NTNU
Kunnskap for en bedre verden

Institutt for Datateknikk of Informasjonsvitenskap

Løsningsforslag for TDT4186 Operativsystemer

Eksamensdato: 22. mai 2015

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.
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)

a) Angi kort hvorfor vi trenger operativsystemer – og hva som skiller dem fra andre store programvaresystemer

SVAR:

Vi trenger operativsystemer for å gjøre det enklere å utnytte datamaskiner – ved å tilby ulike tjenester som er lette å bruke, og mer effektivt å bruke datamaskiner – ved å forvalte ulike ressurser på en god måte.

Operativsystemer skiller seg fra andre store programvaresystemer ved at de ligger mellom ulike typer maskinvare og alle andre programvaresystemer som kjører på tilhørende maskinvare.

b) Diskuter kort ulike typer operativsystemer som typisk er i bruk – og hva som skiller dem fra hverandre

SVAR:

Ulike typer operativsystemer er gjerne utviklet for følgende datamaskintyper

- Stormaskin – 5M $ (Bankapplikasjoner)
- Tjenermaskin – 5K $ (Nettverkstjenester)
- Personlig datamaskin – 500 $ (Bord / Fang)
- Mobil datamaskin – 50 $ (Brett / Mobil)
- Mikrokontroller – 5 $ (Vaskemaskin / Vekkeklokke)
- Kastbar kontroller – 0,5 $ (Hilse-kort / Id-kode)

hvor angitt applikasjonsområde angir bruk – og angitt pris antyder størrelse.

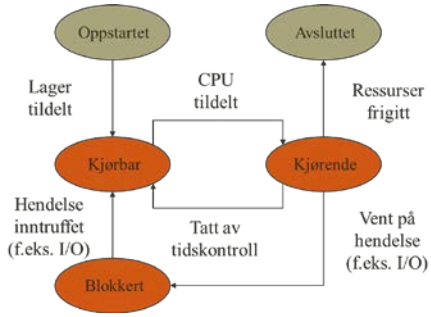
En kan også skille mellom ulike operativsystemer mht datamaskinarkitektur:

- Multiprosessor / Multikjerne maskin (Utvidete maskiner)
- Distribuert system (Sammenkoplete maskiner)

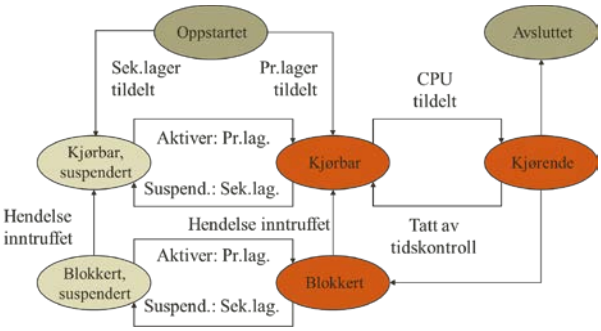
c) Angi kort typiske modeller for prosesstilstander i operativsystemsammenheng – og hva som er forskjellen mellom prosess-skifter og modus-skifter

SVAR:

En basismodell for prosesstilstander er:



En mer detaljert modell for prosesstilstander er:



Et prosess-skifte tilsier

- Bytte CPU fra en prosess til en annen
- Tilstandsendringer for to prosesser
- Modusskifter inn og ut av operativsystem

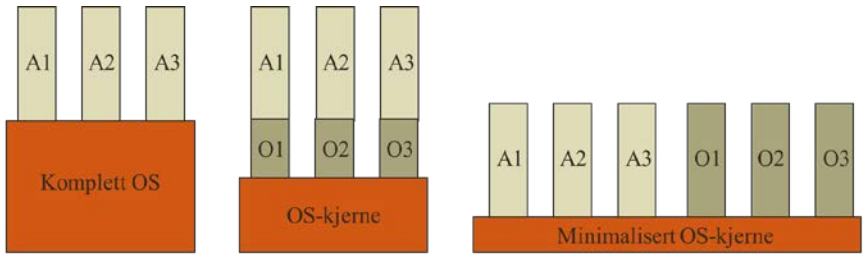
mens et modus-skifte tilsier:

- Bytte adresserom og instruksjonssett
- Eksternt avbrudd, instruksjonsreaksjon, OS-kall (muligens rekursivt for eksterne / interne)

d) Diskuter kort ulike måter å organisere operativsystemer på – og fordeler og ulemper med mikrokjernebaserte operativsystemer

SVAR:

Tre ulike måter å organisere operativsystemer på er



Alt i operativsystemet *Med funksjoner i brukerprosessene* *Via egne systemprosesser – Mikrokjernebasert*

hvor den til venstre (opprinnelig modell) er lite effektiv, men ganske trygg; mens den i midten (tidlig UNIX-modell) er mer effektiv, men samtidig mindre trygg; og den til høyre (moderne modell) er enda mindre effektiv enn den første, men også enda mer trygg enn den første.

