NTNU
Norwegian University of
Science and Technology

Institutt for teknisk kybernetikk
Fakultet for informasjonsteknologi
matematikk og elektroteknikk

Bokmål & Engelsk

Eksamen i TTK4145
Sanntidsprogrammering
Never 3000
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 3 til og med side 47. 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 49 through 93.

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». I utgangspunktet teller alle deloppgavene likt; Unntaket er evt. oppgaver markert med (2x) eller (3x) (osv.) som teller hhv. dobbelt eller tredobbelt.

Generally:

Be short and clear. Enumerate items and arguments, use figures where appropriate. You may be punished for text without substance. All questions have similar weights except possibly where marked with (2x), (3x) etc.

Hjelpemidler:

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

Faglig kontakt under eksamen:

Sverre Hendseth

Telefon 98443807

1 Semaforer og delt variabel synkronisering

1-1) (Used in: v2015)

Den følgende koden er tatt fra en modul som heter "logic" i et gammelt heisprosjekt.

```
1  #include <stdio.h>
2
3  #include "elev.h"
4  #include "control.h"
5  #include "logic.h"
6
7  // The order tables; one cell is unused in each table; ref. the floor panel
8  static int calls_up[N_FLOORS];
9  static int calls_down[N_FLOORS];
10
11 // helper function that is used by place_order(...) and place_call(...)
12 // to store orders in the order tables.
13 static void add_job(int path, int floor);
14
15 // Sverre: More functions omitted...
16
17 void place_order(int destination)
18 {
19   // cancelling emergency stop
20   if (emergency && !cancel_emergency) {
21     restart(destination);
22     return;
23   }
24   // The two border cases
25   if (destination == 0) {
26     calls_up[destination] = TRUE;
27   }
28   else if (destination == N_FLOORS - 1) {
29     calls_down[destination] = TRUE;
30   }
31   // The three normal cases
32   else if (destination > current_floor) {
33     calls_up[destination] = TRUE;
34   }
35   else if (destination < current_floor) {
36     calls_down[destination] = TRUE;
37   }
38   else if (destination == current_floor) {
39     add_job(direction, current_floor);
40   }
41 }
42
43 void restart(int destination)
44 {
45   cancel_emergency = TRUE;
46   // If STOP has been pressed between two floors
47   if (!on_floor()) {
```

```
48      if (destination == current_floor) {
49        // The lift is over current_floor, needs to go down
50        if (ref_floor > current_floor) {
51          direction = DOWN;
52        }
53        // The lift is under current_floor, needs to go up
54        else {
55          direction = UP;
56        }
57      }
58      else if (destination < current_floor) {
59        direction = DOWN;
60      }
61      // destination > current_floor.
62      else {
63        direction = UP;
64      }
65      start_motor();
66    }
67    place_order(destination);
68  }
```

Kritiser koden; Påpek evt. gode og dårlige ting ved koden.

1-2) (Used in: v2015)

I den tilsvarende headerfilen finner vi dette:

```
1  // Takes a command from the comparment panel. If STOP has been pressed
2  // the the lift is to start again, this by calling restart(...). After
3  // this, orders can be entered normally again. If the lift is ordered
4  // to a floor over its position or to the lowest floor the order goes
5  // into the up-table. Oppositely for a floor under its position or the
6  // highest floor. If the lift is ordered to the floor it visited last,
7  // or the floor it is in, the table depends on the direction of the
8  // lift.
9  void place_order(int destination);
10
11 // Is called only in the emergency state by place(order...) and will
12 // then cancel the emergency state. If the function is called to start
13 // the lift after STOP has been pressed in a floor, only returned the
14 // control to place_order(...) and the lift will start. If, however,
15 // the function is called after a press of STOP between floors, the
16 // variable ref_floor (set by update_floor()) is used to decide the
17 // lifts position relative to the last floor. The motor can then be
18 // started in the correct direction by returning the control to
19 // place_order.
20 void restart(int destination);
```

Kritiser koden - dvs. kommentarene; Påpek evt. gode og dårlige ting.

1-3) (Used in: v2015)

Den hele headerfilen ser slik ut (etter at Sverre har fjernet alle kommentarene... vurder koden som om kommentarene aldri var der):

```
1  #ifndef __INCLUDE_LOGIC_H__
2  #define __INCLUDE_LOGIC_H__
3
4  void change_direction(void);
5  void clear_orders(void);
6  void place_call(int direction, int floor);
7  void place_order(int destination);
8  void delete_order(void);
9  void restart(int destination);
10 int called_here(void);
11 int visible_orders(void);
12 void handle_order(void);
13 void debug_logic(void);
14
15 #endif
```

Kritiser koden - dvs. modulgrensesnittet denne gangen; Påpek evt. gode og dårlige ting.

1-4) (Used in: v2015)

Hva er formålet med prosesspar?

1-5) (Used in: v2015)

Akseptansetester er en viktig liten teknikk for prosesspar. Forklar hvordan akseptansetester bidrar til funksjonaliteten til prosesspar.

1-6) (Used in: v2015)

Definer begrepene vranglås (deadlock) og race condition.

1-7) (Used in: v2015)

Kan vranglåser og race conditions forekomme i meldingssendende systemer?

Java-dokumentasjonen sier dette om *hvilken* tråd som blir vekket når notify kalles:

> If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Bruken av ordet “arbitrary” (vilkårlig) her er ganske typisk, og blir brukt om mange tilsvarende synkroniseringsprimitiver.

1-8) (Used in: kont2015 , v2015)

Sverre påstår at det bare er en rimelig måte å implementere dette på — first in, first out (FIFO); den tråden som har ventet lengst blir den som blir vekket — og at alle implementasjoner sannsynligvis vil ha denne oppførselen som default.

Hva gjør FIFO til et så godt valg?

1-9) (Used in: kont2015 , v2015)

Hvis FIFO strategien for å vekke prosesser er så god; hvorfor insisterer dokumentasjonen så ofte på vilkårlighet?

Denne koden er fra Burns & Wellings kapitlet om semaforer (om enn noe forenklet) og har en feil.

```
1  // The priority parameter is either 0 or 1
2  // The mutex M is initialized to 1, the semaphores in the
3  // PS array is initialized to 0
4  void allocate(int priority){    void deallocate(){
5    Wait(M);                        Wait(M);
6    if(busy){                       busy=false;
7      Signal(M);                    waiting=GetValue(PS[1]);
8      Wait(PS[priority]);           if(waiting>0) Signal(PS[1]);
9    }                               else{
10   busy=true;                        waiting=GetValue(PS[0]);
11   Signal(M);                        if(waiting>0) Signal(PS[0]);
12  }                                  else{
13                                       Signal(M);
14                                     }
15                                   }
16                                 }
```

Poenget er at vi har en ressurs som brukes av mange tråder, men istedet for å tilordne ressursen til trådene i den rekkefølge de ba om den, så skal tråder som angir høy prioritet i parameterlisten ha forrang.

1-10) (Used in: v2015)

Hva er feilen: Hva kan skje? Hva er konsekvensene av bugen?

1-11) (Used in: v2015)

Denne feilen er ikke så enkel å fikse gitt at vi bruker bare semaforer. Hvorfor? Hva er vanskeligheten?

1-12) (Used in: v2015)

Gi en kort beskrivelse av hvordan synkroniseringsprimitivene i Java (synchronized

methods, wait, notify and notifyAll) virker.

1-13) (Used in: v2015) **(2x)** Skisser (pseudokode) hvordan du kunne løse problemet fra oppgave 1f med Javas synkroniseringsprimitiver.

1-14) (Used in: v2015)

Gi en kort beskrivelse av hvordan synkroniseringsprimitivene i ADA (Protected Objects, functions, procedures and entries with guards) virker.

1-15) (Used in: v2015)

Vi vurderer å implementere funksjonaliteten fra oppgave 1f i Ada:

På tross av at synkroniseringsmekanismene i ADA og Java har omtrent samme utrykkskraft, så har vi her støtt på et problem som det ikke er like greit å løse i Ada som det var i Java (eller som det ville vært med posix, monitorer, conditional critical regions el.l.).

Hvorfor?

...så vi gjør oppgaven om litt: Vi lager to allocate-funksjoner istedet for en; en for lav prioritet og en for høy. allocate_highPri() og allocate_lowPri().

1-16) (Used in: v2015) **(2x)** Skisser (pseudokode) hvordan du kunne løse dette problemet med Adas synkroniseringsprimitiver.

1-17) (Used in: v2015)

Meldingsbasert interaksjon imellom tråder leder til et veldig forskjellig design i forhold til systemer med delt variabel synkronisering. Hvordan? Beskriv kort forskjellen i design.

1-18) (Used in: v2015) Den underliggende situasjonen i koden fra task 1f — å styre aksessen til en delt ressurs basert på prioritet — ville ikke en gang komme opp i et meldingssendende system.

Beskriv et tilsvarende problem du kan tenke deg i et meldingssendende system (Noe kreativitet trengs kanskje? Men de to prioritetsnivåene bør fortsatt være en del.) og skriv pseudokode til en løsning.

1-19) (Used in: v2015)

Sammenlign kompleksiteten av implementasjonen din i forrige oppgave med den i koden koden i oppgave 1f. Anta at et rimelig mål på kompleksitet er antall tilstander programmet kan være i, hvis du hadde laget en formell modell av det, for eksempel med FSP. (...men du trenger ikke modellere noe.)

Hva er forskjellen i kompleksitet? Forsøk å forklare hvorfor det er slik.

1-20) (Used in: v2015)

Priority ceiling protokollen, i tillegg til å løse "unbounded priority inversion" problemet, har også egenskapen at vi sikrer oss mot vranglåser. Forklar hvordan.

Vi har laget en modul som forvalter en tabell med person-struct'er og som inneholder bl.a. disse to funksjonene:

```
1 void printName(int i){
2   printf("%s",persons[i].firstName);
3   printf("␣%s\n",persons[i].lastName);
4 }
5 void printAddress(int i){
6   printf("%s",persons[i].street);
7   printf("␣%d\n",persons[i].streetNumber);
8 }
```

Den fungerte flott helt til vi gjorde programmet som bruker denne modulen multitrådet...

1-21) (Used in: kont2014)

Av og til skriver programmet ut rare feil når adresselisten genereres (Sverre Hendseths og Ola Normanns adresser er blitt blandet):

```
...
SverreOla Normann
Osloveien 23
 Hendseth
Trondheimsveien 34
...
```

Hva er problemet?

1-22) (Used in: kont2014)

...og enda verre: En sjelden gang kommer det til og med blandinger som er delt midt i ord:

```
...
SveOla Normann
Osrre Hendseth
Trondheimsveien 34
loveien 23
```

...

Hva er problemet her?

1-23) (Used in: kont2014)

Hva kaller vi slike feil i multitrådete programmer som opptrer "av og til"?

1-24) (Used in: kont2014)

Vi løser dette problemet med å beskytte funksjonene i modulen vår med semaforer. Modulen ser nå slik ut:

```
1  // Assume a semaphore personSem initiated to 1 at the start of the program
2  void printName(int i){
3    sem_wait(personSem);
4    printf("%s",persons[i].firstName);
5    printf("␣%s\n",persons[i].lastName);
6    sem_signal(personSem);
7  }
8  void printAddress(int i){
9    sem_wait(personSem);
10   printf("%s",persons[i].street);
11   printf("␣%d\n",persons[i].streetNumber);
12   sem_signal(personSem);
13 }
```

Hvordan løser bruken av semaforer dette problemet?

1-25) (Used in: kont2014)

Det viser seg at etter som programmet vårt vokser så opptrer kombinasjonen av printName(i) og printAddress(i) veldig mange steder i (navnelapper, visittkort, brevhoder og konvolutter...), og vi beslutter å lage funksjonen *printNameAndAddress* som en del av modulen:

```
1  void printNameAndAddress(int i){
2    sem_wait(personSem);
3    printName(i);
4    printAddress(i);
5    sem_signal(personSem);
6  }
```

Vurder modulgrensesnittet i et kodekvalitetsperspektiv: Hva synes du om denne avgjørelsen?

1-26) (Used in: kont2014)

Imidlertid så fungerer ikke dette i det hele tatt: Programmet skriver ikke ut en eneste kombinasjon av navn og adresse; det bare henger.

Hva skjer?

1-27) (Used in: kont2014)

Hva kalles denne typen feil?

1-28) (Used in: kont2014)

I denne situasjonen hadde det kanskje vært et alternativ å droppe semaforbeskyttelsen av *printNameAndAddress*? Hvorfor ville/ville ikke dette være greit?

1-29) (Used in: kont2014)

Dette problemet er ikke lett å løse med semaforer - jeg vil si vi har et eksempel på en svakhet ved semaforers "expressive power" her. Forklar begrepet.

1-30) (Used in: kont2014)

I Java ville dette problemet latt seg lett løse: Beskriv kort hvordan?

1-31) (Used in: kont2014)

I Ada (dvs. med andre mekanismer for synkronisering enn semaforer) ville dette problemet latt seg lett løse: Beskriv kort hvordan?

1-32) (Used in: kont2014)

Med posix mutexer løser ikke problemet seg helt av seg selv; vi må sette mutexen i "recursive" modus (PTHREAD_MUTEX_RECURSIVE). Hvordan tror du dette virker?

1-33) (Used in: kont2014)

Hvis du virkelig måtte beskytte en av modulene dine bare med semaforer; hvordan ville du gripe det an?

1-34) (Used in: kont2014)

Mange av slike "resource allocation" problemer forekommer ikke i rene meldingssendende systemer. Hvorfor ikke?

1-35) (Used in: kont2014) **(2x)** Et litt større utsnitt av modulen vises her:

```
1 #ifndef PERSON_H
2 #define PERSON_H
3
4 typedef struct {
5   char * firstName;
6   char * lastName;
7   char * street;
8   int streetNumber;
9 } TPerson;
```

```
protected body Bounded_Buffer is
  entry Get (Item : out Data_Item) when Num /= 0 is
  begin
    Item := Buf(First);
    First := First + 1; Num := Num - 1;
  end Get;

  entry Put (Item : in Data_Item) when Num /= Buffer_Size is
  begin
    Last := Last + 1; Num := Num + 1;
    Buf(Last) := Item
  end Put;
end Bounded_Buffer;
```

Figur 1: Bounded Buffer in Ada

```

void reallocateArray(int newSize);

TPerson ** getArray();

void printName(int personNumber);
void printAddress(int personNumber);
void printNameAndAddress(int personNumber);
...
#endif
```

Kritiser fra et kodekvalitetsmessig perspektiv inkluderingen av typen *TPerson* og funksjonene *reallocateArray*() og *getArray*() i modulgrensesnittet.

1-36) (Used in: v2014)

Bloom brukte "expressive power" og "ease of use" for å sammenligne synkroniseringsprimitiver. Forklar de to begrepene.

1-37) (Used in: v2014)

Gi en kort beskrivelse av hvordan synkroniseringsprimitivene i ADA (Protected Objects, functions, procedures and entries with guards) og Java (synchronized methods, wait, notify and notifyAll) virker.

1-38) (Used in: v2014)

Sammenlign de gitte (Fig. 1 og 2) implementasjonene av en "Bounded buffer" i Ada og Java i forhold til "expressive power og ease of use" av synkroniseringsprimitivene. Hvilken implementasjon synes du utrykker bounded buffer logikken tydeligst?

1-39) (Used in: v2014)

```
1   public synchronized void put(int item)
2   {
3       while (numberInBuffer == size) wait();
4       last = (last + 1) % size ;
5       numberInBuffer++;
6       buffer[last] = item;
7       notifyAll();
8   }
9
10  public synchronized int get()
11  {
12      while (numberInBuffer == 0) wait();
13      first = (first + 1) % size ;
14      numberInBuffer--;
15      notifyAll();
16      return buffer[first];
17  }
```

Figur 2: Bounded Buffer in Java

I forhold til de synkroniseringsproblemene som Bloom satte opp så feiler Adas protected objects spesielt i ett tilfelle. Forklar.

1-40) (Used in: v2014)

Les den vedlagte siden fra "The little book of semaphores" om "Cigarette smokers problem".

Skriv pseudokode med bruk av Adas protected objects for å løse dette problemet. Du skal skrive et protected object som forvalter ansvaret for å vekke opp riktig røyke-tråd.

Agenten er gitt og uforanderlig; anta at den kaller et tilfeldig par av funksjoner, prosedyrer eller entries (ditt valg) — Tobacco(), Paper() eller Matches() — i objektet vårt for å signalisere at de gitte ressursene er blitt tilgjengelige hver gang bordet er tomt.

Det holder at du skisserer bare objektet, men det bør gå klart frem hvordan røykertrådene vekkes opp.

1-41) (Used in: v2014)

Løs det samme problemet (pseudokode), men denne gangen bare med semaforer som synkroniseringsmekanisme.

Agenten er også her gitt og uforanderlig; anta at den kaller signal() på et tilfeldig par av de tre semaforene sem_paper, sem_match og sem_tobacco hver gang bordet er tomt. Du *kan* bruke tester og/eller tabeller av semaforer :-)

Hint: Det trenger ikke være røyketrådene selv som venter på de tre semaforene. Du kan

```
1 SEMAPHORE NInBuffer(0),NFree(N),Mutex(1);
2
3 put(e){                        get(e){
4   wait(NFree);                    wait(NInBuffer);
5   wait(Mutex);                    wait(Mutex);
6     // enter into buffer           // get e from buffer
7   signal(Mutex);                  signal(Mutex);
8   signal(NInBuffer);              signal(NFree);
9 }                              }
```

Figur 3: Bounded Buffer with semaphores

godt innføre en eller flere tråder med formidlingsansvar.

