Tentamen
Datorteknik och realtidssystem, TSEA81

Datum	2021-01-11
Lokal	Distanstentamen
Tid	08-12
Kurskod	TSEA81
Provkod	TEN1
Kursnamn	Datorteknik och realtidssystem
Institution	ISY
Antal uppgifter	4
Antal sidor (inklusive denna sida)	18
Kursansvarig	Anders Nilsson
Telefon under skrivtiden	013-28 2635
Kursadministratör	Maria Hamnér, 013-28 5715
Tillåtna hjälpmedel	Inga, förutom dator för att skriva lösningar på digital form samt mobiltelefon/scanner för att digitalisera handskrivna lösningar
Betygsgränser	**Poäng** **Betyg** 41-50 5 31-40 4 21-30 3 0-20 U

Viktig information

- Alla svar ska ha en motivation om inget annat anges. Om du svarar med programkod räknas kommentarer i programkoden som motivation. 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*n, (i+1)*n]
- *P2* ska arbeta/köra under 1 tidsenhet i tidsintervallet [i*6, (i+1)*6]
- *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 $n \geq 1$.

Tidräkningen startar vid t=0 för alla processer. Du kan anta att uppstart av processer inte tar någon tid. Eventuellt missat arbete under något tidsintervall ackumuleras *inte* till kommande tidsintervall.

(a) (5p) Antag att schemaläggningsmetoden Earliest Deadline First (EDF) används. Antag att processbyte inte tar någon tid. 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. Antag att man totalt sett vill ha 10% outnyttjad CPU-tid (t ex för processbyte). Processen *P1* ska fortfarande köra så ofta som möjligt.
Vilket värde får då n?
Kommer processerna att uppfylla kraven?
Hur stor blir den faktiska utnyttjandegraden?
Visa vad som händer när programmet körs genom att rita ett tidsdiagram.

Uppgift 2: Teori(12p)

(a) (2p) Vad är syftet med en stark semafor, i jämförelse med en svag semafor, och hur uppnår den starka semaforen sitt syfte?

(b) (2p) Vad är ett deadlock i ett realtidssammanhang, och vad är en typisk orsak till att det uppstår?

(c) (2p) Vad är ett race condition i ett realtidssammanhang, och vad är en typisk orsak till att det uppstår?

(d) (1p) Varför ska ett villkor för en Await-operation alltid kontrolleras med `while`?

(e) (1p) Om det är dags att växla till en ny process, men ingen användarprocess är redo att köra, vad kan schemaläggaren tänkas göra då?

(f) (2p) Ange minst en fördel och minst en nackdel med att använda ett realtidsoperativsystem med processer, jämfört med att använda ett foreground/background-system (som alternativ till ett realtidsoperativsystem).

(g) (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?

Uppgift 3: Semaforer och villkorsvariabler(14p)

Betrakta följande program i Simple-OS.

```
si_semaphore S;          // define semaphore
si_condvar CV0;          // define condition variable 0
si_condvar CV1;          // define condition variable 1

int B0 = 0;              // common resourses
int B1 = 0;
int C = 0;

void p0(void)
{
    while(1)
    {
        si_sem_wait(&S);

        C = !C;          // toggle C

        si_cv_broadcast(&CV0);
        if (C == 1) {
            printf("%d%d\n", B1, B0);
        }
        si_sem_signal(&S);
    }
}

void p1(void)
{
    while(1)
    {
        si_sem_wait(&S);
        while (C == 0) {
            si_cv_wait(&CV0);
        }

        B0 = !B0;        // toggle B0

        si_cv_broadcast(&CV1);
        while (C == 1) {
            si_cv_wait(&CV0);
        }
        si_sem_signal(&S);
    }
}
```

Programkoden fortsätter på nästa sida

```
void p2(void)
{
    while (1)
    {
        si_sem_wait(&S);
        while(B0 == 0) {
            si_cv_wait(&CV1);
        }

        B1 = !B1;       // toggle B1

        while (B0 == 1) {
            si_cv_wait(&CV1);
        }
        si_sem_signal(&S);
    }
}
```

Antag att huvudprogrammet initierar semaforen `S` till 1, och associerar de båda händelsevariablerna `CV0` och `CV1` till semaforen `S`. Funktionerna `p0`, `p1` och `p2` utgör processerna `P0`, `P1` respektive `P2`. Antag att processen `P0` har lägst prioritet, processen `P1` har mellanprioritet och processen `P2` har högst prioritet, och att samtliga processer är körklara samtidigt. Antag att endast en process kan köra åt gången.

