PREV
5. Advanced Algorithms for th

NEXT
7. Monitors

Chapter 6. Semaphores

The algorithms for the critical section problem described in the previous chapters can be run on a *bare machine*, that is, they use only the machine language instructions that the computer provides. However, these instructions are too low-level to be used efficiently and reliably. In this chapter, we will study the *semaphore*, which provides a concurrent programming construct on a higher level than machine instructions. Semaphores are usually implemented by an underlying operating system, but we will investigate them by defining the required behavior and assuming that this behavior can be efficiently implemented.

Semaphores are a simple, but successful and widely used, construct, and thus worth exploring in great detail. We start with a section on the concept of process state. Then we define the semaphore construct and show how it trivially solves the critical section problem that we worked so hard to solve in the previous chapters. Section 6.4 defines invariants on semaphores that can be used to prove correctness properties. The next two sections present new problems, where the requirement is not to achieve mutual exclusion but rather cooperation between processes.

Semaphores can be defined in various ways, as discussed in Section 6.8. Section 6.9 introduces Dijkstra's famous problem of the dining philosophers; the problem is of little practical importance, but it is an excellent framework in which to study concurrent programming techniques. The next two sections present advanced algorithms by Barz and Udding that explore the relative strength of different definitions of semaphores. The chapter ends with a discussion of semaphores in various programming languages.

Process States

A multiprocessor system may have more processors than there are processes in a program. In that case, every process is always *running* on some processor. In a multitasking system (and similarly in a

Find answers on the fly, or master something new. Subscribe today. See pricing options.

At any one time, there will be one running process and any number of ready processes. (If no processes are ready to run, an *idle* process will be run.)

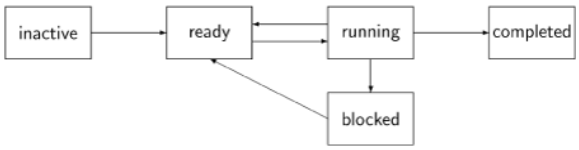
A system program called a *scheduler* is responsible for deciding which of the ready processes should be run, and performing the context switch, replacing a running process with a ready process and changing the state of the running process to ready. Our model of concurrency by interleaving of atomic statements makes no assumptions about the behavior of a scheduler; arbitrary interleaving simply means that the scheduler may perform a context switch at any time, even after executing a single statement of a running process. The reader is referred to textbooks on operating systems [60, 63] for a discussion of the design of schedulers.

Within our model, we do not explicitly mention the concept of a scheduler. However, it is convenient to assume that every process `p` has associated with it an attribute called its *state*, denoted `p.state`. Assignment to this attribute will be used to indicate a change in the state of a process. For example, if `p.state = running` and `q.state = ready`, then a context switch between them can be denoted by:

```
p.state ← ready
q.state ← running
```

The synchronization constructs that we will define assume that there exists another process state called *blocked*. When a process is blocked, it is not ready so it is not a candidate for becoming the running process. A blocked process can be *unblocked* or *awakened* or *released* only if an external action changes the process state from `blocked` to `ready`. At this point the unblocked process becomes a candidate for execution along with all the current ready processes; with one exception (Section 7.5), an unblocked process will not receive any special treatment that would make it the new running process. Again, within our model we do not describe the implementation of the actions of blocking and unblocking, and simply assume that they happen as defined by the synchronization primitives. Note that the `await` statement does not block a process; it is merely a shorthand for a process that is always ready, and may be running and checking the truth of a boolean expression.

The following diagram summarizes the possible state changes of a process:



Initially, a process is *inactive*. At some point it is activated and its state becomes ready. When a concurrent program begins executing, one process is running and the others are ready. While executing, it may encounter a situation that requires the process to cease execution and to become blocked until it is unblocked and returns to the ready state. When a process executes its final statement it becomes *completed*.

In the BACI concurrency simulator, activation of a process occurs when the `coend` statement is executed. The activation of concurrent processes in languages such as Ada and Java is somewhat more complicated, because it must take into account the dependency of one process on another.

Definition of the Semaphore Type

A semaphore `S` is a compound data type with two fields, `S.V` of type non-negative integer, and `S.L` of type set of processes. We are using the familiar dotted notation of records and structures from programming languages. A semaphore `S` must be initialized with a value $k \geq 0$ for `S.V` and with the empty set ∅ for the `S.L`:

```
semaphore S ← (k, ∅)
```

There are two *atomic* operations defined on a semaphore `S` (where `p` denotes the process that is executing the statement):[1]

wait(S)

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

If the value of the integer component is nonzero, decrement its value (and the process `p` can continue its execution); otherwise—it is zero—process `p` is added to the set component and the state of `p` becomes blocked. We say that `p` is blocked *on* the semaphore *S*.

signal(S)

```
if S.L = ∅
   S.V ← S.V + 1
else
   let q be an arbitrary element of S.L
   S.L ← S.L - {q}
   q.state ← ready
```

If `S.L` is empty, increment the value of the integer component; otherwise—`S.L` is nonempty—unblock `q`, an arbitrary element of the set of processes blocked *on* `S.L`. The state of process `p` does not change.

A semaphore whose integer component can take arbitrary non-negative values is called a *general semaphore*. A semaphore whose integer component takes only the values 0 and 1 is called a *binary semaphore*. A binary semaphore is initialized with `(0, ∅)` or `(1, ∅)`. The `wait(S)` instruction is unchanged, but `signal(S)` is now defined by:

```
if S.V = 1
   // undefined
else if S.L = ∅
   S.V ← 1
else // (as above)
   let q be an arbitrary element of S.L
   S.L ← S.L - {q}
   q.state ← ready
```

A binary semaphore is sometimes called a *mutex*. This term is used in pthreads and in `java.util.concurrent`.

Once we have become familiar with semaphores, there will be no need to explicitly write the set of blocked processes `S.L`.

The Critical Section Problem for Two Processes

Using semaphores, the solution of the critical section problem for two processes is trivial (Algorithm 6.1). A process, say `p`, that wishes to enter its critical section executes a preprotocol that consists only

of the `wait(S)` statement. If *S.V* = 1 then `S.V` is decremented and `p` enters its critical section. When `p` exits its critical section and executes the postprotocol consisting only of the `signal(S)` statement, the value of `S.V` will once more be set to 1.

Table 6.1. Critical section with semaphores (two processes)

`binary semaphore S` ← (1, ∅)

p	q
loop forever p1: non-critical section p2: wait(S) p3: critical section p4: signal(S)	loop forever q1: non-critical section q2: wait(S) q3: critical section q4: signal(S)

If `q` attempts to enter its critical section by executing `wait(S)` before `p` has left, *S.V* = 0 and `q` will become blocked on `S`. *S*, the value of the semaphore `S`, will become (0, {*q*}). When `p` leaves the critical section and executes `signal(S)`, the "arbitrary" process in the set `S.L` = {*q*} will be `q`, so that process will be unblocked and can continue into its critical section.

The solution is similar to Algorithm 3.6, the second attempt, except that the definition of the semaphore operations as atomic statements prevents interleaving between the test of `S.V` and the assignment to `S.V`.

