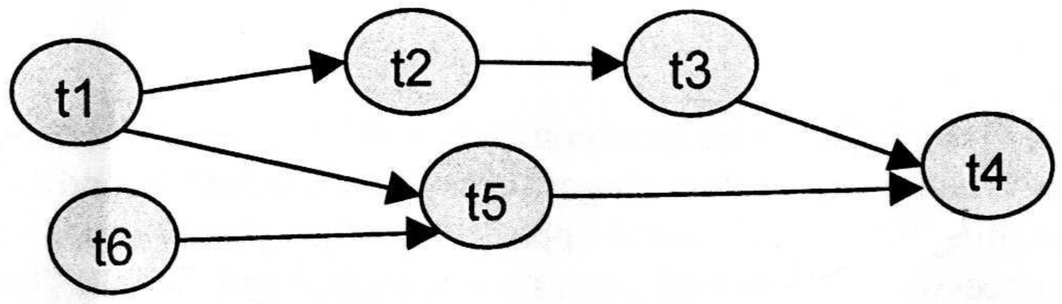
1. In the following graph the six nodes represent tasks (= processes) and arcs indicate the order in which they should be executed. A task may start as soon as all its predecessors have completed:

```
process t_i: {
    while true{
        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.
c) Extend the system with arcs from t4 back to t1 and t6 respectively with the restriction that for each completed execution of t4, there may be at most one more completed execution of t1 and t6. Repeat a) and b) for this.

2. Three ascending integer sequences A, B and C contain at least one common value. Consider the following program consisting of tree concurrent threads aimed to find the smallest of such values, X say. Operation `head(s)` places the cursor on the first item of sequence s and returns its value, and `next(s)` sifts the cursor to the next item of s and returns its value.

```
int a,c,b;
a = head(A); b = head(B); c = head(C);
co   while true if (a < b) a = next(A);
//   while true if (b < c) b = next(B);
//   while true if (c < a) c = next(C);
oc
```

a) Let's assume, that the `if`–statements in loop bodies are executed as atomic actions. Give an argument that I: { (a ≤ X) and (b ≤ X) and (c ≤ X) } is an invariant of the program, and based on that show 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) Why doesn't the program terminate properly? Show how to enhance it for proper termination and explain whether or not R always holds at the termination of your enhanced version for the two different versions of atomicity a) and b) above.

3. A long single-track railway line connects two end stations, 0 and n. It's cut to n rail segments with n-1 intermediate stations each having two parallel tracks capable of hosting at most N trains to each direction so that they may pass each other. Introduce necessary semaphores and fill in their P() and V() operations to the following code for a trains to synchronize them against the following hazards:
- Trains should not collide to each other: At most one train can be on any rail segment at any point of time
- Because trains do not back off, take care that no deadlock may happen
- Pay special attention to the first and last segment
- The opposite traffic from station #n to station #0 is symmetric and not to be repeated.
- Consider and analuze the trade-off between fairness vs. optimal resource usage in this context

```
leave_station(0) & enter_segment(1);
for i = 1 to n-1 {
        run_on_segment(i);
        leave_segment(i)& enter_station(i);
        stay_on_station(i);
        leave_station (i)& enter_segment(i+1)
}
run_on_segment(n);
leave_segment(n)& enter_station(n);
```

4. The simple barrier synchronization monitor has only one method: `Wait_for_all_n()`. Its semantics of is as follows: The calling process is put to wait "behind the barrier" if there are less than n-1 processes waiting "before it". In the opposite case "the barrier is opened", i.e. the process will "release all the waiting processes", and all n processes will continue and pass the "barrier". Consider the following solution written as a traditional monitor using semantics, where all the new calls for `Wait_for_all_n()` monitor operation are blocked as long as there are processes free to proceed inside the monitor.

```
monitor Simple_Barrier {
        int k =0;               // number of processes behind barrier
        cond barrier;           // condition variable to implement the wait

        public operation Wait_for_all_n() {
                k++;
                if (k  <  n) then wait(barrier)
                else {
                        k = 0;
                        signal_all(barrier)
                }

        }
}
```

Write a version using Java. Analyze its safety and fairness.

5. Solve the simple barrier -problem using tuple-space and its primitives.