

IN2140 - kompendium

Emnekode: IN2140

Nicolai Tran og Aleksander Høiness

Innholdsfortegnelse

INNHOLDSFORTEGNELSE	I
1 NOTATER OM HVERT TEMA	1
1.1 OS: prosesser og CPU scheduling.....	1
1.1.1 Prosess	1
1.1.2 Interrupts og exceptions	5
1.1.3 Tilstander	6
1.1.4 Skedulering	6
1.2 OS: Minne	6
1.2.1 Hva er pipes, og hva er forskjellen på pipes og delt minne?.....	6
1.2.2 Overføring av data fra disk til applikasjon	7
1.2.3 Minnebuss.....	7
1.3 OS: Lagring	16
1.3.1 File systems	23
1.3.2 Multiple disks.....	23
1.4 OS: IPC	26
1.5 Datakom: Intro.....	28
1.6 Network structures	29
1.6.1 Structuring the tasks	31
1.6.2 ISO OSI(Open Systems Interconnection)	32
1.7 Datakom: ARP og IP – naming. INTERNET	34
1.7.1 Network byte order.....	36
1.8 Datakom: Addressering på L4 og L5.....	41
1.8.1 IPv6	46
1.9 Datakom: Forbindelsesorientert og forbindelsesløs kommunikasjon	50
1.10 Datakom: Flytkontroll	54
1.10.1 Transport layer.....	58
1.10.2 Hva er context-switch og hvordan gjøres det?	58
1.10.3 Flow control: Stop-and-wait:.....	58
1.10.4 Flow control: Sliding Window	59
1.11 Sliding window: Concept.....	60
1.12 Datakom: Metningskontroll	63
1.13 Datakom: Ruting	67
2 FORKLARING AV KONSEPTER OG PROTOKOLLER	73

2.1.1	Hva er virtuelt minne?	73
2.1.2	Hva er en prosess?	75
2.1.3	Hva er en tråd?	75
2.1.4	Hva er forskjellen mellom en prosess og en tråd.....	75
2.1.5	Hva er forskjellen mellom en prosess, program og kode?	75
2.1.6	Hva er paging?.....	76
2.1.7	Ulike typer minnelayout	76
2.1.7.1	Fixed partition	76
2.1.7.2	Dynamic partitioning.....	77
2.1.7.3	Buddysystem.....	78
2.1.8	Plasseringsalgoritmer	78
2.1.8.1	First fit	79
2.1.8.2	Next fit.....	79
2.1.8.3	Best fit.....	79
2.1.8.4	Buddytildeling	80
2.1.9	Hva er en nettmaske?.....	81
2.1.10	Hva er Multilevel Paging?	81
2.1.11	Hva er forskjellen på ekstern og intern fragmentering?.....	82
2.1.12	Hva er CPU skedulering?	83
2.1.13	Ulike typer skedulering	83
2.1.13.1	FIFO.....	83
2.1.13.2	Round Robin.....	84
2.1.13.3	SJF.....	84
2.1.13.4	EDF.....	84
2.1.14	Hvilken skeduleringsalgoritme er best for hva?	85
2.1.15	Hva er disken?	85
2.1.16	Disk skedulering.....	86
2.1.17	Hva er multicore?	89
2.1.18	Hva er multiprogrammering?	89
2.1.19	Hva er packet switching?.....	90
2.1.20	Hva er Circuit switching?	90
2.1.21	Hva er forskjellen på Circuit og Packet switching?	91
2.1.22	Hva er Virtuell Circuit Switching?	92
2.1.23	Hva er Message switching?.....	93
2.1.24	Hva er IPC?.....	93
2.1.25	Hva er filsystemer?	94
2.1.26	Hva er nettmasker?	94
2.1.27	Hvordan regner man ut nettmasker?	95
2.1.28	Hva er nettmaskeklasser?	96
2.1.29	Hva er forskjellen på UDP og TCP?	96
2.1.30	Hva betyr båndbredde?	97
2.1.31	Hva er forskjellen på IPv4 og IPv6, og hva brukes i dag?.....	98
2.1.32	Hvordan sendes en pakke ved hjelp av TCP?	98

2.1.33	Congestion control	99
2.1.34	Hva er trengselskontroll	99
2.1.35	FIFO vs Round Robin	99
2.1.36	Hvordan brukes heapen og stacken i minnet?	100
2.1.37	Hva er forskjellen på fragmentering, segmentering og partisjonering? 101	
2.1.38	Forskjell på disk og primærminne	102
2.1.39	Forklar Second Chance algoritmen	103
2.1.40	Hva er en kontekstsvitsj og hva brukes det til?	104
2.1.41	Hva er big og small endian og hvordan brukes det?	104
2.1.42	Hva er flytkontroll?	106
2.1.43	Hva er metningskontroll?	106
2.1.44	Hva er DVR (Distance Vector Routing)	106
2.1.45	Hva er DNS (Domain Name System)?	106
2.1.46	Hvordan fungerer Flytkontroll selective repeat	107
2.1.47	Nettmasker, hvorfor, hvordan regne ut	107
2.1.48	Forskjell på preemptiv og ikke-preemptiv skedulering	108
2.1.49	Hva er et filesystem og hva håndterer det?	108
2.1.50	CIDR (Classless InterDomain Routing)	108
2.1.51	Hva er forskjellen på OS-ene Windows, Unix og Linux?	109
3	FORSTÅELSESSPØRSMÅL.....	110
3.1	Lav- og høynivå språk	110
	Språket C for operativsystem	113
3.2	System-, bibliotekskall og skedulering	114
3.2.1	Shortest job first	117
3.2.2	Diskutnyttelse	119
3.2.3	Mål for diskutnyttelse	119
3.3	Heap og stack, partisjonering	120
3.3.1	Oppgave om partisjonering	121
3.3.2	Oppgave om paging	123
3.4	Disk	124
3.4.1	Oppgave om sekundærminne	124
3.4.2	Filsystem	125
3.4.3	EXT4	126
3.5	Signaler, avbrudd	127
3.6	Lagdelingsmodell/ TCP/IP	132
3.6.1	TCP/IP Egenskaper	134
3.7	Protokoller	136
3.8	DNS, Adresseblokk	139
3.8.1	Domenenavn	140
3.8.2	DNS eksmplar og oppgaver	141
3.9	TCP og UDP	143
3.9.1	Sammenlikn UDP og TCP	143
3.9.2	Linjesvitsjing og pakkesvitsjing	144
3.9.3	IPv6	145

3.10	FLYTKONTROLL	146
3.10.1	Stop and Wait	146
3.10.2	Go-Back-N	147
3.10.3	Kredittmekanisme	148
3.10.4	Masse om flytkontroll	148
3.10.5	TCP og piggybacking, ACK	150
3.11	Metningskontroll	150
3.11.1	TCP Renos faser	151
3.11.2	Detektere pakketap	151
3.11.3	Persistent metning	153
3.12	Routing grunnleggende	154
3.12.1	Link State Routing	157
4	OPPSUMMERING AV HVERT KAPITTEL I BOKA	158
4.1	2.1 Processes	158
4.2	2.2 Threads	159
4.3	2.4 Scheduling	160
4.4	<i>3.3 Virtual Memory</i>	161
4.5	<i>3.7 Segmentation</i>	161
4.6	4.1 Files - Modern Operating systems	162
4.7	4.2 Directories	163
4.8	5.1 Principles of I/O Hardware	164
4.9	5.2 Principles of I/O Software	164
4.10	5.3 I/O Software Layers	165
4.11	5.4 Disks	166
4.12	5.6 UI: Keyboard, Mouse, Monitor	167
4.13	11.3 System Structure	167
4.14	11.5 Memory management	168
4.15	4.1 The Channel Allocation Problem	169
4.16	4.8 Data Link Layer Switching	170
4.17	5.1 Network Layer Design Issues	171
4.18	5.6 The network Layer in the Internet	172
5	TIDLIGERE EKSAMENSOPPGAVER	173
5.1	Funksjonskall vs Systemkall	173
5.2	Tråder vs prosesser	174
5.3	Flytkontroll	177
5.4	Virtuelt minne – m utregning	180
6	HVORDAN REGNE UT, MED EKSEMPLER	182
6.1	Virtuelt minne – 1-nivås minneoppslag	182
6.2	Utregning av antall sider som kan adresseres i lang- vs beskyttet-modus	182
6.3	Velge riktig filsystem	184
6.4	Virtuelt minne – Page replacement	185
6.5	CIDR – Classless InterDomainRouting	186
6.6	Vanlig til binærtall	187

6.6.1	Regne antall sideoppføringer (entries) og størrelse på oppføringene	
	187	
6.1	Regne ut nettmaske.....	188
6.2	Regne ut pagingnivåer.....	190

1 Notater om hvert tema

Teksten på denne siden er kun ment for å illustrere hvordan riktig formaterte overskrifter og brødtekst ser ut.

Når du skal begynne å skrive bør du derfor slette all tekst på denne siden, og bruke stiler – Heading (Overskrift) 1 til 5 for overskrifter og normal for brødtekst – for å formatere din egen tekst.

For å oppdatere innholdsfortegnelsen, må du høyreklikke på den og velge «Update field» («Oppdater felt») i menyen. Hvis det dukker opp et vindu med alternativer, må du velge «Update entire table» («Oppdater hele tabellen») og klikke ok.

Innholdsfortegnelsen virker kun som forutsatt (halvautomatisk) hvis du har formatert overskriftene dine med overskrifts-stiler.

1.1 OS: prosesser og CPU scheduling

1.1.1 Prosess

Utførelsen av instruksjoner

- Vi har et program med instruksjoner, og utførelsen av disse instruksjonene kalles prosessen

Fork

En prosess kan opprette en annen prosess ved hjelp av pid_t fork(void) system call (see man 2 fork)

- makes a duplicate of the calling process including a copy of the virtual address space, open file descriptors, etc... (only PIDs are different – locks and signals are not inherited)
- Vil ha forskjellige minnerom, men innholdet i minnet vil sannsynligvis være det samme

returnerer

- ... if parent: child process' PID when successful, -1 otherwise
- ... if child: 0 (if successful - if not, there will not be a child)

Execution

For å gjøre et prosess execute til et program , kan man bruke

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

- Laster inn/kjører et program som erstatter programmet som prosessen i utgangspunktet kjører.

returnerer:

- no return value on success, actually no process to return to
- -1 is returned on failure (and errno set)

Waiting

For å få en prosess til å vente på en annen prosess, kan man bruke

pid_t wait(int *status) system call (see man 2 wait):

- Ofte brukt sammen med fork
- Venter ofte på at en av barne-prosessene terminerer

returnerer:

- -1 if no child processes exist
- PID of the terminated child process and puts the status of the process in status

Kill

Sender et signal til...

En prosess kan terminere hvis:

- Ingen instruksjoner igjen
- En funksjon i et program avslutter med et return -parameter til å returnere status-verdien
- int kill sender et signal til en prosess for å terminere den
- void exit terminerer en prosess og returnerer status-symbolet

Vanligvis indikerer statusverdi 0 suksess, og andre indikerer feil.

Exit

Avslutter prosess

Context switch

Prosessen å endre en kjørende prosess til en annen

1. stop running process 1

2. store the state (like registers, instruction pointer) of process 1 (usually on stack or PCB)
3. restore state of process 2
4. resume operation on program counter for process 2

Prosess vs tråder

- Processes: resource grouping and execution
- Threads (light-weight processes)
 - enable more efficient cooperation among execution units
 - share many of the process resources (most notably address space)
 - have their own state, stack, processor registers and program counter

Scheduling

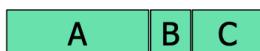
Man bør ha noe som bestemmer rekkefølgen og prioriteringene på prosessene i programmet-

FIFO and Shortest Job First

FIFO and Shortest Job First (SJF)

- FIFO: First in, First Out
- SJF: Select first tasks with *shortest processing requirement (completion time)*
- Example: Arrival order:processing requirement - A:8, B:2, C:4

- FIFO:



- Average wait time: 6
- Average finishing time: 10,67
- simple
- fair?
- long waiting and finishing times

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

- SJF:



- Average wait time: 2,67
- Average finish time: 7,33
- simple
- better average times compared to FIFO
- hard to determine processing requirement
- potentially huge finishing times and starvation (new shorter jobs arrive)

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

FIFO and Round Robin

- Example: 10 jobs and each takes 100 seconds, assuming no overhead (see diagram)
 - FIFO – the process runs until finished
 - start: job1: 0s, job2: 100s, ... , job10: 900s → **average** 450s
 - finished: job1: 100s, job2: 200s, ... , job10: 1000s → **average** 550s
 - some get long waiting time, but some are lucky
 - RR – time slice of 1s
 - start: job1: 0s, job2: 1s, ... , job10: 9s → **average** 4.5s
 - finished: job1: 991s, job2: 992s, ... , job10: 1000s → **average** 995.5s
 - fair, but no one is lucky
- Comparisons
 - FIFO better for long CPU-intensive jobs (there **is** overhead in switching!!)
 - but RR much better for interactivity!
- **But, how to choose the right time slice??**

Scheduling algorithm classification:

- dynamic
 - makes scheduling decisions at run-time
 - flexible to adapt
 - considers only the actual task requests and execution time parameters
 - large run-time overhead finding a schedule
- static
 - makes scheduling decisions off-line (also called pre-run-time)
 - generates a dispatching table for the run-time dispatcher at compile time
 - needs complete knowledge of the task before compiling
 - small run-time overhead
- preemptive
 - running tasks may be interrupted (preempted) by higher priority processes
 - preempted process continues later at the same state
 - overhead of contexts switching
- non-preemptive
 - running tasks will be allowed to finish its time-slot (higher priority processes must wait)
 - reasonable for short tasks like sending a packet (used by disk and network cards)
 - less frequent switches

Preemption

- Tasks wait for processing

- Scheduler assigns priorities
- Task with highest priority will be scheduled first
- Preempt current execution
 - A higher priority task arrives
 - Timeslice is consumed
- Real-time and best effort priorities
 - Real-time processes have higher priority
- To kinds of preemption:
 - Preemption points
 - Predictable overhead
 - Immediate preemption
 - Needed for hard real-time systems
 - Needs special timers and fast interrupt and context switch handling

1.1.2 Interrupts og exceptions

Interrupts

Electronical signals that usually result in a forced transfer of control to an interrupt handling routine.

- Hardware
- Software
- Asynkrone

Exceptions

- Prosessor oppdager en feil/unntakstilstand
- Traps
 - utløses av bestemte hendelser, som en divide-by-zero-feil eller en ulovlig instruksjon. De brukes ofte til feilsøking eller håndtering av feil.
 - Prosessoren treffer en condition som exception handler kan håndtere
- Faults
 - Faults ligner på traps, men de skyldes hardwarefeil eller andre ytre faktorer. For eksempel kan en minnefeil oppstå når et program prøver å få tilgang til en ugyldig minneadresse
 - Prosessoren treffer en feil som exception handler kan fikse(correct)
- Aborts
 - Aborts er unntak som utløses når et program støter på en kritisk feil som ikke kan gjenopprettes fra. Dette kan skje på grunn av et ubehandlet unntak eller en alvorlig systemfeil. En avbrytelse avslutter programmet med det samme, uten videre behandling.
 - Terminerer prosessen pba en uopprettelig feil som prosessen ikke selv kan fikse.

1.1.3 Tilstander

- Kjørende
- Klar
- Terminert
- Blokkert

1.1.4 Skedulering

Policyer

SCHED_OTHER

- Normal policy
- Default på Linux
- CFS

SCHED_FIFO - FIFO

- Sanntidspolicy
- Statisk prioritet for tråder (0-99)

SCHED_RR - Round Robin

- Tilsvarer SCHED_FIFO, men med timeslices

1.2 OS: Minne

1.2.1 Hva er pipes, og hva er forskjellen på pipes og delt minne?

"Pipes" er et konsept i operativsystemer som tillater to prosesser å kommunisere med hverandre ved å sende data gjennom en "pipe". I en pipe, skriver en prosess data til pipen, som deretter kan leses av en annen prosess. Det er to typer av pipes: navnløse pipes og navngitte pipes.

Navnløse pipes er vanligvis brukt for kommunikasjon mellom en foreldreprosess og en barnprosess, mens navngitte pipes kan brukes for kommunikasjon mellom to vilkårlige prosesser på samme system. Pipes gir en enveis kommunikasjon: data flyter fra skriveenden til leseenden av pipen.

Delt minne er en annen form for interprosesskommunikasjon. Med delt minne, har to eller flere prosesser tilgang til samme minneseksjon, og de kan alle lese fra og skrive til det minnet. Delt minne gir en mye raskere metode for kommunikasjon siden data ikke trenger å kopieres mellom prosesser, men det krever mer kompleks synkronisering fordi flere prosesser har tilgang til det samme minnet på samme tid. Her er hovedforskjellene mellom pipes og delt minne:

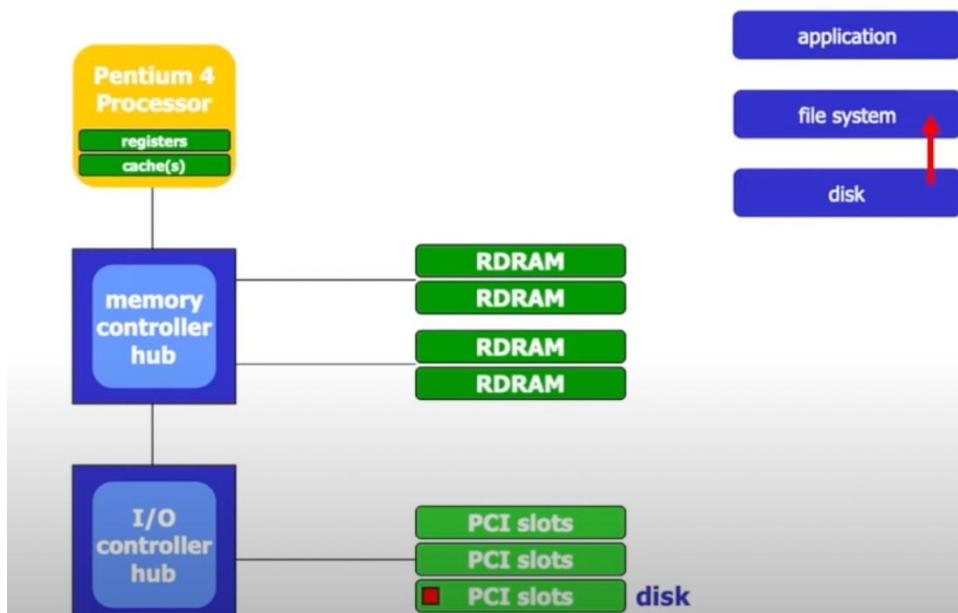
- Kommunikasjonsretning:** Pipes gir enveis kommunikasjon (selv om toveis kommunikasjon kan oppnås ved å bruke to pipes), mens delt minne tillater toveis kommunikasjon.
- Dataflyt:** I en pipe, blir data skrevet til en ende og lest fra den andre enden. Med delt minne, kan alle prosesser lese og skrive til alle deler av minnet.
- Ytelse:** Delt minne kan gi høyere ytelse siden det ikke er nødvendig å kopiere data mellom prosesser, men det krever mer kompleks synkronisering.
- Bruksområder:** Pipes er ofte enklere å bruke for enkle oppgaver og for kommunikasjon mellom beslektede prosesser, mens delt minne kan være mer passende for mer komplekse oppgaver og for kommunikasjon mellom vilkårlige prosesser.

1.2.2 Overføring av data fra disk til applikasjon

Intel modell Pentium

Gjøre kall i applikasjonen og ned i minnet for å hente dataen.

- Først overføre dataen til primærminne slik at programmet kan lese det.
- Overføre gjennom alle kontrollene, over PCI-bussen, gjennom IO controller hub, gjennom memory controller hub over ram interface til et sted på ram brikker.
- Data ligger nå i filsystemet, men pga. Rettigheter osv. Så kan ikke appen ha tilgang til det enda, må kopiere dataen til et minneområdet som appen har tilgang til.
- Kopiere dataen over ram interface gjennom memory controller hub, over systembussen, gjennom CPU og tilbake etterpå. Til et sted i minnet.



1.2.3 Minnebuss

Minnebussen er en type datavei som brukes for å koble forskjellige deler av datamaskinen til hovedminnet (RAM). Den fungerer som en informasjonsmotorvei

som gjør det mulig for data å bevege seg frem og tilbake mellom prosessoren (CPU), hovedminnet og andre komponenter som grafikkort.

Når en CPU trenger å lese eller skrive data til minnet, sender den en forespørsel over minnebussen. Denne forespørselen inneholder informasjon om hva den vil gjøre (lese eller skrive), samt adressen til dataene den er interessert i.

Når RAM mottar forespørselen, vil den enten returnere de forespurte dataene (i tilfelle av en leseforespørsel) eller oppdatere de relevante dataene (i tilfelle av en skriveforespørsel). Denne informasjonen sendes da tilbake til CPUen langs minnebussen.

Det er viktig å merke seg at minnebussens hastighet og bredde (antall bits det kan overføre samtidig) kan ha stor innvirkning på systemets generelle ytelse. En raskere, bredere minnebuss kan overføre mer data på kortere tid, noe som hjelper med å minimere forsinkelsen mellom når CPUen ber om data og når den mottar dem.

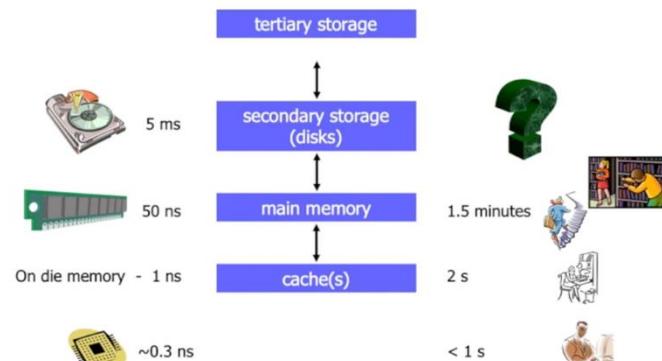
Memory management

- Allokere plass til prosesser
- Beskytte minne-regioner
- Tilby et virtuelt syn på minnet, med mulighet til å bruke mer minne enn det som finnes

Minnehierarki

En prosess trenger typisk mye minne og vi kan ikke aksessere disken hver gang vi trenger data

- Tertiært minne
- Sekundærminne
 - 5 ms
- Hovedminne
 - 50 ns
- Cache
 - 1 ns

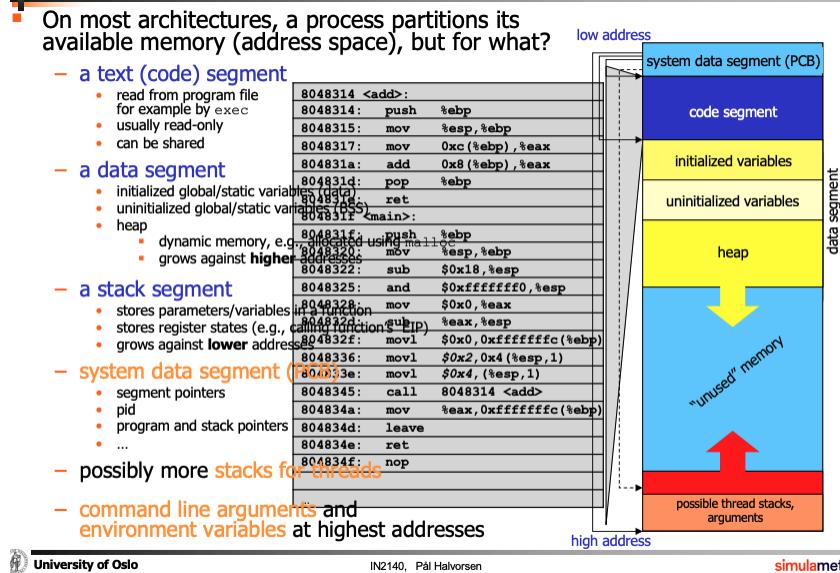


Har forskjellig minnestørrelse, hastighet, og

Absolutt og relativ adressering

- Hardware bruker ofte *absolutt* adressering
- Reservert minneregioner

- leser data via referanser til byte-nummer
- Software
 - Leser absolutt byte
 - Resultat avhenger av en prosess' fysiske lokasjon
 - Absolutt adressering er ikke lurt
 - Bruker relativ adressering
 - Uavhengig av prosess-lokasjon i minnet
 - Adressen er uttrykket relativt til en base-lokasjon



University of Oslo IN2140, Pål Halvorsen simulmet

Memory management for Multiprogramming

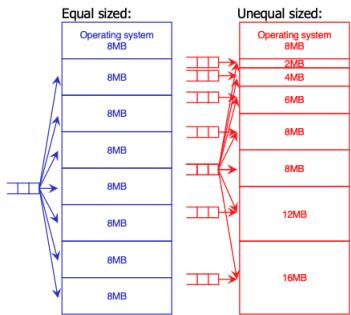
- Use of secondary storage
 - Keeping all programs and their data in memory may be impossible
 - Move parts of a process from memory
- Swapping: remove a process from memory
 - With all its state and data
 - Store it on a secondary medium
- Overlays: manually replace parts of code/data
 - Programmer's rather than OS's work
 - Only for very old and memory-scarce systems
- Segmentation/paging: remove parts of a process from memory
 - Store it on a secondary medium
 - Size of such parts are usually fixed

Partitioning

Fixed

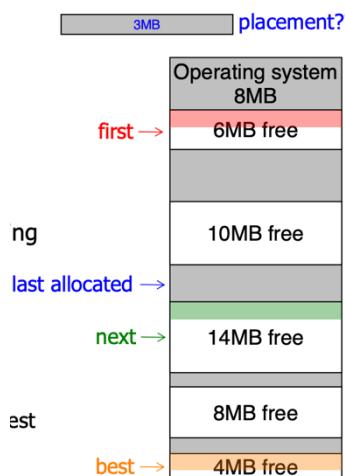
- Deler minnet opp i statiske partisjoner når systemet starter
- Enten:
 - equal sized eller

- Store programmer kan ikke bli eksekvert
- Små programmer bruker ikke hele parisjonen
- Unequal sized
 - store programmer kan bli hentet inn samtidig
 - Mindre intern fragmentasjon
 - men hva hvis det bare er store eller små prosesser?



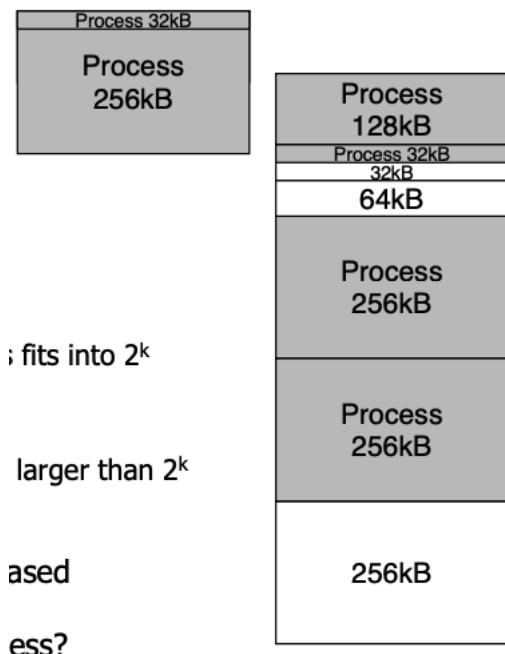
Dynamic

- Deler opp minnet i run-time
 - partisjoner blir laget dynamisk
 - Blir fjernet når jobber er gjort
- Ekstern fragmentering øker i tråd med systemets running time.
- Compaction fjerner fragmenter ved å flytte data inn i minnet
 - tar tid
 - tar opp prosesseringsressurser
- first fit
 - Enklest, raskest, ofte best
- next fit
 - nesten likt
- best fit
 - tregest, mange små fragmenter, og ofte derfor værst.



Buddy system

- Blanding av de to over
- partitions have sizes 2^k , $L \leq k \leq U$
- Holder en list med hull og deres størrelse
- Tildeler minnet til en prosess:
 - Finner minste k slik at prosessen får plass inn i 2^k
 - Finn et hull med størrelse 2^k
 - Hvis det ikke er tilgjengelig, split det minste hullet som er større enn 2^k rekursivt inn i halve
- Merger partisjoer hvis det er mulig når det frigjøres



Segmentering

Krever at en prosess plasseres i sammenhengende minne fører til mye mer fragmentering

- Segmentering
 - Ulike lengder
 - Bestemt av programmerer
 - Minnerammer
- Pros
 - prinsipp som i dynamisk partisjonering - kan ha ulike størrelser
 - ingen intern fragmentering
 - mindre ekstern fragmentering fordi segmentene i gjennomsnitt er mindre
- Cons
 - legger til et trinn i adresseomforming

Adress Lookup

- Finn segment-tabell adresse i registeret
- Ta ut segment nummer fra adressen
- Finn segment adressen ved å bruke segment-nummer som indeks til segment-tabellen

- Finn absolutt adress innad i segment ved å bruke relativ adresse

Paging

- Equal lengths determined by processor
- One page moved into one page frame
- Process is loaded into several frames (not necessarily consecutive)

Fragmentation

- No external fragmentation
- Little internal fragmentation

Can combine segmentation and paging.

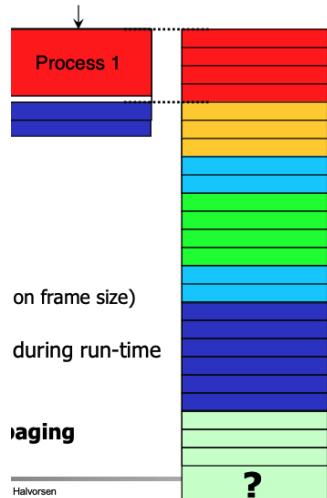
Virtuelt minne

Virtuelt minne er en nøkkelfunksjon i moderne operativsystemer, og det brukes for flere grunner:

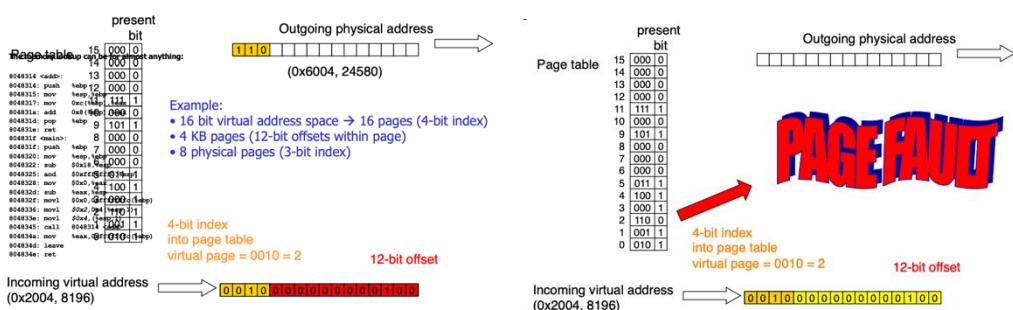
1. Prosessisolering: Virtuelt minne gir hver prosess sin egen adresseplass, som isolerer hver prosess fra hverandre. Dette forhindrer en prosess i å få direkte tilgang til eller endre minnet til en annen prosess.
2. Effektiv minneutnyttelse: Ved å tillate minnesegmentering og sideutbytte, kan operativsystemer utnytte minnet mer effektivt. Når en prosess ikke er aktiv, kan dens minnesider flyttes til disk (dette kalles "paging" eller "swapping"), frigjøre fysisk minne for andre prosesser.
3. Programstørrelse: Med virtuelt minne kan programmer være større enn den fysiske minnestørrelsen på maskinen. Dette er fordi bare deler av programmet ("sider" eller "segmenter") må være i minnet på et gitt tidspunkt.
4. Lett å dele filer mellom prosesser: Med virtuelt minne er det mulig å mappe en fil inn i en prosess' adresseområde, noe som betyr at flere prosesser kan dele samme minneutsnitt for en fil.

De beskrevne partisjonsordningene kan brukes i applikasjoner, men et moderne operativsystem bruker også virtuelt minne:

- et tidlig forsøk på å gi en programmer mer minne enn fysisk tilgjengelig
 - eldre datamaskiner hadde relativt lite hovedminne
 - men selv i dag trenger ikke alle instruksjoner å være i minnet før utførelsen starter



- del programmet inn i mindre uavhengige deler
- last inn aktive deler
- når et program er ferdig med en del, kan en ny del lastes inn
- minnet er delt inn i like store rammer, ofte kalt sider
- noen sider befinner seg i fysisk minne, mens andre er lagret på disk og hentes ved behov
- virtuelle adresser blir oversatt til fysiske adresser (i MMU) ved hjelp av en sidetabell
- både Linux og Windows implementerer en flat lineær 32-bit (4 GB) minnemodell på IA-32
 - Windows: 2 GB (høye adresser) kjerne, 2 GB (lave adresser) brukermodus-tråder
 - Linux: 1 GB (høye adresser) kjerne, 3 GB (lave adresser) brukermodus-tråder



Page fault handling

1. Hardware traps to the kernel saving program counter and process state information
2. Save general registers and other volatile information
3. OS discovers the page fault and determines which virtual page is requested
4. OS checks if the virtual page is valid and if protection is consistent with access
5. Select a page to be replaced
6. Check if selected page frame is "dirty", i.e., updated. If so, write back to disk, otherwise, just overwrite
7. When selected page frame is ready, the OS finds the disk address where the needed data is located and schedules a disk operation to bring in into memory
8. A disk interrupt is executed indicating that the disk I/O operation is finished, the page tables are updated, and the page frame is marked "normal state"
9. Faulting instruction is backed up and the program counter is reset
10. Faulting process is scheduled, and OS returns to the routine that made the trap to the kernel
11. The registers and other volatile information are restored, and control is returned to user space to continue execution as no page fault had occurred

Page replacement algorithms

- Page fault → OS has to select a page for replacement
- How do we decide which page to replace?
 - Determined by the page replacement algorithm
 - several algorithms exists:
 - Random
 - Other algorithms take into account usage, age, etc.(FIFO, not recently used, least recently used, second chance, clock)
- FIFO
 - all pages in memory are maintained in a list sorted by age
 - FIFO replaces the oldest page, i.e, the first in the list
 - Low overhead
 - Non-optimal replacement (and disk accesses are expensive)
 - FIFO is rarely used in its pure form
- Second chance
 - Modification of FIFO
 - R bit: when a page is referenced again, the R bit is set, and the page will be treated as a newly loaded page
 - Second chance is a reasonable algorithm, but inefficient because it is moving pages around the list
- Clock
 - More efficient implementation Second Chance
 - Circular list in form of a clock
 - Pointer to the oldest page:
 - R-bit = 0 → replace and advance pointer
 - R-bit = 1 → set R-bit to 0, advance pointer until R-bit = 0, replace and advance pointer
- Least Recently Used (LRU)
 - Replace the page that has the longest time since last reference
 - Based on the observation that
 - Pages that was heavily used in the last few instructions will probably be used again in the next few instructions
 - Several ways to implement
- Round robin
 - In the Round Robin algorithm, all pages in memory are organized in a circular list. Each page is assigned a fixed time slice or quantum, which represents the maximum amount of time it can stay in the main memory before being replaced. When a page fault occurs, the operating system selects the page that has been in memory the longest, according to the current position of the circular list.
 - The advantage of Round Robin is its low overhead, as it only requires a simple circular list to track the pages. However, it is considered a non-optimal replacement algorithm because it doesn't take into account

the frequency of page usage or other factors. Disk accesses, which are required to fetch replacement pages from secondary storage, can also be expensive in terms of performance.

- It's important to note that Round Robin is primarily used as a CPU scheduling algorithm in the context of process scheduling, rather than a dedicated page replacement algorithm. In the realm of virtual memory, other algorithms such as FIFO (First-In-First-Out), LRU (Least Recently Used), and Second Chance are more commonly employed due to their better overall performance.

Interval caching (IC) is a caching strategy for streaming servers

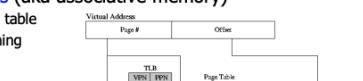
- caches data between requests for same video stream - based on playout intervals between requests
- following requests are thus served from the cache filled by preceding stream
- sort intervals on length, buffer requirement is data size of interval
- to maximize cache hit ratio (minimize disk accesses) the shortest intervals are cached first

Speeding up paging...

- Every memory reference needs a virtual-to-physical mapping
- Each process has its own virtual address space (an own page table)
- Large tables:
 - 32-bit addresses, 4 KB pages → 1.048.576 entries
 - 64-bit addresses, 4 KB pages → 4.503.599.627.370.496 entries

→ Translation lookaside buffers (aka associative memory)

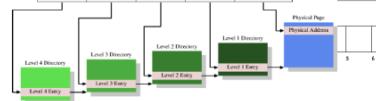
- hardware "cache" for the page table
- a fixed number of slots containing the last page table entries



→ Page size:

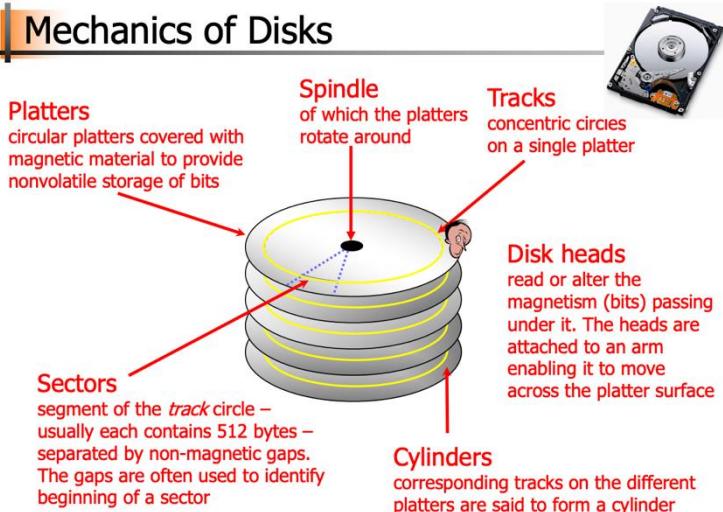
larger page sizes reduce number of p_t

→ Multi-level page tables



1.3 OS: Lagring

Mechanics of Disks



Disks

- are used to have persistent system
- are cheaper compared to main memory
- have more capacity
- are orders orders of magnitude

Disk capacity

- The size (storage space) of the disk is dependent on
 - the number of platters
 - whether the platters use one or both sides
 - number of tracks per surface
 - (average) number of sectors per track

Disk Access Time

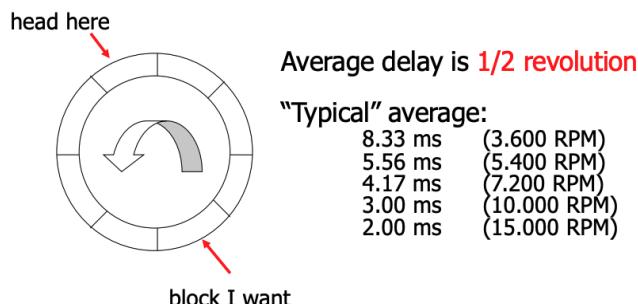
- How do we retrieve data from disk?
 - position head over the cylinder (track) on which the block (consisting of one or more sectors) is located
 - read (or write) the data block as the sectors are moved under the head when the platters rotate
- The time between the moment issuing a disk request and the time the block is resident in memory is called disk latency or disk access time
- Calculation
 - Seek time
 - Seek time is the time to position the head
 - time is used for actually moving the head - roughly proportional to the number of cylinders traveled
 - time to start and stop moving the head

- time to move the head

$$\alpha + \beta\sqrt{n}$$

└── number of tracks
 └── seek time constant
 └── fixed overhead

- Rotational delay
 - Time for the disk platters to rotate so the first of the required sectors are under the disk head



- Transfer time
 - Time for data to be read by the disk head, i.e., time it takes the sectors of the requested block to rotate under the head
 - Transfer time is dependent on data density and rotation speed
 - Transfer time = (Amount Of Data To Read * TimePerRotation)/amount Of DataPerTrack
- other delays
 - CPU time to issue and process I/O
 - Contention for controller, bus, memory

Writing and modifying blocks

- A write operation is analogous to read operations
 - must potentially add time for block allocation
 - a complication occurs if the write operations has to be verified - must usually wait another rotation and the r
 - Total write time = readtime(+ time for one rotation)
- A modification operation is similar to read and write operations
 - cannot modify a block directly
 - read block into main memory
 - modify the block
 - write new content back to disk

Disk controller

- To manage the different parts of the disk, we use a disk controller, which is a small processor capable of:
 - moving the head to the desired track
 - transferring data between main memory and disk

Second storage usage

- must take into account the use of secondary storage
 - Large gaps in access times between disks and memory, i.e., a disk access will probably dominate the total execution time
 - Huge performance improvements if we reduce the number of disk accesses
 - a “slow” algorithm with few disk access will probably outperform a “fast” algorithm with many disk accesses

Disk Scheduling

Seek time is the dominant factor of the total disk I/O time

disk scheduling ≠ CPU scheduling

- A mechanical device - hard to determine (accurate) access times
- disk accesses can/should not be preempted - run until they finish

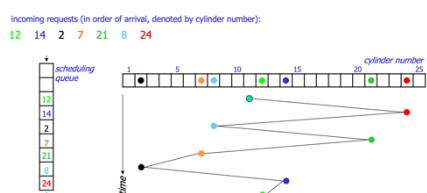
General goals:

- short response time
- high overall throughput
- fairness

Algorithms

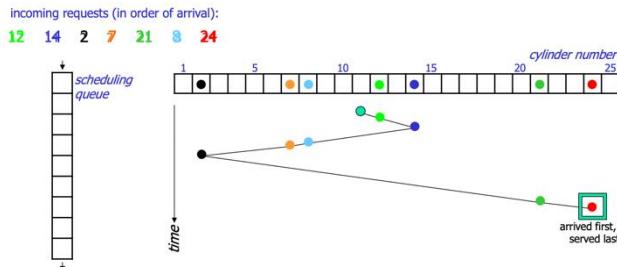
First-Come-First-Serve (FCFS)

- Serves the first arriving request first
 - long seeks
 - “Short” response time for all



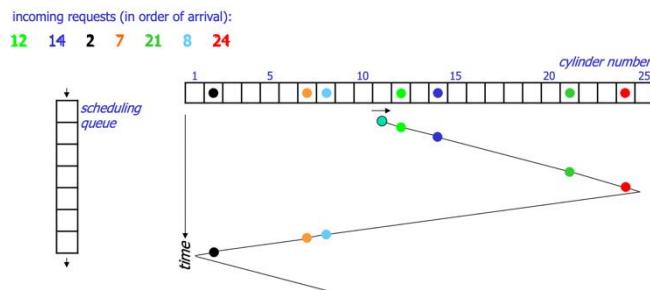
Shortest Seek Time First (SSTF)

- SSTF serves closest request first:
 - short seek times
 - longer maximum response times - **may even lead to starvation**



SCAN

- moves head edge to edge and serves requests on the way:
 - bi-directional
 - compromise between response time and seek time optimizations

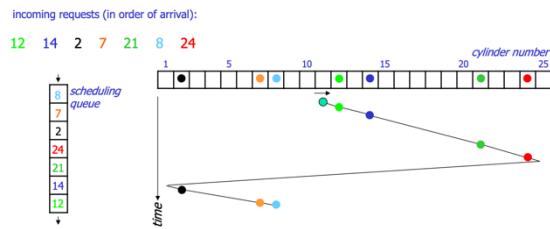


SCAN vs. FCFS

- Disk scheduling makes a difference
- In this case we see that SCAN requires much less head movement compared to FCFS
 - 37 vs 75
 - Imagine having a lot more data

C-SCAN

- Circular-SCAN moves head from edge to edge
 - optimization of SCAN
 - serves requests on one way - uni-directional
 - do not serve on the way back
 - improves response time (fairness)



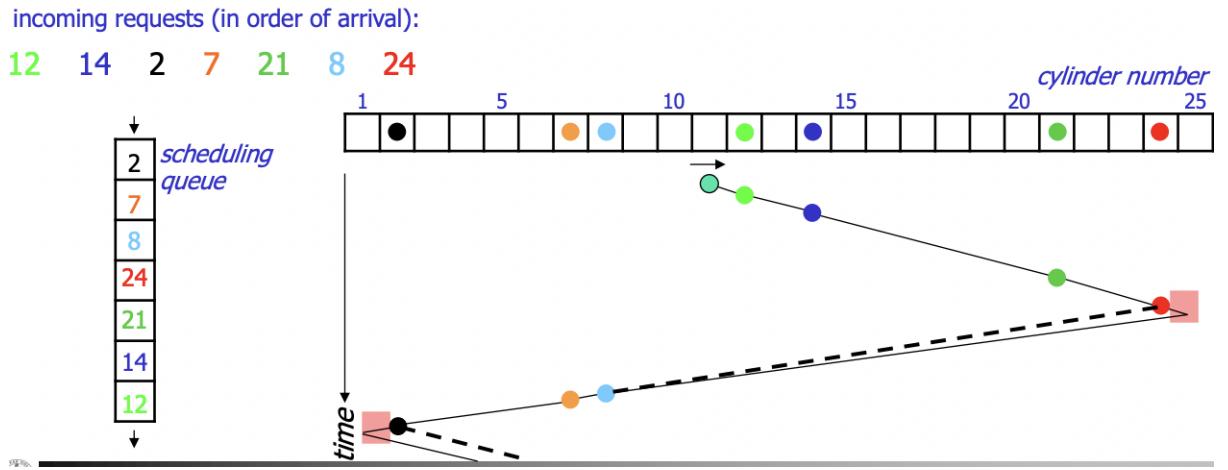
SCAN vs- C-SCAN

- Why is C-SCAN in average better in reality than SCAN when they both service the same number of requests in two passes
 - modern disks must accelerate (speed up and down) when seeking
 - head movement formula

SCAN	C-SCAN
bi-directional	uni-directional
 requests: n avg. dist: $2x$ (spread over both directions) total cost: $n \times \sqrt{2x} = (n \times \sqrt{2}) \times \sqrt{x}$	 requests: n avg. dist: x (over one direction only + one full pass) total cost: $\sqrt{n \times x} + n \times \sqrt{x} = (\sqrt{n} + n) \times \sqrt{x}$
$n \times \sqrt{2} > \sqrt{n} + n$	
if n is large: $n \times \sqrt{2} > \sqrt{n} + n$	

LOOK and C-LOOK

- LOOK is a variation of SCAN (C-SCAN)
 - same schedule as ACAN
 - does not run to the edges
 - stops and returns at outer- and innermost requests
 - increased efficiency
 - SCAN vs. LOOK example:



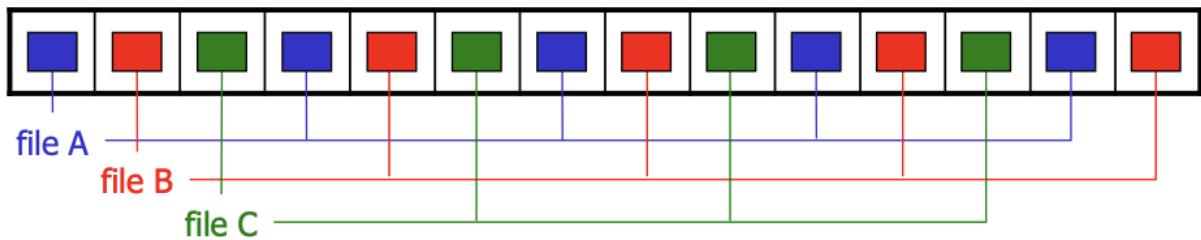
Modern Disk Scheduling:

- Disk used to be simple devices and disk scheduling used to be performed by OS only
- But new disks are more complex
 - Hide their true layout
 - Transparently move blocks to spare cylinders
 - have different zones
 - head accelerates - most algorithms assume linear movement overhead
 - on device buffer caches may use read-ahead prefetching
 - “smart” with build-in low-level scheduler (usually SCAN-derivative)
 - We cannot fully control the device
- Linux
 - NOOP
 - FCFS with request merging
 - Deadline I/O
 - C-SCAN based
 - 3 queues: 1 sorted (elevator) queue, and 2 deadline queues (one for read and one for write)
 - Anticipatory
 - same queues as in Deadline I/O
 - delays decisions to be able to merge more requests
 - Completely Fair Queuing (CFQ)
 - 1 queue per process (periodic access, but period length depends on load)
 - gives time slices and ordering according to priority level (real-time, best-effort, idle)
 - selects requests from queues in RR for the final elevator sorting
 - work-conserving
- Windows
 - I/O priorities
 - Individual I/O operations

- to avoid low-priority starvation, a timer enforce ONE I/O per time unit (.5 sec)
- reservations
- many special functions

Data placement

- on Disk
 - Interleaved placement tries to store blocks from a file with a fixed number of other blocks in-between each block

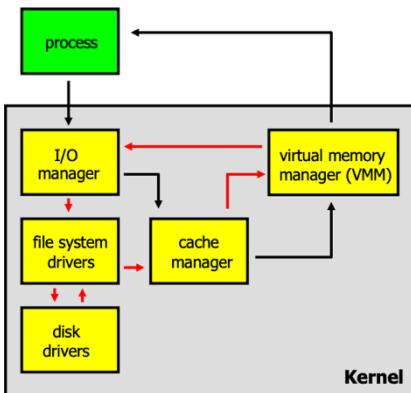


- minimal disk arm movement reading the files A, B and C (starting at the same time)
 - Good for predictable workloads reading multiple files
 - no gain if we have unpredictable disk accesses
- Non-interleaved placement can be used for highly unpredictable workloads
- Contiguous placement stores disk blocks contiguously on disk
 - minimal disk arm movement reading the whole file
 - pros/cons
 - within a file, head must not move between reads - no seeks/rotational delays
 - can approach theoretical transfer rate
 - but usually we read other files as well (giving possible large inter-file seeks)
 - real advantage
 - whatever amount to read, at most track-to-track seeks are performed within one request
- Data placement on disk
 - Organ-pipe placement consider the ‘average’ disk head position
 - place most popular data head is most often
 - center of disk is in ‘average’ closest to the head
 - but, a bit outward for zoned disks

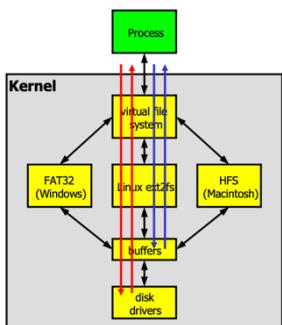
Memory Caching

- Buffer Caching Windows
 - An I/O manager performs caching

- centralized facility to all components
 - I/O request from processing



- Buffer Caching Linux
 - A file system performs caching
 - caches disk data(blocks) only
 - may hint on caching decisions
 - prefetching

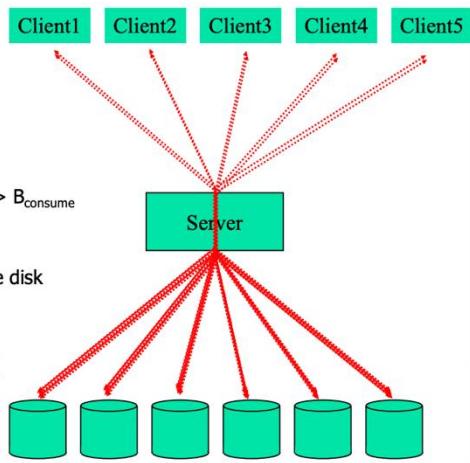


1.3.1 File systems

- A file is a collection of data - often for a specific purpose
 - unstructured files, e.g., Unix and Windows
 - structured files, e.g., early MacOS
- File systems organize data in files and manage access regardless of device type:
 - Storage management (bottom-up view) - allocating space for files on secondary storage
 - file management (top-down view) - mechanisms for files to be stored, referenced, shared, secured, ...
 - file integrity mechanisms - ensuring that information is not corrupted, intended content only
 - access methods - provide methods to access stored data

