Group number and student names:

gruppe 22

Aleksander Kolsrud

Emmanuel Medard

Erik Aasen Eskedal

Kevinas Maksevicus

Kristian Skibrek

Tobias Vetrhus

Course Code:	***IS-105***
Course Name:	***Datakommunikasjon og operativsystem***
Responsible teacher/professor:	***Janis Gailis***
Deadline / Due date:	
Number for pages included in the deliverable:	*10*
Title:	*modul 3*

We confirm hereby that we are not citing or using in other ways other's work without stating that clearly and that all of the sources that we have used are listed in the references	Yes	No
We allow the use of this work for educational purposes	Yes	No
We hereby confirm that every member of the group named on this page has contributed to this deliverable/product	Yes	No

Innhold

Innledning **4**

Kapittel 1 **4**

1.1 4
1.2 5
1.3 5
1.4 6
1.5 7
1.6 7
1.7 8
1.8 8
1.9 9
1.10 10
1.11 10
1.20 11
1.26 11
1.27 12
1.29 12
1.30 12

Kapittel 2 **13**

2.1 13
2.2 15
2.3 16
2.4 16
2.5 17
2.6 17
2.7 18
2.8 18
2.9 18
2.10 20
2.11 20
2.14 20
2.16 21
2.18 21
2.19 22
2.21 22
2.22 23
2.23 23
2.24 24
2.26 25

Kapittel 3 **27**

3.1 28

3.2 28
3.3 29
3.4 29
3.5 30
3.6 31
3.7 31
3.8 32
3.9 33
3.10 33
3.11 34
3.12 34
3.19 36
3.21 38
3.22 40
3.23 43
3.25 45
3.26 50

Kapittel 6 **53**
6.1 53
6.2 53
6.3 55
6.4 57
6.5 58
6.25 58

Kapittel 8 **59**
8.1 59
8.2 60
8.3 60
8.4 60
8.5 61
8.6 62
8.7 63
8.8 64
8.9 65
8.10 66
8.11 66

Kapittel 9 **69**
9.1 69
9.2 69
9.3 70
9.4 70

Kapittel 14 **71**

14.1 71
14.9 72
14.15 72
14.16 73
14.17 73
14.23 73
14.24 74

Kapittel 15: **74**
15.1 74
15.2 74
15.3 75
15.8 75
15.9 76
15.10 76
15.12 77
15.14 78

Kilder: **78**

IS 105 modul 3

Innledning

Denne teksten tar for seg oppgavene tildelt av faglærer. Oppgavene hører til boka *Operating System Concepts - 9th Edition* Det er forventet at leser har denne boken tilgjengelig mens teksten leses for å se oppgaveteksten. Oppgaver med programmer er lastet opp til https://github.com/TheSkibb/Gruppe-22-IS-105

Kapittel 1

1.1

What are the three main purposes of an operating system?

De tre hovedfunksjonene til et operativsystem er (1) at det skal gi funksjonalitet slik at en bruker kan kjøre programmer på maskinvaren. Operativsystemet skal være et bindeledd mellom brukeren av datamaskinen og datamaskinens maskinvare. Videre er operativsystemets oppgave (2) å allokere og holde styr på de ulike ressursene i datamaskinens

maskinvare for å løse et problem. Det sørger for lasting av programmer, kobling mellom prosessor og lagringsmedier som for eksempel RAM (Liseter, 2020), hvor allokeringen skal være så effektiv som mulig. Sist skal (3) operativsystemet sørge for at bruker unngår feil og opprettholde tilgangskontroller, Det skal hindre skadelig bruk av datamaskinen gjennom at de vitale delene av maskinen kun er tilgjengelig i kjernemodus, sammen med andre sikkerhetsringer som hindrer at systemet krasjer. Samtidig skal det skal holde styr på funksjonaliteten og kontrollering av I/O enheter (Silberschatz, Galvin, Gagne, 2013, s. 1)**.** Det kan diskuteres at operativsystemet har flere hovedområder enn 3. GeeksforGeeks har for eksempel liste med 9 hovedfunksjoner (Geeksforgeeks, 2020), mens Daria.no har utnevnt 4 hovedområder (Daria, 2000). Det er dermed ingen universell definisjon på hva hovedoppgavene er.

1.2

We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful?

Det er passende for operativsystemet å «sløse» ressurser når systemet er designet for å støtte én bruker som skal utnytte ressursene, eller når det er nødvendig å prioritere andre spesifikke prosesser framfor andre. Operativsystemet kan for eksempel være designet for 'ease of use' og med noe fokus på performance, men ikke noe særlig fokus på ressursallokering. GUI kan 'sløse' noen ressurser, men det optimaliserer brukerens opplevelse av systemet, noe som er det viktigste i denne situasjonen. Det er ikke nødvendigvis sløsing av ressurser, siden systemet kun er designet for bruk av en person av gangen. Det er vesentlig viktigere med god ressursfordeling når flere brukere aksesserer samme maskinvare (Silberschatz et al., 2013, s. 3)**.**

1.3

What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?

Real-time systemer er den mest vanlige formen for datamaskin vi har, og finnes i alt fra bilmotorer og DVD'er til roboter. De har som regel en veldig spesifikk oppgave, så operativsystemet har dermed liten funksjonalitet. Real-time operativsystemer har oppgaver med ulik proritetsverdi, som vil si at de høyest prioriterte oppgavene blir utført - noe som

resulterer i et svært responsivt system. Et RTOS (realtime operativsystem) har svært spesifikke oppgaver, som begrenser dens bruksområder sammenlignet med OS, som Windows og MacOS (Highintegritysystems, u.å.).

Hovedproblemet en programmerer må håndtere når hen skriver et real-time operativsystem er at prosesseringen må bli utført med fullt definerte begrensninger og innen spesifikke tidsrammer, ellers vil systemet feile. Systemfunksjonalitetene må returnere det korrekte resultatet ut i fra tiden som brukes til å utføre den definerte oppgaven (Silberschatz, et al., 2013, s. 45).

1.4

Keeping in mind the various definitions of operating system, consider whether the operating system should include applications such as web browsers and mail programs. Argue both that it should and that it should not, and support your answers.

Operativsystem burde ikke inneholde nettlesere og post-systemer fordi operativsystemets oppgave er å kontrollere datamaskinens maskinvare og koordinere bruken av programmene. Operativsystemet har heller ikke alltid med applikasjoner å gjøre, fordi vi finner de i biler, mikrobølgeovnen, båter og andre maskiner som ikke alle krever applikasjonsprogrammer som regneark, Word-dokument eller en nettleser. Operativsystem er det som rett og slett 'opererer systemet', mens applikasjonsprogrammer er alt som ikke er assosiert med det.

Likevel er et av operativsystemets viktigste oppgaver å løse problemer basert på det brukeren vil oppnå. Applikasjonsprogrammene definerer hvordan ressursene skal bli brukt til å løse brukerens dataproblemer. Operativsystemet skal kontrollere og koordinere maskinvaren i henhold til brukerens bruk av disse programmene. Applikasjonsprogrammer er dermed en viktig del av hvordan operativsystemer opererer for å løse brukerens oppgave - maskinvare alene er heller ikke lett å bruke, noe som er selve grunnen til at applikasjonsprogrammer blir utviklet (Silberschatz, et al., 2013, ch. 1)

Chrome OS er et eksempel på et internett OS, som baserer seg på Linux og open source, noe tilsier at alle kan utforske systemets kode, samtidig som det er gratis å bruke. Et av hovedfokuset er enkelhet og brukervennlighet. Slike OS er likevel ikke egnet for alle, og har begrensede bruksområder, for eksempel kan det ikke kjøre store spill (Rutnik, 2021). En av de største fordelene er likevel open source, som sikrer full transparens.

1.5

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) system?

Forskjellen mellom 'kernel mode' og 'user mode' er at kernel mode har ubegrenset tilgang til maskinvaren, kan utføre CPU instruksjoner og finne frem minneadresser - det har generelt ansvaret for de viktigste funksjonene av et operativsystem. User mode er koden som brukeren forsørger. En annen forskjell er at krasj i kernel er katastrofisk for maskinen, mens en krasj i brukermodus kan gjenopprettes. Det meste av koden på datamaskinen utføres i brukermodus (Simmons, 2014). Ut ifra dette får vi 'mode bit' som definerer kernel som 0 og bruker som 1, og ved hjelp av dette kan vi skille mellom en oppgave som utføres på vegne av operativsystemet og en som skal utføres på vegne av brukeren.

Hver gang det skjer noe galt bytter systemet fra brukermodus til kernelmodus – bit blir 0. Dette sørger for at vi kan beskytte operativsystemet for skadelig bruk. Det finnes såkalte «privileged instructions» som er instruksjoner som maskinvaren kun kan utføre i kernelmodus, og som ikke vil bli tillatt i brukermodus. Selve instruksjonen som bytter til kernelmodus, I/O kontroll, timer- og forstyrrelse håndtering er eksempler på privileged instructions (Silberschatz, et al., 2013, s. 22)**.**

1.6

Which of the following instructions should be privileged?

a) **Set value of timer**

   Å sette timerverdi burde være priviligert siden det er kernelen som har kontroll på timerhåndtering (Silberschatz, et al., 2013, s. 24)**.**

b) **Read the clock**

   Å lese klokken trenger ikke å være priviligert, siden enhver bruker har tilgang til å lese klokken.

c) **Clear memory**

   Å fjerne minne er priviligert, siden bruker kan skade programvaren hvis ikke (Rishabhjain12, 2019).

d) **Issue a trap instruction**

   'Issue a trap instruction' er en ikke-priviligert instruksjon fordi den kan brukes av brukeren til å bytte til kernelmodus (Rishabhjain12, 2019).

e) **Turn of interrupts**

‘Turn off interrupts’ på den andre siden er en priviligert instruksjon, siden hvis en interrupt inntreffer skal kontrollen returneres til kernelen, det vil derfor være en ulovlig handling i brukermodus å skru av dette (Rishabhjain12, 2019).

f) **Modify entries in device-status table**

‘Modify entries in the Device-status table’ er en privileged instruction (Bohyar, 2017).

g) **Switch from user to kernel mode**

Å bytte fra bruker- til kernelmodus er en priviligert instruksjon, siden kernelen skal være beskyttet.

h) **Access I/O device**

Å aksessere I/O enheter er også en priviligert instruksjon slik at brukere ikke skal misbruke dem, og slik at systemet kan sjekke om du er autorisert til å utføre visse I/O operasjoner (Bohyar, 2017)

1.7

Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme

Det er vanskelig å si hva forfatteren definerer som “early computers”, men datamaskinene kom likevel under 2. verdenskrig hvor de skulle bli brukt til svært spesifikke og ballistiske, militære oppgaver (Computer Sciences, u. å.). Problemer med slike tidlige systemer er at dataen som kreves av operativsystemet ville vært tilgjengelig for hvem som helst, samtidig som minnet ikke ville vært beskyttet, som gjerne inneholder passord, kontrollere og annen informasjon (Shah Abdul Latif University, 2018). Siden operativsystemet ikke kan modifiseres ville det blitt svært alvorlig hvis det skulle forekomme feil eller en bug i systemet - de kan ikke korrigeres.

1.8

Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?

Bruk av ‘multiple modes’ er for eksempel VMM som har flere tilgjengeligheter en brukerprosesser. Denne kan lage og holde styr på virtuelle maskiner, hvor den utnytter de

ekstra privilegiene til å få CPUen til å gjøre det. Det finnes flere moduser for å kunne gjøre andre mer spesifikke oppgaver, som for eksempel å la USB enheter kjøre på PC-en, uten at det trengs å bytte til kernelmodus (Silberschatz, et al., 2013, s. 22/23).

Medium.com nevner også at det er to andre moduser mellom kernel og bruker. De har fordelt den som ringer, hvor "Ring 0" er hypervisoren som styrer virtuelle maskiner, som for eksempel VMM, og brukerens kernel er "Ring 1". Brukerens enhetsdrivere er "Ring 2", og brukerens applikasjoner er "Ring 3" (RealWorldCyberSecurity, 2020).

Videre finnes det "negative" ringer under kernelen, som for eksempel SMM (System Management Mode). Her finner vi blant annet BIOS og flere andre initialisering oppgaver når maskinen starter. En annen negativ ring er ME (Management Engine), som har egne prosessorer som kjører hele tiden, selv om maskinen er av. Den har tilgang til hovedprosessoren og all minnet. ME implementerer sikker oppstart av maskinen og ulike funksjonaliteter, men funksjonaliteten har ikke enda blitt formelt dokumentert (RealWorldCyberSecurity, 2020). Det skal sies at dette gjelder Intels prosessorarkitektur.

1.9

Timers could be used to compute the current time. Provide a short description of how this could be accomplished.

Vi har timere slik at et program ikke skal bli sittende i en uendelig loop, og aldri returnere kontrollen tilbake til operativsystemet, samt at den forsikrer en rask syklus av prosesser gjennom CPUen og generelt korrekt operasjon (Silberschatz, et al., 2013, s. 36). En timer kan implementeres gjennom en "fixed-rate clock" og en teller. Operativsystem setter timeren (Silberschatz, et al., 2013, s. 24).

Timere kan computere den nåværende tiden gjennom å bruke såkalte 'timer interrupts'. Det kan settes en time limit, hvor for eksempel programmet skal vare i 7 minutter. Da har timeren en 420 sekunders grense. Hvert sekund avbryter timeren og telleren dekrementers med 1. Når telleren etter hvert får negative tall vil programmet bli terminert (Silberschatz, et al., 2013, s. 24)**.**

1.10

Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

Caches (mellomlager/hurtigbuffer) er nyttige når to eller flere komponenter må utveksle data, og komponentene utfører overføringer i ulike hastigheter. Cache løser overføringsproblemer ved hurtigbuffer. Hurtigbuffer tas i bruk for å redusere trafikken via tungt belastede kommunikasjonslinjer. Særlig viser hurtigbuffer seg nyttig i henhold mellom indre og ytre lager av betydelige gevinster, da søk i og innlesing fra en buffer går flere ganger hurtigere enn selv fra de aller raskeste magnetiske lagringsmediene (Bratbergsengen, 2019).

Når det gjelder kapasiteten på enheten er det viktig at dataene i cache holdes i samsvar med dataene i komponentene. Hvis en komponent har en endring av dataverdien, og referansen også er i cache, må hurtigbufferen også oppdateres. Dette er spesielt et problem på flere prosesser systemer der mer enn en prosess kan få tilgang til et referansepunkt. En komponent kan elimineres med en cache av samme størrelse, men bare hvis: (a) cache og komponenten har tilsvarende hoved lagringskapasitet (det vil si at hvis komponenten beholder dataene når elektrisitet fjernes, må cache beholde data også), og (b) cache er rimelig, fordi raskere lagring ofte blir dyrere (Silberschatz, et al., 2013, s.28-29).

1.11

Distinguish between the client–server and peer-to-peer models of distributed systems

The client-server model (Klient-servermodellen) skiller rollene til klienten og serveren. Under denne modellen ber klienten om tjenester som leveres av serveren.
Peer-to-peer-modellen har ikke så strenge roller. Dessuten, blir alle noder i systemet betraktet på lik nivå som medfører til at de dermed kan fungere klienter ("peers") eller servere, eller begge deler. En node kan be om en tjeneste fra en annen kollega, eller noden kan faktisk tilby en slik tjeneste til andre personer i systemet ((Silberschatz, et al., 2013, s.40)

1.20

Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.
1. How does the CPU interface with the device to coordinate the transfer?
2. How does the CPU know when the memory operations are complete?
3. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

CPUen kan starte en DMA-operasjon ved å hente verdier i spesielle registre som enheten kan få tilgang til, uavhengig av hverandre. Enheten starter den tilsvarende operasjonen når den mottar en kommando fra CPUen. Når enheten er ferdig med sin drift, avbryter den CPUen for å signalisere at operasjonen er fullført. Både enheten og CPUen vil kunne få tilgang til minne samtidig. Minnekontrolleren gir tilgang til minnebussen i en fairmanner til disse to enhetene. En CPU kan derfor ikke være i stand til å utstede minneoperasjoner i topphastigheter, siden den må konkurrere med enheten for å få tilgang til minnebussen. (Silberschatz, et al., 2013, s.25-27)

1.26

Which network configuration—LAN or WAN—would best suit the following environments?
Nettverk karakteriseres ut fra avstandene mellom noder. Et lokalt nettverk (LAN) kobler datamaskiner i et rom, i en bygning, eller et campus. Et bredt nettverk (WAN) kobler vanligvis bygninger, byer eller land. Et globalt selskap eller statlig universitet kan ha et WAN for å koble sine kontorer over hele verden eller landet, derav disse nettverkene kan for eksempel kjøre en protokoll eller flere protokoller.
1. A campus student union
   - LAN.
2. Several campus locations across a statewide university system
   - WAN.
3. A neighborhood
   - LAN eller WAN.

