Tentamen
Datorteknik och realtidssystem, TSEA81

Datum	2023-01-09
Lokal	TER4, U10
Tid	8-12
Kurskod	TSEA81
Provkod	TEN1
Kursnamn	Datorteknik och realtidssystem
Institution	ISY
Antal uppgifter	5
Antal sidor (inklusive denna sida)	8
Kursansvarig	Kent Palmkvist
Lärare som besöker skrivsalen *Telefon under skrivtiden*	Kent Palmkvist 013-28 1347
Besöker skrivsalen	Ca 9 och 11
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/kärna, dvs endast en process åt gången kan exekvera. Följande krav gäller:

- *P1* ska arbeta/köra under n tidsenheter i tidsintervallet [i*12, (i+1)*12]
- *P2* ska arbeta/köra under 3 tidsenheter i tidsintervallet [i*8, (i+1)*8]
- *P3* ska arbeta/köra under 1 tidsenhet i tidsintervallet [i*3, (i+1)*3]

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

Tidräkningen 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 mycket som möjligt, dvs beräkna största möjliga värde på n så att specifikationerna uppfylls. Visa sedan vad som händer när processerna 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 processerna körs genom att rita ett tidsdiagram.
Kommer specifikationerna ovan att uppfyllas? Hur stor blir den faktiska utnyttjandegraden?

Uppgift 2: Wait / Signal(8p)

Följande programrader är listningar av funktionerna Wait och Signal för en ***svag*** semafor. 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_sem_wait` och `si_sem_signal`. 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  /* Wait */
01  void si_sem_wait(si_semaphore *sem)
02  {
03      int task_id;
04      DISABLE_INTERRUPTS;
05      while (sem->counter > 0)
06      {
07          task_id = task_get_task_id_running();
08          wait_list_remove(
                sem->wait_list, WAIT_LIST_SIZE, task_id);
09          ready_list_insert(task_id);
10      }
11      schedule();
12      sem->counter++;
13      ENABLE_INTERRUPTS;
14  }
15
16  /* Signal */
17  void si_sem_signal(si_semaphore *sem)
18  {
19      int task_id;
20      int call_schedule = 1;
21      DISABLE_INTERRUPTS;
22      if (!wait_list_is_empty(
            sem->wait_list, WAIT_LIST_SIZE))
23      {
24          task_id = wait_list_remove_highest_prio(
                sem->wait_list, WAIT_LIST_SIZE);
25          ready_list_insert(task_id);
26          call_schedule = 0;
27      }
28      if (call_schedule == 1)
29      {
30          schedule();
31      }
32      sem->counter--;
33      ENABLE_INTERRUPTS;
34  }
```

Uppgift 3: Semaforer(10p)

Betrakta följande program i Simple-OS:

```
#include <simple_os.h>
#include <stdio.h>
#define STACK_SIZE 5000

stack_item A_stack[STACK_SIZE];     // stack space for A
stack_item B_stack[STACK_SIZE];     // stack space for B

si_semaphore S;             // define semaphore

void A(void)                /* task A */
{
    si_wait_n_ms(500);      // sleep for 500 ms
    printf("E\n");
    si_sem_wait(&S);        // wait on semaphore
    printf("H\n");
    si_wait_n_ms(500);      // sleep for 500 ms
    printf("O\n");
    si_sem_signal(&S);      // signal on semaphore
    printf("S\n");
    si_wait_n_ms(1000);     // sleep for 1000 ms
    printf("L\n");
    si_sem_wait(&S);        // wait on semaphore
    printf("I\n");
    si_wait_n_ms(2000);     // sleep for 2000 ms
    printf("M\n");
    si_sem_signal(&S);      // signal on semaphore
    while(1);
}

void B(void)                /* task B */
{
    si_sem_wait(&S);        // wait on semaphore
    printf("S\n");
    si_wait_n_ms(1000);     // sleep for 1000 ms
    printf("M\n");
    do_work();              // do some work for some time
    printf("A\n");
    si_sem_signal(&S);      // signal on semaphore
    printf("P\n");
    si_sem_wait(&S);        // wait on semaphore
    printf("R\n");
    si_sem_signal(&S);      // signal on semaphore
    printf("E\n");
    si_wait_n_ms(500);      // sleep for 500 ms
    while(1);
}
```

```
void do_work()              /* do_work function */
{
    int i;
    volatile int work_data;
    for(i=0; i < 77000;i++){
        work_data=i;
    }
}


int main(void)              /* main program */
{
    /* initialise simple OS kernel */
    si_kernel_init();
    /* initialise semaphore to 1 */
    si_sem_init(&S, 1);
    /* create tasks */
    si_task_create(A, &A_stack[STACK_SIZE-1], 20); // low priority
    si_task_create(B, &B_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 från det att de båda processerna A och B 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. Ange också den resulterande utskriften.