1-42) (Used in: v2014)

Som du leste, så foreslo Patil "Cigarette smokers problem" som et argument for at "Expressive power" til semaforer er begrenset. Synes du han lyktes med dette argumentet? (Begrunn svaret kort)

1-43) (Used in: v2014)

En bounded buffer er en buffer, hvor en prosess/tråd som prøver å lese fra en tom buffer, eller som prøver å fylle i en full buffer vil bli blokkert. Konseptuelt sett kan vi modellere en bounded buffer slik i FSP:

```
1 BUF(N=5) = BUF[0],
2 BUF[n:0..N] = (when n < N put -> BUF[n+1]
3               | when n > 0 get -> BUF[n-1]).
```

Anta N=3: Tegn transisjonsdiagrammet for dette systemet.

1-44) (Used in: v2014)

Se på Fig. 3 som er bounded buffer med semaforer. Modeller denne semafor-løsningen i FSP. (Hint: sjekk oppave a for eksempel på syntaksen.)

1-45) (Used in: v2014)

Anta at bufferet bare har plass til ett element (N=1): Tegn transisjonsdiagrammet.

1-46) (Used in: v2014)

Hvis du har fått oppgave 2a og 2c riktig vil du se at modellen av den semaforbaserte implementasjonen av en Bounded Buffer er mer komplisert enn den konseptuelle modellen — i hvert fall hvis vi ser for oss bufre av samme størrelse. Hvorfor er det slik?

1-47) (Used in: v2014)

Pseudokode: Skisser hvordan en bounded buffer kunne blitt implementert i et regime av meldingsendende prosesser. (Hint: *Synkrone* meldinger forenkler det litt...)

1-48) (Used in: v2014)

Kommenter på den forventede (du trenger ikke skrive kode her) kompleksiteten av transisjonsdiagrammet for en slik meldingsbaserte implementasjon av et bounded buffer.

1-49) (Used in: v2015 , v2014)

Meldingsbaserte systemer er ikke ansett som så egnet til å implementere systemer med sanntidskrav. Hvorfor?

1-50) (Used in: v2014)

I listingen under gjengis et modulgrensesnitt (en C header fil) fra en gammel prosjektinnlevering. Modulens navn er "cost". Kritiser, kort og punktvis, designet.

```
1  #ifndef lift_cost_h
2  #define lift_cost_h
3
4  int calculateCost(int currentFloor, int direction, int orderedFloor, int
       orderedDirection);
5
6  int downCost[MAX_ELEVATORS][N_FLOORS];
7  int upCost[MAX_ELEVATORS][N_FLOORS];
8
9  void fillCostArrays();
10 void clearCosts(void);
11
12 int lowestCostFloor(int elevator);
13 int lowestCostDirection(int elevator);
14
15 int findBestElevator(int floor, int direction);
16
17 void designateElevators();
18 void clearDesignatedElevator();
19
20 int designatedElevator[N_FLOORS][2];
21
22 #endif
```

1-51) (Used in: v2014)

Dette er et annet modulgrensesnitt fra et annet prosjekt. Denne gangen med navnet "jobqueue". Kritiser på samme måte; Sammenlign gjerne med det forrige.

```
1  #ifndef _JOB_QUEUE_H
2  #define _JOB_QUEUE_H
3
4  #include <time.h>
```

```
5
6  //Set flag at given floor in the given direction.
7  void set_flag(int floor, int direction);
8
9  //Deletes flag at given floor in the given direction.
10 void delete_flag(int floor, int direction);
11
12 //Returns one if flag at given direction at given gloor is set.
13 int get_flag(int floor, int direction);
14
15 //Deletes all flags in table.
16 void delete_all_flags();
17
18 //Returns one if there are no flags set in table.
19 int flag_table_empty();
20
21 //Returns one if there are any flags above the given floor.
22 int any_flags_above(int floor);
23
24 //Returns one if there are any flags below the given floor.
25 int any_flags_below(int floor);
26
27 //Prints the table of flags to the terminal.
28 void print_flag_table();
29
30 #endif
```

1-52) (Used in: v2014)

Listingen under er funksjonen "delete_flag" fra modulen i forrige oppgave. Kritiser (kort og punktvis).

```
1  //Delete flag in table.
2  void delete_flag(int floor, int direction) {
3
4          if(floor == 0) {
5                  flag_table[floor][DIR_UP] = 0;
6                  flag_table[floor][DIR_DOWN] = 0;
7                  elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
8          }
9          else if(floor == 3) {
10                 flag_table[floor][DIR_UP] = 0;
11                 flag_table[floor][DIR_DOWN] = 0;
12                 elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
13         }
14         else{
15                 if(!any_flags_above(floor) && get_current_direction() ==
                       DIR_UP) {
16                     flag_table[floor][DIR_UP] = 0;
17                     flag_table[floor][DIR_DOWN] = 0;
18
19                     elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
20                 }
```

```
21                  else if(!any_flags_below(floor) && get_current_direction() ==
                         DIR_DOWN) {
22                      flag_table[floor][DIR_UP] = 0;
23                      flag_table[floor][DIR_DOWN] = 0;
24
25                      elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
26                  }
27
28                  flag_table[floor][direction] = 0;
29          }
30
31          if(direction == DIR_UP && floor < 3) { elev_set_button_lamp(
                BUTTON_CALL_UP, floor, 0); }
32          if(direction == DIR_DOWN && floor > 0) { elev_set_button_lamp(
                BUTTON_CALL_DOWN, floor, 0);}
33
34          elev_set_button_lamp(BUTTON_COMMAND, floor, 0);
35  }
```

1-53) (Used in:

1-54) (Used in: h2009)

Feil bruk av semaforer i et program kan få programmet til å gå i vranglås (deadlock): Hva er vranglås? Gi et eksempel på hvordan dette kan skje.

1-55) (Used in: v2013 , h2009)

Generelt; List opp metoder for å unngå at vranglåser skal bli et problem i et programvaresystem.

1-56) (Used in: h2009)

Akseptansetester er ansett som et viktig virkemiddel for feilhåndtering. Hva skiller akseptansetester fra de mer tradisjonelle sjekkene på feiltilstander?

1-57) (Used in: v2014 , v2013 , v2011kont , h2010)

Akseptansetester er ansett som et viktig virkemiddel for feilhåndtering. Hva oppnår vi ved å bruke akseptansetester heller enn bare de mer tradisjonelle sjekkene på feiltilstander?

1-58) (Used in: v2014)

Gi eksempler på hva en kan teste på for å lage akseptansetester.

1-59) (Used in: v2014)

Når en akseptansetest feiler så vet vi at *noe* er galt, men ikke nødvendigvis nøyaktig hva. Hvordan kan vi håndtere en slik situasjon?

1-60) (Used in: v2013)

Hva oppnår vi med å slå sammen feilmoder?

1-61) (Used in: h2009)

Hva er et systems feilmoder, og hva oppnår vi med å slå sammen feilmoder?

1-62) (Used in: v2011kont , h2009)

Optimistic Concurrency Control er en teknikk for å unngå overheaden ved låsing av ressurser. Hvordan virker dette?

1-63) (Used in: h2010)

"Ressurskontroll" er mer enn gjensidig utelukkelse... Hva mer kan vi ønske oss fra et system for ressurskontroll enn gjensidig utelukkelse?

1-64) (Used in:

(3x) Forklar kort følgende begreper:

- En guard slik begrepet er brukt i den opprinnelige definisjonen av monitorer, eller om ADA entries.
- Optimistic Concurrency Control
- Race condition
-

1-65) (Used in: v2011kont , h2009)

Delt variabel-synkronisering kan kritiseres for dårlig skalerbarhet. Gjengi kort argumentene for dette.

1-66) (Used in: h2010)

Å designe parallelle systemer med meldingssending er et alternativ til å basere det på synkronisering. Gi kort og punktvis fordeler og ulemper med bruk av disse to i sanntidssystemer i forhold tilhverandre.

Vårt mål er å lage en barriere/rendezvous imellom 3 tråder ved hjelp av semaforer. Dvs. at trådene skal vente på hverandre og ikke forstette før alle tre har kommet frem til synkroniseringspunktet.

1-67) (Used in: h2009)

```
1 if(getWaiting(A) == 2){ // GetWaiting returns the number of
2                         // processes waiting for the semaphore
3   // The two others are here already - release and continue.
4   signal(A);
5   signal(A);
6 }else{
7   // The number of waiting threads is less than two - must wait.
8   wait(A)
9 }
```

Denne koden har en race condition. Hva er en race condition? Hva er problemet med koden?

1-68) (Used in: h2009)

```
1 count++;
2 if(count==3){
3   // All here, signal the others, reset and continue
4   signal(A);
5   signal(A);
6   count = 0;
7 }else{
8   // Not everyone have arrived, must wait
9   wait(A);
10 }
```

Denne koden har minst ett problem knyttet til seg: Hvilke(t)?

1-69) (Used in: h2009)

(implementasjon/pseudokode,3x) Lag en barriere som virker.

1-70) (Used in: v2011kont , h2009)

(implementasjon,pseudokode,2x) Vi har en ressurs i systemet vårt som mange tråder bruker og hvor tilgangen må synkroniseres. Når det er flere ventende tråder skal den *sist* ankomne få prioritet. Skriv pseudokode for allocate() og free() som gir denne egenskapen. Bruk synkroniseringsmekanismene i C/POSIX, Java eller Ada som du foretrekker.

1-71) (Used in: v2013)

Vi tenker oss et lager med begrenset størrelse, hvor elementer kan puttes inn og tas ut. To operasjoner skal implementeres (pseudokode java m. $wait()$, $notify()$ og $notifyall()$), $void\ put(Element\ e)$ og $Element\ get()$. Fyll ut funksjonene gitt under.

Håndteringen av elementene, eller strategien for hvilket element $get()$ skal returnere er ikke viktig. Du trenger bare å vise synkroniseringen: *put* skal blokkere hvis lageret er fullt, og *get* skal blokkere hvis det er tomt.

```
1 public class FiniteCapacityStorage {
```

```
2
3       private int current_storage_size;
4       final int   storage_capacity = 50;
5       // Make assumptions here as necessary.
6
7       public  FiniteCapacityStorage()
8       {
9           this.current_storage_size = 0;
10      }
11
12      public synchronized void put(Element new_item)
13      {
14          // You do the rest!
15      }
16
17      public synchronized Element get()
18      {
19          Element dummy = 0;
20
21          // You do the rest!
22
23          return dummy;
24      }
25  }
```

For å øke abstraksjonsnivået og gi høyere fleksibilitet til synkroniseringsmekansmene i forhold til semaforer, har POSIX, Java og Ada landet på forskjellige varianter av monitorer: POSIX lar en kombinere mutexer og condition variable, Java har synchronized metoder og *wait*/*notify*/*notifyAll*, og Ada har guardede entries i protected objects.

1-72) (Used in: v2013 , v2011kont , h2009)

(2x) Beskriv kort hvordan disse tre virker.

1-73) (Used in: v2013 , v2011kont , h2009)

Sammenlign Java og Ada sine mekanismer her, angi de viktigste fordelene og ulempene.

1-74) (Used in: v2013 , h2010)

Nevn noen maskinvare- (/maskinkode-) mekanismer som det er vanlig å bruke for å oppnå grunnleggende synkronisering.

1-75) (Used in: v2015 , v2013 , v2011kont , h2010)

Suspend() og *Resume*(*Thread*)-kallene har vært beskrevet som ubrukelige til å lage trygg synkronisering imellom tråder. Forklar hvorfor.

1-76) (Used in: v2013 , h2010)

Javas *wait* og *notify* (og POSIX condition variables) ligner en del på *suspend* og *resume*. Hvordan kan disse anses som bedre eller høyerenivå synkroniseringsmekanismer?

1-77) (Used in: v2013 , v2011kont , h2010)

"Resumption model" brukes for å beskrive mulige implementasjoner av både signaler, asynchronous notification generelt og exceptions. I mange sammenhenger er det ansett som mindre egnet enn "Termination model". Hva er forskjellen? Hvorfor er Resumption model mindre egnet?

1-78) (Used in: kont2015 , v2013 , v2011kont , h2010 , h2009)

Et generelt råd ved Java-programmering er å unngå $notify()$-kallet til fordel for $notifyAll()$ kombinert med while-løkker rundt $wait()$-setningene. Hvorfor?

1-79) (Used in: v2013kont)

I et sanntids programsystem har vi gjerne flere paralellt kjørende, samarbeidende tråder. Hvorfor er dette en rimelig måte å organisere funksjonaliteten i et sanntidssystem?

1-80) (Used in: v2013kont)

Det at trådene er samarbeidende resulterer i at trådene har et visst behov for interaksjon. Denne interaksjonen løser vi gjerne enten ved "synkronisering" eller ved "kommunikasjon". Forklar kort og/eller gi eksempler på disse to mekanismene.

1-81) (Used in: v2013kont)

Sammenlign kort og punktvis de to ut fra hvor egnet de er for bruk i et sanntidssystem.

1-82) (Used in: v2013 , h2010)

Feildeteksjon — det å stadfeste at noe er feil — kan gjøres på flere måter enn å sjekke feilreturer. List opp prinsipper som kan brukes til å detektere feil.

1-83) (Used in: v2013)

Vi ønsker for en Atomic Action å ha kontroll på deltagerene og utstrekningen av aksjonen i tid. Den synkroniseringen som er nødvendig for dette kan gis gjenbrukbar form og legges inn i et ActionController objekt.

I koden under er "entryProtokolen" som holder rede på om de to deltagerene er ankommet, og styrer at vi ikke har flere deltagere samtidig i hver rolle, ferdig implementert.

Du skal implementere (java pseudo-code m. $wait()$, $notify()$ og $notifyAll()$-kallene) den manglende koden i exitProtookollen som sikrer at begge deltagerne avslutter ak-

sjonen på samme tid.

```
1  public class ActionController {
2
3      Boolean firstHere, secondHere;
4      int allDone; // Better name: doneCounter ?
5      final int numberOfParticipants = 2;
6
7
8      public ActionController()
9      {
10
11         firstHere = secondHere = false;
12         allDone = 0;
13
14     }
15
16     public synchronized void entryProtocolOne()
17     {
18         while(firstHere) wait();
19         firstHere = true;
20     }
21     public synchronized void entryProtocolTwo()
22     {
23         while(secondHere) wait();
24         secondHere = true;
25     }
26
27     synchronized void exitProtocolBoth()
28     {
29         // You do the rest!
30     }
31
32 }
```

1-84) (Used in: v2013)

Hva oppnår vi, feilhåndteringsmessig, ved å bruke statisk redundans?

1-85) (Used in: v2013)

Hva oppnår vi, feilhåndteringsmessig, ved å bruke dynamisk redundans?

1-86) (Used in: h2009)

Backward error recovery anses av og til som ukurant i forhold til sanntidsprogrammering. Hva er backward error recovery og hvorfor er det ukurant?

1-87) (Used in: h2009)

Hva er forward error recovery?

Både forward og backward error recovery kan generaliseres fra entråds systemer til multitrådssystemer.

1-88) (Used in: h2009)

Backward error recovery kan ved generalisering til multitrådssystemer gi dominoeffekten. Forklar. Hvordan kan dominoeffekten unngås?

1-89) (Used in: kont2014)

Backward error recovery kan ved generalisering til multitrådssystemer gi dominoeffekten. Forklar. Hvordan kan dominoeffekten unngås?

1-90) (Used in: h2009)

Hvis vi vil ha forward error recovery i en multitråd setting oppstår behovet for at de forskjellige trådene kjenner til feil som oppstår i andre tråder. List mekanismer som kan brukes til å formidle slik informasjon.

1-91) (Used in: h2009)

Det å skrive til log er et alternativ til å lage recovery points. Hvordan virker dette i kontekst av (singel-tråd) backward error recovery?

1-92) (Used in: kont2014)

Det å skrive til log kan også være et alternativ til å lage recovery points. (!) Hvordan virker dette i kontekst av (singel-tråd) backward error recovery?

1-93) (Used in: h2009)

Håndtering av log kan bli litt mer uoversiktlig hvis vi ser for oss flere parallellt pågående oppgaver, hvor noen lykkes og noen mislykkes og alle genererer log. Hvordan kan vi generalisere log'en til å håndtere dette (fortsatt i kontekst av backward recovery)?

1-94) (Used in: v2014 , v2013 , h2009)

Hva er en Atomic Action? Hvilke(t) problem(er) er Atomic Actions ment å løse?

1-95) (Used in: v2014)

Beskriv en måte å implementere en Atomic Action på.

1-96) (Used in: v2014)

Operasjoner som låser ressurser blir alltid antatt å være atomiske i seg selv. Hvorfor er dette så viktig?

1-97) (Used in: h2010)

Hva er fordelene med å strukturere funksjonaliteten i programmet ditt i Atomic Actions? Ser du noen ulemper?

1-98) (Used in: v2015 , h2009)

Gi en kort beskrivelse av hvordan prosesspar virker.

1-99) (Used in: v2011kont , h2010)

Prosesspar er en teknikk for å lage tilgjengelige (i tillegg til pålitelige) prosesser. Hvilke av de følgende punktene er sentrale byggestener/prinsipper for å implementere prosesspar?

