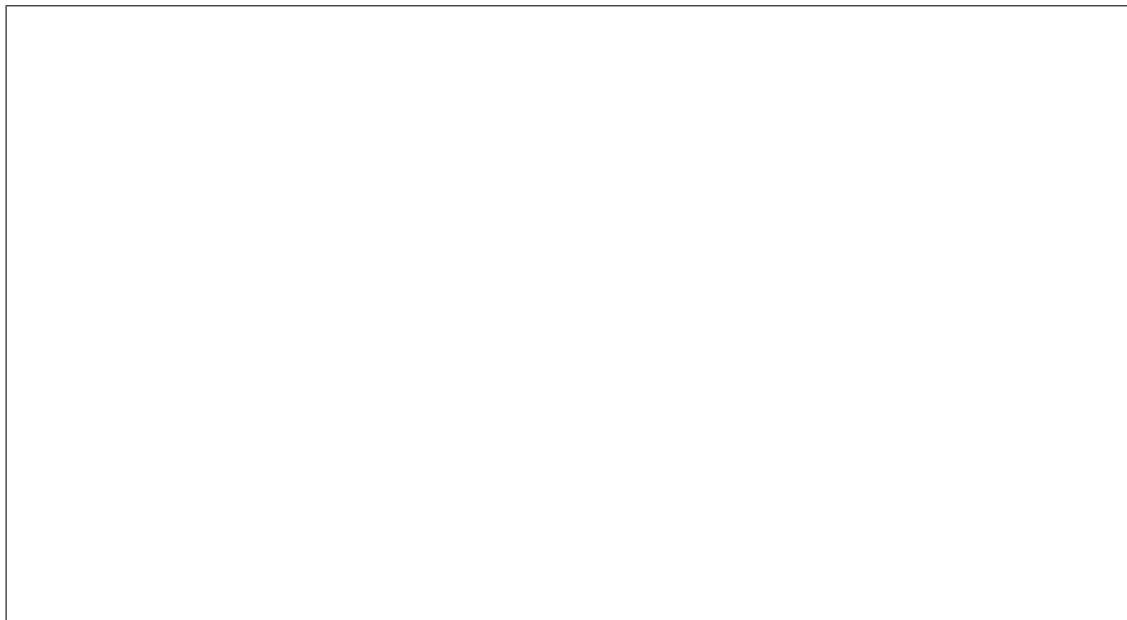


1 Code Quality

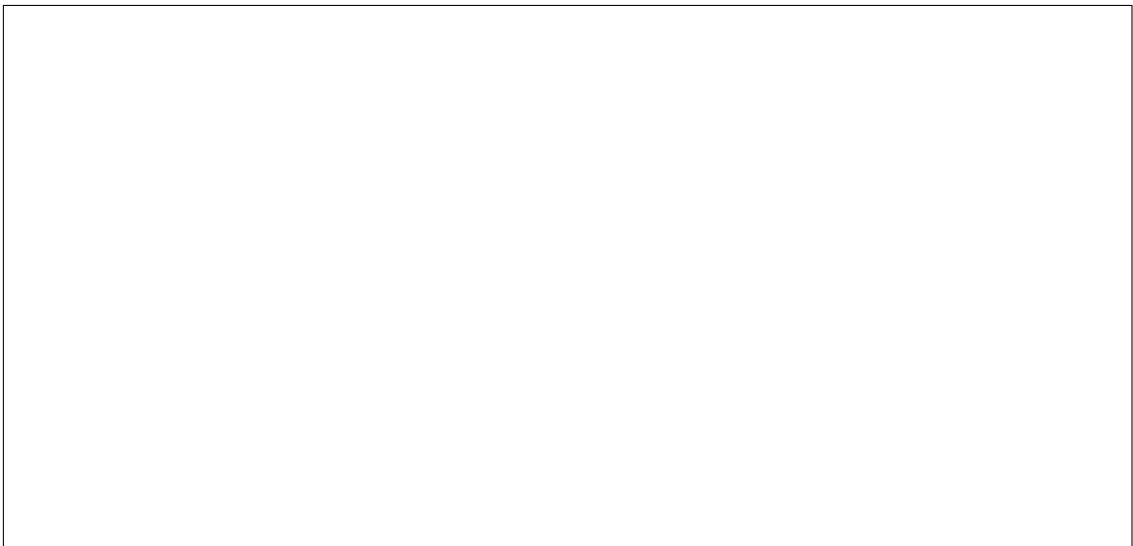
1-1) In the listing under is a module interface (a C header file) from an old project deliverable. The modules name is “cost”. Criticize (concisely, with bullet points) the design.

```
1 #ifndef lift_cost_h
2 #define lift_cost_h
3
4 int calculateCost(int currentFloor, int direction, int orderedFloor, int
    orderedDirection);
5
6 int downCost[MAX_ELEVATORS][N_FLOORS];
7 int upCost[MAX_ELEVATORS][N_FLOORS];
8
9 void fillCostArrays();
10 void clearCosts(void);
11
12 int lowestCostFloor(int elevator);
13 int lowestCostDirection(int elevator);
14
15 int findBestElevator(int floor, int direction);
16
17 void designateElevators();
18 void clearDesignatedElevator();
19
20 int designatedElevator[N_FLOORS][2];
21
22 #endif
```



