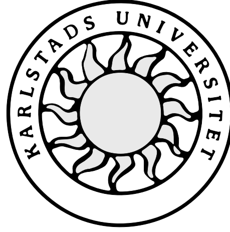



Datavetenskap

**Erika Lindblad**

# En jämförelse mellan realtidsoperativsystemen RUBUS och OSE

Examensarbete, C-nivå

2000:05

# En jämförelse mellan realtidsoperativsystemen RUBUS och OSE

**Erika Lindblad**

© 2000 Författaren och Karlstads universitet

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport som inte är mitt eget har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Erika Lindblad

Godkänd, 2000-02-09

Handledare: Stefan Lindskog

Examinator: Stefan Lindskog

## Sammanfattning

Denna rapport utgör ett 10-poängs examensarbete på C-nivå i ämnet datavetenskap, och ingår som en obligatorisk del i dataingenjörsutbildningen vid Karlstads universitet. Uppgiften var att jämföra realtidoperativsystemen Rubus och OSE, med syfte att ta reda på om OSE kan erbjuda en bättre laborationsmiljö än RUBUS i kursen Realtidssystem, 5 poäng (DAV C01). För att söka svar på den frågan har bland annat laborationsuppgiften i den aktuella kursen lösts med hjälp av de två systemen. Laborationen går i korthet ut på att konstruera en applikation som styr två elektriska modell-lok krockfritt. Rapporten innehåller i huvudsak en jämförelse av realtidsegenskaper som erbjuds i de två systemen.

# A Comparison of the Real-time Operating Systems RUBUS and OSE


## Abstract

This document contains a 10 points project work at C level necessary to obtain a Bachelor's degree in Computer Science at Karlstad University. The assignment was to compare two different real-time operating systems, RUBUS and OSE Delta, in order to find out whether the OSE operating system can offer a better development environment than RUBUS when doing the programming lab in the real-time system course, 5 points (DAV C01). The lab has been done in both operating systems. The lab consists of writing an application that controls two electrical trains in an eight/zero road avoiding crashes. The main purpose of this document is to compare the real-time behavior between OSE and RUBUS.

# Innehållsförteckning

**1 Inledning** ........ **1**

1.1 Bakgrund ........ 1

1.2 Syfte med arbetet ........ 1

1.3 Genomförande ........ 1

1.4 Disposition ........ 2

**2 Realtidssystem** ........ **2**

2.1 Realtidsoperativsystem ........ 3

2.1.1 Huvudtyper av RTOS

2.1.2 Funktionalitet i RTOS

2.2 Realtidsprogrammeringsspråk ........ 5

**3 RUBUS** ........ **5**

3.1 Processer ........ 6

3.2 Trådschemaläggning ........ 7

3.3 Trådkommunikation och synkronisering ........ 8

3.4 Felhantering ........ 8

3.5 Minneshantering ........ 9

3.6 Utvecklingsverktyg ........ 9

**4 OSE** ........ **9**

4.1 Processer ........ 10

4.2 Schemaläggning av processer ........ 11

4.3 Processkommunikation och synkronisering ........ 12

4.4 Felhantering ........ 13

4.5 Minneshantering ........ 13

4.6 Utvecklingsverktyg ........ 14

**5 Resultat** .......... **14**

**6 Rekommendation** .......... **15**

**7 Slutsats** .......... **16**

**Referenser** .......... **17**

**A Förkortningar** .......... **18**

# Figurförteckning

Figur 1. Systemarkitektur i RUBUS ........ 6

Figur 2. Exekveringsschema i RUBUS ........ 7

Figur 3. Systemarkitektur i OSE ........ 10

# 1 Inledning

Innehållet i denna rapport är resultatet av en jämförelse mellan realtidsoperativsystemen RUBUS och OSE. I det här kapitlet presenteras bakgrund, syfte och genomförande.

## 1.1 Bakgrund

På dataingenjörsprogrammet vid Karlstads universitet ingår kursen Realtidssystem (DAV C01) som en obligatorisk kurs. I kursen finns ett moment som går ut på att styra två elektriska modell-lok. Dessa tåg skall köra olika rutter. Den ena skall köra en så kallad ”noll-rutt” och det andra tåget en ”åtta”. Tågen skall kunna köra utan att krocka. Applikation konstrueras med hjälp av RUBUS, som är ett realtidoperativsystem med stöd för såväl tidsstyrda som händelsestyrda system. För att styra loken behöver man arbeta med tågadresser, hastigheter, växlar och sensorer.

## 1.2 Syfte med arbetet

Under ett par års tid har RUBUS används i kursen realtidssystem. Från kursutvärderingarna har det bland annat framkommit att laborationsuppgiften är bra, men att laborationsmiljön borde bytas. Syftet med det här arbetet är därför att undersöka huruvida realtidsoperativsystemet OSE skulle kunna ersätta RUBUS i kursen.

