Monitors

Monitor

Purpose: To consolidate the wait and signal operations in a single class.

Instead of having semaphores and critical sections spread throughout the code of different processes, put the critical sections into methods of the monitor class.

Algorithm 7.1

n is an attribute of the monitor instead of being a global variable.

Solves the critical section problem.

Monitor methods are guaranteed to execute atomically.

Algorithm 7.1: Atomicity of monitor operations

```
monitor CS
    integer n ← 0

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

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

Java monitors

There is no special monitor type.

Any class can be a monitor.

The keyword `synchronized` makes a method atomic.

CoSc 450: Source Code

CriticalSection.java

```
package algorithm0701;

import static util450.Util450.*;

public class CriticalSection {
   private int n = 0;

   public synchronized void increment() throws InterruptedException {
      int temp;
      temp = n;
      randomDelay(40);
      n = temp + 1;
   }

   public synchronized int get() {
      return n;
   }
}
```

CoSc 450: Source Code

Algorithm0701
.java

```
public class Algorithm0701 extends Thread {

   private int processID;
   private CriticalSection cs;

   Algorithm0701(int pID, CriticalSection criticalSection) {
      processID = pID;
      cs = criticalSection;
   }
```

CoSc 450: Source Code

Algorithm0701.java

```
public void run() {
    if (processID == 1) { // Process p
       for (int i = 0; i < 10; i++) {
          try {
             System.out.println("p.i == " + i);
             cs.increment();
          } catch (InterruptedException e) {
          }
       }
    } else if (processID == 2) { // Process q
       for (int i = 0; i < 10; i++) {
          try {
             System.out.println("q.i == " + i);
             cs.increment();
          } catch (InterruptedException e) {
          }
       }
    }
 }
```

CoSc 450: Source Code

Algorithm0701
.java

```
    public static void main(String[] args) {
        CriticalSection cs = new CriticalSection();
        Algorithm0701 p = new Algorithm0701(1, cs);
        Algorithm0701 q = new Algorithm0701(2, cs);
        p.start();
        q.start();
        try {
            p.join();
            q.join();
        } catch (InterruptedException e) {
        }
        System.out.println("The value of n is " + cs.get());
    }
}
```

C++ monitors

There is no special monitor type.

You construct a monitor using a `mutex` and a `lock_guard` to make the operations atomic.

CoSc 450: Source Code

Algorithm-7-1
.cpp

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

class CriticalSection {
private:
    int n = 0;
    mutex csMutex;
public:
    void increment() {
        lock_guard<mutex> guard(csMutex);
        int temp;
        temp = n;
        randomDelay(40);
        n = temp + 1;
    }

    int get() {
        lock_guard<mutex> guard(csMutex);
        return n;
    }
};
```

`mutex` for mutual exclusion in monitor

`lock_guard` with mutex for RAII

CoSc 450: Source Code

`Algorithm-7-1.cpp`

```
CriticalSection cs;

void pRun() {
    for (int i = 0; i < 10; i++) {
        cout << "p.i == " << i << endl;
        cs.increment();
   }
}

void qRun() {
    for (int i = 0; i < 10; i++) {
        cout << "q.i == " << i << endl;
        cs.increment();
    }
}

int main() {
    thread p(pRun);
    thread q(qRun);
    p.join();
    q.join();
    cout << "The value of n is " << cs.get() << endl;
    return EXIT_SUCCESS;
}
```

C++ RAII design pattern

RAII – Resource Acquisition Is Initialization

Pronounced "R, A, double I"

Resource acquisition happens during initialization.

Resource deallocation happens during destruction.

RAII in Algorithm-7-1 `increment()` method

`guard` is a local variable of type `lock_guard`, allocated on the run-time stack on the stack frame for `increment()`.

It is created when the method is called, and destroyed automatically when the method terminates.

When `guard` is created it locks `mutex`.
When `guard` is destroyed it unlocks `mutex`.

Therefore, mutual exclusion is guaranteed.

C++ RAII design pattern benefits

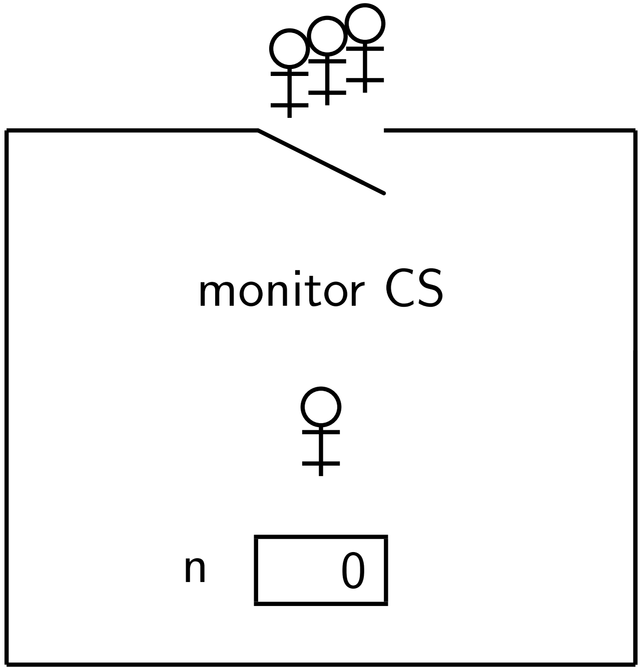
Non-void functions would be difficult, if not impossible, to implement atomically with only `mutex`.

RAII is exception safe.

RAII simplifies resource management.

Most C++ libraries follow the RAII design pattern.

Executing a Monitor Operation



Condition variable

A special monitor variable that has a queue (FIFO) of blocked processes.

A monitor can have more than one condition variable. There is a queue of blocked processes for each condition variable.

Condition variable

There are three operations on condition variable $cond$.

Condition variable

There are three operations on condition variable *cond*.

```
waitC($cond$)
    append $p$ to $cond$ queue
    $p.state \leftarrow$ blocked
    $monitor.lock \leftarrow$ released
```

Condition variable

There are three operations on condition variable *cond*.

```
waitC($cond$)
    append $p$ to $cond$ queue
    $p.state \leftarrow$ blocked
    $monitor.lock \leftarrow$ released

signalC($cond$)
    if $cond$ queue $\neq \emptyset$
        remove head of $cond$ queue and assign to $q$
        $q.state \leftarrow$ ready
```

Condition variable

There are three operations on condition variable $cond$.

```
waitC($cond$)
    append $p$ to $cond$ queue
    $p.state \leftarrow$ blocked
    $monitor.lock \leftarrow$ released

signalC($cond$)
    if $cond$ queue $\neq \emptyset$
        remove head of $cond$ queue and assign to $q$
        $q.state \leftarrow$ ready

empty($cond$)
    return $cond$ queue isEmpty
```

Semaphore	Ben-Ari monitor

Semaphore	Ben-Ari monitor
1. wait(S) may or may not block.	1. waitC($cond$) always blocks.

Semaphore	Ben-Ari monitor
1. wait(S) may or may not block.	1. waitC($cond$) always blocks.
2. signal(S) always has an effect.	2. signalC($cond$) has no effect if *cond* queue is empty.

Semaphore	Ben-Ari monitor
1. wait(S) may or may not block.	1. waitC($cond$) always blocks.
2. signal(S) always has an effect.	2. signalC($cond$) has no effect if *cond* queue is empty.
3. Process unblocked by signal(S) might not resume execution immediately.	3. Process unblocked by signalC($cond$) resumes executing immediately.

The Ben-Ari monitor

Ben-Ari defines his monitor to have "the immediate resumption requirement."

When signalC(cond) executes, the blocked process, if any is blocked, immediately resumes.

The process that executed signalC is put in a signaling queue. (No waiting queue necessary)

Known as "Hoare semantics".

Notes on monitors

Buhr, et. al., “Monitor Classification”, Computing Surveys, March 1995.

Monitor Classification

Peter A. Buhr and Michel Fortier

Dept. of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Michael H. Coffin
EDS Research and Development, 901 Tower Drive, 1st Floor, Troy Michigan 48007-7019, U. S. A.

General monitor

All procedures are mutually exclusive. Each monitor has

- One entry queue
- One queue for each condition variable
- One waiting queue
- One signaler queue

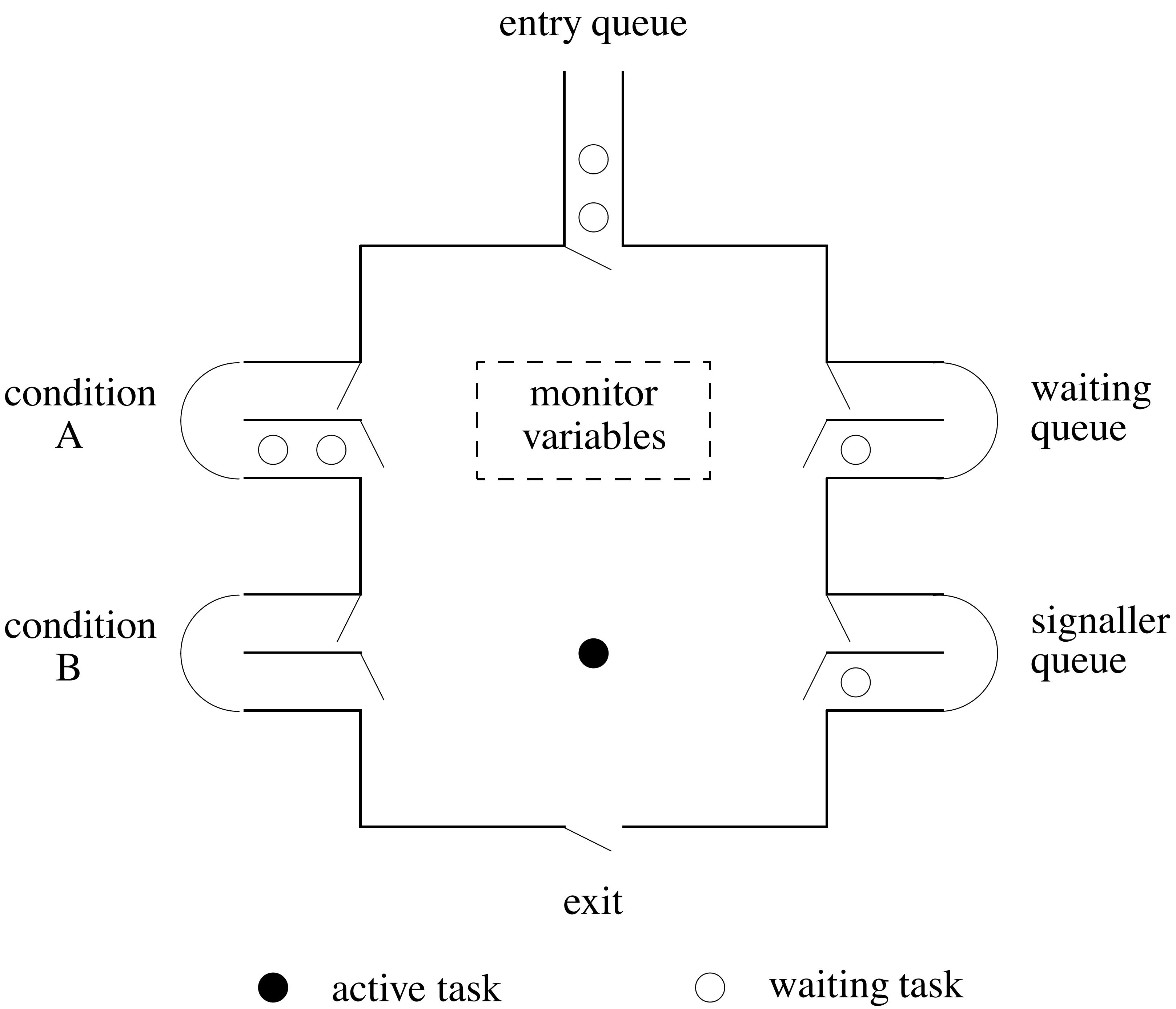


Figure 3: Processes Waiting to use a Monitor

General actions

waitC(cond)
    Blocked on condition queue for cond

General actions

waitC(cond)
    Blocked on condition queue for cond
signalC(cond)
    Signaler moved to signaler queue
    Signaled moved from condition queue to
        wait queue

General actions

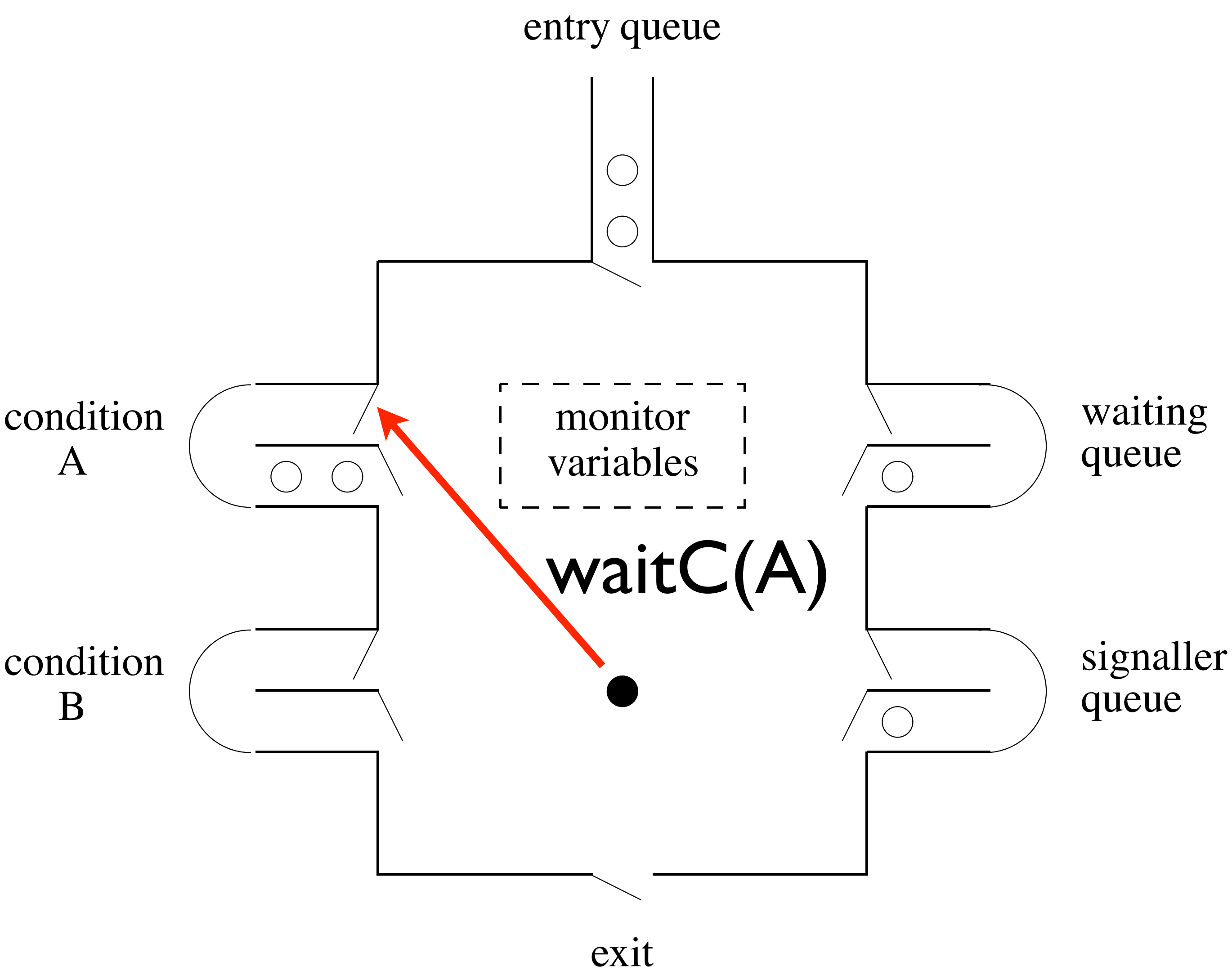
waitC(cond)
- Blocked on condition queue for cond

signalC(cond)
- Signaler moved to signaler queue
- Signaled moved from condition queue to wait queue

