T-106.420 Concurrent Programming Examination 31.1.2005 /hsa

1. Consider the following concurrent program aimed to terminate in a state: x = y = m, where m is some value < K, and both m and K are divisible by 3 and 5. Assume that the underlying hardware provides the normal LOAD and STORE instructions, so that in the program each individual reference (i.e. read or write) to an integer can be considered atomic.

```
int   x = 3, y = 5;
co    while (x != y) x = (x + 3)%K;
//    while (x != y) y = (y + 5)%K;
oc    {P}
```

a)Does the program meet the requirements of the At-Most-Once Property? Explain. (b) Give a trace where the program terminates. (c) Does it always satisfy P: {x = y = m < K} upon termination? Give an invariant and proof outline. (d) Does the program terminate always, and if not, give a trace that does not end. e) What is required from the scheduling to terminate the program?

(a) No. Eg. expression `(x != y)` has two critical references, one to x and one to y.
(b) Assume K =30. `(x,y)` pairs: (3,5); (6,5); (9,5); (12,5); (15,5); (15,10); (15,15); (termination)
(c) Yes. If the program terminates, it terminates in a state where P holds.
Invariant: I ={(0 ≤ x < K) and (x%3 = 0) and (0 ≤ y < K) and (y%5 = 0)} follows immediately.

```
int   x = 3, y = 5; {I}
co    {I}while (x != y) x = (x + 3)%K;     #px
//    {I}while (x != y) y = (y + 5)%K;     #py
oc    {I}
```

Suppose, without loosing generality, that px terminates first after evaluating `(x != y)` to be false, so that {x=y=m }must hold for px and for some m. Because of I, {m%3=0 and m%5=0} must hold also. After px terminates {x = m} will be preserved. Now, because K is finite, py is bound to reach the state: {y=m }, so that also it terminates, and the whole program terminates so that P holds.
(d) No. Any trace where there is no (x,y) pair such that {(x%5=0) and (y%3=0)}
(e) With total randomness the probability of such a pair is 1/15 and probability of a trace with length n with such a pair is: $1-(14/15)^n$. Answer: A scheduling policy which will exclude this anomaly.

Grading: a), b), d) and e) 1cp, c) 2cp.

2. Consider the following solution to the critical section problem:

```
int lock = 0;
process CS[i = 1 to n] {
       while (true) (
              <await (lock == 0)>; lock = i; Delay;  ##
          while (lock != i) {
              <await (lock == 0)>; lock = i; Delay;
          }
          critical section;
          lock = 0;
          noncritical section;
       }
}
```

a)Why is the line marked by "##" included? Explain. (b) Suppose the *Delay* code is deleted. Does the protocol ensure mutual exclusion? Does it avoid deadlock? Does it avoid unnecessary delay? Does it ensure eventual entry for everybody? Explain. (c) Suppose the processes execute with true concurrency on a multiprocessor. Suppose the *Delay* code spins for long enough to ensure that every process i that waits in: `<await (lock==0)>` has time to execute the: `lock = i,` before any other process j may proceed to the test: `while (lock != j).` Does the protocol now ensure mutual exclusion, avoid deadlock, avoid unnecessary delay and ensure eventual entry?

---

a) To reduce the memory contention on the variable `lock` by the first entry into the competition.

b) No mutual exclusion. Assume two processes p and q are waiting on `<await (lock == 0)>;` for a third process that upon exit of the CS will execute `lock = 0;` Suppose: (1) that both p and q are able to evaluate `(lock == 0)>` and get out from the `<await (lock == 0)>;` before neither of them has been able to set `lock = i;` (where `i` is `p` or `q`) and (2) both of the are able to execute the two statements: `lock = i; while (lock != i)` without interruption, then both will enter the critical section at the same time. Example trace:

```
p: <await (lock == 0)>;
q: <await (lock == 0)>;
p: lock = p; while (lock != p); CS
q: lock = q; while (lock != q); CS
```

No unnecessary delay (livelock). If there is no process in the CS `(lock == 0)` and several are competing for the entry, lock may be set by several process in: `lock = i;` before any test in: `while (lock !=i)` but the last `lock = i;` will stay intact and the corresponding process will get in.

No unnecessary delay => No deadlock.

