NTNU
Norwegian University of
Science and Technology

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi
matematikk og elektroteknikk

Eksamen i TTK4145
Sanntidsprogrammering
Never 3000
9.00-13.00
Sensor veiledning

Sensuren vil bli avsluttet i henhold til gjeldende regelverk.

Generelt:

Vær konsis og klar. Skriv kort. Sett opp momenter og argumenter punktvis, bruk figurer der det egner seg. Du kan få trekk for «vås». I utgangspunktet teller alle deloppgavene likt; Unntaket er oppgavene markert med (2x) eller (3x) som teller hhv. dobbelt eller tredobbelt.

Generally:

Be short and clear. Enumerate items and arguments, use figures where appropriate. You may be punished for text without substance. All questions have similar weights except where marked with (2x) or (3x).

Hjelpemidler:

Tillatte hjelpemidler: D - Ingen trykte eller håndskrevne hjelpemidler tillatt. Bestemt, enkel kalkulator tillatt.

1 Semaphores and shared variable synchronization

1-1) (Used in: v2015)

The following code is taken from a module called “logic” in an old lift control project.

```
1  #include <stdio.h>
2
3  #include "elev.h"
4  #include "control.h"
5  #include "logic.h"
6
7  // The order tables; one cell is unused in each table; ref. the floor panel
8  static int calls_up[N_FLOORS];
9  static int calls_down[N_FLOORS];
10
11 // helper function that is used by place_order(...) and place_call(...)
12 // to store orders in the order tables.
13 static void add_job(int path, int floor);
14
15 // Sverre: More functions omitted...
16
17 void place_order(int destination)
18 {
19   // cancelling emergency stop
20   if (emergency && !cancel_emergency) {
21     restart(destination);
22     return;
23   }
24   // The two border cases
25   if (destination == 0) {
26     calls_up[destination] = TRUE;
27   }
28   else if (destination == N_FLOORS - 1) {
29     calls_down[destination] = TRUE;
30   }
31   // The three normal cases
32   else if (destination > current_floor) {
33     calls_up[destination] = TRUE;
34   }
35   else if (destination < current_floor) {
36     calls_down[destination] = TRUE;
37   }
38   else if (destination == current_floor) {
39     add_job(direction, current_floor);
40   }
41 }
42
43 void restart(int destination)
44 {
45   cancel_emergency = TRUE;
46   // If STOP has been pressed between two floors
47   if (!on_floor()) {
```

```
48      if (destination == current_floor) {
49        // The lift is over current_floor, needs to go down
50        if (ref_floor > current_floor) {
51          direction = DOWN;
52        }
53        // The lift is under current_floor, needs to go up
54        else {
55          direction = UP;
56        }
57      }
58      else if (destination < current_floor) {
59        direction = DOWN;
60      }
61      // destination > current_floor.
62      else {
63        direction = UP;
64      }
65      start_motor();
66    }
67    place_order(destination);
68  }
```

Criticize the code; point out any strengths and weaknesses.

Expectations:

I'll try to order my comments from important to less important;

- The code itself is tidy and easy to read. The level of commenting is appropriate.
- place_order should place an order. Nobody could guess that it also started the lift after an emergency stop.
- The code is too dependent on some global variables (direction,emergency,cancel_emergency, ref_floor...).
- The mutual recursion between the two functions can hardy be motivated.
- the add_job helper function is not used other than in the final case... Why?
- "logic" will never be a good name for a module.
- current_floor is not the floor the lift is in?? Very confusing.
- What they achieve by letting the order handling also care about lift control is that they manage with only two ordertables. Cool!
- start_lift would be a better name than restart.
- why does restart just set cancel_emergency? It has just been tested on by the calling function. What motivates *this* as the correct time to do this? Maybe this is just terminating the recursion... Confusing.

- ...

The student is not expected to reconstruct this list in any way, but should at least touch on some of the 3-4 important ones.

1-2) (Used in: v2015)

In the corresponding header file we find:

```
1  // Takes a command from the comparment panel. If STOP has been pressed
2  // the the lift is to start again, this by calling restart(...). After
3  // this, orders can be entered normally again. If the lift is ordered
4  // to a floor over its position or to the lowest floor the order goes
5  // into the up-table. Oppositely for a floor under its position or the
6  // highest floor. If the lift is ordered to the floor it visited last,
7  // or the floor it is in, the table depends on the direction of the
8  // lift.
9  void place_order(int destination);
10
11 // Is called only in the emergency state by place(order...) and will
12 // then cancel the emergency state. If the function is called to start
13 // the lift after STOP has been pressed in a floor, only returned the
14 // control to place_order(...) and the lift will start. If, however,
15 // the function is called after a press of STOP between floors, the
16 // variable ref_floor (set by update_floor()) is used to decide the
17 // lifts position relative to the last floor. The motor can then be
18 // started in the correct direction by returning the control to
19 // place_order.
20 void restart(int destination);
```

Criticize the code — that is, the comments; point out any strengths and weaknesses.

Expectations:

I have little love for comments like this; I find the code easier to understand than the comments, and that the comments did not help me understand the code in any significant way.

However the students may appreciate it, and I shall not judge.

Any reasonable comments should be rewarded.

1-3) (Used in: v2015)

The complete header file looks like this (after Sverre removed all the comments; evaluate the code as if the comments never were there):

```
1  #ifndef __INCLUDE_LOGIC_H__
2  #define __INCLUDE_LOGIC_H__
3
4  void change_direction(void);
```

```
5  void clear_orders(void);
6  void place_call(int direction, int floor);
7  void place_order(int destination);
8  void delete_order(void);
9  void restart(int destination);
10 int called_here(void);
11 int visible_orders(void);
12 void handle_order(void);
13 void debug_logic(void);
14
15 #endif
```

Criticize the code - that is, the module interface this time; point out any strengths and weaknesses.

Expectations:

- The module have two purposes; keeping track of orders and some odd functions having to do with the lift. It would be better to separate these responsibilities.
- The keeping track of orders part of the interface is nice enough (clear_orders, place_call, place_order, delete_order)
- ...even though I cannot guess what visible_orders() and handle_order() does
- Some of the exported functions are not called from the outside I would guess. (They exported all functions to get to write all function documentation in the headerfile, I speculate?) This is unnecessary bloat.
- "Order" vs "call" is differing nicely between compartment orders and corridor orders, but the naming falls together when order suddenly means both.

1-4) (Used in: v2015)

What purpose does process pairs fill?

Expectations:

Obviously fault recovery, but also importantly availability, minimizing service down-time.

We could speculate further for bonuspoints; paving the road for online upgrade?

1-5) (Used in: v2015)

Acceptance tests are an "enabeling technology" for process pairs. Explain specificly how acceptance tests contributes to the functionality of a process pair.

Expectations: It is extremely important that no (unexpected) errors propagate from the primary to the backup. The status messages *must* be error free.

Acceptance tests is the mechanism that ensures this.

A perfect answer should contain the sequence;

Do work — perform acceptance test — send status to backup — do side-effects.

If the primary crashes, the slave executes the (possibly duplicate) side effects.

1-6) (Used in: v2015)

Define the terms deadlock and race condition.

Expectations:

Deadlock: More parts of the system waits for each other in a curcular wait, locking the system in a state it cannot get out of.

Race Condition: A bug that surfaces by unfortunate timing or order of events.

1-7) (Used in: v2015)

Can deadlocks and race conditions happen in a message passing system?

Expectations: Yes

The Java documentation states this about *which* thread is awakened when notify is called:

> If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

The use of "arbitrary" here is quite typical, and descriptive of the documentation of many synchronization primitives.

1-8) (Used in: kont2015 , v2015)

Sverre states that there is only one reasonable way to implement this — first in, first out (FIFO); the thread that has been waiting the longest will be the one that is awakened — and that all implementations most probably will choose this one as the default behavior.

Why is FIFO a reasonable choice here?

Expectations: Any other strategy might lead to starvation, depending on the applica-

tion.

1-9) (Used in: kont2015 , v2015)

If the FIFO strategy for waking processes is the reasonable one; why does the documentation so often insist on "arbitrary"?

Expectations: If the application did make assumptions on the detailed workings of the kernel here, it would introduce unwanted dependencies between the parts of the program, breaking module boundaries.

Having to make "assumptions" about other modules is a break of module boundaries itself.

This code is from Burns and Welling (if somewhat simplified) and has a bug.

```
1  // The priority parameter is either 0 or 1
2  // The mutex M is initialized to 1, the semaphores in the
3  // PS array is initialized to 0
4  void allocate(int priority){    void deallocate(){
5    Wait(M);                        Wait(M);
6    if(busy){                       busy=false;
7      Signal(M);                    waiting=GetValue(PS[1]);
8      Wait(PS[priority]);           if(waiting>0) Signal(PS[1]);
9    }                               else{
10   busy=true;                        waiting=GetValue(PS[0]);
11   Signal(M);                        if(waiting>0) Signal(PS[0]);
12  }                                  else{
13                                       Signal(M);
14                                     }
15                                   }
16                                 }
```

We have a resource used by a lot of threads, but rather than allocating the resource to the threads in the order they requested it, threads that gave the high priority flag as a parameter should get it first.

1-10) (Used in: v2015)

What is the bug: What happens? What is the consequence of the bug?

Expectations:

The second last user of the resource calls deallocate and interrupts the last user in the process of calling allocate, just after the signal(M) at line 5. When the last thread then waits on its semaphore, it will not be awoken (until after any next thread uses the resource).

Switching order of two requests, switching priorities, delaying a request indefinitely.

1-11) (Used in: v2015)

This error is not easily fixed using only semaphores. Why? What is the difficulty?

Expectations: Throwing the students out on deep water here. Any halfway reasonable answer must be rewarded :-)

Sverres answer: This is an example demonstrating the limitations of semaphores. Semaphores are not suited for solving some of the more complex synchronization problems.

1-12) (Used in: v2015)

Give a short description of how the synchronization primitives in Java (synchronized methods, wait, notify and notifyAll) works.

Expectations: Java: Any method in a java object can be denoted "synchronized", which means that calls to this method will happen under mutual exclusion with other synchronized methods.

wait()**:** A call to *wait*() will suspend the current thread; It will be resumed by a call to one of the *notify*() calls (or by somebody calling *interrupt*() on the thread)

notify() wakes an (arbitrary?) thread blocked by this objects lock.

notifyAll() wakes all threads blocked by this objects lock.

1-13) (Used in: v2015) **(2x)** Sketch (pseudo-code) how you could solve the problem from task 1f with Javas synchronization primitives.

Expectations: Using the "catch-all" code from the lectures will be sufficient here. I am sure other solutions exist.

We assume a queue data-structure on (threadId,priority), sorted on the priority.

```
1  synchronized allocate(int pri){
2    if(busy){
3      queue.insertSorted(myThreadId,pri);
4      while(busy || queue.getFirst() != myThreadId) wait();
5      queue.removeFirst();
6    }
7    busy = true;
8    // notifyAll() //? If more can be allocated at the same time.
9  }
10
11 synchronized free(){
12   busy = false;
13   notifyAll();
14 }
```

1-14) (Used in: v2015)

Give a short description of how the synchronization primitives in ADA (Protected Objects, functions, procedures and entries with guards) works.

Expectations: Ada: A protected object is a module, a collection of functions, procedures and entries along with a set of variables.

Functions are read-only, and can therefore be called concurrently by many tasks, but not concurrently with procedures and entries.

Procedures may make changes to the state of the object, and will therefore run under mutual exclusion with other tasks.

Entries The important thing is that these are protected by guards - boolean tests - so that if the test fails, the entry will not be callable - the caller will block waiting for the guard to become true. These tests can only be formulated using the object's private variables.

1-15) (Used in: v2015)

We think about implementing the functionality of 1f in Ada:

Even though the Java and Ada synchronization mechanisms are of approximately the same expressive power, this problem is not so straightforward to solve in Ada as it were in Java (or it would be with posix, monitors, conditional critical regions etc.).

Why?

Expectations: The guards of Ada entries can not test on entry parameters. (... forcing us to use some kind of double interactions or using language mechanisms like requeue or entry families.)

... so we make two allocate functions instead of one; one for requests on low priorities (allocate_lowPri()) and one for requests on high priorities (allocate_highPri()).

1-16) (Used in: v2015) **(2x)** Sketch (pseudo-code) how you could solve this problem with Adas synchronization primitives.

Expectations: Sverres shoot-from-the-hip Ada Pseudo-code:

```
1 protected object resource
2
3   private busy = false;
4
5   entry allocate_highPri when not busy
6   begin
```

```
7       busy = true;
8     end
9
10    entry allocate_lowPri when not busy and allocate_highPri`count == 0
11    begin
12      busy = true;
13    end
14
15    entry free
16    begin
17      busy = false;
18    end
19  end
```

1-17) (Used in: v2015)

Message based interaction between threads leads to a very different design than shared variable synchronization. How? Describe shortly the difference in designs.

Expectations: Shared variable synchronization focuses on avoidance of the problems with more threads sharing common resources. Apart from the added complexity of synchronization the threads look like "normal"; programs working on data.

Messagepassing systems have ideally no shared resources; each resource is managed by a thread, and other threads must access the resource by communicating with this.

Most threads in a messagepassing system is built around the while-select loop.

There are usually far more threads in a messagepassing system since we have more reasons to create them (...like managing resources)

1-18) (Used in: v2015) **(2x)** The underlying situation in the code from task 1f — controlling access to a shared resource dependent on priority — would not occur in a system based on messagepassing. Describe a corresponding situation in the messagepassing domain (some creativity might be neded? But the two priority levels should be present.) and write pseudo-code for a solution.

Expectations: For example: A server handling requests for operations on two channels ch_high_pri and ch_low_pri.

```
1  while (1) {
2    do { // while there is something on the high-pri channel.
3
4      select {
5        request ? ch_high_pri {
6          // Handle request
7          high_pri_empty = false;
8        }
9        default: high_pri_empty = true
10     }
```

```
11    } while (!high_pri_empty)
12
13    select {
14      request ? ch_low_pri {
15        // Handle request
16      }
17      default: // Do nothing
18    }
19  }
```

1-19) (Used in: v2015)

Compare the complexity of your implementation in the previous task with the code given in task 1f. Assume that a reasonable measure of complexity is the number of states the program can be in, if you made a formal model of it, e.g. with FSP. (...but you do not need to do any modeling)

Explain why the two cases compare as they do?

Expectations: This is an open question, and any mature answer yields points.

The nominal answer would be that there are significantly more states in the semaphore-based system than in the message based system.

1-20) (Used in: v2015)

The priority ceiling protocol, in addition to solving the unbounded priority inversion problem, also have the property that it avoids deadlocks in the system. Explain how.

Expectations:

The trick is that since we know beforehand which resources a given thread uses, and that the priority of this resource is set to max+1 of all the using threads, it is impossible for any thread owning a given resource to be interrupted by any other thread also (potentially) wanting the same resource.

As soon as T1 has allocated resource A, T2 will not even get to run (so that it can allocate resource B), since it has lower priority than T1 now has.