To prove the correctness of the solution, consider the abbreviated algorithm where the `non-critical section` and `critical section` statements have been absorbed into the following `wait(S)` and `signal(S)` statements, respectively.

Table 6.2. Critical section with semaphores (two proc., abbrev.)

`binary semaphore S` ← (1, ∅)

p	q
loop forever p1: wait(S) p2: signal(S)	loop forever q1: wait(S) q2: signal(S)

The state diagram for this algorithm is shown in Figure 6.1. Look at the state in the top center, which is reached if initially process `p` executes its `wait` statement, successfully entering the critical section. If process `q` now executes its `wait` statement, it finds *S.V* = 0, so the process is added to *S.L* as shown in the top right state. Since `q`

is blocked, there is no outgoing arrow for `q`, and we have labeled the only arrow by `p` to emphasize that fact. Similarly, in the bottom right state, `p` is blocked and only process `q` has an outgoing arrow.

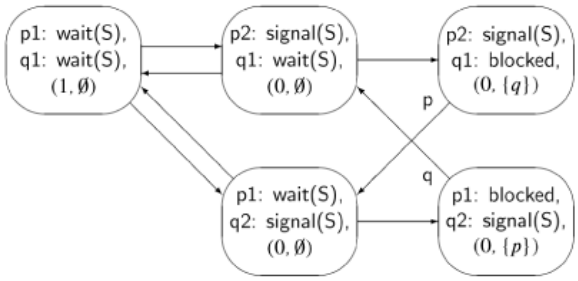


Figure 6.1. State diagram for the semaphore solution

A violation of the mutual exclusion requirement would be a state of the form (`p2: signal(S)`, `q2: signal(S)`, . . .). We immediately see that no such state exists, so we conclude that the solution satisfies this requirement. Similarly, there is no deadlock, since there are no states in which both processes are blocked. Finally, the algorithm is free from starvation, since if a process executes its `wait` statement (thus leaving the non-critical section), it enters either the state with the `signal` statement (and the critical section) or it enters a state in which it is blocked. But the only way out of a blocked state is into a state in which the blocked process continues with its `signal` statement.

Semaphore Invariants

Let *k* be the initial value of the integer component of the semaphore, #*signal*(*S*) the number of `signal(S)` statements that have been executed and #*wait*(*S*) the number of `wait(S)` statements that have been executed. (A process that is blocked when executing `wait(S)` is *not* considered to have successfully executed the statement.)

Example 6.1. Theorem

A semaphore `S` satisfies the following invariants:

Equation 6.1.

$$S.V \geq 0,$$

Equation 6.2.

$$S.V = k + \#signal(S) - \#wait(S),$$

Proof: By definition of a semaphore, $k \geq 0$, so 6.1 is initially true, as is 6.2 since #*signal*(*S*) and #*wait*(*S*) are zero. By assumption, non-semaphore operations cannot change the value of (either component of) a semaphore, so we only need consider the effect of the `wait` and `signal` operations. Clearly, 6.1 is invariant under both operations. If either operation changes *S.V*, the truth of 6.2 is preserved: If a `wait` operation decrements *S.V*, then #*wait*(*S*) is incremented, and similarly a `signal` operation that increments *S.V* also increments #*signal*(*S*). If a `signal` operation unblocks a blocked process, *S.V* is not changed, but both #*signal*(*S*) and #*wait*(*S*) increase by one so the righthand side of 6.2 is not changed.

Example 6.2. Theorem

The semaphore solution for the critical section problem is correct: there is mutual exclusion in the access to the critical section, and the program is free from deadlock and starvation.

Proof: Let *#CS* be the number of processes in their critical sections. We will prove that

Equation 6.3.

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

is invariant. By a trivial induction on the structure of the program it follows that

Equation 6.4.

$$\#CS = \#wait(S) - \#signal(S)$$

is invariant. But invariant (6.2) for the initial value $k = 1$ can be written $\#wait(S) - \#signal(S) = 1 - S.V$, so it follows that $\#CS = 1 - S.V$ is invariant, and this can be written as $\#CS + S.V = 1$.

Since $S.V \geq 0$ by invariant (6.1), simple arithmetic shows that $\#CS \leq 1$, which proves the mutual exclusion property.

For the algorithm to enter deadlock, both processes must be blocked on the `wait` statement and $S.V = 0$. Since neither process is in the critical section, $\#CS = 0$. Therefore $\#CS + S.V = 0$, which contradicts invariant (6.3).

Suppose that some process, say `p`, is starved. It must be indefinitely blocked on the semaphore so $S = (0, S.L)$, where p is in the list $S.L$. By invariant (6.3), $\#CS = 1 - S.V = 1 - 0 = 1$, so the other process `q` is in the critical section. By progress, `q` will complete the critical section and execute `signal(S)`. Since there are only two processes in the program, $S.L$ must *equal* the singleton set $\{p\}$, so `p` will be unblocked and enter its critical section.

The Critical Section Problem for *N* Processes

Using semaphores, the same algorithm gives a solution for the critical section problem for N processes:

Table 6.3. Critical section with semaphores (*N* proc.)

```
binary semaphore S ← (1, ∅)

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

The proofs in Theorem 6.2 remain unchanged for N processes as far as mutual exclusion and freedom from deadlock are concerned. Unfortunately, freedeom from starvation does not hold. For the abbreviated algorithm:

Table 6.4. Critical section with semaphores (*N* proc., abbrev.)

```
binary semaphore S ← (1, ∅)
```

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

consider the following scenario for three processes, where the statement numbers for processes `q` and `r` correspond to those for `p` in Algorithm 6.4:

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

Line 7 is the same as line 3 and the scenario can continue indefinitely in this loop of states. The two processes `p` and `r` "conspire" to starve process `q`.

When there were only two processes, we used the fact that *S.L* is the singleton set *{p}* to claim that process `p` must be unblocked, but with more than two processes this claim no longer holds. Starvation is caused by the fact that, in our definition of the semaphore type, a

`signal` operation may unblock an *arbitrary* element of `S.L`. Freedom from starvation does not occur if the definition of the semaphore type is changed so that `S.L` is a queue not a set. (See Section 6.8 for a comparison of semaphore definitions.)

In the exercises, you are asked to show that if the initial value of `S.V` is *k*, then at most *k* processes can be in the critical section at any time.

Order of Execution Problems

The critical section problem is an abstraction of synchronization problems that occur when several processes compete for the same resource. Synchronization problems are also common when processes must coordinate the *order of execution* of operations of different processes. For example, consider the problem of the concurrent execution of one step of the mergesort algorithm, where an array is divided into two halves, the two halves are sorted concurrently and the results are merged together. Thus, to mergesort the array `[5, 1, 10, 7, 4, 3, 12, 8]`, we divide it into two halves `[5, 1, 10, 7]` and `[4, 3, 12, 8]`, sort them obtaining `[1, 5, 7, 10]` and `[3, 4, 8, 12]`, respectively, and then merge the two sorted arrays to obtain `[1, 3, 4, 5, 7, 8, 10, 12]`.

To perform this algorithm concurrently, we will use three processes, two for sorting and one for merging. Clearly, the two `sort` processes work on totally independent data and need no synchronization, while the `merge` process must not execute its statements until the two other have completed. Here is a solution using two binary semaphores:

Table 6.5. Mergesort

