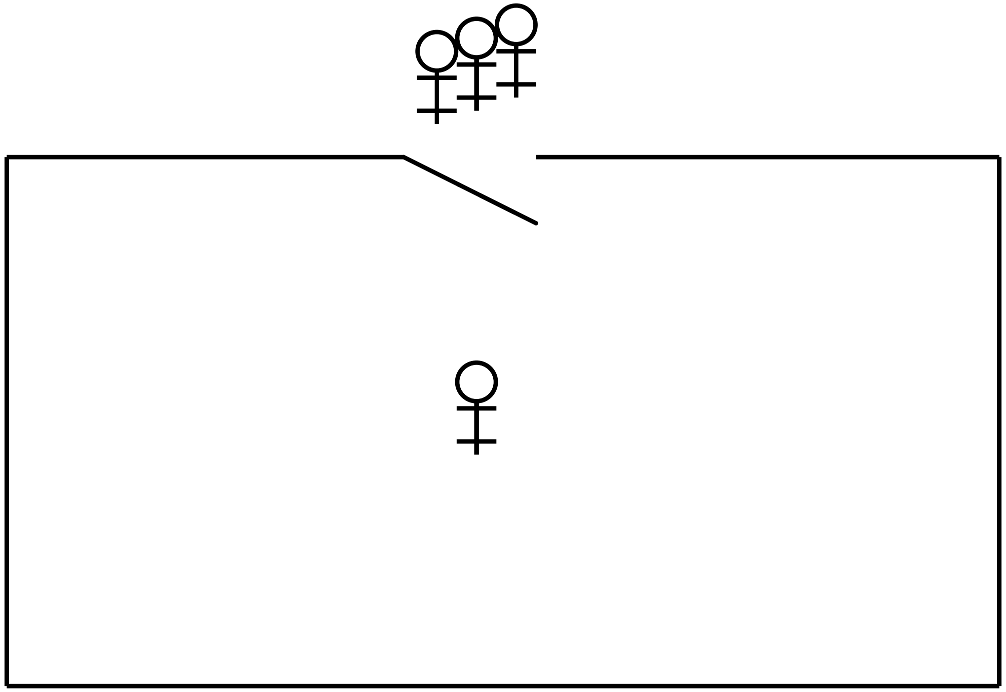
p

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

q

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

Critical Section



Algorithm 3.2: First attempt

```
integer turn ← 1
```

p

```
    loop forever
p1:     non-critical section
p2:     await turn = 1
p3:     critical section
p4:     turn ← 2
```

q

```
    loop forever
q1:     non-critical section
q2:     await turn = 2
q3:     critical section
q4:     turn ← 1
```

Algorithm 3.3: History in a sequential algorithm

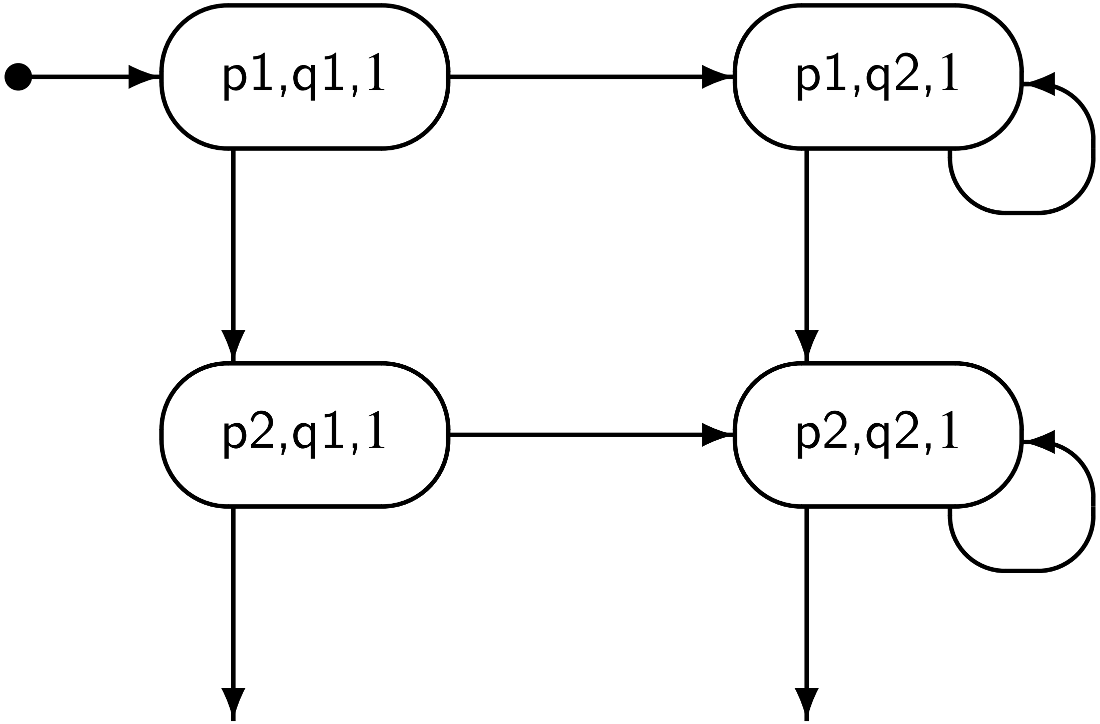
```
integer a ← 1, b ← 2
p1: Millions of statements
p2: a ← (a+b)*5
p3: . . .
```

Algorithm 3.4: History in a concurrent algorithm

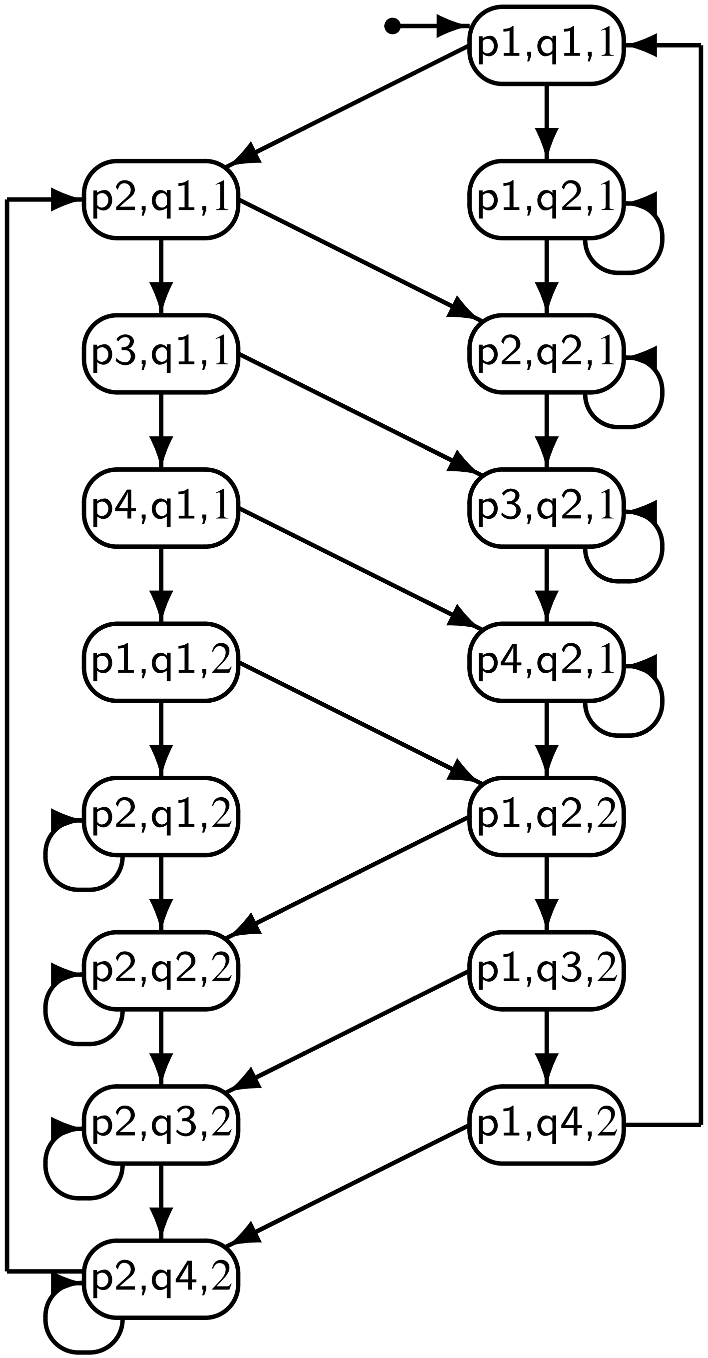
integer a ← 1, b ← 2

p	q
p1: Millions of statements	q1: Millions of statements
p2: a ← (a+b)*5	q2: b ← (a+b)*5
p3: . . .	q3: . . .

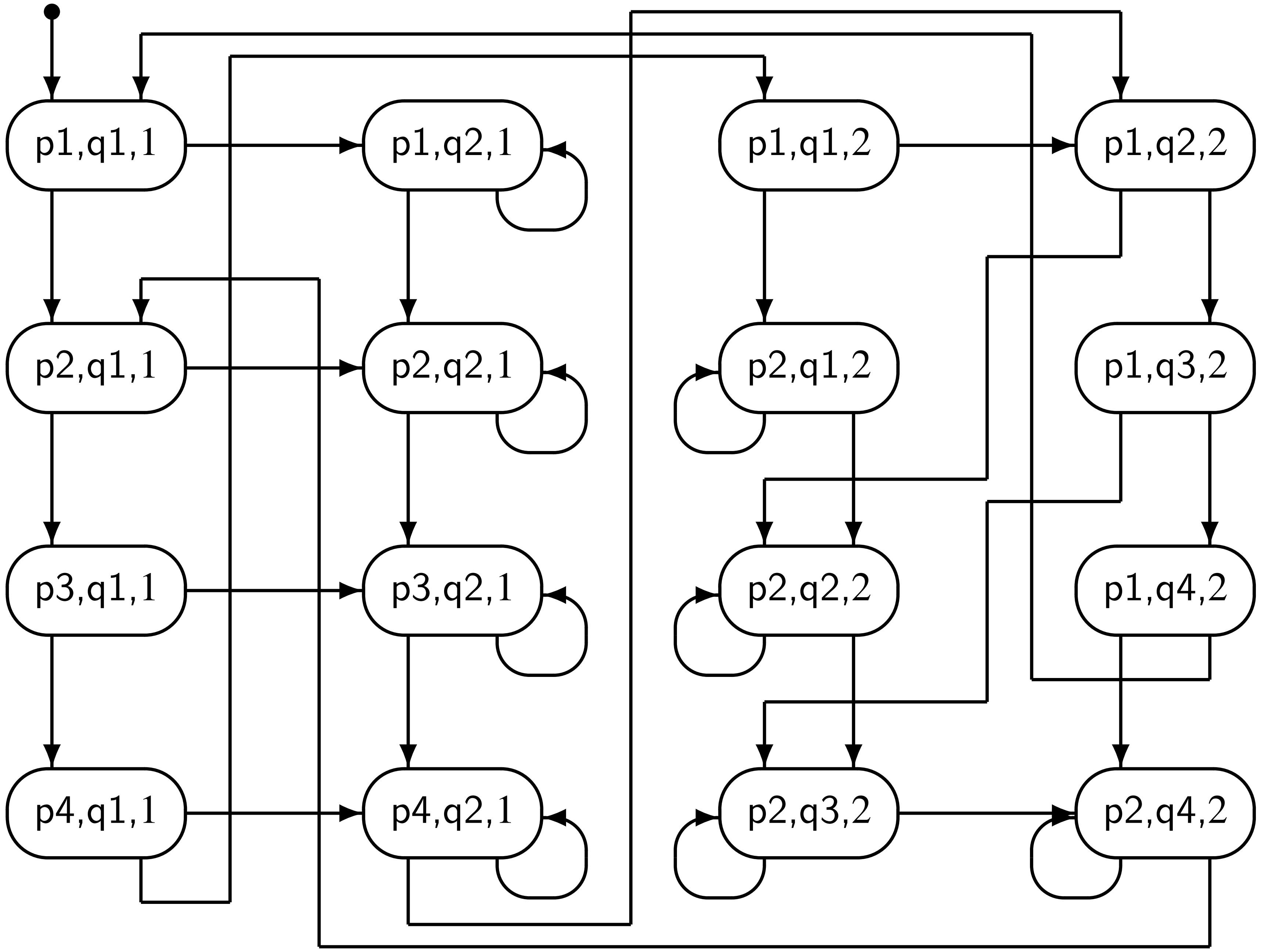
First States of the State Diagram



State Diagram for the First Attempt



Alternate Layout for the First Attempt (Not in the Book)



Algorithm 3.5: First attempt (abbreviated)

integer turn ← 1

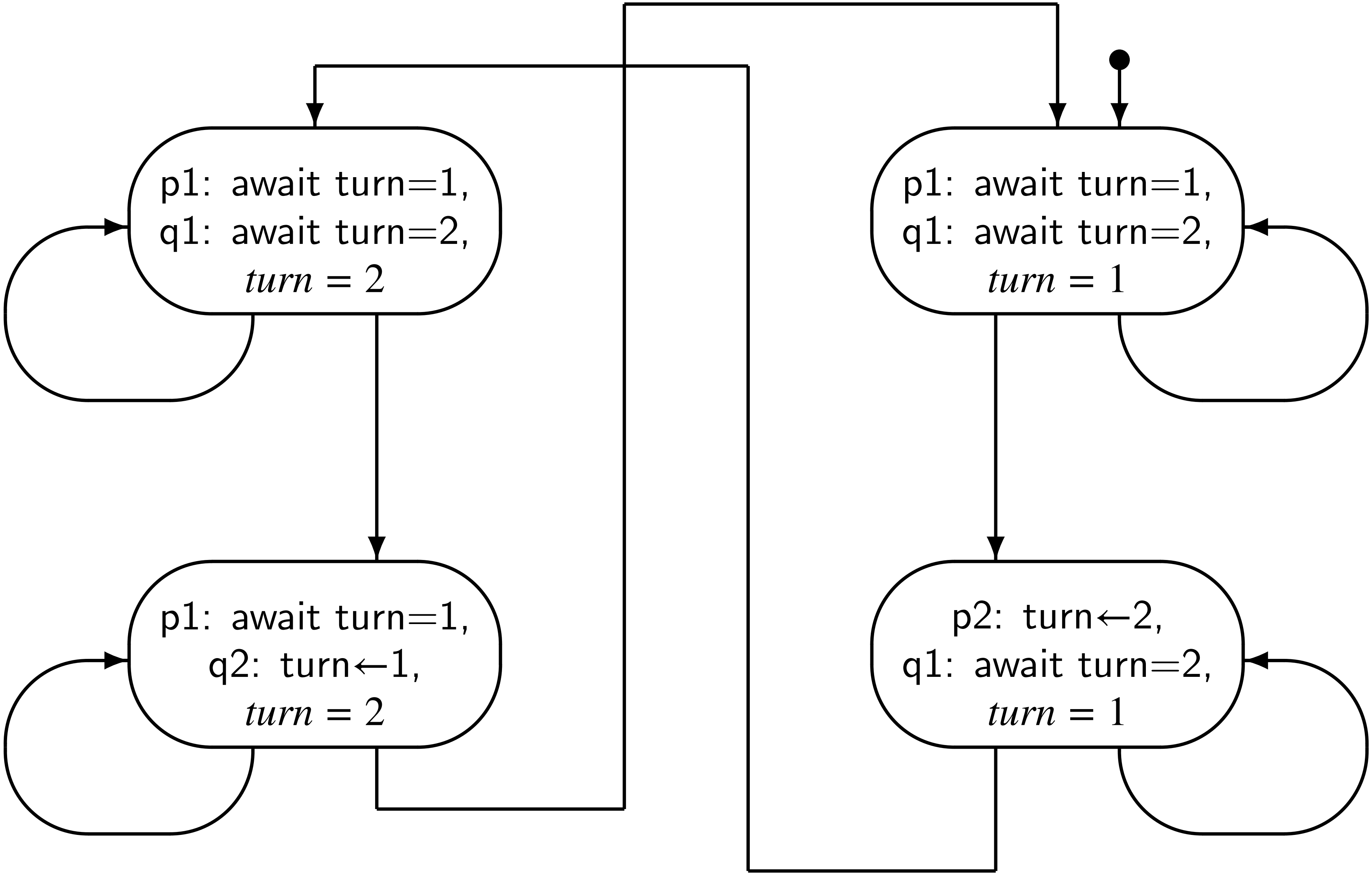
p

```
      loop forever
p1:       await turn = 1
p2:       turn ← 2
```

q

```
      loop forever
q1:       await turn = 2
q2:       turn ← 1
```

State Diagram for the Abbreviated First Attempt



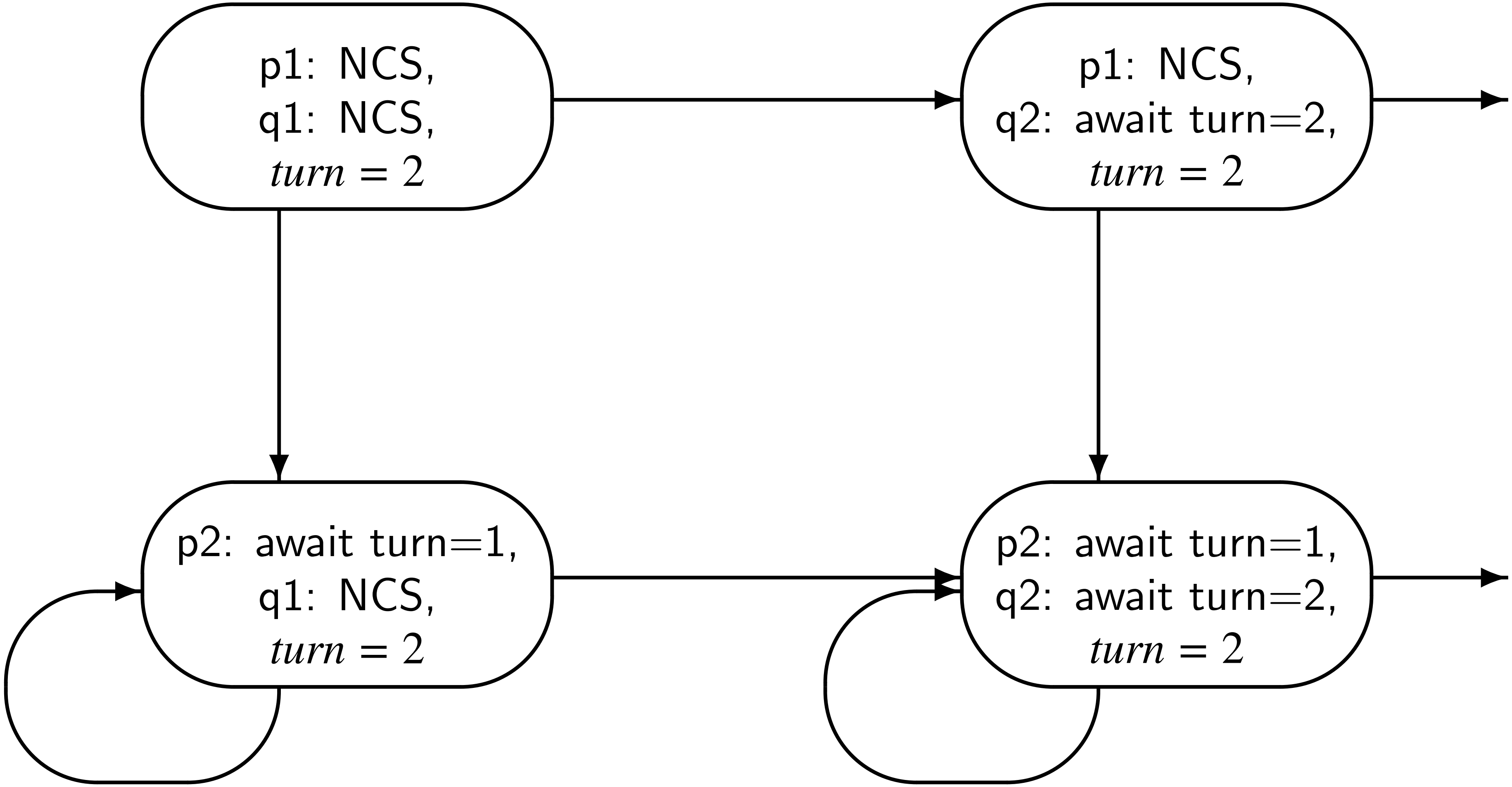
p1: await turn=1,
q1: await turn=2,
$turn = 1$

p1: await turn=1,
q1: await turn=2,
$turn = 2$

p2: turn←2,
q1: await turn=2,
$turn = 1$

p1: await turn=1,
q2: turn←1,
$turn = 2$

Fragment of the State Diagram for the First Attempt



p1: NCS,
q1: NCS,
turn = 2

p1: NCS,
q2: await turn=2,
turn = 2

p2: await turn=1,
q1: NCS,
turn = 2

p2: await turn=1,
q2: await turn=2,
turn = 2

Algorithm 3.6: Second attempt

```
boolean wantp ← false, wantq ← false
```

p

```
        loop forever
p1:         non-critical section
p2:         await wantq = false
p3:         wantp ← true
p4:         critical section
p5:         wantp ← false
```

q

```
        loop forever
q1:         non-critical section
q2:         await wantp = false
q3:         wantq ← true
q4:         critical section
q5:         wantq ← false
```

Algorithm 3.7: Second attempt (abbreviated)

```
boolean wantp ← false, wantq ← false
```

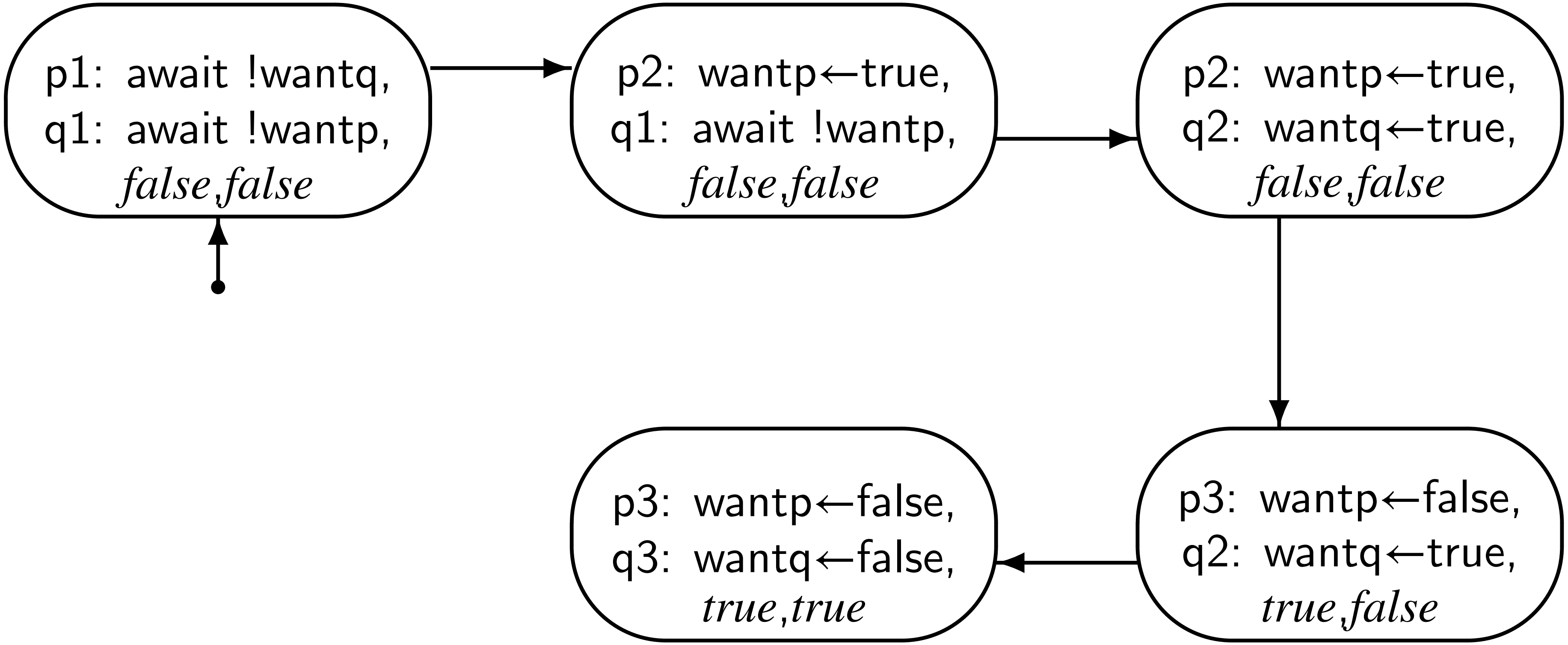
p

```
        loop forever
p1:         await wantq = false
p2:         wantp ← true
p3:         wantp ← false
```

q

```
        loop forever
q1:         await wantp = false
q2:         wantq ← true
q3:         wantq ← false
```

Fragment of the State Diagram for the Second Attempt



p1: await !wantq,
q1: await !wantp,
false,*false*

p2: wantp←true,
q1: await !wantp,
false,*false*

p2: wantp←true,
q2: wantq←true,
false,*false*

p3: wantp←false,
q2: wantq←true,
true,*false*

p3: wantp←false,
q3: wantq←false,
true,*true*

Scenario Showing that Mutual Exclusion Does Not Hold

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*

Algorithm 3.8: Third attempt

```
boolean wantp ← false, wantq ← false
```

p

```
    loop forever
p1:     non-critical section
p2:     wantp ← true
p3:     await wantq = false
p4:     critical section
p5:     wantp ← false
```

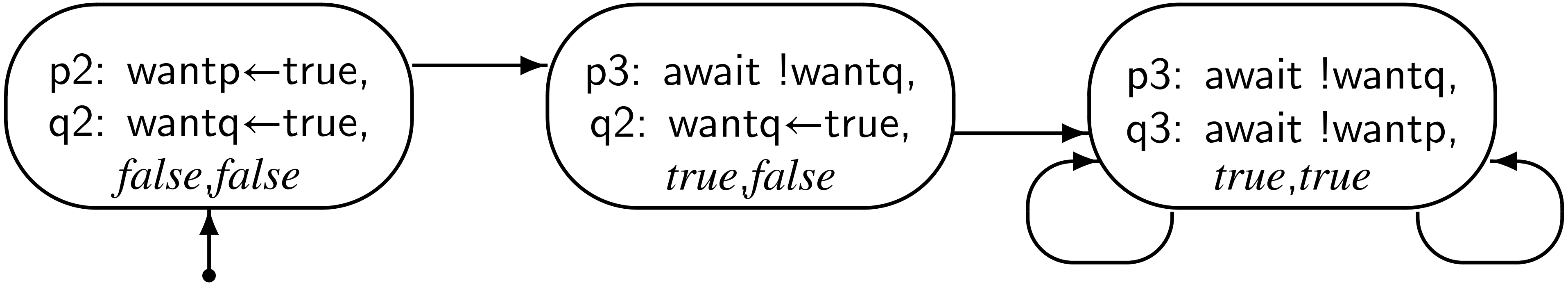
q

```
    loop forever
q1:     non-critical section
q2:     wantq ← true
q3:     await wantp = false
q4:     critical section
q5:     wantq ← false
```

Scenario Showing Deadlock in the Third Attempt

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*

Fragment of the State Diagram Showing Deadlock



Algorithm 3.9: Fourth attempt

boolean wantp ← false, wantq ← false

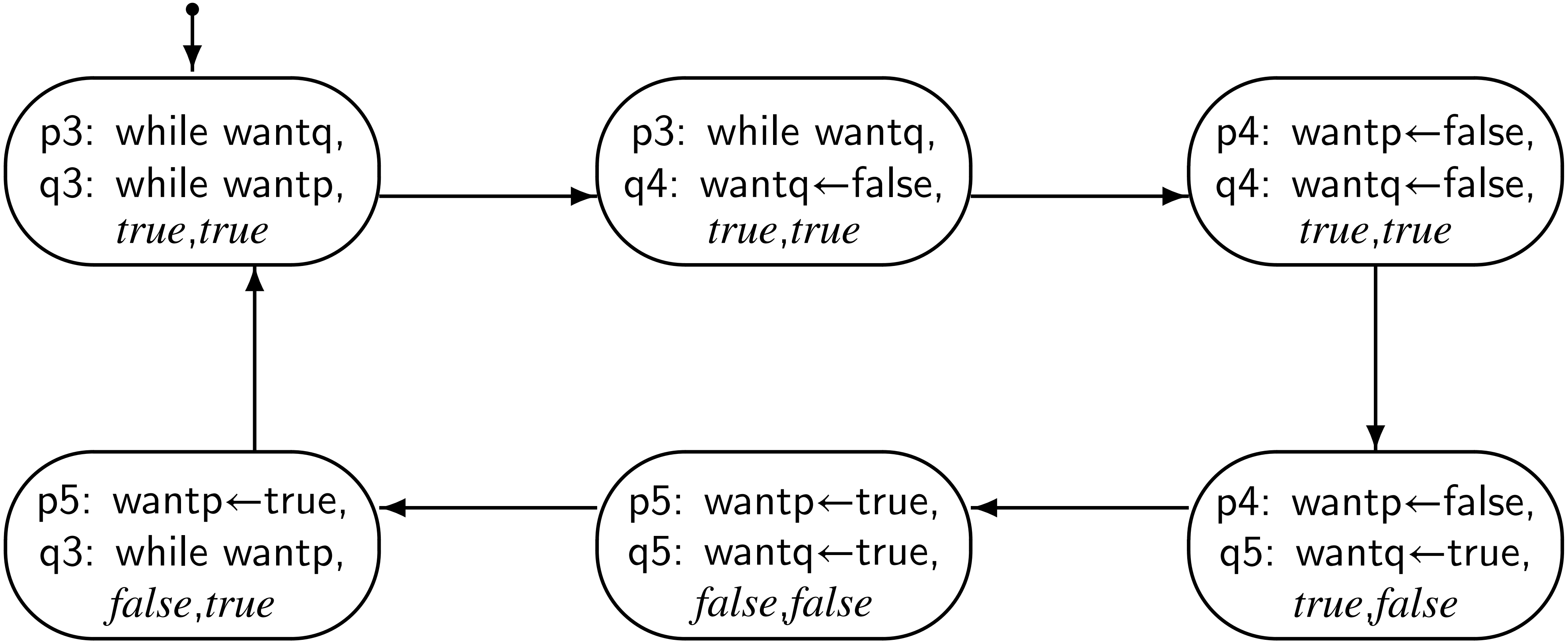
p

```
        loop forever
p1:         non-critical section
p2:         wantp ← true
p3:         while wantq
p4:             wantp ← false
p5:             wantp ← true
p6:         critical section
p7:         wantp ← false
```

q

```
        loop forever
q1:         non-critical section
q2:         wantq ← true
q3:         while wantp
q4:             wantq ← false
q5:             wantq ← true
q6:         critical section
q7:         wantq ← false
```

Cycle in the State Diagram for the Fourth Attempt



p3: while wantq,
q3: while wantp,
true,*true*

p3: while wantq,
q4: wantq←false,
true,*true*

p4: wantp←false,
q4: wantq←false,
true,*true*

p4: wantp←false,
q5: wantq←true,
true,*false*

p5: wantp←true,
q5: wantq←true,
false,*false*

p5: wantp←true,
q3: while wantp,
false,*true*

Algorithm 3.10: Dekker's algorithm

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

p

```
     loop forever
p1:     non-critical section
p2:     wantp ← true
p3:     while wantq
p4:        if turn = 2
p5:           wantp ← false
p6:           await turn = 1
p7:           wantp ← true
p8:     critical section
p9:     turn ← 2
p10:    wantp ← false
```

q

```
     loop forever
q1:     non-critical section
q2:     wantq ← true
q3:     while wantp
q4:        if turn = 1
q5:           wantq ← false
q6:           await turn = 2
q7:           wantq ← true
q8:     critical section
q9:     turn ← 1
q10:    wantq ← false
```

Algorithm 3.11: Critical section problem with test-and-set

```
integer common ← 0
```

p

```
     integer local1
     loop forever
p1:     non-critical section
        repeat
p2:        test-and-set(
              common, local1)
p3:     until local1 = 0
p4:     critical section
p5:     common ← 0
```

q

```
     integer local2
     loop forever
q1:     non-critical section
        repeat
q2:        test-and-set(
              common, local2)
q3:     until local2 = 0
q4:     critical section
q5:     common ← 0
```

Algorithm 3.12: Critical section problem with exchange

integer common ← 1

p

```
      integer local1 ← 0
      loop forever
p1:      non-critical section
         repeat
p2:         exchange(common, local1)
p3:      until local1 = 1
p4:      critical section
p5:      exchange(common, local1)
```

q

```
      integer local2 ← 0
      loop forever
q1:      non-critical section
         repeat
q2:         exchange(common, local2)
q3:      until local2 = 1
q4:      critical section
q5:      exchange(common, local2)
```

Algorithm 3.13: Peterson's algorithm

```
boolean wantp ← false, wantq ← false
integer last ← 1
```

p

```
    loop forever
p1:     non-critical section
p2:     wantp ← true
p3:     last ← 1
p4:     await wantq = false or
                last = 2
p5:     critical section
p6:     wantp ← false
```

q

```
    loop forever
q1:     non-critical section
q2:     wantq ← true
q3:     last ← 2
q4:     await wantp = false or
                last = 1
q5:     critical section
q6:     wantq ← false
```

Algorithm 3.14: Manna-Pnueli algorithm

integer wantp ← 0, wantq ← 0

p

```
     loop forever
p1:     non-critical section
p2:     if wantq = −1
           wantp ← −1
         else wantp ← 1
p3:     await wantq ≠ wantp
p4:     critical section
p5:     wantp ← 0
```

q

```
     loop forever
q1:     non-critical section
q2:     if wantp = −1
           wantq ← 1
         else wantq ← −1
q3:     await wantp ≠ − wantq
q4:     critical section
q5:     wantq ← 0
```

Algorithm 3.15: Doran-Thomas algorithm

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

p

```
     loop forever
p1:      non-critical section
p2:      wantp ← true
p3:      if wantq
p4:          if turn = 2
p5:              wantp ← false
p6:              await turn = 1
p7:              wantp ← true
p8:          await wantq = false
p9:      critical section
p10:     wantp ← false
p11:     turn ← 2
```

q

```
     loop forever
q1:      non-critical section
q2:      wantq ← true
q3:      if wantp
q4:          if turn = 1
q5:              wantq ← false
q6:              await turn = 2
q7:              wantq ← true
q8:          await wantp = false
q9:      critical section
q10:     wantq ← false
q11:     turn ← 1
```

Algorithm 4.1: Third attempt

boolean wantp ← false, wantq ← false

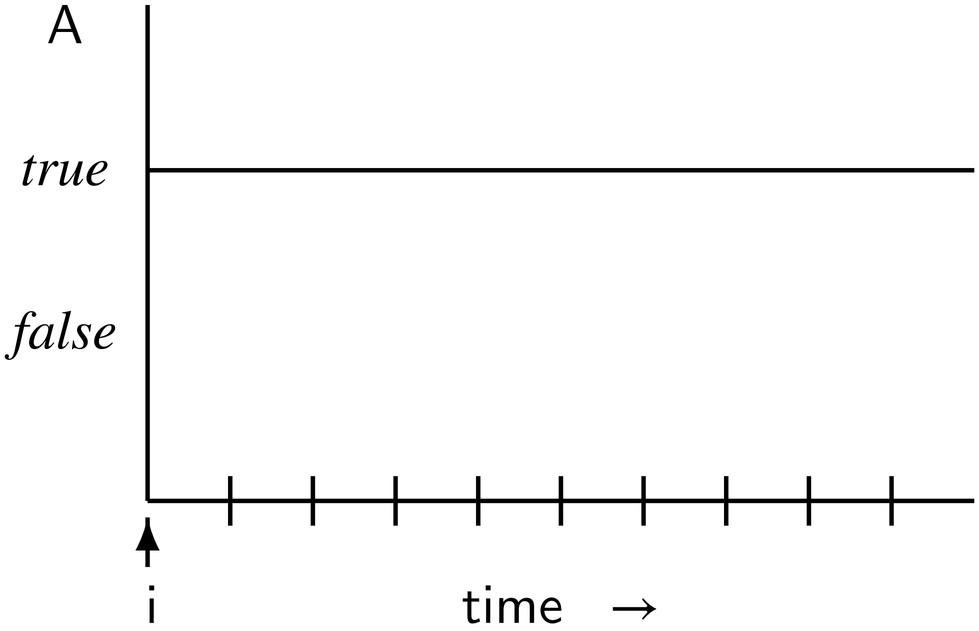
p

```
     loop forever
p1:     non-critical section
p2:     wantp ← true
p3:     await wantq = false
p4:     critical section
p5:     wantp ← false
```

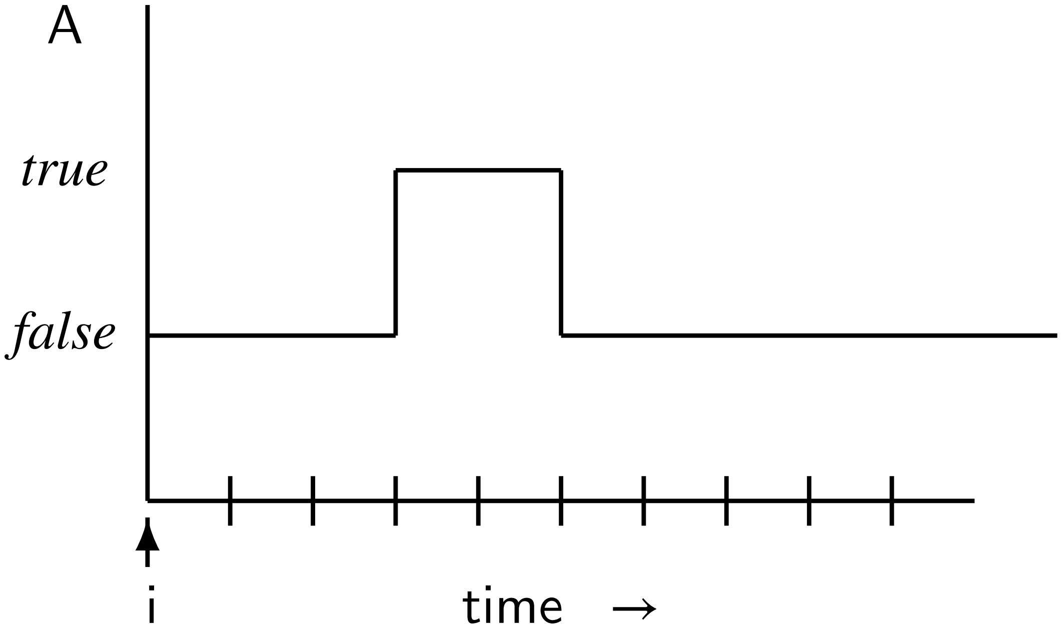
q

```
     loop forever
q1:     non-critical section
q2:     wantq ← true
q3:     await wantp = false
q4:     critical section
q5:     wantq ← false
```

