Sensorveiledning eksamen ttk4145-04.

Dette er slik jeg ville svart på oppgavene (vel, noe kort, kanskje :-) Det kan være at studentene kan svare på andre måter som fortjener uttelling, og det er godt mulig at jeg ikke har med alle momenter som burde være med (slik at løsningsforslaget ikke vil få full score...).

1a) Ulempene ved bruk av semaforer:

- De føyer seg ikke godt til noe modul-begrep (*brukeren* av resursene må sørge for låsingen) -> En må kjenne hele programmet for å kunne si at det virker.
- Låsing av flere semaforer blir fort uoversiktlig. -> vranglåser. Vi burde hatt muligheten til å frigi semaforen midlertidig ved blokkering inne i regionen.

I stort perspektiv: Dårlig skalerbarhet, elendig vedlikeholdbarhet.

1b) I Occam gir mulig aksess fra flere prosesser av en variabel (annet enn hvis alle leser) --> kompileringsfeil. Mao. vidre synkronisering trenges ikke.

1c) Monitorer er bedre enn semaforer fordi:

- Beskyttelsen kommer på innsiden av *modulen* - vi har tatt et skritt i retning av at modulen passer seg selv. -> bedre vedlikeholdbarhet.
- Vi har fått muligheten til, på strukturert vis, å håndtere samarbeide med andre grunner til å blokke siden blokking fra innsiden av en monitor kan frigi monitoren.

1d) Ulemper med monitorer:

- Denne mekanismen for "samarbeide med andre grunner til å blokke" virker ikke når en monitor kaller en annen. --> Fortsatt dårlig skalerbarhet.
- Selv om vi har tatt et skritt i retning av at "modulen passer seg selv", er det ikke langt nok - brukeren må fortsatt ta hensyn til modulens "blokkeringsmodi" og det må fortsatt gjøres globale analyser for å unngå vranglåser.

1e)

NOTE: Det er en del overlapp i svarene for 1e, 1f og 1g.

- Det er behov for beskyttelse av kommunikasjonen. I MasterMedSlave modus sender både P1 og P2 meldinger. Likedan kan flere tråder detektere at den andre prosessen er nede samtidig, og prøve å opprette ny kontakt samtidig.
- Bildet er litt mer komplisert for g_mode. Det er bare "tidyAndReconnect" som setter denne, og bruken av variabelen er jo atomisk, så selv om de forskjellige trådene som kjører like etter settingen dermed kan ha forskjellig oppfatning av hvilken modus vi er i, så gjør ikke dette skade slik trådene er utformet nå. Deteksjon av at partneren er borte kan imidlertid komme i ras, noe som betyr at mange instanser av tidyAndReconnect kan kjøre samtidig. Hvem av dem som får endelig bestemme hva g_mode skal være er usikkert, men det løser ikke problemet å beskytte g_mode. Mao: Slik programmet er nå har det ingen hensikt å beskytte variabelen g_mode. Konseptuelt sett så burde *tilstandstransisjonene* vært håndtert bedre.

NOTE: Hvis en ser på implementasjonen i appendix er "tidyAndReconnect" allerede "beskyttet". Dette kan kanskje lede til forvirring – Uansett er dette en beskyttelse som ikke virker...

1f)

Programmet virker ikke - semaforen på g_mode kan forsøkes reserveres flere ganger av samme tråd -> Deadlock. Det største problemet som er at tidyAndReconnect kan kalles fra flere tråder samtidig forsøkes ikke løses. Beskyttelsen av g_mode på denne måten har ingen hensikt.

1g)

Kommunikasjonen: Beskyttelsen av sending og mottak er to forskjellige ting, og en monitor-lignende greie for å beskytte sendingen er ok. Slik strukturen er nå kan kommunikasjonsmodulen bli hengende på aksessen til g_mode. Dette er unødvendig, kommunikasjonsmodulen bør designes om til å bli uavhengig av g_mode (dvs. selv holde rede på hvilke linjer som går ut og inn) Mottaket: Det bør sikres at bare en tråd leser meldinger, og det er et spørsmål om denne tråden bør være en del av modulen. Hvis flere tråder *har* behov for å motta meldinger må vi enten ha en "de-multiplekser"-tråd som tar i mot, eller bruke flere sockets.

g_mode: denne variabelen beskriver systemets tilstand som alle trådene er avhengige av. Beskyttelsen bør ikke legges på *variabel-aksessen* men gjøres til en generell måte å gjøre transisjoner imellom tilstandene på; F.eks. Innføre en egen do-nothing "undecided" tilstand, vente til alle tråder har akseptert denne, og så gjøre transisjonen dit vi skulle.

2a)

Normen er her "rate monotonic priority ordering".

2b)

I tillegg til punktet over er det kriteriene på side 466 i boken:

2. alle prosesser kjent
3. bare periodiske prosesser.
4. Uavhengige prosesser.
5. Overhead kan sees bort ifra
6. Deadlines == periods.
7. Vi kjenner WCET.

