Sanntidskompendium

Helgesen
Hosen

May 2013

Contents

1 Introduction

Real time systems:

> Any information processing activity or system which has to respond to externally generated input stimuli whitin a finite and specified delay.

- Soft real-time systems: I et soft real time system er det frister på når/hvor fort oppgaver må fullføres innen. Likevel takler systemet at prosesser kan overgå disse tidsfristene, og det er ikke krise om man overgår tidsfrister
- Hard real-time systems: Overtråkk av timeout-tid kan gi store katastrofale følger, og kan også føre til malfunksjonalitet av systemet.
- Reactive systems: Et system hvor prosessene må forholde seg til relative tider. Det kan f.eks være en ventil som må åpnes/lukkes hvert 50ende millisekund.
  - Kan være tidstrigget (periodisk) eller event-trigget(aperiodisk/sporadisk).
- Time-Aware systems: Et system hvor man har absolutte tidspunkter og tidsbegrensninger. Det kan f.eks være et bankhvelv som må være steng mellom 16:00 og 08:00.

2 Reliability and fault tolerance

Fire ting som kan føre til at sanntidssystem feiler

- Utilstrekkelig spesifikasjon eller misforstå sammenheng mellom programmet og omgivelsene.
- Feil i software
- Feil pga hardware feil
- Forstyrrelser i et støttende komunikasjonssystem

2.1 Reliability, failure and faults

- Når oppførsel til system avviker fra spesifikasjonen kalles det en feil(failure).
- Et systems interne tilstand er summen av tilstandene til alle komponentene i systemet. En intern tilstand som ikke er spesifisert kalles en feil(error) og komponenten som forårsaket den ulovlige tilstanden sies å feile (be faulty).
- Man skiller på tre typer feil (faults):
  - Transient faults: Midlertidige feil som oppstår ved bestemte tider. Systemet har denne feilen i en tid, før feile forsvinner. Dette kan f.eks være hardware bugs som følge av elektrisk interferens av "nabosystemer" etc. Slike feil oppstår typisk i kommunikasjonssystemer. (Enda et eksempel kan være mobiltelefonen som skaper støylyd på musikkanlegget)
  - Permanent faults: Starter ved en tid og forblir værende i systemet inntil de er reparert. F.eks brudd på ledning eller software feil (deadlocks osv).
  - Intermittent faults: Tilbakevendende feil som ofte oppstår periodisk. Disse kan sies å være tilbakevendende midlertidige feil (transistent faults). Eksempel er overopphetet CPU - den virker en stund, må stoppes pga varme, startes på nytt, fungerer en stund, varmes opp igjen og må stoppes ...

Det skilles vanligvis mellom to typer bugs i software:

- Bohrbugs - Bugs som kan reproduseres lett og identifiseres gjennom testing.
- Heisenbugs - Bugs som kun oppstår i noen spesifikke situasjoner. Dette kan for eksempel være feil som oppstår når to tråder utfører koden sin likt. Slike bugs er vanskelige å identifisere. Et annet eksempel er minnelekasje som ikke merkes før det er gått lang tid hvis den er liten.

2.2 Failure modes

- Når man benytter et system for å designe et annet, gjør man ofte antagelser om de forskjellige feil-modii: hvis disse antagelsene er feil kan man få uventede feil i det andre systemet.
- Et system utfører tjenester. Hvis det er en feil på en verdi under en slik tjeneste kalles det verdifeil (value failure)
- Time failure: En tjeneste leveres til feil tid: Kan være for tidlig, for sent (performance error) eller aldri (omission failure).
- En tjeneste som ikke forventes og leveres kalles commission eller impromptu failures.
- Et system kan feile på følgende måter
  - Fail uncontrolled: et system som produserer ulike feil både i verdi og til feil tid (dette gjelder også commissions/impromptu errors).
  - Fail late: produserer riktig verdier, men leverer resultatet for sent.
  - Fail silent: Leverer riktig verdier til riktig til, inntil det feiler; eneste mulige feil er omission failure - systemet leverer aldri et resultat det er ment å levere. Dette fører til at etterfølgende prosesser heller ikke vil kunne fullføre.
  - Fail stop: Ganske likt "fail silent", men tillater andre systemer å oppdage at det har gått i feil-modus.
  - Fail controlled: et system som feiler på en spesifisert kontrollert måte.
  - Fail never. Et system som produserer riktig verdier til riktig tid, og dermed aldri feiler.

2.3 Fault preventionn and fault tolerance

To metoder for å forbedre påliteligheten er forebygging(eliminere flest mulige feil i poftware før systemet tas i bruk) og feil toleranse(la systemet fungere selv med feil ved å håndtere disse på en måte).

2.3.1 Forebygging av feil

- Unngå feil: Forebygge ved å velge det beste tilgjengelige hardware komponenter innenfor budsjettrammene. Kan og optimalisere software ved å spesifisere virekmåten, benytte UML, analysere koden med analyseverktøy, benytte kjente kodespråk med hjelpemidler.

- Fjerne feil: Dette er ikke alltid like lett siden de må oppdages. Testing kan kun føre til at feil oppdages, ikke utelukkes. I tillegg kan det være vanskelig å gjennomføre testing i relevante omgivelser.

2.3.2 Fault tolerance

Tre typer feilhåndtering

- Full fault tolerance - Systemet fortsetter å virke selv om feil eksisterer uten dårligere funksjonalitet.
- Graceful degradation(fail soft) - Systemet fortsetter å virke, men mister noe funksjonalitet ved feil.
- Fail safe: Systemet opprettholder sin integritet ved å tillate midlertidige "krasj" i sin operasjon.

2.3.3 Redundancy

- Beskyttende redundans er å legge til komponenter/moduler som kun er der for å håndtere feil. Feilhåndtering ønsker å minimere redundansen, men å maksimere påliteligheten som tilføres systemet.
- Statisk redundans er komponter som skal skjule feil. Eksempler kan være Trippel eller N Modular Redundancy hvor man har tre like komponenter som skal produsere lik output - hvis en komponent gir ut ulik output, blir denne outputten forkastet.
- Dynamisk redundans er redundans i en komponent som indikerer at utgangen har en feil ved seg. Da oppdager vi kun feil og en annen komponent må finne en løsning.

2.4 N-version programming

- Handler om statisk redundans.
- Software-versjonen av N Modular Redundancy. Har forskjellige programmer som kalkulerer samme problem - kan være skrevet i forskjellige språk/benyttet forskjellige kompilatorer osv... Output sammenlignes, og man forkaster fravikende resultater. Man antar dermed at de forskjellige programmene feiler på forskjellige eventer/tider/måter.
- Det er "Driver process" som starter programmene, synker etter kalkulering og sammenligner resultater.

- Når driveren oppdager avvik mellom de forskjellige versjonene bestemmer den hva de forskjellige versjonene skal gjøre. Det kan være å forstette som normalt, avslutte sin versjon eller å fortsette etter en endring i verdi tilpasset de andre versjonene.

2.4.1 Vote Comparison

Mulig problem med N-version programming er hvis f.eks 3 versjoner skal sammenligne to verdier (trykk og temperatur eller lignende). Da kan to av versjonene ligge så vidt under et akseptintervall og siste versjon i intervallet ved temperaturmålingen. Da vil to versjoner korrigere. Hvis så de to programmene som hadde lik temperaturmåling har forskjellig trykkmåling vil alle versjoner følge en forskjellig programsyklus. Dette oppstår når alle er uenige om noe og da vil alle følge forskjellig programutførelse.

2.5 Software dynamic redundancy

Feil toleranse har fire faser:

- 1.Oppdage feil - Deles inn i to klasser
  - Feil som oppdages i omgivelsene når programmet utføres. Inkluderer overflow og andre feil som oppdages av hardware.
  - Feil som oppdages av applikasjonen selv.
- 2. Bestemme hvilken innflytelse feilen har på systemet - Viktig å vite om feilen har forårsaket at feil data har blitt sendt rundt. Henger derfor tett sammen med systemdesign hvor vi har to hovedfremgangsmåter:
  - Modular decomposition - Alt deles inn i moduler hvor kommunikasjon mellom modulene kun foregår gjennom kjent interface. Dette gjør det vanskeligere for en feil å spre seg. Atomic actions - Aktivitet til komponent er atomisk hvis det ikke er noe samhandling mellom aktiviteten og systemet under en action. For systemet er en atomic action usynlig og skjer momentant. Ingen ting i en atomic action kan sendes til systemet og vice versa.
- 3. Transformere systemet til en trygg tilstand, om nødvendig med begrenset funksjonalitet. To fremgangsmåter:
  - Forward recovery - Systemet prøver å fortsette fra en feiltilstand ved å korrigere noe ved tilstanden. Krever at feilen er kjent slik at det ikke blir ren tipping om hva som må endres.

  - Gjenopprette en kjent trygg tilstand systemet har vært i før feilen oppstod. Håper da at ikke tilsvarende feil oppstår på nytt(alternativ del av programmet utføres). Recovery point er punktet som systemet gjenopprettes til. Trenger checkpoints underveis for å vite om trygge tilstander og mulige recovery points. Med denne metoden er det ikke nødvendig å vite hva som forårsaket feilen. Ulempen er at er at ikke noe av det som er utført i hardware pga feilen kan korrigeres for. F.eks at et missil er skutt ut. I tillegg tar det tid og kan bryte med krav om oppetid f.eks i real time systems. Et annet problem kan være at recovery i en prossess krever recovery i en annen prosess hvis disse har utvekslet informasjon etter feil oppstod. Dette kalles Domino-effekten og er en annen ulempe med "backward recovery". Det kan føre til at begge prosessene i praksis avbrytes og startes på nytt. Derfor må man ved mange samarbeidende prosesser ha **recovery lines** så en feil i en prosess ikke gjør at alle andre prosesser den samarbeider med må hoppe langt tilbake.

- 4. Prøve å finne ut hva som forårsaket feilen for å forebygge fremtidige feil. Veldig vanskelig siden det kan kreve at designet endres. Noen antar derfor at feil er transiente og kun skjer en gang, mens andre antar at feilhåndteringen er god nok til at feilene ikke er noe problem. To deler her og
  - Lokalisering av feil. Ved feil i hardware kan komponenten erstattes. Ved feil i software må ny versjon av koden evt. lages
  - Reperasjon - Fikse feilen.

2.6 The recovery block approach to software fault tolerance

Recovery blocks som ved inngangen har et recovery point og ved utgangen har en akseptansetest. Hvis punktet ikke aksepteres går vi tilbake til recovery point og utfører en alternativ strategi før en ny akseptansetest. Hvis alle mulige strategier feiler, må systemet gjenopprettes på et høyere nivå enn aktuelt nivå. Akseptansetesten tester om feilen fortsatt er i systemet. Recovery blocks er vanligvsis ikke støttet direkte i kodespråk, men kan lages av programmereren.

2.7 A comparison between N-version programming and recovery blocks

- N-vesion er basert på statisk redundans. Recovery blocks er dynamiske siden alternativ strategi kun utføres når feil er oppdaget.
- Begge metoder fører til økte implementeringskostnader. N version krever Driver og recovery blocks krever akseptansetester. Ved kjøring krever N-version N ganger så mange ressureser som en enkelt versjon. For recovery blocks er det kun en versjon, men dyrt å lage recover lines.
- Begge gir mulgihet til å løse feil som ikke predikeres.
- N version bruker stemming, mens recovery block benytter akseptansetest.
- N version antar at alle versjoner er atomic - ingen versjoner kommuniserer, bare med Driver. Backward recovery kritiseres fordi det ikke gjør opp for eventuelle feil som har spredd seg til omgivelsene. Dette er ikke problem for N-version siden de ikke kommuniserer med hverandre.

2.8 Dynamic redundancy and exceptions

En "exception" er når en feil(error) oppstår. Å gi en exception condition til den som er ansvarlig for operasjonen hvor feilen oppstod kalles "raising(signalling eller throwing) the exception", og responsen til den ansvarlige kalles "handling(eller catching) the exception". Exception handling kan ses på som en forward error recovery. Feilhåndteringsmekanismer (her exception-handling) ble ikke lagt til i programmeringsspråk for å tilfredstille design feil, men det kan gjøres(kap 3.4). Exceptions og exception-handling kan brukes til blant annet:

- Takle unormale situasjoner som oppstår i omgivelsene
- Gjøre at design feil tolereres.
- Gi støtte for en generell måte å oppdage feil og gjennopprettelsemuligheter.

2.9 Measuring and predicting the reliability of software

2.10 Safety, reliability and dependability

Software sikkerhet er definert ut i fra **mishaps** som er en hendelse/serie av hendelser som ikke er planlagt som kan føre til død,

skade eller skade på omgivelsene. Pålitelighet er definert som sannsynlighet for suksess, men sikkerhet er ssh for at avhengigheter som kan føre til *mishaps* ikke oppstår uavhengig av om den tiltenkte operasjonen utføres eller ikke.

Pålitelighet (dependability) kan beskrives av tre komponenter

- Threats (trusler): omstendigheter som fører til ikke-pålitelighet
- Means (midler): metodene, verktøy og løsninger nødvendig for å levere pålitelig service.
- Attributes (attributter): Måten og målene på hvordan kvaliteten på en pålitelig service kan bli vurdert.

3 Exceptions and exception handling

Det er en rekke generelle krav for exception-handling rammeverk:

- Enkelt å forstå og bruke
- Koden for exception handling bør ikke gjøre at den vanlige koden blir vanskeligere å forstå. Det kan gi mindre pålitelig system.
- Mekanismen bør designes slik at det kun utføres ekstra arbeid når en exception håndteres.
- Bør takle exceptions uniformt uavhengig av om de er oppdaget i hardware eller pga feil i software (overflow og assertion-feil bør gi en lik responsstrategi).
- Bør gi støtte for gjenopprettelses handlinger.

3.1 Exception handling in older real-time languages

3.1.1 Unusual return values and status flags

Primitiv form hvor en funksjon returnerer en spesiell verdi ved feil. Hvis en funksjon ikke kan designes med en slik returverdi kan flags benyttes. Dette er statussignal som deles og kan settes og testes. I c har vi errno som hver applikasjon kan gi en kode som indikerer feil. Variabelen husker siste feil. Denne metoden overholder ikke alle kravene.

3.1.2 Forced branch

I assembly språk er typisk mekanisme for å exception handling å la subrutiner droppe return statement. Da gjøres ikke instruksjonen som følger rett etter subrutinen og dette indikerer feil.

3.1.3 Non-local goto and error procedures

Dette kan være å benytte seg av goto-funksjonen i C. Dette fører til et hopp i programflyten. Et slikt hopp indikerer også at en feil har skjedd. Likevel er ikke goto god kodeskikk siden en goto statement kan forverre programflyten for leseren. Non-local goto's ødelegger programflyten. Dvs man hopper fra en funksjon til en annen. Dette går ikke med vanlige goto-statements siden labels har restriksjoner på synlighet innenfor scope, og man benytter derfor setjmp() og longjmp() i C.

3.2 Modern exception handling

Gamle metoder førte til at den normale koden ble endret for å teste feil. I dag prøver man å integrere exception handling direkte i språket.

3.2.1 Exceptions and their representation

Exception kan løses synkront eller asynkront. Ved synkron løsning vil feil utløse exception umiddelbart ved at en prøver på noe ulovlig på en linje. Ved asynkron exception utløser exception en stund etter at feilen oppstod. Enten i prosessen som opprinnelig gjorde den ulovlige operasjonen eller i en annen prosess. Deler inn i fire klasser exceptions:

- Oppdaget av omgivelsenene og utløst synkront. (dele på 0 f.eks)
- Oppdaget av applikasjon og utløst synkront (feil på assertion sjekk)
- Oppdaget av omgivelsene og utløst asynkront. (feil på grunn av "power failure")
- Oppdaget av applikasjon og utløst asynkront. (En prosess oppdager feil som vil føre til at annen prosess ikke kan klare å overholde en tidsfrist)

Ada krever at exceptions deklarerer som konstanter. I Java er exceptions objekter av en subklasse i klassen Throwable. I c++ kan exceptions av alle objekttyper kastes uten å predeklareres.

3.2.2 The domain of an exception handler

Kan være flere "handlers" for en exception i et program. Til hver "håndterer" er et domene som spesifiserer hvor i koden den kan utløses. Nøyaktigheten på domenet bestemmer hvor lett det er å lokalisere en exception.

Ada-syntax for exeptions:
begin:
— *Gjør noe*
exeption:
— *Håndter feil*
end;

3.2.3 Exception Propagation

Hva i huleste gjør man hvis man får et exception, men IKKE har en handler for exception'et?!?! To mulig løsningsstrategier:

- Se på fraværet av en exceptionhandler som en programmeringsfeil som burde vært raportert/fikset ved kompilering. Løsning = ??? Ingen anelse - gjerne prøv å les dette avsnittet i boka Håkon :-) Det kan særlig være et problem der exceptions ikke løses i de rutinene der de faktisk utløses. Dette kan være funksjoner inne i funksjoner som løser exceptions f.eks. Skal kompilator kunne merke dette må det spesifiseres hvilke exceptions som løses i hvilke subrutiner. Dette gjøres f.eks i CHIDL (CHILL). Java og C++ har også støtte for slikt oppsett, men krever ikke handler til hver exception som CHILL.
- Se etter exceptionhandlers lenger oppe i callstacken. Dette kalles også propagating (forplantning). Ada, Java, C++ tillater dette. En løsning er å bruke catch-all-blokker hvis exception'et handler utenfor scopet til handleren.

