# Tentamen
# Datorteknik och realtidssystem, TSEA81

| | |
|---|---|
| *Datum* | 2020-08-21 |
| *Lokal* | Distanstentamen |
| *Tid* | 14-18 |
| *Kurskod* | TSEA81 |
| *Provkod* | TEN1 |
| *Kursnamn* | Datorteknik och realtidssystem |
| *Institution* | ISY |
| *Antal uppgifter* | 4 |
| *Antal sidor (inklusive denna sida)* | 19 |
| *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. |
| *Betygsgränser* | **Poäng** **Betyg**<br>41-50 5<br>31-40 4<br>21-30 3<br>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 $n$ tidsenheter i tidsintervallet [$i$*12, ($i$+1)*12]
- *P2* ska arbeta/köra under 2 tidsenheter i tidsintervallet [$i$*8, ($i$+1)*8]
- *P3* ska arbeta/köra under 1 tidsenhet i tidsintervallet [$i$*4, ($i$+1)*4]
- *P4* 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 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: Teori(13p)

(a) (3p) Vilka är de centrala delarna i ett realtidssystem, och vad används dom till?

(b) (2p) Ge två exempel på hur processer (dvs inte trådar) kan kommunicera.

(c) (2p) Peterson's algoritm är en enkel implementation av ömsesidig uteslutning (mutual exclusion). Nämn två huvudsakliga nackdelar med Peterson's algoritm, jämfört med konventionella metoder för ömsesidig uteslutning i ett RTOS.

(d) (3p) Var lagrar CPU:n sin kontext, respektive var lagrar en process sin kontext?

(e) (3p) Ge exempel på tre situationer som garanterat leder till ett processbyte.

## Uppgift 3: Semaforer och villkorsvariabler(15p)

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 w1 = 5;
int w2 = 2;
int w3 = 1;

int r1 = 12;
int r2 = 6;
int r3 = 4;


int a(void)
{
  int v = 0;
  if (w3 > 0) {
    v = 3;
  }
  else if (w2 > 0) {
    v = 2;
  }
  else if (w1 > 0) {
    v = 1;
  }

  return v;
}
```

*Programkoden fortsätter på nästa sida*

```
void b(void)
{
  r1--;
  if (r1 == 0) {
    r1 = 12; w1 = 5;
  }
  r2--;
  if (r2 == 0) {
    r2 = 6; w2 = 2;
  }
  r3--;
  if (r3 == 0) {
    r3 = 4; w3 = 1;
  }
while ((r1 == 12) && (r2 == 6) && (r3 == 4));
}


void p1(void)
{
  while(1) {
    si_sem_wait(&S);
    while (a() != 1) {
      si_cv_wait(&CV);
    }
    printf("p1:%d\n", w1);
    w1--;
    b();
    si_cv_broadcast(&CV);
    si_sem_signal(&S);
  }
}

void p2(void)
{
  while(1) {
    si_sem_wait(&S);
    while (a() != 2) {
      si_cv_wait(&CV);
    }
    printf("p2:%d\n", w2);
    w2--;
    b();
    si_cv_broadcast(&CV);
    si_sem_signal(&S);
  }
}
```

*Programkoden fortsätter på nästa sida*

```
void p3(void)
{
  while(1) {
    si_sem_wait(&S);
    while (a() != 3) {
      si_cv_wait(&CV);
    }
    printf("p3:%d\n", w3);
    w3--;
    b();
    si_cv_broadcast(&CV);
    si_sem_signal(&S);
  }
}

/* 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], 10); // high priority
  si_task_create(p2, &p2_stack[STACK_SIZE-1], 15); // middle priority
  si_task_create(p3, &p3_stack[STACK_SIZE-1], 20); // low priority


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

  return 0;
}
```

(a) (2p) Ange den resulterande utskriften från programmet.

(b) (12p) 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 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 `P1` 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.

(c) (1p) Vad skulle man kunna säga att programmet simulerar?

## Uppgift 4: Korsa vatten(12p)

Någonstans längs Stångån finns en båt som används av Linux- respektive Windows-programmerare för att ta sig över ån. Båten rymmer 4 passagerare, och båten måste vara full för att kunna korsa Stångån. För att säkert ta sig över ån får dock inte passagerarna blandas så att det finns 3 Linux-programmerare och 1 Windows-programmerare eller tvärtom. Bara Linux- eller bara Windows-programmerare eller 2 av varje går dock bra.

Betrakta följande delar av programkod. Funktionen `board_boat` används för att gå ombord på båten. Funktionen `row_boat` används för att ro båten, vid lämpligt tillfälle.

