

## Forelesningsvideo:

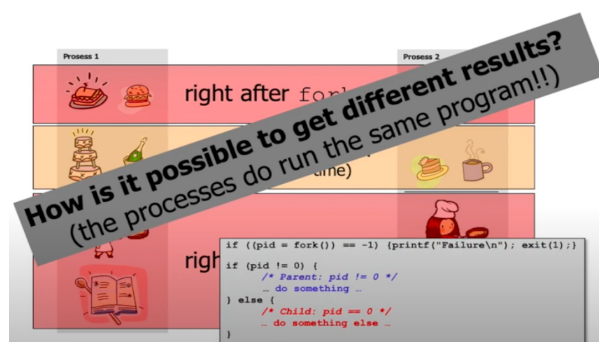
[https://www.youtube.com/watch?v=CZ4f4mMsbVs&list=PLCWgwIU3Ksi6n9i\\_z61pVLpG8SbLtCyAy&index=22&ab\\_channel=IN2140](https://www.youtube.com/watch?v=CZ4f4mMsbVs&list=PLCWgwIU3Ksi6n9i_z61pVLpG8SbLtCyAy&index=22&ab_channel=IN2140)

## Processes/Prosesor

En prosess vil si eksekveringen av et program, dvs eksekvering av alle instruksjoner som ligger til rette for det gitte program. Kan tenke på at instruksjoner for å lage en matrett. En prosess vil anvende cpu-sykler(bruker ressurser) til å utføre instruksene vist i programmet sitt. Husk tråder fra IN1010, disse lettvekts-programmene som utfører en gitt arbeidsoppgave. Dette er den samme tankegangen ved prosessor hvor vi har flere prosesser som eksekverer et gitt program. En prosess kan lage en annen prosess ved å bruke

### **pid\_t fork(void) system call (see man 2 fork):**

- Lager en duplikat av prosessen som ble kalt og inkluderer en kopi av den virtuelle minneadressen, "open file descriptors" osv.
- Begge prosessene vil fortsette å kjøre i parallell
- Returverdien ved systemkallet gjør at vi kan skille mellom en prosess som lager en prosess(prosess1) og prosessen som ble laget(prosess2).
- If parent: child prosess'PID when successful, -1 otherwise
- if child: 0 (if successful - if not, there will not be a child altså prosess2)
- Vi ser at dette ligner på tråder, at vi har parallelle tråder som utfører en gitt oppgave likt.



La oss si at vi har en prosess1 som skal lage en prosess2 gjennom fork() som vist over. Vi har fått en kopi/mellomtilstand av funksjonene som er blitt kjørt så langt ved prosess1, til prosess2. Prosess2 vil da igangsette en funksjon(lag kake()), deretter prosess1 utfører sin funksjon(lag storkake()) også gjør prosess2 en annen funksjon (lag kaffe()) og tilslutt prosess1 utfører sin funksjon(lag champagne()). Problemet som blir vist her, ligger i at prosessene har samme instruksjoner når det kommer til programmet altså hva vi ønsker å gjøre ved programmet men avhengig av hva "fork"-en returnerer over, så kan vi forskjellige

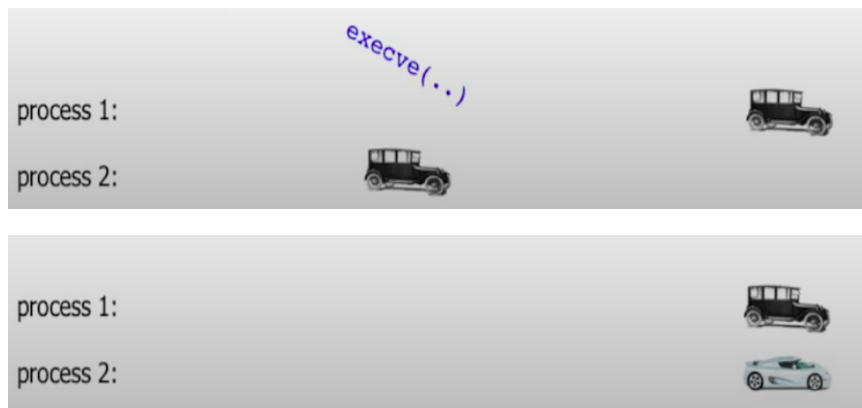
resultater ved disse prosessene. Altså prosess1 kan utføre en instruks imotsetning til prosess2 avhengig av "fork"-en.

