Innhold

1 Innledning

Et sanntidssystem er et system hvor programvaren må forholde seg til "sann tid", dvs. tiden slik den løper i den virkelige verden. For programmer som du kjører på den jevne PC er slike kriterier tilstede bare i liten grad. "Så fort som mulig" er ikke et sanntids krav. Derimot krav som at "denne beregningen skal være ferdig før klokken ett", "denne beregningen skal være ferdig nøyaktig kl. ett", "denne beregningen skal starte kl. ett" eller "denne beregningen skal gjøres en gang hvert millisekund" er det.

Sanntidskrav til programvare er derimot veldig vanlig i reguleringstekniske systemer og i innvevde systemer. Typisk er tiden ofte en parameter i ligningene for regulatoren - hvis tiden imellom to beregninger av pådragsverdi ikke er som forventet, vil ikke regulatoren virke som forutsatt.

For sanntidssystemer gjelder vanligvis også andre, sekundære krav som går på sikkerhet, stabilitet, pålitelighet, forutsigbarhet, tilgjengelighet osv.

Denne teksten er delt i følgende hoveddeler:

- Andre kapittel er en kort introduksjon til hvordan vi kan skille imellom gode mekanismer, formalismer, kode, design etc. og dårlige.

- Tredje kapittel beskriver en standard grunnstruktur i implementasjonen av sanntidssystemer, med en del sentrale begreper.
- Fjerde kapittel gir en veldig kort oversikt over programmeringsspråk for sanntidsapplikasjoner.
- Til slutt, i kapittel fem, kommer en beskrivelse av tilstandsmaskiner - en hensiktsmessig formalisme for modellering og design av mange sanntidssystemer.

2 Kodekvalitet og vedlikeholdbarhet

Hva er forskjellen på god og dårlig programkode ? Hva gjør et design bedre enn det neste ?

Det finnes utallige teorier, metrikker, metoder, prinsipper, standarder osv. som svarer på slike spørsmål, men noe veldig klart og entydig bilde har vi ikke. La oss her sette *vedlikeholdbarhet* opp som kriteriet for et stykke programvare: Hvor lett er det å gjøre endringer, utvidelser eller å rette feil i systemet ? Vi kan måle dette ved f.eks. den forventede innsatsen som må til for å rette den neste feilen som blir funnet.

Dette vil si at vi ikke ser på ting som hvor mange feil det er i koden, i hvilken grad systemet er nyttig, har mange features eller oppfyller spec'en. Disse er også vanlige mål på progravarekvalitet. Imidlertid kommer de inn indirekte ved at i vedlikeholdbare systemer er feil enkle og rette, og endringer og utvidelser er billige å lage.

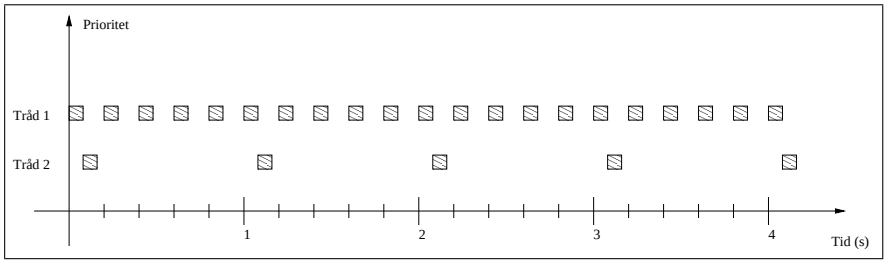
En temmelig allment akseptert "lov" for programvare er at kvaliteten forvitrer etter som tiden går. Om ikke dette skyldes at systemet vokser, eller at ad-hoc løsninger for utvidelser og feil-rettinger forkludrer det opprinnelige designet, så glemmer programmererne (eller blir byttet ut), verktøyene blir eldre (og dermed glemt, siden nye versjoner kommer), og kulturen — normene som systemet ble laget under opprinnelig — endrer seg.

Virkemiddel nr 1 for å bedre vedlikeholdbarheten er å dele koden i deler hvor hver del kan vedlikeholdes for seg selv, og er liten nok til å være oversiktlig. For at dette skal virke, imidlertid, må;

- delen kunne vedlikeholdes uten å kjenne til resten av systemet.
- delen kunne brukes uten å kjenne den indre virkemåten.

Hvis disse to ikke er oppfyllt så får vi ikke de ønskede gevinstene av oppdelingen.

Denne oppdelingen foregår på alle nivåer. På laveste nivå bruker vi linjeskift, innrykk og blanke linjer for å gruppere statements som har med hverandre å gjøre. Funksjoner, prosedyrer og metoder lages ikke først og



Figur 1: CPU-fordeling for pumpestyringen

fremst for å oppnå gjenbruk, men for å gruppere statements bak grensesnittet som er funksjonsnavnet og parameterlisten. Likedan for objekter som har et sett av samhørende metoder, og andre former for moduler og biblioteker.

Tenk på dette når du programmerer. Kan du, når du ser på en skjermside kode, argumentere for at denne koden virker, uten å forutsette ting om resten av systemet ? Kan du si om en funksjon at den alltid gjør det den var ment til å gjøre, uavhengig av sammenhengen den kalles i ? Og omvendt; kan du, ved å se på et funksjonskall, gjette riktig og lett hva funksjonen gjør, uten å måtte se på selve funksjonen ? — Og tilsvarende for objekter, moduler, biblioteker etc.

Virkemiddel nr. 2 for å oppnå vedlikeholdbarhet, er å velge/bygge riktig *abstraksjon* for ditt problem, og både tråder og synkronisering, kommuniserende prosesser, og tilstandsmaskiner, som blir beskrevet i de følgende kapitlene, er eksempler på slike abstraksjoner.

