Avdeling for informatikk og e-læring
Geir Maribu

1. Prosesser

Resymé: Prosessbegrepet står meget sentralt i operativsystemet. I denne leksjonen får du vite hav en prosess er, hvorfor den er så viktig i operativsystemer og hvordan den virker. Du lærer hvordan prosesser administreres, om forholdet mellom prosess og program, om hvordan prosesser bruker ressurser etc. Videre vil du også lære om bruken av trådprosesser, hvorfor disse er så viktige i moderne operativsystemer.

1.1. Prosessbegrepet

Hva er et flerprosess-operativsystem? Hvordan virker det? Hva er spesielt med operativ-systemer som kan håndtere flere prosesser som kjører samtidig. Hvordan er det mulig at flere personer kan jobbe mot den samme datamaskinen?

1.2. En definisjon

"En prosess er et program under utførelse" er en mye brukt definisjon på prosess. Dette er en enkel og grei definisjon, men det kan sies ganske mye mer om prosesser før en får den hele og fulle forståelse av begrepet.

1.3. Hva er en prosess - noen sammenligninger

Program vs prosess

Et dataprogram kan gjerne sammenlignes med ei lærebok. På samme måte som kjøringen av et program er en prosess, kan en også si at lesing av ei bok er en pågående aktivitet som kan kalles en prosess.

Et annet eksempel er å bake ei kake. Da har du ei kakeoppskrift og en del ressurser som f.eks mixer, kakeform og ovn. I tillegg har du egg, mel og sukker. Kakeoppskriften tilsvarer programmet, mens mixer og kakeform er ressurser. Egg, mel og sukker er data.

Kakeoppskriften er "helt død" så lenge den står der i kokeboka. Det er først når du setter i gang en aktivitet med kakebaking at det oppstår en prosess du kan kalle kakebaking.

Flere prosesser, samme programkode

Det at to musikere kan dele på et noteark når de spiller et musikkstykke kan sammenlignes med at to prosesser kjører akkurat det samme programmet. På datamaskinen skjer det når f.eks flere personer bruker akkurat samme editoren. Men det finnes også en vesentlig forskjell. Musikerne utfører programmet (notene) helt simultant, mens i eksemplet med

editoren kan prosessene være på helt forskjellige steder i programmet. En kan f.eks. kjøre programkoden som genererer innholdsfortegnelse, en annen kjører programkoden som flytter tekstblokker, mens en tredje kjører programkoden som søker etter et gitt ord.

Skifting mellom prosesser

Når jeg leser ei bok vil jeg sette et merke for hvor langt jeg er kommet for deretter å legge fra meg boka. Dattera mi ser boka og fatter interesse for den og starter å lese. I en periode vil vi begge lese boka, og dermed konkurrere om hvem som skal lese boka. Dette kan sammenlignes med to prosesser som kjører samme programmet (slik som i editor-eksemplet ovenfor). Men i dette tilfellet er det bare mulig for en av oss å lese boka om gangen, og det foregår ved hjelp av tidsdeling. På samme måte vil også operativsystemet fordele cpu-tid til prosesser - kun en prosess kan kjøre om gangen, forutsatt at det er kun en cpu i maskinen.

Avbrudd og prosesser

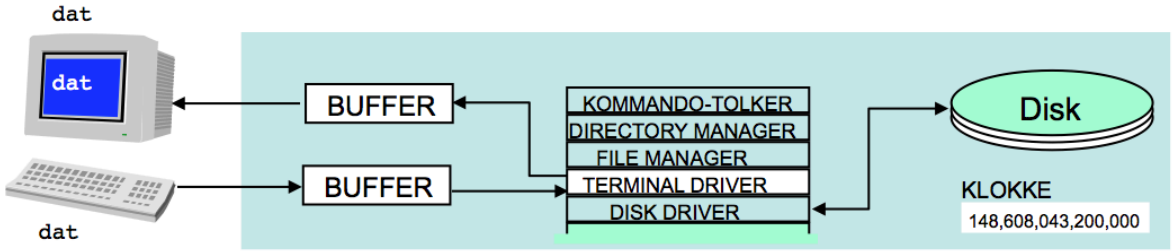
Hvis telefonen ringer når jeg leser i boka vil jeg sette et merke i boka, legge den fra meg, og deretter svare telefonen. Når jeg er ferdig med telefonsamtalen går jeg tilbake til boka for å lese videre. På samme måte fungerer det også i datamaskiner - en prosess som kjører blir avbrutt av at du f.eks trykker ned en tast på tastaturet, eller at diskkontrolleren sier fra at nå er datablokka skrevet til disken. Når dette avbruddet er ferdig behandlet vil operativsystemet føre kontrollen tilbake til den prosessen som kjørte idet avbruddet kom.

1.4. Prosesser i datamaskinsystemet - et eksempel

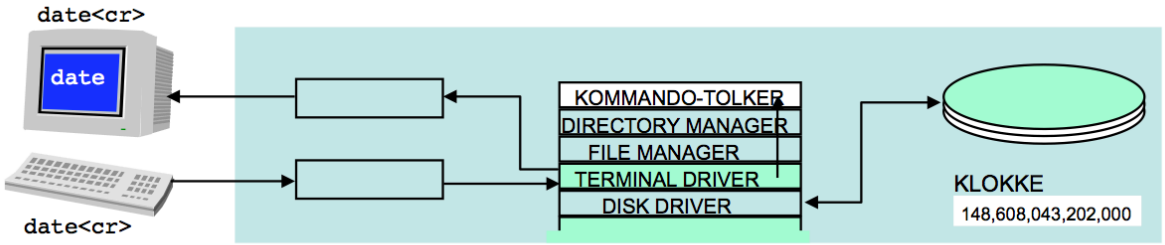
I det følgende skal vi se på et enkelt eksempel som involverer flere prosesser. Det starter med en kommando som tastes inn og avslutter med at teksten skrives ut til skjerm. Kommandoen som gis er *date*-kommandoen som leser av systemklokka i maskinen og skriver ut hvilken dato det er. Eksemplet er hentet fra Scientific American, september 1984.

Eksemplet illustrerer godt alle de aktivitetene som settes i gang når en kommando gis i et operativsystem. Dette er aktiviteter som vi i det følgende skal kalle for prosesser. En prosess for hver aktivitet. Figuren viser til enhver tid innholdet i primærlageret (RAM) og hvilke prosesser som er aktive for øyeblikket.

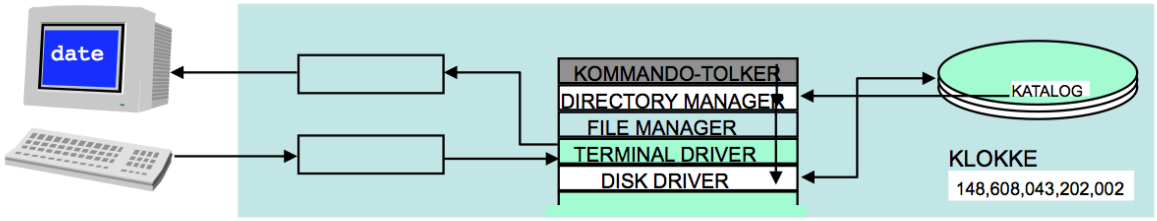
Til hver figur som følger i dette eksemplet vil teksten som forklarer figuren følge rett etter figuren.



Kjøring av kommandoen date vil avstedkomme en rekke aktiviteter i datamaskinen. Hver aktivitet foregår som en egen prosess. Hvert tegn som tastes på tastaturet mottas av prosessen terminaldriver som også gir ekko til skjerm.



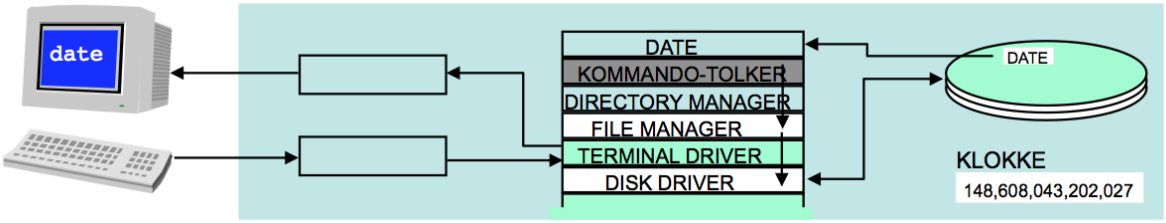
Når Enter-tasten trykkes vil terminaldriveren sende tekststrengen "date" over til kommandotolker-prosessen som sjekker tekststrengen og finner ut at det er en kommando.



