Concurrent Programming Notes

v0.024 – January 20, 2025

Eduardo Bonelli

Preface

These course notes provide supporting material for CS511. They are currently under construction.

Contents

Part I

Shared Memory

Chapter 1

Shared Memory Model and Transition Systems

This chapter presents the Shared Memory Model. Threads communicate with each other by sharing some part of the memory. One example is when threads share a variable. Another is when they share an object in the heap.

1.1 Shared Memory Model

We begin with an example of a program in Groovy. The aim of this section is to introduce both the Groovy syntax and, in particular, how it allows us to succinctly model threads.

```
int x = 0

Thread.start { //P
   x = 1
}

Thread.start { //Q
   x = 2
}
```

ex1.groovy

This program declares a shared variable `x`, sets it to `0` and then spawns two threads. The first thread sets `x` to 1 and the second to `2`. After this program terminates, the value of `x` may either be 1 or 2. The variable `x` is said to be shared in the sense that it is visible to (or its scope includes) both threads[1].

Assuming this program is stored in a file called `ex1.groovy`, it may be executed using the terminal as follows:

[1]From the point of view of Groovy, `x` is a local variable that is declared in the `run` method that the Groovy compiler will generate. It will not be visible outside of the script main body. In Groovy, global variables are declared by omitting the type annotation. For our purposes, whether such variables are declared with or without a type annotation, makes no difference.

```bash
$ groovy ex1
$
```

bash

Since our program contains no output statements, there is no visible effect from its execution. The following example, waits for P and Q to terminate using the built-in method `join` and then prints the value of `x`:

```groovy
int x = 0

P = Thread.start { //P
      x = 1
}

Q = Thread.start { //Q
      x = 2
}

P.join()  // Wait for P to terminate
Q.join()  // Wait for Q to terminate
println x
```

ex2.groovy

Assuming this program is stored in a file called `ex2.groovy`, it may be executed using the terminal as follows:

```bash
$ groovy ex2
2
$
```

bash

Repeated execution will most likely produce 2 since P is spawned before Q and runs immediately. It is entirely possible, however, to obtain 1 as a result.

> We shall not compile Groovy code; rather we shall execute Groovy programs by using the groovy interpreter, as exemplified above. One may, however, compile groovy programs to produce bytecode. For example, `groovyc ex1.groovy` will produce a series of `.class` files. A groovy program is implemented as Java subclass of a built in class called `groovy.lang.Script`.

The following example is a Groovy program that prints characters.

```groovy
Thread.start { //P
   print "A"
   print "B"
}

Thread.start { //Q
   print "C"
}
```

What are the possible outputs one may obtain from executing it? It can print three possible sequences of characters, namely `ABC`, `ACB`, `CAB`. What about the following program?

```
Thread.start { //P
    print "A"
    print "B"
}

Thread.start { //Q
    print "C"
    print "D"
}
```

Clearly the number of possible executions, also called interleavings, grows exponentially with the number of instructions in each thread. Indeed, if P has m instructions and Q has n instructions, the number of interleavings is:

$$\binom{m+n}{m} = \frac{(m+n)!}{m!n!}$$

This makes it difficult to reason about concurrent programs: there are simply too many interleavings to consider; we never know whether one such interleaving might lead our code to produce an unwanted result.

Consider the following program:

```
int x=0

P = Thread.start {
      x = x+1
}
Q = Thread.start {
      x = x+1
}

P.join() // wait for P to terminate
Q.join() // wait for Q to terminate
println x
```

Its execution can produce 1 as output!

```
$ groovy ex1
1
$
```

bash

How is that possible? We clearly need a detailed model of what it means to execute a concurrent program.

1.2 Transition Systems

This section introduces transition systems, a formalism we use to model the run-time behavior of concurrent programs. After defining transition systems, we illustrate how to associate a transition system to Groovy programs. By doing so, we assign "meaning" to our concurrent programs. It should be mentioned that we will associate transition systems only to a small subset of Groovy programs, not arbitrary ones.

A **Transition System** $\mathcal{A}$ is a tuple $(S, \to, I)$ where

- S is a set of states;
- $\rightarrow \subseteq S \times S$ is a transition relation; and
- $I \subseteq S$ is a set of initial states.

We say that $\mathcal{A}$ is finite if S is finite. Also, we write $s \rightarrow s'$ for $(s, s') \in \rightarrow$.

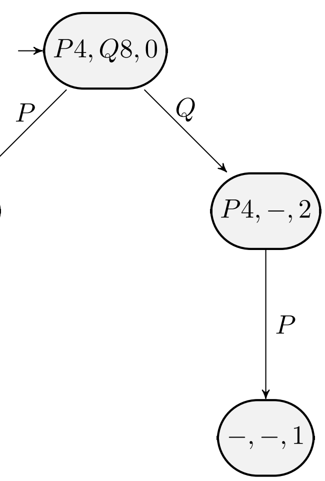
We illustrate, in this first example, how to model the runtime execution of Example 1.1 on page 3, repeated below:

```
int x = 0

Thread.start { //P
   x = 1
}

Thread.start { //Q
   x = 2
}
```

The states of our transition system will consist of 3-tuples containing the instruction pointer for `P`, the instruction pointer for `Q` and the value of `x`. The initial state is signalled with a small arrow. The hyphen indicates that there are no further instructions to be executed by that thread.



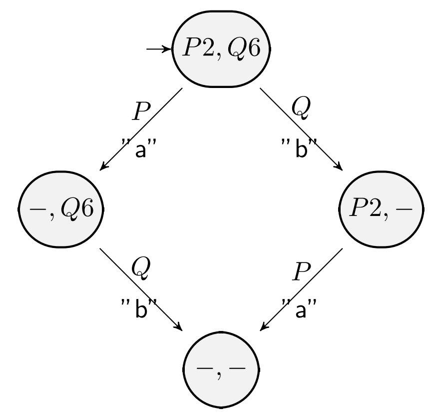
We next show how to model other basic features of GROOVY in Transition systems.

- Dealing with print statements. A statement of the form `print e` is represented as a transition decorated with the string value resulting from evaluating `e`. As an example, consider the following program.

```
Thread.start { //P
   print "a"
}

Thread.start { //Q
   print "b"
}
```

Its associated transition system is depicted below:



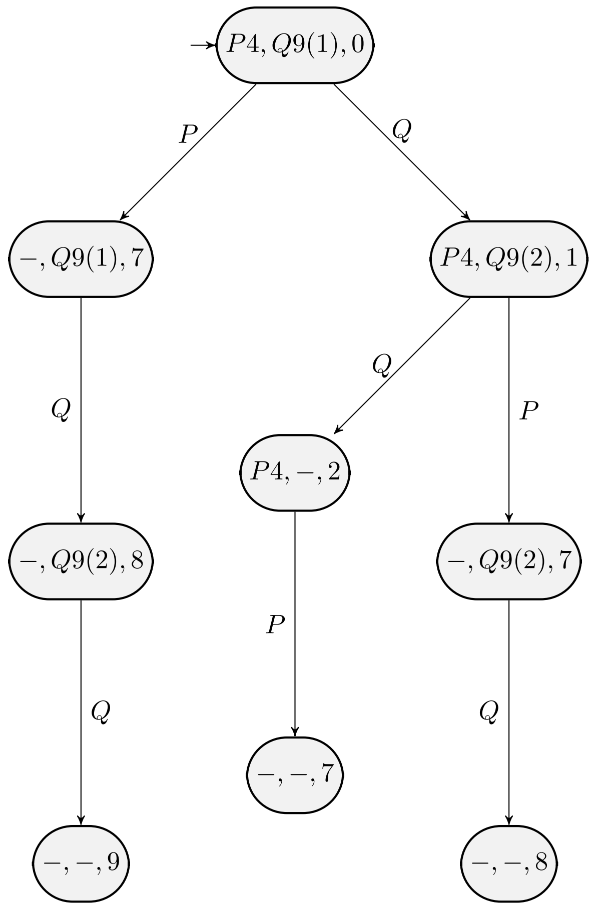
- Dealing with bounded iteration (i.e. "for"-loops). Consider the following program.

```
1  int x = 0

3  Thread.start { //P
      x = 7
5  }

7  Thread.start { //Q
      2.times {
9        x = x+1
      }
11 }
```

Its associated transition system is depicted below. The instruction pointer for P starts at line 4. The instruction pointer for Q starts at line 9. Note the index indicating the iteration next to the instruction pointer for Q. For example, in the initial state `Q9(1)` states that Q is ready to run line 9 and that this line is part of the first iteration of the loop. The iteration index avoids states $(-, Q9(1), 7)$ and $(-, Q9(2), 7)$ from being incorrectly identified.



- Dealing with local variables. Note how we distinguish local variables with the same name using ”local_P” and ”local_Q” in the state format.

```
int x = 0 // shared variable

Thread.start { //P
   int local = x
   x = local+1 // atomic
}

Thread.start { //Q
   int local = x
   x = local+1 // atomic
}
```

- Dealing with “while”-loops.

```
int x=0 // shared variable

Thread.start { //P
```

```
    while (x<1) {
      print x
    }
}

Thread.start { //Q
    x = x + 1
}
```

1.3 Atomicity and Race Conditions

1.3.1 Atomicity

Consider the following program:

```
int x=0

Thread.start { //P
    x = x + 1
    println x
}

Thread.start { //Q
    x = x + 1
    println x
}
```

One would expect 1 and 2, or 2 and 2 to be printed. These are indeed possible outputs. However, 1 and 1 is also possible:

```bash
$ groovy ex3
1
1
$
```

The reason is that assignment is not an atomic operation, rather it is decomposed into more fine grained (bytecode) operations. It is the latter that are interleaved. Let's take a closer look at those fine grained operations. Consider the following Java class that spawn two threads, each of which updates a shared variable:

```java
class A implements Runnable {
    static int x=0;

    public void run() {
       x=x+1;
    }

    public static void main(String[] args) {
       new Thread(new A()).start();
       new Thread(new A()).start();
    }
}
```
A.java