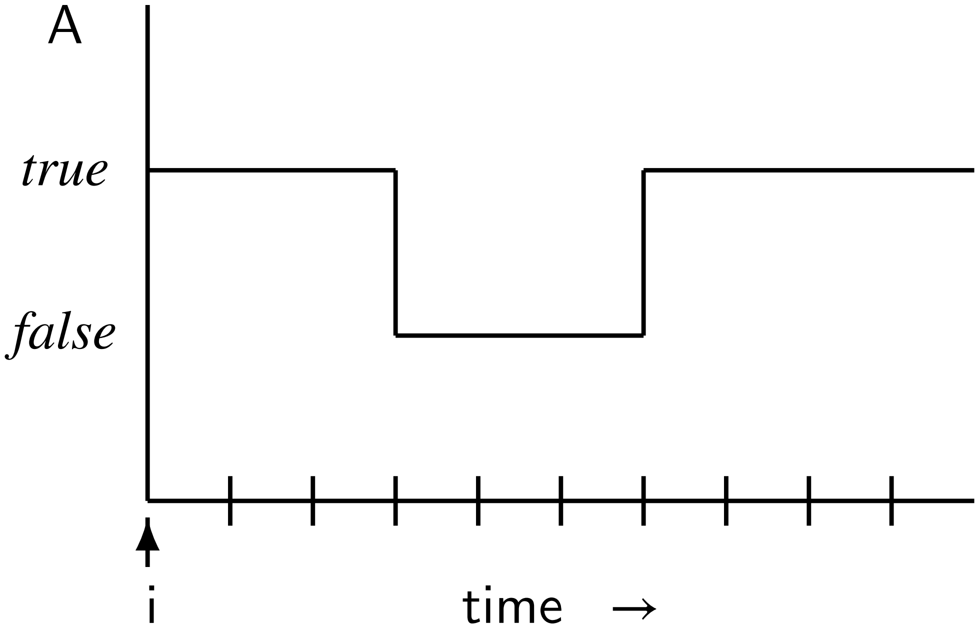
$\Box A$



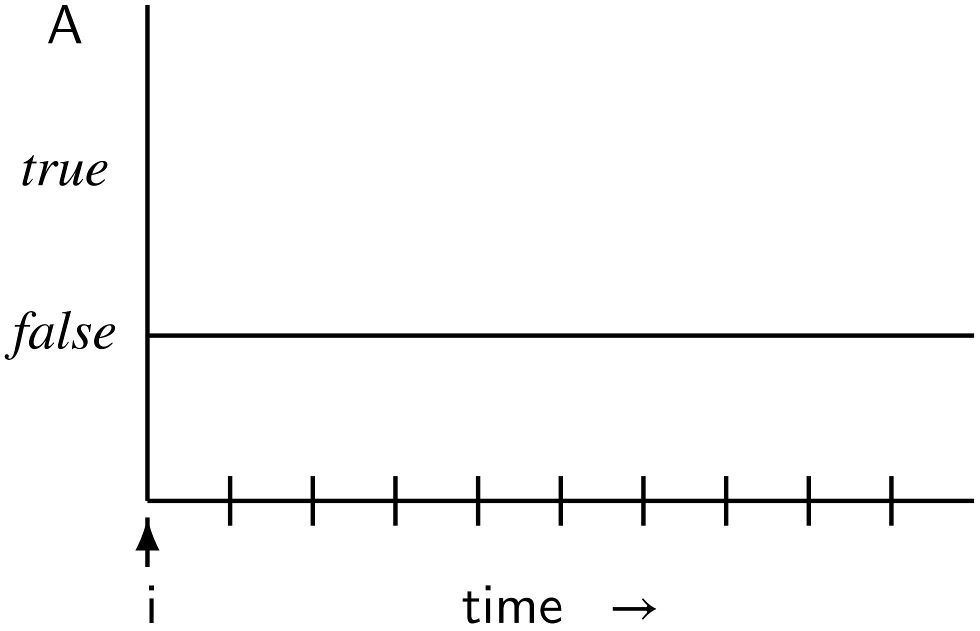
$\Diamond A$



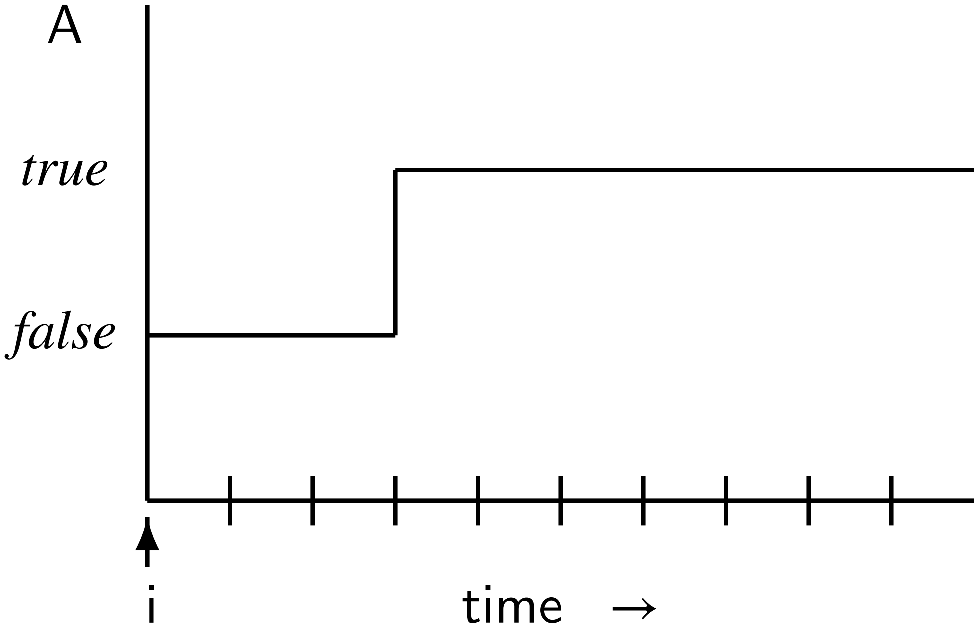
Duality: $\neg \Box A$



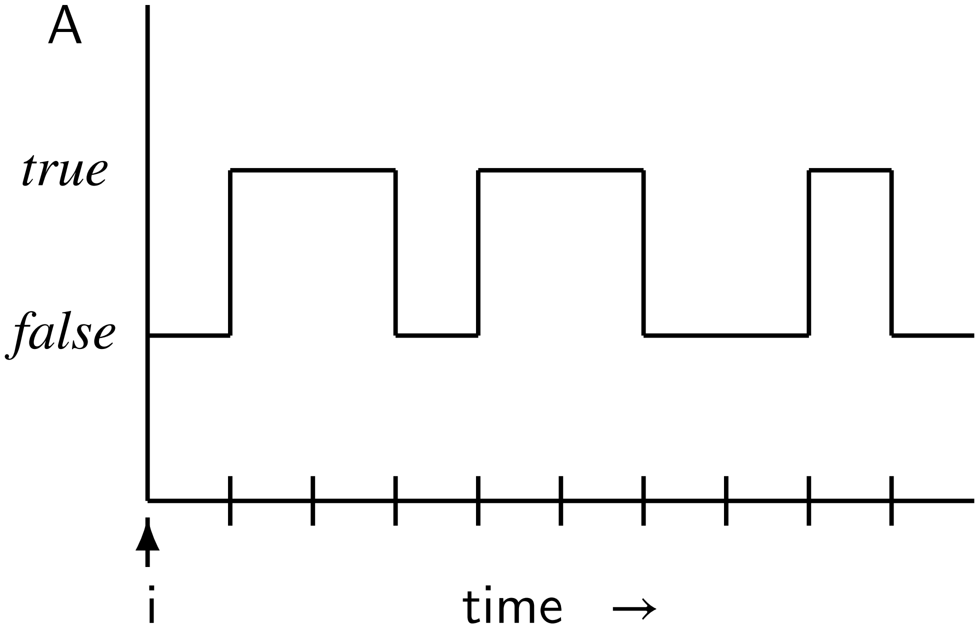
Duality: $\neg \Diamond A$



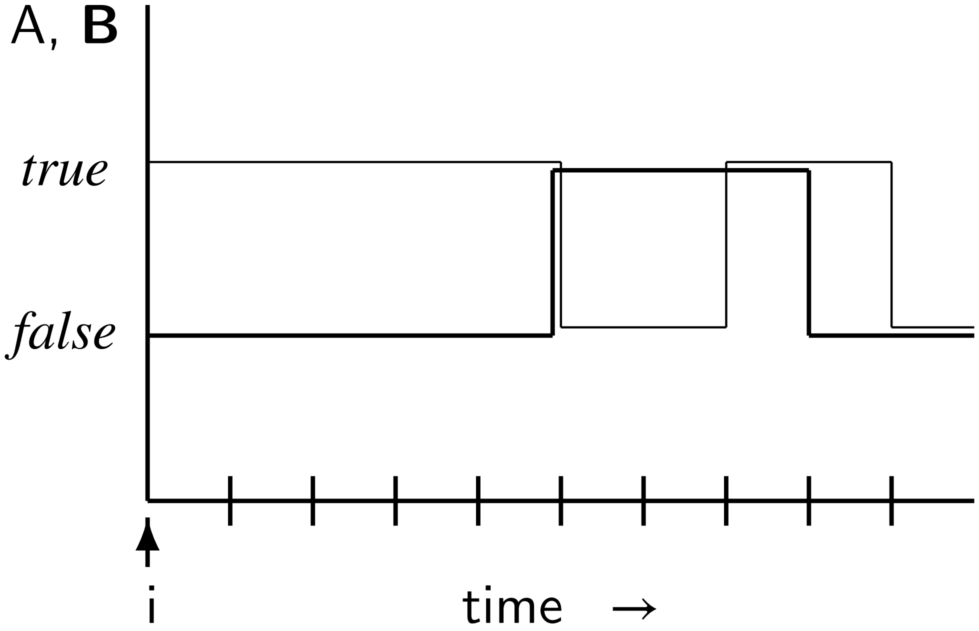
$\Diamond\Box A$



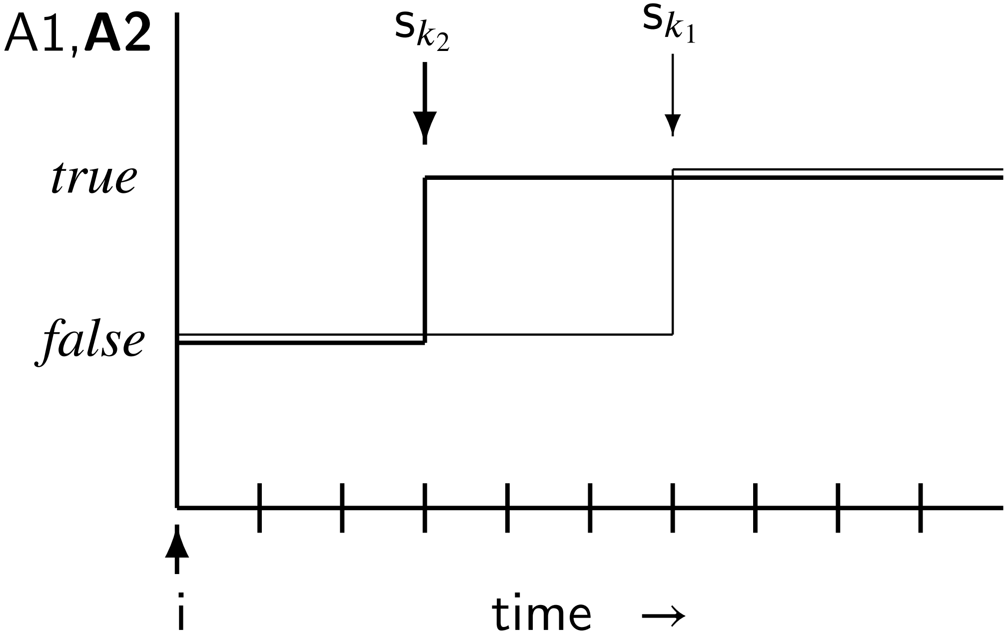
$\Box\Diamond A$



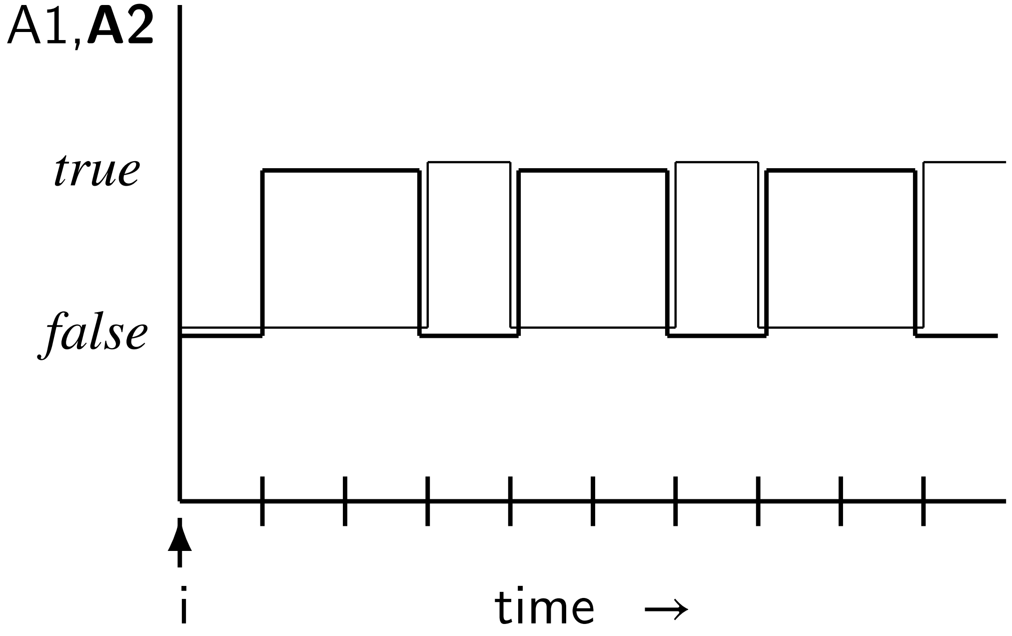
$A \,\mathcal{U}\, B$



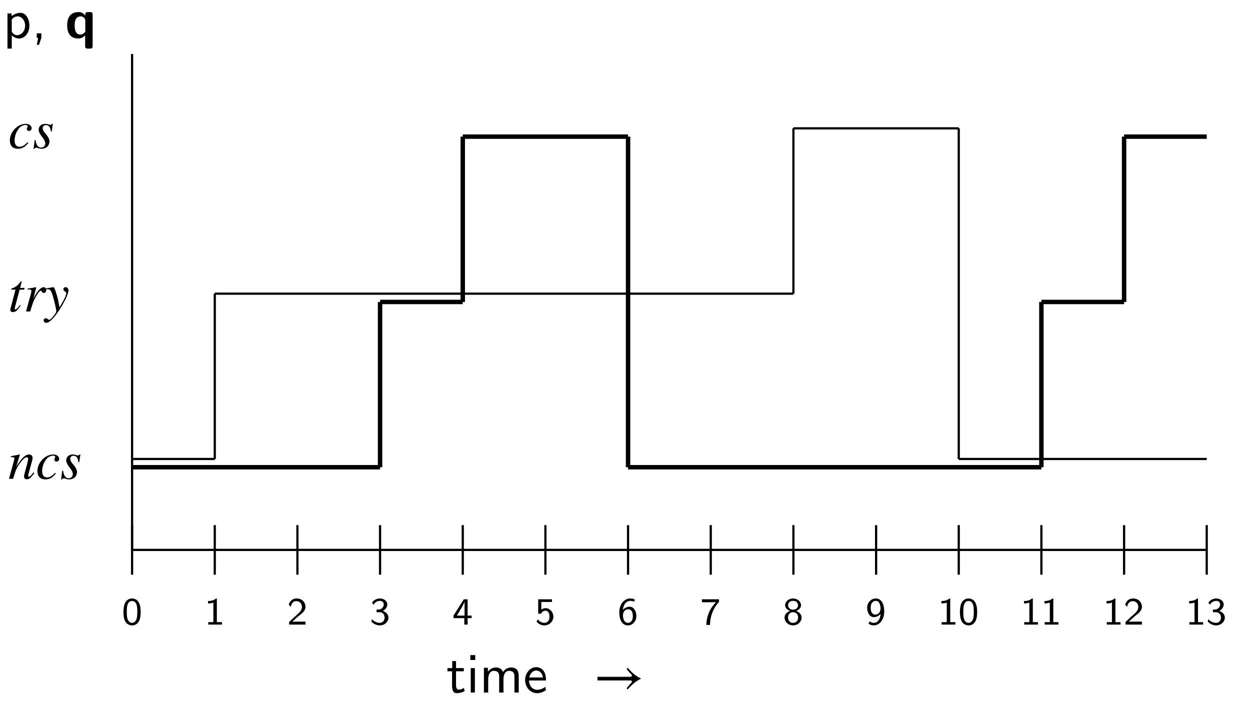
$\Diamond\Box A1 \land \Diamond\Box A2$



$\Box\Diamond A1 \wedge \Box\Diamond A2$



Overtaking: $try_p \rightarrow (\neg\, cs_q)\ \mathcal{W}\ (cs_q)\ \mathcal{W}\ (\neg\, cs_q)\ \mathcal{W}\ (cs_p)$



Algorithm 4.2: Dekker's algorithm

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

p

```
       loop forever
p1:       non-critical section
p2:       wantp ← true
p3:       while wantq
p4:          if turn = 2
p5:             wantp ← false
p6:             await turn = 1
p7:             wantp ← true
p8:       critical section
p9:       turn ← 2
p10:      wantp ← false
```

q

```
       loop forever
q1:       non-critical section
q2:       wantq ← true
q3:       while wantp
q4:          if turn = 1
q5:             wantq ← false
q6:             await turn = 2
q7:             wantq ← true
q8:       critical section
q9:       turn ← 1
q10:      wantq ← false
```

Dekker's Algorithm in Promela

```
bool wantp = false, wantq = false; byte turn = 1;
active proctype p() {
  do :: wantp = true;
    do :: !wantq -> break;
       :: else ->
          if :: (turn == 1)
             :: (turn == 2) ->
                wantp = false; (turn == 1); wantp = true
          fi
    od;
    printf("MSC: p in CS\n") ;
    turn = 2; wantp = false
 od
}
```

Specifying Correctness in Promela

```
byte critical = 0;

bool PinCS = false;
#define nostarve PinCS  /* LTL claim <> nostarve */

active proctype p() {
  do ::
      /* preprotocol */
      critical ++;
      assert(critical <= 1);
      PinCS = true;
      critical --;
      /* postprotocol */
  od
}
```

LTL Translation to Never Claims

```
never {     /* !(<>nostarve) */
accept_init:
T0_init:
    if
    :: (! ((nostarve))) -> goto T0_init
    fi ;
}
```

LTL Translation to Never Claims

```
never {     /* !([]<>nostarve) */
T0_init:
    if
    :: (! ((nostarve))) -> goto accept_S4
    :: (1) -> goto T0_init
    fi;
accept_S4:
    if
    :: (! ((nostarve))) -> goto accept_S4
    fi;
}
```

Algorithm 5.1: Bakery algorithm (two processes)

integer $np \leftarrow 0$, $nq \leftarrow 0$

p

```
    loop forever
p1:     non-critical section
p2:     $np \leftarrow nq + 1$
p3:     await $nq = 0$ or $np \leq nq$
p4:     critical section
p5:     $np \leftarrow 0$
```

q

```
    loop forever
q1:     non-critical section
q2:     $nq \leftarrow np + 1$
q3:     await $np = 0$ or $nq < np$
q4:     critical section
q5:     $nq \leftarrow 0$
```

Algorithm 5.2: Bakery algorithm (N processes)

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

        loop forever
p1:         non-critical section
p2:         number[i] ← 1 + max(number)
p3:         for all other processes j
p4:             await (number[j] = 0) or (number[i] ≪ number[j])
p5:         critical section
p6:         number[i] ← 0
```

Algorithm 5.3: Bakery algorithm without atomic assignment

```
                boolean array[1..n] choosing ← [false,. . . ,false]
                integer array[1..n] number ← [0,. . . ,0]

     loop forever
p1:     non-critical section
p2:     choosing[i] ← true
p3:     number[i] ← 1 + max(number)
p4:     choosing[i] ← false
p5:     for all other processes j
p6:        await choosing[j] = false
p7:        await (number[j] = 0) or (number[i] ≪ number[j])
p8:     critical section
p9:     number[i] ← 0
```

Algorithm 5.4: Fast algorithm for two processes (outline)

integer gate1 ← 0, gate2 ← 0

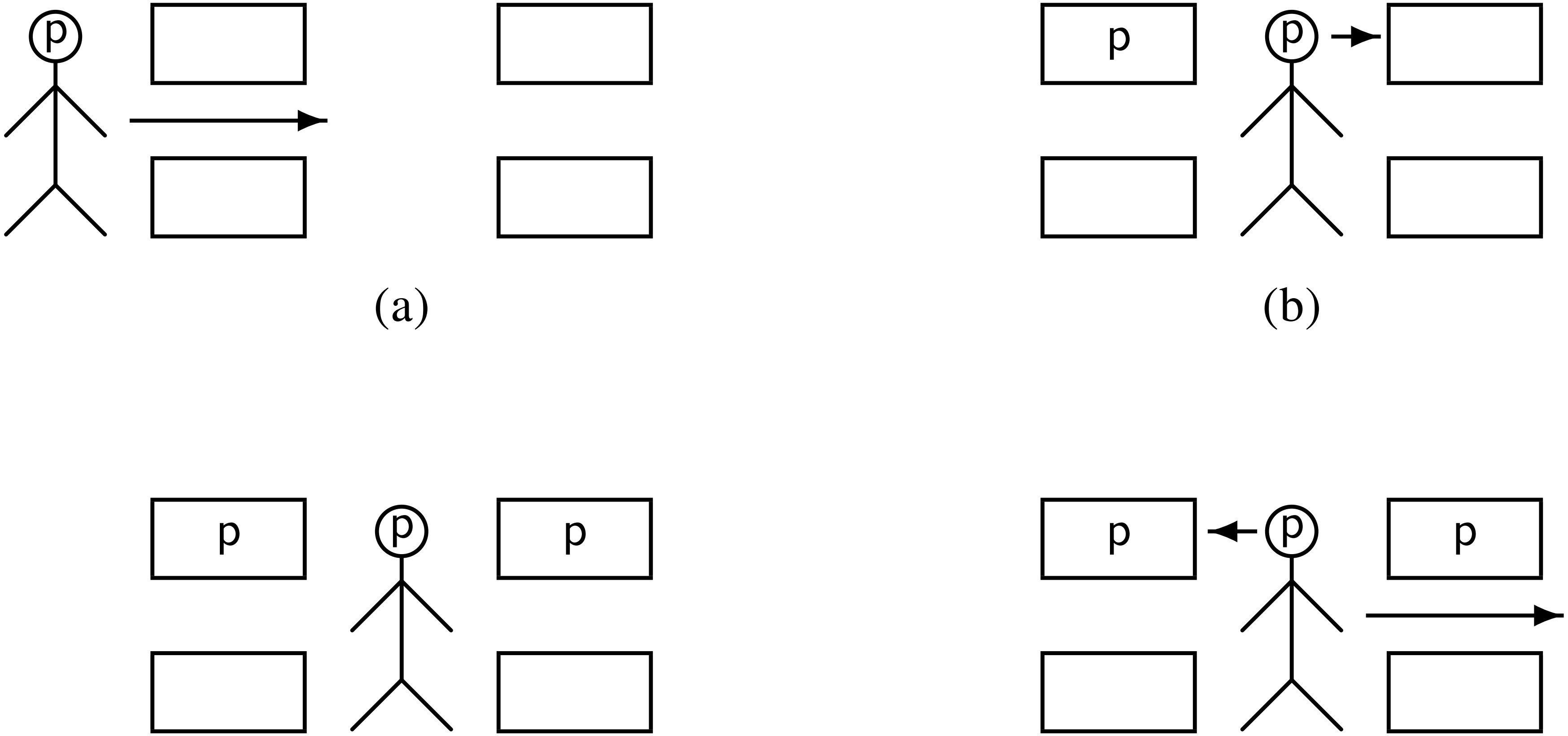
p

```
    loop forever
        non-critical section
p1:     gate1 ← p
p2:     if gate2 ≠ 0 goto p1
p3:     gate2 ← p
p4:     if gate1 ≠ p
p5:         if gate2 ≠ p goto p1
        critical section
p6:     gate2 ← 0
```

q

```
    loop forever
        non-critical section
q1:     gate1 ← q
q2:     if gate2 ≠ 0 goto q1
q3:     gate2 ← q
q4:     if gate1 ≠ q
q5:         if gate2 ≠ q goto q1
        critical section
q6:     gate2 ← 0
```

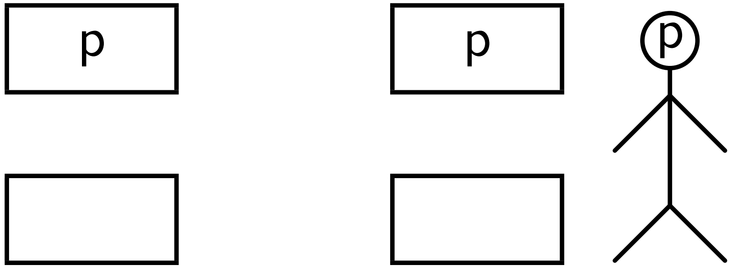
Fast Algorithm - No Contention (1)



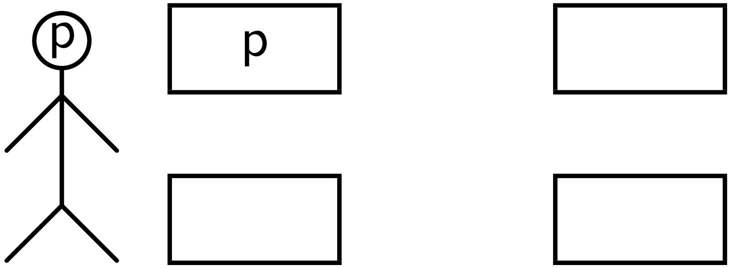
(a) (b)

(c) (d)

Fast Algorithm - No Contention (2)

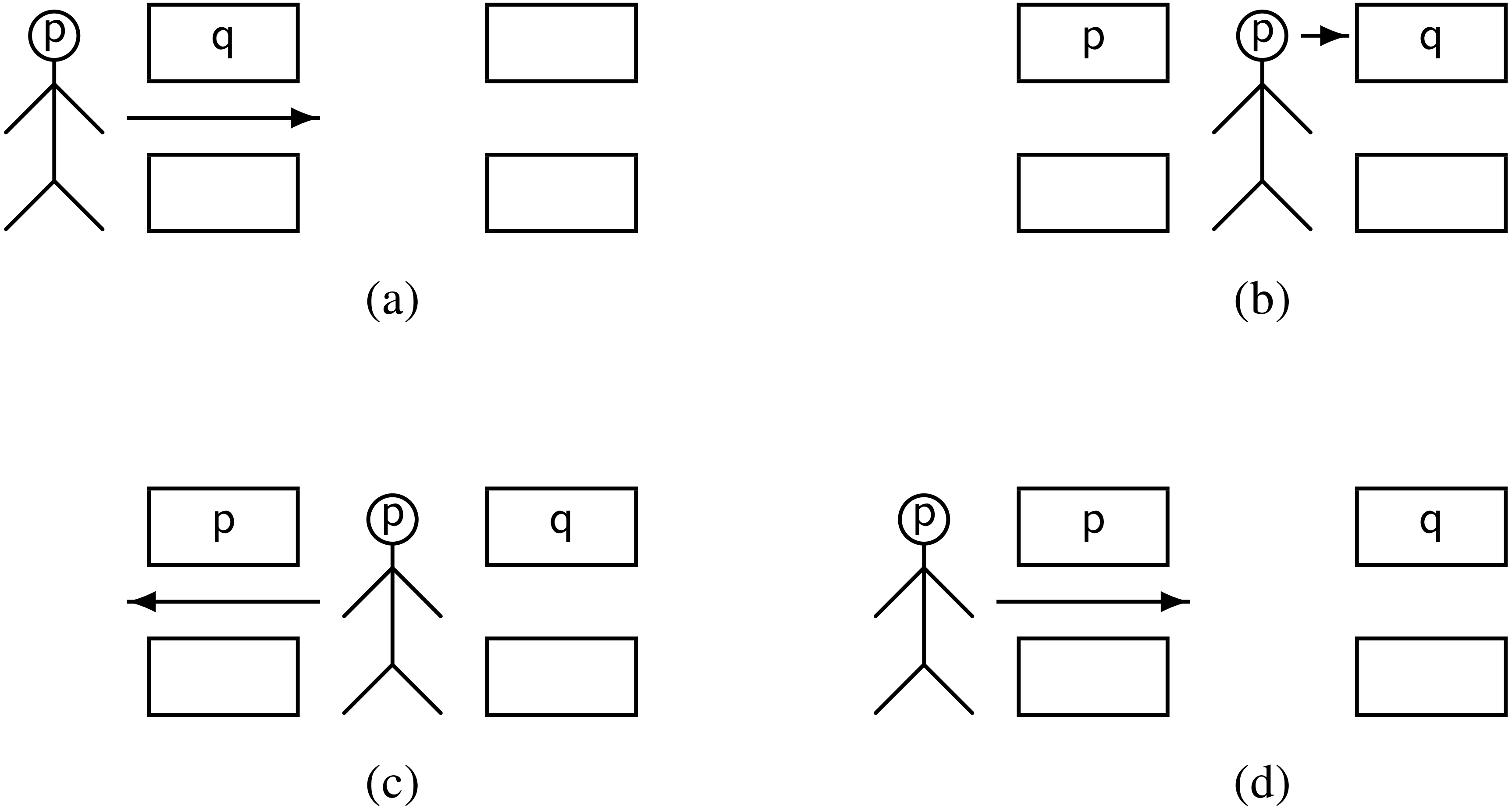


(e)

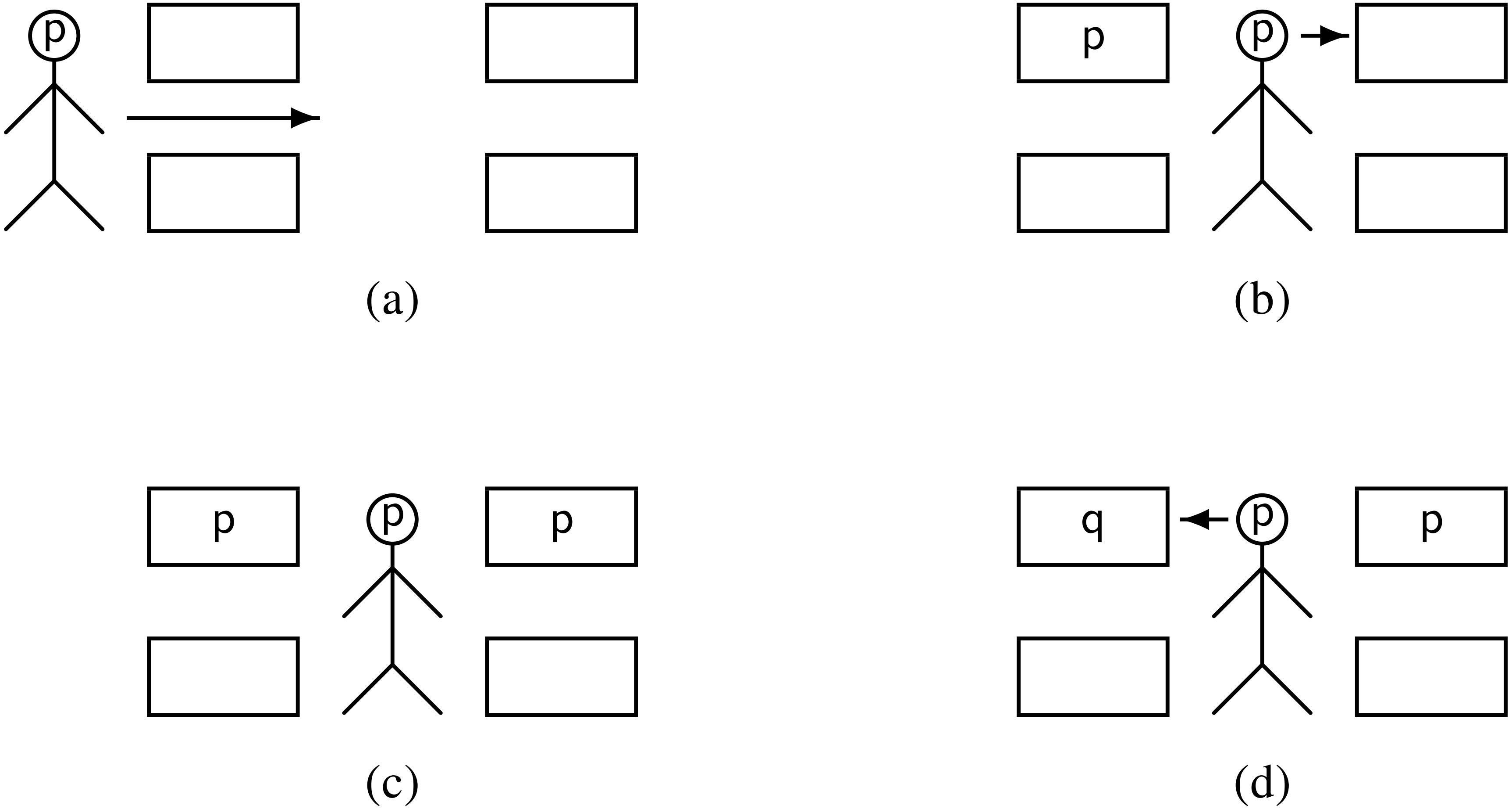


(f)

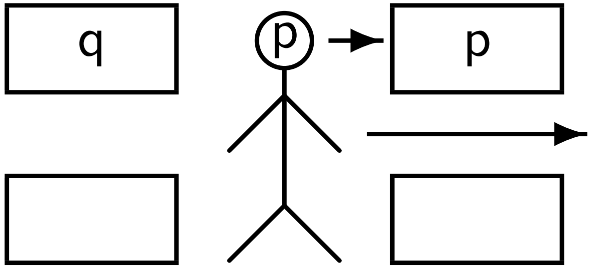
Fast Algorithm - Contention At Gate 2



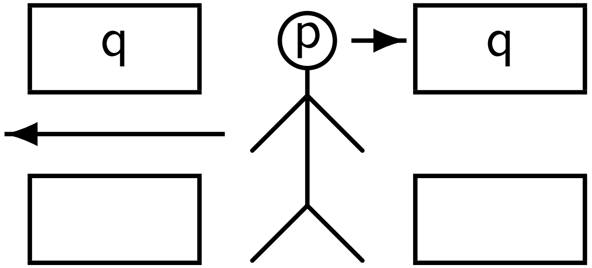
Fast Algorithm - Contention At Gate 1 (1)



Fast Algorithm - Contention At Gate 1 (2)



(e)



(f)

Algorithm 5.5: Fast algorithm for two processes (outline)

integer gate1 ← 0, gate2 ← 0

p

```
          loop forever
              non-critical section
p1:           gate1 ← p
p2:           if gate2 ≠ 0 goto p1
p3:           gate2 ← p
p4:           if gate1 ≠ p
p5:               if gate2 ≠ p goto p1
              critical section
p6:           gate2 ← 0
```

q

```
          loop forever
              non-critical section
q1:           gate1 ← q
q2:           if gate2 ≠ 0 goto q1
q3:           gate2 ← q
q4:           if gate1 ≠ q
q5:               if gate2 ≠ q goto q1
              critical section
q6:           gate2 ← 0
```

Algorithm 5.6: Fast algorithm for two processes

```
integer gate1 ← 0, gate2 ← 0
boolean wantp ← false, wantq ← false
```

p

```
p1:   gate1 ← p
      wantp ← true
p2:   if gate2 ≠ 0
          wantp ← false
          goto p1
p3:   gate2 ← p
p4:   if gate1 ≠ p
          wantp ← false
          await wantq = false
p5:       if gate2 ≠ p goto p1
          else wantp ← true
      critical section
p6:   gate2 ← 0
      wantp ← false
```

q

```
q1:   gate1 ← q
      wantq ← true
q2:   if gate2 ≠ 0
          wantq ← false
          goto q1
q3:   gate2 ← q
q4:   if gate1 ≠ q
          wantq ← false
          await wantp = false
q5:       if gate2 ≠ q goto q1
          else wantq ← true
      critical section
q6:   gate2 ← 0
      wantq ← false
```

Algorithm 5.7: Fisher's algorithm

```
integer gate ← 0

    loop forever
        non-critical section
        loop
p1:         await gate = 0
p2:         gate ← i
p3:         delay
p4:     until gate = i
        critical section
p5:     gate ← 0
```

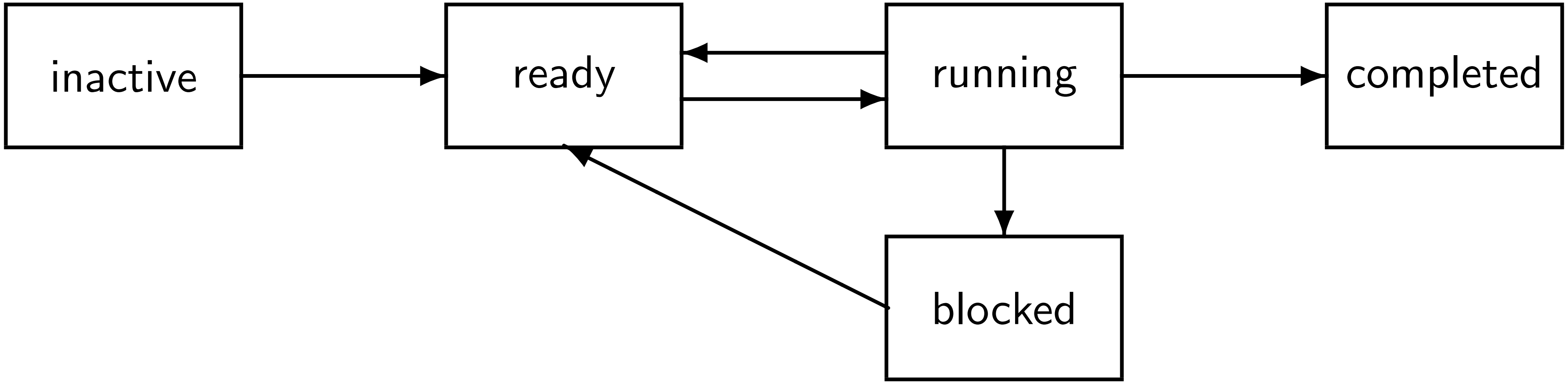
Algorithm 5.8: Lamport's one-bit algorithm

```
boolean array[1..n] want ← [false,...,false]

    loop forever
        non-critical section
p1:     want[i] ← true
p2:     for all processes j < i
p3:         if want[j]
p4:             want[i] ← false
p5:             await not want[j]
                goto p1
p6:     for all processes j > i
p7:         await not want[j]
        critical section
p8:     want[i] ← false
```

Algorithm 5.9: Manna-Pnueli central server algorithm

```
integer request ← 0, respond ← 0

client process i

     loop forever
        non-critical section
p1:     while respond ≠ i
p2:        request ← i
        critical section
p3:     respond ← 0

server process

     loop forever
p4:     await request ≠ 0
p5:     respond ← request
p6:     await respond = 0
p7:     request ← 0
```

State Changes of a Process



Algorithm 6.1: Critical section with semaphores (two processes)

binary semaphore S ← $(1, \emptyset)$

p

```
    loop forever
p1:     non-critical section
p2:     wait(S)
p3:     critical section
p4:     signal(S)
```

q

```
    loop forever
q1:     non-critical section
q2:     wait(S)
q3:     critical section
q4:     signal(S)
```

Algorithm 6.2: Critical section with semaphores (two proc., abbrev.)

binary semaphore S $\leftarrow (1, \emptyset)$

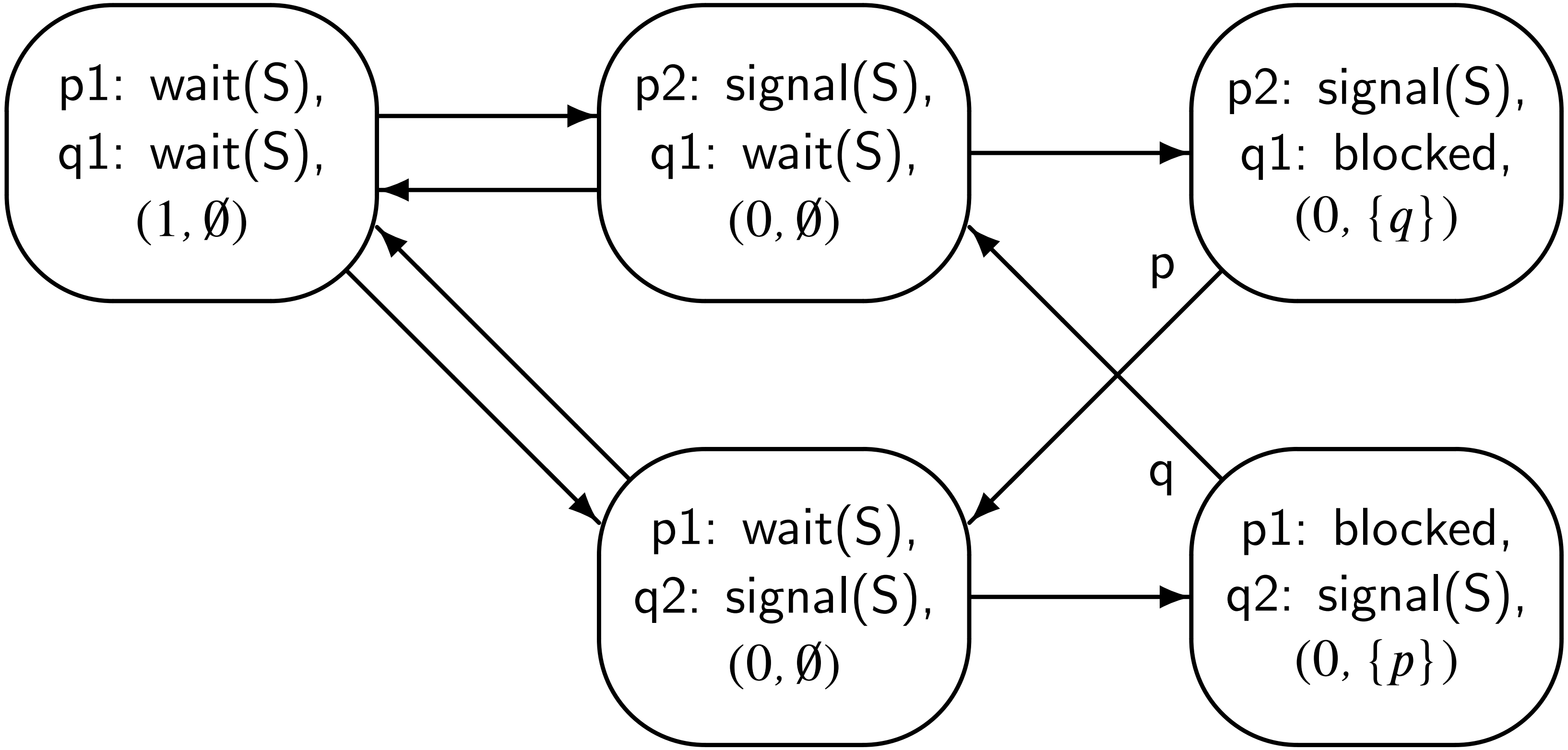
p

```
    loop forever
p1:     wait(S)
p2:     signal(S)
```

q

```
    loop forever
q1:     wait(S)
q2:     signal(S)
```

State Diagram for the Semaphore Solution



p1: wait(S),
q1: wait(S),
$(1, \emptyset)$

p2: signal(S),
q1: wait(S),
$(0, \emptyset)$

p2: signal(S),
q1: blocked,
$(0, \{q\})$

p1: wait(S),
q2: signal(S),
$(0, \emptyset)$

p1: blocked,
q2: signal(S),
$(0, \{p\})$

Algorithm 6.3: Critical section with semaphores (N proc.)

binary semaphore S $\leftarrow (1, \emptyset)$

```
     loop forever
p1:     non-critical section
p2:     wait(S)
p3:     critical section
p4:     signal(S)
```

Algorithm 6.4: Critical section with semaphores (N proc., abbrev.)

