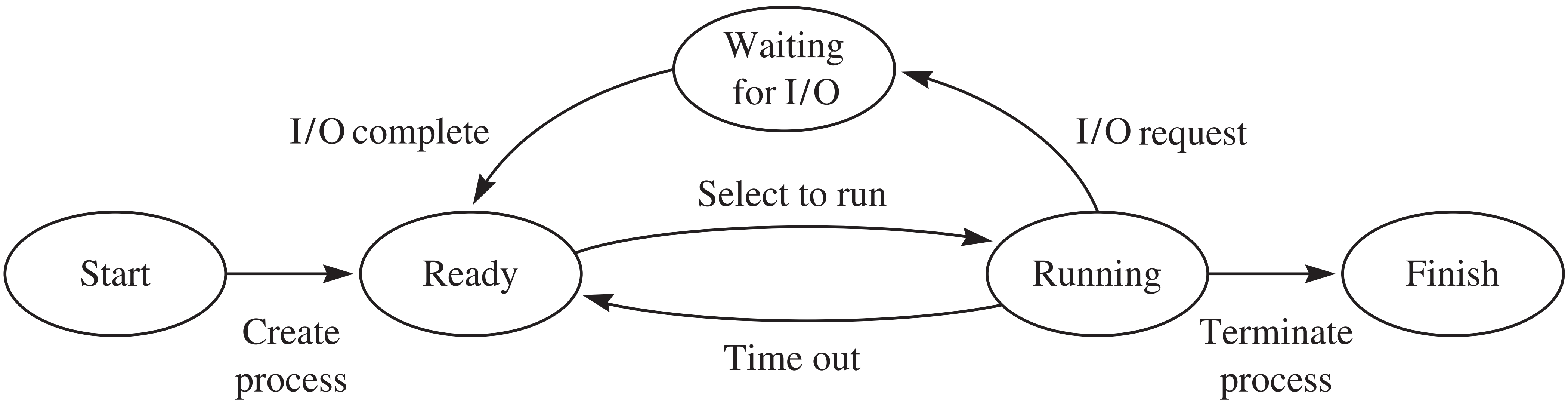
Semaphores

Semaphore

Purpose: To prevent inefficient spin lock of the await statement.

Idea: Instead of spinning, the process calls a method that puts it in a special queue of PCBs, the “blocked on semaphore” queue.


Waiting
for I/O
I/O complete
I/O request
Select to run
Start
Ready
Running
Finish
Create
process
Time out
Terminate
process

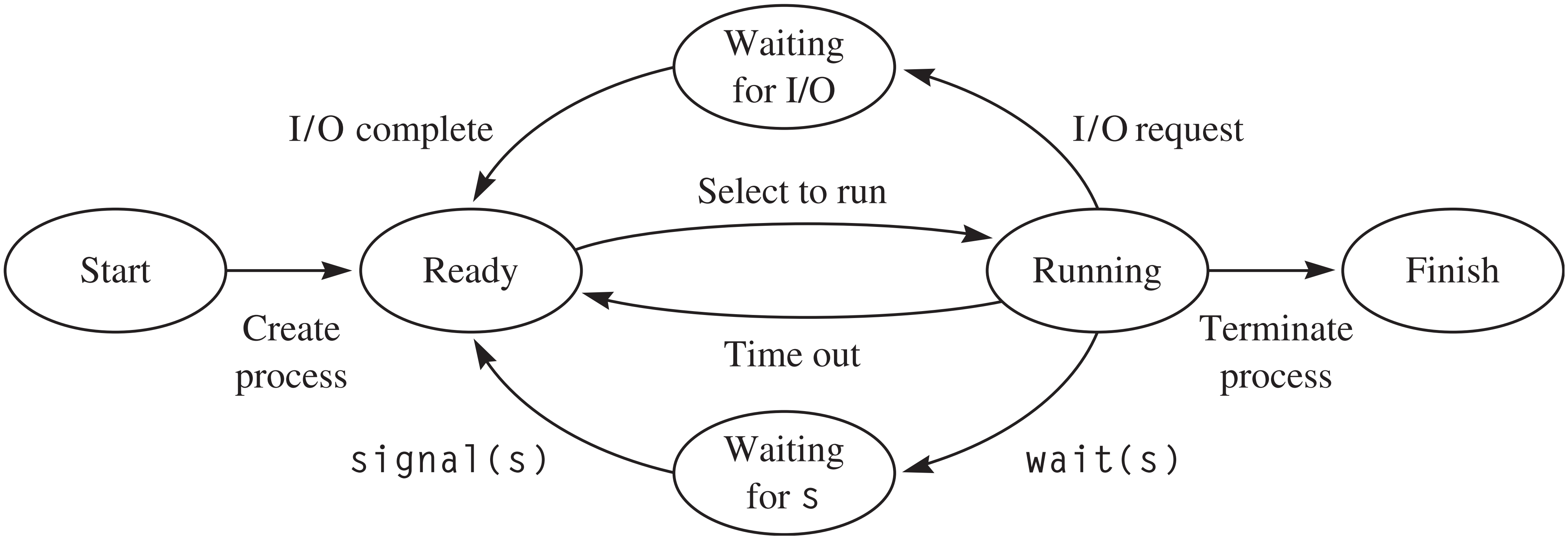
Add fourth state for a process

1. Ready in the ready queue

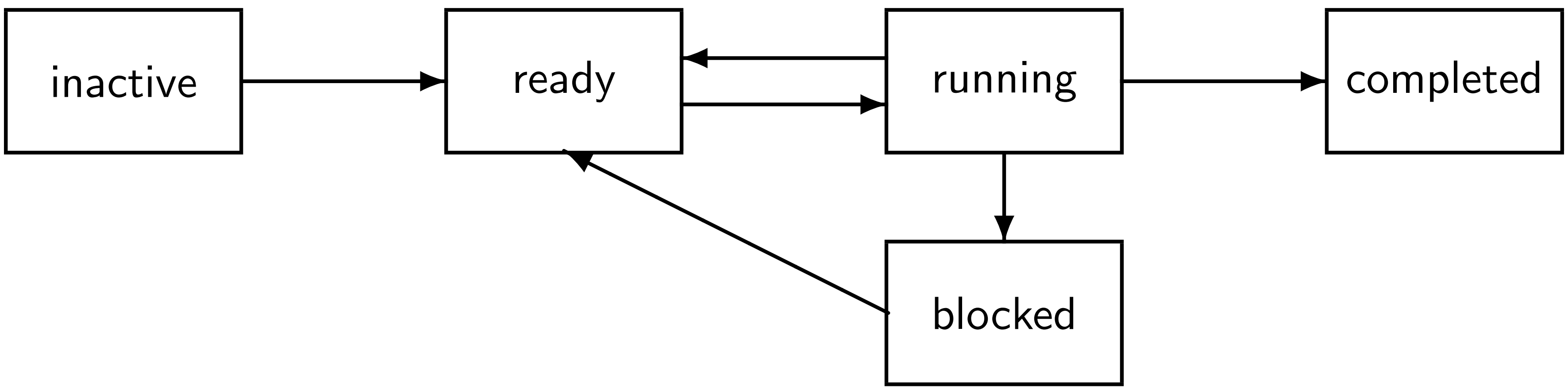
2. Running in the cpu

3. Waiting for I/O in the I/O-wait queue

4. Blocked in the blocked-on-semaphore queue


Waiting
for I/O
I/O complete
I/O request
Select to run
Start
Ready
Running
Finish
Create
process
Time out
Terminate
process
signal(s)
Waiting
for s
wait(s)

State Changes of a Process



Semaphore type

Semaphore S has two fields:

integer $S.V$

queue $S.L$

Semaphore atomic operations
Memorize this

Semaphore atomic operations
Memorize this

Initialization

semaphore $S \leftarrow (k, \emptyset)$

Semaphore atomic operations
Memorize this

Initialization

semaphore $S \leftarrow (k, \emptyset)$

wait(S)

    if $S.V > 0$

        $S.V \leftarrow S.V - 1$

    else

        $S.L \leftarrow S.L \cup p$

        $p.state \leftarrow$ blocked

Semaphore atomic operations
Memorize this

Initialization

semaphore $S \leftarrow (k, \emptyset)$

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(S)

if $S.L = \emptyset$

$S.V \leftarrow S.V + 1$

else

let q be some process in $S.L$

$S.L \leftarrow S.L - \{q\}$

$q.state \leftarrow$ ready

Binary semaphore

The value *S.V* is only allowed to be 0 or 1.

