Tentamen
Datorteknik och realtidssystem, TSEA81

Datum	2022-01-10
Lokal	R41, R37, FE246
Tid	08-12
Kurskod	TSEA81
Provkod	TEN1
Kursnamn	Datorteknik och realtidssystem
Institution	ISY
Antal uppgifter	5
Antal sidor (inklusive denna sida)	15
Kursansvarig	Kent Palmkvist
Lärare som besöker skrivsalen *Telefon under skrivtiden*	Anders Nilsson 013-28 2635
Besöker skrivsalen	Ca 09 och 11
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 3 tidsenheter i tidsintervallet [*i**8, (*i*+1)*8]
- *P2* ska arbeta/köra under *x* tidsenheter i tidsintervallet [*i**6, (*i*+1)*6]
- *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 Rate Monotonic Scheduling (RMS) används. Antag samma värde på *x* som i a-uppgiften.
Visa vad som händer när programmet körs genom att rita ett tidsdiagram.
Kommer processerna att uppfylla kraven?
Hur stor blir den faktiska utnyttjandegraden?
Kan sambandet för RMS som säger att om $U_e \leq n(2^{1/n} - 1)$ så uppfylls kraven, användas i detta fall för att avgöra om kraven uppfylls eller inte? Motivera svaret.
Ledning:
$1(2^{1/1} - 1) = 1.00$
$2(2^{1/2} - 1) \approx 0.83$
$3(2^{1/3} - 1) \approx 0.78$
$4(2^{1/4} - 1) \approx 0.76$
...

Uppgift 2: Teori(8p)

(a) (2p) Ge två exempel på situationer där det finns grund för processbyte (dvs schemaläggaren `schedule()` anropas), men som av något skäl *inte* leder till ett processbyte.

(b) (2p) Ge exempel på två situationer som garanterat leder till ett processbyte.

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

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

Uppgift 3: Starka och svaga semaforer(10p)

Betrakta programkoden för de två processerna nedan. Antag att semaforen `S` är initierad till `1` och att både `P1` och `P2` initialt är körklara. Funktionen `do_work()` tar en viss obestämd tid att utföra, men använder inte funktionerna `si_sem_wait()`, `si_sem_signal()` eller `si_wait_n_ms()`.

```
/* task P1 */
void P1(void)
{
    si_sem_wait(&S);      // wait on semaphore
    printf("A1\n");
    si_wait_n_ms(2000);   // sleep for 2000 ms
    printf("B1\n");
    si_sem_signal(&S);    // signal on semaphore
    printf("C1\n");
    do_work();            // do some work for some time
    printf("D1\n");
    si_sem_wait(&S);      // wait on semaphore
    printf("E1\n");
    si_wait_n_ms(2000);   // sleep for 2000 ms
    printf("F1\n");
    si_sem_signal(&S);    // signal on semaphore
    printf("G1\n");
    si_wait_n_ms(1000);   // sleep for 1000 ms
    printf("H1\n");
    while(1);             // wait forever
}

/* task P2 */
void P2(void)
{
    si_sem_wait(&S);      // wait on semaphore
    printf("A2\n");
    si_wait_n_ms(1000);   // sleep for 1000 ms
    printf("B2\n");
    do_work();            // do some work for some time
    printf("C2\n");
    si_sem_signal(&S);    // signal on semaphore
    printf("D2\n");
    si_sem_wait(&S);      // wait on semaphore
    printf("E2\n");
    si_sem_signal(&S);    // signal on semaphore
    printf("F2\n");
    si_wait_n_ms(1000);   // sleep for 1000 ms
    printf("G2\n");
    while(1);             // wait forever
}
```

Uppgiften fortsätter på nästa sida

(a) (2p) Antag att semaforen `S` är en svag semafor samt att `P2` har högre prioritet än `P1`. Vad kommer programmet att skriva ut efter att det startats? Endast utskrift behöver redovisas.

