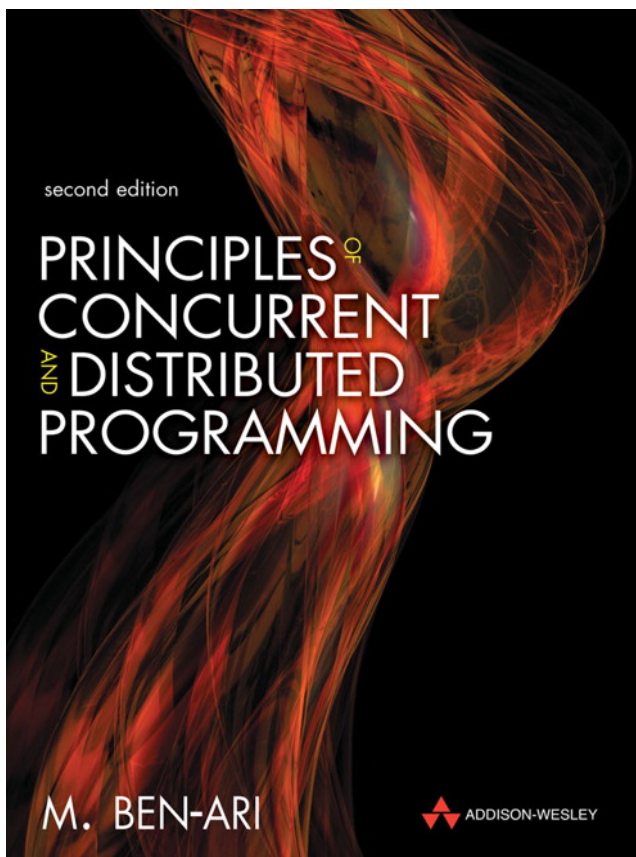


Mordechai Ben-Ari

Principles of the Spin Model Checker

Springer, 2008

ISBN: 978-1-84628-769-5



Mordechai Ben-Ari

Principles of Concurrent and Distributed Programming

Second Edition
Addison-Wesley, 2008

ISBN: 978-0-32131-283-9

<http://www.springer.com/computer/swe/book/978-1-84628-769-5>

Supplementary material (zip, 38 kB)

978-1-84628-769-5-additional material.zip

Chapter 3

Concurrency

A **concurrent program** is a set of sequential programs that can be executed in parallel.

We use the word **process** for the sequential programs that comprise a concurrent program and save the term **program** for this set of processes.

Traditionally, the word **parallel** is used for systems in which the executions of several programs overlap in time by running them on separate processors.

The word **concurrent** is reserved for potential parallelism, in which the execution may, but need not, overlap; instead, the parallelism may only be apparent since it may be implemented by sharing the resources of a small number of processors, often only one.

The challenge in concurrent programming comes from the need to **synchronize** the execution of different processes and to enable them to **communicate**.

It turns out to be extremely difficult to implement safe and efficient synchronization and communication.

Multitasking is a simple generalization from the concept of overlapping I/O with a computation to overlapping the computation of one program with that of another.

Multitasking is the central function of the *kernel* of all modern operating systems.

A **scheduler** program is run by the operating system to determine which process should be allowed to run for the next interval of time. The scheduler can take into account priority considerations, and usually implements *time-slicing*, where computations are periodically interrupted to allow a fair sharing of the computational resources, in particular, of the CPU.

A **concurrent program** consists of a finite set of (sequential) processes. The processes are written using a finite set of **atomic statements**. The execution of a concurrent program proceeds by executing a sequence of the atomic statements obtained by **arbitrarily interleaving** the atomic statements from the processes. A **computation** is an execution sequence that can occur as a result of the interleaving. Computations are also called **scenarios**.

During a computation the **control pointer** (*instruction pointer, location counter*) of a process indicates the next statement that can be executed by that process. Each process has its own control pointer.

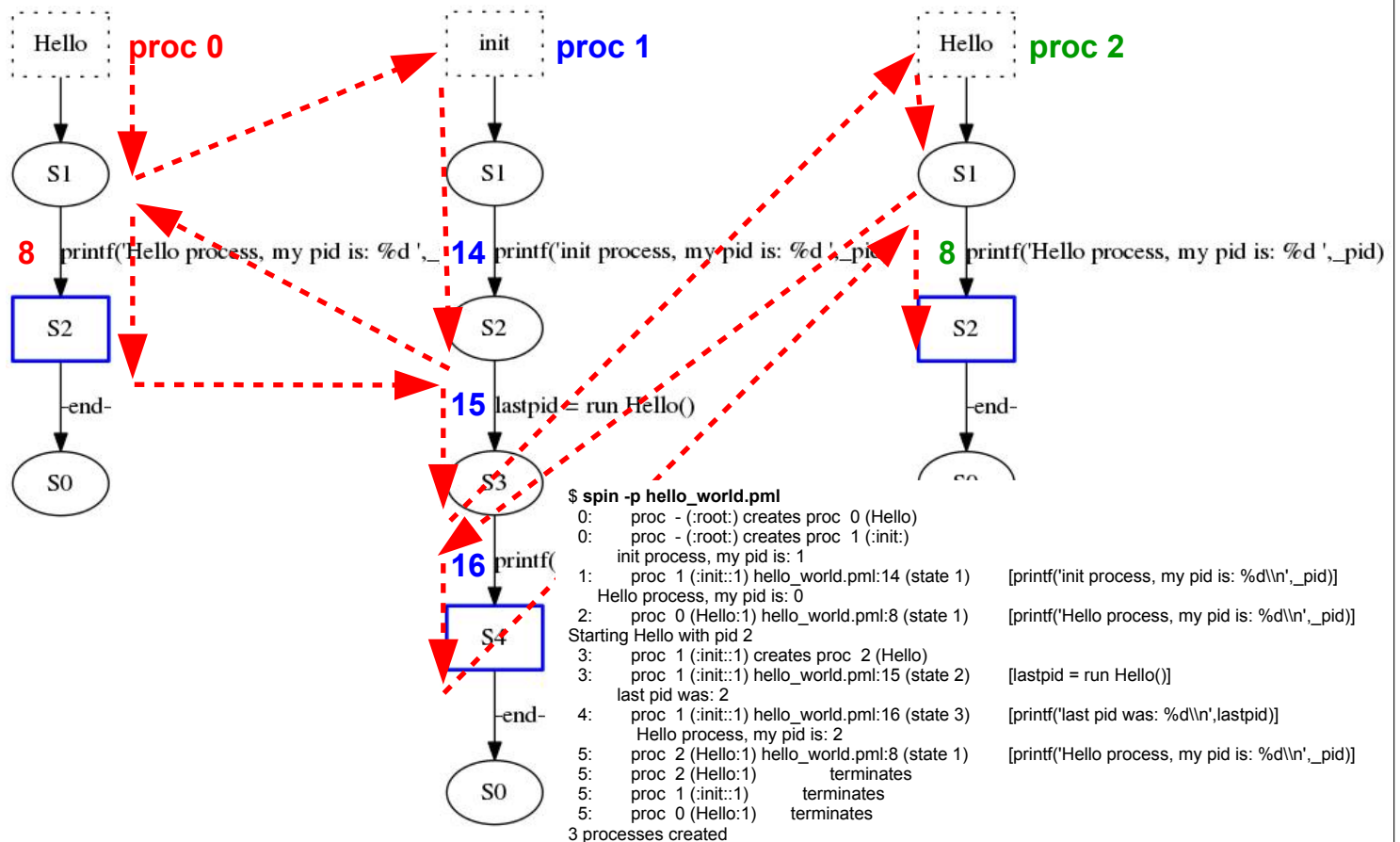
Hello World! (hello_world.pml, simulation 1)

```
1  /*
2      A "Hello World" Promela model for SPIN
3      Theo C. Ruys - SPIN Beginners' Tutorial
4      spinroot.com/spin/Doc/SpinTutorial.pdf
5  */
6
7  active proctype Hello() {
8      printf("Hello process, my pid is: %d\n", _pid)
9  }
10
11  init {
12      int lastpid;
13
14      printf("init process, my pid is: %d\n", _pid);
15      lastpid = run Hello();
16      printf("last pid was: %d\n", lastpid);
17  }
```