Also called “mutex” for mutual exclusion.

Critical section

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)

Class exercise

Construct the first part of the state transition diagram: p, p, q, q, p, p, (p and q), ...

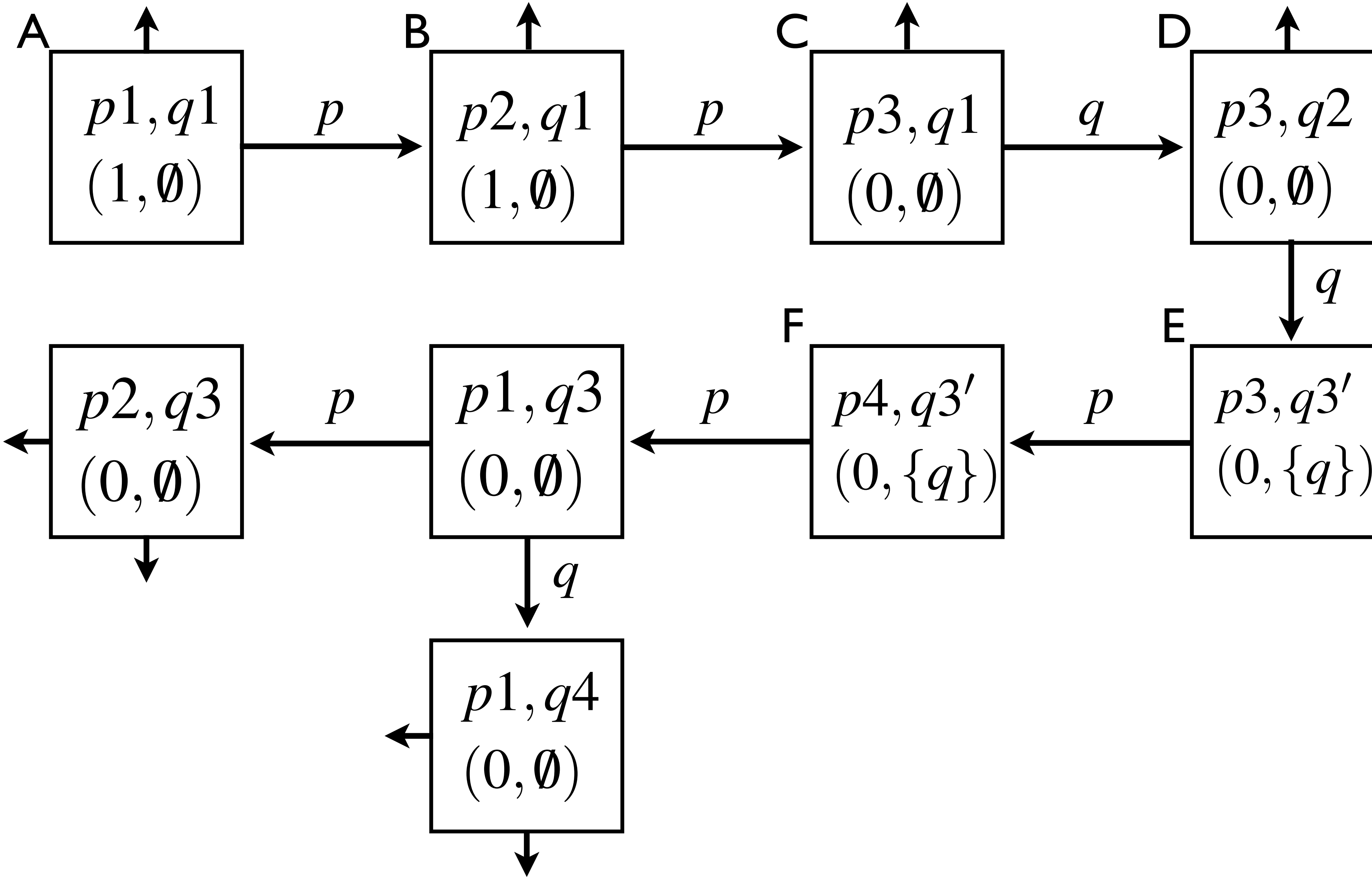
Algorithm 6.1: Critical section with semaphores (two processes)	
binary semaphore S ← $(1, \emptyset)$	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

$\text{wait}(S)$

- if $S.V > 0$
  - $S.V \leftarrow S.V - 1$
- else
  - $S.L \leftarrow S.L \cup p$
  - $p.state \leftarrow \text{blocked}$

$\text{signal}(S)$

- if $S.L = \emptyset$
  - $S.V \leftarrow S.V + 1$
- else
  - let q be some process in $S.L$
  - $S.L \leftarrow S.L - \{q\}$
  - $q.state \leftarrow \text{ready}$



States E and F have no transition on q because q is blocked. q3’ means q3 cannot execute.

