NTNU
Norwegian University of
Science and Technology

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi
matematikk og elektroteknikk

Eksamen i TTK4145 Sanntidsprogrammering 15. Aug 2015 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 Shared Variable Synchronization

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.

1-1) 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.

1-2) 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.

1-3) 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.

1-4) (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.

```
public synchronized void StartWrite() {
  while(readers > 0 || writing) wait();
  writing = true;
}

public synchronized void StopWrite() {
  writing = false;
  notifyAll();
}

public synchronized void StartRead() {
    while(writing) wait();
    readers++;
  }

  public synchronized void StopRead() {
    readers--;
    if(readers == 0) notifyAll();
  }
```

1-5) 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.

1-6) 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).

1-7) (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".

```
  Semaphore mutex(1), roomEmpty(1);
  int activeReaders = 0;

  void StartWrite() {
    roomEmpty.wait();
  }

  void StopWrite() {
    roomEmpty.signal();
  }

  void StartRead() {
    mutex.wait();
    activeReaders++;
    if(activeReaders == 1) roomEmpty.wait();
    mutex.signal();
  }

  void StopRead() {
    mutex.wait();
    activeReaders--;
    if(activeReaders == 0) roomEmpty.signal();
    mutex.signal();
  }
```

1-8) What is starvation?

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

1-9) 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.

1-10) 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.

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.

1-11) 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 application.

1-12) 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.

1-13) 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 "ABAAAAAAAAAAAAAAAAAAAA..." (A's continuing)

1-14) 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.

2 Fault Tolerance

2-1) (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

2-2) 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.

3 Code Quality

3-1) 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)

```
  61 channels.h
  39 elevator.c
 151 elev_driver.c
  42 elev_driver.h
 344 elev_FSM.c
 139 elev_FSM.h
  86 io.c
  62 io.h
  20 Makefile
 645 network.c
 142 network.h
 171 sch_dist_calc.c
  24 sch_dist_calc.h
 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.

3-2) 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,
  the elevator tram may sometimes drive beyond the end floors,
  getting the elevator stuck below the ground floor, or above the top floor.
*/

#include "elev_driver.h"
#include "timer.h"

#define FALSE                    0
#define TRUE                     1

// For the elevator FSM
// Directions
#define direction_HERE           0
#define direction_UP             1
#define direction_DOWN           2

// States
#define state_UP                 0
#define state_DOWN               1
#define state_OPEN_DOOR          2
#define state_IDLE               3
#define state_STOP               4
#define state_ACCOMPLISHED       5
#define state_OBSTRUCTION        6

// Times (delays)
#define Time_doorOpen            2500        // Milliseconds
#define Time_brake                  5        // Milliseconds
#define Time_transportLimit      3000        // Milliseconds

// Velocities
#define speed_UP                 200
#define speed_DOWN               -200
#define speed_STOP               0

/*
```

```c
  Holds all information about the elevator FSM,
  in order to make it behave in an more or less intelligent way.
  All of these fields are available to the module using this module,
  but none should be altered, only read.
*/
typedef struct // The elevator FSM type
{
  // States
  int previousState;
  int currentState;

  // Info about the position
  int positionUnknown;                         // TRUE if the position is unknown. Only used during startup. Makes the FSM use sensors to determine the position
  int atAFloor;                                 // TRUE if the elevator is situated at a floor (and not in a gap)
  int currentFloor;
  int currentFloorGap;

  // Booleans describing trigger
  // Internal (6 triggers)
  int trig_enteringFloor;
  int trig_leavingFloor;
  int trig_stopButtonPressed;
  int trig_doorTimerDownAndNotObstruction;
  int trig_transportTimerDown;
  int trig_obstruction;

  // External (4 triggers)
  int trig_acknowledgeTaskSuccess;        // TRUE if the function acknowledge() has been run from previus iteration
  int trig_setTask;                          // TRUE if the task was updated since last time. Then usually destination_Floor is also updated
  int trig_cancelTask;
  int trig_resumeTaskAfterStop;


  // Help variables for deciding specific branches etc
  int destinationFloor;
  int openDoorOnArrival;
  int defect;
  int defectEngine;

  // Help variables
  timerStruct doorTimer;                       // The timer for the door
  timerStruct transportTimer;                  // The timer for the transportation
  int sensor_stopButtonPrevValue;              // The stop button sensor's previous value

  // Sensor information. In order to avoid reading the sensors more than once per iteration.
  int sensor_floorSensor;
  int sensor_stopButton;
  int sensor_obstruction;
} FSM_data;


int init_elev_FSM(FSM_data *FSM);
void kill_elev_FSM(FSM_data *FSM);

/*
  Update the FSM. Absolutly no inputs or outputs from the scheduler through
      this one,
  just let the FSM process the info recieved from last time
*/
void update_elev_FSM(FSM_data *FSM);


// FETCH INFO
/*
  Returns TRUE if the elevator can be assigned a new task. This is only the
      case when the state is IDLE, UP or DOWN,
  in addition to some minor conditions (like: the elevator is functional).
      This must be true when calling trig_setTask()
*/
int FSM_get_canAssignNewTask(FSM_data *FSM);


// TRIGGERS FROM THE SCHEDULER
void FSM_trig_acknowledgeTaskSuccess(FSM_data *FSM);

/*
  Sets a new destination. Must only be called when get_canAssignNewTask() is
      true.
*/
void FSM_trig_setTask(FSM_data *FSM, int destination);

/*
  Drives to the nearest floor, and enters state IDLE, without opening the
      door
*/
void FSM_trig_cancelTask(FSM_data *FSM);

/*
  Resumes the previous task after the stop button was pressed.
  This trigger is not in use.
*/
void FSM_trig_resumeTaskAfterStop(FSM_data *FSM);


#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.
- ...

3-3) Here is a function from the elev_FSM.c file from the same project. Comment on the few most important points, good or bad.

