Tentamen
Datorteknik och realtidssystem, TSEA81

Datum	2022-03-15
Lokal	TER4
Tid	14-18
Kurskod	TSEA81
Provkod	TEN1
Kursnamn	Datorteknik och realtidssystem
Institution	ISY
Antal uppgifter	5
Antal sidor (inklusive denna sida)	18
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 4 tidsenheter i tidsintervallet [i*14, (i+1)*14]
- *P2* ska arbeta/köra under x tidsenheter i tidsintervallet [i*7, (i+1)*7]
- *P3* ska arbeta/köra under 1 tidsenhet i tidsintervallet [i*4, (i+1)*4]

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.

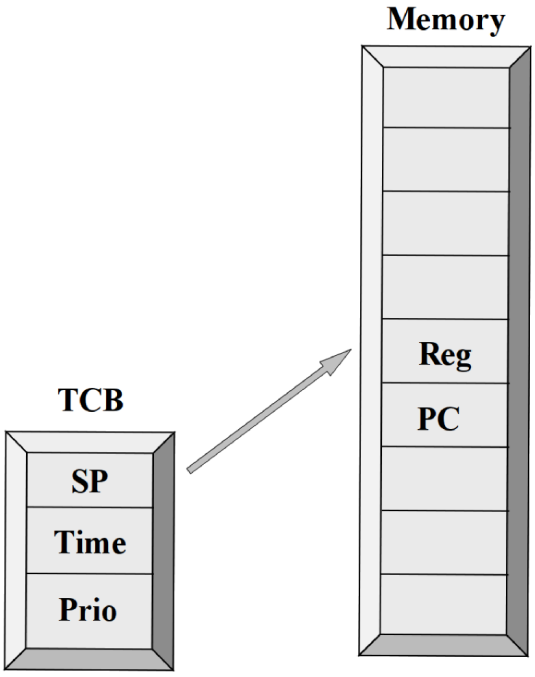
(a) (5p) Antag att schemaläggningsmetoden Earliest Deadline First (EDF) används. Man vill att *P2* ska köra så mycket som möjligt. Alltså, beräkna största möjliga värde på x så att specifikationerna uppfylls. Visa sedan vad som händer när programmet körs genom att rita ett tidsdiagram.
Hur stor blir den faktiska utnyttjandegraden?

(b) (5p) Antag att schemaläggningsmetoden Priority Based Preemptive Scheduling används. Antag samma värde på x som i a-uppgiften, samt att prioriteten mellan processerna är P3 > P1 > P2. Visa vad som händer när programmet körs genom att rita ett tidsdiagram.
Kommer processerna att uppfylla kraven?
Hur stor blir den faktiska utnyttjandegraden?

Uppgift 2: Skapa och starta process(10p)

Betrakta situationen i figur 1 nedan. Den visar TCB (Task Control Block) för en icke körande process med tillhörande minnesutrymme. Vi kan kalla denna process för *P0*.

Beskriv de olika steg som genomgås för att skapa och sedan starta ytterligare en process *P1* (antag att *P1* skapas med högre prioritet än *P0*). Tänk på att i din beskrivning ta med vilka listor som är aktuella, hur minnet påverkas, hur och vilka delar av processorn som används. De olika stegen måste beskrivas i rätt ordning för full poäng.



Figur 1: En process, icke körande.

Uppgift 3: Ätande filosofer(10p)

Man önskar ta fram ett realtidsprogram i C som simulerar det klassiska problemet med ätande filosofer. Filosoferna ägnar sitt liv åt att äta och att filosofera om vartannat. Programmet ska uppfylla följande krav:

- Filosoferna sitter vid ett runt bord och har var sin tallrik med mat samt var sin gaffel.
- För att äta måste dock en filosof ha två gafflar, dvs förutom den egna gaffeln, som ligger till vänster, måste en filosof använda gaffeln från sin högra bordsgranne.
- Det finns totalt *N_FORKS* gafflar.
- När en filosof ska äta får hen bara ta upp en gaffel åt gången, dvs om filosofen börjar med att ta upp vänster gaffel, så måste någon annan ha möjlighet att ta upp eller lägga ned en gaffel, innan filosofen tar upp den högra gaffeln.
- Om en filosof bara får tag i en gaffel vid ett försök att börja äta så måste den gaffeln läggas tillbaka innan ett nytt försök görs.
- Det får inte uppstå deadlock eller svält i systemet.
- Programmet får inte vara onödigt processorintensivt.
- En filosof ska alternerande äta och filosofera, men får inte övergå till att filosofera om hen inte först ätit. När filosofen filosoferat ska hen övergå till att äta.
- En filosof får inte blockera någon annan filosof från att börja äta då det är möjligt.
- Det tar fyra sekunder för en filosof att äta, samt fyra sekunder för en filosof att filosofera. I övergången från att filosofera till att äta kan det dock ta en obestämd tid beroende på hur snart filosofen får tag på två gafflar.
- Programmet ska lösas med en monitor, innehållande en semafor och en händelsevariabel. Utöver detta får monitorn förses med nödvända variabler för att lösa uppgiften.

Det finns ett huvudprogram *main* som ser ut enligt följande:

```
int main(void)
{
  /* initialise kernel */
  si_kernel_init();

  /* set the table */
  table = create_table()

  /* create tasks */
  int i;
  for (i=0; i<N_FORKS; i++)
  {
    /* create task with arguments (task_function, stack_space, priority) */
    si_task_create(philo_task, &Philo_stack[i][STACK_SIZE-1], 10+i);
  }

  /* start the kernel */
  si_kernel_start();

  /* will never be here */
  return 0;
}
```

Uppgiften fortsätter på nästa sida

Vissa definitioner och funktioner är också redan gjorda:

```
/* any number of forks more than 1 should work */
#define N_FORKS ...
#define STACK_SIZE ...

stack_item Philosopher_stack[N_FORKS][STACK_SIZE];

/* data structure for dinner table */
typedef struct
{
  /* the forks, 0=unused, 1=used */
  int fork[N_FORKS];

  /* philosopher id */
  int id;

  /* semaphore for protection of the dinner table */
  si_semaphore mutex;

  /* event variable to indicate if a fork has been put down or lifted up */
  si_condvar change;
} dinner_table_type;

typedef dinner_table_type* table_type;

table_type table;

/* create table: creates a table and initialises the
created table */
table_type create_table(void)
{
  /* reference to the created table */
  table_type table;

  /* allocate memory */
  table = (table_type) malloc(sizeof(dinner_table_type));

  /* no forks initially used */
  int i;
  for (i=0; i<N_FORKS; i++) {
    table->fork[i] = 0; }

  /* first philospher id */
  table->id = 0;

  /* initialise semaphore and event variable */
  si_sem_init(&table->mutex, 1);
  si_cv_init(&table->change, &table->mutex);

  return table;
}
```

Uppgiften fortsätter på nästa sida

Slutligen den ofärdiga filosofprocessen:

```
void philo_task(void)
{
  /* unfinished */
}
```

Du får inte ta bort något av huvudprogrammet *main*, men eventuella nödvändiga tillägg får göras. Nödvändiga programbibliotek antas vara inkluderade och nödvändiga definitioner för det givna huvudprogrammet antas vara gjorda. Du får deklarera och lägga till egna funktioner.

Följande funktioner i Simple-OS finns tillgängliga:

```
/* signal operation on semaphore sem */
void si_sem_signal(si_semaphore *sem)

/* wait operation on semaphore sem */
void si_sem_wait(si_semaphore *sem)

/* broadcast operation on cv */
void si_cv_broadcast(si_condvar *cv)

/* wait operation on cv */
void si_cv_wait(si_condvar *cv)

/* makes the calling process wait n_ms milliseconds */
void si_wait_n_ms(int n_ms)

/* initialisation of semaphore sem */
void si_sem_init(si_semaphore *sem, int init_val)

/*initialisation of condvar cv */
void si_cv_init(si_condvar *cv, si_semaphore *mutex)
```

Din uppgift är att skriva filosofprocessen `philo_task`.

Glöm inte att programkoden måste vara kommenterad för full poäng.

Uppgift 4: Teori(8p)

(a) (2p) *Wait* och *Signal* är realtidsoperationer för en semafor. Ge exempel på och förklara hur det kan vara riskfyllt att använda en semafor i ett avbrott.