3 Overordnet design av sanntidssystemer.

Det allment aksepterte design-mønsteret for å kunne møte sanntidskrav er at systemet deles i flere oppgaver, (**tråder**, **prosesser**, **aktiviteter**), som registreres hos en **scheduler**. Scheduleren har som oppgave å holde oversikten over alle trådene og bestemme hvilken av dem som skal få kjøre til ethvert tidspunkt.

Denne modellen er ikke spesiell for sanntidsapplikasjoner; Den er en del av alle moderne operativsystemer. Imidlertid er det stor variasjon i hvilke utvidelser av dette grunnleggende mønstret de forskjellige implementasjonene tilbyr. Noen er mer egnet for sanntidsapplikasjoner enn andre...

La oss ta som et eksempel en pumpe som skal fylle en tank med væske; Reguleringen av pumpen må skje med sampling 5 ganger i sekundet, mens avlesning av nivået i tanken og beregning av referansen til pumpen skjer bare en gang hvert sekund. Her vil vi ha to tråder i systemet, hver med egne krav til når de skal kjøres.

Det vanligste og enkleste designet av en scheduler er at trådene blir tilordnet prioriteter, og at scheduleren sorterer de trådene som er klare til kjøring

etter prioritet, og lar den som er først i denne køen kjøre. En grov antakelse er her at tråder som kjører ofte har høyere prioritet en de som ikke kjører så ofte.

Trådene er her selv ansvarlige for å passe på samplingstiden sin, men scheduler-delen av systemet må tilby fuksjonalitet for at trådene skal kunne si at "jeg skal ikke kjøre igjen før tidspunkt x". Dvs. at scheduleren ikke bare må organisere tråder som er klare til kjøring, men også tråder som venter til betingelsene for kjøring er klare.

Denne løsningen ligger tett opp til designet av ikke-sanntids schedulere og det vil fungere greit så lenge vi har nok regnekraft.

Legg merke til at vi ikke håndterer sanntidskravene eksplisitt på denne måten. I stedet rangerer vi oppgaver i prioritet og mer eller mindre håper at systemet skal virke etter forutsetningene. Noen spørsmål for videre-kommende: Hvorfor bruker vi ikke heller et programmeringsspråk som hånd-terer sanntidskravene eksplisitt ? Hvordan kunne en slik formalisme for å la oss spesifisere sanntidskrav eksplisitt sett ut ?

Ofte er det operativsystemet som tilbyr scheduleren og det finnes et utall forskjellige implementasjoner og strategier med forskjellige egenskaper. Vi skal skissere noen av områdene for variasjon i det følgende.

3.1 Preemptive kontra nonpreemptive scheduling

3.1.1 Nonpreemptive scheduling

Nonpreemptive scheduling er når scheduleren må vente til en tråd er ferdig med å kjøre - "frivillig oppgir kontrollen" - før den kommer til igjen. Slike schedulere er veldig sårbare for oppgaver som bruker lang tid.

Tenk deg at vi har en lavprioritets tråd som logger data til den terminalen hvor brukergrensesnittet kjøres. Hvis denne av en eller annen grunn blir stående og vente (noen restartet terminalen) eller har mye å gjøre (tråden hadde ikke fått kjørt på en stund, så data har hopet seg opp), kan det være at høyprioritetstrådene ikke får kjørt når de skal. Eller rett og slett at den tråden som beregner referansen til pumpen bruker mer enn 1/5 sekund på å gjøre beregningene sine.

Vi har i prinsippet nok regnekraft til å nå alle tidsfristene, men vi kommer i situasjoner hvor tråder med lav prioritet og høyt regnekraftbehov akkurat har startet beregningene sine når betingelesene for at en høyprioritets tråd kan kjøre (igjen) blir oppfyllt.

For å løse slike problemer må alle oppgaver designes slik at de gir fra seg kontrollen med jevne mellomrom, selv om de ikke er helt ferdige. Informasjon om hvor langt de er kommet, må lagres og administreres.

Typisk ender en opp med et temmelig komplisert design av trådene, og ofte blir det å variere prioritetene på trådene med prøve og feile-metoden i etterkant av implementasjonen, en egen del av prosjektet.

For de som husker windows før windows 95 var det en slik scheduler som lå i bunnen - hvis du skrev et program som mot formodning gikk i "evig løkke" var det Control-Alt-Delete neste :-)

Likevel; gitt at *predikterbarhet* er så viktig i sanntidssystemer som det er, kan nonpreemptive scheduling være et alternativ. Dersom en bygger opp under implementasjonen med verktøy som f.eks. forholder seg til de reelle sanntidskravene og automatisk deler berengingene i en tråd i mindre deler med retur til scheduleren imellom, kan det være et veldig godt alternativ.

3.1.2 Preemptive scheduling

Preemptive scheduling er når scheduleren tar tilbake kontrollen fra den kjørende tråden etter en gitt tid, uavhengig av hvor tråden er i kjøringen sin. Dette skjer på veldig lavt abstraksjonsnivå - alle prosessorens registre må lagres unna, inkludert instruksjonspeker og stackpeker, og scheduleren står fritt i valget av hvilken tråd som skal få kjøre som neste. Den enkelte tråd er uvitende om at den blir avbrutt. Med dette har vi løst flesteparten av problemene vi hadde med nonpremptive scheduling.