1-2) This is another module interface. This time with the name of “jobqueue”. Criticize it; feel free to compare it to the previous one.

```
1  #ifndef _JOB_QUEUE_H
2  #define _JOB_QUEUE_H
3
4  #include <time.h>
5
6  //Set flag at given floor in the given direction.
7  void set_flag(int floor, int direction);
8
9  //Deletes flag at given floor in the given direction.
10 void delete_flag(int floor, int direction);
11
12 //Returns one if flag at given direction at given floor is set.
13 int get_flag(int floor, int direction);
14
15 //Deletes all flags in table.
16 void delete_all_flags();
17
18 //Returns one if there are no flags set in table.
19 int flag_table_empty();
20
21 //Returns one if there are any flags above the given floor.
22 int any_flags_above(int floor);
23
24 //Returns one if there are any flags below the given floor.
25 int any_flags_below(int floor);
26
27 //Prints the table of flags to the terminal.
28 void print_flag_table();
29
30 #endif
```



1-3) This listing (under) is the function “delete_flag” from the module in the previous task. Criticize (shortly and in bullet points).

```

1 //Delete flag in table.
2 void delete_flag(int floor, int direction) {
3
4     if(floor == 0) {
5         flag_table[floor][DIR_UP] = 0;
6         flag_table[floor][DIR_DOWN] = 0;
7         elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
8     }
9     else if(floor == 3) {
10        flag_table[floor][DIR_UP] = 0;
11        flag_table[floor][DIR_DOWN] = 0;
12        elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
13    }
14    else{
15        if(!any_flags_above(floor) && get_current_direction() ==
16            DIR_UP) {
17            flag_table[floor][DIR_UP] = 0;
18            flag_table[floor][DIR_DOWN] = 0;
19
20            elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
21        }
22        else if(!any_flags_below(floor) && get_current_direction() ==
23            DIR_DOWN) {
24            flag_table[floor][DIR_UP] = 0;
25            flag_table[floor][DIR_DOWN] = 0;
26
27            elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
28        }
29
30        flag_table[floor][direction] = 0;
31    }
32
33    if(direction == DIR_UP && floor < 3) { elev_set_button_lamp(
34        BUTTON_CALL_UP, floor, 0); }
35    if(direction == DIR_DOWN && floor > 0) { elev_set_button_lamp(
36        BUTTON_CALL_DOWN, floor, 0);}
37
38    elev_set_button_lamp(BUTTON_COMMAND, floor, 0);
39 }

```

2 Fault Tolerance

2-1) Acceptance tests is seen as an important tool for handling errors. What do we gain from using acceptance tests in addition to the more traditional tests on error conditions?

2-2) Give examples of what one can test for when making acceptance tests. (Hint: A good answer would be the 'learn-by-heart' list; 7 items).

2-3) Backward error recovery is some times seen as not suited in a real time system. What is backward error recovery, and why is it not suited?

2-4) Backward error recovery may, when generalizing to multi-thread systems give the domino effect. Explain. How can we avoid the domino effect?

2-5) When doing forward error recovery in a multi thread setting, the need arises for the different threads to get to know about errors that happen in other threads. List mechanisms that can be used to convey such information between threads.

2-6) Writing to log is an alternative to creating recovery points. How does this work in the context of (single-thread) backward error recovery?

