Forelesningen: https://www.youtube.com/watch?v=NMBixLEbOEk

Hva er Scheduling?

En **"task"** er en "schedululable" entitet, kan rett og slett si noe som skal kjøre et program som utnytter CPU´en. Det kan være prosessorer, tråder, en pakke overføres til kommunikasjonssystem(pakke til tcp-port) eller en disk forespørsel gjennom filsystemet. I et multi-taskingssystem så kan det hende at vi har en rekke "tasks" som ønsker å utnytte en gitt ressurs(cpu-sykler, hardware-komponenter) parallelt. Vi husker fra prosessorer at ikke alle prosessorer kan aksessere CPU´en, det er kun de som får rett til å gjøre det basert på tilstanden deres. Dette er lik tankegang men bare at vi har noe som tar vare for denne organisering for oss. Dette kalles en **"Scheduler"** som holder orden på disse **"taskene"** og bestemmer hvem som skal få lov til å utnytte en ressurs og bestemmer rekkefølgen av hvilke forespørsler fra enkelte "tasks" som skal utføres ved bruk av **"Scheduling algorithm"**.

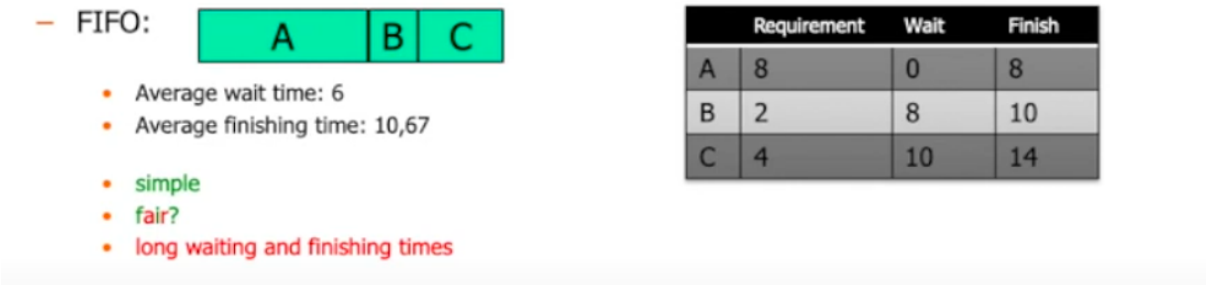
Hvorfor trenger vi Scheduling?

Forestill deg eksempelet vist fra forelesningen, vi har en bil som har innført en såkalt "priority-support" som hjelper til når bilen er i høy hastighet at den ikke rekker å stoppe i tide. Mens vi har en annen bil som ikke har denne "supporten". Det som skjer er at, skulle begge bilene havne i en slik situasjon, så er det bilen med "priority-support" som får stoppet bilen sin i tide. Den andre bilen derimot vil havne i en ulykke. Denne tankegangen gjelder også for "Scheduling" hvor det er viktig at enkelte "tasks" har "deadlines/frister" for å hvor lenge de kan holde på med å utføre sin "task", utnytte CPU´en osv.

I tillegg så er dette knyttet til hvordan vi kan best utnytte ressursene våres effektivt, fordi skal vi først utføre la alle prosessorene utføre sine CPU´operasjoner så I/O`operasjonene etter? Vel det er vel litt dumt siden forestill når den første prosessoren er ferdig med CPU´en, da kan ikke I/O-prosessoren få lov til å utføre sin operasjonen sin vi har lagd "Scheduleren" slik at den skal gjøre alle CPU´operasjonen først så I/O´operasjonene etter. Dette er ikke effektiv og som igjen styrker grunnen til hvorfor "Scheduling" er viktig og spesielt "god Scheduling".

Vi har diverse algoritmer for hvordan vi kan sette "Scheduler" slik at vi unngår tilfeller som beskrevet i avsnittet over hvor "Scheduler" ikke er satt opp på den beste måten mulig. Disse er vist i de kommende sidene.