\$ spin hello_world.pml

```
init process, my pid is: 1
Hello process, my pid is: 0
last pid was: 2
Hello process, my pid is: 2
3 processes created
```

Hello World! (hello_world.pml, simulation 1)



Hello World! (hello_world.pml, simulation 2)

```
1  /*
2   A "Hello World" Promela model for SPIN
3   Theo C. Ruys - SPIN Beginners' Tutorial
4   spinroot.com/spin/Doc/SpinTutorial.pdf
5  */
6
7  active proctype Hello() {
8      printf("Hello process, my pid is: %d\n", _pid)
9  }
10
11  init {
12      int lastpid;
13
14      printf("init process, my pid is: %d\n", _pid);
15      lastpid = run Hello();
16      printf("last pid was: %d\n", lastpid);
17  }
```

\$ spin hello_world.pml

init process, my pid is: 1

Hello process, my pid is: 2

Hello process, my pid is: 0

last pid was: 2

3 processes created

14, 15, 8, 8, 16

Hello World! (hello_world.pml, simulation 3)

```
1  /*
2    A "Hello World" Promela model for SPIN
3    Theo C. Ruys - SPIN Beginners' Tutorial
4    spinroot.com/spin/Doc/SpinTutorial.pdf
5  */
6
7  active proctype Hello() {
8      printf("Hello process, my pid is: %d\n", _pid)
9  }
10
11  init {
12      int lastpid;
13
14      printf("init process, my pid is: %d\n", _pid);
15      lastpid = run Hello();
16      printf("last pid was: %d\n", lastpid);
17  }
```

\$ spin hello_world.pml 8, 14, 15, 8, 16

 Hello process, my pid is: 0
 init process, my pid is: 1
 Hello process, my pid is: 2
 last pid was: 2

3 processes created

Hello World! (hello_world.pml, simulation 4)

```
1  /*
2    A "Hello World" Promela model for SPIN
3    Theo C. Ruys - SPIN Beginners' Tutorial
4    spinroot.com/spin/Doc/SpinTutorial.pdf
5  */
6
7  active proctype Hello() {
8      printf("Hello process, my pid is: %d\n", _pid)
9  }
10
11  init {
12      int lastpid;
13
14      printf("init process, my pid is: %d\n", _pid);
15      lastpid = run Hello();
16      printf("last pid was: %d\n", lastpid);
17  }
```

\$ spin hello_world.pml 14, 8, 15, 8, 16

 init process, my pid is: 1 14, 15, 8, 8, 16
 Hello process, my pid is: 0
 Hello process, my pid is: 2
 last pid was: 2

3 processes created

Hello World! (hello_world.pml.pml)

```
1  /*
2    A "Hello World" Promela model for SPIN
3    Theo C. Ruys - SPIN Beginners' Tutorial
4    spinroot.com/spin/Doc/SpinTutorial.pdf
5  */
6
7  active proctype Hello() {
8      printf("Hello process, my pid is: %d\n", _pid)
9  }
10
11  init {
12      int lastpid;
13
14      printf("init process, my pid is: %d\n", _pid);
15      lastpid = run Hello();
16      printf("last pid was: %d\n", lastpid);
17  }
```

8, 14, 15, 8, 16
8, 14, 15, 16, 8
14, 8, 15, 8, 16
14, 8, 15, 16, 8

14, 15, 8, 8, 16
14, 15, 8, 16, 8
14, 15, 8, 8, 16
14, 15, 16, 8, 8

14, 15, 8, 16, 8
14, 15, 16, 8, 8

Section 3.1

Interleaving

Listing 3.1. Interleaving statements (interleave1.pml)

```
3  byte    n = 0;
4
5  active proctype P() {
6      n = 1;
7      printf("Process P, n = %d\n", n);
8  }
9
10 active proctype Q() {
11     n = 2;
12     printf("Process Q, n = %d\n", n);
13 }
```

```
$ spin interleave1.pml
    Process P, n = 2
    Process Q, n = 2
2 processes created

$ spin interleave1.pml
    Process P, n = 1
    Process Q, n = 1
2 processes created

$ spin interleave1.pml
    Process P, n = 1
    Process Q, n = 2
2 processes created

$ spin interleave1.pml
    Process Q, n = 2
    Process P, n = 1
2 processes created

$ spin interleave1.pml
    Process P, n = 1
    Process Q, n = 2
2 processes created
```

Process	Statement	n	Output

P	n = 1	0	
P	printf(P)	1	
Q	n = 2	1	P, n = 1
Q	printf(Q)	2	
			Q, n = 2

Six possible computations of the program:

1	2	3	4	5	6
=====					
n = 1	n = 1	n = 1	n = 2	n = 2	n = 2
printf(P)	n = 2	n = 2	printf(Q)	n = 1	n = 1
n = 2	printf(P)	printf(Q)	n = 1	printf(Q)	printf(P)
printf(Q)	printf(Q)	printf(P)	printf(P)	printf(P)	printf(Q)

We say that the computations of a program are obtained by ***arbitrarily interleaving*** of the statements of the processes.