Uppgift 4: Meddelandehantering(12p)

En realtidsapplikation som använder meddelandehantering i Simple OS ska implementeras. Applikationen ska utgöra en timer med alarm.

Följande funktioner finns redan:

```
int get_input(void)
// Returnerar ett eventuellt inmatat heltal, dock först efter att
// användaren tryckt på enter-tangenten. OBSERVERA att eftersom denna
// funktion stannar och väntar på inmatning så kommer endast
// den process som anropat funktionen kunna vara körande, till dess
// att funktionen returnerar förstås.

void show_time(int timer_value)
// Uppdaterar timerns display, och returnerar sedan

void start_alarm(void)
// Startar alarm-ljudet på timern, och returnerar sedan
// Alarm-ljudet fortsätter, oavsett vad som händer för övrigt, ända
// tills stop_alarm aktiverats

void stop_alarm(void)
// Stoppar alarm-ljudet på timern, och returnerar sedan
```

I Simple OS finns även färdiga funktioner för att skicka och ta emot meddelanden:

```
void si_message_send(const char message[], int length, int receive_pid)
// Skickar meddelande, av längden length tecken, till processen
// med id receive_pid.
// Anropande process väntar om meddelandebuffern är full.

void si_message_receive(char message[], int *length, int *send_pid)
// Tar emot meddelande, med längden *length tecken, från processen
// med id *send_id.
// Anropande process väntar om meddelandebuffern är tom.
```

Samt en lämplig funktion för att vänta en viss tid:

```
void si_wait_n_ms(int n_ms)
// Anropande process väntar n_ms millisekunder
```

Följande krav finns:

- Timern ska använda sig av sekundnoggrannhet (ungefär)
- Det ska finns fyra processer i lösningen vars huvudsakliga uppgift ska vara följande (dvs du får införa mer funktionalitet, men inte mindre, om du behöver):
  1. tick_task : hålla reda på när det gått en sekund
  2. timer_task : hålla reda på återstående timer-tid
  3. alarm_task : starta och stoppa alarm-ljudet
  4. user_task : ta emot inmatning från användaren

- Du får INTE deklarera fler nya funktioner utöver dom som redan finns ovan. Du får förstås använda dig av redan existerande funktioner från C-språket och Simple OS om du vill.
- Du kan räkna med att nödvändiga header-filer finns inkluderade samt att meddelandestackar för processerna finns deklarerade och initierade
- Du måste själv bestämma vilka och hur många meddelandetyper som behövs. Obs, glöm inte att visa detta.
- Du måste själv bestämma hur meddelandestrukturen ser ut. Obs, glöm inte att visa detta.
- Användaren ska kunna mata in en tid endast då timern står stilla. Så fort användaren matat in en tid ska timern börja räkna tiden.
- När den inmatade tiden har gått ska alarmet starta
- Användaren ska, genom att trycka på enter-tangenten, kunna bekräfta (stoppa) alarmet först när alarmet startat. Därefter ska användaren kunna mata in en ny tid.