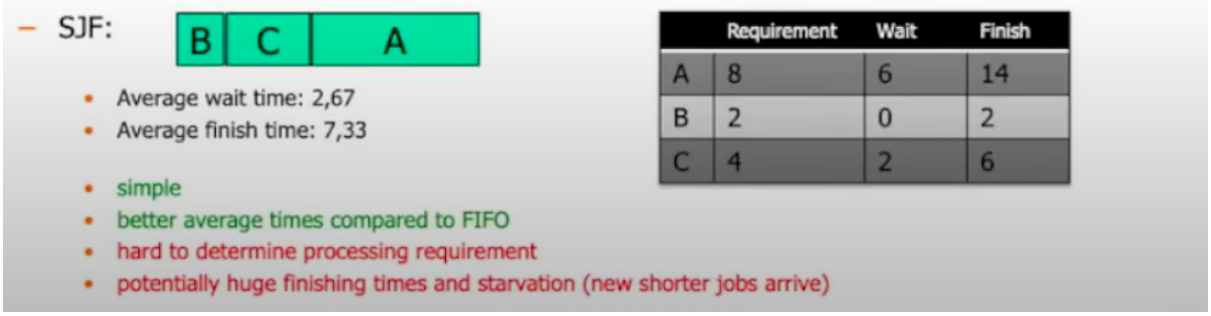
FIFO(First in first out)



	Requirement	Wait	Finish
A	8	0	8
B	2	8	10
C	4	10	14

- Dette er kjent fra tidligere emner hvor “tasks” som blir lagt til først, er også de som utfører sin “task” først.
- I bildet over så ser vi hvordan det fungerer, hvor “task A” utføres først, så kommer “task B” også tilslutt “task C”.
- Dette er en ganske simpel måte sette opp “Scheduler” på, men vi ser også ulemper ved dette. Hva hvis “task A” tok dobbelt så lengre enn både “task B” og “task C” tilsammen, da måtte “task B” og “task C” vente veldig lenge med å få utført sine “tasks”. Dette er da problemet som kan forekomme ved en FIFO-Scheduler.

SFJ (Shortest Job First)



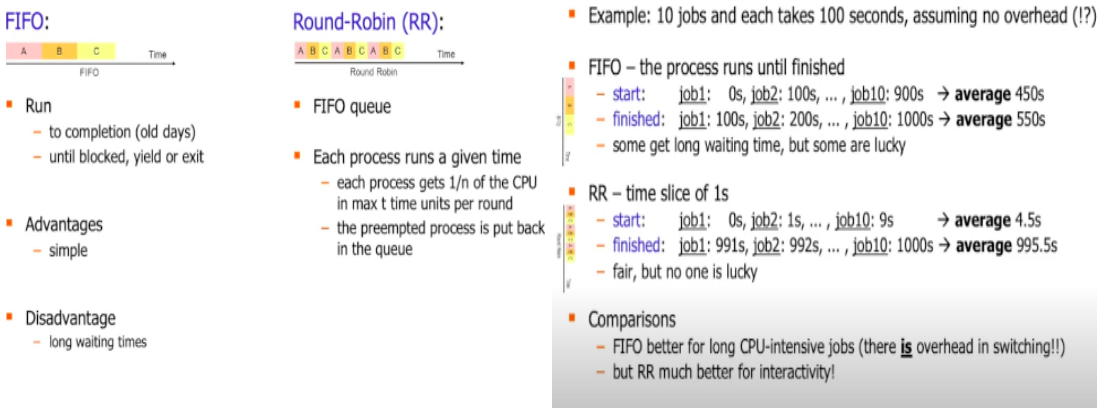
	Requirement	Wait	Finish
A	8	6	14
B	2	0	2
C	4	2	6

- Som navnet sier, så prioriteres de de “taskene” som har kortest tid til å bli ferdig med sin prosess.
- Dette er igjen en ganske simpel måte å sette opp “Scheduleren”
- Men denne har også ulemper gjennom at, hva hvis det kommer nye “tasks” med lavere tid enn de tre over. Da må “B,C og A” vente siden de ikke lenger er de korteste. Så tiden det tar for å fullføre disse kan være enorme altså “task B, C og A” hvis det alltid kommer kortere tidsfullførelser enn de andre.

