Tentamen
Datorteknik och realtidssystem, TSEA81

Datum	2021-08-20
Lokal	TER1, FE249
Tid	14-18
Kurskod	TSEA81
Provkod	TEN1
Kursnamn	Datorteknik och realtidssystem
Institution	ISY
Antal uppgifter	5
Antal sidor (inklusive denna sida)	16
Kursansvarig	Kent Palmkvist
Lärare som besöker skrivsalen *Telefon under skrivtiden*	Anders Nilsson 013-28 2635
Besöker skrivsalen	Ca 15 och 17
Kursadministratör	Maria Hamnér, 013-28 5715
Tillåtna hjälpmedel	Inga
Betygsgränser	**Poäng** **Betyg** 41-50 5 31-40 4 21-30 3 0-20 U

Viktig information

- Alla svar ska ha en motivering om inget annat anges. Om du svarar med programkod räknas kommentarer i programkoden som motivering. Svar som ej är motiverade kan leda till poängavdrag.
- Om inget annat anges ska du anta att schemaläggningsmetoden som används är *priority based preemptive scheduling*.
- Om inget annat anges antas semaforer vara starka.
- Om du är osäker på det exakta namnet för en viss funktion, skriv en kommentar om vad funktionen gör så kommer vi troligtvis att förstå vad du menar. (Detsamma gäller syntaxen för programspråket C.)
- Tänk igenom din lösning NOGGRANT och använd dig av de lösningsprinciper som kursen förevisar. Okonventionella och tvetydiga lösningar ger poängavdrag.
- Svara ALDRIG med pseudokod, om det inte specifikt efterfrågas. Pseudokod blir lätt tvetydig och därmed inte bedömningsbar.
- Lämna INTE in denna tentamen tillsammans med lösningarna. En inlämnad tentamen med eventuella anteckningar kommer inte att beaktas som en lösning.
- Skriv läsbart! Oläsbar text kan inte bedömas och ger därmed inga poäng.

Lycka till!

Uppgift 1: Schemaläggning(10p)

Ett realtidssystem med ett antal processer ska schemaläggas på en dator som enbart har en processor/kärna, dvs endast en process åt gången kan exekvera. Följande krav gäller:

- *P1* ska arbeta/köra under *x* tidsenheter i tidsintervallet [*i**4, (*i*+1)*4]
- *P2* ska arbeta/köra under 3 tidsenheter i tidsintervallet [*i**7, (*i*+1)*7]

där *i* är ett heltal och $i \geq 0$, och *x* är ett heltal och $x \geq 0$.

Tidräkningen startar vid t=0 för alla processer. Eventuellt missat arbete under något tidsintervall ackumuleras *inte* till kommande tidsintervall.

(a) (5p) Antag att schemaläggningsmetoden Earliest Deadline First (EDF) används. Man vill att *P1* ska köra så mycket som möjligt samtidigt som hela systemet beräknas behöva 5% slack-marginal bland annat för processbyte. Dvs, under 5% av tiden ska inga processer köra. Alltså, beräkna största möjliga värde på *x* så att specifikationerna uppfylls. Visa sedan vad som händer när programmet körs genom att rita ett tidsdiagram.
Hur stor blir den faktiska utnyttjandegraden?

(b) (5p) Antag att schemaläggningsmetoden Rate Monotonic Scheduling (RMS) används. Antag samma värde på *x* som i a-uppgiften.
Visa vad som händer när programmet körs genom att rita ett tidsdiagram.
Kommer processerna att uppfylla kraven?
Hur stor blir den faktiska utnyttjandegraden?
Kan sambandet för RMS som säger att om $U_e \leq n(2^{1/n} - 1)$ så uppfylls kraven, användas i detta fall för att avgöra om kraven uppfylls eller inte? Motivera svaret.

Uppgift 2: Teori(8p)

(a) (2p) Beskriv kortfattat de steg som utförs när en process anropar *Await* på en händelsevariabel.

(b) (3p) Beskriv minst tre huvudsakliga skillnader mellan en *Mutex* och en *Semafor*.

(c) (3p) Förklara kortfattat, vad är prioritetsinversion? Namnge en vanlig metod för att motverka prioritetsinversion.

