

Concurrent Programming Notes

v0.017 – August 28, 2023

Eduardo Bonelli

Preface

These course notes provide supporting material for CS511.

Contents

Preface	i
I Shared Memory	1
1 Shared Memory Model and Transition Systems	3
1.1 Shared Memory Model	3
1.2 Transition Systems	5
1.3 Atomicity	8
1.4 The Mutual Exclusion Problem	10
2 Semaphores	11
2.1 Introduction	11
2.2 The MEP Problem Revisited	11
2.3 More Examples	13
2.3.1 Thread Dumps	14
2.4 Classical Synchronization Problems	16
2.4.1 Producers/Consumers	16
2.4.2 Readers/Writers	18
2.4.3 Barrier Synchronization	18
3 Monitors	23
3.1 A monitor implementing a semaphore	24
3.2 Producers/Consumers	25
3.3 Readers/Writers	26
II Message Passing	31
4 Message Passing in Erlang	33
4.1 Examples	33
4.1.1 Semaphores	33
4.1.2 A Cyclic Barrier	34
4.1.3 Guessing Game	35
4.1.4 Producers/Consumers	36

III	Model Checking	39
5	Promela	41
5.1	Syntax	41
5.1.1	Shared Variable in Promela	41
5.1.2	Examples involving Loops	42
5.1.3	Expressions as blocking commands	43
5.1.4	Macros	44
5.2	Assertion-Based Model Checking	47
5.2.1	The Bar Problem Revisited	47
5.2.2	The MEP Problem	50
5.2.3	The Feeding Lot Problem Revisited	52
5.3	Non-Progress Cycles	54
6	Solution to Selected Exercises	61

Part I

Shared Memory

Chapter 1

Shared Memory Model and Transition Systems

This chapter...

1.1 Shared Memory Model

We begin with an example of a program in Groovy.

```
1  int x = 0
3  Thread.start { //P
    x = 1
5  }
7  Thread.start { //Q
    x = 2
9  }
```

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 threads¹.



Semicolons are optional in Groovy

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

¹From the point of view of Groovy it is actually a local variable. In Groovy, global variables are declared by omitting the type annotation.

```
$ groovy ex1
2 $
```

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`:

```
int x = 0
2
3 P = Thread.start { //P
4     x = 1
5 }
6
7 Q = Thread.start { //Q
8     x = 2
9 }
10
11 P.join() // Wait for P to terminate
12 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:

```
$ groovy ex2
2 2
3 $
```

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.

The following example is a Groovy program that prints characters.

```
1 Thread.start { //P
2     print "A"
3     print "B"
4 }
5
6 Thread.start { //Q
7     print "C"
8 }
```

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?

```
2 Thread.start { //P
3     print "A"
4     print "B"
5 }
6
7 Thread.start { //Q
8     print "C"
9     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, then there are

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

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. We clearly need a rigorous notation to be able to model all such possible interleavings and check whether they satisfy the intended properties. This notation should thus describe the run-time execution of a concurrent program. There is a further, equally important reason, why we need such a notation. Consider the following program:

```
1  int x=0
3  P = Thread.start {
      x = x+1
5  }
7  Q = Thread.start {
      x = x+1
9  }
11 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
2 1
$
```

bash

How is that possible?

1.2 Transition Systems

This section introduces transition systems, a device 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 subset of simple Groovy programs, not arbitrary ones.

A **Transition System** \mathcal{A} is a tuple (S, \rightarrow, 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 $(s, s') \in \rightarrow$.

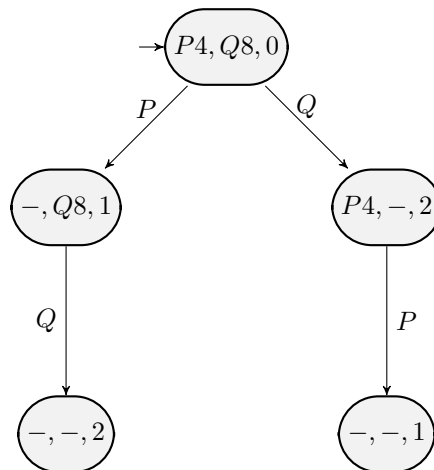
We illustrate, in this first example, how to model the runtime execution of Example 1.1, repeated below:

```

1  int x = 0
3  Thread.start { //P
    x = 1
5  }
7  Thread.start { //Q
    x = 2
9  }

```

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.

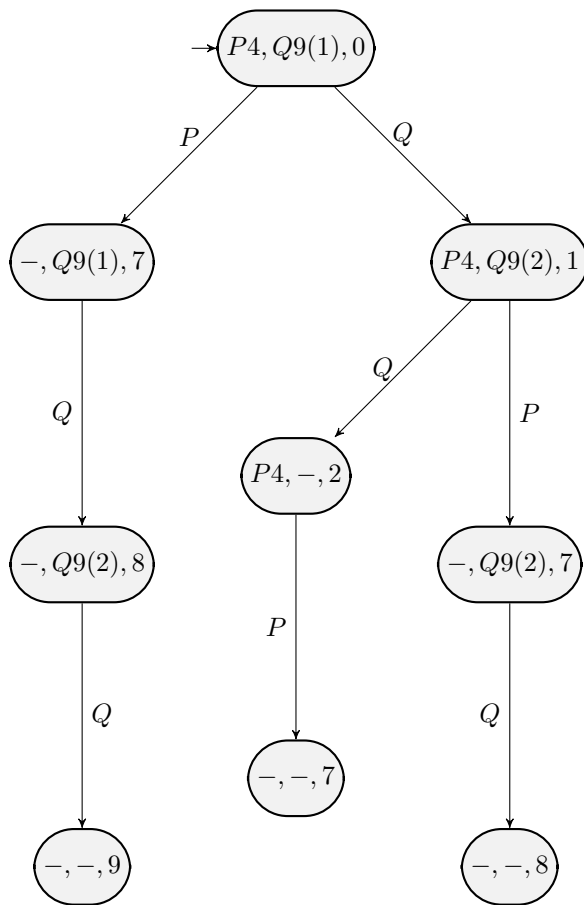


How we hardcode “for”-loops with a constant upper bound.

```

1  int x = 0
2
3  Thread.start { //P
4      x = 7
5  }
6
7  Thread.start { //Q
8      2.times {
9          x = x+1
10     }
11 }

```



How we distinguish local variables with the same name using "local_P" and "local_Q" in the state format.

```

1  int x = 0 // shared variable
3  Thread.start { //P
    int local = x
5   x = local+1 // atomic
  }
7
  Thread.start { //Q
9   int local = x
    x = local+1 // atomic
11  }

```

How we deal with "while"-loops.

```

1  int x=0 // shared variable
3  Thread.start { //P
    while (x<1) {
5     print x
    }

```

```

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

```

How we decorate transitions with the output string of a print

```

1  int x = 0 // shared variable
3  Thread.start { //P
    x = x + 1
5   x = x + 1
   }
7
9  Thread.start { //Q
    while (x >= 2)
        print x
11 }

```

```

1  int counter=0 // shared variable
3  P = Thread.start {
    50.times {
5     counter = counter+1
    }
7  }
9  Q = Thread.start {
    50.times {
        counter= counter+1
11     }
    }
13
15 P.join() // wait for P to finish
   Q.join() // wait for Q to finish
17
19 println counter // print value of counter

```

1.3 Atomicity

Consider the following program:

```

1  x=0
3  Thread.start { //P
    x = x + 1
5     println x
   }
7
9  Thread.start { //Q
    x = x + 1
11     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:

```

1 $ groovy ex3
1
3 1
$

```

bash

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:

```

class A implements Runnable {
2
    static int x=0;
4
    public void run() {
6
        x=x+1;
    }
8
    public static void main(String[] args) {
10
        new Thread(new A()).start();
        new Thread(new A()).start();
12
    }
}

```

A.java

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

```

$ javac A.java
$ javap -c A
Compiled from "A.java"
4 class A implements java.lang.Runnable {
    static int x;
6
    A();
8    Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
        4: return
12
    public void run();
14    Code:
        0: getstatic     #7                  // Field x:I
        3: iconst_1
        4: iadd
        5: putstatic     #7                  // Field x:I
        8: return
16
    public static void main(java.lang.String[]);
20    Code:
        0: new           #13                 // class java/lang/Thread
        3: dup
22

```

```

26      4: new          #8          // class A
      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
40     static {};
      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:

```

2      0: getstatic      #7          // Field x:I
      3: iconst_1
      4: iadd
4      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:

```

2      0(P): getstatic      #7          // Field x:I
      0(Q): getstatic      #7          // Field x:I
      3(P): iconst_1
4      3(Q): iconst_1
      4(P): iadd
6      4(Q): iadd
      5(P): putstatic      #7          // Field x:I
8      5(Q): putstatic      #7          // Field x:I

```

These instructions end up storing 1 in x .

1.4 The Mutual Exclusion Problem

Chapter 2

Semaphores

2.1 Introduction

2.2 The MEP Problem Revisited

Consider the following solution to the MEP problem using a binary semaphore presented in listing 2.1¹.

