Sensorveiledning eksamen ttk4145 – Sanntidsprogrammering h05.

Dette er ikke ment som en «ideal-besvarelse», men heller som en meta-nivå beskrivelse av hva som forventes evt. i hvilken retning en løsning kan finnes. Det kan godt være at det finnes fullgode svar på enkelte oppgaver utenom det som er antydet her.

Oppgave 1a) – Maskinvare mekanismer

De to boken nevner er «disable interrupt» og «Test&Set/Swap assembly instruksjoner».

Oppgave 1b) – Semaforer

Dette bryter med prinsippet om at en «modul» skal kunne vedlikeholdes uten å kjenne hele systemet. Semaforer har preg av globale variable som kan aksesseres fra alle steder i programmet; for at kode som bruker semaforer skal kunne vedlikeholdes, må en kjenne/ta hensyn til alle andre steder som bruker semaforen.

Et greit poeng, men som ikke gir full kredit alene kan være om at vranglåser oppstår lett når en har mange semaforer.

Oppgave 1c) – Monitorer

For monitorer er modul-begrepet ryddet opp, så her ligger svakheten i at moduler ikke kan bygges på moduler (composition...); Monitorer som kaller andre monitorer er uggent.

Mindre punkter sim ikke gir full kredit alene er: Boken nevner også at konseptuelt så er mutual exclusion-delen og condision variable (wait/notify) delen på forskjellige abstraksjonsnivå et problem. Arving i java er komplisert ved synchronized-objekter.

Oppgave 1d) – Vranglåsunngåelse

Dette inneholder både «prevention» og «avoidance» i boken:

4 forutsetninger for vranglås; hvis en ikke er tilstede kan vi ikke få vranglås.

- Mutual exclusion (ikke gjør variable beskyttet som ikke trenger det)
- Hold&Wait: Hvis det ikke er mulig å blokke inne i beskyttede regioner kan ikke vranglåser oppstå -> Resever alle nødvendige resurser i en udelelig operasjon.
- Circular Wait: Det må være en sirkulær avhengighet tilstede for vranglås -> Alle kan reservere resurser i samme rekkefølge.
- No preemption: Hvis prosesser kan bli fratatt resurser (ved å abortere, kanskje) kan vi unngå vranglåser

I tillegg:

- Formell verifisering
- Algoritmer for resursallokering som analyserer situasjonen
- Schedulingsalgoritmer (Priority Ceiling).
- (Strukturert og oversiktlig programmering reduserer sjansen :-).

Oppgave 1e) – Race conditions ved suspend/resume

```
t1(){
  if(!condition)
    suspend();
}
t2(){
  condition = true;
  resume(t1);
}
```

Race conditions oppstår her ved at t1 avbrytes etter å ha gjort testen, men før suspend kalles. t2's resume har da ingen effekt, og t1 kan bli hengende på suspend'en uendelig. Det er mange race condisions av denne typen her og generelt har vel problemet med at «condition»-variabelen ikke er knyttet direkte til synkroniseringen.

Oppgave 1f) – Blooms Criteria

Referert rett fra boken:

- Type of Request
- Order of requests
- Parameters of request
- State of server
- Priority of client

Siden Blooms criteria er nevnt som hint må disse gi full score, men følgende kan gi kredit også:

- Bokføring av hvem som eier hva.
- Feil-recovery (f.eks. løse opp ressurser hvis eieren dør)
- Vranglåsunngåelsesalgoritmer.
- ...