(a) (2p) Ange den resulterande utskriften från programmet, fram till det att den upprepar sig.

(b) (12p) Beskriv steg för steg vad som händer ifrån det att processerna `P0`, `P1` och `P2` är körklara. Var noggrann med att tala om vilken process som är körande, vilka listor processerna 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. Redovisa detta fram till det att process `P2` gör `si_sem_signal()` första gången. Redovisa gärna stegvis i tabellform där det framgår vad som är orsak och verkan. Ge en kommentar för varje steg.

Uppgift 4: Meddelandehantering och prioritetsinversion(14p)

Programkoden nedan avser att implementera meddelandehantering mellan tre processer.
Processen `reg_proc` är en högprioriterad samplingsprocess vars syfte är att ta 10 sampel, därefter skriva dessa sampel (ett i taget) till en buffer-process `buf_proc` som sköter en gemensam buffer. Därefter väntar `reg_proc` en slumpartad tid innan den börjar om att sampla på nytt.
Processen `log_proc` är en lågprioriterad process som ska läsa, dvs begära, 10 sampel (ett i taget) från buffer-processen och därefter utföra en fft-transform på dessa 10 sampel, innan den börjar om för att göra samma sak igen.
Processen `buf_proc` sköter en gemensam buffer, och måste hantera meddelanden till och från de andra processerna för att åstadkomma detta korrekt.

Konsistensen mellan dessa 10 sampeldata måste bevaras.

Det kan även antas finnas andra processer i systemet, med olika prioriteter.

Ett problem i sammanhanget är att man vill undvika prioritetsinversion, dvs andra processer kan tänkas hindra `log_proc` från att köra, som i sin tur då hindrar `reg_proc` från att skriva till bufferten. Detta vill man alltså undvika, på konventionellt sätt, men bara då det är nödvändigt. Dvs, prioriteter ska inte förändras i onödan.

Det finns ett antal globala definitioner, enligt följande:

```
#define LPRIO 10      // low priority
#define HPRIO 20      // high priority

#define PORT_REG 1    // port for reg_proc
#define PORT_LOG 2    // port for log_proc
#define PORT_BUF 3    // port for buf_proc


typedef enum {
    WR_LOCK,          // message to lock buffer for writing
    ACCEPT,           // message to accept request
    WRITE,            // message to write data to buffer
    UNLOCK            // message to unlock buffer
} message_type;


struct message {
    message_type type;
    int data;
};
```

Programkoden fortsätter på nästa sida

Processen `reg_proc` är färdig, enligt följande:

```
void reg_proc(void)
{
    struct message *m;
    char buf[4096];
    m = (struct message *) buf;

    int samples[10];
    int i;

    set_prio(HPRIO);

    while (1)
    {
        for (i=0;i<10;i++)
        {
            samples[i] = sample();
        }

        m->type = WR_LOCK;
        message_send((char*) m, sizeof(*m), PORT_BUF, 0);

        do {
            message_receive(buf, 4096, PORT_REG);
            m = (struct message *) buf;
        } while (m->type != ACCEPT);

        for (i=0;i<10;i++)
        {
            m->type = WRITE;
            m->data = samples[i];
            message_send((char*) m, sizeof(*m), PORT_BUF, 0);
        }

        m->type = UNLOCK;
        message_send((char*) m, sizeof(*m), PORT_BUF, 0);

        sleep_random();
    }
}
```

