

Examensarbete

**Kravanalys och implementering  
av ett realtidssystem  
med höga dataprestanda**

**Patrik Lantto**

LiTH-IDA-Ex-9709

1997-03-04

**PATRIK LANTTO**

# **Kravanalys och implementering av ett realtidssystem med höga dataprestanda**

Examensarbete utfört vid Linköpings Tekniska Högskola  
och OPQ Systems AB, Linköping.

Examinator: Anders Törne

Handledare: Olof Atterfors

# Sammanfattning

---

Denna rapport presenterar ett examensarbete utfört på OPQ Systems AB i Linköping. OPQ konstruerar system för mätning av ett antal olika parametrar på vägar. Beräkningarna görs i realtid, huvudsakligen på signalprocessorer.

Nya beräkningar med högre prestandakrav än dagens system kan tillgodose, och önskemål om att bättre kunna utnyttja gemensamma beräkningar för olika mätparametrar gör att OPQ undersöker hur ett nytt system skulle kunna konstrueras.

Syftet med examensarbetet är att undersöka vilka krav som bör ställas på ett nytt system, främst utgående från teorin om realtidssystem. Därefter föreslås lösningar som uppfyller kraven, och delar av systemet implementeras på en av OPQ utvald processor.

Resultatet visar att den utvalda processorns prestanda är fullt tillräckliga för att kunna ersätta dagens system och uppfylla nya önskemål, och att de implementerade lösningarna uppfyller de krav som ställts.

Även om en annan processor väljs bör ett nytt system utvecklas baserat på teorin om realtidssystem, och då är de förslagna lösningarna är en bra utgångspunkt.

# Abstract

---

This report presents a master degree thesis carried out at OPQ Systems AB in Linköping, Sweden. OPQ develops systems for measurement of a number of road surface parameters. The calculations are performed in real-time, primarily on digital signal processors.

New calculations with higher demands on performance than can be met by the current system, and the wish to be able to share calculations that are common for different parameters has led to OPQ's investigating how a new system could be designed.

The purpose of this thesis is to analyse the requirements for a new system, primarily based on the theory of real-time systems. Proposed solutions to fulfil the requirements are presented, and parts of the system are also implemented on a processor selected by OPQ.

The result shows that the chosen processor and the new system are fully capable of replacing the current system, and that the implemented solutions fulfil the presented requirements.

Even if another processor is chosen, a new system should be developed based on the real-time theory, and the proposed solutions are then an adequate basis for such development.

# Innehåll

---

*Sammanfattning iii*

*Abstract iv*

*Innehåll v*

*Figurer och tabeller viii*

**1 Inledning 1**

1.1 Bakgrund 1

1.2 Syfte 3

Realtidsegenskaper 3

Implementering 4

1.3 Avgränsningar och genomförande 4

1.4 Läsanvisningar 4

Formler 5

Språk 5

**2 Teoretisk referensram 6**

2.1 Samtidig exekvering 6

Processtillstånd 7

Klassificering 8

2.2 Realtidssystem 10

Tillgång till klocka 11

---

Fördröjning av processer	11
Maxtider	12
Tidsgränser	12
2.3 Restriktioner på realtidssystem	12
2.4 Schemaläggning av processer	13
Tidsattribut för processer	13
2.5 Statiska algoritmer	14
Cyclic executive	14
Preemptive preference priority-based scheduling	14
Rate-monotonic scheduling	15
Deadline monotonic scheduling	15
Kritiskt ögonblick	15
Optimal priority assignment	16
Tidsförluster i systemet	17
2.6 Dynamiska algoritmer	18
2.7 Schemaläggning av slacktid	18
<b>3 <i>Kravanalys</i></b>	<b>19</b>
3.1 Krav på processor	19
Datainläsning	19
Beräkningar på data	20
Applikationsspecifika beräkningar	21
Övrig processortid	22
Sammanfattning av processorkrav	22
3.2 Krav på schemaläggare	23
Tidsgränser	24
Uppstart av processer	24
Uppmätning av beräkningstid	24
Sammanfattning av systemkrav	25
<b>4 <i>Undersökning och design</i></b>	<b>26</b>
4.1 Undersökning av processor	26
Prestanda	27
Datainläsning	27
Övrigt	27
4.2 Schemaläggare	28
Algoritm	28
Schemaläggning av slacktid	29
Uppstart av processer	30
<b>5 <i>Systemimplementering</i></b>	<b>31</b>
5.1 Prioritetstilldelning	31
5.2 Realtidssystem	36

---

Aktivering av processer	36
Schemaläggare	37
Dispatcher	37
Uppstart av processer	38
Uppmätning av beräkningstid	38
<b>6 Resultat och slutsatser</b>	<b>40</b>
6.1 Resultat	40
Test av schemaläggning	42
Tidsåtgång i systemet	43
6.2 Slutsatser och rekommendationer	43
<b>A Programkod</b>	<b>45</b>
A.1 Optimal priority assignment	45
<b>Referenser</b>	<b>48</b>
<b>Index</b>	<b>50</b>

# Figurer och tabeller

---

## *Figurer*

Figur 1: Dataflödesmodell 3

Figur 2: Processövergångar 7

Figur 3: Tidsattribut för processer 14

Figur 4: Omvandling från tids- till spatialdomän 21

Figur 5: Tidsdiagram över exekvering av Audsleys processuppsättning  
42

Figur 6: Tidsdiagram över exekvering av fiktiva mätapplikationer 43

## *Tabeller*

Tabell 1: Processer från Audsleys rapport (tider i ms) 41

Tabell 2: Processer motsvarande mätapplikationer (tider i ms) 42

# 1 Inledning

---

Detta kapitel ger en bakgrund till examensarbetet, en presentation av företaget, en kommentar om avgränsningar och genomförande samt läsanvisningar för rapporten.

## 1.1 Bakgrund

Företaget, på vilket examensarbetet utförs, heter OPQ Systems AB och har i drygt 10 års tid sysslat med konstruktion av utrustning för mätning av olika parametrar på vägar. Mätutrustningen är monterad i en skåpbil, och huvudprincipen är att mätningen ska kunna utföras utan att störa övrig trafik, samt att behovet av efterbehandling ska vara så litet som möjligt. Detta ställer två krav på mätsystemet; dels måste det hantera hastigheter kring 90 km/h och dessutom visa upp realtidsegenskaper när det gäller att ta hand om de data som kommer in i systemet.

Mätutrustningen består av ett stort antal givare, t. ex. laserkameror för att mäta vägytan, accelerometrar, inklinometrar och gyro för att mäta hur bilen gungar, lutar respektive svänger, samt hjulpulsräknare för att hålla

reda på hastighet och distans. Data från givarna tas in till DSP-kort (processorkort som bygger på en DSP, *digital signal processor*) som i realtid utför beräkningar på data.

Exempel på mätapplikationer är textur som mäter vägens kornighet för olika våglängder, IRI (*international roughness index*) som mäter jämnhet enligt en internationellt fastställd standard, sprickmätning som räknar och kategoriserar olika spricktyper, längs- och tvärprofil för vägens lutning, samt vattendjup som beräknar hur mycket vatten som kan samlas i eventuella hjulspår.

För att kompensera bort bilens rörelser används olika metoder; texturapplikationen mäter på så korta våglängder att bilens gungningar kan filtreras bort med ett högpasfilter. För IRI däremot används en accelerometer för att kunna kompensera laservärdet mot bilens gungningar.

De DSP-kort som används är utvecklade av OPQ, och är baserade på två stycken TMS320C26-processorer från Texas Instruments. Då korten konstruerades 1989 var dessa processorer bland de snabbare på marknaden, och eftersom dessa hastigheter inte var tillräckliga för mer än en process på varje processor var det inte aktuellt att lägga ner utvecklingstid på ett operativsystem för processorerna. Istället utvecklades ett övergripande system, MCA, som styr alla DSP-kort, och dess funktion har vissa likheter med den funktion ett operativsystem har, dvs. meddelandehantering, dataprotokoll m. m.

Den databuss över vilken DSP-korten och MCA kommunicerar är tillräckligt snabb för överföring av färdigberäknade data från applikationerna till MCA, men där ligger också dess gräns. Även om vissa beräkningar är gemensamma för flera applikationer, räcker inte databussen till för att överföra resultat från en applikation till en annan. Detta innebär att vissa beräkningar måste utföras parallellt på flera olika processorer.

Utöver de parametrar som mäts idag finns även önskemål om nya mätparametrar med betydligt högre prestandakrav. Med dagens hårdvara kan många av dessa inte implementeras utan att dra ner på noggrannheten, eller genom att gå ifrån kraven på realtidsegenskaper och utföra efterbehandling av data.

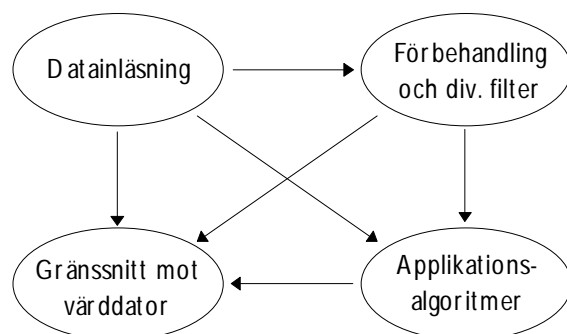
För att undvika redundanta beräkningar, och för att motsvara nya krav vill OPQ undersöka hur ett nytt system kan konstrueras.

## 1.2 Syfte

Bland de signalprocessorer som finns på marknaden idag finns många processorer som är flera gånger snabbare än de TMS320C26-processorer som används av OPQ idag. Ett byte skulle möjliggöra mer exakta implementeringar för mätning av parametrar med höga prestandakrav, men framför allt kan flera applikationer dela på en processor, och därigenom ges bättre förutsättningar att utnyttja gemensamma beräkningar.

Att låta flera applikationer utnyttja data från samma beräkning skulle inte bara spara processorkraft, utan också framtvinga tydligare avgränsningar mellan de olika beräkningsstegen i en applikation. Att på detta sätt införa en högre abstraktionsgrad kommer också medföra enklare testning och en tydligare koppling mellan algoritm och implementering. Detta kommer, förhoppningsvis, resultera i kortare utvecklingstid för nya applikationer.

En modell över dataflödet som är önskvärd att använda, men som idag inte är applicerbar, visas i figur 1. Varje delkomponent har ett tydligt gränssnitt mot andra komponenter; ett dataflöde in och ett ut. Genom att dela upp varje applikation i sådana delkomponenter kan beräkningar enkelt återanvändas genom att »ansluta« en komponent till önskat dataflöde.



Figur 1: Dataflödesmodell

### Realtidsegenskaper

De realtidsegenskaper som krävs under mätning har i dagens system analyserats genom att i varje applikation implementera olika grad av funktion för att upptäcka då tiden inte räcker till. Därefter har applikationerna provkörts för att undersöka om allting hinns med som det är tänkt, eller om delar av applikationen måste implementeras om för att bli snabbare. Så länge man bara har en process på varje processor är

denna analysmetod fullt tillräcklig.

I ett system med multiprocesshantering på processorerna ökar dock komplexiteten markant. Alla applikationer blir beroende av varandra, och undersökningar – både praktiska och teoretiska – måste göras med alla applikationer samtidigt.

Syftet med detta examensarbete är att utreda dessa problem, dvs.:

- att undersöka vilka krav som måste ställas på ett nytt system baserat på en snabbare processor som delas mellan flera applikationer;
- att utifrån vald algoritm ta fram analysmetoder för undersökning av systemets realtidsegenskaper;
- att implementera valda delar av systemet för att kunna utvärdera de olika lösningarna.

### Implementering

Implementeringen sker på en av OPQ redan utvald processer, ADSP-2106x SHARC från Analog Devices. Att visa att dess prestanda är tillräckliga ingår som en deluppgift i examensarbetet, och detta görs genom att jämföra med prestandakrav baserade på dagens system och önskade förbättringar.

## 1.3 Avgränsningar och genomförande

De avgränsningar som görs i examensarbetet gäller huvudsakligen implementering av de olika delarna i systemet. För att hålla ner omfattningen på en rimlig nivå implementeras bara de mest grundläggande delarna i schemaläggaren. Målet är att ändå kunna visa att ställda krav uppfylls genom att provköra systemet.

