NTNU
Norwegian University of
Science and Technology

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi
matematikk og elektroteknikk

Eksamen i TTK4145 Sanntidsprogrammering 28. May 2014 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 Expressive power and Ease of Use

1-1) 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?

1-2) 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.

$wait()$**:** 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.

1-3) 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.

```
protected body Bounded_Buffer is
  entry Get (Item : out Data_Item) when Num /= 0 is
  begin
    Item := Buf(First);
    First := First + 1; Num := Num - 1;
  end Get;

  entry Put (Item : in Data_Item) when Num /= Buffer_Size is
  begin
    Last := Last + 1; Num := Num + 1;
    Buf(Last) := Item
  end Put;
end Bounded_Buffer;
```

Figur 1: Bounded Buffer in Ada

```
  public synchronized void put(int item)
  {
      while (numberInBuffer == size) wait();
      last = (last + 1) % size ;
      numberInBuffer++;
      buffer[last] = item;
      notifyAll();
  }

  public synchronized int get()
  {
      while (numberInBuffer == 0) wait();
      first = (first + 1) % size ;
      numberInBuffer--;
      notifyAll();
      return buffer[first];
  }
```

Figur 2: Bounded Buffer in Java

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.

1-4) 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.

1-5) Read the attached page from "The little book of semaphores" about "Cigarette smokers 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:

```
protected body Table is

  entry WakeMatchesThread when TobaccoOnTable and PaperOnTable
  begin
  end

  entry WakeTobaccoThread when PaperOnTable and MatchesOnTable
  begin
  end

  entry WakePaperThread when MatchesOnTable and TobaccoOnTable
  begin
  end

  procedure Tobacco()
  begin
    TobaccoOnTable = True
  end

  procedure Matches()
  begin
    MatchesOnTable = True
  end

  procedure Paper()
  begin
    PaperOnTable = True
  end

private
  MatchesOnTable : Boolean = False;
  TobaccoOnTable : Boolean = False;
  PaperOnTable   : Boolean = False;

end Table;
```

1-6) 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.

```
// 3 statusvariables
bool tobaccoOnTable = false;
bool paperOnTable = false;
bool matchesOnTable = false;

// Make a mutex to avoid race conditions
Semaphore pusherMutex(1);

// Make the semaphores for the smoker threads to wait on.
Semaphore sem_wakeTobacco(0);
Semaphore sem_wakeMatches(0);
Semaphore sem_wakePaper(0);

void pusherTobacco(){ // Waits primarily on sem_tobacco
  wait(sem_tobacco);
  // We have tobacco!

  wait(pusherMutex)
    // Now, another incredient may already have been detected
    if(paperOnTable){
      // We have both paper and tobacco; wake the matches smoker
      signal(sem_wakeMatches)
      paperOnTable = false;
    }else if(matchesOnTable){
      // We have both matches and tobacco; wake the paper smoker
      signal(sem_wakePaper)
      matchesOnTable = false;
    }else{
      // No, This was first ingredient
      tobaccoOnTable = true;
    }
  signal(pusherMutex)
}

void pusherPaper(){ ... }
void pusherMatches(){ ... }
```

1-7) 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.

2 Modeling the Bounded Buffer

2-1) 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:

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

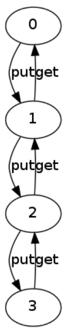
Assume N=3: Draw the transition diagram for this system.

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

2-2) 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:

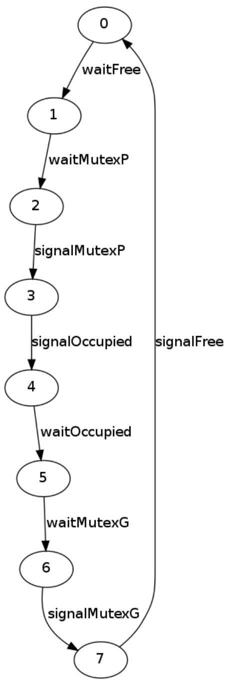
```
PUT = (waitFree -> waitMutexP -> signalMutexP -> signalOccupied -> PUT).
GET = (waitOccupied -> waitMutexG -> signalMutexG -> signalFree -> GET).
M = (waitMutexP -> signalMutexP -> M
  |  waitMutexG -> signalMutexG -> M).
Free(N=5) = Free[N],
Free[i] = when i > 0 waitFree -> Free[i-1]
     | signalFree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

2-3) Assume that the buffer is of size 1 (N=1). Draw the transition diagram.

Expectations:



A Onebuffer bounded buffer

2-4) 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 :-))

2-5) 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