Oppgave 1g) – Resource manager implementasjon

Det er noen eksempler på dette i boken som ikke er særlig enkle... Vanskeligheten dannes av at boken insisterer på at scheduleren skal holde rede på de ventende trådene, og ikke vil gjøre det selv (ref. min lenkede liste). Boken **nevner** imidlertid muligheten for å transformere andre resursmanagere over til «server state» resursmanagere, jeg har fremhevet det på forelesningen og det faller ganske likt ut som løsningsforslaget til eksamen i fjor... (Når det er sagt er det en bug i det fra ifjor :-)

Jeg forutsetter listen og sorteringen (list_insertSorted,list_getFirstId og list_removeEntry) og grensesnitt til mem_alloc og mem_free. Antar vidre java-lignende monitorer.

```
// ---- Monitor_RM
void *
allocate(int threadId,int size){
  list_insertSorted(threadId,size);
```

```
  while(true){
    // Repeat until we get the memory we want.
    if(list_getFirstId() != threadId){
      // I am not first, wait
      wait();
    }else{
      // I am first in line, give it a try.
      void * p = mem_alloc(size);
      if(p == 0){
        // Not enough memory available, wait
        wait();
      }else{
        // OK – got it, happily leave the queue.
        list_removeEntry(threadId);
        // When I leave the queue, give the next one a try
        notifyAll();
        return p;
      }
    }
  }// End while
}

free(void * p){
  mem_free(p);
  notifyAll();
}
```

Oppgave 2a) – N-versjonsprog:

Det er mye å gripe fatt i her; The Voting Problem er viktig, og det at en ikke kan sikre seg uavhengige implementasjoner. Pris selvfølgelig. Feil i driveren (som ikke er replikert). Større total system gir vedlikeholdbarehets byrde og feilrate som er mer enn proposjonal med størrelsen.

Oppgave 2b) – Domino effekten

Her dukker vel en og annen figur opp, tenker jeg, som tar utgangspunkt i at trådene har etablert sine egne sekvenser av «recovery points» og illustrerer at siden konsekvenser av mulige feil tillates å kommuniseres imellom trådene, kan vi mislykkes i å sette klare grenser for når feilen tidligst kan ha skjedd. (Dette er motivasjon for at tråder må koordinere recovery-punktene sine.)

Oppgave 2c) – Exceptions

Problemet som kan oppstå er at f() kaster en exception og at exceptionen bobler opp gjennom den viste blokken for så å håndteres på et nivå over. Resultatet av dette er at semaforen blir reservert og ikke frigitt.

For å fikse dette må en håndtere og re-raise exceptioner («last whishes» er begrepet boken bruker på dette og Java og Ada har språkmekanismer for det.).

Oppgave 2d) – Boundaries

Start boundary er den fremforhandlingen av felles recoverypunkter som ble diskutert i oppgave 2b. Implementasjonene i boken har preg av en «påmelding» til (trans)aksjonen. Den må ikke nødvendigvis synkroniseres i tid, men det må sikres at påmeldte tråder ikke kommuniserer med ikke påmeldte og at alle som skal være med er registrert før vi avslutter. Verdien av alle relevante variable ved dette startpunktet må lagres unna (Dvs. hvis vi ikke logger) hvis det er aktuelt å håndtere feil ved «backward recovery» (og det er det jo).

Side boundary er å sikre at deltagere i aksjonen ikke kommuniserer (direkte eller indirekte) med andre enn hverandre. To hovedmomenter: Det å ikke sende meldinger el.l. til andre enn deltagere i aksjonen har ikke stort fokus i boken. Her antas det at vi kjenner deltagerene/rollene i en gitt aksjon ved programmeringstid og at det dermed er trivielt. Alternativet er å legge transaksjonsId ved alle meldinger, og så la deltagerene melde seg på den gitte transaksjonen etter behov. Det andre momentet som det derimot gjøres et stort poeng ut av er hvordan «resurser» håndteres (som delte variable). Standardløsningen er at alle slike resurser låses og blir utilgjengelige for andre enn deltagerene så lenge transaksjonen varer. «Growing» og «Shrinking» phase er begreper som beskriver standardimplementasjonen her, men vi har også vært innom «optimistic ...» hvor det ikke låses, men sjekkes og håndteres i etterkant.