```
1 Semaphore mutex= new Semaphore(1)
2
3 Thread.start { //P
4     while (true) {
5         mutex.acquire()
6         mutex.release()
7     }
8 }
9 Thread.start { //Q
10    while (true) {
11        mutex.acquire()
12        mutex.release()
13    }
14 }
```

Listing 2.1: Solution to MEP using a binary semaphore

One easy way to verify that all three properties of MEP 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 blocked on

¹Groovy 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.

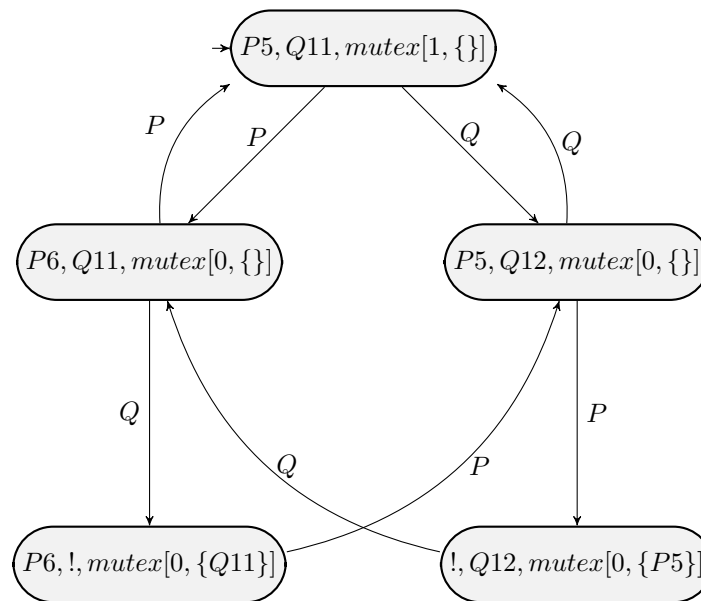


Figure 2.1: Transition System for the solution to MEP using a binary semaphore

`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 MEP for any number of threads.

```

1 Semaphore mutex= new Semaphore(1)
2
3 Thread.start { //P
4     while (true) {
5         mutex.acquire()
6         mutex.release()
7     }
8 }
9 Thread.start { //Q
10    while (true) {
11        mutex.acquire()
12        mutex.release()
13    }

```

```

14 }
15 Thread.start { //R
16     while (true) {
17         mutex.acquire()
18         mutex.release()
19     }
20 }

```

Listing 2.2: Attempt at solving the MEP using a binary semaphore for $N=3$

2.3 More Examples

```

Thread.start {
2     println "A"
    println "B"
4 }
Thread.start {
6     println "C"
    println "D"
8 }

```

```

Semaphore cAfterA = new Semaphore(0)
2
Thread.start {
4     println "A"
    mutex.release()
6     println "B"
}
8 Thread.start {
    mutex.acquire()
10    println "C"
    println "D"
12 }

```

Consider the following example which prints any (infinite) sequence of “a”s and “b”s:

```

Thread.start { //P
2     while (true) {
        print "a"
4     }
}
6
Thread.start { //Q
8     while (true) {
        print "b"
10    }
}

```

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
2

```

```

Semaphore a = new Semaphore(2)
4 Semaphore b = new Semaphore(0)

6 Thread.start { //P
    while (true) {
8         a.acquire()
        print "a"
10        b.release()
    }
12 }

14 Thread.start { //Q
    while (true) {
16        b.acquire(2)
        print "b"
18        a.release(2)
    }
20 }

```

2.3.1 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

2 Semaphore a = new Semaphore(2)
3 Semaphore b = new Semaphore(0)

6 Thread.start { //P
    Thread.currentThread().setName("P Thread");
8    while (true) {
        a.acquire()
10        // print "a"
        b.release()
    }
12 }

14 Thread.start { //Q
16    Thread.currentThread().setName("Q Thread");
    while (true) {
18        b.acquire()
        b.acquire()
20        // print "b"
        a.release()
22        a.release()
    }
24 }

```

Now we run it in the background, use `jstack` to obtain the stack trace of the bash job and send the output to a text file `thead-dump.txt`

```

$ groovy ex1 &
2 [1] 23275
$ jstack -l 23275 > thread-dump.txt

```

```
4 $ kill %1
    [1] + 23275 exit 143    groovy ex1
6 $ emacs thread-dump.txt
```

```
bash
```

The dump contains information on all threads involved in our application. We'll just show an excerpt 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.