No eventual entry ensured. A process may be bypassed by others without limits when competing for the entry.

c) Because of the *Delay*, assumption (2) does not hold anymore. So mutual exclusion is guaranteed. Other properties do not depend on it, so they won't change.

Grading: a) 1 cp, b) 4cp , c) 1cp

3. Dining philosopher problem: Five philosophers sit around a round table spending their lives thinking and eating. There are five forks on the table one between every two philosophers. For eating a philosopher needs exclusive access to both of the forks beside him. Consider the following solution using semaphores. Each philosopher has three states: think, eat and *hungry,* where he wants to eat but can't, because one or both of his neighbors are eating. The state of philosopher i is coded in *s[i]* and a semaphore *ph[i]* is used enable a hungry philosopher i to wait for a proper time to eat.

```
#define ln (i-1)%5  /* to abbreviate the references to */
#define rn (i+1)%5  /* the neighbors of the philosopher i */
#define lln (i-2)%5
#define rrn (i+2)%5
sem ph[i = 0 to 4] = {0,0,0,0,0}, mutex = 1;
s[i = 0 to 4] = (think,think,think,think,think};
process Philosopher [i = 0 to 4] {
    while (true)  {
       P(mutex);
          if (s[ln]!=eat and s[rn]!=eat) s[i]=eat
          else {s[i]= hungry; P(ph[i])};
       V(mutex);
       eat;
       P(mutex);
          s[i] = think;
          if (s[lln]!=eat and s[ln]==hungry) {s[ln]= eat; V(ph[ln]};
          if (s[rrn]!=eat and s[rn]==hungry) {s[rn]= eat; V(ph[rn]};
       V(mutex);
       }
}
```

(a) There is a fatal error in this. Demonstrate it with a trace, and correct the solution. (b) Is the solution safe? May it deadlock? Does it exclude starvation and livelock ? (c) What can be said about utilization of the forks in this solution? Explain. (d) Replace the semaphores with a monitor and condition variables.

---

a) Trace:

1. Philosopher[1] goes to eat: `P(mutex); s[1]= eat; V(mutex); eat…`
2. Philosopher[2] wants to eat, but because (`s[1]== eat`) it gets hungry and waits: `P(mutex); … s[2]= hungry; P(ph[i])` ***WAIT***
3. Philosopher[1] wants back from eat to think: `…eat; P(mutex}` ***WAIT***
4. ***DEADLOCK BECAUSE mutex is still "hold" by Philosopher[2]***

Take the wait for philosopher specific condition "`P(ph[i])`" out from the critical section:

```
P(mutex);
    if (s[ln]!=eat and s[rn]!=eat) {s[i]=eat; V(mutex)}
    else {s[i]= hungry; V(mutex); P(ph[i])};
```

b)

- Safe: Yes, we can prove the following invariance to hold always outside the critical sections:
  `Philosopher[i]` in "`eat;`" `=> (s[i]== eat) => (s[ln]!=eat and s[rn]!=eat)`
- No Deadlock:
  1. No process can get stuck in P-operation in a CS guarded by:
     `P(mutex);… V(mutex);`
  2.*) The following invariance holds outside of the CS's:
     `Philosopher[i]` waits in `P(ph[i])}`
     `=> (s[i]== hungry)`
     `=> (s[ln]==eat or s[rn]==eat) =>` No circular wait.
- No livelock:
  No loops included in the solution.
- Starvation not excluded:
  The (four!) neighbours of philosopher[i] could conspire against him to by alternating at the table so that he will not get to eat. Demonstrate with a trace.

c) Because of property 2.*)
`Philosopher[I]` waits
=> `Philosopher[ln]` eats or `Philosopher[rn]` eats
=> Resource utilization is optimal.

d) The CS's *in the original version* are simply replaced by monitor procedures, and likewise the `ph`-semaphores and operations on them with corresponding cond-variables and -operations:

```
monitor Dining_Room {
  int s[i = 0 to 4] = (think,think,think,think,think};
  cond ph[0:4];
  /* constants ln,rn,lln,rrn depending on i are defined as before */

  procedure enter_eat(int i){
     if (s[ln]!=eat and s[rn]!=eat) s[i]=eat
     else {s[i]= hungry; wait(ph[i])
     }
  procedure exit_eat(int i){
     s[i] = think;
     if (s[lln]!=eat and s[ln]==hungry) {s[ln]= eat; signal(ph[ln])};
     if (s[rrn]!=eat and s[rn]==hungry) {s[rn]= eat; signal(ph[rn])};
     }
}

Philosopher[i]:…    while true {
                           think;
                           enter_eat(i);
                           eat;
                           exit_eat(i);
                           }
```