(Silberschatz, et al., 2013, s.38)

1.27

Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs

De største utfordringene som kommer ved å designe operativesystemer for mobilenheter tilsammenligning med tradisjonelle Pcer er at mindre/begrensende lagringskapasitet på mobilenheten gjør at operativsystemet må administrere minne nøye. I tillegg må operativsystemet òg håndtere forbruket av strømmen nøye (Silberschatz, et al., 2013, s.37-39)

1.29

Describe some distributed applications that would be appropriate for a peer-to-peer system

Hovedsakelig så er det alt som gir innhold, i tillegg til eksisterende tjenester som filtjenester, distribuerte katalogtjenester som domene navnetjenester og distribuerte e-posttjenester. I denne modellen skilles ikke klienter og servere fra hverandre. I stedet betraktes alle noder i systemet som peers, og hver kan fungere som enten en klient eller en server, avhengig av om den ber om eller å tilby en tjeneste. Peer-to-peer systemer gir en fordel i forhold til tradisjonelle klientserver-systemer. I et klienserver-system er serveren en flaskehals, men i et peer-to-peer system kan tjenester leveres av flere distribuerte noder gjennom hele nettverket. Skype er et eksempel på et applikasjon som benytter peer-to-peer system. Det tillater klienter å ringe og videosamtaler samt sende tekstmeldinger over Internettet ved hjelp av en teknologi kjent som Voice over IP. Skype bruker en hybrid peer-peer-tilnærming. Den inkluderer en sentralisert påloggingsserver, men den inneholder også
desentraliserte jevnaldrende og lar to jevnaldrende kommunisere. (Silberschatz, et al., 2013, ch.1, s.39-40).

1.30

Identify several advantages and several disadvantages of open-source operating systems. Include the types of people who would find each aspect to be an advantage or a disadvantage.

Åpne kildekode-operativsystemer har fordelene ved å ha folk som jobber med dem, mange feilsøker dem, enkel adgang og distribusjon samt raske oppdatering sykluser. Videre, for

studenter og programmerere er det et verktøy bygd på informasjonsdeling, derav programvare er et fellesgode som hører til alle studenter og programmere hvorav de kan se, vurdere og endre kildekoden. Vanligvis er operativsystemer med åpen kildekode gratis til en viss grad, og krever som regel bare betaling for støttetjenester (Teknologirådet, 2004, s.33, 36. Kommersielle operativsystemer bedrifter misliker som regel konkurransen som åpen kildekode-operativsystemer bringer grunnet funksjonene er vanskelig å måle seg mot. Noen åpen kildekode tilbyr ikke betalt støtteprogram. Noen selskaper unngår åpne kildeprosjekter fordi de trenger betalt støtte, slik at de har en enhet som kan holdes ansvarlige dersom det er et feil eller de trenger hjelp til å løse et feil. Når det gjelder programvarepatenter kan det hevdes at den medfølger flere negative sider. Årsaken er grunnet den kan true åpen kildekode og fri konkurranse, i følge Teknologirådets eksperter kan åpen kildekoding svekkes av programvarepatenter, grunnet flere prosjekter har som regel ikke midler til å søke patenter eller forsvare seg mot patentsøksmål. Dette kan føre til at for eksempel Linux kan være sårbar mot angrep fra kommersielle aktørere som oppfatter Linux som en stor nok konkurrent (Teknologirådet, 2004, s.41).

Kapittel 2

2.1

What is the purpose of system calls?

Et systemkall (system call) er innen informatikk hvordan et program ber om en tjeneste fra kjernen til operativsystemet det utføres på. Et systemkall er en måte for programmer å samhandle med operativsystemet. Et dataprogram foretar et systemkall når det ber om operativsystemets kjerne. Systemkall gir tjenestene til operativsystemet til brukerprogrammene cia Application Program Interface (API). Det gir et grensesnitt mellom en prosess og et operativsystem for å la prosesser på brukernivå be om tjenester fa operativsystemet. Systemkall er de eneste inngangsnøklene til kjernesystemet. Alle programmer som trenger ressurser må bruke systemkall. Disse systemkallene er vanligvis tilgjengelige som rutiner skrevet i C og C++. (Silberschatz, et al., 2013, ch. 2, s.9)

Tjenester som leveres av systemkall

1. Prosessoppretting og styring
2. Styring av hovedminne

3. Filadministrasjon, katalog og filsystem administrasjon

4. Enhetshåndtering (I / O)

5. Beskyttelse

6. Nettverk, osv.

Typer systemkall: Det finnes fem ulike kategorier innen systemkall -

1. Prosesskontroll: slå av, avbryte, opprette, avslutte, tildele og frigjøre minne.
2. Filbehandling: opprette, åpne, lukke, slette, lese fil osv.
3. Enhetsadministrasjon
4. Informasjonsvedlikehold
5. Kommunikasjon

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	write()
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

figur fra Silberchatz side 68

2.2

What are the five major activities of an operating system with regard to process management?

En prosess er et program i utførelse. En prosess er mer enn programkoden, som noen ganger er kjent som tekstdelen. Den inkluderer også den nåværende aktiviteten, som representeres av programmets verdi og innholdet i prosessorregistrene.

De fem store aktivitetene er:

1.Opprettelse og sletting av både bruker- og systemprosesser.

- Dette skjer når du slår på datamaskinen, operativsystemet åpner prosesser for å kjøre tjenester alt fra utskriftskøen til datasikkerheten. Når du logger på datamaskinen og starter programmer, skaper programmene avhengige prosesser. En prosess er ikke selve programmet, men heller instruksjonene som CPU bruker for å utføre programmet. En prosess tilhører enten Windows eller et annet program du har installert.

2.Suspensjon og gjenopptakelse av prosesser.

- Tilstanden til en prosess kan være "opprettet", "kjører", "venter" eller "blokkeres." En prosess "venter" etter at man starter det overordnede programmet, og før det er behandlet av CPUen. En prosess "kjører" når prosessoren behandler den. En kan vurdere en prosess som er "blokkert" hvis datamaskinen ikke har nok minne til å behandle den, eller hvis filer som er tilknyttet prosessen ikke kan bli funnet. Alle operativsystemer har et slags prosess håndteringssystem, selv om de har forskjellige navn for hver tilstand.

3.Tilveiebringelse av mekanismer for prosesssynkronisering.

- Når prosesser kjører, trenger operativsystemet en måte å sikre at ingen prosesser får tilgang til de samme ressursene samtidig. To prosesser kan ikke forsøke å utføre samme kodeområde samtidig. Grunnet det kan medføre til at det kan oppstå et krasj når de prøver å kalle på de samme filene og sende de samme instruksjonene til CPUen samtidig. Dersom to prosesser må kjøre den samme koden, må den ene vente til den andre er ferdig før den fortsetter.

4.Tilveiebringelse av mekanismer for prosesskommunikasjon.

- Datamaskinen må sikre at prosesser kan kommunisere med prosessen og med hverandre. Et program kan for eksempel ha mange prosesser, og hver prosess kan ha et annet tillatelsesnivå. Et tillatelsesnivå er en indikasjon på tilgangsnivået en prosess

skal ha til systemet. Prosesskommunikasjon sikrer at datamaskinen kan bestemme tillatelsene til hver prosess. Dette er svært viktig for å forhindre skadelige programvare i å slette systemfiler eller legge til instruksjoner til selve operativsystemet.

5.Tilveiebringelse av mekanismer for driftslåsing.

- Til slutt må datamaskinen ha en måte å sikre at prosesser ikke blir fastlåst. Deadlock oppstår når to prosesser hver krever en ressurs som den andre bruker underveis, eller når ingen av prosessene kan fullføre jobben de gjør. Ressursene kan ikke frigjøres, og programmene låses opp. Denne situasjonen kan også ses på som en "sirkulær ventetid". Operativsystemer forhindrer deadlock på ulike måter, men den vanligste metoden er å tvinge en prosess til å erklære ressursene den trenger før den kan starte opp. Eventuelt kan en prosess bli tvunget til å be om ressurser i blokker, og deretter frigjøre ressursene når den er ferdig med dem.

(Silberschatz, et al., 2013, ch.2, s.25)

2.3

What are the three major activities of an operating system with regard to memory management?

Memory management (minnehåndtering) handler om den delen av et operativsystem som kontrollerer og håndterer dataminnet, og gir deler til forskjellige løpende program for å optimalisere gjennomføringen.

De tre hovedaktivitetene i minnehåndtering er:

1. Å ha kontroll på hvilke deler av minnet som blir brukt, i tillegg til å ha kontroll på hvem det er som bruker det
2. Å etter behov distribuere og tildele minneplass
3. Når minneplass blir tilgjengelig eller ledig, må man bestemme hvilke prosesser som skal lastes inn i minnet

(Whatis, 2012)

2.4

What are the three major activities of an operating system with regard to

secondary-storage management?

Secondary-storage management er den "non-volatile" lagringsplassen for data og programmer. Det vil si at det er et dataminne som selv etter strømmen har gått kan beholde lagret informasjon.

De tre hovedaktivitetene i secondary-storage management er:

1. Lagringsallokering
2. Diskplanlegging
3. Administrasjon for ledig plass

(Meador, 2018)

2.5

What is the purpose of the command interpreter? Why is it usually separate from the kernel?

Kernel er selve koden og den sentrale modulen for et operativsystem, og den har kontroll over alt som er i datamaskinen. Og kommando tolkeren (shell) er den delen av systemet som ser kommandoene som er lagt til av menneskene, og da tolker de.

Så en av måtene en bruker kan samhandle med operativsystemet er via kommando tolken. Hovedoppgaven til kommando tolken er å tyde og utføre instruksjoner, som den deretter konverterer til systemkall. Kernelen er operativsystemets kjernemodul. Siden kernelen er hjertet i operativsystemet, vil det være risikabelt å ha kode som er utsatt for endring som en del av kernelen. Så det er viktig å tenke nøye gjennom endringer eller oppdateringer som blir gjort i kernelen.

(Incomputersolutions, 2021)

2.6

What system calls have to be executed by a command interpreter or shell in order to start a new process?

For å starte en ny prosess i Unix systemer kan et kall på "fork" systemer, etterfulgt av "exec" systemer fungere. Fork-kallet dupliserer den gjeldende operasjonen, mens exec-samtalen oppretter en ny prosess basert på en annen kjørbar og legges over toppen av calling prosessen.

(Silberschatz et al., 2013, s. 67)

2.7

What is the purpose of system programs?

System programmer gjør at programmer kan gjennomføres og utvikles i et bestemt miljø. De hjelper brukere ved å gi dem en veldig simpel funksjonalitet slik at brukere ikke trenger å skrive lange programmer for å få løst enkle problemer. De gir en sammenheng mellom brukergrensesnitt og systemkall.

Eksempler kan være kompilator, debugger osv - som da kompilerer et program og forsikrer seg at det ikke er noen feil.

(Geeksforgeeks, 2021)

2.8

What is the main advantage of the layered approach to system design?

What are the disadvantages of the layered approach?

The layered approach er et type "systemdesign", der operativsystemet er delt inn i ulike lag. Det laveste laget er hardware og det høyeste laget er brukergrensesnitt.

Fordel:

- Hovedfordelen er rett og slett enkelheten med debugging og modifikasjon ettersom endringer kun påvirker én seksjon av systemet istedenfor alle.

Ulemper:

- Er at denne approachen ofte er mindre effektiv enn andre implementasjoner.
    - Når for eksempel et bruker program utfører en I/O-operasjon, fanger den kallet i I/O-laget, som deretter kaller minne-håndteringslaget, som deretter kaller CPU-planleggingslaget, som deretter sender samtalen til maskinvaren.
- Og det kan være vanskelig å definere de ulike lagene.

(Silberschatz et al., 2013, s. 79-81)

2.9

List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.

1. Filsystem manipulasjon:

- Her kan brukeren be om at data skal hentes ut, eller være lagret i filsystemer. Filsystemene er ofte delt inn i direktorier som gjør det enkelt å bruke og bevege seg rundt. Det kan ikke stoles på for et brukernivå ettersom de ikke forsikrer noe beskyttelse.

2. Program gjennomføring:
    - Det er flere aktiviteter fra system programmer til brukerprogrammer som f.eks. filservere til navneservere osv - som operativsystemer kan håndtere. Disse aktivitetene er innkapslet som en prosess. Det skjer en gjennomføring ved at operativsystemet laster inn filer eller generelt innhold til minnet. Brukernivå programmer kan ikke stoles på her til å allokere CPU tid.
3. I/O operasjoner:
    - Input/output systemer skal overføre informasjon mellom den utenforstående verden, og dataminnet. Så dette skjer gjennom input/output operasjoner - f.eks. en I/O transaksjon kan være å lese en datablokk fra disk til minne. En fil eller et I/O apparat kreves ofte i en Input/Output operasjon. Brukere kan vanligvis ikke kontrollere I/O apparater pga beskyttelse og effektivitet.
4. Kommunikasjoner
    - Kommunikasjon gjennom operativsystemer er åpenbart veldig greit for brukerne ettersom det er en enkel og grei måte å nå hverandre. Kommunikasjon kan bli implementert gjennom meldingsoverføring, eller via delt minne. Og der blir informasjonspakker flyttet mellom prosesser av operativsystemet. Brukerprogrammer får muligens ikke tilgang til nettverksenheten.
5. Feilregistrering
    - Feil er noe operativsystemet hele tiden må registrere og rette opp i, og dette skjer på både software og hardware nivå - som igjen er veldig greit for brukeren. Brukerne kunne ha gjort dette selv, men det gir mye mer mening å la OSen gjøre denne kjedelige jobben for oss.

(Tutorialspoint, 2021)
(El-Ghazawi & Frieder, 2003, p. 874-879)

2.10

Why do some systems store the operating system in firmware, while others store it on disk?

Firmware er en type software som opererer uten å gå gjennom APIs eller operativsystemet, og det er inngravert direkte til en slags del hardware. Grunnen til å bruke firmware som lagringsplass for operativsystemet, kan være om selve apparatet eller hardwaren ikke har disk eller kan holde på disk.
(Chakraborty, 2021)

2.11

How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

Et lite stykke kode kjent som bootstrap-programmet eller bootstrap loader lokaliserer kernelen, laster den inn i hovedminnet og begynner kjøringen på de fleste datasystemer. En totrinnsmekanisme brukes av noen datasystemer, for eksempel PC-er, der en grunnleggende bootstrap-laster henter et mer komplisert oppstartsprogram fra disken, som deretter laster kernelen. Bootstrap-programvaren er i stand til et bredt spekter av oppgaver. En av de vanligste oppgavene er og kjør diagnostikk for å se hva oppstartstrinnene er. Det kan også sette opp hele maskinen, fra CPU-registrene til datamaskin kontrollene og innholdet i hovedminnet.
(Answers, u.å.)

2.14

Describe how you could obtain a statistical profile of the amount of time spent by a program executing different sections of its code. Discuss the importance of obtaining such a statistical profile.

For å skaffe oss en statistisk profil som viser hvor lang tid en prosess tar når den går gjennom deler av koden sin, så må vi lage timer interrupts. Du kan lage periodic timer interrupts og når interruptene kommer fram så kan man sjekke hvilken deler av koden som blir gjort først. Det vi da får er et "profiling system" som kan monitorere kode som blir utført. Dette kan bli brukt senere for å optimalisere kode som tar opp mye av CPU ressursene.

2.16

What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?

Å kunne bruke en system-call interface for å manipulere både filer og enheter kan ha sine gode trekk og feiltrekk. Det å bruke system calls for å manipulere både filer og enheter vil føre til at enhetene i bildet blir "gjort om" eller aksessert til filer i filsystemet. Det positive med dette vil være at bruker koden vil være det samme for å få tak i både filene og enhetene men med forskjellige parametere. Det negative med dette er at det blir vanskelig å få full ytelse av enkelte enheter, noe som fører til mindre funksjonalitet.

2.18

What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

De to modellene for kommunikasjon på tvers av prosesser er "Message-passing" modellen og "Shared-memory" modellen. "Message-passing" modellen går ut på at prosesser kommuniserer med hverandre gjennom en felles buffer, men denne bufferen må være åpnet før de kan kommunisere, og her kan prosessene dele meldinger både direkte og indirekte.

Pros:

- Den er veldig nyttig for å dele små mengder data
- Er bortimot ingen konflikter å slippe unna
- Lett å implementere

Cons:

- På grunn av de implementeringene av prosessene i denne metoden gjør det den mye tregere enn den andre metoden.

Shared memory modellen tillater to eller flere prosesser å dele informasjon ved å lese og skrive data inn i delt lagringsplass/minne. Den delte lagringsplassen kan bli aksessert av flere prosesser samtidig. Et system som kan gjøre dette er f.eks. POSIX systemer, og Windows OS.

Pros:

- Prosessene kan samhandle så raskt som mulig
- Bedre kommunikasjon mellom flere prosesser

Cons:

- Kan skape sikkerhetsproblemer gjennom mindre tilgang
- Prosessene som bruker den delte lagringsplassen/minne må sørge for at de ikke skriver i samme minne.
- Kan oppstå vanskeligheter med synkronisering.

2.19

Why is the separation of mechanism and policy desirable?

Å separere mekanismer og retningslinjer vil føre til å skape mer fleksible systemer. Systemene vil være enklere å ha med å gjøre ved at modifikasjoner er lettere å implementere. Det vil gjøre at en hvilken som helst designer kan skape mekanismer som forblir uendret mens retningslinjene endres i retningen av det man ønsker.