```
binary semaphore S ← $(1, \emptyset)$

    loop forever
p1:     wait(S)
p2:     signal(S)
```

Scenario for Starvation

n	Process p	Process q	Process r	S
1	**p1: wait(S)**	q1: wait(S)	r1: wait(S)	$(1, \emptyset)$
2	p2: signal(S)	**q1: wait(S)**	r1: wait(S)	$(0, \emptyset)$
3	p2: signal(S)	q1: blocked	**r1: wait(S)**	$(0, \{q\})$
4	**p1: signal(S)**	q1: blocked	r1: blocked	$(0, \{q, r\})$
5	**p1: wait(S)**	q1: blocked	r2: signal(S)	$(0, \{q\})$
6	p1: blocked	q1: blocked	**r2: signal(S)**	$(0, \{p, q\})$
7	p2: signal(S)	q1: blocked	**r1: wait(S)**	$(0, \{q\})$

Algorithm 6.5: Mergesort

```
integer array A
binary semaphore S1 ← $(0, \emptyset)$
binary semaphore S2 ← $(0, \emptyset)$
```

sort1	**sort2**	**merge**
p1: sort 1st half of A	q1: sort 2nd half of A	r1: wait(S1)
p2: signal(S1)	q2: signal(S2)	r2: wait(S2)
p3:	q3:	r3: merge halves of A

Algorithm 6.6: Producer-consumer (infinite buffer)

```
infinite queue of dataType buffer ← empty queue
semaphore notEmpty ← $(0, \emptyset)$
```

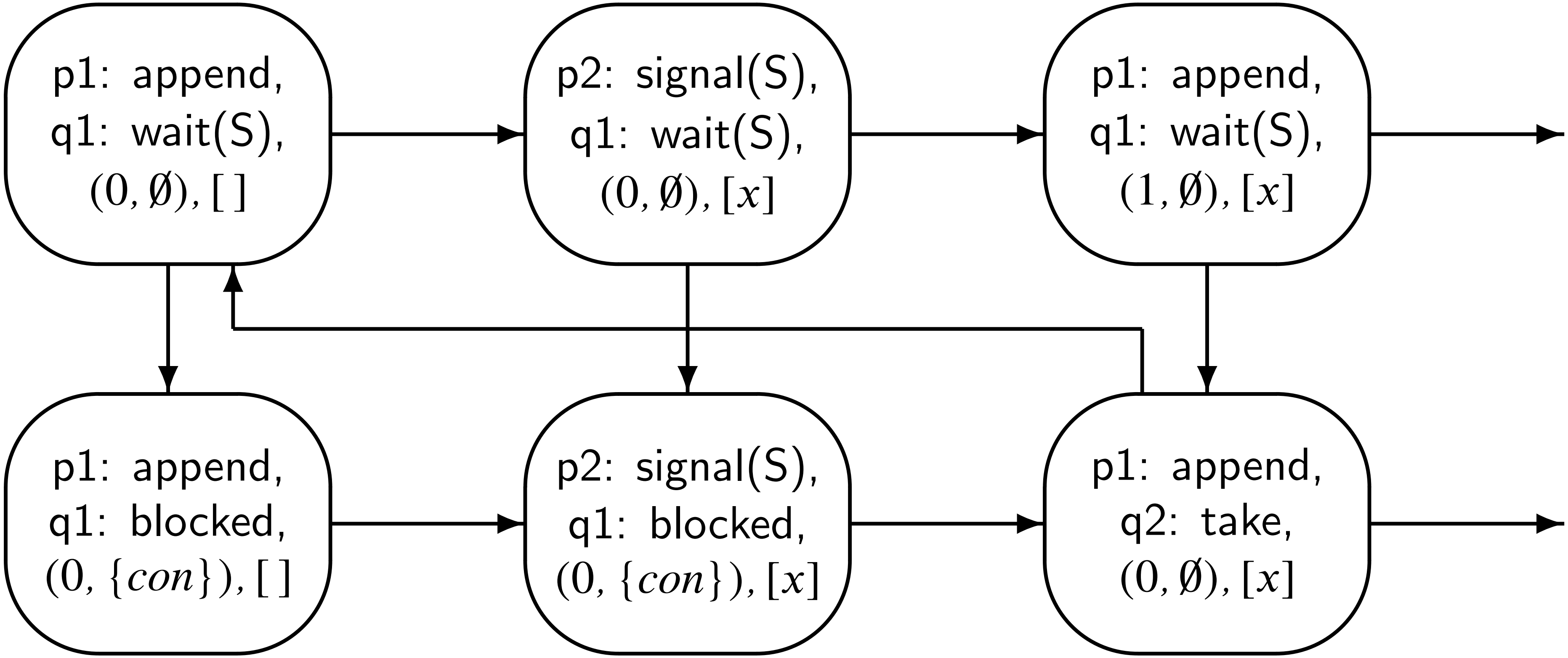
producer

```
    dataType d
    loop forever
p1:     d ← produce
p2:     append(d, buffer)
p3:     signal(notEmpty)
```

consumer

```
    dataType d
    loop forever
q1:     wait(notEmpty)
q2:     d ← take(buffer)
q3:     consume(d)
```

Partial State Diagram for Producer-Consumer with Infinite Buffer



p1: append,
q1: wait(S),
$(0, \emptyset), [\,]$

p2: signal(S),
q1: wait(S),
$(0, \emptyset), [x]$

p1: append,
q1: wait(S),
$(1, \emptyset), [x]$

p1: append,
q1: blocked,
$(0, \{con\}), [\,]$

p2: signal(S),
q1: blocked,
$(0, \{con\}), [x]$

p1: append,
q2: take,
$(0, \emptyset), [x]$

Algorithm 6.7: Producer-consumer (infinite buffer, abbreviated)

```
infinite queue of dataType buffer ← empty queue
semaphore notEmpty ← $(0, \emptyset)$
```

producer

```
     dataType d
     loop forever
p1:     append(d, buffer)
p2:     signal(notEmpty)
```

consumer

```
     dataType d
     loop forever
q1:     wait(notEmpty)
q2:     d ← take(buffer)
```

Algorithm 6.8: Producer-consumer (finite buffer, semaphores)

```
finite queue of dataType buffer ← empty queue
semaphore notEmpty ← $(0, \emptyset)$
semaphore notFull ← $(N, \emptyset)$
```

producer

```
    dataType d
    loop forever
p1:     d ← produce
p2:     wait(notFull)
p3:     append(d, buffer)
p4:     signal(notEmpty)
```

consumer

```
    dataType d
    loop forever
q1:     wait(notEmpty)
q2:     d ← take(buffer)
q3:     signal(notFull)
q4:     consume(d)
```

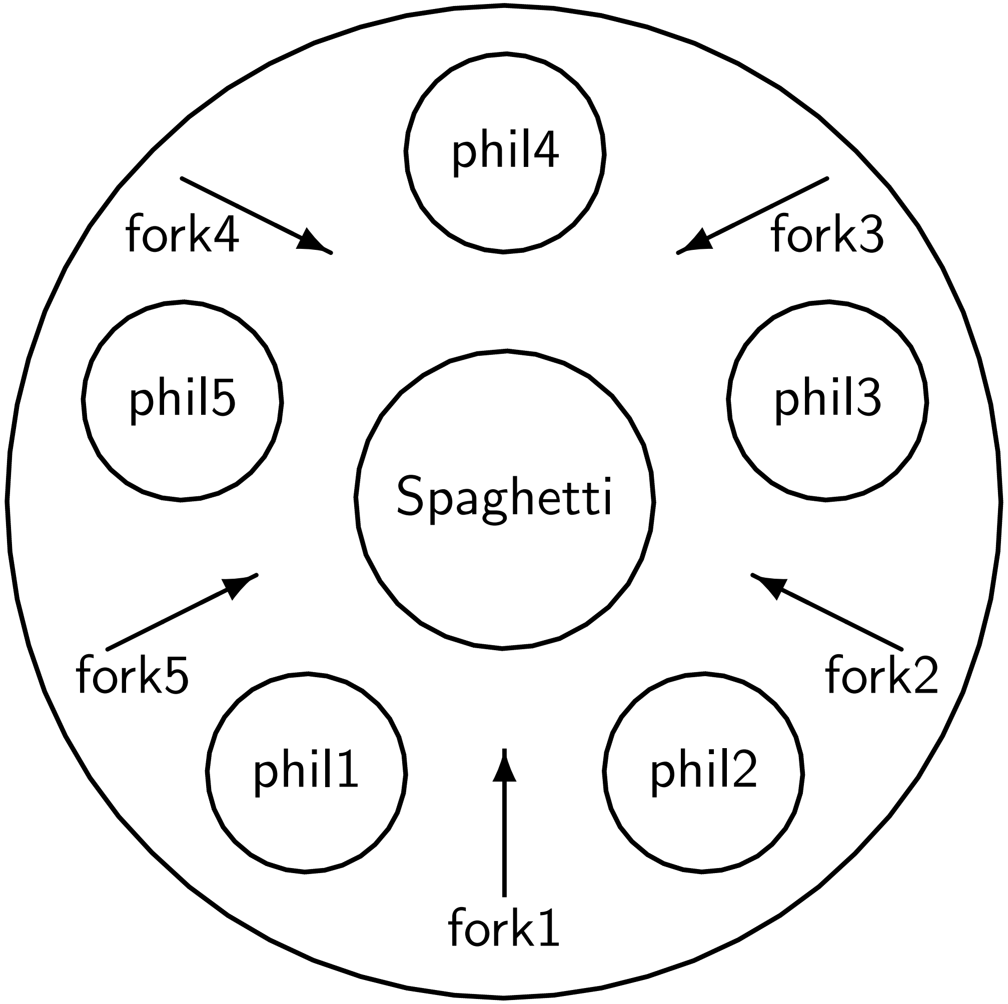
Scenario with Busy Waiting

n	Process p	Process q	S
1	**p1: wait(S)**	q1: wait(S)	1
2	p2: signal(S)	**q1: wait(S)**	0
3	**p2: signal(S)**	q1: wait(S)	0
4	**p1: wait(S)**	q1: wait(S)	1

Algorithm 6.9: Dining philosophers (outline)

```
     loop forever
p1:     think
p2:     preprotocol
p3:     eat
p4:     postprotocol
```

The Dining Philosophers



Algorithm 6.10: Dining philosophers (first attempt)

```
semaphore array [0..4] fork ← [1,1,1,1,1]

     loop forever
p1:     think
p2:     wait(fork[i])
p3:     wait(fork[i+1])
p4:     eat
p5:     signal(fork[i])
p6:     signal(fork[i+1])
```

Algorithm 6.11: Dining philosophers (second attempt)

```
semaphore array [0..4] fork ← [1,1,1,1,1]
semaphore room ← 4

loop forever
    think
    wait(room)
    wait(fork[i])
    wait(fork[i+1])
    eat
    signal(fork[i])
    signal(fork[i+1])
    signal(room)
```

Algorithm 6.12: Dining philosophers (third attempt)

```
semaphore array [0..4] fork ← [1,1,1,1,1]
```

philosopher 4

```
    loop forever
p1:     think
p2:     wait(fork[0])
p3:     wait(fork[4])
p4:     eat
p5:     signal(fork[0])
p6:     signal(fork[4])
```

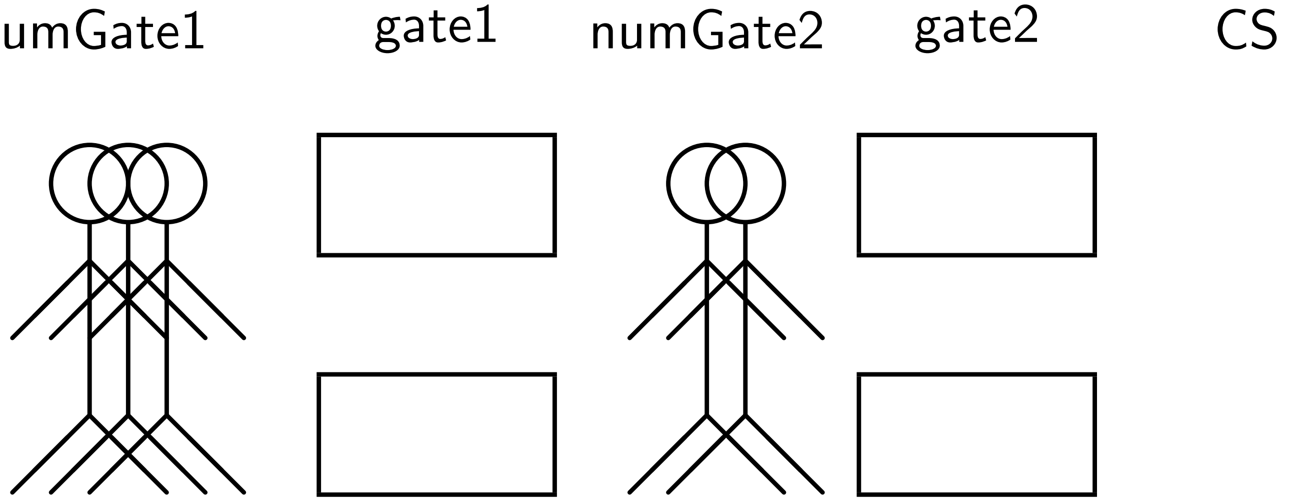
Algorithm 6.13: Barz's algorithm for simulating general semaphores

```
                      binary semaphore S ← 1
                      binary semaphore gate ← 1
                      integer count ← k

     loop forever
        non-critical section
p1:     wait(gate)
p2:     wait(S)                            // Simulated wait
p3:     count ← count − 1
p4:     if count > 0 then
p5:        signal(gate)
p6:     signal(S)
        critical section
p7:     wait(S)                            // Simulated signal
p8:     count ← count + 1
p9:     if count = 1 then
p10:       signal(gate)
p11:    signal(S)
```

Algorithm 6.14: Udding's starvation-free algorithm

```
semaphore gate1 ← 1, gate2 ← 0
integer numGate1 ← 0, numGate2 ← 0

wait(gate1)
numGate1 ← numGate1 + 1
signal(gate1)
wait(gate1)
numGate2 ← numGate2 + 1
numGate1 ← numGate1 − 1        // Statement is missing in the book
if numGate1 > 0
    signal(gate1)
else signal(gate2)
wait(gate2)
numGate2 ← numGate2 − 1
critical section
if numGate2 > 0
    signal(gate2)
else signal(gate1)
```

Udding's Starvation-Free Algorithm



Scenario for Starvation in Udding's Algorithm

n	Process p	Process q	gate1	gate2	nGate1	nGate2
1	**p4: wait(g1)**	q4: wait(g1)	1	0	2	0
2	**p9: wait(g2)**	q9: wait(g2)	0	1	0	2
3	**CS**	q9: wait(g2)	0	0	0	1
4	**p12: signal(g2)**	q9: wait(g2)	0	0	0	1
5	p1: wait(g1)	**CS**	0	0	0	0
6	**p1: wait(g1)**	q13: signal(g1)	0	0	0	0
7	p1: blocked	**q13: signal(g1)**	0	0	0	0
8	p4: wait(g1)	**q1: wait(g1)**	1	0	1	0
9	**p4: wait(g1)**	q4: wait(g1)	1	0	2	0

Semaphores in Java

```
import java.util.concurrent.Semaphore;
class CountSem extends Thread {
  static volatile int n = 0;
  static Semaphore s = new Semaphore(1);

  public void run() {
    int temp;
    for (int i = 0; i < 10; i++) {
      try {
        s.acquire();
      }
      catch (InterruptedException e) {}



```

Semaphores in Java

```
            temp = n;
            n = temp + 1;
            s. release ();
        }
    }

    public static void main(String[] args) {
        /* As before */
    }
}
```

Semaphores in Ada

```
protected type Semaphore(Initial: Natural) is
  entry Wait;
  procedure Signal;
private
  Count: Natural := Initial  ;
end Semaphore;
```

Semaphores in Ada

```
protected body Semaphore is
   entry Wait when Count > 0 is
   begin
      Count := Count - 1;
   end Wait;

   procedure Signal is
   begin
      Count := Count + 1;
      end Signal;
end Semaphore;
```

Busy-Wait Semaphores in Promela

```
/* Copyright (C) 2006 M. Ben-Ari. See copyright.txt */
/* Definition of busy-wait semaphores */
inline wait( s ) {
          atomic { s > 0 ; s-- }
}

inline signal ( s ) { s++ }
```

Weak Semaphores in Promela (three processes)

```
/* Copyright (C) 2006 M. Ben-Ari. See copyright.txt */
/* Weak semaphore */
/* NPROCS - the number of processes - must be defined. */
/* THIS VERSION is specialized for exactly THREE processes */

/* A semaphore is a count plus an array of blocked processes */
typedef Semaphore {
 byte count;
 bool blocked[NPROCS];
};

/* Initialize semaphore to n */
inline initSem(S, n) {
 S.count = n
}
```

Weak Semaphores in Promela (three processes)

```
/* Wait operation: */
/* If count is zero, set blocked and wait for unblocked */
inline wait(S) {
    atomic {
      if
      :: S.count >= 1 -> S.count--
      :: else -> S.blocked[_pid-1] = true; !S.blocked[_pid-1]
      fi
    }
}

/* Signal operation: */
/* If there are blocked processes, remove one nondeterministically */
inline signal (S) {
    atomic {
```

Weak Semaphores in Promela (three processes)

```
        if
        :: S.blocked[0]  -> S.blocked[0] = false
        :: S.blocked[1]  -> S.blocked[1] = false
        :: S.blocked[2]  -> S.blocked[2] = false
        :: else  -> S.count++
        fi
    }
}
```

Weak Semaphores in Promela (N processes)

```
/* Copyright (C) 2006 M. Ben-Ari. See copyright.txt */
/* Weak semaphore */
/* NPROCS - the number of processes - must be defined. */

/* A semaphore is a count plus an array of blocked processes */
typedef Semaphore {
 byte count;
 bool blocked[NPROCS];
     byte i, choice;
};

/* Initialize    semaphore to n */
inline initSem(S, n) {
 S.count = n
}
```

Weak Semaphores in Promela (N processes)

```
/* Wait operation: */
/* If count is zero, set blocked and wait for unblocked */
inline wait(S) {
    atomic {
      if
      :: S.count >= 1 -> S.count--
      :: else -> S.blocked[_pid-1] = true; !S.blocked[_pid-1]
      fi
    }
}

/* Signal operation: */
/* If there are blocked processes, remove each one and */
/* nondeterministically decide whether to replace it in the channel */
/* or exit the operation. */
```

Weak Semaphores in Promela (N processes)

```
inline signal (S) {
  atomic {
    S.i = 0;
    S.choice = 255;
    do
    :: (S.i == NPROCS) -> break
    :: (S.i < NPROCS) && !S.blocked[S.i] -> S.i++
    :: else ->
       if
       :: (S.choice == 255) -> S.choice = S.i
       :: (S.choice != 255) -> S.choice = S.i
       :: (S.choice != 255) ->
       fi ;
       S.i ++
    od;
```

Weak Semaphores in Promela (N processes)

```
        if
        :: S.choice == 255 -> S.count++
        :: else -> S.blocked[S.choice] = false
        fi
    }
}
```

Barz's Algorithm in Promela

```
#define NPROCS 3
#define K      2
byte gate = 1;
int  count = K;
byte critical   = 0;
active [NPROCS] proctype P () {
  do ::
    atomic { gate > 0; gate--; }
    d_step {
      count--;
      if
      ::  count > 0 -> gate++
      ::  else
      fi
    }
```

Barz's Algorithm in Promela

```
        critical ++;
         assert (critical  <= 1);
        critical --;
        d_step {
            count++;
            if
            :: count == 1 -> gate++
            :: else
            fi
        }
    od
}
```

Algorithm 6.15: Semaphore algorithm A

semaphore S ← 1, semaphore T ← 0

p	**q**
p1: wait(S)	q1: wait(T)
p2: write("p")	q2: write("q")
p3: signal(T)	q3: signal(S)

Algorithm 6.16: Semaphore algorithm B

semaphore S1 ← 0, S2 ← 0

p	q	r
p1: write("p")	q1: wait(S1)	r1: wait(S2)
p2: signal(S1)	q2: write("q")	r2: write("r")
p3: signal(S2)	q3:	r3:

Algorithm 6.17: Semaphore algorithm with a loop

```
semaphore S ← 1
boolean B ← false
```

p

```
p1:  wait(S)
p2:  B ← true
p3:  signal(S)
p4:
```

q

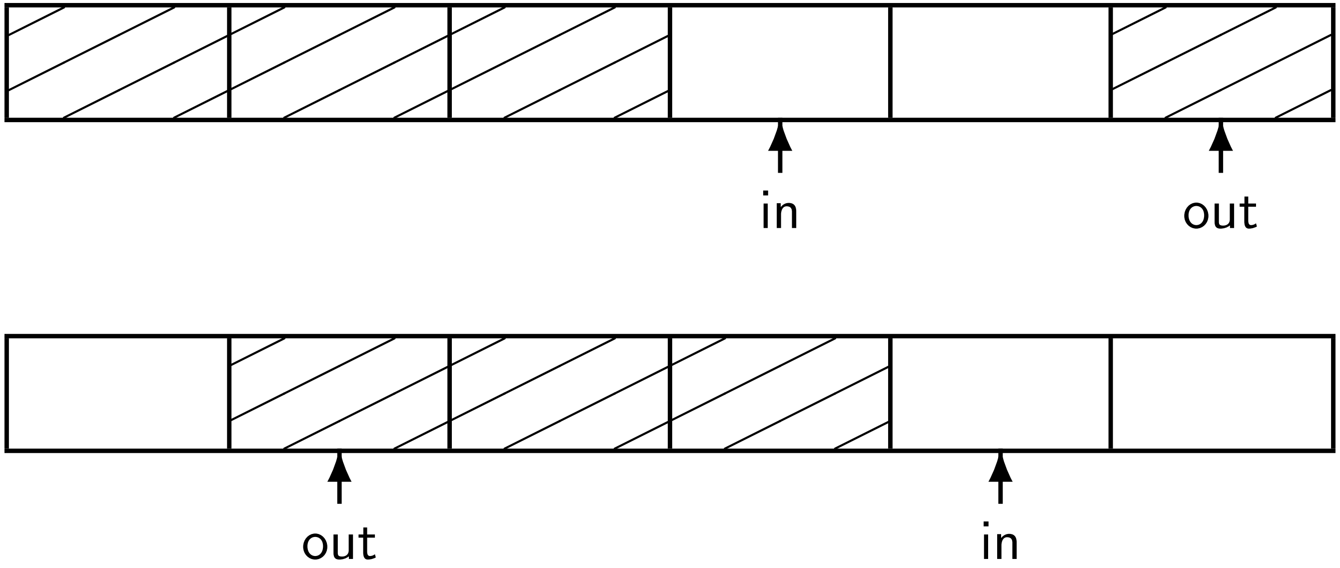
```
q1:  wait(S)
q2:  while not B
q3:      write("*")
q4:  signal(S)
```

Algorithm 6.18: Critical section problem (k out of N processes)

```
binary semaphore S ← 1, delay ← 0
integer count ← k
```

```
integer m
loop forever
   non-critical section
   wait(S)
   count ← count − 1
   m ← count
   signal(S)
   if $m \leq -1$ wait(delay)
   critical section
   wait(S)
   count ← count + 1
   if $count \leq 0$ signal(delay)
   signal(S)
```

Circular Buffer



Algorithm 6.19: Producer-consumer (circular buffer)

```
dataType array [0..N] buffer
integer in, out ← 0
semaphore notEmpty ← $(0, \emptyset)$
semaphore notFull ← $(N, \emptyset)$
```

producer

```
    dataType d
    loop forever
p1:     d ← produce
p2:     wait(notFull)
p3:     buffer[in] ← d
p4:     in ← (in+1) modulo N
p5:     signal(notEmpty)
```

consumer

```
    dataType d
    loop forever
q1:     wait(notEmpty)
q2:     d ← buffer[out]
q3:     out ← (out+1) modulo N
q4:     signal(notFull)
q5:     consume(d)
```

Algorithm 6.20: Simulating general semaphores

```
binary semaphore S $\leftarrow$ 1, gate $\leftarrow$ 0
integer count $\leftarrow$ 0
```

wait

```
p1:  wait(S)
p2:  count $\leftarrow$ count $-$ 1
p3:  if count $<$ 0
p4:      signal(S)
p5:      wait(gate)
p6:  else signal(S)
```

signal

```
p7:  wait(S)
p8:  count $\leftarrow$ count $+$ 1
p9:  if count $\leq$ 0
p10:     signal(gate)
p11: signal(S)
```

Weak Semaphores in Promela with Channels

```
/* Weak semaphore */
/* NPROCS − the number of processes − must be defined. */

/* A semaphore is a count plus a channel */
/* plus a couple of local variables */
typedef Semaphore {
 byte count;
 chan ch = [NPROCS] of { pid };
 byte temp, i;
};

/* Initialize semaphore to n */
inline initSem(S, n) {
 S.count = n
}
```

Weak Semaphores in Promela with Channels

```
/* Wait operation: */
/* If count is zero, place your _pid in the channel */
/*   and block until it is removed. */
inline wait(S) {
   atomic {
     if
     :: S.count >= 1 -> S.count--;
     :: else -> S.ch ! _pid; !(S.ch ?? [eval(_pid)])
     fi
   }
}
/* Signal operation: */
/* If there are blocked processes, remove each one and */
/* nondeterministically decide whether to replace it in the channel */
/* or exit the operation. */
```

Weak Semaphores in Promela with Channels

```
inline signal (S) {
    atomic {
       S.i = len(S.ch);
       if
       :: S.i == 0 -> S.count++ /*No blocked process, increment count*/
       :: else ->
        do
          :: S.i == 1 -> S.ch ? _; break /*Remove only blocked process*/
          :: else -> S.i--;
             S.ch ? S.temp;
             if :: break :: S.ch ! S.temp fi
          od
       fi
    }
}
```

Algorithm 6.21: Readers and writers with semaphores

```
semaphore readerSem ← 0, writerSem ← 0
integer delayedReaders ← 0, delayedWriters ← 0
semaphore entry ← 1
integer readers ← 0, writers ← 0
```

SignalProcess

```
if writers = 0 or delayedReaders > 0
    delayedReaders ← delayedReaders − 1
    signal(readerSem)
else if readers = 0 and writers = 0 and delayedWriters > 0
    delayedWriters ← delayedWriters − 1
    signal(writerSem)
else signal(entry)
```

Algorithm 6.21: Readers and writers with semaphores

StartRead

```
wait(entry)
if writers > 0
    delayedReaders ← delayedReaders + 1
    signal(entry)
    wait(readerSem)
readers ← readers + 1
SignalProcess
```

EndRead

```
wait(entry)
readers ← readers − 1
SignalProcess
```

Algorithm 6.21: Readers and writers with semaphores

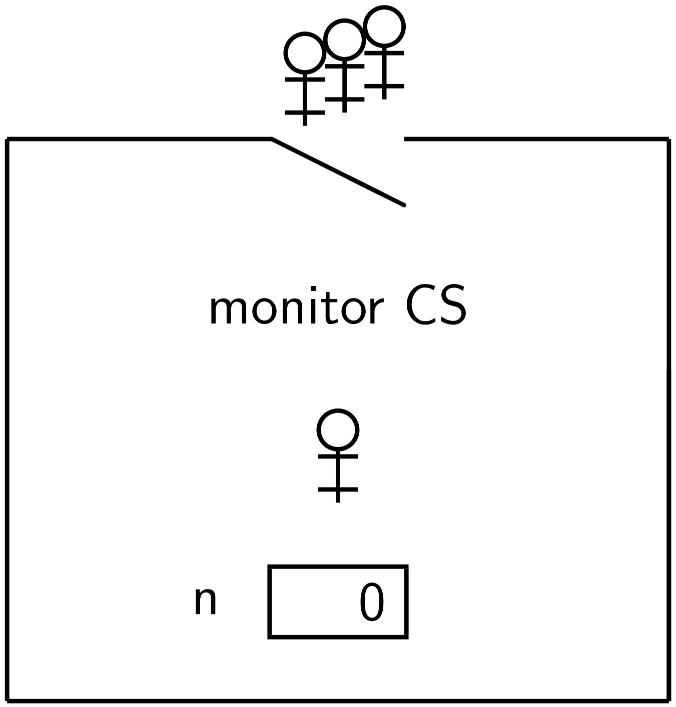
```
     StartWrite
p11: wait(entry)
p12: if writers > 0 or readers > 0
p13:     delayedWriters ← delayedWriters + 1
p14:     signal(entry)
p15:     wait(writerSem)
p16: writers ← writers + 1
p17: SignalProcess
     EndWrite
p18: wait(entry)
p19: writers ← writers − 1
p20: SignalProcess
```

Algorithm 7.1: Atomicity of monitor operations

```
monitor CS
    integer n ← 0

    operation increment
        integer temp
        temp ← n
        n ← temp + 1
```

p	q
p1: CS.increment	q1: CS.increment

Executing a Monitor Operation



Algorithm 7.2: Semaphore simulated with a monitor

```
monitor Sem
    integer s ← k
    condition notZero
    operation wait
        if s = 0
            waitC(notZero)
        s ← s − 1
    operation signal
        s ← s + 1
        signalC(notZero)
```

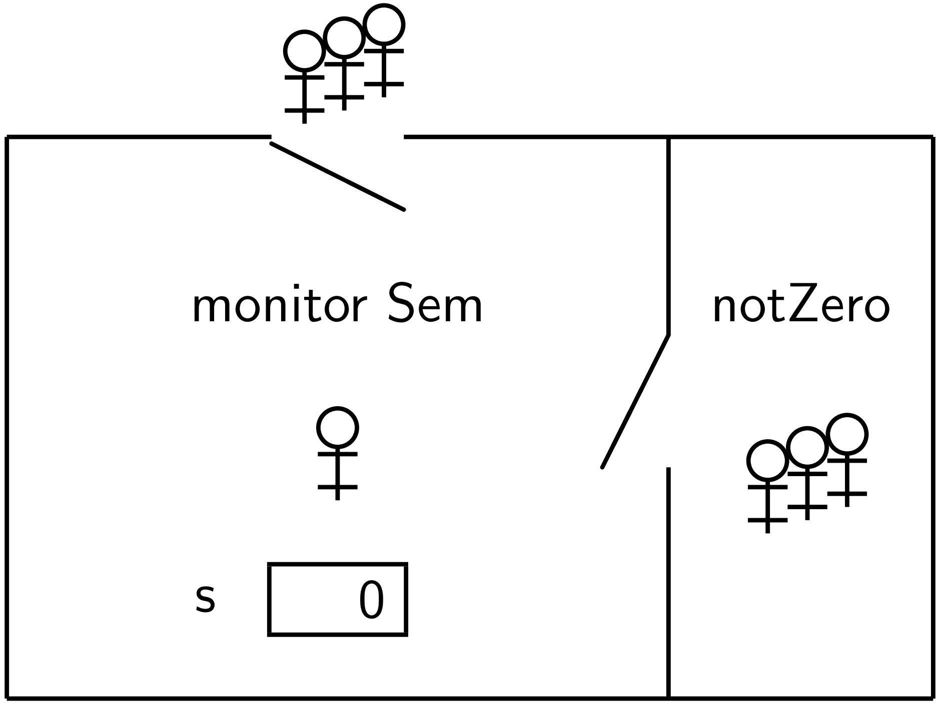
p

```
      loop forever
          non-critical section
p1:       Sem.wait
          critical section
p2:       Sem.signal
```

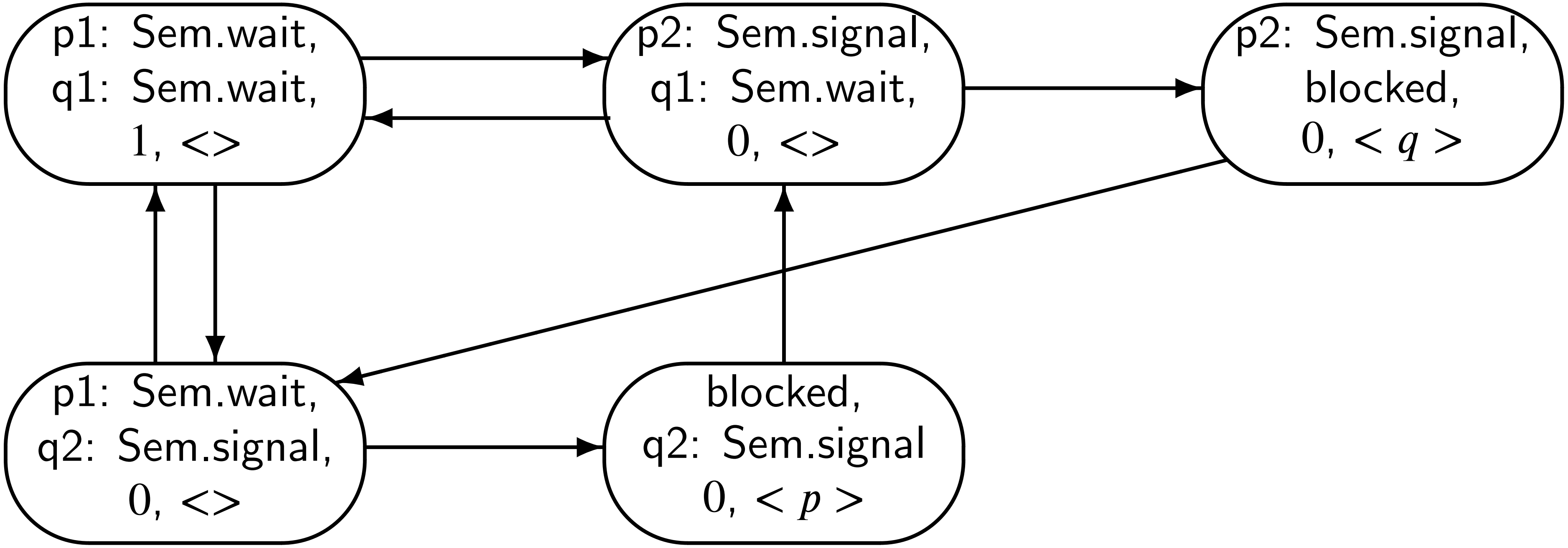
q

```
      loop forever
          non-critical section
q1:       Sem.wait
          critical section
q2:       Sem.signal
```

Condition Variable in a Monitor



State Diagram for the Semaphore Simulation



p1: Sem.wait,
q1: Sem.wait,
1, <>

p2: Sem.signal,
q1: Sem.wait,
0, <>

p2: Sem.signal,
blocked,
0, $< q >$

p1: Sem.wait,
q2: Sem.signal,
0, <>

blocked,
q2: Sem.signal
0, $< p >$

Algorithm 7.3: Producer-consumer (finite buffer, monitor)

```
monitor PC
    bufferType buffer ← empty
    condition notEmpty
    condition notFull
    operation append(datatype V)
        if buffer is full
            waitC(notFull)
        append(V, buffer)
        signalC(notEmpty)
    operation take()
        datatype W
        if buffer is empty
            waitC(notEmpty)
        W ← head(buffer)
        signalC(notFull)
        return W
```

Algorithm 7.3: Producer-consumer (finite buffer, monitor) (continued)

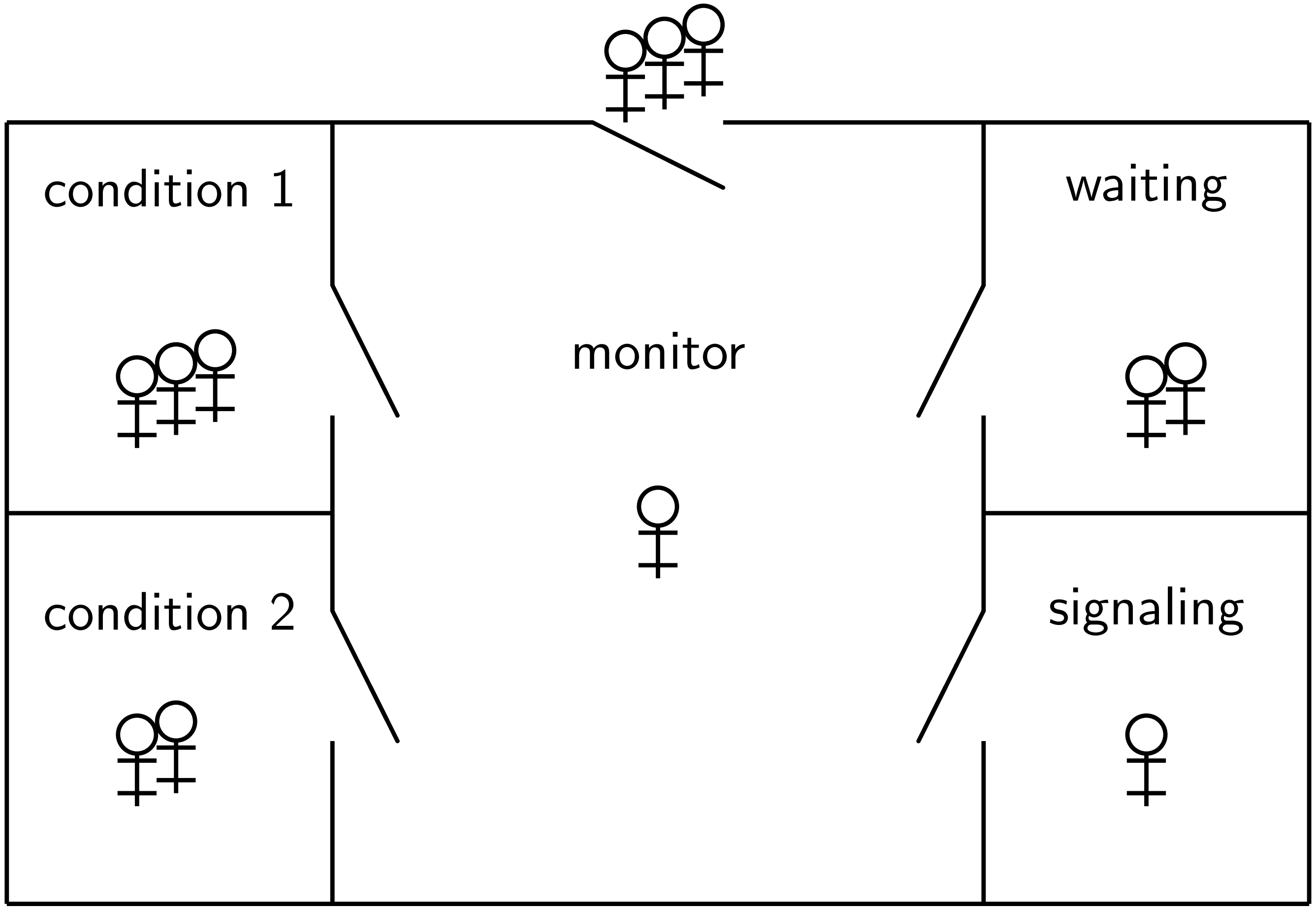
producer