(b) (2p) *Await* och *Cause* är realtidsoperationer för en händelsevariabel. Förklara varför det är en dålig idé att använda dessa för synkronisering (symmetrisk eller asymmetrisk) av processer.

(c) (2p) I ett realtidsoperativsystem startas processer/trådar genom att anropa t ex `fork` eller `pthread_create`. Varför kan man inte starta en process/tråd genom att bara direkt anropa den funktion som utgör processen/tråden?

(d) (2p) Antag att en gemensam resurs används av två processer *P1* och *P2*. Antag att den ena processen, *P1*, läser från den gemensamma resursen, och den andra processen, *P2*, skriver till den gemensamma resursen. En semafor, initierad till ett, används av *P2* i dess kritiska region när data skrivs.
Varför är det viktigt att även *P1* använder en semafor till sin kritiska region när data bara läses?
Är semaforen nödvändig när data läses av *P1*, om *P1* har högre prioritet än *P2*?
Motivera dina svar.

```
/* P1 */                          /* P2 */
wait(S)                           wait(S)
    read common resource              write common resource
signal(S)                         signal(S)
```

Uppgift 5: Semaforer och villkorsvariabler(12p)

Betrakta följande program i Simple-OS.

```
#include <simple_os.h>
#include <stdio.h>

#define STACK_SIZE 5000

/* define task stack spaces */
stack_item p1_stack[STACK_SIZE];
stack_item p2_stack[STACK_SIZE];
stack_item p3_stack[STACK_SIZE];

si_semaphore S; // define semaphore
si_condvar CV;  // define condition variable

int item = 1;


void P1(void)
{
    printf("A1\n");
    si_sem_wait(&S);        // Wait
    printf("B1\n");
    while (item != 1)       // item not 1
    {
        printf("C1\n");
        si_cv_wait(&CV);    // Await
        printf("D1\n");
    }
    item=2;
    printf("E1\n");
    si_cv_broadcast(&CV);   // Cause
    printf("F1\n");
    si_sem_signal(&S);      // Signal
    printf("G1\n");
    while(1);
}
```

Uppgiften fortsätter på nästa sida

```
void P2(void)
{
    printf("A2\n");
    si_sem_wait(&S);        // Wait
    printf("B2\n");
    while (item != 2)       // item not 2
    {
        printf("C2\n");
        si_cv_wait(&CV);    // Await
        printf("D2\n");
    }
    item=3;
    printf("E2\n");
    si_cv_broadcast(&CV);   // Cause
    printf("F2\n");
    si_sem_signal(&S);      // Signal
    printf("G2\n");
    while(1);
}

void P3(void)
{
    printf("A3\n");
    si_sem_wait(&S);        // Wait
    printf("B3\n");
    while (item != 3)       // item not 3
    {
        printf("C3\n");
        si_cv_wait(&CV);    // Await
        printf("D3\n");
    }
    item = 1;
    printf("E3\n");
    si_cv_broadcast(&CV);   // Cause
    printf("F3\n");
    si_sem_signal(&S);      // Signal
    printf("G3\n");
    while(1);
}
```

Uppgiften fortsätter på nästa sida

```
/* main program */
int main(void)
{
    /* initialise simple OS kernel */
    si_kernel_init();

    /* initialise semaphore to 1 */
    si_sem_init(&S, 1);

    /* associate condition variable with semaphore */
    si_cv_init(&CV, &S);

    /* create tasks */
    si_task_create(P1, &p1_stack[STACK_SIZE-1], 20); // low priority
    si_task_create(P2, &p2_stack[STACK_SIZE-1], 15); // middle priority
    si_task_create(P3, &p3_stack[STACK_SIZE-1], 10); // high priority

    /* start the kernel, also starting tasks */
    si_kernel_start();

    return 0;
}
```

Beskriv steg för steg vad som händer ifrån det att processerna `P1`, `P2` och `P3` är körklara. Var noggrann med att tala om vilken process som är körande, vilka listor processerna ligger i vid olika tillfällen, semaforens värde, värdet på variabeln `item` samt motivera varför olika händelser sker. Listornas exakta namn är inte viktigt, bara det framgår vad deras syfte är.

