NTNU
Norges teknisk-vitenskapelige universitet

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi,
matematikk og elektroteknikk

Bokmål & Engelsk

Eksamen
i
TTK4145
Sanntidsprogrammering
7. desember 2005
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 maskinvare(/maskinkode) mekanismer som det er vanlig å bruke for å oppnå grunnleggende synkronisering. (Boken nevner to...)

Oppgave 1b) Semaformekanismen blir kritisert for å ikke "være skalerbar" på den måten at store systemer basert på semaforer lett blir vanskelig å vedlikeholde. Hvorfor ?

Oppgave 1c) Monitorer (Ada protected objects og Java synchronized methods) blir også kritisert på samme måte. Hvorfor ?

Oppgave 1d) Nevn metoder for vranglåsunngåelse: Hvordan kan du sikre deg at det ikke kan intreffe vranglåser i systemet ditt ?

Oppgave 1e) Tankeløs bruk av Suspend og Resume kan lett føre til "race condisions". Forklar.

Oppgave 1f) Nevn ting vi kan ønske oss fra en resursmanager i tillegg til den rene gjensidige utelukkelsen. (Blooms Criteria lister en del slike...)

Oppgave 1g) (Implementasjon/pseudokode) Du skal skrive en resursmanager som forvalter et stykke minne med begrenset størrelse. Kallet til "void * Allocate(int size)" tar mengden minne som parameter inn og returnerer en peker til et minneområde. "Free(void *)" tar en slik peker til et minneområde og frigir minnet igjen. For å unngå utsulting ønsker du å gi prioritet til de største henvendelsene.

Fokus er på synkroniseringen; Gjør de forutsetningene du vil når det gjelder grensesnittet til selve bokføringen omkring ledig/opptatt minne og evt. datastrukturer du vil bruke.

Baser pseudokoden på hvilke synkroniseringsmekanismer du vil (POSIX/Java/Ada...).

Oppgave 2: Feilhåndtering

Oppgave 2a) List opp ulemper med N-versjonsprogrammering.

Oppgave 2b) Når feil oppstår i systemer med flere samarbeidende tråder/prosesser risikerer vi at flere tråder også må samarbeide om feilhåndteringen. Forklar "dominoeffekten" i denne sammenhengen.

Oppgave 2c) 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 2d) For å legge tilrette for feilhåndtering i systemer med flere tråder/prosesser kan vi bruke "Atomic Actions" eller transaksjoner. Forklar begrepene "start boundary", "side boundary" og "end boundary" og hvordan vi programmeringsteknisk kan sikre oss disse grensene.

Oppgave 2e) En "Action Controller" eller en "Transaction Manager" er en mer eller mindre gjenbrukbar modul for å støtte opp programmering av Atomic Actions/transaksjoner. Hva slags logikk inngår i en slik ?

Oppgave 2f) 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 2g) Prosesspar er et grunnleggende konsept for å oppnå høy tilgjengelighet i programvaresystemer vha. redundans. Forklar grunnprinsippene.

Oppgave 2h) List opp (programvaremessige) virkemidler for å lage pålitelig kommunikasjon ut av upålitelig (hvor meldinger kan mistes, komme bare delvis frem, komme frem i feil rekkefølge, komme frem til feil mottager, ha bit-feil...).

Oppgave 2i) "Asynchronous transfer of control" og en del tilsvarende begreper kom opp i læreboken i samme kapittel som Atomic Actions... Hvorfor ? Hva er koblingen imellom Atomic Actions og Asynchronous transfer of control ?

Oppgave 2j) Hva kan grunnene være til å skulle foretrekke Asynchronous Notification (Javas AsynchronouslyInterruptedException og Adas select then abort) fremfor det å sende meldinger imellom trådene eller polle på statusvariable ?

Oppgave 3: Meldingsbasert synkronisering

Oppgave 3a) Følgende OCCAM-program er en buffer-prosess som kan lagre et data-element ("in" og "out" er kanaler):

```
PROC buf (CHAN OF INT in,out)
  INT v:
  WHILE TRUE
    SEQ
      in ? v
      out ! v
:
```

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

Oppgave 3b) Gitt følgende FSP-modell:

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

Tegn tilstandsdiagramet for `||AB`.

Oppgave 3c) (Implementasjon/pseudokode) Fasinert som du er av design basert på fingranulære prosesser og synkron ubufret kommunikasjon ønsker du å bruke synkron kommunikasjon selv om du skal implementere i C (/et C-lignende språk) med semaforer som synkroniseringsmekanisme. Implementer "send(int value)" og "receive(int & value)" kallene imellom to tråder slik at mottakstråden, selvfølgelig, må vente til sendertråden har sendt og sender-tråden vil vente til meldingen er lest før de går vidre. Bruk semaforer som synkroniseringsmekansime. Pseudokode er ok.

Oppgave 3d) Disse fingranulære prosessene er også attraktive, siden du har gjort slike prosesser til modulbegrepet i designet ditt. Kan du foreslå, stikkordsmessig, hvordan en kan tenke seg å implementere et slikt "prosessbegrep" i C ?

Hint: Tenk deg, om du vil, at du har et OCCAM-program (altså med PAR, ALT etc.) du vil oversette på direkte og systematisk vis til C. Å bruke POSIX tråder er ikke løsningen... (Hvordan ville du *simulert* dining philosohers-problemstillingen i C uten å bruke tråder ?)