We compile it and look at the resulting bytecode by using `javap`, the Java class file disassembler:

```bash
   $ javac A.java
2  $ javap -c A
   Compiled from "A.java"
4  class A implements java.lang.Runnable {
     static int x;
6
     A();
8      Code:
          0: aload_0
10        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
          4: return
12
     public void run();
14     Code:
          0: getstatic     #7                  // Field x:I
16        3: iconst_1
          4: iadd
18        5: putstatic     #7                  // Field x:I
          8: return
20
     public static void main(java.lang.String[]);
22     Code:
          0: new           #13                 // class java/lang/Thread
24        3: dup
          4: new           #8                  // class A
26        7: dup
          8: invokespecial #15                 // Method "<init>":()V
28       11: invokespecial #16                 // Method java/lang/Thread."<init>":(Ljava
         14: invokevirtual #19                 // Method java/lang/Thread.start:()V
30       17: new           #13                 // class java/lang/Thread
         20: dup
32       21: new           #8                  // class A
         24: dup
34       25: invokespecial #15                 // Method "<init>":()V
         28: invokespecial #16                 // Method java/lang/Thread."<init>":(Ljava
36       31: invokevirtual #19                 // Method java/lang/Thread.start:()V
         34: return
38
     static {};
40     Code:
          0: iconst_0
42        1: putstatic     #7                  // Field x:I
          4: return
44 }
```

bash

The only lines we are interested are lines 15 to 18. Each thread has a JVM stack. Every time a method is called, a new frame is created (heap-allocated) and stored on the JVM stack for that thread. Each frame has its own array of local variables, its own operand stack, and a reference to the run-time constant pool of the class of the current method. The instruction `x=x+1` is compiled to four bytecode instructions whose meaning can be read off from their opcodes:

```
      0: getstatic     #7                  // Field x:I
      3: iconst_1
      4: iadd
      5: putstatic     #7                  // Field x:I
```

It is these operations, for each thread, that get interleaved. Thus, it is possible to have the following interleaving:

```
      0(P): getstatic     #7               // Field x:I
      0(Q): getstatic     #7               // Field x:I
      3(P): iconst_1
      3(Q): iconst_1
      4(P): iadd
      4(Q): iadd
      5(P): putstatic     #7               // Field x:I
      5(Q): putstatic     #7               // Field x:I
```

These instructions end up storing 1 in `x`.

1.3.2 Race Condition

The fact that assignment is not atomic, as exemplified in Example 1.3.1, is not a problem in itself unless the resulting atomic operations, into which assignment is compiled, "interfere" with each other. If that happens, then unexpected behaviour may occur such as Example 1.3.1 producing 11 as output. Such interference occurs when race conditions are present.

Definition 1.3.1. *A race condition arises if two or more threads access the same variables or objects concurrently and at least one does updates.*

Once thread P starts doing something, it needs to "race" to finish it because if thread Q looks at the shared variable before P is done, it may see something inconsistent

Another example of unexpected behavior due to race conditions is the following one. Each thread simulates the behavior of a turnstile which signals the arrival of a person by incrementing a shared counter. Its execution typically produces values between 10 and 20. However, any value between 2 and 20 is possible.

```
int counter=0 // shared variable

P = Thread.start {
      10.times {
        counter = counter+1
      }
}
Q = Thread.start {
      10.times {
        counter= counter+1
      }
    }

P.join() // wait for P to finish
Q.join() // wait for Q to finish
```

```
println counter // print value of counter
```

1.4 The Mutual Exclusion Problem

This Mutual Exclusion (ME) problem, also referred as the Critical Section problem, was introduced by Edsger W. Dijkstra in 1965. It is the guarantee of mutually exclusive access to a single shared resource when there are several competing processes [Dij65]. This arises in many areas including operating systems, database systems, computer networks, and others. Dijkstra's paper is considered to be the starting point of concurrency from a Computer Science perspective [Lam15].

1.4.1 Problem Statement

We next state the ME problem for two threads; the case of more than two threads will be presented later. Consider two threads `P` and `Q` that continuously go in and out of a region of code called a critical section. The critical section is represented below with the comment `// CRITICAL SECTION`.

```
Thread.start { // P
  while(true) {
  // non-critical section
  entry to critical section;
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

```
Thread.start { // Q
  while(true) {
  // non-critical section
  entry to critical section
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

A solution to the problem should enjoy the following three properties:

- **Mutex**. At most one thread can be executing inside its critical section.
- **Absence of livelock**. If both threads try to enter the critical section, at least one of them will succeed. This property is also referred to as the "progress property".
- **Freedom from starvation**. A thread trying to enter its critical section will eventually be able to do so.

Some fundamental assumptions are required before we even attempt at solving it.

- There are no shared variables between the critical section and the non critical section (nor with the entry/exit protocol).
- The critical section always terminates. Failure to do so would starve the other thread.
- The scheduler is (weakly) fair: if a statement in a process is executable infinitely long, then it is eventually executed.

We next introduce several failed attempts at solving the CS problem in order to gain intuition about it. Then we present two solutions (Peterson and Dekker) for the case of two threads. Finally, we discuss the filter lock and the Bakery algorithm, which solve the ME problem for any number of threads.

1.4.2 Naive Attempts

1.4.3 Peterson

1.4.4 Dekker

1.4.5 The CS problem for $N > 2$

Filter Lock

Bakery

Chapter 2

Semaphores

2.1 Introduction

2.2 The ME Problem Revisited

Consider the following solution to the ME problem using a binary semaphore presented in listing 2.1[1].

```
Semaphore mutex= new Semaphore(1)

Thread.start { //P
    while (true) {
        mutex.acquire()
        mutex.release()
    }
}
Thread.start { //Q
    while (true) {
        mutex.acquire()
        mutex.release()
    }
}
```

Listing 2.1: Solution to ME problem using a binary semaphore

One easy way to verify that all three properties of ME problem are upheld is to construct its transition system and then analyze these properties. This requires a means for representing semaphores. Since a semaphore is an object with state and the latter includes the number of permits and the set of blocked processes, we shall model `mutex` using the expression `mutex[i,S]` where `i` is the number of permits and `S` is a set of blocked processes. Moreover, we use the "!" symbol as instruction pointer in the states of our transition systems to indicate that there are no instructions ready to execute. For example, a state such as $P6, !, mutex[0, \{Q11\}]$, reflects that only P can be scheduled for execution, there are no permits available in `mutex` and one thread is

[1] Groovy requires that you import the Semaphore class in order to be able to use it. All code excerpts involving semaphores should thus include, at the top, the line `import java.util.concurrent.Semaphore`. This is typically omitted in our examples.

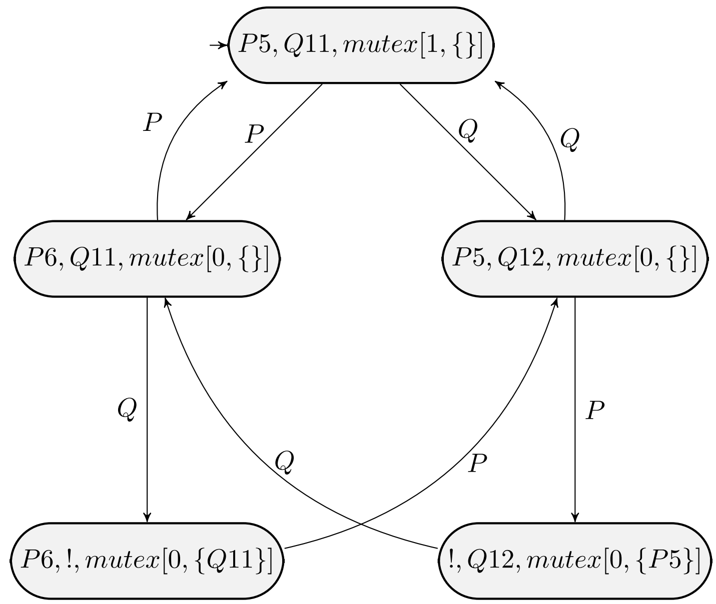


Figure 2.1: Transition System for the solution to ME problem using a binary semaphore

blocked on `mutex` waiting for a permit to become available, namely Q. Figure **??** is the transition system for the listing in Figure 2.1.

Consider the setting where the above solution is applied to three threads wanting to access their CS. This is illustrated in Listing 2.2. Although Mutex and Absence of Livelock are upheld, Freedom From Starvation is not. Indeed, consider the scenario where `P` goes in and `Q` and `R` try to get in and are both blocked and placed in the set of blocked processes for `mutex`. [COMPLETE]**??**

This is easily solved by having the set of blocked processes in `mutex` be a queue. Such semaphores are called fair semaphores. This is achieved by using an alternative constructor for semaphores that includes a fairness parameter

```
Semaphore(int permits, boolean fair)
```

Replacing line 1 in Listing 2.2 with `Semaphore mutex= new Semaphore(1,true)` suffices to obtain a correct solution to the ME problem for any number of threads.

```
Semaphore mutex= new Semaphore(1)

Thread.start { //P
    while (true) {
        mutex.acquire()
        mutex.release()
    }
}
Thread.start { //Q
    while (true) {
        mutex.acquire()
        mutex.release()
    }
```

```
  }
Thread.start { //R
    while (true) {
        mutex.acquire()
        mutex.release()
    }
}
```

Listing 2.2: Attempt at solving the ME problem using a binary semaphore for N=3

2.3 More Examples of Simple Thread Synchronization

2.3.1 "cd after ab" Example

Consider the following program. There are six possible sequences of letters that may be printed. Suppose we wanted to ensure that only the sequence `"cdab"` is printed. We could do so by having `"a"` be printed only after `"d"`.

```
Thread.start { //P
    print "a"
    print "b"
}
Thread.start { //Q
    print "c"
    print "d"
}
```

This can be achieved with semaphores as follows:

```
import java.util.concurrent.Semaphore
Semaphore a_after_d = new Semaphore(0)

Thread.start { //P
    a_after_d.acquire()
    print "a"
    print "b"
}
Thread.start { //Q
    print "c"
    print "d"
    a_after_d.release()
}
```

2.3.2 "$(aab)^\omega$" Example

Consider the following example which prints any (infinite) sequence of "a"s and "b"s[2]:

```
Thread.start { //P
    while (true) {
      print "a"
    }
}
```

[2] Under the assumption of fairness of the scheduler, it outputs any sequence of the form $(a^+b + b^+a)^\omega$.

```
Thread.start { //Q
    while (true) {
      print "b"
    }
}
```

Using semaphores, how would you ensure that only the infinite sequence `"aabaabaab..."` is printed? Hint: make use of two semaphores, `a` and `b`, enabling the execution of an iteration in P and an iteration in Q, respectively.

