NTNU
Norges teknisk-vitenskapelige universitet

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi,
matematikk og elektroteknikk

Bokmål & Engelsk

Eksamen
i
TTK4145
Sanntidsprogrammering
20. desember 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: Delt variabel-basert synkronisering

Oppgave 1a) Nevn noen grunnleggende (på mikroprosessornivå) mekanismer som kan hjelpe oss å oppnå synkronisering imellom to kjørende tråder. (Vi antar her at vi arbeider med vårt egenutviklede sanntids operativsystem).

Oppgave 1b) Hva er en *Race Condision* ? Gi et eksempel.

Oppgave 1c) Hva er en "guard" (slik vi kan ha dem f.eks. i et ADA entry eller i ADA/OCCAM's select/alt-konstruksjoner) ? Hva er forskjellen på en guard og en standard "if-test" ?

Oppgave 1d) Et av problemene med semaforer er dette med midlertidig frigivelse: I tråden

```
t(){
  wait(semA);
  wait(semB);
  // Gjør noe med A og B i kombinasjon
  ...
}
```

hvor semaforene beskytter resursene A og B, vil vi, hvis B er opptatt, kunne låse A selv om vi ikke får brukt den til noe, noe som kan gi lavere effektivitet og større sjanse for vranglåser enn nødvendig. Både POSIX, Java og Ada sine monitor-begreper tilbyr mekanismer som hjelper på dette; Hvilke ?

Oppgave 1e) Blooms kriterier lister en del problemstillinger innenfor resursallokering som vi kan bruke til å vurdere språkmekanismer for synkronisering opp mot hverandre. (Prioritering mhp. Type of Request,Order,Server State,Parameters,Priority) Adas gurardede entries kommer uheldig ut av det på et av disse. Hvilket og hvorfor ?

Oppgave 1f) Et generelt råd ved java-programmering er å unnå notify()-kallet til fordel for notifyAll() og while(...)-løkker rundt wait()-setningene. Hvorfor ?

Oppgave 1g) Et av de fire vilkårene for å kunne få en vranglås er "mutual exclusion" - det at vi har en form for resurser i systemet som må beskyttes fra samtidig aksess. Hvordan kan vi, i vranglåsunngåelsens navn, gjøre noe med dette ?

Oppgave 1h) (Implementasjon, pseudokode) En sensordriver tilbyr to funksjoner i grensesnittet sitt; config(...) og measure(...). Kall til config() skal ha prioritet over kall til measure() og du må anta flere tråder som kan kalle begge funksjonene. Implementer synkroniseringsdelen av disse to funksjonene. Ta utgangspunkt i hvilket språk/hvilke synkroniseringsmekanismer du vil.

Oppgave 2: Feilhåndtering & Konsistens

Oppgave 2a) Gi et eksempel på dynamisk redundans og et på statisk redundans.

Oppgave 2b) Gi et eksempel på backward error recovery, og et på forward error recovery.

Oppgave 2c) Et *recovery point* er sentralt ved backward error recovery. Forklar begrepet.

Oppgave 2d) Tankeløs generalisering av backward error recovery til flere deltagende tråder/prosesser kan lede til *domino effekten*. Forklar begrepet.

Oppgave 2e) En Atomic Action overholder gjerne *start*, *side* og *end boundaries*. For hver av disse tre; forklar hva de innebærer og angi en måte de kan implementeres på.

Oppgave 2f) Å teste på forskjellige feiltilstander for så å håndtere den eventuelle feilen er en temmelig standard måte å gjøre ting på, men med den ulempen at vi ikke fanger opp feil som vi ikke har forutsett. Hva kan vi gjøre for å detektere *uforutsette* feil ?

Oppgave 2g) Hva er forskjellen på en Atomic Action som beskrevet i Burns&Wellings og en (Atomisk) Transaksjon ?

Oppgave 2h) *To-fase commit* er en standard teknikk/mønster. Hvilke(t) problem(er) løser det og hvordan virker det ?

Oppgave 2i) En *(Trans)Action Manager* er et standard mønster/modul. Hvilke(t) problem(er) løser det og hvordan virker det ?

Oppgave 2j) I et stort, distribuert system har du basert designet ditt på "oppgaver" som implementeres som Atomic Actions. Etterhvert som antallet forskjellige oppgaver har økt, har du imidlertid blitt oppmerksom på at systemet ditt går i vranglås innimellom. List opp mulige strategier for å løse dette problemet.

Oppgave 3: Meldingsbasert synkronisering

Oppgave 3a) Hvorfor er ikke "i=i+1/i=i-1"-problemet (dvs. at to tråder/prosesser aksesserer en variabel samtidig, og overskriver hverandres resultater) relevant i OCCAM ?