```

1  "P Thread" #17 prio=5 os_prio=31 cpu=5910.73ms elapsed=10.91s tid=0x00007f7d9412b400 nid=27655 runnable
2  java.lang.Thread.State: RUNNABLE
3      at jdk.internal.misc.Unsafe.unpark(java.base@18.0.1.1/Native Method)
4      at java.util.concurrent.locks.LockSupport.unpark(java.base@18.0.1.1/LockSupport.java:177)
5      at java.util.concurrent.locks.AbstractQueuedSynchronizer.signalNext(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:432)
6      at java.util.concurrent.locks.AbstractQueuedSynchronizer.releaseShared(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:432)
7      at java.util.concurrent.Semaphore.release(java.base@18.0.1.1/Semaphore.java:432)
8      at java.lang.invoke.LambdaForm$DMH/0x0000000800d28000.invokeVirtual(java.base@18.0.1.1/LambdaForm$DMH/0x0000000800e32c00.invoke(java.base@18.0.1.1/LambdaForm$MH)
9      at java.lang.invoke.LambdaForm$MH/0x0000000800e2b400.guardWithCatch(java.base@18.0.1.1/LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
10     at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder.java:577)
11     at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
12     at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder.java:577)
13     at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
14     at java.lang.invoke.Invokers$Holder.linkToCallSite(java.base@18.0.1.1/Invokers$Holder)
15     at ex1$_run_closure1.doCall(ex1.groovy:12)
16     at ex1$_run_closure1.doCall(ex1.groovy)
17     at java.lang.invoke.DirectMethodHandle$Holder.invokeSpecial(java.base@18.0.1.1/DirectMethodHandle$Holder.java:577)
18     at java.lang.invoke.LambdaForm$MH/0x0000000800c1c800.invoke(java.base@18.0.1.1/LambdaForm$MH)
19     at java.lang.invoke.Invokers$Holder.invokeExact_MT(java.base@18.0.1.1/Invokers$Holder)
20     at jdk.internal.reflect.DirectMethodHandleAccessor.invokeImpl(java.base@18.0.1.1/DirectMethodHandleAccessor.java:577)
21     at jdk.internal.reflect.DirectMethodHandleAccessor.invoke(java.base@18.0.1.1/DirectMethodHandleAccessor.java:577)
22     at java.lang.reflect.Method.invoke(java.base@18.0.1.1/Method.java:577)
23     at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:343)
24     at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:328)
25     at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:279)
26     at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:1009)
27     at groovy.lang.Closure.call(Closure.java:418)
28     at groovy.lang.Closure.call(Closure.java:412)
29     at groovy.lang.Closure.run(Closure.java:500)
30     at java.lang.Thread.run(java.base@18.0.1.1/Thread.java:833)
31
32  Locked ownable synchronizers:
33      - None
34
35  "Q Thread" #18 prio=5 os_prio=31 cpu=6110.53ms elapsed=10.91s tid=0x00007f7d9411fa00 nid=28163 runnable
36  java.lang.Thread.State: WAITING (parking)
37      at jdk.internal.misc.Unsafe.park(java.base@18.0.1.1/Native Method)
38      - parking to wait for <0x00000006180b9960> (a java.util.concurrent.Semaphore$NonfairSync)
39      at java.util.concurrent.locks.LockSupport.park(java.base@18.0.1.1/LockSupport.java:211)
40      at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:432)
41      at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:432)
42      at java.util.concurrent.Semaphore.acquire(java.base@18.0.1.1/Semaphore.java:318)
43      at java.lang.invoke.LambdaForm$DMH/0x0000000800d28000.invokeVirtual(java.base@18.0.1.1/LambdaForm$DMH/0x0000000800e32c00.invoke(java.base@18.0.1.1/LambdaForm$MH)
44      at java.lang.invoke.LambdaForm$MH/0x0000000800e2b400.guardWithCatch(java.base@18.0.1.1/LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
45      at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder.java:577)
46      at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
47      at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder.java:577)
48      at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
49      at java.lang.invoke.Invokers$Holder.linkToCallSite(java.base@18.0.1.1/Invokers$Holder)
50      at ex2$_run_closure1.doCall(ex2.groovy:12)
51      at ex2$_run_closure1.doCall(ex2.groovy)
52      at java.lang.invoke.DirectMethodHandle$Holder.invokeSpecial(java.base@18.0.1.1/DirectMethodHandle$Holder.java:577)
53      at java.lang.invoke.LambdaForm$MH/0x0000000800c1c800.invoke(java.base@18.0.1.1/LambdaForm$MH)
54      at java.lang.invoke.Invokers$Holder.invokeExact_MT(java.base@18.0.1.1/Invokers$Holder)
55      at jdk.internal.reflect.DirectMethodHandleAccessor.invokeImpl(java.base@18.0.1.1/DirectMethodHandleAccessor.java:577)
56      at jdk.internal.reflect.DirectMethodHandleAccessor.invoke(java.base@18.0.1.1/DirectMethodHandleAccessor.java:577)
57      at java.lang.reflect.Method.invoke(java.base@18.0.1.1/Method.java:577)
58      at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:343)
59      at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:328)
60      at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:279)
61      at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:1009)
62      at groovy.lang.Closure.call(Closure.java:418)
63      at groovy.lang.Closure.call(Closure.java:412)
64      at groovy.lang.Closure.run(Closure.java:500)
65      at java.lang.Thread.run(java.base@18.0.1.1/Thread.java:833)
66
67  Locked ownable synchronizers:
68      - None
69
70  "R Thread" #19 prio=5 os_prio=31 cpu=6210.53ms elapsed=10.91s tid=0x00007f7d9411fa00 nid=28163 runnable
71  java.lang.Thread.State: WAITING (parking)
72      at jdk.internal.misc.Unsafe.park(java.base@18.0.1.1/Native Method)
73      - parking to wait for <0x00000006180b9960> (a java.util.concurrent.Semaphore$NonfairSync)
74      at java.util.concurrent.locks.LockSupport.park(java.base@18.0.1.1/LockSupport.java:211)
75      at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:432)
76      at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:432)
77      at java.util.concurrent.Semaphore.acquire(java.base@18.0.1.1/Semaphore.java:318)
78      at java.lang.invoke.LambdaForm$DMH/0x0000000800d28000.invokeVirtual(java.base@18.0.1.1/LambdaForm$DMH/0x0000000800e32c00.invoke(java.base@18.0.1.1/LambdaForm$MH)
79      at java.lang.invoke.LambdaForm$MH/0x0000000800e2b400.guardWithCatch(java.base@18.0.1.1/LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
80      at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder.java:577)
81      at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
82      at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder.java:577)
83      at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
84      at java.lang.invoke.Invokers$Holder.linkToCallSite(java.base@18.0.1.1/Invokers$Holder)
85      at ex3$_run_closure1.doCall(ex3.groovy:12)
86      at ex3$_run_closure1.doCall(ex3.groovy)
87      at java.lang.invoke.DirectMethodHandle$Holder.invokeSpecial(java.base@18.0.1.1/DirectMethodHandle$Holder.java:577)
88      at java.lang.invoke.LambdaForm$MH/0x0000000800c1c800.invoke(java.base@18.0.1.1/LambdaForm$MH)
89      at java.lang.invoke.Invokers$Holder.invokeExact_MT(java.base@18.0.1.1/Invokers$Holder)
90      at jdk.internal.reflect.DirectMethodHandleAccessor.invokeImpl(java.base@18.0.1.1/DirectMethodHandleAccessor.java:577)
91      at jdk.internal.reflect.DirectMethodHandleAccessor.invoke(java.base@18.0.1.1/DirectMethodHandleAccessor.java:577)
92      at java.lang.reflect.Method.invoke(java.base@18.0.1.1/Method.java:577)
93      at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:343)
94      at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:328)
95      at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:279)
96      at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:1009)
97      at groovy.lang.Closure.call(Closure.java:418)
98      at groovy.lang.Closure.call(Closure.java:412)
99      at groovy.lang.Closure.run(Closure.java:500)
100     at java.lang.Thread.run(java.base@18.0.1.1/Thread.java:833)
101
102  Locked ownable synchronizers:
103     - None
104
105  "S Thread" #20 prio=5 os_prio=31 cpu=6310.53ms elapsed=10.91s tid=0x00007f7d9411fa00 nid=28163 runnable
106  java.lang.Thread.State: WAITING (parking)
107     at jdk.internal.misc.Unsafe.park(java.base@18.0.1.1/Native Method)
108     - parking to wait for <0x00000006180b9960> (a java.util.concurrent.Semaphore$NonfairSync)
109     at java.util.concurrent.locks.LockSupport.park(java.base@18.0.1.1/LockSupport.java:211)
110     at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:432)
111     at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(java.base@18.0.1.1/AbstractQueuedSynchronizer.java:432)
112     at java.util.concurrent.Semaphore.acquire(java.base@18.0.1.1/Semaphore.java:318)
113     at java.lang.invoke.LambdaForm$DMH/0x0000000800d28000.invokeVirtual(java.base@18.0.1.1/LambdaForm$DMH/0x0000000800e32c00.invoke(java.base@18.0.1.1/LambdaForm$MH)
114     at java.lang.invoke.LambdaForm$MH/0x0000000800e2b400.guardWithCatch(java.base@18.0.1.1/LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
115     at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder.java:577)
116     at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH)
117     at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMethodHandle$Holder.java:577)
118     at java.lang.invoke.LambdaForm$MH/0x00000
```

```

48  at java.lang.invoke.LambdaForm$MH/0x0000000800e27800.guard(java.base@18.0.1.1/LambdaForm$MH
   at java.lang.invoke.DelegatingMethodHandle$Holder.delegate(java.base@18.0.1.1/DelegatingMet
50  at java.lang.invoke.LambdaForm$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$Holder)
52  at ex1$_run_closure2.doCall(ex1.groovy:19)
   at ex1$_run_closure2.doCall(ex1.groovy)
54  at java.lang.invoke.DirectMethodHandle$Holder.invokeSpecial(java.base@18.0.1.1/DirectMethod
   at java.lang.invoke.LambdaForm$MH/0x0000000800c1c800.invoke(java.base@18.0.1.1/LambdaForm$M
56  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/DirectMeth
58  at jdk.internal.reflect.DirectMethodHandleAccessor.invoke(java.base@18.0.1.1/DirectMethodHa
   at java.lang.reflect.Method.invoke(java.base@18.0.1.1/Method.java:577)
60  at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:343)
   at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:328)
62  at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.jav
   at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:1009)
64  at groovy.lang.Closure.call(Closure.java:418)
   at groovy.lang.Closure.call(Closure.java:412)
66  at groovy.lang.Closure.run(Closure.java:500)
   at java.lang.Thread.run(java.base@18.0.1.1/Thread.java:833)
68
   Locked ownable synchronizers:
70   - 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”.

2.4 Classical Synchronization Problems

This section addresses some classical synchronization problems using semaphores.

2.4.1 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
2
Semaphore consume = new Semaphore(0)
4 Semaphore produce = new Semaphore(1)

6 Thread.start { // Prod
    Random r = new Random()
8     while (true) {
        produce.acquire()
10        buffer = r.nextInt(10000) // produce()
        println "produced "+buffer
12        Thread.sleep(1000)
        consume.release()
14    }
}

16 Thread.start { // Cons
18

```

```

20     while (true) {
21         consume.acquire()
22         println "consumed "+buffer
23         buffer = null // consume(buffer)
24         produce.release()
25     }
26 }

```

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

```

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

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

8 Thread.start { // Prod
10     Random r = new Random()
11     while (true) {
12         produce.acquire()
13         mutexP.acquire()
14         buffer[start] = r.nextInt(10000) // produce()
15         println id+" produced "+buffer[start] + " at index "+start
16         start = (start + 1) % N
17         mutexP.release()
18         consume.release()
19     }
20 }

22 Thread.start { // Cons
23     while (true) {
24         consume.acquire()
25         mutexC.acquire()
26         println id+ " consumed "+buffer[end] + " at index "+end
27         buffer[end] = null // consume(buffer)
28         end = (end + 1) % N
29         mutexC.release()
30         produce.release()
31     }
32 }

```

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
2 Integer[] buffer = [0] * N

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

```

```

8  int start = 0
   int end = 0
10
12  5.times {
13      Thread.start { // Prod
14          Random r = new Random()
15          while (true) {
16              produce.acquire()
17              mutexP.acquire()
18              buffer[start] = r.nextInt(10000) // produce()
19              println Thread.currentThread().getId()+" produced "+buffer[start] + " at index "+start
20              start = (start + 1) % N
21              mutexP.release()
22              consume.release()
23          }
24      }
25
26  5.times{
27      Thread.start { // Cons
28          while (true) {
29              consume.acquire()
30              mutexC.acquire()
31              println Thread.currentThread().getId()+" consumed "+buffer[end] + " at index "+end
32              buffer[end] = null // consume(buffer)
33              end = (end + 1) % N
34              mutexC.release()
35              produce.release()
36          }
37      }
38  }