Uhåndterte exceptions fører til at sekvensielle programmer terminerer.

Handlers kan være både statisk og dynamisk assosiert. Statiske assosiasjoner gjøres ved kompilering, og man kan dermed ikke ha propagation siden kjeden av invokers (kallende funksjoner) ikke er kjent. Dynamisk assosiering gjøres i kjøretid og tillater derfor propagation. Dynamisk assosiering er mer fleksibel, men krever mer kjøretidsoverhead siden handleren må søkes opp. Ved statisk assosiasjon settes exceptionhandleren addresse ved kompilering.

3.2.4 Resumption versus termination model

Det er tre mulige måter å håndtere et exception på etter at handleren er ferdig:

- The resumption or notify model: prosessen hvor exceptionet ble utløst fortsetter (resume) etter at exceptionet har blitt håndtert, som om ingenting har skjedd.
- Termination or escape model: Kontrollen returneres ikke tilbake til prosessen etter at handleren har tatt hånd om exceptionet.
- The Hybrid modell: Handleren bestemmer hvorvidt prosessen skal få fortsette eller terminere.

3.3 Exception handling in Ada, Java and C

3.3.1 Ada

Tilater terminering samt propagation av uhåndterte exceptions. Les mer på dette, og lag lettfattelige kodeeksempler.

3.3.2 Exception declaration

Du oppretter en exception med navn "navn" koden:

```
navn : exception;
```

Alle exceptions som opprettes har en ID som kan hentes ut og benyttes for å identifisere en bestemt exception. Ada har to definerte exceptions som kan utløses av programmet selv:

- Constraint_Error - Utløses ved deling på 0, oppslag i tabell utenfor grensene f.eks.
- Storage_Error - Dynamisk minne oppbrukt så nytt minne ikke kan allokeres.

3.3.3 Raising an exception

Exceptions utløses(rais) i Ada med nøkkelordet *raise*. Exceptioneet *Io_Error* utløses på følgende måte:

```
if Io_Device_In_Error then
    raise Io_Error;
end if
```

Når man utløses er exception slik kalles det en exception occurence. Verdien *Ada.Exceptions.Exception_Occurence* får etter at handleren er ferdig informasjon om hva som forårsaket exceptionet. Hvis exceptionet hadde blitt deklarert med ID direkte måtte kallet *Ada.Exceptions.Raise_Exception* utløst exceptionet. Stort eksempel av *Exceptions* pakken i Ada finnes på side 71.

3.3.4 Exception handling

Exception håndteres i kodesnutter med keywordet when. Følgende kodesnutt viser opprettelse av tre exceptions, og hvordan de håndteres;

```
declare
    excpA, excpB, excpC : exception;
begin
    noe utløser exceptions
exception
    when E: excpA | expB =>
        -- Korriger noe
        -- Hvis excpA eller excpB er utløst
        -- ligger grunnen (occurence) i variabelen E.
    when excpC =>
        .... Her lagres ikke occurence i noen variabel.
    when C: others=>
        --Eventuelle andre exceptions som ikke håndteres av de
        --andre whensetningenen kan håndteres felles her.
end
```

3.3.5 Exception propagation

Hvis det ikke er en handler til en exception i rutinene du er i (eller subrutine eller akseptstatement) utløses exceptionet på nytt. Dette er propagering. Hvis man ikke finner en handler søker man i rutinene som kaller rutinen som utløser exceptionet. Se omfattende eksempel s. 73-74.

3.3.6 Last Wishes

En exception kan utløses på nytt inne i en handler ved å benytte nøkkelordet raise. Da utløses siste exception på nytt. Dette kan for eksempel gjøres i prosedyrer der du vil rette opp noe etter at exceptionet er løst. Litt vanskelig delkapittel, ta en titt på side 74-75 om det virker eksamensrelevant.

3.3.7 Suppressing exceptions

I noen tilfeller kan ADA fremgangsmåten bryte med krav 3. Dette kravet sier at ikke ekstra kode skal gjennomføres så lenge ingen exceptions er utløst. Dette kan skje med standardexceptionene som f.eks. *Constraint_Error*. Denne utløser f.eks en exception når en NULL-pointer prøver og aksesseres. Hvis det ikke sjekkes på noen måte om en peker man aksesserer er NULL-pekeren vil kompilatoren selv generere kode og sjekke for dette. Derfor vil denne koden kjøres selv om et exception ikke er utløst. Med mange pekere kan dette

gjøre programmet tregere. Ada kan alikevel rette for dette med suppress pragmaet. Da droppes noen av run-time testene hvis de virker for kostbare for en spesiell applikasjon.

3.3.8 Difficulties with the Ada model of exceptions

- Exceptions and packages - Exceptions som kan utløses med bruk av en pakke er deklarert i pakken sammen med subrutiner som kan kalles. Det er ikke åpenbart hvilke subrutiner som kan utløse hvilke exceptions. Dette kan føre til at det er vanskelig å håndtere alle exceptions riktig om man ikke har skrevet pakken selv. Bør oppgi dette som kommentar.
- Parameter passing - Ada tillater ikke at mange parametere sendes til handlers. Kun en tekstreng. Derfor vanskelig å rette opp objekter f.eks.
- Scope and propagation - Noen exceptions kan propageres ut av scopet hvor de er deklarert. Kan da kun fanges av when others statements.

3.3.9 Java

Integrert exceptionhandling i den objektorienterte modellen.

3.3.10 Exception Declaration

I Java er alle exceptions subklasser av Throwable klassen. Se figur 3.3 i boka på s.78. Exception er en underklasse i Throwable. Exception defineres videre som alle underklasser i throwable (ikke i Exceptions). I underklassen Error ligger interne feil som fullt minne etc. Lite programmet kan gjøres med disse når de utløses. Objekter i exceptions klassen er feil programmet selv kan håndtere og potensiellt "throw"/kaste selv. I tillegg til egendefinerte exceptions ligger klassen RunTimeException (dele på 0 etc) som en underklasse av Exceptions. "Kastbare" objekter fra Exceptionsklassen kalles **unchecked** exceptions.

I en klassse defineres det i deklarasjon hvilke exceptions som kan kastes på føglende måte:

```
import exceptionLibrary.IntegerConstraintError;
public class Temperature {
    private int T;
    public Temperature(int initialValue) throws IntegerConstraintError {
    // Constructor, kan kaste exceptions av typen IntegerCo...
    }
}
```

Ser da direkte hvilke type exceptions en klasse kan kaste i stedet for å kommentere som i Ada. Kan føre opp flere exceptions ved å liste opp flere etter throws. Hvis et exception som ikke er nevnt i throws kastes får vi kompileringsfeil.

3.3.11 Throwing an exception

Å utløse en exception kalles å kaste den (throwing) i Java. Kodeeksempel:

```
if(value > 100 || value < 0){
    throw new IntegerConstraintError(\emph{argumenter til konstruktør});
}
```

3.3.12 Exception handling

Excpetion håndteres inne i en try-blokk. Hver handler er markert med en catch-blokk. Kodeeksempel (har klasse Temperaturecontroller, og exceptions IntegerCOntraintError og ActuatorDead):

```
try{
    TemperatureController TC = new Tempeaturecontroller(20);
    // Manipuler temperatur på en måte, ka
}
catch (IntegerConstraintError error){
    // Print ut beskjeden som nå ligger i objekt error
    System.out.println(error.getMessage());
    // getMessage metode i exception klassen
}
catch (ActuatorDead error{
    // Gjør det samme
}
```

En handler av type T catcher et exception av type E om:

- T og E er av samme klasse
- T er parent til E ved kastingspunktet

Derfor kan man fange alle egendefinerte exceptions med kommando:

```
catch(Exception E) {
    // Alle egendefinerte exceptions er jo subklasse av Exceptions.
    // Blir likt som others statement i Ada
}
```

3.3.13 Exception propagation and last wishes

Hvis exceptions ikke kan hådnteres der de utløses termineres den kallende rutinen og handler søkes etter i der rutinen er kalt. Derfor støttes propagering som i Ada.

Etter cactch statementene kan man benytte en finally blokk(helt lik syntaks som catch, men catch byttes ut med finally og ingen argumenter). Det som står inne her utføres uavhengig av om en exception utløses i en try blokk eller ikke. Sikrer derfor at det inne i blokka utføres uansett. Det samme kan gjøres i Ada med when others og kontrollvariabler. Fullt eksempel på Java er i boka på side 83.

3.3.14 C

Ikke noe innebygd exception-løsning, men man kan bruke setjmp() og longjmp() sammen med Macroer (defines) til å lage exception-funksjonalitet. Setjmp() lagrer programstatus og returnerer 0. Longjmp() gjenoppretter status og sørger for at programmet restarter fra setjmp() posisjonen. Andre rutiner avbrytes da med setjmp() kall. Setjmp returnerer da verdi levert av longjmp.

3.4 Recovery blocks and exceptions

Ved feil, som feks exceptions, kan man håndtere dette i recovery blocks som kjent fra fault-tolerance-kapitlet. Kan da håndtere for uforutsette feil, noe som ikke er mulig i forward-recovery.

4 Concurrent programming

Programmering med potensiell paralellisme, og løsninger på de synkroniserings- og kommuniseringesproblemer det byr på.

Hvorfor concurrent programming?

- Verden vi lever i er parallell - dvs ting skjer i parallell. For å modellere f.eks roboter er vi avhenging av parallellprogramering.
- For å utnytte prosessoren til det fulle kan vi ikke tillate oss å la prosessor stå å vente på input fra f.eks IO. Derfor kan vi ha andre prosesser som kjører mens en prosess venter.
- Ved å benytte flere prosessorer til å løse et problem kan vi øke ytelsen til systemet betraktelig. Dette krever parallell utførbare oppgaver.

Ahmdals lov: Gitt P andelen av en algoritme som kan nyte godt av å kjøre i parallell, og N antall prosessorer, er maksimal speedup som kan oppnås:

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{1}$$

4.1 Processes and tasks/threads

Pascal, C, Fortran og COBOL er sekvensielle programmeringsspråk (en hovedtråd som styrer programflyten). Et Concurrent program kan sees på som flere autonome sekvensielle programmer som utføres i parallell - dvs hver prosess har kontroll over seg selv.

Når man snakker om concurrency (samtidighet/parallellitet), skiler man mellom to ting:

- Concurrency mellom programmer (dvs. prosesser). En prosess er en eller flere tråder som opererer innenfor prosessens egne delte minneområde (prosessen har et eget minneområde som deles mellom trådene i prosessen).
- Concurrency i programmer (tasks/tråder). Tråder deler minne.

Skiller ofte mellom tråder som er synlige for operativsystemet(kernel-level tråder) og de som er usynlige(library-level tråder). Library-level trådene er tråder som startes i bibliotekfunksjoner.

RTSS (Run-Time Support System) = Støtte i operativsystemet for å kjøre flertrådede (concurrent) programmer. RTSS kan implementeres på forskjellige måter, og kan sees på som en kobling mellom software og hardware.

- Kan være et softwareprogram som en del av applikasjonen
- Et standard softwaresystem generert fra programkoden av kompilatoren
- En harwarestruktur kodet inn på prosessoren.

Tråder kan utføres/implementeres forskjellig. I hovedsak skiller vi på 3 måter:

- Multiplekset utførelse på en enkelt prosessor (Dvs alle tråder kjører på samme prosessor. Man trenger da en Scheduler)
- Multiplekset utførelse på et multiprosessorsystem hvor alle prosessorer har tilgang til det samme delte minneområdet.
- Multiplekset utførelse på et multiprosessorsystem hvor prosessorene IKKE har tilgang til det samme delte minneområdet (distribuert system).
- Man kan også ha hybrider av disse systemene. Feks systemer med noe delt og noe separat minne.

Nøkkelordet **executable** indikerer at en task/tråd kan utføres når en prosessor er ledig. RTSS eller Run-Time Kernel deler ut tasks til forskjellige prosessorer. Den kan gjøre dette på tre måter:

- I C og C++ er det støtte for å legge opp fordelingen selv. Da legges det inn i applikasjonen hvordan det skal gjøres.
- I Java og Ada er det vanligvis et standard software system som fordeler og lages under kompilering
- Hardware struktur som "mikrokodes" direkte inn i prosessoren for effektivitet.

Når man snakker om kjørende tråder skiller man på tre operasjonsmodi:

- Uavhengig: Trådene hverken kommuniserer eller synkroniseres sammen.
- Samarbeidende: Kommuniserer og synkroniserer mot hverandre for å nå et felles mål. Trådene samarbeider for å løse en oppgave.
- Konkurrerende: Uavhengige tråder som likevel snakker med hverandre for å allokere delte resursser ol. Dvs. man kan ha forskjellige tråder med delte resursser (f.eks. minne), som alle jobber uavhengig mot sine egne mål, men disse vil konkurrere om de samme resurssene og må derfor kommunisere for å sørge for at resurssene ikke blir brukt av flere samtidig.

4.2 Concurrent execution

Strukturen til en tråd klassifiseres på følgende måte:

- Statisk: antall tråder er fixed, og bestemmes ved kompilering
- Dynamisk: tråder kan opprettes når som helst.

Nivå (scope) av parallellitet skilles ved:

- Nested: Tråder kan defineres i hvilket som helst nivå i programmet. Sagt på en annen måte: tråder kan defineres i andre tråder (eksempler på språk med nested nivå: C, Java, Ada, Occam2).
- Flat: Tråder kan kun defineres i det ytterste nivået av programmet.

For tråder med "nested" parallellitet kan man skille på to typer parallellitet:

- Coarse grain (grovkornet): Program som har relativt få tråder, men hvor hver tråd har en signifikant leveperiode. Programmer i Ada vanligvis designet slik.
- Fine grain (finkornet): Et stort antall tråder, hvor trådene gjerne er mindre enn i Coarse grain og lever i kortere perioder. Dette er typisk tråder som skapes for å løse et bestemt problem 1 gang.

Terminering av tråder kan skje på forskjellige måter, bla.: tråden er ferdigkjørt, tråden selvterminerer, blir avsluttet av en annen tråd osv.

Guardian/Dependant: en tråd er ansvarlig, mens en annen er avhengig. En ansvarlig tråd kan ikke terminere før alle tråder som er avhengig av den er terminert. Dette fører til at man er sikker på at alle tråder er terminert før man avslutter et program. En guardian trenger ikke alltid være den tråden som opprettet childen (dependanten) - childen kan være opprettet av en parent som ikke nødvendigvis må være den samme som guardianen (den ansvarlige). Dvs. guardian kan være, men trenger ikke være, den samme som parent av en child/dependant.

4.2.1 Tasks and objects

Deler inn i to typer objekter i en tråd:

- Active - De påtar seg spontane handlinger og muliggjør at tråden kan fortsette. Inneholder selv en eksplisitt eller implisitt tråd).

- Reactive - Gjør kun en handling når de får beskjed av et aktivt objekt.

Et passivt reaktivt objekt tillater aktive objekter til å benytte de uansett(krever ekstern tråd som styrer når metodene skal utføres). Noen reaktive objekter kan hindre aktive objekter å ta de i bruk hvis de for eksempel allerede er brukt av noen andre. Tilgjengelige ressurser er ofte reaktive og vi trenger da en for for kontroll, en control agent. Hvis denne agenten selv er passiv (semafor f.eks) så sier vi at den er **protected** (protected resource) eller **sunchronized**. Et protected objekt deles gjerne av flere tråder og trenger ekstern tråd som styrer når den utfører sine metoder (har synkroniseringsbegrensinger). Det kalles **server** hvis agenten er aktiv. Et slikt objekt deles og av flere tråder og har synkroniseringsbegrensinger.

4.3 Task representation

Det finnes forskjellige måter å representere tråder på. Videre følger tre vanlige representasjonsmåter:

4.3.1 Fork and Join

Fork: start tråd. Join: synkroniser etter sluttføring av tråd. Eksempel:

```
procedure P is
begin
    ...
    C := fork F; -- dette starter funksjon F i parallell tråd C.
    .   -- Prosedyre P forsetter sin utførelse
    .
    .
    J := join C -- Tråd C og prosedyre P synkroniserer før P kan fortsette.
    ...
end P;
```

4.3.2 Cobegin

Gir mulighet til å starte en haug med tråder som kan synkroniseres ved slutt. Eksempel:

```
cobegin
    S1;
    S2;
    S3;
```

```
    .
    .
    .
    Sn
coend
```

4.3.3 Explicit task declaration

Rutinene bestemmer selv om de skal utføres i parallell eller ikke (Hva nå enn det betyr).