(b) (8p) Antag att semaforen `S` är en stark semafor samt att `P1` har högre prioritet än `P2`. Beskriv steg för steg vad som händer från det att programmet startats. Var noga med att tala om vilka processer som är körande, vilka listor dom ligger i vid olika tillfällen, semaforens status, samt motivera varför olika händelser sker. Listornas exakta namn är inte viktigt, bara det framgår vad deras syfte är.

Uppgift 4: Leksaksbil(12p)

En automatiserad fabrik som tillverkar leksaksbilar önskar utföra en enkel simulering för tillverkningen av bilarna. Leksaksbilarna består av tre olika typer komponenter: ratt, chassi och däck. En komplett bil byggs samman av en ratt, ett chassi och fyra däck.

Varje komponenttyp produceras i slumpmässig ordning och kommer via ett löpande band fram till en monteringsrobot. Monteringsroboten plockar åt sig delarna i den ordning de kommer och så snart roboten har delarna till en bil, dvs minst en ratt, minst ett chassi och minst fyra däck så ska den bygga en bil. Inga komponentdelar får förkastas, dvs alla ska någon gång användas (så länge simuleringen pågår). Roboten får alltså lägga hittills oanvända delar på hög tills den har tillräckligt med delar för att bygga en bil.

Varje komponenttyp simuleras av en programtråd. Huvudprogrammet (nästa sida) skapar programtrådarna (dvs komponenttyperna) slumpmässigt. Din uppgift är att skriva programkoden för de tre programtrådarna, `SW_thread` (en ratt), `C_thread` (ett chassi) och `T_thread` (ett däck). Utöver det behöver du deklarera eventuella egna funktioner, variabler, semaforer, mutexar och/eller händelsevariabler.

Att bygga en bil innebär bara att programtrådarna för respektive komponent som behövs till en bil, ska avslutas, men bara de trådar som används för att bygga bilen ska avslutas, innan nästa bil byggs. Simuleringen behöver dock inte använda delarna i den ordning de kommer på det löpande bandet, dvs i den ordning huvudprogrammet skapar trådarna.

T ex. Antag att det finns (roboten har) två st rattar, två st chassin och tre st däck. Då går det inte att bygga en bil, men om det då tillkommer ett däck så ska en bil byggas, dvs en ratt-tråd ska avslutas, en chassi-tråd ska avslutas och fyra däck-trådar ska avslutas.

Lösningen ska inte skapa ytterligare trådar, och lösningen ska inte använda meddelandehantering. Du kan anta att nödvändiga programbibliotek är inkluderade. Tänk på att deklarera och initera egna variabler, men det räcker om dessa initieringar finns som kommentar, dvs det måste inte vara programkod, bara det är tydligt med vad som avses. Kommentera programkoden när det inte är uppenbart vad den gör.

Uppgiften fortsätter på nästa sida

Huvudprogrammet nedan producerar kontinuerligt programtrådarna (dvs komponenterna) till leksaksbilen, med lämplig slumpmässighet.

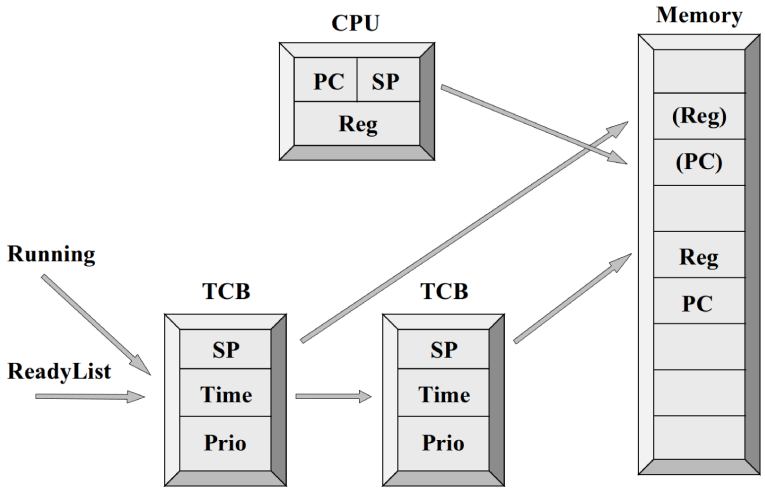
```
int main(void)
{
    int r = 0;

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

    while (1)
    {
        r = rand() % 6;

        if (r == 0)
        {   // create one steering wheel
            pthread_t thread_handle;
            pthread_create(&thread_handle, NULL, SW_thread, 0);
            pthread_detach(thread_handle);
        }

        else if (r == 1)
        {   // create one chassi
            pthread_t thread_handle;
            pthread_create(&thread_handle, NULL, C_thread, 0);
            pthread_detach(thread_handle);
        }

        else
        {   // create one tire
            pthread_t thread_handle;
            pthread_create(&thread_handle, NULL, T_thread, 0);
            pthread_detach(thread_handle);
        }
    }
    return 0;
}
```