## 1.3 Genomförande

Arbetet inleddes med att ta reda på hur operativsystemet OSE fungerar, för att kunna jämföra det med RUBUS. Instudering av OSE tog en stort del av tiden i det inledande skedet. För att få en praktisk förståelse för hur systemet fungerar, installerade jag OSEs *soft kernel* på en standard-PC. För att få en känsla för OSEs funktionalitet och slutligen kunna realisera laborationsuppgiften utförde jag små programmeringsuppgifter där jag bland annat använde mig av OSEs olika processtyper och mekanismer för processynkronisering. Under hela arbetets gång har jag dokumenterat resultaten från experimenten för att kunna använda dem senare vid lösningen av laborationen.

Nästa steg var att lösa laborationsuppgiften. Här fick jag gå tillbaka och studera olika lösningsalternativ med RUBUS. En del svårigheter uppstod eftersom OSEs *soft kernel* inte har inbyggt stöd för att *accessa* serieporten, vilket behövdes för att kunna styra de elektriska loken. Problemen löstes genom att använda en del assemblerfunktioner som utvecklats av Niklas Rigemo och Per Lindqvist under kursen som genomfördes under hösten 1998. I min laborationslösning har jag lagt en hel del tid på att prova mig fram för att kunna få loken att fungera på ett så bra sätt som möjligt.

Rapportskrivning har pågått under hela arbetets gång och resultat som tillkommit med hjälp av övningarna med OSEs *soft kernel* ligger till grund för denna  jämförelser med RUBUS. De praktiska övningarna har varit av stor betydelse för att kunna uppnå de mål som hade satts upp innan arbetet påbörjades. Målet med arbetet kan sammanfattas i följande punkter:

- Beskriv och jämför RUBUS och OSE
- Lös laborationsuppgiften i kursen realtidsystem med hjälp av OSE
- Gör en jämförelse mellan lösningarna utförda med de två olika systemen

## 1.4 Disposition

I kapitel 2 beskrivs realtidssystem allmänt. Här redogör jag för olika typer av realtidssystem, vad, varför och vilka funktioner som bör finns i ett realtidsoperativsystem (RTOS), vilka specifika programspråk för realtidstillämpningar som finns, etc. I kapitel 3 och 4 beskrivs RUBUS respektive OSE.

Systemarkitektur, processer, processchemaläggning, minneshantering, felhantering, samt utvecklingsmiljöer beskrivs för de olika systemen. Resultat, rekommendation och slutsats presenteras i kapitel 5, 6 och 7. I bilaga A finns en lista över förkortningar som har använts i rapporten.

# 2 Realtidssystem

Realtidssystem blir allt vanligare i vår omvärld. De finns bland annat i telekommunikationssystem, processindustri, tillverkningsindustri, missiler, bilar, båtar, tåg och flygplan. Kännetecknande för sådana system är att de ska vara tillförlitliga, feltoleranta och kunna utföra sin

uppgift inom givna tidsramar. Dessutom ska de reagera på externt genererad indata inom given tidsperiod och den tid det tar för systemet att producera utdata är av stor betydelse.

Realtidssystem är med andra ord ett system vars korrekthet inte enbart beror av det logiska resultat av en beräkning, utan även av den tid då resultat är producerat. Ofta skiljer man mellan hårda och mjuka realtidssystem. Hårda realtidssystem är system i vilka en missad *deadline* (tiden inom vilken svar produceras) kan det leda till en katastrof [5]. I mjuka realtidsystem å andra sidan är svarstiderna viktiga, men enstaka missade *deadlines* kan tillåtas.

## 2.1 Realtidsoperativsystem

Realtidsoperativsystem (RTOS) kommer till användning då det är önskvärt att kunna köra flera processer samtidigt och där målet är att klara alla *deadlines*. Dessa system kan styra omgivningen genom att svara på externa händelser inom en viss tid. Ett RTOS erbjuder en virtuell maskin för applikationsprogrammen som önskvärt är hårdvaruoberoende [3].

### 2.1.1 Huvudtyper av RTOS

Realtidsoperativsystem kan i huvudsak delas in i två typer: händelsestyrda eller tidsstyrda.

- **Händelsestyrda RTOS.** I händelsestyrda RTOS, även kallade aperiodiska, tilldelas varje process en prioritet. Om flera processer vill exekvera, får den process som har högst prioritet köra. Om ingen process är redo, körs en så kallad *idle*-process. Denna process får inte göra systemanrop, eftersom den alltid ska vara redo. Med andra ord får den bara befinna sig i processtillstånden *ready* och *running* [9]. I händelsestyrda RTOS kan processer skapas i drift (eng. *run-time*).
- **Tidsstyrda RTOS.** Kallas även periodiska RTOS. Här exekverar processerna enligt ett förutbestämt schema. Det får till följd att processer inte kan skapas dynamiskt. Den här typen av RTOS är vanliga i system med hårda realtidskrav.