Imidlertid innfører denne løsningen minst like mange problemer som den løser. Se på følgende to tråder;

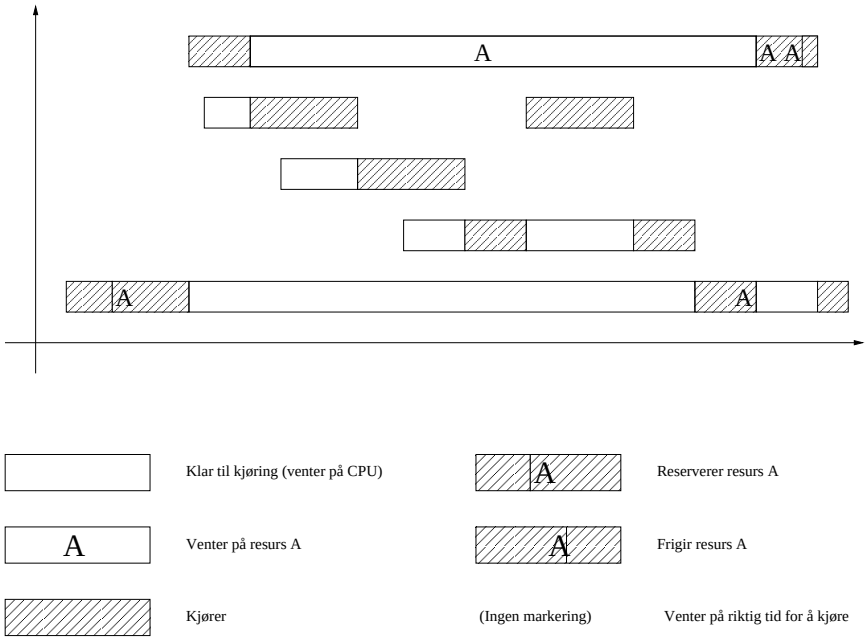
```
tråd1: {
  i = i+1;
}

tråd2: {
 i = i-1;
}
```

Begge trådene refererer samme variabel "i", de skal kjøre akkurat like ofte, og vi har nok regnekraft til å gjennomføre dette. i initialiseres til 0 ved oppstart, og dette er de eneste to referansene vi har til i. Vi skulle forvente at verdien av i ville bare kunne ha verdiene [-1,0,1], men dette holder ikke ved preemptive scheduling. Kan du se hvorfor ?

Problemet er at tråd1 kan være midt i beregningen av den nye verdien til i når den blir avbrutt. Den har lest verdien som i har (la oss si 0), men har ikke tilordnet resultatet (1) til i igjen. Så får tråd2 kjøre, leser i (0), beregner ny verdi (-1) og skriver den til i. Så kommer tråd1 til igjen og fortsetter der den slapp - og overskriver glatt i med 1. Fra nå av vil i ha verdiene [0,1,2].

Vi ser at for preemptive scheduling må aksess til minne behandles med en helt annen forsiktighet enn ved non-preemptive scheduling. Og dette gjelder ikke bare aksess til felles variable - alle biblioteker, systemkall og hardware som brukes av flere tråder må enten designes med tanke på dette eller beskyttes så de ikke blir kalt av flere tråder "samtidig".

Se for deg vanskeligheten med følgende situasjon; Vi har en hel rekke med tråder som går på forskjellige prioriteter. Alle er klare til å kjøre, men



Figur 2: Priority Inversion problemet. Høy-prioritetstråden ender med å måtte vente på alle de andre trådene på grunn av at lavprioritetstråden ble avbrutt mens den eide resurs A.

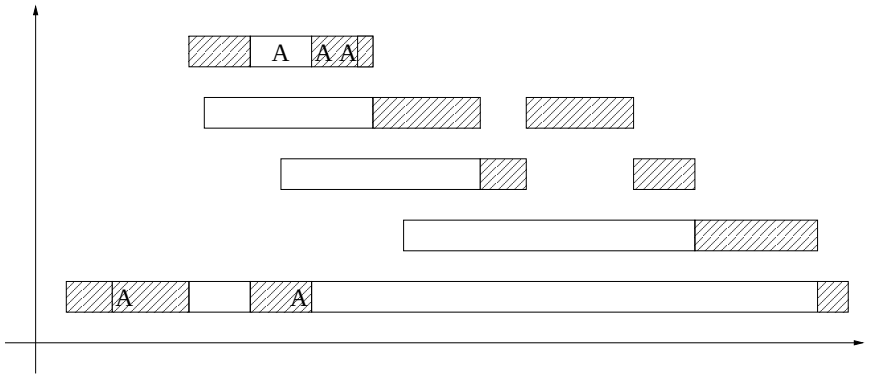
brukergrensesnitt-tråden er den som kjører akkurat nå. Brukergrensesnitt-tråden, som går på laveste prioritet, er i ferd med å sette en ny referanse til den tråden som har høyest prioritet. Den har låst aksessen til denne variabelen (mer om disse mekanismene senere), men er ikke ferdig. Så blir den avbrutt. Høyesteprioritetstråden blir stoppet i det den ber om å få lese referansen sin, siden aksess hit er låst. Hvem får kjøre ? Hvem burde fått kjørt ? Dette problemet heter *priority inversion*.

En vanskelighet som oppstår her er at slike problemer kan oppstå sjelden - det er ikke sikkert at de viser seg i det hele tatt når systemet testes. På den måten har vi mistet noe av den verdifulle forutsigbarheten som vi trenger i sanntidssystemer.

På den annen side er problemstillingene omkring preemptive scheduling relevante også for andre enn sanntidsanvendelser, så verktøy og metoder er grundig utforsket og analysert.

3.2 Nødvendige utvidelser av funksjonalitet for sanntid.

En vanlig fordom omkring sanntidssystemer er at en kan kjøpe seg ut av vanskelighetene omkring utvikling av sanntidsystemer ved å kjøpe rask nok maskinvare. Dette er riktig nok, så lenge trådene er uavhengige av hverandre - så lenge regnekraften er eneste delte resurs. Imidlertid er dette ikke



Figur 3: "Priority inversion" problemet løst med "priority inheritance". Straks høyprioritetstråden prøver å aksessere resusrs A, blir prioriteten på lavprioritetstråden satt opp.

hovedregelen for et sanntidssystem. Trådene i et sanntidssystem er til for å løse en felles oppgave så det er uunngåelig at de må gå i interaksjon med hverandre.

