TSEA81 Datorteknik och realtidsystem

Föreläsning 5

LINKÖPINGS UNIVERSITET

Agenda

- Reflektion Lab2 och Lab3
- Repetition Lab4
- Real-time kernel + Real-time OS
- Stark/Svag semafor + semafor med time-out
- Information till Lab5

Reflektion Lab 2 och Lab 3

Reflektion : Lab 2

main.c

```
void * clock_task(void *unused)
{
  int hours, minutes, seconds;

  while(1) {
    clock_get_time(&hours,&minutes,&seconds);
    display_time(hours,minutes,seconds);

    if (clock_time_for_alarm()) {
      clock_activate_alarm();
    }

    ...
    clock_nanosleep( ...
    clock_increment_time();
  }
}
```

clock.c (module)

```
/* clock data type */
typedef struct {
  time_data_type time;
  time_data_type alarm_time;
  int alarm_enabled;
  pthread_mutex_t mutex;
  sem_t start_alarm;
} clock_data_type;

static clock_data_type Clock;

void clock_init( ...

void clock_set_time( ...

void clock_set_alarm( ...

Void clock_increment_time( ...
```

Lämpligen bor trådarna i main.c tillsammans med huvudprogrammet.

Reflektion : Lab 2

alarm_thread1:

```
while(1)
{
  sem_wait(&S);
  display_alarm_text();
  usleep(1500000);
  if (alarm_still_enabled())
  {
    sem_post(&S);
  }
}
```

alarm_thread2:

```
while(1)
{
  sem_wait(&S);
  while (alarm_still_enabled())
  {
    display_alarm_text();
    usleep(1500000);
  }
}
```

Alarm_thread1 är ”oortodox”. Man kan t ex inte kan skilja på en yttre och inre aktivering (via `sem_post`). Alarm_thread2 är en bättre lösning.

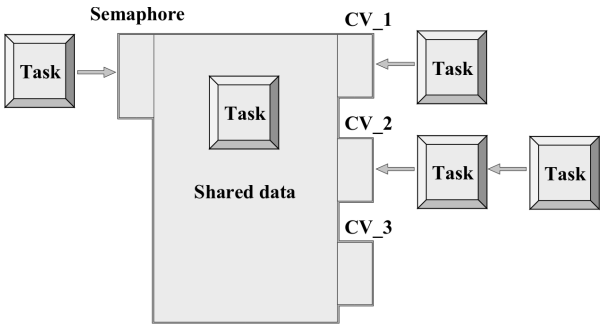
`alarm_still_enabled()` läser av gemensam resurs `alarm_enabled` inom mutex:ar.

Reflektion : Lab 3

- Vad handlade Lab 3 om egentligen?
- Skulle figurerna åka hiss och ”**bara hinna**” gå av/på innan hissen åkte vidare?
- **Nej**!
  - Väntetider (usleep) är bara till för att vi människor ska hinna se vad som händer.
  - Alla trådar ska fungera, dvs figurerna ska kunna gå av/på utan problem, även om det inte finns några väntetider (usleep).
  - Dvs, uppgiften var att få samtliga trådar att göra sitt arbete skyddat via en monitor, dvs mutex, händelsevariabel och tillhörande funktioner.
  - I Lab 3 sköter varje resande (tråd) sig själv, dvs går av/på hissen av egen kraft.

Reflektion : Lab 3

Monitor:



- En abstrakt datatyp (struct) med datafält för delade resurser
- Semaforer samt händelsevariabler för att skydda delade resurser
- Funktioner för att via semaforer och händelsevariabler använda delade resurser

Det räcker inte med att *bara* skydda enskilda datafält, utan även *konsistensen (samhörigheten)* mellan datafälten behöver skyddas.

Reflektion : Lab 3

thread1:

```
while(1)
{
  mutex_lock(&M)
  ...
  ...
}
```

thread2:

```
while(1)
{
  ...
  ...
  mutex_unlock(&M)
}
```

- Mutex funkar inte mellan trådar, dvs samma tråd måste både låsa och låsa upp.
- (Det går dock med en semafor, via sem_wait och sem_post)
- Däremot kan olika trådar använda samma mutex, förstås.
- (Vilket är nödvändigt för att skydda samma gemensamma resurs)

Reflektion : Lab 3

```
while(1)
{
  if (villkor)
   mutex_lock(&M);
     ...
     ...
  if (villkor)
   mutex_unlock(&M);
}
```

```
while(1)
{
  mutex_lock(&M);
  while (villkor)
   cond_wait(&C,&M);
     ...
  cond_broadcast(&C);
  mutex_unlock(&M);
}
```

Det funkar inte att skapa villkorligt kritiska regioner genom att via ett villkor köra mutex_lock (dvs Wait) eller mutex_unlock (dvs Signal). Dvs, använd villkorsvariabler och Await och Cause till villkorligt kritiska regioner.

