Datalogisk Institut
Københavns Universitet
Projekt nr. 74-12-3
maj 1975

MIK - et korutineorienteret styresystem til en mikrodatamat

Resumé:

MIK er et styresystem til en mikrodatamat. Systemet afvikler prioriterede reentrante korutiner og omfatter mekanismer til indbyrdes synkronisering og kommunikation, synkronisering med sand tid, og retningslinjer for behandling af ydre enheder. MIK er et skelet, som kan udbygges til forskellige dedikerede systemer. Det er implementeret på en intel8080 centralenhed.
Rapporten behandler de underliggende problemer i parallel programmering, gennemgår 4 ældre lignende systemer og beskriver MIK som system, over for brugere og med henblik på vedligeholdelse.

Bodil Schrøder

Indholdsfortegnelse

I.1 Indledning

Formålet med dette projekt er at lave et styresystem til en mikrodatamat. Mikrodatamaterne vinder større og større udbredelse til overvågning og styring af maskiner, processtyring, behandling og transmission af information o.s.v. Det er nærliggende at undersøge i hvor høj grad de teknikker, som benyttes, når man bruger minidatamater til dedikerede anvendelser, med rimelighed kan overføres til mikrodatamaterne.

Først må man se på hvad en mikrodatamat er. Der er to synsvinkler hvorfra man kan betragte fænomenet:

Den ene mulighed er at betragte mikrodatamaten som en programmerbar elektronisk komponent, hvis funktion for en given anvendelse er bestemt ved det indlagte program. Set ud fra dette synspunkt erstatter mikrodatamaten et antal traditionelle logiske komponenter, og programmering af mikrodatamaten træder i stedet for sammensætning af disse enkeltdele. Programmerne vil ofte være lagret i læselager (ROM eller read-only memory) som er programmeret en gang for alle eller i programmerbart læselager (PROM eller programmable read-only memory), som kan ændres, men ikke ved at mikrocentralenheden skriver i det. Set fra et materielsynspunkt er fordelen at man kan erstatte utallige forskellige komponenter med en enkelt eller nogle få typer mikrodatamater, som nok er mere komplicerede at fremstille, men som til gengæld kan laves i så store mængder, at de bliver billige. Artikler som for få år siden bestod af et stort antal elektroniske komponenter er erstattet af andre, som er mindre og billigere, fordi man har formået at samle komponenterne i integrerede kredse; men hvor dette ikke har været realisabelt, fordi varen ikke fremstilles i tilstrækkelig store mængder til at det har kunnet betale sig at producere de specielle integrerede kredse der var brug for, der kommer mikrodatamaterne nu og åbner de tekniske og prismæssige fordele ved masseproduktion, samtidig med at man ved hjælp af programmer kan honorere specielle krav.

Den anden mulighed er at betragte mikrodatamaterne som egentlige datamater. Mikrodatamaterne kendetegnes som datamater ved at de fylder meget lidt og er meget billige. Som eksempel kan nævnes at man kan få mikrocentralenheder, som kun fylder en enkelt integreret kreds, og som kun koster nogle få tusinde kroner. Til gengæld er der en tendens til at ordrerepertoiret er spartansk, f.eks. er multiplikations- og divisionsordrer noget af et særsyn. Det er ikke til at sige noget generelt om udførsels-

tider, idet de i hvert fald kan svinge med en faktor 100 fra den ene type til den anden. Da mikrodatamaterne er så små og billige er der en tendens til at bruge dem til opgaver, hvor det tidligere har været utænkeligt at anvende datamater enten på grund af pris eller på grund af størrelse. På den anden side er der også mange eksempler på at mikrodatamater kan klare opgaver, som man tidligere har brugt store, dyre datamater til.

Af disse to synspunkter på mikrodatamater er det især det andet jeg vil bygge på. Jeg vil tænke på mikrodatamaten som en datamat, men jeg vil også tænke på at det mikrodatamaten skal bruges til ofte vil være opgaver som tidligere har været klaret ved sammensætning af logiske komponenter. Et typisk eksempel er overvågnings- og styringsopgaver, som imidlertid vil kræve, at datamaten kan klare flere ting på én gang. Dette kan klares ved hjælp af flere mikrodatamater, men i mange tilfælde vil opgaverne ikke være uafhængige, og derfor vil en rimelig løsning kræve samkørsel på mikrodatamaten. Formålet med denne opgave bliver derfor at udvikle et styresystem, så det bliver muligt at lave programmer til en mikrodatamat, så den kan klare flere opgaver på en gang på en forholdsvis simpel måde.

Formålet med denne rapport er dels at beskrive, hvordan jeg har udformet dette styresystem, og dels at begrunde hvorfor jeg netop har gjort det på den måde.
I første omgang vil jeg diskutere de generelle problemer, der opstår, når flere opgaver skal løses samtidig i en datamat. Disse problemer er i udstrakt grad de samme som opstår i et generelt operativsystem til en stor datamat, og jeg har i denne forbindelse grundigt studeret Brinch Hansens bog: Operating System Principles (1). Afsnit II i denne rapport er en gennemgang af forskellige metoder, der kan benyttes, når man vil lave et styresystem af den påtænkte slags.
I afsnit III vil jeg kigge på fire konkrete minidatamatsystemer, som alle har fællestræk med det jeg selv har udviklet. Det drejer sig om DIXI (2), som er et terminalkoncentratorsystem på Datalogisk Instituts PDP 11/20, hvilket er et eksempel på noget mit system eventuelt kunne benyttes til. Det drejer sig om RC 3500 Monitor I og The NBB Semaphore Monitor (3 og 4), som begge er generelle styresystemer til dedikerede anvendelse af bestemte minidatamater, og endelig drejer det sig om VACS (5), som er et styresystem til en Varian-minidatamat, der er tilknyttet Bispebjerg Hospitals

gammakamera.

I afsnit IV følger beskrivelsen af MIK, som er navnet på det styresystem jeg har lavet (et MIkrodatamatorienteret Korutinestyresystem), og som jeg har valgt at kode til en intel8080 centralenhed. Redegørelsen bygger på de teknikker jeg har diskuteret i afsnit II og III, og skulle være tilstrækkeligt både til at gøre det muligt at benytte MIK og til eventuelt at foretage udvidelser eller ændringer. For at give et begreb om hvordan et færdigt system opbygget omkring MIK kan se ud, er der givet et par eksempler i afsnit V.

Afsnit VI er en beskrivelse af den intel8080-simulator, som jeg har skrevet for at kunne afprøve MIK. Simulatoren kører på RC 4000.

II.2 Notation

I hele rapporten har jeg forsøgt at overholde dansk standard, som den er defineret i EDB-ordbogen og forskellige forslag til tilføjelser til denne (19,20 og 21).
Imidlertid har der været en del termer der manglede, og andre som jeg mente dækkede for dårligt. Desuden var der en del udtryk, som jeg aldrig havde hørt før jeg slog dem op, og som jeg derfor vil nævne her for at sikre, at læsere af rapporten ved hvad de står for når jeg bruger dem. Nedenstående er en liste over disse gloser med angivelse af den tilsvarende engelske betegnelse eller en anden form for forklaring:

læselager	ROM eller read-only memory. Lager hvis indhold kun kan læses, ikke ændres. Det er sædvanligt at programmer til mikrodatamater er lagret i læselager.
mikrodatamat	microcomputer eller nogen gange microprocessor. Den sidste betegnelse bør absolut undgås da den også benyttes om mikroprogrammerbare centralenheder.
mikrocentralenhed	centralenheden på en mikrodatamat. I rapporten omtales mikrocentralenhederne intel8080 og intel8008.
dedikeret	et dedikeret system er det samme som et system med formålsbunden drift (MK8).
instruktion	ordre (J9). Den direkte oversættelse af instruction er desværre smuttet imellem et par steder i rapporten.
samkørsel(M50)	multiprogrammering
assembler	indsætter (J29). Jeg mener at det standardiserede danske udtryk er misvisende og bruger derfor konsekvent anglicismen.
processtyring	et tilfælde af tidstro styring (M21).

styresystem	monitor (K7). Jeg har undgået denne glose da jeg opfatter den som engelsk, men jeg fandt den til min overraskelse i EDB-ordbogen.
drivprogram	driver. Oversættelsen er hentet fra (21). Jeg er ikke særlig begejstret for den, men har ikke kunnet finde nogen bedre.
rutine (A51)	bruges flittigt - specielt i følgende forbindelser:
korutine	coroutine. Se iøvrigt II.1 og II.2.A.
subrutine	underprogram (K20). Jeg kan bedre lide subrutine.
afbrydelsesrutine	bruges om den kode der udføres, når der kommer en afbrydelse. Afbrydelsesrutinen er ofte karakteriseret ved at den starter med at gemme centralenhedsstatus, ordretæller og registre, og slutter med at retablere dem.
ventepunkt	se afsnit II.1 og II.2.A.
strækninger	stretches - bruges om koden imellem to ventepunkter i en korutine.
tidskvant	timeslice. Et bedre udtryk på dansk ville være velkomment. Dette er hentet fra (21).
reentrant	der foreligger ingen oversættelse, og jeg synes det lyder meget godt udtalt på dansk. Benyttes om kode der kan udføres af flere processer på en gang, hvilket dels kræver at koden ikke er selvmodificerende, og dels at hver proces arbejder på sine egne variable.
baglås	deadlock. Fortrinlig oversættelse fra (21), den vil jeg gerne slå et slag for.

besked	direkte oversættelse af message. Formel kommunikationsenhed mellem parallelle processer.
tidssignal	oversættelse af timeout, som jeg har mødt flere steder i betydningen et "vækkeur" der ringer når der er gået et vist stykke tid.
cyklisk aktivering	round-robin scheduling. Udtrykket er hentet fra (21) og er temmelig beskrivende.
procesafvikling	scheduling
skrivemaskineterminal	den bedste oversættelse jeg har kunnet finde på for en teletype.
taktsignal (C75)	klokpuls. Et signal der kommer fra en pulsgenerator med faste mellemrum.
semafor	oversættelse af semaphore (se II.2.C.)
generel semafor	operationer wait/signal
binær semafor	operationer lås/åbn
besked semafor	operationer receive/send
trafiklys semafor	operationer passér/start/stop
postkasse semafor	operationer vent/tøm

Ovennævnte 6 linjer er hvad jeg har fået ud af et virvar af forskellige betegnelser. Betydningen omtales i II.2.C.
Udtrykket generel semafor må slås bravt med simpel semafor og blot semafor, mens wait/signal lader til at være noget der ligner standard.
Binær semafor er en standardbetegnelse, mens lås/åbn (lock/unlock) stadig må slås med de fæle "P" og "V" operationer.
Beskedsemaforen er min egen betegnelse, som skyldes at næsten alle de steder jeg har mødt mekanismen kædes den sammen med begrebet "message"; der er dog ingen af dem der kalder den for "message-semaphore". De gode og beskrivende navne på operationerne "receive" og "send" har jeg fra Brinch Hansen.
Trafiklys og postkasser med tilhørende operationsnavne er mine egne, som dels er beskrivende og dels er tilpas forskellige fra de øvrige.

Jeg har bibeholdt de engelske betegnelser wait/signal og receive/send i overensstemmelse med navnene på systemkaldene i MIK, som er kodet og kommenteret på engelsk.

Al skitsekode er holdt i pseudo-SIMULA, men kan efter behag opfattes som psedo-algol undtagen i programdokumentationen, hvor SIMULAS klasser benyttes.

I.3 Projektets forløb

MIK-projektet startede i efteråret 1974 inspireret af Edda Sveinsdottir, og kom rigtigt i gang ca. 1. december.

Den første måneds tid forløb med overvejelser over hvilken mikrodatamat styresystemet skulle laves til og i sammenhæng hermed undersøgelser af hvilket støtteprogrammel - assemblere og simulatorer - til mikrodatamater, der fandtes på forskellige anlæg. Jeg forestillede mig en overgang at kode i det PL/I-lignende programmeringssprog PL/M til mikrodatamaten MCS-8, men det viste sig at MCS-8 på forskellige punkter var for primitiv.

Parallelt hermed startede jeg et grundigt studium af litteratur om datamater med flere centralenheder (multiprocessorer); jeg læste en bog samt ca. 15 artikler, idet det var min tanke at udforme styresystemet med henblik på flere mikrocentralenheder. Imidlertid så jeg mig senere nødsaget til at afskære den del af projektet, dels fordi det ikke faldt helt så simpelt ud, som jeg i første omgang havde forestillet mig, og dels fordi arbejdet var rigelig stort endda.

Min inspiration var i første omgang nogle noter fra et mikrodatamatkursus på Datalogisk Institut i foråret 1974, et specialearbejde af Per Gade Christensen om styresystemer til afvikling af tidstro programmer på minidatamater fra 1972 (6) samt DIXI (2) og VACS (5), som er omtalt i indledningen, fra hhv. 1973 og 1974. Kort efter fik jeg fat i Brinch Hansens bog om operativsystemer (1), og udfra dette gik jeg i gang med at udvikle MIK i slutningen af januar i år. Efter ca. 1 måned blev jeg tilrådet at se på beskrivelserne af The NBB Semaphore Monitor (4) og RC 3500 Monitor 1 (3), og de faldt i forbløffende grad i tråd med en del af de overvejelser jeg selv havde været igennem, samtidig med at de gav mig flere gode ideer til ændringer i det jeg allerede havde kodet. Den næste måned var helliget den resterende del af kodningen samt indkøring og afprøvning, som dog først kom i gang efter at jeg havde brugt et par uger til indkøring og afprøvning af simulatoren. I slutningen af marts begyndte jeg at skrive denne rapport, hvori jeg udover at beskrive MIK har forsøgt at sammenligne forskellige teknikker, som kunne have været anvendt.

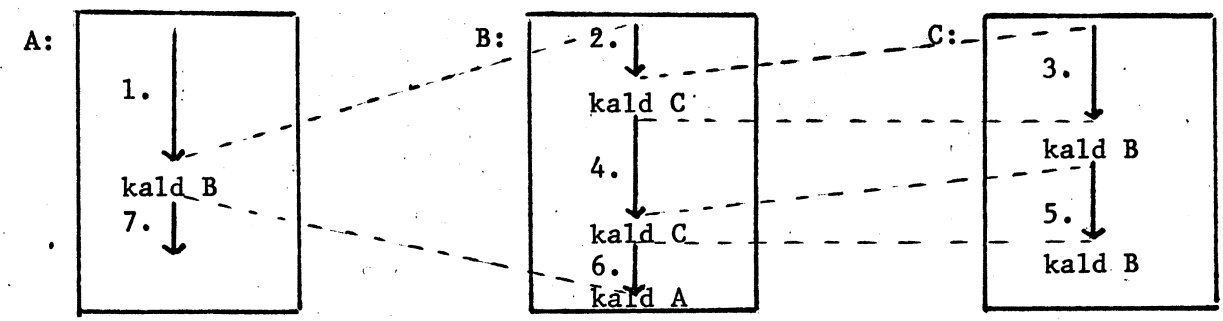
En ting jeg har savnet meget har været fornuftige litteraturhenvisninger. Under den sidste fase af rapportskrivningen har jeg fra forskellig side fået adskillige gode råd om hvad jeg også burde læse om, med det har været for sent til at jeg har kunnet udnytte det. For fuldstændighedens skyld angiver jeg nogle titler på ting, som jeg kun har læst meget kursorisk - det drejer sig om (13),(14),(15) og (16). (15)-(Intern struktur af Boss 2) lader til at være kilden til en del af de grundlæggende begreber, men det fandt jeg først ud af via en reference i (13).

II. GENERELLE BETRAGTNINGER OVER PARALLEL PROGRAMMERING

II.1 Korutiner

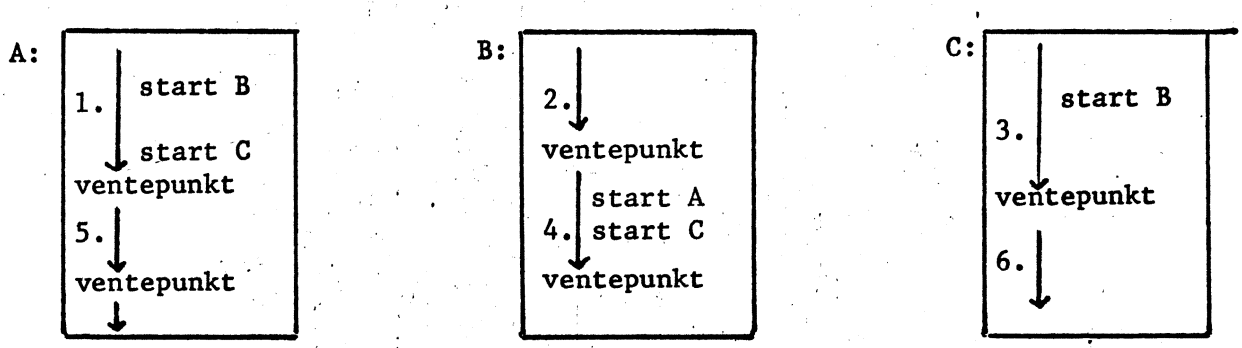
Korutiner og subrutiner:

Det oprindelige korutinebegreb har jeg fra Knuth (7), som anfører Conway (8) som kilde. Korutiner betegner her en udvidelse af subrutinebegrebet, idet korutiner er delprogrammer, der kan kalde hinanden, og som når de bliver kaldt fortsætter på det sted de var nået til, som illustrationen søger at vise:



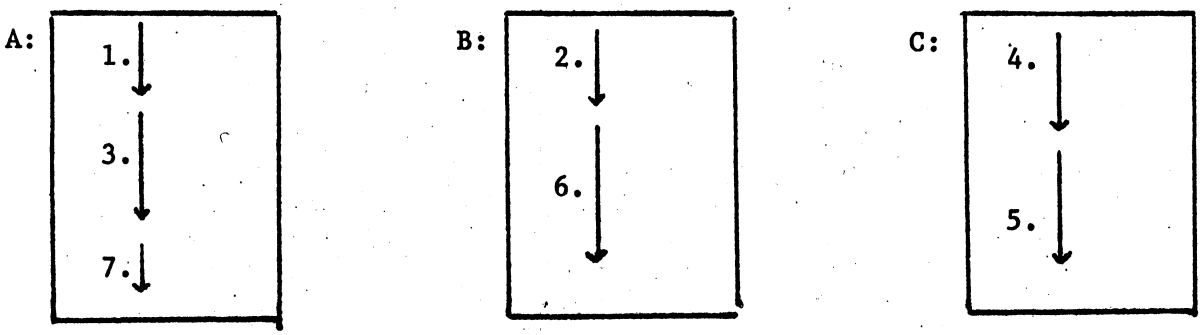
Samme måde at opfatte korutiner på møder man i Simula 67 (9), hvor processer, der anvender "call" og "resume" er korutiner i denne forstand.

Både i Simula og hos Knuth møder man også korutiner, som kan starte flere andre korutiner, inden de selv stopper (Simula: "activate P after current" og "activate P delay 0"; Knuth: Elevatorsimulation). De steder i koden hvor korutinerne afløser hinanden kaldes ventepunkter. Udvælgelsen af hvilken korutine der skal køre, når en korutine har nået til et ventepunkt, bliver overladt til en centrallogik. Illustrationen viser et eksempel:



Disse korutinebegreber er imidlertid ikke dækkende for det begreb der anvendes i de systemer, som omtales i det følgende. Det har ikke været mig

muligt at spore deres oprindelse, men evt. stammer de fra BOSS 2 (15). Korutinebegrebet er her blevet mere abstrakt, idet et ventepunkt svarer til en betingelse, som normalt repræsenteres ved hjælp af en semafor. En korutine standser, når den når til en sådan betingelse, og fortsætter igen, når betingelsen bliver opfyldt. Det vil normalt ske på et eller andet tidspunkt på grund af noget en anden korutine udfører, evt. i forbindelse med afslutningen af en transport på en ydre enhed. Det der adskiller disse korutiner fra de andre typer er at korutinerne ikke ved om de operationer de udfører eventuelt starter andre korutiner. Nedenstående er det nærmeste jeg kan komme en fornuftig illustration:



Korutiner og parallelle processer:

Udtrykkene korutiner og parallelle processer bruges nogen steder i litteraturen om temmelig forskellige begreber, mens de andre steder nærmest behandles som synonymer. Dette skyldes at man udfra to forskellige målsætninger når frem til de samme metoder.

Samkørsel er en metode til at udnytte datamater effektivt ved at lade dem udføre flere programmer parallelt. Ideen er at mens nogle af programmerne venter på ydre enheder, kan et program benytte centralenheden, som normalt vil være den ressource, som bestemmer datamatens udnyttelsesgrad.

Korutiner er en måde at opfatte programmer på, hvor større programmer opdeles i delprogrammer - korutiner- som kun er forbundet løst med hinanden, og som hver for sig er lette at overskue og dermed gør, at det samlede program bliver simplere at udvikle og indkøre.

Nu behøver samkørende programmer på den ene side ikke at være uafhængige, og på den anden side kan programmer opdelt i korutiner ofte bringes til at køre effektivt ved at udføre nogle af korutinerne parallelt. Herved er man nået frem til begrebet parallelle processer, som benyttes om programmer eller delprogrammer, som udføres parallelt, og som kan være helt uafhængige eller have behov for indbyrdes kommunikation.

De anvendelser, som specielt har interesse her, er systemer, hvor flere forskellige, men ikke nødvendigvis uafhængige opgaver skal udføres samtidig, og til dette formål er begrebet parallelle processer ideelt. Senere vil jeg vende tilbage til hvordan det ser ud, når de indgående processer er korutiner.

II.2 Diskussion af de enkelte problemer

Ved en proces vil jeg forstå den sekventielle udførelse af et program, som kan indeholde løkker, betingede hop, subrutinekald osv. Betegnelsen styresystem (ofte kaldet en monitor) dækker over den samling rutiner, der skal sørge for at få maskinen til at optræde, som om alle processerne blev udført samtidig. Disse definitioner er hentet fra beskrivelserne af RC3500 Monitor 1 (3) og The NBB Semaphore Monitor (4), hvor de findes næsten enslydende. For at gøre det muligt for styresystemet at administrere afviklingen af de enkelte processer og kommunikationen mellem dem, findes der en procesbeskrivelse for hver proces. Normalt vil en del af processerne være i en tilstand, hvor de venter på at en eller anden betingelse skal blive opfyldt; det kan f.eks. være en ydre enhed eller en anden proces, der skal sende et signal. De øvrige - som vil blive omtalt som de aktive processer - vil stå parat til at få rådighed over centralenheden, når den bliver ledig.
De problemer der opstår i et sådant system vil jeg indordne under følgende kategorier:

Udelelighedsproblemer
Synkronisering og informationsudveksling
Procesafvikling og prioritering
Drivprogrammer - styring af ydre enheder
Tidsmåling

Resten af dette afsnit vil rumme en diskussion af hvert af punkterne. I næste afsnit vil jeg se hvordan problemerne er angrebet i nogle konkrete systemer, og denne analyse bruges så som udgangspunkt for udformningen af mit eget system.

II.2.A Udelelighedsproblemer

Når flere programmer skal køre parallelt, og der kun er én centralenhed på datamaten, må de enkelte processer nødvendigvis afgive centralenheden til hinanden nu og da.
Afgivelsen af centralenheden kan foregå på to måder:
Den ene mulighed er at processen eksplicit kan angive, hvor det er tilladt at afbryde den. Sådanne steder vil i det følgende blive kaldt ventepunkter, og her vil det være veldefineret, hvad der kan være ændret i registre, centralenhedsstatus og lager, når processen atter får rådighed over centralenheden.
Den anden mulighed går ud på at afbrydelser er noget der kommer udefra, uden at processen har kontrol over det. Imidlertid skal processen have mulighed for at fortsætte på samme måde, hvadenten den har været afbrudt eller ej. Derfor må afbrydelser kun ske på veldefinerede tidspunkter, hvor det er muligt at gemme centralenhedens tilstand (registre, statusbits, ordretæller), så den kan retableres, når den intetanende proces får lov til at køre igen. I praksis er dette altid klaretved at afbrydelser kun kan finde sted imellem udførelsen af to ordrer.