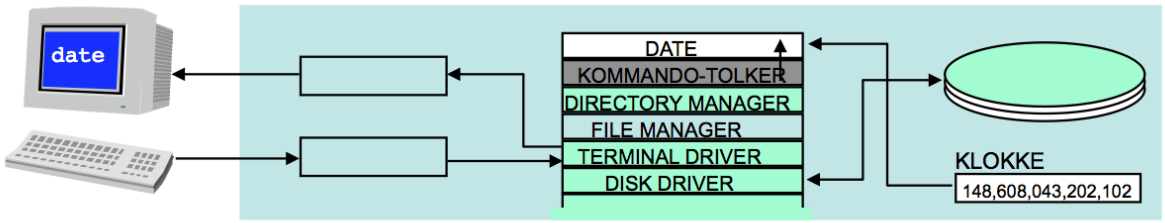
Directory-manager er den prosessen som holder rede på alle programmer/kommandoer som finnes på disken. Kommandotolkeren spør directory-manager om å søke i katalogene sine etter kommandoen date. Directory-manager må ha katalogene inn i minnet for å kunne søke i disse. Directory-manager spør derfor disk-driver om å kopiere kataloger til et bufferområde i minnet, et bufferområde som disponeres av directory-manager.

Legg også merke til arbeidsdelingen. Directory-manager kan ikke selv hente inn data fra disken. Directory-manager sin oppgave er å holde rede på kataloger, finne fram i kataloger etc. Den har ikke som oppgave å hente data fra disken på egen hånd. Til dette må den ha hjelp av disk-driveren som har dette som spesialoppgave.

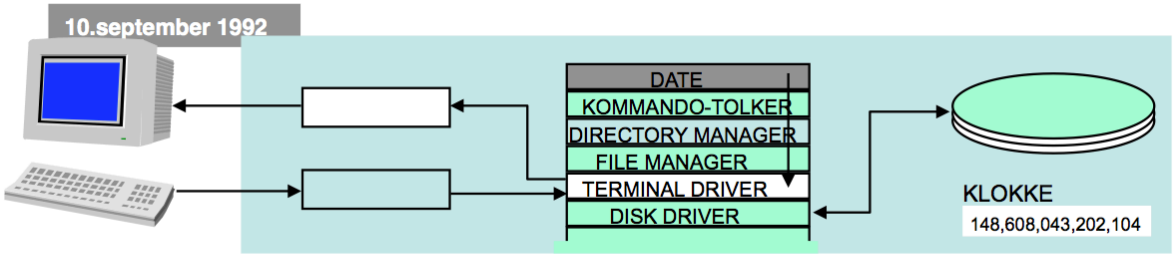
Denne måten å organisere aktivitetene i datamaskiner på er svært vanlig. Hver prosess har sin spesielle oppgave å utføre. Hvis andre prosesser trenger å utføre oppgaver som andre kan bidra med må de sende en forespørsel først. Prosessene må altså kommunisere med hverandre.



Når Directory-manager har funnet date-kommandoen vil Kommandotolkeren be File-manager om å hente inn date-programmet fra disk til primærlager. File-manager vil også sjekke om du har lov til å kjøre dette programmet, etc. Når dine rettigheter til date-programmet er sjekket og funnet i orden blir Disk-driveren spurt om å hjelpe til for å hente inn date-programmet. For å utføre denne oppgaven trenger Disk-driveren å vite hvilke blokker den skal hente fra på disken. Disse opplysningene får den av File-manager.



Kommando-tolkeren starter opp date-programmet, som leser klokka. Deretter omformer date-programmet verdien som leses fra klokka slik at det er forståelig for mennesker.



Date-programmet regner ut hvilken dato det er i dag på grunnlag av tallet lest fra klokka. Deretter viser date-programmet datoen på skjermen ved hjelp av Terminal-driveren.

De forskjellige aktivitetene/prosessene vi har sett i dette eksemplet er typiske for moderne operativsystemer. Hver prosess har sine spesialoppgaver, og for å utføre en større og mer sammensatt oppgave må prosessene samarbeide (kommunisere) med hverandre slik som vist i dette eksemplet med date-kommandoen.

1.5. En prosess for hver aktivitet

En kan si at et program er en sekvens av instruksjoner. Utførelse (kjøring) av en slik sekvens av instruksjoner er da en prosess. I enkle operativsystemer (som f.eks ms-dos) kjøres det kun en slik sekvens av instruksjoner om gangen. Da måtte en prosess avslutte helt før den neste kunne starte opp. Dette ble kalt for enprosess-system.

I et flerprosess operativsystem tillates det at flere sekvenser av instruksjoner kjører samtidig i datamaskinen. Med samtidig menes her ikke nødvendigvis virkelig sann samtidighet, men en slags *tidsdeling av cpu-en* der hver av prosessene som kjører på cpu-en skifter på å bruke cpu-en. Først kjører den ene prosessen et lite øyeblikk, deretter den neste etc. Slik fortsetter det, og i løpet av f.eks 10 sekund kan en observere at alle prosessene på maskinen har kommet et stykke videre. Dette kalles å kjøre flere prosesser samtidig (kvasiparallell).

Det er operativsystemet som må sørge for denne vekslingen mellom prosessene ("context switching"). I maskiner som inneholder flere cpu'er (flerprosessor-systemer) kan en få til virkelig samtidig utførelse av prosesser. Men også i slike systemer vil en som oftest ha flere prosesser enn en har prosessorer så også her trenger en den ovenfor beskrevne tidsdelingen av cpu-en.

Operativsystemene har forskjellige regler (algoritmer) å gå etter når det skal avgjøres hvilken prosess som skal få kjøre på cpu-en, og hvor lenge den skal få kjøre. I et tidsdelt operativsystem som f.eks Linux tillates det at hver prosess kjører en liten tidsperiode hver, slik at alle prosesser som ønsker å kjøre får kjørt innen rimelig tid. En prosess beholder cpu-

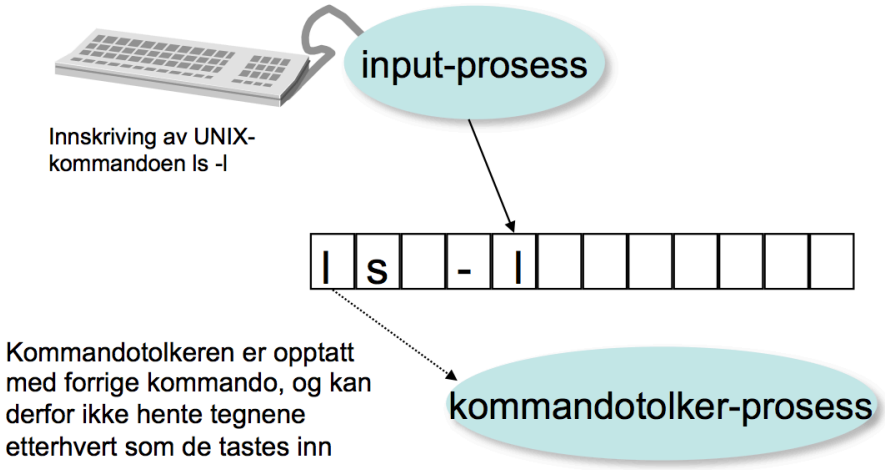
en inntil den ikke har mer arbeid å utføre, eller til det kommer en ny prosess med høyere prioritet, eller operativstemet "finner ut" at kjørende prosess har kjørt lenge nok (suspenderes). Deretter går cpu-en til en annen prosess.

Suspendering av prosesser er helt normalt. Det er på denne måten operativsystemet greier å fordele cpu-tid til alle prosessene. Når en prosess er suspendert må det lagres unna en del status-informasjon om prosessen slik at det er lett å gjenoppta kjøringen av prosessen på samme sted som den ble stoppet, dvs nøyaktig på samme instruksjon. Denne statusinformasjonen lagres i en såkalt *prosessdeskriptor* (kalles også for Process-Control-Block, eller bare en PCB-blokk). Mer om prosessdeskriptorer nedenfor. Operativsystemet gjemmer unna en del informasjon om hver prosess, informasjon som trengs for å kunne starte opp igjen prosessen på en enkel måte. Informasjon som lagres er f.eks:

- innholdet av cpu-registrene idet prosessen ble stoppet
- adressen til neste instruksjon som skal utføres
- adresse til kode-område (program)
- adresse til data-område
- forbruk av cpu-tid så langt, etc

Når det er denne prosessen sin tur å kjøre igjen vil operativsystemet gjenopprette status slik den var idet prosessen ble stoppet, dvs. cpu-registrene fylles opp med data fra prosessdeskriptor. Adressen til neste instruksjon som skal utføres (kjøres) er spesielt viktig her, fordi etter at den er lagt inn kan prosessen start opp på nøyaktig samme sted som den ble stoppet. Også innholdet i de andre cpu-registrene vil være gjenopprettet da også dette er hentet fra prosessdeskriptoren.