Redovisa gärna stegvis i tabellform där det framgår vad som är orsak och verkan. Ge en kommentar för varje steg.

Ange också den resulterande utskriften.

Lösningsförslag fråga 1

Följande notation gäller:

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

1a

För EDF gäller att kraven uppfylls om utnyttjandegraden $U_e \leq 1$, dvs:
$4/14 + x/7 + 1/4 \leq 1$
$8/28 + 4x/28 + 7/28 \leq 28/28$
$4x \leq 13$
$x \leq 13/4$
$13/4$ är större än 3 men mindre än 4, alltså, största värde på x är 3.

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_ # # # _ _ _|_ _ _ # # # _|# # _ # _ _ _|_ _ # # # _ .|
P3# _ _ _|# _ _ _|# _ _ _|_ # _ _|# _ _ _|# _ _ _|_ _ # .|
 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 (förstås) sina deadlines och den faktiska utnyttjandegraden blir 27/28 (ty x=3 ger Ue=27/28). 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=3, samt att den faktiska utnyttjandegraden blir 27/28)

1b

Prioriteten är enligt uppgift satt till P3 > P1 > P2. Schemaläggningsmetoden priority based preemptive scheduling ger då:

```
P1_ # # # _ # _ _ _ _ _ _ . _ .|# # _ # # _ _ _ _ _ _ _ . . .|
P2_ _ _ _ _ _ #/# _ # # . _ .|_ _ _ _ _ # _/# # # _ . . .|
P3# _ _ _|# _ _ _|# _ _ .|# . _ _|# _ _ _|# _ _ _|# . . .|
 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 *P3* klarar sina deadlines. *P2* missar deadlines vid t=7 och t=21. Den faktiska utnyttjandegraden blir 23/28. 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, 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 23/28.

Lösningsförslag fråga 2:

Att skapa process *P1* sker i följande ordning:

1. Skapa ett TCB för *P1*, fyll i värden för prioritet och tid (typiskt 0), i TCB för P1.
2. Skapa ett stackutrymme för *P1* i minnet.
3. Skriv värdet för PC, dvs startadressen för den funktion som utgör processen, i stacken tillhörande *P1*.
4. Skriv värden för önskat starttillstånd hos registeruppsättningen, i stacken.
5. Processorns stackpekare SP pekar nu på toppen av stacken, så spara SP i TCB för *P1*.
6. Process *P1* är nu redo att starta, lägg dess TCB i *ReadyList*.

Att starta process *P1* sker i följande ordning:

1. Markera process *P1* med en pekare kallad *Running*.
2. Läs SP från TCB för processen markerad med *Running* (dvs *P1*:s TCB) in till processorns stackpekare.
3. Läs in register från *P1*:s stack till processorn register
4. Läs in PC från *P1*:s stack till processorn PC. Därmed hoppar exekveringen till processens adress och processen startar.

Lösningsförslag fråga 3:

Nedan redovisas hela programmet.

```
#include <simple_os.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* any number of forks more than 1 should work */
#define N_FORKS ...
#define STACK_SIZE ...

stack_item philo_stack[N_FORKS][STACK_SIZE];

/* data structure for dinner table */
typedef struct
{
    /* the forks, 0=unused, 1=used */
    int fork[N_FORKS];

    /* philosopher id */
    int id;

    /* semaphore for protection of the dinner table */
    si_semaphore mutex;

    /* event variable to indicate if a fork has been put down or lifted up */
    si_condvar change;
} dinner_table_type;

typedef dinner_table_type* table_type;

table_type table;


/* create table: creates a table and initialises the
created table */
table_type create_table(void)
{
    /* reference to the created table */
    table_type table;

    /* allocate memory */
    table = (table_type) malloc(sizeof(dinner_table_type));

    /* no forks initially used */
    int i;
    for (i=0; i<N_FORKS; i++) {
        table->fork[i] = 0;
    }

    /* first philospher id */
    table->id = 0;

    /* initialise semaphore and event variable */
    si_sem_init(&table->mutex, 1);
```