Here is a solution.

```
import java.util.concurrent.Semaphore
Semaphore a = new Semaphore(2)
Semaphore b = new Semaphore(0)

Thread.start { //P
    while (true) {
      a.acquire()
      print "a"
      b.release()
    }
}

Thread.start { //Q
    while (true) {
      b.acquire(2)
      print "b"
      a.release(2)
    }
}
```

2.4 Thread Dumps

We can check the current thread dump of the our GROOVY/Java application as follows. Let's use the example above. First we modify our code so that we give our threads an easy to spot name and remove the lines that print. The result is below; we'll call it `ex1.groovy`.

```
import java.util.concurrent.Semaphore
Semaphore a = new Semaphore(2)
Semaphore b = new Semaphore(0)

Thread.start { //P
    Thread.currentThread().setName("P Thread");
    while (true) {
      a.acquire()
      // print "a"
      b.release()
    }
}

Thread.start { //Q
    Thread.currentThread().setName("Q Thread");
    while (true) {
```

```
        b.acquire(2)
        // print "b"
        a.release(2)
    }
}
```

Now we run it in the background, use `jstack`[3] to obtain the stack trace of each thread of the java process and send the output to a text file `thead-dump.txt`

```
$ groovy ex1 &
[1] 23275      # job no. 1 started, 23275 is PID of groovy ex1 command
$ jstack -l 23275 > thread-dump.txt
$ kill %1
[1]  + 23275 exit 143    groovy ex1
$ emacs thread-dump.txt
```

bash

The dump contains information on all threads involved in our application. We'll just show an exerpt that mentions `P` and `Q`. We can see that the former is in a RUNNABLE state and the latter is in a WAITING state. We can also see the current instruction being executed in each thread.

```
"P Thread" #17 prio=5 os_prio=31 cpu=5910.73ms elapsed=10.91s tid=0
    x00007f7d9412b400 nid=27655 runnable  [0x000070000c858000]
   java.lang.Thread.State: RUNNABLE
    at jdk.internal.misc.Unsafe.unpark(java.base@18.0.1.1/Native Method)
    at java.util.concurrent.locks.LockSupport.unpark(java.base@18.0.1.1/
    LockSupport.java:177)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.signalNext(java.
    base@18.0.1.1/AbstractQueuedSynchronizer.java:611)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.releaseShared(
    java.base@18.0.1.1/AbstractQueuedSynchronizer.java:1095)
    at java.util.concurrent.Semaphore.release(java.base@18.0.1.1/Semaphore.
    java:432)
    at java.lang.invoke.LambdaForm$DMH/0x0000000800d28000.invokeVirtual(java.
    base@18.0.1.1/LambdaForm$DMH)
    at java.lang.invoke.LambdaForm$MH/0x0000000800e32c00.invoke(java.base@18
    .0.1.1/LambdaForm$MH)
    at java.lang.invoke.LambdaForm$MH/0x0000000800e2b400.guardWithCatch(java.
    base@18.0.1.1/LambdaForm$MH)
    at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18
    .0.1.1/DelegatingMethodHandle$Holder)
    at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18
    .0.1.1/LambdaForm$MH)
    at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18
    .0.1.1/DelegatingMethodHandle$Holder)
    at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18
    .0.1.1/LambdaForm$MH)
    at java.lang.invoke.Invokers$Holder.linkToCallSite(java.base@18.0.1.1/
    Invokers$Holder)
    at ex1$_run_closure1.doCall(ex1.groovy:12)
    at ex1$_run_closure1.doCall(ex1.groovy)
    at java.lang.invoke.DirectMethodHandle$Holder.invokeSpecial(java.base@18
    .0.1.1/DirectMethodHandle$Holder)
```

[3] https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstack.html

```
	at java.lang.invoke.LambdaForm$MH/0x0000000800c1c800.invoke(java.base@18.0.1.1/LambdaForm$MH)
	at java.lang.invoke.Invokers$Holder.invokeExact_MT(java.base@18.0.1.1/Invokers$Holder)
	at jdk.internal.reflect.DirectMethodHandleAccessor.invokeImpl(java.base@18.0.1.1/DirectMethodHandleAccessor.java:154)
	at jdk.internal.reflect.DirectMethodHandleAccessor.invoke(java.base@18.0.1.1/DirectMethodHandleAccessor.java:104)
	at java.lang.reflect.Method.invoke(java.base@18.0.1.1/Method.java:577)
	at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:343)
	at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:328)
	at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:279)
	at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:1009)
	at groovy.lang.Closure.call(Closure.java:418)
	at groovy.lang.Closure.call(Closure.java:412)
	at groovy.lang.Closure.run(Closure.java:500)
	at java.lang.Thread.run(java.base@18.0.1.1/Thread.java:833)

   Locked ownable synchronizers:
	- None

"Q Thread" #18 prio=5 os_prio=31 cpu=6110.53ms elapsed=10.91s tid=0x00007f7d9411fa00 nid=28163 runnable  [0x000070000c95b000]
   java.lang.Thread.State: WAITING (parking)
	at jdk.internal.misc.Unsafe.park(java.base@18.0.1.1/Native Method)
	- parking to wait for  <0x00000006180b9960> (a java.util.concurrent.Semaphore$NonfairSync)
	at java.util.concurrent.locks.LockSupport.park(java.base@18.0.1.1/LockSupport.java:211)
	at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:715)
	at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:1047)
	at java.util.concurrent.Semaphore.acquire(java.base@18.0.1.1/Semaphore.java:318)
	at java.lang.invoke.LambdaForm$DMH/0x0000000800d28000.invokeVirtual(java.base@18.0.1.1/LambdaForm$DMH)
	at java.lang.invoke.LambdaForm$MH/0x0000000800e32c00.invoke(java.base@18.0.1.1/LambdaForm$MH)
	at java.lang.invoke.LambdaForm$MH/0x0000000800e2b400.guardWithCatch(java.base@18.0.1.1/LambdaForm$MH)
	at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder)
	at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
	at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder)
	at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
	at java.lang.invoke.Invokers$Holder.linkToCallSite(java.base@18.0.1.1/Invokers$Holder)
	at ex1$_run_closure2.doCall(ex1.groovy:19)
	at ex1$_run_closure2.doCall(ex1.groovy)
	at java.lang.invoke.DirectMethodHandle$Holder.invokeSpecial(java.base@18.0.1.1/DirectMethodHandle$Holder)
```

```
    at java.lang.invoke.LambdaForm$MH/0x0000000800c1c800.invoke(java.base@18.0.1.1/LambdaForm$MH)
    at java.lang.invoke.Invokers$Holder.invokeExact_MT(java.base@18.0.1.1/Invokers$Holder)
    at jdk.internal.reflect.DirectMethodHandleAccessor.invokeImpl(java.base@18.0.1.1/DirectMethodHandleAccessor.java:154)
    at jdk.internal.reflect.DirectMethodHandleAccessor.invoke(java.base@18.0.1.1/DirectMethodHandleAccessor.java:104)
    at java.lang.reflect.Method.invoke(java.base@18.0.1.1/Method.java:577)
    at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:343)
    at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:328)
    at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:279)
    at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:1009)
    at groovy.lang.Closure.call(Closure.java:418)
    at groovy.lang.Closure.call(Closure.java:412)
    at groovy.lang.Closure.run(Closure.java:500)
    at java.lang.Thread.run(java.base@18.0.1.1/Thread.java:833)

   Locked ownable synchronizers:
    - None
```

One could also make use of online tools that analyse these thread dumps to help identify potential issues. For example, you can try and upload `thread-dump.txt` to this site `fastthread.io` and click on "analyze".

2.5 Classical Synchronization Problems

This section addresses some classical synchronization problems using semaphores.

2.5.1 Producers/Consumers

Buffer of size 1, one producer and one consumer. The code below also works if there were multiple producers and multiple consumers.

```
Integer buffer // shared buffer
Semaphore consume = new Semaphore(0)
Semaphore produce = new Semaphore(1)

Thread.start { // Prod
    Random r = new Random()
    while (true) {
       produce.acquire()
       buffer = r.nextInt(10000) // produce()
       println "produced "+buffer
       Thread.sleep(1000)
       consume.release()
    }
}

Thread.start { // Cons
    while (true) {
       consume.acquire()
       println "consumed "+buffer
```

```
        buffer = null // consume(buffer)
        produce.release()
    }
}
```

Buffer of size `N` with one producer and one consumer. Also known as a blocking queue.

```
final int N=10
Integer[] buffer = [0] * N // shared buffer

Semaphore consume = new Semaphore(0)
Semaphore produce = new Semaphore(N)
int start = 0
int end = 0

Thread.start { // Prod
    Random r = new Random()
    while (true) {
        produce.acquire()
        mutexP.acquire()
        buffer[start] = r.nextInt(10000) // produce()
        println id+" produced "+buffer[start] + " at index "+start
        start = (start + 1) % N
        mutexP.release()
        consume.release()
    }
}

Thread.start { // Cons
    while (true) {
        consume.acquire()
        mutexC.acquire()
        println id+ " consumed "+buffer[end] + " at index "+end
        buffer[end] = null // consume(buffer)
        end = (end + 1) % N
        mutexC.release()
        produce.release()
    }
}
```

Buffer of size `N` with multiple producers and multiple consumers.

