

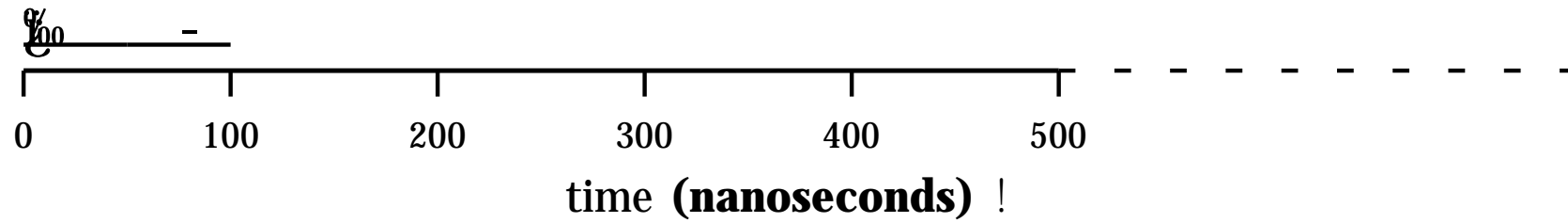
M. Ben-Ari

Principles of Concurrent and Distributed Programming

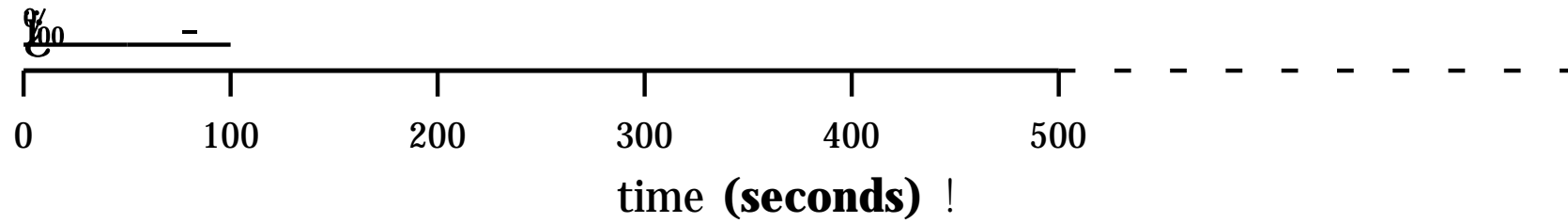
Second Edition

Addison-Wesley, 2006

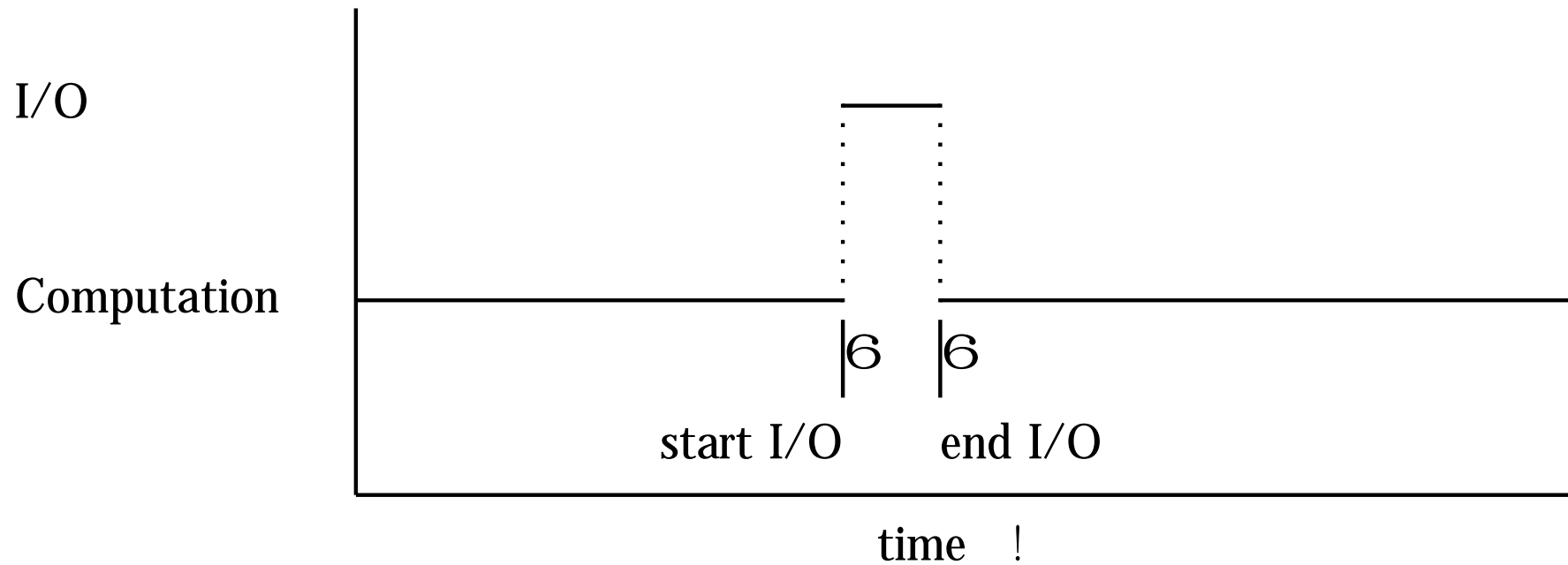
Computer Time



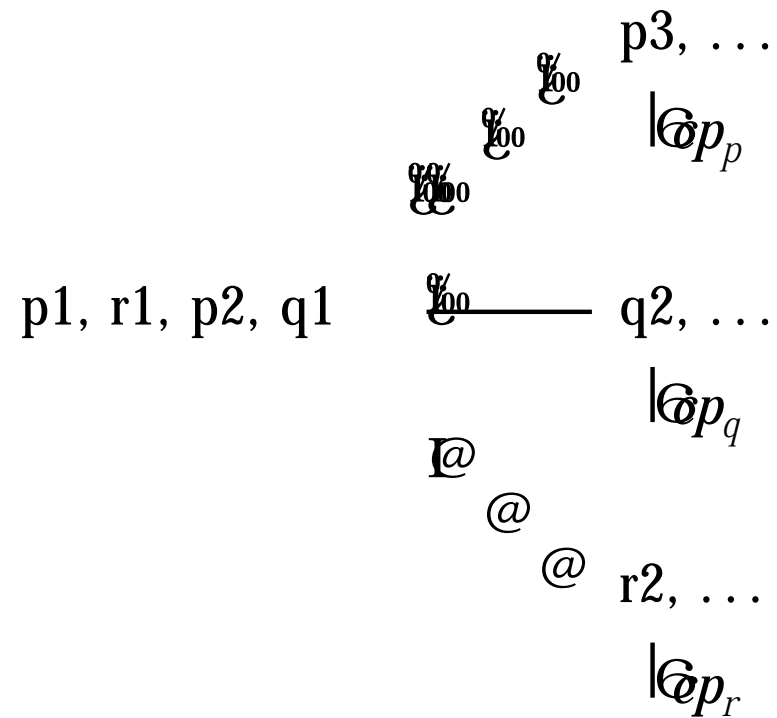
Human Time



Concurrency in an Operating System



Interleaving as Choosing Among Processes



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 $n \leftarrow 0$

p

integer $k1 \leftarrow 1$

p1: $n \leftarrow k1$

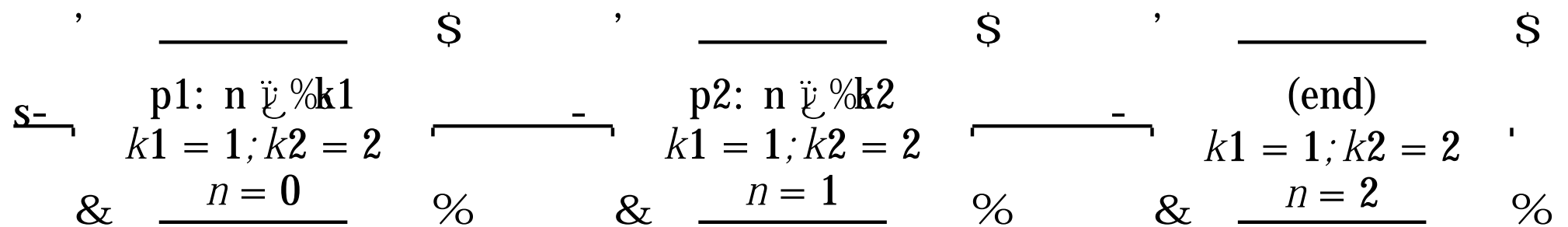
q

integer $k2 \leftarrow 2$

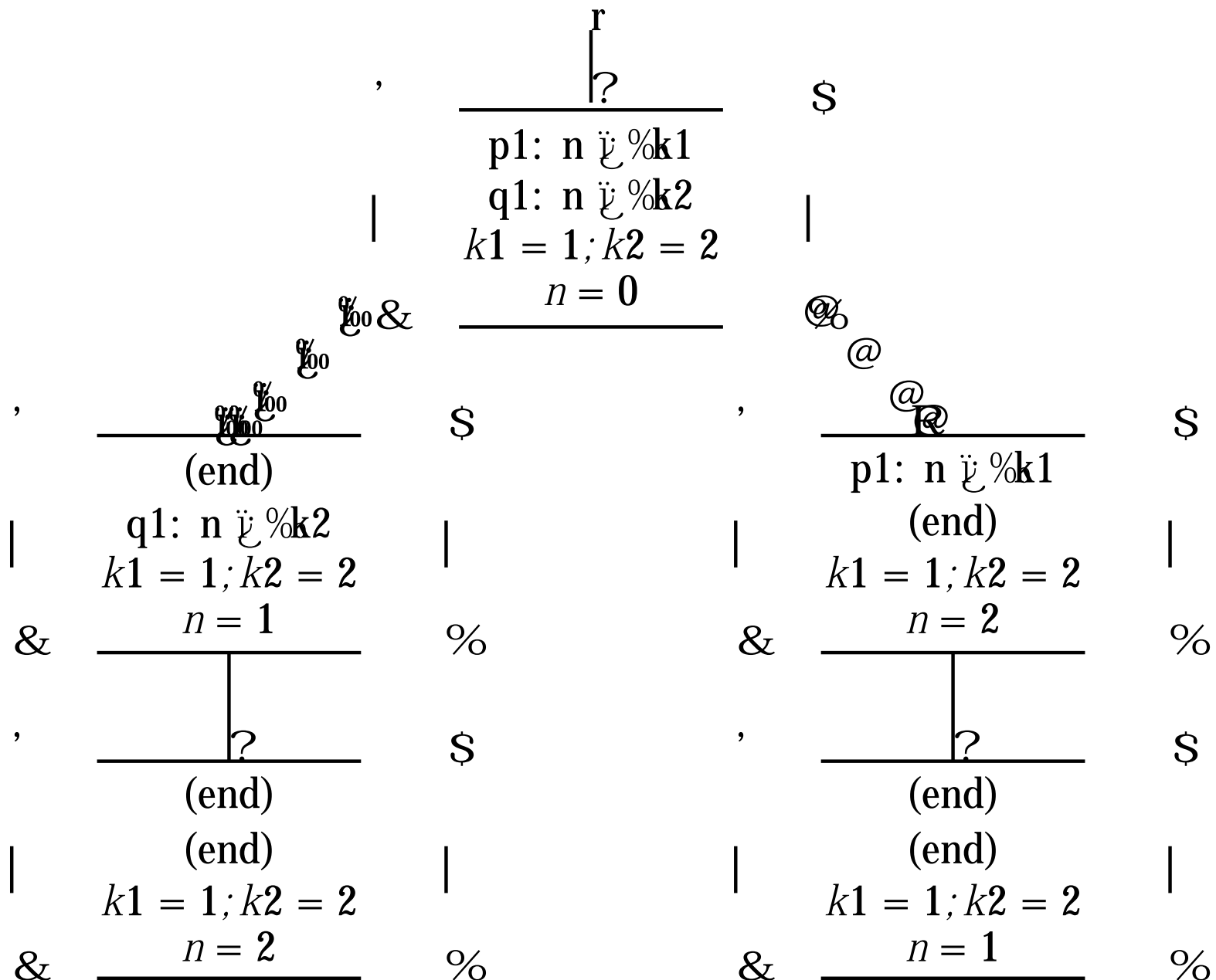
q1: $n \leftarrow k2$

Algorithm 2.2: Trivial sequential program
integer n \leftarrow 0
integer k1 \leftarrow 1 integer k2 \leftarrow 2 p1: n \leftarrow k1 p2: n \leftarrow k2

State Diagram for a Sequential Program



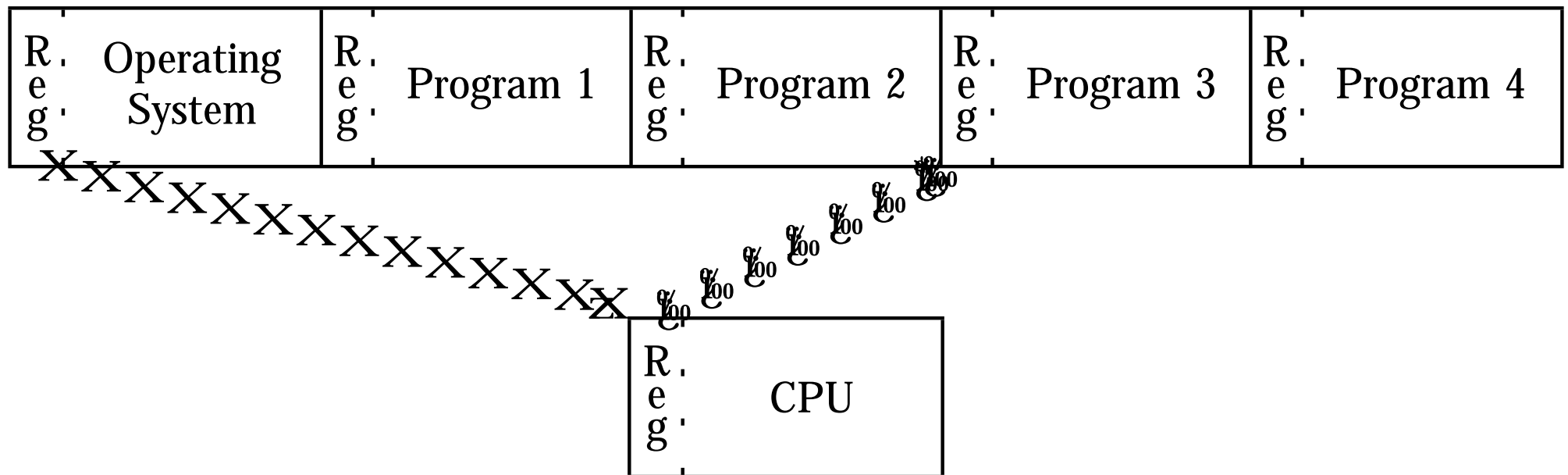
State Diagram for a Concurrent Program



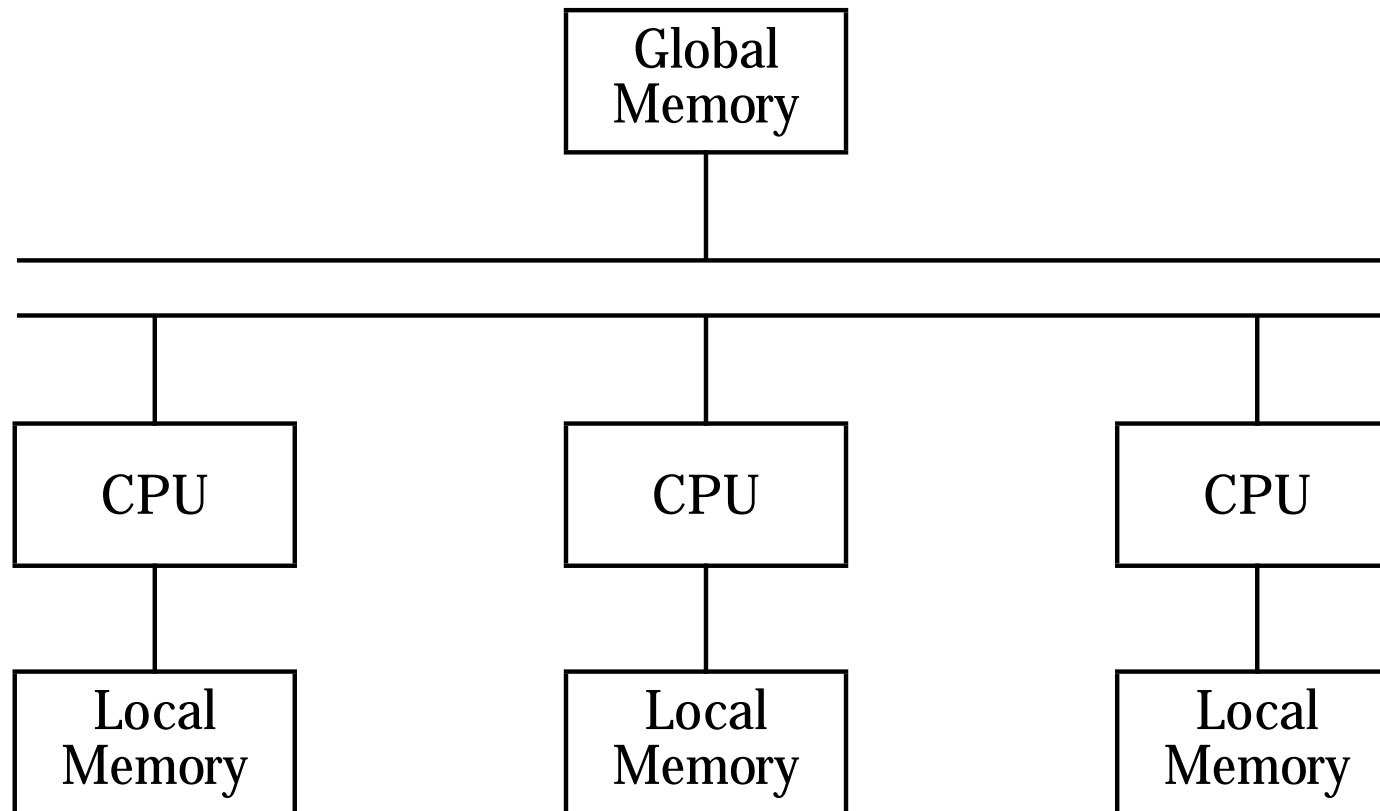
Scenario for a Concurrent Program

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

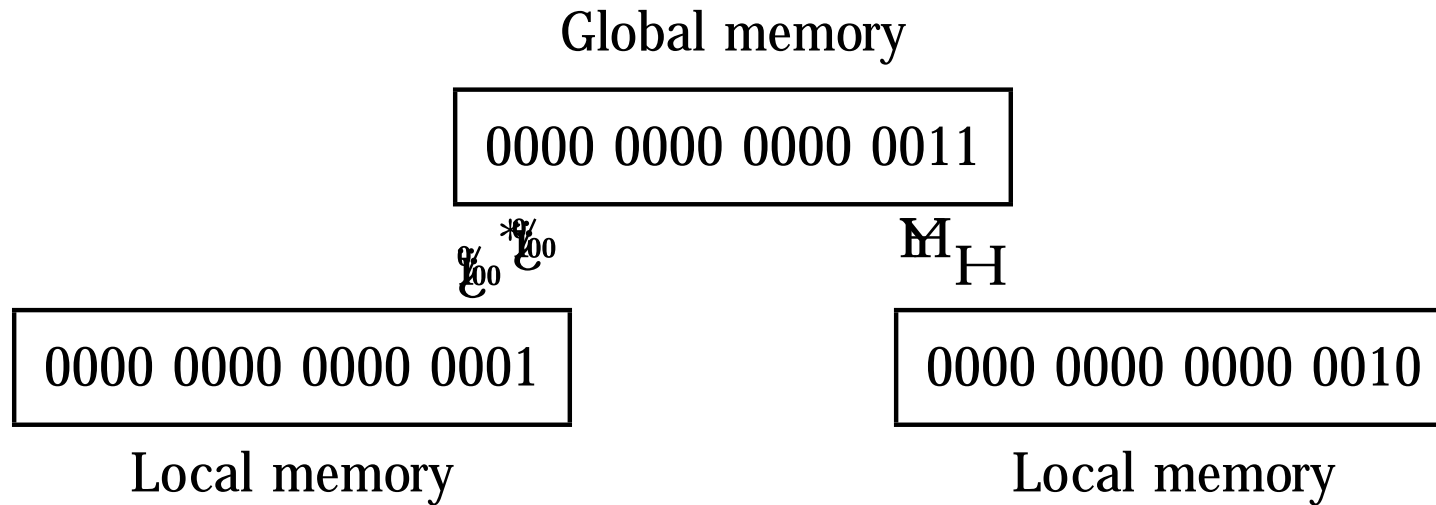
Multitasking System



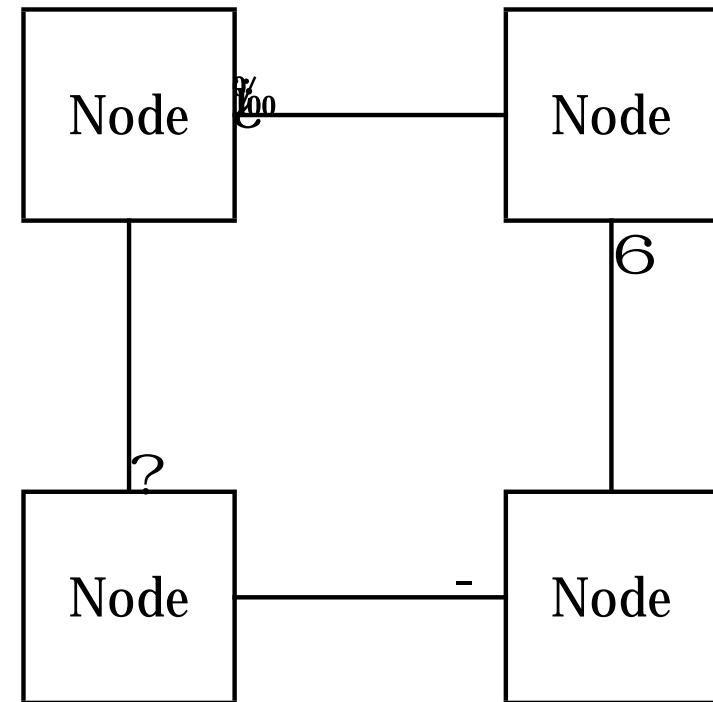
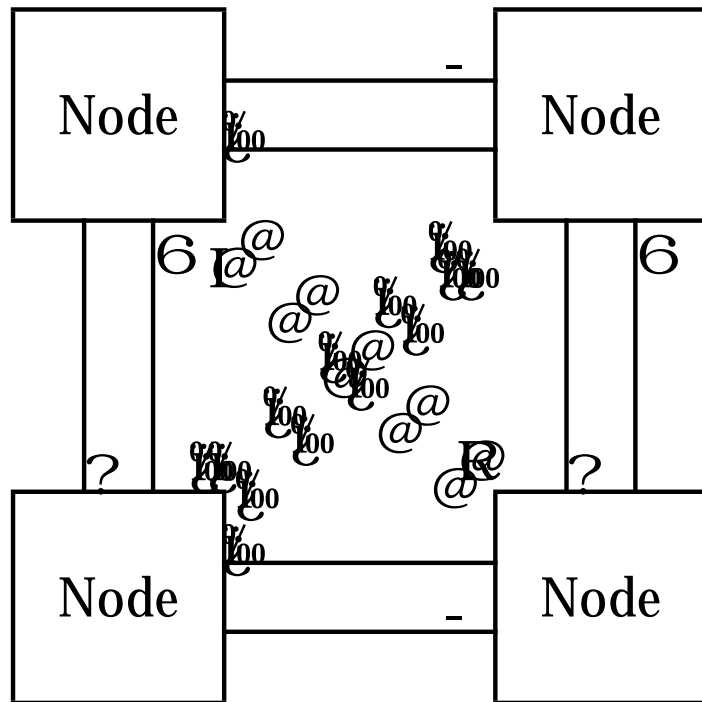
Multiprocessor Computer



Inconsistency Caused by Overlapped Execution



Distributed Systems Architecture



Algorithm 2.3: Atomic assignment statements

integer $n \in \mathbb{Z}$

p

q

p1: $n \leftarrow n + 1$

q1: $n \leftarrow n + 1$

Scenario for Atomic Assignment Statements

Process p	Process q	n
p1: $n \leftarrow n + 1$	q1: $n \leftarrow n + 1$	0
(end)	q1: $n \leftarrow n + 1$	1
(end)	(end)	2

Process p	Process q	n
p1: $n \leftarrow n + 1$	q1: $n \leftarrow n + 1$	0
p1: $n \leftarrow n + 1$	(end)	1
(end)	(end)	2

Algorithm 2.4: Assignment statements with one global reference

integer $n \in \mathbb{N}$

p

integer temp

p1: temp $\leftarrow n$

p2: $n \leftarrow \text{temp} + 1$

q

integer temp

q1: temp $\leftarrow n$

q2: $n \leftarrow \text{temp} + 1$

Correct Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp \leftarrow n	q1: temp \leftarrow n	0	?	?
p2: n \leftarrow temp + 1	q1: temp \leftarrow n	0	0	?
(end)	q1: temp \leftarrow n	1	0	?
(end)	q2: n \leftarrow 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 ← 0	q1: temp ← 0	0	?	?
p2: n ← temp + 1	q1: temp ← 0	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 $n \in \mathbb{N}$
 boolean $eg \in \{true, false\}$

p

q

p1: while $eg = false$
 p2: $n \in \mathbb{N} \wedge 1 \leq n$

q1: $eg \in \{true\}$
 q2:

Algorithm 2.6: Assignment statement for a register machine

integer $n \in \mathbb{Z}$

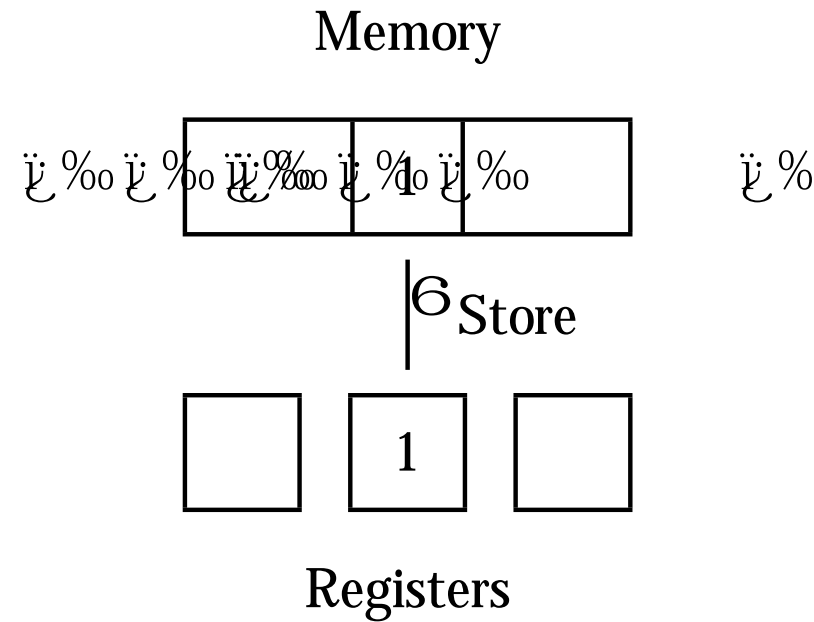
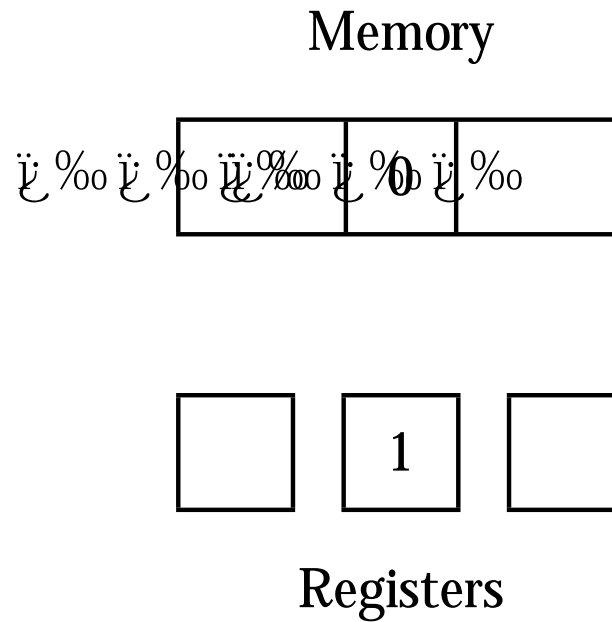
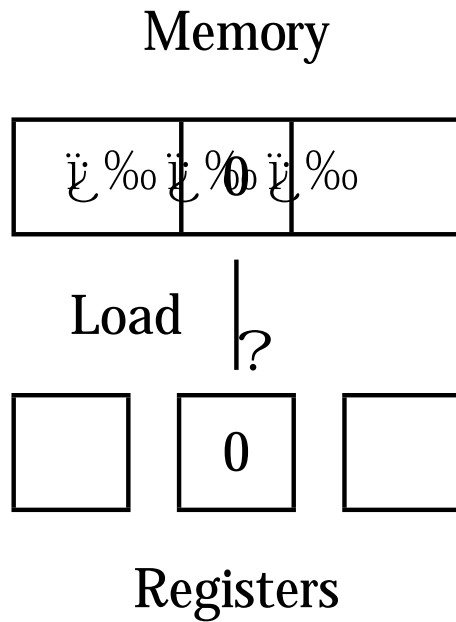
p

p1: load R1,n
p2: add R1,#1
p3: store R1,n

q

q1: load R1,n
q2: add R1,#1
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 $n \in \mathbb{N}$

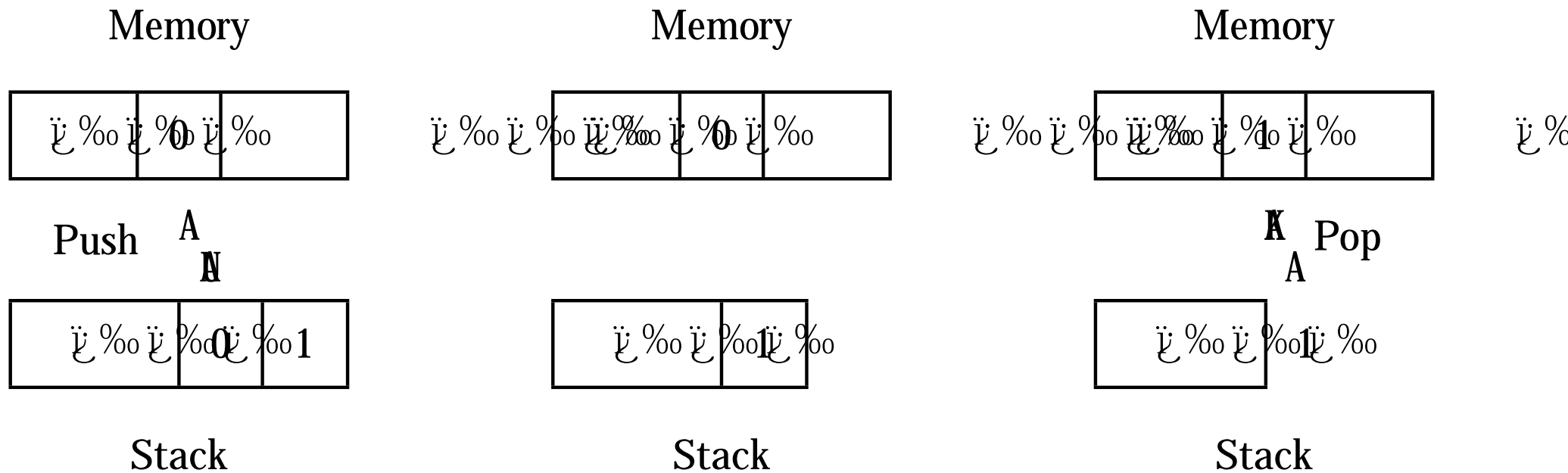
p

p1: push n
p2: push #1
p3: add
p4: pop n

q

q1: push n
q2: push #1
q3: add
q4: pop n

Stack Machine



Algorithm 2.8: Volatile variables

integer $n \in \mathbb{N}$

p

q

integer local1, local2

integer local

p1: $n \in \mathbb{N}$ some expression

q1: $local \in \mathbb{N}$ $n + 6$

p2: computation not using n

q2:

p3: $local1 \in \mathbb{N}$ $(n + 5) \in \mathbb{N}$

q3:

p4: $local2 \in \mathbb{N}$ $n + 5$

q4:

p5: $n \in \mathbb{N}$ $local1 * local2$

q5:

Algorithm 2.9: Concurrent counting algorithm

integer $n \geq 0$

p

q

integer temp

integer temp

p1: do 10 times

q1: do 10 times

p2: temp $\leftarrow n$

q2: temp $\leftarrow n$

p3: $n \leftarrow \text{temp} + 1$

q3: $n \leftarrow \text{temp} + 1$

Concurrent Program in Pascal

```
1  program count;  
2  var n: integer := 0;  
3  
4  procedure p;  
5  var temp, i: integer;  
6  begin  
7    for i := 1 to 10 do  
8      begin  
9        temp := n;  
10       n := temp + 1  
11      end  
12 end;  
13  
14  
15
```

Concurrent Program in Pascal

```
16  procedure q;  
17  var temp, i: integer;  
18  begin  
19    for i := 1 to 10 do  
20      begin  
21        temp := n;  
22        n := temp + 1  
23      end  
24  end;  
25  
26  begin  
27    cobegin p; q coend;  
28    writeln(' The value of n is ', n)  
29  end.
```

Concurrent Program in C

```
1  int n = 0;
2
3  void p() {
4      int temp, i;
5      for (i = 0; i < 10; i++) {
6          temp = n;
7          n = temp + 1;
8      }
9  }
10
11
12
13
14
15
```

Concurrent Program in C

```
16  void q() {  
17      int temp, i;  
18      for (i = 0; i < 10; i++) {  
19          temp = n;  
20          n = temp + 1;  
21      }  
22  }  
23  
24  void main() {  
25      cobegin { p(); q(); }  
26      cout << "The value of n is " << n << "\n";  
27  }
```


Concurrent Program in Ada

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Count is
3      N: Integer := 0;
4      pragma Volatile(N);
5
6      task type Count_Task;
7      task body Count_Task is
8          Temp: Integer;
9      begin
10         for I in 1..10 loop
11             Temp := N;
12             N := Temp + 1;
13         end loop;
14     end Count_Task;
15
```

Concurrent Program in Ada

```
16  begin
17      declare
18          P, Q: Count_Task;
19      begin
20          null;
21      end;
22      Put_Line("The value of N is " & Integer'Image(N));
23  end Count;
```

Concurrent Program in Java

```
1  class Count extends Thread {  
2      static volatile int n = 0;  
3  
4      public void run() {  
5          int temp;  
6          for (int i = 0; i < 10; i++) {  
7              temp = n;  
8              n = temp + 1;  
9          }  
10     }  
11  
12  
13  
14  
15
```

Concurrent Program in Java

```
16    public static void main(String[] args) {  
17        Count p = new Count();  
18        Count q = new Count();  
19        p.start ();  
20        q.start ();  
21        try {  
22            p.join ();  
23            q.join ();  
24        }  
25        catch (InterruptedException e) { }  
26        System.out.println ("The value of n is " + n);  
27    }  
28 }
```

Concurrent Program in Promela

```
1  #include "for.h"
2  #define TIMES 10
3  byte    n = 0;
4
5  proctype P() {
6      byte temp;
7      for (i,1, TIMES)
8          temp = n;
9          n = temp + 1
10     rof (i)
11 }
12
13
14
15
```

Concurrent Program in Promela

