NTNU
Norges teknisk-vitenskapelige universitet

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi,
matematikk og elektroteknikk

Bokmål & Engelsk

Eksamen
i
TTK4145
Sanntidsprogrammering
16. August 2006
9.00 – 13.00

Sensuren vil bli avsluttet i henhold til gjeldende regelverk.

Oppgavesettet inneholder en engelsk og en norsk versjon. Den norske versjonen finnes fra side 2 til og med side 5. Du kan velge om du vil besvare eksamen på norsk eller engelsk.

This exam contains two versions, one in norwegian and one in english. The english version is found on pages 6 through 9.

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».

Generally:
Be short and clear. Enumerate items and arguments, use figures where appropriate. You may be punished for text without substance.

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

Faglig kontakt under eksamen:
Sverre Hendseth
Telefon 98443807

Oppgave 1: Forskjellig

Oppgave 1a) Forklar begrepene vranglås og utsulting, gi et eksempel på hver.

Oppgave 1b) Fire kriterier må alle være oppfyllt for at en vranglås skal kunne forekomme. Hvilke ?

Oppgave 1c) "Resumption"- og "Termination"- begrepene kom inn som motsetninger i pensum både under diskusjon av exceptions og asynch. notification. Forklar kort forskjellen. Er exceptions resumption eller termination ? Hva med POSIX signaler ? Hva med ADA's select-then-abort statement ?

Oppgave 1d) Hva er en "race condision" ? Gi et eksempel på en slik.

Oppgave 1e) Implementasjon/Pseudokode: Du skal beskytte en resurs med en lese-skrive lås hvor du kan tillate aksess av mange lesere samtidig, men bare en skriver, og hvor skriverne har prioritet over leserne. Implementer funksjonene startRead, stopRead, startWrite og stopWrite. Ta utgangspunkt i hvilke synkroniseringsmekanismer du vil.

Oppgave 2: Delt variabel-basert synkronisering

Oppgave 2a) På den ene side kan du beskytte en modul med en semafor, hvor alle funksjoner reserverer semaforen mens de kjører, og på den andre siden beskytte den med en monitor. Hva er fordelene med det siste ?

Oppgave 2b) Eksemplet i kap 8.4.6 (side 243) i Burns & Wellings (vedlagt) har altså en bug. Hva er problemet ?

Oppgave 2c) Disse funksjonene i oppgave 2b skulle ikke være mye mer komplisert eller større før det blir urealistisk å finne bug'en ved å stirre på koden. Feilen er også av den typen som kan opptre skjelden, og bare under bestemte omstendigheter, slik at det ikke er sikkert at du finner den ved testing. Hva kan du gjøre for å sikre deg at slike feil ikke opptrer i kode du skriver ?

Oppgave 2d) (Implementasjon/Pseudokode) Hvordan ville du implementert denne funksjonaliteten i oppgave 2b ? (Programmet tar altså henvendelser med 3 forskjellige prioriteter inn, og lar de med høyest prioritet få kjøre først.)

Oppgave 3: Feilhåndtering

Oppgave 3a) Forklar forskjellen på forover eller bakover feilhåndtering (forward or backward error recovery).

Oppgave 3b) I et regime med unntakshåndtering (exceptions) kan følgende kodebit gi oss problemer (forårsake vranglås):

```
{
        sem_wait(sem1);
        f();
        sem_signal(sem1);
}
```

Forklar hva problemet er og hvordan det kan løses.

Oppgave 3c) Det å håndtere alle feil som kan oppstå i et program kan komplisere koden vesentlig;

1. Hver feil er forskjellig og kan kreve en spesialisert håndtering.
2. "Prøv igjen" (maks N ganger) er av og til en løsning - dette innfører en løkke rundt koden og feilhåndteringen med "break" og "continue" i hytt og pine.
3. Vi trenger konvensjoner for å formidle feilen oppover i kallstakken - Noen feil må håndteres på høyere abstraksjonsnivå. (standardisert bruk av returverdi, standard status-parameter ?)
4. Bare noen av alle mulige feil er relevante i en gitt situasjon - det å avgjøre dette krever detaljert kjennskap både til feilmodiene til de kalte funksjonene og sammenhengen kallene blir gjort i.
5. Feilhåndteringslogikken blir blandet med den andre programlogikken, noe som gjør programmet vanskeligere å lese/vedlikeholde.

