Tentamen
Datorteknik och realtidssystem, TSEA81

Datum	2023-08-18
Lokal	FB113
Tid	14-18
Utb.kod	TSEA81
Modul	TEN1
Utb.kodnamn	Datorteknik och realtidssystem
Institution	ISY
Antal uppgifter	5
Antal sidor (inklusive denna sida)	15
Kursansvarig	Kent Palmkvist
Lärare som besöker skrivsalen *Telefon under skrivtiden*	Kent Palmkvist 013-28 1347
Besöker skrivsalen	Ca 15 och 17
Kursadministratör	Ulrika Ericsson, 013-28 2379
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, dvs endast en process åt gången kan exekvera. Följande krav gäller:

- *P1* ska arbeta/köra under 1 tidsenheter i tidsintervallet [i*n, (i+1)*n]
- *P2* ska arbeta/köra under 3 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 $i \geq 0$.

Tideräkningen startar vid t=0 för alla processer. Du kan anta att uppstart av processer och processbyte inte tar någon tid alls. 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 ofta som möjligt, dvs beräkna minsta möjliga värde på n 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, samt att n har samma värde som i uppgift (a). Visa vad som händer när programmet körs genom att rita ett tidsdiagram.
Kommer specifikationerna ovan att uppfyllas? Hur stor blir den faktiska utnyttjandegraden?

Uppgift 2: Await / Cause(10p)

Följande programrader är listningar av funktionerna Await och Cause. Tyvärr förekommer ett antal fel i programkoden. Rätta till felen genom att t ex ange vilka rader som behöver ändras och vad det ska stå där för att få korrekt funktion. Alternativt, skriv upp hela den korrekta funktionen för `si_ev_wait` och `si_ev_cause`. Du behöver inte kommentera programkoden.

OBSERVERA! Om du ändrar på det som redan är korrekt så medför det poängavdrag! Uppgiften kan dock inte ge mindre än 0 poäng.

```
00  /* Await */
01  void si_ev_wait(si_event *ev)
02  {
03      int pid;
04      DISABLE_INTERRUPTS;
05      if (!ready_list_is_empty(ev->wait_list,
06                                   WAIT_LIST_SIZE))
07      {
08          pid = wait_list_insert_highest_prio(ev->wait_list,
09                                              WAIT_LIST_SIZE);
10          wait_list_insert(pid);
11      }
12      else
13      {
14          ev->mutex->counter++;
15      }
16      pid = process_get_pid_running();
17      wait_list_remove(pid);
18      wait_list_insert(ev->wait_list, WAIT_LIST_SIZE, pid);
19      schedule();
20      ENABLE_INTERRUPTS;
21  }
22
23  /* Cause */
24  void si_ev_cause(si_event *ev)
25  {
26      int done;
27      int pid;
28      done = wait_list_is_empty(ev->wait_list, WAIT_LIST_SISZE);
29      DISABLE_INTERRUPTS;
30      ev->mutex->counter--;
31      while(!done)
32      {
33          pid = ready_list_remove_one(ev->mutex->wait_list,
34                                          WAIT_LIST_SIZE);
35          wait_list_remove(ev->mutex->wait_list, WAIT_LIST_SIZE, pid);
36          done = wait_list_is_empty(ev->wait_list, WAIT_LIST_SIZE);
37      }
38      ENABLE_INTERRUPTS;
39  }
```

Uppgift 3: Stark/svag semafor(12p)

Betrakta följande program i Simple-OS. Antag att semaforerna fungerar på så sätt som beskrivits under kursens föreläsningar. Antag att prioritetsbaserad påtvingad schemaläggning används.

```
#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];

si_semaphore S1;          // define semaphore


/* do some dummy work for an
unspecified time */
void do_work()
{
    int i;
    volatile int dummy;
    for(i=0; i < 99000000;i++){
        dummy=i;
    }
}


/* task p1 */
void p1(void)
{
    si_sem_wait(&S1);       // wait on semaphore
    printf("A\n");
    si_wait_n_ms(2000);     // sleep for 2000 ms
    printf("B\n");
    si_sem_signal(&S1);     // signal on semaphore
    printf("C\n");
    do_work();              // do some work for some time
    printf("D\n");
    si_sem_wait(&S1);       // wait on semaphore
    printf("E\n");
    si_wait_n_ms(2000);     // sleep for 2000 ms
    printf("F\n");
    si_sem_signal(&S1);     // signal on semaphore
    printf("G\n");
    while(1);               // wait for ever
}
```

```
/* task p2 */
void p2(void)
{
    si_wait_n_ms(1000);    // sleep for 1000 ms
    printf("H\n");
    si_sem_wait(&S1);      // wait on semaphore
    printf("I\n");
    do_work();             // do some work for some time
    printf("J\n");
    si_sem_signal(&S1);    // signal om semaphore
    printf("K\n");
    while(1);              // wait for ever
}


/* main program */
int main(void)
{
    /* initialise simple OS kernel */
    si_kernel_init();
    /* initialise semaphore to 1 */
    si_sem_init(&S1, 1);
    /* create tasks */
    si_task_create(p1, &p1_stack[STACK_SIZE-1], 10); // high priority
    si_task_create(p2, &p2_stack[STACK_SIZE-1], 20); // low priority
    /* start the kernel, also starting tasks */
    si_kernel_start();

    return 0;
}
```

