SEMAPHORES

Dr. Emmanuel S. Pilli

Semaphores

- The previous algorithms for Critical Section problems can be run on a bare machine and use only the machine language instructions that the computer provides
- They are too low level to be used efficiently and reliably
- Semaphore provides a concurrent programming construct which is higher level than machine instructions
- They are simple, successful and widely used

Semaphores

- Semaphores are usually implemented by an underlying Operating System
- However we attempt to define the behavior of a semaphore and assume that this behavior can be implemented
  - Define Semaphore Construct and solve CSP
  - Invariants on Semaphores to prove correctness
  - Various types of Semaphores
  - New problems where requirement is to achieve cooperation between process and not mutex
  - Producer-Consumer, Reader-Writer, Dining Philosopher

Process States

- Multiprocessor system may have more processors than processes in a program
  - Every process is always running on some processer
- In a multitasking system (and like multiprocessor system, where there are more processes than processors)
  - Several processes will share the computing resources of a single CPU
- **Running** – process in execution
- **Ready** – process that wants to run
  - One running and many ready processes at any given instant
- **Idle** – process which runs when no processes are running

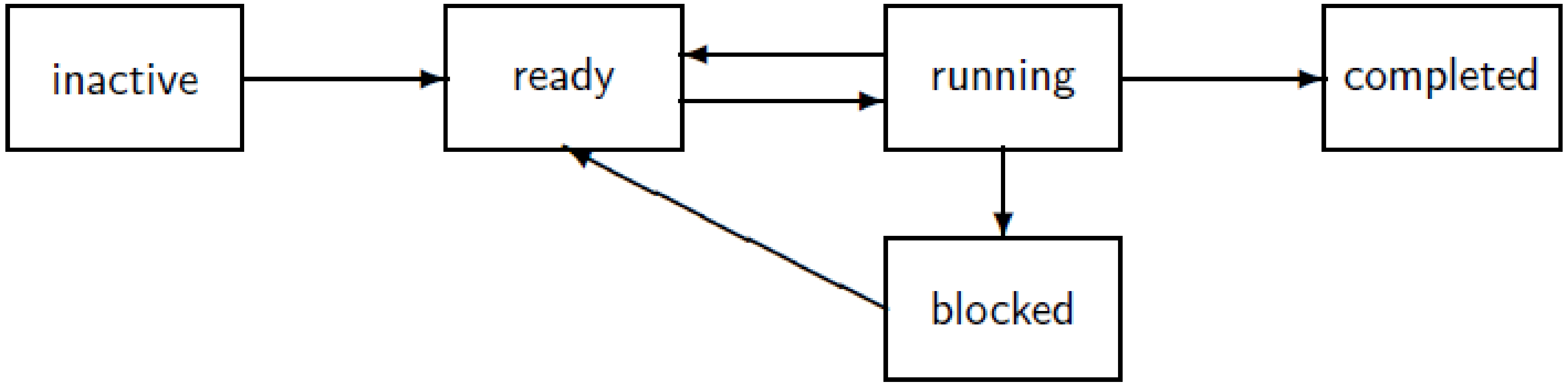
Process States

- Scheduler is a system program responsible for:
  - deciding which of the ready processes should run
  - perform the context switch
  - replace the running process with a ready process
  - changing the state of the running process to ready
- Arbitrary interleaving simply means that the scheduler may perform context switch at any time
- Refer text books on Operating System on design of schedulers

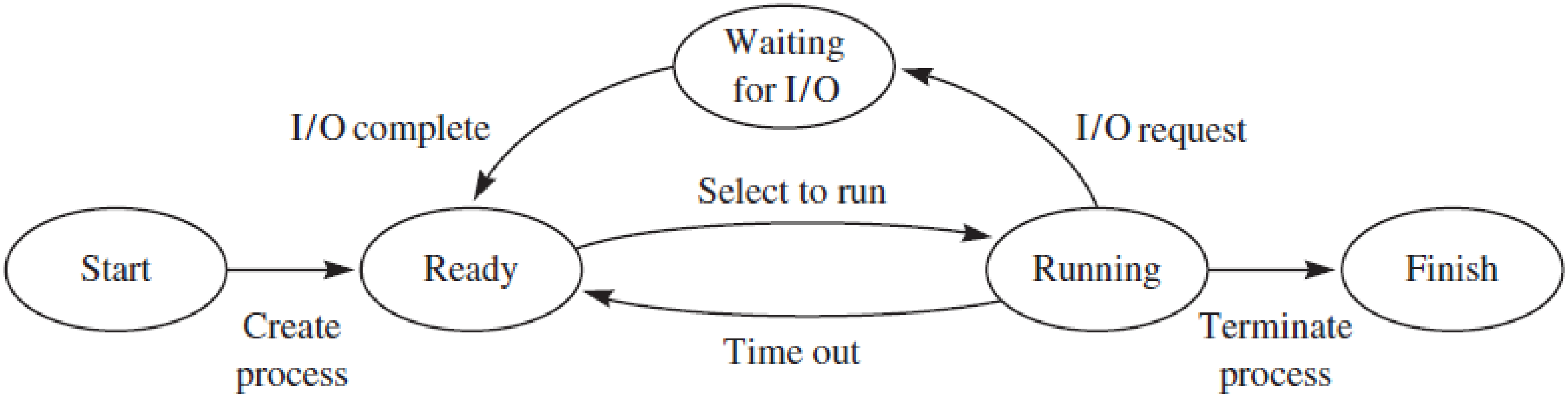
Process States

- **Blocked** – when a process is blocked, its not **ready** and is not a candidate for becoming a **running** process
- A process can be **Unblocked**, **Awakened** or **Released** – only if an external action changes the process state from **blocked** to **ready**
- The unblocked process becomes a candidate for execution along with all current **ready** processes
- **Inactive –** Initial state of a process. Process is activated at some point and becomes **ready**
- **Complete –** When a process executes its final statement

Process States



Process States



Definition

- Semaphore S is a compound data type with two fields
  - Non-negative integer S.V
  - Set of processes in a queue S.L
- A Semaphore whose integer component can take arbitrary non negative values is called a general semaphore
- There are two atomic operations defined on a semaphore – wait and signal
- Semaphore S must be initialized with a value of $k \geq 0$ for S.V and with empty set ϕ for S.L

Initialization and Wait

Initialization

```
semaphore $S \leftarrow (k, \phi)$
```

wait (S)

```
    if S.V > 0
        S.V ← S.V − 1
    else
        S.L ← S.L ∪ p
        p.state ← blocked
```

Wait

- If the value of integer component is non zero, decrement its value and process **p** can continue its execution
- If it is zero, process **p** is added to set component and the state of the **p** becomes blocked
- Process **p** is said to have been blocked on semaphore

Signal

```
signal (S)
    if $S.L = \phi$
        $S.V \leftarrow S.V + 1$
    else
        Let q be some process in S.L   S.
L $\leftarrow$ S.L−{q}
  q.state $\leftarrow$ ready
```

Signal

- If S.L is empty, increment the value of the integer component
- If S.L is non empty – unblock **q** an arbitrary element of the set of processes blocked on S.L.
- The status of **p** does not change

- A semaphore whose integer component can take arbitrary non negative values is called a *general semaphore*

Binary Semaphore

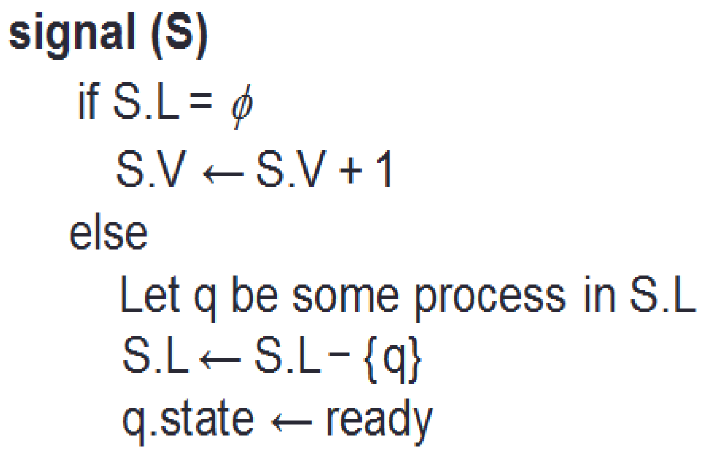
- A semaphore whose integer component takes only the values 0 and 1 is called a *binary semaphore*
- The value S.V is only allowed to be 0 or 1
- It is also called “mutex” for mutual exclusion
- Binary semaphore is initialized with (0, ϕ) *or* (1, ϕ)
- wait (S) instruction is unchanged but signal (S) is changed