Systemet körs på ett utvärderingskort, EZ-LAB från Bittware Research Systems. Kortet är utrustat med en ADSP-21062, som är den mindre varianten av processorn (med 2 istället för 4 Mbit minne), och sitter som instickskort i en PC.

## 1.4 Läsanvisningar

Efter inledningen, som övergripande presenterar problemen och syftet med arbetet, följer ett kapitel med den teori som ligger till grund för hela arbetet. Kapitel 3 presenterar sedan problemen mer ingående utifrån den

presenterade teorin. De metoder som används i implementeringen beskrivs och motiveras i kapitel 4, och i kapitel 5 behandlas mer ingående själva implementeringen. Slutligen sammanställs resultat och slutsatser i kapitel 6.

Den programkod som producerats under arbetet ingår inte i sin helhet i rapporten, men de delar som diskuteras mer ingående ges även i sitt sammanhang i bilaga A.

## Formler

De formler som ges använder samma notation som övrig litteratur på området. Av dessa kan två skrivsätt behöva en förklaring:

- Intervall anges med hakparentes för ett slutet intervall och med vanlig parentes för ett öppet, t. ex. betyder  $[a, b)$  alla tal  $x$  med  $a \leq x < b$ .
- $\lceil a \rceil$  innebär det heltal  $b$  sådant att  $b \geq a$ .

## Språk

Även om dataområdet i allmänhet, och mer specifika delar som realtidssystem i synnerhet, har många engelska termer som ännu inte översatts, har jag valt att skriva denna rapport på svenska. I första hand för att visa att tekniska rapporter inte behöver skrivas på engelska bara för att terminologin är engelsk, men även för att kunna kombinera den stringens ämnet kräver med ett varierat och intressant språk som inte »tråkar ut« läsaren. Det senare har jag svårt att uppnå på annat språk än svenska.

De svenska termer som används är valda med omsorg; i förekommande fall utifrån befintlig litteratur, och i vissa fall med hjälp från TNC, Tekniska nomenklaturcentralen. Vid första användandet av en ny term ges även motsvarande engelska term inom parentes för att ge läsaren möjlighet att jämföra med engelsk litteratur på området. I index finns också hänvisningar från engelska termer till svenska termer.

Vissa namn på algoritmer har fått behålla sina engelska namn, trots att de ofta är en beskrivning av algoritmen som skulle kunna översättas. Namnen är dock så väl vedertagna på området att man snarare kan se dem som egennamn än beskrivningar. Dessa namn är satta med kursivt typsnitt.

## 2 Teoretisk referensram

---

Grunden i realtidssystem är samtidig exekvering av processer (*concurrent execution*). Något förenklat finns egentligen bara en skillnad mellan ett generellt operativsystem med samtidig exekvering och ett realtidssystem – tidsbegreppet.

I det här kapitlet ges en teoretisk bakgrund till projektet genom att förklara begreppen samtidig exekvering samt dess utvidgning till realtidssystem. Beskrivningen av samtidig exekvering finns med som bakgrund till genomgången av realtidssystem. Utöver detta ges en genomgång av några utvalda algoritmer för schemaläggning av processer.

### 2.1 Samtidig exekvering

*Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems.*

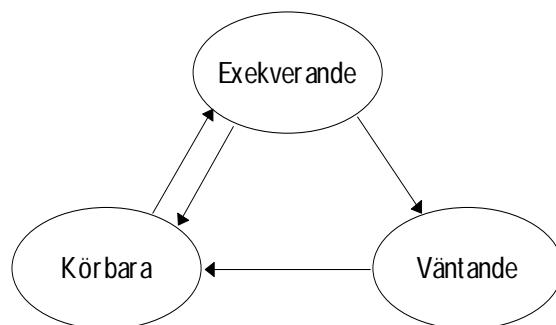
(Ben-Ari, 1982)

Samtidig exekvering innebär – som citatet ovan antyder – inte nödvändigtvis att flera saker utförs samtidigt, men att möjligheten finns att de gör det. Grunden i samtidig exekvering är processer, som kan ses som sekventiella program. Beroende på hur samtidig exekvering implementeras kommer en mängd processer att exekveras, antingen genom att turas om på en processor, eller genom att en eller flera processer exekveras på olika processorer i ett multiprocessorsystem. Bara i det senare fallet finns möjligheten till äkta parallell exekvering av två processer.

De olika algoritmer som finns för att låta flera processer dela på en processor, dvs. schemaläggning (*scheduling*), tas inte upp i det allmänna fallet. Istället behandlas algoritmer som tar hänsyn till tid i avsnitt 2.4.

### Processtillstånd

Mängden av processer som exekveras i ett multiprocessorsystem brukar delas in i ett antal delmängder som betecknar det tillstånd processen är i; exekverande (*running*), körbara (*ready*), och väntande (*waiting*). De exekverande processerna är de som för stunden exekveras på processorerna i systemet, dvs. högst en för varje processor. De körbara processerna är de som är färdiga att exekvera på processorn när den blir ledig, och de väntande är de som väntar på någon intern eller extern händelse, t.ex. meddelande från annan process eller någon användarstyrd händelse. Övergångarna mellan de olika mängderna visas i figur 2.



Figur 2: Processövergångar

Den streckade pilen mellan de exekverande processerna och de körbara beror på hur schemaläggningen implementeras. Bara om en process kan avbrytas av schemaläggaren innan processen är färdig, s.k. *preemptive scheduling*, kan en exekverande process övergå till en körbar.

## Klassificering

När man ska konstruera ett system med samtidig exekvering finns det ett antal faktorer som påverkar hur systemet implementeras (Burns och Wellings, 1990):

- struktur (*structure*);
- nivå (*level*);
- granularitet (*granularity*);
- initialisering (*initialization*);
- terminering (*termination*);
- representation (*representation*).

*Strukturen* för ett system är antingen statisk eller dynamisk. Statisk struktur innebär att systemet har ett fixt antal processer som är bestämt vid initialisering. Med dynamisk struktur kan processer skapas när som helst vilket innebär att antalet processer endast är känt under exekvering.

Nästa faktor påverkar inte systemet i sig så mycket, utan mer programspråket: ett systems *nivå* anger var processer är definierade. Om processer kan definieras i alla nivåer av systemet säger man att systemet har nästadsnivå. Motsatsen, då processer bara kan definieras i systemets översta nivå kallas plan nivå. Nästadsnivå innebär t.ex. att processer kan skapas som underprocesser i andra processer och dela funktioner och variabler.

Med *granularitet* karaktäriseras antalet processer, samt deras livslängd. Om det är få processer med lång livslängd har systemet grov granularitet, med många processer som bara utför några få saker sägs granulariteten vara fin. Ett exempel på fin granularitet är konstruktionen `cobegin` i programspråket Edison<sup>1</sup>. Beräkningarna av `X` och `Y` kommer att utföras parallellt i nedanstående program:

```
cobegin
  X := sin(V);
  Y := sin(V);
coend
```

*Initialisering* av en ny process kan ske antingen genom att skicka med parametrar till processen, eller genom att kommunicera med processen när den har startats upp.

---

1. Ett programspråk avsett för inbyggda system (*embedded systems*) med flera processorer (Burns och Wellings, 1990).

*Terminering* av processer kan ske:

- efter den sista raden i processens program;
- genom att processen »begår självmord«, dvs. anropar någon form av avslutningsfunktion;
- genom att en process terminerar en annan process;
- vid fel som inte tas om hand av processen, t. ex. ogiltig instruktion;
- aldrig, processen består av en icke-terminerande loop;
- när processen inte längre behövs.

Den enklaste typen av terminering att implementera i ett system är naturligtvis att förutsätta att processer aldrig avslutas. Utöver det blir antalet implementerade varianter en kompromiss mellan systemkraven och implementeringens komplexitet. Att låta en process avsluta en annan ställer t. ex. krav på att systemet kan frigöra de resurser som allokerats av den process som termineras.

Den sista egenskapen är *representation* – hur samtidig exekvering representeras i programspråket. Det finns tre grundläggande metoder: *fork och join*, *cobegin*, och explicit deklaration av processer.

Den enklaste varianten att implementera är *fork och join*, då den inte kräver någon speciell programspråkskonstruktion, utan kan implementeras som funktionsanrop. I nedanstående exempel i programspråket Mesa<sup>2</sup> kommer den del av programmet som ligger mellan *fork* och *join* att exekveras parallellt med funktionen *F*. Värdet som returneras av *fork* är en identifierare för den nya process som skapas. Denna identifierare ges sedan som argument till *join* som väntar på att den identifierade processen terminerar.

```
function F return ... ;

procedure P;
  ...
  C := fork F;
  .
  .
  J := join C;
  ...
end P;
```

---

2. Ett programspråk som används av Xerox i deras kontorsmaskiner (Burns och Wellings, 1990).

Den andra varianten, *cobegin* (se exempel på fin granularitet ovan), ger stöd för att uttrycka t.ex. parallella algoritmer på ett tydligt sätt. Implementeringsmässigt kräver den dock både stöd i programspråket och en systemkonstruktion som kan hantera fin granularitet.

Den sista grundläggande varianten är *explicit deklaration* av processer. Med *fork och join*, och *cobegin* exekveras en procedur parallellt om den anropas med *fork* respektive om den anropas mellan *cobegin* och *coend*. Med *explicit deklaration* är det istället proceduren själv som anger att den ska utföras parallellt. Ett anrop till en sådan procedur kommer att skapa en ny process i vilken proceduren utförs. Följande exempel i Modula-1 skapar två processer:

```
MODULE main;
  PROCESS control(pos : integer);
  BEGIN
    ...
  END control;

BEGIN
  control(0);
  control(1);
END main.
```

## 2.2 Realtidssystem

*... any information processing activity or system which has to respond to externally-generated stimuli within a finite and specified period.*

(Young, 1982)

I Youngs definition av realtidssystem hittar man två grundläggande kännetecken för ett realtidssystem: systemet styrs av externa händelser och svarstidens längd är av betydelse. För att ett samtidigt exekverande system även ska uppfylla definitionen för ett realtidssystem måste alltså begreppet tid införas, vilket kan beskrivas med följande krav (Burns och Wellings, 1990):

- Tillgång till en klocka för att kunna mäta tidsintervall.
- Möjlighet att fördröja en process en bestämd tid.
- Programmerbara maxtider (*time-outs*) för att kunna upptäcka och

- hantera när en händelse inte sker inom ett definierat tidsintervall.
- Möjlighet att specificera tidsgränser (*deadlines*) för processer samt en schemaläggare som tar hänsyn till dessa.

De fyra kraven bygger på varandra, varför alla måste implementeras i ett system. Däremot behöver inte alla implementeras så att de är tillgängliga på processnivå. T.ex. för ett system med enbart periodiska processer (se avsnitt 2.4) behöver inte de två första kraven vara tillgängliga för processerna.

### Tillgång till klocka

Att implementera en klocka kräver naturligtvis i första hand en realtidsklocka i hårdvara. Den enklaste varianten är processornas interna klocka. I de fall då det finns två klockor i processorn kan en av dessa helt avsättas till att hålla reda på tiden. I annat fall krävs viss eftertanke då samma klocka troligen även måste användas till schemaläggaren.

En andra variant är att använda sig av en extern klocka, och lägga in stöd i systemet för att kommunicera med den. En nackdel med extern klocka är dock att man gör sig beroende av ytterligare hårdvara.

När man bestämt sig för vilken hårdvaruklocka man ska använda sig av återstår att bestämma sig för hur tiden ska representeras för processerna. Noggrannheten, dvs. minsta tidsenheten, får vägas mot hur lång tid som maximalt ska kunna representeras, samt hur många bitar man använder för värdet. Om tiden representeras med t.ex. ett 32-bitars tal och en minsta tidsenhet på 1  $\mu$ s kommer den maximala tiden som kan representeras att vara drygt 70 minuter.

### Fördröjning av processer

För att förhindra att processer fördröjer sig själva genom att ligga i en loop och vänta på att klockräknaren får ett visst värde (s.k. *busy-wait*) måste stöd för fördröjning implementeras i systemet så att en annan process kan exekveras den tid som den första väntar. Viktigt att notera är att fördröjningen kan bli längre, men inte kortare, än önskad tid beroende på andra processers prioritet. Detta är dock ingen skillnad mot *busy-wait* eftersom processer med högre prioritet även i det fallet kan avbryta exekveringen strax innan klockräknaren slår om till det värde som inväntades.