If each process p_i were run by itself, a computation of the process would be a sequence of states $(s_i^0, s_i^1, s_i^2, \dots)$, where state s_i^{j+1} follows state s_i^j if and only if it is obtained by executing the statement at the location counter of p_i in s_i^j .

The word "interleaving" is intended to represent the image of "selecting" a statement from the possible computations of the individual processes and "merging" then into a computation of all the processes of the system.

For the program in Listing 3.1, a state is a triple consisting of the value of **n** and the location counters of processes **P** and **Q**.

The computation obtained by executing the processes by themselves can be represented as

$(0, 6, -) \rightarrow (1, 7, -) \rightarrow (1, 8, -)$

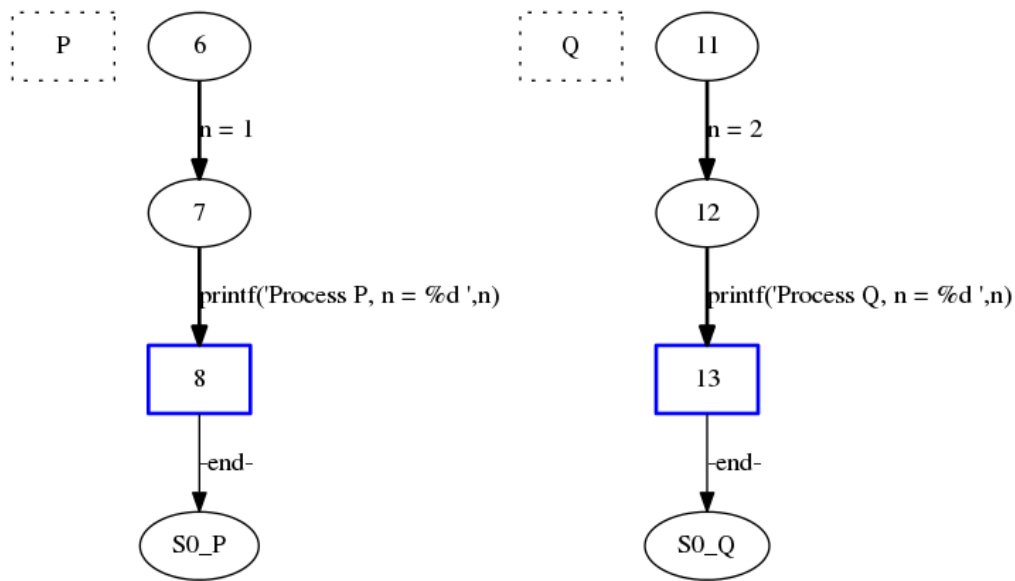
for process **P**, and

$(0, -, 11) \rightarrow (2, -, 12) \rightarrow (2, -, 13)$

for process **Q**.

The third computation above is obtained by interleaving the two separate computations:

$(0, 6, -) \rightarrow \text{"select" from P}$
 $(1, 7, 11) \rightarrow \text{"select" from Q}$
 $(2, 7, 12) \rightarrow \text{"select" from Q}$
 $(2, 7, 13) \rightarrow \text{"select" from P}$
 $(2, 8, 13)$



```

$ spin -p -g interleave1.pml # print all statements and global vars
0:  proc - (:root:) creates proc 0 (P)
0:  proc - (:root:) creates proc 1 (Q)
1:  proc 1 (Q:1) interleave1.pml:11 (state 1)      [n = 2]
      n = 2
      Process Q, n = 2
2:  proc 1 (Q:1) interleave1.pml:12 (state 2)      [printf('Process Q, n =
%d\\n',n)]
3:  proc 0 (P:1) interleave1.pml:6 (state 1)      [n = 1]
      n = 1
      Process P, n = 1
4:  proc 0 (P:1) interleave1.pml:7 (state 2)      [printf('Process P, n =
%d\\n',n)]
4:  proc 1 (Q:1)      terminates
4:  proc 0 (P:1)      terminates
2 processes created

```

Section 3.2

Atomicity

Statements in PROMELA are *atomic*. At each step, the statement pointed to by the location counter of some (arbitrary) process is executed in its entirety.

So, for example, in Listing 3.1 it is not possible for the assignment statements to overlap in a way that causes n to receive some value other than 1 or 2.

Warning

Expressions in PROMELA are *statements*.

In an **if**- or **do**-statement it is possible for interleaving to occur between **the evaluation of the expression** (statement) that is the guard and **the execution of the statement** after the guard.

In the following example, assume that **a** is a global variable; you **cannot** infer that division by zero is impossible:

```
if  
  :: a != 0 ->  
    c = b / a  
  :: else ->  
    c = b  
fi
```

Collatz conjecture (collatz.py) (1/3)

```
$ cat -n collatz.py | expand
 1 import sys,threading
 2
 3 def odd():
 4     global n
 5     while True:
 6         if n==1: break
 7         if n& 1:
 8             n= 3*n+1
 9             print(n,end=' ')
10
11 def even():
12     global n
13     while True:
14         if n==1: break
15         if not n& 1:
16             n>>= 1
17             print(n,end=' ')
18
...
```

Collatz conjecture (collatz.py) (2/3)

```
...
19 try:
20     n= int(sys.argv[1])
21 except IndexError:
22     print("Missed o wrong input")
23     sys.exit(1)
24
25 assert n>0, "Must be positive integer"
26 print(n,end=' ')
27 t1= threading.Thread(target=odd)
28 t2= threading.Thread(target=even)
29 t1.start(), t2.start()
30 t1.join(), t2.join()
31 print()

$ python3 collatz.py 12
12 6 3 10 5 16 8 4 2 1

$ python3 collatz.py 19
19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```


Collatz conjecture (collatz.py) (3/3)