Monitor is unlocked

General actions

waitC(cond)
    Blocked on condition queue for cond
signalC(cond)
    Signaler moved to signaler queue
    Signaled moved from condition queue to wait queue
Monitor is unlocked
Monitor chooses from one of the queues which process gets to enter

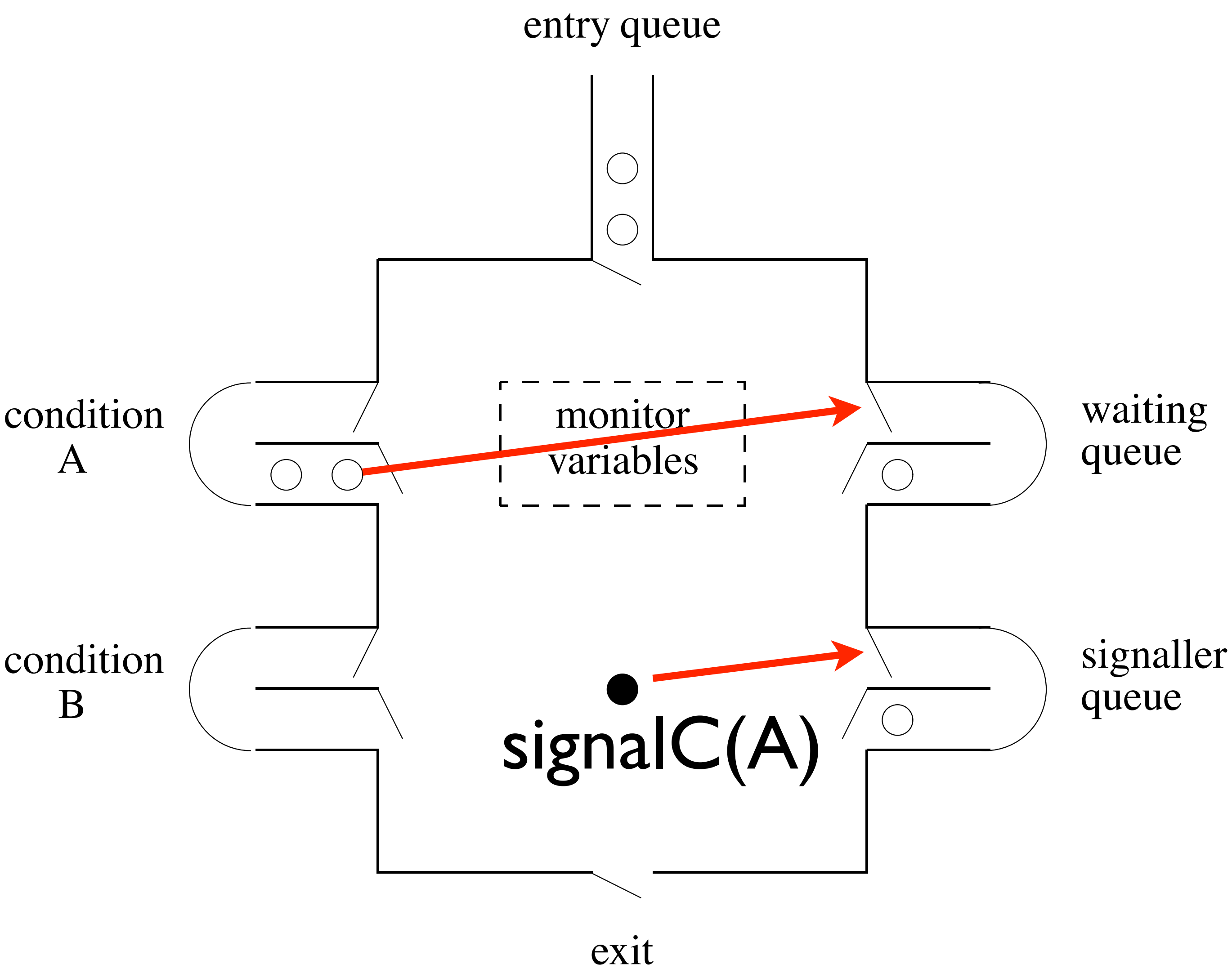


Action of waitC(A)

Process blocked on condition queue A

Monitor is unlocked

An unblocked process is selected to continue



Action of signalC(A)

Signaler to signaler queue, signaled to wait queue

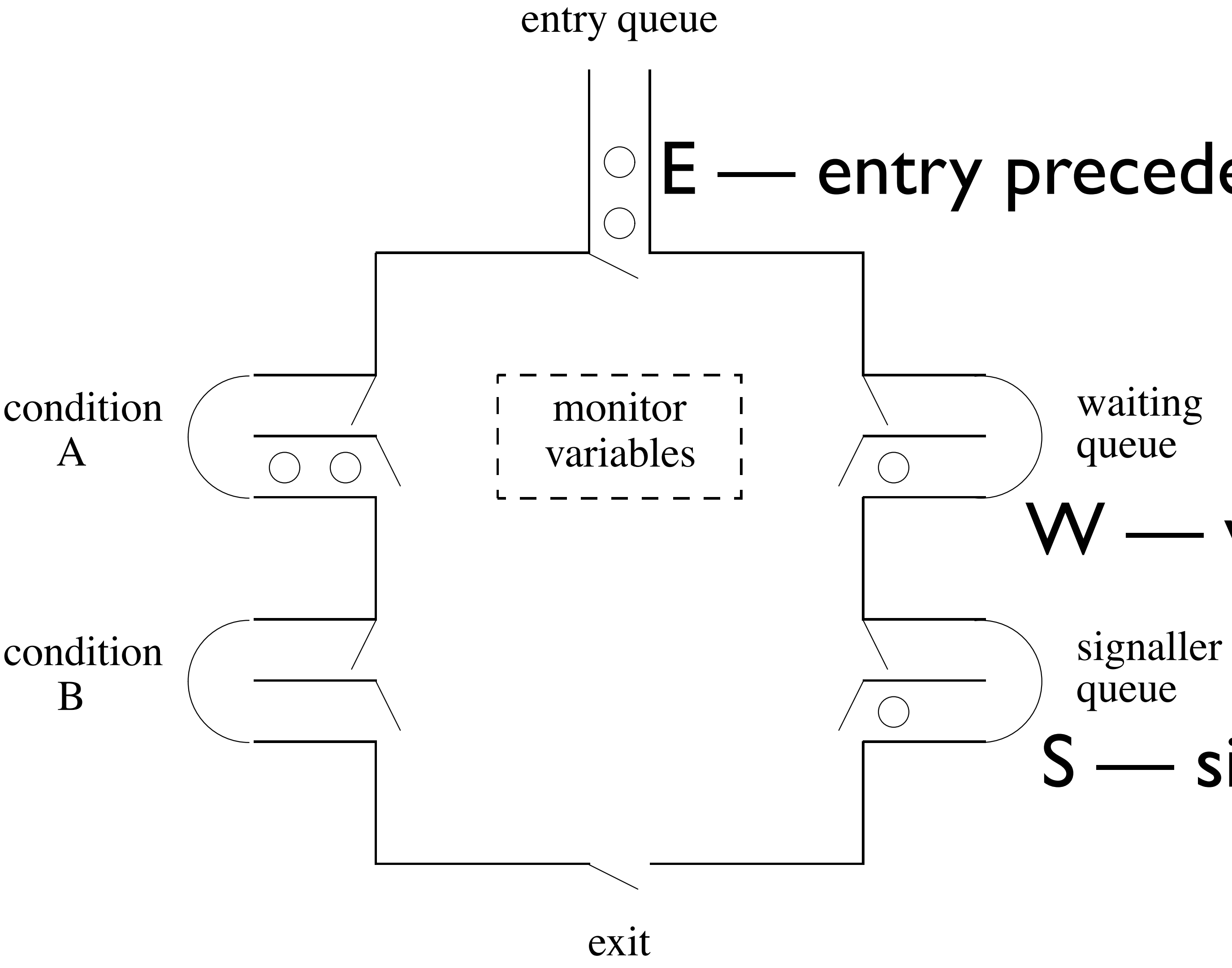
Monitor is unlocked

An unblocked process is selected to continue

Types of monitors

The type of monitor is determined by how the monitor chooses which process gets to enter. Each queue has a specific precedence:

* E — entry precedence
* W — waiting precedence
* S — signaler precedence



E — entry precedence

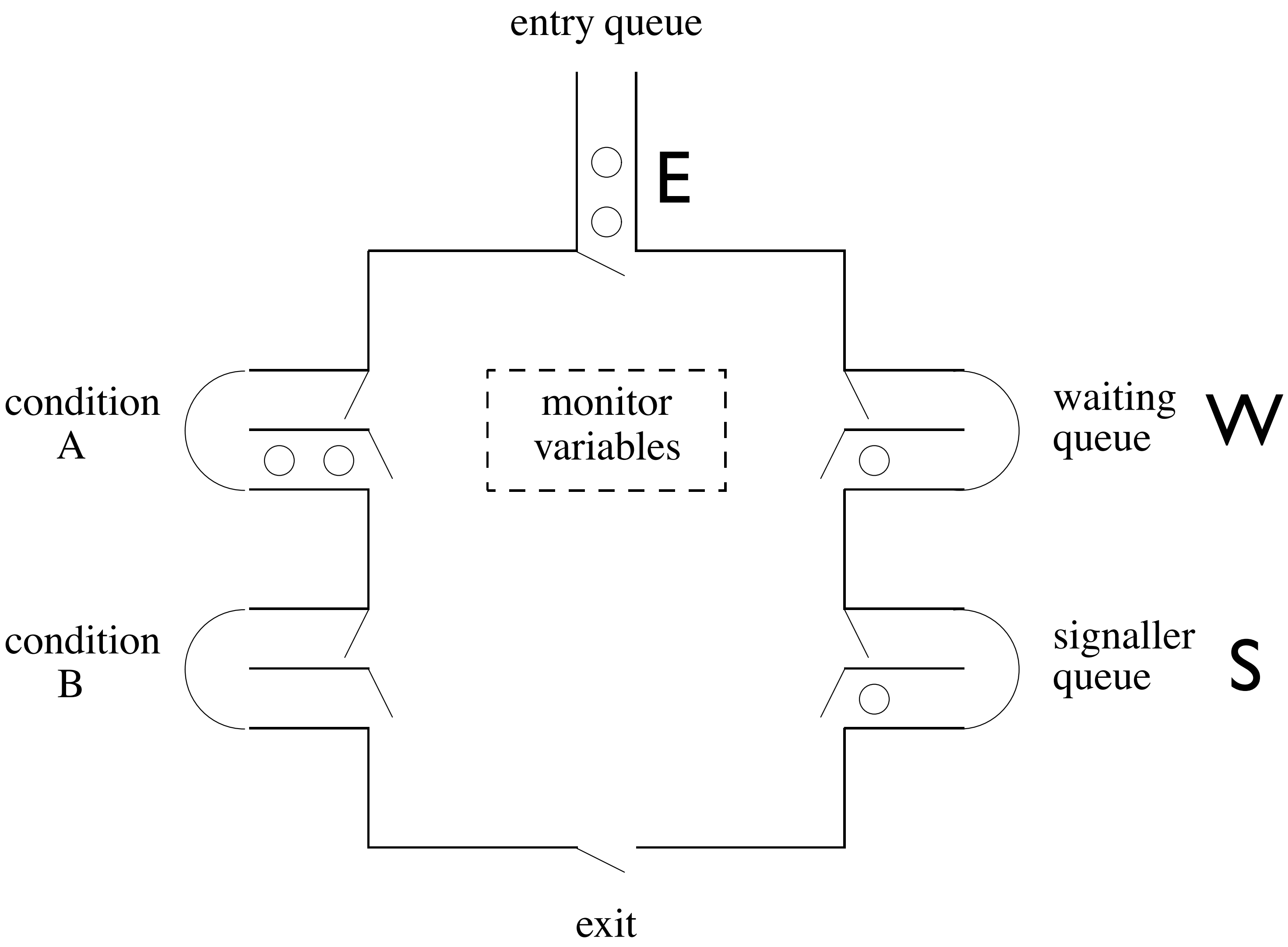
W — waiting precedence

S — signaler precedence

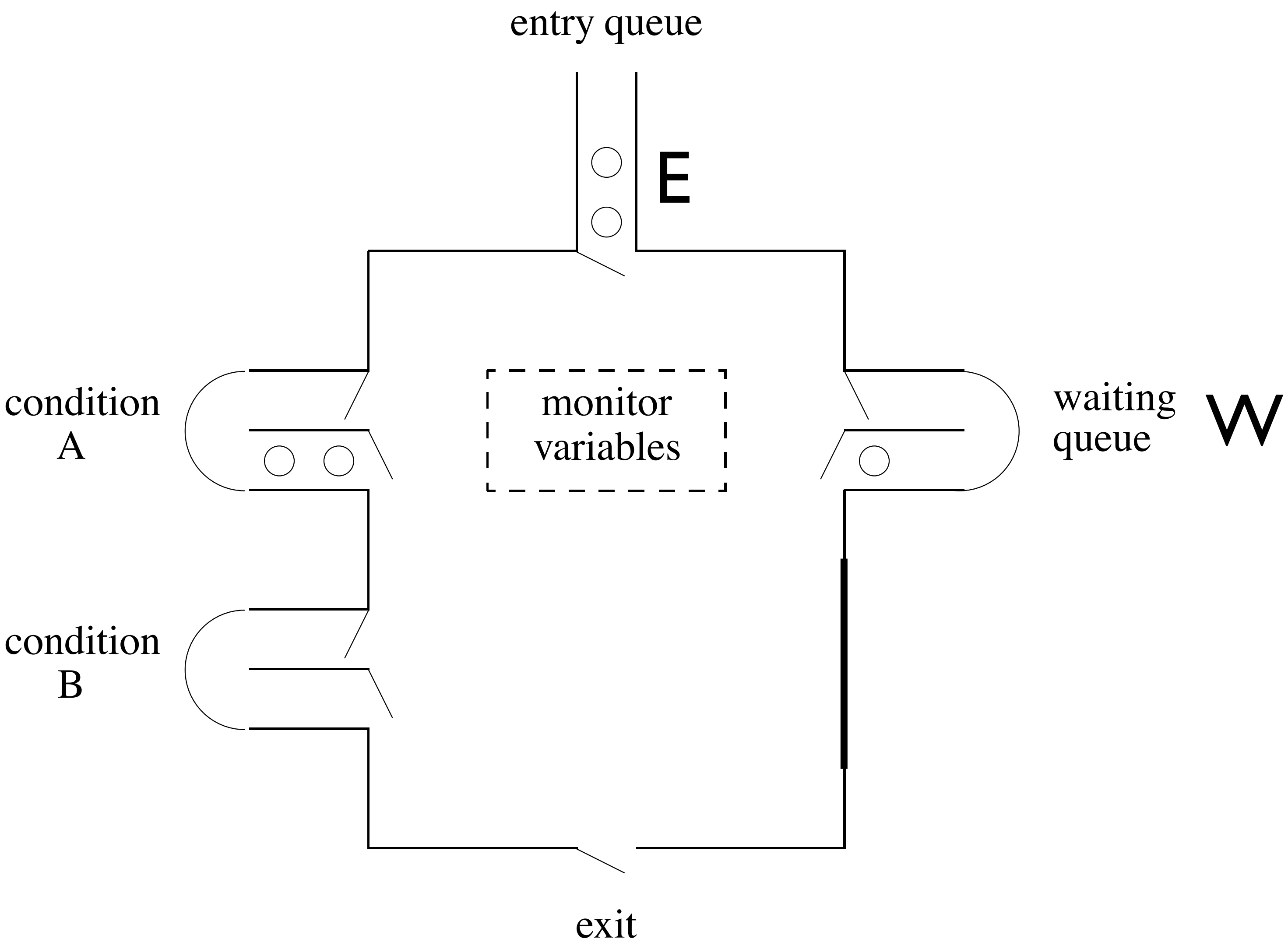
	relative priority	traditional monitor name
1	$E_p = W_p = S_p$	
2	$E_p = W_p < S_p$	Wait and Notify [Lampson and Redell 1980]
3	$E_p = S_p < W_p$	Signal and Wait [Howard 1976a]
4	$E_p < W_p = S_p$	
5	$E_p < W_p < S_p$	Signal and Continue [Howard 1976b]
6	$E_p < S_p < W_p$	Signal and Urgent Wait [Hoare 1974]
7	$E_p > W_p = S_p$	(rejected)
8	$E_p = S_p > W_p$	(rejected)
9	$S_p > E_p > W_p$	(rejected)
10	$E_p = W_p > S_p$	(rejected)
11	$W_p > E_p > S_p$	(rejected)
12	$E_p > S_p > W_p$	(rejected)
13	$E_p > W_p > S_p$	(rejected)

Table 1: Relative Priorities for Internal Monitor Queues

Mesa Semantics, E < W < S
Buhr, C++

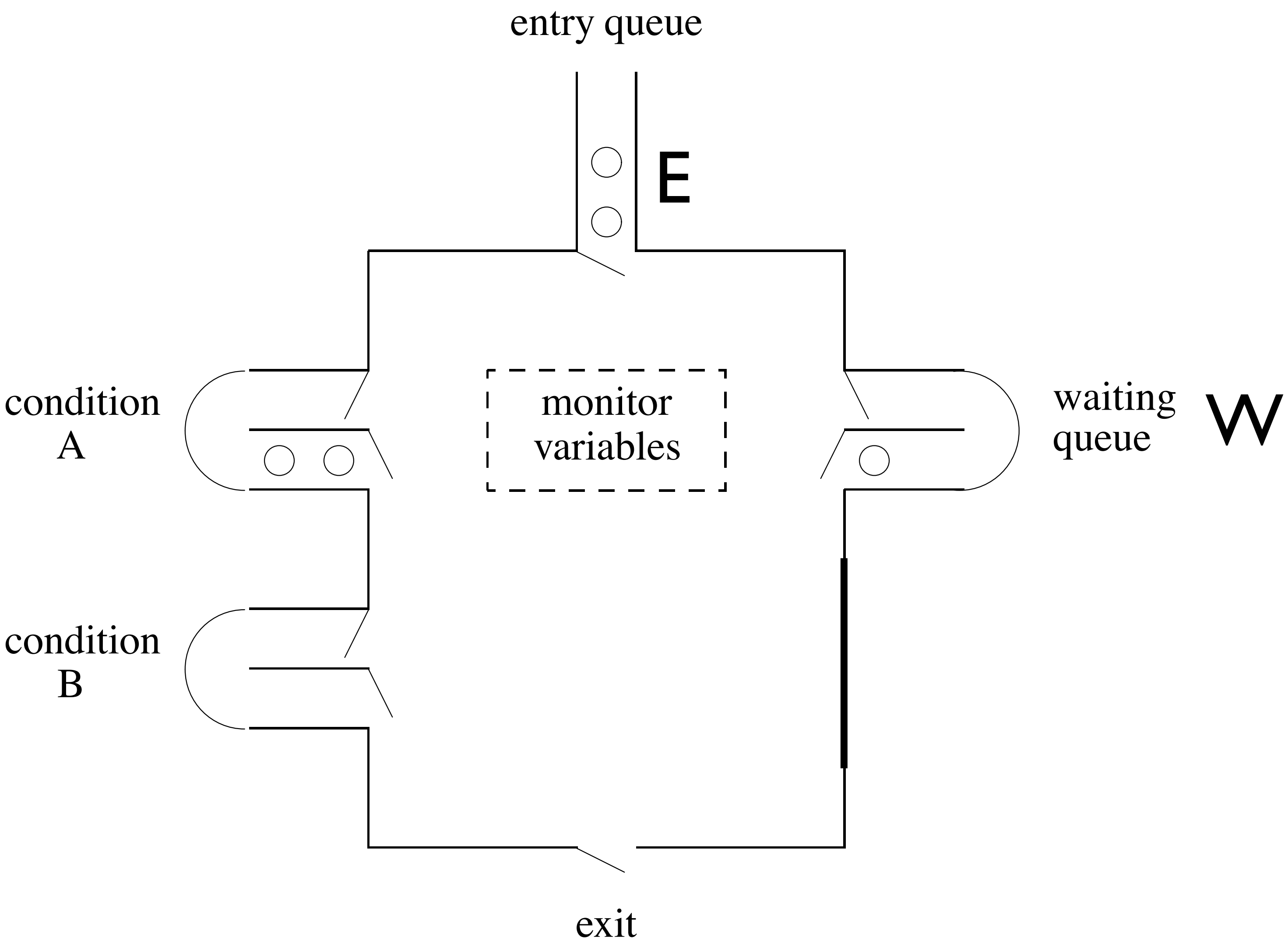


Mesa Semantics, E < W < S
Buhr, C++



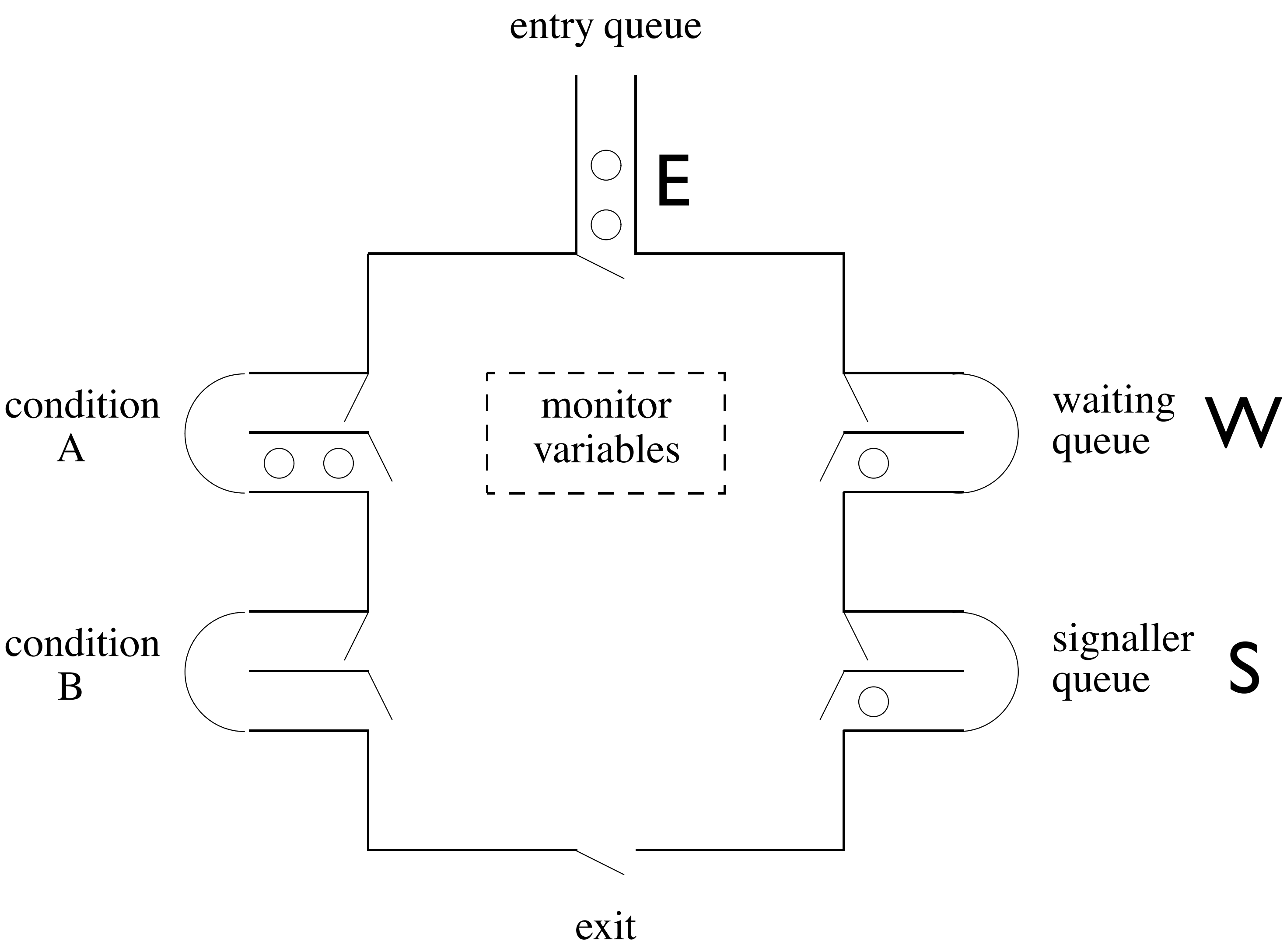
Signaler always picked.
Signaler queue not necessary.

Mesa Semantics, E < W < S
Buhr, C++

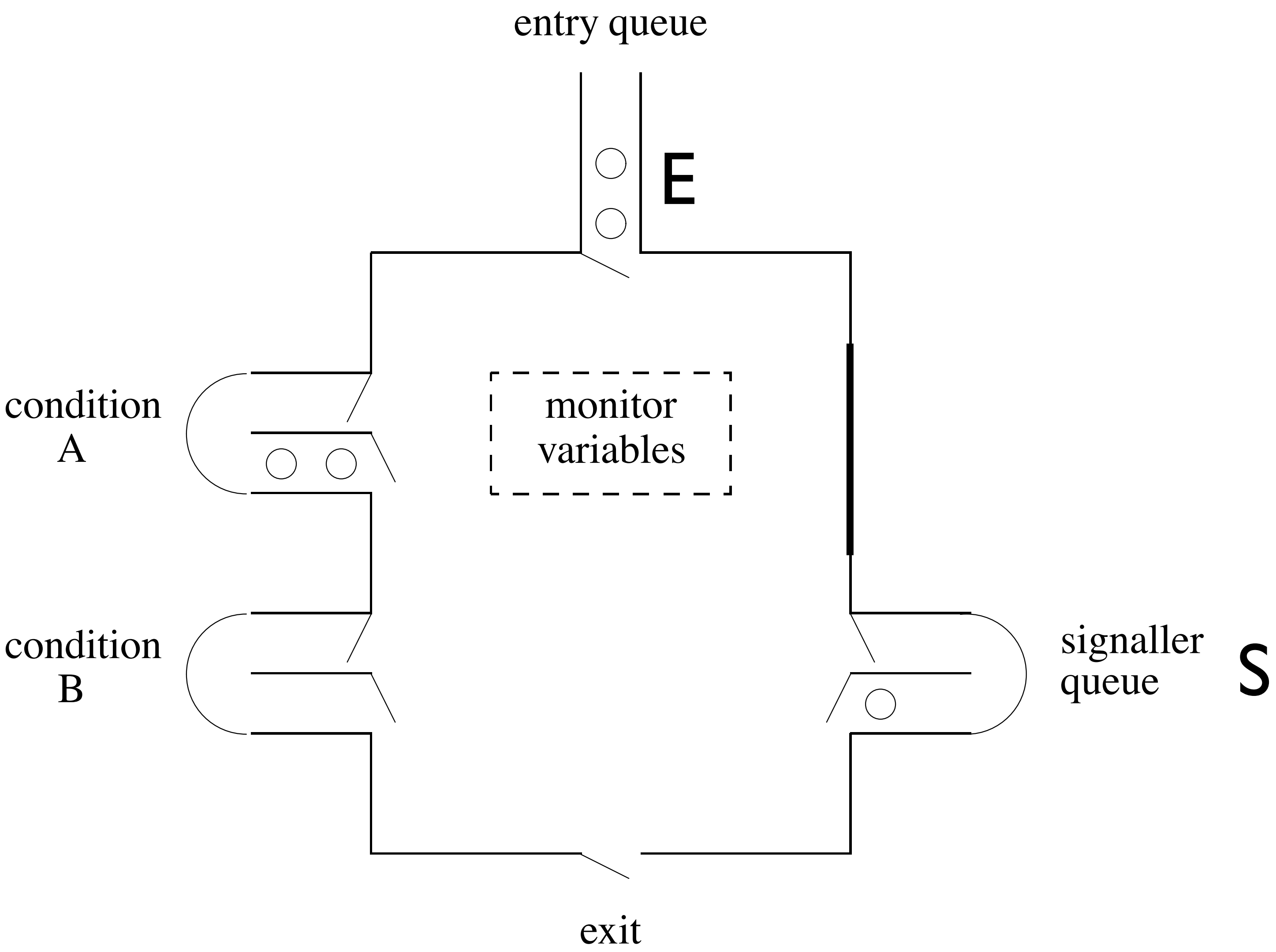


When w eventually picked, condition may no longer be met. May need waitC in the body of a loop instead of an if.