### 2.1.2 Funktionalitet i RTOS

En RTOS bör åtminstone kunna erbjuda följande funktionalitet:

- Processchemaläggning
- Processkommunikation och processynkronisering

- Minneshantering

#### 2.1.2.1 Processchemaläggning

Processchemaläggning går till på olika sätt i tids- respektive händelsestyrda RTOS.

- **Händelsestyrda RTOS.** Här görs schemaläggning enligt följande: *on-line*, det vill säga under exekvering, ofta med *Earliest Deadline First (EDF)* [4], [5] som schemaläggningsalgoritm. Högst prioritet tilldelas den process som har kortast tid till sin deadline.
- **Tidsstyrda RTOS.** I tidsstyrda system kan schemaläggning göras *off-line.* En vanlig schemaläggningsalgoritm är *Rate monotonic* (RM) [4], [5] där den mest frekventa processen ges högst prioritet.

#### 2.1.2.2 Processkommunikation och processynkronisering

Processkommunikation kan genomföras med delade variabler, som är globala för flera processer. I sådana fall är det viktigt att skrivning är atomär, det vill säga att en skrivning är odelbar. En annan form av processkommunikation sker med hjälp av meddelandeköer som fungerar som en sorts brevlådor. Realtidsoperativsystemet ansvarar för att brevlådan uppdateras atomärt. Här kan såväl lokala som globala brevlådor förekomma.

Några viktiga aspekter som bör beaktas är storleken på brevlådorna och i vilken ordning brev ska lagras. Andra sätt att kommunicera mellan processer är genom så kallade rendezvous:er, som betyder möte. Rendezvous:er används i bland annat programspråket Ada [2]. I Ada sker datautbytet mellan två processer synkront.

I system där flera processer ska använda delade resurser är det viktigt att ha mekanismer för att garantera ömsesidig uteslutning (eng. *mutual exclusion*), för att synkronisera användningen av resurser. Semaforer [9] kan användas för ändamålet. En annan mekanism som sköter synkronisering är monitorer. Monitorer [9] är en slags abstrakt mekanism som ser till att hantera delade resurser.

#### 2.1.2.3 Minneshantering

Minnet är en viktig resurs som kommer att delas mellan de olika processerna som ingår i en realtidsapplikation. Minnet ägs av RTOS:et. Ett RTOS bör åtminstone erbjuda funktionerna `alloc()` och `dealloc()`, som är funktioner som allokerar och frigör minnesplatser.

## 2.2 Realtidsprogrammeringsspråk

Behövs speciellkonstruerade programmeringsspråk för realtidstillämpningar? Nej, men det är bra om det åtminstone finns ett inbyggt processbegrepp i språket. Tyvärr har endast ett fåtal språk det. Om det inte finns realtidsstöd i språket måste det finnas i RTOS:et. Följande egenskaper är önskvärda i realtidsprogrammeringsspråk [10]:

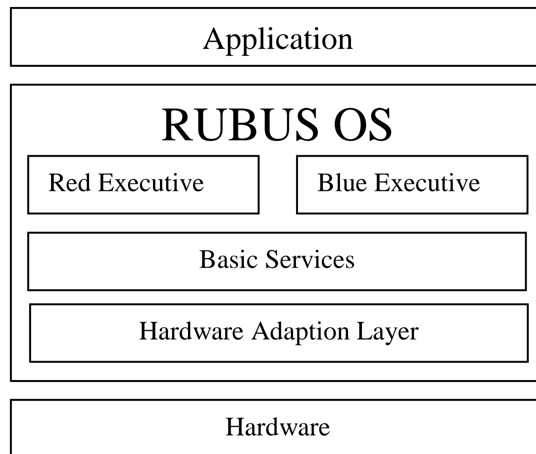
- Förutsägbarhet
- Robusthet och pålitlighet
- Modularitet
- Hårdvaruaccess
- Underhållbarhet
- Effektivitet

# 3 RUBUS

I det här kapitlet kommer RUBUS att beskrivas översiktligt. En detaljerad beskrivning av systemet finns i [7], [8]. RUBUS är en realtidsoperativsystem som tillhandahåller en plattform för utveckling av realtidsapplikationer. Dessa applikationer byggs upp med hjälp av så kallade delsystem. RUBUS erbjuder två olika typer av delsystem:

- röda delsystem
- blå delsystem

Denna indelning gör det möjligt att utveckla applikationer som kan klassificeras som antingen röda, blå eller bådadera. Det röda delsystemet är baserad på det tidsstyrda paradigmet. Det blå delsystemet, som är händelsestyrt, kan användas för att utveckla mindre tidskritiska system, som till exempel monitorering och operatörsgränssnitt. Det är även möjligt att utveckla applikationer med båda delsystem.