Reflektion : Lab 3

Type casting

```
void *user_thread(void *unused)
{
  int id;
  ...
  pthread_create(...,
                 passenger_thread,
                 (void *) &id);
  sem_wait(&S);
  id++;
  ...
}
```

```
void *passenger_thread(void *idptr)
{
  int *tmp = (int *) idptr;
  int id = *tmp;
  sem_post(&S);
  ...
}
```

Även följande fungerar:

```
int id = *(int*) idptr;
```

Tillverkaren av `pthread_create` kan inte veta vilken typ av argument användaren vill använda, därav `(void *)` en pekare av obestämd typ, som hos den startade tråden sedan kan tolkas (cast:as) till rätt typ.

Reflektion : Lab 3

- Skilj på dessa

Semafor-operation	Linux	Simple-OS
Wait	sem_wait(&S)	si_sem_wait(&S)
Signal	sem_post(&S)	si_sem_signal(&S)

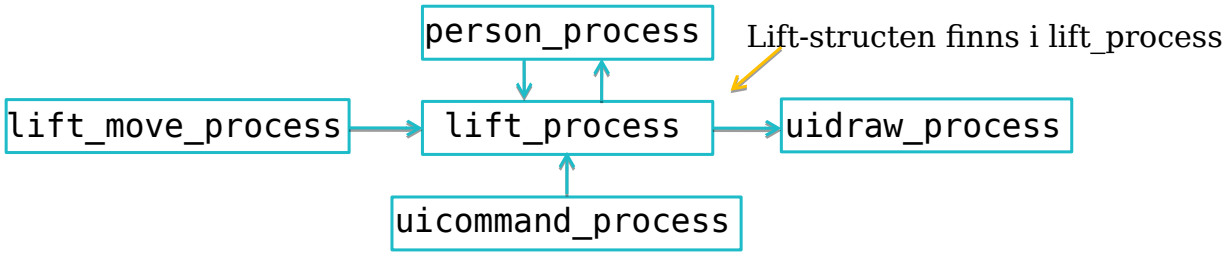
Mutex-operation	Linux	Simple-OS
Lock	pthread_mutex_lock(&M)	
Unlock	pthread_mutex_unlock(&M)	

Händelse-operation	Linux	Simple-OS
Await	pthread_cond_wait(&C, &M)	si_ev_await(&C)
Cause	pthread_mutex_broadcast(&C, &M)	si_ev_cause(&C)

Repetition Lab 4

Lab4 : Message passing

- Inga trådar, inga mutexar, inga semaforer, bara processer och meddelanden
- person_process skickar ett TRAVEL-meddelande till lift_process:
  - Skapa en person på våning X och skicka ett TRAVEL_DONE till mig när personen nått sin våning Y.
  - lift_move_process skickar MOVE-meddelanden varje gång hissen ska flyttas en våning



- Notering: draw_lift får bara anropas från en process. Detta sker ifrån uidraw_process. (Lift-strukturen skickas till uidraw-processen som ett meddelande.)
- I Lab 4 sköter hissen varje resande, dvs flyttar dessa av/på hissen. Återanvänd (och modifiera) lift.c och lift.h från Lab 3.

Message passing

Funktioner: (ingår i messages.c / messages.h i uppgift 4)

```
int message_receive(char *msg, int length, int queueid);
// Tar emot meddelanden på angiven meddelandekö, returnerar längden.
// Meddelanden skrivs till buffern msg och får max vara längden length.
// Blocking (väntar på ett meddelande innan den returnerar)


void message_send(char *msg, int length, int queueid, int priority);
// Skickar meddelandet i buffern msg av längden length till angivet
// queue-id med angiven prioritet.
// Non-blocking (om det finns plats i kön, annars blocking)
```

Implementationen av messages liknar put_item/get_item

Lab 4 : Message passing

Type casting

```
void liftmove_process(void)
{
  struct lift_msg m;
  ...
  m.type = LIFT_MOVE;
  message_send((char *) &m,
               sizeof(m),
               QUEUE_LIFT,
               0);
  ...
}
```

```
void lift_process(void)
{
  char msgbuf[4096];
  ...
  int len = message_receive(msgbuf,
                            4096,
                            QUEUE_LIFT);
  ...
  struct lift_msg *m;
  m = (struct lift_msg *) msgbuf;
  switch (m->type) {
    case LIFT_MOVE:
```

Tillverkaren av `message_send` vill att meddelandet ska vara en sträng `(char *)`, vilket medför att användaren måste tala om hur stort meddelandet är. Mottagaren, `message_receive`, måste sedan tolka om (cast:a) strängen till rätt typ innan användning.