We have made a module that maintains a table of Person-structs and that contains these functions among others:

```
1  void printName(int i){
2    printf("%s",persons[i].firstName);
3    printf("␣%s\n",persons[i].lastName);
4  }
5  void printAddress(int i){
6    printf("%s",persons[i].street);
7    printf("␣%d\n",persons[i].streetNumber);
```

```
8 }
```

The module worked well until we made the program multithreaded...

1-21) (Used in: kont2014)

Some times the program prints strange errors in the address list (Sverre Hendseth and Ola Normanns entries are mixed up):

```
...
SverreOla Normann
Osloveien 23
 Hendseth
Trondheimsveien 34
...
```

What is the problem?

Expectations: One thread gets interrupted while printing a person, leaving another thread to print its person before the first one gets to finish.

1-22) (Used in: kont2014)

...and even worse: Occasionally this can happen even inside of words:

```
...
SveOla Normann
Osrre Hendseth
Trondheimsveien 34
loveien 23
...
```

What is the problem here?

Expectations: The library that printf comes from is not reentrant.

1-23) (Used in: kont2014)

What do we call such faults that occur “occasionally”?

Expectations: Race Conditions

1-24) (Used in: kont2014)

We solve the problem by protecting the functions in the module with semaphores. The module now looks like this:

```
1  // Assume a semaphore personSem initiated to 1 at the start of the program
2  void printName(int i){
3    sem_wait(personSem);
4    printf("%s",persons[i].firstName);
5    printf("␣%s\n",persons[i].lastName);
6    sem_signal(personSem);
7  }
8  void printAddress(int i){
9    sem_wait(personSem);
10   printf("%s",persons[i].street);
11   printf("␣%d\n",persons[i].streetNumber);
12   sem_signal(personSem);
13 }
```

How do using semaphores like this solve the problem?

Expectations: This is basicly "how do semaphores work". If a thread is interrupted while inside of a wait/signal region, other threads cannot wait on the semaphore, and will be blocked rather than be allowed to print anything.

1-25) (Used in: kont2014)

As the program grows it turns out that the combination of printName() and printAddress() occurs many places in the program (when printing name tags, letterheads, business cards, envelopes,...), and we decide to make the function *printNameAndAddress* as a part of the module:

```
1  void printNameAndAddress(int i){
2    sem_wait(personSem);
3    printName(i);
4    printAddress(i);
5    sem_signal(personSem);
6  }
```

Look at the module interface; What do you think about this decision in a code quality perspective?

Expectations: The interface is not minimal any more (which is a bad thing).

Continuing this trend will lead to code duplication in the module. (More obscurely: We get dependencies between functions in the module which increases module complexity).

But of course; we sometimes do make convenience functions, if the convenience is great enough :-)

1-26) (Used in: kont2014)

However, this does not work at all: The program never prints any combination of name and address; It just hangs.

What happens?

Expectations: The semaphore is wait'ed on two times after each other. At the second call to wait the semaphore is 0 and the thread will block.

1-27) (Used in: kont2014)

What do we call this kind of fault?

Expectations: Deadlock

1-28) (Used in: kont2014)

In this situation it might be an alternative to drop the semaphore protection of *printNameAndAddress*? Why would/wouldn't this be ok?

Expectations: Not ok; we would be back to erroneous printing in the address list.

1-29) (Used in: kont2014)

This problem is not easily solved with semaphores. I would say that we have an example of the limited "expressive power" of semaphores here. Explain the term "expressive power".

Expectations: Expressive power: The range of problems that can be solved (reasonably effective) with the language mechanism.

1-30) (Used in: kont2014)

In Java this problem would be easy to solve: Describe shortly how?

Expectations: Make the module an object, and the functions synchronized. Only one thread at a time can call functions in the object. (When the object lock is owned by the thread, it will not be block when calling another synchronized function in the same object.)

1-31) (Used in: kont2014)

In Ada (that is, not using semaphores) this problem would be easy to solve: Describe shortly how?

Expectations: Make the module a protected object, and the functions procedures of the object. (When the object lock is owned by the thread, it will not be block when calling another synchronized function in the same object.)

1-32) (Used in: kont2014)

With POSIX mutexes the problem does not solve itself; we have to set the mutex in "recursive" mode (PTHREAD_MUTEX_RECURSIVE). How do you think a mutex in

this mode works?

Expectations: The student should be able to reason on this.

If one thread waits on a mutex more than once, a counter is incremented that keeps track of the number of signals necessary before the semaphore is released.

1-33) (Used in: kont2014)

If you really have to protect one of your modules using only semaphores; How would you approach it?

Expectations: Not have functions in the interface that calls each other.

In our case: Make printName_imp() and printAddress_imp() private functions that do not reserve the semaphore, but that that both printName/printAddress and printNameAndAddress calls.

1-34) (Used in: kont2014)

More of these "resource allocation" problems does not occur in pure message passing systems ("message-based synchronization" as Burns&Wellings calls it). Why?

Expectations: There are no shared resources.

1-35) (Used in: kont2014)

(2x) A slightly larger part of our module is shown here:

```
1  #ifndef PERSON_H
2  #define PERSON_H
3
4  typedef struct {
5    char * firstName;
6    char * lastName;
7    char * street;
8    int streetNumber;
9  } TPerson;
10
11 void reallocateArray(int newSize);
12
13 TPerson ** getArray();
14
15 void printName(int personNumber);
16 void printAddress(int personNumber);
17 void printNameAndAddress(int personNumber);
18 ...
19 #endif
```

Criticize, from a code quality perspective, the inclusion of the type TPerson, and the two functions *reallocateArray*() and *getArray*()

Expectations:

- The type should not be a part of the module interface - breaks the encapsulation!
- returning the list in getArray() is *really* terrible! - breaks the encapsulation!
- reallocateArray() breaks encapsulation revealing more of the implementation than should be necessary; Ideally it should not be here, but anyway it represents a non-consistent abstraction.

1-36) (Used in: v2014)

Bloom used "expressive power" and "ease of use" to compare mechanisms for synchronization. Explain the two terms.

Expectations:

Expressive power: What problems can be solved (reasonably) by the language/language mechanisms.

Ease of Use: Is the solution, using the mechanisms, elegant, straightforward, easy to understand and maintain?

1-37) (Used in: v2014)

Give a short description of how the synchronization primitives in ADA (Protected Objects, functions, procedures and entries with guards) and Java (synchronized methods, wait, notify and notifyAll) works.

Expectations: Ada: A protected object is a module, a collection of functions, procedures and entries along with a set of variables.

Functions are read-only, and can therefore be called concurrently by many tasks, but not concurrently with procedures and entries.

Procedures may make changes to the state of the object, and will therefore run under mutual exclusion with other tasks.

Entries The important thing is that these are protected by guards - boolean tests - so that if the test fails, the entry will not be callable - the caller will block waiting for the guard to become true. These tests can only be formulated using the object's private variables.

Java: Any method in a java object can be denoted "synchronized", which means that calls to this method will happen under mutual exclusion with other synchronized methods.

```
1  protected body Bounded_Buffer is
2    entry Get (Item : out Data_Item) when Num /= 0 is
3    begin
4      Item := Buf(First);
5      First := First + 1; Num := Num - 1;
6    end Get;
7
8    entry Put (Item : in Data_Item) when Num /= Buffer_Size is
9    begin
10     Last := Last + 1; Num := Num + 1;
11     Buf(Last) := Item
12   end Put;
13 end Bounded_Buffer;
```

Figur 1: Bounded Buffer in Ada

wait()**:** A call to *wait*() will suspend the current thread; It will be resumed by a call to one of the *notify*() calls (or by somebody calling *interrupt*() on the thread)

notify() wakes an (arbitrary?) thread blocked by this objects lock.

notifyAll() wakes all threads blocked by this objects lock.

1-38) (Used in: v2014)

Compare the given (Fig. 4 and 5) implementations of a "Bounded Buffer" in Ada and Java wrt. "expressive power and ease of use". Which implementation do you think express the Bounded Buffer logic more clearly.

Expectations: The expressive power is obviously ok for both Ada and Java. *I* would say that the Ada solution is a tiny bit clearer; the guards directly express what condition is necessary. The student is allowed a different opinion, so this is a question of wether he/she has a reasonable argument.

1-39) (Used in: v2014)

In one of the synchronization problems that Bloom put up, Adas mechanisms fails. Explain.

Expectations: Ada fails at the case where an entry *parameter* is necessary for the resource allocation - like when allocating N of a resource. Guards cannot test on parameters (only private variables), leading (supposedly - the book does not demonstrate this) to "double interactions" or to the more complex application of the "requeue" or "entry families" mechanisms.

1-40) (Used in: v2014)

Read the attached page from "The little book of semaphores" about "Cigarette smokers

```
1   public synchronized void put(int item)
2   {
3       while (numberInBuffer == size) wait();
4       last = (last + 1) % size ;
5       numberInBuffer++;
6       buffer[last] = item;
7       notifyAll();
8   }
9
10  public synchronized int get()
11  {
12      while (numberInBuffer == 0) wait();
13      first = (first + 1) % size ;
14      numberInBuffer--;
15      notifyAll();
16      return buffer[first];
17  }
```

Figur 2: Bounded Buffer in Java

problem".

Write pseudo-code with Adas protected objects to solve this problem. You are to sketch the protected object that has the responsibility to wake the correct "smoker"-thread.

The agent is given and unchangeable; Assume that it calls a random pair of the three functions, procedures or entries (your choice) — Tobacco(), Paper() or Matches() — in our object to signal that the two given resources are avaiable each time the table is empty.

You need to write only the protected object, but it should be clear how the "smoker"-threads are awoken.

Expectations: What about:

```
1  protected body Table is
2
3    entry WakeMatchesThread when TobaccoOnTable and PaperOnTable
4    begin
5    end
6
7    entry WakeTobaccoThread when PaperOnTable and MatchesOnTable
8    begin
9    end
10
11   entry WakePaperThread when MatchesOnTable and TobaccoOnTable
12   begin
13   end
14
15   procedure Tobacco()
```

```
16    begin
17      TobaccoOnTable = True
18    end
19
20    procedure Matches()
21    begin
22      MatchesOnTable = True
23    end
24
25    procedure Paper()
26    begin
27      PaperOnTable = True
28    end
29
30  private
31    MatchesOnTable : Boolean = False;
32    TobaccoOnTable : Boolean = False;
33    PaperOnTable   : Boolean = False;
34
35  end Table;
```

1-41) (Used in: v2014)

Solve the same problem (pseudo-code), but this time with only semaphores as the synchronization mechanism.

The agent is also now given and unchangeable; Assume it calls signal on an arbitrary pair of the three semaphores sem_paper, sem_match and sem_tobacco each time the table is empty. (You *may* use if-statements or arrays of semaphores :-))

Hint: It does not have to be the smoker-threads themselves that wait on these semaphores. You may introduce one or more threads as mediators.

Expectations: We must make mediator threads that waits for the semaphores and checks the conditions for waking the threads.

```
1  // 3 statusvariables
2  bool tobaccoOnTable = false;
3  bool paperOnTable = false;
4  bool matchesOnTable = false;
5
6  // Make a mutex to avoid race conditions
7  Semaphore pusherMutex(1);
8
9  // Make the semaphores for the smoker threads to wait on.
10 Semaphore sem_wakeTobacco(0);
11 Semaphore sem_wakeMatches(0);
12 Semaphore sem_wakePaper(0);
13
14 void pusherTobacco(){ // Waits primarily on sem_tobacco
15   wait(sem_tobacco);
16   // We have tobacco!
```

```
17
18    wait(pusherMutex)
19      // Now, another incredient may already have been detected
20      if(paperOnTable){
21        // We have both paper and tobacco; wake the matches smoker
22        signal(sem_wakeMatches)
23        paperOnTable = false;
24      }else if(matchesOnTable){
25        // We have both matches and tobacco; wake the paper smoker
26        signal(sem_wakePaper)
27        matchesOnTable = false;
28      }else{
29        // No, This was first ingredient
30        tobaccoOnTable = true;
31      }
32    signal(pusherMutex)
33  }
34
35  void pusherPaper(){ ... }
36  void pusherMatches(){ ... }
```

1-42) (Used in: v2014)

As you read, Patil suggested the Cigarette smokers problem as an argument to indicate that the expressive power of semaphores are limited. Do you think he succeeded with his argument? (Explain shortly)

Expectations: This is also an open question, where the quality of the students argument is deciding. *I* think Patil fails - the not using conditionals assumption is too strong.

But then; Semaphores do have (imho) serious problems with expressive power, so it is just this case that does not demonstrate it well... We have been discussing protecting a module where the functions call each other, or reserving resource A, B or both at the same time as difficult cases in the lectures.

1-43) (Used in: v2014)

A bounded buffer is a buffer where a process/thread that attempts to read from an empty buffer or write into a full one gets blocket. Conceptually this can be modelled in FSP as:

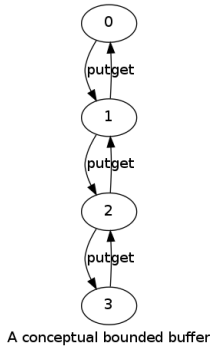
```
1  BUF(N=5) = BUF[0],
2  BUF[n:0..N] = (when n < N put -> BUF[n+1]
3                 | when n > 0 get -> BUF[n-1]).
```

Assume N=3: Draw the transition diagram for this system.

Expectations: Something like this:

```
SEMAPHORE NInBuffer(0),NFree(N),Mutex(1);

put(e){                          get(e){
  wait(NFree);                     wait(NInBuffer);
  wait(Mutex);                     wait(Mutex);
    // enter into buffer             // get e from buffer
  signal(Mutex);                   signal(Mutex);
  signal(NInBuffer);               signal(NFree);
}                                }
```

Figur 3: Bounded Buffer with semaphores



A conceptual bounded buffer

1-44) (Used in: v2014)

Look at Fig. 6 which is the bounded buffer implemented with semaphores. Make a model in FSP of this implementation.

Hint: Look at assignment 2a for some relevant syntax.

Expectations:

```
PUT = (waitFree -> waitMutexP -> signalMutexP -> signalOccupied -> PUT).
GET = (waitOccupied -> waitMutexG -> signalMutexG -> signalFree -> GET).
M = (waitMutexP -> signalMutexP -> M
  |  waitMutexG -> signalMutexG -> M).
Free(N=5) = Free[N],
Free[i] = when i > 0 waitFree -> Free[i-1]
     | signalFree -> F[i+1]
Occupied(N=5) = Occupied[N],
Occupied[i] = when i > 0 waitOccupied -> Occupied[i-1]
     | signalOccupied -> F[i+1]
```

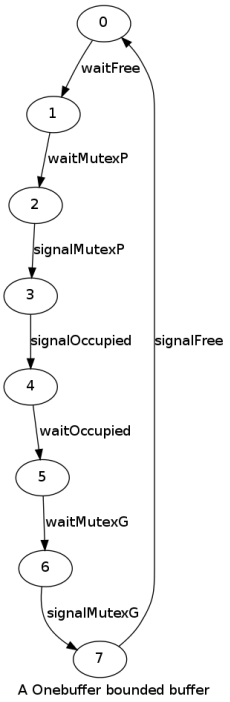
This is the complete model: Lots of code repetition here that the student can ... skip;

Any attemt at making a reusable semaphore model and using it for the three semaphores is good.