Viktige fordeler med mikrokjernebaserte operativsystemer er

- Enklere programvare
- Lettere å tilpasse
- Lettere å utvide
- Lettere å flytte
- Distribuert tankegang
- Objektorientert tankegang

mens noen ulemper med mikrokjernebaserte operativsystemer er:

- Endring fra eksisterende systemer
- Unødvendig for mindre systemer

Oppgave 2: Synkronisering av prosesser (Process Synchronization)

a) Angi kort hva gjensidig utelukkelse (Mutual Exclusion) er – og hvorfor operativsystemer må tilby mekanismer for det

SVAR:

Gjensidig utelukkelse betyr at to eller flere kodesegmenter aldri kan overlappe i tid utførings-messig sett – men må bli utført etter hverandre i en eller annen rekkefølge.

Operativsystemer må tilby gjensidig utelukkelse for å unngå uheldige effekter av parallell utførelse av to eller flere kodesegmenter – ifm aksess av felles variable i ulike varianter.

b) Diskuter kort forskjellen mellom aktiv venting og passiv venting i operativsystemsammenheng – og når vi må og kan benytte hver av de to variantene

SVAR:

Aktiv venting tilsier at det forbrukes CPU-kraft mens en prosess venter på å komme videre – for eksempel i kortere / lengre testløkker, mens passiv venting tilsier at det ikke brukes CPU-kraft mens en prosess venter på å komme videre – for eksempel ved å gi fra seg CPU-en for kortere / lengre tidsrom.

Aktiv venting kan brukes ifm korte kritiske regioner – og må brukes på laveste nivå ifm implementering av kritiske regioner (gjerne maskinvarebaserte løsninger), mens passiv venting må brukes ifm lange kritiske regioner – og kan brukes på høyere nivå ifm implementering av kritiske regioner (gjerne programvarebaserte løsninger).

c) Diskuter kort forskjellene mellom å umuliggjøre vranglåser (Deadlock Prevention), å unngå vranglåser (Deadlock Avoidance), og å oppdage og rette opp vranglåser (Deadlock Detection) – samt fordeler og ulemper med hver av de tre måtene å håndtere vranglåser på

SVAR:

Å umuliggjøre vranglåser vil si å sikre seg at vranglåser ikke kan skje, mens å unngå vranglåser vil si å sikre seg at vranglåser ikke vil skje, og å oppdage og rette opp vranglåser vil si å la vranglåser skje.

Umuliggjøre vranglåser:

- Vil slippe gjenoppretting (+)
- God til noen problemstillinger (+)
- Ikke anvendbar til alle problemstillinger (-)
- Vil kreve mye overhead (-)

Unngå vranglåser:

- Vil slippe gjenoppretting (+)
- God til noen problemstillinger (+)
- Ikke anvendbar til alle problemstillinger (-)
- Vil kreve mye overhead (-)

Oppdage og rette opp vranglåser:

- Vil slippe mye overhead (+)
- Anvendbar til stort sett alle problemstillinger (+)
- Kan gi mye dødtid (-)
- Kan kreve mye gjenoppretting (-)

d) Beskriv helt konkret hvordan en Bundet Buffer (Bounded Buffer) kan implementeres med monitorer (Monitors)

SVAR:

En produsent/konsument-løsning med 2 begrensete bufre – basert på vanlige monitorer, er:

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                  /* buffer pointers */
int count;                                  /* number of items in buffer */
cond notfull, notempty;         /* condition variables for synchronization */

void append (char x)
{
     if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                      /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                   /* one fewer item in buffer */
     csignal(notfull);                       /* resume any waiting producer */
}
{                                                          /* monitor body */
     nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Med Mesa monitorer vil Append-/Take-rutinene i første del måtte endres til:

```
void append (char x)
{
     while(count == N) cwait(notfull);   /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;                                      /* one more item in buffer */
     cnotify(notempty);                       /* notify any waiting consumer */
}

void take (char x)
{
     while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                   /* one fewer item in buffer */
     cnotify(notfull);                       /* notify any waiting producer */
}
```

Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor

Oppgave 3: Håndtering av lager (Memory Management)

a) Angi kort hva virtuelt minne (Virtual Memory) er – og hvorfor operativsystemer må tilby mekanismer for det

SVAR:

Virtuelt minne betyr at bare deler av en prosess ligger i primærlageret mens den utføres, mens resten av / hele prosessen ligger på sekundærlageret.