(a) (2p) Antag att programmet ovan använder sig av starka semaforer. Vad blir den resulterande utskriften när programmet kör?

(b) (8p) Beskriv steg för steg vad som händer i uppgift (a) från det att de båda processerna p1 och p2 är körklara. Var noga med att tala om vilka processer som är körande, vilka listor dom ligger i vid olika tillfällen, semaforens värde samt motivera varför olika händelser sker. Listornas exakta namn är inte viktigt, bara det framgår vad deras syfte är.

(c) (2p) Antag att programmet ovan använder sig av svaga semaforer. Vad blir den resulterande utskriften när programmet kör?

Uppgift 4: Monitor(12p)

Man önskar ta fram ett program med tre processer, *A*, *B* och *C*. Processerna ska kommunicera via en gemensam buffer.

Processen *A* ska kontinuerligt producera/skriva data till buffern.

Processen *B* ska konsumera/läsa var tredje data som skrivits i buffern, med början på det första datat, samt skriva ut det data som lästs.

Processen *C* ska konsumera/läsa två på varandra följande data i buffern, med början på det andra datat, samt skriva ut de data som lästs.

Låt processen *A* producera data enligt en uppräkning, såsom: 0, 1, 2, 3, 4, 5, 6, ... osv.

Processen *B* ska då konsumera data enligt följande: 0, 3, 6, ... osv.

Processen *C* ska då konsumera data enligt följande: 1, 2, 4, 5, ... osv.

Utskriften skulle alltså kunna se ut enligt följande:

```
B:0
C:1
C:2
B:3
C:4
C:5
B:6
...
```

Din uppgift är att skriva programkoden (C-kod) för processerna *A*, *B* och *C*.

Själva huvudprogrammet och initieringar är redan gjorda enligt nedan. Du får INTE göra ytterligare globala definitioner eller programsatser. Dvs, du får endast skriva den kod som ingår i processerna *A*, *B* och *C*.

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

#define BUFFER_SIZE 10

/* buffer */
struct
{
    int data[BUFFER_SIZE];
    int in_pos;
    int out_pos;
    int count;
    pthread_mutex_t M;
    pthread_cond_t C;
} buf;

sem_t S1;
```

```
sem_t S2;
sem_t S3;

void *A_thread(void *unused)
{
    ...
}

void *B_thread(void *unused)
{
    ...
}

void *C_thread(void *unused)
{
    ...
}

int main(int argc, char **argv)
{
    pthread_mutex_init(&buf.M, NULL);
    buf.in_pos = 0;
    buf.out_pos = 0;
    buf.count = 0;

    sem_init(&S1, 0, 0);
    sem_init(&S2, 0, 0);
    sem_init(&S3, 0, 0);

    pthread_t A_thread_handle;
    pthread_t B_thread_handle;
    pthread_t C_thread_handle;

    pthread_create(&A_thread_handle, NULL, A_thread, 0);
    pthread_create(&B_thread_handle, NULL, B_thread, 0);
    pthread_create(&C_thread_handle, NULL, C_thread, 0);

    pthread_join(A_thread_handle, NULL);
    pthread_join(B_thread_handle, NULL);
    pthread_join(C_thread_handle, NULL);

    while(1);
}
```

Observera, din programkod måste vara kommenterad för full poäng.

Uppgift 5: Teori(6p)

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

(b) (1p) Om två processer kör parallellt i var sin kärna i samma processor och exakt samtidigt försöker låsa samma mutex, hur bestäms det då vilken process som får låsa mutexen?

(c) (3p) Ange minst tre orsaker till att ett processbyte sker. Förutsättningen behöver framgå och det ska vara orsaker där processbytet faktiskt sker, inte eventuellt sker.

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:
$1/n + 3/7 + 1/4 \leq 1$
$1/n + 12/28 + 7/28 \leq 28/28$
$1/n \leq 9/28$
$28/9 \leq n$
$n \geq 28/9$
Alltså, n=3 räcker inte (3=27/9) så minsta värde på n är 4.

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 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
  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 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), korrekt dra slutsatsen om vilka processer som klarar sina deadlines, att värdet på n=4, 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. Då det är angivet att P1 ska köras så ofta som möjligt kan P1 anses ha högre än prioritet än P2 trots att båda har samma deadlines. Prioriteten bli alltså P1 > P2 > P3. Därefter används schemaläggningsmetoden priority based preemptive scheduling.

