Realtidsprogrammering definieras som *programmering av datorsystem med speciella krav på svarstider*. Det finns mjuka realtidssystem och hårda realtidssystem beroende på hur kraven på svarstider är formulerat.

Gemensamma resurser är resurser som används an två eller flera processer. Vill man att enbart en process ska kunna nå resurserna samtidigt kallas detta *ömsesidig uteslutning*. En sekvens som använder gemensamma resurser kallas för en *kritisk region*. En region är *relativt odelbar* om enbart en process kan göra anspråk på de gemensamma resurserna samtidigt, men ändå kan bli avbruten. Å andra sidan innebär *absolut odelbar* att ingen annan process får exekvera samtidigt som en process är i en kritisk region.

En *villkorligt kritisk region* är en region där användadet av en gemensam resurs är förknippat med ett villkor.

Några begrepp

Dödläge - en situation där processer väntar på varandra och där detta väntetillstånd inte upphävs.

Svält - en situation där en process kontinuerligt får tillgång till en gemensam resurs och en annan resurs aldrig kommer åt detsamma.

Asymmetrisk synkronisering - när en process ska vänta in en annan process som ska leverera data.

Symmetrisk synkronisering - när två processer ska vänta in varandra för att exempelvis utväxla information.

Semafor

Värde - har ett värde som är ≥ 0 och initialiseras till ett visst värde (till 1 ifall den första process som vill nå gemensamma resurser ska få göra det).

Operationen `wait` - om semaforens värde är större än noll, minska semaforens värde med ett och forsätt exekvera. Ifall semaforens värde är noll, försätt processen i ett väntetillstånd. Väntetillståndet är sådant att processen placeras i en väntekö associerad med semaforen. Nu *väntar processen på semaforen.*

Operationen `signal` - om ingen process väntar på semaforen, öka dess värde med ett. Om en eller flera processer väntar på semaforen, se till att en av dessa processer görs körklar.

Händelsevariabel

Till semafor - kopplas till en semafor som används för att skydda en gemensam resurs. Kallas för *den associerade semaforen.*

Operationen `await` - sätter processen i ett väntetillstånd samtidigt som den gemensamma resursen släpps. Alltså utförs en `signal`-operation samtidigt som processen sätts i vänteläge. Vänteläget är sådant att processen placeras i en väntekö associerad med den händelsevariabel på vilken operationen utförs.

Operationen `cause` - flyttar alla processer som väntar på händelsevariabeln till att vända på den associerade semaforen. Om inga processer väntar på händelsevariabeln händer ingenting.

Monitor

Abstrakt datatyp - innehåller funktionalitet för ömsesidig uteslutning samt villkorligt kritiska regioner.

Godtyckliga villkor - ska kunna hantera alla möjliga fall av gemensamma resurser.

Schemaläggning

Prioritetsbaserad schemaläggning - den process med högst prioritet och är körklar får exekvera.

Frivillig schemaläggning - innebär att ett processbyte inte kan tvingas fram, utan måste initieras av den process som exekverar

Påtvingad schemaläggning - ett processbyte kan tvingas fram, exempelvis via avbrott.

Round-robin scheduling - används för att fördela CPU-tid mellan processer med samma prioritet. Processerna får dela rättvist på CPU-tiden.

Prioritetsinversion - en process som gör anspråk på en resurs måste lämna plats på en annan process som har högre prioritet men inte är kopplad till den gemensamma resursen. Trots att det finns processer med högre prioritet som vill nå den gemensamma resursen får de ingen CPU-tid eftersom de väntar på den semafor som har låsts av tidigare processen. Skenbart har den lågprioriterade processen en högre prioritet.

Prioritetsärvning - en lösning till problemet med prioritetsinversion. När en process, P1, vill ha tillgång till en gemensam resurs och det finns en annan process, P2, som har tillgång till resursen, höjs prioriteten på P2 till den prioritet som P1 har. Efter det att P2 är klar med den gemensamma resursen återställs processens prioritet.