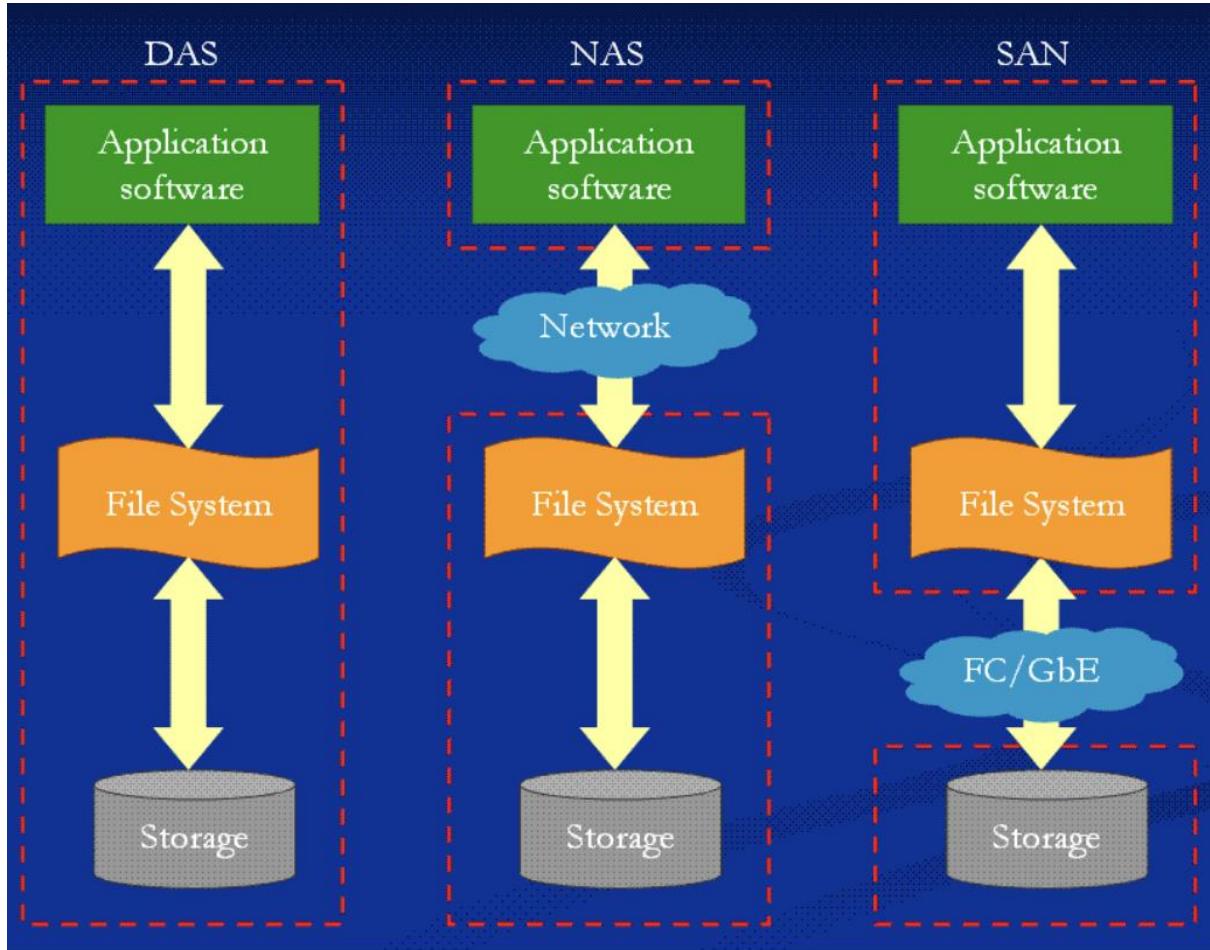
1.3.2 Multiple disks

- Striping
 - A reason to use multiple disks is when one disk cannot deliver requested data rate
 - In such a scenario, one might use several disks for striping:
 - bandwidth disk: B_{disk}
 - required bandwidth: $B_{consume}$
 - $B_{disk} < B_{consume}$
 - read from n disks in parallel: $n B_{disk} > B_{consume}$
 - Advantages
 - higher transfer rate compared to one disk



- Interleaving (compound striping)
 - Divide traditional striping group into sub-groups, e.g., staggered striping
 - Advantages
 - multiple clients can still be served in parallel
 - more efficient disk operations
 - potentially shorter response time
 - Potential drawback/challenge
 - load balancing (all clients access same group)

DAS vs. NAS vs SAN



Summary

- Disks are the main persistent secondary storage device
- The main bottleneck is often disk I/O performance due to disk mechanics: seek time and rotational delays
- Much work has been performed to optimize disk performance
 - scheduling algorithms try to minimize seek overhead
 - memory caching can save disk I/Os
 - additionally, many other ways
 - world today more complicated
 - new disks are “smart”, cannot fully control the device
- File systems provide
 - file management - store, share, access, ...
 - storage management - of physical storage
 - access methods - functions to read, write, seek

1.4 OS: IPC

Message Passing

- What is message-passing for?
 - communication across address spaces and protection domains
- What we need (generic API):
 - send(dest, &msg)
 - recv(src, &msg)
- What should “dest” and “src” be?
 - pid
 - file, e.g., a pipe
 - port: network address, etc
 - no dest: send to all
 - no src: receive any message
- What should “msg” be? We need..
 - buffer to store the message
 - size? ... for variable sized messages
 - type? ... to distinguish between messages

Direct Communication

- Must explicitly name the sender/receiver (“dest” and “src”) processes
- Requires buffers...
 - ... at the receiver
 - more than one process may send messages to the receiver
 - to receive from a specific sender, it requires searching through the whole buffer
 - ... at each sender
 - a sender may send messages to multiple receivers

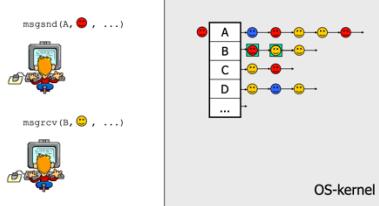
Indirect Communication

- “dest” and “src” are shared (unique) queues
 - a shared queue also allows many-to-many communication
- Where should the buffer be?
 - a buffer should be at the mailbox

Mailboxes

- Mailboxes are implemented as message queues sorting messages according to FIFO
 - messages are stored as a sequence of bytes

▪ Example:



Pipes

- Classic IPC method under UNIX:

`ls -l | more`

- shell runs two processes ls and more which are linked via a pipe
- the first process(ls) writes data (e.g., using write) to the pipe and the second(more) reads data (e.g., using read) from the pipe

- The system call pipe (fd[2])

creates one file descriptor for reading (fd[0]) and one for writing (fd[1])

- allocates a temporary file with an inode and a memory page to hold data

Mailboxes vs. Pipes

- Message types
 - mailboxes may have messages of different types
 - pipes do not have different types
- Buffer
 - pipes - one or more pages storing messages contiguously
 - mailboxes - linked list of messages of different types
- More than two processes
 - a pipe often (not in Linux) implies one sender and one receiver
 - many can use a mailbox

Shared Memory

- Shared memory is an efficient and fast way for processes to communicate
 - multiple processes can attach a segment of physical memory to their virtual address space
 - create a shared segment: `shmid = shmget(key, size, flags)`
 - attach a shared segment: `shmat(shmid, *shmaddr, flags)`
 - detach a shared segment: `shmdt(*shmaddr)`
 - control a shared segment: `shmctl(shmid, cmd, *buf)`

- o if more than one process can access a segment, an outside protocol or mechanism should enforce consistency/avoid collisions

Signals

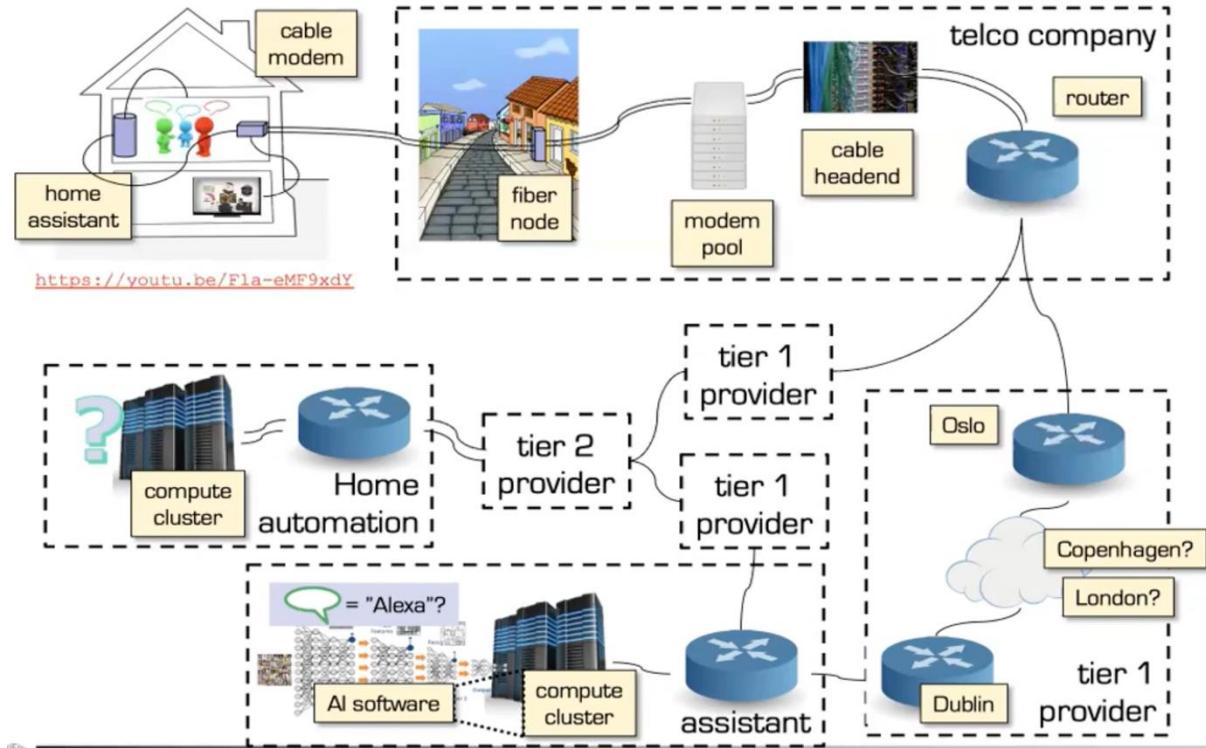
- Signals are software generated “interrupts” sent to a process
 - o hardware conditions
 - o software conditions
 - o input/output notifications
 - o process control
 - o resource control
- Sending signals
 - o `kill(pid, signal)` - system call to send any *signal* to *pid*
 - o `raise(signal)` - call to send signal to current process

Summary

- Many ways to send messages or perform IPC within a machine
 - o mailboxes - FIFO, messages have types
 - o pipes - FIFO, no type
 - o shared memory - shared memory mapped into virtual space
 - o (signals - send a signal which can invoke a special handler)

1.5 Datakom: Intro

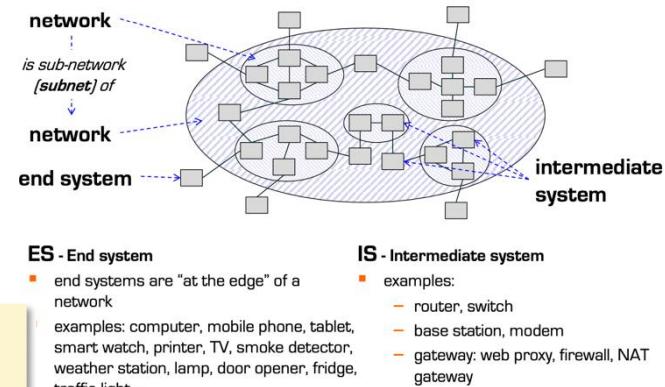
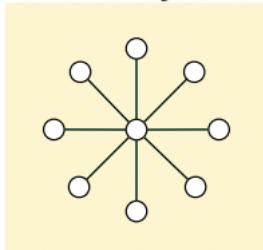
A possible communication path



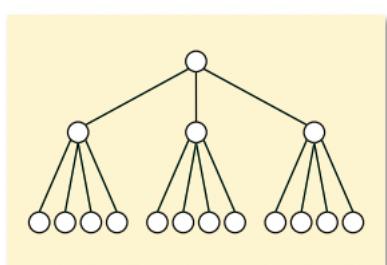
1.6 Network structures

Structures

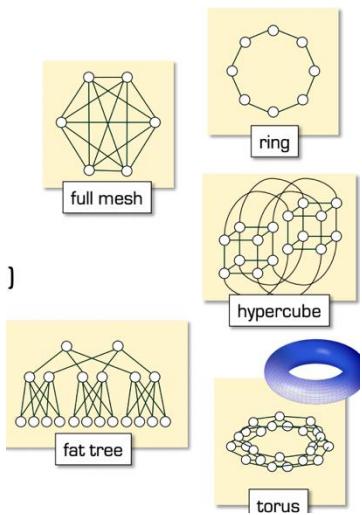
- Components
- Point-to-point channels
 - Docsis (cable modems)
 - star
 - Kabel-tv
 - Gigabit Ethernet
 - Star or tree



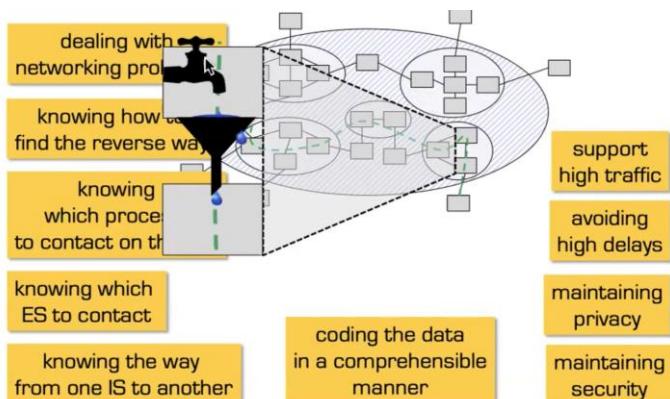
topology



- Other



What is the networks tasks?



File download and Web browsing

- Avoiding high delays
 - uncritical
- support high traffic
 - uncritical
- dealing with networking problems
 - critical

Textual commands and textual chat

- avoiding high delays
 - uncritical
- support high traffic
 - uncritical
- dealing with networking problems
 - critical

Live and on-Demand Streaming

- avoiding high delays
 - some ok
- Support high traffic
 - critical
- dealing with networking problems
 - some ok

AV chat and AV conferencing

- avoiding high delays
 - some ok
- Support high traffic
 - important
- dealing with networking problems
 - some ok

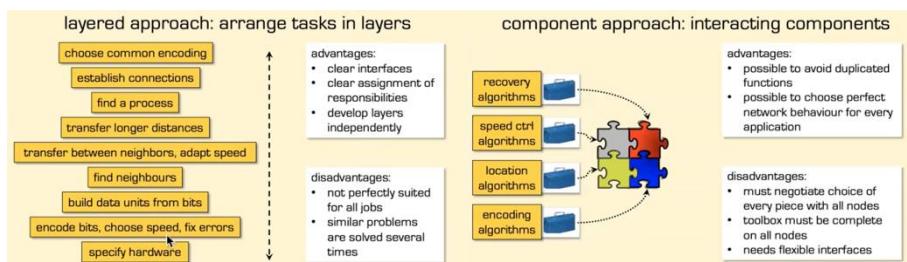
Haptic Interaction

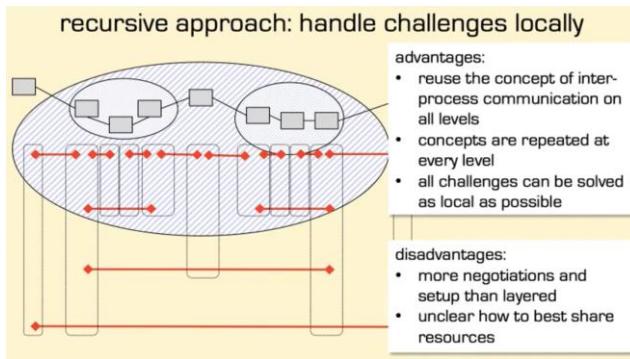
- avoiding high delays
 - critical
- Support high traffic
 - uncritical
- dealing with networking problems
 - some ok

1.6.1 Structuring the tasks

Approaches to structure the tasks

- Vi bruker layered approach



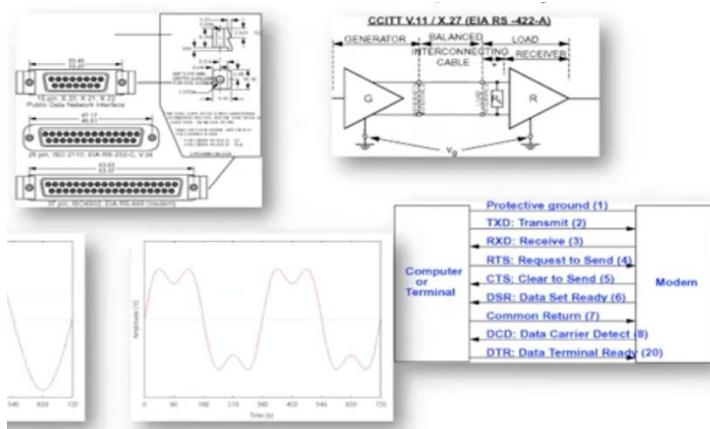


1.6.2 ISO OSI(Open Systems Interconnection)

Layering model

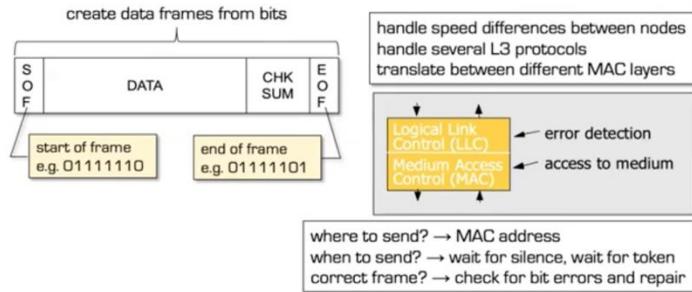
1. Physical layer

1. Insecure bitstream between adjacent systems
 1. mechanics
 2. electronics
 3. procedural



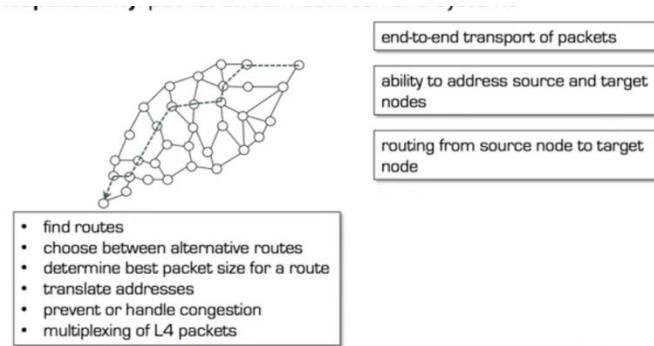
2. Data link layer

1. error-recovering frame stream, adjacent systems
 1. reliable data transfer between adjacent stations with frames



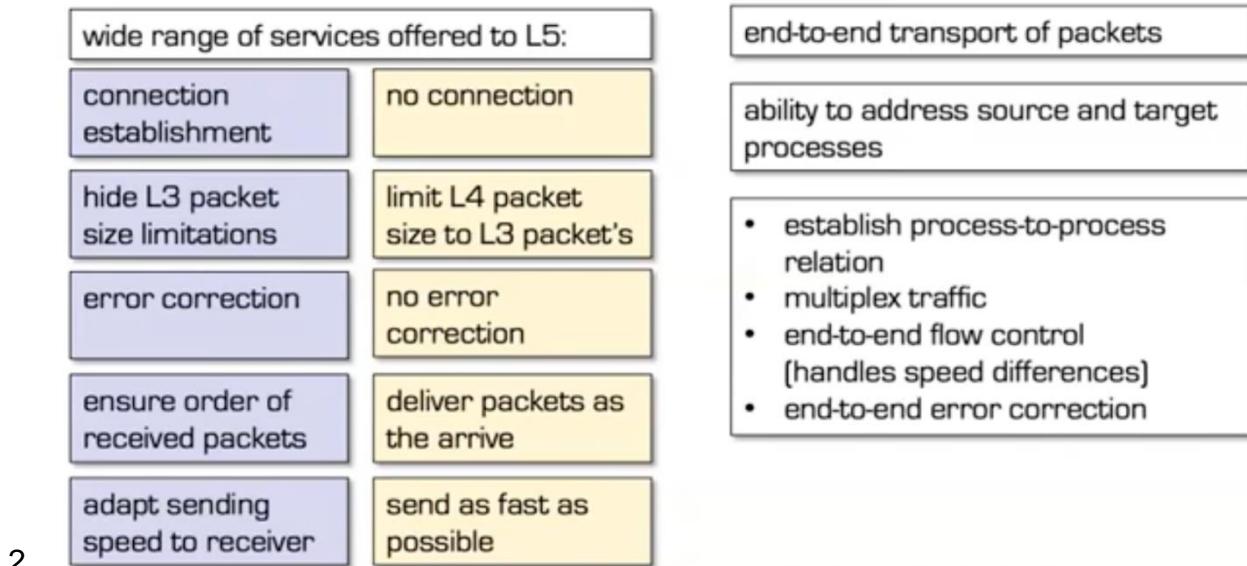
3. Network layer-

1. packet between end systems
 1. end-to-end transport of packets



4. Transport layer

1. end-to-end message stream between processes



5. Session layer / IP

1. structured dialogue
 1. support a session over a longer period

<p>session management</p> <ul style="list-style-type: none"> • establishing identities • assigning writes • tracking identities [cookies] 	<p>token management</p> <ul style="list-style-type: none"> • passing permission to speak in a (large) tele-conference • write-locking of networked files • transaction management in databases
<p>checkpointing</p> <ul style="list-style-type: none"> • make program snapshots to disk • restart after crash 	<p>Google OT (operation transformation) allows Google Docs to work</p> <ul style="list-style-type: none"> • user identify when multiple devices are used • several inputs on the same document at the same time • conflict resolution when writing to the same location
<p>synchronization</p> <ul style="list-style-type: none"> • lip synchronization of speech and video in tele-conferencing • show live football concurrently on all devices 	

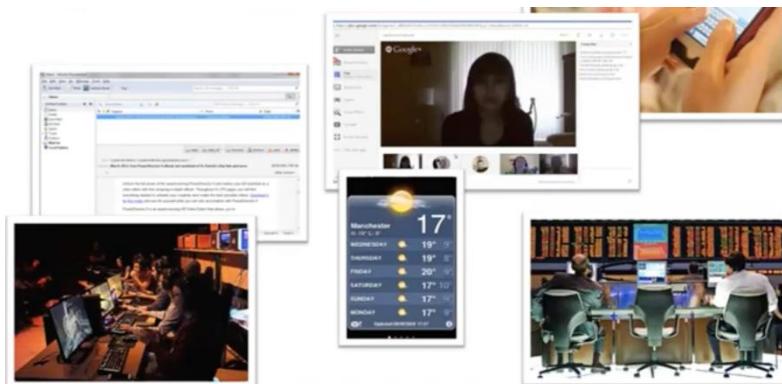
6. Presentation layer

1. exchange of data (semantics)

<p>encoding of int</p> <ul style="list-style-type: none"> • big endian • little endian • XML [as string] • ASN.1 [shortest possible big endian bit sequence] • XDR [4-byte big endian] 	<p>encoding of structs ["serialization"]</p> <ul style="list-style-type: none"> • ASN.1 • XDR • XML • JSON • Java serialization • Google protocol buffers 	<p>file name representation</p> <ul style="list-style-type: none"> • /mnt/user/n.txt • m:\user\n.txt
<p>encoding of strings</p> <ul style="list-style-type: none"> • ASCII • UTF-8 • Unicode • EBCDIC 	<p>encoding of date</p> <ul style="list-style-type: none"> • seconds since 1.1.1970 • nanoseconds since 1.1.1601 • string "12 March 2019 13:32:54 UTC" 	<p>image formats</p> <ul style="list-style-type: none"> • JPG, PNG
		<p>compression</p> <ul style="list-style-type: none"> • zip, gzip, bzip2 <p>encryption methods</p> <ul style="list-style-type: none"> • PGP, S/MIME <p>semantics</p> <ul style="list-style-type: none"> • NOK, EUR, USD

7. Application layer

1. cooperating entities



1.7 Datakom: ARP og IP – naming. INTERNET

(N)-Layer

- abstraction level with defined tasks

(N)-Entity

- active elements within a layer
- process or intelligent I/O module

- peer entities: corresponding entities on different systems

(N)-Service Access Point, (N)-SAP

- service identification
- describes how layer N provides a service for layer N+1
- an Entity can offer several services

(N)-Protocol

- A multitude of rules for transferring data between peers

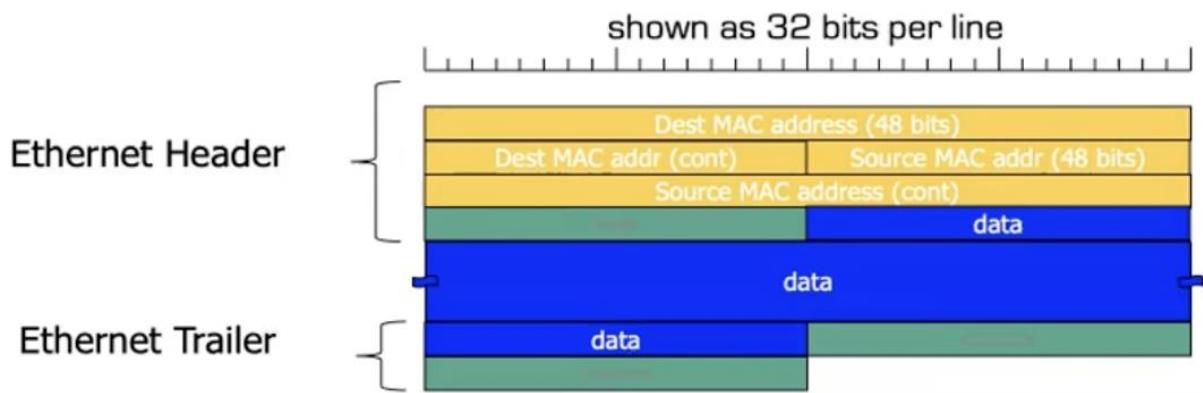
Protocol:

- Defines
 - the format
 - the order of messages
 - exchanged between two or more communicating entities
 - as well as the actions taken on transmission and/or reception of a message or other events
- Does not define
 - the services offered to layer N+1
 - the services used(N-1 -SAP)
- What is it?
 - Protocol syntax: rules for formatting
 - Protocol semantics: rules for actions in case of a message or event
 - Note: semantics must be defined as behaviour of all communicating peers

Transport layer header: UDP example

- port
- the term in Internet protocol for the address of a process on an end system
- the transport layer address
- 64 bit header

Data link layer headers: Ethernet example



Big and small endian:

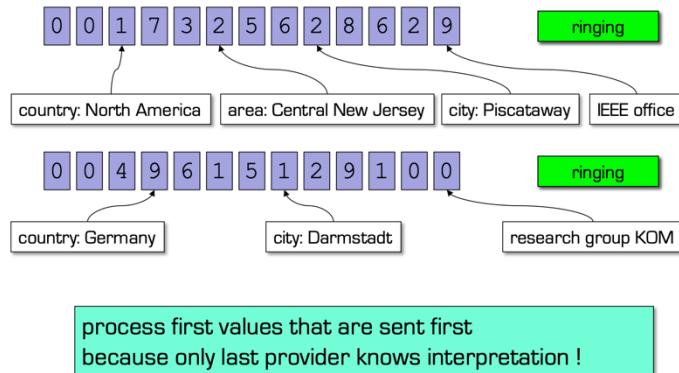
- Big endian and small endian are two ways that computers can store and read data in memory.
- In big endian format, the most significant byte (or the leftmost byte) of a multi-byte data type (such as an integer) is stored first, followed by the rest of the bytes in decreasing order of significance. This means that the data is stored in memory with the most important bits first. It's like writing the date as "year-month-day" instead of "day-month-year".
- In small endian format, the least significant byte (or the rightmost byte) of a multi-byte data type is stored first, followed by the rest of the bytes in increasing order of significance. This means that the data is stored in memory with the least important bits first. It's like writing the date as "day-month-year" instead of "year-month-day".
- The choice between big endian and small endian can affect how data is interpreted and manipulated by the computer. Some computer architectures use one format or the other, and it's important to make sure that data is properly converted between the two formats when necessary.

1.7.1 Network byte order

Big endian:

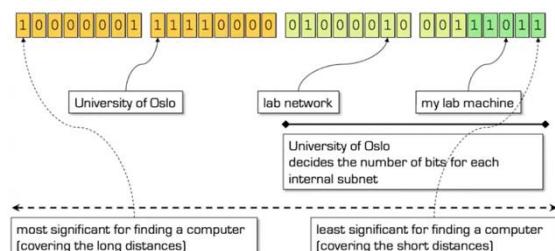
- L5 sends bytes to L4
- L4 passes packets to L3
- L3 adds a header for routing (and more)
- L3 passes frame content to L2
- L2 adds frame header for addressing (and more)
- L2 passes bytes to L1
- L1 transfers bytes
- L1 transfer starts at low memory addresses then continuing to high memory addresses

- speed matters
- headers are in front to process before all bits have arrived
 - analogue in telephone numbers



Bonus for Big Endian

- my lab machine in our lab network
129.240.66.59
0x81 FO 42 3B



Addressing

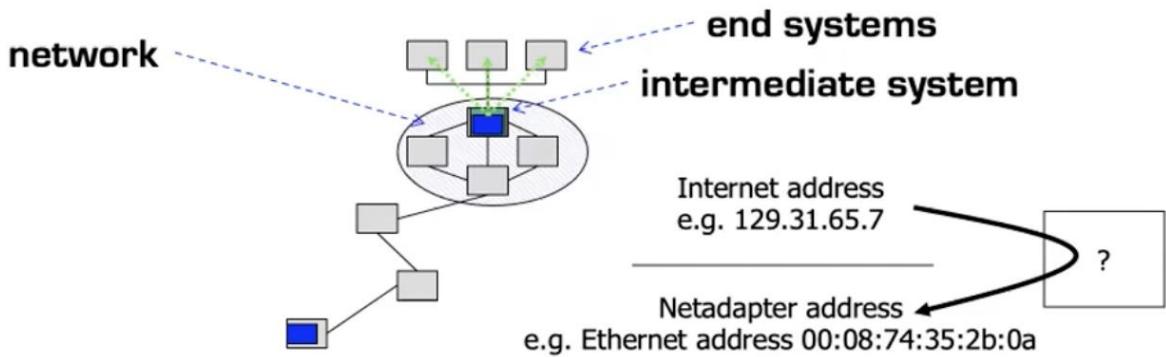
MAC addresses

- are important in. a true broadcast network

Challenge

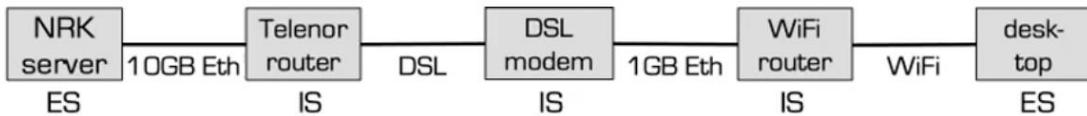
- MAC addresses have only local meaning
- nodes on the “other side” of an IS(intermediate system) may not know them

Address resolution



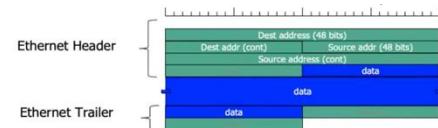
■ Problem

- Potentially every link can use a different L2 protocol



- Different L2 protocols have different address styles
- IP address must be mapped onto the MAC address
48 bit for Ethernet and WiFi, DSL may use 20 or 48 bits

- Ethernet example

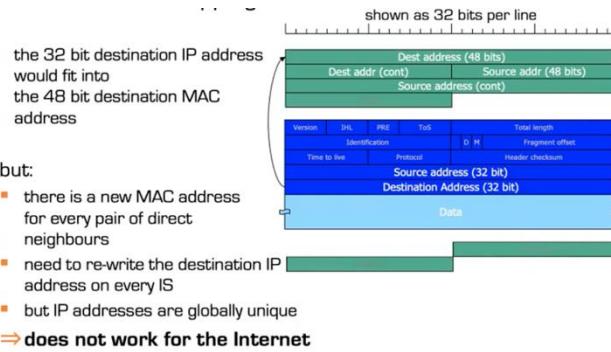


- MAC address structure
 - Ethernet and WiFi are L2 layers using "EUI-48":
Extended Unique Identifier with 48 bits
 - 6 bytes, written like this: f2:18:98:3a:b8:9t
 - to recognize easily that the text is supposed to mean a MAC address
- Ethernet MAC addresses should be globally unique

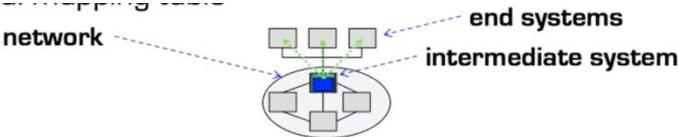
- IANA and IEEE decide how to split the address space
 - first 3 bytes explain whether an address is special
OR
 - first 3 bytes determine who owns the address range
 - e.g.:
 - F0:18:98 : Apple, Inc.
 - 78:45:C4: Dell Inc.
 - 00:50:56: VMWare, Inc.
 - B8:AC:6F: Dell Inc.

IANA - Internet Assigned Number Authority
IEEE - Institute of Electrical and Electronics Engineers

- 1st idea: direct mapping



- **2nd idea: mapping table**



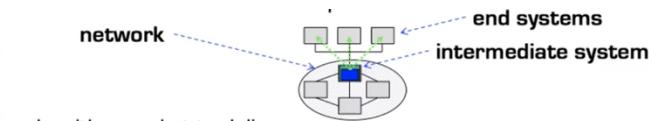
every node maintains a table that maps
IP address ↔ MAC address
for every network interface and
for every directly reachable node (L2 neighbour)

idea 2.1: manually maintained by people

- a lot of work, but not unrealistic – IFI allows only well-known MAC addresses in well-known network plugs – **could be used for this but is not**

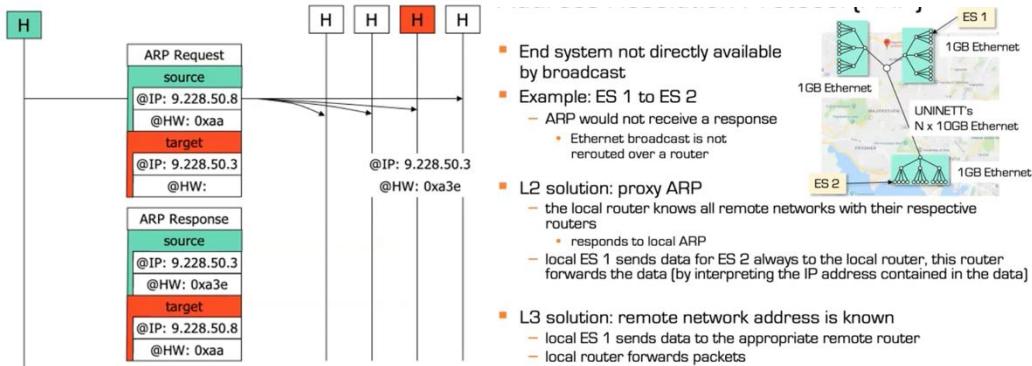
idea 2.2: established by broadcasts from stations

3rd idea: address resolution protocol



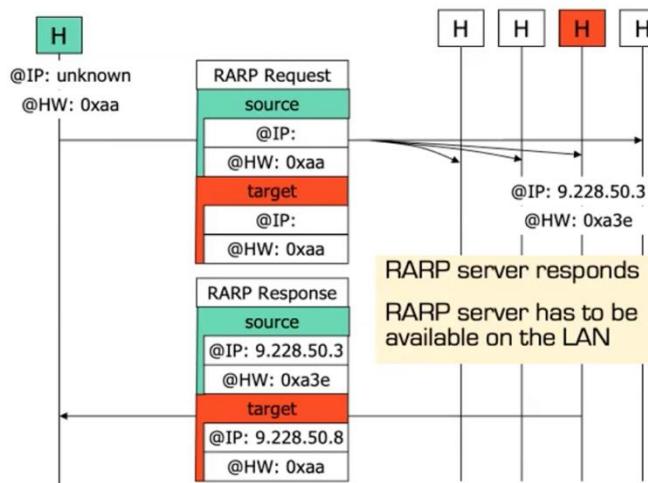
node with a packet to deliver:
if a local cache contains **IP address ↔ MAC address**
send packet & update cache removal timeout
else
send broadcast to all stations
"Who has IP address?"
if one node responds
add **IP address ↔ MAC address** mapping to cache
set timeout for removal from cache to some minutes
send packet
else
drop packet

Address Resolution Protocol (ARP)



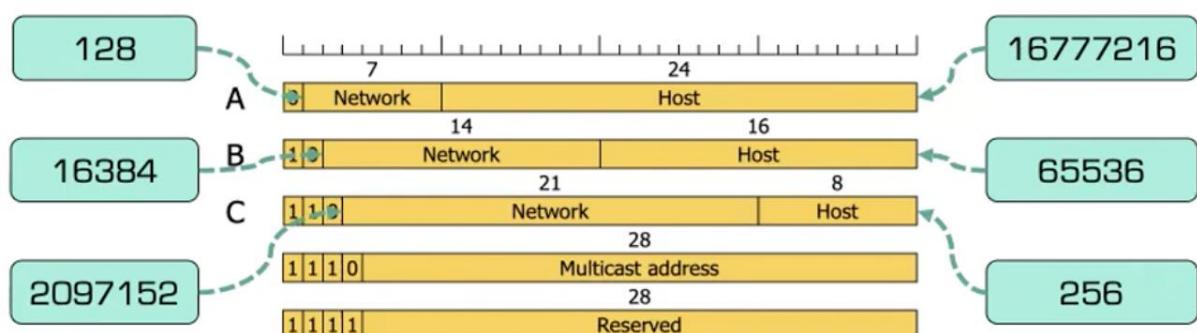
Reverse Address Resolution Protocol (RARP)

- Bruker det ikke i dag



Internet Addresses and Internet Subnetworks

- Original global addressing concept for the Internet
 - For addressing end systems and intermediate systems
 - each network interface has its own unique address
 - 5 classes

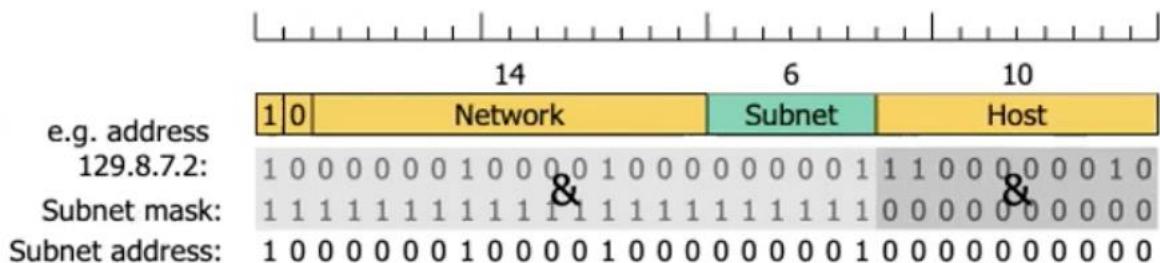


- ICANN (Internet Corporation for Assigned Numbers and Names)

- manages network numbers
 - delegates parts of the address space to regional authorities

Internet Address and Internet Subnetworks

- Networks grow and should be somehow structured
 - several networks instead of one preferable
 - but getting several addresses areas is hard
 - since address space is limited
 - e.g.m university may have started with class B address, doesn't get second one
 - Problem
 - class A, B, C refer to one network, not collection of LANs
 - Allow a network to be split into several parts
 - for internal use
 - still looks like single network to outside world
 - Idea
 - local decision for subdividing host share into subnetwork portion and end system portion



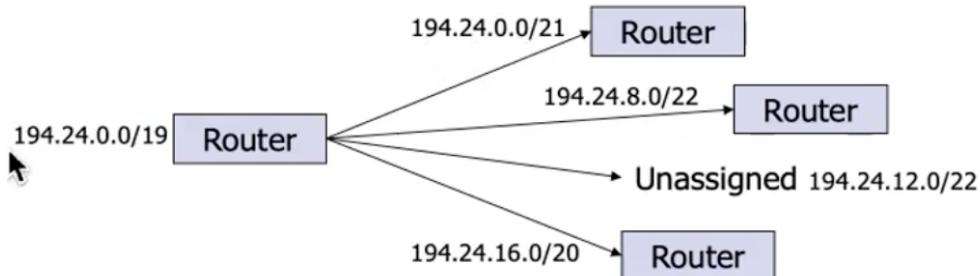
- Use “subnet mask” to distinguish network and subnet part from host part
 - Computers inside the network 129.8.4.0/22 can make 3 checks
 - algorithm in router
 - packet to another network (& with Network mask is different)
 - packet to local end system (& with Network and Subnet masks are the same)
 - packet to other subnetwork(& with Network mask is the same, but Subnet is not)

1.8 Datakom: Addressering på L4 og L5

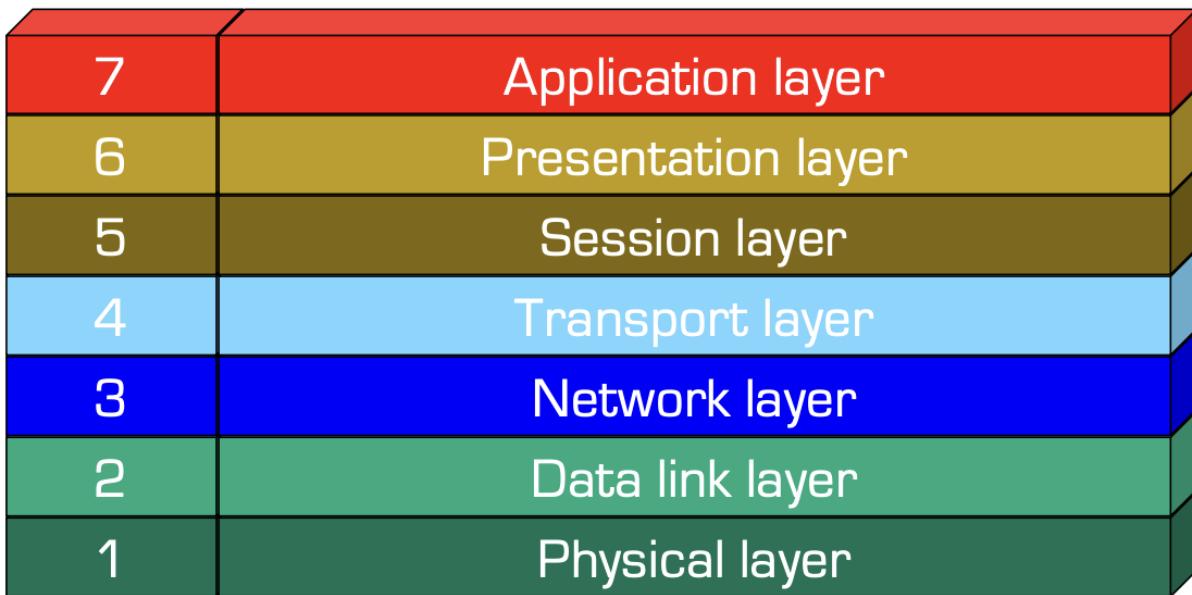
CIDR Classless InterDomain Routing

- Subnetting not good enough
 - Too many organizations require addresses

- in principle many addresses due to 32-bit address space
 - but inefficient allocation due to class-based organization
 - class A network with 16 million addresses too big for most cases
 - class C network with 256 addresses is too small
 - most organizations are interested in class B network, but there are only 16 384 (in reality, class B too large for many organizations)
- Principle
 - to allocate IP addresses in variable-sized blocks
 - e.g. a request for 2000 addresses would lead to
 - assignment of 2048 address block starting on 2048 byte boundary
- dropping classes makes forwarding more complicated
- Large number of networks leads to large routing tables
- Search for longest matching prefix
 - if several entries with different subnet mask length may match
 - then use one with the longest mask
 - i.e. AND operation for address & mask must be done for each table entry
- Entries may be aggregated to reduce routing tables



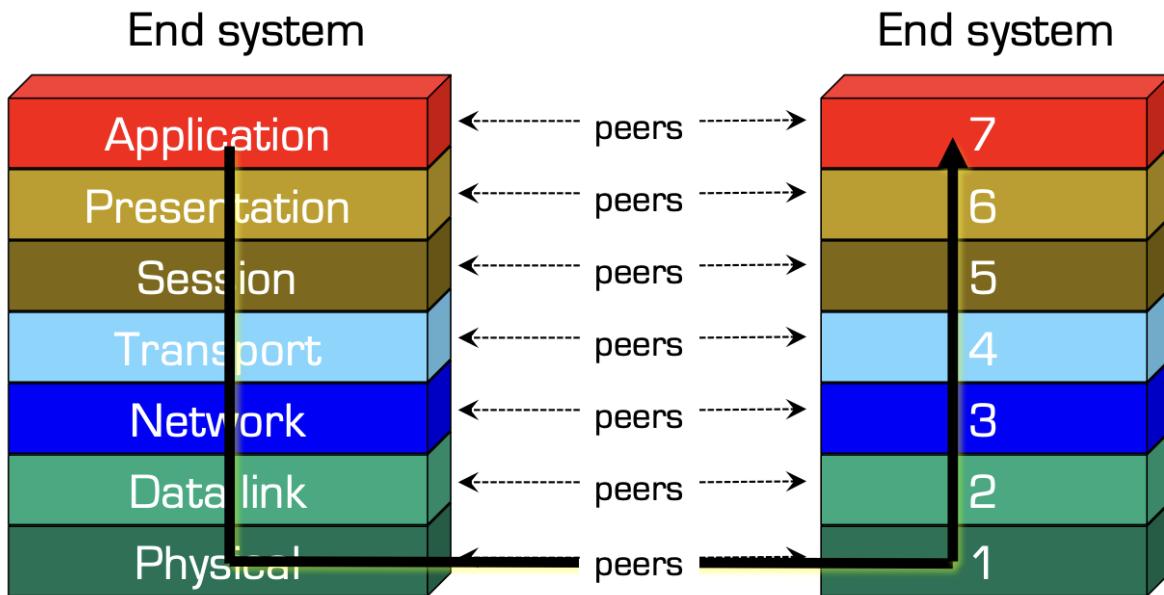
Network Layers



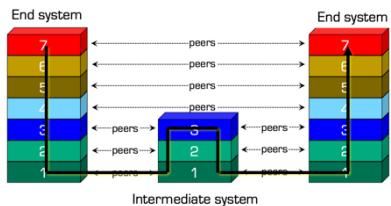
Responsibilities for the Layers

1. Physical
 1. Insecure bitstream between adjacent systems
 1. mechanics
 2. electronics
 3. procedural
2. Data link
 1. Error-recovering fram stream, adjacent systems
 1. Reliable data transfer between adjacent stations with frames
3. Network
 1. Packet stream between end systems
 1. end-to-end transport of packets
 2. ability to address source and target nodes
 3. find routes
4. Transport
 1. end-to-end message stream between processes
5. Session/IP.
 1. structured dialogue
 1. support a “session” over a longer period
6. Presentation
 1. exchange og data (semantics)
7. Application
 1. cooperating entities

Data flow between adjacent systems

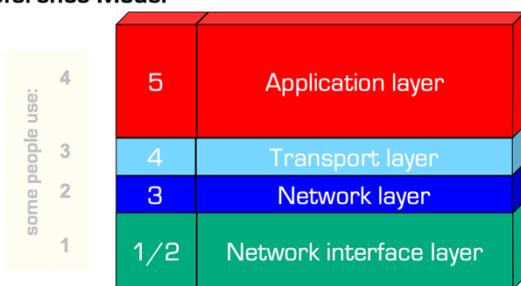


data flow between two non-adjacent systems



TCP/IP

OSI Reference Model



Why no clear separation of upper layers?