1-45) (Used in: v2014)

Assume that the buffer is of size 1 (N=1). Draw the transition diagram.

Expectations:



A Onebuffer bounded buffer

1-46) (Used in: v2014)

If you have got task 2a and 2c correct you will see that the model of the semaphore-based implementation of the Bounded Buffer is more complicated than the conceptual model — at least if we imagine buffers of the same size. Why?

Expectations: This is much of a maturation question again; Any sensible answer is good. In the lectures I discuss these three:

- Something wrong with how we model?
- The modeling abstraction level?
- The way the bounded buffer was implemented...

...where the last one is where I would place my bet. Semaphores is not *that* well suited for implementing something like a bounded buffer. (An "expressive power and ease of use" argument :-))

1-47) (Used in: v2014)

Pseudo-code: Sketch how a bounded buffer could have been implemented in a setting of message-passing processes. (Hint: Synchronous messages simplifies a bit.)

Expectations:

What about something like

```
1  BufferProcess(){
2    while(1){
3      select {
4        when (bufferNotFull) element <- putCannel:
5          -- enter Element Into Buffer.
6        when (bufferNotEmpty) <- getChannel:
7          -- retrieve an element and send it as reply
8      }
9    }
10 }
```

I assume guards in the select statement here. Go does not support this directly; one must manipulate the channel variables.

The while/select constellation should be there (great!), and the crucial point for a full score is that if the buffer is full we do not even listen to the put channel.

1-48) (Used in: v2014)

Comment on the expected (you do not need to write any code here) complexity of the transition diagram for the message based implementation of the bounded buffer.

Expectations: It will be the same model as for the conceptual model. (N+1 states)

1-49) (Used in: v2015 , v2014)

Message passing systems are not traditionally seen as very suited for implementing systems with real-time demands. Why?

Expectations:

- Schedulability proofs are not well developed.
- Traditionally we have in RT systems been closer to HW, maybe even without an OS. The message passing infrastructure might not be available,
- ...or it might not be to heavy/slow.
- In synchronization-based RT systems we have “One thread per timing demand” and we handle these threads with priorities. While in processoriented systems we

make threads of other reasons also, possibly making it difficult to assign priorities to them in any meaningful way.

- ... There are many other reasonable arguments here.

1-50) (Used in: v2014)

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
```

Expectations:

Any reasonable comment the student does should be rewarded. However, the "ideal solution" argues along the lines of the Code Complete checklists.

The main issue with this module is that this is not a module interface! It is (probably) a list of all functions in the module, including the variables!

- Ideally it should be clear from the interface exactly what the responsibility of the module is, and how it should be used correctly. It is not.

- There are (probably) nonobvious dependencies between the functions in the interface. (When do you need to call clearCosts for example?)

- The interface is not minimal. The functionalities of calculateCost, findBestElevator, lowestCostFloor and lowestCostDirection is overlapping. Also probably clearCosts and fillCostArrays.
- The abstraction is not consistent. The name "Cost" indicates that the module calculates or manages costs in some way. Either *calculateCost* or *findBestElevator* must be the main purpose of the module? But then there is the manipulation of some "costArrays" in addition - and keeping track of a "designatedElevator"?
- The data members are not encapsulated.
- (Having non-external variables in the headerfile like this is a bug, in addition to the fact that exporting the module's data is bad form)

Possibly some of these thinks could have been mitigated by commenting, but this would still be a badly designed module interface.

1-51) (Used in: v2014)

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 gloor 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
```

Expectations: This module is(/seems) far better; We have a module governing some collection of flags. The function names are good - we can see what they do (...or... that is... until in fact looking at them - see the next task).

But there are still some smaller issues:

- The interface is not minimal; 8 functions for governing a flag table is a bit much.
- A hint of what a "flag" represents would be nice. The title probably gives it away, but then, maybe "order" or "job" might be better than "flag"? (Remember that the function name should be clear also when reading the calling code.)
- The comments are partly redundant.
- I cannot see why time.h is included.

1-52) (Used in: v2014)

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() ==
                       DIR_UP) {
16                     flag_table[floor][DIR_UP] = 0;
17                     flag_table[floor][DIR_DOWN] = 0;
18
19                     elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
20                 }
21                 else if(!any_flags_below(floor) && get_current_direction() ==
                        DIR_DOWN) {
22                     flag_table[floor][DIR_UP] = 0;
23                     flag_table[floor][DIR_DOWN] = 0;
24
25                     elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
```

```
26                  }
27
28                  flag_table[floor][direction] = 0;
29          }
30
31          if(direction == DIR_UP && floor < 3) { elev_set_button_lamp(
                BUTTON_CALL_UP, floor, 0); }
32          if(direction == DIR_DOWN && floor > 0) { elev_set_button_lamp(
                BUTTON_CALL_DOWN, floor, 0);}
33
34          elev_set_button_lamp(BUTTON_COMMAND, floor, 0);
35  }
```

Expectations: Puh. What? Delete a flag?

The one big issue with this function is that it does not do what it says. And this makes the one comment (that was useless to start with) worse than useless!

What it does is in addition to the two-liner it should have been, surprisingly adds functionality to set lamps. And this is not done very clearly either; an all too large number of nested if-statements is used, and the code for clearing flags in mixed and repeated more places.

- The name does not describe what the routine does.
- It does not "do one thing and do it well".
- The structure of the if statements are unclear, also hiding the logic and purpose of the lammp control.

1-53) (Used in:

Expectations:

1-54) (Used in: h2009)

Wrong use of semaphores may lead to deadlocks: What is a deadlock? Give an example on how this can happen.

Expectations:

"The system locked in cicular waiting" or "A state that the program cannot leave". We expect the standard wait(A);wait(B) / wait(B);wait(A) solution here. But other examples must also be accepted.

1-55) (Used in: v2013 , h2009)

Generally; List methods you can use to avoid that deadlocks becomes a problem in a software system.

Expectations: The following is the perfect answer I assume, but Burns and Wellings have reduced the contents of this chapter in the latest edition. - It is only 2-3 pages; Read throught it.

Long list expected here, both from deadlock prevention, avoidance, and detection and recovery. Alt fra godt design, til prioity ceiling & bonkers alg.

- Deadlock Prevention (Fjerne en av de 4 betingelsene)
  - Optimistic concurrency control (!)
  - Allokere alle ressurser samtidig.
  - Preemption: (timeout, priority...)
  - Global standard allokeringsrekkefølge.
  - Pluss denne: Global analyse - modellere & bevise fravær av deadlocks.
- Deadlock Avoidance:
  - Resource allocation (Bankers algorithm)
  - Scheduling algorithms (Priority Ceiling)
- Deadlock detection & Recovery:
  - Detection:
    - Analyse av hvem som eier og forespør hva
    - Timeout/Watchdog
  - Recovery:
    - Breaking mutual exclusion
    - Preemption (Ex. -> Forward Error Recovery)
    - Abort av Tråd eller Atomic Action (-> backward E.R.)

1-56) (Used in: h2009)

Acceptance tests is seen as an important tool for handling errors. How do acceptance tests differ from the more traditional tests on error conditions?

Expectations: De stiller krav til riktig tilstand, dvs. at de setter oss i posisjon til å oppdage uforutsette feil

1-57) (Used in: v2014 , v2013 , v2011kont , h2010)

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?

Expectations: We can detect "unexpected" errors. It also has an "merging of failure modes" effect.

1-58) (Used in: v2014)

Give examples of what one can test for when making acceptance tests.

Expectations: This is the "learn-by-heart" list from the book.

- Replication Checks
- Timing Checks
- Reversal Checks
- Coding Checks
- Reasonableness Checks
- Structural Checks
- Dynamic Reasonableness Checks

1-59) (Used in: v2014)

When an acceptance test fails, we are left with the knowledge that *something* is wrong, but we do not necessarily know exactly what. How can we recover i such a situation?

Expectations:

Backward or forward error recovery; just get to any consistent state. Merging of failure modes is also relevant.

1-60) (Used in: v2013)

What do we achieve by merging error modes?

Expectations:

We achieve a simpler system. Even though it would be possible to handle all different errors differently / do forward error recovery it is not always feasible since errors happen seldomly anyway.

Some errors might get more drastic consequences than strictly necessary, but again - it happens seldomly.

1-61) (Used in: h2009)

What is a systems error modes, and what do we achieve by merging error modes?

Expectations: We achieve a simpler system. Even though it would be possible to handle all errors / do forward error recovery it is not always feasible.

If something is said about error modes as breaking encapsulation it is good, but not necessary.

1-62) (Used in: v2011kont , h2009)

Optimistic Concurrency Control is a technique that can be used to avoid the overhead of locking. How does this work?

Expectations: It works by assuming, optimisticly, that conflicts will not happen, and detects and handles it as an error if it happens.

1-63) (Used in: h2010)

"Resource control" is more than mutual exclusion... What more can we expect from a system for resource control?

Expectations: Open answer here, but the 5 applications of Blooms criteria should be in (Different criteria for who gets the resource when more are waiting):

1. Different types of requests (Ex. r/w locks)
2. Order of requests (Ex. FIFO/LIFO)
3. Server State (Ex. history, state, mode)
4. Parameters of the request (Ex size, amount, importancy)
5. Priority of client

1-64) (Used in:

Expectations:

1-65) (Used in: v2011kont , h2009)

Shared variable synchronization may be criticized for poor scalability. Give, shorly, arguments for this.

Expectations: Synkroniseringsmekanismene har preg av globale variable. Å sjekke om det virker blir en global analyse.

Vi kan ikke se på en kodesnutt om den virker, uten å kjenne all bruk og interaksjon med synkronisering i hele programmet.

Andre gode argumenter må også belønnes.

1-66) (Used in: h2010)

Designing parallel systems with message-sending is an alternative to basing the design on synchronization. Give shortly and by bullet points benefits and drawbacks of these two applied to real-time systems design.

Expectations: Message based systems are more maintainable, scale better, makes better encapsulated modules. Not as well understood in RT context. Difficult to argue schedulability. Requires more infrastructure, and puts demands on thread model. Better abstraction for most situations. Design will be very different - we leave one thread per rt demand.

synch: scale badly, difficult to get right. Intuitive low level primitives.

Our goal is to make a barrier/rendezvous between three threads by using semaphores. That is, all threads should block at the synchronization point until all three are ready to proceed.

1-67) (Used in: h2009)

```
1 if(getWaiting(A) == 2){ // GetWaiting returns the number of
2                         // processes waiting for the semaphore
3   // The two others are here already - release and continue.
4   signal(A);
5   signal(A);
6 }else{
7   // The number of waiting threads is less than two - must wait.
8   wait(A)
9 }
```

This code has a race condition. What is a race condition? What is the problem with the code?

Expectations: If a thread is interrupted just after the call to getWaiting have returned a number less than 2, but before the thread waits, the program will deadlock if the other threads arrives.

1-68) (Used in: h2009)

```
1  count++;
2  if(count==3){
3    // All here, signal the others, reset and continue
4    signal(A);
5    signal(A);
6    count = 0;
7  }else{
8    // Not everyone have arrived, must wait
9    wait(A);
10 }
```

This code has at least one problem: Which?

Expectations: If one thread gets interrupted after the count increment, then the last thread arriving might reset count before it gets to test on it.

There is a race condition connected to the *if* that will lead to more threads executing the true-part rather than the false-part.

These two are the most important bugs to find.

Also there is a potential problem with the count variable itself if count++ is not atomic.

Reusability of the barrier is also an issue; is the barrier properly prepared for the next time the barrier is used? If a thread tries to enter again before the others have left?

1-69) (Used in: h2009)

(implementation/pseudo-code,3x) Make a barrier that works.

Expectations: What about something so simple (Lots of credit to the one who make it!):

```
1 // Signal that I am ready to both the other two
2 signal(A);
3 signal(A);
4 // Then wait for the others.
5 wait(B);
6 wait(C);
```

There are more general solutions in "The little book of semaphores".

1-70) (Used in: v2011kont , h2009)

(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.

Expectations: The standard solution is usable here also:

```
synch. allocate(){
  if(busy){
    queue.insertFirst(myThreadId);
    while(busy || queue.getFirst() != myThreadId) wait();
    queue.removeFirst();
  }
  busy = true;
  // notifyAll()?
}

synch free(){
```

```
  busy = false;
  notifyAll();
}
```

1-71) (Used in: v2013)

We imagine a storage with limited size, where elements can be entered and taken out. Two operations should be implemented (pseudo-code java w. $wait()$, $notifyall()$), $void\ put(Element\ e)$ and $Element\ get()$. Fill out the functions given under.

The handling of the elements, or the strategy for which element should be retrieved by $get()$ is not important: You only need to show the synchronization: *put* should block when the storage is full, and *get* should block if it is empty.

```
1  public class FiniteCapacityStorage {
2
3      private int current_storage_size;
4      final int   storage_capacity = 50;
5      // Make assumptions here as necessary.
6
7      public  FiniteCapacityStorage()
8      {
9          this.current_storage_size = 0;
10     }
11
12     public synchronized void put(Element new_item)
13     {
14         // You do the rest!
15     }
16
17     public synchronized Element get()
18     {
19         Element dummy = 0;
20
21         // You do the rest!
22
23         return dummy;
24     }
25 }
```

Expectations: Large variation can be expected here... What about some structures like