Uppgift 3: Semaforer(10p)

Nedanstående programlistning är ett utdrag av funktionerna `sem_wait` samt `sem_post` från ett hypotetiskt operativsystem:

```
void sem_wait(struct sem *s)
{
    ***;

    if( *** > 0){
        ***;
    }else{
        ***;
        ***;
        ***;
    }

    ***;
}

void sem_post(struct sem *s)
{
    ***;

    if (!empty( *** )){
        ***;
        ***;
        ***;
    }else{
        ***;
    }

    ***;
}
```

Uppgiften fortsätter på nästa sida

Platser som markerats med `***` har dock raderats. Varje förekomst av `***` ska ersättas med någon av följande kodsnuttar:

- `ev->mutex->wait_list`
- `ev->wait_list`
- `ev->mutex->counter++`
- `ev->mutex->counter--`
- `int task_id = remove_highest_prio(s->wait_list)`
- `ready_list_insert(task_id)`
- `ready_list_insert`
- `ready_list_remove`
- `ready_list_remove(current_task_id)`
- `schedule()`
- `s->counter++`
- `s->counter--`
- `s->mutex`
- `s->counter`
- `s->event`
- `s->wait_list`
- `s->ready_list`
- `DISABLE_INTERRUPTS`
- `ENABLE_INTERRUPTS`
- `wait_list_insert(s->wait_list,current_task_id)`
- `wait_list_remove_highest_prio`
- `wait_list_remove_one`
- `wait_list_is_empty`
- `wait_list_insert`

(a) (8p) Din uppgift är att skriva ner de kompletta implementationerna av `sem_wait` respektive `sem_post`. För full poäng måste källkoden vara kommenterad med kommentarer som förklarar vad som händer och varför!

(b) (2p) Är detta strukturen för en stark eller en svag semafor? Vad är signifikant för en sådan semafor?

Uppgift 4: Starka och svaga semaforer(10p)

Betrakta programkoden för de två processerna nedan. Antag att semaforen `S` är initierad till `1` och att både `P1` och `P2` initialt är körklara. Funktionen `do_work()` tar en viss obestämd tid att utföra, men använder inte funktionerna `si_sem_wait()`, `si_sem_signal()` eller `si_wait_n_ms()`.

```
/* task P1 */
void P1(void)
{
    si_sem_wait(&S);      // wait on semaphore
    printf("A\n");
    si_wait_n_ms(2000);   // sleep for 2000 ms
    printf("B\n");
    si_sem_signal(&S);    // signal on semaphore
    printf("C\n");
    do_work();            // do some work for some time
    printf("D\n");
    si_sem_wait(&S);      // wait on semaphore
    printf("E\n");
    si_wait_n_ms(2000);   // sleep for 2000 ms
    printf("F\n");
    si_sem_signal(&S);    // signal on semaphore
    printf("G\n");
    si_wait_n_ms(1000);   // sleep for 1000 ms
    printf("H\n");
    while(1);             // wait forever
}

/* task P2 */
void P2(void)
{
    si_sem_wait(&S);      // wait on semaphore
    printf("I\n");
    si_wait_n_ms(1000);   // sleep for 1000 ms
    printf("J\n");
    do_work();            // do some work for some time
    printf("K\n");
    si_sem_signal(&S);    // signal on semaphore
    printf("L\n");
    si_sem_wait(&S);      // wait on semaphore
    printf("M\n");
    si_sem_signal(&S);    // signal on semaphore
    printf("N\n");
    si_wait_n_ms(1000);   // sleep for 1000 ms
    printf("O\n");
    while(1);             // wait forever
}
```

Uppgiften fortsätter på nästa sida

(a) (2p) Antag att semaforen `S` är en svag semafor samt att `P1` har högre prioritet än `P2`. Vad kommer programmet att skriva ut efter att det startats?

(b) (8p) Antag att semaforen `S` är en stark semafor samt att `P2` har högre prioritet än `P1`. Beskriv steg för steg vad som händer från det att programmet startats. Var noga med att tala om vilka processer som är körande, vilka listor dom ligger i vid olika tillfällen, semaforens status, samt motivera varför olika händelser sker. Listornas exakta namn är inte viktigt, bara det framgår vad deras syfte är.