End boundary er å fastsette det punktet i tid hvor vi vet at aksjonen har lyktes (og at evt. feil som oppstår i systemet ikke lengre kan skyldes transaksjonen) Her kan vi frigi resurser igjen, så dette punktet er overgangen imellom fasene. Dette **er** et punkt i sann tid og fremkommer ved en avstemming (to-fase commit protocol) imellom alle deltagerene.

Oppgave 2e) – Action Controlleren

I utgangspunktet håndterer den påmeldingen til å begynne med, og avstemmingen på slutten.

Det kan sies en del om feilhåndtering her også – håndtering av deltagere som faller fra, kanskje til og med etter å ha stemt... Bonus til dem som tar noe med om dette.

Oppgave 2f) – Log

Grunnleggende: Alltid *før* vi gjør noe, skriver vi til loggen (som lagres på et trygt sted) hva vi har tenkt å gjøre. Dermed kan vi alltid etter en restart komme opp i den tilstanden vi var i ved å lese log'en og utføre alle «kommandoene» som ligger der.

I tillegg er det vanlig å lagre relevant før-status på variable som endres også, slik at vi kan «rulle tilbake» typisk til recovery points ved backward recovery.

Noe eksotisk er det vel å bruke log-begrepet til replikering av data – hvis en master f.eks. gjør jobben og en slave er interessert i å være oppdatert om hva som er status kan dette fikses ved at slaven eksekverer logen fra masteren. Se neste punkt.

Oppgave 2g) – Prosess par

Forsøk på å tegne figuren på side xx i lefsen er bra. Ellers er grunnprinsippene følgende:

- En slave som eksekverer log fra master.
- En IAmAlive protokoll som gjør at slaven tar over hvis master dør.
- Oppstartsprotokoll; Hvis en nystartet node finner masteren så blir den slave, ellers blir den master.
- Broadcasting av IAmMaster til alle ved overtagelse.

Oppgave 2h) – Kommunikasjon

Dette spørsmålet er direkte om siste tredjedel fra lefsen...

- Ack
- sesjoner
- Timeout/resending
- sjekksummer som del av meldingen
- sekvensnummer som del av meldingen
- Avsender og mottageradresse som del av meldingen

Det bør gjøres klart at detektering av alle feil på en melding fører til at meldingen kastes og systemet går på timeout/resending (eller hva annet som skjer.)

Oppgave 2i) – Atomic Actions vs Async. transfer of control.

Poenget her er egentlig ikke verre enn at når flere tråder samarbeider om å løse en oppgave sammen, hender det at de må samarbeide om feilhåndtering også. Dermed trenges mekanismer for å formidle feil og feilinformasjon fra en tråd til de andre. (Ada og Java tilbyr slike mekanismer.).

Atomic actions er et rammeverk for å muliggjøre feilhåndtering ved flere deltagende tråder. Mao. ATC er en mekanisme som kan brukes for å gjøre feilhåndtering innenfor en AA.

Det som gjør oppgaven ikketriviell er at for Atomic Actions er «backward error recovery» så grundig diskutert, og i dette tilfellet er boken førnøyd med avstemmingen/synkroniseringen på slutten. Mao. det bør gå frem at denne koblingen finnes hvis vi snakker om feil***håndtering*** *(forward* error recovery*)* og ikke bare tilbakerulling til recoverypunkt etter feilet avstemming.

Oppgave 2j) – ATC vs meldinger/polling

På midsemesterprøven skulle de velge den beste blant disse (dårlig oppgave, men);

- Speed; This is the fastest way to convey information between threads.
- Structural; This is a question of design, and having chosen to use shared variable-based synchronization rather than message-based synchronization, and wanting to avoid distributing polling and error handling code everywhere in the threads, Asynchroneous Notification is the only alternative.
- Conceptual; When some situation arises, making the current computation-in-progress meaningless, the *conceptually right thing to do* is to interrupt/abort it like AN techinques allows.
- Exceptional; What we try to do here is to handle error or exception situations, (though the scene is more complex, here, with more participating threads): It is only reasonable that such situations is handled by an extension to the exception language formalism like it is done with Java RTs AsynchronouslyInterruptedException.