```
1  put(){
2    while(current_storage_size==storage_capacity) wait();
3    //Enter new_element into buffer
4    current_storage_size++;
5  }
```

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.

1-72) (Used in: v2013 , v2011kont , h2009)

(2x) Describe shortly how these three works.

Expectations: Det finnes selvfølgelig flere detaljer, men det følgende er ok:

POSIX: I wait-kallet på en condition variabel kan en angi en mutex som skal frigis midlertidig. - Ikke noe kobling til modulbegrep ut over hvordan en velger å bruke mutextene.

Java: Alle objekter kan beskytte metoder mot samtidig aksess fra flere tråder ved å merke metodene som synchronized. Fra innsiden av en synchronized metode kan en kalle wait for å suspensere seg selv, og frigi objekt-låsen midlertidig. notify vekker en slik suspendert prosess for objektet, mens notifyAll vekker alle.

Ada: Her har vi funksjoner, prosedyrer og entries som aksesserer et protected object. Blokkering fra innsiden av objektet er ulovlig (noe som rydder godt opp i dette med composition - det blir ikke mulig). Funksjoner kan bare lese interne variable, og kan dermed kjøre mange i slengen. Prosedyrer og entries krever eksklusiv aksess, og entriene kan være guardede - guardene kan bare teste på objektets private variable. Nevne requeue kanskje?

1-73) (Used in: v2013 , v2011kont , h2009)

Compare the Java and Ada mechanisms here, give the main strengths and weaknesses.

Expectations: Blooms criteria, expressive power and ease of use, bør komme med her som standardmåten å sammenligne slikt på (men de trenger ikke listes og gås systematisk igjennom). Adas svakhet at vi ikke kan ha guarder på parametre bør komme frem som et minimum. Det at guarder kommer heldig ut på ease-of-use-siden er flott om det nevnes.

1-74) (Used in: v2013 , h2010)

Mention some hardware (/assembly) mechanisms that are used for achieving basic synchronization.

Expectations:

Disable interrupt, test and set/ swap, spin locks.

1-75) (Used in: v2015 , v2013 , v2011kont , h2010)

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

Expectations: Basicly code will end up with race conditions; depending on what happens first - one thread suspending itself or the other one resuming it. Getting around this is very difficult if not impossible.

Testing on conditions for suspending itself does not work since we may be interrupted after the execution of the test and the suspend call.

1-76) (Used in: v2013 , h2010)

Javas *wait* and *notify* (and POSIX condition variables) works in many ways similar to *suspend* and *resume*. How can these be seen as better or more high-level synchronization mechanisms?

Expectations: Because they are assumed called from inside a monitor (making them uninterruptable, and making testing on conditions safe), and releases the monitor when blocking.

1-77) (Used in: v2013 , v2011kont , h2010)

"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?

Expectations: The difference is on where the execution continues after the signal/exception/notification. If the execution continues where it left off nothing is gained - the program must poll to check wether it has been interrupted.

1-78) (Used in: kont2015 , v2013 , v2011kont , h2010 , h2009)

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?

Expectations:

If there are more different reasons for waiting, you cannot easily know who you awaken with a notify. If you awaken somebody waiting for something else than what got ready, the system may misbehave. Even in a system with only one reason for waiting, this may change due to future reuse, maintenance or inheritance.

From the module perspective using only notify may demand knowledge of the usage patterns of the object. That is, yielding not so good encapsulation.

1-79) (Used in: v2013kont)

In a real-time program we often have more concurrently running, cooperating threads. Why is this a reasonable way of organizing the functionality of a real-time system?

Expectations: Keeping deadlines is difficult in a coupled system. Dividing the system into one thread per deadline is a simplification.

1-80) (Used in: v2013kont)

That the threads are cooperating means that they will have some need for interaction. This interaction usually takes the form of either "synchronization" or "communication". Explain these two shortly and/or give examples.

Expectations: Synchronization: access to a common global variable is restricted by mutual exclusion with e.g. semaphores.

Communication: One thread sends a message to the other.

1-81) (Used in: v2013kont)

Compare shortly and in bullet points how well suited these two are for a real-time system.

Expectations: Though message passing systems are well suited for large concurrent systems - scales well - easy to modularize as clients and servers, fantastic way of dividing a system into modules etc...

... message passing systems are not so well developed for real-time applications - no/-few schedulability proofs and inviting a larger number of (cooperating) threads which again makes scheduling & scheduling predicability harder.

Maturity questions again - any well-funded reasonable answer may receive score.

1-82) (Used in: v2013 , h2010)

Error detection — to decide whether something is wrong – can be done in more ways than checking for error returns. List principles that can be used to detect errors.

Expectations: This is the "learn-by-heart" list from the book.

- Replication Checks
- Timing Checks
- Reveral Checks
- Coding Checks
- Reasonableness Checks
- Structural Checks
- Dynamic Reasonableness Checks

1-83) (Used in: v2013)

In an atomic action we wish to have control over the participants and the duration of the action. The necessary synchronization that is necessary for this can be given a reusable form and put into an ActionController object.

In the code below is the entry protocol, that keeps track of the participants, and ensures that no role has more than one participant at the same time, already implemented.

You shall implement (java pseudo-code w. $wait()$, $notify()$ and $notifyAll()$) the missing part of the exit-protocol, that ensures that both participants exits the action at the same time.

```
public class ActionController {

    Boolean firstHere, secondHere;
    int allDone; // Better name: doneCounter ?
    final int numberOfParticipants = 2;


    public ActionController()
    {

        firstHere = secondHere = false;
        allDone = 0;

    }

    public synchronized void entryProtocolOne()
    {
        while(firstHere) wait();
        firstHere = true;
    }
    public synchronized void entryProtocolTwo()
    {
        while(secondHere) wait();
        secondHere = true;
    }

    synchronized void exitProtocolBoth()
    {
        // You do the rest!
    }

}
```

Expectations:

Tatt rett fra boken: Noen vil kanskje legge til noe tofase commit ting her - dette er rimelig, men det blir egentlig ikke spurt om...

```
  allDone++;
```

```
2   if(allDone == numberOfParticipants){
3     notifyAll();
4   }else{
5     while(allDone == numberOfParticipants) wait();
6   }
7
8   // Here comes code to prepare for the next action...
9   toExit--;
10  if(toExit == 0){
11    initialize all state and notify any threads that hangs on the entry
          protocol.
12  }
```

1-84) (Used in: v2013)

What do we achieve (in the domain of error handling) by using static redundancy?

Expectations: Error masking (of all errors as long as the replicas are independent!) and detection.

Given modules with failure probabilities, we can throw replicas at the problem until the probabilities are acceptable.

1-85) (Used in: v2013)

What do we achieve (in the domain of error handling) by using dynamic redundancy?

Expectations:

(Acceptance tests must take care of error detection...) We increase the probability of being able to successfully recover from any errors, and get a flexible framework for how this is done — If the first way of doing something did not work, maybe we can do it in another way. (Or; if the first actor could not do it, maybe the backup actor like in process pairs e.g.)

1-86) (Used in: h2009)

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?

Expectations:

- The "giving up to preserve consistency" error mode is often not acceptable since we have interactions with other systems that may be dependent on us to behave correctly.
- We have no time to waste; when we report back to the module that initiated the failed operation it might be too late to retry/fix it.

1-87) (Used in: h2009)

What is forward error recovery?

Expectations: We try to compensate and/or correct the error we have just detected.

Both forward and backward error recovery can be generalized from single thread to multi thread systems.

1-88) (Used in: h2009)

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

Expectations: Everything that looks like the "domino effect figure" is great for the first part of the question. Coordinating recovery points is the solution to the second.

1-89) (Used in: kont2014)

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

Expectations: Everything that looks like the "domino effect figure" is great for the first part of the question. Coordinating recovery points is the solution to the second.

1-90) (Used in: h2009)

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.

Expectations: Avstemmingen i etterkant av en AA. En kan polle feilstatus variable, eller hvis systemet er meldingsbasert kan en sende feilmeldinger. Ellers er asynchronous transfer of control det som er mest behandlet i boken: select then abort i Ada, AsynchrouneslyIterruptedExceptions i Java, og setjump/longjump-trikset eller pthread_cancel i C/POSIX.

1-91) (Used in: h2009)

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

Expectations: The introduction to the log was motivated by getting up after a restart to a consistent state. Here, however, we ask for the other usage for the log: To also write before-state into log records, and UNDO log records. A recovery point then becomes just a position in the log.

It is reasonable (but not expected) also to mix AA/transactions into this answer, since that infrastructure adds much to the log functionality.

1-92) (Used in: kont2014)

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

Expectations: The introduction to the log was motivated by getting up after a restart to a consistent state. Here, however, we ask for the other usage for the log: To also write before-state into log records, and UNDO log records back to the wanted state in time. A recovery point then becomes just a position in the log.

It is reasonable (but not expected) also to mix AA/transactions into this answer, since that infrastructure adds much to the log functionality.

1-93) (Used in: h2009)

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)?

Expectations: Ikke verre enn å operere med "TransaksjonsId'er" som knytter log-recordene til den enkelte oppgave. Recovery-punktene i logen kan da settes per-oppgave.

1-94) (Used in: v2014 , v2013 , h2009)

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

Expectations: More other good answers exist for the first part here. Pointing out the three boundaries (side, start and end), possibly with standard mechanisms to achieve them (locking, explicit membership, and two-phase commit protocol) is reasonable enough.

The problem to be solved: If more participants cooperates on something, they must possibly also cooperate in handling errors. AA provides the framework for achieving this, containing the errors and avoiding the domino effect.

"A mechanism/infrastructure for error containment when we have cooperating threads." basicly covers both questions quite well.

1-95) (Used in: v2014)

Describe one way of implementing an Atomic Action.

Expectations: The clear and systematic approach here is to tell how the tree boundaries are made:

Start boundary can be implemented by the ActionController keeping track of ActionMembers through some "entry protocol".

Side boundary can be implemented by some kind of locking of resources. Or limitations on communication: Participants can not communicate with nonpartisipants.

End boundary can be implemented by tho-phase commit protocol (or another kind of barrier)

1-96) (Used in: v2014)

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

Expectations: Locking is often an integral part of the infrastructure allowing error handling (like in an AA). We would like to avoid that the lock manager needs to get involved in error handling together with the action participants. (this would increase the complexity of the error handling, and possibly demand knowledge in the lock manager of the Action.)

1-97) (Used in: h2010)

What is the benefits by structuring the functionality of your program into Atomic Actions? Any drawbacks?

Expectations:

Good: Error handling framework generalizeable to more participants! Errors do not have consequences beyond the AA.

Bad: Bad composition, some infrastructure needed, one-design-fits-all ? Handles RT systems and side effects not that well.

1-98) (Used in: v2015 , h2009)

Give a short description of how process pairs work.

Expectations: Figure is good showing the master and backup processes sending IAmAlive, status and IAmMaster messages.

That the backup takes over when the master dies should be clear, and it is very good if the consistency of service is argued.

1-99) (Used in: v2011kont , h2010)

Process pairs is a technique for achieving available (in addition to reliable) processes. Which of the following points are the central building blocks/principles for implementing process pairs?

1 An "I Am Alive" protocol that lets the backup process detect whether the primary process is alive and working.

2 A (two-phase) commit protocol between primary and backup processes ensuring that they have a consistent view of the current state.

3 Sending state updates from the primary to the backup so that the backup will start working in the current state if/when it becomes primary.

4 Highly reliable and available messages must ensure that all IamAlive and State Update messages reach their destination and in the correct order.

5 Some mechanism for letting clients relate to the process that currently is primary (like broadcasting "IamPrimary" also to clients).

6 Process independent message queues that ensures that messages are not lost even though a process restarts.

Expectations:

1,3,5.

"Standard" error handling is to test for error situations and then make code that handles the detected errors. The fault tolerance part of this course is however motivated by the fact that this is not good enough:

1. A real-time or embedded system often have higher demands to reliability; We must handle also unexpected errors. (The errors that you did not think of when making the program and the bugs that you are unaware of.)
2. We also often have more cooperating threads in a real-time software system. Sometimes these threads must cooperate also on error handling.

1-100) (Used in: v2015 , v2011)

How can we detect these unexpected errors ? (explain shortly.)

Expectations: Acceptance tests: Give demands to the correct state/result rather than testing on error situations.

Static redundancy: Given intermittent errors or independent systems this catches errors that would impossible in other ways.

1-101) (Used in: v2015 , v2011)

Given that we have detected such an unexpected error... How can we know what we must do to handle the error ?

Expectations: Difficulty is that we did not know the cause... But it is often solvable by merging error modes: "I failed, no matter the reason I now must do..."

A reference to AA or transactions should be included: This lets us reason on and put limits on the possible consequences of the error.

Recovery points / backward error recovery is a catch-all: Lets us go back to known consistent state (and possibly try again).

1-102) (Used in: v2015 , v2011kont , v2011)

The failure modes is the ways a system can fail. To "merge failure modes" is a technique: What do we gain by doing this ?

Expectations:

- Simplification of the system. (If handling the worstcase error anyway, maybe all other errors can be handled the same way)
- Error modes is part of module interface: Fewer error modes enhances modularity / maintenance / composition by reducing size of interface.
- Handling unexpected errors, since merging of failure modes also can encompass unknown error modes...

1-103) (Used in: kont2014 , v2013kont)

Explain the terms "backward error recovery" and "recovery points".

Expectations: Backward error recovery: If an error is detected we go back to a previous, known-to-be-consistent, state.

Recovery point: One of these known-to-be-consistent, states. Typicly a complete snapshot of the programs state.

1-104) (Used in: v2013kont)

Explain the domino effect.

Expectations: If there are more interacting participants/threads, if the recoverypoints we aim to go back to are not sychronized/consistent *with each other* we may have to roll back to the beginning of program execution.

There is a standard figure illustrating this well...

1-105) (Used in: v2013kont)

How can the domino effect be avoided?

Expectations: Koordinate the making of the recovery points, typically when entering an atomic action.

1-106) (Used in: v2013kont)

If the program state is big or we have more concurrent ongoing operations that can fail individually, then *log* might be a good alternative. How does this work in context of backwards error recovery?

Expectations: Just before any part of program state is changed, the before-state of that part of the program state is stored away as a log-record. These are also typically labeled by the operation they are a part of so that the operations (the atomic action...) can fail individually.

When an error is detected the log records are "undone" in the opposite order they was created.

1-107) (Used in: v2013kont)

Another purpose of the log could be to enable program restart, so that a restarted program can get initialized to a current, consistent state e.g. after a crash. How?

Expectations: By, just before any side effect or program state change, the *intended effect/new value* is stored in the log - typically as part of the same log record as the old value.

At restart all (relevant) logrecords are "executed" in order.

1-108) (Used in: v2013kont)

We can imagine that after some time such a program can generate huge amounts of log. "Checkpoints" is a design pattern that lets us delete old log. How does this work?

Expectations: Into the log, at suitable intervals, we write a (non-consistent since there may be active operations going on) complete snapshot of the program state along with a list of active operations.

Log that is older than the last checkpoint which lists no operations that is still currently active can be deleted.

Recovery will then start by intitalizing to this checkpoint, and then executing the relevant logrecords from then on.

1-109) (Used in: kont2014)

We can imagine that after some time such a program can generate huge amounts of log. “Checkpoints” is a design pattern that lets us delete old log. How does this work? (Be aware that the word “Checkpoint” is used for different things in different parts of the curriculum.)

Expectations: Into the log, at suitable intervals, we write a (non-consistent since there may be active operations going on) complete snapshot of the program state along with a list of active operations.

Log that is older than the last checkpoint which lists no operations that is still currently active can be deleted.

Recovery will then start by intitalizing to this checkpoint, and then executing the relevant logrecords from this point.

An Atomic Action has start, side and end boundaries.

1-110) (Used in: v2015 , v2011kont)

What is the purpose of these boundaries ?

Expectations: To make clear limitations to what consequences an (unexpected) error can have, so that error handling gets possible.

Start: To establish which participants may be affected, and to set a safe, consistent, starting point (if an error have occured, it must have happened after the start point.)

Side: Limiting communication (restricting messages, locking variables...) to members to hinder error spreading out of the Action.

End: Ensuring a consistent system before leaving the action so that any errors did not spread / have consequenses after end of Action.

1-111) (Used in: v2015 , v2011kont)

How can each of them (start, side and end) be realized?

Expectations:

Start: In static systems this may be hardcoded. If not, some kind of explicit membership list is ok, A action manager can keep track of the members of each action. Any recovery points may also be established at start boundary, if preparing for backward recovery.

Side: Typically some kind of resource locking of resource to action. From transactions we learn that the transaction id should be part of all communication, meaning that all threads wanting to act on a message, must join the transaction.

End: Acceptance tests, and any vote or synchronization; Twophase commit protocol.

I guess other good answers may exist.

Asynchronous notification - to interrupt threads in what they are doing to communicate something, is discussed in relation to atomic actions.

1-112) (Used in: v2011kont)

Why ? Why is asynchronous notification discussed in relation to atomic actions?

Expectations:

When one participant detects an error *something* must be done... For those errors that may have spread to more participants already (which is all by common assumption: ref. merging of failure modes) the other threads must be made aware of the error. The (better? faster?) alterntive to polling or waiting for the prepareToCommit is to immediately notfy - asynchronously.

1-113) (Used in: v2014)

What is "Asynchronous Transfer of Control"?

Expectations: ...that one thread can interrupt, with **termination mode** (so that it does not continue where it left off after the interruption) another thread.

1-114) (Used in: v2014)

Why is Asynchronous Transfer of Control seen as relevant for use in Atomic Actions?

Expectations: Because for example when an error is detected by one thread, it might mean that the work done by another thread is not meaningful anymore -> It needs to be interrupted.

1-115) (Used in: v2011kont)

How can we get «termination mode» asynchronous transfer of control in POSIX/C? What about Java(/RT Java)? And ADA?

Expectations:

C: Canceling of threads or the setjmp/longjmp trick...

RT Java: AsynchronouslyInterrupted Exception (Java: canceling of threads.)

Ada: select-then-abort.

Gray had the thought that we could build fault tolerant computer systems if we only had reliable data storage, reliable communication and reliable calculations...

1-116) (Used in: v2011)

Refer shortly how we can achieve reliable storage (that is, built on unreliable storage)

Expectations: Redundancy is the keyword, but for a full score *some* more details should be mentioned: refresh-thread, writeback on error, something on the error modes, ...

1-117) (Used in: v2011)

Refer shortly how we correspondingly can achieve reliable communication.

Expectations: Redundancy is the keyword also here in the form of ack/timeout/resending. Also here some more details should be included: Sessions, checksums, sequence numbers, ...

1-118) (Used in: v2011)

Reliable calculations are the most difficult in this triplet since the failure modes is difficult to describe in the general case. The curriculum sets up "Checkpoint/Restart" as the first alternative: Describe how this works. What is the error modes, and how are they detected?

Expectations: Error mode: not delivering the next correct side effect. Detected by acceptance tests.

Just before any side effect is done, a set of acceptance tests are run: If ok, the complete state of the program is written to safe storage - the checkpoint. If tests are not ok, the program is restarted, and loads the previous checkpoint, continuing from there.

Gray had the thought that we could build fault tolerant computer systems if we only had reliable data storage, reliable communication and reliable calculations...

1-119) (Used in: kont2014)

Reliable calculations are the most difficult in this triplet since the failure modes is difficult to describe in the general case. The curriculum sets up "Checkpoint/Restart" as the first alternative: Describe how this works.

Expectations: Just before any side effect a set of acceptance tests are run, if ok the complete state of the program is written to safe storage - the checkpoint. If tests are not ok, the program is restarted, and loads the previous checkpoint, continuing from there.

1-120) (Used in: kont2014)

A simple and straightforward alternative to checkpoints like this is to write to the "log". How does this work, and what do we achieve compared to the checkpoints?

Expectations: Rather then writing the complete state to the storage, only the changed part of the state is written: As each part of the state (i.e. variable) is changed, the new value of the variable is written.

When doing the recovery, the latest state is reconstructed by executing all the log-records.

This lets us avoid writing the complete state every time - which may be very big. (Many more things to mention here...)

1-121) (Used in: kont2014)

What is, in this context the error mode(s) of a "calculation", and how are they detected?

Expectations: Error mode: not delivering the next correct side effect. Detected by acceptance tests.

Gray had the thought that we could build fault tolerant computer systems if we only had reliable data storage, reliable communication and reliable calculations...

1-122) (Used in:

Reliable calculations are the most difficult in this triplet since the failure modes is difficult to describe in the general case. The curriculum sets up "Checkpoint/Restart" as the first alternative: Describe how this works. What is the error modes, and how are they detected?

Expectations: Error mode: not delivering the next correct side effect. Detected by acceptance tests.

Just before any side effect a set of acceptance tests are run, if ok the complete state of the program is written to safe storage - the checkpoint. If tests are not ok, the program is restarted, and loads the previous checkpoint, continuing from there.

1-123) (Used in: kont2014 , v2011)

(2x) Another way to achieve reliable calculations is process pairs. Give a short explanation on how this works. What do we gain here compared to checkpoint/restart?

Expectations: We get availability here - checkpoint/restart is dependent on the restart and any repair.

Two programs, the primary and the backup are at the same time. The primary does the side effects and sends the checkpoints to the backup along with IAmAlive messa-

ges. The backup broadcasts IAmMaster (both to the clients any other "servers") when enough IAmAlive messages have been missed - and continues from the last checkpoint, probably starting with the last prepared side effect/reply.

1-124) (Used in: kont2014)

"Transactions" are almost the same as "Atomic Actions" in the Burns&Wellings book, and are relevant when making fault tolerant computer systems. How do transactions contribute to fault tolerance.

Expectations:

The perfect answer here would be the ACID properties of a transaction: Atomicity (it happens all-or-nothing), Consistency (a transformation from one consistent/correct state to another), Isolation (partial results are not propagated out of the transaction before it has finished so that errors can be handled) and Durability (the result of a calculation is never lost), but I am willing to accept a full score if properties corresponding to A and I are mentioned.

Good answers can also be formulated describing the corresponding properties of AA.

1-125) (Used in: kont2014)

What is the difference between Tansactions and Atomic Actions? How does this difference make Atomic Actions more attractive to use in a real-time system?

Expectations: Transactions have one and only one error mode: ABORT. While an AA is more of a transactional framework where both forward and backward error recovery is possible. ABORT might not be an option when you have a deadline, since we have wasted some of our time already.

1-126) (Used in: kont2014)

Grays third alternative way to make reliable calculations assumes that all data (the whole program state) is safely stored in a database, that all calculations are formulated as transactions, and that we have a system for running these transactions which also, automaticly handles any errors.

In addition to the answer in the previous question; Why do we not make all (embedded, real-time) systems like this?

Expectations: Large, heavy, less flexible than we need in an embedded setting.

...and it does not exist, generally.

1-127) (Used in: v2013kont)

What do we achieve by using process pairs?

Expectations: Fault tolerance by (dynamic) redundancy. High availability since the switching between replicas happens immediately.

1-128) (Used in: v2013kont)

Explain shortly how process pairs work.

Expectations: Two programs, the primary and the backup are run at the same time. The primary does the side effects (like "send answer to the client") and sends the program state/checkpoints to the backup (though in the opposite order!) along with IAmAlive messages. The backup broadcasts IAmMaster when enough IAmAlive messages have been missed - and continues from the last checkpoint.

A number of techniques/design patterns has been set up to handle a number of challenges in error handling and consistency in multi thread systems:

- Writing log records to the log
- Writing Checkpoints to the log
- The two-phase commit protocol
- Two-phase execution (growing and shrinking phases)
- Introducing a transaction manager
- (Read/Write-) Locking of accessed resources.
- Introducing a LogManager
- Introducing a LockManager

For each of the tasks under, you shall pick the most central technique to achieve what the task asks for.

1-129) (Used in:

For the application you are writing it is critical that you keep track of what you have done, so that you may start in the current state even after a restart.

Expectations: This is the log.

1-130) (Used in:

After having programmed 3 different "atomic actions" you observe that you have code for coordinating start and end boundaries 3 different places in the code. You want to collect reusable code into a module.

Expectations: This is the transaction manager.

1-131) (Used in:

You have decided that this growing and shrinking phase concept is a nice way to keep track of the possible consequences of errors. You need however to coordinate, between the participating threads, the point in time where the growing phase ends and the shrinking phase begins.

Expectations: This is the Two-phase commit protocol.

1-132) (Used in: v2011kont , h2009)

Use of exceptions in a language that supports it (like Java) simplifies some things. List the benefits of using exceptions / What gets better or simpler by using exceptions?

Expectations:

- Bubbling the error up to the right handler is taken well care of.
- We get proper division between “normal execution” code and “error handling code”.
- No performance overhead except when there are errors.
- ...

1-133) (Used in: v2011kont , h2009)

Are there any disadvantages by using exceptions in a program, real-time system or programming language. (Answer briefly).

Expectations:

- Error handling gets very expensive.
- Must get final wishes right. These precautions is dependent on other parts of the code - breaking encapsulation.
- Does it make the code easier to maintain? - “a goto where you do not know where you came from and where you are going”? - And the mechanism is not allways simple to understand itself.
- ...

1-134) (Used in: v2011kont , h2010)

In a system where exception handling is used, the following code may cause a deadlock:

```
{
  Wait(semaphoreA);
  f();
  Signal(semaphoreA);
}
```

Explain what the problem is and how it is solved.

Expectations:

f may throw an exception, causing the semaphore never to be signaled.

We solve this with "final wishes", a piece of code that will allways be executed - even when an exception is thrown.

1-135) (Used in: v2013kont)

A number of modern programming languages comes with built-in mechanisms for exception handling. What are we hoping to achieve with these mechanisms?

Expectations:

- Easy-to-understand mechanism to make errorhandling simple.
- Separation of error handling code and normal-operation code.
- Same mechanism for handling different types of exceptions.
- No overhead in normal operation?
- Allow recovery actions :-)

1-136) (Used in: v2013kont)

Explain shortly how exception handling works. Use C++, Java or ADA as example language if you want.

Expectations: Basicly: At error a program can "throw" an exception. Up in the call hierarchy a catch-clause may trig, handle the error and the program will(if it does not rethrow) continue operation after the catch-clause.

1-137) (Used in: v2013kont)

Which downsides exist to using exception handling in a program/real-time system.

Expectations:

- Partly difficult and non-intuitive semantics.

- invisible program flow paths.
- goto without knowing where you came from or where you are going
- Large and unpredictable overhead at error.

We have used the following demands on a principle for dividing a system into modules:

1. We must be able to use a module without knowing its internals.
2. We must be able to maintain a module without knowing its usage patterns.
3. Composition: Super-modules can be made out of sub-modules.

In this perspective: Criticize (shortly) the following principles for dividing a system into modules. Will they be suited for contructing a large software system ?

1-138) (Used in: v2011kont)

Dividing functionality into *Threads*, like in Java or POSIX.

Expectations: Shared variable synchronization lies in the context here. This is global mechanisms for synchronization that makes at least point 2 fall, probably also point 1.

Point 3 also falls: building superthreads from subthreads is not a good option.

1-139) (Used in: v2011kont)

Monitors

Expectations: Point 1 and 2 is ok I guess: A monitor is a module like any other.

We have problems with composition since blocking in an inner monitor does not release the outer monitor.

1-140) (Used in: v2011kont)

Communicating processes (as in OCCAM or Google Go - communication by messages, no shared resources).

Expectations: Since a process can control which messages to receive and which other processes to communicate with, we are in a very good position to achieve both 1 and 2.

Point 3 is also well achievable by naming a group of processes a superprocess, since this superprocess is defined by its communication links, just as the sub-processes.

1-141) (Used in: v2011kont)

Atomic Actions

Expectations: Composition lacks unfortunately, but the two first points is ok (if the actions are designed well) like with monitors.

We have the following three FSP processes:

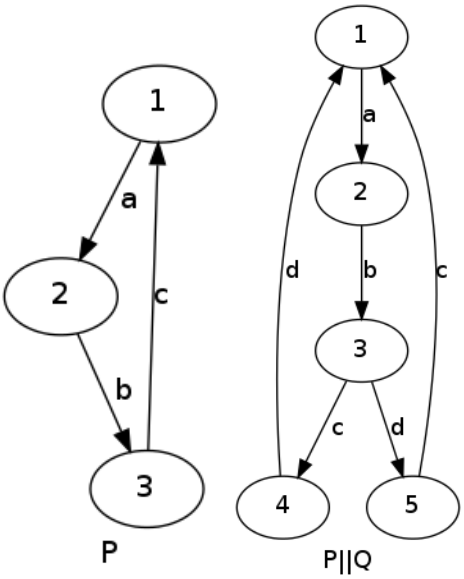
$$P = (a \to b \to c \to P).$$
$$Q = (a \to b \to d \to Q).$$
$$R = (d \to b \to c \to R).$$

A possible interpretation of an event that two processes participates in is that there is (synchronous) communication between them. If there are more than two participating processes, it may be a barrier/rendezvous.

1-142) (Used in: kont2015 , h2009)

Draw the transition diagram for P and $P||Q$.

Expectations:



1-143) (Used in: kont2015 , h2009)

$P||Q||R$ deadlocks. Why?/What happens? How can you recognize a deadlock in a transition diagram?

Expectations: *R* tries, before the barrier, to communicate with *Q* (event *d*), but *Q* will only communicate after the barrier. No need to draw transition diagram here.

1-144) (Used in: kont2015 , v2011kont , h2010 , h2009)

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

Expectations: A live-lock is a bug where a subset of states that does not fill the whole functionality of the system is entered, and where there is no way of leaving this subset.

A potential live-lock can be recognized in a transition diagram if not all states are reachable from all other states.

I guess a transition diagram illustrating such a subset is ok.

1-145) (Used in: kont2015 , h2009)

Model the following semaphore program in FSP.

```
T1(){
  while(1){
    wait(A);
    wait(M);
    ...
    signal(M);
    signal(A);
  }
}
```

```
T2(){
  while(1){
    wait(B);
    wait(M);
    ...
    signal(M);
    signal(B);
  }
}
```

Expectations:

P1 = (wa->wam->sam->sa->P1).

P2 = (wb->wbm->sbm->sb->P2).

A=(wa->sa->A).

B=(wa->sa->B).

M=(wam->sam->M | wbm->sbm->M).

A reasonable way of getting synchronous communication using semaphores can be like in the following program stubs, where thread 1 sends a sequence of integers to thread 2.