Uppgift 5: Udda/jämna tal(12p)

Man önskar konstruera ett program med tre processer P1, P2 och P3 som var och en har en egen specifik uppgift:

- P1 ska kontinuerligt generera slumptal, som är heltal mellan 0-99, och skriva dessa till en buffer som rymmer fyra tal
- P2 ska läsa talen från samma buffer i tur och ordning och avgöra om P1's slumptal är udda eller jämna
- P3 ska för *varje enskilt tal* som P2 läser, kontrollera om P2 kom till rätt slutsats

Utöver ovanstående gäller även följande förutsättningar/krav:

- Alla processerna P1, P2 och P3 antas kunna köra parallellt
- Processen P3 ska skriva ut "`CORRECT`" om P2 kom till rätt slutsats, i annat fall ska P3 skriva ut "`WRONG`", dock bara en utskrift för varje enskilt tal.
- Den färdiga lösningen ska förstås vara sådan att P2 *alltid* kommer till rätt slutsats, dvs så att P3 *alltid* skriver ut "`CORRECT`"

Det finns en färdig funktion som kan anropas för att generera lämpliga slumptal från 0 till 99:

```
/* Generate and return random number 0-99 */
int get_random_value(void)
{
    return rand_r(&rand_r_state) % 100;
}
```

Du kan räkna med att huvudprogrammet initierar processer/trådar på ett sätt som fungerar för din lösning, dock utan någon betydande prioritetsordning. Du måste dock själv initiera globala variabler och dylikt, men det räcker om dessa initieringar finns som kommentar, dvs det måste inte vara programkod, bara det är tydligt med vad som avses.

Din uppgift är alltså att skriva programkoden i C för de tre processerna P1, P2 och P3 samt komplettera med egna definitioner och initieringar som du finner nödvändigt. Du kan anta att nödvändiga header-filer redan finns inkluderade.

Observera, att lösningen *inte* får använda meddelandehantering.

Lösningsförslag fråga 1

Följande notation gäller:

```
# = process running
_ = process not running
. = no process running (unused time slot)
| = deadline (met)
/ = deadline (missed)
```

1a

En marginal innebär att det behövs 5% (minst) outnyttjad tid, dvs 1/20 av utnyttjandegraden. För EDF gäller att kraven uppfylls om utnyttjandegraden $U_e \leq 1$, dvs:
$x/4 + 3/7 + 1/20 \leq 1$
$x/4 + 67/140 \leq 140/140$
$x/4 \leq 73/140$
$x \leq 73/35$
$73/35$ är större än 2 men mindre än 3, alltså, största värde på x är 2.

Med EDF schemaläggs alltid den process som har kortast tid kvar till deadline. Då flera processer har lika lång tid kvar och en av dom redan kör låter man den processen fortsätta för att slippa ett processbyte.

```
P1# # _ _|_ # # _|# # _ _|# # _ _|# # _ .|# # _ _|_ # # .|
P2_ _ # # # _ _|# _ _ # # _ _|# # _ _ # . _|_ # # # _ _ .|
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
```

Alla processer klarar sina deadlines och den faktiska utnyttjandegraden blir 13/14 (ty x=2 ger Ue=26/28=13/14). Vid tidpunkten t=28 blir samtliga processer samtidigt redo att köra igen, dvs där börjar förloppet om.

(Kommentar: Lösningen behöver visa hela tidsförloppet (inklusive deadlines) fram till t=28, att körande process fortsätter (i förekommande fall), att värdet på x=2, samt att den faktiska utnyttjandegraden blir 13/14)

1b

MED RMS får processerna prioriteter utefter hur ofta de ska köras, dvs ju kortare tidsintervall ju högre prioritet. Prioriteten bli alltså P1 > P2. Därefter används schemaläggningsmetoden priority based preemptive scheduling.

```
P1# # _ _|# # _ _|# # _ _|# # _ _|# # _ .|# # _ _|# # _ .|
P2_ _ # # _ _ #|# _ _ # # _ _|# # _ _ # . _|_ # # _ _ # .|
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
```

P1 och *P2* klarar sina deadlines. Den faktiska utnyttjandegraden blir 13/14. Vid tidpunkten t=28 blir samtliga processer samtidigt redo att köra igen, dvs där börjar

förloppet om.