Programkoden fortsätter på nästa sida

Det finns ett antal färdiga funktioner, enligt:

```
// sample : take one sample
int sample(void);

// fft : calculate fft on given array
void fft(int *array);

// write_buf : write data to buffer
void write_buf(int data);

// read_buf : read data from buffer
int read_buf(void);

// set_prio : set priority of the calling process,
// higher prio value yields higher priority
void set_prio(int prio);

// sleep_random : sleep a random amount of time
void sleep_random(void);

// message_send : send message to a certain queue.
void message_send(char *msg, int length, int queueid, int priority);

// message_receive : receive message from a certain queue.
int message_receive(char *msg, int length, int queueid);
```

Huvudprogrammet startar upp processerna, enligt:

```
int main(void)
{
    int pid;

    message_init();     // initialize messaging system

    pid = fork();
    if (!pid) {reg_proc();}

    pid = fork();
    if (!pid) {log_proc();}

    buf_proc();

    while(1);
}
```

Uppgiften fortsätter på nästa sida

Din uppgift är att skriva programkoden för log-processen `log_proc` och buffer-processen `buf_proc`, så att dessa uppfyller beskrivningarna ovan. Du kan lägga till flera meddelandetyper till definitionen av `message_type` om du vill, flera fält till structen `message` om du vill (dock inte ha mer än ett sampeldata per meddelande) och definiera flera funktioner om du vill. Log-processen `log_proc` ska när den startar vara en lågprioriterad process, med prioritet `LPRIO`.

Du behöver inte bekymra dig om själva buffern, utan kan betrakta den som att det alltid går att skriva data till buffern (den blir aldrig full) och det går alltid att läsa data från buffern (den blir aldrig tom). Uppgiften handlar istället om hur man strukturerar programkod med meddelandehantering och undviker prioritetsinversion.

Lägg ingen vikt vid ordningen för hur processerna startar i huvudprogrammet, utan betrakta det hela som att alla processer är redo att köra direkt, dvs redo att ta emot och skicka meddelanden.

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:
$4/n + 1/6 + 1/3 \leq 1$
$4/n + 3/18 + 6/18 \leq 18/18$
$4/n \leq 9/18$
$8 \leq n$
Alltså, minsta värde på n är 8.