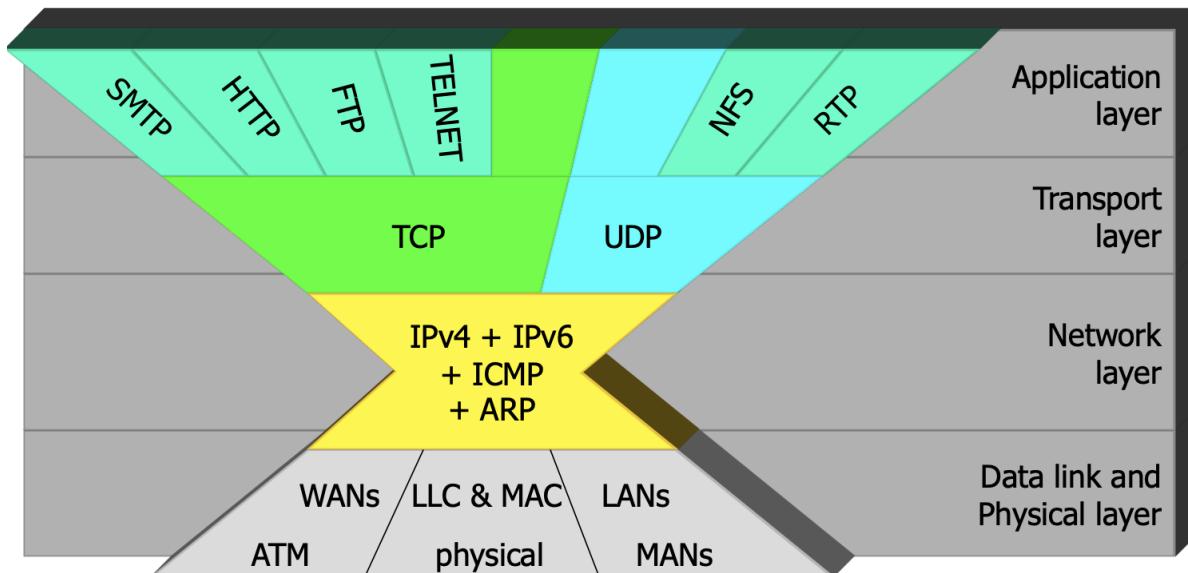
- Layers 1-4 are essential for co-existence on the internet
 - e.g different congestion control mechanisms on different hosts can lead to strong congestion
- session and presentation layer functions provide mostly application support

Layers 3 and 4 are not clearly separated

- Transport protocol and network protocol IP

- Sometimes hard to draw a clear line where TCP ends and IP begins

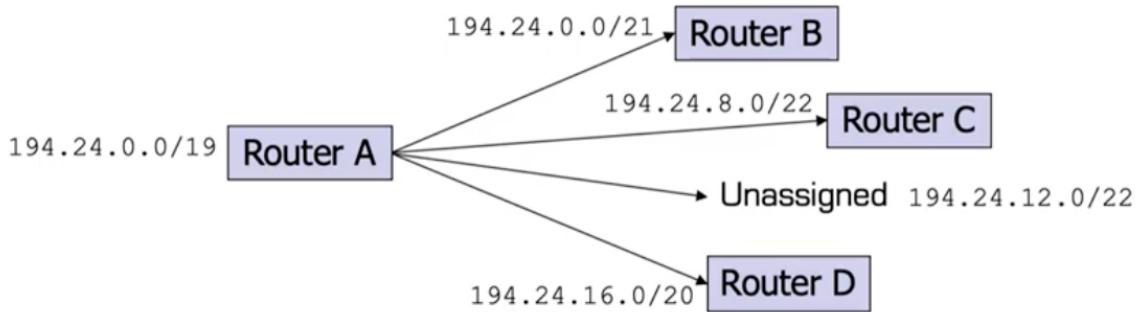
Internet Protocol Stack



Nickname: “Hourglass Model”

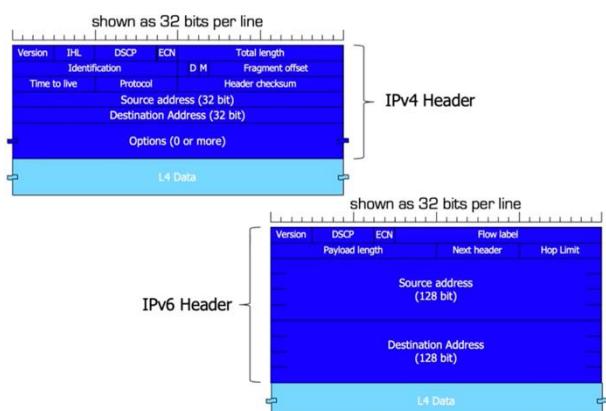
Classless InterDomain Routing

- Idea
 - Local decision for subdividing host share into network portion and end system portion
- Use “netmask” to distinguish network part from host part
- Routing with 3 levels of hierarchy
 - end system: compute “dst addr & netmask == subnet addr”
 - TRUE: packet to local end system
 - FALSE: packet to another network(send to this network’s router)
 - router: compute “dst addr & netmask == subnet addr”
 - TRUE: packet to local end system
 - FALSE: packet to another network(look up in routing table, send to other router)
- Rule of the longest match
 - if several entities with different subnet mask length may match then use the one with the longest mask



1.8.1 IPv6

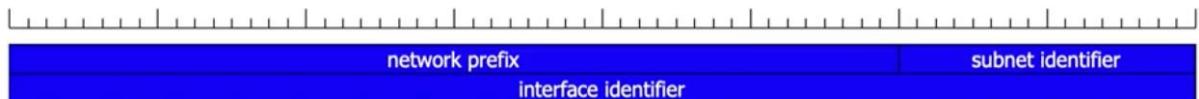
- Motivation for IPv6 and problems with IPv4
 - Too few addresses
 - Bad support for QoS
 - Bad support for mobility
 - Many other shortcomings
- Objectives -
 - Scalability
 - To support billions of end systems
 - longer addresses
 - To reduce routing tables
 - to simplify protocol processing
 - Addressing IPv4 limitations
 - To increase security
 - To support real-time data traffic
 - To provide multicasting
 - To support mobility
 - Coexistense
 - To be open for change
 - To coexists with existing protocols



Example of the IPv6 address spaces

- Veldig lik som IPv4 sine, men har 4 ganger så mange bits

shown as 64 bits per line



a typical routed address

- Oversatt til hexadecimale verdier

2 a 0 0 : 1 4 5 0 : 4 0 0 f : 0 8 0 a : 0 0 0 0 : 0 0 0 0 : 0 0 0 0 : 2 0 0 4

a bit more compact:

2a00:1450:400f:080a:0000:0000:0000:2004

This is an address for www.google.com

- IPv6 addresses are written in sets of 2 bytes in hexadecimal notation
 - 2a00:1450:400f:080a:0000:0000:0000:2004
 - sets of zero can be compressed
 - 2a00:1450:400f:80a::2004
 - this address is part of the network
 - 2a00:1450:400f::/48
 - which is known to be used by Google since 12/2018

IPv6 addresses

Acquiring IPv6 addresses for mobile computers

1. Getting a non-routable IPv6 address using auto-configuration
 1. self-assigns an IPv6 address consisting of prefix FE80::0 followed by the interface identifier (RFC4291), which is created from the MAC address (RFC 8064)
 2. before using the address, “Neighbour Solicitation” ICMP message must be sent to ensure the address is not in use yet - variation of ARP probe
 2. DHCPv6 to ask for a routable address
 1. required auto-configured local address first
 3. Mobile IPv6 to transfer routable home address to visited network
 1. requires auto-configured local address first

Layer 4 Address Resolution - Transportlaget

Tasks

1. Addressing
2. End-to-end connection management
3. Transparent data transfer between processes
4. Quality of service options
 - o Error recovery
 - o Reliability
 - o Flow control
 - o Congestion control

Addressing at the Transport Layer

Port Number	Service	Service name
22	SSH	Secure Shell
25	SMTP	Simple Mail Transfer Protocol
53	DOMAIN	Domain Name Systems
80	HTTP	Hypertext Transport Protocol
110	POP3	Post Office Protocol, version 3
123	NTP	Network Time Protocol
143	IMAP	Internet Message Access Protocol

- TCP and UDP have separate assignments
 - o same port number - different service
 - o The table displays for TCP

Multiplexing

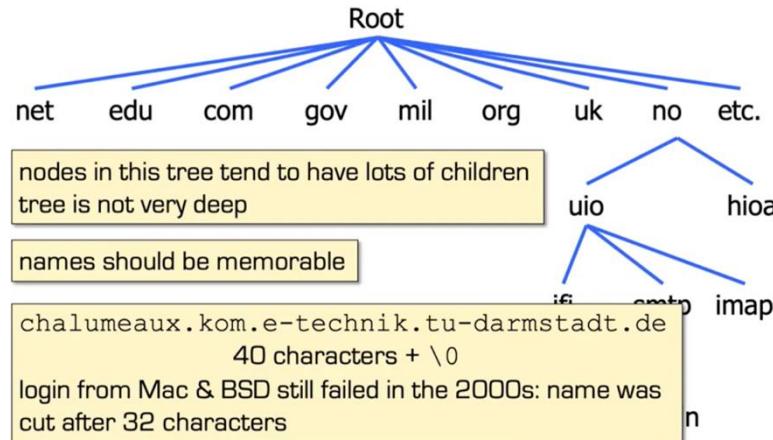
- Multiplexing and demultiplexing task of the transport layer
- Accessing a web page with video element
 - o HTTP for web page
 - o RTSP for video control
 - o RTP for video data

Connection - Addressing

- TCP service obtained via service endpoints on sender and receiver
 - o Typically socket
 - o Socket number consists of 3-tuple
 - IP address of host and
 - 16-bit local number(port)
 - TCP protocol identifier
- Transport Service Access Point
 - o Port
- TCP connection is clearly defined by a 5-tuple consisting of
 - o IP address of sender and receiver

- Port address of sender and receiver
- TCP protocol identifier

Naming Hierarchy



Server Hierarchy

Functions of each DNS server

- Authority over a portion of the hierarchy
 - No need to store all DNS names
- Stores all the records for hosts/domains in its zone
 - Must be replicated for robustness
- Know the addresses of the root servers
 - Resolve queries for unknown names

Recursive DNS Query

Classical approach

- Must keep state for every request in a server until answered
- Allows every node along the path to cache results
- Concentrates the data flow at the central servers
- Keeps a lot of state on central servers

Newer approach

- Redirects requests
- Keep state only at local server until answered
- Allows few nodes to cache results
- Halves number of requests at central servers
- Avoids state on central servers entirely

Caching vs. Freshness

- Caching reduces DNS resolution latency
- Caching reduces server load
- Caching delays updates

Multicast DNS

- A way of discovering services by announcing them with IP multicast
 - RFC 6762: multicast DNS
 - records announce services
 - records are never authoritative and mDNS can never redirect or recurs

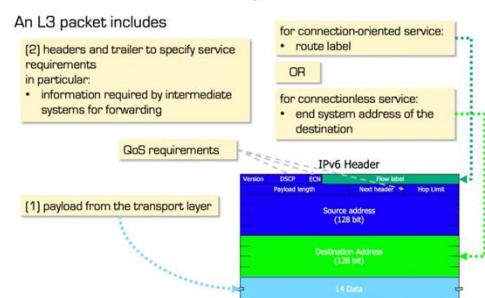
1.9 Datakom: Forbindelsesorientert og forbindelsesløs kommunikasjon

Network Layer

Provide service to the transport layer:

- Connectionless
 - Everyone has the responsibility to find its own way
- Connection-oriented service
 - Everyone follows route
- Uniform addressing
- Internetworking
 - Provide transitions between networks
- Routing
- Metningskontroll
- Quality of service

Inside the Network Layer:



An L3 packet includes

- Payload from the transport layer

- Headers and trailers
 - End system address or route label
 - QoS requirements
- Knowledge required by intermediate system
 - Subnetwork topology
 - Address / localization of the end system
 - Packet / data stream communication requirements
 - Network status
- Intermediate system can route and switch packets

Terminology - Routing and Switching

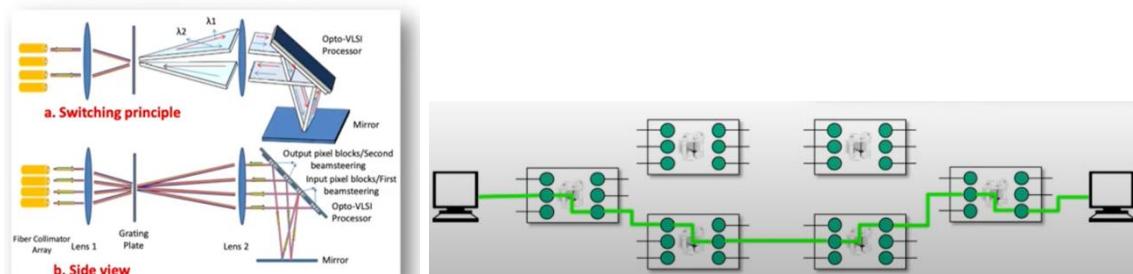
- Routing and Switching have one thing in common:
 - A packet arrives at an IS, and the IS (if it is not the target) decides to choose the right interface for forwarding it
- In the Internet, there is a strong historical association of switching with L2 and routing with L3
- In packet-based networks like the Internet, we call something
 - **routing**, when an IS reads a destination address from an arriving packet, computes which of its direct neighbors is best suited for reaching that destination, and sends the packet to the neighbor
 - **switching**, when an IS reads an identifier from an arriving packet, looks it up in a pre-filled mapping table that translates the identifier to a direct neighbor, and sends the packet to the neighbor

Switching

Circuit switching - Linjesvistsjing

- Principle
 - Connection exists physically for the duration of the conversation
 - Refers to
 - Switching centers
 - Connections between switching centers

- Today: optical networks [e.g. WDM]
- Setting coupling points in circuits



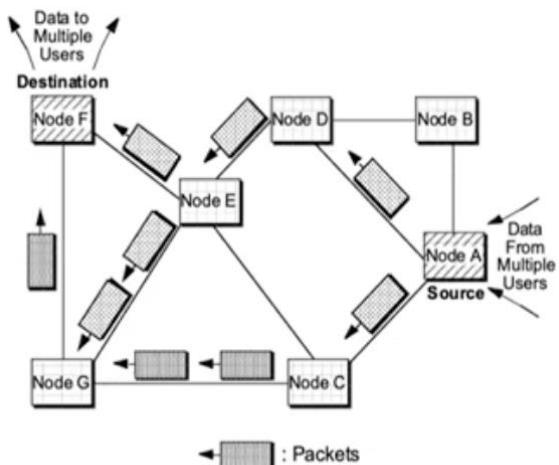
<p>■ Circuit switching</p> <ul style="list-style-type: none"> — Connection establishment can take a long time — Bandwidth is reserved <ul style="list-style-type: none"> • No danger of congestion • Possibly poor bandwidth utilization (burst traffic) 	<p>■ Packet switching</p> <ul style="list-style-type: none"> — No connect phase — No allocation of bandwidth <ul style="list-style-type: none"> • Danger of congestion • Potentially 100% bandwidth utilization
<ul style="list-style-type: none"> — constant transmission time — all data is transmitted over the same path 	<ul style="list-style-type: none"> — varying transmission time — packets between same end systems may use different paths

- Properties
 - Connection has to occur before transmission
 - Establishing a connection takes time
 - Resource allocation is rigid

Packet switching - pakkesvitsjing

- Principle
 - Packets of limited size
 - Dynamic route search
 - No dedicated path from source to destination

Internet:



- Properties
 - Possibly only reservation of average bandwidth

- Possibility of congestion
- High utilization of resources

Comparing Circuit and Packet Switching

Virtual Circuit Switching

- Principle
 - setup path from source to destination for entire duration of call
 - Using state information in nodes but no physical connection
 - Connection setup: defines data path
 - Messages: as in packet switching, but:
 - No routing decision per packet
 - Simpler addressing: associate with virtual circuit instead of destination

Comparison: Virtual Circuit and Packet Switching

- Connection set-up phase
 - select a path
 - assign a VC identifier (Virtual Circuit)
 - IS (intermediate system) stores path information
 - network reserves all resources required for the connection
- Data transfer phase: all packets follow the selected path
 - every packet contains VC number
 - identification of connection,
 - IS uses stored path information to determine next node
 - maybe update VC number in packet
- Disconnect phase:
 - IS releases reserved resources
 - IS forgets the VC
- Properties
 - Possible to ensure QoS
 - Possible to maintain order of packets

Implementation Virtual Circuit

- Packet passes through the network as an isolated unit
 - has complete source and destination addresses
 - individual route selection for each packet
 - generally no resource reservation

- correct sequence not guaranteed

Packet Switching vs. Virtual Circuit

Message Switching

- Principle
 - All data to be sent are treated as a “message”
 - Store and forward network:
 1. Accepted
 2. Treatment of possible errors
 3. Stored
 4. Forwarded
- Ancient history
 - dial-up networks
 - and its protocol UUCP - Unix to Unix Copy
- Properties
 - High memory requirements at the node
 1. message size is unlimited
 2. usually stored on secondary repository

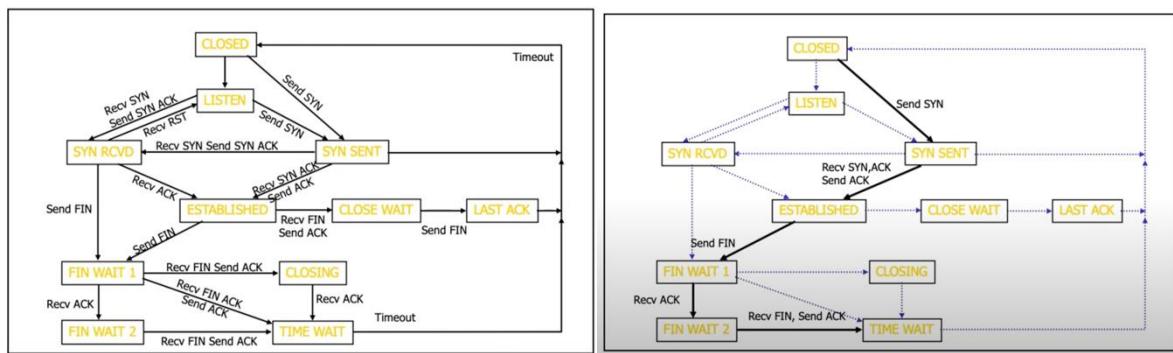
1.10 Datakom: Flytkontroll

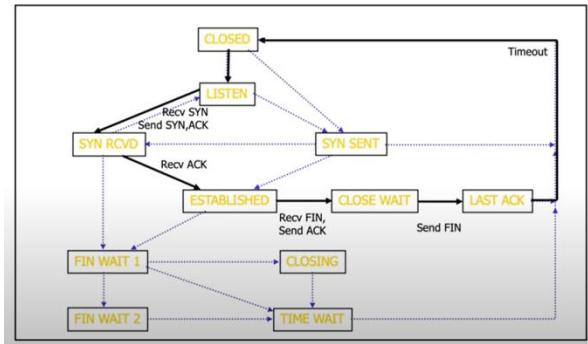
TCP

- TCP is the main transport protocol of the Internet
- Provides
 - Reliable end-to-end byte stream over an unreliable network service
- Motivation: Network with connectionless service
 - Packets and messages may be
 - duplicated, in wrong order, faulty
 - i.e., with such service only, each application would have to provide recovery
 - Network or service can
 - impose packet length
 - define additional requirements to optimize data transmission
 - application would have to be adapted
- Characteristics

- Data stream oriented
 - TCP transfers serial byte stream
 - Maintains sequential order
- Unstructured byte stream
 - Application often has to transmit more structured data
 - TCP does not support such groupings into structures within byte stream
- Buffered data transmission
 - Byte stream not message stream: message boundaries are not preserved
 - no way for receiver to detect the units in which data were written
 - For transmission for the sequential data stream is
 - Divided into segments
 - Delayed if necessary
- Virtual connection
 - Connection established between communication parties before data transmission
- Two-way communications
 - Data may be transmitted simultaneously in both directions over a TCP connection
- Point-to-point
 - Each connection has exactly two endpoints
- Reliable
 - Fully ordered, fully reliable
 - Sequence maintained

TCP State Machine





Message format:

- Går serveren inn i listen-tilstand så
 - Ser man på **Destination Address** og **Destination port**
 - Brukt til demultiplexing
- Hvis man vil tillate at serveren skal kunne vente på flere klienter, mens den håndterer den første
 - Ser man på **Source address** og **Source port** i tillegg til nevnt over

TCP in Use & Application Areas

- Reliable data transmission
 - Efficient data transmission despite complexity
 - Can be used with LAN and WAN for
 - low data rates
 - high data rates
- Disadvantages when compared with UDP
 - Higher resource requirements
 - buffering, status information, timer usage
 - Connection set-up and disconnect necessary
- Applications
 - everything on the Web including video-on-demand
 - File transfer
 - Interactive terminal
 - E-mail

Transport layer

Bandwidth

- Various wooly “bandwidth” terms
 - Nominal bandwidth
 - Bandwidth of a link when there is no traffic
 - Available bandwidth:
 - (Nominal bandwidth - traffic)... during a specific interval
 - Bottleneck bandwidth

- smallest nominal bandwidth along a path, but sometimes also smallest available bandwidth along a path
- Throughput:
 - Bandwidth seen by the receiver
- Goodput:
 - bandwidth seen by the receiving application
 - TCP: goodput \neq throughput
 - Difference is not only message headers - more required to ensure reliability, ordering, flow control, etc.

Delay

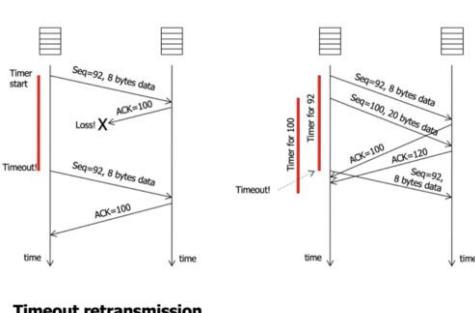
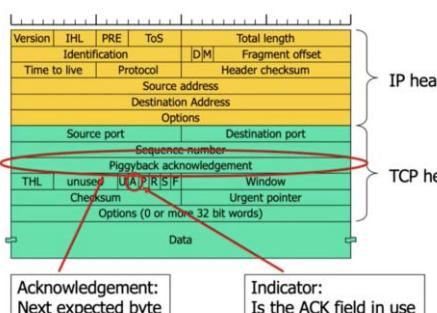
- Propagation delay
 - time to transfer a bit
- Latency
 - propagation delay + msg_length/bottleneck bandwidth + queuing delay
- Jitter
 - delay fluctuations, very critical for most real-time applications
- End-to-end delay
 - Time a message needs from a sending entity to a receiving entity
 - end-to-end delay on the application layer includes TCP retransmission
 - on the transport layer does not
- Round-trip time (RTT) - time a message needs to go from sender to receiver and back

Reliability

- TCP/IP has only two tools for ensuring reliable transfer
 - detecting duplicate byte ranges
 - retransmitting data that has not arrived and not arrived correctly

TCP Reliability

- ICP/IP retransmits segments until they are acknowledged
- ACK field is used for that



TCP Round trip time and timeout

- How to set the retransmission timer?
 - Must be longer than round-trip time
 - Too short: many spurious retransmissions
 - Too long: slow reaction, flow control buffer fills up
- Idea
 - Estimated RTT
 - Use a SampleRTT: Measure time from segment sending and ACK received
 - Compute EstimatedRTT as a floating average
 - Increase the estimatedRTT with a safety margin that is big when SampleRTT changes a lot

$$x = 0.1$$

$$\text{Deviation} = (1 - x) * \text{Deviation} + x * |\text{SampleRTT} - \text{EstimatedRTT}|$$

$$\text{EstimatedRTT} = (1 - x) * \text{EstimatedRTT} + x * \text{SampleRTT}$$

$$\text{Timeout} = 3 * \text{EstimatedRTT} + 4 * \text{Deviation}$$

Flow control and error treatment

- Problem
 - sender can send faster than receiver can receive
- Without flow control
 - receiver loses packets despite error-free transmission
- With flow control
 - sender can adapt to receiver's abilities by feedback
- Comment
 - Error control and flow control are usually interlinked

1.10.1 Transport layer

1.10.2 Hva er context-switch og hvordan gjøres det?

1.10.3 Flow control: Stop-and-wait:

1. The sender sends a data unit (e.g., a message or a part of a file) to the receiver.
2. After sending the data unit, the sender waits for an acknowledgment (ACK) from the receiver to confirm that the data unit has been received correctly.

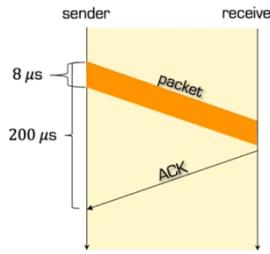
3. The receiver receives the data unit and checks if it is error-free. If the data unit is correct, the receiver sends an ACK to the sender to confirm that the data unit has been received correctly.
4. If the receiver detects errors in the data unit, it sends a negative acknowledgment (NAK) to the sender, requesting retransmission of the data unit.
5. Upon receiving an ACK from the receiver, the sender knows that the data unit has been received correctly and then proceeds to send the next data unit if there is more data to be transmitted. If the sender receives a NAK, it resends the same data unit.
6. After sending a data unit or a retransmission, the sender starts a timer.
7. If the sender receives an acknowledgment within the time interval defined by the timer, it goes back to step 1 and sends the next data unit. If the timer expires before the sender receives an acknowledgment, it assumes that the data unit or retransmission was lost, and it resends the same data unit.

Stop-and-Wait

- sender can never send new packet before ACK or timeout
- connection is idle most of the time
- poor throughput

Gigabit Ethernet channel

- transmission rate: 1 Gbps
- roundtrip delay 200 μ s (2×0.1 ms)
- packet size: 8000 bit
- in comparison
→ ACK is short and negligible

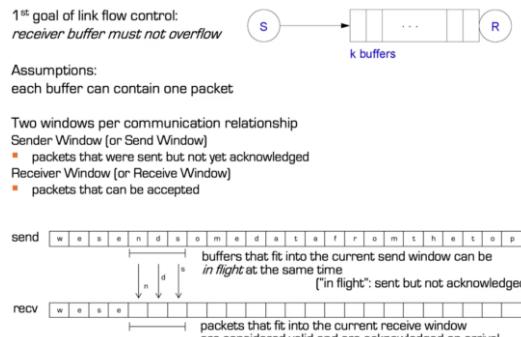


this means

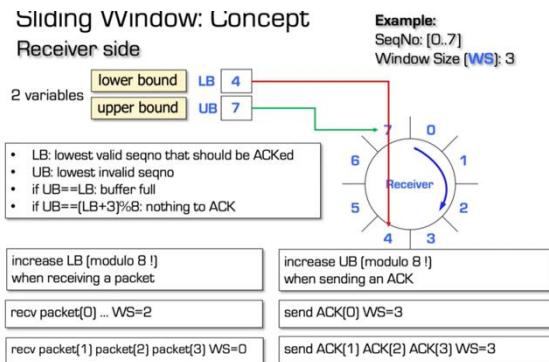
- sending takes 8000 bit / $1,000,000,000$ bps = 8μ s
 - sender is blocked for 192μ s of 200μ s
- Channel utilization 4%

1.10.4 Flow control: Sliding Window

- Improve utilization: pipelining
- Flow control: sliding window mechanism



1.11 Sliding window: Concept



Here's how the sliding window protocol works:

1. Sender's Perspective:

- The sender divides the data stream into smaller data units, each assigned a unique sequence number.
- The sender maintains a sending window that represents the range of sequence numbers for which it can send data units.
- Initially, the sending window is empty, and the sender starts transmitting data units within the window.
- As the sender receives acknowledgments (ACK) from the receiver for the transmitted data units, it advances the sending window, allowing it to send new data units.
- If the sender does not receive an ACK for a particular data unit within a specified timeout period, it assumes the data unit was lost and retransmits it.

2. Receiver's Perspective:

- The receiver maintains a receiving window that represents the range of sequence numbers it is ready to accept.
- Initially, the receiving window is set to the first expected sequence number.
- As the receiver receives data units within the receiving window, it sends acknowledgments (ACK) to the sender to confirm their receipt.
- The receiver buffers the received data units for further processing.
- If the receiver receives an out-of-order data unit, it discards it but sends an ACK with the highest in-order sequence number to inform the sender about the missing data units.
- The receiver also acknowledges duplicate data units to handle potential retransmissions.

The sliding window protocol allows for pipelining, where multiple data units can be in transit simultaneously, thereby increasing the network's throughput. It provides a balance between the sender's transmission rate and the receiver's processing

capability. The size of the sliding window can be dynamically adjusted based on network conditions, such as available bandwidth and receiver buffer space.

Stored packets at the sender

- the sender must store all unacknowledged packets and be able to retransmit
- maximum number defined by sender's window size
- the packets not yet acknowledged by the receiver

Stored packets at the receiver

- not necessary to store any packets
- not useful to store more than one receiver window size

ACK sent by receiver only if the packet

- has been identified as being correct
- can be transmitted correctly to the application

Retransmission

Retransmission is an important aspect of the sliding window protocol, as it ensures the reliable delivery of data units in the presence of errors or lost packets. When a data unit is not acknowledged within a specified time period, the sender assumes that it has been lost or damaged and initiates a retransmission.

Here's how retransmission works in the sliding window protocol:

1. Sender's Perspective:

- The sender maintains a timer for each data unit it sends. When a data unit is sent, the corresponding timer is started.
- If the sender does not receive an acknowledgment (ACK) for a specific data unit within the timeout period, it assumes that the data unit was lost or damaged during transmission.
- Upon timeout, the sender retransmits the lost or damaged data unit. The sender may choose to retransmit only the specific data unit or the entire window, depending on the protocol variant and network conditions.
- The sender updates the sending window to accommodate the retransmitted data unit.

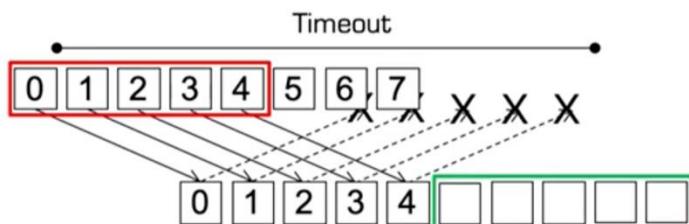
2. Receiver's Perspective:

- When the receiver detects an error in a received data unit, it discards the corrupted data unit and does not send an acknowledgment (ACK).

- The receiver may send a negative acknowledgment (NAK) or use selective repeat to request the retransmission of the missing or corrupted data unit.
- Upon receiving a retransmission request (NAK or selective repeat), the sender retransmits the requested data unit or data units.
- The receiver buffers the received data units in case they arrive out of order. Once all the expected data units have been received successfully, the receiver delivers them to the higher layers of the protocol stack.

Retransmission in the sliding window protocol ensures that lost or corrupted data units are retransmitted until they are successfully received by the receiver. This mechanism helps maintain the reliability of the data transmission and ensures that the receiver can reconstruct the original data stream accurately.

Selective repeat



Correlation between

- window size and
 - number of possible sequence numbers
- max. window size $\leq 1/2$ range of sequence numbers

Example for **incorrect** window size:

- amount of sequence numbers 8
- window size 5
- **all ACKs are lost**, and the packet that has been lost last is the first one to arrive at the receiver again

Credit Mechanism

- Flow control
 - Credit mechanism
- Buffer reservation
 - Receiver allocates buffers dynamically for the connections
 - Allocation depends on the actual situation
- Principle
 - Sender requests required buffer amount

- Receiver reserves as many buffers as the actual situation permits
- Receiver returns ACKs and buffer-credits separately
- Sender is blocked when all credits are used up
- Dynamic adjustment to
 - Buffer situation
 - Number of open connections

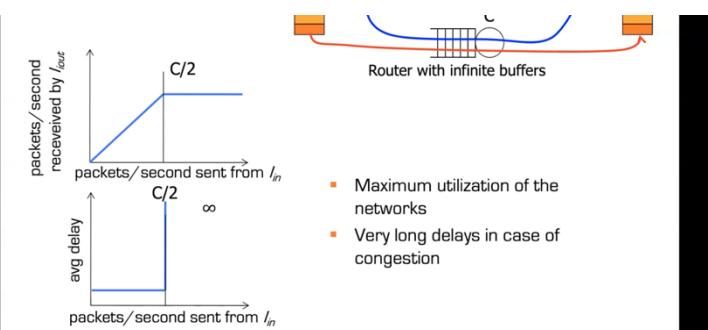
1.12 Datakom: Metningskontroll

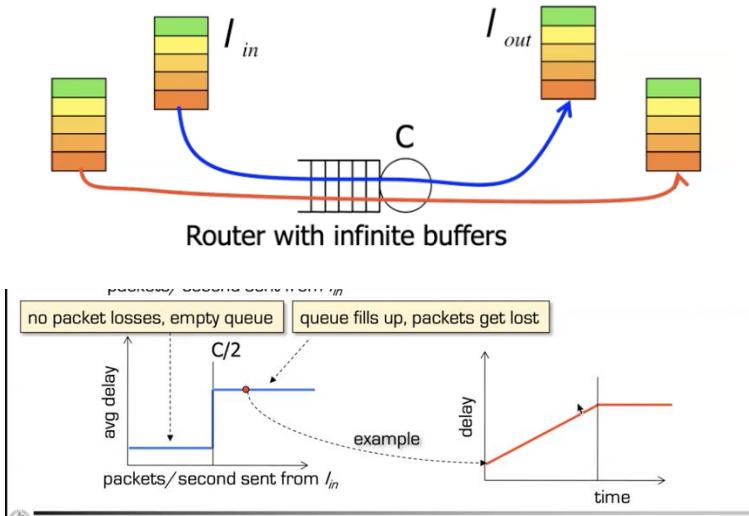
Congestion

- Traffic
 - All traffic from all sources
- Persistent congestion
 - Router stays congested for a long time
 - Excessive traffic offered
- Transient congestion
 - Congestion occurs for a while
 - Router is temporarily overloaded
 - Often due to burstiness

Reasons for congestion

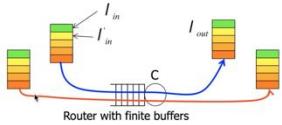
- Two senders, two receivers
- One router, infinite buffer
- No retransmissions





Case 2

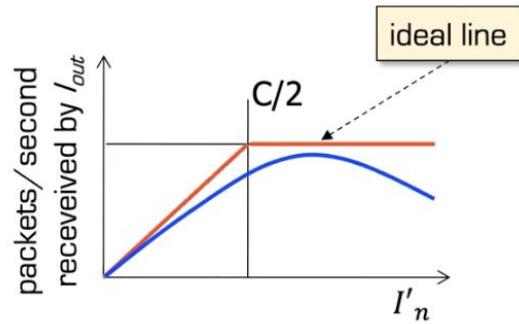
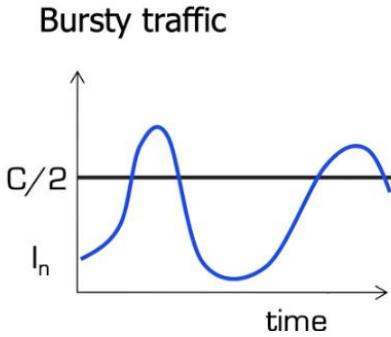
- Two senders, two receivers
- One router, finite buffers
- Retransmission of lost packets



I_{in} Data rate sent by the application

I'_{in} Higher data rate sent by the transport layer including retransmissions

- Retransmission means that I_{out} receives all data that I_{in} sends
- When there is loss, retransmission is happening, and $I_{in} > I_{out}$
- Retransmission of delayed (but not lost) packets increases I_{in} above the perfect value, *without increasing I_{out}*
- “Cost of congestion”:
 - more work (retransmissions) for a desired throughput
 - Useless retransmissions, some links transmit several copies of the same packet



Conclusion:

- Protocols should implement congestion control
- Congestion control algorithms
 - algorithms to adapt the offered load to the capacity at the bottleneck router

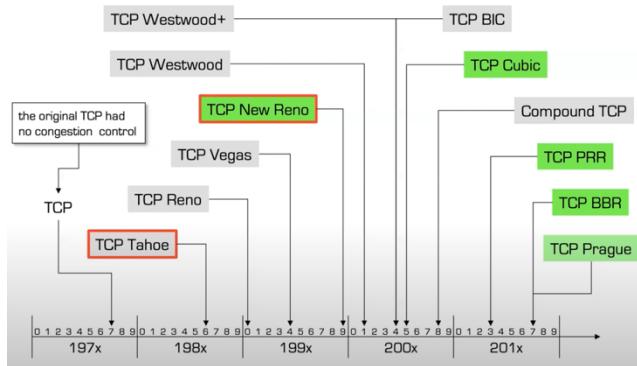
Internet congestion control history:

- around 1986: first congestion collapse
- 1988: “Congestion Avoidance and Control” (Jacobson)
 - Combined congestion/flow control for TCP
- Idea: packet loss = congestion, so throttle the rate; increase otherwise
- Goal: stability - in equilibrium, no packet is sent into the network until an old packet leaves
 - ack clocking “conservation of packets” principle
 - made possible thorough window based stop+go - behaviour
- superposition of stable systems = stable → network based on TCP with congestion control = stable

Congestion Control in TCP

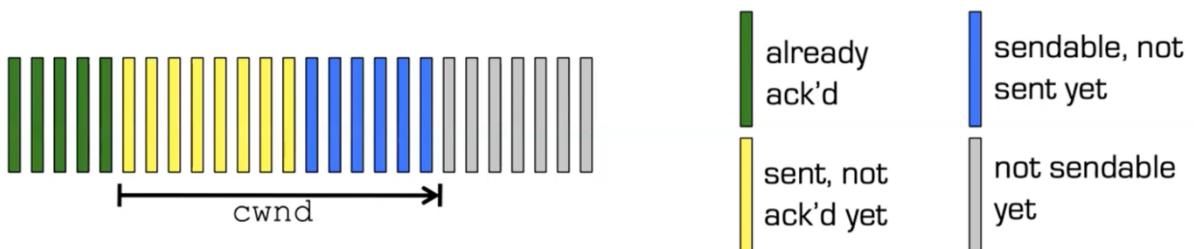
- TCP limits sending rate as a function of perceived network congestion
 - little traffic - increase sending rate
 - much traffic - reduce sending rate
- Congestion algorithm has three major “components”:
 - additive-increase, multiplicative-decrease(AIMD)
 - slow-start
 - reaction to timeout events

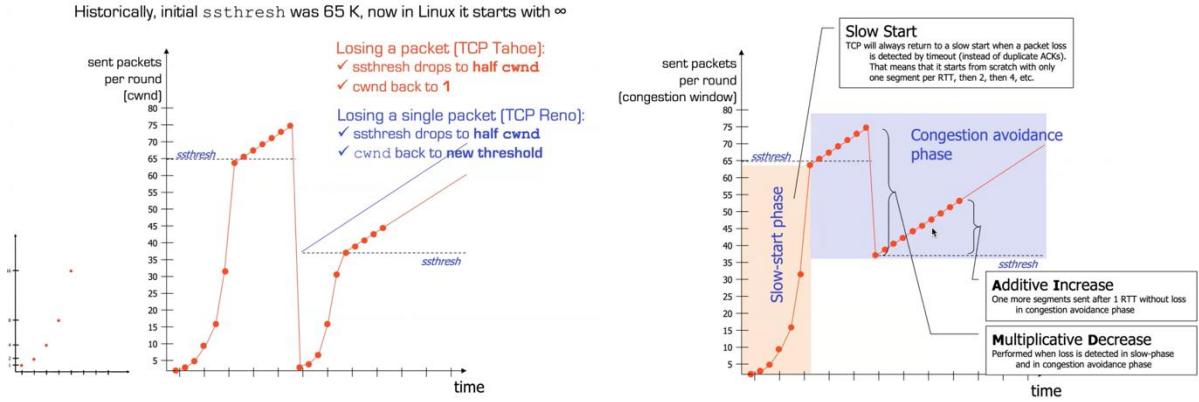
Algorithms



Basic terms:

- *congestion window(cwnd)*
 - largest amount of data (of a connection) that can be in the network at a time
- *Maximum segment size*
 - largest number of bytes that a TCP entity sends at once
 - always in IPv6 usually in IPv6: largest payload that fits into an IP packet behind the TCP header
 - TCP header size is minimal 20 bytes
 - with options maximal 60 bytes
- Testing for available bandwidth
 - ideally: Send as fast as possible without loss
 - Increase cwnd until you have loss
 - If loss, reduce cwnd try increasing again
- End-to-end control (no support from the network layer)
- Send rate is limited by the size of a congestion window, cwnd
- cwnd bytes can be sent in each RTT:
 - $\$ \text{sending rate} = \text{cwnd}/\text{RTT}$
- very often the permitted bytes are sent in $\$ \text{cwnd}/\text{MSS} \$$ packets





TCP Fairness

- Goal of fairness
 - When N TCP streams share a bottleneck, each TCP stream should receive an n^{th} of the bottleneck bandwidth
- More realistic demand
 - When N TCP streams with the same RTT and loss rate share a bottleneck, and they are infinitely long each TCP stream receives an n^{th} of the bottleneck bandwidth
- But the approximation is in many cases good
- How does the AIMD algorithm achieve congestion control?
 - interpret every packet loss as a congestion signal
 - force TCP entities as a congestion signal
 - force TCP entities to slow down when loss is experienced

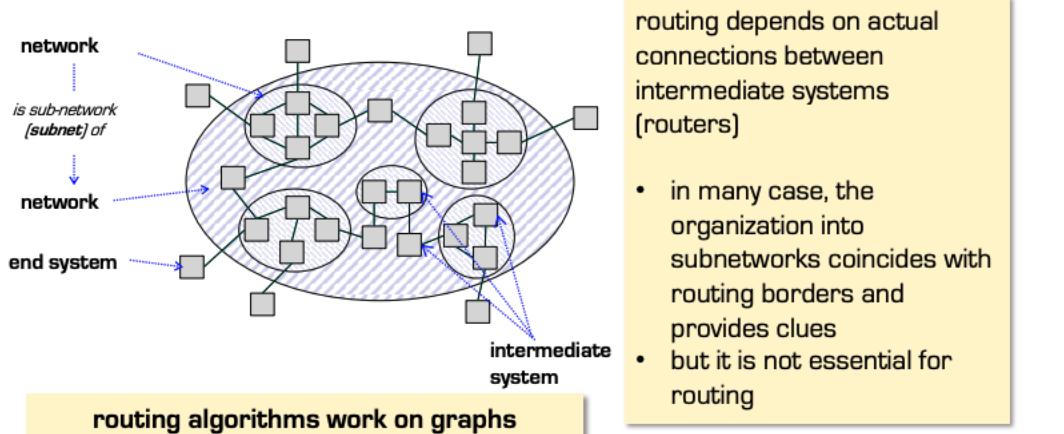
1.13 Datakom: Ruting

Primary task from a layer model perspective

- To provide service to the transport layer
 - Connectionless or connection-oriented service
 - Uniform addressing
 - Internetworking: provide transitions between networks
 - Routing
 - Congestion control
 - Quality of Service (QoS)

The main L3 task is

- Enable data transfer from end system to end system
 - Several hops, subnetworks
 - Compensate for differences between end systems during transmission
- The Intermediate systems are often called Routers



Inside the Network layer

An L3 packet includes

- headers and trailer to specify service requirements
in particular:
- information required by intermediate systems for forwarding

for connection-oriented service:

- route label

OR

for connectionless service:

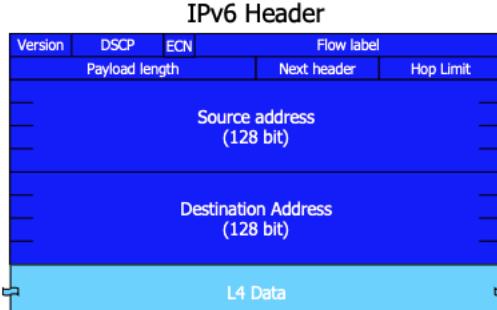
- end system address of the destination

virtual circuits require routing during connection setup

- route label is used later

packet switching requires routing for every packet

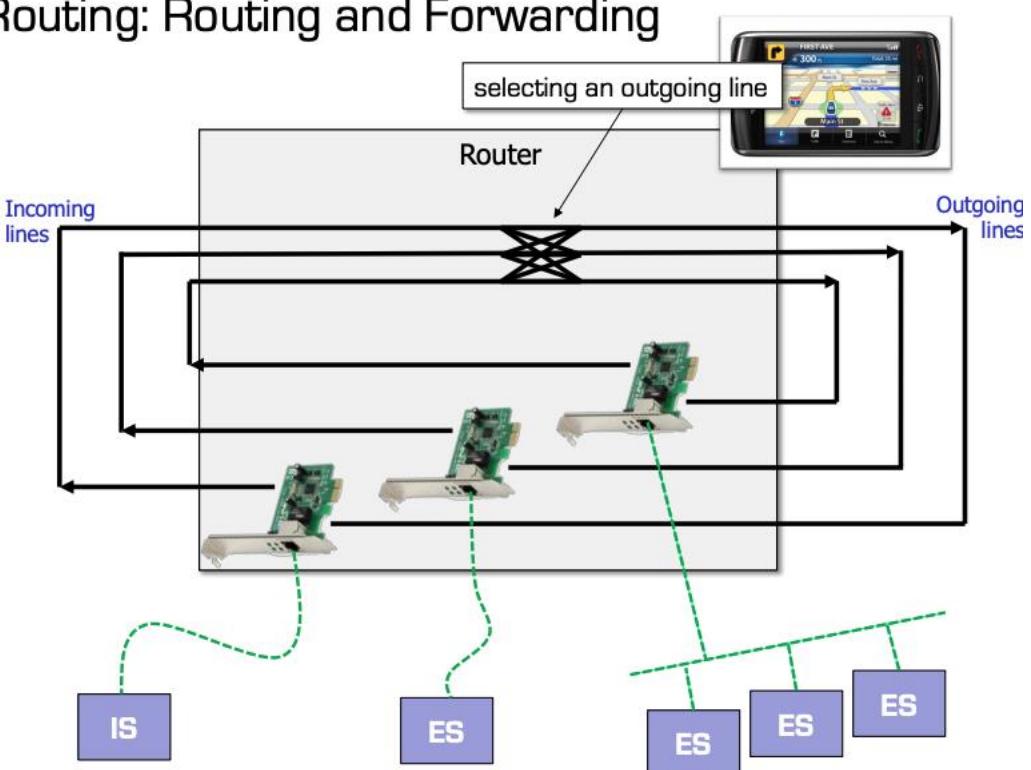
- destination address is used for every packet



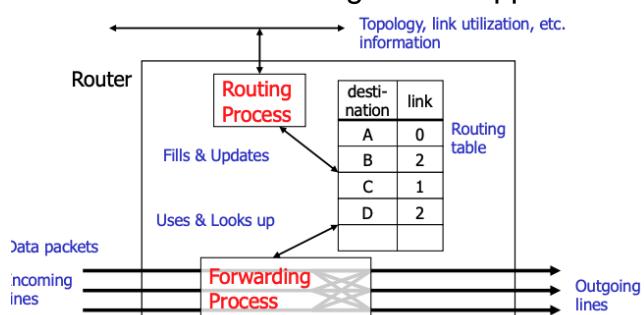
Network layer

- Task
 - TO define the route of packets through the network
 - From the source
 - To the destination system
- Routing algorithm
 - Defines on which outgoing line an incoming packet will be transmitted
- Route determination
 - Packet
 - Routing algorithm makes individual decision for each packet
 - Virtual circuit
 - Routing algorithm runs during connect

Routing: Routing and Forwarding



- Distinction can be made
 - Routing: makes decision which route to use
 - Forwarding: what happens when a packet arrives

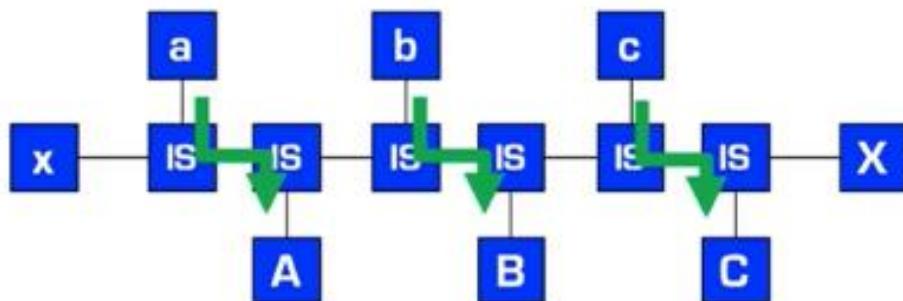


Good properties for Routing Algorithms

- Correctness
- Simplicity
 - Minimize load of ISes
- Robustness
 - Compensation for IS and link failures
 - Handling of topology and traffic changes
- Stability
 - Consistent results
 - No volatile adaptions to new conditions
- Fairness
 - Among different sources compared to each other

Routing Algorithms: Conflicting Properties

- Often conflicting: fairness and optimization
- Some different optimization criteria
 - Average packet delay
 - Total throughput
 - Individual delay
 - Conflict
- Example
 - communication among a -> A, b -> B, c -> C uses full capacity of horizontal line
 - optimized throughput, but
 - no fairness for x -> X - Tradeoff between fairness and optimization



- Therefore often
 - Hop minimization per packet
 - It tends to reduce delays and decrease required bandwidth
 - Also tends to increase throughput

■ Class: Non-adaptive Algorithms

- Current network state not taken into consideration
 - Assume average values
 - All routes are defined off-line before the network is put into operation
 - No change during operation (static routing)
- With knowledge of the overall topology
 - Spanning tree
 - Flow-based routing
- Without knowledge of the overall topology
 - Flooding



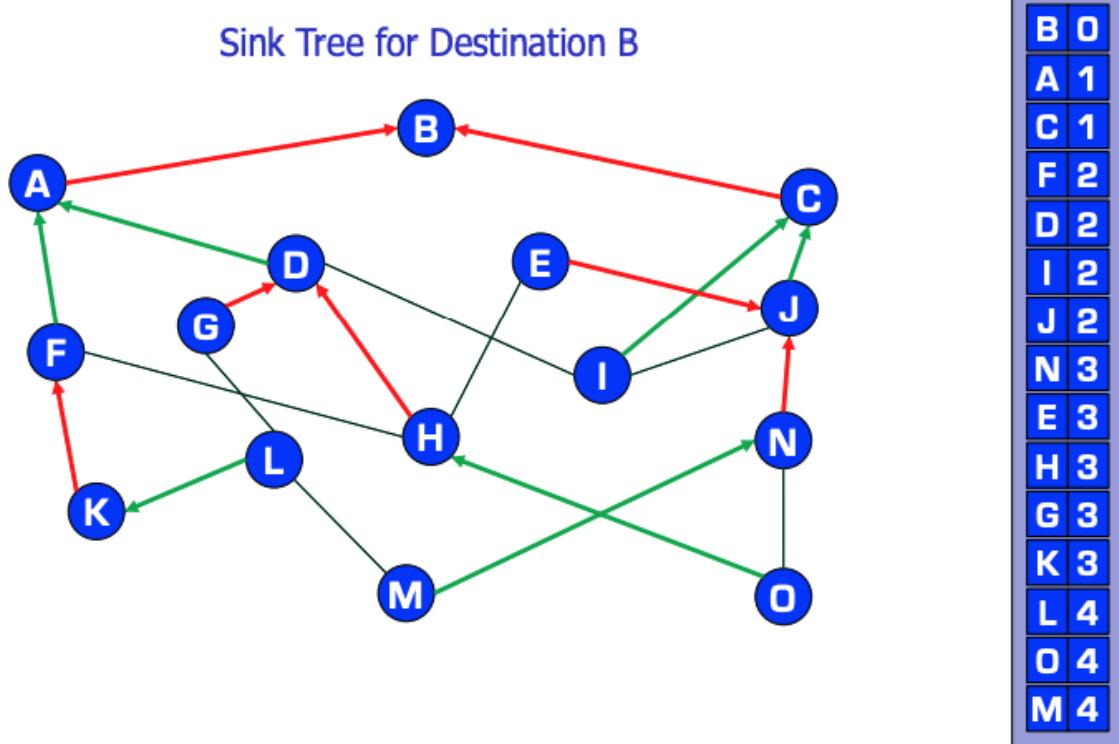
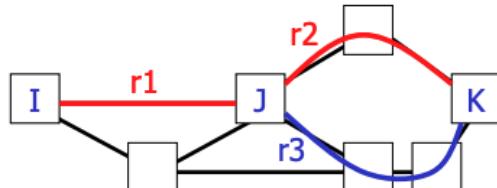
■ Class: Adaptive Algorithms

- Decisions are based on current network state
 - Measurements / estimates of the topology and the traffic volume
- Further sub-classification into
 - Centralized algorithms
 - Isolated algorithms
 - Distributed algorithms



Optimality Principle and Sink Tree

- Starting idea: using a route has a *cost*
 - number of hops, delay, ...
 - General statement about optimal routes
 - if router J is on the optimal path from router I to router K
 - then the optimal path from router J to router K uses the same route
 - Idea of the proof
 - best route from I to K is like this:
 - r1: from I to J, then
 - r2: from J to K
 - then r2 is also the best route from J to K
 - if better route r3 from J to K would exist
 - then concatenation of r1 and r3 would improve route from I to K
 - Set of optimal routes
 - from all sources
 - to a given destination
- form a tree rooted at the destination: ***Sink Tree***



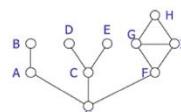
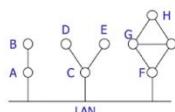
- Comments
 - Tree: no loops
 - Each optimal route is finite with bounded number of hops
 - Not necessarily unique
 - Other trees with same path lengths may exist

- Goal of all routing algorithms
 - Discover and use Sink Trees for all routers
- Not realistic to use Sink Trees as real-life routing algorithm
 - Need complete information about topology
 - Sink Tree is only a benchmark for routing algorithms

Link State Routing

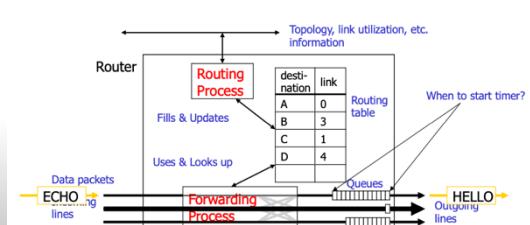
- A very frequently used routing protocol
 - IS-IS (Intermediate System-Intermediate System)
 - OSPF (Open Shortest Path First)
- Basic principles
 - IS measures the “distance” to the directly adjacent IS
 - Distributes information
 - Calculates the ideal route
- Procedure
 - Determine the address of adjacent IS
 - Measure the “distance” to neighbouring IS
 - Organize the local link state information in a packet
 - Distribute the information to all IS
 - Calculate the route based on the information of all IS

1. Phase: gather information about the adjacent intermediate systems



2. Phase: measure the "distance"

- Definition of distance needed
 - Usually delay
 - Where to measure?