1 En "I Am Alive" protokoll som lar backupprosessen vite at primærprosessen er oppe og går.

2 En (tofase) commit protokoll imellom primær og backup som sørger for at de har et konsistent verdensbilde.

3 Det å sende tilstandsoppdateringer fra primæren til backupen slik at backupen kan starte i en riktig tilstand hvis den må ta over som primær.

4 Pålitelig kommunikasjon imellom backup og primær som sørger for at alle meldinger kommer frem og i riktig rekkefølge.

5 En mekanisme for å la klientene forholde seg til den (av de to) prosessene som er aktiv primær (som å sende "IAmPrimary" også til klientene).

6 Meldingskøer som er uavhengig av prosessene som sørger for at meldinger ikke går tapt selv om prosessene restartes.

"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 strengere krav til pålitelighet av 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.

1-100) (Used in: v2015 , v2011)

Hva kan vi gjøre for å detektere *uforutsette* feil ? Forklar kort.

1-101) (Used in: v2015 , v2011)

Gitt at vi har detektert en slik uforutsett feil... Hvordan kan vi vite hva som trengs å gjøres for å håndtere feilen ?

1-102) (Used in: v2015 , v2011kont , v2011)

Feilmodene til et system er de måtene systemet kan feile på. Det å "slå sammen feilmodi" er en teknikk: Hva kan vi oppnå ved å slå sammen feilmodi ?

1-103) (Used in: kont2014 , v2013kont)

Forklar begrepene "backward error recovery" og "recovery points".

1-104) (Used in: v2013kont)

Forklar dominoeffekten.

1-105) (Used in: v2013kont)

Hvordan kan vi unngå dominoeffekten?

1-106) (Used in: v2013kont)

Hvis programtilstanden er stor, eller vi har flere samtidig pågående operasjoner som kan feile hver for seg, så er kanskje *log* et godt alternativ til å lagre unna recovery points. Hvordan fungerer dette i kontekst av backwards error recovery?

1-107) (Used in: v2013kont)

Noe annet vi kan bruke loggen til er å legge tilrette for at et program kan komme opp i en riktig, konsistent tilstand etter å ha blitt restartet. Hvordan?

1-108) (Used in: v2013kont)

Etter at et slikt program har kjørt en stund så kan vi se for oss at loggen kan bli vilkårlig stor. "Checkpoints" er et design-mønster som lar oss slette gammel log. Hvordan virker dette?

1-109) (Used in: kont2014)

Etter at et slikt program har kjørt en stund så kan vi se for oss at loggen kan bli vilkårlig stor. "Checkpoints" er et design-mønster som lar oss slette gammel log. Hvordan virker dette? (Husk at ordet "Checkpoint" brukes om flere forskjellige ting i forskjellige deleer av pensum.)

En Atomic Action har start, side og end boundaries.

1-110) (Used in: v2015 , v2011kont)

Hva er formålet med disse avgrensningene ?

1-111) (Used in: v2015 , v2011kont)

Hvordan kan hver av dem (start, side og end) realiseres ?

Asynkron notification - det å avbryte tråder i det de holder på med for å formidle noe, kommer inn i forbindelse med Atomic Actions.

1-112) (Used in: v2011kont)

Hvorfor ? Hvorfor kommer Asynkron Notification opp som et tema ifbm. Atomic Actions ?

1-113) (Used in: v2014)

Hva er "Asynchronous Transfer of Control"?

1-114) (Used in: v2014)

Hvorfor er Asynchronous Transfer of Control sett som relevant for implementasjon av Atomic Actions?

1-115) (Used in: v2011kont)

Hvordan kan vi skaffe oss "termintaion mode" Asynch transfer of control i POSIX / C ? Hva med Java(/RT Java) ? Og ADA ?

Gray hadde en tanke om at vi kunne bygge feiltolerante datasystemer hvis vi bare hadde pålitelig lagring, kommunikasjon og beregninger...

1-116) (Used in: v2011)

Gjengi kort hvordan vi kan sikre oss pålitelig datalagring (altså bygd over datalagring som ikke er pålitelig.)

1-117) (Used in: v2011)

Gjengi kort hvordan vi tilsvarende kan sikre oss pålitelig kommunikasjon.

1-118) (Used in: v2011)

Pålitelige beregninger er den vanskeligste i denne trioen siden feilmodene er vanskelig

å beskrive godt for et generelt program. Pensum setter først opp "Checkpoint/Restart" som et alternativ: Beskriv kort hvordan dette virker. Hva er feilmodene og hvordan detekteres de?

Gray hadde en tanke om at vi kunne bygge feiltolerante datasystemer hvis vi bare hadde pålitelig lagring, kommunikasjon og beregninger...

1-119) (Used in: kont2014)

Pålitelige beregninger er den vanskeligste i denne trioen siden feilmodene er vanskelige å beskrive godt for et generelt program. Pensum setter først opp "Checkpoint/Restart" som et alternativ: Beskriv kort hvordan dette virker.

1-120) (Used in: kont2014)

Et enkelt og greit alternativ til slike sjekkpunkter er å skrive til "log". Hvordan virker dette og hva oppnår vi i forhold til sjekkpunktene?

1-121) (Used in: kont2014)

Hva er i denne konteksten feilmoden(e) til en "beregning" og hvordan detekteres feil?

Gray hadde en tanke om at vi kunne bygge feiltolerante datasystemer hvis vi bare hadde pålitelig lagring, kommunikasjon og beregninger...

1-122) (Used in:

Pålitelige beregninger er den vanskeligste i denne trioen siden feilmodene er vanskelig å beskrive godt for et generelt program. Pensum setter først opp "Checkpoint/Restart" som et alternativ: Beskriv kort hvordan dette virker. Hva er feilmodene og hvordan detekteres de?

1-123) (Used in: kont2014 , v2011) **(2x)** En annen måte å skaffe oss pålitelige beregninger på er prosesspar. Beskriv hvordan prosesspar virker. Hva får vi her i forhold til checkpoint/restart løsningen?

1-124) (Used in: kont2014) "Transaksjoner" er et begrep som er nesten det samme som "Atomic Actions" i læreboken og som er relevant når vi skal bygge feiltolerante datasystemer. Hvordan bidrar transaksjoner til feiltoleranse?

1-125) (Used in: kont2014)

Hva er forskjellen på en Transaksjon og en Atomic Action? Hvordan gjør denne forskjellen Atomic Actions mer attraktive å bruke i sanntidssystemer?

1-126) (Used in: kont2014)

Grays tredje forslag for å skaffe oss pålitelige beregninger forutsetter at alle data (hele programmets tilstand) er trygt lagret i en database, at alle beregninger er formulert som transaksjoner, og at vi har et system for å kjøre disse transaksjonene som også automatisk håndterer feil som oppstår.

I tillegg til svaret i forrige deloppgave; Hvorfor lager vi ikke alle (embedded/sanntids) systemer slik?

1-127) (Used in: v2013kont)

Hva er det vi oppnår ved å bruke prosesspar?

1-128) (Used in: v2013kont)

Forklar kort hvordan prosesspar virker.

Et antall teknikker/design patterns er blitt satt opp for å håndtere en del problemstillinger omkring feilhåndtering og konsistens i multitråd eller distribuerte systemer:

- Writing log records to the log
- Writing Checkpoints to the log
- The two-phase commit protocol
- Two-phase execution (growing and shrinking phases)
- Introducing a transaction manager
- (Read/Write-) Locking of accessed resources.
- Introducing a LogManager
- Introducing a LockManager

For hver av oppgavene under skal du plukke ut den mest sentrale teknikken for å oppnå det oppgaven spør om.

1-129) (Used in:

For det programmet du skriver er det viktig at programmet holder rede på hva det har gjort, slik at det kan starte i en riktig tilstand etter en restart.

1-130) (Used in:

Etter at du har skrevet 3 forskjellige Atomic Actions legger du merke til at du har kode for å koordinere start og end boundaries på 3 forskjellige steder i koden din. Du ønsker å samle denne funksjonaliteten i en gjenbrukbar modul.

1-131) (Used in:

Du har bestemt deg for at at å låse ressurser i økende og minkende faser er en god ide for å holde rede på de mulige konsekvensene av feil. Men du oppdager at du trenger å koordinere alle trådene i overgangen imellom de to fasene.

1-132) (Used in: v2011kont , h2009)

Bruk av exceptions i et språk som støtter det (f.eks. Java) forenkler/forbedrer en del ting. Nevn hva fordelene med exceptions er: Hva blir enklere/bedre ved bruk av exceptions?

1-133) (Used in: v2011kont , h2009)

Finnes det noen ulemper ved å ta i bruk exceptions i et programsystem, sanntidssystem eller programmeringsspråk? List kort.

1-134) (Used in: v2011kont , h2010)

I et regime med unntakshåndtering (exceptions) kan følgende kodebit gi oss problemer (forårsake vranglås):

```
{
  Wait(semaphoreA);
  f();
  Signal(semaphoreA);
}
```

Forklar hva problemet er og hvordan det kan løses.

1-135) (Used in: v2013kont)

En del moderne programmeringsspråk kommer med innebygde mekanismer for "exception handling". Hva er det vi ønsker å oppnå med disse mekanismene ?

1-136) (Used in: v2013kont)

Forklar kort hvordan exceptionhandling fungerer (Velg gjerne C++, java eller ADA som eksempelspråk om du vil).

1-137) (Used in: v2013kont)

Hvilke ulemper er knyttet til bruk av disse mekanismene i et programsystem (/sanntidssystem)?

Vi har satt opp følgende krav til *modulbegreper* – en måte å dele et system i moduler på:

- Vi må kunne vedlikeholde en modul uten å kjenne bruksmønstrene.
- Vi må kunne bruke en modul uten å kjenne indre virkemåte.
- Composition: Supermoduler må kunne lages av submoduler.

Ut ifra dette: Vurder (kort) de følgende modulbegrepene. Vil de være egnet til å lage store programsystemer med ?

1-138) (Used in: v2011kont)

Tråder, som i Java eller POSIX.

1-139) (Used in: v2011kont)

Monitorer

1-140) (Used in: v2011kont)

Kommuniserende prosesser (som i OCCAM eller Google Go - Meldingbasert system, ingen delte ressurser).

1-141) (Used in: v2011kont)

Atomic Actions

Vi har følgende tre FSP prosesser:

$$P = (a \to b \to c \to P).$$
$$Q = (a \to b \to d \to Q).$$
$$R = (d \to b \to c \to R).$$

En mulig tolkning av et event som to prosesser deltar i kan være at det er (synkron) kommunikasjon dem imellom. Hvis det er flere deltagende prosesser så kan det være en barriere/rendezvous.

1-142) (Used in: kont2015 , h2009)

Tegn transisjonsdiagrammer for P og $P||Q$.

1-143) (Used in: kont2015 , h2009)

$P||Q||R$ går i vranglås. Hvorfor/Hva skjer? Hvordan kan en vranglås gjenkjennes i et transisjonsdiagram?

1-144) (Used in: kont2015 , v2011kont , h2010 , h2009)

Hva er en livelock, og hvordan kjennes en slik igjen i et transisjonsdiagram?

1-145) (Used in: kont2015 , h2009)

Modeller følgende semafor-program i FSP.

```
T1(){
  while(1){
    wait(A);
    wait(M);
    ...
    signal(M);
    signal(A);
  }
}
```

```
T2(){
  while(1){
    wait(B);
    wait(M);
    ...
    signal(M);
    signal(B);
  }
}
```

En rimelig måte å skaffe seg synkron kommunikasjon på basert på semaforer kan være som i de følgende programsnuttene hvor tråd 1 sender en sekvens av heltall til tråd 2.

```
1
2  int ch; // The channel
3  Semaphore ch_sendReady(0),ch_receiveReady(0);
4
5  thread1(){
6    int i=0;
7    while(1){
8
9      // send i
10     ch = i;
11     ch_sendReady.signal();
12     ch_receiveReady.wait();
13     i++;
14   }
15 }
16
17 thread2(){
18   int i;
19   while(1){
20
21     // receive i
22     ch_sendReady.wait();
23     i = ch;
24     ch_receiveReady.signal();
25
26   }
27 }
```

1-146) (Used in: v2013)

Modeller systemet i FSP (Finite State Process-notasjon) - se den vedlagte referansen hvis det er nødvendig.

1-147) (Used in: v2013)

Tegn transisjonsdiagrammet for hele systemet. Hvor mange tilstander kan systemet være i?

Hvis vi forestiller oss et språk med synkron komunikasjon innebygd i språket, som Go eller Occam, så kan tilsvarende kode se slik ut:

```
1  T1(int channel ch){
2    while(1){
3      int i;
4      ch <- i;
5      i++;
6    }
7  }
8
9  T2(int channel ch){
10   while(1){
11     int i;
12     i <- ch;
13     //...
14   }
15 }
```

1-148) (Used in: v2013)

Modeller dette systemet i FSP. (Hint: Forvent en ganske enkel modell)

1-149) (Used in: v2013)

Tegn transisjonsdiagrammet for hele systemet. Hvor mange tilstander kan systemet være i? (Hint: Færre enn sist)

1-150) (Used in: v2013)

Vi har i begge implementasjonene to tråder som overleverer informasjon fra den ene til den andre her: Reflekter kort over hvorfor den ene modellen er enklere enn den andre.

Hvis vi tenker oss et system av tråder som sender meldinger imellom seg ved bruk av to semaforer per kanal som i oppgaven over, så kan antallet semaforer bli veldig stort i systemet.

1-151) (Used in: v2013)

Gi et eksempel på hvordan vranglåser kan oppstå i et slikt meldingssendende system.

1-152) (Used in: v2013)

Og spesielt for det meldingssendende systemet vårt: Av punktene dine i forrige spørsmål; Hvilke anser du som mest relevant for dette systemet ? (Evt. om det er andre ting du vil nevne for å sikre fravær av vranglåser...)

1-153) (Used in:

Var dette en tilfeldighet?, eller kan vi si at systemer basert på synkron meldingssending gir opphav til enklere systemer (med færre mulige tilstander) enn synkroniserende systemer ? Reflekter kort.

Et klassisk eksempel på et system med en vranglås er "dining philosophers", hvor N filosofer sitter ved et rundt bord og spiser. Hver trenger to gafler for å spise og må dele med begge sidemennene de N gaflene som ligger fordelt imellom filosofene.

1-154) (Used in: kont2014 , v2013kont , v2011)

Vis hvordan filosofene og gaflene kan modelleres i FSP.

1-155) (Used in: kont2014 , v2013kont , v2011)

Tegn transisjonsdiagram for en filosof og for en gaffel.

1-156) (Used in: kont2014 , v2013kont , v2011)

Tegn transisjonsdiagrammet for et totalsystem med to filosofer og to gafler. Pek på vranglåsen.

1-157) (Used in: h2010)

Boken setter opp fire betingelser som alle må være oppfyllt for at en vranglås skal kunne inntreffe. Hvilke var dette? Og hvordan kan en unngå vranglåser ved å (for hver av dem) sørge for at de ikke blir oppfyllt?

1-158) (Used in: h2010)

I tradisjonelle sanntidssystemer vet vi ofte nøyaktig hvilke tråder som kan trenge hvilke ressurser. Dette åpner for noen teknikker som lar unngå problemer med vranglåser.

1-159) (Used in: h2010)

Imidlertid er det ikke alltid mulig å forutse hvilke ressurser alle tråder trenger på forhånd; La oss se for oss et system av intelligente sensorkjøretøyer som deler informasjon via en basestasjon, eller et prosess- styresystem hvor operatørene selv kan programme-

re inn alarm-situasjoner med tilhørende aksjoner i form av skript. I begge disse tilfellene kan basen/systemet få ikke-forutsigelige ressursallokeringer.

Hvordan kan vi unngå eller løse vranglåsproblemer i slike systemer?

1-160) (Used in: v2013kont , v2011)

Hva er en vranglås ? Gi et eksempel med semaforer.

1-161) (Used in: v2011)

Vranglåser kan selvfølgelig også oppstå i et system hvor interaksjon imellom trådene er basert på meldingssending. Konstruer et enkelt eksempel herifra.

1-162) (Used in: scheduling , v2013kont)

Riktig scheduling er viktig i sanntidsprogrammering for å sikre at alle tidsfrister i systemet blir nådd. Hvordan kan scheduling-strategier legge til rette for dette?

1-163) (Used in: kont2015 , kont2014 , v2013kont)

(2x) Vi kan bevise at tidsfristene blir nådd i et system ved responstidsanalyse og utilization analyse. Forklar kort hvordan disse to virker.

1-164) (Used in: kont2015 , v2013kont)

Det ligger en del nødvendige antagelser til grunn for at utilization analyse skal være anvendbart. ("The simple task model") Hvilke ?

1-165) (Used in: v2014)

Det er en del nødvendige antagelser til grunn for at utilization analyse skal være anvendbart. ("The simple task model") Hvilke ? Kommenter kort på hvor realistiske de er.

1-166) (Used in: kont2015)

Det er nok ikke for alle sanntidssystemer ute i verden at schedulerbarhetsbevisene faktisk blir gjennomført. Hvorfor tror du industrien er tilbakeholdne med å gjennomføre disse bevisene?

1-167) (Used in: v2013kont)

Hva er "Priority Inversion" ?

1-168) (Used in: v2013kont)

Hvordan påvirker priority inversion problemet schedulerbarhetsanalysene?

1-169) (Used in: kont2014 , v2013)

Hvorfor er "scheduling" så viktig i sanntidsprogrammering? — Hvilke egenskaper er det vi ønsker at systemet vårt skal få ved å velge riktige schedulingsstrategier?

