NTNU

Kunnskap for en bedre verden

Institutt for Datateknikk og Informasjonsvitenskap

# Løsningsforslag for TDT4186 Operativsystemer

**Eksamensdato: 9. august 2016**

**Eksamenstid (fra-til): 09:00-13:00**

**Hjelpemiddelkode/Tillatte hjelpemidler:**

D: Ingen trykte eller håndskrevne hjelpemidler tillatt. Bestemt, enkel kalkulator tillatt.

**Annen informasjon:**

Det ønskes korte og konsise svar på hver av oppgavene.
Les oppgaveteksten meget nøye, og vurder hva det spørres etter i hver enkelt oppgave/deloppgave.
Dersom du mener at opplysninger mangler i oppgaveformuleringene, beskriv de antagelsene du gjør.
Hver av de fem oppgavene teller like mye, og hver av de fire deloppgavene teller like mye.

## Oppgave 1: Operativsystemer (Operating Systems) / Prosesser og tråder (Processes and Threads)

a) Angi klart hvilken / hvilke oppgave(r) operativsystemer generelt bør løse

SVAR:

- Å tilby tjenester til brukere/programmer på en enklere måte enn hva maskinen tilbyr
- Å forvalte ressursene på maskinen på en effektiv måte sett fra systemets side
- Å støtte utvikling over tid av slike tjenester og slik forvaltning fleksibelt og billig

b) Drøft kort om moderne operativsystemer må håndtere andre utfordringer – og i så fall hvilke, enn hva eldre operativsystemer måtte

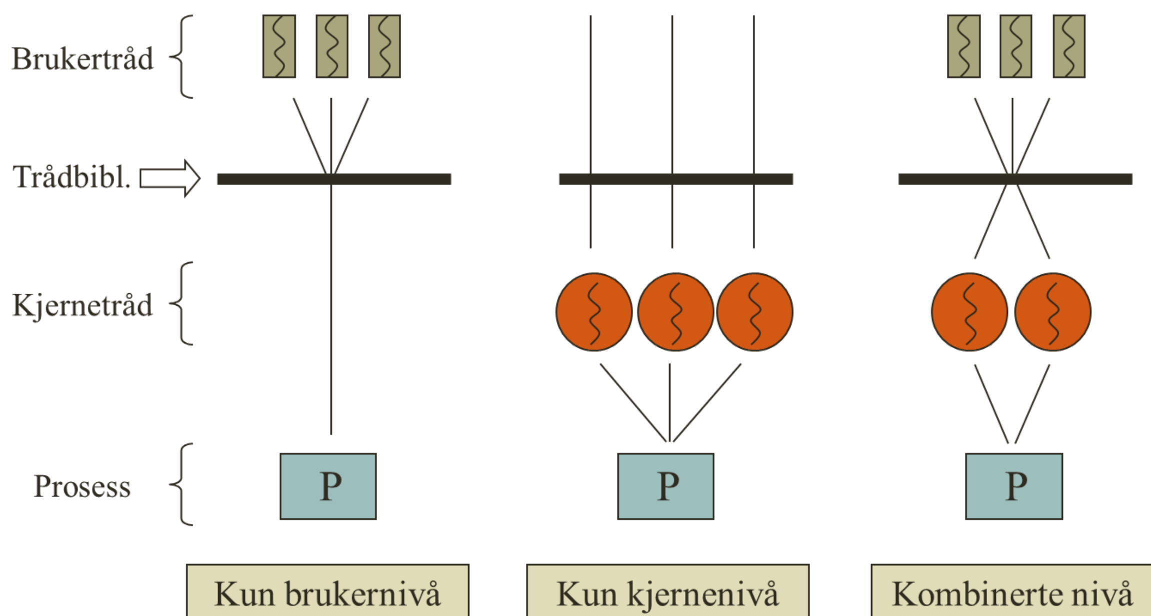
SVAR:

- Tilbudsutvikling:
  Økende spenn mellom ulike maskintyper – mht ytelse (store, små)
- Teknologiutvikling:
  Økende spenn mellom ulike maskindeler – mht ytelse (prosessor, lager, I/O)
- Funksjonalitetsutvikling:
  Fra singelprosessor/singelkjerne til multiprosessor & multikjerne
- Etterspørselsutvikling:
  Fra batchorientering til interaktivitet – og fra nodeisolering til sammenkopling

c) Angi klart minst to måter å implementere tråder på – og sammenlign dem kort mht relevans og ytelse

SVAR:

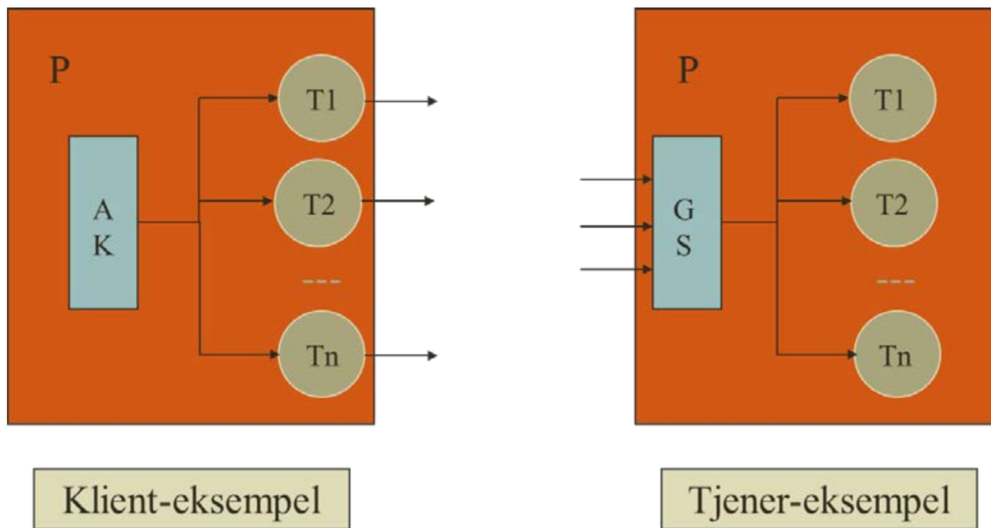
- Brukertråder og kjernetråder samt evt. kombinasjon av begge typer



- Ytelse: Brukertråder gir billige, men også blokkerende systemkall – mens kjernetråder gir dyre, men også ikke-blokkerende systemkall
- Relevans: Brukertråder vil ikke kunne utnytte multiprosessorer, mens kjernetråder vil kunne utnytte multiprosessorer

d) Beskriv og illustrer med tekst og figurer hvordan tråder kan være bedre å bruke enn prosesser i en klient-tjener situasjon (client server context)

SVAR:




Klient-eksempel

Tjener-eksempel

- Klientsiden: En klientapplikasjon kan sette i gang flere parallelle aktiviteter ved å operere med en separat prosess/tråd for hver av dem – og tråder er enklere, raskere og billigere å initiere, utnytte og terminere enn prosesser

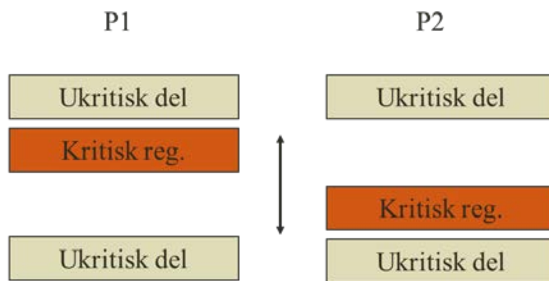
- Tjenersiden: En tjenerapplikasjon kan også respondere med flere parallelle aktiviteter ved å operere med en separat prosess/tråd for hver av dem – og tråder er igjen enklere, raskere og billigere å initiere, utnytte og terminere enn prosesser