```
01 pthread_mutex_t mutex;
02 int board_Linux = 0;
03 int board_Windows = 0;

04 void board_boat(char C)
05 {
06   pthread_mutex_lock(&mutex);
07   switch(C)
08     {
09     case 'L':
10       board_Linux++;
11       break;
12     case 'W':
13       board_Windows++;
14       break;
15     default:
16       printf("Wrong kind\n");
17     }
18   pthread_mutex_unlock(&mutex);
19 }

20 void row_boat(void)
21 {
22   pthread_mutex_lock(&mutex);
23   if ((board_Linux == 1) ||
24       (board_Linux == 3) ||
25       (board_Windows == 1) ||
26       (board_Windows == 3) ||
27       (board_Linux + board_Windows != 4))
28     {
29      printf("Illegal mix\n");
30     }
31   else
32     {
33       printf("Rowing\n");
34       board_Linux = 0;
35       board_Windows = 0;
36     }
37  pthread_mutex_unlock(&mutex);
```

```
38 }
```

Det finns också en färdig barriär-funktion `barrier`, som man kan använda om man vill.

```
39 pthread_mutex_t BM;  // barrier mutex
40 pthread_cond_t BV;   // barrier condition variable
41 int num = 0;         // number of processes in barrier

42 void barrier(int N)
43 {
44   pthread_mutex_lock(&BM);
45   while (state == EXITING)
46     {
47       pthread_cond_wait(&BV, &BM);
48     }
49   num++;
50   pthread_cond_broadcast(&BV);
51   while (state == ENTERING)
52     {
53       pthread_cond_wait(&BV, &BM);
54       if (num == N)
55         {
56           state = EXITING;
57           pthread_cond_broadcast(&BV);
58         }
59     }
60   num--;
61   if (num == 0)
62     {
63       state = ENTERING;
64     }
65   pthread_cond_broadcast(&BV);
66   pthread_mutex_unlock(&BM);
67 }
```

*Programkoden fortsätter på nästa sida*

Huvudprogrammet `main` skapar hela tiden nya passagerare, med sannolikheten 1/2 för Linux-programmerare och 1/2 för Windows-programmerare.

```
68 int main(void)
69   {
70   pthread_mutex_init(&BM, NULL);     // init barrier mutex
71   pthread_mutex_init(&mutex, NULL);      // init boat mutex

72   srand(getpid()); // set random seed

73   while (1)
74     {
75       if ((rand() % 10) >= 5)
76         {
77           pthread_t thread_handle;
78           pthread_create(&thread_handle, NULL, Linux_thread, 0);
79           pthread_detach(thread_handle);
80         }
81       else
82         {
83           pthread_t thread_handle;
84           pthread_create(&thread_handle, NULL, Windows_thread, 0);
85           pthread_detach(thread_handle);
86         }
87     }
88   return 0;
89 }
```

*Programkoden fortsätter på nästa sida*

Varje passagerare (Linux- eller Windows-programmerare) simuleras av var sin programtråd, `Linux_thread` respektive `Windows_thread`.

```
void *Linux_thread(void *arg)
{
...
board_boat('L');
...
}

void *Windows_thread(void *arg)
{
...
board_boat('W');
...
}
```

Din uppgift är att fullfärdiga programkoden för programtrådarna `Linux_thread` samt `Windows_thread`, så att båten alltid korsar Stångån på ett säkert sätt och att programmet aldrig skriver ut något annat än `Rowing`.

Du behöver inte ta hänsyn till vilken sida om ån som båten befinner sig på, utan kan resonera som att det finns tillgängliga passagerare på bägge sidor och att det bara handlar om att göra säkra resor med båten.

Huvudprogrammet `main` producerar/skapar ständigt nya passagerare/programtråder, så de passagerare/programtrådar som gjort en resa via `row_boat` ska konsumeras/upphöra någon gång efter att funktionen `row_boat` gjort sitt.

Du får inte ta bort redan existerande programkod, men däremot lägga till. Ange i så fall mellan vilka rader den nya koden ska placeras och vad som ska stå där.

Någon gång ska `row_boat` anropas.

Se till att deklarera och initiera egna variabler och funktioner.

# 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:
$n/12 + 2/8 + 1/4 + 1/3 \leq 1$
$2n/24 + 6/24 + 6/24 + 8/24 \leq 1$
$2n \leq 4$
$n \leq 2$
Alltså, största värde på n är 2.

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