### **Program eksekvering**

For å kunne eksekvere et program, så kan vi benytte av følgende system kall:

**int execve(char \*filename, char \*params[], char \*envp[]) system call (see man 2 execve)**

- Ved kall på "execve" så vil programmet lagt i "filename" bli eksekvert gjennom parameterne gitt i "params" og miljøet gitt av "envp".
- Returverdier er enkelt,
  - den vil returnere "-1" ved feil, ellers så
  - blir det ingen returverdier som antyder at programmet ble eksekvert.



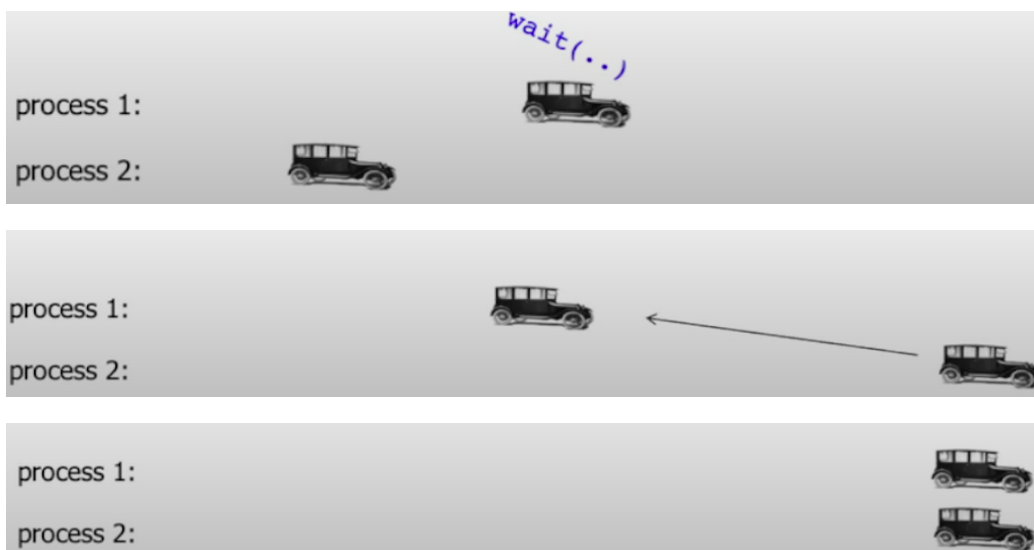
Bildene over er ment for å beskrive hvordan "program eksekvering" fungerer. Vi ser ved første bildet så kjører både prosess1 og prosess2 samme instruks, hvor prosess1 er raskere/lenger unna prosess2. Prosess2 kaller da på execve() og blir en helt annen bil. Dermed når den prosess1 og er i lik linje. Slik fungerer execve() gjennom denne analogien. Slik fungerer program eksekvering ved prosesser hvor vi kan ha to prosesser som har ulike "execve" metoder hvor de kjører forskjellige arbeidsoppgaver hver.

## Prosess venting

For å gjøre at en prosess venter på en annen prosess, kan man benytte av følgende:

**pid\_t wait(int \*status) system call (see man 2 wait):**

- Ved kall på dette system kallet, så vil en gitt prosess vente inntil noen av barne-prosess(prosessene som blir lagd av en annen prosess) er blitt terminert.
- Returnerer
  - -1 hvis ingen barneprosesser ble funnet
  - PID(prosessorid) av det gitte barneprosessoren som ble terminert og setter statusen til prosessoren som ventet, til "status".
- Vi har også andre funksjoner som
  - waitpid som legger til en parameter pid som kan være en spesifikk pid eller noe annet
  - wait3, wait4 - returnerer ressursbruken



Fungerer akkurat som vist i bildet hvor prosess1 kaller på `wait()`, da for prosess2 utføre sitt program. Når prosess2 er ferdig og er terminert, sendes det et signal til prosess1 og da får prosess1 kjøre sitt program. Akkurat som i IN1010 hvor vi hadde tråder som måtte vente i en barriere og inntill alle trådene var ferdig med sin arbeidsoppgave, så kunne main-tråden fortsette å eksekvere programmet vårt.