Sambandet $U_e \leq n(2^{1/n} - 1)$ kan i detta fall inte användas för att avgöra om processerna klarar sina krav. Även om sambandet inte uppfylls (ty U_e=13/14 $\geq n(2^{1/n} - 1)$ = $2(2^{1/2} - 1) \approx 0.8$) så kan processerna fortfarande klara sina krav, vilket dom gör i detta fall.

(Kommentar: Lösningen behöver visa hela tidsförloppet (inklusive deadlines) fram till t=28, tala om de inbördes prioriteterna för P1 och P2 vid RMS, korrekt dra slutsatsen om vilka processer som klarar sina deadlines (och visa var och vilka som inte gör det), visa outnyttjade tidsintervall, samt att den faktiska utnyttjandegraden blir 13/14, samt motivera varför sambandet för U_e inte kan användas.)

Lösningsförslag fråga 2:

2a

Stäng av avbrott. Om väntelistan för semaforen inte är tom, så flytta processen med högst prioritet från väntelistan för semaforen till ready-listan, i annat fall räkna upp semaforen med 1. Därefter flytta den process som anropar Await från ready-listan till väntelistan för händelsevariabeln. Anropa schemaläggaren. Slå på avbrott.

Kommentar: Observera, att uppgiften efterfrågar vilka steg som utförs när en process anropar Await, inte hur Await används i en process tillsammans med övrig programkod.

2b

En Mutex kan bara ha värdet 0 (låst) eller 1 (olåst). En Semafor kan ha värdet 0 eller större. En Mutex måste låsas och låsas upp av samma tråd. En Semafor kan låsas och låsas upp av olika trådar. En Semafor kan användas till assymetrisk eller symmetrisk synkronisering. Det går inte att göra med en Mutex.

Kommentar: Uppgiften efterfrågar huvudsakliga skillnader mellan en Mutex och en Semafor, inte bara egenskapen för den ena av dom underförstått egenskapen för den andra. Att bara säga att "En Mutex kan vara 0 eller 1" säger inget om vad en Semafor kan vara. Kan den vara allt utom 0 eller 1? Kan den vara negativ?

2c

Prioritestinversion är när en högprioriterad process blockeras tillgång till en gemensam resurs som innehas av en lågprioriterad process, samtidigt som en mellanprioriterad process (utan att använda den gemensamma resursen) kan avbryta den lågprioriterade processen och därmed få köra (och indirekt blockera den högprioriterade processen) trots lägre prioritet än den högprioriterade processen. Prioritestärvning kan motverka prioritetsinversion.

*Kommenter: Det behöver framgå att det finns ett gemensamt beroende mellan två processer, samt att en tredje oberoende process via sin mellanprioritet indirekt kan blockera den högprioriterade processen. Namnet på metoden för att motverka prioritetsinversion efterfrågas, inte *bara* hur metoden fungerar.*

Lösningsförslag fråga 3

3a

```
void sem_wait(struct sem *s)
{
  DISABLE_INTERRUPTS; // Skydda semaforens datastruktur genom att
                      // inte tillåta avbrott när sem_wait kör
  if(s->counter > 0){
    s->counter--; // Om semaforen är tillgänglig behöver vi enbart räkna
                  // ner den med ett
  }else{
    // Semaforen är otillgänglig. Se till så att
    // processen hamnar i semaforens väntelista och
    // inte längre är markerad som körklar genom att
    // ta bort processen från listan med
    // körklara processer.
    ready_list_remove(current_task_id);
    wait_list_insert(s->wait_list,current_task_id);

    schedule();       // Hitta den högst prioriterade körklara processen
                      // och växla till denna.
  }

  ENABLE_INTERRUPTS;  // Skydd av datastruktur upphör
}

void sem_post(struct sem *s)
{
  DISABLE_INTERRUPTS; // Skydda semaforens datastruktur genom att
                      // inte tillåta avbrott när sem_post kör
  if (!empty(s->wait_list)){
    // Det är minst en process som ligger och väntar i sem_wait()
    // på denna semafor. Ta bort den högst prioriterade processen som
    // väntar ifrån semaforens väntelista och gör denna körklar.
    // (Vi räknar inte upp semaforen här eftersom sem_wait isåfall
    // direkt skulle behöva räkna ner den.)
    int task_id = remove_highest_prio(s->wait_list);
    ready_list_insert(task_id);

    schedule(); // Hitta den högst prioriterade körklara processen
                // och växla till denna.
  }else{
    // Ingen väntar på semaforen just nu, gör semaforen tillgänglig
```

```
      // genom att räkna upp semaforens värde.
      s->counter++;
    }

    ENABLE_INTERRUPTS; // Skydd av datastruktur upphör
  }
```