```
P1_ _ _ _ _ _ _ # # _ _ _|_ _ _ _ _ # _ # _ _ _ _|
P2_ _ # _ # _ _ _|_ _ _ # _ # _ _|_ _ _ _ # # _ _|
P3_ # _ _|_ # _ _|_ # _ _|_ _ # _|# _ _ _|_ _ # _|
P4# _ _|# _ _|# _ _|_ # _|# _ _|# _ _|# _ _|_ _ #|
   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=2 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=2, samt att den faktiska utnyttjandegraden blir 1)*

## 1b

MED RMS får processerna prioriteter utefter hur ofta de ska köras, dvs ju kortare tidsintervall ju högre prioritet. Prioriteten bli alltså P4 > P3 > P2 > P1. Därefter används schemaläggningsmetoden priority based preemptive scheduling.

```
P1_ _ _ _ _ _ _ # _ _ _ _/_ _ # _ _ _ _ _ _ _ # .|
P2_ _ # _ _ # _ _|_ _ # # _ _ _ _|_ # _ # _ _ _ .|
P3_ # _ _|# _ _ _|# _ _ _|_ # _ _|# _ _ _|# _ _ .|
P4# _ _|# _ _|# _ _|# _ _|# _ _|# _ _|# _ _|# _ .|
   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 23/24 (intervallet t=[23,24] 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, P3 och P4 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 23/24.)*

## Lösningsförslag fråga 2

### 2a

Avbrott : för att kunna styra programflödet genom att avbryta en process och återuppta en annan.
Processorregister : för att hålla kontexten (status) för den programkod som för tillfället exekverar.
Stack : för att kunna spara/återhämta kontexten vid processbyte, även för att lagra temporärdata, för att lagra returadresser, parameteröverföring till funktioner

### 2b

Via meddelandehantering (POSIX message queues), via filer, via signaler (t ex CTRL-C sickar SIGINT-meddelande), via sockets, via piper

### 2c

Peterson's algoritm är komplex att implementera för många processer (fler än två). Peterson's algoritm bygger på vanlig programmering. Dvs, man måste t ex hantera problem med att kompilatorn inte vet att minnespositioner kan ändras av andra processer (m h a volatile), och problem med att processorn kan orsaka OoOE (Out of Order Execution).

### 2d

En CPU lagrar sin kontext i CPU:ns register, CPU:ns PC och CPU:ns stackpekare.
En process lagrar sin kontext i en egen stack och i ett processkontrollblock.

### 2e

En ny process med högst prioritet skapas
Körande process anropar Wait (på en semafor), och blir väntande
Körande process gör sleep
En process har väntat färdigt på en timer och har högre prioritet än körande process
En process gör signal/post på en semafor som en annan process med högre prio väntar på

# Lösningsförslag fråga 3:

## 3a

Programmet skriver ut:
p3:1 p2:2 p2:1 p1:5 P3:1 p1:4 p2:2 p2:1 p3:1 p1:3 p1:2 p1:1

## 3b

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

| | Orsak | Verkan | | | | | |
|---|---|---|---|---|---|---|---|
| | | Kör | S | R | WS | WCV | Kommentar |
| 1) | Init | - | 1 | P1,P2,P3 | | | |
| 2) | Sched | P1 | 1 | P1,P2,P3 | | | |
| 3) | P1:Wait | P1 | 0 | P1,P2,P3 | | | |
| 4) | P1:Await | P1 | 1 | P2,P3 | | P1 | |
| 5) | Sched | P2 | 1 | P2,P3 | | P1 | |
| 6) | P2:Wait | P2 | 0 | P2,P3 | | P1 | |
| 7) | P2:Await | P2 | 1 | P3 | | P1,P2 | |
| 8) | Sched | P3 | 1 | P3 | | P1,P2 | |
| 9) | P3:Wait | P3 | 0 | P3 | | P1,P2 | |
| 10) | P3:Cause | P3 | 0 | P3 | P1,P2 | | P3:1 |
| 11) | P3:Signal | P3 | 0 | P1,P3 | P2 | | |