```
integer array A
binary semaphore S1 ← (0, ∅)
binary semaphore S2 ← (0, ∅)
```

sort1	sort2	merge

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

The integer components of the semaphores are initialized to zero, so process `merge` is initially blocked on `S1`. Suppose that process `sort1` completes before process `sort2`; then `sort1` signals `S1` enabling `merge` to proceed, but it is then blocked on semaphore `S2`. Only when `sort2` completes will `merge` be unblocked and begin executing its algorithm. If `sort2` completes before `sort1`, it will execute `signal(S2)`, but `merge` will still be blocked on `S1` until `sort1` completes.

The Producer–Consumer Problem

The producer–consumer problem is an example of an order-of-execution problem. There are two types of processes in this problem:

Producers. A producer process executes a statement `produce` to create a data element and then sends this element to the consumer processes.

Consumers. Upon receipt of a data element from the producer processes, a consumer process executes a statement `consume` with the data element as a parameter.

As with the `critical section` and `non-critical section` statements, the `produce` and `consume` statements are assumed not to affect the additional variables used for synchronization and they can be omitted in abbreviated versions of the algorithms.

Producers and consumers are ubiquitous in computing systems:

Producer	Consumer
Communications line	Web browser
Web browser	Communications line
Keyboard	Operating system
Word processor	Printer
Joystick	Game program
Game program	Display screen
Pilot controls	Control program
Control program	Aircraft control surfaces

When a data element must be sent from one process to another, the communications can be *synchronous*, that is, communications cannot take place until both the producer and the consumer are ready to do so. Synchronous communication is discussed in detail in Chapter 8.

More common, however, is *asynchronous communications* in which the communications channel itself has some capacity for storing data elements. This store, which is a queue of data elements, is called a *buffer*. The producer executes an `append` operation to place a data element on the tail of the queue, and the consumer executes a `take` operation to remove a data element from the head of the queue.

The use of a buffer allows processes of similar average speeds to proceed smoothly in spite of differences in transient performance. It is important to understand that a buffer is of no use if the average speeds of the two processes are very different. For example, if your web browser continually produces and sends more data than your communications line can carry, the buffer will always be full and there is no advantage to using one. A buffer can also be used to resolve a clash in the structure of data. For example, when you download a compressed archive (a *zip* file), the files inside cannot be accessed until the entire archive has been read.

There are two synchronization issues that arise in the producer–consumer problem: first, a consumer cannot take a data element from an empty buffer, and second, since the size of any buffer is finite, a producer cannot append a data element to a full buffer. Obviously, there is no such thing as an infinite buffer, but if the buffer is very large compared to the rate at which data is produced, you might want to avoid the extra overhead required by a finite buffer algorithm and risk an occasional loss of data. Loss of data will occur when the producer tries to append a data element to a full buffer; either the operation will not succeed, in which case that element is lost, or it will overwrite one of the existing elements in the buffer (see Section 13.6).

Infinite Buffers

If there is an infinite buffer, there is only one interaction that must be synchronized: the consumer must not attempt a `take` operation from an empty buffer. This is an order-of-execution problem like the mergesort algorithm; the difference is that it happens repeatedly within a loop.

Table 6.6. producer–consumer (infinite buffer)

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

producer	consumer

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

Since the buffer is infinite, it is impossible to construct a finite state diagram for the algorithm. A *partial* state diagram for the abbreviated algorithm:

Table 6.7. producer–consumer (infinite buffer, abbreviated)

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

producer	consumer

```
      dataType d
      loop forever
p1:     append(d, buffer)
p2:     signal(notEmpty)
```

```
      dataType d
      loop forever
q1:     wait(notEmpty)
q2:     d ← take(buffer)
```

is shown in Figure 6.2. In the diagram, the value of the buffer is written with square brackets and a buffer element is denoted by *x*; the consumer process is denoted by *con*. The horizontal arrows indicate execution of operations by the producer, while the vertical arrows are for the consumer. Note that the left two states in the lower row have no arrows for the consumer because it is blocked.

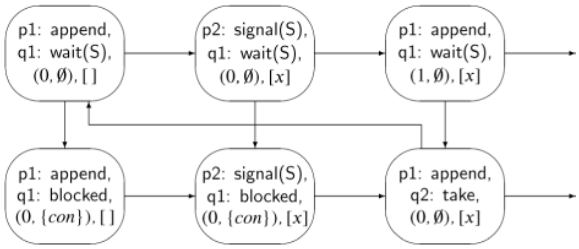


Figure 6.2. Partial state diagram for producer–consumer with infinite buffer

Assume now that the two statements of each process form one atomic statement each. (You are asked to remove this restriction in an exercise.) The following invariant holds:

notEmpty.V = *#buffer*,

where *#buffer* is the number of elements in `buffer`. The formula is true initially since there are no elements in the buffer and the value of the integer component of the semaphore is zero. Each execution of the statements of the producer appends an element to the buffer and also increments the value of *notEmpty.V*. Each execution of the statements of the consumer takes an element from the buffer and also decrements the value of *notEmpty.V*. From the invariant, it easily follows that the consumer does not remove an element from an empty buffer.

The algorithm is free from deadlock because as long as the producer continues to produce data elements, it will execute `signal(notEmpty)` operations and unblock the consumer. Since there is only one possible blocked process, the algorithm is also free from starvation.

Bounded Buffers

The algorithm for the producer–consumer problem with an infinite buffer can be easily extended to one with a finite buffer by noticing that the producer "takes" empty places from the buffer, just as the consumer takes data elements from the buffer (Algorithm 6.8). We can use a similar synchronization mechanism with a semaphore `notFull` that is initialized to *N*, the number of (initially empty) places in the finite buffer. We leave the proof of the correctness of this algorithm as an exercise.

Table 6.8. producer–consumer (finite buffer, semaphores)

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

producer

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

consumer

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

Split Semaphores

Algorithm 6.8 uses a technique called *split semaphores*. This is not a new semaphore type, but simply a term used to describe a synchronization mechanism built from semaphores. A split semaphore is a group of two or more semaphores satisfying an invariant that the sum of their values is at most equal to a fixed number *N*. In the case of the bounded buffer, the invariant:

notEmpty + *notFull* = *N*

holds at the beginning of the loop bodies.

In the case that *N* = 1, the construct is called a *split binary semaphore*. Split binary semaphores were used in Algorithm 6.5 for mergesort.

Compare the use of a split semaphore with the use of a semaphore to solve the critical section problem. Mutual exclusion was achieved by performing the `wait` and `signal` operations on a semaphore within a single process. In the bounded buffer algorithm, the `wait` and `signal` operations on each semaphore are in different processes. Split semaphores enable one process to wait for the completion of an event in another.

Definitions of Semaphores

There are several different definitions of the semaphore type and it is important to be able to distinguish between them. The differences have to do with the specification of liveness properties; they *do not* affect the safety properties that follow from the semaphore invariants, so the solution we gave for the critical-section problem satisfies the mutual exclusion requirement whatever definition we use.

Strong Semaphores

The semaphore we defined is called a *weak semaphore* and can be compared with a *strong semaphore*. The difference is that `S.L`, the set of processes blocked on the semaphore `S`, is replaced by a queue:

wait(S)

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

signal(S)

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

Recall (page 114) that we gave a scenario showing that starvation is possible in the solution for the critical section problem with more than two processes. For a strong semaphore, starvation is impossible for any number N of processes. Suppose `p` is blocked on S, that is, `p` appears in the queue `S.L`. There are at most $N-1$ processes in `S.L`, so there are at most $N-2$ processes ahead of `p` on the queue. After at most $N-2$ `signal(S)` operations, `p` will be at the head of the queue `S.L` so it will unblocked by the next `signal(S)` operation.

Busy-Wait Semaphores

A *busy-wait semaphore* does not have a component `S.L`, so we will identify `S` with `S.V`. The operations are defined as follows:

wait(S)