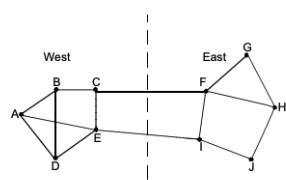


Initialization procedure

- New IS
 - Sends a HELLO message over each L2 channel
- Adjacent IS
 - Responds with its own address, unique within the network

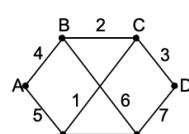
2. Phase: measure the "distance"

- Queuing delay
 - Measuring without does not take load into account
 - Measuring with does ⇒ usually better



3. Phase: organizing the information as link state packet

- Including own address, sequence number, age, "distance"
- Timing problems: validity and time of sending
 - Periodically
 - In case of major changes



Link State Packets:					
A	B	C	D	E	F
Seq.	Seq.	Seq.	Seq.	Seq.	Seq.
Age	Age	Age	Age	Age	Age
B 4	A 4	B 2	C 3	A 5	B 6
E 5	C 2	D 3	F 7	C 1	D 7
	F 6	E 1		F 8	E 8

■ But

- Possibility for oscillations (route flapping)
- Once per routing table update

Link State Routing

4. Distributing the local information to all IS

- By applying the flooding procedure (very robust)
 - Therefore sequence number in packets
- Problem: inconsistency
 - Varying states simultaneously available in the network
 - Indicate and limit the age of packet,
 - i. e. IS removes packets that are too old

5. Computing new routes

- Each IS for itself
- Possibly larger amount of data available

2 Forklaring av konsepter og protokoller

2.1.1 Hva er virtuelt minne?

Virtuelt minne er en teknologi som gjør at en datamaskin kan utvide sin fysiske RAM ved å bruke en del av harddisken som midlertidig lagringsplass for data som ikke umiddelbart trengs. Virtuelt minne gjør det mulig for datamaskiner å kjøre programmer samtidig og håndtere store datamengder.

Virtuelt minne kan overstige den fysiske RAM-en til en datamaskin og bruke andre lagringsressurser også. Vi ønsker å bruke det for å tillate prosesser å bruke en stor mengde minne uten mye innsats fra programmereren. Det krever maskinvarestøtte (MMU) som kan (a) oversette adressebruken til en kjørende prosess fra virtuell adresse til fysisk adresse på en transparent måte, (b) oppdage at en virtuell adresse for øyeblikket ikke har en fysisk adresse (sidefeil), og starte prosedyren for sideutskifting, (c) oppdatere MMU-dataene.

PAGE FAULT dersom man oppdager en virtuell adresse som for øyeblikket ikke har en fysisk adresse.

Fordeler:

1. Økt tilgjengelighet av RAM: Virtuelt minne gjør det mulig å øke mengden av tilgjengelig minne uten å legge til mer fysisk RAM. Dette gjør det mulig å kjøre flere programmer samtidig og håndtere store datamengder uten at det går utover ytelsen
2. Bedre multitasking
3. Mindre krav til fysisk RAM

Ulemper:

1. Lavere ytelse: Fordi det bruker harddisken som midlertidig lagringsplass, er det tregere enn å bruke fysisk RAM
2. Risiko for krasj: Hvis datamaskinen ikke har tilstrekkelig virtuelt minne
3. Økt slitasje på harddisken

Virtuelt minne benyttes fordi det gir maskiner økt kapasitet til å håndtere store datamengder og flere programmer samtidig.

Virtuelt minne er en teknikk som brukes av operativsystemer for å gi hver prosess en illusjon av å ha tilgang til en større mengde minne enn det som faktisk er tilgjengelig fysisk. Det oppnås ved å bruke en kombinasjon av fysisk RAM (Random Access Memory) og diskplass.

Når en prosess kjører, blir dens minnebehov delt opp i mindre enheter som kalles sider. Disse sidene blir plassert i den fysiske RAM hvis de er aktivt i bruk. Men hvis en side ikke er i bruk, kan den flyttes midlertidig fra RAM til en del av disken som kalles pagefilen eller swapområdet.

Når en prosess trenger å aksessere en side som ikke er i RAM, utløses en sidefeil. Operativsystemet tar da ansvar for å hente den manglende siden fra disk og plassere den tilbake i RAM. Dette kalles sideinnlasting eller paging. Denne prosessen skjer transparent for prosessen, som fortsetter å kjøre som om hele det virtuelle minnet er tilgjengelig.

Virtuelt minne gir flere fordeler. Det tillater at flere prosesser kan kjøre samtidig uten at de konkurrerer om begrenset fysisk minne. Det muliggjør også effektiv bruk av minne ved å plassere sjeldent brukte sider på disk og prioritere ofte brukte sider i RAM for raskere tilgang.

2.1.2 Hva er en prosess?

Utførelsen av et program er ofte kalt en prosess. En prosess referer til en serie med instruksjoner eller operasjoner som utføres av en datamaskin eller et datasystem

- En prosess kan opprette en annen prosess ved hjelp av fork systemkallet
 - Det lager et duplikat av kalle-prosessen i tillegg til en kopi av det virtuelle adresserommet.
- En prosess kan utføre et program ved hjelp av execve
- En prosess kan vente på en annen prosess ved hjelp av wait
 - waits until one of the child processes terminates (if there are running processes)

2.1.3 Hva er en tråd?

Tråder er en lettare/enklere form for prosesser. Gjør det mulig for effektivt samarbeid mellom execution units. De deler mange av prosess ressursene. De har deres egen state, stack, prosessor, registere, prosessor register og program counter.

I hovedsak er en prosess en kjørende instans av et program og fungerer som en isolert enhet, mens en tråd er en mindre enhet for utførelse som eksisterer innenfor en prosess og deler ressurser med andre tråder i samme prosess. Tråder brukes ofte for å oppnå parallellisering og forbedre ytelsen til en prosess.

2.1.4 Hva er forskjellen mellom en prosess og en tråd

I sammenheng kan man si at en prosess er et program som kjører uavhengig, mens en tråd er en mindre enhet innenfor en prosess som kan kjøre parallelt med andre tråder innenfor samme prosess. Tråder gir mulighet for mer effektiv parallellisering og deling av ressurser, mens prosesser gir bedre isolasjon og beskyttelse mellom ulike instanser av et program. Når en prosess stanses, så påvirker det ikke de andre prosessene, men hvis en tråd stanser kan det påvirke andre tråder i samme prosess.

2.1.5 Hva er forskjellen mellom en prosess, program og kode?

Kode: Kode er instruksjoner skrevet på et programmeringsspråk som datamaskiner kan tolke og utføre. Kode er grunnlaget for all programvare. For eksempel kan du skrive kode på Python, Java, C++ eller et annet programmeringsspråk for å instruere datamaskinen om hva den skal gjøre.

Program: Når koden kompileres eller tolkes, blir den et program eller programvare. Et program er en samling av instruksjoner som utfører en spesifik oppgave når de kjøres på en datamaskin. For eksempel, Microsoft Word, Google Chrome, og operativsystemet på datamaskinen din er alle programmer.

Prosess: En prosess er en kjørende instans av et program. Når du åpner et program, blir det lastet inn i datamaskinenes minne og begynner å kjøre. Hver prosess har sin egen dedikerte minneplass og kjører uavhengig av andre prosesser, med mindre de eksplisitt kommuniserer med hverandre. For eksempel, hvis du åpner to instanser av Google Chrome, har du to prosesser kjørende på datamaskinen din.

For å oppsummere, kode er det vi skriver for å instruere datamaskiner om hva de skal gjøre. Når denne koden er kompilert eller tolket, blir den et program som kan kjøres på en datamaskin. Og når et program kjører på en datamaskin, blir det en prosess.

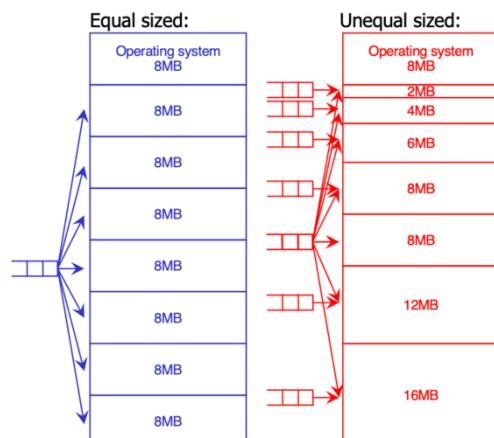
2.1.6 Hva er paging?

- Paging innenfor data refererer vanligvis til en teknikk for å organisere og behandle store datasett eller databaser. I dette tilfellet, betyr paging å dele opp datasettet i mindre enheter, kalt sider, som kan behandles uavhengig av hverandre.
- Paging brukes ofte i situasjoner der hele datasettet ikke kan passe i den tilgjengelige hukommelsen på en datamaskin eller annen enhet. I stedet for å laste inn hele datasettet i hukommelsen, laster systemet inn bare en del av datasettet, eller side, på en gang. Dette reduserer mengden minne som kreves for å behandle datasettet og gjør det mulig å arbeide med større datasett enn ellers ville være mulig.
- Paging i datasystemer bruker vanligvis en lignende tilnærming som paging i minnehåndteringssystemer, der det er en mekanisme for å hente og lagre data på en effektiv måte. Det kan også innebære caching av data for å redusere behovet for å hente data fra sekundær lagring, for eksempel en database på en harddisk. Ved å organisere data på denne måten, kan paging bidra til å forbedre ytelsen og skalerebarheten til datasystemer.

2.1.7 Ulike typer minnelayout

2.1.7.1 Fixed partition

- Deler minnet opp i statiske partisjoner når systemet starter opp
 - Fordelene er at det er lett å implementere og at det støtter bytting av prosesser



Tenk deg at du har en stor tavle (dette er RAM i datamaskinen). Du deler tavlen i flere faste deler, eller "partisjoner". Størrelsen på disse partisjonene er bestemt på forhånd og forblir den samme. La oss si at du har delt tavlen i fire like deler.

Partisjon 1, Partisjon 2, Partisjon 3, Partisjon 4

Når et program skal kjøres, plasserer operativsystemet programmet i en av disse partisjonene. Hvis programmet er for stort til å passe i en partisjon, vil det ikke kunne kjøre. La oss si at vi har tre programmer, A, B, og C. Program A og B kan passe inn i en partisjon hver, mens program C er for stort og krever to partisjoner for å kjøre. Operativsystemet plasserer A i partisjon 1 og B i partisjon 2. Imidlertid kan ikke C kjøre fordi det ikke er noen tilstøtende partisjoner ledige. Dette problemet kalles "ekstern fragmentering".

En annen ulempe med fast partitioning er at det kan være en del av partisjonen som ikke brukes. Hvis for eksempel program A kun bruker halvparten av partisjon 1, vil den andre halvdelen være ubrukt. Dette problemet kalles "intern fragmentering".

2.1.7.2 *Dynamic partitioning*

- Deler opp minnet mens det kjører
 - Partisjoner opprettes dynamisk
 - Blir fjernet etter jobber er gjort
- Compaction
 - Removes fragments by moving data in memory
 - Takes time
 - Consumes processing resources

Vi har igjen tre programmer A, B, og C, men denne gangen vil vi plassere dem i minnet dynamisk.

1. Program A trenger plass og operativsystemet allokerer akkurat nok plass på tavlen til A. La oss si at A fyller 1/4 av tavlen.
2. Deretter kommer program B og trenger plass. Operativsystemet allokerer plass for B rett ved siden av A. La oss si at B også tar opp 1/4 av tavlen.
3. Så kommer program C, som er ganske stort og trenger halvparten av tavlen. Operativsystemet finner ledig plass og plasserer C i det resterende minnet.

I dette scenariet kan alle programmene kjøre fordi de får plass etter behov. I motsetning til fixed partitioning, er det ingen intern fragmentering fordi hver partisjon er akkurat stor nok til å inneholde programmet den inneholder.

Men hva skjer når program A er ferdig og frigjør sin plass? Da har vi et hull i minnet. Hvis et nytt program D kommer inn og det er større enn plassen A etterlot seg, men mindre enn total ledig plass (plassen etter A og noen annen ledig plass), vil det ikke

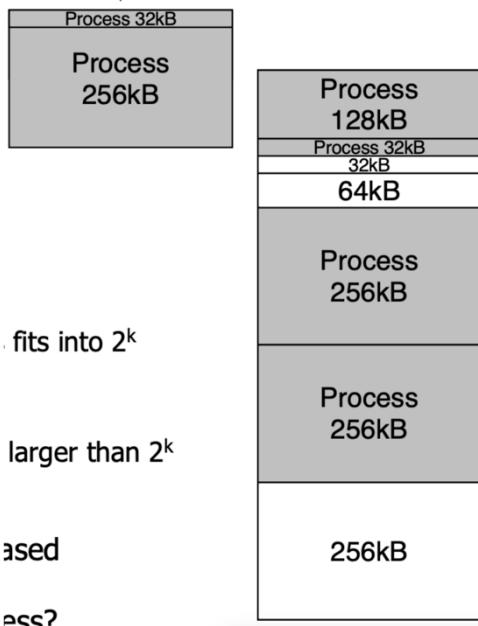
kunne plasseres. Dette problemet er kjent som "ekstern fragmentering".

Kompaktering kan være en løsning på dette problemet, som innebærer å flytte innholdet i minnet slik at all ledig plass samles i en stor blokk.

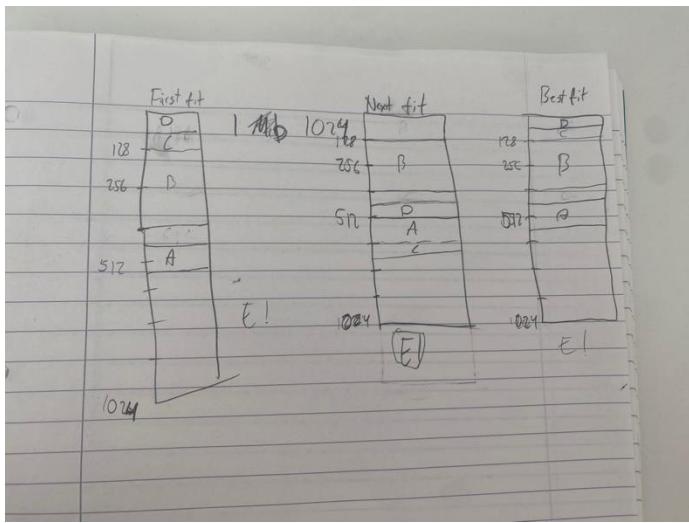
Dynamic partitioning er mer fleksibel enn fast partitioning og kan håndtere en rekke forskjellige størrelser på programmer, men det har også sine ulemper, som behovet for mer kompleks minnehåndtering og potensialet for ekstern fragmentering.

2.1.7.3 Buddysystem

- Mix of fixed and dynamic partitioning
- Maintain a list of holes with sizes
- Assigning memory to a process:
 - find the smallest k so that the process fits into 2^k
 - find a hole of size 2^k
 - if not available, split the smallest hole larger than 2^k recursively into halves
- Merge partitions if possible when released



2.1.8 Plasseringsalgoritmer



2.1.8.1 First fit

Forestil deg en lang gate (som representerer datamaskinenes minne) med flere ledige parkeringsplasser av varierende størrelse (som representerer ledig minne).

Nå kommer en bil (som representerer et program) som trenger en parkeringsplass. First Fit-algoritmen vil kjøre langs gaten og parkere bilen på den første ledige plassen den finner som er stor nok til å romme bilen. Hvis plassen er større enn bilen, blir resten av plassen stående tom (det vil være intern fragmentering).

Det er en enkel og rask algoritme, men den kan ende opp med å etterlate store ledige områder bak de små bilene, noe som kan gjøre det vanskeligere å finne plass til større biler (programmer) senere.

2.1.8.2 Next fit

Next Fit-algoritmen er ganske lik First Fit, men med en liten forskjell: stedet hvor den sist parkerte bilen (programmet) ble plassert, blir husket.

Så, når en ny bil kommer inn for å finne en parkeringsplass, starter Next Fit sin søk fra stedet hvor den sist parkerte bilen ble plassert, ikke fra begynnelsen av gaten. For eksempel, hvis den forrige bilen ble parkert på midten av gaten, starter Next Fit derfra og fortsetter til slutten. Hvis den ikke finner en passende plass, fortsetter den fra begynnelsen av gaten til den når det stedet den startet fra.

På denne måten fordeler Next Fit bruken av minne mer jevnt over hele gaten (minnet), men det kan fortsatt oppstå fragmentering, akkurat som med First Fit.

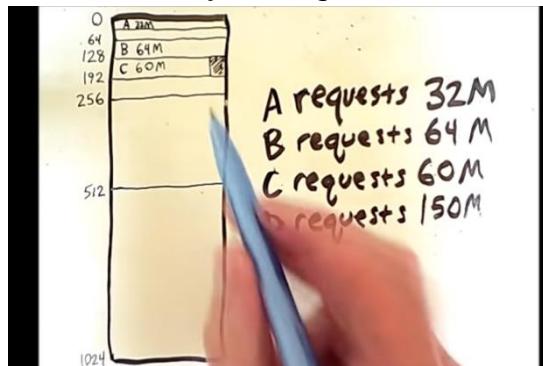
2.1.8.3 Best fit

Best Fit-algoritmen ser etter den ledige parkeringsplassen som best passer størrelsen på bilen, med minst mulig ledig plass til overs. Med andre ord, den prøver å minimere intern fragmentering ved å finne det tetteste mulige stedet.

For eksempel, hvis det finnes to ledige parkeringsplasser, en som er litt større enn bilen og en som er mye større, vil Best Fit velge den litt større plassen, for å spare den store plassen til en større bil senere.

Mens denne metoden kan virke ideell for å minimere intern fragmentering, har den ulempen at den kan skape mange små ubrukte minneblokker over tid, noe som kan gjøre det vanskelig å finne plass til større programmer senere. Dette er også kjent som ekstern fragmentering. Dessuten kan Best Fit være treg, siden den må søke gjennom hele minnet for å finne den "beste" passende plassen.

2.1.8.4 Buddytildeling



Vi halverer minnet til vi finner en partisjon med passende størrelse.

I stedet for å ha en lang gate med parkeringsplasser av varierende størrelse, forestill deg en gate hvor alle parkeringsplassene er av en fast størrelse, for eksempel 8 meter. Disse er dine "buddies".

Hvis en bil (program) kommer inn og trenger 8 meter, kan den bare bruke en av disse parkeringsplassene. Men hva om bilen bare trenger 2 meter? I stedet for å kaste bort 6 meter av plassen, deler Buddy System denne 8 meter parkeringsplassen i to "buddies" på 4 meter hver. Hvis 4 meter fortsatt er for stort, kan en av disse 4-meters "buddies" deles på nytt til to 2-meters "buddies".

På denne måten, Buddy System prøver å minimere intern fragmentering ved å alltid dele minne i partisjoner som er så nære som mulig til størrelsen som trengs av programmet.

Når et program er ferdig og frigir sin "buddy", hvis det finnes en annen ledig "buddy" av samme størrelse ved siden av, vil disse to "buddies" kombinieres tilbake til en større "buddy". Dette hjelper med å bekjempe ekstern fragmentering.

Buddy System er effektiv for å håndtere anmodninger av forskjellige størrelser, og det er relativt raskt å finne en passende "buddy" for et program. Men det har også

ulempen at det kan føre til intern fragmentering hvis programmet ikke bruker hele sin tildelte "buddy".

2.1.9 Hva er en nettmaske?

En nettmaske, også kjent som subnet mask på engelsk, er en tallrekke som brukes sammen med en IP-adresse for å definere nettverksstrukturen. Den identifiserer hvilken del av IP-adressen som tilhører nettverket og hvilken del som tilhører enhetene (vertene) i nettverket.

En nettmaske består av en sekvens med binære tall som er like lang som IP-adressen. Den består vanligvis av en sammenhengende rekke av 1-biter etterfulgt av en sammenhengende rekke av 0-biter.

Et eksempel på en nettmaske er 255.255.255.0. Denne nettmasken brukes ofte i IPv4-nettverk for å opprette et subnett med et Class C IP-adresseområde. Ved å bruke denne nettmasken kan de første tre oktettene av IP-adressen identifisere nettverket, mens den siste oktetten kan brukes til å adressere de individuelle enhetene (vertene) i nettverket.

For eksempel, hvis en enhet har IP-adressen 192.168.1.50 og nettmasken 255.255.255.0, indikerer nettmasken at de tre første oktettene (192.168.1) identifiserer nettverket, mens den siste oktetten (50) identifiserer den spesifikke enheten i nettverket.

Ved å bruke nettmasker kan man dele IP-adresseområder inn i mindre subnett og effektivt administrere nettverksressurser. Nettmasken brukes av rutere og enheter i nettverket for å bestemme hvilke datapakker som skal rutes til riktig destinasjon innenfor det lokale nettverket eller til eksterne nettverk.

2.1.10 Hva er Multilevel Paging?

Multilevel paging, også kjent som hierarkisk paging, er en teknikk innenfor virtuell minnehåndtering som brukes i moderne operativsystemer for å håndtere store virtuelle adresserom mer effektivt. Det er en utvidelse av vanlig paging, som deler den fysiske hukommelsen i mindre sider (f.eks. 4 KB hver) og den virtuelle adressen i virtuelle sider av samme størrelse.

I et multilevel paging-system organiseres de virtuelle sidene og de fysiske sidene i et hierarki av tabeller. Hver tabell kalles en sidekarttabell, og hvert nivå av tabeller representerer en del av den totale virtuelle adressen.

La oss ta et eksempel for å illustrere hvordan multilevel paging fungerer. La oss si at den virtuelle adressen er på 32-bits, og hver side er 4 KB (12 bits).

Det første nivået av tabellen, kalt den ytre sidekarttabellen, inneholder oppføringer for å peke til den neste nivået av tabeller. La oss si at hver oppføring i den ytre sidekarttabellen peker til en midt-nivå sidekartabell.

Den midtre sidekarttabellen inneholder oppføringer som peker til den indre sidekarttabellen. Hver oppføring i den midtre sidekarttabellen peker på en side med indre sidekarttabell.

Den indre sidekarttabellen inneholder endelig oppføringer som peker direkte til de fysiske sidene i minnet. Hver oppføring i den indre sidekarttabellen peker på en faktisk side i den fysiske hukommelsen.

Ved å bruke dette hierarkiet av tabeller kan operativsystemet raskt finne den riktige fysiske adressen for en gitt virtuell adresse. Den virtuelle adressen blir brutt ned i ulike deler, og hver del brukes til å indeksere de forskjellige nivåene av sidekarttabellene. Denne indekseringen hjelper til med å begrense søket i tabellene og effektivt finne riktig oversettelse fra virtuell til fysisk adresse.

Multilevel paging har flere fordeler. Først muliggjør det effektiv håndtering av store virtuelle adresserom ved å dele opp tabellene i mindre deler. Dette reduserer behovet for å holde hele sidekarttabellen i minnet samtidig. Videre reduserer det også fragmentering av det fysiske minnet, siden sider kan plasseres spredt rundt uten behov for å være sammenhengende. Dette gir fleksibilitet og optimal utnyttelse av den fysiske hukommelsen.

Samlet sett bidrar multilevel paging til en mer effektiv og skalerbar administrasjon av virtuell minnehåndtering i moderne operativsystemer.

Page cache er en mellomlagringsplass for nylig brukte sider, noe som gir raskere tilgang til sider som allerede er lastet inn i RAM. Når en side ikke er tilgjengelig i RAM, blir den lastet inn fra disk til RAM for å sikre at den er tilgjengelig for prosessen.

Dirty Pages: Når en side blir endret i RAM, blir den merket som "dirty". Dette sikrer at endringene blir riktig lagret til disk.

2.1.11 Hva er forskjellen på ekstern og intern fragmentering?



Intern fragmentering er innad i minneblokken, mens ekstern fragmentering er hull mellom de eksisterende minneblokkene.

Intern fragmentering oppstår når det er ubrukt plass innenfor en enkelt allokeringsenhet, for eksempel en minneblokk eller en disksektor. Dette kan skje når objekter eller filer ikke fyller helt opp den tildelte plassen. Den ubrukten plassen blir ikke utnyttet effektivt og kan føre til ressurssløsing.

Ekstern fragmentering oppstår når det er spredte, små ledige områder mellom allokeringseenheter. Dette kan skje over tid når objekter eller filer blir tildelt og frigjort i minnet eller på disken. Den totale mengden ledig plass kan være tilstrekkelig, men det er ikke nok sammenhengende plass til å tilfredsstille større allokingsforespørslar. Dette kan føre til ineffektiv bruk av ressurser og kan kreve en defragmenteringsprosess for å samle ledig plass til større blokker.

2.1.12 Hva er CPU skedulering?

CPU skedulering bestemmer hvilke tasks som de skal prioritere og i hvilken rekkefølge ting skal skje.

Mål for skedulering:

- Kernel
 - Resource management
 - processor utilization, throughput, fairness
- User
 - Interactivity
 - response time (Example: when playing a game, we will not accept waiting 10s each time we use the joystick)
 - Predictability
 - identical performance every time (Example: when using the editor, we will not accept waiting 5s one time and 5ms another time to get echo)
- “Most reasonable” criteria depend on target system
 - Most/All types of systems
 - fairness – giving each process a fair share
 - balance – keeping all parts of the system busy
 - Batch systems
 - turnaround time – minimize time between submission and termination
 - throughput – maximize number of jobs per hour
 - (CPU utilization – keep CPU busy all the time)
 - Interactive systems
 - response time – respond to requests quickly
 - proportionality – meet users’ expectations
- Real-time systems
 - meet deadlines – avoid loosing data
 - predictability – avoid quality degradation in multimedia systems

2.1.13 Ulike typer skedulering

2.1.13.1 FIFO

- First in, First out
- Prioriterer det som kommer inn først, først. Dette gjøres uavhengig av antatt kjøretid osv.
- Fordelen med FIFO er: rettferdig mtp. At den er basert på ankommstid, gir en forutsigbar skedulering, enkelt å implementere

2.1.13.2 Round Robin

- Hver prosess blir tildelt et fast tidskvantum, også kjent som en time quantum. Prosessene blir utført i en sirkulær rekkefølge, der hver prosess får en sjanse til å kjøre for sin tildelte time-quantum før den blir pre-empted og gir CPU-en til neste prosess i køen.
- Hvis en prosess fullfører utførelsen innenfor sin tildelte time-quantum, blir den fjernet fra køen. Hvis imidlertid en prosess ikke fullfører utførelsen innenfor sin tildelte tid, blir den satt bakerst i køen igjen slik at andre prosesser kan fullføre
- Round Robin-skeduleringsalgoritmen sikrer at hver prosess får en lik andel av CPU-tiden, og forhindrer at noen enkeltprosesser monopoliserer CPU-en. Det sikrer også at prosesser blir utført på en rettferdig og forutsigbar måte, der hver prosess får en sjanse til å utføre innenfor en rimelig tidsramme.

2.1.13.3 SJF

- Shortest Job First
- Velger tasks med kortest prosesseringstid
- Fordelen med Shortest Job first er at den er simpel og minimerer den gjennomsnittlig ferdigstillelsestid for oppgavene.

2.1.13.4 EDF

Earliest Deadline First

- Den er basert på å gi høyere prioritet til oppgaver med kortere tidsfrister
- I EDF-skaleringsmetoden blir hver oppgave tildelt en absolutt tidsfrist, og skeduleringsalgoritmen velger alltid den oppgaven med kortest tidsfrist som skal utføres neste. Hvis to oppgaver har samme tidsfrist, blir oppgaven med høyest prioritet utført først.
- Når en oppgave kommer til CPU-en for utførelse, kjøres den til den er fullført, eller til en annen oppgave med kortere tidsfrist ankommer, og dermed blir den utført først. Hvis en oppgave ikke fullføres innen tidsfristen, anses den som mislykket eller ikke oppfylt, og skeduleringsalgoritmen tar umiddelbart opp neste oppgave med kortest tidsfrist.

- EDF-skaleringsmetoden er viktig i sanntids systemer der det er viktig å oppfylle tidsfrister for oppgaver. Det er effektivt i å minimere gjennomsnittlig ventetid og gir en høy grad av forutsigbarhet og pålitelighet i skeduleringsprosessen.

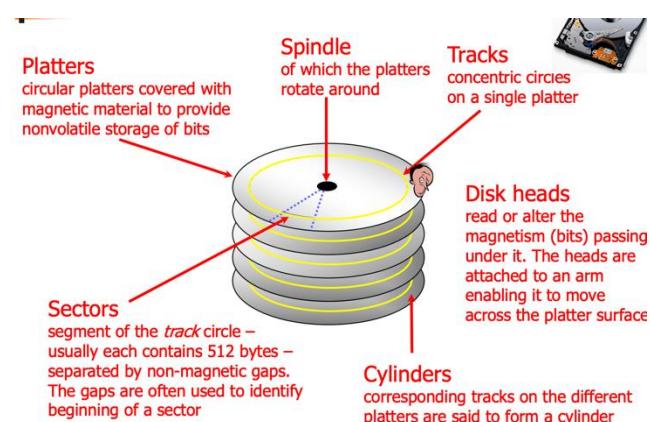
2.1.14 Hvilken skeduleringsalgoritme er best for hva?

RR algoritmen gir hver oppgave en fast tidskvote (eller "timeslice") og roterer gjennom alle oppgavene. Fordi RR gir alle prosesser en sjanse til å kjøre regelmessig, kan det gi god responsitet for interaktive systemer, siden ingen prosess må vente lenge før den får kjøre.

First-Come, First-Served (FCFS): Dette er den enkleste algoritmen, og den kan fungere godt hvis jobbene er omtrent like lange. Men hvis det er store forskjeller i jobblengder, kan kortere jobber ende opp med å vente lenge på lengre jobber (dette kalles "convoy effect").

2.1.15 Hva er disken?

Disk er sekundærlagring, billigere enn hovedminne og har mer kapasitet.



Diskens kapasitet avhenger av

- antall platters
- om platterne bruker en eller begge sider
- antall tracks per surface

- gjennomsnittlig antall sektor per track
- antall bytes per sektor

Diskens access time avhenger av

- Seek time
 - tid det tar for å få riktig posisjon på hodet
 - omrent proporsjonel til antall sylinder den har beveget seg over

$$\alpha + \beta\sqrt{n}$$

number of tracks
 seek time constant
 fixed overhead

- Rotational delay
 - tid det tar for disk platters å rottere sånn at den første av de nødvendige sectorene er under diskhodet.
- Transfer time
 - Tid det tar at data blir lest av hodet, mao., tid det tar at sektorene fra den ønskede blocked roterer under hodet.

■ Transfer rate = $\frac{\text{amount of data per track}}{\text{time per rotation}}$

■ Transfer time = $\frac{\text{amount of data to read}}{\text{transfer rate}} = \frac{\text{amount of data to read} * \text{time per rotation}}{\text{amount of data per track}}$

- Other delays
 - CPU time to issue and process I/O
 - contention for controller, bus, memory
 - verifying block correctness with checksums (retransmissions)
 - waiting in scheduling queue

Disk kontrollere

- Kontrollerer **aktuatoren** beveger hodet til den ønskede tracken
- velger hvilket hode (platter og surface) å bruke
- Vite når den riktige sektoren er under hodet
- Overføre data mellom hovedminnet og disken

2.1.16 Disk skedulering

Søketid er hovedfaktoren til den totale diskens IO tid. Disk skedulering \neq CPU skedulering.

Disk scheduling handler om å organisere og prioritere tilgangen til diskoperasjoner for å minimere søketid og øke effektiviteten ved å velge den mest hensiktsmessige rekkefølgen for lesing eller skriving av data til og fra disken.

CPU-scheduling handler derimot om å bestemme rekkefølgen der prosessene blir utført på CPUen, og tildeler CPU-tid til hver prosess for å oppnå rettferdig ressursutnyttelse, responsivitet og optimal gjennomstrømning.

Mål:

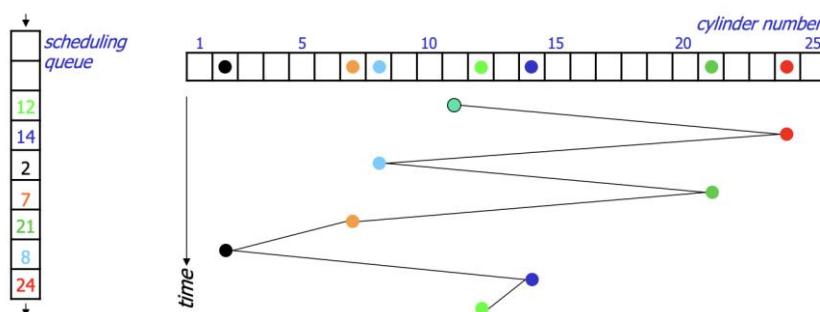
- kort respons tid
- høy total gjennomstrømning
- Rettferdighet, i konteksten av diskskedulering, refererer til prinsippet om at alle disk I/O-anmodninger skal behandles på en jevn og balansert måte, slik at ingen prosess konsekvent favoriseres over en annen, og ingen prosess blir ignorert eller blir sittende fast ubestemt.

Ulike algoritmer:

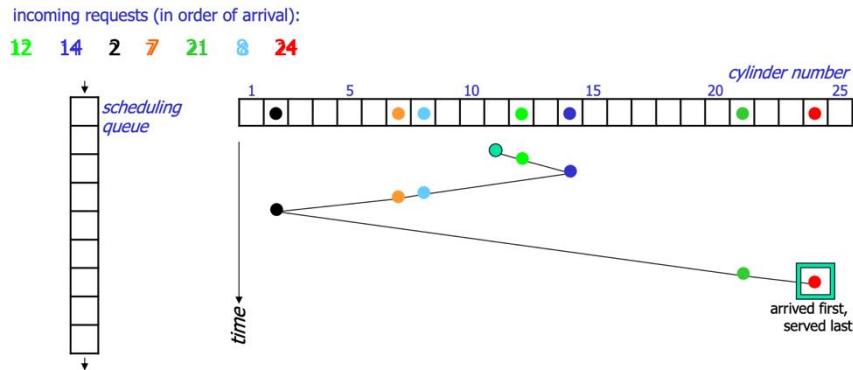
- First-Come-First-Serve - FCFS
 - Håndterer den forespørslen som kom kronologisk først.

incoming requests (in order of arrival, denoted by cylinder number):

12 14 2 7 21 8 24

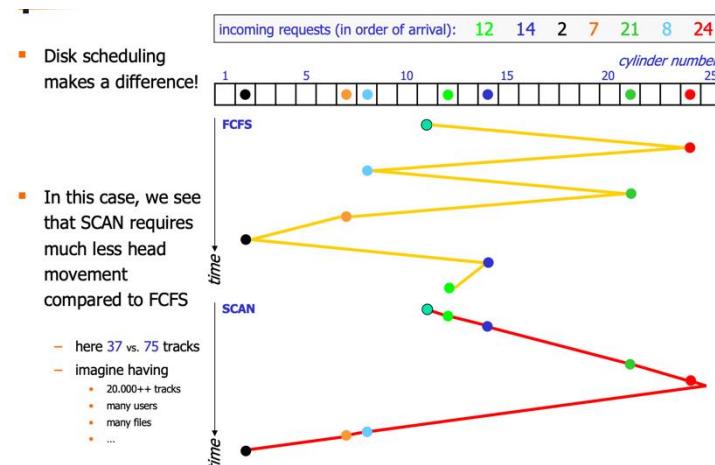


- Shortest Seek Time First (SSTF)
 - Håndterer nærmeste forespørsel først
 - korte seek times
 - lengre max respons tid - may even lead to starvation



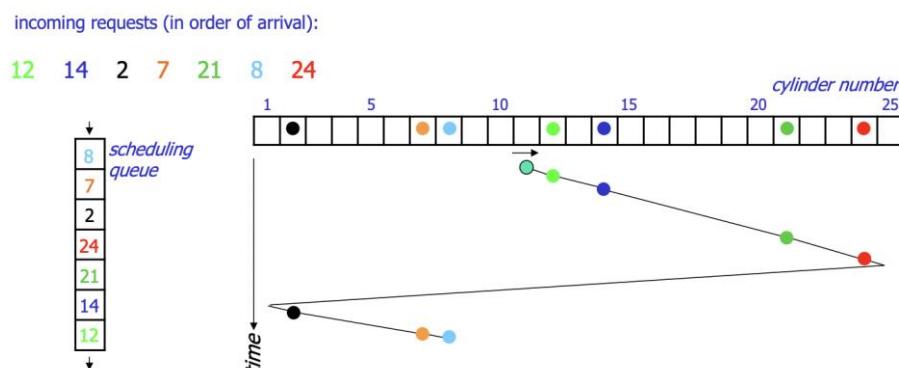
- SCAN

- Beveger hode fra side til side og håndterer forespørsler på veien:
 - bi-directional
 - compromise between response time and seek time optimizations

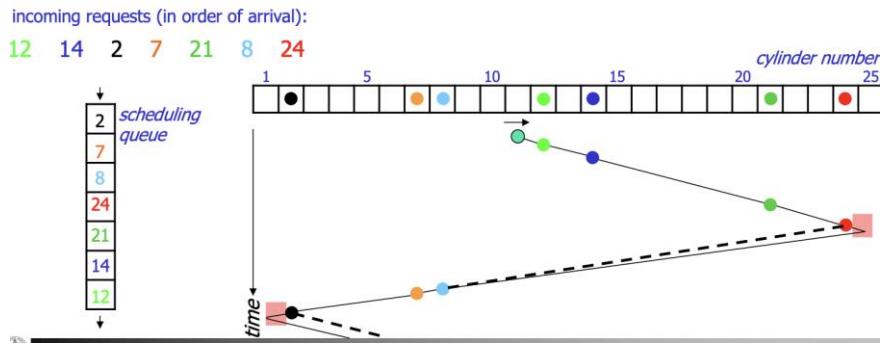


- C-SCAN

- Beveger hode fra ende til ende
- Optimisering av SCAN
- Håndter bare forespørsler en vei - uni-directional



- Look
 - Variasjon av SCAN
 - samme schedule som scan
 - kjører ikke fra ende til ende
 - kjører fra de ytterste og innerste forespørslene



2.1.17 Hva er multicore?

1 queue

- Potensiell bottleneck
 - locking/contention on the single queue

Multiple queues

- Potensiell bottleneck?
 - load balancing
 - Linux checks every 200 ms

Kombineres ofte sammen med virtuelle maskiner. Og når man bruker flere kjerner med maskiner som bruker samme minne, kan man se for seg at duplikasjon av data kan bli et problem. Men **deduplication** hindrer dette. Vi har transparent page sharing og content-based page sharing. Generelt handler teknologien om å skanne minnet til hver virtuelle maskin på en host og hashe memory pagene. Should some pages produce an identical hash, the system has to first check to see if they really are the same, and if so, deduplicate them, creating one page with the actual content and two references to that page. Since the hypervisor controls the nested (or shadow) page tables, this mapping is straightforward. Of course, when either of the guests modifies a shared page, the change should not be visible in the other virtual machine(s)

2.1.18 Hva er multiprogrammering?

Det er en teknikk som brukes i datasystemer for å muliggjøre samtidig kjøring av flere programmer. I et multiprogrammeringsmiljø administrerer datamaskinen

operativsystem ressursene (slik som CPU-tid, minne og inngangs-/utgangsenheter) slik at flere programmer kan kjøre samtidig.

Operativsystemet tildeler en del av CPU-tiden til hvert program, slik at de kan gjøre fremskritt og dele datamaskinen ressurser. Dette gjør det mulig for brukerne å kjøre flere programmer samtidig, uten å måtte vente på at ett program skal fullføre før de starter et annet.

Multiprogrammering brukes vanligvis på store datasystemer, der det er vanlig at mange brukere kjører forskjellige programmer samtidig. Det er en viktig teknikk som brukes i moderne operativsystemer for å muliggjøre effektiv bruk av datamaskinen ressurser og for å gi en responsiv og interaktiv brukeropplevelse.

2.1.19 Hva er packet switching?

Packet switching er en kommunikasjonsmetode som brukes i datanettverk for å effektivt overføre data mellom avsender og mottaker. I stedet for å sende kontinuerlige strømmer av data, deles informasjonen opp i mindre enheter kalt pakker. Hver pakke inneholder en del av dataene, sammen med nødvendig informasjon som kildeadresse, destinasjonsadresse og kontrollinformasjon.

I et pakkesvitsjet nettverk blir de individuelle pakkene rutingen uavhengig av hverandre. De sendes individuelt over nettverket og kan følge forskjellige ruter til destinasjonen. Dette står i kontrast til kretssvitsjing, der en dedikert kommunikasjonskanal opprettes og forblir åpen under hele overføringen.

Fordeler med pakkesvitsjing inkluderer:

- Effektiv utnyttelse av nettverksressurser: Ved å dele informasjonen i mindre pakker kan nettverket utnytte ledig båndbredde mer effektivt. Pakker kan sendes over forskjellige ruter og utnytte multipleksing og deling av ressurser.
- Fleksibilitet og skalerbarhet: Pakkesvitsjing muliggjør en mer fleksibel nettverksinfrastruktur. Nye ruter og koblinger kan enkelt legges til nettverket uten å påvirke eksisterende kommunikasjon. Dette gjør det også enklere å skalere nettverket for å håndtere økt trafikk.
- Feiltoleranse: Hvis en rute eller en node i nettverket feiler, kan pakkene rutes rundt problemområdet ved å velge alternative ruter. Dette bidrar til å opprettholde kontinuerlig kommunikasjon selv om det oppstår feil.
- Støtte for forskjellige datatyper: Pakkesvitsjing støtter ulike typer datatrafikk, inkludert tale, video og data. Hver pakke behandles individuelt og kan bli prioritert etter behov.

Pakkesvitsjing er den grunnleggende teknologien som brukes i Internett og de fleste moderne datanettverk. Det muliggjør effektiv og pålitelig kommunikasjon mellom avsender og mottaker ved å dele informasjonen i mindre pakker og rute dem over nettverket på en adaptiv og effektiv måte.

2.1.20 Hva er Circuit switching?

Circuit switching er en kommunikasjonsmetode som brukes i telekommunikasjonsnettverk for å opprette en dedikert kommunikasjonsvei mellom to enheter for hele varigheten av en samtale eller en dataoverføring.

I et circuit-switched nettverk opprettes en fysisk forbindelse mellom avsender og mottaker før dataene overføres. Denne forbindelsen er dedikert og reservert for bruk av de to partene under hele kommunikasjonen.

Prosessens med circuit switching kan beskrives som følger:

- Oppsett: Når en samtale eller en dataoverføring initieres, etablerer nettverket en forbindelse mellom avsenderen og mottakeren. Dette innebærer å reservere og opprette en dedikert kommunikasjonskanal som bare brukes av de to partene.
- Overføring: Når forbindelsen er etablert, kan dataene overføres mellom avsender og mottaker. Dataene blir delt opp i faste størrelser, kalt "pakker" eller "celler", og sendes sekvensielt over den dedikerte kanalen.
- Avslutning: Etter at dataene er overført, blir den dedikerte forbindelsen frigjort og kan brukes av andre kommunikasjonssesjoner. Ressursene som ble tildelt for forbindelsen, blir frigitt og gjort tilgjengelig igjen.

Fordeler med circuit switching inkluderer pålitelighet og forutsigbarhet. Siden det er en dedikert forbindelse, er det lav sannsynlighet for tap eller forsinkelse av data. I tillegg er overføringshastigheten konstant og forutsigbar for hele kommunikasjonen. Dette gjør circuit switching spesielt egnet for talekommunikasjon og andre applikasjoner som krever stabil og kontinuerlig datastrøm.

Ulempen med circuit switching er at ressursene (båndbredde) forblir dedikert til en enkelt forbindelse, selv om den ikke utnyttes fullt ut hele tiden. Dette kan føre til ineffektiv bruk av nettverksressurser, spesielt når trafikken er uregelmessig eller lav. I tillegg kan etablering av en dedikert forbindelse ta tid og ressurser, spesielt i store nettverk med mange deltagere.

2.1.21 Hva er forskjellen på Circuit og Packet switching?

<p>■ Circuit switching</p> <ul style="list-style-type: none"> — Connection establishment can take a long time — Bandwidth is reserved <ul style="list-style-type: none"> • No danger of congestion • Possibly poor bandwidth utilization (burst traffic) 	<p>■ Packet switching</p> <ul style="list-style-type: none"> — No connect phase — No allocation of bandwidth <ul style="list-style-type: none"> • Danger of congestion • Potentially 100% bandwidth utilization
<ul style="list-style-type: none"> — constant transmission time — all data is transmitted over the same path 	<ul style="list-style-type: none"> — varying transmission time — packets between same end systems may use different paths

2.1.22 Hva er Virtuell Circuit Switching?

Virtual circuit switching, også kjent som virtual circuitry eller virtuell linjesvitsjing, er en kommunikasjonsmetode som kombinerer egenskapene til både circuit switching og packet switching.

I en virtuell krets opprettes en midlertidig, virtuell forbindelse mellom avsender og mottaker før dataene overføres. Denne virtuelle forbindelsen fungerer som om den er en dedikert krets, selv om den egentlig deler nettverksressurser med andre virtuelle kretser og pakkebaserte forbindelser.

Prosessen med virtuell linjesvitsjing kan beskrives som følger:

- Oppsett: Når en kommunikasjonssesjon initieres, etableres en virtuell forbindelse mellom avsenderen og mottakeren. Denne forbindelsen kan være permanent (for hele sesjonen) eller midlertidig (for en definert periode).
- Signalerings: Under oppsettet utveksler avsenderen og nettverket signaleringsmeldinger for å konfigurere den virtuelle forbindelsen. Dette inkluderer informasjon om ruting, kvalitetstjenester, og andre parametere som er nødvendige for overføringen.
- Overføring: Når den virtuelle forbindelsen er etablert, blir dataene delt opp i mindre enheter kalt pakker eller celler, på samme måte som i pakkesvitsjing. Disse pakkene sendes over den virtuelle forbindelsen i sekvensiell rekkefølge.
- Avslutning: Etter at dataene er overført, blir den virtuelle forbindelsen frigjort og kan brukes av andre sesjoner. Ressursene som ble tildelt for forbindelsen, blir frigitt og gjort tilgjengelig igjen.

Ved bruk av virtuell linjesvitsjing kan nettverket opprettholde en tilstandsinformasjon for den virtuelle forbindelsen, som rutingstabeller og bufferstørrelser, for å sikre at pakkene følger den forhåndsdefinerte ruten og kvalitetstjenestene.

Denne tilnærmingen kombinerer fordelene ved circuit switching, som forutsigbarhet og garantert båndbredde, med fleksibiliteten og effektiviteten til packet switching, som deling av nettverksressurser og optimal bruk av båndbredde.

Virtual circuit switching brukes blant annet i protokoller som Frame Relay og ATM (Asynchronous Transfer Mode) for å etablere midlertidige forbindelser mellom nettverkselementer og støtte pålitelige dataoverføringer med forutsigbarhet og kontroll.

■ Virtual circuit switching	■ Packet switching
- Connection establishment can take a long time	- No connect phase
- Bandwidth can be reserved	- No allocation of bandwidth
• Provider can guarantee loss probability of package losses and probability of waiting times	• Danger of congestion
• Better bandwidth utilization than circuit switching	• Potentially 100% bandwidth utilization
- varying transmission time	- varying transmission time
- all data is transmitted over the same path	- packets between same end systems may use different paths

2.1.23 Hva er Message switching?

Message switching er en kommunikasjonsmetode der data blir delt opp i separate meldinger eller pakker og sendt uavhengig av hverandre gjennom et nettverk. I motsetning til circuit switching, der det opprettes en dedikert og kontinuerlig forbindelse mellom avsender og mottaker, behandles hver melding individuelt og kan følge forskjellige ruter gjennom nettverket.

I message switching blir hver melding adressert med avsenderens og mottakerens identifikasjon, i tillegg til annen nødvendig kontrollinformasjon. Meldingene sendes fra en node i nettverket til en annen basert på adressen og rutetabeller som bestemmer den optimale veien. Når en melding ankommer en node, blir den midlertidig lagret før den videresendes til neste node i ruten.

En viktig egenskap ved message switching er at meldingene kan følge forskjellige ruter gjennom nettverket basert på aktuelle nettverksforhold, som trafikkbelastning eller ledig kapasitet. Dette gjør det mulig å oppnå bedre utnyttelse av nettverksressurser og tilpasse seg dynamiske forhold.

Message switching kan brukes i ulike nettverksscenerier, som datanettverk og telekommunikasjonsnettverk. Det tillater effektiv overføring av data ved å segmentere dem i mindre enheter og sende dem uavhengig av hverandre. I tillegg støtter message switching muligheten for å håndtere feil og rute om meldinger ved problemer i nettverket.

Det finnes forskjellige protokoller og teknologier som implementerer message switching, for eksempel datanettverksprotokoller som IP (Internet Protocol) for pakkesvitsjing og X.25-protokollen som brukes i telekommunikasjonsnettverk.

2.1.24 Hva er IPC?

IPC (Interprocess Communication) er en mekanisme som lar prosesser som kjører på en datamaskin, utveksle informasjon og samordne handlingene sine. I moderne operativsystemer kan flere prosesser kjøre samtidig på en datamaskin, og de kan ha behov for å kommunisere med hverandre for å dele ressurser, utveksle data eller samordne aktiviteter.

Det finnes flere IPC-mekanismer tilgjengelig i moderne operativsystemer, inkludert:

- Fellesminne: Denne IPC-mekanismen lar to eller flere prosesser dele et felles område av minnet som er tilgjengelig for alle. Dette området av minnet kan brukes til å utveksle data eller dele ressurser mellom prosessene.
- Meldingspassering: Denne IPC-mekanismen innebærer å sende meldinger mellom prosesser, som kan inneholde data, signaler eller forespørsler. Denne mekanismen kan være enten synkron eller asynkron, avhengig av om avsenderen blokkerer til mottakeren bekrefter meldingen.
- Pipes: Et rør er en unidireksjonal kommunikasjonskanal mellom to prosesser. Én prosess skriver data til røret, og den andre prosessen leser data fra røret. Rør brukes vanligvis i skallrørledninger for å koble to kommandoer sammen.
- Sockets: Sockets er en mekanisme for kommunikasjon mellom prosesser over et nettverk. De kan brukes til å utveksle data mellom prosesser som kjører på forskjellige datamaskiner som er koblet til internett.

IPC er en avgjørende funksjon i moderne operativsystemer, da det gjør det mulig for prosesser å samarbeide og dele ressurser effektivt. IPC introduserer imidlertid også utfordringer som synkronisering, dødlås og konflikter som må håndteres nøyne for å sikre korrekt og effektiv drift av systemet

2.1.25 Hva er filsystemer?

- Organiserer data i filer og håndterer aksess uavhengig av device type:
 - storage management
 - allocating space for files on secondary storage
 - file management

mechanisms for files to be stores, referenced, shared, secured

- file integrity mechanisms - ensuring that information is not corrupted, intended content only
- access methods - provide methods to access stored data

2.1.26 Hva er nettmasker?

Nettmasker er en måte å definere størrelsen på et IP-nettverk på. En nettmasken består av et sett med bits som forteller hvilke deler av en IP.-adresse som representerer nettverksdelen og hvilke som representerer verts-delen.

En IP-adresse består av 32 bits i IPv4 eller 128 bits i IPv6, og nettmasken brukes til å skille mellom nettverksdelen og verts-delen av IP-adressen. For eksempel, hvis nettmasken er satt til 255.255.255.0 i en IPv4-adresse, betyr det at de første 24 bits av IP-adressen representerer nettverksdelen, mens de siste 8 bits representerer verts-delen.