Figur 1. Systemarkitektur i RUBUS

I Figur 1 visas RUBUS relation till hårdvara och applikationer. Trådar som exekverar i det röda delsystemet kallas röda trådar. Applikationer som använder sig av sådana trådar kommer att ges högst prioritet. Det blå delsystemet ger stöd för händelsestyrd exekvering. I det blå delsystemet används en så kallad *preemptive scheduler* [9]. *Basic services* tillhandahåller gemensamma funktioner för det röda och blå delsystemet. Här finns bland annat funktioner för hantering av tid. *Hardware Adaption Layer* (HAL) är hårdvaruberoende och har till uppgift att bland annat ansvara för avbrottshantering. Tanken med HAL är att systemet ska bli lättare att portera mellan olika datorarkitekturer.

RUBUS stödjer ingen specifik programmeringsmodell utan användaren får själv välja den modell som är mest lämplig för ändamålet [7].

## 3.1 Processer

RUBUS har inget direkt stöd för processer, däremot erbjuds stöd för trådar (eng. *threads*). En applikation kan innehålla ett godtyckligt antal användardefinierade trådar. Som redan nämnts kan ett system bestå av enbart det röda delsystemet, enbart det blå delsystemet eller en kombination av rött och blått. Det finns därför två typer av användardefinierade trådar: blå och röda. Dessa hanteras av respektive delsystem.

Samtliga röda trådar som ska ingå i ett applikation måste vara fördefinierade. Det innebär att inga sådana trådar kan skapas under drift. Röda trådar definieras i en konfigurationsfil tillsammans med ett eller flera exekveringsscheman.

En händelsestyrd applikation konstrueras med hjälp av det blå delsystemet. Blå trådar som ska ingå i en applikation kan fördefinieras i en konfigurationsfil. Ytterligare blå trådar kan senare skapas dynamiskt i *run-time*.

## 3.2 Trådschemaläggning

Trådar i RUBUS schemaläggs olika beroende på om de är definierade som röda eller blå. Röda trådar följer ett i förväg specificerat schema, som har beskrivits i en konfigurationsfil. Ett schema består av periodtid, starttider (eng. *release time*), *deadlines* samt information om vilka trådar som ska exekvera. I Figur 2 visas ett exempel på två exekveringsscheman (redSchedule1 och redSchedule2) som skulle kunna finnas i en konfigurationsfil.

```
SCHEDULE redSchedule1  PERIODTIME 2000
  RELEASETIME 0
    REDTHREAD A ENTRY redAMain ARGS NULLP DEADLINE 8
  RELEASETIME 10
    REDTHREAD B ENTRY redBMain ARGS redBArg  DEADLINE 12;
SCHEDULE redSchedule2 PERIODTIME 50
  RELEASETIME 0
    REDTHREAD C ENTRY redCMain ARGS redCArg DEADLINE 1;
```

Figur 2. Exekveringsschema i RUBUS

När en applikation initieras i det röda delsystemet anges vilket exekveringsschemana som ska följas. Byte av schema kan sedan ske under drift. Det första schemat (redSchedule1) i Figur 2 har en periodtid på 2000 millisekunder. Två olika starttider finns också specificerade i schemat. Starttiden anger när en viss tråd tidigast får startas relativt periodstart. Det innebär att tråd A i Figur 2 kan startas vid periodens start och tråd B 10 sekunder efter periodens start. Notera att tråd C kommer endast att exekvera när det andra schemat (redSchedule2) är aktiverat.

Schemaläggning av trådar i det blå delsystemet görs under drift. Varje blå tråd har en prioritet (0-15, där 15 är den högsta prioriteten) och den tråden som har högst prioritet får exekvera först. Om en applikation utnyttjar både det röda och blå delsystemet, kommer samtliga röda trådar att exekvera före de blå.

## 3.3 Trådkommunikation och synkronisering

Kommunikation mellan två röda trådar eller mellan en röd och en blå tråd kan ske genom meddelandeköer, även kallade loggar. Dessutom kan delade variabler alltid användas för kommunikation mellan trådar.

Tre olika mekanismer för kommunikation och synkronisering mellan två blå trådar tillhandahålls av RUBUS.