```
await S > 0
S ← S - 1
```

signal(S)

```
S ← S + 1
```

Semaphore operations are *atomic* so there is no interleaving between the two statements implementing the `wait(S)` operation.

With busy-wait semaphores you cannot ensure that a process enters its critical section even in the two-process solution, as shown by the following scenario:

n	Process p	Process q	S
1	**p1: wait(S)**	`q1: wait(S)`	1
2	`p2: signal(S)`	**q1: wait(S)**	0
3	**p2: signal(S)**	`q1: wait(S)`	0
4	**p1: wait(S)**	`q1: wait(S)`	1

Line 4 is identical to line 1 so these states can repeat indefinitely. The scenario is fair, in the technical sense of Section 2.7, because process `q` is selected infinitely often in the interleaving, but because it is always selected when $S = 0$, it never gets a chance to successfully complete the `wait(S)` operation.

Busy-wait semaphores are appropriate in a multiprocessor system where the waiting process has its own processor and is not wasting CPU time that could be used for other computation. They would also be appropriate in a system with little contention so that the waiting process would not waste too much CPU time.

Abstract Definitions of Semaphores[A]

The definitions given above are intended to be behavioral and not to specify data structures for implementation. The set of the weak semaphore is a mathematical entity and is only intended to mean that the unblocked process is chosen arbitrarily. The queue of the strong semaphore is intended to mean that processes are unblocked in the order they were blocked (although it can be difficult to specify what order means in a concurrent system [37]). A busy-wait semaphore simply allows interleaving of other processes between the signal and the unblocking of a blocked process. How these three behaviors are implemented is unimportant.

It is possible to give abstract definitions of *strongly fair* and *weakly fair* semaphores, similar to the definitions of scheduling fairness given in Section 2.7 (see [58, Section 10.1]). However, these definitions are not very useful, and any real implementation will almost certainly provide one of the behaviors we have given.

The Problem of the Dining Philosophers

The problem of the dining philosophers is a classical problem in the field of concurrent programming. It offers an entertaining vehicle for comparing various formalisms for writing and proving concurrent programs, because it is sufficiently simple to be tractable yet subtle enough to be challenging.

The problem is set in a secluded community of five philosophers who engage in only two activities—*thinking* and *eating*:

Table 6.9. Dining philosophers (outline)

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

Meals are taken communally at a table set with five plates and five forks (Figure 6.3). In the center of the table is a bowl of spaghetti that is endlessly replenished. Unfortunately, the spaghetti is hopelessly tangled and a philosopher needs two forks in order to eat.[2] Each philosopher may pick up the forks on his left and right, but only one at a time. The problem is to design pre- and postprotocols to ensure that a philosopher only eats if she has two forks. The solution should also satisfy the correctness properties that we described in the chapter on mutual exclusion.

The correctness properties are:

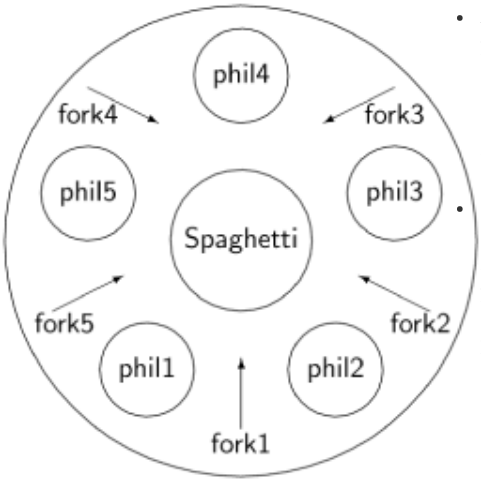


Figure 6.3. The dining philosophers

- A philosopher eats only if she has two forks.
- Mutual exclusion: no two philosophers may hold the same fork simultaneously.
- Freedom from deadlock.
- Freedom from starvation (pun!).
- Efficient behavior in the absence of contention.

Here is a first attempt at a solution, where we assume that each philosopher is initialized with its index *i*, and that addition is implicitly modulo 5.

Table 6.10. Dining philosophers (first attempt)

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

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

Each fork is modeled as a semaphore: `wait` corresponds to taking a fork and `signal` corresponds to putting down a fork. Clearly, a philosopher holds both forks before eating.

Example 6.3. Theorem

No fork is ever held by two philosophers

Proof: Let $\#P_i$ be the number of philosophers holding fork *i*, so that $\#P_i = \#wait(fork[\,i]) - \#signal(fork[\,i])$. By the semaphore invariant (6.2), $fork[\,i] = 1 + (-\#P_i)$, or $\#P_i = 1 - fork[\,i]$. Since the value of a semaphore is non-negative (6.1), we conclude that $\#P_i \le 1$.

Unfortunately, this solution deadlocks under an interleaving that has all philosophers pick up their left forks—execute `wait(fork[i])`—before any of them tries to pick up a right fork.

Now they are all waiting on their right forks, but no process will ever execute a signal operation.

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

Table 6.11. Dining philosophers (second attempt)

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