Det er mange fordeler med å ha flere prosesser aktive samtidig i datamaskinen. For det første så kan hver prosess konsentrere seg om helt bestemte oppgaver uten å bry seg om hva andre prosesser i systemet måtte holde på med. Tenk f.eks på følgende eksempel: I et operativsystem finnes følgende to prosesser - en *input*-prosess som leser tegn fra tastaturet og lagrer hvert tegn i et buffer, og en *kommandotolker* som henter tegn fra dette bufferet og tolker dette som kommandoer som skal utføres. Input-prosessen kan lagre hvert tegn etter hvert som de skrives inn uavhengig av om kommandotolkeren er klar til å motta en ny kommando. Du kan altså skrive inn kommandoer uten at kommandotolkeren er klar til å ta imot (type-ahead). Å lage disse prosessene som ett program ville gitt en mye mer komplisert løsning.



Bruk av flere prosesser gir også en fordel når operativsystemet trenger å håndtere alle de forskjellige i/o-enhetene som er koblet til maskinen. Spesielt gjelder dette de i/o-enhetene som også sender avbruddssignaler til cpu-en, f.eks tastatur og harddisk. Kommunikasjonen med i/o-enheter er som oftest asynkron med hensyn til kjøringen av selve programmet. Avbruddsignaler kommer sjelden akkurat når programmet forventer det.

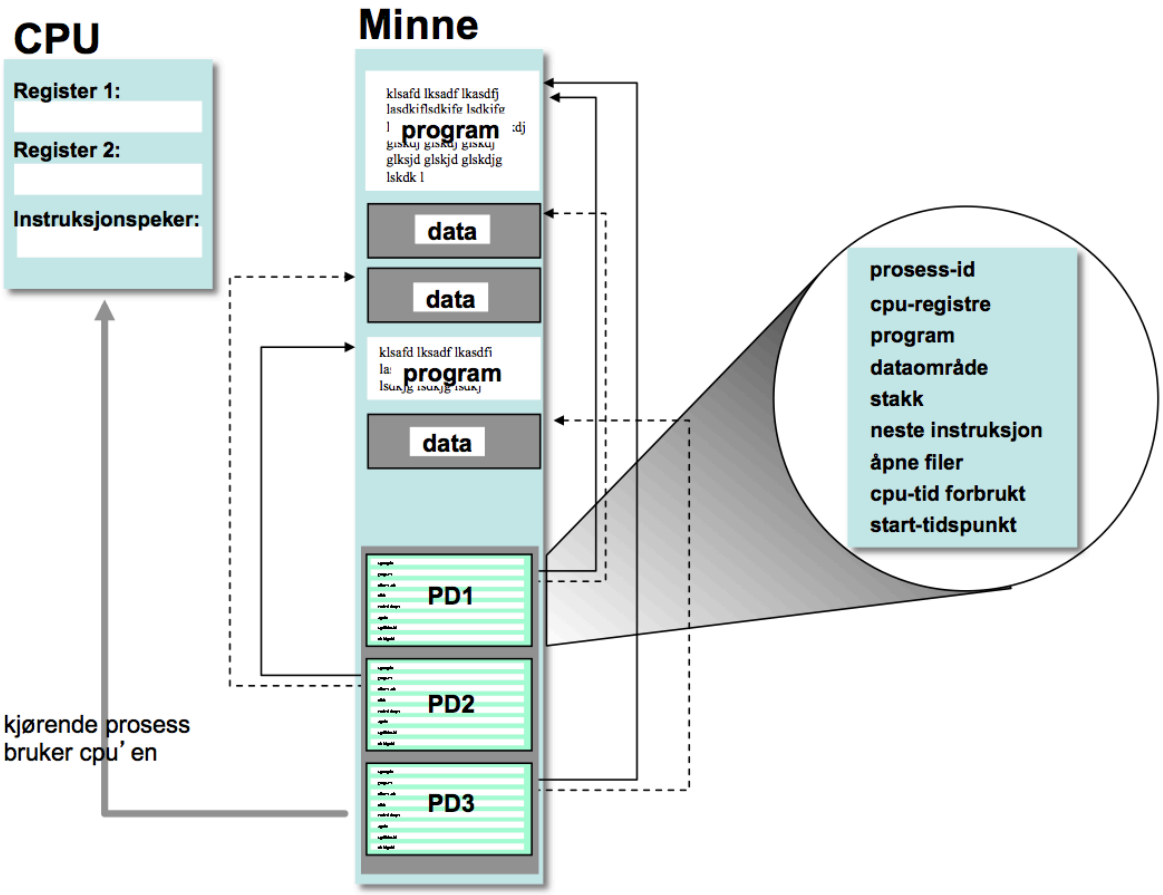
I et operativsystem basert på flere samtidige prosesser er ikke dette noe problem. Hver gang det kommer et avbruddsignal fra en i/o-enhet, kan den prosessen som kjører akkurat nå suspenderes og den prosessen som er ansvarlig for akkurat denne i/o-enheten kan kjøre et lite øyeblikk på cpu-en. Input-prosessen ovenfor er et godt eksempel på en slik prosess som kontrollerer en i/o-enhet. Hver gang noe trykker ned en tast på tastaturet vil denne prosessen kjøre et lite øyeblikk på cpu-en.

1.6. Prosessdeskriptor

Foran så vi at operativsystemet hele tiden skifter på hvilke prosesser som skal få bruke cpu-en. Vi har også sett at operativsystemet må ha en mekanisme for å lagre unna status-informasjon om hver prosess slik at det er lett å restarte prosessen når den får tilbake cpu-en igjen seinere. Poenget er at tilstanden slik den var idet prosessen ble fratatt cpu-en skal gjenopprettes når prosessen igjen får adgang til cpu-en

Nedenfor ser du en figur som illustrerer hvordan operativsystemet greier å gjemme unna statusinformasjon om hver prosess. Operativsystemet bruker en såkalt *prosessdeskriptor* for hver prosess. Her lagres all relevant informasjon om prosessen, så som innholdet på cpu-registrene, adressen til programkoden og til dataområdet, adresse til neste instruksjon som skal utføres, etc.

På figuren er det vist tre prosessdeskriptorer (PD'er) nederst i minnet. Det er også vist hvordan det fra hver slik prosessdeskriptor er adresser som peker til prosessenes program-kode, dataområde, etc.



Hver gang det skjer et skifte av prosess som skal bruke cpu-en, vil først innholdet til cpu-registrene kopieres til PD'en til den aktuelle prosessen. Disse skal brukes neste gang denne prosessen får adgang til cpu-en. Før denne nye prosessen kan starte opp må innholdet fra PD'en til denne prosessen legges inn i cpu-registrene.

Hver gang det skiftes på hvilken prosess som kjører skjer det en slik lagring av status-informasjon fra cpu til PD for avtroppende prosess, og en innlegging av informasjon fra PD til cpu for påtroppende prosess.

I figuren over ser vi også at to av prosessene bruker akkurat samme programkode. Dette kan f.eks være programkoden til en editor. Legg også merke til at de har forskjellig dataområde. Dette er jo selvsagt nødvendig da disse prosessene sannsynligvis tilhører to forskjellige brukere, og det er ikke ønskelig at teksten som disse to skriver, skal blandes sammen i ei "suppe". Prosessene skal altså ha egne dataområder, selv om de deler på programkoden.

1.7. Prosesstilstander

Det er viktig å få med seg at alle de andre prosessene, de som ikke bruker cpu-en i øyeblikket, også finnes i datamaskinsystemet. Noen venter f.eks. på data fra brukeren, andre venter på datablokker som skal hentes på disken. Alle disse prosessene sier vi ligger i i/o-kø. De venter altså på at etterspurte data skal bli tilgjengelig.

Når dataene er tilgjengelig flyttes prosessene over fra i/o-køen til cpu-køen der de ligger og venter på å få kjørt på cpu-en. Prosesser som ligger i cpu-køen har fått tilgang til ressurser utenom en, nemlig cpu-en. Det er ingen vits i å ha prosesser som venter på inndata (f.eks fra

tastatur) liggende i cpu-køen. Selv om de får cpu-en kan de ikke benytte seg av den før de får overført dataene sine fra f.eks tastaturet.