```
$ python3 collatz.py 19
^C19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 4 Traceback
(most recent call last):
  File "collatz.py", line 30, in <module>
    t1.join(), t2.join()
  File "/usr/lib/python3.6/threading.py", line 1056, in join
    self._wait_for_tstate_lock()
  File "/usr/lib/python3.6/threading.py", line 1072, in
_wait_for_tstate_lock
    elif lock.acquire(block, timeout):
KeyboardInterrupt
^CException ignored in: <module 'threading' from
'/usr/lib/python3.6/threading.py'>
Traceback (most recent call last):
  File "/usr/lib/python3.6/threading.py", line 1294, in _shutdown
    t.join()
  File "/usr/lib/python3.6/threading.py", line 1056, in join
    self._wait_for_tstate_lock()
  File "/usr/lib/python3.6/threading.py", line 1072, in
_wait_for_tstate_lock
    elif lock.acquire(block, timeout):
KeyboardInterrupt
```

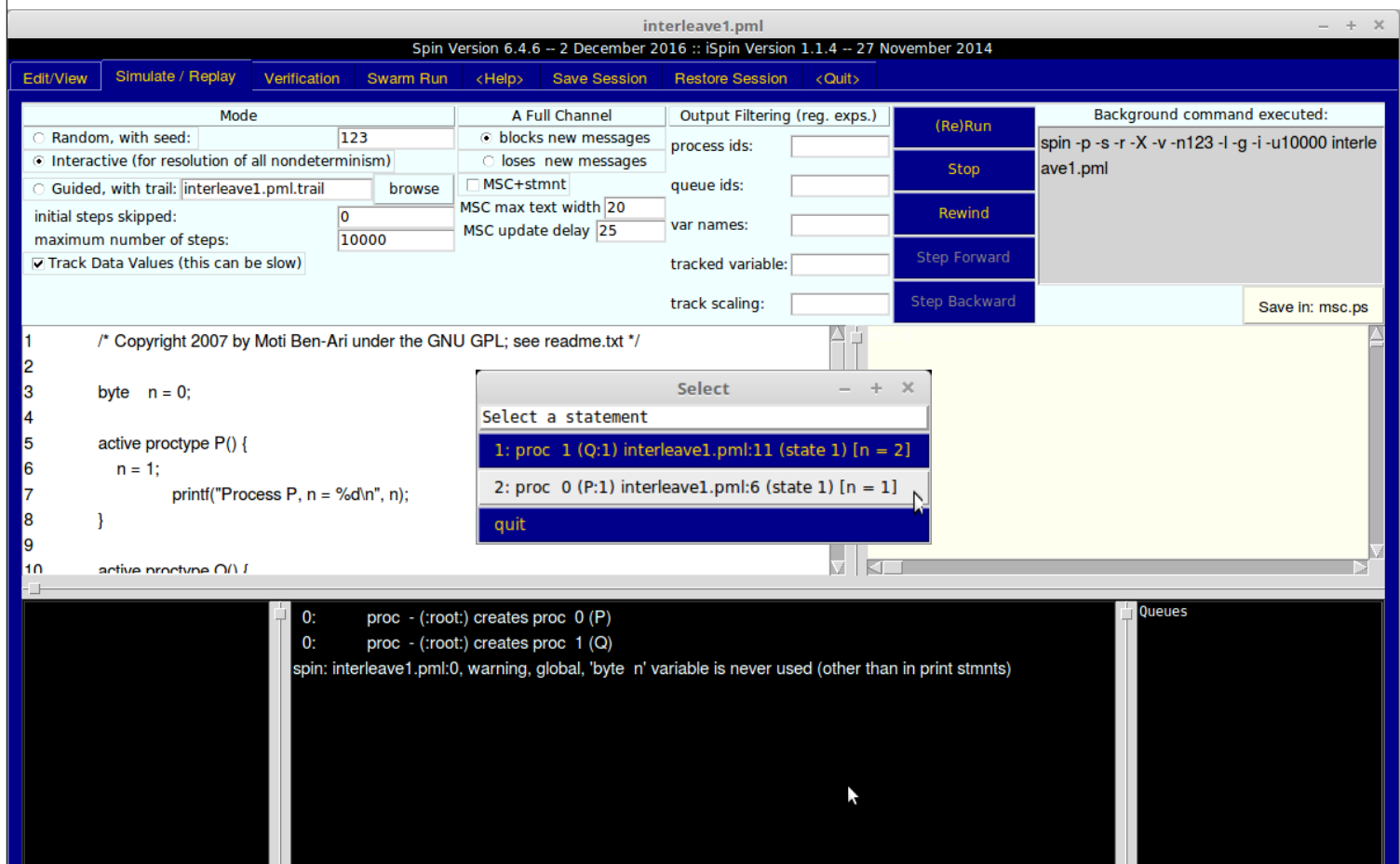
Section 3.3

Interactive simulation

```

$ spin -i interleave1.pml # interactive (random simulation)
Select a statement
    choice 1: proc 1 (Q:1) interleave1.pml:11 (state 1) [n = 2]
    choice 2: proc 0 (P:1) interleave1.pml:6 (state 1) [n = 1]
Select [1-2]: 1
Select a statement
    choice 1: proc 1 (Q:1) interleave1.pml:12 (state 2) [printf('Process Q, n
= %d\\n',n)]
    choice 2: proc 0 (P:1) interleave1.pml:6 (state 1) [n = 1]
Select [1-2]: 2
Select a statement
    choice 1: proc 1 (Q:1) interleave1.pml:12 (state 2) [printf('Process Q, n
= %d\\n',n)]
    choice 2: proc 0 (P:1) interleave1.pml:7 (state 2) [printf('Process P, n =
%d\\n',n)]
Select [1-2]: q
$

```



Section 3.4

Interference between processes

Listing 3.2. Interference between two processes (interleave2.pml)

```
3  byte    n = 0;
4
5  active proctype P() {
6    byte temp;
7    temp = n + 1;
8    n = temp;
9    printf("Process P, n = %d\n", n)
10 }
11
12 active proctype Q() {
13   byte temp;
14   temp = n + 1;
15   n = temp;
16   printf("Process Q, n = %d\n", n)
17 }
```

Sentencias atómicas!

