Universitetet i Oslo
Institutt for Informatikk
PMA
Olaf Owe, Martin Steffen

INF 4140: Models of Concurrency

Høst 2015 — **Series 3** — 14. 9. 2015

Topic: Semaphores

Issued: 14. 9. 2015

Exercise 1 (CS with coordinator) In the critical section protocols in the book, every process executes the same algorithm; these are *symmetric solutions*. It is also possible to solve the problem using a coordinator process. In particular, when a regular process CS[i] wants to enter its critical section, it tells the coordinator, then waits for the coordinator to grant permission.

Assume there are n processes numbered 1 to n. Develop entry and exit protocols for the regular processes and code for the coordinator process. Use flags and `await`-statements for synchronization. The solution must work, if regular processes terminate outside the critical section.

Exercise 2 (Semaphores to pass control) Given the following routine:

```
print() {

  process P1 {
    write(``line 1''); write(``line 2'');
  }

  process P2 {
    write(``line 3''); write(``line 4'');
  }

  process P3 {
    write(``line 5''); write(``line 6'');
  }

}
```

1. How many different outputs could this program produce? Explain your reasoning.
2. Add semaphores to the program so that the six lines of output are printed in the order $1, 2, 3, 4, 5, 6$. Declare and initialize any semaphores you need and add `P` and `V` operations to the above processes.

Exercise 3 (Semaphores for synchronization) Several processes share a resource that has U units. Processes request one unit at a time, but may release several. The routines `request` and `release` are atomic operations as shown below.

```
1  int free := U;
2
3  request() :            # < await (free > 0) free := free - 1; >
4
5  release(int number): # < free := free + number; >
```

Develop implementations of `request` and `release`. Use semaphores for synchronization. Be sure to declare and initialize additional variables you may need.

Exercise 4 (Termination, deadlock, interleaving) Consider the following program:

```
1  int x = 0, y = 0, z = 0;
2  sem lock1 = 1, lock2 = 1;
3
4  process foo {          process bar {
5    z := z + 2;            P(lock2);
6    P(lock1);              y := y + 1;
7    x := x + 2;            P(lock1);
8    P(lock2);              x := x + 1;
9    V(lock1);              V(lock1);
10   y := y + 2;            V(lock2);
11   V(lock2);              z := z + 1;
12 }                      }
```

1. This program might deadlock. How?
2. What are the possible final values of `x,y,` and `z` in the deadlock state?
3. What are the possible final values of `x,y,` and `z` if the program terminates? (Remember that an assignment `z := z + 1` consists of two atomic operations on `z`.)

Exercise 5 (Fetch-and-add ([1, Exercisise 4.3])) Implement P and V with `fetch-and-add` (FA). The behavior of fetch-and-add is given as follows:

```
1
2  FA(var , incr):
3     <int tmp := var;
4       var := var+incr;
5       return(tmp); >
```

Note: the `inc` may be a negative integer, which is being added.

Side remark: fetch-and-add is, in some HW architectures an atomic instruction (for instance, variants in X86-architectures). Atomic instructions such as fetch-and-add, which are more powerful than simple loads and stores (= reading and writing) are offered in the instruction set with the purpose to allow efficient implementation of synchronization primitives in operating systems running on that platform (for instance semaphore operations). Fetch-and-add is only one example of HW-supported atomic synchronization operations.

Exercise 6 (Precedence graph ([1, Exercise 4.4a])) Use semaphores to "implement" the shown precedence/dependence graph.

```
T1 -> T2 -> T4 -> T5

T1 ----> T3 ----> T5
```

Exercise 7 (Implementing await ([1, Exercise 4.13])) Consider the following piece of code, which is intended as implementation of the `await`-statement.

```
1  sem e  := 1, d := 0     # entry and delay sem.
2  int nd := 0             # delay counter
3
4  P(e);
5
6  while (B = false) {
7    nd := nd+1;
8    V(e);
9    P(d);
10   P(e)  };
11
12 S;                      # protected statement
13
14 while (nd > 0)
15   { nd := nd-1; V(d) };
16 V(e);
```

1. Is the code execited atomically?
2. Is it *deadlock free*?
3. Does the code guarantee, that `B` is true before `S` is executed?

Exercise 8 (Exchange function ([1, Exercise 4.29])) Impement exchange function. Exchanging 2 values requires a form of rendez-vouz.

Exercise 9 (Request and release ([1, Exercise 4.34a])) Request and release, sharing *two* printers. The request should return the identity of a free printer, if available (otherwise block). The identity of the free printer is given as argument to the release-procedure.

Exercise 10 (Bear and honeybees 4.36) Program the synchronization problems of one bear + n bees

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.