1.8. Prosesser og ressurser

Foran har vi sett generelt på prosess-begrepet for å kunne forestille oss hva en prosess er for noe. Bildet er ikke komplett før vi også bringer inn *ressursbegrepet*. Et program er nemlig ikke en prosess før den har knyttet til seg en del ressurser.

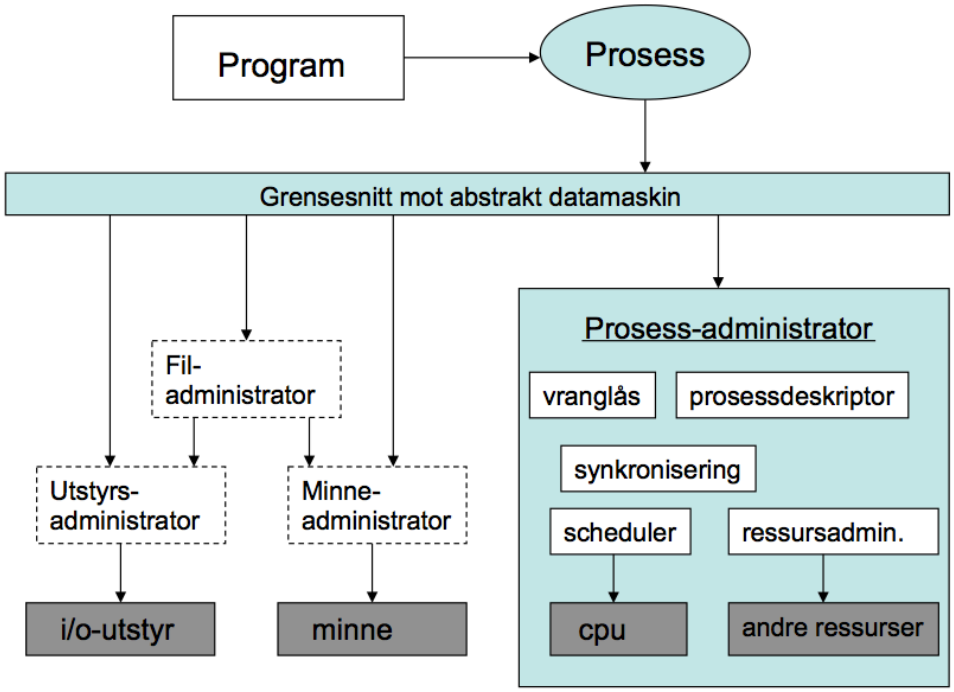
En prosess spesifiserer sitt ressursbehov til prosess-administratoren i operativsystemet. Prosess-administratoren sørger for at prosessen får de ressursene den spør etter, og at prosessen blir kjørt når det er dennes tur. Uten ressurser er det jo bare snakk om et program (dvs programinstruksjoner), og ikke en kjørende prosess.

Prosess-administratoren er den delen av operativsystemet som skaper en modell av hver ressurs, dvs en datastruktur som representerer ressursen og programkode som opererer på ressursens datastruktur. Prosess-administratoren kan via denne modellen tilby prosessene ressurser, deriblant cpu-en. Vi har sett på dette allerede. I korthet går det ut på at prosessene får cpu-en etter tur, og når prosesser ber om en ressurs, og den ikke er tilgjengelig, må de vente i kø. Det er prosess-administratoren sin oppgave å holde orden på alt dette. Mye av operativsystemet sin kompleksitet dreier seg faktisk om å holde orden på de mange prosessene som vil allokere til seg et begrenset antall ressurser på samme tid.

Grunnlaget for at prosess-administratoren kan gjøre jobben sin er følgende:

- *Ressurser* og *prosesser* er representert av datastrukturer, og tilhørende programkode som opererer på disse datastrukturene.
- Metoder for deling av cpu, såkalt *scheduling*.
- *Synkronisering* av prosesser.
- Metoder for å håndtere *vranglås*.

Figuren nedenfor viser en del viktige relasjoner i sammenheng med prosesser. Det er programmet som blir til en prosess når det får tak i ressurser. Prosessen ser en abstrakt modell av datamaskinsystemet, en modell som er skapt av operativsystemet. Figuren viser også de fire sentrale administrative enhetene i operativsystemet: 1) prosessadministrator, 2) filadministrator, 3) minneadministrator, 4) utstyrsadministrator. Vi kommer til å se nærmere på hver av disse seinere.



I boksen til høyre i figuren ovenfor ser vi de oppgavene som prosess-administratoren må forholde seg til. Den har en prosessdeskriptor for hver prosess for å kunne administrere prosessene. Den må forholde seg til vranglås og synkronisering, og den må kunne skifte på hvilken prosess som bruker cpu-en. Prosess-administrator har også som oppgave å fordele andre ressurser (dvs utenom cpu-en) til prosessene etter hvert som de blir etterspurt. Disse ressursene kan være ei fil på disken, utskrift til printer, adgang til et dataområde etc. Poenget er at prosessadministrator har et dataområde som representerer hver av disse ressursene, og kan derfor administrere disse i forhold til at flere prosesser spør om å få bruke disse samtidig.

1.9. Prosess-modellen

Vi har snakket litt om prosess-modellen tidligere, i kapitlet *Prosessdeskriptor* på side 6. Nå skal vi si litt mer om den modellen av prosessen som ligger rundt prosessdeskriptoren.

Når operativsystemet starter opp for første gang lages det en såkalt første prosess (init-prosess). Det finnes jo ingen mekanismer for å opprette prosesser på det tidspunkt, så her er det snakk om å ”håndlage” en prosess. Når operativsystemet er oppe og kjører, har operativ-systemet en funksjon tilgjengelig som kan lage nye prosesser på grunnlag av en eksisterende prosess. I Linux heter denne funksjonen *fork*. Prosessadministrator lager da nye prosesser etter behov ved å bruke fork-funksjonen, som i all sin enkelhet lager en kopi av prosessdeskriptoren til en allerede eksisterende prosess. For å kunne referere til prosessdeskriptoren, og dermed også til prosessen, følger det også med en såkalt *prosess-identifikator*. Denne kalles ofte bare for *pid*. Den brukes bl.a når du skal ta livet av prosessen.

En prosess består av følgende deler:

- Et *program* som definerer hva prosessen skal gjøre. Det er altså en beskrivelse av oppførselen til prosessen.
- Et *dataområde* som prosessen opererer på. Her finnere man de data prosessen har ved start, og de resultatene som prosessen produserer.
- Et sett av *ressurser* som prosessen trenger for å kunne kjøre.

- En *prosessdeskriptor* for å holde rede på progresjonen under kjøringen av prosessen. Via prosessdeskriptoren får en tilgang til prosessens dataområde, programkode, stakkområde, kopi av innhold på cpu-registre, info om åpne filer etc.

Etter at prosessdeskriptoren er opprettet, er det tid for å allokere ressurser til prosessen, en jobb som utføres av prosess-administratoren. En åpenbar ressurs som prosessen trenger er plass i minnet til programkode, data og stakk.

Når alt dette er gjort er en ny prosess klar til kjøring på cpu-en. Cpu-en er nemlig den siste ressursen man får tildelt. I en fler-prosess-datamaskin er det som oftest mange prosesser som er klare til å kjøre. Det betyr at de har fått tak i alle ressursene, men mangler kun cpu-en. Prosessene må stå i en cpu-kø for å få anledning til å kjøre på cpu-en, dvs bruke ressursen cpu. Det finnes nemlig bare en cpu tilgjengelig på de fleste datamaskinene.

Prosessen som bruker cpu-en i øyeblikket, må gi fra seg cpu-en før andre kan slippe til. Det skjer f.eks når

- Kjørende prosess skal utføre en i/o-operasjon.
- Prosess-administrator fjerner kjørende prosess fra cpu-en fordi den har kjørt lenge nok. Tildelt tidskvant er med andre ord utløpt.

Når prosess-administrator skifter til ny prosess på cpu-en må all nødvendig informasjon om avsluttende prosess tas vare på i prosessdeskriptor. Tilsvarende må statusinformasjon for den nye prosessen hentes fra prosessdeskriptoren og legges inn i cpu-registrene.

1.10. Ressursmodellen

Her er det snakk om å allokere ressurser til prosesser etter hvert som de ber om det, samtidig som det må holdes øye med den totale ressurssituasjonen i datamaskinsystemet. Det er nemlig bare et begrenset antall ressurser tilgjengelig, og derfor må prosessene stilles i en ressurskø dersom ressursene er i bruk når de etterspørres. Det er *ressurs-administratoren* som utfører oppgavene som har med ressursadministrasjon å gjøre.