```c
/*
        Needs to be called once per iteration.
        This is the brain of the elevator's logic.
*/
void update_elev_FSM(FSM_data *FSM) {
  priv_fetchSensorSignals(FSM);
  priv_internalTriggers(FSM);


  if(FSM->trig_obstruction){
    if (FSM->currentState != state_OBSTRUCTION)
      priv_setNewState(FSM, state_OBSTRUCTION);
  }

  if(FSM->currentState == state_OBSTRUCTION && FSM->trig_obstruction==0)
    priv_setNewState(FSM, state_IDLE);

  if (FSM->trig_stopButtonPressed) {
    if (FSM->currentState != state_STOP)
      priv_setNewState(FSM, state_STOP);
  }

  if (FSM->trig_transportTimerDown) {
    if ((FSM->currentState == state_UP) || (FSM->currentState == state_DOWN))
        {
      FSM->defect                      = TRUE;
      FSM->defectEngine                = TRUE;
      priv_elevatorStopEngine(FSM);
    }
  }

  if (FSM->trig_doorTimerDownAndNotObstruction)
    if ((FSM->currentState == state_OPEN_DOOR) && (!(FSM->defect)))
      priv_setNewState(FSM, state_ACCOMPLISHED);

  if (FSM->trig_enteringFloor) {
    FSM->atAFloor        = TRUE;
    FSM->currentFloor    = FSM->sensor_floorSensor;

    if (FSM->defect)
      priv_setNewState(FSM, state_OPEN_DOOR);
    else
      {
        if (FSM->positionUnknown) {
          FSM->positionUnknown = FALSE;
          FSM->openDoorOnArrival = TRUE;
          priv_setNewState(FSM, state_IDLE);
        } else {
          if (!FSM->openDoorOnArrival) {
            FSM->openDoorOnArrival = TRUE;
            priv_setNewState(FSM, state_IDLE);
          } else {
            if (priv_traveledTooDestination(FSM))
              priv_setNewState(FSM, state_OPEN_DOOR);
            if (priv_traveledTooFar(FSM)) {
              FSM->defect = TRUE;
              priv_elevatorStopEngine(FSM);
            }
            if (priv_traveledTooShort(FSM))
              startTimer(&(FSM->transportTimer)); // Reset timer
          }
        }
      }
  }

  if (FSM->trig_leavingFloor) {
    FSM->atAFloor = FALSE;
    if (FSM->currentState == state_UP)
      FSM->currentFloorGap   = FSM->currentFloor;
    if (FSM->currentState == state_DOWN)
      FSM->currentFloorGap   = FSM->currentFloor - 1;

    startTimer(&(FSM->transportTimer)); // Reset timer
  }

  if (FSM->trig_acknowledgeTaskSuccess)
    if (FSM->currentState == state_ACCOMPLISHED)
      {
        FSM->trig_acknowledgeTaskSuccess = FALSE;
        priv_setNewState(FSM, state_IDLE);
      }

  if (FSM->trig_cancelTask) {
    FSM->trig_cancelTask = FALSE;
    FSM->openDoorOnArrival = FALSE;
  }

```

```c
  if (FSM->trig_setTask) {
    if ((!(FSM->defect)) && ((FSM->currentState == state_IDLE) || (FSM->
        currentState == state_STOP) ||
                              (FSM->currentState == state_UP) || (FSM->
                                 currentState == state_DOWN))) {
      FSM->trig_setTask = FALSE;
      FSM->openDoorOnArrival = TRUE;
      int directon = priv_chooseDirecton(FSM);
      if (directon == direction_UP)
        priv_setNewState(FSM, state_UP);
      else if (directon == direction_DOWN)
        priv_setNewState(FSM, state_DOWN);
      else
        priv_setNewState(FSM, state_OPEN_DOOR);
    }
  }

  if (FSM->trig_resumeTaskAfterStop) {
    FSM->trig_resumeTaskAfterStop = FALSE;
    priv_setNewState(FSM, FSM->previousState);        // Resume the previous
        state
  }

  elev_FSM_UpdateLights(FSM);
}
```

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.

4 Scheduling

4-1) (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.

4-2) 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.

4-3) 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)

5 Modeling of concurrent programs - FSP

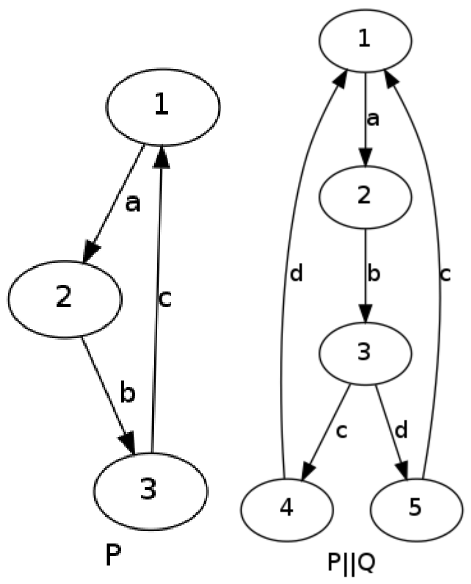
We have the following three FSP processes:

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

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.

5-1) Draw the transition diagram for P and $P||Q$.

Expectations:



5-2) $P||Q||R$ 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.

5-3) 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.

5-4) 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).