```
$ spin interleave2.pml
    Process Q, n = 2
    Process P, n = 2
2 processes created

$ spin interleave2.pml
    Process Q, n = 1
    Process P, n = 2
2 processes created

$ spin interleave2.pml
    Process Q, n = 1
    Process P, n = 1
2 processes created

$ spin interleave2.pml
    Process Q, n = 2
    Process P, n = 2
2 processes created

$ spin interleave2.pml
    Process P, n = 1
    Process Q, n = 2
2 processes created

$ spin interleave2.pml
    Process Q, n = 1
    Process P, n = 1
2 processes created
```

Un pequeño ejercicio:

Usando la simulación interactiva,
obtener que el valor de la variable **n**
se imprime dos veces como

n = 1

Fig. 3.1. Perfect interleaving

Process	Statement	<i>n</i>	P: <i>temp</i>	Q: <i>temp</i>	Output
P	<i>temp</i> = <i>n</i> + 1	0	0	0	
Q	<i>temp</i> = <i>n</i> + 1	0	1	0	
P	<i>n</i> = <i>temp</i>	0	1	1	
Q	<i>n</i> = <i>temp</i>	1	1	1	
P	printf(P)	1	1	1	
Q	printf(Q)	1	1	1	P, <i>n</i> = 1 Q, <i>n</i> = 1

This program is a simple model of a CPU that performs computation in registers:

```

load    R1, n
add     R1, #1
store   R1, n

```

In the Promela program the variable *n* represents a memory cell and the variables *temp* represent the register. In a multiprocess system each process has its own copy of the contents of the registers, which is loaded into the CPU registers and saved in memory during a context switch.

Section 3.5

Sets of processes

Listing 3.3. Instantiating two processes (interleave3.pml)

```
1 byte    n = 0;
2
3 active [2] proctype P() {
4     byte temp;
5     temp = n + 1;
6     n = temp;
7     printf("Process P%d, n = %d\n", _pid, n)
8 }
```

```
$ spin interleave3.pml
    Process P1, n = 1
    Process P0, n = 1
2 processes created

$ spin interleave3.pml
    Process P1, n = 2
    Process P0, n = 2
2 processes created
```

Listing 3.4. The `init` process (`init.pml`)

```
3  byte n;
4
5  proctype P(byte id; byte incr) {
6      byte temp;
7      temp = n + incr;
8      n = temp;
9      printf("Process P%d, n = %d\n", id, n)
10 }
11
12 init {
13     n = 1;
14     atomic {
15         run P(1, 10);
16         run P(2, 15)
17     }
18 }
```

Warning

The formal parameters of a **proctype** are separated with semicolons, not commas.

Advanced: The `run` operator

run is an *operator*, so **run** `P()` is an expression, not a statement, and it returns a value: the process ID of the process that is instantiated, or zero if the maximum number of processes (255) have already been instantiated.

Section 3.6

Interference revisited

Count_s.java (1/4)

```
1  /* Copyright (C) 2006 M. Ben-Ari */
2
3  class Count_s extends Thread {
4      static volatile int n = 0;
5      static int N;
6
7      public void run() {
8          int temp;
9          for (int i = 0; i < N; i++) {
10             /* temp = n; n = temp + 1; */
11             n++;
12         }
13     }
14
15     ...
```


Count_s.java (2/4)

```
15  public static void main(String[] args) {
16      if (args.length > 0) {
17          try { N = Integer.parseInt(args[0]); }
18          catch (NumberFormatException e) {
19              System.err.println("Argument" + " must be an integer");
20              System.exit(1);
21          }
22      }
23
24      Count_s p = new Count_s();
25      Count_s q = new Count_s();
26      p.start();
27      q.start();
28      try { p.join(); q.join(); }
29      catch (InterruptedException e) { }
30      System.out.println(N + " + " + N + " = " + n);
31  }
32 }
```

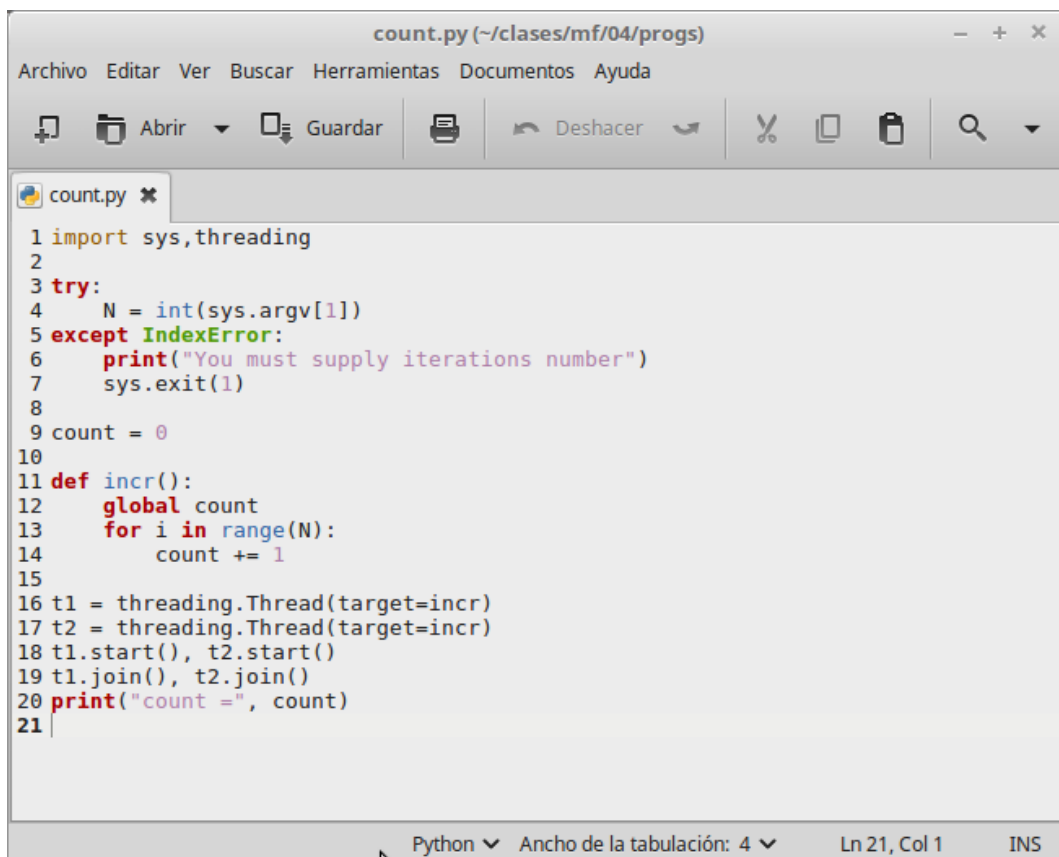
Count_s.java (3/4)