## Oppgave 2: Prosess-synkronisering (Process Synchronization)

a) Angi klart hvilket / hvilke problem prosess-synkronisering generelt bør løse

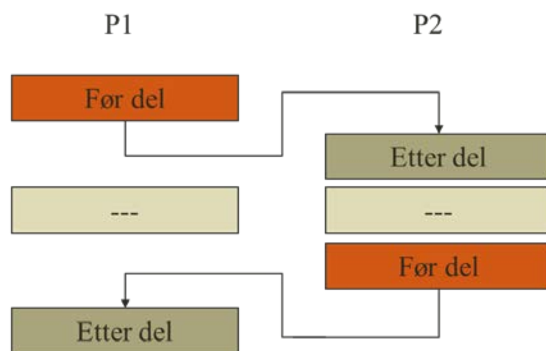
SVAR:

- Deling av felles ressurser – gjennom samarbeid eller konkurranse, må kontrolleres slik at ikke gale / uheldige resultater oppstår
- Kommunikasjon om felles oppgaver – gjennom samarbeid, må også kontrolleres slik at ikke gale / uheldige resultater oppstår




Kritiske regioner må ikke overlappe

*Eks: Bank – Konkurranse / Samarbeid via deling*




Tidsordning kan medføre venting

*Eks: LPC – Samarbeid via kommunikasjon*

b) Drøft kort om moderne prosess-synkronisering må håndtere andre utfordringer – og i så fall hvilke, enn hva tilfellet var i eldre operativsystemer

SVAR:

- Spesielle nye utfordringer - med referanse til Oppgave & Løsning i/på 1b):

- Funksjonalitetsutvikling: Multiprosessorer & multikjerner
- Etterspørselsutvikling: Interaktivitet & sammenkopling

c) Angi klart minst to måter å implementere monitorer (monitors) på – og sammenlign dem kort mht relevans og ytelse

SVAR:

- Vanlig monitor og Mesa-monitor

- I en Vanlig monitor vil den oppvekkende aktør – den som gjør Csignal, temporært tre ut av monitoren slik at den oppvekkete aktør – den som har gjort Cwait, kan fortsette uten å teste betingelsen på nytt igjen først
- I en Mesa-monitor vil ikke den oppvekkende aktør – den som gjør Cnotify, temporært tre ut av monitoren slik at den oppvekkete aktør – den som har gjort Cwait, ikke kan fortsette uten å teste betingelsen på nytt igjen først

- I en Vanlig monitor vil det manglende behovet for retesting av betingelser være billig for applikasjonene, mens de mange resulterende prosess/tråd-skiftene vil være dyrt for systemet
- I en Mesa-monitor vil det eksisterende behovet for retesting av betingelser være dyrt for applikasjonene, mens de få resulterende prosess/tråd-skiftene vil være billig for systemet

- Mesa-monitorer har også en Cbroadcast funksjon – hvor en aktør kan vekke opp mer enn en annen prosess/tråd, i tillegg til en Cnotify funksjon – hvor en aktør kan vekke opp kun enn annen prosess/tråd; Vanlige Monitorer har bare en Csignal funksjon – hvor en aktør kan vekke opp kun enn annen prosess/tråd

d) Illustrer konkret med bruk av monitorer hvordan problemet med de spisende filosofene (dining philosophers problem) kan løses

SVAR:

```
monitor dining_controller;
cond ForkReady[5];            /* condition variable for synchronization */
boolean fork[5] = {true};     /* availability status of each fork */

void get_forks(int pid)       /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork[left])
     cwait(ForkReady[left]);          /* queue on condition variable */
   fork[left] = false;
   /*grant the right fork*/
   if (!fork[right])
     cwait(ForkReady[right]);         /* queue on condition variable */
   fork[right] = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])     /*no one is waiting for this fork */
     fork[left] = true;
   else                          /* awaken a process waiting on this fork */
     csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])    /*no one is waiting for this fork */
     fork[right] = true;
   else                          /* awaken a process waiting on this fork */
     csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]         /* the five philosopher clients */
{
   while (true) {
     <think>;
     get_forks(k);           /* client requests two forks via monitor */
     <eat spaghetti>;
     release_forks(k);       /* client releases forks via the monitor */
   }
}
```

