NTNU
Norwegian University of
Science and Technology

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi
matematikk og elektroteknikk

Eksamen i TTK4145
Sanntidsprogrammering
22. May 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

1-1) 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.

1-2) Can deadlocks and race conditions happen in a message passing system?

Expectations: Yes

1-3) 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.

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.

1-4) 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.

1-5) 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.

1-6) 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.

1-7) 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.

1-8) 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.

1-9) (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 }
```

1-10) 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.

1-11) 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()).

1-12) (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
```

2 Messagepassing

2-1) 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)

2-2) (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 }
```

2-3) 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.

2-4) 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 Scheduling

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)

3-1) 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.

3-2) 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.

3-3) 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.

3-4) 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.

3-5) 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.

4 Fault Tolerance

"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.

4-1) 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.

4-2) 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).

4-3) 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...

4-4) What purpose does process pairs fill?

Expectations:

Obviously fault recovery, but also importantly availability, minimizing service downtime.

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

4-5) 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.

4-6) 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.

An Atomic Action has start, side and end boundaries.

4-7) 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.

4-8) 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.

5 Code Quality

5-1) 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.

5-2) 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.

5-3) 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.