```
P1# _ _ _|_ # _ _|# _ _ _|# _ _ _|# _ _ _|# _ _ _|# _ _ _|
P2_ _ # # # _ _|# _ _ # # _ _|# # _ _ # _ _|_ # # _ _ # _|
P3_ # _ _|_ _ # _|_ # _ _|_ # _ _|_ # _ _|_ # _ _|_ # _ _|
  0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
  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 möter sina deadlines. Den faktiska utnyttjandegraden blir 26/28. 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, tala om de inbördes prioriteterna för P1, P2 och P3 vid RMS, korrekt dra slutsatsen om vilka processer som klarar sina deadlines, visa outnyttjade tidsintervall, samt att den faktiska utnyttjandegraden blir 13/14.)

Lösningsförslag fråga 2:

Antingen bara de korrigerade programraderna:

```
05      if (!wait_list_is_empty(ev->mutex->wait_list,

08          pid = wait_list_remove_highest_prio(ev->mutex->wait_list,

10          ready_list_insert(pid);

17      ready_list_remove(pid);

28      DISABLE_INTERRUPTS;
29      done = wait_list_is_empty(ev->wait_list, WAIT_LIST_SISZE);

30

33          pid = wait_list_remove_one(ev->wait_list,

35          wait_list_insert(ev->mutex->wait_list, WAIT_LIST_SIZE, pid);
```

... eller den fullständiga programkoden (här med kommentarer):

```
00  /* Await */
01  void si_ev_wait(si_event *ev)
02  {
03      int pid;
04      DISABLE_INTERRUPTS;
05      if (!wait_list_is_empty(ev->mutex->wait_list,
06                                  WAIT_LIST_SIZE))
07      {
08          pid = wait_list_remove_highest_prio(ev->mutex->wait_list,
09                                              WAIT_LIST_SIZE);
10          ready_list_insert(pid);
11      }
12      else
13      {
14          ev->mutex->counter++;
15      }
16      pid = process_get_pid_running();
```

```
17      ready_list_remove(pid);
18      wait_list_insert(ev->wait_list, WAIT_LIST_SIZE, pid);
19      schedule();
20      ENABLE_INTERRUPTS;
21  }
22
23  /* Cause */
24  void si_ev_cause(si_event *ev)
25  {
26      int done;
27      int pid;
28      DISABLE_INTERRUPTS;
29      done = wait_list_is_empty(ev->wait_list, WAIT_LIST_SISZE);
30
31      while(!done)
32      {
33          pid = wait_list_remove_one(ev->wait_list,
34                                     WAIT_LIST_SIZE);
35          wait_list_insert(ev->mutex->wait_list, WAIT_LIST_SIZE, pid);
36          done = wait_list_is_empty(ev->wait_list, WAIT_LIST_SIZE);
37      }
38      ENABLE_INTERRUPTS;
39  }
```

(Kommentar: Korrekta korrektioner ger 1p. Felaktiga korrektioner (dvs ändring av sådant som redan är korrekt) ger -1p. Om slutsumman blir negativ ges 0p.)

Lösningsförslag fråga 3:

3a

A H B C D I J E K F G

Eftersom en stark semafor förhindrar svält så kan inte p1 får tillgång till semaforen en andra gång (efter D) eftersom p2 då begärt tillgång till semaforen under tiden som p1 hade tillgång till semaforen första gången.

3b

Här finns tre tillfällen som kan orsaka ett processbyte. När en process gör sleep (och en annan process är körklar), 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).

	p1	p2	körande	counter	T	W	R
1)			p1	1			p1,p2
2)	wait		p1	0			p1,p2
3)	sleep(2)	sleep(1)	-	0	p1,p2		-
4)		wait	(p2)	0	p1	p2	(p2)
5)	signal		p1	0			p1,p2
6)	wait		p2	0		p1	p2
7)		signal	p1	0			p1,p2
8)	sleep(2)		p2	0	p1		p2
9)	signal		p1	1			p1,p2

1) Från det att båda processerna är körklara blir p1 (högst prioritet) körande.