Periodiska processer

Det kan finnas tidskrav på periodiska processer. Man kan associera en *sluttid* till varje process. En sluttid är en absolut tid vid vilken processens exekvering

måste vara klar. Antag att processen *P1* ska, under tidsintervallen

$$[i5, (i+1)5],\ \forall i \in \mathbb{Z}$$

exekvera 2 tidsenheter. Processen *P2* ska, under tidsintervallen

$$[i3, (i+1)3],\ \forall i \in \mathbb{Z}$$

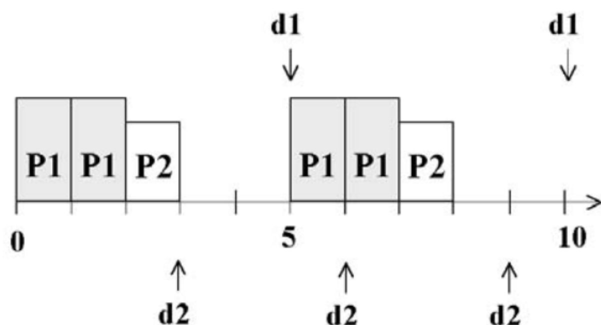
exekvera 1 tidsenhet.

Alltså har *P1* en sluttid var femte tidsenhet och *P2* en sluttid var tredje tidsenhet. *Utnyttjandegraden* definieras av hur mycket processerna använder processorn. Ett system men n processer, där varje process är periodisk med periodtid p_i och behöver exekvera e_i tidsenheter har en utnyttjandegrad U_e på

$$U_e = \sum_{i=1}^{n} \frac{e_i}{p_i}.$$

För *P1* och *P2* fås $U_e \approx 0.73$.

Ifall *P1* har högre prioritet än *P2* fås figur 1.



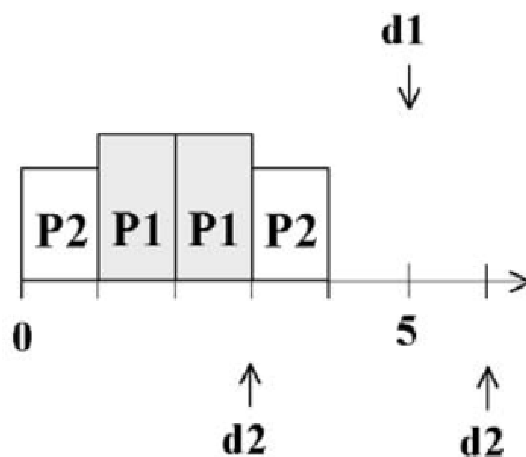
Figur 1: P1 har högst prioritet

P1 möter sina deadlines medan *P2* inte gör det.

I figur 2 har *P2* högst prioritet och därmed möter inte *P1* sina deadlines.

I både figur 1 och 2 har processerna en pseudokod enligt följande.

```
void P_i(void)
{
  while(1)
  {
    Exekvera i e_i tidsenheter
    Vänta i w_i tidsenheter
  }
}
```



Figur 2: P2 har högst prioritet

Där e_i och w_i fås av nedanstående tabell.

Var.	***P1***	***P2***
e_i	2	1
w_i	3	2

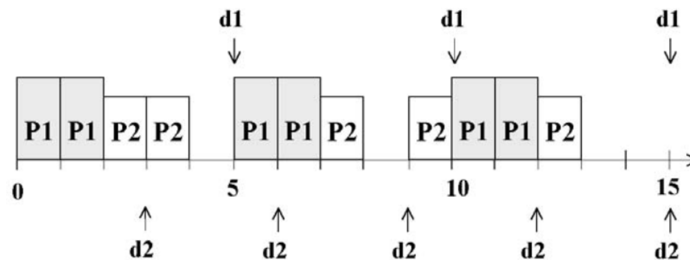
Problem: processer väntar alltid lika länge oberoende när de påbörjade sin väntan.
Lösning: låt processerna drivas med pseudokod enligt nedan.

```
void P_i(void)
{
  while(1)j
  {
    Exekvera i e_i tidsenheter
    Vänta till nästa tidpunkt som är jämnt delbar med t_i
  }
}
```