Wait

```
**wait (S)**
    if $S.V > 0$
        $S.V \leftarrow S.V - 1$
    else
        $S.L \leftarrow S.L \cup p$
        $p.state \leftarrow blocked$
```

Signal

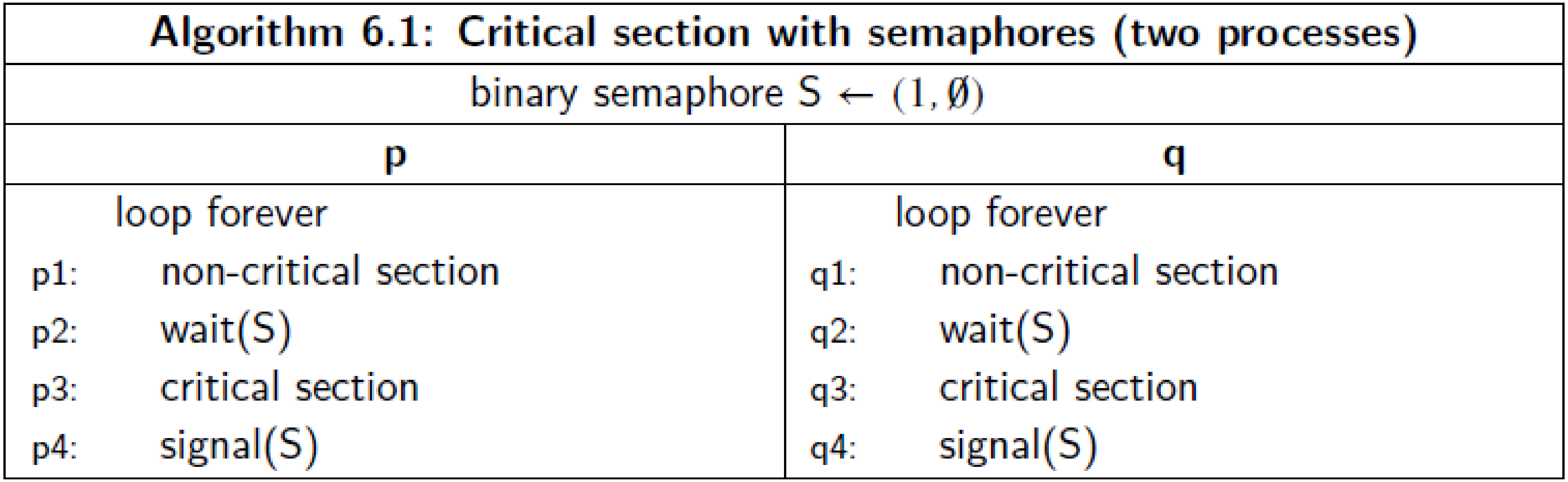
```
signal (S)
    if S.V = 1
        // undefined
    else if S.L = $\phi$
      S.V $\leftarrow$ 1
    else // (as above)
        let q be some process in S.L
        S.L $\leftarrow$ S.L−{q}
        q.state $\leftarrow$ ready
```



```
signal (S)
    if S.L = $\phi$
      S.V $\leftarrow$ S.V + 1
    else
      Let q be some process in S.L
      S.L $\leftarrow$ S.L − {q}
      q.state $\leftarrow$ ready
```

Critical Section with two Processes

- The critical section problem is trivial when you have semaphores.



Algorithm 6.1: Critical section with semaphores (two processes)	
binary semaphore S ← $(1, \emptyset)$	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

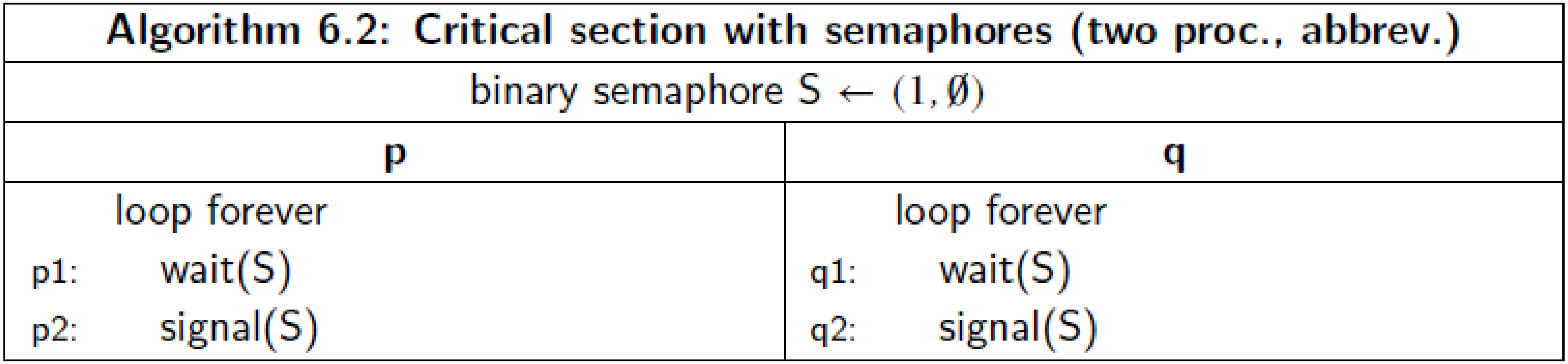
Critical Section with two Processes

- A process **p** that wishes to enter its critical section executes a preprotocol that consists only of the wait(S) statement
- If S.V=1 then S.V is decremented and **p** enters critical section
- When **p** exits critical section and executes the post protocol consists of only of the signal(S) statement, the value of S will once more be set to 1