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
```

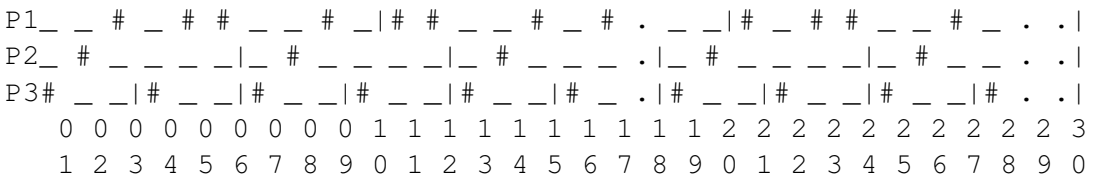
Alla processer klarar sina deadlines och den faktiska utnyttjandegraden blir 1 (ty n=8 ger Ue=1). Vid tidpunkten t=24 blir samtliga processer samtidigt redo att köra igen, dvs där börjar förloppet om.

(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=8, samt att den faktiska utnyttjandegraden blir 1)

1b

För att få 10% till processbyte gäller att utnyttjandegraden $U_e \leq 9/10$ (antaget att processerna klarar kraven), enligt:
$4/n + 1/6 + 1/3 \leq 9/10$
$4/n + 5/30 + 10/30 \leq 27/30$
$4/n \leq 12/30$
$10 \leq n$
Alltså, minsta värde på n är 10.

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.



Alla processer klarar sina deadlines. Den faktiska utnyttjandegraden blir 27/30. Vid tidpunkten t=30 blir samtliga processer samtidigt redo att köra igen, dvs där börjar förloppet om.

(Kommentar: Lösningen behöver visa hela tidsförloppet (inklusive deadlines) fram till t=30, att värdet på n=10, 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 27/30.)

Lösningsförslag fråga 2

2a

Syftet är att förhindra svält för en lågprioriterad process (dvs att den får tillgång till gemensam resurs någon gång), och det uppnås genom att neka en högprioriterad process återkommande tillgång till semaforen/resursen då en lågprioriterad process begärt tillgång under tiden som den högprioriterade processen har tillgång till semaforen.

2b

Deadlock är ett cykliskt beroende mellan processer som hindrar dem att köra vidare, typiskt på grund av att gemensamma resurser hålls av respektive process och för att frigöra den egna resursen behöver den ena processen tillgång till den andra processens resurs, och vice versa.

2c

Race condition beror på ordningsföljden mellan processer, där en viss ordning gör att systemet fungerar och en annan ordning (t ex samtidigt) gör att systemet inte fungerar, typiskt på grund av avsaknad av synkronisering mellan processerna.

2d

När en process på nytt blir körande (efter en Await-operation), så måste villkoret kontrolleras på nytt för att se om man ska gå vidare eller vänta igen.

2e

Vanligtvis finns en s k idle-task som alltid är redo att köra, som inte gör något annat än att försätta processorn i low power mode.

2f

En fördel med ett RTOS är att med processer blir programmet med modulärt och lättare att underhålla. En annan fördel är att processorn kan utnyttjas mer effektivt då ett RTOS typiskt växlar till en annan process när en process blir väntande.
En nackdel med ett RTOS är att risken för deadlock ökar. En annan nackdel är att systemet blir svårare att analysera och felsöka.

2g

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.

Lösningsförslag fråga 3:

3a

Programmet skriver ut:
00, 11, 10, 01 (om och om igen)

3b

I följande tabell gäller att S är semaforens värde, R är Ready-listan, WS är väntelistan för semaforen S, WCV0 och WCV1 är väntelistor för händelsevariabelerna CV0 resp CV1.

	Orsak	Verkan						
		Kör	S	R	WS	WCV0	WCV1	Kommentar
1)	Init	-	1	P0,P1,P2				C:B1:B0 = 000
2)	Sched	P2	1	P0,P1,P2				
3)	P2:Wait	P2	0	P0,P1,P2				
4)	P2:Await	P2	1	P0,P1			P2	
5)	Sched	P1	1	P0,P1			P2	
6)	P1:Wait	P1	0	P0,P1			P2	
7)	P1:Await	P1	1	P0		P1	P2	
8)	Sched	P0	1	P0		P1	P2	
9)	P0:Wait	P0	0	P0		P1	P2	C:B1:B0 = 100
10)	P0:Cause	P0	0	P0	P1		P2	Utskrift:00
11)	P0:Signal	P0	0	P0,P1			P2	
12)	Sched	P1	0	P0,P1			P2	C:B1:B0 = 101