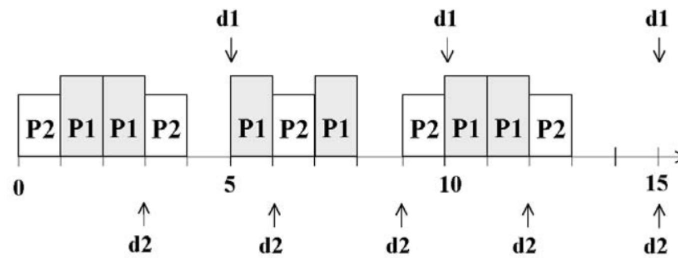
Då fås figur 3 och 4.

Nu håller processerna den teoretiska utnyttjandegraden på 11/15.

Earliest Deadline First (EDF) är en schemaläggningsmetod som innebär att den process som har kortast tid kvar till sin nästa sluttid får exekvera. Den *relativa sluttiden* för en process definieras som tiden från att processen ska bli körklar, fram till processens nästa sluttid. För periodiska processer, med relativa sluttider som sammanfaller med processernas periodtider, och med en förväntad



Figur 3: P2 har högst prioritet



Figur 4: P2 har högst prioritet

$U_e \leq 1$ kommer EDF leda till att processerna håller sina sluttider. Förutsätter att påtvingad schemaläggning används. Processerna får nu pseudokod enligt nedan.

```
void P_i(void)
{
  while(1)
  {
    Exekvera i e_i tidsenheter
  }
}
```

Eftersom väntan hanteras i schemaläggningsalgoritmen istället. En ordning enligt figur 5 fås.

Rate Monotonic Scheduling är inte en metod för schemaläggning utan istället ett sätt att bestämma prioriteter på processer.Prioriteter ska tilldelas så att en kortare periodtid innebär en högre prioritet. Om man använder Rate Monotonic Scheduling och påtvingad schemaläggning med n periodiska processer vars relativa sluttider sammanfaller med periodtiderna och U_e uppfyller

$$U_e \leq n\left(2^{1/n} - 1\right) \to \ln(2) \text{ då } n \to \infty$$

Tid	Δd_1	Δd_2	*Process*	*Kommentar*
0-1	5	3	*P2*	*P2* kortast tid till sluttid
1-2	4	2	*P1*	*P2* ej körklar
2-3	3	1	*P1*	*P2* ej körklar
3-4	2	3	*P2*	*P1* ej körklar
4-5	1	2		*P1* och *P2* ej körklara
5-6	5	1	*P1*	*P2* ej körklar
6-7	4	3	*P2*	*P2* kortast tid till sluttid
7-8	3	2	*P1*	*P2* ej körklar
8-9	2	1		*P1* och *P2* ej körklara
9-10	1	3	*P2*	*P1* ej körklar
10-11	5	2	*P1*	*P2* ej körklar
11-12	4	1	*P1*	*P2* ej körklar
12-13	3	3	*P2*	*P1* ej körklar
13-14	2	2		*P1* och *P2* ej körklara
14-15	1	1		*P1* och *P2* ej körklara

Figur 5: Schemaläggning när EDL används

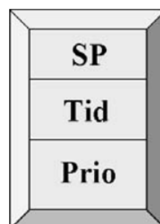
kommer alla processer att hålla sina sluttider.

Uppbyggnad av en realtidskärna

Process

Stack - varje process har en stack, som är ett minnesutrymme reserverat för den aktuella processen. Här lagras data av temporär natur, exempelvis programräknaren och CPU:ns register.