```
$ java Count_s 10
10 + 10 = 20
$ java Count_s 10
10 + 10 = 20
$ java Count_s 10
10 + 10 = 12
$ java Count_s 10
10 + 10 = 20
$ java Count_s 10
10 + 10 = 20
$ java Count_s 100
100 + 100 = 200
$ java Count_s 100
100 + 100 = 200
$ java Count_s 100
100 + 100 = 200
$ java Count_s 100
100 + 100 = 200
```

Count_s.java (4/4)

```
$ java Count_s 1000
1000 + 1000 = 1006
$ java Count_s 1000
1000 + 1000 = 1440
$ java Count_s 1000
1000 + 1000 = 1432
$ java Count_s 1000
1000 + 1000 = 2000
$ java Count_s 1000000
1000000 + 1000000 = 1464643
$ java Count_s 1000000
1000000 + 1000000 = 993030
$ java Count_s 1000000
1000000 + 1000000 = 1290665
$ java Count_s 1000000
1000000 + 1000000 = 1073888
$ java Count_s 1000000
1000000 + 1000000 = 1608788
```

count.py (1/2)



```
count.py (~/clases/mf/04/progs)
Archivo  Editor  Ver  Buscar  Herramientas  Documentos  Ayuda

count.py x
1 import sys, threading
2
3 try:
4     N = int(sys.argv[1])
5 except IndexError:
6     print("You must supply iterations number")
7     sys.exit(1)
8
9 count = 0
10
11 def incr():
12     global count
13     for i in range(N):
14         count += 1
15
16 t1 = threading.Thread(target=incr)
17 t2 = threading.Thread(target=incr)
18 t1.start(), t2.start()
19 t1.join(), t2.join()
20 print("count =", count)
21
```

Python Ancho de la tabulación: 4 Ln 21, Col 1 INS

```
$ python3 count.py 1000
count = 2000

$ python3 count.py 10000
count = 20000

$ python3 count.py 100000
count = 107188

$ python3 count.py 100000
count = 199654

$ python3 count.py 1000000
count = 1219083

$ python3 count.py 1000000
count = 1559081
```

Listing 3.5. Counting with interference

```
$ cat -n count_v6.pml
1 byte    n = 0, i = 0
2
3 proctype P() {
4     byte i, temp
5     for (i : 1 .. 10) {
6         temp = n
7         n = temp + 1
8     }
9 }
10
11 init {
12     atomic {
13         run P()
14         run P()
15     }
16     (_nr_pr == 1)
17     printf("The value is %d\n", n)
18 }
```

Listing 3.5. Counting with interference

```
$ spin count_v6.pml
    The value is 17    /* 16, 18, 15, 19, ... */
```

¿Es posible el valor final 20?
¿Lo puede obtener? ¿Cómo?

¿Cómo se obtiene el valor final 2?

Counting without interference en Promela

```
$ cat -n count_v6_short.pml
1 byte    n = 0
2
3 proctype P() {
4     byte i
5     for (i : 1 .. 10) {
6         n = n + 1
7     }
8 }
9
10
11 init {
12     atomic {
13         run P()
14         run P()
15     }
16     (_nr_pr == 1)
17     printf("The value is %d\n", n)
18     assert (n == 20)
19 }
```

```
$ spin -a count_v6_short.pml  
$ gcc pan.c -o pan  
$ ./pan  
...  
State-vector 36 byte, depth reached 71, errors: 0
```

Section 3.7

Deterministic sequences of statements

Listing 3.6. Deterministic step(dstep.pml)

```
3  byte      n = 0;
4
5  active proctype P() {
6      byte temp;
7      d_step {
8          temp = n + 1;
9          n = temp
10     }
11     printf("Process P, n = %d\n", n)
12 }
13
14 active proctype Q() {
15     byte temp;
16     d_step {
17         temp = n + 1;
18         n = temp
19     }
20     printf("Process Q, n = %d\n", n)
21 }
```

```
$ spin -p -g -l dstep.pml
0:   proc  - (:root:) creates proc  0 (P)
0:   proc  - (:root:) creates proc  1 (Q)
    Q(1):temp = 1
2:   proc  1 (Q:1) dstep.pml:18 (state 2)    [n = temp]
    n = 1
    Process Q, n = 1
3:   proc  1 (Q:1) dstep.pml:20 (state 4)    [printf('Process Q, n = %d\n',n)]
3:   proc  1 (Q:1) terminates
    P(0):temp = 2
5:   proc  0 (P:1) dstep.pml:9 (state 2)     [n = temp]
    n = 2
    Process P, n = 2
6:   proc  0 (P:1) dstep.pml:11 (state 4)    [printf('Process P, n = %d\n',n)]
6:   proc  0 (P:1) terminates
2 processes created
```

¿Puede verificar que el resultado siempre será 2?

Section 3.8

Verification with assertions

Counting with interference (count_v6_for2.pml)

```
$ cat -n count_v6_for2.pml
1 byte    n = 0
2
3 proctype P() {
4     byte i, temp
5     for (i : 1 .. 10) {
6         temp = n
7         n = temp + 1
8     }
9 }
10
11 init {
12     atomic {
13         run P()
14         run P()
15     }
16     (_nr_pr == 1)
17     printf("The value is %d\n", n)
18
19     assert (n > 2)
20 }
```

```
$ spin count_v6_for2.pml
The value is 16

$ spin count_v6_for2.pml
The value is 15

$ spin count_v6_for2.pml
The value is 18

$ spin count_v6_for2.pml
The value is 17

$ spin count_v6_for2.pml
The value is 8

...
```

```
$ spin -a count_v6_for2.pml

$ gcc -o pan pan.c

$ ./pan
pan:1: assertion violated (n>2) (at depth 90)
pan: wrote count_v6_for2.pml.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
+ Partial Order Reduction

