Universitetet i Oslo
Institutt for Informatikk

PMA

Einar Broch Johnsen, Martin Steffen

INF 4140: Models of Concurrency

Høst 2014 **Series 3** 15. 9. 2014

Topic Semaphores

Issued: 15. 9. 2014

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 terminates 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 printet 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.

```
    int free := U;

    request() :           # < await (free > 0) free := free - 1; >

    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 Consider the following program:

```
int x = 0, y = 0, z = 0;
sem lock1 = 1, lock2 = 1;

process foo {           process bar {
  z := z + 2;             P(lock2);
  P(lock1);               y := y + 1;
  x := x + 2;             P(lock1);
  P(lock2);               x := x + 1;
  V(lock1);               V(lock1);
  y := y + 2;             V(lock2);
  V(lock2);               z := z + 1;
}                       }
```

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 (FA (4.3 [1])) Implement P and V with `FA`. See here

```

FA(var , incr):
   <int tmp := var;
     var := var+incr;
     return(tmp); >
```

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

```
T1 -> T2 -> T4 -> T5
T1 ----- T3 ----> T5
```

Exercise 7 (Implementing await (4.13)) Consider the following:

```
sem e  := 1, d := 0     # entry and delay sem.
int nd := 0             # delay counter

P(e);

while (B = false) {
  nd := nd+1;
  V(e);
  P(d);
  P(e)  };

S;                     # protected statement

while (nd > 0)
  { nd := nd-1; V(d) };
V(e);
```

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

Exercise 9 (4.34a) Request and release, sharing *two* printers.

Exercise 10 (Bears and honeybees 4.36)

References

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