### Maxtider

Om en process ska kunna upptäcka avsaknaden av en extern händelse, t.ex. för att upptäcka om en givare inte ger några mätvärden, måste de funktioner som väntar på en händelse utökas med en maxtid. En funktion som väntar på en angiven extern händelse skulle då utökas med ett argument för maxtiden, och indikerar genom returvärdet om någon händelse inträffade, eller om maxtiden passerades. Även här är det viktigt att notera att den tid det tar innan funktionen returnerar kan vara längre än den maxtid som specificerades. Däremot bör den tid med vilken händelsen jämförs vara den som angavs. Detta innebär att en händelse kan ha inträffat även då funktionen returnerar för att tiden tog slut, men i så fall efter maxtiden.

### Tidsgränser

De tre krav som beskrivits ovan ger en process möjligheten att upptäcka om en beräkning tagit för lång tid, eller om en extern händelse inte inträffat innan en specificerad maxtid. Även om en process utför en viss beräkning tillräckligt snabbt finns det inget som garanterar att processen får tillgång till processorn så lång tid som den behöver, eller att den får det i tid. Därför måste processer även kunna specificera hur lång tid de behöver exekvera samt när beräkningen måste vara färdig. Att utifrån dessa specifikationer schemalägga processerna så att önskemålen, i den mån det är möjligt, uppfylls, samt att på något bestämt sätt hantera de fall när tiden inte räcker till är den viktigaste funktionen i ett realtidssystem. Då forskningen på området fortfarande är relativt ung finns det ingen självklar lösning av problemet. Ett antal, mer eller mindre olika algoritmer finns presenterade i forskarrapporter (se avsnitt 2.4 nedan). Utifrån systemkraven får man välja någon lämplig algoritm och sedan göra eventuella anpassningar.

## 2.3 Restriktioner på realtidssystem

Om man ska kunna förutsäga ett realtidssystem's egenskaper krävs det att de tider som är specificerade för varje process är så noggranna som möjligt. Anger man för snäva tidsgränser kan ett system fungera på papperet, men inte i verkligheten. Om man istället anger för höga värden kan det resultera i låg nyttjandegrad (andel processortid som används av processer).

För att minimera skillnaden mellan det högsta och lägsta värdet (i vissa

fall för att överhuvudtaget kunna räkna ut ett högsta värde) måste vissa system- och programspråkskonstruktioner undvikas. Dessa är bl. a. dynamisk allokering av minne, dynamisk processtruktur (se avsnitt 2.1 ovan), och rekursion. Man kan dock tänka sig att tillåta dessa om de används under en initialiseringsfas innan tidsgränser har specificerats.

## 2.4 Schemaläggning av processer

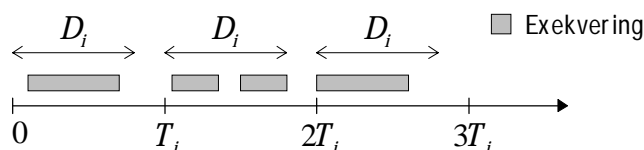
Vid schemaläggning i ett realtidssystem brukar man dela in processerna i två grundtyper: periodiska och aperiodiska processer. Exempel på periodiska processer är processer som samlar in data, och som måste hämta nya värden från givare med jämna mellanrum. Typiskt för aperiodiska processer är att deras exekvering är beroende av externa händelser, som t. ex. interaktion med användare. I det generella fallet görs inga specifikationer om hur ofta en aperiodisk process aktiveras. Detta medför att man inte kan göra något antagande om processens maximala krav på beräkningstid, varför man ofta brukar definiera en minimitid mellan två aktiveringar av en aperiodisk process. För att skilja dessa processer från generella aperiodiska kallas de för sporadiska processer (Burns och Wellings, 1990).

Ytterligare en distinktion för processer är hur viktigt det är att den klarar sina tidsgränser. Även om gränsen är flytande brukar man dela in processerna i hårda och mjuka processer. En hård process kan inte missa någon tidsgräns, då misslyckas processen. Om systemet används i t. ex. ett flygplan skulle en missad tidsgräns kunna vara katastrofal. En mjuk process kan däremot hantera en sådan situation; om datainläsning görs från en buffert förloras inte data så länge man bara missar tidsgränsen med kort tid. Det som gör gränsen flytande är att man även talar om feltoleranta processer, där en process klarar en missad tidsgräns förutsatt att en speciell rutin anropas av systemet. Även dessa brukar dock benämnas som hårda processer.

### Tidsattribut för processer

Som grund för schemaläggningen används ett antal attribut för varje process  $\tau_i$ . Viss variation förekommer beroende på algoritm, men de grundläggande är periodtid (*period*), tidsgräns (*deadline*) och beräkningstid (*computation time*). Periodtiden  $T_i$  betecknar den önskvärda cykeltiden för aktivering av process  $\tau_i$ ; tidsgränsen  $D_i$  är den tidpunkt relativt aktivering då  $\tau_i$  måste vara färdig; beräkningstid  $C_i$ , är hur mycket processortid  $\tau_i$  behöver i värsta fall. För att klara av tidsgränsen krävs

alltså tillgång till processorn i  $C_i$  tidsenheter under alla intervall  $[nT_i, nT_i + D_i]$  (se figur 3).



Figur 3: Tidsattribut för processer

## 2.5 Statiska algoritmer

Statiska algoritmer för schemaläggning kännetecknas av att schemaläggningen huvudsakligen görs *off-line*, t.ex. under kompilering av programmet, externt innan systemet startas upp, eller i systemet innan processerna börjar exekvera.

Motsatsen, då den större delen av schemaläggning görs samtidigt som systemet exekverar, kallas för dynamiska algoritmer. Det är dock ingen tydlig gräns dem emellan; en större eller mindre del av schemaläggningen kan göras statiskt respektive dynamiskt.

De algoritmer som brukar kallas statiska utgår från ett fixt antal processer utifrån vilka en tur- eller prioritetsordning bestäms. Denna ordning används sedan av den, mycket enkla, dynamiska del som exekveras på processorn.

### Cyclic executive

Den mest statiska metoden är att vid kompilering bestämma turordningen för alla processer, och sedan implementera dem som ett enda program, *cyclic executive*. Alla processer utförs sekventiellt, och vid behov väntar man in en viss tidpunkt innan programmet fortsätter. Vanligtvis används en tabell med olika tidpunkter, och den funktion som ska anropas.

Nackdelen med en *cyclic executive* är att den kan bli svår att överblicka, och införa förändringar i. Den används därför mest i mindre system.

### Preemptive preference priority-based scheduling

En algoritm som behåller processbegreppet som enskilda program, men som fortfarande är relativt enkel att implementera är *preemptive preference priority-based scheduling*. Den körbara process som har högst prioritet får exekvera på processorn, och om en annan process med högre prioritet än den exekverande blir körbar, avbryts den första processen.

Algoritmen är enkel att överblicka eftersom processerna implementeras som enskilda delar, men dess predikterbarhet vid hög belastning är otillräcklig i de flesta fall; den enda process som med säkerhet klarar sina tidsgränser är den som har högst prioritet (under förutsättning att processorns prestanda är tillräckliga). Det har visats att algoritmen kan misslyckas med att schemalägga en mängd processer vid en nyttjandegrad så låg som 60 procent (Burns och Wellings, 1990).

### Rate-monotonic scheduling

År 1973 presenterade Liu och Layland en algoritm som gav förbättrade egenskaper för *preemptive preference priority-based scheduling* som kallas *rate-monotonic scheduling* (Liu och Layland, 1973). I stället för att sätta processernas prioritet utifrån programmerarens önskemål sätts de omvänt mot periodtiden  $T_i$ ; kortare period ger högre prioritet. Genom denna enkla förändring ökas den undre gränsen för nyttjandegraden så att alla mängder av processer med nyttjandegrad lägre än 69 procent kan schemaläggas. Algoritmen är optimal i hänseendet att om en statisk algoritm kan schemalägga en mängd processer kan man göra det med *rate-monotonic scheduling*. För att det ska gälla måste dock ett antal krav uppfyllas, bl.a. att det ska vara ett fixt antal processer, att samtliga ska vara periodiska med  $D_i = T_i$ , dvs. tidsgräns satt till perioden, och att alla processer aktiveras vid tidpunkt noll.

### Deadline monotonic scheduling

En annan algoritm, som även hanterar fall där  $D_i < T_i$  är *deadline monotonic scheduling* (Audsley, 1990). I den sätts prioriteten omvänt mot tidsgränsen  $D_i$  istället för  $T_i$ . Även denna algoritm är optimal med samma krav som *rate-monotonic scheduling* med undantaget att även  $D_i < T_i$  tillåts (för  $D_i = T_i$  blir algoritmen identisk med *rate-monotonic scheduling*).

Vidare utveckling av de statiska algoritmerna har lättat även på andra krav. I en sammanställning om forskningen kring statisk schemaläggning (Burns, 1995) ges översiktliga beskrivningar, samt hänvisningar till förbättringar. Bl.a. tillåts både hårda och mjuka aperiodiska processer, variationer av beräkningstid,  $C_i$  och  $T_i$  och fall där  $D_i > T_i$ .

### Kritiskt ögonblick

Värsta fallet vid schemaläggning är då alla processer aktiveras samtidigt, s.k. kritiskt ögonblick (*critical instant*). Om samtliga processer aktiveras vid tidpunkt noll (enligt krav för *rate-monotonic scheduling*) kommer

kritiska ögonblick att inträffa för varje tidpunkt  $t = m \cdot lcm(T_1, T_2, \dots, T_n)$  där  $lcm$  är minsta gemensamma multipel (*least common multiple*). För att bevisa att en uppsättning processer kan schemaläggas räcker det därför med att visa att ingen process missar sin tidsgräns efter aktivering vid ett kritiskt ögonblick, dvs. simulera systemet för  $t = [0, P)$  där  $P = \max(D_1, D_2, \dots, D_n)$ .

### Optimal priority assignment

Ett krav för att både *rate-monotonic scheduling* och *deadline monotonic scheduling* ska vara optimala är att alla processer aktiveras vid tidpunkt noll. Det finns dock tillfällen då det är önskvärt att vänta med aktiveringen av en process, t.ex. för att vänta in data som beräknas av en annan process. Därför har man infört ett attribut för fördröjning (*offset*),  $O_i$ .

Med införandet av fördröjning är det inte längre säkert att kritiska ögonblick inträffar, varför beviset för att en mängd processer kan schemaläggas försvåras. Dessutom gäller alltså inte optimaliteten för *deadline monotonic scheduling* och *rate-monotonic scheduling*. Som lösning på dessa problem har Audsley (1991) presenterat *optimal priority assignment*, en algoritm som är optimal även för  $O_i \neq 0$ .

Utgångspunkten för algoritmen är två satser. Om  $\Delta^*$  är en mängd av  $n$  processer, och  $\Phi_X$  en funktion för prioritetstilldelning sådan att  $\Phi_X(\tau_A) = n$  (observera att lägre prioritet, dvs. mindre viktig, motsvaras av ett högre värde), så gäller följande satser:

- Sats 1: Om  $\tau_A$  har lägsta prioritet,  $n$ , och inte klarar sina tidsgränser, finns ingen prioritetsordning med  $\Phi(\tau_A) = n$  för vilken  $\Delta^*$  kan schemaläggas utan att missa någon tidsgräns.
- Sats 2: Om  $\tau_A$  har lägsta prioriteten,  $n$ , och klarar sina tidsgränser gäller att om någon prioritetsordning för  $\Delta^*$  existerar för vilken  $\Delta^*$  kan schemaläggas utan att missa någon tidsgräns, så finns en prioritetsordning där  $\tau_A$  tilldelas lägsta prioritet.

Dessa satser gör att en algoritm inte behöver prova alla prioritetsordningar, utan kan titta på en prioritetsnivå i taget, den lägsta. Om man hittar en process som klarar sina tidsgränser för lägsta prioritet  $n$ , med godtycklig ordning på de  $n-1$  övriga processerna, kan den tilldelas prioritet  $n$ . Därefter tittar man på prioritet  $n-1$ , och då behöver de processer som redan är tilldelade prioritet inte räknas med. Om ingen process hittas för någon nivå kan algoritmen avbrytas eftersom ingen prioritetsordning då existerar.