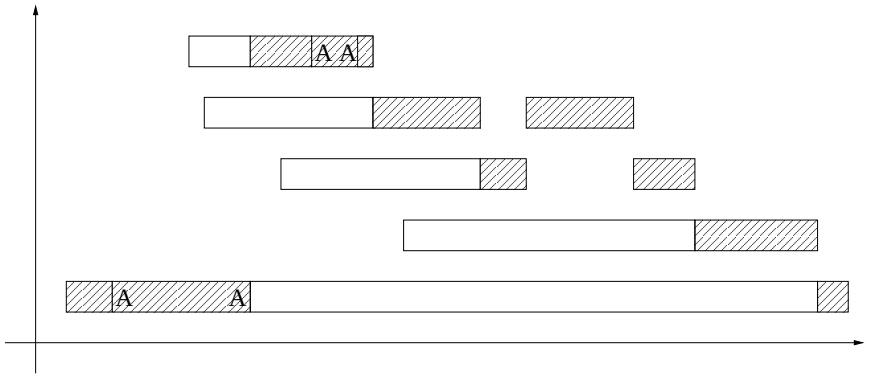
Vi har sett at regnekraft, aksess til felles minne, biblioteker, systemkall og maskinvare er ting som må reguleres. Den mest "rett frem" typen av regulering er "synkronisering" og noen vanlige synkroniseringsmekanismer er:

- **Semaforen:** I utgangspunktet er dette et flagg (altså en boolsk verdi, true eller false), som forteller om en gitt (gjerne abstrakt) resurs er ledig. Når en tråd trenger resursen, spør den scheduleren om den er ledig. Hvis den er det blir den reservert til tråden, hvis ikke står scheduleren fritt til å legge tråden i en ventekø til resursen blir ledig, og til å kjøre andre tråder. Tråden som reserverte resursen må huske å frigi den når den er ferdig med den. Som regel er de semafor-mekanismene som tilbys generaliserte slik at de bl.a. holder rede på et antall resurser av en type som er ledige.

- **Synkroniserte metoder:** Programmeringsspråket Java har "synchronized" som en del av språket. Bare en tråd i gangen kan kalle synchronized metoder i et gitt objekt.

- **Priority Inheritance:** Løser problemet **Priority Inversion**. Dette er problemet som ble nevnt over (og vist i Fig. 2) med at en lavprioritetstråd har reservert en resurs som trenges av en høyprioritetstråd. Ved priority inheritance vil lavprioritetstråden midlertidig arve prioriteten til høyprioritetstråden helt til resursen er frigitt. Se Fig. 3

- **Priority Ceiling:** En annen løsning på priority inversion. Her er en



Figur 4: "Priority inversion" problemet løst med "priority ceiling". Her er *resurs* A tilordnet en (høy) prioritet som alle tråder kjører på mens de eier resursen.

fast (høy) prioritet tilordnet *resursen*. Tråder som reserverer resursen kjører på denne prioriteten til resursen er frigitt. Se Fig. 4

- **Event:** En eller flere tråder kan vente på et event. Helt til eventet har skjedd blir trådene låst.
- **Lese/Skrive låser:** Ofte har vi situasjonen at det er trygt for mange tråder å *lese* f.eks. innholdet av variable samtidig, såfremt ingen tråder holder på å skrive til dem. Når en tråd ønsker å skrive til variabelen vil ikke flere lesere innvilges tilgang, og skriveren vil få aksess når alle aktive lesere er ferdige.
- Etc.

Dette er på langt nær alle begrepene og mekanismene som finnes, og begrepene brukes til tider forskjellig. Også kan de fleste av disse implementeres vha. hverandre.

I tillegg til slike grunnleggende mekanismer finnes det en god del abstraksjoner og mer eller mindre standardiserte overbygninger. En kø-datastruktur kan i sanntids-kontekst f.eks. implisitt være beskyttet av mekanismer som passer på at en tråd vil bli lagt i ventekøen hvis den prøver å legge til et element i køen når den er full, eller hvis den prøver å hente et element fra køen hvis den er tom.

Behovet for slike mekanismer er mindre ved non-preemptive scheduling. Evt. er det lettere å greie seg uten, siden vi har full kontroll over de punktene hvor en tråd mister eller gir fra seg kontrollen. F.eks. er eksemplet over med verdien av variabel 'i' irrelevant for non-preemptive scheduling.

Likevel kan en godt tenke seg applikasjoner hvor det kan være hensiktsmessig å la scheduleren/systemet administrere tilgang til resurser selv for nonpreemptive scheduling:

- Tenk på en tråd som over omkring 5 sekunder iterativt forbedrer et estimat av en tilstand for så å begynne et nytt løp. Noen andre tråder vil bruke "beste verdi hittil", men andre vil bare vite sluttresultatet av hver beregning. Scheduleren må her kunne håndtere andre kriterier for oppvekking av tråder enn at passe mye tid har gått.

- La oss si at vi har en ultralydsensor som bruker 0.3 sekunder på skaffe resultatet av en måling etter at den er initiert. I denne perioden hadde det vært hensiktsmessig at tråden som igangsatte målingen kunne oppgi kontrollen mens den venter, men det må da passes på at ingen andre tråder får tilgang til sensoren.

- Osv.

Vi ser at sanntidssystemer kan fort bli kompliserte hvis vi har mange tråder som er avhengige av hverandre. Et par av de problemene som kan oppstå er de følgende:

- **Vranglåser** (deadlocks): Hvis en tråd har reservert resurs A, men trenger resurs B for å fortsette, og en annen tråd har reservert B, men trenger A for å fortsette er vi i en situasjon hvor begge låsene er låst for evig tid. Dette er det enklest mulige eksemplet på en vranglås. I praksis kan det i et større system være urimelig vanskelig å forutse om du har potensielle vranglåser i et system.