4 er det viktigste og teller langt mer enn 1/7.

Anne kom opp med at det at scheduleren er preemptive er en forutsetning.

2c) Ulemper EDF

Vi har noen punkter på s. 474:

1. FPS lettere å implementere (men ærlig talt ikke mye :-). FPS har mindre overhead (Jo, men...) - Ikke det viktigste punktet...
2. EDF forholder seg *dårlig* til prosesser uten deadlines.
3. deadline != viktighet (som ved FPS), men å mekke på prioriteten for å kompensere for dette går an med FPS, men tilsvarende er urimelig kompliert ved EDF.
4. EDF kollapser fullstendig ved overload.

I stort perspektiv: 2 og 3 gir manglende fleksibilitet, mens FPS igjen er veldig fleksibel mhp alle

mulige slags utvidelser.

I tillegg kan det komme en kommentar på at spørsmålet er feil stilt; De to formlene er ikke sammenlignbare siden den ene er nødvendig og tilstrekkelig, mens den andre bare tilstrekkelig.

2d) Scheduling-strategi med vranglåsunngåelse:

Dette er altså Priority Ceiling Protokollen - det er riktignok to varianter av disse beskrevet i boken (s. 490 og 492)...

Den enkleste er den andre (immediate ceiling priority protocol)!

- Hver prosess har en (statisk) prioritet.
- Hver "resurs" har en prioritet svarende til prioriteten til den prosessen med høysest prioritet som bruker resursen.
- En prosess tar over prioriteten til evt. resurser den låser.

Den andre (dvs. den første) varianten er formulert slik at prioriteten til en prosess som har en resurs ikke økes når resursen allokeres, men først når det ville hatt en låsemessig konsekvens. Beskrivelsen her er adskillig vanskeligere å få riktig :-)

For full score bør det, siden spørsmålet er motivert fra vranglåsunngåelse, bør det argumenteres vidre til hvordan denne faktisk unngår vranglåser:

Når vi fikk kjøre slik at vi fikk reservert en resurs, vet vi at ingen høyereprioritetstråder (som dermed kunne ha "mer bruk for" resursen enn oss.) kunne kjøre. Etter at vi fikk resursen har vi prioriteten til resursen og vi vet av alle prosesser som har høyere prioritet enn oss *ikke* bruker den resursen som vi har.

3a) A unngå problemer med vranglåser: s400 i boken.

Tre hovedkategorier i boken, med forskjellige underpunkter...

- Prevention - sørge for at minst en av de 4 forutsetningene for deadlock ikke forekommer:
  - Alle prosesser reserverer alle resurser de trenger samtidig.
  - Mekanisme for stjeling av resurser.
  - Sikre seg mot sirkulære avhengigheter...
- Avoidance
  - Legge planlegging for vranglåsunngåelse inn i scheduleren.
  - Her kommer vel priority ceiling inn også...
- Detection and recovery
  - timeout, analyse av avhengigheter.
  - abort av prosesser, abort av atomic actions, andre former for stjeling av resurser.

3b) Atomic Actions

Problemet: Feilhåndtering når vi har flere samarbeidende prosesser.

Løsningen: (En av flere måter å beskrive dette på...) Alle deltagende prosesser kjenner start og slutt-punkter for en AA i tillegg kommuniseres ikke konsekvenser av AA til utenforstående prosesser før AA er ferdig (på den ene eller andre måten...) ("isolation").

Studenten kan selvfølgelig gå inn på *hvordan* vi gjør selve feilhåndteringen: Et langt lerret å bleke...

Standardmetode - synkroniserer mot en "AAController" som synker entry og exit (Dette er bare AA-mekanismen - før feilhåndteringen.)

Backward Error Recovery: Rull tilbake til utgangspunktet. - Ikke beskrevet i seg selv i boken, men kan dekkes av mekanismene under.

Forward Error Recovery: I praksis kommer vi her inn på Asynchronous Notification som implementasjonsmekanisme -> Studenten har rotet seg bort...

3c) Resurskontroll

Fra boken p. 382

- Rekkefølgen viktig ? (FIFO ?)
- Prioriteten viktig ?
- Typen (lese skrive f.eks.)
- Tilstnaden/historien
- Parametrene

3d) Sammenligning imellom delt-variabel (dvs) og meldings synkronisering (ms).

- DVS skalerer dårlig, alle betraktninger om hvordan MS skalerer er positivt :-)
- Ingen skikkelige modulbegreper ved DVS - ved MS har vi serveren
- Vi får gjerne flere prosesser ved MS, Overhead ?
- MS ligger an til distribuerte systemer.

3e) N-versjons programmering:

- Budskjett
- Uavhengighet i implementasjonene
- Feil i spekken vil ikke finnes.
- Avstemmingsmekanismen (OK for "kontinuerlige systemer", men hva med terskler ?)
- Feil i "driveren"

3f) Feilsjekker:

s. 116 - Godt stoff.