```
16  init {  
17      atomic {  
18          run P();  
19          run P()  
20      }  
21      (_nr_pr == 1);  
22      printf ("MSC: The value is %d\n", n)  
23  }
```

Frog Puzzle

[illegible]

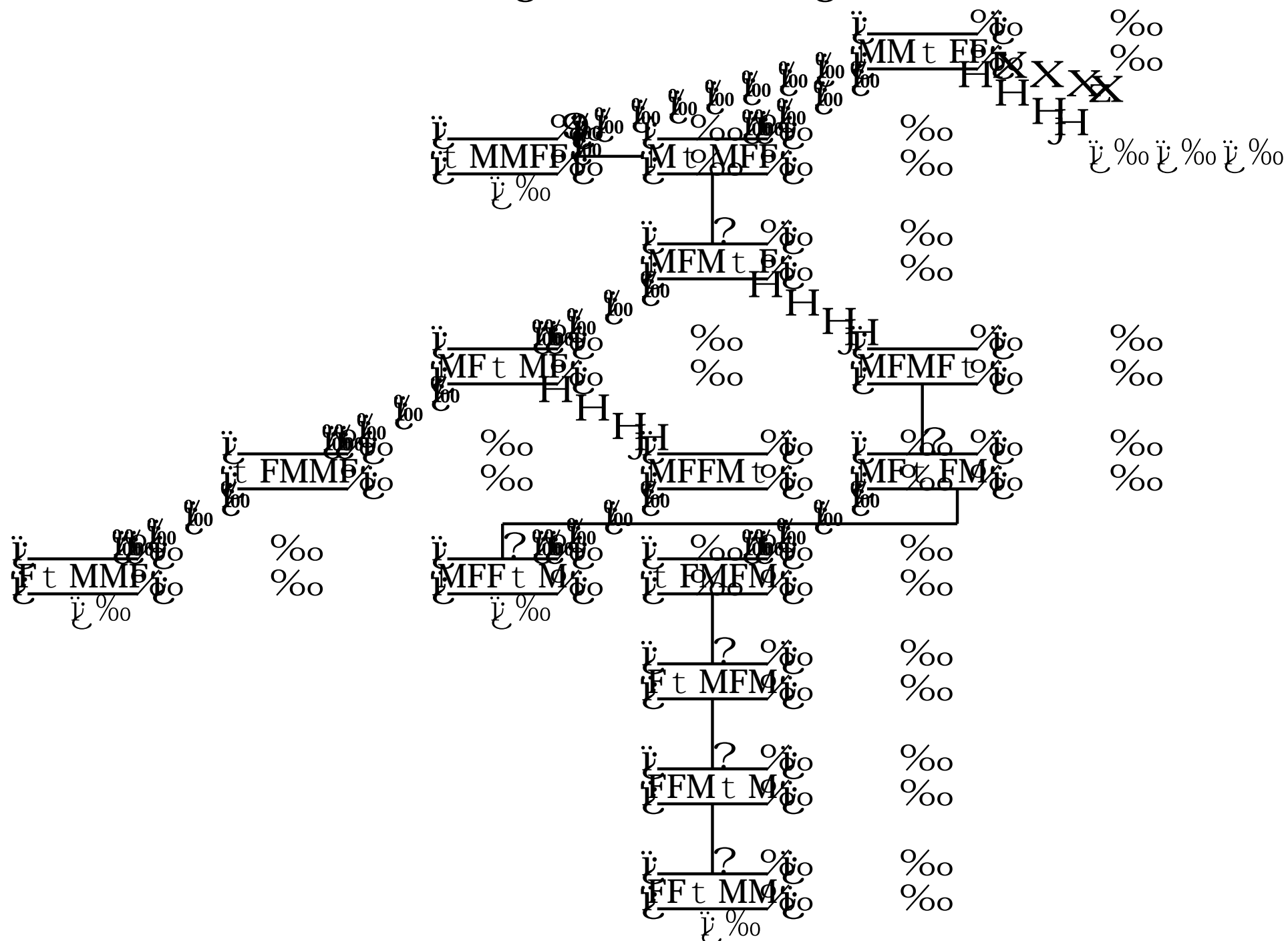
One Step of the Frog Puzzle

[illegible]

Final State of the Frog Puzzle

[illegible]

(Partial) State Diagram for the Frog Puzzle



Algorithm 2.10: Incrementing and decrementing

integer $n \geq 0$

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

q

integer $i \leftarrow 0$

p1: found $\leftarrow \text{false}$

p2: while not found

p3: $i \leftarrow i + 1$

p4: found $\leftarrow f(i) = 0$

integer $j \leftarrow 1$

q1: found $\leftarrow \text{false}$

q2: while not found

q3: $j \leftarrow j \oplus 1$

q4: found $\leftarrow f(j) = 0$

Algorithm 2.12: Zero B	
boolean found $\% \text{false}$	
p	q
integer i $\% 0$ p1: while not found p2: i $\% i + 1$ p3: found $\% f(i) = 0$	integer j $\% 1$ q1: while not found q2: j $\% j \% 1$ q3: found $\% f(j) = 0$

Algorithm 2.13: Zero C	
boolean found \leftarrow false	
p	q
integer i \leftarrow 0	integer j \leftarrow 1
p1: while not found	q1: while not found
p2: i \leftarrow i + 1	q2: j \leftarrow j + 1
p3: if f(i) = 0	q3: if f(j) = 0
p4: found \leftarrow true	q4: found \leftarrow true

Algorithm 2.14: Zero D

boolean found \leftarrow false
integer turn \leftarrow 1

p

q

integer i \leftarrow 0

integer j \leftarrow 1

p1: while not found

q1: while not found

p2: await turn = 1

q2: await turn = 2

 turn \leftarrow 2

 turn \leftarrow 1

p3: i \leftarrow i + 1

q3: j \leftarrow j + 1

p4: if f(i) = 0

q4: if f(j) = 0

p5: found \leftarrow true

q5: found \leftarrow true

Algorithm 2.15: Zero E

boolean found \leftarrow false

integer turn \leftarrow 1

p

integer i \leftarrow 0

p1: while not found

p2: await turn = 1

 turn \leftarrow 2

p3: i \leftarrow i + 1

p4: if f(i) = 0

p5: found \leftarrow true

p6: turn \leftarrow 2

q

integer j \leftarrow 1

q1: while not found

q2: await turn = 2

 turn \leftarrow 1

q3: j \leftarrow j + 1

q4: if f(j) = 0

q5: found \leftarrow true

q6: turn \leftarrow 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 $n \in \mathbb{Z}$

p

q

p1: while $n < 2$

p2: write(n)

q1: $n \leftarrow n + 1$

q2: $n \leftarrow n + 1$

Algorithm 2.18: Concurrent algorithm C

integer $n \in \mathbb{Z}$

p

q

p1: while $n < 1$

p2: $n \leftarrow n + 1$

q1: while $n \geq 0$

q2: $n \leftarrow n - 1$

Algorithm 2.19: Stop the loop B

integer $n \in \mathbb{N}$
 boolean $eg \in \{false, true\}$

p

p1: while $eg = false$
 p2: $n \leftarrow n + 1$
 p3:

q

q1: while $eg = false$
 q2: if $n = 0$
 q3: $eg \leftarrow true$

Algorithm 2.20: Stop the loop C

integer n $\leftarrow 0$
 boolean go \leftarrow false

p

p1: while $go = \text{false}$
 p2: $n \leftarrow 1$

q

q1: while $n = 0$ // Do nothing
 q2: $go \leftarrow \text{true}$

Algorithm 2.21: Welfare crook problem

integer array[0..N] a, b, c $\in \mathbb{O}..$ (as required)

integer i $\in \mathbb{O}$, j $\in \mathbb{O}$, k $\in \mathbb{O}$

loop

p1: if condition-1

p2: i $\leftarrow i + 1$

p3: else if condition-2

p4: j $\leftarrow j + 1$

p5: else if condition-3

p6: k $\leftarrow k + 1$

else exit loop

Algorithm 3.1: Critical section problem

global variables

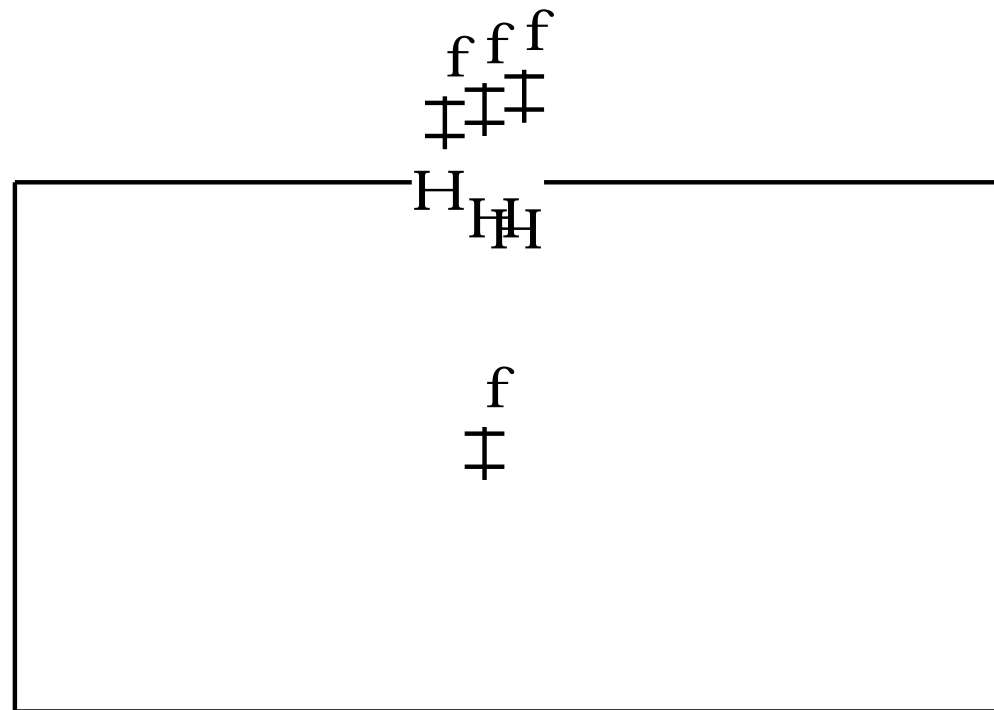
p

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 $\in \{1, 2\}$

p

loop forever

p1: non-critical section

p2: await turn = 1

p3: critical section

p4: turn $\leftarrow \text{turn} \% 2$

q

loop forever

q1: non-critical section

q2: await turn = 2

q3: critical section

q4: turn $\leftarrow \text{turn} \% 1$

Algorithm 3.3: History in a sequential algorithm
integer $a \in \mathbb{N}, b \in \mathbb{N}$
p1: Millions of statements p2: $a \leftarrow (a+b)*5$ p3: ...

Algorithm 3.4: History in a concurrent algorithm

integer $a \in \mathbb{N}, b \in \mathbb{N}$

p

q

p1: Millions of statements

p2: $a \leftarrow (a+b)*5$

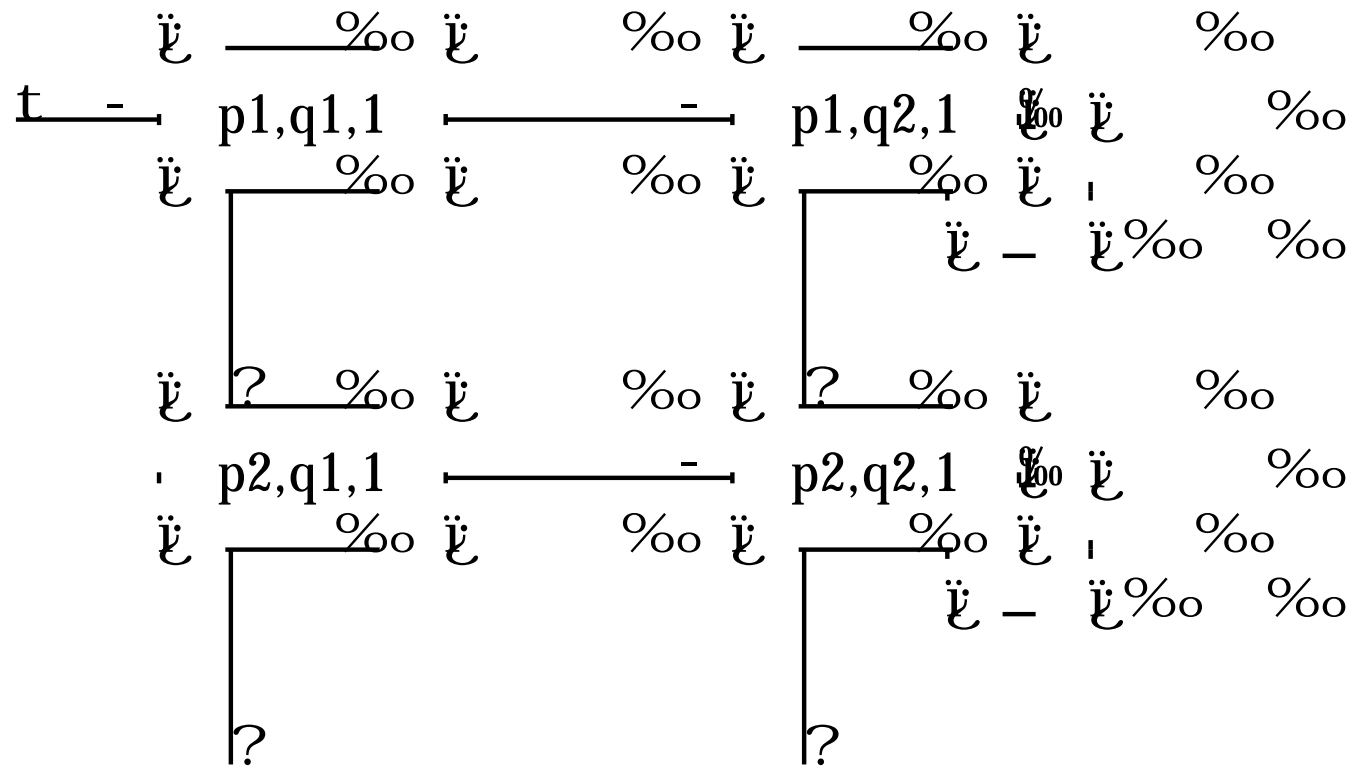
p3: ...

q1: Millions of statements

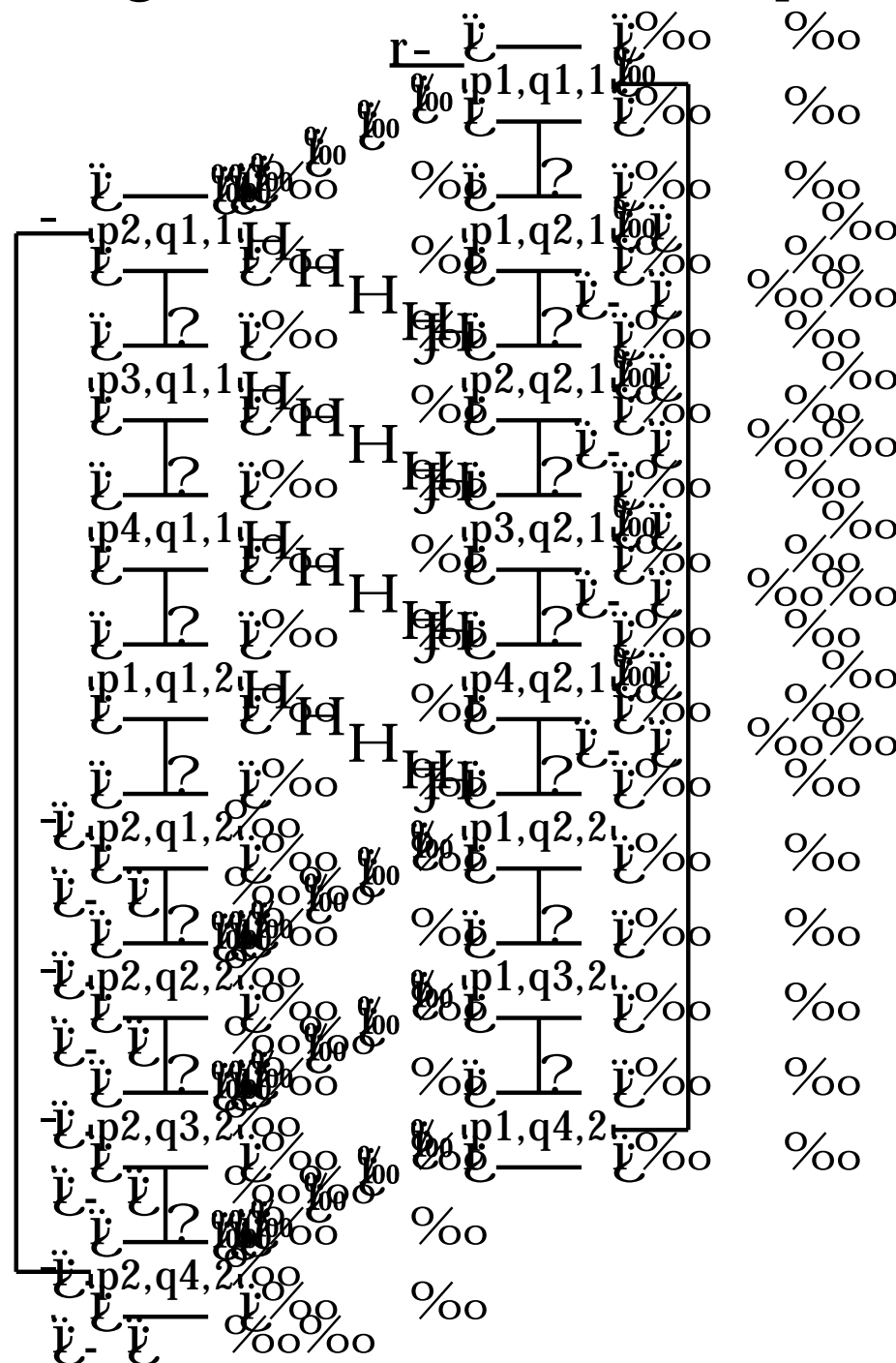
q2: $b \leftarrow (a+b)*5$

q3: ...

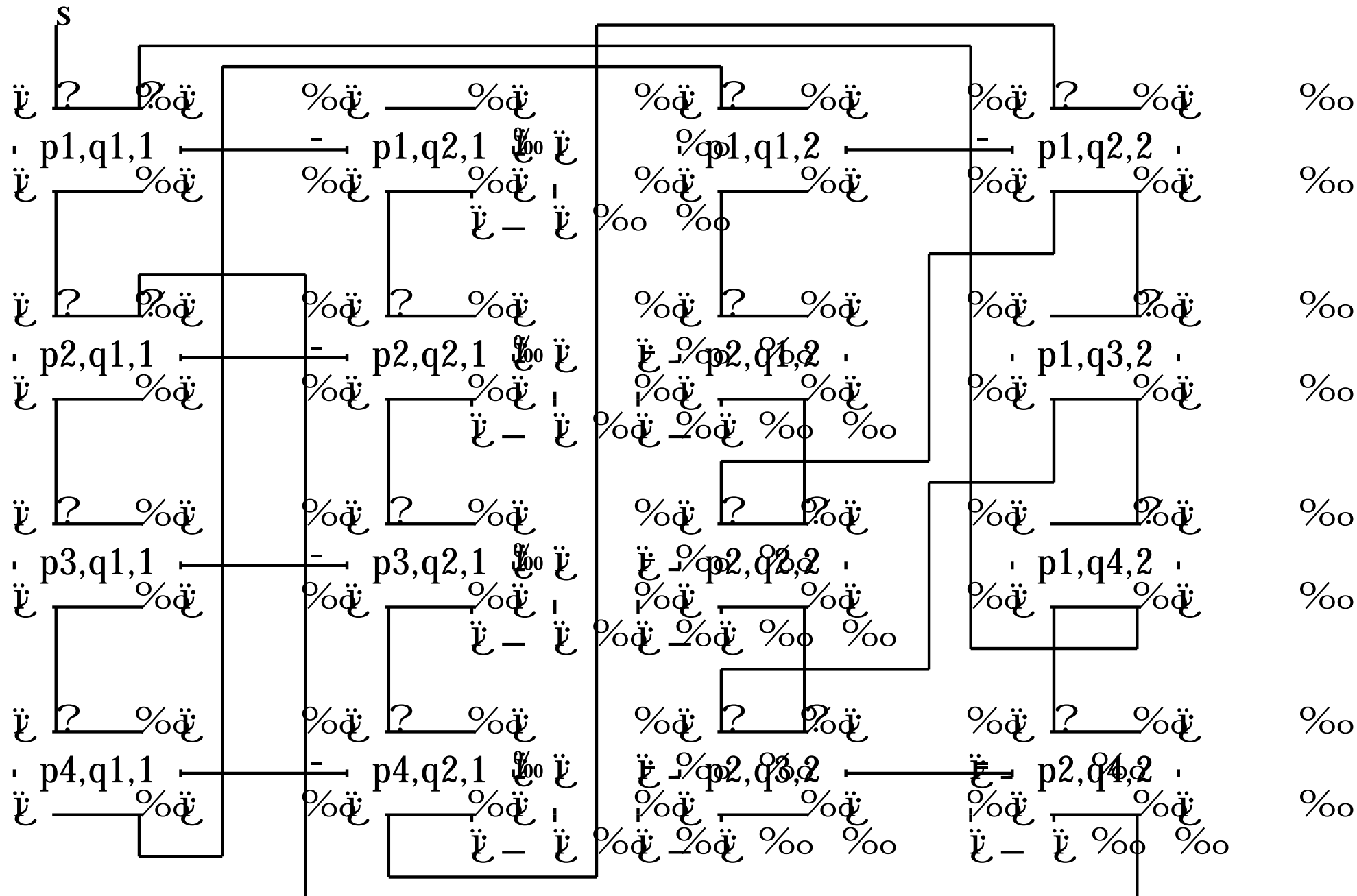
First States of the State Diagram



State Diagram for the First Attempt



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



Algorithm 3.5: First attempt (abbreviated)

integer turn $\in \{0, 1\}$

p

loop forever

p1: await turn = 1

p2: turn $\leftarrow 1$

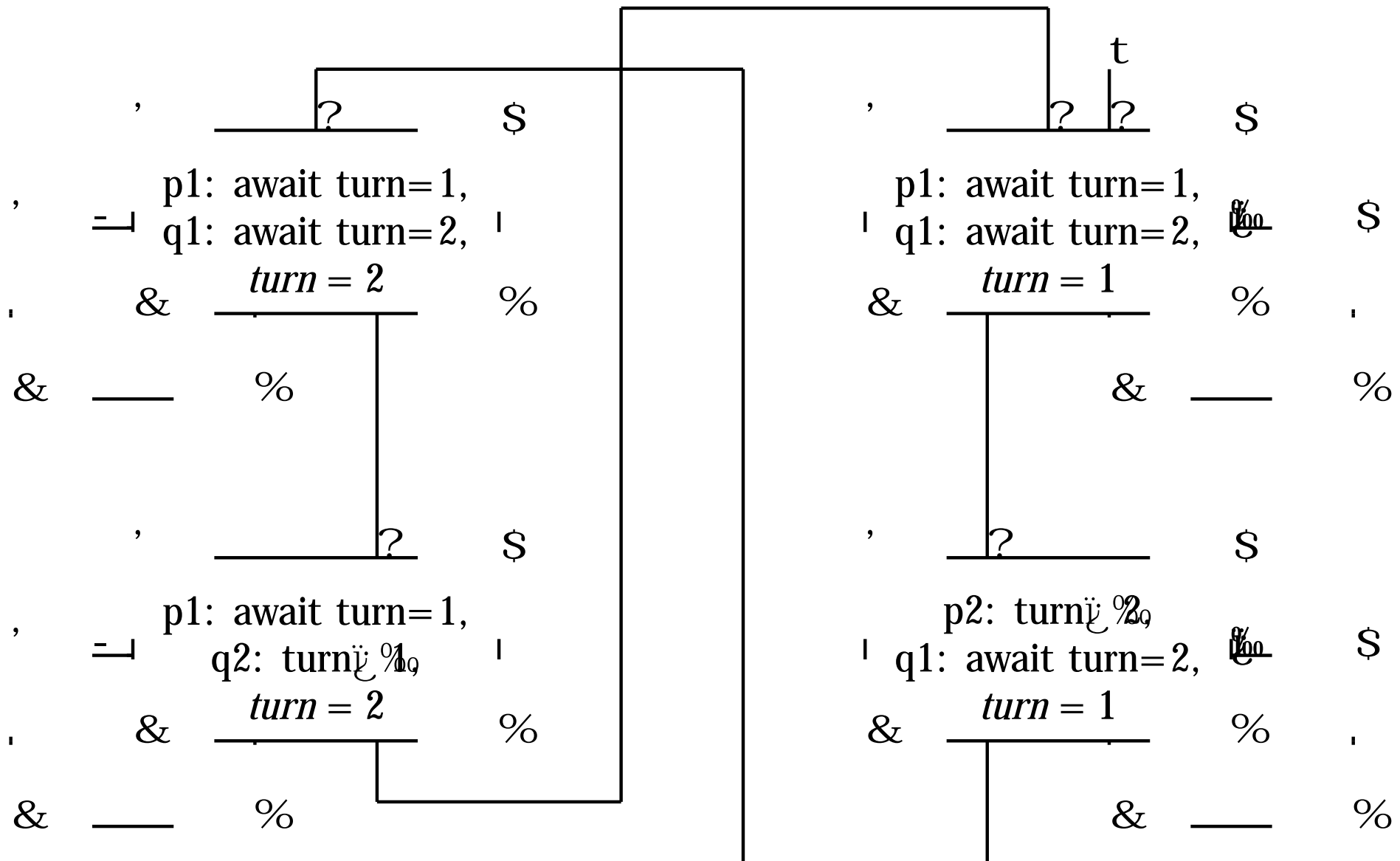
q

loop forever

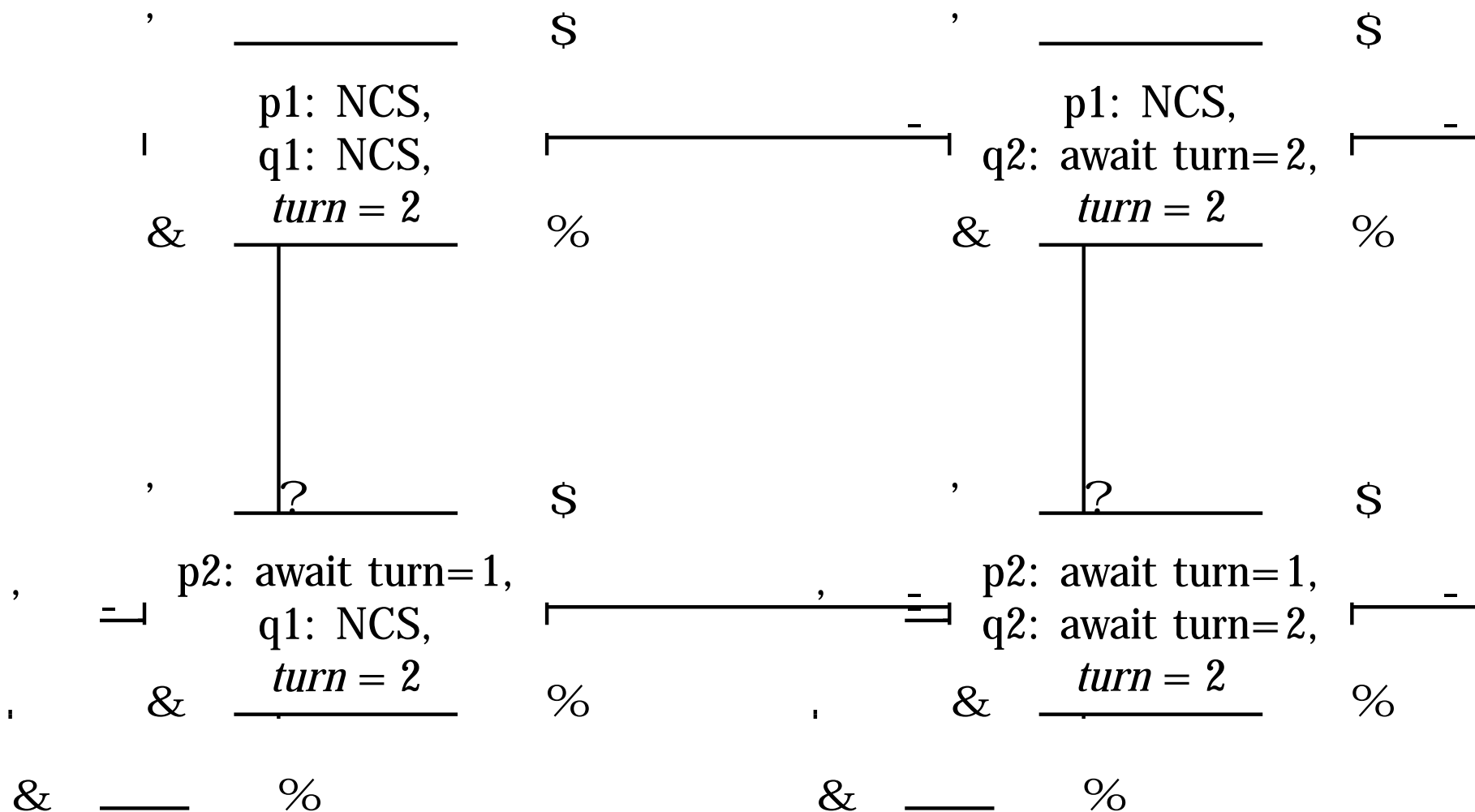
q1: await turn = 2

q2: turn $\leftarrow 0$

State Diagram for the Abbreviated First Attempt



Fragment of the State Diagram for the First Attempt



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

Algorithm 3.7: Second attempt (abbreviated)

boolean wantp $\dot{\leftarrow}$ false, wantq $\dot{\leftarrow}$ false

p

loop forever

p1: await wantq = false

p2: wantp $\dot{\leftarrow}$ true

p3: wantp $\dot{\leftarrow}$ false

q

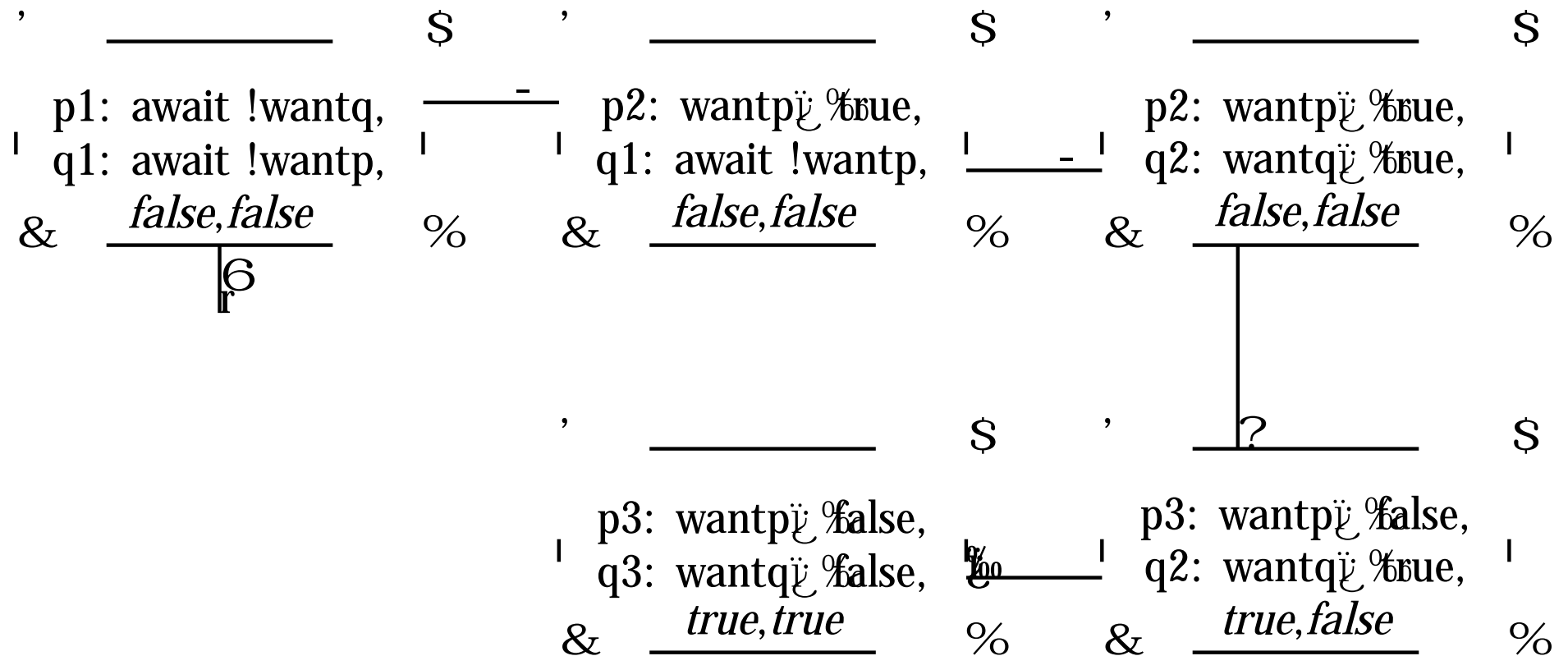
loop forever

q1: await wantp = false

q2: wantq $\dot{\leftarrow}$ true

q3: wantq $\dot{\leftarrow}$ false

Fragment of the State Diagram for the Second Attempt



Scenario Showing that Mutual Exclusion Does Not Hold

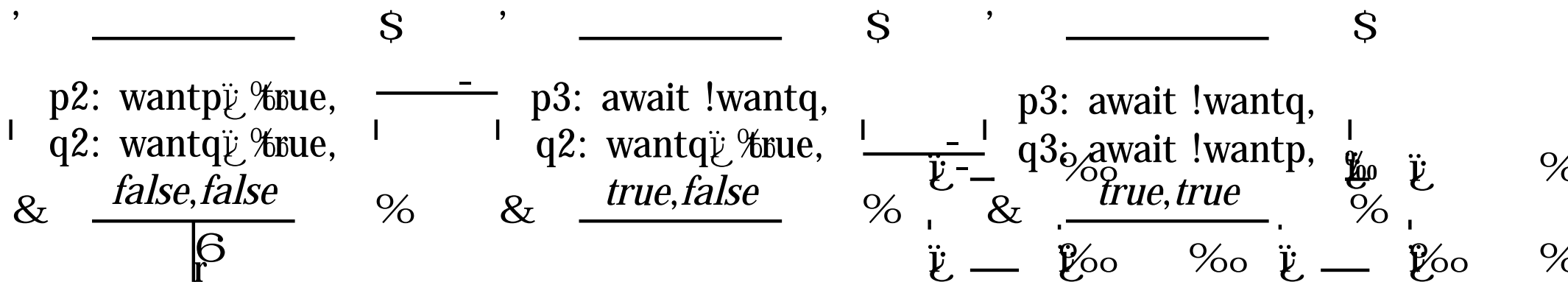
Process p	Process q	wantp	wantq
p1: await wantq=false	q1: await wantp=false	<i>false</i>	<i>false</i>
p2: wantp \rightarrow true	q1: await wantp=false	<i>false</i>	<i>false</i>
p2: wantp \rightarrow true	q2: wantq \rightarrow true	<i>false</i>	<i>false</i>
p3: wantp \rightarrow false	q3: wantq \rightarrow true	<i>true</i>	<i>false</i>
p3: wantp \rightarrow false	q3: wantq \rightarrow false	<i>true</i>	<i>true</i>

Algorithm 3.8: Third attempt	
boolean wantp $\dot{\leftarrow}$ %false, wantq $\dot{\leftarrow}$ %false	
p	q
loop forever p1: non-critical section p2: wantp $\dot{\leftarrow}$ %true p3: await wantq = false p4: critical section p5: wantp $\dot{\leftarrow}$ %false	loop forever q1: non-critical section q2: wantq $\dot{\leftarrow}$ %true q3: await wantp = false q4: critical section q5: wantq $\dot{\leftarrow}$ %false

Scenario Showing Deadlock in the Third Attempt

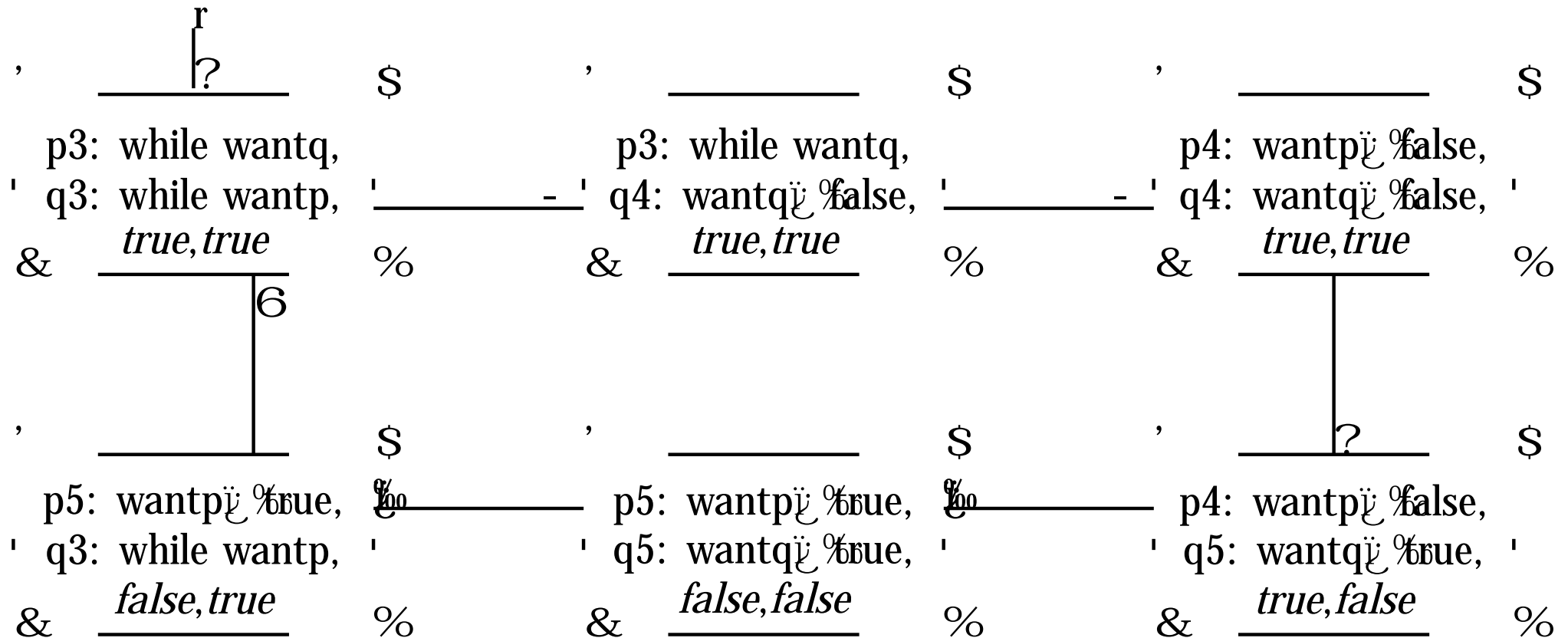
Process p	Process q	wantp	wantq
p1: non-critical section	q1: non-critical section	<i>false</i>	<i>false</i>
p2: wantp = <i>true</i>	q1: non-critical section	<i>false</i>	<i>false</i>
p2: wantp = <i>true</i>	q2: wantq = <i>true</i>	<i>false</i>	<i>false</i>
p3: await wantq = <i>false</i>	q2: wantq = <i>true</i>	<i>true</i>	<i>false</i>
p3: await wantq = <i>false</i>	q3: await wantp = <i>false</i>	<i>true</i>	<i>true</i>

Fragment of the State Diagram Showing Deadlock



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

Cycle in the State Diagram for the Fourth Attempt



Algorithm 3.10: Dekker's algorithm	
boolean wantp \checkmark %false, wantq \checkmark %false integer turn \checkmark %1	
p	q
loop forever p1: non-critical section p2: wantp \checkmark %true p3: while wantq p4: if turn = 2 p5: wantp \checkmark %false p6: await turn = 1 p7: wantp \checkmark %true p8: critical section p9: turn \checkmark %2 p10: wantp \checkmark %false	loop forever q1: non-critical section q2: wantq \checkmark %true q3: while wantp q4: if turn = 1 q5: wantq \checkmark %false q6: await turn = 2 q7: wantq \checkmark %true q8: critical section q9: turn \checkmark %1 q10: wantq \checkmark %false

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

integer common $\text{t} \in \{0, 1\}$

p

q

integer local1

loop forever

p1: non-critical section

repeat

p2: test-and-set(
common, local1)

p3: until local1 = 0

p4: critical section

p5: common $\text{t} \in \{0, 1\}$

integer local2

loop forever

q1: non-critical section

repeat

q2: test-and-set(
common, local2)

q3: until local2 = 0

q4: critical section

q5: common $\text{t} \in \{0, 1\}$

Algorithm 3.12: Critical section problem with exchange

integer common $\in \{0, 1\}$

p

integer local1 $\in \{0, 1\}$

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 $\in \{0, 1\}$

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 \leftarrow false, wantq \leftarrow false
integer last \leftarrow 1

p

q

loop forever

loop forever

p1: non-critical section

q1: non-critical section

p2: wantp \leftarrow true

q2: wantq \leftarrow true

p3: last \leftarrow 1

q3: last \leftarrow 2

p4: await wantq = false or
last = 2

q4: await wantp = false or
last = 1

p5: critical section

q5: critical section

p6: wantp \leftarrow false

q6: wantq \leftarrow false

Algorithm 3.14: Manna-Pnueli algorithm

integer wantp $\in \mathbb{N}$, wantq $\in \mathbb{N}$

p

q

loop forever

loop forever

p1: non-critical section

q1: non-critical section

p2: if wantq = 0
 wantp $\in \mathbb{N}$ 1

q2: if wantp = 0
 wantq $\in \mathbb{N}$ 1

 else wantp $\in \mathbb{N}$ 1

 else wantq $\in \mathbb{N}$ 1

p3: await wantq \in wantp

q3: await wantp \in wantq

p4: critical section

q4: critical section

p5: wantp $\in \mathbb{N}$ 0

q5: wantq $\in \mathbb{N}$ 0

Algorithm 3.15: Doran-Thomas algorithm

boolean wantp $\dot{\in} \%$ false, wantq $\dot{\in} \%$ false
integer turn $\dot{\in} \%$ 1

p

q

loop forever

loop forever

p1: non-critical section

q1: non-critical section

p2: wantp $\dot{\in} \%$ true

q2: wantq $\dot{\in} \%$ true

p3: if wantq

q3: if wantp

p4: if turn = 2

q4: if turn = 1

p5: wantp $\dot{\in} \%$ false

q5: wantq $\dot{\in} \%$ false

p6: await turn = 1

q6: await turn = 2

p7: wantp $\dot{\in} \%$ true

q7: wantq $\dot{\in} \%$ true

p8: await wantq = false

q8: await wantp = false

p9: critical section

q9: critical section

p10: wantp $\dot{\in} \%$ false

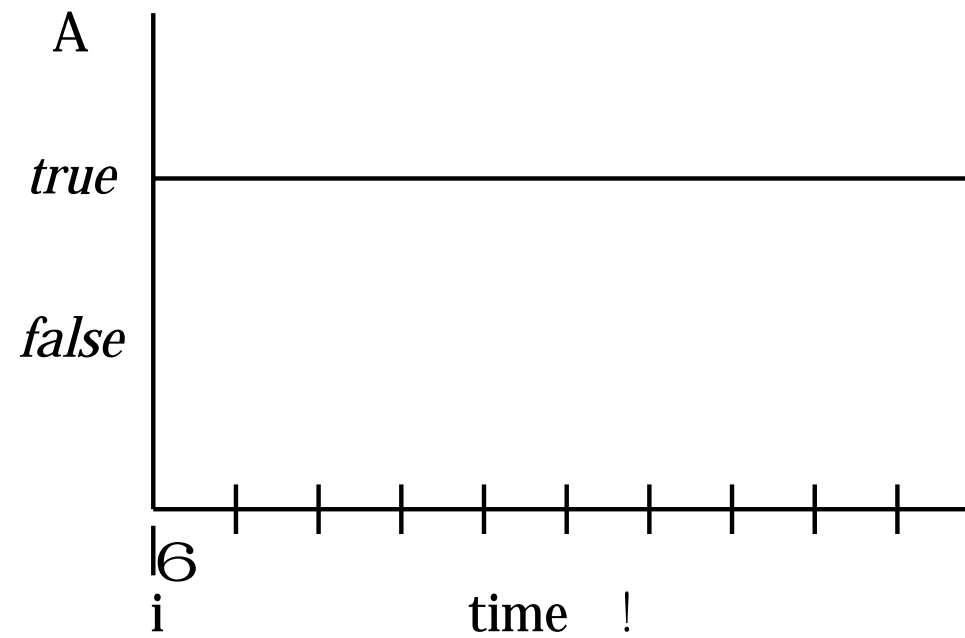
q10: wantq $\dot{\in} \%$ false

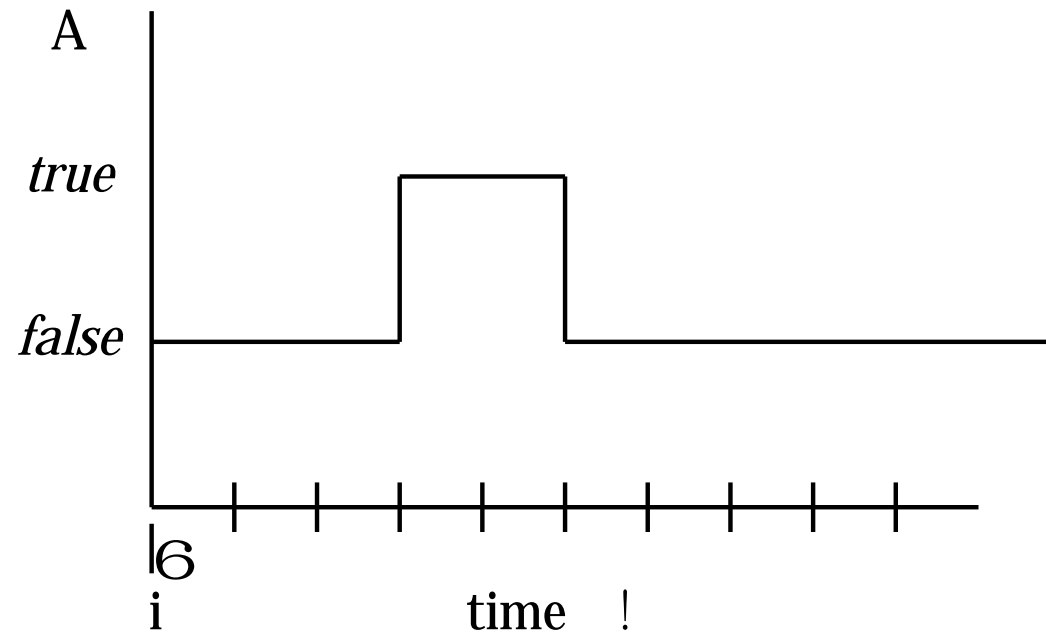
p11: turn $\dot{\in} \%$ 2

q11: turn $\dot{\in} \%$ 1

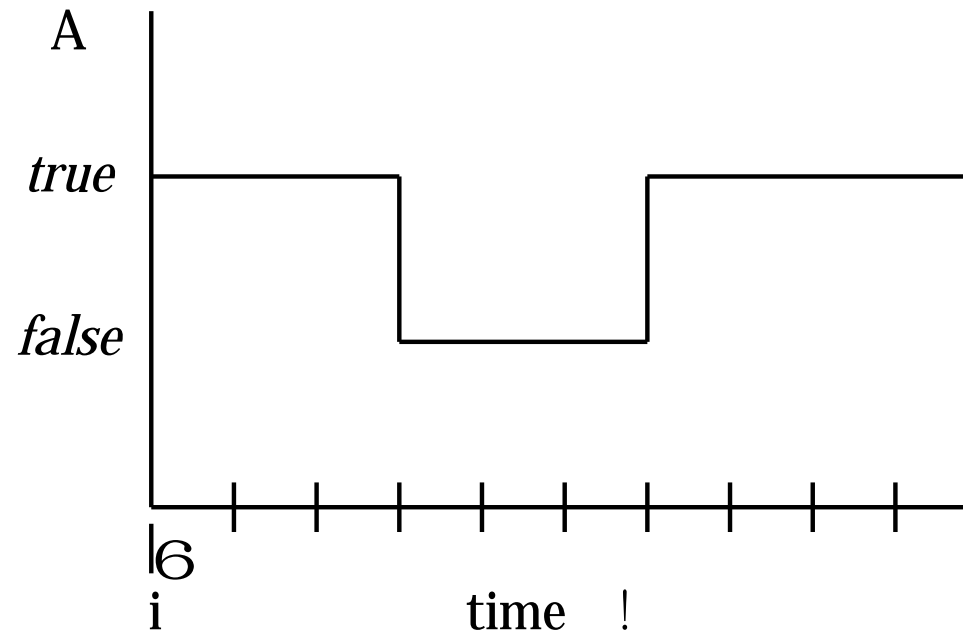
Algorithm 4.1: Third attempt	
boolean wantp \leftarrow false, wantq \leftarrow false	
p	q
loop forever p1: non-critical section p2: wantp \leftarrow true p3: await wantq = false p4: critical section p5: wantp \leftarrow false	loop forever q1: non-critical section q2: wantq \leftarrow true q3: await wantp = false q4: critical section q5: wantq \leftarrow false

$2A$

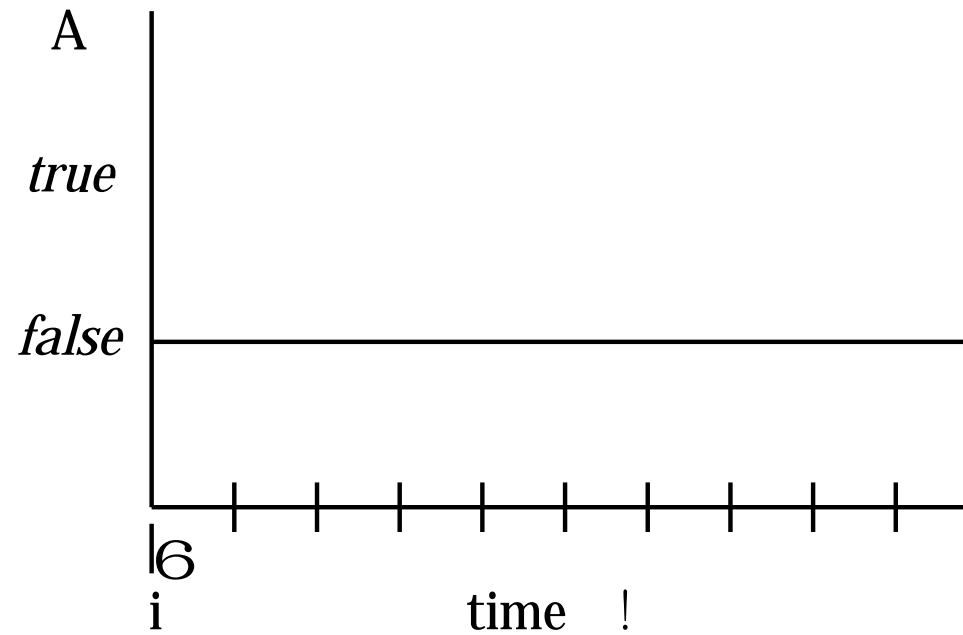


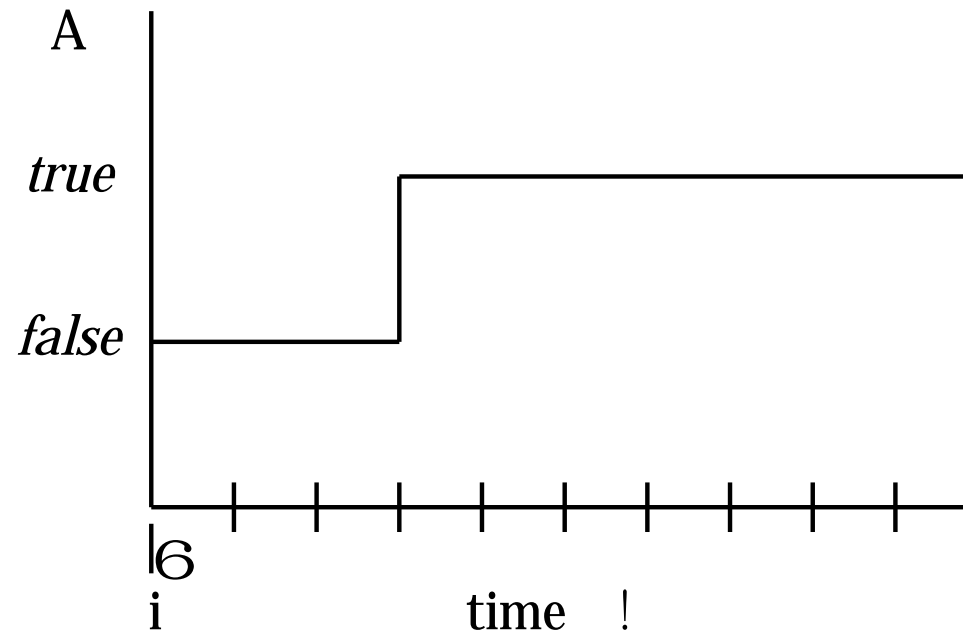


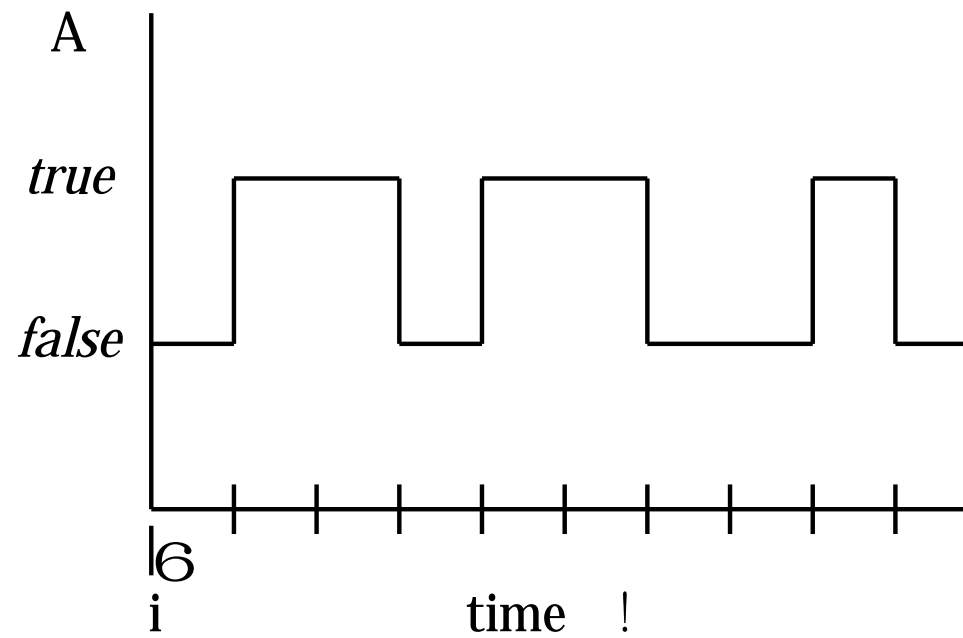
Duality: $\neg 2A$



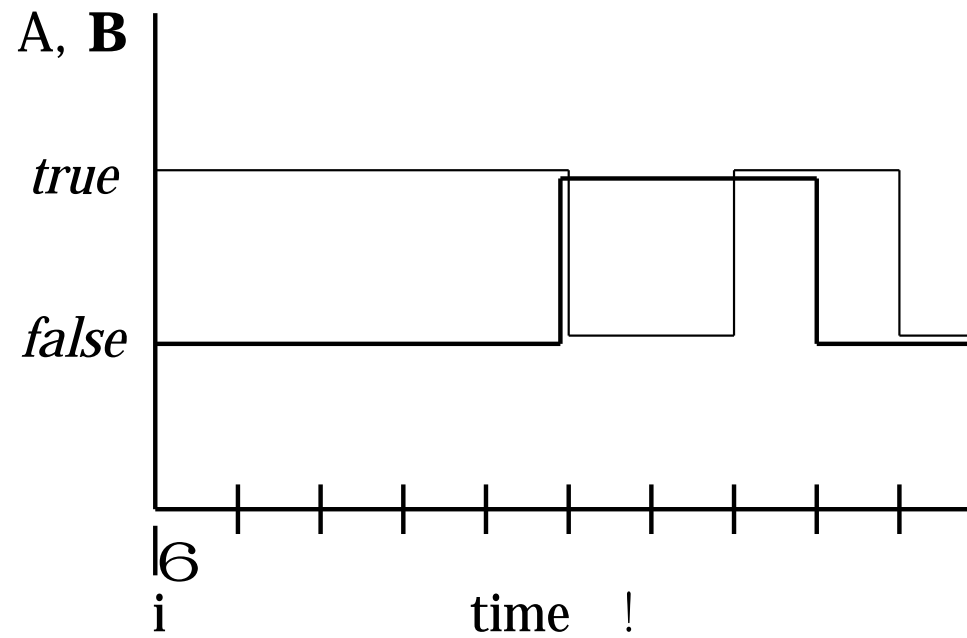
Duality: $\exists A$



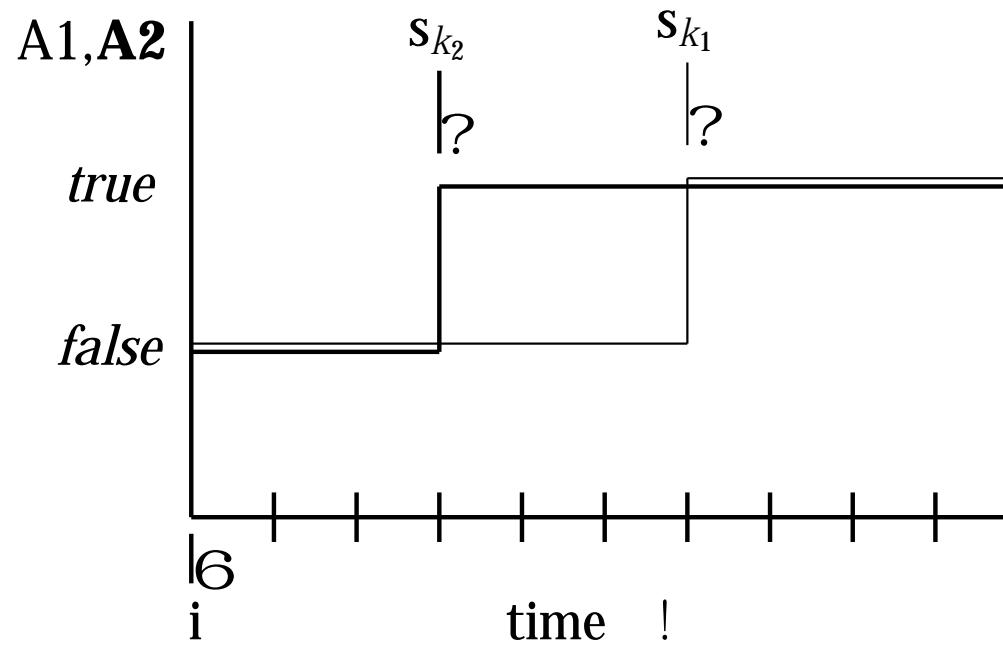




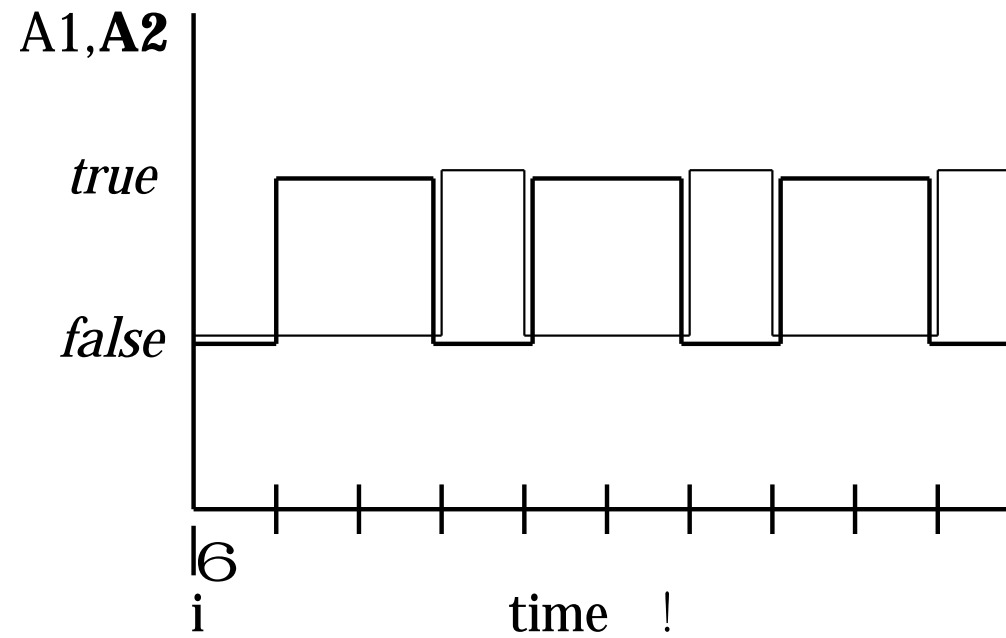
$A \cup B$



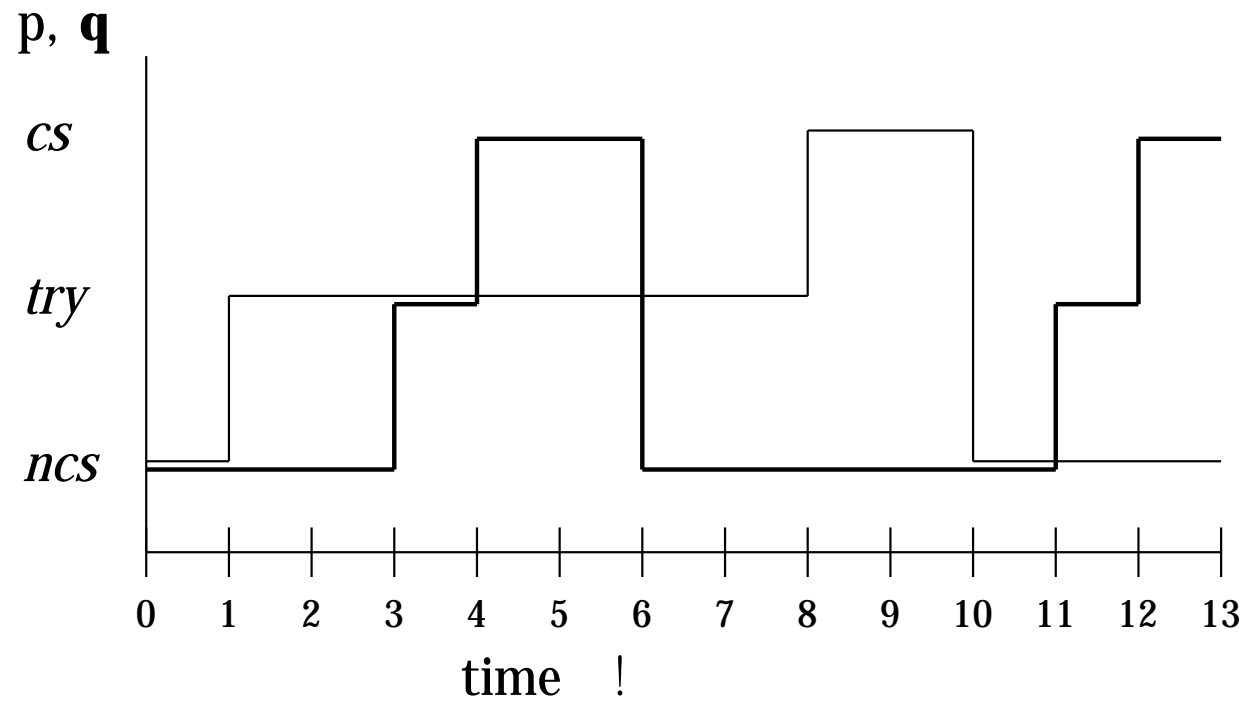
$$32A1 \wedge 32A2$$



$$23A1 \wedge 23A2$$



Overtaking: $try_p ! (: cs_q) W (cs_q) W (: cs_q) W (cs_p)$



Algorithm 4.2: Dekker's algorithm	
boolean wantp \checkmark %false, wantq \checkmark %false integer turn \checkmark %1	
p	q
loop forever p1: non-critical section p2: wantp \checkmark %true p3: while wantq p4: if turn = 2 p5: wantp \checkmark %false p6: await turn = 1 p7: wantp \checkmark %true p8: critical section p9: turn \checkmark %2 p10: wantp \checkmark %false	loop forever q1: non-critical section q2: wantq \checkmark %true q3: while wantp q4: if turn = 1 q5: wantq \checkmark %false q6: await turn = 2 q7: wantq \checkmark %true q8: critical section q9: turn \checkmark %1 q10: wantq \checkmark %false

Dekker's Algorithm in Promela