- **Utsulting** (starvation): Dette er når en tråd *i praksis* ikke får kjørt så mye som den burde, fordi f.eks. en annen tråd er mye mer offensiv på reservering av en felles resurs. Her må scheduleren være veldig bevisst på at tråder på samme prioritet behandles rettferdig, og programmereren må være bevisst på hvordan scheduleren virker så en unngår urettferdig scheduling.

4 Programmeringsformalismer for sanntid

4.1 C, C++, Pascal, Modula etc.

Dette er programmeringsspråk som ikke har innebygde mekanismer for tråder, scheduling og synkronisering i seg selv. Imidlertid kan de gjøre kall til operativsystemet eller til biblioteker som gir et mer eller mindre komplett sett av egenskaper.

Spesielt er språket C sentralt, siden dette er den kompilatoren som vanligvis først blir skrevet for en ny mikroprosessor. Det er også maskinnært nok til å kunne forholde seg til all maskinvare. Drivere og operativsystem er vanligvis skrevet i C.

4.2 POSIX

Posix er et standard sett med grensesnitt for å aksessere operativsystemet, med forhåpning å kunne skrive portable programmer på tvers av OS. Dette inkluderer "pthreads" som er mekanismer for å skape tråder og for å synkronisere slike tråder.

Den grunnleggende synkroniseringsmekanismen her er Mutex'en som fungerer som en binær semafor, og som navnet sier er den rettet mot å løse "mutual exclusion" problemet - at vi har en eller flere kodebiter som ikke må kjøres av flere tråder "samtidig".

I tillegg til dette har vi "Condition Variables" som fungerer som eventer, bare med utvidelsen at hvis du blir blokkert av en condition variabel fra innsiden av en mutex-region så kan mutexen frigis mens du venter!

Hvis du skal skrive flere-trådete programmer i C er det sannsynligvis POSIX du kommer bort i, selv om de enkelte operativsystemene av og til også tilbyr egne, mer grunnleggende, grensesnitt.

4.3 Ada, Java etc.

Her er begreper om tråder og et utvalg synkroniseringsmekanismer gjort til en del av programmeringsspråket - og ikke et spørsmål om operativsystem eller bibliotekskall.

I Java opprettes objekter av tråd-type som kan start()'es. Synkronisering skjer ved at metoder blir definert som "synchronized", som gjør at en tråd trenger objektets mutual exclusion-lås for å kjøre dem. Også her har vi en ventemekanisme ("wait()") som kan kalles fra en synchronized metode og som frigjør låsen mens tråden er blokkert.

I ADA har vi tilsvarende tråd-"objekter" - tasks, men hvor synkronisering skjer ved enten "protected objects" - et sett funksjoner som kjører under mutual exclusion, eller "rendezvous" hvor to tråder venter på hverandre, og det skjer en toveis informasjonsdeling og et stykke kode kan kjøres av de to trådene "felles" før begge får fortsette med sitt.

For protected objects har vi ikke muligheten til å frigi objektet mens vi venter på noe annet, men vi har derimot muligheten til å "gi opp og prøv igjen" - requeue.

4.4 OCCAM og CSP

OCCAM er et språk som er designet ut ifra en helt annen filosofi enn de andre språkene, nemlig *Communicating Sequential Processes* - CSP.

De aller fleste programmeringsspråk er *imperative* på den måten at et program, grovt sett, består av sekvenser med kommandoer som vi ønsker at maskinen skal utføre. Men tenk på SQL; Dette *er* en beskrivelse av noe vi ønsker at maskinen skal gjøre, men den er ikke gitt ved deloppgaver som skal utføres i sekvens.

I OCCAM er "primitivene" prosesser som i utgangspunktet uavhengig av hverandre - i utgangspunktet kan de bli utført i vilkårlig rekkefølge eller parallellt, kanskje på forskjellige maskiner eller CPU-kjerner. Avhengigheter imellom prosessene blir ansett som (behov for) kommunikasjon og håndtert av egne primitiver for kommunikasjon.

CSP er en matematisk formalisme for å beskrive slike parallelle prosesser. Hvis du har skrevet programmet ditt i OCCAM kan du overføre dette relativt greit til en CSP beskrivelse og det å bevise at programmet ditt er f.eks. fritt for potensielle vranglåser blir som å bevise et teorem i en annen gren av matematikken. Når det er sagt, er dette i praksis ingen farbar vei, men uansett er OCCAM et språk hvor mye av det som blir uoversiktlig i de imperative språkene blir tindrende klart og hvor det er mulig å resonnere på hvordan programmet virker i større grad enn for de andre.

Vi har ingen delte variable i OCCAM, og ingen preemptive scheduling - ingen synkroniseringsmekanismer, bare kommunikasjon.

For andre ikke-imperative språk, titt på haskell eller prolog.

5 Tilstandsmaskiner

Til forskjell fra programmeringsspråk, bibliotekskall (som ved POSIX) og operativsystemegenskaper, er tilstandsmaskiner ikke mekanismer du vanligvis bruker direkte i programmeringen av sanntidssystemer. Det er en abstraksjon som det ofte er hensiktsmessig å bruke når du *designer* systemet ditt. Mange beregninger er av den karakter at vi har et sett av mulige hendelser (eventer) i systemet som vi må forholde oss til, og at hva vi skal gjøre som følge av et gitt event er avhengig av hva som har skjedd tidligere eller hvilken modus systemet er i - *tilstanden*. Implementasjon av slike systemer kan bli urimelig komplisert om en ikke har en godt gjennomtenkt underliggende struktur i koden. Tilstandsmaskiner er først og fremst *en måte å tenke på* i slike situasjoner. Det finnes likevel et utall verktøy som genererer kode for deg, gitt en beskrivelse av tilstandsmaskinen din, og til og med et og annet programmeringsspråk hvor tilstandsmaskiner er et eksplisitt begrep.