| 12) | Sched | P1 | 0 | P1,P3 | P2 | | |
| 13) | P1:Await | P1 | 0 | P2,P3 | | P1 | |
| 14) | Sched | P2 | 0 | P2,P3 | | P1 | |
| 15) | P2:Cause | P2 | 0 | P2,P3 | P1 | | P2:2 |
| 16) | P2:Signal | P2 | 0 | P1,P2,P3 | | | |
| 17) | Sched | P1 | 0 | P1,P2,P3 | | | |
| 18) | P1:Await | P1 | 1 | P2,P3 | | P1 | |
| 19) | Sched | P2 | 1 | P2,P3 | | P1 | |
| 20) | P2:Wait | P2 | 0 | P2,P3 | | P1 | |
| 21) | P2:Cause | P2 | 0 | P2,P3 | P1 | | P2:1 |
| 22) | P2:Signal | P2 | 0 | P1,P2,P3 | | | |
| 23) | Sched | P1 | 0 | P1,P2,P3 | | | |
| 24) | P1:Cause | P1 | 0 | P1,P2,P3 | | | P1:5 |
| 25) | P1:Signal | P1 | 1 | P1,P2,P3 | | | |

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

2) Schemaläggaren startar P1 (högst prio i R)

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

4) P1 gör Await (ty a()!=1), ingen väntar på Sem så S++, P1 till WCV

5) Schemaläggaren väljer P2 (högst prio i R)

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

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

8) Schemaläggaren väljer P3 (högst prio i R)

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

10) P3 gör Cause, så P1 och P2 till WS, print P3:1”

11) P3 gör Signal, P1 (högst prio i WS) till R, och S oförändrad

12) Schemaläggaren väljer P1 (högst prio i R)

13) P1 gör Await (ty a()!=1), P2 (högst prio i WS) till R, och S oförändrad, P1 till WCV

14) Schemaläggaren väljer P2 (högst prio i R)

15) P2 gör Cause, så P1 till WS, print P2:2”

16) P2 gör Signal, P1 (högst prio i WS) till R, och S oförändrad

17) Schemaläggaren väljer P1 (högst prio i R)

18) P1 gör Await (ty a()!=1), ingen i WS så S++, P1 till WCV

19) Schemaläggaren väljer P2 (högst prio i R)

20) P2 gör Wait, ingen i WS så S–

21) P2 gör Cause, så P1 till WS, print P2:1”

22) P2 gör Signal, P1 (högst prio i WS) till R, och S oförändrad

23) Schemaläggaren väljer P1 (högst prio i R)

24) P1 gör Cause, ingen i WCV, print P1:5”

25) P1 gör Signal, ingen i WS så S++

*(Kommentar: 0.5 poäng ges för varje korrekt utfört steg enligt ovan.*
*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.*
*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.)*

### 3c

Programmet simulerar en schemaläggning av processerna `P1`, `P2` och `P3`, med prioritetsbaserad påtvingad schemaläggning där prioriteterna är `P3` > `P2` > `P1`.

## Lösningsförslag fråga 4:

Följande lösning använder sig av tre semaforer för att synkronisera de båda processerna `Linux_thread` och `Windows_thread`.

Semaforen `M` fungerar som en mutex, men används här istället för en vanlig mutex för att kunna låsa denna ”mutex” i ena tråden och låsa upp den i den andra.
Semaforen `LPQ` fungerar som en kö för att hålla reda på antalet Linux-passagerare som får borda båten.
Semaforen `LPQ` fungerar som en kö för att hålla reda på antalet Windows-passagerare som får borda båten.

När den sista passageraren (Linux eller Windows) ska borda båten blir denna passagerare kapten (`isCaptain`) och signalerar då att det är dags att ro över båten när alla passagerare kommit ombord. Detta vet man först när samtliga trådar kommit in i barriären, så där kan `row_boat` anropas (och `isCaptain` nollställas).