- **Mutex**. En mutex kan liknas vid semaforer. I RUBUS har dessa implementerats med en så kallad prioritetsarvsmekanism (eng. *priority inheritance mechanism*) [4] för att minimera blockeringstiden för högprioriterade trådar. En mutex har en ägare. Detta gör att mutexen kan endas befrias av den tråd som har låst den.
- **Signals**. Med hjälp av signaler kan två trådar synkroniseras. En tråd som väntar på att en annan tråd ska slutföra en viss uppgift kan anropa funktionen `blueSigTimedWait()`. Anroparen kommer då att placeras i en väntekö till dess att någon ”väcker upp” den. När den andra tråden sedan är klar med uppgiften anropar den funktionen blueSigSend(), vilket får till följd att tråden som tidigare anropade `blueSigTimedWait()` ”väcks upp”. Den tråd som väntar på en signal kan specificera vilken signal den väntar på. Signaler specificeras i en bit mask.
- **Meddelandeköer**. När trådar vill utbyta information kan meddelandeköer användas. Meddelandeköer kan skapas dynamisk. En meddelandekö har ingen ägare utan alla trådar inom en applikation kan skicka och ta emot meddelande. En övre gräns på meddelandeköns storleken anges när den skapas. Dessutom anges vilken typ av objekt som ska kunna placeras i kön.

## 3.4 Felhantering

Både det röda och det blå delsystemet tillhandahåller felhanteringsmekanismer (eng. *error handler*). De har implementerats på följande sätt: om ett fel inträffar, så kommer en användardefinierad funktion med ett fördefinierat namn att anropas. Funktionerna ska heta `redError()` respektive `blueError()`. Den röda felhanteraren kommer exempelvis att anropas av den röda kärnan (eng. *kernel*) om en röd tråd skulle missa sin specificerade *deadline*.

## 3.5 Minneshantering

RUBUS tillhandahåller endast ett fåtal minneshanteringsfunktioner. Inget direkt stöd för allokering och återlämning av minnesutrymme erbjuds, förutom allokering av stackutrymme i de två delsystemen. I det röda delsystemet exekverar alla trådar med en gemensam stack, medan trådar i det blå delsystemet har varsin stack. Allokering och återlämning av minnesutrymme för dataareor måste därför tillhandahållas av programspråket C, som använts för att skriva applikationerna.

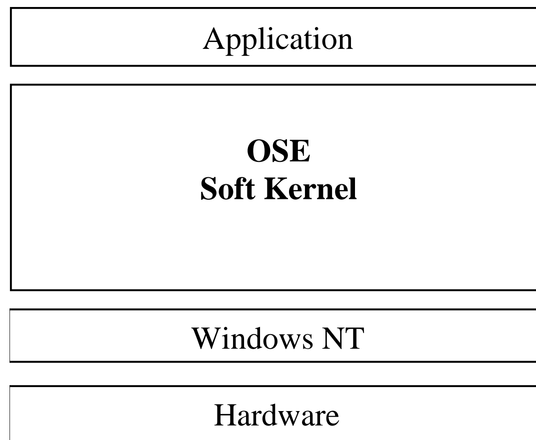
Ett rudimentärt minnesskydd har dock implementeras i meddelandeköerna. Vid skapandet av dessa får utvecklaren ange en övre gräns för hur många meddelanden som kan läggas i kön. Om kön blir full, kan kärnan bete sig på två olika sätt: antingen slänger kärnan nyinkomna meddelanden eller så läggs de först i kön. Önskat beteendet specificeras av utvecklaren.

## 3.6 Utvecklingsverktyg

Vid utveckling av applikationer har Borland C version 3.1 använts med tillhörande verktyg. Vidare levereras RUBUS med ett konfigurationsverktyg som kallas Bore. Bore har till uppgift att översätta konfigurationsfiler, se kapitel 3.2, till C-kod. I dessa filer har bland annat följande specificeras: klockupplösning, exekveringsscheman, röda och blå trådar, meddelandeköer, semaforer, etc.

# 4 OSE

Det finns två olika typer av kärnor i OSE-familjen. Den ena kallas OSE *Real-Time Kernel* (RTK) och är designat för inbäddade (eng. *embedded*) system. Den andra heter OSE *Soft Kernel* (SK) som är tänkt att användas i en värddator, exempelvis en standard PC. SK exekverar ovanpå ett konventionellt operativsystem, det vill säga UNIX, Windows 95/98 eller Windows NT. I de experiment som har genomförts under det här arbetet har endast en SK använts.



Figur 3. Systemarkitektur i OSE

Figur 3 illustrerar uppbyggnaden av ett system baserat på en SK och Windows NT som underliggande operativsystem. En sådan konfiguration ger större flexibilitet jämfört med om en RTK hade använts. Exempelvis får användaren (i det här fallet oftast en utvecklare) i en sådan konfiguration tillgång till en konsol, ett filsystem, nätverksfunktioner, etc.

Med hjälp av en programhanterare (eng. *program handler*) laddas, startas och stoppas OSE-applikationer. Denna kan dessutom tillhandahålla diverse programinformation. Vidare kan en systemdebugger knytas till OSE-kärnan, som ger möjlighet att kommunicera och ändra ett systems status under drift.

## 4.1 Processer

Processer är det viktigaste byggblocket i OSE. Det är genom processer som systemet fördelar CPU-tid mellan olika aktiviteter. I OSE finns stöd för såväl statiska som dynamiska processer.