Hoare Semantics, E < S < W
Ben-Ari, C--

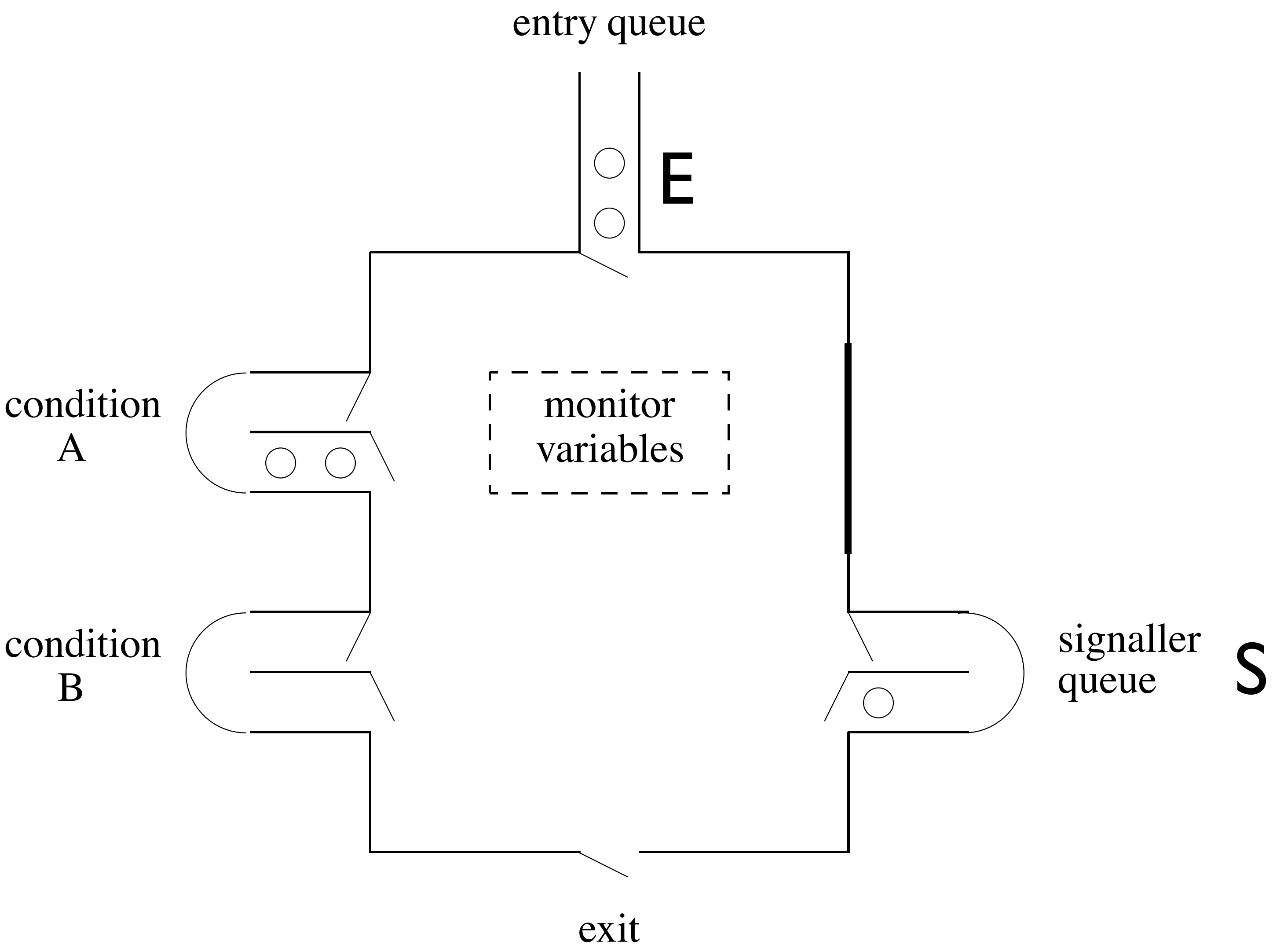


Hoare Semantics, E < S < W
Ben-Ari, C--



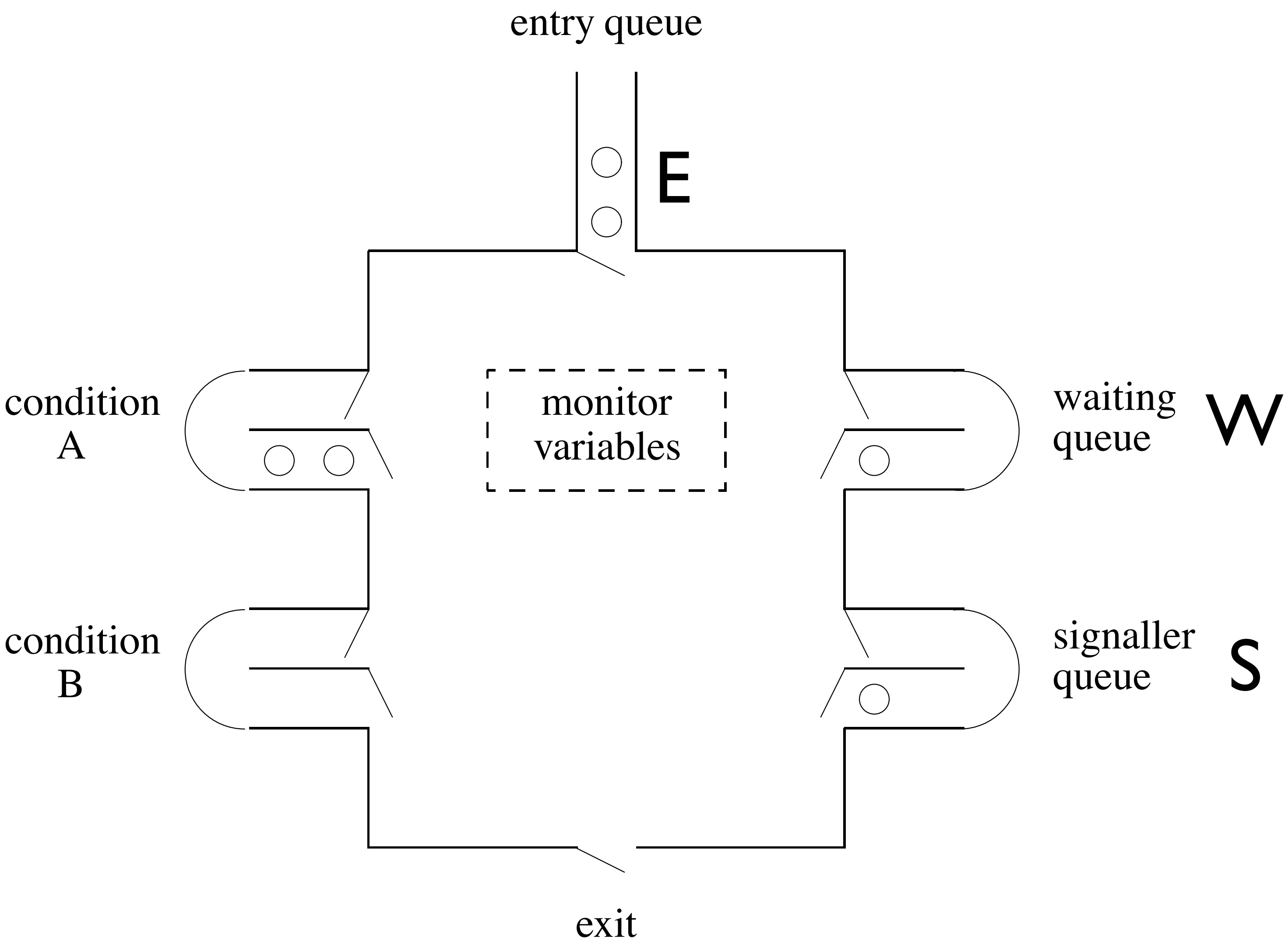
Signaled always picked
Waiting queue not necessary

Hoare Semantics, E < S < W
Ben-Ari, C--

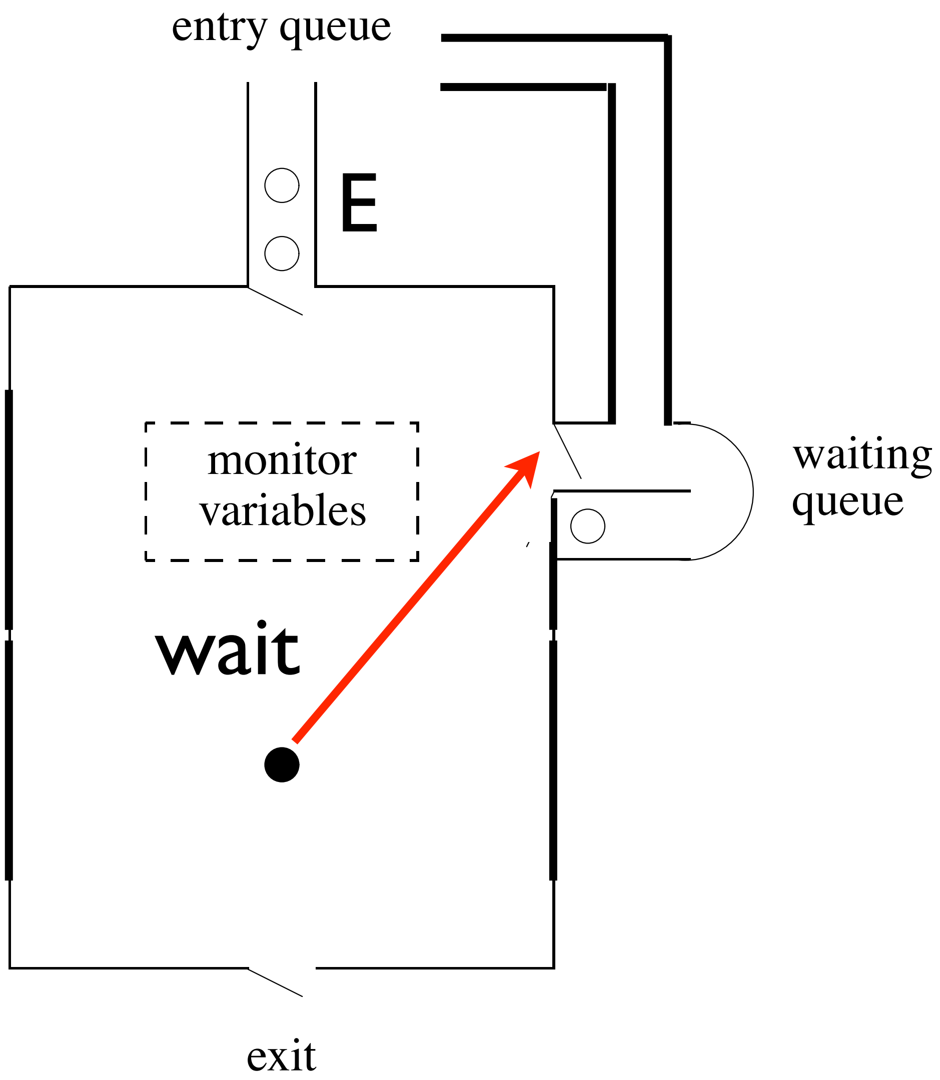


Can have waitC in the body of an if statement.
However, signalC should be the *last* statement of operation.

Java, Wait and Notify
E = W < S



Java, Wait and Notify
E = W < S



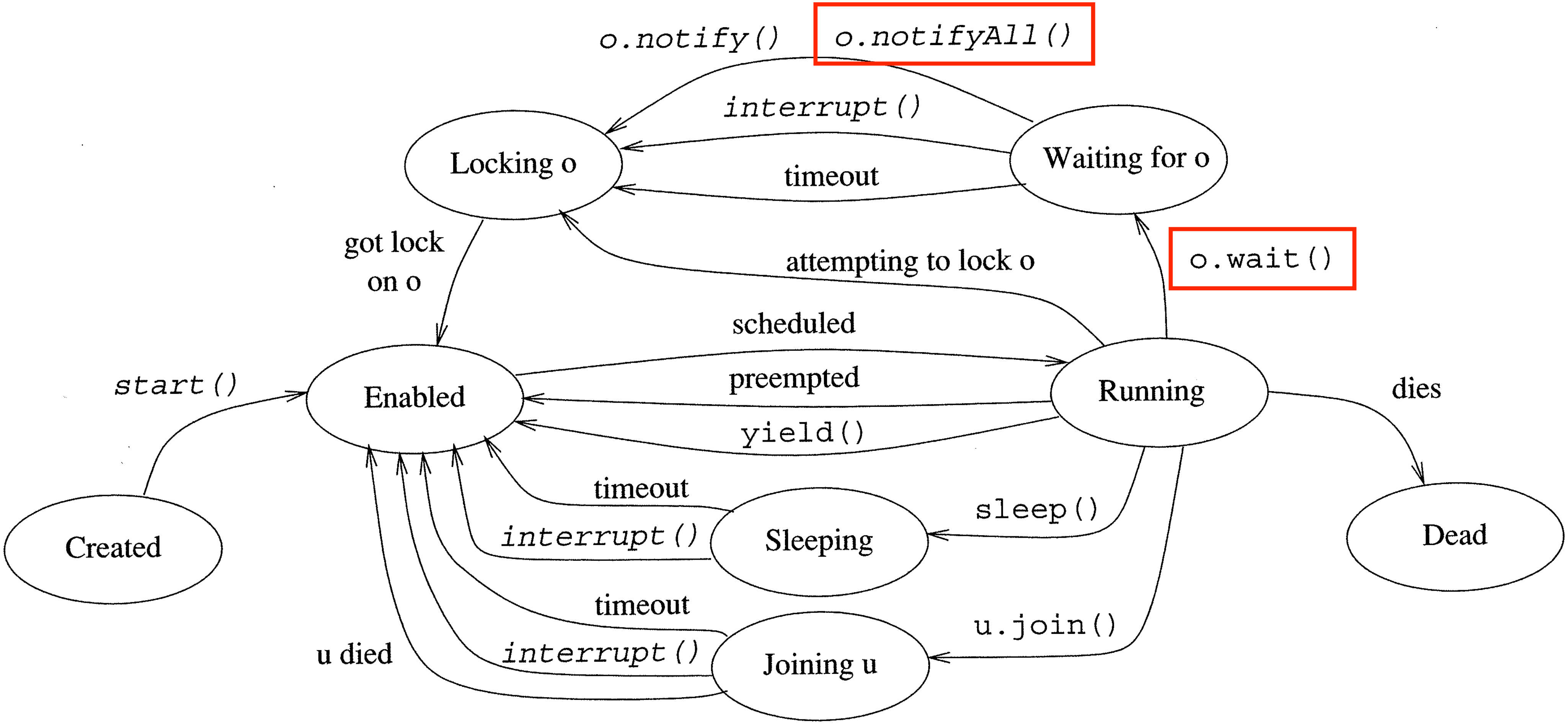
Signaler always picked. Signaled waits in entry queue.
There are no condition variables.