1-170) (Used in: kont2014 , v2013)

Vi tenker oss et klassisk sanntidssystem med tråder som synkroniserer vha. semaforer: Hvordan bestemmes det (typisk) hvem av trådene som får kjøre (for å oppnå målene i forrige punkt)?

1-171) (Used in: v2013)

I det samme systemet: Hvordan bestemmes det (typisk) hvilken av flere ventende tråder som blir vekket opp når en semafor blir ledig ?

1-172) (Used in: kont2014)

I det samme systemet: Hvordan bestemmes det (typisk) hvilken av flere ventende tråder som blir vekket opp når en semafor blir ledig ? Hvorfor?

1-173) (Used in: v2013)

Hva er "unbounded priority inversion" problemet?

1-174) (Used in: v2013)

Hvordan kan "unbounded priority inversion" problemet løses ved hjelp av schedulingstrategier?

Gitt disse tre oppgavene:

Task	Period	Execution time
A	80	20
B	60	20
C	40	10

1-175) (Used in: v2014 , v2013)

Vis på en grafisk måte hvordan oppgavene A, B og C i tabellen over vil kjøre under rate monotonic priority assignment. Marker tydelig dersom systemet evt. feiler i å nå tidsfrister. (Anta at alle oppgaver blir klare første gang ved tid 0)

1-176) (Used in:

Hvor langt frem i tid var det egentlig nødvendig å sjekke for feilede tidsfrister ?

Jeg droppet denne i 2013

1-177) (Used in: v2014 , v2013)

Foreta en utilization test for dette systemet.

$$\left(Hint : U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N(2^{1/N} - 1) \right)$$

1-178) (Used in: v2014 , v2013)

Foreta en responstidsanalyse test for dette systemet.

$$\left(Hint : w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \right)$$

Du skal lage et sanntidssystem, dvs. at du har sanntidskrav til systemet, og en plikt på deg til å kunne argumentere for at systemet vil nå sanntidskravene sine. (Dvs. ut over at "du har testet systemet, og det ser ut til å virke")

1-179) (Used in: v2015 , v2011)

Hvordan vil du strukturere systemet for å legge til rette for en slik argumentasjon ? (Mao. kan du si noe generelt om programvaredesignet?)

1-180) (Used in: v2015 , v2011)

Og til selve argumentene: Skisser hvordan vil du legge an argumentasjonen.

1-181) (Used in: v2015 , v2011)

Interaksjon imellom forskjellige parallelle oppgaver vil kunne føre til "inversion"-problemer. (Eks Priority inversion) Forklar hva problemet er.

1-182) (Used in: v2015 , v2011)

Forklar kort hvordan en Priority Ceiling protokoll fungerer. (Dvs. velg en av dem.)

To velkjente schedulingsstrategier er FPS (Fixed Priority Scheduling) og EDF (Earliest Deadline First).

1-183) (Used in: v2011)

Forklar begrepene FPS og EDF.

1-184) (Used in: v2011)

Hvorfor er FPS mest brukt ?

Vi har startet et antall tråder $T_1 ... T_N$. T_1 initialiserer en del datastrukturer som alle trådene bruker slik at disse skal blokkeres helt til initialiseringen er ferdig. En måte å gjøre dette på med semaforer er den følgende: (A er en binær semafor som er initialisert til 0.)

```
T1(){
  // Do the initialization
  signal(A);
  // Continue working
}
```

```
T2(){
  wait(A);
  signal(A);
  // Data structures are initialized,
  // continue working
}
```

$T_3 ... T_N$ er implementert som T_2.

1-185) (Used in: v2011kont , h2010)

Anta at N=3. Modeller systemet i FSP. (Hvis du vil unngå problemstillingene omkring terminerende prosesser, så modeller "continue working" som $DW = (doWork \rightarrow DW)$.) Se vedlagte FSP referanse hvis nødvendig.

1-186) (Used in: v2011kont , h2010)

Med N fortsatt 3, tegn transisjonsdiagrammet for totalsystemet.

1-187) (Used in: v2011kont , h2010)

(Implementasjon/pseudokode) Hvordan ville du løse problemet med å holde tilbake trådene $T_2 ... T_N$ i Java (ved å bruke synchronized, wait, notify og notifyAll)?

1-188) (Used in: v2011kont , h2010)

(Implementasjon/Pseudokode) Hvordan ville du løse problemet med å holde tilbake trådene $T_2 ... T_N$ i Ada (protected objects, funksjoner, prosedyrer, entries, guarder.)?

Se på den gitte semaforversjonen av programmet igjen. Dette programmet etterlater semaforen A i åpen tilstand. Nå ønsker vi å gjenbruke semaforen og denne signaliseringen av at initialiseringen er ferdig, og må dermed resette semaforen. (Se for deg en initialisering som må skje flere ganger mens programmet kjører.)

1-189) (Used in: v2011kont , h2010)

Nå ser programmet slik ut. Anta at *g_N* og *g_nOfThreadsInitialized* er globale variable som er initialisert til hhv. antall tråder og 0.

```
T1(){
  while(1){
    // Do the initialization
    signal(A);
    g_nOfThreadsInitialized++;
    if(g_nOfThreadsInitialized == g_N){
      g_nOfThreadsInitialized = 0;
      wait(A);
    }
    // Continue working

  }
}
```

```
T2(){
  while(1){
    wait(A);
    signal(A);
    g_nOfThreadsInitialized++;
    if(g_nOfThreadsInitialized == g_N){
      g_nOfThreadsInitialized = 0;
      wait(A);
    }
    // Data structures are initialized,
    // continue working
  }
}
```

Dette programmet har en race-condition (minst). Hva er en race condition? Hva er problemet(-ene)?

Du skal skrive en ressursmanager som forvalter et stykke minne med begrenset størrelse. Kallet til $void * Allocate(int\ nOfBytes)$ 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.

1-190) (Used in: v2013kont , h2010)

(Implementasjon/pseudokode) Du skal bruke delt variabelsynkronisering, men kan bruke hvilke mekanismer for dette som du vil.

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

1-191) (Used in: v2013kont , h2010)

Hva er utsulting (starvation)?

1-192) (Used in: v2013kont , h2010)

Kommenter kort på faren for utsulting ift. minneallokeringsoppgaven over. (Hvis du ikke hadde en implementasjon over som var fri for utsultingsproblemer:) Foreslå en strategi for å prioritere imellom flere ventende Allocate-kall som ikke gir denne faren.

1-193) (Used in: v2013kont)

Vi tenker oss et klassisk sanntidssystem med prioriterte tråder som synkroniserer vha. semaforer: Hvordan bestemmes det (typisk) hvilken av flere ventende tråder som blir vekket opp når en semafor blir ledig?

1-194) (Used in: v2013kont , h2010)

Adas menkanismer (protected objects m. funksjoner, prosedyrer og entries med guarder) er ikke veldig egnet til å løse akkurat denne minneallokeringsoppgaven... Hvorfor?

1-195) (Used in:

(implementasjon/Pseudokode) Skisser hvordan designet av denne minneallokerings problemstillingen kunne tatt seg ut i et meldingsbasert regime (Allocate & Free er ikke nødvendigvis prosedyrer, du har synkron kommunikasjon, (arrayer av) kanaler, select/alt med guards på kanalene & fingranulære/veldig lette prosesser.)

Du har allerede laget en modul i C som holder rede på personnavn. Den har bl.a. funksjonene *getFirstName*, *getLastName*, *setFirstName* og *getName*. Den siste kaller de to første for å sette sammen hele navnet før den returnerer. Modulen fungerte feilfritt helt til det ble laget en multitrådet versjon av programmet...

1-196) (Used in: v2011kont , v2011)

Hva slags problemer med en slik modul kan komme til overflaten når programmet gjøres multitrådet?

1-197) (Used in: v2011)

I hvert fall bestemmer du deg for å beskytte tilgangen til modulen med semaforer ved å allokere (*wait*) en gitt binærsemafor i begynnelsen av hver funksjon, og frigi den (*signal*) rett før funksjonene returnerer. Dette funker ikke: Hva skjer ?

1-198) (Used in: v2011)

Hvordan ville du fikset dette problemet? (Du har fortsatt bare semaforer til rådighet for synkronisering.)

1-199) (Used in: v2011)

Beskriv kort hvordan Javas monitorlignende mekanismer for synkronisering virker ("synchronized objects").

1-200) (Used in: v2011)

(Java pseudokode) Du skal lage en tilsvarende modul i Java: Skisser hvordan de fire funksjonene her kan bli beskyttet mot samtidig aksess fra flere tråder.

1-201) (Used in: v2011)

Beskriv kort hvordan Adas monitorlignende mekanismer for synkronisering virker ("protected objects").

1-202) (Used in: v2011)

(Ada pseudokode) Du skal lage nå modulen i Ada: Skisser hvordan de fire funksjonene her kan bli beskyttet mot samtidig aksess fra flere tråder. Har denne løsningen vesentlige fordeler ift. Java-løsningen ?

1-203) (Used in: v2011)

Det å begrense tilgangen til en modul til bare en tråd i gangen er kanskje unødvendig: Anta at mange av funksjonene i modulen faktisk kunne kalles samtidig av flere tråder uten problemer. Hva er argumentene for å evt. lage en litt mer avansert tilgangskontroll til modulen for å tillate dette ?

1-204) (Used in: v2011kont , v2011)

Nestede kall til monitorer (det at en funksjon i en monitor kaller en funksjon i en annen monitor) er et problem som fører til at systemer basert på monitorer ikke skalerer veldig godt... Hva er problemet ?

1-205) (Used in: v2011)

Hvordan har Ada unngått problemstillingen med nestede monitorkall ?

1-206) (Used in: v2011)

Inheritance Anomaly er et problem som melder seg når vi vurderer å arve ned klasser (i et objektorientert regime som C++ eller Java) hvor metodene synkroniserer seg. Forklar kort hvor problemet ligger.

1-207) (Used in: v2011)

Jeg har antydet at systemer basert på delt variabelbasert synkronisering skalerer dårlig. Kan du underbygge dette ?

1-208) (Used in: kont2015) **(7x)** Flere teknikker som kan bidra til feiltoleranse har blitt presentert i løpet av faget. For hver av de under: Beskriv kort hvordan de virker/hva de innebærer og hvordan de bidrar til feiltoleranse.

- Akseptansetester
- Sammenslåing av feilmoder
- Statisk Redundans

- Dynamisk Redundans
- Recovery Points
- Backward Error Recovery
- Forward Error Recovery
- N-versjons programering
- Recovery Blocks
- Atomic Actions
- Transaksjoner
- Prosesspar
- Error Injection
- Fail Fast
- Checkpoint/Restart

1-209) (Used in: kont2015)

Sanntids programvare er vel og bra, men ingen sanntidskrav blir oppfyllt når programmet ikke kjører.

En av de vanskelige egenskapene å oppnå i et sanntidssystem er muligheten av å oppgradere programvaren til ny versjon, uten at vi får avbrudd i tjenesten.

Fortell kort hvordan du ville nærmet deg designet av et system med denne egenskapen.

Lese/skrive-låser er en variant av mutual exclusion hvor vi innser at vi ikke trenger mutual exclusion imellom flere "lesere". "Skriverne" skal fortsatt kjøre under mutual exclusion, med hverandre og med leserne.

1-210) (Used in: kont2015)

Hva er det som gjør at vi ikke trenger mutual exclusion imellom flere lesere, når det er nødvendig imellom skriverne?

1-211) (Used in: kont2015)

Hvorfor/Når ville vi foretrekke lese/skrivelåser fremfor vanlig mutual exclusion?

1-212) (Used in: kont2015)

Med Adas Protected Objects er lese/skrive-låser veldig lette å lage. Forklar hvordan.

1-213) (Used in: kont2015) **(2x)** En liten utfordring: Skisser (pseudokode) hvordan en kan lage lese/skrive-låser med Javas synkroniseringsprimitiver (synchronized methods, wait, notify, notifyAll). Hint: Lag funksjonene startRead, stopRead, startWrite og stopWrite.

1-214) (Used in: kont2015)

Hvis vi ser for oss veldig mange lesere og skrivere så kan imidlertid slik bruk av $notifyAll$ være uheldig. Hvorfor?

1-215) (Used in: kont2015) **(2x)** En større utfordring: Skisser (pseudokode) hvordan en kan lage lese/skrive-låser med semaforer.

1-216) (Used in: kont2015)

Hva er utsulting (starvation)?

1-217) (Used in: kont2015)

Lese/Skrivelåser er et interessant utgangspunkt for å diskutere utsulting. Hvorfor?

1-218) (Used in: kont2015)

Hvis du mener at en av løsningene i 1-4 eller 1-7 er trygge mot utsulting; Forklar hvorfor. Hvis ikke; foreslå (Pseudokode ikke nødvendig) hvordan en kunne lage en versjon som unngår utsulting. (for enten Java- eller semaforer-versjonen)

1-219) (Used in: kont2015)

Anta at Ada scheduleren virker på en måte som gjør at den trivielle implementasjonen i oppgave 1-3 av lese/skrive låser faktisk beskytter mot utsulting. Hva vil du si om denne implementasjonen i et kodekvalitetsperspektiv?

En enkel sammenligning imellom programmeringsspråket Google Go og det halvdøde språket Occam avslørte følgende (for noen år siden): Begge programer hadde to tråder på samme prioritet; den ene skrev ut 'A'er og den andre 'B'er.

Google Go-programmet skrev ut en nesten-regulær serie med A'er og B'er: ABABABABAABABABABABBAABABAB...

Occam-programmet skrev ut "ABAAAAAAAAAAAAAAAAAA..." (resten A'er)

1-220) (Used in: kont2015)

Kommenter på hvordan disse to run-time system strategiene kan gi utslag i forskjellig

kodekvalitet.

1-221) (Used in: kont2015)

En gruppe leverte for noen år siden dette heisprosjektet. Hva vil du si om inndelingen deres i moduler, modulnavning, eller om prosjektet generelt? (Kommenter på noen få punkter: de viktigste, positive eller negative)

```
   61 channels.h
   39 elevator.c
  151 elev_driver.c
   42 elev_driver.h
  344 elev_FSM.c
  139 elev_FSM.h
   86 io.c
   62 io.h
   20 Makefile
  645 network.c
  142 network.h
  171 sch_dist_calc.c
   24 sch_dist_calc.h
  296 sch_elev_manager.c
   22 sch_elev_manager.h
  135 sch_main.c
   14 sch_main.h
  304 sch_orders.c
   95 sch_orders.h
  123 sch_packets.c
   35 sch_packets.h
  145 sch_peers.c
   39 sch_peers.h
   76 sch_print_info.c
   18 sch_print_info.h
  102 sch_types.h
  188 storeOrders.c
   30 storeOrders.h
   44 timer.c
   32 timer.h
```

Tallene er antall linjer i hver fil; Hele prosjektet består av 1183 linjer med ';' noe som gjør det til et prosjekt av omtrent gjennomsnitts størrelse.