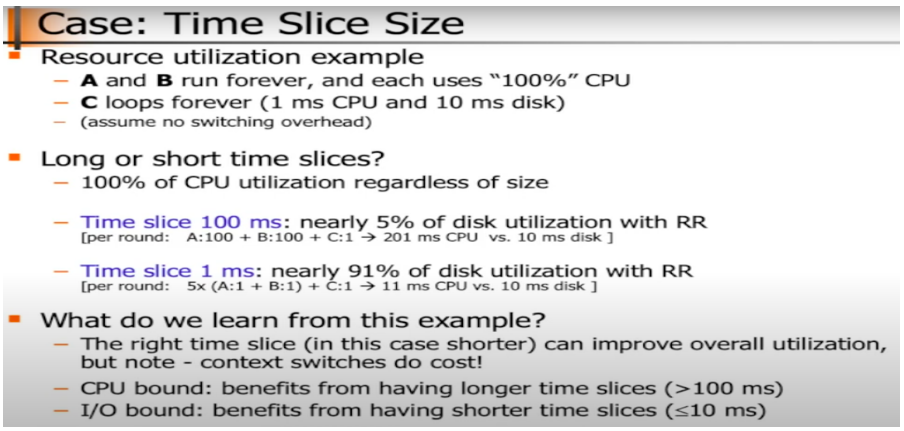
RR(Round Robin)

- Benytter en FIFO kø, men forskjellen er at hver "task" blir gitt en tid for hvor lenge den kan kjøre i.
- Det ser vi i bildet over hvor "task A" kjører i så kort tid, så er det "B", deretter er det "C" og tilslutt så start "A" igjen så repeteres denne sykelen.

FIFO and RR

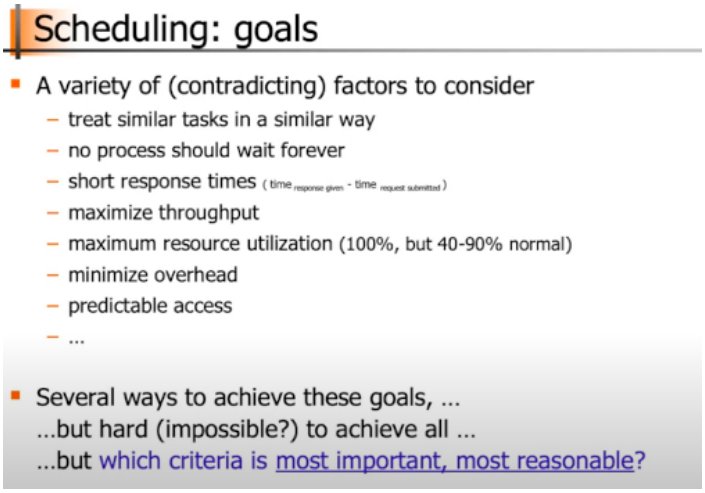


Bildene over er kun ment for å vise forskjeller mellom diverse algoritmer. Ser tidsforskjellen mellom start og slutt for både RR og FIFO til å se viktigheten av de å velge den beste algoritmen. Det er akkurat samme tankegang ved IN2010 hvor vi velger den beste algoritmen etter kjøretid og avhengig av datastrukturen vi er blitt gitt. FIFO er f.eks bedre for lange CPU-intensive arbeid imens RR er bedre for I/O-taskene.



- Her ser vi et eksempel på tre "tasks A,B og C" som bruker CPU´en med RR.
- "A" og "B" vil utnytte CPU´en i 100% som vil si at ved 100 ms så kjører de i 100 ms.
- Dette bildet er ment for å vise fordelene ved å velge den rette RR-tiden for en runde.

Scheduling: Mål



Bildet over viser de ulike målene vi har når vi skal bygge en "Scheduler".

- F.eks så er det viktig at de "tasks" som utfører samme prosess så skal disse "taskene" får samme tid for å utnytte cpu´en f.eks.
- Ingen "tasks" skal vente for lenge, de skal være effektive som når en prosessor er ferdig, så skal neste få muligheten til å utføre sin prosess.
- Miniminere "Overhead" som vil si at du miniminerer antall CPU-sykler slik at ikke CPU´en blir maksimalt tapt på ressurser.

For å finne hvilket mål som er mest hensiktsmessig, så kommer det anpå hva du ønsker å gjøre.