Grading: a) 2cp, b) 2cp, c) 1 cp, d) 1cp

4. Consider the following Java solution for Readers and Writers problem:

```
class ReadersWriters {
  int nr = 0;
  private synchronized void startRead() {
    nr++;
  }
  private synchronized void endRead() {
    nr--;
    if (nr==0) notify();  //awaken waiting Writers
  }
  public void read() {
    startRead();
    …                        // embed read operation here!!
    endRead();
  }
  public synchronized void write() {
    while (nr>0)
      try { wait(); }
         catch (InterruptedException ex) {return;}
    …                        // embed write operation here!!
    notify();
  }
}
```

(a) Does it provide concurrent access to the readers? Does it exclude concurrent access during writing ? (b) What is the purpose of the line: `catch (InterruptedException ex) {return;}`? (c) Explain the roles of the three methods: `public void read()`, `private synchronized void startRead()`, `private synchronized void endRead()` included in the class. (d) Is the solution fair? Explain. (e) Modify the solution so that it gives preference to the writers.

---

a) Yes and Yes.

b) It enables to catch an exception to continue without writing in case of some fault in other process.

c) `public void read()` is needed to provide one interface to the clients. `private synchronized void startRead()`, `private synchronized void endRead()` are used for the synchronization needed for reading.

d) Writers may have to wait indefinitely if there is always at least one reader i.e. (nr > 0) holds.

e) Four changes:

 # 1. To exclude reading when the number of *interested writers*, `iw` is postive, i.e. `(iw>0)`, `start_read()` procedure will put the readers to wait in order to give preference to writers.
 # 2. To exclude writers from each other a `wlock` is needed to indicate a write is ongoing.
 # 3. In order to be able to schedule the waiting writers against the readers, the writers have to wait inside the monitor for a condition. So writing have to be split to three procedures like reading: `startWrite()`, `endWrite()` and `write()`
 # 4. Because of Java's restriction to have only one condition variable per synchronized class, both readers and writers have to wait behind it for their specific signaling condition in some random order. In order to enable all the waiting processes to evaluate their eligibility to proceed after signal they all have to be notified with `notifyAll();`

```
class ReadersWriters {
  int nr = 0, iw = 0,                                          # 1.
      wlock = 0;                                               # 2.

  private synchronized void startRead() {
     while (iw>0)                                              # 1.
       try { wait(); }
          catch (InterruptedException ex) {return;}
     nr++;
  }
  private synchronized void endRead() {
    nr--;
    if (nr==0) notifyAll(); //awaken possible Writers          # 4.
  }
  public void read() {
    startRead();
    …                          // embed read operation here!!
    endRead();
  }
  private synchronized void startWrite() {                     # 3.
     iw++;                                                     # 1.
     while (nr>0 or wlock != 0)                                # 2.
       try { wait(); }
          catch (InterruptedException ex) {return;}
     wlock++;                                                  # 2.
  }
  private synchronized void endWrite() {                       # 3.
     wlock--;                                                  # 2.
     iw--;                                                     # 1.
     notifyAll();       //awaken all possible waiters!!       # 4.
  }
  public void write(){
    startWrite();
    …                          // embed write operation here!!
    endWrite();
  }
}
```