Critical Section with two Processes

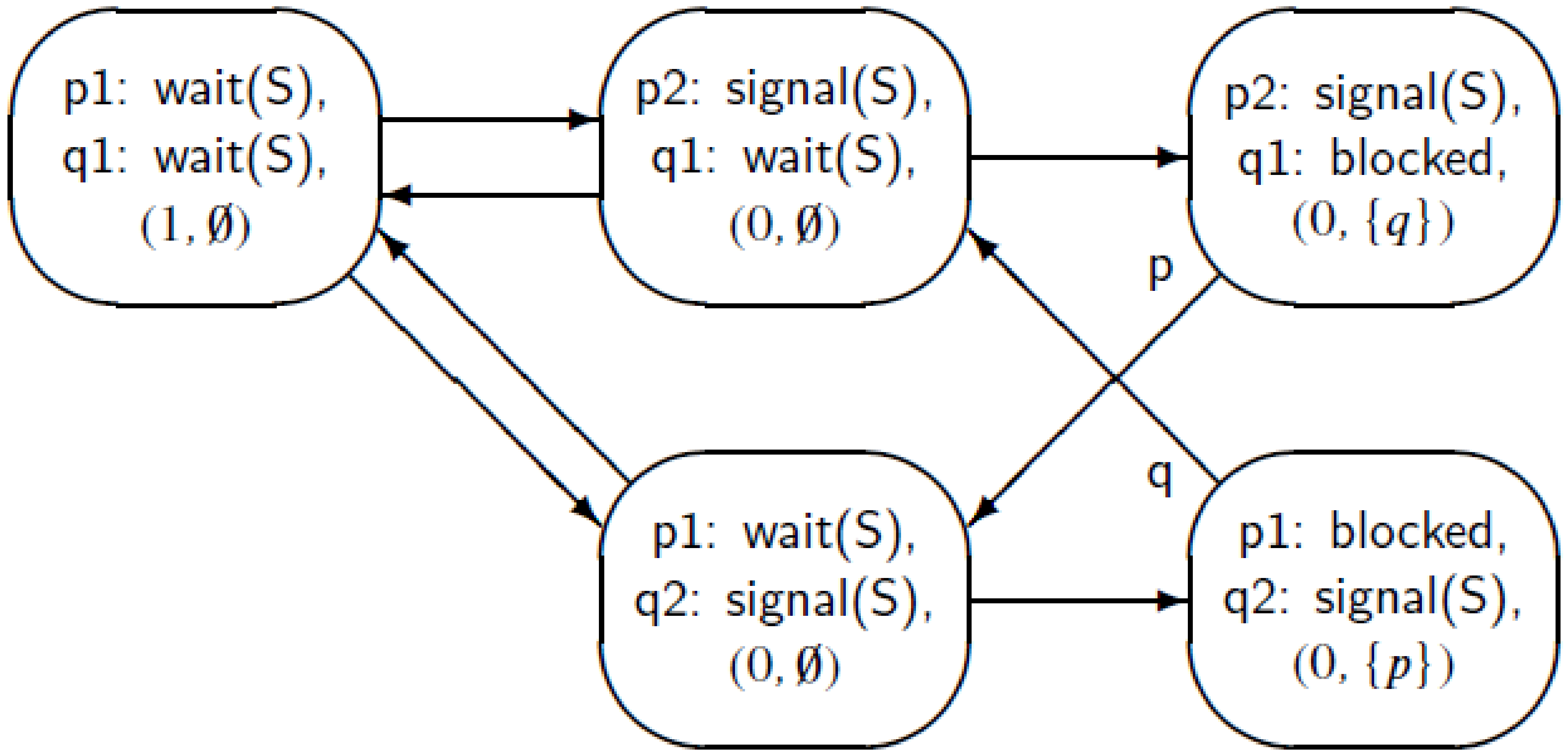
- If **q** attempts to enter the critical section by executing wait(S) before **p** has left, S.V=0 and q will become blocked on S
- The value of semaphore will be (0, {q})
- When **p** leaves the critical section and executes signal(S), the arbitrary process in the S.L = {q} will be **q**, so that the process will be unblocked and proceed into its critical section
- The solution is similar to the Second Attempt except that
  - the interleaving between test and assignment is prevented
  - definition of semaphore operations as atomic statements

Abbreviated Form



Algorithm 6.2: Critical section with semaphores (two proc., abbrev.)	
binary semaphore $S \leftarrow (1, \emptyset)$	
p	**q**
loop forever	loop forever
p1: wait(S)	q1: wait(S)
p2: signal(S)	q2: signal(S)

State Diagram



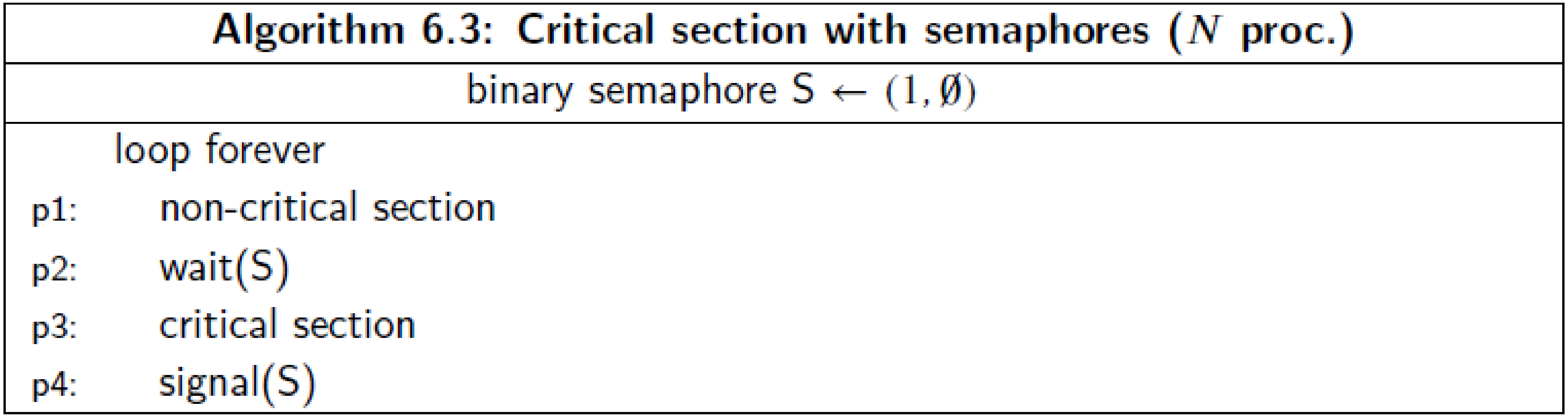
p1: wait(S), q1: wait(S), $(1, \emptyset)$
p2: signal(S), q1: wait(S), $(0, \emptyset)$
p2: signal(S), q1: blocked, $(0, \{q\})$
p1: wait(S), q2: signal(S), $(0, \emptyset)$
p1: blocked, q2: signal(S), $(0, \{p\})$
p
q

Correctness Specifications

- A violation of the mutual exclusion requirement would be a state of the form (p2: signal(S), q2: signal(S), …)
- No such state exists
- There is no deadlock as there are no states in which both processes are blocked
- The algorithm is free from starvation since if a process executes its wait statement, it enters either a state with the signal statement or it enters a state in which it is blocked
- The only way out of a blocked state is into a state in which the blocked process continues with the signal statement

Critical Section with N processes

- The same algorithm gives the solution for CS problem for N processes using semaphores:



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

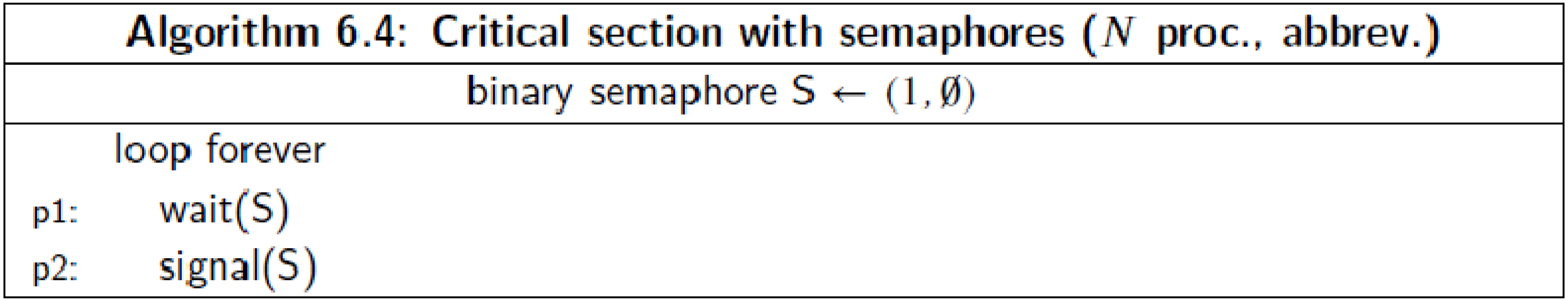
binary semaphore S $\leftarrow (1, \emptyset)$

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

Critical Section with N processes

- Mutual exclusion and Freedom from deadlock holds for N processes
- Freedom for starvation does not hold