För att hantera fördröjningar måste systemet simuleras längre tid än för  $t = [0, P)$  enligt ovan. Audsley visar att en tillräcklig period är  $t = [S_i, S_i + P_i)$  där  $S_i$  och  $P_i$  ges av:

$$S_i = \left\lceil \frac{O_{max} - O_i}{T_i} \right\rceil T_i = \left\lceil \frac{O_{max}}{T_i} \right\rceil T_i$$

där  $O_{max} = (O_1, O_2, \dots, O_{i-1})$   
och under antagandet att  $O_i = 0$   
 $P_i = lcm(T_1, T_2, \dots, T_i)$

Antagandet att  $O_i = 0$  kan sättas genom omfördela fördröjningstiderna. En noggrannare beskrivning av hur detta görs tillsammans med övriga delar i *optimal priority assignment* presenteras i avsnitt 5.1.

### Tidsförluster i systemet

De algoritmer som nämnts har alla utgått från varje process tidsattribut vid schemaläggning. När det är dags att köra systemet tillkommer dock processortid för schemaläggaren och systemet i sig, t.ex. byte av exekverande process och hantering av avbrott för externa händelser. I simuleringar för att undersöka om en prioritetstilldelning klarar alla tidsgränser brukar tiden för byte av process tillföras den process man byter till.

För *optimal priority assignment* blir det lite annorlunda eftersom man vid simulering bara undersöker tidsgränsen för den process som har lägst prioritet. Prioritetsordningen på de övriga ska då, enligt sats 1 inte spela någon roll. För att kunna lägga till tiden för byte krävs dock att man känner till antalet byten, och det kan variera beroende på ordningen på de övriga processerna. Om sats 1 fortfarande ska gälla måste man beräkna det största antalet byten genom att testa alla kombinationer, och då har man förlorat vitsen med algoritmen. Lösningen är att istället utöka beräkningstiden med både byte till ny process och byte tillbaka till föregående process. Mängden tillförd tid för schemaläggning blir då densamma oberoende av processernas prioritetsordning, och *optimal priority assignment* kan användas oförändrad.

Den tid som går åt för hantering av avbrott är normalt mycket liten, men om inte så är fallet kan tiden modelleras som en sporadisk process, dvs. en process där  $T_i$  är satt till den minsta tiden mellan två aktiveringar, och där  $C_i$  är tiden för att hantera avbrottet.

## 2.6 Dynamiska algoritmer

I dynamiska algoritmer görs schemalaggningsen samtidigt som systemet exekverar genom att för varje tidpunkt bestämma vilken process som ska exekveras närmast. Den stora fördelen är att aperiodiska processer hanteras lika bra som periodiska; nackdelen är att algoritmen inte tittar nog långt framåt i tiden för att veta om mängden processer klarar alla tidsgränser eller inte. Den gör det den anser är bäst vid varje tidpunkt, varför algoritmer av den här typen även kallas *best-effort scheduling*.

De två vanligaste dynamiska algoritmerna är *earliest deadline first* (EDF) och *least slack time*. I EDF exekveras den process vars tidsgräns ligger närmast i tiden; i *least slack time* den process som har minst slacktid kvar, dvs. tid kvar innan processen måste börja exekvera för att hinna färdigt innan sin tidsgräns.

Nästan alla kommersiella realtidssystem bygger idag på statiska algoritmer, vilket främst beror på de dynamiska algoritmernas oförmåga att nog långt i förväg förutsäga överbelastning. Mycket forskning bedrivs dock för att förbättra predikterbarheten i de dynamiska algoritmerna, främst utgående från EDF.

## 2.7 Schemalaggnings av slacktid

När alla hårda processer har fått den processortid de behöver kan det finnas tid över innan nästa process ska påbörja exekvering, s.k. slacktid. Denna tid kan utnyttjas för t.ex. exekvering av mjuka aperiodiska processer eller olika beräkningar för att förbättra schemalaggningsen av de övriga processerna, *imprecise computations* (Burns, 1995).

För att exekvera processer i slacktiden finns ett antal metoder. Den som är enklast att implementera kräver ingen förändring av schemalaggningsaren; alla mjuka processer får helt enkelt lägre prioritet än de hårda processerna. Mer avancerade metoder som t.ex. håller reda på hur mycket slacktid som finns vid varje tidpunkt har presenterats av bl.a. Lehoczky och Ramos-Thuel (1995).

## 3 Kravanalys

---

I detta kapitel presenteras de krav som ställs på systemet utgående från dagens system och önskade förbättringar. Kraven på schemaläggaren specificeras utifrån den teori som presenterats i kapitel 2, och några av de problem som kan tänkas uppkomma vid designbeslut identifieras.

### 3.1 Krav på processor

De krav som ställs på den processor som ska användas baseras främst på den mängd data som ska tas om hand samt komplexiteten på de beräkningar som ska utföras. Kommunikation mellan processer analyseras inte så noggrant; den datamängd som kommer in i systemet reduceras kraftigt i första beräkningsledet varför datamängden mellan processer antas relativt liten.

#### **Datainläsning**

De data som kommer in i dagens system är 16-bitars dataord i 32 kHz. De

olika givarna varierar från system till system, men en grunduppsättning består av följande givare (se avsnitt 1.1 för en kort beskrivning av de olika givarna):

- 15–20 laserkameror;
- 2 accelerometrar;
- 2 inklinometrar;
- gyro;
- hjulpulser (hastighet och distans).

Datahastigheten på en sådan uppsättning blir antal givare · 32 kHz · 16 bit vilket resulterar i c:a 12 Mbit/s. För att slippa ligga och vänta, och läsa ett värde i taget finns det idag ett minne med FIFO-arkitektur (*first in – first out*) mellan processorns och den databuss på vilken givarna är inkopplade.

En önskvärd funktion i den nya processorn är en höghastighetsport samt möjlighet till DMA-överföring (*direct memory access*) mellan port och minne, för att ersätta det externa FIFO-minnet med tillhörande logik. Eftersom önskemål finns om att öka frekvensen till 64 kHz, samt att möjliggöra ett större antal givare – uppemot 50 st. måste porten ha en kapacitet på minst 50 Mbit/s.

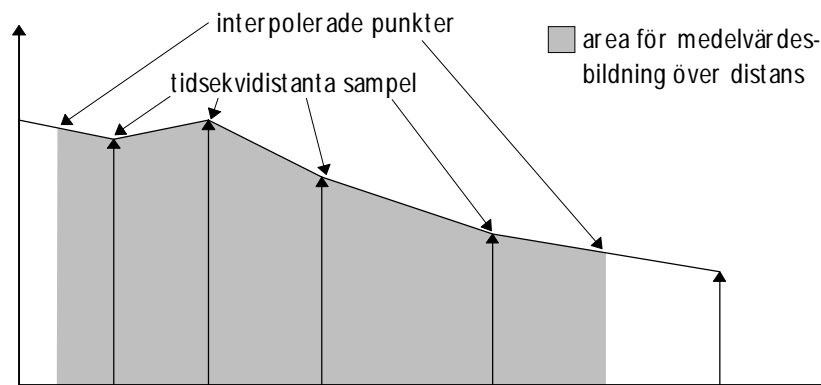
### Beräkningar på data

De beräkningar som utförs på data kan delas in i två delar; förbehandling och applikationsspecifika beräkningar. Under förbehandlingen sorteras ogiltiga data bort, men den viktigaste uppgiften är att konvertera data från tidsdomän (lika lång tid mellan två värden) till spatialdomän (lika långt avstånd mellan två värden), dvs. kompensera data för bilens hastighet för att vara oberoende av densamma. Ur matematisk synvinkel ger det ett mer korrekt resultat att göra compensationen direkt på inkommande data än senare i beräkningskedjan, t.ex. med filterkoefficienter som funktion av hastigheten. I det senare fallet kommer filtret aldrig riktigt svänga in då dess egenskaper till viss del förändras hela tiden.

Beräkningarna kan visserligen utföras med heltal, men att slippa analysera talområden och noggrannhet skulle reducera utvecklingstiden och sannolikheten för fel i implementeringen. Därför bör processorn hantera flyttal.

Omvandlingen från tids- till spatialdomän görs genom att interpolera fram de två sidovärdena, och sedan räkna fram ett medelvärde över de två sidovärdena och alla värden däremellan genom att beräkna arean under

kurvan (se figur 4).



Figur 4: Omvandling från tids- till spatialdomän

Avståndet mellan två värden i spatialdomän beror på vilken applikation som ska använda värdet. Ungefär hälften av laserkamerorna behöver omvandlas till 0,5 mm distans, och för resterande räcker det med 25 mm. Parallellt med omvandlingen beräknas även ett medelvärde över en längre distans, 100 mm.

Den hastighet som systemet ska klara av är minst 90 km/h (25 m/s). För 0,5 mm distans innebär det en omvandlingsfrekvens på 50 kHz, och i fallet 25 mm en frekvens på 1 kHz.

De steg, och antal instruktioner som krävs i omvandlingen är:

- minnesåtkomst:  $2 \cdot$  antal värden;
- detektering av ogiltiga värden samt omvandling till flyttal:  $2 \cdot$  antal värden;
- interpolation: 4 instruktioner;
- medelvärdesbildning:  $4 \cdot$  (antal värden + 1);
- medelvärdesbildning över längre distans:  $1 \cdot$  antal värden.

Antal instruktioner som ska utföras blir en funktion av antalet värden,  $x$ , i ett intervall;  $f(x) = 8 + 9x$ . Multiplicerat med antal givare och omvandlingsfrekvens (för 90 km/h) får man ett krav på 10 miljoner instruktioner per sekund (Mips) för 0,5 mm distans och 6 Mips för 25 mm distans. Siffrorna är naturligtvis mycket ungefärliga, och kan variera mycket beroende på t.ex. instruktionsuppsättning.

### Applikationsspecifika beräkningar

Av de applikationsspecifika beräkningarna är det de som använder sig av

spatialdata på 0,5 mm distans som kräver hög processorkapacitet. Idag finns två applikationer som kräver den noggrannheten: textur och sprickmätning.

Den implementering av textur som används idag utgår från data i tidsdomän och räknar sedan om filterkoefficienter utifrån hastigheten. Beräkningen görs parallellt för 1–4 lasrar. Med data i spatialdomän skulle implementeringen reduceras till ett andra gradens IIR-filer. Antalet instruktioner för att implementera ett sådant filter är ungefär 8 instruktioner per sampel. Om textur beräknas på 5 lasrar ger detta ett krav på  $8 \cdot 5 \cdot 50 \text{ kHz} = 2 \text{ Mips}$ .

Sprickmätningens funktion är att räkna antalet sprickor inom olika kategorier (djup och bredd). Varje sampel jämförs med ett löpande medelvärde för marknivån. Om det understiger värdet med ett visst djup räknar man antal sampel till värdet går upp igen. De två parametrar som sedan används för att kategorisera sprickorna är maxdjupet och längden. Att uppskatta dess tidsåtgång är extra svårt då den är kontextberoende, dvs. utför olika saker beroende på aktuellt tillstånd, men utifrån befintlig kod uppskattas tidsåtgången till 20 instruktioner per sampel. Sprickor beräknas idag på upp till 6 lasrar, vilket ger ett krav på  $20 \cdot 6 \cdot 50 \text{ kHz} = 6 \text{ Mips}$ .

Eftersom övriga beräkningar använder sig av data med den längre sampeldistansen blir tidsåtgången betydligt mindre, även med ganska hög komplexitet på beräkningarna. Med t.ex. 25 mm sampeldistans räcker 1 Mips till 1000 instruktioner per sampel, vilket mer än väl räcker till för t.ex. minsta- kvadratanpassning för alla 20 lasrar (för att få fram vägens lutning). En uppskattning som väl bör täcka dessa beräkningar är 5 Mips.

### Övrig processortid

Utöver de rena beräkningarna tillkommer tid för schemaläggning samt kommunikation mellan processer och värddator. En schemaläggare bör maximalt utnyttja några procent av processorns kapacitet, och detta tillsammans med kommunikation mellan processer och mellan processer och värddator uppskattas till c:a 5 Mips.