1-222) (Used in: kont2015)

Her er elev_FSM.h filen fra det samme prosjektet. Kommenter på de få viktigste punktene - både gode og dålige.

```
#ifndef __INCLUDE_ELEV_FSM_H__
#define __INCLUDE_ELEV_FSM_H__

/*
  About this module:
  A Finite State Machine controlling the elevator.
```

```
7
8     The FSM is assigned a destination floor,
9     and does best effort in reaching this floor.
10    It flags itself as ``defect'' it something goes wrong.
11
12    Known bugs:
13    - Due to different specifications in the elevator hardwares,
14    the elevator tram may sometimes drive beyond the end floors,
15    getting the elevator stuck below the ground floor, or above the top floor.
16  */
17
18  #include "elev_driver.h"
19  #include "timer.h"
20
21  #define FALSE                   0
22  #define TRUE                    1
23
24  // For the elevator FSM
25  // Directions
26  #define direction_HERE          0
27  #define direction_UP            1
28  #define direction_DOWN          2
29
30  // States
31  #define state_UP                0
32  #define state_DOWN              1
33  #define state_OPEN_DOOR         2
34  #define state_IDLE              3
35  #define state_STOP              4
36  #define state_ACCOMPLISHED      5
37  #define state_OBSTRUCTION       6
38
39  // Times (delays)
40  #define Time_doorOpen           2500        // Milliseconds
41  #define Time_brake                 5        // Milliseconds
42  #define Time_transportLimit     3000        // Milliseconds
43
44  // Velocities
45  #define speed_UP                200
46  #define speed_DOWN              -200
47  #define speed_STOP              0
48
49  /*
50    Holds all information about the elevator FSM,
51    in order to make it behave in an more or less intelligent way.
52    All of these fields are available to the module using this module,
53    but none should be altered, only read.
54  */
55  typedef struct // The elevator FSM type
56  {
57    // States
58    int previousState;
59    int currentState;
```

```
60
61    // Info about the position
62    int positionUnknown;                          // TRUE if the position is
          unknown. Only used during startup. Makes the FSM use sensors to
          determine the position
63    int atAFloor;                                  // TRUE if the elevator is
          situated at a floor (and not in a gap)
64    int currentFloor;
65    int currentFloorGap;
66
67    // Booleans describing trigger
68    // Internal (6 triggers)
69    int trig_enteringFloor;
70    int trig_leavingFloor;
71    int trig_stopButtonPressed;
72    int trig_doorTimerDownAndNotObstruction;
73    int trig_transportTimerDown;
74    int trig_obstruction;
75
76    // External (4 triggers)
77    int trig_acknowledgeTaskSuccess;        // TRUE if the function acknowledge
          () has been run from previus iteration
78    int trig_setTask;                          // TRUE if the task was updated
          since last time. Then usually destination_Floor is also updated
79    int trig_cancelTask;
80    int trig_resumeTaskAfterStop;
81
82
83    // Help variables for deciding specific branches etc
84    int destinationFloor;
85    int openDoorOnArrival;
86    int defect;
87    int defectEngine;
88
89    // Help variables
90    timerStruct doorTimer;                      // The timer for the door
91    timerStruct transportTimer;                 // The timer for the
          transportation
92    int sensor_stopButtonPrevValue;             // The stop button sensor's
          previous value
93
94    // Sensor information. In order to avoid reading the sensors more than once
           per iteration.
95    int sensor_floorSensor;
96    int sensor_stopButton;
97    int sensor_obstruction;
98  } FSM_data;
99
100
101 int init_elev_FSM(FSM_data *FSM);
102 void kill_elev_FSM(FSM_data *FSM);
103
104 /*
```

```
105   Update the FSM. Absolutly no inputs or outputs from the scheduler through
          this one,
106   just let the FSM process the info recieved from last time
107 */
108 void update_elev_FSM(FSM_data *FSM);
109
110
111 // FETCH INFO
112 /*
113   Returns TRUE if the elevator can be assigned a new task. This is only the
          case when the state is IDLE, UP or DOWN,
114   in addition to some minor conditions (like: the elevator is functional).
          This must be true when calling trig_setTask()
115 */
116 int FSM_get_canAssignNewTask(FSM_data *FSM);
117
118
119 // TRIGGERS FROM THE SCHEDULER
120 void FSM_trig_acknowledgeTaskSuccess(FSM_data *FSM);
121
122 /*
123   Sets a new destination. Must only be called when get_canAssignNewTask() is
          true.
124 */
125 void FSM_trig_setTask(FSM_data *FSM, int destination);
126
127 /*
128   Drives to the nearest floor, and enters state IDLE, without opening the
          door
129 */
130 void FSM_trig_cancelTask(FSM_data *FSM);
131
132 /*
133   Resumes the previous task after the stop button was pressed.
134   This trigger is not in use.
135 */
136 void FSM_trig_resumeTaskAfterStop(FSM_data *FSM);
137
138
139 #endif // #ifndef __INCLUDE_ELEV_FSM_H__
```

1-223) (Used in: kont2015)

Her er en funksjon fra elev_FSM.c filen fra det samme prosjektet. Kommenter på de få viktigste punktene - både gode og dålige.

```
1 /*
2         Needs to be called once per iteration.
3         This is the brain of the elevator's logic.
4 */
5 void update_elev_FSM(FSM_data *FSM) {
6   priv_fetchSensorSignals(FSM);
7   priv_internalTriggers(FSM);
```

```
8
9
10    if(FSM->trig_obstruction){
11      if (FSM->currentState != state_OBSTRUCTION)
12        priv_setNewState(FSM, state_OBSTRUCTION);
13    }
14
15    if(FSM->currentState == state_OBSTRUCTION && FSM->trig_obstruction==0)
16      priv_setNewState(FSM, state_IDLE);
17
18    if (FSM->trig_stopButtonPressed) {
19      if (FSM->currentState != state_STOP)
20        priv_setNewState(FSM, state_STOP);
21    }
22
23    if (FSM->trig_transportTimerDown) {
24      if ((FSM->currentState == state_UP) || (FSM->currentState == state_DOWN))
            {
25        FSM->defect                       = TRUE;
26        FSM->defectEngine                 = TRUE;
27        priv_elevatorStopEngine(FSM);
28      }
29    }
30
31    if (FSM->trig_doorTimerDownAndNotObstruction)
32      if ((FSM->currentState == state_OPEN_DOOR) && (!(FSM->defect)))
33        priv_setNewState(FSM, state_ACCOMPLISHED);
34
35    if (FSM->trig_enteringFloor) {
36      FSM->atAFloor        = TRUE;
37      FSM->currentFloor    = FSM->sensor_floorSensor;
38
39      if (FSM->defect)
40        priv_setNewState(FSM, state_OPEN_DOOR);
41      else
42        {
43          if (FSM->positionUnknown) {
44            FSM->positionUnknown = FALSE;
45            FSM->openDoorOnArrival = TRUE;
46            priv_setNewState(FSM, state_IDLE);
47          } else {
48            if (!FSM->openDoorOnArrival) {
49              FSM->openDoorOnArrival = TRUE;
50              priv_setNewState(FSM, state_IDLE);
51            } else {
52              if (priv_traveledTooDestination(FSM))
53                priv_setNewState(FSM, state_OPEN_DOOR);
54              if (priv_traveledTooFar(FSM)) {
55                FSM->defect = TRUE;
56                priv_elevatorStopEngine(FSM);
57              }
58              if (priv_traveledTooShort(FSM))
59                startTimer(&(FSM->transportTimer)); // Reset timer
```

```
60            }
61          }
62        }
63    }
64
65    if (FSM->trig_leavingFloor) {
66      FSM->atAFloor = FALSE;
67      if (FSM->currentState == state_UP)
68        FSM->currentFloorGap   = FSM->currentFloor;
69      if (FSM->currentState == state_DOWN)
70        FSM->currentFloorGap   = FSM->currentFloor - 1;
71
72      startTimer(&(FSM->transportTimer)); // Reset timer
73    }
74
75    if (FSM->trig_acknowledgeTaskSuccess)
76      if (FSM->currentState == state_ACCOMPLISHED)
77        {
78          FSM->trig_acknowledgeTaskSuccess = FALSE;
79          priv_setNewState(FSM, state_IDLE);
80        }
81
82    if (FSM->trig_cancelTask) {
83      FSM->trig_cancelTask = FALSE;
84      FSM->openDoorOnArrival = FALSE;
85    }
86
87    if (FSM->trig_setTask) {
88      if ((!(FSM->defect)) && ((FSM->currentState == state_IDLE) || (FSM->
           currentState == state_STOP) ||
89                               (FSM->currentState == state_UP) || (FSM->
                                  currentState == state_DOWN))) {
90        FSM->trig_setTask = FALSE;
91        FSM->openDoorOnArrival = TRUE;
92        int directon = priv_chooseDirecton(FSM);
93        if (directon == direction_UP)
94          priv_setNewState(FSM, state_UP);
95        else if (directon == direction_DOWN)
96          priv_setNewState(FSM, state_DOWN);
97        else
98          priv_setNewState(FSM, state_OPEN_DOOR);
99      }
100   }
101
102   if (FSM->trig_resumeTaskAfterStop) {
103     FSM->trig_resumeTaskAfterStop = FALSE;
104     priv_setNewState(FSM, FSM->previousState);        // Resume the previous
          state
105   }
106
107   elev_FSM_UpdateLights(FSM);
108 }
```

1 Semaphores and shared variable synchronization

1-1) (Used in: v2015)

The following code is taken from a module called “logic” in an old lift control project.

```
#include <stdio.h>

#include "elev.h"
#include "control.h"
#include "logic.h"

// The order tables; one cell is unused in each table; ref. the floor panel
static int calls_up[N_FLOORS];
static int calls_down[N_FLOORS];

// helper function that is used by place_order(...) and place_call(...)
// to store orders in the order tables.
static void add_job(int path, int floor);

// Sverre: More functions omitted...

void place_order(int destination)
{
  // cancelling emergency stop
  if (emergency && !cancel_emergency) {
    restart(destination);
    return;
  }
  // The two border cases
  if (destination == 0) {
    calls_up[destination] = TRUE;
  }
  else if (destination == N_FLOORS - 1) {
    calls_down[destination] = TRUE;
  }
  // The three normal cases
  else if (destination > current_floor) {
    calls_up[destination] = TRUE;
  }
  else if (destination < current_floor) {
    calls_down[destination] = TRUE;
  }
  else if (destination == current_floor) {
    add_job(direction, current_floor);
  }
}

void restart(int destination)
{
  cancel_emergency = TRUE;
  // If STOP has been pressed between two floors
  if (!on_floor()) {
```

```
48      if (destination == current_floor) {
49        // The lift is over current_floor, needs to go down
50        if (ref_floor > current_floor) {
51          direction = DOWN;
52        }
53        // The lift is under current_floor, needs to go up
54        else {
55          direction = UP;
56        }
57      }
58      else if (destination < current_floor) {
59        direction = DOWN;
60      }
61      // destination > current_floor.
62      else {
63        direction = UP;
64      }
65      start_motor();
66    }
67    place_order(destination);
68  }
```

Criticize the code; point out any strengths and weaknesses.

1-2) (Used in: v2015)

In the corresponding header file we find:

```
1  // Takes a command from the comparment panel. If STOP has been pressed
2  // the the lift is to start again, this by calling restart(...). After
3  // this, orders can be entered normally again. If the lift is ordered
4  // to a floor over its position or to the lowest floor the order goes
5  // into the up-table. Oppositely for a floor under its position or the
6  // highest floor. If the lift is ordered to the floor it visited last,
7  // or the floor it is in, the table depends on the direction of the
8  // lift.
9  void place_order(int destination);
10
11 // Is called only in the emergency state by place(order...) and will
12 // then cancel the emergency state. If the function is called to start
13 // the lift after STOP has been pressed in a floor, only returned the
14 // control to place_order(...) and the lift will start. If, however,
15 // the function is called after a press of STOP between floors, the
16 // variable ref_floor (set by update_floor()) is used to decide the
17 // lifts position relative to the last floor. The motor can then be
18 // started in the correct direction by returning the control to
19 // place_order.
20 void restart(int destination);
```

Criticize the code — that is, the comments; point out any strengths and weaknesses.

1-3) (Used in: v2015)

The complete header file looks like this (after Sverre removed all the comments; evaluate the code as if the comments never were there):

```
1  #ifndef __INCLUDE_LOGIC_H__
2  #define __INCLUDE_LOGIC_H__
3
4  void change_direction(void);
5  void clear_orders(void);
6  void place_call(int direction, int floor);
7  void place_order(int destination);
8  void delete_order(void);
9  void restart(int destination);
10 int called_here(void);
11 int visible_orders(void);
12 void handle_order(void);
13 void debug_logic(void);
14
15 #endif
```

Criticize the code - that is, the module interface this time; point out any strengths and weaknesses.

1-4) (Used in: v2015)

What purpose does process pairs fill?

1-5) (Used in: v2015)

Acceptance tests are an "enabeling technology" for process pairs. Explain specificly how acceptance tests contributes to the functionality of a process pair.

1-6) (Used in: v2015)

Define the terms deadlock and race condition.

1-7) (Used in: v2015)

Can deadlocks and race conditions happen in a message passing system?

The Java documentation states this about *which* thread is awakened when notify is called:

> If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

The use of "arbitrary" here is quite typical, and descriptive of the documentation of many synchronization primitives.

1-8) (Used in: kont2015 , v2015)

Sverre states that there is only one reasonable way to implement this — first in, first out (FIFO); the thread that has been waiting the longest will be the one that is awakened — and that all implementations most probably will choose this one as the default behavior.

Why is FIFO a reasonable choice here?

1-9) (Used in: kont2015 , v2015)

If the FIFO strategy for waking processes is the reasonable one; why does the documentation so often insist on "arbitrary"?

This code is from Burns and Welling (if somewhat simplified) and has a bug.

```
1  // The priority parameter is either 0 or 1
2  // The mutex M is initialized to 1, the semaphores in the
3  // PS array is initialized to 0
4  void allocate(int priority){    void deallocate(){
5    Wait(M);                        Wait(M);
6    if(busy){                       busy=false;
7      Signal(M);                    waiting=GetValue(PS[1]);
8      Wait(PS[priority]);           if(waiting>0) Signal(PS[1]);
9    }                               else{
10   busy=true;                        waiting=GetValue(PS[0]);
11   Signal(M);                        if(waiting>0) Signal(PS[0]);
12  }                                  else{
13                                       Signal(M);
14                                     }
15                                   }
16                                 }
```

We have a resource used by a lot of threads, but rather than allocating the resource to the threads in the order they requested it, threads that gave the high priority flag as a parameter should get it first.

1-10) (Used in: v2015)

What is the bug: What happens? What is the consequence of the bug?

1-11) (Used in: v2015)

This error is not easily fixed using only semaphores. Why? What is the difficulty?

1-12) (Used in: v2015)

Give a short description of how the synchronization primitives in Java (synchronized methods, wait, notify and notifyAll) works.

1-13) (Used in: v2015) **(2x)** Sketch (pseudo-code) how you could solve the problem

from task 1f with Javas synchronization primitives.

1-14) (Used in: v2015)

Give a short description of how the synchronization primitives in ADA (Protected Objects, functions, procedures and entries with guards) works.

1-15) (Used in: v2015)

We think about implementing the functionality of 1f in Ada:

Even though the Java and Ada synchronization mechanisms are of approximately the same expressive power, this problem is not so straightforward to solve in Ada as it were in Java (or it would be with posix, monitors, conditional critical regions etc.).

Why?

... so we make two allocate functions instead of one; one for requests on low priorities (allocate_lowPri()) and one for requests on high priorities (allocate_highPri()).

1-16) (Used in: v2015) **(2x)** Sketch (pseudo-code) how you could solve this problem with Adas synchronization primitives.

1-17) (Used in: v2015)

Message based interaction between threads leads to a very different design than shared variable synchronization. How? Describe shortly the difference in designs.

1-18) (Used in: v2015) **(2x)** The underlying situation in the code from task 1f — controlling access to a shared resource dependent on priority — would not occur in a system based on messagepassing. Describe a corresponding situation in the messagepassing domain (some creativity might be neded? But the two priority levels should be present.) and write pseudo-code for a solution.

1-19) (Used in: v2015)

Compare the complexity of your implementation in the previous task with the code given in task 1f. Assume that a reasonable measure of complexity is the number of states the program can be in, if you made a formal model of it, e.g. with FSP. (...but you do not need to do any modeling)

Explain why the two cases compare as they do?

1-20) (Used in: v2015)

The priority ceiling protocol, in addition to solving the unbounded priority inversion problem, also have the property that it avoids deadlocks in the system. Explain how.

We have made a module that maintains a table of Person-structs and that contains these functions among others:

```
1 void printName(int i){
2   printf("%s",persons[i].firstName);
3   printf("␣%s\n",persons[i].lastName);
4 }
5 void printAddress(int i){
6   printf("%s",persons[i].street);
7   printf("␣%d\n",persons[i].streetNumber);
8 }
```

The module worked well until we made the program multithreaded...

1-21) (Used in: kont2014)

Some times the program prints strange errors in the address list (Sverre Hendseth and Ola Normanns entries are mixed up):

```
...
SverreOla Normann
Osloveien 23
 Hendseth
Trondheimsveien 34
...
```

What is the problem?

1-22) (Used in: kont2014)

...and even worse: Occasionally this can happen even inside of words:

```
...
SveOla Normann
Osrre Hendseth
Trondheimsveien 34
loveien 23
...
```

What is the problem here?

1-23) (Used in: kont2014)

What do we call such faults that occur “occasionally”?

1-24) (Used in: kont2014)

We solve the problem by protecting the functions in the module with semaphores. The module now looks like this:

```
1  // Assume a semaphore personSem initiated to 1 at the start of the program
2  void printName(int i){
3    sem_wait(personSem);
4    printf("%s",persons[i].firstName);
5    printf("␣%s\n",persons[i].lastName);
6    sem_signal(personSem);
7  }
8  void printAddress(int i){
9    sem_wait(personSem);
10   printf("%s",persons[i].street);
11   printf("␣%d\n",persons[i].streetNumber);
12   sem_signal(personSem);
13 }
```

How do using semaphores like this solve the problem?

1-25) (Used in: kont2014)

As the program grows it turns out that the combination of printName() and printAddress() occurs many places in the program (when printing name tags, letterheads, business cards, envelopes,...), and we decide to make the function *printNameAndAddress* as a part of the module:

```
1  void printNameAndAddress(int i){
2    sem_wait(personSem);
3    printName(i);
4    printAddress(i);
5    sem_signal(personSem);
6  }
```

Look at the module interface; What do you think about this decision in a code quality perspective?

1-26) (Used in: kont2014)

However, this does not work at all: The program never prints any combination of name and address; It just hangs.

What happens?

1-27) (Used in: kont2014)

What do we call this kind of fault?

1-28) (Used in: kont2014)

In this situation it might be an alternative to drop the semaphore protection of *printNameAndAddress*? Why would/wouldn't this be ok?

1-29) (Used in: kont2014)

This problem is not easily solved with semaphores. I would say that we have an example of the limited "expressive power" of semaphores here. Explain the term "expressive power".

1-30) (Used in: kont2014)

In Java this problem would be easy to solve: Describe shortly how?

1-31) (Used in: kont2014)

In Ada (that is, not using semaphores) this problem would be easy to solve: Describe shortly how?

1-32) (Used in: kont2014)

With POSIX mutexes the problem does not solve itself; we have to set the mutex in "recursive" mode (PTHREAD_MUTEX_RECURSIVE). How do you think a mutex in this mode works?

1-33) (Used in: kont2014)

If you really have to protect one of your modules using only semaphores; How would you approach it?

1-34) (Used in: kont2014)

More of these "resource allocation" problems does not occur in pure message passing systems ("message-based synchronization" as Burns&Wellings calls it). Why?

