NTNU
Norges teknisk-vitenskapelige universitet

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi,
matematikk og elektroteknikk

Bokmål

Eksamen
i
TTK4145
Sanntidsprogrammering
7. desember 2007
9.00 – 13.00

Sensuren vil bli avsluttet i henhold til gjeldende regelverk.

Du kan velge om du vil besvare eksamen på norsk eller engelsk.

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».

Alle deloppgaver teller likt unntatt implementasjonsoppgavene som teller dobbelt.

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

Faglig kontakt under eksamen:
Sverre Hendseth
Telefon 98443807

Oppgave 1: Feilhåndtering

«Standard» feilhåndtering er det å teste på forskjellige feiltilstander for så å ha kode for å håndtere de eventuelle feilene. Feilhåndteringsdelen av pensum er imidlertid motivert av at dette ikke er godt nok:

1. Vi har i et sanntidssystem sterkere krav til systemet og må også håndtere *uforutsette* feil.
2. Vi har også som regel flere tråder/prosesser som samarbeider om oppgavene. Av og til må disse også samarbeide om feilhåndteringen.

Oppgave 1a) Hva kan vi gjøre for å detektere *uforutsette* feil ?

«Recovery punkter» kan være nyttige ved «backward error recovery», men tankeløs bruk av dem i systemer med flere tråder/prosesser kan føre til dominoeffekten.

Oppgave 1b) Forklar forskjellen på backward og forward error recovery ?

Oppgave 1c) Hva er et recovery punkt ?

Oppgave 1d) Forklar domino effekten.

Oppgave 1e) Hvordan kan vi unngå domino-effekten ?

Det å skrive til «log» er en alternativ teknikk til å bruke recovery-punkter.

Oppgave 1f) Hvordan virker dette ?

Oppgave 1g) Hvordan kan vi sikre oss at loggen ikke vokser og blir vilkårlig stor ?

Hvis vi generaliserer *forward* error recovery til flere tråder får vi et behov for å formidle informasjon om hva som gikk galt imellom tråder.

Oppgave 1h) Meldingssending og polling på (globale) feilstatusvariable er to måter: Nevn svakheter ved disse.

ADA og Real-Time Java har språkmekanismer for «Asynchroneous Transfer of Control» (ATC).

Oppgave 1i) Hva ligger i begrepet ATC ?

Oppgave 1j) Hvilke er disse språkmekanismene i hhv. ADA og Java ?

Oppgave 1k) Hvordan kan en slik mekanisme brukes til formidling av feilinformasjon imellom tråder ? (velg en av dem, hvis du vil være konkret) .

Oppgave 1l) Hvordan kan vi skaffe oss ATC i C/POSIX ?

Oppgave 1m) «Resumption»- og «Termination»-begrepene er brukt for å klassifisere både exceptions, signaler, Asynchroneous Notification og ATC. Forklar kort begrepene. Hvorfor anser vi Termination-modellen som mer anvendelig enn Resumption ?

Oppgave 1n) «Sammenslåing av failure modes» er en teknikk. Hva er en feilmodus og hva oppnår vi med slå dem sammen ? Gi et eksempel på et system hvor det kan være hensiktsmessig å slå dem sammen.

Oppgave 1o) Prosesspar er et grunnleggende konsept for å oppnå høy tilgjengelighet i programvaresystemer vha. redundans. Forklar kort grunnprinsippene.

Oppgave 1p) Atomic Actions er et rammeverk for å oppnå noe på feilhåndteringssiden. Nøyaktig hva er det Atomic Actions legger til rette for ?

Oppgave 1q) Databasefolk er lykkelige med ABORT/backward error recovery som standard feilhåndtering innenfor slike (trans-) aksjoner. Hvorfor er dette mindre kurant for «oss sanntidsfolk» ?

Oppgave 2: Låsing av resurser

Du har av forskjellige årsaker bestemt deg for å lage en "lock manager" i systemet ditt.

En av de tingene du ønsker deg, er at alle prosesser kan allokere alle nødvendige låser i en atomisk operasjon. Dvs: Enten får prosessen låst alle resursene den ber om, eller så blir den blokkert til den kan få dem.

Oppgave 2a) Hvordan vil du argumentere for at systemet ditt nå vil bli fritt for vranglåser ?