### Sammanfattning av processorkrav

Summan av kraven kan nu summeras till ungefär 35 Mips. Som tidigare nämnts är det många faktorer som spelar in på denna siffra, så jämförelsen kan inte göras direkt med en tilltänkt processor, utan även instruktionsuppsättningen bör beaktas. Processorns prestanda bör dock, oavsett instruktionsuppsättning, inte vara lägre än denna siffra.

En sammanfattning av de krav som ställts är att:

- den bör vara av flyttalstyp;
- processorns prestanda bör vara minst 35 Mips (Mflops);
- den ska vara utrustad med höghastighetsport, >50 Mbit/s, för datainläsning;
- DMA-överföring ska finnas från höghastighetsport till minne.

## 3.2 Krav på schemaläggare

De processer som kommer att vara aktuella i ett nytt system kan delas in i tre grupper: processer som behandlar data i tidsdomän, processer som behandlar data i spatialdomän, och övriga processer. Processer i de två första grupperna är periodiska och de övriga är aperiodiska.

Processerna som arbetar med data i tidsdomän, dvs. de som gör omvandlingen från tids- till spatialdomän, har en periodtid som är beroende av datafrekvensen och storleken på minnesbufferten för indata. Då beräkningarna är enkla bör variationerna i beräkningstiden vara små.

Något som är svårare att modellera på ett exakt sätt är de processer som arbetar med data i spatialdomän. Även här är periodtiden beroende av datafrekvens och buffertstorlek, men datafrekvensen är inte längre konstant utan varierar med bilens hastighet. Detta medför att periodtid och/eller beräkningstid kommer att variera, med högre prestandakrav ju snabbare bilen kör.

Om man schemalägger processerna med någon algoritm för periodiska processer måste man sätta periodtid och beräkningstid utifrån bilens maximala hastighet. Detta kan vara alltför pessimistiska värden som leder till dålig nyttjandegrad vid normala hastigheter. Så länge hela systemet kan köra på en processor är dålig nyttjandegrad inget problem, men en alltför pessimistisk beräkning kan ange att en processors kapacitet inte räcker till, och då kan man felaktigt tro att systemet måste utökas med fler processorer.

Den sista gruppen av processer, de aperiodiska processerna, kommer att hantera signaler för att markera kontrollpunkter i mätningarna. Signalen kan genereras av t.ex. fotoceller placerade vid vägen, eller av knapptryckningar. Dessa signaler innebär inte att något ska förändras för övriga processer, utan bara att någon slags markering ska göras för tidpunkten. Dess beräkningstid blir därför mycket kort i förhållande till övriga processer i systemet. Om man förutsätter att systemet noterar tidpunkten för när en händelse inträffar kan dessutom dessa processers

tidsgränser ses som mjuka.

### Tidsgränser

De periodiska processernas tidsgränser beror på indatafrekvensen och storleken på de buffertar där indata lagras. Genom att variera fyllnadsgraden av dessa buffertar kan man förändra vilken typ av tidsgräns processen har. Med 50 procent fyllnadsgrad, dvs. data behandlas på en halv buffert i taget, får man  $D_i = T_i$ ; beräkningarna på en halva får inte ta längre tid än det tar att fylla den andra halvan. Ökar man fyllnadsgraden får man  $D_i < T_i$ , och med minskad fyllnadsgrad  $D_i > T_i$ . Det sista fallet kan också utnyttjas till viss feltolerans; om fyllnadsgraden är 40 procent och  $D_i = T_i$  kan en tidsgräns passeras med 20 procent enstaka gånger utan att data förloras.

Något krav på vilken typ av tidsgräns som ska användas ställs inte, men den bör väljas så att man kan undersöka om en uppsättning processer är möjlig att schemalägga i förväg. En sådan funktion är också ett krav på implementeringen – antingen externt, eller i realtidssystemet i samband med uppstart av nya processer. En sådan funktion kan också komma till nytta om systemet utökas med flera processorer för att fördela processer mellan processorerna.

### Uppstart av processer

Oavsett om en statisk eller en dynamisk algoritm används kan problem uppstå om en process schemaläggs som viktig under uppstart. I en prioritetsstyrd algoritm kommer den process som har högst prioritet blockera alla andra under hela dess initialisering. Detta kan ta betydligt längre tid än vissa andra processers periodtid, och det resulterar i missade tidsgränser.

Detta problem bör lösas genom att processen markerar när den börjar med sin tidskritiska del, t.ex. genom att sätta en flagga i den struktur som beskriver processen. Först därefter schemaläggs den för att klara sina tidsgränser. Med en sådan lösning man också tillåta några av de konstruktioner som normalt förbjuds i realtidssystem (se avsnitt 2.3) så länge processen inte befinner sig i den tidskritiska delen av programmet.

### Uppmätning av beräkningstid

Av de processattribut som associeras med varje process kan periodtid  $T_i$  och tidsgräns  $D_i$  räknas ut som en funktion av indatafrekvens och buffertstorlek. Beräkningstid  $C_i$  är däremot svårare att bestämma

teoretiskt. Det är därför önskvärt att utrusta systemet med en funktion för uppmätning av en processers beräkningstid.

En sådan funktion skulle också kunna möjliggöra detektering av överbelastning i systemet under drift. Även om syftet med statistiska algoritmer är att upptäcka det redan vid prioritetstilldelningen, och att schemaläggaren inte ska behöva hålla reda på det, så kan det vara användbart vid felsökning i systemet.

### **Sammanfattning av systemkrav**

De krav som ställs på schemaläggaren och systemet är:

- extern analys av hur systemet klarar tidsgränserna;
- möjlighet att tillfälligt låta en process arbeta utan tidsgränser under t. ex. uppstart;
- uppmätning av beräkningstid för processer.

## 4 Undersökning och design

---

I detta kapitel undersöks den av OPQ utvalda processorn för att visa att dess prestanda är tillräckliga, och designbeslut om val av algoritm för schemaläggare presenteras och motiveras.

### 4.1 Undersökning av processor

Den processor som valts ut av OPQ är ADSP-2106x SHARC från Analog Devices (Analog Devices, 1995). Då den har både minne, diverse in- och utportar, stöd för sammankoppling av flera processorer och gränssnitt mot värddator integrerat på chipet är det förhållandevis enkelt att konstruera egna system utifrån den, vilket också är den primära anledningen att den valts ut.

Processorn finns i två modeller, ADSP-21060 och ADSP-21062, där skillnaden är storleken på minnet; 4 respektive 2 Mbit. Klockfrekvensen är 40 MHz.

## Prestanda

Processorns prestanda är enligt Analog Devices själva 80 Mflops, med 120 Mflops toppprestanda. Siffrorna nås genom att cykeltiden är 25 ns, och att vissa instruktioner kan utföra upp till tre flyttalsoperationer parallellt. Dessa *multifunction computations* är konstruerade för vanliga algoritmer, t.ex. finns en instruktion för parallell multiplikation och medelvärdesbildning,  $F_x = F_y * F_z$ ,  $F_a = (F_b + F_c) / 2$ , som lämpar sig utmärkt för integralskattning enligt trapetsregeln. Samtliga beräkningssinstruktioner kan även kombineras med en minnesaccess för att ladda eller spara register.

Dessa specialinstruktioner gör att den uppskattade tidsåtgången för beräkningarna (se avsnitt 3.1) i flera fall kan reduceras. För omvandlingen från tids- till spatialdomän kan beräkningstiden reduceras till en tredjedel, c:a 6 Mips. Även övriga beräkningars tidsåtgång reduceras något, varför processorns 40 Mips bedöms som tillräckliga prestanda för att exekvera samtliga applikationer som körs idag.

## Datainläsning

Främst för att kunna överföra data mellan processorer, men även för extern inläsning, har SHARC:en sex 4-bitars länkportar (*link ports*) som var och en hanterar datahastigheter upp till 320 Mbit/s. Portarna kan kopplas till DMA-kanaler, som i sin tur har en funktion, *DMA chaining*, som möjliggör automatisk initialisering av en ny överföring när den förra är färdig. På så sätt kan inläsning av data göras helt automatisk efter en första initialisering. Det interna minnet har separat adress- och databuss för DMA-överföring vilket innebär att exekveringen på processorn inte påverkas av inläsningen.

Hastigheten på en port är nog hög för att inte utgöra någon begränsning. Teoretiskt sett räcker hastigheten till för över 70 givare med 32-bitars data i 128 kHz.

## Övrigt

Utöver det som nämnts ovan har SHARC:en ytterligare några egenskaper som kan vara användbara: serieportar, multiprocessorstöd, och värddatorgränssnitt.

Vissa givare passar inte in i den generella beskrivningen med t.ex. 16-bitars dataord i 32kHz. Som exempel finns planer på positionsbestämning med GPS, *Global Positioning System*, och den mottagare som företaget har idag kommunicerar seriellt. I ett framtida system kan denna enkelt kopplas direkt till en av SHARC:ens två serieportar.

Processorns multiprocessorstöd består av färdig logik för en extern buss på vilken upp till sex SHARC:ar och en värddator kan kopplas samman. Var och en av de sex processorerna och värddatorn kan adressera de sex processorernas minnen och kontrollregister för portar, DMA etc. (s.k. IOP-register). Bussarbitreringen (hur bussen delas mellan processorerna) kan sättas till fix eller roterande prioritet, och dessutom kan en processor låsa bussen under en längre tid för att hantera semaforer.

Multiprocessorstödet ger en möjlighet till utbyggnad av systemet om det visar sig att en processors prestanda inte räcker till. Man kan då tänka sig att inläsning och förbehandling av data görs i en processor, och att övriga beräkningar görs i ytterligare processorer. Om inte den externa bussens kapacitet ( $40 \cdot 32$  Mbit/s) räcker till kan länkportar kopplas mellan de olika processorerna.

## 4.2 Schemaläggare

Av de tre grupper av processer som identifierats för systemet, konstant periodiska (processer i tidsdomän), variabelt periodiska (processer i spatialdomän) och sporadiska, är de två första de viktiga. Dels för att de upptar majoriteten av alla beräkningar, men också för att de, till skillnad från de sporadiska, har hårda tidsgränser. Valet av algoritm görs därför utifrån dessa processers egenskaper, och de sporadiska processerna exekveras i systemets slacktid.

De periodiska processer som har variabel periodtid modelleras med fix periodtid satt efter värsta fallet, dvs. mätbilens högsta hastighet. Förenklingen kan, som nämndes i avsnitt 3.2, vara en alltför pessimistisk uppskattning, men genom att sätta den maximala hastigheten för mätbilen nära den normala hastigheten minimeras felet. Normal hastighet under mätning ligger på 70–90 km/h, men det finns sällan anledning att köra snabbare. T.ex. görs inga omkörningar under mätning eftersom man då skulle förlora data för vägens tvärprofil. Den maximala hastigheten kan därför sättas till t.ex. 100 km/h. Felet jämfört med 90 km/h blir då 11 procent, vilket är en rimlig, maximal försämring av nyttjandegraden.

### Algoritm

Efter att de variabelt periodiska processerna modellerats med fix periodtid har problemet reducerats till schemaläggning av periodiska processer med konstant period. Enligt diskussion i avsnitt 3.2 är tidsgränsen beroende av buffertstorlek. Då minnet på processerna är

relativt stort behöver inte fyllnadsgraden sättas högre än 50 procent vilket innebär att  $D_i \geq T_i$ . Eftersom algoritmer för  $D_i > T_i$  fortfarande befinner sig på forskningsstadiet sätts  $D_i = T_i$ , och fyllnadsgraden till 50 procent. Eventuellt kan fyllnadsgraden minskas något om det visar sig att någon process lågfrekvent överskrider sin beräkningstid, för att därigenom tolerera viss överbelastning.

Med dessa förutsättningar – en mängd periodiska processer – är valet av algoritm enkelt: de statiska algoritmernas styrka är just periodiska processer. Att  $D_i = T_i$  gör att *rate-monotonic scheduling* skulle kunna vara aktuell, men det kräver också att  $O_i = 0$  vilket kan vara en begränsning. För många av processerna gäller att indata kommer från en annan process, och då kan det vara aktuellt att sätta  $O_i \neq 0$ . Valet av algoritm för prioritetstilldelning faller därför på *optimal priority assignment*. Detta medför också att kravet på möjlighet att testa om en mängd processer kan schemaläggas uppfylls.