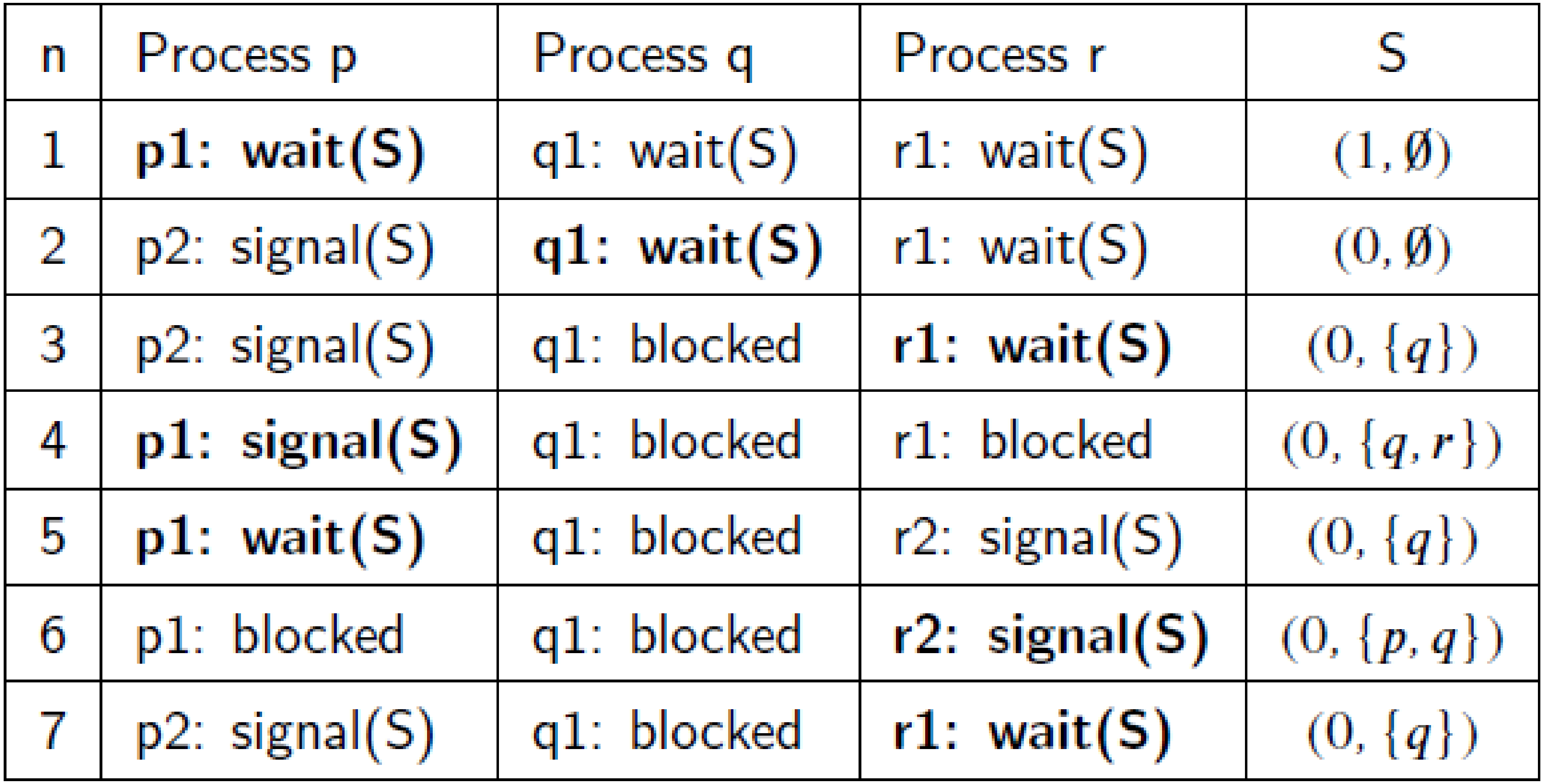
- Consider the abbreviated algorithm:



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

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

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

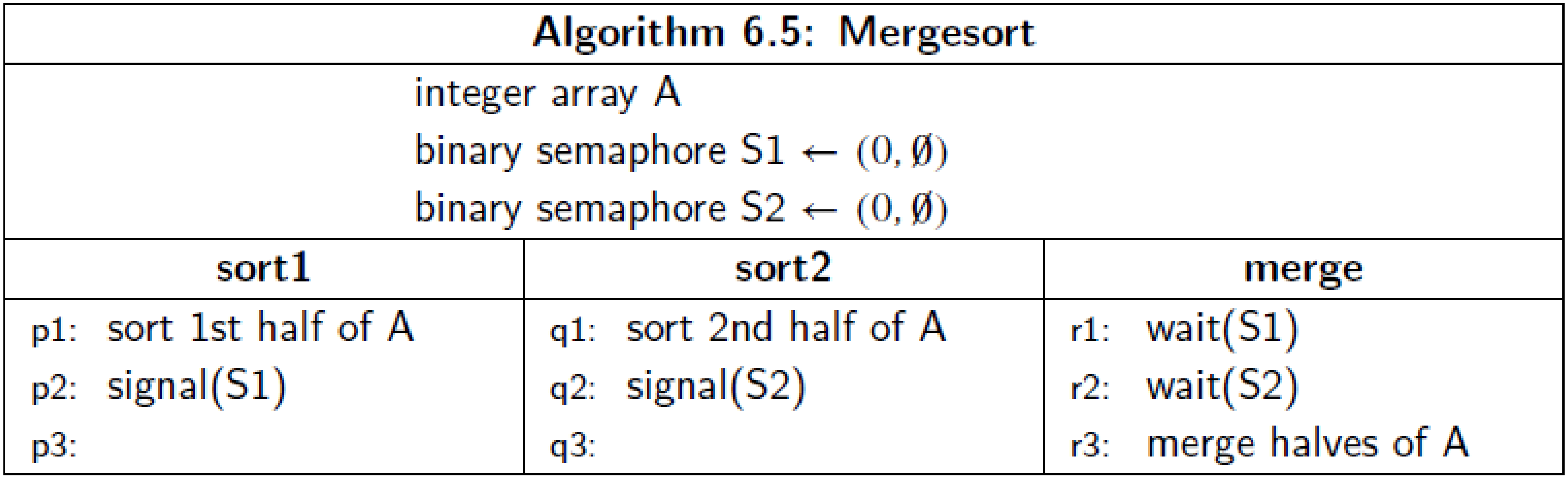
Scenario for Starvation



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

Order of Execution

- Synchronization problems are also common when processes coordinate the order of execution of operations of different processes.
- Uses a split binary semaphore (discussed later)



Algorithm 6.5: Mergesort

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

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

Types of Semaphores

- Several different types of semaphores
- The differences are due to the specification of liveness properties
- They do not affect the safety properties that follow from the semaphore invariants, so any definition we use does not affect the mutual exclusion

Types of Semaphores

Strong Semaphore:

- Weak semaphore has S.L, a set of processes blocked on semaphore S
- This is replaced by a queue to become a strong semaphore

```
wait (S)
    if S.V > 0
        S.V ← S.V – 1
    else
        S.L ← append (S.L,p)
        p.state ← blocked
```

```
signal (S)
    if S.L = ϕ
        S.V ← S.V + 1
    else
        q ← head (S.L)
        S.L ← tail (S.L)
        q.state ← ready
```

Types of Semaphores

Busy-Wait Semaphore:

- It does not have a component S.L and S is identified only by S.V
- Busy-Wait Semaphores are appropriate in a multi processor system where the waiting process has its own processor and is not wasting CPU time that could be used for other computation

```
wait (S)
          await $S > 0$
                    $S \leftarrow S - 1$

signal (S)
          $S \leftarrow S + 1$
```

Producer Consumer Problem

- **Producers:** Producer process executes a statement **produce** to create a data element and then sends this element to the consumer process
- **Consumers:** Upon receipt of the data element from the producer processes, a consumer process executes a statement **consume** with the data element as a parameter
- Communication can be synchronous or asynchronous

