T-106.5600 Concurrent Programming Examination 2.9.2008 /hsa

1. Consider the following program:

```
co   < await  (x mod 3 = 0)   x =  x/3; >
//   < await  (x mod 5 = 0)   x =  x/5; >
//   < await  (x  ==  1)   x =  x + 44; >
oc
```

(a) For what initial values of x does the program terminate assuming that scheduling is weakly fair? What are the corresponding final values? (b) Write a proof outline to demonstrate your claim.

2. Suppose we have to write a busy-wait and weakly fair implementation for semaphores for a machine that has atomic increment and decrement instructions. INC(x) atomically adds 1 to integer variable x, and DEC(x) atomically subtracts 1 from integer variable x. Both INC(x) and DEC(x) return the modified value of x. Answer with an concise argument to the following questions: (a) Is the solution safe ? (b) Is the solution free from deadlock (c) Is the solution free from livelock? (d) What is wrong with it performance wise? Correct the solution in cases needed. (e) Is your solution fair after this ?

```
/* P(s): */
while (DEC(s) < 0) {
       INC(s);  #undo decrement
       }

/* V(s): */
INC (s)
```

3. Dining philosopher on a row: 100 philosophers sit along a long table spending their time thinking and eating. There are 101 forks on the table, one on each side of every philosopher. For eating a philosopher needs exclusive access to both of the forks.

```
sem fork[101]= {1, 1, 1 , 1 , 1,… 1 };
process Philosopher [i = 0 to 99] {
       while (alive)  {
              P(fork[i]); P(fork[i+1]);
              eat;
              V(fork[i]); V(fork[i+1]);
              think;
       }
}
```

(a) Suppose all the philosophers want to eat simultaneously. How many of them can do so in the worst, most restricted, case? Demonstrate your claim with a trace. (b) Improve your solution so that it will scale to a row of arbitrary length, i.e. it will not cause a bottleneck where all forks must be requested waiting behind the same semaphore(s) (= centralized solution).

4. Explain the differences in synchronization mechanisms between Java synchronized class and the classical monitor. Give an outline of how the implement a monitor using Java synchonized class.

5. A simple roundabout (traffic circle) is a circular junction of streets where traffic flows in a one-way and one-lane circular stream around a central island. Vehicles in the roundabout have priority over the entering vehicle. A vechile may enter from any street and leave for any street. Assume that a roundabout connecting symmetrically N streets contains N "one vechile" slots on each cross-road connecting a street to the circle. In order for a car to get through from street s to t it has to occupy all consequtive slots one by one from s to t-1. Outline a synchronization protocol which allows the cars to use the shared resource of the roundabout in way that they do not crash to each other on the cross-roads, nor on each other's back in the circle and give priority to those in the roundabout. Use either semaphores or monitors for synchronization.

1. Consider the following program:

```
co    < await   (x   >=   3)    x =   x - 3; >
//    < await   (x   >=   2)    x =   x - 2; >
//    < await   (x   ==   1)    x =   x + 5; >
oc
```

(a) For what initial values of x does the program terminate assuming that scheduling is weakly fair? What are the corresponding final values? (b) Write a proof outline to demonstrate your claim.

2. Suppose we have to write a busy-wait and weakly fair implementation for semaphores for a machine that has atomic increment and decrement instructions. INC(x) atomically adds 1 to integer variable x, and DEC(x) atomically subtracts 1 from integer variable x. Both INC(x) and DEC(x) return the modified value of x. Answer with an concise argument to the following questions: (a) Is the solution safe ? (b) Is the solution free from deadlock (c) Is the solution free from livelock? (d) What is wrong with it performance wise? Correct the solution in cases needed. (e) Is your solution fair after this ?

```
/* P(s): */
while (DEC(s) < 0) {
       INC(s);  #undo decrement
       }

/* V(s): */
INC (s)
```

3. Dining philosopher on a row: Five philosophers sit along a long table spending their time thinking and eating. There are six forks on the table, one on each side of every philosopher. For eating a philosopher needs exclusive access to both of the forks.

```
sem fork[6]= {1, 1, 1 , 1 , 1,  1};
process Philosopher [i = 0 to 4] {
       while (alive)  {
              P(fork[i]); P(fork[i+1]);
              eat;
              V(fork[i]); V(fork[i+1]);
              think;
       }
}
```

(a) What is bad with this solution? Demonstrate your claim with a trace. (b) Improve your solution so that it will still scale easily to an arbitrary number of N philosophers.