The static method `currentMethod()` returns a reference to the currently executing thread object. Every thread has a unique id. It may be obtained by using the `getId()` method.

```
final int N=10
Integer[] buffer = [0] * N

Semaphore consume = new Semaphore(0)
Semaphore produce = new Semaphore(N)
Semaphore mutexP = new Semaphore(1) // mutex to avoid race conditions on start
Semaphore mutexC = new Semaphore(1) // mutex to avoid race conditions on end
int start = 0
int end = 0

```

```
5.times {
    Thread.start { // Prod
      Random r = new Random()
      while (true) {
          produce.acquire()
          mutexP.acquire()
          buffer[start] = r.nextInt(10000) // produce()
          println Thread.currentThread().getId()+" produced "+buffer[start] + " at index "+start
          start = (start + 1) % N
          mutexP.release()
          consume.release()
      }
    }
}

5.times{
    Thread.start { // Cons
      while (true) {
          consume.acquire()
          mutexC.acquire()
          println Thread.currentThread().getId()+ " consumed "+buffer[end] + " at index "+end
          buffer[end] = null // consume(buffer)
          end = (end + 1) % N
          mutexC.release()
          produce.release()
      }
   }
}
```

2.5.2 Readers/Writers

2.5.3 Barrier Synchronization

A barrier is a point in the program (which we call the synchronization point) where a thread must wait for the some other group of threads before it can proceed. The barrier may be seen to be “lowered” until all threads in the group have arrived, at which time it is “raised”. This type of synchronization is parameterized over the number of threads in the system `N` and the size of the barrier `B`. A one-time use barrier is a barrier in which, once all threads have reached the synchronization point, then the barrier is considered exhausted; all subsequent threads that reach the synchronization point need no longer wait for the others.

```
// One-time use barrier
final int N=3 // Threads in the system
final int B=3 // Barrier size
N.times {
  Thread.start {
    while (true) {
      // barrier arrival protocol
      // barrier
    }
  }
}
```

A solution to the on-time-use barrier is stated in listing [?].

```
import java.util.concurrent.Semaphore
// One-time use barrier
final int N=3 // Threads in the system
final int B=3 // Barrier size
int t=0
Semaphore barrier = new Semaphore(0)
Semaphore mutex = new Semaphore(1)
N.times {
  Thread.start {
     while (true) {
        // barrier arrival protocol
        mutex.acquire()
        if (t<B) {
           t++
           if (t==B) {
              barrier.release(B) // raise barrier
           }
        }
        mutex.release()
        // barrier
        barrier.acquire()
        barrier.release()
       }
   }
}
```

Listing 2.3: One-time use barrier

For example, the following program will always print the letters before the numbers. For example, "0:a 1:a 0:1 1:1" is possible but not "0:a 1:1 1:a 0:1".

```
import java.util.concurrent.Semaphore
// One-time use barrier
final int N=2 // Threads in the system
final int B=2 // Barrier size
int t=0
Semaphore barrier = new Semaphore(0)
Semaphore mutex = new Semaphore(1)
N.times {
    int id = it
    Thread.start {
      println (id+":a ")
      // barrier arrival protocol
      mutex.acquire()
      if (t<B) {
          t++
          if (t==B) {
             barrier.release(B)
          }
      }
      mutex.release()
      // barrier
      barrier.acquire()
      barrier.release()
      println (id+":1 ")
    }
}
```

Using cascaded signalling:

```
import java.util.concurrent.Semaphore
// One-time use barrier
final int N=3 // Threads in the system
final int B=3 // Barrier size
int t=0
Semaphore barrier = new Semaphore(0)
Semaphore mutex = new Semaphore(1)
N.times {
  Thread.start {
    while (true) {
      // barrier arrival protocol
      mutex.acquire()
      if (t<B) {
        t++
        if (t==B) {
          barrier.release()
        }
      }
      mutex.release()
      // barrier
      barrier.acquire() // Cascaded signalling
      barrier.release()
    }
  }
}
```

Cyclic (or reusable) barrier. Failed attempt:

```
import java.util.concurrent.Semaphore
// Cyclic (ie. Reusable)  barrier
final int N=3 // Threads in the system
final int B=3 // Barrier size
Semaphore mutex = new Semaphore(1)
Semaphore barrier = new Semaphore(0)
int t=0

N.times {
    Thread.start {
    while (true) {
        // arrival
        mutex.acquire()
            if (t<B) {
               t++;
               if (t==B) {
                   barrier.release(B)
                   t=0 // attempt to reset barrier counter
               }
            }
        mutex.release()

        // barrier
        barrier.acquire()
    }
    }
}
```

Listing 2.4: Cyclic Barrier - Failed Attempt

Exercise 2.5.1. *Show that in listing 2.4, a thread can get an unbounded number of iterations ahead of the other two threads. Do so by exhibiting an appropriate path in the transition system.*

One easy way to verify that it is incorrect is to count the number of times a thread cycles passed the barrier. Then, notice that some threads can race far ahead of others in terms of the difference in number cycles; this difference can be larger than 1.

A solution follows. We use a second barrier to wait for all threads to fall through the first barrier, thus avoiding any one thread getting ahead of the others.

```
import java.util.concurrent.Semaphore

// Cyclic (ie. Reusable) barrier
final int N=3 // Threads in the system
final int B=3 // Barrier size
Semaphore mutex = new Semaphore(1)
Semaphore barrier = new Semaphore(0)
Semaphore barrier2 = new Semaphore(0)
int t=0

N.times {
  int id = it
  Thread.start {
     while (true) {
       // arrival
       mutex.acquire()
       t++;
       if (t==B) {
          barrier.release(B)
       }
       mutex.release()

       // barrier
       barrier.acquire()

       mutex.acquire()
       t--
       if (t==0) {
          barrier2.release(B)
       }
       mutex.release()

       barrier2.acquire()
     }
  }
}
```

Listing 2.5: Cyclic Barrier

Exercise 2.5.2. *Show that the program in Listing 2.5 may deadlock if `N>B`. Also, it may allow a thread to traverse the barrier, without having to wait for others. Propose a solution. Hint: have a separate counter for threads arriving at the second barrier.*

Chapter 3

Monitors

A monitor is a program module that encapsulates data and operations and, moreover, guarantees mutual exclusion in the execution of the operations.

Listing 3.1 implements two turnstiles each of which accesses a shared counter. The counter is implemented using a monitor. This monitor supports operations `inc()`, `dec()` and `read()`. The `synchronized` qualifier ensures mutual exclusion in the execution of these methods. Every object has a built in lock called an intrinsic lock. When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. Threads will block when they attempt to acquire the lock.

> The shared variable `c` need not be declared `volatile` when using `synchronized`[1].

```
// Monitor declaration
  class Counter {
    private int c

    public synchronized void inc() {
       c++
    }

    public synchronized int read() {
       return c
    }
}

// Sample use of the monitor
Counter ctr = new Counter()

P = Thread.start {
         10.times {
             ctr.inc()
```

[1] https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

```
        }
      }

Q = Thread.start {
        10.times {
            ctr.inc()
        }
     }

P.join()
Q.join()
println (ctr.read())
```

Listing 3.1: Avoiding race conditions on a shared counter using a monitor

3.1 A monitor implementing a semaphore

Listing 3.2 shows how we may implement a semaphore using monitors. We assume that the number of permits in our semaphores is always positive.

```
class Semaphore {
    private int permits

    Semaphore(int init) {
      permits=init
    }

    public synchronized void acquire() {
      while (permits==0) {
          wait()
      }
      permits--
    }

    public synchronized void release() {
      notify()
      permits++
    }
}

Semaphore mutex = new Semaphore(1)
int c=0

P = Thread.start {
      10.times {
          mutex.acquire()
          c++
          mutex.release()
      }
   }

Q = Thread.start {
      10.times {
         mutex.acquire()
         c++
         mutex.release()
```

```
      }
    }

P.join()
Q.join()
println c
```

Listing 3.2: Implementing a Semaphore using Monitors

This solution is not starvation free. Waiting threads can be overtaken by arriving threads (called "barging"); this could take place indefinitely so long as new threads arrive continuously. A starvation-free solution[2] is given in Listing 3.3. A different approach is followed in [Car96].

```
class Semaphore {
    private int permits
    private long startWaitingTime=0
    private static final long startTime=System.currentTimeMillis()
    private int waiting=0

    Semaphore(int init) {
       permits=init
    }

    private static final long age() {
       return System.currentTimeMillis() - startTime
    }

    synchronized void acquire() {
       if (waiting>0  permits==0) {
           long arrivalTime = age()
           while (arrivalTime>startWaitingTime  permits==0) {
               waiting++
               wait()
               waiting--
           }
       }
       permits--
   }

    synchronized void release() {
       permits++
       startWaitingTime = age()
       notify()
    }
}
```

Listing 3.3: Starvation-free semaphores

3.2 Producers/Consumers

```
class PC {
    private Object buffer;
```

[2] The idea of using age is from [Har98] which uses it in an attempt to propose a fair solution for readers/writers. Unfortunately, the proposed solution is not fair (after an `endWrite` operation, a writer could steal the lock even though there are waiting readers).

```
    public synchronized void produce(Object o) {
      while (buffer!=null) {
          wait()
      }
      buffer = o
      notifyAll()
    }

    public synchronized Object consume() {
      while (buffer==null) {
          wait()
      }
      Object temp = buffer
      buffer=null
      notifyAll()
      return temp
    }
}

PC pc = new PC()

10.times {
    Thread.start {
      println (Thread.currentThread().getId()+" consumes")
      pc.consume()
    }}

10.times {
    Thread.start {
      println (Thread.currentThread().getId()+" produces")
      pc.produce((new Random()).nextInt(33))
    }}
```

Replacing each of the two `notifyAll()` with `notify()` leads to an incorrect solution where one can end up having a producer and consumer both blocked in the wait-set. Hint: C1,C2,P1,P2. This pitfall is called the lost-wakeup problem.

Disadvantages:

Use multiple condition variables

Condition variables.

3.3 Readers/Writers

Listing 3.4 presents a correct solution to the readers/writers problem. However, it is not starvation-free since both readers and writers may starve. In other words, a reader may have to wait indefinitely before being able to read (and similarly with a writer).

```
import java.util.concurrent.locks.*

class RW {
    private int readers, writers;
    static final Lock lock = new ReentrantLock();
    static final Condition okToRead = lock.newCondition();
    static final Condition okToWrite = lock.newCondition();

```

```
    RW() {
      readers=writers=0;
    }

    void start_read() {
      lock.lock();
      try {
          while (writers>0) {
              okToRead.await();
          }
          readers++;
      } finally {
          lock.unlock();
      }
    }

    void stop_read() {
      lock.lock();
      try {
          readers--;
          if (readers==0) {
              okToWrite.signal();
          }
      } finally {
          lock.unlock();
      }
    }

    void start_write(Object item) {
      lock.lock();
      try {
          while (readers>0  writers>0) {
              okToWrite.await();
          }
          writers++;
      } finally {
          lock.unlock();
      }
    }

    void stop_write() {
      lock.lock();
      try {
          writers--;
          okToWrite.signal();
          okToRead.signalAll();
      } finally {
          lock.unlock();
      }
    }
}

RW rw = new RW();

r =  { //R
    Random r = new Random();
    rw.start_read();
    println Thread.currentThread().getId()+" reading..."
```

```
    Thread.sleep(r.nextInt(1000));
    println Thread.currentThread().getId()+" done reading..."

    rw.stop_read();
}

w = { //W
    Random r = new Random();
    rw.start_write();
    println Thread.currentThread().getId()+" writing..."
    Thread.sleep(r.nextInt(1000));
    println Thread.currentThread().getId()+" done writing..."
    rw.stop_write();
}

200.times {
    Thread.start(r)
    Thread.start(w)
}
```