```
1  bool wantp = false, wantq = false; byte turn = 1;
2  active proctype p() {
3      do :: wantp = true;
4          do :: !wantq ; % break;
5              :: else ; %
6                  if :: (turn == 1)
7                      :: (turn == 2) ; %
8                          wantp = false; (turn == 1); wantp = true
9                      ; %
10         od;
11         printf ("MSC: p in CS\n") ;
12         turn = 2; wantp = false
13     od
14 }
```

Specifying Correctness in Promela

```
1  byte critical    = 0;
2
3  bool PinCS = false;
4  #define def no_nostarve PinCS  /ï %LTL claim < > nostarve ÿ %00
5
6  active proctype p() {
7      do ::
8          /ï %preprotocol ÿ %00
9          critical ++;
10         assert(critical <= 1);
11         PinCS = true;
12         critical ÿ %0;ÿ %00
13         /ï %postprotocol ÿ %00
14     od
15 }
```

LTL Translation to Never Claims

```

1  never {      /j %(< > nostarve) j %00
2  accept_init:
3  T0_init:
4      if
5      :: (! ((nostarve))) j %0 goto T0_init
6      j %0
7  }
8
9
10
11
12
13
14
15

```

LTL Translation to Never Claims

```
16  never {      /j %([< > nostarve)  j %o
17  T0_init:
18      if
19      :: (! ((nostarve)))  j %o goto accept_S4
20      :: (1)  j %o goto T0_init
21      %o
22  accept_S4:
23      if
24      :: (! ((nostarve)))  j %o goto accept_S4
25      %o
26  }
```


Algorithm 5.1: Bakery algorithm (two processes)

integer $np \in \mathbb{N}, nq \in \mathbb{N}$

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 $\in [0, \dots, 0]$

loop forever

p1: non-critical section

p2: $\text{number}[i] \leftarrow 1 + \max(\text{number})$

p3: for all other processes j

p4: await ($\text{number}[j] = 0$) or ($\text{number}[i] \leq \text{number}[j]$)

p5: critical section

p6: $\text{number}[i] \leftarrow 0$

Algorithm 5.3: Bakery algorithm without atomic assignment

boolean array[1..n] choosing \in {false, ..., false}

integer array[1..n] number \in {0, ..., 0}

loop forever

p1: non-critical section

p2: choosing[i] \leftarrow true

p3: number[i] \leftarrow 1 + max(number)

p4: choosing[i] \leftarrow false

p5: for all other processes j

p6: await choosing[j] = false

p7: await (number[j] = 0) or (number[i] \leq number[j])

p8: critical section

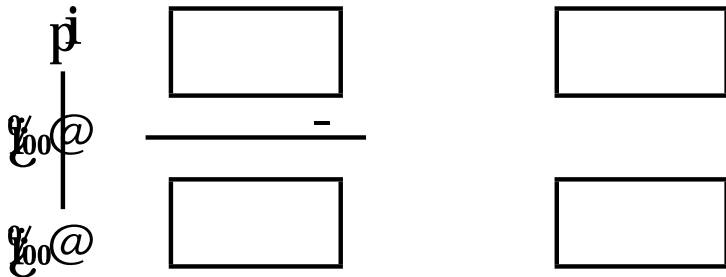
p9: number[i] \leftarrow 0

Algorithm 5.4: Fast algorithm for two processes (outline)

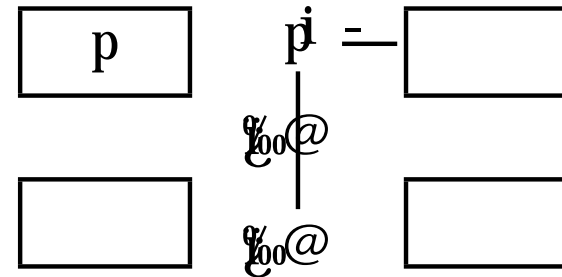
integer gate1 $\in \{0, 1\}$, gate2 $\in \{0, 1\}$

p	q
loop forever non-critical section p1: gate1 $\in \{0, 1\}$ p2: if gate2 $\in \{0, 1\}$ goto p1 p3: gate2 $\in \{0, 1\}$ p4: if gate1 $\in \{0, 1\}$ p5: if gate2 $\in \{0, 1\}$ goto p1 critical section p6: gate2 $\in \{0, 1\}$	loop forever non-critical section q1: gate1 $\in \{0, 1\}$ q2: if gate2 $\in \{0, 1\}$ goto q1 q3: gate2 $\in \{0, 1\}$ q4: if gate1 $\in \{0, 1\}$ q5: if gate2 $\in \{0, 1\}$ goto q1 critical section q6: gate2 $\in \{0, 1\}$

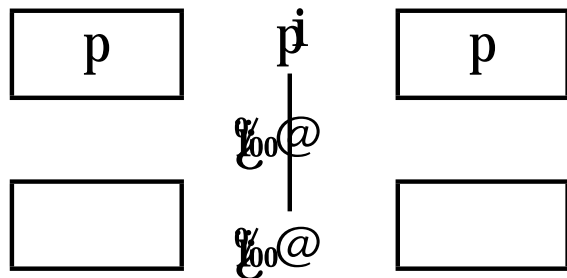
Fast Algorithm - No Contention (1)



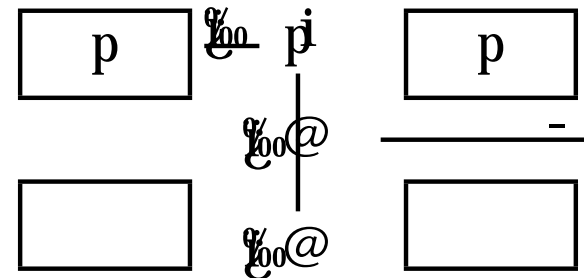
(a)



(b)

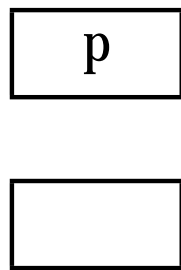


(c)

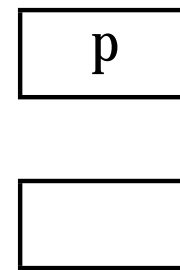
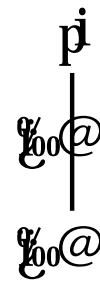
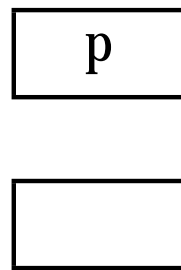


(d)

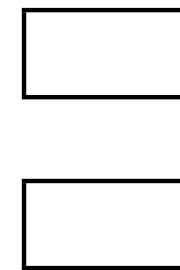
Fast Algorithm - No Contention (2)



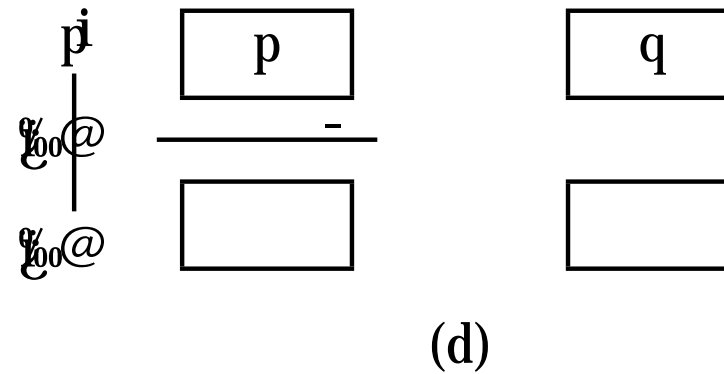
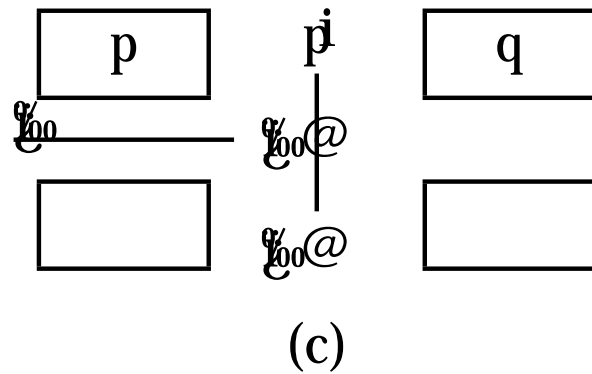
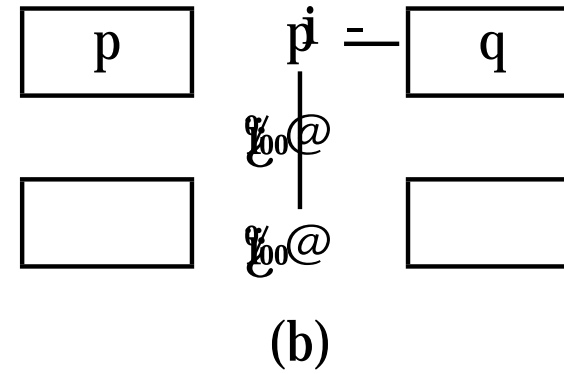
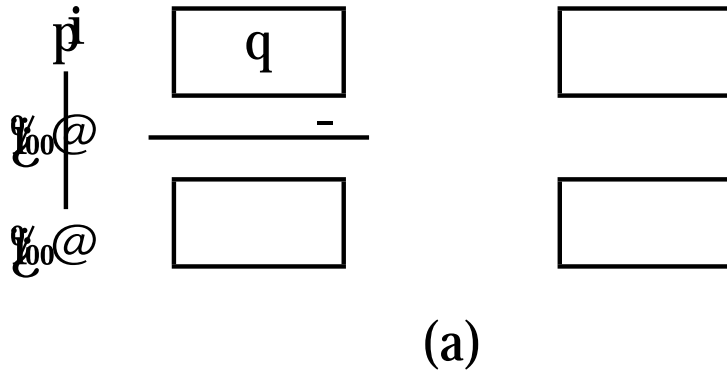
(e)



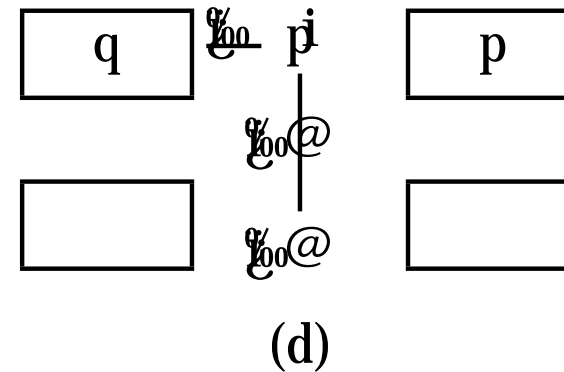
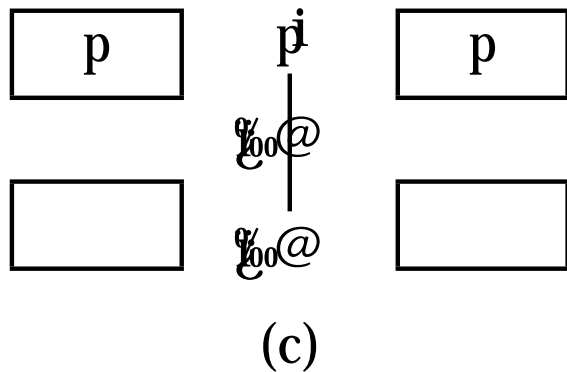
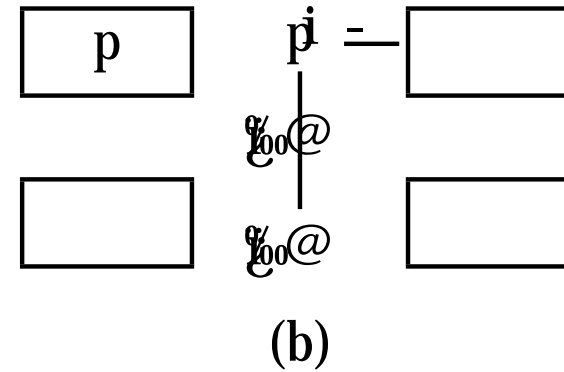
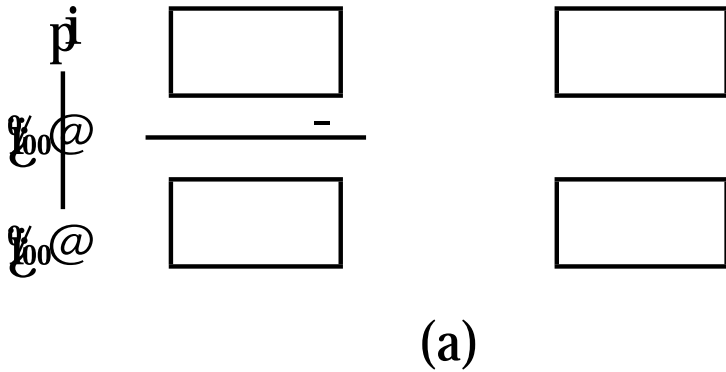
(f)



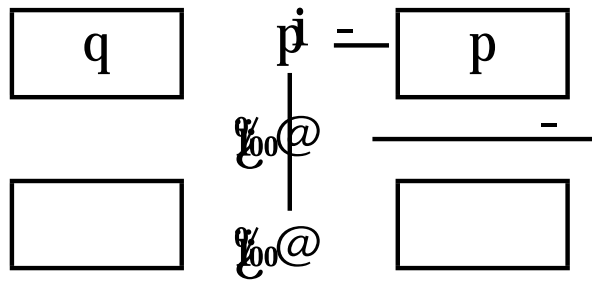
Fast Algorithm - Contention At Gate 2



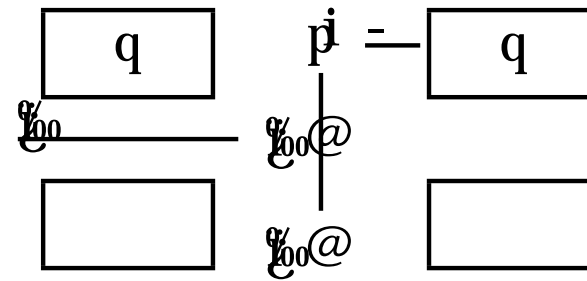
Fast Algorithm - Contention At Gate 1 (1)



Fast Algorithm - Contention At Gate 1 (2)



(e)



(f)

Algorithm 5.5: Fast algorithm for two processes (outline)

integer gate1 $\in \{0, 1\}$, gate2 $\in \{0, 1\}$

p	q
loop forever non-critical section p1: gate1 $\in \{0, 1\}$ p2: if gate2 $\in \{0, 1\}$ goto p1 p3: gate2 $\in \{0, 1\}$ p4: if gate1 $\in \{0, 1\}$ p5: if gate2 $\in \{0, 1\}$ goto p1 critical section p6: gate2 $\in \{0, 1\}$	loop forever non-critical section q1: gate1 $\in \{0, 1\}$ q2: if gate2 $\in \{0, 1\}$ goto q1 q3: gate2 $\in \{0, 1\}$ q4: if gate1 $\in \{0, 1\}$ q5: if gate2 $\in \{0, 1\}$ goto q1 critical section q6: gate2 $\in \{0, 1\}$

Algorithm 5.6: Fast algorithm for two processes

integer gate1 $\in \{0, 1\}$, gate2 $\in \{0, 1\}$

boolean wantp $\in \{\text{false}, \text{true}\}$, wantq $\in \{\text{false}, \text{true}\}$

p	q
<p>p1: gate1 $\in \{0, 1\}$ wantp $\in \{\text{false}, \text{true}\}$</p> <p>p2: if gate2 $\in \{0, 1\}$ wantp $\in \{\text{false}, \text{true}\}$ goto p1</p> <p>p3: gate2 $\in \{0, 1\}$</p> <p>p4: if gate1 $\in \{0, 1\}$ wantp $\in \{\text{false}, \text{true}\}$ await wantq = false</p> <p>p5: if gate2 $\in \{0, 1\}$ goto p1 else wantp $\in \{\text{false}, \text{true}\}$ critical section</p> <p>p6: gate2 $\in \{0, 1\}$ wantp $\in \{\text{false}, \text{true}\}$</p>	<p>q1: gate1 $\in \{0, 1\}$ wantq $\in \{\text{false}, \text{true}\}$</p> <p>q2: if gate2 $\in \{0, 1\}$ wantq $\in \{\text{false}, \text{true}\}$ goto q1</p> <p>q3: gate2 $\in \{0, 1\}$</p> <p>q4: if gate1 $\in \{0, 1\}$ wantq $\in \{\text{false}, \text{true}\}$ await wantp = false</p> <p>q5: if gate2 $\in \{0, 1\}$ goto q1 else wantq $\in \{\text{false}, \text{true}\}$ critical section</p> <p>q6: gate2 $\in \{0, 1\}$ wantq $\in \{\text{false}, \text{true}\}$</p>

Algorithm 5.7: Fisher's algorithm

integer gate $\in \{0, 1\}$

loop forever

non-critical section

loop

p1: await gate = 0

p2: gate $\leftarrow i$

p3: delay

p4: until gate = i

critical section

p5: gate $\leftarrow 0$

Algorithm 5.8: Lamport's one-bit algorithm

boolean array[1..n] want $\%[false, \dots, false]$

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 $j > i$

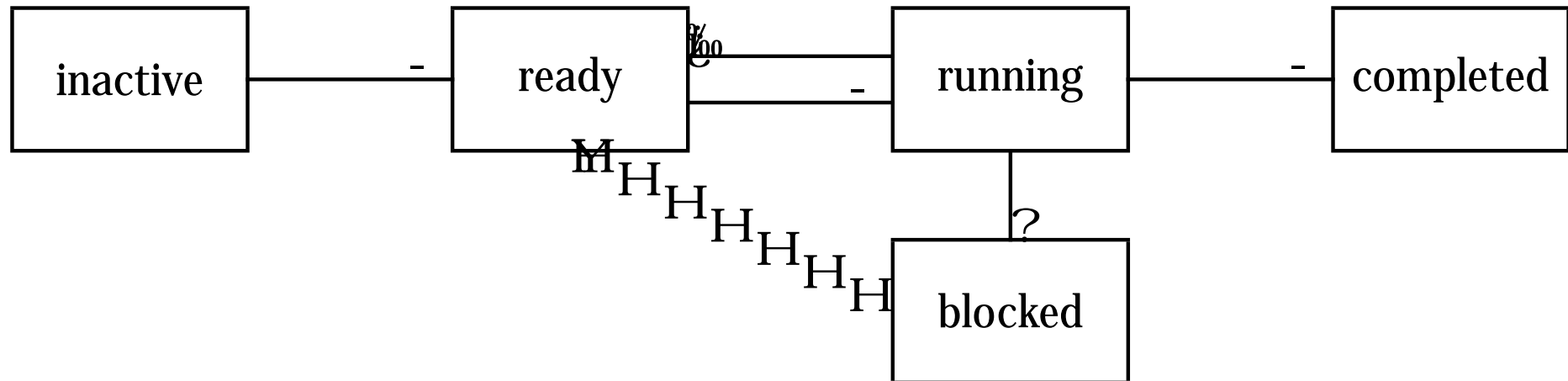
p7: await not want[j]

critical section

p8: want[i] $\%false$

Algorithm 5.9: Manna-Pnueli central server algorithm	
integer request $\in \mathbb{N}$, respond $\in \mathbb{N}$	
client process i	
	loop forever non-critical section p1: while respond $\neq i$ p2: request $\in \mathbb{N}$ critical section p3: respond $\in \mathbb{N}$
server process	
	loop forever p4: await request $\neq 0$ p5: respond $\in \mathbb{N}$ request p6: await respond = 0 p7: request $\in \mathbb{N}$

State Changes of a Process



Algorithm 6.1: Critical section with semaphores (two processes)

binary semaphore $S \in \{0, 1\}$

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 $\{0,1\}$

p

loop forever

p1: wait(S)

p2: signal(S)

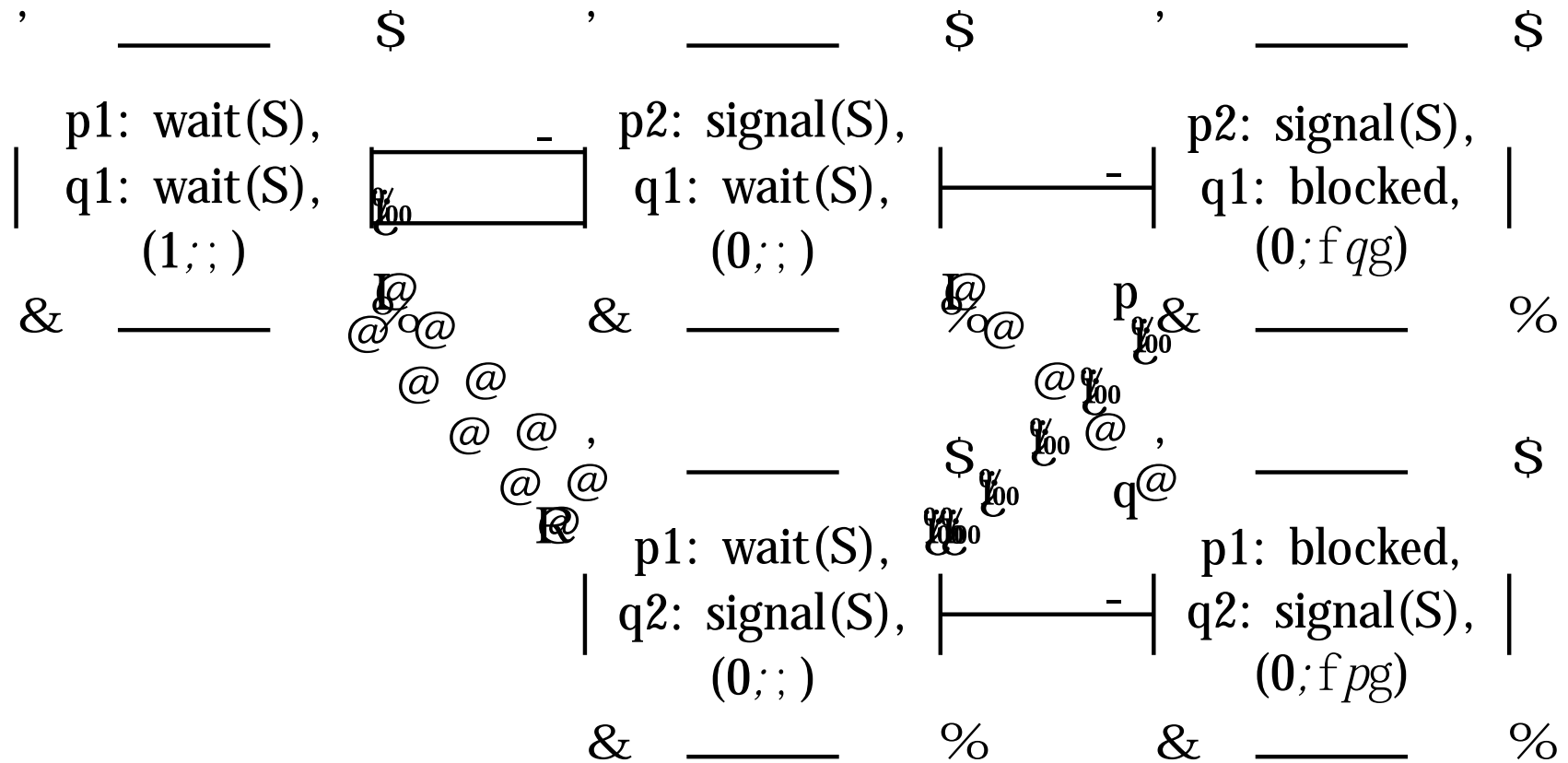
q

loop forever

q1: wait(S)

q2: signal(S)

State Diagram for the Semaphore Solution



Algorithm 6.3: Critical section with semaphores (N proc.)
binary semaphore $S \in \{0, 1\}$
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 $S \in \{0, 1\}$
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;;)
2	p2: signal(S)	q1: wait(S)	r1: wait(S)	(0;;)
3	p2: signal(S)	q1: blocked	r1: wait(S)	(0;f qg)
4	p1: signal(S)	q1: blocked	r1: blocked	(0;f q; rg)
5	p1: wait(S)	q1: blocked	r2: signal(S)	(0;f qg)
6	p1: blocked	q1: blocked	r2: signal(S)	(0;f p; qg)
7	p2: signal(S)	q1: blocked	r1: wait(S)	(0;f qg)

Algorithm 6.5: Mergesort

integer array A

binary semaphore S1 $\text{init}(0; ;)$

binary semaphore S2 $\text{init}(0; ;)$

sort1

p1: sort 1st half of A

p2: signal(S1)

p3:

sort2

q1: sort 2nd half of A

q2: signal(S2)

q3:

merge

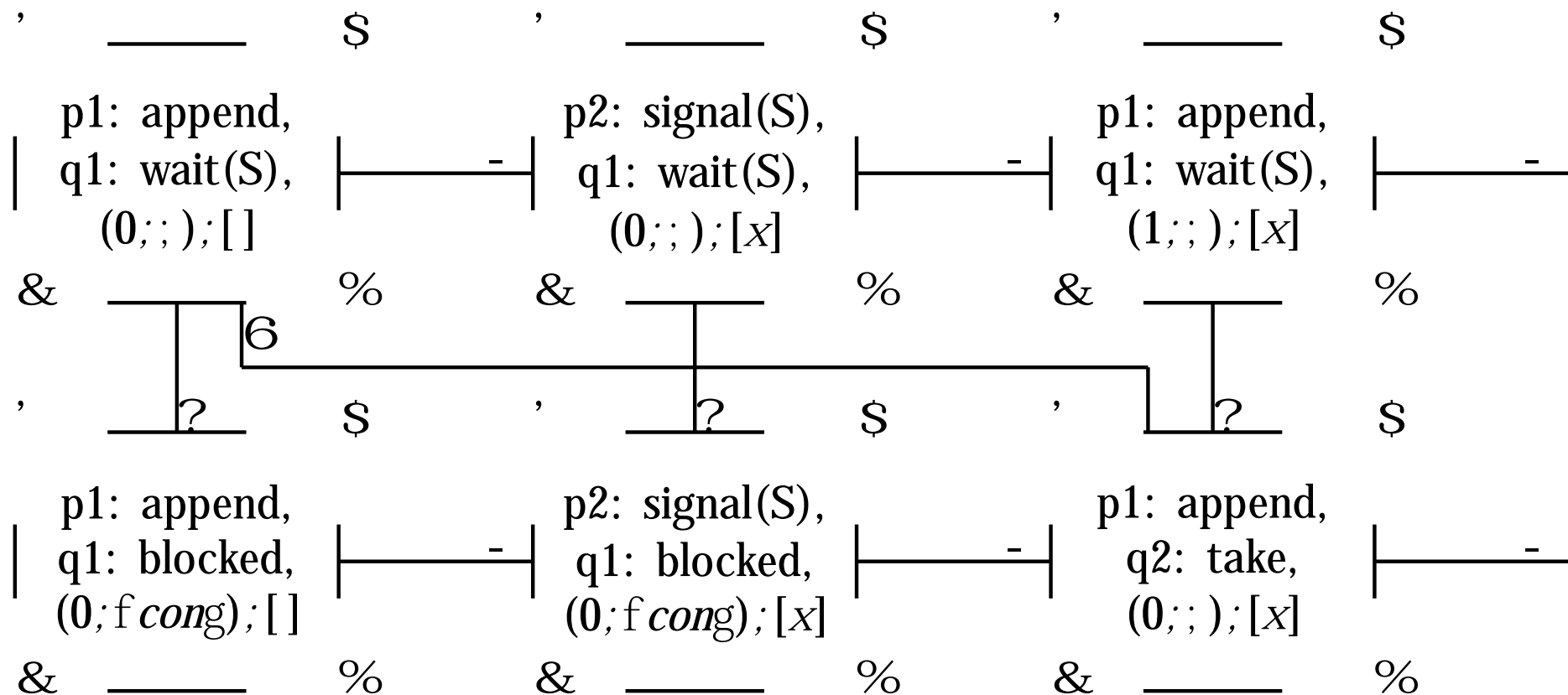
r1: wait(S1)

r2: wait(S2)

r3: merge halves of A

Algorithm 6.6: Producer-consumer (infinite buffer)	
infinite queue of dataType buffer; // empty queue semaphore notEmpty // (0;;)	
producer	consumer
dataType d loop forever p1: d // produce p2: append(d, buffer) p3: signal(notEmpty)	dataType d loop forever q1: wait(notEmpty) q2: d // take(buffer) q3: consume(d)

Partial State Diagram for Producer-Consumer with Infinite Buffer



Algorithm 6.7: Producer-consumer (infinite buffer, abbreviated)	
infinite queue of dataType buffer; // empty queue semaphore notEmpty // (0;;)	
producer	consumer
dataType d loop forever p1: append(d, buffer) p2: signal(notEmpty)	dataType d loop forever q1: wait(notEmpty) q2: d = take(buffer)

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