Den mest sentrale ressursen som prosessene bruker, er som vi allerede har sett, cpu-en. Den ressursen trengs for at prosessene skal kunne kjøre. Underveis i kjøringen kan prosessen få behov for andre ressurser, og vil da etterspørre denne ressursen. Dette kan ta litt tid, fordi ressursen kan være opptatt. Det kan være i/o-enheter som er langsomme enheter etc. Derfor vil det alltid skje et prosesskifte på cpu-en ved en slik ressursforespørsel.

Mens en av prosessene kjører på cpu-en vil alle de andre prosessene befinne seg i en eller annen ressurskø. Det kan f.eks være

- Prosessen bruker en i/o-enhet, og venter akkurat nå på at data fra i/o-enheten skal bli tilgjengelig.
- Prosessen venter på at den forespurte ressursen skal bli ledig.
- Prosessen har fått alle de ressursene den har spurt etter, og venter nå bare på cpu-ressursen for å kunne kjøre videre.

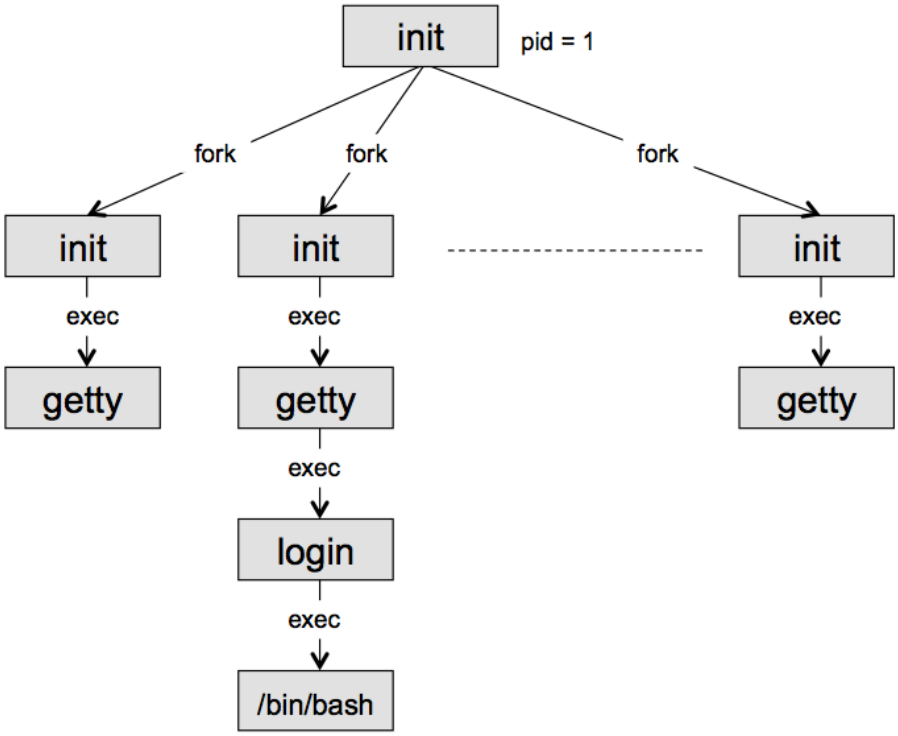
Ressursadministrasjon foregår på lignende måte som prosessadministrasjon. Hver ressurs har en *ressursdeskriptor*, en datastruktur som inneholder data som beskriver ressursen. Ved hjelp av ressursdeskriptoren kan en ressurs allokeres til en prosess ved forespørsel. Når prosessen er ferdig med ressursen frigjøres den fra prosessen og allokeres til en annen prosess, eller den havner på ei liste av ledige ressurser.

1.11. Oppstart av operativsystem

Selve oppstarten av operativsystemet gir oss et godt bilde av hva en prosess er, og hvilken rolle den har i operativsystemet. Derfor skal vi her kort beskrive selve oppstarten (bootingen) av en typisk Linux-maskin.

Det hele starter med at du slår på maskinen. I maskinvaren (i ROM) finnes det noen enkle programsetninger som sørger for at et såkalt boot-program hentes fra disken og inn i minnet. Dette boot-programmet er i stand til å hente inn selve operativsystemet fra disken. Dette er kun en primitiv versjon av operativsystemet. Det finnes jo ingen prosesser der fra før, og den første jobben til operativsystemet er derfor å sette opp et miljø for prosesser og ressurser, og for oppretting av nye prosesser. Straks etter at operativsystemet er lastet inn i minnet, tar det kontrollen over alle maskinvareenhetene, initialiserer diverse datastrukturer og setter status på i/o-enheter. Etter at disse operasjonene er utført har operativsystemet opprettet et abstrakt ressursmiljø som prosessene kan benytte seg av.

Svært tidlig i oppstarten av maskinen blir det også laget en *init*-prosess. Dette er den første prosessen som opprettes på maskinen. Siden det på dette tidspunktet under oppstarten ikke finnes noe funksjoner for å opprette nye prosesser, må denne "lages for hånd". Men når det først er gjort kan prosess-administratoren lage nye prosesser etter behov ved bruk av *fork*-systemkallet, som enkelt lager en kopi av en eksisterende prosess, i dette tilfellet av init-prosessen, som dermed kan kalles alle "prosessers mor".



Init-prosessen fork'er av (dvs lager kopi av) en ny prosess for hver terminal som er koblet opp til maskinen. Det betyr at det oppstår flere nye prosesser i maskinen, alle sammen helt lik init-prosessen, men med sitt eget pid-nummer (pid = prosess-identifikator). I eksemplet vi benytter oss av her er det snakk om en større Linux-maskin med flere tilknyttete terminaler via serieporter. Det er altså ikke snakk om en vanlig personlig Linux arbeidsstasjon på PC som du har på pulten din. Dette er terminaler som brukeren slår på, og bruker til å logge seg

inn på maskinen. Vi bruker dette eksemplet her fordi det gir en god illustrasjon på prosesser og hvordan disse oppfører seg på en Linux-maskin.

Ok, tilbake til eksemplet. Det fork'es altså en ny prosess for hver tilknyttede terminal – en prosess som skal overvåke terminallinja. Dette er vist i figuren ovenfor som den første rekka av init-prosesser. Det er ikke bruk for alle disse prosessene til å utføre init-programkode. Derfor utfører hver av init-prosessene (dvs kopiene) et *exec*-systemkall for å skifte ute sin eksisterende *init*-programkode med *getty*-programkode. Det betyr altså at hver prosess koblet opp mot terminalene nå har skiftet ut programkoden sin til en programkode som er mer tilpasset den oppgaven den skal utføre. Merk at prosessen er den samme, det er bare programmet som er skiftet ut. Pid er fortsatt den samme.

Prosessen *getty* gjør en del operasjoner på terminal og terminal-driver, bl.a setter terminalens hastighet, skriver en velkomstmelding på skjermen og venter deretter på at du skal skrive inn ditt login-navn.

Etter at login-navn er skrevet inn er det ikke lenger bruk for getty-prosessen. Den skifter derfor ut programkoden sin med login-programkode. Prosessen er altså den samme, det er bare programkoden som skiftes ut (og det skal være forståelig for deg som nå har vært gjennom en del forklaringer og eksempler på det med prosesser ☺). Login-prosessen slår opp i passord-fila (dvs fila /etc/passwd) og sjekker om den oppgitte brukeren og tilhørende passord er registrert på denne maskinen.

Dersom login-operasjonen går greit vet operativsystemet nå hvem du er, og setter stående katalog til hjemmekatalogen din. Videre er standardshellet ditt startet opp, og kommando-prompten skrevet ut for å indikere at du er velkommen til å skrive inn en kommando.

Merk at disse tingene som er beskrevet ovenfor skjer for hver terminal som er koblet opp mot en Linux-maskin, og når du logger deg inn via konsollet på din egen personlige Linux-maskin.

Når du logger deg inn på en fjern Linux-maskin via Internett og bruker terminalprogrammet putty er det tilsvarende ting som vist ovenfor som skjer. På Linux-maskinen finnes det en prosess (en såkalt daemon-prosess) som overvåker innkommende forespørsler, og for hver forespørsel som kommer inn til den aktuelle daemon-prosessen forkes det av en ny prosess på lignende måte som vist ovenfor, og slik fortsetter det helt til du er innlogget og kan gi dine kommandoer.

Merk: Vi kommer til å si mye mer om fork og exec seinere når vi spesifikt skal snakke om Linux-prosesser.