- Håndtering av exceptions "utenifra" (x/0-exception)
- regn ut på to måter og sammenlign
- Whatchdog
- Sjekke at tidskrav overholdes.
- Reversal - gjør omvendte beregningen (sett løsningen inn i ligningen.)
- Sjekksummer ++
- Rimelighetssjekker
- Structural checks (konsistens av datastrukturer)
- "Dynamic reasonableness" - ikke for store sprang i et pådrag.

Oppgave 4) Design

Noen sentrale problemstillinger jeg kan komme på:

- Grensesnitt mot porten - Siden info fra porten aldri skal mistes, må porten si ifra om at noen vil gjennom og få kvittering på dette før personen slippes igjennom. Punktet bør også dekke hva som skjer hvis noden kræsjer inne i dette løpet.
- Avlesing av antall personer inne. Alle porter må stenges før en avlesning, og åpnes etterpå.
- Konsistens i bildet av hvor mange som er inne - hvis noen har gått igjennom en port, må de andre portene få beskjed om dette, (og, bare en gang!)

(Om noen fremstiller dette som use-caser er det jo greit :-)

Hvis noen gir svar på disse, ville de overbevist meg om at de kunne gjøre jobben :-)

Det er ingen krav til ytelse her, så en del forenklende antagelser er vel ok:

- Det kan være akseptabelt at folk blir stengt inne i en port, hvis noden går ned.
- Kanskje bare en port kan slippe noen igjennom i slengen ?

Mitt design:

- Plaser nodene i en ring, hvor alle noder holder rede på om noden til venstre for seg er oppe ved å pinge. Hvis nabo-noden er nede forholder noden seg til neste nabo.
- Jeg tenker meg porten som en sluse, med to dører og deteksjon av om noen er inne i den

Sekvensdiagram med 3 aktører; Porten, Noden og Neste Node:
(Mennesket kommer og vil inn – blir låst inne i slusen)
P->N: Noen vil gjennom
N->NN: Jeg har tenkt å slippe noen gjennom
NN->N: OK
N->P: OK, slipp ham inn.
P->N: Han er inne, porten er tom.
N->NN: Her er nytt antall. (Denne går hele ringen rundt.)

- Når en node kommer opp må den få status fra når før den gikk ned. Her kan en spørre både porten og nabonoden om deres oppfatning av hva som er status.
- Hver node holder rede på antall mennesker inne, og hver gang noen går igjennom sendes en endringsmelding (node,sekvensnummer,endring) rundt hele ringen. Nodene holder rede på siste sekvensnummer for hver node.
- Ved avlesning sendes en stengOgRapporterAntall (spørreNode, mittAntall) melding rundt ringen. Det sjekkes konsistens. Når avlesningen er godkjent sendes åpnePorter (spørreNode) rundt. Det må telles opp hvor mange slike stenginger som er aktive.
- Vi bør vel ha en konsistenssjekk-protokoll også, som går regelmessig...

Det kommer en del nodeId'er og sekvensnummer med i meldingene... Det
skulle være grunnlag her for en god del tilstandsmaskiner og sekvensdiagrammer.

Oppgave 5) Implementasjon;

5a) Occam:

Vi skal ha 20 prosesser som gjør en iterasjon:

```
PROC iter (CHAN of REAL in,out)
  SEQ
    in ? x
    in ? ei
    res = (ei+(x/ei))/2
    out ! x
    out ! res
:
```

Vi trenger en server - en prosedyre som setter opp disse 20 prosessene; Vi trenger 21 kanaler.

```
PROC squareRoot (CHAN of REAL in,out)
  CHAN of REAL [21] channels:
  PAR
    SEQ -- feed the chain
      in ? x
      e1 = x/2
      channels[0] ! x
      channels[0] ! e1
    PAR i=0 FOR 20 -- Set up the 20 processes
      iter(channels[i],channels[i+1])
    SEQ -- Deliver the result
      channels[21] ? x
      channels[21] ? res
      out ! res
:
```

5b) Resource Manager.

Cluet her er en RM monitor som holder rede på en sortert liste av klienter. Når resursen frigis vekkes alle ventende klienter opp, og sjekker om de ligger først i køen. Hvis de gjør det, får de resursen, ellers må de sove igjen.

Jeg forutsetter listen og sorteringen (list_insertSorted,list_getFirstId og list_removeFirst).

```
// ---- Monitor_RM

bool g_resourceAvailable = true;

allocate(int threadId,int importance){
  list_insertSorted(threadId,importance);
  while not (g_resourceAvailable && (list_getFirstId() == threadId)){
    // Not my turn - must wait
    wait();
  }
  // OK, got it!
  g_resourceAvailable = false;
}

free(){
  list_removeFirst();
  g_resourceAvailable = true;
  notifyAll();
}
```