```
     datatype D
     loop forever
p1:     D ← produce
p2:     PC.append(D)
```

consumer

```
     datatype D
     loop forever
q1:     D ← PC.take
q2:     consume(D)
```

The Immediate Resumption Requirement



Algorithm 7.4: Readers and writers with a monitor

```
monitor RW
    integer readers ← 0
    integer writers ← 0
    condition OKtoRead, OKtoWrite
    operation StartRead
        if writers ≠ 0 or not empty(OKtoWrite)
            waitC(OKtoRead)
        readers ← readers + 1
        signalC(OKtoRead)
    operation EndRead
        readers ← readers − 1
        if readers = 0
            signalC(OKtoWrite)
```

Algorithm 7.4: Readers and writers with a monitor (continued)

```
operation StartWrite
    if writers $\neq$ 0 or readers $\neq$ 0
        waitC(OKtoWrite)
    writers $\leftarrow$ writers + 1

operation EndWrite
    writers $\leftarrow$ writers $-$ 1
    if empty(OKtoRead)
        then signalC(OKtoWrite)
        else signalC(OKtoRead)
```

reader	**writer**
p1: RW.StartRead	q1: RW.StartWrite
p2: read the database	q2: write to the database
p3: RW.EndRead	q3: RW.EndWrite

Algorithm 7.5: Dining philosophers with a monitor

```
monitor ForkMonitor
   integer array[0..4] fork ← [2, . . . , 2]
   condition array[0..4] OKtoEat
   operation takeForks(integer i)
      if fork[i] ≠ 2
         waitC(OKtoEat[i])
      fork[i+1] ← fork[i+1] − 1
      fork[i−1] ← fork[i−1] − 1

   operation releaseForks(integer i)
      fork[i+1] ← fork[i+1] + 1
      fork[i−1] ← fork[i−1] + 1
      if fork[i+1] = 2
         signalC(OKtoEat[i+1])
      if fork[i−1] = 2
         signalC(OKtoEat[i−1])
```

Algorithm 7.5: Dining philosophers with a monitor (continued)

philosopher i

```
     loop forever
p1:     think
p2:     takeForks(i)
p3:     eat
p4:     releaseForks(i)
```

Scenario for Starvation of Philosopher 2

n	phil1	phil2	phil3	$f0$	$f1$	$f2$	$f3$	$f4$
1	**take(1)**	take(2)	take(3)	2	2	2	2	2
2	release(1)	take(2)	**take(3)**	1	2	1	2	2
3	release(1)	**take(2) and waitC(OK[2])**	release(3)	1	2	0	2	1
4	**release(1)**	(blocked)	release(3)	1	2	0	2	1
5	**take(1)**	(blocked)	release(3)	2	2	1	2	1
6	release(1)	(blocked)	**release(3)**	1	2	0	2	1
7	release(1)	(blocked)	**take(3)**	1	2	1	2	2

Readers and Writers in C

```
monitor RW {
  int readers = 0, writing = 1;
  condition OKtoRead, OKtoWrite;

  void StartRead() {
    if (writing || !empty(OKtoWrite))
        waitc(OKtoRead);
    readers = readers + 1;
    signalc(OKtoRead);
  }
  void EndRead() {
    readers = readers - 1;
    if (readers == 0)
        signalc(OKtoWrite);
  }
```

Readers and Writers in C

```
    void StartWrite() {
      if (writing || (readers != 0))
          waitc(OKtoWrite);
      writing = 1;
    }

    void EndWrite() {
      writing = 0;
      if (empty(OKtoRead))
          signalc(OKtoWrite);
      else
          signalc(OKtoRead);
    }
}
```

Algorithm 7.6: Readers and writers with a protected object

```
protected object RW
    integer readers ← 0
    boolean writing ← false
    operation StartRead when not writing
        readers ← readers + 1
    operation EndRead
        readers ← readers − 1
    operation StartWrite when not writing and readers = 0
        writing ← true

    operation EndWrite
        writing ← false
```

reader

```
      loop forever
p1:       RW.StartRead
p2:       read the database
p3:       RW.EndRead
```

writer

```
      loop forever
q1:       RW.StartWrite
q2:       write to the database
q3:       RW.EndWrite
```

Context Switches in a Monitor

Process reader	Process writer
waitC(OKtoRead)	operation EndWrite
(blocked)	**writing ← false**
(blocked)	**signalC(OKtoRead)**
readers ← readers + 1	return from EndWrite
signalC(OKtoRead)	return from EndWrite
read the data	**return from EndWrite**
read the data	. . .

Context Switches in a Protected Object

Process reader	Process writer
when not writing	operation EndWrite
(blocked)	**writing ← false**
(blocked)	**when not writing**
(blocked)	**readers ← readers + 1**
read the data	...

Simple Readers and Writers in Ada

```
protected RW is
  procedure Write(I: Integer );
  function Read return Integer;
private
  N: Integer  := 0;
end RW;
```

Simple Readers and Writers in Ada

```
protected body RW is
  procedure Write(I: Integer ) is
  begin
    N := I;
  end Write;
  function Read return Integer  is
  begin
    return N;
  end Read;
end RW;
```

Readers and Writers in Ada

```
protected RW is
    entry StartRead;
    procedure EndRead;
    entry Startwrite ;
    procedure EndWrite;
private
    Readers: Natural :=0;
    Writing: Boolean := false ;
end RW;
```

Readers and Writers in Ada

```
protected body RW is
   entry StartRead
     when not Writing is
   begin
      Readers := Readers + 1;
   end StartRead;

   procedure EndRead is
   begin
      Readers := Readers - 1;
   end EndRead;
```

Readers and Writers in Ada

```
   entry StartWrite
     when not Writing and Readers = 0 is
   begin
      Writing := true;
   end StartWrite;

   procedure EndWrite is
   begin
      Writing := false ;
   end EndWrite;
end RW;
```

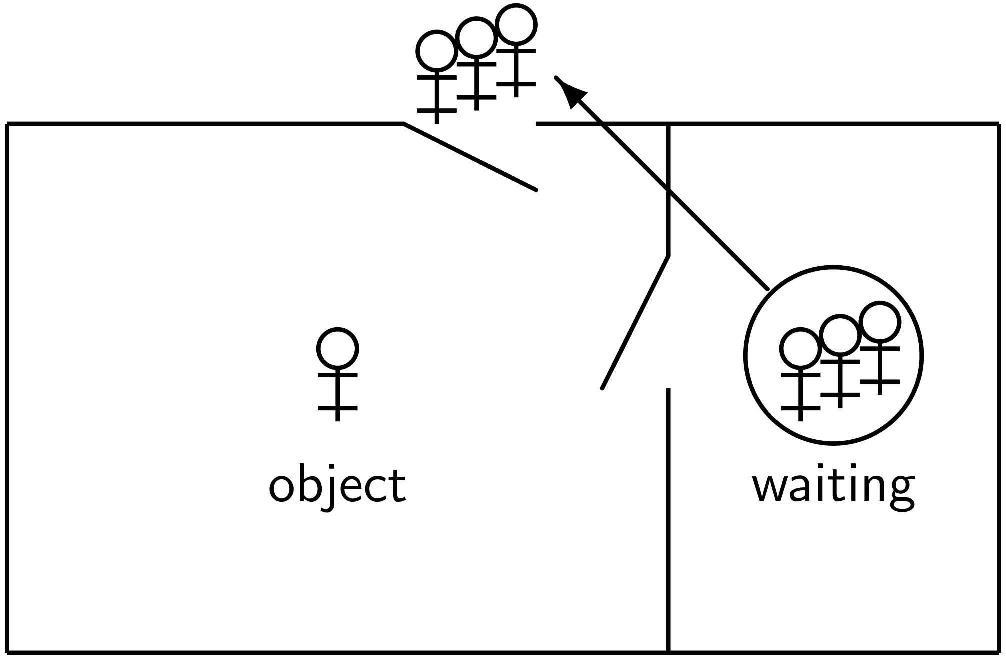
Producer-Consumer in Java

```
class PCMonitor {
  final int N = 5;
  int Oldest = 0, Newest = 0;
  volatile int Count = 0;
  int Buffer[] = new int[N];
  synchronized void Append(int V) {
    while (Count == N)
      try {
         wait();
      } catch (InterruptedException e) {}
    Buffer[Newest] = V;
    Newest = (Newest + 1) % N;
    Count = Count + 1;
    notifyAll ();
  }
```

Producer-Consumer in Java

```
  synchronized int Take() {
    int  temp;
    while (Count == 0)
      try {
         wait();
      } catch (InterruptedException  e) {}
    temp = Buffer[Oldest];
    Oldest  = (Oldest + 1) % N;
    Count = Count - 1;
    notifyAll  ();
    return temp;
  }
}
```

A Monitor in Java With notifyAll



Java Monitor for Readers and Writers

```
class RWMonitor {
  volatile int readers = 0;
  volatile boolean writing = false;
  synchronized void StartRead() {
    while (writing )
      try {
          wait();
      } catch (InterruptedException e) {}
    readers = readers + 1;
    notifyAll ();
  }
  synchronized void EndRead() {
    readers = readers − 1;
    if (readers == 0) notifyAll();
  }
```

Java Monitor for Readers and Writers

```
  synchronized void StartWrite() {
    while (writing || (readers != 0))
      try {
          wait();
      } catch (InterruptedException e) {}
    writing = true;
  }
  synchronized void EndWrite() {
    writing = false;
    notifyAll ();
  }
}
```

Simulating Monitors in Promela

```
/* Copyright (C) 2006 M. Ben-Ari. See copyright.txt */
/* Definitions for monitor */
bool lock = false;

typedef Condition {
    bool gate;
    byte waiting;
}

inline enterMon() {
    atomic {
        !lock;
        lock = true;
    }
}
```

Simulating Monitors in Promela

```

inline leaveMon() {
    lock = false;
}

inline waitC(C) {
    atomic {
        C.waiting++;
        lock = false;       /* Exit monitor */
        C.gate;             /* Wait for gate */
        lock = true;        /* IRR */
        C.gate = false;     /* Reset gate */
        C.waiting--;
    }
}
```

Simulating Monitors in Promela

```

inline signalC(C) {
    atomic {
        if
            /* Signal only if waiting */
        :: (C.waiting > 0) ->
            C.gate = true;
            !lock;        /* IRR - wait for released lock */
            lock = true; /* Take lock again */
        :: else
        fi;
    }
}

#define emptyC(C) (C.waiting == 0)
```

Readers and Writers in Ada (1)

```
 protected RW is

    entry Start_Read;
    procedure End_Read;
    entry Start_Write;
    procedure End_Write;

private
    Waiting_To_Read : integer := 0;
    Readers :  Natural := 0;
    Writing :  Boolean := false ;

end RW;
```

Readers and Writers in Ada (2)

```
protected RW is

   entry StartRead;
   procedure EndRead;
   entry Startwrite ;
   procedure EndWrite;
   function NumberReaders return Natural;

private
   entry ReadGate;
   entry WriteGate;
   Readers: Natural :=0;
   Writing: Boolean := false ;

end RW;
```

Algorithm 8.1: Producer-consumer (channels)

channel of integer ch

producer

```
    integer x
    loop forever
p1:     x ← produce
p2:     ch ⇐ x
```

consumer

```
    integer y
    loop forever
q1:     ch ⇒ y
q2:     consume(y)
```

Algorithm 8.2: Conway's problem

```
constant integer MAX ← 9
constant integer K ← 4
channel of integer inC, pipe, outC
```

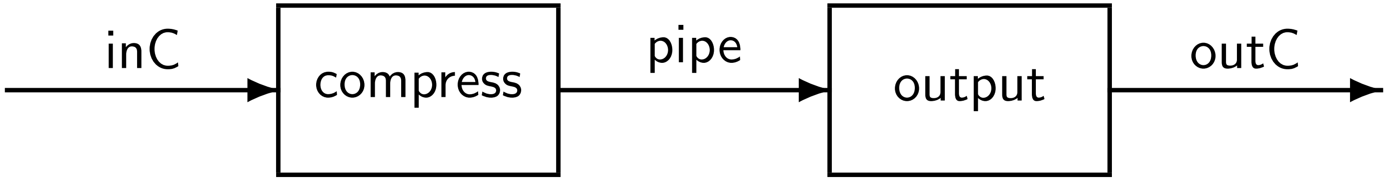
compress

```
char c, previous ← 0
integer n ← 0
inC ⇒ previous
loop forever
    inC ⇒ c
    if (c = previous) and
            (n < MAX − 1)
        n ← n + 1
    else
        if n > 0
            pipe ⇐ intToChar(n+1)
            n ← 0
        pipe ⇐ previous
        previous ← c
```

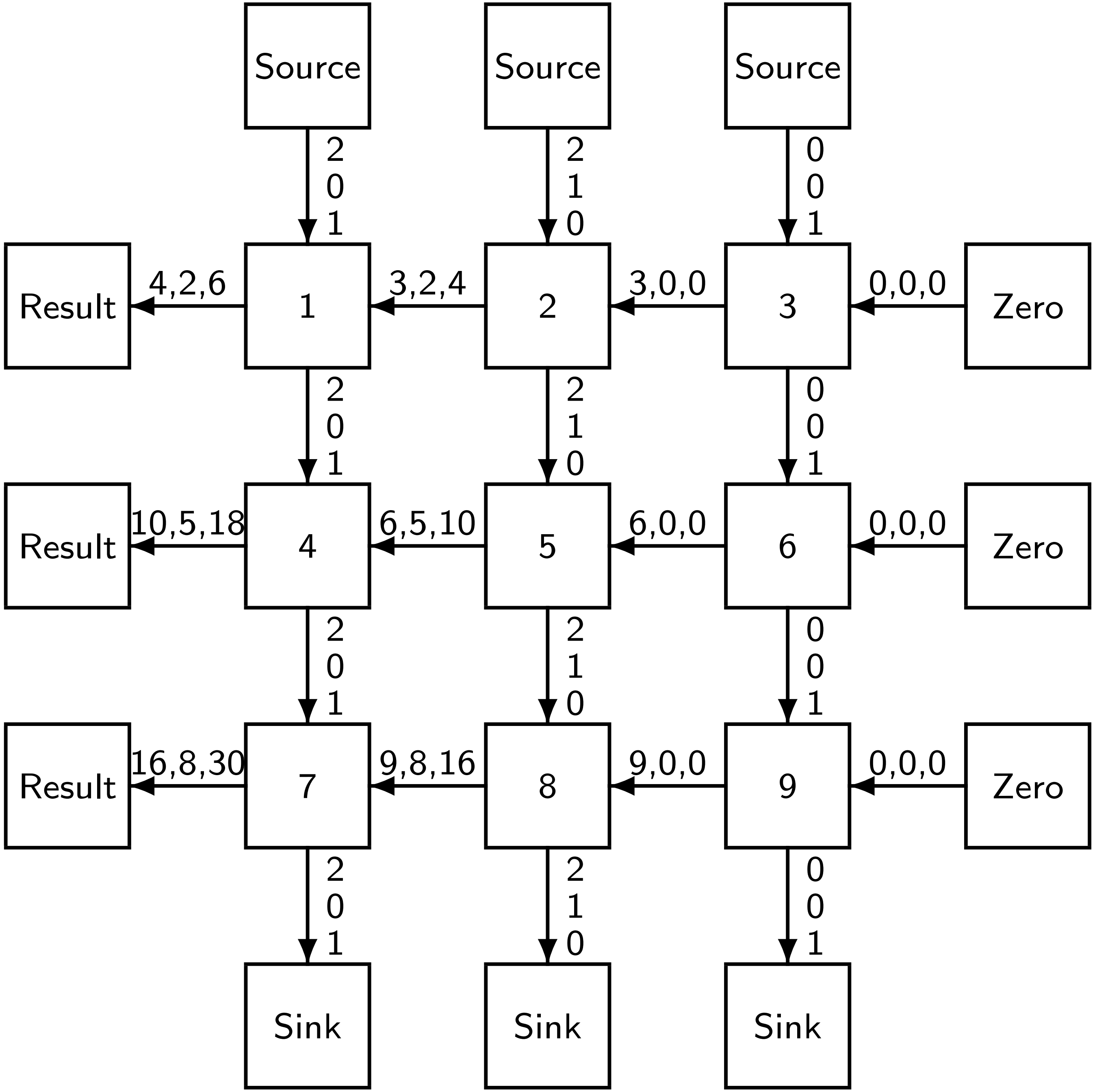
output

```
char c
integer m ← 0

loop forever
    pipe ⇒ c
    outC ⇐ c

    m ← m + 1

    if m >= K
        outC ⇐ newline
        m ← 0
```

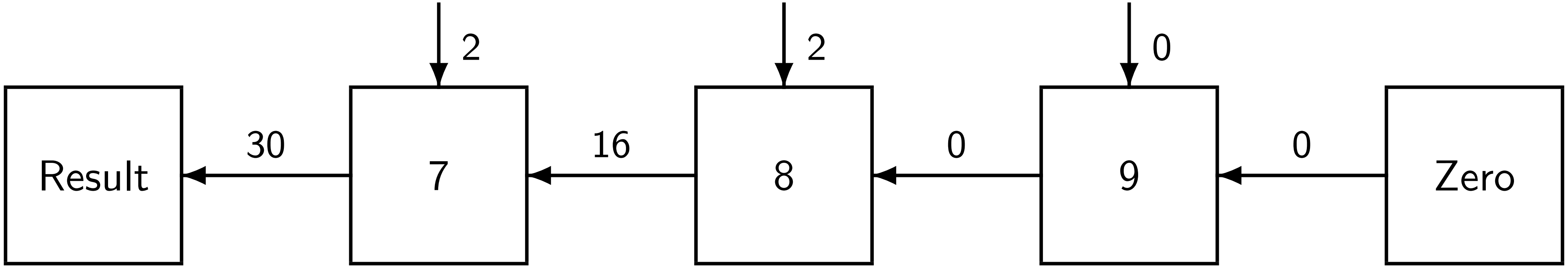
Conway's Problem



Process Array for Matrix Multiplication



Computation of One Element



Algorithm 8.3: Multiplier process with channels

```
integer FirstElement
channel of integer North, East, South, West
integer Sum, integer SecondElement

    loop forever
p1:     North ⇒ SecondElement
p2:     East ⇒ Sum
p3:     Sum ← Sum + FirstElement · SecondElement
p4:     South ⇐ SecondElement
p5:     West ⇐ Sum
```

Algorithm 8.4: Multiplier with channels and selective input

```
                    integer FirstElement
                    channel of integer North, East, South, West
                    integer Sum, integer SecondElement

    loop forever
      either
p1:     North ⇒ SecondElement
p2:     East ⇒ Sum
      or
p3:     East ⇒ Sum
p4:     North ⇒ SecondElement
p5:   South ⇐ SecondElement
p6:   Sum ← Sum + FirstElement · SecondElement
p7:   West ⇐ Sum
```

Algorithm 8.5: Dining philosophers with channels

channel of boolean forks[5]

philosopher i

```
     boolean dummy
     loop forever
p1:     think
p2:     forks[i] ⇒ dummy
p3:     forks[i+1] ⇒ dummy
p4:     eat
p5:     forks[i] ⇐ true
p6:     forks[i+1] ⇐ true
```

fork i

```
     boolean dummy
     loop forever
q1:     forks[i] ⇐ true
q2:     forks[i] ⇒ dummy
q3:
q4:
q5:
q6:
```

Conway's Problem in Promela

```
#define N 9
#define K 4
chan inC, pipe, outC = [0] of { byte };

active proctype Compress() {
  byte previous, c, count = 0;
  inC ? previous;
  do
  :: inC ? c ->
     if
     :: (c == previous) && (count < N-1) -> count++
     :: else ->
```

Conway's Problem in Promela

```
            if
            :: count > 0 ->
                pipe ! count+1;
                count = 0
            :: else
            fi ;
            pipe ! previous ;
            previous  = c;
        fi
    od
}
```

Conway's Problem in Promela

```
active proctype Output() {
    byte c, count = 0;
    do
    :: pipe ? c;
       outC ! c;
       count++;
       if
       :: count >= K ->
          outC ! '\n';
          count = 0
       :: else
       fi
    od
}
```

Multiplier Process in Promela

```
proctype Multiplier (byte Coeff;
    chan North; chan East; chan South; chan West) {
  byte Sum, X;
  for (i ,0, SIZE-1)
    if :: North ? X -> East ? Sum;
       :: East  ? Sum -> North ? X;
    fi ;
    South ! X;
    Sum = Sum + X*Coeff;
    West ! Sum;
  rof (i )
}
```

Algorithm 8.6: Rendezvous

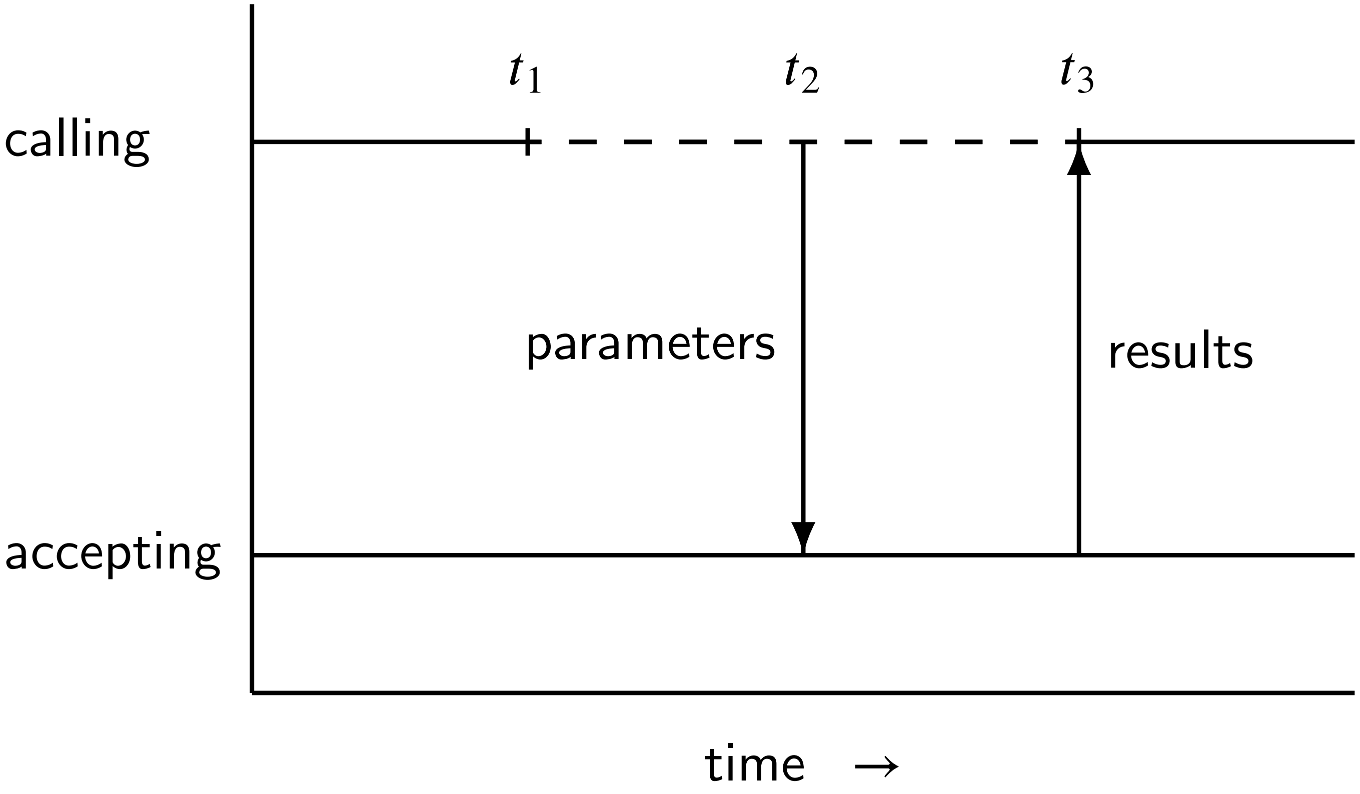
client

```
integer parm, result
loop forever
   parm ← . . .
   server.service(parm, result)
   use(result)
```

server

```
integer p, r
loop forever

   accept service(p, r)
      r ← do the service(p)
```

Timing Diagram for a Rendezvous



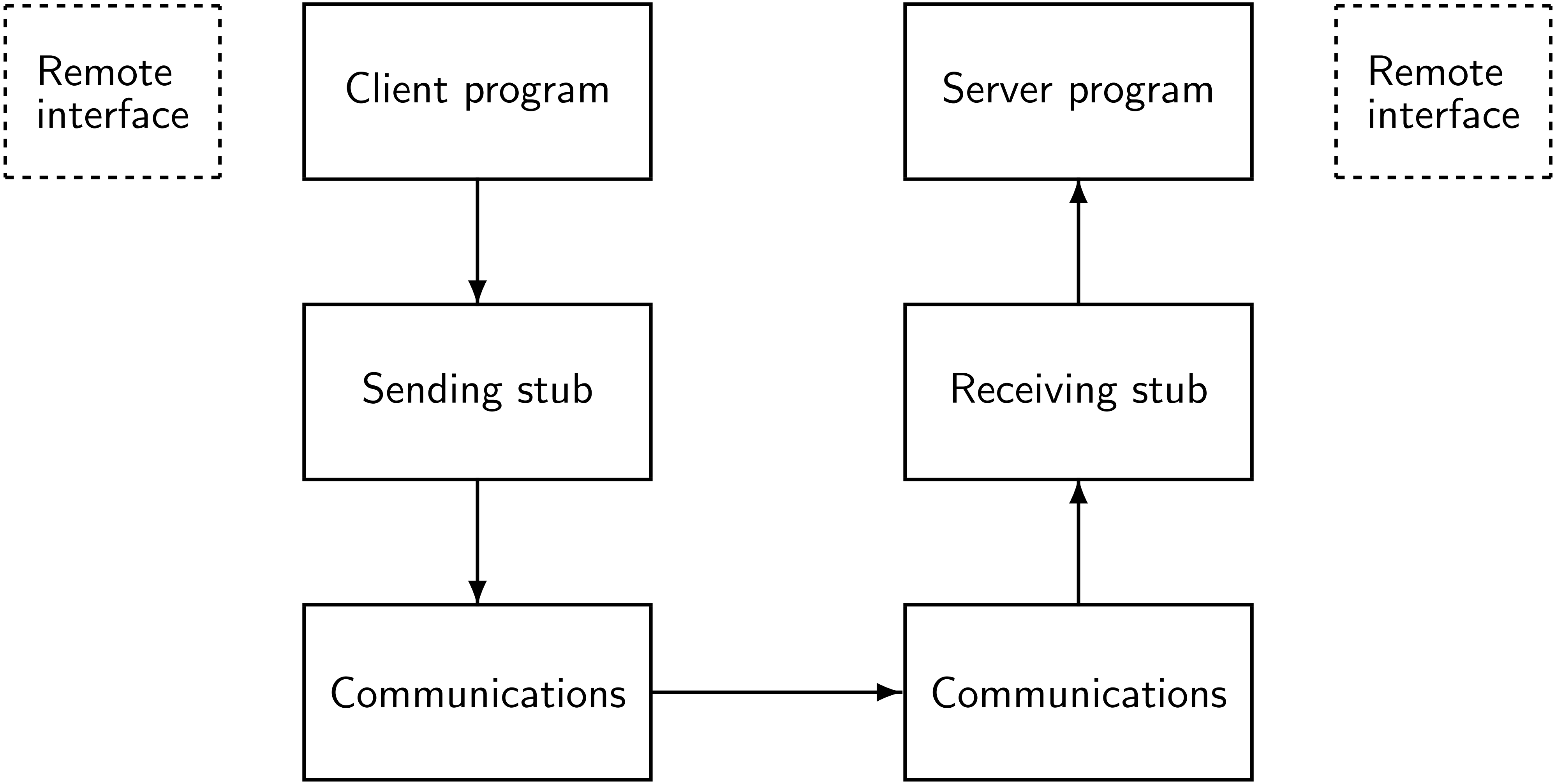
Bounded Buffer in Ada

```
task body Buffer is
  B: Buffer_Array;
  In_Ptr, Out_Ptr, Count: Index := 0;

begin
  loop
    select
      when Count < Index'Last =>
        accept Append(I: in Integer ) do
            B(In_Ptr) := I;
        end Append;
      Count := Count + 1; In_Ptr := In_Ptr + 1;
    or


```

Bounded Buffer in Ada

```
         when Count > 0 =>
            accept Take(I: out Integer ) do
                  I := B(Out_Ptr);
            end Take;
         Count := Count - 1; Out_Ptr := Out_Ptr + 1;
      or
            terminate;
      end select;
   end loop;
end Buffer;
```

Remote Procedure Call



Remote interface

Client program

Sending stub

Communications

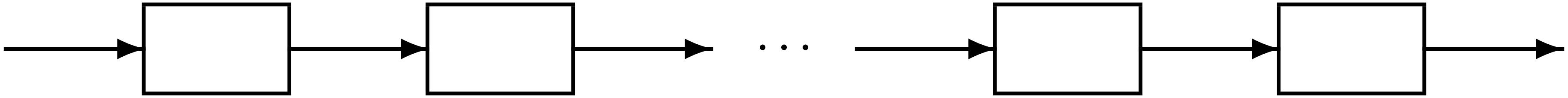
Server program

Receiving stub

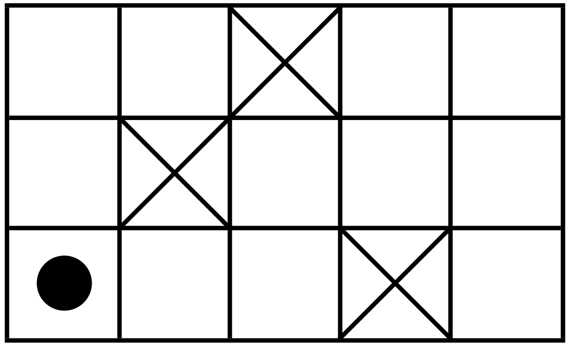
Communications

Remote interface

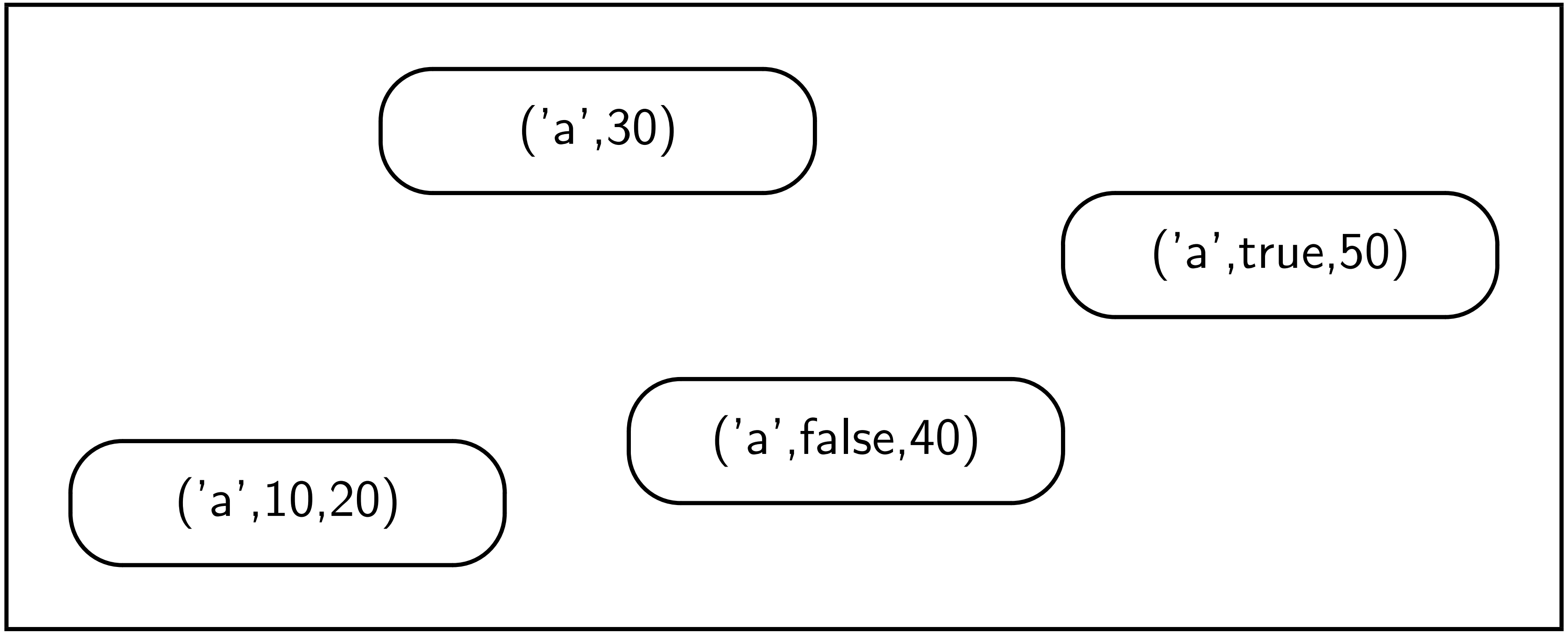
Pipeline Sort



Hoare's Game



A Space



('a',30)

('a',true,50)

('a',false,40)

('a',10,20)

Algorithm 9.1: Critical section problem in Linda

```
     loop forever
p1:     non-critical section
p2:     removenote('s')
p3:     critical section
p4:     postnote('s')
```

Algorithm 9.2: Client-server algorithm in Linda

client

```
    constant integer me ← . . .
    serviceType service
    dataType result, parm
p1: service ← // Service requested
p2: postnote('S', me, service, parm)
p3: removenote('R', me, result)
```

server

```
    integer client
    serviceType s
    dataType r, p
q1: removenote('S', client, s, p)
q2: r ← do (s, p)
q3: postnote('R', client, r)
```

Algorithm 9.3: Specific service

client

```
     constant integer me ← . . .
     serviceType service
     dataType result, parm
p1:  service ← // Service requested
p2:  postnote('S', me, service, parm)
p3:
p4:  removenote('R', me, result)
```

server

```
     integer client
     serviceType s
     dataType r, p
q1:  s ← // Service provided
q2:  removenote('S', client, s=, p)
q3:  r ← do (s, p)
q4:  postnote('R', client, r)
```

Algorithm 9.4: Buffering in a space

producer

```
     integer count ← 0
     integer v
     loop forever
p1:     v ← produce
p2:     postnote('B', count, v)
p3:     count ← count + 1
```

consumer

```
     integer count ← 0
     integer v
     loop forever
q1:     removenote('B', count=, v)
q2:     consume(v)
q3:     count ← count + 1
```

Algorithm 9.5: Multiplier process with channels in Linda

```
parameters: integer FirstElement
parameters: integer North, East, South, West
integer Sum, integer SecondElement

integer Sum, integer SecondElement

loop forever
p1:    removenote('E', North=, SecondElement)
p2:    removenote('S', East=, Sum)
p3:    Sum ← Sum + FirstElement · SecondElement
p4:    postnote('E', South, SecondElement)
p5:    postnote('S', West, Sum)
```

Algorithm 9.6: Matrix multiplication in Linda

constant integer n ← . . .

master

```
integer i, j, result
integer r, c

for i from 1 to n
    for j from 1 to n
        postnote('T', i, j)
for i from 1 to n
    for j from 1 to n
        removenote('R', r, c, result)
        print r, c, result
```

worker

```
integer r, c, result
integer array[1..n] vec1, vec2
loop forever
    removenote('T', r, c)
    readnote('A', r=, vec1)
    readnote('B', c=, vec2)
    result ← vec1 · vec2
    postnote('R', r, c, result)
```

Algorithm 9.7: Matrix multiplication in Linda with granularity

```
constant integer n ← . . .
constant integer chunk ← . . .
```

master

```
integer i, j, result
integer r, c

for i from 1 to n
    for j from 1 to n step by chunk
        postnote('T', i, j)
for i from 1 to n
    for j from 1 to n
        removenote('R', r, c, result)
        print r, c, result
```

worker

```
integer r, c, k, result
integer array[1..n] vec1, vec2
loop forever
    removenote('T', r, k)
    readnote('A', r=, vec1)
    for c from k to k+chunk-1
        readnote('B', c=, vec2)
        result ← vec1 · vec2
        postnote('R', r, c, result)
```

Definition of Notes in Java

```
public class Note {
    public String id;
    public Object[] p;

    // Constructor for an array of objects
    public Note (String id, Object[] p) {
        this.id = id;
        if (p != null) this.p = p.clone();
    }

    // Constructor for a single integer
    public Note (String id, int p1) {
        this(id, new Object[]{new Integer(p1)});
    }

```

Definition of Notes in Java

```
    // Accessor for a single integer value
    public int get(int i) {
        return ((Integer)p[i]).intValue();
    }
}
```

Matrix Multiplication in Java

```
private class Worker extends Thread {
    public void run() {
        Note task = new Note("task");
        while (true) {
            Note t = space.removenote(task);
            int row = t.get(0), col = t.get(1);
            Note r = space.readnote(match("a", row));
            Note c = space.readnote(match("b", col));
            int ip = 0;
            for (int i = 1; i <= SIZE; i++)
                ip = ip + r.get(i)*c.get(i);
            space.postnote(new Note("result", row, col, ip));
        }
    }
}
```

Matrix Multiplication in Promela

```
chan space = [25] of { byte, short, short, short, short };

active[WORKERS] proctype Worker() {
    short row, col, ip, r1, r2, r3, c1, c2, c3;
    do
    :: space ?? 't', row, col, _, _;
       space ?? <'a', eval(row), r1, r2, r3>;
       space ?? <'b', eval(col), c1, c2, c3>;
       ip = r1*c1 + r2*c2 + r3*c3;
       space ! 'r', row, col, ip, 0;
    od;
}
```

Algorithm 9.8: Matrix multiplication in Linda (exercise)

constant integer $n \leftarrow \ldots$

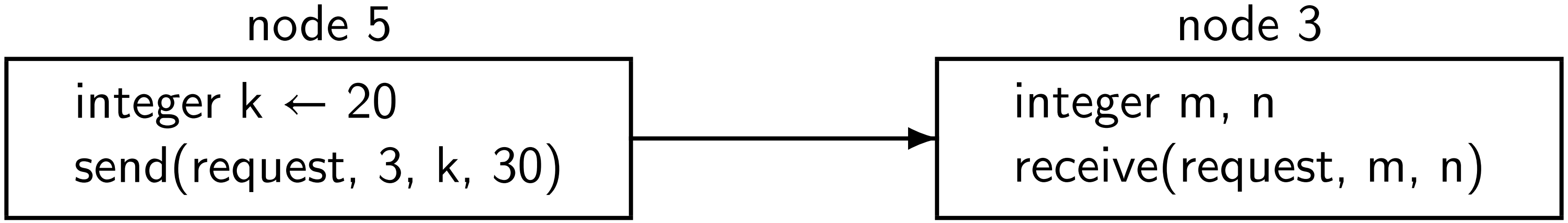
master

```
integer i, j, result
integer r, c

postnote('T', 0)




for i from 1 to n
    for j from 1 to n
        removenote('R', r, c, result)
        print r, c, result
```

worker