```
    loop forever
p1:   think
p2:   wait(room)
p3:   wait(fork[i])
p4:   wait(fork[i+1])
p5:   eat
p6:   signal(fork[i])
p7:   signal(fork[i+1])
p8:   signal(room)
```

The addition of the semaphore `room` obviously does not affect the correctness of the safety properties we have proved.

Example 6.4. Theorem

The algorithm is free from starvation.

Proof: We must assume that the semaphore `room` is a blocked-queue semaphore so that any philosopher waiting to enter the room will eventually do so. The `fork` semaphores need only be blocked-set semaphores since only two philosophers use each one. If philosopher *i* is starved, she is blocked forever on a semaphore. There are three cases depending on whether she is blocked on `fork[i]`, `fork[i+1]` or `room`.

Case 1: Philosopher *i* is blocked on her left fork. Then philosopher $i - 1$ holds `fork[i]` as her right fork, that is, philosopher $i - 1$ has successfully executed both her `wait` statements and is either eating, or about to signal her left or right fork semaphores. By progress and the fact that there are only two processes executing operations on a fork semaphore, eventually she will release philosopher *i*.

Case 2: Philosopher *i* is blocked on her right fork. This means that philosopher $i + 1$ has successfully taken her left fork (`fork[i+1]`) and will never release it. By progress of eating and signaling, philosopher $i + 1$ must be blocked forever on her right fork. By induction, it follows that if *i* is blocked on her right fork, then so must all the other philosophers: $i + j$, $1 \le j \le 4$. However, by the semaphore invariant on `room`, for some *j*, philosopher $i + j$ is not in the room, thus obviously not blocked on a fork semaphore.

Case 3: We assume that the `room` semaphore is a blocked-queue semaphore, so philosopher *i* can be blocked on `room` only if its value is zero indefinitely. By the semaphore invariant, there are at most four philosophers executing the statements between `wait(room)` and `signal(room)`. The previous cases showed that these philosophers are never blocked indefinitely, so one of them will eventually execute `signal(room)`.

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

Table 6.12. Dining philosophers (third attempt)

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

philosopher 4

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

Again it is obvious that the correctness properties on eating are satisfied. Proofs of freedom from deadlock and freedom from starvation are similar to those of the previous algorithm and are left as an exercise.

A solution to the dining philosophers problem by Lehmann and Rabin uses random numbers. Each philosopher "flips a coin" to decide whether to take her left or right fork first. It can be shown that with probability one no philosopher starves [48, Section 11.4].

Barz's Simulation of General Semaphores[A]

In this section we present Hans W. Barz's simulation of a general semaphore by a pair of binary semaphores and an integer variable [

6

]. (See [67] for a survey of the attempts to solve this problem.)

The simulation will be presented within the

Table 6.13. Barz's algorithm for simulating general semaphores

```
binary semaphore S ← 1
binary semaphore gate ← 1
integer count ← k
```

```
    loop forever
      non-critical section

p1:   wait(gate)                  // Simulated wait
p2:   wait(S)
p3:   count ← count - 1
p4:   if count > 0 then
p5:      signal(gate)
p6:   signal(S)

      critical section

p7:   wait(S)                     // Simulated signal
p8:   count ← count + 1
```

```
p9:   if count = 1 then
p10:     signal(gate)
p11:  signal(S)
```

context of a solution of the critical section problem that allows $k > 0$ processes simultaneously in the critical section (Algorithm 6.13).

k is the initial value of the general semaphore, statements `p1..6` simulate the `wait` statement, and statements `p7..11` simulate the `signal` statement. The binary semaphore `gate` is used to block and unblock processes, while the variable `count` holds the value of the integer component of the simulated general semaphore. The second binary semaphore `S` is used to ensure mutual exclusion when accessing `count`. Since we will be proving only the safety property that mutual exclusion holds, it will be convenient to consider the binary semaphores as busy-wait semaphores, and to write the value of the integer component of the semaphore `gate` as *gate*, rather than *gate.V*.

It is clear from the structure of the algorithm that `S` is a binary semaphore; it is less clear that `gate` is one. Therefore, we will not assume anything about the value of `gate` other than that it is an integer and prove (Lemma 6.5(6), below) that it only takes on the values 0 and 1.

Let us start with an informal description of the algorithm. Since `gate` is initialized to 1, the first process attempting to execute a simulated `wait` statement will succeed in passing `p1: wait(gate)`, but additional processes will block. The first process and each subsequent process, up to a total of $k - 1$, will execute `p5: signal(gate)`, releasing additional processes to successfully complete `p1: wait(gate)`. The `if` statement at `p4` prevents the *k* th process from executing `p5: signal(gate)`, so further processes will be blocked at `p1: wait(gate)`.

When *count* = 0, a single simulated `signal` operation will increment `count` and execute `p11: signal(gate)`, unblocking one of the processes blocked at `p1: wait(gate)`; this process will promptly decrement `count` back to zero. A sequence of simulated `signal` operations, however, will cause `count` to have a positive value, although the value of *gate* remains 1 since it is only signaled once. Once `count` has a positive value, one or more processes can now successfully execute the simulated `wait`.

An inductive proof of the algorithm is quite complex, but it is worth studying because so many incorrect algorithms have been proposed for this problem.

In the proof, we will reduce the number of steps in the induction to three. The binary semaphore `S` prevents the statements `p2..6` from interleaving with the statements `p7..11`. `p1` can interleave with statements `p2..6` or `p7..11`, but cannot affect their execution, so the effect is the same as if *all* the statements `p2..6` or `p7..11` were executed before `p1`.[3]

We will use the notation *entering* for *p*2..6 and *inCS* for *p*7. We also denote by *#entering*, respectively *#inCS*, the number of processes for which *entering*, respectively *inCS*, is true.

Example 6.5. Lemma

The conjunction of the following formulas is invariant.

(1)	$entering \rightarrow (gate = 0)$,
(2)	$entering \rightarrow (count > 0)$,
(3)	$\#entering \leq 1$,
(4)	$((gate = 0) \wedge \neg\, entering) \rightarrow (count = 0)$,
(5)	$(count \leq 0) \rightarrow (gate = 0)$,
(6)	$(gate = 0) \vee (gate = 1)$.

Proof: The phrase "by (n), *A* holds" will mean: by the inductive hypothesis on formula (n), *A* must be true before executing this statement. The presentation is somewhat terse, so before reading further, make sure that you understand how material implications can be falsified (Appendix B.3).

Initially:

1. All processes start at $p1$, so the antecedent is false.
2. As for (1).
3. $\#entering = 0$.
4. $gate = 1$ so the antecedent is false.
5. $count = k > 0$ so the antecedent is false.
6. $gate = 1$ so the formula is true.

Executing p1:

1. By (6), $gate \leq 1$, so `p1: wait(gate)` can successfully execute only if $gate = 1$, making the consequent $gate = 0$ true.
2. *entering* becomes true, so the formula can be falsified only if the consequent is false because $count \leq 0$. But `p1` does not change the value of `count`, so we must assume $count \leq 0$ before executing the statement. By (5), $gate = 0$, so the `p1: wait(gate)` cannot be executed.
3. This can be falsified only if *entering* is true before executing `p1`. By (1), if *entering* is true, then $gate = 0$, so the `p1: wait(gate)` cannot be executed.
4. *entering* becomes true, so the antecedent $\neg$ *entering* becomes false.
5. As for (1).
6. As for (1).

Executing p2..6:

1. Some process must be at `p2` to execute this statement, so *entering* is true, and by (3), *#entering* = 1. Therefore, the antecedent *entering* becomes false.

2. As for (1).

3. As for (1).

4. By (1), *gate* = 0, and by (2), *count* > 0. If *count* = 1, the consequent *count* = 0 becomes true. If *count* > 1, `p3`, `p4` and `p5` will be executed, so that *gate* becomes 1, falsifying the antecedent.

5. By (1), *gate* = 0, and by (2), *count* > 0. If *count* > 1, after decrementing its value in `p3`, *count* > 0 will become true, falsifying the antecedent. If *count* = 1, the antecedent becomes true, but `p5` will not be executed, so *gate* = 0 remains true.

6. By (1), *gate* = 0 and it can be incremented only once, by `p5`.

Executing p7..11:

1. The value of *entering* does not change. If it was true, by (2) *count* > 0, so that *count* becomes greater than 1 by `p8`, ensuring by the `if` statement at `p9` that the value of *gate* does not change in `p10`.

2. The value of *entering* does not change, and the value of *count* can only increase.

3. Trivially, the value of *#entering* is not changed.

4. Suppose that the consequent *count* = 0 were true before executing the statements and that it becomes false. By the `if` statement at `p9`, statement `p10: signal(gate)` is executed, so the antecedent *gate* = 0 is also falsified. Suppose now that both the antecedent and the consequent were false; then the antecedent cannot become true, because the value of *entering* does not change, and if *gate* = 0 is false, it certainly cannot become true by executing `p10: signal(gate)`.

5. The consequent can be falsified only if *gate* = 0 were true before executing the statements and `p10` was executed. By the `if` statement at `p9`, that can happen only if *count* = 0. Then *count* = 1 after executing the statements, falsifying the antecedent. If the antecedent were false so that *count* > 0, it certainly remains false after incrementing `count` in `p8`.

6. The formula can be falsified only if *gate* = 1 before executing the statements and `p10: signal(gate)` is executed. But that happens only if *count* = 0, which implies *gate* = 0 by (5).

Example 6.6. Lemma

The formula *count* = *k* − *#inCS* is invariant

Proof: The formula is initially true by the initialization of `count` and the fact that all processes are initially at their non-critical sections. Executing `p1` does not change any term in the formula. Executing `p2..6` increments *#inCS* and decrements *count*, preserving the invariant, as does executing `p7..11`, which decrements *#inCS* and increments *count*.

Example 6.7. Theorem

Mutual exclusion holds for Algorithm 6.13, that is, *#inCS* ≤ *k* is invariant

Proof: Initially, *#inCS* ≤ *k* is true since *k* > 0. The only step that can falsify the formula is `p2..6` executed in a state in which *#inCS* = *k*. By Lemma 6.6, in this state *count* = 0, but *entering* is also true in that state, contradicting (2) of Lemma 6.5.

The implementation of Barz's algorithm in Promela is discussed in Section 6.15.

Udding's Starvation-Free Algorithm[A]

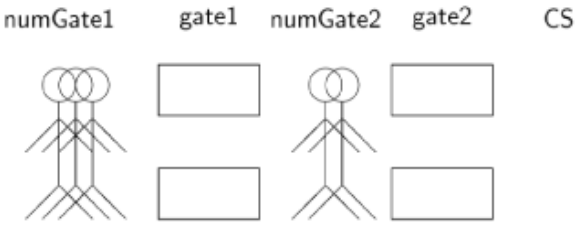
There are solutions to the critical-section problem using weak semaphores that are free of starvation. We give without proof a solution developed by Jan T. Udding [68]. Algorithm 6.14 is similar to the fast algorithms described in Section 5.4, with two *gates* that must be passed in order to enter the critical section.

Table 6.14. Udding's starvation-free algorithm