Notering: För full poäng räcker det inte att skriva kommentarer som enbart förklarar vad varje rad kod gör utan att sätta in raderna i ett större sammanhang. Dvs, kod som kommenteras i stil med nedan gör ingen särskild glad (oavsett om det är på tentan eller i verkliga livet).

```
void sem_post(struct sem *s)
{
    DISABLE_INTERRUPTS; // Stäng av interrupts

    if(!empty(s->wait_list)){ // Kolla om väntelistan är tom
```

3b

Det är en stark semafor. Signifikant för en stark semafor är att den förebygger/förhindrar svält då en högre prioriterad process hindras att få tillgång till semaforen direkt igen om en lägre prioriterad process begärde tillgång till semaforen under tiden som den högre prioriterade processen hade tillgång till semaforen.

Lösningsförslag fråga 4:

4a

Programmet skriver ut: A B C D E F G I H

Kommentar: Det finns en avgörande punkt när P1 begär tillgång till semaforen en andra gång precis innan utskrift av "E". P2 har dessförinnan begärt tillgång till semaforen, och för en svag semafor medför det att P1 får semaforen och svälter ut P2. I ett senare skede, efter utskrift av "G", när P1 sover utan att ha semaforen får P2 chansen att skriva ut "I" och sedan sova. P1 vaknar, skriver ut "H", och går sedan in i oändligheten vilken inte avbryts av P2 med lägre prioritet. Rätt fram till och med "E" ger 1p, allt rätt (varken mer eller mindre) ger 2p.

4b

Här finns flera tillfällen som kan orsaka ett processbyte. När en process gör sleep (och en annan process är körklar), när en process återkommer från sleep och har högre prioritet, när en process anropar wait (och semaforens värde är 0) samt när en process kör signal (och en högre prioriterad process är körklar).

Tre listor blir aktuella, en time-list för då sleep anropas (T), en wait-list för semaforen (W) och en ready-lista för körklara processer (R). Semaforens värde finns i kolumnen (S).

	Orsak		**Verkan**				
	P1	P2	kör	S	T	W	R
1)	”S C H E D U L E”		P2	1			P1,P2
2)		wait(S)	P2	0			P1,P2
3)		sleep(1)	P1	0	P2		P1
4)	wait(S)		–	0	P2	P1	–
5)		signal(S)	P2	0			P1,P2
6)		wait(S)	P1	0		P2	P1
7)	sleep(2)		–	0	P1	P2	–
8)	signal(S)		P2	0			P1,P2
9)		signal(S)	P2	1			P1,P2
10)		sleep(1)	P1	1	P2		P1
11)	wait(S)		P1	0	P2		P1
12)	sleep(2)		–	0	P1,P2		–
13)		while(1)	P2	0	P1		P2
14)		while(1)	P2	0			P1,P2

1) Från det att båda processerna är körklara blir P2 (högst prioritet) körande.

2) P2 gör wait(S), S>0 så S räknar ned

3) printf(I), P2 gör sleep(1), varpå P1 (redo) blir körande

4) P1 gör wait(S), S=0 så P1 läggs i W, ingen process redo, ingen kör

5) P2 vaknar, blir redo och körande, printf(J), kör do_work(), printf(K), gör signal(S) varpå P1 flyttas till R, P2 högst prio fortsätter, S räknar inte upp ty stark semafor, printf(L)

6) P2 gör wait(S), S=0 så P2 läggs i W, varpå P1 (redo) blir körande

7) P1 printf(A), gör sleep(2), ingen redo, ingen kör

8) P1 vaknar blir redo och körande, printf(B), gör signal(S), varpå P2 blir redo och körande ty högre prio