```

2.4.2 Readers/Writers

2.4.3 Barrier Synchronization

```

// One-time use barrier
// Barrier size = N
// Total number of threads in the system = N
4
final int N=3
6  N.times {
7      Thread.start {
8          while (true) {
9              // barrier arrival protocol
10
11              // barrier
12          }
13      }
14  }

```

```

import java.util.concurrent.Semaphore
2  // One-time use barrier
// Barrier size = N
4  // Total number of threads in the system = N

```



```

6  final int N=3
   int t=0
8  Semaphore barrier = new Semaphore(0)
   Semaphore mutex = new Semaphore(1)
10 N.times {
    Thread.start {
12     while (true) {
        // barrier arrival protocol
14     mutex.acquire()
        if (t<N) {
16         t++
            if (t==N) {
18             N.times { barrier.release() }
            }
20     } else {
        barrier.release()
22     }
        mutex.release()
24     // barrier
        barrier.acquire()
26     }
    }
28 }

```

Using cascaded signalling:

```

import java.util.concurrent.Semaphore
2 // One-time use barrier
   // Barrier size = N
4 // Total number of threads in the system = N

6 final int N=3
   int t=0
8 Semaphore barrier = new Semaphore(0)
   Semaphore mutex = new Semaphore(1)
10 N.times {
    Thread.start {
12     while (true) {
        // barrier arrival protocol
14     mutex.acquire()
        if (t<N) {
16         t++
            if (t==N) {
18             barrier.release()
            }
20     }
        mutex.release()
22     // barrier
        barrier.acquire() // Cascaded signalling
24     barrier.release()
    }
26 }
}

```

Cyclic (or reusable) barrier. Failed attempt:

```

1 import java.util.concurrent.Semaphore

```

```

3 // Cyclic (ie. Reusable) barrier
4 // Barrier size = N
5 // Total number of threads in the system = N
6
7 Semaphore mutex = new Semaphore(1)
8 Semaphore barrier = new Semaphore(0)
9 final int N = 3
10 int t=0
11
12 N.times {
13     Thread.start {
14         while (true) {
15             // arrival
16             mutex.acquire()
17             t++;
18             if (t==N) {
19                 N.times { barrier.release()}
20                 t=0 // attempt to reset barrier counter
21             }
22             mutex.release()
23
24             // barrier
25             barrier.acquire()
26         }
27     }
28 }

```

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.

```

1 import java.util.concurrent.Semaphore
2
3 // Cyclic (ie. Reusable) barrier
4 // Barrier size = N
5 // Total number of threads in the system = N
6
7 Semaphore mutex = new Semaphore(1)
8 Semaphore barrier = new Semaphore(0)
9 Semaphore barrier2 = new Semaphore(0)
10 final int N = 3
11 int t=0
12
13 N.times {
14     int id = it
15     Thread.start {
16         while (true) {
17             // arrival
18             mutex.acquire()
19             t++;
20             if (t==N) {
21                 N.times { barrier.release() }
22             }
23             mutex.release()
24
25             // barrier
26             barrier.acquire()
27
28             // barrier2
29             barrier2.acquire()
30
31             // barrier
32             barrier.acquire()
33
34             // barrier2
35             barrier2.release()
36
37             // barrier
38             barrier.release()
39
40             // barrier2
41             barrier2.release()
42
43             // barrier
44             barrier.acquire()
45
46             // barrier2
47             barrier2.release()
48
49             // barrier
50             barrier.release()
51
52             // barrier2
53             barrier2.release()
54
55             // barrier
56             barrier.acquire()
57
58             // barrier2
59             barrier2.release()
60
61             // barrier
62             barrier.release()
63
64             // barrier2
65             barrier2.release()
66
67             // barrier
68             barrier.acquire()
69
70             // barrier2
71             barrier2.release()
72
73             // barrier
74             barrier.release()
75
76             // barrier2
77             barrier2.release()
78
79             // barrier
80             barrier.acquire()
81
82             // barrier2
83             barrier2.release()
84
85             // barrier
86             barrier.release()
87
88             // barrier2
89             barrier2.release()
90
91             // barrier
92             barrier.acquire()
93
94             // barrier2
95             barrier2.release()
96
97             // barrier
98             barrier.release()
99
100            // barrier2
101            barrier2.release()
102
103            // barrier
104            barrier.acquire()
105
106            // barrier2
107            barrier2.release()
108
109            // barrier
110            barrier.release()
111
112            // barrier2
113            barrier2.release()
114
115            // barrier
116            barrier.acquire()
117
118            // barrier2
119            barrier2.release()
120
121            // barrier
122            barrier.release()
123
124            // barrier2
125            barrier2.release()
126
127            // barrier
128            barrier.acquire()
129
130            // barrier2
131            barrier2.release()
132
133            // barrier
134            barrier.release()
135
136            // barrier2
137            barrier2.release()
138
139            // barrier
140            barrier.acquire()
141
142            // barrier2
143            barrier2.release()
144
145            // barrier
146            barrier.release()
147
148            // barrier2
149            barrier2.release()
150
151            // barrier
152            barrier.acquire()
153
154            // barrier2
155            barrier2.release()
156
157            // barrier
158            barrier.release()
159
160            // barrier2
161            barrier2.release()
162
163            // barrier
164            barrier.acquire()
165
166            // barrier2
167            barrier2.release()
168
169            // barrier
170            barrier.release()
171
172            // barrier2
173            barrier2.release()
174
175            // barrier
176            barrier.acquire()
177
178            // barrier2
179            barrier2.release()
180
181            // barrier
182            barrier.release()
183
184            // barrier2
185            barrier2.release()
186
187            // barrier
188            barrier.acquire()
189
190            // barrier2
191            barrier2.release()
192
193            // barrier
194            barrier.release()
195
196            // barrier2
197            barrier2.release()
198
199            // barrier
200            barrier.acquire()
201
202            // barrier2
203            barrier2.release()
204
205            // barrier
206            barrier.release()
207
208            // barrier2
209            barrier2.release()
210
211            // barrier
212            barrier.acquire()
213
214            // barrier2
215            barrier2.release()
216
217            // barrier
218            barrier.release()
219
220            // barrier2
221            barrier2.release()
222
223            // barrier
224            barrier.acquire()
225
226            // barrier2
227            barrier2.release()
228
229            // barrier
230            barrier.release()
231
232            // barrier2
233            barrier2.release()
234
235            // barrier
236            barrier.acquire()
237
238            // barrier2
239            barrier2.release()
240
241            // barrier
242            barrier.release()
243
244            // barrier2
245            barrier2.release()
246
247            // barrier
248            barrier.acquire()
249
250            // barrier2
251            barrier2.release()
252
253            // barrier
254            barrier.release()
255
256            // barrier2
257            barrier2.release()
258
259            // barrier
260            barrier.acquire()
261
262            // barrier2
263            barrier2.release()
264
265            // barrier
266            barrier.release()
267
268            // barrier2
269            barrier2.release()
270
271            // barrier
272            barrier.acquire()
273
274            // barrier2
275            barrier2.release()
276
277            // barrier
278            barrier.release()
279
280            // barrier2
281            barrier2.release()
282
283            // barrier
284            barrier.acquire()
285
286            // barrier2
287            barrier2.release()
288
289            // barrier
290            barrier.release()
291
292            // barrier2
293            barrier2.release()
294
295            // barrier
296            barrier.acquire()
297
298            // barrier2
299            barrier2.release()
300
301            // barrier
302            barrier.release()
303
304            // barrier2
305            barrier2.release()
306
307            // barrier
308            barrier.acquire()
309
310            // barrier2
311            barrier2.release()
312
313            // barrier
314            barrier.release()
315
316            // barrier2
317            barrier2.release()
318
319            // barrier
320            barrier.acquire()
321
322            // barrier2
323            barrier2.release()
324
325            // barrier
326            barrier.release()
327
328            // barrier2
329            barrier2.release()
330
331            // barrier
332            barrier.acquire()
333
334            // barrier2
335            barrier2.release()
336
337            // barrier
338            barrier.release()
339
340            // barrier2
341            barrier2.release()
342
343            // barrier
344            barrier.acquire()
345
346            // barrier2
347            barrier2.release()
348
349            // barrier
350            barrier.release()
351
352            // barrier2
353            barrier2.release()
354
355            // barrier
356            barrier.acquire()
357
358            // barrier2
359            barrier2.release()
360
361            // barrier
362            barrier.release()
363
364            // barrier2
365            barrier2.release()
366
367            // barrier
368            barrier.acquire()
369
370            // barrier2
371            barrier2.release()
372
373            // barrier
374            barrier.release()
375
376            // barrier2
377            barrier2.release()
378
379            // barrier
380            barrier.acquire()
381
382            // barrier2
383            barrier2.release()
384
385            // barrier
386            barrier.release()
387
388            // barrier2
389            barrier2.release()
390
391            // barrier
392            barrier.acquire()
393
394            // barrier2
395            barrier2.release()
396
397            // barrier
398            barrier.release()
399
400            // barrier2
401            barrier2.release()
402
403            // barrier
404            barrier.acquire()
405
406            // barrier2
407            barrier2.release()
408
409            // barrier
410            barrier.release()
411
412            // barrier2
413            barrier2.release()
414
415            // barrier
416            barrier.acquire()
417
418            // barrier2
419            barrier2.release()
420
421            // barrier
422            barrier.release()
423
424            // barrier2
425            barrier2.release()
426
427            // barrier
428            barrier.acquire()
429
430            // barrier2
431            barrier2.release()
432
433            // barrier
434            barrier.release()
435
436            // barrier2
437            barrier2.release()
438
439            // barrier
440            barrier.acquire()
441
442            // barrier2
443            barrier2.release()
444
445            // barrier
446            barrier.release()
447
448            // barrier2
449            barrier2.release()
450
451            // barrier
452            barrier.acquire()
453
454            // barrier2
455            barrier2.release()
456
457            // barrier
458            barrier.release()
459
460            // barrier2
461            barrier2.release()
462
463            // barrier
464            barrier.acquire()
465
466            // barrier2
467            barrier2.release()
468
469            // barrier
470            barrier.release()
471
472            // barrier2
473            barrier2.release()
474
475            // barrier
476            barrier.acquire()
477
478            // barrier2
479            barrier2.release()
479