Det følgende eksempel viser at det ikke er ligegyldigt, hvornår en proces bliver afbrudt:

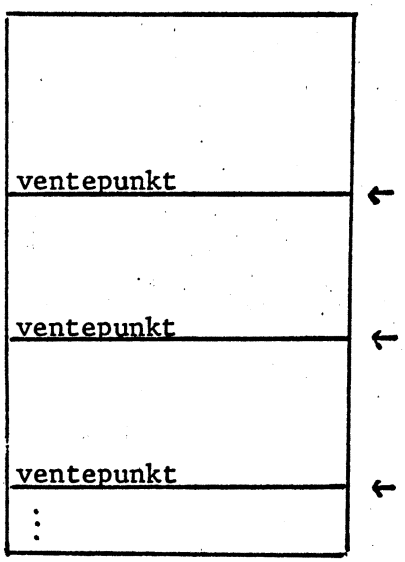
Eks. 1.
Proces A aflæser nu og da en variabel "ur", som angiver hvad klokken er, og som består af de to variable "time" og "minut". Proces B tæller med 1 minuts mellemrum "ur" op med 1 minut.
Nu kan man forestille sig, at A vil aflæse "ur" på et tidspunkt, hvor klokken er 16:59. A når at aflæse "time", som er 16, men på det tidspunkt afbrydes A, og B kommer til og tæller "ur" op, så det nu viser 17:00. Når A kommer til igen aflæser den "minut" og tror derpå at klokken er 16:00.

Eksemplet illustrerer, at hvis flere processer arbejder med de samme data, må hver af dem kunne sikre sig, at de i en periode ikke ændres af andre. Fuldstændig det samme problem møder man, når to processer ønsker at bruge den samme ydre enhed. Det er f.eks. umuligt at udskrive to tekster samtidig på den samme terminal. I det følgende vil jeg omtale alle sådanne tilfælde som problemer med at opnå udelelig tilgang til fælles ressourcer.

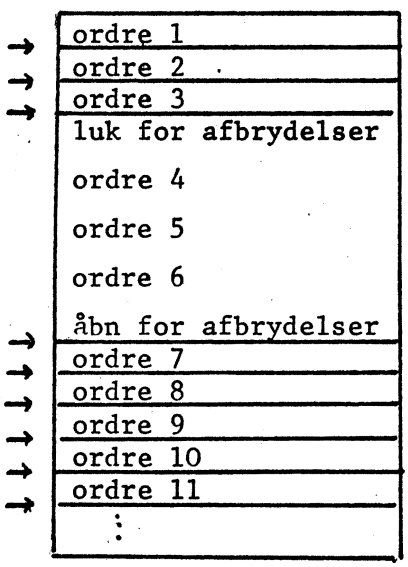
Hvis man benytter ventepunkter er man sikker på at andre ikke kommer til de fælles ressourcer, så længe man befinder sig mellem to ventepunkter. Hvis man benytter metoden med udefra kommende afbrydelser, må man klare sine udelelighedsproblemer på anden vis. En mulighed er at lukke for alle afbrydelser i datamaten, når processen kommer til et stykke kode, som skal udføres udeleligt, og tilsvarende åbne igen, når man er færdig med at benytte de fælles ressourcer. En anden mulighed, som ikke er helt så drastisk, er at benytte semaforer til formålet; dette omtales dog først i afsnit II.2.C.

Man kan illustrere de to metoder således:

1. Metode med ventepunkter:



2. Metode med lukning for afbrydelser:



Figuren viser at koden i begge tilfælde opdeles i strækninger mellem ventepunkter. I 1. er ventepunkterne eksplicit angivet, mens der i 2. er implicitte ventepunkter mellem hver ordre, undtagen når der eksplicit er spærret for afbrydelser. Med hensyn til udelelighed er der altså ingen principiel forskel på hvad de to metoder kan benyttes til, men i praksis er der naturligvis alligevel forskel. For nogen anvendelser er det praktisk at udelelighed er noget man næsten automatisk har, mens det andre steder ikke betyder så meget, om man skal anstrenge sig lidt for at sikre udelelig adgang til visse ressourcer, hvis man så til gengæld kan skifte fra den ene proces til den anden næsten altid.

Parallelle processer med eksplicit angivne ventepunkter er hvad jeg i det følgende vil forstå ved korutiner.

Begrebet korutiner kan imidlertid forfines. Det viste sig at afbrydelser har betydning, hvis en proces bliver afbrudt, så en anden proces får adgang til en ressource, som den første er i gang med at benytte. Jeg vil derfor snakke om at et sæt af processer er indbyrdes korutiner, når hver af disse processer kun kan komme til at køre, hvis de andre i sættet befinder sig i et ventepunkt, uanset at andre processer kan afbryde mellem ventepunkterne. Specielt kan en proces, der aktiveres ved en afbrydelse fra en ydre enhed, godt få lov til at afbryde en korutine, uden at der opstår udelelighedsproblemer, hvis blot den ikke har ressourcer fælles med korutinen. Da den afbrudte korutine ikke befinder sig i et ventepunkt ved afbrydelsen, skal den have lov til at fortsætte før nogen af de processer, der tilhører samme sæt af korutiner, bliver startet. Ved at betragte korutiner på denne måde får man i stedet for en egenskab ved alle processerne i et system en ækvivalensrelation i mængden af processer. Vi skal siden se hvordan dette kan benyttes til at arbejde med korutinesæt på forskellige prioritetsniveauer.

II.2.B. Procesafvikling

Når det er afgjort, hvor det er tilladt at afbryde en given proces, skal man til at finde ud af, hvornår den faktisk skal afbrydes, og hvilken anden proces der skal til i stedet for.

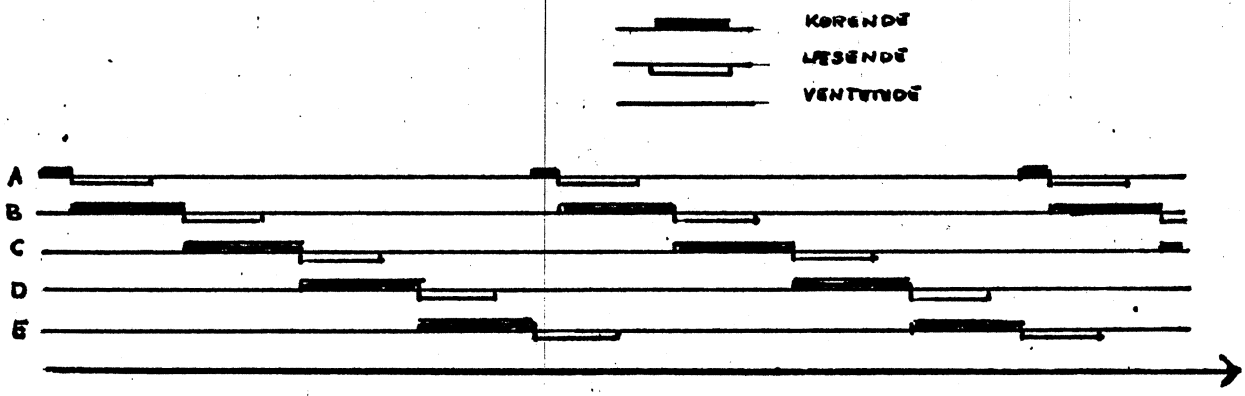
En meget anvendt metode er at give hver proces et tidskavnt efter tur. Metoden kan varieres på flere måder. Tidskvantets længde behøver ikke at være konstant, og processerne får ikke nødvendigvis tildelt køretid i den rækkefølge de har anmodet om den. Strategien med hensyn til hvad der skal ske, når en proces ikke ønsker mere køretid og der er mere tilbage af dens tidskvant, er ikke ens fra system til system, ligesom afbrydelser fra ydre enheder, som skal behandles øjeblikkelig, kan give anledning til forskellige fremgangsmåder, når det skal indordnes i tidsdelingssystemet.

Her vil jeg kigge på en anden løsning, som går ud på at lade hver proces køre så længe den vil. Hvis der er tale om et dedikeret system, hvor processerne er beregnet på at køre sammen, kan man gå ud fra, at ingen proces vil monopolisere maskinen.
Man kan derfor evt. give hver proces lov til at køre til den ikke vil mere, og derefter vælge at give centralenheden til den proces, der har ventet længst. Denne løsning har helt oplagt mange fordele. Administrationen er meget simpel og optager meget lidt af den samlede tid, alle processer kommer til nu og da, og de kører udeleligt i forhold til hinanden. Denne strategi benyttes f.eks. i DIXI (2). Her sender man information frem og tilbage mellem terminaler og datamatcentre. Hver proces foretager omtrent følgende: Behandl den information, der er modtaget - send den videre - vent på ny information. Det er oplagt mest retfærdigt, at processerne behandles efter tur, så systemet nok kører langsomt, når det har meget at lave, men lige langsom for alle; i modsat fald kunne man risikere, at nogen terminaler kom igennem med det samme, når systemet blev belastet, mens andre gik helt i stå.

Nu er det imidlertid ofte sådan at nogen processer er vigtigere end andre:

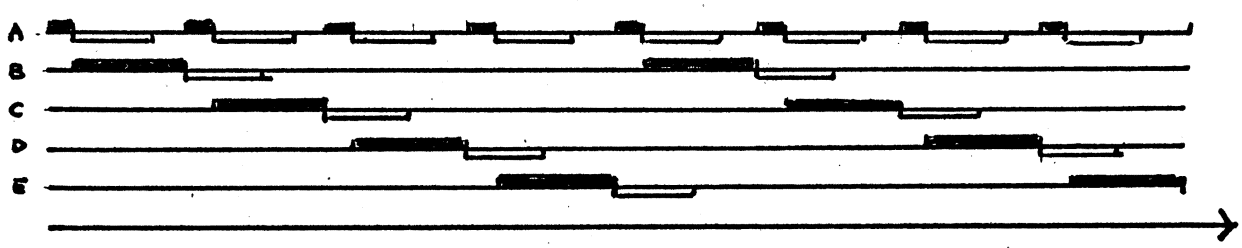
Eks. 2:
Vi ser på et system med 5 processer A, B, C, D og E. Alle processerne kører i en uendelig løkke, hvor de først læser nogle data og derpå be-

handler dem. Læsetiden, hvor de ikke optager centralenheden, er den samme for alle (t_1), men A's behandlingstid (t) er kun en femtedel af den de andre bruger (T). Hvis processerne afvikles således at den proces der har ventet længst kommer til, når centralenheden bliver ledig, kommer det til at se således ud:



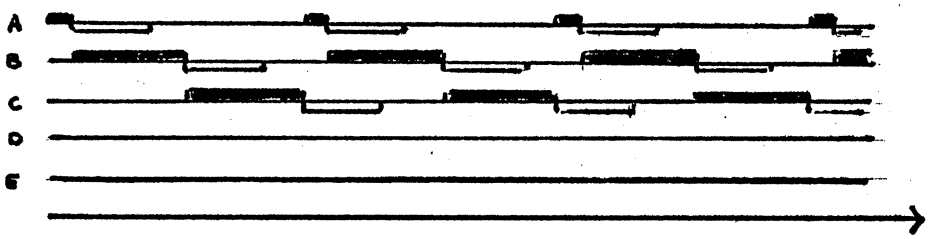
Afviklingsrækkefølgen er: A,B,C,D,E,A,B,C,D,E,A,B,...

Hvis A derimod altid får lov til at køre, når den har behov for det, kommer det til at se sådan ud:



Afviklingsrækkefølgen er nu: A,B,A,C,A,D,A,E,A,B,A,C,A,..., og man ser at A er kommet til at køre næsten fem gange så hurtigt, uden at de andre processer forsinkes væsentligt.

Metoden i eksempel 2 implementeres ved at give hver proces en prioritet og altid vælge den proces der skal køre, som den med højest prioritet, som har ventet længst.
Hvis f.eks. proces B og C også fik høj prioritet i det foregående eksempel, ville de to sammen med A skiftes til at sidde på centralenheden, mens D og E aldrig kom til:



Afviklingsrækkefølge: A,B,C,A,B,C,A,B,...

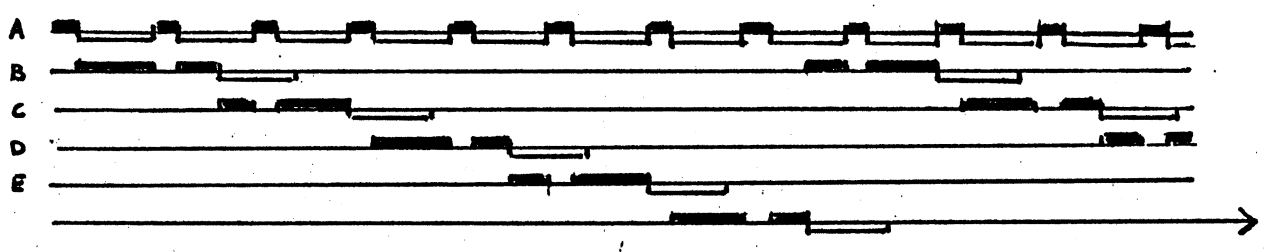
Dette er et eksempel på at lavt prioriterede processer kan blive forsinkede i det uendelige, hvis systemet er overbelastet, eller hvis valget af prioriteter er forkert.

Hvis der ikke er tale om et dedikeret system, men f.eks. et generelt operativsystem, kan ovenstående metode ikke benyttes. Dels kan man ikke tillade at en proces kører så længe den har lyst, for det kunne jo være et fejlbehæftet program, der gik i uendelig løkke, og derved blokerede hele maskinen. Dels kan man ikke lade processerne bestemme deres egne prioriteter, da man risikerer, enten at alle processerne kører med højeste prioritet, og da er fordelene ved prioritering væk, eller også at nogle få kører på højeste prioritet og tilsammen udelukker alle andre.

Vi har altså set at tidskvantmetoden ikke er strengt nødvendig til dedikerede systemer, men hermed ikke sagt, at den ikke kan bruges. Man kan forestille sig en proces, som ikke er særlig vigtig at få færdig, og som foretager nogle langsommelige beregninger, hvor det ikke er nødvendigt for den at vente på ydre enheder eller andre processer. Man kan lade den køre på lavest mulig prioritet, men det er ikke nok, hvis den netop er begyndt at køre, når en vigtigere proces bliver aktiv. Man kan klare problemet ved at processen nu og da laver et ventepunkt,hvor den siger: "Jeg er ikke færdig, men hvis der er andre der vil til, må de godt komme foran". Det kan imidlertid være besværligt at indsætte disse ventepunkter på de rigtige steder, og i nogen tilfælde vil det være en fordel at have tidskvantmetoden til at klare den slags problemer.

I det foregående gik vi ud fra at vi havde prioriterede korutiner i den forstand, at prioriteringen kun fik betydning, når en proces frigav centralenheden, og en ny skulle vælges blandt de aktive. Imidlertid kunne

man også forestille sig at en proces med højere prioritet altid kom til at køre frem for en med lavere. Hver gang en ny proces bliver aktiv, undersøger man hvem der er højest prioriteret og giver den proces lov til at køre. På denne måde vil man ofte komme til at afbryde en proces før den er færdig, og hvad skal man så gøre når der atter skal vælges en proces på pågældende prioritetsniveau - skal man vælge den der har kørt uden dog at blive færdig, eller skal den bagest i rækken? Hvis man vælger at tage den der allerede har kørt kommer vores eksempel med proces A på højere prioritet end de øvrige til at se således ud:



Ved at lade processerne afbryde hinanden i utide har vi ødelagt korutine-egenskaben, men hvis vi sørger for at en proces altid får lov til at gøre sig færdig, inden en proces med samme prioritet kommer til - altså den første af de to ovennævnte muligheder - opnår vi at processer på samme prioritetsniveau kører som korutiner. Problemer med fælles ressourcer bliver således kun aktuelle, hvis det drejer sig om processer med forskellig prioritet.

Denne metode er ikke helt så simpel som de to foregående, idet man kommer til at bruge lidt mere tid på administration. Til gengæld opnår man, at højere prioriterede processer overhovedet ikke forsinkes af de lavere prioriterede. Hvis processerne har fælles ressourcer kan man dog alligevel risikere at en højere prioriteret proces må vente på en lavere prioriteret i denne forbindelse.

II.2.C Kommunikation mellem processer

Processers behov for at kommunikere falder i to punkter:

1. Synkronisering

Denne form for kommunikation er f.eks. nødvendig, hvis en proces sørger for indlæsning af data, som en anden proces skal viderebehandle. I så fald må den anden proces på en eller anden måde have besked om at den anden har data klar.

2. Udelelig tilgang

Som vi har set i afsnittet om udelelighed, kan der opstå problemer, når flere processer har ikke-udelelig adgang til de samme ressourcer. Det blev omtalt, at det kunne klares ved hjælp af eksplicitte ventepunkter eller midlertidig lukning for afbrydelser. Det er imidlertid ret voldsomt at lukke for alle afbrydelser, hvis det f.eks. kun drejer sig om to processer, der nu og da opdaterer den samme variabel. En ydre enhed, som f.eks. en skrivemaskineterminal, kan normalt kun benyttes af én proces ad gangen, så det er rimeligt at sikre den enkelte proces udelelig brug af den, men det ville være helt uacceptabelt om processen skulle forhindre alle andre processer i at komme til, mens den skrev eller læste, idet en enkelt linje nemt kunne tage flere sekunder, hvor centralenheden for det meste bare stod og ventede. Vi vil derfor se på andre metoder til at sikre udelelig tilgang til ressourcer.

Semaforer:

Det har vist sig at begge problemer - synkronisering og udelelig tilgang - kan løses ved hjælp af semaforer. Da en semafor imidlertid ikke er noget entydigt begreb, vil jeg tillade mig at bruge betegnelsen om en hel klasse af mekanismer, som benyttes af processer til at fortælle hinanden, om de uhindret kan fortsætte fra et givet punkt i deres kode, eller om de skal vente til en passende betingelse er blevet opfyldt. I dette afsnit vil jeg kigge på 5 forskellige semafortyper, som jeg har mødt i de systemer jeg har beskæftiget mig med. Først vil jeg definere hver enkelt type og undersøge hvad den er velegnet til. Dernæst vil jeg se på hvilke semafortyper, der kan erstatte hinanden.

Generelle semaforer

Dette er den oprindelige semafor, indført i 1965 af Dijkstra (10). Semaforen består af et heltal S, som kan antage værdierne 0,1,2,3,... samt en kø af processer, der evt. kan være tom. Der findes to udelelige operationer på semaforen "signal" og "wait" med følgende virkning:

signal: Hvis proceskøen er tom: S := S + 1

ellers: Processen forrest i køen aktiveres

wait: Hvis S er 0: Den proces, der udfører wait-operationen, sættes i kø og fortsætter først, når den bliver aktiveret ved at en anden proces kalder "signal".

ellers: S := S - 1

S er altid 0 når proceskøen ikke er tom. Evt. kan semaforen implementeres så S kan antage negative værdier svarende til antallet af ventende processer, men det giver ingen principiel forskel.

Anvendelser:

Eksempel 1. Synkronisering:

Proces A skal læse data ind til et område i lageret, hvorefter proces B skal benytte disse data til nogle beregninger. Når B er færdig skal A læse nye data ind osv.:

proces A:

```
A: -
   wait(R);
   indlæsning af data;
   signal(S);
   -
   goto A;
```

proces B:

```
B: -
   wait(S);
   anvendelse af data;
   signal(R);
   -
   goto B;
```

Den generelle semafor R er initialiseret til 1, hvilket skal betyde, at dataområdet er klar til at A kan foretage sin indlæsning. Den generelle semafor S er initialiseret til 0 for at vise, at der ikke ligger data parat til B's beregninger. Når A har kaldt wait(R) har R værdien 0 indtil B kalder signal(R), svarende til at A må læse netop én gang, imellem hver gang B foretager sine beregninger. Tilsvarende har S værdien 1, når der ligger data parat til B.

Eksempel 2. Udelelighed:
Processerne A og B må ikke bruge skrivemaskineterminalen samtidig. Semaforen TTY er initialiseret til 1, således at den proces, der kommer først, får skrivemaskineterminalen først, mens den anden må vente:

proces A:

```
A: =
   wait(TTY);
   skriv på skrivemaskine-
            terminalen;
   signal(TTY);
   =
   goto A;
```

proces B:

```
B: =
   wait(TTY);
   læs fra skrivemaskine-
            terminalen;
   signal(TTY);
   =
   goto B;
```

Man bemærker forskellen mellem de to eksempler: I eks. 1 kommer A og B til at køre skiftevis i modsætning til eks. 2, hvor den ene proces godt kan bruge skrivemaskineterminalen mange gange uden at den anden har brugt den i mellemtiden. I øvrigt kan eks. 2 umiddelbart udvides til flere processer.

Eksempel 3:
En ydre enhed laver afbrydelser, som skal forårsage udførelsen af et stykke kode - det kan f.eks. være en geigertæller, der har talt et vist antal partikler. Afbrydelserne kan risikere at komme så tæt i en periode, at man ikke kan nå at udføre hele kodestykket, og det er ikke reentrant. Denne opgave kan klares ved hjælp af en generel semafor S således:

afbrydelsesrutine:

```
signal(S);
```

proces K:

```
K: wait(S);
   den kode der skal udføres
   for hver afbrydelse;
   goto K;
```

Hvis man er sikker på at signal(S) altid tager så kort tid, at det kan nås mellem to afbrydelser, vil værdien af semaforen altid angive, hvor mange gange koden mangler at blive udført, og hvis der så blot er tid nok til at udføre kodestumpen det nødvendige antal gange i det lange løb, er problemet klaret på denne måde.

Binære semaforer

En binær semafor beatår af en variabel, der kan antage de to værdien "åben" og "låst", samt en proceskø, der evt. kan være tom. Der findes to udelelige operationer på semaforen:

operation:	tilstand:	virkning:
åbn	"åben"	ingen effekt
	"låst"	hvis proceskøen er tom ændres tilstanden til "åben", ellers aktiveres processen forrest i proceskøen
lås	"åben"	tilstanden ændres til "låst"
	"låst"	den proces der foretager oparationen sættes i kø

Hvis man kigger nærmere efter ligner en binær og en generel semafor hinanden en del. "Åbn" og "lås" svarer til "signal" og "wait"; "låst" svarer til S= 0, mens "åben" svarer til s > 0. Man kan sige at en binær semafor er en generel semafor, hvor tilstanden "S = 1" er absorberende, idet flere kald af "signal" efter hinanden i denne tilstand ikke får nogen virkning, som de ville have på en almindelig generel semafor.

Anvendelser:

Som det fremgår af ovennævnte bemærkning minder de binære og de generelle semaforer en del om hinanden. Hvis den generelle semafor aldrig risikerer at antage værdier, som er større end 1, kan man derfor lige så godt benytte en binær. Dette var tilfældet i eksempel 1 og 2, men ikke i eksempel 3, hvor afbrydelsesrutinen kan nå at kalde "signal(S)" flere gange inden processen kommer til, og antallet af kald bliver netop repræsenteret ved den generelle semafors værdi. Det modsatte - altså et eksempel, hvor den binære semafor er mere velegnet end den generelle - skal nok kunne konstrueres, men det har ikke været mig muligt at finde noget, der blot var nogenlunde realistisk.

Besked-semaforer

Semaforen er repræsenteret ved en kø. Denne kø kan indeholde en eller flere processer, en eller flere beskeder eller den kan være tom. En besked er blot nogle data, som kan være et tal, en adresse, en kode eller evt. en sammensætning af disse elementer.

Operationer på beskedsemaforen:

send(besked)	proceskø	- den forreste proces i køen får rådighed over beskeden og bliver aktiveret
	beskedkø	- beskeden sættes i kø
	tom kø	- beskeden sættes i kø
receive	proceskø	- den kaldende proces sættes i kø
	beskedkø	- den forreste besked udleveres til den kaldende proces
	tom kø	- den kaldende proces sættes i kø

Når man ser nærmere efter viser det sig, at en generel semafor er en slags specialtilfælde af beskedsemaforen, nemlig svarende til at en besked altid er tom; "send" svarer til "signal", "receive" til "wait" og værdien af semaforen er antallet af køelementer - positivt, hvis det er beskeder, og negativt, hvis det er processer.

Anvendelser:

Eksempel 4. Administration af ressourcer:

I et system er der adskillige processer, der alle nu og da ønsker at skrive på en linjeskriver. Der er to linjeskrivere L1 og L2, og processerne er ligeglade med hvilken linjeskriver de skriver på. Alle processer kan se således ud:

```
=
=
receive(PULJE,linjeskriver);
skriv på den linjeskriver beskeden beskriver;
send(PULJE,linjeskriver);
=
=
```

PULJE er en beskedsemafor, der er initialiseret til at indeholde to beskeder, hvor hver besked er en beskrivelse af en af linjeskriverne.
Man ser at dette er en generalisering af eksempel 2, som drejer sig om generelle semaforer, og som kun gjaldt for en ydre enhed, mens en beskedsemafor lige så nemt kan klare flere fysisk forskellige ydre enheder af samme type.

Eksempel 5.