Nettmasker brukes til å identifisere hvilke enheter som er en del av det samme IP-nettverket og dermed kan kommunisere direkte med hverandre. Enheter som er i forskjellige IP-nettverk, må kommunisere via rutere.

Nettmasker kan også brukes til å definere subnett, som er mindre deler av et større IP-nettverk. For eksempel, ved å bruke en nettmasken på 255.255.255.128 kan man dele et IPv4-nettverk i to subnett, hvor hver har plass til 126 enheter.

I IPv6 brukes også nettmasker, men på grunn av det store antallet adresser som er tilgjengelige, brukes vanligvis prefiks lengder i stedet for nettmasker for å definere størrelsen på nettverk.

2.1.27 Hvordan regner man ut nettmasker?

Nettmasken er en bitsekvens som angir hvilken del av en IP-adresse som representerer nettverket, og hvilken del som representerer enheter (eller "verter") på nettverket. For å regne ut en nettmasken, trenger du å vite hvor mange enheter du planlegger å ha på nettverket ditt.

La oss si at du planlegger å ha plass til 254 enheter på nettverket ditt. For å representere 254 enheter, trenger du 8 bits, siden $2^8 = 256$. Vi trekker fra to siden både nettverksadressen og broadcastadressen opptar en adresse hver, noe som gir oss 254 brukbare adresser for enheter.

En IP-adresse består av 32 bits. Hvis vi bruker 8 bits for enhetene, betyr det at de resterende 24 bits brukes for nettverksadressen. En nettmasken på 24 bits er representeret som 255.255.255.0 i desimal form.

Dette er fordi hver 8-bits sekvens i nettmasken kan representeres som et desimaltall mellom 0 og 255:

- 255 representerer en 8-bits sekvens med alle bits satt til 1 (11111111 i binær form)
- 0 representerer en 8-bits sekvens med alle bits satt til 0 (00000000 i binær form)

Så for nettverket vårt med plass til 254 enheter, er nettmasken 255.255.255.0.

I praksis er det vanlig å se nettmasker representert ved bruk av CIDR notasjon. For eksempel, i stedet for å si "nettverket 192.168.1.0 med nettmaske 255.255.255.0", kan du si "nettverket 192.168.1.0/24". Tallet etter skråstrekken er antall bits i nettmasken.

2.1.28 Hva er nettmaskeklasser?

IP-adresser og nettmasker ble opprinnelig delt inn i klasser for å forenkle rutingen på Internett. Denne klassedelingen, kjent som klassefull adressering, ble senere erstattet av Classless Inter-Domain Routing (CIDR), men det er fortsatt nyttig å forstå klassene for historisk og pedagogisk kontekst.

Det er fem klasser av IP-adresser, fra A til E:

1. **Klasse A:** Nettmasken for Klasse A er 255.0.0.0, som betyr at den første oktetten (de første 8 bits) representerer nettverket, og de resterende tre oktettene representerer verter. Klasse A adresser spenner fra 1.0.0.0 til 126.0.0.0.
2. **Klasse B:** Nettmasken for Klasse B er 255.255.0.0, som betyr at de to første oktettene representerer nettverket, og de to siste oktettene representerer verter. Klasse B adresser spenner fra 128.0.0.0 til 191.255.0.0.
3. **Klasse C:** Nettmasken for Klasse C er 255.255.255.0, som betyr at de tre første oktettene representerer nettverket, og den siste oktetten representerer verter. Klasse C adresser spenner fra 192.0.0.0 til 223.255.255.0.
4. **Klasse D:** Denne klassen er reservert for multicast adresser. Klasse D adresser spenner fra 224.0.0.0 til 239.255.255.255.
5. **Klasse E:** Denne klassen er reservert for fremtidig bruk og eksperimentell bruk. Klasse E adresser spenner fra 240.0.0.0 til 254.255.255.255.

For Klasse A, B, og C, vil første oktett i en IP-adresse avgjøre klassen. For eksempel, hvis den første oktetten er mellom 1 og 126, vil den være en klasse A adresse. Hvis den første oktetten er mellom 128 og 191, vil den være en klasse B adresse, osv.

Det er viktig å merke seg at denne klassedelingen er for det meste historisk og at CIDR er mye mer vanlig i moderne nettverk. CIDR gir mer fleksibilitet og mer effektiv bruk av IP-adresserommet.

2.1.29 Hva er forskjellen på UDP og TCP?

UDP (User Datagram Protocol) og TCP (Transmission Control Protocol) er begge protokoller som brukes til å sende data over et nettverk, men de har noen viktige forskjeller i hvordan de fungerer:

1. **Tilkoblingsorientert vs. tilkoblingsløs:** TCP er en tilkoblingsorientert protokoll, noe som betyr at det opprettes en pålitelig og stabil forbindelse mellom avsender og mottaker før dataoverføringen starter. UDP er derimot en tilkoblingsløs protokoll, noe som betyr at den ikke oppretter en slik forbindelse før data sendes.

2. Pålitelighet: TCP garanterer levering av data til mottakeren. Det innebærer at det overvåker og håndterer tapte eller korrupte pakker, og sørger for at dataene kommer frem i riktig rekkefølge. UDP har ingen innebygd mekanisme for pålitelighet. Det sender bare dataene og håper på det beste. Dette gjør UDP raskere, men også mindre pålitelig enn TCP.
3. Pakkelevering: TCP sørger for at pakkene kommer frem i riktig rekkefølge og uten tap. Hvis en pakke går tapt, vil TCP be om en ny levering av den manglende pakken. UDP leverer pakkene uavhengig av rekkefølgen de sendes i, og det kan også oppstå tapte pakker uten at det oppdages.
4. Overhead: TCP legger til en betydelig mengde overhead til dataene som sendes på grunn av behovet for å opprette og vedlikeholde forbindelsen samt sikre pålitelig levering. UDP har mindre overhead sammenlignet med TCP, noe som gjør det mer egnet for rask og effektiv dataoverføring der pålitelighet ikke er like kritisk.
5. Bruksområder: TCP er ofte foretrukket for applikasjoner som krever pålitelig overføring av data, for eksempel webtrafikk, e-post og filoverføring. UDP brukes ofte der lav latenstid og høyere hastighet er viktigere enn pålitelighet, for eksempel i streaming av medieinnhold, videospill og VoIP (Voice over IP)-kommunikasjon.

Det er viktig å velge riktig protokoll basert på kravene til applikasjonen eller tjenesten du bruker, enten du trenger pålitelighet og sekvensiell levering (TCP) eller rask og mindre pålitelig levering (UDP).

2.1.30 Hva betyr båndbredde?

Båndbredde refererer til den målte kapasiteten eller hastigheten på en kommunikasjonskanal. Det er et mål på hvor mye data som kan overføres over en nettverksforbindelse i løpet av en gitt tidsperiode. Båndbredde måles vanligvis i bits per sekund (bps) eller i større enheter som kilobits per sekund (Kbps), megabits per sekund (Mbps) eller gigabits per sekund (Gbps).

Båndbredde kan betraktes som bredheten på veien gjennom hvilken data kan reise. Jo høyere båndbredde, desto større mengde data kan overføres raskt. Høy båndbredde muliggjør raskere nedlasting og opplasting av filer, streaming av videoer og overføring av store mengder data over nettverket.

Det er viktig å merke seg at båndbredde kan variere avhengig av nettverksinfrastrukturen og tilkoblingen. Faktorer som nettverksbelastning, kvaliteten på forbindelsen, og antall brukere som deler båndbredden, kan påvirke den faktiske hastigheten og ytelsen som oppnås.

Båndbredde er et viktig begrep i forbindelse med internettforbindelser, nettverk og kommunikasjonssystemer, da det påvirker brukeropplevelsen og muligheten til å overføre data effektivt. Høy båndbredde er spesielt viktig for oppgaver som krever

rask dataoverføring, som streaming av videoinnhold, videokonferanser, skytjenester og andre nettverksintensive applikasjoner.

2.1.31 Hva er forskjellen på IPv4 og IPv6, og hva brukes i dag?

IPv4 (Internet Protocol version 4) og IPv6 (Internet Protocol version 6) er to forskjellige versjoner av Internett-protokollen som brukes til å identifisere og rute datapakker over internett. Her er forskjellene mellom IPv4 og IPv6:

IPv4:

- Adressedesign: IPv4 bruker 32-biters adresser og tillater omtrent 4,3 milliarder unike adresser. Disse adressene er vanligvis representert som fire tallsekksjoner, kjent som dotted-decimal notation (for eksempel 192.168.0.1).
- Pakkeformat: IPv4-pakker har et fast størrelse på 20 bytes for hodeinformasjonen. De støtter også alternativer og tjenester som Quality of Service (QoS) og Fragmentering av pakker.
- Natting: På grunn av den begrensedes tilgjengeligheten av IPv4-adresser, blir Network Address Translation (NAT) ofte brukt for å oversette private IP-adresser til offentlige IP-adresser, slik at flere enheter kan dele en enkelt offentlig IP-adresse.

IPv6:

- Adressedesign: IPv6 bruker 128-biters adresser og støtter omtrent $3,4 \times 10^{38}$ unike adresser. Adressene er vanligvis representert som åtte grupper av fire heksadesimale sifre (for eksempel 2001:0db8:85a3:0000:0000:8a2e:0370:7334).
- Pakkeformat: IPv6-pakker har et fast størrelse på 40 bytes for hodeinformasjonen. De forenkler protokollformatet og fjerner mange av alternativene som finnes i IPv4, for å forbedre ytelse og enklere rutning.
- Auto-konfigurasjon: IPv6 har innebygd støtte for adressekonfigurasjon uten bruk av DHCP (Dynamic Host Configuration Protocol). Enheter kan generere sine egne unike adresser ved hjelp av en kombinasjon av MAC-adressen og informasjon fra nettverket.

I dagens nettverksinfrastruktur er både IPv4 og IPv6 i bruk. IPv4 har vært i bruk i mange år og er fortsatt dominerende. Men på grunn av den økende etterspørsmålet etter IP-adresser og behovet for å støtte flere enheter og tjenester, er overgangen til IPv6 i gang. IPv6 blir gradvis implementert for å løse begrensningene til IPv4 og sikre tilstrekkelig adresseplass og støtte for fremtidige nettverkstilpasninger. Mange operativsystemer, rutere og nettverksinfrastrukturer støtter nå både IPv4 og IPv6 for en sømløs overgang og fremtidig vekst.

2.1.32 Hvordan sendes en pakke ved hjelp av TCP?

Steg for steg:

1. Avsender sender en SYN til mottakeren
2. Mottakeren sender en SYN ACK tilbake som bekrefter at den fikk meldingen

3. Avsenderen sender en ACK Received tilbake
4. Etter det er forbindelsen opprettet
5. Avsenderen sender data til mottakeren

Garanterer at alle pakker blir sendt

2.1.33 Congestion control

Congestion control, eller metningskontroll på norsk, er en metode og en prosess som brukes i nettverkssystemer for å regulere og begrense trafikkmengden i nettverket. Formålet med trengselskontroll er å forhindre overbelastning av nettverket ved å sikre at mengden av sendt data ikke overskridt nettverkets kapasitet.

Når mange datamengder sendes gjennom et nettverk, kan det oppstå trengsel eller flaskehalsar som fører til tregere overføringshastighet, pakketap eller feil i nettverket. Congestion control implementerer mekanismer som overvåker og kontrollerer trafikkmengden for å opprettholde en stabil og effektiv drift av nettverket.

En vanlig metode for metningskontroll er TCP (Transmission Control Protocol), som er en pålitelig transportprotokoll som brukes i internett. TCP implementerer en rekke mekanismer for metningskontroll, inkludert flytkontroll og vinduskontroll.

Flytkontrollen begrenser mengden av data som senderen kan sende uten å overbelaste mottakeren. Vinduskontrollen regulerer antallet ubekrefte datapakker som kan være i nettverket samtidig.

2.1.34 Hva er trengselskontroll

Trengselskontroll kan også omfatte andre metoder som køhåndtering, pakkeprioritering, rutingalgoritmer og Quality of Service (QoS)-mekanismer. Målet er å sikre at nettverket fungerer optimalt, unngå trengsel og opprettholde kvaliteten på tjenesten for brukerne.

Ved å implementere trengselskontroll i nettverkssystemer kan man sikre at trafikken blir effektivt håndtert og at nettverket opererer innenfor sine kapasitetsgrenser. Dette bidrar til å opprettholde pålitelighet, redusere pakketap og forsinke, samt sikre jevn og stabil ytelse i nettverket.

2.1.35 FIFO vs Round Robin

- Ordningen av oppgaver: FIFO følger prinsippet om at oppgavene som kommer først, blir utført først. Det betyr at oppgavene blir behandlet i den rekkefølgen de ankommer. Round Robin, derimot, bruker en syklistisk tilnærming der oppgavene blir tildelt en fast tidskvote eller kvantum. Hver

oppgave får en viss tid til å kjøre, og deretter går prosessoren videre til neste oppgave, uavhengig av om den første oppgaven er fullført eller ikke.

- Prosesseringstid: I FIFO-algoritmen utføres hver oppgave til den er ferdig, og deretter blir den neste oppgaven utført. Dette betyr at en lang oppgave kan ta opp mye prosessortid og forsinke behandlingen av påfølgende oppgaver. I Round Robin-algoritmen får hver oppgave en tidskvote, og selv om en oppgave ikke er ferdig innenfor sin kvote, blir den satt på vent og prosessoren går videre til neste oppgave. Dette sikrer at ingen oppgave monopoliserer prosessortiden for lenge.
- Responsivitet: I FIFO-algoritmen kan det oppstå situasjoner der oppgaver med lengre kjøretid forsinkes, noe som kan føre til høyere responstid for andre oppgaver. I Round Robin-algoritmen får hver oppgave en lik andel av prosessortiden, noe som gir bedre responsivitet og jevnere kjøring av oppgavene.
- Kontekstbytte: FIFO-algoritmen krever ikke hyppige kontekstbytter, ettersom hver oppgave blir utført til den er ferdig. Round Robin-algoritmen, derimot, krever hyppige kontekstbytter mellom oppgaver når tidskvotene utløper. Dette kan medføre en viss overhead i systemet.
- Bruksområder: FIFO-algoritmen er enkel og vanligvis egnet for enkle scenarier der oppgavene ikke har spesifikke tidsbegrensninger eller prioriteringer. Round Robin-algoritmen er mer egnet for systemer der ressursene må deles rettferdig mellom flere oppgaver, for eksempel i fleroppgavesystemer og tidsdelingsoperativsystemer.
- FIFO er en ikke-preemptiv algoritme, mens RR er en preemptiv algoritme

2.1.36 Hvordan brukes heapen og stacken i minnet?

Heap og stack er to forskjellige områder i dataminnet som brukes til å lagre variabler og data under kjøringen av et program. Her er noen grunnleggende forskjeller mellom heap og stack:

1. Struktur: Stacken er en lineær datastruktur som følger LIFO (Last-In-First-Out) prinsippet, mens heapen er en mer dynamisk datastruktur som tillater tilfeldig tilgang til dataene.
2. Hukommelsesallokering: Stacken bruker automatisk hukommelsesallokering, der variabler blir tildelt og frigitt automatisk når de kommer inn og går ut av omfanget (scope). Heapen bruker manuell hukommelsesallokering, der programmøren er ansvarlig for å tildele og frigi minnet.
3. Størrelse og kapasitet: Stacken har en begrenset størrelse og har en fast kapasitet. Den har vanligvis en forhåndsdefinert størrelse som blir satt ved programmets kjøretid. Heapen derimot er mye større og kan vokse og krympe dynamisk under kjøretiden av programmet.
4. Levetid: Variabler som er plassert på stacken, lever bare så lenge som omfanget der de er definert. Når omfanget avsluttes, blir variablene

automatisk frigjort. Variabler som er tildelt på heapen, kan leve lenger enn omfanget der de er opprettet. Det er programmørens ansvar å frigi minnet når det ikke lenger er nødvendig for å unngå minnelekkasjer.

5. Fragmentering: Heapen kan lide av fragmentering, der minnet blir delt opp i mindre biter over tid når minnet allokeres og frigjøres. Dette kan føre til at det blir vanskeligere å finne sammenhengende ledige blokker av minne. Stacken lider ikke av denne typen fragmentering, da variabler blir plassert i kontinuerlig minne og frigjøres i motsatt rekkefølge de ble opprettet.

Bruken av stacken og heapen avhenger av programmets behov. Generelt sett brukes stacken til å lagre lokale variabler, funksjonsparametere, returadresser og andre data som er nødvendige for funksjonskall og retur. Heapen brukes vanligvis når det er behov for dynamisk allokering av minne, for eksempel når objekter opprettes i kjøretiden eller når man arbeider med store datamengder som ikke får plass i stacken.

2.1.37 Hva er forskjellen på fragmentering, segmentering og partisjonering?

Fragmentering:

- Fragmentering er en situasjon der det oppstår spredning av ledige og ubrukte minneområder i dataminnet som ikke kan brukes til å lagre nye data.
Fragmentering kan oppstå på to måter:
- Intern fragmentering: Dette skjer når det er en sløsing av minne innenfor en enhet, som en blokk eller et segment, der det er ubrukt plass på grunn av minsteallokeringsenheten. For eksempel, hvis en blokk er 64 bytes, og en 50-byte fil blir lagret i den, vil det være 14 ubrukte bytes.
- Ekstern fragmentering: Dette skjer når det er ledige områder mellom forskjellige blokker eller segmenter av data som ikke er sammenhengende, og derfor ikke kan brukes til å lagre større data. Selv om det kan være nok totalt ledig minne, kan det ikke være tilstrekkelig sammenhengende minne tilgjengelig for å tilfredsstille en forespørsel om å lagre store data.

Fragmentering kan redusere effektiviteten og utnyttelsen av minnet, og det kan være nødvendig å utføre defragmentering eller andre metoder for å gjenopprette sammenhengende og optimal utnyttelse av minnet.

Segmentering:

- Segmentering er en minneadministrasjonsmetode der minnet er delt inn i logiske segmenter eller blokker. Hvert segment representerer en logisk enhet av programmet, for eksempel kode, data, stabler eller heap. Segmentene har forskjellige størrelser og kan vokse eller krympe dynamisk avhengig av programmets behov.

Fordeler med segmentering inkluderer fleksibilitet og logisk organisering basert på programstrukturen. Imidlertid kan segmentering også føre til fragmentering og kompleks administrasjon av segmentert minne.

Partisjonering:

- Partisjonering, også kjent som paging, er en minneadministrasjonsmetode der minnet er delt inn i like store blokker som kalles sider eller rammer. Disse sidene er vanligvis på samme størrelse som sider i virtuelt minne. Programmet blir delt inn i like store deler kalt sider, som tilsvarer størrelsen på sidene i den fysiske minnedelen.

Fordeler med partisjonering inkluderer enkel administrasjon og reduksjon av fragmentering. Imidlertid kan partisjonering også føre til intern fragmentering og manglende fleksibilitet når det gjelder å tilpasse seg dynamiske minnebehov.

Både segmentering og partisjonering er metoder for å organisere minne, men de har forskjellige tilnæringer og konsekvenser. Valget mellom dem avhenger av spesifikke krav og egenskaper ved systemet og programmene som kjører på det. Fragmentering er en uønsket tilstand som kan oppstå i både segmenterte og partisjonerte minnesystemer og reduserer effektiviteten og ytelsen til dataminnet.

2.1.38 Forskjell på disk og primærminne

Diskminne og primærminne er to forskjellige typer minne som brukes i datamaskiner. Her er forskjellene mellom diskminne (sekundærminne) og primærminne (hovedminne):

Primærminne (Hovedminne):

Primærminne refererer til den direkte tilgjengelige minnetilgangen for datamaskinen i sanntid. Den består vanligvis av RAM (Random Access Memory) og cache-minne. Primærminnet er raskt, men har begrenset kapasitet sammenlignet med diskminne.

Forskjeller med primærminne inkluderer:

- Direkte tilgang: Primærminnet gir direkte tilgang til data og instruksjoner som er nødvendige for øyeblikkelig behandling av CPU (sentral prosessorenhet). Det er flyktig, noe som betyr at dataene blir tapt når strømmen slås av eller maskinen startes på nytt.
- Raskere tilgangstid: Primærminnet har en mye raskere tilgangstid sammenlignet med diskminnet. Dataene kan hentes fra primærminnet på mikrosekundnivå.
- Begrenset kapasitet: Primærminnet har typisk mindre kapasitet enn diskminnet. Det er vanligvis i størrelsesorden gigabyte (GB) eller terabyte (TB).

Diskminne (Sekundærminne):

Diskminne, som ofte refereres til som sekundærminne, inkluderer harddisker (HDD), solid state drives (SSD), optiske disker (CD/DVD) og andre lagringsenheter. Diskminnet brukes til å lagre data permanent, selv når strømmen er slått av.

Forskjeller med diskminne inkluderer:

- Ikke-direkte tilgang: Dataene på diskminnet må overføres til primærminnet før de kan behandles av CPU. Dette tar betydelig lengre tid sammenlignet med primærminnet.
- Langsommere tilgangstid: Tilgangstiden til data på diskminnet er mye lengre enn primærminnet. Den måles i millisekunder eller enda lengre, avhengig av lagringseenheten og dataens plassering på disken.
- Stor kapasitet: Diskminnet har betydelig større kapasitet enn primærminnet. Det kan variere fra flere hundre gigabyte til flere terabyte.
- Permanent lagring: Dataene som er lagret på diskminnet, er ikke flyktige og forblir bevart selv etter at strømmen er slått av. Diskminnet brukes til langsiktig lagring av filer, programmer og operativsystemer.

Begge typer minne har forskjellige roller og funksjoner i datamaskinsystemet. Primærminnet er nødvendig for umiddelbar tilgang til data og instruksjoner under kjøring av programmer, mens diskminnet brukes til langsiktig lagring av data og programfiler.

2.1.39 Forklar Second Chance algoritmen

Second Chance-algoritmen, også kjent som Clock-algoritmen, er en sideutskiftningsalgoritme som brukes i operativsystemer for å håndtere minnehåndtering. Når operativsystemet trenger å hente en side fra disk, men har ikke lenger ledig plass i RAM, må det velge en side for å bytte ut. Second Chance-algoritmen er en måte å gjøre dette valget på.

1. Hver side i minnet får en referansebit som er satt til 0 når siden først lastes inn i minnet.
2. Når en side refereres til (det vil si lest eller skrevet til), blir referansebitten satt til 1.
3. Når operativsystemet trenger å bytte ut en side, ser det på den eldste siden først (som i en First-In, First-Out-algoritme). Hvis referansebitten til denne siden er 0, blir siden valgt for utskifting.
4. Hvis referansebitten til den eldste siden er 1, får siden en "second chance". Referansebitten blir satt tilbake til 0, og siden blir flyttet til enden av listen over sider (som om den var den nyeste siden). Operativsystemet går deretter videre til neste eldste side og gjentar prosessen.
5. Dette fortsetter til operativsystemet finner en side med en referansebit på 0, eller det har gått gjennom alle sidene.

Second Chance-algoritmen er en forbedring av en ren FIFO-sideutskiftningsalgoritme, da den tar hensyn til nylig brukte sider og unngår å bytte ut en side som aktivt brukes.

2.1.40 Hva er en kontekstsvitsj og hva brukes det til?

En kontekstsvitsj (context switch) er en kjernefunksjonalitet i et operativsystem som gjør det mulig å bytte kontekst mellom forskjellige prosesser eller tråder som kjører på en datamaskin. Når en kontekstsvitsj oppstår, blir tilstanden til den gjeldende prosessen eller tråden lagret, og kontrollen over prosessoren overføres til en annen prosess eller tråd.

Formålet med en kontekstsvitsj er å oppnå multitasking eller fleroppgavekjøring, der flere prosesser eller tråder kan kjøre samtidig på en enkelt prosessor.

Kontekstsvitsjer gjør det mulig å gi hver prosess eller tråd sin egen virtuelle prosessortid og ressurser, slik at det virker som om de kjører samtidig.

2.1.41 Hva er big og small endian og hvordan brukes det?

Big-endian og little-endian er to forskjellige måter å lagre eller hente data på i datasystemer. Disse terminologiene blir primært brukt innenfor datavitenskap og programmering for å beskrive hvordan bytes ordnes i minnet.

Hva betyr big-endian og little-endian?

Når det gjelder et flerbyte-datavarehus (som en 32-biters heltall), er forskjellen mellom big-endian og little-endian hvor den mest signifikante byten (MSB) og den minst signifikante byten (LSB) blir lagret og hentet i hukommelsen.

Big-Endian: I et big-endian-system lagres den mest signifikante byten først (i den laveste minneadressen). For eksempel, for et 4-byte heltall ville det se ut som dette i hukommelsen:

```
0x400 | MSB  
0x401 |  
0x402 |  
0x403 | LSB
```

Little-Endian: I et little-endian-system lagres den minst signifikante byten først (i den laveste minneadressen). Her er hvordan det ville se ut for et 4-byte heltall:

```
0x400 | LSB  
0x401 |  
0x402 |  
0x403 | MSB
```

Disse konvensjonene er viktige i nettverksprotokoller, filformater og systemarkitekturen. For eksempel bruker Intel x86-arkitekturen little-endian, mens mange nettverksprotokoller bruker big-endian.

En programmerer må være klar over endianess når de arbeider på tvers av forskjellige systemer (for eksempel når du sender data over nettverket mellom forskjellige typer datamaskiner), fordi tolkningen av data kan variere mellom big- og little-endian systemer. Det er nødvendig å konvertere fra den ene til den andre hvis de to systemene bruker forskjellige konvensjoner. Dette kalles "endianess konvertering".

Man bruker Little-Endian og Big-Endian på grunn av forskjellige faktorer og behov i ulike systemer:

Little-Endian:

- Effektiv datomanipulasjon: Little-Endian er spesielt gunstig for datomanipulasjon og beregninger, spesielt på arkitekturen som bruker denne ordningen. Ved å lagre den minst signifikante byten først, kan prosessoren enklere utføre aritmetiske operasjoner, som addisjon eller multiplikasjon, ved å jobbe seg oppover i minnet.

Kompatibilitet med Intel x86-arkitektur: Intel x86-arkitekturen er en av de mest utbredte arkitekturene i bruk i dag, og den bruker Little-Endian som standard. Derfor er det enklere og mer effektivt å behandle data i Little-Endian-format på Intel-baserte systemer uten behov for konvertering.

Big-Endian:

- Network Byte Order: Mange nettverksprotokoller, som TCP/IP og HTTP, bruker Big-Endian som standard, også kjent som Network Byte Order. Dette skyldes historiske grunner og konsensus blant tidlige nettverksprotokoller og implementasjoner.

Enklere menneskelig lesbarhet: Big-Endian har den fordelen at det samsvarer med den måten mennesker leser tall på, hvor den mest signifikante delen kommer først. Dette gjør det lettere for mennesker å tolke tall og data som er lagret i Big-Endian-format.

Det er viktig å merke seg at valget mellom Little-Endian og Big-Endian er avhengig av faktorer som arkitektur, applikasjonskrav, og kompatibilitet med eksisterende systemer og protokoller. Derfor vil valget av endianess variere avhengig av bruksområdet og systemet det gjelder.

2.1.42 Hva er flytkontroll?

Flytkontroll er som en kommunikasjonsregel mellom to enheter som sender og mottar data. Ette forhindrer tap av data ved å sikre at senderen bare sender så mye data som mottakeren kan håndtere på et gitt tidspunkt.

2.1.43 Hva er metningskontroll?

Metningskontroll er en mekanisme som brukes i datanettverk for å begrense og kontrollere mengden trafikk som sendes gjennom nettverket. Formålet med metningskontroll er å hindre overbelastning av nettverket ved å begrense mengden data som blir sendt på en gitt tid.

2.1.44 Hva er DVR (Distance Vector Routing)

1. Rutertabeller: Hver ruter i nettverket har en rutertabell som inneholder informasjon om de tilgjengelige rutene og kostnadene knyttet til disse rutene. Denne informasjonen brukes til å ta rutingsbeslutninger.
2. Distansevektor: Hver ruter sender periodisk en distansevektor, som er en liste over avstander eller kostnader til forskjellige destinasjoner i nettverket, til sine naboer. Distansevektoren inkluderer informasjon om den lokale ruten og de beste rutene den kjenner til.
3. Utveksling av distansevektorer: Ruter utveksler sine distansevektorer med sine naboer ved hjelp av rutingsprotokollen. Disse distansevektorene blir sendt som meldinger og oppdaterer rutertabellene til mottakerruterne.
4. Beregning av beste rute: Basert på mottatte distansevektorer, oppdaterer hver ruter sin egen rutertabell ved å beregne den beste ruten til hver destinasjon. Dette gjøres ved å vurdere de mottatte avstandene og legge til sin egen kostnad til destinasjonen.
5. Spredning av informasjon: Når en ruter oppdaterer sin rutertabell, sender de endrede rutene til sine naboer, slik at endringene spres gjennom nettverket. Dette gjentas periodisk for å sikre at rutertabellene er oppdaterte.
6. Konvergens: Over tid konvergerer distance vector routing-protokollen mot en stabil rutetilstand der alle ruter har oppdaterte rutertabeller og kjenner de beste rutene til destinasjoner.

Det er viktig å merke seg at distance vector routing bruker begrenset informasjon om nettverket, spesielt kun informasjonen som er utvekslet mellom naboer. Ruter vet ikke nødvendigvis den eksakte topologien til hele nettverket, men baserer seg på informasjonen de mottar fra sine naboer for å ta beslutninger om rutene.

2.1.45 Hva er DNS (Domain Name System)?

DNS står for Domain Name System, og det er en distribuert og hierarkisk database som brukes til å oversette menneskelig lesbare domenenavn til numeriske IP-adresser som datamaskiner bruker for å kommunisere over nettverk. DNS fungerer som en telefonkatalog for internett, der det knytter domenenavn til IP-adresser.

VCR (Virtual Circuit Switching)

2.1.46 Hvordan fungerer Flytkontroll selective repeat

Selective Repeat er en flytkontrollmekanisme som brukes i nettverkskommunikasjon for å sikre pålitelig overføring av data

1. Data blir delt opp i mindre pakker før de sendes over nettverket. Hver pakke får et unikt nummer.
2. Senderen sender pakkene til mottakeren, og hver pakke har et nummer som identifiserer den.
3. Mottakeren mottar pakkene og lagrer dem midlertidig. Den sender også bekreftelser tilbake til senderen for å si at pakkene er mottatt.
4. Hvis en pakke går tapt eller blir skadet under overføringen, sender ikke mottakeren en bekreftelse for den pakken.
5. Når senderen ikke får en bekreftelse for en pakke, forstår den at pakken gikk tapt. Den sender derfor pakken på nytt.
6. Mottakeren kan identifisere manglende pakkene ved å sjekke pakkens nummer i forhold til rekkefølgen de skal være i. Den sender en melding tilbake til senderen og ber om å få manglende pakkene sendt på nytt.
7. Senderen retransmitterer bare de pakkene som ble etterspurtt av mottakeren.

Dette reduserer unødvendig retransmisjon og øker effektiviteten.

Selective Repeat-mekanismen tillater mottakeren å be om og motta de manglende pakkene, og senderen svarer ved å sende dem på nytt. Dette sikrer pålitelig overføring av data selv om noen pakkene går tapt eller blir skadet underveis.

2.1.47 Nettmasker, hvorfor, hvordan regne ut.

En nettverksmaske er et numerisk verktøy som brukes i nettverk for å definere og skille nettverksdelen og vertsdelen i en IP-adresse. Den hjelper med å identifisere hvilken del av IP-adressen som tilhører nettverket og hvilken del som tilhører enheten.

Tenk på en IP-adresse som et husnummer, og nettverksmasken som en veiskilt som forteller hvilken del av gaten som tilhører nettverket. Ved å sammenligne IP-adressen med nettverksmasken, kan man avgjøre hvilket nettverk enheten tilhører.

En nettverksmaske består av en sekvens med tall som bestemmer hvor nettverksdelen og vertsdelen begynner. Den bruker et binært system der 1-tall angir nettverksdelen og 0-tall angir vertsdelen.

Nettverksmasker brukes av flere grunner i nettverkssammenheng. Først og fremst brukes nettverksmasker til å identifisere og skille mellom nettverksdelen og vertsdelen i en IP-adresse. Dette er viktig for å organisere og administrere IP-adresser på en strukturert måte.

2.1.48 Forskjell på preemptiv og ikke-preemptiv skedulering

Preemptiv skedulering: Preemptiv skedulering betyr at en aktiv prosess eller tråd kan bli avbrutt eller "preempted" av systemet. Dette skjer når en høyere prioritet prosess eller tråd blir klar til å utføre, eller når en tidsenhet (timeslice) er brukt opp for den aktive prosessen. I preemptiv skedulering kan en prosess eller tråd bli midlertidig stoppet, og kontrollen blir overført til en annen prosess eller tråd med høyere prioritet. Dette tillater systemet å reagere raskt på viktige hendelser og sikrer jevn ressursallokering.

Ikke-preemptiv skedulering: I motsetning til preemptiv skedulering tillater ikke-preemptiv skedulering en aktiv prosess eller tråd å kjøre uavbrutt til den fullfører sin kjøretid eller frivillig gir fra seg kontrollen. I denne tilnærmingen beholder en prosess eller tråd kontrollen til den er ferdig eller frivillig slipper taket. Systemet griper ikke inn for å avbryte den aktive prosessen med mindre den frivillig gir fra seg kontrollen.

Forskjellen mellom preemptiv og ikke-preemptiv skedulering påvirker hvordan systemet prioriterer og håndterer ressursallokering. Preemptiv skedulering gir muligheten til å håndtere prioritet og tidsdeling på en mer dynamisk måte, da ressursene kan fordeles til prosesser eller tråder med høyere prioritet eller ved bruk av tidskvote. Ikke-preemptiv skedulering er mer forutsigbart, da en aktiv prosess eller tråd har full kontroll over ressursene inntil den frivillig gir fra seg kontrollen.

Preemptiv: Round Robin, Shortest Remaining Time First

Ikke-preemptiv: First-Come-First-Serve, Shortest Job First

2.1.49 Hva er et filsystem og hva håndterer det?

Samlet sett er filsystemet en viktig del av operativsystemet som håndterer oppgaver knyttet til organisering, tilgang, rettighetsstyring, vedlikehold og gjenfinning av filer og kataloger. Det sikrer at dataene er organisert, tilgjengelige, sikre og pålitelige.

2.1.50 CIDR (Classless InterDomain Routing)

Tradisjonelt var IP-adresser delt inn i fem klasser (A, B, C, D og E) basert på det første oktetet i adressen. Klassene A, B og C ble brukt til å identifisere ulike typer nettverk av varierende størrelser. Men dette klassifiserte systemet viste seg å være ineffektivt og resulterte i mangel på IP-adresser.

For å løse dette problemet, ble CIDR introdusert. CIDR tillater variabel lengde på prefikset, noe som gjør at IP-adresser kan deles inn i flere, mindre nettverk. Dette fører til en mer effektiv bruk av IP-adresseområdet.

CIDR-notasjon er en metode for å representere en subnettmasker som en del av IP-adressen. Dette er representert som $x.x.x.x/n$ der $x.x.x.x$ er IP-adressen og n er antall bits som brukes for nettverksidentifikasjon. For eksempel, en IP-adresse kan være 192.168.1.0/24. Her er '192.168.1.0' IP-adressen, og '24' (etter skråstrekken) er antall bits som er satt til 1 i subnettmasken.

2.1.51 Hva er forskjellen på OS-ene Windows, Unix og Linux?

Windows, Unix og Linux er tre forskjellige operativsystemer som brukes i datamaskiner og servermiljøer. Her er noen grunnleggende forskjeller mellom dem:

1. Eierskap og utvikling:
 - a. Windows: Windows-operativsystemet er utviklet og eies av Microsoft Corporation. Det er et proprietært operativsystem, noe som betyr at kildekoden ikke er offentlig tilgjengelig.
 - b. Unix: Unix er et tidligere operativsystem som opprinnelig ble utviklet av Bell Labs i 1969. Det har siden blitt videreutviklet og er nå tilgjengelig i flere varianter, inkludert proprietære og åpne varianter.
 - c. Linux: Linux er et åpent kildekodebasert operativsystem som er basert på Unix-konsepter. Det er utviklet av Linus Torvalds og en stor fellesskapsutvikling. Linux er distribuert under forskjellige lisenser, hovedsakelig GNU General Public License (GPL).
2. Lisens og tilgjengelighet:
 - a. Windows: Windows er et kommersielt operativsystem som krever kjøp av lisens for bruk. Det er hovedsakelig rettet mot personlige datamaskiner og brukes bredt i forbrukermarkedet.
 - b. Unix: Unix er tilgjengelig både som proprietære og åpne lisensvarianter. Proprietære Unix-varianter krever lisensiering, mens åpne Unix-varianter er gratis tilgjengelig.
 - c. Linux: Linux er distribuert under åpen kildekode-lisenser, som gjør det fritt tilgjengelig for bruk og modifikasjon. Det er vanligvis gratis å bruke og distribuere.
3. Brukervennlighet og grensesnitt:
 - a. Windows: Windows er kjent for sitt brukervennlige grensesnitt og er utviklet med tanke på enkel navigasjon og bruk av mus og tastatur. Det har et bredt utvalg av programvare og er populært blant vanlige brukere og bedrifter.
 - b. Unix: Unix-operativsystemer har tradisjonelt hatt et tekstbasert grensesnitt og en mer kommandolinjebasert tilnærming. De er ofte brukt i servermiljøer og av avanserte brukere og utviklere.

- c. Linux: Linux tilbyr både tekstbaserte grensesnitt og grafiske brukergrensesnitt (GUI). Det har et bredt utvalg av programvare og er populært blant utviklere, servere og avanserte brukere.
4. Skalering og anvendelse:
- a. Windows: Windows-operativsystemet er hovedsakelig rettet mot personlige datamaskiner, men det er også tilgjengelig i serversversjoner for nettverksmiljøer. Det brukes mye i bedriftsmiljøer og hjemmemiljøer.
 - b. Unix: Unix-operativsystemer er kjent for sin skalerbarhet og pålitelighet, og de brukes ofte i servermiljøer og kritiske systemer som banktjenester og telekommunikasjon.
 - c. Linux: Linux er også svært skalerbart og brukes i en rekke applikasjoner og miljøer, inkludert servere, superdatamaskiner, innebygde systemer og mobile enheter.

Det er viktig å merke seg at Unix og Linux er beslektede, da Linux er basert på Unix-konsepter og deler mye av den samme funksjonaliteten. Windows skiller seg ut som et eget proprietært operativsystem som er utviklet uavhengig av Unix- og Linux-systemene.

3 Forståelsesspørsmål

4.1 Forståelsesspørsmålene er systematisert i henhold til ukene i semesteret, og hver uke er delt inn i tre separate bolker med spørsmål. Spørsmålene er fremhevet med fet skrift, etterfulgt av de tilhørende svarene. Det bør bemerkes at det ikke finnes forståelsesspørsmål for alle ukene i dette semesteret, og derfor vil ikke alle ukenummerene være representert.

3.1 Lav- og høynivå språk

Programmeringsspråk som C, Java og Python, inndeles ofte i to-nivåer: vi snakker om såkalte lavnivå- og høynivåspråk. Selv om denne dikotomien ikke er entydig, har den likevel forklaringsverdi. I hovedsak er et språk på lavt nivå nærmere maskinkoden, de faktiske instruksjonene som kjører på CPU, enn et språk på høyt nivå er.

Hva er det mest lavnivåspråket du kan tenke på, og hvordan begrunner du dette?

- Assembler
- Argumentere for abstraksjonsnivåer
- Flytende grense mellom lavnivå og høynivå

Det mest lavnivåspråket jeg kan tenke på er maskinspråk. Maskinspråk er direkte representasjon av instruksjonene som en datamaskin forstår og utfører. Det er binære koder som representerer ulike maskininstruksjoner og minneadresser.

Maskinspråk er det laveste nivået av programmeringsspråk fordi det er direkte knyttet til maskinvaren. Instruksjonene i maskinspråket er spesifikke for hver maskinarkitektur, og de utnytter de mest grunnleggende funksjonene i prosessoren og minnet.

Maskinspråk krever en dyp forståelse av maskinarkitekturen og er svært avhengig av den spesifikke datamaskinen som brukes. Programmering i maskinspråk er tidkrevende og komplekst, da hver instruksjon må skrives og optimaliseres manuelt. Det er også lite portabelt, da maskinspråkprogrammer vanligvis ikke kan kjøres på forskjellige datamaskiner uten modifikasjoner.

På grunn av disse faktorene betraktes maskinspråk som det mest lavnivåspråket.

Fra brukerens snarere enn maskinens perspektiv, hva er noen særtrekk ved begge typer språk? Vurder for eksempel brukervennlighet, feilsøking, portabilitet og mer.

Både lavnivåspråk og høynivåspråk har særtrekk som påvirker brukervennlighet, feilsøking, portabilitet og mer. Her er noen generelle forskjeller mellom de to typene:

Lavnivåspråk:

Brukervennlighet: Lavnivåspråk som maskinspråk og assemblér er svært komplekse og krever en dyp forståelse av maskinvarearkitektur. Programmering i lavnivåspråk kan være krevende og mer tidkrevende sammenlignet med høynivåspråk.

Feilsøking: Feilsøking i lavnivåspråk kan være utfordrende, da det er vanskeligere å lese og tolke maskinkode eller assemblérinstruksjoner sammenlignet med høynivåspråk. Feil kan være mer subtile og vanskeligere å identifisere og rette opp.

Portabilitet: Lavnivåspråk er vanligvis ikke portabelt og avhenger av maskinvarearkitekturen. Programmer som er skrevet i lavnivåspråk, må i de fleste tilfeller omskrives eller tilpasses for å kunne kjøre på forskjellige plattformer.

Høynivå språk:

Brukervennlighet: Høynivåspråk som Python, Java og C er utviklet med fokus på enklere syntaks og abstraksjonsnivå. Dette gjør dem mer brukervennlige sammenlignet med lavnivåspråk. De tilbyr ofte innebygde funksjoner, biblioteker og rammer som forenkler utviklingsprosessen.

Feilsøking: Høynivåspråk gir vanligvis bedre feilsøkingsmuligheter ettersom de har mer robuste og forståelige feilmeldinger. De kan også ha innebygde verktøy som gjør det enklere å finne og fikse feil, for eksempel ved å inkludere støtte for unntakshåndtering.

Portabilitet: Høynivåspråk er generelt sett mer portabelt enn lavnivåspråk. De er vanligvis utviklet for å være plattformuavhengige, noe som betyr at programmene kan kjøres på ulike operativsystemer og maskinarkitekturen uten større endringer.

**Nevn noen fordeler og ulemper med begge typer språk, avhengig av formålet.
Vurder for eksempel systemprogrammering vs applikasjonsprogrammering.**

Lav nivå

Fordel	Ulempe
<ul style="list-style-type: none"> ○ Ytelse: Lavnivåspråk som C og C++ gir nærmere tilgang til maskinvaren, noe som gjør dem ideelle for systemprogrammering der optimal ytelse er avgjørende. ● ○ Kontroll: Lavnivåspråk gir utvikleren større kontroll over systemressurser, minnehåndtering og maskinens oppførsel, noe som er viktig for å lage systemnære programmer som drivere eller operativsystemkomponenter. 	<ul style="list-style-type: none"> ○ Kompleksitet: Lavnivåspråk krever en dyp forståelse av maskinvarearkitektur og kan være mer komplekse å programmere og feilsøke. De kan også være mer sårbar for minnelekkasjer og sikkerhetssårbarheter hvis de ikke brukes riktig. ● ○ Produktivitet: På grunn av den lavere abstraksjonsnivået kan utviklingstiden være lengre sammenlignet med høynivåspråk, da det krever mer manuelt arbeid og detaljstyring.

Høy nivå

Fordel	Ulempe
<ul style="list-style-type: none"> ○ Produktivitet: Høynivåspråk som Python og Java gir høyere abstraksjonsnivå, innebygde funksjoner og biblioteker, noe som gjør utviklingsprosessen raskere og enklere. ● ○ Plattformuavhengighet: Høynivåspråk er ofte 	<ul style="list-style-type: none"> ○ Ytelse: Høynivåspråk kan ha høyere abstraksjonskostnad og dermed lavere ytelse sammenlignet med lavnivåspråk. Dette kan være en ulempe når det gjelder systemintensive applikasjoner der ytelse er avgjørende. ● ○ Begrensete kontrollmuligheter: Høynivåspråk abstraherer bort

<p>utviklet med fokus på portabilitet og kan kjøres på forskjellige plattformer og operativsystemer uten større endringer.</p> <ul style="list-style-type: none"> • ○ Feilhåndtering: Høynivåspråk kan ha innebygde mekanismer for unntakshåndtering og feilmeldinger, som gjør feilsøking enklere og bidrar til mer robuste programmer. 	<p>detaljer i maskinvaren, noe som betyr at utvikleren har mindre kontroll over systemressurser og lavnivåfunksjoner.</p> <ul style="list-style-type: none"> •
--	---

C regnes som et programmeringsspråk på lavt nivå. Oppfunnet for mer enn et halvt århundre siden, har den fortsatt en sterk posisjon innen programvareutvikling: Mer enn 95 % av Linux-kjernen er implementert i C. Den er også dominerende i andre operativsystemer og innebygde systemer.

Språket C for operativsystem

Hva gjør C spesielt godt egnet for programmering av operativsystemer?

- **Ytelse:** C er et språk med lavt nivå som gir utviklere direkte tilgang til maskinvaren og lavnivåfunksjoner. Dette gjør det mulig å skrive kode som er optimalisert for ytelse, noe som er avgjørende i operativsystemer der ressurseffektivitet er viktig. C-kode kan utnytte maskinarespesifikke funksjoner og bruke minneeffektive teknikker som direkte minnetilgang og bitmanipulasjon.
- **Portabilitet:** C er et bærbart språk som kan kjøres på ulike maskinarkitekturen og operativsystemer med relativt få endringer. Operativsystemer må kunne kjøre på et bredt spekter av maskinvareplattformer, og C gir mulighet for slik portabilitet. C-kode kan skrives på en plattform og deretter kompileres for å kjøre på forskjellige systemer uten store tilpasninger.
- **Lavt nivåkontroll:** C gir utviklere direkte kontroll over maskinvarefunksjoner, minnehåndtering og systemressurser. Dette er viktig i operativsystemer, hvor detaljert styring av prosesser, tråder, minneallokering og I/O-operasjoner er nødvendig. C gjør det mulig å skrive kode som kan kommunisere direkte med maskinvaren og samhandle tett med operativsystemkjernen.
- **Eksisterende kodbaser og biblioteker:** C har vært i bruk i lang tid og har derfor en omfattende eksisterende kodbase og et bredt spekter av biblioteker og verktøy. Dette gjør det enklere å utvikle operativsystemer ved å dra nytte av eksisterende implementasjoner, standarder og ressurser.

Hvordan skiller lavnivå- og høynivåspråk seg i sin tilnærming til hukommelse?

Hvordan påvirker denne forskjellen deres brukbarhet for programmering av operativsystemer?

Lavnivåspråk som C gir direkte kontroll over hukommelsen, mens høynivåspråk som Java og Python har automatisk minnehåndtering. Denne forskjellen påvirker brukbarheten for operativsystemprogrammering ved at lavnivåspråk gir finjustert kontroll og optimalisering av minnebruk, mens høynivåspråk forenkler utviklingsprosessen, men kan ha mer overhead og mindre kontroll over minneadministrasjon.

Er det mulig å implementere et operativsystem på et språk på høyt nivå, slik som Lisp?

Hva med et annet språk på høyt nivå, for eksempel om Python? Oppgi grunnene dine.

Ja, det er teoretisk mulig å implementere et operativsystem på et språk på høyt nivå som Lisp eller Python. Språk på høyt nivå som Lisp og Python har utviklingsmiljøer og funksjonaliteter som gjør det mulig å håndtere systemnære oppgaver.

Likevel er det flere utfordringer som kan påvirke brukbarheten for operativsystemutvikling:

- **Ytelse:** Høynivåspråk kan ha høyere abstraksjonskostnad og overhead sammenlignet med lavnivåspråk. Dette kan resultere i lavere ytelse for operativsystemets kjernekomponenter som trenger å kjøre raskt og effektivt.
- **Minnehåndtering:** Automatisk minnehåndtering i høynivåspråk kan føre til uforutsigbarhet og overhead i ressursbruk. Operativsystemer krever ofte finjustert og kontrollert minnehåndtering for å optimalisere ressursutnyttelsen.
- **Plattformavhengighet:** Høynivåspråk er vanligvis mer plattformuavhengige, noe som betyr at de abstraherer bort maskinvarefunksjoner og -detaljer. Dette kan gjøre det vanskelig å oppnå direkte tilgang til maskinvaren og implementere operativsystemspesifikke funksjoner.
- **Systemressurser:** Operativsystemer krever dyp integrasjon med systemressurser som enheter, minnehåndtering og maskinvareinteraksjon. Høynivåspråk kan ha begrensninger når det gjelder slike ressurshåndteringsmekanismer

3.2 System-, bibliotekskall og skedulering

Spørsmål 1

Systemkallet `read()` forsøker å lese `count` bytes inn i bufferen pekt på av `buf` fra filen tilknyttet den åpne filendeskriptoren `fd`. Ved suksess returnerer den antall leste bytes. Ved feil returnerer den negativ én.

```
ssize_t read(int fd, void *buf, size_t count);
```

Som nesten alle systemkall i Linux, aktiveres read() gjennom en bibliotekfunksjon med samme navn. Det finnes også andre bibliotekfunksjoner som er avhengige av read(). Hvis dette virker forvirrende, les man-sidene man 2 read (systemkall) og man 3 read (biblioteksfunksjon).

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

Systemkall fungerer som inngangspunkter til kjernen, hvor selve «tunghøftingen» av for eksempel lesing av bytes fra en fil lagret på disken inn i RAM utføres. Kjernefunksjonen som tilsvarer read() er sys_read().

Hvis man ser bort fra forkunnskaper, finnes det noen generell måte å avgjøre om en signatur tilhører et systemkall eller en bibliotekfunksjon?

En generell måte å avgjøre om en signatur tilhører et systemkall eller en bibliotekfunksjon er å se på parametrene og returtypen i funksjonssignaturen.

Systemkall:

- Systemkall har vanligvis færre parametere og bruker spesifikke typer for å representere systemressurser eller operasjonstilkoblinger, som filendeskriptorer (fd) i Unix/Linux-systemkall.
- Returtypen er ofte en spesifikk type som indikerer status eller resultatet av systemoperasjonen, for eksempel ssize_t eller int.

Bibliotekfunksjon:

- Bibliotekfunksjoner kan ha flere parametere som kreves for å utføre en spesifikk funksjonalitet eller manipulere data på et høyere nivå.
- Returtypen kan variere og er ofte relatert til funksjonens formål eller oppgave.

Ved å vurdere antall parametere, typene av parameterne og returtypen kan man generelt skille mellom systemkall og bibliotekfunksjoner. Systemkall er vanligvis mer knyttet til lavnivåoperasjoner og interaksjon med kjernen, mens bibliotekfunksjoner tilbyr høyere abstraksjonsnivå og tjenester på et mer brukernivå.

Det er viktig å merke seg at dette er generelle retningslinjer, og det kan være unntak eller variasjoner avhengig av operativsystem og programmeringsspråk. I tvilstilfeller er det nyttig å konsultere dokumentasjonen for å få en klarere forståelse av en spesifikk funksjonsart.

Uansett hvordan det kan se ut på overflaten fungerer funksjonskall og systemkall forskjellig internt. Systemkall utsteder TRAP-instruksjonen for å gå inn i kjernen, mens funksjonskall

utsteder CALL-instruksjonen for å kalle en annen funksjon. På hvilke to grunnleggende måter er oppførselen til disse instruksjonene forskjellig?