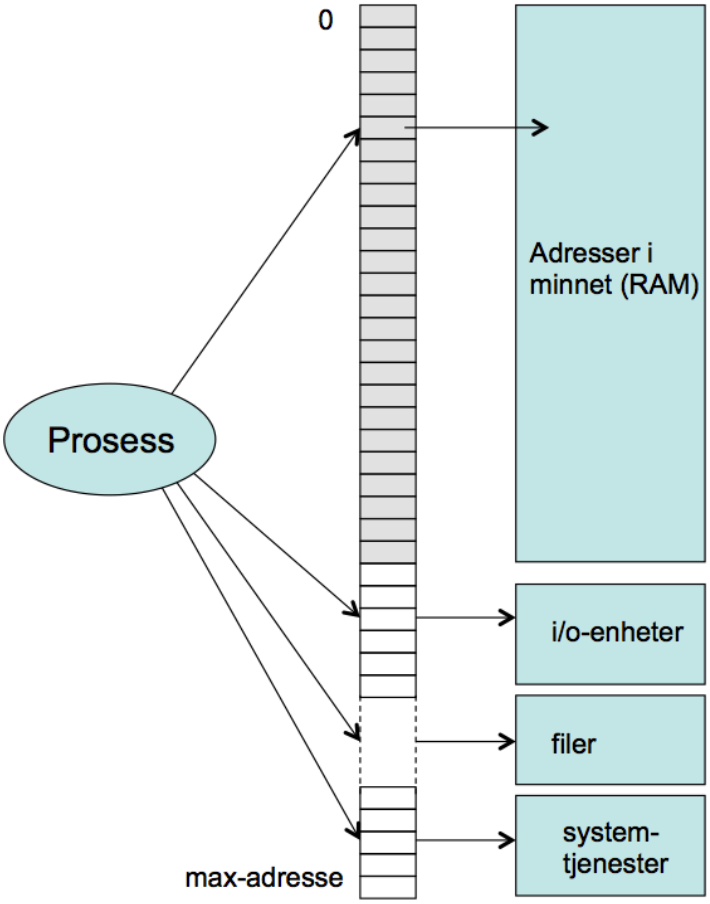
1.12. Prosessens adresserom

Hver prosess har det som kalles et *adresserom*. Dette er en serie av minnelokasjoner som prosessen kan lese fra og skrive til. Adresserommet starter på adresse 0 og går opp til en max-adresse, og inneholder programinstruksjonene til prosessen, samt dataområde og stakk. Videre er det snakk om adresser til registre i cpu og registre som representerer ressurser. Alt dette utgjør prosessens adresserom.

Programmet består som oftest av flere deler. Det er snakk om den delen av programmet som programmereren lager, og det er alle de biblioteksfunksjonene som programmet kaller opp. Det endelige programmet settes sammen av alle disse bitene til et komplett program med adresser fra 0 og opp til en max-adresse. Fra brukerprogrammet (de instruksjonene som er skrevet av brukerprogrammereren) er det referanser til dataområde, til andre deler av

programmet, til funksjoner i biblioteket, og fra biblioteksfunksjoner er det referanse til andre funksjoner i andre bibliotek etc. Alle disse referansene foregår innenfor prosessens adresserom. Referansen er av typen lese data fra et dataområde, skrive data, hopp-instruksjoner til andre deler av programkoden som følge av en if-setning eller en while-løkke.

Størstedelen av adresserommet består av minnelokasjoner i primærminnet (RAM) der programinstruksjoner, data og stakk befinner seg. De andre ressursene som prosessen trenger, kan også nås via minneadresser. Da må det brukes en spesiell teknikk som kalles *memory-mapping* der aksess til f.eks registre i en i/o-kontroller går via ei adresse i minnet.



Når prosessen kjører, foregår det hele tiden operasjoner innenfor adresseområdet. Neste instruksjon hentes fra ei adresse, data hentes fra og skrives til adresser, aksess til i/o-enheter foregår via adresser. Det er selve programmet, dvs programinstruksjonene som foreskriver hvordan dette skal gå for seg.

Den delen av adresserommet som utgjør minne-adressene (RAM) lages når kildeprogrammet oversettes av en kompilator, og dette lenkes sammen med biblioteksfunksjoner som er brukt i programmet (eksempel på en biblioteksfunksjon er printf i C). Da ender du opp med en kjørbar versjon av programmet, også kalt en lastbar modul i operativsystem-sammenheng, fordi dette er programmet som kan lastes inn i minnet og kjøres. I Linux får denne modulen automatisk navnet *a.out* dersom du ikke har angitt annet navn under kompilering/lenking av kildefilen.

Når programmet skal kjøres hentes den lastbare modulen (f.eks med navnet a.out) inn i minnet der cpu dekoder og kjører instruksjonene. Alle adressereferanser foregår nå innenfor prosessens adresseområde. Detaljer omkring hvordan dette foregår, hvordan bare deler av

programmet trenger å ligge i minnet til enhver tid, hvordan programmodulen kan kastes ut av minnet og hentes inn igjen etter behov, må vi utsette til seinere, når vi skal se nærmere på minneadministrasjon.

Et viktig poeng her er at prosessen bare får lov å operere innenfor sitt eget adresseområde av sikkerhetsgrunner. Referanser til adresser utenfor adresserommet blir betraktet som brudd på sikkerheten. Operativsystemet vil da gripe inn og stoppe den aktuelle prosessen.

Men som vi seinere skal se, i noen tilfeller er det aktuelt at prosesser har tilgang til hverandres dataområder. Det kan f.eks skje ved at det opprettes et eget felles dataområde for dette formålet.

1.13. Innholdet til prosessdeskriptoren

I datamaskinen finnes det flere prosesser som kjører samtidig, og som kjemper om å få bruke et begrenset antall ressurser. Det vil ofte føre til at flere prosesser kan komme til å spørre etter de samme ressursene, eller at ressurser er opptatt av annen prosess. Når flere prosesser spør etter ressursen, vil en prosess få tilgang til ressursen, mens de andre må vente til den er ledig igjen. Operativsystemet må holde rede på hvilke prosesser som har fått love til å låne en ressurs, og hvilke prosesser som venter på hvilke ressurser. Til dette formålet finnes prosessdeskriptorer og ressursdeskriptorer.

Ved hjelp av prosessdeskriptoren kan operativsystemet holde rede på status til alle prosessene, dvs om prosessen er kjørende, om den venter på en ressurs, om den venter i cpu-køen etc. I prosessdeskriptoren finner en informasjon om

- Innholdet til cpu-registrene siste gang prosessen var kjørende.
- Prosessens tilstand, f.eks kjørende, ventende etc.
- Prosessens eier.
- Adresseområdet til prosessen.
- Adresse til programkode, dataområde og stakk
- Ressursene som er allokert til prosessen
- Ressurser som den venter på

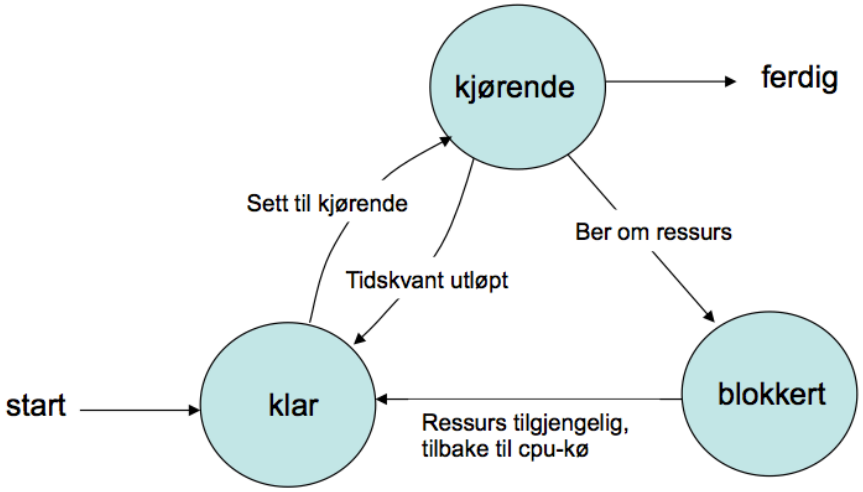
En prosessdeskriptor opprettes i samme øyeblikk som prosessen lages, og den fjernes når prosessen dør.