Listing 3.4: Readers/Writers

Checking for waiting writers. Avoids starvation of writers. However, readers may still starve.

```
  import java.util.concurrent.locks.*

class RW {
    private int readers, writers;
    private int writers_waiting;
    static final Lock lock = new ReentrantLock();
    static final Condition okToRead = lock.newCondition();
    static final Condition okToWrite = lock.newCondition();

    RW() {
      readers=writers=0;
      writers_waiting=0;
    }

    void start_read() {
      lock.lock();
      try {
          while (writers>0  writers_waiting>0) {
              okToRead.await();
          }
          readers++;
      } finally {
          lock.unlock();
      }
    }

    void stop_read() {
      lock.lock();
      try {
          readers--;
          if (readers==0) {
              okToWrite.signal();
          }
      } finally {
          lock.unlock();
```

```java
      }
    }

    void start_write(Object item) {
      lock.lock();
      try {
          while (readers>0  writers>0) {
              writers_waiting++;
              okToWrite.await();
              writers_waiting--;
          }
          writers++;
      } finally {
          lock.unlock();
      }
    }

    void stop_write() {
      lock.lock();
      try {
          writers--;
          okToWrite.signal();
          okToRead.signalAll();
      } finally {
          lock.unlock();
      }
    }
}
```

An incorrect attempt at a starvation-free solution to RW is presented in Listing 3.5. One situation that may lead to deadlock is: W1,R1,W2. Another is: R1, W1, R2.

```java
import java.util.concurrent.locks.*

class RW {
   private int readers, writers;
   private int readers_waiting, writers_waiting;
   static final Lock lock = new ReentrantLock();
   static final Condition okToRead = lock.newCondition();
   static final Condition okToWrite = lock.newCondition();

   RW() {
     readers=writers=0;
     readers_waiting=writers_waiting=0;
   }

   void start_read() {
       lock.lock();
       try {
         while (writers>0  writers_waiting>0) {
               readers_waiting++;
               okToRead.await();
               readers_waiting--;
           }
           readers++;
       } finally {
           lock.unlock();
       }
```

```
    }

    void stop_read() {
        lock.lock();
        try {
            readers--;
            if (readers==0) {
                okToWrite.signal();
            }
        } finally {
            lock.unlock();
        }
    }

    void start_write(Object item) {
        lock.lock();
        try {
            while (readers>0  writers>0  readers_waiting>0) {
                writers_waiting++;
                okToWrite.await();
                writers_waiting--;
            }
            writers++;
        } finally {
            lock.unlock();
        }
    }

    void stop_write() {
        lock.lock();
        try {
            writers--;
            okToWrite.signal();
            okToRead.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

Listing 3.5: Incorrect attempt at a fair solution to RW; may deadlock

If we replace `stop_write` with the following code, then our solution may deadlock. Hint: Consider W1,R1,W2.

```
void stop_write() {
    lock.lock();
    try {
        writers--;
        if (readers_waiting==0) {
        okToWrite.signal();
        } else {
        okToRead.signalAll();
        }
    } finally {
        lock.unlock();
    }
    }
```

Exercise 3.3.1. *Explain why the following proposed solution to the train problem may deadlock.*

```
import java.util.concurrent.locks.*;

class TrainStation {
    boolean nt=false;
    boolean st=false;
    final Lock lock = new ReentrantLock();
    final Condition northTrack = lock.newCondition();
    final Condition southTrack = lock.newCondition();

    void acquireNorthTrackP() {
      lock.lock();
      try {
        while (nt) {
          northTrack.await();
        }
        nt=true;
      } finally {
        lock.unlock();
      }
    }

    void releaseNorthTrackP() {
      lock.lock();
        try {
          nt=false;
          northTrack.signal();
        } finally {
          lock.unlock();
        }
    }

    void acquireSouthTrackP() {
      lock.lock();
        try {
        while (st) {
        southTrack.await();
        }
        st=true;
        } finally {
        lock.unlock();
        }
    }

    void releaseSouthTrackP() {
      lock.lock();
        try {
        st=false;
        southTrack.signal();
        } finally {
        lock.unlock();
        }
    }

    void acquireTracksF() {
      lock.lock();
        try {
```

```
        while (nt   st) {
        northTrack.await();
        southTrack.await();
        }
        nt=true;
        st=true;
        } finally {
        lock.unlock();
        }
    }

    void releaseTracksF() {
    lock.lock();
        try {
        nt=false;
        st=false;
        southTrack.signal();
        northTrack.signal();
        } finally {
        lock.unlock();
        }
    }
}
```

Part II

Message Passing

Chapter 4

Message Passing in Erlang

4.1 Erlang

Erlang is programming language best suited for implementing distributed systems. A distributed Erlang system consists of a number of Erlang runtime systems communicating with each other. Each Erlang runtime system is called a node. Nodes must be given a name. The `erl` program starts an Erlang runtime system. If a name is provided, then a node with that name is created. If no name is provided, then no node is created. In these notes we will focus on concurrent rather than distributed programming, hence we will not be creating nodes.

```bash
$ erl
Erlang/OTP 27 [erts-15.1.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [dtrace
Eshell V15.1.2 (press Ctrl+G to abort, type help(). for help)
1> node().
nonode@nohost
2>
```

bash

4.2 Examples

> A set of so called Built-in Functions (BIFs) are preloaded[1] in every Erlang session. If you try to define a function in your own module whose name clashes with that of a BIF, then the compiler will issue an error. You can selectively avoid loading BIFs using the following attribute `-compile({no_auto_import,[length/1]}).`, which in this example avoids loading the `length/1` function.

4.2.1 Semaphores

[1] https://github.com/erlang/otp/blob/master/erts/preloaded/src/erlang.erl

```
-module(sem).
-compile(nowarn_export_all).
-compile(export_all).

make(N) ->
    spawn(?MODULE,sem_loop,[N]).

acquire(S) ->
    S!{acquire,self()},
    receive
      {ok} ->
        ok
    end.

release(S) ->
   S!{release}.

sem_loop(0) ->  %% no permits available
    receive
      {release} ->
        sem_loop(1)
    end;
sem_loop(N) when N>0 ->  %% permits available
    receive
      {acquire,From} ->
        From ! {ok},
        sem_loop(N-1);
      {release} ->
        sem_loop(N+1)
end.
```

sem.erl

```
-module(semcl).
-compile(nowarn_export_all).
-compile(export_all).

start() ->
    S = sem:make(0),
    spawn(?MODULE,client1,[S]),
    spawn(?MODULE,client2,[S]),
    ok.

client1(S) ->
    sem:acquire(S),
    io:format("a"),
    io:format("b").

client2(S) ->
    io:format("c"),
    io:format("d"),
    sem:release(S).
```

semcl.erl

4.2.2 A Cyclic Barrier

```
-module(barr).
-compile(nowarn_export_all).
-compile(export_all).

make(N) ->
    spawn(?MODULE,coordinator,[N,N,[]]).

reached(B) ->
    B!{reached,self()},
    receive
    ok ->
        ok
    end.

% coordinator(N,M,L)
% N: size of the barrier
% M: number of processes YET to arrive at the barrier
% L: list of PIDs of the processes that have already arrived at the barrier
coordinator(N,0,L) ->
    [ PID!ok || PID <- L],
    coordinator(N,N,[]);
coordinator(N,M,L) when M>0 ->
    receive
      {reached,From} ->
          coordinator(N,M-1,[From|L])
    end.
```

barr.erl

```
-module(barrcl).
-compile(nowarn_export_all).
-compile(export_all).

start() ->
    B = barr:make(3),
    spawn(?MODULE,client1,[B]),
    spawn(?MODULE,client2,[B]),
    spawn(?MODULE,client3,[B]),
    ok.

client1(B) ->
    io:format("a"),
    barr:reached(B),
    io:format("1"),
    client1(B).

client2(B) ->
    io:format("b"),
    barr:reached(B),
    io:format("2"),
    client2(B).

client3(B) ->
    io:format("c"),
    barr:reached(B),
    io:format("3"),
    client3(B).
```

barrcl.erl

4.2.3 Guessing Game

```
-module(gg).
-compile(nowarn_export_all).
-compile(export_all).

start() ->
    S = spawn(?MODULE,server_loop,[]),
    [ spawn(?MODULE,client,[S]) || _ <- lists:seq(1,100)].

client(S) ->
    S!{self(),start},
    receive
    {ok,Servlet} ->
        client_loop(Servlet,rand:uniform(100))
    end.

client_loop(Servlet, G) ->
    Servlet!{G,self()},
    receive
    {youGotIt,T} ->
        io:format("~w got it in ~w tries~n",[self(),T]);
    {tryAgain} ->
        client_loop(Servlet,rand:uniform(100))
    end.

server_loop() ->
    receive
    {From,start} ->
        ServLet = spawn(?MODULE,servlet,[rand:uniform(100),0]),
        From!{ok,ServLet},
        server_loop()
    end.

servlet(N,T) ->
    receive
    {Guess,From} when Guess==N ->
        From!{youGotIt,T};
    {Guess,From} when Guess/=N ->
        From!{tryAgain},
        servlet(N,T+1)
end.
```

gg.erl

4.2.4 Producers/Consumers

```
-module(pc).
-compile(nowarn_export_all).
-compile(export_all).

start(Cap,NofP,NofC) ->
    RS = spawn(?MODULE,resource,[0,Cap,0,0]),
    [ spawn(?MODULE,producer,[RS]) || _ <- lists:seq(1,NofP)],
    [ spawn(?MODULE,consumer,[RS]) || _ <- lists:seq(1,NofC)],
    ok.
```

```

%% client code
producer(RS) ->
    startProduce(RS),
    %% produce
    timer:sleep(rand:uniform(100)),
    stopProduce(RS).

consumer(RS) ->
    startConsume(RS),
    %% consume
    timer:sleep(rand:uniform(100)),
    stopConsume(RS).

