Concurrent Programming Notes

v0.01 – September 12, 2022

Eduardo Bonelli

Preface

These course notes provide supporting material for CS511.

Contents

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.

```groovy
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].

Semi-colons are optional in Groovy

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 it is actually a local variable. In Groovy, global variables are declared by omitting the type annotation.

```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.

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?

```groovy
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, 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 some rigorous device to be able to model all such possible interleavings and check whether they satisfy our intended properties. A device that describes the run-time execution of a concurrent program. There is a further, equally important reason, why we need this device. Consider the following program:

```
int x=0 // shared variable

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 produces 1 as output!

```bash
$ groovy ex1
1
$
```

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 $s \to s'$ for $(s, s') \in \to$.

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

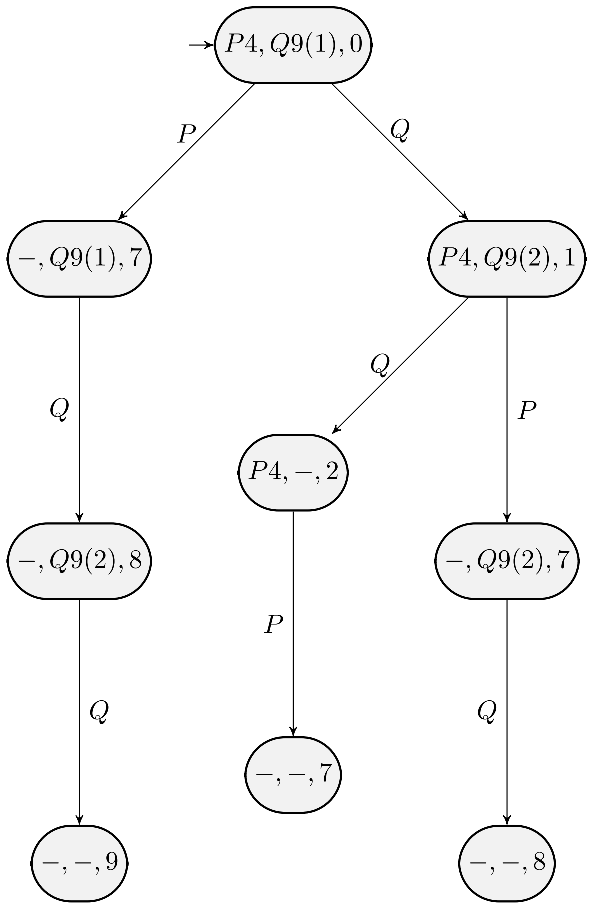
$$\begin{tikzcd}
{} \arrow[r] & P4, Q8, 0 \arrow[dl, "P"'] \arrow[dr, "Q"] & \\
-, Q8, 1 \arrow[d, "Q"'] & & P4, -, 2 \arrow[d, "P"] \\
-, -, 2 & & -, -, 1
\end{tikzcd}$$

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

```
int x = 0

Thread.start { //P
   x = 7
}

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



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
}
```

How we deal with "while"-loops.

```
int x=0 // shared variable

Thread.start { //P
    while (x<1) {
      print x
    }
}

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

How we decorate transitions with the output string of a print

```
int x = 0  // shared variable

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

Thread.start { //Q
  while (x >= 2)
    print x
}
```

```
int counter=0 // shared variable

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

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

println counter // print value of counter
```

1.3 Atomicity

Consider the following program:

```
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:

```
$ groovy ex3
1
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 {

    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:

```
$ javac A.java
$ javap -c A
Compiled from "A.java"
class A implements java.lang.Runnable {
  static int x;

  A();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void run();
    Code:
       0: getstatic     #7                  // Field x:I
       3: iconst_1
       4: iadd
       5: putstatic     #7                  // Field x:I
       8: return

  public static void main(java.lang.String[]);
    Code:
       0: new           #13                 // class java/lang/Thread
       3: dup
       4: new             #8                    // class A
       7: dup
       8: invokespecial   #15                   // Method "<init>":()V
      11: invokespecial   #16                   // Method java/lang/Thread."<init>":(Ljava
      14: invokevirtual   #19                   // Method java/lang/Thread.start:()V
      17: new             #13                   // class java/lang/Thread
      20: dup
      21: new             #8                    // class A
      24: dup
      25: invokespecial   #15                   // Method "<init>":()V
      28: invokespecial   #16                   // Method java/lang/Thread."<init>":(Ljava
      31: invokevirtual   #19                   // Method java/lang/Thread.start:()V
      34: return