```
    si_cv_init(&table->change, &table->mutex);

    return table;
}


void philosopher_eat(table_type table, int philosopher_id)
{
    int left_fork = philosopher_id;
    int right_fork = (philosopher_id + 1) % N_FORKS;
    int finished_eating = 0;

    /* reserve table */
    si_sem_wait(&table->mutex);

    while (!finished_eating)
    {

        /* left fork available? */
        if (table->fork[left_fork] == 0)
        {
            /* pick up left fork */
            table->fork[left_fork] = 1;

            /* give someone else a chance so pick up or put down a fork */
            si_sem_signal(&table->mutex);
            /* wait a randomised time to avoid deadlock */
            si_wait_n_ms((rand() % 2000) + 500);
            /* reserve table */
            si_sem_wait(&table->mutex);

            /* right fork available? */
            if (table->fork[right_fork] == 0)
            {
                /* pick up right fork */
                table->fork[right_fork] = 1;

                /* don't block others while eating */
                si_sem_signal(&table->mutex);

                printf("Philosopher %d eating\n", philosopher_id);
                /* simulate eating */
                si_wait_n_ms(4000);
                /* reserve table */
                si_sem_wait(&table->mutex);
                finished_eating = 1;

                /* put down right fork */
                table->fork[right_fork] = 0;

                /* put down left fork */
                table->fork[left_fork] = 0;

                /* tell others table has changed */
                si_cv_broadcast(&table->change);

            }
```

```
            else
            {
                /* right fork not available, so put down left fork
                to give someone else a chance to eat */
                table->fork[left_fork] = 0;

                /* tell others table has changed */
                si_cv_broadcast(&table->change);
            }
        }

        if (!finished_eating)
        /* wait for a change, a chance to eat */
        si_cv_wait(&table->change);

    }

    /* release table */
    si_sem_signal(&table->mutex);
}


void philosopher_think(int philosopher_id)
{
    printf("Philosopher %d thinking\n", philosopher_id);
    /* simulate thinking */
    si_wait_n_ms(4000);
}


void philo_task(void)
{
    /* reserve table */
    si_sem_wait(&table->mutex);
    /* generate unique id */
    int id = table->id++;
    /* release table */
    si_sem_signal(&table->mutex);

    while(1)
    {
        /* eat */
        philosopher_eat(table, id);
        /* think */
        philosopher_think(id);
    }
}


/* main */
int main(void)
{
    /* initialise kernel */
    si_kernel_init();

    /* set the table */
    table = create_table();
```

```
    /* create philosopher tasks */
    int i;
    for (i=0; i<N_FORKS; i++)
    {
        si_task_create(philo_task,
        &Philo_stack[i][STACK_SIZE-1], 10+i);
    }

    /* start the kernel */
    si_kernel_start();