Et godt eksempel på hvordan dette har god påvirkning vil være dørlåser og nøkler. Hoteller bruker gjerne kort nøkler for å få tilgang til rommene. I dette tilfelle vil ikke mekanismene (magnetiske kortlesere, fjernstyrte låser og connection til sikkerhetsservere) ha noen limitasjoner i forhold til policyene (som vil være hvem som får tilgang til hvilket rom/dør på hvilket som helst tidspunkt).

Hvis man brukte vanlige fysiske nøkler og vanlige låser; hvis du vil endre på hvem som kan åpne døra til f.eks. rom 302, så må du gi personen en ny nøkkel og endre på låsen. Dette gjør at mekanismene er avhengige av policyene, som gjør alt mye vanskeligere å holde kontroll på. (*Separation of mechanism and policy*, 2021)

2.21

What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

Fordelene med en micro-kernel metodisk tilnærmelse har flere fordeler:

- Det gjør det enklere å utvide operativsystemet

- OS er enklere å flytte fra hardware til hardware
- Færre endringer trengs å gjøres når man modifiserer kernelen
- Fører til at den er "tryggere" å bruke på grunn av det simple kernel designet og funksjonaliteten.
- Tilbyr bedre sikkerhet fordi metoden bruker message sharing metoden for å kommunisere.

Micro-Kernelen må være et slags mellomledd mellom programmene og de ulike tjenestene som kjøres. I mellomleddet kommuniserer de gjennom å bruke message sharing metoden. De kommuniserer aldri direkte og sender kun meldinger gjennom micro kernelen.
I en sammenlikning av micro-kernel og monolithic kerneler fant vi noen svakheter mellom disse to modellene:
Når det kom fram til utførelse av kommandoer er micro-kernelen tregere enn monolithic kernelen. Det trengs også mer kode til for å skape en micro-kernel (akash1295, 16.Aug 2019). Det er også dyrere å utføre service i et micro-kernel system i forhold til monolithic kerneler.

2.22

What are the advantages of using loadable kernel modules?
Fordelene med å bruke loadable kernel modules er at du trenger ikke implementere all funksjonaliteten inn i systemet. De blir kun brukt når man f.eks. starter opp eller når operativsystemet brukes.
Ulempen med å ikke ha loadable kernel modules er at uten de, så vil et operativsystem måtte inkludere alle mulige funksjonalitetene kompilert direkte i base-kernelen. All den funksjonaliteten vil ligge ubrukt i memory og ta opp masse plass. Det ville også ført til at brukere må re-builde og reboote base-kernelen hver gang de trenger en ny funksjonalitet (*Loadable Kernel Module*, 2021)
Loadable kernel modules blir til daglig brukt i OS som Linux, FreeBSD, macOS, NetWare, VxWorks og Solaris.

2.23

How are iOS and Android similar? How are they different?
Likhetene med iOS og Android er at begge er laget på samme type kernels og er laget på en måte som utnytter software stacks. Forskjellene er at iOS er skrevet i Objektiv-C språk og er

kodet som “native code”, altså i maskinspråk. Derimot er Android skrevet i Java og bruker en VM.

Noen flere likheter med iOS og Android er:

- De generelle funksjonene i begge systemene er veldig like, som f.eks.: Ringing, sende meldinger, facetime(video chat) og stemmeaktivering.
- Begge systemene kan utnytte 4G nett, noe som er uhyrlig viktig siden mobile enheter med internett tilgang trenger nettverksmuligheter utenfor sitt eget hjem.
- Begge systemene har gode sikkerhets muligheter gjennom apper de kan laste ned og bruke, dersom appene gir brukerne tilgang og eventuelle begrensninger som minsker sjansen for “leaking” av data.

Noen flere ulikheter med iOS og Android er:

- iOS er et mye mer avstengt system enn det Android er. Brukere har veldig lite kontroll/tilgang til iOS systemet mens Android brukere har mye mer tilgang og kan gjøre endringer i systemet som de ønsker.
- Siden Android sitt software er tilgjengelig for flere å bruke som f.eks. Samsung og LG, så er det større sjans for å finne problemer med kvaliteten på billigere mobiler. iOS derimot er kun eid av Apple, noe som gjør at kvaliteten er nesten garantert siden Apple ikke har så mange forskjellige modeller på systemet.
- Systemene har forskjell på hastighet utover bruksperioden. iOS har bedre levetid enn Android i forhold til hastighet.

(Castro K, 2018)

2.24

Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.

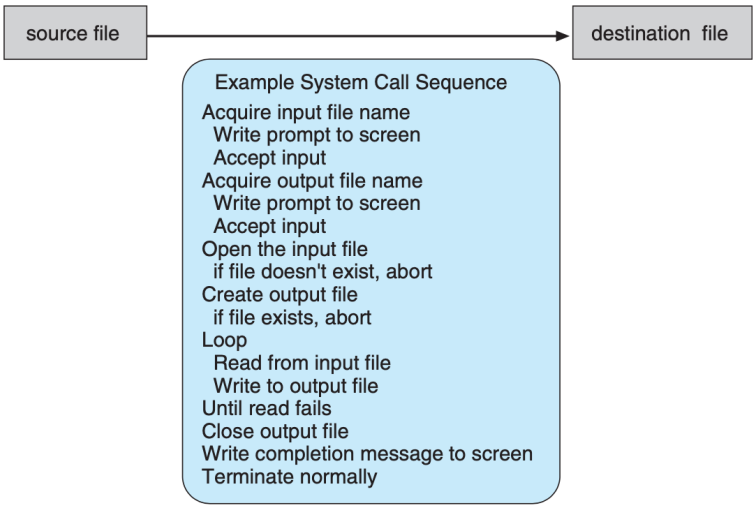
Standard Java API og Virtual machines blir vanligvis brukt til desktop og server systemer. Opprinnelig er de ikke egnet til mobiler som Android og apple. Siden de ikke er egnet, så har Google laget en ny API og virtual machine som funker på mobiler. Den er kjent som “Dalvik virtual machine.”

Dalvik virtual machine (DVM) bruker noen viktige komponenter/features fra java som memory management og multi-threading. Programmene som blir skapt blir først konvertert inn i JVM (Java Virtual Machine) og blir deretter interpretert til DVM bytecode.(D. Meador,

2019) All koding fra JVM blir omformulert gjennom DVM for å passe til mindre håndholdte mekanismer som ikke har like stor strømkapasitet.

2.26

In Section 2.3, we described a program that copies the contents of one file to a destination file. This program works by first prompting the user for the name of the source and destination files. Write this program using either the Windows or POSIX API. Be sure to include all necessary error checking, including ensuring that the source file exists.
Once you have correctly designed and tested the program, if you used a system that supports it, run the program using a utility that traces system calls. Linux systems provide the strace utility, and Solaris and Mac OS X systems use the dtrace command. As Windows systems do not provide such features, you will have to trace through the Windows version of this program using a debugger.



figur 2.5 fra Silberchatz side 63

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
```

```
    //input filnavnet man vil kopiere fra, legges is variabelen readFile
    printf("hvilken fil vil du kopiere?: ");
    char inputStr[20];
    scanf("%s", inputStr);
    const char* readFile = inputStr;
    //åpner filen som er gitt som input og gir den en file descriptor
    //O_RDONLY gjør at programmet kun kan lese fra filen
    int fdRead = open(readFile, O_RDONLY);

    //feilsjekking
    if (fdRead == -1) {
        perror("feil under åpning av input filen");
        exit(EXIT_FAILURE);
    }
    else{
        printf("(%s mottat)\n\n", inputStr);
    }

    //tar imot input for hva den nye filen skal hete
    printf("Hva vil du at den nye filen skal hete: ");
    scanf("%s", inputStr);
    const char* writeFile = inputStr;

    //åpner denne filen og gir den en file descriptor
    //O_CREAT hvis filen ikke eksisterer fra før av, lager programmet den
    //O_WRONLY programmet kan kun skrive til filen,
    //O_TRUNC trunkerer filen (sletter alt eksisterende innhold)
    //S_IRUSR gir brukeren tillatelse til å lese fra filen
    //S_IWUSR git brukeren tillatelse til å skrive i filen
    int fdWrite = open(writeFile, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);

    if (fdRead == -1||fdWrite == -1) {
        perror("feil under åpning av output fil");
        exit(EXIT_FAILURE);
    }
    else
    {
        printf("(%s mottat)\n\n", inputStr);
    }

    //variabler til å lese fra filen
    char c;
    int bytes;
    //loop for å lese fra den ene filen og skrive det over i den andre
    while((bytes = read(fdRead, &c, sizeof(c))) > 0){
        write(fdWrite, &c, sizeof(c));
    }

    //printer i terminalen at skrivingen er ferdig
    printf("Tekst kopiert!\n");

    //lukker filene
    close(fdRead);
    close(fdWrite);

    exit(EXIT_SUCCESS);
}
```

Output i terminalen:

eksempel suksessfull kopiering:

```
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$ ls
ReadWriteProgram.c      readFile.txt
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$ gcc ReadWriteProgram.c -o ReadWriteProgram
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$ ./ReadWriteProgram
hvilken fil vil du kopiere?: readFile.txt
(readFile.txt mottat)

Hva vil du at den nye filen skal hete: writeFile.txt
(writeFile.txt mottat)

Tekst kopiert!
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$ ls
ReadWriteProgram        ReadWriteProgram.c      readFile.txt            writeFile.txt
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$
```

eksempel feil (read fil finnes ikke):

```
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$ ls
ReadWriteProgram.c      readFile.txt
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$ gcc ReadWriteProgram.c -o ReadWriteProgram
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$ ./ReadWriteProgram
hvilken fil vil du kopiere?: read.txt
feil under åpning av input filen: No such file or directory
Kristians-MacBook-Pro:oppgave2_26 kristianskibrek$
```

Kommentar til programmet:

Programmet er skrevet på macOS big sur og ligger i Github. Det tar utgangspunkt i programmet beskrevet i seksjon 2.3. Programmet skal kopiere informasjonen i en fil til en annen fil. Det første programmet skal gjøre er å spørre brukeren om navnet på filene. Så må programmet åpne input filen. Her kan det skje feil som at filen ikke finnes, eller at filen har beskyttet tilgang. Tilhørende feilsjekking tar derfor sted. Hvis filen du vil skrive til ikke eksisterer fra før, så lager programmet den selv, om den finnes fra før av så trunkerer programmet filen (sletter alt eksisterende innhold), når den åpnes). Programmet går så inn i en loop hvor den leser data fra den ene filen og skriver det over til den andre, en byte av gangen. Når det er ferdig lukker programmet begge filene og avslutter. Oppgaven spesifiserer at programmet skal skrives etter enten Windows eller POSIX APIen.

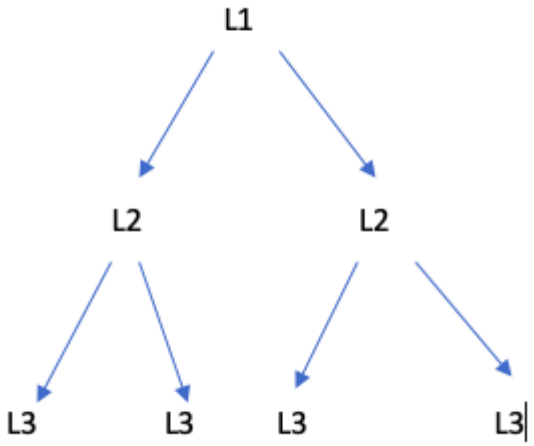
Kapittel 3

3.1

Using the program shown in Figure 3.30, explain what the output will be at LINE A.
Her har vi et «parent» som vil få value 5 på grunn av linjen int value = 5. Dette betyr at «child» til parent får det samme, men hvis child blir påvirket så kan ikke det påvirke parent siden child er en «duplicate». Så output i linje A blir at valuen til parenten blir 5 (Michael Kerrisk, 2021).

3.2

Including the initial parent process, how many processes are created by the program shown in Figure 3.31?
Hver gang en fork system call blir kalt lager det en ny prosess som inkluderer en kopi av original prosessen. Dette gjør det enklere for parent prosessen til å kommunisere med barna sine. Siden en fork blir kalt så har den laget et «parent og child». Så ser med at fork system call skjer igjen så må den duplisere de to som gjør at med har 4. Dette gjør med igjen siden vi ser at det blir en til fork system call så blir dette 4 + 4 som er 8. Det er 8 prosesser en parent og 7 children.



Dette er trestrukturen som prosessene danner. Her ser vi ikke forelderen, her ser vi bare barnene i denne trestrukturen. (Kadem Patel, “Geeksforgeeks.com”, 2019)

3.3

Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

"Concurrent processing" betyr at flere prosesser konkurrerer om ressurser. I en cpu core må en prosess stoppe for å la andre prosesser begynne. Apple brukte ikke denne funksjonen ikke på grunn av den ikke kunne det, men heller siden det ville tatt opp mye batteri og det var bekymringer for at det ville ta opp mye minne som på den tiden der telefoner ikke hadde såpass mye plass var en stor grunn hvorfor de måtte ta dette opp. Og en til er «perfomance». Apple ville helst at telefonene ville funke optimalt så det som skjer I bakgrunnen kunne lett påvirke det som skjedde på skjermen til brukeren spesielt på IOS4. I dag bruker Apple concurrent processing på flere måter i sine enheter ved å gjøre multitasking mulig ved å bytte mellom programmer både på telefonen og mac.(Silberschatz, et al., 2013, s .115) (Apple. 2012) .

3.4

The Sun UltraSPARC processor has multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

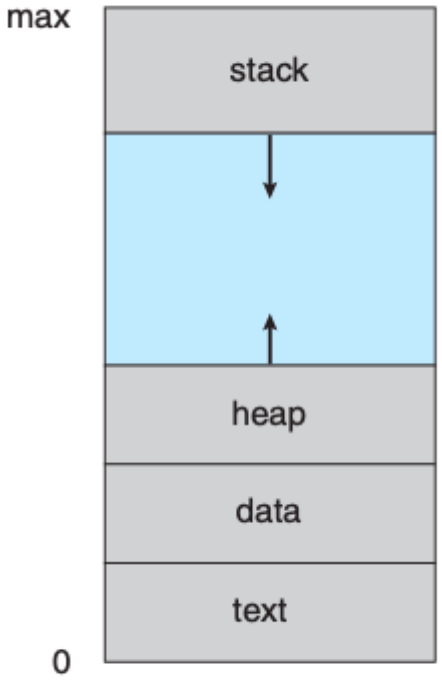
Det er enkelt siden det skifter bare pekeren til den nåværende «register set». Hvis det er mer aktive prosesser enn "register sets" så følger systemet med kopier av «register data» til og fra minen. Hvis den er i minen så må det bli lastet inn i en «register set» (Silberschatz, et al., 2013, s .138) .

3.5

When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?

a. **Stack**
b. **Heap**
c. **Shared Memory segments**

Når en prosess blir lagt ved fork så deler parent og child **stack** som inneholder midlertidig data som funksjonens parameter, returadresse og lokale variable. De kan og dele **heap** som er minne som er dynamisk tildelt i løpet av prosessens driftstid(Silberschatz, et al., 2013, s .106). Men "shared memory segments" blir ikke kopiert til barne prosessen.



Figur "3.1" fra boken "Operating system concepts ninth edition" (Silberschatz, et al., 2013, s .107.)

Her ser vi figur 3.1 fra boken som viser strukturen av en prosess i minen (Silberschatz, et al., 2013, s .107).

3.6

Consider the “exactly once”semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether “exactly once” is still preserved.

«Exactly once» semantikken sørger for at en prosedyre blir utført bare en gang altså «Exactly once» men blir kombinert med noen funksjoner av “at most once” . Det innføres og at vi tar risikoen at serveren aldri ta imot forespørselen derfor er det viktig å implementere “at most once” og. Forskjellen for “exactly once ” og “at most once” er at serveren tilrettelegger at RPC(Remote procedure call) kallet var utført og mottatt ved å sende et ACK(acknowledgement) til “exactly once” semantikken. ACK meldinger er vanlig blant nettverker. Bakgrunnen for denne algoritmen er altså at klienten skal sende en RPC til serveren med et tidsstempel som serveren lagrer . Etter dette så venter klienten på to utfall enten at de for en ACK meldingen fra serveren eller det vil skje et pause. Hvis det skjer et pause med ACK så vil det antyde at serveren kunne ikke gjennomføre prosedyren , så må klienten sende på ny hver RPC kall med jevne mellomrom til de får ACK for et RPC kall. . Hvis klienten ikke for ACKen er det to grunner til dette enten at RPC var aldri mottatt av serveren eller at RPC var mottatt av serveren, men ACK var mistet under prosessen. Ved at RCP var ikke mottatt av serveren skjer det at serveren ved hjelp av ACK kan serveren klare å motta RPC og utføre det. Hvis utfallet ved at ACK er mistet under prosessen skjer, då vil serveren ser på tidsstemplene og identifiserer at RPC er et duplikat og utfører ikke det for andre gang. Når dette skjer så må serveren sørge for å sende en til ACK til klienten for å si at det har utført RPC kallet.(Silberschatz, et al., 2013, s .140-142) (Remote Procedure Call(RPC). n .d).

3.7

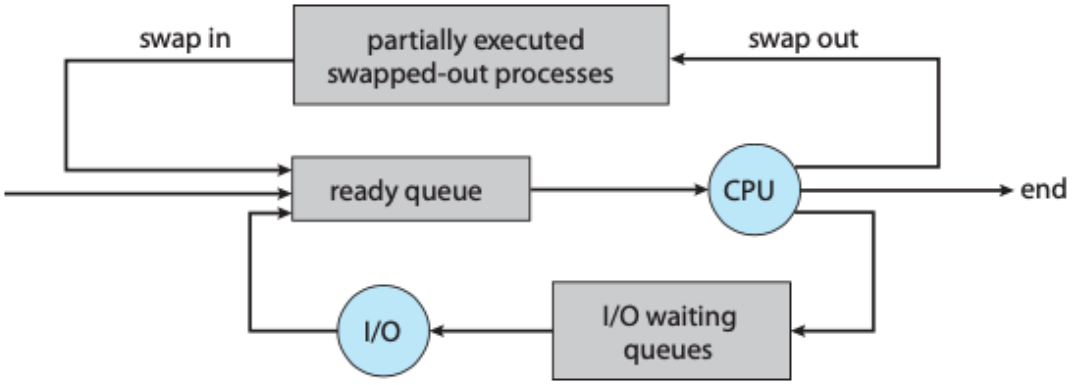
Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the “exactly once” semantic for execution of RPCs?

For å utføre en “exactly once” må vi ta risikoen at serveren vil aldri ta imot forespørselen. For å gjøre dette må serveren implementere “at most once”. I tillegg til dette må klienten få en ACK(acknowledgement) melding at RPC(Remote Procedure Call) meldingen kallet ble mottatt og utført. De fleste servererer følger opp og lagre informasjon som kommer i følge av RPC operasjoner som ble mottatt, og om hvordan de utførte oppgaven enten det var utført eller andre resultater ifølge disse oppgaver/prosesser. Men når det oppstår en «crash» og en RPC melding kommer fram kan då serveren sjekke om det var en RPC som var utført før og kan gjøre «exactly once» utførelser av RPC(Silberschatz, et al., 2013, s .132, s.140-143) (Remote Procedure Call(RPC) n.d).

3.8

Describe the differences among short-term, medium-term, and long- term scheduling.

En short term scheduling velger mellom de prosessene som er klare til å bli utført og tildeler CPU til en av dem. The long term scheduler velger prosesser fra for eksempel en disk og lader dem inni minne for å bli utført mens en medium term scheduler noen ganger kan være fordelaktig til å fjerne en prosess fra minne og derfor redusere graden av multiprogrammeringen. Senere kan prosessen bli tildelt igjen i minne og utførelsen kan fortsette hvor den ble kuttet av. Altså en medium scheduler bytter mellom prosessene dette kan være bra siden hvis det blir for mye tildelt i minne, må minne bli litt mer fritt opp som gjør en bytte mellom prosesser en lur ide. I medium-term scheduler så bytter de mellom prosessene, altså når jeg sier de fjerner en prosess så mener jeg at de bytter om prosesser. Først tar de en prosess ut så tar de en annen prosess inn igjen. Senere kan de ta i bruk den prosessen de byttet ut.



Figur “3.7” fra boken “Operating system concepts ninth edition” (Silberschatz, et al., 2013, s .114.)
Her ser vi medium-term scheduling. Her ser du at en process blir byttet ut og stiller seg i kø til å bli byttet inn igjen. Då kommer en ny prossess som er byttet inn i CPUen helt til det blir utført eller byttet ut igjen.

3.9

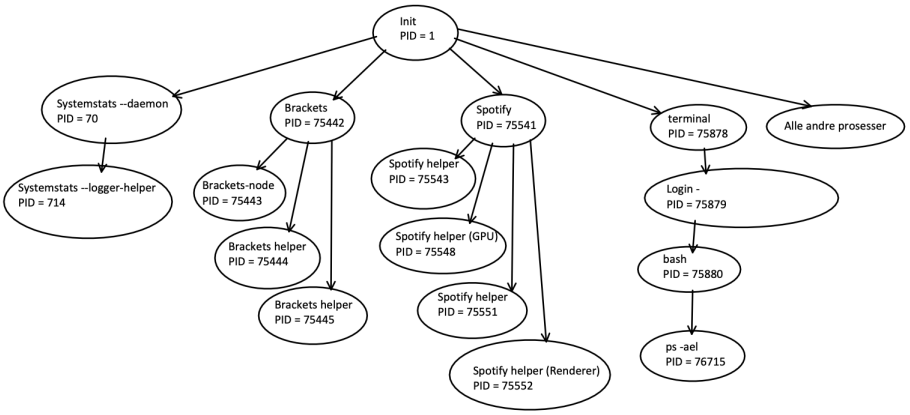
Describe the actions taken by a kernel to context-switch between processes.
Når en «context switch» blir utført må kernel lagre konteksten av den gamle prosessen inne i sin PCB også lade den lagrede konteksten av den nye prosessen som er planlagt til å bli utført (Silberschatz, et al., 2013, s .114).

3.10

Construct a process tree similar to Figure 3.8. To obtain process information for the UNIX or Linux system, use the command ps -ael.

Når ps -ael ble kjørt kom det opp over 400 prosesser, men de fleste av prosessene hadde ingen barne-prosesser. Jeg satte derfor de barneløse prosessene som en gren på treet kalt "andre prosesser"
For å lage treet er det to ting ved prosessene man skal se etter. Det er PID og PPID. PID står for Process ID, alle prosesser har hver sin unike PID. PPID står for Parent Process ID og sier hva PIDen til forelder prosessen er. De fleste prosessene hadde PPID 1 som betyr at de er barn av init prosessen, som har PID 1.

Treet:



3.11

Explain the role of the init process on UNIX and Linux systems in regard to process termination.

Init-processen har PID 1, og alle brukerprosesser er avkom av denne prosessen. I henhold til prosess termination så har init prosessen funksjonen å ta over foreldreløse prosesser. Foreldreløse prosesser er prosesser hvor foreldre-prosessen har blitt terminated, men den har ikke kalt wait(). Init prosessen kaller wait() periodisk, som gjør at de exit statusen til de foreldreløse-prosessene. ((Silberschatz, et al., 2013, s 116)

3.12

Including the initial parent process, how many processes are created by the program shown in Figure 3.32?

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;

    for (i = 0; i < 4; i++)
       fork();

    return 0;
}
```