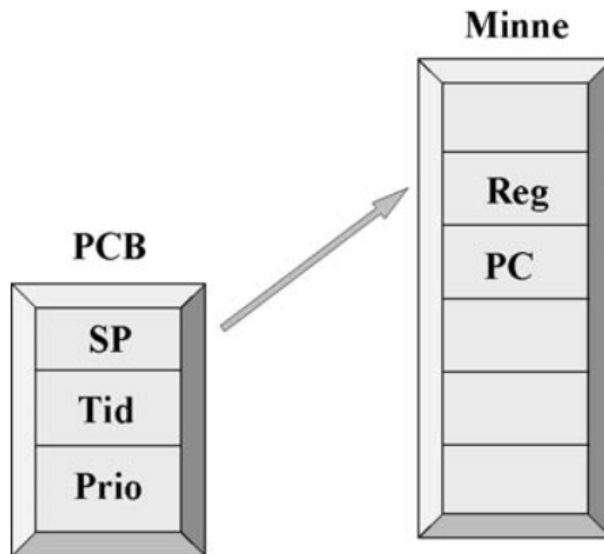
Process Control Block - (PCB) som är en datapost med data specifikt för processen. Bland dessa finns stackpekaren, väntetid och processens prioritet, se figur 6.



Figur 6: Ett PCB med process-specifik information

Att skapa en process

1. Skapa ett PCB, fyll i processens prioritet samt värdet 0 i fältet *T*.

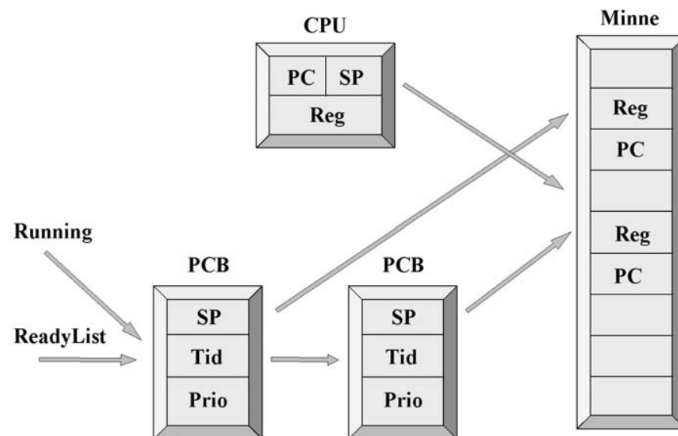


Figur 7: Ett PCB för en process som ej exekverar

2. Skapa en stack i minnet och fyll i startadressen till programkoden på fältet *PC* och *Reg* enligt ett givet mönster.
3. Sätt fältet *SP* i PCB att referera till den stack som skapades.
4. Placera processens PCB i *ReadyList*.

Start av en process

1. Markera processens PCB med en pekare kallad *Running*.
2. Sätt fältet *SP* att referera till samma minnesplats som fältet *SP* i processens PCB.
3. Kopiera innehållet i fältet *Reg* i processens stack till fältet *Reg* i CPU:n genom att *poppa* värdet.
4. Kopiera innehållet från fältet *PC* i processorns stack till motsvarande fält i CPU:n Detta kan göras med ett återhopp från en subrutin, CPU:ns stackpekare uppdateras samtidigt som kopieringen görs. Situationen är nu enligt figur 8.



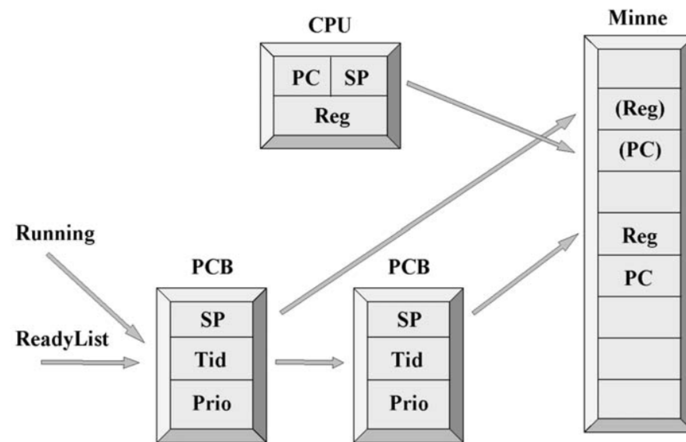
Figur 8: PCB för två körklara processer där processen *Running* ska starta sin exekvering

Byte av process

Antag nu att situationen är enligt figur 9 och att *Running* har exekverat en funktion som väntar en viss tid.