## Prosess terminering - avslutte en prosess

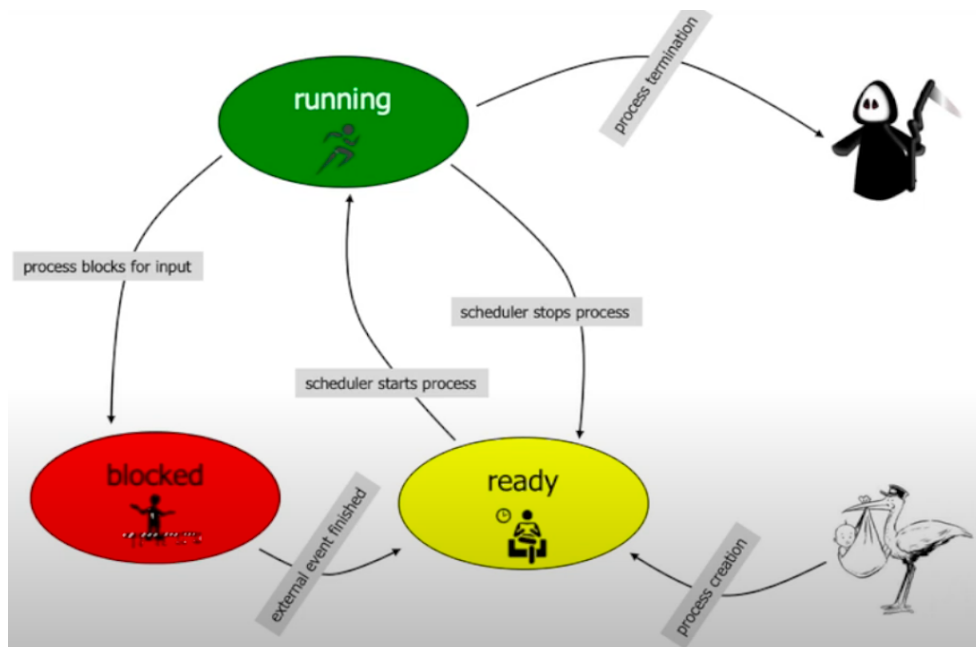
En prosess kan terminere på diverse måter:

- Når det er ingen flere instruksjoner å eksekvere i programmet,
- En funksjon i programmet fullfører med en "return" parameter som returnerer statusverdien.
- Et system kall på "void exit(int status)" som terminerer en prosess og returnerer dets status verdi).
- Et system kall på "int kill(pid\_t pid, int sig)" som sender en signal til prosessen som skal termineres.

Vanligvis en status verdi av 0 betyr suksess altså at prosessen ble terminert riktig, andre verdier antyder feil.

## Prosess tilstander

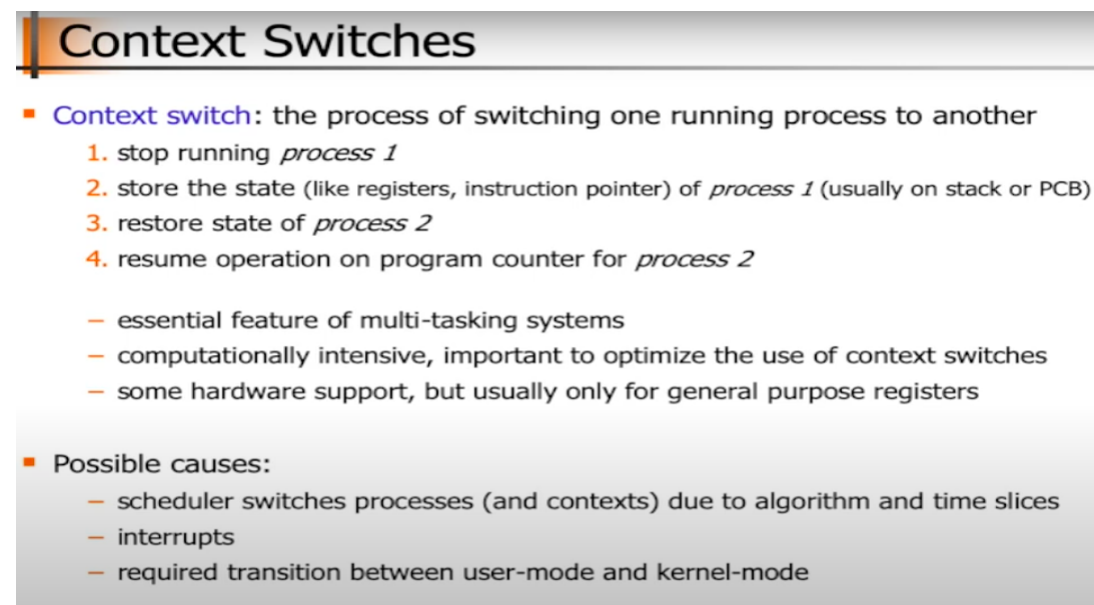
Vi husker i IN1010 at tråder kan ha ulike tilstander: tråden kan gå fra å være i en "kjørende" tilstand til å bli en "ventende" tilstand, som når vi hadde barrierer hvor en tråd måtte vente inntill alle andre tråder var blitt ferdig med å utføre sin oppgave. Prosessorer utnytter cpu'en så det er ikke slik at alle får muligheten til å kjøre sitt program. Hvis en prosessor anvender cpu'en så må noen andre vente på sin tur til de kan utføre sitt program. Bildet under er ment til å beskrive de ulike tilstandene i en prosessor:



I en "running-tilstand" så vil prosessoren kjøre så lenge det er instruksjoner å utføre. Hvis den er satt på vent eller noe lignende til å utføre neste instruksjoner, så er den satt til en "blocked-tilstand" som er knyttet til "process blocks for input". Årsaken er at da for andre prosessorer muligheten til å aksessere cpu'en til å utføre sine programmer. For en

såkal “**ready-tilstand**”, så vil prosessoren bli satt til en slik tilstand hvis den har brukt opp sin tid til å benytte av cpu’en. Dette er knyttet til “**scheduler stops process**”, dvs at prosessoren har nok ikke bli ferdig med programmet sitt så derfor er den ikke terminert, men er klar til enhver tid til å utføre sitt program. Problemet er bare at den har brukt opp tiden den ble tildelt til å utnytte cpu’en og må vente til den har fått lov igjen til å utføre oppgaven sin. Da vil en annen prosessor få lov til å kjøre sin prosess som er knyttet til “**scheduler starts process**”. En prosessor kan være satt til å være “**blocked**” til å bli “**ready**” hvis en ekstern hendelse har tillatt prosessoren å få lov til å eksekvere sitt program. Dette er da knyttet til “**external event finished**”. Vi har også “**process creation**” som beskriver at prosessoren er klar til å kjøre sitt program siden den er blitt lagd, ganske selvfølgelig fra bildet. Tilslutt har vi tilstanden hvor prosessoren har utført programmet sitt og blir terminert som er da “**process terminated**”.

## Context Switches



### Context Switches

- **Context switch:** the process of switching one running process to another
  1. stop running *process 1*
  2. store the state (like registers, instruction pointer) of *process 1* (usually on stack or PCB)
  3. restore state of *process 2*
  4. resume operation on program counter for *process 2*
  - essential feature of multi-tasking systems
  - computationally intensive, important to optimize the use of context switches
  - some hardware support, but usually only for general purpose registers
- **Possible causes:**
  - scheduler switches processes (and contexts) due to algorithm and time slices
  - interrupts
  - required transition between user-mode and kernel-mode

I bildet over så ser vi “Context switch” altså prosessen av å bytte en kjørende prosess til å la en annen prosess få lov til å kjøre. “Context switch” beskriver altså hvordan denne overgangen gjøres og er meget nyttig. La oss si at vi stopper en prosess som ikke er ferdig med å utføre sitt program. Da kan ikke vi bare terminere prosessen, vi må lagre den i et eller annet sted i minne slik at vi ikke mister den gitte tilstanden som prosessoren befinner seg i når det kommer til å eksekvere programmet sitt. Derfor har vi “Context switch” som forklarer hvordan vi utfører denne operasjonen. Dette sees i bildet hvor det beskriver en slik “Context switch” markert gjennom numrene.

## Forskjellen mellom Prosesser og Tråder



### **Tråder (Lettvektsprosessor)**

- Flere kjørende eksekverende enheter i samme prosess som tilsvarer en liten segment av prosessen.
- Dette sees over i bildet hvor trådene har hver sin egen "status", "register", "stack" osv.
- I tillegg til at trådene kan lese hverandres minne imens prosesser ikke gjør det.