```
integer i, r, c, result
integer array[1..n] vec1, vec2
loop forever
    removenote('T' i)
    if $i < (n \cdot n) - 1$
        postnote('T', i+1)
    $r \leftarrow (i / n) + 1$
    $c \leftarrow (i \text{ modulo } n) + 1$
    readnote('A', r=, vec1)
    readnote('B', c=, vec2)
    $result \leftarrow vec1 \cdot vec2$
    postnote('R', r, c, result)
```

Sending and Receiving Messages



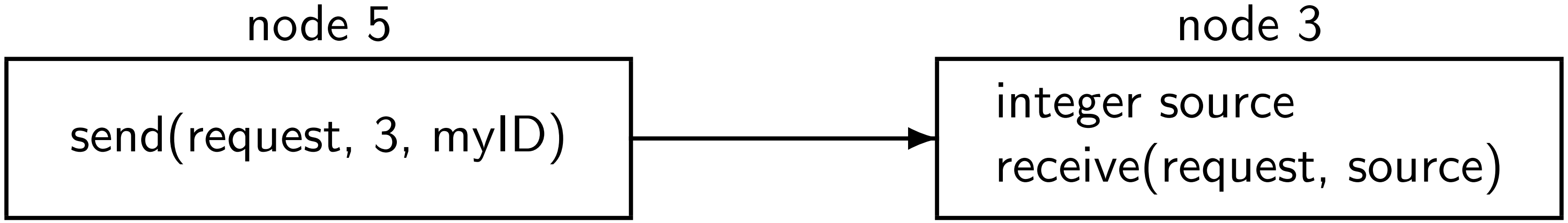
node 5

integer k ← 20
send(request, 3, k, 30)

node 3

integer m, n
receive(request, m, n)

Sending a Message and Expecting a Reply



node 5

send(request, 3, myID)

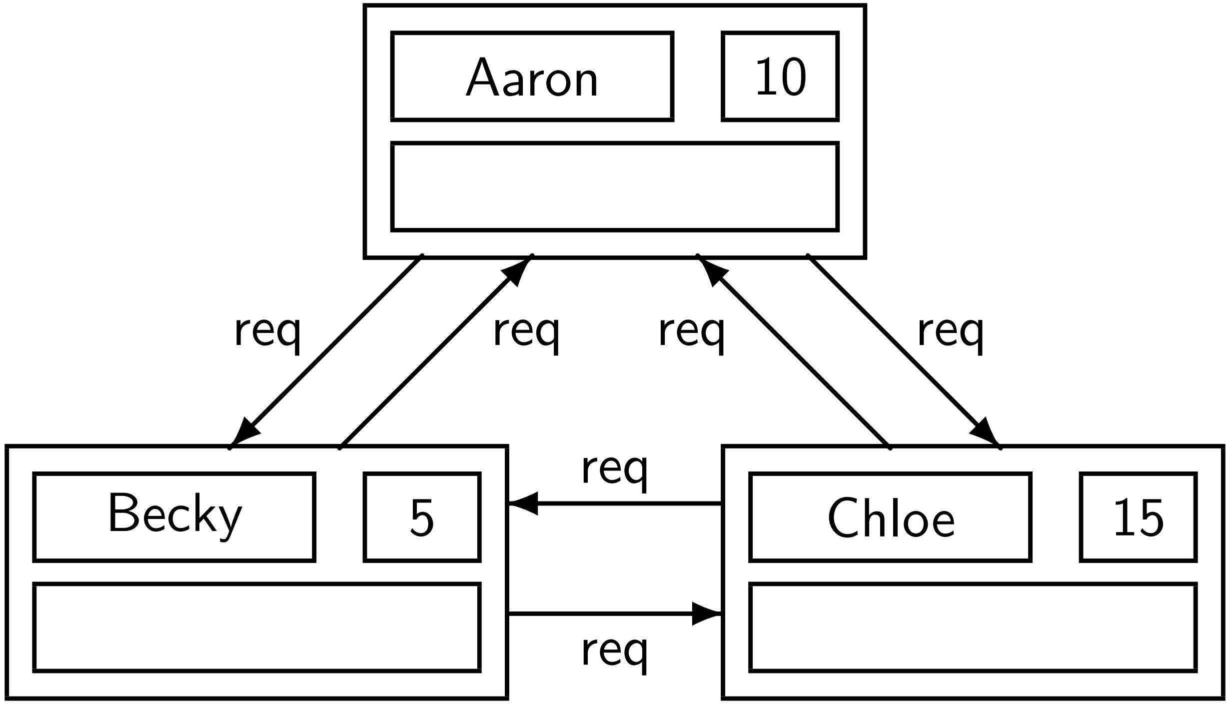
node 3

integer source
receive(request, source)

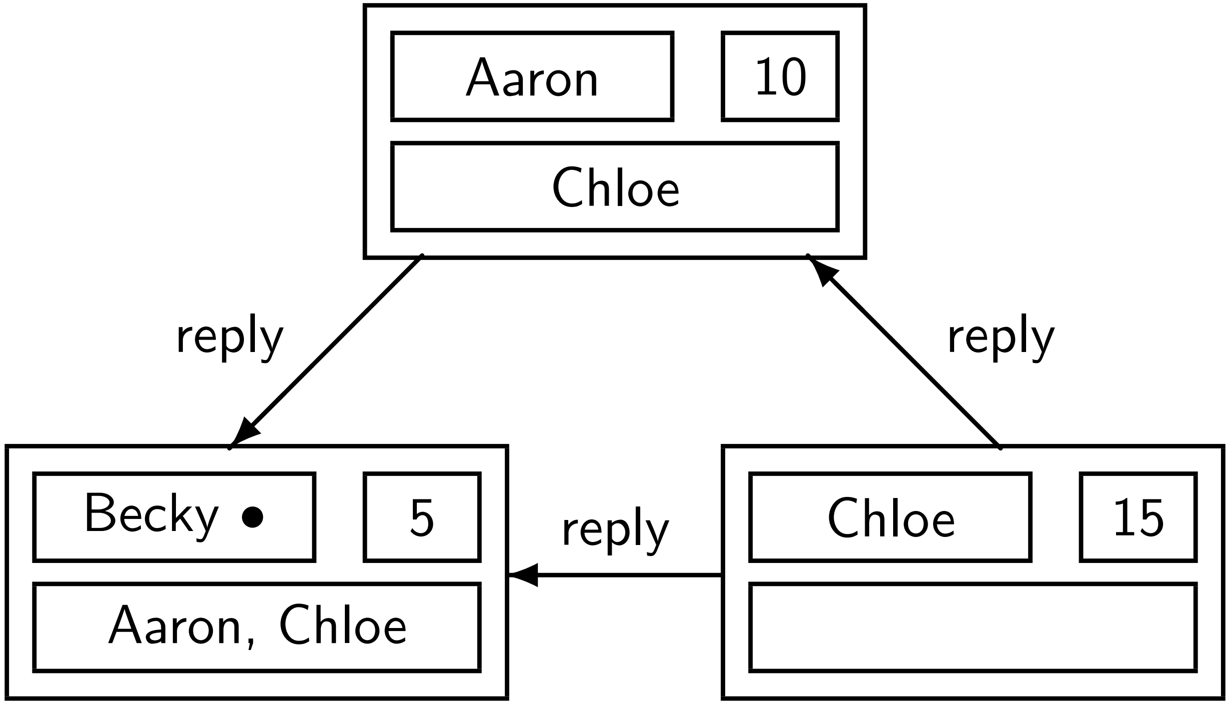
Algorithm 10.1: Ricart-Agrawala algorithm (outline)

```
                    integer myNum ← 0
                    set of node IDs deferred ← empty set

    main

        non-critical section
        myNum ← chooseNumber
        for all other nodes N
            send(request, N, myID, myNum)
        await reply's from all other nodes
        critical section
        for all nodes N in deferred
            remove N from deferred
            send(reply, N, myID)

    receive

    integer source, reqNum
        receive(request, source, reqNum)
        if reqNum < myNum
            send(reply,source,myID)
        else add source to deferred
```

RA Algorithm (1)

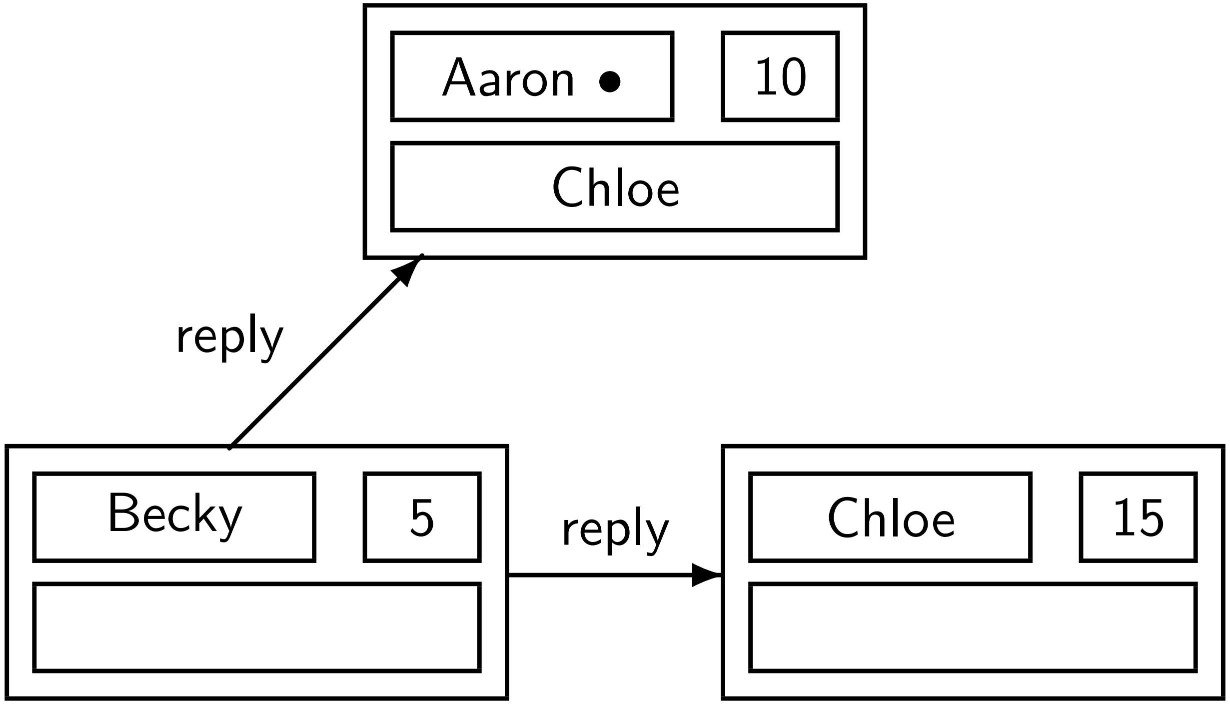


RA Algorithm (2)

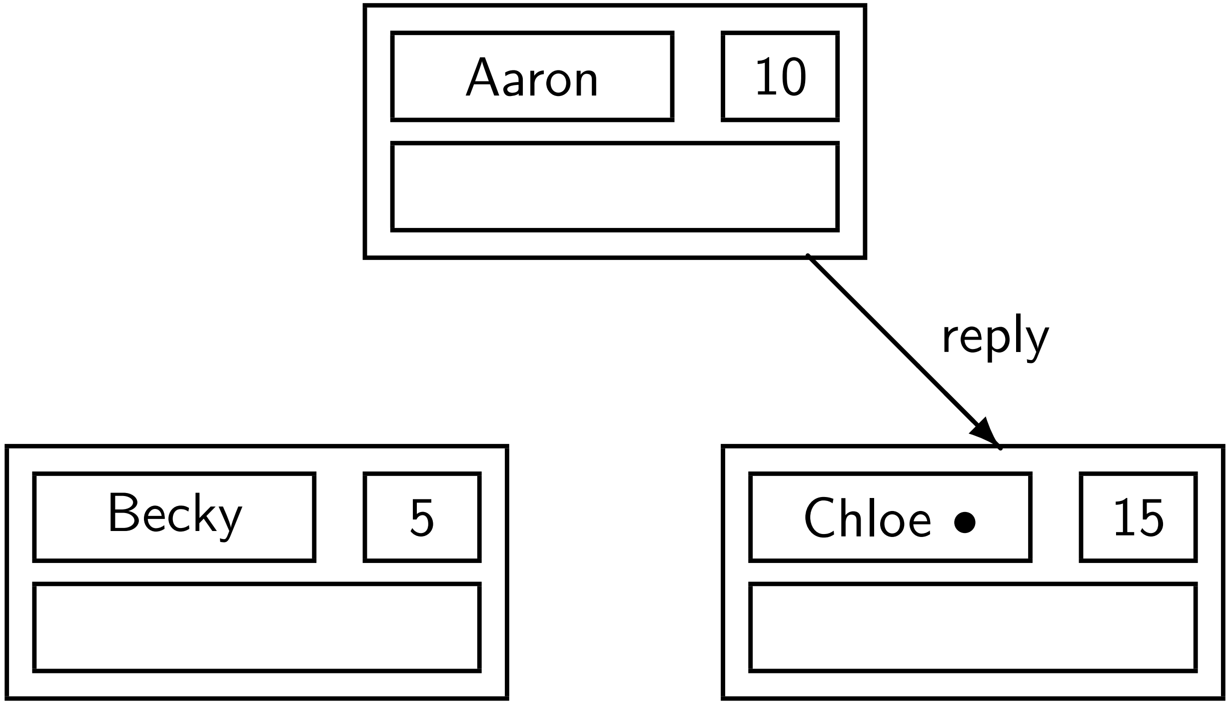


Virtual Queue in the RA Algorithm

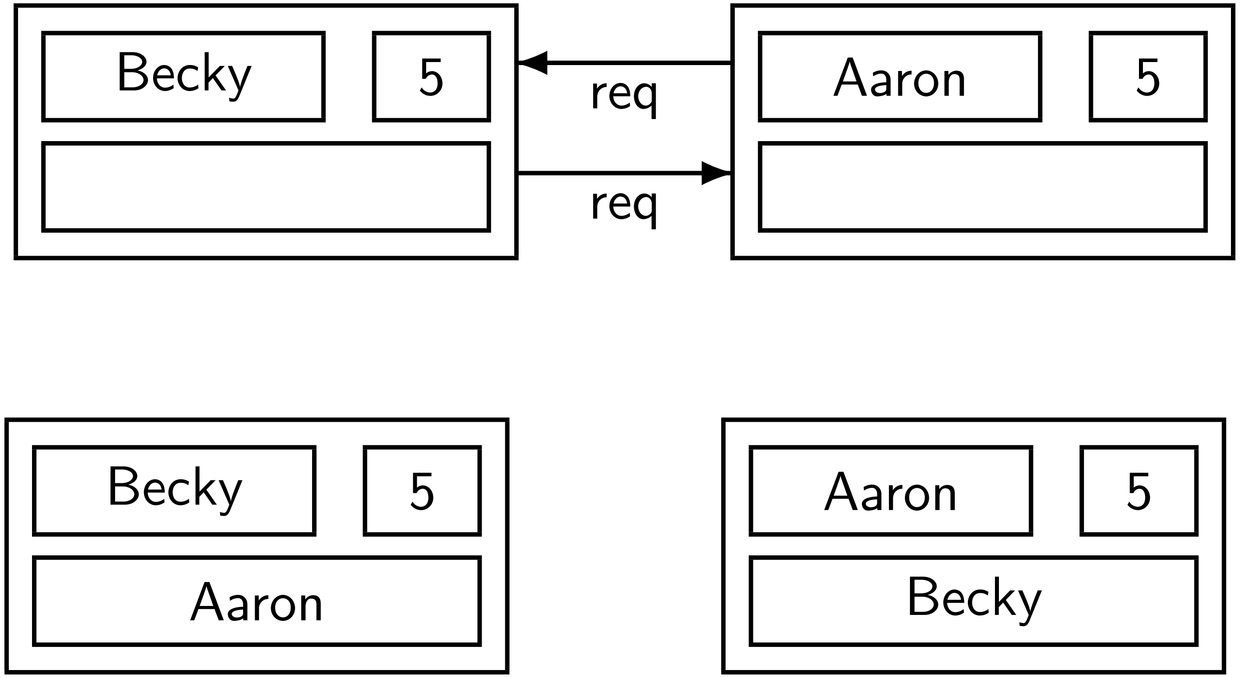
RA Algorithm (3)



RA Algorithm (4)

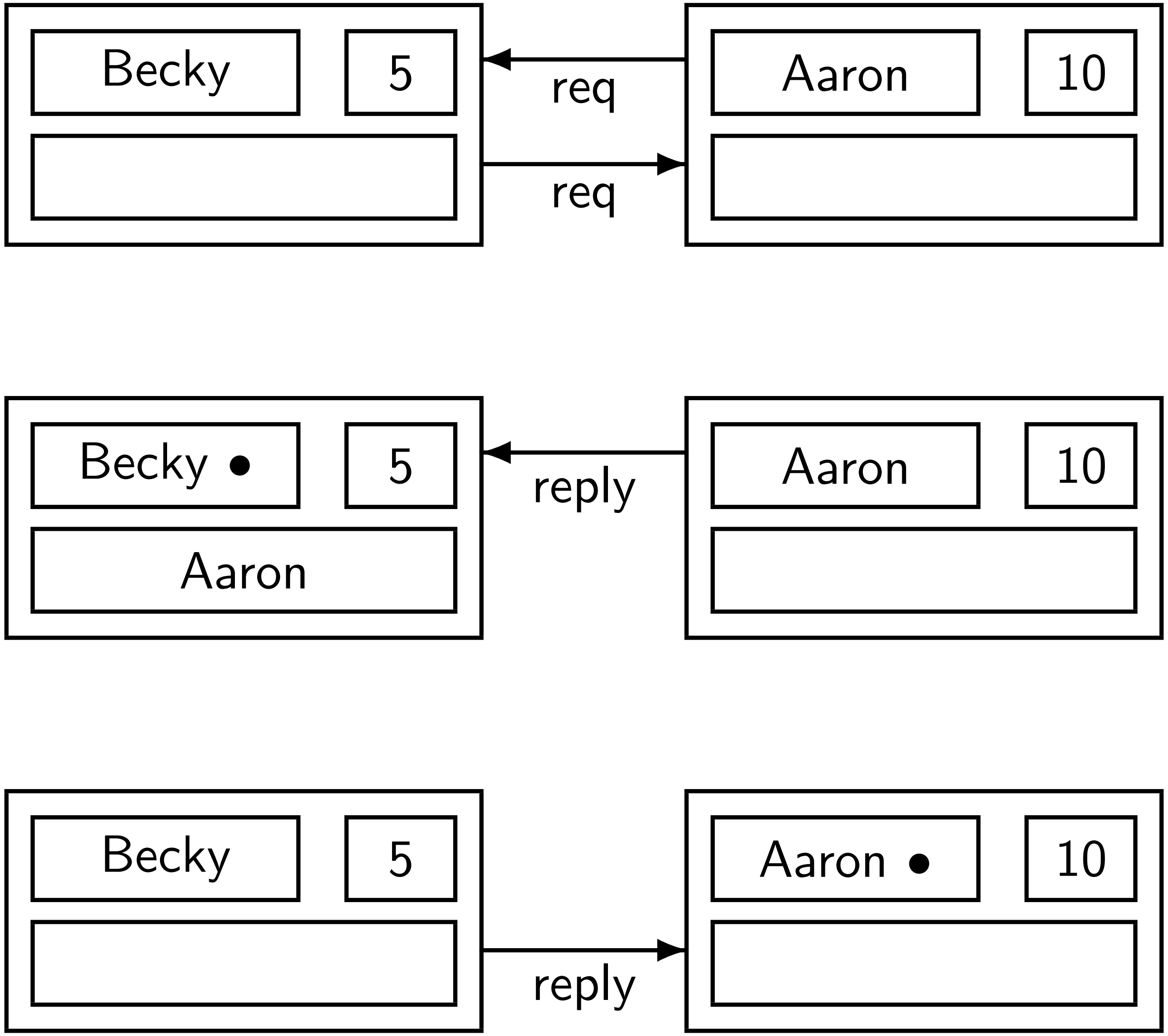


Equal Ticket Numbers



Note: This figure is not in the book.

Choosing Ticket Numbers (1)



Becky 5

req

req

Aaron 10

Becky • 5

Aaron

reply

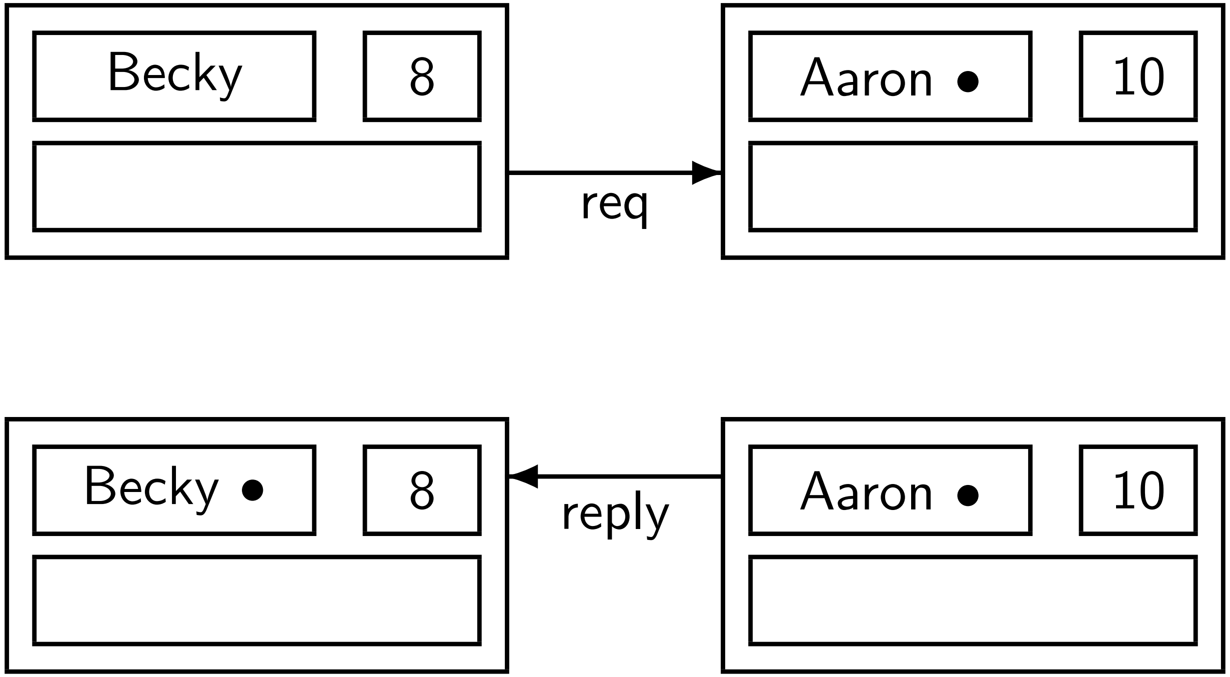
Aaron 10

Becky 5

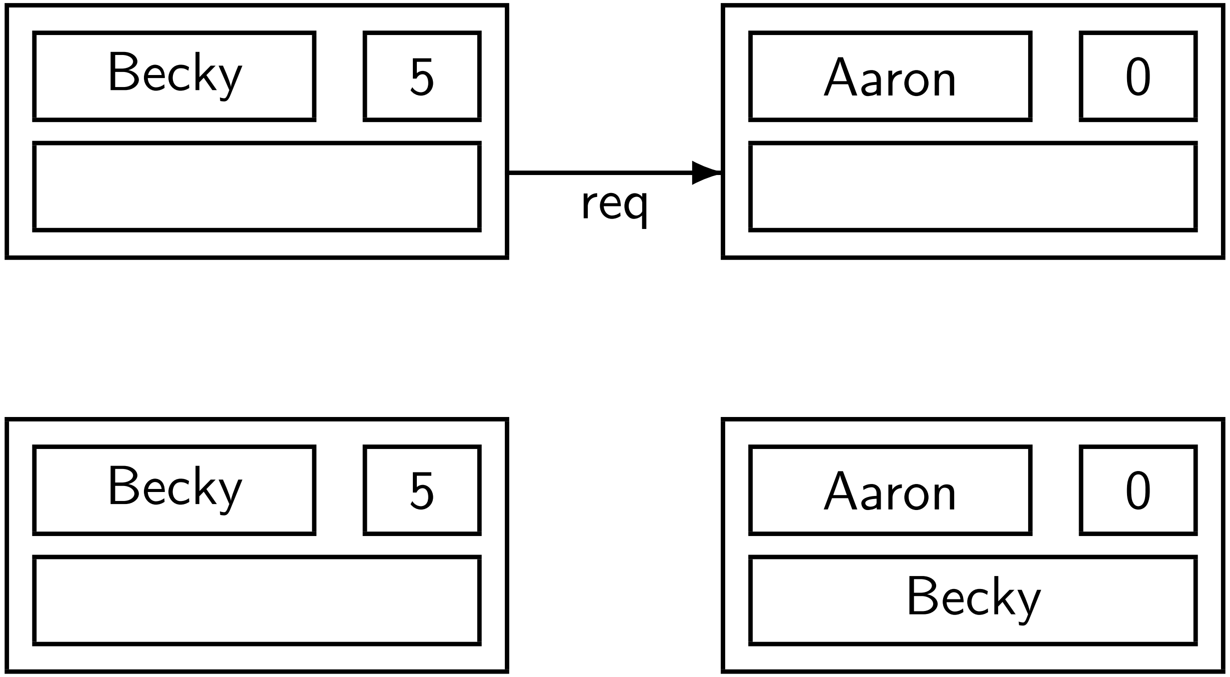
reply

Aaron • 10

Choosing Ticket Numbers (2)



Quiescent Nodes



Algorithm 10.2: Ricart-Agrawala algorithm

```
integer myNum ← 0
set of node IDs deferred ← empty set
integer highestNum ← 0
boolean requestCS ← false
```

Main

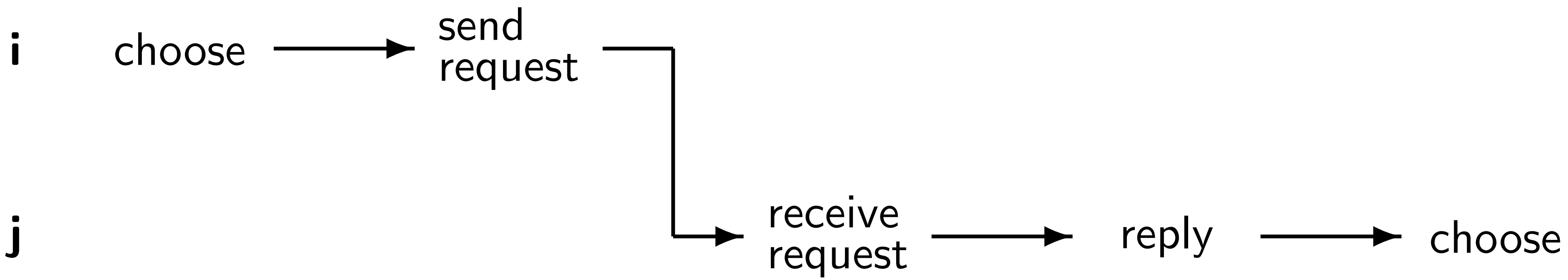
```
loop forever
   non-critical section
   requestCS ← true
   myNum ← highestNum + 1
   for all other nodes N
      send(request, N, myID, myNum)
   await reply's from all other nodes
   critical section
   requestCS ← false
   for all nodes N in deferred
      remove N from deferred
      send(reply, N, myID)
```

Algorithm 10.2: Ricart-Agrawala algorithm (continued)

Receive

```
integer source, requestedNum
loop forever
   receive(request, source, requestedNum)
   highestNum $\leftarrow$ max(highestNum, requestedNum)
   if not requestCS or requestedNum $\ll$ myNum
      send(reply, source, myID)
   else add source to deferred
```

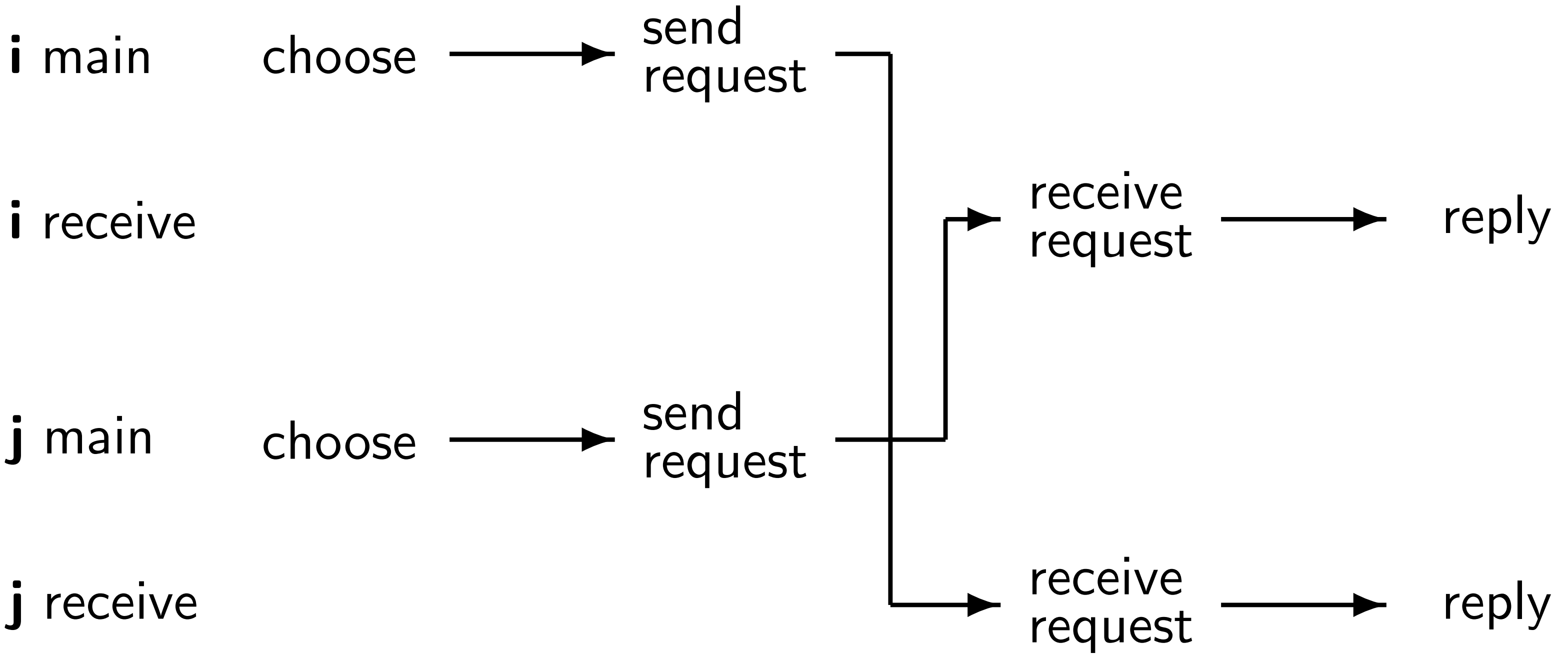
Correct of the RA Algorithm (Case 1)



i choose → send request

j receive request → reply → choose

Correct of the RA Algorithm (Case 2)



i main

i receive

j main

j receive

choose

send request

choose

send request

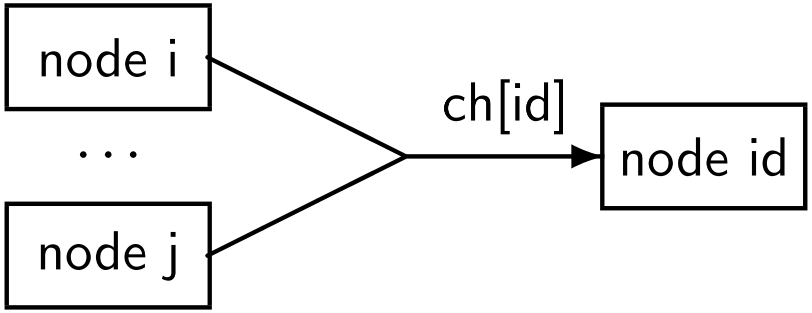
receive request

reply

receive request

reply

Channels in RA Algorithm in Promela



RA Algorithm in Promela – Main Process

```
proctype Main( byte myID ) {
    do ::
        atomic {
            requestCS[myID] = true ;
            myNum[myID] = highestNum[myID] + 1 ;
        }

        for  (J,0, NPROCS−1)
            if
            ::  J != myID −>
                ch[J]  !  request ,  myID, myNum[myID];
            ::  else
            fi
        rof  (J);

```

RA Algorithm in Promela – Main Process

```

        for (K,0,NPROCS-2)
            ch[myID] ?? reply, _ , _;
        rof (K);
        critical_section ();
        requestCS[myID] = false;

        byte N;
        do
            :: empty(deferred[myID]) -> break;
            :: deferred [myID] ? N -> ch[N] ! reply, 0, 0
        od
    od
}
```

RA Algorithm in Promela – Receive Process

```
proctype Receive( byte myID ) {
    byte reqNum, source;
    do ::
        ch[myID] ?? request, source, reqNum;
        highestNum[myID] =
            ((reqNum > highestNum[myID]) ->
                reqNum : highestNum[myID]);
        atomic {
            if
            :: requestCS[myID] &&
                ( (myNum[myID] < reqNum) ||
                ( (myNum[myID] == reqNum) &&
                        (myID < source)
                ) ) ->
                    deferred [ myID] ! source
```

RA Algorithm in Promela – Receive Process

```
            :: else ->
                 ch[source] ! reply, 0, 0
            fi
        }
    od
}
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm

```
        boolean haveToken ← true in node 0, false in others
        integer array[NODES] requested ← [0,. . . ,0]
        integer array[NODES] granted ← [0,. . . ,0]
        integer myNum ← 0
        boolean inCS ← false
```

```
sendToken
    if exists N such that requested[N] > granted[N]
        for some such N
            send(token, N, granted)
            haveToken ← false
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm (continued)

Main

```
loop forever
   non-critical section
   if not haveToken
      myNum ← myNum + 1
      for all other nodes N
         send(request, N, myID, myNum)
      receive(token, granted)
      haveToken ← true
   inCS ← true
   critical section
   granted[myID] ← myNum
   inCS ← false
   sendToken
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm (continued)

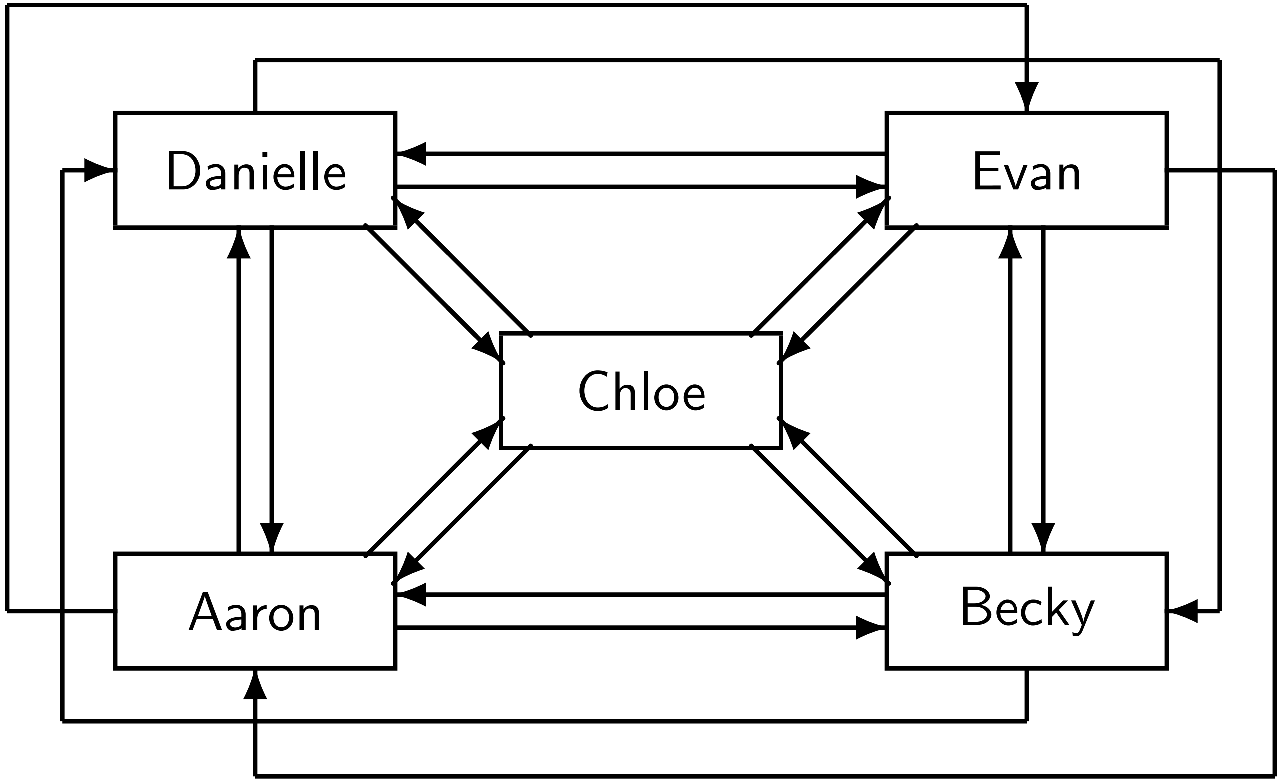
Receive

```
integer source, reqNum
loop forever
   receive(request, source, reqNum)
   requested[source] $\leftarrow$ max(requested[source], reqNum)
   if haveToken and not inCS
      sendToken
```

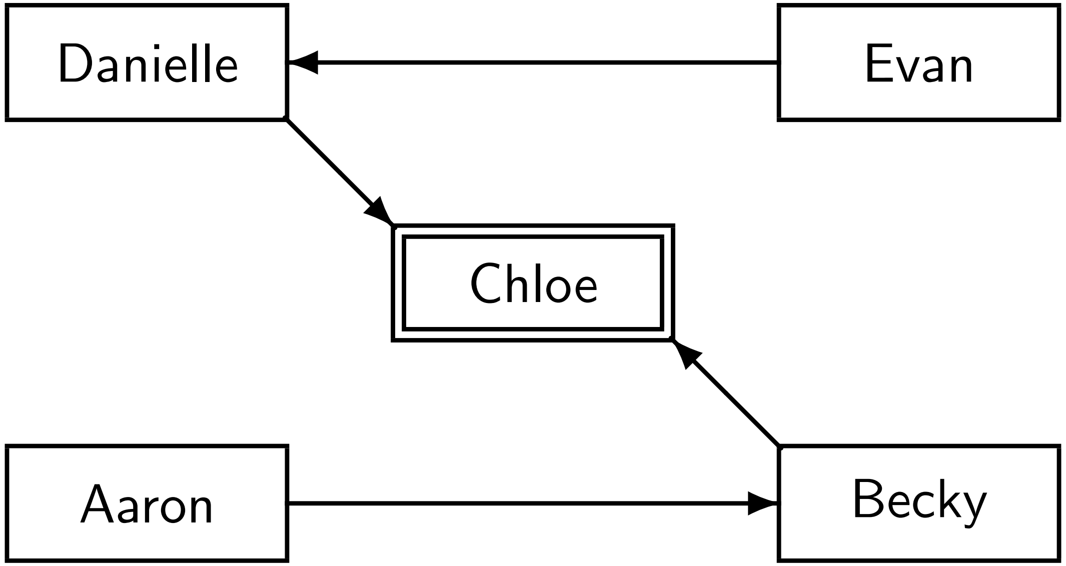
Data Structures for RA Token-Passing Algorithm

	Aaron	Becky	Chloe	Danielle	Evan
requested	4	3	0	5	1
granted	4	2	2	4	1

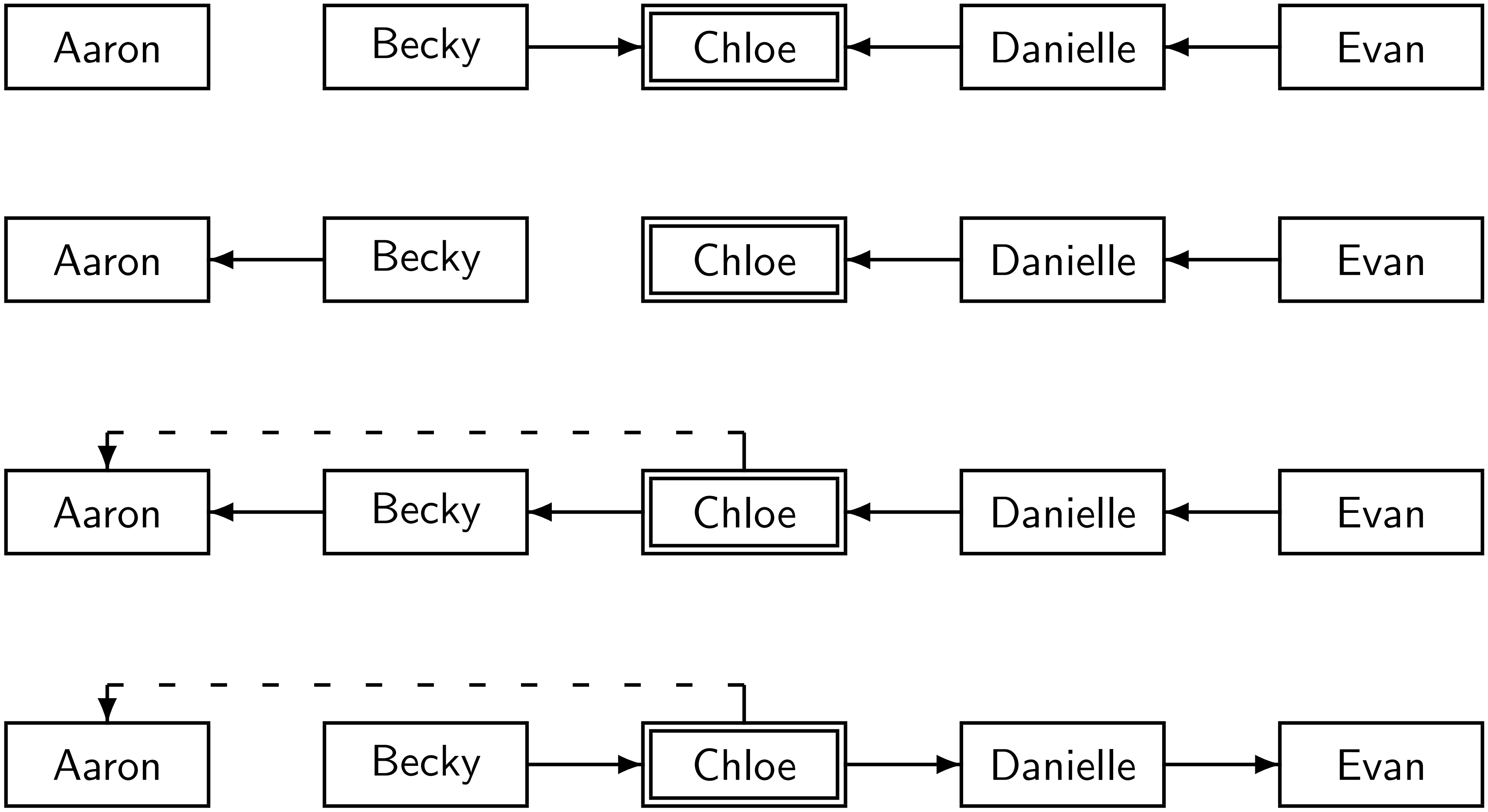
Distributed System for Neilsen-Mizuno Algorithm