Lab 4 : Message passing

- Zombie-processer eller Orphan-processer
  - En Zombie-process är egentligen avslutad, men den finns fortfarande kvar i processlistan
  - En Orphan-process är en kvarvarande (föräldralös) process
- Om programmet kraschar eller på annat sätt avslutar på ”felaktigt” sätt, kan man behöva slå ihjäl kvarvarande processer:

```
-bash-4.2$ ps aux | grep lift_messages
andni65  13932  0.0  0.0   8612   664 pts/0    S    17:49   0:00 ./lift_messages
andni65  14002  0.0  0.0   8612   124 pts/0    S    17:49   0:00 ./lift_messages
andni65  14003  0.0  0.0   8612   336 pts/0    S    17:49   0:00 ./lift_messages
andni65  14004  0.0  0.0   8612   124 pts/0    S    17:49   0:00 ./lift_messages
andni65  14221  0.0  0.0 112812   988 pts/0    R+   17:50   0:00 grep –color=auto lift_messages
-bash-4.2$ killall -9 lift_messages
[1]+  Killed                  ./lift_messages
-bash-4.2$
```

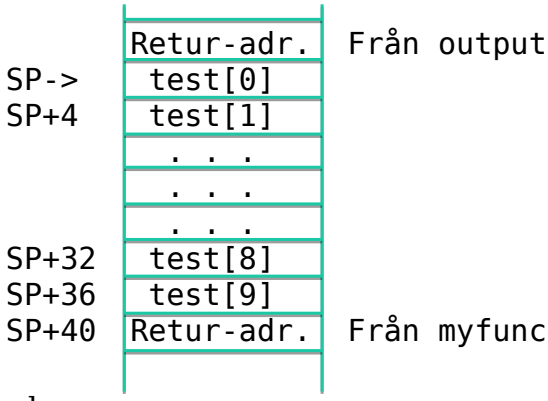
Real-time kernel + Real-time OS

Stackens användningsområde

- JSR / RTS : Returadress
- Temporärdata (push / pop)
- Parameteröverföring till funktioner

```
void myfunc(void) {
  int i; // hamnar kanske i register
  int test[10]; // hamnar på stacken
  for (i=0; i<10; i++) {
    test[i]=42;
  }
  output(&test);
}
```

Stackens innehåll vid anrop av output:



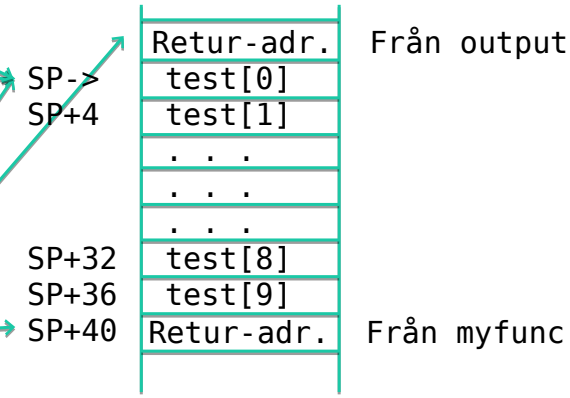
Stacken är en mycket viktig del av tillståndet för en task

Stackens användningsområde

68000-kod för subrutinen:

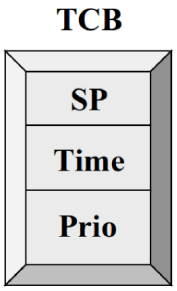
```
myfunc:
  move    #10,d0     ; initiera loopräknare
  add.l   #-40,a7    ; justera stackpekare a7
  move.l  a7,a0      ; kopiera a7 till a0
loop:
  move.l  #42,(a0)+ ; skriv 42 till stack,a0++
  add     #-1,d0     ; minska loopräknare
  bne     loop       ; nästa loopvarv

  move.l  a7,a0      ; a0 pekar på test[]
  jsr     output     ; anropa subr. output
  add.l   #40,a7    ; återställ stackpekare
  rts
```

Stackens innehåll vid anrop av output:



Stackens användningsområde

- En CPU lagrar sitt tillstånd på följande ställen (förenklat):
  - CPU:ns Register (flera stycken)
  - CPU:ns PC
  - CPU:ns Stackpekare
- En process lagrar sitt tillstånd i:
  - En egen stack
  - Ett TCB – Task Control Block