Java, Wait and Notify
$E = W < S$

* waitC is called wait in Java.
* signalC is called notify in Java.
* notifyAll moves all processes from the waiting queue to the entry queue.
* Signaler usually executes notifyAll, and waiting processes loop on their boolean expressions.

Sestoft, page 67

States and State Transitions of a Thread. A thread's transition from one state to another may be caused by a method call performed by the thread itself (shown in the `monospace` font), by a method call possibly performed by another thread (shown in the *`slanted monospace`* font); and by timeouts and other actions.



Java, Wait and Notify
E = W < S

E = W criticized by Buhr:

“In all cases, the no-priority property complicates the proof rules, makes performance worse, and makes programming more difficult. ... Therefore, we have rejected all no-priority monitors from further consideration.”

Semaphore / monitor equivalence

Semaphores and monitors have equivalent capabilities.

You can construct a semaphore with a monitor.

You can construct a monitor with a semaphore.

Algorithm 7.2: Semaphore simulated with a monitor

Hoare semantics

```
monitor Sem
    integer s $\leftarrow$ k
    condition notZero
    operation wait
        if $s = 0$
            waitC(notZero)
        $s \leftarrow s - 1$
    operation signal
        $s \leftarrow s + 1$
        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
```

Algorithm 7.2: Semaphore simulated with a monitor

Hoare semantics

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

Mesa semantics

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

p

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

q

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

Semaphore simulated with a monitor
C++ implementation of Algorithm 7.2

`signal()` uses `lock_guard` for mutual exclusion.

`wait()` uses `unique_lock` for mutual exclusion and the condition on which to wait.

`wait()` takes two parameters:
- A `unique_lock`
- A predicate that must be true to unblock the process

CoSc 450: Source Code

`Util450.cpp`

```
class Semaphore {
private:
    int s;
    condition_variable notZero;
    mutex semMutex;

public:

    Semaphore(int k) { s = k; }

    void wait() {
        unique_lock<mutex> guard(semMutex);
        notZero.wait(guard, [this]{return s != 0;});
        s--;
    }

    void signal() {
        lock_guard<mutex> guard(semMutex);
        s++;
        notZero.notify_one();
    }
};
```

Lambda expression passing function as a parameter.

Spurious wakeup — Problem

Mesa semantics: $E < W < S$, signaled unblocked, signaler continues.

There is no guarantee to the waiting process that the boolean expression it waited on is still true.

Another process may have changed the value of the expression between the signal execution and the resumption of the waiting.

Spurious wakeup — Solution

Signaled must first execute a loop on the condition to guarantee that the condition is met.

C++ `condition_variable wait()` method does the spurious wakeup loop automatically.

```
wait(unique_lock lock, Predicate pred)
```

is equivalent to

```
while (!pred()) { wait(lock); }
```

C++ lambda syntax

[captured variables] (parameters) { function code }

In class Semaphore: `[this]{return s != 0;}`
the captured variable `this` allows access to class attribute `s` in the function code.

Suppose you also have local variable `n` that you need to access in your function:

```
[this, n]{return s != n;}
```

C++ lambda syntax

[captured variables] (parameters)   {   function code   }

In class Semaphore: `[this]{return s != 0;}`
the function has no parameters, so you can omit the parentheses `()`.

Functional programming!

Scheme

```
(lambda (n)
  (* n n))

(define square
  (lambda (n)
    (* n n)))

> (square 5)
25
>
```

C++

```
function< int(int) > square;

square = [](int n) { return n * n; };

cout << square(5);

25
```

`lock_guard` vs `unique_lock`

Constructor for both lock the mutex.
Destructor for both unlock the mutex.

`unique_lock` is required for condition variables.

Programmer can lock and unlock a `unique_lock`.

```
guard.lock()
guard.unlock()
```

The producer-consumer problem with a finite buffer

Two condition variables: notEmpty and notFull

The producer calls append(D). Only the producer can be in the notFull queue of blocked processes.

The consumer calls take( ). Only the consumer can be in the notEmpty queue of blocked processes.

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

producer

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

consumer

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

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

Hoare semantics: $E < S < W$

```
monitor PC
    bufferType buffer ← empty
    condition notEmpty
    condition notFull
    ----------------------------------
    operation append(datatype V)
        if buffer is full
            waitC(notFull)          ←  waitc in the body of an if
        append(V, buffer)
        signalC(notEmpty)           ←  signalc the last statement of the operation
    ----------------------------------
    operation take()
        datatype W
        if buffer is empty
            waitC(notEmpty)         ←  waitc in the body of an if
        W ← head(buffer)
        signalC(notFull)            ←  signalc the last possible statement of the operation
        return W
```

Java implementation
of
the producer-consumer problem with a finite buffer

Java implementation has four classes/files:

* Algorithm0703.java for main program
* PCMonitor.java for the monitor
* Producer.java for the producer
* Consumer.java for the consumer

Algorithm0703

The main program:

* Allocates the monitor
* Allocates the consumer, passing it a pointer to the monitor, so the consumer can access the monitor
* Allocates the producer, passing it a pointer to the monitor, so the producer can access the monitor

CoSc 450: Source Code

Algorithm0703.java

```
class Algorithm0703 {

   public static void main(String[] args) {
      PCMonitor pc = new PCMonitor();
      Consumer consumer = new Consumer(pc);
      consumer.start();
      Producer producer = new Producer(pc);
      producer.start();
      try {
         consumer.join();
         producer.join();
      } catch (InterruptedException e) {
      }
   }
}
```

CoSc 450: Source Code

PCMonitor.java

Java semantics: E = S < W

```
final class PCMonitor {
   final int n = 5;
   int out = 0, in = 0;
   volatile int count = 0;
   final int[] buffer = new int[n];

   synchronized void append(int v) {
      while (count == n) {
         try {
            wait();
         } catch (InterruptedException e) {
         }
      }
      buffer[in] = v;
      in = (in + 1) % n;
      count = count + 1;
      System.out.println("Producer put " + v);
      notifyAll();
   }
```

← Waiting processes loop on their conditions

← Signaler executes notifyAll

CoSc 450: Source Code

PCMonitor.java

```
   synchronized int take() {
      int temp;
      while (count == 0) {
         try {
            wait();
         } catch (InterruptedException e) {
         }
      }
      temp = buffer[out];
      out = (out + 1) % n;
      count = count - 1;
      System.out.println("Consumer got " + temp);
      notifyAll();
      return temp;
   }
}
```

← Waiting processes loop on their conditions

← Signaler executes notifyAll

CoSc 450: Source Code

Producer.java

```
class Producer extends Thread {

   private final PCMonitor pc;

   Producer(PCMonitor pc) {
      this.pc = pc;
   }

   public void run() {
      int d;
      System.out.println("Producer started.");
      for (int i = 0; i < 15; i++) {
         try {
            randomDelay(60);
            d = 10 * i;
            pc.append(d);
         } catch (InterruptedException e) {
         }
      }
      System.out.println("Producer finished.");
   }
}
```

CoSc 450: Source Code

`Consumer.java`

```
class Consumer extends Thread {

   private final PCMonitor pc;

   Consumer(PCMonitor pc) {
      this.pc = pc;
   }

   public void run() {
      int d;
      System.out.println("Consumer started.");
      for (int i = 0; i < 15; i++) {
         try {
            randomDelay(100);
            d = pc.take(); // Ignore returned value
         } catch (InterruptedException e) {
         }
      }
      System.out.println("Consumer finished.");
   }
}
```

C++ implementation
of
the producer-consumer problem with a finite buffer

CoSc 450: Source Code

Algorithm-7-3.cpp

Mesa semantics: E < W < S

```
class PCMonitor {
private:
    static const int n = 5;
    int out = 0, in = 0;
    volatile int count = 0;
    int buffer[n];
    mutex pcMutex;
    condition_variable notEmpty;
    condition_variable notFull;

public:
    void append(int v) {
        unique_lock<mutex> guard(pcMutex);
        notFull.wait(guard, [this]{return count != n;});
        buffer[in] = v;
        in = (in + 1) % n;
        count = count + 1;
        cout << "Producer put " << v << endl;
        notEmpty.notify_one();
    }
```

Automatic spurious wakeup loop

Signaler executes notify_one

CoSc 450: Source Code

`Algorithm-7-3.cpp`

```
    int take() {
        unique_lock<mutex> guard(pcMutex);
        notEmpty.wait(guard, [this]{return count != 0;});
        int temp = buffer[out];
        out = (out + 1) % n;
        count = count - 1;
        cout << "Consumer got " << temp << endl;
        notFull.notify_one();
        return temp;
    }
};
```

Automatic spurious wakeup loop

Signaler executes notify_one

CoSc 450: Source Code

`Algorithm-7-3.cpp`

```
PCMonitor pc;

void producerRun() {
    int d;
    cout << "Producer started." << endl;
    for (int i = 0; i < 15; i++) {
        randomDelay(60);
        d = 10 * i;
        pc.append(d);
    }
    cout << "Producer finished." << endl;
}

void consumerRun() {
    int d;
    cout << "Consumer started." << endl;
    for (int i = 0; i < 15; i++) {
        randomDelay(100);
        d = pc.take(); // Ignore returned value
    }
    cout << "Consumer finished." << endl;
}
```

CoSc 450: Source Code

`Algorithm-7-3.cpp`

```
int main() {
    thread consumer(consumerRun);
    thread producer(producerRun);
    consumer.join();
    producer.join();
    return EXIT_SUCCESS;
}
```

The dining philosopher’s problem

* fork[i] is how many forks are available to philosopher[i]. Initialized to 2 because two forks are initially available.

The dining philosopher’s problem

* fork[i] is how many forks are available to philosopher[i]. Initialized to 2 because two forks are initially available.

* Before eating, decrement number of forks available to neighbor by 1 each. No interleaving.

Algorithm 7.5: Dining philosophers with a monitor

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

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

Algorithm 7.5: Dining philosophers with a monitor (continued)

philosopher i

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

The dining philosopher’s problem
Algorithm 7.5

- This solution has mutual exclusion and is deadlock-free but can starve.

Scenario for starvation of Philosopher 2

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

The readers and writers problem

* There is a shared database with many readers and writers.

* There can be many readers at one time.

* But there can only be one writer.

* Following solution is starvation-free.

Algorithm 7.4: Readers and writers with a monitor

Hoare semantics: E < S < W

```
monitor RW
    integer readers $\leftarrow 0$
    integer writers $\leftarrow 0$
    condition OKtoRead, OKtoWrite
    operation StartRead
        if $writers \neq 0$ or not empty(OKtoWrite)
            waitC(OKtoRead)              ← waitc in the body of an if
        $readers \leftarrow readers + 1$
        signalC(OKtoRead)                ← signalc the last statement of the operation
    operation EndRead
        $readers \leftarrow readers - 1$
        if $readers = 0$
            signalC(OKtoWrite)           ← signalc the last statement of the operation
```

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

Hoare semantics: E < S < W

```
operation StartWrite
    if writers $\neq$ 0 or readers $\neq$ 0
        waitC(OKtoWrite)                      waitc in the body of an if
    writers $\leftarrow$ writers + 1
```

```
operation EndWrite
    writers $\leftarrow$ writers − 1
    if empty(OKtoRead)
        then signalC(OKtoWrite)               signalc the last statement of the operation
        else signalC(OKtoRead)                signalc the last statement of the operation
```

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

Readers and Writers

* New writers are blocked if any readers are active or if a writer is active.

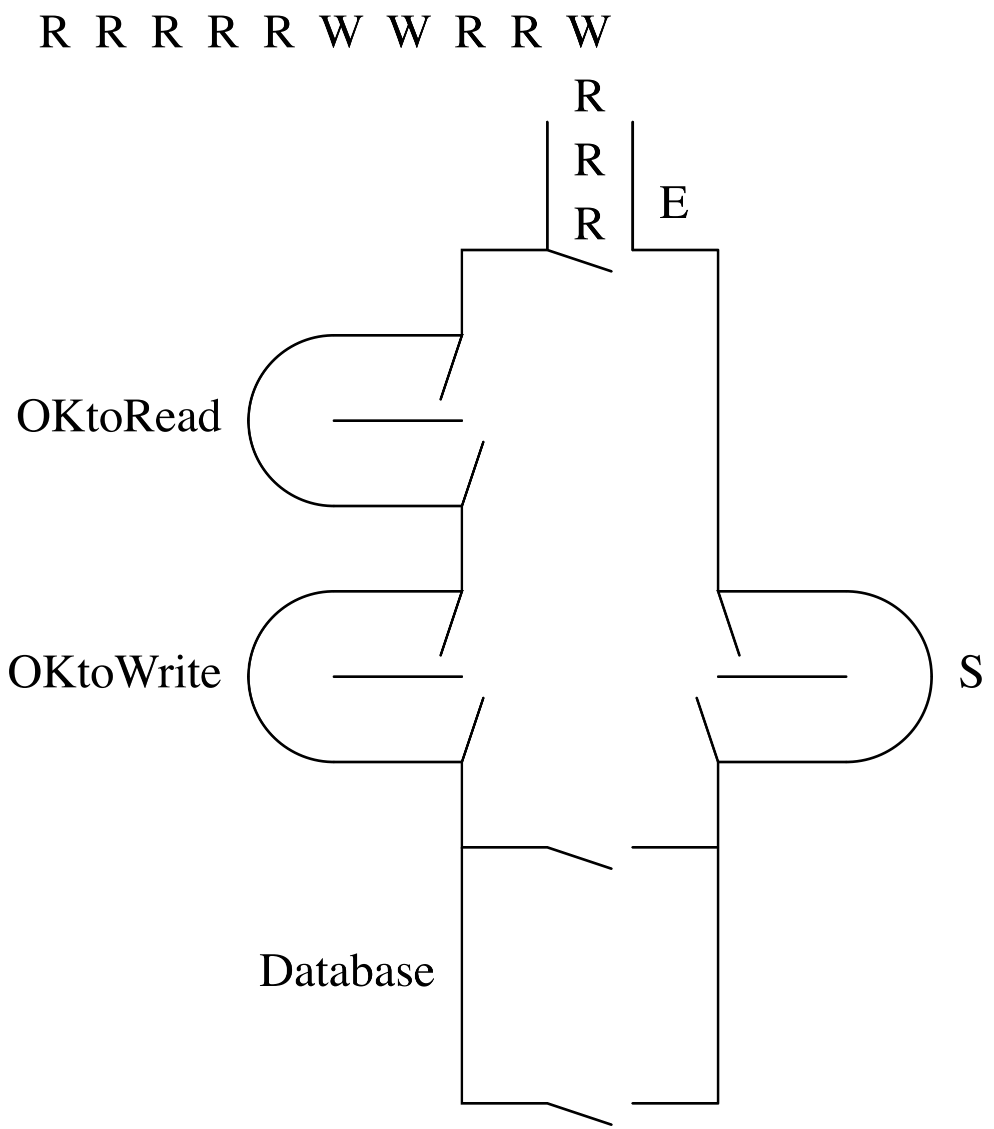
* An exiting writer unblocks any blocked reader rather than a blocked writer. Consequently, all blocked readers enter.