```

note queue of dataType buf;  // empty queue
semaphore notEmpty // (0;;)
semaphore notFull  // (N;;)
    
```

producer

```

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

consumer

```

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

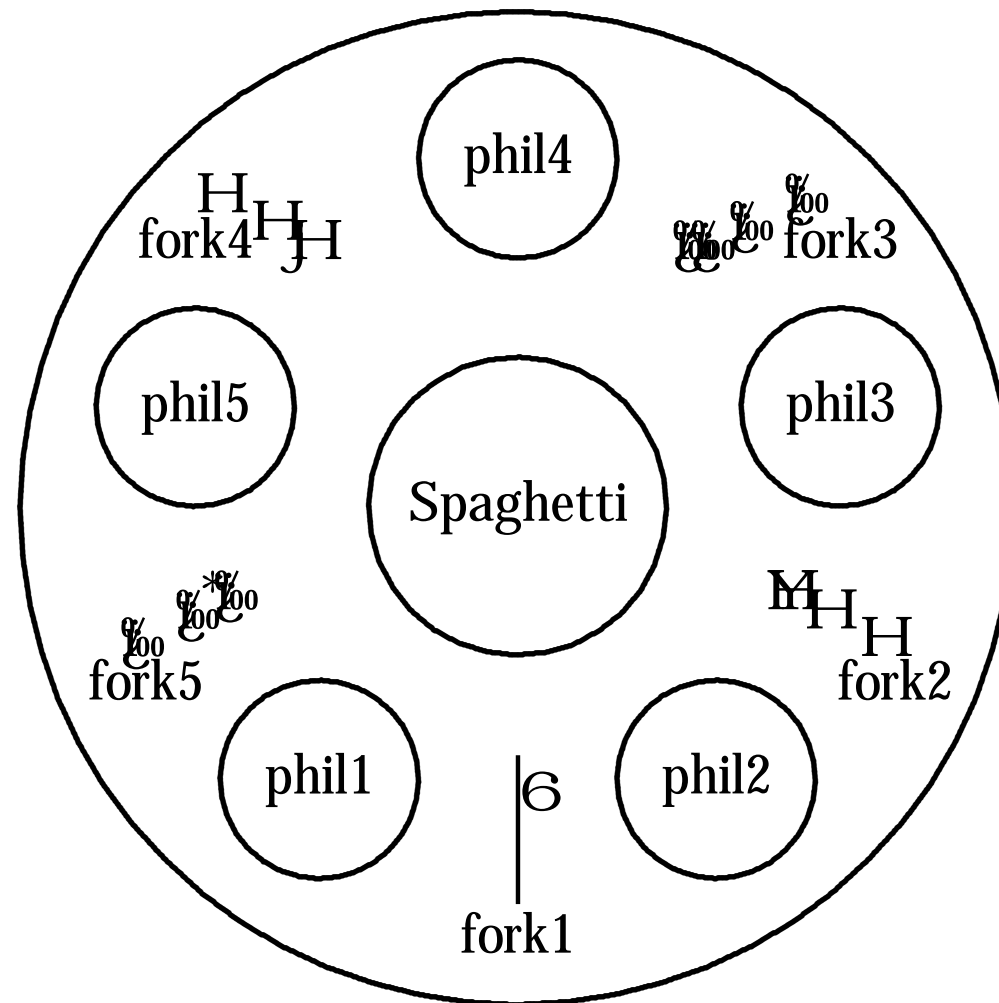
Scenario with Busy Waiting

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 (1st attempt)

semaphore array [0..4] fork $\bar{c}[1,1,1,1,1]$

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

p1: think

p2: wait(room)

p3: wait(fork[i])

p4: wait(fork[i+ 1])

p5: eat

p6: signal(fork[i])

p7: signal(fork[i+ 1])

p8: signal(room)

Algorithm 6.12: Dining philosophers (third attempt)
semaphore array [0..4] fork $\leftarrow [1,1,1,1,1]$
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 $S \in \{0, 1\}$
binary semaphore gate $\in \{0, 1\}$
integer count $\in \{0, \dots, k\}$

loop forever

 non-critical section

p1: wait(gate)

p2: wait(S) // Simulated wait

p3: count \leftarrow count + 1

p4: if count > 0 then

p5: signal(gate)

p6: signal(S)

 critical section

p7: wait(S) // Simulated signal

p8: count \leftarrow count - 1

p9: if count = -1 then

p10: signal(gate)

p11: signal(S)

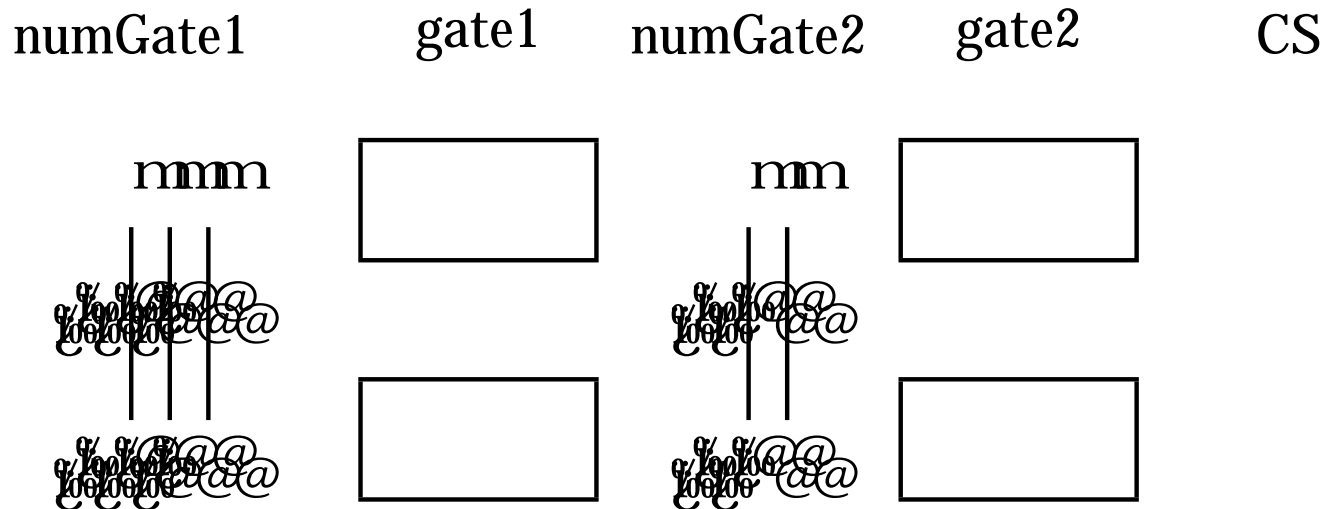
Algorithm 6.14: Udding's starvation-free algorithm

semaphore gate1 ≤ 1 , gate2 ≤ 0

integer numGate1 ≤ 0 , numGate2 ≤ 0

```
p1:  wait(gate1)
p2:  numGate1  $\leq$  numGate1 + 1
p3:  signal(gate1)
p4:  wait(gate1)
p5:  numGate2  $\leq$  numGate2 + 1
      numGate1  $\leq$  numGate1  $\leq 0$       // Statement is missing in the book
p6:  if numGate1 > 0
p7:      signal(gate1)
p8:  else signal(gate2)
p9:  wait(gate2)
p10: numGate2  $\leq$  numGate2  $\leq 0$ 
      critical section
p11: if numGate2 > 0
p12:     signal(gate2)
p13: 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

```
1  import java.util . concurrent. Semaphore;
2  class CountSem extends Thread {
3      static volatile int n = 0;
4      static Semaphore s = new Semaphore(1);
5
6      public void run() {
7          int temp;
8          for (int i = 0; i < 10; i++) {
9              try {
10                 s.acquire ();
11             }
12             catch (InterruptedException e) {}
13
14
15
```

Semaphores in Java

```
16         temp = n;
17         n = temp + 1;
18         s.release ();
19     }
20 }
21
22 public static void main(String[] args) {
23     // As before
24 }
25 }
```

Semaphores in Ada

```
1  protected type Semaphore(Initial: Natural) is  
2      entry Wait;  
3      procedure Signal;  
4  private  
5      Count: Natural := Initial ;  
6  end Semaphore;  
7  
8  
9  
10  
11  
12  
13  
14  
15
```

Semaphores in Ada

```
16  protected body Semaphore is  
17      entry Wait when Count > 0 is  
18      begin  
19          Count := Count - 1;  
20      end Wait;  
21  
22      procedure Signal is  
23      begin  
24          Count := Count + 1;  
25      end Signal;  
26  end Semaphore;
```


Busy-Wait Semaphores in Promela

```
1  /ï%Copyright (C) 2006 M. Benï%Ari. See copyright.txt  ÿ%00
2  /ï%Definition of busy ÿwait semaphores  ÿ%00
3  inline wait( s ) {
4      atomic { s > 0 ; sï%00}ï%00
5  }
6
7  inline signal ( s ) { s++ }
```

Weak Semaphores in Promela (three processes)

```
1  /i %Copyright (C) 2006 M. Ben-Ari. See copyright.txt  i %  
2  /i %Weak semaphore  i %  
3  /i %NPROCS  i %the number of processes  i %must be defined.  i %  
4  /i %THIS VERSION is specialized for exactly THREE processes  i %  
5  
6  /i %A semaphore is a count plus an array of blocked processes  i %  
7  typedef Semaphore {  
8    byte count;  
9    bool blocked[NPROCS];  
10 };  
11  
12 /i %initialize    semaphore to n  i %  
13 inline initSem(S, n) {  
14   S.count = n  
15 }
```

Weak Semaphores in Promela (three processes)

```
16  /\% Wait operation:  %\%
17  /\% If count is zero, set blocked and wait for unblocked  %\%
18  inline wait(S) {
19      atomic {
20          if
21              :: S.count >= 1 % S.count %\%
22              :: else % S.blocked[_pid] % = true; !S.blocked[_pid] %
23          %\%
24      }
25  }
26
27  /\% Signal operation:  %\%
28  /\% If there are blocked processes, remove one nondeterministically  %\%
29  inline signal(S) {
30      atomic {
```

Weak Semaphores in Promela (three processes)

```
31      if
32      :: S.blocked[0] ȷȷȷȷ S.blocked[0] = false
33      :: S.blocked[1] ȷȷȷȷ S.blocked[1] = false
34      :: S.blocked[2] ȷȷȷȷ S.blocked[2] = false
35      :: else ȷȷȷȷ S.count++
36      ȷȷȷȷ
37  }
38 }
```

Weak Semaphores in Promela (N processes)

```
1  /i %Copyright (C) 2006 M. Ben-Ari. See copyright.txt  i %  
2  /i %Weak semaphore  i %  
3  /i %NPROCS  i %the number of processes  i %must be defined.  i %  
4  
5  /i %A semaphore is a count plus an array of blocked processes  i %  
6  typedef Semaphore {  
7    byte count;  
8    bool blocked[NPROCS];  
9    byte i, choice;  
10 };  
11  
12 /i %initialize semaphore to n  i %  
13 inline initSem(S, n) {  
14   S.count = n  
15 }
```

Weak Semaphores in Promela (N processes)

```

16  /ï %Wait operation:  ÿ %
17  /ï %If count is zero, set blocked and wait for unblocked  ÿ %
18  inline wait(S) {
19      atomic {
20          if
21              :: S.count >= 1  ÿ %S.count  ÿ %
22              :: else  ÿ %S.blocked[_pid  ÿ % = true; !S.blocked[_pid  ÿ %
23              ÿ %
24          }
25  }
26
27  /ï %Signal operation:  ÿ %
28  /ï %If there are blocked processes, remove each one and  ÿ %
29  /ï %nondeterministically decide whether to replace it in the channel  ÿ %
30  /ï %or exit the operation.  ÿ %

```

Weak Semaphores in Promela (N processes)

```
31 inline signal (S) {
32     atomic {
33         S.i = 0;
34         S.choice = 255;
35         do
36             :: (S.i == NPROCS) ÷ % break
37             :: (S.i < NPROCS) && !S.blocked[S.i] ÷ % S.i++
38             :: else ÷ %
39                 if
40                     :: (S.choice == 255) ÷ % S.choice = S.i
41                     :: (S.choice != 255) ÷ % S.choice = S.i
42                     :: (S.choice != 255) ÷ %
43                 ÷ %
44                 S.i++
45         od;
```

Weak Semaphores in Promela (N processes)

```
46      if
47      :: S.choice == 255 % S.count++
48      :: else % S.blocked[S.choice] = false
49      %
50      }
51 }
```


Barz's Algorithm in Promela

```

1  #define NPROCS 3
2  #define K      2
3  byte gate = 1;
4  int count = K;
5  byte critical = 0;
6  active [NPROCS] proctype P () {
7      do ::
8          atomic { gate > 0; gate = 0; }
9          d_step {
10             count = count - 1;
11             if
12                 :: count > 0 { gate = 1; }
13                 :: else
14                     { }
15             }

```

Barz's Algorithm in Promela

```
16    critical ++;  
17    assert (critical <= 1);  
18    critical %%  
19    d_step {  
20        count++;  
21        if  
22            :: count == 1 %% gate++  
23            :: else  
24                %%  
25        }  
26    od  
27 }
```

Algorithm 6.15: Semaphore algorithm Asemaphore S $\leftarrow 1$, semaphore T $\leftarrow 0$ **p**

p1: wait(S)
p2: write("p")
p3: signal(T)

q

q1: wait(T)
q2: write("q")
q3: signal(S)

Algorithm 6.16: Semaphore algorithm B

semaphore S1 $\dot{=} 0$, S2 $\dot{=} 0$

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 $\leftarrow 1$
 boolean B $\leftarrow \text{false}$

p

p1: wait(S)
 p2: B $\leftarrow \text{true}$
 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 $S \in \{0, 1\}$, delay $\in \mathbb{R}_{\geq 0}$

integer count $\in \{0, \dots, k\}$

integer m

loop forever

p1: non-critical section

p2: wait(S)

p3: count \leftarrow count + 1

p4: $m \leftarrow$ count

p5: signal(S)

p6: if $m \neq 0$ then wait(delay)

p7: critical section

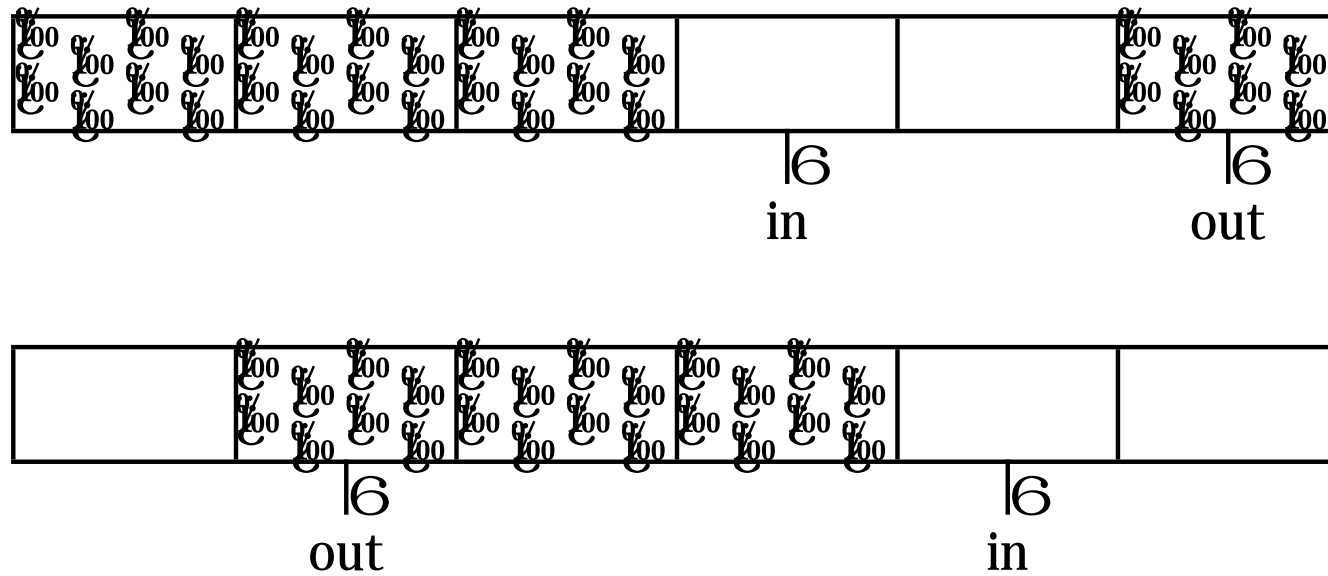
p8: wait(S)

p9: count \leftarrow count - 1

p10: if count ≤ 0 then signal(delay)

p11: signal(S)

Circular Buffer



Algorithm 6.19: Producer-consumer (circular buffer)

```

dataType array [0..N] buffer
integer in, out := 0
semaphore notEmpty := 0;
semaphore notFull := N;

```

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 $S \in \{1, 0\}$, gate $\in \{0, 1\}$
integer count $\in \mathbb{Z}$

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 > 0
p10: signal(gate)
p11: signal(S)

Weak Semaphores in Promela with Channels

```
1  /ï %Weak semaphore ÿ %  
2  /ï %NPROCS ÿ %the number of processes ÿ %must be defined.  ÿ %  
3  
4  /ï %A semaphore is a count plus a channel ÿ %  
5  /ï %plus a couple of local variables  ÿ %  
6  typedef Semaphore {  
7    byte count;  
8    chan ch = [NPROCS] of { pid };  
9    byte temp, i;  
10 };  
11  
12 /ï %initialize semaphore to n ÿ %  
13 inline initSem(S, n) {  
14   S.count = n  
15 }
```

Weak Semaphores in Promela with Channels

```
16  /ï %Wait operation: ÿ %  
17  /ï %If count is zero, place your _pid in the channel ÿ %  
18  /ï %and block until it is removed. ÿ %  
19  inline wait(S) {  
20      atomic {  
21          if  
22              :: S.count >= 1 ÿ % S.count ÿ %  
23              :: else ÿ % S.ch ! _pid; !(S.ch ?? [eval(_pid)])  
24          }  
25      }  
26  }  
27  /ï %Signal operation: ÿ %  
28  /ï %If there are blocked processes, remove each one and ÿ %  
29  /ï %nondeterministically decide whether to replace it in the channel ÿ %  
30  /ï %or exit the operation. ÿ %
```

Weak Semaphores in Promela with Channels

```
31 inline signal (S) {
32     atomic {
33         S.i = len(S.ch);
34         if
35             :: S.i == 0 %0 S.count++ /%0 No blocked process, increment count
36             :: else %0
37             do
38                 :: S.i == 1 %0 S.ch ? _; break /%0 Remove only blocked process
39                 :: else %0 S.i
40                     S.ch ? S.temp;
41                     if :: break :: S.ch ! S.temp %0
42                 od
43             %0
44     }
45 }
```

Algorithm 6.21: Readers and writers with semaphores

```
semaphore readerSem  $\dot{=}$  0, writerSem  $\dot{=}$  0  
integer delayedReaders  $\dot{=}$  0, delayedWriters  $\dot{=}$  0  
semaphore entry  $\dot{=}$  1  
integer readers  $\dot{=}$  0, writers  $\dot{=}$  0
```

SignalProcess

```
if writers = 0 or delayedReaders > 0  
    delayedReaders  $\dot{=}$  delayedReaders + 1  
    signal(readerSem)  
else if readers = 0 and writers = 0 and delayedWriters > 0  
    delayedWriters  $\dot{=}$  delayedWriters + 1  
    signal(writerSem)  
else signal(entry)
```

Algorithm 6.21: Readers and writers with semaphores

StartRead

p1: wait(entry)
p2: if writers > 0
p3: delayedReaders $\dot{+}$ delayedReaders + 1
p4: signal(entry)
p5: wait(readerSem)
p6: readers $\dot{+}$ readers + 1
p7: SignalProcess

EndRead

p8: wait(entry)
p9: readers $\dot{-}$ readers $\dot{-}$ 1
p10: SignalProcess

Algorithm 6.21: Readers and writers with semaphores

StartWrite

p11: wait(entry)
p12: if writers > 0 or readers > 0
p13: delayedWriters \downarrow % delayedWriters + 1
p14: signal(entry)
p15: wait(writerSem)
p16: writers \downarrow % writers + 1
p17: SignalProcess

EndWrite

p18: wait(entry)
p19: writers \downarrow % writers \downarrow % do
p20: SignalProcess

Algorithm 7.1: Atomicity of monitor operations

monitor CS

integer $n \geq 0$

operation increment

integer temp

temp $\leftarrow n$

$n \leftarrow \text{temp} + 1$

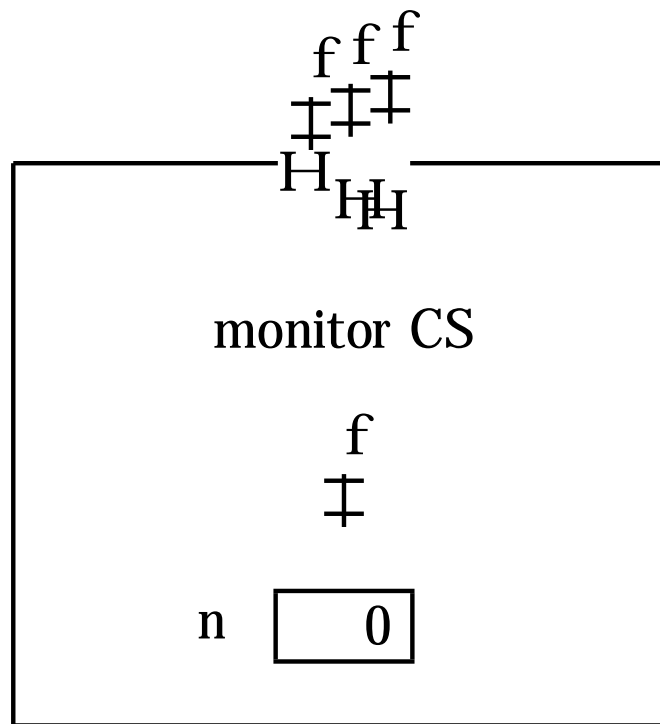
p

p1: CS.increment

q

q1: CS.increment

Executing a Monitor Operation



Algorithm 7.2: Semaphore simulated with a monitor

```

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

p

```

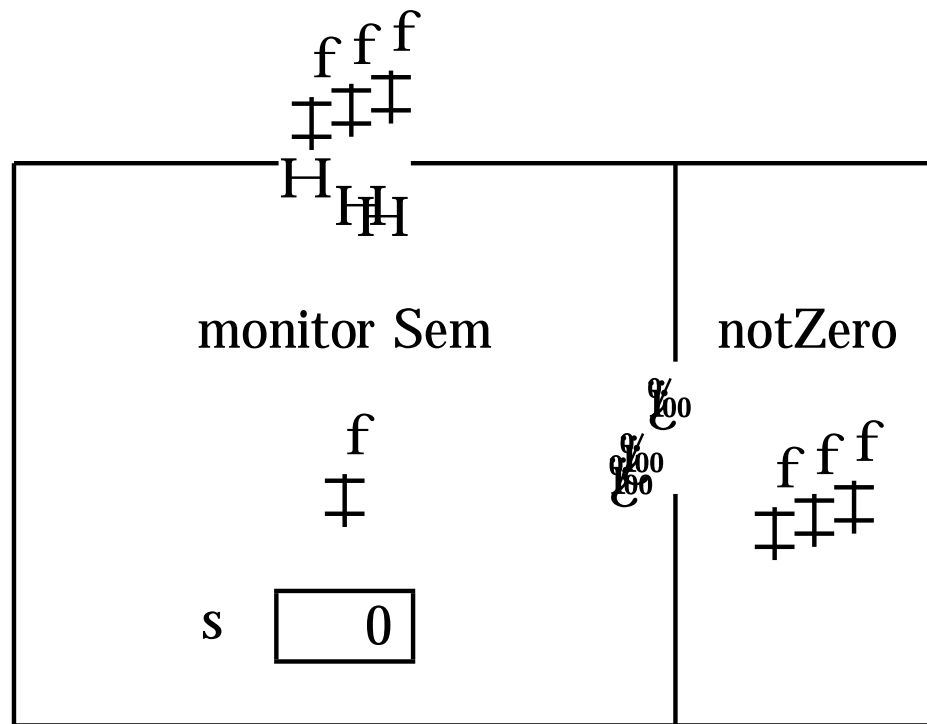
loop forever
  non-critical section
p1:  Sem.wait
    critical section
p2:  Sem.signal
  
```

q

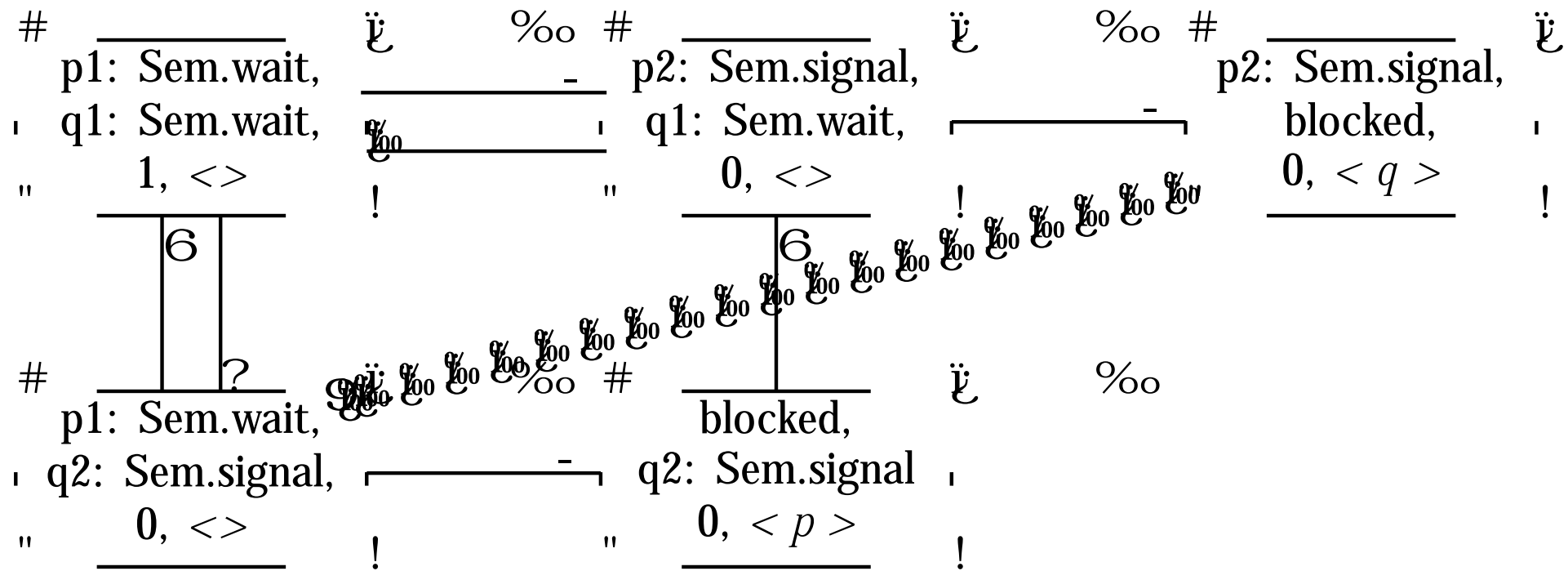
```

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

Condition Variable in a Monitor



State Diagram for the Semaphore Simulation



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

monitor PC

buftype buftail %empty

condition notEmpty

condition notFull

operation append(datatype V)

if buftail is full

waitC(notFull)

append(V, buftail)

signalC(notEmpty)

operation take()

datatype W

if buftail is empty

waitC(notEmpty)

W ← head(buftail)

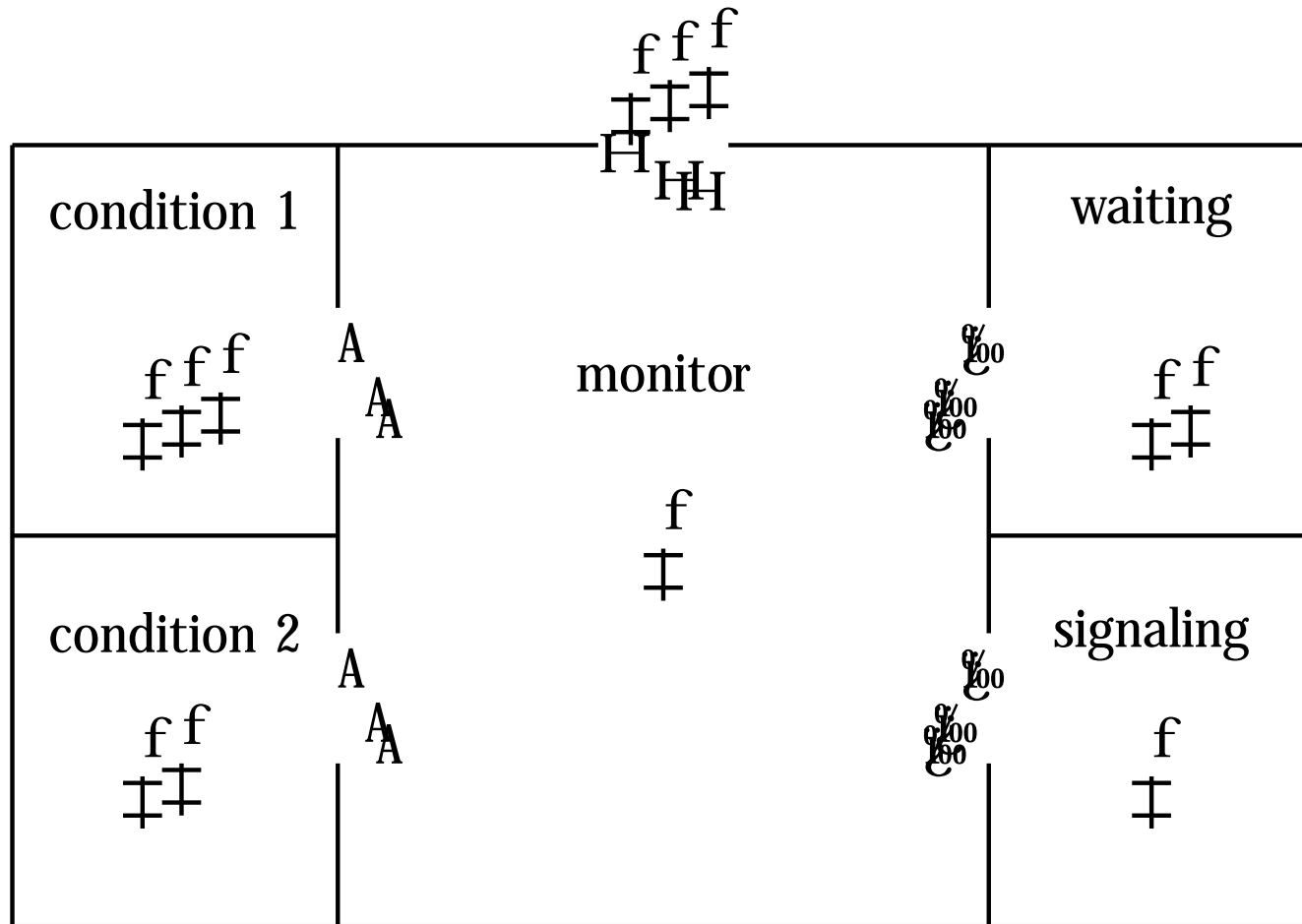
signalC(notFull)

return W

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

producer	consumer
<pre> datatype D loop forever p1: D ← %produce p2: PC.append(D) </pre>	<pre> datatype D loop forever q1: D ← %PC.take q2: consume(D) </pre>

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

p1: RW.StartRead

p2: read the database

p3: RW.EndRead

writer

q1: RW.StartWrite

q2: write to the database

q3: RW.EndWrite

Algorithm 7.5: Dining philosophers with a monitor

monitor ForkMonitor

integer array[0..4] fork $\in [2, \dots, 2]$

condition array[0..4] OKtoEat

operation takeForks(integer i)

if fork[i] $\in 2$

waitC(OKtoEat[i])

fork[i+1] \in fork[i+1] $\in 1$

fork[i-1] \in fork[i-1] $\in 1$

operation releaseForks(integer i)

fork[i+1] \in fork[i+1] $+ 1$

fork[i-1] \in 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	f_0	f_1	f_2	f_3	f_4
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

```
1  monitor RW {
2      int readers = 0, writing = 1;
3      condition OKtoRead, OKtoWrite;
4
5      void StartRead() {
6          if (writing && !empty(OKtoWrite))
7              waitc(OKtoRead);
8          readers = readers + 1;
9          signalc(OKtoRead);
10     }
11     void EndRead() {
12         readers = readers - 1;
13         if (readers == 0)
14             signalc(OKtoWrite);
15     }
```

Readers and Writers in C

```
16  void StartWrite () {  
17      if (writing && (readers != 0))  
18          waitc(OKtoWrite);  
19      writing = 1;  
20  }  
21  
22  void EndWrite() {  
23      writing = 0;  
24      if (empty(OKtoRead))  
25          signalc (OKtoWrite);  
26      else  
27          signalc (OKtoRead);  
28  }  
29 }
```

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 \rightarrow false
(blocked)	signalC(OKtoRead)
readers \rightarrow 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 \rightarrow false
(blocked)	when not writing
(blocked)	readers \rightarrow readers + 1
read the data	...

Simple Readers and Writers in Ada

```
1  protected RW is  
2      procedure Write(I: Integer );  
3      function Read return Integer;  
4  private  
5      N: Integer := 0;  
6  end RW;  
7  
8  
9  
10  
11  
12  
13  
14  
15
```

Simple Readers and Writers in Ada

```
16  protected body RW is  
17      procedure Write(I: Integer ) is  
18      begin  
19          N := I;  
20      end Write;  
21      function Read return Integer is  
22      begin  
23          return N;  
24      end Read;  
25  end RW;
```

Readers and Writers in Ada

```
1  protected RW is  
2      entry StartRead;  
3      procedure EndRead;  
4      entry Startwrite ;  
5      procedure EndWrite;  
6  private  
7      Readers: Natural := 0;  
8      Writing: Boolean := false ;  
9  end RW;  
10  
11  
12  
13  
14  
15
```

Readers and Writers in Ada

```
16    protected body RW is
17        entry StartRead
18            when not Writing is
19        begin
20            Readers := Readers + 1;
21        end StartRead;
22
23        procedure EndRead is
24        begin
25            Readers := Readers - 1;
26        end EndRead;
27
28
29
30
```

Readers and Writers in Ada

```
31      entry StartWrite
32          when not Writing and Readers = 0 is
33      begin
34          Writing := true;
35      end StartWrite;
36
37      procedure EndWrite is
38      begin
39          Writing := false ;
40      end EndWrite;
41  end RW;
```

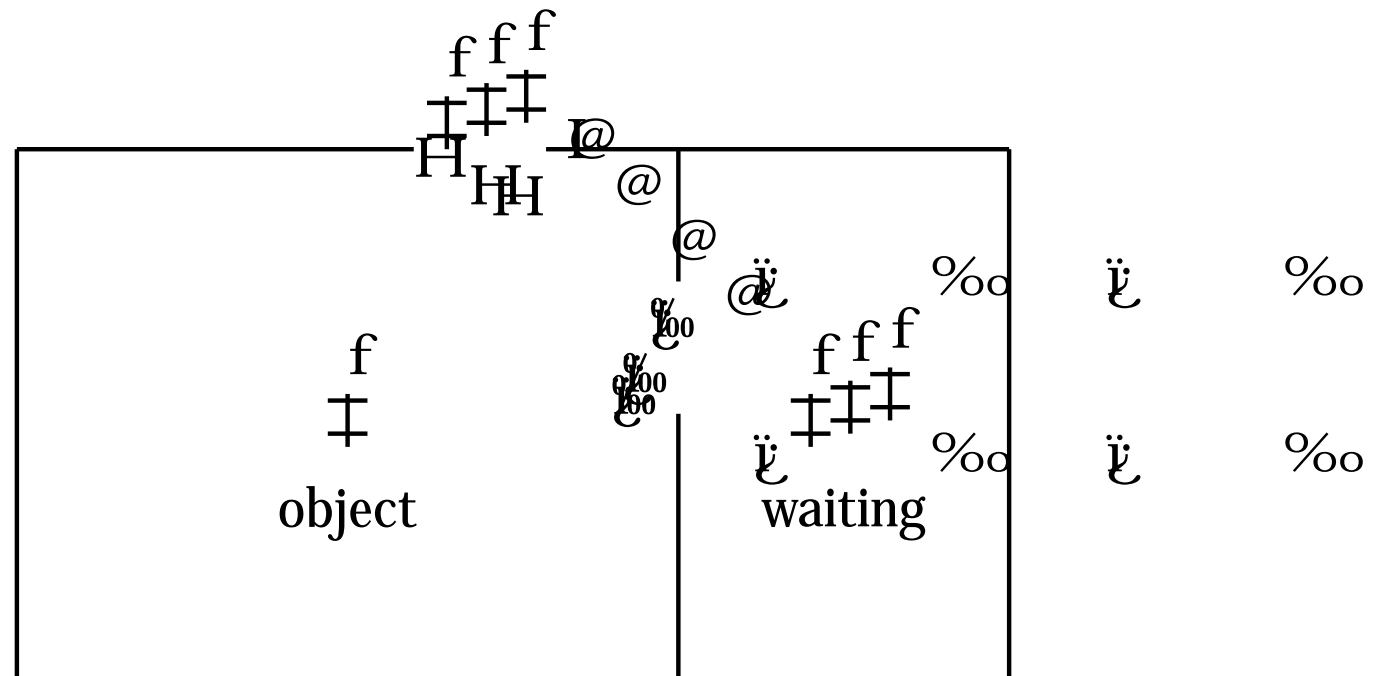
Producer-Consumer in Java

```
1  class PCMonitor {  
2      final int N = 5;  
3      int Oldest = 0, Newest = 0;  
4      volatile int Count = 0;  
5      int Buf%[] = new int[N];  
6      synchronized void Append(int V) {  
7          while (Count == N)  
8              try {  
9                  wait();  
10             } catch (InterruptedException e) {}  
11             Buf%[Newest] = V;  
12             Newest = (Newest + 1) % N;  
13             Count = Count + 1;  
14             notifyAll ();  
15     }
```

Producer-Consumer in Java

```
16    synchronized int Take() {  
17        int temp;  
18        while (Count == 0)  
19            try {  
20                wait();  
21            } catch (InterruptedException e) {}  
22        temp = Buffer[Oldest];  
23        Oldest = (Oldest + 1) % N;  
24        Count = Count + 1;  
25        notifyAll ();  
26        return temp;  
27    }  
28 }
```


A Monitor in Java With notifyAll



Java Monitor for Readers and Writers

```
1  class RWMonitor {
2      volatile int readers = 0;
3      volatile boolean writing = false;
4      synchronized void StartRead() {
5          while (writing )
6              try {
7                  wait ();
8              } catch (InterruptedException e) {}
9          readers = readers + 1;
10         notifyAll ();
11     }
12     synchronized void EndRead() {
13         readers = readers - 1;
14         if (readers == 0) notifyAll ();
15     }
```

Java Monitor for Readers and Writers

```
16  synchronized void StartWrite() {  
17      while (writing && (readers != 0))  
18          try {  
19              wait();  
20          } catch (InterruptedException e) {}  
21      writing = true;  
22  }  
23  synchronized void EndWrite() {  
24      writing = false;  
25      notifyAll ();  
26  }  
27 }
```

Simulating Monitors in Promela

```
1  /ï¿½ Copyright (C) 2006 M. Ben-Ari. See copyright.txt  ï¿½ï¿½ï¿½
2  /ï¿½ Definitions for monitor  ï¿½ï¿½ï¿½
3  bool lock = false;
4
5  typedef Condition {
6      bool gate;
7      byte waiting;
8  }
9
10 inline enterMon() {
11     atomic {
12         !lock;
13         lock = true;
14     }
15 }
```

Simulating Monitors in Promela

```
16
17 inline leaveMon() {
18     lock = false;
19 }
20
21 inline waitC(C) {
22     atomic {
23         C.waiting ++;
24         lock = false;    /ÿ %Exit monitor ÿ %00
25         C.gate;          /ÿ %Wait for gate ÿ %00
26         lock = true;    /ÿ %RR ÿ %00
27         C.gate = false; /ÿ %Reset gate ÿ %00
28         C.waiting ÿ %0ÿ %00
29     }
30 }
```

Simulating Monitors in Promela

```

31
32 inline signalC (C) {
33     atomic {
34         if
35             /ï%Signal only if waiting ÿ%00
36             :: (C.waiting > 0) ÿ%00
37                 C.gate = true;
38                 !lock;      /ï%RR ÿ%wait for released lock ÿ%00
39                 lock = true; /ï%Take lock again ÿ%00
40             :: else
41                 ÿ%00
42         }
43     }
44
45     #define emptyC(C) (C.waiting == 0)

```

Readers and Writers in Ada (1)

```
1  protected RW is  
2  
3      entry Start_Read;  
4      procedure End_Read;  
5      entry Start_Write;  
6      procedure End_Write;  
7  
8  private  
9      Waiting_To_Read : integer := 0;  
10     Readers : Natural := 0;  
11     Writing : Boolean := false ;  
12  
13 end RW;
```

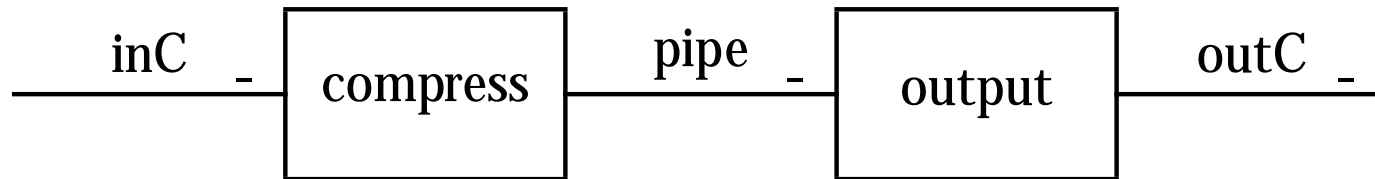
Readers and Writers in Ada (2)

```
1  protected RW is  
2  
3      entry StartRead;  
4      procedure EndRead;  
5      entry Startwrite ;  
6      procedure EndWrite;  
7      function NumberReaders return Natural;  
8  
9  private  
10     entry ReadGate;  
11     entry WriteGate;  
12     Readers: Natural := 0;  
13     Writing: Boolean := false ;  
14  
15  end RW;
```