Hvilke av disse avhjelpes ved bruk av exceptions ?

Oppgave 3d) Prosesspar er et grunnleggende konsept for å oppnå høy tilgjengelighet i programvaresystemer vha. redundans. Forklar grunnprinsippene.

Oppgave 3e) Hvilket problem løses egentlig av Atomic Actions ?

Oppgave 3f) Forklar hvordan en Atomic Action virker / kan implementeres.

Oppgave 3g) Burns & Wellings anser "Atomic Actions" som mer egnet/tilpasset til sanntids applikasjoner enn Transaksjoner / "(Atomic) Transactions". Hvorfor ? Hva var egentlig forskjellen ?

Oppgave 3h) "Asynchronous transfer of control" eller "Asynchronous Notification" (Javas AsynchronouslyInterruptedException og Adas select then abort) blir ofte ansett som krevende teknikker å bruke riktig og sikkert. Hva består utfordringene i ?

Oppgave 3i) Det å skrive til en *logg* er et konsept som hjelper oss å finne frem (eller tilbake) til en konsistent tilstand. Forklar hvordan dette virker.

Oppgave 4: Meldingsbasert synkronisering

Oppgave 4a) Følgende OCCAM-program er en prosess som legger sammen to tall ("in1", "in2" og "out" er kanaler):

```
PROC buf (CHAN OF INT in1,in2,out)
  INT v1,v2:
  WHILE TRUE
    SEQ
      in1 ? v1
      in2 ? v2
      out ! v1+v2
:
```

Modeller denne prosessen i FSP (pseudokode); Modeller bare synkroniseringseffekten, ignorer mulige verdier av v1 og v2. (se evt. vedlagt referanse.) Hvor mange tilstander kan denne prosessen være i ?

Oppgave 4b) Gitt følgende FSP-modell:

```
A = (x -> y -> A).
B = (u -> B).
||AB = (A || B).
```

Tegn tilstandsdiagramet for `||AB`.

Oppgave 4c) Forklar begrepene vranglås/"deadlock" og "livelock" i kontekst av slike prosess-tilstandsdiagrammer som i oppgaven over.

Oppgave 5: Design

Du skal implementere en triviell regulator ("P-regulator"), men under ekstreme krav til tilgjengelighet og pålitelighet. Sjefen din har kjøpt inn maskinvare så det holder, bare for å få prosjektet fort i gang, men uten å ha noen plan for systemet - 3 sensorer, 3 regneenheter, doble nettverk, separate kraftforsyninger, et pådragsorgan som har intern feilhåndtering og N innganger hvor den sist utsatte verdien teller - og så vidre. Du kan kjøpe mer om du trenger, og du trenger ikke bruke alt om du ikke trenger det.

Oppdraget er å designe systemet slik at det er robust mot *alle* typer feil; Restart av regneenhet, regnefeil, kabelbrudd, kosmiske stråler som snur bits i minnet/registre osv. Du kan anta at bare en feil skjer samtidig.

Du skal gjøre grovdesignet av dette systemet; Du skal planlegge hvordan det skal gjennomføres/ implementeres. Gjør de antagelsene du trenger.

"Grovdesign" innebærer her at du skal overbevise om at du er på riktig spor når det gjelder designet, dvs. at du skal ta de design-avgjørelsene som har størst konsekvenser for resten av designet og de "vanskelige" design-avgjørelsene, hvor det ikke er åpenbart om det finnes en løsning eller hva som er løsningen.

Vær oppmerksom på at presentasjonen i seg selv er en viktig del av svaret på denne oppgaven.

Noen velmente hint:

- Ta eksplisitt stilling til hvilke problemstillinger som du sikter på å løse i dette grovdesignet. Ser du flere "grovdesign" type problemstillinger enn du vil/rekker å svare på, så gjør et utvalg.
- Kladd designet ditt først – og så presenter det i besvarelsen. Ikke lever sider med "betraktninger underveis".

1: Misc

1a) Explain the terms deadlock and starvation, give one example of each.

1b) Four criteria must be filled for deadlock to occur. Which ?