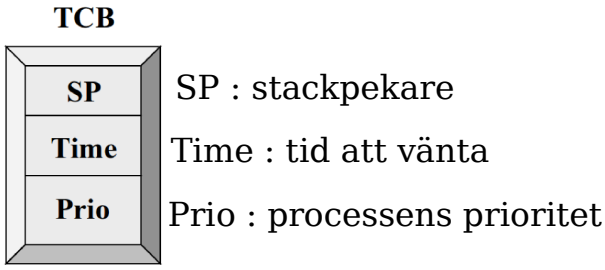
SP : stackpekare

Time : tid att vänta

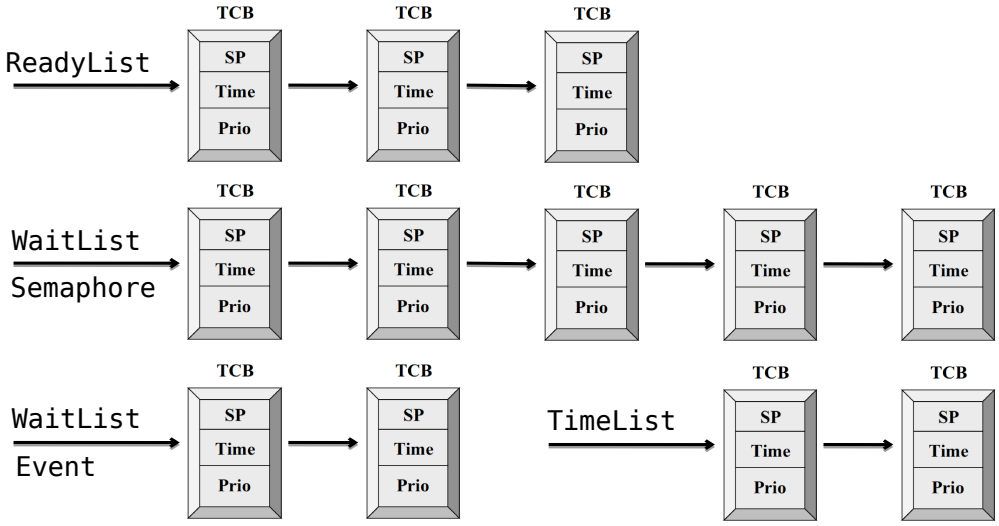
Prio : processens prioritet

TCB : datastruktur

- TCB i ett mycket enkelt OS:
  - Plats för att spara SP (vid processbyte)
    - Ingen plats behövs i TCB för PC och register, då dessa sparas på stacken
  - Plats för timervärde (för usleep t ex)
  - Prioritet (för att avgöra vilken process som ska få köra)
  - I SimpleOS, två fält till:
    - Task ID
    - valid-flagga
- I Linux är TCB nästan 6 kilobyte stor!



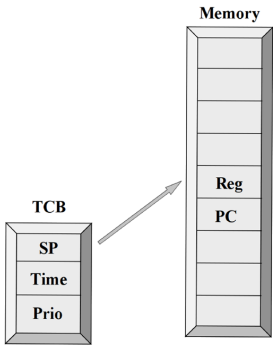
TCB : Olika listor



Tasks / Processer

- En process kan implementeras som en C-funktion
- Processen har ett eget stackutrymme (skapas vid t ex pthread_create)
- Vid ett processbyte (task switch) inaktiveras en process och en annan återupptas (eller startas för första gången)
- En process sparar in sin kontext när den inaktiveras, och återställer kontexten när körning av processen återupptas
- Kontexten (CPU:ns register och programräknare PC, för körande process) sparas på processens stack enligt en given konvention (ordning)

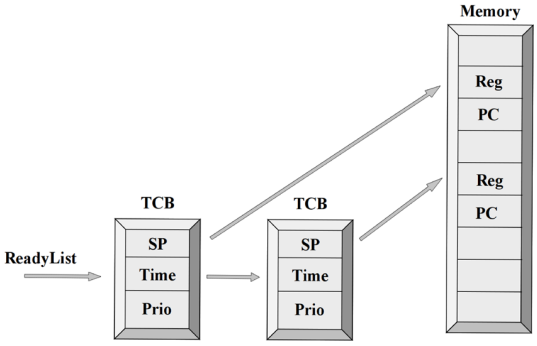
- En icke körande process:



Processer : starta

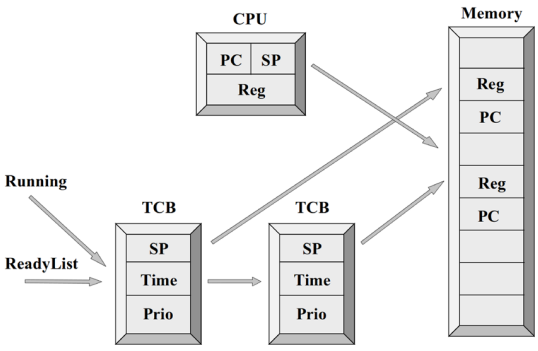
- Skapa ett TCB, fyll i värden för prioritet och tid (typiskt 0)
- Skapa ett stackutrymme för processen i minnet
- Skriv värdet för PC, vilket är startadressen för den funktion som utgör processen, i stacken
- Skriv värden för önskat starttillstånd hos registeruppsättningen, i stacken SP pekar nu på toppen av stacken, så spara SP i TCB
- Processen är nu redo att starta, lägg in TCB i en ReadyList

- Två processer, redo att köra:



Processer : starta

- I SimpleOS finns en ReadyList, som innehåller ID för alla processer som vill köra
- Den av dessa processer som har högst prioritet markeras som Running och kommer att få köra genom att en assemblerrutin:
  - Läser SP från TCB till CPU:ns stackpekare
  - Läser in register från processens stack till CPU:n
  - Läser in PC från processens stack till CPU:n, och därmed sker hopp till processens adress och processen startar
- Processens kontext har återställts till CPU:n
- Två processer, den ena startar:



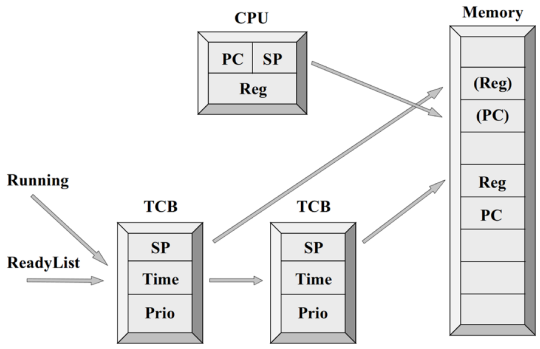
Processer : växla/byta : orsak

- När byter man process?
  - En ny process skapas med högst prioritet
  - Körande process anropar wait, på en semafor och blir väntande
  - Körande process anropar wait, på en timer (`usleep()` el. `si_wait_n_ms()`)
  - En process har väntat färdigt på en timer och har högre prio än körande process
  - En process gör signal/post på en semafor som som en process med högre prioritet väntar på
  - Processbyte kan även initieras av något avbrott, t ex då körande process kört tillräckligt länge och det är dags för nästa process att köra enligt någon prioritetsregel
- Anrop av si_wait_n_ms() (motsv. usleep())
  - Uppdatera TCB med fördröjningsvärdet
  - Spara ner pc och register på stacken, sp i TCB
  - Ta bort processen från ReadyList[]
  - Lägg in processen i TimeList[]
  - Växla till högst prioriterad process i ReadyList[] (dvs anropa Schedule)
- Med jämna mellanrum ifrån ett klockavbrott:
  - Räkna ned fördröjningsvärdet för alla processer i TimeList[]
  - Om värdet nått noll, flytta tillbaka processen till ReadyList[]
  - Växla till högst prioriterad process i ReadyList[] (dvs anropa Schedule)

Processer : växla/byta : justera TCB

- Justera fälten i TCB för körande process, om det behövs:
  - usleep() el. si_wait_n_ms() har anropats
  - prioriteten för processen har ändrats
- Observera att (Reg) och (PC) på stacken för körande process nu betraktas som inaktuella då de kan ha skrivits över av temporärdata under tiden processen har kört.
- SP i TCB för körande process pekar just nu fel, men kommer snart att uppdateras under själva bytet.

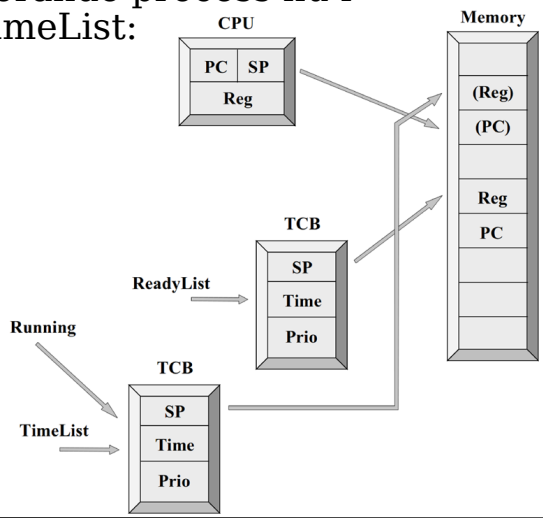
- Processbytet förbereds:



Processer : växla/byta : byt lista

- Flytta TCB för körande process från ReadyList till en annan lista:
  - TimeList, om t ex usleep() el. si_wait_n_ms() anropats
  - Någon annan ”WaitList”, beroende vad man väntar på, t ex en semafor/mutex eller en händelsevariabel

- Körande process nu i TimeList:



Processer : växla/byta : spara kontext

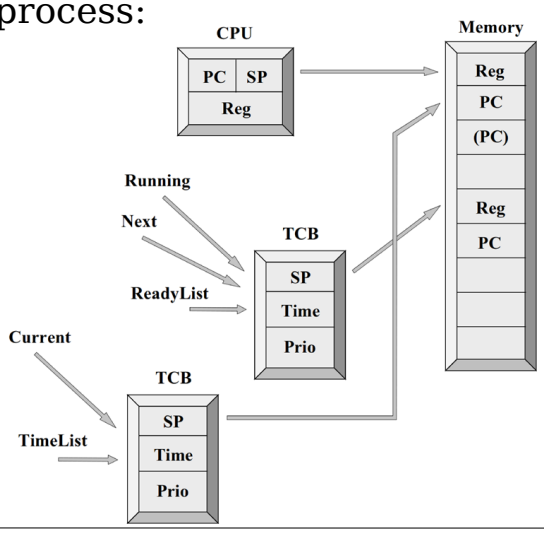
- Leta upp nästa process, den med högst prioritet i ReadyList, och markera den med Next
- Markera körande process med Current
- Markera den process som ska bli körande, dvs Next, med Running
- Kopiera PC i CPU till stacken för körande process [%]
- Kopiera Reg i CPU till stacken för körande process [%]

- Vi har ännu inte bytt process, men flyttad körande process till TimeList och sparat processens kontext på stacken.

  (PC) på stacken upptas av temporärdata för körande process.

  [%] Dessa steg kan åstadkommas med ett mjukvaruavbrott.