4.4 Concurrent execution in Ada

Eksempel:

```
procedure Example1 is
    task A; -- deklarerer task objectene/variablene
    task B;

    task body A is
    -- lokal deklarering av task A
    begin
    -- sekvens av statements: det som faktisk skal utføres i A
    end A;

    task body B is
    -- lokal deklarering av task AB
    begin
    -- sekvens av statements: det som faktisk skal utføres i B
    end B;
begin
    -- task A og B startes før den første statementen i prosedyren utføres
    .
    .
    .
end Example1;
```

Ada-knep: for å definere flere tasks, kan man definere Task til en type ved følgende:

```
 task type T; -- ganske likt typedef
 A, B: T; -- oppretter A og B av type task. Dette er det samme
          -- som deklareringen i Eksempel1
```

4.4.1 Task identification

Alle trådtyper (task types) i Ada er private. Hvis man har to variabler av samme type er det ikke lov å skrive

```
taskA.all := taskB.all
```

Vi kan altså ikke kopiere over all info i taskB til en ny taskA. Det er dog lov å skrive

```
taskA := taskB
```

Da peker taskA til samme som taskB. Det vil si at hvis taskA inneholdt en task vil aksesspunktet til denne være borte om ikke en annen variabel og inneholder den. Dette betyr at tasken er blitt **anonym**. I noen situasjoner kan det være bedre at en task(fra nå tråd) har en unik ID enn et variabelnavn. Dette kan for eksempel være når en server tråd må vite at tråden den kommuniserer med er samme tråd som den har kommunisert med før for å jobbe videre. Da er det lettere å jobbe med en unik ID som aldri slettes. Dette er det støtte for i pakken *Ada.Task_Identification*. MEN, nå en slik ID benyttes må man være forsiktig fordi den tilhørende tråden ikke nødvendigvis er aktiv eller i scopet lengre (kan være død).

4.4.2 Task termination

I Ada terminerer en tråd/task hvis den er ferdigkjørt, det blir kjørt en terminerkommando etter en select-statement (likt if-else) eller den blir aborted (avsluttet fra utsiden)

4.4.3 Task abortion

Enhver tråd kan avslutte en annen tråd hvis den har tilgang til task variablen i sitt scope. Når en task blir aborted, vil også alle dens dependants bli aborted.

4.4.4 OOP and concurrency

4.5 Concurrent execution in Java

To muligheter: Kan benytte seg av en klasse som heter Thread (minner om pthread i C), eller en klasse som heter Runnable.

```
// Runnable klassen
public interface Runnable{
    public void run();
}
```

Runnable gjør at ikke alle tråder må være barn av Thread-klassen. Alle klasser som vil utføres parallelt må da selv legge til runnable i klassen og benytte runnable sin metode for å starte tråd. Stort eksempel s.112. To måter å lage tråder på:

- Deklarere klassen som subklasse av thread og overlagre run-metoden (se side 113). Da kan instans av subklassen allokeres og startes. I Java lar man en klasse være en subklasse av Threads ved å skrive

  ```
  public class klasseA extends Thread {...}
  ```

- Deklarere klasse som implementerer Runnable interface. Må så lage passende run-metode(overlagre?).

  ```
  public class klasseB extends Runnable {..}
  ```

I begge tilfeller må man deretter opprette to objekter av typen klasseA og klasseB. Deretter opprette et objekt av typen Thread og gi argument til konstruktør(objekt av type klasseA eller klasseB). Til slutt må man kalle start metoden som ligger i thread klassen og starter selve tråden:

```
// Har de nevnte klassene
testA = new klasseA(..argument til konstruktør..);
testB = new klasseB(..argument til konstruktør..);
Thread A = new Thread(testA);
Thread B = new Thread(testB);
A.start();
B.start();
```

LES MER PÅ DETTE... evt. lær Java..?

4.5.1 Thread identification

To måter tråder kan identifiseres på:

- Hvis koden til tråden er subklasse av Threads kan man definere ID ved å skrive "Thread threadID;". Enhver subklasse av Thread kan bruke en slik variabel.

- Hvis man bruker Runnable må man selv lage en variabelID med "Runnable threadID;". Merk at når denne variablelen er opprettet får du ikke gjort noe med den. Alle trådoperasjoner ligger i Thread så da er det bedre å benytte Thread klassen om man er avhengig av tråd ID.

4.5.2 Thread termination

En Java-tråd terminerer når den er ferdigkjørt eller hvis den får et uhåndtert exception. Trådene i Java kan være av to typer: **user** eller **daemon** threads. Daemon tråder er tråder som yter generelle tjenester og som ofte aldri vil terminere. Etter at alle user-tråder har terminert, vil daemon-trådene terminere før main-programmet terminerer.

4.5.3 Thread-related exceptions

Det kan kastes forskjellige exceptions fra tråder i Java. Et eksempel er *IllegalThreadStateException* som kastes når:

- man prøver å starte en tråd som allerede er startet.
- man kaller setDaemon på en tråd som allerede er startet.

InterruptedException og flere andre unntak (exceptions) kan også kastes, men er vel strengt tatt ikke puggemateriale.

4.5.4 Real-time threads

Java mangler en del støtte for sanntid, og derfor lager man nye trådklasser for java. Disse skal vi ikke gå inn på.

4.6 Concurrent excecution in C/Real-Time POSIX

POSIX har tre metoder for å opprette parallelle aktiviteter. Den første er *fork* som vi husker fra kapitlet om trådrepresentering (kap 4.3.1). Fork oppretter en kopi av hele prosessen som skal kjøres. Den andre er *spawn* hvor funksjon laster og utfører en subrutine.

POSIX tillater hver prosess å kjøre flere tråder. Disse trådene har tilgang til samme minneområde og kjøres på egne adresseområder og har egen stack. *pthread_create* benyttes for å sette igang en slik tråd i C/POSIX, og man kan benytte *pthread_setconcurrency* for å bestemme graden av parallellitet (en detalj man bare trenger å vite eksisterer). **Guard:** et overflowbuffer som hver tråd har. Guarden er en av flere attributter som er individuell for alle tråder. Andre

attributter er stack-størrelse etc. Terminering skjer når funksjonen tråden kjører er ferdig eller ved bruk av *pthread_exit, pthread_cancel, pthread_join*. Ved terminering bør man rydde opp etter tråden: dette kalles såkalt detaching (norsk: demontering/frakobling). Denne detachingen gjøres ved pthread_join eller pthread_detach

4.7 Multiprocessor and distributed systems

Når man har flere prosessorer antar man at alle tråder kan kjøres på hvilken som helst prosessor. Dette kalles global dispatching (global sending), og vil si at en tråd kan migrere fra en prosessor til en annen iløpet av kjøretiden til programmet. Om man setter restriksjoner på hvilke prosessorer forskjellige tråder kan kjøres, oppnår man økt forutsigbarhet og dette kalles ofte **processor affinity** (prosessortilhørighet).

- Partisjonering: å dele et system i egnete deler.
- Konfigurering: Partisjonerte deler av et program assosieres med bestemte prosessorer i systemet.
- Allokering: Gjør om det konfigurerte systemet til en samling av eksekverbare moduler og laster disse ned på prosessorene.
- Transparent excecution (gjennomsiktig utførelse): Utførelse av distribuert software slik at eksterne ressureser kan aksesseres på en måte uavhengig av lokasjon - Vanligvis ved at beskjeder sendes mellom ressurs og tråd?
- Rekonfigurering: dynamisk endring i lokasjon av en softwarekomponent eller ressurs.

4.7.1 Ada

Ada gir muligheten til multiprosessor-systemer, men gir ikke programmereren mulighet til å velge partisjonering av sitt program til systemets prosessorer (ikke noe standardstøtte for dette). Ada definerer distribuert system til å være kobling mellom en eller flere prosessnoder(som både kan utføre beregninger og lagre noe) sammen med null eller flere lagringsnoder(kan kun lagre ting her, men mer en en prosessnode kan aksessere den). Partisjoner i Ada kan være aktive og passive. Bibliotekdeler bestående av aktive partiosjoner er lokalisert og utføres på samme prosessor. Bibliotekenheter med passive partisjoner er lokalisert på minneelement som kan aksesseres direkte for akitve partisjoner som trenger dem. På denne måten kan ikke variable i en aktiv partisjon aksesseres fra en annen aktiv partisjon. Må da gå via passiv partisjon de to aktive f.eks. kan dele.

4.7.2 Java

Java gir muligheten til multiprosessor-systemer, men gir ikke programmereren mulighet til å velge partisjonering av sitt program til systemets prosessorer. Kan dog bestemme max antall prosessorer som skal tas i bruk av et program. Det finnes to måter å lage distribuerte systemer i Java: 1. Kjøre Java-programmet på separate maskiner og benytte Java-netverksmoduler for å snakke sammen. 2. Benytte remote-objekter[1].

4.7.3 C/Real-Time POSIX

Kan helle ikke her bestemme hvilke partisjoner som kjører på hvilke prosessorer.

4.7.4 Processor affinity

Hverken Ada, Java eller C gir programmeren kontroll over hvilken prosessor som skal ta hvilken task i et multiprosessor system med delt minne.

4.8 A simple embedded system

4.9 Language-supported versus operating-system-supported concurrency

Concurrency i programmeringsspråket gir følgende positive sider: mer oversiktlig kode, gir mer portabel kode mellom OS, en embedded data har kanskje ikke OS tilgjengelig. Negative sider er: Det kan være vanskelig å implementere god parallellitet på toppen av OS'ets modell.

[1] remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network)

5 Shared variable-based synchronization and communication

Kommunisering mellom tasks/tråder kan skje på to måter: delte variable eller ved meldingsbasert. Delte variable er variable som mer enn en tråd har tilgang til.

5.1 Mutual exclusion and condition synchronization

Problem: aksessering av samme variabel til samme tid. Løsning: Mutual exclusion (Mutex). En mutual exlusion hindrer to tasks å aksessere samme område (kritisk område: critical section) til samme tid. Et kritisk område er den delen av koden en tråd må utføre før den delte variabelen benyttes av en annen tråd.

Condition synchronization: benyttes hvis en tasks operasjon er avhengig av at en annen task er i en definert tilstand.

Man kan benytte buffer for å kommunisere via. Å bruke et buffer for å koble to tråder sammen er vanlig og kalles et **producer-consumer** system. Skal man benytte buffer, må man ha to condition-variables: et som angir hvorvidt bufferet er fullt(gitt at man ønsker legge noe inn på bufferet), og et som angir om bufferet er tomt (man ønsker hente noe fra bufferet). Man må også ha mutual exclusion slik at f.eks. neste-åpne-plass-index ikke blir korrupt.

5.2 Busy waiting

Dette er en metode for synkronisering. Den fungerer best for condition synkronisering. Lag en løkke som står å venter på at et flagg skal settes til en verdi (wait). En annen funksjon setter denne verdien (signal). Busy-waiting er generelt ineffektivt, da dette bruker prosessorkraft på en sjekk som ikke gjør noe nyttig *arbeid*. Et problem med dette er at det kan føre til **Livelock** som vil si at tråder er låst i en loop uten at den får fortsatt. Fra et tilstandsperspektiv kan dette være en sykel av tilstander det er umulig å komme ut av. Forskjellen fra Deadlock er at programflyten ikke har stoppet opp. De største ulempene med metoden er:

- Vanskelig å designe protokoller som benytter busy-loop og vanskelig å bevise at de er riktige.
- Vanskelig å få testet alle situasjoner så man kan ikke vite at det fungerer.
- Ineffektivt

- En tråd ikke til å stole på (hvorfor ha en slik slem tråd?) kan fysisk og psykisk misbruke en variabel slik at hele systemet blir korrupt. (tulle med flaggsetting?)

5.3 Suspend and resume

I stedet for å polle på flagget i en løkke, kan man suspendere tråden som venter på at flagget skal settes. Eksempel i Ada:

```
task P1;
    ...
    if flag = down do
        suspend; -- suspenderer P1
    end;
    flag := down;
    ...
end P1;

task P2;
    ...
    flag := up;
    resume p1; -- gjenstarter P1 hvis den er suspended
    ...
end P2;
```

Man har også lignende muligheter i Java i Thread-klassen, men her får man race-condition-problemet[2]. Dette kan dog løses ved at man annonserer at man tenker å suspende før man gjør det (eller noe? se side 143 nederst til 144 øverst).

5.4 Semaphores

En semafor er en ikkenegativ int-variabel som kun kan manipuleres av *wait* [3] og *signal* [4]. Semaforene fjerner nødvendigheten av busy-wait-løkker og forenkler synkroniseringen. Wait og signal er atomiske operasjoner slik at det ikke kan gå galt selv om to tråder kaller wait likt.

[2]Interaction between two or more tasks whereby the result is unexpected and critically dependent on the sequence or timing of accesses to shared data

[3]Dekrementerer med 1 hvis verdien til semaforen er over 0. Hvis ikke venter den og dekrementerer med en når den er over 0 igjen.

[4]Inkrementerer med 1

5.4.1 Suspended tasks

Når en task kaller Wait på en semafor med verdi 0 legges den i suspendert-køen, og fjernes fra prosessoren. Når en task kaller signal, vil så RTSS (real-time support system) velger en av de suspenderte taskene som skal gjennopprettes og kjøres. Denne gjennopprettingen gjøres ofte FIFO, men kan også implementeres med prioritet. Pseudokode:

```
wait(S):
    if (S>0) then
        S:=S-1
    else
        number_suspended += 1;
        suspend calling_task
    end;
end;
signal(S):
    if number_suspended > 0 then
        number_suspended -= 1;
        resume_one_suspended_task;
    else
        S:=S+1;
    end;
end;
```

5.4.2 Implementation

Det er ganske rett frem å implementere wait og signal. Det vanskelige er å gjøre de indivisibile. Det betyr at tråden må fullføre wait eller signal før den får fortsette med noe annet. Med hjelpemidler fra RTSS er det ganske enkelt. Viktig at OS sørger for at ikke hardware f.eks. kan komme med interrupts når trådene venter i en wait-statement. Derfor kan interrupts deaktiveres i slike tilfeller (problematisk med mange tråder og delt minne hvis de bruker samme semafor). Konklusjon: Lett å sørge for at tråden fullfører wait internt, men eksterne signaler fra hardware kan ødelegge.

5.4.3 Liveness provision

Deadlocks kan oppstå når man innfører mutex. Da suspenderes alle tråder og de kan umulig fortsette. Indefinite postponement (norsk: ubestemt utsettelse) (lockout or starvation) er en mindre alvorlig feiltilsand hvor en task ønsker å få tilgang til en ressurs, men aldri får tilgang siden det er andre tasks som alltid får denne tilgangen først.

Hvis en task er fri for deadlocks, livelocks og ubestemte utsettelser sier at vi at tasken er i besittelse av *liveness*.

5.4.4 Binary and quantity semaphores

Semafor: ikke-negativ integer. **Binær** semafor er en semafor som påtar seg verdien 0 eller 1. Dvs signal(S) setter alltid S til 1. *Kvantitets semafor* derimot kan påta alle verdier større enn eller lik 0, og vil for vært signal(S,n)-kall inkrementere S med n, mens wait(S,m) vil dekrementere med m.

5.4.5 Example semaphore programs in Ada

Best å lese i boka.

5.4.6 Semaphore programming using Java

Java har pakker som støtter bruk av semaforer.

5.4.7 Semaphore programming using C/Real-Time POSIX

Har mutexing i POSIX. Har også metoder som kjører non-blocking wait.

5.4.8 Criticisms of semaphores

Trenger kun å plassere en semafor feil før hele programmet kollapser. Det kan fort oppstå deadlock i detaljerte programmer. Dette skjer f.eks. hvis man låser en mutex i C i en parent tråd og så prøver en subrutine å låse samme mutex. Ved vanlig blocking-mutex har vi deadlock så enkelt. Derfor bør man ha veldig god oversikt over hele programflyten før man fyller programmet med mutexer. Ingen språk benytter bare semaforer, men det er viktig historisk sett. Ikke tilstrekkelig for real time applications.

5.5 Conditional critical regions

Kode som er garantert å kjøre i mutual exclusion. Dvs bare en tråd kan kjøre kodesnutten til enhver tid. Når en task ønsker å gå inn i kritisk region må den evaluere guarden for å sjekke om den kan gå inn i regionen eller om den må vente. Ytelsesproblem: en task må reevaluere guarden hver gang en task forlater CCR, hvis noen allerede har gått inn i CCR suspenderes tasken igjen.

5.6 Monitors

Problem med conditional critical regions er at de kan være spredd over hele programmet. Monitor er strukturerte kontroll regioner. Lager en modul av flere prosedyrer og variabler. Denne modulen må aksesseres under mutual exclusion, og dermed kan kun maks en task ha tilgang til denne modulen til enhver tid. Alle prosedyrer i modulen er dermed garantert mutual exclusion. Man finner monitorer i språk som Modula-1, Concurrent Pascal og Mesa. Må alikevel ha condition synkronisering(ssørge for at annen tråd er i definert tilstand om nødvendig) inne i monitoren om det er behov for det.

5.6.1 Nested monitor calls

Hvis en monitor-prosedyre(f.eks. funksjon) kaller en prosedyre definert i en annen monitor, kaller vi det et nøstet-monitor-kall. Dette kan by på problemer fordi prosessen i den nøstede monitoren kan bli utsatt pga en conditon variable. Da vil også hovedmonitoren utsettes. Det kan blokkere andre prosesser som vil bruke hovedmonitoren. Dette vil begrense paralelliteten.

5.6.2 Critticisms of monitors

Elegant, men dårlig på betinget synkronisering (condititon synchronizationss).

5.7 Mutexes and condition variables in C/Real-Time POSIX

Enhver monitor har sin mutex, og alle kritiske regioner starter med pthread_mutex_lock() og avsluttes med pthread_mutex_unlock(). Condition synchronization leveres ved å assosiere condition variable med mutexen. Når man er i wait pga en conditionvariabel låses mutexen opp og gjøres tilgjengelig for andre. Når condition variabelen er godkjent låses mutexet igjen om det er ledig og programmet fortsetter.

Man kan også skille på mutexing på lesing og skriving - dvs. i noen tilfeller kan det være hensiktsmessig å la flere tråder lese samme variabel parallelt.

