NTNU
Norwegian University of
Science and Technology

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi
matematikk og elektroteknikk

Eksamen i TTK4145 Sanntidsprogrammering 25. May 2016 9.00-13.00 Sensorveiledning

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 possibly where marked with (2x), (3x) etc.

Feel free to give your answers in English or Norwegian (...or any reasonable mix of these...)

Be certain your handwriting is readable.

Permitted examination aids:

D: No printed or hand-written support material is allowed. A specific basic calculator is allowed.

Contact person during the exam:

Sverre Hendseth

Telephone 98443807

1 Fault Tolerance

1-1) The term “Fault Tolerance” encompasses more/something else than just minimizing the number of bugs in the system. What?

Expectations: ...that the system should behave as specified *even though* there are bugs there.

1-2) Testing your system is seen as an insufficient technique for making a fault tolerant system. Why?

Expectations: Fault tolerance is always about being tolerant to the bug that is not (yet) found.

Ensuring an fault-free system (which testing also cannot do - we can only test for the **presence** of bugs - not the absense) would be an alternative strategy to fault tolerance.

1-3) In traditional real-time software (using “shared variable synchronization” between concurrent threads), we have a class of bugs that makes the shortcommings of testing especially clear. Which class of bugs? Give an example.

Expectations: Race Condisions.

Ex: Typcally the suspend/resume bug, or the classical deadlock, or dining philosophers.

For each of the three systems in the next tasks you should point out how you would approach making them fault tolerant. List the techniques you would use together with short explanations of how they contribute to making the system fault tolerant.

1-4) System 1 is a multi threaded system to be written in C, using semaphores to protect shared resources. The size of the system and the interactions between the threads are such that you see no way of guaranteeing the complete absence of deadlocks. The system should be made tolerant to deadlocks.

Expectations: The task here is to handle the fact that deadlocks happen - that is; detect deadlocks and then loosen the knot in some way.

- Detection: Watchdog is a simple way. Introducing a lock manager that detects deadlocks is another.
- Handling: We need to introduce “preemption of resources” in some manner - aborting/restarting either threads or tasks. The problem here is to make this preemption without leaving the system in an inconsistent state. I would say structuring the systems functionality into atomic actions/transactions is the feasible

way.

Any mature answer should be credited.

1-5) System 2 is the controller program for a single lift as you know it from the project. That is, there is no coordination of more lifts, but the spec still says that no button presses should be lost, and you should protect against power outages, hard disk crashes etc.

Expectations:

- Redundancy! If one controller/controller/hard disk fails, we should rely on the other one — We have a number of patterns here; static redundancy, process pairs, n-version programming...
- Merging of failure modes: *If* something/anything goes wrong, then fall back on trusting the redundancy.
- Acceptance tests: This way of detecting errors ensures that even unexpected errors are handled.

1-6) System 3 is a system where timing behaviour is critical, but where making a system that gurarantees that all deadlines are met is seen as infeasible/ too conservative. Your system should be tolerant to timing errors.

Expectations: This has not been discussed deeply in the lectures; The students should still be able to reason on the problem.

Any mature answer should be credited.

Also here detection and handling is a way to go: Deadline misses and/or overload situations can be detected by comparing the time of the sideffects with their deadlines - or by having timers interrupt us when a task is out of time.

Handling must be application dependent and if the students answer hints of understanding this i think it is very good.

2 Not Fault Tolerance

We now leave fault tolerance and assume you are challenged to argue that your system has no bugs — like “it has no deadlocks”, “all deadlines are met”, “all sent messages are received”, or even “The system behaves as specified”;

2-1) Also here 'testing' fails as a technique: Why?

Expectations: We can only test for the **presence** of bugs - not the absense. The argument that "I have tested the system very thoroughly"is not an argument that there are no bugs left.

You are given two systems to comment on:

2-2) System 1: The setting is much like in task 1-4; A shared variable system written in C using semaphores, but here it may be possible to make a system that is guaranteed to be without deadlocks. List a number of approaches to making deadlock-free systems.