1.14. Prosessens tilstander

Det er bare en prosess som til enhver tid kan være kjørende prosess, siden det bare finnes en cpu i maskinen (Merk: Selv om vi sier at operativsystemet kjører flere samtidige prosesser, så er det allikevel kun en prosess som kan kjøre om gangen, fordi skifte av prosesser skjer så fort at over litt tid kan det se ut som prosessene kjører samtidig). De andre prosessene som ikke kjører befinner seg enten i cpu-køen og venter på tilgang til cpu-en, eller de befinner seg i en ressurskø og venter på at forespurt ressurs skal bli ledig.

Prosessen kan befinne seg i forskjellige tilstander. Det er prosessadministrator som sørger for at prosessen skifter tilstand. Det skjer f.eks når en kjørende prosess ber om en ressurs eller

den skal utføre en i/o-operasjon. Da kan ikke prosessen kjøre videre på cpu-en, og må derfor inn i en ventekø. Ny prosess hentes da fra cpu-køen og settes i kjørende tilstand.



La oss se litt på hva som kan skje med en typisk prosess. Rett etter at en prosess er opprettet havner den i tilstanden *klar*. Det betyr at den er klar til å kjøre på cpu-en. Den har alle ressurser den trenger unntatt cpu. Flere prosesser kan befinne seg i klar-tilstand. Alle disse organiseres i en kø, som ofte kalles for cpu-køen, fordi de venter på cpu.

Prosessen går deretter over i *kjørende* tilstand. Der blir den inntil ett av tre kan skje:

1. Den tida som er satt av til prosessen (tidskvantet) utløper, og prosessen må tilbake til klar-tilstanden. En annen prosess får kjøre på cpu-en.
2. Prosessen ber om en ressurs. Det vil ta noe tid å få tak i denne, f.eks dersom det er snakk å hente data fra ei fil. Derfor må prosessen gi slipp på cpu-en og gå over i en *blokkert* tilstand, der den befinner seg inntil ressursen er blitt tilgjengelig.
3. Prosessen er ferdig, og avslutter.

Den tredje tilstanden prosessen kan befinne seg i er blokkert. Prosessen er da i en ventekø der den venter på en gitt ressurs som den har etterspurt. Når ressursen er tilgjengelig skifter prosessen tilstand fra blokkert til klar. Legg merke til at prosessen ikke kan gå direkte fra blokkert til kjørende. Den må alltid innom klar-tilstanden og cpu-køen der alle de andre prosessene som er klar for cpu-en befinner seg.

1.15. ”Pre-emptive” prosesser

Moderne operativsystemer er det som vi kaller for *preemptive*. Det betyr at operativsystemet har full kontroll med prosessene. Det er operativsystemet som bestemmer hvor lenge en prosess kan bruke cpu-en. Operativsystemet fjerner en prosess fra cpu-en når tidskvantet utløper og setter inn neste prosess fra cpu-køen.

Motsatsen til preemptive operativsystemer er såkalte non-preemptive operativsystemer, eller også kalt ”cooperative multitasking”. Gamle Windows (fra før Windows 95) var av typen non-preemptive. Det innebar at operativsystemet var prisgitt programmene som kjørte, at disse var samarbeidsvillige. Det betyr igjen at programmereren hadde lagt inn programsetninger som sjekket om andre prosesser ønsket å bruke cpu-en. I så fall ville

kjørende prosess overlate cpu-en til neste prosess. Alle programmer måtte lages på denne måten, dvs. de var høflige og spurte om andre prosesser ønsket cpu-en. Denne måten å organisere operativsystemet på hadde flere ulemper:

- Et program som ”henger seg” eller går inn i evig løkke vil aldri komme fram til den ”høflige” delen av programkoden der det spørres om andre programmer ønsker å kjøre. Det vil derfor aldri skje et prosesskifte. Maskinen låser seg.
- Det er opp til programmereren å lage programmer som er samarbeidsvillige. Dersom dette ikke gjøres vil et program ha cpu-en helt fram til det er ferdig kjørt uten at operativsystemet kan gripe inn.

Operativsystemer som Windows XP og Linux er selvsagt preemptive, og har dermed full kontroll med prosessene. En prosess kan ikke ”ta over styringen” slik som det er mulig i non-preemptive systemer.

Merk også at vi skiller mellom prosesser som kjører i brukermodus og i kjernemodus (mer om dette et annet sted). Inntil ganske nylig var Linux preemptive kun i brukermodus. Det betydde at når prosesser kjørte i kjernemodus kunne de ikke avbrytes. Noe stort problem var ikke dette siden kjernemodus er et kontrollert og ”vennlig miljø”, og prosessene som kjører i kjernemodus vil være samarbeidsvillige.

Men fra og med Linux-kjerne versjon 2.6 (desember 2003) er også kjernen i Linux preemptive. Det betyr at prosesser som kjører i kjernemodus kan avbrytes, og dermed kan andre prosesser kjøre mens en kjerneprosess må vente på et eller annet. Som et resultat av dette responderer Linux mye raskere på hendelser i systemet.

1.16. Tråd-prosesser

1.17. Hva er en tråd?

Vi har tidligere sett viktigheten av at prosesser kjører i isolasjon, dvs ingen prosesser må aksessere dataområdet til en annen prosess. Hver prosess opererer innenfor sitt eget adresseområde, og kun der. Dette er et viktig prinsipp i forhold til sikkerheten i systemet.

Vi skal etter hvert se at vi i kontrollerte former må løsne litt på dette prinsippet. Det er nemlig lurt å la prosesser kunne kommunisere med hverandre fra tid til annen. Det gir nemlig store designfordeler i systemer, dersom man lar prosessene komme hverandre ”tettere inn på livet”.

Vi ønsker altså flere prosesser som deler det samme adresseområdet. Da kan nemlig disse prosessene utveksle data, og dermed kunne kommunisere med hverandre via dataområdet innenfor det felles adresseområdet. Det er her begrepet *tråder* kommer inn. Tråder er separate kjørende programsekvenser som opererer innenfor samme adresseområde.

Så langt har diskusjonen rundt prosesser dreid seg om to sentrale begreper (eller operasjoner). Det er prosessen som

1. Enhet for *ressursallokering*, der forespurte ressurser blir tilordnet prosessen.
2. Enhet for *scheduling*, dvs for utvelgelse til kjøring på cpu-en, slik prosessen framstår som en enhet i cpu-køen.

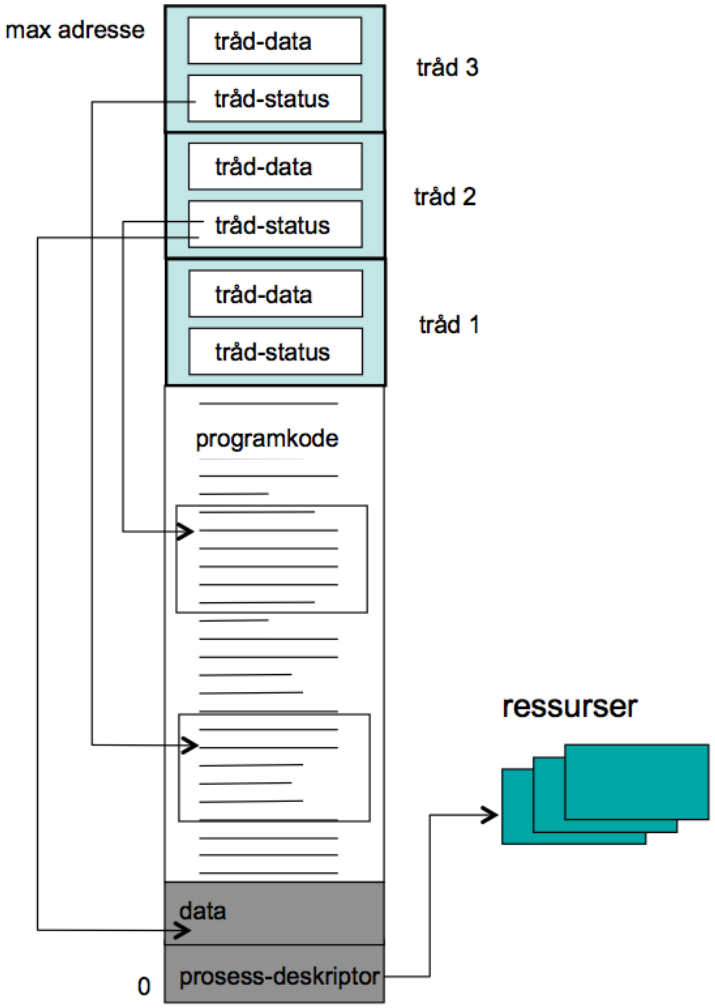
Noen ganger kan det være lurt å skille på disse to tingene, dvs se på scheduling og ressurs-allokering som to helt separate ting. Det er akkurat det som gjøres i forbindelse med tråder (også kalt tråd-prosesser og lettvektsprosesser. Her blir de kalt for tråder).