I C har man barrierer som fungerer som synkroniseringspunkter; det er kjekt. Ved en barriere vil trådene stå å vente til det definerte antall tråder står på barrieren før de kan kjøre videre.

5.8 Protected objects in Ada

En **protected** monitor er en monitor hvor conditon variable er erstattet med guards. Kun Ada har denne funksjonaliteten. Protected objects er Objekter som inneholder variabler og subfunksjoner (set/get-funksjoner) implementert slik at man er sikker på at ting blir utført i riktig/ok rekkefølge slik at data er til en hver tid oppdatert under mutual exclusion. Condition synkronisering leveres ved at det er boolean variabler(guards, men kalles barrierer i Ada) som må være True før en tråd får aksessere. Enkel implementering av en beskyhttet type som sikrer mutual exclusion er:

```
protected type shared_int(initValue : Integer) is
    function Read return Integer;
    procedure Write(newValue : Integer);
private
    data: Integer := initValue;
end shared_int
```

Dataen kan nå kun aksesseres med rutinene write og read. Her kan read utføres likt av mange tråder, men write kun av en tråd om gangen. Dette er fordi beskytteted funksjoner gir read-only aksess til data i funksjonen, mens beskyttede prosedyrer gir mutually exclusiv read/write. Beskyttede funksjoner må alikevel sjekke at det ikke er noen beskyttede prosedyrer som utføres før den aksesserer det den skal. En beskyttet entry har samme egenskaper som beskyttet prosedyre. Men i tillegg må en entry boolean evaluere til True før den får tilgang. Dermed kan condition synkronisering gjøres slik ved at variabelen ikke blir True før de andre trådene er i sikre tilstander.

En beskyttet Procedure i Ada gir mutual exclusive kjøring. En procedure kan ikke kjøres hvis en parallell function kjører. En beskyttet Function i Ada tillater parallell lesing, men kan ikke kjøres hvis en procedyre kjører.

5.8.1 Entry calls and barriers

For å utføre et kall til et beskyttet objekt kan en tråd enkelt og greit navngi objektet og gi nødvendig entry eller subprogram. En entry barrier til objekt evalueres når en tråd kaller en beskyttet entry og assosierende barriere kan være endret siden forrige sjekk eller når en tråd som har stått i kø må evaluere barrieren på nytt fordi den kan være endret. Barrierene evalueres kun når trådene forlater objektet. Når en tråd kaller en beskyttet entry eller subprogram kan objektet som eier de allerede være låst av annen tråd. Hvis en eller flere tråder utfører beskyttede funksjoner, sier vi at objektet har en aktiv **read**

lock. Hvis en beskyttet prosedyre utføres, sier vi at objektet har en aktiv **read/write lock**. Hvis flere tråder kaller en låst barrier vil de settes i FIFO kø. Køstrukturen kan visstnok endres i kapittel 12.3. Noen store kodeeksempler i ADA står på side 167-168.

5.8.2 Protected objects and object-oriented programming

Følgende grensesnitt er relevant for ADA når det gjelder funskjoner/prosedyrer:

- synchronized - Funksjoner og prosedyrer som kan implementeres av task type og protected type.
- Protected - Funksjoner og prosedyrer som kun kan implementeres av en protected type.
- Task - Funksjoner og prosedyrer som kun kan implementeres av task type.

Synchronization i Ada oppnås ved å kalle en entry(eller beskyttet subprogram) (shared variable communication) eller meldingsbasert kommunikasjon(kap 6). Når proggeren ikke bryr seg om formen for synkronisering er synchronized en fin metode. Når proggeren krever en spesiell synkronisering bør beskyttede grensesnitt eller tasks benyttes direkte.

5.9 Synchronized methods in Java

Monitorer kan implementeres i form av klasser og objekter. I Java har hvert objekt en lås, men den kan ikke aksesseres direkte av applikasjonen. Den er avhengig av

- The method modifier *synchronized;* - Når metode blir merket med denne modifieren kan aksess til metoden kun skje når låsen tilknyttet objektet er innhentet. Har derfor mutually exclusive aksess til data i objektet hvis objektet kun akesseres av andre metoder med den samme modifieren. Metoder uten denne modifieren kan kalles hele tiden. For full mutual exclusion må derfor alle metoder merkes med modifieren.
- block synchronization - Blokk kan merkes som synchronized.

Nøkkelordet *synchronized* tar inn det objektet som den trenger å innhente låsen fra før den får fortsette. Når blokk synkronisering brukes kan det ødelegge fordelen med monitorstrukturen litt fordi det ikke vil være mulig å se ut av et objekt hva som kreves av synkronisering. Dette fordi andre tråder kan merke objektet i et synchronized statement med blokk synkronisering.

Synchronized metoder ikke tilstrekkelig når data i et objekt er merket som static. Static data er delt av alle objekter fra samme klasse. Da må låsen tilknyttet hver klasse benyttes for å få mutual exclusive funksjonalitet.

5.9.1 Waiting and notifying

For å oppnå conditional synkronisering må vi ha flere mekanismer. Dette får vi gjennom Objects klassen. De er designet slik at de kun kan brukes i funksjoner som holder låsen til objektet. Hvis ikke kastes et exception. Tre hovedmetoder i denne klassen:

- wait() - Blokkerer kallende tråd og slipper låsen tilknyttet objektet.
- notify() - Vekker en ventende tråd så den kan fortsette. Den må alikevel vente på låsen(hvis den er låst) siden kallet ikke frigjør denne.
- notifyAll() - Vekker alle ventende tråder.

Java skiller seg ut på et punkt selv om det virker som om Java gir samme fasiliteter til monitorer. Dette er at Java ikke har noen eksplisitte condition variable. Derfor vet ikke en tråd som vekkes om grunnen til at den ble suspendert er borte. Ikke stort problem i praksis siden denne grunnen ofte er mutually exclusive.

The readers-writers problem Mange lesere og skrivere vil aksessere en stor datastruktur. Leserne kan aksessere data likt, men skriverne krever mutual exclusion. Skriverne har alltid prioritet og får skrive så fort det ikke er andre skrivere i kø. I ekstreme tilfeller kan det føre til starvation av leserne. Stort kodeeksempel i Java på side 175-176.

5.9.2 Synchronizers and locks

Java har mange pakker som støtter concurrent programmering. De utvider mekanismene beskrevet så langt. Et eksempel er locks biblioteket som gir støtte til f.eks. eksplisitte condition variable.

5.9.3 Inheritance and synchronization

Inheritance anomaly eksisterer hvis synkronisering mellom operasjoner i en klasse ikke er lokal, men kan avhenge av alle operasjoner for klassen. Når en subklasse legger til operasjoner kan det være nødvendig å endre synronisering for parentklassen for å håndtere den nye operasjonen.

5.10 Shared memory multiprocessors

Selv om et parallelt program virker strålende på en enkel prosessor, kan det hende at det feiler hvis det kjøres på et multiprosessorsystem. Kan f.eks. oppstå deadlock her som ikke var mulig hos en prosessor. Et symmetrisk multiprosessor system kalles SMP. Slike systemer har prosessorer med delt aksess til hovedminne. Et SMP program er *sekvensielt konsistent* hvis resultatet av enhver gjennomkjøring av programmet er lik som om alle instruksjoner i alle prosessorer er utført sekvensielt. I tillegg skal instruksjonene til hver tråd i en sekvens være lik de som er spesifisert av programmet. Kan ikke endre rekkefølgen på instruksjoner så det begrenser dagens prosessorer som ofte gjør dette for effektivitet.

5.10.1 The Java memory model

En trace modell kan beskrive et parallellt program. En trace modell definerer meningen med en tråd som settet av alle sekvenser(traces) som tråden kan utføre. Program execution er derfor settet med alle tråd traces. **Memory model** beskriver, gitt et program og settet med tråd traces, om execution tracen er lovlig. (lovlig program-flyt eller ikke? diffust begrep). Minnemodellen i Java bryr seg om **memory actions** som er lese og skrive operajsoner på delte variable mellom tråder. Har følgende sammenheng mellom variabler og tråder:

- Når en tråd starter en annen vil endringer gjort av parent tråden før den startes med start kallet være synlige for child tråden når den utfør sitt arbeid.
- Når en tråd venter på at en annen terminerer vil endringer gjort av tråden som terminerer bli synlig for den ventende tråden når den kan fortsette.
- Når en tråd avbryter en annen vil endringer gjort av den avbrytende tråden før interupten være synlig for den som ble avbrutt når den oppdager interupten.
- Når tråder leser og skriver til samme lokale(volatile) minne(volatile variable - lesing og skriving rett på registre eller cache) vil endringer gjort av den skrivende tråden til delt data(før den skriver til volatile minnet) bli synlig for en annen tråd som leser samme volatile minnet.

5.10.2 Ada and shared variables

Volatile pragma: Alle lese og skriveoperasjoner på en variabel går direkte til minnet. Atomic pragma: Alle lese og skrivinger av variabel

må gi samme resultat/må være konsistente.

5.11 Simple embedded system revisited

6 Message-based synchronization and communication

I stedet for delte variable, kan man benytte meldinger for å kommunisere og synkronisere mellom tråder.

6.1 Process synchronization

Hvis en tråd prøver å motta en melding før en melding har blitt sendt, will denne tråden suspenderes til meldingen kommer frem. Sendingen av meldinger kan sendes på tre måter:

- Asynchronous (no-wait). Senderen fortsetter umiddelbart etter sending uansett om meldingen kom frem eller ikke (eksempel: brev i posten). Problem: man vet ikke om meldingene som er sendt har blitt behandlet, så man kan ha fyllt opp mottaksbufferet på den andre siden. Man trenger derfor uendelig store buffre for å være sikker på at man ikke mister data.
- Synchronous: senderen fortsetter etter at meldingen er mottatt (å ringe med telefon: senderen venter til han har fått kontakt med personen på den andre siden før han fortsetter).
- Remote invocation: Senderen fortsetter etter at den har mottatt et svar fra mottakeren. (å ringe med telefon: hvis personen som ringer krever å få svar på et spørsmål fra personen i den andre enden før han/hun fortsetter med livet sitt)

Ved hjelp av asynkron kommunikasjon kan de andre metodene konstrueres(sende acknowledge etter mottak f.eks, men sender venter på denne). Det er alikevel noen problemer ved kun å bruke asynkron meldingssending:

- Krever uendelig store bufre fordi man aldri vet når en melding er lest.
- Asynkron sending er gammeldags og det er som oftest lagd slik at asynkron kun benyttes for å designe synrkon sending.
- Krever mer kommunikasjon og kompliserer programmet.
- Vanskeligere å bevise korrekthet.

6.2 Task naming and message structure

Beskriver til hvem meldingen skal sendes til og hva som skal sendes. Eksempel på en direktemelding

```
send <message> to <task-name>
```

Eksempel på indirekte melding (kan være til en kanal, mailboks, link eller pipe):

```
send <message> to <mailbox>
```

Et "Naming scheme" for en synkron direkte meldingsutveksling kan være slik. utvekslingen er synkron hvis både senderen og mottakeren navngir hverandre:

```
send <message> to <task-name>
wait <message> from <task-name>
```

Mens et "Naming scheme" for en synkron indirekte meldingsutveksling kan være slik:

```
send <message> to <mailbox>
wait <message> from <mailbox>
```

En meldingsutveksling er asynkron hvis mottaker ikke spesifiserer hvem den ønsker motta meldinger fra.

```
wait <message>
```

Eksempler på asynkron utveksling er server/klient der serveren mottar meldinger fra hvilken som helst klient.

6.2.1 Message structure

Kan være problematisk med meldinger å sende objekter av forskjellig typer siden disse kan ha forskjellige representasjoner på sender- og mottakersiden.

6.3 Message passing in Ada

Ada har mekanismer for kommunisering og synkronisering via meldinger.

6.3.1 The Ada model

Remote invocation minner om funksjon som kaller en annen funksjon, så utfører denne et arbeid og returnerer et resultat. Derfor tar Ada inspirasjon fra prosedyrer og entries. For at en tråd skal motta en melding må den definere en **entry**:

```
task type Screen_Output(ID : Screen_Identifier) is
    -- Definer en task
    entry Call(Value : character, X, Y : Integer);
end Screen_Output
-- Opprette en task:
Display: Screen_Output(parA) // parA = Screen_Identifier
```

Entry er nå en prosedyre hvor man sender en char og to ints. Det er altså en funksjon andre tråder kan benytte for å sende noe. En entry kan deklareres som privat ved å skrive private på linja over. Da er det kun tråder lokalt i tråden som kan kalle entryen. Ada støtter også opprettelse av **entry families** som er en array av entries. En MUX med 7 innganger f.eks. kan da definere sine innganger ved en entry familiy i stedet fr 7 separate entries. For å sende en beskjed til tråden navngir man tråden og entryen direkte:

```
Display.Call(charA, intB, intC);
```

Display er navnet på en tråd av typen Screen_Output(den typen har entry som heter call). Hvis en entry kalles på en tråd som ikke lenger kjører utløses exception'et *Tasking_Error*. For å motta en melding må man akseptere den riktige entryen:

```
accept Call(C : Character, I,J : Integer) do
Local_Array(I,J) := C;
```

En accept statement må stå deklarert inne i task body'en. Alle entries bør ha en accept prosedyre definert. Deklarasjonen kan til og med stå inne i en annen accept deklarasjon, dog med sin egen entry. Tråden som sender noe kan ikke identifiseres på mottakssiden uten å legge ved en ID. Hvis mange tråder sender meldinger til en entry vil meldingene legges inn i FIFO-kø.

6.3.2 Exception handling and the rendezvous

Siden Ada kode kan utføres i en remote invocation(kalles og rendezvous) kan det faktisk utløses et exception inne i en accept statement. Da vil accept statement terminere som normalt hvis det er definert en handler inne i accepten eller i en blokk nøstet inn i accept statementet. Hvis ikke vil accept statementet terminere umiddelbart. I det siste tilfellet vil exception'et bli utløset på nytt(reraised) både i den kallende tråden og tråden som ble kalt. Tråden som ble kalt vil utløse exceptionet på nytt umiddelbart etter accept statementet, mens den kallende tråden vil gjøre det etter entry-kallet. Scope problemer kan gjøre at exceptionet er anonymt for den kallende tråden slik at den ikke får gjort noe. Hvis man har

en tråd som ber en annen tråd åpne en fil, kan det oppstå to exceptions. Det første unntaket er *Device_Off_Line* som løses i acceptstatementet(mottakertråden). Den vil prøve å reboote enheten og sender vil ikke vite noe om denne aktiviteten. Det andre unntaket er *File_Does_Not_Exist* som kommer av feil kall fra sender(feil filnavn etc.). Dette løses derfor ikke inne i accept, men propagerer ut til den kallende tråden som må løse det her.

6.3.3 Message passing via task interfaces

En tråd spesifikasjon i Ada bestemmer hvordan andre tråder kan kommunisere med den. Dette betyr vanligvis at klienten(tråden som ønsker noe) må navngi en annen spesifikk tråd. Noen ganger er dette tungvint hvis tråden kun vil ha utført en generell tjeneste mange tråder kan utføre. Da benyttes task interfaces som deklareres slik:

```
package Simple_Task_interface is
    type Simple_TI is task interface;
    procedure OP1(TI : in out Simple_TI) is abstract;
    type Any_Simple_TI is access all Simple_TI'Class;
end Simple_Task_Interface;
```

Et objekt av aksesstypen kan referere enhver tråd som implementerer denne interfacen. Følgende objekter kan referes til på denne måten:

```
task type EX_1 is new Simple_TI with
    overrriding entry OP1;
end EX_1;
```

6.4 Selective waiting

Hittil har vi sett på kommunikasjon hvor mottakeren av en beskjed kun venter på en beskjed fra en bestemt tråd eller channel. Dette er for restriktivt. Vil ha mottakere som kan vente på beskjeder fra mange tråder(som server venter på mange klienter). En guard er noe som sørger for at en kommando kun utføres hvis en betingelse evaluerer til true(i prinsippet likt if). MEN guarder som listes etter hverandre hvor kun en av de velges (legger til firkant foran alle guarder under første) vil skape ikke deterministisk oppførsel hvis mange evalueres til True. Grunnen er at da velges en av guardene som er oppfylt helt tilfeldig(skiller seg fra if-strukturer her). Hvis ingen guarder evalueres til True anses det som en feil og statementet og tråden som utførte det terminerer. Hvis en kommando som beskyttes er en operator som mottar meldinger kalles det selektiv venting siden du da kan lage prioriteringsrekkefølger. I tillegg kan

du få en av flere tilfeldige tråder til å utføre noe om det ikke er viktig hvem som tar den. Dette bør gi bedre arbeidsfordeling. Kan også implementeres for sender i noen språk.