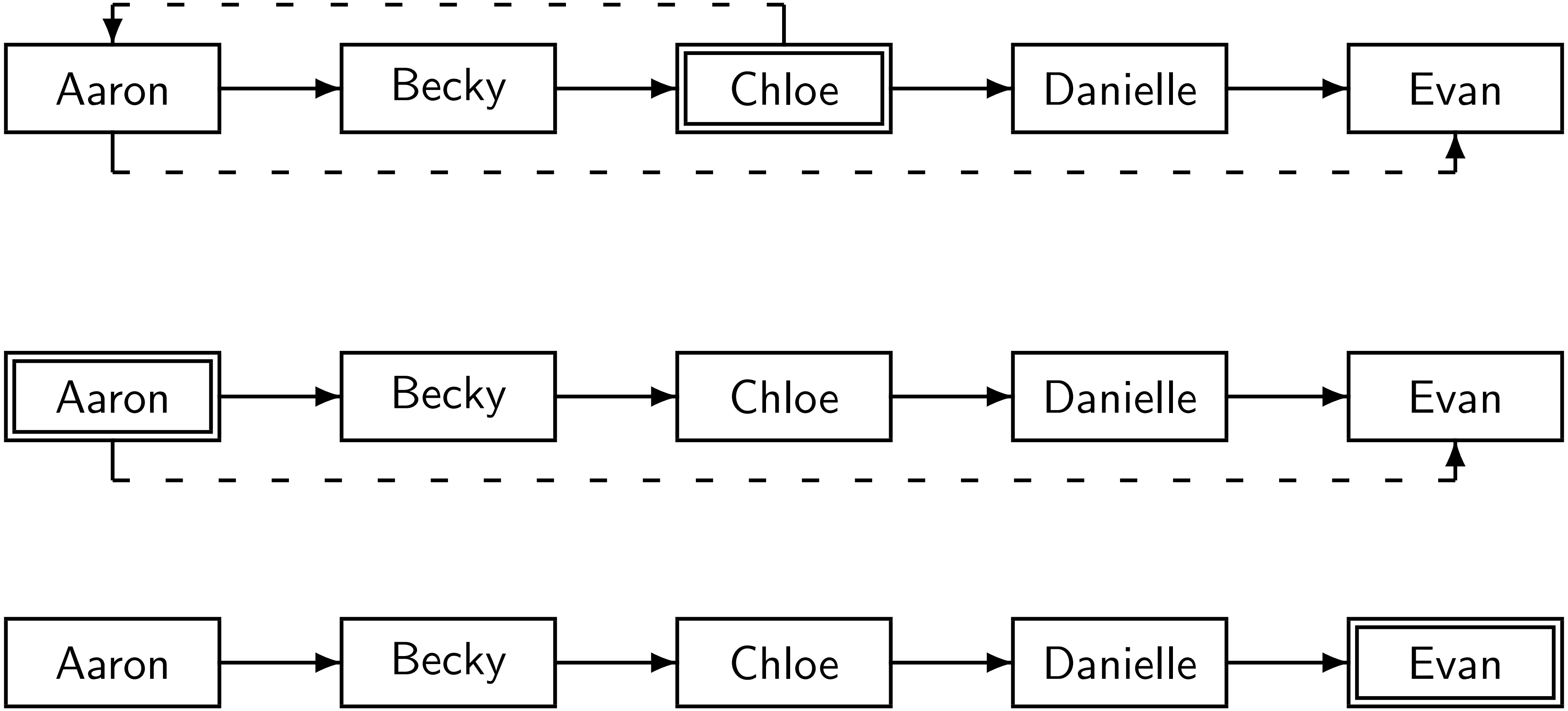
Spanning Tree in Neilsen-Mizuno Algorithm



Neilsen-Mizuno Algorithm (1)



Neilsen-Mizuno Algorithm (2)



Algorithm 10.4: Neilsen-Mizuno token-passing algorithm

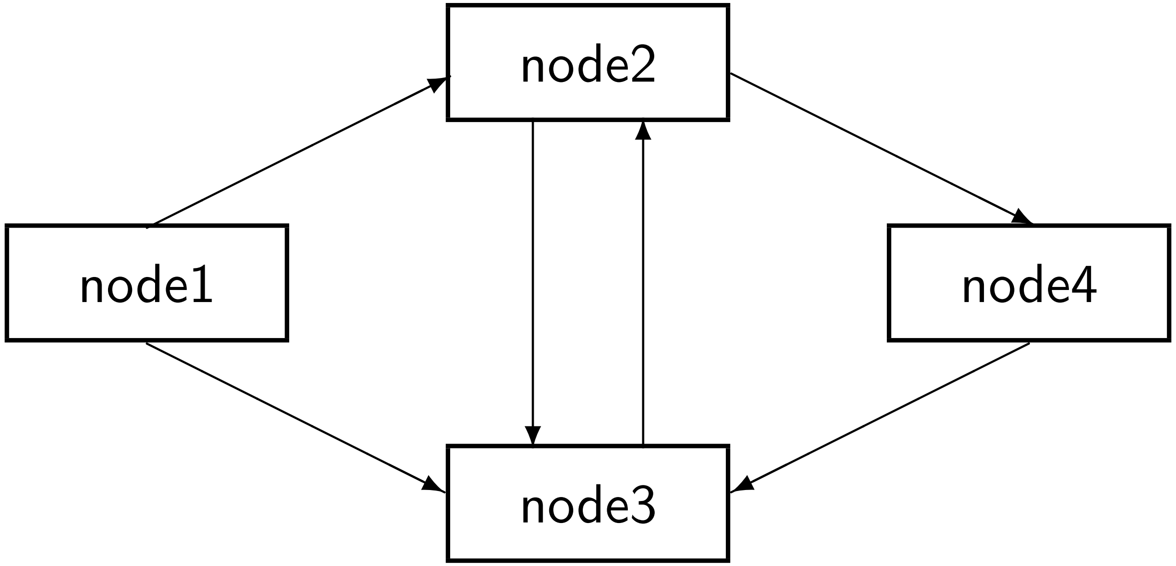
```
                 integer parent ← (initialized to form a tree)
                 integer deferred ← 0
                 boolean holding ← true in the root, false in others
       Main
       loop forever
p1:       non-critical section
p2:       if not holding
p3:          send(request, parent, myID, myID)
p4:          parent ← 0
p5:          receive(token)
p6:       holding ← false
p7:       critical section
p8:       if deferred ≠ 0
p9:          send(token, deferred)
p10:         deferred ← 0
p11:      else holding ← true
```

Algorithm 10.4: Neilsen-Mizuno token-passing algorithm (continued)

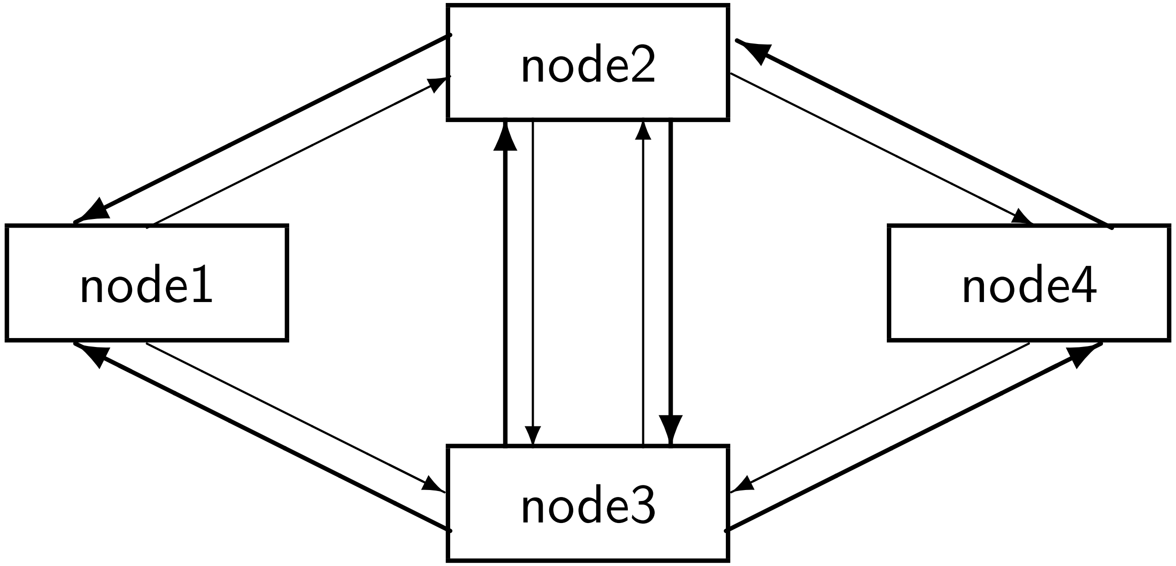
Receive

```
integer source, originator
loop forever
    receive(request, source, originator)
    if parent = 0
        if holding
            send(token, originator)
            holding ← false
        else deferred ← originator
    else send(request, parent, myID, originator)
    parent ← source
```

Distributed System with an Environment Node



Back Edges



Algorithm 11.1: Dijkstra-Scholten algorithm (preliminary)

```
                    integer array[incoming] inDeficit ← [0,. . . ,0]
                    integer inDeficit ← 0, integer outDeficit ← 0

      send message
p1:  send(message, destination, myID)
p2:  increment outDeficit

      receive message
p3:  receive(message, source)
p4:  increment inDeficit[source] and inDeficit

      send signal
p5:  when inDeficit > 1 or
            (inDeficit = 1 and isTerminated and outDeficit = 0)
p6:     E ← some edge E with inDeficit[E] ≠ 0
p7:     send(signal, E, myID)
p8:     decrement inDeficit[E] and inDeficit

      receive signal
p9:  receive(signal, _)
p10: decrement outDeficit
```

Algorithm 11.2: Dijkstra-Scholten algorithm (env., preliminary)

```
integer outDeficit ← 0
```

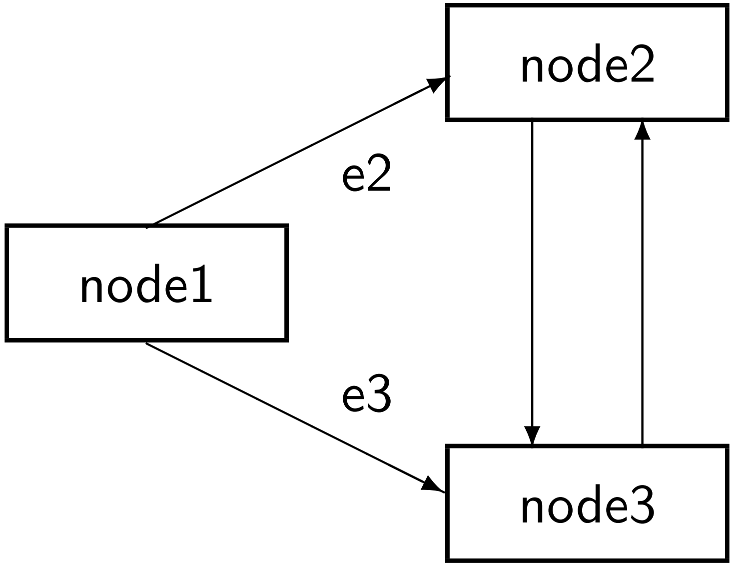
computation

```
for all outgoing edges E
    send(message, E, myID)
    increment outDeficit
await outDeficit = 0
announce system termination
```

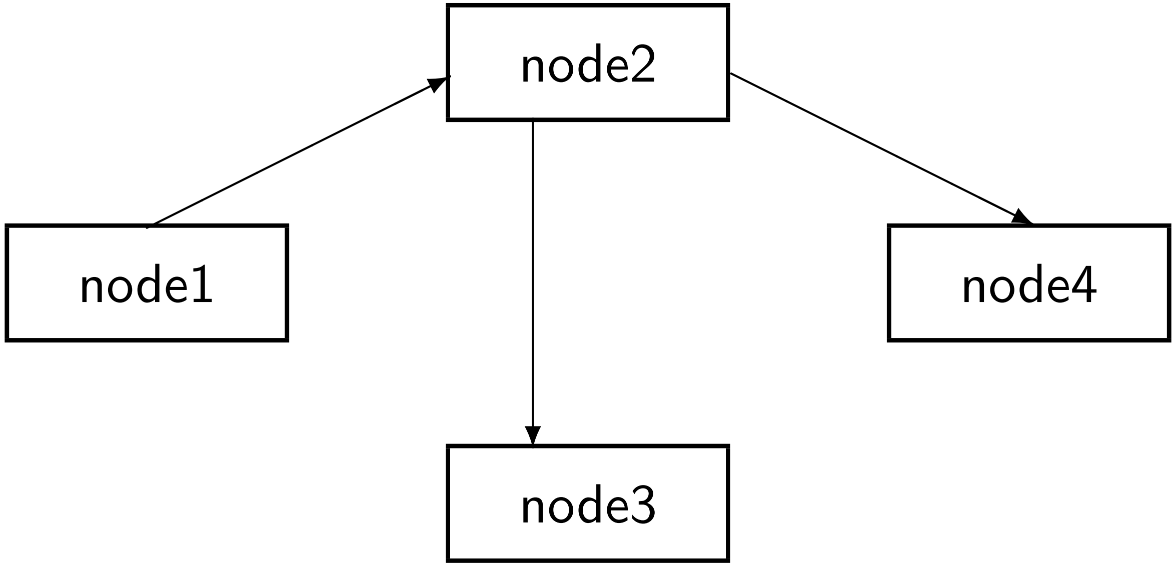
receive signal

```
receive(signal, source)
decrement outDeficit
```

The Preliminary DS Algorithm is Unsafe



Spanning Tree



node2

node1

node4

node3

Algorithm 11.3: Dijkstra-Scholten algorithm

```
integer array[incoming] inDeficit ← [0,. . . ,0]
integer inDeficit ← 0
integer outDeficit ← 0
integer parent ← −1
```

send message

```
p1: when parent ≠ −1            // Only active nodes send messages
p2:     send(message, destination, myID)
p3:     increment outDeficit
```

receive message

```
p4: receive(message,source)
p5: if parent = −1
p6:     parent ← source
p7: increment inDeficit[source] and inDeficit
```

Algorithm 11.3: Dijkstra-Scholten algorithm (continued)

send signal

```
p8:  when $inDeficit > 1$
p9:      E ← some edge E for which
             $(inDeficit[E] > 1)$ or $(inDeficit[E] = 1$ and $E \neq parent)$
p10:     send(signal, E, myID)
p11:     decrement inDeficit[E] and inDeficit
p12: or when $inDeficit = 1$ and isTerminated and $outDeficit = 0$
p13:     send(signal, parent, myID)
p14:     inDeficit[parent] ← 0
p15:     inDeficit ← 0
p16:     parent ← −1
```

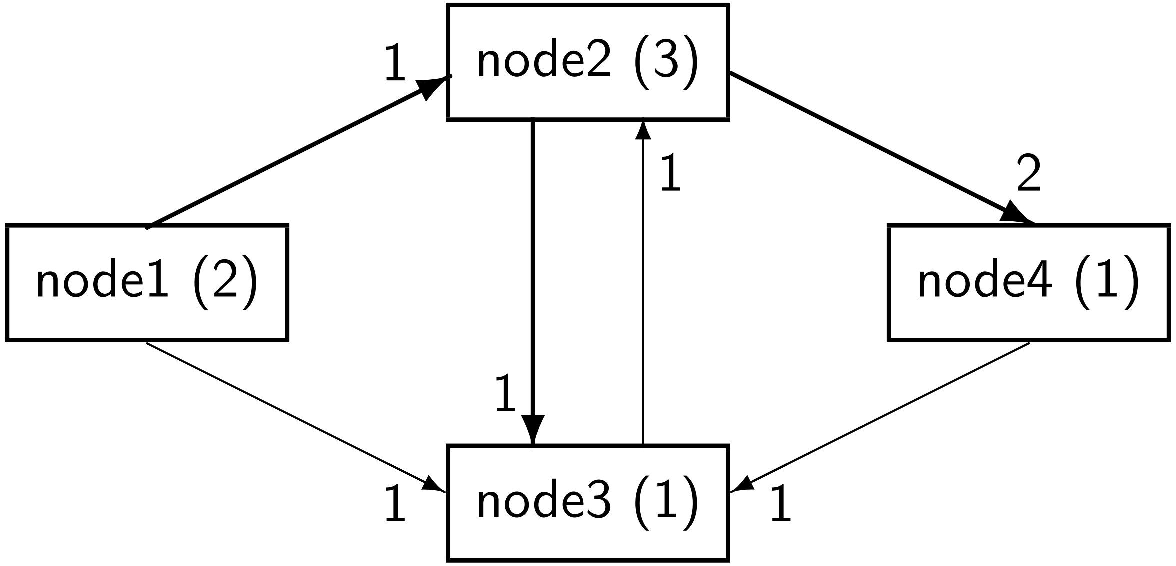
receive signal

```
p17: receive(signal, _)
p18: decrement outDeficit
```

Partial Scenario for DS Algorithm

Action	node1	node2	node3	node4
1 ⇒ 2	(-1,[ ],0)	(-1,[0,0],0)	(-1,[0,0,0],0)	(-1,[0],0)
2 ⇒ 4	(-1,[ ],1)	(1,[1,0],0)	(-1,[0,0,0],0)	(-1,[0],0)
2 ⇒ 3	(-1,[ ],1)	(1,[1,0],1)	(-1,[0,0,0],0)	(2,[1],0)
2 ⇒ 4	(-1,[ ],1)	(1,[1,0],2)	(2,[0,1,0],0)	(2,[1],0)
1 ⇒ 3	(-1,[ ],1)	(1,[1,0],3)	(2,[0,1,0],0)	(2,[2],0)
3 ⇒ 2	(-1,[ ],2)	(1,[1,0],3)	(2,[1,1,0],0)	(2,[2],0)
4 ⇒ 3	(-1,[ ],2)	(1,[1,1],3)	(2,[1,1,0],1)	(2,[2],0)
	(-1,[ ],2)	(1,[1,1],3)	(2,[1,1,1],1)	(2,[2],1)

Data Structures After Completion of Partial Scenario



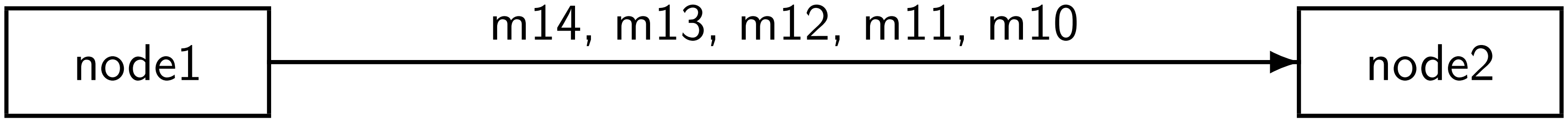
Algorithm 11.4: Credit-recovery algorithm (environment node)

```
float weight ← 1.0

computation
p1: for all outgoing edges E
p2:     weight ← weight / 2.0
p3:     send(message, E, weight)
p4: await weight = 1.0
p5: announce system termination

receive signal
p6: receive(signal, w)
p7: weight ← weight + w
```

Algorithm 11.5: Credit-recovery algorithm (non-environment node)

```
           constant integer parent ← 0 // Environment node
           boolean active ← false
           float weight ← 0.0

     send message

p1:  if active          // Only active nodes send messages
p2:     weight ← weight / 2.0
p3:     send(message, destination, myID, weight)

     receive message

p4:  receive(message, source, w)
p5:  active ← true
p6:  weight ← weight + w

     send signal

p7:  when terminated
p8:     send(signal, parent, weight)
p9:     weight ← 0.0
p10:    active ← false
```

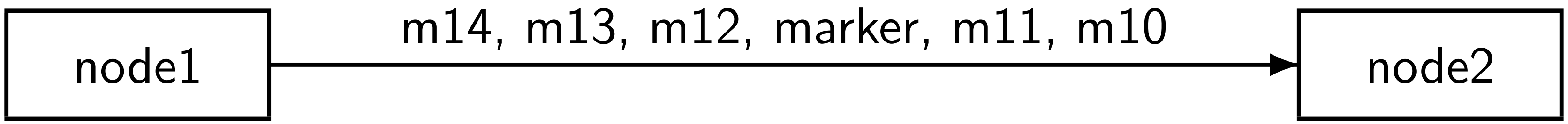
Messages on a Channel


node1

m14, m13, m12, m11, m10

node2

Sending a Marker



Algorithm 11.6: Chandy-Lamport algorithm for global snapshots

```
integer array[outgoing] lastSent ← [0, . . . , 0]
integer array[incoming] lastReceived ← [0, . . . , 0]
integer array[outgoing] stateAtRecord ← [−1, . . . , −1]
integer array[incoming] messageAtRecord ← [−1, . . . , −1]
integer array[incoming] messageAtMarker ← [−1, . . . , −1]
```

send message

```
p1: send(message, destination, myID)
p2: lastSent[destination] ← message
```

receive message

```
p3: receive(message,source)
p4: lastReceived[source] ← message
```

Algorithm 11.6: Chandy-Lamport algorithm for global snapshots (continued)

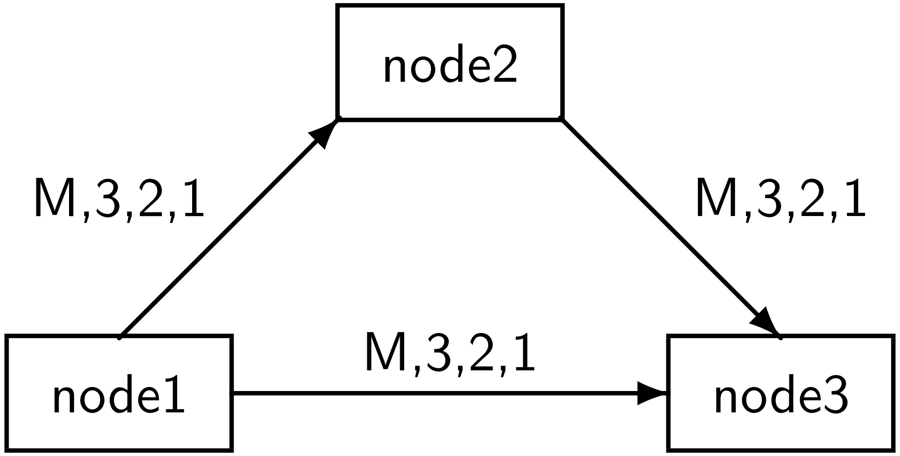
receive marker

```
p6:  receive(marker, source)
p7:  messageAtMarker[source] ← lastReceived[source]
p8:  if stateAtRecord = [$-1$,...,$-1$]      // Not yet recorded
p9:      stateAtRecord ← lastSent
p10:     messageAtRecord ← lastReceived
p11:     for all outgoing edges E
p12:         send(marker, E, myID)
```

record state

```
p13: await markers received on all incoming edges
p14: recordState
```

Messages and Markers for a Scenario



node2

M,3,2,1

M,3,2,1

M,3,2,1

node1

node3

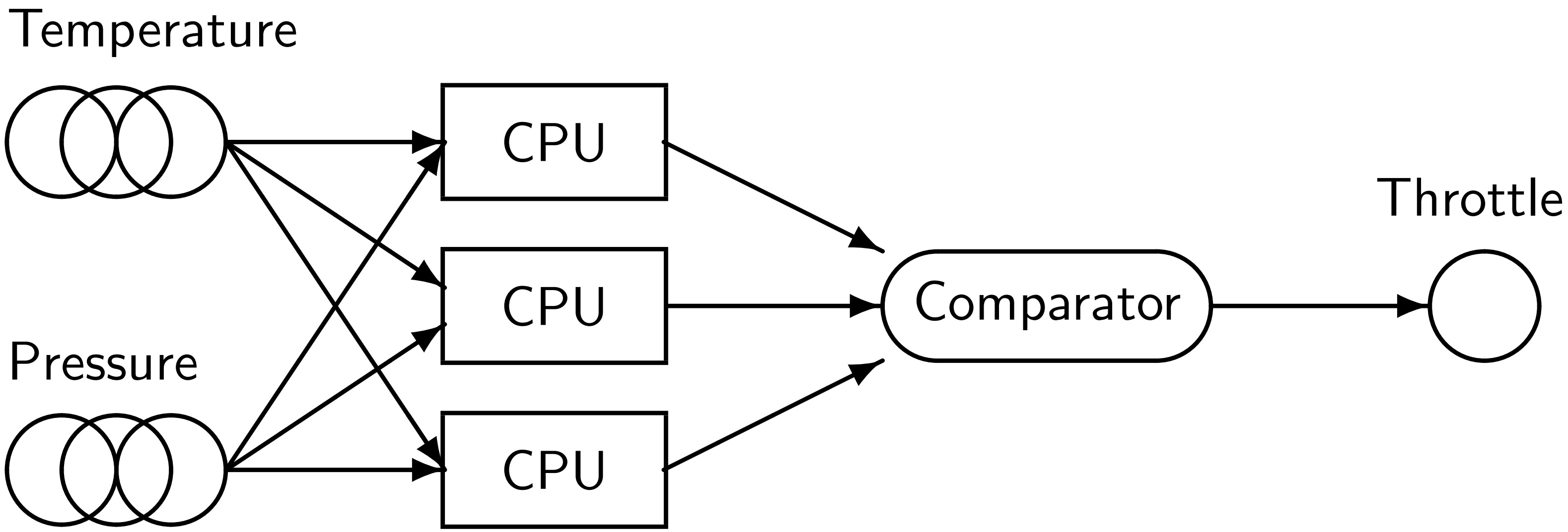
Scenario for CL Algorithm (1)

Action	node1					node2				
	ls	lr	st	rc	mk	ls	lr	st	rc	mk
	[3,3]					[3]	[3]			
1M⇒2	[3,3]		[3,3]			[3]	[3]			
1M⇒3	[3,3]		[3,3]			[3]	[3]			
2⇐1M	[3,3]		[3,3]			[3]	[3]			
2M⇒3	[3,3]		[3,3]			[3]	[3]	[3]	[3]	[3]

Scenario for CL Algorithm (2)

Action	node3				
	ls	lr	st	rc	mk
3⇐2					
3⇐2		[0,1]			
3⇐2		[0,2]			
3⇐2M		[0,3]			
3⇐1		[0,3]		[0,3]	[0,3]
3⇐1		[1,3]		[0,3]	[0,3]
3⇐1		[2,3]		[0,3]	[0,3]
3⇐1M		[3,3]		[0,3]	[0,3]
		[3,3]		[0,3]	[3,3]

Architecture for a Reliable System



Temperature

CPU

CPU

CPU

Pressure

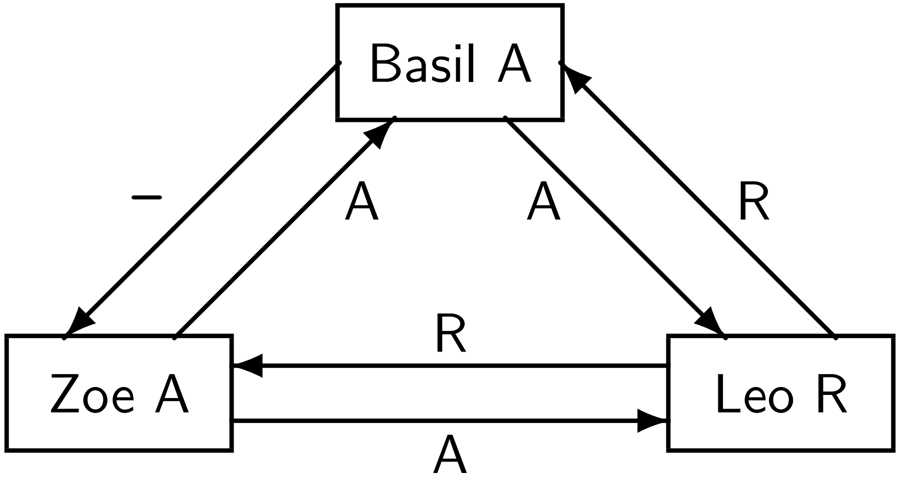
Comparator

Throttle

Algorithm 12.1: Consensus - one-round algorithm

```
planType finalPlan
planType array[generals] plan

plan[myID] ← chooseAttackOrRetreat
for all other generals G
    send(G, myID, plan[myID])
for all other generals G
    receive(G, plan[G])
finalPlan ← majority(plan)
```

Messages Sent in a One-Round Algorithm



Data Structures in a One-Round Algorithm

Leo	
general	plan
Basil	A
Leo	R
Zoe	A
majority	A

Zoe	
general	plans
Basil	–
Leo	R
Zoe	A
majority	R

Algorithm 12.2: Consensus - Byzantine Generals algorithm

```
planType finalPlan
planType array[generals] plan, majorityPlan
planType array[generals, generals] reportedPlan

plan[myID] ← chooseAttackOrRetreat
for all other generals G                          // First round
    send(G, myID, plan[myID])
for all other generals G
    receive(G, plan[G])
for all other generals G                          // Second round
    for all other  generals G' except G
        send(G', myID, G, plan[G])
for all other generals G
    for all other generals G' except G
        receive(G, G', reportedPlan[G, G'])
for all other generals G                          // First vote
    majorityPlan[G] ← majority(plan[G] ∪ reportedPlan[*, G])
majorityPlan[myID] ← plan[myID]                   // Second vote
finalPlan ← majority(majorityPlan)
```

Data Structure for Crash Failure - First Scenario (Leo)

Leo				
general	plan	reported by		majority
		Basil	Zoe	
Basil	A		–	A
Leo	R			R
Zoe	A	–		A
majority				A

Data Structure for Crash Failure - First Scenario (Zoe)

Zoe				
general	plan	reported by		majority
		Basil	Leo	
Basil	–		**A**	A
Leo	R	–		R
Zoe	A			A
majority				A

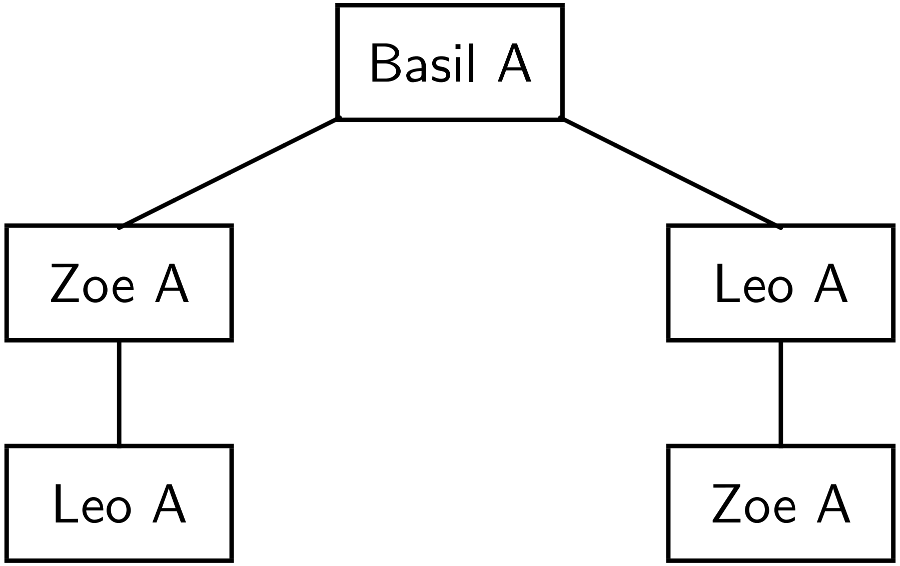
Data Structure for Crash Failure - Second Scenario (Leo)

Leo				
general	plan	reported by		majority
		Basil	Zoe	
Basil	A		A	A
Leo	R			R
Zoe	A	A		A
majority				A

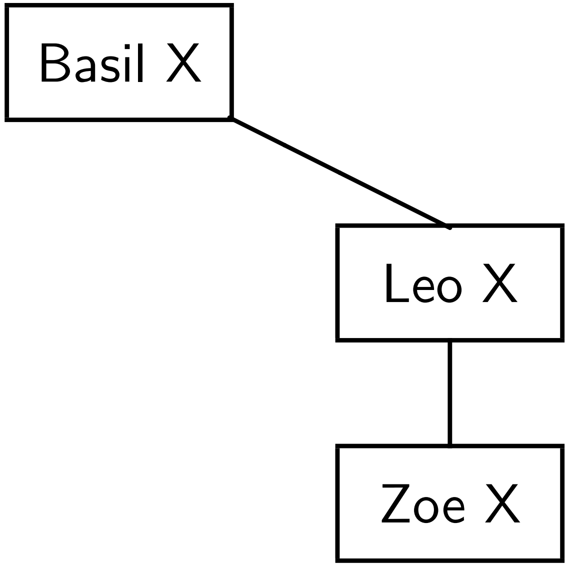
Data Structure for Crash Failure - Second Scenario (Zoe)

Zoe				
general	plan	reported by		majority
		Basil	Leo	
Basil	A		A	A
Leo	R	–		R
Zoe	A			A
majority				A

Knowledge Tree about Basil for Crash Failure - First Scenario



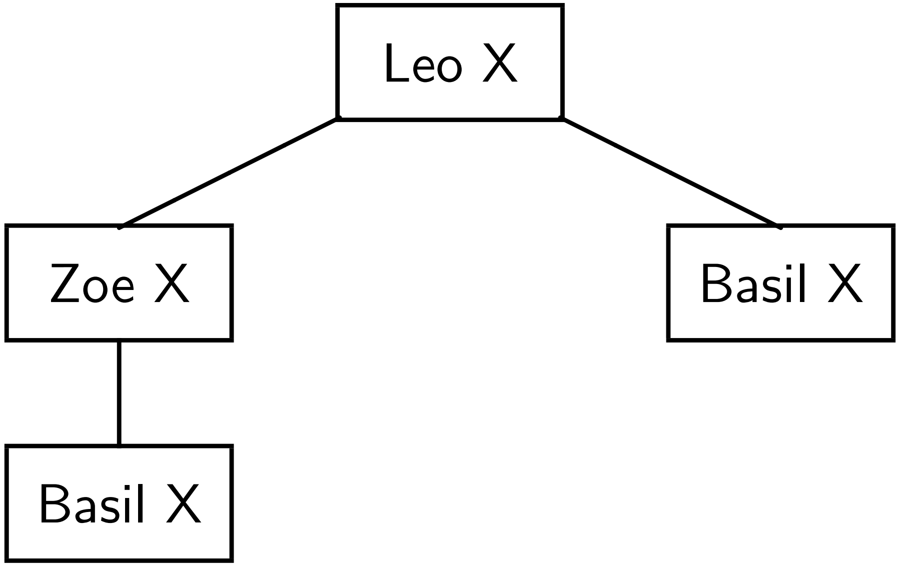
Knowledge Tree about Basil for Crash Failure - Second Scenario


Basil X

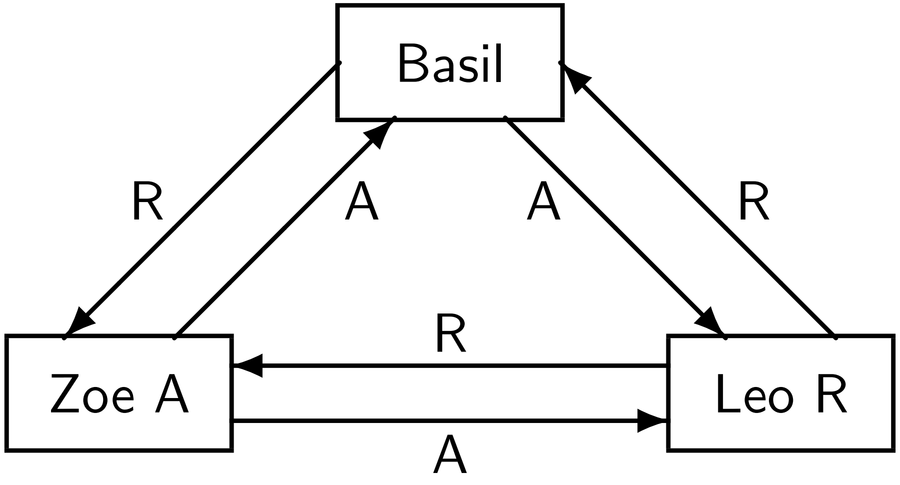
Leo X

Zoe X

Knowledge Tree about Leo for Crash Failure



Messages Sent for Byzantine Failure with Three Generals



Data Stuctures for Leo and Zoe After First Round

Leo	
general	plans
Basil	A
Leo	R
Zoe	A
majority	A

Zoe	
general	plans
Basil	R
Leo	R
Zoe	A
majority	R

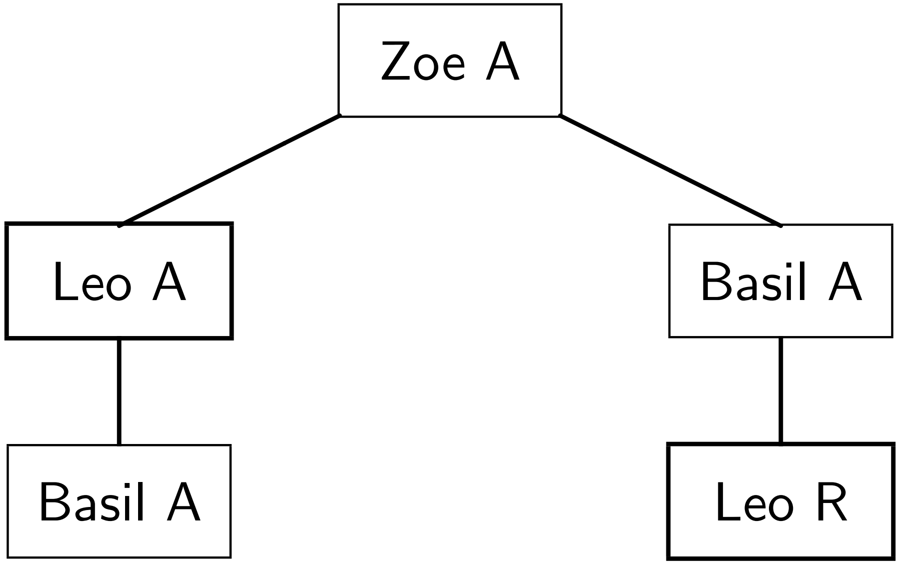
Data Stuctures for Leo After Second Round

Leo				
general	plans	reported by		majority
		Basil	Zoe	
Basil	A		A	A
Leo	R			R
Zoe	A	**R**		R
majority				R

Data Stuctures for Zoe After Second Round

Zoe				
general	plans	reported by		majority
		Basil	Leo	
Basil	A		A	A
Leo	R	R		R
Zoe	A			A
majority				A

Knowledge Tree About Zoe



Zoe A

Leo A

Basil A

Basil A

Leo R

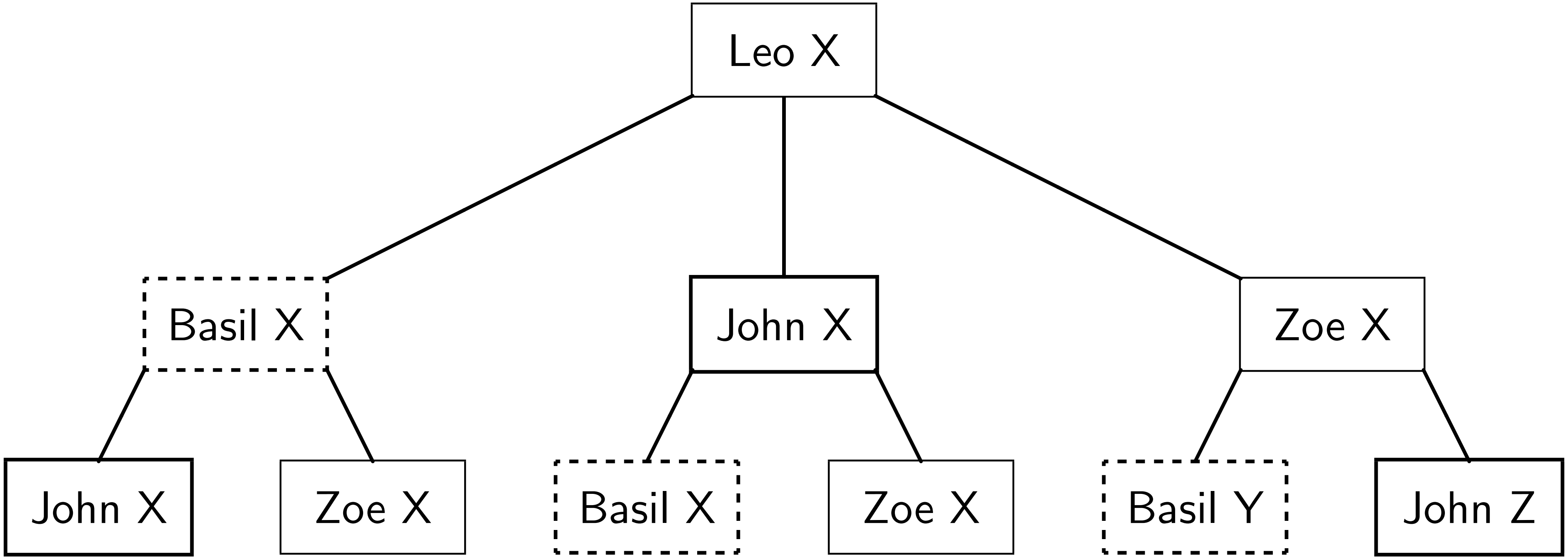
Four Generals: Data Structure of Basil (1)

Basil					
general	plan	reported by			majority
		John	Leo	Zoe	
Basil	A				A
John	**A**		**A**	?	A
Leo	R	R		?	R
Zoe	?	?	?		?
majority					?