* Any subsequent incoming readers will be blocked by the blocked writer, who will eventually enter when the last active reader exits.

CoSc 450: Programming Paradigms

Algorithm 7.4

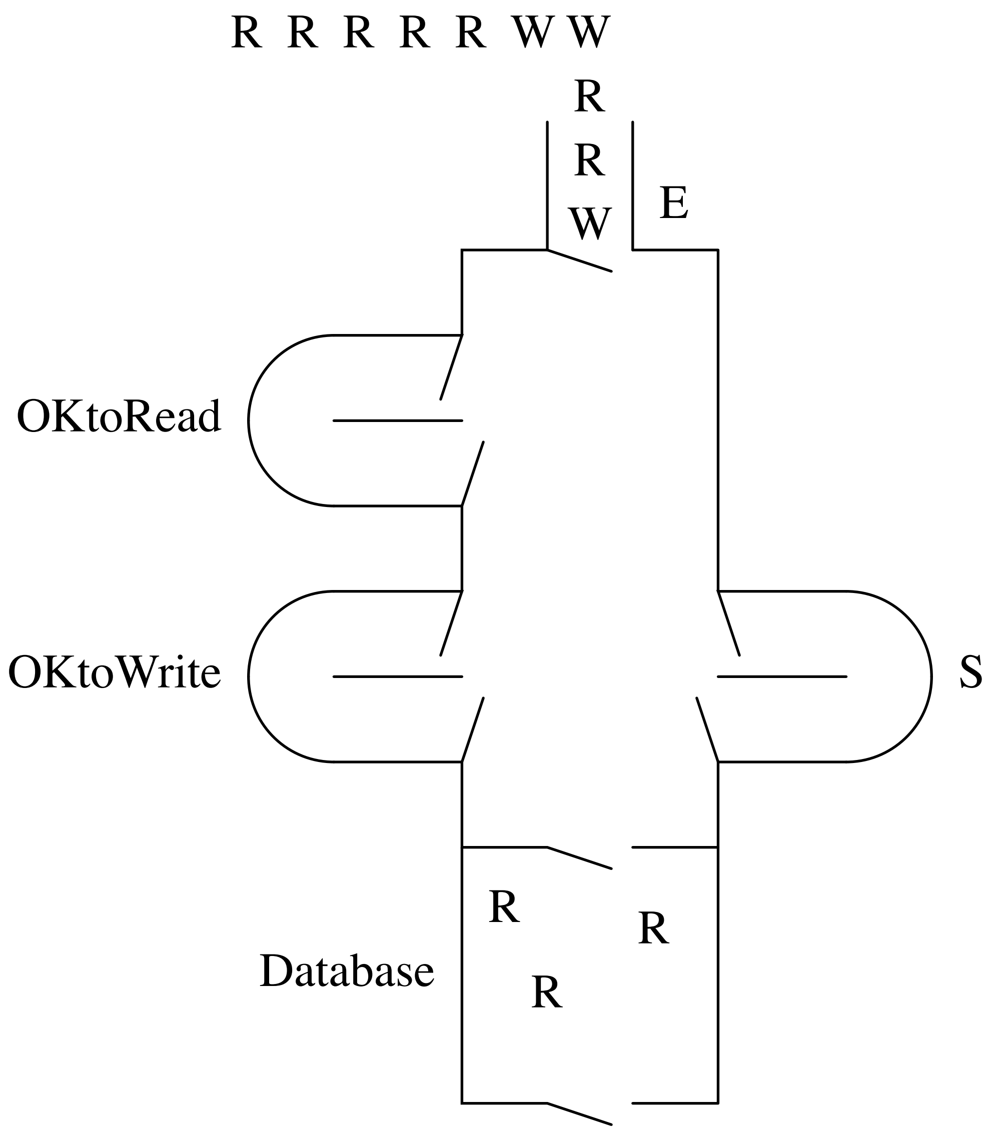
Hoare semantics: E < S < W



Initial state

CoSc 450: Programming Paradigms

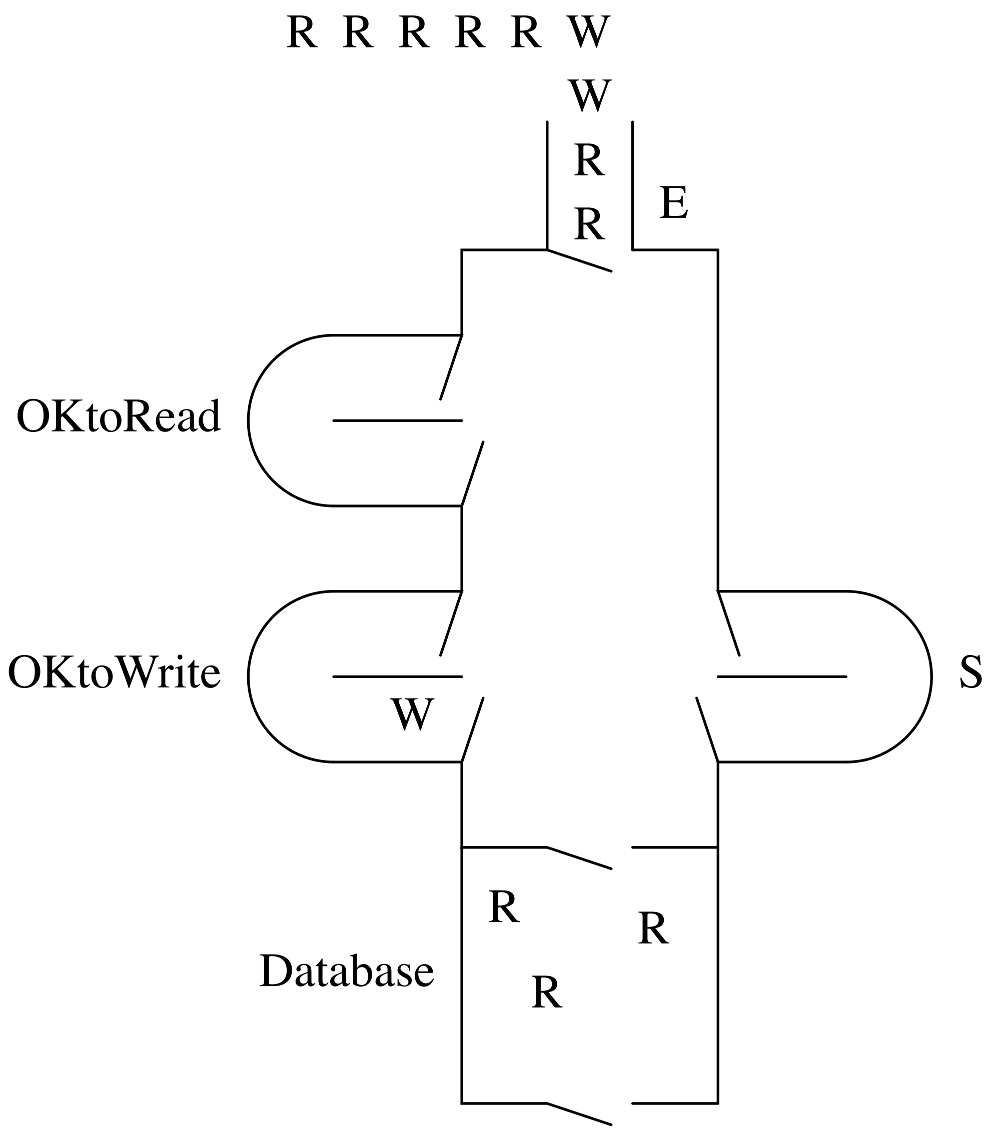
Hoare semantics: E < S < W



StartRead
Three readers enter because
writers = 0 and OKtoWrite is empty.

CoSc 450: Programming Paradigms

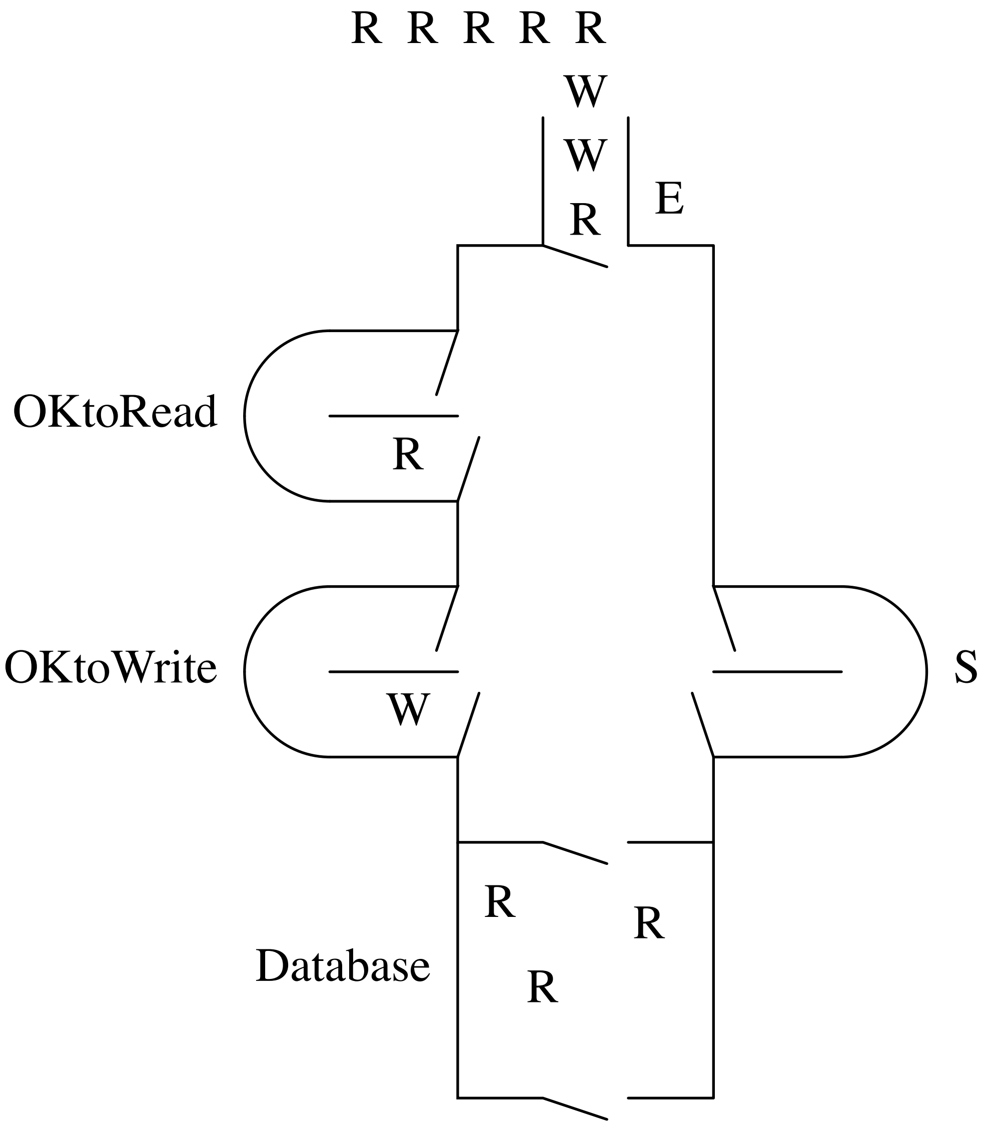
`Algorithm 7.4`



Hoare semantics: E < S < W

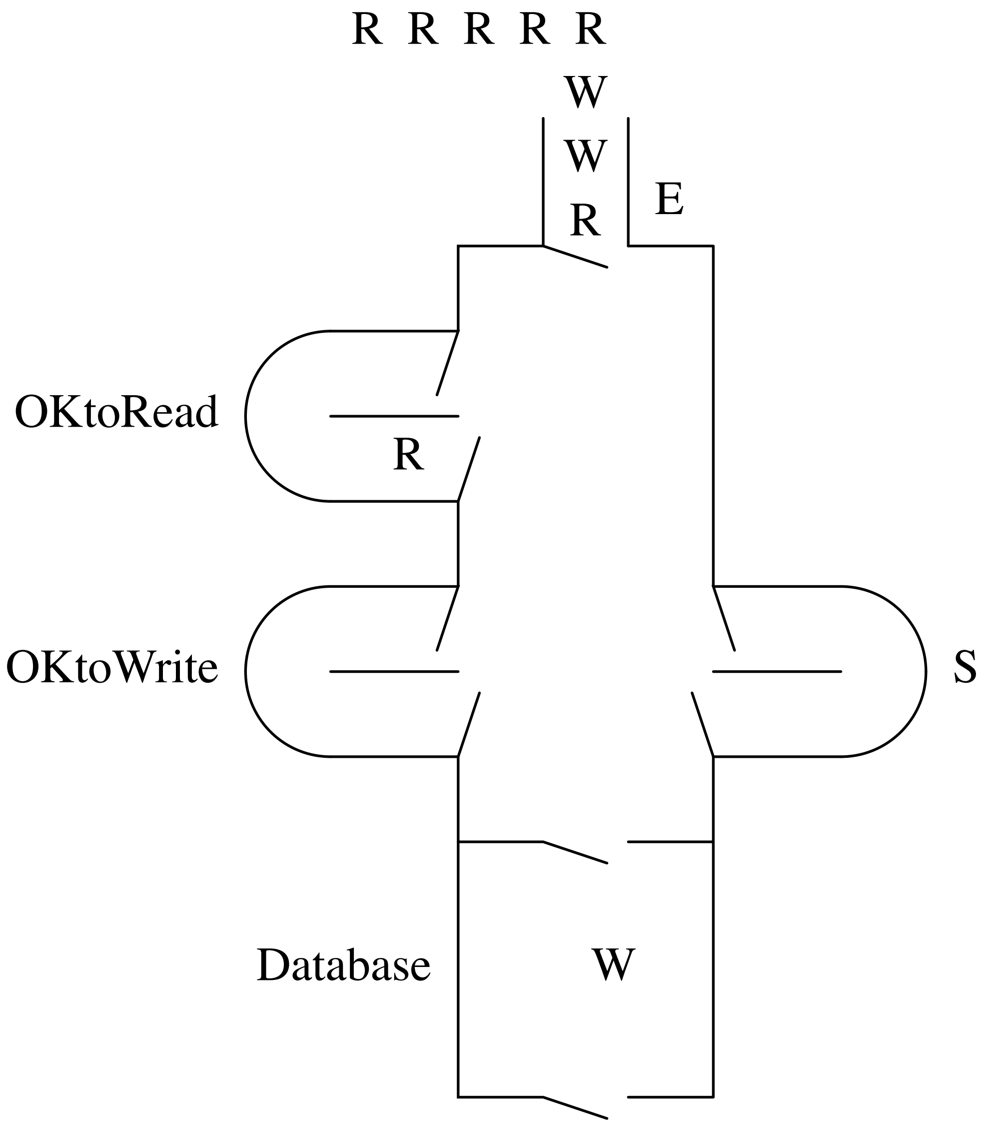
StartWrite
Writer is blocked on OKtoWrite
because readers = 3