Huvudprogrammet ser ut enligt följande:

```
int main(int argc, char **argv)
{
    /* initialise kernel */
    si_kernel_init();

    /* initialise message handling */
    si_message_init();

    /* create tasks and id's */
    si_task_create(tick_task, &tick_task_stack[STACK_SIZE-1], 16);
    int Tick_Task_Id = 1;
    si_task_create(timer_task, &timer_task_stack[STACK_SIZE-1], 14);
    int Timer_Task_Id = 2;
    si_task_create(alarm_task, &alarm_task_stack[STACK_SIZE-1], 12);
    int Alarm_Task_Id = 3;
    si_task_create(user_task, &user_task_stack[STACK_SIZE-1], 10);
    int User_Task_Id = 4;

    /* start the kernel */
    si_kernel_start();

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

Din uppgift är alltså, utöver deklaration av meddelandetyper och meddelandestruktur enligt kraven ovan, att skriva koden för de fyra processerna `tick_task`, `timer_task`, `alarm_task` och `user_task`.

Uppgift 5: Teori(10p)

(a) (3p) En kritisk region kan vara *relativt odelbar* eller *absolut odelbar*. Vad är skillnaden? Ge något exempel på var det finns en *absolut odelbar* kritisk region i ett realtidssystem.

(b) (2p) Beskriv två huvudsakliga skillnader mellan en *semafor* och en *mutex*. Beskrivningen får inte underförstå några egenskaper hos en *semafor* eller en *mutex*.

(c) (2p) Ge ett exempel på någon del i ett realtidssystem som måste skrivas i assembler, dvs som inte går att skriva i ett högnivåspråk som C, samt förklara varför den delen måste skrivas i assembler?

(d) (2p) Om prioritetsbaserad påtvingad schemaläggning används och det finns flera processer med samma högsta prioritet att växla till vid ett processbyte, vilka vanliga metoder skulle schemaläggaren kunna använda då? Beskriv två metoder.

(e) (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?

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:
$n/12 + 3/8 + 1/3 \leq 1$
$2n/24 + 9/24 + 8/24 \leq 24/24$
$2n \leq 7$
$n \leq 7/2$
Alltså, största värde på n 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 1 1 1 1 1 1 1 1 1 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
```

Alla processer klarar sina deadlines och den faktiska utnyttjandegraden blir 23/24 (ty n=3 ger Ue=23/24). Vid tidpunkten t=24 blir samtliga processer samtidigt redo att köra igen, dvs där börjar förloppet om.

(Kommentar: Lösningen behöver visa hela tidsförloppet (inklusive deadlines) fram till t=24, att körande process fortsätter (i förekommande fall), korrekt dra slutsatsen om vilka processer som klarar sina deadlines, att värdet på n=3, samt att den faktiska utnyttjandegraden blir 23/24)

1b

MED RMS får processerna prioriteter utefter hur ofta de ska köras, dvs ju kortare tidsintervall ju högre prioritet. Prioriteten bli alltså P3 > P2 > P1. 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
   1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
```

P1 missar sin deadline vid tidpunkten t=12. P2, P3 och P4 klarar sina deadlines. Den faktiska utnyttjandegraden blir 22/24 (intervallet t=[22,24] blir outnyttjat). Vid

tidpunkten t=24 blir samtliga processer samtidigt redo att köra igen, dvs där börjar förloppet om.

(Kommentar: Lösningen behöver visa hela tidsförloppet (inklusive deadlines) fram till t=24, tala om de inbördes prioriteterna för P1, P2 och P3 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 22/24.)

Lösningsförslag fråga 2

Signal för en svag semafor:
Om någon/några processer väntar ska den med högst prio flyttas till ready-listan. En svag semafor räknar alltid upp (via Signal) oavsett om någon annan väntar på den eller inte (Detta för att kunna ta semaforen på nytt, även om någon annan väntade (svält)). Dvs, för en svag semafor kommer semaforens värde alltid att vara större än 0 (noll) när semaforen lämnas över till en annan process. Dock ska schemaläggaren anropas om någon annan väntar (den som väntar kan ju ha högre prioritet).

Wait för en svag semafor:
Eftersom semaforens värde alltid är större än noll när den lämnas över till en annan process, så ska den alltid räknas ned i Wait. Dock måste man först kontrollera att semaforen är ledig (eftersom man kan ha blivit väntande på den i ett tidigare skede då den inte var ledig), och i annat fall vänta på den och anropa schemaläggaren.

Med dessa kunskaper är det möjligt att komma fram till följande programkod:

Antingen bara de korrigerade programraderna:

```
05      while (sem->counter == 0)

08      ready_list_remove(task_id);

09      wait_list_insert(
            sem->wait_list, WAIT_LIST_SIZE, task_id);

10      schedule();
11      }

12      sem->counter--;

20      int call_schedule = 0;

26      call_schedule = 1;

28      sem->counter++;
29      if (call_schedule == 1)
30      {
31          schedule();
32      }
```

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