%% PC code
startProduce(RS) ->
    RS!{startProduce,self()},
    receive
    {ok} ->
        ok
    end.

stopProduce(RS) ->
    RS!{stopProduce}.

startConsume(RS) ->
  RS!{startConsume,self()},
    receive
    {ok} ->
        ok
    end.

stopConsume(RS) ->
    RS!{stopConsume}.

resource(Size,Cap,SP,SC) ->
    receive
    {startProduce,From} when Size + SP =< Cap ->
        From!{ok},
        resource(Size,Cap,SP+1,SC);
    {stopProduce} ->
        resource(Size+1,Cap,SP-1,SC);
    {startConsume,From} when Size - SC > 0 ->
        From!{ok},
        resource(Size,Cap,SP,SC+1);
    {stopConsume} ->
        resource(Size-1,Cap,SP,SC-1)
    end.
```

pc.erl

Part III

Model Checking

Chapter 5

Promela

5.1 Syntax

We begin with a brief introduction to PROMELA through a series of examples. Programs in PROMELA are called models. A model typically declares and runs some number of processes. Processes communicate with other processes by either sharing variables or using channels.

5.1.1 Shared Variable in Promela

The following PROMELA model spawns two threads each of which writes to a shared variable n and then prints a message to standard output. The built-in variable _pid holds the pid of the thread that is currently running. The semi-colon is a statement separator, not a terminator.

```
byte n=0;

active proctype P() {
  n=1;
  printf("P has pid %d. n=%d\n",_pid,n)
};

active proctype Q() {
  n=2;
  printf("Q has pid %d. n=%d\n",_pid,n)
}
```

eg1.pml

Executing a PROMELA model is referred to as "simulation run of the model".

```
$ spin eg1.pml
          Q has pid 1. n=2
      P has pid 0. n=2
2 processes created
```

bash

Each process is a assigned a pid, starting from 0. By default, during simulation runs, SPIN arranges for the output of each active process to appear in a different column: the pid number is used to set the number of tab stops used to indent each new line of output that is produced by a process. You can use the `-T` option to supress indentation.

```
$ spin -T eg1.pml
P has pid 0. n=1
Q has pid 1. n=1
2 processes created
```

bash

5.1.2 Examples involving Loops

```
byte sum=0;

active proctype P() {
  byte i=0;
  do
   :: i>10 -> break
   :: else ->
         sum = sum + i;
         i++
  od;

  printf("The sum of the first 10 numbers is %d\n",sum)
}
```

The following example is one of an infinite loop. Run it and note also how SPIN reports overflows errors.

```
byte i=0;

active proctype P() {
 do
  :: i++;
     printf("Value of i: %d\n. ",i)
 od
}
```

An example using a for loop:

```
byte sum=0;

active proctype P() {
  byte i;
  for (i:1..10) {
    sum = sum + i
```

```
  }

  printf("The sum of the first 10 numbers is %d\n",sum)
}
```

5.1.3 Expressions as blocking commands

In PROMELA every statement is either executable or blocked. Most statements are always executable, examples being print statements, assignments, `skip`, `assert`, `break`, etc. Some are not. For example, `run` is executable only if there are less than 255 processes alive. Boolean expressions may be considered as statements. Such expressions are executable iff they evaluate to true. For example, `(5 < 7)` is always executable but `(x < 7)` blocks if `x` is not less than 7. Expressions used as statements cannot include side-effecting operations. For example, `(i == j++)` is not allowed. In our example below `finished==2` is executable once both P and Q have completed their execution.

```
byte c=0;
byte finished = 0;
proctype P() {
  c++;
  finished++
}

proctype Q() {
  c++;
  finished++
}

init {
  atomic {
    run P();
    run Q()
  };
  finished==2;
  printf("c is %d\n",c)
}
```

Equivalently, one may do the following:

```
byte c=0;

proctype P() {
  c++
}

proctype Q() {
  c++
}

init {
  atomic {
    run P();
    run Q()
  };
  _nr_pr==1;
```

```
  printf("c is %d\n",c)
}
```

However, the following variation does not have the expected outcome. When a process terminates, it can only die and make its _pid number available for the creation of another process, if and when it has the highest _pid number in the system. This means that processes can only die in the reverse order of their creation (in stack order). In summary, a process terminates when it reaches the end of its code; it dies when it has terminated and all processes created after it have died.

```
active proctype P() {
2   printf("A");
}
4
active proctype Q() {
6   printf ("B");
}
8
init {
10   printf("Pr %d",_nr_pr);
    _nr_pr==1;
12    printf("Done")
}
```

termination.pml

For example, consider what happens if we simulate a run:

```
$ spin termination.pml
      A           B                Pr 3       timeout
#processes: 3
  3:     proc  2 (:init::1) termination.pml:11 (state 2)
  3:     proc  1 (Q:1) termination.pml:7 (state 2) <valid end state>
  3:     proc  0 (P:1) termination.pml:3 (state 2) <valid end state>
3 processes created
```

bash

It deadlocks at line 11 (_nr_pr==1) of the file termination.pml. This boolean expression is blocked since processes 0 and 1 cannot terminate until 2 does. If we attempt to verify this program we will obtain an invalid end-state error at line 11.

5.1.4 Inline Definitions

An inline definition works much like a preprocessor macro, in the sense that it just defines a replacement text for a symbolic name, possibly with parameters. The PROMELA parser replaces each point of invocation of an inline with the text of the inline body. For example, the simulation of the following model will produce `Value of a is 1 and b is 1`.

```
inline example(x, y) {
        y = a;
        x = b;
}

init {
```

```
  int a, b;
  a=1;
  b=2;
  example(a,b);
  printf("Value of a is %d and b is %d\n",a,b)
}
```

5.1.5 Record Structures

```
typedef date {
  byte day, month, year;
}
active proctype P() {
  date d;
  d.day = 1;
  d.month = 7;
  d.year = 62
}
```

```
typedef vector {
  int vec[10]
}
active proctype P() {
  vector matrix[5];
  matrix[3].vec[6] = 17;
}
```

5.1.6 Channels

Channels provide a means to model distributed systems where nodes communicate with each other. A channel is a FIFO queue that can be used to exchange messages among processes. Two types of channels are supported: buffered channels and synchronous channels. The latter are also known as rendezvous ports.

5.2 Modeling Semaphores

Semaphores can be modeled in Promela using `inline` definitions. The simplest semaphore to model would be the busy-wait semaphore [BA90, Sec.6.8]. This is modeled in Listing 5.1, with the only difference that our model blocks on `s>0` rather than busy-waiting.

```
byte s=0;

inline acquire(s) {
 atomic {
  s>0;
  s--
 }
}

inline release(s) {
```

```
 s++
}
```

Listing 5.1: Busy-Wait Semaphore (sem.h)

```
#include "sem.h"
byte s=0;

/* AB after CD */
proctype P() {
 acquire(s);
 printf("A");
 printf("B")
}

proctype Q() {
 printf("C");
 printf("D");
 release(s)
}

init {
 atomic {
  run P();
  run Q()
 }
}
```

Problems if you drop the "atomic" in "acquire":

```
int s=1;
int c=0;

inline acquire(s) {
  s>0 -> s--
}

inline release(s) {
 s++
}

proctype P() {
  int temp;
  acquire(s);
  temp=c;
  c=temp+1;
  release(s)
}

init {
 atomic {
  run P(); // Spawn P
  run P()  // Spawn another copy of P
 }
 (_nr_pr==1);
 printf("C is %d ",c)
}
```

```
bool wantP = false;
bool wantQ = false;
byte cs=0;

proctype P() {
 do
  :: wantP = true;
     !wantQ;
     cs++;
     assert (cs==1);
     cs--;
     wantP=false
 od
}

proctype Q() {
 do
  :: wantQ = true;
     !wantP;
     cs++;
     assert (cs==1);
     cs--;
     wantQ=false
 od
}

init {
 atomic {
  run P();
  run Q()
 }
}
```

Figure 5.1: Attempt III in Promela

```
   #include "sem.h"
2  byte ticket=0;
   byte mutex=1;
4
   active [5] proctype Jets() {
6   acquire(mutex);
    acquire(ticket);
8   acquire(ticket)
    release(mutex)
10 }

12 active [5] proctype Patriot() {
    release(ticket);
14 }
```

Figure 5.2: Solution to Bar Problem in Promela

Exercise: would executing lines 7-8 and 18-19 in atomic block avoid deadlock? What about inverting lines 7 and 8 and then placing them in an atomic block (and likewise with lines 18 and 19)?

The semaphores of Chapter 2, known as weak semaphores, behave differently. A `Semaphore.acquire` operation suspends a process when there are no permits; the `Semaphore.release` wakes an arbitrary suspended process or else increments the number of permits if there are none suspended.

5.3 Assertion-Based Model Checking

5.3.1 The Bar Problem Revisited

Listing 5.2 presents the solution to the Bar Problem in Promela. We'll verify that this solution is correct in the sense of upholding the problem invariant, namely that there at least two patriots fans for every jets fan. Before doing so, however, let us first run a simulation of this model.

```bash
> spin bar.pml
      timeout
#processes: 5
        ticket = 0
        mutex = 0
 23:    proc  4 (Jets:1) bar.pml:4 (state 4)
 23:    proc  3 (Jets:1) bar.pml:4 (state 4)
 23:    proc  2 (Jets:1) bar.pml:19 (state 15) <valid end state>
 23:    proc  1 (Jets:1) bar.pml:19 (state 15) <valid end state>
 23:    proc  0 (Jets:1) bar.pml:4 (state 12)
10 processes created
```

bash

The `timeout` indicates that the simulation did not run to completion, it got stuck at a state that is not a valid end state. In other words, it reached a deadlock. From the output above we can see that indeed there are three processes that are deadlocked: 0, 3 and 4. The fact that they are all stuck at line 4 means they are blocked at an acquire. Since there are no available permits in `mutex`, clearly processes 3 and 4 are blocked on the `acquire(mutex)` and 0 at the second `acquire(ticket)`.

A process that terminates must do so after executing its last instruction, otherwise it is said to be in an invalid end state. SPIN checks for this by default. One can insert end state labels to indicate that if execution reaches a certain point and fails to terminate, this should not be considered as an invalid end state. Such valid end state labels must be prefixed with the word end. For example, if we replaced the acquire operation in 5.2 with the following one:

```
inline acquire(permits) {
  skip;
end1:
  atomic {
    permits>0;
    permits--
  }
}
```

then the end states mentioned above are no longer reported as such:

```
> spin bar.pml
      timeout
#processes: 5
        ticket = 0
        mutex = 0
 34:    proc  4 (Jets:1) bar.pml:7 (state 4) <valid end state>
 34:    proc  3 (Jets:1) bar.pml:7 (state 4) <valid end state>
 34:    proc  2 (Jets:1) bar.pml:22 (state 18) <valid end state>
 34:    proc  1 (Jets:1) bar.pml:7 (state 14) <valid end state>
 34:    proc  0 (Jets:1) bar.pml:22 (state 18) <valid end state>
10 processes created
```

bash

Let us get back to the task of verifying that the solution is correct. In order to do so we add two counters. Listing 5.3.1 exhibits the updated code.