```

```

25     // barrier
    barrier.acquire()

27     mutex.acquire()
    t--
29     if (t==0) {
31         N.times { barrier2.release() }
    }
33     mutex.release()

35     barrier2.acquire()
    }
37 }

39 }

1  import java.util.concurrent.Semaphore

3  // Cyclic (ie. Reusable) barrier
  // Barrier size = N
5  // Total number of threads in the system = N

7  Semaphore mutex = new Semaphore(1)
  Semaphore barrier = new Semaphore(0)
9  Semaphore barrier2 = new Semaphore(0)
  final int N = 3
11 int t=0
  int[] c = new int[N]

13 N.times {
15     int id = it
    Thread.start {
17         1000.times { //while (true) {
            // arrival
19             mutex.acquire()
            c[id]++
21             t++
            if (t==N) {
23                 N.times { barrier.release() }
            }
25             mutex.release()

27             // barrier
            println id + " reached barrier. c="+c[id]
29             barrier.acquire()
            println id + " passed barrier. c="+c[id]

31             mutex.acquire()
33             t--
            if (t==0) {
35                 N.times { barrier2.release() }
            }
37             mutex.release()

39             barrier2.acquire()
        }
41     }
}

```

43 }

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 global 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. The other thread will block when it attempts to acquire the lock.

```
// Monitor declaration
2  class Counter {
    private int c
4
    public synchronized void inc() {
6        c++
    }
8
    public synchronized void dec() {
10       c--
    }
12
    public synchronized int read() {
14       return c
    }
16 }

18 // Sample use of the monitor
Counter ctr = new Counter()
20
22 P = Thread.start {
    10.times {
24         ctr.inc()
    }
26 }
```

```

Q = Thread.start {
28     10.times {
        ctr.inc()
30     }
    }
32
P.join()
34 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

```

1  class Semaphore {
    private int permits
3
    Semaphore(int init) {
5        permits=init
    }
7
    public synchronized void acquire() {
9        while (permits==0) {
            wait()
11       }
        permits--
13    }
15
    public synchronized void release() {
        notify()
17        permits++
    }
19 }
21
Semaphore mutex = new Semaphore(1)
int c=0
23
P = Thread.start {
25     10.times {
        mutex.acquire()
27         c++
        mutex.release()
29     }
    }
31
Q = Thread.start {
33     10.times {
        mutex.acquire()
35         c++
        mutex.release()
37     }
    }
39
P.join()
41 Q.join()
println c

```

This solution is not fair on the threads that are sleeping since an outside thread could “steal” the permit what is made available through a call to `release`. An alternative that is fair in this sense¹ is given in Listing 3.2. A different approach is followed in [Car96].

```

import java.util.concurrent.Semaphore

2
class Semaphore {
4     private int permits;
     private long startWaitingTime=0;
6     private static final long startTime = System.currentTimeMillis();
     private int waiting=0;

8
     Semaphore(int n) {
10         permits=n;
     }

12
     synchronized protected static final long age() {
14         return System.currentTimeMillis() - startTime;
     }

16
     synchronized void acquire() {
18         if (waiting>0 || permits==0) {
             long arrivalTime = age();
20             while (arrivalTime>startWaitingTime || permits==0) {
                 waiting++;
22                 wait();
                 waiting--;
24             }
         }
26         permits--;
     }

28
     synchronized void release() {
30         permits++;
         startWaitingTime = age();
32         notify();
     }
34 }

```

Listing 3.2: Fair semaphores

3.2 Producers/Consumers

```

class PC {
2     private Object buffer;

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

```

¹The 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).

```

10     }
12     public synchronized Object consume() {
13         while (buffer==null) {
14             wait()
15         }
16         Object temp = buffer
17         buffer=null
18         notifyAll()
19         return temp
20     }
21 }
22 PC pc = new PC()
23
24 10.times {
25     Thread.start {
26         println (Thread.currentThread().getId()+" consumes")
27         pc.consume()
28     }
29 }
30
31 10.times {
32     Thread.start {
33         println (Thread.currentThread().getId()+" produces")
34         pc.produce((new Random()).nextInt(33))
35     }
36 }

```

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

Naive solution. Correct but unfair on writers.

```

import java.util.concurrent.locks.*
2
class RW {
4     private int readers;
5     private int writers;
6     static final Lock lock = new ReentrantLock();
7     static final Condition okToRead = lock.newCondition();
8     static final Condition okToWrite = lock.newCondition();
9
10    RW() {
11        readers=0;
12        writers=0;
13    }
14
15    void start_read() {
16        lock.lock();
17        try {

```



```
18     while (writers>0) {
19         okToRead.await();
20     }
21     readers++;
22 } finally {
23     lock.unlock();
24 }
25 }
26
27 void stop_read() {
28     lock.lock();
29     try {
30         readers--;
31         if (readers==0) {
32             okToWrite.signal();
33         }
34     } finally {
35         lock.unlock();
36     }
37 }
38
39 void start_write(Object item) {
40     lock.lock();
41     try {
42         while (readers>0 || writers>0) {
43             okToWrite.await();
44         }
45         writers++;
46     } finally {
47         lock.unlock();
48     }
49 }
50
51 void stop_write() {
52     lock.lock();
53     try {
54         writers--;
55         okToWrite.signal();
56         okToRead.signalAll();
57     } finally {
58         lock.unlock();
59     }
60 }
61
62 }
63
64 RW rw = new RW();
65
66 r = { //R
67     Random r = new Random();
68     rw.start_read();
69     println Thread.currentThread().getId()+" reading..."
70     Thread.sleep(r.nextInt(1000));
71     println Thread.currentThread().getId()+" done reading..."
72
73     rw.stop_read();
74 }
```

```

76 w = { //W
    Random r = new Random();
78    rw.start_write();
    println Thread.currentThread().getId()+" writing..."
80    Thread.sleep(r.nextInt(1000));
    println Thread.currentThread().getId()+" done writing..."
82    rw.stop_write();
    }
84
200.times {
86    Thread.start(r)
    Thread.start(w)
88 }

```

Checking for waiting writers. Places priority on writers but unfair on readers.

```

import java.util.concurrent.locks.*

2
class RW {
4    private int readers;
    private int writers;
6    private int writers_waiting;
    static final Lock lock = new ReentrantLock();
8    static final Condition okToRead = lock.newCondition();
    static final Condition okToWrite = lock.newCondition();

10
    RW() {
12        readers=0;
        writers=0;
14        writers_waiting=0;
    }

16
    void start_read() {
18        lock.lock();
        try {
20            while (writers>0 || writers_waiting>0) {
                okToRead.await();
22            }
            readers++;
24        } finally {
            lock.unlock();
26        }
    }

28
    void stop_read() {
30        lock.lock();
        try {
32            readers--;
            if (readers==0) {
34                okToWrite.signal();
            }
36        } finally {
            lock.unlock();
38        }
    }

40
    void start_write(Object item) {
42        lock.lock();

```

```

44     try {
45         while (readers>0 || writers>0) {
46             writers_waiting++;
47             okToWrite.await();
48             writers_waiting--;
49             writers++;
50         } finally {
51             lock.unlock();
52         }
53     }
54
55     void stop_write() {
56         lock.lock();
57         try {
58             writers--;
59             okToWrite.signal();
60             okToRead.signalAll();
61         } finally {
62             lock.unlock();
63         }
64     }
65 }

```

An attempt at a fair solution to RW is presented in Listing 3.3. One situation that may lead to deadlock is: W1,R1,W2. Another is: R1, W1, R2.