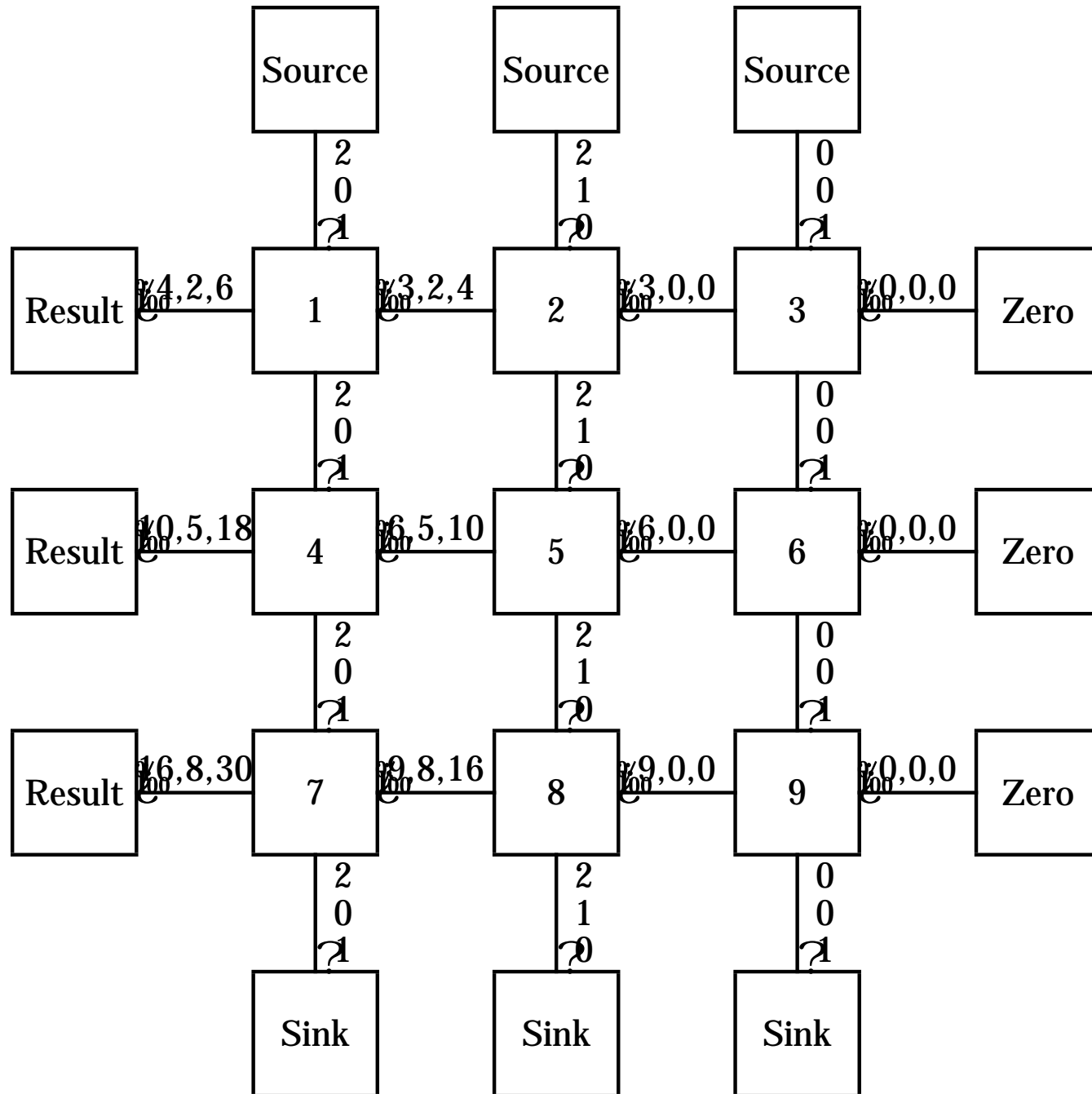

Algorithm 8.1: Producer-consumer (channels)	
channel of integer ch	
producer	consumer
integer x loop forever p1: x := produce p2: ch (x	integer y loop forever q1: ch) y q2: consume(y)

Algorithm 8.2: Conway's problem	
constant integer MAX ¶%9 constant integer K ¶%4 channel of integer inC, pipe, outC	
compress	output
char c, previous ¶%0 integer n ¶%0 inC) previous loop forever p1: inC) c p2: if (c = previous) and (n < MAX ¶%0) p3: n ¶%n + 1 else p4: if n > 0 p5: pipe (intToChar(n+1) p6: n ¶%0 p7: pipe (previous p8: previous ¶%0	char c integer m ¶%0 loop forever q1: pipe) c q2: outC (c q3: m ¶%m + 1 q4: if m >= K q5: outC (newline q6: m ¶%0 q7: q8:

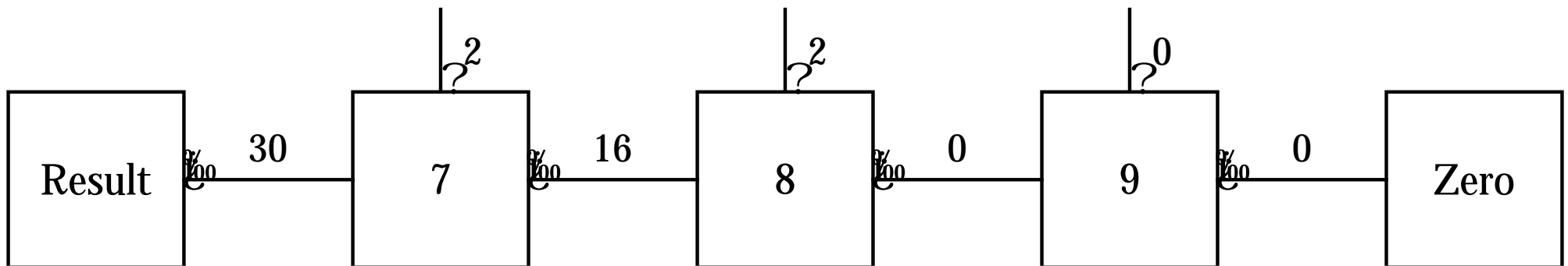
Conway's Problem



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	fork 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	boolean dummy loop forever q1: forks[i] (true q2: forks[i]) dummy q3: q4: q5: q6:

Conway's Problem in Promela

```

1  #define N 9
2  #define K 4
3  chan inC, pipe, outC = [0] of { byte };
4
5  active proctype Compress() {
6      byte previous, c, count = 0;
7      inC ? previous ;
8      do
9          :: inC ? c ;
10         if
11             :: (c == previous) && (count < N) ; count++
12             :: else ;
13
14
15

```

Conway's Problem in Promela

```
16      if
17      :: count > 0 %0
18          pipe ! count+1;
19          count = 0
20      :: else
21          %0
22          pipe ! previous ;
23          previous = c;
24      %0
25  od
26  }
27
28
29
30
```

Conway's Problem in Promela

```
31  active proctype Output() {  
32      byte c, count = 0;  
33      do  
34          :: pipe ? c;  
35              outC ! c;  
36              count++;  
37          if  
38              :: count >= K % 10  
39                  outC ! '\n';  
40                  count = 0  
41              :: else  
42                  % 10  
43          od  
44  }
```

Multiplier Process in Promela

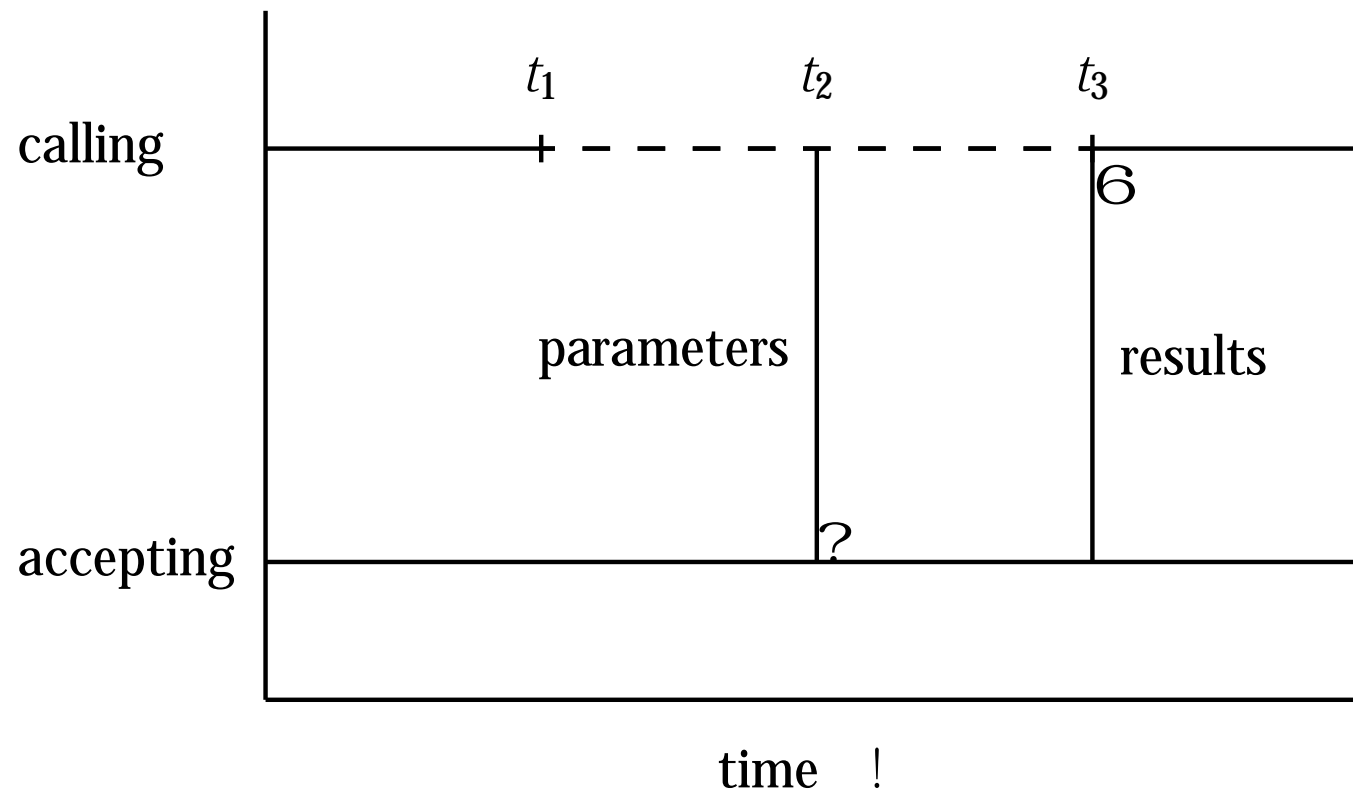
```

1  proctype Multiplier (byte Coef)
2      chan North; chan East; chan South; chan West) {
3      byte Sum, X;
4      for (i, 0, SIZE)
5          if :: North ? X
6              :: East ? Sum
7              :: South ? Sum
8              :: West ? Sum
9              Sum = Sum + X
10         West ! Sum;
11     rof (i)
12 }
```

Algorithm 8.6: Rendezvous

Algorithm 8.6: Rendezvous	
client	server
integer parm, result loop forever p1: parm \leftarrow ... p2: server.service(parm, result) p3: use(result)	integer p, r loop forever q1: q2: accept service(p, r) q3: r \leftarrow do the service(p)

Timing Diagram for a Rendezvous



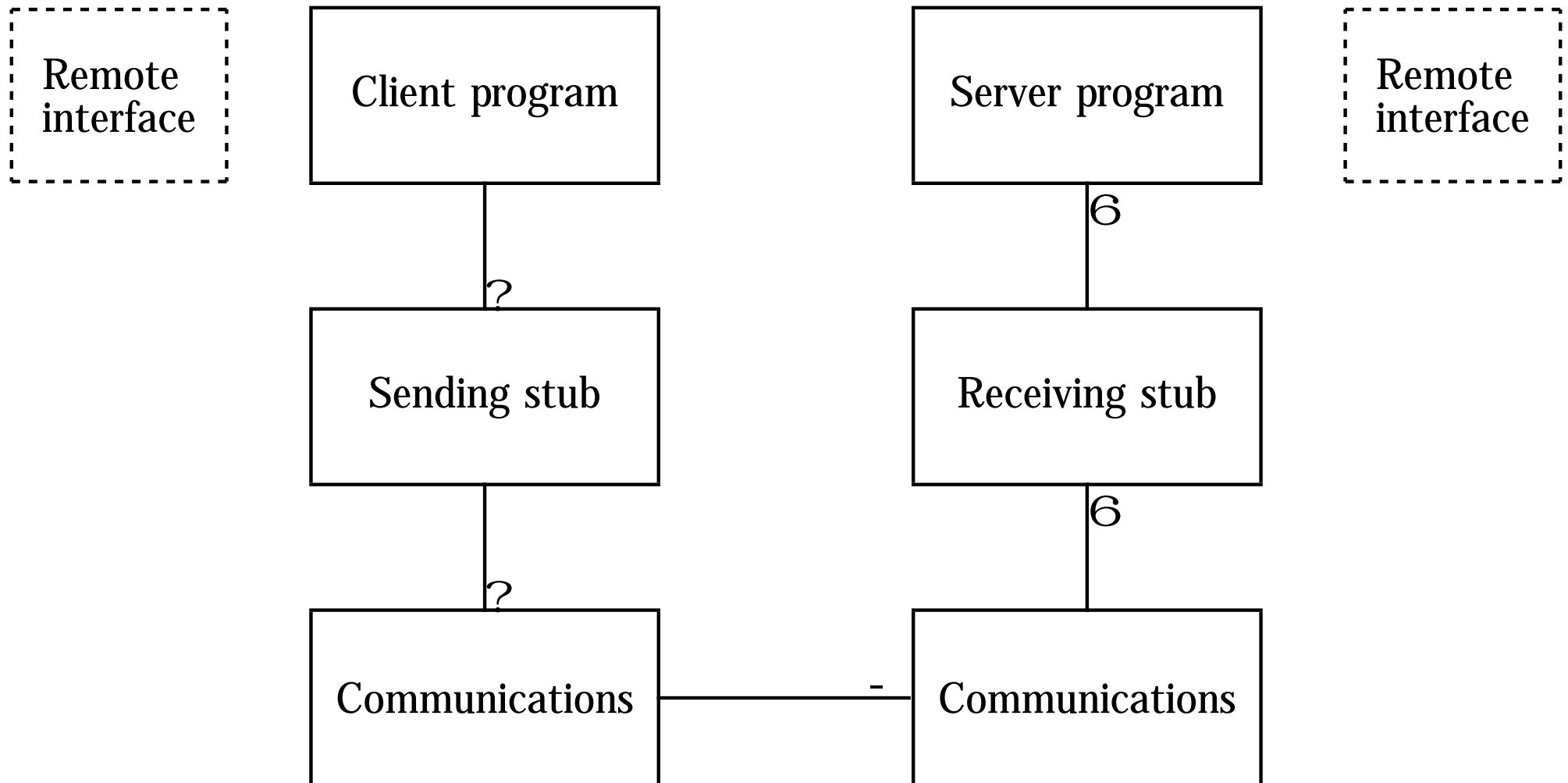
Bounded Buffer in Ada

```
1  task body Buf60 is
2      B: Buf60 Array;
3      In_Ptr, Out_Ptr, Count: Index := 0;
4
5  begin
6      loop
7          select
8              when Count < Index'Last =>
9                  accept Append(I: in Integer) do
10                     B(In_Ptr) := I;
11                     end Append;
12                     Count := Count + 1; In_Ptr := In_Ptr + 1;
13          or
14
15
```

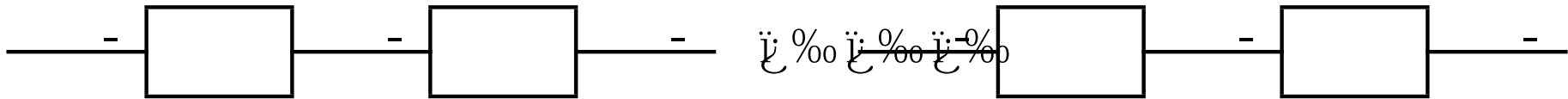
Bounded Buffer in Ada

```
16      when Count > 0 =>
17          accept Take(I: out Integer) do
18              I := B(Out_Ptr);
19          end Take;
20          Count := Count - 1; Out_Ptr := Out_Ptr + 1;
21      or
22          terminate;
23      end select;
24  end loop;
25 end Buffer;
```

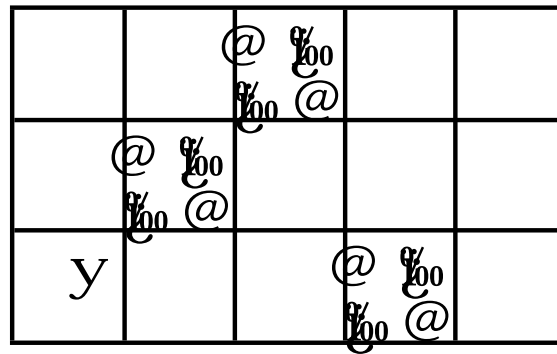

Remote Procedure Call



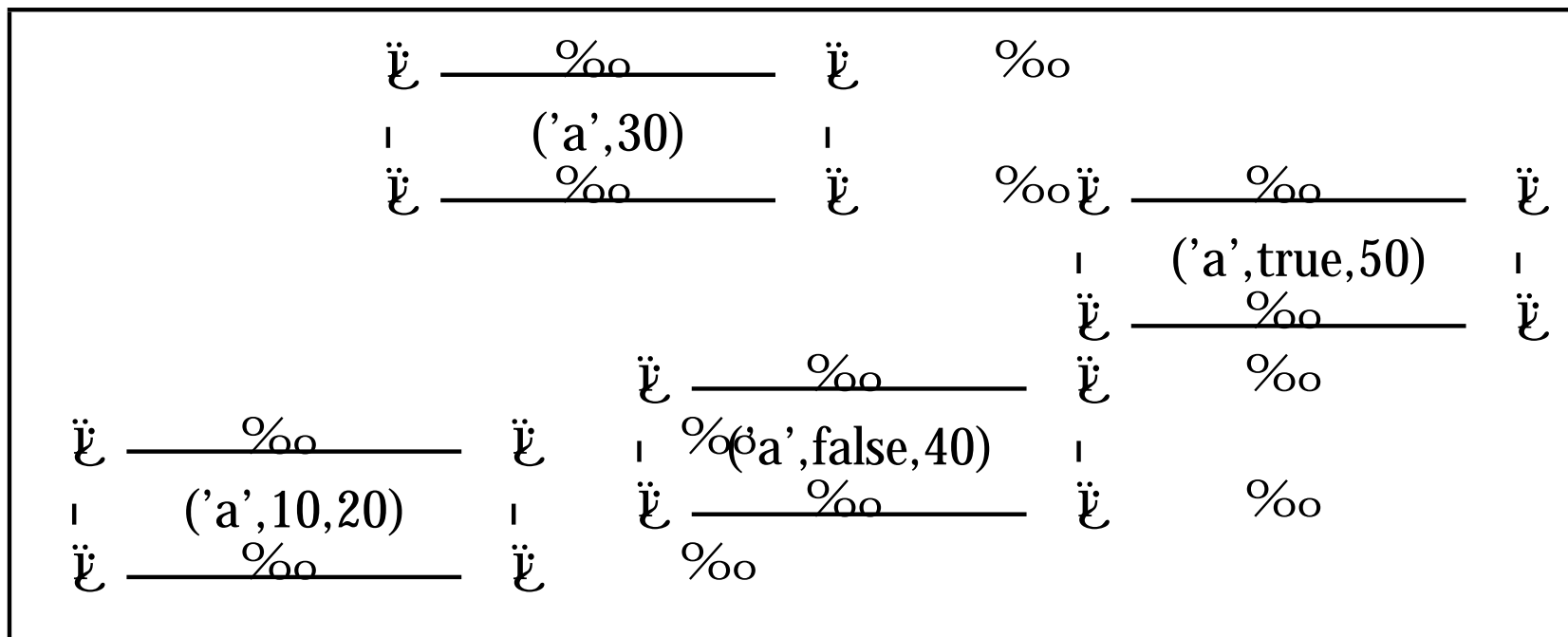
Pipeline Sort



Hoare's Game



A Space



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

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

Algorithm 9.4: Buffering in a space	
producer	consumer
<pre> integer count ← 0 integer v loop forever p1: v ← produce p2: postnote('B', count, v) p3: count ← count + 1 </pre>	<pre> integer count ← 0 integer v loop forever q1: remove('B', count, v) q2: consume(v) q3: count ← count + 1 </pre>

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 $n \geq 0$..

master	worker
integer i, j, result integer r, c p1: for i from 1 to n p2: for j from 1 to n p3: postnote('T', i, j) p4: for i from 1 to n p5: for j from 1 to n p6: removenote('R', r, c, result) p7: print r, c, result	integer r, c, result integer array[1..n] vec1, vec2 loop forever q1: removenote('T', r, c) q2: readnote('A', r=, vec1) q3: readnote('B', c=, vec2) q4: result \leftarrow vec1 \times vec2 q5: postnote('R', r, c, result) q6: q7:

Algorithm 9.7: Matrix multiplication in Linda with granularity

constant integer $n \gg 0$..
 constant integer chunk $\gg 0$..

master	worker
integer i, j, result integer r, c p1: for i from 1 to n p2: for j from 1 to n step by chunk p3: postnote('T', i, j) p4: for i from 1 to n p5: for j from 1 to n p6: removenote('R', r, c, result) p7: print r, c, result	integer r, c, k, result integer array[1..n] vec1, vec2 loop forever q1: removenote('T', r, k) q2: readnote('A', r=, vec1) q3: for c from k to k+chunk-1 q4: readnote('B', c=, vec2) q5: result \leftarrow vec1 \times vec2 q6: postnote('R', r, c, result) q7:

Definition of Notes in Java

```
1  public class Note {  
2      public String id;  
3      public Object[] p;  
4  
5      // Constructor for an array of objects  
6      public Note (String id, Object[] p) {  
7          this.id = id;  
8          if (p != null) this.p = p.clone();  
9      }  
10  
11     // Constructor for a single integer  
12     public Note (String id, int p1) {  
13         this (id, new Object[]{new Integer(p1)});  
14     }  
15
```

Definition of Notes in Java

```
16      // Accessor for a single integer value
17      public int get(int i) {
18          return ((Integer)p[i]).intValue();
19      }
20 }
```

Matrix Multiplication in Java

```
1  private class Worker extends Thread {  
2      public void run() {  
3          Note task = new Note("task");  
4          while (true) {  
5              Note t = space.removeNote(task);  
6              int row = t.get(0), col = t.get(1);  
7              Note r = space.readNote(match("a", row));  
8              Note c = space.readNote(match("b", col));  
9              int ip = 0;  
10             for (int i = 1; i <= SIZE; i++)  
11                 ip = ip + r.get(i) * c.get(i);  
12             space.postNote(new Note("result", row, col, ip));  
13         }  
14     }  
15 }
```

Matrix Multiplication in Promela

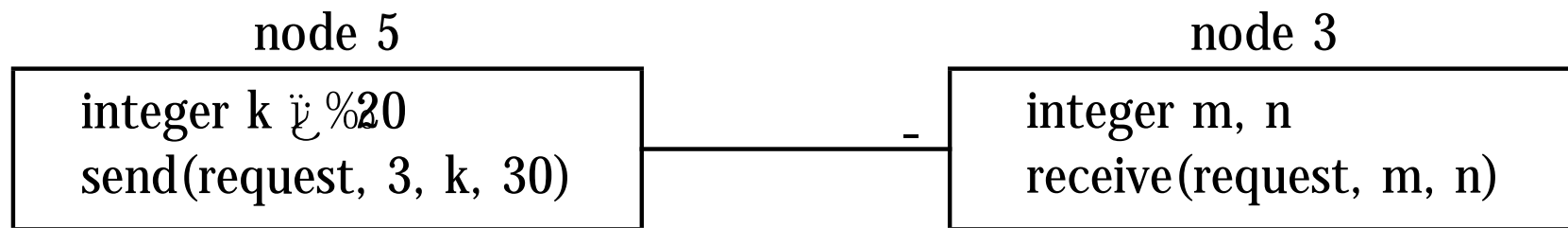
```
1  chan space = [25] of { byte, short, short, short, short };
2
3  active[WORKERS] proctype Worker() {
4      short row, col, ip, r1, r2, r3, c1, c2, c3;
5      do
6          :: space ?? 't', row, col, _, _;
7             space ?? <'a', eval(row), r1, r2, r3>;
8             space ?? <'b', eval(col), c1, c2, c3>;
9             ip = r1 * c1 + r2 * c2 + r3 * c3;
10            space ! 'r', row, col, ip, 0;
11      od;
12 }
```

Algorithm 9.8: Matrix multiplication in Linda (exercise)

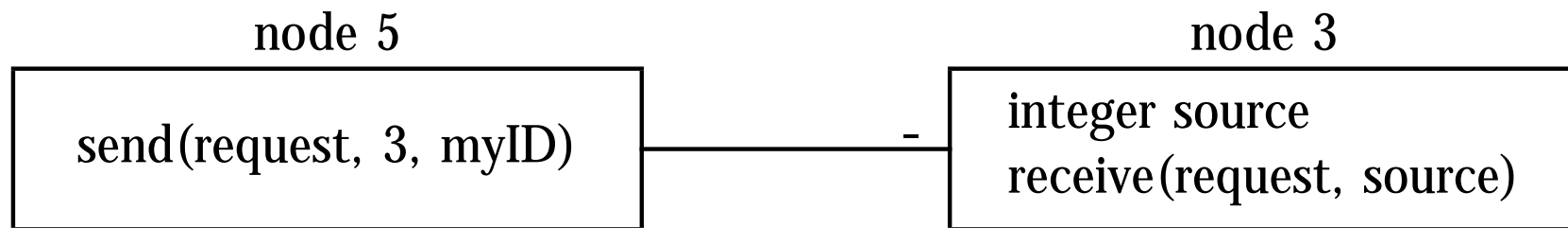
constant integer $n \gg 0$..

master	worker
integer i, j, result integer r, c p1: postnote('T', 0) p2: p3: p4: p5: p6: for i from 1 to n p7: for j from 1 to n p8: removenote('R', r, c, result) p9: print r, c, result	integer i, r, c, result integer array[1..n] vec1, vec2 loop forever q1: removenote('T' i) q2: if i < (n \div 10) \div 10 q3: postnote('T', i+1) q4: r \div (i = n) + 1 q5: c \div (i modulo n) + 1 q6: readnote('A', r=, vec1) q7: readnote('B', c=, vec2) q8: result \div vec1 \div vec2 q9: postnote('R', r, c, result)

Sending and Receiving Messages



Sending a Message and Expecting a Reply



Algorithm 10.1: Ricart-Agrawala algorithm (outline)

integer myNum $\in \mathcal{ID}$

set of node IDs deferred $\in \mathcal{ID}$ empty set

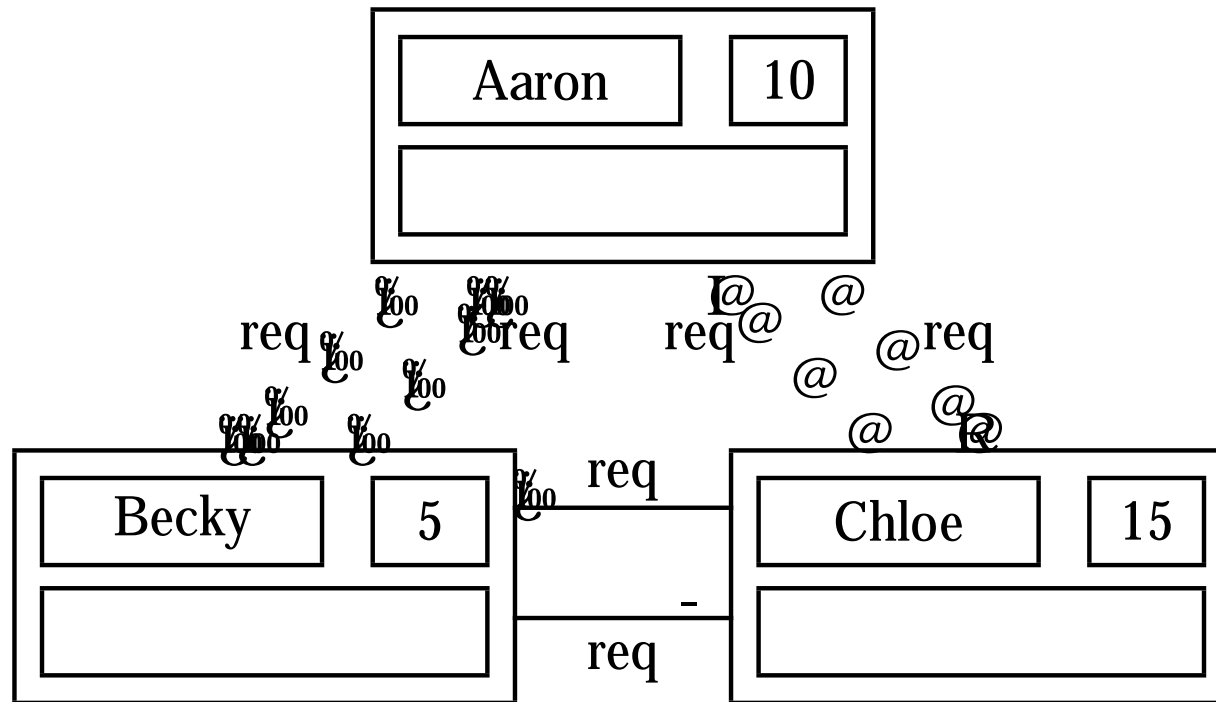
main

```
p1:  non-critical section
p2:  myNum  $\in \mathcal{ID}$  chooseNumber
p3:  for all other nodes N
p4:    send(request, N, myID, myNum)
p5:  await reply's from all other nodes
p6:  critical section
p7:  for all nodes N in deferred
p8:    remove N from deferred
p9:    send(reply, N, myID)
```

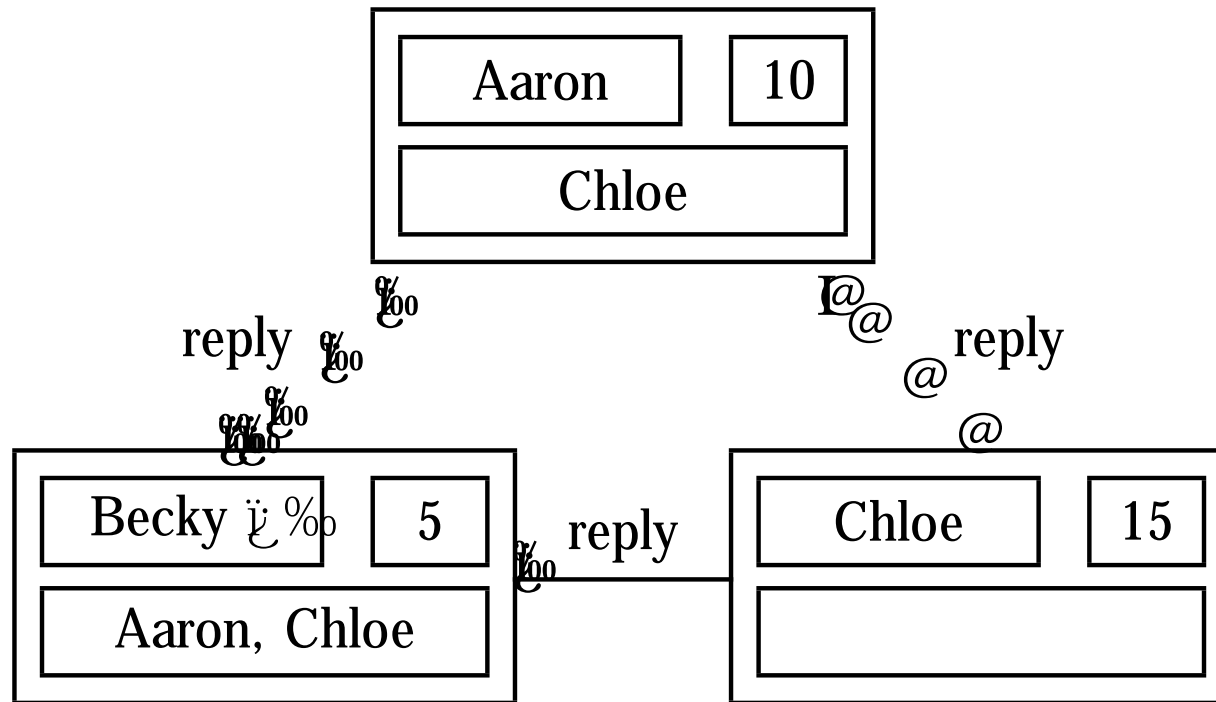
receive

```
integer source, reqNum
p10: receive(request, source, reqNum)
p11: if reqNum < myNum
p12:   send(reply, source, myID)
p13: else add source to deferred
```

RA Algorithm (1)



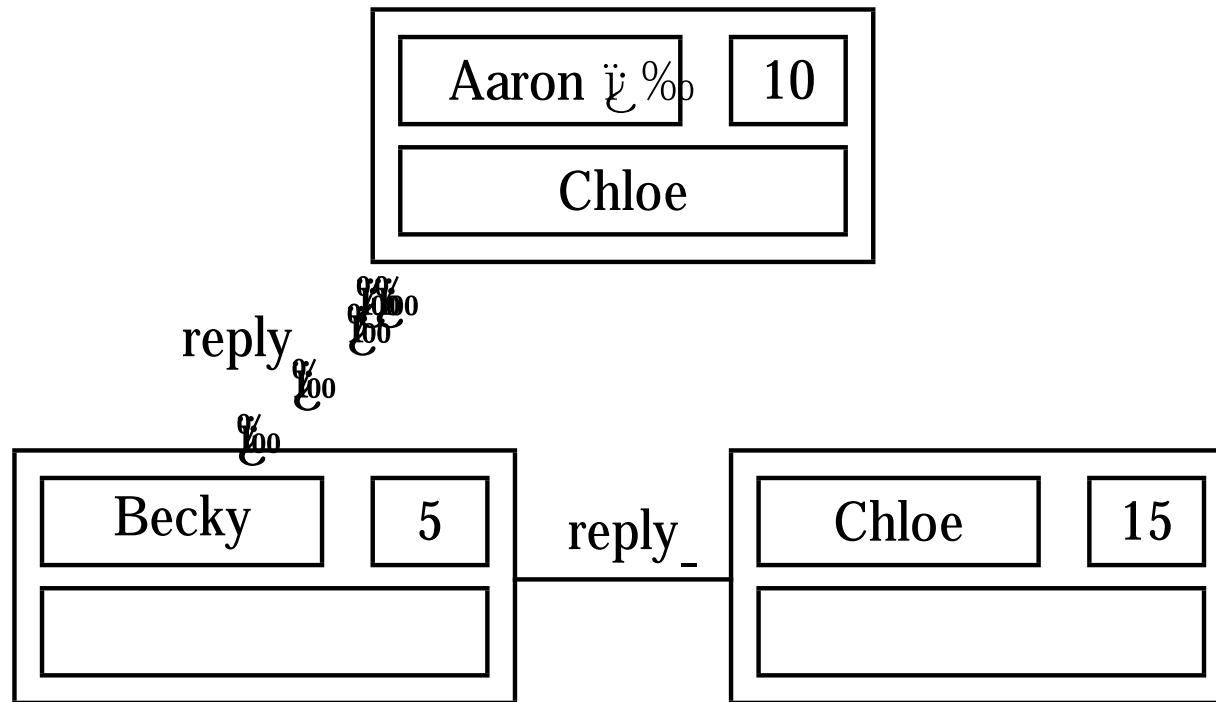
RA Algorithm (2)



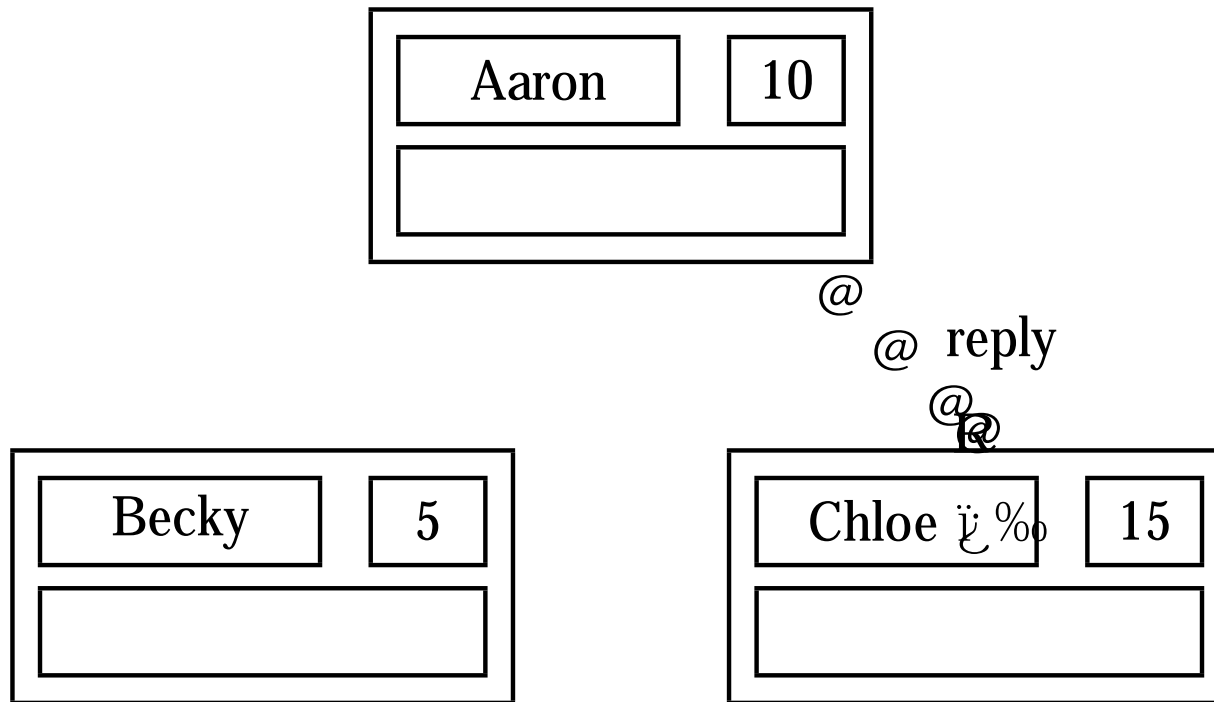
Virtual Queue in the RA Algorithm



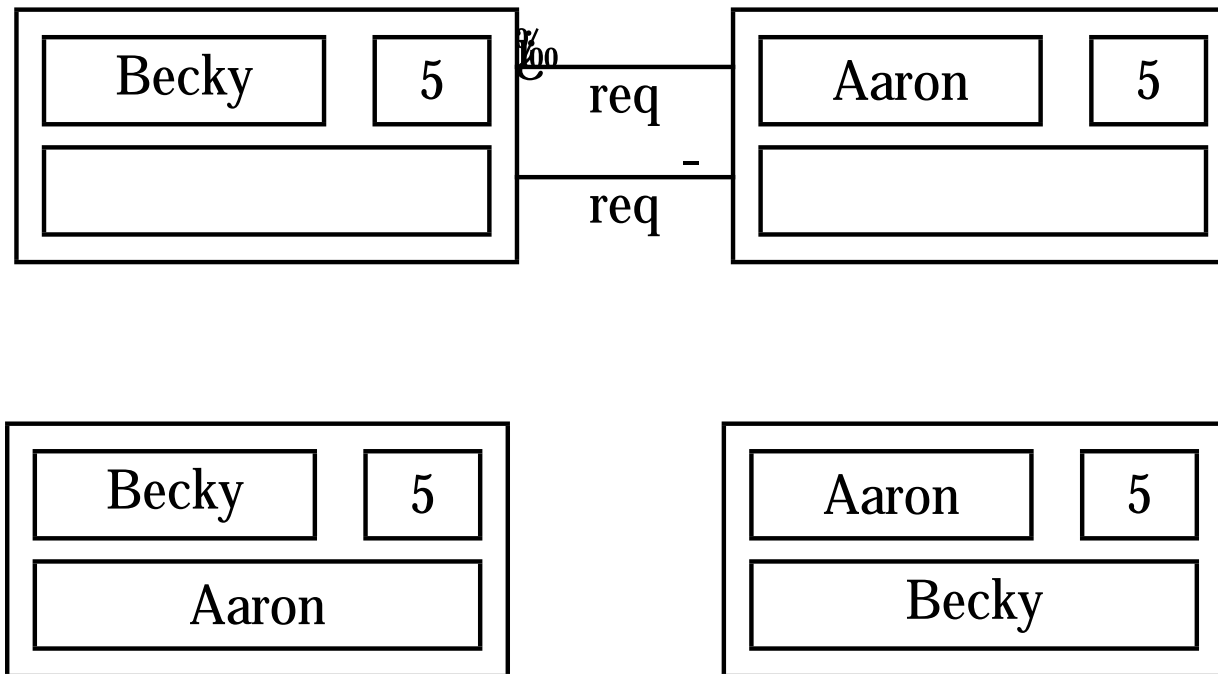
RA Algorithm (3)



RA Algorithm (4)

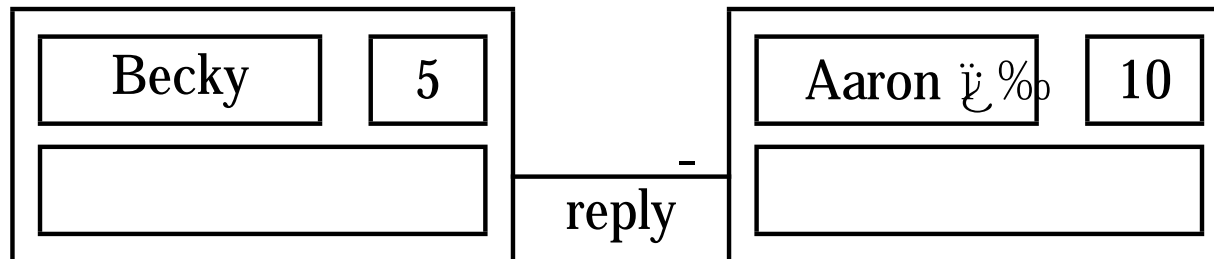
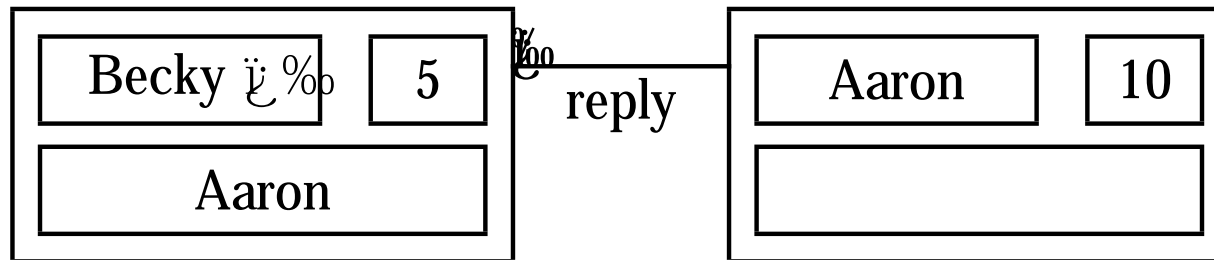
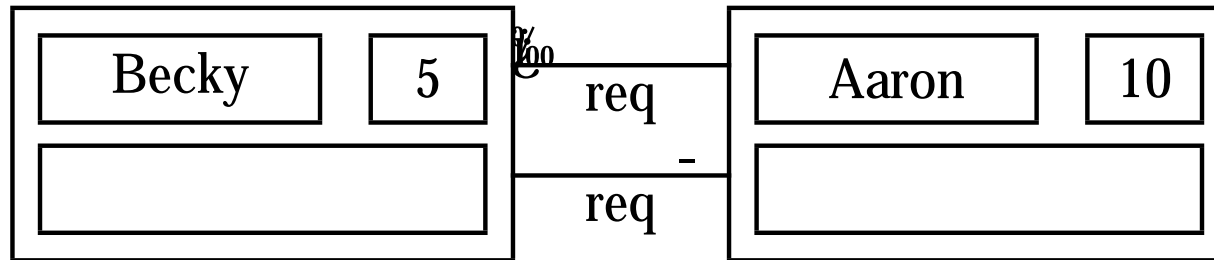


Equal Ticket Numbers

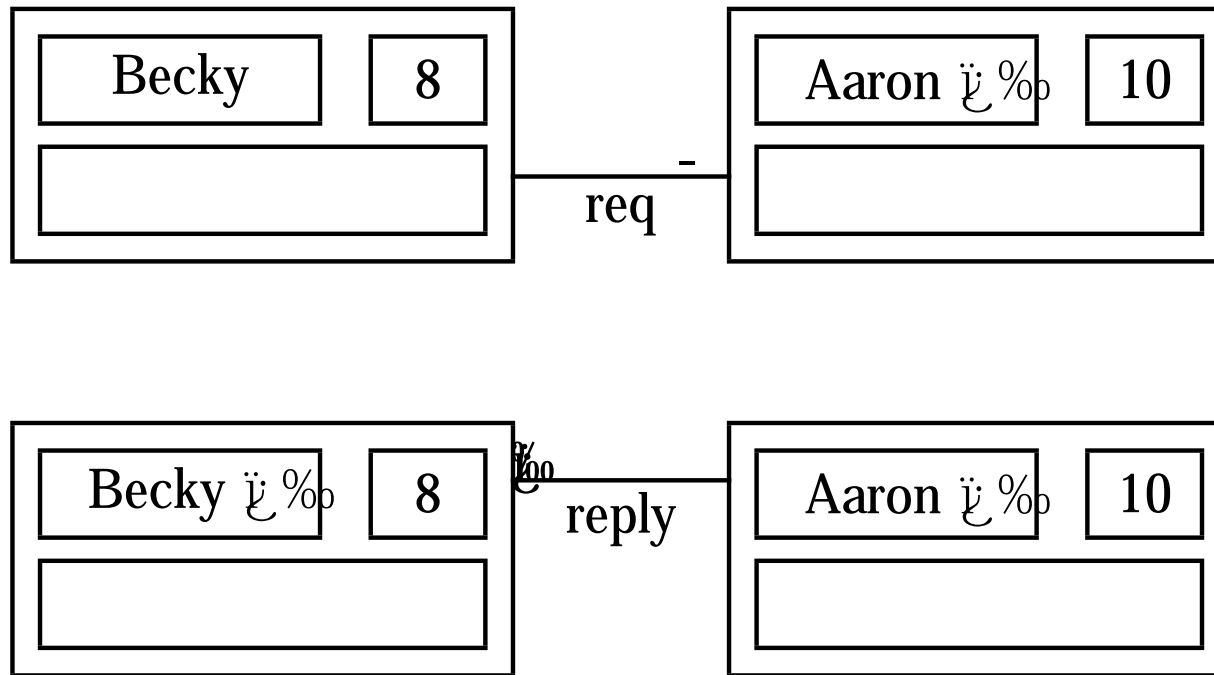


Note: This figure is not in the book.