```
byte mutex=1;
byte ticket=0;
byte j=0;
byte p=0;

inline acquire(permits) {
  skip;
end1:
  atomic {
    permits>0;
    permits--
  }
}

inline release(permits) {
  permits++
}

active [5] proctype Jets() {
```

```
  acquire(mutex);
  acquire(ticket);
  acquire(ticket);
  release(mutex)
  j++;
  assert (j*2<=p)
}

active [5] proctype Patriots() {
  release(ticket)
  p++;
  assert (j*2<=p)
}
```

We now verify that our solution is correct.

```bash
$ spin -a bar.pml
$ gcc -o pan pan.c
$ ./pan

pan:1: assertion violated ((j*2)<=p) (at depth 34)
pan: wrote bar.pml.trail

(Spin Version 6.5.1 -- 20 December 2019)
Warning: Search not completed
    + Partial Order Reduction

Full statespace search for:
    never claim             - (none specified)
    assertion violations    +
    acceptance   cycles     - (not selected)
    invalid end states  +

State-vector 92 byte, depth reached 47, errors: 1
    18104 states, stored
    18718 states, matched
    36822 transitions (= stored+matched)
        0 atomic steps
hash conflicts:       147 (resolved)

Stats on memory usage (in Megabytes):
    2.072   equivalent memory usage for states (stored*(State-vector + overhead))
    1.071   actual memory usage for states (compression: 51.69%)
            state-vector as stored = 34 byte + 28 byte overhead
  128.000   memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m10000)
  129.511   total actual memory usage

pan: elapsed time 0.02 seconds
pan: rate    905200 states/second
```

bash

It seems that this is not the case since an assertion violation is reported. An inspection of the offending trail shows that when the patriots perform a `release(ticket)` but before incrementing the `p` counter, a jets fan can go in. There are two ways we can fix our code. One is to increment the p counter before performing the release. Another one is to perform the release and increment the counter in one atomic block.

A final remark on atomic blocks. An atomic block that includes blocking statements loses its atomicity and regains it once the statement is enabled and scheduled for execution. As a consequence,

```
acquire(mutex);
acquire(ticket);
acquire(ticket);
release(mutex)
```

is not equivalent to

```
atomic {
  acquire(ticket);
  acquire(ticket)
}
```

5.3.2 The ME Problem

Dekker

Consider the code for Dekker's solution to the ME problem from Fig. **??**. The Promela code is listed in Fig. **??**. We have inserted a variable `cs` to help count when a process enters its critical section. Note how the await in line 12 has been coded as a do-loop: we want this loop to cycle while it waits for the condition to hold.

```
   int turn = 1;
2  boolean wantP = false;
   boolean wantQ = false;
4
   Thread.start { //P
6    while (true) {
      // non-CS
8     wantP = true
      while wantQ
10      if (turn == 2) {
          wantP = false
12        await (turn==1)
          wantP = true
14      }
      // CS
16    turn = 2
      wantP = false
18    // non-CS
     }
20 }

22 Thread.start { //Q
     while (true) {
24    // non-CS
```

```
    wantQ = true
    while wantP
      if (turn == 1) {
        wantQ = false
        await (turn==2)
        wantQ = true
      }
    // CS
    turn = 1
    wantQ = false
    // non-CS
  }
}
```

```
bool wantp = false;
bool wantq = false;
byte turn = 1;
byte cs=0;

active proctype P() {
     do
     ::  wantp = true;
         do
         :: !wantq -> break;
         :: else ->
             if
             :: (turn == 2) ->
                 wantp = false;
                 do
                 :: turn==1 -> break
                 :: else
                 od;
                 wantp = true
             :: else  /* leaves if, if turn<>2 */
             fi
         od;
         cs++;
         assert(cs==1);
         cs--;
         wantp = false;
         turn = 2
     od
}

active proctype Q() {
     do
     ::  wantq = true;
         do
         :: !wantp -> break;
         :: else ->
             if
             :: (turn == 1) ->
                 wantq = false;
                 do
                 :: turn==2 -> break
                 :: else
                 od;
                 wantq = true
```

```
            :: else  /* leaves if, if turn<>2 */
            fi
        od;
        cs++;
        assert(cs==1);
        cs--;
        wantq = false;
        turn = 1
    od
}
```

```
$ spin -a dekker.pml
$ gcc -o pan pan.c
$ ./pan

(Spin Version 6.5.1 -- 20 December 2019)
    + Partial Order Reduction

Full statespace search for:
    never claim             - (none specified)
    assertion violations    +
    acceptance   cycles     - (not selected)
    invalid end states  +

State-vector 28 byte, depth reached 74, errors: 0
      172 states, stored
      173 states, matched
      345 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.009   equivalent memory usage for states (stored*(State-vector + overhead))
    0.287   actual memory usage for states
  128.000   memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m10000)
  128.730   total actual memory usage


unreached in proctype P
    dekker.pml:29, state 28, "-end-"
    (1 of 28 states)
unreached in proctype Q
    dekker.pml:54, state 28, "-end-"
    (1 of 28 states)

pan: elapsed time 0 seconds
```

bash

Binary Semaphores

Listing 5.2 is the PROMELA encoding of the GROOVY code of Listing 2.1. It is easy to verify in SPIN that it enjoys mutex and eventual entry (if one of the two threads wants to enter its CS, one of them will). Note that livelock is not possible since the semaphore operations do not perform busy waiting. Somewhat surprisingly, it fails freedom from starvation. Indeed, if we prefix line 8 with the label `progress1:`, then SPIN will report a fair, non-progress cycle. The reason is that our encoding of semaphores in PROMELA (*c.f.*Listing 5.1) does model the same behavior as the acquire and release operations of the GROOVY semaphores. In particular, line 5 of Listing 5.1, is not always enabled for `P` and hence the scheduler need not select it for execution.

```
   #include "bw_sem.h"
2  byte mutex = 1;

4  proctype P() {
     do
6      :: acquire(mutex);
          /* CS   */
8         release(mutex)
     od
10 }

12 proctype Q() {
     do
14     :: acquire(mutex);
          /* CS */
16        release(mutex)
     od
18 }

20 init {
     atomic {
22     run P();
       run Q()
24   }
   }
```

Listing 5.2: Solution to ME problem using a binary semaphore in PROMELA

Weak (and strong) semaphores, can be modeled using channels. Listing 5.3 presents a PROMELA encoding of weak semaphores [BA90].

```
/* Weak semaphore */
/* NPROCS - the number of processes - must be defined.   */

/* A semaphore is a count plus a channel plus two local variables */
typedef Semaphore {
    byte count;
    chan ch = [NPROCS] of { pid };
    byte temp, i;
};

/* Initialize semaphore to n */
inline initSem(S, n) {
    S.count = n
}

```

```
/* Wait operation: If count is zero, place your _pid in the channel */
/*   and block until it is removed. */
inline acquire(S) {
   atomic {
     if
     :: S.count >= 1 -> S.count--;
     :: else -> S.ch ! _pid; !(S.ch ?? [eval(_pid)])
     fi
   }
}

/* Signal operation: */
/* If there are blocked processes, remove each one and nondeterministically */
/*   decide whether to replace it in the channel or exit the operation. */
inline release(S) {
   atomic {
     S.i = len(S.ch);
     if
     :: S.i == 0 -> S.count++  /* No blocked process, increment count */
     :: else ->
         do
       :: S.i == 1 -> S.ch ? _; break  /* Remove single blocked process */
       :: else ->
           S.i--;
         S.ch ? S.temp;
         if :: break :: S.ch ! S.temp fi
       od
     fi
   }
}
```

Listing 5.3: PROMELA encoding of weak semaphores

Let us assume that the code in Listing 5.3 is placed in a file called `weak_sem_ch.h`. Then replacing lines 1 and 2 in Listing 5.2 with the following code, adding `initSem(mutex, 1);` just after line 20, and then checking for non-progress cycles will now not produce any.

```
#define NPROCS 2
#include "weak_sem_ch.h"
Semaphore mutex;
```

5.3.3 The Feeding Lot Problem Revisited

Consider the Feeding Lot Problem discussed in Exercise **??**:

> A farm breeds cats and dogs. It has a common feeding area for both of them. Although the feeding area can be used by both cats and dogs, it cannot be used by both at the same time for obvious reasons. Provide a solution using semaphores. The solution should be free from deadlock but not necessarily from starvation.

A solution in Promela is given in Listing 5.3.

Exercise 5.3.1. *Show that if lines 20, 28, 44 and 52 are removed, then deadlock is possible. Explain the deadlock situation that can arise.*

```
#include "sem.h"
byte dogs=0;
byte cats=0;
byte mutexDogs=1;
byte mutexCats=1;
byte mutex=1;

active [3] proctype Dog() {
  acquire(mutex);
  acquire(mutexDogs);
  dogs++;
  if
  :: dogs==1 -> acquire(mutexCats);
  :: else -> skip;
  fi
  release(mutexDogs);
  release(mutex);
  // Feed
  acquire(mutexDogs);
  dogs--;
  if
  :: dogs==0 -> release(mutexCats);
  :: else -> skip;
  fi
  release(mutexDogs);
}

active [3] proctype Cat() {
  acquire(mutex);
  acquire(mutexCats);
  cats++;
  if
  :: cats==1 -> acquire(mutexDogs);
  :: else -> skip;
  fi
  release(mutexCats);
  release(mutex);
  // Feed
  acquire(mutexCats);
  cats--;
  if
    :: cats==0 -> release(mutexDogs);
    :: else -> skip;
  fi
  release(mutexCats);
}
```

Figure 5.3: Feeding Lot Problem in Promela

Exercise 5.3.2. *Show, using assertions, that there cannot be felines feeding, if there are dogs feeding and, likewise, there cannot be dogs feeding, if there are felines feeding.*

Exercise 5.3.3. *Show that the following is an alternative solution to the problem by introducing assertions and checking them in Spin.*

```
byte mutexCats=1;
byte mutexDogs=1;
byte mutex=1;
byte resource=1;
byte cats=0;
byte dogs=0;

// Code for acquire and release omitted for brevity

active [3] proctype Cat(){
  acquire(mutex);
  acquire(mutexCats);
  if
  :: cats==0 -> acquire(resource)
  :: else -> skip
  fi;
  cats++;
  release(mutexCats);
  release(mutex);

  acquire(mutexCats);
  cats--;
  if
  :: cats==0 -> release(resource)
  :: else -> skip
  fi;
  release(mutexCats);
}

active [3] proctype Dog(){
  acquire(mutex);
  acquire(mutexDogs);
  if
    :: dogs==0 -> acquire(resource)
    :: else -> skip
  fi;
  dogs++;
  release(mutexDogs);
  release(mutex);

  acquire(mutexDogs);
  dogs--;
  if
    :: dogs==0 -> release(resource)
    :: else -> skip
  fi;
  release(mutexDogs);
}
```

5.3.4 Cyclic Barrier

We would like to verify that our implementation for the cyclic barrier in listing 2.5 is correct. One way to do so is to ensure that no one thread gets "ahead" of any other.