Når man kaller fork(); vil programmet splittes i to. Første runde i whileloopen blir en prosess til to, og neste runde vil det være to prosesser som blir splittet i to, resulterende i fire prosesser. Til slutt vil man stå igjen med 2^4 = 16 prosesser. Man kan se det ved å sette en printf() før return. Siden det ikke er noen if statements om hvilken prosess som skal utføre printen, så vil alle prosessene gjøre det, resulterende i at stringen blir printet 16 ganger.

```c
#include <stdio.h>
#include <unistd.h>

int main(){
   int i;

   for(i = 0; i < 4; i++){
      fork();
   }

   printf("Hello World \n");

   return 0;
}
```

kode 1.

Output i terminalen:

3.19

Using either a UNIX or a Linux system, write a C program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds. Process states can be obtained from the command

ps -l

The process states are shown below the S column; processes with a state of Z are zombies. The process identifier (pid) of the child process is listed in the PID column, and that of the parent is listed in the PPID column.

Perhaps the easiest way to determine that the child process is indeed a zombie is to run the program that you have written in the background (using the &) and then run the command ps -l to determine whether the child is a zombie process. Because you do not want too many zombie processes existing in the system, you will need to remove the one that you have created. The easiest way to do that is to terminate the parent process using the kill command. For example, if the process id of the parent is 4884, you would enter

kill -9 4884

Vi har tatt utgangspunkt i dette eksemplet på hvordan man lager en zombie prosess.
https://www.geeksforgeeks.org/zombie-and-orphan-processes-in-c/

Det har blitt modifisert slik at prosessene skriver ut at de er ferdige rett før de avslutter. På denne måten er det lettere å forstå hva som skjer. man ser at child-prosessen. Programmet er skrevet på og for et MacOS system.

Programmet:

```
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
```

```
{
   // Fork returns process id
   // in parent process
   pid_t child_pid = fork();

   // Parent process
   if (child_pid > 0){
      sleep(10);
         printf("parent process done \n");
         }
   // Child process
   else
         printf("child process done \n");
      exit(0);

   return 0;
}
```

kode 2

Output i terminalen:

kompilerer og kjører programmet

```
Kristians-MacBook-Pro:modul3 kristianskibrek$ gcc oppgave3_19.c -o oppgave19
Kristians-MacBook-Pro:modul3 kristianskibrek$ ./oppgave19
child process done
```

ps -l

```
Kristians-MacBook-Pro:modul3 kristianskibrek$ ps -l
  UID   PID  PPID        F CPU PRI NI       SZ    RSS WCHAN     S             ADDR TTY           TIME CMD
  501   629   627     4006   0  31  0  4296972   1728 -       S                0 ttys000    0:00.21 -
  501   742   629     4006   0  31  0  4270332    716 -       S+               0 ttys000    0:00.00 .
  501   743   742     2006   0   0  0        0      0 -       Z+               0 ttys000    0:00.00 (
  501   717   716     4006   0  31  0  4296972_  1656 -       S                0 ttys001    0:00.06 -
```

programmet er ferdig

```
Kristians-MacBook-Pro:modul3 kristianskibrek$ ./oppgave19
child process done
parent process done
```

Kommentar til programmet:

Programmet er skrevet på macOS. En zombie process skjer når en barneprosess termineres, men foreldreprosessen har ikke kalt "wait()"(Silberschatz, et al., 2013 side 121). I dette programmet skjer dette ved at vi forker programmet. I foreldreprosessen kaller vi sleep(10), som gjør at foreldreprosessen ikke gjør annet enn å bare vente. I mellomtiden så utfører barneprosessen hele programmet sitt og termineres. Siden foreldreprosessen fortsatt sleeper, og ikke ennå har kalt wait(); blir barneprosessen en zombie, ettersom den ikke får gitt foreldreprosessen exit statusen sin og bli tatt ut av minnet (geeks for geek, 2021). Vi kan nå kalle ps -l, som gir oss informasjon om prosessene som skjer i dette øyeblikket (IBM n.d).Under S (status) kolonnen kan vi se at prosessen med PID (process ID) 2843 er en zombieprosess, fordi den har status Z. Foreldreprosessen kan man finne ved å se på 2843 sin PPID (parent process ID). Vi ser at det er prosess 2842 som er foreldreprosessen, denne har status S som betyr at den har en "sleeping state". At prosessene har en + foran betyr at de opererer i forgrunnen (Kodipills S, 2014).

3.21

The Collatz conjecture concerns what happens when we take any positive integer *n* and apply the following algorithm:

$$n = \begin{cases} n/2, & \text{if n is even} \\ 3 \times n + 1, & \text{if n is odd} \end{cases}$$

The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1. For example, if *n* = 35, the sequence is
35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Write a C program using the fork() system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output 8, 4, 2, 1. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the wait() call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line.

Utgangspunktet til collatz delen er hentet fra

```c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(){

   //input med error checking, sjekker om input er en integer
   int num;
   printf("\nEnter an integer: ");
   int conv = scanf("%d", &num);

   if(conv == 0){ //om input ikke er en int, feiler programmet
      printf("that is not an integer\n");
   }
   else{

   //deler programmet it to prosesser
   pid_t pid = fork();

   if(pid == 0){ //child process

      printf("%u", num); //printer første nummer (input)

      //colatz conj.
      while(num > 1){

         if(num%2 > 0){ // oddetall
            num = 3*num + 1;
         }

         else{ //partall
            num /= 2;
         }
         printf(", %u", num);
         }

      printf("\n");
   }
   else{//parent process
      wait(NULL); //venter med å terminate til child prosenssen er ferdig
   }
}
         return 0;
}
```

kode 3

output i terminalen:

```
Kristians-MacBook-Pro:modul3 kristianskibrek$ gcc oppgave3_21.c -o oppgave21
Kristians-MacBook-Pro:modul3 kristianskibrek$ ./oppgave21

Enter an integer: 49
49, 148, 74, 37, 112, 56, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Kommentar:

Programmet er skrevet på macOS Big Sur. Det kan ta imot hvilke som helt integer (heltall), om man gir den noe annet, som en float eller string, vil programmet si at du ha gitt feil type input. Verifiseringen er hentet fra C - How To Verifiy If Input Type Is Integer, koden for collatz algoritmen er fra Gunasekaran, A. P. Programmet deler seg i to prosesser med fork(), en foreldreprosess og en barneprosess. Barneprosessen står for å utføre collatz algoritmen og printe resultatet. Foreldreprosessen kaller wait(), som gjør at den venter til barneprosessen er terminert før den terminerer seg selv.

3.22

In Exercise 3.21, the child process must output the sequence of numbers generated from the algorithm specified by the Collatz conjecture because the parent and child have their own copies of the data. Another approach to designing this program is to establish a shared-memory object between the parent and child processes. This technique allows the child to write the contents of the sequence to the shared-memory object. The parent can then output the sequence when the child completes. Because the memory is shared, any changes the child makes will be reflected in the parent process as well.
This program will be structured using POSIX shared memory as described in Section 3.5.1. The parent process will progress through the following steps:

a. **Establish the shared-memory object (shm open(), ftruncate(), and mmap()).**
b. **Create the child process and wait for it to terminate.**
c. **Output the contents of shared memory.**
d. **Remove the shared-memory object.**

One area of concern with cooperating processes involves synchronization issues. In this exercise, the parent and child processes must be coordinated so that the parent does not output the sequence until the child finishes execution. These two processes will be synchronized using the wait() system call: the parent process will invoke wait(), which will suspend it until the child process exits.

```c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <math.h>

int main(){
    //variabler for å lage shared memory objekt
    const int SIZE = 4069; //størrelse
    const char *name = "OS"; //navn

    int shm_fd; //file descriptor
    void *ptr; //pointer


    //input med error checking, sjekker om input er en integer
    int num;
    printf("\nEnter an integer: ");
    int conv = scanf("%d", &num);

    if(conv == 0){ //om input ikke er en int, feiler programmet
        printf("that is not an integer\n");

    }
    else{

    pid_t pid = fork();

    if(pid == 0){//child process
        //lager et shared-memory object
        shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
        //konfigurerer størrelsen
        ftruncate(shm_fd, SIZE);
        //memory mapper objektet
        ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

        const char *space = " ";

        //legger til første nummer
        sprintf(ptr, "%u%s", num, space);
        int lenghtOfNum = floor(log10(abs(num))) + 1;
        ptr += lenghtOfNum + strlen(space);

        //collatz
        while(num > 1){
```

```
        if(num%2 > 0){
            num = 3*num + 1;
        }
        else{
            num /= 2;
        }
        //legger til det nye nummeret
        sprintf(ptr, "%u%s", num, space);
        int lenghtOfNum = floor(log10(abs(num))) + 1;
        ptr += lenghtOfNum + strlen(space);
        }
    }
    else{//parent process
        wait(NULL);

        //åpner shared memory object
        shm_fd = shm_open(name, O_RDONLY, 0666);
        ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

        //skriver ut fra objektet
        printf("\n%s\n", (char *)ptr);

        //sletter objektet
        shm_unlink(name);

        printf("\n");
    }
}
    return 0;
}
```

kode 4

Output i terminalen:

```
Kristians-MacBook-Pro:modul3 kristianskibrek$ gcc oppgave3_22.c -o oppgave22
Kristians-MacBook-Pro:modul3 kristianskibrek$ ./oppgave22

Enter an integer: 49

49 148 74 37 112 56 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Kommentar:

Programmet er skrevet på macOS. Dette programmet tar utgangspunkt i kode 3. Forskjellen i dette programmet er at istedenfor at barneprosessen selv printer ut resultatet, så er det foreldreprosessen som gjør det. Fordi barneprosesser og foreldre prosesser har hvert sitt sett med data, blir det nødvendig å ha en måte for prosessene å utveksle dataen. Dette blir gjort

igjennom et shared memory object. Et shared memory object gir prosessene mulighet til å kommunisere ved at de kan skrive informasjon og lese informasjon fra det. Det er prosessene selv som bestemmer hvor og hvordan informasjonen lagres i objektet, ikke operativsystemet (Silberschatz, et al., 2013 s. 124-125). Det er også prosessene som har ansvar for at de ikke skriver til samme lokasjon samtidig, og at ikke en prosess leser samtidig som den andre skriver, og dermed ikke får med seg all informasjonen. Det er mange måter å løse dette på. I dette programmet er det løst ved at barneprosessen er en produsent som legger inn informasjon i objektet. Foreldre prosessen er en konsument og leser informasjonen. For å passe på at all informasjonen er skrevet inn i objektet før foreldreprosessen leser av så kaller den wait(), som sørger for at barneprosessen er helt ferdig. Metoden for å programmere shared memory object er hentet fra Silberschatz side 133 og 134.