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

State-vector 36 byte, depth reached 92, errors: 1

...
```



```

$ spin -t -p -g count_v6_for2.pml
using statement merging
Starting P with pid 1
1:  proc 0 (:init::1) count_v6_for2.pml:13 (state 1)      [(run P())]
Starting P with pid 2
2:  proc 0 (:init::1) count_v6_for2.pml:14 (state 2)      [(run P())]
3:  proc 2 (P:1) count_v6_for2.pml:5 (state 1)           [i = 1]
4:  proc 2 (P:1) count_v6_for2.pml:5 (state 2)           [((i<=10))]
5:  proc 1 (P:1) count_v6_for2.pml:5 (state 1)           [i = 1]
...
41: proc 2 (P:1) count_v6_for2.pml:7 (state 4)           [n = (temp+1)]
      n = 9
42: proc 2 (P:1) count_v6_for2.pml:5 (state 5)           [i = (i+1)]
43: proc 2 (P:1) count_v6_for2.pml:5 (state 2)           [((i<=10))]
44: proc 1 (P:1) count_v6_for2.pml:7 (state 4)           [n = (temp+1)]
      n = 1
...
81: proc 1 (P:1) count_v6_for2.pml:7 (state 4)           [n = (temp+1)]
      n = 10
82: proc 1 (P:1) count_v6_for2.pml:5 (state 5)           [i = (i+1)]
83: proc 1 (P:1) count_v6_for2.pml:8 (state 6)           [else]
84: proc 2 (P:1) count_v6_for2.pml:7 (state 4)           [n = (temp+1)]
      n = 2
85: proc 2 (P:1) count_v6_for2.pml:5 (state 5)           [i = (i+1)]
86: proc 2 (P:1) count_v6_for2.pml:8 (state 6)           [else]
87: proc 2 terminates
88: proc 1 terminates
89: proc 0 (:init::1) count_v6_for2.pml:16 (state 4)      [((_nr_pr==1))]
      The value is 2
90: proc 0 (:init::1) count_v6_for2.pml:17 (state 5)      [printf('The value is %d\\n',n)]
spin: count_v6_for2.pml:19, Error: assertion violated
spin: text of failed assertion: assert((n>2))
...

```

Saved as count_v6_for2in5.pml

count_v6_for2in5.pml

Spin Version 6.4.6 -- 2 December 2016 :: iSpin Version 1.1.4 -- 27 November 2014

Edit/View Simulate / Replay Verification Swarm Run <Help> Save Session Restore Session <Quit>

Safety

- ☒ + invalid endstates (deadlock)
- ☒ + assertion violations
- ☐ + xr/xs assertions

Storage Mode

- ☒ exhaustive
 - ☐ + minimized automata (slow)
 - ☐ + collapse compression
- ☐ hash-compact
- ☐ bitstate/supertrace

Search Mode

- ☒ depth-first search
 - ☒ + partial order reduction
 - ☐ + bounded context switching
- ☐ breadth-first search
- ☐ iterative search for short trail

Never Claims

- ☒ do not use a never claim or ltl property
- ☐ use claim

claim name (opt):

Run Stop Save Result In: pan.out