Oppgave 2b) Er det nødvendig at *opplåsing* også skjer i en atomisk operasjon for å unngå vranglåser, eller kan opplåsing skje for resursene enkeltvis ?

Oppgave 2c) Hva er ulempene med denne måten (altså «alle eller ingen») å organisere låsing av resurser på ?

Oppgave 2d) (**Implementasjon**) Bestem grensesnittet og implementer de funksjonene du trenger for denne funksjonaliteten. Baser det på hvilke språk/synkroniseringsmekanismer du vil.

Oppgave 2e) Hvilke andre grunner kan du ha til å ønske en lock manager i systemet ditt ? / Hvilke andre nyttige egenskaper kan en legge inn i en slik ?

I et styresystem bruker vi en sensor som også med jevne mellomrom rekalibreres (av en annen tråd). Dette er en tidkrevende prosess og vi må heller bruke den gamle måleverdien enn å vente på at kalibreringen skal bli ferdig.

```
if(readSem(S) != 0){ // Sensor available
  wait(S)
  // Read sensor value
  signal(S)
}else{
  // Use old value
}
```

readSem(...) er et ikke-blokkerende kall som bare returnerer status på semaforen.

Oppgave 2f) Hvorfor vil ikke denne koden virke ?

Oppgave 2g) Skisser en bedre løsning.

Oppgave 2h) I et ADA protected object kan du ikke bli blokkert fra innsiden av et kall i objektet. Nevn fordeler og ulemper med dette. Hvilken språkmekanisme kan brukes hvis behovet for å vente fra innsiden av et objekt oppstår ?

Oppgave 2i) En kritikk av monitorer går på at hvis en ifra en monitor kaller en annen, og tråden blir blokkert i den indre monitoren, så blir den ytre monitoren *ikke* frigitt. Hva er problemet(-ene) med dette?

Oppgave 2j) Sammenlign kort synkroniseringsmekanismene ADAs guardede entries og Javas synchronized objects (fordeler og ulemper).

Oppgave 3: Misc

Oppgave 3a) Guard-mekanismer kommer inn både i ADAs entries og OCCAMs ALT. Forklar hvordan en guard virker.

Oppgave 3b) (**Implementasjon**) Du skal implementere, ved hjelp av semaforer, en «barriere» - et punkt i flere tråders (n tråder, hvor n er kjent) utførelse hvor ingen skal fortsette før alle er kommet frem til dette synkroniseringspunktet. (Et multitråd CSP/FSP event om du vil). Kodebiten du skriver skal kalles fra alle trådene på det punktet de ønsker å synkronisere seg.

Oppgave 4:

Følgende er kode for å synkronisere et bounded buffer med semaforer:

```
thread_put(e){
  while(1){
    wait(NFree);
    wait(Mutex);
      // enter e into buffer
    signal(Mutex);
    signal(NInBuffer);
  }
}
```

```
thread_get(e){
  while(1){
    wait(NInBuffer);
    wait(Mutex);
      // get e from buffer
    signal(Mutex);
    signal(NFree);
  }
}
```

Oppgave 4a) Modeller en av de to trådene og en av de tellende semaforene i FSP. Bufferstørrelsen er kjent, anta max 2 elementer i bufferen hvis du trenger. Vær nøye med event-navnene.

Oppgave 4b) Tegn transisjonsdiagrammene til de tilsvarende prosessene.

Oppgave 4c) Anta bare ett element i bufferet og tegn transisjonsdiagrammet for totalprosessen.

Oppgave 4d) Hvordan kan du ut ifra et slikt diagram argumentere for et fravær av vranglåser i systemet ? Hva med fravær av livelocks ?

Oppgave 4e) (**Implementasjon**) I koden over er bufferet et «passivt» objekt beskyttet av semaforer. En alternativ implementasjon til å ville vært å gjøre bufferet til en «aktiv» enhet: Selve datastrukturen blir en lokal variabel i en serverprosess som håndterer put og get-meldinger. Ved å bruke synkron kommunikasjon og la serveren differensiere på hvilke meldingstyper/kanaler den er villig til å lese fra oppnår vi også blokkeringen ved full og tom buffer. Skisser denne serverprosessen i OCCAM (pseudokode).

Oppgave 4f) Hvor mange tilstander vil denne prosessen (altså den fra oppgave 4e) kunne være i hvis bufferstørrelsen er 1 ?

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