- **Statiska processer.** De vanligaste processerna i små och medelstora system. Statiska processer initieras oftast av kärnan vid systemstart. Statiska processer existerar från det att de har initierats till dessa att applikationen har avslutats, vilket innebär att man normalt inte kan ta bort en statisk process.
- **Dynamiska processer.** Den här typen av processer kan skapas och tas bort under exekvering. En anledning till att det finns dynamiska processer är att med hjälp av dessa kan multipla instanser av samma kod köras. Antalet instanser behöver inte vara känt vid kompileringstillfället.

Förutom statiska och dynamiska processer finns det i OSE fem olika processtyper, som används för olika ändamål. Var och en av dessa beskrivs nedan.

- **Prioritetsprocesser.** Prioritetsprocesser är den vanligaste typen av processer. En exekverande prioritetsprocess kommer att låtas exekvera så länge ingen avbrottsprocess eller en annan prioritetsprocess med högre prioritet önskar exekvera.
- **Bakgrundsprocesser.** Bakgrundsprocesser exekverar i strikt tidsdelnings-*mode* [9]. De har alltid lägre prioritet än samtliga prioritetsprocesser. Operativsystemets tidsdelningsmekanism fördelar den överblivna CPU-tiden mellan bakgrundsprocesser som är reda att exekvera enligt en *First-Come-First-Served* (FCFS)-algoritm. Om en bakgrundsprocess inte lyckas slutföra sin beräkning inom sin tilldelade tidslucka, ställs den sist i kön och bakgrundsprocessen som står först i kön får tillgång till CPUn.
- **Avbrottsprocesser.** Anropas som ett resultat av antingen ett hårdvaruavbrott eller en programvaruhändelse. Processen kommer att exekvera från början till slut, om inte en annan avbrottsprocess med högre prioritet vill exekvera. Avbrottsprocesser är skrivna som subrutiner och har alltid högre prioritet än både prioritetsprocesser och bakgrundsprocesser.
- **Timeravbrottsprocesser.** Fungerar på exakt samma sätt som avbrottsprocesser med den skillnad att de aktiveras som ett resultat av att en timer har ändrat värde. Ett exempel på användning är att låta en sådan process aktiveras en gång per sekund.
- **Fantomprocesser.** Den här typen av processer innehåller ingen exekverbar kod och kan därför inte ta emot signaler. När en signal sänds till en sådan process skickas den vidare till en så kallad *redirection table*. Med hjälp av fantomprocesser kan logiska kanaler skapas. Detta speciellt användbart i en distribuerad miljö där samverkande processer kan finnas spridda på olika datorer.

## 4.2 Schemaläggning av processer

Principen för processchemaläggningen beror av processtyp. Dock gäller alltid att en högre prioriterad process kan tvinga en lägre prioriterade process att lämna ifrån sig CPUn. Denna princip kallas populärt för *preemption* [9]. Nedan redogörs för hur prioritets-, bakgrunds och timeravbrottsprocesser schemaläggs eller kan schemaläggas.

- **Prioritetsprocesser.** Processer av den här typen kommer att exekvera så länge inget avbrott inträffar och att ingen högre prioriterad process anländer till redokön samt att processen själv inte behöver vänta på en händelse av något slag. Processer med samma prioritet kommer att schemaläggas enligt en *round-robin*-algoritm [9].

- **Bakgrundsprocesser.** Round-robin-algoritmen används uteslutande för att schemalägga bakgrundsprocesser.
- **Timeravbrottsprocesser.** Timeravbrottsprocesser kan schemaläggs cykliskt, det vill säga att de kommer att aktiveras med jämna mellanrum. Detta kan vara användbart vid exempelvis pollning av sensorer.

## 4.3 Processkommunikation och synkronisering

I OSE finns olika sätt för kommunikation och synkronisering mellan processer. Nedan beskrivs signaler, semaforer, *fast* semaforer och miljövariabler.

- **Signaler.** En signal är ett meddelande som sänds från en process till en annan. För att kunna kommunicera med en annan process måste mottagande process' identiteter vara känd. Alla processer har en identitet. En process som ska skicka en signal måste själv allokera en minnesbuffert för signalen. Först måste signalnumret lagras i bufferten, och den egentliga datan kommer omedelbart därefter. När en mottagande process får tillgång till signalbufferten blir den ägaren till den. Endast den processen får utföra operationer på bufferten. En process kan själv ange vilka signaler den är intresserad av. I vissa fall måste signaler bearbetas av någon eller några andra processer innan avsedda mottagaren kan ta del av informationen. Processer måste i dessa fall ha en *redirection table*. En sådan tabell innehåller information om var en viss signal ska skickas vidare.
- ***Fast* semaforer.** Varje process har knutet till sig en så kallad *fast* semafor (FS). Dessa är tänkta att kunna underlätta vid synkronisering av processer och därmed uppnå ömsesidig uteslutning (eng. *mutual exclusion*). FS är betydligt snabbare vid synkronisering än signaler, men inte lika kraftfulla—de kan ju inte bära någon data. `wait_fsem` och `signal_fsem` är i huvudsak de operationer som kan användas på FS. Observera dock att avbrottsprocesser inte kan använda operationen `wait_fsem`.
- **Semaforer.** Semaforer i OSE fungerar ungefär på samma sätt som FS. Dock är inte semaforer knutna till en viss process. Alla processer förutom fantomprocesser och avbrottsprocesser kan utföra operationen *wait* på en semafor. Semaforer är tänkta att användas för att skydda kritiska sektioner i koden. De används ofta för att skydda delade resurser.