Hoare semantics: E < S < W



StartRead
Reader is blocked on OKtoRead
because OKtoWrite is not empty

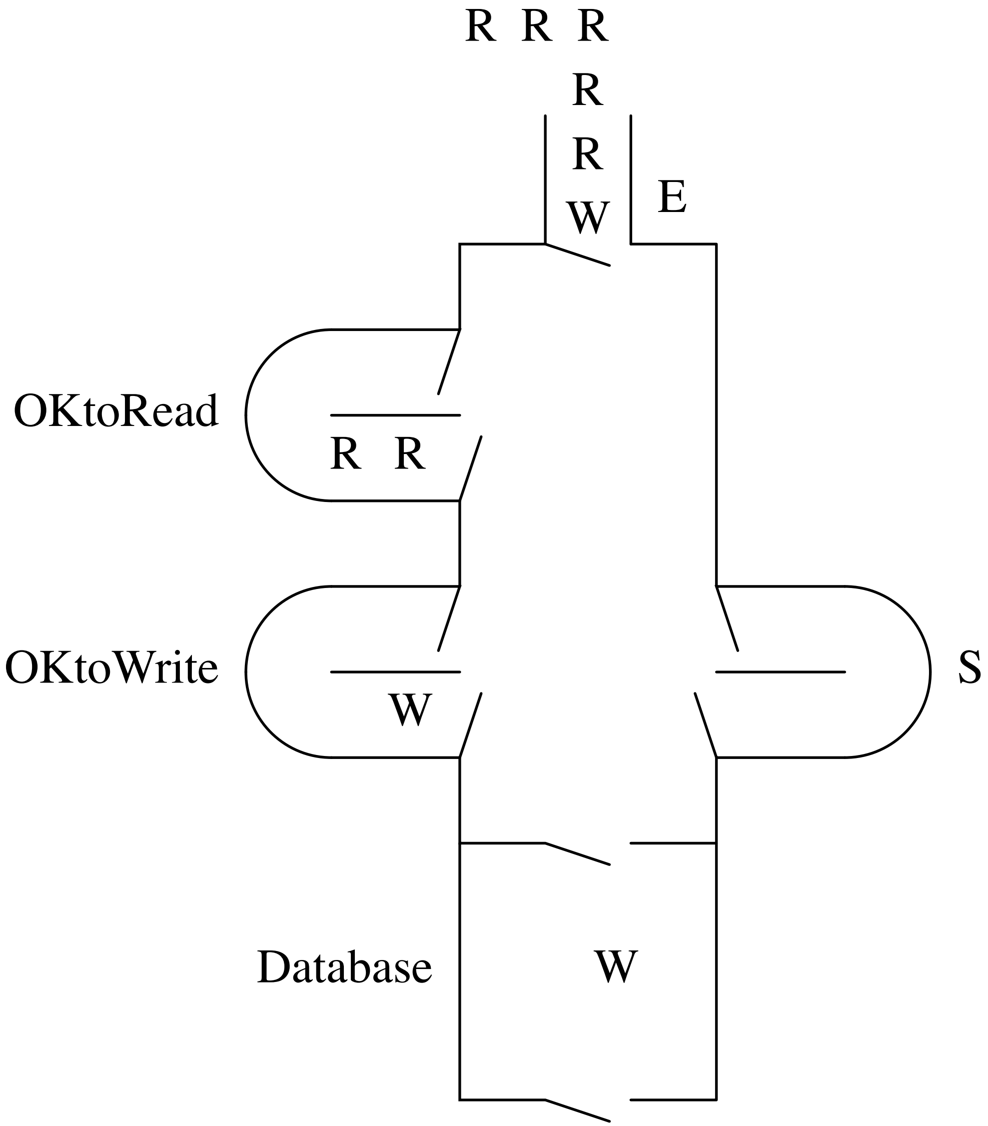
Hoare semantics: E < S < W



EndRead
The first reader to exit does not signal.
The second reader to exit does not signal.
The third reader to exit signals OKtoWrite because readers = 0.

The writer is unblocked and writers = 1.

Algorithm 7.4



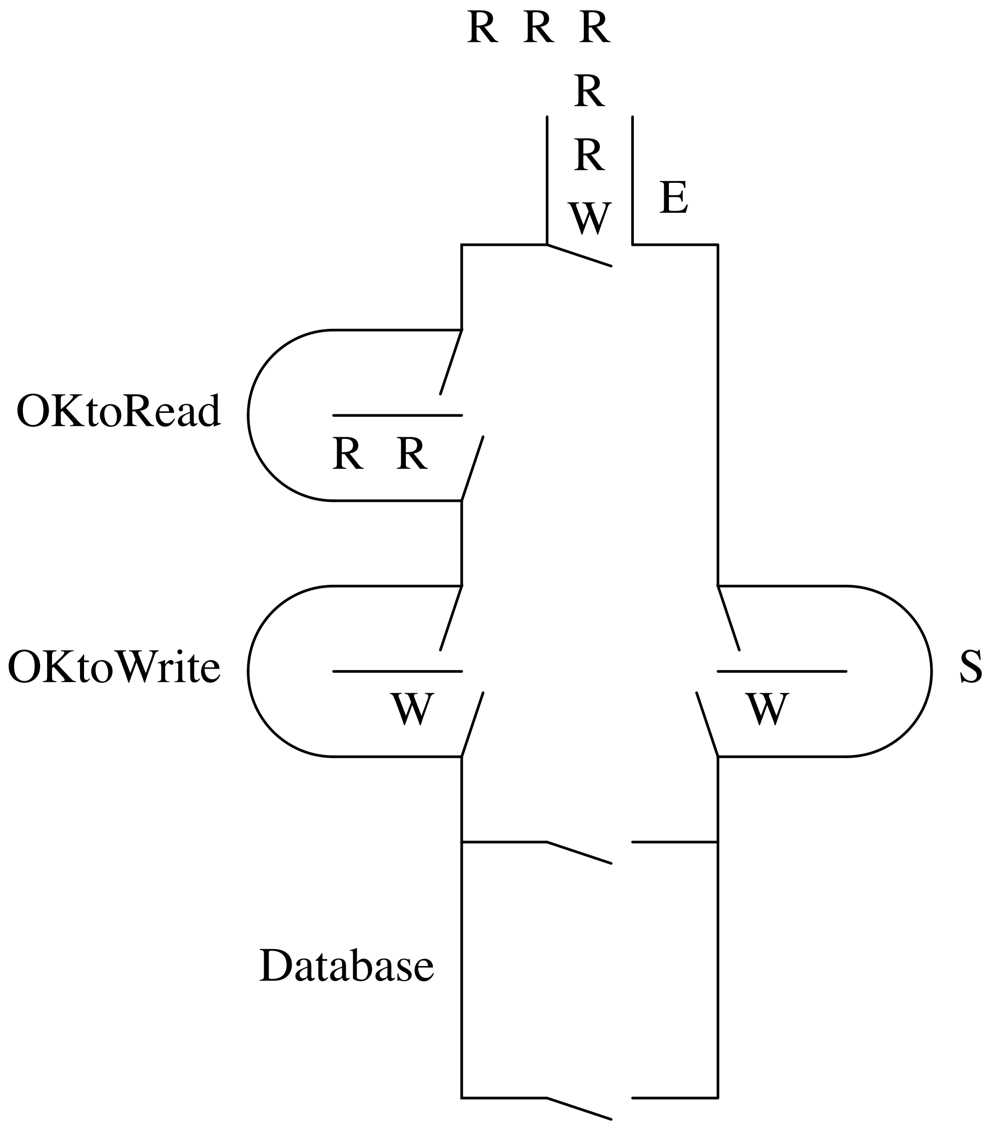
Hoare semantics: E < S < W

StartRead
The next reader to enter is blocked on OKtoRead because writers = 1.

StartWrite
The next writer to enter is blocked on OKtoWrite because writers = 1.

Algorithm 7.4

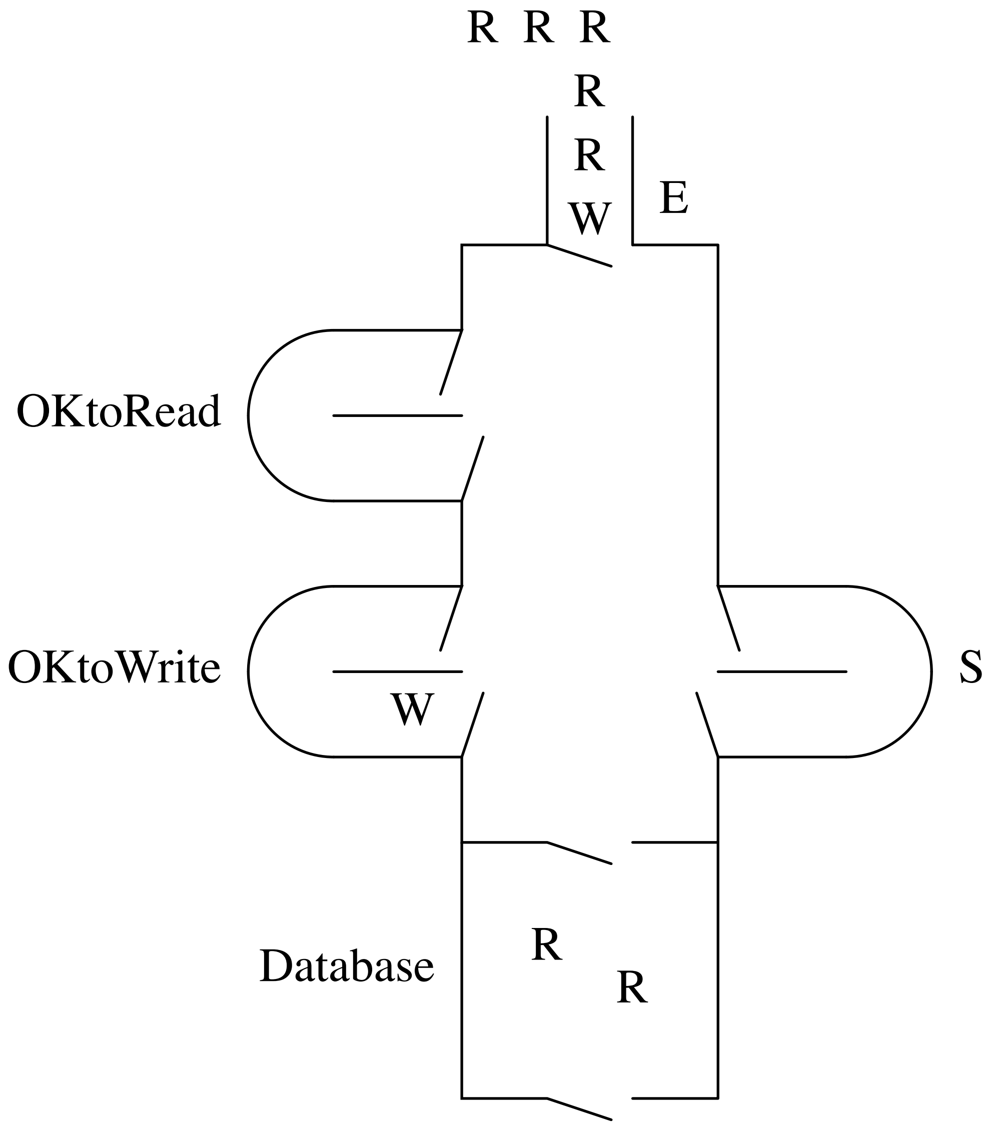
Hoare semantics: E < S < W



EndWrite
writers = 0.
The exiting writer signals OKtoRead because OKtoRead is not empty.

The exiting writer may be blocked on S, but that is inconsequential because signalc is the last statement of the operation. It will immediately exit because E < S.

Hoare semantics: E < S < W



EndWrite, continued
The first signaled reader resumes.
readers = 1.
It signals the next reader, which immediately resumes by Hoare semantics. If there were a waiting queue with Mesa semantics, the next reader would be blocked.

So, the next reader resumes and readers = 2.

C++ implementation
of the readers and writers problem

Ben-Ari's solution depends on Hoare semantics.

Problem: C++ condition variables use Mesa semantics.
Algorithm 7.4 will deadlock with Mesa semantics.

Solution: Use `shared_mutex` type and `shared_lock` type with C++17.

`shared_mutex` and `shared_lock`

Two levels of access:

- Shared - Several threads can share ownership of the same mutex.
- Exclusive - Only one thread can own the mutex.

If one thread has acquired the exclusive lock, no other threads can acquire the lock. The shared lock can be acquired by multiple threads (readers) only when the exclusive lock has not been acquired by any thread (a writer).

C++ implementation
of the readers and writers problem

`ReadersWritersA`

C++ has built in the Terekhov algorithm for shared and exclusive use of the `shared_mutex` as a solution for the readers and writers problem.

CoSc 450: Source Code

ReadersWritersA.cpp

```
class RWDataBase {
private:
    int myData = 5;
    shared_mutex rwMutex;
    mutex coutMutex;

public:
    void readMyData(int readerID) {
        shared_lock<shared_mutex> guard(rwMutex);
        coutMutex.lock();
        cout << "Reader " << readerID << " is about to read" << endl;
        coutMutex.unlock();
        randomDelay(60);
        coutMutex.lock();
        cout << "Reader " << readerID << " read " << myData << endl;
        coutMutex.unlock();
    }

    void writeMyData(int writerID) {
        lock_guard<shared_mutex> guard(rwMutex);
        cout << "Writer " << writerID << " is about to write" << endl;
        randomDelay(60);
        myData += 5;
        cout << "Writer " << writerID << " wrote " << myData << endl;
    }
};
```

← Shared access

← Exclusive access

CoSc 450: Source Code

`ReadersWritersA.cpp`

```
RWDataBase rwDataBase;

void readerRun(int readerID) {
    for (int i = 0; i < 3; i++) {
        randomDelay(60);
        rwDataBase.readMyData(readerID);
    }
}

void writerRun(int writerID) {
    for (int i = 0; i < 3; i++) {
        randomDelay(60);
        rwDataBase.writeMyData(writerID);
    }
}
```

CoSc 450: Source Code

ReadersWritersA.cpp

```
int main() {
    thread reader0(readerRun, 0);
    thread reader1(readerRun, 1);
    thread reader2(readerRun, 2);
    thread writer0(writerRun, 0);
    thread writer1(writerRun, 1);
    reader0.join();
    reader1.join();
    reader2.join();
    writer0.join();
    writer1.join();
    return EXIT_SUCCESS;
}
```

C++ implementations
of the readers and writers problem

`ReadersWritersA` – With `shared_mutex`
`ReadersWritersB` – The Terekhov algorithm without `shared_mutex`
`ReadersWritersC` – The optimized Terekhov algorithm
`ReadersWritersD` – Algorithm 7.4, which deadlocks

`ReadersWritersA`

When a lock is allocated on the run-time stack, its constructor locks the mutex.
The lock operation is not visible in the code.