1-35) (Used in: kont2014)

(2x) A slightly larger part of our module is shown here:

```
1  #ifndef PERSON_H
2  #define PERSON_H
3
4  typedef struct {
5    char * firstName;
6    char * lastName;
7    char * street;
8    int streetNumber;
9  } TPerson;
10
11 void reallocateArray(int newSize);
12
13 TPerson ** getArray();
```

```
1  protected body Bounded_Buffer is
2    entry Get (Item : out Data_Item) when Num /= 0 is
3    begin
4      Item := Buf(First);
5      First := First + 1; Num := Num - 1;
6    end Get;
7
8    entry Put (Item : in Data_Item) when Num /= Buffer_Size is
9    begin
10     Last := Last + 1; Num := Num + 1;
11     Buf(Last) := Item
12   end Put;
13 end Bounded_Buffer;
```

Figur 4: Bounded Buffer in Ada

```
14
15 void printName(int personNumber);
16 void printAddress(int personNumber);
17 void printNameAndAddress(int personNumber);
18 ...
19 #endif
```

Criticize, from a code quality perspective, the inclusion of the type TPerson, and the two functions *reallocateArray*() and *getArray*()

1-36) (Used in: v2014)

Bloom used "expressive power" and "ease of use" to compare mechanisms for synchronization. Explain the two terms.

1-37) (Used in: v2014)

Give a short description of how the synchronization primitives in ADA (Protected Objects, functions, procedures and entries with guards) and Java (synchronized methods, wait, notify and notifyAll) works.

1-38) (Used in: v2014)

Compare the given (Fig. 4 and 5) implementations of a "Bounded Buffer" in Ada and Java wrt. "expressive power and ease of use". Which implementation do you think express the Bounded Buffer logic more clearly.

1-39) (Used in: v2014)

In one of the synchronization problems that Bloom put up, Adas mechanisms fails. Explain.

```
1   public synchronized void put(int item)
2   {
3       while (numberInBuffer == size) wait();
4       last = (last + 1) % size ;
5       numberInBuffer++;
6       buffer[last] = item;
7       notifyAll();
8   }
9
10  public synchronized int get()
11  {
12      while (numberInBuffer == 0) wait();
13      first = (first + 1) % size ;
14      numberInBuffer--;
15      notifyAll();
16      return buffer[first];
17  }
```

Figur 5: Bounded Buffer in Java

1-40) (Used in: v2014)

Read the attached page from "The little book of semaphores" about "Cigarette smokers problem".

Write pseudo-code with Adas protected objects to solve this problem. You are to sketch the protected object that has the responsibility to wake the correct "smoker"-thread.

The agent is given and unchangeable; Assume that it calls a random pair of the three functions, procedures or entries (your choice) — Tobacco(), Paper() or Matches() — in our object to signal that the two given resources are avaiable each time the table is empty.

You need to write only the protected object, but it should be clear how the "smoker"-threads are awoken.

1-41) (Used in: v2014)

Solve the same problem (pseudo-code), but this time with only semaphores as the synchronization mechanism.

The agent is also now given and unchangeable; Assume it calls signal on an arbitrary pair of the three semaphores sem_paper, sem_match and sem_tobacco each time the table is empty. (You *may* use if-statements or arrays of semaphores :-))

Hint: It does not have to be the smoker-threads themselves that wait on these semaphores. You may introduce one or more threads as mediators.

```
1  SEMAPHORE NInBuffer(0),NFree(N),Mutex(1);
2
3  put(e){                          get(e){
4    wait(NFree);                     wait(NInBuffer);
5    wait(Mutex);                     wait(Mutex);
6      // enter into buffer             // get e from buffer
7    signal(Mutex);                   signal(Mutex);
8    signal(NInBuffer);               signal(NFree);
9  }                                }
```

Figur 6: Bounded Buffer with semaphores

1-42) (Used in: v2014)

As you read, Patil suggested the Cigarette smokers problem as an argument to indicate that the expressive power of semaphores are limited. Do you think he succeeded with his argument? (Explain shortly)

1-43) (Used in: v2014)

A bounded buffer is a buffer where a process/thread that attempts to read from an empty buffer or write into a full one gets blocket. Conceptually this can be modelled in FSP as:

```
1  BUF(N=5) = BUF[0],
2  BUF[n:0..N] = (when n < N put -> BUF[n+1]
3                | when n > 0 get -> BUF[n-1]).
```

Assume N=3: Draw the transition diagram for this system.

1-44) (Used in: v2014)

Look at Fig. 6 which is the bounded buffer implemented with semaphores. Make a model in FSP of this implementation.

Hint: Look at assignment 2a for some relevant syntax.

1-45) (Used in: v2014)

Assume that the buffer is of size 1 (N=1). Draw the transition diagram.

1-46) (Used in: v2014)

If you have got task 2a and 2c correct you will see that the model of the semaphore-based implementation of the Bounded Buffer is more complicated than the conceptual model — at least if we imagine buffers of the same size. Why?

1-47) (Used in: v2014)

Pseudo-code: Sketch how a bounded buffer could have been implemented in a setting of message-passing processes. (Hint: Synchronous messages simplifies a bit.)

1-48) (Used in: v2014)

Comment on the expected (you do not need to write any code here) complexity of the transition diagram for the message based implementation of the bounded buffer.

1-49) (Used in: v2015 , v2014)

Message passing systems are not traditionally seen as very suited for implementing systems with real-time demands. Why?

1-50) (Used in: v2014)

In the listing under is a module interface (a C header file) from an old project deliverable. The modules name is "cost". Criticize (concisely, with bullet points) the design.

```
1  #ifndef lift_cost_h
2  #define lift_cost_h
3
4  int calculateCost(int currentFloor, int direction, int orderedFloor, int
       orderedDirection);
5
6  int downCost[MAX_ELEVATORS][N_FLOORS];
7  int upCost[MAX_ELEVATORS][N_FLOORS];
8
9  void fillCostArrays();
10 void clearCosts(void);
11
12 int lowestCostFloor(int elevator);
13 int lowestCostDirection(int elevator);
14
15 int findBestElevator(int floor, int direction);
16
17 void designateElevators();
18 void clearDesignatedElevator();
19
20 int designatedElevator[N_FLOORS][2];
21
22 #endif
```

1-51) (Used in: v2014)

This is another module interface. This time with the name of "jobqueue". Criticize it; feel free to compare it to the previous one.

```
1  #ifndef _JOB_QUEUE_H
2  #define _JOB_QUEUE_H
3
4  #include <time.h>
5
```

```
6  //Set flag at given floor in the given direction.
7  void set_flag(int floor, int direction);
8
9  //Deletes flag at given floor in the given direction.
10 void delete_flag(int floor, int direction);
11
12 //Returns one if flag at given direction at given gloor is set.
13 int get_flag(int floor, int direction);
14
15 //Deletes all flags in table.
16 void delete_all_flags();
17
18 //Returns one if there are no flags set in table.
19 int flag_table_empty();
20
21 //Returns one if there are any flags above the given floor.
22 int any_flags_above(int floor);
23
24 //Returns one if there are any flags below the given floor.
25 int any_flags_below(int floor);
26
27 //Prints the table of flags to the terminal.
28 void print_flag_table();
29
30 #endif
```

1-52) (Used in: v2014)

This listing (under) is the function "delete_flag" from the module in the previous task. Criticize (shortly and in bullet points).

```
1  //Delete flag in table.
2  void delete_flag(int floor, int direction) {
3
4          if(floor == 0) {
5                  flag_table[floor][DIR_UP] = 0;
6                  flag_table[floor][DIR_DOWN] = 0;
7                  elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
8          }
9          else if(floor == 3) {
10                 flag_table[floor][DIR_UP] = 0;
11                 flag_table[floor][DIR_DOWN] = 0;
12                 elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
13         }
14         else{
15                 if(!any_flags_above(floor) && get_current_direction() ==
                       DIR_UP) {
16                     flag_table[floor][DIR_UP] = 0;
17                     flag_table[floor][DIR_DOWN] = 0;
18
19                     elev_set_button_lamp(BUTTON_CALL_DOWN, floor, 0);
20                 }
21                 else if(!any_flags_below(floor) && get_current_direction() ==
```

```
                    DIR_DOWN) {
22                      flag_table[floor][DIR_UP] = 0;
23                      flag_table[floor][DIR_DOWN] = 0;
24
25                      elev_set_button_lamp(BUTTON_CALL_UP, floor, 0);
26              }
27
28              flag_table[floor][direction] = 0;
29      }
30
31      if(direction == DIR_UP && floor < 3) { elev_set_button_lamp(
            BUTTON_CALL_UP, floor, 0); }
32      if(direction == DIR_DOWN && floor > 0) { elev_set_button_lamp(
            BUTTON_CALL_DOWN, floor, 0);}
33
34      elev_set_button_lamp(BUTTON_COMMAND, floor, 0);
35  }
```

1-53) (Used in:

1-54) (Used in: h2009)

Wrong use of semaphores may lead to deadlocks: What is a deadlock? Give an example on how this can happen.

1-55) (Used in: v2013 , h2009)

Generally; List methods you can use to avoid that deadlocks becomes a problem in a software system.

1-56) (Used in: h2009)

Acceptance tests is seen as an important tool for handling errors. How do acceptance tests differ from the more traditional tests on error conditions?

1-57) (Used in: v2014 , v2013 , v2011kont , h2010)

Acceptance tests is seen as an important tool for handling errors. What do we gain from using acceptance tests in addition to the more traditional tests on error conditions?

1-58) (Used in: v2014)

Give examples of what one can test for when making acceptance tests.

1-59) (Used in: v2014)

When an acceptance test fails, we are left with the knowledge that *something* is wrong, but we do not necessarily know exactly what. How can we recover i such a situation?

1-60) (Used in: v2013)

What do we achieve by merging error modes?

1-61) (Used in: h2009)

What is a systems error modes, and what do we achieve by merging error modes?

1-62) (Used in: v2011kont , h2009)

Optimistic Concurrency Control is a technique that can be used to avoid the overhead of locking. How does this work?

1-63) (Used in: h2010)

"Resource control" is more than mutual exclusion... What more can we expect from a system for resource control?

1-64) (Used in:

1-65) (Used in: v2011kont , h2009)

Shared variable synchronization may be criticized for poor scalability. Give, shorly, arguments for this.

1-66) (Used in: h2010)

Designing parallel systems with message-sending is an alternative to basing the design on synchronization. Give shortly and by bullet points benefits and drawbacks of these two applied to real-time systems design.

Our goal is to make a barrier/rendezvous between three threads by using semaphores. That is, all threads should block at the synchronization point until all three are ready to proceed.

1-67) (Used in: h2009)

```
1 if(getWaiting(A) == 2){ // GetWaiting returns the number of
2                         // processes waiting for the semaphore
3   // The two others are here already - release and continue.
4   signal(A);
5   signal(A);
6 }else{
7   // The number of waiting threads is less than two - must wait.
8   wait(A)
9 }
```

This code has a race condition. What is a race condition? What is the problem with the code?

1-68) (Used in: h2009)

```
count++;
if(count==3){
  // All here, signal the others, reset and continue
  signal(A);
  signal(A);
  count = 0;
}else{
  // Not everyone have arrived, must wait
  wait(A);
}
```

This code has at least one problem: Which?

1-69) (Used in: h2009)

(implementation/pseudo-code,3x) Make a barrier that works.

1-70) (Used in: v2011kont , h2009)

(implementation,pseudo code,2x) We have a resource in our system that is used by many threads, creating the need for synchronizing access. When there are more waiting threads the *last* request should be given priority. Write pseudo code for the allocate() and free() functions that achieves this. Use the synchronization mechanisms in C/POSIX, Ada or Java as you prefer.

1-71) (Used in: v2013)

We imagine a storage with limited size, where elements can be entered and taken out. Two operations should be implemented (pseudo-code java w. $wait()$, $notify()$ and $notifyall()$), $void\ put(Element\ e)$ and $Element\ get()$. Fill out the functions given under.

The handling of the elements, or the strategy for which element should be retrieved by $get()$ is not important: You only need to show the synchronization: *put* should block when the storage is full, and *get* should block if it is empty.

```
public class FiniteCapacityStorage {

    private int current_storage_size;
    final int   storage_capacity = 50;
    // Make assumptions here as necessary.

    public  FiniteCapacityStorage()
    {
        this.current_storage_size = 0;
    }

    public synchronized void put(Element new_item)
    {
        // You do the rest!
    }

```

```
17      public synchronized Element get()
18      {
19          Element dummy = 0;
20
21          // You do the rest!
22
23          return dummy;
24      }
25  }
```

To increase the abstraction level and give flexibility compared to semaphores, POSIX, Java and Ada have landed on different variants of monitors: POSIX combines mutexes and condition variables, Java has synchronized methods and *wait*/*notify*/*notifyAll*, and Ada has guarded entries in protected objects.

1-72) (Used in: v2013 , v2011kont , h2009)

(2x) Describe shortly how these three works.

1-73) (Used in: v2013 , v2011kont , h2009)

Compare the Java and Ada mechanisms here, give the main strengths and weaknesses.

1-74) (Used in: v2013 , h2010)

Mention some hardware (/assembly) mechanisms that are used for achieving basic synchronization.

1-75) (Used in: v2015 , v2013 , v2011kont , h2010)

The *Suspend*() and *Resume*(*Thread*) calls have been described as unusable when it comes to programming error free synchronization between threads. Explain why.

1-76) (Used in: v2013 , h2010)

Javas *wait* and *notify* (and POSIX condition variables) works in many ways similar to *suspend* and *resume*. How can these be seen as better or more high-level synchronization mechanisms?

1-77) (Used in: v2013 , v2011kont , h2010)

"Resumption model" is used to describe possible implementations of both signals, asynchronous notification in general and exceptions. In most cases it is seen as less useful than "termination model". What is the difference? Why is resumption model less useful?

1-78) (Used in: kont2015 , v2013 , v2011kont , h2010 , h2009)

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?

1-79) (Used in: v2013kont)

In a real-time program we often have more concurrently running, cooperating threads. Why is this a reasonable way of organizing the functionality of a real-time system?

1-80) (Used in: v2013kont)

That the threads are cooperating means that they will have some need for interaction. This interaction usually takes the form of either "synchronization" or "communication". Explain these two shortly and/or give examples.

1-81) (Used in: v2013kont)

Compare shortly and in bullet points how well suited these two are for a real-time system.

1-82) (Used in: v2013 , h2010)

Error detection — to decide whether something is wrong – can be done in more ways than checking for error returns. List principles that can be used to detect errors.

1-83) (Used in: v2013)

In an atomic action we wish to have control over the participants and the duration of the action. The necessary synchronization that is necessary for this can be given a reusable form and put into an ActionController object.

In the code below is the entry protocol, that keeps track of the participants, and ensures that no role has more than one participant at the same time, already implemented.

You shall implement (java pseudo-code w. $wait()$, $notify()$ and $notifyAll()$) the missing part of the exit-protocol, that ensures that both participants exits the action at the same time.

```
1  public class ActionController {
2
3      Boolean firstHere, secondHere;
4      int allDone; // Better name: doneCounter ?
5      final int numberOfParticipants = 2;
6
7
8      public ActionController()
9      {
10
11         firstHere = secondHere = false;
12         allDone = 0;
13
```

```
14      }
15
16      public synchronized void entryProtocolOne()
17      {
18          while(firstHere) wait();
19          firstHere = true;
20      }
21      public synchronized void entryProtocolTwo()
22      {
23          while(secondHere) wait();
24          secondHere = true;
25      }
26
27      synchronized void exitProtocolBoth()
28      {
29          // You do the rest!
30      }
31
32  }
```

1-84) (Used in: v2013)

What do we achieve (in the domain of error handling) by using static redundancy?

1-85) (Used in: v2013)

What do we achieve (in the domain of error handling) by using dynamic redundancy?

1-86) (Used in: h2009)

Backward error recovery is some times seen as not suited in a real time system. What is backward error recovery, and why is it not suited?

1-87) (Used in: h2009)

What is forward error recovery?

Both forward and backward error recovery can be generalized from single thread to multi thread systems.

1-88) (Used in: h2009)

Backward error recovery may, when generalizing to multi-thread systems give the domino effect. Explain. How can we avoid the domino effect?

1-89) (Used in: kont2014)

Backward error recovery may, when generalizing to multi-thread systems give the domino effect. Explain. How can we avoid the domino effect?

1-90) (Used in: h2009)

When doing forward error recovery in a multi thread setting, the need arises for the different threads to get to know about errors that happen in other threads. List mechanisms that can be used to convey such information.

1-91) (Used in: h2009)

Writing to log is an alternative to creating recovery points. How does this work in the context of (single-thread) backward error recovery?

1-92) (Used in: kont2014)

Writing to log is an alternative to creating recovery points. How does this work in the context of (single-thread) backward error recovery?

1-93) (Used in: h2009)

Handling the log gets more difficult if we have more parallel tasks, where some succeeds and some fails, and all generate log. How can we extend the log (compared to the single-thread version) to handle this (still in a backward error recovery perspective)?

1-94) (Used in: v2014 , v2013 , h2009)

What is an Atomic Action? Which problem(s) is Atomic Actions meant to solve?

1-95) (Used in: v2014)

Describe one way of implementing an Atomic Action.

1-96) (Used in: v2014)

Operations for locking resources are always assumed to be atomic. Why is this so important?

1-97) (Used in: h2010)

What is the benefits by structuring the functionality of your program into Atomic Actions? Any drawbacks?