Det er fire begreper som er sentrale i en tilstandsmaskin-modell.

- **Tilstander**: Systemet kan være i en av et endelig antall tilstander. Hvilken av disse tilstandene det er i, er avgjørende for hva som skal gjøres som reaksjon på de eventene som kan komme.
- **Eventer**: Dette er de hendelsene i systemet utenfor tilstandsmaskinen som krever en reaksjon fra tilstandsmaskinen.
- **Transisjoner**: Dette er en overgang i tilstandsmaskinen fra en tilstand til en annen. Som regel er det langt færre lovlige transisjoner enn settet av alle mulige kombinasjoner av fra- og til-tilstand.

- **Aksjoner**: Dette er de virkningene som tilstandsmaskinen har på systemet utenfor tilstandsmaskinen.

5.1 Et eksempel:

Du skal implementere en digital styring av en kassettspiller: Den har et standard sett av knapper: Play, Stop, FastForward, Rewind, Record, Pause og Eject. I tillegg har vi følgende deler av systemet:

- Mekanikken tar følgende kommandoer: stop, play, fastforward, rewind, pause og eject, og du kan avlese om motorene beveger seg eller ikke, om kassetten er skrivbar eller ikke og om det hele tatt er kassett i maskinen.
- Elektronikken har modi record og play og i tillegg kan du styre en lampe som skal vise om du faktisk tar opp.

5.1.1 Feil måte å gjøre det på

La oss ta et sidesprang her og bekrive den gale måten å nærme seg problemet på (Noen vil kanskje kalle det den naturlige eller normale måten :-)

Du estimerer kvikt at denne jobben vil du bruke en uke på siden du kjenner utviklingssystemet og driverne mot maskinvaren. Du tenker at når en trykker på en knapp bør dette få en gitt reaksjon, du lager en løkke som sjekker hvilke knapper som er trykket inn og begynner å skrive slike aksjoner:

- Når du trykker Play starter du mekanikken i play modus.
- Når du trykker FastForward starter du mekanikken i fastforward-modus.
- Når du trykker Record setter du elektronikken i recordmodus.
- etc.

På denne måten kommer du fort opp og får et system som "virker" - du kan styre kassettspilleren med knappene!

Men etterhvert utover uken kommer du til å tenke på følgende - i verste fall er det ting som du oppdager i det du begynner å teste systemet:

- Når du trykker Play og spilleren allerede spoler må systemet stoppe først. Likedan når du trykker FastForward eller Rewind og spilleren allerede går.
- Play bør egentlig ikke gjøre noe, når systemet allerede er i play modus.
- Hvis mekanikken ikke er i stop modus og motorene ikke går, har vi kommet til enden av båndet og systemet skal stoppe.

- Recordknappen skal også kunne resette recordmodusen til elektronikken.
- Stopknappen skal også resette recordmodusen til elektronikken — hvis denne er satt.
- Recordknappen skal bare ha noen funksjon når mekanikken er i stop modus.
- etc.

Disse tingene er alle ganske enkle å fikse (selv om det er mange av dem); du trenger variable som husker hvilke modi systemet er i og if-setninger som tester på dem i aksjonene for de forskjellige knappene.

Du ender med funksjoner av denne typen:

```
/* Globale variable */
bool play;
bool record;
bool pause;
bool ff;
bool rw;

play(){
  if(play == false && kassett()){
    if(ff || rw){
      mekanikk_stop();
      ff = false;
      rw = false;
    }
    mekanikk_play();
    motor_play();
    if(record == false){
      elektronikk_play();
    }
    play = true;
  }
}
```

Kan du ved å se på denne funksjonen greie å overbevise deg selv om at den alltid vil virke slik den var tenkt ?

Når uken er gått har du et virkende system.

Men på demoen du skal ha for oppdragsgiverne kommer følgende frem:

- En tulling fra markedsavdelingen begynner å trykke på flere knapper samtidig - og holder knappene inne lenge - og avdekker flere svakheter i implementasjonen din.
- Recordlampen skal bare lyse når du faktisk tar opp - den skal settes på når du trykker play og elektronikken er i recordmodus.
- Eject skal virke som stop når mekanikken ikke er i stopmodus. (En skal måtte trykke eject to ganger for å få ut kassetten.)

- FastForward og Rewind knappene skal disables når systemet er i record-modus.

- etc.

Du forsvarer deg godt med at de fleste av disse tingene ikke var nevnt i spesifikasjonene du fikk. Litt skeptisk til deg selv (evt. selvrettferdig), gir du en uke i estimat på å fikse "endringene i spec'en". Nå vet du jo hvor komplisert systemet er - du har 5 globale statusvariable rundt omking i systemet og de verste aksjonene begynner å bli temmelig kompliserte allerede.

Vel og bra - etter en uke til (og et par sene kvelder på slutten) får du godkjent programmet ditt. Godt gjennomført prosjekt!

Dine neste oppgave i jobben blir å lage et tilsvarende styresystem for en annen modell kassettspiller. Her er play og pauseknappen slått sammen, mekanikken har samme funksjon, men forskjellige drivere, og det er ikke noe "equalizer-display" noe som gjør at record-lampen må virke annerledes. Klok av skade estimerer du to uker på dette prosjektet.

I løpet av de 10 neste kassettspillerene oppjusterer du estimatet fra 2 til 3 uker, noe som knapt holder siden du stadig får inn feilmeldinger på de kassettspillerne du allerede har laget. (Det viser seg bl.a. at hvis du går ut av record-modus ved å trykke Eject i stedet for Stop eller Record, så blir ikke den variabelen som husker om systemet er i record-modus resatt, noe som fører til at hvis du skal ta opp igjen like etterpå, så vil spilleren gå i vanlig play-modus...) Det blir mange sene netter, men du bygger ditt rykte i bedriften som en hardtarbeidende fikser, en mester i kassettspiller-styringer. Du er godt på vei til guru-status - og det er *før* du får ansvaret for videospillerne!