2-7) Handling the log gets more difficult if we have more parallel tasks, where some succeeds and some fails, and all generate log. How can we extend the log (compared to the single-thread version) to handle this (still in a backward error recovery perspective)?

2-8) What is an Atomic Action? Which problem(s) is Atomic Actions meant to solve?

2-9) Describe one way of implementing an Atomic Action.

2-10) Operations for locking resources are always assumed to be atomic. Why is this so important?

3 Shared Variable Synchronization

You have already made a module for keeping track of names of people. It has, among other parts of the interface, these functions: *getFirstName*, *getLastName*, *setFirstName* and *getName*. The last one calls the two first ones before it returns the complete name. The module worked perfectly until a multithreaded version of the program was made...

3-1) What kind of problems with such a module can surface when it is used in a multithreaded program?

3-2) The *Suspend()* and *Resume(Thread)* calls have been described as unusable when it comes to programming error free synchronization between threads. Explain why.

3-3) “Resumption model” is used to describe possible implementations of both signals, asynchronous notification in general and exceptions. In most cases it is seen as less useful than “termination model”. What is the difference? Why is resumption model less useful?

3-4) Shared variable synchronization may be criticized for poor scalability. Give, shortly, arguments for this.

To increase the abstraction level and give flexibility compared to semaphores, POSIX, Java and Ada have landed on different variants of monitors: POSIX combines mutexes and condition variables, Java has synchronized methods and *wait/notify/notifyAll*, and Ada has guarded entries in protected objects.

3-5) (2x) Describe shortly how these three works.

3-6) Compare the Java and Ada mechanisms here, give the main strengths and weaknesses.

3-7) (implementation,pseudo code,2x) We have a resource in our system that is used by many threads, creating the need for synchronizing access. When there are more waiting threads the *last* request should be given priority. Write pseudo code for the `allocate()` and `free()` functions that achieves this. Use the synchronization mechanisms in C/POSIX, Ada or Java as you prefer.

3-8) Nested calls to monitors (that one function in a monitor makes a call to another monitor) is a problem that leads to monitorbased systems not scaling well. What is the problem?

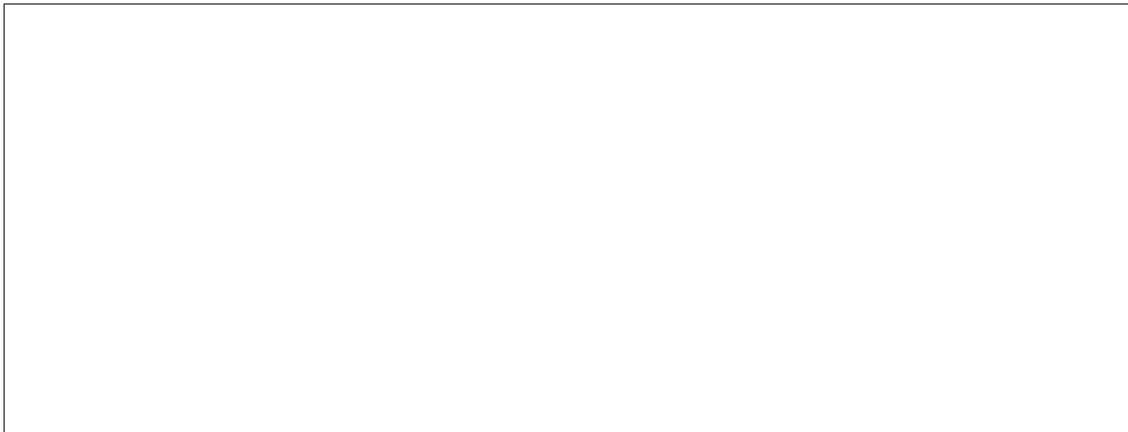
We have started a number of threads T_1 to T_N . T_1 initializes some data structures that the other threads use so that they should be blocked until T_1 has finished initializing. One way of doing this with semaphores is the following: (A is a binary semaphore initialized to 0.)

```
T1() {
    // Do the initialization
    signal(A);
    // Continue working
}

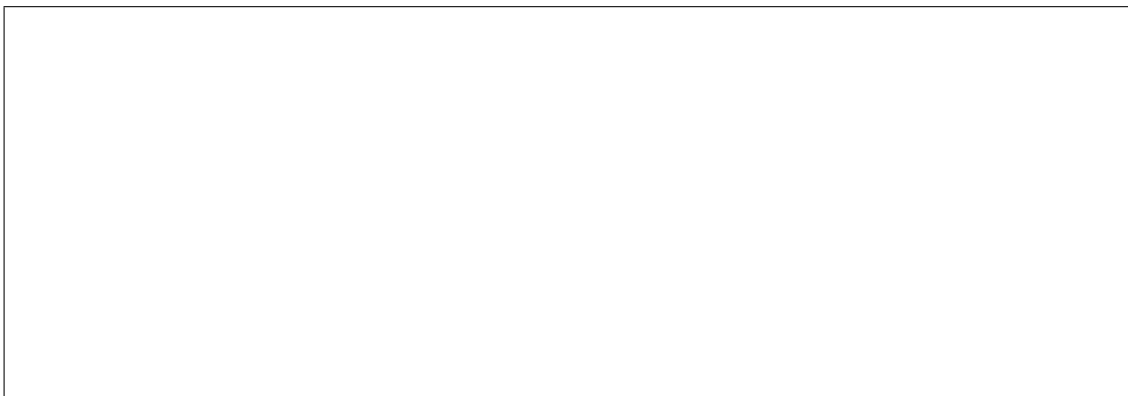
T2() {
    wait(A);
    signal(A);
    // Data structures are initialized,
    // continue working
}
```

