PREV
6. Semaphores

NEXT
8. Channels

Chapter 7. Monitors

Introduction

The semaphore was introduced to provide a synchronization primitive that does not require busy waiting. Using semaphores we have given solutions to common concurrent programming problems. However, the semaphore is a low-level primitive because it is unstructured. If we were to build a large system using semaphores alone, the responsibility for the correct use of the semaphores would be diffused among all the implementers of the system. If one of them forgets to call `signal(S)` after a critical section, the program can deadlock and the cause of the failure will be difficult to isolate.

Monitors provide a structured concurrent programming primitive that concentrates the responsibility for correctness into modules. Monitors are a generalization of the *kernel* or *supervisor* found in operating systems, where critical sections such as the allocation of memory are centralized in a privileged program. Applications programs request services which are performed by the kernel. Kernels are run in a hardware mode that ensures that they cannot be interfered with by applications programs.

The monitors discussed in this chapter are decentralized versions of the monolithic kernel. Rather than have one system program handle all requests for services involving shared devices or data, we define a separate monitor for each object or related group of objects that requires synchronization. If operations of the same monitor are called by more than one process, the implementation ensures that these are executed under mutual exclusion. If operations of different monitors are called, their executions can be interleaved.

class can be allocated, and the operations of the class invoked on the fields of the object. The monitor adds the requirement that only one process can execute an operation on an object at any one time. Furthermore, while the fields of an object may be declared either public (directly accessible outside the class) or private (accessible only by operations declared within the class), the fields of a monitor are all private. Together with the requirement that only one process at a time can execute an operation, this ensures that the fields of a monitor are accessed consistently.

Actual implementations of monitors in programming languages and systems are quite different from one another. We begin the chapter by describing the classical monitor, a version of which is implemented in the BACI concurrency simulator. Then we will discuss protected objects of Ada, followed by synchronized methods in Java which can be used to implement monitors. It is extremely important that you learn how to analyze different implementations: their advantages, their disadvantages and the programming paradigms appropriate to each one.

Declaring and Using Monitors

In Section 2.5, we discussed the importance of atomic statements. Even the trivial Algorithm 2.4 showed that interleaving a pair of statements executed by each of two processes could lead to unexpected scenarios. The following algorithm shows the same two statements encapsulated within a monitor:

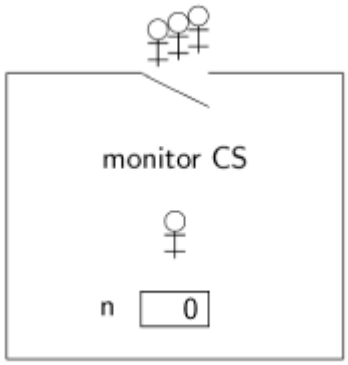
Table 7.1. Atomicity of monitor operations

```
monitor CS
  integer n ← 0

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

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

The monitor `CS` contains one variable `n` and one operation `increment`; two statements are contained within this operation, together with the declaration of the local variable. The variable `n` is not accessible outside the monitor. Two processes, `p` and `q`, each call the monitor operation `CS.increment`. Since by definition only one process at a time can execute a monitor operation, we are ensured mutual exclusion in access to the variable, so the only possible result of executing this algorithm is that `n` receives the value 2. The following diagram shows how one process is executing statements of a monitor operation while other processes wait outside:



This algorithm also solves the critical section problem, as can be seen by substituting an arbitrary `critical section` statement for the assignment statements. Compare this solution with the solution to the critical section problem using semaphores given in Section 6.3. The statements of the critical section are encapsulated in the monitor rather than replicated in each process. The synchronization is implicit and does not require the programmers to correctly place `wait` and `signal` statements.

The monitor is a static entity, not a dynamic process. It is just a set of operations that "sit there" waiting for a process to invoke one of them. There is an implicit lock on the "door" to the monitor, ensuring only one process is "inside" the monitor at any time. A process must open the lock to enter the monitor; the lock is then closed and remains closed until the process leaves the monitor.

As with semaphores, if there are several processes attempting to enter a monitor, only one of them will succeed. *There is no explicit queue associated with the monitor entry, so starvation is possible.*

In our examples, we will declare single monitors, but in real programming languages, a monitor would be declared as a type or a class and you can allocate as many objects of the type as you need.

Condition Variables

A monitor implicitly enforces mutual exclusion to its variables, but many problems in concurrent programming have explicit synchronization requirements. For example, in the solution of the producer–consumer problem with a bounded buffer (Section 6.7), the producer must be blocked if the buffer is full and the consumer must be blocked if the buffer is empty. There are two approaches to providing synchronization in monitors. In one approach, the required condition is named by an explicit *condition variable* (sometimes called an *event*). Ordinary boolean expressions are used to test the condition, blocking on the condition variable if necessary; a separate statement is used to unblock a process when the condition becomes true. The alternate approach is to block directly on the expression and let the implementation implicitly unblock a process when the expression is true. The classical monitor uses the first approach, while the second approach is used by protected objects (Section 7.10). Condition variables are one of the synchronization constructs in pthreads, although they are not part of an encapsulating structure like monitors.

Simulating Semaphores

Let us see how condition variables work by simulating a solution to the critical section problem using semaphores:

Table 7.2. Semaphore simulated with a monitor