Oppførselen til TRAP-instruksjonen (for systemkall) og CALL-instruksjonen (for funksjonskall) er forskjellig på to grunnleggende måter:

Privilegienivå:

TRAP-instruksjonen brukes til å gå fra brukermodus til kjerne-/systemmodus, som har høyere privilegienivå og tillater tilgang til ressurser som er begrenset for brukerprogrammer. Systemkall blir utført i kjerne-/systemmodus.

CALL-instruksjonen blir brukt til å kalle en funksjon innenfor samme modus (som vanligvis er brukermodus), der programmet kjører med begrensede rettigheter og tilgang til systemressurser.

Kontekstendring:

TRAP-instruksjonen for systemkall fører til en kontekstendring, der programflyten skifter fra brukermodus til kjerne-/systemmodus. Denne endringen innebærer at prosessens registerstatus og kjørekontekst blir lagret og gjenopprettet.

CALL-instruksjonen for funksjonskall fører ikke nødvendigvis til en kontekstendring. Den utfører et hopp til en annen funksjon og fortsetter utførelsen i samme modus uten å endre registerstatus eller bytte kontekst.

Generelt sett utsteder TRAP-instruksjonen (for systemkall) en overgang til en annen privilegienivå og utløser kontekstendring, mens CALL-instruksjonen (for funksjonskall) utfører et lokalt funksjonshopp innenfor samme modus uten å bytte privilegienivå eller endre kontekst. Disse forskjellene er essensielle for å skille mellom kjerneoperasjoner og vanlig funksjonskall i operativsystemet.

Kjernefunksjoner pakket inn i systemanrop pakket inne i bibliotekfunksjoner ... Hvorfor ikke spare bryderiet og kalle sys_read() direkte fra programmet? Er det mulig? I så fall, eller hvis det var det, ville stress spart virkelig vært større enn stress skapt?

Ja, det er mulig å kalle kjernefunksjonen sys_read() direkte fra et program uten å bruke mellomliggende bibliotekfunksjoner som read(). Dette krever imidlertid en dypere kunnskap om den underliggende operativsystemkjernen, inkludert hvordan systemanrop blir implementert og hvordan de skal brukes riktig.

Å kalle sys_read() direkte kan være nyttig i visse tilfeller, spesielt når man ønsker å omgå overheadet og ytelsesbortfallet som kan følge med å bruke bibliotekfunksjoner. Dette kan være relevant for spesialiserte applikasjoner eller situasjoner der man trenger finere kontroll over systemressursene eller ønsker å oppnå høyere ytelse.

Imidlertid er det noen viktige forhold man må ta i betraktning:

1. Plattformavhengighet: Kalle sys_read() direkte vil gjøre koden din avhengig av den spesifikke kjernen og dens grensesnitt. Dette kan begrense portabiliteten til programmet ditt.
2. Sikkerhet: Direkte kall til kjernefunksjoner kan bryte isolasjonen og sikkerhetsmekanismene som er implementert av operativsystemet. Det er viktig å ha en grundig forståelse av sikkerhetssammenhenger og mulige sårbarheter før man går videre med direkte kall.
3. Vedlikehold og lesbarhet: Å bruke bibliotekfunksjoner som read() kan bidra til lesbarhet og vedlikeholdbarhet i koden din ved å tilby et høyere nivå av abstraksjon og standardisering.

I de fleste tilfeller er det vanlig å bruke bibliotekfunksjoner som read() i stedet for å kalle kjernefunksjoner direkte. Dette sikrer portabilitet, enklere vedlikehold og bruk av et veldefinert grensesnitt. Kjernefunksjoner bør reserveres for spesifikke situasjoner der ytelse eller finere kontroll over systemressurser er avgjørende, og der man har nødvendig kunnskap og erfaring med å håndtere dem riktig.

Oppgave 2

3.2.1 Shortest job first

For ulike typer datamaskiner og forventet arbeidsmengde vil vi bruke skeduleringsalgoritmer med svært forskjellige egenskaper. Vennligst vurder følgende spørsmål.

Beskriv hvordan shortest job first (SJF)-skedulerer fungerer. Hva er fordelen med SJF-skedulering?

Fremgangsmåte:

1. Når en ny prosess kommer inn i systemet, blir den sammenlignet med kjøretiden til de eksisterende prosessene i køen.
 2. Prosessen med den korteste forventede kjøretiden blir valgt som neste prosess som skal utføres.
 3. Den valgte prosessen blir utført til den er ferdig eller blir avbrutt av en høyere prioritert prosess som kommer inn i køen.
 4. Prosessen som blir avbrutt, blir plassert tilbake i køen, og algoritmen velger igjen prosessen med kortest forventede kjøretid.
- Sortere oppgaver etter lengde, også når prosessen er klar til å bli utført så velger SJF-skedulering den som tar kortest tid
 - Fordelen med SJF-skedulering er at gjennomsnittlige tiden blir redusert når den prioritiserer først de korte oppgavene og utfører dem.

Fordel:

Fordelen med SJF-skedulering er at den har potensial til å gi lav gjennomsnittlig ventetid for prosessene. Ved å prioritere de korteste jobbene først, kan man redusere gjennomsnittlig

ventetid og øke gjennomstrømningen i systemet. Dette kan være spesielt gunstig når det er mange små jobber eller kortvarige oppgaver som trenger å bli utført raskt.

Hvorfor brukes ikke SJF-skedulering som den primære skeduleringsstrategien for stasjonære datamaskiner som brukes interaktivt for typisk kontorarbeid?

- Forutsigbarhet: SJF-skedulering krever at systemet har fullstendig informasjon om forventet kjøretid for hver prosess. Dette kan være vanskelig å estimere nøyaktig, spesielt for interaktive applikasjoner der brukerinteraksjon og varierende oppgaver kan påvirke kjøretiden. Det er ofte vanskelig å forutsi hvilke prosesser som er kortest eller lengst.
- Blokkering av prosesser: Hvis en lengre prosess kommer først i køen, kan den kortere jobben bli blokkert og må vente lenge før den blir utført. Dette kan føre til en forsinkelse i responsid for interaktive applikasjoner, noe som ikke er ønskelig i et kontormiljø der brukere forventer umiddelbar respons på handlinger.
- Sulting av prosesser: SJF-skedulering kan føre til "sulting" av lengre prosesser, der de blir utkonkurrert av kortere prosesser og aldri får en sjanse til å bli utført. Dette kan være ueffektiv hvis det er viktige oppgaver eller prosesser med lengre kjøretid som må fullføres.
- Ikke godt tilpasset interaktive applikasjoner
- Problemer hvis noe skal starte opp, pga. Det er en tung oppgave
- Starvation og lang ventetid

Du administrerer en server som er ment for stor, beregningstung arbeidslast uten interaktivitet. Forklar hva slags scheduleringsalgoritme du bruker på denne serveren og hvorfor.

På en server som er ment for stor, beregningstung arbeidslast uten interaktivitet, kan en passende scheduleringsalgoritme være en variant av Round Robin (RR) skedulering eller en annen form for timebasert skedulering.

Round Robin-algoritmen er kjent for å være rettferdig og gi lik tidsdeling til hver arbeidsoppgave. Den deler tiden jevnt mellom alle tilgjengelige oppgaver i køen og gir en forutsigbar og jevn ressursallokering. Dette er spesielt nyttig når man har mange beregningstunge oppgaver som trenger å bli utført på serveren.

Ved å bruke en RR-algoritme kan man unngå situasjoner der enkelte oppgaver tar opp for mye tid og monopoliserer serverressursene. Det gir også muligheten til å prioritere oppgaver ved å justere tidskvantumet for hver oppgave i henhold til deres viktighet eller ressursbehov.

En time-basert skedulering, der oppgaver får en bestemt tidskvote for å utføre beregninger før de blir byttet ut, kan også være passende for denne serveren. Dette gir forutsigbarhet og kontroll over bruk av ressurser og sikrer at alle oppgaver får sin tildelte andel av serverkapasiteten.

Valget av scheduleringsalgoritme vil avhenge av serverens spesifikke behov og de forventede egenskapene til arbeidslasten. Det er også viktig å overvåke og justere

skeduleringsparametrene basert på ytelsesanalyse og tilbakemeldinger for å optimalisere serverens ressursutnyttelse og effektivitet.

Oppgave 3

3.2.2 Diskutnyttelse

Interaktive programmer krever en form for preemptiv skedulerer for å tilby god ytelse. Som et hobbyprosjekt jobber du med et operativsystem for arbeidsstasjoner. For å gjøre det enkelt har du valgt skeduleringsalgoritmen Round Robin. Figur 2 viser utbruddsmønstrene for to typiske prosesser som kan kjøre på systemet: én CPU-bunden og én I/O-bunden. Tiden mellom utbruddene brukes på diskoprasjoner.



Hvordan vil lengden på tidsintervallet påvirke diskutnyttelse? Vurder intervaller på henholdsvis 50 ms og 1 ms. Det later til at kortere intervaller sørger for bedre diskutnyttelse. Hvorfor ikke gjøre intervallene så korte som mulig?

Diskutnyttelse: Med lengre tidsintervaller, for eksempel 50 ms, vil hver prosess få tildelt en lengre kontinuerlig kjøretid på CPU-en før den byttes ut. Dette kan være gunstig for diskutnyttelsen, da prosessen kan utføre flere diskoprasjoner sammenhengende uten avbrudd. Dette kan føre til mer effektiv bruk av diskressursene og redusert overhead knyttet til bytte av diskoprasjoner.

Kontekstbytteoverhead: Med kortere tidsintervaller, for eksempel 1 ms, vil kontekstbytte-overhead øke. Jo hyppigere kontekstbytte skjer, desto mer tid brukes på å bytte mellom prosessene i stedet for å utføre prosessene. Dette kan resultere i redusert effektivitet og økt overhead knyttet til selve skeduleringsprosessen.

Å gjøre tidsintervallene så korte som mulig kan føre til hyppige kontekstbytter og høyere overhead, spesielt hvis prosessene er CPU-bundne og ikke trenger hyppige skifter. Dette kan føre til tap av ytelse og ressurser som brukes til skeduleringsaktiviteter i stedet for reell prosessutførelse.

Det er derfor en balanse mellom tidsintervallene og kravene til diskutnyttelse. Kortere intervaller kan være fordelaktige når det er behov for rask respons og interaktivitet, spesielt for interaktive programmer. Imidlertid kan det være ineffektivt å ha altfor korte intervaller for ikke-interaktive, CPU-bundne prosesser som ikke har behov for hyppige skifter.

Det er viktig å vurdere prosessenes natur og oppgavene de utfører, sammen med ytelsesanalyse og testing, for å finne en passende balanse mellom tidsintervaller og skeduleringsbehovene til systemet.

3.2.3 Mål for diskutnyttelse

Hvilke andre mål bør veies mot maksimal diskutnyttelse når man velger skeduleringsalgoritme for et slikt operativsystem?

Når man velger skeduleringsalgoritme for et operativsystem, bør man veie flere mål og faktorer mot maksimal diskutnyttelse. Her er noen viktige mål som bør vurderes:

- Responsivitet: Responsivitet refererer til hvor raskt systemet kan svare på brukerinteraksjoner eller forespørsler. Et interaktivt operativsystem bør prioritere rask respons og lav ventetid for brukeren. Dette kan kreve bruk av preemtiv skedulering for å tillate prioritering av interaktive prosesser eller brukergrensesnittet.
- Rettferdighet: En rettferdig skeduleringsalgoritme sikrer at alle prosesser får tilstrekkelig tid og ressurser til å utføre oppgavene sine. Det er viktig å unngå at noen prosesser blir "sultet" eller utkonkurrert av andre. Rettferdig fordeling av ressurser er spesielt viktig i flerbruksystemer eller når det er behov for å støtte flere parallele oppgaver.
- Latens: Latens er tiden det tar for en oppgave å starte eller fullføre. En lav latens er viktig for oppgaver som krever umiddelbar respons, som sanntidsapplikasjoner eller systemer med tidskritiske operasjoner. En skeduleringsalgoritme som reduserer latensen og gir rask oppstart av oppgaver kan være foretrukket.
- Skalering og ytelse: Skeduleringsalgoritmen bør skalere godt med økende antall prosesser eller kjerneenheter. Den bør kunne opprettholde god ytelse selv under høy arbeidsbelastning eller på systemer med mange kjerner.

3.3 Heap og stack, partisjonering

Oppgave 1

De fleste moderne operativsystemer utruster prosesser med privat og partisjonert minne. Dette sikrer at prosesser ikke forstyrre hverandres data; det holder også forskjellige typer data atskilt i hver prosess. Segmentene er vanligvis navngitt som følger, med antydning til bruken: systemdatasegmentet (også kjent som prosesskontrollblokken (PCB)), kodesegmentet (eller tekstsegmentet), det initialiserte datasegmentet (eller bare datasegmentet), det uinitialiserte datasegment (også kjent som BSS-segmentet), heapen og stakken.

Vanligvis er et program først lagret på disk, hvorfra det bringes inn i minnet. Vil du argumentere for at kodesegmentet inneholder (deler av) programmet eller prosessen – eller begge to?

Bare kodesegmentet i et program inneholder vanligvis selve programmet, som består av en serie instruksjoner og data som utfører spesifikke oppgaver. Disse instruksjonene er oversatt fra kildekode til maskinkode og består av en rekke instruksjoner som CPU-en kan utføre for å utføre ønskede operasjoner.

Heapen og stakken vokser mot hverandre og motsatte ender av adresserommet (ettersom data lagres til dem). Hva er hensikten med denne ordningen? Hva skjer hvis rommet mellom dem blir oppbrukt?

Hensikten med at heapen og stacken vokser mot hverandre og på motsatt ende av adresserommet er for å unngå kollisjoner mellom de to datastrukturene. Minnet vil kunne bli

utnytte på en optimal måte og risikoen for kollisjoner vil minimeres. Hvis rommet mellom dem blir oppbrukt vil det ikke være mer plass til å alllokere variabler på stacken eller mer data på heapen. Det kan føre til en "stack overflow" eller "heap overflow", som igjen fører til at programmet krasjer eller oppfører seg unormalt.

**Heapen og stakken inneholder begge prosessens kjøretidsdata. Å velge hvilken som bør brukes når er ikke alltid åpenbart for programmereren.
Hvordan drar en prosess nytte av disse to minnestrukturer? Ville bare én være tilstrekkelig?**

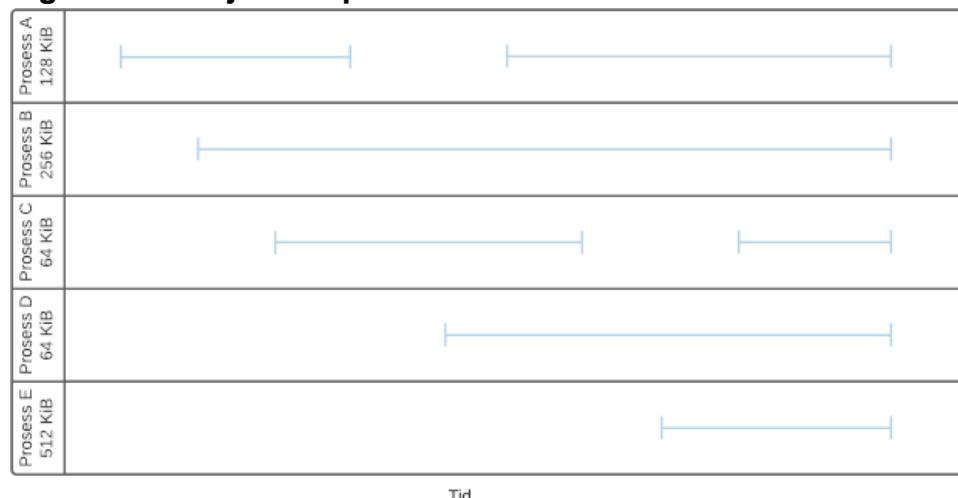
I datasegmentet starter heapen på en lavere minneadresse, og utvider seg mot en høyere adresse. Stacken starter på den høyeste minneadressen og utvider seg mot lavere adresse. Begge er dynamisk minne. Stacken lagrer parametere og i større grad lokale variabler, i tillegg til at den holder styr på register. Heapen holder i større grad på globale og statiske variabler. Altså holder Heapen på "tyngre" data og er tregere enn stacken. Altså er hovedgrunnen til at vi bruker det at de vokser dynamisk mot i hverandre og at de holder på ulike ting.

Spørsmål 2

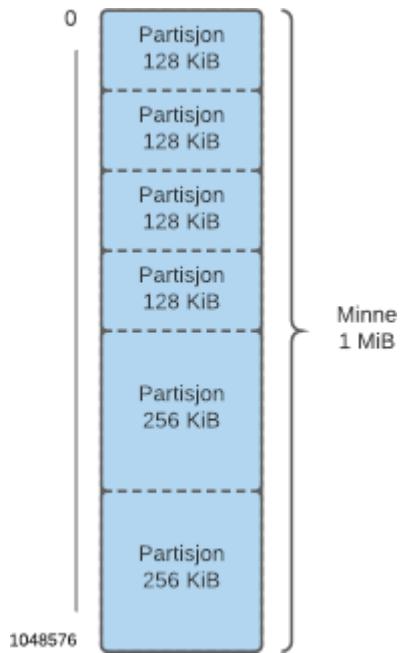
3.3.1 Oppgave om partisjonering

Fem programmer kjører på et meget lite system, som en værstasjon til hjemmebruk, med beskjedne 1 MiB minne. Når Program A eksekveres, kjører det i en prosess som tar opp 128 KiB. For enkelhets skyld kaller vi det Prosess A hver gang det kjører. Når Prorgam B eksekveres, tar dets prosess opp 256 KiB. Prosesser for C og D tar opp 64 KiB. Prosesser for E tar opp 512 KiB.

Figur 1 viser kjøretidsplanen.



Det finnes flere metoder for å tildele prosesser minne. Dette er en viktig del av minnehåndteringen. Tre slike metoder behandles her: statisk partisjonering (med partisjoner av ulik størrelse), dynamisk partisjonering og buddy-tildeling. Segmentering anvendes ikke; én prosess må få plass innenfor én partisjon.



Ta i betraktning den statiske partisjoneringsplanen vist i Figur 2. Hvordan er prosessene plassert i minnet like før slutt? Hva skjer med Prosess E? Hva slags og hvor mye fragmentering forekommer?

Ved en statisk partisjoneringsplan vil prosessen være plassert i følgende rekkefølge i minnet like før slutt: D, C, A, B. Prosess E vil ikke kjøre på grunn av at størrelsen til E på 512 KiB ikke passer inn i noen av partisjonene. Det vil oppstå intern fragmentering som følge av at prosess C og D tar opp 64 KiB, mens partisjonene de blir tildelt er på 128 KiB.

Minnelayouten uten forhåndsbestemte partisjoner. Dette støtter dynamisk partisjonering. Hvordan er resultatet annerledes når dynamisk partisjonering brukes med hver av tildelingsalgoritmene First Fit, Next Fit og Best Fit? Hvilken mekanisme kan hjelpe Prosess E? Når man velger en tildelingsalgoritme, hvilke andre kriterier – foruten fragmentering – er verdt å ta i betraktning?

First Fit vil legge prosesser inn der det er plass først, og rekkefølgen vil derfor bli: D, C, B, A og E får ikke plass. Next Fit legger prosesser inn etter siste som ble lagt inn, og rekkefølgen blir: B, D, A, C. Best-Fit vil legge prosesser inn der de passer best, og gi: D, C, B, A og ikke gi plass nok til E pga. Fragmentering. På grunn av størrelsene til prosessene og kjøretidsplanen vil derfor Next-Fit tildelingsalgoritmen hjelpe prosess E.

Andre kriterier som er verdt å ta til betraktnign når en skal velge en tildelingsalgoritme er effektivitet, slik at tildeling skjer raskt og bruker minimalt med minne. En kan også ta fleksibilitet og robusthet til betraktnign, da dette kan være viktig hvis det kommer unøyaktige inputer, eller andre feil oppstår underveis.

Buddy-tildeling henter inspirasjon fra både statisk og dynamisk partisjonering – som en hybrid mellom dem. Hvilket resultat gir buddy-tildeling?

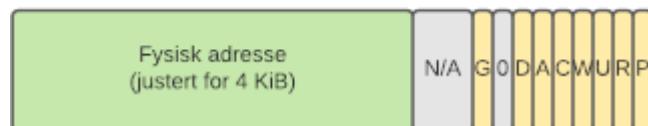
Med Buddy-tildeling vil vi få resultatet: A, C, D, B, E.

Spørsmål 3

3.3.2 Oppgave om paging

Som teknikk løser virtuelt minne visse problemer, men introduserer også noen nye. Et abstraksjonslag legges over fysisk minne, slik at applikasjoner (eller snarere de som skriver dem) ikke behøver å bry seg med de rent fysiske detaljene ved minnehåndtering; operativsystemet gjør den tunge jobben i så måte. Sidetabellen, som knytter virtuelle adresser til fysiske adresser, brukes for å utføre denne funksjonen. Den bærer likevel med seg en viss kostnad i form av tid og rom: Sidetabellen må nødvendigvis lagres noensteds, mens sidetabelloppslag forbruker CPU-sykler.

31 ————— 12 11—9 8 7 6 5 4 3 2 1 0



G – Global; D – Dirty; A – Accessed; C – Cache Disabled;
W – Write-Through; U – User; R – Read; P – Present

En sidetabelloppføring på IA-32-arkitekturen er vist i Figur 1. Gitt at sidestørrelsen er 4 KiB, hvor stor er sidetabellen? (Husk at antall sider kan utledes fra sidestørrelsen.)

Med IA-32-arkitekturen inneholder en sidetabelloppføring som regel informasjon om en side med virtuelt minne. Størrelsen på sidetabellen kommer an på antallet virtuelle sider i systemet, som bestemmes av størrelsen på hver side. Hvis sidestørrelsen er 4KiB er altså hver virtuelle side 4KiB stor. Antall sider kan utledes fra sidestørrelsen ved å dele størrelsen på adressrommet med sidestørrelsen. Med IA-32-arkitekturen er det virtuelle adressrommet på 32 bit, altså 4GiB. Dermed får vi $4\text{GiB} / 4\text{KiB} = 2^{20}$ sider. Vi trenger da en tabell med 2^{20} innganger, hvor hver inngang er 4 byte stor. Vi ganger 2^{20} med 4 byte og får at sidetabellen vil være 4MiB med en sidestørrelse på 4KiB.

Hvis det viser seg at sidetabellen ikke er overdrevent stor, hvorfor ikke lagre alt dets innhold i minnet til enhver tid? Med andre ord: Hvorfor bry seg med multilevel paging i utgangspunktet? Hvis det behøves, kan ikke sidetabellen simpelthen swappes ut til disk?

Selv om sidetabellen kanskje ikke er overdrevent stor, kan det likevel være ineffektivt å lagre alt dets innhold i minnet til enhver tid. Dette skyldes at selv om systemet har nok minne til å holde hele sidetabellen, vil det fortsatt være behov for plass til andre operativsystemprosesser og programmer som kjører samtidig. Multilevel paging er en metode som tillater at den virtuelle adressen oversettes til en fysisk adresse ved hjelp av flere nivåer med sidetabeller. Dette gir fleksibilitet og effektivitet i minnehåndteringen ved å tillate at deler av sidetabellen kun blir lastet inn i minnet når det er nødvendig. Systemet kan dermed bruke mindre minne når sidetabellen ikke er i bruk. Når det gjelder å swappe ut

sidetabellen til disk, kan det gjøres om nødvendig, men dette kan være en tidskrevende prosess som i betydelig grad kan svekke ytelsen. Dette er fordi at å swappe sidetabellen til disk i utgangspunktet involverer å kopiere hele tabellens innhold til disk og deretter lese det tilbake inn i minne når det trengs.

Forandrer multilevel paging situasjonen når det gjelder å swappe ut sidetabellen? Hvordan i så fall?

Multilevel paging kan gjøre swapping av sidetabellen mer effektiv. Ettersom multilevel paging tillater systemet å håndtere større adresserom ved kun å ha de nødvendige delene av tabellen i minnet til en hver tid, reduseres behovet for å swappe hele sidetabellen til disk. Det vil da bare være de unødvendige delene av tabellen som må swappes, ikke hele tabellen.

3.4 Disk

Oppgave 1

3.4.1 Oppgave om sekundærminne

Tilgang til sekundær lagring som mekaniske harddisker tar mye tid sammenlignet med tilgang til minne i RAM. Forklar kort hvorfor og hvilke operasjoner tilgangsforsinkelsen består av

Seek time: Flytte diskhodet

Rotasjonsforsinkelse: Vente til platen har rotert så mye slik at biten ligger under hodet. Tiden det tar for disken å rotere til ønsket bit ligger under diskhode.

Transfertime: Tiden det tar for diskhodet å lese av dataen

Andre delays

Husk å nevne hvorfor problemene oppstår

Hva gjøres vanligvis for å forbedre ytelsen til mekaniske disker i dag når det gjelder tilgangsforsinkelser og dataoverføringer? Nevn 2 tilnærmingen og forklar dem

For å effektivisere bruk av sekundær lagring kan vi benytte oss av "Disk Schedulering" eller benytte seg av flere disker.

Søketid er en av de dominante faktorene til diskens totale I/O tid, et mål med "Disk Schedulering" er å oppnå kortere responstider, høy gjennomstrømning og rettferdighet. Iddet her er å la OS eller disk kontrolleren avgjøre hvilken rekkefølge den utfører forespørsler om å lese/endre data avhengig av diskhodets plassering. Det finnes mange ulike algoritmer for dette, feks. SCAN og FCFS.

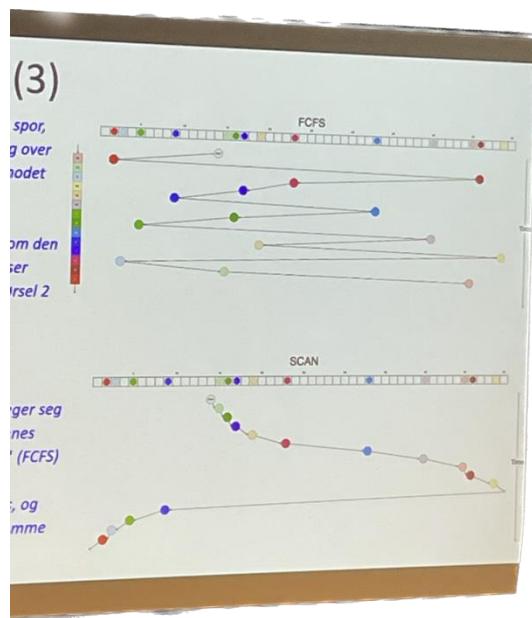
Å benyttes seg av flere disker forbedrer ytelsen på mange måter. Avhengig av hva slags egenskaper man ønsker kan man benytte seg av forskjellige leveler av Redundant Array of Independent Disks (RAID). Eksempel til dette er RAID 0 (striping). RAID 0 fordeler dataen utover flere disker slik at man kan få n (antall drivere) ganger høyere dataoverføringshastighet. Et annet eksempel er RAID 3 som benytter seg av "interleaved

parity". At den er "interleaved" betyr at den benytter seg av striping i subgrupper som minst 3 disker.

La oss anta at vi har en (veldig liten) disk som har 50 spor, nummerert fra 0 til 49, og som diskhodet beveger seg over og leser data fra. Anta at på et gitt tidspunkt er diskhodet plassert over og leser data fra spor 14. Etter denne forespørselen er følgende forespørsler i diskscheduleringskøen (hvert nummer angir sporet som den forespurte datablokken er lagret på, og sekvensen viser rekkefølgen forespørslene ankom i køen, dvs. forespørsel 2 er den første som ankom systemet):

45 15 3 49 19 40 5 16 33 9 17 23 46 2

Lag en grafisk figur som viser hvordan diskhodet beveger seg over de forskjellige sporene (i hvilken rekkefølge betjenes forespørslene) hvis vi bruker "First-Come-First-Serve" (FCFS) og SCAN (hodet beveger seg for øyeblikket fra lavt nummerert spor til høynummererte spor), henholdsvis, og forutsetter at alle forespørsler i køen kan betjenes i samme runde.



Oppgave 2

3.4.2 Filsystem

Den fysiske representasjonen av data, uansett medium, er sjeldent så ren og ryddig som det kan se ut som for applikasjonen eller applikasjonsprogrammereren. Når det gjelder disklagring, er det filsystemet som gir logikken som forvandler det underliggende «kaoset» til orden.

Bortsett fra å tildele lagringsplass til filer, hva er noen andre viktige oppgaver som filsystemet håndterer?

Filsystemet håndterer også fil-lagringen som feks. Hvem som har aksess, eier, modifiserer og leser og skriver. Håndheve at bare de som har rettigheter får tilgang til filen. I tillegg passer filsystemet på at filene ikke blir korrupt.

Top-down view betyr å se på filsystemet fra et overordnet perspektiv, hvor man starter med den øverste nivåstrukturen og deretter dykker nedover i underkataloger og filer. Dette perspektivet fokuserer på hierarkiet og strukturen til filsystemet, og hvordan mapper og filer er organisert innenfor forskjellige kataloger.

Bottom-up view betyr å se på filsystemet fra et detaljnívå og bygge oppover. Det innebærer å starte med individuelle filer eller objekter og deretter gruppere dem i mapper og kataloger basert på forskjellige kriterier. Dette perspektivet kan være nyttig når man ønsker å organisere filer basert på egenskaper, metadata eller andre attributter.

Begge synspunktene kan være nyttige avhengig av situasjonen eller målet. Top-down view er vanligvis mer brukt når man ønsker å få en oversikt over filsystemstrukturen, navigere hierarkisk og finne bestemte filer eller mapper. Bottom-up view kan være nyttig når man ønsker å organisere filer basert på spesifikke kriterier eller attributter, som for eksempel filtrering etter filtype eller metadata.

3.4.3 EXT4

Et mye brukt filsystem er EXT4 på Linux, som bruker inoder for lagring av metadata. Forklar hvordan EXT4 kan støtte svært store filer.

EXT4 benytter seg av en tilnærming for håndtering av filblokk, kalt "extent-basert allokering", for å lagre store filer. Det innebærer at store filer blir lagret i en eller flere kontinuerlige blokker i lagringsenheten, som kalles extents. Dette reduserer fragmentering og gjør at filene er raskere å aksessere, siden de ikke er spredt rundt i lagringsenheten som individuelle blokker. Mengden metadata som må opprettes reduseres også siden filsystemet bare trenger å opprette for hver enhet

Kan ha extents av forskjellige størrelser. Var litt uklar besvarelse rundt extents. En syntes ja og en nei. Ville hatt med at et extent er en mekanisme for å slå sammen flere tradisjonelle diskblokker til et større segment. Samling av blokker som ligger inntil hverandre på disken, som kan leses sammen.

Du har til oppgave å designe et system som skal lagre store datamengder, og som også krever høyst tilgjengelig og redundant lagring. Velg en teknologi som tjener disse formålene og forklar hvordan den gjør det.

Et system som kan lagre store datamengder og som også krever høyst tilgjengelig og redundant lagring er feks. RAID 5. Denne er en redundant array som brukes diskstriping med paritet. Striping er en metode for å få høyere overføringsrate ved å bruke flere disker for leting av en fil og lese de parallelt. RAID 5 stripere alle diskene bortsett fra en disk, den siste disken brukes til å lagre en parity disk. En parity disk sikrer at dataene som er lagret på diskene overføres riktig og hjelper til med å rekonstruere manglende data basert på informasjonen fra andre disker. Krever minst 3 disker.

Oppgave 3

Fra brukerens perspektiv fremstår en fil som en enkelt, sammenhengende enhet.
Dette er imidlertid sjeldent hvordan blokkene er lagret på disk. En fils blokker kan spenne over flere sektorer, spor og plater, sammenflettet med annen ledig og brukt diskplass. Det er filsystemet som sørger for nødvendig mapping mellom blokker og filer.

Hvor er selve filsystemet lagret?

Filsystem er som regel plassert på en spesifikk partisjon av disken. Den nøyaktige posisjonen er avhengig av hvilket filsystem som er brukt. Ofte er det slik at filsystemer legge på spesielle posisjoner slik at OS og systemer generelt under oppstart skal vite hvor de skal lete etter.

Hvis filsystemet ødelegges, går alle data på disken – i det minste i betydningen lesbare filer og kataloger) – tapt. Hva kan gjøres for å redusere denne sårbarheten?

Bruk **antivirusprogram** og hold operativsystemet oppdatert: Antivirusprogrammer kan redusere risikoen for at innstengere skader filsystemet ditt. Det er viktig å holde operativsystemet og annet programvare du bruker oppdatert, ettersom det stadig avdekkes sårbarheter som rettes opp av leverandøren.

Skylagring: Ved å lagre dataen din på skytjenester vil du enkelt kunne hente dataen igjen hvis dine fysiske enheter blir ødelagt.

Sikkerhetskopi: Ta regelmessige sikkerhetskopier av filsystemet ditt og lagre det på en ekstern trygg enhet, eller gjennom skylagring.

Kjeding i media og kjeding i kart er to teknikker for å holde oversikt over hvilke blokker som tilhører hvilke filer. Sammenlign disse teknikkene og forklar hvordan sistnevnte forbedrer effektiviteten for tilfeldig aksess.

Kjeding i media bruker pekere til å lenke sammen blokke. Når en fil lagres på disken, blir enhver blokk i filen lenket sammen i en lenket liste, som gjør det mulig for operativsystemet å traverse gjennom filen.

Kjeding i kart bruker derimot block allocation map (BAM) for å holde styr på hvilke blokker som tilhører hvilke filer. BAM gir en oversikt over om en blokk er ledig eller om den er allokerert til en fil.

Kjeding i kart forbedrer effektiviteten for tilfeldig aksess fordi den er raskere å slå opp i en map, enn det å traversere i en lenket liste. I tillegg er det lettere for operativsystemet å finne ledige blokker i en map, ettersom det slipper å traversere

3.5 Signaler, avbrudd

Oppgave 1

Signaler brukes til å utløse bestemt atferd i en prosess, for eksempel avslutning eller feilhåndtering. De utgjør en enkel måte å samhandle med prosesser som er felles for alle UNIX-lignende operativsystemer, som OpenBSD, Linux og MacOS. (Windows bruker ikke signaler, i hvert fall ikke den POSIX-kompatible typen, men bruker andre former for interprosesskommunikasjon for å oppnå de samme målene.)

Du har allerede lært om avbrudd og unntak i forbindelse med CPU. Funksjonelt ligner signaler avbrudd ved at begge får prosessen til å avvike fra dens normale kjøreflyt. Hva er de viktigste forskjellene mellom dem?

Signaler er programvare avbrudd levert til en prosess av OS-et eller av en annen prosess, mens avbrudd er maskinvare signaler som genereres av eksterne enheter som for eksempel mus eller tastatur, eller av prosessoren selv. Avbrudd kan brukes for å gjøre obs på at en prosess krever umiddelbar oppmerksomhet feks. I ferdigstilling av I/O-operasjoner

Altså er signaler en mekanisme for internprosess-kommunikasjon innad i et system, mens avbrudd er en mekaniske for kommunikasjon mellom maskinvare og programvare.

Vanligvis kan en kjørende prosess tilknyttet terminalen avsluttes ved å trykke Ctrl+C på tastaturet. Dette sender SIGINT-signalet til prosessen. Imidlertid kan prosessen håndtere dette signalet på en annen måte og dermed overstyre standardoppførselen, som er å avslutte. Det samme gjelder SIGTERM, som vanligvis ikke har en hurtigtast. Generelt kan signal-systemanropet brukes til å sette egne signalbehandlere, som er vanlige funksjoner, på bestemte signaler. Er det på denne måten mulig å lage en prosess som nekter å dø av noe signal? Forklar svaret ditt.

Selv om det er mulig å lage en signalbehandler som ignorerer eller håndterer signaler på en annen måte enn standardoppførelsen, kan ikke en prosess nekte å dø av signaler. Dette skyldes at noen signaler ikke kan ignoreres eller håndteres av en signalbehandler. For eksempel vil SIGKILL og SIGSTOP alltid føre til at prosessen blir drept eller stoppet, selv om det er satt opp en signalbehandler for disse signalene. Dette er fordi disse signalene er ment å være uoverstigelige og skal brukes i kritiske situasjoner hvor en prosess må stoppes umiddelbart.

Etter å ha kalt på funksjonen som er egendefinert vil også maskinen gå tilbake til standard håndteringen av signalet. Derfor må man redefinere funksjonene hver gang de skal brukes. Så i teorien vil det være mulig hvis man lager evige mange funksjoner, men dette går ikke i praksis.

Alice har oppdaget at en segmenteringsfeil faktisk får kjernen til å sende SIGSEGV-signalet til den skyldige prosessen. Dette signalet, som andre, kan overstyres. Prosessen trenger ikke krasje, argumenterer hun: Når signalet mottas, kan signalbehandleren ganske enkelt la prosessen fortsette sin glade

ferd. Fra nå av vil segmenteringsfeil være en bekymring fra fortiden. Bob er skeptisk. Han vet ikke helt hvordan han skal sprekke Alices buble, men synes ideen høres for god ut til å være sann. Hvordan vil du argumentere?

Når en prosess mottar et SIGSEGV-signal, betyr det at den har forsøkt å aksessere minne som den ikke har tillatelse til å aksessere eller som ikke eksisterer. Dette kan forårsake uforutsigbar oppførsel i prosessen og kan til og med korrumpe dens tilstand eller krashe den. Å ignorere eller overstyre SIGSEGV-signalet i signalhåndteren kan forhindre at prosessen crasher, men det løser ikke det underliggende problemet med at prosessen forsaker å aksessere ugyldig minne.

I tillegg kan det være farlig å overstyre SIGSEGV-signalet, da det kan føre til at prosessen fortsetter å kjøre i en corrupt eller udefinert tilstand. Dette kan føre til vanskelig feil og sikkerhetssårbarheter.

Oppgave 2

Når du utveksler data mellom tråder, må du beskytte de kritiske områdene i koden for å unngå inkonsistens. En C-programmerer gjør dette ved å lage en mutex-variabel (gjensidig ekskludering), som låses når koden går inn i det kritiske området og låses opp når koden forlater den. Funksjonskallene er pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock og, til slutt, pthread_mutex_destroy. Forelesningen har vist at du kan også dele minne mellom prosesser. For å beskytte kritiske koderegioner i forskjellige prosesser, kan du ikke bruke mutexen av typen ovenfor. Slik kode må bruke semaforer i stedet. For å lage en slik semafor kan man enten bruke sem_init eller sem_open, for å minske en semafor, "sem_wait" brukes og "sem_post" for å øke den. Når en prosess ikke lenger trenger semaforen bruker den "sem_close", og den kan til slutt bli slettet med "sem_unlink".

Med kunnskap om at kritiske områder kan beskyttes mellom tråder og mellom prosesser, kan vi bruke enten delt minne mellom tråder eller mellom prosesser. Det er fortsatt forskjeller mellom tilgjengeligheten av minne. Vennligst oppgi 2 viktige forskjeller og forklar hvorfor de er viktige.

En mutex (forkortelse for mutual exclusion) er en programmeringsmekanisme som brukes for å sikre at bare en tråd eller prosess kan få tilgang til en ressurs av gangen. Mutexen fungerer som en lås som styrer tilgangen til en delt ressurs og unngår situasjoner der flere tråder eller prosesser prøver å endre eller lese fra ressursen samtidig.

Addresserom er en viktig forskjell fordi den påvirker enkelheten i å dele data mellom tråder og prosesser. Med delt minne mellom tråder kan data deles ved å aksessere de samme minnelokasjonene. Men med delt minne mellom prosesser må data plasseres i et delt minneområde. Dette gjør internprosesskommunikasjon mer kompleks og potensielt treigere.

Minnebeskyttelse er viktig fordi den påvirker sikkerhet og stabiliteten til systemet. Hvis en tråd aksesserer en annen tråds minne ved et uhell, kan den ødelegge tilstanden til hele prosessen. I motsetning til dette, hvis en prosess aksesserer et annet prosessminne, er det vanligvis et sikkerhetsbrudd og kan forhindres av operativsystemet.

Det er mulig å bruke pipes med mer enn 2 prosesser. For eksempel hvis en serverprosess distribuerer arbeid til så mange like klientprosesser som datamaskinen har CPU-kjerner, og tiden disse klientprosessene bruker for å fullføre dette arbeidet varierer mye, kan det være lurt å bruke en enkel pipe der serveren skriver nye arbeidsoppgaver til pipen og en av klientprosessene som for øyeblikket er inaktiv leser en av dem fra pipen. Denne operasjonen inneholder et kritisk område som du selv må beskytte. Forklar hvorfor det er tilfelle og hvordan du kan løse det.

Det er et kritisk område når serverprosessen skriver til pipen og en klientprosess leser fra den, fordi hvis flere klientprosesser leser fra pipen samtidig, kan de lese samme oppgave, noe som kan føre til at noen blir oversett eller dobbeltbehandlet. Derfor må serverprosessen sørge for at bare en klientprosess leser fra pipen om gangen.

For å løse dette problemet kan man bruke synkroniseringsteknikker som mutex. Før en klientprosess får tilgang til pipen, må den skaffe seg en lås på mutexen, som er knyttet til pipen. Hvis låsen allerede holdes av en annen klientprosess, vil forespørselen bli blokkert til låsen blir frigjort.

Når en klientprosess har skaffet seg låsen, kan den lese fra pipen og behandle arbeidsenheten. Etter behandlingen frigjør klientprosessen låsen slik at andre prosesser kan få tilgang til pipen.

Mange Unix-operativsystemer gjør det mulig å lage et filsystem i RAM. Selvfølgelig forsvinner filer i disse filsystemer under en omstart, men hvis et program trenger midlertidige filer er det mye raskere å lagre disse filer i et slikt filsystem. Du kan bruke semaforer for å beskytte samtidig tilgang. Gi argumenter for og imot å bruke slike veldig raske midlertidige filer for delt-minne-kommunikasjon mellom prosesser.

For:

1. Hastighet: Siden RAM-baserte filsystemer lagres i minnet, gir de raskere lese- og skrivehastigheter sammenlignet med tradisjonelle diskbaserte filsystemer. Dette kan være spesielt gunstig for midlertidig filer som må brukes ofte av flere prosesser.
2. Sikkerhet: Siden filene i et RAM-basert filsystem lagres i minnet, er de ikke sårbare mot angrep rettet mot diskbaserte filsystemer. Dette kan gjøre dem til et mer sikert alternativ for å lagre midlertidige filer som inneholder sensitiv data.

Mot:

1. Begrenset lagring: Siden RAM er en begrenset ressurs, vil mengden lagring tilgjengelig i et RAM-basert filsystem også være begrenset. Dette betyr at filsystemet kanskje ikke er egne for å lagre store mengder data eller for langsiktig lagring.
2. Volatilitet: RAM-baserte filsystemer er ustabile, noe som betyr at dataene som er alget i dem vil gå tapt hvis systemet slås av eller hvis strømmen går. Dette kan være en stor ulempe hvis dataen er viktig og må bevares.

Spørsmål 3

Du har allokeret og delt en stor blokk med minne mellom to prosesser ved å bruke `shmget` og `shmat` kallene. Du tenker på denne store minneblokken som en lang array som består av elementer som dette: `struct element { char unused; char message[15]; struct element* next; };` Nå vil du implementere en lenket liste som består av små strukturer i denne delte minneblokken. Til å begynne med setter du "unused" for hvert element i arrayet til 1 (for å bety "true"). Når du ønsker å legge til et nytt element til listen din, finner du et ubrukt element, setter `unused=0`, fyller ut `message` og legger det til. Når du fjerner et element fra listen, setter du `unused=1` igjen. En av prosessene produserer data og legger til elementer i slutten av listen, den andre prosessen fjerner elementer fra starten av listen og forbruker dem (så listen er et eksempel på det svært viktige "producer-consumer-queue"-mønsteret). Du bruker semaforer1 for å beskytte kritiske områder av koden din.

Sammenlignet med å implementere en slik lenket liste i minnet som deles mellom tråder, hva er den store utfordringen når man ønsker å koble elementene sammen, og hvordan kan man løse det?

En stor utfordring når man kobler elementene sammen i en lenket liste som deles mellom prosesser er at prosessene har separate minneadresser og kan ikke manipulere pekere direkte. En løsning er å bruke relative adresser eller offset i stedet for absolutte minneadresser. Ved å beregne avstanden eller offseten til hvert element i forhold til starten av den delte minneblokken, kan man etablere koblingene mellom elementene. Semaforer kan brukes til å sikre synkronisering mellom produsent- og forbrukerprosessene i den delte listen.

Hva er fordelene og ulempene ved å implementere en producer-consumer-relasjon mellom prosesser med en slik delt minnekonstruksjon i stedet for pipes?

Fordeler:

Raskere kommunikasjon: Delt minne tillater raskere IPC enn pipes, fordi det eliminerer overhead i forbindelse med switching og kopiering av data.

Mer effektiv bruk av ressurser i systemet: delt minne bruker færre ressurser av et system enn pipes, siden det krever mindre minne og kortere tid for CPU å overføre data.

Delt tilgang til data: tillater flere prosesser å aksessere samme minne samtidig, som legger til rette for mer effektiv kommunikasjon mellom prosesser.

Ulempor:

Problemer med synkronisering: siden flere prosesser kan aksessere samme minne samtidig, så er det en risiko for å få problemer med synkronisering, som feks. Race conditions, deadlocks og inkonsistens i tilgang på data.

Mangel på sikkerhet: Delt minne har ikke alle de sikkerhetsmessige egenskapene som pipes tilbyr, som kryptering, tilgangskontroll og datavalidering. Dette kan gjøre delt minne mindre egnet for å håndtere sensitivt data.

Mer kompleks implementasjon: delt minne krever mer kompleks implementasjon enn pipes, siden det involverer håndtering av segmenter med delt minne, allokering og deadllokering av minne, og sikre overholdelse av synkronisering mellom prosesser.

Hva må du gjøre for å utvide forslaget ovenfor til å gjøre det like fleksibelt og kraftig som mailbox?

For å gjøre forslagene like fleksible og kraftig som mailboxes, må man legge til funksjonalitet som lar prosessene sende meldinger til hverandre og motta meldinger fra hverandre på en asynkron måte. Et forslag er å legge til et felles "meldingsområde" i den delte minneblokken som inneholder meldinger som er sendt mellom prosessene.

Man kan også behandle meldinger forskjellig, ved å prioritere de. De med høy prioritet blir handlet før meldinger med lav. For å gjøre dette kan man bruke flagg og semaforer, i tillegg til signaler for å varsle prosessene om nye meldinger.

3.6 Lagdelingsmodell/ TCP/IP

Oppgave 1

Lagdelingsmodellen for nettverkskommunikasjon antar at lag N gir tjenester til laget over N+1. For å få til dette bruker laget tjenestene til lag N-1 og en protokoll for å kommunisere med tilsvarende lag på en annen node.

Protokollinformasjon blir utvekslet ved å legge til headere (og av og til trailer) til nyttelasten fra lag N+1.

Se for deg en applikasjon på et endesystem i et TCP/IP-nettverk som kommuniserer med en fjerntliggende applikasjon gjennom flere mellomliggende noder.

På hvert lag av nettverkstakken kommuniserer en instans av lag N med en tilsvarende, fjerntliggende instans av lag N. På hvilken node i nettverket finner vi den fjerntliggende instansen for hvert lag? Sagt på en annen måte: Hvor prosesseres protokollheadere?

Lag 1-3 blir prosessert på hver node ettersom den informasjonen trengs for å sende videre. Lag 4 og 5 er bare viktig for endesystemene/endepunktene og vil derfor kun bli prosessert i de nodene.

Ved proxy og mellomsystemer kan/må også Lag 4 og 5 brukes, men dette undervises ikke om i dette kurset.

TCP/IP-modellen har 2 lag mindre enn OSI-modellen (eller 3 hvis man også anser lagene 1 og 2 som sammenslått). Hvilke argumenter taler for en slik lagreduksjon? Gi 2 argumenter og forklar dem nærmere.

Et argument som taler for en lagreduksjon som TCP/IP har gjort fra OSI er at det ikke er nødvendig å standardisere protokoller for lag 5 og 6.

Det er kun lag 1-4 som er essensielle for kommunikasjon på tvers av nettverk. Lag 5 og 6 er ikke nødvendig for all nettverkskommunikasjon, men kan være viktig for noen typer applikasjoner og protokoller og det er også mange applikasjoner som ikke tar i bruk disse lagene i det hele tatt. En applikasjon må selv velge hvordan sesjonen skal se ut og hva slags presentasjonslag de skal velge, og så lager man applikasjonen på toppen av dette. Det er ikke noe krisje for internettet dersom ulike applikasjoner har ulike presentasjons- og sesjonsideer. Derfor er det ikke viktig at alt på lag 5 og 6 er standardisert for alle maskiner i hele verden.

Et annet argument er at skillet mellom transport- og nettverkslaget ikke er tydelig nok. I tillegg har vi klart definert lag 3 og 4 (nettverk (IP) og transport (UDP og TCP), men det er ikke like lett å se skillet mellom disse to lagene, og de to lagene utveksler mer informasjon enn lag i OSI modellen ville gjort.

5-6 er ikke essensielle for internettkommunikasjon
Greit at applikasjonen velger selv lag.

Når fysiske brev blir transportert mellom land, blir de samlet sammen i sekker og bokser for å redusere ressurskravet som må til for å transportere dem. Kan du se for deg en tilsvarende tilnærming for IP pakker?

Pakkeaggregering er en teknikk for å gruppere flere mindre pakker inn i en større pakke. I stedet for å sende hver enkelt pakke separat, blir mindre pakker samlet til en større.

Multiplexing, altså en måte å samle flere ip pakker i en større pakke for å mer effektiv bruk av ressurser. Det er mer effektivt fordi du sparer plassen du bruker på headers ved å kun bruke en istedenfor å ha flere.

Oppgave 2

Bortsett fra lagdelingsmodellen, finnes også andre nettverksarkitekturmodeller, for eksempel den komponentbaserte modellen og den rekursive modellen. Disse

modellene har mottatt betydelig forskningsinteresse, men har ikke blitt satt i praksis utover prototyper og småskalainstallasjoner. Likevel tilbyr de noen egne fordeler.

Forklar kort de fremtredende forskjellene mellom de ulike nettverksarkitekturmodellene.

Lagdelingsmodellen: Deler funksjoner opp i en hierarkisk rekkefølge av lag, hvor hvert lag har spesifikke oppgaver og ansvar. Dette gir tydelig interface, tydelig fordeling av ansvar, og hvert lag kan utvikles separat, og endres uavhengig av hverandre.

Komponentbasert modell: Deler funksjonalitet inn i separate komponenter, hvor hver komponent har ansvar for en bestemt funksjon. Dette gjør at en kan unngå dupliserte funksjoner og muligens gi perfekt nettverksadferd for hver enkelt applikasjon. Men denne modellen trenger en fleksibel interface.

Rekursiv modell: Fokuserer på at nettverket består av flere undergrupper av nettverk, og hver av undergruppe har nivåer og hvert nivå gir enn viss funksjonalitet. Dette gjør at problemer kan bli løst lokalt, og gir skalerbarhet. Det kan være vanskelig å administrere og overvåke pga. Kompleks struktur.

3.6.1 TCP/IP Egenskaper

Hvilke egenskaper ved TCP/IP-arkitekturen har vært avgjørende for dens suksess?

Åpenhet. Utviklet med åpne standarder, og kunne bli benytte av hvem som helst. Ført til utbredelse og støtte fra mange.

Skalerbarhet.