2) p1 gör wait (W tom) counter räknar ned

3) printf(A), p1 gör sleep(2), p2 blir körande och gör sleep(1) varpå båda ligger i S

4) p2 vaknar först, printf(H), p2 gör wait (counter==0) och läggs i W

5) p1 vaknar, printf(B), kör signal (p2 i W, så counter oförändrad, p2 till R), och p1 med högst prio fortsätter, printf(C), do_work, printf(D)

6) p1 gör wait (counter==0) och läggs i W, så p2 blir körande, printf(I), do_work, printf(J)

7) p2 gör signal (p1 i W och högst prio), p1 blir körande (counter oförändrad), printf(E)

8) p1 gör sleep(2) (p1 till S), p2 blir körande, printf(K), p2 går in i oändlig while-loop

9) p1 vaknar, har högst prio och blir körande, printf(F), gör signal (W tom så counter räknar upp), printf(G), p1 går in i oändlig while-loop

3c

A H B C D E F G

Eftersom en svag semafor tillåter svält så kan p1 få tillgång till semaforen direkt även den andra gången. Processbyte till p2 kommer endast att ske när p1 gör sleep (efter A), därefter kör p1 hela tiden fram till den oändliga while-loopen varefter inga fler processbyten sker. Dvs, p2 fastnar för evigt vid wait.

Lösningsförlag fråga 4:

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

#define BUFFER_SIZE 10

/* buffer */
struct
{
```

```
    int data[BUFFER_SIZE];
    int in_pos;
    int out_pos;
    int count;
    pthread_mutex_t M;
    pthread_cond_t C;
} buf;

sem_t S1;
sem_t S2;
sem_t S3;

void *A_thread(void *unused)
{
    int num = 0;
    while(1)
    {
        pthread_mutex_lock(&buf.M);
        while (buf.count == BUFFER_SIZE)
        {
            pthread_cond_wait(&buf.C, &buf.M);
        }

        buf.data[buf.in_pos++] = num++;

        buf.count++;
        pthread_cond_broadcast(&buf.C);

        if (buf.in_pos == BUFFER_SIZE)
            buf.in_pos = 0;
        pthread_mutex_unlock(&buf.M);

        sem_post(&S1);
    }
}


void *B_thread(void *unused)
{
    int data;
    while(1)
    {
        sem_wait(&S1);

        pthread_mutex_lock(&buf.M);

        while(buf.count == 0)
        {
            pthread_cond_wait(&buf.C, &buf.M);
        }
```

```
        data = buf.data[buf.out_pos++];

        buf.count--;
        pthread_cond_broadcast(&buf.C);

        if (buf.out_pos == BUFFER_SIZE)
            buf.out_pos = 0;

        printf("B:%d\n", data);

        pthread_mutex_unlock(&buf.M);

        sem_post(&S2);
        sem_wait(&S3);
    }
}


void *C_thread(void * unused)
{
    int data;
    int i;
    while(1)
    {
        sem_wait(&S2);

        pthread_mutex_lock(&buf.M);

        for (i=0; i<2; i++)
        {
            while(buf.count == 0)
            {
                pthread_cond_wait(&buf.C, &buf.M);
            }

            data = buf.data[buf.out_pos++];
            buf.count--;

            pthread_cond_broadcast(&buf.C);

            if (buf.out_pos == BUFFER_SIZE)
                buf.out_pos = 0;

            printf("C:%d\n", data);
        }

        pthread_mutex_unlock(&buf.M);

        sem_post(&S3);
```

```
    }
}

int main(int argc, char **argv)
{
    pthread_mutex_init(&buf.M, NULL);
    buf.in_pos = 0;
    buf.out_pos = 0;
    buf.count = 0;

    sem_init(&S1, 0, 0);
    sem_init(&S2, 0, 0);
    sem_init(&S3, 0, 0);

    pthread_t A_thread_handle;
    pthread_t B_thread_handle;
    pthread_t C_thread_handle;

    pthread_create(&A_thread_handle, NULL, A_thread, 0);
    pthread_create(&B_thread_handle, NULL, B_thread, 0);
    pthread_create(&C_thread_handle, NULL, C_thread, 0);

    pthread_join(A_thread_handle, NULL);
    pthread_join(B_thread_handle, NULL);
    pthread_join(C_thread_handle, NULL);

    while(1);
}
```

Lösningsförslag fråga 5

5a

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.

5b

Det avgörs i CPU:ns hårdvara, via speciella atomiska assemblerinstruktioner, t ex TAS (Test And Set) eller CAS (Compare And Swap).

5c

En ny process skapas med högst prioritet. Körande process anropar wait på en semafor och semaforen är upptagen. En process gör signal på en semafor som en annan process med högre prioritet väntar på.

*Kommentar: Det behöver tydligt framgå av vilket skäl som processbytet sker och vilka förutsättningarna är. Observera att det efterfrågas orsaker till att ett processbyte *sker*, inte orsaker till att det *eventuellt* sker. Det är t ex otillräckligt att säga "Ett avbrott startar en process med högst prioritet", eller liknande. Vilket avbrott? Av vilket skäl börjar i så fall en ny process köra?*