```
semaphore gate1 ← 1, gate2 ← 0
integer numGate1 ← 0, numGate2 ← 0
```

```
loop forever
  non-critical section

p1:   wait(gate1)
p2:   numGate1 ← numGate1 + 1
p3:   signal(gate1)

p4:   wait(gate1)              // First gate
p5:   numGate2 ← numGate2 + 1
p6:   if numGate1 > 0
p7:      signal(gate1)
p8:   else signal(gate2)

p9:   wait(gate2)              // Second gate
p10:  numGate2 ← numGate2 - 1

      critical section
p11:  if numGate2 > 0
p12:     signal(gate2)
p13:  else signal(gate1)
```

The two semaphores constitute a split binary semaphore satisfying 0 ≤ *gate1* + *gate2* ≤ 1, and the integer variables count the number of processes about to enter or trying to enter each gate.



The semaphore `gate1` is reused to provided mutual exclusion during the initial increment of `numGate1`. In our model, where each line is an atomic operation, this protection is not needed, but freedom from starvation depends on it so we have included it.

The idea behind the algorithm is that processes successfully pass through a gate (complete the `wait` operation) as a group, without interleaving `wait` operations from the other processes. Thus a process `p` passing through `gate2` into the critical section may quickly rejoin the group of processes in front of `gate1`, but all of the colleagues from its group will enter the critical section before `p` passes the first gate again. This ensures that the algorithm is free from starvation.

The attempted solution still suffers from the possibility of starvation, as shown by the following outline of a scenario:

n	Process p	Process q	gate1	gate2	n
1	**p4: wait(g1)**	q4: wait(g1)	1	0	2
2	**p9: wait(g2)**	q9: wait(g2)	0	1	0
3	**CS**	q9: wait(g2)	0	0	0
4	**p12: signal(g2)**	q9: wait(g2)	0	0	0
5	p1: wait(g1)	**CS**	0	0	0
6	**p1: wait(g1)**	q13: signal(g1)	0	0	0
7	p1: blocked	**q13: signal(g1)**	0	0	0
8	p4: wait(g1)	**q1: wait(g1)**	1	0	1
9	**p4: wait(g1)**	q4: wait(g1)	1	0	2

Two processes are denoted `p` and `q` with appropriate line numbers; the third process `r` is omitted because it remains blocked at `r1: wait(g1)` throughout the scenario. Variable names have been shortened: `gate` to `g` and `numGate` to `nGate`.

The scenario starts with processes `p` and `q` at gates `p4` and `q4`, respectively, while the third process `r` has yet to pass the initial `wait` operation at `r1`. The key line in the scenario is line 7, where `q` unblocks his old colleague `p` waiting again at `p1` instead of `r` who has been waiting at `r1` for quite a long time.

By adding an additional semaphore, we can ensure that at most one process is at the second gate at any one time. The semaphore `onlyOne` is initialized to `1`, a `wait(onlyOne)` operation is added before `p4` and a `signal(onlyOne)` operation is added before `p9`. This ensures that at most one process will be blocked at `gate1`.

Semaphores in BACI[L]

Types `semaphore` and `binarysem` with the operations **`wait`** and **`signal`** are defined in both the C and Pascal dialects of BACI. The implementation is of weak semaphores: when a **`signal`** statement is executed, the list of processes blocked on this semaphore is searched starting from a random position in the list.

Semaphores in Ada[L]

The Ada language does not include a semaphore type, but it is easily implemented using protected types:[4]

```
1  protected type Semaphore(Initial : Natural) is
2    entry Wait;
3    procedure Signal;
4  private
5    Count: Natural := Initial;
6  end Semaphore;
7
8  protected body Semaphore is
9    entry Wait when Count > 0 is
10   begin
11     Count := Count - 1;
12   end Wait;
13
14   procedure Signal is
15   begin
16     Count := Count + 1;
17     end Signal;
18 end Semaphore;
```

The private variable `Count` is used to store the integer component of the semaphore. The queue of the entry `Wait` is the queue of the processes blocked on the semaphore, and its barrier **`when`** `Count > 0` ensures that the wait operation can be executed only if the value of `Count` is positive. Since entry queues in Ada are defined to be FIFO queues, this code implements a strong semaphore.

To use this semaphore, declare a variable with a value for the discriminant `Initial` and then invoke the protected operations:

```
S: Semaphore(1);
. . .
S.wait;
-- critical section
S.signal;
```

Semaphores in Java[L]

The `Semaphore` class is defined within the package `java.util.concurrent`. The terminology is slightly different from the one we have been using. The integer component of an object of class `Semaphore` is called the number of *permits* of the object, and unusually, it can be initialized to be negative. The `wait` and `signal`

operations are called `acquire` and `release`, respectively. The `acquire` operation can throw the exception `InterruptedException` which must be caught or thrown.

Here is a Java program for the counting algorithm with a semaphore added to ensure that the two assignment statements are executed with no interleaving.