Expectations:

- The priority ceiling schedulig protocol. (If all tasks reasource usage is known at compile time.)
- Or using a lock manager avoiding risking deadlocks (bankers alg.) (If all tasks reasource usage is known at compile time.)
- Formal verification. Like LTSA, FSP etc.
- Any deadlock detecting/handling scheme.
- Invalidating any of the 4 necessry conditions of deadlocks.
- ...

2-3) System 2: Here we have timing demands on the system, like in 1-6, but we explore the possibility of making a system that guarantees all deadlines. Explain shortly how you would go about making such a system.

Expectations: Divide the system into one thread per timing demand, choose a predictable scheduler, make safe estimates of WCET, perform a schdulability proof.

3 Shared Variable Synchronization

3-1) "Shared variable synchronization" has its challenges: A number of problems (classes of bugs) may occur in a shared variable system that are not present in single thread systems. Which?

Expectations: ...from the lecture notes:

Some standard problems:

- Deadlock: system blocked in circular wait
- Livelock: system locked in a subset of states (like deadlock but we use CPU)
- Starvation: A thread does by accidentnot get necessary resources. Ex: Unfair scheduling. (Ref discussion on whether you should make assumptions on how the sceduling works. Ref. Go vs. Occam: & Who is waked by a signal?)
- Race Condision: A bug that surfaces by unfortunate timing or order of events.

Any complete subset of these are full score.

3-2) Ada entries offers "guards" as in the 'when' statement under (taken from the Ada bounded buffer implementation):

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

Explain how guards in Ada works.

Expectations: If the guard is not satisfied, the process 'calling' the event will block. — until the guard 'opens'.

3-3) Ada guards are severly limited compared to the guards of the (imagined) "conditional critical regions" construct. How? Can you give an example of how this limitation restricts the "expressive power" of Ada?

Expectations: Adas guards can only test on the protected objects private variables.

...so we cannot test on "request parameters" - the parameter of a entry call. The commonly used example is when reserving N amounts of a resource.

3-4) Why are Adas guards limited in this way compared to the guards of Conditional Critcal Regions?

Expectations: The question is "when is it necessary to re-evaluate the guards"? This question has no answer unless one restricts the expressive power of the guards...

4 The Dining Savages

Assume one thread (the cook) bulk reading requests from disk into a buffer (the pot) for a number of worker threads (the savages) to handle. When the pot is empty the cook must refill the pot. See the attached introduction to the “dining savages problem” from the little book of semaphores.

Your task is to implement this system (pseudocode is ok) using semaphores (this is hardest), monitors (like java or posix) and Adas protected objects.

4-1) Show how you can make the synchronization between the savages and the cook with semaphores.

Expectations: Ref. little book of semaphores: (There exists (probably) more valid solutions)

A global variable, and some semaphores:

```
  servings = 0
  mutex = Semaphore(1)
  emptyPot = Semaphore(0)
  fullPot = Semaphore(0)
```

The cook

```
while True:
  emptyPot.wait()
  putServingsInPot(M)
  fullPot.signal()
```

The savage:

```
while True:
  mutex.wait()
    if servings == 0:
      emptyPot.signal()
      fullPot.wait()
      servings = M
    servings -= 1
    getServingFromPot()
  mutex.signal()
  eat()
```

4-2) Show how you can make the synchronization between the savages and the cook with java, using synchronized, wait and notfy/notifyAll (posix version is also acceptable if this is more familiar to you).

Expectations:

Sverres terrible Java pseudocode:

```
class Pot {
  private unsigned int m_servingsLeft = 3;
  private bool m_awakenCook = False;

  synchronized void eat(){
    if(m_servingsLeft > 0){
      m_servingsLeft--;
    }else{
      m_awakenCook = True;
      notifyAll(); // Be certain to wake the cook.
      while(m_servingsLeft == 0) wait();
      m_servingsLeft--;
    }
  }
  synchronized void cook(){
    while(!m_awakenCook) wait();
    m_awakenCook = False;
    m_servingsleft = 3;
    notifyAll(); // Wake all waiting savages.
  }
}
```

4-3) Show how you can make the synchronization between the savages and the cook with Ada using protected objects; functions, procedures, entries w. guards.