Four Generals: Data Structure of Basil (2)

Basil					
general	plans	reported by			majority
		John	Leo	Zoe	
Basil	A				A
John	A		A	?	A
Leo	R	R		?	R
Zoe	**R**	**A**	**R**		R
					R

Knowledge Tree About Loyal General Leo


Leo X

Basil X

John X

Zoe X

John X

Zoe X

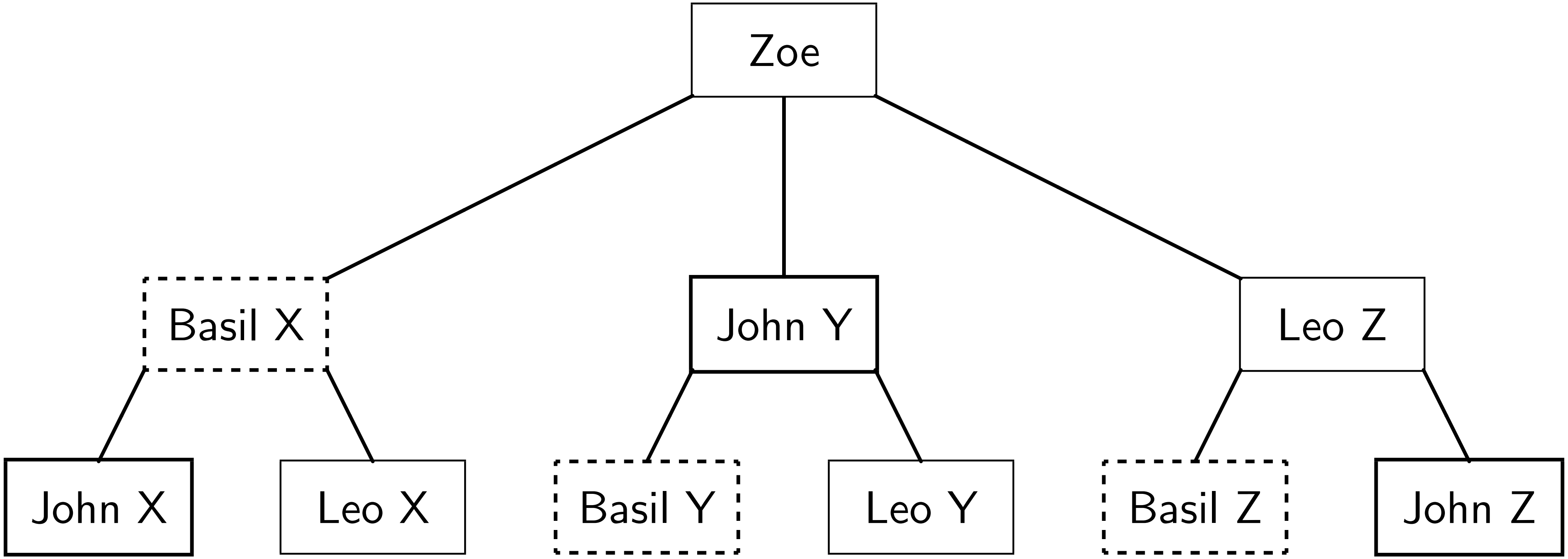
Basil X

Zoe X

Basil Y

John Z

Knowledge Tree About Traitor Zoe



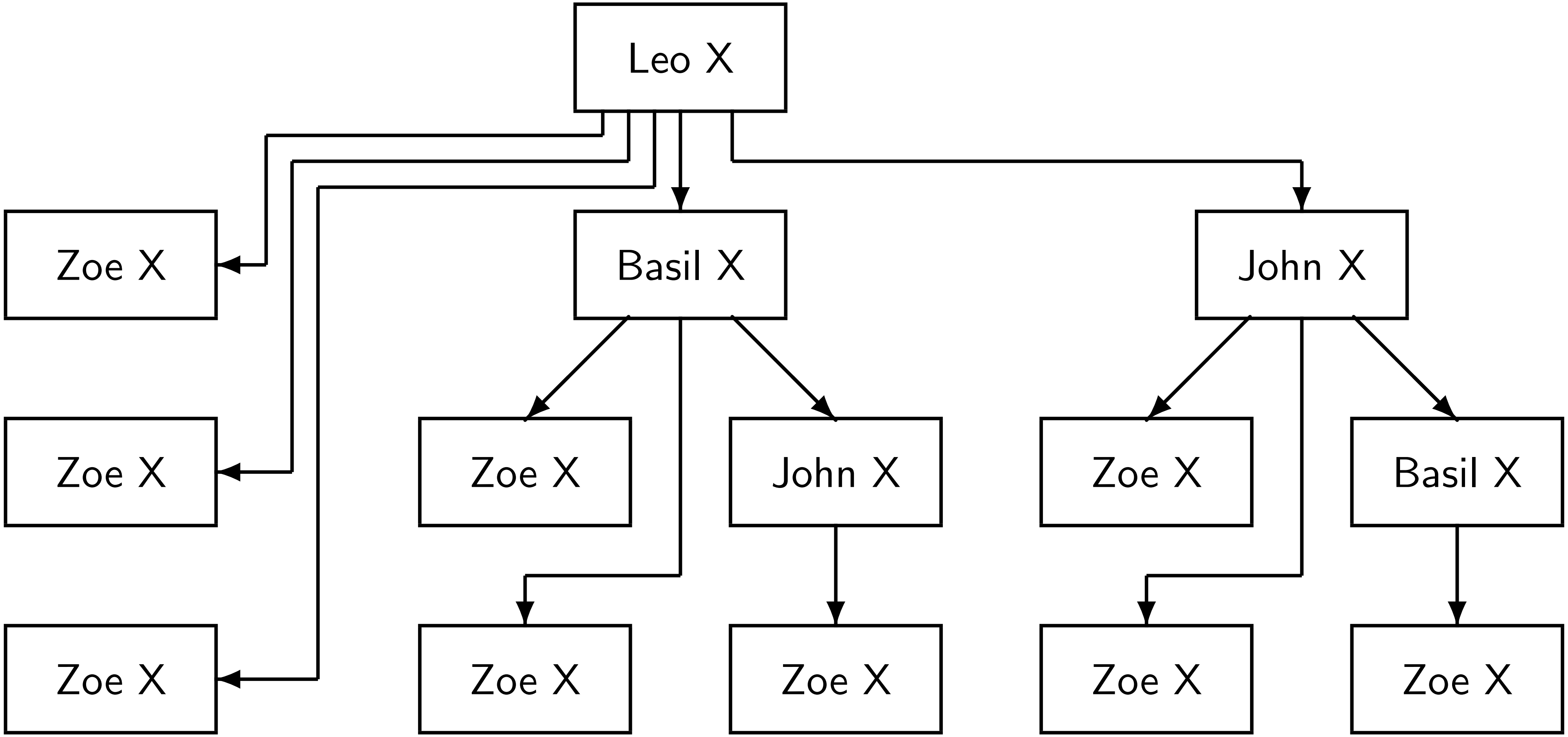
Complexity of the Byzantine Generals Algorithm

traitors	generals	messages
1	4	36
2	7	392
3	10	1790
4	13	5408

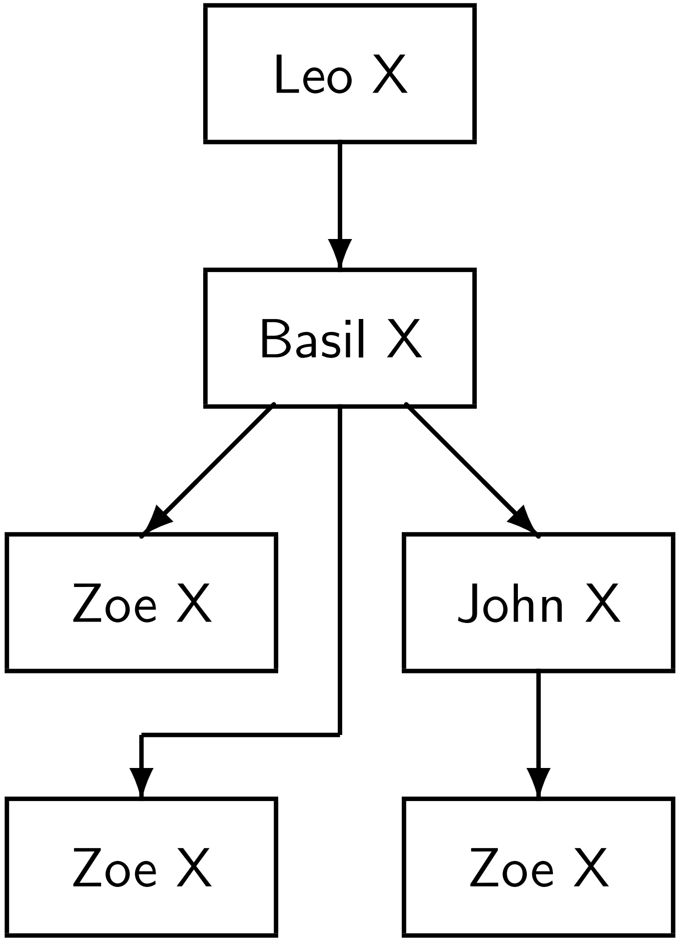
Algorithm 12.3: Consensus - flooding algorithm

```
planType finalPlan
set of planType plan ← { chooseAttackOrRetreat }
set of planType receivedPlan

do $t + 1$ times
    for all other generals G
        send(G, plan)
    for all other generals G
        receive(G, receivedPlan)
        plan ← plan ∪ receivedPlan
finalPlan ← majority(plan)
```

Flooding Algorithm with No Crash: Knowledge Tree About Leo



Flooding Algorithm with Crash: Knowledge Tree About Leo (1)



Leo X

Basil X

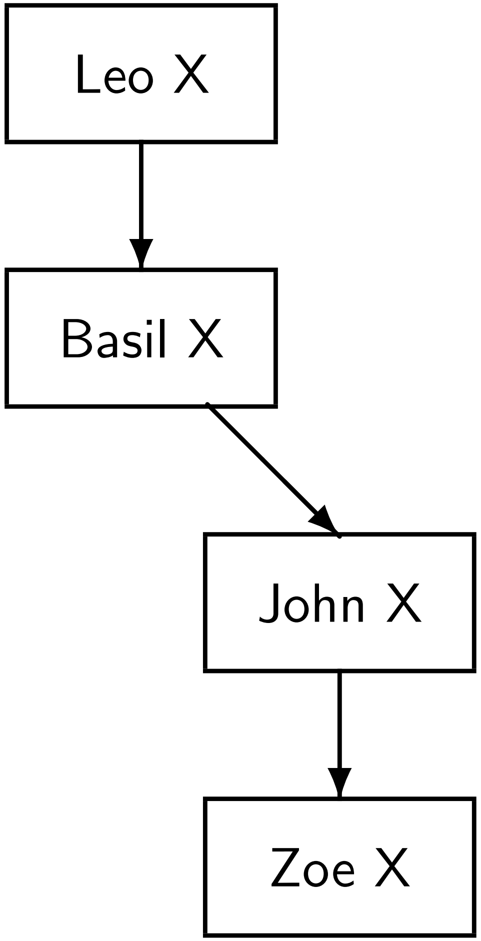
Zoe X

John X

Zoe X

Zoe X

Flooding Algorithm with Crash: Knowledge Tree About Leo (2)



Leo X

Basil X

John X

Zoe X

Algorithm 12.4: Consensus - King algorithm

```
planType finalPlan, myMajority, kingPlan
planType array[generals] plan
integer votesMajority
```

```
plan[myID] ← chooseAttackOrRetreat

do **two** times
    for all *other* generals G                // First and third rounds
        send(G, myID, plan[myID])
    for all *other* generals G
        receive(G, plan[G])
    myMajority ← majority(plan)
    votesMajority ← number of votes for myMajority
```

Algorithm 12.4: Consensus - King algorithm (continued)

```
p9:     if my turn to be king              // Second and fourth rounds
p10:        for all other generals G
p11:            send(G, myID, myMajority)
p12:        plan[myID] ← myMajority
        else
p13:        receive(kingID, kingPlan)
p14:        if votesMajority > 3
p15:            plan[myID] ← myMajority
            else
p16:            plan[myID] ← kingPlan

p17: finalPlan ← plan[myID]                // Final decision
```

Scenario for King Algorithm - First King Loyal General Zoe (1)

Basil							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
A	A	R	R	R	R	3	

John							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
A	A	R	A	R	A	3	

Leo							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
A	A	R	A	R	A	3	

Zoe							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
A	A	R	R	R	R	3	

Scenario for King Algorithm - First King Loyal General Zoe (2)

Basil							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R							R

John							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
	R						R

Leo							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
		R					R

Zoe							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
				R			

Scenario for King Algorithm - First King Loyal General Zoe (3)

Basil							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R	R	R	?	R	R	4–5	

John							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R	R	R	?	R	R	4–5	

Leo							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R	R	R	?	R	R	4–5	

Zoe							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R	R	R	?	R	R	4–5	

Scenario for King Algorithm - First King Traitor Mike (1)

Basil							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R							R

John							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
	A						A

Leo							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
		A					A

Zoe							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
				R			R

Scenario for King Algorithm - First King Traitor Mike (2)

Basil							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R	A	A	?	R	?	3	

John							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R	A	A	?	R	?	3	

Leo							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R	A	A	?	R	?	3	

Zoe							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
R	A	A	?	R	?	3	

Scenario for King Algorithm - First King Traitor Mike (3)

Basil							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
A							A

John							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
	A						A

Leo							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
		A					A

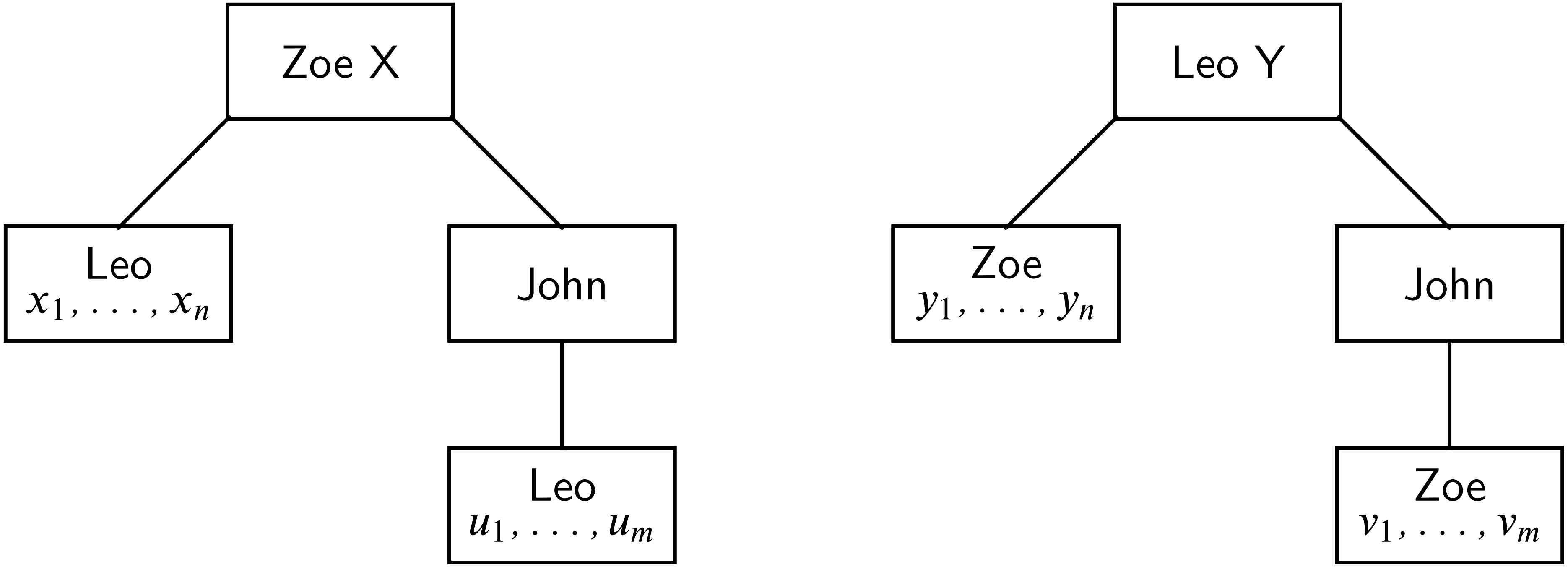
Zoe							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
				A			

Complexity of Byzantine Generals and King Algorithms

traitors	generals	messages
1	4	36
2	7	392
3	10	1790
4	13	5408

traitors	generals	messages
1	5	48
2	9	240
3	13	672
4	17	1440

Impossibility with Three Generals (1)



Zoe X

Leo
$x_1, \ldots, x_n$

John

Leo
$u_1, \ldots, u_m$

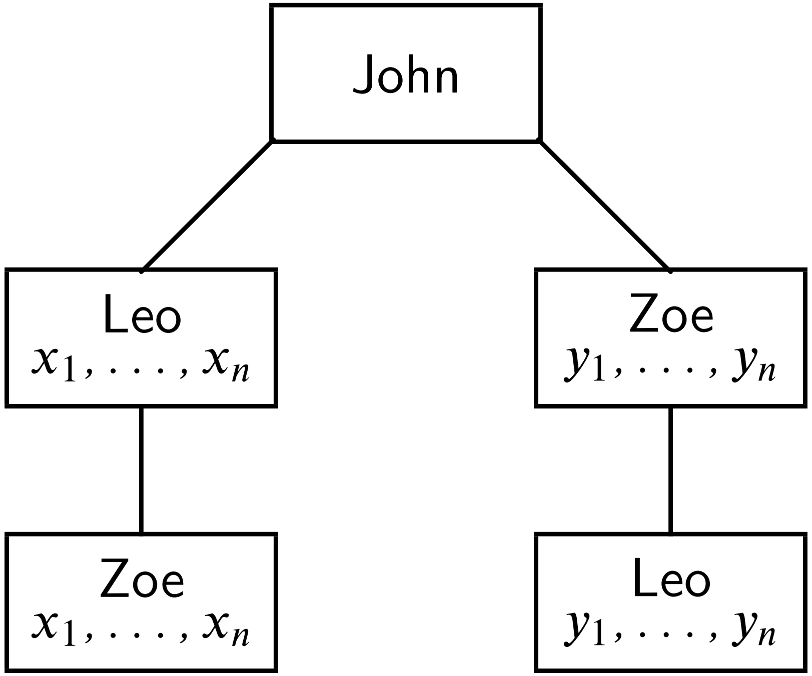
Leo Y

Zoe
$y_1, \ldots, y_n$

John

Zoe
$v_1, \ldots, v_m$

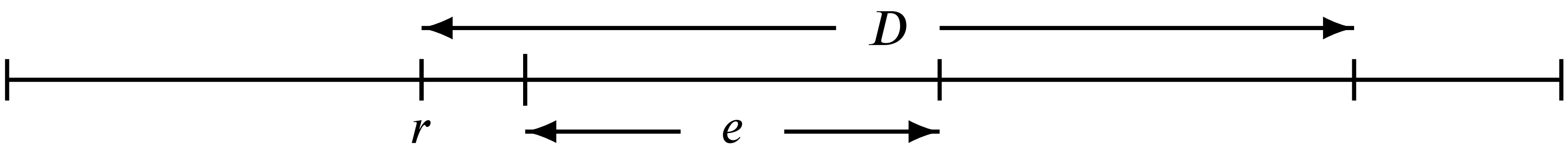
Impossibility with Three Generals (2)



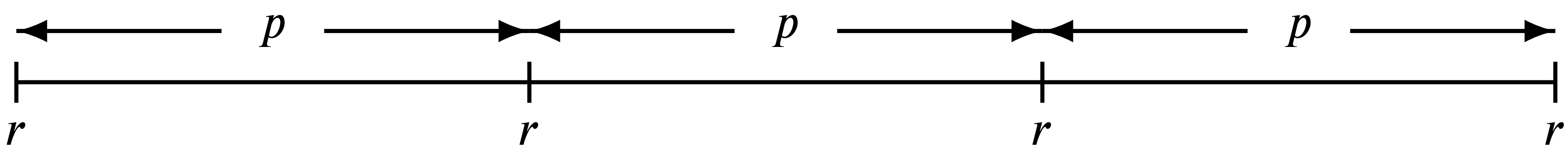
Exercise for Byzantine Generals Algorithm

Zoe					
general	plan	reported by			majority
		Basil	John	Leo	
Basil	R		A	R	?
John	A	R		A	?
Leo	R	R	R		?
Zoe	A				A
					?

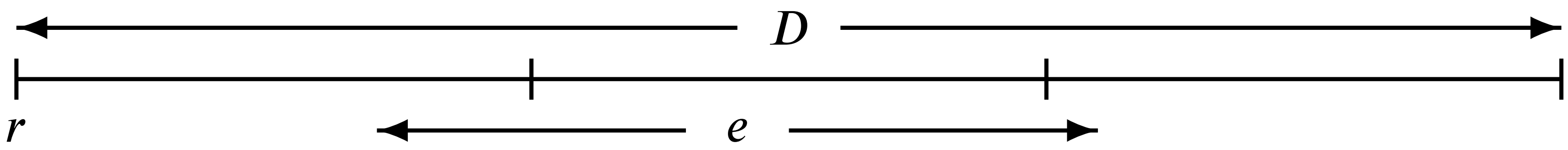
Release Time, Execution Time and Relative Deadline



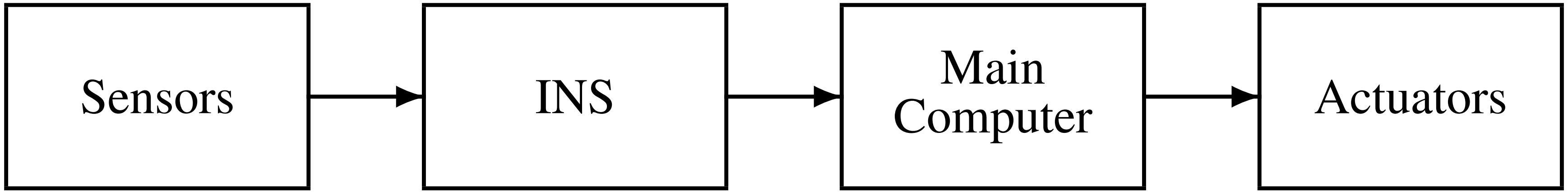
Periodic Task



Deadline is a Multiple of the Period



Architecture of Ariane Control System



Sensors

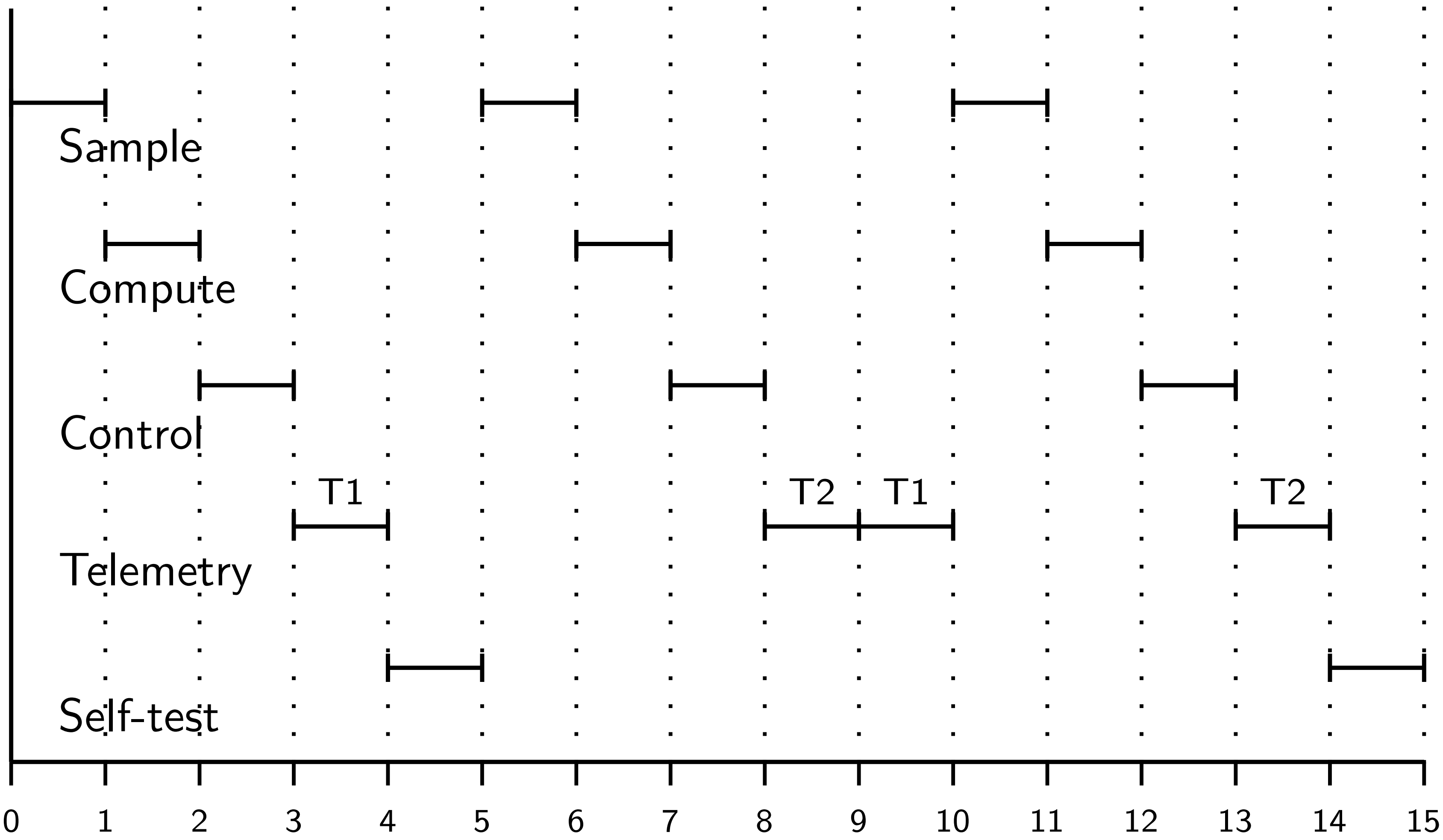
INS

Main
Computer

Actuators

Synchronization Window in the Space Shuttle

Synchronous System



Synchronous System Scheduling Table

0	1	2	3	4
Sample	Compute	Control	Telemetry 1	Self-test

5	6	7	8	9
Sample	Compute	Control	Telemetry 2	Telemetry 1

10	11	12	13	14
Sample	Compute	Control	Telemetry 2	Self-test

Algorithm 13.1: Synchronous scheduler

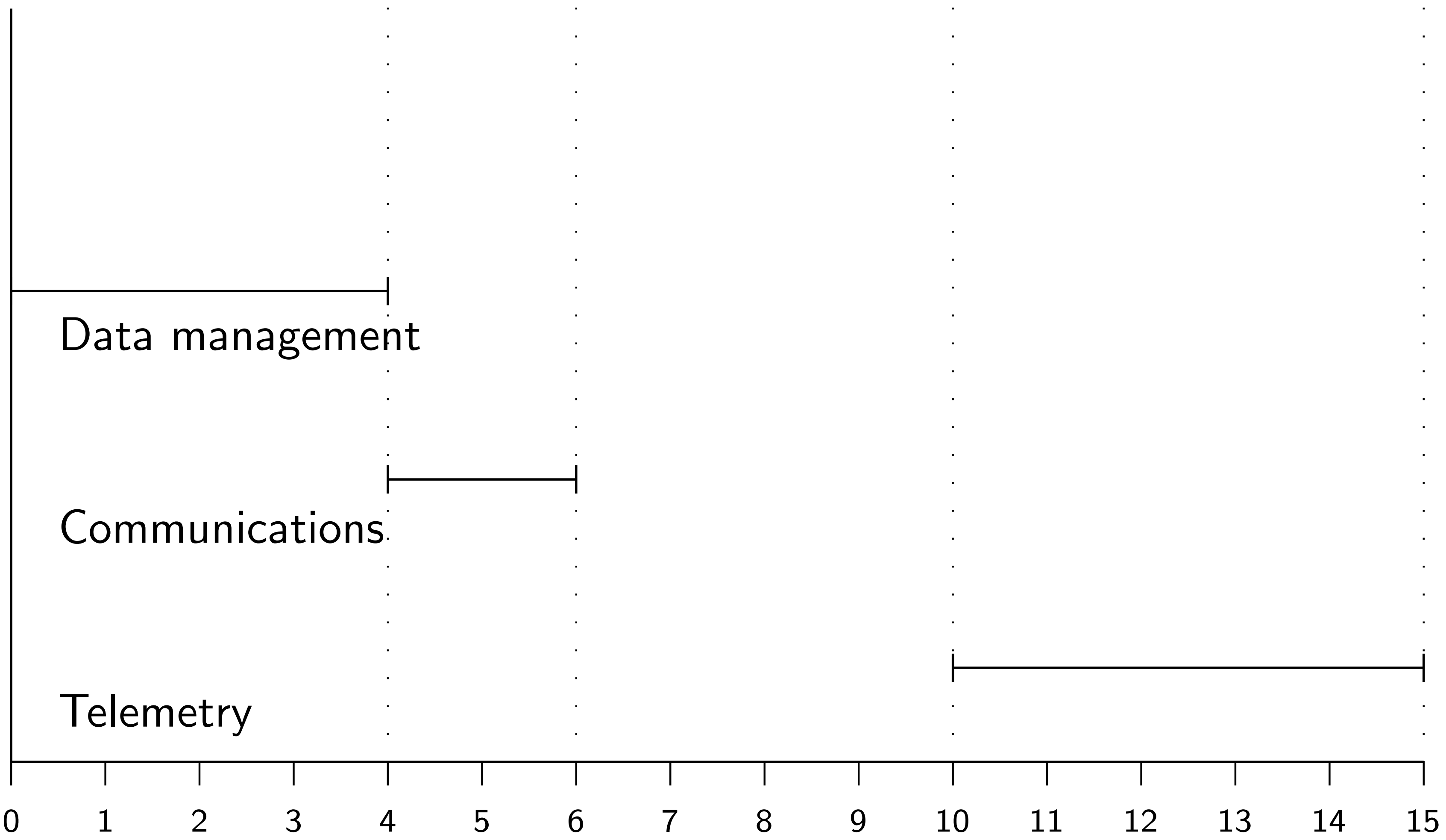
```
taskAddressType array[0..numberFrames-1] tasks ←
        [task address,. . . ,task address]
integer currentFrame ← 0

loop
    await beginning of frame
    invoke tasks[currentFrame]
    increment currentFrame modulo numberFrames
```

Algorithm 13.2: Producer-consumer (synchronous system)

queue of dataType buffer1, buffer2

sample	**compute**	**control**
dataType d	dataType d1, d2	dataType d
p1: d ← sample	q1: d1 ← take(buffer1)	r1: d ← take(buffer2)
p2: append(d, buffer1)	q2: d2 ← compute(d1)	r2: control(d)
p3:	q3: append(d2, buffer2)	r3:

Asynchronous System



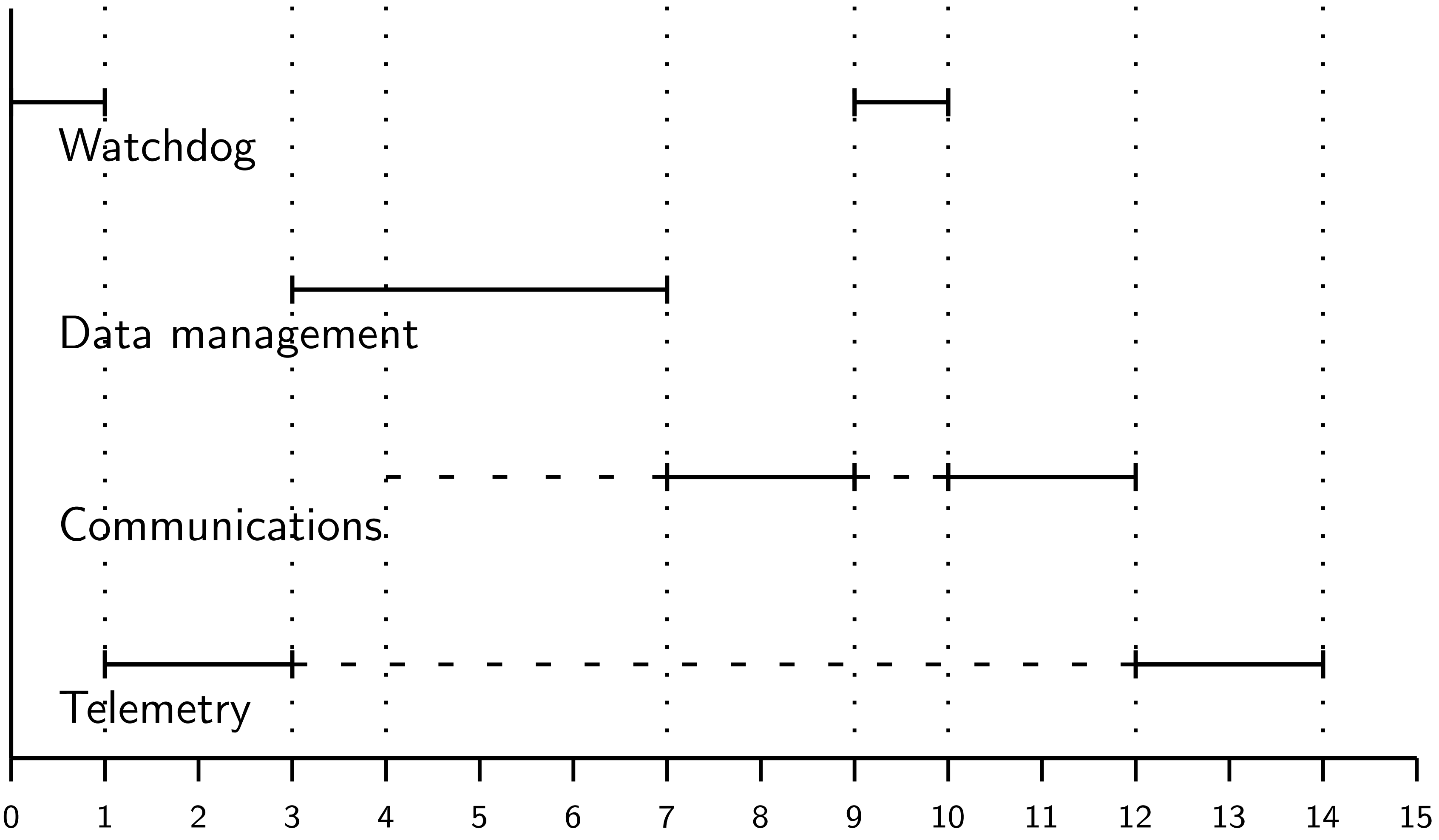
Algorithm 13.3: Asynchronous scheduler

```
queue of taskAddressType readyQueue ← . . .
taskAddressType currentTask

    loop forever
p1:     await readyQueue not empty
p2:         currentTask ← take head of readyQueue
p3:         invoke currentTask
```

Algorithm 13.4: Preemptive scheduler

```
queue of taskAddressType readyQueue ← . . .
taskAddressType currentTask

loop forever
    await a scheduling event
    if currentTask.priority < highest priority of a task on readyQueue
        save partial computation of currentTask and place on readyQueue
        currentTask ← take task of highest priority from readyQueue
        invoke currentTask
    else if currentTask's timeslice is past and
            currentTask.priority = priority of some task on readyQueue
        save partial computation of currentTask and place on readyQueue
        currentTask ← take a task of the same priority from readyQueue
        invoke currentTask
    else resume currentTask
```

Preemptive Scheduling



Algorithm 13.5: Watchdog supervision of response time

boolean ran ← false

data management

```
loop forever
    do data management
    ran ← true
    rejoin readyQueue
```

watchdog

```
loop forever
    await ninth frame
    if ran is false
        notify response-time overflow
    ran ← false
    rejoin readyQueue
```

Algorithm 13.6: Real-time buffering - throw away new data

```
queue of dataType buffer ← empty queue
```

sample

```
     dataType d
     loop forever
p1:     d ← sample
p2:     if buffer is full do nothing
p3:     else append(d,buffer)
```

compute

```
     dataType d
     loop forever
q1:     await buffer not empty
q2:     d ← take(buffer)
q3:     compute(d)
```

Algorithm 13.7: Real-time buffering - overwrite old data

```
queue of dataType buffer ← empty queue
```