Primærlagerplass er vanligvis en begrenset ressurs, og derfor vil virtuelt minne gjøre at en datamaskin kan ha flere kjørende prosesser / større kjørende prosesser inne i primærlageret enn uten.

b) Diskuter kort hvordan Buddy-systemer virker – og på hvilke måter de kan sies å kombinere dynamisk partisjonering (Dynamic Partitioning) og fast partisjonering (Fixed Partitioning) i operativsystemsammenheng

SVAR:

Buddy-systemer tilsier at det kun allokeres lagerplass i størrelser som er en potens av 2 – 1K, 2K, 4K, 8K, 16K osv., med evt. oppdeling av større lageromåder i mindre slike ved lagertildeling og evt. sammenslåing av mindre lageromåder til større slike ved lagerfrigiving.

Således oppnår vi på samme tid en slags dynamisk partisjonering - ved at flere ulike lagerstørrelser kan allokeres, og en slags fast partisjonering - ved at kun visse bestemte lagerstørrelser kan allokeres.

c) Diskuter kort hvorfor algoritmer for sideutbytting (Page Replacement) er så viktige i operativsystemsammenheng – og angi kort noen gode algoritmer for sideutbytting i eksisterende operativsystemer

SVAR:

Spennet mellom ytelsen til gode og dårlige algoritmer for sideutbytting er veldig stort, og da slike algoritmer må kjøres veldig ofte; blir det viktig å velge riktig algoritme for sideutbytting.

Noen gode algoritmer for sideutbytting i eksisterende systemer er:

- 2-sjansers klokkebasert algoritme (U-CLOCK) – krever et referert-bit (+)
- 4-sjansers klokkebasert algoritme (UM-CLOCK) – krever både referert- og endret bit (+)
- Minst nylig referert (LRU) – krever et tidsmerke for hver side til en prosess (-)
- Minst ofte referert (LFU) – krever en referanseteller for hver side til en prosess (-)

d) Beskriv helt konkret hvordan 2-Sjanser algoritmen (CLOCK/U-CLOCK) for sideutbytting virker

SVAR:

Ved behov for sideutbytting søkes det i en sirkulær sammenstilling etter neste ramme hvor det er en side med resatt referert bit:

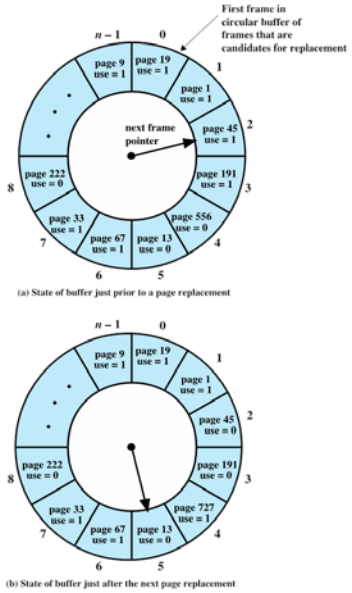
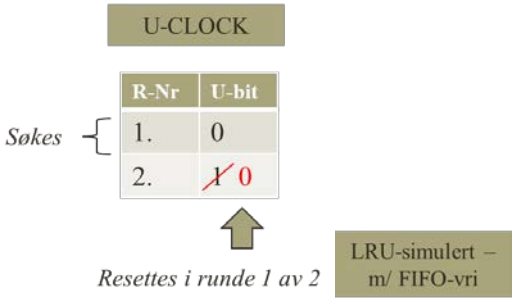


Figure 8.15 Example of Clock Policy Operation



Oppgave 4: Tidsstyring av prosesser (Process Scheduling)

a) Angi kort ulike mål for tidsstyring av prosesser – og hvordan systemer med sanntidsfokus har andre mål enn systemer uten sanntidsfokus

SVAR:

	Brukerfokus		Systemfokus	
Ytelses-orientering	Responstid	SinglePros	Gjennomstrømning	SinglePros
	Gjennomløpstid	SinglePros	Ressursutnyttelse	SinglePros
	Tidsfrister	SannTid	Overhead	MultiPros
Ikke ytelses-orientering	Utsulting	SinglePros	Rettferdighet	SinglePros
	Forutsigbarhet	SannTid	Ressursbalansering	MultiPros
	Ressursbibehold	SinglePros	Prosessprioritering	SinglePros

Systemer med sanntidsfokus har som hovedmål at prosesser/tråder skal kunne kjøres ferdig innen en viss tid fra problemet oppstår eller innen en viss tidsfrist, mens systemer uten sanntidsfokus har som hovedmål at prosesser/tråder skal levere korrekte resultater ved ferdiggjøring.

b) Diskuter kort ekstra utfordringer som henholdsvis multiprosessorarkitekturer (Multiprocessor Architectures) og multikjernearkitekturer (Multicore Architectures) byr på i operativsystem-sammenheng – og angi kort noen typiske relevante måter å løse dem på

SVAR:

Mens det i singleprosessor systemer er viktig å holde CPU-en i virksomhet med kjøring av en eller annen prosess, kan det i multiprosessor systemer være viktigere å koordinere kjøring av flere tråder i en prosess på flere CPU-er.

Følgende algoritmealternativer bør da vurderes for multiprosessor systemer:

- Lastdeling – som tilsier selvtilordning av prosessorer
- Samkjøring – som sikrer at flere/alle tråder innen en prosess får en prosessor hver
- Prosessorholding – som sikrer at flere/alle tråder innen en prosess får og beholder en prosessor hver
- Antallsvariasjon – som tilsier at antall tråder innen en prosess kan variere dynamisk med tilhørende prosessortilordning

Mens det i multiprosessor systemer kan være viktig å sikre høy utnyttelse av prosessorkraft, er det i multikjerne systemer viktigere å holde felles lageraksess innen brikken.

Følgende retningslinjer bør da benyttes for multikjerne systemer:

- Delvis felles cacher:
  Bruk nabokjerner - som deler cache, ved behov for felles lageraksess
- Delvis felles cacher:
  Bruk ikke-nabokjerner - som ikke deler cache, ved behov for høy utnyttelse

c) Diskuter kort hva som menes med invertering av prioriteter (Priority Inversion) – og når og hvorfor dette typisk brukes

SVAR:

Det vil være en viss utfordring hvis en høyprioritert prosess må vente på en lavprioritert prosess - f.eks. ved ressurs-låsing, og det vil være en enda større utfordring hvis en høyprioritert prosess må vente på både en lavprioritert prosess - f.eks. ved ressurs-låsing, og også en annen urelatert prosess; dette kan tilsi behov for å bytte om prioriteter på prosesser i slike tilfeller.

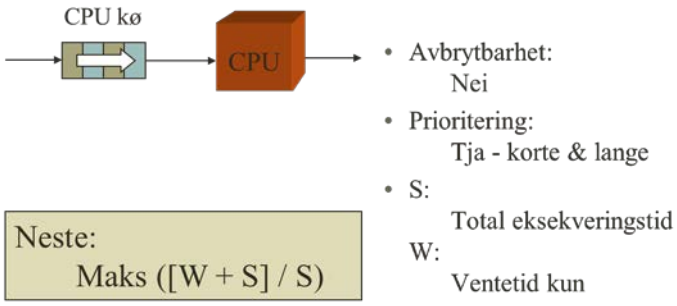
To måter å håndtere dette på er:

- Prioritertsarving - En lavprioritert prosess arver prioriteten til en høyprioritert prosess som stoppes
- Prioritetstak - En ressursbrukende prosess overtar prioriteten fra en topprangert ressurs ved ressursbruk

d) Beskriv helt konkret hvordan Høyeste-Responsforhold-Først algoritmen (Highest Response Ratio Next) virker

SVAR:

Ved avslutning av en prosess velges den prosess som har høyeste angitte forhold mellom Ventetid så Langt + Angitt Eksekveringstid og Angitt Eksekveringstid til å bli kjørt:



Oppgave 5: Håndtering av I/O (I/O Management)

a) Angi kort ulike måter å betjene samtidige diskforespørsler på – og ulike resultatparametre dette har innvirkning på

SVAR:

Noen vanlige algoritmer for å velge mellom samtidige diskforespørsler på er:

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

Noen tilhørende resultatparametre – med angivelse av hvilken algoritme som gjør det tilsvarende bra, er:

- 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

b) Diskuter kort forskjellene mellom bruk av sammenhengende enheter (Contiguous Allocation), kjedete enheter (Chained Allocation) og indekserte enheter (Indexed Allocation) – samt fordeler og ulemper med hver av de tre måtene å håndtere plassallokering på disker på

SVAR:

Med sammenhengende enheter allokeres én gruppe med et visst antall blokker etter hverandre, mens med kjedete enheter allokeres flere grupper med ulike antall naboblokker i hver og lenker mellom de forskjellige gruppene, og med indekserte enheter allokeres flere grupper med ulike antall naboblokker i hver og indekser til de forskjellige gruppene.