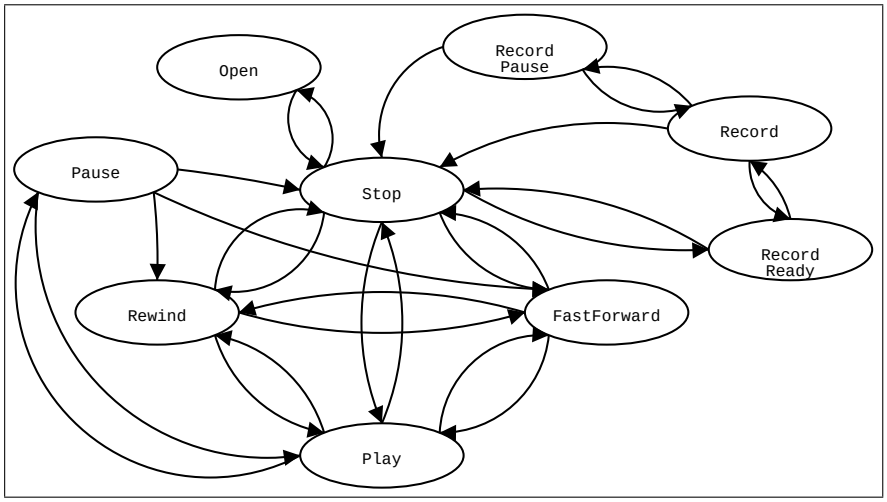
Etter fem år slår den tanken deg at det hadde vært morsomt å gjøre noe annet. Du har følgende alternativer:

- Insistere på å få andre oppgaver. Dette er i og for seg lett nok, men du innser at det virkelige problemet er å bli kvitt de gamle :-) Alle gamle styresystemer utgjør en vesentlig vedlikeholdsbyrde og det er bare du som har oversikt. Dessuten er det jo ingen som har så god greie på kassettspillere som du.

- Skifte jobb.

Konsekvensen av å ha delt dine betraktninger her med sjefen er at du blir gruppeleder - du får 3 testere (som skal teste koden din) på gruppen og går opp nesten 50arbeidsoppgavene er de samme...

Programvarenorge er fullt av slike guruer...



Figur 5: Tilstandsmaskin

5.1.2 Tilstandsmaskinmodellering.

Her starter vi med å sette opp hvilke *tilstander* spilleren skal kunne være i. Hvis du kommer inn i et rom hvor det står en kassettspiller, kan denne bare være i et begrenset antall tilstander; Stop, Play, FastForward, Rewind, Pause, RecordReady, Record, RecordPause, DoorOpen. Det er ganske greit og oversiktlig å fastslå at du ikke har glemt noen her.

Knappetrykkene blir her *eventer* som skal bringe spilleren fra den ene til den andre tilstanden. Det å skifte tilstand er en *transisjon*, og det å endre på omgivelsene (mekanikken og elektronikken) er aksjoner.

Eventene er altså Stop, Play, FF, RW, Pause, Record, og i tillegg kommer en EndOfTape som blir internt generert av mekanikken. Det er nå også ganske greit å gå gjennom hva hvert event skal ha som konsekvens i de forskjellige tilstandene.

Nå kan vi tegne tilstands/transisjons-diagrammet i Fig. 5 og sette opp tabellen over aksjoner for de forskjellige eventene i de forskjellige tilstandene:

```
Fratilstand TilTilstand Eventer      Betingelser Aksjoner
Stop        Play        Play                     Play
Play        Stop        Stop                     Stop
                        Eject
Stop        RecordReady Record       Skrivbar    RecordOn
RecordReady Record      Play                     Play
Stop        Open        Eject                    Open
Open        Stop        Kassett
Rewind      FastForward FastForward              Stop
                                                 FastForward
...
```

Legg merke til at ikke alle tilstander er modellert her - vi skiller f.eks. ikke imellom det å spille en skrivebeskyttet kassett fra å spille en ikke

skrivebeskyttet kassett. Skrivbarheten kommer inn som en ekstra test bare når du trykker Record. Likedan eventene Kassett (som betyr at du har satt i en kassett i spilleren) og EndOfTape (som betyr at motorene er i gang, men ikke beveger seg) må spesialhåndteres. Dette er avveininger som må gjøres.

Så, når tabellen er komplett må vi implementere systemet. Det er mange forskjellige måter å gjøre dette på. Poenget er at det kan gjøres mer eller mindre automatisk - alle designavgjørelser ligger i tabellen. Her vil vi skrive en funksjon for hvert event, hvor alle inneholder en switch/case setning for å skille imellom tilstandene.

- Vi trenger *en* variabel som forteller hvilken tilstand systemet er i ("g_state" som kan ha verdier som "State_Open", "State_Stop" etc.).
- Vi trenger en løkke som sjekker status på knappene og i systemet og genererer eventer (dvs. i vårt tilfelle; kaller eventfunksjonene). Denne er din "main-funksjon" - den som blir kalt når systemet starter.
- Vi trenger en prosedyre for hver aksjon som står for selve styringen av elektronikken og mekanikken. Disse er i hovedsak korte - stort sett bare rene kall til driverne.
- Vi trenger å implementere en funksjon for hvert event.

En eventfunksjon kan se slik ut (i programmeringsspråket C):

```
void
event_play(){
  switch (g_state) {
    case State_Open: break;
    case State_Stop: action_play(); g_state = State_Play; break;
    case State_Pause: action_play(); break;
    case State_Rewind: action_stop(); action_play(); g_state = State_Play; break;
    case State_FastForward: action_stop(); action_play(); g_state = State_Play; break;
    case State_Play: break;
    case State_RecordReady: action_play(); g_state = State_Record; break;
    case State_Record: break;
    case State_RecordPause: action_play(); g_state = State_Record; break;
  }
}
```

Du ser at dette er en "ren avskrift" av tabellen.