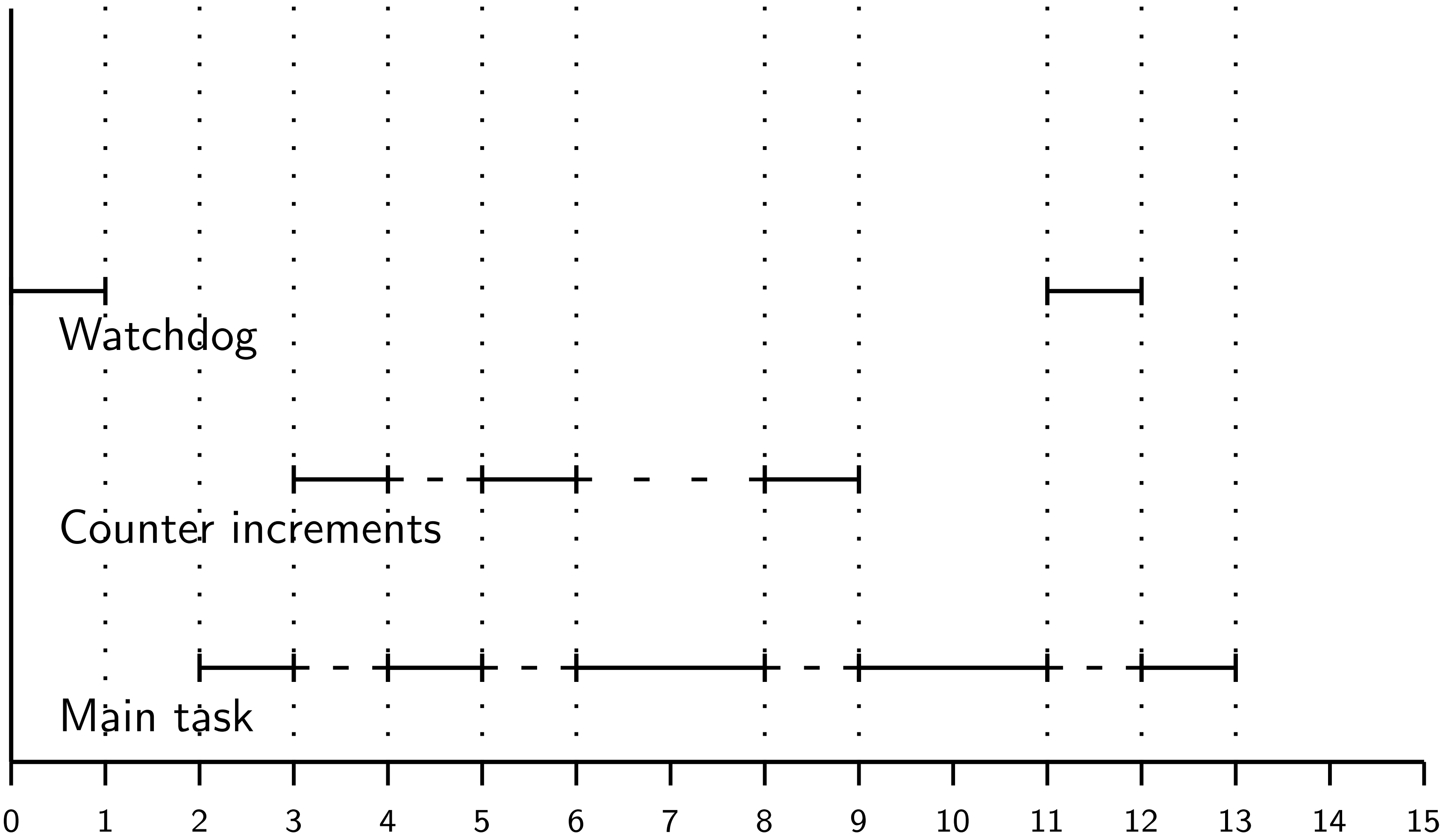
sample

```
     dataType d
     loop forever
p1:     d ← sample
p2:     append(d, buffer)
p3:
```

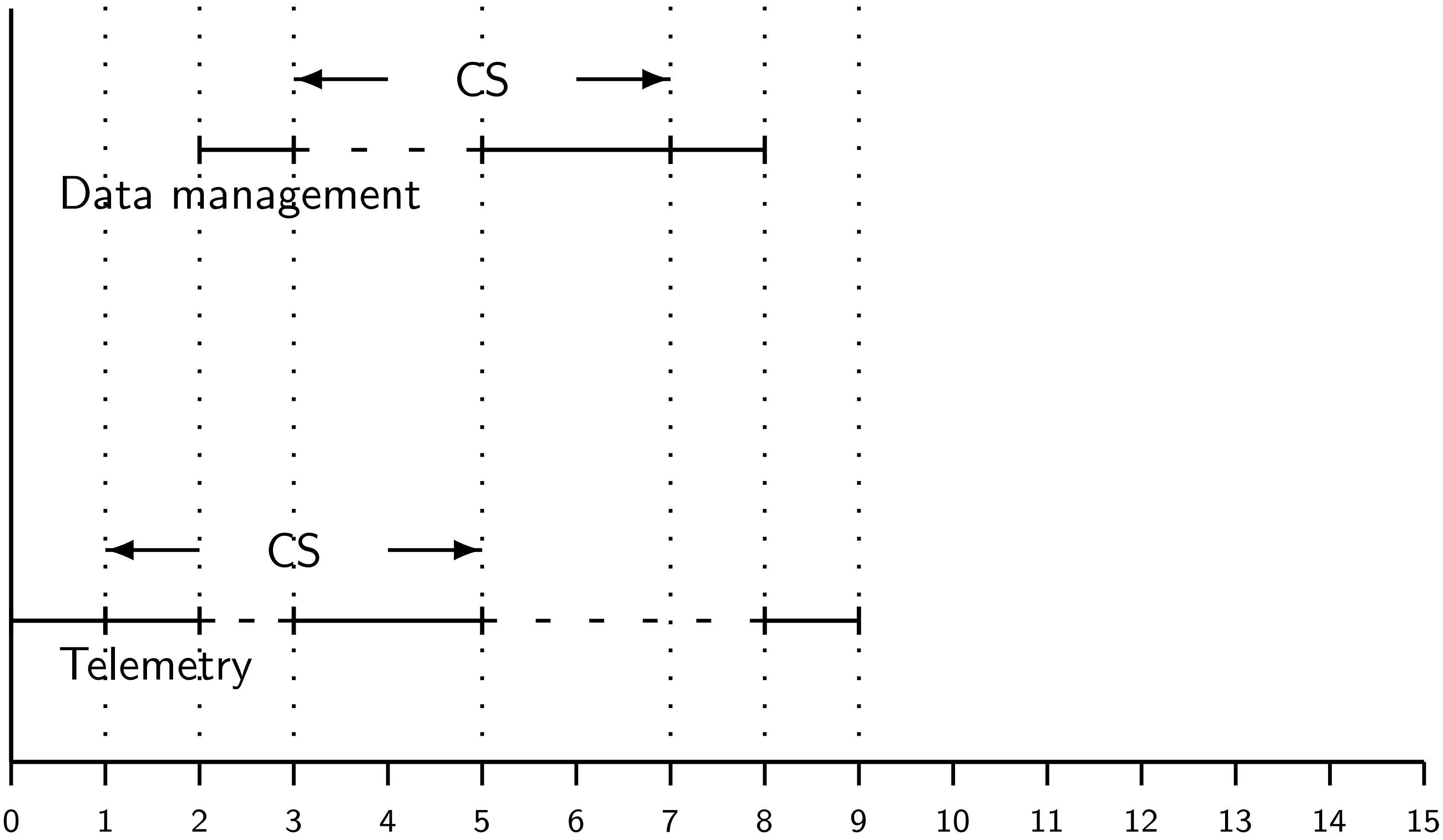
compute

```
     dataType d
     loop forever
q1:     await buffer not empty
q2:     d ← take(buffer)
q3:     compute(d)
```

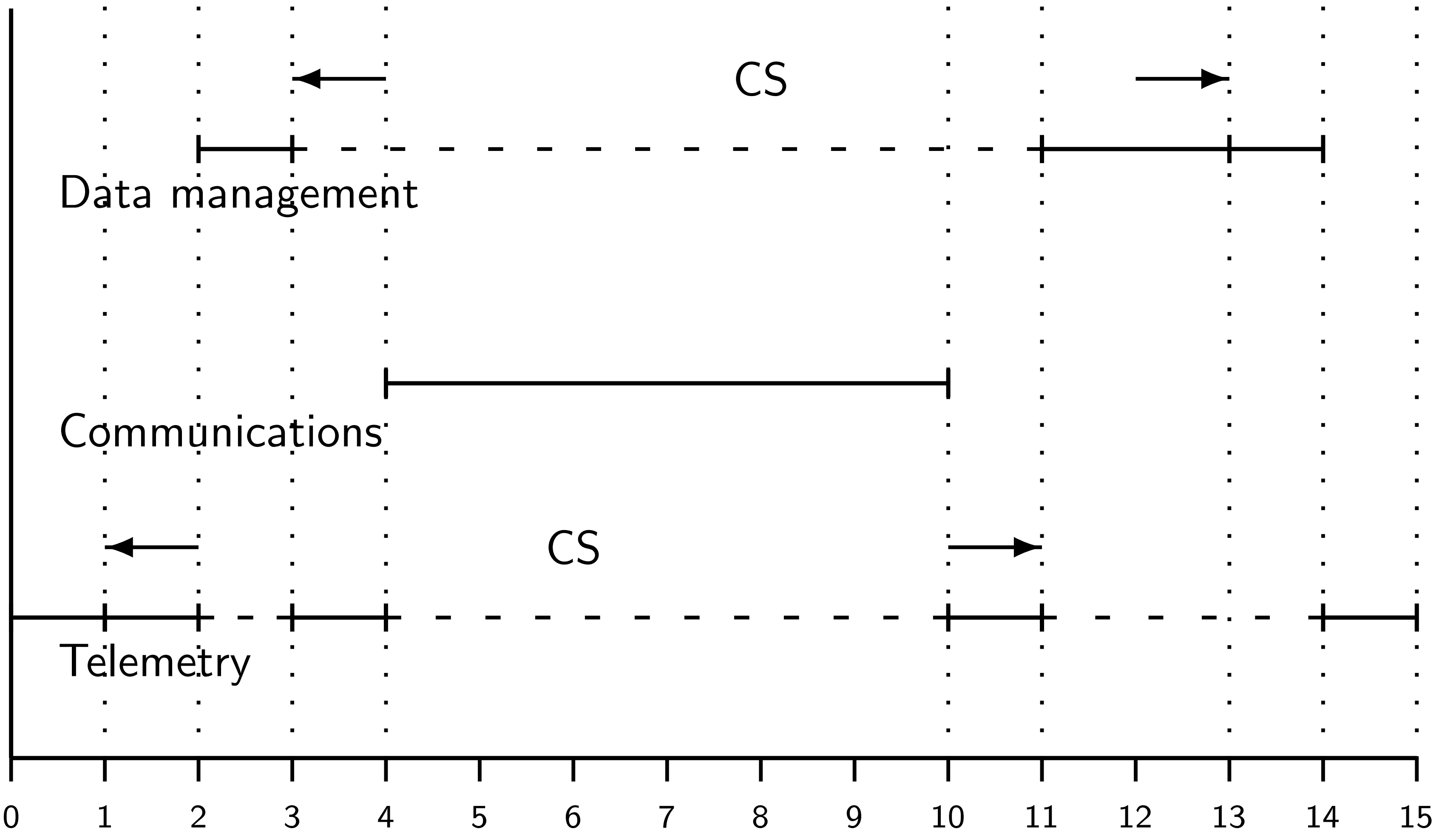
Interrupt Overflow on Apollo 11



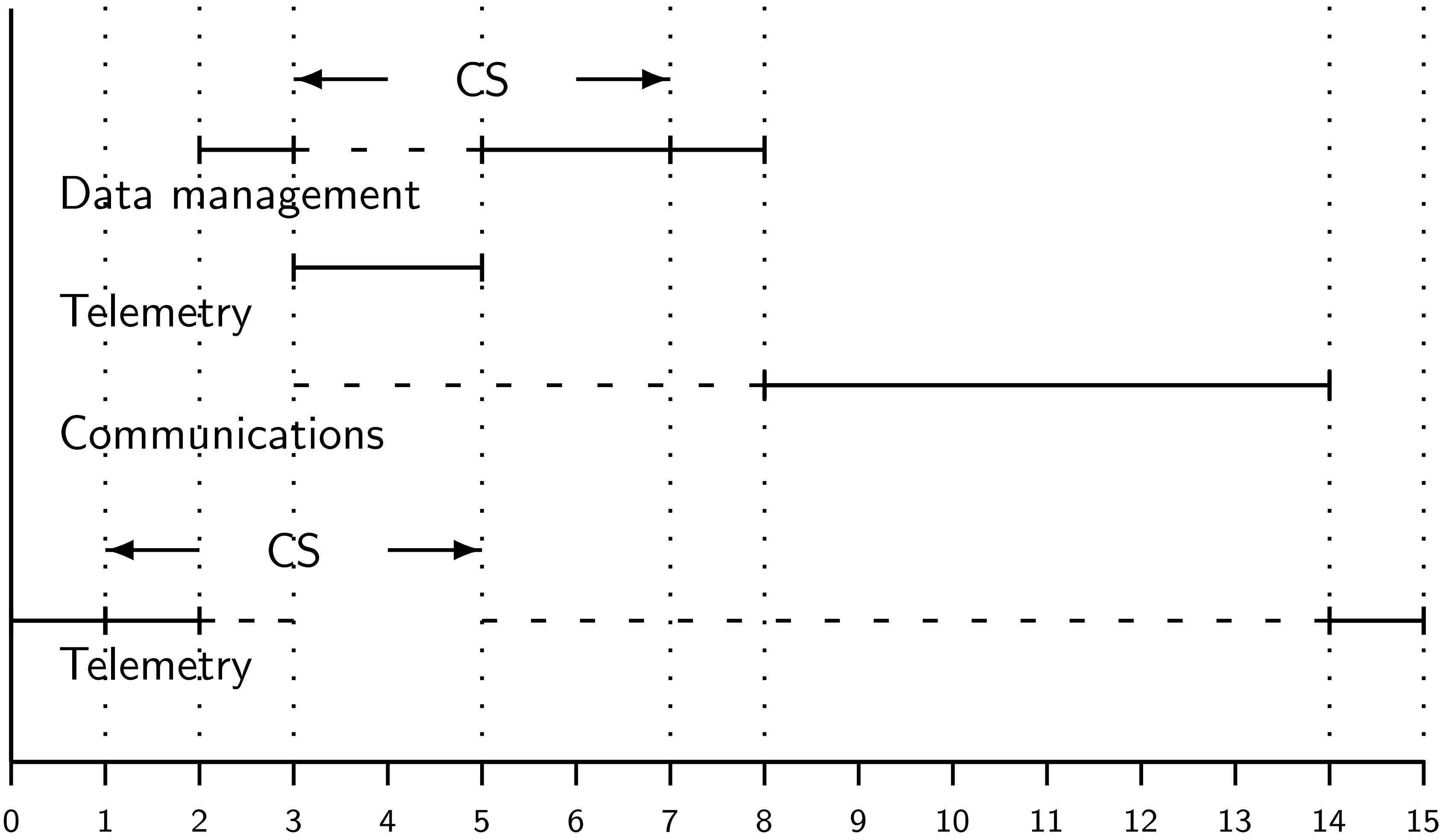
Priority Inversion (1)



Priority Inversion (2)



Priority Inheritance



Priority Inversion in Promela (1)

```
mtype = { idle, blocked,  nonCS, CS, long };
mtype data = idle, comm = idle, telem = idle;
#define ready(p)  (p != idle  && p != blocked)

active proctype Data() {
    do
    ::   data = nonCS;
         enterCS(data);
         exitCS(data);
         data = idle ;
    od
}
```

Priority Inversion in Promela (1)

```
active proctype Comm() provided (!ready(data)) {
    do
    ::  comm = long;
        comm = idle;
    od
}
active proctype Telem()
        provided (!ready(data) && !ready(comm)) {
    do
    ::  telem = nonCS;
        enterCS(telem);
        exitCS(telem);
        telem = idle;
    od
}
```

Priority Inversion in Promela (2)

```
bit sem = 1;

inline enterCS(state) {
    atomic {
        if
        :: sem == 0 ->
            state = blocked;
            sem != 0;
        :: else ->
        fi ;
        sem = 0;
        state = CS;
    }
}

```

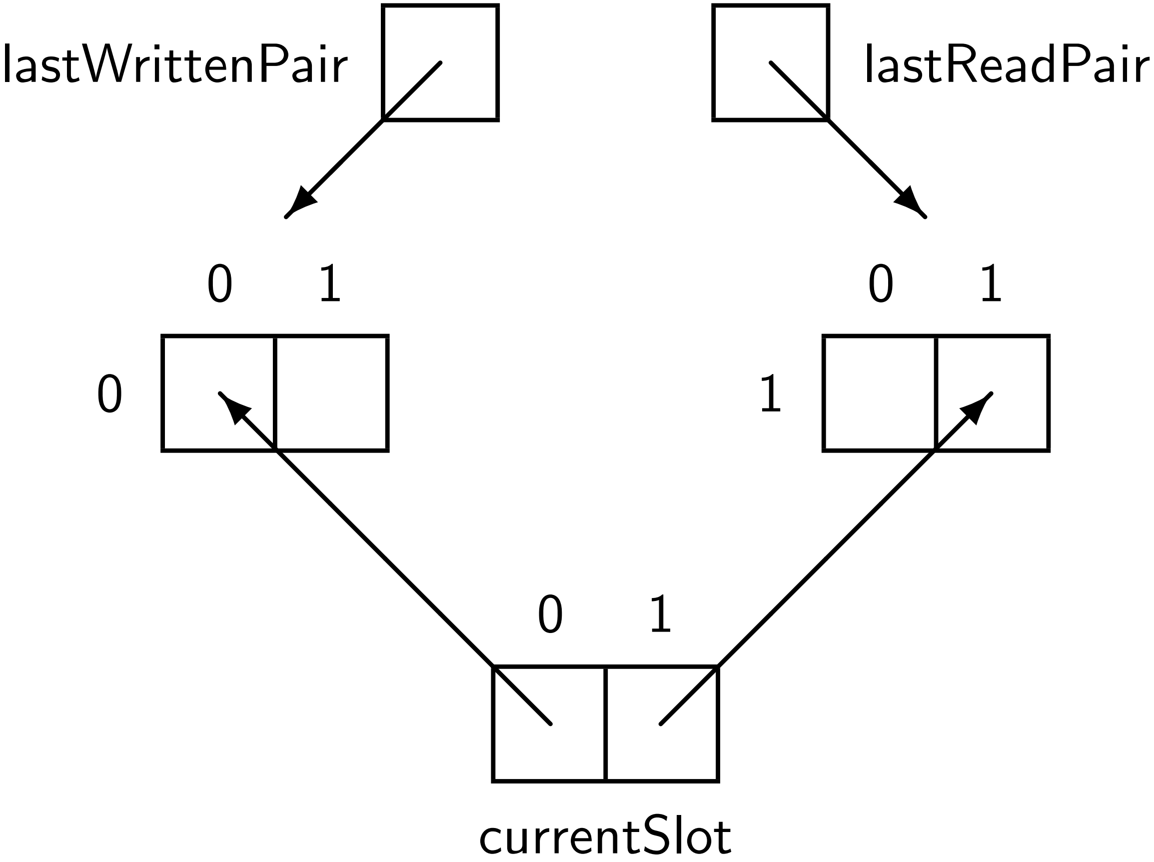
Priority Inversion in Promela (2)

```
inline exitCS(state) {
    atomic {
        sem = 1;
        state = idle
    }
}
```

Priority Inheritance in Promela

```
#define inherit(p) (p == CS)
active proctype Data() {
  do
  ::   data = nonCS;
       assert( ! (telem == CS && comm == long) );
       enterCS(data); exitCS(data);
       data = idle;
  od
}
active proctype Comm()
  provided (!ready(data) && !inherit(telem))
    { ...  }
active proctype Telem()
  provided (! ready(data) && !ready(comm) || inherit(telem))
    { ...  }
```

Data Structures in Simpson's Algorithm



Algorithm 13.8: Simpson's four-slot algorithm

```
            dataType array[0..1,0..1] data ← default initial values
            bit array[0..1] currentSlot ← { 0, 0 }
            bit lastWrittenPair ← 1, lastReadPair ← 1
```

writer

```
    bit writePair, writeSlot
    dataType item
    loop forever
p1:     item ← produce
p2:     writePair ← 1− lastReadPair
p3:     writeSlot ← 1− currentSlot[writePair]
p4:     data[writePair, writeSlot] ← item
p5:     currentSlot[writePair] ← writeSlot
p6:     lastWrittenPair ← writePair
```

Algorithm 13.8: Simpson's four-slot algorithm (continued)

reader

```
     bit readPair, readSlot
     dataType item
     loop forever
p7:     readPair ← lastWrittenPair
p8:     lastReadPair ← readPair
p9:     readSlot ← currentSlot[readPair]
p10:    item ← data[readPair, readSlot]
p11:    consume(item)
```

Algorithm 13.9: Event signaling

binary semaphore s ← 0

p

```
p1: if decision is to wait for event
p2:     wait(s)
```

q

```
q1: do something to cause event
q2: signal(s)
```

Suspension Objects in Ada

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object)
    return Boolean;
  procedure Suspend_Until_True(
    S : in out Suspension_Object);
private
  -- not specified by the language
end Ada.Synchronous_Task_Control;
```

Algorithm 13.10: Suspension object - event signaling

Suspension_Object SO ← (false by default)

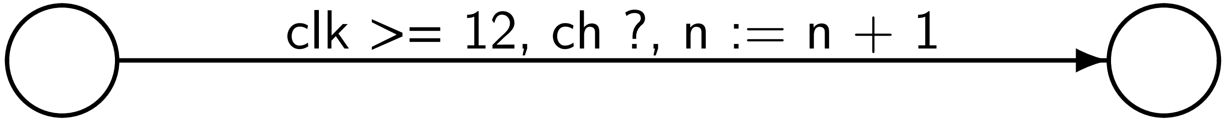
p

```
p1: if decision is to wait for event
p2:     Suspend_Until_True(SO)
```

q

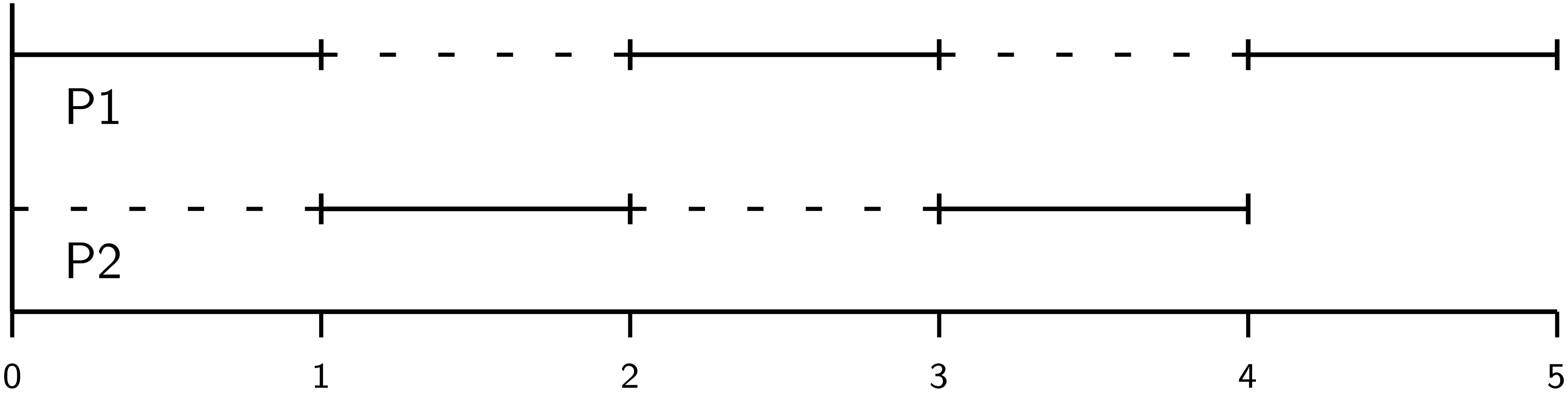
```
q1: do something to cause event
q2: Set_True(SO)
```

Transition in UPPAAL

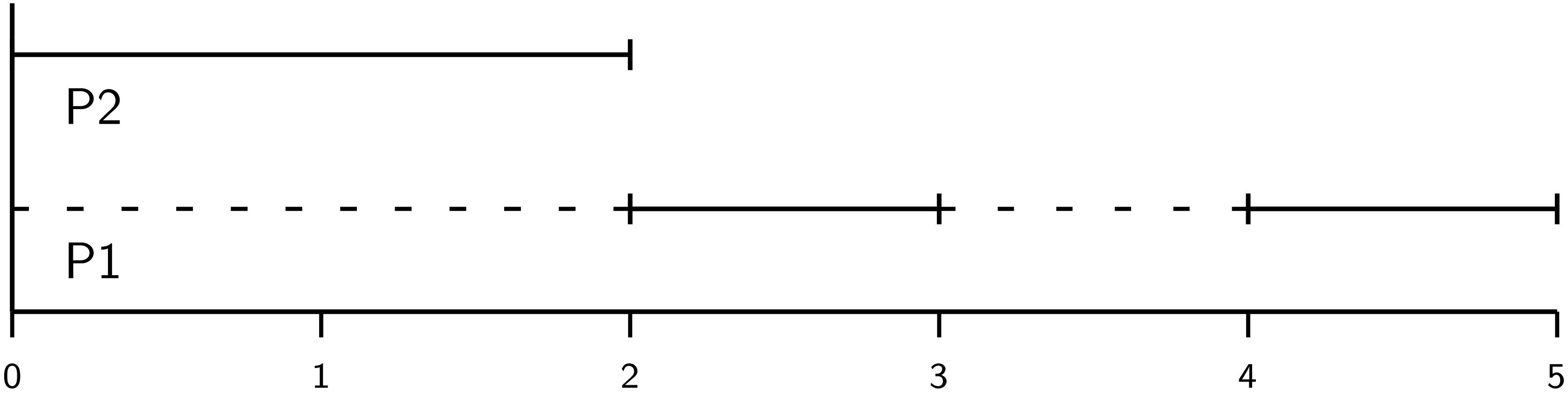


clk >= 12, ch ?, n := n + 1

Feasible Priority Assignment



Infeasible Priority Assignment



Algorithm 13.11: Periodic task

```
constant integer period ← . . .

    integer next ← currentTime
    loop forever
p1:     delay next − currentTime
p2:     compute
p3:     next ← next + period
```

Semantics of Propositional Operators

A	$v(A_1)$	$v(A_2)$	$v(A)$
$\neg A_1$	T		F
$\neg A_1$	F		T
$A_1 \vee A_2$	F	F	F
$A_1 \vee A_2$	otherwise		T
$A_1 \wedge A_2$	T	T	T
$A_1 \wedge A_2$	otherwise		F
$A_1 \rightarrow A_2$	T	F	F
$A_1 \rightarrow A_2$	otherwise		T
$A_1 \leftrightarrow A_2$	$v(A_1) = v(A_2)$		T
$A_1 \leftrightarrow A_2$	$v(A_1) \neq v(A_2)$		F

Wason Selection Task

$p3$

$p5$

$flag = 1$

$flag = 0$

Algorithm B.1: Verification example

```
integer x1, integer x2
integer y1 ← 0, integer y2 ← 0, integer y3

p1:  read(x1,x2)
p2:  y3 ← x1
p3:  while y3 ≠ 0
p4:      if y2+1 = x2
p5:          y1 ← y1 + 1
p6:          y2 ← 0
p7:      else
p8:          y2 ← y2 + 1
p9:      y3 ← y3 − 1
p10: write(y1,y2)
```

Spark Program for Integer Division

```
--# main_program;
procedure Divide(X1,X2: in Integer ; Q,R : out Integer )
--# derives Q, R from X1,X2;
--# pre (X1 >= 0) and (X2 > 0);
--# post (X1 = Q * X2 + R) and (X2 > R) and (R >= 0);
is
    N: Integer ;
```

Spark Program for Integer Division

```
begin
    Q := 0; R := 0; N := X1;
    while N /= 0
    --# assert (X1 = Q*X2+R+N) and (X2 > R) and (R >= 0);
    loop
        if  R+1 = X2 then
            Q := Q + 1; R := 0;
        else
            R := R + 1;
        end if;
        N := N – 1;
    end loop;
end Divide;
```

Integer Division

```
procedure Divide(X1,X2: in Integer ;  Q,R : out Integer )  is
     N: Integer ;
begin
    -- pre (X1 >= 0) and (X2 > 0);
    Q := 0; R := 0; N := X1;
    while N /= 0
    -- assert (X1 = Q*X2+R+N) and (X2 > R) and (R >= 0);
    loop
        if  R+1 = X2 then Q := Q + 1; R := 0;
        else  R := R + 1;
        end if;
        N := N - 1;
    end loop;
    -- post (X1 = Q * X2 + R) and (X2 > R) and (R >= 0);
end Divide;
```

Verification Conditions for Integer Division

Precondition to assertion:

$$(X1 \geq 0) \wedge (X2 > 0) \rightarrow$$
$$(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0).$$

Assertion to postcondition:

$$(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0) \wedge (N = 0) \rightarrow$$
$$(X1 = Q \cdot X2 + R) \wedge (X2 > R) \wedge (R \geq 0).$$

Assertion to assertion by then branch:

$$(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0) \wedge (R + 1 = X2) \rightarrow$$
$$(X1 = Q' \cdot X2 + R' + N') \wedge (X2 > R') \wedge (R' \geq 0).$$

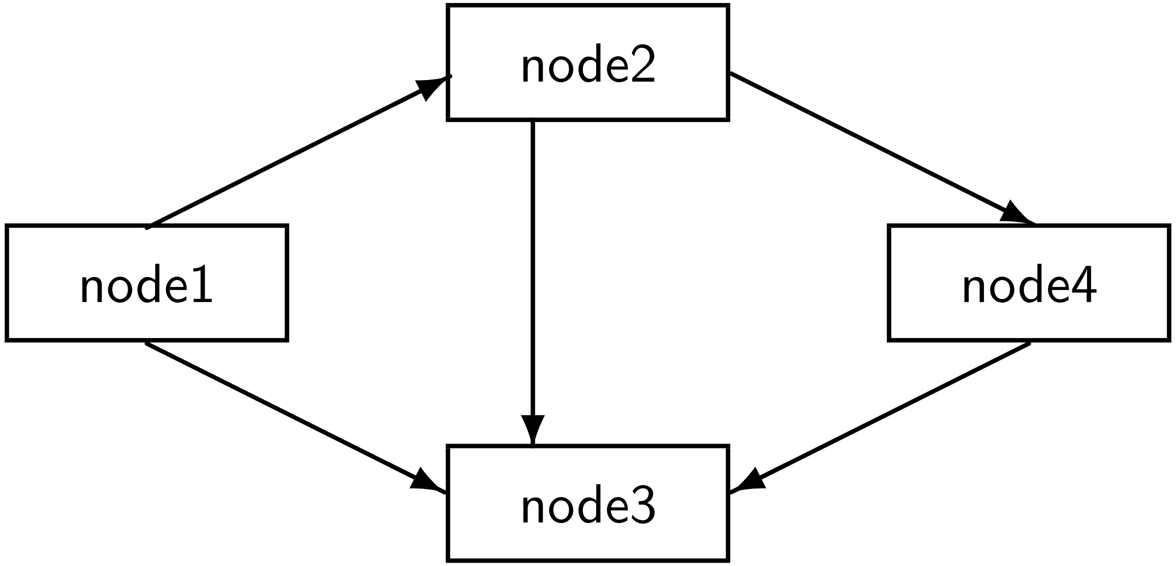
Assertion to assertion by else branch:

$$(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0) \wedge (R + 1 \neq X2) \rightarrow$$
$$(X1 = Q' \cdot X2 + R' + N') \wedge (X2 > R') \wedge (R' \geq 0).$$

The Sleeping Barber

n	producer	consumer	Buffer	notEmpty
1	**append(d, Buffer)**	wait(notEmpty)	[ ]	0
2	**signal(notEmpty)**	wait(notEmpty)	[1]	0
3	append(d, Buffer)	**wait(notEmpty)**	[1]	1
4	append(d, Buffer)	**d ← take(Buffer)**	[1]	0
5	**append(d, Buffer)**	wait(notEmpty)	[ ]	0

Synchronizing Precedence



Algorithm C.1: Barrier synchronization

global variables for synchronization

```
    loop forever
p1:     wait to be released
p2:     computation
p3:     wait for all processes to finish their computation
```

The Stable Marriage Problem

Man	List of women			
1	2	4	1	3
2	3	1	4	2
3	2	3	1	4
4	4	1	3	2

Woman	List of men			
1	2	1	4	3
2	4	3	1	2
3	1	4	3	2
4	2	1	4	3

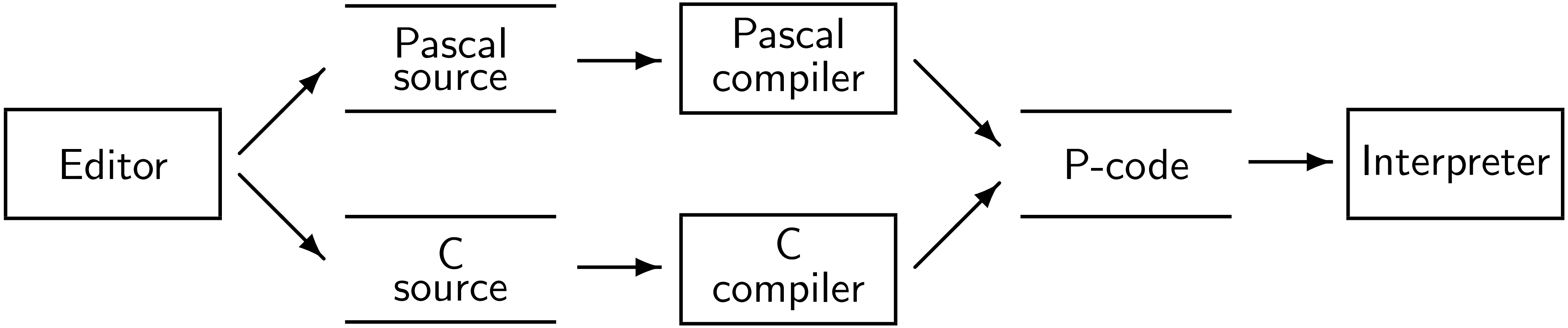
Algorithm C.2: Gale-Shapley algorithm for stable marriage

```
integer list freeMen ← {1,. . . ,n}
integer list freeWomen ← {1,. . . ,n}
integer pair-list matched ← ∅
integer array[1..n, 1..n] menPrefs ← . . .
integer array[1..n, 1..n] womenPrefs ← . . .
integer array[1..n] next ← 1

p1: while freeMen ≠ ∅, choose some m from freeMen
p2:     w ← menPrefs[m, next[m]]
p3:     next[m] ← next[m] + 1
p4:     if w in freeWomen
p5:         add (m,w) to matched, and remove w from freeWomen
p6:     else if w prefers m to m' // where (m',w) in matched
p7:         replace (m',w) in matched by (m,w), and remove m' from freeMen
p8:     else // w rejects m, and nothing is changed
```

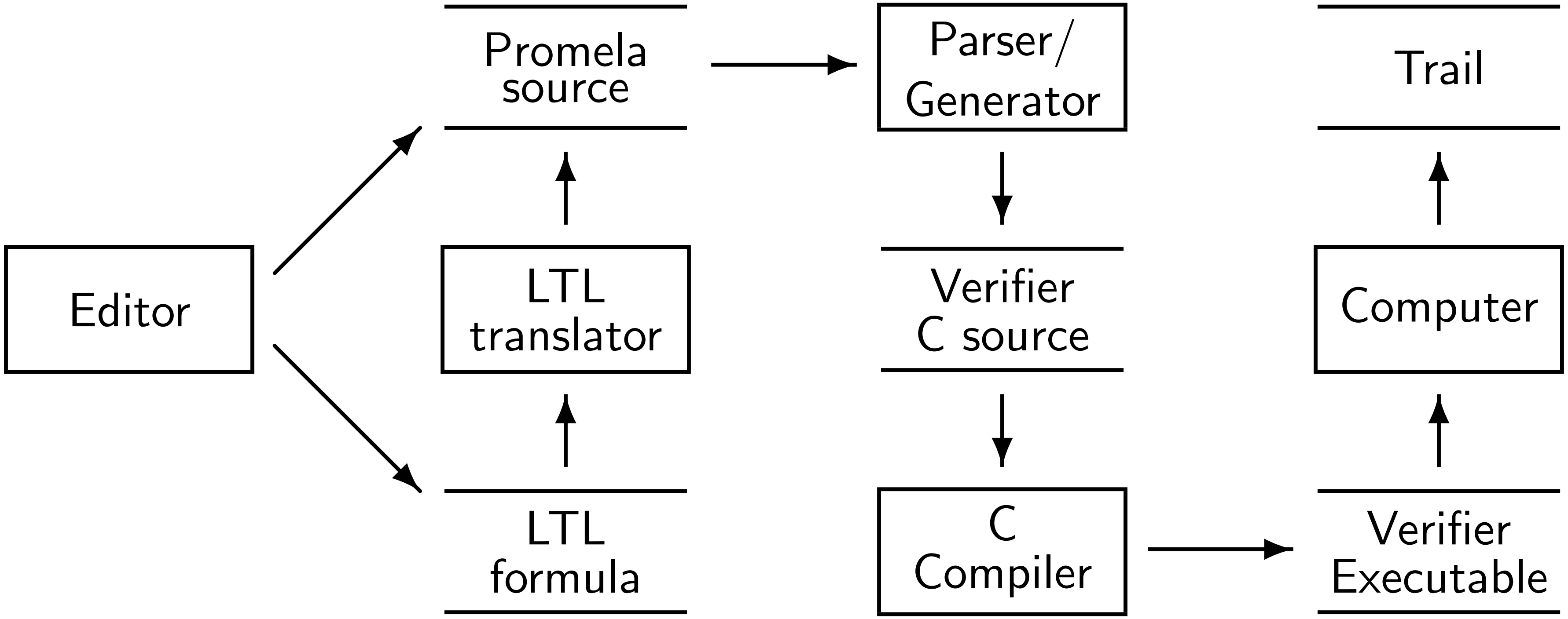
The n-Queens Problem

1	Q							
2							Q	
3					Q			
4								Q
5		Q						
6				Q				
7						Q		
8			Q					
	1	2	3	4	5	6	7	8

The Architecture of BACI



The Architecture of Spin



Editor

Promela source

LTL translator

LTL formula

Parser/ Generator

Verifier C source

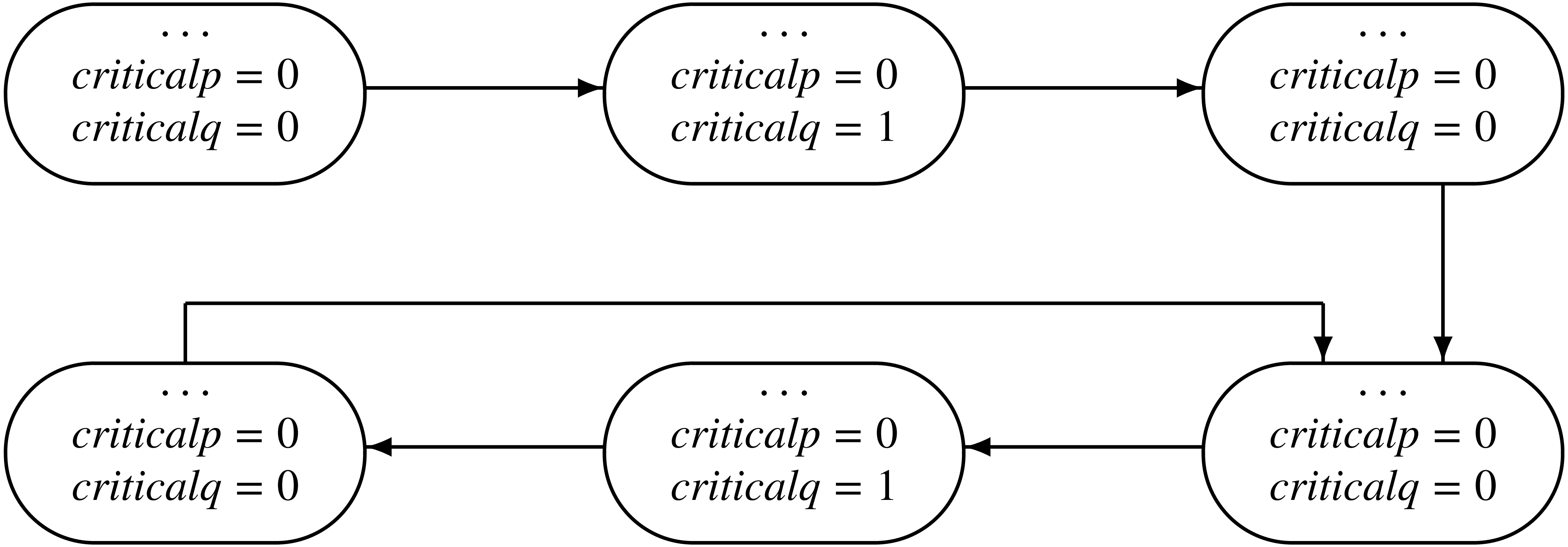
C Compiler

Trail

Computer

Verifier Executable

Cycles in a State Diagram



$\ldots$
$criticalp = 0$
$criticalq = 0$

$\ldots$
$criticalp = 0$
$criticalq = 1$

$\ldots$
$criticalp = 0$
$criticalq = 0$

$\ldots$
$criticalp = 0$
$criticalq = 0$

$\ldots$
$criticalp = 0$
$criticalq = 1$

$\ldots$
$criticalp = 0$
$criticalq = 0$