Dette er en generalisering af eksempel 1. Dengang havde vi et enkelt dataområde, som A læste ind i og B brugte til beregninger. Hvis vi benytter beskedsemaforer kan vi lade FRI være en beskedsemafor, som er initialiseret med en beskedkø, hvor hver besked er adressen på en af N databuffere, mens OPTAGET er en beskedsemafor, der starter med at være tom. Ideen er nu, at proces A tager en buffer fra FRI, fylder data i og sender den til OPTAGET, hvorfra proces B henter den og bruger indholdet til sine beregninger, for derefter at give den tilbage til FRI.
Koden ser således ud:

proces A:

```
A: =
   receive(FRI,bufA);
   indlæs data til bufA;
   send(OPTAGET,bufA);
   =
   goto A;
```

proces B:

```
B: =
   receive(OPTAGET,bufB);
   anvend data fra bufB;
   send(FRI,bufB);
   =
   goto B;
```

Man ser at koden næsten er identisk med den i eksempel 1, men udover at klare enkeltbuffersituationen kan den uden videre klare dobbelt, tredobbelt eller flerdobbelt bufring, hvilket kan udnyttes til at effektivisere kørslen eller til at udjævne spidsbelastninger.

Eksempel 6.

D er en proces, der administrerer en ydre enhed. Indholdet af en besked, der sendes til D er (kommando,databuffer,svarsemafor), hvor kommandoen kan være "skriv" eller "læs". Når en kommando er udført bliver databufferen returneret, idet beskeden sendes til den angivne svarsemafor; i stedet for en kommando indeholder beskeden nu en status for den udførte dataoverførsel:

proces D:

```
D:  receive(Dsemafor,besked);
    kommando := besked(1);
    databuffer := besked(2);
    svarsemafor := besked(3);
    if kommando = "skriv" then skriv(databuffer)
                          else læs(databuffer);
    status := ... ;
    send(svarsemafor,(status,databuffer));
    goto D;
```

proces i:

```
Pi: =
    =
    send(Dsemafor,(skriv/læs,databuffer,semafor(i)));
    =
    =
    receive(semafor(i),besked);
    =
    =
```

Da beskedsemaforer specielt kan benyttes som generelle, kan eksempler på anvendelse af generelle semaforer umiddelbart oversættes til beskedsemaforer. Da beskedsemaforen næppe kan implementeres så beskeder ikke optager plads selvom de er tomme, kan det i visse tilfælde være en dårlig løsning, at lade semaforværdien for en generel semafor repræsentere som det samme antal tomme beskeder hægtet op i en kø. Dette gælder naturligvis især, hvis antallet risikerer at blive stort, hvilket er muligt i eksempel 3.

Man ser at beskedsemaforer er gode til at sikre udelelighed. Udover metoden " receive(MUTEX),.. den udelelige operation.., send(MUTEX, -)", som svarer til hvad man gør med generelle og binære semaforer, kan udeligheden klares ved at pågældende ressource sendes rundt mellem processerne i en besked. Dette kan enten ske direkte eller ved hjælp af en pulje, som ressourcen hentes fra og leveres tilbage til.

Trafiklys-semaforer

Semaforen kan være i to tilstande "rød" og "grøn", og har tilknyttet en proceskø, som altid er tom når tilstanden er "grøn". Følgende operationer kan foretages på en trafiklys-semafor:

start	Alle ventende processer startes og tilstand := "grøn"
stop	Tilstand := "rød"
passér	Hvis tilstanden er "rød" må processen vente til den bliver "grøn". Hvis tilstanden er "grøn" fortsætter processen uhindret.

Anvendelser:

Eksempel 6.

A og B er to processer der skiftes til at køre:

proces A:

```
A:  passér(S);
    =
    =
    stop(S);
    start(T);
    goto A;
```

proces B:

```
B:  stop(T);
    start(S);
    passér(T);
    =
    =
    goto B;
```

Ligegyldig hvordan de to trafiklys-semaforer S og T er initialiseret vil koden i A og B blive udført skiftevis startende med A.

Generelt er trafiklys-semaforer kun anvendelige i temmelig specielle tilfælde, som de to nedenstående eksempler. Til mange anvendelser kan man klare sig med sindrige sammensætninger af mange "trafiklys", men det vil ofte være svært at overbevise sig selv om at det går godt i alle tilfælde, og specielt at der ikke kan opstå baglås-situationer.

Eksempel 7.

Proces B starter en anden proces A. Efter et stykke tid vil B være sikker på at A er færdig:

```
A:  =                          B:  =
    start(T);                      her startes proces A
                                   =
                                   passér(T);
                                   =
```

Eksempel 8.

Vi tænker os at vi har et system af parallelle processer, som udfører en del forskellige opgaver. En af disse opgaver udføres af en proces, der kun kræver køretid med mellemrum, men så kræver den til gengæld en meget stor del af datamatens samlede kapacitet. Det kan ikke nytte at give denne proces lov til at få rådighed over al centralenhedens tid, da der kan være transporter i gang på ydre enheder, som ikke må gå tabt, men man kan ved hjælp af et "trafiklys" sørge for at kun igangværende transporter gøres færdige og ingen nye startes:

```
den periodiske proces:          de øvrige processer:

P:  stop(T);                    =
    =                           passér(T);
    start(T);                   =
    vent en periode;            passér(T);
    goto P;                     =
```

Hver gang en af de øvrige processer skal i gang med en ny tidskrævende opgave undersøger den om den må fortsætte ved hjælp af et kald af "passér". Man kan på denne måde sikre at P kan udnytte maskinen fuldt ud efter et vist tidsrum, ved at indsætte kald af passér med passende mellemrum. I dette eksempel må vi gå ud fra et dedikeret system, hvor alle processer er indrettet til at passe sammen.

Postkasse-semaforer

Semaforen består af en proceskø, hvorpå følgende operationer kan foretages:

tøm	Alle ventende processer aktiveres
vent	den kaldende proces må vente til "postkassen tømmes"

Anvendelser:
Postkassesemaforen bruges til at signalere fra én proces til en eller flere andre at en betingelse er opfyldt.

Eksempel 9. Simpel synkronisering af to processer:

```
proces A: =                    proces B: =
          vent(P);                       tøm(P);
          =                              =
```

A fortsætter først fra "vent" når B er nået til "tøm". Denne anvendelse forudsætter at man er sikker på at A kommer til"vent" før B kommer til "tøm".

Eksempel 10.
En datamat har tilknyttet en samling skrivemaskinetermenaler, som alle kan skrive et tegn hvert 0.1 sekund. Til hver terminal er der knyttet en proces der skriver på den. Hvis behandlingen af afbrydelser er uforholdsmæssig dyr, kan det belaste datamaten hårdt, hvis alle aktive terminaler hvert 0.1 sekund forårsager en afbrydelse, der starter en afbrydelsesrutine, som ikke gør andet end at aktivere terminalprocessen. Det samme kan da klares med en postkassesemafor og én afbrydelsesrutine, der aktiveres hvert 0.1 sekund:

```
afbrydelsesrutine: tøm(P);      terminalprocesser: skriv et tegn;
                                                   =
                                                   vent(P);
                                                   goto terminalproces;
```

Sammenligning af semaforer

Den følgende sammenligning vil tage sigte på at undersøge hvilke semafortyper, der er så anvendelige, at de også kan benyttes til det de øvrige typer er velegnede til. Sådanne semafortyper vil nemlig være velegnede til at medtage i et generelt styresystem. Derimod tages der f.eks. ikke hensyn til at binære semaforer er særlig nemme at implementere på visse datamater. I nogle tilfælde vil det være af betydning om der er tale om korutiner eller ej, og i disse tilfælde vil det blive bemærket.

Når jeg ønsker at vise at semafortype A kan benyttes til at implementere semafortype B, skriver jeg en skitsekode af operationerne på B ved hjælp af A's operationer. I nogle tilfælde er mine påstande åbentlyst rigtige; andre gange kan det være sværere at indse. Jeg kunne enten prøve at forsvare mine postulater med lange, uformelle forklaringer, som jeg frygtede ville være værdiløse, eller jeg kunne forsøge at bevise dem efter samme recept, som Brinch Hansen benytter i kapitel 3 i sin bog (1). Det sidste har jeg dog ment ville falde uden for rammen af denne rapport. Jeg har derfor valgt at lade skitsekoden tale for sig selv, eller i hvert fald højst knytte nogle få bemærkninger til den.

Jeg vil benytte overskriften " X -) Y ", når jeg vil vise at semafortype Y kan implementeres ved hjælp af X's operationer.

Generel -) binær

```
generel semafor MUTEX (1)
generel semafor G     (0)
heltalsvariabel antal (0)
```

(Her og i det følgende vil størrelsen i () efter en erklæring være initialiseringsværdien)

"lås"

```
$wait(MUTEX);
 antal := antal - 1;
 signal(MUTEX);
+$wait(G);
 $
```

"åbn"

```
$wait(MUTEX);
 if antal < 1 then begin
                       antal := antal + 1;
                       signal(G);
                   end;
 signal(MUTEX);
$
```

Den kode der står mellem to $-tegn udføres udeleligt. Uden for de udelelige områder gælder:
" G-antal = antallet af processer, som befinder sig i + ".
Det udnyttes at G er generel, idet der godt kan hænge flere "signal"-operationer og vente på processer, som er blevet afbrudt i punktet + , svarende til at G kan blive større end 1.

Hvis man arbejder med korutiner får man i stedet:

"lås"

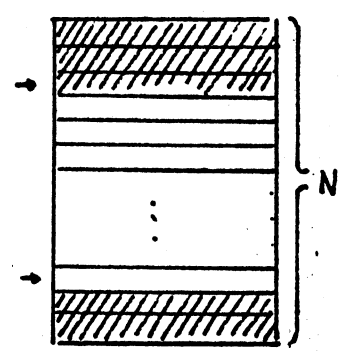
```
antal := antal - 1;
wait(G);
```

"åbn"

```
if antal < 1 then
begin
     antal := antal + 1;
     signal(G);
end;
```

Man ser at det er MUTEX, der bliver overflødig, og samtidig bliver det langt nemmere at se, at det er det rigtige der sker, når man er sikker på at kodestumperne udføres udeleligt.

Generel -) besked



buffer(0 : N-1)

generelle semaforer:
FRI (N) - antallet af fri bufferelementer
OPT (0) - antallet af bufferelementer med beskeder
MUTEX (1)

heltalsvariable:
bund (0)
top (0)

send(besked)

```
wait(FRI);                          - vent til der er en fri buffer
wait(MUTEX);
buffer(top) := besked;              - indsæt besked i den første fri buffer
top :=  if top = N then 0
        else top+1;                 - angiv at bufferen er brugt
signal(MUTEX);
signal(OPT);                        - fortæl at der er en besked klar
```

receive(besked)

```
wait(OPT);                          - vent til der er en besked klar
wait(MUTEX);
besked := buffer(bund);             - hent den ældste besked
bund := if bund = N then 0
        else bund+1;                - angiv at den er hentet
signal(MUTEX);
signal(FRI);                        - fortæl at der er en fri buffer parat
```

Man ser at en besked bliver begrænset til en fast størrelse, men det er ingen begrænsning, da det f.eks. kan være en pegepind til det sted hvor den "rigtige" besked står. Derimod er det en begrænsning at der er et fast antal bufferelementer, men man kan få samme virkning som med besked-semaforerne, hvis man giver antallet en passende størrelse, f.eks. antallet af linjeskrivere i eksempel 4 og antallet af buffere i eksempel 5.

Hvis der er tale om korutiner bliver MUTEX overflødig, da den kun bruges til at sikre udelelighed, og igen ser man at det bliver nemmere at indse, at virkningen i alle tilfælde er den samme som af de rigtige "send" og "receive".

Generel -) trafiklys

generelle semaforer:
MUTEX (1)
T (0)

variable:
farve (mulige værdier: rød,grøn) (grøn)
antal (heltal) (0)

stop:

```
wait(MUTEX);
farve := rød;
signal(MUTEX);
```

passér:

```
wait(MUTEX);
if farve = rød then
begin
     antal := antal + 1;
     signal(MUTEX);
    $wait(T);
end
else signal(MUTEX);
```

start:

```
wait(MUTEX);
for i := 1 step 1 until antal do
signal(T);
antal := 0;
farve := grøn;
signal(MUTEX);
```

Det eneste punkt, hvor man kunne forestille sig at der kunne komme problemer, var hvis "passér" blev afbrudt ved $ og "start" blev kaldt af en anden proces. Uden for "start" gælder der imidlertid altid:
" T = antallet af processer der hænger og venter i $ " .
Processer som venter i $ mens start bliver udført vil derfor bare fortsætte igennem wait(T) efterhånden som de igen får rådighed over centralenheden.

Også her gælder det at MUTEX kan undværes for korutiner, så operationerne bliver nemmere at implementere og det bliver nemmere at se at der sker det rigtige.

Generel -) postkasse

generelle semaforer: P (0) heltalsvariabel: antal (0)
MUTEX (1)

"vent"

```
wait(MUTEX);
antal := antal + 1;
signal(MUTEX);
wait(P);
```

"tøm"

```
wait(MUTEX);
for i := 1 step 1 until antal do
signal(P);
antal := 0;
signal(MUTEX);
```

Man ser at en postkasse er et trafiklys, som aldrig kommer i tilstanden "grøn", og "oversættelsen" til generelle semaforer er da også næsten den samme.
MUTEX kan undværes for korutiner.

Besked -) generel

Dette er helt trivielt, da en generel semafor bare er en beskedsemafor, hvor indholdet af beskeden er uden betydning.
Heraf følger umiddelbart:

besked -) binær , besked -) trafiklys , besked -) postkasse

Trafiklys -) postkasse

trafiklys T (rød)

"vent"

```
passér(T);
```

"tøm"

```
start(T);
stop(T);
```

Næsten oplagt - hvis "vent afbryder mellem start(T) og stop(T) svarer det til at postkassen ikke var tømt endnu.

Mellem kladdeskrivning og renskrift af denne rapport blev jeg gjort opmærksom på, at en artikel i BIT nr. 15, 1975 af Arne Maus (11) indeholdt et bevis for at binære semaforer kan benyttes til at implementere generelle semaforer. Jeg havde ellers med grundlag i forelæsningsnoter til Datalogi 2 anset dette problem for uløst. Som kuriosum vil jeg formulere her uden at det i øvrigt får indflydelse i det følgende:

binær -) generel

```
binær semafor B      (låst)
binær semafor MUTEX  (åben)
heltalsvariabel n    (0)
```

"wait"

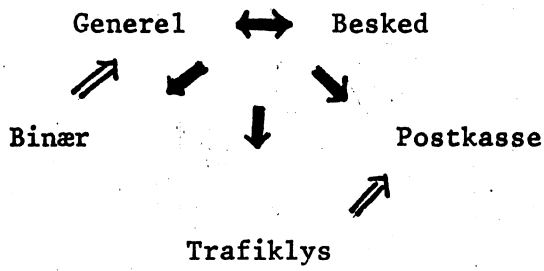
```
lås(B);
lås(MUTEX);
n := n - 1;
if n ≠ 0 then åbn(B);
åbn(MUTEX);
```

"signal"

```
lås(MUTEX);
if n = 0 then åbn(B);
n := n + 1;
åbn(MUTEX);
```

Konklusion

Vi har nu:



hvilket viser, at hvis man har én af semafortyperne "generel" og "besked", behøver man principielt ingen af de andre. I praksis kunne man imidlertid godt forestille sig at nogle af de øvrige typer var gode at have ved siden af. Imidlertid viste det sig, at de steder hvor man virkelig kunne bruge en binær semafor, kunne man nøjagtig lige så godt benytte en generel, og trafiklys- og postkassesemaforer var kun velegnede til meget specielle formål, mens de ikke var meget værd til mere sædvanlige udeleligheds- og synkroniseringsopgaver.

I et generelt system står man altså over for valget mellem beskedsemaforer og generelle semaforer og af dem er beskedsemaforen langt det stærkeste værktøj. Til gengæld er operationerne på en generel semafor simplere at implementere og derfor ofte hurtigere at udføre end de tilsvarende operationer for beskedsemaforer. Endvidere kan det koste en del overflødig lagerplads at benytte beskedsemaforer med tomme beskeder, når det i virkeligheden er en generel semafor man har brug for.

Konklusionen må blive, at det er rimeligt at medtage såvel beskedsemaforer som generelle semaforer i et generelt system.

II.2.D Tidsmåling

Dedikerede systemer til små datamater vil meget ofte have tidstro opgaver, og det vil tit være nødvendigt at synkronisere en eller flere processer med et ur under en eller anden form.

En metode er at lade systemet vedligeholde et systemur, som tælles op, når der med faste mellemrum kommer afbrydelser fra et taktsignal. Tidsmålingen kan nu indføres ved at en proces kan bede systemuret om at blive "vækket" efter et givet tidsrum. Udfra systemuret kan det absolutte tidspunkt for denne vækning beregnes, og procesbeskrivelsen kan indsættes i en tidskø efter stigende aktiveringstidspunkter. Hver gang en ny proces skal udvælges til at køre, skal systemet undersøge om tiden er inde til at aktivere nogen proces i tidskøen.
En anden metode fås uden at vedligeholde et systemur ved at operere med en tidskø, hvor de ventende processer står med en angivelse af det antal taktsignaler de ønsker at vente før de reaktiveres. Urafbrydelsesrutinen tæller hver gang den kører denne tæller ned med 1 for hver procesbeskrivelse i tidskøen. Når en tæller bliver 0 aktiveres den tilsvarende proces.

Hvis beskedsemaforer er implementeret i systemet kan de også benyttes til tidsmåling. I dette tilfælde sender en proces en besked til en særlig systemsemafor. I beskeden angives et tidsinterval og en svarsemafor. Efter det angivne tidsinterval bliver beskeden sendt tilbage til den svarsemafor der er opgivet. En proces kan afsende en sådan besked og straks sætte sig til at vente på svarsemaforen. I dette tilfælde opnås samme virkning som ved de to ovennævnte metoder:

```
=
send(UR,(tidsinterval,S));
receive(S);
=
```

Løsningen med beskedsemaforer giver imidlertid væsentlig bedre muligheder for tidsmåling. En forbedring er at en proces ikke behøver at sætte sig til at vente på svarsemaforen med det samme. Det kan udnyttes til at aktivere en proces med faste mellemrum:

```
P:  send(UR,(periode,S));
    den periodiske opgave
    receive(S);
    goto P;
```

En anden væsentlig fordel er at svarsemaforen kan modtage beskeder fra mere end ét sted. Det kan udnyttes til at få tidssignaler, hvis en operation ikke er færdig inden for et givet tidsrum; det kan man specielt have glæde af ved behandling af ydre enheder. Metoden virker således:

```
P:  send(Asem,kommando til A);
    send(UR,(tidsinterval,S);
    receive(S);
    undersøg om beskeden er et
    svar eller et tidssignal
    og vælg udfra det en pas-
    sende aktion
    =
```

```
A:  receive(Asem);
    udfør kommando;
    send(S,svar);
```

Tidsmåling ved hjælp af beskedsemaforer kan implementeres ved at tidsintervallet i de beskeder, der hænger på UR-semaforen,tælles ned for hver taktsignalafbrydelse, indtil det bliver 0 og beskeden sendes til svarsemaforen. En anden mulighed er at indføre et systemur og anbringe de beskeder der skal returneres i en tidskø med voksende tider for returneringen. Styresystemet må så på passende tidspunkter undersøge om det er tid til at sende nogen beskeder til de tilsvarende svarsemaforer.

II.2.E Drivprogrammer

Et drivprogram er betegnelsen for et program, der styrer en ydre enhed. I de systemer jeg i det følgende afsnit vil kigge på er opfattelsen af drivprogrammernes placering i sammenhæng med korutiner eller parallelle processer temmelig varieret.
Drivprogrammet består normalt af noget kode, der starter den ydre enhed - initieringsrutinen - og en afbrydelsesrutine, der aktiveres hver gang den pågældende ydre enhed afbryder. Det særlige ved afbrydelsesrutinen er, at den udføres øjeblikkelig når afbrydelsen kommer, uden at man giver sig tid til at skifte den aktuelle procesbeskrivelse ud. Når afbrydelsesrutinen er færdig fortsætter den afbrudte proces.
Der er forskel på om drivprogrammet opfattes som en proces hhv. korutine på linje med de øvrige i systemet, eller om den betragtes som noget, der ligger uden for systemet, og som derfor skal behandles specielt. I nogle af systemerne optræder forskellige mellemformer. Forskellene optræder i følgende spørgsmål:
Om drivprogrammet kan have en procesbeskrivelse.
Om drivprogrammet kan kommunikere normalt til de andre processer.
Om man kan kommunikere normalt til drivprogrammet fra andre processer.

De begrundelser der fremføres for at behandle drivprogrammer specielt er dels effektivitetshensyn og dels hensyn til at drivprogrammernes kode skal have mulighed for at være reentrant. Jeg mener at begge hensyn kan tilgodeses i rimelig grad inden for systemets rammer, som det også er gjort i mindst ét af de diskuterede systemer (NBB Monitor og til dels VACS).

Den resterende omtale af drivprogrammer er henlagt til omtalen af de enkelte systemer, og her specielt til beskrivelsen af mit eget.

III. BESKRIVELSE AF 4 EKSISTERENDE STYRESYSTEMER

III.1 DIXI

Systemtype kørende dedikeret system.

Anvendelse Datalogisk Instituts PDP 11/20 er forbundet dels med et antal datamatcentre og dels med et antal terminaler. DIXI gør det muligt at køre fra en vilkårlig terminal på et vilkårligt center.

Procestyper korutiner
drivprogrammer
"command"

Prioriteter nej

Semaforer beskedsemaforer

Omfang fuldt implementeret system.

Beskrivelse:

Opbygning:

Systemet består af følgende typer kode:

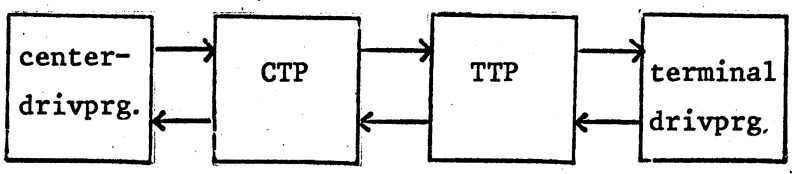
Korutiner der foretager det egentlige arbejde, som f.eks. at læse en linje fra en terminal, konvertere tegnene til et andet alfabet og sende linjen til en anden korutine. En korutine består af to dele:
1) En korutinebeskrivelse som identificerer korutinen.
2) Selve koden som er fælles for alle korutiner af samme type.

Driv-programmer der har mere specialiserede opgaver. Et drivprogram overvåger en ydre enhed. Det aktiveres af en korutine, som f.eks. kan bede det om at læse en linje. Når dets opgave er slut aktiverer det korutinen igen. Der findes to typer drivprogrammer - centerdrivprogrammer og terminaldrivprogrammer. Drivprogrammer kan betjene flere forbindelser på en gang.

Central-logik der sørger for kommunikationen, dels mellem korutiner og dels mellem drivprogrammer og korutiner.

Command som behandler alle kommandoer til systemet og sørger for at forbindelser bliver oprettet og nedlagt.

Forbindelser:



En forbindelse består af to drivprogrammer og to korutiner:

1. Et terminaldrivprogram der tager sig af transporter til og fra brugerterminalen.

2. En terminal-tjener-proces, TTP, der tager sig af de terminalbestemte, ikke tidsafhængige opgaver.

3. En center-tjener-proces, CTP, der tager sig af de ~~terminal~~bestemte, ikke tidsafhængige opgaver.

4. Et centerdrivprogram, der tager sig af transporter til og fra centret.

Når en forbindelse er etableret kan der begynde at glide information gennem systemet. Hver af korutinerne kan midlertidigt blive ejer af en databuffer; denne kan - eller rettere en reference til den kan - sendes som besked til det tilsvarende drivprogram sammen med f.eks. en læsekommando. Når drivprogrammet har læst en linje og har anbragt den i bufferen returneres denne til korutinen. Her bliver indholdet behandlet, hvorpå bufferen sendes videre til makkerkorutinen, som atter behandler indholdet, før den sender bufferen til sit drivprogram som uddata. Til slut returneres den tomme buffer fra drivprogram til korutine og videre til sin ejer. Hele kommunikationen foregår ved hjælp af beskedsemaforer, hvor beskeden typisk består af en kode, en databufferadresse, en status og en svarsemaforadresse.