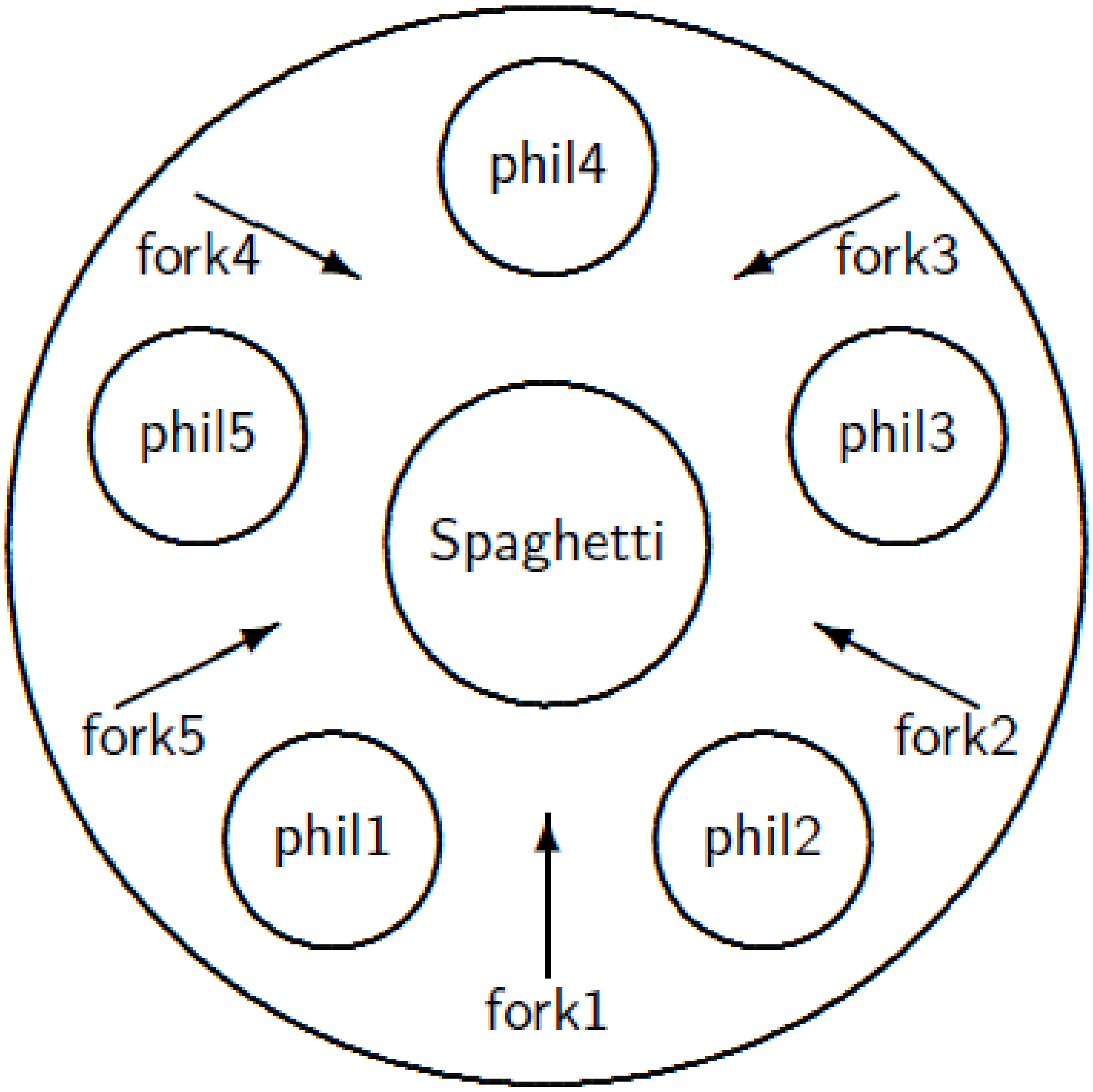
Producer Consumer Problem

- Asynchronous communication involves a channel which has some capacity for storing elements
- The store is a queue of elements called a *buffer*
- Producer executes an **append** operation to place a data element on the tail of the queue
- Consumer executes a **take** operation to remove a data element from the head of the queue
- The use of buffer is to allow processes of average similar speeds to proceed smoothly
- Buffer can also be used to resolve a clash in the structure of data

Dining Philosophers Problem

- Classical problem in field of concurrent programming
- Sufficiently simple to be tractable yet subtle enough to be challenging
- The problem is set in a secluded community of **five** philosophers who engage in only two activities – ***thinking*** and ***eating***
- Meals are taken communally at a table set with **five plates** and **five forks** with a bowl of spaghetti that is endlessly replenished

Dining Philosophers Problem



Dining Philosophers Problem

- Philosopher needs two forks in order to eat
- Each philosopher may pick up the forks on his left and right but only one at a time
- The problem is to design pre and post protocols to ensure that a philosopher only eats if she has two forks
- Solution should satisfy the correctness properties

Dining Philosophers Problem

Algorithm 6.9: Dining philosophers (outline)

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

Readers Writers Problem

- Similar to Mutex problem where several processes are competing for access to a critical section
- Readers are processes which require to exclude writers but not other readers
- Writers are processes which require to exclude both readers and other writers
- Several processes can read data concurrently, but writing or modifying data must be done under Mutex to ensure consistency of Data

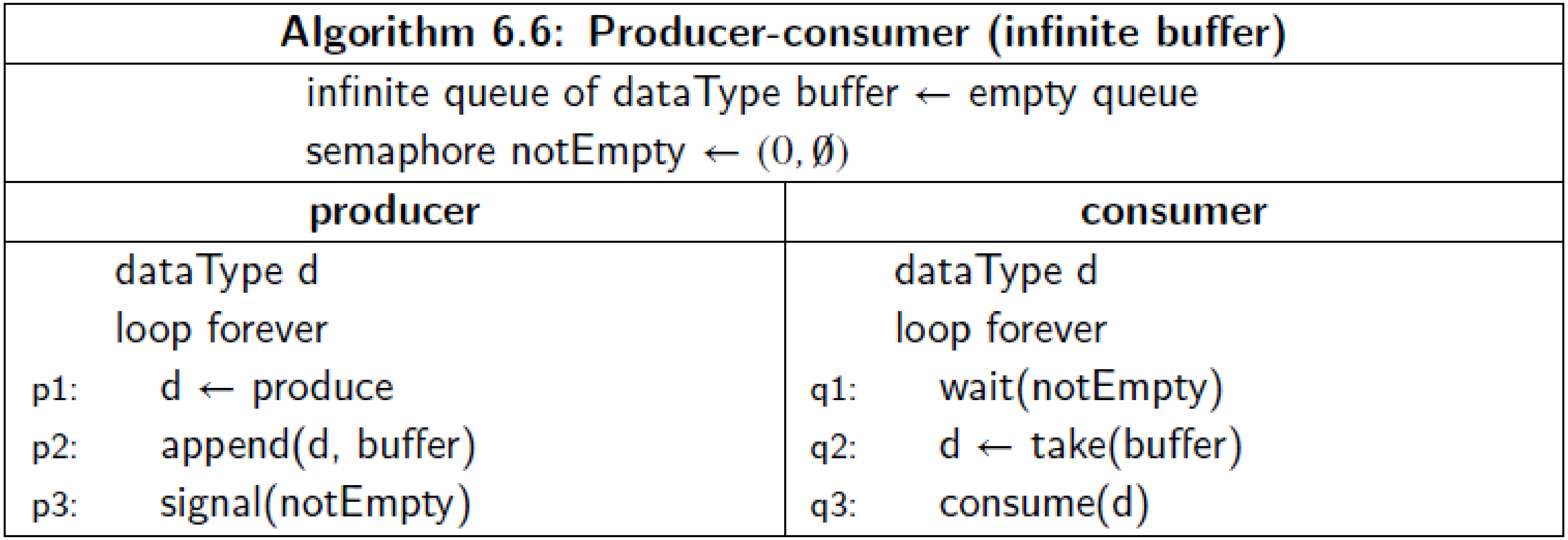
Producer Consumer Problem

- Two Synchronization issues:
- Consumer cannot take data from an empty buffer
- Producer cannot append a data element to a full buffer as the buffer size is finite

- We consider buffer can be infinite or bounded to devise a solution

Infinite Buffer

- If there is an infinite buffer, there is only one interaction that must be synchronized: consumer must not attempt a take operation from an empty buffer