When a lock is deallocated on function termination, its destructor unlocks the mutex.
The unlock operation is not visible in the code.

That is why the code is simple to write.

`ReadersWritersB`

`ReadersWritersB` shows the lock( ) and unlock( ) operations of a `shared_mutex` by programming them explicitly without using a `shared_mutex`.

StartRead corresponds to `lock_shared()`.
EndRead corresponds to `unlock_shared()`.
StartWrite corresponds to `lock()`.
EndWrite corresponds to `unlock()`.

The Terekhov algorithm

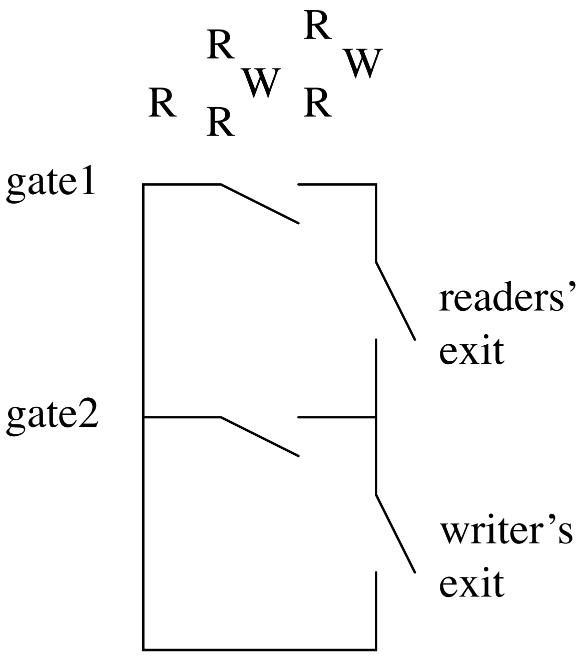
Two condition variables, `gate1` and `gate2`.

One `int readers` for the number of readers inside `gate1`.

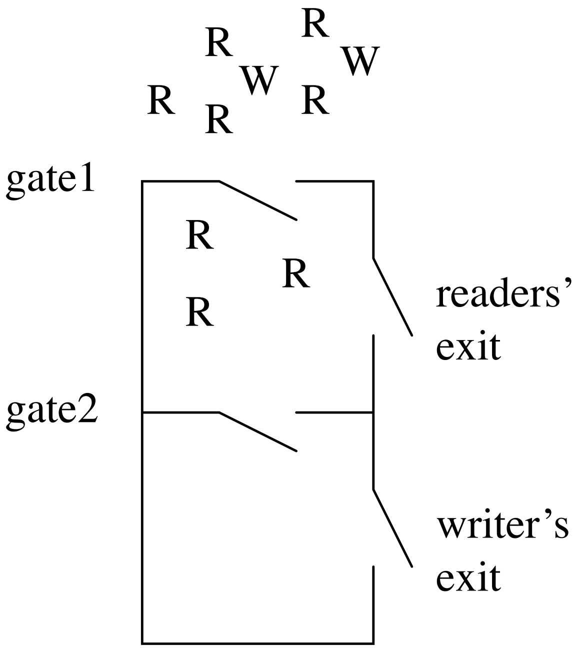
One `bool writer` if a writer is inside `gate1` or `gate2`.

There are four rules: (next slide)

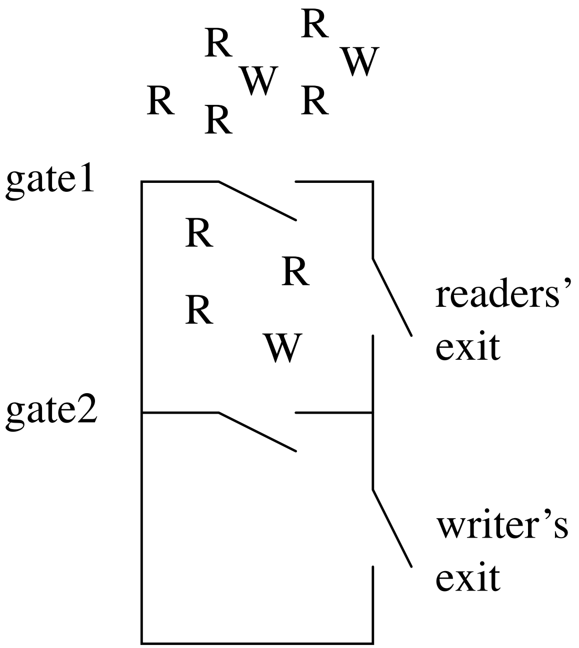
- When a reader enters `gate1`, it has read access. However, a writer must enter first `gate1` and then `gate2` to have write access.
- There can be any number of readers and at most one writer inside `gate1`. There cannot be any readers inside `gate2`.
- No one can enter `gate1` if a writer is inside `gate1` or `gate2`. If a reader or writer tries to enter it is blocked on `gate1`.
- A writer can only enter `gate2` when the number of readers inside `gate1` drops to 0. If it tries to enter `gate2` when there are readers inside `gate1` it is blocked on `gate2`.



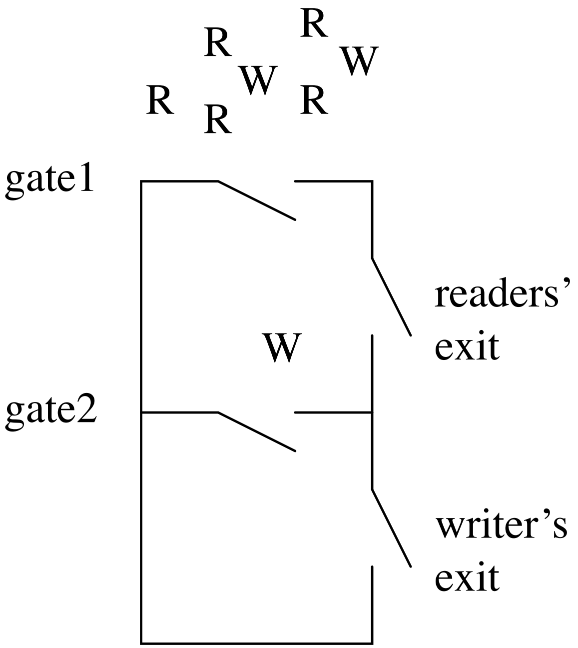
Initial state



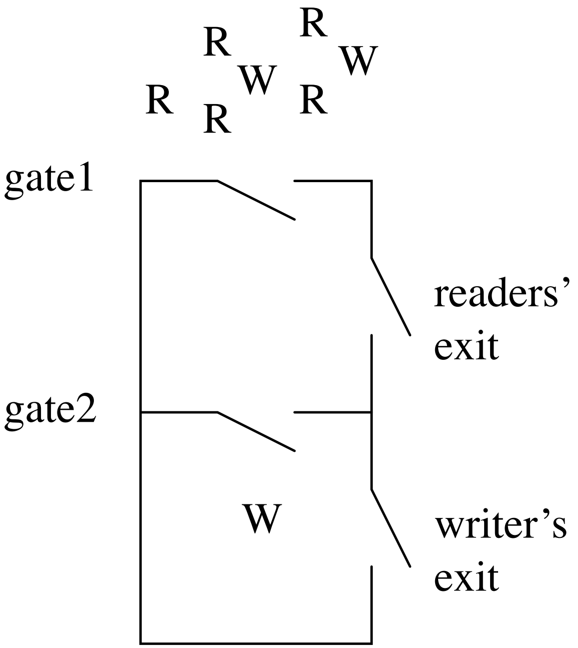
Some readers enter gate1 and exit.



A writer enters gate1.
Readers and writers are blocked on gate1.



The number of readers inside gate1 drops to 0.



The writer enters gate2.
Readers and writers are still blocked on gate1.
When the writer exits, the system returns to its initial state.

CoSc 450: Source Code

ReadersWritersB
.cpp

The Terekhov Algorithm
Mesa semantics: E < W < S

```
class RWMonitor {
private:
    mutex rwMutex;
    condition_variable gate1;
    condition_variable gate2;
    int readers = 0;
    bool writer = false;

public:
    void startRead() {
        unique_lock<mutex> guard(rwMutex);
        gate1.wait(guard, [this] { return !writer; });
        readers++;
    }

    void endRead() {
        unique_lock<mutex> guard(rwMutex);
        readers--;
        if (writer && (readers == 0)) {
            gate2.notify_one();
        }
    }
```

CoSc 450: Source Code

ReadersWritersB
.cpp

The Terekhov Algorithm
Mesa semantics: E < W < S

```
    void startWrite() {
        unique_lock<mutex> guard(rwMutex);
        gate1.wait(guard, [this] { return !writer; });
        writer = true;
        gate2.wait(guard, [this] { return readers == 0; });
    }

    void endWrite() {
        unique_lock<mutex> guard(rwMutex);
        readers = 0;
        writer = false;
        gate1.notify_all();
    }
};
```

CoSc 450: Source Code

ReadersWritersB
.cpp

```
class RWDataBase {
private:
    RWMonitor rwMonitor;
    int myData = 5;
    mutex coutMutex;
public:
    void readMyData(int readerID) {
        rwMonitor.startRead();
        coutMutex.lock();
        cout << "Reader " << readerID << " is about to read" << endl;
        coutMutex.unlock();
        randomDelay(60);
        coutMutex.lock();
        cout << "Reader " << readerID << " read " << myData << endl;
        coutMutex.unlock();
        rwMonitor.endRead();
    }

    void writeMyData(int writerID) {
        rwMonitor.startWrite();
        cout << "Writer " << writerID << " is about to write" << endl;
        randomDelay(60);
        myData += 5;
        cout << "Writer " << writerID << " wrote " << myData << endl;
        rwMonitor.endWrite();
    }
};
```

`ReadersWritersC`

`ReadersWritersC` is an optimized version of `ReadersWritersB`. It is the reference implementation for `shared_mutex` in C++17.

In place of `int readers` and `bool writer` is a single unsigned integer named `state`. The first bit of `state` is 1 if writer is true and 0 otherwise. The remaining bits are the count of readers. 8-bit example:

`state`: 0000 0110, 6 readers and no writer
`state`: 1000 0111, 7 readers and 1 writer

Masks for accessing readers and writer

`readerMask`: First bit 1, remaining bits 0.
`writerMask`: First bit 0, remaining bits 1.

Expression	Meaning
`state & writerMask`	True iff a writer is inside gate1 or gate2
`state & readerMask`	Number of readers inside gate1
`(state & readerMask)` `== readerMask`	True iff the number of readers inside gate1 is the maximum we can count
`readers == readerMask - 1`	True iff the number of readers inside gate1 is one less than the maximum we can count

`unsigned readers =`
`(state & readerMask) + 1;`
`state &= writerMask;`
`state |= readers;`
	Adds 1 to number of readers

`state |= writerMask;`	Sets `state` to specify that a writer is inside

The optimized code also programs the spurious wakeup loop explicitly without the predicate parameter in the `wait()` function.

```
gate1.wait(guard, [this] { return !writer; });
```

is coded as

```
while (state & writerMask)
    gate1.wait(guard);
```

CoSc 450: Source Code

ReadersWritersC
.cpp

```
class RWMonitor {
private:
    mutex rwMutex;
    condition_variable gate1;
    condition_variable gate2;
    unsigned state = 0;
    static const unsigned writerMask = 1U << (sizeof(unsigned) * CHAR_BIT - 1);
    static const unsigned readerMask = ~writerMask;

public:
    void startRead() {
        unique_lock<mutex> guard(rwMutex);
        while ((state & writerMask) || (state & readerMask) == readerMask)
            gate1.wait(guard);
        unsigned readers = (state & readerMask) + 1;
        state &= writerMask;
        state |= readers;
    }
```

CoSc 450: Source Code

ReadersWritersC
.cpp

```
void endRead() {
    unique_lock<mutex> guard(rwMutex);
    unsigned readers = (state & readerMask) - 1;
    state &= writerMask;
    state |= readers;
    if (state & writerMask) {
        if (readers == 0)
            gate2.notify_one();
    } else {
        if (readers == readerMask - 1)
            gate1.notify_one();
    }
}
```

CoSc 450: Source Code

ReadersWritersC.cpp

```
    void startWrite() {
        unique_lock<mutex> guard(rwMutex);
        while (state & writerMask)
            gate1.wait(guard);
        state |= writerMask;
        while (state & readerMask)
            gate2.wait(guard);
    }

    void endWrite() {
        unique_lock<mutex> guard(rwMutex);
        state = 0;
        gate1.notify_all();
    }
};
```

`ReadersWritersD`

`ReadersWritersD` is Algorithm 7.4, which assumes Hoare semantics. C++17 uses Mesa semantics. Allgorithm 7.4 deadlocks with Mesa semantics.

There is no `empty()` method in C++17 for checking the status of the condition variable queue. This implementation maintains a count of blocked processes for that purpose.

CoSc 450: Source Code

ReadersWritersD
.cpp

```
class RWMonitor {
private:
    mutex rwMutex;
    condition_variable okToRead;
    condition_variable okToWrite;
    int readers = 0;
    int writers = 0;
    int blockedOnOKtoRead = 0;
    int blockedOnOKtoWrite = 0;
```

CoSc 450: Source Code

ReadersWritersD
.cpp

```
public:
    void startRead() {
        unique_lock<mutex> guard(rwMutex);
        if (writers == 0 || blockedOnOKtoWrite != 0) {
            blockedOnOKtoRead++;
            okToRead.wait(guard,
                [this] { return writers != 0 && blockedOnOKtoWrite == 0; });
            blockedOnOKtoRead--;
        }
        readers++;
        okToRead.notify_one();
    }

    void endRead() {
        unique_lock<mutex> guard(rwMutex);
        readers--;
        if (readers == 0) {
            okToWrite.notify_one();
        }
    }
```

CoSc 450: Source Code

ReadersWritersD.cpp

```
    void startWrite() {
        unique_lock<mutex> guard(rwMutex);
        if (blockedOnOKtoRead == 0) {
            blockedOnOKtoWrite++;
            okToWrite.wait(guard,
                [this] { return writers == 0 && readers == 0; });
            blockedOnOKtoWrite--;
        }
        writers++;
    }

    void endWrite() {
        unique_lock<mutex> guard(rwMutex);
        if (blockedOnOKtoRead == 0) {
            okToWrite.notify_one();
        } else {
            okToRead.notify_one();
        }
    }
};
```