Distribuert natur: dataene og ressursene er spredt over hele nettverket og ikke et sentralisert sted. Ført til høy grad av tilgjengelighet og robusthet. Motstandsdyktig mot feil og angrep.

Protokollens fleksibilitet. Designet for å støtte en rekke forskjellige applikasjoner og tjenester

Evne til å takle forskjellige nettverksteknologier. Designet for at fra Ethernet til Wifi og mobildatanettverk. Gjør at enheter på ulik teknologi kan kommunisere sammen.

Recursive Internetwork Architecture (RINA) har blitt foreslått som et mer effektivt alternativ til den eksisterende TCP/IP-arkitekturen. Hvilke tenkelige utfordringer kan oppstå ved arbeidet med å erstatte TCP/IP med RINA i bred skala?

Høye kostnader, tidsbruk og ressurser som kreves for å implementere RINA.

Opplæring av IT folk.

Motstand fra industrien som er vant til en standard. Risikabelt og unødvendig
Ulike protokoller kan føre til kompatibilitetsproblemer mellom de to arkitekturene.

Må utvikle nye applikasjoner og tjenester som er optimalisert for RINA.

Oppgave 3

I en ideell verden gir internett topp ytelse for ethvert brukstilfelle: ubegrenset båndbredde, null latens, ubrytelig sikkerhet og mer. I den virkelige verden må vi klare oss med det vi har. Ethvert brukstilfelle stiller imidlertid ikke nødvendigvis høye krav til alle slags ytelse. For de tre brukstilfellene som er oppført nedenfor, diskuter kort hvilke kriterier som virker mest relevante, og hvilke kriterier som virker mindre relevante.

Cloud gaming

Kriterier som er viktig for cloud gaming er lav latens. Dette hjelper med å sikre at spillet er stabilt og at bruker ikke må vente på respons. Noe annet som er relevant er også høy båndbredde, fordi det sørger for at videostrømmingen av spillet er uten forsinkelser og av høy kvalitet (Båndbredde kun relevant for Cloud gaming ikke vanlig gaming).

Noe som er et mindre relevant kriterie er tilgjengelighet. En ønsker helst at cloud gaming skal være tilgjengelig hele tiden, men hvis en antar 90% oppetid så er det ikke så viktig, for eksempel når en tjener er nede for vedlikehold.

Instant messaging

Når det kommer til instant messaging er det viktigst med høy pålitelighet. Dersom det forekommer en feil i meldingene som er sendt, vil det være ubruklig å sende meldinger over nett. Sikkerhet er også et viktig aspekt, da det er viktig å ivareta det slik at man trygt kan sende meldinger over nettet uten at uvedkommende kan lese meldingen som er sendt.

Det er også relevant at meldinger kommer fort frem, men det er ikke så farlig hvis det er få sekunder forsinkelse.

Autonome kjøretøy

Lav latens er svært viktig. Dette er på grunn av at autonome kjøretøy er sanntidssystemer, hvor forsinkelser vil føre til at bilen kanskje misforstår trafikkbilde eller så kan forsinkelse føre til at bilen ikke gjøre beregninger i tide.

Sikkerhet er særlig viktig for autonome kjøretøy. Særlig tilgjengelighet og integritet. Dersom kjøretøyets data mangler eller er feil, kan også kjøringen bli påvirket av dette og medføre farer for passasjerere.

Relevant: lav latens, høy pålitelighet, sikkerhet

Mindre relevant: minimal forsinkelse eller buffering.

3.7 Protokoller

Oppgave 1

Protokollene på datalinklaget har ansvar for å sende pakker til direkte naboer. Hva betyr det at to noder er direkte naboer til hverandre i et kabelt Ethernet-nettverk med svitsjer?

I et kabelt Ethernet-nettverk med svitsjer betyr det at to noder er direkte naboer til hverandre hvis de er koblet til samme svitsj eller til to svitsjer som er koblet sammen med en direkte kabel.

Hvis node A er en direkte nabo til node B, og node C er også en direkte nabo til node B, betyr det alltid at node A og node C er direkte naboer til hverandre? Forklar!

Nei, det betyr ikke nødvendigvis at node A og node C er direkte naboer til hverandre selv om de begge er direkte naboer til node B.

For eksempel kan node A være koblet til node B gjennom en trådløs forbindelse eller en annen ikke-kablet forbindelse, mens node C kan være koblet til node B gjennom en kablet forbindelse. I dette tilfellet vil ikke node A og node C være direkte koblet sammen, men de kan likevel kommunisere med hverandre gjennom node B ved å sende datapakker til node B som vil videresende dem til riktig mottaker.

På samme måte kan det være tilfeller der både node A og node C er koblet til node B via kablede forbindelser, men ikke direkte til hverandre. Kanskje de er koblet til forskjellige porter på node B, som igjen er koblet til en felles svitsj. I dette tilfellet kan node A og node C kommunisere med hverandre ved å sende datapakker gjennom node B, men de er ikke direkte koblet sammen.

Så selv om node A og node C begge er direkte naboer til node B, betyr ikke det nødvendigvis at de er direkte naboer til hverandre.

Hvis to noder er direkte naboer til hverandre, betyr det at de er fysisk nære hverandre?

Nei, at to noder er direkte naboer i et nettverk betyr ikke nødvendigvis at de er fysisk nærmere hverandre. De kan for eksempel være koblet til hverandre gjennom en kjempelang ethernet kabel. En satellitt som kommuniserer med en enhet på bakken vil også være direkte nabo til denne enheten, men ikke fysisk nærmere.

Oppgave 2

IP-addresser gjør det mulig for TCP/IP- nettverkslaget å finne en sti mellom maskiner. Hvordan brukes nettmasker for å gruppere adresser?

Nettmasker brukes til å definere størrelsen og strukturen til subnettene i et IP-nettverk. Ved å anvende en nettmaske på en IP-adresse, kan man avgjøre hvilken del av adressen som tilhører nettverksdelen og hvilken del som tilhører vrtsdelen.

En nettmaske består av en sekvens av enere og nuller, der lengden på sekvensen angir størrelsen på nettverksdelen. For eksempel, hvis nettmasken er 255.255.255.0, betyr det at de første 24 bitene (3 oktetter) tilhører nettverksdelen og de siste 8 bitene (1 oktet) tilhører vrtsdelen.

Ved å bruke forskjellige nettmasker kan man segmentere IP-adressene i ulike subnett. Hvert subnett kan inneholde en gruppe av IP-adresser som er logisk relaterte og som kan kommunisere direkte med hverandre uten å måtte gå gjennom en ruter. Dette muliggjør mer effektiv administrasjon av nettverket ved å redusere belastningen på rutere og forbedre ytelsen.

Gruppering av IP-adresser ved hjelp av nettmasker er også nyttig for sikkerhet og nettverksisolasjon. Ved å opprette separate subnett kan man begrense tilgangen mellom forskjellige deler av nettverket, slik at kun autoriserte enheter kan kommunisere med hverandre.

Kort sagt, nettmasker brukes til å dele opp IP-adresser i subnett, slik at nettverksressurser kan organiseres og administreres mer effektivt, samtidig som man opprettholder sikkerhet og kontroll over nettverket.

Hvorfor er det viktig for ruting i internettet at IP-adresser er gruppert?

Det er viktig for ruting i internettet at IP-adresser er gruppert fordi det gjør det mulig å identifisere og rute trafikk mellom forskjellige nettverk og subnett. Nettverksadresser brukes til å identifisere et bestemt nettverk, og subnettadresser brukes til å identifisere et bestemt subnett innenfor dette nettverket.

Ved å gruppere IP-adresser i nettverk og subnett, kan ruterne enklere håndtere ruting av trafikken og unngå unødvendig trafikk på nettverket. Dette bidrar til å sikre effektiv og pålitelig kommunikasjon på internettet.

Standard antall IPv6 adresser i et subnett er 264, noe som er et stort tall. Hva tror du om ideen å gi hver prosess på en maskin sin egen IPv6 adresse?

Tror ikke det er så lurt pga. mye administrativt arbeid. Gjør det vanskeligere å kommunisere med datamaskiner i en helhet. Øker kompleksiteten for mye. Sikkerhetsårsaker, vanskelig å håndtere og kontrollere kommunikasjon mellom så mange adresser.

Oppgave 3

På internett brukes i dag nettmasker for å dele opp en IPv4-addresse i den delen som adresserer nettverket (eller subnettverket) og den delen som adresserer den mottakende datamaskinen i dette nettverket.

Nettmasker fungerer på en slik måte at nettverk (og subnettverk) alltid tildeles et antall adresser som er en potens av to. Forklar grunnen til dette.

Nettmasker brukes for å dele opp en IPv4-adresse i nettverksdelen og vertsdelen.

Nettverksdelen identifiserer selve nettverket, mens vertsdelen identifiserer en spesifikk datamaskin (eller annen enhet) i dette nettverket. Nettmasken består av en rekke bits som er satt til 1 i nettverksdelen og 0 i vertsdelen.

Grunnen til at nettverk og subnett alltid tildeles et antall adresser som er en potens av to, har å gjøre med måten IP-adresser og nettmasker fungerer på. IP-adresser er representert som en serie med binære tall som består av 32 bits i IPv4-formatet. En nettmakse består også av en serie med bits som er satt til 1 eller 0.

Når nettmasken er satt, kan antall mulige adresser i nettverket beregnes ved å telle antall bits som er satt til 0 i vertsdelen. Dette gir et binært tall som kan konverteres til et desimaltall. For eksempel, hvis det er 8 bits som er satt til 0 i vertsdelen, gir dette $2^8=256$ mulige adresser.

Antall mulige adresser i et nettverk eller subnett er derfor begrenset av antall bits som er tilgjengelige i vertsdelen, og dette må være en potens av to for å sikre at alle mulige adresser kan brukes. Hvis antallet adresser i et nettverk ikke er en potens av to, vil det være noen adresser som ikke kan brukes og som blir "bortkastet". Dette vil føre til ineffektiv bruk av adressressursene, noe som ikke er ønskelig i et nettverk.

En ruter mottar en pakke med fra-adresse '9.228.14.11' og til-adresse '80.80.80.80'; den må finne ut hvilken node den skal sende pakken til. Ruter har tre nettverkskort, henholdsvis med nettverksadressene '80.0.0.0', '9.228.0.0' og '143.166.0.0'. Er dette nok informasjon for å finne ut hvor ruten skal sende pakken nå? I tilfelle ikke: Hva mangler?

Nei, dette er ikke nok informasjon for ruter å finne ut hvilken node den skal sende pakken til. For å kunne gjøre dette, trenger ruter informasjon om nettverksmasker som brukes i nettverkene.

Nettverksmasken brukes til å skille nettverksdelen og vertsdelen i en IP-adresse. Hver av de tre nettverksadressene som er oppgitt, vil ha en tilsvarende nettverksmaske som vil angi hvilken del av IP-adressen som tilhører nettverksdelen og vertsdelen.

Når ruter mottar en pakke, vil den bruke nettverksmasken til å sammenligne nettverksadressen i pakken med nettverksadressene som er konfigurert på sine grensesnitt. Basert på denne sammenligningen vil ruter avgjøre hvilket grensesnitt pakken skal sendes videre til.

Så for å avgjøre hvor ruten skal sende pakken nå, trenger vi å vite nettverksmaskene for hvert av nettverkskortene. Uten denne informasjonen, vil ruter ikke kunne avgjøre hvilket grensesnitt pakken skal videresendes til.

Når en datamaskin er vert for flere virtuelle maskiner, kan disse konfigureres på en slik måte at alle de virtuelle maskinene tilhører det samme lokale nettverket, men hver virtuell maskin har sin egen IP adresse. Skaper dette noen komplikasjoner for å adressere prosesser som kjører på en bestemt virtuell maskin? Forklar svaret ditt.

Ja, dette kan skape noen komplikasjoner når man skal adressere prosesser som kjører på en bestemt virtuell maskin.

Hver virtuelle maskin har en egen IP-adresse, men de kan være konfigurerert til å være på det samme lokale nettverket. Dette betyr at alle virtuelle maskiner har forskjellige IP-adresser, men de kan kommunisere med hverandre på det samme nettverket.

Når man skal adressere prosesser som kjører på en bestemt virtuell maskin, må man vite IP-adressen til den virtuelle maskinen og portnummeret til prosessen som man ønsker å adressere. Hvis det er flere virtuelle maskiner på det samme lokale nettverket med forskjellige IP-adresser, kan dette gjøre det vanskelig å vite hvilken virtuell maskin man skal adressere.

En mulig løsning på dette problemet er å bruke en form for tjenestediscovery, hvor man kan registrere og søke etter tjenester som kjører på virtuelle maskiner. Dette kan gjøre det enklere å finne og adressere prosesser som kjører på en bestemt virtuell maskin, uavhengig av hvilken IP-adresse den har.

3.8 DNS, Adresseblokk

Spørsmål 1

Hovedmotivasjonen for utviklingen (og den fortsatt pågående) utrullingens av internettprotokoll versjon 6 (IPv6), var uttømmingen av IPv4-adresserommet, som var så å si fullført sent i 2019. Innen da var poolen tømt. Et mottiltak, satt i verk flere år tidligere, var privat adressetildeling. Private adresser, som i tilfellet IPv4 tilhører adresseblokkene 10.0.0.0/8, 172.16.0.0/12 og 192.168.0.0/16, er rutbare kun innen lokale nettverk – ikke på det offentlige internettet. På denne måten kan private adresser også gjenbrukes innenfor separate lokale nettverk.

IPv6 råder bot på problemet med sitt enorme adresserom. Likevel kan nettverksadministratører ønske å ta i bruk private adresser innenfor IPv6-nettverk også. For denne hensikten gir IPv6 unike lokale adresser (ULA), som tar i bruk prefikset fc00::/7. Adresseblokken fc00::/7 er partisjonert til to halvdeler med lik størrelse, altså fc00::/8 og fd00::/8, blant hvilke kun den andre foreløpig er definert for lokalt tilordnede adresser.

Hvor mange private nettverk kan adresseblokken fd00::/8 støtte? Du behøver ikke å stave tallet fullt ut. (Merk at adresseblokken selv har et prefiks.)

Adresseblokken fd00::/8 kan støtte 2^{120} adresser, noen av det er nettverksadresser og noen er broadcastadresser som ikke kan brukes, men det er 2^{120} .

Notasjonen "/8" betyr at subnettmasken har 1-bits i de åtte første posisjonene og 0-bits i de resterende 120 posisjonene. Med 16 bits for å identifisere undernettverk (se tabellen under), kan hvert nett støtte 2^{16} undernettverk, og hvert undernettverk kan støtte 2^{64} verter.

For å svaret helt konkret på spørsmålet så er det 2^{56} mulige private nettverk, pga. Det er mulig å adressere nettverk for alle Global ID og Subnett ID.

8 bits	40 bits	16 bits	64 bits
FD	Global ID	Subnet ID	Interface ID

Tabell hentet fra (<https://geek-university.com/ipv6-unique-local-address/>) den 14.04.2023

Etter nettverkprefikset har en lokalt tilordnet adresse 16 bits som identifiserer undernettverket. Hvor mange undernettverk kan ett enkelt nettverk støtte? Hvor mange verter kan ett enkelt undernettverk støtte?

Konvensjonen er i dag 2^{16} undernettverk, men IPv6 tillater egentlig flere. Etter hvert som IPv6 utvikler seg vil dette gi mer frihet til å velge og oppdele bitsene.

En skeptisk venn av deg tror ikke på IPv6. «Det er oppskrytt», sier han. «Akkurat som alle i 1981 trodde at IPv4 ville gi oss nok adresser, tror alle i dag at IPv6 vil gjøre det samme. Vent fire tiår til, og du vil se at fortellingen gjentar seg. Se for deg når vi har nanoroboter så små som virus – og like tallrike – som alle krever sin egen unike adresse!» Som IPv6-evangelist, hvordan vil du overbevise ham om noe annet?

IPv6 tilbyr 2^{128} adresser. Dette vil forsikre oss om at det er nok IP-er for fremtiden i henhold til veksten av tilkoblede tjenester/utstyr. Den eneste måten IPv6 vil bli utdatert er hvis det skjer en umåtelig økning i devices, som er høyest usannsynlig.

Spørsmål 2

3.8.1 Domenenavn

Domenenavnsystemets (DNS) hovedhensikt er å oversette tekstuelle domenenavn til numeriske IP-adresser. Det første er bedre egnet for mennesker, og det andre er bedre egnet for datamaskiner. Dets hierarkiske og desentraliserte arkitektur tillater organisasjoner på ulike nivåer å håndtere sine spesifikke delmengder av domenenavnrommet.

For eksempel, det fiktive selskapet Nameria Inc., som eier nameria.com, håndterer underdomenene under dette andrenivådomenet. Som nettverkadministrator for Nameria Inc., har du oppgaven å konfigurere sonefilen, som inneholder ressursoppføringene.

De primære og sekundære navnetjenerne er tilgjengelige fra ns1.namaria.com og ns2.namaria.com; deres IP-adresser er henholdsvis 120.80.80.100 og 120.80.80.101.

Vevtjeneren er tilgjengelig fra www.namaria.com eller simpelthen namaria.com. Den deler IP-adresse med den primære navnetjeneren. E-posttjeneren er tilgjengelig fra mail.namaria.com. Dens IP-adresse er 120.80.80.105.

Skriv en sonefil som tilfredsstiller kravene ovenfor. Du kan anta standard direktiver og Start of Authority (SOA)-verdier.

```
$ORIGIN nameria.com
$TTL    86400
@       IN      SOA     ns1.nameria.com. admin.nameria.com. (
                          1000000000 ; serial number
                          10800      ; refresh period
                          3600       ; retry period
                          604800    ; expire time
                          3600)     ; minimum ttl

                      NS      ns1.nameria.com
                      NS      ns2.nameria.com
                      MX    10   mail.nameria.com

ns1      IN      A      120.80.80.100
ns2      IN      A      120.80.80.101
www      CNAME   ns1
nameria.com. CNAME   ns1
mail     IN      A      120.80.80.105
```

Denne starter med å definere at det er sonefil for namaria.com
Neste linje er livstid på 86400 sekunder.

Det neste som kommer er diverse verdier i recorden:

- Serienummer som øker hver gang det er forandring
- Refresh for da andre DNS servere skal hente informasjon igjen
- Retry tid, hvor lenge man skal vente etter et feil på et oppslag før man spør engang til
- Expire, maksimal tid som man har lov til å beholde datablokken i DNS server
- Minimum, dvs. Hvor lang tid man kan cache det DNS utgjør et typisk bredt og grunt tre: noder kan ha et svært stort antall barn, men de fleste bladnoder er plassert ikke så langt fra rotten (som f.eks. .com). Hvorfor fremviser DNS denne strukturen?

NS (name server)

MX (Egen record med prioritet 10) som gir navn til mail server

De to neste linjene definerer adressene.

Deretter defineres alias for ns1, slik at man ikke må skrive ut det fullnavnet med alle prefiks hver gang.

Spørsmål 3

3.8.2 DNS eksmplar og oppgaver

En av dine kolleger er nysgjerrig på DNS. «Hvordan fungerer det?», spør han. «Når noen besøker nettstedet, hvordan blir www.nameria.com, skrevet inn i nettleserens adressefelt, oversatt til vevtjenerens IP-addresse og rapportert tilbake til den besøkende?». Du forklarer at DNS fungerer liksom en telefonkatalog for internett.

DNS har et hierarkisk, i motsetning til et flatt, navnerom. Forklar forskjellen mellom disse to måtene å organisere et navnerom på.

Et hierarkisk navnerom er organisert som et tre, med rotnode og grener. Dette danner et hierarki av domener med underdomener. En node kan ha flere barn og hvert barn kan ha egne barn. Dette tillater effektiv administrasjon og delegering av domener. Feks. Store bedrifter eller lignende siden det er en struktur for domenet.

Et flatt navnerom er organisert som en liste over unike domener, uten implisitte forhold mellom domene. Denne tilnærmingen er enklere å gjennomføre og administrere for små navnerom. Det kan fort bli veldig stort og vanskelig å vedlikeholde når antall domener vokste.

DNS utgjør et typisk bredt og grunt tre: noder kan ha et svært stort antall barn, men de fleste bladnoder er plassert ikke så langt fra roten (som f.eks. .com). Hvorfor fremviser DNS denne strukturen?

DNS ønsker at et grunt og bredt tre for det er enklere å huske på. Gjør routing kjappere. Et dypere tre vil føre til at det tar lenger tid å traversere gjennom. Man vil at et grunt tre slik at det er mulig for oss å huske adresser, det er det absolutt viktigste.

DNS-spørninger kan håndteres på én av to måter: rekursivt eller iterativt. Forklar forskjellen mellom dem, og fremhev deres fordeler og ulemper.

Når en klient skal aksessere et domene, sender den en forespørsel til DNS serveren som håndterer IP adressen for serveren. I rekursiv form vil DNS sende forespørselen videre dersom IP adressen ikke finnes på den gjeldende DNS serveren. DNS serveren sender da forespørselen til en annen DNS server og dette gjøres rekursivt frem til IP-en er funnet.

Ved en iterativ form vil klienten sende en forespørsel til DNS serveren, og serveren svarer klienten med et delsvar som gir instruerer klienten til å sende en annen forespørsel til en annen DNS server. Dette gjøres frem til hele IP-adressen er løst.

Den iterative formen er mer effektiv for DNS serveren og kan potensielt redusere flaskehals. Den er mindre effektiv for klienten som må gjøre flere DNS forespørsler. Klienten må også skjønne logikken for domene navn løsning og kan ikke etterlate jobben til DNS serveren. Fordel med iterativ form er at den er sikrere mot Ddos angrep, på grunn av at forespørselene er splittet opp mellom flere DNS servere.

Den rekursive formen er mer effektiv for brukeren, men ikke like effektiv for DNS servere, som må sende forespørselen videre og dette kan potensielt lede til en flaskehals.

Kompleksiteten reduseres også, da klienten ikke trenger å forstå logikken bak domene navn løsning. Den rekursive måten er også mer sårbar mot Ddos angrep, da klienten bare "snakker" til en DNS server, og derav fordeles ikke belastningen på flere DNS servere. Ved å bruke rekursiv måte kan man også cache dataen.

3.9 TCP og UDP

Spørsmål 1

De to hovedprotollene på transportlaget, UDP og TCP, tilbyr henholdsvis forbindelsesfrie og forbindelsesorienterte tjenester. Den ene er ikke i seg selv overlegen den andre; hvilken som passer avhenger av det enkelte brukstilfelle. Som sjefsteknolog for DatagramCam, en oppstartsbedrift som utvikler en videokonferanseløsning, må du på et tidlig stadium avgjøre hvilken protokoll å anvende for å overføre lyd og video i sanntid mellom to eller flere klienter.

3.9.1 Sammenlikn UDP og TCP

Nevn noen fordeler og ulemper ved hver protokoll.

TCP	UDP
Forbindelsesorientert, som betyr at det må opprettes en forbindelse mellom sender/mottaker	Forbindelsesfri, trenger ikke å etablere forbindelse før den sender/mottar
Fordel: garanterer at pakker som sendes kommer fra, og at det kommer fram i riktig rekkefølge. Dette gjør den mer pålitelig. TCP tilpasser overføringshastigheten ved å ta hensyn til nettverksforbindelsen og mottakers evne til å behandle data for å forhindre overlastning/tap av data	Fordel: Raskere, da den ikke trenger å opprette en forbindelse og ikke må garantere at pakker kommer fram eller i riktig rekkefølge. Det vil heller ikke ha noe å si for andre pakker, dersom en pakke ikke kommer fram. Bedre overføringstid og ytelse. Krever mindre overhead som fører til mindre forbruk av båndbredde. Støtter multicasting
Ulemper: Støtter ikke multicasting. Hvis en pakke ikke kommer fram, vil dette gi store problemer og forbindelsen vil avsluttes etter hvert.	Ulemper: Har ikke metningskontroll. Har ikke flytkontroll, som kan føre at det sendes for store mengder data som mottaker ikke får behandlet

(Multicasting er at protokollen kan sende en pakke til en gruppe datamaskiner på en gang, også er det nettverket som fordeler pakker til de som skal motta.)

Hvilken protokoll ville du valgt til formålet? Gi din begrunnelse.

I dette tilfelle ville jeg valgt UDP. Dette er på grunn av behovet for lyd og video i sanntid og muligens skal broadcaste til flere enheter. TCP er ikke svært egnet for sanntids konferanse.

Det vil i dette tilfelle være bedre at noen piksler går tapt, eller midlertidig dårligere lyd/bildekvalitet, fremfor at brukeren ikke klarer å ha en tilnærmet samtale i sanntid. Dersom

det i tillegg skulle være en chatte-funksjon med test, ville det kanskje vært mer fordelaktig å ta i bruk TCP.

Ville det vært mulig å implementere løsningen ved å bruke den andre protokollen? Hvilke mulige problemer kunne oppstå?

Ja det ville vært mulig å implementere løsningen med TCP, men dette kan føre til høyere overhead og dermed høyere latens. Ved bruke av TCP kan dette også føre til merkbare forsinkelser for både video og lyd, noe som ikke er optimalt ved sanntids videokonferanser. Når det er snakk om videokonferanser er det viktigere at video og lyd kommer frem så fort som mulig, enn at dataen er 100% korrekt.

Spørsmål 2

3.9.2 Linjesvitsjing og pakkesvitsjing

Selv om nesten alle moderne datamaskinnettverk, inkludert internett, avhenger av pakkesvitsjing, så har linjesvitsjing fortsatt teknisk interesse. Forskjellen mellom de to teknikkene er sammenliknbar med forskjellen mellom gammeldagse post- og telefontjenester: Der to brev fra samme avsender til samme mottaker kan ta forskjellige stier – og dele segmenter av stien med andre brev underveis – så bruker en telefonsamtale en dedikert og konstant forbindelse mellom de to kommuniserende partene.

Hvordan kan linjesvitsjing oppnås på moderne, pakkesvitsjede nettverk?

Hvordan yter denne teknikken sammenliknet med vanlig pakkesvitsjing?

Linjesvitsjing på moderne, pakkesvitsjede nettverk kan oppnås ved bruk av VCS (Virtual Circuit Switching). Ved å etablere en virtuell sti mellom de to endepunktene før data blir sendt, vil dette fungere som en midlertidig dedikert rute. Dataen blir sendt gjennom denne som om det var en fysisk forbindelse. Det er tre egenskaper som forbedres: garantert båndbredde, lav forsinkelse og lav jitter (variasjon i tid mellom ankomst til datapakker). En ulempe med dette er at hvis en velger garantert båndbredde på alt, vil det være den samme tapsrate (den samme reduserte ytelsen som ved circuit switching nettverk).

Hvilke forskjeller ser du mellom linjesvitsjing og virtuell linjesvitsjing? Hvilke fordeler har den ene over den andre?

Ved vanlig linjesvitsjing må en forbindelse fysisk eksistere gjennom hele datautvekslingen, forbindelsen må settes opp før utvekslingen begynner og dette tar tid. Ved virtuell linjesvitsjing reserveres båndbredde til en datapakke som sendes nettverket når den trengs, men det er ingen fysisk kobling. Meldingen sendes som i pakkesvitsjing, men addreseringen er knyttet til en virtuell krets i stedet for en destinasjon. Flere meldinger kan sendes samme vei, men alle har reservert en del av båndbredden på forhånd.

Fordel med linjesvitsjing er at den er pålitelig og sikrer at pakkene kommer frem, siden ingen av pakkene må konkurrere om båndbredden. Ulempen er at man må reservere en hel linje, uten at man faktisk skal bruke hele.

Fordel med virtuell linjesvitsjing har man flere valg, og man må ikke reservere alt av en kanal/frekvens/kabel. Man kan reservere å kunne sende i halvparten av tiden, slik at andre

kan bruke resten. Med andre ord kan man fleksibelt reservere den båndbredden man trenger. Ulempen er at båndbredden ikke er ubegrenset.

Hovedforskjellen mellom linjesvitsjing og virtuell linjesvitsjing ligger i måten de etablerer forbindelser for kommunikasjon på. Linjesvitsjing krever en dedikert fysisk forbindelse mellom to noder for varigheten av en kommunikasjonsesjon, noe som sikrer en konstant dataoverføringshastighet og forsinkelse, men kan være ressurskrevende da båndbredden forblir reservert selv i perioder uten aktiv kommunikasjon. På den annen side, i virtuell linjesvitsjing, blir en virtuell forbindelse, kjent som en virtuell krets, opprettet mellom nodene. Denne virtuelle kretsen er ressurseffektiv fordi den kan opprettes og avsluttes ettersom behovet oppstår, noe som gjør det mulig å dele og gjenbruke nettverksressurser mer effektivt. Imidlertid kan dette føre til varierende dataoverføringshastigheter og forsinkelser avhengig av nettverkstrafikken. Dermed er valget mellom linjesvitsjing og virtuell linjesvitsjing ofte et kompromiss mellom ytelse og ressurseffektivitet.

TCP, en transportlagsprotokoll, tilbyr også forbindelsesorientert tjeneste. Kan TCP erstatte linjesvitsjing på det fysiske laget? Hvordan – eller hvordan ikke?
Selv om TCP og linjesvitsjing fungerer på lignede måte, hører TCP til på transportlaget (lag 3), mens linjesvitsjing hører til på det fysiske laget (lag 1).

Gitt at man skal beholde lagfordelingen for nettverkskommunikasjon som en er organisert i dag, mener vi derfor at det ikke vil være mulig for TCP å erstatte linjesvitsjing fordi de operer på ulike lag og har ulike ansvarsområder.

Det er slik at TCP på lag 4, leveres på toppen av et forbindelsesløst lag 3 (vanligvis). Hvis en sti mellom nett blir ødelagt, kan TCP likevel opprettholde forbindelsen, og pakkene på lag 3 kan finne en annen sti. Hvis man derimot har dette på lag 1, må man retablere en ny forbindelse fra ende til ende som koster mye mer. TCP kommer aldri til å ha en båndreservering.

Spørsmål 3

3.9.3 IPv6

Internettprotokoll versjon 6 (IPv6) tilbyr, som kjent, et langt større adresserom enn IPv4. På andre måter er den nyere versjonen imidlertid en forenkling sammenliknet med den eldre versjonen.

IPv6-pakkeheaderen har til forskjell fra IPv4-pakkeheaderen intet sjekksumfelt, som brukes for påse pakkenes integritet. Hvis denne funksjonen ikke lenger utføres på nettverkslaget, hvor utføres den da? Hvorfor ble det ikke gjort slik i utgangspunktet, tror du?

Feilkontroll og verifisering av dataintegritet blir både utført på transportlaget og datalinklaget. TCP og UDP inneholder begge sjekkesumfelt i deres headere som kontrollere dataene på transportlaget. På datalinklaget inneholder rammene også en sjekk som brukes for å verifisere data over det fysiske lag.

Da IPv4 ble utviklet var det ikke like mange sjekker i de andre lagene, så derfor var det lurt med en sjekk på nettverkslaget.

Sammenlign IPv6-pakkeheaderen med IPv4-pakkeheaderen. Foruten sjekksumfeltene finnes flere vesentlige forskjeller. Identifiser noen av disse forskjellene og forklar hva du tror årsaken bak er.

IPv6 har adresselengde på 128 bits mens IPv4 har en lengde på 32 bits. I IPv6 er det to nye felt kalt "Traffic Class" og "Flow Label". Traffic Class-feltet erstatter Type of Service-feltet i IPv4 og gir bedre støtte for å håndtere ulike typer trafikk med ulik prioritering. Flow Label-feltet, som ikke finnes i IPv4, er designet for å forbedre ytelsen for sannidsapplikasjoner ved å tillate rutere å identifisere og håndtere pakker som tilhører en spesifikk dataflyt.

I IPv4 håndteres fragmentering både av rutere og avsender underveis. Dette fører til en økt kompleksitet og kan redusere ytelsen. I IPv6 er fragmentering flyttet til avsenderen ved hjelp av en spesiell Fragment Extension Header, og den maksimale pakkestørrelsen har økt slik at færre datapakker må fragmenteres og reduserer flyt. Dette forenkler behandlingen av pakker i rutere og øker nettverksytelsen.

Det finnes flere endringer enn disse, men felles for de er at de er lagret for å adressere noen av begrensingene og utfordringene i IPv4, samt forbedre ytelsen, skalerbarhet og funksjonalitet.

Hvordan vil du bruke IPv6 headered for å sende IPv6 pakker over avsnitt i Internettet som bruker virtuell kretssvitsjing fremfor pakkesvitsjing?

En IPv6 pakke kan bli sendt over en del av internettet som bruker virtuell linjesvitsjing ved å bruke "Flow" labelen i headeren for den pakken. En pakke som blir sendt med virtuell linjesvitsjing må ha en slags Virtual Circuit Identification (VCI) for at en node i nettverket skal kunne vite hvilken vei pakken må sendes til for å komme til destinasjonen sin. Da kan "Flow" labelen brukes for å identifisere akkurat dette. Siden pakken med samme flow må ta samme vei gjennom et nettverk som bruker Virtual Circuit Switching.

3.10 FLYTKONTROLL

Spørsmål 1

Flytkontroll gjør at senderen kan unngå å oversvømme mottakeren. En viktig familie av flytkontrollmekanismer er sliding window protokollene. Sliding window har ytterligere 3 forskjellige variasjoner: Go-Back-N, Selective Repeat og Credit Mechanism.

Hvordan funker Stop-and-wait og hva er fordelene og ulempene dette medfører?

3.10.1 Stop and Wait

I Stop-and-Wait-protokollen deler senderen dataene inn i mindre datapakker og sender en pakke til mottakeren. Deretter venter senderen på en bekreftelse fra mottakeren. Mottakeren mottar pakken, sjekker om den er riktig og sender en bekreftelse tilbake hvis pakken er i orden. Hvis pakken er skadet eller feil, sender mottakeren ingen bekreftelse, og senderen

retransmitterer (på nytt) pakken. Dette fortsetter til senderen mottar bekreftelse for den riktige pakken. Når senderen mottar bekreftelse, går den videre til å sende neste pakke, og prosessen gjentas for hver pakke som skal sendes. På denne måten sikrer Stop-and-Wait-protokollen pålitelig overføring av data ved å vente på bekreftelse før den sender neste pakke og retransmittere ved behov. (Retransmitter betyr å sende pakken på nytt bare)

Fordeler	Ulemper
Enkel implementering: Stop-and-Wait er en enkel protokoll som er enkel å implementere og forstå. Det krever ikke komplekse mekanismer eller algoritmer.	Lav utnyttelse av båndbredde: Stop-and-Wait har lav utnyttelse av båndbredde. Senderen må vente på bekreftelse før den kan sende neste pakke. Dette fører til unødvendig ventetid og reduserer overføringshastigheten.
Pålitelig overføring av data: Protokollen sikrer pålitelig overføring av data. Senderen venter på bekreftelse før den sender neste pakke, og mottakeren sender bekreftelse for hver mottatte pakke. Dette reduserer risikoen for datatap og feil i overføringen.	Høy latens: Protokollen har høy latens, da både senderen og mottakeren må vente på bekreftelse for hver enkelt pakke før de kan fortsette overføringen. Dette kan føre til forsinkelser i overføringen av data.

Vennligst forklar hvordan Go-back-N or Selective Repeat fungerer. Hvilken av disse to sliding window-protokollene ville du brukt for trådløse nettverk da tilfeldig pakketap skjer ofte (men som oftest bare ett pakketap om gangen) og hvorfor?

3.10.2 Go-Back-N

Go-Back-N er en sliding window-protokoll der senderen sender flere datapakker uten å vente på bekreftelse fra mottakeren. Senderen har et vindu som angir antall pakker den kan sende på rad før den venter på bekreftelse. Mottakeren sender en bekreftelse for hver mottatte pakke. Hvis en pakke går tapt eller detekteres en feil, forkaster mottakeren pakken og alle påfølgende pakker i vinduet. Senderen retransmitterer deretter hele vinduet, inkludert de pakkene som allerede er mottatt av mottakeren. Go-Back-N er en enkel protokoll, men kan føre til unødvendig retransmisjon av pakker som allerede er mottatt av mottakeren.

Selective Repeat er også en sliding window-protokoll der senderen sender flere datapakker uten å vente på bekreftelse. Mottakeren sender en bekreftelse for hver mottatte pakke. Hvis en pakke går tapt eller detekteres en feil, sender mottakeren en bekreftelse som ber om retransmisjon av den spesifikke pakken. Senderen trenger ikke å retransmittere hele vinduet, bare de pakkene som ble berørt av feil eller tap. Selective Repeat krever at både sender og mottaker opprettholder en buffer for å lagre de mottatte pakkene inntil de kan leveres i riktig rekkefølge. Denne protokollen reduserer unødvendig retransmisjon, men krever mer kompleksitet og bufferstyring sammenlignet med Go-Back-N.

I scenarioet med tilfeldig pakketap som oftest bare involverer ett pakketap om gangen, ville Selective Repeat være mer egnet for trådløse nettverk. Selective Repeat gir bedre utnyttelse av båndbredde og reduserer unødvendig retransmisjon. Ved å retransmittere bare de tapte pakkene oppnår man mer effektiv overføring av data og bedre utnyttelse av nettverksressurser. Go-Back-N vil resultere i unødvendig retransmisjon av flere pakker, noe som kan føre til mer nettverkstrafikk og lavere effektivitet i trådløse nettverk der pakketap er vanlig.

3.10.3 Kredittmekanisme

Hvordan funker en kredittmekanisme og hvordan håndterer du situasjoner der ingen kreditt gjenstår?

En kredittmekanisme tildeler senderen et visst antall kreditter. Senderen kan sende datapakker så lenge det er kreditter tilgjengelig. Mottakeren sender kreditter tilbake for å indikere ledig kapasitet. Situasjoner uten kreditt kan håndteres ved venting, bufferstyring eller reduksjon av overføringshastighet.

Spørsmål 2

3.10.4 Masse om flytkontroll

Flytkontroll virker mellom avsender og mottaker i en nettverksforbindelse og begrenser hastigheten på dataoverføring mellom de kommuniserende partene.

Hensikten er å forhindre at avsenderen oversvømmer mottakeren med data som den ikke kan motta og behandle i tide – som en flottørventil i et vannsystem. Flytkontroll er implementert på forskjellige lag av protokollstakken: vanligvis på datalinklaget og transportlaget. Begrensningen kan håndheves på enkeltpakkebasis, det vil si stopp-og-vent-policyen, eller på flerpakkebasis, karakteristisk for skyvevindufamilien av policyer.

Hvorfor er stopp-og-vent hensiktsmessig for flytkontroll på datalinklaget – men ikke så mye i transportlaget?

Stopp-og-vent flytkontroll er mer hensiktsmessig på datalinklaget enn på transportlaget fordi datalinklaget vanligvis opererer over pålitelige fysiske koblinger med lav forsinkelse og lite tap. På dette lavere laget er tap og forsinkelser sjeldnere, og tap av en enkelt pakke er mer sannsynlig å være et resultat av feil på fysisk lag. Derfor kan stopp-og-vent-metoden være tilstrekkelig for å sikre pålitelig overføring på datalinklaget.

På transportlaget, der protokollene som TCP opererer, er tap og forsinkelser mer vanlige. Transportlaget er ansvarlig for pålitelig overføring av data over et potensielt upålitelig nettverk. Her kan tap skyldes overbelastede ruter, nettverksfeil eller midlertidige forbindelsesproblemer. På grunn av disse utfordringene, er en mer sofistikert flytkontrollprotokoll som sliding window-protokoller (som Go-Back-N eller Selective Repeat) mer egnet for å håndtere tap og forsinkelser på transportlaget. Disse protokollene tillater mer effektiv utnyttelse av båndbredde og bedre håndtering av pakketap og forsinkelser.

Hvorfor er flytkontroll nødvendig på datalinklaget, når det utføres på transportlaget uansett?

Flytkontroll er nødvendig på datalinklaget for å begrense hastigheten på dataoverføringen mellom avsender og mottaker. Dette sikrer at mottakeren ikke blir overveldet med data den ikke kan håndtere i tide. På transportlaget utføres også flytkontroll, men det fokuserer på flere funksjoner i tillegg til hastighetsbegrensning.

Hvorfor kan det være hensiktsmessig for en applikasjon å implementere flykontroll på applikasjonslaget?

Det kan være hensiktsmessig for en applikasjon å implementere flytkontroll på applikasjonslaget fordi det gir direkte kontroll over applikasjonsytelse, muligheten til å tilpasse flytkontrollen til applikasjonsbehov og evnen til å håndtere varierende nettverksforhold mer fleksibelt. Dette lar applikasjonen optimalisere ytelsen, tilpasse seg spesifikke krav og opprettholde effektiv overføring og ytelse i ulike nettverksmiljøer.

Spørsmål 3

Flytkontroll er motivert som metode som kan forhindre at mottakernoder oversvømmes av en altfor høy pakkerate sendt fra senderen. Men samtidig blander faktisk eksisterende flytkontrollmekanismer denne oppgaven med metoder for å korrigere pakketap.

Hvis et transportlag som utfører flytkontroll kunne stole et hypotetisk nettverkslag som yter pålitelig tjeneste med garantert rekkefølge, flykontrollen kunne vært mye enklere. Hva ville du likevel trenge for å lage en glidende-vindu protokoll?

Flytkontroll er vanligvis implementert for å forhindre at mottakeren blir overveldet av en høy pakkerate fra senderen. Imidlertid integrerer eksisterende flytkontrollmekanismer også metoder for å håndtere pakketap. Hvis et hypotetisk nettverkslag kunne leve pålitelig tjeneste med garantert rekkefølge, ville flytkontrollen på transportlaget være mye enklere. Likevel vil en sliding window-protokoll fortsatt trenge følgende elementer for å fungere:

- en vindustørrelse for å begrense antall sendte pakker
- vindusflytstyring for å justere vindusstørrelsen
- en mekanisme for pakkebekreftelse for å sikre pålitelig overføring
- en retransmisjonsmekanisme for å håndtere pakketap
- bufferstyring for å håndtere forsinkelser og jevn strøm av data.

Selv om et ideelt nettverkslag ville forenkle flytkontrollen, vil en sliding window-protokoll likevel kreve disse mekanismene for å oppnå pålitelig og effektiv dataoverføring.

TCP er uvanlig ved sitt valg om å definere en bytebasert skyvevinduprotokoll (med kredittmekanisme) i stedet for en pakkebasert protokoll. Har dette noen fordeler eller ulemper ved korrektur av pakketap, spesielt for TCP-baserte applikasjoner som SSH?

Fordel:

Bedre utnyttelse av nettverkskapasitet: En bytebasert protokoll tillater mer fleksibel bruk av tilgjengelig båndbredde, da senderen kan fylle vinduet med flere bytes før det mottar bekreftelse. Dette kan føre til en mer effektiv utnyttelse av nettverksressurser.

Mindre overføringsforsinkselse: Ved bruk av bytebasert protokoll kan senderen umiddelbart sende nye bytes etter å ha mottatt bekreftelse for en del av dataene. Dette reduserer overføringsforsinkelsen sammenlignet med å vente på bekreftelse for hver enkelt pakke.

Ulempe:

Samlingsproblemer: Hvis en pakke blir tapt, kan det føre til tap av flere påfølgende bytes i en bytebasert protokoll. Dette kan påvirke ytelsen til applikasjoner som SSH, spesielt i situasjoner med høy pakketap, da det kreves retransmisjon av en større mengde data.

Tap av punktlig informasjon: I en bytebasert protokoll kan tap av en enkelt byte føre til tap av punktlig informasjon. Dette kan være spesielt problematisk for applikasjoner som SSH, der tap av data kan påvirke påliteligheten og integriteten til informasjonen som overføres.

3.10.5 TCP og piggybacking, ACK

TCP bruker piggybacking av ACK-meldinger til meldinger som inneholder data i motsatt retning. Den åpenbare årsaken er at dette reduserer overhead. Men er det et problem når mottakeren på den ene siden er mye bedre til å levere data fra TCP-enheten til applikasjonen enn den andre?

Når mottakeren er betydelig bedre til å levere data fra TCP-enheten til applikasjonen enn senderen, kan det være et problem å bruke piggybacking av ACK-meldinger med data i motsatt retning. Dette skyldes at piggybacking kan føre til en ujevn fordeling av datalevering til applikasjonene, der mottakerens applikasjon kan bli forsiktig på grunn av venting på ACK-meldinger fra senderens applikasjon. Dette kan påvirke ytelsen og responstiden til applikasjonen som er mottakerens side av TCP-tilkoblingen.

3.11 Metningskontroll

SPØRSMÅL 1

Metningskontroll er essensielt for å opprettholde normal operasjon av et datanettverk som ikke har allokering av ressurser.

Forklar kort hva metningskontroll er og hvordan mellomsystemer og endesystemer håndterer det i internettet.

Metningskontroll er en funksjonalitet for å forhindre overbelastning på et nettverk som fører til tap av data.

Mellomsystemer overvåker trafikken og sørger for at en ikke blir overbelastet ved å kommunisere med resten av nettverket. Den kan også utsette videre overføringen for å ikke overbelastes.

Endesystemer håndterer det med å justere på hastigheten ved dataoverføringen og begrense mengden data den sender på en gang. Den regulerer dette så når nettverket har mye trafikk reduserer den mengden data den sender.

Forklar kort hvordan TCP New Renos metningskontroll er designet og hvilke faser den består av.

3.11.1 TCP Renos faser

TCP New Renos er designet for å regulere datatrafikk og utføre metningskontroll i nettverk. Denne varianten er utviklet for å håndtere situasjoner der nettverket blir overbelastet og det oppstår tap av datapakker.

TCP New Renos metningskontroll består av tre faser:

1. Slow Start: Starter med en lav mengde dataoverføring, gjerne med cwnd lik 10, slik at det ikke oppstår situasjoner hvor nettverket blir overbelastet. Den største mengden data, fra hver Connection, som kan være i nettverket samtidig, cwnd, vil så dobles for hver RTT. Denne fordoblingen av cwnd vil foregå fram til den oppnår ssthresh, eller når man oppdager at en pakke har gått tapt.
2. Congestion avoidance: Overvåker tilstanden i nettverket, hvis det skulle oppstå en overbelastning i nettverket vil TCP gradvis redusere dataoverføringen. Hvis dette derimot ikke er tilfellet, også det har oppstått et problem underveis, vil man gå inn i en fast recovery fase
3. Fast recovery: Hvis TCP oppdager tap av datapakker, settes det i gang en rask gjenoppretting. Som oftest skjer dette på grunn av overbelastning i nettverket. Ved fast recovery vil ssthresh og cwnd synke ned til halvparten av den siste cwnd-verdien. Videre vil man gå tilbake til Congestion avoidance fasen.

Pakketap blir brukt som et signal for TCP New Renos metningskontrol. Forklar kort hvordan pakketap blir detektert og hvilke svakheter disse deteksjonene har.

3.11.2 Detektere pakketap

Duplikat-bekreftelse. Når senderen mottar duplike bekreftelser for samme sekvens av data, antar senderen at en pakke har gått tapt. Da vil TCP New Renos raskt sende en ny kopi av den tapte pakken. Ved å håndtere dette før tidsavbruddstimeren, oppnår TCP New Renos raskere gjenoppretting av tapte pakker og reduserer ventetiden.

TCP New Renos bruker også tidsavbruddstimer, som utløses når en sender ikke mottar en bekreftelse innen en viss tidsperiode. Da antar senderen at pakken er tapt og sender dermed en ny kopi av pakken. Duplike bekreftelser kan oppstå uten at det nødvendigvis foreligger et tap av en pakke. Dette kan føre til unødvendig retransmisjon av pakker.

Når en pakke går tapt i nettverket, vil ikke mottakeren kunne sende en bekrefte tilbake til senderen om at pakken er mottatt. Og hvis ikke TCP mottar en bekrefte for en pakke innen en gitt tid, så antar den at pakken er tapt og vil da redusere 'hastigheten på datatrafikken.

Svakheten med disse teknikkene er feks. Hvis tidsavbruddstimeren er satt for kort, kan dette utløse unødvendige overføring av pakker som vil påvirke ytelsen. Samme hvis tidsavbruddstimeren er satt for lang, da vil det bli unødvendig lang ventetid før TCP oppdager at en pakke er tapt.

SPØRSMÅL 2

Trafikkmetting forekommer i datanettverk omrent som det gjør på virkelige veier: Et stort antall pakker – eller kjøretøy – konkurrerer om den begrensede ressursen som er transportkapasitet. Forskjellige midler har blitt anvendt på problemet i hver sine manifestasjoner.

I dagens internett foregår metningskontroll først og fremst på transportlaget, der TCP benytter seg av en algoritme for å unngå metning, som begrenser antall pakker i transitt mellom to endenoder.

To åpenbare mål for virksom og effektiv metningskontroll er å minimere forsinkelse og pakketap. Hvilke andre mål er også verdt å vurdere?

Fairness: Dersom vi har n noder på et nettverk, må hver node få n -del av hele nettverk kapasitet.

Goodput: Vi ønsker at goodput (pakker som ankommer vellykket) skal øke

Stability: I en nettverkssammenheng refererer stabilitet til systemets evne til å opprettholde forutsigbar og konsistent ytelse under ulike forhold. I forhold til metningskontroll betyr det at nettverket skal kunne håndtere varierende trafikkmengder uten å komme i en tilstand av vedvarende metning, noe som kan føre til uakseptabel forsinkelse eller tap av pakker. En stabil metningskontrollmekanisme bør kunne tilpasse seg endringer i nettverkstrafikk og opprettholde god ytelse, selv når belastningen øker.

Responsivnes: Responsivitet refererer til systemets evne til raskt å reagere på endringer i nettverkets tilstand. Dette kan inkludere endringer i trafikkmønster, nettverksfeil, eller endringer i kapasitet. En responsiv metningskontrollmekanisme bør raskt kunne tilpasse seg disse endringene for å minimere effekten på nettverksytelsen. For eksempel, hvis en rute plutselig blir utilgjengelig, bør metningskontrollmekanismen raskt kunne omplassere trafikk til andre ruter for å unngå forsinkelser og pakketap.

Hvordan er metningskontroll forskjellig fra flytkontroll? Er de bare forskjellige midler for samme mål, eller tjener de vidt forskjellige mål?

Metningskontroll har som mål å regulere pakkene nettverket får inn, slik at nettverket ikke blir overbelastet, hvis nettverket blir overbelastet så vil nettverket forkaste pakkene den får inn. Dermed vil metningskontroll hindre at pakketapet vil skje i nettverket mellom to endesystemer.

Flytkontroll har som mål å regulere pakke-mengden som blir sendt fra et ende-system til et annet, slik at mottakeren ikke blir overbelastet, og dermed unngår man at pakker blir tapt hos mottakeren.

Begge tjenestene har samme mål, som er å kunne overføre data effektivt og trygt mellom ende-systemer, men gjennom forskjellige midler.

En teknikk for å unngå pakketap i overbelastede nettverk er å utstyre mellomnoder (som rutere) med buffere som kan holde tilbake pakker og videresende dem senere – i stedet for å forkaste dem. Man har imidlertid oppdaget at mye bufring faktisk kan føre til dårligere nettverksytelse enn ingen buffering i det hele tatt. Hvordan kan dette ha seg, tror du?

Pakkene vil bli lagret i bufferet dersom nodene som sender pakker til ruteren overbelaster CPU-en eller den utgående linken til mottaker nodene. Bufferet i ruteren sørger for at pakkene som kommer gjennom ruteren ikke blir forkastet dersom det skjer en form for metning. For at det skal fungere slik må dette bufferet være så stort at det kan holde på alle ekstra pakkene.