1-98) (Used in: v2015 , h2009)

Give a short description of how process pairs work.

1-99) (Used in: v2011kont , h2010)

Process pairs is a technique for achieving available (in addition to reliable) processes. Which of the following points are the central building blocks/principles for implementing process pairs?

1 An "I Am Alive" protocol that lets the backup process detect whether the primary process is alive and working.

2 A (two-phase) commit protocol between primary and backup processes ensuring that they have a consistent view of the current state.

3 Sending state updates from the primary to the backup so that the backup will start working in the current state if/when it becomes primary.

4 Highly reliable and available messages must ensure that all IamAlive and State Update messages reach their destination and in the correct order.

5 Some mechanism for letting clients relate to the process that currently is primary (like broadcasting "IamPrimary" also to clients).

6 Process independent message queues that ensures that messages are not lost even though a process restarts.

"Standard" error handling is to test for error situations and then make code that handles the detected errors. The fault tolerance part of this course is however motivated by the fact that this is not good enough:

1. A real-time or embedded system often have higher demands to reliability; We must handle also unexpected errors. (The errors that you did not think of when making the program and the bugs that you are unaware of.)

2. We also often have more cooperating threads in a real-time software system. Sometimes these threads must cooperate also on error handling.

1-100) (Used in: v2015 , v2011)

How can we detect these unexpected errors ? (explain shortly.)

1-101) (Used in: v2015 , v2011)

Given that we have detected such an unexpected error... How can we know what we must do to handle the error ?

1-102) (Used in: v2015 , v2011kont , v2011)

The failure modes is the ways a system can fail. To "merge failure modes" is a technique: What do we gain by doing this ?

1-103) (Used in: kont2014 , v2013kont)

Explain the terms “backward error recovery” and “recovery points”.

1-104) (Used in: v2013kont)

Explain the domino effect.

1-105) (Used in: v2013kont)

How can the domino effect be avoided?

1-106) (Used in: v2013kont)

If the program state is big or we have more concurrent ongoing operations that can fail individually, then *log* might be a good alternative. How does this work in context of backwards error recovery?

1-107) (Used in: v2013kont)

Another purpose of the log could be to enable program restart, so that a restarted program can get initialized to a current, consistent state e.g. after a crash. How?

1-108) (Used in: v2013kont)

We can imagine that after some time such a program can generate huge amounts of log. “Checkpoints” is a design pattern that lets us delete old log. How does this work?

1-109) (Used in: kont2014)

We can imagine that after some time such a program can generate huge amounts of log. “Checkpoints” is a design pattern that lets us delete old log. How does this work? (Be aware that the word “Checkpoint” is used for different things in different parts of the curriculum.)

An Atomic Action has start, side and end boundaries.

1-110) (Used in: v2015 , v2011kont)

What is the purpose of these boundaries ?

1-111) (Used in: v2015 , v2011kont)

How can each of them (start, side and end) be realized?

Asynchronous notification - to interrupt threads in what they are doing to communicate something, is discussed in relation to atomic actions.

1-112) (Used in: v2011kont)

Why ? Why is asynchronous notification discussed in relation to atomic actions?

1-113) (Used in: v2014)

What is "Asynchronous Transfer of Control"?

1-114) (Used in: v2014)

Why is Asynchronous Transfer of Control seen as relevant for use in Atomic Actions?

1-115) (Used in: v2011kont)

How can we get «termination mode» asynchronous transfer of control in POSIX/C? What about Java(/RT Java)? And ADA?

Gray had the thought that we could build fault tolerant computer systems if we only had reliable data storage, reliable communication and reliable calculations...

1-116) (Used in: v2011)

Refer shortly how we can achieve reliable storage (that is, built on unreliable storage)

1-117) (Used in: v2011)

Refer shortly how we correspondingly can achieve reliable communication.

1-118) (Used in: v2011)

Reliable calculations are the most difficult in this triplet since the failure modes is difficult to describe in the general case. The curriculum sets up "Checkpoint/Restart" as the first alternative: Describe how this works. What is the error modes, and how are they detected?

Gray had the thought that we could build fault tolerant computer systems if we only had reliable data storage, reliable communication and reliable calculations...

1-119) (Used in: kont2014)

Reliable calculations are the most difficult in this triplet since the failure modes is difficult to describe in the general case. The curriculum sets up "Checkpoint/Restart" as the first alternative: Describe how this works.

1-120) (Used in: kont2014)

A simple and straightforward alternative to checkpoints like this is to write to the "log". How does this work, and what do we achieve compared to the checkpoints?

1-121) (Used in: kont2014)

What is, in this context the error mode(s) of a "calculation", and how are they detected?

Gray had the thought that we could build fault tolerant computer systems if we only had reliable data storage, reliable communication and reliable calculations...

1-122) (Used in:

Reliable calculations are the most difficult in this triplet since the failure modes is difficult to describe in the general case. The curriculum sets up "Checkpoint/Restart" as the first alternative: Describe how this works. What is the error modes, and how are they detected?

1-123) (Used in: kont2014 , v2011)

(2x) Another way to achieve reliable calculations is process pairs. Give a short explanation on how this works. What do we gain here compared to checkpoint/restart?

1-124) (Used in: kont2014)

"Transactions" are almost the same as "Atomic Actions" in the Burns&Wellings book, and are relevant when making fault tolerant computer systems. How do transactions contribute to fault tolerance.

1-125) (Used in: kont2014)

What is the difference between Tansactions and Atomic Actions? How does this difference make Atomic Actions more attractive to use in a real-time system?

1-126) (Used in: kont2014)

Grays third alternative way to make reliable calculations assumes that all data (the whole program state) is safely stored in a database, that all calculations are formulated as transactions, and that we have a system for running these transactions which also, automaticly handles any errors.

In addition to the answer in the previous question; Why do we not make all (embedded, real-time) systems like this?

1-127) (Used in: v2013kont)

What do we achieve by using process pairs?

1-128) (Used in: v2013kont)

Explain shortly how process pairs work.

A number of techniques/design patterns has been set up to handle a number of challenges in error handling and consistency in multi thread systems:

- Writing log records to the log
- Writing Checkpoints to the log
- The two-phase commit protocol
- Two-phase execution (growing and shrinking phases)
- Introducing a transaction manager
- (Read/Write-) Locking of accessed resources.
- Introducing a LogManager
- Introducing a LockManager

For each of the tasks under, you shall pick the most central technique to achieve what the task asks for.

1-129) (Used in:

For the application you are writing it is critical that you keep track of what you have done, so that you may start in the current state even after a restart.

1-130) (Used in:

After having programmed 3 different "atomic actions" you observe that you have code for coordinating start and end boundaries 3 different places in the code. You want to collect reusable code into a module.

1-131) (Used in:

You have decided that this growing and shrinking phase concept is a nice way to keep track of the possible consequences of errors. You need however to coordinate, between the participating threads, the point in time where the growing phase ends and the shrinking phase begins.

1-132) (Used in: v2011kont , h2009)

Use of exceptions in a language that supports it (like Java) simplifies some things. List the benefits of using exceptions / What gets better or simpler by using exceptions?

1-133) (Used in: v2011kont , h2009)

Are there any disadvantages by using exceptions in a program, real-time system or programming language. (Answer briefly).

1-134) (Used in: v2011kont , h2010)

In a system where exception handling is used, the following code may cause a deadlock:

```
{
  Wait(semaphoreA);
  f();
  Signal(semaphoreA);
}
```

Explain what the problem is and how it is solved.

1-135) (Used in: v2013kont)

A number of modern programming languages comes with built-in mechanisms for exception handling. What are we hoping to achieve with these mechanisms?

1-136) (Used in: v2013kont)

Explain shortly how exception handling works. Use C++, Java or ADA as example language if you want.

1-137) (Used in: v2013kont)

Which downsides exist to using exception handling in a program/real-time system.

We have used the following demands on a principle for dividing a system into modules:

1. We must be able to use a module without knowing its internals.
2. We must be able to maintain a module without knowing its usage patterns.
3. Composition: Super-modules can be made out of sub-modules.

In this perspective: Criticize (shortly) the following principles for dividing a system into modules. Will they be suited for contructing a large software system ?

1-138) (Used in: v2011kont)

Dividing functionality into *Threads*, like in Java or POSIX.

1-139) (Used in: v2011kont)

Monitors

1-140) (Used in: v2011kont)

Communicating processes (as in OCCAM or Google Go - communication by messages, no shared resources).

1-141) (Used in: v2011kont)

Atomic Actions

We have the following three FSP processes:

$$P = (a \to b \to c \to P).$$
$$Q = (a \to b \to d \to Q).$$
$$R = (d \to b \to c \to R).$$

A possible interpretation of an event that two processes participates in is that there is (synchronous) communication between them. If there are more than two participating processes, it may be a barrier/rendezvous.

1-142) (Used in: kont2015 , h2009)

Draw the transition diagram for P and $P||Q$.

1-143) (Used in: kont2015 , h2009)

$P||Q||R$ deadlocks. Why?/What happens? How can you recognize a deadlock in a transition diagram?

1-144) (Used in: kont2015 , v2011kont , h2010 , h2009)

What is a live-lock, and how can this be recognized in a transition diagram?

1-145) (Used in: kont2015 , h2009)

Model the following semaphore program in FSP.

```
T1(){
  while(1){
    wait(A);
    wait(M);
    ...
    signal(M);
    signal(A);
  }
}
```

```
T2(){
  while(1){
    wait(B);
    wait(M);
    ...
    signal(M);
    signal(B);
  }
}
```

A reasonable way of getting synchronous communication using semaphores can be like in the following program stubs, where thread 1 sends a sequence of integers to thread 2.

```
1
2  int ch; // The channel
3  Semaphore ch_sendReady(0),ch_receiveReady(0);
4
5  thread1(){
6    int i=0;
7    while(1){
8
9      // send i
10     ch = i;
11     ch_sendReady.signal();
12     ch_receiveReady.wait();
13     i++;
14   }
15 }
16
17 thread2(){
18   int i;
19   while(1){
20
21     // receive i
22     ch_sendReady.wait();
23     i = ch;
24     ch_receiveReady.signal();
25
26   }
27 }
```

1-146) (Used in: v2013)

Make a model of the system in FSP (Finite State Process-notation)- look at the attached FSP reference if necessary.

1-147) (Used in: v2013)

Draw the transition diagram for the total system. How many states are there ?

If we imagine a language with synchronous communication built into the language, the corresponding code can look like this:

```
1  T1(int channel ch){
2    while(1){
3      int i;
4      ch <- i;
5      i++;
6    }
7  }
8
9  T2(int channel ch){
10   while(1){
11     int i;
12     i <- ch;
13     //...
14   }
15 }
```

1-148) (Used in: v2013)

Model this system in FSP. (Hint: Expect a quite simple model)

1-149) (Used in: v2013)

Draw the transition diagram for the total system. How many states are there? (Hint: Fewer than last time)

1-150) (Used in: v2013)

We have in both examples two threads that passes information between them. Reflect shortly on why one model is simpler than the other.

If we imagine a larger system of threads that communicate by this two-semaphores-per-channel pattern, the number of semaphores in the system may become very large.

1-151) (Used in: v2013)

Give an example showing how deadlocks can occur in such a message passing system.

1-152) (Used in: v2013)

And specifically for our message passing system: Of the points you mentioned in the previous question; Which do you see as most relevant here? (...or if you have new points to mention to ensure the absence of deadlocks...)

1-153) (Used in:

A classic example of a system with a deadlock is “dining philosophers”, where N philosophers sits eating at a round table. Each need two forks to eat, and must share with both his neighbours the N forks distributed between the philosophers.

1-154) (Used in: kont2014 , v2013kont , v2011)

Show how the philosophers and the forks can be modeled in FSP.

1-155) (Used in: kont2014 , v2013kont , v2011)

Draw a transition diagram for a philosopher and for a fork.

1-156) (Used in: kont2014 , v2013kont , v2011)

Draw a transition diagram for a compund system with two philosophers and two forks. Point out the deadlock.

1-157) (Used in: h2010)

The book sets up four necessary conditions for deadlocks to occur. Which? And how can deadlocks be prevented by seeing to it that each of these are not met?

1-158) (Used in: h2010)

In a traditional real-time system we often know exactly which threads that uses which resources. This opens up for some techniques that lets us avoid the problem of deadlocks. Which?

1-159) (Used in: h2010)

However, predicting which threads use which resources is not always possible; Imagine a system of mobile sensor vehicles sharing updates to via a base station, or a process control plant where the operators are allowed to program alarm scripts. In both these cases we may get unpredictable resource allocations.

How can we avoid or solve any problems with deadlocks in such systems?

1-160) (Used in: v2013kont , v2011)

What is a deadlock ? Give an example of a deadlock with semaphores.

1-161) (Used in: v2011)

Deadlocks can of course also happen in systems where interaction between threads is message based. Construct an example from this domain.

1-162) (Used in: scheduling , v2013kont)

Good scheduling is important in real-time programming to ensure that all deadlines in the system are satisfied. How do we achieve this by scheduling strategies?

1-163) (Used in: kont2015 , kont2014 , v2013kont)

(2x) We can prove that the deadlines in the system will be satisfied by response time analysis and utilization-based schedulability tests. Explain shortly how these two works.

1-164) (Used in: kont2015 , v2013kont)

There are a number of assumptions/conditions that must be true for these tests to be usable. ("The simple task model") Which?

1-165) (Used in: v2014)

There are a number of assumptions/conditions that must be true for these tests to be usable. ("The simple task model") Which? Comment (shortly) on how realistic they are.

1-166) (Used in: kont2015)

The schedulability proofs probably have not been performed for most real-time systems out there in the world. Why do you think the industry is reluctant to perform these proofs?

1-167) (Used in: v2013kont)

What is "Priority Inversion" ?

1-168) (Used in: v2013kont)

How does priority inversion influence the schedulability analyses?

1-169) (Used in: kont2014 , v2013)

Why is "scheduling" so important in real-time programming? — Which features are we hoping to gain in our system by choosing good scheduling strategies?

1-170) (Used in: kont2014 , v2013)

We imagine a classical real-time system with threads that synchronize using semaphores: How, typically, is it decided which thread gets to run (to achieve the goals in the previous question)?

1-171) (Used in: v2013)

In the same system: How, typically, is it decided which of more threads waiting on a semaphore, gets to run when the semaphore is signaled?

1-172) (Used in: kont2014)

In the same system: How, typically, is it decided which of more threads waiting on a semaphore, gets to run when the semaphore is signaled? Why?

1-173) (Used in: v2013)

What is the "unbounded priority inversion" problem?

1-174) (Used in: v2013)

How can the "unbounded priority inversion" problem be solved by scheduling strategies?

Given these three tasks:

Task	Period	Execution time
A	80	20
B	60	20
C	40	10

1-175) (Used in: v2014 , v2013)

Show in a graphical way how task A, B and C in the table over will run under rate monotonic priority assignment. Mark clearly if and where any deadlines are missed. (Assume that all tasks are ready for the first time at time zero)

1-176) (Used in:

1-177) (Used in: v2014 , v2013)

Make an utilization test for this system.

$$\left(Hint : U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N(2^{1/N} - 1) \right)$$

1-178) (Used in: v2014 , v2013)

Make a response time analysis of this system.

$$\left(Hint : w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \right)$$

You are to make a real-time system, which means that you have real-time demands to the system, and a duty to be able to argue that all demands will be met. (That is, beyond the common "you have tested the system and it seems to work" strategy)

1-179) (Used in: v2015 , v2011)

How would you structure the system to be able to argue that the realtime demands will be met ? (In other words: Can you say something about your software design.)

1-180) (Used in: v2015 , v2011)

And the argumentation itself: Sketch along which lines you would argue that the real-time demands are met.

1-181) (Used in: v2015 , v2011)

Interaction between different parallel tasks may lead to "inversion" problems (like priority inversion). Explain what the problem is.

1-182) (Used in: v2015 , v2011)

Explain shortly how a Priority Ceiling protocol works. (That is, choose one of them.)

Two wellknown scheduling strategies is FPS (Fixed Priority Scheduling) and EDF (Earliest Deadline First).

1-183) (Used in: v2011)

Explain the terms FPS and EDF.

1-184) (Used in: v2011)

Why is FPS more commonly in use ?

We have started a number of threads T_1 to T_N. T_1 initializes some data structures that the other threads use so that they should be blocked until T_1 has finished initializing. One way of doing this with semaphores is the following: (A is a binary semaphore initialized to 0.)

```
T1(){                          T2(){
  // Do the initialization         wait(A);
  signal(A);                       signal(A);
  // Continue working              // Data structures are initialized,
}                                  // continue working
```

}

T_3...T_N is like T_2.

1-185) (Used in: v2011kont , h2010)

Assume N=3. Model the system in FSP. (If you want to avoid the problem of terminating processes you can model "Continue Working" as $DW = (doWork \rightarrow DW)$.) See the attached FSP reference if necessary.

1-186) (Used in: v2011kont , h2010)

With N still equal to 3: Draw the transition diagram for the composed system.

1-187) (Used in: v2011kont , h2010)

(Implementation/Pseudo-code) How would you solve this problem of holding back threads T_2...T_N in Java (using synchronized, wait, notify and notifyAll)?

1-188) (Used in: v2011kont , h2010)

(Implementation/Pseudo-code) How would you solve this problem of holding back threads T_2...T_N in Ada (using protected objects, functions, procedures, entries and guards)?