Uppgift 5: Processbyte(10p)

Betrakta situationen i figur 1 nedan. Två processer finns i *ReadyList* och en är körande, markerad med pekaren *Running*. CPU:n har under körning använt den körande processens stack och därmed kanske skrivit över tidigare värden för *(Reg)* och *(PC)* varför dessa satts inom parentes då de nu får anses vara inaktuella.

Utgå från denna situation och beskriv steg för steg hur ett processbyte går till, med avseende på vad som händer i CPU:n, processernas stackar och TCB.

Om du vill kan du i din beskrivning förutom pekarna *Running* och *ReadyList* använda pekarna *Current* (som tillfällig markering under bytet av körande process), *Next* (för att markera nästkommande körande process) samt *WaitList* eller eventuellt *TimeList*.



Figur 1: Två processer, en körande.

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:
$3/8 + x/6 + 1/4 \leq 1$
$9/24 + 4x/24 + 6/24 \leq 24/24$
$4x \leq 9$
$x \leq 9/4$
$9/4$ är större än 2 men mindre än 3, alltså, största värde på x 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# _ _ _|_ _ # _|_ # _ _|_ # _ _|# _ _ _|_ _ # .|
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 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
```

Alla processer klarar (förstås) sina deadlines och den faktiska utnyttjandegraden blir 23/24 (ty x=2 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), att värdet på x=2, samt att den faktiska utnyttjandegraden blir 23/24)

1b

MED RMS får processerna prioriteter utefter hur ofta de ska köras, dvs ju kortare tidsintervall (periodtid) 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 0 1 1 1 1 1 1 1 1 1 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
```

P2 och *P3* klarar sina deadlines, *P1* missar sin deadline vid t=8. Den faktiska utnyttjandegraden blir 22/24. Vid tidpunkten t=24 blir samtliga processer samtidigt redo

att köra igen, dvs där börjar förloppet om.

Sambandet $U_e \leq n(2^{1/n} - 1)$ kan bara säga att kraven uppfylls när sambandet är uppfyllt, men kan inte säga om kraven uppfylls eller inte om sambandet inte är uppfyllt. Dvs, i detta fall kan sambandet *inte* användas för att avgöra om processerna klarar sina krav. För även om sambandet inte uppfylls (ty U_e=23/24 $\geq n(2^{1/n} - 1)$ = $3(2^{1/3} - 1)$ $\approx$ 0.78) så kan processerna fortfarande klara sina krav, vilket dom dock inte gör i detta fall.

(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, samt motivera varför sambandet för U_e inte kan användas.)

Lösningsförslag fråga 2:

2a

Möjliga svar:
En ny process skapas, men den har inte högre prioritet än körande process.
En process har väntat färdigt på en timer, men den processen har inte högre prioritet än körande process.
En process gör Signal på en semafor som en annan process väntar på, men den väntande processen har inte högre prioritet än den körande processen.

2b

Möjliga svar:
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

2c

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.

2d

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.