9) P2 printf(M), gör signal(S), W tom så S räknar upp, printf(N)

10) P2 gör sleep(1), varpå P1 (redo) blir körande

11) P1 printf(C), do_work(), printf(D), gör wait(S), S>0 så S räknar ned

12) P1 printf(E), gör sleep(2), ingen redo, ingen kör

13) P2 vaknar blir redo och körande, printf(O), kör while(1)

14) P1 vaknar blir redo

Programmet skriver ut: I J K L A B M N C D E O

Kommentar: Beskrivningar här är av största vikt. Bara en tabell där det inte framgår vad som är orsak och verkan blir mycket svårbedömt. Det behöver framgå vilken process och vilken operation (orsak) som leder till körande process, semaforens värde och listornas innehåll (verkan). Mot slutet, steg 11 ovan, kan man fundera på om P2 hinner vakna och eventuellt avbryta P1 under do_work(). Uppgiften säger inget om hur lång tid do_work() tar, så när skulle avbrottet i så fall ske? Dvs, det är egentligen inte relevant att anta någon särskild exekveringstid för do_work() och utgå från det i lösningen. Det behöver i så fall motiveras mycket väl.

Lösningsförslag fråga 5:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

/* Common resources */
int Buf[4];
int In_Pos = 0;
int Out_Pos = 0;
int Size = 0;

/* Mutex and event variable to protect common resourses */
pthread_mutex_t M;
pthread_cond_t C;

/* Semaphores to synchronize P2 and P3 */
sem_t S1, S2;

/* Common variables for P2 and P3 */
int Number = 0;          // the number
int Even_Number = 1;     // 1=even, 0=odd


/* Generate and return random number 0-99 */
int get_random_value(void)
{
    return rand_r(&rand_r_state) % 100;
}


/* Process to generate random number */
void *P1(void *unused)
{
    while(1) {
        pthread_mutex_lock(&M);
        while (Size == 4) {
            pthread_cond_wait(&C, &M);
        }
        Buf[In_Pos] = get_random_value();
        In_Pos = (In_Pos + 1) % 4;
        Size++;
        pthread_cond_broadcast(&C);
        pthread_mutex_unlock(&M);
    }
}
```

```
    /* Process to determine odd or even number */
    void *P2(void *unused)
    {
        while(1) {
            pthread_mutex_lock(&M);
            while (Size == 0) {
                pthread_cond_wait(&C, &M);
            }
            Number = Buf[out_pos];
            Out_Pos = (Out_Pos + 1) % 4;
            Size--;
            Even_Number = 1 - (Number % 2);
            pthread_cond_broadcast(&C);
            pthread_mutex_unlock(&M);
            sem_post(&S1);
            sem_wait(&S2);
        }
    }


    /* Process to check if P2 is correct */
    void *P3(void *unused)
    {
        while(1) {
            sem_wait(&S1);
            if ((Number % 2 == 0) & (Even_Number == 0) ||
            (Number % 2 == 1) & (Even_Number == 1)) {
            printf("WRONG!\n");
            exit(1);
        }
        else {
            printf("CORRECT!\n");
        }
        sem_post(&S2);
    }
}


int main(int argc, char **argv)
{
    /* seed for random number */
    unsigned int rand_r_state = time(NULL);

    pthread_mutex_init(&M, NULL);
    sem_init(&S1, 0, 0);
    sem_init(&S2, 0, 0);

    pthread_t P1_handle;
    pthread_t P2_handle;
    pthread_t P3_handle;
```

```
    pthread_create(&P1_handle, NULL, P1, 0);
    pthread_create(&P2_handle, NULL, P2, 0);
    pthread_create(&P3_handle, NULL, P3, 0);

    pthread_join(P1_handle, NULL);
    pthread_join(P2_handle, NULL);
    pthread_join(P3_handle, NULL);

    while(1);
}
```

(Kommentar: Korrektioner som måste göras i inlämnad lösning för att den ska fungera ger olika mycket poängavdrag beroende på hur mycket som måste korrigeras för att lösningen ska fungera. Om lösningen inte har tillräcklig stomme som ger någon grundläggande funktion alls, ges inga poäng)