Look at the given semaphore variant of the program again. This program leaves the semaphore A in an open state. Now we want to reuse the semaphore and the signaling effect that the initialization is finished, and must therefore reset the semaphore. (Imagine an initialization process that happens more times during the execution of the program.)

1-189) (Used in: v2011kont , h2010)

Now the program looks like this. Assume that g_N and $g\_nOfThreadsInitialized$ is global variable initialized to the number of threads and 0 respectively.

```
T1(){
  while(1){
    // Do the initialization
    signal(A);
    g_nOfThreadsInitialized++;
    if(g_nOfThreadsInitialized == g_N){
      g_nOfThreadsInitialized = 0;
      wait(A);
    }
    // Continue working

  }
}
```

```
T2(){
  while(1){
    wait(A);
    signal(A);
    g_nOfThreadsInitialized++;
    if(g_nOfThreadsInitialized == g_N){
      g_nOfThreadsInitialized = 0;
      wait(A);
    }
    // Data structures are initialized,
    // continue working
  }
}
```

This program has a race condition (at least). What is a race condition and what is the problem(s)?

You are to write a resource manager that administers a memory area of limited size. The call to $void * Allocate(int\ nOfBytes)$ takes the amount of memory to reserve as a parameter and returns a pointer to a memory area. $Free(void*)$ takes such a pointer to a memory area and releases the memory again.

1-190) (Used in: v2013kont , h2010)

(Implementation/pseudo-code) You are to use shared variable synchronization, but can use any mechanisms for this.

Focus is on the synchronization; Make suitable assumptions when it comes to the memory book keeping and any data structures you need.

1-191) (Used in: v2013kont , h2010)

What is starvation?

1-192) (Used in: v2013kont , h2010)

Comment on the danger for starvation regarding the memory allocation task over. (If you did not have an implementation that was free of possible starvation problems:) Suggest a strategy for memory allocation to waiting threads that does not have this problem.

1-193) (Used in: v2013kont)

We imagine a classical real-time system with prioritized threads that synchronize using semaphores: How, typically, is it decided which of more threads waiting on a semaphore, gets to run when the semaphore is signaled?

1-194) (Used in: v2013kont , h2010)

Adas mechanisms (protected objects with functions, procedures and entries with guards) are not that suited to solve this memory allocation problem... Why?

1-195) (Used in:

You have already made a module for keeping track of names of people. It has, among other parts of the interface, these functions: *getFirstName*, *getLastName*, *setFirstName* and *getName*. The last one calls the two first ones before it returns the complete name. The module worked perfectly until a multithreaded version of the program was made...

1-196) (Used in: v2011kont , v2011)

What kind of problems with such a module can surface when it is used in a multithreaded program?

1-197) (Used in: v2011)

Anyway, you decide to protect the module with semaphores by allocating (*wait*) a given binary semaphore at the start of each function, and releasing it (*signal*)just before returning. This does not work: What happens ?

1-198) (Used in: v2011)

How would you fix this problem? (You still have only semaphores available for synchronization.)

1-199) (Used in: v2011)

Describe shortly how Javas monitor-like mechanisms for synchronization works ("synchronized objects").

1-200) (Used in: v2011)

(Java pseudo-code) You are making a corresponding module in Java: Sketch how the four functions could be protected from concurrent use by more threads here.

1-201) (Used in: v2011)

Describe shortly how Adas monitor-like mechanisms for synchronization works ("protected objects").

1-202) (Used in: v2011)

(Ada Pseudo-code) You are making a corresponding module in Ada: Sketch how the four functions could be protected from concurrent use by more threads here. Is this

solution in any way better than the Java solution ?

1-203) (Used in: v2011)

Limiting access to the module to only one thread at the time may be unnecessary: Assume that a lot of the functions in the module *can* be called by different threads at the same time without problems. What is the arguments for making a potentially more complex access control for the module that would allow this ?

1-204) (Used in: v2011kont , v2011)

Nested calls to monitors (that one function in a monitor makes a call to another monitor) is a problem that leads to monitorbased systems not scaling well. What is the problem?

1-205) (Used in: v2011)

How does Ada avoid any of these problems with nested monitor calls.

1-206) (Used in: v2011)

Inheritance Anomaly is a potential problem when we imagine inheriting from a class (in an objectoriented setting like Java or c++) where any methods synchronize. Explain where the problem lies.

1-207) (Used in: v2011)

I have suggested that systems based on shared variable-based synchronization scales badly. Can you argue this more generally?

1-208) (Used in: kont2015) **(7x)** Many techniques that can contribute to fault tolerance have been presented in this course. For each of those under: Describe shortly how they work/what they entail, and how they contribute to fault tolerance.

- Acceptance Tests
- Merging of Error Modes
- Static Redundancy
- Dynamic Redundancy
- Recovery Points
- Backward Error Recovery
- Forward Error Recovery
- N-version Programming

- Recovery Blocks
- Atomic Actions
- Transactions
- Process Pairs
- Error Injection
- Fail Fast
- Checkpoint/Restart

1-209) (Used in: kont2015)

Real-time software is well and good, but no real-time demands is met when the program is not running.

One of the difficult features to give a system is to allow upgrading the system to a new version without loosing service.

Outline shortly how you would approach making a system with this feature.

Readers/Writers locks is a variant of mutual exclusion where we recognize that we do not need mutual exclusion between more readers. The writers still needs to run under mutual exclusion — with each other and with the readers.

1-210) (Used in: kont2015)

How is it that we do not need mutual exclusion between multiple readers, when it is necessary between the writers?

1-211) (Used in: kont2015)

Why/When would we use readers/writers locks in place of ordinary mutual exclusion?

1-212) (Used in: kont2015)

With Adas Protected Objects, readers/writers locks are very easy to make; Explain how.

1-213) (Used in: kont2015) **(2x)** A small challenge: Sketch (pseudo-code) how one can make readers/writers locks with Javas synchronization mechanisms (synchronized methods, wait, notify, notifyAll). Hint: Make the functions startRead, stopRead, startWrite og stopWrite.

1-214) (Used in: kont2015)

If we envision a lot of readers and writers, then such use of *notifyAll* may be unfortunate. Why?

1-215) (Used in: kont2015) **(2x)** A larger challenge: Sketch (pseudo-code) how one can make readers/writers locks with semaphores.

1-216) (Used in: kont2015)

What is starvation?

1-217) (Used in: kont2015)

Readers/writers locks is an interesting case for discussing starvation. Why?

1-218) (Used in: kont2015)

If you mean that one of the solutions in 1-4 or 1-7) is safe against starvation; Explain why. If not; suggest how (Pseudo-code not necessary) one could make a version that avoids starvation. (for either the Java or the semaphores version)

1-219) (Used in: kont2015)

Assume that the Ada scheduler works in a way that makes the trivial readers/writers lock implementation from 1-3 safe against starvation. Comment on this implementation from a code quality perspective.

A simple comparison between the Google Go programming language and the almost-dead programming language Occam was done (a few years ago). Both programs had two threads on the same priority, one writing 'A's and the other 'B's.

The Google Go program produces an almost-regular sequence of A's and B's: "ABABABABAABABABABABBAABABAB..."

The Occam program produced "ABAAAAAAAAAAAAAAAAAA..." (A's continuing)

1-220) (Used in: kont2015)

Comment on how these two different run-time system strategies can lead to differences in code quality.

1-221) (Used in: kont2015)

A group some years ago handed in this lift project. What can you say about their division of the project into modules, module naming, or on the project as a whole? (Comment only on the few most important things, good or bad)

```
1     61 channels.h
2     39 elevator.c
```

```
  151 elev_driver.c
   42 elev_driver.h
  344 elev_FSM.c
  139 elev_FSM.h
   86 io.c
   62 io.h
   20 Makefile
  645 network.c
  142 network.h
  171 sch_dist_calc.c
   24 sch_dist_calc.h
  296 sch_elev_manager.c
   22 sch_elev_manager.h
  135 sch_main.c
   14 sch_main.h
  304 sch_orders.c
   95 sch_orders.h
  123 sch_packets.c
   35 sch_packets.h
  145 sch_peers.c
   39 sch_peers.h
   76 sch_print_info.c
   18 sch_print_info.h
  102 sch_types.h
  188 storeOrders.c
   30 storeOrders.h
   44 timer.c
   32 timer.h
```

The numbers are the line count of each file. The total project consists of 1183 lines with ';' making it of approximately average size.

1-222) (Used in: kont2015)

Here is the elev_FSM.h file from the same project. Comment on the few most important points, good or bad.

```
#ifndef __INCLUDE_ELEV_FSM_H__
#define __INCLUDE_ELEV_FSM_H__

/*
  About this module:
  A Finite State Machine controlling the elevator.

  The FSM is assigned a destination floor,
  and does best effort in reaching this floor.
  It flags itself as ''defect'' it something goes wrong.

  Known bugs:
  - Due to different specifications in the elevator hardwares,
  the elevator tram may sometimes drive beyond the end floors,
  getting the elevator stuck below the ground floor, or above the top floor.
*/
```

```
17
18 #include "elev_driver.h"
19 #include "timer.h"
20
21 #define FALSE                   0
22 #define TRUE                    1
23
24 // For the elevator FSM
25 // Directions
26 #define direction_HERE          0
27 #define direction_UP            1
28 #define direction_DOWN          2
29
30 // States
31 #define state_UP                0
32 #define state_DOWN              1
33 #define state_OPEN_DOOR         2
34 #define state_IDLE              3
35 #define state_STOP              4
36 #define state_ACCOMPLISHED      5
37 #define state_OBSTRUCTION       6
38
39 // Times (delays)
40 #define Time_doorOpen           2500        // Milliseconds
41 #define Time_brake                 5        // Milliseconds
42 #define Time_transportLimit     3000        // Milliseconds
43
44 // Velocities
45 #define speed_UP                200
46 #define speed_DOWN              -200
47 #define speed_STOP              0
48
49 /*
50   Holds all information about the elevator FSM,
51   in order to make it behave in an more or less intelligent way.
52   All of these fields are available to the module using this module,
53   but none should be altered, only read.
54 */
55 typedef struct // The elevator FSM type
56 {
57   // States
58   int previousState;
59   int currentState;
60
61   // Info about the position
62   int positionUnknown;                        // TRUE if the position is
         unknown. Only used during startup. Makes the FSM use sensors to
         determine the position
63   int atAFloor;                               // TRUE if the elevator is
         situated at a floor (and not in a gap)
64   int currentFloor;
65   int currentFloorGap;
66
```

```
67    // Booleans describing trigger
68    // Internal (6 triggers)
69    int trig_enteringFloor;
70    int trig_leavingFloor;
71    int trig_stopButtonPressed;
72    int trig_doorTimerDownAndNotObstruction;
73    int trig_transportTimerDown;
74    int trig_obstruction;
75
76    // External (4 triggers)
77    int trig_acknowledgeTaskSuccess;        // TRUE if the function acknowledge
          () has been run from previus iteration
78    int trig_setTask;                        // TRUE if the task was updated
          since last time. Then usually destination_Floor is also updated
79    int trig_cancelTask;
80    int trig_resumeTaskAfterStop;
81
82
83    // Help variables for deciding specific branches etc
84    int destinationFloor;
85    int openDoorOnArrival;
86    int defect;
87    int defectEngine;
88
89    // Help variables
90    timerStruct doorTimer;                    // The timer for the door
91    timerStruct transportTimer;               // The timer for the
          transportation
92    int sensor_stopButtonPrevValue;           // The stop button sensor's
          previous value
93
94    // Sensor information. In order to avoid reading the sensors more than once
           per iteration.
95    int sensor_floorSensor;
96    int sensor_stopButton;
97    int sensor_obstruction;
98  } FSM_data;
99
100
101 int init_elev_FSM(FSM_data *FSM);
102 void kill_elev_FSM(FSM_data *FSM);
103
104 /*
105   Update the FSM. Absolutly no inputs or outputs from the scheduler through
          this one,
106   just let the FSM process the info recieved from last time
107 */
108 void update_elev_FSM(FSM_data *FSM);
109
110
111 // FETCH INFO
112 /*
113   Returns TRUE if the elevator can be assigned a new task. This is only the
```

```
        case when the state is IDLE, UP or DOWN,
114   in addition to some minor conditions (like: the elevator is functional).
        This must be true when calling trig_setTask()
115 */
116 int FSM_get_canAssignNewTask(FSM_data *FSM);
117
118
119 // TRIGGERS FROM THE SCHEDULER
120 void FSM_trig_acknowledgeTaskSuccess(FSM_data *FSM);
121
122 /*
123   Sets a new destination. Must only be called when get_canAssignNewTask() is
        true.
124 */
125 void FSM_trig_setTask(FSM_data *FSM, int destination);
126
127 /*
128   Drives to the nearest floor, and enters state IDLE, without opening the
        door
129 */
130 void FSM_trig_cancelTask(FSM_data *FSM);
131
132 /*
133   Resumes the previous task after the stop button was pressed.
134   This trigger is not in use.
135 */
136 void FSM_trig_resumeTaskAfterStop(FSM_data *FSM);
137
138
139 #endif // #ifndef __INCLUDE_ELEV_FSM_H__
```

1-223) (Used in: kont2015)

Here is a function from the elev_FSM.c file from the same project. Comment on the few most important points, good or bad.

```
1  /*
2          Needs to be called once per iteration.
3          This is the brain of the elevator's logic.
4  */
5  void update_elev_FSM(FSM_data *FSM) {
6    priv_fetchSensorSignals(FSM);
7    priv_internalTriggers(FSM);
8
9
10   if(FSM->trig_obstruction){
11     if (FSM->currentState != state_OBSTRUCTION)
12       priv_setNewState(FSM, state_OBSTRUCTION);
13   }
14
15   if(FSM->currentState == state_OBSTRUCTION && FSM->trig_obstruction==0)
16     priv_setNewState(FSM, state_IDLE);
17
```

```
18    if (FSM->trig_stopButtonPressed) {
19      if (FSM->currentState != state_STOP)
20        priv_setNewState(FSM, state_STOP);
21    }
22
23    if (FSM->trig_transportTimerDown) {
24      if ((FSM->currentState == state_UP) || (FSM->currentState == state_DOWN))
            {
25        FSM->defect                       = TRUE;
26        FSM->defectEngine                 = TRUE;
27        priv_elevatorStopEngine(FSM);
28      }
29    }
30
31    if (FSM->trig_doorTimerDownAndNotObstruction)
32      if ((FSM->currentState == state_OPEN_DOOR) && (!(FSM->defect)))
33        priv_setNewState(FSM, state_ACCOMPLISHED);
34
35    if (FSM->trig_enteringFloor) {
36      FSM->atAFloor        = TRUE;
37      FSM->currentFloor    = FSM->sensor_floorSensor;
38
39      if (FSM->defect)
40        priv_setNewState(FSM, state_OPEN_DOOR);
41      else
42        {
43          if (FSM->positionUnknown) {
44            FSM->positionUnknown = FALSE;
45            FSM->openDoorOnArrival = TRUE;
46            priv_setNewState(FSM, state_IDLE);
47          } else {
48            if (!FSM->openDoorOnArrival) {
49              FSM->openDoorOnArrival = TRUE;
50              priv_setNewState(FSM, state_IDLE);
51            } else {
52              if (priv_traveledTooDestination(FSM))
53                priv_setNewState(FSM, state_OPEN_DOOR);
54              if (priv_traveledTooFar(FSM)) {
55                FSM->defect = TRUE;
56                priv_elevatorStopEngine(FSM);
57              }
58              if (priv_traveledTooShort(FSM))
59                startTimer(&(FSM->transportTimer)); // Reset timer
60            }
61          }
62        }
63    }
64
65    if (FSM->trig_leavingFloor) {
66      FSM->atAFloor = FALSE;
67      if (FSM->currentState == state_UP)
68        FSM->currentFloorGap   = FSM->currentFloor;
69      if (FSM->currentState == state_DOWN)
```

```
70        FSM->currentFloorGap   = FSM->currentFloor - 1;
71
72      startTimer(&(FSM->transportTimer)); // Reset timer
73    }
74
75    if (FSM->trig_acknowledgeTaskSuccess)
76      if (FSM->currentState == state_ACCOMPLISHED)
77        {
78          FSM->trig_acknowledgeTaskSuccess = FALSE;
79          priv_setNewState(FSM, state_IDLE);
80        }
81
82    if (FSM->trig_cancelTask) {
83      FSM->trig_cancelTask = FALSE;
84      FSM->openDoorOnArrival = FALSE;
85    }
86
87    if (FSM->trig_setTask) {
88      if ((!(FSM->defect)) && ((FSM->currentState == state_IDLE) || (FSM->
          currentState == state_STOP) ||
89                                (FSM->currentState == state_UP) || (FSM->
                                     currentState == state_DOWN))) {
90        FSM->trig_setTask = FALSE;
91        FSM->openDoorOnArrival = TRUE;
92        int directon = priv_chooseDirecton(FSM);
93        if (directon == direction_UP)
94          priv_setNewState(FSM, state_UP);
95        else if (directon == direction_DOWN)
96          priv_setNewState(FSM, state_DOWN);
97        else
98          priv_setNewState(FSM, state_OPEN_DOOR);
99      }
100   }
101
102   if (FSM->trig_resumeTaskAfterStop) {
103     FSM->trig_resumeTaskAfterStop = FALSE;
104     priv_setNewState(FSM, FSM->previousState);        // Resume the previous
          state
105   }
106
107   elev_FSM_UpdateLights(FSM);
108 }
```