De förbättringar som gjorts av statiska algoritmer efter *optimal priority assignment* har inte förändrat schemaläggningen av periodiska processer, utan utökat stödet för bl.a. hårda aperiodiska processer. Då samtliga hårda processer är periodiska finns ingen anledning att implementera någon av dessa förbättringar i första versionen. Valet innebär dock ingen återvändsgränd, utan då det blir aktuellt med fler, eventuellt hårda aperiodiska processer kan schemaläggaren utökas.

Att inte en dynamisk algoritm väljs beror till största delen på att forskningen ännu är så ung på området – bevis för att en mängd processer är möjlig att schemalägga är mer omständliga än för statiska algoritmer. Eftersom processmängden är mer eller mindre fix under mätning är det en fördel att ha en så liten del som möjligt av schemaläggaren i realtidssystemet, och lägga en större del *off-line*.

### Schemaläggning av slacktid

De sporadiska processer som ingår i systemet får exekvera i den slacktid som återstår då de periodiska processerna fått den processortid de behöver. De sporadiska processer som är aktuella idag har samtliga mjuka tidsgränser. Därför kan schemaläggningen av slacktid göras så enkel som möjligt, vilket är att sätta prioriteten på de sporadiska processerna lägre än den periodiska process som har lägst prioritet.

För att säkerställa att det finns nog mycket slacktid kan man utöka *optimal priority assignment* så att nyttjandegraden för de periodiska processerna räknas ut. Andelen slacktid ges sedan av 100 minus nyttjandegraden i procent.

### Uppstart av processer

För att hantera det problem med uppstart av en process som nämndes i avsnitt 3.2, tilldelas alla processer lägsta prioritetsnivå vid uppstart i systemet. Detta innebär att de inte avbryter någon av de tidskritiska processerna. När sedan processen är färdig med sin initialisering anropar den en systemprocedur för att sätta sin prioritet till den nivå som tilldelats av *optimal priority assignment*.

## 5 Systemimplementering

---

Implementeringen av realtidssystemet består av två delar: ett program för prioritetstilldelning av en mängd processer utgående från processernas tidsegenskaper, och en prioritetsstyrd schemaläggare för SHARC-processorn. Schemaläggaren är också utökad med en funktion för mätning av beräkningstid.

Avsnittet om *optimal priority assignment* består av många formler tillsammans med motsvarande pseudokod, vilket kan uppfattas som svårsmält när det läses första gången. Algoritmen i sig är dock så komplex att mer än en genomläsning kan behövas.

### 5.1 Prioritetstilldelning

Den algoritm som ska användas för prioritetstilldelning, *optimal priority assignment*, finns beskriven i en rapport av Audsley (1991) där också beviset för dess funktionalitet ges. Den pseudokod som presenterades innehöll dock tämligen många fel – förmodligen slarvfel – så den implementering som presenteras här bygger snarare på teorin i rapporten

än på pseudokoden.

Den yttersta loopen i algoritmen itererar från lägsta till högsta prioritet över de prioritetsnivåer som ska tilldelas (med lägre prioritet representerat av ett högre tal). För varje prioritetsnivå undersöks om det finns någon process som kan schemaläggas på den nivån. Övriga processer antas ha högre prioritet, utan bestämd ordning. Enligt sats 2 i avsnitt 2.5 gäller då, att om en process hittas för vilken aktuell prioritetsnivå gör hela uppsättningen möjlig att schemalägga, så kan den processen tilldelas den prioriteten. Nästa steg är då att hitta en process för vilken nästa, högre, prioritetsnivå är lämplig. De processer som redan tilldelats prioritet behöver då inte räknas med.

Om ingen process hittas för en viss prioritetsnivå gäller, enligt sats 1 i avsnitt 2.5, att uppsättningen överhuvudtaget inte kan schemaläggas.

Nedanstående pseudokod motsvarar den yttre loop som beskrivits ovan:

```
PriorityAssignment()
BEGIN
  FOR i IN (numproc..1)
    unassigned = TRUE
    FOR j IN (i..1)
      swapproc(i, j)           // Move process to i:th pos
      IF (Feasible(i)) THEN  // If i:th process is
                             // feasible for priority i.
        proc[i].P = i      // Assign priority
        unassigned = FALSE
      ENDIF
    ENDFOR
    IF (unassigned)
      exit                // No priority assignment exists
    ENDIF
  ENDFOR
END
```

Nästa steg är funktionen `Feasible(i)` som undersöker om en process  $\tau_i$  får den processortid den behöver innan tidsgränsen, under förutsättningen att  $\tau_i$  tilldelas prioritet  $i$ . Undersökningen görs genom att simulera exekveringen av processerna vid alla aktiveringar av  $\tau_i$  under en bestämd tidsperiod. De processer som redan tilldelats (en lägre) prioritet behöver inte tas med i simuleringen, bara de övriga för vilka prioritetsordningen är godtycklig (se ovan). Som nämnts i avsnitt 2.5

visar Audsley att denna period definieras av  $t = [S_i, S_i + P_i)$  där  $S_i$  och  $P_i$  ges av:

$$S_i = \left\lceil \frac{O_{\max} - O_i}{T_i} \right\rceil \cdot T_i = \left\lceil \frac{O_{\max}}{T_i} \right\rceil \cdot T_i$$

där  $O_{\max} = (O_1, O_2, \dots, O_{i-1})$

och under antagandet att  $O_i = 0$

$$P_i = lcm(T_1, T_2, \dots, T_i)$$

Då process  $\tau_i$  aktiveras för alla tidpunkter  $O_i + nT_i$  kan man observera att  $O_i$  kan förändras med multipler av  $T_i$  utan att dessa tidpunkter förändras med undantag av en period i början. Man kan också observera, att om alla  $O_i$  förändras med ett värde  $l$  kommer förhållandet mellan alla processers aktiveringar att vara detsamma.

Med hjälp av dessa observationer kan man göra en omfördelning av fördröjningstiderna,  $O_1, \dots, O_i$  så att  $O_i = 0$  enligt krav:

```
FOR j IN (1..i)
  proc[j].O = proc[j].O - proc[i].O
  WHILE (proc[j].O < 0) // Add period(s)
    proc[j].O = proc[j].O + proc[j].T // until 0 >= 0
  ENDWHILE
ENDFOR
```

Efter omfördelningen kan  $S_i$  och  $P_i$  räknas ut:

```
Omax = proc[1].O
FOR j IN (2..i)
  Omax = max(Omax, proc[j].O)
ENDFOR
S = ceil(Omax / proc[i].T) * proc[i].T

P = proc[1].T
FOR j IN (2..i)
  P = lcm(P, proc[j].T)
ENDFOR
```

För att visa att en process  $\tau_i$  klarar sin tidsgräns efter aktivering vid tidpunkt  $t$ , räknar man ut den blockeringstid (*interference*)  $I_i^t$  process  $\tau_i$  får

vänta på andra processer med högre prioritet under perioden  $[t, t + D_i)$ . Om villkoret  $I_i^t + C_i \leq D_i$  gäller klarar processen sin tidsgräns efter aktivering vid tidpunkt  $t$ .

$I_i^t$  kan i sin tur delas upp i två termer;  $R_i^t$  och  $K_i^t$ .  $R_i^t$  kallas återstående blockeringstid (*remaining interference*), och definieras som den tid som fortfarande återstår att exekvera för processer med högre prioritet vid tidpunkt  $t$ .  $K_i^t$  kallas för skapad blockeringstid (*created interference*) och definieras som den tid processer med högre prioritet behöver för exekvering efter aktivering i intervallet  $[t, t + D_i)$ .

Det kompletta villkoret för att en uppsättning processer är möjliga att schemalägga kan nu skrivas som:

$$\forall i : 1 \leq i \leq n$$

$$\forall t \in B_i : R_i^t + K_i^t + C_i \leq D_i$$

$$B_i = \{t \mid t \in (S_i, S_i + T_i, S_i + 2T_i, \dots, S_i + P_i)\}$$

För att beräkna  $R_i^t$  vid tidpunkt  $t = mT_i$ , dvs. då  $\tau_i$  ska exekvera gång  $m+1$ , utgår man från en mängd tupler,  $\beta$  på formen  $(C_j, t)$  som motsvarar ett önskemål från process  $\tau_j \in \{\tau_1, \tau_2, \dots, \tau_{i-1}\}$  om  $C_j$  beräkningstid vid tidpunkt  $t \in [(m-1)T_i + D_i, mT_i)$ .

Om  $L_i^{m-1}T_i + D_i > 0$ , där  $L_i^t$  representerar återstående beräkningstid för processerna  $\tau_1.. \tau_i$  vid tidpunkt  $t = 0, D_i, T_i + D_i, 2T_i + D_i, \dots$ , lägger man till ytterligare en tupel,  $(L_i^{m-1}T_i + D_i, (m-1)T_i + D_i)$  till  $\beta$ . Denna tid motsvarar alltså den del av  $R_i^t$  och  $K_i^t$  för  $t = (m-1)T_i$  som inte exekverades under det intervall som simulerades för att räkna ut dessa.

Genom att sedan iterera över icke-minskande  $t$ -värden i  $\beta$  kan man räkna ut den beräkningstid som återstår för process  $\tau_i$  vid tidpunkt  $mT_i$ .

I pseudokoden nedan har användandet av  $L_i^t$  förenklats. Om  $L_i^t > 0$  för något  $t \geq D_i$  innebär detta att process  $\tau_i$  (och eventuellt ytterligare processer) inte klarade sin tidsgräns, och då avbryts algoritmen ändå. Det  $L_i^t$  som behöver tas hänsyn till är då  $L_i^t$  för  $t = S_i - T_i + D_i$  vilket kan räknas ut genom att vid första exekveringstillfället även ta med de tupler  $(C_j, t)$  där  $t \in [0, S_i - T_i + D_i)$  i  $\beta$ . Variabeln *starttime* antas därför ha värdet  $(m-1)T_i + D_i$  för  $m > 1$  och 0 för  $m = 1$  i pseudokoden nedan. Variabeln *t* har värdet av  $mT_i$ .

```

time = starttime
R = 0
FOR tr IN (time..t)
  FOR j in (1..i-1)

```

```

        IF (Released(j, tr)) THEN // If process j is
                                // released at time tr
            IF (tr >= time + R) THEN
                R = 0
            ELSE
                R = R - (tr - time)
            ENDIF
            time = tr
            R += proc[j].C
        ENDIF
    ENDFOR
ENDFOR
R = R - (t - time)
IF (R < 0) THEN
    R = 0
ENDIF

```

Det sista steget i algoritmen är nu att beräkna  $K_i^t$  för  $t = mT_i$ . Detta utförs ekvivalent med uträkningen av  $R_i^t$ , men med mängden  $\eta$  definierad som tupler  $(C_j, t)$  där  $t \in [mT_i, mT_i + D_i)$ . Även här är variabeln  $t$  satt till  $mT_i$ , och  $R$  är det tidigare uträknade värdet av  $R_i^{mT_i}$ .

```

K = 0
next_free = t + R
total_created = R
FOR tr IN (t..t+proc[i].D)
    FOR j IN (1..i-1)
        IF (Released(j, tr)) THEN
            total_created = total_created + proc[j].C
            IF (next_free < tr) THEN
                next_free = tr
            ENDIF
            K = K + min(t + proc[i].D - next_free,
                    proc[j].C)
            next_free = min(t + proc[i].D,
                    next_free + proc[j].C)
        ENDIF
    ENDFOR
ENDFOR

```

Efter beräkning av  $R_i^t$  och  $K_i^t$  återstår bara att testa villkoret

$R_i^t + K_i^t + C_i \leq D_i$ . Om det gäller klarar process  $\tau_i$  sin tidsgräns efter aktivering vid tidpunkt  $t$ .

En fullständig implementering i C av algoritmen ges i bilaga A.

## 5.2 Realtidssystem

Den minimala uppsättning delar som behövs i ett operativsystem för att kunna prova en schemaläggare är en tidsstyrd rutin för att aktivera processer vid bestämda tidpunkter, rutiner för att spara och återställa processers omgivning (*dispatcher*) och något sätt att definiera (starta upp) processer.