3.23

Section 3.6.1 describes port numbers below 1024 as being well known — that is, they provide standard services. Port 17 is known as the quote-of-the-day service. When a client connects to port 17 on a server, the server responds with a quote for that day. Modify the date server shown in Figure 3.21 so that it delivers a quote of the day rather than the current date. The quotes should be printable ASCII characters and should contain fewer than 512 characters, although multiple lines are allowed. Since port 17 is well known and therefore unavailable, have your server listen to port 6017. The date client shown in Figure 3.22 can be used to read the quotes returned by your server.

```
import java.net.*;
import java.io.*;
import java.util.Random;

public class QuotesServer{

   public static void main(String[] args){
       Random randomgenerator = new Random();
       /*quotes from
https://www.briantracy.com/blog/personal-success/26-motivation
al-quotes-for-success/ */
       String quotes[] = {

       "The Best Way To Get Started Is To Quit Talking And
Begin Doing",
       "The Pessimist Sees Difficulty In Every Opportunity.
The Optimist Sees Opportunity In Every Difficulty",
```

```
        "Don’t Let Yesterday Take Up Too Much Of Today",
        "You Learn More From Failure Than From Success.",
        "It’s Not Whether You Get Knocked Down, It’s Whether
You Get Up.",
        "If You Are Working On Something That You Really Care
About, You Don’t Have To Be Pushed. The Vision Pulls You",
        "People Who Are Crazy Enough To Think They Can Change
The World, Are The Ones Who Do."
        };
        try{
            ServerSocket sock = new ServerSocket(6013);

            /*now listen for connections*/
            while(true){
            Socket client = sock.accept();

            PrintWriter pout = new
PrintWriter(client.getOutputStream(), true);

            /*write the quote to the socket*/
pout.println(quotes[randomgenerator.nextInt(quotes.length)]);

            /*close the socket nd resume*/
            /*listenin for connections*/
            client.close();
            }
        }
        catch (IOException ioe){
            System.err.println(ioe);
        }
    }

}
```

kode 5

Output i terminalen:

Server:

```
Kristians-MacBook-Pro:oppgave3_23 kristianskibrek$ javac QuotesServer.java
Kristians-MacBook-Pro:oppgave3_23 kristianskibrek$ java QuotesServer
```

klient:

```
Kristians-MacBook-Pro:oppgave3_23 kristianskibrek$ javac DateClient.java
Kristians-MacBook-Pro:oppgave3_23 kristianskibrek$ java DateClient
People Who Are Crazy Enough To Think They Can Change The World, Are The Ones Who Do.
```

Kommentar:

Programmet er skrevet på macOs big sur. Serveren i kode 5 er modifisert fra figur 3.21 side 138 i Silberschatz. Det har blitt lagt til et array med forskjellige quotes. Istedenfor at serveren sender datoen til klienten så sender den en tilfeldig quote fra arrayet. Man kan bruke programmet i figur 3.22 side 139 i Silberschatz som klient (denne ligger sammen med serveren i Github). Man må som vist ovenfor må man kjøre dem i hver sin terminal. Serveren må kjøres først, den vil bare gi deg en blinkende markør som indikerer at programmet kjører. Når du kjører klienten i en annen terminal vil serveren gi den en tilfeldig quote, og klienten vil printe den.

3.25

echo server

```
import java.net.*;
import java.io.*;

public class EchoServer
{
public static void main(String[] args) throws IOException
  {
   ServerSocket serverSocket = null;

   try {
        serverSocket = new ServerSocket(10007);
       }
   catch (IOException e)
       {
        System.err.println("Could not listen on port:
10007.");
        System.exit(1);
```

```
        }
    Socket clientSocket = null;
    System.out.println ("Waiting for connection.....");
    try {
         clientSocket = serverSocket.accept();
        }
    catch (IOException e)
        {
         System.err.println("Accept failed.");
         System.exit(1);
        }
    System.out.println ("Connection successful");
    System.out.println ("Waiting for input.....");


    PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(),
                                       true);
    BufferedReader in = new BufferedReader(
            new InputStreamReader(
clientSocket.getInputStream()));


    String inputLine;


    while ((inputLine = in.readLine()) != null)
        {
         System.out.println ("Server: " + inputLine);
         out.println(inputLine);


         if (inputLine.equals("Bye."))
             break;
        }
    out.close();
```

```
   in.close();
   clientSocket.close();
   serverSocket.close();
  }
}
```

kode 6

echo klient

```
import java.net.*;
import java.io.*;

public class EchoClient {
   public static void main(String[] args) throws IOException {

       String serverHostname = new String ("127.0.0.1");

       if (args.length > 0)
          serverHostname = args[0];
       System.out.println ("Attemping to connect to host " +
       serverHostname + " on port 10007.");

       Socket echoSocket = null;
       PrintWriter out = null;
       BufferedReader in = null;

       try {
           // echoSocket = new Socket("taranis", 7);
           echoSocket = new Socket(serverHostname, 10007);
           out = new PrintWriter(echoSocket.getOutputStream(),
true);
```

```
            in = new BufferedReader(new InputStreamReader(

echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: " +
serverHostname);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for "
                               + "the connection to: " +
serverHostname);
            System.exit(1);
        }
    BufferedReader stdIn = new BufferedReader(
                                    new
InputStreamReader(System.in));
    String userInput;

        System.out.print ("input: ");
    while ((userInput = stdIn.readLine()) != null) {
        out.println(userInput);
        System.out.println("echo: " + in.readLine());
            System.out.print ("input: ");
    }
    out.close();
    in.close();
    stdIn.close();
    echoSocket.close();
    }
}
```

kode 6

Output i terminalen:

Programmene må kjøres i hver sin terminal

kompilerer og kjører server:

```
Kristians-MacBook-Pro:oppgave3_25 kristianskibrek$ javac EchoServer.java
Kristians-MacBook-Pro:oppgave3_25 kristianskibrek$ java EchoServer
Waiting for connection.....
```

Kompilerer og kjører klient:

```
Kristians-MacBook-Pro:oppgave3_25 kristianskibrek$ javac EchoClient.java
Kristians-MacBook-Pro:oppgave3_25 kristianskibrek$ java EchoClient
Attemping to connect to host 127.0.0.1 on port 10007.
input:
```

Gir klient input, som fører til at serveren sender inputen tilbake:

```
Attemping to connect to host 127.0.0.1 on port 10007.
input: hello World
echo: hello World
input:
```

Server side etter at input er blitt gitt:

```
Waiting for connection.....
Connection successful
Waiting for input.....
Server: hello World
```

Kommentar:

Java echo server er hentet fra:

https://www.cs.uic.edu/~troy/spring05/cs450/sockets/EchoServer.java

Java echo klien er hentet fra:

https://www.cs.uic.edu/~troy/spring05/cs450/sockets/EchoClient.java

Disse programmene fungerer veldig likt som i forrige oppgave, de har et server-klient forhold. For å forstå serveren så kan vi dele den opp i tre deler: setup, input og output. Den første delen, setup, er fra første linje til linje 22. Denne delen av programmet kan tolkes som

delen hvor serveren blir en server. Den setter opp hvilken socket som skal lyttes til, i tillegg til å gi feilmeldinger om forskjellige feil skjer, for eksempel at ikke socketen kan brukes eller at tilkoblingen til klienten ikke fungerer. Neste delen, input er fra 22 helt til linje 42. Denne delen har ansvar for å ta imot input fra klienten. Den gjør dette med getOutPutStream() og lagrer det i variabelen inputLine. Siste delen av programmet, output, skjer i while loopen på linje 43. Hvis InputLine har innhold vil loopen kjøre. I loopen sendes InputLine tilbake til klienten og skrives ut i serverprogrammet. Om det klienten har sendt er "Bye.", vil programmet avslutte, noe som gjør at dette er en evig loop helt frem til klienten sier "Bye.". Klientprogrammet kan deles opp i setup og output/input. Klienten er på mange måter lik som serveren. den starter med å se om den finner en server å koble seg til. Om den ikke gjør det, gir den feilmelding. Om den suksessfullt kobler til serveren, fortsetter programmet. Input og output befinner seg begge i samme while loop, hvor det først tas imot input fra brukeren, så sendes det til serveren, hvor det sendes tilbake igjen til klienten.

3.26

Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process.For example, if the first process sends the message Hi There, the second process will return hI tHERE. This will require using two pipes, one for sending the original message from the first to the second process and the other for sending the modified message from the second to the first process. You can write this program using either UNIX or Windows pipes.

```
d#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include<sys/wait.h>
#include<ctype.h>

#define BUFFER_SIZE 18
#define READ_END 0
#define WRITE_END 1

//function
```

```
void revereseCaseString(char arr[], int start, int end)
{
   for (int i = start; i < end; i++){

      char bokstav = arr[i];

      if(islower(bokstav))
         arr[i] = toupper(bokstav);

      else
         arr[i] = tolower(bokstav);
         }
}

int main(void){

int fd[2];


int fd_2[2];

pid_t pid;

   /*create the pipe*/
   if (pipe(fd)==-1){
      fprintf(stderr, "Pipe failed");
      return 1;
   }
   if (pipe(fd_2)==-1){
      fprintf(stderr, "pipe failed");
      return 1;
   }

   /*fork a child process*/
   pid = fork();

   if (pid < 0){
      fprintf(stderr, "Fork Failed");
      return 1;
   }

   if (pid > 0){/*the parent process*/
      /*variables for messages reading and writing*/
      char write_msg[BUFFER_SIZE] = "THis Is a meSSage";
      char read_msg[BUFFER_SIZE];

      /*close the unused end of the pipe*/
      close(fd[READ_END]);
      close(fd_2[WRITE_END]);
```

```
    /*write to the pipe*/
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /*close the write end of the pipe*/
    close(fd[WRITE_END]);
    wait(NULL); //venter til child prosessen er ferdig med å lese fra pipen
    read(fd_2[READ_END], read_msg, BUFFER_SIZE);
    printf(" parent end : %s\n", read_msg); //printer det parent prosessen mottar fra pipen
}

else{/*child process*/

    char read_msg[BUFFER_SIZE];
    /*close the unused end of the pipe*/
    close(fd[WRITE_END]);
    close(fd_2[READ_END]);

    /*read from the pipe*/
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf(" child end : %s\n", read_msg); //printer det child prosessen mottar fra pipen

    revereseCaseString(read_msg, 0, BUFFER_SIZE);

    write(fd_2[WRITE_END], read_msg, strlen(read_msg)+1);

    /*close the write end of the pipe*/
}

return 0;
```

kode 7

Output i terminalen:

kompilerer og kjører programmet:

```
Kristians-MacBook-Pro:modul3 kristianskibrek$ gcc oppgave3_26.c -o oppgave26
Kristians-MacBook-Pro:modul3 kristianskibrek$ ./oppgave26
 child end : THis Is a meSSage
 parent end : thIS iS A MEssAGE
```

Kommentar:

Programmet er skrevet i macOS big sur. Fordi det er skrevet i macOS blir det et eksempel på UNIX pipes. Det består av to prosesser, hvor den første prosessen sender en string til den andre prosessen. Den andre prosessen inverterer strengen så store bokstaver blir små og små blir store. Så sender den andre prosessen den inverterte stringen tilbake til den første prosessen, hvor den printes. For å synkronisere dem er den første prosessen en forelder til den andre. Når foreldreprosessen har sendt stringen kaller den wait(), som gjør at den ikke utfører resten av programmet sitt før barneprosessen, hvor stringen inverteres og returners, er ferdig.

Utgangspunktet til programmet er figur 3.26 på side 155 av Silberschatz, som er et eksempel på et program med en pipe. Dette programmet har blitt modifisert til å bruke to pipes, en til hver vei.

Kapittel 6

6.1

A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of n.

Det finnes ti ulike CPU-scheduling algoritmer; First come first serve (FCFS), Shortest job first (SJF), Longest job first (LJF), Shortest remaining time first (SRTF), Longest remaining time first (LRTF), Round Robin scheduling, priority based scheduling (Non-preemptive), Highest response ratio next (HRRN), Multilevel queue scheduling og Multi level feedback queue scheduling.

Om vi har non-preemtive scheduling (preemtive og nonpreemtive scheduling er nærmere beskrevet i neste oppgave), så er antall mulige schedules n! (n factorial = n x n - 1 x n -2 x .. x 2 x 1), hvor n er antallet prosesser (Silberschatz, et al., 2013, ch.6, s.269). Med preemtive scheduling blir det vanskeligere å finne en formel for hvor mange mulige schedules det er, fordi noen prosesser vil bli utført i flere omganger.

6.2

Explain the difference between preemptive and nonpreemptive scheduling

Preemption innebærer at operativsystemet går inn og avbryter et program, uten behov for et samarbeid med det, med det formål å la programmet fortsette med oppgaven ved en senere anledning.

Forskjellene mellom preemtive scheduling og nonpreemtive scheduling er at:

Preemptive scheduling brukes når en prosess skifter fra kjøremodus til klarmodus, eller fra ventemodus til kjøremodus. Ressursene (hovedsakelig CPU-sykluser) blir anvist til prosessen i en begrenset periode, og blir deretter tatt bort, og prosessen blir igjen plassert i den klarkøen

hvis prosessen fremdeles har CPU-burst-tid igjen. Denne prosessen holder seg i køen til den skal utføre en oppgave på et senere tidspunkt.
Algoritmer som er baserte på preemptive schedulig er bl.a: Round Robin, Shorterst remaining time first, Priority (preemptive versjonen).

Non-preemptive scheduling brukes når en prosess avsluttes, eller når den går fra å kjøremodus til ventemodus. Når ressursene (CPU-sykluser) blir anvist til en prosess, holder prosessen CPUen til den blir avsluttet eller når den når ventemodus. Dersom non-preemptive scheduling ikke avbryter en prosess som kjører CPU midt i utførelsen, venter den i stedet til prosessen fullfører CPU-burst-tiden, og deretter kan den anvise CPU til en annen prosess. Algoritmer baserte på non-preemptive scheduling er bl.a: Shortest job first (SJF basically non preemptive) og Priority (non preemptive versjon). (Silberschatz, et al., 2013, ch.6, s.264).

Hovedforskjellene mellom dem er at:

1. Ved Preemptive scheduling er CPU tildelt prosessene i en begrenset periode, mens i non-preemtive scheduling blir CPU tildelt prosessen til den avsluttes eller skifter til ventemodus.
2. Utførelsesprosessen i preemptive scheduling avbrytes midt i utførelsen når en høyere prioritet kommer, mens utførelsesprosessen inon-preemptive scheduling ikke blir avbrutt midt i utførelsen og venter til den blir utført.
3. I preemptive scheduling er det overhead å bytte prosessen fra startmodus til kjøremodus, vise vers og opprettholde startkøen. Det er derimot ikke overhead å endre prosessen fra kjøremodus til startmodus i non-preemptive scheduling.
4. Preemptive scheduling fungerer ved at dersom en prosess som er høy prioritet kommer i startkøen, må prosessen som er lav prioritet vente lenge, i noen tilfeller ender den med å avbryte. På den annen side, i non-preemptive scheduling, hvis CPU blir tildelt til prosessen som har større burst-tid, kan det hende at prosessene med liten burst-tid må avbrytes.
5. Preemptive scheduling er fleksibelt, men det er òg COST assosiert (COST= European cooperation in science and technology) samt driftskostnader(overhead). Non preemptive scheduling har ikke driftskostnader, og er ikke fleksibelt samt ikke er COST assosiert.

6.3

Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

FCFS (First Come, First Served) er en scheduling algoritme hvor den første prosessen som blir lastet inn i queue er den første til å utføres. Vi regner i denne oppgaven med non-preemtive scheduling, betyr at at prosessen får utføre helt ferdig før neste prosess begynner å utføre.

Turnaround tid er forskjellen mellom arrival tid of completion tid (Geeks for Geeks 2020). Vi vet arrival time (AT) fra tabellen, og må derfor først finne ut når prosessene blir ferdige. Vi fant completion time (CT) i tabellen under ved å plusse sammen burst time i rekkefølgen prosessene ankommer queue.

P1 (8)	P2 (4)	P3 (1)

0 8 12 13

TAT = CT - AT

$TAT_{P1} = 8 - 0 = 8$

$TAT_{P2} = 12 - 0.4 = 11,6$

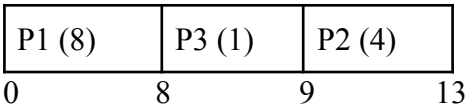
$TAT_{P3} = 13 - 1 = 12$

Gjennomsnittlig turnaround tid:
(8 + 11,6 + 12) / 3 = 10,53. (Silberschatz, et al., 2013, ch.6, s.266-267).

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

SJF (Shortest Job First) scheduling er en algoritme hvor den korteste jobben blir behandlet først. I likhet med forrige oppgave regner vi med non-preemtive scheduling.

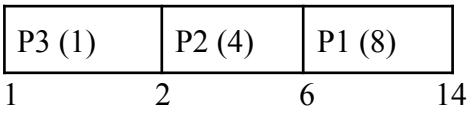
P1 har minst arrival time, og begynner derfor å utføres først. Siden P1 har en burst time på 8, så arriver både P2 og P3 i queue, samtidig som den utføres. Når P1 er ferdig er det på tide å laste inn en ny prosess. Den neste prosessen bestemmes utfra hvilken av prosessene i queue som er kortest. P3 har mindre burst time enn P2 og prosesseres derfor først. Vi bruker samme teknikk som i forrige oppgave for å finne completion time.