Oppgave 3b) Følgende to tråder aksesserer to binære semaforer

```
t1(){
  wait(SemA);
  wait(SemB);
  // ...
  signal(SemA);
  signal(SemB);
}
t2(){
  wait(SemB);
  wait(SemA);
  // ...
  signal(SemB);
  signal(SemA);
}
```

Modeller trådene og semaforene i FSP (pseudokode er ok). Se evt. vedlagte FSP referanse.

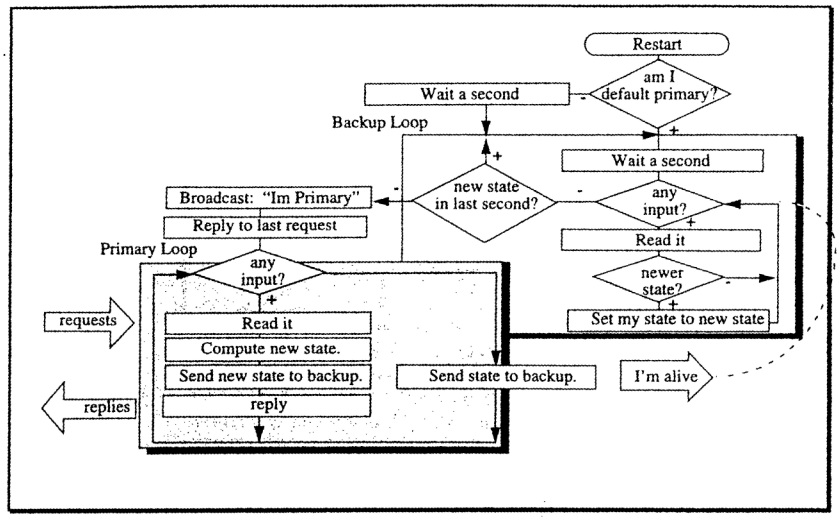
Oppgave 3c) Tegn transisjonsdiagrammer for (FSP) prosessene i 3b). Tegn også transisjonsiagrammet for totalsystemet.

Oppgave 3d) Definer begrepet *vrangsås* i kontekst av slike transisjonsdiagrammer.

Oppgave 3e) (Implementasjon, pseudokode) Synkron kommunikasjon kommer ut som ganske enkelt å modellere på denne måten, samtidig som kommunikasjon imellom to tråder er en abstraksjon som er ganske tett på hva vi ofte trenger synkronisering *til*. Lag et java objekt(/klasse) som danner en slik synkron kanal. Det antas brukt av to tråder hvor den ene sender og den andre mottar.

Oppgave 3f) Vi leter etter flere slike høynivå mekanismer som vil gi enkle FSP-modeller. En retning å gå vil være å tillate flere enn 2 deltagere i hendelsen/aksjonen. Dvs. at det er flere enn to tråder som utveksler informasjon eller "gjør noe sammen" i en slik synkron setting. Beskriv stikkordsmessig hvordan vi kan tenke oss noe slikt implementert.

Oppgave 4: Design



Figuren bør minne deg på hvordan prosess-par virker. Vi har sett for oss i denne standard-implementasjonen at oppdateringene som sendes fra Primær til Backup inneholder den komplette tilstanden til systemet. I denne oppgaven skal du imidlertid se på konsekvensene av å ønske å *ikke* sende hele tilstanden i hver melding.

Anta at tilstanden utgjøres av en diger tabell som er for stor til å sendes med hver melding. Hvordan vil du nå legge opp kommunikasjonen imellom Primær og Backup ?

Du skal gjøre grovdesignet av denne logikken; Du skal planlegge hvordan den skal gjennomføres/ implementeres i systemet. 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. Shared Variable-based synchronization.

1a) Mention some basic (microprocessor/assembly) mechanisms that let us achieve synchronization between two running threads. (Assume a kernel-writing perspective here.)

1b) What is a *Race Condision* ? Give an example.

1c) What is a *Guard* (like we have them in e.g. ADA's guarded entries, or ADA/OCCAM's select/alt statements) ? What is the difference between a guard and a plain "if-test" ?

1d) One of the problems with semaphores is that we cannot "temporarily" release them: In the thread:

```
t(){
  wait(semA);
  wait(semB);
  // Do something with A and B in combination
  ...
}
```

where the semaphores protect resources A and B, will, if B is occupied, A be reserved even if it cannot be used. This leads to inefficient execution and increased chance of deadlocks. The monitor-like mechanisms in POSIX, Java and ADA all tries to solve this... How ?

1e) Blooms Criteria lists a number of problems relating to recource allocation that we use for comparing languages and synchronization mechanisms. (Giving priority according to: Type of Request,Order,Server State,Parameters,Thread Priority) ADAs guarded entries fails for one of these. Which and Why ?

1f) 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 ?

1g) One of the four conditions necessary for getting a deadlock is "mutual exclusion" - that there is some resources in the system that cannot be accessed by more threads at the same time. In the context of avoiding deadlocks; How can we stop this condision from being true ?