```
1  import java.util . concurrent. Semaphore;
2  class CountSem extends Thread {
3    static volatile int n = 0;
4    static Semaphore s = new Semaphore(1);
5
6    public void run() {
7      int temp;
8      for (int i = 0; i < 10; i ++) {
9        try {
10         s. acquire ();
11       }
12       catch (InterruptedException e) {}
13       temp = n;
14       n = temp + 1;
15       s. release ();
16     }
17   }
18
19   public static void main(String[] args) {
20       // (as before)
21   }
22 }
```

The constructor for class `Semaphore` includes an extra boolean parameter used to specify if the semaphore is fair or not. A fair semaphore is what we have called a strong semaphore, but an unfair semaphore is more like a busy-wait semaphore than a weak semaphore, because the `release` method simply unblocks a process but does not acquire the lock associated with the semaphore. (See Section 7.11 for more details on synchronization in Java.)

Semaphores in Promela^L

In Promela it is easy to implement a busy-wait semaphore:

```
#define wait(s) atomic {s > 0; s-- }
#define signal(s) s++
```

Statements within **`atomic`** are executed without interleaving, except that if a statement blocks, statements from other processes will be executed. In `wait(s)`, either the two instructions will be executed as one atomic statement, or the process will block because $s = 0$; when it can finally be executed because $s > 0$, the two statements will be executed atomically.

To implement a weak semaphore, we must explicitly store `S.L`, the set of blocked processes. We declare a semaphore type definition containing an integer field for the `V` component of the semaphore and a boolean array field for the `L` component with an element for each of `NPROCS` processes:

```
typedef Semaphore {
   byte count;
   bool blocked[NPROCS];
};
```

Inline definitions are used for the semaphore operations, beginning with an operation to initialize the semaphore:

```
inline initSem(S, n) { S.count = n }
```

If the count is positive, the wait operation decrements the count; otherwise, the process is blocked waiting for this variable to become false:

```
inline wait(S) {
    atomic {
      if
      :: S.count >= 1 -> S.count--
      :: else ->
           S.blocked[ _pid-1] = true;
           ! S.blocked[ _pid-1]
      fi
    }
}
```

The semaphore is initialized in the **`init`** process whose `_pid` is zero, so the other processes have `_pid`s starting from one.

The signal operation nondeterministically sets one of the blocked processes to be unblocked. If there are no blocked processes, the count is incremented. The following code assumes that there are three processes:

```
inline signal (S) {
    atomic {
      if
      :: S.blocked[0] -> S.blocked[0] = false
      :: S.blocked[1] -> S.blocked[1] = false
      :: S.blocked[2] -> S.blocked[2] = false
      :: else -> S.count++
      fi
    }
}
```

The code for an arbitrary number of processes is more complicated:

```
inline signal (S) {
    atomic {
      S.i = 0;
      S.choice = 255;
      do
      :: (S.i == NPROCS) -> break
      :: (S.i < NPROCS) && !S.blocked[S.i] -> S.i++
      :: else ->
          if
          :: (S.choice == 255) -> S.choice = S.i
          :: (S.choice != 255) -> S.choice = S.i
          :: (S.choice != 255) ->
          fi;
          S.i ++
      od;
      if
      :: S.choice == 255 -> S.count++
      :: else -> S.blocked[S.choice] = false
      fi
    }
}
```

Two additional variables are added to the **typedef**: `i` to loop through the array of blocked processes, and `choice` to record the choice of a process to unblock, where `255` indicates that a process has yet to be chosen. If some process is blocked—indicated by `S.blocked[ S.i ]` being true—we nondeterministically choose either to select it by assigning its index to `S.choice` or to ignore it.

The implementation of strong semaphores using channels is left as an exercise. We also present an implementation of weak semaphores using channels, that is not restricted to a fixed number of processes.

Proving Barz's Algorithm in Spin[A]

A Promela program for Algorithm 6.13, Barz's algorithm, is shown in Listing 6.1. The semaphore `gate` is implemented as a general semaphore; then we *prove* that it is a binary semaphore by defining

a symbol `bingate` as `(gate <= 1)` and proving the temporal logic formula `[]bingate`. Mutual exclusion is proved by incrementing and decrementing the variable `critical` and checking `assert (critical<=K)`.

It is more difficult to check the invariants of Lemma 6.5. The reason is that some of them are not true at every statement, for example, between `p8: count←count+1` and `p10: signalB(gate)`. Promela enables us to do what we did in the proof: consider all the statements `p2..6` and `p7..11` as single statements. **d_step**, short for *deterministic step*, specifies that the included statements are to be executed as a single statement. **d_step** differs from **atomic** in that it cannot contain any statement that blocks; this is true in Listing 6.1 because within the scope of the **d_steps** there are no blocking expressions like `gate>0`, and because the **`if`** statements include an **`else`** alternative to ensure that they don't block.

Transition

Semaphores are an elegant and efficient construct for solving problems in concurrent programming. Semaphores are a popular construct, implemented in many systems and easily simulated in others. However, before using semaphores you must find out the precise definition that has been implemented, because the liveness of a program can depend on the specific semaphore semantics.

Despite their popularity, semaphores are usually relegated to low-level programming because they are not structured. Variants of the monitor construct described in the next chapter are widely used because they enable encapsulation of synchronization.

Example 6.1. Barz's algorithm in Promela

```
#define NPROCS 3
#define K     2
byte gate = 1;
int count = K;
byte critical = 0;
active [ NPROCS] proctype P () {
  do ::
    atomic { gate > 0; gate--; }
    d_step {
      count--;
      if
      :: count > 0 -> gate++
      :: else
      fi
    }
    critical ++;
    assert (critical <= 1);
    critical --;
    d_step {
      count++;
      if
      :: count == 1 -> gate++
      :: else
      fi
    }
  od
}
```

Exercises

1. Consider the following algorithm:

   Table 6.15. Semaphore algorithm A

`semaphore S ← 1, T ← 0`

p	q
p1: wait(S) p2: write("p") p3: signal(T)	q1: wai q2: wri q3: sig

What are the possible outputs of this algorithm?

2. What are the possible outputs if we erase the state

3. What are the possible outputs if we erase the state

2. What are the possible outputs of the following algorithm

Table 6.16. Semaphore algorithm B

```
semaphore S1 ← 0, S2 ← 0
```

p	q
p1: write("p") p2: signal(S1) p3: signal(S2)	q1: wait(S1) q2: write("q") q3:

3. What are the possible outputs of the following algorithm

Table 6.17. Semaphore algorithm with a loop