6.5 The Ada select statement

Når en server må takle kall til to eller flere entries fra andre kreves selektiv venting noe som gis gjennom select-statement. Ser for oss at vi har server med to entries(S1 og S2), kan da utføre to tjenester.

```
task body server is
    ...
begin
    loop
        ... Prepare for service
        select
            when (bool uttrykk) =>
                accept S1(..) do
                    .. kode
            end S1;
        or
            accept S2 // og så videre nedover
```

Hvis ingen av acceptene over utføre(kan ha boolean ved or og sikkert) er det tre muligheter:

- Termineringsalternativ - Hvis ingen andre tråder kan kalle entryene lenger kan tråden termineres(serveren). Ikke behov for server mer.
- Else alternativ - Utføres hvis det er definert og ingen av statementene over utføres.
- Delay - Ikke relevant(kap 9 ikke pensum)

6.6 Non-determinism, selective waiting and synchronization primitives

Grunnen til at det velges tilfeldig blant select statementene er at concurrent språk vanligvis ikke antar noe om hvilke tråder som utføres først. Antar at scheduler er ikke-deterministisk. Derfor slippes semaforer tilfeldig og det er ikke noe køhndtering når det kommer til semaforkøer. Entry køer og monitorkøer er derimot ofte lagd FIFO for å slippe starvation (kan alikevel oppstå, men mindre sannsynlig).

6.7 C/Real-Time POSIX message queues

Støtte for asynkron indirkete meldingsoverføring gjennom meldingskøer. Kan legge prioritet ved beskjedene for å snike i køen. Meldingskøer er vanligvis burkt for kommunikasjon mellom tråder i forskjellige prosesser selv om det er mulig med sending mellom tråder i samme prosess(mutex og delt minne mer effektivt). Køene har definert maxlengde på meldinger og et max antall meldinger det er plass til (kan settes ved attributter). Stort kodeeks s.207.

6.8 Distributed systems

OBS: FINT OM DU LESER GJENNOM HELE DELKAPITLET. JEG SLITER MED Å FORSTÅ HVA SOM ER VIKTIG OG SER IKKE SAMMENHENGENE HER.

Tråder kjører på forskjellige noder slik at man vil gjøre kommunikasjon enklest mulig. Trenge dessverre i praksis kompliserte protokoller siden vi vil gi støtte for at:

- Tråder som sender skal slippe å gi det den sender passende format og konvertere til bistrenger og dele opp i pakker.
- Tråder som mottar kan anta at alle pakker som kommer frem er hele og inntakt og den mottar kun pakke når alle delpakkene er nøstet sammen. Hvis et bit er korrupt må det korrigeres eller meldingen ikke overføres til mottakertråd.
- Meldinger som mottas er på det formatet som mottaker forventer
- Kommunikasjon skal ikke bare kunne utføres på predefinerte datatyper. Vil ha data som er relevant for mottaker der og da(abstrakte typer og klasser kan brukes).

6.8.1 Remote procedure calls (RPC)

Måte distribuerte programmer kan kommunisere med hverandre. RPC benyttes ofte i kommunikasjon mellom programmer skrevet i samme språk. Sett fra klient-server applikasjon har vi to mellomledd: **server stub** og **client stub**. Dette er mellomledd som sørger for at kravene over opprettholdes.

Remote procedure calls and remote invocation

På maksiner med en prosessor utfører tasks forskjellige funksjoner for å gi kontroll fra en del av koden til en annen. Remote procedure utvider ideen slik at en tråd som utfører noe på en prosessor

kan utføre noe annet på en annen prosessor. Så må kanskje de to prosedyrene synkronisere slik at delte variable eller meldingssending benyttes.

VANSKELIG Å FINNE ESSENSEN HER

6.8.2 The distributed object model

Distrubuted object model tillater at:

- Dynamisk opprettelse av objekter i ethvert språk kan skje på ekstern(remote) maskin(annen prosessor?)
- Idinetifisering av objekt som bestemmes kan holdes på enhver maskin.
- Transparent kall på en ekstern metode som om den var lokal.
- Transparent run-time levering av et metodekall over nettverk

Ikke alle språk støtter denne funksjonaliteten, men vi skal se på noen språk.

6.8.3 Ada

Ada's modell for å kommunisere mellom partisjoner i distribuert program er via kall til eksternt subprogram (prosedyre eller funksjonskall). Det er tre måter en kallende partisjon kan gjennomføre et eksternt subprogram kall:

- Ved å kalle subprogram som er deklarert i eksternt kall-interface pakke i en annen partiosjon direkte.
- ved å derefere en peker til et eksternt subprogram
- Benytte run-time levering til en metode i et eksternt objekt.

6.8.4 Java

Java gir praktisk måte å aksessere nettverksprotokoller, men de er komplekse. Derfor støtter også Java bruk av distributed object model gjennom eksterne objekter. I Java benyttes det en pakke kalt *rmi* som er bygd på TCP protokollen. Inne i denne pakken er det et *Remote* grensesnitt. Utviding av dette grensesnittet brukes til å gi støtte til distribuerte systemer. Er så egne egne klasser definert i pakkene som kan benyttes som server(inne i rmi).

6.8.5 COBRA

Gir den mest generelle distribuerte modellen. Har som mål å legge til rette for samarbeid mellom applikasjoner skrevet i forskjellige språk. Modellen bygger på:

- **Object Request Broker(ORB)** - software kommunikasjonsbuss som gir støtte for infrastruktur mellom heterogene applikasjoner.
- **obejct services** - Samling grunnleggende tjenester som støtter ORB(opprette objekter, navngiving og aksesskontroll).
- **common facilities** - Samling funksjoner som er felles for mange applikasjoner(user interfaces, docoument and database managment)
- **domain interfaces** - Gruppe av interfaces som støtter spesifikke applikasjonsdomenener (bank, finans eller telecommunications for eksempel).

7 Atomic actions, concurent tasks and reliability

7.1 Atomic actions

Har sett på synkronisering og kommunisering mellom tråder som deler ressurser. Nå skal det ses på hvordan man kan strukturere grupper av tråder slik at de får koordinert aktivitenene sine. Har hittil kun sett på to tråder som kommuniserer, men i praksis kan det være nædvendig at mange flere tråder samarbeider. Det kreves at hver gruppe med tråder utfører sine felles aktiviteter som en **atomic action**. En enkel tråd kan også ønske å beskytte seg fra forstyrrelser. Derfor kan en atomic action inneholde en eller flere tråder. En atomic action er blitt definert på forskjellige måter:

- En handling(action) er atomic hvis trådene som utfører den ikke vet om noen andre aktive tråder, og at ingen andre aktive tråder vet om aktivitetene til trådene som utfører handlingen mens de utfører handlingene.
- En handling er atomisk hvis trådene som utfører den ikke kommunserer med andre tråder(enn de som utfører handlingen) underveis.
- En handling er atomisk hvis trådene som utfører den ikke kan oppdage tilstandsendringer annet enn de endringene de utfører selv. I tillegg skal de ikke avsløre sin tilstand før handlingen er helt ferdig.
- Handlinger er atomiske hvis de kan betraktes som usynlige og momentane slik at de kan ses på som sekvensielle i stedet for parallelle.

Til tross for at atomic actions betraktes som usynlige kan de ha en intern struktur. Da kommer begrepet **nested atomic action** inn. Her har vi en nøstet handling som består av et subset av trådene som utfører den ytre action. Hvis ikke kunne en annen tråd smugle ut info om den ytre handlingen slik at den ikke lenger er usynlig.

7.1.1 Two-phase atomic actions

Ideelt sett skulle alle tråder som inneholder atomic actions få de ressursene som trengs for å utføre en handling før man starter å utføre den. Ressursenene slippes så etter at handlingen er ferdig. Dessverre fungerer ikke dette optimalt i praksis. Det gir dårlig ressursfordeling og må løses på andre måter. Derfor startes ofte atomic actions uten at alle nødvendige ressurser er ledige. Optimalt

skulle en atomic action ha sluppet ressurser den ikke lenger bruker underveis. Dessverre vil dette bryte med å oppdage endring i tilstand hos andre siden man om man henter inn ressursen igjen kan se endringer hos andre som har lånt ressursen i mellomtiden. Derfor er en løsning en to-fase politikk. I første fase kan man be om flere ressurser. I andre fase kan man frigi ressurser, men aldri be om flere. Derfor slippes ressurser først når man vet at man aldri vil trenge flere ressurser igjen. Et problem med å slippe ressurser underveis er hvis handlingen feiler og man må begynne en recovery. Da kan den frigitte ressursen ha endret andre tilstander og forverre muligheten for å utføre handlingen på nytt.

7.1.2 Atomic transactions

En atomic transaction besitter alle egenskaper til en atomoc action, men har i tillegg følgende egenskaper:

- failure atomicity - Handlingen må feile slik at den ikke har noen effekt på systemet eller terminere plettfritt.
- synchronization atomicity - Helt usynlig slik at del deler som utføres ikke kan observers av noen andre parallelle trasactions.

Hovedpoenget med atomic transactions er at hvis de feiler skal man ha lagret systemets tilstand for de delene som påvirkes av handlingen slik at man kan en komplett trygg backup til der handlingen ble startet. Dette er ikke tilfelle for en atomic action hvor systemet ved feil kan komme i en udefinert tilstand. Atomic transactions er gode for applikasjoner som endrer en database, men passer ikke for fault-tolerance siden recovery metodene gis av operativsystemet her og ikke koden.

7.1.3 Requirements for atomic actions

- **Well-defined boundaries** - Alle atomic actions må ha en definert start, slutt og sidegrenser(deler trådenene som er involvert i handlingen fra de som ikke er det).
- **Indvisibility (isolation)** - Kan ikke tillate utveksling av info mellom tråder aktive i handlingen og de utenfor. Ingen tråder får lov til å forlate en action før alle er enige om det(synkronisering til slutt).
- **Nesting** - Kan nøstes så lenge de ikke overlapper med andre atomic actions. Det vil si at en indre action må være helt omsluttet av den ytre.

- **Concurrency** - Bør være mulig å utføre forskjellige actions parallellt. Resultatet av dette må være det samme som om hadlingene ble utført sekvensielt.
- Må tillate at recovery prosedyrer kan programmeres.

Obsobs: Noen definisjoner av Atomic Actions krever at alle tasks må synkroniseres både ved start og slutt.

7.2 Atomic actions in C/Real-Time POSIX, Ada and Real-Time Java

Antas i dette kapitlet at ressurser enten kan deles eller ikke deles. Ressursfordeling kommer i kap.8. Antas og at alle atomic actions er to-fase som beskrevet tidligere.

7.2.1 Atomic actions and C/Real-Time Posix mutexes

Trenger mutex, condition variable og delte variable. I tillegg er det bool variable som lagrer tilstand i en action. Kodeeksempler i boka på s.233-234. For å gå inn i en action må kontrolleren først sørge for at den har mutual exclusion og deretter lage en lås så den ikke blir forstyrret. Så sjekker den at ingen andre tråder utfører handlingen allererede. Hvis det er det tvinger kontrolleren den kallende tråden til å vente til action'en er ledig. Antar så at to tråder sammen utfører en action. Når den ene er ferdig sjekker den om den andre er ferdig og venter hvis den ikke er det. Når begge er ferdig returnerer trådene slik at de er klare for en ny handling. Det eneste problemet er at man ikke kan forsikre seg mot at trådene kommunisrer med andre tråder underveis.

7.2.2 Atomic actions in Ada

Lik som fremgangsmåte i C/Re.. , men benytter guards i stedet for condition variable. Antar at vi har en definert action(actionX) med tre oppgaver som skal utføres i en atomic action. Kontrolleren kan da implementeres som følger:

```
package body actionX is
    protected Action_Controller is
        entry First;
        entry Second;
        entry Third;
        entry Finished;
    private
```

```
        First_Here : Boolean := False;
        ..
        Third_Here : Boolean := False;
        Release : Boolean := False;
    end Action_Controller;
```

Når en tråd melder seg for tjeneste settes bool n_Here til True(benytter en entry). Når nok tråder har meldt seg utføres oppgaven. Så fort en tråd er ferdig benytter den entry finished. Her blir den hengende til alle er ferdig. Barrieren blir da True og alle slippes. Implementering av to entries følger(disse skal ligge inne i kroppen til Action_Controller):

```
entry Second when not Second_Here is
begin
    second_Here := True
end Second

entry Finished when Release or Finished'Count = 3 is
begin
    if Finished'Count = 0 then
        Release := False;
        First_Here := False;
        Second_Here := False;
        Third_Here := False;
    else
        Release := True
    end if
end Finished
```

7.2.3 Atomic actions in Java

Kan følgende lignende struktur i Java, men dette delkapitlet ser heller på hvordan metoden kan ekspanderes til å benytte arv. Det er hårreisende mye kode i boka fra side 237-240. Poenget er at du har en hoveklasse bestående av en action som kan utføres av tre tråder:

```
public interface ThreeSomeAtomicAction{
    public void role1();
    ..
    public void role3();
}
```

Denne strukturen kan inkluderes i andre klasser slik at de kan lage forskjellige oppgaver for tre tråder. Denne klassen kan igjen inklud-

eres i en ny en slik at mulighetene ved å benytte arv er store. Implementeringen av ting som sørger for at kravene opprettholdes må nesten studeres i boka.

7.3 Recoverable atomic actions

I strategiene over må proggeren selv sørge for at trådene i en atomic action ikke kommuniserer med andre. Antar og at ingen tråder kan terminere uten videre. Har nå noen nye antakelser. Vi antar nå at tråder som går inn i en action ikke blokkeres. Den blokkeres kun hvis den må vente på en ressurs eller hvis den prøver å kommunisere med en annen tråd i action'et som enten er aktiv i action'et, men ikke kan kommunisere nå eller at den enda ikke er aktiv. Tråder kan forlate en action når alle aktive tråder vil forlate den. Her kan altså et subset av tråder gå inn i og forlate en action. Dette løser **deserter** problemet som er at alle tråder venter i en action på en tråd som ikke har gått inn i action'et(siden ingen tråder kunne forlate en action før alle var ankommet).

7.3.1 Atomic actions and backward error recovery

Atomic actions danner en recovery line av seg selv slik at domino effekten unngås. Hvis en feil oppstår med en tråd i en atomic action kan den hoppe tilbake til start og utføres på nytt(evt med alternativ strategi). Kontrolleren sørger for at tråden ikke utleverer korrupt informasjon til andre tråder. Når atomic actions brukes slik kalles det **converations**. Da har hver action statement en recovery blokk knyttet til seg. Det fungerer på følgende måte:

- Ved entry til en converation lagres tilstand. Da danner alle entrypunkter en recovery line.
- Trådene kan kun prate med de inne i en action som vi vet.
- For å forlate en conversation må alle aktive tråder gå gjennom akseptansetesten. I så fall kastes alle recovery punktene.
- Hvis ikke alle passerer testen må ALLE tråder hoppe tilbake og utføre en alternativ strategi.
- converationss kan nøstes, men bare ved at hele den nøstede atomic action ligger inne i den ytre.
- Hvis alle alternative strategier feiler må recovery skje på et høyere nivå.

Ulempen med denne fremgangsmåten er at alle tråder må gjøre arbeidet på nytt om en feiler. Dette er ineffektivt og ofte ikke beste strategi.

7.3.2 Atomic actions and forward error recovery

Ved forward error recovery vil det hvis en tråd har utløst en exception bli utløst samme exception hos samtilige tråder i en atomic action. Exceptionet sies å være asynkront siden det utløses av en annen tråd og ikke samtidig hos alle. Hvis en termineringsstrategi benyttes vil en atomic action terminere normalt hvis alle tråder går gjennom en handler uten å utløse flere exceptions. Hvis en resumption(gjenopptakelse) modell benyttes vil de når alle er ferdig med sine handles fortsette der exceptionet oppstod. Hvis en tråd feiler i å gjennomføre sin handle eller det ikke eksisterer relevant handle vil en atomic action feile og exception utløses.

Resolution of concurrently raised exceptions

Hva skjer hvis to forskjellige exceptions utløses samtidig når flere tråder er aktive? Da kan hver tråd ha en handling for hver exception, men den vet ikke hvilken den bør gjennomføre. Dette kan løses gjennom et exception tre. Da vil det exceptionet som inneholder alle andre i et subtre(rota i ssubtreet) bli gjennomført først. Hver action kan definere sitt eget tre. (Lenka liste eller?)

Exceptions and internal atomic actions

Ved nøstede atomic actions kan en tråd utløse et exception mens andre aktive tråder jobber inne i den nøstete atomic action. Vi vet jo at da må alle aktive tråder bidra i recovery. Dette går ikke siden den indre pr. def skal være usynlig. En løsning er å holde tilbake exceptionet til den interne actionen er ferdig, men dette er ikke god løsning fordi det kan være at exceptionet kommer av en timeout og det kan få fatale konsekvenser og la det vente da. Det kan også være et exception som sørger for at den interne actionen vil komme i en deadlock slik at alt henger seg. Den andre løsningen som i stedet benyttes er at den interne har et eget exception for slike tilfeller. Da må det indre avbrytes slik at den ytre kan gjøre en recovery. Hvis den interne ikke kan avbryte seg selv må den utløse exception om at alt feiler atomic_action_failure.

7.4 Asynchronous notification

I reele sanntidssystemer kan det være nødvendig å kombinere forward og backward recovery. Dette fordi en trygg tilstand må gjenopptas og eventuelle feil som er gitt til omgivelsene må gjøres opp for med forward recovery. For å få til dette(recoverable action) må det være en metode for å få oppmerksomheten til en tråd i en action og

si at en feil har oppstått i en annen tråd (skjer asynkront). Det er to grunnleggende modeller for dette akkurat som med exceptions, resumption(gjenopptakelse) og terminering.

Gjenopptakelsemodellen kalles ofte **event handling** og oppfører seg som en software interrupt. En tråd markerer hvilke events den er villig til å ta. Når en slik event signaliseres blir tråden interrupted og en event handler gjennomføres før tråden igjen får fortsette som normalt. Dette er likt som gjenopptakelsemodellen for exceptions, men forskjellen er at tråden vanligvis ikke signaliseres av tråden som feilet, men signaliseres asynkront (av en kontroller kanskje).