```
monitor Sem
  integer s ← k
  condition notZero

  operation wait
    if s = 0
       waitC(notZero)
    s ← s - 1

  operation signal
    s ← s + 1
    signalC(notZero)
```

p	q

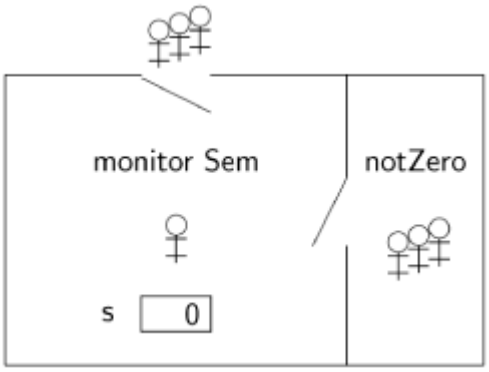
loop forever
  non-critical section
	loop forever
	  non-critical section

p1: Sem.wait critical section p2: Sem.signal	q1: Sem.wait critical section q2: Sem.signal

We refrain from giving line numbers for the statements in the monitor because each operation is executed as a single atomic operation.

The monitor implementation follows directly from the definition of semaphores. The integer component of the semaphore is stored in the variable `s` and the condition variable `cond` implements the queue of blocked processes. By convention, condition variables are named with the condition you want to be true. `waitC(notZero)` is read as "wait for `notZero` to be true," and `signalC(notZero)` is read as "signal that `notZero` is true." The names `waitC` and `signalC` are intended to reduce the possibility of confusion with the similarly-named semaphore statements.

If the value of `s` is zero, the process executing `Sem.wait`—the simulated semaphore operation—executes the monitor statement `waitC(notZero)`. The process is said to be blocked *on the condition*:



A process executing `waitC` blocks unconditionally, because we assume that the condition has been tested for in a preceding `if` statement; since the entire monitor operation is atomic, the value of the condition cannot change between testing its value and executing `waitC`. When a process executes a `Sem.signal` operation, it unblocks the first process (if any) blocked on that condition.

Operations on Condition Variables

With each condition variable is associated a *FIFO queue of blocked processes*. Here is the definition of the execution of the *atomic* operations on condition variables by an arbitrary process `p`:

waitC(cond)

```
append p to cond
p.state ← blocked
monitor.lock ← release
```

Process `p` is blocked on the queue `cond`. Process `p` leaves the monitor, releasing the lock that ensures mutual exclusion in accessing the monitor.

signalC(cond)

```
if cond ≠ empty
    remove head of cond and assign to q
    q.state ← ready
```

If the queue `cond` is nonempty, the process at the head of the queue is unblocked.

There is also an operation that checks if the queue is empty:

empty(cond)

```
return cond = empty
```

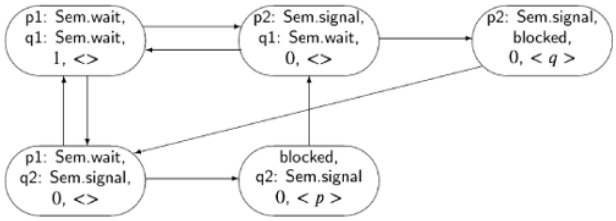
Since the state of the process executing the monitor operation `waitC` becomes blocked, it is clear that it must release the lock to enable another process to enter the monitor (and eventually signal the condition).[2] In the case of the `signalC` operation, the unblocked process becomes ready and there is no obvious requirement for the signaling operation to leave the monitor (see Section 7.5).

The following table summarizes the differences between the monitor operations and the similarly-named semaphore operations:

Semaphore	Monitor
`wait` may or may not block	`waitC` always blocks
`signal` always has an effect	`signalC` has no effect if queue is empty
`signal` unblocks an arbitrary blocked process	`signalC` unblocks the process at the head of the queue
a process unblocked by `signal` can resume execution immediately	a process unblocked by `signalC` must wait for the signaling process to leave monitor

Correctness of the Semaphore Simulation

When constructing a state diagram for a program with monitors,*all* the statements of a monitor operation can be considered to be a single step because they are executed under mutual exclusion and no interleaving is possible between the statements. In Algorithm 7.2, each state has four components: the control pointers of the two processes `p` and `q`, the value of the monitor variable `s` and the queue associated with the condition variable `notZero`. Here is the state diagram assuming that the value of `s` has been initialized to 1:



Consider, first, the transition from the state (`p2: Sem.signal`, `q1: Sem.wait`, 0,<>) at the top center of the diagram to the state (`p2: Sem.signal`, `blocked`, 0,< *q* >) at the upper right. Process `q` executes the `Sem.wait` operation, finds that the value of `s` is 0 and executes the `waitC` operation; it is then blocked on the queue for the condition variable `notZero`.

Consider, now, the transition from (`p2: Sem.signal`, `blocked`, 0,< *q* >) to the state (`p1: Sem.wait`, `q2: Sem.signal`, 0,<>) at the lower left of the diagram. Process `p` executes the `Sem.signal` operation, incrementing `s`, and then `signalC` will unblock process `q`. As we shall discuss in Section 7.5, process `q` *immediately resumes* the execution of its `Sem.wait` operation, so this can be considered as part of the same atomic statement. `q` will decrement `s` back to zero and exit the monitor. Since `signalC(nonZero)` is the last

statement of the `Sem.signal` operation executed by process `p`, we may also consider that that process exits the monitor as part of the atomic statement. We end up in a state where process `q` is in its critical section, denoted in the abbreviated algorithm by the control pointer indicating the next invocation of `Sem.signal`, while process `p` is outside its critical section, with its control pointer indicating the next invocation of `Sem.wait`.

There is no state of the form (`p2: Sem.signal`, `q2: Sem.signal`, $\cdots$, $\cdots$) in the diagram, so the mutual exclusion requirement is satisfied.

The state diagram for a monitor can be relatively simple, because the internal transitions of the monitor statements can be grouped into a single transition.

The Producer–Consumer Problem

Algorithm 7.3 is a solution for the producer–consumer problem with a finite buffer using a monitor. Two condition variables are used and the conditions are explicitly checked to see if a process needs to be suspended. The entire processing of the buffer is encapsulated within the monitor and the buffer data structure is not visible to the producer and consumer processes.

Table 7.3. producer–consumer (finite buffer, monitor)