$T_3 \dots T_N$ is like T_2 .

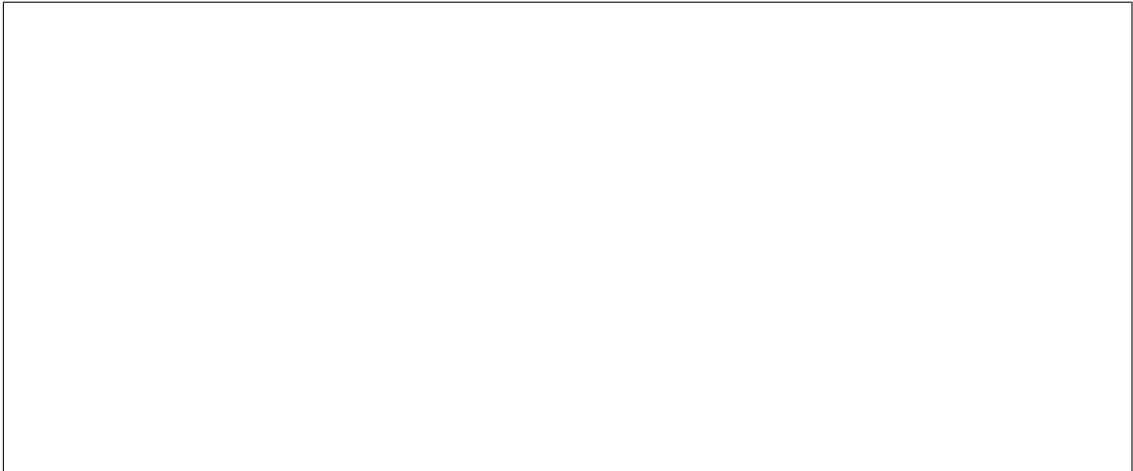
3-9) Assume $N=3$. Model the system in FSP. (If you want to avoid the problem of terminating processes you can model “Continue Working” as $DW = (doWork \rightarrow DW)$.)