```
00  /* Wait */
01  void si_sem_wait(si_semaphore *sem)
02  {
        /* task id */
03      int task_id;
        /* disable interrupts */
04      DISABLE_INTERRUPTS;
        /* as long as counter == 0 */
05      while (sem->counter == 0)
06      {
            /* get task_id of running task */
07          task_id = task_get_task_id_running();
            /* remove it from ready list */
08          ready_list_remove(task_id);
            /* insert it into the semaphore waiting list */
09          wait_list_insert(
                sem->wait_list, WAIT_LIST_SIZE, task_id);
            /* call schedule */
10          schedule();
11      }
        /* decrement */
12      sem->counter--;
        /* enable interrupts */
13      ENABLE_INTERRUPTS;
14  }
15
16  /* Signal */
17  void si_sem_signal(si_semaphore *sem)
18  {
        /* task id */
19      int task_id;
        /* clear flag for calling schedule */
20      int call_schedule = 0;
        /* disable interrupts */
21      DISABLE_INTERRUPTS;
        /* check if tasks are waiting */
22      if (!wait_list_is_empty(
        sem->wait_list, WAIT_LIST_SIZE))
23      {
            /* get task_id with highest priority */
24          task_id = wait_list_remove_highest_prio(
                sem->wait_list, WAIT_LIST_SIZE);
            /* make this task ready to run */
25          ready_list_insert(task_id);
            /* set flag for calling schedule */
26          call_schedule = 1;
27      }
        /* increment counter */
28      sem->counter++;
```

```
        /* check if schedule should be called */
29      if (call_schedule == 1)
30      {
            /* call schedule */
31          schedule();
32      }
        /* enable interrupts */
33      ENABLE_INTERRUPTS;
34  }
```

(Kommentar: Korrekta korrektioner ger 1p (7 fall), förutom flytt av rad som ger 0.5p (2 fall). Felaktiga korrektioner (dvs ändring av sådant som redan är korrekt) ger -1p. Om slutsumman har halva poäng avrundas den uppåt.)

Lösningsförslag fråga 3:

Programmet skriver ut: S E M A P H O R E S

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).

	Orsak		Verkan				
	A	B	körande	counter	T	W	R
1)	prio L	prio H	B	1			A,B
2)		wait	B	0			A,B
3)		sleep(1)	A	0	B		A
4)	sleep(0.5)		-	0	A,B		-
5)	wait		-	0	B	A	-
6)		signal	B	0			A,B
7)		wait	A	0		B	A
8)	sleep(0.5)		-	0	A	B	-
9)	signal		B	0			A,B
10)		signal	B	1			A, B
11)		sleep(0.5)	A	1	B		A
12)	sleep(1)		-	1	A,B		-
13		while(1)	B	1	A		B

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

2) B gör wait (W tom) counter−−, B fortsätter

3) B gör printf(S), B gör sleep(1) och läggs i T, A blir körande

4) A gör sleep(0.5) och läggs i T

5) A vaknar först (läggs i R), A gör printf(E), A gör wait (counter==0) och läggs i W

6) B vaknar och gör printf(M), do_work, printf(A), B gör signal (counter oförändrad ty A i W, men A till R och schedule anropas), och B med högst prio fortsätter och gör printf(P)

7) B gör wait (counter==0) och hamnar i W, (counter oförändrad ty counter==0), A blir körande och gör printf(H)

8) A gör sleep(0.5) och läggs i T, B ligger i W, dvs ingen körande

9) A vaknar gör printf(O) och signal, (counter oförändrad ty B i W, men B till R och schedule anropas), och B med högst prio fortsätter

10) B gör printf(R) och signal (counter++ ty W tom), B gör printf(E)

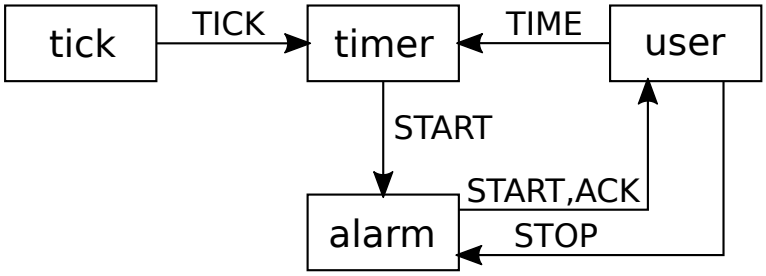
11) B gör sleep(0.5) och läggs i T, och A blir körande, A gör printf(S)

12) A gör sleep(1) och läggs i T

13) B vaknar och gör while(1) varefter inga fler processbyten sker, ty B högst prio

Lösningsförslag fråga 4:

Om man för sig själv börjar med att rita upp en figur över de aktuella processerna med meddelandetyper och kommunikationsvägar så blir jobbet sedan bara att översätta den figuren till kod.



Processen `tick_task` skickar TICK-meddelanden till `timer_task` en gång i sekunden.

Processen `user_task` börjar med inmatningsfunktionen `get_input`, vilket leder till att inga andra processer därefter blir körande förrän användaren tryckt på enter-tangenten och `get_input` returnerar. Då skickar `user_task` ett TIME-meddelande till `timer_task`, som därefter kan börja räkna tiden.

När det är dags att starta alarmet så skickar `timer_task` ett START-meddelande till `alarm_task`, som startar alarmet och först därefter informerar `user_task` (via ännu ett START-meddelande) att detta skett. Då kan `user_task` övergå till att bekräfta alarmet via inmatningsfunktionen `get_input` och skicka ett STOP-meddelande till `alarm_task` som stoppar alarmet och sedan bekräftar detta till `user_task` med ett ACK-meddelande. Då kan `user_task` återgå till att ta in ett nytt timer-värde via inmatningsfunktionen `get_input`.