```
monitor PC
  bufferType buffer ← empty
  condition notEmpty
  condition notFull

  operation append(datatype v)
    if buffer is full
       waitC(notFull)
    append(v, buffer)
    signalC(notEmpty)

  operation take()
    datatype w
    if buffer is empty
       waitC(notEmpty)
    w ← head(buffer)
    signalC(notFull)
    return w
```

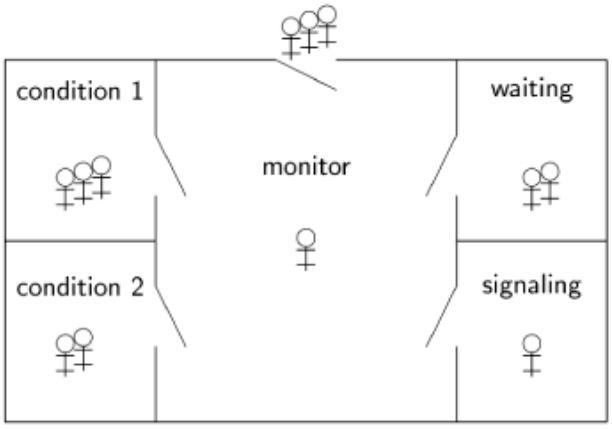
producer	**consumer**
`datatype d` `loop forever`	`datatype d` `loop forever`
`p1: d ← produce` `p2: PC.append(d)`	`q1: d ← PC.take` `q2: consume(d)`

The Immediate Resumption Requirement

The definition of `signalC(cond)` requires that it unblock the first process blocked on the queue for `cond`. When this occurs, the signaling process (say `p`) can now continue to execute the next statement after `signalC(cond)`, while the unblocked process (say `q`) can execute the next statement after the `waitC(cond)` that caused it to block. But this is not a valid state, since the specification of a monitor requires that at most one process at a time can be executing statements of a monitor operation. Either `p` or `q` will have to be blocked until the other completes its monitor operation. That is, we have to specify if signaling processes are given precedence over waiting processes, or vice versa, or perhaps the selection of the next process is arbitrary. There may also be processes blocked on the entry to the monitor, and the specification of precedence has to take them into account.

Note that the terminology is rather confusing, because the waiting processes are processes that have just been *released* from being blocked, rather than processes that are waiting because they are blocked.

The following diagram shows the states that processes can be in: waiting to enter the monitor, executing within the monitor (but only one), blocked on condition queues, on a queue of processes just released from waiting on a condition, and on a queue of processes that have just completed a `signal` operation:



Let us denote the precedence of the signaling processes by *S*, that of the waiting processes by *W* and that of the processes blocked on the entry by *E*. There are thirteen different ways of assigning precedence, but many of them make no sense. For example, it does not make sense to have *E* greater than either *W* or *S*, because that would quickly cause starvation as new processes enter the monitor before earlier ones have left. The classical monitor specifies that $E < S < W$ and later we will discuss the specification $E = W < S$ that is implemented in Java. See [15] for an analysis of monitor definitions.

The specification $E < S < W$ is called the *immediate resumption requirement (IRR)*, or *signal and urgent wait*. It means that when a process blocked on a condition variable is signaled, it immediately begins executing ahead of the signaling process. It is easy to see the rationale behind IRR. Presumably, the signaling process has changed the state of the monitor (modified its variables) so that the condition now holds; if the waiting process resumes immediately, it can assume the condition does hold and continue with the statements in

its operation. For example, look again at the monitor that simulates a semaphore (Algorithm 7.2). The `Sem.signal` operation increments `s` just before executing the `signalC(notZero)` statement; if the `Sem.wait` operation resumes immediately, it will find `s` to be nonzero, the condition it was waiting for.

Similarly, in Algorithm 7.3, the monitor solution for the producer–consumer problem, `append(v, Buffer)` immediately precedes `signalC(notEmpty)`, ensuring that the buffer is not empty when a consumer process is unblocked, and the statement `w ← head(Buffer)` immediately precedes `signalC(notFull)`, ensuring that the buffer is not full when a producer process is unblocked. With immediate resumption there is no need to recheck the status of the buffer.

Without the IRR, it would be possible for the signaling process to continue its execution, causing the condition to again become false. Waiting processes would have to recheck the boolean expression for the condition in a `while` loop:

```
while s = 0
   waitC(notZero)
s ← s - 1
```

The disadvantage of the IRR is that the signaling process might be unnecessarily delayed, resulting in less concurrency than would otherwise be possible. However, if—as in our examples—`signalC(cond)` is the last statement in a monitor operation, there is no need to block the execution of the process that invoked the operation.

The Problem of the Readers and Writers

The problem of the readers and writers is similar to the mutual exclusion problem in that several processes are competing for access to a critical section. In this problem, however, we divide the processes into two classes:

Readers. Processes which are required to exclude writers but not other readers.

Writers. Processes which are required to exclude both readers and other writers.

The problem is an abstraction of access to databases, where there is no danger in having several processes read data concurrently, but writing or modifying data must be done under mutual exclusion to ensure consistency of the data. The concept database must be understood in the widest possible sense; even a couple of memory words in a real-time system can be considered a database that needs mutual exclusion when being written to, while several processes can read it simultaneously.

Algorithm 7.4 is a solution to the problem using monitors. We will use the term “reader” for any process executing the statements of `reader` and “writer” for any process executing the statements of `writer`.

Table 7.4. Readers and writers with a monitor

```
monitor RW
  integer readers ← 0
  integer writers ← 0
  condition OKtoRead, OKtoWrite

  operation StartRead
    if writers ≠ 0 or not empty(OKtoWrite)
       waitC(OKtoRead)
    readers ← readers + 1
    signalC(OKtoRead)

operation EndRead
  readers ← readers - 1
  if readers = 0
     signalC(OKtoWrite)

operation StartWrite
  if writers ≠ 0 or readers ≠ 0
     waitC(OKtoWrite)
  writers ← writers + 1