```

1  import java.util.concurrent.locks.*
2
3  class RW {
4      private int readers;
5      private int writers;
6      private int writers_waiting;
7      private int readers_waiting;
8      static final Lock lock = new ReentrantLock();
9      static final Condition okToRead = lock.newCondition();
10     static final Condition okToWrite = lock.newCondition();
11
12     RW() {
13         readers=0;
14         writers=0;
15         writers_waiting=0;
16         readers_waiting=0;
17     }
18
19     void start_read() {
20         lock.lock();
21         try {
22             while (writers>0 || writers_waiting>0) {
23                 readers_waiting++;
24                 okToRead.await();
25                 readers_waiting--;
26             }
27             readers++;
28         } finally {
29             lock.unlock();
30         }
31     }

```

```

33     void stop_read() {
34         lock.lock();
35         try {
36             readers--;
37             if (readers==0) {
38                 okToWrite.signal();
39             }
40         } finally {
41             lock.unlock();
42         }
43     }

45     void start_write(Object item) {
46         lock.lock();
47         try {
48             while (readers>0 || writers>0 || readers_waiting>0) {
49                 writers_waiting++;
50                 okToWrite.await();
51                 writers_waiting--;
52             }
53             writers++;
54         } finally {
55             lock.unlock();
56         }
57     }

59     void stop_write() {
60         lock.lock();
61         try {
62             writers--;
63             okToWrite.signal();
64             okToRead.signalAll();
65         } finally {
66             lock.unlock();
67         }
68     }
69 }

```

Listing 3.3: 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.

```

2     void stop_write() {
3         lock.lock();
4         try {
5             writers--;
6             if (readers_waiting==0) {
7                 okToWrite.signal();
8             } else {
9                 okToRead.signalAll();
10            }
11        } finally {
12            lock.unlock();
13        }
14    }

```

Part II

Message Passing

Chapter 4

Message Passing in Erlang

4.1 Examples

4.1.1 Semaphores

```
1 -module(sem).
2 -compile(nowarn_export_all).
3 -compile(export_all).
4
5 make(N) ->
6     spawn(?MODULE, sem_loop, [N]).
7
8 acquire(S) ->
9     S!{acquire, self()},
10    receive
11    {ok} ->
12        ok
13    end.
14
15 release(S) ->
16     S!{release}.
17
18 sem_loop(0) -> %% no permits available
19    receive
20    {release} ->
21        sem_loop(1)
22    end;
23 sem_loop(N) when N>0 -> %% permits available
24    receive
25    {acquire, From} ->
26        From ! {ok},
27        sem_loop(N-1);
28    {release} ->
29        sem_loop(N+1)
30    end.
```

sem.erl

```
-module(semcl).
```

```

2 -compile(nowarn_export_all).
  -compile(export_all).

4
start() ->
6     S = sem:make(0),
      spawn(?MODULE,client1,[S]),
8       spawn(?MODULE,client2,[S]),
        ok.

10
client1(S) ->
12     sem:acquire(S),
      io:format("a"),
14     io:format("b").

16
client2(S) ->
      io:format("c"),
18     io:format("d"),
      sem:release(S).

```

semcl.erl

4.1.2 A Cyclic Barrier

```

1 -module(barr).
  -compile(nowarn_export_all).
3 -compile(export_all).

5
make(N) ->
      spawn(?MODULE,coordinator,[N,N,[]]).
7
reached(B) ->
9     B!{reached,self()},
      receive
11     ok ->
        ok
13     end.

15 % coordinator(N,M,L)
    % N is the size of the barrier
17 % M is the number of processes YET to arrive at the barrier
    % L is a list of the PIDs of the processes that have already arrived at the barrier
19
coordinator(N,0,L) ->
21     [ PID!ok || PID <- L ],
      coordinator(N,N,[]);
23 coordinator(N,M,L) when M>0 ->
      receive
25     {reached,From} ->
        coordinator(N,M-1,[From|L])
27     end.

```

barr.erl

```

1 -module(barrcl).
  -compile(nowarn_export_all).
3 -compile(export_all).

```



```

5 start() ->
    B = barr:make(3),
7     spawn(?MODULE,client1,[B]),
    spawn(?MODULE,client2,[B]),
9     spawn(?MODULE,client3,[B]),
    ok.
11
12 client1(B) ->
13     io:format("a"),
    barr:reached(B),
15     io:format("1"),
    client1(B).
17
18 client2(B) ->
19     io:format("b"),
    barr:reached(B),
21     io:format("2"),
    client2(B).
23
24 client3(B) ->
25     io:format("c"),
    barr:reached(B),
27     io:format("3"),
    client3(B).

```

barrcl.erl

4.1.3 Guessing Game

```

-module(gg).
2 -compile(nowarn_export_all).
  -compile(export_all).
4
5 start() ->
6     S = spawn(?MODULE,server_loop,[]),
    [ spawn(?MODULE,client,[S]) || _ <- lists:seq(1,100)].
8
9 client(S) ->
10     S!{self(),start},
    receive
12     {ok,Servlet} ->
        client_loop(Servlet,rand:uniform(100))
14     end.
15
16 client_loop(Servlet, G) ->
    Servlet!{G,self()},
18     receive
    {youGotIt,T} ->
20         io:format("~w got it in ~w tries~n",[self(),T]);
    {tryAgain} ->
22         client_loop(Servlet,rand:uniform(100))
    end.
24
25 server_loop() ->
26     receive
    {From,start} ->
28         Servlet = spawn(?MODULE,servlet,[rand:uniform(100),0]),

```

```

    From!{ok,ServLet},
    server_loop()
end.

32
servlet(N,T) ->
34     receive
    {Guess,From} when Guess==N ->
36         From!{youGotIt,T};
    {Guess,From} when Guess/=N ->
38         From!{tryAgain},
            servlet(N,T+1)
40 end.

```

gg.erl

4.1.4 Producers/Consumers

```

-module(pc).
2  -compile(nowarn_export_all).
  -compile(export_all).
4
start(Cap,NofP,NofC) ->
6     RS = spawn(?MODULE,resource,[0,Cap,0,0]),
    [ spawn(?MODULE,producer,[RS]) || _ <- lists:seq(1,NofP)],
8     [ spawn(?MODULE,consumer,[RS]) || _ <- lists:seq(1,NofC)],
    ok.
10
%% client code
12 producer(RS) ->
    startProduce(RS),
14     %% produce
    timer:sleep(rand:uniform(100)),
16     stopProduce(RS).

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

24 %% PC code
startProduce(RS) ->
26     RS!{startProduce,self()},
    receive
28     {ok} ->
        ok
30     end.

32 stopProduce(RS) ->
    RS!{stopProduce}.
34
startConsume(RS) ->
36     RS!{startConsume,self()},
    receive
38     {ok} ->
        ok
40     end.

```

```
42 stopConsume(RS) ->
    RS!{stopConsume}.
44
46 resource(Size, Cap, SP, SC) ->
    receive
    {startProduce, From} when Size + SP < Cap ->
48         From!{ok},
        resource(Size, Cap, SP+1, SC);
    {stopProduce} ->
50         resource(Size+1, Cap, SP-1, SC);
    {startConsume, From} when Size - SC > 0 ->
52         From!{ok},
        resource(Size, Cap, SP, SC+1);
    {stopConsume} ->
54         resource(Size-1, Cap, SP, SC-1)
56 end.
```

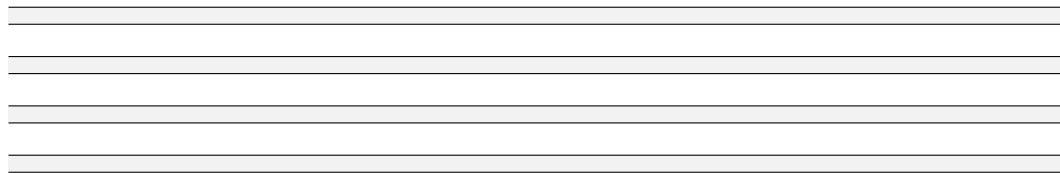
pc.erl

Part III

Model Checking

Chapter 5

Promela



5.1 Syntax

We begin with a brief introduction to Promela through a series of examples.

5.1.1 Shared Variable in Promela

```
1 byte n=0;
3 active proctype P() {
    n=1;
5     printf("P has pid %d. n=%d\n",_pid,n)
    };
7
9 active proctype Q() {
    n=2;
    printf("Q has pid %d. n=%d\n",_pid,n)
11 }
```

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

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

bash

Each process is 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 suppress indentation.