Utregning av avarage turnaround:
$T_{P1} = 8 - 0 = 8$
$T_{P3} = 9 - 1 = 8$
$T_{P2} = 13 - 0.4 = 12.6$
Gjennomsnittlig turnaround tid: (8 + 8 + 12.6) / 3 = 9,53

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P1 and P2 are waiting during this idle time, so their waiting time may increase. This algorithm could be called future-knowledge scheduling.

Ved å vente å ha CPUen idle (utfører ingen prosesser) i 1 unit så rekker alle prosessene å arrive før de begynner å utføres, siden P3 er den korteste prosessen, med en burst time på 1 vil denne settes til å utføres først. P2 er nest kortest, derfor blir den satt som prosess nummer to, og til slutt den lengste prosessen: P1.



Utregning av avarage turnaround:
$T_{P3} = 2 - 1 = 1$
$T_{P2} = 6 - 0.4 = 5.6$
$T_{P1} = 13 - 0 = 13$
Gjennomsnittlig turnaround tid: (1 + 5.6 + 13) / 3 = 6.53

6.4

What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?

Prosesser som trenger hyppigere service, for eksempel interaktive prosesser som redaktører, kan stå i kø med et mindre tidsvote. Prosesser uten behov for hyppig service kan være i kø med større kvante, og krever færre kontekstbrytere for å fullføre behandlingen, og dermed gjøre datamaskinen mer effektiv.

Hvis vi tar for oss round-robin (RR) scheduling som et eksempel så er den dens algoritme spesielt designert for tidsdelingssystemer. Det ligner på FCFS scheduling, men preemption blir lagt til for å muliggjøre systemet å bytte mellom prosesser. En liten tidsenhet, kalt tidskvote (time-quantum) er definert. En tidskvote er vanligvis fra 10 til 100 millisekunder i lengde. Den ferdige køen behandles som en sirkulær kø.

CPU scheduler går rundt den klare køen, og anviser CPUen til hver sin prosess i et tidsintervall på opptil 1 gang kvote (1 time quantum). For å implementere RR scheduling behandler vi igjen den klare køen som en FIFO kø av prosesser. Nye prosesser legges til

halen på den klare køen. CPU scheduler velger den første prosessen fra den klare køen, setter en tidtaker til å avbryte etter 1 gang tid kvote, og sender prosessen. (Silberschatz, et al., 2013, ch.6, s.271-273).

6.5

Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithm for each queue, the criteria used to move processes between queues, and so on. These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?

a. Priority and SJF

- Jobben med kortest tid blir høyest prioritert.

b. Multilevel feedback queues and FCFS

- Det laveste nivået av MLFQ er FCFS.

c. Priority and FCFS

- FCFS prioriterer den som har vært lengst

d. RR and SJF

- Ingen.

6.25

Using the Windows scheduling algorithm, determine the numeric priority of each of the following threads.

Vi bruker Schilberchatz side 295 for å finne denne tabellen, som viser hvordan man regner ut den numeriske prioriteten til threads:

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

a. A thread in the REALTIME PRIORITY CLASS with a relative priority of NORMAL

24

b. A thread in the ABOVE NORMAL PRIORITY CLASS with a relative priority of HIGHEST

12

c. A thread in the BELOW NORMAL PRIORITY CLASS with a relative priority of ABOVE NORMAL

7

Kapittel 8

8.1

Name two differences between logical and physical addresses.

To forskjeller mellom fysisk og logistisk adresser er at fysiske adresser er unike sammenlignet med logiske. Dette er for eksempel at fysiske minner blir delt opp i et program på forskjellige plasser og hver plass har en unik adresse altså fysisk adresse. Et til forskjell mellom adressene er at når et program blir kjørt då blir det generert en logistisk adresse, men en fysisk som nevnt er en plass i minen. Vi referer og til at logistisk adresse er virtuell adresse (Silberschatz, et al., 2013, s .355-357)

8.2

Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base– limit register pairs are provided: one for instructions and one for data. The instruction base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

Fordeler med en slik CPU er at koden blir mer sikker ved å gjennomføre på den måten, det vil si at det blir mer sikker mot dårlig/feil modifikasjoner som ville kommet av brukeren. En til fordel er at å dele data blir enklere mellom felter som trenger samme eller lik data i forhold til hva de trenger. Noen ulemper med dette blir då at dette kan ta tid å spleise koden slik med instruksjoner og data, men det er hvordan programmet skal utføre det, det gjelder. En til ulempe er at det kan koste mye siden en slik CPU kommer til å ha flere registre som driver fram utviklingen som vil drive fram kostnaden.(Silberschatz, et al., 2013, s. 352-353)

8.3

Why are page sizes always powers of 2?

Sider størrelser er kraft av 2 siden det er enklere til å bryte ned hver adresse inni bits som representerer en side, men og bits som representerer en offset. Dette kan tillate oss til å komme inn på disse sidene raskere og enklere istedenfor å bruke flere funksjoner for å oppnå det. Altså det blir enklere å kalkulere og raskere til å gjennomføre det for programme som til slutt gjør det enklere for oss å bruke (Silberschatz, et al., 2013, s .369-372) .

8.4

Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

- **How many bits are there in the logical address?**
- **How many bits are there in the physical address?**

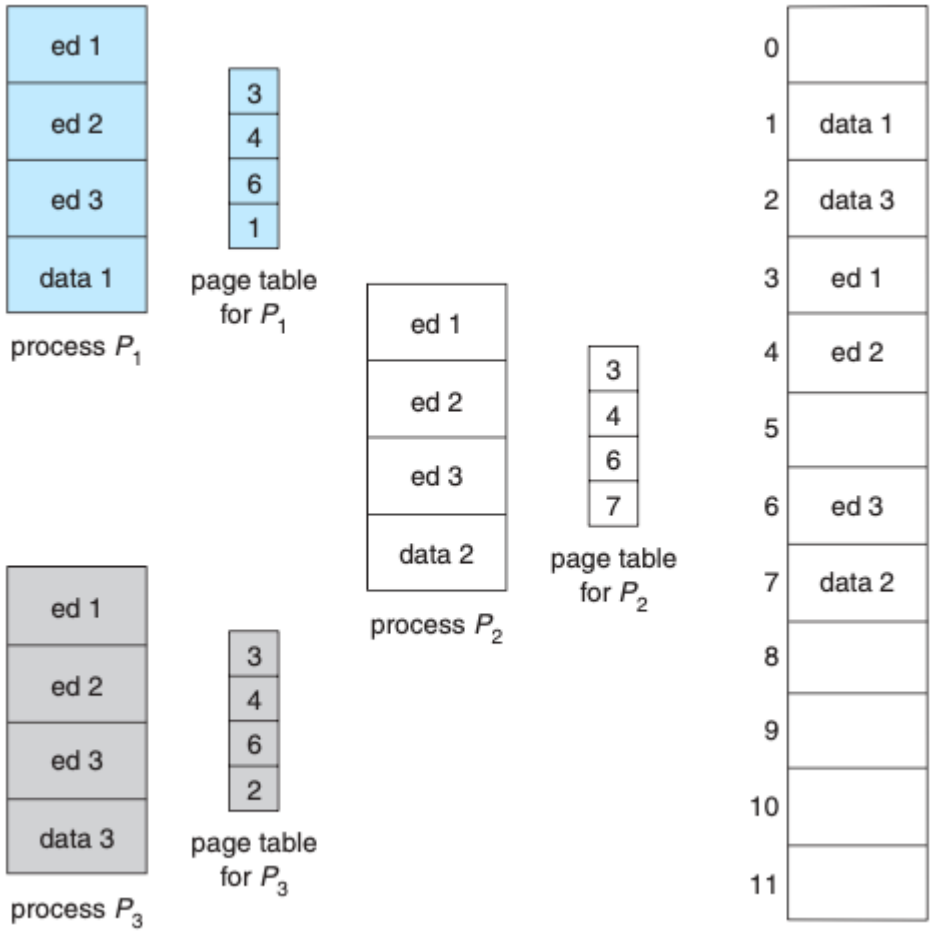
Hvis logistisk minne har 64 sider og hver eneste av disse sidene inneholder 1024 ord hver må vi ha 10 bits for ordene så trenger vi og bits for sidene som vil regne til slutt blir 6 til bits. Til sammen trenger vi **16 bits** for logistisk adresse. For fysisk minne er delt inntil 32 rammer og det tar altså 5 bits til å adressere hver eneste av de 32 rammene, ikke bare det men til å

adressere noen av de ordene (1024) trenger vi 10 bits. Til sammen så blir det **15 bits** for fysisk minne(Silberschatz, et al., 2013, s .370) .

8.5

What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on the one page have on the other page?

En stor fordel å ha to post i en sidetabell peker til en samme side ramme er at vi trenger ikke å lage kopier av den dataen. Et eksempel på dette er hvis vi har et par prosesser som skal dele samme data trenger ikke de å duplisere den i et fysisk minne. En til fordel er at det er kanskje billigere for at de prosessene kan enkelt lese fra der, dette er for eksempel hvis pekeren endres som da er jo en logisk adresse som oversettes til den fysiske adressen i minnerammen, dette gjør prosessen billigere. Når en oppdaterer en byte på en av sidene vil det bli oppdatert en byte i fysisk minne som vil si at det vil bli oppdatert. Ulempe med dette er at hvis dataen skulle være skilt fra hverandre så skaper det et problem. Dette er siden rammen i den fysiske minnen bør kanskje ikke endres heller den skal være read only, og hvis den endres så må den dupliseres nesten som “shared memory”, dette ligner og på forkin i Github. Hvis dette hadde skjedd så hadde et operativt system kopiert data fra før og tilført det til en prosessor som kommer ikke til å forandre det.(Silberschatz, et al., 2013, s .376-377)



Figur “8.16” fra boken fra boken “*Operating system concepts ninth edition*” (Silberschatz, et al., 2013, s .377.)

Her er Figuren som viser fram hvordan å dele kode er i en “paging environment”

8.6

Describe a mechanism by which one segment could belong to the address space of two different processes.

Når en bruker først, gjennomfører noe i selve logistiske adresse skjer det flere ting. Siden en logistisk adresse har to deler , et segment nummer s og en offset inntil segment d. Segment nummer er brukt som en index i segment tabellen. Selve segment tabellen har forskjellige innganger for hvert segment, dette inneholder vanlig (base) adressen og en grense for den vanlige adressen. Hvis offset er større enn selve grensen til kernelen , betyr det at brukeren

bruker feil adresse til å gjennomføre. Hvis vi har to prosesser så betyr det at vi har to ulike adress mellomrom som vil hende at deres instruksjons mappe blir plassert forskjellig i fysisk minne. Hvis vi vil dele en av segmentene mellom adressene sin mellomrom så må segmentet enten være nøytral som vil bety at det ikke tilhører noen av disse segmentene, men vil heller ligge en felles plass av delt minne. Dette vil tillate begge prosessene til å gå innpå det delte minne og eventuelt bruke det hvis det trengs. Altså her vil det kunne bruke to adresser på et segment som vil tillate prosessene til å bruke et felles plass for delt minne som de kan bruke. (Silberschatz, et al., 2013, s .364, s.366)

8.7

Sharing segments among processes without requiring that they have the same segment number is possible in a dynamically linked segmentation system.

a. **Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.**
b. **Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.**

a. Statisk linking betyr at all kode og biblioteker brukt i programmet blir inkludert i executable filen på slutten av kompileringsprosessen. Man vil da stå igjen med en executable fil hvor alt som trengs for å kjøre den er inkludert. Ulempen med dette er at om to programmer bruker kode fra samme bibliotek, så vil det være lagret to ganger. Dynamisk linking løser dette ved å ikke linke bibliotekene til executable filen før runtime. Dette gjør at flere programmer kan bruke samme kopi av bibliotekene (Kumar K n.d).

   Om man med statisk linking skal kunne dele segmenter mellom prosesser, så ville en løsning vært å bruke pipes, som gir mulighet for utveksling av informasjon mellom prosesser.

b. Med paging er det veldig lett for flere prosesser å dele kode. Om flere prosesser har "reentrant kode" (kode som ikke endrer seg under utførelse), vil det være lett for flere prosesser å bruke denne koden. Kun en kopi av koden trenger å lagres i minnet og

prosessene mappes til samme fysiske kopi (Silberschatz, et al., 2013, s. 376-377). Problemet med denne metoden er at page numrene i page tablene må være de samme for at det skal funke. Derfor mener vi at løsningen for paging vil bli ganske lik som i oppgave A), men istedenfor å sende en segment adresse/id igjennom pipes ville man laget en page som man kan sende adressen eller id-en via pipes.

8.8

In the IBM/370, memory protection is provided through the use of keys. A key is a 4-bit quantity. Each 2-K block of memory has a key (the storage key) associated with it. The CPU also has a key (the protection key) associated with it. A store operation is allowed only if both keys are equal or if either is 0. Which of the following memory-management schemes could be used successfully with this hardware?

a. **Bare machine**
b. **Single-user system**
c. **Multiprogramming with a fixed number of processes**
d. **Multiprogramming with a variable number of processes**
e. **Paging**
f. **Segmentation**

a. En "bare machine" refererer til en maskin som executer instruksjoner direkte på logisk hardware uten noe operativsystem (“Bare machine” 2021). Fordi det ikke er noe operativsystem er det heller ingen standard I/O operasjoner eller lignende. Alt er opp til programmereren. Det er ingen minnebeskyttelse når det kommer til bare maskiner. Løsningen blir å sette CPUens key til å alltid være 0, for programmereren trenger å ha tilgang til alt i maskinen. Å passe på at minnet ikke overskrider ting det ikke skal blir også opp til programmereren.

b. Et “single user system” er et system som bruker et single user OS, et operativsystem som er designet for at kun en bruker om gangen (Wiesen G. 2021). Det er noen funksjoner man ikke vil at brukerprogrammer skal kunne utføre. Disse kalles for “priviligerte instruksjoner” Derfor vil det være nødvendig å ha forskjellige "moduser" i kjernen, en til brukerprogrammer og en til priviligerte instruksjoner kalt “supervisor mode” (Barnes R, 2017). I supervisor mode vil CPUens key settes til 0.

c. Siden hver 2K blokk i IBM/370 har en egen key så må prosessene plasseres inn i disse blokkene. Om en prosess kun er 1K må den likevel plasseres i en blokk på 2K. Disse blokkene vil da ha en nøkkel som kan brukes for å få tilgang til å bruke store operasjonen i denne blokken.
d. Samme som i oppgave C)
e. Med paging så måtte frames bli satt til å være 2K, igjen har man nøkler som tilhører frames og kan brukes for å bruke store operasjonen.
f. Med segmentering måtte segment størrelsen settes til 2K slik som i C), D) og E). Man finner fram til segmentene ved å bruke deres nøkkel.

8.9

Explain the difference between internal and external fragmentation.

For å forklare intern og ekstern fragmentering er det lurt å forklare paging og segmentering. Når prosesser skal lastes inn i minnet står man overfor det logisk problemet at man må ha et system for hvilke prosesser som skal plasseres hvor i minnet. Paging og segmentering er to løsninger på dette problemet. Paging er kort sagt at minnet deles i virtuelt minne og fysisk minne. Det fysiske minnet deles i mange deler kalt frames, og det virtuelle minnet deles i mange mindre deler kalt pages (pages og frames er som regel like store). Prosessene måles i hvor mange pages de opptar. I segmentering deles prosessene opp i forskjellige deler av forskjellige størrelser, som så lastes inn i minnet forskjellige steder (Parahar M, 2013).

Intern fragmentering skjer når man bruker paging. Fordi prosessene fordeles ut på pages med satte størrelser, så vil man ende opp med at prosesser ikke bruker alt det allokerte minnet. Om for eksempel man har pages på 4 bytes og man skal laste inn en prosess på 6 bytes, vil den bli tildelt to frames. Fordi to frames tilsvarer 8 bytes vil vi da stå igjen med 2 bytes minne som vi ikke får brukt til noe (tech differences n.d)

Ekstern fragmentering kan tenkes på som det motsatte av intern fragmentering. Det skjer når man bruker segmentering. Prosessene tar forskjellig mengde plass, forskjellige steder i minnet. Etter at prosesser har blitt tatt ut og inn av minnet ender man opp med at det er små blokker mellom prosessene som står ubrukt, og ikke har plass til prosesser. Løsningen på

dette problemet er å prøve å få plassert de ubrukte minneblokkene sammen, så de kan slå seg sammen til større blokker. (tech differences n.d).

8.10

Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linkage editor is used to combine multiple object modules into a single program binary. How does the linkage editor change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linkage editor to facilitate the memory-binding tasks of the linkage editor?