13)	P1:Cause	P1	0	P0,P1	P2			
14)	P1:Await	P1	0	P0,P2		P1		
15)	Sched	P2	0	P0,P2		P1		C:B1:B0 = 111
16)	P2:Await	P2	1	P0		P1	P2	
17)	Sched	P0	1	P0		P1	P2	
18)	P0:Wait	P0	0	P0		P1	P2	C:B1:B0 = 011
19)	P0:Cause	P0	0	P0	P1		P2	
20)	P0:Signal	P0	0	P0,P1			P2	
21)	Sched	P1	0	P0,P1			P2	
22)	P1:Signal	P1	1	P0,P1			P2	
23)	P1:Wait	P1	0	P0,P1			P2	
24)	P1:Await	P1	1	P0		P1	P2	
25)	Sched	P0	1	P0		P1	P2	
26)	P0:Wait	P0	0	P0		P1	P2	C:B1:B0 = 111
27)	P0:Cause	P0	0	P0	P1		P2	Utskrift:11
28)	P0:Signal	P0	0	P0,P1			P2	
29)	Sched	P1	0	P0,P1			P2	C:B1:B0 = 101
30)	P1:Cause	P1	0	P0,P1	P2			
31)	P1:Await	P1	0	P0,P2		P1		
32)	Sched	P2	0	P0,P2		P1		
33)	P2:Signal	P2	1	P0,P2		P1		

1) P0,P1 och P2 blir körklara, S initieras till 1.

2) Schemaläggaren startar P2 (högst prio i R)

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

4) P2 gör Await (ty B0==0), ingen väntar på Sem så S++, P2 till WCV1

5) Schemaläggaren väljer P1 (högst prio i R)

6) P1 gör Wait, ingen väntar på Sem så S–

7) P1 gör Await (ty C==0), ingen väntar på Sem så S++, P1 till WCV0

8) Schemaläggaren väljer P0 (högst prio i R)

9) P0 gör Wait, ingen väntar på Sem så S–

10) P0 gör Cause på CV0, så P1 till WS, print ”00”

11) P0 gör Signal, P1 (högst prio i WS) till R, och S oförändrad

12) Schemaläggaren väljer P1 (högst prio i R), B0=1

13) P1 gör Cause på CV1, så P2 till WS

14) P1 gör Await, ty (C==1), P2 (högst prio i WS) till R, S oförändrad, P1 till WCV0

15) Schemaläggaren väljer P2 (högst prio i R), B1=1

16) P2 gör Await, ty (B0==1), ingen i WS så S++, P2 till WCV1

17) Schemaläggaren väljer P0 (högst prio i R)

18) P0 gör Wait, ingen i WS så S–, C=0

19) P0 gör Cause på CV0, så P1 till WS

20) P0 gör Signal, P1 (högst prio i WS) till R, och S oförändrad

21) Schemaläggaren väljer P1 (högst prio i R)

22) P1 gör Signal, ingen i WS så S++

23) P1 gör Wait, ingen i WS så S–

24) P1 gör Await, ty (C==0), ingen i WS så S++, P1 till WCV0

25) Schemaläggaren väljer P0 (högst prio i R)

26) P0 gör Wait, ingen i WS så S–, C=1

27) P0 gör Cause på CV0, så P1 till WS, print ”11”

28) P0 gör Signal, P1 (högst prio i WS) till R, och S oförändrad

29) Schemaläggaren väljer P1 (högst prio i R), B1=0

30) P1 gör Cause på CV1, så P2 till WS

31) P1 gör Await, ty (C==1), P2 ()högst prio i WS) till R, S oförändrad, P1 till WCV0

32) Schemaläggaren väljer P2 (högst prio i R)

33) P2 gör Signal, ingen i WS så S++

Det krävs även att värdet på semaforen S är korrekt, att listor för Ready-list, väntelista för Semafor och väntelista för händelsevariabel CV0 och CV1 finns och är korrekta. 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.)

Lösningsförslag fråga 4:

Svårigheten i uppgiften ligger nog främst i processen `buf_proc` som måste kunna hantera meddelanden från både `reg_proc` och `log_proc`, komma ihåg vilken av dom som för tillfället har access till bufferten och dessutom ge meddelande till `log_proc` när det är dags att höja prioriteten för att utföra ”prioritetsärvning” i syfte att motverka prioritetinversion, och det bör då ske när `reg_proc` vill ha access till bufferten under tiden som `log_proc` har accessen.

Processen `log_proc` blir strukturellt väldigt lik `reg_proc`, med den skillnaden att `log_proc` måste kunna hantera ett meddelande om att höja prioriteten, samt att återställa prioriteten när alla 10 data har lästs.

Processen `log_proc` skulle kunna lösas enligt följande:

```
void log_proc(void)
{
    struct message *m;
    char buf[4096];
    m = (struct message *) buf;

    int samples[10];
    int i;

    set_prio(LPRIO);

    while(1)
    {
        m->type = RD_LOCK;
        message_send((char*) m, sizeof(*m), PORT_BUF, 0);

        do {
            message_receive(buf, 4096, PORT_LOG);
            m = (struct message *) buf;
        } while (m->type != ACCEPT);


        i=0;
        while(i<10)
        {
            m->type = READ;
            message_send((char*) m, sizeof(*m), PORT_BUF, 0);

            message_receive(buf, 4096, PORT_LOG);
            m = (struct message *) buf;

            if (m->type == PRIO)
            {
                set_prio(HPRIO);
            }
            if (m->type == DATA)
            {
                samples[i] = m->data;
                i++;
            }
        }

        fft(samples);

        m->type = UNLOCK;
        message_send((char*) m, sizeof(*m), PORT_BUF, 0);

        set_prio(LPRIO);
    }
}
```

Processen `buf_proc` skulle kunna lösas enligt följande:

```
void buf_proc()
{
    struct message *m;
    char buf[4096];
    m = (struct message *) buf;


    typedef enum {
        WRITE_MODE,
        READ_MODE,
        UNLOCKED
    } bufmode;

    bufmode mode = UNLOCKED;

    typedef enum {
        WRITE_REQ,
        READ_REQ,
        NONE
    } reqmode;

    reqmode req = NONE;

    while(1)
    {
        message_receive(buf, 4096, PORT_BUF);
        m = (struct message *) buf;

        switch (m->type) {
            case WR_LOCK:
            if (mode == UNLOCKED)
            {
                mode = WRITE_MODE;
                m->type = ACCEPT;
                message_send((char*) m, sizeof(*m), PORT_REG, 0);
            }
            if (mode == READ_MODE)
            {
                req = WRITE_REQ;
                m->type = PRIO;
                message_send((char*) m, sizeof(*m), PORT_LOG, 0);
            }
            break;

            case RD_LOCK:
            if (mode == UNLOCKED)
            {
                mode = READ_MODE;
```

```
                m->type = ACCEPT;
                message_send((char*) m, sizeof(*m), PORT_LOG, 0);
            }
            if (mode == WRITE_MODE)
            {
                req = READ_REQ;
            }
            break;

            case WRITE:
            if (mode == WRITE_MODE)
            {
                write_buf(m->data);
            }
            break;

            case READ:
            if (mode == READ_MODE)
            {
                m->type = DATA;
                m->data = read_buf();
                message_send((char*) m, sizeof(*m), PORT_LOG, 0);
            }
            break;

            case UNLOCK:
            mode = UNLOCKED;
            if (req == READ_REQ)
            {
                m->type = ACCEPT;
                message_send((char*) m, sizeof(*m), PORT_LOG, 0);
            }
            if (req == WRITE_REQ)
            {
                m->type = ACCEPT;
                message_send((char*) m, sizeof(*m), PORT_REG, 0);
            }
            break;

            default:
            break;
        }

    }
}
```

Fyra nya meddelanden har lagts till `message_type`:

```
typedef enum {
    WR_LOCK,          // message to lock buffer for writing
```

```
    ACCEPT,          // message to accept request
    WRITE,           // message to write data to buffer
    UNLOCK,          // message to unlock buffer
    RD_LOCK,         // message to lock buffer for reading
    READ,            // message to read data from buffer
    PRIO,            // message to raise prio in log_proc
    DATA             // message to send sampel data to log_proc
} message_type;
```

*(Kommentar: Lösningar som har ett "enkelriktat flöde" där buf_proc tar emot sampeldata från reg_proc för att sedan omedelbart skicka dessa vidare till log_proc, och där log_proc inte **begär** att få läsa från buf_proc, förkastas. I en sådan lösning bestämmer buf_proc när det är dags för log_proc att läsa sampeldata och buf_proc agerar inte buffer utan tvingar då log_proc att istället buffra data i inkommande meddelandekö. Det skulle kunna leda till meddelandekön till log_proc blir full, varpå buf_proc kan bli väntande, varpå reg_proc kan bli väntande.*
Övriga korrektioner som behövs för att koden ska fungera leder till -2p (eller möjligen -1p) för varje problem, beroende på problemets storlek.