- Kontext på stacken för körande process:

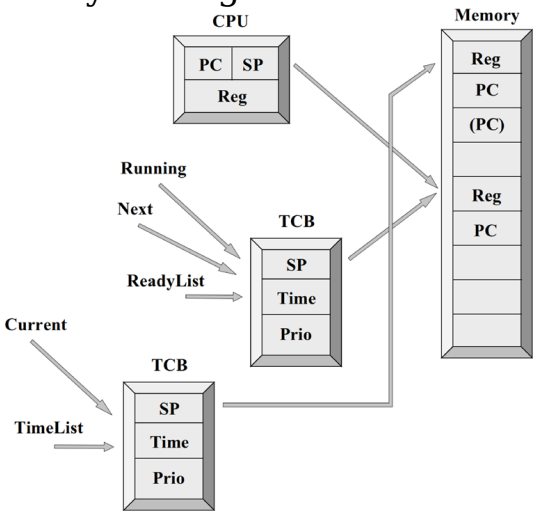


Processer : växla/byta : byt stack

- Sätt SP i TCB för körande process (Current) till samma som SP i CPU
- Sätt SP i CPU till SP i TCB för den process som ska starta, dvs Next

- Nu har vi bytt stack i CPU:n från nuvarande process (Current) till nästa process (Next).

- Stackbytet är genomfört:

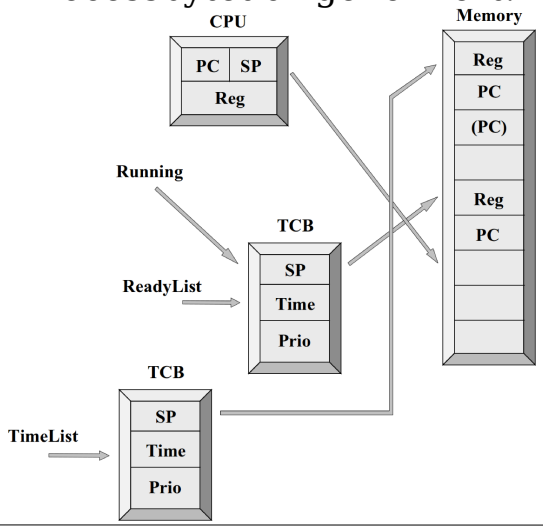


Processer : växla/byta : återta kontext

- Kopiera Reg från stacken för den process som ska starta, dvs Next, till Reg i CPU:n
- Kopiera PC från samma stack till PC i CPU:n

- I och med att PC i CPU:n får ett nytt värde så fortsätter exekveringen vid den adressen, dvs vid den nya processen som nu är körande (Running)
- Stegen ovan att återta Reg och PC till CPU:n kan typiskt (beroende på processorarkitektur) göras med ett återhopp från ett avbrott, dvs den assemblerinstruktion som gör det.

- Processbytet är genomfört:



Processer : växla/byta : avbrott

- Funktionaliteten för kontext-bytet måste skrivas i assembler, eftersom operationerna är arkitekturberoende och går inte att styra via ett högnivåspråk som C.
- Funktionen `si_wait_n_ms()` måste köras med avbrott avslagna! Detta bl a för att Schedule anropas och ett avbrott kan initiera ett nytt processbyte, med nytt anrop av Schedule (under pågående Schedule), vilket kan ge oanade konsekvenser.
- Även andra kritiska funktioner i realtidssystemets kärna, såsom Wait och Signal, måste köras med avbrott avslagna

Princip för `si_wait_n_ms()`:

```
void WaitTime (int time) {
     DISBLE_INTERRUPTS;
     Update_TCB(time);
     Save_Regs();
     int id=Remove_from_ReadyList();
     Add_to_TimeList(id);
     Schedule();
     ENABLE_INTERRUPTS();
 }
```

Schedule ovan (kanske) anropar ny process som ska köra.
-När returnerar man från Schedule()?
-När kommer vi tillbaka till `ENABLE_INTERRUPTS()`?

Processer : växla/byta : avbrott

Princip för si_sem_wait(): ”sem_wait() i Linux”

```
void si_sem_wait(struct sem *s) {
   DISABLE_INTERRUPTS;
   if (s->counter>0) {
     s->counter--;
   } else {
     // flytta oss från ReadyList[]
     int id=Remove_from_ReadyList();
     // till semaforens WaitList[]
     Add_to_WaitList(s->wait_list,id);
     // byt till process med högst prio
     Schedule();
   }
   ENABLE_INTERRUPTS;
 }
```

Princip för si_sem_signal(): ”sem_post() i Linux

```
void si_sem_signal(struct sem *s) {
   DISABLE_INTERRUPTS;
   if (!empty(s->wait_list)) {
     // hitta process som väntar med högst prio
     int id=remove_highest_prio(s->wait_list);
     // lägg till i ReadyList[]
     Add_to_ReadyList(id);
     // byt till process med högst prio
     Schedule();
   } else {
     s->counter++;
   }
   ENABLE_INTERRUPTS;
 }
```


Processer : växla/byta : avbrott

Princip för si_ev_await(): ”pthread_cond_wait()” i Linux

```
void si_ev_await(si_event *ev)
{
  int pid;
  DISABLE_INTERRUPTS;
  if (!wait_list_is_empty(
    ev->mutex->wait_list, WAIT_LIST_SIZE))
  {
    pid = wait_list_remove_highest_prio(
      ev->mutex->wait_list, WAIT_LIST_SIZE);
    ready_list_insert(pid);
  } else {
    ev->mutex->counter++;
  }
  pid = process_get_pid_running();
  ready_list_remove(pid);
  wait_list_insert(ev->wait_list,WAIT_LIST_SIZE,pid);
  schedule();
  ENABLE_INTERRUPTS;
}
```