- **Miljövariabler.** Miljövariabler (eng. *environment variables*) är strängar som har ett namn och som knyts till ett block[1] eller en process. Dessa variabler kan initieras och modifieras under exekveringens gång. Med hjälp av miljövariabler kan processer konfigureras. De kan också användas för att sprida information mellan processer inom ett och samma block.

## 4.4 Felhantering

I OSE kan användardefinierade felhanterare (eng. *error handler*) definieras. Fördelen med att definiera felhanterare är att om ett fel detekteras och kan åtgärdas, så kan applikationen fortsätta sin exekvering. En felhanterare kan också användas för att städa upp efter det att ett fel har inträffat. Eventuellt kan också ett felmeddela skrivas ut som anger vad som har gått fel. Vid fel kan felhanteraren välja mellan omstart, återhämtning eller att avsluta applikationen.

Fel i OSE kan antingen vara fatala eller icke-fatala. Den senare typen kallas varning och inträffar när dataareor har blivit skadade, när en minnespool är full eller när en felaktig parameter har skickats till ett systemanrop. För många varningar kan vara ödesdigra och slutligen resultera i ett fatalt fel. Fatala fel är exempelvis när kärnans data har blivit skadat.

## 4.5 Minneshantering

OSE använder sig av en global minnespool som administreras av kärnan. Processer kan allokera minnesutrymme från denna. Problemet med en global minnespool är att hela systemet kommer att krascha vid fel. Som redan nämnts kan ett antal tillhörande processer grupperas i ett block. Varje block kan ha en egen lokal minnespool, som kan användas av de ingående processerna för allokering av minne. Vid fel i minnespoolen kommer det endast att påverka processerna inom blocket, det vill säga systemets sårbarhet minskar. Detta är mycket användbart i stora och komplexa system.

När signaler skickas mellan processer skickas endast en pekare till en signalbuffert tillhörande den sändande process. Det innebär att den process som får pekaren får tillgång till den sändande process minnesarea. Nackdelen med detta är att den mottagande processen kan förstöra den sändande processens minnesarea.

---

[1] Ett block i OSE är en mängd processer som tillhör samma processgrupp.

## 4.6 Utvecklingsverktyg

Vid utveckling av applikationer har vi använt oss av Visual C++ 5 med tillhörande verktyg. Ytterligare ett antal verktyg levereras med eller kan införskaffas för OSE. Bland annat finns en systemdebugger som kan användas för att bevaka signalkommunikationen mellan processer. Illuminator är ett annat verktyg för OSE som håller reda på alla aktiverade processers status. Vidare kan SKn betraktas som ett utvecklingsverktyg, eftersom riktiga realtidsapplikationer **inte** är avsedda att exekvera i en sådan miljö. När applikationen fungerar stabilt i SK är tanken att de ska flyttas till en miljö med en RTK.

# 5 Resultat

För att kunna utnyttja en operativsystem till fullo krävs att användaren har tid att sätta sig in hur den fungerar genom att läsa och experimentera. Det krävs att personen i frågan har mycket tid och tillgång till support. OSE kan verka en aning mer komplicerade en RUBUS eftersom det finns fler typer av processer att välja mellan. I både RUBUS och OSE är det möjligt att bygga upp system med begränsade kunskaper. Det är dock svår att förutse alla möjligheter och hur bra eller dåligt system blir eftersom det krävs att man testar systemet under en längre tid, vilket på grund av tidsbrist inte varit möjligt.

Det kan verka som att OSE är lättare att förstå, vilket kan bero på att jag hade gått kursen realtidssystem innan jag började arbeta med OSE och hade förkunskaper i ämnet. När vi implementerade applikationen i RUBUS hade jag inga förkunskaper i ämnet vilket gjorde det svårare att sätta sig in i hur det fungerar. Min uppfattning är att för att kunna jämföra OSE Soft Kernel och RUBUS på ett rättvisare sätt skulle det ha varit behövligt att arbeta parallellt med båda operativsystemen. Mina slutsatser vad det gäller RUBUS har baserat sig på kunskaper inhämtade vid tiden då jag gick realtidssystemkursen. Jag har kunnat studera OSE på ett mera ingående sätt och har provat mig fram på ett djupare sätt och tycker att jag har haft bättre kontroll över arbetet.