Grading: a), b), c), d) 1 cp, e) 2cp

5. Sleeping barber problem: Several barbers work in a packed shop -where only one person can move at a time- with one barber chair for each barber and a waiting bench. When there are no customers, all free barber waits sleeping. When customer enters and finds a barber sleeping he awakens the barber, sits in the chair and falls to sleep waiting for the barber to finish the haircut. If all barbers are busy, the customers wait sleeping on the bench. After giving the haircut a barber opens the door for the customer to leave and awakens him and waits for the customer exit. After it he starts with a new customer, or if there is no one waiting he falls to sleep waiting for a new customer to arrive. A monitor for n barbers:

```
monitor Barber_Shop {      # for n barbers
  int barber=0, chair[n]=0, open[n]=0;
  queue free_barbers;      # int queue of process id's for free barbers
  cond barber_available;   # signaled when barber>0
  cond chair_occupied[n];  # signaled when chair[]>0
  cond door_open[n] ;      # signaled when open[]>0
  cond customer_left[n];   # signaled when open[]==0

  procedure get_haircut {
     int BarberID;
     while (barber == 0) wait(barber_available);
     BarberID= get(free_barbers);barber= barber - 1;
     chair[BarberID]= chair[BarberID] + 1;
     signal(chair_occupied[BarberID]);
     while (open[BarberID]== 0) wait(door_open[BarberID]);
     open[BarberID]= open[BarberID]-1; signal(customer_left[BarberID])
     }

  procedure get_next_customer(int BarberID){
     put (free_barbers,BarberID); barber= barber+1;
     signal(barber_available);
     while (chair[BarberID] == 0) wait(chair_occupied[BarberID]);
     chair[BarberID]= chair[BarberID]-1;
     }

  procedure finished_cut(int BarberID){
     open[BarberID]= open[BarberID]+ 1; signal(door_open[BarberID]);
     while (open[BarberID] > 0) wait(customer_left[BarberID])
     }
}
```

(a) Assume that the system is designed for a large set of barbers and customer, so that the contention for one common monitor should be minimal in order to avoid it to become a bottleneck. In the above solution most of the code in the later parts of the monitor procedures is involved in an interaction between a specific customer-barber pair, which is formed when the customer acquires the barber ID, So why not separate this part of the monitor functionality to barber specific monitors to avoid overloading the common monitor? Draft the necessary changes to the program and explain. (b) The customer gets the barber ID who is serving him. Symmetrically the barber would also like to get the customer ID. Draft the necessary changes to the program code and explain.

---

a) The binding between the barber and the customer is done in the `Barber_Shop` monitor. The rest of the functionality, indicated in the above code by ***bolded italics*** , is synchronization between this pair, and can be done in the barber specific monitor: `Barber[i]`

```
monitor Barber_Shop {      # for n barbers
  int barber =0,
  queue free_barbers;      # int queue of process id's for free barbers
  cond barber_available;   # signaled when barber>0

  procedure get_free_barber() returns int { #customer wants a free barber
     int BarberID;
```

```
      while (barber == 0) wait(barber_available);
      BarberID= get(free_barbers);barber= barber - 1;
      return BarberID;
      }
   procedure get_next_customer(int BarberID){ #barber wants a customer
      put (free_barbers,BarberID); barber= barber+1;
      signal(barber_available);
      }
}

monitor Barber[n] {
   int chair =0, open =0;
   cond chair_occupied;     # signaled when chair >0
   cond door_open;          # signaled when open >0
   cond customer_left;      # signaled when open ==0

   procedure get_haircut{   # customer sits in the chair of the barber
      chair = chair + 1;    # assigned to him and starts to wait
      signal(chair_occupied);
      while (open== 0) wait(door_open);
      open = open-1; signal(customer_left)
      }
   procedure start_cut(){   # barber waits for the customer to sit in oder
      while (chair== 0) wait(chair_occupied); # to start the haircut
      chair = chair-1;
      }
   procedure finish_cut(){  # barbers is done and wakeups the customer
      open= open + 1; signal(door_open);
      while (open > 0) wait(customer_left)
      }
}
```

b) In order to transmit the customer id to the barber we may use any of the monitor static varaibales e.g. `chair` so that whenever a customer sit in the chair, it's nonzero `CustomerID` is passed as parameter and assigned to the variable `chair` , from which it can be passed to the barber process:

```
procedure get_haircut(int CustomerID){   # customer sits in the chair of
      chair = CustomerID;   #the barber assigned to him and starts to wait
      signal(chair_occupied);
      while (open== 0) wait(door_open);
      open = open-1; signal(customer_left)
}
procedure start_cut() returns int;{ # barber waits for the customer to
      int CustomerID;                # sit in oder to start the haircut
      while (chair== 0) wait(chair_occupied);
      CustomerID = chair;
      chair = 0;
      return (CustomerID);
}
```

The code outline of the processes:

```
customer_process[i]
        …
        barber = Barber_shop.get_free_barber();
        Barber[barber].get_haircut(i);
        …
barber_process[i]
        …
        Barber_shop.get_next_customer(i);
        customer = Barber[i].start.cut();
        Barber[i].finish_cut();
```

Grading: a 4cp, b) 2cp.