Class exercise

Construct the complete state transition diagram

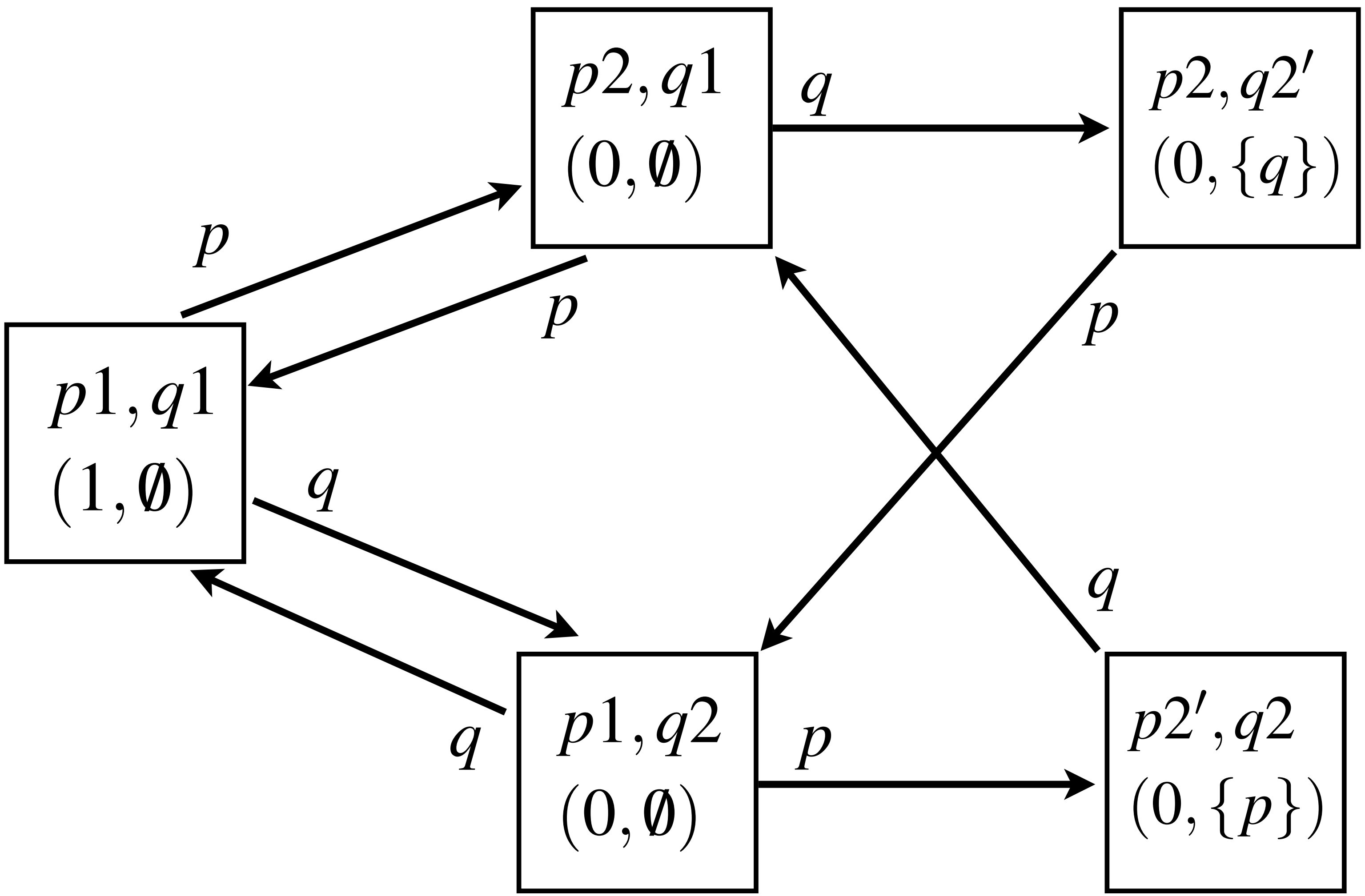
Algorithm 6.2: Critical section with semaphores (two proc., abbrev.)			
binary semaphore S ← $(1, \emptyset)$			
p		**q**	
	loop forever		loop forever
p1:	wait(S)	q1:	wait(S)
p2:	signal(S)	q2:	signal(S)

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(S)

- if $S.L = \emptyset$
  - $S.V \leftarrow S.V + 1$
- else
  - let q be some process in $S.L$
  - $S.L \leftarrow S.L - \{q\}$
  - $q.state \leftarrow$ ready


$p2, q1$
$(0, \emptyset)$
$p2, q2'$
$(0, \{q\})$
$p1, q1$
$(1, \emptyset)$
$p1, q2$
$(0, \emptyset)$
$p2', q2$
$(0, \{p\})$
p
p
q
q
q
p
p
q

Correctness

Correctness

Mutual exclusion: Yes, no state $p2, q2$

Correctness

Mutual exclusion: Yes, no state $p2, q2$

Deadlock free: Yes, no state with both blocked

Correctness

Mutual exclusion: Yes, no state $p2, q2$

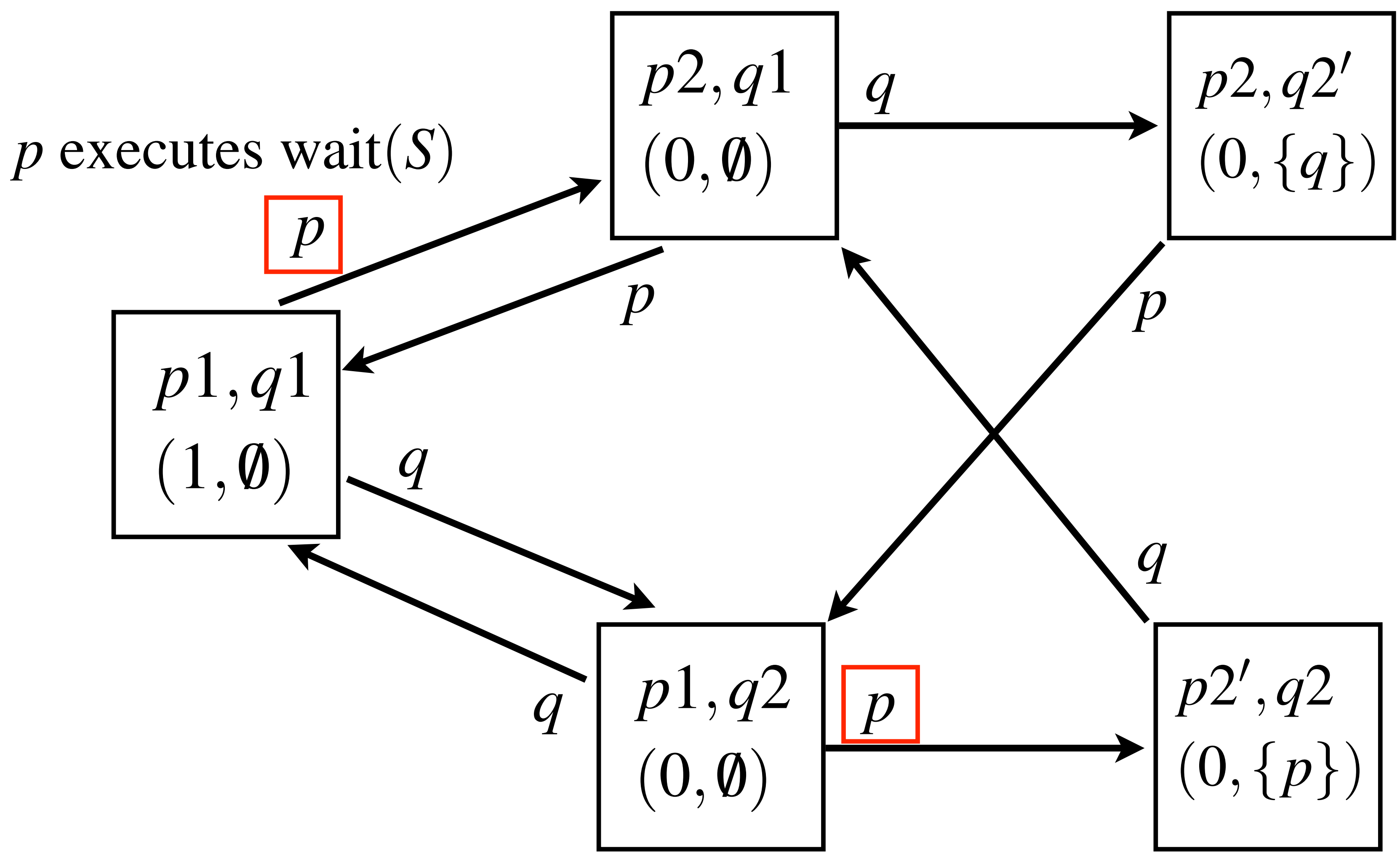
Deadlock free: Yes, no state with both blocked

Starvation free: Yes (next slide)