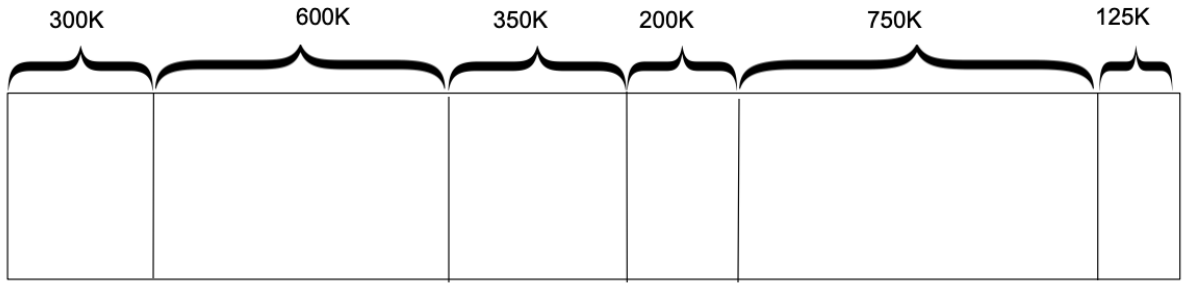
Jobben til linkage editoren er å kombinere koden til flere moduler sammen til et enkelt binært program (the free dictionary 2012). Som nevnt i oppgave 8.7 har vi to typer linking; statisk og dynamisk. Kort sagt er statisk linking at alle moduler som er en del av programmet er en del av en executable fil, man kan dermed ikke endre programmet uten å rekompilere det. Med dynamisk linking så blir ikke modulene i programmet satt sammen før runtime. Dermed kan endringer bli gjort i de forskjellige modulene i programmet, for eksempel en oppdatering av et bibliotek, uten at hele programmet må rekompileres.

Først tar vi for oss statisk linking. I kildekoden er referansene til andre moduler kun symbolske. Det er så jobben til kompilatoren å binde disse til re-alokerbare adresser som linkeren binder til absolutte adresser (Silberchatz, et al., 2013, s 354). Kompilatoren må derfor gi linkeren en liste over de re-alokerbare adressene som linkeren må gjøre om.

8.11

Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Rank the algorithms in terms of how efficiently they use memory.

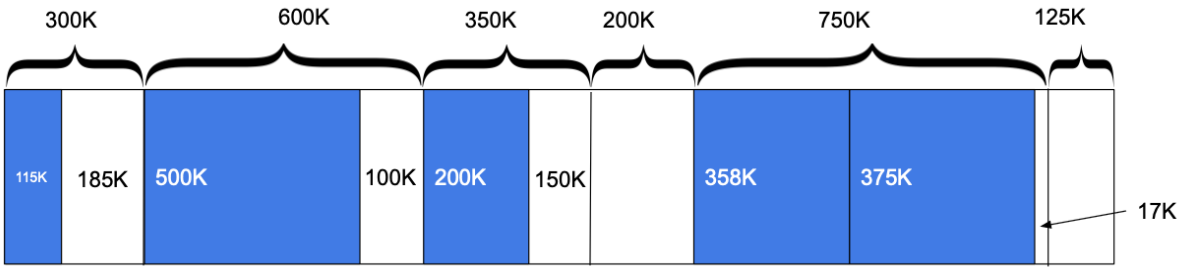
Vi kan lage en figur for å visualisere hva som skjer, først har vi minnet, delt i deler:



Så prosessene av forskjellige størrelser:

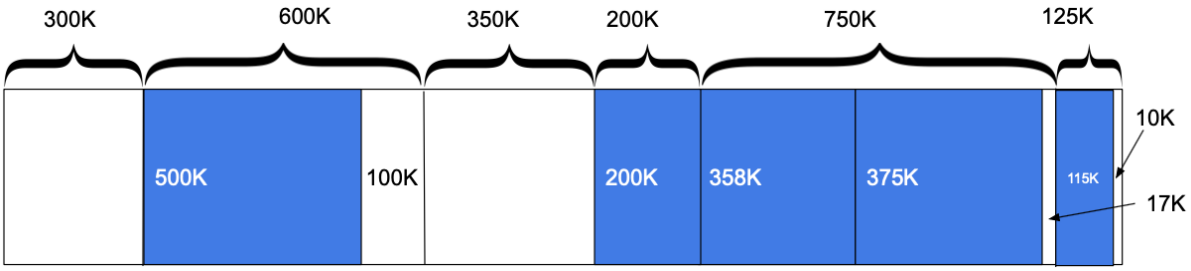
First fit algoritmen handler om å allokere første hullet i minnet som er stort nok. Søket starter ofte på starten av minnet og stopper med en gang det et stort nok hull har blitt funnet (Silberschatz, et. al., 2013, s. 363)

Den først prosessen (115K) vil derfor havne i 300K, 500K i 600K, 358K har ikke plass i 350K eller 200K, derfor havner den i 750K (dette legger igjen et hull på 392K), 200K havner da i 350K, og 375K i 392K (som var igjen fra 750K) (Silberschatz, et al., 2013, s.363)



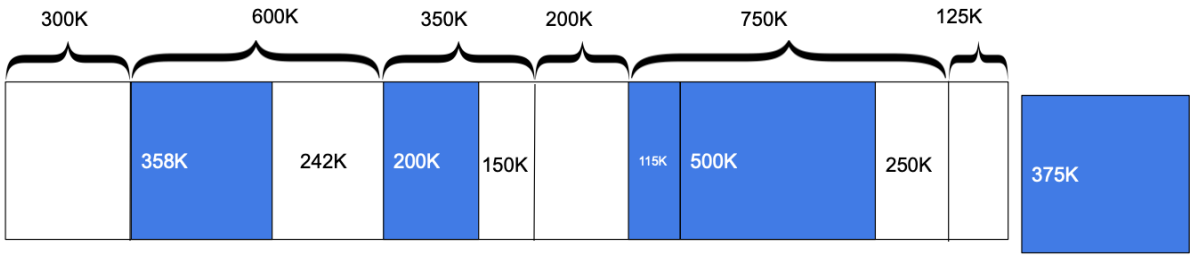
Best fit algoritmen handler om å finne det minste hullet som er stort nok. Derfor må hele minnet søkes gjennom først (ved mindre den er arrangert etter størrelse) (Silberschatz, et al., 2013, s. 363).

115K puttes i 125K, 500K i 600K, 358K i 750K, 200K i 200K og 375K i 392K



Worst fit algoritmen handler om å finne det største hullet. Dette gjør at det hullet som er igjen er størst mulig.

115K havner i 750K (legger igjen hull på 635K), 500K havner i 635K, 358K i 600K, 200K i 350K, det er ingen hull igjen med plass til 375K, så denne prosessen må vente til noen av de forrige blir ferdige og mer plass blir frigjort (Silberschatz, et al., 2013, s. 363)..



Hvilken av algoritmene er best?

Generelt er regelen at first fit og best fit er bedre enn worst fit på både tid og minneutnyttelse. Mellom first fit og best fit er det ikke en av dem som er klart best, men first fit er som regel raskere (Silberschatz, et al., 2013 s. 363). Vi ser i dette eksempelet at denne regelen stemmer bra. Både first fit og best fit fikk plass til alle prosessene, men det gjorde ikke worst fit algoritmen. Som tidligere nevnt er first fit raskere enn best fit, siden den ikke må lete gjennom hele minnet hver gang. Om man ville at systemet skulle være raskest mulig ville denne algoritmen vært det beste valget. Hvis minneutnyttelse er det viktigste i systemet mener jeg at i dette eksemplet at best fit ville vært det beste valget, ettersom at man endte opp med mindre intern fragmentering enn first fit. Det kan være uønsket med mye intern fragmentering i et system med stor vekt på minneutnyttelse.

Kapittel 9

9.1

Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

'Page faults' forekommer når prosesser prøve å aksessere en 'page' som ikke er i minnet. Selv om dette er svært usannsynlig vil operativsystemet, hvis det skjer, ha en løsning. Hvis pagen er invalid vil det bli satt en 'trap' til operativsystemet, som resulterer i at prosessen blir terminert, mens andre valide page vil bli satt inn. Instruksjonen blir dermed startet på nytt, men nå er de ønskede 'page' i minnet og tilgjengelig (Silberschatz, et al., 2013, ch. 9, s. 403-404). Hva operativsystemet gjør når det forekommer page faults:

1. Den interne tabellen sjekkes for å se om referansen har en gyldig minne adresse eller ikke.
2. Om referansen er ugyldig, termineres prosessen. Om den er gyldig, setter vi inn den gyldige pagen.
3. Vi finner en ledig frame
4. Vi setter opp en disk operasjon som skal lese inn pagen til den ledige framen.
5. Når operasjonen er ferdig, modifiserer vi tabellen for å indikere at pagen er i minnet.
6. Vi restarter instruksjonen som var avbrutt av en trap. Nå kan prosessen akksessere pagen, som om den alltid har vært i minnet (Silberschatz, et al., 2013, ch. 9, s. 403).

9.2

Assume that you have a page-reference string for a process with *m* frames (initially all empty). The page-reference string has length *p,* and *n* distinct page numbers occur in it. Answer these questions for any page-replacement algorithms

a) **What is a lower bound on the number of page faults?**
b) **What is an upper bound on the number of page faults?**

N – hvor mange frames i P.
M – mengde frames

a) ‘Lower bound’ på mengder page-faults er N

Laveste grense à beste case

Vi regner med at M > N

Dermed blir N lower bound page faults

b) ‘Upper bound’ på mengder page-faults er P

Høyeste grense à worst case

Vi regner med at M = 1

Da P page faults N – hvor mange frames i P.

9.3

Consider the page table shown in figure 9.30 for a system with 12-bit virtual and physical addresses and with 256-byte pages. The list of free page frames is *D, E, F* (that is, *D* is at the head of the list, *E* is second and *F* is last).

Programmer danner virtuelle adresser, mens prosessoren trenger fysiske adresser for å aksessere minnet (Silberschatz, et al., 2013, ch. 9, s. 412).

Virtuell adresse	Fysisk adresse
9EF	OEF
111	211
700	D00
0FF	EFF

9.4

Consider the following page-replacement algorithms. Rank these algo- rithms on a five-point scale from “bad” to “perfect” according to their page-fault rate. Separate those algorithms that suffer from Belady’s anomaly from those that do not.

Page-replacement algoritmer rangert på skala fra 1 (dårlig) til 5 (perfekt):

a) LRU replacement: Denne algoritmen erstatter pagen som ikke har blitt brukt på lengst tid av alle pagene. Når en page skal bli erstattet velger LRUen den med lengst ubrukte tidsperiode. Den har så og si like høy page-fault rate som OPT algoritmen. LRU-algoritmen kan også fjerne pages som skal til å bli brukt, likevel har den færre 'faults' enn FIFO-replacement. LRU blir ikke påvirket av Belady's anomaly, siden det er en del av såkalte «stack algoritmer». Implementasjonen av LRU ville ikke vært mulig uten maskinvare assistanse (Silberschatz, et al., 2013, ch. 9, s. 416).
Score: 3

b) FIFO replacement er den enkleste formen for page-replacement algoritmer. Den noterer seg når pagen ble lagt til I minnet. Når en page skal erstattes er den eldste valgt. Den er enkel å forstå og programmere, men ytelsen er ikke alltid så god. Det kan skje dårlige erstatningsvalg som øker page-fault raten og senker hastigheten på prosesser. Den er også påvirket av Belady's anomaly, som vil si at page-faults øker ved økt mengde allokerte frames (Silberschatz, et al., 2013, ch. 9, s. 413-414).
Score: 1

c) Optimal replacement har færrest page-faults, og er ikke påvirket av Belady's anomaly. Den erstatter pagen som ikke vil bli brukt i løpet av den lengste tidsperioden. Den er mye bedre enn FIFO-algoritmen med kun ni page-faults sammenlignet med FIFOs femten. Algoritmen er likevel vanskelig å implementere, fordi den krever fremtidig kunnskap om referanse-stringen. Den har dermed et begrenset bruksområde (Silberschatz, et al., 2013, ch. 9, s. 414-415).
Score: 4

d) Second-chance replacement. Inspiserer referanse-bit, og hvis verdien er 0 erstattes pagen, men hvis verdien er 1, får den en «ny sjanse». Den bygger på FIFO-replacement algoritmen, og lider dermed også av Belady's anomaly (Silberschatz, et al., 2013, ch. 9, s. 418-419).
Score: 2

Kapittel 14

14.1

What are the main differences between capability lists and access lists?

Hovedforskjellene mellom ‘capability list’ og ’access list’ er at access list handler om hvem som har tilgang, til for eksempel å lese en fil, altså det definerer et domene med et sett av tilgangsrettigheter til et objekt. Capability list derimot, er en liste for et domene som inneholder objektene og de funksjonene som er tillatt på disse. Funksjonen som skal utføres på et objekt føres gjennom et parameter (Silberschatz, et al., 2013, ch. 14, s. 636-637).

Capability lister er ikke direkte tilgjengelig, men er beskyttet av operativsystemet. Access lister kan derimot korrespondere direkte med det brukeren ønsker. Brukeren kan definere domenene som har tilgang til objektet, samt hvilke funksjoner det har. Det må likevel sjekkes at domenene har tilgang-rettigheter, noe som kan ta lang tid hvis access listen er lang. Selv om capability lister ikke kan interagere direkte med brukerens behov, kan de være nyttige for å lokalisere informasjon om en prosess (Silberschatz, et al., 2013, ch. 14, s. 638).

14.9

Why is it difficult to protect a system in which users are allowed to do their own I/O?

Det er vanskelig å beskytte et system hvor brukerne kan utføre sine egne I/O, fordi det er vanligvis kun kernelen som har tilgang til I/O funksjonaliteter; hvis brukeren dermed får kontroll over dette kan ikke sikkerheten garanteres når det ikke er maskinen som har kontroll; altså beskyttelse og andre egenskaper som forsikrer at systemet ikke blir misbrukt og at system-integritet blir opprettholdt (Silberschatz, et al., 2013, ch. 1, s. 22).

14.15

Discuss the strengths and weaknesses of implementing an access matrix using access lists that are associated with objects.

Fordelene med å implementere en access matrix gjennom en access list, er at brukeren får direkte tilgang, man kan selv definere domenet og funksjonene til objektet. En ulempe som følger er at disse domenene må sjekkes for tilgang-rettigheter noe som kan være tidkrevende hvis access listen er for lang. Det må sjekkes hver gang et objekt skal brukes, noe som er langtekkelig og kostbart i lengden (Silberschatz, et al., 2013, ch. 14, s. 638).

En annen styrke er at man effektivt kan sjekke en liste med generelle tilgangsrettigheter som kan bli definert på forhånd, før man eventuelt må gå gjennom den faktiske access listen (Silberschatz, et al., 2013, ch. 14, s. 636).

14.16

Consider a computing environment where a unique number is associated with each process and each object in the system. Suppose that we allow a process with number n to access an object with number m only if n > m. What type of protection structure do we have?

Vi har en hierarkisk ring-struktur.

(Silberschatz et al., 2013, s. 630-632)

14.17

Consider a computing environment where a process is given the privilege of accessing an object only n times. Suggest a scheme for implementing this policy.

Et forslag kan være å legge til en teller, så etter objektet er åpnet kan det få den til å øke for den. Så før du åpner selve objektet burde du sjekke telleren.

14.23

How does the principle of least privilege aid in the creation of protection systems?

The “principle of least privilege” går ut på at brukere, programmer og systemer skal ha nok privilegier til å utføre sine egne oppgaver. Hvis det skjer datafeil, skal det gjøre så lite som mulig skade på et operativsystem som utnytter seg av least privilege prinsippet. Så hvis det skjer en feil, skal ikke den fjerne brukeren få tilgang til hele systemet.

Man kan bruke et eksempel der en sikkerhetsvakt holder på en sikkerhetsnøkkel. Men denne nøkkelen har kun tilgang til de offentlige rommene i bygget, og ikke de private mer begrensede rommene, som gjerne kan ha mer sårbar informasjon. Så hvis denne

sikkerhetsvakten mister denne nøkkelen, skal det ikke være et stort problem, ettersom de som har fått tak i den ikke har tilgang til de "farlige" områdene av bygningen.

(Silberschatz et al., 2013, s. 626-627)

14.24

How can systems that implement the principle of least privilege still have protection failures that lead to security violations?

Et problem kan være at du implementerer beskyttelsesordningene i feil aspekter av operativsystemet, eller at de rett og slett foreløpig er ufullstendige, noe som kan gjøre andre deler sårbare. Hvis store bedrifter som f.eks. Google bruker denne type protection system, har de ekstremt mange områder, tjenester og linjer kode som krever beskyttelse.

(Silberschatz et al., 2013, s. 627-628)

Kapittel 15:

15.1

Buffer-overflow attacks can be avoided by adopting a better programming methodology or by using special hardware support. Discuss these solutions.

Man kan prøve å forhindre å bruke kode som er plassert i stack-segmentet til en prosess adresse sitt rom. Hvis du forhindrer å bruke kode fra stack-segmentet, kan man unngå problemet med buffer-overflow angrep. Man kan gjennomføre en slags tilnærming som bruker noen type grensekontroller som skal være imot buffer-overflows.

(Silberschatz et al., 2013, s. 663-666)

15.2

A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.