Termineringsmodellen vil hver tråd spesifisere et domene i sin utførelse hvor den kan forstyrres og termineres. Dette kallees **asynchronous transfer of control**(ATC). Når en tråd er ferdig med å utføre det den skal da returneres kontrollen til tråden, men den fortsetter ikke på samme sted. Hvis tråden er på et sted i koden den ikke kan avbrytes settes "interrupten" i en kø eller ignoreres. Ada og Java støtter denne metoden.

7.4.1 The user need for asynchronous notification

Det fundamentale behoved for dette er å la en tråd reagere raskt på noe en annen tråd oppdager. I teorien kan man vente til tråden har tid, men i praksis er ikke dette raskt nok. Dette gjeløder blant annet følgende situasjoner:

- **Error recovery** - Vet at i atomic actions må alle tråder bidra og det kan være at det er utrygt å fortsette fordi en betingelse som tidligere har gjort at en tråd kan gjøre noe plutselig ikke er oppfyllt lenger.
- **Mode changes** - Et fly endrer modus midt i take-off fra take-off til emergency vil måtte håndteres umiddelbart. Da kan ikke take-off prosedyrene gjøre seg ferdige.
- **Scheduling using partial/imprecise computations** - I store regnestykker kommer nøyaktighet an på tid og det kan være at mellomresultater med bedre og bedre nøyaktighet gis ut. Ved feil må slik regning avbrytes slik at resultatet ikke endres etter en feil.
- **User interrupts** - Hvis bruker oppdager en feil og vil starte på nytt.

7.5 Asynchronous notification in C/Real-Time POSIX

Støtter en gjenopprettelsesmodell basert på signaler og en modell for å avbryte tråder.

7.5.1 C/Real-Time POSIX signals

Støtter gjenopprettelsesmodellen. Det er mange forhåndsdefinerte signaler som har en tilhørende int-verdi. (SIGALARM f.eks.). De fleste slike har også en handler. Man kan definere signaler selv, men bare de med intverdi mellom *SIGRTMIN* og *SIGRTMAX* regnes som real-time signaler. Et real-time signal er definert som et signal som kan gi ekstra info til en handler av prosessen som genererer det og i tilegg settes de i kø. Det er tre måter et program kan varsles om at et signal er generert.

- SIGEV_NONE - Ingen varsling
- SIGEV_SIGNAL - settes i kø
- SIGEV_THREAD - Ny tråd starter og håndterer signalet.

Det er igjen tre måter en prosess kan håndtere et signal:

- Blokkere det og så håndtere det senere eller akseptere det(har en liste over signaler som er blokkert). OBS: Noen signaler kan ikke blokkeres.
- Kalle funksjon. Denne funksjonen kan væreforhåndsdefinert eller predefinert. En funksjon som håndterer et signal kan settes opp med *sigaction*.
- Ignorere det helt, signalet forsvinner da og håndteres ikke.

Når det gjelder å håndtere et signal gjennom å kalle en funksjon skiller man mellom vanlige signaler og real-time signaler. Funksjoner definert for real-time signaler kan ta inn en pointer til en struct(ferdiglagd, der den ekstra informasjonen kommer inn), mens vanlige signaler kun kan ta inn en int. Hvis flere real-time signaler står i kø velges det med lavest int verdi først. En prosess kan generere et signal som sendes til en annen prosess på to måter. Den ene er via funksjonen *kill* og den andre er via funksjonen *sigqueue*(kan kun sende real-time signaler). En prosess kan også be om at et signal sendes til seg selv.

Signals and threads

- Signaler generert som følge av synkron feil(feil minneaksessering f.eks) gis kun til tråd som foårsaket feilen.
- Andre signaler kan være ment for en hel prosess, men kun en tråd vil motta signalet.
- sigaction funksjonen setter handler for alle tråder i en prosess.
- Det er ikke definert hvilken tråd som mottar signal om flere tråder kan motta det. Hvis en handler bestemmer at terminering må til termineres hele prosessen og ikke bare en tråd.

Noen av systemkallene i POSIX er definert som **async-signal unsafe** og **async-cancel unsafe**. Det er ikke definert hva som skjer hvis et signal avbryter en slik funksjon.

7.5.2 Asynchronous transfer of control and thread cancellation

Setjump og *longjmp* er tidligere brukt til å definere termineringsmodell for exceptions. Disse funskjonene kunne vært brukt til termineringsmodell her, men funksjonene er async-signal unsafe så de passer ikke. Det er støtte for å avbryte tråderasynkront i C/Real-Time POSIX, men det finnes måter disse mekanismene kan blokkeres på for en tråd. Tråder kan selv bestemme når og om de skal kunne termineres. Det kan også lages funksjoner som kalles når en tråd terminerer slik at dette skjer trygt. Dette kan innebære å frigjøre allokert minne.

Det er lettest å implementere Atomic actions hvis termineringsmodellen for signaler benyttes. Man bruker da de innebygde trådfunksjonene for å styre når tråder skal terminere.

7.6 Aynchronous notification in Ada

I Ada er det støtte slik at en applikasjon kan svare på:

- Events som signaliseres asynkront fra en ekstern omgivelse.
- Events som trigges av at det går tid.
- Asynchronous transfer of control(ATC) forespørsler til en tråd som støtter termineringsmodellen. Ingen mekanisme for gjenopprettelsesmodellen(resumption).
- Stans av tråd.

7.6.1 Asynchronous transfer of control

Select-statementet introdusert tidligere har også en asynkron form som kan gi en asynkron notifiseringsmekanisme med terminering-modellen. Utførelsen av et select statement begynner med å trigge et entry call. Dette kan utføres umiddelbart eller settes i kø. Hvis det utføres umiddelbart vil parameterne evalueres som normalt og kallet utføres. Hvis kallet settes i kø vil en sekvens av statements i en del som kan avbrytes utføres. Hvis denne sekvensen blir ferdig før entryen er ferdig vil det gjøres et forsøk på å kansellere entryen. Da er selecten ferdig. Hvis ikke delen som kan avbrytes blir ferdig før entryen, vil delen avbrytes. Til slutt vil da en sekvens av valgfrie statements utføres(må være definert på forhånd i en optional del).

```
\textbf{select}
    Trigger.Event;
    -- Optional sequence of statements to be executed after
    -- the event has been received
\textbf{then abort}
   -- abortable sequence of statements
\textbf{end select}
```

7.6.2 Exceptions and ATC

Potensielt kan to aktiviteter skje samtidig med denne formen for select. Både delen som kan avbrytes og entryen kan kjøres likt. Da kan det utløses exception i begge deler som ikke håndteres. Heldigvis propagerer kun ett av disse siden det i den delen som kan avbrytes vil slette exceptionet når den er ferdig.

7.6.3 Task abortion

Tråder kan avbrytes med en abort statement i Ada og tråder kan avbryte andre tråder på denne måten. Når en tråd avbrytes sier vi at den blir *abnormal*. Da hindres den i å kommunisere med andre tråder i de tilfellene hvor den ikke termineres umiddelbart(noen tråder har funksjoner som kjøres ved terminering).

7.6.4 Ada and atomic actions

Ada ATC sammen med exception handling kan benyttes til å implementere backward og forward recovery.

Backward error recovery

Massive eksempler fra s.260-262. I utgangspunktet defineres pakke for å lagre en tråds tilstand slik at recovery kan utføres. Kontrolleren som styrer en action sørger for at feil i en tråd propageres til alle andre tråder, at vedvarende data lagres og gjenopprettes i et recovery cache og at alle tråder forlater en action likt. Kontrolleren har derfor følgende forskjellig entries som trådene kan benytte:

- *Wait_abort* er en asynkron event som sier at trådene venter mens de utfører sin del av en action.
- Hver tråd kaller *Done* når de er ferdige.
- Hver tråd kaller *Cleanup* hvis de har måttet utføre en recovery.

- Hvis en tråd oppdager en feil vil den kalle *Signal_Abort* og må gjenopprettes. Flagget *Killed* settes til True. Når dette flagget blir True vil alle motta en event (feilen, en exception f.eks.). Når feilen er løst må trådene vente på Cleanup entryen så de kan terminere sammen.

Fin figur avoppsettet på side 262.

Forward error recovery

Mye likt som for backward når det gjelder kontroller(propagering av exceptions og kun kommunikasjon internt mellom trådene i en action). Antar nå at tre tråder sammen utfører en atomic action. Hver tråd har lik struktur og utfører en select-statement med en del som kan avbrytes(abortable part). I delen som kan avbrytes ligger den faktiske koden som utføres, mens entryen til selecten kun brukes når exception utløses og ikke håndteres av tråden som oppdaget den. Hvis delen som kan avbrytes fullføres, har tråden gjort sin del og kan vente på at de andre blir ferdige. Hvis en exception utløses mens delen utføres vil kontrolleren informeres og identiteten til exceptionet gis. En forskjell fra backward recovery er at årsaken til en eventuell exception må gis til trådene. Det er to problemer med forward recovery:

- Bare det første exceptionet som utløses vil sendes til kontroller og utløses hos alle tråder. Dette fordi alle exceptions i delen som kan avbrytes da vil slettes om de oppstår likt.
- Fremgangsmåten håndterer i utgangspunktet ikke "deserter" problemet (alle tråder i en action venter på en som ikke er ankommet).

Vanvittig figur på side 266.

7.7 Asynchronous notification in Real-Time Java

Real-time Java støtter både termineringsmodell og gjenopprettelsesmodell, men gir ikke direkte støtte til asynkron tråd terminering. Gjenopprettelsesmodellen gis gjennom en asynkron event-handling fasilitet hvor hver handle er en planlagt enhet i stedet for at den utføres som følge av en tråd som avbrytes av en interrupt. Termineringsmodellen baserer seg på asynkron exception handling for real-time tråder.

Resumption-modellen benytter asynkron event handling (interrupt/exception), hvor interruptrutinene er planlagte.

7.7.1 Asynchronous event handling

En real-time event i Java er ekvivalent med et signal i C/Real-Time POSIX.

7.7.2 Asynchronous transfer of controll in Real-Time Java

Tidligere kunne man i Java la en tråd asynkront påvirke en annen tråd, men det er nå erstattet med noe mer hendig: En tråd kan påvirke en annen tråd (signal'e et interrupt) ved å sende et interrupt. Hvis tråden som blir påvirket er i wait, sleep eller join, vil den vekkes opp og et exception vil kastes. Hvis tråden er kjørende, vil et interruptflag settes - tråden må periodisk polle på disse interruptflaggene for å merke at det har blitt interruptet.

I seg selv møter ikke egenskapene overfor kravene som er satt for asynkron notification. Derfor er det en alternativ måte å interrupte en tråd basert på ATC. Det er likt som Ada på den måte at delen av koden som kan motta en ATC forespørsel må defineres, men RT JAva skiller seg ut på to viktige punkter:

- Real-Time Java har java-exceptions integrert, mens Ada benytter select-statements og entry-handling.
- Real-Time Java krever at alle metoder indikerer at de er klare for at et ATC (Asyncronous Transfer of control) kan skje. I Ada er ATC tillatt by default.

Denne modellen (Real-Time Java ATC-modellen) fører sammen Java's exception handling og en utviding av trådinterrupts. Når en RT tråd interruptes vil et asynkront exception leveres til tråden. Her skiller det seg fra vanlig Java hvor tråden selv regelmessig må sjekke for interrupts.

Problem: hva hvis A kaller metode B og B kaller metode C. Hvis det oppstår et exception i C og C ikke har noen exception-handler propageres exceptionet opp i hierkiet; dvs. til B. Hvis B heller ikke har en exception-handler propageres exceptionet videre opp til A. Hvordan skal man da kunne hoppe tilbake til C og fortsette hvor exceptionet ble kastet ? Dette er et svært vanskelig programmeringsproblem, og man vil i mange tilfeller ende opp med en kode som fortsetter i A hvor exceptionet ble håndtert.

RT Java's løsning til dette er å kreve at alle metoder som er klare for å motta et asynkront exception setter dette exceptionet i sin *throws* liste. Disse metodene kalles **AI-methods** (Asynchronously Interruptable). Metoder som ikke settes på denne lista vil kjøre som normalt. Trådene vi derfor ikke avbrytes før de kjører en funksjon som er en AI-metode.

For at ATC skal håndteres på en trygg måte krever RT Java at:

- ATC blir utsatt under kjøring/utføring av synkroniserte metoder eller statiske intitialiseringer. Dette er for å sørge for at ingen data blir satt til konsistente tilsander. RT Java kaller disse kode-seksjonene som ikke er AI-metorer for ATC-deferred sections (ATC-utsatte metoder).
- En ATC kan kun bli håndtert i kode som er ATC-deferred. Dette er for å sikre at en håndtering av et ATC ikke kan bli forstyrret av et annet ATC.

Bruk av ATC fasiliteter krever tre aktiviteter:

- Deklarere et exception som skal sjekkes.
- Identifisere metodene som kan interruptes(og legge de til i throws lista).
- Signalisere (sende ut) et exception til andre tråder

7.7.3 Real-Time Java and atomic actions

Hvordan implementere atomic actions med forward recovery ved bruk av RT Java og ATC ? Er du jævlig keen på å finne ut dette bør du sjekke ut: Hinsides eksempel fra side 275-277.

8 Resource Control

Tråder kan ofte dele ressurser med andre tråder. Det kan dreie seg om filer, eksterne enheter osv. Resource control handler om å styre tilgangen til slike delte ressurser. Hvis en passiv kontroll agent er nok for å styre tilgangen sier vi at den er **protected** eller **synchronized**. Hvis en aktiv kontrollagent kreves, sier vi at den er en **server**.

8.1 Resource control and atomic actions

Tråder må kommunisere og synkronisere for å få allokert resursser, men dette trenger man ikke gjøre som en atomic action. Ved å innføre en resource-controller er man i stand til å sørge for global aksess og aksept for endringer på data. Koden som skal til for at en bestemt tråd kommuniserer med ressurskontrolleren bør være en atomic action så den ikke kan avbrytes midt i en allokering eller mens tråden frigjør en ressurs.

8.2 Resource management

Modularitetskrav gjør at resursser må være inkapslet og aksessert gjennom prosedyre-grensesnitt. I Ada bør man f.eks. benytte *packages*. I C/RT-POSIX og i Java's synkroniseringsklasser kan man ha beskyttede variabler i Monitorer.

8.3 Expressive power and ease of use

Som resource-manager kan man benytte:

- Monitorer: Benytter condition synchronization.
- Servere: Bruker meldingsbasert grensesnitt.
- Protected resources (beskyttede resursser): beskyttede objecter.

Expressive power: muligheten for et språk å utrykke nødvendige begrensninger på synkronisering.

Ease of use: Enkelheten/brukervennligheten i å uttrykke hver av disse synkroniseringsbegrensningene. Enkelheten i hvordan den tillater begrensninger å kombineres til mer komplekse synkroniseringsordninger.

Informasjonen som kreves for å uttrykke disse begrensningene kan kategoriseres:

- Service-requestens type
- Rekkefølgen som requestene har kommet i

- Tilstanden til serveren og hvor mange objecter den er server for
- Requestens parametre.
- Klientens prioritet

Det er to måter å sette begrensninger på tilgangen til en service: **Conditional wait:** alle requester aksepteres, men alle som ikke kan taes umiddelbart blir satt i kø. Dette kan f.eks gjøres med monitorer. **Avoidance** er den andre måten: Requester aksepteres kun hvis de kan håndteres. Dette bestemmes av en guard.

8.3.1 Request type

Informasjon om hvilken type handling/service som ønskes kan benyttes til å gi preferanse for en handling fremfor en annen. I Ada kan forskjellige requesttyper representeres med forskjellige entries i serveren. Disse requestene kan så queues opp, og man kan etter gi prioritet til de forskjellige requestene ved f.eks å telle opp antallet i en entry.

8.3.2 Request order

Rekkefølgen requester ankommer med kan benyttes til å gi rekkefølge for utførelse. F.eks. monitorer som som regel benytter FIFO-kø. I Ada kan man benytte FIFO på utestående requester av samme type.

8.3.3 Server state

Noen operasjoner kan være tillatt kun når serveren og det objektet den er tilegnet til er i en gitt tilstand. F.eks: en ressurs kan kun allokeres når den er frigjort, eller at et element kan settes inn i et buffer kun hvis det er en ledig plass etc.

8.3.4 Request parameters

Informasjon om handlingen som skal utføres av serveren kan ligge i parametrene som sendes med requesten. Det kan f.eks være størrelsen på requesten (størrelse på en array og lignende). Hvis ikke nok ressurser er tilgjengelig for å utføre handlingen vil requesten suspenderes, men når ressurser på nytt frigjøres vil alle klienter våkne for å se om deres request nå kan utføres.

Resource allocation and Ada - an example

En måte å takle problemet på i Ada er å gi en entryfamilie til hver type request til resource controlleren. Da vil hver tillate parameter ha en egen plass i familien(være en egen entry). Dette er bare passende om parameterne er diskrete typer. I køen velges requesten som er størst først til enhver tid. En tråd ha en to-stegs kommunikasjon med manageren for å få en ressurs. Først er det en "sign-in" request(legger inn størrelsen på requesten i køen) og så er det en "allocate" request hvor den får utdelt ressursen. Mellom disse stegene kan klienten måtte vente.