If p executes wait(S) either

(a) p not blocked, can execute signal(S) so q can proceed

(b) p blocked, q will proceed

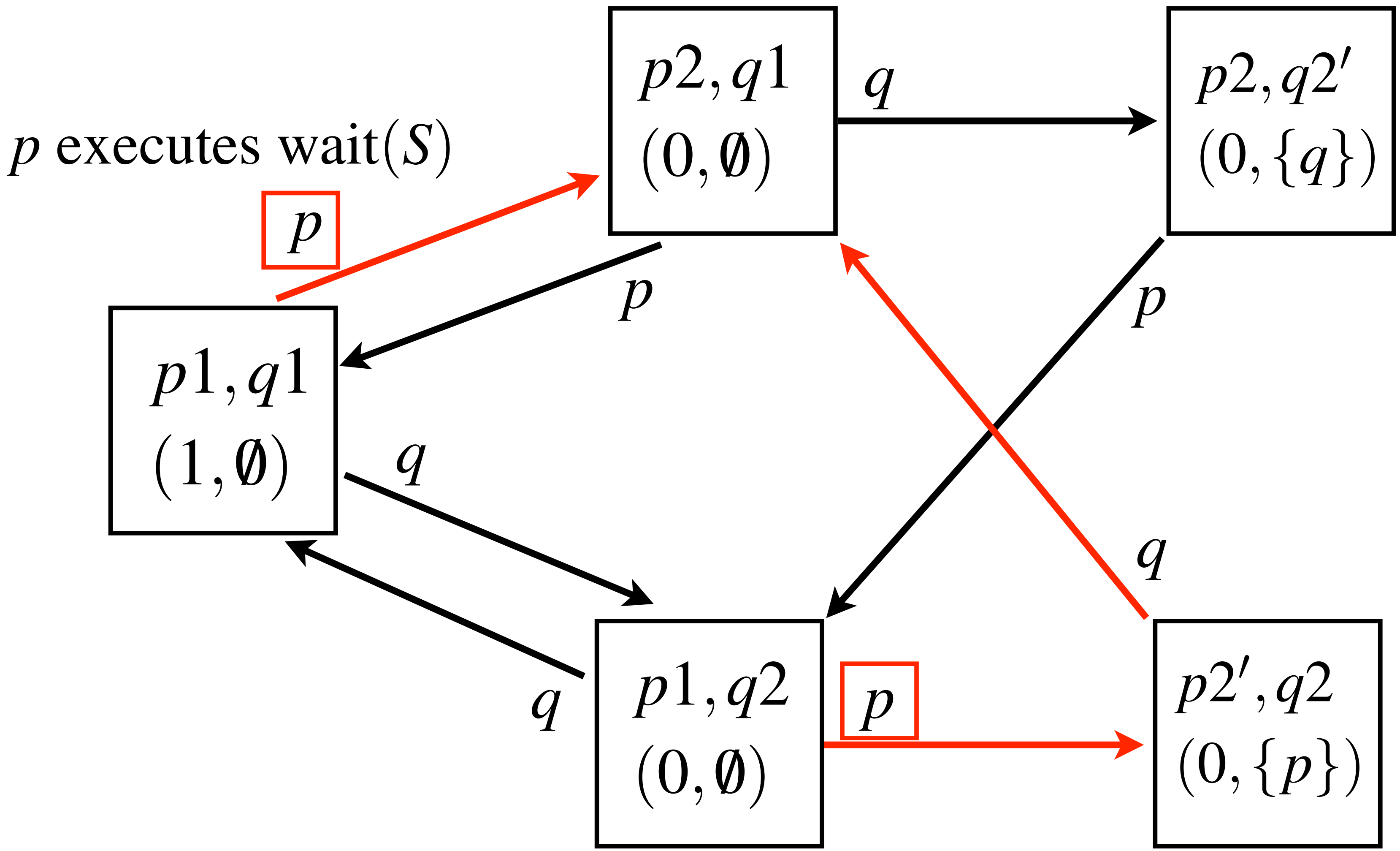

p executes wait(S)
$p2, q1$
$(0, \emptyset)$
$p2, q2'$
$(0, \{q\})$
$p1, q1$
$(1, \emptyset)$
$p1, q2$
$(0, \emptyset)$
$p2', q2$
$(0, \{p\})$
p
q
p
q
p
q
q
p

Complete execution of wait(S)

When wait(S) causes a process to block, the wait statement has not been completely executed.

The wait(S) statement completes its execution when it is unblocked.

Complete execution of wait(S)



The strength of a semaphore

A strong semaphore uses a queue (FIFO) of blocked processes. The process unblocked is the one in the queue for the longest time.

A weak semaphore uses a set of blocked processes. The process unblocked is unpredictable.

The semaphore policy is distinct from the scheduling policy in the ready queue.

Semaphores in C++

C++ has the mutex binary semaphore class.

`wait(S)` is `s.lock()`

`signal(S)` is `s.unlock()`

Demo `Algorithm-6-1.cpp`

Algorithm-6-1.cpp

```
#include <cstdlib>
#include <iostream>
#include <thread>
#include <mutex>
#include "Util450.cpp"
using namespace std;

volatile int n = 0;
mutex s;
```

Algorithm-6-1.cpp

```
void pRun() {
    int temp;
    for (int i = 0; i < 10; i++) {
        randomDelay(20);
        cout << "p.i == " << i << endl;
        // Preprotocol
        s.lock();
        // Critical section
        temp = n;
        randomDelay(10);
        n = temp + 1;
        // Postprotocol
        s.unlock();
    }
}
```

Algorithm-6-1.cpp

```
void qRun() {
    int temp;
    for (int i = 0; i < 10; i++) {
        randomDelay(10);
        cout << "q.i == " << i << endl;
        // Preprotocol
        s.lock();
        // Critical section
        temp = n;
        randomDelay(10);
        n = temp + 1;
        // Postprotocol
        s.unlock();
    }
}
```

Algorithm-6-1.cpp

```
int main(int argc, char **argv) {
    thread p(pRun);
    thread q(qRun);
    p.join();
    q.join();
    cout << "The value of n is " << n << endl;
    return EXIT_SUCCESS;
}
```

Counting Semaphores in C++

C++ does not have a counting semaphore class.

You can build one using a mutex and a condition variable (chapter 7).

Demo `Algorithm-6-1b.cpp`

Predict when `s` is initialized to 2.
Predict when `s` is initialized to 0.

Algorithm-6-1b.cpp

```
#include <cstdlib>
#include <iostream>
#include <thread>
#include <mutex>
#include "Util450.cpp"
using namespace std;

volatile int n = 0;
Semaphore s(1);
```

Semaphore class implemented in Util450.cpp

Algorithm-6-1b.cpp

```
void pRun() {
    int temp;
    for (int i = 0; i < 10; i++) {
        randomDelay(20);
        cout << "p.i == " << i << endl;
        // Preprotocol
        s.wait();
        // Critical section
        temp = n;
        randomDelay(10);
        n = temp + 1;
        // Postprotocol
        s.signal();
    }
}
```

Algorithm-6-1b.cpp

```
void qRun() {
    int temp;
    for (int i = 0; i < 10; i++) {
        randomDelay(10);
        cout << "q.i == " << i << endl;
        // Preprotocol
        s.wait();
        // Critical section
        temp = n;
        randomDelay(10);
        n = temp + 1;
        // Postprotocol
        s.signal();
    }
}
```

Algorithm-6-1b.cpp

```
int main(int argc, char **argv) {
    thread p(pRun);
    thread q(qRun);
    p.join();
    q.join();
    cout << "The value of n is " << n << endl;
    return EXIT_SUCCESS;
}
```

Semaphores in Java

Constructor:

- The first parameter is the integer value of S.V
- The second optional parameter is true for fair scheduling (FIFO, a strong semaphore).
- The default value is false (a weak semaphore).