```
semaphore S ← 1
boolean B ← false
```

p	q
p1: wait(S) p2: B ← true p3: signal(S) p4:	q1: wait q2: whi q3: writ q4: sig

4. Show that if the initial value of `S.v` in Algorithm 6.3 is *k*, the critical section at any time.

5. The following algorithm attempts to use binary semapho problem with at most k out of N processes in the critical

Table 6.18. Critical section problem (k out of N proces

```
binary semaphore S ← 1, delay ← 0
integer count ← k
```

```
    integer m
    loop forever
p1:   non-critical section
p2:   wait(S)
p3:   count ← count - 1
p4:   m ← count
p5:   signal(S)
p6:   if m ≤ -1
p7:      wait(delay)
p8:   critical section
p9:   wait(S)
p10:  count ← count + 1
p11:  if count ≤ 0
p12:     signal(delay)
p13:  signal(S)
```

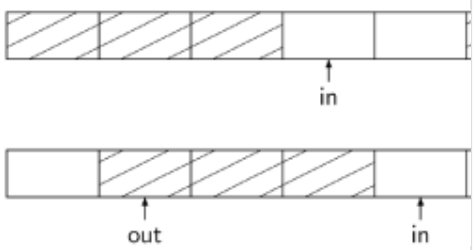
For $N = 4$ and $k = 2$, construct a scenario in which *delay* = a binary semaphore. Show this also for $N = 3$ and $k = 2$.

6. Modify Algorithm 6.5 to use a single general semaphore.

7. Prove the correctness of Algorithm 6.7 without the assur

8. Develop an abbreviated version of Algorithm 6.8. Assum statements of each process, show that *notFull*.$V = N - \#B$ correctness property can you conclude from this invaria

9. A bounded buffer is frequently implemented using a circ that is indexed modulo its length:



One variable, `in`, contains the index of the first empty sp index of the first full space. If *in* > *out*, there is data in `bu` there is data in `buffer[out..N]` and `buffer[0..in-1`

empty. Prove the correctness of the following algorithm
problem with a circular buffer:

Table 6.19. producer–consumer (circular buffer)

```
dataType array[0..N] buf
integer in, out ← 0
semaphore notEmpty ← (0
semaphore notFull ← (N,
```

producer	consumer

`dataType d`
`loop forever`
`p1: d ← produce`
`p2: wait(notFull)`
`p3: buffer[in] ← d`
`p4: in ← (in+1) modulo N`
`p5: signal(notEmpty)`
	`dat`
	`loo`
	`q1: w`
	`q2: d`
	`q3: o`
	`q4: s`
	`q5: c`

10. Prove the correctness of the asymmetric solution to the c
(Algorithm 6.12).

11. A partial scenario for starvation in Udding's algorithm w
complete the scenario.

12. Prove or disprove the correctness of Algorithm 6.20 for i
semaphore by binary semaphores.

13. Implement a strong semaphore in Promela.

Table 6.20. Simulating general semaphores

```
binary semaphore S ← 1, gate
integer count ← 0
```

- **wait**

```
p1:  wait(S)
p2:  count ← count - 1
p3:  if count < 0
p4:     signal(S)
p5:     wait(gate)
p6:  else signal(S)
```

- **signal**

```
p7:  wait(S)
p8:  count ← count + 1
p9:  if count ≤ 0
p10:    signal(gate)
p11: signal(S)
```

14. Here is a version of the implementation of the signal ope
Promela that does not use the variable `choice`. Is the pr

```
inline signal (S) {
    atomic {
      S.i = 0;
      do
      :: S.i == NPROCS -> break
      :: (S.i < NPROCS) && S.blocked[S.i] -> break
      :: (S.i < NPROCS) && S.blocked[S.i] -> S.i++
      :: (S.i < NPROCS) && !S.blocked[S.i] -> S.i++
      od;
      if
      :: S.i == NPROCS -> S.count++
      :: else -> S.blocked[S.i] = false
      fi
    }
}
```

15. Weak semaphores can be implemented in Promela by st
a channel:

```
chan ch = [NPROCS] of { pid };
```

Explain the following implementations of the `wait` and

```
inline wait(S) {
   atomic {
     if :: S.count >= 1 -> S.count--;
        :: else ->
             S.ch ! _pid;
             !(S.ch ?? [eval(_pid)])
     fi
  }
}
```

```
inline signal (S) {
   atomic {
     S.i = len(S.ch);
     if :: S.i == 0 -> S.count++
        :: else ->
           do :: S.i == 1 -> S.ch ? _; break
              :: else ->
                   S.i --;
                   S.ch ? S.temp;
                   if
                   :: break
                   :: S.ch ! S.temp
                   fi
           od
     fi
  }
}
```

16. [2 , Section 4.4][6] Algorithm 6.21 is a solution of the pro

writers using semaphores. `SignalProcess` is a subprog
each of the start and end operations.

1. Prove that `entry`, `readerSem` and `writerSem` forn
   that is, prove that

   $0 \leq entry + readerSem + writerSem \leq 1$.

   Prove the correctness of the algorithm by showing

   $(writers > 0) \rightarrow (writers = 1) \wedge (readers = 0)$.

2. The subprogram `SignalProcess` is uniform for al
   algorithm by writing separate code for each of the

3. Show that readers are given precedence over write

4. Modify the algorithm to give writers precedence ov

5. Modify the algorithm to give alternate precedence

Table 6.21. Readers and writers with semaphores

```
semaphore readerSem ← 0,
integer delayedReaders ←
semaphore entry ← 1
integer readers ← 0, wri
```

SignalProcess

```
if writers = 0 or delayedReaders > 0
   delayedReaders ← delayedReaders - 1
   signal(readerSem)
else if readers = 0 and writers = 0 and delay
   delayedWriters ← delayedWriters - 1
   signal(writerSem)
else signal(entry)
```

- **StartRead**

```
p1:  wait(entry)
p2:  if writers > 0
p3:     delayedReaders ← delayedReaders + 1
p4:     signal(entry)
p5:     wait(readerSem)
p6:  readers ← readers + 1
p7:  SignalProcess
```

- **EndRead**

```
p8:  wait(entry)
p9:  readers ← readers - 1
p10: SignalProcess
```

- **StartWrite**

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

- **EndWrite**

```
p18: wait(entry)
p19: writers ← writers - 1
p20: SignalProcess
```

[1] The original notation is `P(S)` for `Wait(S)` and `V(S)` for `signal(S)`, the letters `P` and `V` taken by Dijkstra from words in Dutch. This notation is still used in research papers and some systems.

[2] It is part of the folklore of computer science to point out that the story would be more believable if the bowl contained rice and the utensils were chopsticks.

[3] Formally, wait statements are *right movers* [46].

[4] Protected types and objects are explained in Sections 7.10–7.10.

[5] My thanks to Gerard Holzmann for his assistance in developing this implementation.

[6] This exercise should be attempted only after studying the description of the problem and monitor solution in Section 7.6.

PREV
5. Advanced Algorithms for the Critical …

NEXT
7. Monitors