### Aktivering av processer

Den rutin som aktiverar processer styrs med processorns interna klocka. Klockräknaren sätts till den tid som återstår till nästa processaktivering för någon process, och genererar då ett avbrott där rutinen anropas. De processer som ska aktiveras ändrar då tillstånd från väntande till körbara. Därefter anropas schemaläggaren som undersöker om någon av de aktiverade processerna har högre prioritet än den exekverande processen, och om så är fallet byts exekverande process.

```
ActivateProcesses()  
BEGIN  
  FOR p IN waitingList  
    IF (ReleaseTime(p) <= currentTime) THEN  
      waitingList.Remove(p)  
      readyList.Insert(p)  
    ENDIF  
  ENDFOR  
  
  tval = FindShortestTime(waitingList, readyList)  
  SetTimer(tval - currentTime)  
  Scheduler(false)  
END
```

Då en körbar process med kort periodtid kan ha sin nästa aktivering innan en väntande process med lång periodtid, måste alla processer – både väntande och körbara – undersökas för att hitta nästa tidpunkt för processaktivering.

## Schemaläggare

Förutom när nya processer aktiveras anropas schemaläggaren även då en process är färdig med beräkningarna. I det fallet sparas processomgivningen, och tillståndet sätts till väntande. Därefter sätts den körbara process som har högst prioritet till exekverande process genom att återställa dess omgivning.

När nya processer aktiverats måste schemaläggaren jämföra prioriteterna mellan den exekverande processen och de som aktiverades, och eventuellt byta exekverande process.

I pseudokoden nedan indikerar variabeln `ready` om den exekverande processen är färdig, eller om schemaläggaren anropats för att nya processer aktiverats.

```
Scheduler(bool ready)
BEGIN
  IF (ready) THEN
    DispatchSave(activeProcess)
    waitingList.Insert(activeProcess)
    activeProcess = readyList.First()
    DispatchRestore(activeProcess)
  ELSE
    p = readyList.First()
    IF (activeProcess.P > p.P) THEN
      DispatchSave(activeProcess)
      readyList.Remove(p)
      readyList.Insert(activeProcess)
      DispatchRestore(p)
      activeProcess = p
    ENDIF
  ENDIF
END
```

## Dispatcher

När aktiv process ska avbrytas måste register, programräknare, statusflaggor, hårdvarustackar osv. sparas undan för att kunna återställas nästa gång processen ska exekvera. Det är denna uppgift en *dispatcher* har. Då SHARC-processorn har ett stort antal register (16 generella register, 64 adressregister samt ett antal specialregister), och dessutom tre hårdvarustackar för bl.a. programräknaren är *dispatch*-rutinen tämligen omfattande.

Normalt sett behöver vissa register sparas undan när schemaläggaren aktiveras, oavsett om en aktuell process ska bytas eller inte. Detta för att schemaläggaren själv ska få tillgång till några register. I SHARC-processorn finns dock en lösning som gör att man kommer undan detta problem – alla register finns i två oberoende uppsättningar. Genom att sätta en flagga aktiverar man den ena eller andra delen och kan då låta systemrutinerna arbeta med en egen uppsättning.

### Uppstart av processer

Uppstart av processer är tänkt att göras dynamiskt – när systemet är initialiserat kommunicerar värddatorn med systemet och startar processerna en och en. I denna testversion ingår istället processerna i systemet, och startas i samband med systemets initialisering. Anledningen till detta är dels att hålla nere implementeringens komplexitet, men främst att den version av SHARC-kompilatorn som finns på OPQ inte kan generera relokerbar kod, dvs. kod som kan placeras var som helst i minnet. Detta innebär att det för tillfället inte finns något sätt att dela upp ett program i flera delar och ladda ner dem separat till systemet.

Testversionen kommer inte heller hantera lösningen med lägre prioritetnivå under uppstart (se avsnitt 4.2).

### Uppmätning av beräkningstid

För att mäta upp beräkningstid för processer modifieras schemaläggaren så att den noterar när en process inleder exekvering. När sedan processen avbryts eller är färdig med beräkningarna adderas skillnaden mellan aktuell tid och starttid till en variabel som läses av då processen är färdig.

Den extra kod som behövs är markerad med fet stil i pseudokoden nedan:

```
Scheduler(bool ready)
BEGIN
  IF (ready) THEN
    activeProcess.Time = activeProcess.Time +
      current_time - activeProcess.Start
    if (activeProcess.Time > activeProcess.MaxTime)
      activeProcess.MaxTime = activeProcess.Time;
    activeProcess.Time = 0

    DispatchSave(activeProcess)
```

```
waitingList.Insert(activeProcess)
activeProcess = readyList.First()

activeProcess.Start = current_time

DispatchRestore(activeProcess)
ELSE
p = readyList.First()
IF (activeProcess.P > p.P) THEN
activeProcess.Time = activeProcess.Time +
current_time - activeProcess.Start

DispatchSave(activeProcess)
readyList.Remove(p)
readyList.Insert(activeProcess)

activeProcess = p
activeProcess.Start = current_time
DispatchRestore(activeProcess)
ENDIF
ENDIF
END
```

## 6 Resultat och slutsatser

---

I detta kapitel presenteras resultat från provkörningar av både algoritmen för prioritetstilldelning, och den förenklade version av realtidssystemet som implementerats. Utifrån dessa ges sedan slutsatser och rekommendationer angående fortsatt utveckling av systemet.

### 6.1 Resultat

Att utvärdera systemets funktion är svårt utan riktiga data att ta hand om, men för detta är varken hård- eller mjukvara färdig. Därför provkörs systemet dels med en uppsättning processer med känd prioritetsfördelning som används som exempel i Audsleys rapport (1991), och en uppsättning vars tidsattribut motsvarar några mätapplikationers tider.

Audsleys processuppsättning (se tabell 1) är intressant främst för den prioritetstilldelning som ges av *optimal priority assignment*. Både *rate-monotonic scheduling* och *deadline monotonic scheduling* skulle sätta prioritetsordningen till den ordning de står i tabellen, vilket i båda fallen

medför att tidsgränser missas. *Optimal priority assignment* väljer istället att sätta prioriteten för t.ex.  $\tau_2$  till 4, trots dess korta period och tidsgräns. Därmed klarar också alla processer sina tidsgränser fastän uppsättningen har full nyttjandegrad, 100 procent.

Processuppsättningen med fiktiva mätapplikationer (se tabell 2) provkors för att få uppfattning om hur exekveringen fördelas i ett framtida system, och hur mycket tid som används av schemaläggaren. Nyttjandegraden i uppsättningen är 52 procent. Att  $D_i \neq T_i$  för  $\tau_2$  och  $\tau_3$  beror på att sprickmätningen utnyttjar beräkningar både från textur och spatialomvandlingen. Därför måste texturs beräkningar vara färdiga när sprickberäkningarna påbörjas.

Processernas tidsattribut och prioritet för de båda uppsättningarna visas i tabell 1 respektive tabell 2. Observera att beräkningstiderna anges inklusive den tid som går åt för att spara undan och återställa processomgivningarna vid processbyte (se avsnitt 2.5). Tiden har mätts upp till 18  $\mu\text{s}$ .

Process	Period $T_i$	Tidsgräns $D_i$	Beräknings- tid $C_i$	Fördröj- ning $O_i$	Prioritet
$\tau_1$	10	1	1	4	1
$\tau_2$	10	2	1	5	4
$\tau_3$	20	6	5	0	2
$\tau_4$	40	9	8	7	3
$\tau_5$	40	14	8	27	6
$\tau_6$	40	30	6	0	5

Tabell 1: Processer från Audsleys rapport (tider i ms)

Process	Period $T_i$	Tidsgräns $D_i$	Beräknings- tid $C_i$	Fördröj- ning $O_i$	Prioritet
$\tau_1$ – Tid till spatial	1	1	0,22	0	1
$\tau_2$ – Textur 1	4	2	0,16	4	2
$\tau_3$ – Textur 2	4	2	0,16	4	3
$\tau_4$ – Sprickor 1	4	4	0,40	6	4
$\tau_5$ – Sprickor 2	4	4	0,40	6	5
$\tau_6$ – Längsprofil	40	40	0,20	40	6
$\tau_7$ – Tvärprofil	40	40	0,20	40	7
$\tau_8$ – Spårdjup	40	40	0,30	40	8

$\tau_9$ – Position (GPS)	500	500	0,50	0	9
------------------------------	-----	-----	------	---	---

Tabell 2: Processer motsvarande mätapplikationer (tider i ms)

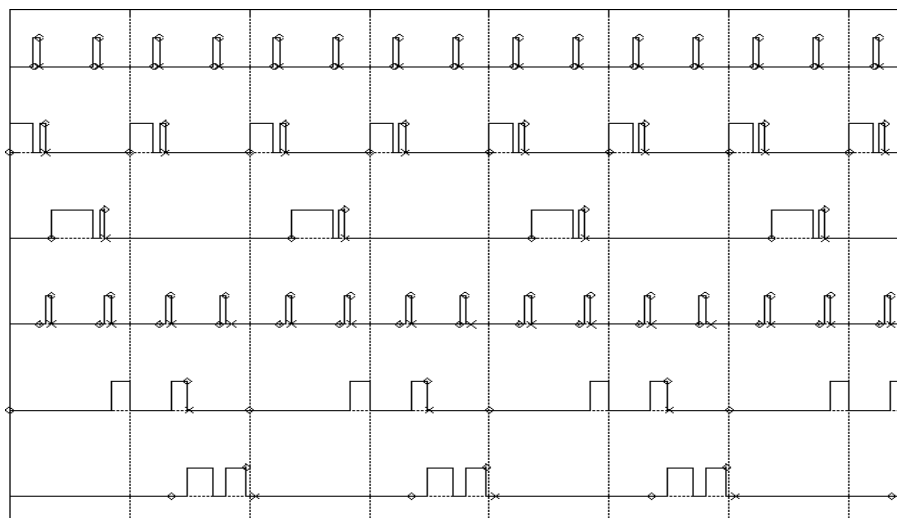
Utöver dessa processer tillkommer för båda uppsättningarna en bakgrundsprocess med lägsta prioritet. Den kommer att utnyttja den tid som inte används av någon av de tidskritiska, periodiska processerna, och därigenom ge ett mått på mängden slacktid. För att kunna observera systemets funktion modifieras även schemaläggaren så att den lagrar alla processbyten, och aktuell tid för bytet. Dessutom sparar den undan den processortid som används av schemaläggaren i sig.

### Test av schemaläggning

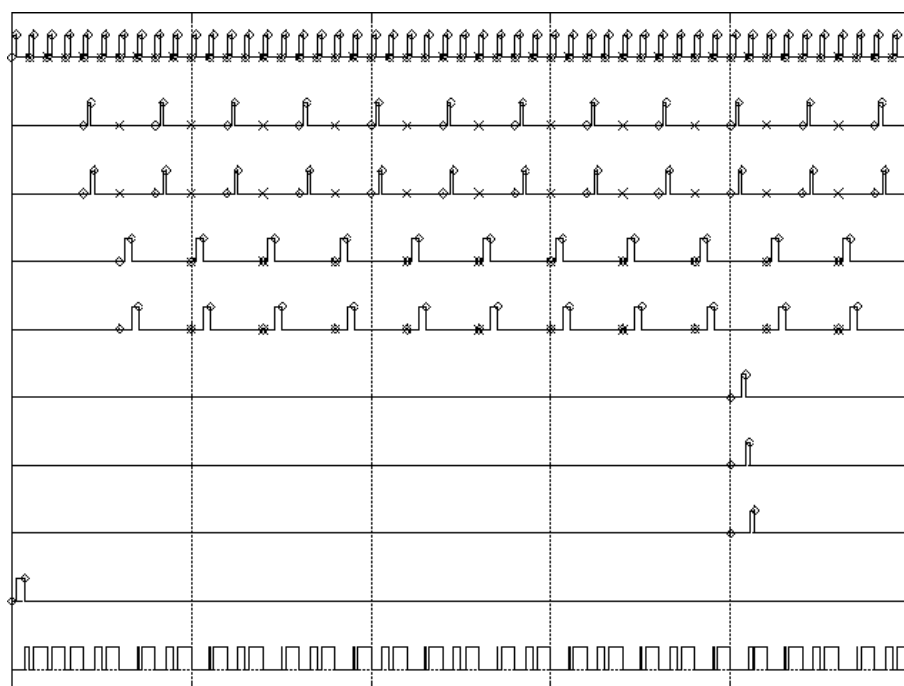
I figur 5 och figur 6 nedan visas hur processortiden fördelas mellan processerna i respektive uppsättning. Ringarna vid baslinjen motsvarar den tidpunkt då en process aktiveras, och de andra ringarna då processen är färdig med beräkningarna. Kryssen markerar tidsgränserna. Processerna är sorterade efter prioritetsordning med högst prioritet överst.