Semaphores in Java

`wait(S)` is `s.acquire()`

`signal(S)` is `s.release()`

Demo `Algorithm0601.java`

Algorithm0601.java

```
public class Algorithm0601 extends Thread {

   static volatile int n = 0;
   static Semaphore s = new Semaphore(1);
   private int processID;

   Algorithm0601(int pID) {
      processID = pID;
   }
```

Algorithm0601.java

```
public void run() {
   int temp;
   if (processID == 1) { // Process p
      for (int i = 0; i < 10; i++) {
         try {
            randomDelay(20);
            System.out.println("p.i == " + i);
            // Preprotocol
            s.acquire();
            // Critical section
            randomDelay(20);
            temp = n;
            randomDelay(20);
            n = temp + 1;
            // Postprotocol
            s.release();
         } catch (InterruptedException e) {
         }
      }
```

<u>Algorithm0601.java</u>

```
    } else if (processID == 2) { // Process q
        for (int i = 0; i < 10; i++) {
            try {
                randomDelay(20);
                System.out.println("q.i == " + i);
                // Preprotocol
                s.acquire();
                // Critical section
                randomDelay(20);
                temp = n;
                randomDelay(20);
                n = temp + 1;
                // Postprotocol
                s.release();
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Algorithm0601.java

```
   public static void main(String[] args) {
      Algorithm0601 p = new Algorithm0601(1);
      Algorithm0601 q = new Algorithm0601(2);
      p.start();
      q.start();
      try {
         p.join();
         q.join();
      } catch (InterruptedException e) {
      }
      System.out.println("The value of n is " + n);
   }
}
```

Semaphore invariants

For proving correctness without model checking

Theorem 6.1
Memorize this.

Let

$k =$ initial value of $S.V \geq 0$
$\#\text{signal}(S) =$ the number of $signal(S)$ statements executed
$\#\text{wait}(S) =$ the number of $wait(S)$ statements completely executed

Theorem 6.1
Memorize this.

Let

$k =$ initial value of $S.V \geq 0$
$\#\text{signal}(S) =$ the number of $\mathit{signal}(S)$ statements executed
$\#\text{wait}(S) =$ the number of $\mathit{wait}(S)$ statements completely executed

The following expressions are invariant:

$$S.V \geq 0$$
$$S.V = k + \#\text{signal}(S) - \#\mathit{wait}(S)$$

Theorem 6.1
Memorize this.

Let

$k =$ initial value of $S.V \geq 0$
$\#\text{signal}(S) =$ the number of $signal(S)$ statements executed
$\#\text{wait}(S) =$ the number of $wait(S)$ statements completely executed

The following expressions are invariant:

$$S.V \geq 0$$
$$S.V = k + \#\text{signal}(S) - \#wait(S)$$

Can be proved simply by mathematical induction.

Theorem 6.1

Ideas behind proof

Theorem 6.1

Ideas behind proof

$S.V \geq 0$

wait(S) only subtracts 1 from $S.V$ when $S.V > 0$
signal(S) only adds 1 to $S.V$ when the queue is empty

Theorem 6.1

Ideas behind proof

$S.V \geq 0$

wait(S) only subtracts 1 from $S.V$ when $S.V > 0$
signal(S) only adds 1 to $S.V$ when the queue is empty

$S.V = k + \#\text{signal}(S) - \#wait(S)$

If wait(S) blocks a process, it does not subtract 1 from $S.V$,
but its execution has not completed
If signal(S) unblocks a process, it does not add 1 to $S.V$,
but it also triggers the completion of a $\#wait(S)$ statement

Algorithm 6.2

Let $\#CS$ be the number of processes in their critical sections

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

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

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

Algorithm 6.2

Let $\#CS$ be the number of processes in their critical sections

Lemma

$$\#CS + S.V = 1$$

Proof

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

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

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

Algorithm 6.2

Let $\#CS$ be the number of processes in their critical sections

Lemma

$$\#CS + S.V = 1$$

Proof

From the code of Algorithm 6.2

$$\#CS = \#\mathrm{wait}(S) - \#\mathrm{signal}(S)$$

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

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

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

Algorithm 6.2

Let $\#CS$ be the number of processes in their critical sections

Lemma

$$\#CS + S.V = 1$$

Proof

From the code of Algorithm 6.2

$$\#CS = \#\text{wait}(S) - \#\text{signal}(S)$$

= $\langle$Theorem 6.1 with $k = 1\rangle$

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

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

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

Algorithm 6.2

Let $\#CS$ be the number of processes in their critical sections

Lemma

$$\#CS + S.V = 1$$

Proof

From the code of Algorithm 6.2

$$\#CS = \#\text{wait}(S) - \#\text{signal}(S)$$

= $\langle$Theorem 6.1 with $k = 1\rangle$

$$\#CS = 1 - S.V$$

= $\langle$math$\rangle$

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

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

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

Algorithm 6.2

Let $\#CS$ be the number of processes in their critical sections

Lemma

$\#CS + S.V = 1$

Proof

From the code of Algorithm 6.2

$\#CS = \#\text{wait}(S) - \#\text{signal}(S)$

= $\langle$Theorem 6.1 with $k = 1\rangle$

$\#CS = 1 - S.V$

= $\langle$math$\rangle$

$\#CS + S.V = 1$ //

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

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

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

Algorithm 6.2

Mutual exclusion

Proof

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

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

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

Algorithm 6.2

Mutual exclusion

Proof

By the previous Lemma

$\#CS = 1 - S.V$

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