1h) (Implementation, pseudocode) A sensor driver has two functions in its interface; config(...) and measure(...). Calls to config() should be given priority over calls to measure(), and you must assume more threads calling both functions. Implement the synchronization part of the two functions. Choose your language/synchronization mechanisms freely.

2. Error Handling.

2a) Give one example of dynamic redundancy and one example of static redundancy.

2b) Give one example of backward error recovery and one example of forward error recovery.

2c) A *recovery point* is important when doing backward error recovery. Explain the term.

2d) Thoughtless generalization of backward error recovery to more threads/processes may lead to the *domino effect*. Explain the term.

2e) An Atomic Action has *start, side* and *end boundaries.* For each of these three; explain what they aim to achieve and give one way they can be implemented.

2f) To test on different error condisions and then handle any detected errors is a common way of handling errors, but with the drawback that only anticipated errors can be handled. What can we do to help us detect *unanticipated* errors ?

2g) What is the difference between an Atomic Action as described in Burns&Wellings and an (Atomic) TransAction ?

2h) *Two-phase commit* is a standard technique/pattern. Which problem(s) does it solve and how does it work ?

2i) A (T*rans)Action Manager* is a standard module/pattern. Which problem(s) does it solve and how does it work ?

2j) In a large, distributed, system you have based your design on "tasks" that are implemented as Atomic Actions. As the number of different actions have increased, you have, unforunately, discovered that your system deadlocks once in a while. List possible strategies to solve this problem.

3. Message-based synchronization.

3a) Why is not the "i=i+1/i=i-1"-problem - that two threads can access a variable at the same time and overvwite each others updates - relevant in OCCAM ?

3b) The two following threads access two binay semaphores:

```
t1(){
  wait(SemA);
  wait(SemB);
  // ...
  signal(SemA);
  signal(SemB);
}
t2(){
  wait(SemB);
  wait(SemA);
  // ...
  signal(SemB);
  signal(SemA);
}
```

Model the threads and the semaphores in FSP (pseudocode is ok). See the FSP reference in appendix.

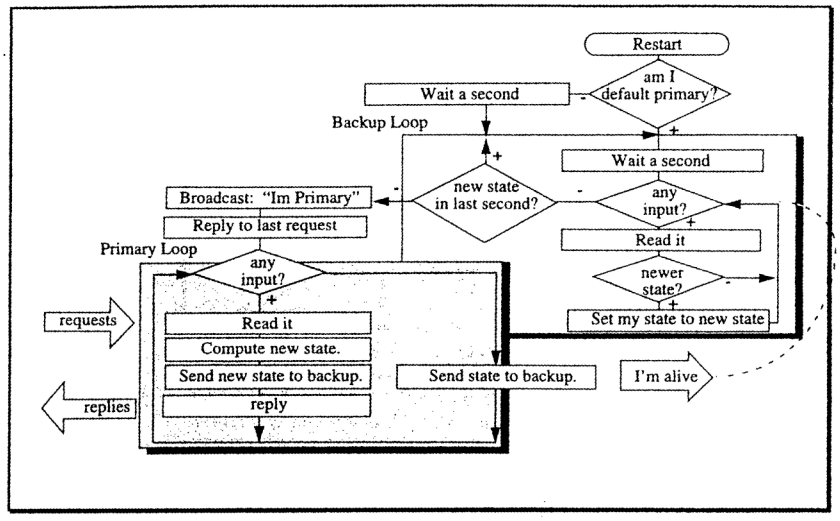
3c) Draw transition diagrams for the processes in 3b). Draw also the transition diagram for the complete, combined system.

3d) Define the term *deadlock* in context of such transition diagrams.

3e) (Implementation, pseudocode) Synchronous communication makes for simple FSP models while also corresponding well to the underlying need for thread synchronization. Make a Java object(/class) that implements such a synchronous channel. It is assumed used by two threads where one sends and the other receives.

3f) We are searching for more highlevel mechanisms that yields simple FSP models, and one reasonable extension might be to allow more than two participants in the event/action. That is, more than two threads exchanges information or "does something together" in a synchronous setting. Describe shortly (keywords only) how you can imagine such a mechanism implemented.

4. Design.



The figure should remind you of how process pairs works. In our discussions of process pairs so far we have assumed that the complete system state is sent from the primary to the backup as part of every IAmAlive message. Here, you shall explore the consequences of *not* sending the complete state in every message.

Assume that the state consist of one table too large to be sent as part of every message. How would you now make the communication between the primary and the backup ?

You are to do the high-level design of this protocol/functionality; 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 in designing the system. You should decide on the "large" design decisions – that has consequences for large parts of the system, and the difficult ones – where it is not obvious whether 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. If you see more high-level design decitions than you want/choose to answer, make a selection.
- 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