```
$ spin -T eg.pml
2 P has pid 0. n=1
  Q has pid 1. n=1
4 2 processes created
```

bash

Semicolon is a separator, not a terminator.

5.1.2 Examples involving Loops

```
byte sum=0;
2
active proctype P() {
4   byte i=0;
   do
6     :: i>10 -> break
     :: else ->
8       sum = sum + i;
       i++
10  od;
12  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.

```
1 byte i=0;
3
active proctype P() {
   do
5     :: i++;
       printf("Value of i: %d\n. ",i)
7   od
}
```

An example using a for loop:

```
byte sum=0;
2
active proctype P() {
4   byte i;
   for (i:1..10) {
6     sum = sum + i
   }
8
10  printf("The sum of the first 10 numbers is %d\n",sum)
}
```


5.1.3 Expressions as blocking commands

```

byte c=0;
2 finished = 0;
proctype P() {
4   c++;
   finished++;
6 }

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

12 init {
14   atomic {
       run P();
16   run Q();
   };
18   finished==2;
   printf("c is %d\n",c)
20 }

```

Equivalently, one may do the following:

```

byte c=0;
2
proctype P() {
4   c++
}

6
proctype Q() {
8   c++
}

10
init {
12   atomic {
       run P();
14   run Q();
   };
16   _nr_pr==1;
   printf("c is %d\n",c)
18 }

```

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).

```

active proctype P() {
2   printf("A");
}

4
active proctype Q() {
6   printf("B");
}

8

```

```

init {
10  printf("Pr %d", _nr_pr);
    _nr_pr++;
12  printf("Done")
}

```

termination.pml

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

```

1  $ spin termination.pml
    A          B          Pr 3          timeout
3  #processes: 3
    3:   proc  2 (:init::1) termination.pml:11 (state 2)
5    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>
7  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 Macros

Semaphores can be modeled in Promela using an `inline` definition. 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.

```

byte s=0;
2
inline acquire(s) {
4  atomic {
    s>0 -> s--
6  }
}
8
inline release(s) {
10 s++
}
12
/* AB after CD */
14 proctype P() {
    acquire(s);
16  printf("A");
    printf("B")
18 }
20 proctype Q() {
    printf("C");
22  printf("D");
    release(s)
24 }
26
init {
    atomic {

```

```

28   run P();
    run Q()
30 }
}

```

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

```

1  byte s=1;
   byte c=0;

3
   inline acquire(s) {
5     s>0 -> s--
   }

7
   inline release(s) {
9     s++
   }

11
   proctype P() {
13     acquire(s);
        c++;
15 }

17
   proctype Q() {
        acquire(s);
19     c++;
   }

21 // /\ AB after CD */
   /* proctype P() { */
23 /* acquire(s); */
   /* printf("A"); */
25 /* printf("B") */

27 /* } */

29 /* proctype Q() { */

31 /* printf("C"); */
   /* printf("D"); */
33 /* release(s) */
   /* } */

35
   init {
37     atomic {
        run P();
39     run Q()
   }
41     (_nr_pr==1);
        printf("C is %d ",c)
43 }
}

```

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)?

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

4
proctype P() {
6   do
      :: wantP = true;
8     !wantQ;
      cs++;
10    assert (cs==1);
      cs--;
12    wantP=false
   od
14 }

16 proctype Q() {
   do
18     :: wantQ = true;
      !wantP;
20     cs++;
      assert (cs==1);
22     cs--;
      wantQ=false
24   od
   }
26

28 init {
   atomic {
30     run P();
      run Q()
   }
32 }
```

Figure 5.1: Attempt III in Promela

```

byte ticket=0;
2 byte mutex=1;

4 inline acquire(sem) {
    atomic {
6         sem>0 -> sem--
    }
8 }

10 inline release(sem) {
    sem++
12 }

14 active [5] proctype Jets() {
    acquire(mutex);
16     acquire(ticket);
    acquire(ticket)
18     release(mutex)
}

20
22 active [5] proctype Patriot() {
    release(ticket);
}

```

Figure 5.2: Solution to Bar Problem in Promela

5.2 Assertion-Based Model Checking

5.2.1 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.

```

1 > spin bar.pml
    timeout
3 #processes: 5
    ticket = 0
5     mutex = 0
23:    proc 4 (Jets:1) bar.pml:4 (state 4)
7     23:    proc 3 (Jets:1) bar.pml:4 (state 4)
    23:    proc 2 (Jets:1) bar.pml:19 (state 15) <valid end state>
9     23:    proc 1 (Jets:1) bar.pml:19 (state 15) <valid end state>
    23:    proc 0 (Jets:1) bar.pml:4 (state 12)
11 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:

```

1 inline acquire(permits) {
2     skip;
3 end1:
4     atomic {
5         permits>0;
6         permits--
7     }
8 }

```

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

```

> spin bar.pml
2     timeout
#processes: 5
4     ticket = 0
    mutex = 0
6 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>
8 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>
10 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 5.2.1 exhibits the updated code.

```

1 byte mutex=1;
2 byte ticket=0;
3 byte j=0;
4 byte p=0;
5
6 inline acquire(permits) {
7     skip;
8 end1:
9     atomic {
10         permits>0;
11         permits--
12     }
13 }
14
15 inline release(permits) {
16     permits++
17 }
18
19 active [5] proctype Jets() {
20
21     acquire(mutex);

```

```

23   acquire(ticket);
24   acquire(ticket);
25   release(mutex)
26   j++;
27   assert (j*2<=p)
28 }
29 active [5] proctype Patriots() {
30   release(ticket)
31   p++;
32   assert (j*2<=p)
33 }

```

We now verify that our solution is correct.

```

1  $ spin -a bar.pml
2  $ gcc -o pan pan.c
3  $ ./pan

5  pan:1: assertion violated ((j*2)<=p) (at depth 34)
6  pan: wrote bar.pml.trail
7
8  (Spin Version 6.5.1 -- 20 December 2019)
9  Warning: Search not completed
10         + Partial Order Reduction
11
12  Full statespace search for:
13      never claim          - (none specified)
14      assertion violations  +
15      acceptance cycles    - (not selected)
16      invalid end states   +
17
18  State-vector 92 byte, depth reached 47, errors: 1
19      18104 states, stored
20      18718 states, matched
21      36822 transitions (= stored+matched)
22      0 atomic steps
23  hash conflicts:          147 (resolved)
24
25  Stats on memory usage (in Megabytes):
26      2.072   equivalent memory usage for states (stored*(State-vector + overhead))
27      1.071   actual memory usage for states (compression: 51.69%)
28              state-vector as stored = 34 byte + 28 byte overhead
29      128.000 memory used for hash table (-w24)
30      0.534   memory used for DFS stack (-m10000)
31      129.511 total actual memory usage
32
33  pan: elapsed time 0.02 seconds
34  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.

5.2.2 The MEP Problem

Consider the code for Dekker's solution to the MEP 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
26        while wantP
            if (turn == 1) {
28              wantQ = false
              await (turn==2)
30              wantQ = true
            }
32        // CS
          turn = 1
34          wantQ = false
          // non-CS
36        }
      }

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

5
active proctype P() {
7   do

```



```

9      :: wantp = true;
      do
10         :: !wantq -> break;
11         :: else ->
            if
12             :: (turn == 2) ->
                wantp = false;
13             do
14                 :: turn==1 -> break
15                 :: else
16                 od;
17                 wantp = true
18             :: else /* leaves if, if turn<>2 */
19             fi
20         od;
21         cs++;
22         assert(cs==1);
23         cs--;
24         wantp = false;
25         turn = 2
26     od
27 }

31 active proctype Q() {
32     do
33         :: wantq = true;
34         do
35             :: !wantp -> break;
36             :: else ->
37                 if
38                     :: (turn == 1) ->
39                     wantq = false;
40                     do
41                         :: turn==2 -> break
42                         :: else
43                         od;
44                     wantq = true
45                 :: else /* leaves if, if turn<>2 */
46                 fi
47             od;
48             cs++;
49             assert(cs==1);
50             cs--;
51             wantq = false;
52             turn = 1
53         od
54     }

```

```

$ spin -a dekker.pml
2 $ gcc -o pan pan.c
$ ./pan

4
(Spin Version 6.5.1 -- 20 December 2019)
6   + Partial Order Reduction

8 Full statespace search for:

```

```

10      never claim          - (none specified)
      assertion violations  +
12      acceptance cycles   - (not selected)
      invalid end states   +

14 State-vector 28 byte, depth reached 74, errors: 0
      172 states, stored
16      173 states, matched
      345 transitions (= stored+matched)
18      0 atomic steps
hash conflicts:          0 (resolved)

20 Stats on memory usage (in Megabytes):
22      0.009   equivalent memory usage for states (stored*(State-vector + overhead))
      0.287   actual memory usage for states
24      128.000 memory used for hash table (-w24)
      0.534   memory used for DFS stack (-m10000)
26      128.730 total actual memory usage

28 unreachable in proctype P
30      dekker.pml:29, state 28, "-end-"
      (1 of 28 states)
32 unreachable in proctype Q
      dekker.pml:54, state 28, "-end-"
34      (1 of 28 states)

36 pan: elapsed time 0 seconds

```

bash

5.2.3 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 5.2.1. Show that if lines 20, 28, 44 and 52 are removed, then deadlock is possible. Explain the deadlock situation that can arise.