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

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

Algorithm 6.2

Mutual exclusion

Proof

By the previous Lemma

$\#CS = 1 - S.V$

$\Rightarrow \quad \langle \text{Theorem } 6.1, S.V \geq 0 \rangle$

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

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

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

Algorithm 6.2

Mutual exclusion

Proof

By the previous Lemma

$\#CS = 1 - S.V$

$\Rightarrow \quad \langle \text{Theorem 6.1}, S.V \geq 0 \rangle$

$\#CS \leq 1 \quad //$

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

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

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

Algorithm 6.2

Deadlock free

Proof

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

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

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

Algorithm 6.2

Deadlock free

Proof

Deadlock

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

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

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

Algorithm 6.2

Deadlock free

Proof

Deadlock

$\Rightarrow$ $\langle$Code inspection$\rangle$

State is $p1, q1$, both blocked, and $S.V = 0$

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

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

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

Algorithm 6.2

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

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

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

Deadlock free

Proof

Deadlock

$\Rightarrow$ $\langle$Code inspection$\rangle$

State is $p1, q1$, both blocked, and $S.V = 0$

$\Rightarrow$ $\langle$Lemma, $\#CS + S.V = 1\rangle$

Algorithm 6.2

Deadlock free

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

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

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

Proof

Deadlock

$\Rightarrow$ $\langle$Code inspection$\rangle$

State is $p1, q1$, both blocked, and $S.V = 0$

$\Rightarrow$ $\langle$Lemma, $\#CS + S.V = 1\rangle$

$\#CS = 1$, Contradiction //

Algorithm 6.2

Starvation free

Proof

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

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

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

Algorithm 6.2

Starvation free

Proof

p is starved

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

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

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

Algorithm 6.2

Starvation free

Proof

p is starved

$\Rightarrow \quad \langle p \text{ is blocked} \rangle$

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

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

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

Algorithm 6.2

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

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

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

Starvation free

Proof

$$
\begin{aligned}
& p \text{ is starved} \\
\Rightarrow \quad & \langle p \text{ is blocked} \rangle \\
& S = (0, \{p\}) \lor S = (0, \{p, q\})
\end{aligned}
$$

Algorithm 6.2

Starvation free

Proof

$$
\begin{array}{ll}
 & p \text{ is starved} \\
\Rightarrow & \langle p \text{ is blocked} \rangle \\
 & S = (0, \{p\}) \vee S = (0, \{p, q\}) \\
\Rightarrow & \langle \text{Lemma}, \#CS = 1 - S.V = 1 - 0 = 1 \rangle
\end{array}
$$

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

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

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

Algorithm 6.2

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

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

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

Starvation free

Proof

$$
\begin{aligned}
& p \text{ is starved} \\
\Rightarrow \quad & \langle p \text{ is blocked} \rangle \\
& S = (0, \{p\}) \vee S = (0, \{p, q\}) \\
\Rightarrow \quad & \langle \text{Lemma}, \#CS = 1 - S.V = 1 - 0 = 1 \rangle \\
& \#CS = 1, \text{there is one process in its critical section}
\end{aligned}
$$

Algorithm 6.2

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

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

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

Starvation free

Proof

$$
\begin{aligned}
& p \text{ is starved} \\
\Rightarrow \quad & \langle p \text{ is blocked} \rangle \\
& S = (0, \{p\}) \vee S = (0, \{p, q\}) \\
\Rightarrow \quad & \langle \text{Lemma}, \#CS = 1 - S.V = 1 - 0 = 1 \rangle \\
& \#CS = 1 \text{, there is one process in its critical section} \\
\Rightarrow \quad & \langle \text{There are only two processes in the program} \rangle
\end{aligned}
$$

Algorithm 6.2

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

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

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

Starvation free

Proof

$$
\begin{array}{ll}
 & p \text{ is starved} \\
\Rightarrow & \langle p \text{ is blocked} \rangle \\
 & S = (0, \{p\}) \vee S = (0, \{p, q\}) \\
\Rightarrow & \langle \text{Lemma}, \#CS = 1 - S.V = 1 - 0 = 1 \rangle \\
 & \#CS = 1, \text{there is one process in its critical section} \\
\Rightarrow & \langle \text{There are only two processes in the program} \rangle \\
 & q \text{ is in its critical section and } S = (0, \{p\})
\end{array}
$$

Algorithm 6.2

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

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

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

Starvation free

Proof

p is starved

$\Rightarrow \quad \langle p \text{ is blocked} \rangle$

$S = (0, \{p\}) \vee S = (0, \{p, q\})$

$\Rightarrow \quad \langle \text{Lemma}, \#CS = 1 - S.V = 1 - 0 = 1 \rangle$

$\#CS = 1$, there is one process in its critical section