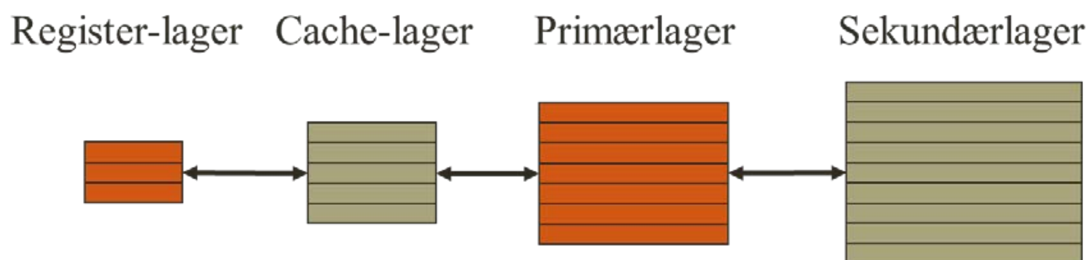
- Begge gafler kan tas samtidig – dvs. uten at en annen filosof kan gjøre noe imens, på grunn av den gjensidige utelukkelsen som automatisk oppnås med en monitor

## Oppgave 3: Lagerhåndtering (Memory Management)

a) Angi klart hvilken / hvilke oppgave(r) lagerhåndtering generelt bør løse

SVAR:

- For å holde en/flere prosessor(er) aktiv(e), trenger vi å holde flere programmer i "lageret" samtidig; Hvordan bør det gjøres – gitt at vi da trenger "flere ulike nivå" av lagerenheter!?




- Når flyttes data nedover?
- Når flyttes data oppover?
- Hvor plasseres data?
- Hvor utbyttes data?
- Hvor mye data trengs?
- Hvor manges data tåles?

b) Drøft kort om moderne lagerhåndtering står overfor andre utfordringer – og i så fall hvilke, enn hva tilfellet var i eldre operativsystemer

SVAR:

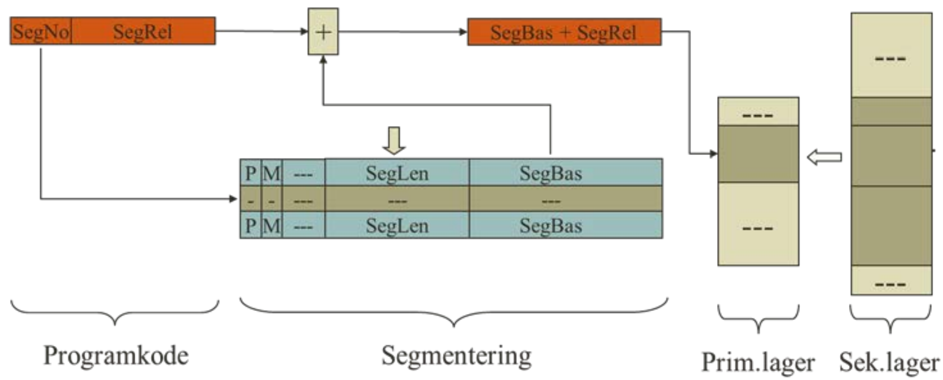
- Spesielle nye utfordringer - med referanse til Oppgave & Løsning i/på 1b):

- Tilbudsutvikling: Ytelsesspenn
- Teknologiutvikling: Ytelsesspenn

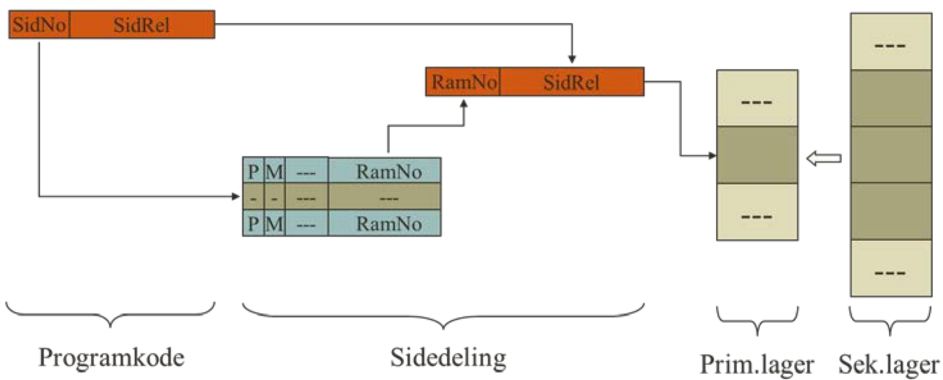
c) Angi klart minst tre måter å implementere virtuelt lager (virtual memory) på – og sammenlign dem kort mht relevans og ytelse