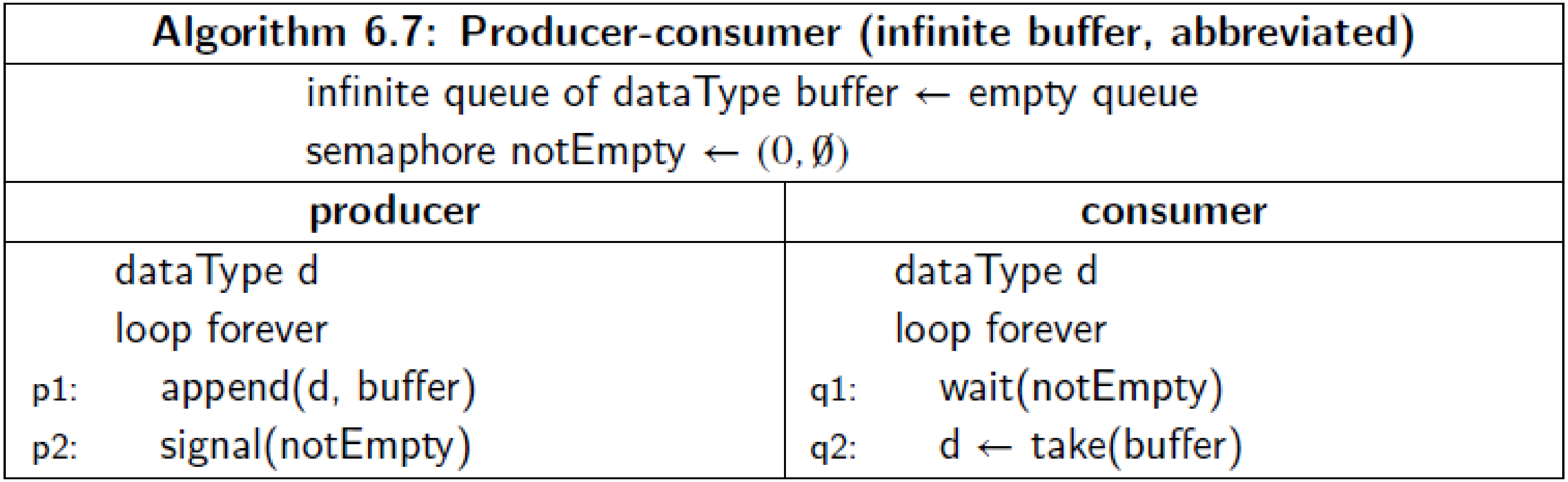
Algorithm 6.6: Producer-consumer (infinite buffer)

```
infinite queue of dataType buffer ← empty queue
semaphore notEmpty ← (0, ∅)
```

producer	consumer
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: append(d, buffer)	q2: d ← take(buffer)
p3: signal(notEmpty)	q3: consume(d)

Infinite Buffer - Abbreviated

- As buffer is infinite, it is impossible to construct a finite state diagram for the algorithm



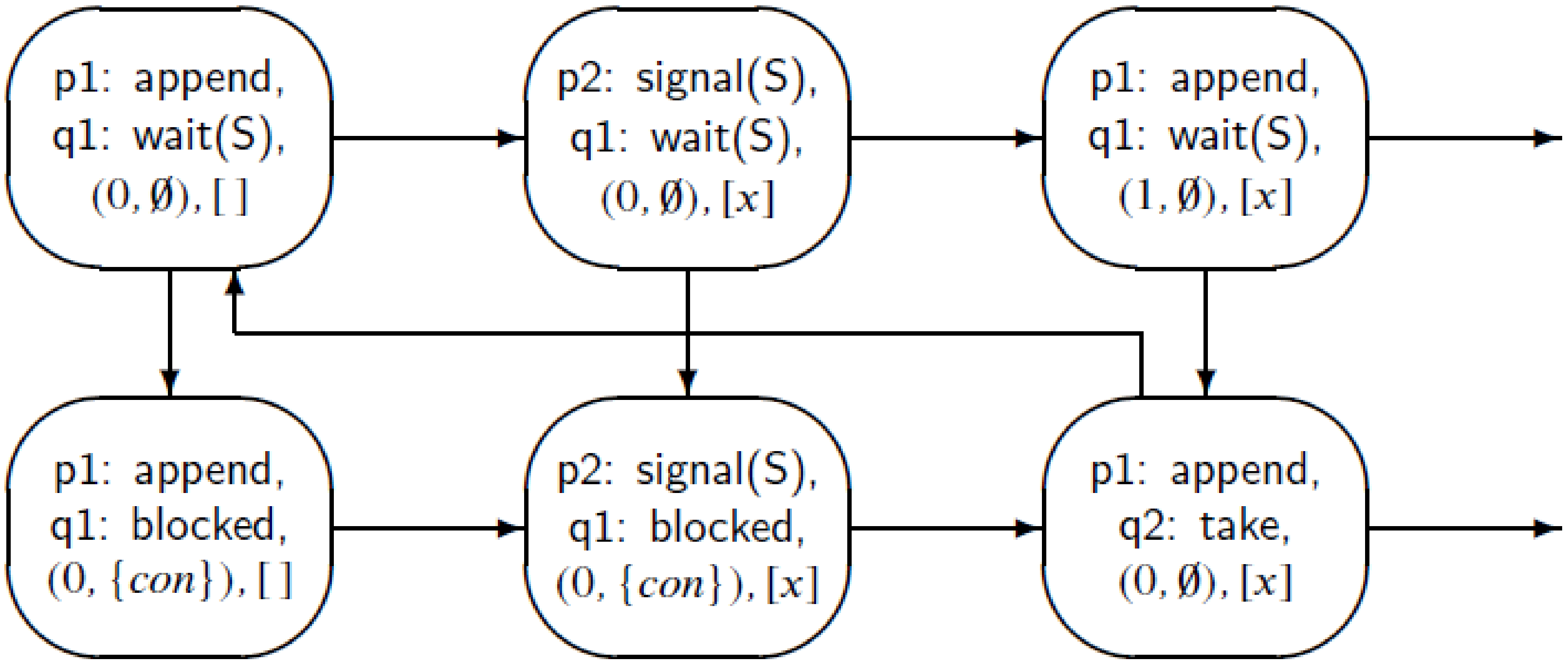
Algorithm 6.7: Producer-consumer (infinite buffer, abbreviated)

```
infinite queue of dataType buffer ← empty queue
semaphore notEmpty ← (0, ∅)
```

producer	consumer
dataType d	dataType d
loop forever	loop forever
p1: append(d, buffer)	q1: wait(notEmpty)
p2: signal(notEmpty)	q2: d ← take(buffer)

Infinite Buffer – State Diagram

- Value of buffer is written with square brackets and a buffer element is denoted by x, consumer process is denoted by con.
- Horizontal rows indicate execution of operations by the producer while vertical rows are for the consumer



p1: append, q1: wait(S), $(0, \emptyset)$, [ ]
p2: signal(S), q1: wait(S), $(0, \emptyset)$, $[x]$
p1: append, q1: wait(S), $(1, \emptyset)$, $[x]$
p1: append, q1: blocked, $(0, \{con\})$, [ ]
p2: signal(S), q1: blocked, $(0, \{con\})$, $[x]$
p1: append, q2: take, $(0, \emptyset)$, $[x]$

Infinite Buffer - Correctness

- Consumer does not remove an element from an empty buffer
- Algorithm is free from deadlock – because – as long as producer continues to produce data elements, it will execute signal(notEmpty) operations and unblock the consumer
- It is also free from starvation as there is only one possible blocked process

Bounded Buffer

- The algorithm for producer-consumer problem with an infinite buffer can be easily extended to one with a finite buffer
- Producer takes empty places from a buffer just as the consumer takes data elements from the buffer
- We use a similar synchronization mechanism with a semaphore notFull that is initialized to N, the number of initially empty spaces in the finite buffer.