Et forslag kan være å gjøre det tilgjengelig på systemloggen for hver gang en bruker har vært innlogget i systemet, slik at man kan oppdage når noen har vært innlogget. Dette er enkelt og greit fordi da har man en god kontroll på hver gang noen har logget inn, og hvis det er noen mistenkelige innloggingstider blir det ihvertfall oppdaget.
For eksempel hvis du jobber innenfor et yrke der du holder på konfidensiell informasjon. Da vil det være veldig lurt å holde kontroll på innloggingstidene ettersom man da holder kontroll på om noen har logget inn på systemet på uønskede/mistenksomme tidspunkt.

15.3

What is the purpose of using a "salt" along with the user-provided password? Where should the "salt" be stored, and how should it be used?

Salt er et begrep som brukes i relasjon til kryptografi, hvor et salt er en mengde tilfeldig data som blir brukt som en ekstra input til en enveis funksjon som hasher data eller passord. De brukes til å gi ekstra sikkerhet til passord. Alle passord og data har et eget unikt salt. (Salt(kryptografi), 2021)
Når en bruker lager et passord, så vil systemet generere et tilfeldig tall (som er saltet) og kobler det til det bruker gitte passordet, krypterer stringen som oppstår og lagrer resultatet og saltet i en passord fil. Når man da skal sjekke passordet, så vill passordet bli presentert av brukeren og bli sammensatt med saltet for å sjekke likhet med det lagrede passordet. Siden saltet er forskjellig fra bruker til bruker, så kan ikke en passord knekker knekke 1 brukers passord ved å sjekke passordet, kryptere det og matche det opp mot alle de krypterte passordene samtidig.

15.8

Argue for or against the judicial sentence handed down against Robert Morris, Jr., for his creation and execution of the Internet worm discussed in Section 15.3.1.

Et argument mot dommen er at den var litt overflødig. Etter denne hendelsen har programmerere og generelt alle blitt mer våken mot slike svakheter og angrep på internettet.

Et argument for dommen er at etter ormen ble sluppet ut, tok det ekstremt lang tid og mye penger å fikse alt det ormen hadde gjort, så derfor føler jeg dommen var definitivt rett dømt.

15.9

Make a list of six security concerns for a bank's computer system. For each item on your list, state whether this concern relates to physical, human, or operating-system security.

1) Systemet burde være på et beskyttet sted med fysisk menneskelig vakthold.

2) Nettverket systemet opererer på må være sikret mot utestående mennesker som vil tukle med systemet. Fysisk, menneskelig og OS sikkerhet.

3) Modem tilgang enten fjernet eller begrenset: Fysisk, menneskelig sikkerhet.

4) Uautorisert deling av data blir stoppet eller logget: menneskelig, OS sikkerhet.

5) Backup media beskyttet med fysisk menneskelig vakthold.

6) Programmerere og ansatte må være til å stole på: menneskelig sikkerhet.

15.10

What are two advantages of encrypting data stored in the computer system?

Kryptering er en metode for å holde data trygg fra folk som ikke skal ha tilgang til den, eller fra folk du ikke vil skal bruke det eller se det. Kort sagt er kryptering en måte å sikre data på. Så to fordeler ved å kryptere data er at det sikrer og beskytter data, og det er lettere å ha kontroll over hvem som har tilgang og hvem har ikke tilgang.

Et godt eksempel på kryptering av data vil være enkel kryptering som f.eks. å bytte ut bokstaver i et ord med tall. La oss si du har laget passordet: Abcdefg. For å kryptere det kan vi enkelt og greit gjøre de om til tall ut i fra hvilken plass i alfabetet de har. Da blir passordet: 1234567. Når man da lager et skikkelig passord som du vil skal være komplisert, så kan du kryptere det på denne måten om du ønsker å gi tilgang til andre brukere (Ikke den vanskeligste måten å kryptere på men det er likevel kryptering).

15.12

Compare symmetric and asymmetric encryption schemes, and discuss the circumstances under which a distributed system would use one or the other.

Symmetrisk kryptering brukes vanligvis til kryptering av filer. Det innebærer at den samme krypteringsnøkkelen brukes til både å krypter og dekryptere informasjonen. Altså man bruker kun 1 nøkkel når man bruker symmetrisk kryptering." Det er en sikker løsning så lenge nøkkelen (koden) som brukes, kan holdes hemmelig, og den brukes gjerne til kryptering av filer på servere og sikkerhetskopier."

"Asymmetrisk kryptering er en form for kryptering hvor det lages to krypteringsnøkler som hører sammen". Når man bruker asymmetrisk kryptering må man ha 2 nøkler, du må ha en "privat nøkkel" og en "offentlig nøkkel".

Den private nøkkelen er som i navnet "privat", så den må holdes hemmelig, mens den offentlige nøkkelen er "offentlig", så den kan gis til hvem som helst.

I spørsmålets tilfelle, hvilket tilfelle skal disse brukes? Skatteetaten er et godt eksempel på asymmetrisk kryptering. Skatteetaten har en offentlig nøkkel som kan brukes av offentlige virksomheter, som da kan kryptere informasjon før det sendes videre. Denne informasjonen kan da kun dekrypteres med skatteetatens private nøkkel. Symmetrisk kryptering blir brukt i Microsoft Office. Her kan man kryptere dokumenter for videresending hvor man da lager eget passord. Men det kommer en sikkerhetsrisiko med dette, ved at du må sende nøkkelen til mottakeren siden det kun er 1 nøkkel i bruk.

15.14

Discuss how the asymmetric encryption algorithm can be used to achieve the following goals.

a. **Authentication: the receiver knows that only the sender could have generated the message.**
b. **Secrecy: only the receiver can decrypt the message.**
c. **Authentication and secrecy: only the receiver can decrypt the message, and the receiver knows that only the sender could have generated the message.**

Her kan vi lage vår egen simulasjon av asymmetrisk kryptering mellom 2 personer.
Vi kan gjøre følgende:

123 er public for senderen
456 er public for mottakeren
321 er privat for senderen
654 er privat for mottakeren

Disse er offentlige og private nøkler for bruk av kryptering og dekryptering.

a) Autentisering blir gjennomført ved å få senderen sende en melding som er kryptert med 321.
b) Sikkerhet blir garantert ved å ha senderen kode meldingen med 456.
c) Både autentisering og sikkerhet blir garantert ved å gjøre dobbel kryptering ved bruk av både 321 og 456.

Kilder:

Apple inc.(13.12.2012) Concurrency and Application Design. *developer.apple.com*. Hentet 16.05.2021 fra https://developer.apple.com/library/archive/documentation/General/Conceptual/ConcurrencyProgrammingGuide/ConcurrencyandApplicationDesign/ConcurrencyandApplicationDesign.html

Arne Jansen.(26.11.2018) Datakryptering. En måte å beskytte data mot innsyn på er å kryptere informasjonen. *ndla.no* Hentet:

https://ndla.no/nb/subject:25/topic:1:193107/topic:1:193135/resource:1:124782?filters=urn:filter:d97809a8-47b6-4d26-ae5c-1839f4c27940

Answers. (2012, 16. April). *How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?* Answers.com. Hentet 03.05.2021 https://www.answers.com/Q/How_could_a_system_be_designed_to_allow_a_choice_of_operating_system_to_boot_from._what_would_the_bootstrap_program_need_to_do

Bratbergsengen, K (2019). Cache. *SNL.* Hentet: 12.03.21. Fra: https://snl.no/cache
Boy R (2014, 15. oktober) Reading and Writing Files with POSIX, *youtube.com* hentet 05.05.2021 https://www.youtube.com/watch?v=TAOCe-pHSDw

Bhoyar, Kishor. (2017, 2. november). Why Should “accessing I/O device” be a privileged instruction? *Stackoverflow.com.* Hentet: https://stackoverflow.com/questions/22377709/why-should-accessing-i-o-device-be-a-privileged-instruction

Barnes R (17.08.2021) Supervisor mode *tutorialspoint.com* https://www.tutorialspoint.com/Supervisor-Mode-Privileged-Mode

Castro K. (10.10.2018) Message Passing Model of Process Communication. hentet (26.04.2021) https://www.tutorialspoint.com/message-passing-model-of-process-communication

Chakraborty, K. (2021, 25. Februar). *Firmware*. Techopedia.com. Hentet 14.03.2021 https://www.techopedia.com/definition/2137/firmware

Computer Sciences. (u. å). *Early Computers.* Encyclopedia.com. Hentet 04.05.2021 fra https://www.encyclopedia.com/computing/news-wires-white-papers-and-books/early-computers

Daria. (2000, 26. juli). Operativsystemets 4 hoveddeler. hentet 27.04.2021 fra https://www.daria.no/skole/?tekst=956

Eksepertgruppe: Aas, Einar, Espedal, Magnus, Schartum, Dag, Snaprud, Michael, Sørgard, Lars, Lie, Håkon, Undheim, Trond, Hafskjold, Christine (2004). Programvarepolitikk for fremtiden. *Teknologirådet.no*. Hentet: 15.03.2021. Fra: s.33, 36, 41: https://teknologiradet.no/wp-content/uploads/sites/105/2013/08/Rapport-Programvarepolitikk-for-fremtiden.pdf

El-Ghazawi, T., Frieder, G. (2003). Input-output operations. In (Ed.), *Encyclopedia of Computer Science*(Winter 2003). Hentet 14.03.2021 https://dl.acm.org/doi/10.5555/1074100.1074484

Geeks for geeks (08.03.2021) Wait system call in c. *geeksforgeeks.org* Hentet: 23.04.2021

https://www.geeksforgeeks.org/wait-system-call-c/

Geeks for geeks (2019) POSIX Shared Memory API *geeksforgeeks.org* Hentet 05.04.2021

https://www.geeksforgeeks.org/posix-shared-memory-api/

Geeks for Geeks (28. mars 2020) Difference between Turn Around Time (TAT) and Waiting Time (WT) in CPU Scheduling lastet ned 11.05.2021 fra https://www.geeksforgeeks.org/difference-between-turn-around-time-tat-and-waiting-time-wt-in-cpu-scheduling/

Geeksforgeeks. (2021, 1. april). *System Programs in Operating Systems*. Geeksforgeeks.org. Hentet 10.03.2021 https://www.geeksforgeeks.org/system-programs-in-operating-system/

Geeksforgeeks. (2020, 23. desember). Functions of Operating System. hentet 27.04.2021 fra https://www.geeksforgeeks.org/functions-of-operating-system/

Gunasekaran A P (06.10.2021) youtube_tutorials *Github.com* hentet 05.04.2021 https://github.com/arunprasaad2711/Youtube_Tutorials/blob/master/C_programs/14_While_loop/Collatz.c

Highintegritysystems. (u.å.). What is an RTOS. High Integrity Systems. hentet 02.05.2021 fra: https://www.highintegritysystems.com/rtos/what-is-an-rtos/

IBM (n.d) ps - Return the status of process *ibm.com* Hentet: 23.04.2021

https://www.ibm.com/docs/en/zos/2.1.0?topic=descriptions-ps-return-status-process

Incomputersolutions. (2021). *Question: Why Is The Kernel Separate From The Operating System?* Incomputersolutions.com. Hentet 10.03.2021 https://incomputersolutions.com/qa/why-is-the-kernel-separate-from-the-operating-system.html

Kadem Patel. (2019, 9. desember) fork() in C. Geeksforgeeks.com. Hentet fra: https://www.geeksforgeeks.org/fork-system-call/

Krishnamurthy A (2004) Multics: dynamic linking, *homes.cs.washington.edu* hentet 20.042.2021 https://homes.cs.washington.edu/~arvind/cs422/lectureNotes/dynlink-6.pdf

Kumar K (n.d) difference between static and dynamic linking, *cs-fundementals.com* Hentet 23.04.2021 https://cs-fundamentals.com/tech-interview/c/difference-between-static-and-dynamic-linking

Kodipilla, S (30.04.2014) Linux process state codes *thelinuxstuff.blogspot.com* Hentet 23.04.2021

http://thelinuxstuff.blogspot.com/2013/04/what-does-status-drsz-indicate-when-i.html

Kristi Castro (05.09.2018) *How are iOS and Android similar? How are they different?* Hentet fra: https://www.tutorialspoint.com/how-are-ios-and-android-similar-how-are-they-different

Liseter, Ivar M. (2020, 22. august), operativsystem i Store norske leksikon på snl.no. Hentet27. april 2021 fra https://snl.no/operativsystem

Meador, D. (2018, 3. September). *Major Activities of an Operating System with Regard to Secondary Storage Management*. Tutorialspoint.com. Hentet 10.03.2021 https://www.tutorialspoint.com/major-activities-of-an-operating-system-with-regard-to-secondary-storage-management

Meador D.(10. September 2018) Why Java programs running on Android systems do not use the standard Java API and virtual machine? hentet 30.04.2021 fra: https://www.tutorialspoint.com/why-java-programs-running-on-android-systems-do-not-use-the-standard-java-api-and-virtual-machine

Michael Kerrisk.(2021, 22. mars). Fork(2) - Linux Manual page. Man7.org. Hentet fra: https://man7.org/linux/man-pages/man2/fork.2.html

Onsman A. (10.10.2018) Shared Memory Model of Process Communication. hentet (26.04.2021)

https://www.tutorialspoint.com/message-passing-model-of-process-communication

Parahar M (18.11.2019), Difference between paging and segmentation tutorialspoint.com. Hentet 24.04.2021
https://www.tutorialspoint.com/difference-between-paging-and-segmentation

Rishabhjain12. (2019, 14. august). Privileged and Non-Privileged Instructions in Operating System. Geeksforgeeks.com. Hentet: https://www.geeksforgeeks.org/privileged-and-non-privileged-instructions-in-operating-system/

RealWorldCyberSecurity. (2020, 22. april). *Negative Rings in Intel Architecture: The Security Threats That You've Probably Never Heard Of.* Start it up. Hentet 04.05.2021 fra https://medium.com/swlh/negative-rings-in-intel-architecture-the-security-threats-youve-probably-never-heard-of-d725a4b6f831

Remote Procedure Call (RPC) (n.d). *Remote Procedure Call (RPC)*. Hentet 16.05.2021 fra https://web.cs.wpi.edu/~cs4514/b98/week8-rpc/week8-rpc.html

Rutnik, Mitja. (2021, 9. mars). What is Chrome OS and who is it for? Android Authority. hentet 02.05.2021 fra: https://www.androidauthority.com/what-is-chrome-os-1137371/

Silberschatz, A, Galvin, P B, Gagne, G (2013). Operating System Concepts - 9th Edition. New Jersey: John Wiley & Sons inc. Hentet: 12.03.21.
(Silberschatz et al., 2013, p. 67)

Shah Abdul Latif University. (2018). Ch1 introduction - Solution to Chapter 1 of Operating System Concept 9th Edition. *Studocu.com*. Hentet: https://www.studocu.com/row/document/shah-abdul-latif-university/operating-system/practical/ch1-introduction-solution-to-chapter-1-of-operating-system-concept-9th-edition/7025092/view

Sorber J (1. mai 2018) Reading and Writing Files in C, two ways (fopen vs. open) hentet 05.05.2021

https://www.youtube.com/watch?v=BQJBe4IbsvQ

Simmons, Chris. (2014, 21. mars). What is the difference between user and kernel modes in operating systems? Stackoverflow. Hentet 02.05.2021 fra https://stackoverflow.com/questions/1311402/what-is-the-difference-between-user-and-kernel-modes-in-operating-systems

Tutorialspoint. (2021). *Operating System - Services*. Tutorialspoint.com. Hentet 14.03.2021 https://www.tutorialspoint.com/operating_system/os_services.htm

Tech Differences (n.d), Difference between internal and external fragmentation *techdifferces.com*. Hentet 24.04.2021 https://techdifferences.com/difference-between-internal-and-external-fragmentation.html#Conclusion

Taylor C (23.09.2019), paging and segmentation *enterprisestorageforum.com* Hentet 24.04.2021 https://www.enterprisestorageforum.com/hardware/paging-and-segmentation/

The Free Dictionary (2012) Linkage editor *thefreedictionary.com* hentet (24.05.2021) https://www.thefreedictionary.com/linkage+editor

Whatis. (2012, September). Memory management. *Whatis.techtarget.com*. Hentet 10.03.2021 https://whatis.techtarget.com/definition/memory-management

Wikipedia (04.03.2021) Bare machine *wikipedia.com* hentet 23.05.2021 https://en.wikipedia.org/wiki/Bare_machine

Wiesen G (23.01.2021) What is a single user system? easytechjunkie.com hentet: 24.05.2021 https://www.easytechjunkie.com/what-is-a-single-user-operating-system.htm

Wikipedia (sist endret: 22. April 2021) Separation of mechanism and policy. Hentet 30.04.2021 fra: https://en.wikipedia.org/wiki/Separation_of_mechanism_and_policy

Wikipedia(sist endret: 30. April 2021) Loadable kernel module. Hentet 30.04.2021 fra: https://en.wikipedia.org/wiki/Loadable_kernel_module

Wikipedia(15.05.2021) Salt(cryptography) *wikipedia.com* hentet 15.05.2021 fra: https://en.wikipedia.org/wiki/Salt_(cryptography)