Double interactions and atomic actions

Så i forrige delkapittel at to-stegs kommunikasjon med manager var nødvendig. For å lage en resource control til å stole på må atomic actions brukes til å implementere denne strukturen. Dette er ikke mulig i Ada fordi klienten mellom de to kallene(sign-in og allocate) kan være i en tilstand som kan observeres utenfor den tilhørende atomic action. Dette kan skje ved at annen tråd avbryter klienten mens den venter. HVis serveren antar at klienten vil komme med allocate-kallet vil manageren være i en deadlock. Hvis manageren har en timeout på klientene kan den avbryte en klient som er treg.

8.3.5 Requester priority

Siste kriterie for evaluering av synkroniseringsprimitives for resorge managment involverer klient prioritet. Hvis mange tråder kan kjøre vil de deles inn etter prioritet. Manageren har dog ingen styring over suspenderte tråder som venter på ressurser. Derfor må manageren dele inn sine operasjoner i en rekkefølge som overholder prioteten på trådene.

8.4 The requeue facility

En måte å øke brukervennligheten til avoidance synchronization er å legge til requeue fasilitet, Ada har dette innebygd. Requeue vil si å flytte en tråd som har vært gjennom en guard/barriere bak en annen guard. Ada tillater requeue mellom task entries og protected object entries. Reque kan være til ny entry eller samme. Hovedsaklig brukes requeue til å sende den kallende tråden til en annen entry i samme enhet som den allerede var i.

8.4.1 Semantics of requeue

Problem: Når en requeue er fra et beskyttet objekt til et annet objekt vil mutual exclusion hos opprinnelig objekt gis opp så fort tråden er satt i køen på nytt. Andre tråder som venter på å aksessere det første objektet vil da kunne gjøre det. Men når reque er internt i objektet(en entry til en annen) vil mutual exclusion beholdes. Entryen som det skal byttes til ved en reque må enten ha ingen parametre eller helt like parametre som entryen tråden skal flyttes fra.

Vanligvis når en tråd er i en entry kø blir den der til den har fått hjelp hvis den ikke blir avbrutt asynkront. Når en tråd aksepteres vil aborten utsettes til den kommer ut av entryen igjen. HVa skjer med reque?? To mulige måter å se det på:

- Siden første kall er akseptert bør aborten forbli utsatt til tråden eller det beskyttede objektet kan være sikker på at det andre kallet er der.
- Den andre måten er at reque putter tilbake den kallende tråden til en entry kø slik at abort igjen bør være mulig.

Begge måter kan programmeres i Ada. Det avgjørende poenget er om det beskyttede objektet som har satt tråden i reque forventer at den er der når guarden åpnes. Hvis riktig oppførsel krever at tråden er der da kan ikke med abort benyttes(metode 2).

8.4.2 Requeing to other entries

Vanligvis benyttes reque til å flytte tråden foran samme entry på nytt, men noen ganger kan det benyttes ved å flytte en tråd til en annen entry. Vi ser for oss en router som kan velge mellom tre kommunikasjonslinjer(A,B og C). A er hovedlinje, men når den blir overbelastet benyttes linje B. Tilsvarende benyttes linje C når B er overbelastet. Hver linje overvåkes av en servertråd og en beskyttet enhet er grensesnitt til routeren. Den bestemmer hvilken linje som skal benyttes og sender da request til riktig server. Antar at reque benyttes til å flytte en tråd til riktig linje.

8.5 Asymmetric naming and security

I språk med symmetrisk navngiving vil en server tråd(eller beskyttet ressurs) alltid vite navnet til klienten. Dette er også situasjon med indirekte navngiving. Asymmetrisk navngiving derimot gjør at serveren ikke vet identiteten til klienten. Fordelen når serveren ikke vet identiteten er at generelle kommunikasjonsmetoder kan designes,

men det kan gi dårlig beskyttelse for bruk av ressurser. Serveren kan særlig være interessert i klient ID slik at:

- En request kan avvises som følge av at deadlock skal hindres eller pga fairness(rettferdig ressursfordeling).
- Den kan garantere at en ressurs slippes av klienten som fikk den.

8.6 Resource usage

Når to konkurrerende tråder trenger en ressurs er vanlig oppførsel at de ber om den(request) og venter om nødvendig. Deretter benytter de ressursen før de frigir den. Ressursen kan brukes på to måter(modier). Shared access vil si at to tråder kan benytte den parallellt(f.eks read-only fil). Exclusive access krever at en tråd aksesserer den om gangen(fysisk ressurs som en printer f.eks). Noen ressurser kan brukes i begge modier.

Siden tråder kan blokkeres når de ber om ressurser er det viktig at de ikke ber om dem før de virkelig trenger den. De bør i tillegg slippe de så fort som mulig. Hvis ikke vil systemets ytelse gå ned. Dessverre er det slik at hvis tråder slipper ressurser tidlig og så feiler kan ressursen ha feilaktig data. Derfor må to-fase ressursbruken modifiseres. Ressursene kan nå ikke slippes før en action er ferdig. Enhver recovery prosedyre må slippe ressursen når den er suksessfull eller slippe den sånn den var opprinnelig(omgjøre alt ressursen har gjort). Det kan løses med forward recover og exception handling. Q: HVA GJØR MAN OM RESSURSEN ER EN PRINTER OG HAR PRINTA ET ARK?? :S NPC problem!!!

8.7 Deadlock

Ser for oss to tråder T1 og T2 som begge vil bruke ressursene R1 og R2. Hvis T1 holder på R1 og trenger R2 for å fortsette, men T2 allerede holder på denne og trenger R1 har vi en deadlock. En deadlock betyr at to tråder utsettes i det uendelige. Livelock er tidligere nevnt og betyr at trådene fortsetter å utføre sin del, men er låst i evig sykel(og gjør unyttig arbeid). Husker også starvation som vil si at en tråd alltid blokkeres av andre tråder i kampen om en ressurs. Liveness betydde at i parallelle systemer at hvis en tråd venter på en ressurs vil den før eller senere få den. Det er generellt fire betingelser som må være oppfyllt for at en deadlock skal oppstå:

- Mutual exclusion - Kun en tråd kan bruke ressursen om gangen
- Hold and wait - Eksisterer tråder som holder ressurser og venter på andre

- No preemption - Ressurs kan kun slippes frivillig av tråd.
- Circular wait - Sirkulær lenke av tråder må eksistere slik at hver tråd holder en ressurs som neste i sykelen vil ha.

For å finne ut om et system har deadlock/livelock kan man benytte FSP. Dette er høyst eksamensrelevant og FSP syntaks bør repeteres.

9 ch9

10 ch10

11 Scheduling real-time systems

I parallellprogrammering er det ikke nødvendig å vite i akkurat hvilken rekkefølge ting skjer(hvilken tråd som får gjøre hva når). Så lenge putput er likt uavhengig av rekkefølge er programmet godt nok. Men i RT systemer kan det være krav til at noen tråder må være ferdig fortere enn andre. Derfor vil vi i RT legge noen begrensninger på den ikke-deterministiske oppførselen. Dette er SCHEDULING. Generelt gir et scheduling scheme to features:

- En algoritme for å fordele systemressursene(spesielt CPUene)
- Måte å predikere worst-case oppførsel når algoritmen benyttes.

Et scheduling scheme kan være statisk som vil si at prediksjonene foretas før utførelsen eller dynamiske som vil si at run-time bestemmelser brukes. Her ser vi først og fremst på statiske.

11.1 The cyclic executive approach

Med fast sett av tråder som gjør noe periodisk kan en komplett schedule(plan) lages slik at trådene benytter korrekt tid. En cyclic executive er en tabell av prosedyrekall hvor hver prosedyre representerer en del av koden til en tråd. Hele tabellen kalles og en **major cycle**. En slik består vanligvis av **minor cycles** med fast lengde(fixed i tid). Har noen egenskaper ved denne fremgangsmåten:

- Ingen faktiske tråder eksisterer ved run-time. Hver minor cycle er kun sekvens av prosedyrekall.
- Prosedyrene deler adresserom og kan oveføre data til hverandre uten å beskytte den(semaforer etc) siden parallell aksess er umulig.
- Alle trådperioder må være multipel av "the minor cycle".

Den siste egenskapen er hovedulempen med executive cycle. I tillegg er det vanskelig å implementere the cyclic executive. Det finnes og noen andre store ulemper. Derfor brukes ikke fremgangsmåten for større programmer. Da brukes task-based scheduling. For små programmer med periodisk oppførsel er det en grei fremgangsmåte og det krever ingen scheduletest siden skjemaet er bevis i seg selv.

11.2 Task-based scheduling

Alternativ fremgangsmåte er å la notasjon med trådene i run-time brukes. Da støttes utførelsen av en tråd direkte og det er styring av hvilke tråder som kjører når som er viktig. Da er hver tråd i en av tre tilstander så lenge de ikke kommuniserer internt. De kan være runnable(klare til og utføres), de kan være suspendert å vente på en timing event(passende for periodiske tråder) eller de kan være suspenderte, men vente på en ikke periodisk-event.

11.2.1 Scheduling approaches

Ser på tre fremgangsmåter for scheduling:

- **Fixed-Priority Scheduling(FPS)** - MEst vanlige. HVer tråd har fast statisk prioritet som regnest ut før run-time og trådene kjører etter denne prioriteten.
- **Earliest Deadline First(EDF) scheduling** - Tråd kjøres etter rekkefølgen på hvem som må bli fortest ferdig til enhver tid. Tid til deadline ikke alltid lett å bestemme, men generellt regnes det ut i run-time så det er dynamisk.
- **Value-Based scheduling(VBS)** - Hvis et system kan overbelastes holder det ikke med statiske prioriteringer eller deadlines. Da kreves mer **adaptive** skjema. Da gis en verdi til hver tråd og en algoritme regner ut hvilken tråd som skal utføres neste gang.

Se på foiler fra Mladen for mer om dette.

11.2.2 Scheduling characteristics

En schedulability test er definert til å være tilstrekkelig(sufficient) hvis positivt resultat garanterer at alle deadliner oppfylles. En test er nødvendig hvis feil i testen medfører at en deadline ikke overholdes. En slik test er vanligvis relatert til worst-case oppførselen til applikasjonen. En test sies å være **sustainable** hvis den korrekt predikerer at et scheduable system vil forbli det når noen operasjonsparametere endres.

11.2.3 Preemption and non-preemption

Med prioritetsbasert scheduling kan man komme i en situasjon hvor en høy-prioritets tråd slippes mens en med lavere prioritet utfører sin oppgave. I et **preemptive** scheme vil tråden med høy prioritet få utføre sin kode umiddelbart. I et **non-preemptive** scheme

vil tråden med lav prioritet gjøre seg ferdig. **Deferred preemptio(cooperative dispatching)** scheme vil la tråden med lav prioritet få fortsette en viss tid.

11.2.4 Simple task model

Et eksempel som skal brukes senere i kapittelet og.. Verdt å merke seg fra dette kapitellet er begrepet **critical instant**, som er et tilfelle hvor alle tasks blir satt i gang samtidig (kanskje husker du eksemplet fra timen når 3 studenter ble bedt om å løpe ut døren fra forelesningssalen på akkurat samme tidspunkt).

11.3 Fixed-priority scheduling(FPS)

Rate monotonic prioritet vil si at trådene får prioritet ut i fra periodetid. De med kortest prioritet får høyest prioritet. En prioritet på 1 er for øvrg lavere enn en prioritet på 2 her.

11.4 Utilization-based schedulability tests for FPS

Finnes flere måter å sjekke om et set av tasks er schedulable på. En måte er å tegne timeline, mens en annen måte er å sette opp matematiske uttrykk. En enkel, men ikke helt eksakt test for FPS er:

$$\sum_{i=1}^{N}(\frac{C_i}{T_i}) \leq N(2^{1/N} - 1) \tag{2}$$

Her er N antall tråder og hvis ulikheten er sann vil alle N tråder nå sine deadlines. Ci er beregningstid og Ti er periodetid for tråd i.

11.4.1 Improved utilization-based tests for FPS

I stedet for å la N være antall tasks i programmet, kan N defineres til å være antall task-families. En task-familie defineres ved at de oppstår med multipler av en felles periode. 3 tasks med periodetid på hhv. 8, 16 og 64 er en familie. En utilizationtest utført på disse tre taskene kan derfor være å benytte N = 1 i ligning 2.

En annen utilizationtest, som skal være bedre, ble utviklet av Bini:

$$\Pi_{i=1}^{N}(\frac{C_i}{T_i} + 1) \leq 2 \tag{3}$$

11.5 Response time analysis(RTA) for FPS

Testene hittil har to ulemper. De er ikke eksakte og de er ikke passende for en mer generell task modell. En annen test vil derfor presenteres. Den går over to steg. I det første steget predikeres worst-case response time for hver tråd. Så sammenlignes disse verdiene med tidsfristene. Derfor må hver tråd analyseres inviduelt. For den tråden med høyest prioritet vil worst-case responstid være lik sin egen nødvendig beregningstid(R=C). Andre tråder vil oppleve **interference** fra høyere prioritetstråd. En enkel måte å bestemme prioritet på, er ved å la tasks med lav periode få høy prioritet. For generell tråd i hvor forrige tråd har høyere prioritet er worst-case response time

$$R_i = C_i + I_i \tag{4}$$

I_i er max interference en tråd kan oppleve. Dette oppstår når alle tråder med høyere prioritet slippes fri likt. Så er det en haug av ligninger og tull som er (u)mulig å pugge:

$$I_i = \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil \cdot C_j \tag{5}$$

Merk at dette gir et uttrykk med R_i både på høyre og venstre side av lingningen:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil \cdot C_j \tag{6}$$

11.6 Sporadic and aperiodic tasks

Utvider nå modell fra 11.2.4 for å takle sporadiske og aperiodiske tråder. I stedet for å definere en periodetid defineres det en minimumstid mellom to påfølgende iterasjoner for tråden. I tillegg må definsjon på deadline modifiseres. Den forrige modellen antok at D = T(deadline lik periodetid). Dette fungerer ikke lenger siden en sporadisk oppgave plutselig må kunne håndtere warnings og errorhandling routines. Selv om slike feil inntrer sjeldent må de løses raskt når de først inntreffer. Tillater nå D å være mindre enn T.

11.6.1 Hard and soft tasks

For sporadiske tråder er det stor forskjell på worst-case og average tid. Derfor får vi dårlig ytelse om det alltid regnes med worst-case tid. Derfor kan man ofte benytte average tid i testing, men følgende regler må alltid overholdes:

- Alle tråder bør være schedulable med average utføringstid og average arrival rate(gjennomsnittlig tid mellom hver gang en tråd melder seg for scheduler).
- Alle harde RT tråder bør være schedulable med worst-case utføringstid og worst-case arrival rate.

Konsekvens av første regel er at det kan være situasjoner hvor det ikke er mulig å overholde alle tidsfrister. Dette kalles da **transient overload**.

11.6.2 Aperiodic tasks and fixed-priority executing-time servers

En enkel måte å schedule aperiodiske tråder er å gi de prioritet under harde tråder. Dette kan føre til at softe tråder mister sine tidsfrister pga lav prioritet. En **server** kan benyttes for å hindre dette slik at alle harde tråder for kjøre med en gang, men så prioriteres de softe trådene før de aperiodiske. Har to slike hovedmetoder, **Deferrable Server(DS)** og **Sporadic Server(SS)**.

11.7 Task systems with D < T

Hvordan utføre utilization test når deadline er mindre enn perioden. Det kan f.eks. være feiltoleranse: en feil oppstår hvert 2 minutt, men må håndteres og løses ila. 3 sek. Husker at kort periode T gir høy prioritet P. Nå kan vi ha T ulik D, og vi lar derfor P være bestemt av deadline, D. Lav D gir høy prioritet P. Dette kaller vi deadline monotonic priority ordering ($D_i < D_j \Rightarrow P_i > P_j$).

11.7.1 Proof that DMPO is optimal

Et bevis er et bevis..

11.8 Task interactions and blocking

Med en gang tråder må synkroniseres, ha tilgang til samme ressurser osv. blir analysen vanskeligere. Det blir vanskeliger å finne og å definere en worst case tid for de forskjellige taskene. En task t1 kan f.eks bli stående i vente på en ressurs fra en annen tråd t2, og dette vil føre til at kjøretiden til t1 er avhengig av t2. Vi kaller det **priority inversion** når en høy-prioritetstråd må vente på en tråd med lavere prioritet (dette kan f.eks. oppstå fordi en høypri.tråd trenger kalkuleringsresulater fra en lavpri.tråd). Når fenomenet **priority inversion** oppstår sier vi at høypri.tråden er **blocked**. Ønsker vi å

teste et system for schedulability, må vi vite at blockingen er bundet, målbar og liten.

Fra dette kapitlet kan det være lurt å se på timelinene som er vist i boken. Det er nok svært eksamensrelevant å kunne tegne slike timeliner når man har et sett av tasks med ulik prioritet, kritiske regioner som kan låses, og forskjellige releasetider.

Maksimal blocking time er den maksimale tiden en tråd/task kan bli blokkert, og kan uttrykkes ved formelen:

$$B_i = \sum_{k=1}^{K} usage(k, i)C(k) \tag{7}$$

Hvor K er antall kritiske seksjoner, usage(k,i) = 1 hvis resurssen k er brukt av minst en task med høyere eller lik prioritet som P_i (usage(k,i) er 0 ellers). C(k) er worst case kjøre tid av den kritiske seksjonen k. En kritisk seksjon kan f.eks være en monitor, mutex, etc.