OSE Soft Kernel har lättlästa manualer och fler verktyg för att förenkla konstruktionen av applikationer. Vad beträffar laborationsuppgiften tyckte jag att det var lättare och gick snabbare att få de elektriska loken att köra krockfritt med OSE. Jag har på grund av tidsbrist gjort en väldigt enkel laborationslösning, som kan göras betydligt bättre om det går att få Illuminator att kunna följa processernas gång. Det var enklare att bygga applikationen med OSE eftersom man bara behövde koncentrera sig på synkronisering och processkommunikation.

Det var svårare att göra laborationen med RUBUS eftersom jag inte hade de kunskaper som jag har nu. Vad beträffar accessen till serieporten är det så att ingen av de båda stödjer direkt-access till denna. De problem som upplevdes med missade sensorvärden kan inte lösas genom att ha det ena eller det andra operativsystemet, utan beror snarare på att Märklins hårdvara är för långsam.

# 6 Rekommendation

Jag tycker att man borde prova att använda OSE Soft Kernel i realtidssystemskursen. Min erfarenhet är att det är enklare att sätta sig i OSEs händelsestyrda system, som dessutom är väl dokumenterade. Något som kan vara bra med OSE är att man kan ha det i en Windows NT-miljö vilket gör att man får tillgång till andra programvaror så som ordbehandling, Internet-uppkoppling, etc.

Något som är viktigt att poängtera är att oavsett vilket system som väljs så måste studenternas laborationsmiljö vara korrekt installerade från början. Dessutom behövs en bra beskrivning av miljön och vad som ska lösas i laborationen, det vill säga vilka filer som behövs, hur system länkas samman, etc. för att studenterna ska slippa att ”slösa” för mycket av den redan snålt tilltagna tiden på kursen.

1. Nedan följer en lista över de fördelar som jag har uppskatta mest i OSE.
2. OSE har utvecklats i Sverige, vilket har inneburit att det har varit lätt att få tillgång till professionell hjälp.
3. Bra debuggingmiljö, många trevliga verktyg som till exempel Illuminator erbjuds.
4. En Soft Kernel finns för Windows NT, vilket gör det möjligt att använda andra verktyg såsom Internet, ordbehandling, etc. samtidig som man arbetar med realtidslaborationerna.
5. Få systemanrop att hålla reda på.
6. Lättanvänd felhantering.
7. Bra prestanda.
8. Tillhandahåller minnesskydd
9. Stödjer Java VM.

Den största nackdelen är att kostnaden för en komplett utvecklingsmiljö är 5-6 gånger högre för OSE jämfört med RUBUS.

# 7 Slutsats

Från beskrivningarna i kapitel 3 och 4 framgår det att de två systemen erbjuder en tämligen snarlik funktonalitet. Dock känns OSE betydligt mer professionellt och har en rad trevliga och användbara utvecklingsverktyg, som förutom att vara till stor hjälp vid felsökning även kan vara ett pedagogiskt verktyg för att bättre förstå hur realtidssystem fungerar.

# Referenser

[1] Al Kelley and Ira Pohl. A Book on C, third edition. Benjamin Cummings, 1995.

[2] J. G. P. Barnes. Programming in Ada, 2nd edition. Addison-Wesley, 1984.

[3] J. E. Cooling, Software Design for Real-Time Systems, Thompson, 1996.

[4] Stefan Lindskog. Artikelkompendium, Realtidssytem, DAV C01. 1998-08-01.

[5] Stefan Lindskog. Föreläsningsanteckningar, Realtidssystem, DAV C01, ht 98.

[6] OSE Volume 1, Kernel Refence Manual.Document, No:420e/OSE93-1 R1.0.1.

[7] RUBUS OS Real-Time Operating System for Dependable Real-Time System, Version 1.3, Tutorial. Articus Systems, February 18 1997.

[8] RUBUS OS Real-Time Operating System for Dependable Real-Time Systems, Version 1.3, Reference Manual. Articus Systems, March 24 1997.

[9] A. Silberschatz and P. B. Galvin. Operating System Concepts, 4th edition. Addison-Wesley, 1994.

[10] Alexander D. Stoyenko, The evolution and State-of-the Art of Real-Time Languages. Journal of Systems and Software, pages 61-84, April 1992.

# A Förkortningar

| | |
|---|---|
| EDF | Earliest Deadline First |
| HAL | Hardware Adaption Layer |
| RM | Rate Monotonic |
| RTK | OSE Real-Time Kernel |
| RTOS | Real-Time Operating System |
| SK | OSE Soft Kernel |