$\Rightarrow \quad \langle \text{There are only two processes in the program} \rangle$

q is in its critical section and $S = (0, \{p\})$

$\Rightarrow \quad \langle q \text{ must execute } signal(S) \rangle$

Algorithm 6.2

Starvation free

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

Proof

p is starved

$\Rightarrow \quad \langle p \text{ is blocked} \rangle$

$S = (0, \{p\}) \vee S = (0, \{p, q\})$

$\Rightarrow \quad \langle \text{Lemma}, \#CS = 1 - S.V = 1 - 0 = 1 \rangle$

$\#CS = 1$, there is one process in its critical section

$\Rightarrow \quad \langle \text{There are only two processes in the program} \rangle$

q is in its critical section and $S = (0, \{p\})$

$\Rightarrow \quad \langle q \text{ must execute } signal(S) \rangle$

p enters its critical section and is not starved //

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

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

The CS problem with more than two processes

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)

The CS problem with more than two processes

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)

Mutual exclusion: yes

The CS problem with more than two processes

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)

Mutual exclusion: yes
Deadlock free: yes

The CS problem with more than two processes

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)

Mutual exclusion: yes
Deadlock free: yes
Starvation free: Only if the semaphore is strong.

Algorithm 6.4: Critical section with semaphores (N proc., abbrev.)	
binary semaphore S $\leftarrow (1, \emptyset)$	
	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)**	$(0, \{q\})$

Concurrent merge sort

Sort the first half and the second half concurrently.

Constrain the scheduling so that the merge operation can start only after the two sorts have completed.

This is a prerequisite scheduling problem.

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

The producer consumer problem

Assumptions

The producer consumer problem

Assumptions

- Operation append(d, buffer) appends data d

The producer consumer problem

Assumptions

- Operation append(d, buffer) appends data d

- Operation take(buffer) deletes an item and returns it

The producer consumer problem

Assumptions

- Operation append(d, buffer) appends data d
- Operation take(buffer) deletes an item and returns it
- The capacity of buffer is infinite

The producer consumer problem

Assumptions

- Operation append(d, buffer) appends data d
- Operation take(buffer) deletes an item and returns it
- The capacity of buffer is infinite
- Must not delete from an empty buffer

Algorithm 6.6: Producer-consumer (infinite buffer)

infinite queue of dataType buffer ← empty queue

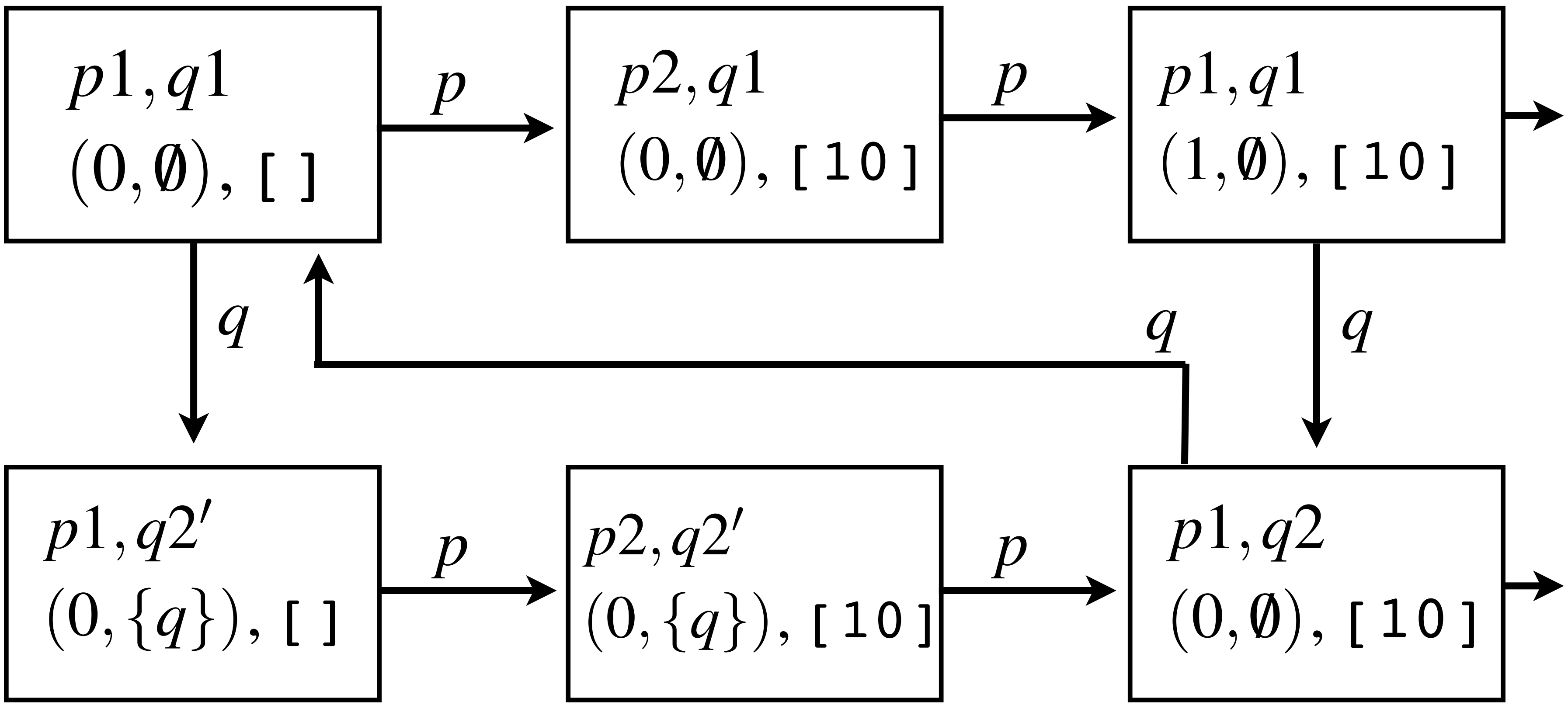
semaphore notEmpty ← $(0, \emptyset)$

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)

Class exercise

Construct the beginning of the state transition diagram for p, p, q, q, ... and q, p, p, ... for the abbreviated algorithm.

Algorithm 6.7: Producer-consumer (infinite buffer, abbreviated)	
infinite queue of dataType buffer ← empty queue semaphore notEmpty ← $(0, \emptyset)$	
producer	**consumer**
dataType d loop forever p1: append(d, buffer) p2: signal(notEmpty)	dataType d loop forever q1: wait(notEmpty) q2: d ← take(buffer)



$p1, q1$
$(0, \emptyset)$, [ ]
p
$p2, q1$
$(0, \emptyset)$, [10]
p
$p1, q1$
$(1, \emptyset)$, [10]
q
q
q
$p1, q2'$
$(0, \{q\})$, [ ]
p
$p2, q2'$
$(0, \{q\})$, [10]
p
$p1, q2$
$(0, \emptyset)$, [10]

The producer-consumer problem

Invariant

In state $p1, q1, nonEmpty.V = \#\text{buffer}$

The producer-consumer problem

Invariant

In state $p1, q1, nonEmpty.V = \#\text{buffer}$

Proof by mathematical induction

Ideas behind proof:

The producer-consumer problem

Invariant

In state $p1, q1$, $nonEmpty.V = \#\text{buffer}$

Proof by mathematical induction

Ideas behind proof:

When the producer produces, it executes signal$(notEmpty)$, which adds 1 to $nonEmpty.V$

The producer-consumer problem

Invariant

In state $p1, q1, nonEmpty.V = \#\text{buffer}$

Proof by mathematical induction

Ideas behind proof:

When the producer produces, it executes signal$(notEmpty)$, which adds 1 to $nonEmpty.V$

When the consumer consumes, it executes wait$(notEmpty)$, which subtracts 1 from $nonEmpty.V$

The bounded buffer producer-consumer problem

The bounded buffer producer-consumer problem

Assumptions

- The capacity of buffer is finite

The bounded buffer producer-consumer problem

Assumptions

- The capacity of buffer is finite

- Must not delete from an empty buffer

- Must not insert into a full buffer

The bounded buffer producer-consumer problem

Assumptions

- The capacity of buffer is finite

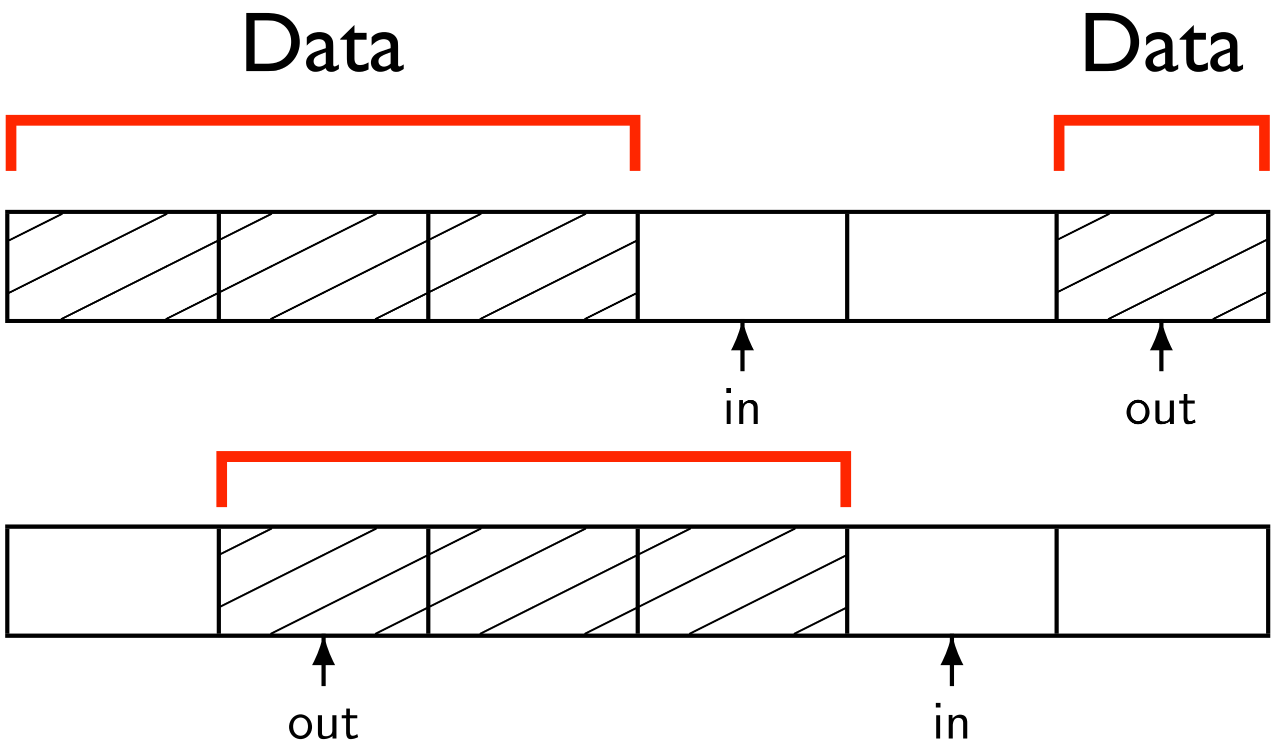
- Must not delete from an empty buffer

- Must not insert into a full buffer

Implementation

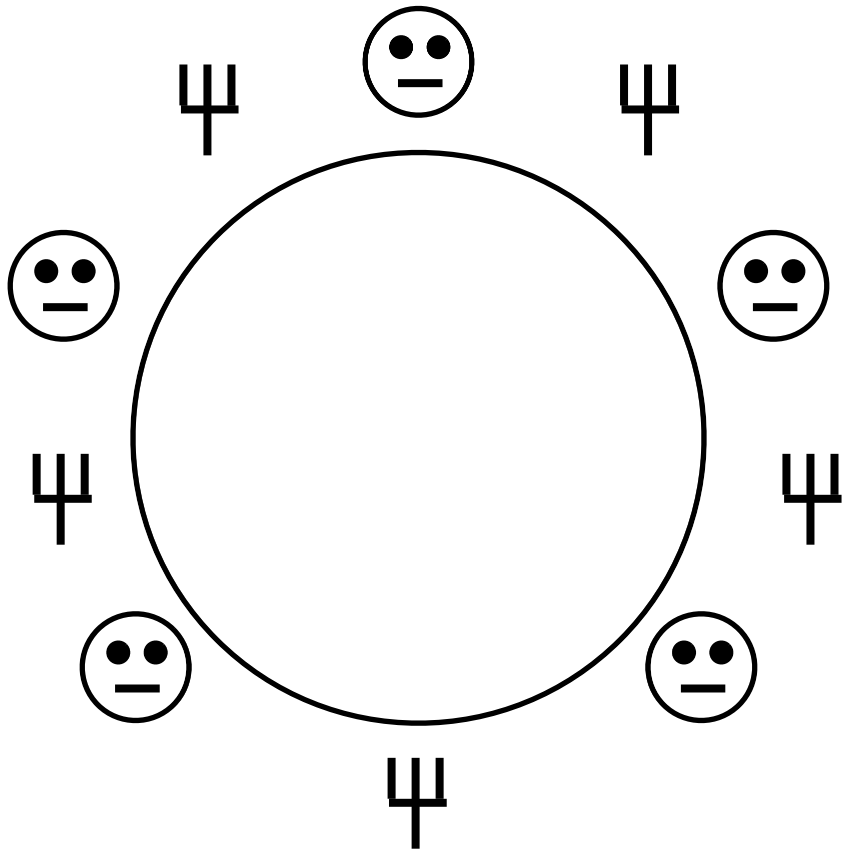
- A circular queue (Exercise for the student)

Circular Buffer



Algorithm 6.19: Producer-consumer (circular buffer)	
dataType array [0..N] buffer integer in, out ← 0 semaphore notEmpty ← $(0, \emptyset)$ semaphore notFull ← $(N, \emptyset)$	
producer	**consumer**
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: wait(notFull)	q2: d ← buffer[out]
p3: buffer[in] ← d	q3: out ← (out+1) modulo N
p4: in ← (in+1) modulo N	q4: signal(notFull)
p5: signal(notEmpty)	q5: consume(d)

The dining philosophers



Activity of each philosopher

- Pick up forks
- Eat
- Put down forks
- Think

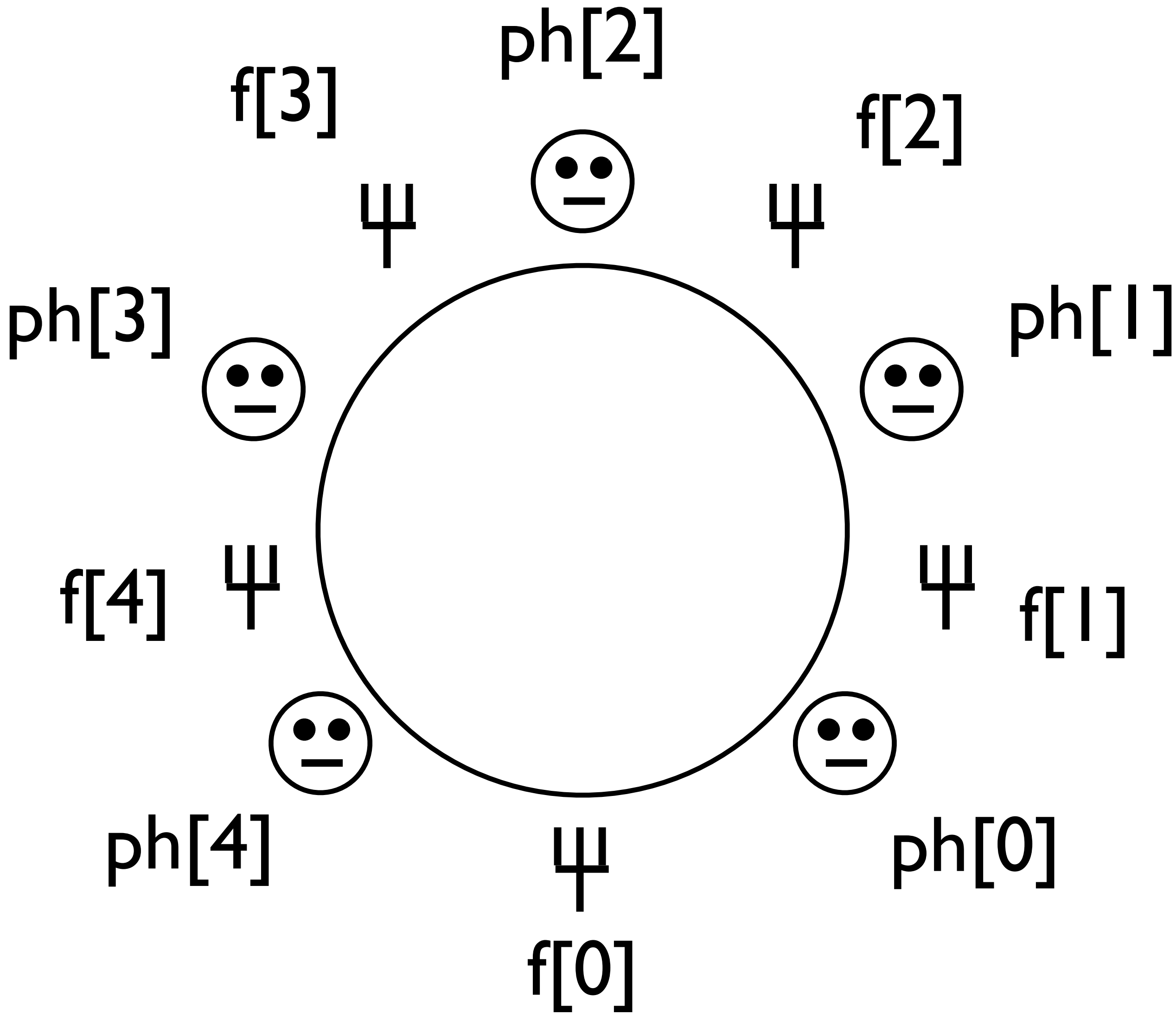
Algorithm 6.9: Dining philosophers (outline)

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

Dining philosophers implementation

- Each philosopher is a process
- Each fork is a semaphore

Array of forks and philosophers



Algorithm 6.10: Dining philosophers (first attempt)
semaphore array [0..4] fork ← [1,1,1,1,1]

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

Algorithm 6.10

Mutual exclusion

Proof