```
1
2 int ch; // The channel
3 Semaphore ch_sendReady(0),ch_receiveReady(0);
4
```

```
thread1(){
  int i=0;
  while(1){

    // send i
    ch = i;
    ch_sendReady.signal();
    ch_receiveReady.wait();
    i++;
  }
}

thread2(){
  int i;
  while(1){

    // receive i
    ch_sendReady.wait();
    i = ch;
    ch_receiveReady.signal();

  }
}
```

1-146) (Used in: v2013)

Make a model of the system in FSP (Finite State Process-notation)- look at the attached FSP reference if necessary.

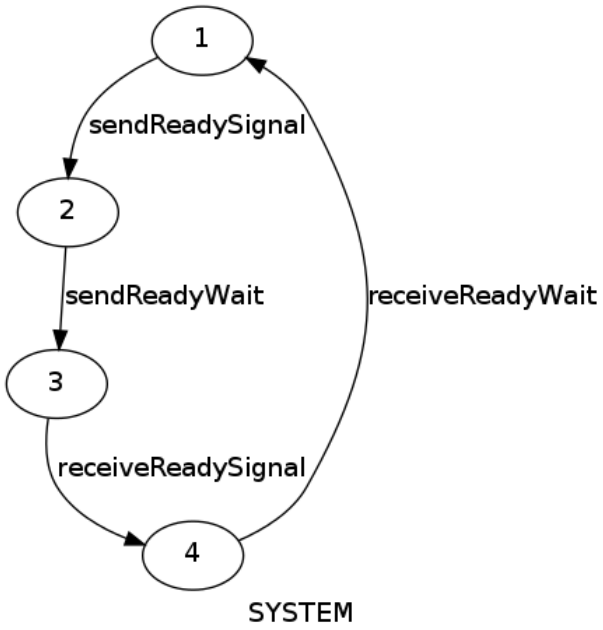
Expectations:

4 processes are necessary:

$$
\begin{aligned}
T1 &= (sendReadySignal \rightarrow receiveReadyWait \rightarrow T1).\\
T2 &= (sendReadyWait \rightarrow receiveReadySignal \rightarrow T2).\\
SendSem &= (sendReadySignal \rightarrow sendReadyWait \rightarrow SendSem).\\
ReceiveSem &= (receiveReadySignal \rightarrow receiveReadyWait \rightarrow ReceiveSem).\\
||SYSTEM &= (T1||T2||SendSem||ReceiveSem).
\end{aligned}
$$

1-147) (Used in: v2013)

Draw the transition diagram for the total system. How many states are there ?



Expectations: ...4 states.

If we imagine a language with synchronous communication built into the language, the corresponding code can look like this:

```
1  T1(int channel ch){
2    while(1){
3      int i;
4      ch <- i;
5      i++;
6    }
7  }
8
9  T2(int channel ch){
10   while(1){
11     int i;
12     i <- ch;
13     //...
14   }
15 }
```

1-148) (Used in: v2013)

Model this system in FSP. (Hint: Expect a quite simple model)

Expectations:

$$\begin{aligned} T1 &= (communicate \rightarrow T1). \\ T2 &= (communicate \rightarrow T2). \\ ||SYSTEM &= (T1||T2). \end{aligned}$$

1-149) (Used in: v2013)

Draw the transition diagram for the total system. How many states are there? (Hint: Fewer than last time)

Expectations: ...1 state.

1-150) (Used in: v2013)

We have in both examples two threads that passes information between them. Reflect shortly on why one model is simpler than the other.

Expectations: There is no correct answer here: Any well-founded or mature answers us given credit. Possible reflections can be:

- The two threads communicate, and passing messages is a better abstraction for this leading to a more precise and simple implementation.
- ...

If we imagine a larger system of threads that communicate by this two-semaphores-per-channel pattern, the number of semaphores in the system may become very large.

1-151) (Used in: v2013)

Give an example showing how deadlocks can occur in such a message passing system.

Expectations: Thread1 to tries to send something to Thread2 and vice versa - nobody listens.

1-152) (Used in: v2013)

And specifically for our message passing system: Of the points you mentioned in the previous question; Which do you see as most relevant here? (...or if you have new points to mention to ensure the absence of deadlocks...)

Expectations:

To avoid circular "dependencies" I would say was the correct answer here. Look at the communication arrows; use client server pattern, turn communication arrows by buffered "I need to communicate" signals a'la Øyvind Teigs guest lecture.

Going for buffered communication is a common solution and a reasonable answer (that we do not like so much...)

1-153) (Used in:

Expectations:

A classic example of a system with a deadlock is "dining philosophers", where N philosophers sits eating at a round table. Each need two forks to eat, and must share with both his neighbours the N forks distributed between the philosophers.

1-154) (Used in: kont2014 , v2013kont , v2011)

Show how the philosophers and the forks can be modeled in FSP.

Expectations: No fancy FSP is necessary:

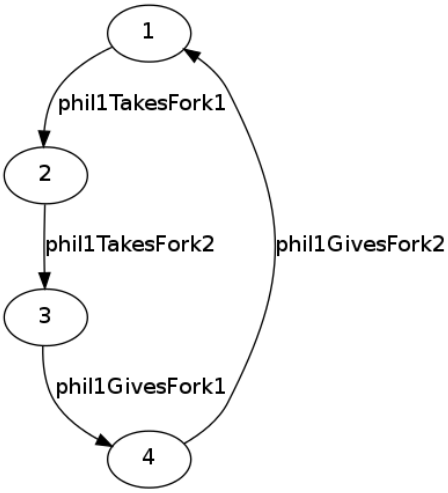
FORK1 = (phil1TakesFork1 -> phil1GivesFork1 -> FORK1 | phil2TakesFork1 -> phil2GivesFork1 -> FORK1).

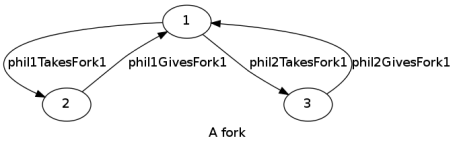
PHIL1 = (phil1TakesFork1 -> phil1TakesFork2 -> phil1Eat ->phil1GiveFork1 -> phil1GivesFork2 -> PHIL1).

1-155) (Used in: kont2014 , v2013kont , v2011)

Draw a transition diagram for a philosopher and for a fork.

Expectations:

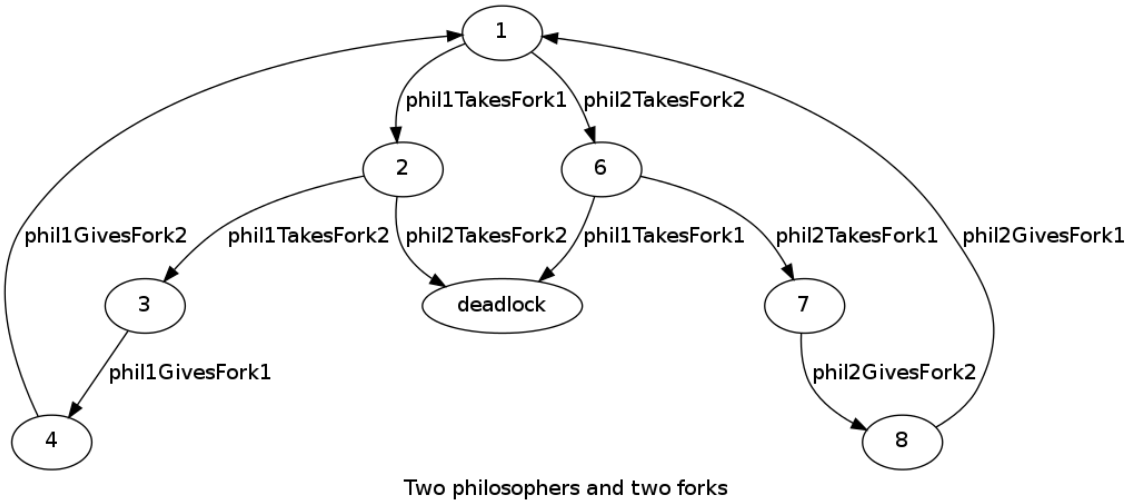




1-156) (Used in: kont2014 , v2013kont , v2011)

Draw a transition diagram for a compund system with two philosophers and two forks. Point out the deadlock.

Expectations: The philosophers releasing their forks in the other order would yield a slightly more complex image.



1-157) (Used in: h2010)

The book sets up four necessary conditions for deadlocks to occur. Which? And how can deadlocks be prevented by seeing to it that each of these are not met?

Expectations:

Necessary Conditions:

1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Circular Wait

Deadlock Prevention (removing one condition):

1. Optimistic concurrency control.
2. Allocate all resources at once.
3. Preemption.
4. Global allocation order.

1-158) (Used in: h2010)

In a traditional real-time system we often know exactly which threads that uses which resources. This opens up for some techniques that lets us avoid the problem of deadlocks. Which?

Expectations:

Priority Ceiling and Bankers algorithm, possibly also "Global allocation order" and "allocate all resources at once" and formal verification.

1-159) (Used in: h2010)

However, predicting which threads use which resources is not always possible; Imagine a system of mobile sensor vehicles sharing updates to via a base station, or a process control plant where the operators are allowed to program alarm scripts. In both these cases we may get unpredictable resource allocations.

How can we avoid or solve any problems with deadlocks in such systems?

Expectations:

```
Detection:
 - Who owns and asks for what (detect circles)
 - Timeout/Watchdog
Recovery:
```

```
- Breaking mutual exclusion
- Preemption
- Abort of thread or Atomic Action
```

1-160) (Used in: v2013kont , v2011)

What is a deadlock ? Give an example of a deadlock with semaphores.

Expectations: The classic example:

T1: wait(A); wait(B); dowork; signal(B); signal(A)

T2: wait(B); wait(A); dowork; signal(A); signal(B)

... but if a student insists that wait(A) is a deadlock it is :-)

1-161) (Used in: v2011)

Deadlocks can of course also happen in systems where interaction between threads is message based. Construct an example from this domain.

Expectations: T1: recv(T2); send(T2);

T2: recv(T1); send(T1);

Should really be trivial. - recv(T3);...

1-162) (Used in: scheduling , v2013kont)

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

Expectations: First we can schedule the most "critical" thread first like in EDF (or even rate-monotonic priority ordering), kind of increasing the chance that all deadlines will be met.

But predictability is as important paving the way for schedulability analysis.

1-163) (Used in: kont2015 , kont2014 , v2013kont)

(2x) 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 works.

Expectations: Utilization: we have a formula which guarantees schedulability for N threads if the threads have rate-monotonic priorities and the sum of utilizations are less than a given number (dependent only in N).

Response time: For each thread we can calculate its worst case response time from its max execution time and the number of times it can be interrupted by higher-priority

threads.

1-164) (Used in: kont2015 , v2013kont)

There are a number of assumptions/conditions that must be true for these tests to be usable. (“The simple task model”) Which?

Expectations:

- Fixed set of tasks
- Periodic tasks, known periods
- The threads must be independent.
- Overheads, switching times can be ignored
- Deadline == Period
- Fixed, known Worst Case Execution Time.
- and in addition: Rate-Monotonic Priority Ordering or deadline first.

1-165) (Used in: v2014)

There are a number of assumptions/conditions that must be true for these tests to be usable. (“The simple task model”) Which? Comment (shortly) on how realistic they are.

Expectations:

- Fixed set of tasks (No sporadic tasks... Not optimal but fair deal)
- Periodic tasks, known periods (Realistic)
- The threads must be independent. (Not realistic at all in an embedded system)
- Overheads, switching times can be ignored (Sometimes yes, sometimes no)
- Deadline == Period (Not optimal but Fair deal)
- Fixed Worst Case Execution Time. (Not realistic to know a tight estimate here.)
- and in addition: Rate-Monotonic Priority ordering. (our choice, so ok)

1-166) (Used in: kont2015)

The schedulability proofs probably have not been performed for most real-time systems out there in the world. Why do you think the industry is reluctant to perform these proofs?

Expectations: The assumptions does not hold. The execution time bounds are all too conservative. The SW is too complex to fit into the standard model. (System seems to work well enough after testing)

1-167) (Used in: v2013kont)

What is “Priority Inversion” ?

Expectations: That one task ends up waiting for a lower priority task. This may happen when they share a resource that the lowpriority task holds.

It is ok if the student also explains unbounded priority inversion, but these two should not be confused.

1-168) (Used in: v2013kont)

How does priority inversion influence the schedulability analyses?

Expectations: Invalidates the assumptions for the (basic form of) analyses to be valid.

Can partly be compensated for by more complicated analyses, and more conservative results in terms of utilization.

Difficult question so we can give credit for any mature answer.

1-169) (Used in: kont2014 , v2013)

Why is “scheduling” so important in real-time programming? — Which features are we hoping to gain in our system by choosing good scheduling strategies?

Expectations: Predicatbility in timing domain, so that systems can be analyzed. Simple analysis, not too conservative analysis - facilitating high utilization. Ensuring that all thread meed their deadlines is ok.

Comments on resource allocation (of other resource than CPU) and Deadlock avoidance are also good.

Performance and fairness is bad answers.

1-170) (Used in: kont2014 , v2013)

We imagine a classical real-time system with threads that synchronize using semaphores: How, typically, is it decided which thread gets to run (to achieve the goals in the previous question)?

Expectations: The threads are classified by priorities; The runnable thread with the highest priority gets to run.

1-171) (Used in: v2013)

In the same system: How, typically, is it decided which of more threads waiting on a semaphore, gets to run when the semaphore is signaled?

Expectations: Unfortunally priorities are the wrong answer here; First come, first serve is the normal behavior. This is to avoid starvation...

1-172) (Used in: kont2014)

In the same system: How, typically, is it decided which of more threads waiting on a semaphore, gets to run when the semaphore is signaled? Why?

Expectations: Unfortunally priorities are the wrong answer here; First come, first serve is the normal behavior.

This is to avoid starvation...

1-173) (Used in: v2013)

What is the "unbounded priority inversion" problem?

Expectations: That a high priority thread potentially might be waiting forever (unbounded time) for lower priority threads.

This happens when a lowpriority thread owns a resource that the high-priority thread needs. Unboundedness comes from the unknown number of intermediate priority threads potentially wanting to run.

Mention of the "intermediate-priority" threads is necessary for a good answer.

1-174) (Used in: v2013)

How can the "unbounded priority inversion" problem be solved by scheduling strategies?

Expectations: Priority Inheritance or Ceiling protocols. That the lowpriority thread get temporary increased priority - over the intermediate threads - in these situations.

Given these three tasks:

Task	Period	Execution time
A	80	20
B	60	20
C	40	10

1-175) (Used in: v2014 , v2013)

Show in a graphical way how task A, B and C in the table over will run under rate

monotonic priority assignment. Mark clearly if and where any deadlines are missed. (Assume that all tasks are ready for the first time at time zero)

Expectations:

A nice and correct figure gives 4; Beeing conscious on where along the time axis to stop drawing is perfect.

1-176) (Used in:

Expectations:

1-177) (Used in: v2014 , v2013)

Make an utilization test for this system.

$$\left(Hint : U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N(2^{1/N} - 1) \right)$$

Expectations:

The test fails: For full score it should be mentioned that the the test is indecisive on the schedulability. Concluding wrongly that the system is not schedulable should pull down a bit more.

1-178) (Used in: v2014 , v2013)

Make a response time analysis of this system.

$$\left(Hint : w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \right)$$

Expectations:

The test succeeds... (Not making a complete solution here...) The system is schedulable.

You are to make a real-time system, which means that you have real-time demands to the system, and a duty to be able to argue that all demands will be met. (That is, beyond the common “you have tested the system and it seems to work” strategy)

1-179) (Used in: v2015 , v2011)

How would you structure the system to be able to argue that the realtime demands will be met ? (In other words: Can you say something about your software design.)

Expectations: One thread per real-time demand. Set priorities according to deadlines. Use shared variable synchronisation and keep careful track of resource usage for the different threads so that you can comment on maximum blocking times.

1-180) (Used in: v2015 , v2011)

And the argumentation itself: Sketch along which lines you would argue that the real-time demands are met.

Expectations: Estimate worst case execution times for threads and blocking times, go through a schedulability proof. Complete detail of a schedulability proof is not necessary, but the answer should be convincing.

1-181) (Used in: v2015 , v2011)

Interaction between different parallel tasks may lead to "inversion" problems (like priority inversion). Explain what the problem is.

Expectations: If a task with high priority is dependent on a resource owned by a low priority process it will be blocked waiting for something that may not run for a long time given the low priority of the resource holder.

1-182) (Used in: v2015 , v2011)

Explain shortly how a Priority Ceiling protocol works. (That is, choose one of them.)

Expectations: Immediate: Allocating a priority value to shared resources in the system, and letting a thread allocating the resource get the resource's priority (or really: max(current,resource)) while it owns it. The resource priorities should be equal to the max of that of all threads using the resource.

Two wellknown scheduling strategies is FPS (Fixed Priority Scheduling) and EDF (Earliest Deadline First).

1-183) (Used in: v2011)

Explain the terms FPS and EDF.

Expectations: FPS: All tasks get a fixed priority, and the scheduler allways lets the runnable highest priority thread run.

EDF: No predetermined priorities are given; the scheduler allways runs the task with the earliest (absolute) deadline.

1-184) (Used in: v2011)

Why is FPS more commonly in use ?

Expectations: This is surprising since EDF comes out better in more schedulability tests...

The book lists:

- Easier to implement
- Easier to incorporate tasks without natural or hard deadlines.
- Though peroid (as used by FPS/RMS for setting priorities) is not really a good measure of importance, but it is far better than absolute deadlines. FPS priorities can be tweeked...
- The overload behavior of FPS may be preferred - see previous point.
- The FPS/RMS schedulability test is unnecessary pessimistic (SH: seriously? are we making systems that may or may not work?)

Do not demand the last point from the students.

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...T_N is like T_2.

1-185) (Used in: v2011kont , h2010)

Assume N=3. Model the system in FSP. (If you want to avoid the problem of terminating processes you can model "Continue Working" as $DW = (doWork \rightarrow DW)$.) See the attached FSP reference if necessary.

Expectations:

The perfect (LTSA-eatable) solution:

```
T1 = (t1.sa->DW1),
  DW1 = (dw1->DW1)+{t1.wa}.
T2 = (t2.wa->t2.sa->DW2),
  DW2 = (dw2->DW2).
T3 = (t3.wa->t3.sa->DW3),
  DW3 = (dw3->DW3).
A = (sa->wa->A).

||SYSTEM = (T1||T2||T3||{t1,t2,t3}::A).
```

This model is not as complicated as the "perfect solution" looks like though:

Minimum solution for full score:

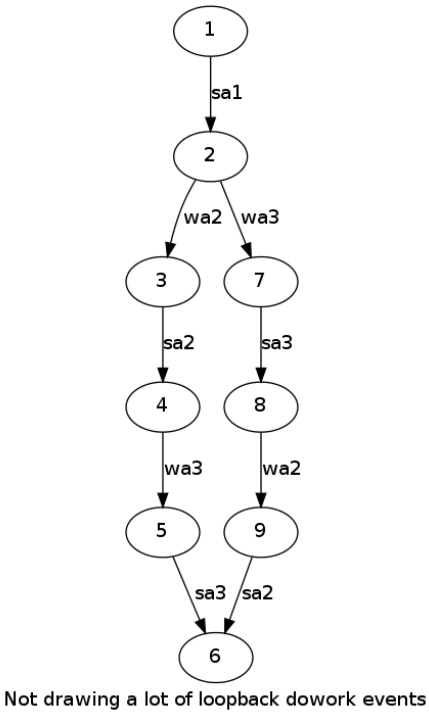
```
T1 = sa1 -> DW1.
T2 = wa2 -> sa2 -> DW2.
T3 = wa3 -> sa3 -> DW3.

A = (sa1 -> AFree|
     sa2 -> AFree|
     sa3 -> AFree),
AFree = (wa2 -> A|
         wa3 -> A).
```

Capturing the semaphore model, and keeping the processes' wa and sa events from each other should be expected though.

1-186) (Used in: v2011kont , h2010)

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

Expectations:



1-187) (Used in: v2011kont , h2010)

(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)?

Expectations:

```
class InitSync {
  private bool initialized = false;

  synchronized
  initialize(){
    // Called by T1
    initialized = true;
    notifyAll()
  }

  synchronized
  isInitialized(){
    // Called by T2..TN
    while(!initialized) wait();
  }
}
```

1-188) (Used in: v2011kont , h2010)

(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)?

Expectations:

```
protected body Init is
  initOk : Boolean := False;

  entry InitOk when initOk is
  begin
  end InitOk;

  procedure SetInitOk is
  begin
    initOk := True;
  end SetInitOk;

end Action_Controller;
```

Task 1 just calls SetInitOk while tasks 2...N enter InitOk entry.

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.)

1-189) (Used in: v2011kont , h2010)

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)?

Expectations:

Race Condition: A bug that surfaces by unfortunate timing or order of events.

1. The statement $g\_nOfThreadsInitialized++;$ is not atomic and not protected; The variable may never reach N.
2. Interruption just before the if statement may lead to more threads trying to reset the semaphore, thereby deadlocking.

You are to write a resource manager that administers a memory area of limited size. The call to $void*Allocate(int\ nOfBytes)$ takes the amount of memory to reserve as a parameter and returns a pointer to a memory area. $Free(void*)$ takes such a pointer to a memory area and releases the memory again.

1-190) (Used in: v2013kont , h2010)

(Implementation/pseudo-code) You are to use shared variable synchronization, but can use any mechanisms for this.

Focus is on the synchronization; Make suitable assumptions when it comes to the memory book keeping and any data structures you need.

Expectations: This is the plain standard Java solution:

```
synch. allocate(int size){
  if(!memory.canAllocate(size)){
```

```
    queue.insertSorted(myThreadId);
    while(!memory.canAllocate(size) || queue.getFirst() != myThreadId)
      wait();
    queue.removeFirst();
  }
  return memory.allocate(size)
}

synch free(p){
  memory.release(p);
  notifyAll();
}
```

1-191) (Used in: v2013kont , h2010)

What is starvation?

Expectations: One thread does not get the resources it needs due to unfortunate scheduling.

1-192) (Used in: v2013kont , h2010)

Comment on the danger for starvation regarding the memory allocation task over. (If you did not have an implementation that was free of possible starvation problems:) Suggest a strategy for memory allocation to waiting threads that does not have this problem.

Expectations: We have the choice here between either blocking threads "unnecessary" - they are asking for little enough memory that the request could have been fulfilled - or starving the big requests.

First come, first serve is a strategy that avoids this starvation (the price is blocking smaller requests unecessary...).

1-193) (Used in: v2013kont)

We imagine a classical real-time system with prioritized threads that synchronize using semaphores: How, typically, is it decided which of more threads waiting on a semaphore, gets to run when the semaphore is signaled?

Expectations: Unfortunally priorities are the wrong answer here; First come, first serve or "arbitrary" is the normal behavior. This is to avoid starvation...

1-194) (Used in: v2013kont , h2010)

Adas mechanisms (protected objects with functions, procedures and entries with guards)

are not that suited to solve this memory allocation problem... Why?

Expectations: Guards cannot test on parameters, leading to complicated code; double interactions, Entry families or use of requeue.

1-195) (Used in:

Expectations:

A server that responds to messages on alloc & free channels

```
memoryServer(chan alloc){
  while(1){
    select {
      alloc
    }
  }
}
```

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...

1-196) (Used in: v2011kont , v2011)

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

Expectations: The module is not reentrant; Any data structures (shared resources) used by the module may be accessed by more threads at the same time, or even worse; references to internal data-structures may be returned to the caller for concurrent access other places in the program.

1-197) (Used in: v2011)

Anyway, you decide to protect the module with semaphores by allocating (*wait*) a given binary semaphore at the start of each function, and releasing it (*signal*)just before returning. This does not work: What happens ?

Expectations: Since *getName* calls *getFirstName* and both allocates the semaphore we will deadlock.

1-198) (Used in: v2011)

How would you fix this problem? (You still have only semaphores available for synchronization.)

Expectations: Hmm. This is close to impossible to get correct if you start testing on semaphore status or thread ids. Any good ansers should of course be rewarded anyway... I would build a new interface to the module, still protected by semaphores, but avoiding dependencies between functions in the interface.

1-199) (Used in: v2011)

Describe shortly how Javas monitor-like mechanisms for synchronization works ("synchronized objects").

Expectations: *synchronized*, *wait*, *notify* and *notifyAll* should be explained.

1-200) (Used in: v2011)

(Java pseudo-code) You are making a corresponding module in Java: Sketch how the four functions could be protected from concurrent use by more threads here.

Expectations: This is trivial; 3 synchronized public functions in the class. The problem we had with semaphores is nonexistent here.

1-201) (Used in: v2011)

Describe shortly how Adas monitor-like mechanisms for synchronization works ("protected objects").

Expectations: The difference between functions, procedures and entries should be explained, also the possibilities to have guards on entries.

Requeue & entry families is too much, not necessary for full score :-)

1-202) (Used in: v2011)

(Ada Pseudo-code) You are making a corresponding module in Ada: Sketch how the four functions could be protected from concurrent use by more threads here. Is this solution in any way better than the Java solution ?

Expectations: The code does not have to look like Ada, but the point that 3 of these functions can be *functions*, while the set function must be a procedure, thereby getting implicit readlocks is clear in the perfect answer.

Otherwise it should be trivial.

1-203) (Used in: v2011)

Limiting access to the module to only one thread at the time may be unnecessary: Assu-

me that a lot of the functions in the module *can* be called by different threads at the same time without problems. What is the arguments for making a potentially more complex access control for the module that would allow this ?

Expectations: When we divided the systems into threads in the first place, we did it for a reason... (e.g. to decouple modules, to make reasoning about time simpler, to get performance improvement by paralallism etc...). A too strict accesscontrol will serialize the program again, nullifying these gains.

1-204) (Used in: v2011kont , v2011)

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?

Expectations: The concept with monitors is that when blocked from inside, the monitor lock is released... For this to work the possible blocking points must be clear, leaving the monitor in a reasonably consistent state for interruption.

When blocking in the inner monitor call the question arises whether or not to release the outer monitor lock. This question have no good answers.

1-205) (Used in: v2011)

How does Ada avoid any of these problems with nested monitor calls.

Expectations: In Ada blocking - for any reason - from the inside of a monitor is a run-time error. Monitor calls is assumed to be “short”. Nested monitor calls *are* possible, just not blocking on one.

Unnecessary detail for full score: Blocking on monitor access itself is not defined as blocking, but blocking on a guard is.

1-206) (Used in: v2011)

Inheritance Anomaly is a potential problem when we imagine inheriting from a class (in an objectoriented setting like Java or c++) where any methods synchronize. Explain where the problem lies.

Expectations: For objectoriented inheritance we can add to or override any features of the base class. It is not given (and is in fact false) that extending synchronization by the same mechanisms will work at all. The interaction between synchronication code in parent and child classes becomes complex, and even in some cases the base class synchronization code becomes impossible to tweek into meaningful child class synchronization...

Any example will of course also do.

1-207) (Used in: v2011)

I have suggested that systems based on shared variable-based synchronization scales badly. Can you argue this more generally?

Expectations: At minimum, my comments could be repeated: Nested monitor calls, inheritance anomaly.

Anyway reasoning is expected to start at maintainability:

Synchronization mechanisms are operating systems mechanisms; global entities like global variables.

Semaphores are global, the Java object lock can be reserved from the outside by a "o.synchronized " block.

Analysing the system for deadlocks (and other race conditions) is a global analysis.

1-208) (Used in: kont2015) **(7x)** Many techniques that can contribute to fault tolerance have been presented in this course. For each of those under: Describe shortly how they work/what they entail, and how they contribute to fault tolerance.

- Acceptance Tests
- Merging of Error Modes
- Static Redundancy
- Dynamic Redundancy
- Recovery Points
- Backward Error Recovery
- Forward Error Recovery
- N-version Programming
- Recovery Blocks
- Atomic Actions
- Transactions
- Process Pairs
- Error Injection
- Fail Fast

- Checkpoint/Restart

Expectations:

- Acceptance tests
- Merging of Error Modes
- Static Redundancy
- Dynamic Redundancy
- Recovery Points
- Backward Error Recovery
- Forward Error Recovery
- N-version Programming
- Recovery Blocks
- Atomic Actions
- Transactions
- Process Pairs
- Error Injection; Lurespørsmål: Bidrar ikke til Fault tolerance, bare muliggjør testing av det.
- Fail Fast
- Checkpoint/Restart

1-209) (Used in: kont2015)

Real-time software is well and good, but no real-time demands is met when the program is not running.

One of the difficult features to give a system is to allow upgrading the system to a new version without loosing service.

Outline shortly how you would approach making a system with this feature.

Expectations: Process pairs comes a long way; We can take down the backup, and replace it with a version running the new version of the software, before taking down the primary and provoking a take-over into the new version. The only thing to be aware

of is the status messages that must be versioned, along with the new version of the software being able to relate to 'old' status messages.

Another challenge with process pairs is when the state of the program is too large to fit in a status message - any reflections on this is great, but outside of scope here.

Readers/Writers locks is a variant of mutual exclusion where we recognize that we do not need mutual exclusion between more readers. The writers still needs to run under mutual exclusion — with each other and with the readers.

1-210) (Used in: kont2015)

How is it that we do not need mutual exclusion between multiple readers, when it is necessary between the writers?

Expectations: A reader (per definition, here) does not change the (value of) the resource, meaning that all readers will see the same, consistent state - that is, no problem.

More writers may if they interrupt each other see intermediate (inconsistent) states or overwrite each other partial writes.

1-211) (Used in: kont2015)

Why/When would we use readers/writers locks in place of ordinary mutual exclusion?

Expectations: This is a performance issue - typically we have a lot of readers/reads and few writers/writes, and locking/serializing these calls hampers performance/removes parallelism.

1-212) (Used in: kont2015)

With Adas Protected Objects, readers/writers locks are very easy to make; Explain how.

Expectations: Ada differs between functions (that cannot change the object state) and procedures (that can), and allows functions to "run in parallel". That is, use functions for readers, and procedures for writers.

1-213) (Used in: kont2015) **(2x)** A small challenge: Sketch (pseudo-code) how one can make readers/writers locks with Javas synchronization mechanisms (synchronized methods, wait, notify, notifyAll). Hint: Make the functions startRead, stopRead, startWrite og stopWrite.

Expectations: The clue is that we need variables (int readers,bool writing) for keeping track of active readers and active writers.

```
1   public synchronized void StartWrite() {
2     while(readers > 0 || writing) wait();
```

```
3       writing = true;
4     }
5
6     public synchronized void StopWrite() {
7       writing = false;
8       notifyAll();
9     }
10
11    public synchronized void StartRead() {
12      while(writing) wait();
13      readers++;
14    }
15
16    public synchronized void StopRead() {
17      readers--;
18      if(readers == 0) notifyAll();
19    }
```

1-214) (Used in: kont2015)

If we envision a lot of readers and writers, then such use of *notifyAll* may be unfortunate. Why?

Expectations: We can have a lot of waiters at a time, and waking all of them, only to loop and go to sleep again is a waste (and bad abstraction).

1-215) (Used in: kont2015) **(2x)** A larger challenge: Sketch (pseudo-code) how one can make readers/writers locks with semaphores.

Expectations: This is taken from the little book of semaphores. The clue is keeping track of the "empty room".