Gitt at ATC er et slikt svineri som det er, vil alle unntatt det første åpne diskusjonen "Er det *virkelig* nødvendig" (mens dersom farten/responstiden trenges *må* vi faktisk.) Mao. hastighetsmomentet er en kritisk del av dette svaret.

Men: vanskelig oppgave å vurdere.

Oppgave 3a) – OCCAM til FSP

BUF = (in -> out -> BUF).

To tilstander (klar til å lese, klar til å skrive).

Oppgave 3b) – FSP til Tilstandsdiagram.

Fire tilstander (0, x, u, xu),

8 transisjoner:

x: 0->x

x: u->xu

u: 0->u

u: x->xu

y: x->0

y: xu->u

v: u->0

v: xu -> x

Oppgave 3c) – synkron send & receive

Antar 2 semaforer initialisert til 0, og en buffer til å holde verdien; Siden det bare er to tråder som kaller disse funksjonene, trenger vi ikke å beskytte oss mot flere kall av send/receive samtidig (Noe som ville tatt flere semaforer... grei løsning dette også.).

Hva med dette ?

```
send(int value){
  buffer = value;
  signal(sem_send_ok);
  wait(sem_received_ok);
}
receive(int & value){
  wait(sem_send_ok);
  *value = buffer;
  signal(sem_receive_ok);
}
```

Oppgave 3d) – Fingranulære tilstander i C

Vel, av alle spørsmålene jeg fikk på eksamensdagen var 90% om denne :-) Det er vel ikke usannsynlig at den var urimelig vanskelig.

Vi har i de foregående oppgavene demonstrert at slike fingranulære (OCCAM) prosesser kan

modelleres som tilstandsmaskiner og siden jeg bare ber om stikkord gir ordet “tilstandsmaskin” *minst* 50% uttelling. Mao. for å implementere fingranulære prosesser i C kan vi “simulere” den tilstandsmaskinen som prosessene utgjør. En kan komme godt ut av det med “tilstandsmaskinlignende” switch/case strukturer også.

Hvis noen har tenkt lengre og prøvd å se for seg en komposisjon av flere prosesser, eller tenkt litt på hvordan kanaler kommer inn i dette nærmer vi oss full score; Det er ikke hensiktsmessig (ville jeg tro) å slå sammen flere tilstandsmaskiner til en og dermed få nxm tilstander (som i oppgave 3b) – antall tilstander blir fort for høyt. “Et system av koblede tilstandsmaskiner” er vel full score.

Oppgave 4) – Design

Gitt den bakgrunnen vi har omkring atomic actions og transaksjoner vil noen som sier at de vil implementere denne overgangen som en AA eller transaksjon komme veldig langt bare med det. (Det er ikke så mye mer å si – oppgaven er ikke veldig stor eller vanskelig.)

Problemstillinger de kan ta tak i;

- Å bestemme overgangstidspunktet: en transaksjonsmeneger tar agjørelse (ved avstemming a'la tofase commit).
- Å distribuere overgangstidspunkt: polle på tilstandsvariabel i begynnelsen av hver sample ?
- Å håndtere feil/avbryte overgangen: Hvis noen stemmer nei blir det nei.
- ...

Retteskjema TTK4145 2005 ordinær eks. Stud Nr_____

1a)	Maskinvare mekanismer	
1b)	Semaforer	
1c)	Monitorer	
1d)	Vranglåser	
1e)	Race cond.	
1f)	Bloom	
1g)	Resource Manager	
2a)	N-versjonsprog	
2b)	Domino	
2c)	Exceptions	
2d)	Boundaries	
2e)	ActionController	
2f)	Log	
2g)	Prosess par	
2h)	Kommunikasjon	
2i)	AA vs. ATC	
2j)	ATC vs. polling	
3a)	Occam -> FSP	
3b)	FSP -> tilstander	
3c)	Synkron send&receive	
3d)	Fingranulære prosesser	
4)	modusskift i prosessanlegg.	