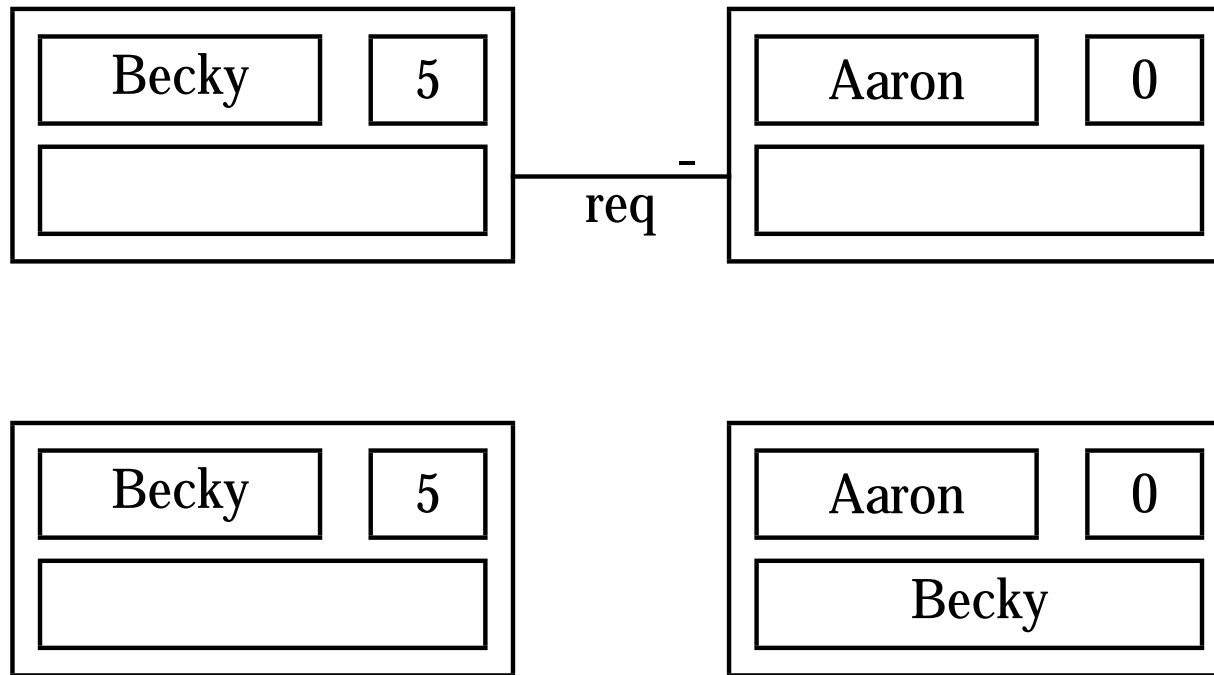
Choosing Ticket Numbers (1)



Choosing Ticket Numbers (2)



Quiescent Nodes



Algorithm 10.2: Ricart-Agrawala algorithm

integer myNum $\in \mathcal{I}$
set of node IDs deferred $\in \emptyset$ set
integer highestNum $\in \mathcal{I}$
boolean requestCS $\in \text{false}$

Main

```
loop forever
p1:   non-critical section
p2:   requestCS  $\in \text{true}$ 
p3:   myNum  $\in \text{highestNum} + 1$ 
p4:   for all other nodes N
p5:     send(request, N, myID, myNum)
p6:   await reply's from all other nodes
p7:   critical section
p8:   requestCS  $\in \text{false}$ 
p9:   for all nodes N in deferred
p10:    remove N from deferred
p11:    send(reply, N, myID)
```

Algorithm 10.2: Ricart-Agrawala algorithm (continued)

Receive

integer source, requestedNum

loop forever

p1: receive(request, source, requestedNum)

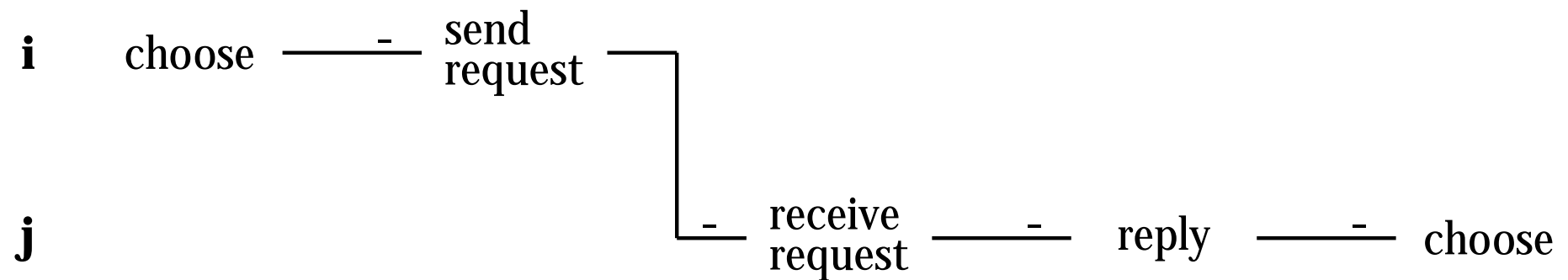
p2: highestNum $\leftarrow \max(\text{highestNum}, \text{requestedNum})$

p3: if not requestCS or requestedNum \neq myNum

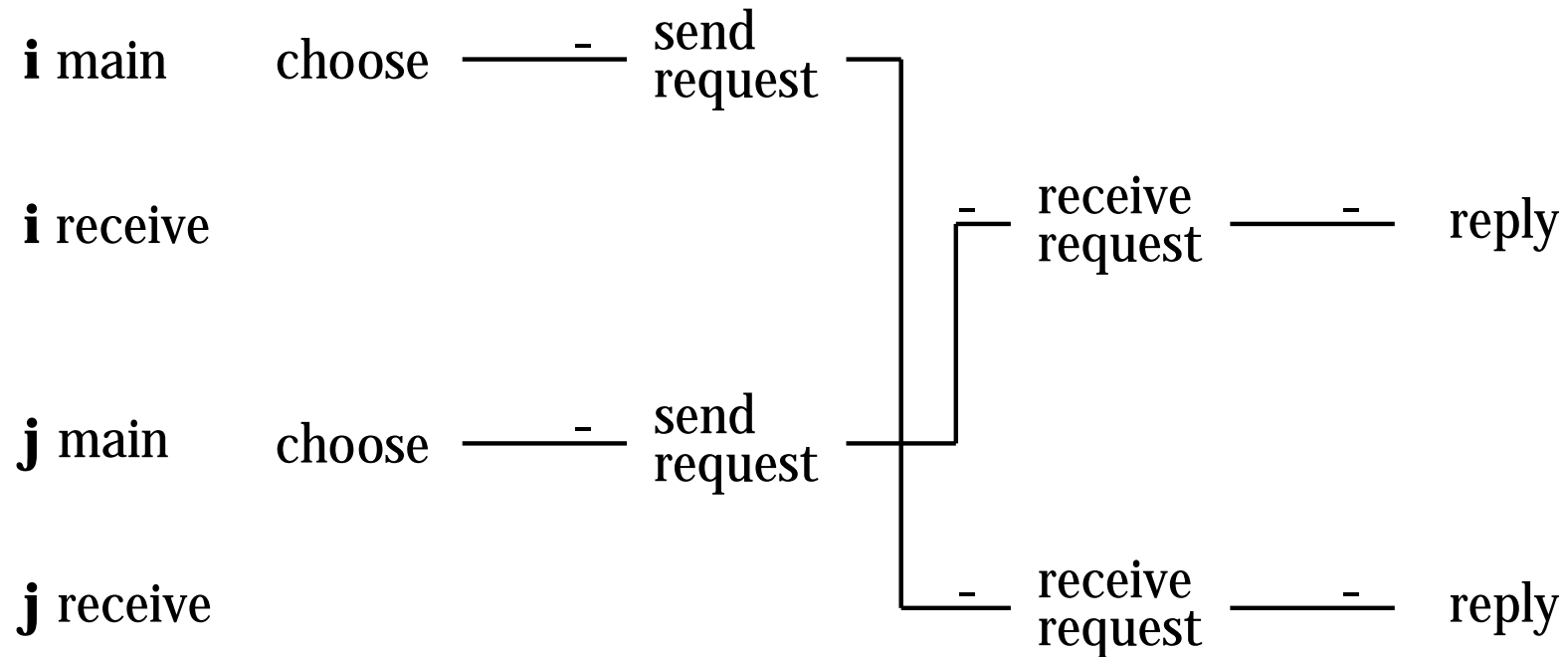
p4: send(reply, source, myID)

p5: else add source to deferred

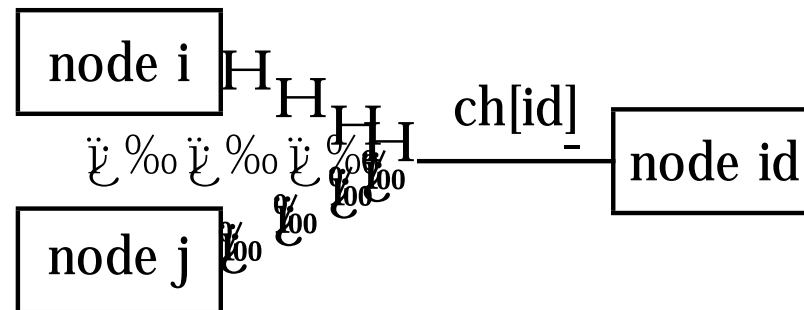
Correct of the RA Algorithm (Case 1)



Correct of the RA Algorithm (Case 2)



Channels in RA Algorithm in Promela



RA Algorithm in Promela

```
1  proctype Main( byte myID ) {
2      do ::
3          atomic {
4              requestCS[myID] = true ;
5              myNum[myID] = highestNum[myID] + 1 ;
6          }
7
8      for (J,0, NPROCS)
9          if
10             :: J != myID
11                 ch[J] ! request, myID, myNum[myID];
12             :: else
13                 %
14             rof (J);
15
```

RA Algorithm in Promela Main Process

```
16
17     for (K,0,NPROCS)
18         ch[myID] ?? reply, _ , _;
19     rof (K);
20     critical_section ();
21     requestCS[myID] = false;
22
23     byte N;
24     do
25         :: empty(deferred[myID]) % break;
26         :: deferred [ myID ] ? N % ch[N] ! reply, 0, 0
27     od
28 od
29 }
```

RA Algorithm in Promela ~~Re~~Receive Process

```

1  proctype Receive( byte myID ) {
2      byte reqNum, source;
3      do ::
4          ch[myID] ?? request, source, reqNum;
5          highestNum[myID] =
6              ((reqNum > highestNum[myID]) ? reqNum : highestNum[myID]);
7
8          atomic {
9              if
10                 :: requestCS[myID] &&
11                     ( myNum[myID] < reqNum ) ||
12                     ( myNum[myID] == reqNum ) &&
13                         (myID < source)
14                 ) ) ? reqNum : highestNum[myID];
15         deferred [ myID ] ! source

```

RA Algorithm in Promela Receive Process

```
16         :: else j % 0
17             ch[source] ! reply , 0, 0
18         % 0
19     }
20 od
21 }
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm

boolean haveToken \hookrightarrow %true in node 0, false in others
integer array[NODES] requested \hookrightarrow % $[0, \dots, 0]$
integer array[NODES] granted \hookrightarrow % $[0, \dots, 0]$
integer myNum \hookrightarrow %0
boolean inCS \hookrightarrow %false

sendToken

if exists N such that requested[N] > granted[N]
 for some such N
 send(token, N, granted)
 haveToken \hookrightarrow %false

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

Main

```
loop forever
p1:   non-critical section
p2:   if not haveToken
p3:       myNum  $\leftarrow$  myNum + 1
p4:       for all other nodes N
p5:           send(request, N, myID, myNum)
p6:       receive(token, granted)
p7:       haveToken  $\leftarrow$  true
p8:       inCS  $\leftarrow$  true
p9:       critical section
p10:      granted[myID]  $\leftarrow$  myNum
p11:      inCS  $\leftarrow$  false
p12:      sendToken
```

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

Receive

integer source, reqNum

loop forever

p13: receive(request, source, reqNum)

p14: requested[source] $\leftarrow \max(\text{requested[source]}, \text{reqNum})$

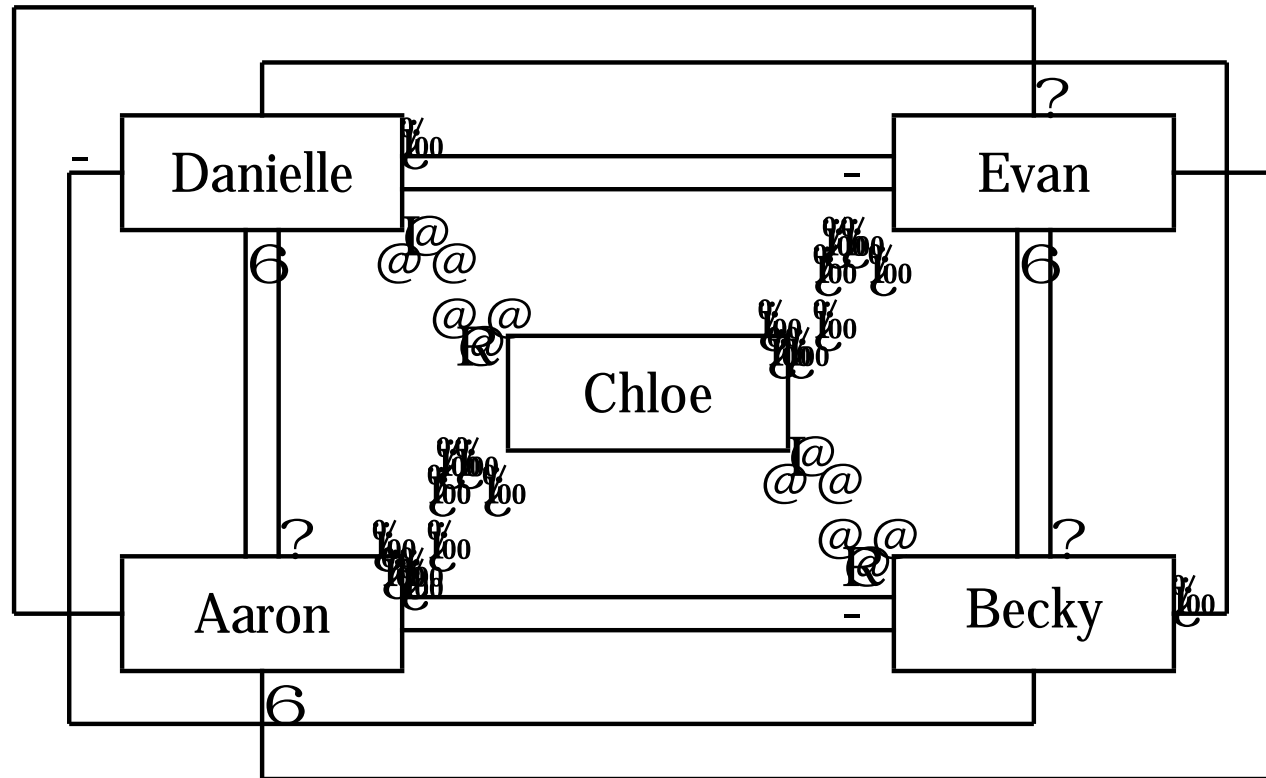
p15: if haveToken and not inCS

p16: sendToken

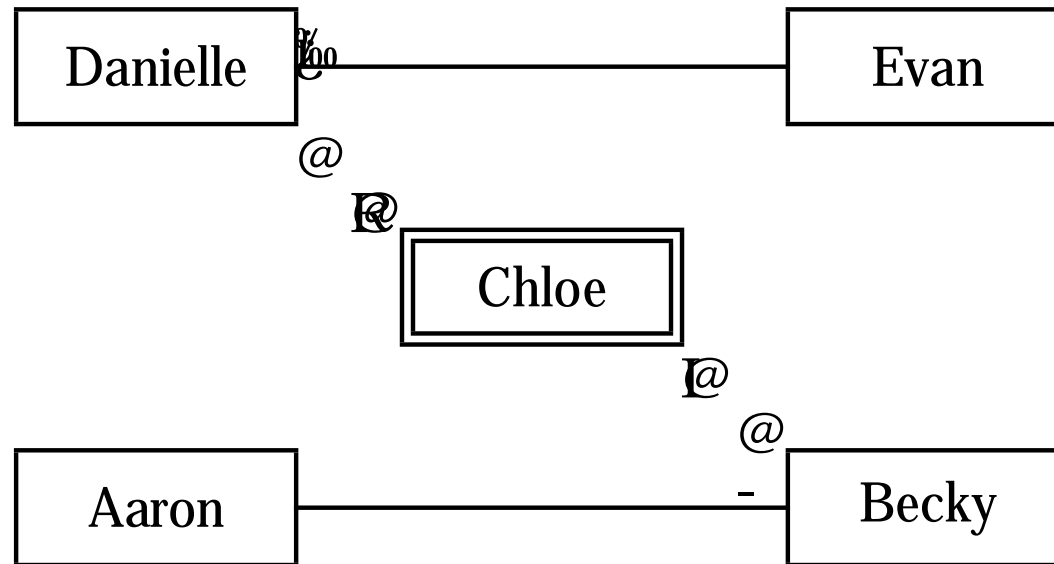
Data Structures for RA Token-Passing Algorithm

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

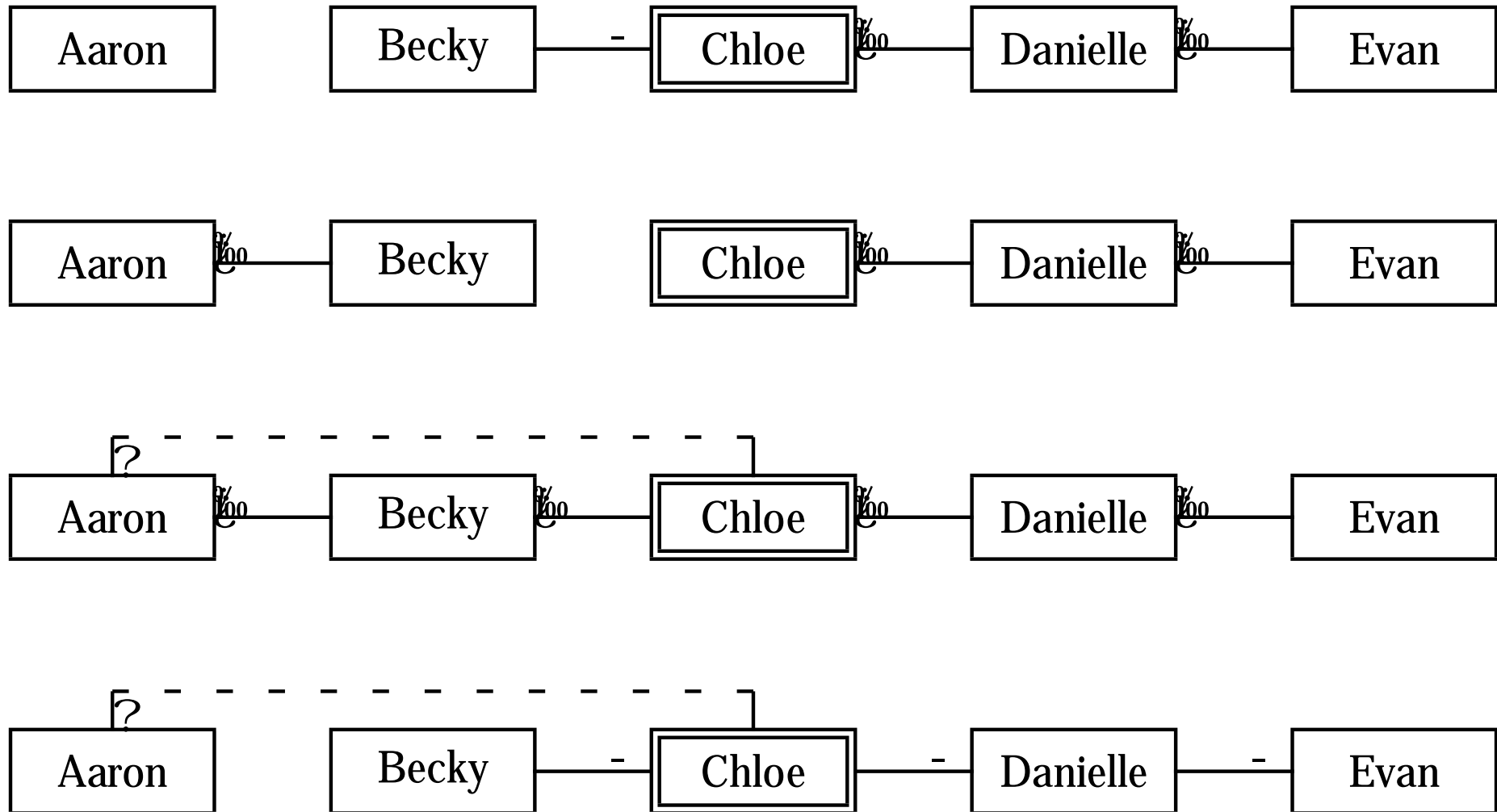
Distributed System for Neilsen-Mizuno Algorithm



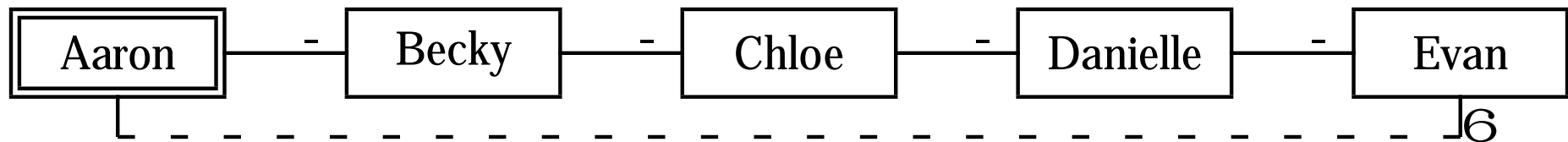
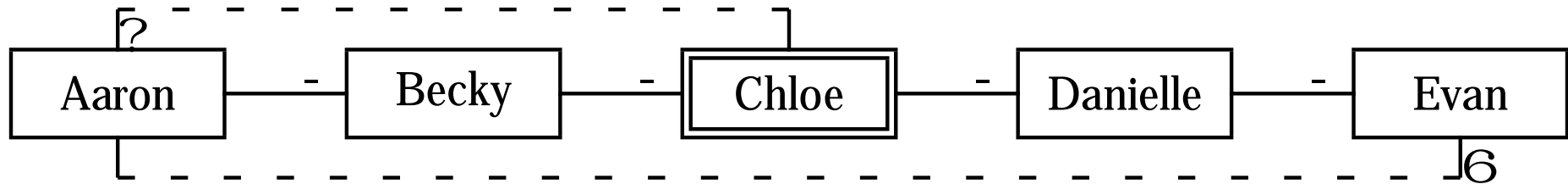
Spanning Tree in Neilsen-Mizuno Algorithm



Neilsen-Mizuno Algorithm (1)



Neilsen-Mizuno Algorithm (2)



Algorithm 10.4: Neilsen-Mizuno token-passing algorithm

integer parent $\in \mathcal{M}$ (initialized to form a tree)
integer deferred $\in \mathcal{M}$
boolean holding $\in \mathcal{B}$ true in the root, false in others

Main

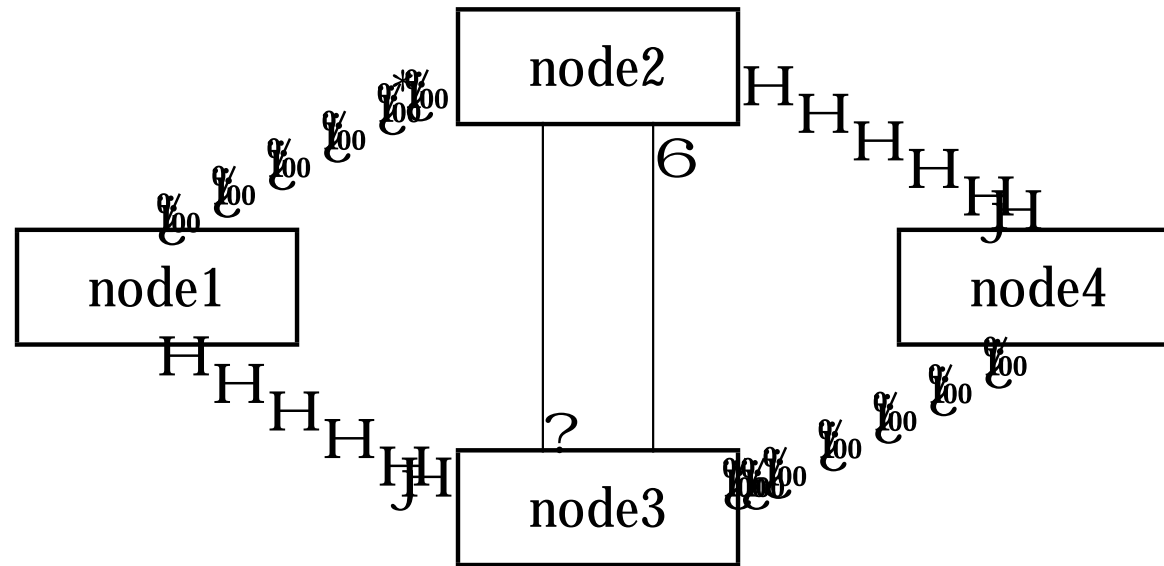
```
loop forever
p1:   non-critical section
p2:   if not holding
p3:     send(request, parent, myID, myID)
p4:     parent  $\leftarrow$  myID
p5:     receive(token)
p6:   holding  $\leftarrow$  true
p7:   critical section
p8:   if deferred  $\neq 0$ 
p9:     send(token, deferred)
p10:    deferred  $\leftarrow$  deferred + 1
p11:  else holding  $\leftarrow$  false
```

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

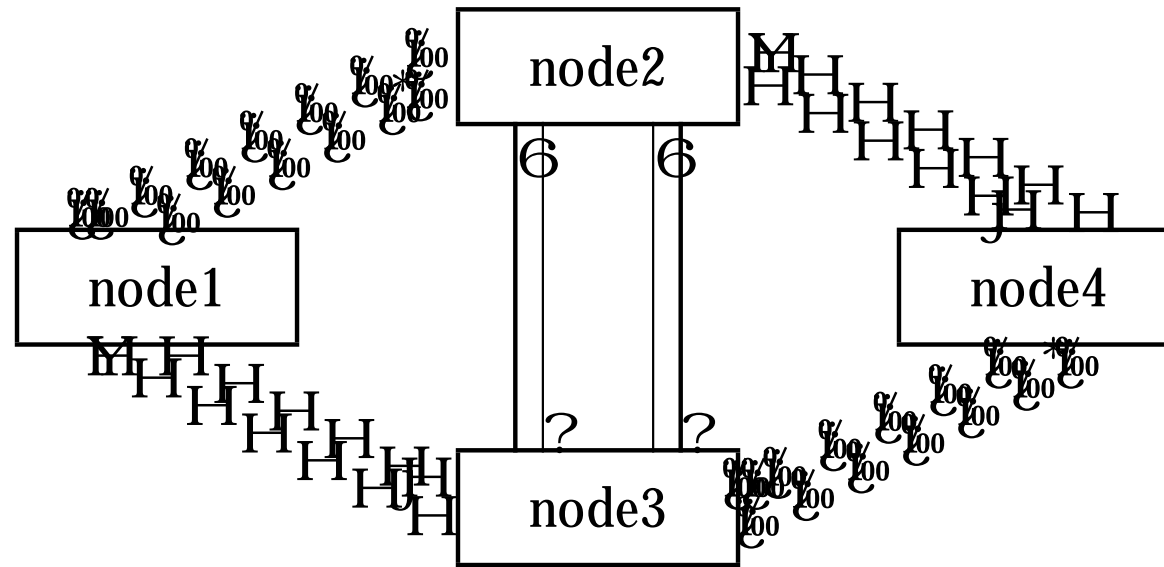
Receive

```
integer source, originator
loop forever
p12:  receive(request, source, originator)
p13:  if parent = 0
p14:    if holding
p15:      send(token, originator)
p16:      holding := false
p17:    else deferred := originator
p18:  else send(request, parent, myID, originator)
p19:  parent := source
```

Distributed System with an Environment Node



Back Edges



Algorithm 11.1: Dijkstra-Scholten algorithm (preliminary)	
integer array[incoming] inDegree $\in [0, \dots, 0]$ integer inDegree $\in [0, \dots, 0]$, integer outDegree $\in [0, \dots, 0]$	
send message	
p1:	send(message, destination, myID)
p2:	increment outDegree
receive message	
p3:	receive(message, source)
p4:	increment inDegree[source] and inDegree
send signal	
p5:	when inDegree > 1 or (inDegree = 1 and isTerminated and outDegree = 0)
p6:	\exists some edge E with inDegree[E] $\in 0$
p7:	send(signal, E, myID)
p8:	decrement inDegree[E] and inDegree
receive signal	
p9:	receive(signal, _)
p10:	decrement outDegree

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

integer outDegree; $\forall i \in \mathcal{N}$

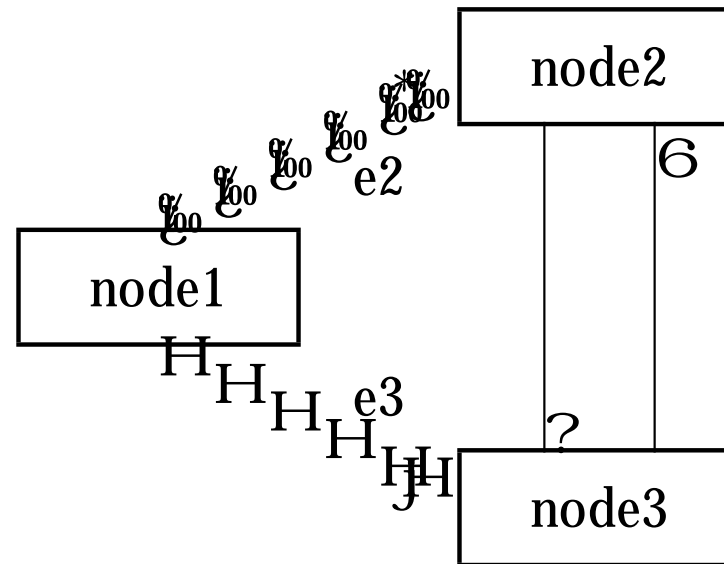
computation

p1: for all outgoing edges E
p2: send(message, E, myID)
p3: increment outDegree
p4: await outDegree = 0
p5: announce system termination

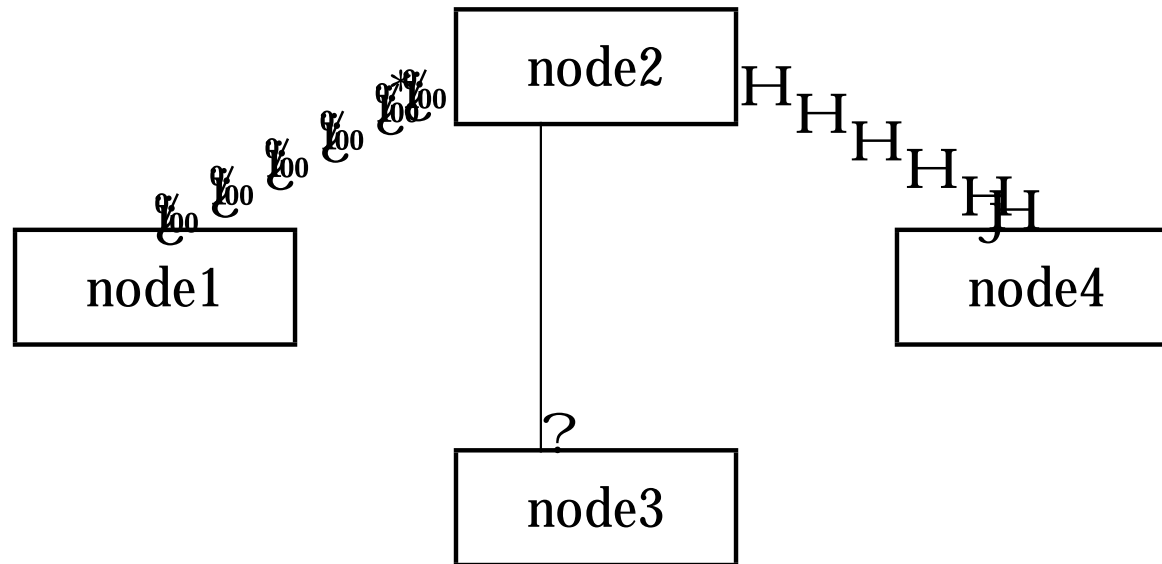
receive signal

p6: receive(signal, source)
p7: decrement outDegree

The Preliminary DS Algorithm is Unsafe



Spanning Tree



Algorithm 11.3: Dijkstra-Scholten algorithm

```

integer array[incoming] inDegree[i] % [0,...,0]
integer inDegree[i] % 0
integer outDegree[i] % 0
integer parent[i] % 1 % 0
    
```

send message

```

p1: when parent[i] < 1 % 0 // Only active nodes send messages
p2:   send(message, destination, myID)
p3:   increment outDegree[i]
    
```

receive message

```

p4: receive(message,source)
p5: if parent[i] = 1 % 0
p6:   parent[i] = source
p7: increment inDegree[i][source] and inDegree[i]
    
```

Algorithm 11.3: Dijkstra-Scholten algorithm (continued)

send signal

```

p8:  when inDegree > 1
p9:    E ← some edge E for which
        (inDegree[E] > 1) or (inDegree[E] = 1 and E ∈ parent)
p10:  send(signal, E, myID)
p11:  decrement inDegree[E] and inDegree
p12:  or when inDegree = 1 and isTerminated and outDegree = 0
p13:    send(signal, parent, myID)
p14:    inDegree[parent] ← 0
p15:    inDegree ← 0
p16:    parent ← 1
    
```

receive signal

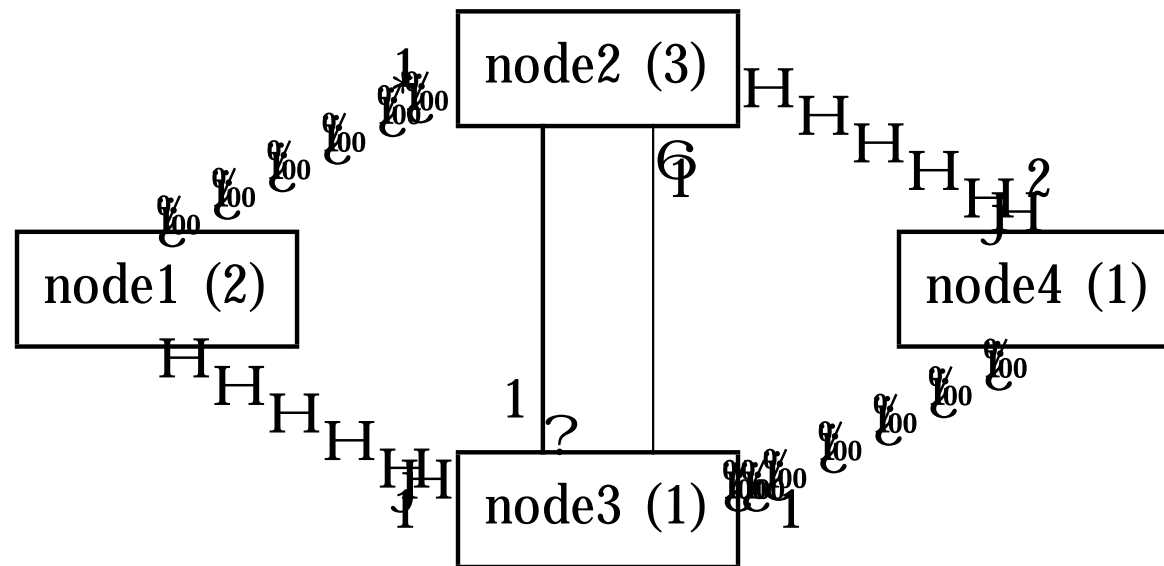
```

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

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)

let weight $\leftarrow 1.0$

computation

p1: for all outgoing edges E
 p2: weight \leftarrow 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 \leftarrow weight + w

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

constant integer parent $\Leftarrow 0$ // Environment node

boolean active $\Leftarrow \text{false}$

float weight $\Leftarrow 0.0$

send message

p1: if active // Only active nodes send messages

p2: weight $\Leftarrow \text{weight} / 2.0$

p3: send(message, destination, myID, weight)

receive message

p4: receive(message, source, w)

p5: active $\Leftarrow \text{true}$

p6: weight $\Leftarrow \text{weight} + w$

send signal

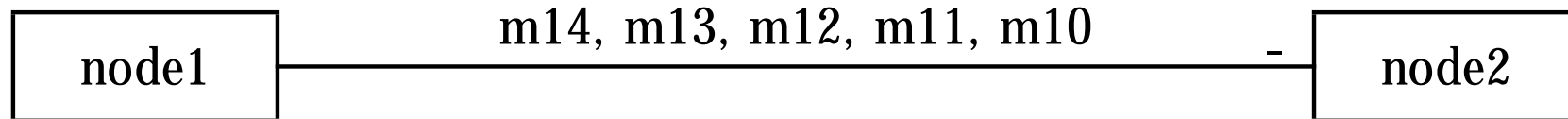
p7: when terminated

p8: send(signal, parent, weight)