En sammenligning av de tre måtene for plassallokering på disker er

Aspekt	Sammen-hengende	Kjedet - fast	Kjedet - variabel	Indeksert - fast	Indeksert - variabel
Tidspunkt	I forkant	I forkant / Ved behov		I forkant / Ved behov	
Enhetstype	Variabel	Fast	Variabel	Fast	Variabel
Størrelse	Stor	Liten	Middels	Liten	Middels
Frekvens	Minimal	Høy	Middels	Høy	Middels
Plass	Minimal	Liten	Middels	Stor	Middels
Tid	Middels	Lang	Middels	Kort	Middels

Tid for allokering:
Fast: Lange lenker
Variabel: Korte lenker

Plass for allokering:
Fast: Små indeksinnslag
Variabel: Store indeksinnslag

hvor de fire siste radene angir fordeler og ulemper for hver av dem.

c) Diskuter kort forskjellene mellom eksplisitt I/O basert på programkode i brukerprosesser og implisitt I/O basert på systembehov ifm virtuelt minne – og angi kort hvorfor Minst-Nylig-Referert algoritmen (Least Recently Used) er mer aktuell som algoritme ifm eksplisitt I/O enn ifm implisitt I/O

SVAR:

Slik eksplisitt I/O kommer fra at en programmerer har angitt i koden at lese- eller skriveoperasjoner skal utføres, mens slik implisitt I/O ikke har noen tilhørende lese- eller skriveoperasjoner i koden; det kommer i stedet fra behov for utflytting til sekundærlager av visse sider/segmenter og/eller nødvendig innlesing av andre sider/segmenter fra sekundærlager ifm at alle sider/segmenter til kjørende prosesser ikke alltid finnes i primærlageret.

Eksplisitt I/O skjer i praksis mye sjeldnere enn implisitt I/O, og derfor kan vi da ta kostnaden ved å bruke den gode LRU-algoritmen - men samtidig kostbare i form av tilhørende lang eksekveringstid - når den nå først skal kjøres vesentlig sjeldnere.

d) Beskriv helt konkret hvordan Frekvensbasert-Stakk algoritmen (Frequency-Based-Replacement) virker ifm med I/O caching

SVAR:

Frekvensbasert-stakk (FBS)-algoritmen finnes i to varianter:

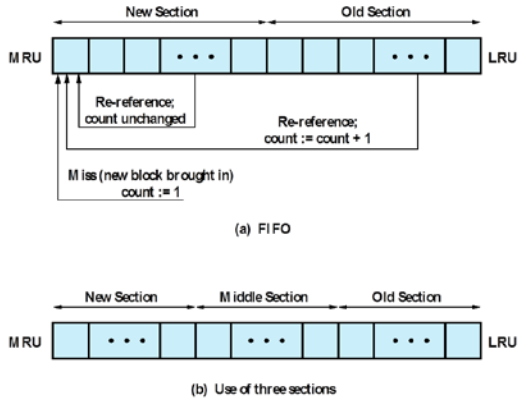


Figure 11.9 Frequency-Based Replacement

Mens LRU krever et tidsmerke implementert som en stakk – for eksempel med en pekerliste, krever LFU en referanseteller implementert direkte med en variabel.

FBS er så en kombinasjon av LRU & LFU:

- Plassering i hvilken sektor (New/Middle/Old): Bestemmes av LRU-prinsipp
- Utvelging fra Old-sektor: Bestemmes etter LFU-prinsipp

- New: ⅂ Fjerne; ⅂ Øke
- Middle: ⅂ Fjerne; men Øke
- Old: Fjerne & Øke

- New+Middle+Old: Fast totalantall
- New vs. Mid. vs. Old: Faste prosentandel

- New + Old: Overkommer LFU-problem (klomping)
- + Middle: Overkommer LRU-problem (oppbygging)