```
#define N 2  // 2 (resp. 3) - requires setting max_depth to 12000 (resp. 22000)
#define B 2

byte mutexE = 1;
byte mutexL = 1;
byte barrier = 0;
byte barrier2 = 0;

byte c[N];
byte enter=0;
byte leaving=0;

inline acquire(s) {
  skip;
end1:atomic {
    s>0;
    s--
  }
}

inline release(s) {
  s++
}

inline absolute(inp,outp) {
  if
    :: inp>0 -> outp = inp
    :: else -> outp = -inp
  fi
}

active[N] proctype P() {
  byte i;
  byte j;
  byte ig;
  int abs;

  for (i: 1..100 ) {
    acquire(mutexE);
    c[_pid]++;
    enter++;
    if
      :: enter==B ->
     for (j: 1 .. B ) {
     release(barrier);
     };
     enter=0
      :: else -> skip
    fi;
    release(mutexE);

    printf("%d reached at cycle %d\n",_pid, c[_pid]);
    acquire(barrier);
```

```
    atomic {
      for (ig: 0..(B-1)) {
      assert (c[_pid]==c[ig])
      }
    };
    printf("%d leaves at cycle %d\n",_pid, c[_pid]);

    acquire(mutexL);
    leaving++;
    if
      :: (leaving==B) ->
     for (j: 1 .. B ) {
     release(barrier2);
     };
     leaving=0
      :: else -> skip
    fi;
    release(mutexL);
    acquire(barrier2);
  }
}
```

Exercise 5.3.4. *Note that the assertion is placed immediately after the* `acquire(barrier)` *line. If we placed it before that line, model checking would fail. Why?*

Exercise 5.3.5. *As a follow up to the previous exercise, how would you modify the assertions so that it may be placed before the* `acquire(barrier)` *line and have the model checking succeed?*

5.4 Non-Progress Cycles

SPIN can check for some simple liveness properties without the need to use Temporal Logic. An infinite computation that does not include infinitely many occurrences of a progress state is called a non-progress cycle. We illustrate this feature by showing that Dekker's algorithm enjoys absence of livelock.

Consider

```
byte x=1;

active proctype P() {

  do
  :: x==1 -> x=2;
  :: x==2 -> x=1;
  od
}
```

Consider

```
byte x=1;

active proctype P() {

  do
  :: x==1 -> x=2;
```

```
  :: x==2 -> progress1: x=1;
  od
}
```

Consider

```
byte x=1;

active proctype P() {

  do
  :: x==1 -> x=2;
  :: x==2 -> progress1: x=1;
  :: x==2 -> x=1;
  od
}
```

We would like to verify that this attempt at solving the ME problem does not enjoy absence of livelock. For that we insert progress labels just before entering the CS.

```
bool wantP=false;
bool wantQ=false;

proctype P() {
  do
    :: wantP=true;
       do
       :: wantQ==false -> break
       :: else
       od;
progress1:
       wantP=false
  od
}

proctype Q() {
  do
    :: wantQ=true;
       do
       :: wantP==false -> break
       :: else
       od;
progress2:
       wantQ=false
  od
}

init {
  atomic {
    run P();
    run Q()
  }
}
```

Selecting Non-Progress in the drop down list and then verifying, SPIN reports a non-progress cycle:

```
2 Q:1    1)   wantQ = 1
```

```
Process Statement          wantQ
1 P:1   1)  wantP = 1      1
Process Statement          wantP      wantQ
2 Q:1   1)  else           1          1
<<<<<START OF CYCLE>>>>>
2 Q:1   1)  else           1          1
1 P:1   1)  else           1          1
2 Q:1   1)  else           1          1
spin: trail ends after 15 steps
```

spin

```
bool wantp = false;
bool wantq = false;
byte turn = 1;

active proctype P() {
    do
    ::  wantp = true;
        do
        :: !wantq -> break;
        :: else ->
            if
            :: (turn == 2) ->
                wantp = false;
                do
                :: turn==1 -> break
                :: else
                od;
                wantp = true
            :: else  /* leaves if, if turn<>2 */
            fi
        od;
progressP:
        wantp = false;
        turn = 2
    od
}

active proctype Q() {
    do
    ::  wantq = true;
        do
        :: !wantp -> break;
        :: else ->
            if
            :: (turn == 1) ->
                wantq = false;
                do
                :: turn==2 -> break
                :: else
                od;
                wantq = true
            :: else  /* leaves if, if turn<>2 */
            fi
        od;
progressQ:
```

```
        wantq = false;
        turn = 1
    od
}
```

"Weak Fairness" should be enabled. Weak fairness means that each statement that becomes enabled and remains enabled thereafter will eventually be scheduled. Consider the example below [?]:

```
byte x=0;

active proctype P() {
  do
  :: true -> x = 1 - x;
  od
}

active proctype Q() {
  do
  :: true -> progress1: x = 1 - x;
  od
}
```

It is possible that `Q` makes no progress if `Q` is never scheduled for execution. Weak fairness guarantees that it eventually will. Verify this in Spin by first enabling weak fairness and then disabling it. In the former case no errors are reported, but in the latter a non-progress cycle is reported:

```
  0 P:1    1)   1
<<<<<START OF CYCLE>>>>>
0 P:1    1)   x = (1-x)
Process Statement          x
0 P:1    1)   1            1
0 P:1    1)   x = (1-x)    1
0 P:1    1)   1            0
```

spin

Consider the code for Attempt IV

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: wantP = true;
     do
     :: wantQ -> wantP = false; wantP = true
     :: else  -> break
     od;
     wantP = false
    od
}

active proctype Q() {
    do
    :: wantQ = true;
```

```
      do
      :: wantP -> wantQ = false; wantQ = true
      :: else -> break
      od;
   wantQ = false
   od
}
```

We know that it does not enjoy freedom from starvation. Freedom from starvation would mean that both `P` and `Q` enter their CS infinitely often. We can verify that it does not enjoy freedom from starvation by inserting a progress label in the critical section of `P`, selecting Non-Progress in the drop down list and then verifying.

```
  bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: wantP = true;
     do
     :: wantQ -> wantP = false; wantP = true
     :: else  -> break
     od;
progress1:
     wantP = false
    od
}

active proctype Q() {
   do
   :: wantQ = true;
      do
      :: wantP -> wantQ = false; wantQ = true
      :: else -> break
      od;
progress2:
   wantQ = false
   od
}
```

Here is the output from Spin

```
1 Q:1   1)  wantQ = 1
Process Statement          wantQ
1 Q:1   1)  else           1
1 Q:1   1)  wantQ = 0      1
0 P:1   1)  wantP = 1      0
Process Statement          wantP      wantQ
1 Q:1   1)  wantQ = 1      1          0
1 Q:1   1)  wantP          1          1
0 P:1   1)  wantQ          1          1
<<<<<START OF CYCLE>>>>>
1 Q:1   1)  wantQ = 0      1          1
1 Q:1   1)  wantQ = 1      1          0
1 Q:1   1)  wantP          1          1
0 P:1   1)  wantP = 0      1          1
1 Q:1   1)  wantQ = 0      0          1
1 Q:1   1)  wantQ = 1      0          0
1 Q:1   1)  else           0          1
```

```
0 P:1    1)   wantP = 1       0           1
0 P:1    1)   wantQ           1           1
1 Q:1    1)   wantQ = 0       1           1
0 P:1    1)   wantP = 0       1           0
1 Q:1    1)   wantQ = 1       0           0
1 Q:1    1)   else            0           1
1 Q:1    1)   wantQ = 0       0           1
0 P:1    1)   wantP = 1       0           0
1 Q:1    1)   wantQ = 1       1           0
1 Q:1    1)   wantP           1           1
Process  Statement            wantP       wantQ
0 P:1    1)   wantQ           1           1
spin: trail ends after 50 steps
```

5.4.1 Weak and Strong Semaphores

In Listing **??** we mentioned that binary semaphores provide a simple solution to the ME problem. Its PROMELA model is given in Listing 5.4. The absence of non-progress cycles fails, attesting that `P` may starve, seemingly contradicting our previous statement that it does constitute a solution to the ME problem. This is due to our PROMELA model of semaphores not coinciding with the weak-semaphores used in Chapter 2.

```
#include "sem.h"
byte sem=1;

proctype P() {
  do
  :: acquire(sem);
progress:
     release(sem)
  od
}

proctype Q() {
  do
  :: acquire(sem);
     release(sem)
  od
}

init {
 atomic {
   run P();
   run Q()
 }
}
```

Listing 5.4: CS implemented with a binary semaphore

Weak semaphores. Strong semaphores.

Chapter 6

Solution to Selected Exercises

Section ??

Answer 6.0.1 (Exercise **??**). *jj*

Section ??

Bibliography

[BA90] M. Ben-Ari. Principles of concurrent and distributed programming. Prentice-Hall, Inc., USA, 1990.

[Car96] Tom Cargill. Specific notification for java thread synchronization. www.dre.vanderbilt.edu/%7Eschmidt/PDF/specific-notification.pdf, 1996.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. Communications of the ACM, 8(9), 1965.

[Har98] Stephen Hartley. Concurrent Programming: The Java Programming Language. Oxford University Press, 1998.

[Lam15] Leslie Lamport. Turing lecturethe computer science of concurrency: the early years. Commun. ACM, 58(6):71–76, may 2015.