    /* will never be here */
    return 0;
}
```

Lösningsförslag fråga 4:

4a

Om avbrottet innehåller *Wait* och blir väntande på semaforen så kommer den process som avbröts (inte själva avbrottet) att läggas i semaforens väntelista, och den processen har måhända inte med avbrottet/semaforen att göra, vilket kan få oväntade konsekvenser.

4b

Await kommer förvisso att flytta körande process till en väntelista (händelsevariabelns), men *Cause* flyttar sedan den process som är tänkt att synkroniseras (starta upp igen) från händelsevariablens väntelista till en annan väntelista (semaforens). Dvs, Det krävs utöver detta en *Signal*-operation för att den synkroniserade processen om möjligt ska kunna bli körande igen.

4c

Om själva funktionen direkt anropas, istället för att startas (t ex med pthread_create, eller via fork) så kommer funktionen bara att köras, inte skapa en tråd/process med tillhörande minnesutrymme för stack och processkontrollblock och inte bli en del av de parallella processerna i systemet mellan vilka en schemaläggare kan växla.

4d

När *P1* läser kan den tänkas bli avbruten, och om läsningen inte föregåtts av en *Wait*-operation som sätter semaforen till 0, kan den avbrytande processen *P2* fortsätta förbi sin *Wait*-operation, modifiera gemensamma data, vilket kan göra det lästa datat inkonsistent för *P1* om läsningen inte är atomär. I ett scenario där den läsande *P1* har högre prioritet än den skrivande *P2* kan *P2* avbrytas. Om läsningen i *P1* då inte föregås av en *Wait*-operation så kan *P1* läsa ej färdigskrivna data, och datat kan bli inkonsistent för *P1*.

Lösningsförslag fråga 5:

Programmet skriver ut:
A3 B3 C3 A2 B2 C2 A1 B1 E1 F1 D3 C3 D2 E2 F2 D3 E3 F3 G3

I följande tabell gäller att S är semaforens värde, R är Ready-listan, WS är vänte-listan för semaforen S, WCV är väntelistan för händelsevariabeln CV, item är värdet på variabeln item.

	Orsak	Verkan					
		Kör	S	R	WS	WCV	item
1)	Init	-	1	P1,P2,P3			1
2)	Sched	P3	1	P1,P2,P3			1
3)	P3:Wait	P3	0	P1,P2,P3			1
4)	P3:Await	P3	1	P1,P2		P3	1
5)	Sched	P2	1	P1,P2		P3	1
6)	P2:Wait	P2	0	P1,P2		P3	1
7)	P2:Await	P2	1	P1		P2,P3	1
8)	Sched	P1	1	P1		P2,P3	1
9)	P1:Wait	P1	0	P1		P2,P3	1
10)	P1:item=2	P1	0	P1		P2,P3	2
11)	P1:Cause	P1	0	P1	P2,P3		2
12)	P1:Signal	P1	0	P1,P3	P2		2
13)	Sched	P3	0	P1,P3	P2		2
14)	P3:Await	P3	0	P1,P2		P3	2
15)	Sched	P2	0	P1,P2		P3	2
16)	P2:item=3	P2	0	P1,P2		P3	3
17)	P2:Cause	P2	0	P1,P2	P3		3
18)	P2:Signal	P2	0	P1,P2,P3			3
19)	Sched	P3	0	P1,P2,P3			3
20)	P3:item=1	P3	0	P1,P2,P3			1
21)	P3:Cause	P3	0	P1,P2,P3			1
22)	P3:Signal	P3	1	P1,P2,P3			1
23)	P3:while(1)						

1) P1,P2 och P3 blir körklara, S initieras till 1.

2) Schemaläggaren startar P3 (högst prio i R)

3) P3 gör Wait, ingen väntar på Sem så S–

4) P3 gör Await (ty item!=3), ingen väntar på Sem så S++

5) Schemaläggaren väljer P2 (högst prio i R)

6) P2 gör Wait, ingen väntar på Sem så S–

7) P2 gör Await (ty item!=2), ingen väntar på Sem så S++

8) Schemaläggaren väljer P1 (högst prio i R)

9) P gör Wait, ingen väntar på Sem så S–

10) P1 gör item=2

11) P1 gör Cause så P2 och P3 till WS

12) P1 gör Signal, P2 och P3 väntar på Sem så P3 (högst prio i WS) till R

13) Schemaläggaren väljer P3 (högst prio i R)

14) P3 gör Await (ty item!=3), P2 väntar på Sem så S oförändrad

15) Schemaläggaren väljer P2 (högst prio i R)

16) P2 gör item=3

17) P2 gör Cause så P3 till WS

18) P2 gör Signal, P3 väntar på Sem så P3 (högst prio i WS) till R

19) Schemaläggaren väljer P3 (högst prio i R)

20) P3 gör item=1

21) P3 gör Cause, ingen väntar i WCS

22) P3 gör Signal, ingen väntar i WS så S++

23) P3 gör while(1), då P3 har högst prio sker inga fler processbyten

(Kommentar: 1 poäng ges för varje korrekt utfört steg enligt ovan, vid följande tillfällen:
Steg 3, 4-5, 6, 7-8, 9, 11, 12-13, 14-15, 17, 18-19, 21, 22.
Det krävs även att värdet på semaforen S är korrekt, att listor för Ready-list, väntelista för Semafor och väntelista för händelsevariabel CV finns och är korrekta, och att värdet på item redovisas.
Kommentarer för stegen blir viktiga då de visar resonemanget och därmed avgör om fortsättningen är poänggivande även efter att något har blivit fel i ett steg.)