operation EndWrite
   writers ← writers - 1
   if empty(OKtoRead)
      then signalC(OKtoWrite)
      else signalC(OKtoRead)
```

reader

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

writer

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

The monitor uses four variables:

readers. The number of readers currently reading the database after successfully executing `StartRead` but before executing `EndRead`.

writers. The number of writers currently writing to the database after successfully executing `StartWrite` but before executing `EndWrite`.

OKtoRead. A condition variable for blocking readers until it is "OK to read."

OKtoWrite. A condition variable for blocking writers until it is "OK to write."

The general form of the monitor code is not difficult to follow. The variables `readers` and `writers` are incremented in the `Start` operations and decremented in the `End` operations to reflect the natural invariants expressed in the definitions above. At the beginning of the `Start` operations, a boolean expression is checked to see if the process should be blocked, and at the end of the `End` operations, another boolean expression is checked to see if some condition should be signaled to unblock a process.

A reader is suspended if some process is currently writing (*writers* ≠ 0) or if some process is waiting to write (¬*empty*(*OKtoWrite*)). The first condition is obviously required by the specification of the

problem. The second condition is a decision to give the first blocked writer precedence over waiting readers. A writer is blocked only if there are processes currently reading (*readers* ≠ 0) or writing (*writers* ≠ 0). `StartWrite` does not check the condition queues.

`EndRead` executes `signalC(OKtoWrite)` if there are no more readers. If there are blocked writers, one will be unblocked and allowed to complete `StartWrite`. `EndWrite` gives precedence to the first blocked reader, if any; otherwise it unblocks a blocked writer, if any.

Finally, what is the function of `signalC(OKtoRead)` in the operation `StartRead`? This statement performs a *cascaded unblocking* of the blocked readers. Upon termination of a writer, the algorithm gives precedence to unblocking a blocked reader over a blocked writer. However, if one reader is allowed to commence reading, we might as well unblock them all. When the first reader completes `StartRead`, it will signal the next reader and so on, until this cascade of signals unblocks all the blocked readers.

What about readers that attempt to start reading during the cascaded unblocking? Will they have precedence over blocked writers? By the immediate resumption requirement, the cascaded unblocking will run to completion before any new reader is allowed to commence execution of a monitor operation. When the last `signalC(OKtoRead)` is executed (and does nothing because the condition queue is empty), the monitor will be released and a new reader may enter. However, it is subject to the check in `StartRead` that will cause it to block if there are waiting writers.

These rules ensure that there is no starvation of either readers or writers. If there are blocked writers, a new reader is required to wait until the termination of (at least) the first write. If there are blocked readers, they will (all) be unblocked before the next write.

Correctness of the Readers and Writers Algorithm[A]

Proving monitors can be relatively succinct, because monitor invariants are only required to hold outside the monitor procedures themselves. Furthermore, the immediate resumption requirement allows us to infer that what was true before executing a signal is true when a process blocked on that condition becomes unblocked.

Let R be the number of readers currently reading and W the number of writers currently writing. The proof of the following lemma is immediate from the structure of the program.

Example 7.1. Lemma

The following formulas are invariant:

$$R \geq 0,$$

$$W \geq 0,$$

R	=	*readers,*
W	=	*writers*.

We implicitly use these invariants to identify changes in the values of the program variables `readers` and `writers` with changes in the variables R and W that count processes. The safety of the algorithm is expressed using the variables R and W:

Example 7.2. Theorem

The following formula is invariant:

$(R > 0 \rightarrow W = 0) \wedge (W \leq 1) \wedge (W = 1 \rightarrow R = 0)$.

In words, if a reader is reading then no writer is writing; at most one writer is writing; and if a writer is writing then no reader is reading.

Proof: Clearly, the formula is initially true. Because of the properties of the monitor, there are eight atomic statements: execution of the four monitor operations from start to completion, and execution of the four monitor operations that include `signalC` statements unblocking a waiting process.

Consider first executing a monitor operation from start to completion:

- **`StartRead`:** This operation increments R and could falsify $R > 0 \rightarrow W = 0$, but the `if` statement ensures that the operation completes only if $W = 0$ is true. The operation could falsify $R = 0$ and hence $W = 1 \rightarrow R = 0$, but again the `if` statement ensures that the operation completes only if $W = 1$ is false.
- **`EndRead`:** This operation decrements R, possibly making $R > 0$ false and $R = 0$ true, but these changes cannot falsify the invariant.
- **`StartWrite`:** This operation increments W and could falsify $W \leq 1$, but the `if` statement ensures that the operation completes only if $W = 0$ is true. The operation could falsify $W = 0$ and hence $R > 0 \rightarrow W = 0$, but the `if` statement ensures that $R > 0$ is false.
- **`EndWrite`:** This operation decrements W, so $W = 0$ and $W \leq 1$ could both become true while $W = 1$ becomes false; these changes cannot falsify the invariant.

Consider now the monitor operations that result in unblocking a process from the queue of a condition variable.

- **`signal(OKtoRead)` in `StartRead`:** The execution of `StartRead` including `signal(OKtoRead)` can cause a reader `r` to become unblocked. But the `if` statement ensures that this happens only if $W = 0$, and the IRR ensures that `r` continues its execution immediately so that $W = 0$ remains true.

- **`signal(OKtoRead)` in `EndWrite`:** By the inductive hypothesis, $W \leq 1$, so $W = 0$ when the reader resumes, preserving the truth of the invariant.
- **`signal(OKtoWrite)` in `EndRead`:** The signal is executed only if $R = 1$ upon starting the operation; upon completion $R = 0$ becomes true, preserving the truth of the first and third subformulas. In addition, if $R = 1$, by the inductive hypothesis we also have that $W = 0$ was true. By the IRR for `StartWrite` of the unblocked writer, $W = 1$ becomes true, preserving the truth of the second subformula.
- **`signal(OKtoWrite)` in `EndWrite`:** By the inductive hypothesis $W \leq 1$, the writer executing `EndWrite` must be the only writer that is writing so that $W = 1$. By the IRR, the unblocked writer will complete `StartWrite`, preserving the truth of $W = 1$ and hence $W \leq 1$. The value of R is, of course, not affected, so the truth of the other subformulas is preserved.

The expressions `not empty(OKtoWrite)` in `StartRead` and `empty(OKtoRead)` in `EndWrite` were not used in the proof because they are not relevant to the safety of the algorithm.

We now prove that readers are not starved and leave the proof that writers are not starved as an exercise. It is important to note that freedom from starvation holds only for processes that start executing `StartRead` or `StartWrite`, because starvation is possible at the entrance to a monitor. Furthermore, both reading and writing are assumed to progress like critical sections, so starvation can occur only if a process is blocked indefinitely on a condition variable.

Let us start with a lemma relating the condition variables to the number of processes reading and writing:

Example 7.3. Lemma

The following formulas are invariant:

$\neg$ *empty* (*OKtoRead*)	$\rightarrow$	$(W \neq 0) \vee \neg$ *empty* (*OKtoWrite*),
$\neg$ *empty* (*OKtoWrite*)	$\rightarrow$	$(R \neq 0) \vee (W \neq 0)$.

Proof: Clearly, both formulas are true initially since condition variables are initialized as empty queues.

The antecedent of the first formula can only become true by executing `StartRead`, but the `if` statement ensures that the consequent is also true. Can the consequent of the formula become false while the antecedent is true? One way this can happen is by executing `EndWrite` for the last writer. By the assumption of the truth of the antecedent $\neg$ *empty* (*OKtoRead*), `signalC(OKtoRead)` is executed. By the IRR, this leads to a cascade of signals that form part of the atomic monitor operation, and the final signal falsifies $\neg$

empty (*OKtoRead*). Alternatively, `EndRead` could be executed for the last reader, causing `signalC(OKtoWrite)` to falsify ¬ *empty* (*OKtoWrite*). But this causes $W \neq 0$ to become true, so the consequent remains true.

The truth of the second formula is preserved when executing `StartWrite` by the condition of the `if` statement. Can the consequent of the formula become false while the antecedent is true? Executing `EndRead` for the last reader falsifies $R \neq 0$, but since ¬ *empty* (*OKtoWrite*) by assumption, `signalC(OKtoWrite)` makes $W \neq 0$ true. Executing `EndWrite` for the last (only) writer could falsify $W \neq 0$, but one of the `signalC` statements will make either $R \neq 0$ or $W \neq 0$ true.

Example 7.4. Theorem

Algorithm 7.4 is free from starvation of readers.

Proof: If a reader is starved is must be blocked indefinitely on `OKtoRead`. We will show that a `signal(OKtoRead)` statement must eventually be executed, unblocking the first reader on the queue. Since the condition queues are assumed to be FIFO, it follows by (numerical) induction on the position of a reader in the queue that it will eventually be unblocked and allowed to read.

To show

¬ *empty* (*OKtoRead*) → $\Diamond$ *signalC*(*OKtoRead*),

assume to the contrary that

¬ *empty*(*OKtoRead*) ∧ *2* ¬ *signalC*(*OKtoRead*).

By the first invariant of Lemma 7.3, ($W \neq 0$) ∨ ¬ *empty* (*OKtoWrite*). If $W \neq 0$, by progress of the writer, eventually `EndWrite` is executed; `signalC(OKtoRead)` will be executed since by assumption ¬ *empty* (*OKtoRead*) is true (and it remains true until some `signalC(OKtoRead)` statement is executed), a contradiction.

If ¬ *empty* (*OKtoWrite*) is true, by the second invariant of Lemma 7.3, ($R \neq 0$) ∨ ($W \neq 0$). We have just shown that $W \neq 0$ leads to a contradiction, so consider the case that $R \neq 0$. By progress of readers, if no new readers execute `StartRead`, all readers eventually execute `EndReader`, and the last one will execute `signalC(OKtoWrite)`; since we assumed that ¬ *empty* (*OKtoWrite*) is true, a writer will be unblocked making $W \neq 0$ true, and reducing this case to the previous one. The assumption that no new readers successfully execute `StartRead` holds because ¬ *empty* (*OKtoWrite*) will cause a new reader to be blocked on `OKtoRead`.

A Monitor Solution for the Dining Philosophers

Algorithm 6.10, an attempt at solving the problem of the dining philosophers with semaphores, suffered from deadlock because there is no way to test the value of two fork semaphores simultaneously. With monitors, the test can be encapsulated, so that a philosopher waits until both forks are free before taking them.

Algorithm 7.5 is a solution using a monitor. The monitor maintains an array `fork` which counts the number of free forks *available to* each philosopher. The `take-Forks` operation waits on a condition

variable until two forks are available. Before leaving the monitor with these two forks, it decrements the number of forks available to its neighbors. After eating, a philosopher calls `releaseForks`, which, in addition to updating the array `fork`, checks if freeing these forks makes it possible to signal a neighbor.

Let *eating* [*i*] be true if philosopher *i* is eating, that is, if she has successfully executed `takeForks(i)` and has not yet executed `releaseForks(i)`. We leave it as an exercise to show that *eating* [*i*] ↔ (*forks*[*i*] = 2) is invariant. This formula expresses the requirement that a philosopher eats only if she has two forks.

Table 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])
```

```
philosopher i
```

```
      loop forever
```