```

1  byte  n = 0
2
3  proctype P() {
4      byte i, temp
5      for (i : 1 .. 5) {
6          temp = n
7          n = temp + 1
8      }
9  }
10
11 init {
12     atomic {
13         run P()
14         run P()
15     }
16     (_nr_pr == 1)
17     printf("The value is %d\\n", n)
18
19     assert (n > 2)
20 }

```

State-vector 28 byte, depth reached 52, errors: 1

3494 states, stored

903 states, matched

4397 transitions (= stored+matched)

1 atomic steps

hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):

0.187 equivalent memory usage for states (stored*(State-vector + overhead))

0.389 actual memory usage for states

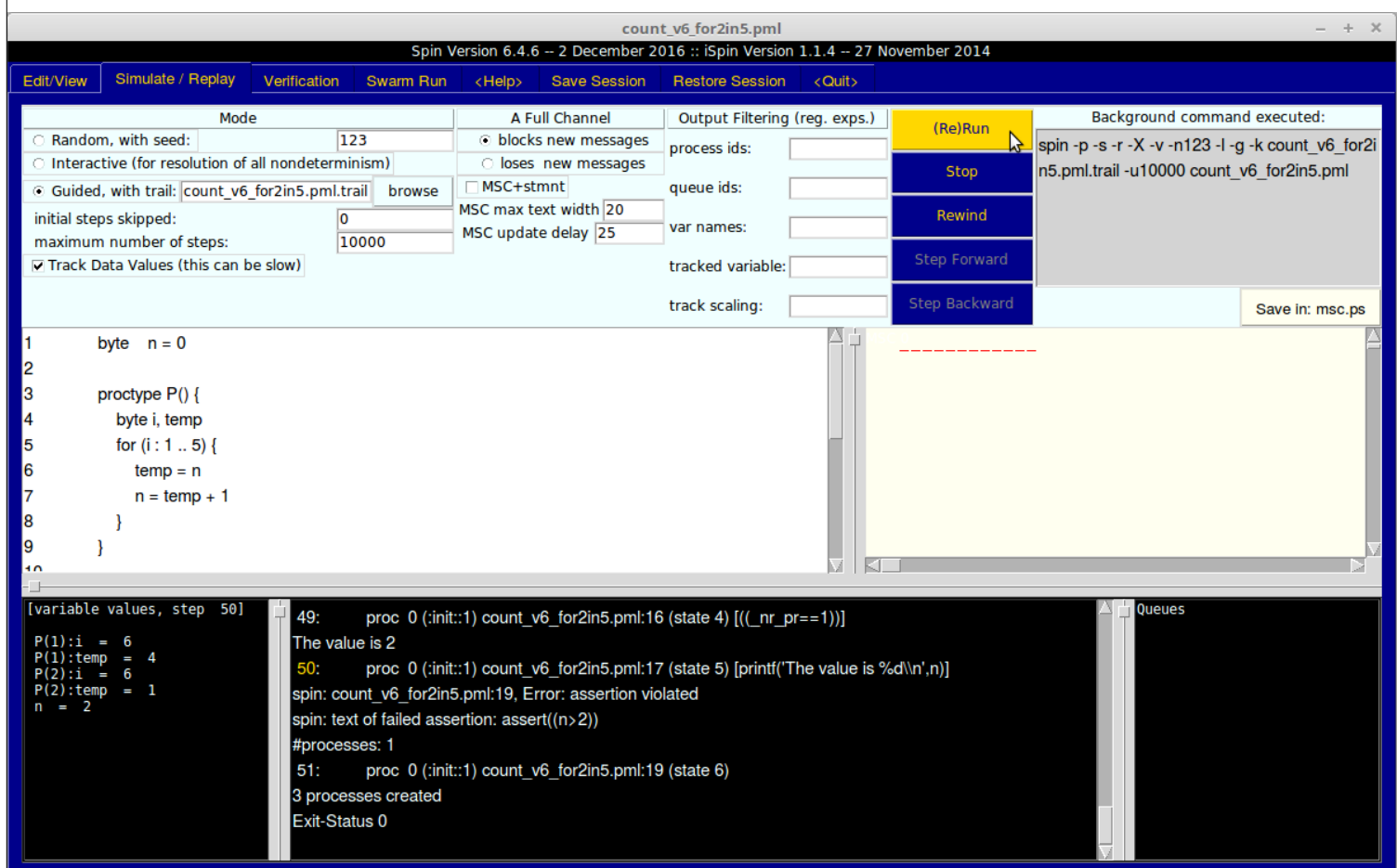
128.000 memory used for hash table (-w24)

0.534 memory used for DFS stack (-m10000)

128.827 total actual memory usage

pan: elapsed time 0 seconds

To replay the error-trail, goto Simulate/Replay and select "Run"



Section 3.9

The critical section problem

The specification of the problem:

A system consists of two or more concurrently executing processes. The statements of each process are divided into *critical* and *noncritical* sections that are repeatedly executed one after the other. A process may halt in its noncritical section, but not in its critical section. Design an algorithm for ensuring that the following specifications hold:

Mutual exclusion

At most one process is executing its critical section at any time.

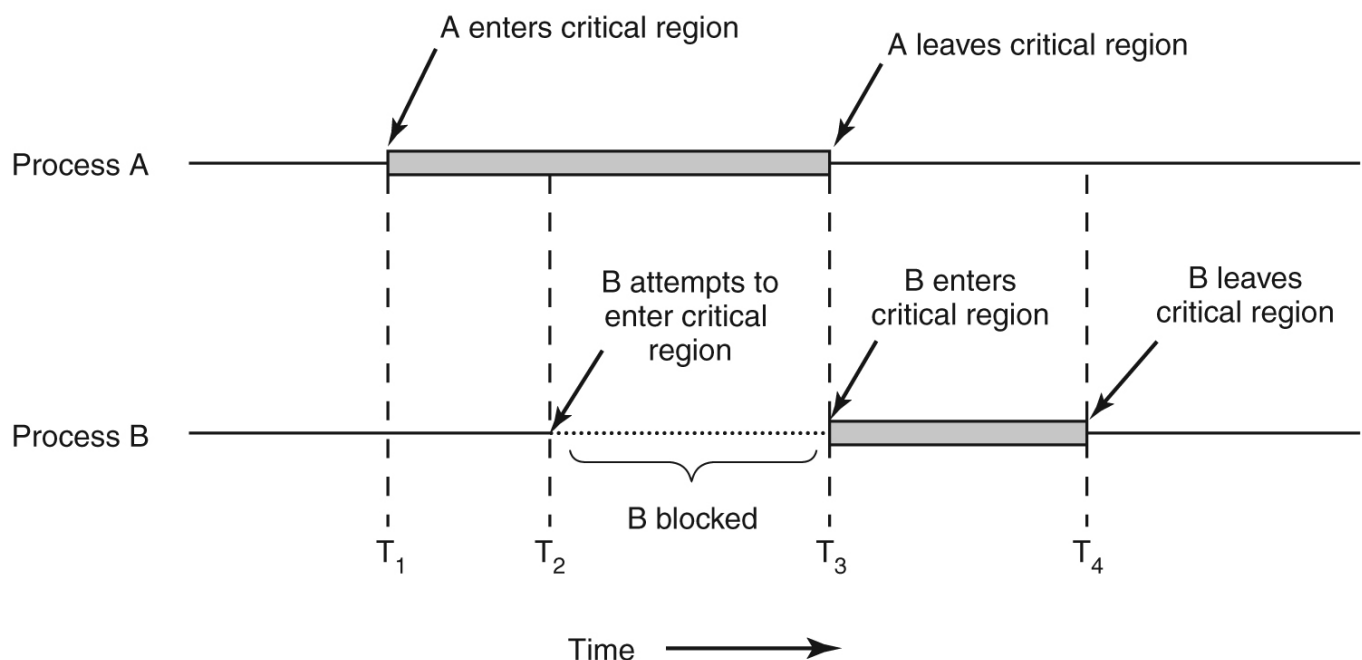


Figure 2-22 (MOS4E). Mutual exclusion using critical regions

Absence of deadlock

It is impossible to reach a state in which *some processes* are trying to enter their critical sections, but *no process* is successful.

Absence of starvation

If *any process* is trying to execute its critical section, then eventually *that process* is successful.

Listing 3.7. Incorrect solution for the critical section problem (cs0.pml)

```
3  bool wantP = false, wantQ = false;
4
5  active proctype P() {
6      do
7          :: printf("Noncritical section P\n");
8             wantP = true;
9             printf("Critical section P\n");
10             wantP = false
11      od
12 }
13
14 active proctype Q() {
15     do
16         :: printf("Noncritical section Q\n");
17            wantQ = true;
18            printf("Critical section Q\n");
19            wantQ = false
20     od
21 }
```

Listing 3.8. Verifying mutual exclusion (cs.pml)

```
3  bool wantP = false, wantQ = false;
4  byte critical = 0;  /* a ghost variable */
5
6  active proctype P() {
7      do
8          :: printf("Noncritical section P\n");
9             wantP = true;
10             critical++;
11             printf("Critical section P\n");
12             assert (critical <= 1);
13             critical--;
14             wantP = false
15         od
16     }
17
18     active proctype Q() {
19         do
20             :: printf("Noncritical section Q\n");
21                wantQ = true;
22                critical++;
23                printf("Critical section Q\n");
24                assert (critical <= 1);
25                critical--;
26                wantQ = false
27            od
28        }
```

```
$ spin -a cs.pml
```

```
$ gcc -o pan pan.c
```

```
$ ./pan
```

```
pan:1: assertion violated (critical<=1) (at depth 22)
pan: wrote cs.pml.trail
```

```
(Spin Version 6.4.8 -- 2 March 2018)
```

```
Warning: Search not completed
+ Partial Order Reduction
```

```
Full statespace search for:
```

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

```
State-vector 28 byte, depth reached 22, errors: 1
```

```
...
```

```
$ spin -t cs.pml
    Noncritical section Q
Noncritical section P
    Critical section Q
    Noncritical section Q
    Critical section Q
Critical section P
    Noncritical section Q
    Critical section Q
spin: cs.pml:24, Error: assertion violated
spin: text of failed assertion: assert((critical<=1))
spin: trail ends after 23 steps
#processes: 2
    wantP = 1
    wantQ = 1
    critical = 2
23:  proc  1 (Q) cs.pml:25 (state 6)
23:  proc  0 (P) cs.pml:12 (state 5)
2 processes created
```