Oppgave 4: Design

I et prosessstyresystem ønsker vi å kunne bytte imellom to operasjonsmodi; En med veldig avanserte, koblede, CPU-krevende og fint tunede regulatorer og en med veldig konservativt tunede, sikre, enkle regulatorer. Det er alltid greit/trygt å skifte fra den første til den andre, men det å skifte den andre veien krever inntuning, initialisering etc. som går over en tidsperiode på hundrevis av samples (mens regulatorene fortsatt går i den sikre modusen). Det er her kritisk at alle trådene har fått gjort seg ferdig med initialiseringen sin og at alle regulatorene har et felles bilde av *når* det endelige modus-skiftet skjer. Hele modusendringen kan måtte avbrytes dersom f.eks. en av tilstandene kommer til ytterkant av arbeidsområdet i løpet av perioden, eller om feilsituasjoner detekteres i en av regulatorene.

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 hardware (/assembly) mechanisms that are used for achieving basic synchronization. (Burns & Wellings mentions two...)

1b) The semaphore mechanism is criticized for not "being scalable" in that large systems built on semaphores often gets difficult to maintain. Why ?

1c) Monitors (Ada protected objects and Java synchronized methods) is criticized for the same thing. Why ?

1d) Mention some methods for deadlock prevention/avoidance: How can you ensure that deadlocks does not occur in your system ?

1e) Thoughtless use of suspend and resume may easily lead to "race conditions". Explain.

1f) List some things we may want from a resource manager in addition to mutual exclusion. (Blooms Criteria lists some of these)

1g) (Implementation/pseudo code) You are to write a resource manager that manages a section of memory with limited size. The call to "void * Allocate(int size)" takes the size of memory to be allocated as a parameter, and returns a pointer to the allocated memory area. "Free(void *)" takes a pointer to such a memory area and frees the memory again. To avoid possible starvation you want to give priority to the largest requests.

Keep focus on the synchronization; Make the assumptions you need regarding the bookkeeping of free/allocated memory, and any data structures you want to use.

Base your pseudo code on the synchronization mechanisms you want (POSIX, Java, Ada...).

2. Error Handling.

2a) List drawbacks/problems with N-version programming.

2b) When errors occur in systems with more cooperating threads/processes there is a chance that more threads must cooperate on the recovery also. Explain "the domino effect" in this context.

2c) 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.

2d) To prepare for handling errors in systems with more threads/processes we may use "Atomic Actions" or transactions. Explain the terms "start boundary", "side boundary" and "end boundary" and how these boundaries can be achieved.

2e) An "Action Controller" or a "Transaction Manager" is a more or less reuseable module that supports programming of Atomic Actions/transactions. Which features/logic does this usually have.

2f) Writing to a *log* is a concept that helps us get back (or "forward") to a consistent state. Explain how this works.

2g) Process pairs is a basic, redundancy-based, concept for achieving high availability in software systems. Explain the basic principles.

2h) List software techniques we can use for making reliable communication from unreliable communication (where messages get lost, be received only partially, be received in the wrong order, be received by the wrong receiver, have data errors...).

2i) "Asynchronous Transfer of Control" and some corresponding terms is discussed in Burns&Wellings in the same chapter as Atomic Actions. Why ? What is the connection between Atomic Actions and Asynchronous Transfer of Control ?

2j) What can be the reasons for using Asynchronous Transfer of Control (Javas AsynchronouslyInterruptedException og Adas select then abort) rather than sending messages between the threads or polling status variables ?

3. Message-based synchronization.

3a) The following OCCAM program is a buffer process that stores one data element ("in" and "out" is channels):

```
PROC buf (CHAN OF INT in,out)
  INT v:
  WHILE TRUE
    SEQ
      in ? v
      out ! v
:
```

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

3b) Given the following FSP model:

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

Draw the state diagram for `||AB.`

3c) (Implementation/pseudo code) Fascinated as you are by the thought of basing your design on fine-granularity processes and synchronous unbuffered communication, you want to use synchronous unbuffered communication also in C (or a similar language) and with semaphores as mechanism for synchronization. Implement "send(int value)" and "receive(int & value)" to be used by two different threads in a way that the sender must wait until the receiver has received the message before continuing, and the receiver must (obviously) wait for the sender. Use semaphores as mechanism for synchronization.

3d) These fine-granularity processes are also attractive, since you have based your design on these. Suggest (just keywords) how such "processes" can be implemented in C ?

Hint: Assume if you want, that you are searching for a systematic way of translating an OCCAM (using PAR, ALT etc.) program to C. Using (POSIX) threads is not the solution. (How would you have *simulated* the dining philosophers in C – without using threads ?)

4. Design.

In a process control system we want to be able to switch between two modes of operation: Mode one has very advanced, coupled, CPU-demanding, optimally tuned controllers, and mode two has simple, safely tuned, robust controllers. It is always easy/safe to switch from mode one to two, but from two to one the switch demands an initialization phase that takes hundreds of samples (while the mode two controllers still does the work). It is critical here that all threads that need such initialization gets to finish successfully before the switch, and that all threads switch at the same time. The switch may be called off if, for example, one of the (controller) states reaches an unsafe boundary, or if some error situation is detected.

You are to do the high-level design of this switching logic; 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