La meg lage en kort versjon av din karriere med dette utgangspunktet: Du rekker ikke riktig å ha systemet demonstrerbart etter en uke siden du har mer infrastruktur å drasse på, men når du etter 1 1/2 uke demonstrerer systemet blir funksjonaliteten godkjent med små endringer. (Endringene gir seg bare utslag i tabellen, og de er trivielle å implementere.) Det eneste større punktet som kommer opp er at etter at du har trykket stop i play-modus er knappene "døde" i det halve sekundet det tar for mekanikken å stoppe. Men beslutningen er å levere systemet alikevel.

Du estimerer 1 uke på neste kassettspiller og har tid til overs på slutten av uken.

Ved tredje kassettspiller ber du om to uker ekstra for å lage et verktøy som leser tabellen din og en liste av tilstander og automatisk generere koden for eventfunksjonene. Fra nå av slipper du det manuelle arbeidet med å oversette fra tabellen til koden.

Med dette verktøyet på repertoaret finner du fort ut at den variabelen som anga om kassetten var skrivbar eller ikke, egentlig er en liten tilstandsmaskin i seg selv, med to tilstander - og implementerer det slik i neste versjoner av styringene dine.

Nå ser du også den gode løsningen på problemet med at du ikke får sjekket status på knappene mens mekanikken jobber; Du innfører to tråder i systemet, en som leser knappene og bygger en kø av eventer, og en som leser køen og kaller event-funksjonene. Saken er at denne køen egentlig er en tilstandsmaskin i seg selv som holder rede på hva "brukeren egentlig vil", gitt at han hamrer på knappene fortere enn mekanikken henger med. F.eks. vil sekvensen Stop-Play-Stop-Play kollapses til Stop-Play. Dette gir en langt bedre opplevelse av systemet for brukeren av kassettspilleren, - bedre enn for mange av konkurrentene.

Gå hjem og test hvordan de forskjellige spillerne du har der håndterer at du trykker flere knapper i gangen, eller hamrer på knappene raskere enn at mekanikken henger med - på eget ansvar :-) Hjemme hos meg har vi to CD-spillere og to kassettspillere hvor en av hver sort ikke reagerer i det hele tatt mens mekanikken arbeider, mens de andre tydeligvis har en strategi hvor siste knappetrykk huskes selv om mekanikken arbeider. Og videomaskinen har den irriterende egenskapen at om jeg vil spole til enden av tapen, så går den automatisk i rewind-modus når den kommer frem...

Så, etter at du er lei av at markedsavdelingen ombestemmer seg mhp. hvordan knappene egentlig skal virke (det gjør de hele tiden, nemlig), holder du et seminar for dem hvor du sier at du vil ha spesifikasjonene i form av en tabell :-) Nå er ditt arbeide med hver spiller redusert til mindre enn en dag per kassettspiller - du må hjelpe markedsfolkene med spesifikasjonene på den første videospilleren, men ellers er du klar for nye oppgaver.

6 Konklusjon / Moral

"Stiller du et komplisert spørsmål risikerer du å få et komplisert svar."

Implementasjonen av et sanntidssystem er til en viss grad et slikt komplisert spørsmål. Det er mange ting å være oppmerksom på; du må kjenne systemet du skal jobbe med, spesifikasjonene, og de formalismene du skal bruke, godt, og du må være oppmerksom på de vanlige farene ved de forskjellige teknikkene. Og se det i øynene - det er en grense for hvor kompliserte systemer du kan håndtere.

Ditt sterkeste våpen i denne kampen er menneskets frihet til å velge hva du vil tenke; Du kan velge hvilke begreper du bruker når du designer eller implementerer. Problemer som blir urimelig kompliserte når de beskrives med ett sett av begreper kan bli overkommelige med et annet. Den vanligste fallgruven er at du tenker med de begrepene som programmeringsspråket eller utviklingssystemet tilbyr deg. Et programmeringsspråk er nødvendigvis laget med tanke på å være "general purpose" - ikke med tanke på applikasjonsområdet ditt. Som jeg håper at jeg greide å illustrere, kan da selv noe som er så enkelt som styringen til kassettspilleren din bli så komplisert at det blir vanskelig å få den riktig og vedlikeholdbar.

Mao. hvis problemet ditt har preg av tilstander og transisjoner, så tenk tilstandsmaskiner. Hvis problemet har preg av flere oppgaver som skal løses og er bare delvis avhengige av hverandre, tenk tråder og synkronisering (eller kanskje enda bedre; tråder og meldingssending).

Tilstandsmaskiner og tråder er bare to av mange slike "designformalismer" og kunsten er å bruke (eller finne opp) den rette for din applikasjon.

Etterhvert som du føyer slike begrepssett og verktøy til ditt repertoar vil du bli i stand til å gyve løs på problemer som før ikke var gjennomførbare.

Jeg beskrev under nonpreemptive scheduling den arbeidskrevende lavprioritetstråden som måtte returnere til scheduleren med jevne mellomrom selv om den ikke var ferdig, og jeg sa at dette ofte ble så komplisert at det ble til et argument mot nonpreemptive scheduling... Kan du se at denne kanskje bør modelleres som en tilstandsmaskin, hvor tilstanden beskriver hvor langt den er kommet i beregningene ?

Vær oppmerksom på disse tingene etterhvert som du lærer mer programmering utover i studiet, selv om du nå kanskje strever nok med å føye programmeringsspråkets if/ while/ switch/ for/ function/ module -begrepssett til repertoaret. Vær nysgjerrig når du kommer bort i en formalisme, et verktøy eller et programmeringsspråk som krever at du *tenker på en annen måte.*