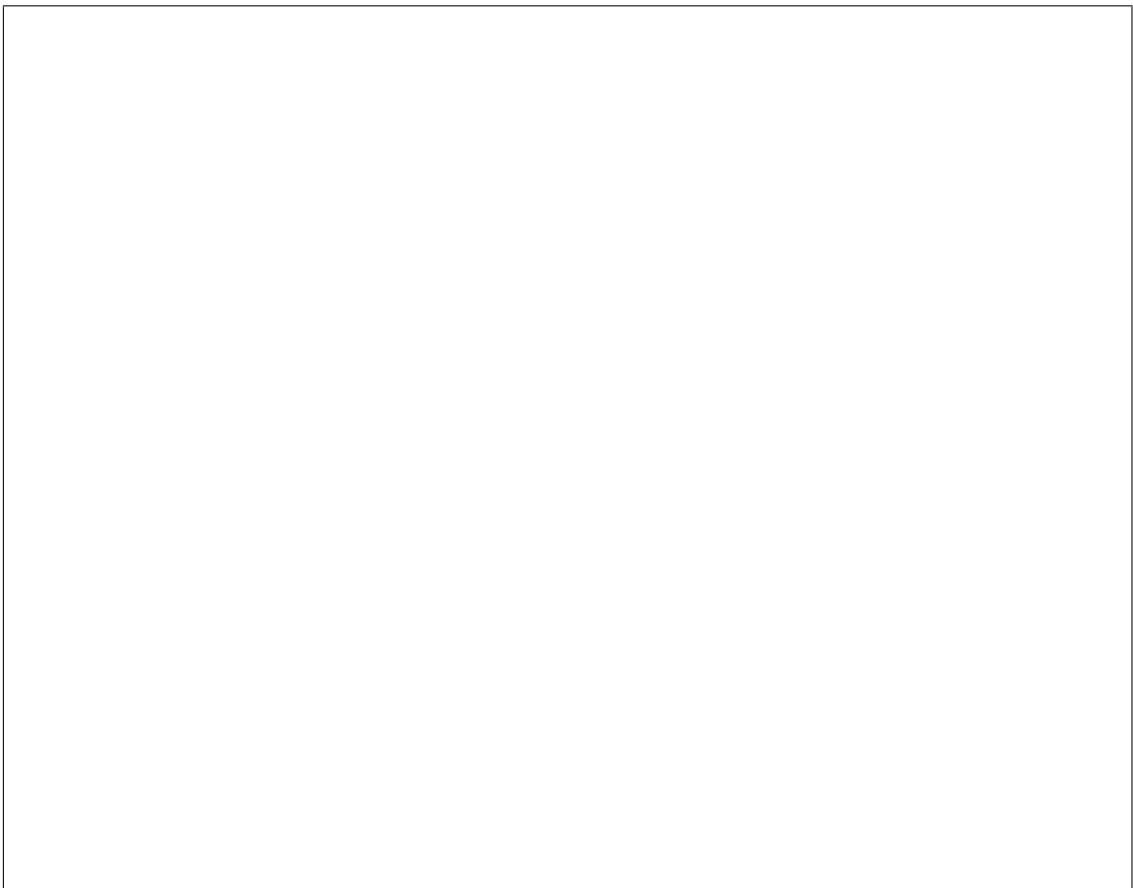
3-10) What is a live-lock, and how can this be recognized in a transition diagram?



3-11) With N still equal to 3: Draw the transition diagram for the composed system.



3-12) (Implementation/Pseudo-code) How would you solve this problem of holding back threads $T_2...T_N$ in Java (using synchronized, wait, notify and notifyAll)?



3-13) A general advice on Java programming is to avoid using *notify()* and rather use *notifyAll()* combined with while loops around all the *wait()*'s. Why?

3-14) (Implementation/Pseudo-code) How would you solve this problem of holding back threads $T_2...T_N$ in Ada (using protected objects, functions, procedures, entries and guards)?

Look at the given semaphore variant of the program again. This program leaves the semaphore A in an open state. Now we want to reuse the semaphore and the signaling effect that the initialization is finished, and must therefore reset the semaphore. (Imagine an initialization process that happens more times during the execution of the program.)

3-15) Now the program looks like this. Assume that g_N and $g_nOfThreadsInitialized$ is global variable initialized to the number of threads and 0 respectively.

```
T1() {
    while(1) {
        // Do the initialization
        signal(A);
        g_nOfThreadsInitialized++;
        if(g_nOfThreadsInitialized == g_N) {
            g_nOfThreadsInitialized = 0;
            wait(A);
        }
        // Continue working
    }
}

T2() {
    while(1) {
        wait(A);
        signal(A);
        g_nOfThreadsInitialized++;
        if(g_nOfThreadsInitialized == g_N) {
            g_nOfThreadsInitialized = 0;
            wait(A);
        }
        // Data structures are initialized,
        // continue working
    }
}
```

This program has a race condition (at least). What is a race condition and what is the problem(s)?

4 Scheduling

4-1) Good scheduling is important in real-time programming to ensure that all deadlines in the system are satisfied. How do we achieve this satisfying deadlines by scheduling strategies?

4-2) We can prove that the deadlines in the system will be satisfied by response time analysis and utilization-based schedulability tests. Explain shortly how these two work.