Exercise 5.2.2. 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 5.2.3. Show that the following is an alternative solution to the problem by introducing assertions and checking them in Spin.

```

byte dogs=0;
2 byte cats=0;
byte mutexDogs=1;
4 byte mutexCats=1;
byte mutex=1;

6
inline acquire(sem) {
8   atomic {
sem>0;
10   sem--
}
12 }

14 inline release(sem) {
sem++
16 }

18 active [3] proctype Dog() {
acquire(mutex);
20 acquire(mutexDogs);
dogs++;
22 if
:: dogs==1 -> acquire(mutexCats);
24 :: else -> skip;
fi
26 release(mutexDogs);
release(mutex);
28 // Feed
acquire(mutexDogs);
30 dogs--;
if
32 :: dogs==0 -> release(mutexCats);
:: else -> skip;
34 fi
release(mutexDogs);
36 }

38 active [3] proctype Cat() {
acquire(mutex);
40 acquire(mutexCats);
cats++;
42 if
:: cats==1 -> acquire(mutexDogs);
44 :: else -> skip;
fi
46 release(mutexCats);
release(mutex);
48 // Feed
acquire(mutexCats);
50 cats--;
if
52 :: cats==0 -> release(mutexDogs);
:: else -> skip;
54 fi
release(mutexCats);
56 }

```

Figure 5.3: Feeding Lot Problem in Promela

```

byte mutexCats=1;
2 byte mutexDogs=1;
byte mutex=1;
4 byte resource=1;
byte cats=0;
6 byte dogs=0;

8 // Code for acquire and release omitted for brevity

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

20    acquire(mutexCats);
22    cats--;
    if
24    :: cats==0 -> release(resource)
    :: else -> skip
26    fi;
    release(mutexCats);
28 }

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

40    acquire(mutexDogs);
42    dogs--;
    if
44    :: dogs==0 -> release(resource)
    :: else -> skip
46    fi;
    release(mutexDogs);
48 }

```

5.3 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

```

1 byte x=1;
2
3 active proctype P() {
4
5     do
6         :: x==1 -> x=2;
7         :: x==2 -> x=1;
8     od
9 }

```

Consider

```

1 byte x=1;
2
3 active proctype P() {
4
5     do
6         :: x==1 -> x=2;
7         :: x==2 -> progress1: x=1;
8     od
9 }

```

Consider

```

1 byte x=1;
2
3 active proctype P() {
4
5     do
6         :: x==1 -> x=2;
7         :: x==2 -> progress1: x=1;
8         :: x==2 -> x=1;
9     od
10 }

```

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

```

1 bool wantP=false;
2 bool wantQ=false;
3
4 proctype P() {
5     do
6         :: wantP=true;
7         do
8             :: wantQ==false -> break
9             :: else
10                od;
11 progress1:
12         wantP=false
13     od
14 }
15
16 proctype Q() {
17     do

```

```

18  :: wantQ=true;
    do
20  :: wantP==false -> break
    :: else
22  od;
progress2:
24  wantQ=false
    od
26  }

28  init {
    atomic {
30  run P();
    run Q()
32  }
}

```

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

```

1  2 Q:1  1) wantQ = 1
   Process Statement      wantQ
3  1 P:1  1) wantP = 1    1
   Process Statement      wantP      wantQ
5  2 Q:1  1) else        1          1
   <<<<<START OF CYCLE>>>>>
7  2 Q:1  1) else        1          1
   1 P:1  1) else        1          1
9  2 Q:1  1) else        1          1
   spin: trail ends after 15 steps

```

spin

```

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

4
active proctype P() {
6  do
    :: wantp = true;
8  do
    :: !wantq -> break;
    :: else ->
10  if
12  :: (turn == 2) ->
        wantp = false;
14  do
        :: turn==1 -> break
        :: else
16  od;
        wantp = true
18  :: else /* leaves if, if turn<>2 */
20  fi
    od;
22 progressP:
    wantp = false;
24  turn = 2

```

```

26 }
28 active proctype Q() {
29     do
30         :: wantq = true;
31         do
32             :: !wantp -> break;
33             :: else ->
34                 if
35                     :: (turn == 1) ->
36                         wantq = false;
37                         do
38                             :: turn==2 -> break
39                             :: else
40                                 od;
41                             wantq = true
42                             :: else /* leaves if, if turn<>2 */
43                                 fi
44                         od;
45 progressQ:
46         wantq = false;
47         turn = 1
48     od
49 }

```

“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 [?]:

```

1 byte x=0;
3 active proctype P() {
4     do
5         :: true -> x = 1 - x;
6     od
7 }
9 active proctype Q() {
10    do
11        :: true -> progress1: x = 1 - x;
12    od
13 }

```

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:

```

1 0 P:1 1) 1
  <<<<START OF CYCLE>>>>
3 0 P:1 1) x = (1-x)
  Process Statement      x
5 0 P:1 1) 1              1
  0 P:1 1) x = (1-x)      1
7 0 P:1 1) 1              0

```

spin

Consider the code for Attempt IV

```

1  bool wantP = false, wantQ = false;

3  active proctype P() {
    do
5     :: wantP = true;
        do
7         :: wantQ -> wantP = false; wantP = true
           :: else -> break
        od;
9     wantP = false
    od
11 }

13 active proctype Q() {
    do
15     :: wantQ = true;
        do
17         :: wantP -> wantQ = false; wantQ = true
           :: else -> break
        od;
21     wantQ = false
    od
23 }

```

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.

```

1  bool wantP = false, wantQ = false;

3  active proctype P() {
    do
5     :: wantP = true;
        do
7         :: wantQ -> wantP = false; wantP = true
           :: else -> break
        od;
9     progress1:
        wantP = false
    od
13 }

15 active proctype Q() {
    do
17     :: wantQ = true;
        do
19         :: wantP -> wantQ = false; wantQ = true
           :: else -> break
        od;
21     progress2:
        wantQ = false
23 }

```



```

25 }
    od

```

Here is the output from SPIN

```

1  1 Q:1  1)  wantQ = 1
    Process Statement          wantQ
3  1 Q:1  1)  else           1
    1 Q:1  1)  wantQ = 0      1
5  0 P:1  1)  wantP = 1      0
    Process Statement          wantP      wantQ
7  1 Q:1  1)  wantQ = 1      1          0
    1 Q:1  1)  wantP          1          1
9  0 P:1  1)  wantQ          1          1
    <<<<START OF CYCLE>>>>
11 1 Q:1  1)  wantQ = 0      1          1
    1 Q:1  1)  wantQ = 1      1          0
13 1 Q:1  1)  wantP          1          1
    0 P:1  1)  wantP = 0      1          1
15 1 Q:1  1)  wantQ = 0      0          1
    1 Q:1  1)  wantQ = 1      0          0
17 1 Q:1  1)  else           0          1
    0 P:1  1)  wantP = 1      0          1
19 0 P:1  1)  wantQ          1          1
    1 Q:1  1)  wantQ = 0      1          1
21 0 P:1  1)  wantP = 0      1          0
    1 Q:1  1)  wantQ = 1      0          0
23 1 Q:1  1)  else           0          1
    1 Q:1  1)  wantQ = 0      0          1
25 0 P:1  1)  wantP = 1      0          0
    1 Q:1  1)  wantQ = 1      1          0
27 1 Q:1  1)  wantP          1          1
    Process Statement          wantP      wantQ
29 0 P:1  1)  wantQ          1          1
    spin: trail ends after 50 steps

```


Chapter 6

Solution to Selected Exercises

Section ??

Answer 6.0.1 (Exercise ??). *jj*

Section ??

Bibliography

- [Car96] Tom Cargill. Specific notification for java thread synchronization. www.dre.vanderbilt.edu/%7Eeschmidt/PDF/specific-notification.pdf, 1996.
- [Har98] Stephen Hartley. Concurrent Programming: The Java Programming Language. Oxford University Press, 1998.