Expectations:

Sverres terrible Ada pseudocode:

```
Protected Object Pot:
  private unsigned int servingsLeft = 3;
  entry eat when servingsLeft > 0:
    servingsLeft = servingsLeft-1;

  entry cook when servingsLeft == 0:
    servingsLeft = M;
```

4-4) Comparing the monitors of Java or Posix with the condisional critical regions of Ada: Which would you prefer using for your next synchronization problem? Why?

Expectations: My answer would be to use CCR/Ada-like mechanisms unless testing on request parameters was necessary. I find the guards express wery clearly the conditions necessary not to block. Downside is the limitation of guards that they can only test on private variables of the object.

Ah well - as soon as you get used to the while(!cond) wait(); loops in Java, I guess they are a perfect answer also.

But any mature answer or founded argument gives credit.

We now want to verify that your implementation of the cook and savages behaves as specified, The strategy will be to let the LTSA program compare a model of your implementation (let us say for the semaphore version in task 3-1, modeling the semaphores' 'wait' and 'signal' as events) with a conceptual model of the problem — "A model of the specification". LTSA can check that the implementation does not do anything that is not allowed by the specification.

Our task will now be to build this conceptual model representing the specification. Assume only one savage and a pot size of three.

I give you the FSP models of the savage and the pot:

```
SAVAGE = (eat -> SAVAGE
       | emptyPot -> awakeCook -> eat -> SAVAGE).

POT(M=3) = POT[0],
POT[n:0..M] = (when n==0 fillPot -> POT[M]
            |  when n==0 emptyPot -> POT[0]
            |  when n>0 eat -> POT[n-1]).
```

4-5) Draw the transition diagrams for the savage and the pot.

Expectations: This will have to wait...

4-6) Write the FSP model for the cook.

Expectations: COOK = (awakeCook -> fillPot -> COOK).

4-7) Draw the transition diagram for the complete system.

Expectations: This will have to wait...

4-8) LTSA can also check your system for deadlocks and live locks; How would you recognize a deadlock in a transition diagram?

Expectations: A state with no arrows leading out of it.

4-9) How would you recognize a live lock in a transition diagram?

Expectations: ...that not all states are reachable from all states.

5 Code Quality

5-1) Recommend (give an as general rule as possible) how a function should be named. (A function returns a value but have no side effects.)

Expectations: Functions should be named after their return values.

Code Complete checkpoint: "Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?"

5-2) Criticize the following procedure names shortly (picked from the 2016 projects). Feel free to group them if relevant.

- void elev_master_running()
- void start_elevator()
- int get_q(int floor, int button_type);
- int elev_get_floor_sensor_signal(void);
- void* orders_ctrl_main(void* arg);
- void* comm_ctrl_loop(void* arg);
- void* manage_slave(void* slave_id_void_ptr);

Expectations: Code Complete checkpoint: "Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?"

Also: "The name of a routine is an indication of its quality. If the name is bad and it's accurate, the routine might be poorly designed. If the name is bad and it's inaccurate, it's not telling you what the program does. Either way, a bad name means that the program needs to be changed."

- Describe everything the routine does
- Avoid meaningless, vague, or wishy-washy verbs
- Don't differentiate routine names solely by number

- To name a function, use a description of the return value
- To name a procedure, use a strong verb followed by an object
- Use opposites precisely
- Establish conventions for common operations

5-3) Naming variables well is also an issue: Give some (3-4?) principles guiding variable naming.

Expectations:

Any few favorites (3-4?) from the checklist/key points on page 288-289 of the code complete chapter or corresponding should give a full score.

Sverres favourites are:

- "Does the name fully and accurately describe what the variable represents?"
- "Is the name long enough that you don't have to puzzle it out?"
- "Does the convention distinguish among local, class, and global data?" and "Does the convention distinguish among type names, named constants, enumerated types, and variables?". Sverre likes this; A name could easily signal a lot of its 'context'.
- "Are all words abbreviated consistently?". Sverre: More generally (not only abbreviations): consistence on all levels –> guessability!