Princip för si_ev_cause(): ”pthread_cond_broadcast()” i Linux

```
void si_ev_cause(si_event *ev)
{
  int pid, done;
  DISABLE_INTERRUPTS;
  done = wait_list_is_empty(
    ev->wait_list, WAIT_LIST_SIZE);
  while (!done)
  {
    pid = wait_list_remove_one(
      ev->wait_list, WAIT_LIST_SIZE);
    wait_list_insert(
      ev->mutex->wait_list, WAIT_LIST_SIZE, pid);
    done = wait_list_is_empty(
      ev->wait_list, WAIT_LIST_SIZE);
  }
  ENABLE_INTERRUPTS;
}
```

Observera! Det är olika väntelistor i programkoden ovan: `ev->wait_list` för att vänta på händelsevariabeln
`ev->mutex->wait_list` för att vänta på ”semaforen”

Stark / Svag semafor

... Och semafor med time-out

Stark/Svag semafor

Stark semafor

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

Svag semafor

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

Principiell struktur för stark respektive svag semafor tagen ur läroboken.

Stark/Svag semafor

Stark semafor

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```

Svag semafor

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```


Principiell struktur för stark respektive svag semafor tagen ur läroboken.

Stark/Svag semafor

Antag följande scenario:

Task P1: (hög prioritet)

```
while(1)
{
  si_sem_wait(&S);
  //KRITISK REGION
   ...
  si_sem_signal(&S);
}
```

Task P2: (låg prioritet)

```
while(1)
{
  si_sem_wait(&S);
  //KRITISK REGION
   ...
  si_sem_signal(&S);
}
```

I den kritiska regionen kan det tänkas att något händer (t ex usleep) som orsakar processbyte.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

1) *P1 blir körande, eftersom den har högst prioritet*

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

2) *P1 gör wait, S--*

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

3) *Antag nu ett processbyte, t ex P1 gör sleep => P2 kör*
OBS, P1 har inte släppt semaforen

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

4) *P2 gör wait, P2 flyttas till WS då S=0*

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

5) *P1 gör signal, P2 -> R, OBS S=0 fortfarande*

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

6) *P1 har högst prio i R och får fortsätta köra*

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

7) P1 gör wait (igen), S=0 så P1 -> WS och P2 får köra

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

8) P2 gör signal, P1 -> R och får köra (ty högst prio)

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	0	P1, P2		5)
	P1	0	P1, P2		6)
P1:wait	P2	0	P2	P1	7)
P2:signal	P1	0	P1, P2		8)
P1:signal	P1	1	P1, P2		9)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

9) *P1 gör signal, ingen i WS så S++, P1 fortsätter köra*

```
void Wait(SCB *Mutex)
{
  avbrott av
  om count>0 i Mutex
  {
    count-- i Mutex
  }
  annars
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  om WaitList i Mutex är en tom lista
  {
    count++ i Mutex
  }
  annars
  {
    flytta TCB med högst prio från
      WaitList i Mutex till ReadyList,
    anropa Schedule
  }
  avbrott på
}
```


Exemplet ovan kommer ur läroboken, men här i tabellform.

Stark semafor

- Vad hände?
  - Trots att P1 har högre prioritet får P2 köra (rad 7)
- För en **stark semafor** kan inte en högprioriterad process göra anspråk på en resurs flera gånger i följd då en annan process (även om den har lägre prioritet) gjort anspråk på resursen under tiden den var reserverad av den höprioriterade processen.
- Det garanterar att den process som aktiveras av signal får köra (kanske inte direkt) någon gång, och *undviker svält* då en lågprioriterad process inte kan stängas ute.
- Funktionaliteten kan ändras till att bli en **svag semafor**, som då *tillåter svält* och en högprioriterad process garanteras att alltid få köra.

Svag semafor

Orsak	Verkan				
	Kör	S	R	WS	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	1	P1, P2		5)
	P1	1	P1, P2		6)
P1:wait	P1	0	P1, P2		7)
P1:signal	P1	1	P1, P2		8)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

1) P1 blir körande, eftersom den har högst prioritet

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```

LINKÖPINGS UNIVERSITET

Svag semafor

Orsak	Verkan				
	Kör	S	R	WS	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	1	P1, P2		5)
	P1	1	P1, P2		6)
P1:wait	P1	0	P1, P2		7)
P1:signal	P1	1	P1, P2		8)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

2) P1 gör wait, S--

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```

LINKÖPINGS UNIVERSITET

Svag semafor

Orsak	Verkan				
	Kör	S	R	WS	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	1	P1, P2		5)
	P1	1	P1, P2		6)
P1:wait	P1	0	P1, P2		7)
P1:signal	P1	1	P1, P2		8)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

3) Antag nu ett processbyte, t ex P1 gör sleep
=> P2 kör
OBS, P1 har inte släppt semaforen

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```

LINKÖPINGS UNIVERSITET

Svag semafor

Orsak	Verkan				
	Kör	S	R	WS	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	1	P1, P2		5)
	P1	1	P1, P2		6)
P1:wait	P1	0	P1, P2		7)
P1:signal	P1	1	P1, P2		8)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