```
01 pthread_mutex_t mutex;
02 int board_Linux = 0;
03 int board_Windows = 0;

   int isCaptain = 0;

04 void board_boat(char C)
05 {
06   pthread_mutex_lock(&mutex);
07   switch(C)
08     {
09     case 'L':
10       board_Linux++;
11       break;
12     case 'W':
13       board_Windows++;
14       break;
15     default:
16       printf("Wrong kind\n");
17     }
18   pthread_mutex_unlock(&mutex);
19 }

20 void row_boat(void)
21 {
22   pthread_mutex_lock(&mutex);
23   if ((board_Linux == 1) ||
24       (board_Linux == 3) ||
25       (board_Windows == 1) ||
26       (board_Windows == 3) ||
27       (board_Linux + board_Windows != 4))
28     {
29      printf("Illegal mix\n");
30     }
31   else
32     {
33       printf("Rowing\n");
34       board_Linux = 0;
35       board_Windows = 0;
36    }

    isCaptain = 0;

37  pthread_mutex_unlock(&mutex);
38 }
```

```
39 pthread_mutex_t BM;  // barrier mutex
40 pthread_cond_t BV;   // barrier condition variable
41 int num = 0;         // number of processes in barrier

42 void barrier(int N)
43 {
44   pthread_mutex_lock(&BM);
45   while (state == EXITING)
46     {
47       pthread_cond_wait(&BV, &BM);
48     }
49   num++;
50   pthread_cond_broadcast(&BV);
51   while (state == ENTERING)
52     {
53       pthread_cond_wait(&BV, &BM);
54       if (num == N)
55         {
56           state = EXITING;
57           pthread_cond_broadcast(&BV);

             if (isCaptain == 1)
               {
                 row_boat();
                 sem_post(&M);
               }

58         }
59     }
60   num--;
61   if (num == 0)
62     {
63       state = ENTERING;
64     }
65   pthread_cond_broadcast(&BV);
66   pthread_mutex_unlock(&BM);
67 }


68 int main(void)
69   {
70   pthread_mutex_init(&BM, NULL);     // init barrier mutex
71   pthread_mutex_init(&mutex, NULL);      // init boat mutex

     sem_init(&M, 0, 1);
     sem_init(&LPQ, 0, 0);
     sem_init(&WPQ, 0, 0);
```

```
72   srand(getpid()); // set random seed

73   while (1)
74     {
75       if ((rand() % 10) >= 5)
76         {
77           pthread_t thread_handle;
78           pthread_create(&thread_handle, NULL, Linux_thread, 0);
79           pthread_detach(thread_handle);
80         }
81       else
82         {
83           pthread_t thread_handle;
84           pthread_create(&thread_handle, NULL, Windows_thread, 0);
85           pthread_detach(thread_handle);
86         }
87     }
88   return 0;
89 }


sem_t M;  // "Mutex" semaphore

int Linux_Passengers = 0;
sem_t LPQ;  // Linux Passenger queue semaphore

void *Linux_thread(void *arg)
{
  sem_wait(&M);
  Linux_Passengers += 1;
  if (Linux_Passengers == 4)
    {
      sem_post(&LPQ);
      sem_post(&LPQ);
      sem_post(&LPQ);
      sem_post(&LPQ);
      Linux_Passengers = 0;
      isCaptain = 1;
    }
  else if ((Linux_Passengers == 2) && (Windows_Passengers >= 2))
    {
      sem_post(&LLQ);
      sem_post(&LLQ);
      sem_post(&WWQ);
      sem_post(&WWQ);
      Windows_Passengers -= 2;
      Linux_Passengers = 0;
      isCaptain = 1;
    }
```

```
  else
    {
      sem_post(&M);
    }

  sem_wait(&LPQ);

  board_boat('L');

  barrier(4);
}

int Windows_Passangers = 0;
sem_t WPQ;  // Windows Passenger queue semaphore

void *Windows_thread(void *arg)
{
  sem_wait(&M);
  Windows_Passengers += 1;
  if (Windows_Passengers == 4)
    {
      sem_post(&WPQ);
      sem_post(&WPQ);
      sem_post(&WPQ);
      sem_post(&WPQ);
      Windows_Passengers = 0;
      isCaptain = 1;
    }
  else if ((Windows_Passengers == 2) && (Linux_Passengers >= 2))
    {
      sem_post(&WPQ);
      sem_post(&WPQ);
      sem_post(&LPQ);
      sem_post(&LPQ);
      Linux_Passengers -= 2;
      Windows_Passengers = 0;
      isCaptain = 1;
    }
  else
    {
      sem_post(&M);
    }

  sem_wait(&WPQ);

  board_boat('W');

  barrier(4);
}
```

*(Kommentar: Lösningar som för övrigt inte fungerar får -2p för varje korrektion som*

*måste göras, dock maximalt tre korrektioner då lösningen annars blir att betrakt som någon annans. Dvs, lösningar som bedöms behöva för många åtgärder för att fungera får inga poäng.*