SVAR:

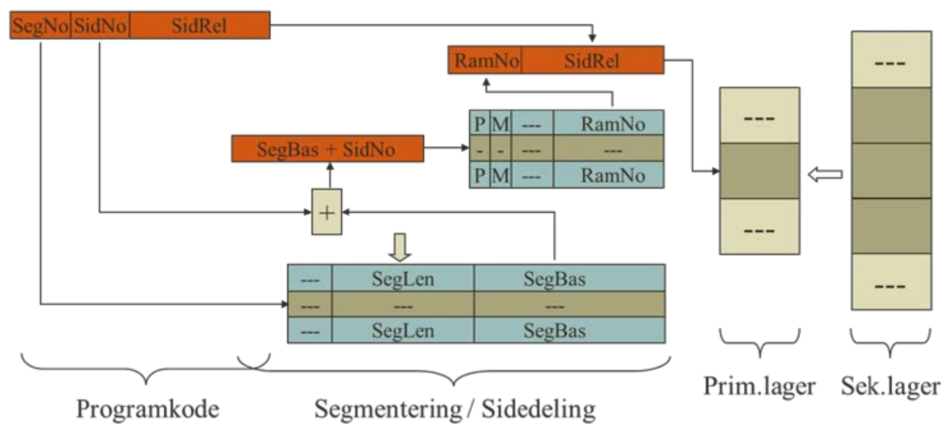
- Segmentering, sidedeling samt kombinering av segmentering og sidedeling




Enkel segmentering: Alle seg., ikke samlet
Virt. lager m/ segmentering: Ikke alle seg., ikke samlet




Enkel sidedeling: Alle sider, ikke samlet
Virt. lager m/ sidedeling: Ikke alle sider, ikke samlet




Kombinasjon av segmentering og sidedeling:
Inkluderer fordeler (og ulemper) fra begge

SegLen: # Sider
SegBas: -> SideTab

- Segmentering:
  Passer med brukeres logiske entiteter – av varierende størrelse
  Gir ekstern fragmentering
- Sidedeling:
  Passer med systemets fysiske rammer – av fast størrelse
  Gir intern fragmentering
- Kombinering av segmentering og sidedeling:
  Fordeler (og ulemper) fra begge
  Mye brukt

d) Beskriv og illustrer med tekst og grafer hvordan ytelsen til sidedeling (paging) avhenger av sidestørrelsen (page size)

SVAR:

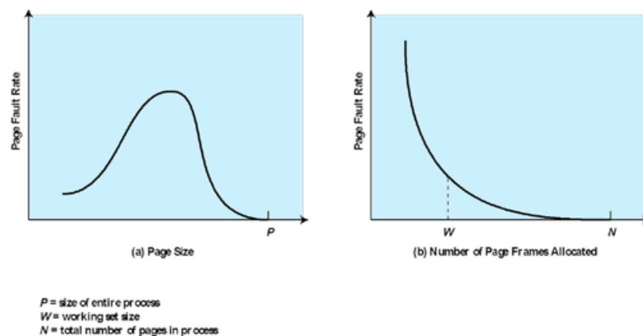



Figure 8.10 Typical Paging Behavior of a Program

- Figur a) reflekterer to forhold:

- Hvis sidestørrelsen blir stor nok, rommes en hel prosess/tråd innen en enkelt side – og vi får aldri sidefeil men har heller ikke plass til så mange prosesser/tråder inne i primærlageret.

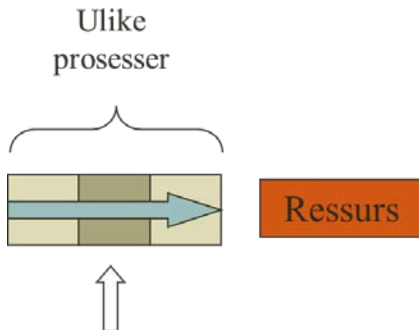
- Mens hvis sidestørrelsen blir liten nok, rommes mange nok deler av angjeldende prosess/tråd i tildelt primærlagerområde – og vi får sjelden sidefeil men har igjen ikke plass til så mange prosesser/tråder inne i primærlageret grunnet enorm overhead til sidetabeller etc.