Lösningsförslag fråga 3:

3a

Programmet skriver ut: A2 B2 C2 D2 E2 F2 A1 G2

Kommentar: Det finns en avgörande punkt när P2 begär tillgång till semaforen en andra gång precis innan utskrift av "D2". P1 har dessförinnan begärt tillgång till semaforen, och för en svag semafor medför det att P2 får semaforen och svälter ut P1. I ett senare skede, efter utskrift av "F2", när P2 sover utan att ha semaforen får P1 chansen att skriva ut "A1" och sedan sova. P2 vaknar först, skriver ut "G2", och går sedan in i oändligheten vilken inte avbryts av P1 med lägre prioritet. Rätt fram till och med "E2" ger 1p, allt rätt (varken mer eller mindre) ger 2p.

3b

Här finns flera tillfällen som kan orsaka ett processbyte. När en process gör sleep (och en annan process är körklar), när en process återkommer från sleep och har högre prioritet, 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). Semaforens värde finns i kolumnen (S).

	Orsak	Verkan				
		kör	S	R	W	T
1)	Sched	P1	1	P1,P2		
2)	P1:wait(S)	P1	0	P1,P2		
3)	P1:sleep(2)	P2	0	P2		P1
4)	P2:wait(S)	–	0	–	P2	P1
5)	P1:signal(S)	P1	0	P1,P2		
6)	P1:wait(S)	P2	0	P2	P1	
7)	P2:sleep(1)	–	0	–	P1	P2
8)	P2:signal(S)	P1	0	P1,P2		
9)	P1:sleep(2)	P2	0	P2		P1
10)	P2:wait(S)	–	0	–	P2	P1
11)	P1:signal(S)	P1	0	P1,P2		
12)	P1:sleep(1)	P2	0	P2		P1
13)	P2:signal(S)	P2	1	P2		P1
14)	P2:sleep(1)	–	1	–		P1,P2
15)	P1:while(1)	P1	1	P1		P2
16)	P1:while(1)	p1	1	P1,P2		

1) Från det att båda processerna är körklara blir P1 (högst prioritet) körande.

2) P1 gör wait(S), S>0 så S räknas ned (S–)

3) P1 gör printf(A1), P1 gör sleep(2) så P1 till T, varpå P2 (redo) blir körande

4) P2 gör wait(S), S=0 så P2 till W, ingen process redo, ingen kör

5) P1 vaknar, blir redo och körande, printf(B1), gör signal(S) varpå P2 flyttas till R, P1 högst prio fortsätter, S räknar inte upp ty P2 väntade på S stark semafor, printf(C1), do_work(), print(D1)

6) P1 gör wait(S), S=0 så P1 till W, varpå P2 (redo) blir körande

7) P2 gör printf(A2), gör sleep(1), ingen redo, ingen kör

8) P2 vaknar blir redo och körande, printf(B2), do_work(), print(C2), gör signal(S), varpå P1 blir redo och körande ty högre prio, S räknar inte upp ty P1 väntade på S stark semafor

9) P1 gör printf(E1), gör sleep(2) P1 till T, varpå P2 (redo) blir körande

10) P2 gör print (D2), wait(S), S=0, så P2-¿W, ingen redo, ingen kör

11) P1 vaknar, gör print(F1), signal(S), P2 i W till R, P1 fortsätter ty högst prio, printf(G1)

12) P1 gör sleep(1), så P1 till T, P2 redo blir körande

13) P2 gör print(E2), signal(S), ingen väntar i W så S++, gör print(F2)

14) P2 gör sleep(1), så P2 till T, ingen redo, ingen kör

15) P1 vaknar, gör print(H1) sedan while(1), dvs inga fler processbyten

16) P2 vaknar (via timeravbrott, inte processbyte), P2 till R

Programmet skriver ut: A1 B1 C1 D1 A2 B2 C2 E1 D2 F1 G1 E2 F2 H1

Kommentar: Beskrivningar här är av största vikt. Bara en tabell där det inte framgår vad som är orsak och verkan blir mycket svårbedömt. Det behöver framgå vilken process och vilken operation (orsak) som leder till körande process, semaforens värde och listornas innehåll (verkan).