Bounded Buffer

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

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

producer	**consumer**
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: wait(notFull)	q2: d ← take(buffer)
p3: append(d, buffer)	q3: signal(notFull)
p4: signal(notEmpty)	q4: consume(d)

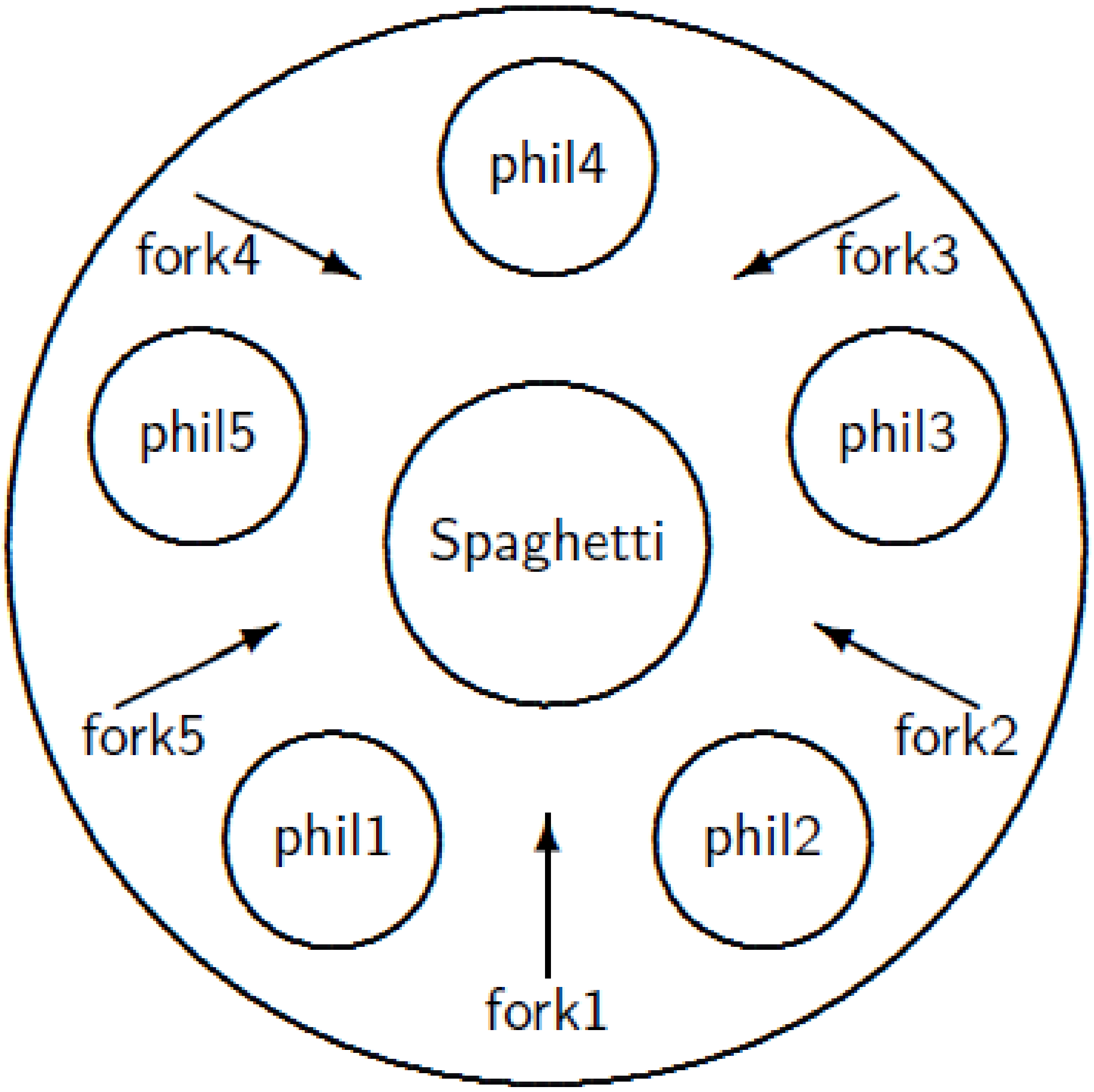
Split Semaphore

- The previous algorithm uses a technique called Split Semaphore
- This is not a new Semaphore type but simply a term used to describe a synchronization mechanism built from semaphores.
- A split semaphore is a group of two or more semaphores satisfying an invariant that the sum of their values is at most equal to a fixed number N.
- notEmpty + notFull = N
- In case if N=1, its called a split *binary* semaphore
- Split semaphores enable one process to wait for the completion of an event in another

Dining Philosophers Problem

- Classical problem in field of concurrent programming
- Sufficiently simple to be tractable yet subtle enough to be challenging
- The problem is set in a secluded community of **five** philosophers who engage in only two activities – ***thinking*** and ***eating***
- Meals are taken communally at a table set with **five plates** and **five forks** with a bowl of spaghetti that is endlessly replenished

Dining Philosophers Problem



Dining Philosophers Problem

Algorithm 6.9: Dining philosophers (outline)

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

Dining Philosophers Problem

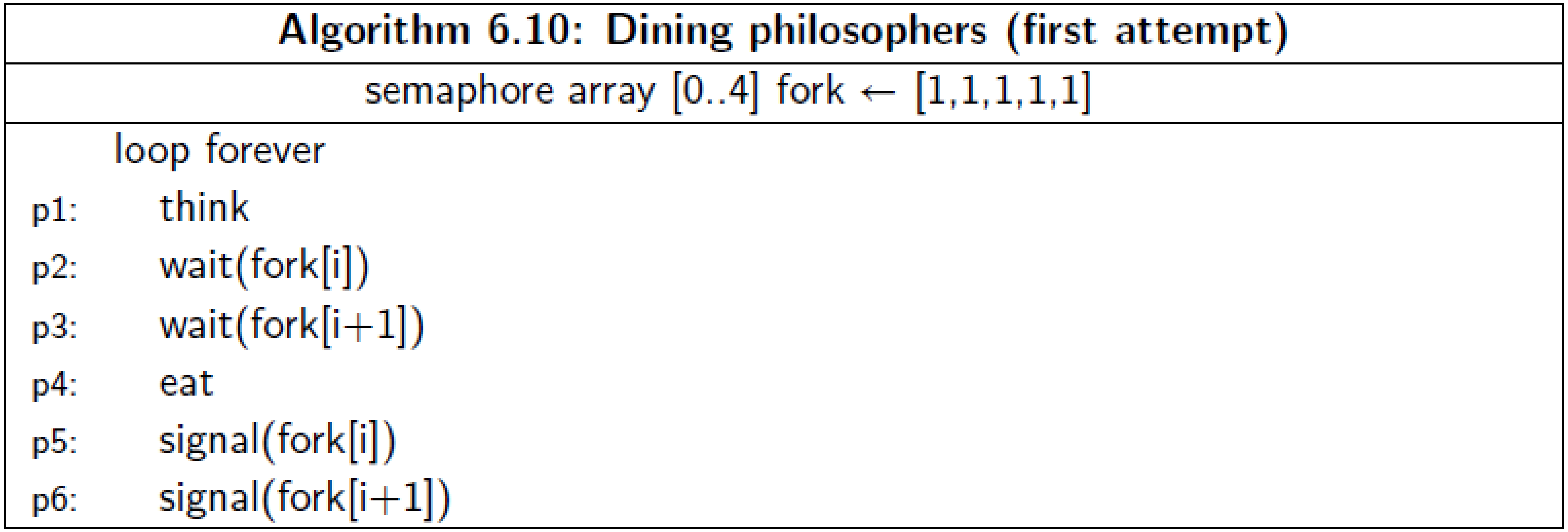
- Philosopher needs two forks in order to eat
- Each philosopher may pick up the forks on his left and right but only one at a time
- The problem is to design pre and post protocols to ensure that a philosopher only eats if she has two forks
- Solution should satisfy the correctness properties

Correctness Properties

- A philosopher can eat only if she has two forks
- Mutual Exclusion: No two philosophers may hold the same fork simultaneously
- Freedom from deadlock
- Freedom from starvation
- Efficient behavior in the absence of contention

Dining Philosophers – First Attempt

- We assume that each philosopher is initialized with its index i, and that addition is implicitly modulo 5.
- Each fork is modelled as a semaphore: wait corresponds to taking a fork and signal corresponds to putting down a fork
- Philosopher hold both forks before eating



Algorithm 6.10: Dining philosophers (first attempt)

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

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

Dining Philosophers – First Attempt

- This solution deadlocks under an interleaving that has all philosophers pick up their left forks – execute wait (fork[i]) – before any of them tries to pick up a right fork
- All are waiting for a right fork and no process will execute a single instruction

Dining Philosophers – Second Attempt

- One way to ensure a liveness in a solution to the dining philosophers problem is to limit the number of philosophers entering the room to four