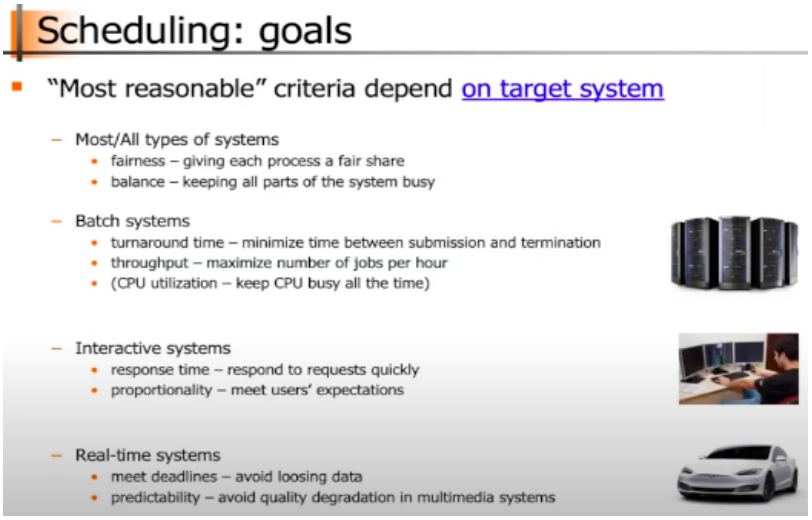
Kjernen/kernel

- Handler det om ressursershåndtering så burde man tilpasse blant annet CPU utnyttelse f.eks

User

- Interaktivitet er vel viktig for bruker, altså når en bruker trykker på en knapp så burder I/O-prosessorene utføres fremfor CPU-prosessorer

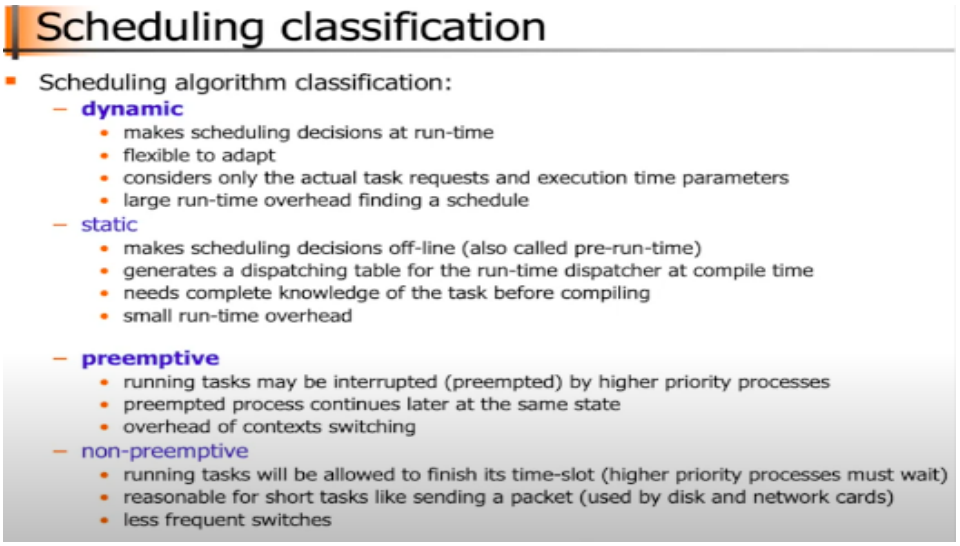
- Predictability i den forstand at det skal være lik tid for hver gang bruker interagerer, ingen forskjeller i tiden ved enhver input



Tilslutt hvilket system du har som mål, har også en betydning.

- Batch systems, vil si knyttet til CPU´en som å utnytte CPU´en blant annet
- Interaktivte systemer ønsker som vist i bildet, at når bruker trykker på høyretast så skal det ikke ta lang tid ved dette forespørselet.
- Real-time systems hvor vi da ønsker å ikke miste data.