1c) The terms “Resumption” and “Termination” is used as opposites when discussing both exceptions and asynch notification. Explain the difference. Are exceptions resumption or termination ? What about POSIX signals ? What about ADA's select-then-abort statement ?

1d) What is a race condision ? Give one example.

1e) Implementation/Pseudocode: You are to protect a resource with a read-write lock where more readers can access the resource at the same time, but only one writer. Writers have priority over readers. Implement the functions startRead, stopRead, startWrite and stopWrite. Base your implementation on the synchronization mechanism you want.

2: Shared Variable-based synchronization.

2a) One one hand you can protect a module with a semaphore, where all functions reserve the semaphore while they run, and on the other hand you can protect it with/as a monitor. Why is the second approach better ?

2b) The example in chapter 8.4.6 (page 243) in Burns & Wellings (attached) has a bug. What is the problem ?

2c) The functions in 2b can not be much more complicated for it to be unreasonable to detect the bug by inspecting the code. Such errors are also of the type that can occur seldomly, and only under special condisions, so we can not be certain to find them by testing. How can you make certain that such errors are not present in code you write ?

2d) Implementation/Pseudocode: How would you implement the functionality in 2b ? (The function takes requests with 3 different prioritites and lets those with highest priority run first.)

3: Error Handling

3a) Explain the difference between forward and backward error recovery.

3b) In a system where exception handling is used, the following code may cause a deadlock:

```
{
        sem_wait(sem1);
        f();
        sem_signal(sem1);
}
```

Explain what the problem is and how it is solved.

3c) Handling all possible errors in a program can complicate the code significantly;

1. We need code for handling different error situations.
2. We need a loop for "trying again" around both the code and the error handling.
3. We need conventions for returning errors up in the call hierarchy (Some errors must be handled on a higher "level of abstraction")
4. Only a subset of the possible errors may be relevant in a given situation – you need detailed knowledge of both the failure modes of the called functions and the calling context to decide which are relevant.
5. The code for error handling gets mixed with the "basic functionality" code making this difficult to read/maintain.

Which of these are helped/solved by exception handling ?

3d) Process pairs is a basic, redundancy-based, concept for achieving high availability in software systems. Explain the basic principles.

3e) Which problem is solved by Atomic Actions ?

3f) Explain how Atomic Actions works/can be implemented.

3g) Burns & Wellings regards Atomic Actions to be more suited for Real-Time applications than (Atomic) Transactions. Why ? What is the difference ?

3h) "Asynchronous transfer of control" or "Asynchronous Notification" (Javas AsynchronouslyInterruptedException and Adas select then abort) is often seen as difficult mechanisms to use correctly. What makes the challenge(s) ?

3i) Writing to a *log* is a concept that helps us get back (or "forward") to a consistent state. Explain how this works.

4:

4a) The following OCCAM program is a buffer process that stores one data element ("in1", "in2" and "out" is channels):

```
PROC buf (CHAN OF INT in1,in2,out)
  INT v1,v2:
  WHILE TRUE
    SEQ
      in ? v1
      in ? v2
      out ! v1+v2
:
```

Make an FSP (pseudo code) model of this program; Model only the synchronization, ignore the possible values of v1 and v2. Use the FSP reference on page 10 if necessary. How many states can the process have ?

4b) Given the following FSP model:

```
A = (x -> y -> A).
B = (u -> B).
||AB = (A || B).
```

Draw the state diagram for `||AB.` How many states is there ?

4c) Explain the terms deadlock and livelock in context of such process-state-machines as in 4b.

5: Design

You are to implement a trivial controller (e.g. a “P-controller”), but under extreme demands on availability and reliability. Your boss has bought a lot of hardware to get the project started, but without any plan for the system; 3 sensors, 3 CPU boards, double networks, separate power systems, an actuator that has its own error handling and N inputs where the last received value is the one that counts - etc. You can buy more if you need and need not use everyting...

You should design the system to be robust to *every* type of error; Restart of CPU unit, cable break, cosmic rays that switch bits in memory or registers, etc. Assume that only one error happens at a time.

You are to do the high-level design of this system; you should plan how it can be done/ implemented. Make assumptions as you see fit.

"High-level design" means that you should convince that you are on the right track of designing the system, You should decide on the "large" design decisions – that has consequences for large parts of the system -, and the difficult decisions – where it is not obvious that there is a solution, or what the solution is.