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)
```

Dining Philosophers – Third Attempt

- Another solution that is free from starvation is an asymmetric algorithm which has first four philosophers execute the original solution but the fifth philosopher waits first for the right fork and then for the left fork

Algorithm 6.12: Dining philosophers (third attempt)

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

philosopher 4

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

Possible Solutions

- Use a waiter
- Execute different code for odd/even
- Give them another chopstick
- Allow at most 4 philosophers at the table
- Randomized (Lehmann-Rabin)

Reader's Writer's Problem

- **A data structure is shared among a number of concurrent processes:**
  - Readers – Only read the data; They do not perform updates.
  - Writers – Can both read and write.

- **Problem**
  - Allow multiple readers to read at the same time.
  - Only one writer can access the shared data at the same time.
  - If a writer is writing to the data structure, no reader may read it.

First Solution

- Shared Data:
  - The data structure
  - Integer readcount initialized to 0.
  - Semaphore mutex initialized to 1.
    - Protects readcount
  - Semaphore write initialized to 1.
    - Makes sure the writer doesn't use data at the same time as any readers

First Solution

- The structure of a **writer** process:

```
while (true) {
        P (write) ;

            writing is performed

        V (write) ;
}
```

First Solution – Readers have Priority

- The structure of a **reader** process:

```
while (true) {
        P (mutex) ;
            readcount ++ ;
            if (readcount == 1)
        P (write) ;
      V (mutex)
        reading is performed
       P (mutex) ;
            readcount - - ;
            if (readcount  == 0)
                V (write) ;
        V (mutex) ;
  }
```

First Solution – Readers have Priority

- The structure of a **reader** process:

```
while (true) {
        P (mutex) ;
            readcount ++ ;
            if (readcount == 1)
        P (write) ;
      V (mutex)
         reading is performed
       P (mutex) ;
            readcount - - ;
            if (readcount  == 0)
                V (write) ;
         V (mutex) ;
  }
```

This solution is not perfect:

What if a writer is waiting to write but there are readers that read all the time?

Writers are subject to starvation!

Second Solution – Writer Priority

- Extra semaphores and variables:
  - Semaphore read initialized to 1 – restrains readers when a writer wants to write.
  - Integer writecount initialized to 0 – counts waiting writers.
  - Semaphore write_mutex initialized to 1 – controls the updating of writecount.
  - Semaphore mutex now called read_mutex
  - Queue semaphore used only in the reader

Second Solution – Writer Priority

The writer:

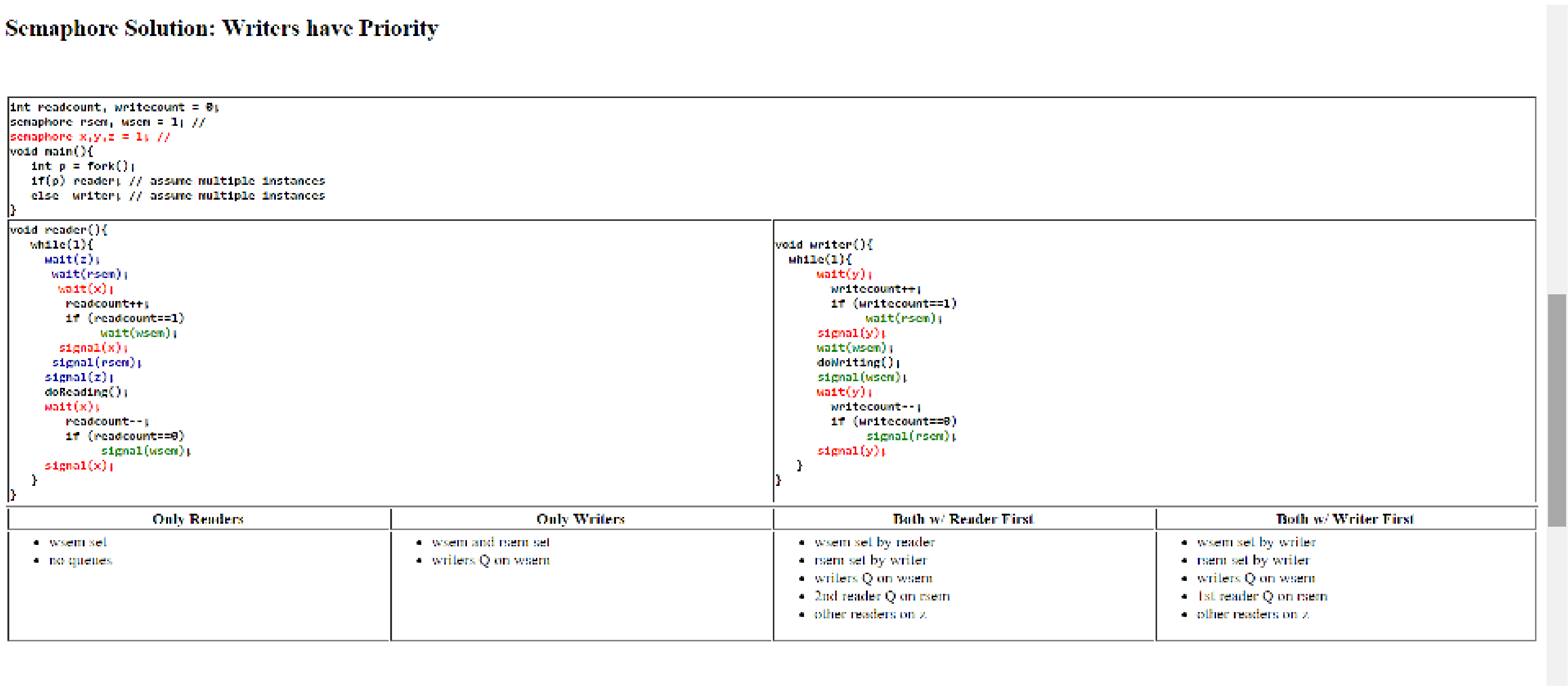
```
while (true) {
    P(write_mutex)
        writecount++;   //counts number of waiting writers
        if (write_count ==1)
            P(read)
    V(write_mutex)

    P (write) ;
        writing is performed
     V(write) ;

    P(write_mutex)
        writecount--;
        if (writecount ==0)
            V(read)
    V(write_mutex)
}
```

Second Solution – Writer Priority

The reader:

```
while (true) {
    P(queue)
        P(read)
          P(read_mutex) ;
            readcount ++ ;
            if (readcount == 1)
            P(write) ;
          V(read_mutex)
       V(read)
    V(queue)
    reading is performed
    P(read_mutex) ;
       readcount - - ;
       if (readcount  == 0)
          V(write) ;
    V(read_mutex) ;
}
```

Queue semaphore, initialized to 1: Since we don't want to allow more than one reader at a time in this section (otherwise the writer will be blocked by multiple readers when doing P(read).)



Semaphore Solution: Writers have Priority

```
int readcount, writecount = 0;
semaphore rsem, wsem = 1; //
semaphore x,y,z = 1; //
void main(){
   int p = fork();
   if(p) reader; // assume multiple instances
   else  writer; // assume multiple instances
}
```

```
void reader(){
   while(1){
     wait(z);
      wait(rsem);
       wait(x);
        readcount++;
        if (readcount==1)
             wait(wsem);
       signal(x);
      signal(rsem);
     signal(z);
     doReading();
     wait(x);
        readcount--;
        if (readcount==0)
             signal(wsem);
     signal(x);
   }
}
```

```
void writer(){
  while(1){
      wait(y);
        writecount++;
        if (writecount==1)
             wait(rsem);
      signal(y);
      wait(wsem);
      doWriting();
      signal(wsem);
      wait(y);
        writecount--;
        if (writecount==0)
             signal(rsem);
      signal(y);
   }
}
```

Only Readers	Only Writers	Both w/ Reader First	Both w/ Writer First
• wsem set • no queues	• wsem and rsem set • writers Q on wsem	• wsem set by reader • rsem set by writer • writers Q on wsem • 2nd reader Q on rsem • other readers on z	• wsem set by writer • rsem set by writer • writers Q on wsem • 1st reader Q on rsem • other readers on z