- Målet blir å finne en sidestørrelse mellom de to ytterpunktene i Figur a) slik at resulterende sidefeilsfrekvens holdes under en angitt grense – som i Figur b) igjen koples til tildelt primærlagerområde.

## Oppgave 4: Prosess-tidsstyring (Process Scheduling)

a) Angi klart hvilket / hvilke problem prosess-tidsstyring generelt bør løse

SVAR:




- Det må avgjøres i hvilken rekkefølge ressurser skal tildeles prosesser / tråder som ønsker tilgang til dem; Skal da noen kunne prioriteres fremfor andre, og skal da noen kunne fratas en ressurs etter at den er blitt tildelt!?

b) Drøft kort om moderne prosess-tidsstyring må håndtere andre utfordringer – og i så fall hvilke, enn hva tilfellet var i eldre operativsystemer

SVAR:

- Spesielle nye utfordringer - med referanse til Oppgave & Løsning i/på 1b):

- Tilbudsutvikling: Ytelsesspenn
- Funksjonalitetsutvikling: Multiprosessorer & multikjerner

c) Angi klart minst tre algoritmer til implementering av prosess-tidsstyring med multi-prosessorer (multi processors) – og sammenlign dem kort mht relevans og ytelse

SVAR:

- Algoritmer (for single- og multiprosessorer; for prosesser – men ikke for tråder)

- Først-inn-først-ut (FCFS)
- Kontinuerlig rundgang (RR)
- Korteste totaltid først (SPN)
- Korteste gjenværende tid først (SRT)
- Høyeste responsforhold først (HRRN)
- Tilbakekopling (FB)

- Relevans & Ytelse

| Algo-ritme | Respons-tid | Gjennom-strømning | Ut-sulting | Rett-ferdighet | |
|---|---|---|---|---|---|
| FCFS | Last-avh. | Last-avh. | Nei | Små / IO: ÷ | ← ¬ Bra |
| RR | God: små | Kvant-avh. | Nei | Balansert | Bra |
| SPN | God: små | Høy | Mulig | Store: ÷ | Bra |
| SRT | God | Høy | Mulig | Store: ÷ | Bra |
| HRRN | God | Høy | Nei | Balansert | Bra |
| FB | Last-avh. | Kvant-avh. | Mulig | Ikke-IO: ÷ | ← Bra ? |

d) Beskriv detaljert rekkefølgespesifikasjonen i og betingelsen for å anvende periodebasert tidsstyring (rate monotonic scheduling) på sanntidssystemer (real-time systems)

SVAR:

- Rekkefølgespesifikasjon (hvor $T_i$ angir tidsperiode for prosess/tråd i)

$$T_1 < T_2 < \ldots < T_N$$

*Høyest prioritet: Minst periode (Må være kjent)*

- Betingelse (hvor $C_i$ videre angir tidsbehov for prosess/tråd i)

$$C_1 / T_1 + C_2 / T_2 + \ldots + C_N / T_N \leq N\,(2^{1/N} - 1) \rightarrow 0.693;\ N \rightarrow \infty$$

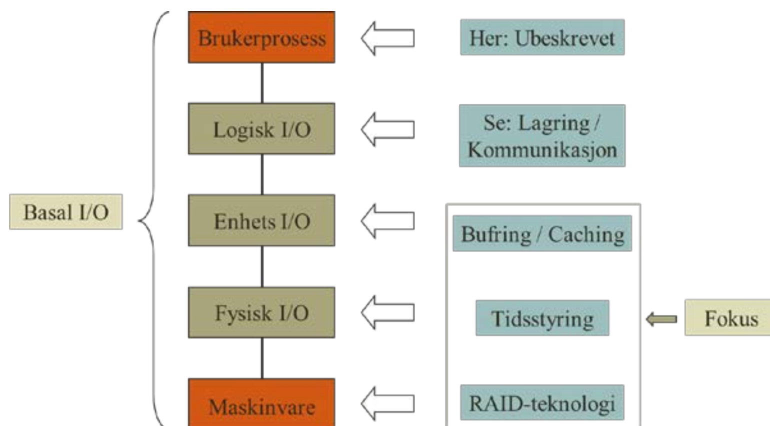
*Sikringskrav: Overdrevent pessimistisk (Tidligste tidsfrist først - EDF: $\leq 1$)*