Lösningsförslag fråga 4:

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

sem_t M;  // "Mutex" semaphore for common variables

sem_t SW; // Steering Wheel semaphore
sem_t T;  // Tire semaphore
sem_t C;  // Chassi semaphore


// Common variables
int swheel = 0;     // number of produced steering wheels
int tire = 0;       // number of produced tires
int chassi = 0;     // number of produces chassis


void build_car()
{
    // Decrease the number of items used to build a car,
```

```
    // and for each item signal via a semaphore to a
    // corresponding thread to exit
    swheel--;
    sem_post(&SW);
    chassi--;
    sem_post(&C);
    tire -= 4;
    sem_post(&T);
    sem_post(&T);
    sem_post(&T);
    sem_post(&T);
}


// SteeringWheel thread
void *SW_thread(void *arg)
{
    sem_wait(&M);
    swheel++;
    if ((swheel >= 1) && (chassi >= 1) && (tire >= 4))
    {
        build_car();
    }
    sem_post(&M);
    // Wait to exit
    sem_wait(&SW);
}

// Tire thread
void *T_thread(void *arg)
{
    sem_wait(&M);
    tire++;
    if ((swheel >= 1) && (chassi >= 1) && (tire >= 4))
    {
        build_car();
    }
    sem_post(&M);
    // Wait to exit
    sem_wait(&T);
}


// Chassi thread
void *C_thread(void *arg)
{
    sem_wait(&M);
    chassi++;
    if ((swheel >= 1) && (chassi >= 1) && (tire >= 4))
    {
```

```
        build_car();
    }
    sem_post(&M);
    // Wait to exit
    sem_wait(&C);
}


// Main program for producing car components
int main(void)
{
    sem_init(&M, 0, 1);
    sem_init(&SW, 0, 0);
    sem_init(&T, 0, 0);
    sem_init(&C, 0, 0);

    int r = 0;

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

    while (1)
    {
        r = rand() % 6;

        if (r == 0)
        {   // create one steering wheel
            pthread_t thread_handle;
            pthread_create(&thread_handle, NULL, SW_thread, 0);
            pthread_detach(thread_handle);
        }

        else if (r == 1)
        {   // create one chassi
            pthread_t thread_handle;
            pthread_create(&thread_handle, NULL, C_thread, 0);
            pthread_detach(thread_handle);
        }
        else
        {   // create one tire
            pthread_t thread_handle;
            pthread_create(&thread_handle, NULL, T_thread, 0);
            pthread_detach(thread_handle);
        }
    }
    return 0;
}
```

(Kommentar: Korrektioner som måste göras i inlämnad lösning för att den ska fungera ger olika mycket poängavdrag beroende på hur mycket som måste korrigeras för att lösningen ska fungera. Om lösningen inte har tillräcklig stomme som ger någon grundläggande funktion alls, ges inga poäng)

Lösningsförslag fråga 5:

Processbytet sker i följande ordning:

1. Justera fälten i TCB för körande process, om det behövs. T ex för att si_wait_n_ms() har anropats eller att prioriteten har ändrats.
2. Flytta TCB för körande process från *ReadyList* till en annan lista: *TimeList* om si_wait_n_ms() har anropats, eller någon *WaitList* beroende på vad man väntar på.
3. Leta upp nästa process, den med högst prioritet i *ReadyList*, och markera den med *Next*.
4. Markera nu körande process med *Current*, och markera den process som ska bli körande, dvs *Next*, med *Running*.
5. Kopiera PC i CPU till stacken för nu körande process, dvs den markerad med *Current*. (görs t ex via PUSH)
6. Kopiera Reg i CPU till stacken för nu körande process, dvs den markerad med *Current*. (görs t ex via PUSH)
7. Sätt SP i TCB för körande process (*Current*) till samma som SP i CPU.
8. Sätt SP i CPU till SP i TCB för den process som ska starta (*Next*).
9. Kopiera Reg från stacken för den process som ska starta (*Next*) till Reg i CPU. (görs t ex via POP)
10. Kopiera PC från stacken för den process som ska starta (*Next*) till PC i CPU. (görs t ex via POP)

I och med sista steget så sker ett hopp till den nya process som ska köra, dvs den som nu är markerad med *Running*. Pekarna *Next* och *Current* har längre ingen betydelse.