En forbindelse oprettes ved at "command" henter to korutinebeskrivelser fra en pulje og knytter dem til hhv.koden for en terminal-tjener-proces og koden for en center-tjener-proces, samtidig med at der indsættes oplysninger i korutinebeskrivelsen om hvilken terminal og hvilket center, der er tale om. Beskedbuffere og databuffere ligger også i hver sin pulje og må hentes og frigives af en korutine hver gang de skal bruges. Dette sker for at kunne betjene flere forbindelser end der er tilsvarende buffere, og det sker på retfærdig vis ved at den korutine, der har

ventet længst på en buffer, får den første der bliver frigivet. Det samme princip gør sig gældende ved tildeling af køretid til de enkelte korutiner; der er ingen der er prioriteret højere end andre, de kommer til i den rækkefølge de har bedt om centralenheden.
Alle de ovennævnte puljer af korutinebeskrivelser, af beskedbuffere og af databuffere administreres af en beskedsemafor, således at et puljeelement fås ved en "receive"-operation og frigives igen ved en "send"-operation.

Drivprogrammer

Der findes to slags drivprogrammer i DIXI, centerdrivprogrammer og terminaldrivprogrammer; jeg vil her nøjes med at omtale sidstnævnte.
Et drivprogram aktiveres - omend på speciel vis - ved at en TTP sende en besked til det, og TTP'en aktiveres igen ved at drivprogrammet sender en besked tilbage til den - denne gang traditionelt via en beskedsemafor.
Et terminaldrivprogram kan behandle én inddata eller uddata transaktion for hver terminal ad gangen. Til hjælp har drivprogrammet en terminaltabel, hvor der for hver terminal findes følgende oplysninger:

- Terminalens tilstand (ledig, aktiv eller fjernet)
- Pegepind til den besked der beskriver den aktuelle operation
- Adresse på det næste tegn der skal sendes/lagres
- Slutadresse
- Tidsinterval
- Terminaltype
- Terminalidentifikation
- TTP-korutinebeskrivelse, hvis terminalen er aktiv
- Centernummer, hvis terminalen er aktiv

Når en TTP ønsker at skrive på/læse fra en terminal foretager den et subrutinehop til drivprogrammet. Her undersøges om den pågældende terminal er ledig. I så fald indsættes de rigtige værdier i terminaltabellen og den første transport startes. I modsat fald sættes beskeden i kø til den pågældende terminal. Drivprogrammet aktiveres for hvert tegn ved afbrydelser. Når en operation på en terminal er afsluttet sendes en besked tilbage til den angivne svarsemafor og derpå undersøger drivprogrammet om der står nogen besked i kø til den pågældende terminal. Hvis der gør går drivprogrammet i gang med at udføre de operationer de beskriver.

Tidsmåling

Når et drivprogram f.eks. får besked om at læse en linje fra en terminal, sender det besked til systemuret om at give et tidssignal efter et passende stykke tid. Tidssignaler administreres ved at systemuret har en tabel, hvori der for hver terminal kan angives en absolut tid og en adresse. Ved altid at holde øje med den laveste absolutte tid for nogen terminal kan urrutinen hoppe til den angivne adresse, hvis et tidssignal ikke når at blive afbestilt inden tidspunktet nås. På denne måde sikrer man sig at besked- og databuffere altid bliver returneret inden for rimelig tid.

Uret benyttes også til statistik. F.eks. tælles antallet af bestilte og antallet af udførte tidssignaler, antal elementer i hver kø, antal gange hver kommando udføres m.m. Ved hjælp af uret får statistikrutinen et tidssignal med jævne mellemrum og kan da udskrive disse oplysninger.

III.2 RC3500 Monitor I

Systemtype Generelt styresystem til dedikerede systemer på en RC3500.

Anvendelser Ingen bestemte anvendelser nævnt i beskrivelsen.

Procestyper Parallelle processer, som specielt er korutiner hvis de kører på samme prioritetsniveau.
Drivprogrammer

Prioriteter Ja

Semaforer Beskedsemaforer og generelle semaforer.

Omfang Systemfunktioner, initialiseringsprogram, testudskrifter, eksempler på en korutine og et drivprogram.

Beskrivelse:

Et færdigt system består af en række processer, som kører parallelt. Processerne er delt i to kategorier, drivprogrammer som kører på afbrydelsesniveau > 0, og de øvrige, som kører på afbrydelsesniveau 0. Styresystemet kører på afbrydelsesniveau 0.

Synkronisering mellem almindelige processer sker ved hjælp af beskedsemaforer eller generelle semaforer. Kommunikation fra et drivprogram til en anden proces kan ikke foregå på denne måde, fordi drivprogrammet kører på et højere afbrydelsesniveau end styresystemet, og derfor kunne risikere at ødelægge den udelelige tilgang til semaforkøerne m.m., hvis de lavede almindelige systemkald. Det kunne undgås ved at styresystemet kørte med afbrydelsessystemet sat ud af kraft, men det har man ikke ønsket. Når et drivprogram ønsker at foretage en "signal"- eller "send"-operation, kalder den derfor en rutine "the interrupter", der kører med det maximale afbrydelsesniveau, og denne sætter drivprogrammets systemkald ind i en speciel kø "monitor queue". Ved at lade "the interrupter" foretage dette opnår drivprogrammet at andre, mere priviligerede drivprogrammer ikke kan afbryde imens, og ved blot at sætte systemkaldet i kø, sørger man for at styresystemet kun behøver at lukke for afbrydelser, mens systemkald hentes i "monitor queue".

Hver af processerne på afbrydelsesniveau 0 har tilordnet en prioritet. Ved hvert systemkald kan der ske en ændring i sættet af aktive processer, idet et sådant kald dels i sig selv kan være skyld i, at der bliver en aktiv proces mere eller mindre, og dels kan give plads for systemkald fra drivprogrammer, som kan aktivere processer. Når dette er sket vælges en ny kørende proces blandt de aktive, som den der er højest prioriteret og har været aktiv længst. På denne måde fås at processerne er korutiner inden for samme prioritetsniveau.

Drivprogrammer

Kommunikation til et drivprogram foregår på følgende måde:
En beskedsemafor kan være i 4 tilstande i stedet for de sædvanlige 3:

"open"	mindst én ventende besked
"passive"	hverken ventende beskeder eller ventende processer
"closed"	mindst én ventende proces
"stopped"	den nye tilstand, som vil blive forklaret i det følgende

Et drivprogram kan aflæse tilstanden på den semafor, som administrerer beskeder med kommandoer til drivprogrammet. Hvis den er "open" henter drivprogrammet selv en besked - dette sker udeleligt, da drivprogrammet kører på et højere afbrydelsesniveau end styresystemet. Hvis den er "påssive" betyder det at drivprogrammet ikke har noget at lave. I dette tilfælde ændrer drivprogrammet semaforens tilstand til "stopped". Når en proces sender en besked til en semafor der er "stopped", sker der et hop til en adresse, der er knyttet til semaforen. Dette vil være et hop til drivprogrammet, som henter beskeden og initialiserer en transport, før den returnerer til styresystemet.

Tidsmåling

I Monitor I operarer man med en urproces. En proces kan sende en besked til urprocessen indeholdende et tidsrum og en svarsemafor. Derefter vil beskeden blive returneret til svarsemaforen efter det givne tidsrum.
Når urprocessen modtager beskeden beregner den det tidspunkt, hvor beskeden skal tilbagesendes, og hægter beskeden op i en kø, som er ordnet efter stigende tider. Når det interne ur får samme værdi som den første besked i køen bliver denne sendt til den svarsemafor, som den indeholder adressen på.

III.3 The NBB Semaphore Monitor

Systemtype Generelt styresystem for dedikerede systemer til en PDP 11.

Anvendelse Procesovervågning f.eks. på kraftværker.

Procestyper Parallelle processer, som specielt kan være korutiner, hvis de kører på samme prioritetsniveau, og som specielt kan være drivprogrammer, hvis de administrerer en ydre enhed.

Prioriteter Ja

Semaforer Beskedsemaforer og generelle semaforer

Omfang Systemfunktioner, initialiseringsprogram, testudskrifter, strømsvigtrutiner.

Beskrivelse:

Et kørende system består af parallelle processer, hvoraf nogle - drivprogrammerne udelukkende beskæftiger sig med styring af ydre enheder. Al kommunikation mellem processer sker ved hjælp af semaforoperationer på generelle semaforer eller beskedsemaforer. Sættet af aktive processer kan skifte hver gang en eller anden proces laver en semaforoperation, og efter en sådan udvælges en ny kørende proces, som den højest prioriterede, som har været aktiv længst. I modsætning til RC3500 Monitor I, hvor drivprogrammernes systemkald blev sat i kø, og skift til ny kørende proces derfor kun kunne ske ved de almindelige processers systemkald, kan en proces her når som helst blive afbrudt af en højere prioriteret. Principielt er der dog ikke den store forskel på de to systemer; en proces bliver ved med at være aktiv til den foretager et systemkald, der kan resultere i at den kommer til at vente, og i mellemtiden kan kun processer med højere prioritet komme til at køre. I NBB Semaphore Monitor gælder det dog ikke helt at processerne er korutiner indenfor samme prioritet, idet ét prioritetsniveau bliver behandlet specielt. Processer med prioritet 1 er ikke korutiner, men bliver 10 gange i sekundet rokeret, så den, der stod først i køen til at blive aktiv, nu bliver sidst. Denne simple teknik til cyklisk aktivering (round-robin) kan være nyttig, når der er tale om processer, hvis køretidsforbrug er svært at overskue.

Drivprogrammer

Et drivprogram er som nævnt en proces som alle andre med en procesbeskrivelse og en prioritet. Når drivprogrammet er frit venter det på en semafor, til der er sendt en besked med en kommando til det, og derpå bliver det med tiden kørende i henhold til sin prioritet. På dette tidspunkt kan det så starte en transport, og når det er gjort kalder det en system-

funktion "wait interrupt", der bevirker at processens prioritet ændres til at være højest mulig - det såkaldte "ligthning level". Sammen med kaldet gives en adresse i hvilken processen fremover skal aktiveres ved hver afbrydelse fra den pågældende ydre enhed. Når operationen på denne måde er afsluttet kalder processen "end lightning", hvorpå den atter får sin sædvanlige prioritet og kan begynde at sende og modtage beskeder. Ved "wait interrupt" angives desuden en maximal tid drivprogrammet ønsker at vente på afbrydelser fra den ydre enhed; hvis afbrydelsen ikke er kommet inden for dette tidsrum bliver drivprogrammet aktiveret af uret. Visse drivprogrammer som f.eks. urdrivprogrammet kører altid på "lightning level".
Da drivprogrammet er en sædvanlig proces kan den specielt bestå af en procesbeskrivelse for hver af flere ens ydre enheder og et stykke reentrant kode.

Tidsmåling

Urdrivprogrammet har 4 funktioner:

1) Opdatering af tidspunkt og dato.
2) Implementering af tidssignaler til drivprogrammer.
   Når et drivprogram kalder "wait interrupt" opgiver det et antal 2-sekunders perioder, som gemmes i procesbeskrivelsen. Hvert andet sekund tæller urdrivprogrammet dette felt ned med 1 i de relevante procesbeskrivelser. Hvis en tæller når 0 inden tidssignalet annulleres, bliver processen aktiveret med et passende fejlmærke sat. I stedet for perioder på 2 sekunder kan drivprogrammerne vælge perioder på 0.1 sekund. Disse administreres parallelt på samme måde.
3) Beskeder kan returneres efter et opgivet tidsrum.
   Processerne sender en besked indeholdende svarsemafor og tidssignaltæller til en af semaforerne "del0.1"(tidssignal hvert 0.1 sekund) eller "del2" (tidssignal hvert 2. sekund). Tidssignal-tællerne tælles ned med de valgte perioder, og når de bliver 0 bliver beskeden sendt til svarsemaforen.
4) Urdrivprogrammet administrerer den tidligere nævnte cykliske aktivering af aktive processer med prioritet 1.

III.4 VACS

Systemtype Styresystem til en Varian minidatamat, med specielt henblik på en bestemt konfiguration.

Anvendelse Systemet er lavet til at forbedre styringen af Bispebjerg Hospitals gammakamera ved hjælp af Varianmaskinen. Imidlertid er systemet lavet så generelt at det også kan bruges på andre Varian-anlæg.

Procestyper korutiner
afbrydelsesrutiner

Prioriteter Ja

Semaforer Trafiklyssemaforer og postkassesemaforer

Omfang Systemfunktioner og læse/skrive-rutiner til en del af de ydre enheder samt et omfattende overvågnings- og styringsprogram.

Beskrivelse:

Systemet arbejder udelukkende med korutiner og kommer på den måde udover de fleste udelelighedsproblemer, samtidig med at et reservationssystem forhindrer korutiner og afbrydelsesrutiner i at have behov for fælles data. Synkronisering af afbrydelsesrutiner og korutiner sker normalt ved hjælp af postkassesemaforer, hvor en korutine sætter sig til at vente på et "flag", som "hejses" af afbrydelsesrutinen, når den har fuldført sin opgave. I den foreliggende anvendelse er der ikke brug for korutiner med fælles kode, men det ser ud til at være muligt.

Drivprogrammer

Et drivprogram består af en korutine og en afbrydelsesrutine, som tilsammen styrer en ydre enhed. De øvrige processer reserverer en ydre enhed ved hjælp af systemfunktionen "RESERVE" og frigiver den igen efter brug ved hjælp af "RELEASE". Hvis enheden var optaget ved "RESERVE"-kaldet bliver den kaldende korutine sat i kø til den kan komme til. "RESERVE"-funktionen er indrettet så baglås i forbindelse med samtidig reservation af flere ydre enheder undgås. Når en korutine er blevet "ejer" af en ydre enhed er den også "ejer" af de systemsubrutiner der hører til, og ved hjælp af disse kan den udskrive et tal på decimal form, læse en linje, læse eller skrive en blok via en bloktransportenhed, skrive et enkelt tegn osv. En korutine har også mulighed for at kalde en systemsub-

rutine, der sørger for at en tekst givet i en databuffer bliver udskrevet på en skrivemaskineterminal uden at korutinen behøver at vente på det. Korutinen behøver ikke at være "ejer" af skrivemaskineterminalen for at kalde subrutinen. Subrutinen indsætter nemlig databufferen i en tekstkø, som udskrives af en systemkorutine, som kører parallelt med brugerkorutinerne. Systemkorutinen reserverer skrivemaskineterminalen på linje med de øvrige og gør sig selv passiv, når den ikke har mere at lave. Systemsubrutinerne arbejder ved at starte en transport, der når den er slut forårsager at der startes en afbrydelsesrutine. Afbrydelsesrutinen laver så vidt muligt ikke andet end at hejse et "flag", som aktiverer den tilhørende korutine, der - når den bliver kørende - foretager selve afbrydelsesbehandlingen. En undtagelse er en afbrydelsesrutine, der startes af urafbrydelser med passende mellemrum for at aflæse nogle tællere, som står i forbindelse med gammakameraet.

Tidsmåling

Systemuret opdaterer en absolut tid.
En korutine kan kalde en systemfunktion "HOLD" med en parameter, der angiver et tidsrum. Korutinebeskrivelsen bliver derved hægtet op på en kø "WATCH" efter voksende aktiveringstidspunkt. Før der skal vælges en ny kørende proces undersøger styresystemet om der er korutiner på "WATCH", der skal aktiveres, så de kan komme med i betragtning.

IV. BESKRIVELSE AF MIK

IV.1.A Grundlæggende ideer

Følgende er en gennemgang af de generelle krav jeg har fundet det rimeligt at stille til systemet:

1. Det må ikke forudsætte noget som helst om ydre enheder.

For et tilsvarende system til en minidatamat vil man normalt kunne regne med at der findes en skrivemaskineterminal eller lignende, og for større installationer ville man endvidere kunne forudsætte baggrundslager. Intet af dette er tilfældet her; man kan udmærket have en mikrodatamat, som overhovedet ikke har mulighed for at skrive noget nogen steder. Den kan f.eks. benyttes til at overvåge en maskine, og i så fald behøver den ikke at have andre ydre enheder end denne maskine. Det betyder bl.a. at styresystemet godt kan konstatere fejl i de kørende programmer, men ikke kan regne med at der er muligheder for fejludskrifter; tilsvarende er der ikke muligheder for at lave testudskrifter, men det er der heller ikke behov for, hvis et system skal indkøres på en simulator.

2. Det må ikke fylde for meget.

De væsentlige egenskaber ved en mikrodatamat i forhold til andre datamater er prisen og størrelsen. Anvendelsen af et styreststem må ikke betyde en væsentlig forøgelse af lagerkravet med det til følge at prisen vokser og datamaten fylder mere.

3. Det skal være simpelt.

Da ideen med styresystemet er at give mulighed for simplere opbygning af dedikerede systemer, og da det er urealistisk at forestille sig, at et system i denne størrelsesorden kan benyttes uden at brugeren er helt fortrolig med hvad der sker, må det ikke forsøge at implementere alt for indviklede metoder. Desuden må systemet nødvendigvis være simpelt, hvis koden ikke skal fylde for meget.

4. Det skal være generelt.

Det ville være urimeligt at udvikle et system af denne slags, som ikke er beregnet på nogen bestemt anvendelse, hvis det ikke kan benyttes til en bred klasse af opgaver.

5. Det skal være modulært.

Det er rimeligt - med begrundelserne 2,3 og 4 - at lave et system, der er så skrabet som overhovedet muligt, og lade alle de funktioner, som ikke er absolut nødvendige, være moduler, som kan medtages eller udelades efter behov.

Udover disse 5 punkter kan man selvfølgelig nævne mere generelle systemudviklingskriterier: Systemet skal være effektivt, beskrivelsen skal være så grundig, at det er muligt at overskue, om det kan benyttes til en given opgave, og endelig skal der til beskrivelsen være knyttet eksempler, så det er til at se, hvordan et system opbygget omkring styresystemet kan tænkes at fungere.

MIK er kodet, så det kan benyttes til enhver mikrodatamat, som er opbygget omkring en intel8080 centralenhed.
Mit valg stod næsten udelukkende mellem centralenhederne intel8008 og intel 8080, da kun disse to syntes at have en rimelig udbredelse og programmelstøtte. Intel8008 stod umiddelbart stærkest med en datamat i produktion opbygget omkring den ved navn MCS-8, to udgaver af simulator og assembler, et tilgængeligt eksemplar af MCS-8 tilsluttet H.C.Ørstedsinstituttets RC4000-anlæg, og endda et højere programmeringssprog PL/M. Dertil kom at jeg havde erfaring med intel8008 fra et tidligere projekt. Intel8080 var mindre tilgængelig, idet det eneste jeg havde adgang til var en assembler - GENASS på RC4000 (17). Til gengæld var intel8080 sin "lillesøster" intel8008 overlegen på alle andre områder idet den bl.a. er hurtigere og har et væsentlig mere avanceret ordrerepertoire. Hertil kom at der var to punkter hvor det var betænkeligt at anvende intel8008. Det ene drejer sig om åbning og lukning for afbrydelsessystemet. Muligheden findes indbygget i intel8080's centralenhed, mens det for intel-8008's vedkommende kun kan klares, hvis datamaten opbygges såspecielle

skriveordrer kommer til at give en tilsvarende virkning. Det andet problem drejer sig om at gemme centralenhedens status, når der kommer en afbrydelse. Det er så svært i en intel8008, at det grænser til det umulige, og det er en klar forudsætning for at få gavn af styresystemet.

Resten af dette afsnit er en beskrivelse af MIK. Beskrivelsen består for en stor del af en tilkendegivelse af hvilke af de metoder, der er diskuteret i de foregående afsnit, som er valgt. Begrundelsen for valgende bygger naturligvis på hvilken datamat, der skal bruges, men langt de fleste argumenter kan overføres til andre mikrodatamater og også til visse anvendelser af minidatamater.

IV.1.B Systemovervejelser

IV.1.B.1 Procestyper

I MIK er alle processer indbyrdes korutiner. Det betyder at en proces altid får lov til at køre indtil den specificerer et ventepunkt. Ventepunktet kan være en angivelse af at processen ønsker at vente på en eller anden nærmere specificeret hændelse, eller det kan blot være et ventepunkt, der er lavet for at processen ikke skal køre så længe i ét stræk, at det kommer til at ødelægge noget for andre i systemet. Metoden er valgt fordi den er generel nok til at kunne ligge til grund for et effektivt system, men samtidig fordi den giver et godt grundlag for en problemløsning, der giver programmer, som er nemme at overskue og dermed at indkøre og vedligeholde, idet delopgaver kan lægges ud i hver sin korutine. Endvidere opnår man at udelelig adgang til variable bliver noget man normalt slet ikke behøver at tænke på. Endelig er styresystemet nemt at implementere, når alle processer er korutiner, hvilket betyder det ikke fylder så meget, samtidig med at den tid det kørende system bruger til administration bliver beskeden.
Korutinerne implementeres ved hjælp af korutinebeskrivelser, som identificerer den enkelte korutine, dels ved at indeholde data om korutinen - specielt en beskrivelse af hvilken kode korutinen forventes at udføre - og dels ved at definere korutinens tilstand. De mulige tilstande er:

KØRENDE — Det vil sige at korutinen for tiden har rådighed over centralenheden. Der vil altid være netop én KØRENDE korutine, undtagen mens styresystemet udfører et ventepunktskald.

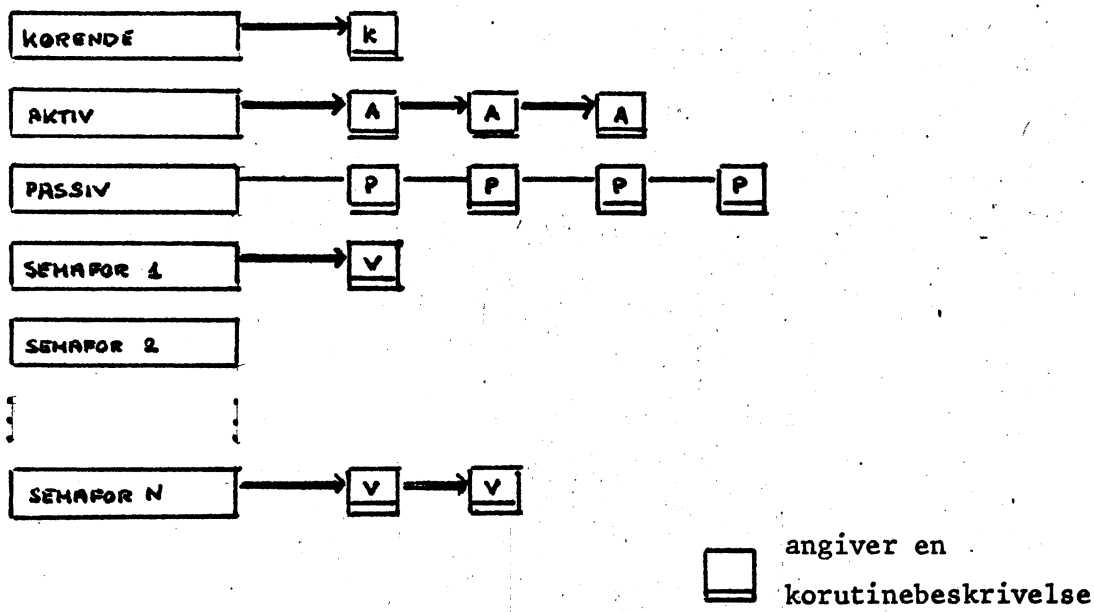
AKTIV — Korutinen venter på at blive KØRENDE.

PASSIV — Korutinen venter for tiden ikke på nogen konkret betingelse, men kan blive aktiveret af andre.

VENTENDE — Korutinen venter på en semafor.

I praksis er en korutines tilstand enten defineret ved at dens korutinebeskrivelse befinder sig i en semaforkø, i AKTIV-køen eller i KØRENDE-køen, eller ved at dens korutinebeskrivelse er mærket som passiv. At tilstanden PASSIV ikke beskrives ved hjælp af en kø, men ved et mærke i korutinebeskrivelsen er bare en kodeteknisk detalje.

Figuren giver en oversigt over hvilke tilstande korutinen kan være i:



IV.1.B.2 Prioriteter

Hver korutine har en prioritet, som hører til korutinebeskrivelsen. Fordelene ved prioritering af korutiner er allerede omtalt, så selvom man kan lave systemer hvor prioritering ikke er nødvendig (DIXI), er muligheden medtaget i MIK. Til gengæld er systemet indrettet så prioritering kan udelades.

Både procestype og afviklingsform er den samme som i VACS. Metoden med et sæt korutiner for hver prioritet som i RC3500 Monitor 1 og NBB Semaphore Monitor virker meget tiltalende, og jeg ville nok have overvejet at implementere den, hvis jeg havde kendt den tidligere i planlægningsfasen. En væsentlig fordel ved den valgte metode er at administrationstiden er mindre.