## Oppgave 5: I/O-Håndtering (I/O Management)

a) Angi klart hvilken / hvilke oppgave(r) I/O-håndtering generelt bør løse

SVAR:

- Ulike oppgaver må løses på ulike nivå – hvor de på nivåene nærmest maskinvaren har fokus i operativsystemsammenheng:




b) Drøft kort om moderne I/O-håndtering står overfor andre utfordringer – og i så fall hvilke, enn hva tilfellet var i eldre operativsystemer

SVAR:

- Spesielle nye utfordringer - med referanse til Oppgave & Løsning i/på 1b):

- Teknologiutvikling: Ytelsesspenn
- Etterspørselsutvikling: Interaktivitet & sammenkopling

c) Angi klart minst tre algoritmer til å håndtere I/O mot disker (disks) – og sammenlign dem kort mht relevans og ytelse

SVAR:

- Algoritmer

- Først-inn-først-ut (FIFO)
- Korteste søk først (SSTF)
- Toveis heis (SCAN)
- Enveis heis (C-SCAN)
- Sist-inn-først-ut (LIFO)
- Høyeste prioritet først (PRIO)
- Blokkvis SCAN (N-SCAN)
- Køvis SCAN (F-SCAN)

- Relevans & Ytelse

| Algoritme | Kommentar |
|---|---|
| FIFO | For rettferdighets fokus |
| SSTF | God ressursutnyttelse |
| LIFO | God lokalitetsutnyttelse |
| PRIO | For sanntids fokus |
| SCAN | Bedre tjenestesnitt |
| C-SCAN | Mindre tjenestevarians |
| N-SCAN | Faktisk tjenestegaranti |
| F-SCAN | Réelt lastavhengig |

d) Beskriv detaljert datainnholdet på og nytten av hver av de syv ulike RAID-nivåene (RAID levels)

SVAR:

- Datainnhold

- 0: Splitting
  - En aksess: Krever relativt små striper
  - Flere aksesser: Krever relativt store striper
- 1: Dublering m/Speiling
  - Hver lesing: En disk, styrt av nærmest
  - Hver skriving: Begge disker, styrt av fjernest
- 2: Inkl. Hamming-kode /
  3: Inkl. Paritets-bit
  2 & 3: En par. Aksess !?
  - Relativt små striper: Mange involvert
  - 2,3: Få involveres via ekstra-skriving
  - 2,3: Én feil kompenseres via ekstra-lesing
- 4,5: Inkl. Single Paritets-blokk /
  6: Inkl. Doble Paritets-blokk
  4,5,6: Flere uavh. Aksesser ?!
  - Relativt store striper: Få involvert
  - 4,5,6: Flere involveres via paritets-skriving
  - 4,5: Én feil kompenseres via paritets-lesing
  - 6: To feil kompenseres via paritets-lesing

- Nytte

| Nivå | Type | Overføringsrate for en | Tjenesterate for flere | Typisk applikasjon |
|---|---|---|---|---|
| 0 | Splitting | Små striper: ++ | Store striper: ++ | Ikkekritiske data |
| 1 | Dublering | R: +, W: - | R: +, W: - | Kritiske data |
| 2 | En parallell aksess | ++ | ÷ | --- |
| 3 | | ++ | ÷ | Høy overføringsrate |
| 4 | Flere uavhengige aksesser | R: -, W: ÷ | R: ++, W: - | --- |
| 5 | | R: -, W: ÷ | R: ++, W: - | Høy tjenesterate |
| 6 | | R: -, W: ÷ | R: ++, W: - | Høy tjenesterate & Ekstrem høy pålitelighet |