Då nyttjandegraden är 100 procent i Audsleys exempel finns ingen slacktid, men för den andra uppsättningen var andelen slacktid 48 procent. Den bakgrundsprocess som exekverar i slacktiden visas längst ner i figur 6.

I figurerna kan man också se att alla tidsgränser klaras.



Figur 5: Tidsdiagram över exekvering av Audsleys processuppsättning



Figur 6: Tidsdiagram över exekvering av fiktiva mätapplikationer

### Tidsåtgång i systemet

Den andel tid som går åt till schemalaggnen är 1,6 procent respektive 7,2 procent. Anledningen att det tar mer tid för den andra processuppsättningen är att processerna har högre frekvens, och då sker byten oftare. Även den högre siffran är dock fullt godtagbar. Om det ändå skulle bli problem kan detta åtgärdas genom att öka storleken på databuffertarna för att öka processernas periodtider.

## 6.2 Slutsatser och rekommendationer

De provkörningar av systemet som gjorts har visat att schemaläggaren fungerar som det var tänkt, och att tidsåtgången i systemet är låg. Det finns därför inga hinder att fortsätta utvecklingen av ett nytt system med resultatet av detta arbete som grund.

Innan det »slutgiltiga« beslutet om att ersätta det befintliga systemet tas bör dock en testversion med mätdata och någon riktig applikation provas. Därför bör nästa delsteg vara att bygga den hårdvara som krävs för att kunna läsa in mätdata till processorn, samt att implementera rutiner i mjukvara för detta.

Om det visar sig att SHARC-processorn, tvärtemot undersökningen, inte

är snabb nog, eller om processorn av annan anledning väljs bort, innebär detta ändå inte att arbetet är bortkastat. Oavsett processorval visar detta arbete att ett nytt system bör utvecklas baserat på teorin om realtidssystem, och då är de föreslagna lösningarna en bra utgångspunkt.

# A Programkod

---

## A.1 Optimal priority assignment

```
struct Process {
    int T, D, C, O;
    int P;
    int Otmp; // To store rearranged offset
};

int Feasible(struct Process *proc, int i)
{
    int j, Omax;
    int t, tr, time, starttime;
    int next_free, total_created;
    int S, P, R, K;

    for (j = 0; j <= i; j++) {
        proc[j].Otmp = proc[j].O - proc[i].O;
        while (proc[j].Otmp < 0)
            proc[j].Otmp += proc[j].T;
    }
}
```

```

Omax = proc[0].O;
for (j = 1; j <= i; j++)
    Omax = max(Omax, proc[j].Otmp);
S = (int)ceil((double)Omax / proc[i].T) * proc[i].T;

P = proc[0].T;
for (j = 1; j <= i; j++)
    P = lcm(P, proc[j].T);

starttime = 0;
for (t = S; t < S + P; t += proc[i].T) {
    R = 0;
    time = starttime;

    for (tr = time; tr < t; tr++) {
        for (j = 0; j < i; j++) {
            if (((tr - proc[j].Otmp) % proc[j].T) == 0) {
                if (tr >= time + R)
                    R = 0;
                else
                    R -= (tr - time);
                time = tr;
                R += proc[j].C;
            }
        }
    }
    R -= (t - time);
    if (R < 0)
        R = 0;

    K = 0;
    next_free = t + R;
    total_created = R;

    for (tr = t; tr < t + proc[i].D; tr++) {
        for (j = 0; j < i; j++) {
            if (((tr - proc[j].Otmp) % proc[j].T) == 0) {
                total_created += proc[j].C;
                if (next_free < tr)
                    next_free = tr;
                K += min(t + proc[i].D - next_free,
                    proc[j].C);
                next_free = min(t + proc[i].D,
                    next_free + proc[j].C);
            }
        }
    }
    if (!(R + K + proc[i].C <= proc[i].D))
        return 0;

    starttime = tr;
}

int PriorityAssignment(struct Process *proc, int num)
{
    int i, j;
    int succeeded = 1;

    for (i = num - 1; i >= 0 && succeeded; i--) {

```

```
        succeeded = 0;
        for (j = i; j >= 0 && !succeeded; j--) {
            swapproc(&proc[i], &proc[j]);
            if (Feasible(proc, i)) {
                proc[i].P = i + 1;
                succeeded = 1;
            }
        }
    }
    return succeeded;
}

int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else if (b > a)
        return gcd(a, b % a);
    else
        return gcd(b, a % b);
}

int lcm(int a, int b)
{
    return a * b / gcd(a, b);
}

void swapproc(struct Process *a, struct Process *b)
{
    struct Process t = *a;
    *a = *b; *b = t;
}

/* end of file */
```

# Referenser

---

- Analog Devices (1995). *ADSP-2106x SHARC Users Manual*, första upplagan. Analog Devices, Inc.
- Audsley, Neil C. (1990). *Deadline monotonic scheduling*, Technical Report YCS 146. Department of Computer Science, University of York.
- Audsley, Neil C. (1991). *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*, Technical Report YCS 164. Department of Computer Science, University of York.
- Ben-Ari, M. (1982). *Principles of concurrent programming*. Prentice-Hall International Inc., ISBN 0-13-701078-8.
- Burns, Alan (1995). »Preemptive priority-based scheduling: an appropriate engineering approach«. *Advances in real-time systems*, Sang H. Son (redaktör). Prentice Hall, Inc. ISBN 0-13-083348-7.
- Burns, Alan och Wellings, Andrew J. (1990). *Real-time systems and their programming languages*, första upplagan. Addison-Wesley Publishing Company, Inc., ISBN 0-201-17529-0.
- Lehoczky, John P. och Ramos-Thuel, Sandra (1995). »Scheduling periodic and aperiodic tasks using the slack stealing algorithm«. *Advances in real-time systems*, Sang H. Son (redaktör). Prentice Hall, Inc. ISBN 0-13-083348-7.
- Liu, C. L. och Layland, J. W. (1973). »Scheduling algorithms for multiprogramming in a hard real-time environment«. *Journal of the ACM*, 20(1), s. 40–61.

Young, Stephen John (1982). *Real Time Languages: design and development*.  
Chichester, Horwood.

# Index

---

- accelerometer 1
- aperiodisk process 13
- beräkningstid 13
- best-effort scheduling 17
- blockeringstid 33
  - skapad 34
  - återstående 34
- busy-wait 11
- cobegin 8, 9, 10
- computation time *se* beräkningstid
- concurrent execution *se* samtidig
  - exekvering
- created interference *se* skapad
  - blockeringstid
- critical instant *se* kritiskt
  - ögonblick
- cyclic executive 14
- deadline *se* tidsgräns
- deadline monotonic scheduling
  - 15
- dispatcher 37
- DSP-kort 2
- dynamiska algoritmer 14, 17
- earliest deadline first 17
- exekverande process 7
- explicit deklaration 9, 10
- feltolerant process 13
- fin granularitet 8
- fork och join 9
- fördröjning 15
- givare 1
  - accelerometer 1
  - gyro 1
  - hjulpulsräknare 1
  - inklinometer 1
  - laserkamera 1
- granularitet 8
  - fin 8
  - grov 8
- granularity *se* granularitet

- grov granularitet 8
- gyro 1
- hjälpulsräknare 1
- hård process 13
- initialisering 8
- initialization *se* initialisering
- inklinometer 1
- interference *se* blockeringstid
- IRI 2
- kritiskt ögonblick 15
- körbar process 7
- laserkamera 1
- least slack time 17
- level *se* nivå
- längsprofil 2
- maxtid 10, 11
- mjuk process 13
- mätapplikationer 2
  - IRI 2
  - längsprofil 2
  - sprickmätning 2
  - textur 2
  - tvärprofil 2
  - vattendjup 2
- nivå 8
- nyttjandegrad 12
- offset *se* fördröjning
- optimal priority assignment 15, 31
- periodisk process 13
- periodtid 13
- preemptive preference priority-based 14
- preemptive scheduling 7
- process 7
  - aperiodisk 13
  - attribut 13
    - beräkningstid 13
    - fördröjning 15
    - periodtid 13
    - tidsgräns 13
  - feltolerant 13
  - hård 13
  - mjuk 13
  - periodisk 13
  - sporadisk 13
  - tillstånd 7
    - exekverande 7
    - körbar 7
    - väntande 7
    - övergångar 7
- rate-monotonic scheduling 14
- ready *se* körbar process
- realtidssystem 10
- remaining interference *se* återstående blockeringstid
- representation 8, 9
- running *se* exekverande process
- samtidig exekvering 6
  - granularitet 8
  - initialisering 8
  - nivå 8
  - representation 8, 9
  - struktur 8
  - terminering 8, 9
- scheduling *se* schemaläggning
- schemaläggning 7, 13
  - algoritmer
    - best-effort scheduling 17
    - cyclic executive 14
    - deadline monotonic scheduling 15
    - dynamiska 14, 17
    - earliest deadline first 17
    - least slack time 17
    - optimal priority assignment 15, 31
    - preemptive preference priority-based 14
    - rate-monotonic scheduling 14
    - statiska 14

- 
- kritiskt ögonblick 15
  - preemptive scheduling 7
  - slacktid 18
  - skapad blockeringstid 34
  - slacktid 18
  - spatialdomän 20
  - sporadisk process 13
  - sprickmätning 2
  - statiska algoritmer 14
  - structure *se* struktur
  - struktur 8
  - termination *se* terminering
  - terminering 8, 9
  - textur 2
  - tidsdomän 20
  - tidsgräns 11, 12, 13
  - time-out *se* maxtid
  - tvärprofil 2
  - waiting *se* väntande process
  - vattendjup 2
  - väntande process 7
  - återstående blockeringstid 34

**Avdelning, Institution**

Division, department

Department of Computer and  
Information Science

Institutionen för datavetenskap

Datum

Date

1997-03-04

**Språk**

Language

 Svenska/Swedish Engelska/English \_\_\_\_\_**Rapporttyp**

Report: category

 Licentiatavhandling Examensarbete C-uppsats D-uppsats Övrig rapport \_\_\_\_\_**ISBN****ISRN****Serietitel och serienummer**

Title of series, numbering

**ISSN**

LiTH-IDA-Ex-9709

**URL för elektronisk version****Titel**

Title

Kravanalys och implementering av ett realtidssystem med höga dataprestanda

Requirement analysis and implementation of a high performance real-time system

**Författare**

Author

Patrik Lantto

**Sammanfattning**

Abstract

Denna rapport presenterar ett examensarbete utfört på OPQ systems AB i Linköping. OPQ konstruerar system för mätning av ett antal olika parametrar på vägar. Beräkningarna görs i realtid, huvudsakligen på signalprocessorer.

Nya beräkningar med högre prestandakrav än dagens system kan tillgodose, och önskemål om att bättre kunna utnyttja gemensamma beräkningar för olika mätparametrar gör att OPQ undersöker hur ett nytt system skulle kunna konstrueras.

Syftet med examensarbetet är att undersöka vilka krav som bör ställas på ett nytt system, främst utgående från teorin om realtidssystem. Därefter föreslås lösningar som uppfyller kraven, och delar av systemet implementeras på en av OPQ utvald processor.

Resultatet visar att den utvalda processorns prestanda är fullt tillräckliga för att kunna ersätta dagens system och uppfylla nya önskemål, och att de implementerade lösningarna uppfyller de krav som ställts.

Även om en annan processor väljs bör ett nytt system utvecklas baserat på teorin om realtidssystem, och då är de förslagna lösningarna en bra utgångspunkt.

**Nyckelord**

Keywords

Realtidssystem, signalprocessor, DSP