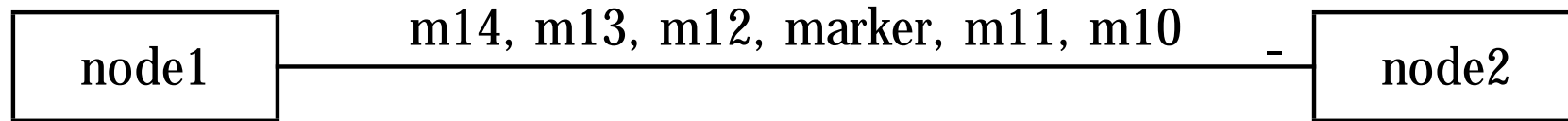
p9: weight $\Leftarrow 0.0$

p10: active $\Leftarrow \text{false}$

Messages on a Channel



Sending a Marker



Algorithm 11.6: Chandy-Lamport algorithm for global snapshots

integer array[outgoing] lastSent $\leftarrow [0, \dots, 0]$
 integer array[incoming] lastReceived $\leftarrow [0, \dots, 0]$
 integer array[outgoing] stateAtRecord $\leftarrow [0, \dots, 0]$
 integer array[incoming] messageAtRecord $\leftarrow [0, \dots, 0]$
 integer array[incoming] messageAtMarker $\leftarrow [0, \dots, 0]$

send message

p1: send(message, destination, myID)
 p2: lastSent[destination] \leftarrow message

receive message

p3: receive(message, source)
 p4: lastReceived[source] \leftarrow message

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

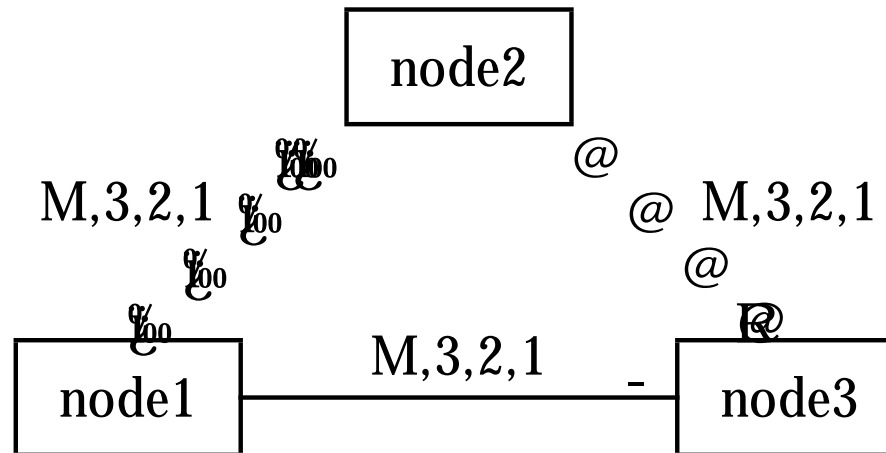
receive marker

```
p6: receive(marker, source)
p7: messageAtMarker[source] ← %lastReceived[source]
p8: if stateAtRecord = [← %0 .., ← %0] // 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



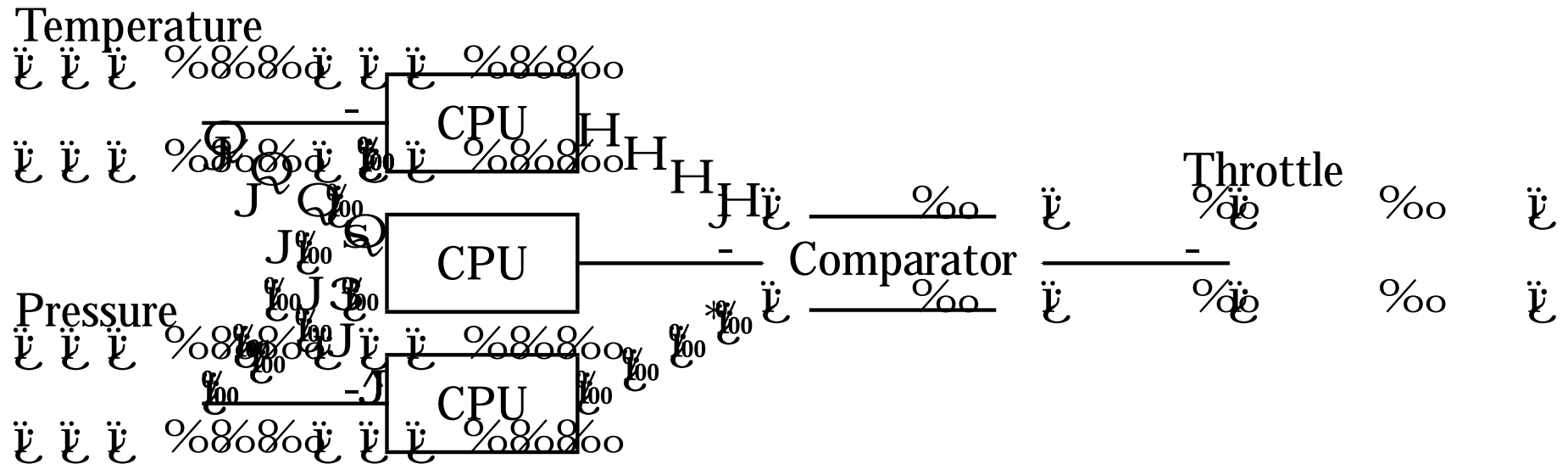
Scenario for CL Algorithm (1)

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



Algorithm 12.1: Consensus - one-round algorithm

```

planType ValPlan
planType array[generals] plan

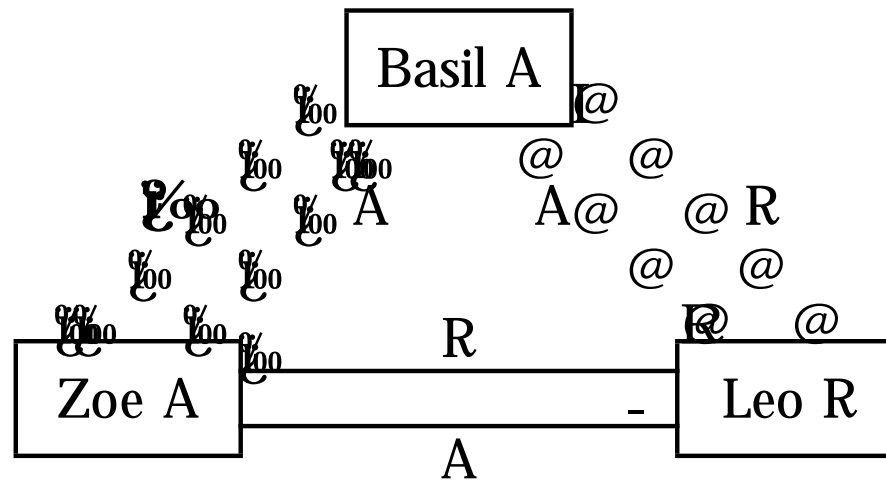
```

```

p1: plan[myID] ∈ %chooseAttackOrRetreat
p2: for all other generals G
p3:   send(G, myID, plan[myID])
p4: for all other generals G
p5:   receive(G, plan[G])
p6: ValPlan ∈ %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	100
Leo	R
Zoe	A
majority	R

Algorithm 12.2: Consensus - Byzantine Generals algorithm

```

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

```

p1: plan[myID] ← %chooseAttackOrRetreat
p2: for all other generals G                                // First round
p3:   send(G, myID, plan[myID])
p4: for all other generals G
p5:   receive(G, plan[G])
p6: for all other generals G                                // Second round
p7:   for all other generals G' except G
p8:     send(G', myID, G, plan[G])
p9: for all other generals G
p10:  for all other generals G' except G
p11:   receive(G, G', reportedPlan[G, G'])
p12: for all other generals G                                // First vote
p13:  majorityPlan[G] ← %majority(plan[G] [ reportedPlan[*, G])
p14: majorityPlan[myID] ← %plan[myID]                       // Second vote
p15: ValPlan ← %majority(majorityPlan)
    
```

Data Structure for Crash Failure - First Scenario (Leo)

Leo				
general	plan	reported by		majority
		Basil	Zoe	
Basil	A		100	A
Leo	R			R
Zoe	A	100		A
majority				A

Data Structure for Crash Failure - First Scenario (Zoe)

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

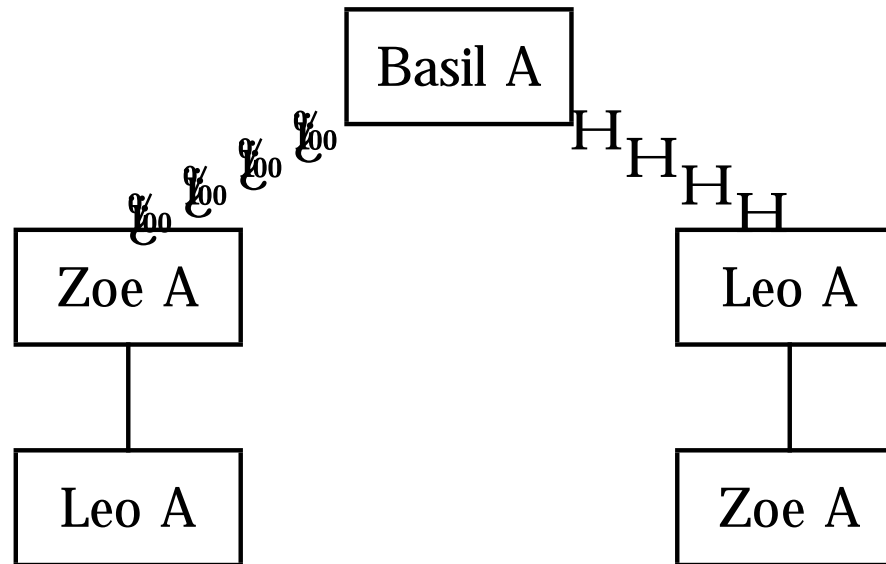
Data Structure for Crash Failure - Second Scenario (Leo)

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

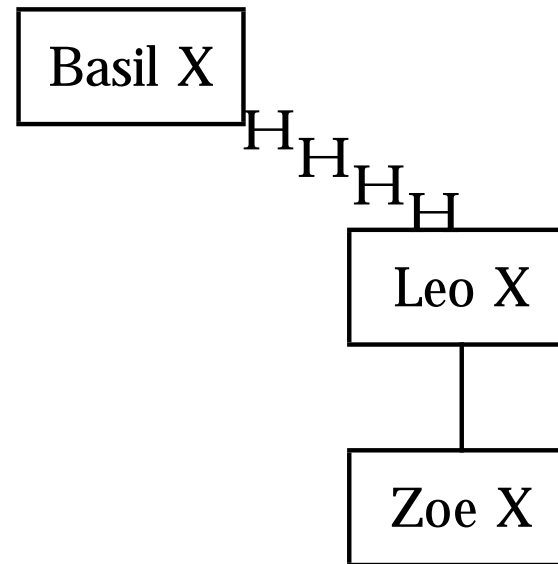
Data Structure for Crash Failure - Second Scenario (Zoe)

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

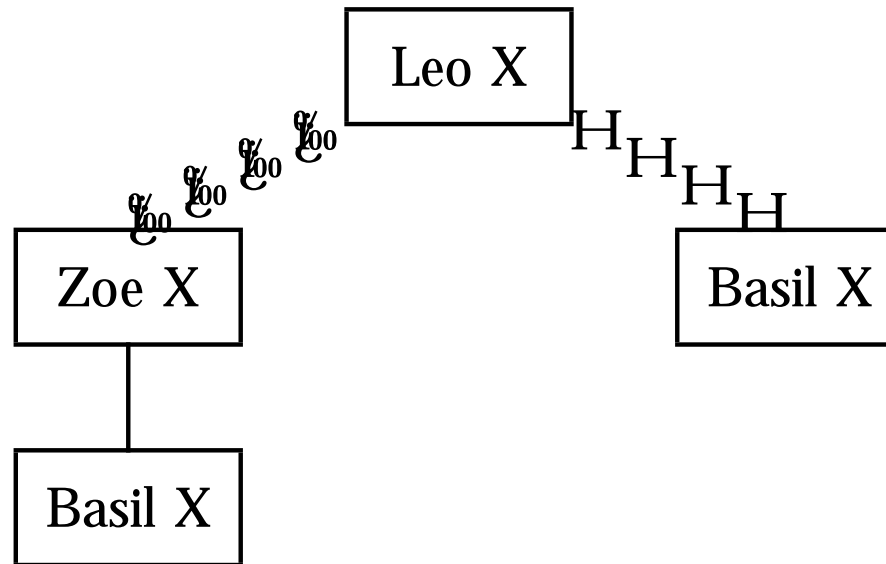
Knowledge Tree about Basil for Crash Failure - First Scenario



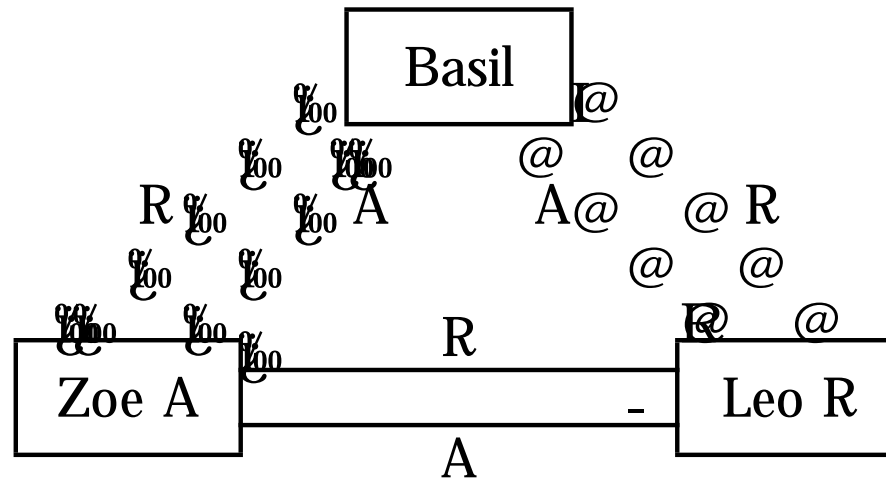
Knowledge Tree about Basil for Crash Failure - Second Scenario



Knowledge Tree about Leo for Crash Failure



Messages Sent for Byzantine Failure with Three Generals



Data Structures 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

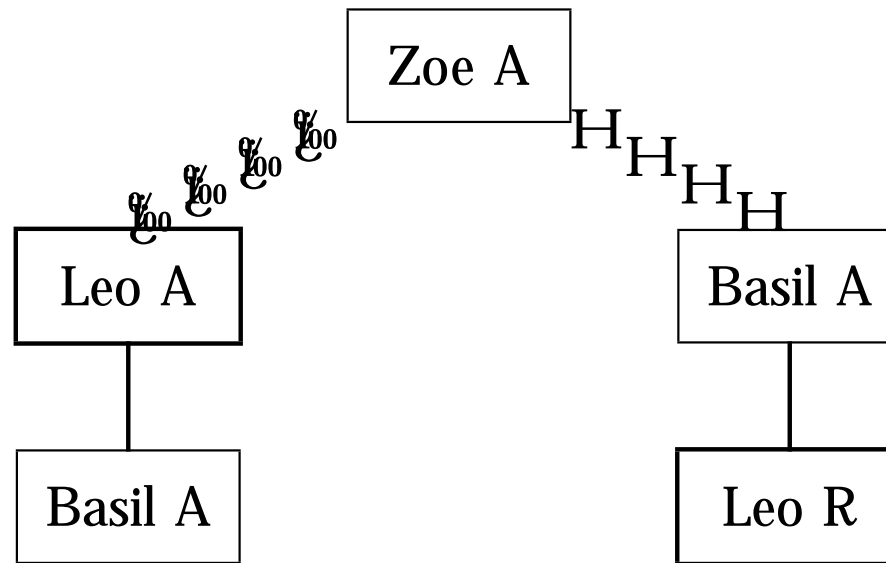
Data Structures for Leo After Second Round

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

Data Structures 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



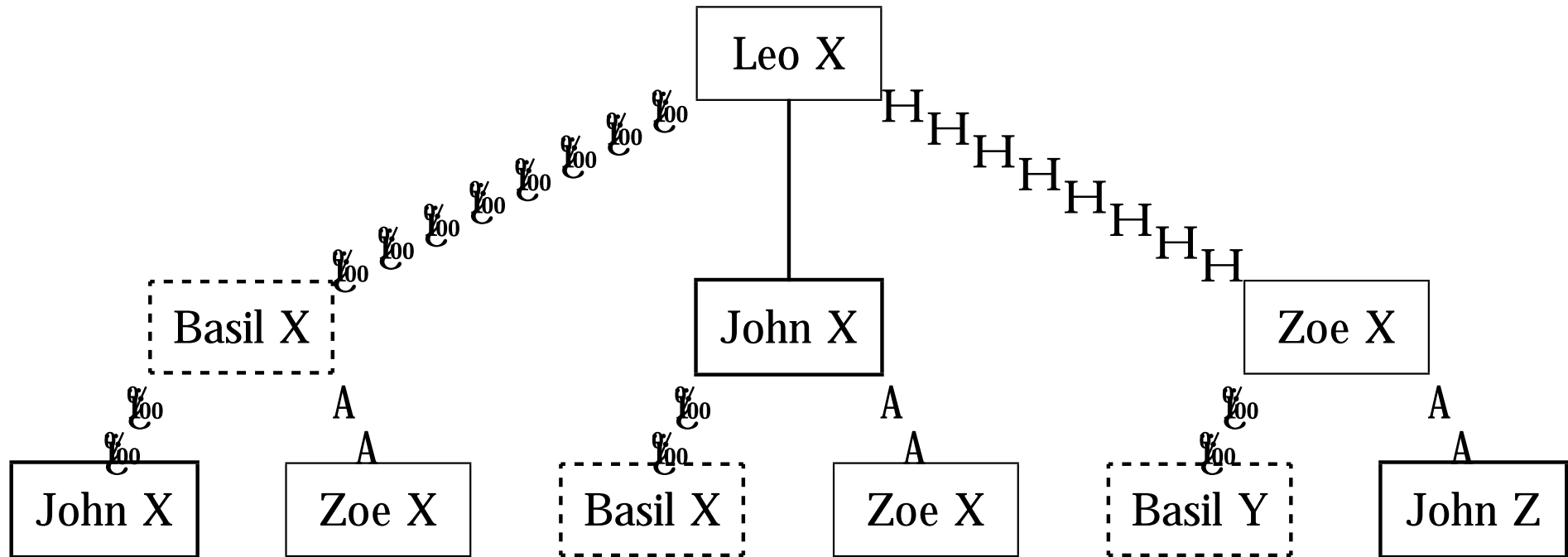
Four Generals: Data Structure of Basil (1)

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

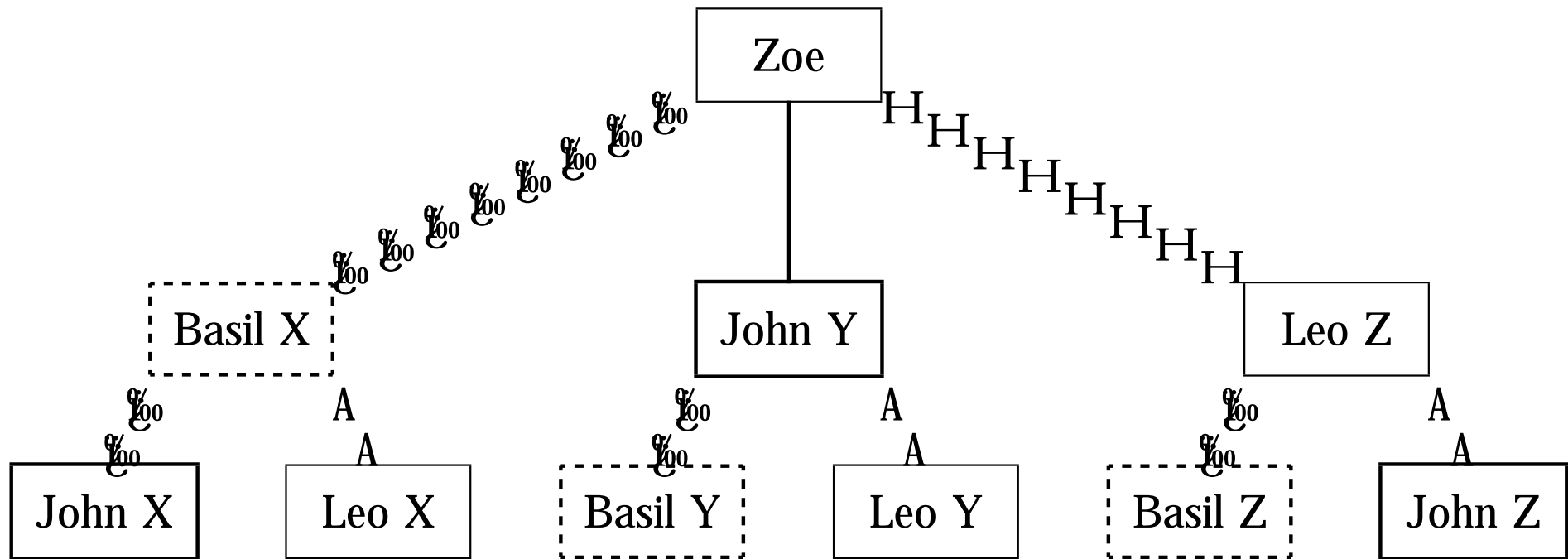
Four Generals: Data Structure of Basil (2)

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



Knowledge Tree About Traitor Zoe



Complexity of the Byzantine Generals Algorithm

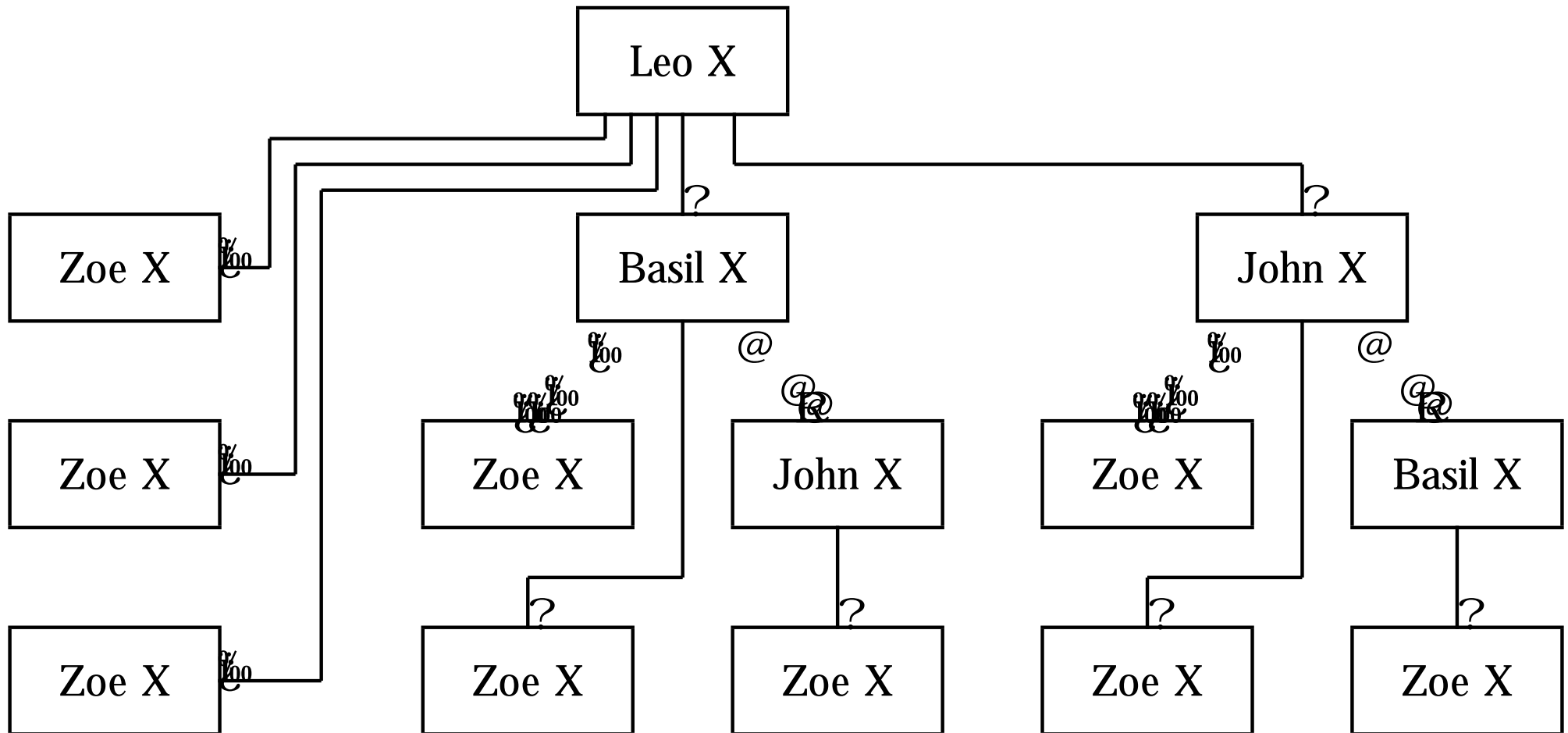
traitors	generals	messages
1	4	36
2	7	392
3	10	1790
4	13	5408

Algorithm 12.3: Consensus - Voting algorithm

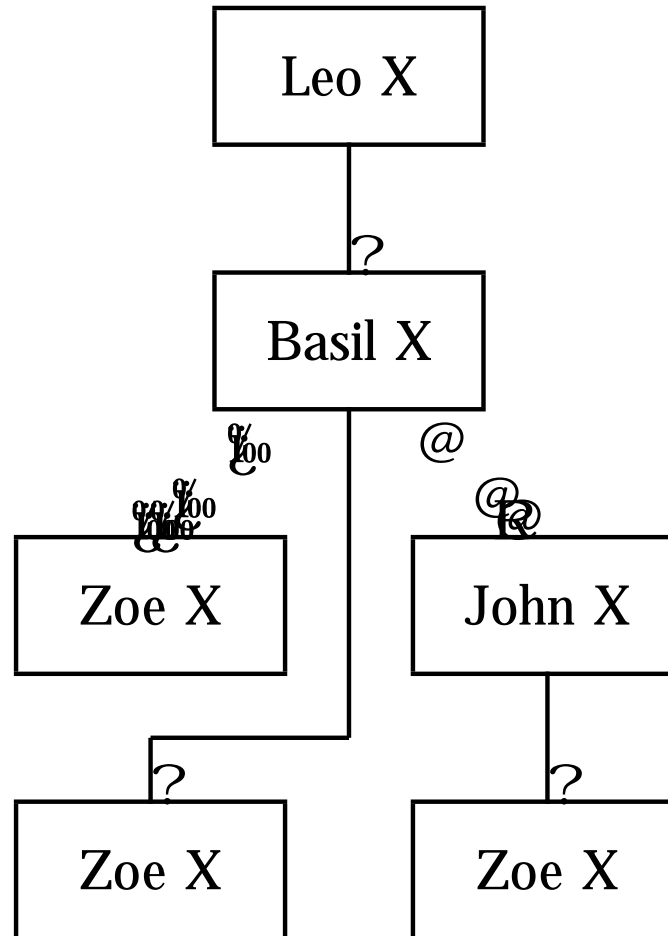
~~planType~~ ValPlan
 set of planType plan $\dot{\cup}$ chooseAttackOrRetreat g
 set of planType receivedPlan

p1: do $t + 1$ times
 p2: for all other generals G
 p3: send(G, plan)
 p4: for all other generals G
 p5: receive(G, receivedPlan)
 p6: plan $\dot{\cup}$ plan [receivedPlan
 p7: ~~ValPlan~~ $\dot{\cup}$ majority(plan)

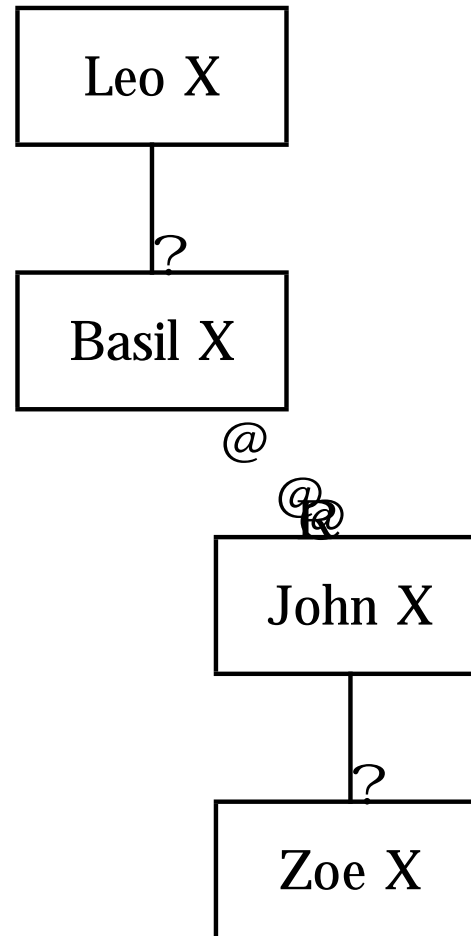
Flooding Algorithm with No Crash: Knowledge Tree About Leo



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



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



Algorithm 12.4: Consensus - King algorithm

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

```
p1: plan[myID]  $\leftarrow$  chooseAttackOrRetreat  
  
p2: do two times  
p3:   for all other generals G           // First and third rounds  
p4:     send(G, myID, plan[myID])  
p5:   for all other generals G  
p6:     receive(G, plan[G])  
p7:   myMajority  $\leftarrow$  majority(plan)  
p8:   votesMajority  $\leftarrow$  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: plan := 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

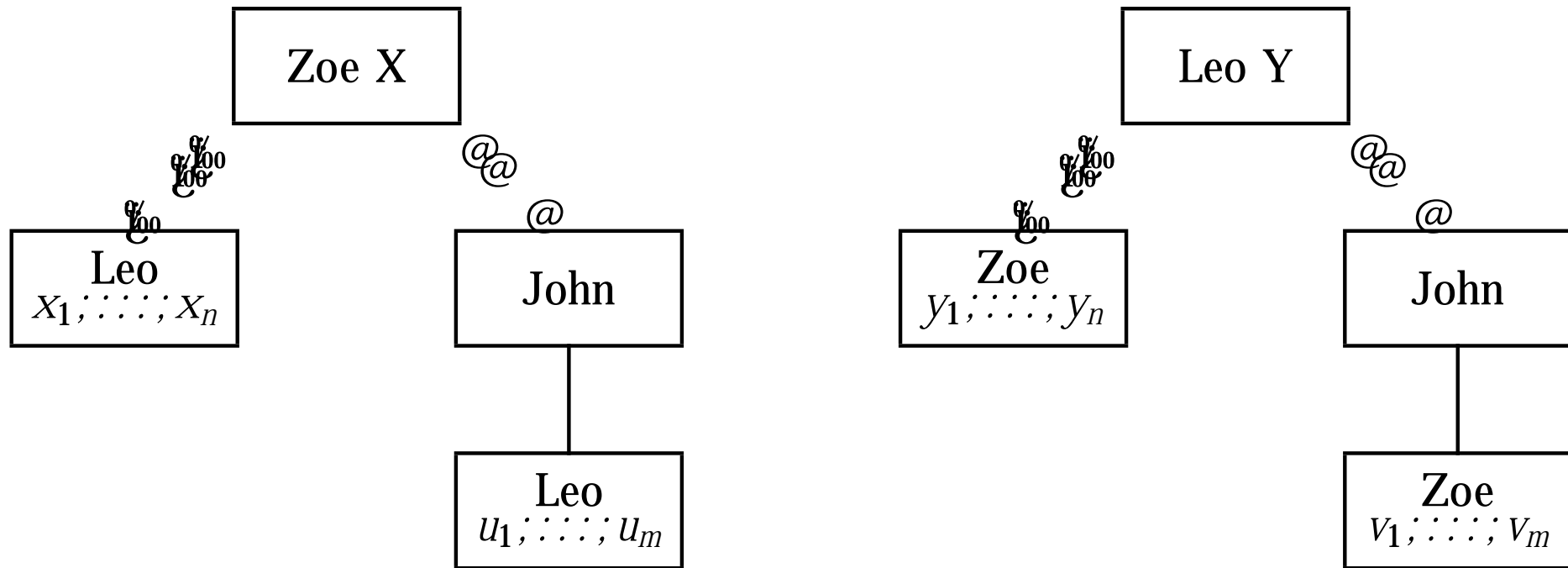
Zoe							
Basil	John	Leo	Mike	Zoe	myMajority	votesMajority	kingPlan
				A			

Complexity of Byzantine Generals and King Algorithms

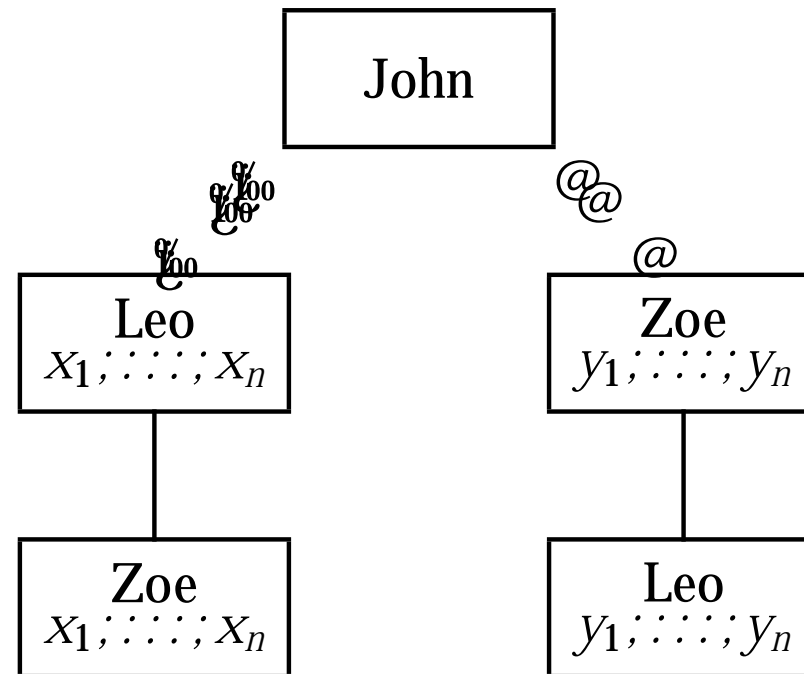
traitors	generals	messages
1	4	36
2	7	392
3	10	1790
4	13	5408

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

Impossibility with Three Generals (1)



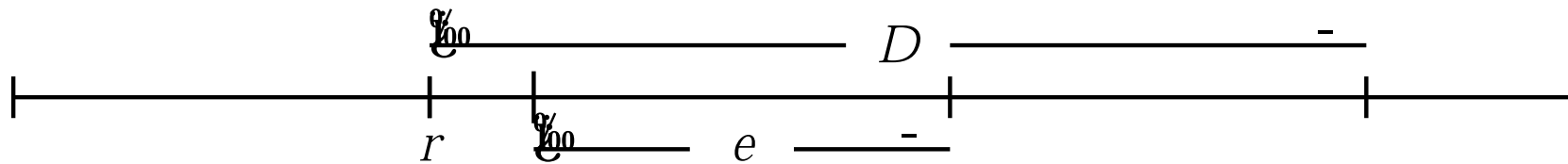
Impossibility with Three Generals (2)



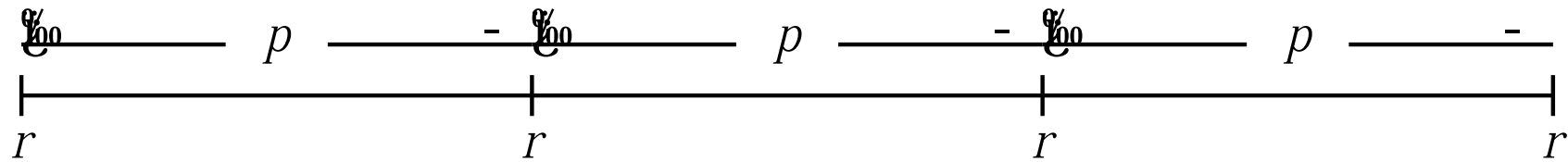
Exercise for Byzantine Generals Algorithm

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

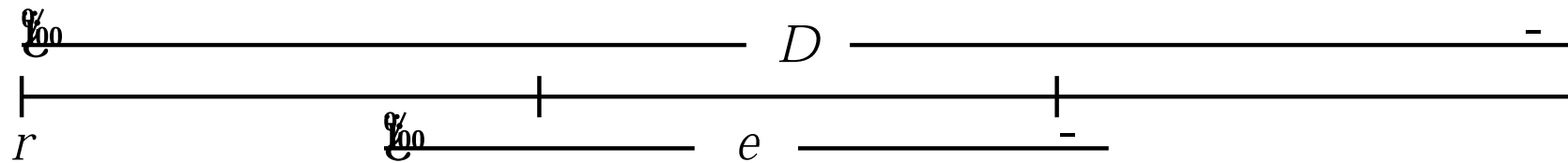
Release Time, Execution Time and Relative Deadline



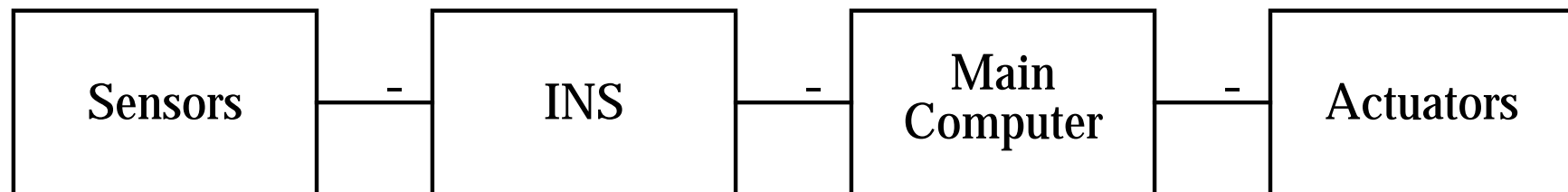
Periodic Task



Deadline is a Multiple of the Period



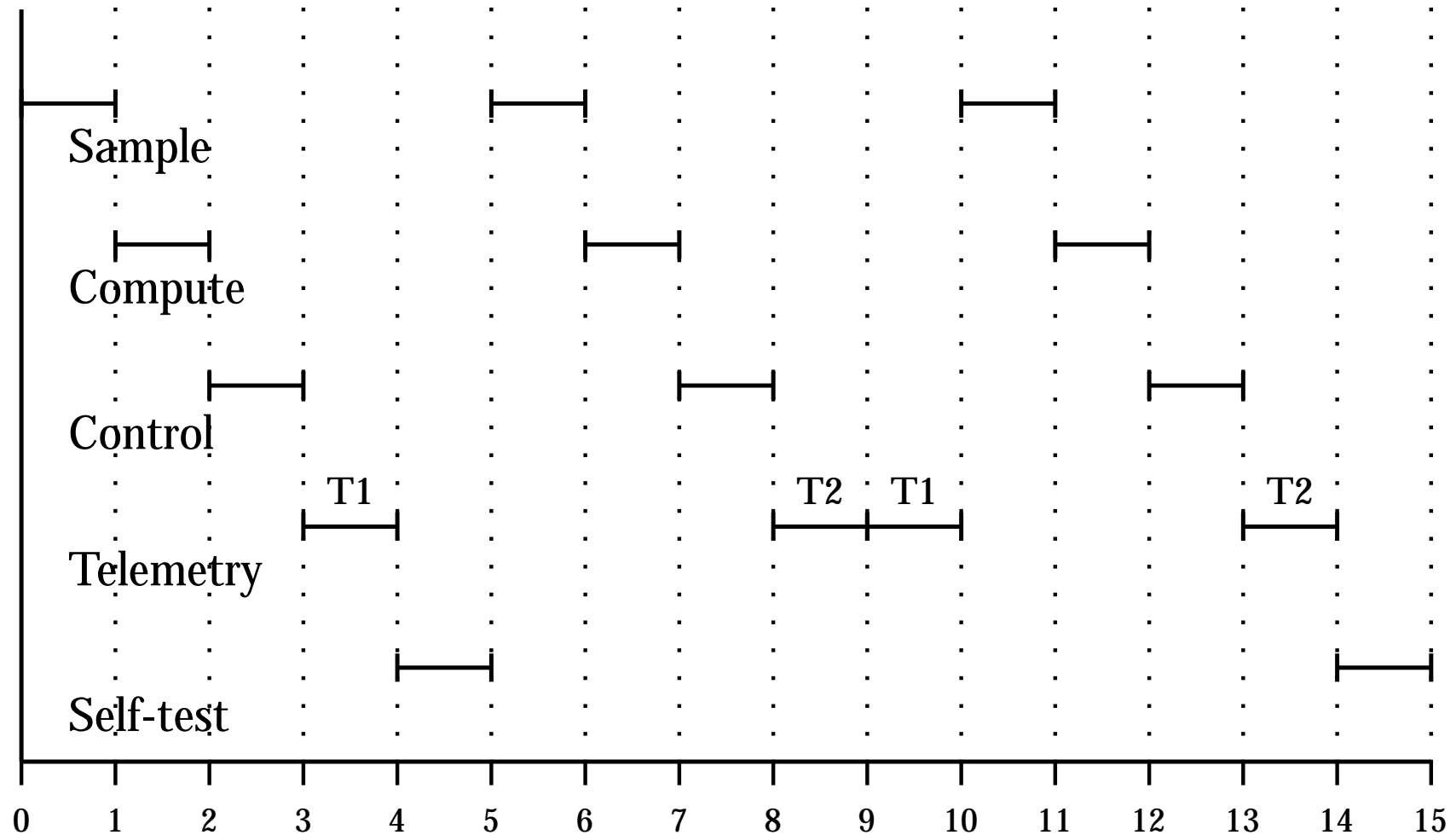
Architecture of Ariane Control System



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 $\%0$
[task address,...,task address]
integer currentFrame $\%0$

p1: loop

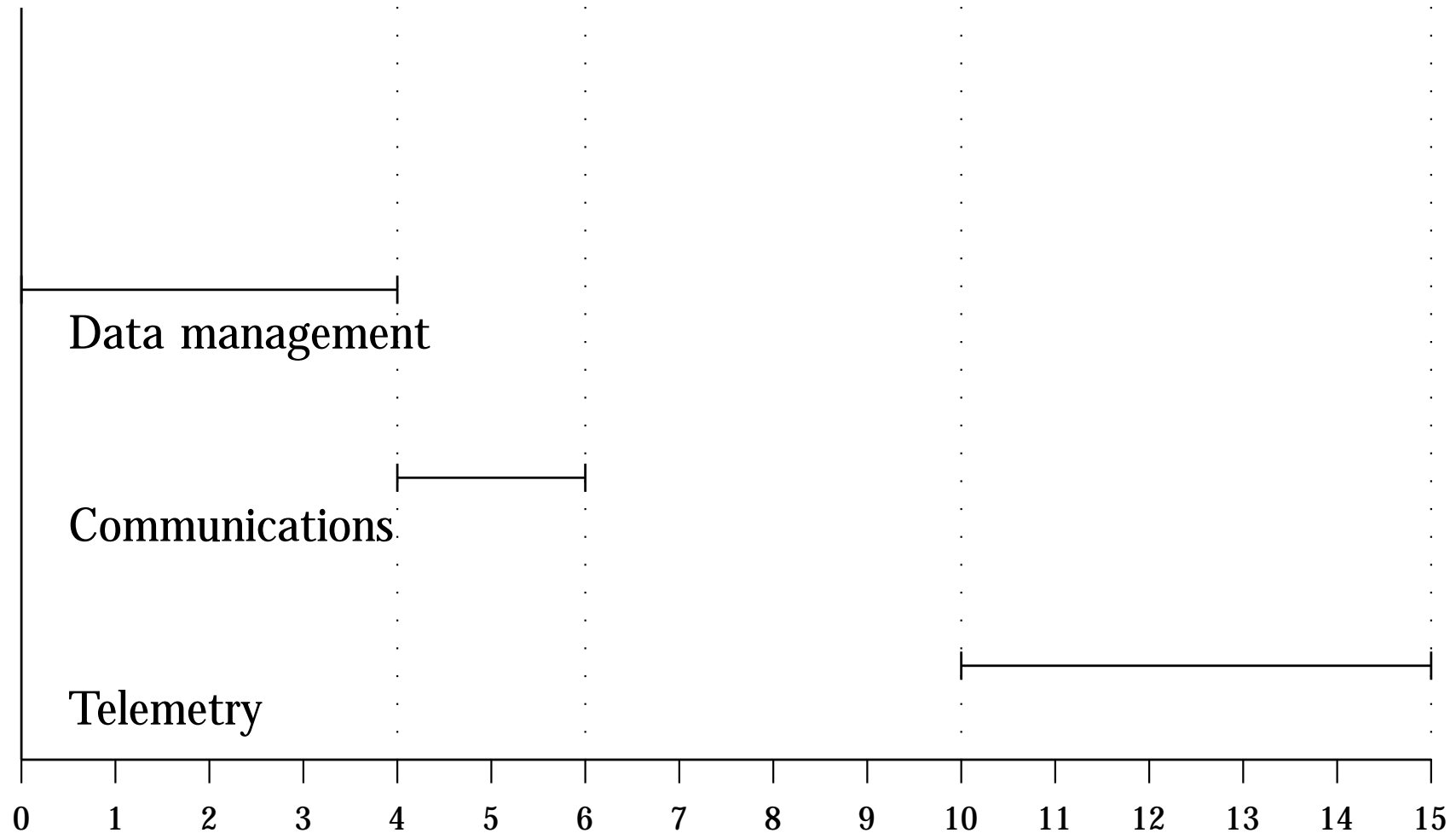
p2: await beginning of frame

p3: invoke tasks[currentFrame]

p4: increment currentFrame modulo numberFrames

Algorithm 13.2: Producer-consumer (synchronous system)		
queue of dataType buf1, buf2		
sample	compute	control
dataType d p1: d %sample p2: append(d, buf1) p3:	dataType d1, d2 q1: d1 %take(buf1) q2: d2 %compute(d1) q3: append(d2, buf2)	dataType d r1: d %take(buf2) r2: control(d) 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