IV.1.B.3 Semaforer

I afsnit II viste det sig at de semafortyper, der med rimelighed kunne komme på tale var generelle semaforer og beskedsemaforer. Begge typer er implementeret, således at man kan vælge at medtage begge dele eller kun den ene slags efter behov. Da det viste sig at de to semafortyper i alle tilfælde kunne bringes til at erstatte hinanden, vil jeg i det følgende forudsætte at begge typer er med i systemet.

Om beskedsemaforer:

Overførslen af en besked fra en korutine til en anden ved hjælp af en beskedsemafor kan foregå på to måder. Den ene mulighed er at sender-korutinen opgiver selve beskeden ved "send"-kaldet, og denne derpå med tiden bliver udleveret til modtagekorutinen. I dette tilfælde må styresystemet internt råde over en pulje af beskedbuffere, som beskederne kan anbringes i når de skal hægtes op i kø. Imidlertid opstår der problemer når den pulje er tom. Det kan klares ved simpelthen at definere at der skal være beskedbuffere nok. Man kan også forsøge at komme over en spidsbelastningssituation ved at lade den korutine, der afsender beskeden, gå i kø til der kommer en fri buffer til beskeden. Det betyder imidlertid, at "send" operationen er nødt til at være et ventepunkt, da man ellers risikerer at systemet går i baglås, og det betyder igen, at "send"-operationen ikke må forekomme i afbrydelsesrutiner, hvor ideen er at man næsten øjeblikkeligt skal vende tilbage til den afbrudte proces. Alle disse komplikationer betyder at jeg vælger den anden mulighed, som går ud på at lade korutinerne udveksle beskedbuffere i stedet for beskeder. Korutinerne kan uden problemer administrere en beskedbufferpulje, for de kan jo benytte beskedsemaforer. I mange tilfælde - nemlig når de beskeder der udveksles alligevel består af en adresse - bliver administrationen i korutinerne ikke forøget.

Utraditionel brug af semaforer:

Det kan i nogle situationer være en fordel at vide, om der er en besked parat, inden en korutine udfører en "receive"-operation; det kan f.eks. dreje sig om en højt prioriteret korutine, der ikke vil vente, hvis der ikke er nogen, men snarere foretage en anden aktion. Her betyder proces-

sernes korutineegenskaber imidlertid, at semaforers værdi frit kan aflæses og hvis det viser sig at der er en besked, vil den stadig være der når "receive" kaldes. Derimod kan det når som helst ske at en afbrydelsesrutine kalder "signal" eller "send", så der kan godt risikere at være flere beskeder.
I MIK indordnes korutiner i en semaforkø i den rækkefølge de har kaldt "receive" eller "wait". Det ville måske i nogle tilfælde være mere rimeligt at indordne dem efter prioritet. På samme måde kunne man forestille sig eksempler, hvor beskeder burde sættes i kø i en speciel rækkefølge. Alle funktioner af den slags er nemme at indpasse i systemet, men er helt overladt til brugeren.

IV.1.B.4 Ventepunkter - subrutinekald

Når det er fastlagt hvilke funktioner styresystemet skal kunne udføre står det stadig tilbage at afgøre, hvilke kald der skal være ventepunkter og hvilke der ikke skal resultere i udskiftning af KØRENDE korutine. Den ene yderlighed er at lade alle kald være ventepunkter, men det giver nemt unødvendig megen administration, og som omtalt i forbindelse med beskedsemaforer volder det problemer, idet afbrydelsesrutiner ikke må kunne udføre ventepunkter, men bør kunne kalde "send" og "signal". Jeg har valgt den modsatte strategi. Der er kald, som nødvendigvis skal være ventepunkter; det gælder f.eks. "wait" og "receive", som netop skal give plads for andre korutiner, hvis de ikke kan fortsætte. Man kunne definere at de kun gav anledning til et ventepunkt, hvis de faktisk skulle vente, men jeg har besluttet at det altid skal være et ventepunkt, da det set fra korutinen alligevel ser sådan ud. De kald der ikke nødvendigvis er ventepunkter er lavet som kald af systemsubrutiner. Der er mulighed for at opdele lange beregninger ved hjælp af eksplicitte ventepunktskald. Alle de steder,hvor systemet piller ved de forskellige køer, er der lukket for afbrydelser udefra, for at sikre,at der ikke skal blive aktiveret en afbrydelsesrutine, som kan foretage et systemkald. Det vil i praksis sige at der altid er lukket for afbrydelser, når styresystemet kører.

IV.1.B.5 Tidsmåling

I MIK har jeg valgt at udføre tidsmåling ved hjælp af beskedsemaforer, da det viste sig at være væsentlig mere anvendeligt end blot at kunne forsinke en korutine med et fast tidsrum.

Den ene mulighed for implementation var ved at indføre et systemur, som ajourføres af en afbrydelsesrutine. Med et taktsignal på 50 Hz og en urvariabel der forøges med 1 for hver afbrydelse fås følgende overløbstider: (1 ord = 8 bit)

1 ord : 5 sekunder
2 ord : 20 minutter
3 ord : 85 timer = 3 1/2 døgn
4 ord : 900 døgn = 2 1/2 år

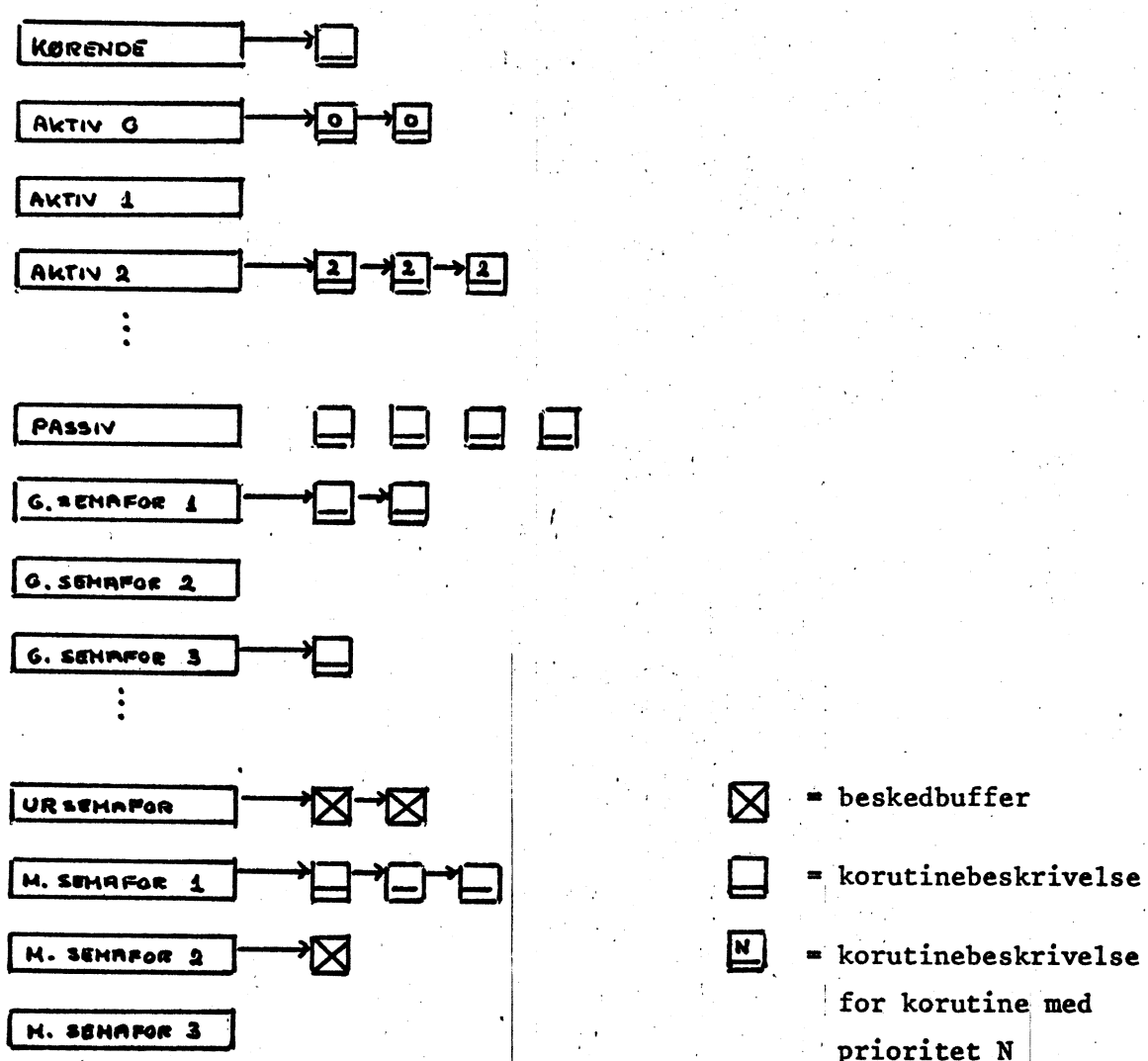
Da det er svært at udtale sig om anvendelser, og dermed om fornuftig taktsignalfrekvens og overløbstid, bliver ræsonnementet lidt løst, men jeg ville ikke anse det for rimeligt med mindre end 4 ord til systemuret.
For hver besked kunne man beregne tidspunktet for tilbagesendelsen og indsætte beskederne i kø efter stigende tider. Undersøgelsen af om der var nogen beskeder klar til returnering, kunne enten udføres før hver udvælgelse af en ny KØRENDE korutine eller for hver urafbrydelse. Med 50 Hz bliver der udført ca. 10.000 lagercykler eller gennemsnitlig ca. 5000 ordrer mellem hver urafbrydelse, så urafbrydelsesrutinen må forventes at køre sjældnest, så arbejdet bør ligge der.

En anden udvej var at anbringe beskederne i en kæde i den rækkefølge de ankom med en angivelse af, hvor mange taktsignaler de ville vente. Urafbrydelsesprocessen skal så løbe den kæde igennem og reducere alle tallene med 1. Når en tæller bliver 0 returneres beskeden. Denne metode har flere fordele. Indplaceringen i kæden kan ske med en almindelig "send"-operation, hvilket er hurtigt og kodebesparende, og tælleren kan være i et dobbeltord svarende til en forsinkelse på maximalt 20 minutter i tilfældet med 50 Hz, hvilket jeg anser for rimeligt. Forskellen mellem at arbejde med operationer på 2 ord og på 2 ord er forbløffende stor og betyder, at det er dennemetode der er implementeret, da det giver både plads- og tidsmæssige fordele. Som ulempe kan nævnes, at udførselstiden for urprocessen vokser med antallet af ventende beskeder.

IV.1.B.6 Drivprogrammer

I MIK er det tanken at et drivprogram skal bestå af en korutine og en afbrydelsesrutine. Korutinen kan kommunikere med de øvrige korutiner på normal vis f.eks. via beskedsemaforer. Når korutinen har modtaget en kommando kan den starte en operation på den ydre enhed. Afbrydelsesrutinen aktiveres når der kommer afbrydelser og kan signalere til korutinen om afsluttede transporter eller mere uventede afbrydelser ved hjælp af "signal" eller "send" operationer. Afbrydelsesrutinerne kører ikke udeleligt i forhold til korutinerne, og de variable de har adgang til skal derfor være private for drivkorutinen. For afbrydelsesrutinerne gælder at de ikke har adgang til variable via beskrivelsen af den KØRENDE korutine, som kan være en vilkårlig korutine i systemet når afbrydelsen kommer, men på anden vis må finde frem til deres egen korutinebeskrivelse.

IV.1.B.7 Systemoversigt



IV.2 Brugervejledning

IV.2.A Korutiner

En korutine i MIK består af en korutinebeskrivelse, som identificerer korutinen, og et stykke kode, som kan være privat, fælles for flere korutiner eller eventuelt kan indeholde kald af subrutiner, som er fælles for flere korutiner. Fælles kode eller subrutiner, som indeholder ventepunkter, skal være reentrant. Korutinebeskrivelsen indeholder følgende oplysninger:

Hægte (2 ord)	Dette felt bruges af systemet til adressen på den næste korutine i køen, når korutinen er AKTIV eller VENTENDE.
Ventepunktsadresse (2 ord)	En adresse der kan opfattes som korutinens private ordretæller. Den indeholder adressen på den næste ordre korutinen skal udføre når den bliver KØRENDE for korutiner der er AKTIVE, VENTENDE eller PASSIVE.
Status (1 bit)	Dette felt er 1 når korutinen er PASSIV og 0 når den er KØRENDE, AKTIV eller VENTENDE.
Prioritet (7 bit)	Her står korutinens prioritet. Den højeste prioritet er 0 og den lavest mulige er 127. Imidlertid frarådes det af plads- og tidsmæssige hensyn, at tage flere prioriteter med end nødvendigt, og det tilrådes at lade de benyttede prioriteter ligge samlet fra 0 og opefter, f.eks. 0,1,2 og 3. En korutine kan uden videre ændre sin egen prioritet, men bør normalt ikke pille ved andres.
Besked (2 ord)	Betydningen af dette felt vil blive givet i forbindelse med vejledningen i brug af beskedsemaforer.

Udover disse felter, som benyttes af styresystemet, kan brugeren tilføje så mange som han har brug for.
I korutinebeskrivelsen kan man f.eks. gemme indholdet af nogle registre, som man ønsker at retablere, når man bliver kørende igen efter et ventepunkt. Registrene kunne gemmes og retableres af systemet, men jeg har

ment at denne administration ville være overflødig i alt for mange tilfælde og har derfor overladt det til de enkelte korutiner, hvis de selv ønsker det.
For korutiner med reentrant kode er det særlig vigtigt, at korutinerne frit kan disponere over korutinebeskrivelsen ud over de første 7 ord; det er nemlig den eneste mulighed de har for at have hver sine variable at operere på. Adressen på korutinebeskrivelsen vil være kendt for korutiner med "privat" kode, mens en korutine med reentrant kode skal bruge systemets pegepind til korutinebeskrivelsen for den aktuelle KØRENDE korutine for at finde frem til deres egen beskrivelse.

Brug af stakken:
Mellem to ventepunkter kan en korutine benytte stakken frit, men ved et ventepunkt skal den være tom; det går i hvert fald galt hvis mere end én korutine forsøger at lade noget overleve på stakken gennem et ventepunkt. Stakken udnyttes ved ventepunktskald, idet styresystemet forventer at finde den adresse, hvorfra den kaldende korutine ønsker at fortsætte når den atter bliver KØRENDE, øverst på stakken. Herfra kan styresystemet så hente den og anbringe den i korutinebeskrivelsen. Det klares normalt ved at ventepunkter kaldes som subrutinekald, hvorved retursdressen kommer til at stå øverst på stakken.

IV.2.B Semaforer

Beskedsemaforer

To korutiner kan udveksle beskeder via en beskedsemafor. En beskedbuffer har følgende format:

1. En hægte der benyttes af systemet (2 ord)
2. Den besked der skal udveksles (0 - ? ord)

Den korutine der udfører "send"-operationen opgiver ved kaldet adressen på beskedbufferen. Korutinen der udfører "receive"-operationen vil have adressen på beskedbufferen i beskedfeltet af sin korutinebeskrivelse, når den igen bliver KØRENDE.

Format for semaforer

Både generelle semaforer og beskedsemaforer har følgende format:

1. Semaforværdien som har forskellig betydning for de to typer (1 ord)
2. Hovedet for semaforkøen som systemet bruger (5 ord)

IV.2.C Systemkald

Der findes to typer af systemkald i MIK: subrutinekald og ventepunktskald. Når en korutine er blevet kørende udfører den kode fra den ordre, som den lokale ordretæller i korutinebeskrivelsen peger på, og indtil den foretager et ventepunktskald; ind i mellem kan korutinen kalde systemsubrutiner så ofte den vil. De mulige kald er følgende:

Ventepunkter:	Subrutiner:
release	intactive
passivate	reactivate
wait	signal
receive	send

Efter subrutinekaldet kan registrene være uændrede, udefinerede eller indeholde resultatet af subrutinen. Stakken er uændret og kan frit bruges til f.eks. at gemme registre, så de kan retableres efter subrutinekaldet.
Efter ventepunktskaldet er indholdet af registrene udefineret, og der kan være ændret i fælles variable af andre korutiner, der har kørt i mellemtiden.

navn: release
type: ventepunkt
parametre: ingen
Release angiver et ventepunkt og benyttes når man har en korutine med et stort sammenhængende tidsforbrug. Hvis en korutine med højere prioritet ikke må risikere at vente for længe med at blive KØRENDE, når den er blevet AKTIV, kan ekstra ventepunkter indsættes i de lavere prioriterede korutiner på denne måde.

navn: passivate
type: ventepunkt
parametre: ingen
En korutine der ikke ønsker at køre mere kan kalde passivate. Kaldet bevirker at korutinen får status "passiv" og ikke kommer til at køre igen før en anden korutine aktiverer den med kaldet reactivate.

navn: reactivate
type: systemkorutine
parametre: adressen på korutinebeskrivelsen for den korutine der ønskes aktiveret i (D,E) registrene.

registre:	A:	B,C:	D,E:	H,L:
før:	-	-	korutinenavn	-
efter:	status	udefineret	uændret	udefineret

Status er 0 hvis kaldet bevirkede at den angivne korutine blev aktiveret, og 1 hvis den angivne korutine ikke var mærket som "passiv"; i det sidste tilfælde laver subrutinen ikke andet end at sætte status.
Passivate/reactivate kan undværes, idet man kan indføre en semafor for hver korutine, som kan tænkes at komme i tilstanden "passiv"; passivate/reactivate kan da erstattes af wait/signal-operationer på disse semaforer. I mange tilfælde vil det dog nok være simplere og mindre pladskrævende at anvende de specielle monitorkald.

navn: wait
type: ventepunkt
parametre: adressen på en generel semafor i (B,C)
Hvis semaforværdien er 0 forbliver den kaldende korutine AKTIV og semaforværdien mindskes med 1.
Hvis semaforværdien = 0 bliver den kaldende korutine VENTENDE.

navn: signal
type: subrutine
parametre: adressen på en generel semafor i (B,C)

registre:	A:	B,C:	D,E:	H,L:
før:	-	semafor	-	-
efter:	udefineret	udefineret	udefineret	udefineret

Hvis der er mindst én VENTENDE korutine på den angivne semafor, bliver den der har ventet længst gjort AKTIV, ellers øges semaforværdien med 1.

navn: receive
type: ventepunkt
parametre: adressen på en beskedsemafor i (B,C)

Hvis den angivne semafor har nogen ventende beskeder bliver adressen på den beskedbuffer der har ventet længst anbragt i beskedfeltet i den kaldende korutines beskrivelse; den kaldende korutine forbliver AKTIV.
Hvis der ikke er nogen ventende beskeder bliver den kaldende korutine VENTENDE.

navn: send
type: subrutine
parametre: adressen på en beskedsemafor i (B,C)
adressen på en beskedbuffer i (H,L)

registre:	A:	B,C:	D,E:	H,L:
før:	-	semafor	-	besked
efter:	udefineret	udefineret	udefineret	udefineret

Hvis der er nogen VENTENDE korutiner på den angivne semafor, får den korutine, der har ventet længst, anbragt adressen på beskedbufferen i beskedfeltet i sin korutinebeskrivelse, hvopå den bliver gjort AKTIV. I modsat fald anbringes beskeden i semaforens kø.

navn: intactive
type: subrutine
parametre: adressen på korutinebeskrivelsen for den korutine der skal aktiveres i (D,E)

registre:	A:	B,C:	D,E:	H,L:
før:	-	-	korutinenavn	-
efter:	udefineret	udefineret	uændret	udefineret

Denne subrutine bruges ved initialisering til at gøre korutiner AKTIVE.
For at intactive skal kunne bruges må den korutine, der angives som parameter, have status AKTIV, og man skal være sikker på at den ikke hænger i nogen systemkø i forvejen.
Hvis der er mulighed for at subrutinen kan blive afbrudt af en afbrydelsesrutine skal den kaldende korutine lukke for afbrydelser før kaldet (di) og åbne igen efter kaldet (ei).

Eksempel på systemkald:

Et systemkald kan f.eks. se således ud:

```
        lxi  b,bs       ;adressen på beskedsemaforen bs
                        ;anbringes i (B,C)
        lxi  h,bb       ;adressen på beskedbufferen bb
                        ;anbringes i (H,L)
        call send       ;subrutinen send kaldes
        =
        =
bs:     =
        =
bb:     =
        =
```

IV.2.D Tidsmåling

Hvis man har brug for tidsmåling i sit syatem, skal der være en beskedsemafor "sleeping" til brug for styresystemet. En korutine kan da med et kald af "send" sende en besked til "sleeping" med følgende format:

1. Hægteelement til brug for systemet (2 ord)
2. Antal af tidsenheder (2 ord)
3. Svarbeskedsemafor (2 ord)
4. Eventuelt yderligere information (0-? ord)

Når taktsignalet har afbrudt så mange gange som antallet af tidsenheder angiver, bliver beskeden af styresystemet returneret til den svarsemafor, der er opgivet. Korutinen kan altså sende en sådan besked til "sleeping", og derpå udføre så meget den har lyst til inden den sætter sig til at vente på svarsemaforen, som kan være en vilkårlig beskedsemafor.
Det antal tidsenheder der opgives skal ligge i to ord med det mindst betydende ord i den laceste adresse. Det giver mulighed for antal mellem 0 og 65535. Hvis taktsignalet f.eks. afbryder hvert 20. ms (50 Hz), er det største tidsrum man kan opnå lige over 20 minutter, men længere perioder kan måles ved at gentage operationen et passende antal gange.

IV.2.E Drivprogrammer

Dette afsnit rummer forskellige eksempler på hvorledes et drivprogram kan laves ved hjælp af en drivkorutine (dr) og en afbrydelsesrutine (afb). Koden er skitseret i algol; dog er afbrydelsesrutinernes returhopsordrer "ret" medtaget.
Visse faste regler skal overholdes:

1. Afbrydelsesrutiner må kalde systemsubrutiner som signal, send og reactivate, men ventepunktskald er ikke tilladt. Begrundelsen er at afbrydelsesrutinen ikke svarer til KØRENDE korutine når afbrydelsen kommer.

2. Mikrodatamaten vil formodentlig altid være indrettet så der er lukket for nye afbrydelser når en afbrydelse er kommet igennem. Det er afbrydelsesrutinens opgave at åbne for afbrydelser igen med en ei-ordre. Åbningen kan ske med det samme eller lige for ret-ordren på grund af intel8080's stakstruktur, der tillader afbrydelsesrutiner at afbryde hinanden i vilkårlig mange niveauer.

3. Status og registre skal gemmes så snart afbrydelsen kommer og retableres før "ret". Det sker nemmest med ordrerne:

```
push psw
push b          der gemmer status,A,
push d          B,C,D,E,H og L på
push h          stakken, og

pop h
pop d           der
pop b           retablerer
pop psw         dem igen.
```

Eksempel 1

Koden i drivkorutinen skal udføres 1 gang for hver 100 afbrydelser. Hvis afbrydelserne kommer med jævne mellemrum virker det som et ur:

```
afb: tæller := tæller + 1;
     if tæller = 100 then
     begin
          signal(D);
          tæller := 0;
     end;
     ret

dr: wait(D);
    =
    =
    goto D;
```

Eksempel 2 Udskrivning af en databuffer:

Eksemplet viser, hvordan flere processer kan få udskrevet deres data ved at sende dem i en databuffer af vilkårlig længde til en drivkorutine. Metoden sikrer at hele databufferen udskrives samlet.

```
afb: if tegn = sidstetegn then signal(SLUT)
     else
     begin
          tegn := tegn + 1;
          start skrivning af lager(tegn);
     end;
     ret

dr:  receive(D, (databuffer,svarsemafor));
     tegn := første tegn i databufferen;
     sidstetegn := sidste tegn i databufferen;
     start skrivning af det første tegn;
     wait(SLUT);
     send(svarsemafor,databuffer);
     goto dr;

brugerprocesser:  =
                  =
                  send(D,(databuffer,SVAR) );
                  =
                  =
                  receive(SVAR,databuffer);
                  =
                  =
```

Eksempel 3 En anden metode til udskrivning af en databuffer:
Dette eksempel adskiller sig fra eksempel 2 ved at næsten alt arbejdet er flyttet fra afbrydelsesrutinen til drivkorutinen. Metoden er ubetinget mere tung, da der skal udføres en "signal"-operation for hvert tegn. Fordelen er imidlertid, at drivkorutinen kan være meget lavt prioriteret, og udskrivningen af et tegn vil da kun ske når der ikke foregår vigtigere ting i systemet. Med metoden i eksempel 2 er man derimod pisket til at få skrevet hele databufferen ud med den hastighed, som den ydre enhed bestemmer, når man først er begyndt.

```
afb:    signal(KLAR);
        ret

dr:     receive(D, (databuffer,svarsemafor));
        tegn :=       første tegn i databufferen;
        sidstetegn := sidste tegn i databufferen;

skriv:  start skrivning af lager(tegn);
        wait(KLAR);
        if tegn < sidstetegn then
        begin
             tegn := tegn + 1;
            goto skriv;
        end;

        send(svarsemafor,databuffer);
        goto dr;

brugerprocesser: =
                 =
                 send(D ,(databuffer,SVAR);
                 =
                 =
                 receive(SVAR,databuffer);
                 =
                 =
```

Eksempel 4 Afbrydelse af skrivning i eksempel 2:
Hvis afbrydelsesrutinen i eksempel 2 udvides med:

```
afb: bestem hvor afbrydelsen kom fra;
     if afbrydelsestype = BREAK then signal(SLUT);
     else
     if tegn = sidstetegn then signal(SLUT);
     else ... (som før)
```

kan man ved hjælp af en knap eller tast manuelt standse en uønsket udskrift uden at systemet mærker forskel.