```
/* message types */
#define TICK_MESSAGE 0
#define TIME_MESSAGE 1
#define START_ALARM_MESSAGE 2
#define STOP_ALARM_MESSAGE 3
#define ACK_MESSAGE 4

/* message data type */
typedef struct
```

```
{
  int type;
  int timer_value;
} message_data_type;

void tick_task(void)
{
  message_data_type message;
  while(1)
  {
    si_wait_n_ms(1000);
    message.type = TICK_MESSAGE;
    si_message_send((char*) &message, sizeof(message), Timer_Task_Id);
  }
}


void timer_task(void)
{
  message_data_type message;
  int length;
  int send_task_id;
  int timer_value = 0;
  while(1)
  {
    si_message_receive((char *) &message, &length, &send_task_id);
    switch(message.type)
    {
      case TICK_MESSAGE:
      timer_value--;
      show_timer(timer_value);
      if (timer_value == 0) {
        message.type = START_ALARM_MESSAGE;
        si_message_send((char *) &message, sizeof(message), Alarm_Task_Id);
      }
      break;
      case TIME_MESSAGE:
      timer_value = message.timer_value;
      show_timer(timer_value);
      break;
    }
  }
}


void alarm_task(void)
{
  message_data_type message;
  int length;
  int send_task_id;
```

```
  while(1)
  {
    si_message_receive((char *) &message, &length, &send_task_id);
    switch(message.type)
    {
      case START_ALARM_MESSAGE:
      start_alarm();
      message.type = START_ALARM_MESSAGE;
      si_message_send((char *) &message, sizeof(message), User_Task_Id);
      break;
      case STOP_ALARM_MESSAGE:
      stop_alarm();
      message.type = ACK_MESSAGE;
      si_message_send((char *) &message, sizeof(message), User_Task_Id);
      break;
    }
  }
}


void user_task(void)
{
  message_data_type message;
  int length;
  int send_task_id;
  while(1)
  {
    message.timer_value = get_input();
    message.type = TIME_MESSAGE;
    si_message_send((char *) &message, sizeof(message), Timer_Task_Id);
    si_message_receive((char*) &message, &length, &send_task_id);
    message.timer_value = get_input();
    message.type = STOP_ALARM_MESSAGE;
    si_message_send((char*) &message, sizeof(message), Alarm_Task_Id);
    si_message_receive((char*) &message, &length, &send_task_id);
  }
}
```

Lösningsförslag fråga 5

5a

En relativt odelbar kritisk region får avbrytas, men ingen annan process får använda gemensam resurs. En absolut odelbar kritisk region får inte ens avbrytas (avbrott stängs av). Detta förekommer i realtidssystemets kärnfunktioner, t ex Wait, Signal, Await, Cause.

5b

En semafor har ett värde ≥ 0, medan en mutex har värdet 0 (låst) eller 1 (olåst).
En semafor kan användas för asymmetrisk/symmetrisk synkronisering, men det går inte med en mutex då den måste låsas/olåsas inom samma tråd.

5c

De delar av realtidssystemet som växlar kontext vid ett processbyte måste göras i assembler då ett högnivåspråk som C saknar möjligheten att göra saker som att spara undan och återställa CPU:ns kontext på/från processens stack. Specifikt byte av stackpekare, återställande av CPU:ns registerinnehåll samt programräknare.

5d

Tidskvanta : Den process som varit körklar längst tid får köra (s k Round-Robin scheduling)
Köordning : Den process som ligger först i ready-listan får köra (FIFO: First in First Out)

5e

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