Det er ikke slik at tråder skal erstatte prosesser. Både begrepet prosesser og begrepet tråder eksisterer ved siden av hverandre.

Prosessen er viktig når det gjelder ressurser og ressursallokering. Hver prosess har knyttet til seg et sett av ressurser til enhver tid. Prosessen har bl.a et adresserom med programkode, dataområde og stakkområde. I tillegg er det snakk om ressurser som åpne filer, barneprosesser etc. Prosessene er mye enklere å administrere dersom ressursene klumpes sammen på denne måten tilhørende hver sin prosess.

Det andre begrepet (i tillegg til ressurser) som man forbinder med prosesser, er selve kjøringen av programinstruksjonene til prosessen. Dette sees ofte på som en ”tråd” (eller et ”spor”) gjennom programkoden etter de instruksjonene som er kjørt. Tråden har en instruksjonsteller som hele tiden viser neste instruksjon som skal kjøre. Den har plass for å lagre unna sine egne data som den jobber med akkurat nå, og den har plass til sitt eget stakkområde. Tråden kjører innenfor en prosess, og har tilgang på alle de ressursene som prosessen eier, men kan allikevel behandles som en separat enhet når det gjelder scheduling. Flere tråder tilhørende den samme prosessen kan meget greit befinne seg i cpu-en.

Det er akkurat dette vi mener når vi sier at *prosessen tar seg av ressursene, men trådene tar seg av selve kjøringen av programkoden.* Flere tråder kan kjøre samtidig på den samme programkoden (på forskjellige steder i programkoden), og de har alle tilgang til prosessens ressurser. Dette skal vi se har store konsekvenser for hvordan vi designer systemene våre, nå basert på tråder.



Figuren ovenfor viser selve prosessen med dens adresseområde fra adresse 0 til max-adresse. Nederst er prosessens dataområde og prosessdeskriptor. Prosessens programkode er vist som et stort område på midten, og øverst er vist tre forskjellige tråder. Tråden har hvert sitt område for å lagre unna status og lokale data.

Tråden trenger et statusområde for å lagre unna tilstanden til tråden idet den må gi fra seg cpu-en. Denne statusinformasjonen inneholder bl.a data fra cpu-registrene i det øyeblikket tråden måtte gi fra seg cpu-en, peker til dataområde og stakkområde, samt en peker til programkoden. Denne statusinformasjonen er nødvendig fordi tråden er enhet for scheduling, som betyr at det er tråden som må inn og ut av cpu-en for hver gang den skal kjøre.

Det er prosessen som allokerer til seg ressurser som minne og øvrige ressurser. Trådene har full tilgang til disse ressursene siden de er en del av prosessen.

Legg også merke til at alle trådene har tilgang til hele prosessens adresseområde, og dermed også har tilgang til programkoden og til dataområdet. Alle trådene har altså full tilgang til hele adresseområdet til prosessen; hver tråd kan lese, skrive til og eventuelt fjerne alt innholdet i det felles dataområdet.

Det er altså ingen beskyttelse mellom tråder slik som det er mellom prosesser. Det er umulig å bygge inn beskyttelse mellom tråder, og det er heller ikke ønskelig fordi det er akkurat dette vi ønsker. Til forskjell fra ulike prosesser, som ofte opererer i ”fiendeland”, så er trådene innstilt på å samarbeide. De er jo skapt av den samme programmereren, og skal løse oppgaver innenfor samme område. Trådene må med andre samarbeide; de er ikke ute etter å konkurrere med hverandre.

På samme måte som en vanlig prosess, er også trådene i forskjellige tilstander som kjørende, klar og blokkert. Flere tråder tilhørende den samme prosessen kan ligge i cpu-køen på samme tid. Trådene er uavhengige av hverandre i forhold til cpu-en.

Legg også merke til at trådene kan kjøre på forskjellige steder i programkoden. Flere tråder kan også kjøre akkurat den samme delen av programkoden.

1.18. Bruk av tråder

Hovedgrunnen for bruk av tråder er at applikasjonen vil kjøre flere aktiviteter samtidig, og at alle disse aktivitetene har tilgang til det samme adresseområdet. Dette er til forskjell fra vanlige prosesser der hver prosess har sitt eget separate adresseområde utilgjengelig for de andre prosessene. Innføring av tråder gjør at selve programmeringen av avanserte applikasjoner blir enklere, og man kan oppnå helt andre ting enn man ellers ville kunne få til.

En annen viktig grunn for bruk av tråder er at de ikke har knyttet til seg noen ressurser, og dermed er det mye enklere og mindre krevende å opprette nye tråder. Her snakker man om størrelser på opp til 100 ganger raskere. Dette er viktig i f.eks en filtjener der nye tråder opprettes hele tiden, for hver ny forespørsel som kommer til filtjeneren. Her ville de ikke vært tjenlig å opprette nye prosesser for hver forespørsel. Det ville nemlig tatt alt for lang tid. Videre ville det heller ikke vært tjenlig å bare operere med en prosess som skulle betjene alle forespørslene. Da ville responstida bli alt for lang, fordi prosessen ville være utilgjengelig mens den holdt på å betjene en av forespørslene.

Eksempel 1: Stavekontroll i tekstbehandler

Samtidig som du skriver inn tekst f.eks i Word foregår det en stavekontroll. Ord som ikke finnes i ordlista merkes med en rød bølgestrek. Denne aktiviteten foregår hele tiden samtidig mens du skriver. Her snakker vi om tråder som opererer mot det samme dataområdet. Trådene det er snakk om er en tråd for å ta i mot teksten som skrives inn, og en tråd som sjekker

ordene mot ei ordliste. Begge trådene har tilgang til det felles dataområdet som er teksten du skriver inn.

Eksempel 2: Filtjener

Tenk deg en filtjener som står i et lokalnett og betjener hundrevis av PC-er. Det kommer nye forespørsler hele tiden. Tenk deg at en prosess skulle ta seg av disse forespørslene. Hver gang det kommer en forespørsel må denne prosessen ta affære, dvs ta i mot forespørselen, finne fram på disken til aktuell fil, vente til aktuell diskblokk er hentet fram, og til slutt overføre de etterspurte dataene. Mens dette har foregått har alle nye innkommende forespørsler køet seg opp, og brukerne vil sannsynligvis oppleve systemet som tregt.

Med bruk av tråder vil dette skje mye smidigere. For det første vil systemet respondere mye raskere fordi en forespørsel besvares med en gang ved at en ny tråd opprettes for hver forespørsel, og for det andre at designet av systemet er mye enklere (og lettere å programmere) når det brukes tråder.

Husk at tråder er svært kjappe å opprette. Derfor er responstida kort. Med mange aktive tråder som betjener hver sin forespørsel vil cpu-tida bli mye mer effektivt utnyttet. Ventetida mens en diskoperasjon foregår brukes til å kjøre andre tråder.

Eksempel 3: Nedlasting av filer via nettleser

Besøk på nettsteder kan føre til at du laster ned filer. Det kan være pdf-dokumenter, pakkede zip-filer, musikkfiler etc. Nettleseren tillater at du laster ned flere filer samtidig. Da opprettes det en tråd for hver nedlasting, og hver tråd vises som et statusvindu for nedlastingen.

Eksempel 4: Ompaginering

Tenk deg at du jobber i Word med et stort dokument på flere hundre sider. Du sletter et avsnitt helt i starten av dokumentet, og ber deretter Word om å flytte skrivermerket fram til f.eks side 300. Siden du nettopp har slettet et avsnitt er det ikke så lett for Word å vite hvilken setning den skal stoppe ved på side 300. Dokumentet må nemlig ompagineres etter en slik sletting av et avsnitt. Før Word kan bringe deg fram til side 300 må den altså gjennom en tidkrevende ompaginering.

Men med tråder kan dette foregå mye mer effektivt. Etter at du har slettet avsnittet starter Word umiddelbart en tråd som ompaginerer hele dokumentet i bakgrunnen. Når du deretter ber om å bli flyttet fram til side 300 er ompagineringen ferdig, og flyttingen fram til side 300 skjer umiddelbart.

Her er det altså to tråder aktive på samme tid, en tråd som tar imot tegn fra tastaturet og en som ompaginerer. I tillegg kan vi ha en tråd som lagrer sikkerhetskopier av dokumentet med jevne mellomrom, og en tråd som driver stavekontroll av dokumentet.