```
BufferProcess(){
  while(1){
    select {
      when (bufferNotFull) element <- putCannel:
        -- enter Element Into Buffer.
      when (bufferNotEmpty) <- getChannel:
        -- retrieve an element and send it as reply
    }
  }
}
```

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.

2-6) 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)

2-7) 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.

3 Code Quality

3-1) In the listing under is a module interface (a C header file) from an old project deliverable. The modules name is "cost". Criticize (concisely, with bullet points) the design.

```c
#ifndef lift_cost_h
#define lift_cost_h

int calculateCost(int currentFloor, int direction, int orderedFloor, int
    orderedDirection);

int downCost[MAX_ELEVATORS][N_FLOORS];
int upCost[MAX_ELEVATORS][N_FLOORS];

void fillCostArrays();
void clearCosts(void);

int lowestCostFloor(int elevator);
int lowestCostDirection(int elevator);

int findBestElevator(int floor, int direction);

void designateElevators();
void clearDesignatedElevator();

int designatedElevator[N_FLOORS][2];

#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.

3-2) This is another module interface. This time with the name of "jobqueue". Criticize it; feel free to compare it to the previous one.

```c
#ifndef _JOB_QUEUE_H
#define _JOB_QUEUE_H

#include <time.h>

//Set flag at given floor in the given direction.
void set_flag(int floor, int direction);

//Deletes flag at given floor in the given direction.
void delete_flag(int floor, int direction);

//Returns one if flag at given direction at given gloor is set.
int get_flag(int floor, int direction);

//Deletes all flags in table.
void delete_all_flags();

//Returns one if there are no flags set in table.
int flag_table_empty();

//Returns one if there are any flags above the given floor.
int any_flags_above(int floor);

//Returns one if there are any flags below the given floor.
int any_flags_below(int floor);

//Prints the table of flags to the terminal.
void print_flag_table();

#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.

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

```c
//Delete flag in table.
void delete_flag(int floor, int direction) {

        if(floor == 0) {
                flag_table[floor][DIR_UP] = 0;
                flag_table[floor][DIR_DOWN] = 0;
                elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
        }
        else if(floor == 3) {
                flag_table[floor][DIR_UP] = 0;
                flag_table[floor][DIR_DOWN] = 0;
                elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
        }
        else{
                if(!any_flags_above(floor) && get_current_direction() ==
                    DIR_UP) {
                        flag_table[floor][DIR_UP] = 0;
                        flag_table[floor][DIR_DOWN] = 0;

                        elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
                }
                else if(!any_flags_below(floor) && get_current_direction() ==
                     DIR_DOWN) {
                        flag_table[floor][DIR_UP] = 0;
                        flag_table[floor][DIR_DOWN] = 0;

                        elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
                }

                flag_table[floor][direction] = 0;
        }

        if(direction == DIR_UP && floor < 3) { elev_set_button_lamp(
            BUTTON_CALL_UP, floor, 0); }
        if(direction == DIR_DOWN && floor > 0) { elev_set_button_lamp(
            BUTTON_CALL_DOWN, floor, 0);}

        elev_set_button_lamp(BUTTON_COMMAND, floor, 0);
}
```

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.

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 ?

3-4) 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.

3-5) 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.

3-6) Atomic Actions

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

3-7) 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.

4 Scheduling

Given these three tasks:

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

4-1) 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.

4-2) 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.

4-3) 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.

4-4) 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)

5 Fault Tolerance

5-1) Acceptance tests is seen as an important tool for handling errors. What do we gain from using acceptance tests in addition to the more traditional tests on error conditions?

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

5-2) 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

5-3) 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.

5-4) 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.

5-5) 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)

5-6) 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.)

5-7) 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.

5-8) 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.