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

Example 7.5. Theorem

Algorithm 7.5 is free from deadlock.

Proof: Let *E* be the number of philosophers who are eating, In the exercises, we ask you to show that the following formulas are invariant:

Equation 7-1.

$$\neg\, empty(OKtoEat[i]) \;\rightarrow\; (fork[i] < 2).$$

Equation 7-2.

$$\sum_{i=0}^{4} fork[i] = 10 - 2 * E.$$

Deadlock implies $E = 0$ and all philosophers are enqueued on `OKtoEat`. If no philosophers are eating, from (7.2) we conclude $\sum fork[i] = 10$. If they are all enqueued waiting to eat, from (7.1) we conclude $\sum fork[i] \leq 5$ which contradicts the previous formula.

Starvation is possible in Algorithm 7.5. Two philosophers can conspire to starve their mutual neighbor as shown in the following scenario (with the obvious abbreviations):

n	phil1	phil2	phil3	*f*0
1	**take(1)**	take(2)	take(3)	2
2	release(1)	take(2)	**take(3)**	1
3	release(1)	**take(2)** to **waitC(OK[2])**	release(3)	1
4	**release(1)**	(blocked)	release(3)	1
5	**take(1)**	(blocked)	release(3)	2
6	release(1)	(blocked)	**release(3)**	1
7	release(1)	(blocked)	**take(3)**	1

Philosophers 1 and 3 both need a fork shared with philosopher 2. In lines 1 and 2, they each take a pair of forks, so philosopher 2 is blocked when trying to take forks in line 3. In lines 4 and 5 philosopher 1 releases her forks and then immediately takes them again; since *forks* [*2*] does not receive the value 2, philosopher 2 is not signaled. Similarly, in lines 6 and 7, philosopher 3 releases her forks and then immediately takes them again. Lines 4 through 7 of the scenario can be continued indefinitely, demonstrating starvation of philosopher 2.