Scheduling klassifisering



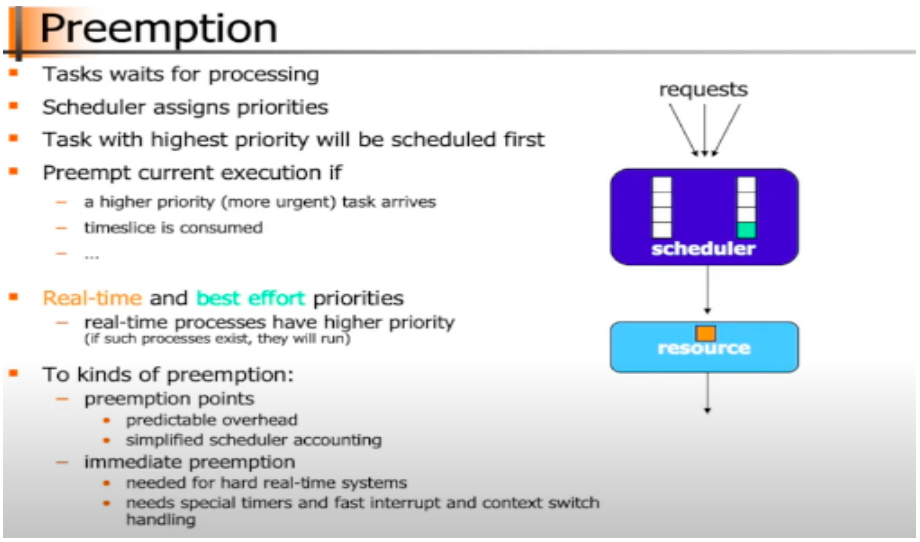
Dette bildet viser hvordan vi kan klassifisere gitte algoritmer basert på hva som passer dem.

- **Dynamisk**
  - Vil si at vi kan være fleksible som at vi kan avbryte gitte "tasks" eller at vi tar valg ved kjøretiden til gitte "tasks".
  - Gjøre avgjørelser mens "tasks" kjører og kan tilpasse gitte valg utifra det.

    - Problemet er det at om vi gjør endringer underveis så kan det forårsake overhead altså at CPU´en utnyttes veldig mye og ressurser også

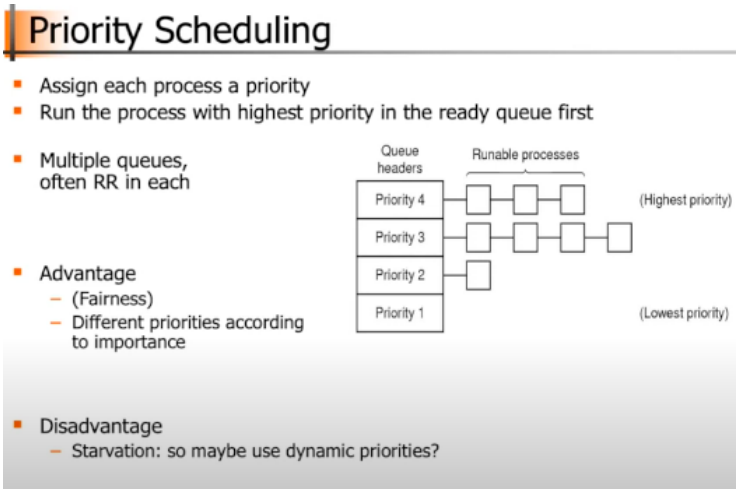
- **Statisk**
    - vil si at vi tar noen valg før vi initierer kjøring av gitte “tasks” og har alt klart på forhånd.
    - Man kan altså ikke være fleksibel som man kan i dynamiske-algoritmer.
- **Preemption**
    - Forestill at vi har en “task1” som kjører også får vi plutselig enn annen “task2” som har høyere prioritet og ønsker å kjøre sitt program. Da vil denne “task2” bli kjørt ettersom den har høyere prioritet enn “task1”.
    - Problemet er at det kan forekommer overhead ved “Context Switching” altså det å håndtere hvordan vi oppbevarer tilstanden til en gitt “task” når en annen “task” skal få lov til å kjøre.
- **Non-preemptiv**
    - Vil si at “tasks” kjører etter tiden hvor tiden en gitt “task” har, bestemmer kjøretiden for den.
    - Noen “tasks” kan ha lang kjøretid, mens andre har kort, er hvordan man kan se på dette.

Preemption



Bildet beskriver hvordan “tasks” gjennom en såkalt “Scheduler-preemption” er satt opp. “Scheduleren” vil sette prioritet for enhver “task” og den “tasken” som har høyest prioritet vil få lov til å kjøre først. Også har vi noen betingelser hvor, si vi har en “task” som har høy prioritet(2), så kommer en annen med prioritet(1), da er det “task” med prioritet(1) som skal kjøre først.

Priority Scheduling



Dette er ganske selvforklarende, en såkalt “Scheduler” hvor vi lar “tasks” kjøre basert på prioriteten deres. Dette benyttes i de fleste tradisjonelle OS som WIndows, Linux, Unix og MacOS.