  static {};
    Code:
       0: iconst_0
       1: putstatic       #7                    // Field x:I
       4: return
}
```
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`.

1.4 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 2.1[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 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

[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.

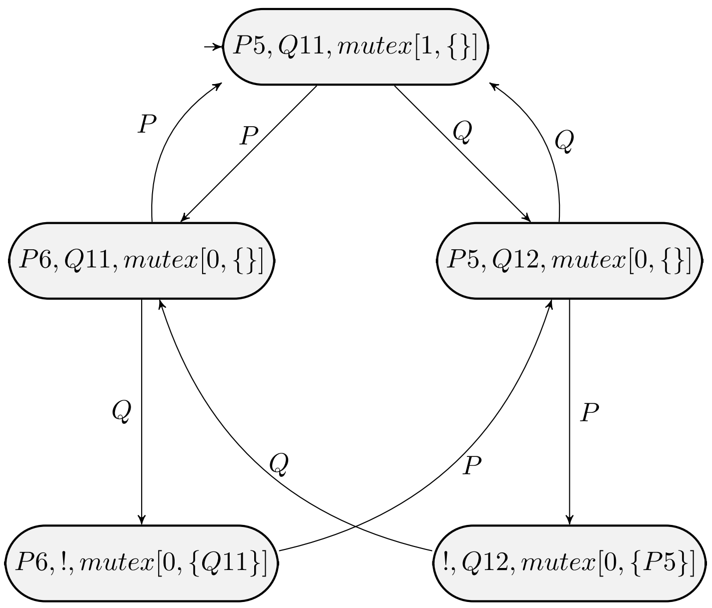

$P5, Q11, mutex[1, \{\}]$
$P6, Q11, mutex[0, \{\}]$
$P5, Q12, mutex[0, \{\}]$
$P6, !, mutex[0, \{Q11\}]$
$!, Q12, mutex[0, \{P5\}]$
P
Q

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.

```
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 MEP using a binary semaphore for N=3

2.3 More Examples

```
Thread.start {
    println "A"
    println "B"
}
Thread.start {
    println "C"
    println "D"
}
```

```
Semaphore cAfterA = new Semaphore(0)

Thread.start {
    println "A"
    mutex.release()
    println "B"
}
Thread.start {
    mutex.acquire()
    println "C"
    println "D"
}
```

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

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

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()
      }
   }
}
```

2.4.2 Readers/Writers

Chapter 3

Monitors

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

    public synchronized void inc() {
       c++
    }

    public synchronized void dec() {
       c--
    }
}

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

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

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

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

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

3.1 A monitor implementing a semaphore

```
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
```

3.2 Producers/Consumers

```
class PC {
    private Object buffer;

    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

Naive solution. Correct but unfair on writers.

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

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

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

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

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

class RW {
    private int readers;
    private int 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=0;
    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();
    }
    }

    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();
    }
    }
}
```

A fair solution.
Two situations that lead to deadlock. Below: R1, W1, R2.

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

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

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

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();
    }
    }
```

Chapter 4

Promela

4.1 Syntax

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

```
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)
}
```

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

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

bash

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.

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

Semi-colon is a separator, not a terminator.

4.1.1 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)
}
```

4.1.2 Expressions as blocking commands

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

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

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

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() {
  printf("A");
}

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

init {
  printf("Pr %d",_nr_pr);
   _nr_pr==1;
   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.

4.1.3 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;

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

inline release(s) {
 s++
}

/* 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":

```
byte s=1;
byte c=0;

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

inline release(s) {
 s++
}

proctype P() {
 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)?

4.2 Assertion-Based Model Checking

4.2.1 The Bar Problem Revisited

Listing 4.2 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.

```
> 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)
```

```
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 4.1: Attempt III in Promela

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

inline acquire(sem) {
 atomic {
  sem>0 -> sem--
 }
}

inline release(sem) {
 sem++
}

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

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

Figure 4.2: Solution to Bar Problem in Promela

```
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)`.

Let us get back to the task of verifying that the solution is correct. In order to do so we add two counters. Listing 4.2.1 exhibits the updated code.

End labels have to have `end` as a prefix.