Eksempel 5 Drivprogram der kun består af en afbrydelsesrutine:

```
afb: reactivate(korutine);
     ret
```

På denne måde kan man lade en ydre hændelse starte en korutine i systemet. Reactivate er indrettet så kaldet ignoreres hvis korutinen skulle vise sig at være ikke-"PASSIV".

Eksempel 6 Reentrant drivprogramkode:

Hvis der i et system f.eks. er mange ens terminaler kan det maske betale sig at lave drimprogramkoden reentrant. Her er eksempel 2 skrevet om:

```
afb: afgør hvilken terminal afbrydelsen kommer fra;
     kb := drivkorutinebeskrivelsen svarende til terminalen;
     if kb.tegn = kb.sidstetegn then signal(kb.SLUT);
     else
     begin
          kb.tegn := kb.tegn + 1;
          start skrivning af lager(kb.tegn)
     end;
     ret

dr:  receive(KØRENDE.D,(databuffer,svarsemafor));
     KØRENDE.databuffer := databuffer;
     KØRENDE.svarsemafor := svarsemafor;
     KØRENDE.tegn :=       første tegn i databuffer;
     KØRENDE.sidstetegn := sidste tegn i databuffer;
     start skrivning på KØRENDE.terminal af første tegn;
     wait(KØRENDE.SLUT);

     send(KØRENDE.svarsemafor,KØRENDE.databuffer);
     goto dr;

brugerprocesser:  =
                  =
                  send(terminal.D, (databuffer,SVAR));
                  =
                  =
                  receive(SVAR,databuffer);
                  =
                  =
```

Til hver terminal svarer en korutinebeskrivelse, og afbrydelsesrutinen er - udfra kendskab til hvilken terminal afbrydelsen kom fra - i stand til at finde den rigtige korutinebeskrivelse. I hver korutinebeskrivelse er der plads til to semaforer, den besked korutinen for tiden arbejder på samt oplysning om hvorlangt korutinen er nået i udskriften af databufferen.

IV.2.F Den tomme korutine

Hvis det kørende system ikke er overbelastet vil det nu og da være i en venteposition, hvor ingen korutiner ønsker at køre. For at klare denne situation skal et system under MIK altid indeholde en korutine, som kører på laveste prioritetsniveau, og som altid er AKTIV. Den behøver ikke at lave andet end:

```
dummy:  call release
        jmp  dummy
```

men kan selvfølgelig udvides efter behov.

IV.2.G Initialisering af systemet

Først skal alle kæder i aktivkøen initialiseres til at have længde 0.

Semaforer skal altid initialiseres, men det er ikke altid de skal have semaforværdi og kølængde sat til 0.

Korutiner kan f.eks. komme ind i systemet således:

kb er adressen på korutinebeskrivelsen

```
lxi  h,startadresse      ;korutinen starter udførelsen i startadresse
shld kb+2                ;når den bliver KØRENDE
mvi  a,prioritet         ;korutinens prioritet sættes samtidig med at
sta  kb+4                ;status sættes til 0 (ikke-passiv)
lxi  d,kb                ;korutinen aktiveres
call intactive
```

Dette kan gøres for alle korutiner i systemet.
Korutiner kan også oprettes og nedlægges løbende; det kan f.eks. ske ved at hente en korutinebeskrivelse fra en pulje og frigive den igen når den ikke har mere at lave. Den nemmeste måde at lade en korutine "uddø" på er at lade den ende med at kalde passivate, da den dermed slipper for at hænge i nogen systemkæde, som den skal fjernes fra.

Når initialiseringen er slut hoppes til systemindgangen "common", hvorpå den korutine der er højest prioriteret og blev aktiveret først bringes til at køre.

IV.3 Programbeskrivelse

Den detaljerede programbeskrivelse foregår dels ved hjælp af pseudo-SIMULA og dels ved hjælp af illustrationer.
Først kommer der er SIMULA-program, der beskriver hele systemets opbygning, bortset fra koden for de enkelte rutiner, som er udeladt på dette sted. Derefter kommer der en oversigt over bestanddelene i systemet - semaforer, korutinebeskrivelser, kæder o.s.v., og endelig følger en beskrivelse af hver enkelt rutine.

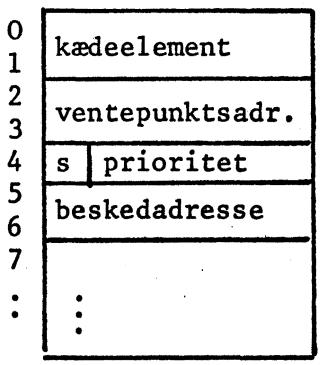
```
class MIK;
begin

simple type address;
begin integer low, high;
end +++ address +++ ;

class element;
begin ref(element) link;
end +++ element +++ ;

element class coroutinedescription;
begin
     address LPC;
     integer status, priority;
     address mes;
end +++ coroutinedescription +++ ;

element class messagebuffer;
begin integer array message(0:?);
end +++ messagebuffer +++ ;

class chain;
begin
     integer length;
     ref(element) first, last;
     procedure into ...;
     procedure firstout ...;
     length := 0;
end +++ chain +++ ;
```

```
chain class semaphore;
begin integer semaphorevalue;
      semaphorevalue := 0;
end +++ semaphore +++ ;

semaphore class gsemaphore;
begin
     procedure wait ...;
     procedure signal ...;
end +++ gsemaphore +++ ;

semaphore class msemaphore;
begin
     procedure receive ...;
     procedure send ...;
end +++ msemaphore +++ ;

procedure release ...;

procedure reactivate ...;

procedure passivate ...;

procedure intactive ...;

address RUNNING;
ref(chain) array ACTIVE(0:maxpriority);
ref(msemaphore) SLEEPING;
ref(coroutinedescription) cd, dummy;
ref(messagebuffer) mess;
integer priority;
integer status, active, passive;

procedure iclock ...;
```

```
for priority := 0 step 1 until maxpriority do
ACTIVE(priority) :- new chain;
SLEEPING :- new msemaphore;
active := 0;
passive := 1;

dummy :- new coroutinedescription;
dummy.priority := maxpriority;
dummy.LPC := DUMMY;
intactive(dummy);


... brugerprogram...;


common: ...


DUMMY: while true do release;

end +++ MIK +++ ;
```

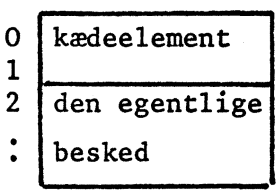
Systemelementer:

Korutinebeskrivelse: Coroutinedescription:



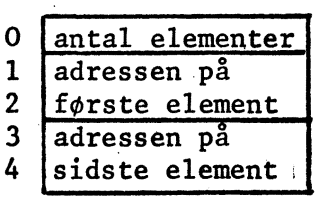
s er status, som fylder 1 bit, der er 1 for passiv og 0 for aktiv. Når korutinen ikke er passiv er hele ordets værdi den samme som prioriteten, som står i de sidste 7 bit og kan antage værdierne 0 - 127. Ud over de faste 7 ord kan korutinebeskrivelsen være så lang det skal være.

Beskedbuffer: Messagebuffer:



Systemet bruger de to første ord, mens resten kan fylde og indeholde hvad som helst.

Kæde: Chain:

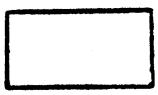


Antallet af elementer i en kæde kan være fra 0 til 127; hvis kæden er tom er de to pegepinde udefinerede. Elementerne i en kæde kan være korutinebeskrivelser eller beskedbuffere.

Systemelementer:

Der gælder altid i flerordsvariable, hvilket i almindelighed vil sige adresser, at det mindst betydende ord ligger i den laveste adresse.

RUNNING:

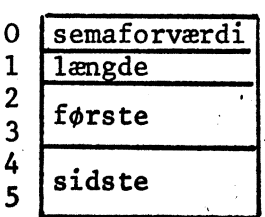


Adressen på korutinebeskrivelsen for den korutine der i øjeblikket er KØRENDE.

ACTIVE:

Aktivkøen består af en kæde af korutinebeskrivelser for hver prioritet. 0 er den højest mulige prioritet og der skal være så mange kæder, som der er benyttede prioriteter. Kædehovederne skal ligge lige efter hinanden i lageret.

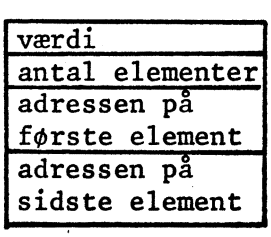
SLEEPING:



Kæden på SLEEPING-semaforen består af de beskeder, som skal returneres til en svarsemafor efter et givet tidsrum. Semaforværdien bliver aldrig -1, svarende til at der aldrig er korutiner, der venter på SLEEPING.

Generel semafor: Gsemaphore:

Besked semafor: Msemaphore:



Værdien af en generel semafor kan være 0,1,2,..., og antallet af elementer er altid antallet af korutinebeskrivelser i kæden.
Værdien af en beskedsemafor kan være -1,0 eller 1. Antallet af elementer er antal elementer i kæden - beskeder når værdi er 1, korutiner når den er -1 og tom når værdien er 0.

INTO: hjælpesubrutine for systemet

registre:

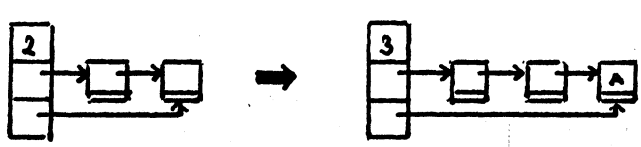
	før:	efter:
A:	-	?
B,C:	chain	?
D,E:	element	element
H,L:	-	?

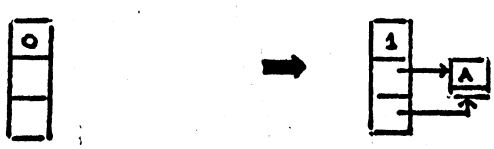
virkning: "element" som er adressen på en korutinebeskrivelse eller en beskedbuffer indsættes i kæden "chain".

kode:

```
procedure into(elem);
ref(element) elem;
begin
     if length > 0 then
     begin
          length := length + 1;
          last.link :- elem;
          last :- elem;
     end
     else
     begin
          length := 1;
          first  :- elem;
          last   :- elem;
     end
end +++ into +++ ;
```

INTO(A):





FIRSTOUT: hjælpesubrutine for systemet

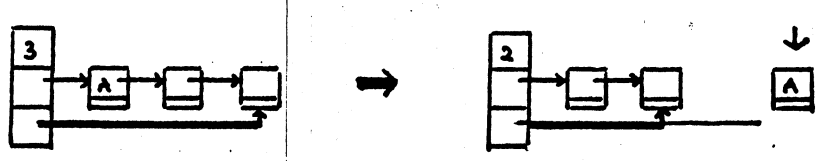
registre:

	før:	efter:
A:	-	status
B,C:	chain	?
D,E:	-	element
H,L:	x	x

virkning: Hvis status er 0 er det forreste element i kæden "chain" fjernet og en pegepind til det anbragt i "element". Hvis dette ikke var muligt fordi kæden var tom er status 1.

kode:

```
ref(elem) procedure firstout;
begin
     if length > 0 then
     begin
          length := length - 1;
          firstout :- first;
          first :- first.link;
          status := 0;
     end
     else status := 1;
end +++ firstout +++ ;
```



WAIT: ventepunkt

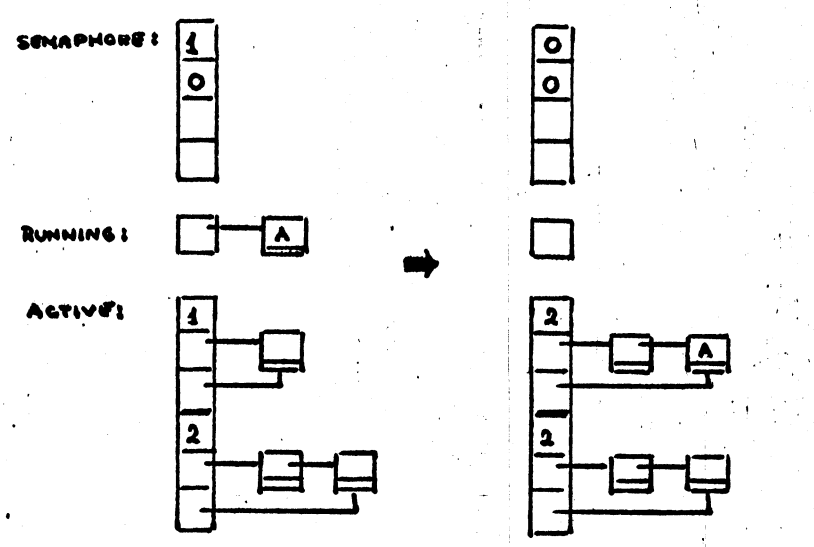
registre: før:

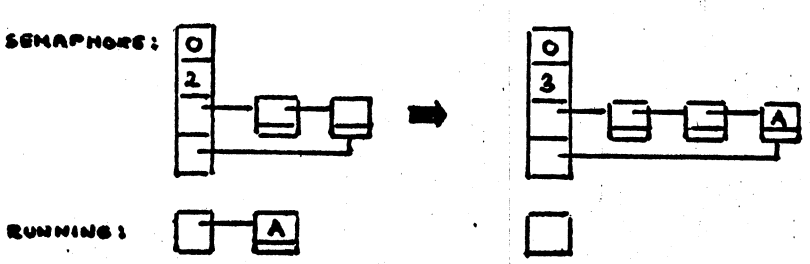
B,C: gsemaphore

virkning: Hvis den generelle semafors værdi er større end 0 skal den reduceres med 1 og den KØRENDE korutine skal forblive AKTIV. I modsat fald indsættes den i semaforkøen.

kode:

```
procedure wait;
begin
     disable interrupts;
     RUNNING.LPC := return address from stack;
     if semaphorevalue > 0 then
     begin
          semaphorevalue := semaphorevalue - 1;
          intactive(RUNNING);
     end
     else into(this chain,RUNNING);
     goto common;
end +++ wait +++ ;
```





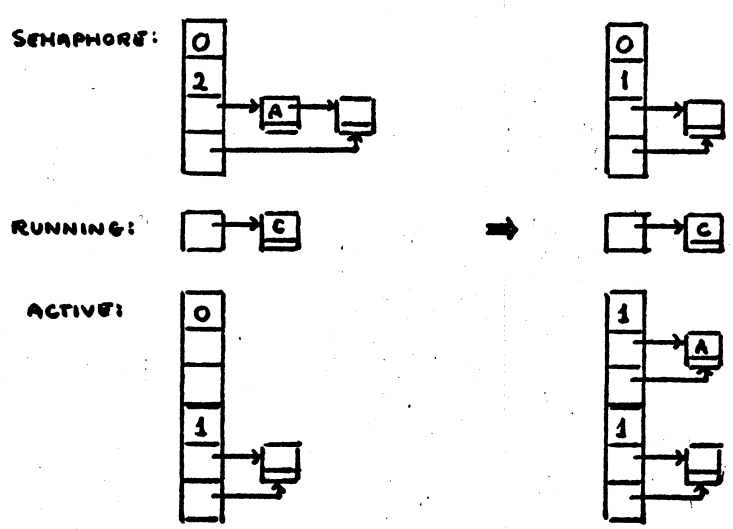
SIGNAL: subrutine

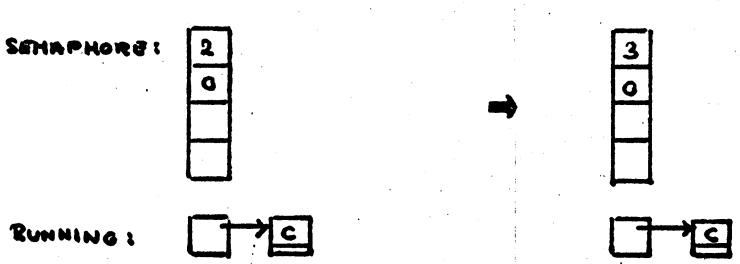
registre:		før:	efter:
	A:	-	?
	B,C:	gsemaphore	?
	D,E:	-	?
	H,L:	-	?

virkning: Hvis der er ventende korutiner aktiveres den forreste i køen, ellers øges semaforværdien med 1.

kode:

```
procedure signal;
begin
     disable interrupts;
     if length = 0 then semaphorevalue := semaphorevalue + 1;
                   else intactive(this chain.firstout);
     enable interrupts;
end +++ signal +++ ;
```





RECEIVE: ventepunkt

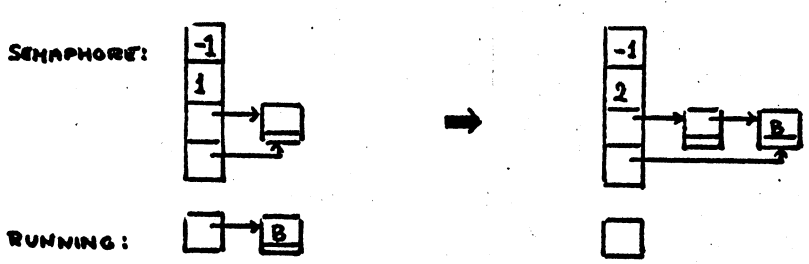
registre: før:

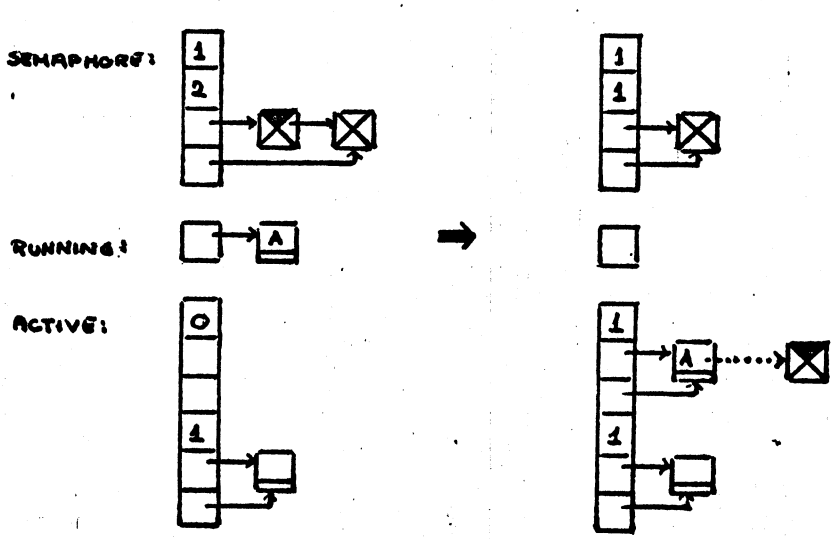
B,C: msemaphore

virkning: Hvis der er en beskedbuffer i kø gives den forreste i køen til den KØRENDE korutine, som da forbliver AKTIV; ellers anbringes den i semaforkøen.

kode:

```
procedure receive;
begin
     disable interrupts;
     RUNNING.LPC := return address from stack;
     if semaphorevalue > 1 then
     begin
          semaphorevalue := -1;
          this chain.into(RUNNING);
     end
     else
     begin
          if length = 1 then semaphorevalue := 0;
          RUNNING.mes := this chain.firstout;
          intactive(RUNNING);
     end;
     goto common;
end +++ receive +++ ;
```





SEND: subrutine

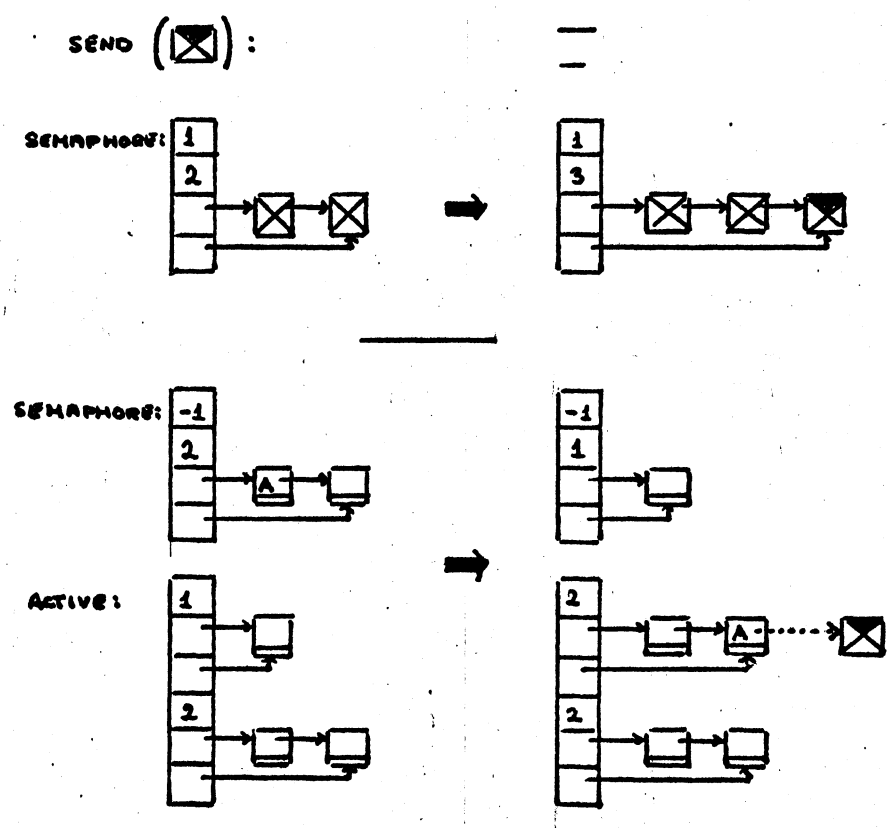
registre:

	før:	efter:
A:	-	?
B,C:	msemaphore	?
D,E:	-	?
H,L:	messagebuffer	?

virkning: Hvis der er korutiner i kø til beskedsemaforen "msemaphore", gives beskeden "messagebuffer" til den forreste, som da aktiveres; ellers anbringes beskeden i semaforkøen.

kode:

```
procedure send(message);
ref (messagebuffer) message;
begin
     disable interrupts;
     if semaphorevalue > -1 then
     begin
          semaphorevalue := 1;
          this chain.into(message);
     end
     else
     begin
          if length = 1 then semaphorevalue := 0;
          cd :- this chain.firstout;
          cd.mes := message;
          intactive(cd);
     end;
     enable interrupts;
end +++ send +++ ;
```



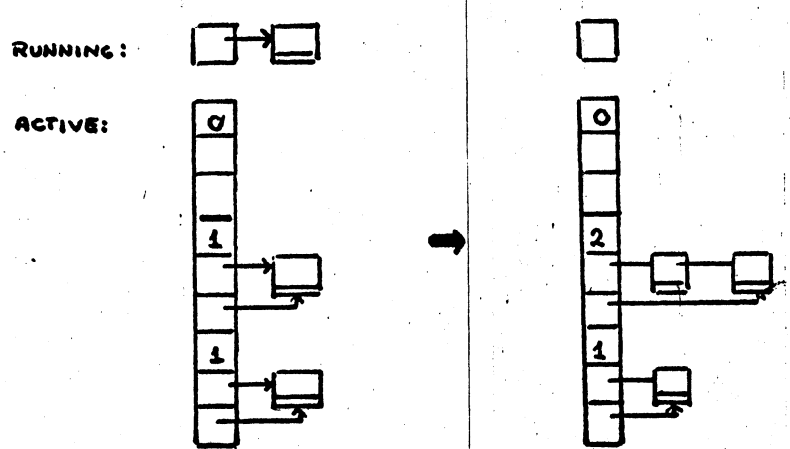
RELEASE: ventepunkt

registre: -

virkning: Den KØRENDE korutine giver plads for andre korutiner og forbliver selv AKTIV.