Monitors in BACI^L

The monitor of the C implementation of Algorithm 7.4 is shown in Listing 7.1. The program is written in the dialect supported by BACI. The syntax and the semantics are very similar to those of Algorithm

7.4, except that the condition queues are not FIFO, and boolean variables are not supported. A Pascal program is also available in the archive.

Protected Objects

In the classical monitor, tests, assignment statements and the associated statements `waitC`, `signalC` and `emptyC` must be explicitly programmed. The protected object simplifies the programming of monitors by encapsulating the manipulation of the queues together with the evaluation of the expressions. This also enables significant optimization of the implementation. Protected objects were defined and implemented in the language Ada, though the basic concepts have been around since the beginning of research on monitors. The construct is so elegant that it is worthwhile studying it in our language-independent notation, even if you don't use Ada in your programming.

Example 7.1. A C solution for the problem of the readers and writers

```
1  monitor RW {
2   int readers = 0, writing = 1;
3   condition OKtoRead, OKtoWrite;
4
5   void StartRead() {
6     if (writing || ! empty(OKtoWrite))
7         waitc(OKtoRead);
8     readers = readers + 1;
9     signalc(OKtoRead);
10  }
11
12  void EndRead() {
13    readers = readers - 1;
14    if (readers == 0)
15        signalc(OKtoWrite);
16  }
17
18  void StartWrite () {
19    if (writing || (readers != 0))
20        waitc(OKtoWrite);
21    writing = 1;
22  }
23
24  void EndWrite() {
25    writing = 0;
26    if (empty(OKtoRead))
27        signalc(OKtoWrite);
28    else
29        signalc(OKtoRead);
30  }
31 }
```

Table 7.6. Readers and writers with a protected object

```
protected object RW
   integer readers ← 0
   boolean writing ← false

   operation StartRead when not writing
     readers ← readers + 1

   operation EndRead
     readers ← readers - 1

   operation StartWrite when not writing and readers = 0
     writing ← true
```

```
    operation EndWrite
      writing ← false
```

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

Algorithm 7.6 is a solution to the problem of the readers and writers that uses a protected object. As with monitors, execution of the operations of a protected object is done under mutual exclusion. Within the operations, however, the code consists only of the trivial statements that modify the variables.

Synchronization is performed upon entry to and exit from an operation. An operation may have a suffix `when expression`, called a *barrier*. The operation may start executing only when the expression of the barrier is true. If the barrier is false, the process is blocked on a FIFO queue; there is a separate queue associated with each entry. (By a slight abuse of language, we will say that the barrier is evaluated and its value is true or false.) In Algorithm 7.6, `StartRead` can be executed only if no process is writing and `StartWrite` can be executed only if no process is either reading or writing.

When can a process become unblocked? Assuming that the barrier refers only to variables of the protected object, their values can be changed only by executing an operation of the protected object. So, when the execution of an operation has been completed, all barriers are re-evaluated; if one of them is now true, the process at the head of the FIFO queue associated with that barrier is unblocked. Since the process evaluating the barrier is terminating its operation, there is no failure of mutual exclusion when a process is unblocked.

In the terminology of Section 7.5, the precedence specification of protected objects is $E < W$. Signaling is done implicitly upon the completion of a protected action so there are no signaling processes. Servicing entry queues is given precedence over new processes trying to enter the protected objects.

Starvation is possible in Algorithm 7.6; we leave it as an exercise to develop algorithms that are free from starvation.

Protected objects are simpler than monitors because the barriers are used implicitly to carry out what was done explicitly with condition variables. In addition, protected objects enable a more efficient implementation. Consider the following outline of a scenario of Algorithm 7.4, the monitor solution to the readers and writers problem:

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

When the `writer` executes `signalC(OKtoRead)`, by the IRR it must be blocked to let a reader process be unblocked. But as soon as the process exits its monitor operation, the blocked signaling process is unblocked, only to exit its monitor operation. These context switches are denoted by the breaks in the table. Even with the optimization of combining `signalC` with the monitor exit, there will still be a context switch from the signaling process to the waiting process.

Consider now a similar scenario for Algorithm 7.6, the solution that uses a protected object:

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

```
read the data                . . .
```

The variables of the protected object are accessible only from within the object itself. When the `writer` resets the variable `writing`, it is executing under mutual exclusion, so it might as well evaluate the barrier for `reader` and even execute the statements of `reader`'s entry! When it is finished, both processes can continue executing other statements. Thus a *protected action* can include not just the execution of the body of an operation, but also the evaluation of the barriers and the execution of the bodies of other operations. This reduces the number of context switches that must be done; as a context switch is usually much more time-consuming than simple statements, protected objects can be more efficient than monitors.

To enable this optimization, it is forbidden for barriers to depend on parameters of the entries. That way there need be only one queue per *entry*, not one queue per *entry call*, and the data needed to evaluate the barrier are globally accessible to all operations, not just locally in the entry itself. In order to block a process on a queue that depends on data in the call, the entry call first uses a barrier that does not depend on this data; then, within the protected operation, the data are checked and the call is *requeued* on another entry. A discussion of this topic is beyond the scope of this book.

Protected Objects in Ada[L]

If you only want to protect a small amount of data (and freedom from starvation is not required), the problem of the readers and writers can be solved simply by using procedures and functions of protected objects:

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

The operations are declared in the public part of the specification, while variables that are global to the operations are declared in the private part. These "monitor variables" are only accessible in the bodies of the operations declared in the body of the protected object and not by its clients. *procedures* are operations that are never blocked except for the ordinary mutual exclusion of access to a protected object. *functions* are limited to read-only access of the variables and more than one task may call a function concurrently. As discussed in the previous section, an *entry* is guarded by a barrier.

Listing 7.2 shows the implementation of Algorithm 7.6 in Ada. Full explanations of protected objects in Ada can be found in [7, 18].