Be aware that *presentation* is an important part of your answer here!

Some hints:

- Be explicit on what design decisions you choose to include in your answer.
- Make a draft version of your design first – then *present it* in your answer. Do not turn in pages with "reflections along the way".

Appendix A
FSP Quick Reference

1. Processes

A process is defined by a one or more local processes separated by commas. The definition is terminated by a full stop. `STOP` and `ERROR` are primitive local processes.

Example

```
Process = (a -> Local),
Local = (b -> STOP).
```

Action Prefix **->**	If `x` is an action and `P` a process then `(x->P)` describes a process that initially engages in the action `x` and then behaves exactly as described by `P`.
Choice **|**	If `x` and `y` are actions then `(x->P|y->Q)` describes a process which initially engages in either of the actions `x` or `y`. After the first action has occurred, the subsequent behavior is described by `P` if the first action was `x` and `Q` if the first action was `y`.
Guarded Action **when**	The choice (**when** `B x -> P | y -> Q`) means that when the guard `B` is true then the actions `x` and `y` are both eligible to be chosen, otherwise if `B` is false then the action `x` cannot be chosen.
Alphabet Extension **+**	The alphabet of a process is the set of actions in which it can engage. `P + S` extends the alphabet of the process `P` with the actions in the set `S`.

Table A.1 - Process operators

2. Composite Processes

A composite process is the parallel composition of one or more processes. The definition of a composite process is preceded by **||**.

Example

```
||Composite = (P || Q).
```

Parallel Composition **||**	If P and Q are processes then (P||Q) represents the concurrent execution of P and Q.
Replicator **forall**	**forall** [i:1..N] P(i) is the parallel composition (P(1) || ... || P(N))
Process Labeling **:**	a:P prefixes each label in the alphabet of P with a.
Process Sharing **::**	$\{a_1,..,a_x\}$::P replaces every label n in the alphabet of P with the labels $a_1.n,...,a_x.n$. Further, every transition (n->Q) in the definition of P is replaced with the transitions $(\{a_1.n,...,a_x.n\}$->Q).
Priority High **<<**	||C =(P||Q)<<$\{a_1,...,a_n\}$ specifies a composition in which the actions $a_1,...,a_n$ have higher priority than any other action in the alphabet of P||Q including the silent action tau. In any choice in this system which has one or more of the actions $a_1,...,a_n$ labeling a transition, the transitions labeled with lower priority actions are discarded.
Priority Low **>>**	||C=(P||Q)>>$\{a_1,...,a_n\}$ specifies a composition in which the actions $a_1,...,a_n$ have lower priority than any other action in the alphabet of P||Q including the silent action tau. In any choice in this system which has one or more transitions not labeled by $a_1,...,a_n$, the transitions labeled by $a_1,...,a_n$ are discarded.

Table A.2 - Composite Process Operators

3. Common Operators

The operators in Table A.3 may be used in the definition of both processes and composite processes.

Conditional **if then else**	The process **if** B **then** P **else** Q behaves as the process P if the condition B is true otherwise it behaves as Q. If the **else** Q is omitted and B is false, then the process behaves as STOP.

Re-labeling **/**	Re-labeling is applied to a process to change the names of action labels. The general form of re-labeling is: /{*newlabel*_1/*oldlabel*_1,... *newlabel*_n/*oldlabel*_n}.
Hiding ****	When applied to a process `P`, the hiding operator `\{`$a_1..a_x$`}` removes the action names $a_1..a_x$ from the alphabet of `P` and makes these concealed actions "silent". These silent actions are labeled `tau`. Silent actions in different processes are not shared.
Interface **@**	When applied to a process `P`, the interface operator `@{`$a_1..a_x$`}` hides all actions in the alphabet of `P` not labeled in the set $a_1..a_x$.

Table A.3 - Common Process Operators

4. Properties

Safety **property**	A safety **property** `P` defines a deterministic process that asserts that any trace including actions in the alphabet of `P`, is accepted by `P`.
Progress **progress**	**progress** `P = {`$a_1,a_2..a_n$`}` defines a progress property `P` which asserts that in an infinite execution of a target system, at least one of the actions $a_1,a_2..a_n$ will be executed infinitely often.

Table A.4 - Safety and Progress Properties