```
semaphore array [0..4] fork ← [1,1,1,1,1]
        loop forever
p1:        think
p2:        wait(fork[i])
p3:        wait(fork[i+1])
p4:        eat
p5:        signal(fork[i])
p6:        signal(fork[i+1])
```

Algorithm 6.10

Mutual exclusion

Proof

By code inspection and mathematical induction, the number of philosophers holding fork i is

$$\#Pi = \#\text{wait}(\text{fork}[i]) - \#\text{signal}(\text{fork}[i])$$

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

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

Algorithm 6.10

```
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])
```

Mutual exclusion

Proof

By code inspection and mathematical induction, the number of philosophers holding fork i is

$$\#Pi = \#\text{wait}(\text{fork}[i]) - \#\text{signal}(\text{fork}[i])$$

$=$ $\langle$Theorem (6.1) $\#\text{wait} - \#\text{signal} = 1 - S.V\rangle$

$$\#Pi = 1 - S.V$$

Algorithm 6.10

Mutual exclusion

Proof

By code inspection and mathematical induction, the number of philosophers holding fork i is

$$\#Pi = \#\text{wait}(\text{fork}[i]) - \#\text{signal}(\text{fork}[i])$$

$= \quad \langle \text{Theorem (6.1) } \#\text{wait} - \#\text{signal} = 1 - S.V \rangle$

$$\#Pi = 1 - S.V$$

$\Rightarrow \quad \langle \text{Theorem (6.1) } S.V \geq 0 \rangle$

$$\#Pi \leq 1 \quad //$$

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

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

Algorithm 6.10

Deadlock free: No

Proof

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

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

Algorithm 6.10

Deadlock free: No

Proof

$P0$ picks up fork 0 on her left.
$P1$ picks up fork 1 on his left.
$P2$ picks up fork 2 on her left.
$P3$ picks up fork 3 on his left.
$P4$ picks up fork 4 on her left.

And now no philosopher can pick up his fork on his right.

```
semaphore array [0..4] fork ← [1,1,1,1,1]
        loop forever
p1:        think
p2:        wait(fork[i])
p3:        wait(fork[i+1])
p4:        eat
p5:        signal(fork[i])
p6:        signal(fork[i+1])
```

Algorithm 6.11

Solves the deadlock problem by simulating a room with a room semaphore that only allows four philosophers in the room at the same time.

To be starvation free, the room semaphore must be strong, but the fork semaphores can be weak.

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

Solves the deadlock problem by having the fourth philosopher pick up his right fork first, and then his left fork. If he blocks on picking up his right fork, his left fork will be available for philosopher 3.

This is an asymmetric solution. One philosopher acts differently from the others.

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])
```