1. Uppdatera fälten i körande process PCB, exempelvis uppdatera fältet *Tid* med argumentet till anropet
2. Flytta körande process PCB från *ReadyList* till exempelvis *TimeList* om ett anrop till en väntefunktion gjorts.
3. Gå igenom *ReadyList* och leta upp det PCB som har högst prioritet. Markera denna process med en pekare *Next*.
4. Sätt en pekare *Current* att referera till den körande processen (den som för nuvarande *Running* pekar på).
5. Flytta pekare *Running* från *Current* till *Next*.
6. Flytta fältet *PC* i CPU:n till stacken.
7. Pusha CPU:ns fält *Reg* till stacken.
8. För processen *Current*, sätt fältet *SP* till samma plats i minnet som *SP* i CPU:n Nu har det översta elementet på stacken markerats.
9. Byt stack. Fältet *SP* i CPU:n ska referera till platsen i minnet som refereras av fältet *SP* i processen *Next*.
10. Uppdatera CPU:ns fält *Reg* med nya processens genom att *poppa* från stacken.

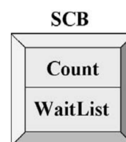
11. Kopiera innehållet från *PC* i nya processens PCB till samma fält i CPU:n genom ett återhopp från en avbrottsrutin.
    - Processbytet är nu genomfört.
    - Variablerna *Current* och *Next* behövs ej längre.



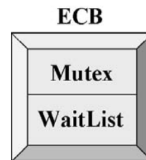
Figur 9: Situation innan *Running* ska bytas till den andra processen. *Running* har sparat ett värde på stacken

Klockavbrott Vid periodiska klockavbrott uppdateras alla processers *Tid*-fält. Flytta processer från *TimeList* till *ReadyList* ifall de har räknat klart. Gå genom *ReadyList* och undersök ifall det finns körklara processer med högre prioritet än nuvarande.

Semaforer och händelsevariabler
En semafor kan definieras med en datastruktur enligt figur 10.
En händelsevariabel kan definieras med en datastruktur enligt figur 11.



Figur 10: Datastruktur för en semafor



Figur 11: Datastruktur för en händelsevariabel

Pseudokod för en semafors funktioner `lock` och `unlock` finns nedan.

```
void sem_lock(semaphore *sem)
{
  DISABLE_INTERRUPTS;

  if (sem->counter > 0) {
    sem->counter--;
  } else {
    remove current process from readylist;
    add current process to wait list for semaphore;
    schedule();
  }
  ENABLE_INTERRUPTS;
}

void sem_unlock(semaphore *sem)
{
  DISABLE_INTERRUPTS;

  if (wait list is empty) {
    sem->counter++;
  } else {
    remove process with highest priority from wait list;
    add same process to ready list;
    schedule();
  }
  ENABLE_INTERRUPTS;
}
```

Pseudokod för en händelsevariabels funktioner `wait` och `broadcast` finns nedan.

```
void ev_wait(event *ev)
{
  DISABLE_INTERRUPTS;

  if (wait list of mutex empty) {
    ev->mutex->counter++;
```

```
  } else {
    remove process with highest priority from wait list of mutex;
    add same process to ready list;
  }

  remove current process from ready list;
  move current process to wait list of event variable;
  schedule();
  ENABLE_INTERRUPTS;
}

void ev_broadcast(event *ev)
{
  DISABLE_INTERRUPTS;

  while(event variable wait list
        is not empty) {
    remove one process from wait list of event variable;
    add same process to wait list of mutex;
  }
  ENABLE_INTERRUPTS;
}
```