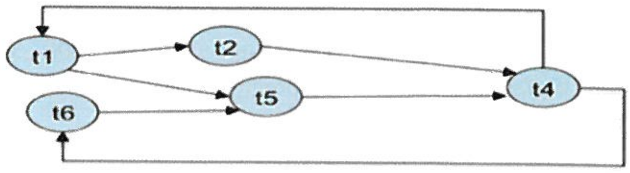
T-106.5600 Concurrent Programming		Examination		August 2010/ hsa

1. In the following graph the five nodes: t_1, t_2, t_4, t_5, t_6 represent repeatable 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 as indicated in the pseudocode on the left. Initially tasks t_1 and t_6 should be able to start immediately without stopping in the wait.

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



a) Show how to synchronize the processes with binary semaphores by repeating the pseudocode for each of the five processes, insert the necessary P() and V() operations in them, and add the semaphore initializations.
b) Revise your solution to a) using general semaphores and minimizing the number of semaphores needed. Be concise and clear. The shorter you can be, the better.

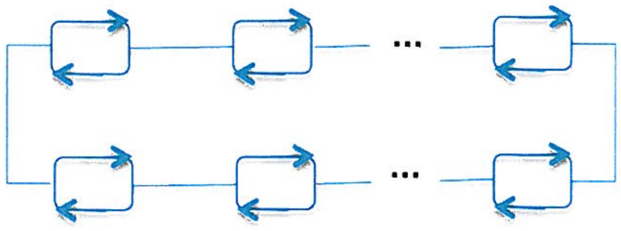
2. Three sets of n (n ≥2) integers A, B and C have to be sorted so that A will contain the n largest, B the middle ones C the n smallest elements, i.e. the final state R should be: {min(A) ≥ max(B) and min(B) ≥ max(C)}. The sets are accessed with atomic operations `getmin(s)`/(`getmax(s)` which pick out the smallest/largest element of the set `s`, and `put(x,s)` which puts the element x back in `s`. Consider a parallel sorting program consisting of three threads AB, BC and AC, where the integer variables: `x,y` are local to each of the threads:

```
co while true {          #thread_AB
     x= getmin(A); y= getmax(B); if (y > x){put(y,A); put(x,B)} else {put(x,A); put(y,B)}}
// while true {          #thread_BC
     x= getmin(B); y= getmax(C); if (y > x){put(y,B); put(x,C)} else {put(x,B); put(y,C)}}
// while true {          #thread_AC
     x= getmin(A); y= getmax(C); if (x > y){put(y,A); put(x,C)} else {put(x,A); put(y,C)}}
oc
```

a) Explain why R will be reached by finding a proper invariant I, and a monotonic function f:(A,B,C) -> int describing progress towards R on each atomic action, or disprove the claim with a counter example.
b) Describe how the program can be easily scaled up to an arbitrary number of sets and threads.
c) The threads do not terminate. Enhance so that they will terminate when R has been reached so that the solution is still scalable, i.e. each thread is using only variables accessed by at most two processes.

3. A long single-track railway loop connects N stations each having two parallel tracks capable of hosting K trains to both directions so that the trains to opposite directions may pass each other. Consecutive trains to each direction go automatically to their own tracks and do not collide to each other. Because trains can't back off, they must be synchronized with semaphores against head-on collision and deadlocks. Introduce necessary general semaphores with proper initializations and fill in their P() and V() operations to the following train code. Initially all the T trains on both directions are evenly distributed to "stay" on stations. Because of symmetry, the solution to one direction suffices. Analyze the safeness and fairness of your solution for different values of K, and T : a) T ≤ K, b) K < T < K*N, c) T = K*N, d) Because of heavy traffic to one direction on any sector of k consecutive stations (k<<N), we are asked to allow: k*K ≤ T < k*(K+1).

```
while true{
    stay_on_station(i);
    i = (i+1)mod N;
    enter_track(i);
    run_on_track(i);
    leave_track(i);
}
```



4. Use Linda tuplespace to solve the five dining philosophers problem. Your solution should be simple, straight-line code containing just a few lines without loops and the only global variables used are the tuples.
Linda tuple-space primitives are: `postnote ('tag', …)`, `readnote ('tag', …)`, `removenote ('tag',…)`. Define clearly the meaning of the different tuple types used and attach appropriate tags to them. Indicate clearly in a `readnote(…)`, or `removenote(…)` operation when a matching tuple with an element value equal to a program variable value is sought for (syntax " `v=`"), from the case where an element value of a otherwise matching tuple is just assigned to a program variable (syntax " `v`"). Define the safety deadlock and fairness for the problem and analyze your solution against them.

5. a) Implement a simple bounded buffer of one String element in Java. The execution environment in this assignment is J2SE 5 (Java 1.5) or Java SE 6 (Java 1.6).

The class should implement the following interface:

```
public interface SimpleList {
      public String get();
      public void put(String element);
}
```

The required semantics are as follows:

If the buffer contains exactly zero elements, the buffer is empty.
If the buffer contains exactly one element, the buffer is full.

The buffer is always either empty or full.

If a value is inserted by put(), it is returned by the next get().

If null is inserted, the behavior of the buffer is undefined. That is, you are allowed to assume null will never be inserted.

get() and put() must not block.

If get() is called when the buffer is full, get() immediately removes the element from the buffer and returns it.

If get() is called when the buffer is empty, get() returns immediately with the return value null.

If put() is called when the buffer is empty, put() immediately inserts the element into the buffer and returns.

If put() is called when the buffer is full, put() throws a RunTimeException (or a subclass thereof).

b) Improve your solution to a) by implementing the following changes to the semantics to allow operations to wait until they can proceed:

get() and put() may block only as required by the following requirements.

If get() is called when the buffer is empty, get() waits until an element is available, removes the element and returns it.

If put() is called when the buffer is full, put() waits until there is space in the list, inserts the element and returns.

c)
Prove that your answer to b) is correct.