kode:

```
procedure release;
begin
     disable interrupts;
     RUNNING.LPC := return address from stack;
     intactive(RUNNING);
     goto common;
end +++ release +++ ;
```



REACTIVATE: subrutine

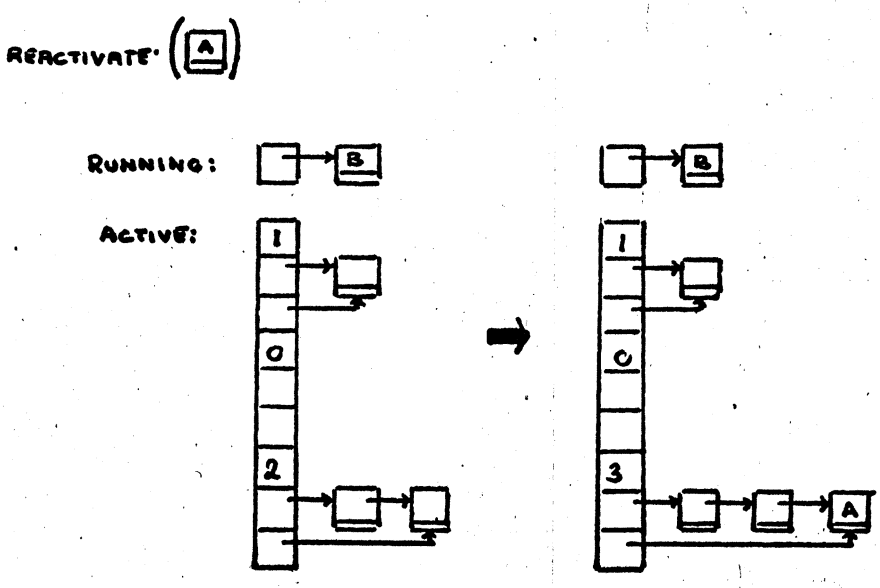
registre:

	før:	efter:
A:	-	status
B,C:	-	?
D,E:	cd	cd
H,L:	-	?

virkning: Hvis korutinen "cd" er PASSIV bliver den gjort AKTIV, og i så fald sættes status til 0, i modsat fald sættes status til 1.

kode:

```
procedure reactivate(cd);
ref(coroutinedescription) cd;
begin
     disable interrupts;
     if cd.status ≠ passive then status := 1;
     else
     begin
          cd.status := active;
          intactive(cd);
          status := 0;
     end;
     enable interrupts;
end +++ reactivate +++ ;
```

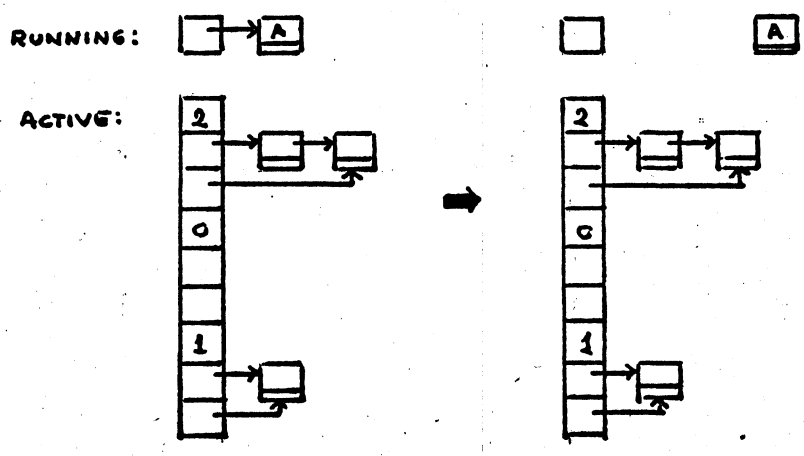


PASSIVATE: ventepunkt

registre: -

virkning: Den KØRENDE korutine får status som PASSIV.

kode:
```
procedure passivate;
begin
     disable interrupts;
     RUNNING.LPC := return address from stack;
     RUNNING.status := passive;
     goto common;
end +++ passivate +++ ;
```



INTACTIVE: subrutine

registre:

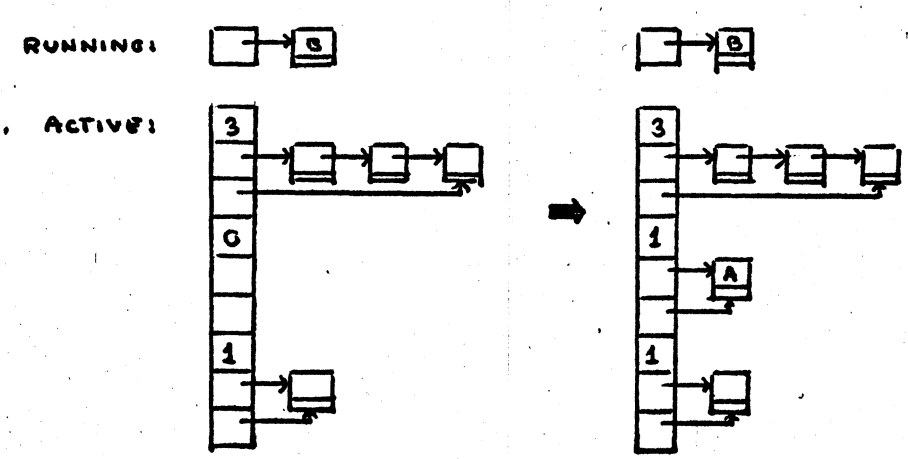
	før:	efter:
A:	-	?
B,C:	-	?
D,E:	cd	cd
H,L:	-	?

virkning: Korutinen "cd" indsættes i AKTIV-køen.

kode:

```
procedure intactive(cd);
ref(coroutinedescription)cd;
begin
     ACTIVE(cd.priority).into(cd);
end +++ intactive +++ ;
```

INTACTIVE (A):

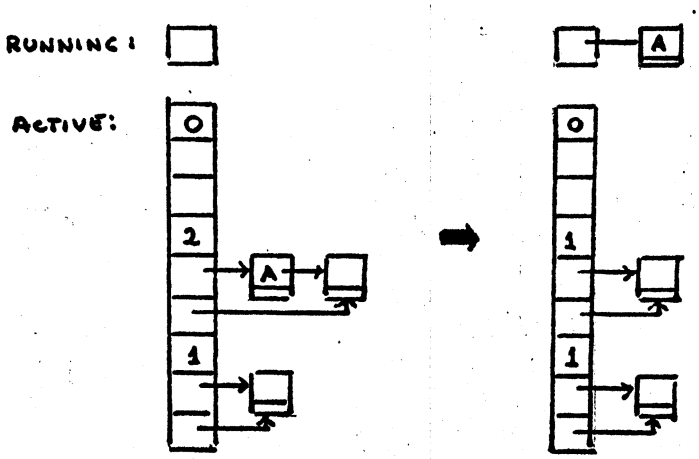


COMMON: fælles kode for alle ventepunkter

før: 1. Der er lukket for afbrydelser.
2. Den gamle KØRENDE korutine er anbragt hvor den skal være.
3. De aktioner der er specielle for det enkelte ventepunkt er udført.

virkning: Den nye KØRENDE korutine findes som den forreste i den første AKTIV-kæde, som ikke er tom.
Der sluttes med et hop til den lokale ordretæller for den nye KØRENDE korutine.

kode:
```
common:
begin integer priority;
      priority := 0;
      while ACTIVE(priority).length = 0 do
            priority := priority + 1;
      RUNNING :- ACTIVE(priority).firstout;
      enable interrupts;
      goto RUNNING.LPC;
end;
```

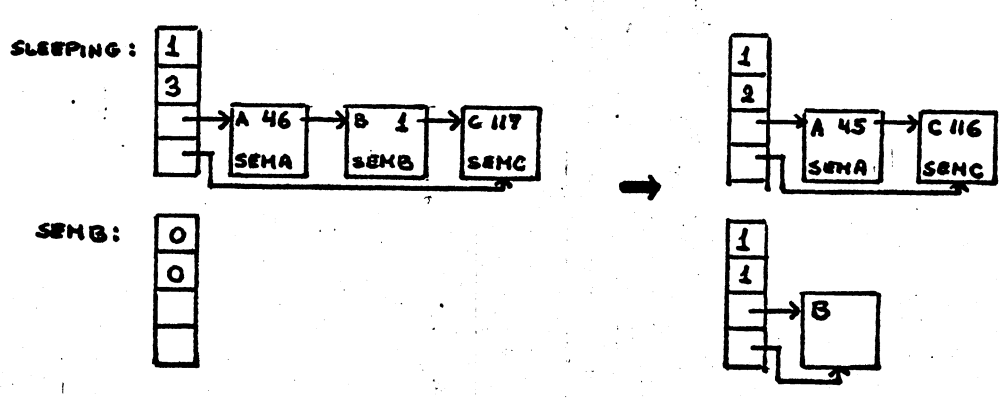


ICLOCK: afbrydelsesrutine for taktsignalet

virkning: Tidssignaltælleren for alle beskeder i SLEEPING-kæden reduceres med 1. Beskeder, hvis tæller bliver 0, pilles ud af kæden og sendes til den svarsemafor, som også står i beskeden.

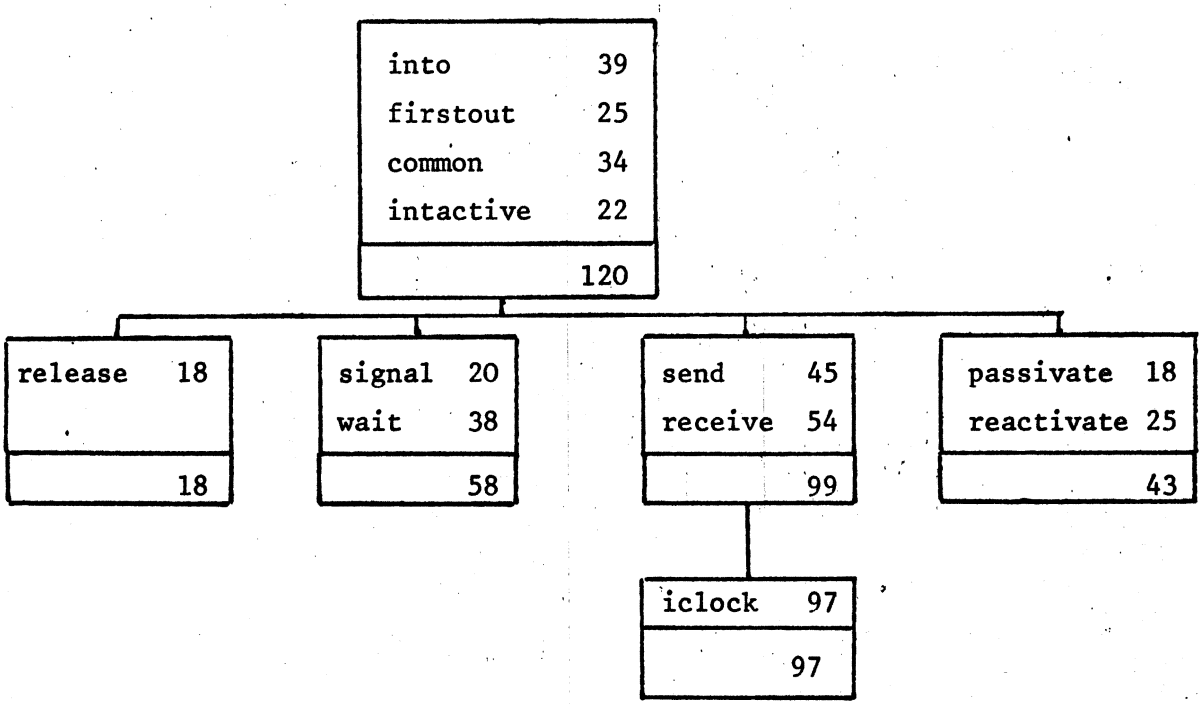
kode:

```
procedure iclock;
begin integer count;
      address x, old;

      save A and status;
      if SLEEPING.length   0 then
      begin
           save B,C,D,E,H,L;
           old   :- SLEEPING + 2;
           x     :- SLEEPING.first;
           count := SLEEPING.length;

           while count > 0 do
           begin
                x.delay := x.delay - 1;
                if x.delay = 0 then
                begin
                     old.link := x.link;
                     if x = SLEEPING.last then
                            SLEEPING.last := old;
                     send(x.answer,x);
                     SLEEPING.length := SLEEPING.length - 1;
                     x := old.link;
                end
                else
                begin
                     old := x;
                     x   := x.link;
                end;
                count := count - 1;
           end;

           restore B,C,D,E,H,L;
      end;
      restore A and status;
      enable interrupts;

end +++ iclock +++ ;
```



IV.4 Modulbeskrivelse

Det har været et grundlæggende krav at MIK skulle være opbygget, så det ikke skulle være nødvendigt at medtage dele, som ikke var nødvendige for den konkrete anvendelse. Nedestående er en redegørelse for hvilke dele der forudsætter hinanden, og hvor meget hver del fylder.



Korutinebeskrivelserne skal altid være på mindst 5 ord (kædeelement, lokal ordretæller, prioritet/status), og dertil kommer de 2 beskedord, hvis operationerne send og receive medtages.

Systemsemaforen sleeping (6 ord) er kun nødvendig hvis iclock medtages.

Aktivkøen skal bestå af så mange kæder à 5 ord, som der medtages prioriteter. Hvis kun prioritet 0 kendes kan man nøjes med 5 ord.

IV.5 Praktiske oplysninger

Assembleren der er anvendt er GENASS (17), som kører på RC4000. Jeg har forsøgt at lave koden så den er fuldt konvertibel med Intel's egen assembler (18) ved ikke at anvende specielle direktiver. Som det fremgår af udskrifterne af koden har jeg anvendt store og små bogstaver, men det har kun betydning for udseendet. Som koden foreligger på ASCII-form er der ikke forskel på store og små bogstaver.

Når systemet oversættes med GENASS kommer der en fejludskrift for hvert sideskift ("garbage"); det kan ignoreres.

I GENASS genkendes alle navne på de 4 første tegn, mens Intel's assembler genkender på 5 tegn, så der opstår ikke problemer. De anvendte navne er angivet i symboltabellen sammen med udskriften af koden i appendix A.

Systemkoden foreligger på papirstrimmel på FLEXO- og på ASCII-form.

IV.6 Tids- og pladsovervejelser

Sammen med udskriften af systemkoden fra assembleren er udlægningsadressen for den enkelte ordre angivet, hvilket gør det nemt at se hvad hver enkelt del af programmet fylder. I sin simpleste form fylder koden 178 ord og med alle implementerede dele 435 ord (se nærmere i afsnit IV.4 Modulbeskrivelse). Denne kode kan ligge i læselager. Dertil kommer "running", "active" og evt. "sleeping" kæderne; de fyldet mindst 7 ord (running + active med 1 prioritet), som ikke kan ligge i læselager, hvilket også gælder korutinebeskrivelserne (min. 5-7 ord), semaforkøer (6 ord) og beskedbuffere (2 ord + besked). Pladsforbruget for hver enkelt rutine i styresystemet er angivet i nedenstående tabel.

De tider der er angivet i tabellen ser bort fra små svingninger (højst 6 enheder), der afhænger af om et element indsættes i en tom kæde eller ej; jeg har altid regnet med den største tid. Enheden er antallet af lagercykler. Det eneste der kan siges om varigheden af en sådan er at den mindst er på 2 mikrosekunder, da det afhænger af andet end centralenheden.

Den kritiske størrelse for systemet er den maximale tid der kan være lukket for afbrydelser, hvilket er tilfældet i alle systemrutiner samt i afbrydelsesrutiner. Skemaet viser at receive er den operation, der kan tage længst tid. Det viser også at det i visse tilfælde vil være mere rimeligt at implementere et urdrivprogram sådan her:

```
afbrydelsesrutine: gem registre;
                   signal(UR);
                   retabler registre;
                   enable interrupts;
                   ret;

drivkorutine:      while true do
                   begin
                        wait(UR);
                        den nuværende afbrydelsesrutine;
                   end;
```

	plads	tid	kommentar til tidsforbrug
into	39	32	
firstout	25	31	8 hvis kæden er tom
intactive	32	55	
common	34	64+p•14	p = prioriteten af den næste KØRENDE korutine
release	18	145+p•14	
reactivate	25	80	
passivate	18	90+p•14	
signal	20	15	hvis der ikke bliver aktiveret nogen korutine
		106	hvis der bliver aktiveret en korutine
wait	38	131+p•14	hvis korutinen skal vente på semaforen
		156+p•14	hvis korutinen forbliver aktiv
send	45	52	hvis der ikke bliver aktiveret nogen korutine
		130	hvis der bliver aktiveret en korutine
receive	54	137+p•14	hvis korutinen skal vente på semaforen
		212+p•14	hvis korutinen forbliver aktiv
iclock	97	16	hvis der ikke er nogen beskeder i sleeping
		42+	
		N•38+	hvor N er antallet af beskeder hvor tælleren bliver talt ned med 1
		T•207+	hvor T er antallet af beskeder der bliver returneret og aktiverer en korutine
		S•129	hvor S er antallet af beskeder der bliver returneret uden at aktivere nogen korutine

V.1 Afprøvning

En "pæn" afprøvning af MIK ville indeholde rutiner til udskrivning af de forskellige kæder i systemet. Imidlertid er afprøvningen et punkt, hvor jeg har måttet se lidt bort fra det pæne for at nå at blive færdig. Afprøvningen er ikke blevet mindre grundig, idet man ved hjælp af simulatoren kan få udskrevet lagerets indhold på et givet tidspunkt, og sammenholdt med en udskrift fra assembleren, hvor alle lagringsadresserne står, kan man se hvad kæderne indeholder. Den afprøvning der medtages her har været kørt på denne måde, men det var ikke særlig pænt at se på. I stedet har jeg udnyttet at afprøvningen er opbygget af korutiner, som - hvis systemet virker korrekt - skal aktiveres i en bestemt rækkefølge. Dokumentationen for afprøvningen består derfor af en simulatorudskrift, der viser at ventepunkterne gennemløbes i den rigtige rækkefølge. Derudover er der medtaget nogle få lagerudskrifter, der viser at beskeder overføres korrekt fra den ene korutine til den anden. Fordelen er at det ikke bliver nødvendigt at fordybe sig i simulatorudskrifter, der ikke er særlig overskuelige, og specielt ikke er særlig læselige, da farvebåndet på den eneste tilgængelige ikke-skærm-terminal ved H.C.Ørstedsinstituttets RC4000-anlæg er i kronisk dårlig forfatning.

Den i rapporten dokumenterede afprøvning forsøger ikke at komme ud i absolut alle tilfælde; men den kommer ud i alle dele af koden, og illustrerer således at alle rutiner kan aktiveres, at semaforerne virker uanset rækkefølgen af signal/wait hhv. send/receive, at urafbrydelsesrutinen fungerer efter hensigten o.s.v.

Afbrydelsesrutinen er ikke afprøvet ved hjælp af afbrydelser, men ved hjælp af en korutine, der kalder afbrydelsesrutinen som subrutine.

Afprøvningen består af 3 programmer:

1. Afprøver beskedsemaforer og generelle semaforer i alle tilstande.

2. Indeholder kald af samtlige systemkorutiner.

3. Afprøver urafbrydelsesrutinen.

```
**********************************************************************
                          TABASCO
**********************************************************************
job nr :        13
tilmeldt fra datbodil

Afprøvningsprogram I

fp

dat2del   2.5.1975  14.41.35

**********************************

bs5
     0    cd1: 0,0,0,0,0,0,0
     7    cd2: 1,1,1,1,1,1,1
    14    cd3: 2,2,2,2,2,2,2
    21    cd4: 3,3,3,3,2,3,3
    28    active: 0,1,1,2,2
    33            0,1,1,2,2
    38            0,1,1,2,2
    43    running: 0,0
    45    sleeping: 0,0,1,1,2,2
    51
    51    lxi  d,cd3
    54    call intactive
    57    lxi  d,cd1
    60    call intactive
    63    lxi  d,cd2
    66    call intactive
    69
    69    lxi  h,l1
    72    shld cd1+2
    75    lxi  h,l2
    78    shld cd2+2
    81    lxi  h,l3
    84    shld cd3+2
    87    jmp  common
    90
    90    l1:
    90
    90    lxi  b,sem1      ]
    93    call signal      |
    96    lxi  b,sem2      |  1
    99    lxi  h,mes1      |
   102    call send        |
   105    lxi  b,sem3      |
   108    call receive     ]
   111
   111    lxi  b,sem2      ]  7
   114    call receive     ]
   117
   117    lxi  b,sem3      ]
   120    lhld cd1+5       |
   123    call send        |  8
   126    lxi  b,sem1      |
   129    call wait        ]
   132
   132    l2:
   132    lxi  b,sem1      ]
   135    call signal      |
   138    lxi  b,sem2      |  2
   141    lxi  h,mes2      |
   144    call send        ]
```

```
147   lxi  b,sem3      ]
150   call receive     ]
153                    - 9
153
153   l3:
153   lxi  b,sem1      ] 3
156   call wait        ]
159
159   lxi  b,sem1      ] 4
162   call wait        ]
165
165   lxi  b,sem2      ] 5
168   call receive     ]
171
171   lxi  b,sem3      ]
174   lhld cd3+5       ]  6
177   call send        ]
180   lxi  b,sem1      ]
183   call wait        ]
186
186   sem1: 0,0,0,0,0,0
192   sem2: 0,0,0,0,0,0
198   sem3: 0,0,0,0,0,0
204   mes1: 0,0,16,17
208   mes2: 0,0,26,27
212   mes3: 0,0,36,37
216
```

bs4 1000;