Using assertion based Mc to verify the Bar exercise.

```

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

inline acquire(permits) {
  skip;
end1:
```

```
10    atomic {
        permits>0;
12      permits--
      }
14  }

16  inline release(permits) {
      permits++
18  }

20  active [5] proctype Jets() {

22    acquire(mutex);
      acquire(ticket);
24    acquire(ticket);
      release(mutex)
26    j++;
      assert (j*2<=p)
28  }

30
    active [5] proctype Patriots() {
32    release(ticket)
      p++;
34    assert (j*2<=p)
    }
```

Exercise: comment out line 18 and then verify your code again.

4.2.2 The MEP Problem

Consider the code for Attempt III from Fig. 4.1. Line 8 and 19 may block. Let us replace these lines with a busy waiting loop, as in the original presentation of Attempt III. The `else` case is important since otherwise the inner do loop would be blocked; we want the inner do loop to cycle while it waits for the condition to hold.

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

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

```
        wantQ=false
  od
}

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

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

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

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

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

```
$ spin -a attemptiii.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 36 byte, depth reached 14, errors: 0
       21 states, stored
       21 states, matched
       42 transitions (= stored+matched)
        1 atomic steps
hash conflicts:          0 (resolved)

Stats on memory usage (in Megabytes):
    0.001   equivalent memory usage for states (stored*(State-vector + overhead))
    0.290   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
    ver.pml:17, state 15, "-end-"
    (1 of 15 states)
unreached in proctype Q
    ver.pml:31, state 15, "-end-"
    (1 of 15 states)
unreached in init
    (0 of 4 states)

pan: elapsed time 0 seconds
```
bash

4.3 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 Attempt III does not enjoy 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 MEP 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

"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
```

4.3.1 The Zoo Problem Revisited

Consider the Zoo Problem discussed in Exercise **??**:

> In a zoo there is a common feeding area for exotic mice and exotic felines. The feeding area has a common feeding lot that holds up to 2 animals. The feeding area can be used by both mice and felines, but cannot be used by mice and felines at the same time for obvious reasons.

A solution in Promela is given in Listing 4.3.

Exercise 4.3.1. *Show that if lines are removed, then deadlock is possible. Explain the deadlock situation that can arise.*

Exercise 4.3.2. *Show, using assertions, that there cannot be felines feeding, if there are mice feeding and, likewise, there cannot be mice feeding, if there are felines feeding.*

Exercise 4.3.3. *You are asked to show that there can be at most two mice and at most two felines in the feeding lot. Since you already know that both cannot be feeding at the same time, you propose the following assertions (only the portion of code that is modified is shown) in the `Mouse` process:*

```
byte mice = 0;
byte felines = 0;
byte mutexMice = 1;
byte mutexFelines =1;
byte feedinglot = 2;
byte mutex = 1;

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

inline release(sem) {
 sem++
}

active [3] proctype Mouse() {
  acquire(mutex);
  acquire(mutexMice);
  mice++;
  if
  :: mice==1 -> acquire(mutexFelines);
    :: else -> skip;
  fi
  release(mutexMice);
  release(mutex);

  acquire(feedinglot);
  // access feeding lot
  release(feedinglot);

  acquire(mutexMice);
  mice--;
  if
    :: mice==0 -> release(mutexFelines);
    :: else -> skip;
  fi
  release(mutexMice);
}

active [3] proctype Feline() {
  acquire(mutex);
  acquire(mutexFelines);
  felines++;
  if
    :: felines==1 -> acquire(mutexMice);
    :: else -> skip;
  fi
  release(mutexFelines);
  release(mutex);

  acquire(feedinglot);
  // use feeding lot
  release(feedinglot);

  acquire(mutexFelines);
  felines--;
  if
    :: felines==0 -> release(mutexMice);
    :: else -> skip;
  fi
  release(mutexFelines);

}
```

Figure 4.3: Zoo Problem in Promela

```
acquire(feedinglot);
// access feeding lot
assert (mice<3);
release(feedinglot);
```

Similarly for the `Feline` process:

```
acquire(feedinglot);
// use feeding lot
assert (felines<3);
release(feedinglot);
```

Unfortunately, when you verify this with Spin, it reports an assertion violation.

1. *Explain why.*
2. *Replace the assertions with new ones that address the problem correctly.*

Chapter 5

Solution to Selected Exercises

Section ??

Answer 5.0.1 (Exercise **??**). *jj*

Section ??