4) P2 gör wait, P2 flyttas till WS då S=0
P1 blir körande

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```

LINKÖPINGS UNIVERSITET

Svag semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	1	P1, P2		5)
	P1	1	P1, P2		6)
P1:wait	P1	0	P1, P2		7)
P1:signal	P1	1	P1, P2		8)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

5) P1 gör signal, P2 flyttas till WS, OBS: S++

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```

LINKÖPINGS UNIVERSITET

Svag semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	1	P1, P2		5)
	P1	1	P1, P2		6)
P1:wait	P1	0	P1, P2		7)
P1:signal	P1	1	P1, P2		8)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

6) P1 fortsätter köra, ty P1 har högst prio

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```

LINKÖPINGS UNIVERSITET

Svag semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	1	P1, P2		5)
	P1	1	P1, P2		6)
P1:wait	P1	0	P1, P2		7)
P1:signal	P1	1	P1, P2		8)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

7) P1 gör wait, S!=0 så S--
Precis samma läge som steg 2) ovan
P2 utsvulten

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```


Svag semafor

Orsak	Verkan				
	Kör	**S**	**R**	**WS**	
Init	P1	1	P1, P2		1)
P1:wait	P1	0	P1, P2		2)
	P2	0	P1, P2		3)
P2:wait	P1	0	P1	P2	4)
P1:signal	P1	1	P1, P2		5)
	P1	1	P1, P2		6)
P1:wait	P1	0	P1, P2		7)
P1:signal	P1	1	P1, P2		8)

Kör : Körande process
S : Semaforens värde
R : Ready-list
WS : Väntelista semafor

8) *P1 gör signal, ingen i WS, S++*

```
void Wait(SCB *Mutex)
{
  avbrott av
  så länge count==0 i Mutex
  {
    flytta körande process TCB från
    ReadyList till WaitList för Mutex,
    anropa Schedule
  }
  count-- i Mutex
  avbrott på
}
```

```
void Signal(SCB *Mutex)
{
  avbrott av
  nollställ flagga för att anropa Schedule
  om WaitList i Mutex inte är en tom lista
  {
    flytta TCB med högst prio från WaitList
      i Mutex till ReadyList,
    sätt flagga för att anropa Schedule
  }
  count++ i Mutex
  om flagga för anrop av Schedule är satt
  {
    anropa Schedule
  }
  avbrott på
}
```

Semafor med time-out

- Man kan implementera en semafor med utökad funktionalitet, t ex time-out.
- Dvs vänta bara en viss tid på att semaforen blir ledig, annars gör nåt annat:

```
-Wait_timeout:
Om resursen är ledig
   minska count med ett och returnera 0
annars
   flytta till WaitList och TimeList och
   vänta en viss maximal tid och
   returnera 1, eller tills någon gör
   signal på semaforen och returnera 0.
```

```
-Signal_timeout:
Om WaitList är tom
   öka count med ett
annars
    ta bort från WaitList och TimeList,
       placera i ReadyList
    kör Schedule
```

Om `0` returneras har resursen reserverats
Om `1` returneras har resursen **inte** reserverats

Funderingar

- Kan ReadyList[] vara tom, dvs ingen process är redo att köra?
  - Ja, antag att vi har bara två processer och både ligger i si_wait_n_ms() (motsv usleep). Då låter man operativsystemet köra en idle-process, t ex:

```
void idle_task(void)
  {
    while(1) {
      enable_low_power_mode();
    }
  }
```

- Man kan alltså lägga idle-processen som en vanlig process i ReadyList[], vilket förstås innebär att även idle-processen under vissa förutsättningar kan flyttas till t ex WaitList[].
- Att vänta på semaforer i avbrottsrutiner är av det skälet en dålig ide.

Funderingar

- Vad händer om en programmerare använder stackpekaren i sitt program till något annat än en stack?
  - Inbyggt system: Programmeraren får skylla sig själv
  - Generellt OS: För att undvika att programmeraren kraschar hela datorn, utan bara sitt eget program, har man en stack för användare och en annan stack för operativsystemet.
    - T ex i 68000: SSP (Supervisor Stack Pointer)
      USP (User Stack Pointer)
    - ARM/x86/etc har motsvarande funktioner

Information till Lab 5

Information till Lab 5

- Modifiera Lab3 och Lab4 : Ta bort usleep, draw_lift, printf ... och inför funktionalitet för att mäta resans tid
- Optimera Lab3 och Lab4 : Fler händelsevariabler (Lab3), fler förplanerade resor (Lab4)
- Gör en valbar (betyder inte frivillig) uppgift:
  - VIP-resande
  - Standardavvikelse, Max-tid, Min-tid
  - Tid för att skicka meddelande, tid för att göra broadcast
  - Vad händer när man använder flera kärnor/processorer?
  - Lab3 där hissen ”sköter allt”. Lab4 där personen ”sköter allt”.
- Visa resultat grafiskt : T ex spredsheet i Libreoffice, för 10, 20 ... 90, 100 resande

LINKÖPINGS UNIVERSITET