11.8.1 Response time calculations and blocking

Gitt B, så kan ligning 4 utvides til

$$R = C + B + I \tag{8}$$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil \cdot C_j \tag{9}$$

Dette gir et mer presist tall for responstiden R.

11.9 Priority ceiling protocols

Original ceiling priority protocol og **immediate ceiling priority protocol**. Felles for begge er:

- En høypri.tråd kan blokkeres maksimum en gang av lavpri.tråder ila kjøreperioden
- Deadlocks unngås
- Transitive blocking unngås (?)
- Mutual exclusive tilgang til ressurser er sikret.

Ceiling protokollen forsikrer oss om at: hvis beskyttet ressurs er låst av en task Ta, og dette kan føre til at høypri.tråden b blir blokkert, da kan ingen ressurs som kan blokkere b tillates å bli låst av anne tråd enn a. **Original ceiling priority protocol**(OCPP) består av følgende regler:

- Hver task har statisk prioritet
- Hver ressurs har statisk ceiling value definert; denne verdien er lik max.prioritet av trådene som benytter denne ressursen. (hvis vi har 3 tasks med $P_a > P_b > P_c$ som benytter ressurs R, vil ceiling verdien være P_a).
- En task kan kun låse en ressurs hvis dens dynamiske prioritet er høyere en ceiling verdien til den låste ressursen.

11.9.1 Immediate ceiling priority protocol

ICPP består av følgende regler:

- Hver task/tråd har statisk prioritet
- Hver ressurs har statisk ceiling verdi definert (definert med lik verdi som i OCPP)
- En task har dynamisk prioritet som er lik sin egen statiske prioritet pluss maks ceiling verdi av de ressursene denne tasken har låst

Oppsummert kan OCPP og ICPP sammenlignes:

- ICPP er enklere å implementere siden blocking-forhold ikke trengs overvåkes.
- ICPP fører til færre context switches as blocking is prior to first execution
- ICPP krever flere prioritetsforandringer. OCPP endrer prioritet kun ved blokkering.

11.9.2 Ceiling protocols, mutual exclusion and deadlock

Ceiling protocol er en måte å unngå deadlocks på.

11.10 An extendible task model for FPS

Ønsker i dette kapitlet å oppnå en generell algoritme for å fastsette prioriteter for tasks. Ønsker en algoritme uten restriksjoner som f.eks at D må være lik T etc.

11.10.1 Release jitter

Hva hvis tasks oppstår uperiodisk? Vi definerer **release jitter** som maksimal variasjon i en tasks release (start). Dette fører til en ny

utvidelse av ligning 4

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{R_i + J_j}{T_j} \rceil \cdot C_j \tag{10}$$

11.10.2 Arbitrary deadlines

Hva hvis deadline D (følgelig også responstid, R) kan være større en periodetiden T? Når deadline er mindre enn T er det nok å se på ett tilfelle, men når deadlinen blir større enn T må man se på et større bilde... Man antar at en task starter ETTER at en annen task av samme type har fullført (F.eks en sensor avlesning og kalkulering: kalkuleringen for tid a må være ferdig før man kalkulerer for data fra tid b).

Så kommer analysen: se på side 392 i boka, og evt ta gode notater på en god dag.

11.10.3 Cooperative scheduling

Frem til nå har vi sett på analyse med preemptive scheduling. Nå kan vi se på et annet tilfelle: nemelig deferred preemption, dvs. tråder tillates å kjøre en hvis tid før de avbrytes. Å benytte deffered preemption fører med seg en rekke fordeler. Ved immediate ceiling protocol er blokkeringen ikke kumulativ (Det vil si at en task ikke både kan blokkeres av en task og en kernel-routine). Cooperative scheduling ser på denne ikke-kumulative egenskapen: La B_{max} være maksimal mulige blokkeringstid in systemet. Applikasjonskoden deles opp i non-preemtive blokker (dvs. at de ikke blir avbrutt i kjøretiden). Ved slutten av kjøretiden til hver av disse blokkene gir applikasjonskoden en de-scheduling request til kjernen (kernel). Hvis en høypri.blokk er kjørbar vil denne få kjøre, hvis ikke vil vi fortsette inn i en ny non-preemtive blokk. Altså: en tråd deles inn i (non-preemtive) blokker, og ved slutt får kjærnen mulighet til å switche task. Denne metoden sikrer mutual exclusion, men krever nøye gjennomtenkte plasseringer av de-schedule-kall.

Ved å bruke deferred schedulability øker vi schedulabiliteten til systemet.

11.10.4 Fault tolerance

Både forward og backward error recovering vil føre til økning i kjøretid. I sanntidssystemer bør deadlines møtes selv når en hvis grad av feilhåndtering blir nødvendig. Dette tillatte nivået av feiltoleranse kaller vi the **fault model**. La oss definere C_i^f som ekstra

kalkuleringstid som følge av feilhåndtering. Da kan vi utvide likning 4 enda et steg:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{R_i + J_j}{T_j} \rceil \cdot C_j + max_{k \in hep(i)} C_k^f \qquad (11)$$

hvor $hep(i)$ er definert som det sett va indekser som svare til tasks med prioritet høyere eller lik prioriteten til task i. Man kan i tillegg inkludere maksimalt antall tillatte feil F og man får følgende likning:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{R_i + J_j}{T_j} \rceil \cdot C_j + max_{k \in hep(i)} FC_k^f \qquad (12)$$

Eventuelt kan man se på minimumsintervallet som feil kan oppstå med:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{R_i + J_j}{T_j} \rceil \cdot C_j + max_{k \in hep(i)} (\lceil \frac{R_i}{T_f} \rceil C_k^f) \quad (13)$$

Hvor T_f er minimum tid mellom hvert tilfelle av feil.

11.10.5 Introducing offsets

Det kan i mange tilfeller være hensiktsmessig å ikke la alle tasks releases samtidig (dvs. unngå lik critical instant). Ved å la tasks starte ulikt (introdusere offsets) kan man i mange tilfeller øke schedulabiliteten til systemet, siden dette kan benyttes til å unngå interferens mellom tråder/tasks. Det skal dog sies at å finne optimal offset anses som et NP-hard problem, og er slett ikke bare-bare. En måte å analysere schedulabilitet av systemer med offset er å slå sammen to eller flere tasks til en task. Ta f.eks b og c med T hhv. 20 og 20, D lik 10 og 12, C lik 4 og 4. Disse to taskene kan slås sammen til n med T = 10, D = 10, C = 4. En analyse kan så benyttes på n og resten av systemet.

11.10.6 Other characteristics

Kan også ha systemer dere kun N av M deadlines kreves overholdt etc.

11.10.7 Priority assignment

Frem til nå har vi sett på to metoder å gi prioritet på: rate og deadline monotonic priority ordering.

11.10.8 Insufficient priorities

Kan i mange tilfeller ha flere tasks med samme prioritet. Knep: gi alle tasks ulik prioritet. Gjør dette systemet schedulable. Gå så gjennom taskenes prioritet fra bunn til topp, og grupper sammen de taskene som skal ha samme prioritet frem til schedulabiliteten ødelegges (ikke ta med den siste). Lag så en ny gruppe.

11.10.9 Execution-time servers

11.11 Earliest deadline first (EDF) scheduling

Et annet alternativ til FPS (fixed priority scheduling).

11.11.1 Utilization-based schedulability for EDF

Hvis man benytter EDF (første deadline) kan følgende test benyttes (her antas D = T):

$$\sum_{i=1}^{N}(\frac{C_i}{T_i}) \leq 1 \tag{14}$$

EDF er på mange måter bedre enn FSP; hvis et sett tasks er schedulable for FSP er det samme settet garantert schedulable ved bruk av EDF (dette gjelder ikke nødvendigvis motsatt). MEN FSP er fortsatt det mest brukte, og det kommer av de fordeler FSP har fremfor EDF:

- FSP er enklere å implementere (siden prioriteten er statisk). EDF er dynamisk, mer kompleks og krever derfor mer kjøretid.
- Deadline er ikke alltid det viktigste, og det er ikke alltid slik at deadline bør benyttes for å gi prioritet.
- Hvis man ikke rekker deadlines kan EDF oppføre seg uforutsigbart, og man har i EDF en tendens til at dette fører til dominoeffekt (dvs. flere tasks ikke rekker sine deadlines). I FSP er dette mer forutsigbart.
- Den enkle utilization-testen i 14 er for EDF både nødvendig og tilstrekkelig (necessary and sufficient). Tilsvarende test er tilstrekkelig (sufficient) for FSP. Altså kan høyere utilization oppnås for FSP.

11.11.2 Processor demand criteria for EDF

En av ulempene ved EDF er at worst-case responstid IKKE er når alle tasks startes ved critical instant. Processor demand criteria

(PDC) formuleres på følgende måte: Anta at et system startes ved tid lik 0. Anta videre at alle tasks ankommer med sin maksimumfrekvens til en hver tid i fremtiden (det vil si så lav T som mulig). Det er da mulig å kalkulere lasten på systemet, $h(t)$. $h(t)$ er jobben som må utføres av systemet før tiden t. Altså er $h(t)$ et mål på mengden av alle jobber som må utføres før tiden t (det er de jobbene som har absolutte deadlines før t). Verdien på $h(t)$ er gitt av:

$$h(t) = \sum_{i=1}^{N} \lfloor \frac{t + T_i - D_i}{T_i} \rfloor C_i \tag{15}$$

Kravet/kriteriet for schedulabilitet er at lasten ($h(t)$) aldri må overstige tiden t:

$$\forall t > 0, h(t) \leq t \tag{16}$$

I PDC-analysen tester vi dette kriteriet for et sett av punktprøver. Antall punktprøver bestemmes av følgende faktorer:

- Kun t-verdier som svarer til task-deadlines sjekkes.
- Vi finnes en øvre grense for t-verdier som skal sjekkes som vi kaller L. Dette betyr at et unschedulable-system vil ha $h(t) > t$ for en eller annen $t < L$.

Den øvre grensen L bestemmes av følgende formler:

$$L_a = max\{D_1, \ldots, D_N, \frac{\sum_{i=1}^{N}(T_i - D_i)C_i/T_i}{1 - U}\} \tag{17}$$

Sammen med

$$\omega^0 = \sum_{i=1}^{N} C_i \tag{18a}$$

$$\omega^{j+1} = \sum_{i=1}^{N} \lceil \frac{\omega^j}{T_i} \rceil C_i \tag{18b}$$

$$L_b = \omega^j | \omega^{j+1} = \omega^j \tag{18c}$$

$$\tag{18d}$$

Dette gir

$$L = min(L_a, L_b) \tag{19}$$

11.11.3 The QPA test

Quick Processor-demand Analysis. For ikke-trivielle systemer, kan L være stor, og antall deadlines mellomm 0 og L mange. Da er det hendig med en måte å kunne redusere antall punktprøver i PDC på. Litt satt på spissen, kan man si at QPA gir oss en måte å redusere antall punktprøver på.

I stedet for å sjekke for alle deadlines fra 0 til L, gjør QPA det på følgende måte: Start ved L og beveg deg tilbake i tidsdomenet. Sjekk kun et nødvendig subset av deadlines. La $h(L) = s$. hvis $s > L$ har vi et ikke-schedulable system. Hvis $s \leq L$, da er $h(t) < t$ for alle $t \in [s \dots L]$. Vi trenger derfor ikke punktprøve flere t-verdier i intervallet fra s til L.

11.11.4 Blocking and EDF

Vi vet av FSP kan lide av *priority inversion*[5]. For EDF har vi noe som kalles *deadline inversion*. Deadline inversion får vi når en ressurs er låst av en annen task med lengre deadline. Dette er løst på samme måte som for FSP; med inheritance og ceiling protocols (disse er dog mer komplisert for EDF enn FSP).

Stack Resource Policy (SRP) er et scheme for EDF. I SRP gis hver task et preemption level. Kortere relativ deadline gir høyere preemption level. Ved kjøretid får ressurser ceiling values basert på maksimum preemption level av taskene som benytter ressursen ($CV_r = max(PL_i), i \in [tasks - that - uses - resource - r]$, hvor CV_r er ressursens ceiling value, PL_i er task i's preemption level. Å benytte denne protokollen svarer til FSP's ICPP - deadlocks unngås, kun single-block etc.

11.11.5 Aperiodic tasks and EDF execution-time servers

blablabla

11.12 Dynamic systems and online analysis

FSP og EDF er flotte greier. EDF gir "optimal" scheduling, men med en gang man får overload på systemet risikerer man å få dominoeffekt (det kan f.eks. være en task som holder opp en ressurs i for lang tid, dette fører til at den neste tasken mister sin deadline osv.). For å lage et mer robust scheme (et scheme som motstår denne dominoeffekten), kan man lage dynamiske skjema. Dette er schemes som beregnes "live". Et klassisk online scheme har to mekanismer:

[5] En høypri.tråd må vente på en tråd med lavere prioritet

- En adgangskontroll for hvilke tasks som skal få lov til å kjempe om prosessorkraft.
- En EDF rutine for å bestemme hvilke tasks som skal få tilgang til prosessorkraften.

En ideell adgangskontroll fungerer slik at prosessoren ikke blir overbelastet med arbeid, og EDF-rutinene fungerer effektivt. Adgangskontrollen gir alle tasks en verdi for relativ viktighet, og det er denne verdien som bestemmer om en task skal få være med på å kjempe om prosesorkraft. Disse verdien klasifiseres typisk på følgende måte:

- Static: tasken har alltid samme verdi når den blir sluppet (released).
- Dynamic: Taskens verdi kalkuleres når tasken slippes (is released), siden verdien kan være avhenging av omgivelsenes og systemets tilstand.
- Adaptive: Siden systemet er dynamisk, vil verdien av tasken endres ila kjøretiden.

Online kalulering av prioriteter benyttes i systemer der kjøretid og ankomsttid ikke er kjent på forhånd. Dette kan være dynamiske soft real-time applikasjoner.

Begrepet **Best-effort** bruker vi om følgende situasjon: hvis et sett av tasks ikke er schedulable, dropper vi den tasken med lavest verdi og repeterer schedule-testen. Dette fungerer bra hvis man ser bort i fra den tiden det tar å utføre testen osv.

Hybrid systems er systemer som inneholder både harde og dynamiske komponenter. Blablabla. Bruker både FSP og EDF på disse systemene.

11.13 Worst-case execution time

Det er to måter å finne Worst-case execution time (WCET) på:

- Eksperimentelt: måle kjøretid. Problem: hvordan vet man at man har fått med WCET ?
- Analytisk: gjøre analyse på systemet. Problem: kreves en effektiv modell av prosessoren, taskene, ressursene etc.

Analysen gjøres i to deler. 1: del opp koden til taskene i en rettet graf av **basic blocks** (en basic block er straightline code). 2: Se på maskinkoden til disse basic blocks og bruk prosessormodellen til å beregne WCET.

Eksempel:

```
for i = 0...10
    if(cond)
        -- do something that has cost 100
    else
        -- do something that has cost 10
    end if;
end for;
```

Her ser man lett at WCET bør bli 10 (løkken) x 100 (if setningen) = 1000. Tar man i tillegg hensyn til kjøretiden selve løkken bruker (f.eks 5) ender man opp på 1005. Vet man derimot at cond vil oppfylles maks 3 ganger, kan man justere tiden ned til 3x100+7x10+5 = 375.

Har man derimot while-løkke, kan det bli vanskelig å beregne WCET offline (uten å faktisk kjøre programmet).

11.14 Multiprocessor scheduling

Har frem til nå sett på en-prosessor-systemer. Nå skal vi flytte fokus over til multiprosessorsystemer. Dette kan være flere datamaskiner som snakker over nettverk, felles bus eller felles minne.

11.14.1 Global or partitioned placement

Placement: mapping av tasks til prosessorene (hvilken prosessor skal ta hvilken task). To typer schemes er mulig: **global** og **partitioned**. Med partitioned bestemmes task-plasseringen på forhånd (før kjøretid). Med dynamisk global plassering bestemmes task-plasseringen i kjøretiden når taskene blir kjørbare.

Noen ligninger (util.-tester) som kanskje man bør se på...

11.14.2 Scheduling the network

Time Division Multiple Access (TDMA): Hver prosessor har en gitt periode på et gitt tidspunkt hvor tasks som kjører på denne prosessoren kan generere meldinger. **Control Area Network** (CAN): Hver melding gis en statisk prioritet og nettverket støtter prioritetsbasert __arbitration__ (megling/utveksling?) Vet ikke helt om jeg fattet meningen med dette delkapittelet.

11.14.3 Mutual exclusion on multiprocessor platforms

Når man har flere prosessorer med tilgang til samme minneområde; hvordan sikre at ingen aksesserer det samme minnet samtidig?! Må

på et eller annet vis ha en global lås, som alle har tilgang til. Når en task prøver å få tak i en lås som allerede er låst av en anne task, sier vi at den blir stående å spinne (**spinning)**. Vi har ikke ekvivalente protokoller for å unngå live- og deadlocks for multiprosessorsystemer som vi har for enkelprosessorsystemer. Vanskeligheten med globale låser fører til at man ønsker benytte låsfrie algoritmer. Multiple kopier av objekter kan være en løsning.

11.15 Scheduling for power-aware systems

11.16 Incorporating system overhead

References