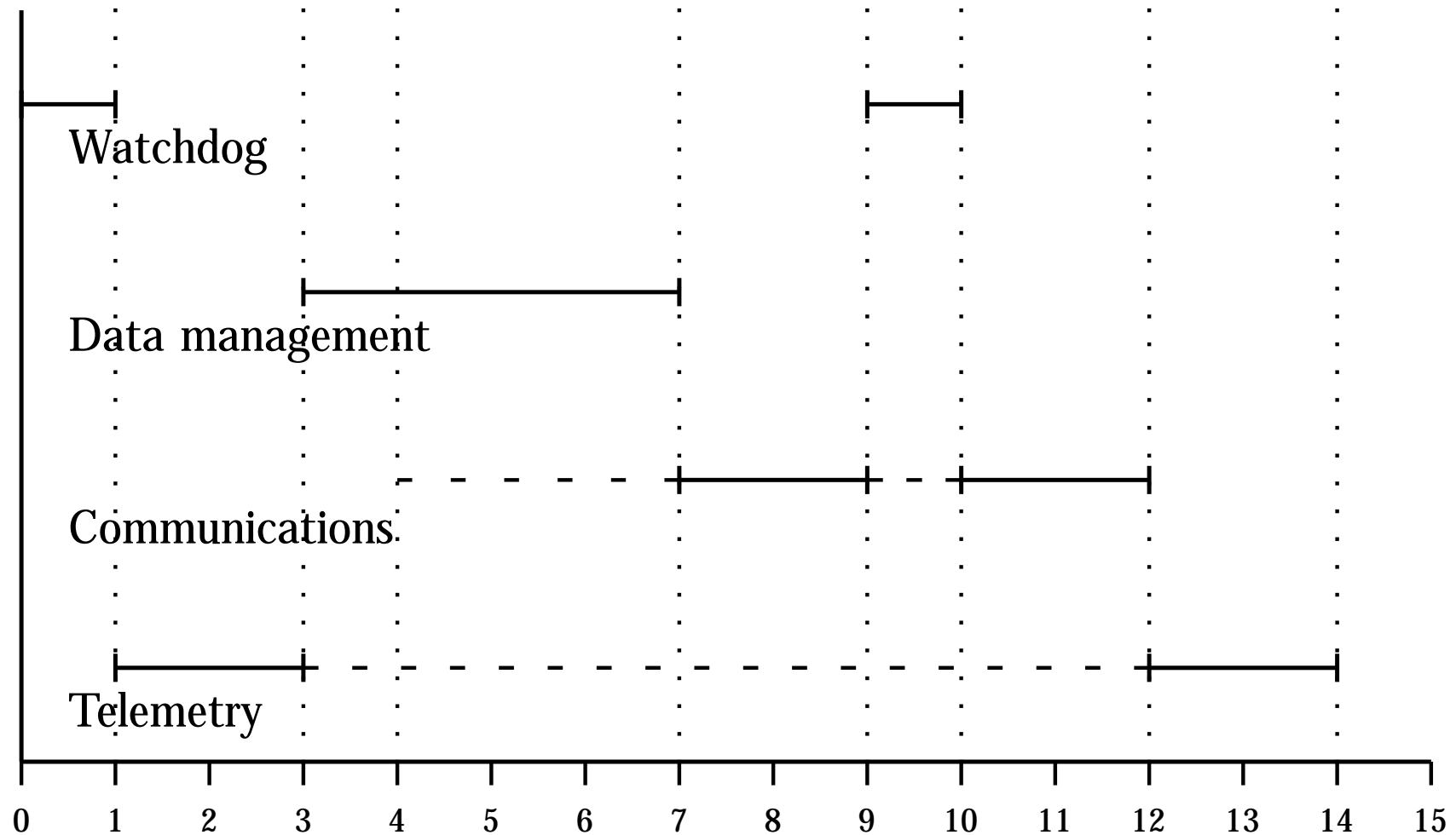
Algorithm 13.4: Preemptive scheduler

queue of taskAddressType readyQueue
taskAddressType currentTask

loop forever

p1: await a scheduling event
p2: if currentTask.priority < highest priority of a task on readyQueue
p3: save partial computation of currentTask and place on readyQueue
p4: currentTask = take task of highest priority from readyQueue
p5: invoke currentTask
p6: else if currentTask's timeslice is past and
 currentTask.priority = priority of some task on readyQueue
p7: save partial computation of currentTask and place on readyQueue
p8: currentTask = take a task of the same priority from readyQueue
p9: invoke currentTask
p10: else resume currentTask

Preemptive Scheduling



Algorithm 13.5: Watchdog supervision of response time

boolean ran \leftarrow false

data management

loop forever

p1: do data management

p2: ran \leftarrow true

p3: rejoin readyQueue

p4:

p5:

watchdog

loop forever

q1: await ninth frame

q2: if ran is false

q3: notify response-time over θ_w

q4: ran \leftarrow false

q5: rejoin readyQueue

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

queue of dataType buf; %empty queue

sample

dataType d
loop forever

p1: d := sample
p2: if buf is full do nothing
p3: else append(d,buf)

compute

dataType d
loop forever

q1: await buf not empty
q2: d := take(buf)
q3: compute(d)

Algorithm 13.7: Real-time buffering - overwrite old data

queue of dataType buf; %empty queue

sample

dataType d

loop forever

p1: d := sample

p2: append(d, buf)

p3:

compute

dataType d

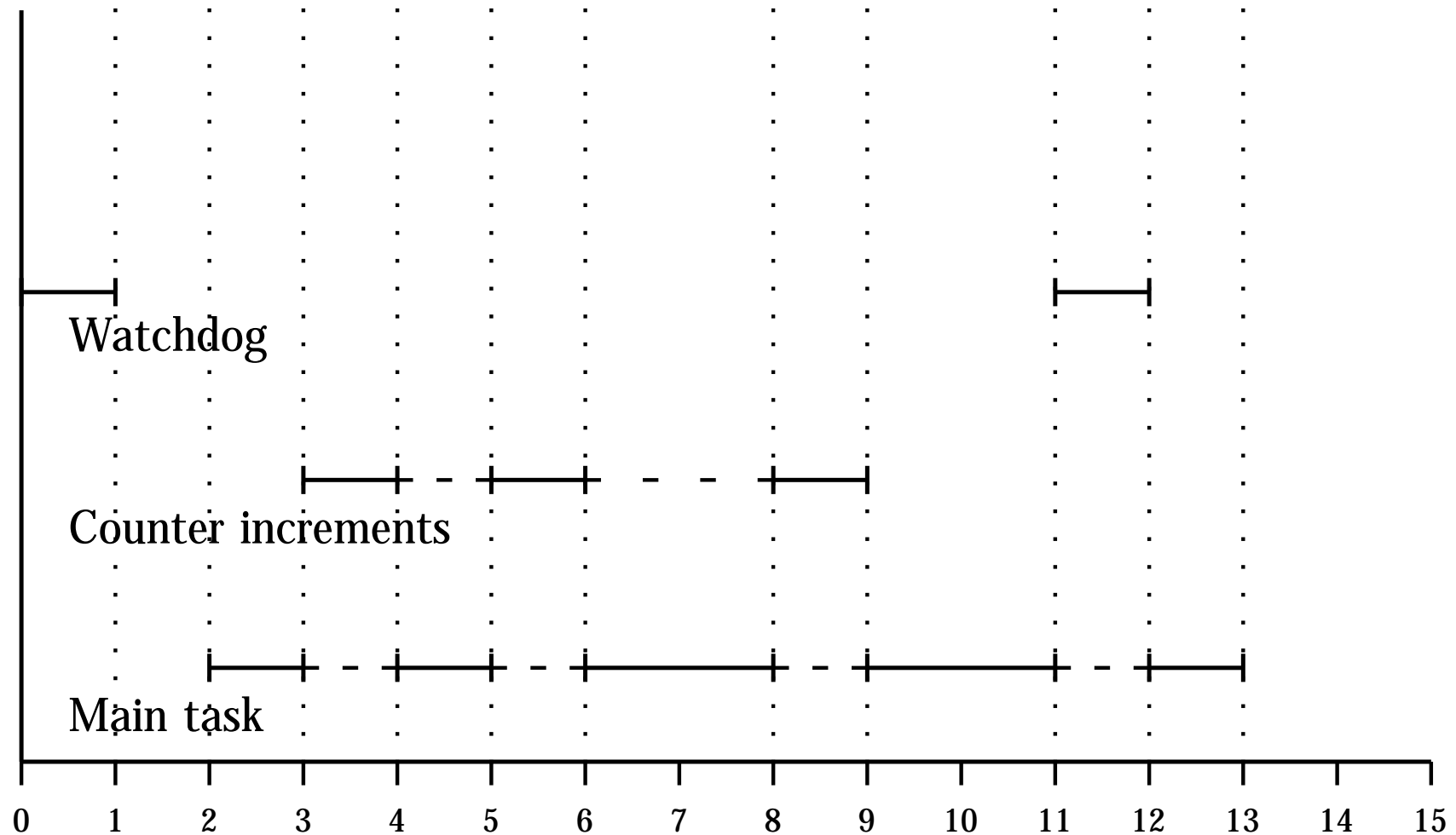
loop forever

q1: await buf not empty

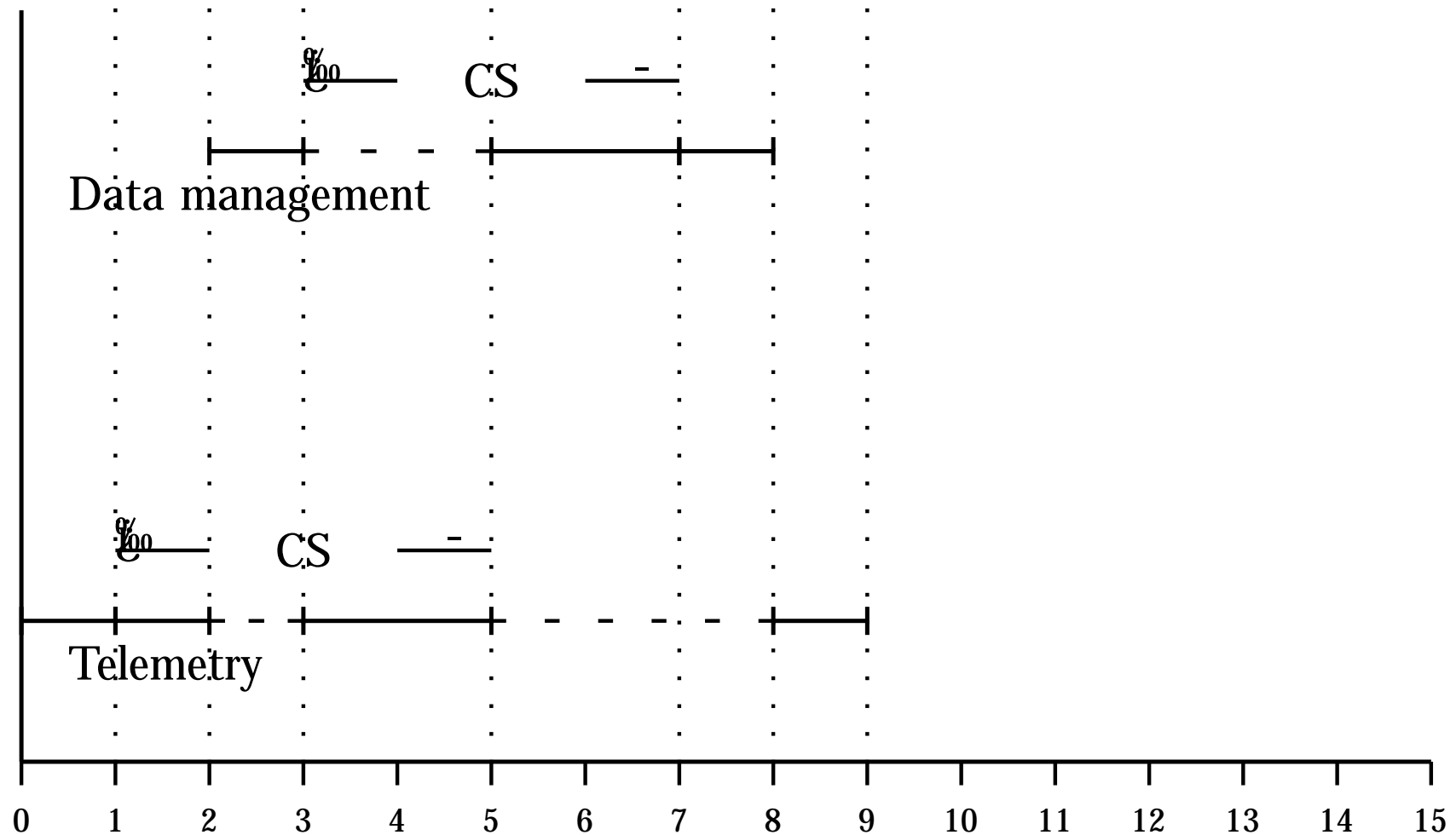
q2: d := take(buf)

q3: compute(d)

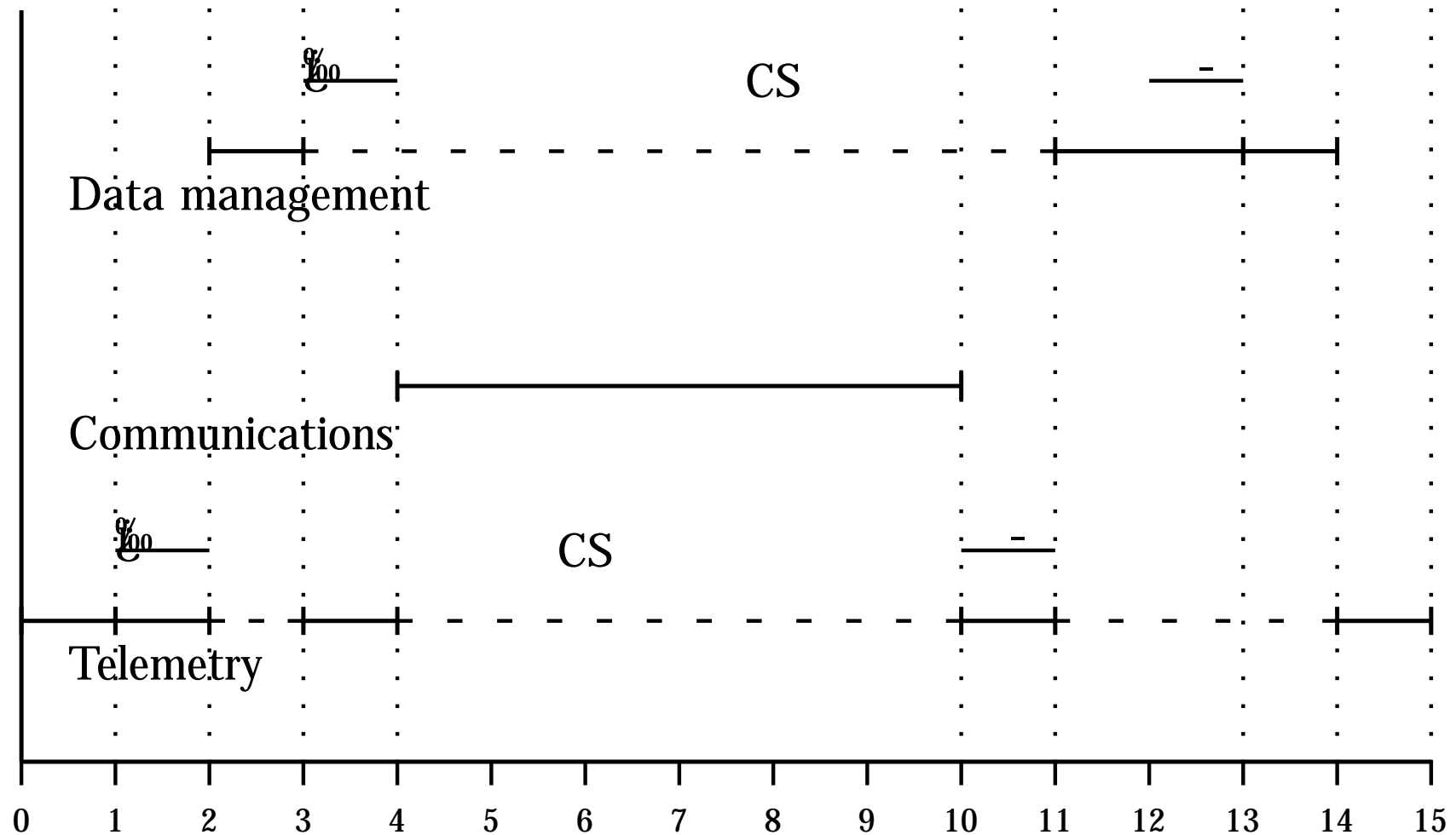
Interrupt Overrun on Apollo 11



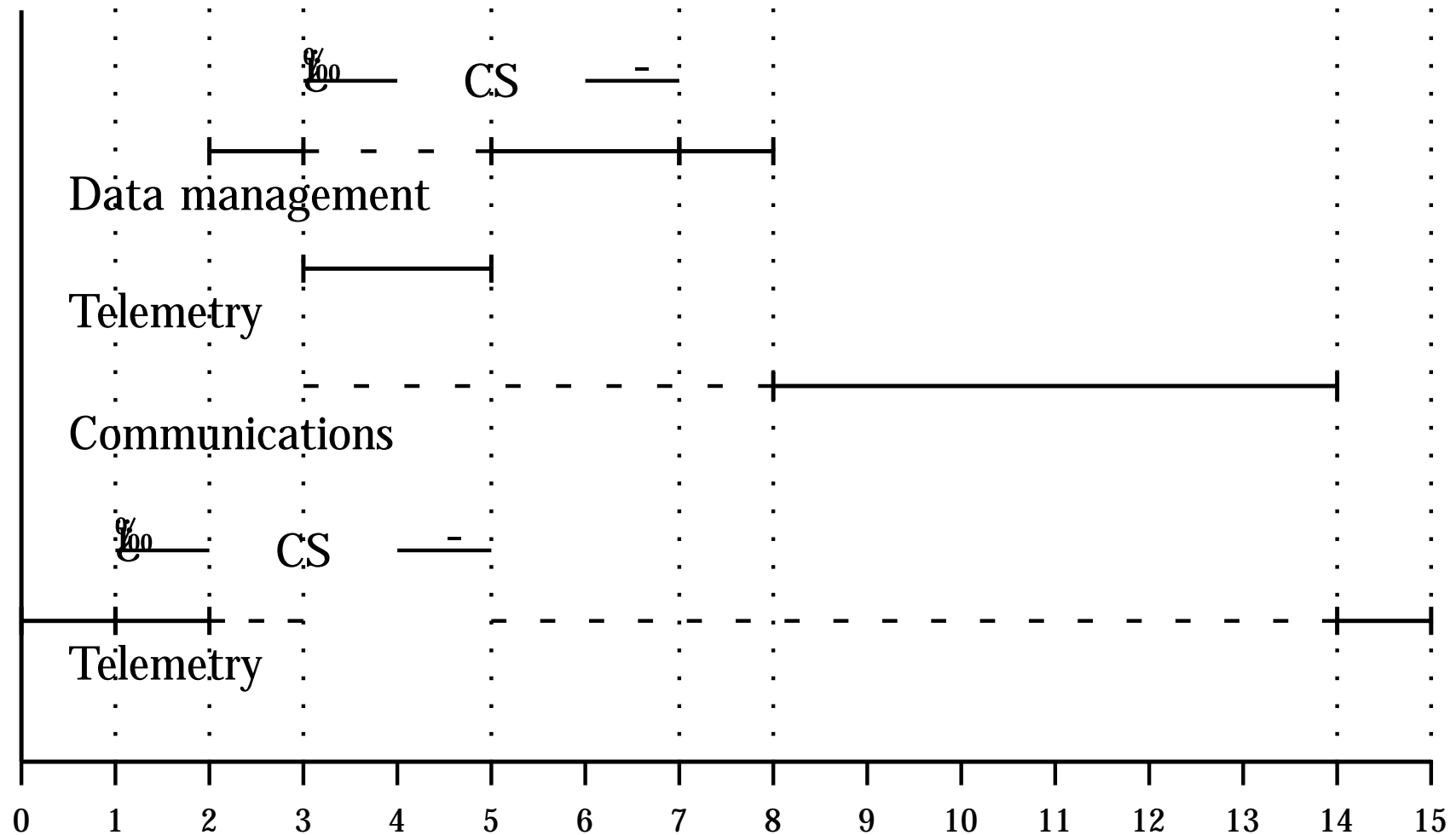
Priority Inversion (1)



Priority Inversion (2)



Priority Inheritance



Priority Inversion in Promela (1)

```
1  mtype = { idle, blocked, nonCS, CS, long };
2  mtype data = idle, comm = idle, telem = idle;
3  #define ready(p) (p != idle && p != blocked)
4
5  active proctype Data() {
6      do
7          :: data = nonCS;
8              enterCS(data);
9              exitCS(data);
10             data = idle;
11      od
12  }
13
14
15
```

Priority Inversion in Promela (1)

```
16  active proctype Comm() provided (!ready(data)) {
17      do
18          ::  comm = long;
19              comm = idle;
20      od
21  }
22  active proctype Telem()
23      provided (!ready(data) && !ready(comm)) {
24      do
25          ::  telem = nonCS;
26              enterCS(telem);
27              exitCS(telem);
28              telem = idle;
29      od
30  }
```

Priority Inversion in Promela (2)

```
1  bit sem = 1;
2
3  inline enterCS(state) {
4      atomic {
5          if
6              :: sem == 0 && !state == blocked;
7                  state = blocked;
8                  sem != 0;
9              :: else
10                 ;
11                 sem = 0;
12                 state = CS;
13         }
14     }
15
```

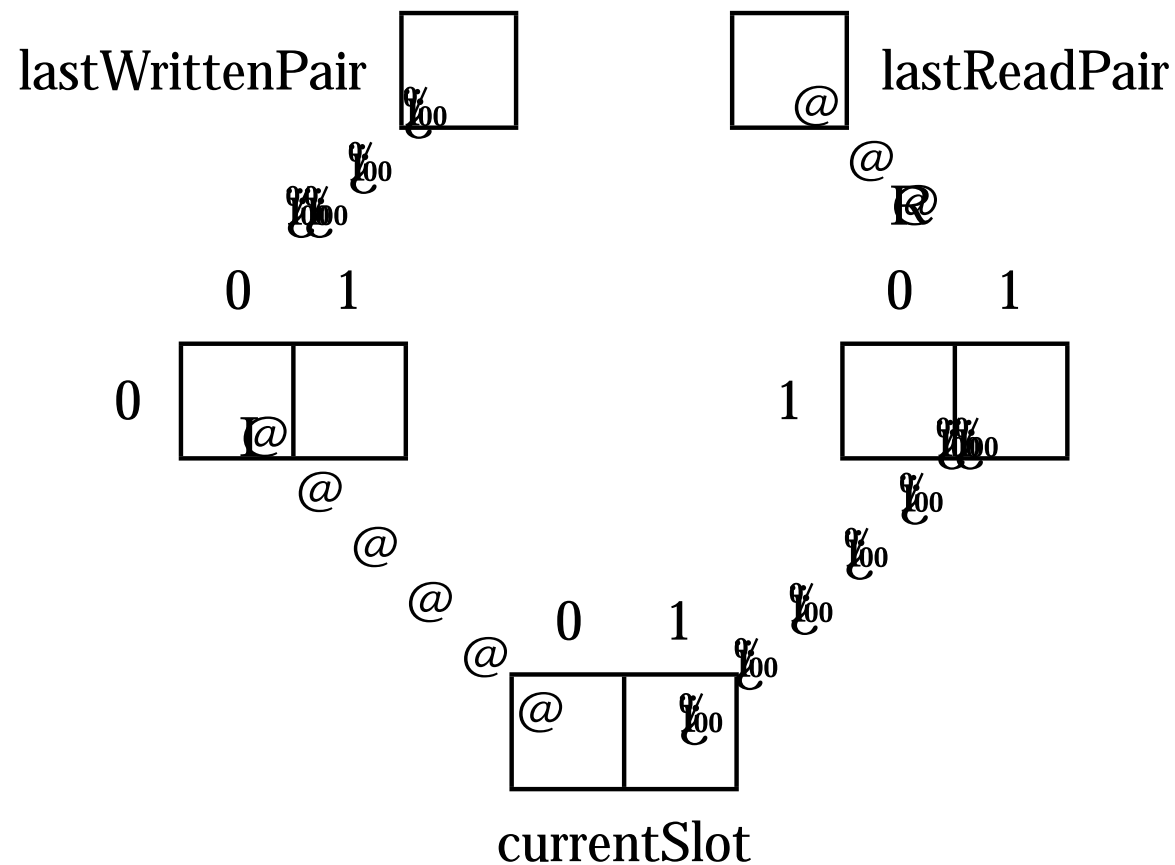
Priority Inversion in Promela (2)

```
16  inline exitCS(state) {  
17      atomic {  
18          sem = 1;  
19          state = idle  
20      }  
21 }
```


Priority Inheritance in Promela

```
1  #define inherit (p) (p == CS)
2  active proctype Data() {
3      do
4          :: data = nonCS;
5              assert( ! (telem == CS && comm == long) );
6              enterCS(data); exitCS(data);
7              data = idle;
8      od
9  }
10 active proctype Comm()
11     provided (!ready(data) && !inherit (telem))
12     { ... }
13 active proctype Telem()
14     provided (!ready(data) && !ready(comm) jj inherit (telem))
15     { ... }
```

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 $\dot{=}$ %lastWrittenPair

p8: lastReadPair $\dot{=}$ %readPair

p9: readSlot $\dot{=}$ %currentSlot[readPair]

p10: item $\dot{=}$ %data[readPair, readSlot]

p11: consume(item)

Algorithm 13.9: Event signaling

binary semaphore $s \in \{0, 1\}$

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

```
1  package Ada.Synchronous_Task_Control is  
2      type Suspension_Object is limited private;  
3      procedure Set_True(S : in out Suspension_Object);  
4      procedure Set_False(S : in out Suspension_Object);  
5      function Current_State(S : Suspension_Object)  
6          return Boolean;  
7      procedure Suspend_Until_True(  
8          S : in out Suspension_Object);  
9  private  
10     %not specified by the language  
11 end Ada.Synchronous_Task_Control;
```

Algorithm 13.10: Suspension object - event signaling

Suspension_Object SO \varnothing (false by default)

p

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

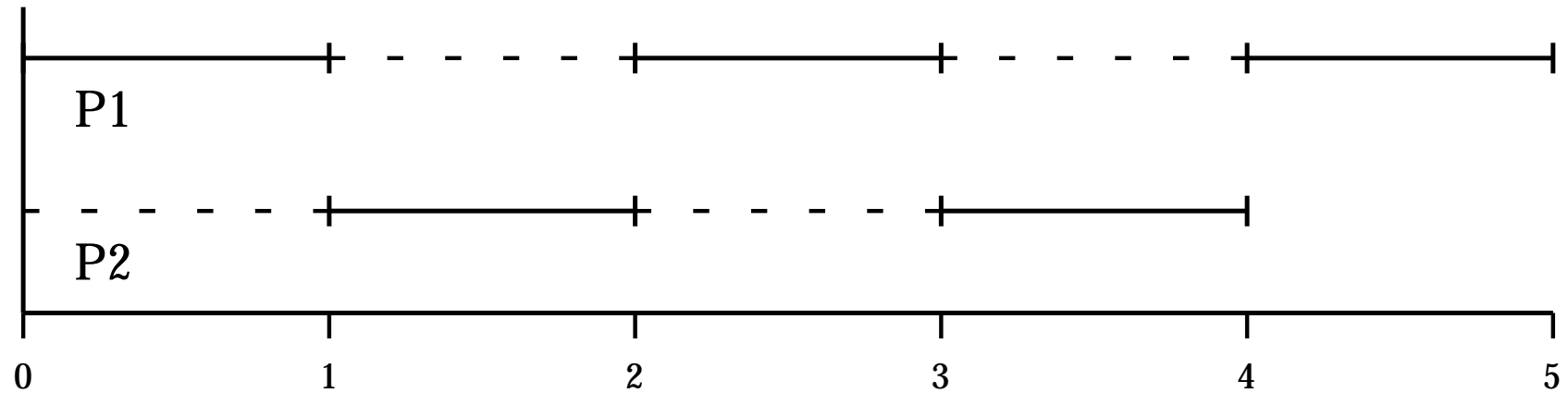
q

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

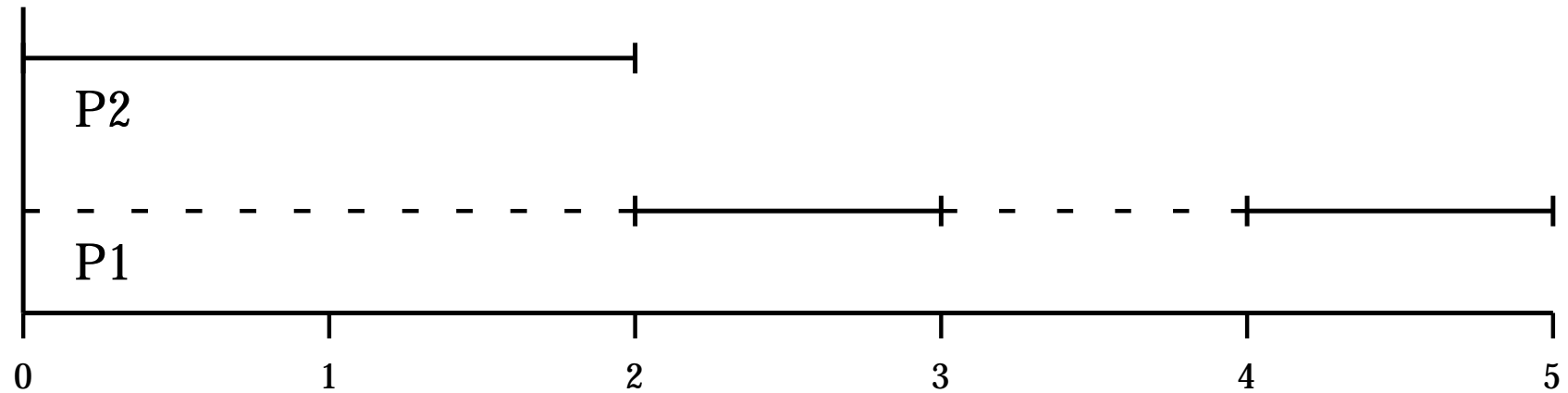
Transition in UPPAAL

$$\frac{\text{clk} \geq 12, \text{ch}?, n := n + 1}{\quad}$$

Feasible Priority Assignment



Infeasible Priority Assignment



Algorithm 13.11: Periodic task

constant integer period $\tau \geq 0$

integer next \leftarrow currentTime

loop forever

p1: delay next \leftarrow currentTime

p2: compute

p3: next \leftarrow 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
$A_1 \oplus A_2$	T	F	F
$A_1 \oplus A_2$	otherwise		T
$A_1 \equiv A_2$	$v(A_1) = v(A_2)$		T
$A_1 \equiv A_2$	$v(A_1) \neq v(A_2)$		F

Wason Selection Task



Algorithm B.1: Verification example

integer x1, integer x2

integer y1 ≥ 0 , integer y2 ≥ 0 , integer y3

p1: read(x1,x2)

p2: y3 $\geq x1$

p3: while y3 $\neq 0$

p4: if y2+1 = x2

p5: y1 $\geq y1 + 1$

p6: y2 ≥ 0

p7: else

p8: y2 $\geq y2 + 1$

p9: y3 $\geq y3 \geq 0$

p10: write(y1,y2)

Spark Program for Integer Division

```
1  ÷%#%main_program;
2  procedure Divide(X1,X2: in Integer; Q,R : out Integer)
3  ÷%#%derives Q, R from X1,X2;
4  ÷%#%pre (X1 >= 0) and (X2 > 0);
5  ÷%#%post (X1 = Q ÷%#% X2 + R) and (X2 > R) and (R >= 0);
6  is
7      N: Integer ;
8
9
10
11
12
13
14
15
```

Spark Program for Integer Division

```
16  begin
17      Q := 0; R := 0; N := X1;
18      while N /= 0
19          invariant assert (X1 = Q * X2 + R + N) and (X2 > R) and (R >= 0);
20      loop
21          if R + 1 = X2 then
22              Q := Q + 1; R := 0;
23          else
24              R := R + 1;
25          end if;
26          N := N - 1;
27      end loop;
28  end Divide;
```


Integer Division

```
1  procedure Divide(X1,X2: in Integer ; Q,R : out Integer) is
2      N: Integer ;
3  begin
4      pre pre (X1 >= 0) and (X2 > 0);
5      Q := 0; R := 0; N := X1;
6      while N /= 0
7      pre assert (X1 = Q * X2 + R + N) and (X2 > R) and (R >= 0);
8      loop
9          if R + 1 = X2 then Q := Q + 1; R := 0;
10         else R := R + 1;
11         end if;
12         N := N - X2;
13     end loop;
14     post post (X1 = Q * X2 + R) and (X2 > R) and (R >= 0);
15 end Divide;
```

Verification Conditions for Integer Division

Precondition to assertion:

$$(X1 \geq 0) \wedge (X2 > 0) !$$

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

Assertion to postcondition:

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

$$(X1 = Q \cdot X2 + R) \wedge (X2 > R) \wedge (R \geq 0) :$$

Assertion to assertion by then branch:

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

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

Assertion to assertion by else branch:

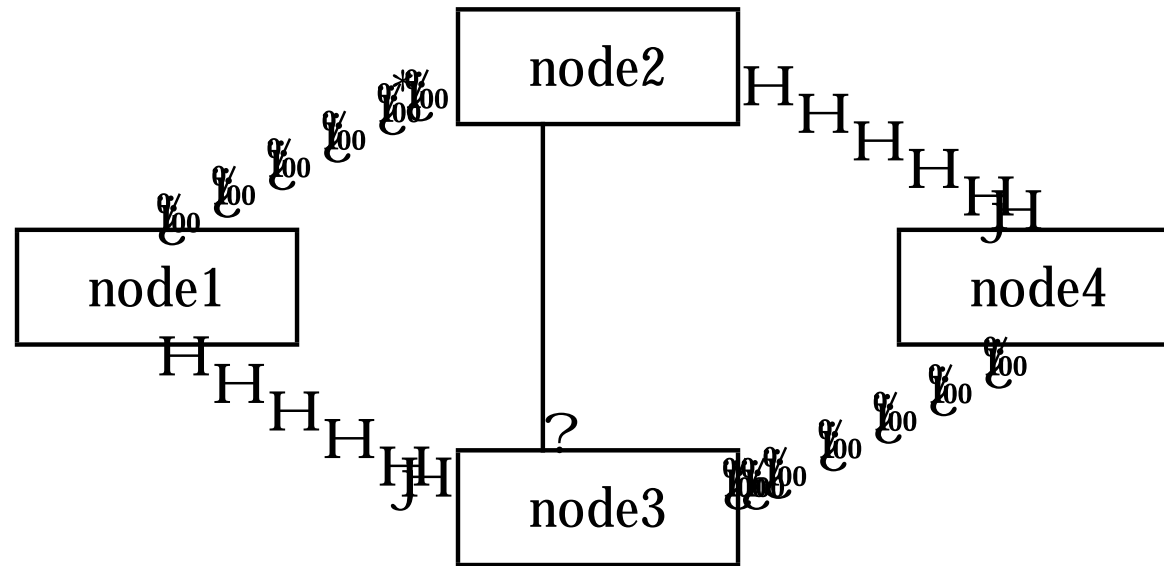
$$(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0) \wedge (R + 1 \leq X2) !$$

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

The Sleeping Barber

n	producer	consumer	Buff	notEmpty
1	append(d, Buff)	wait(notEmpty)	[]	0
2	signal(notEmpty)	wait(notEmpty)	[1]	0
3	append(d, Buff)	wait(notEmpty)	[1]	1
4	append(d, Buff)	d % take(Buff)	[1]	0
5	append(d, Buff)	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  $\in \{1, \dots, n\}$ 
integer list freeWomen  $\in \{1, \dots, n\}$ 
integer pair-list matched  $\in \emptyset$  ;
integer array[1..n, 1..n] menPrefs  $\in \emptyset \dots$ 
integer array[1..n, 1..n] womenPrefs  $\in \emptyset \dots$ 
integer array[1..n] next  $\in \{1\}$ 

```

```

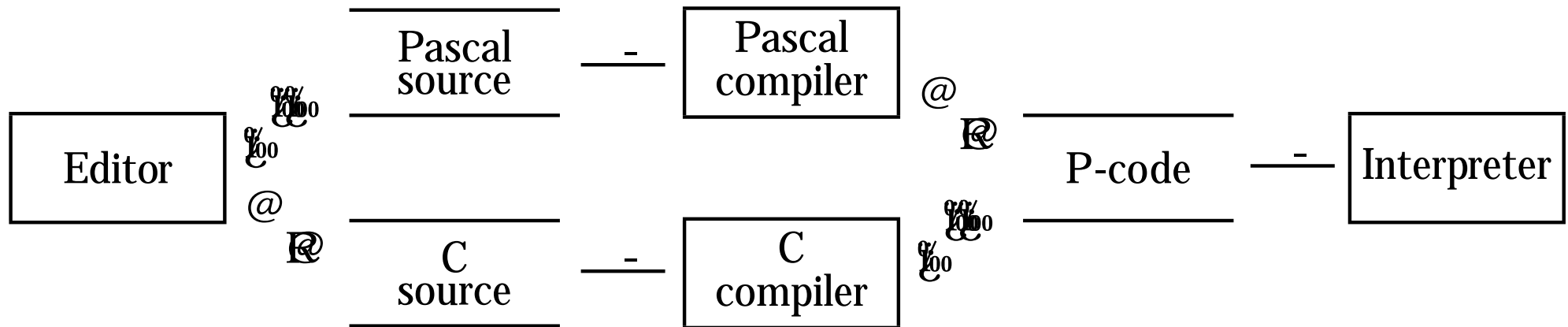
p1: while freeMen  $\neq \emptyset$  ; , choose some m from freeMen
p2:   w  $\in$  menPrefs[m, next[m]]
p3:   next[m]  $\in$  next[m] + 1
p4:   if w in freeWomen
p5:     add (m,w) to matched, and remove w from freeWomen
p6:   else if w prefers m to m' // where (m',w) in matched
p7:     replace (m',w)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

```

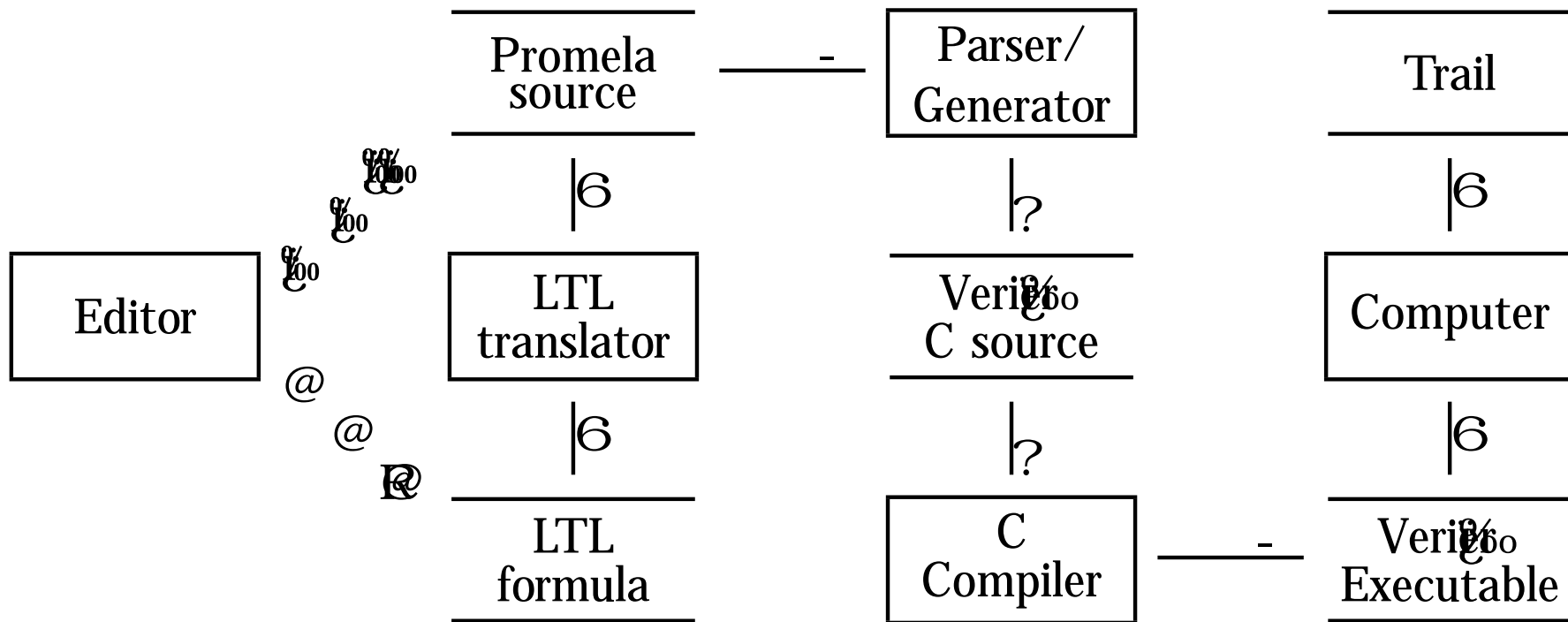
The n-Queens Problem

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



Cycles in a State Diagram