4. Explain the differences in synchronization mechanisms between Java synchronized class and the classical Signal-and-Wait (SW) monitor. Give an outline of how the implement SW-monitor using Java.

5. A simple roundabout (traffic circle) is a circular junction of streets where traffic flows in a one-way and one-lane circular stream around a central island. Vehicles in the roundabout have priority over the entering vehicle. A vechile may enter from any street and leave for any street. Assume that a roundabout connecting symmetrically N streets may contain at most 2N "standard size" vehicles at any time, i.e. one on each cross-road from a connecting street to the circle and one on the circle between each consecutive cross-roads. Outline a synchronization protocol which allows the cars to use the shared resource of the roundabout in way that they do not crash to each other on the cross-roads, nor on each other's back in the circle and give priority to those in the roundabout. Use either semaphores or monitors for synchronization.

T-106.5600 Concurrent Programming Examination 8.1.2008 / hsa

1. Three ascending integer sequences A, B and C contain at least one common value. Consider the following concurrent program aimed to find the smallest of such values, X say:

```
int a,c,b;
a = head(A); b = head(B); c = head(C);     # head(X) places the cursor on the first item of X and
co   while true if (a < b) a = next(A);    # returns its value, next(X) sifts the cursor to the next
//   while true if (b < c) b = next(B);    # item of X and returns its value.
//   while true if (c < a) c = next(C);
oc
```

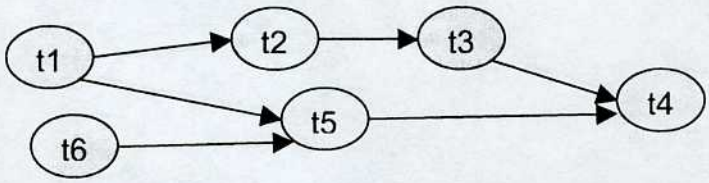
a) Let's assume, that the `if`-statements of three loop bodies are executed as atomic actions. Show that I: { ($a \leq X$) and ($b \leq X$) and ($c \leq X$) } is an invariant of the program, and based on it argue, that the program works correctly, i.e. it reaches the state R: {a = b = c = X }.
b) Assume now that each individual reference to the shared variables a, b and c is atomic. Explain whether or not I still an invariant of the program.
c) Does the program terminate, or not, and why? Show how to enhance it for proper termination and explain that R always holds at the termination.

2. The following algorithm is supposed solve the critical section problem for n processes. a) Why is the *delay* necessary, explain. b) How long it should be ? c) Is there a way to reduce it in low-contention situations?

```
integer gate = 0
process i:
    loop forever
            non-critical section
            loop
    p1:             await gate == 0
    p2:             gate = i
    p3:             delay
    p4:     until gate == i
            critical section
    p5:     gate = 0
```

3. In the following graph nodes represent tasks (= processes) and arcs indicate the order in which they should be executed. A task may start as soon as all its predecessors have completed:

```
process t_i: {
    wait for predecessors if any;
    body of the task;
    signal successor if any;
    }
```



a) Show how to synchronize processes $t_1, t_2, \ldots t_6$ using semaphores. b) Minimize the number of semaphores needed.

4. a) Develop a general monitor which allows a pair of processes to exchange values. The monitor has only one operation `int exchange(int value)`. After two processes have called it, the monitor swaps their values given in the calls and let them proceed further. When the next pair of calls have arrived, the swap is repeated, and so on. State whether your monitor uses signal-and-wait or signal-and-continue semantics. b) Modify your solution to work with Java. Remember that in Java processes released from wait() with a notify() call are put to the same queue with new entries, and that a wait() may also end spontaneously.

5. Solve the producer-consumer problem for many producers and consumers and for an unlimited FIFO-buffer using tuple-space. The data-elements are contained in the tuples. The "put" and "get" operations should be simple, straight–line algorithms without loops.