5-4) Naming communication channels between threads or processes is *not* discussed in our selected Code Complete chapters... What do you think about these channel names (picked from the 2016 google go projects). Feel free to group them.

- c, ch
- incomingMessagesChan
- udpSendChan
- notAlive
- floorChannel
- primalCh
- sendMsgCh

- lightEventChan
- channel_listen
- channel_write
- to_main, from_main
- messageChan

Expectations:

Most of the names includes a naming convention of 'ch', 'chan' or 'channel' to indicate that it is a channel; this is good.

I find no authoritative guides on google on how to name channels so we are reasoning in the free here.

Any well-founded answers should be rewarded.

Sverres answer:

- 'c' and 'ch' are great if the scope is small. (less than 2-3 lines and no nesting, says code complete - Sverre is a bit more flexible...)
- 'udpSendChan' is the winner in Sverres book: All messages on this channels are sent on udp. I could add 'incommingMessagesChan' to this category, understanding that incomming Messages comes from another lift/node. 'sendMsgChannel' and 'channel_listen' is even weaker, and at the bottom here is channel_write.
- sendMsgCh, channel_listen, channel_write: Not specific enough or too dependent on context? Of course it depends on the context and which metaphors are established in the system already.
- messageChan: No seriously: All channels conveys messages....
- notAlive, floorChannel, primalCh: No, these do not give enough, or requires massive amounts of context to be understood.
- lightEventChan: probably good
- to_main, from_main: Ok if the 'main' functionality is well-established.

5-5) Suggest a general rule for how channels should be named.

Expectations: Sverre tends to naming channels by describing the elements carried on the channels, but other parts of the channels context is as relevant sometimes.

That is - there is no definitive answer. Any mature answer receives credit...

5-6) Sverre has an issue with nested ′while(true)′ loops (infinite ′for′ loops, for the go programmers out there :-)), claiming that it is bad form to nest infinite loops.

Either: Explain why nested while(true) loops are bad.

Or: Give an argument or an example why nested infinite loops are sometimes necessary.

Expectations: What you see is not what you get when a loop gets hijacked by an inner loop (infinite, blocking, spinning, polling...).

6 Real-Time Programming

6-1) (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 work.

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.

6-2) In task 1-6 we assume a software system where "making a system that gurarantees that all deadlines are met is seen as infeasible/ too conservative". What are the (main) reasons that this is infeasible or too conservative for some systems?

Expectations:

- The theoretical WCET is often enourmously large due to
  - HW effects (are we assuming cache misses for every memory access?)
  - Dynamism in the algorithms that are run.
- The interactions between the threads in the system, may be so complicated that they are not easily taken into the proof.

Remember; timing analysis necessarily includes deadlock analysis and is of course a global analysis with the consequences this have for scalability.

All mature arguments should give credit.

6-3) Why do you think schedulability proofs are not so well developed for message passing systems (like a go program preferring channel communication over shared resources)?

Expectations:

Sverre thinks that schedulability proofs well could have been better developed than they are... That is, I am not certain how good the argument "it is too hard"is...

Sverres answer:

- messages and channels are most often discussed, not as an abstraction in a predictable task switching scheme, but in the context of communication networks, where, often, probabilistic models or throughput/latency"is the way of modelling. This does not fit well with hard deadlines.
- Programs written with lightwheight threads and using communication mostly because its abstraction of thread interaction (and not that we are "communicating over a distance") also tend to divide the system into more threads than the "one per timing demand". This breaks the priority"concept as we use it. What it means/should mean to build a predictable scheduler for one million lightwheight threads (like one per pixel in an image) is not so easy to define.
- RT programmers have traditionally been used to working "close to the HW". That is, able to think e.g. about memory as a shared resource. The abstraction that a messagepassing system is, might be unfamiliar or unwanted.

Any mature answer should yield credit.

6-4) 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.