```
1   Semaphore mutex(1), roomEmpty(1);
2   int activeReaders = 0;
3
4   void StartWrite() {
5     roomEmpty.wait();
6   }
7
8   void StopWrite() {
9     roomEmpty.signal();
10  }
11
12  void StartRead() {
13    mutex.wait();
14    activeReaders++;
15    if(activeReaders == 1) roomEmpty.wait();
16    mutex.signal();
17  }
18
19  void StopRead() {
20    mutex.wait();
```

```
21      activeReaders--;
22      if(activeReaders == 0) roomEmpty.signal();
23      mutex.signal();
24    }
```

1-216) (Used in: kont2015)

What is starvation?

Expectations: A thread does "by accident" not get necessary resources.

1-217) (Used in: kont2015)

Readers/writers locks is an interesting case for discussing starvation. Why?

Expectations: The need for readers/writers locks is motivated by a lot of readers. If there are a lot of readers, they may overlap in execution, starving any writers.

1-218) (Used in: kont2015)

If you mean that one of the solutions in 1-4 or 1-7) is safe against starvation; Explain why. If not; suggest how (Pseudo-code not necessary) one could make a version that avoids starvation. (for either the Java or the semaphores version)

Expectations: The standard solution here would assume that there are few writers, and therefore writes can be given priority, while the readers will always get to run since all can be released when the writers are finished. We solve this by keeping track of "waitingWriters" and not letting in any readers while this is nonzero.

However, this assumption has not been mentioned, so for a full score this must at least be mentioned.

The general solution - that would lead to a complete robustness against starvation would have to keep track of the order of the requests. Following this line of thought should lead to a full score more easily.

1-219) (Used in: kont2015)

Assume that the Ada scheduler works in a way that makes the trivial readers/writers lock implementation from 1-3 safe against starvation. Comment on this implementation from a code quality perspective.

Expectations: It works, but we cannot see that it does, by looking only at the code; We still have to *know* how Ada works, and accept the dependencies on this behaviour.

By looking at the code we cannot see whether it works or not; We need to know details of the Ada run-time system to claim it does. The code is unmaintainable for anyone not that schooled in Ada; It is bad code.

(Another way of arguing:) There are implicit dependencies between the implementation and the Ada run-time - breaking module boundaries.

A simple comparison between the Google Go programming language and the almost-dead programming language Occam was done (a few years ago). Both programs had two threads on the same priority, one writing 'A's and the other 'B's.

The Google Go program produces an almost-regular sequence of A's and B's: "ABABABABAABABABABABBAABABAB..."

The Occam program produced "ABAAAAAAAAAAAAAAAAAA..." (A's continuing)

1-220) (Used in: kont2015)

Comment on how these two different run-time system strategies can lead to differences in code quality.

Expectations: Go tries to "be fair" and to help the programmer to avoid starvation. A program that works perfectly in Go v. 2.045 might fail miserably in Go v. 2.046 if they change the scheduler slightly.

The Occam approach will refuse any dependencies between the user program and the run-time system, and if you have written a program prone to starvation, it will beat you over the head with it in the testing phase.

Any mature argument from the student deserves points...

..you know what /I/ mean: Occam is the safer path.

1-221) (Used in: kont2015)

A group some years ago handed in this lift project. What can you say about their division of the project into modules, module naming, or on the project as a whole? (Comment only on the few most important things, good or bad)

```
 1    61 channels.h
 2    39 elevator.c
 3   151 elev_driver.c
 4    42 elev_driver.h
 5   344 elev_FSM.c
 6   139 elev_FSM.h
 7    86 io.c
 8    62 io.h
 9    20 Makefile
10   645 network.c
11   142 network.h
12   171 sch_dist_calc.c
13    24 sch_dist_calc.h
14   296 sch_elev_manager.c
```

```
 22 sch_elev_manager.h
135 sch_main.c
 14 sch_main.h
304 sch_orders.c
 95 sch_orders.h
123 sch_packets.c
 35 sch_packets.h
145 sch_peers.c
 39 sch_peers.h
 76 sch_print_info.c
 18 sch_print_info.h
102 sch_types.h
188 storeOrders.c
 30 storeOrders.h
 44 timer.c
 32 timer.h
```

The numbers are the line count of each file. The total project consists of 1183 lines with ';' making it of approximately average size.

Expectations:

I find this a nice division into modules. One must guess that "sch_" is a prefix for a "scheduler" - a supermodule probably using the network module to negociate and distribute the orders on to lifts, but then the responsibilities are relatively clear.

"sch_main" could possibly be better named; There is some confusion on where the main function of the system is. sch_ as a naming convention for a super module is ok as far as it goes; using a folder or a library with a single exported header file is probably more appropriate here if sch_main represents some kind of interface.

This is an open question; Any reasonable answer should be rewarded.

1-222) (Used in: kont2015)

Here is the elev_FSM.h file from the same project. Comment on the few most important points, good or bad.

```
#ifndef __INCLUDE_ELEV_FSM_H__
#define __INCLUDE_ELEV_FSM_H__

/*
  About this module:
  A Finite State Machine controlling the elevator.

  The FSM is assigned a destination floor,
  and does best effort in reaching this floor.
  It flags itself as ''defect'' it something goes wrong.

  Known bugs:
  - Due to different specifications in the elevator hardwares,
```

```
14    the elevator tram may sometimes drive beyond the end floors,
15    getting the elevator stuck below the ground floor, or above the top floor.
16  */
17
18  #include "elev_driver.h"
19  #include "timer.h"
20
21  #define FALSE                   0
22  #define TRUE                    1
23
24  // For the elevator FSM
25  // Directions
26  #define direction_HERE          0
27  #define direction_UP            1
28  #define direction_DOWN          2
29
30  // States
31  #define state_UP                0
32  #define state_DOWN              1
33  #define state_OPEN_DOOR         2
34  #define state_IDLE              3
35  #define state_STOP              4
36  #define state_ACCOMPLISHED      5
37  #define state_OBSTRUCTION       6
38
39  // Times (delays)
40  #define Time_doorOpen           2500        // Milliseconds
41  #define Time_brake                 5        // Milliseconds
42  #define Time_transportLimit     3000        // Milliseconds
43
44  // Velocities
45  #define speed_UP                200
46  #define speed_DOWN              -200
47  #define speed_STOP              0
48
49  /*
50    Holds all information about the elevator FSM,
51    in order to make it behave in an more or less intelligent way.
52    All of these fields are available to the module using this module,
53    but none should be altered, only read.
54  */
55  typedef struct // The elevator FSM type
56  {
57    // States
58    int previousState;
59    int currentState;
60
61    // Info about the position
62    int positionUnknown;                        // TRUE if the position is
          unknown. Only used during startup. Makes the FSM use sensors to
          determine the position
63    int atAFloor;                               // TRUE if the elevator is
          situated at a floor (and not in a gap)
```

```
64    int currentFloor;
65    int currentFloorGap;
66
67    // Booleans describing trigger
68    // Internal (6 triggers)
69    int trig_enteringFloor;
70    int trig_leavingFloor;
71    int trig_stopButtonPressed;
72    int trig_doorTimerDownAndNotObstruction;
73    int trig_transportTimerDown;
74    int trig_obstruction;
75
76    // External (4 triggers)
77    int trig_acknowledgeTaskSuccess;        // TRUE if the function acknowledge
          () has been run from previus iteration
78    int trig_setTask;                        // TRUE if the task was updated
          since last time. Then usually destination_Floor is also updated
79    int trig_cancelTask;
80    int trig_resumeTaskAfterStop;
81
82
83    // Help variables for deciding specific branches etc
84    int destinationFloor;
85    int openDoorOnArrival;
86    int defect;
87    int defectEngine;
88
89    // Help variables
90    timerStruct doorTimer;                    // The timer for the door
91    timerStruct transportTimer;               // The timer for the
          transportation
92    int sensor_stopButtonPrevValue;           // The stop button sensor's
          previous value
93
94    // Sensor information. In order to avoid reading the sensors more than once
           per iteration.
95    int sensor_floorSensor;
96    int sensor_stopButton;
97    int sensor_obstruction;
98  } FSM_data;
99
100
101 int init_elev_FSM(FSM_data *FSM);
102 void kill_elev_FSM(FSM_data *FSM);
103
104 /*
105   Update the FSM. Absolutly no inputs or outputs from the scheduler through
          this one,
106   just let the FSM process the info recieved from last time
107 */
108 void update_elev_FSM(FSM_data *FSM);
109
110
```

```
111 // FETCH INFO
112 /*
113   Returns TRUE if the elevator can be assigned a new task. This is only the
        case when the state is IDLE, UP or DOWN,
114   in addition to some minor conditions (like: the elevator is functional).
        This must be true when calling trig_setTask()
115 */
116 int FSM_get_canAssignNewTask(FSM_data *FSM);
117
118
119 // TRIGGERS FROM THE SCHEDULER
120 void FSM_trig_acknowledgeTaskSuccess(FSM_data *FSM);
121
122 /*
123   Sets a new destination. Must only be called when get_canAssignNewTask() is
        true.
124 */
125 void FSM_trig_setTask(FSM_data *FSM, int destination);
126
127 /*
128   Drives to the nearest floor, and enters state IDLE, without opening the
        door
129 */
130 void FSM_trig_cancelTask(FSM_data *FSM);
131
132 /*
133   Resumes the previous task after the stop button was pressed.
134   This trigger is not in use.
135 */
136 void FSM_trig_resumeTaskAfterStop(FSM_data *FSM);
137
138
139 #endif // #ifndef __INCLUDE_ELEV_FSM_H__
```

Expectations: There is a lot to comment on here; Any reasonable reflections should be rewarded. I'll try to order my list of comments on importance:

- The FSM_data struct is too big; A lift does not have this much state. This will add complexity to algorithms other places in the code, and introduce implicit dependencies between functions...
- Is it necessary to export all that is exported here (The struct type, all the defines)? I would say hopefully not. As it is, this is not a module I would care to be the user of.
- The 'trig' section is a nice list of the real interface here? These functions give me an ok image of how the scheduler communicates with the 'FSM', and the fact that they handle a restart after stop explicitly is good.
- ...

1-223) (Used in: kont2015)

Here is a function from the elev_FSM.c file from the same project. Comment on the few most important points, good or bad.

```
1  /*
2          Needs to be called once per iteration.
3          This is the brain of the elevator's logic.
4  */
5  void update_elev_FSM(FSM_data *FSM) {
6    priv_fetchSensorSignals(FSM);
7    priv_internalTriggers(FSM);
8
9
10   if(FSM->trig_obstruction){
11     if (FSM->currentState != state_OBSTRUCTION)
12       priv_setNewState(FSM, state_OBSTRUCTION);
13   }
14
15   if(FSM->currentState == state_OBSTRUCTION && FSM->trig_obstruction==0)
16     priv_setNewState(FSM, state_IDLE);
17
18   if (FSM->trig_stopButtonPressed) {
19     if (FSM->currentState != state_STOP)
20       priv_setNewState(FSM, state_STOP);
21   }
22
23   if (FSM->trig_transportTimerDown) {
24     if ((FSM->currentState == state_UP) || (FSM->currentState == state_DOWN))
           {
25       FSM->defect                     = TRUE;
26       FSM->defectEngine               = TRUE;
27       priv_elevatorStopEngine(FSM);
28     }
29   }
30
31   if (FSM->trig_doorTimerDownAndNotObstruction)
32     if ((FSM->currentState == state_OPEN_DOOR) && (!(FSM->defect)))
33       priv_setNewState(FSM, state_ACCOMPLISHED);
34
35   if (FSM->trig_enteringFloor) {
36     FSM->atAFloor        = TRUE;
37     FSM->currentFloor    = FSM->sensor_floorSensor;
38
39     if (FSM->defect)
40       priv_setNewState(FSM, state_OPEN_DOOR);
41     else
42       {
43         if (FSM->positionUnknown) {
44           FSM->positionUnknown = FALSE;
45           FSM->openDoorOnArrival = TRUE;
46           priv_setNewState(FSM, state_IDLE);
47         } else {
48           if (!FSM->openDoorOnArrival) {
```

```
49              FSM->openDoorOnArrival = TRUE;
50              priv_setNewState(FSM, state_IDLE);
51            } else {
52              if (priv_traveledTooDestination(FSM))
53                priv_setNewState(FSM, state_OPEN_DOOR);
54              if (priv_traveledTooFar(FSM)) {
55                FSM->defect = TRUE;
56                priv_elevatorStopEngine(FSM);
57              }
58              if (priv_traveledTooShort(FSM))
59                startTimer(&(FSM->transportTimer)); // Reset timer
60            }
61          }
62        }
63    }
64
65    if (FSM->trig_leavingFloor) {
66      FSM->atAFloor = FALSE;
67      if (FSM->currentState == state_UP)
68        FSM->currentFloorGap   = FSM->currentFloor;
69      if (FSM->currentState == state_DOWN)
70        FSM->currentFloorGap   = FSM->currentFloor - 1;
71
72      startTimer(&(FSM->transportTimer)); // Reset timer
73    }
74
75    if (FSM->trig_acknowledgeTaskSuccess)
76      if (FSM->currentState == state_ACCOMPLISHED)
77        {
78          FSM->trig_acknowledgeTaskSuccess = FALSE;
79          priv_setNewState(FSM, state_IDLE);
80        }
81
82    if (FSM->trig_cancelTask) {
83      FSM->trig_cancelTask = FALSE;
84      FSM->openDoorOnArrival = FALSE;
85    }
86
87    if (FSM->trig_setTask) {
88      if ((!(FSM->defect)) && ((FSM->currentState == state_IDLE) || (FSM->
          currentState == state_STOP) ||
89                               (FSM->currentState == state_UP) || (FSM->
                                     currentState == state_DOWN))) {
90        FSM->trig_setTask = FALSE;
91        FSM->openDoorOnArrival = TRUE;
92        int directon = priv_chooseDirecton(FSM);
93        if (directon == direction_UP)
94          priv_setNewState(FSM, state_UP);
95        else if (directon == direction_DOWN)
96          priv_setNewState(FSM, state_DOWN);
97        else
98          priv_setNewState(FSM, state_OPEN_DOOR);
99      }
```

```
100     }
101
102     if (FSM->trig_resumeTaskAfterStop) {
103       FSM->trig_resumeTaskAfterStop = FALSE;
104       priv_setNewState(FSM, FSM->previousState);        // Resume the previous
              state
105     }
106
107     elev_FSM_UpdateLights(FSM);
108   }
```

Expectations: There is a lot to comment on here; Any reasonable reflections should be rewarded. I'll try to order my list of comments on importance: Nah, I give up.

Sverre the prophet thunders from the mountaintop:

- This is not a 'fsm'. Rather than responding to events, this function analyses the all-too-big data-structure, and does actions based on this.
- This is terrible ad-hoc code that nobody ever can say will work, leaving us to forever add features by debugging.
- It could be worse; the function could easily have been longer. — And probably would have if the students had spent more time on the project. As the system is maintained, this function (and possibly the struct) will grow monotonously, leaving larger and larger parts of it dead code that nobody will ever dare to remove.
- "This is the brain of the elevator's logic." In a sense this is good to know; "If there is a bug in the system, it is probably here"... :-)
- Still, since there are some very good things in the higher-level design in the project, the failing of this code may not be irrecoverable.