I
0 51

```
*X90
addr     90     reg   0   0   30   0   3   0   90   0   0   0   1   1
*X132
addr     132    reg   0   0   35   0   10  0   132  0   0   0   1   1
*X153
addr     153    reg   0   0   40   0   17  0   153  0   0   0   1   1
*X159
addr     159    reg   0   0   40   0   17  0   159  0   0   0   1   1
*X165
addr     165    reg   0   0   40   0   17  0   165  0   0   0   1   1
*X171
addr     171    reg   0   0   40   0   17  0   171  0   0   0   1   1
111
addr     111    reg   0   0   30   0   3   0   111  0   0   0   1   1
*X117
addr     117    reg   0   0   30   0   3   0   117  0   0   0   1   1
*X153
addr     153    reg   0   0   35   0   10  0   153  0   0   0   1   1
*0
```

nd 1262

```
 0   cd1: 0,0,0,0,0,0,0
 7   cd2: 1,1,1,1,1,1,1
14   cd3: 2,2,2,2,2,2,2
21   cd4: 3,3,3,3,2,3,3
28   active: 0,1,1,2,2
33           0,1,1,2,2
38           0,1,1,2,2
43   running: 0,0
45   sleeping:0,0,1,1,2,2
51
51   lxi  d,cd3
54   call intactive
57   lxi  d,cd1
60   call intactive
63   lxi  d,cd4
66   call intactive
69   lxi  d,cd2
72   call intactive
75
75   lxi  h,l1
78   shld cd1+2
81   lxi  h,l2
84   shld cd2+2
87   lxi  h,l3
90   shld cd3+2
93   lxi  h,l4
96   shld cd4+2
99   jmp common
102
102  l1:
102  call passivate   ] 1
105
105  mvi  h,100       ]
107  mvi  l,99        ] 5
109  shld mes1+2      ]
112  lxi  h,mes1      ]
115  lxi  b,sem1      ]
118  call send        ]
121  lxi  b,sem2      ]
124  call wait        ]
127  lxi  b,sem1      ] 12
130  call receive     ]
133
133  call passivate   ] 14
136
136
136  l2:
136  call passivate   ] 2
139
139  mvi  h,200       ]
141  mvi  l,199       ] 6
143  shld mes2+2      ]
146  lxi  h,mes2      ]
149  lxi  b,sem1      ]
152  call send        ]
155  lxi  b,sem2      ]
158  call wait        ]
161
161  lxi  h,mes3      ]
164  lxi  b,sem1      ] 13
167  call send        ]
170  lxi  b,sem2      ]
173  call wait        ]
176
176
```

```
176   l3:
176   call passivate      ] 3
179
179   lxi  b,sem1         ] 7
182   call receive        ]
185
185   lxi  b,sem1         ] 9
188   call receive        ]
191
191   lxi  b,sem2         ] 11
194   call signal         ]
197   lxi  b,sem2         ]
200   call signal         ]
203   lxi  b,sem2         ]
206   call signal         ]
209   call release        ]
212
212
212   l4:
212   lxi d,cd2           ] 4
215   call reactivate     ]
218   ana  a              ]
219   jnz  error          ]
222   lxi  d,cd3          ]
225   call reactivate     ]
228   ana  a              ]
229   jnz  error          ]
232                       ]
232   call reactivate     ]
235   ana  a              ]
236   jz   error          ]
239   lxi  d,cd1          ]
242   call reactivate     ]
245   ana  a              ]
246   jnz  error          ]
249                       ]
249   l5:                 ]
249   call release        ]
252
252   jmp  l5             ] 5, 8, 10
255
255   error: hlt
256
256   mes1: 0,0,16,17
260   mes2: 0,0,26,27
264   mes3: 0,0,36,37
268   sem1: 0,0,0,0,0,0
274   sem2: 0,0,0,0,0,0
280   sem3: 0,0,0,0,0,0
286
286
```

```
bs4 1000

*I
 51
*X102
addr   102    reg    0    0   30    0    3    0   102    0    0    0    1    1
*X136
addr   136    reg    0    0   35    0   10    0   136    0    0    0    1    1
*X176
addr   176    reg    0    0   40    0   17    0   176    0    0    0    1    1
*X212
addr   212    reg    0    0   40    0   24    0   212    0    0    0    1    1
*X105
addr   105    reg    0    0   30    0    3    0   105    0    0    0    1    1
*X139
addr   139    reg    0    0   35    0   10    0   139    0    0    0    1    1
*X179
addr   179    reg    0    0   40    0   17    0   179    0    0    0    1    1
*X252
addr   252    reg    0    0   40    0   24    0   252    0    0    0    1    1
*X185
addr   185    reg    0    0   40    0   17    0   185    0    0    0    1    1
*P 19 20
addr    19    0
addr    20    1
*X252
addr   252    reg    0    0   40    0   24    0   252    0    0    0    1    1
*X191
addr   191    reg    0    0   40    0   17    0   191    0    0    0    1    1
*P 19 20
addr    19    1
addr    20    1
*X127
addr   127    reg    0    0   30    0    3    0   127    0    0    0    1    1
*X161
addr   161    reg    0    0   35    0   10    0   161    0    0    0    1    1
*X133
addr   133    reg    0    0   30    0    3    0   133    0    0    0    1    1
*P 5 6
addr    5     8     1
*X176
addr   176    reg    0    0   35    0   10    0   176    0    0    0    1    1
```

```
cd1: 0,0,0,0,0,0,0
     6   cd2: 1,1,1,1,0,1,1
    13   cd3: 2,2,2,2,2,2,2
    20   cd4: 3,3,3,3,2,3,3
    27   active: 0,1,1,2,2
    32           0,1,1,2,2
    37           0,1,1,2,2
    42   running: 0,0
    44   sleeping:0,0,1,1,2,2
    50
    50   lxi  h,c1
    53   shld cd1+2
    56   lxi  h,c2
    59   shld cd2+2
    62   lxi  d,cd1
    65   call intactive
    68   lxi  d,cd2
    71   call intactive
    74
    74   lxi  h,sem1
    77   shld mes1+4
    80   lxi  h,sem2
    83   shld mes2+4
    86   lxi  h,sem3
    89   shld mes3+4
    92   jmp  common
    95
    95
    95
    95   c2:
    95   call iclock          ] 2
    98   call release         ]
   101
   101   jmp  c2              ] 4,5,7,8,10,11,12
   104
   104
   104   c1:
   104   call release         ] 1
   107
   107   mvi  a,4             ]
   109   sta  mes1+2          |  3
   112   mvi  a,2             |
   114   sta  mes2+2          |
   117   mvi  a,7             |
   119   sta  mes3+2          |
   122   lxi  b,sleeping      |
   125   lxi  h,mes1          |
   128   call send            |
   131   lxi  b,sleeping      |
   134   lxi  h,mes2          |
   137   call send            |
   140   lxi  b,sleeping      |
   143   lxi  h,mes3          |
   146   call send            |
   149   lxi  b,sem2          |
   152   call receive         ]
   155
   155   lxi  b,sem1          ] 6
   158   call receive         ]
   161
   161   lxi  b,sem3          ] 9
   164   call receive         ]
   167                        - 13
   167
   167
   167   mes1: 0,0,0,0,0,0
```

```
173    mes2: 0,0,0,0,0,0
179    mes3: 0,0,0,0,0,0
185    sem1: 0,0,0,0,0,0
191    sem2: 0,0,0,0,0,0
197    sem3: 0,0,0,0,0,0
203
```

DOKUMENTATION III

```
bs4 1000
*9
*I
0  51
*X105
addr   105     reg     0     0    30     0     3     0   105     0     0     0     1     1
*X96
addr    96     reg     0     0    30     0    10     0    96     0     0     0     1
*X108
addr   108     reg     0     0    30     0     3     0   108     0     0     0     1     1
*X102
addr   102     reg     0     0    30     0    10     0   102     0     0     0     1     1
*X96
addr    96     reg     0     0    30     0    10     0   102     0     0     0     1     1
*X102
addr   102     reg     0     0    30     0    10     0   102     0     0     0     1
*X156
addr   156     reg     0     0    30     0     3     0   156     0     0     0     1     1
*P5 6
addr     5   174     0
*X102
addr   102     reg     0     0    30     0    10     0   102     0     0     0     1     1
*X96
addr    96     reg     0     0    30     0    10     0   102     0     0     0     1     1
*X102
addr   102     reg     0     0    30     0    10     0   102     0     0     0     1     1
*X162
addr   162     reg     0     0    30     0     3     0   162     0     0     0     1     1
*P5 6
addr     5   168     0
*X102
addr   102     reg     0     0    30     0    10     0   102     0     0     0     1     1
*X168
addr   168     reg     0     0    30     0     3     0   168     0     0     0     1
*P5 6
addr     5   180     0
*P 45 50
addr    15     0     0     0     0    47
addr    50     0
*
```

(adresserne i dokumentationen er forskudt med 1 i forhold til udskriften)

V.2 Eksempel på anvendelse af MIK

Nedenstående er et eksempel på et system, der anvender MIK til udveksling af information mellem korutiner. Eksemplet skal særlig illustrere hvordan systemet skal initialiseres og startes. Nogle af de angivne initialiseringer vil dog eventuelt kunne undværes i nogle tilfælde. Visse centrale dele er ikke fast udformet, men fremtræder blot som kald af en passende subrutine. Eksemplet er lidt for småt til at være rigtig realistisk, men man kan forestille sig systemer, som kører på samme mønster, men med mere arbejde i hver korutine.

I eksemplet er der 6 korutiner:

læs1: Som aflæser en tæller og sender værdien som besked til korutinerne "beregn" og "sammenlign".

læs2: Som aflæser en anden tæller og sender værdien som besked til "sammenlign".

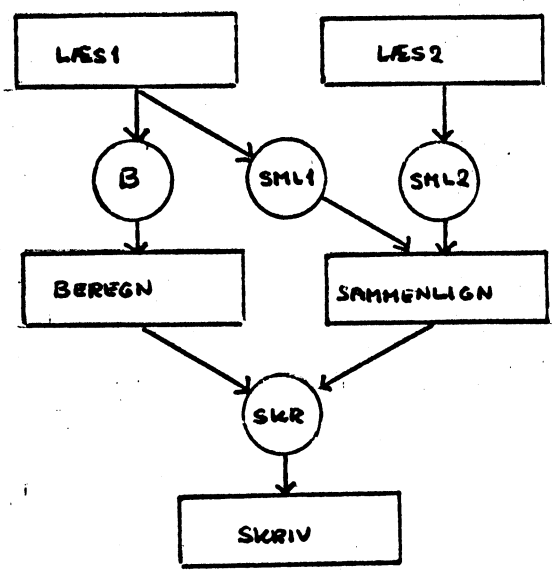
beregn: Som udfra hver værdi den modtager fra "læs1" beregner en ny værdi, som den sender videre til "skriv".

sammenlign: Som sammenligner de to værdier den får fra "læs1" og "læs2" og hvis de er ens sender den fælles værdi videre til "skriv".

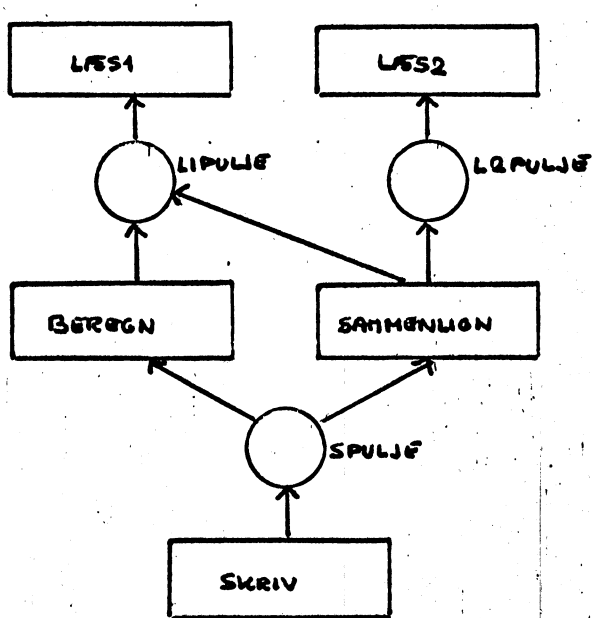
skriv: Som udskriver de værdier den modtager i den rækkefølge de ankommer.

dummy: Som ikke laver noget, og som eventuelt er overflødig, men som bør medtages.

Udvekslingen af beskeder mellem korutinerne foregår ved hjælp af beskedsemaforerne B, SML1, SML2 og SKR, som figuren viser:



Beskedbufrene administreres ved hjælp af beskedsemaforerne L1PULJE, L2PULJE og SPULJE. Umiddelbart skulle man tro, at der kunne bruges én fælles pulje af beskedbuffere, men det ville åbne mulighed for en baglåssituation, hvis f.eks. en af læserutinerne kom til at lægge beslag på alle bufrene samtidig.



Programtekst:

```
running:  0,0
active:   0,r.10      ;der medtages 2 prioritetsniveauer

;korutinebeskrivelser:
kb1læs:   0,r.7
kb2læs:   0,r.7
kbberegn: 0,r.7
kbsml:    0,r.7
kbskriv:  0,r.7
kbdummy:  0,r.7

;beskedsemaforer:
B:        0,r.6
SML1:     0,r.6
SML2:     0,r.6
SKR:      0,r.6
SPULJE:   0,r.6
L1PULJE:  0,r.6
L2PULJE:  0,r.6

;beskedbuffere:
b1:       0,0,0
b2:       0,0,0
b3:       0,0,0
b4:       0,0,0
b5:       0,0,0
b6:       0,0,0
b7:       0,0,0
b8:       0,0,0

;hjælpevariable:
l1gem:    0
begem:    0
smlgem:   0

;INITIALISERING:

;ventepunktsadressen i hver korutinebeskrivelse sættes til startadressen:

lxi  h,læs1
shld kblæs+2
lxi  h,læs2
shld kb2læs+2
lxi  h,beregn
shld kbberegn+2
lxi  h,sammenlign
shld kbsml+2
lxi  h,skriv
shld kbskriv+2
lxi  h,dummy
shld kbdummy+2
```

```
;initialisering af prioriteter -alle får prioritet 0 undtagen dummy:

mvi  a,0
sta  kb1læs+4
sta  kb2læs+4
sta  kbberegn+4
sta  kbsml+4
sta  kbskriv+4

mvi  a,1
sta  kbdummy+4


;initialisering af semaforer - alle får værdi og længde 0:

mvi  h,0
mvi  l,0
shld B
shld SML1
shld SML2
shld SKR
shld SPULJE
shld L1PULJE
shld L2PULJE

;initialisering af puljer:

lxi  d,b1            ;L1PULJE skal bestå af b1, b2 og b3
lxi  b,L1PULJE
call into
lxi  d,b2
lxi  b,L1PULJE
call into
lxi  d,b3
lxi  b,L1PULJE
call into

lxi  d,b4            ;L2PULJE skal bestå af b4 og b5
lxi  b,L2PULJE
call into
lxi  d,b5
lxi  b,L2PULJE
call into

lxi  d,b6            ;SPULJE skal bestå af b6, b7 og b8
lxi  b,SPULJE
call into
lxi  d,b7
lxi  b,SPULJE
call into
lxi  d,b8
lxi  b,SPULJE
call into
```

```
;aktivering af korutinerne:

lxi  d,kb1læs
call intactive
lxi  kb2læs
call intactive
lxi  kbberegn
call intactive
lxi  kbsml
call intactive
lxi  kbskriv
call intactive
lxi  kbdummy
call intactive


jmp  common


;++++
læs1:
;++++

lxi  b,L1PULJE          ;hent en tom buffer fra L1PULJE
call receive

lhld kb1læs+5           ;h,l-registeret peger nu på bufferen
inx  h
inx  h                  ;h,l-registeret peger nu på beskeden

lxi  b,tæller1          ;indlæs anbringer tællerens værdi i a-registeret
call indlæs             ;uden at ændre h,l

sta  l1gem              ;gem værdien
mov  m,a                ;anbring værdien i bufferen
dcx  h
dcx  h                  ;h,l peger nu atter på bufferen
lxi  b,B
call send               ;send(B,buffer)

                        ;den samme besked skal sendes til SML1:
lxi  b,L2PULJE
call receive            ;hent buffer

lhld kb1læs+5
inx  h
inx  h
lda  l1gem              ;hent værdien
mov  m,a                ;gem den i bufferen
dcx  h
dcx  h
lxi  b,SML1
call send               ;send(SML1,buffer)
jmp  læs1
```

```
;++++
læs2:
;++++

lxi  b,L2PULJE          ;hent buffer fra L2PULJE
call receive
lhld kb21æs+5           ;h,l peger på bufferen
inx  h
inx  h

lxi  b,tæller2          ;indlæs anbringer tæller2's værdi i a
call indlæs

mov  m,a                ;gem værdien i bufferen
dcx  h
dcx  h
lxi  b,SML2
call send               ;send(SML2,buffer)

jmp  læs2


;++++++
beregn:
;++++++

lxi  b,B                ;hent buffer fra B
call receive
lhld kbberegn+5
inx  h
inx  h
mov  a,m                ;anbring bufferværdien i a
push psw                ;gem a på stakken

lxi  b,L1PULJE          ;aflever den tomme buffer til L1PULJE
dcx  h
dcx  h
call send

pop  psw                ;hent værdien fra stakken

call beregning          ;denne subrutine beregner en værdi udfra a og
                        ; anbringer den på ny i a

sta  begem              ;gem denne værdi i begem

lxi  b,SPULJE           ;hent en tom buffer fra SPULJE
call receive
lhld kbberegn+5
inx  h
inx  h
lda  begem              ;hent værdien og anbring den i bufferen
mov  m,a

dcx  h
dcx  h
lxi  b,SKR
call send               ;send(SKR,buffer)
jmp beregn
```

```
;++++++++++
sammenlign:
;++++++++++

lxi  b,SML1             ;hent buffer
call receive
shld kbsml+5
inx  h
inx  h
mov  a,m                ;anbring værdien i a
sta  smlgem             ;gem værdien

dcx  h                  ;lever den tomme buffer tilbage
dcx  h
lxi  b,L1PULJE
call send

lxi  b,SML2             ;hent den anden værdi
call receive
shld kbsml+5
inx  h
inx  h
mov  a,m
push  psw               ;og gem den på stakken

dcx  h                  ;lever den tomme buffer tilbage
dcx  h
lxi  b,L2PULJE
call send

pop  psw                ;hent den ene værdi fra stakken
lxi  h,smlgem           ;og sammenlign den med den anden
cmp  m
jnz  sammenlign         ;begynd forfra hvis de er forskellige

lxi  b,SPULJE           ;hent skrivebuffer
call receive
lhld kbsml+5
inx  h
inx  h
lda  smlgem
mov  m,a                ;og anbring den fælles værdi i den

dcx  h
dcx  h
lxi  b,SKR
call send               ;send(SKR,buffer)

jmp  sammenlign
```

```
;+++++++
skriv:
;+++++++

lxi  b,SKR              ;hent en buffer
call receive
shld kbskriv+5
inx  h
inx  h
mov  a,m
push psw                ;gem indholdet på stakken

dcx  h                  ;aflever den tomme buffer
dcx  h
lxi  b,SPULJE
call send

pop  psw                ;hent værdien fra stakken
call udskriv            ;denne subrutine foretager udskriften

jmp skriv



;++++++
dummy:
;++++++

call release

jmp  dummy
```

VI. BESKRIVELSE AF SIMULATOREN

Den simulator der er anvendt til indkøring af MIK er skrevet i algol 6 og kører på H.C.Ørstedsinstituttets RC4000-anlæg.
Da jeg ikke havde adgang til andre simulatorer, og slet ikke til at køre direkte med en intel8080, var jeg nødt til at udvikle en selv. Det var mit ønske at komme så nemt om ved det som muligt, og derfor valgte jeg at bygge den op som en videreudvikling af en allerede eksisterende simulator til intel8008 på RC4000. En stærkt medvirkende årsag var at man på RC4000 har adgang til en assembler til intel8080. Simulatoren, som er skrevet af Jørgen Bang blev oprindelig skrevet fordi RC4000-afdelingen på H.C.Ø. ønskede at benytte den. Da de fik lejlighed til at udføre programmer direkte på en intel8008 blev simulatoren opgivet. Den blev overtaget af Arne Olesen, som foretog nogen ændringer i den. Senere arbejdede jeg selv med i et projekt - "Styring af en grafpen ved hjælp af en mikrodatamat", hvor den skulle benyttes, og i den anledning viste det sig nødvendigt at sætte sig ind i dens virkemåde og rette et par fejl. Det var derfor forholdsvis simpelt at ændre simulatoren til at udføre intel8080 kode i stedet for intel8008 kode.
Intel8080 er en kraftig udvidelse af intel8008, men det meste af grundstrukturen er det samme. Der er 7 registre, adresseringen foregår via H og L registrene, de arithmetiske operationer er de samme o.s.v. Der er kun ganske få punkter, hvor intel8080 ikke er en direkte udvidelse af intel8008; det drejer sig om stakken og om læse/skrive ordrer. Imidlertid var ligheden så stor at der kun skulle ændres væsentligt i tre dele af programmet:

1. Indlæsning af kode til det simulerede lager. Dette skyldes dels at ordrekoderne - også for identiske ordrer - er anderledes, og dels at simulatorerne læser kode fra forskellige assemblere. Den ene (8008) lægger hele lageret ud, mens den anden (8080) lægger koden ud i blokke.
2. Tabellerne der permuterer ordrekoden til a. tidsforbrug, b. antal ord og c. indgang i aktionstabellen.
3. Aktionstabellen, hvor der er kommet en masse nye ordrer til, og hvor visse af de gamle er ændret lidt.

Næsten hele konversationen kunne bevares, og det samme gjaldt store dele af logikken i ordreudførslen. Det betød som ventet at indkørslen blev temmelig overkommelig. Til gengæld er simulatoren så ikke særlig elegant

i konversationen og ikke så hurtig i udførslen som den kunne være. Specielt er kommandoerne, som hver er på et bogstav, efterhånden blevet noget kryptiske.

Kørsel med simulatoren

Kommandoer:

A,B,C,D,E,H,L	-inspektion af det pågældende register, d.v.s. at registerets indhold udskrives, hvorpå man kan angive et nyt indhold, eller man kan give et linjeskift hvorved værdien bibeholdes.
F	-finis - simulatoren er klar til ny kode.
I	-instruction count - inspektion af ordretælleren.
K,M,N,Q	-inspektion af carry, zero, parity og sign.
O	-out - afslutter kørslen.
P m n	-print - (m ≤ n) udskriver indholdet af celle m til og med celle n.
R	-reset - alle registre, tidtæller og ordretæller sættes til 0; stakpegepinden sættes til 1000.
S	-stack - de 8 øverste ord i stakken udskrives.
T	-time - inspektion af tidtælleren; tiden måles i antallet af lagerreferencer.
W n	-word - inspektion af indholdet af celle n
X n	-execute - programudførslen starter og fortsættes indtil ordretælleren får værdien n; d.v.s. at udførslen stopper lige før ordren i celle n udføres. Derpå udskrives: ordretæller, A,B,C, D,E,H,L, carry, xcarry, sign, zero og parity.
G	-inspektion af stakpegepinden.

Start på kørslen:
området indeholder den oversatte kode til intel8080, og
sim8080 indeholder algoloversættelsen af simulatoren.

sim8080
code: området 'antal ord'
*

efter * kan man begynde at give sine kommandoer.

Fejludskrifter og deres betydning:

* *	impossible	n < m i print n udenfor lageret i execute
**	address	n eller m udenfor lageret i print
* *	illegal	assembleren har udlagt flere ord en der er opgivet til simulatoren illegal kommando
**	instruction counter	forsøg på at udføre ordrer uden for lageret
* *	stack pointer	stakoverløb

Lagerstørrelsen er ikke parameter men skal ændres i koden. Ændringen skal foretages følgende steder:

```
linje 11:  boolean array core(0:1023);
linje 87:  max_adr := 1023;
```

Den nuværende lagerstørrelse 1023 ændres til det ønskede.

Bemærk: Kommandoen S som udskriver de 8 øverste elementer i stakken giver indexfejl hvis stakpegepinden er større end 1016.

Ordren daa - decimal adjust accumulator er ikke implementeret og virkar bare som nop-ordre.

I appendix B findes en udskrift af algol-teksten til simulatoren samt en udskrift fra en kørsel med simulatoren, hvor alle kommandoer er kaldt.

VII. LITTERATURLISTE

1.+ Brinch Hansen: Operating System Principles
Prentice-Hall 1973

2.+ P. Howalt, E. Lilholt, R. Einersen, B. Tveden-Jørgensen, H. Bjerregaard: Et message-switching-system (DIXI)
Datalogisk Institut Rapport nr. 73/8

3.+ Bo Bagger Laursen: RC 3500 Monitor 1
A/S Regnecentralen Århus August 1974 (RCSL:52-AA187)

4.+ Søren Lauesen: The NBB Semaphore Monitor - User's Manual
A/S Nordisk Brown Boveri Juni 1974 (NBB Doc: EC-D3-4)

5.+ Jørgen Bang og Ken Schubell:
VACS - Et multiprogrammeret tidstro styresystem til VARIAN/620/f
Datalogisk Institut November 1974 (73-3-2)

6. Per Gade Christensen:
Styresystemer til afvikling af online tidstro programmer på minidatamater. Datalogisk Institut Rapport nr. 72/24

7. Knuth: The Art of Computer Programming - Fundamental Algorithms
Addison-Wesley 1972

8.- Conway: Design of a Separable Transition-Diagram Compiler
CACM 6 1963, 396-408

9. Birthwistle, Dahl, Myhrhaug, Nygaard: Simula begin
Studentlitteratur Lund 1974

10. Dijkstra: The Structure of the "THE"-Multiprogramming System
CACM Vol.11/Number 5/Maj 1968

11. Arne Maus: On Acces to Temporary Resources
BIT nr. 15 1975 (72-84)

12.- A Guide to PL/M Programming
Intel Corporation September 1973

13.- Erik Poulsen: Display Terminal Simulator til GT40
Datalogisk Institut Januar 1973 (73-9-18)

14.- Ole Caprani, Lise Lauesen, Fl. Sejergaard Olsen:
NOOS - et datatransmissionssystem fra RC 4000 til RECKU og RECAU
Datalogisk Institut Rapport 74/6

15.- Søren Lauesen: Intern struktur af BOSS 2
Regnecentralen (?) 15.1.71

16.- Per Brinch Hansen:
Concurrent Pascal - A Programming Language for Operating System Design. California Institute of Technologi April 1974

17. Jørgen Bang: GENASS Brugervejledning for programmører
Datalogisk Institut Rapport nr. 74/5

18. Intel8080 Assembly Language Programming Manual
Intel Corporation 1974

19. Eriksen, Helms, Rømer: EDB-ordbogen
Data-serien Gjellerup 1971

20. Hilden, Leunbach, Naur: Forslag til tilføjelser til EDB-ordbogen
Københavns Universitets fond til tilvejebringelse af læremidler 1971

21. Tom Østerby: Forslag til definition af termer inden for operativ-systemer. Institut for datateknik August 1973 (E-IC 306)

\+ betyder at referencen er vigtig
\- betyder at referencen højst har perifer interesse