Monitors in Java[L]

In Java, there is no special construct for a monitor; instead, *every* object has a lock that can be used for synchronizing access to fields of the object. Listing 7.3 shows a class for the producer–consumer problem. Once the class is defined, we can declare objects of this class:

```
PCMonitor monitor = new PCMonitor();
```

and then invoke its methods:

```
monitor.Append(5);
int   n = monitor.Take();
```

The addition of the **synchronized** specifier to the declaration of a method means that a process must acquire the lock for that object before executing the method. From within a **synchronized** method, another **synchronized** method of the same object can be invoked while the process retains the lock. Upon return from the last synchronized method invoked on an object, the lock is released.

There are no fairness specifications in Java, so if there is contention to call synchronized methods of an object an arbitrary process will obtain the lock.

A class all of whose methods are declared **synchronized** can be called a Java monitor. This is just a convention because—unlike the case with protected objects in Ada—there is no syntactical requirement that all of the methods of an object be declared as **synchronized**. It is up to the programmer to ensure that no synchronization errors occur when calling an unsynchronized method.

Example 7.2. Solution for the problem of the readers and writers in Ada

```
1   protected RW is
2      entry StartRead;
3      procedure EndRead;
4      entry StartWrite;
5      procedure EndWrite;
6   private
7      Readers: Natural :=0;
8      Writing: Boolean := false;
9   end RW;
10
11  protected body RW is
12     entry StartRead
13       when not Writing is
14     begin
15       Readers := Readers + 1;
16     end StartRead;
17
18     procedure EndRead is
19     begin
20       Readers := Readers - 1;
21     end EndRead;
22
23     entry StartWrite
24       when not Writing and Readers = 0 is
25     begin
26       Writing := true;
27     end StartWrite;
28
29     procedure EndWrite is
30     begin
31       Writing := false;
32     end EndWrite;
33  end RW;
```

Example 7.3. A Java monitor for the producer–consumer problem

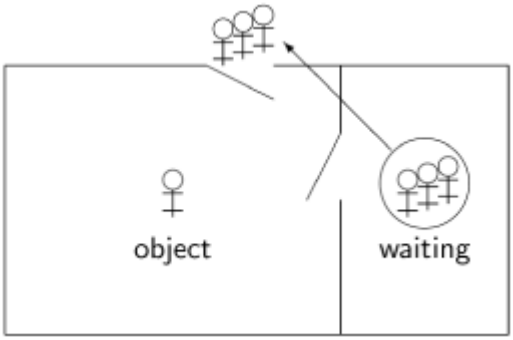
```
class PCMonitor {
  final int N = 5;
  int Oldest = 0, Newest = 0;
  volatile int Count = 0;
  int Buffer [] = new int[N];

  synchronized void Append(int V) {
    while (Count == N)
      try {
         wait();
      } catch (InterruptedException e) {}
    Buffer [ Newest] = V;
    Newest = (Newest + 1) % N;
    Count = Count + 1;
    notifyAll ();
  }

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

If a process needs to block itself—for example, to avoid appending to a full buffer or taking from an empty buffer—it can invoke the method `wait`. (This method throws `InterruptedException` which must be thrown or caught.) The method `notify` is similar to the monitor statement `signalC` and will unblock one process, while `notifyAll` will unblock all processes blocked on this object.

The following diagram shows the structure of a Java monitor:



There is a lock which controls the door to the monitor and only one process at a time is allowed to hold the lock and to execute a synchronized method. A process executing `wait` blocks on this object; the process joins the *wait set* associated with the object shown on the righthand side of the diagram. When a process joins the wait set, it also releases the lock so that another process can enter the monitor. `notify` will release one process from the wait set, while `notifyAll` will release all of them, as symbolized by the circle and arrow.

A process must hold the synchronization lock in order to execute either `notify` or `notifyAll`, and it continues to hold the lock until it exits the method or blocks on a `wait`. The unblocked process or

processes must reacquire the lock (only one at a time, of course) in order to execute, whereas with the IRR in the classical monitor, the lock is passed from the signaling process to the unblocked process. Furthermore, the unblocked processes receive no precedence over other processes trying to acquire the lock. In the terminology of Section 7.5, the precedence specification of Java monitors is $E = W < S$.

An unblocked process cannot assume that the condition it is waiting for is true, so it must recheck it in a `while` loop before continuing:[3]

```
synchronized method1() {
  while (! booleanExpression)
      wait();
  // Assume booleanExpression is true
}
synchronized method2() {
  // Make booleanExpression true
  notifyAll ()
}
```

If the expression in the loop is false, the process joins the wait set until it is unblocked. When it acquires the lock again it will recheck the expression; if it is true, it remains true because the process holds the lock of the monitor and no other process can falsify it. On the other hand, if the expression is again false, the process will rejoin the wait set.

When `notify` is invoked *an arbitrary process may be unblocked* so starvation is possible in a Java monitor. Starvation-free algorithms may be written using the concurrency constructs defined in the library `java.util.concurrent`.

A Java monitor has no condition variables or entries, so that it is impossible to wait on a specific condition.[4] Suppose that there are two processes waiting for conditions to become true, and suppose that process `p` has called `method1` and process `q` has called `method2`, and that both have invoked `wait` after testing for a condition. We want to be able to unblock the process whose condition is now true:

```
synchronized method1() {
  if (x == 0)
    wait();
}
synchronized method2() {
  if (y == 0)
     wait();
}
synchronized method3(...) {
  if (...) {
     x = 10;
     notify (someProcessBlockedInMethod1); // Not legal!!
  }
  else {
     y = 20;
      notify (someProcessBlockedInMethod2); // Not legal!!
  }
}
```

Since an arbitrary process will be unblocked, this use of `notify` is not correct, because it could unblock the wrong process. Instead, we have to use `notifyAll` which unblocks *all* processes; these processes now compete to enter the monitor again. A process must wait within a loop to check if its condition is now true:

```
synchronized method1() {
  while (x == 0)
```

```
      wait();
    }
    synchronized method2() {
      while (y == 0)
       wait();
    }
    synchronized method3(. . .) {
      if (. . .)
         x = 10;
      else
         y = 20;
      notifyAll ();
    }
```

If the wrong process is unblocked it will return itself to the wait set.

However, if only one process is waiting for a lock, or if all the processes waiting are waiting for the same condition, then `notify` can be used, at a significant saving in context switching.

A Java solution for the problem of the readers and writers is shown in Listing 7.4. The program contains no provision for preventing starvation.

Synchronized Blocks

We have shown a programming style using synchronized methods to build monitors in Java. But **synchronized** is more versatile because it can be applied to any object and to any sequence of statements, not just to a method. For example, we can obtain the equivalent of a critical section protected by a binary semaphore simply by declaring an empty object and then synchronizing on its lock. Given the declaration:

```
Object obj = new Object();
```

the statements within any synchronized block on this object in any process are executed under mutual exclusion:

```
synchronized (obj) {
    // critical section
}
```

Example 7.4. A Java monitor for the problem of the readers and the writers

```
 1  class RWMonitor {
 2    volatile int readers = 0;
 3    volatile boolean writing = false;
 4
 5    synchronized void StartRead() {
 6      while (writing)
 7        try {
 8           wait();
 9        } catch (InterruptedException e) {}
10      readers = readers + 1;
11      notifyAll ();
12    }
13    synchronized void EndRead() {
14      readers = readers - 1;
15      if (readers == 0)
16          notifyAll ();
17    }
18    synchronized void StartWrite() {
19      while (writing || (readers != 0))
20        try {
21          wait();
```

```
22        } catch (InterruptedException e) {}
23      writing = true;
24    }
25    synchronized void EndWrite() {
26      writing = false;
27      notifyAll ();
28    }
29  }
```

There is no guarantee of liveness, however, because in the presence of contention for a specific lock, an arbitrary process succeeds in acquiring the lock.

Simulating Monitors in Promela[L]

Promela is not an appropriate formalism for studying monitors, since monitors are a formalism for encapsulation, which is not supported in Promela. We will show how to implement the synchronization constructs of monitors in Promela.

A monitor has a lock associated with it; the code of each procedure of the monitor must be enclosed within `enterMon` and `leaveMon`:

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

The variable `lock` is true when the lock is held by a process, so `enterMon` simply blocks until the lock is false.

Each simulated condition variable contains a field `gate` for blocking the process and a count of processing `waiting` on the condition:

```
typedef Condition {
  bool gate;
   byte waiting;
}
```

The field `waiting` is used only to implement `emptyC`:

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

The name has been changed since **empty** is a reserved word in Promela.

The operation `waitC` sets `lock` to false to release the monitor and then waits for `C.gate` to become true:

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

`signalC` sets `C.gate` to true so that the blocked process may continue:

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

The `signalC` operation does not release the lock, so the `waitC` operation will be able to execute under the IRR. When the waiting process finally leaves the monitor, it will reset the lock, allowing the signaling process to continue.[5]

We leave the implementation of FIFO condition queues as an exercise.

Transition

Monitor-like constructs are the most popular form of synchronization found in programming languages. These constructs are structured and integrate well with the techniques of object-oriented programming. The original formulation by Hoare and Brinch Hansen has served as inspiration for similar constructs, in particular the synchronized methods of Java, and the very elegant protected objects of Ada. As with semaphores, it is important to understand the precise semantics of the construct that you are using, because the style of programming as well as the correctness and efficiency of the resulting programs depend on the details of the implementation.

Both the semaphore and the monitor are highly centralized constructs, blocking and unblocking processes, maintaining queues of blocked processes and encapsulating data. As multiprocessing and distributed architectures become more popular, there is a need for synchronization constructs that are less centralized. These constructs are based upon communications rather than upon sharing. The next chapter presents several models that achieve synchronization by using communications between sending processes and receiving processes.

Exercises

1. Develop a simulation of monitors by semaphores.

2. Show that in Algorithm 7.4, the integer variable `writers` can be replaced by a boolean variable `writing` that is equivalent to *writers* > 0.

3. Prove that there is no starvation of writers in Algorithm 7.4.

4. Modify the solution to the problem of the readers and the writers so as to implement each of the

following rules:

1. If there are reading processes, a new reader may commence reading even if there are waiting writers.
2. If there are waiting writers, they receive precedence over all waiting readers.
3. If there are waiting readers, no more than two writers will write before a process is allowed to read.

---

5. Prove the invariant *eating* [*i*] ↔ (*forks*[*i*] = 2) from Section 7.8 on the problem of the dining philosophers. Prove also that formulas (7.1) and (7.2) are invariant.

---

6. Here is the declaration of a protected object in Ada for a starvation-free solution of the problem of the readers and writers. The solution uses an extra pair of private entries to distinguish between groups of processes that have been waiting to read or write, and new processes that we want to block until the previous group has succeeded. Write the body of the protected object and prove the correctness of the program.

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

---

7. Here is the declaration of a protected object in Ada for a starvation-free solution of the problem of the readers and writers. The solution uses an extra variable `WaitingToRead` to record the group of waiting readers. Write the body of the protected object and prove the correctness of the program.[6]

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

---

8. In Java, what does **`synchronized (this)`** mean?

What is the difference, if any, between:

```
void p() {
  synchronized (this) {
     // statements
  }
}
```

and

```
synchronized void p() {
   // statements
}
```

9. Suppose that we are performing computations with variables of type **int** in Java, and that these variables are likely to be accessed by other processes. Should you declare the variables to be **volatile** or should you use a **synchronized** block around each access to the variables?

10. Implement FIFO condition queues in Promela.

11. In his original paper on monitors, Hoare allowed the `waitC` statement to have an additional integer parameter called a priority:

```
waitC(cond, priority)
```

Processes are stored on the queue associated with the condition variable in ascending order of the priority given in the `waitC` statement. Show how this can be used to implement a discrete event simulation, where each process executes a computation and then determines the next time at which it is to be executed.

[1] In Java, the same unit, the *class*, is used for both physical and logical encapsulation. In Ada, the physical encapsulation unit is the *package*, which may contain more than one declaration of a *type* and its operations.

[2] In pthreads, there is no implicit lock as there is for a monitor, so a mutex must be explicitly supplied to the operation `pthread_cond_wait`.

[3] We have left out the exception block for clarity.

[4] Condition variables can be defined using the interface `java.util.concurrent.locks.Condition`.

[5] The implementation does not give precendence to signaling processes over entering processes.

[6] My thanks to Andy Wellings and Alan Burns for providing this solution.