Hvis vi antar at dette bufferet er uendelig stort så er det garantert at alle pakkene som kommer ikke blir forkastet. I en slik modell vil gjennomsnittlig ventetid for en ny pakke som kommer til ruteren nærme seg uendelig.

Denne modellen er fysisk umulig og derfor urealistisk, men dette skaper en diskusjon over bruk av buffer i rutere. Det den kan forklare oss, er at store buffer ikke er optimalt. Rutere med ingen buffer vil derfor gi bedre ytelse. Slik rutere kan ta i bruk retransmission dersom en pakke ikke får en ACK etter et gitt tidsrom. I dette tilfelle vil mottaker noden måtte vente på en pakke, men den vil eventuelt få den etterspurte pakken.

SPØRSMÅL 3

TCPs metningskontrollalgoritme sørger for at mellomliggende noder i internettet ikke alltid er mettet; de opplever kun midlertidig metning.

3.11.3 Persistent metning

Hva betyr det at en mellomliggende node alltid er mettet (opplever persistent metning)? Hva er én konsekvens av persistent metning?

Hvis en mellomliggende node alltid er mettet (opplever persistent metning) betyr det at noden alltid har en stor kø av ventende pakker og klarer ikke å behanle alle pakkene som kommer inn i en tilstrekkelig hastighet i lang periode av tid.

Dette fører til at ventetiden øker for hver nye pakke som legges til i køen, og dette kan jo da føre til at ventetiden blir uakseptabelt lang og at pakker da til slutt kan bli forkastet eller mistet

Så en konsekvens av persistent metning er at det kan føre til at nettverksytelsen blir dårligere, forsinkelser og pakketap øker. Dette kan igjen føre til at tjenester som er avhengig av pålitelighet og rask dataoverføring (spill, videostreaming) blir påvirket negativt

Hvorfor kan mellomliggende noder bli overbelastet hvis man ikke har metningskontroll?

Dersom man ikke har metningskontroll vil noder bli overbelastet av pakker som blir sendt igjen, og det vil bli "kaos" i noden. Noden vil da få store mengder data å håndtere samtidig, og vil bli overbelastet.

Kan du tenke deg en ytelsesbegrensning som metningskontrollalgoritmen New Reno har, når dens metningsvindu er stort fordi nettverket har stor throughput?

Et problem jeg kan tenke meg at AMID systemer i New Renos har under flere tilkoblinger er at AMID-systemet prøver å være "fair" ved at ssthresh halveres for hvert pakketap, og vil deretter kunne delegerere mer bandwidth til en annen tilkobling. New Renos fikser noe av dette ved at den ikke prøver å halvere flere ganger, som kan gjøre det mindre fair, og mindre ytelse for andre koblinger.

3.12 Routing grunnleggende

Spørsmål 1

Ruting er en grunnleggende Internett-funksjon som endrer seg mye med den økende størrelsen på Internettet. Vennligst vurder de følgende 3 spørsmål.

Vi forbinder driften av en ruter på Internett med de to funksjonene Ruting og Forwarding (Videresending). Vennligst vurder rollene deres og forklar hvorfor en av dem må utføres på en veldig rask og effektiv måte, mens den andre ikke er like tidskritisk.

Ruting og videresending er begge fundamentale funksjoner i en nettverksruter, men de har forskjellige roller og tidskrav

Ruting er prosessen der ruten til en datapakke bestemmes. Dette gjøres ved å bruke rutingtabellen, som inneholder informasjon om veier til ulike vettverk. Ruting er en dynamisk prosess, og rutingtabellen kan endres basert på nettverksforhold, feks. Trafikkbelastning og nedetid. Videresending på den andre siden er den faktiske prosessen med å flytte

datapakker fra ruterens inngangsportal til utgangsportalen, basert på informasjonen i rutingtabellen.

Når det gjelder tidskrav, er videresending mer tidskritisk enn ruting. Grunnen til dette er at videresending skjer i "sanntid" for hver enkelt pakke som passerer gjennom ruten. Hvis videresending ikke er rask og effektiv, kan det føre til betydelig forsinkelse (latency) og potensiell pakketap, noe som påvirker den generelle ytelsen til nettverket.

Ruting, derimot er ikke like tidskritisk. Selv om det er viktig at rutingprosessen er nøyaktig for å sikre optimal nettverksytelse, skjer ikke ruting på en per-pakke basis som videresending. I stedet blir rutingtabellen oppdatert periodisk eller når nettverksforholdene endres.

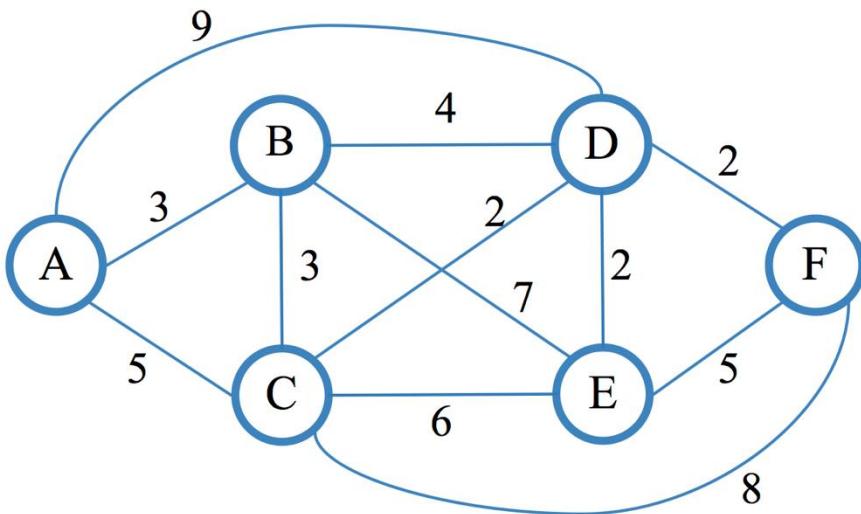
En av funksjonene ovenfor kan faktisk implementeres på en annen datamaskin enn selve ruten. Det er da typisk at denne datamaskinen utfører funksjonen for flere rutere (dette er faktisk vanlig praksis i mange kommersielle nettverk i dag). Hvilken funksjon er det, og hvilke 2 fordeler kan du tenke deg?

Routing kan implementeres på en annen maskin enn selve ruten. En av fordelene ved å sentralisere routingen til en datamaskin er at den kan didikere ressurser samtidig som at resten av nettverket frigjøres for oppgaven. En annen fordel er at hvis alle enheter får rutingtabellen sin fra ett sted, så vil informasjonen om endringer, som at en node går ned, deles raskere enn hvis denne informasjonen måtte spres seg rundt fra maskin til maskin.

Grunnleggende rutingalgoritmer som LSR og DVR inkluderer trinn som krever at de deler informasjon med alle andre rutere i nettverket. Størrelsen på Internett gjør det umulig å utføre en slik prosedyre for alle rutere på Internett. Hva kan du gjøre for å bruke algoritmer som dette på Internett?

For å gjøre LSR og DVR mulig på internettet kan man i stedet for å utveksle individuelle ruter, dele nettverket inn i mindre blokker, og dele ruter mellom disse blokkene. På den måten kan man redusere antall rutere som må utveksles mellom rutere og reduserer dermed informasjonsutvekslingen.

Spørsmål 2



Forklar hvordan Dijkstra's algoritme funker og hvis hvordan du bruker den til å finne den beste stien fra A til F i bildet. I hvilken rekkefølge vil nodene B, C, D, E og F besøkes?

Dijkstras algoritme fungerer ved å utforske grafen stegvis og oppdatere avstanene til nodene fra startnoden. Den starte med å sette avstanden til alle andre noder som uendelig. Så sjekker den avstandene til alle naboen til startnoden, og lagrer disse. Den velger deretter den naboen med kortest avstand fra start, og setter denne til å være den aktive noden. Så sjekker den alle naboen til den aktive noden, og dersom avstanden fra startnoden, via denne naboen er kortere enn direkte fra start (eller noden ikke kan nås fra startnoden og derfor fortsatt har "uendelig" avstand), oppdateres avstanden til den nye verdien. Slik vil den korteste stien lagres, og dette fortsetter til alle noder er besøkt.

Den korteste veien mellom A og F går gjennom A-B-D-F med en totalavstand på 9. Fra A starter jeg med å gi alle direkte tilstøtende noder en etikett med avstanden til A. Så vil D sin etikett være D(9,A), B sin etikett vil være B(3,A), og C sin etikett vil være C(5,A). Jeg velger B og fra B vil jeg oppdatere etikettene, slik at C blir C(6,B) og D blir D(7,B). Jeg merker også E som E(10,B). Jeg velger C og for D velger jeg den forrige etiketten siden den er mindre, D(7,B), og jeg merker E som E(10,C) og F som F(14,C). Jeg velger D og oppdaterer F til F(9,D) og E til E(9,D). Dette når frem til destinasjonen F. Merk at når jeg gjør etikettene permanente, velger jeg alltid den minste avstanden.

Rekkefølge: B, C, D, E, F

Hvordan fungerer en Distance Vector Routing? Oppgi og forklar stegene.

1. Hver node sender ut sin egen distansevektor, som inneholder informasjon om den korteste avstanden den kjenner til alle andre noder i nettverket
2. Når en kode mottar distansevektorer fra sine naboor, oppdaterer den sin egen deistansevektor ved å velge den korteste avstanden til hver destinasjonsnode
3. Når en node oppdaterer sin distansevektor, informerer den sine naboor om endringen ved å sende ut oppdatert informasjon.

- Denne utvekslingen av distansevektorer fortsetter iterativt, og hver node oppdaterer sin distansevektor basert på informasjon fra naboen, inntil alle nodene konvergerer til den beste ruteinformasjon

Du er leder for et selskap som bygger et nettverk i et underutviklet område hvor ressurser som energi er knappe. I dette området kan strømbruddene også føre til koblingsfeil. Hva er de 2 hovedegenskapene du vil vurdere for en rutingsalgoritme som bør fungere godt innenfor dette området. Hvorfor?

I et område hvor ressurser som energi er knappe, og strømbrudd kan føre til koblingsfeil, vil to egenskaper være viktige for en rutingsalgoritme:

- Robusthet: algoritmen bør være robust nok til å håndtere endringer i nettverkets tilstand. Den bør være i stand til å raskt oppdage brudd ved koblinger og tilpasse seg disse endringene ved å finne alternative stier.
- Energieffektivt: siden energiressursene er knappe, bør rutingsalgoritmen optimaliseres for energiforbruk. Den kan gjøre dette ved å velge stier som bruker mindre energi, eller ved å minimere antall overføringer, som kan være energikrevende

Spørsmål 3

3.12.1 Link State Routing

Se for deg et sammenkoblet nettverk med et hundretalls noder hvor hver node har opptil 5 naboor. Routing-algoritmen som brukes er Link State Routing (LSR) og den bruker Dijkstras korteste sti først algoritme for å regne ut rutene.

Forklar hvordan Dijksta's algoritme kan brukes til å regne ut rutene i Link State Routing.

- Hver node i nettverket samler informasjon om avstanden til sine nærmeste naboor
- Denne informasjonen, også kalt ruternes "link state" distribueres til alle andre rutere i nettverket slik at alle ruterne har oppdatert informasjon om nettverkets topologi.
- Deretter kan Djikstras algoritme benyttes til å finne den raskete veien til alle nodene i nettverket fra en gitt node
- Rutetabellene oppdateres basert på resultatet av algoritmen.

En node i nettverket kan måle distanse fra sine direkte naboor ved å sende pakker og måle tiden det tar før den mottar en respons. Disse målings-pakkene kan bli køet som vanlige data-pakker eller "hoppe" over køen på både målings-noden og responderende node. Hvilken forskjell vil valget ha for routing?

Hvis målepasser blir passert i kø som vanlige datapakker, kan det føre til forsinkelser i å få avstandsmålinger og respondere på dem. Denne forsinkelsen kan påvirke nøyaktigheten av de målte avstandene og potensielt føre til suboptimale rutingsbeslutninger

Imidlertid, hvis målepakkene "hopper køen" både på målende og svarende noder, får de høyere prioritet og opplever lavere ventetid. Denne prioriteringen sikrer mer nøyaktige avstandsmålinger, noe som muliggjør mer presise og effektive rutingsbeslutninger.

Kan du identifisere og forklare en fordel med simpelthen å fastsette distansen til en direkte nabo som 1, noe mange virkelige nettverk gjør?

En fordel med å fikse avstanden til en direkte nabo til 1, noe mange virkelige nettverk gjør, er at den forenkler rutingsberegningene.

Ved å betrakte alle direkte nabøer som liker langt unna, kan rutingsalgoritmen behandle dem ensartet uten behov for komplekse avstandsberegninger basert på ventetid eller andre metrikker. Denne forenklingen reduserer den beregningsmessige belastningen og gjør ruteprosessen mer rett frem og effektiv.

4 Oppsummering av hvert kapittel i boka

4.1 2.1 Processes

Computers do several things at the same time. Requests come in, and the server checks to see if the page needed is in the cache. If it is, it is sent back; if not, a disc request is started to fetch it. However, from the CPU's perspective, disk requests take eternity. While waiting for a disk request to complete, many more requests may come in. If there are multiple disks present some or all of the newer ones may be fired off to other disks long before the first request is satisfied. Processes can help here.

The CPU switches unnoticeably from process to process quickly. But the CPU is at every moment only running one process.

A process

Example:

A computer scientist has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program, that is, an algorithm expressed in some suitable notation, the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake. Now imagine that the computer scientist's son

comes running in screaming his head off, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher-priority process (administering medical care), each having a different program (recipe versus first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

Process Model

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of sequential processes, or just processes for short. ***A process is just an instance of an executing program, including the current values of the program counter, registers, and variables.*** The processes are not running parallel, but the CPU is switching from program to program.

Process creation

Initially

1. System initialization.
2. Execution of a process-creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

Fork

- Creates an exact clone of the calling process. After the fork the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. Usually, the child process then executes execve or a similar system call to change its memory image and run a new program.
- For example, when a user types a command, say, sort, to the shell, the shell forks off a child process and the child executes sort. The reason for this two-step process is to allow the child to manipulate its file descriptors after the fork but before the execve in order to accomplish redirection of standard input, standard output, and standard error

4.2 2.2 Threads

Threads are a way of organizing code execution within a process. A thread is a lightweight process that can be scheduled independently by the operating system, and each thread can execute a different piece of code simultaneously.

The chapter first introduces the concept of threads and how they differ from processes. **Threads share the same memory space as the process that created them, which makes communication and synchronization between threads much faster and more efficient than between processes.** Threads can also be created and destroyed more quickly than processes, which makes them ideal for tasks that require frequent switching between tasks or tasks that need to run concurrently.

The author then discusses the benefits of using threads, such as improved performance, increased responsiveness, and easier programming. The chapter also covers thread synchronization mechanisms, such as locks, semaphores, and monitors, which allow threads to coordinate their activities and access shared resources without interfering with each other.

The most important takeaway from this chapter is that threads are an essential component of modern operating systems, and understanding how to use them effectively can improve performance and responsiveness of applications. Developers need to carefully manage thread synchronization to avoid issues like race conditions and deadlocks that can result in incorrect behavior or system crashes.

4.3 2.4 Scheduling

The chapter begins by discussing the importance of scheduling in a multiprogramming environment, where multiple processes or threads are running concurrently on a computer system.

The author then introduces the concept of a scheduler, which is a component of the operating system responsible for deciding which process or thread to run next on the CPU. The scheduler uses a set of algorithms to determine which process or thread should be given CPU time, and for how long.

The chapter then discusses several scheduling algorithms, including First-Come, First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. FCFS is the simplest algorithm, where the first process that arrives is given the CPU until it finishes its execution. SJF is a preemptive algorithm that selects the process with the shortest expected execution time. RR is a preemptive algorithm that allocates CPU time in small time slices, and each process is given a time slice to execute. Priority Scheduling is a non-preemptive algorithm where each process is assigned a priority, and the process with the highest priority is given the CPU time.

The author also discusses several issues with scheduling, such as starvation, where a low-priority process may never get CPU time, and aging, where the priority of a process increases as it waits for CPU time.

The most important takeaway from this chapter is that scheduling is a critical component of modern operating systems, and the choice of scheduling algorithm can have a significant impact on system performance and responsiveness. The author highlights the importance of selecting a scheduling algorithm that is appropriate for the system's workload and characteristics. Additionally, the chapter emphasizes the need to address issues such as starvation and aging to ensure fair and efficient CPU utilization.

4.4 3.3 Virtual Memory

The chapter begins by introducing the concept of demand paging, where pages of memory are only loaded from disk when they are needed. This is more efficient than loading the entire program into memory at once, as it reduces the amount of disk access required.

The chapter then goes on to explain how virtual memory is managed. **Each program is given its own virtual address space, which is mapped to physical memory by the operating system. The operating system keeps track of which portions of the virtual address space are currently in physical memory, and which are stored on disk. When a program attempts to access a portion of memory that is not currently in physical memory, a page fault occurs and the necessary page is loaded from disk.**

The chapter also covers the concept of page replacement algorithms, which determine which pages to remove from physical memory when it becomes full. The two most common algorithms are the Least Recently Used (LRU) algorithm, which removes the page that has not been accessed for the longest time, and the Second Chance algorithm, which gives each page a chance to stay in memory by checking if it has been referenced recently.

The chapter concludes by discussing the benefits and drawbacks of virtual memory. The main benefit is that it allows programs to use more memory than is physically available, which can improve performance. However, virtual memory also introduces overhead due to the need to manage the mapping between virtual and physical memory, and the potential for performance degradation if too much swapping between disk and memory occurs.

4.5 3.7 Segmentation

Segmentation is an alternative memory management scheme to paging.

The chapter begins by introducing the concept of a segment, which is a logical unit of memory that consists of a contiguous range of addresses. Unlike pages in a paging scheme, segments are variable in size and can correspond to different parts of a program, such as the code segment, data segment, and stack segment.

The chapter then goes on to explain how segmentation works. When a program is loaded into memory, it is divided into segments and each segment is assigned a unique segment identifier. The segment identifier and an offset value are used to generate a virtual address, which is then translated to a physical address by the operating system.

The chapter also covers the concept of protection in a segmented system. Each segment is assigned a protection level, which determines whether the segment can be read from, written to, or executed. This provides a higher level of security than a paging scheme, where pages are typically either in memory or on disk and are not protected on a per-segment basis.

The chapter concludes by discussing the benefits and drawbacks of segmentation. One benefit of segmentation is that it allows for more efficient memory allocation and can improve program performance by reducing fragmentation. However, segmentation can also be more complex to implement and manage than a paging scheme and can require additional hardware support for protection and translation. Additionally, segmentation can suffer from similar issues to paging, such as thrashing if too many segments are loaded into memory at once.

4.6 4.1 Files - Modern Operating systems

File systems are used to manage and organize data stored on a computer's storage devices. The chapter covers the basic concepts of file systems, including how files are organized, how they are accessed, and the different types of file systems that exist.

One of the most important concepts in file systems is the file hierarchy, which is the way files are organized into directories or folders. The hierarchy typically starts with a root directory, which contains all other directories and files. Directories can contain other directories and files, and this nesting can continue to create a complex hierarchy of files.

Another important concept in file systems is the file metadata, which includes information about the file such as its name, size, creation date, and access permissions. File metadata is typically stored in a directory or file entry, which is a data structure that contains information about a file or directory.

The chapter also covers the different types of file systems that exist, including local file systems, network file systems, and distributed file systems. Local file systems are used to manage data stored on a single computer, while network file systems allow data to be shared between multiple computers on a network. Distributed file systems allow data to be shared across multiple computers or storage devices, even if they are not physically located in the same place.

The chapter also introduces the concept of file operations, including opening, reading, writing, and closing files. It explains how file systems manage file access, including file locking to prevent multiple processes from accessing the same file simultaneously.

Finally, the chapter discusses some of the challenges and trade-offs involved in designing file systems, including the need to balance performance, security, and reliability. For example, some file systems may prioritize performance by optimizing for fast read and write operations, while others may prioritize security by using encryption or access control mechanisms to protect sensitive data.

Overall, the chapter provides a comprehensive overview of file systems and their role in managing and organizing data on modern operating systems. It highlights the importance of file systems in enabling users and applications to work with data, and emphasizes the need for careful design and implementation to ensure optimal performance, security, and reliability.

4.7 4.2 Directories

A directory is a special type of file that is used to organize and manage files in a hierarchical manner. Directories are essential for organizing files and making them easy to find and access.

The chapter starts by describing the structure of a directory in Unix operating systems. Directories in Unix are arranged in a tree-like structure, with a root directory at the top. Each directory in the tree can contain subdirectories and files. Directories are identified by their absolute path, which specifies the location of the directory in the directory tree.

The chapter then discusses the operations that can be performed on directories, including creating a new directory, deleting a directory, moving a directory, and renaming a directory. Directories can also be used to control access to files by setting permissions on the directory itself.

The chapter also covers the concept of a current working directory, which is the directory that a user is currently working in. This is important for managing files and navigating the directory tree.

The chapter then moves on to discuss directories in other operating systems, including Windows and Mac OS X. While the basic structure and functionality of directories are similar across different operating systems, there are some differences in the way directories are organized and managed.

Finally, the chapter discusses some advanced topics related to directories, including symbolic links, hard links, and mount points. Symbolic links are special files that

point to other files or directories, while hard links are multiple names for the same file. Mount points are used to mount remote file systems onto the local file system.

In conclusion, Chapter 4.2 of "Modern Operating Systems" provides a comprehensive overview of directories in operating systems. Directories are an essential component of any file system, providing a way to organize and manage files in a hierarchical manner. The chapter covers the structure of directories, operations that can be performed on directories, the concept of a current working directory, directories in different operating systems, and some advanced topics

4.8 5.1 Principles of I/O Hardware

It is responsible for transferring data between the computer and external devices.

The chapter begins by introducing the different types of I/O devices, such as keyboards, displays, printers, and storage devices, and explains how they are connected to the computer through various interfaces, such as USB, SATA, and Ethernet.

The chapter then goes on to explain the concept of I/O controllers, which are specialized hardware devices that manage communication between the computer and I/O devices. I/O controllers typically include a buffer, which temporarily stores data during the transfer process, and a set of registers, which control the I/O operation and status.

The chapter also covers the concept of interrupt-driven I/O, which is a common technique used to manage I/O operations. When an I/O operation is initiated, the CPU is free to perform other tasks until the I/O controller generates an interrupt to signal that the operation has completed. The interrupt handler then retrieves the data from the I/O buffer and performs any necessary processing.

The chapter concludes by discussing the benefits and drawbacks of different I/O strategies. One benefit of interrupt-driven I/O is that it allows for efficient use of CPU time, as the CPU can perform other tasks while waiting for I/O operations to complete. However, interrupt-driven I/O can also introduce overhead due to the need to handle interrupts and context switches.

Overall, understanding the principles of I/O hardware is important for developing efficient and effective operating systems, as I/O performance can significantly impact overall system performance.

4.9 5.2 Principles of I/O Software

The primary goal of I/O software is to provide a uniform interface to device drivers and to manage the flow of data between devices and user programs. To achieve

this, I/O software must be able to handle a wide variety of devices and provide a consistent view of them to the user. This requires the implementation of a set of principles that guide the design of I/O software.

The principles of I/O software are organized around three main themes:

- (1) device independence (2) uniform naming (3) buffering.

Device independence means that the I/O software should not be dependent on the specific hardware of a device, but instead provide a consistent interface to all devices. Uniform naming means that all devices should be named in a consistent manner to avoid confusion. Buffers are used to temporarily store data that is being transferred between devices and user programs.

The I/O software is organized into several layers. The layers are designed to provide a modular and hierarchical approach to the implementation of I/O software. The layers are (1) user-level I/O software, (2) device-independent I/O software, and (3) device drivers. User-level I/O software provides a high-level interface to user programs. Device-independent I/O software provides a consistent view of devices to user-level I/O software. Device drivers are responsible for managing the interaction between the device hardware and the device-independent I/O software.

The kernel I/O subsystem is responsible for managing I/O operations at the kernel level. The kernel I/O subsystem provides an interface to device drivers, handles interrupts, and manages the flow of data between devices and user programs. The kernel I/O subsystem is an essential part of the operating system and must be carefully designed to provide efficient and reliable I/O operations.

In conclusion, chapter 5.2 of "Modern Operating Systems" by Tannenbaum provides an overview of the principles of I/O software. The chapter discusses the goals of I/O software, the principles that guide the design of I/O software

4.10 5.3 I/O Software Layers

It outlines the responsibilities of each layer and the techniques used to manage I/O devices efficiently.

The I/O software layers are divided into two categories: the device-independent and device-dependent layers. The device-independent layers provide an interface between the device-dependent layers and the user application programs. These layers are responsible for handling I/O requests, managing the buffers, and translating data formats between different devices.

The device-independent layers consist of three components: the file system, the generic I/O software, and the device drivers. The file system provides an interface to

the user to access data stored on the disk. The generic I/O software provides a uniform interface for all I/O operations. The device drivers are responsible for communicating with the hardware devices and implementing device-specific operations.

The device-dependent layers are responsible for managing the hardware devices and the device drivers. The device-dependent layers consist of three components: the device drivers, the interrupt handlers, and the device controllers. The device drivers are responsible for managing the devices, and the interrupt handlers are responsible for handling the interrupts generated by the devices. The device controllers are responsible for controlling the devices and communicating with the device drivers.

The I/O system uses various techniques to manage I/O devices efficiently. One such technique is buffering, which involves the use of buffers to store data temporarily during the I/O process. This allows for more efficient use of the CPU and I/O devices. Another technique is spooling, which involves using a disk as a buffer to store output data until it can be printed or processed by another device.

In conclusion, chapter 5.3 of "Modern Operating Systems" by Tannenbaum provides an overview of the I/O software layers and the techniques used to manage I/O devices efficiently. The device-independent layers provide an interface between the device-dependent layers and the user application programs, while the device-dependent layers manage the hardware devices and the device drivers. The I/O system uses buffering and spooling techniques to manage I/O devices efficiently.

4.11 5.4 Disks

The chapter discusses the different components of a disk, including the platters, read/write heads, and the controller. It also explores the organization of data on a disk, including the partitioning of disks and the use of file systems.

One of the most important concepts in disk organization is the use of partitions. Partitions divide a single physical disk into multiple logical disks, each with its own file system. This allows for more efficient use of the disk space and provides some level of isolation between different operating systems and data sets. The chapter also describes the use of disk quotas to limit the amount of disk space that can be used by specific users or groups.

The chapter then delves into the specifics of disk access and how it is managed by the operating system. The operating system maintains a disk block allocation table that keeps track of which blocks are free and which are in use by files or directories. When a file is created, the operating system allocates a set of contiguous blocks on the disk to store the file's data. The chapter also discusses the use of caching to

improve disk performance, as well as the use of techniques such as fragmentation and defragmentation to manage the allocation of disk space.

Another important concept covered in the chapter is RAID (Redundant Array of Inexpensive Disks), which is a technique for combining multiple physical disks into a single logical unit for improved performance, reliability, or both. The chapter describes several different RAID configurations, including RAID 0 (striping), RAID 1 (mirroring), and RAID 5 (striping with parity). Each configuration has its own advantages and trade-offs in terms of performance, reliability, and cost.

The chapter concludes with a discussion of disk scheduling, which is the process of deciding which disk requests to service first when multiple requests are pending. The operating system typically uses algorithms such as First-Come-First-Served (FCFS), Shortest Seek Time First (SSTF), and SCAN to prioritize disk requests and optimize disk performance.

In summary, Chapter 5.4 of Modern Operating Systems provides a comprehensive overview of the components, organization, and management of disks in computer systems. The chapter covers important concepts such as partitions, disk access, caching, RAID, and disk scheduling, and provides a foundation for understanding how disk storage is managed by modern operating systems.

4.12 5.6 UI: Keyboard, Mouse, Monitor
4.13 11.3 System Structure

The chapter begins by discussing the challenges of designing an operating system that is flexible, scalable, and reliable.

The author then introduces the concept of a microkernel operating system, which is a modular design that separates the operating system into small, independent components. The microkernel provides only the most essential services, such as interprocess communication, memory management, and scheduling, while other services are implemented as separate modules that run in user space.

The chapter also discusses the benefits of a modular design, such as improved reliability, better scalability, and easier maintenance. However, the author notes that a microkernel design can also introduce performance overhead and make it more difficult to optimize system performance.

The most important takeaway from this chapter is that the design of an operating system plays a critical role in its performance, reliability, and scalability. A modular design, such as a microkernel, can provide several benefits, but it also introduces some challenges that must be carefully considered. Operating system designers need to balance the tradeoffs between modularity, performance, and reliability to create a system that meets the needs of users and applications.

4.14 11.5 Memory management

Memory management is a critical component of an operating system as it provides a virtual address space to each process, which allows processes to operate independently of one another.

The chapter begins by discussing the basics of memory management, including the concept of logical and physical memory. Logical memory is the address space seen by a process, while physical memory is the actual memory installed in a computer. The operating system creates a mapping between the two through the use of page tables.

The chapter then covers the various memory allocation schemes used in modern operating systems, including fixed partitioning, dynamic partitioning, and segmentation.

- Fixed partitioning divides physical memory into fixed-size partitions and allocates each partition to a process.
- Dynamic partitioning, on the other hand, allocates memory to a process as needed.
- Segmentation divides the logical address space of a process into segments, each of which can be independently allocated.

Next, the chapter discusses the concept of virtual memory, which allows the operating system to use disk space as an extension of physical memory. Virtual memory is implemented through the use of demand paging, which only loads pages into memory when they are needed. This technique reduces the amount of physical memory required to run processes, allowing more processes to run concurrently.

The chapter also covers the important concept of memory protection, which is essential for ensuring that processes do not interfere with each other. Memory protection is enforced through the use of page tables, which provide a mechanism for assigning access rights to pages of memory.

Finally, the chapter discusses the issue of memory fragmentation, which can occur in dynamic partitioning schemes. Fragmentation occurs when there is not enough contiguous memory available to allocate to a process. To address this issue, many modern operating systems use a technique called memory compaction, which involves moving memory blocks around to free up contiguous space.

In summary, memory management is a critical component of modern operating systems, and it involves allocating and managing physical and virtual memory for processes. The chapter covers the various memory allocation schemes used in modern operating systems, as well as the important concept of virtual memory and

memory protection. The issue of memory fragmentation is also addressed, along with the technique of memory compaction used to address this problem.

4.15 4.1 The Channel Allocation Problem

It covers the channel allocation problem in communication networks. This problem arises when multiple devices attempt to use the same channel simultaneously, resulting in collisions and degraded network performance. This summary will cover the most important concepts from these chapters.

The channel allocation problem can be addressed using three primary techniques: frequency division multiplexing (FDM), time division multiplexing (TDM), and code division multiplexing (CDM). FDM divides the available bandwidth into multiple non-overlapping frequency bands and assigns each user a separate frequency band. TDM allocates a fixed amount of time to each user and divides the available bandwidth into time slots. CDM assigns a unique code to each user and transmits the coded signal simultaneously.

The efficiency of channel allocation depends on the number of users and the amount of available bandwidth. As the number of users increases, the probability of collisions also increases, leading to decreased network performance. To address this, multiple access techniques such as random access and controlled access can be used.

Random access techniques allow all users to access the network simultaneously, without any pre-defined order. This technique is commonly used in wireless networks, but can result in collisions and decreased performance. Controlled access techniques, on the other hand, use a pre-defined protocol to ensure that only one user transmits at a time, leading to more efficient use of bandwidth.

Carrier sense multiple access (CSMA) is a popular random access technique that uses a carrier sensing mechanism to detect collisions. When a device wants to transmit data, it first listens for any ongoing transmission on the channel. If the channel is free, it begins transmitting. If it detects any ongoing transmission, it waits for a random amount of time before retrying transmission.

Collision detection multiple access (CDMA) is a popular controlled access technique that uses a spread spectrum technique to allow multiple users to transmit simultaneously. CDMA assigns a unique code to each user, which is used to spread the signal across the entire bandwidth. The receiver uses the same code to recover the original signal and distinguish it from other transmissions.

In summary, Chapter 4.1 to 4.1.2 of Tannenbaum's "Computer Networks" v5 provide an overview of the channel allocation problem in communication networks, and

various techniques used to address it. The chapters cover essential topics such as frequency division multiplexing, time division multiplexing, code division multiplexing, multiple access techniques, and carrier sense multiple access and collision detection multiple access. Understanding these concepts is critical for network administrators to design and manage efficient and reliable networks.

4.16 4.8 Data Link Layer Switching

It covers the data link layer switching, its types, and how it operates in a network. This summary will cover the most important concepts from these chapters.

Data link layer switching is a method of forwarding data packets between different networks. Switches are the primary devices used for data link layer switching. They operate at the second layer of the OSI model, and their primary function is to forward packets between different networks.

There are two primary types of data link layer switches: store-and-forward and cut-through. Store-and-forward switches receive the entire data packet before forwarding it to the destination, allowing them to perform error checking before sending the packet. Cut-through switches, on the other hand, start forwarding the packet as soon as they receive the destination address, resulting in lower latency.

Switches maintain a forwarding table that maps the destination MAC address of a packet to a specific port on the switch. When a packet arrives at the switch, the switch looks up the destination MAC address in its forwarding table and forwards the packet to the appropriate port.

Switches also use spanning tree protocol (STP) to prevent network loops. STP creates a logical tree structure by selecting a single switch as the root bridge and blocking redundant links to prevent loops from forming.

Virtual LANs (VLANs) are used to segment a network into smaller broadcast domains. VLANs are created by configuring switches to group specific ports into different VLANs. VLANs allow network administrators to manage network traffic more effectively and improve network security.

Link aggregation control protocol (LACP) allows multiple physical links between switches to be combined into a single logical link. LACP provides increased bandwidth, redundancy, and load balancing between switches.

Ethernet switches operate in full-duplex mode, which allows them to transmit and receive data at the same time. This results in faster and more efficient communication between devices on the network.

In summary, Chapter 4.8 to 4.8.5 of Tannenbaum's "Computer Networks" v5 provide an overview of data link layer switching, including its types, operation, and protocols used to manage network traffic. The chapters cover essential topics such as store-and-forward and cut-through switches, forwarding tables, spanning tree protocol, virtual LANs, and link aggregation control protocol. Understanding data link layer switching is critical for network administrators to design and manage efficient and secure networks.

4.17 5.1 Network Layer Design Issues

It covers the topic of network layer design issues. This summary will cover the most important concepts from these chapters.

The network layer is responsible for providing a logical addressing scheme and routing mechanism to enable communication between hosts in different networks. The design of the network layer involves several important issues, including addressing, routing, fragmentation, and reassembly.

Addressing is a crucial aspect of the network layer design. Each host on a network must have a unique address that identifies it on the network. The most commonly used addressing scheme is the IP addressing scheme, which uses a hierarchical structure of network and host addresses. IPv4 addresses are 32 bits long, while IPv6 addresses are 128 bits long.

Routing is the process of selecting a path for a packet to travel from the source host to the destination host. Routing can be done using different algorithms, including distance-vector routing and link-state routing. The most widely used routing protocol is the Internet Protocol (IP), which is used to route packets between different networks.

Fragmentation and reassembly are used when a packet is too large to be transmitted over a network in a single unit. Fragmentation involves breaking the packet into smaller units, while reassembly involves combining the fragments at the destination host. Fragmentation and reassembly introduce overhead and can impact network performance.

Another important aspect of network layer design is congestion control. Congestion occurs when the network becomes congested due to too much traffic. Congestion control mechanisms, such as flow control and congestion avoidance, are used to prevent congestion and maintain network performance.

Quality of Service (QoS) is also an important consideration in network layer design. QoS ensures that different types of traffic receive different levels of priority based on their importance. QoS mechanisms, such as packet classification, marking, and scheduling, are used to achieve this.

Security is another important aspect of network layer design. Network layer security involves ensuring the confidentiality, integrity, and availability of data transmitted over the network. Techniques such as encryption, authentication, and access control are used to secure the network layer.

In summary, Chapter 5.1 to 5.1.5 of Tannenbaum's "Computer Networks" v5 cover the important issues and concepts involved in network layer design. These chapters cover topics such as addressing, routing, fragmentation, reassembly, congestion control, QoS, and security. By understanding these concepts, network engineers can design and implement effective network layer solutions that meet the requirements of different applications and users.

4.18 5.6 The network Layer in the Internet

It covers the network layer in the internet and its components. This summary will cover the most important concepts from these chapters.

The network layer is responsible for delivering packets from the source to the destination across multiple networks. In the internet, the network layer is implemented using the Internet Protocol (IP). IP is a connectionless, best-effort protocol that provides unreliable, unordered delivery of packets.

IP addresses are used to identify hosts and routers in the network. IPv4 addresses are 32 bits long, while IPv6 addresses are 128 bits long. IP addresses are hierarchically organized, with each address consisting of a network prefix and a host identifier. Subnetting is used to divide a network into smaller subnetworks, which allows for more efficient use of IP addresses.

Routing is the process of selecting the best path for a packet to reach its destination. Routers use routing algorithms to determine the best path based on factors such as the destination address and the cost of the path. Routing protocols are used to exchange routing information between routers to build and maintain routing tables.

The Internet Control Message Protocol (ICMP) is used to send error messages and operational information between devices in the network. ICMP is used by routers to communicate with each other and to diagnose network problems.

The Address Resolution Protocol (ARP) is used to map IP addresses to MAC addresses. When a device wants to send a packet to another device on the same network, it uses ARP to resolve the MAC address of the destination device.

Network Address Translation (NAT) is used to map private IP addresses to public IP addresses. NAT allows devices with private IP addresses to access the internet by sharing a single public IP address.

The Border Gateway Protocol (BGP) is used to exchange routing information between different autonomous systems in the internet. BGP is used by internet service providers (ISPs) to advertise their network routes to other ISPs.

In summary, Chapter 5.6 to 5.6.9 of Tannenbaum's "Computer Networks" v5 provide an overview of the network layer in the internet and its components. The chapters cover essential topics such as IP addressing, subnetting, routing algorithms, routing protocols, ICMP, ARP, NAT, and BGP. Understanding the network layer is critical for network administrators to design and manage efficient and secure networks.

5 Tidligere eksamensoppgaver

Eksamnen 2020

5.1 Funksjonskall vs Systemkall

- Beskriv hvordan funksjonskall fungerer
- Beskriv hvordan systemkall fungerer
- Forklar hvordan de er forskjellige

Funksjonskall og systemkall er to forskjellige mekanismer for å kommunisere med operativsystemet i en datamaskin. Her er en beskrivelse av hvordan hver av dem fungerer og hvordan de er forskjellige:

1. Funksjonskall:

- Funksjonskall er en måte å kalle på en funksjon som er definert i et bibliotek eller en programvaremodul.
- Når du bruker funksjoner fra et bibliotek, inkluderer du vanligvis en header-fil som inneholder deklarasjonene for funksjonene.
- Når koden din når et funksjonskall, hopper den til funksjonens implementasjon i biblioteket og utfører de instruksjonene som er der.
- Etter at funksjonen er fullført, returnerer den kontrollen til den opprinnelige koden som kalte funksjonen.

2. Systemkall:

- Systemkall er en måte å kommunisere med operativsystemkjernen og be om tjenester eller ressurser som er tilgjengelige via operativsystemet.
- Systemkall gir et grensesnitt mellom applikasjoner og operativsystemkjernen, slik at applikasjonene kan be om operativsystemstjenester som filbehandling, nettverkskommunikasjon, minnehåndtering osv.
- Når et program trenger å utføre en operasjon som krever tillatelser fra operativsystemet, for eksempel å åpne en fil eller opprette en nettverkstilkobling, gjør det et systemkall for å be om disse tjenestene.
- Operativsystemkjernen behandler systemkallet og utfører den nødvendige oppgaven på vegne av programmet.

Forskjellene mellom funksjonskall og systemkall inkluderer:

- Kontekst: Funksjonskall skjer innenfor applikasjonskonteksten og utfører instruksjoner som er definert i et bibliotek. Systemkall skjer i operativsystemkonteksten og involverer interaksjon med operativsystemkjernen.

- Tillatelser: Funksjonskall utføres innenfor begrensningene til applikasjonen og har ikke direkte tilgang til systemressurser som krever tillatelser. Systemkall gir tilgang til ressurser og tjenester som er kontrollert av operativsystemet, og de kan kreve tillatelser.
- Abstraksjonsnivå: Funksjonskall gir ofte høyere nivå abstraksjon og er enklere å bruke. Systemkall gir lavnivå abstraksjon og lar programmene kommunisere direkte med operativsystemet.
- Implementasjon: Funksjonskall implementeres vanligvis som en del av programmeringsbiblioteker og kjøres i brukermodus. Systemkall implementeres i operativsystemkjernen og krever en overgang fra brukermodus til kjernemodus for å utføre.

I sum er funksjonskall ment for å kalle funksjoner i applikasjonsbiblioteker for å kalle funksjoner i applikasjonsbiblioteket for å utføre spesifikke oppgaver, mens systemkall brukes for å be om tjenester og ressurser fra OS-kjernen

5.2 Tråder vs prosesser

- Hvilke ressurser må byttes mellom prosesser, men ikke for tråder?
 1. Ressurser som må byttes mellom prosesser, men ikke for tråder:
 - Kontekst: Når en prosessor bytter mellom prosesser, må den lagre og gjenopprette hele konteksten til prosessen, inkludert registerverdier, programteller og annen tilstandsinformasjon. Dette er nødvendig for å sikre at prosessen fortsetter der den slapp når den får tilbake prosessorkontrollen. For tråder er kontekstbytte ikke nødvendig, ettersom alle tråder deler den samme konteksten innenfor prosessen.
 - Adresserom: Prosesser har separate virtuelle adresserom, noe som betyr at minnet til hver prosess er isolert fra hverandre. Ved bytte mellom prosesser må også adressomappingen og minnebeskyttelsesmekanismene byttes ut. For tråder deler de det samme adresserommet, og derfor er ingen bytte av adresserom nødvendig ved trådbytte.
- Er forskjellen mellom prosess og tråd viktig for planlegging på en server med mange brukere og mange prosesser som login.ifi.uio.no? Argumenter for ditt svar.

Ja, forskjellen mellom prosesser og tråder er viktig for planlegging på en server med mange brukere og mange prosesser som login.ifi.uio.no av flere grunner:

- a) Ressursbruk: Hver prosess krever en separat minneallokering, kjøretidskontekst og administrativ overhead. Hvis hver brukerlogginningsforespørrelse ble håndtert som en egen prosess, ville det resultere i høy ressursbruk og potensiell flaskehals. Tråder derimot deler de samme ressursene innenfor en prosess, noe som gjør dem mer ressurseffektive.
- b) Skalerbarhet: Med mange brukere og mange prosesser kan det oppstå utfordringer med å skalere opp ressursbruken og administrasjonen av hver

prosess. Tråder gir bedre skalerbarhet, da de deler ressurser og har lavere overhead. Serveren kan optimalt utnytte flertrådete tilnærmingen for å håndtere flere brukere samtidig.

- c) Kommunikasjon og deling av data: Tråder innenfor en prosess kan enkelt kommunisere og dele data ved å bruke delte variabler og datastrukturer. Dette gjør det enklere å oppnå samarbeid og dele informasjon mellom forskjellige deler av programmet. Med prosesser blir kommunikasjon og datautveksling mer kompleks, vanligvis ved bruk av interprosesskommunikasjon (IPC) mekanismer som meldingskøer eller delt minne.

Så, ved å bruke tråder i stedet for separate prosesser for håndtering av brukerlogginningsforespørslar, kan serveren oppnå bedre ressursutnyttelse, økt skalerbarhet og enklere kommunikasjon og datautveksling mellom de forskjellige trådene. Dette kan bidra til å forbedre ytelsen og effektiviteten til serveren.

Virtuelt minne 2 (2021)

På x86-64-arkitekturen tillater langmodus (64- bits) et mye større virtuelt adresserom enn beskyttet modus (32-bits), selv om de 16 høyordensbitene ikke er i bruk, slik at kun 48 bits faktisk brukes for adressering. Gitt at sidestørrelsen er 4 KiB (4096 bytes), hvor mange sider kan adresseres i hver av de to modusene? Du kan angi svaret ditt som en potens av to. Forklar kort hvordan du kom frem til svaret ditt.

For å beregne antall sider som kan adresseres i hver av modusene, må vi ta hensyn til sidestørrelsen på 4 KiB (4096 bytes).

I langmodus (64-bits):

- 48 bits brukes for adressering, noe som gir 2^{48} mulige unike adresser.
- Deler vi dette med sidestørrelsen på 4096 bytes, får vi: $2^{48} / 4096 = 2^{36}$ sider.

I beskyttet modus (32-bits):

- 32 bits brukes for adressering, noe som gir 2^{32} mulige unike adresser.
- Deler vi dette med sidestørrelsen på 4096 bytes, får vi: $2^{32} / 4096 = 2^{20}$ sider.

Ved å bruke sidestørrelsen og ta hensyn til den faktiske bitbredden som brukes for adressering i hver modus, kan vi beregne antall sider som kan adresseres i hver av de to modusene.

Splitting an address block into networks (2020)

Firmaet ditt har fått tildelt 1024 IPv4-adresser som starter på adressen 9.239.16.0. Firmaet ditt har kontorer på tre forskjellige steder, A, B og C, så du må dele adressene i tre separate, rutbare nettverk, netA, netB og netC. Hver av de tre stedene har akkurat nå behov for 120 offentlige IPv4-adresser, men du

forventer at alle vil trenge mer i fremtiden. Du må velge hvordan du tildeler disse adressene.

Hvor mange adresser tildeler du hvert nettverk, netA, netB og netC? Hva er nettverksadressen og nettmasken for hvert av disse nettverkene? Forklar hvordan du kommer til denne løsningen.

Hvert av kontorene har akkurat nå behov for 120 offentlige IPv4-adresser, men det forventes at de alle vil trenge mer i fremtiden. For å oppfylle dette kravet og gi rom for fremtidig vekst, kan vi tildele hver plassering en /24 blokk. En /24 blokk inneholder 256 adresser, men siden den første adressen er nettverksadressen og den siste adressen er broadcastadressen, er det 254 brukbare adresser. Dette vil være nok til å dekke det nåværende behovet på 120 adresser og gi rom for fremtidig vekst.

Adressene ville bli fordelt som følger:

1. netA: 9.239.16.0/24, nettmaske 255.255.255.0, 254 brukbare adresser
2. netB: 9.239.17.0/24, nettmaske 255.255.255.0, 254 brukbare adresser
3. netC: 9.239.18.0/24, nettmaske 255.255.255.0, 254 brukbare adresser

Hvert påfølgende nettverk starter på det neste ledige /24-nettverket etter det forrige. Så netB starter etter det siste brukbare adressen i netA, og netC starter etter det siste brukbare adressen i netB. Dette er effektiv bruk av adresserommet, og gir også rom for fremtidig vekst ved hvert kontor.

I tillegg vil det fortsatt være en ubrukt blokk (9.239.19.0/24) innenfor den opprinnelige /22-blokken som er tildelt selskapet ditt, som kan brukes for fremtidige behov.

Forklar hvorfor du har valgt å dele adresseområdet ditt i tre nettverk som dette. Vurderte du fremtidig vekst i oppgaven din? Beholdt du noen adresser i reserver? Og hvis ja, hvordan planlegger du å bruke dem?

1. Nåværende behov: Hvert kontor trenger nåværende 120 adresser, noe som er langt under grensen på 254 tilgjengelige adresser i et /24-nettverk.
2. Fremtidig vekst: Selv om hvert kontor bare trenger 120 adresser nå, er det sannsynlig at dette tallet vil vokse i fremtiden. Ved å tildeling av et /24-nettverk til hvert kontor, sørger vi for at det er nok rom for denne veksten uten at vi må rekonfigurere nettverksoppsettet.
5. Effektivitet og enkel administrasjon: Ved å bruke /24-nettverk, forenkles nettverksadministrasjonen. De fleste mennesker er kjent med /24-nettverk, og det er enkelt å identifisere hvilket nettverk en gitt IP-adresse tilhører.
6. Reserverte adresser: Etter å ha tildelt de tre /24-nettverkene, sitter vi igjen med en ubrukt /24-blokk (9.239.19.0/24). Dette gir oss ytterligere rom for

vekst hvis en av de eksisterende plasseringene trenger flere adresser i fremtiden, eller hvis selskapet utvider til en ny plassering.

Planen for de reserverte adressene vil avhenge av selskapets fremtidige behov. Hvis en av de eksisterende plasseringene trenger flere adresser, kan vi tildele den reserverte /24-blokken til den plasseringen. Alternativt, hvis selskapet utvider til en ny plassering, kan vi bruke den reserverte /24-blokken til det nye kontoret.

TCP (2020)

Transmission Control Protocol (TCP) er en transportprotokoll som leverer forbindelsesorienterte, pålitelige byte-streams fra ende til ende som en tjeneste til applikasjonslaget. For mange applikasjoner er det veldig upraktisk at TCP bruker bytestrømmer i stedet for pakker. Forklar hva uttalesen "TCP leverer en byte-stream-tjeneste" betyr.

I TCP blir data sendt som en kontinuerlig strøm av bytes, i stedet for distinkte pakker eller meldinger. Dette betyr at data fra applikasjonslaget ikke er inndelt i separate pakker før de blir sendt over nettverket. I stedet blir dataene bare behandlet som en lang, kontinuerlig sekvens av bytes.

Senderen skriver bytes til TCP-strømmen, og TCP vil deretter pakke disse bytene inn i segmenter (som er TCPs versjon av pakker) og sende dem over nettverket. På mottakersiden vil TCP dekomponere segmentene tilbake til en strøm av bytes. Mottakerapplikasjonen kan da lese fra denne strømmen.

Det som er viktig å merke seg er at siden TCP er en byte-stream-protokoll, er det ingen innebygde grenser mellom meldingene i strømmen. Hvis senderapplikasjonen skriver tre separate meldinger til TCP-strømmen, er det ingen garanti for at mottakerapplikasjonen vil lese de samme meldingene separat. Den kan lese dem alle sammen som en enkelt melding, eller den kan lese en del av den første meldingen og resten av den sammen med den andre meldingen, osv.

Dette er hvorfor uttalesen sier at for mange applikasjoner er det "veldig upraktisk" at TCP bruker byte-streams. For applikasjoner som har behov for å sende data i separate, distinkte meldinger, kan det kreve ekstra arbeid å bruke TCP, fordi applikasjonen selv må legge til noen form for meldingsgrenser i dataene den sender.

5.3 Flytkontroll

- Forklar flytkontrollmekanismen "selective repeat".
- Forklar flytkontrollens kreditt-mekanisme.
- Beskriv forskjellene mellom selective repeat og kreditt-mekanisme. Er det noen forskjell ved håndtering av pakketap, tap av ACKer eller bruk av timeouts? Hvor stort kan sekvensnummerområdet være, og hvor mange av disse sekvensnumrene kan være "in flight" til et gitt tidspunkt?

- Kan du finne et problem med tap av ACK-er i kreditt-mekanismen som ikke eksisterer i selective repeat, og har du et løsningsforslag?

Hva betyr byte-stream-tjenesten for applikasjoner som bruker socket funksjoner (Berkeley-sockets) som send() og recv() for å utveksle data? Hva må programmerere gjøre annerledes når de bruker TCP i stedet for en transportprotokoll som gir pakkebasert tjeneste?

Når man bruker TCP i stedet for en pakkebasert tjeneste, må programmere gjøre følgende endringer og håndtere følgende aspekter:

1. Datainndeling: Applikasjonen må selv håndtere inndelingen av dataene i mindre enheter (pakker eller meldinger) før de blir sendt over TCP-tilkoblingen. Dette innebærer at programmere må håndtere buffering og segmentering av dataene for å sikre at de blir sendt og mottatt korrekt.
2. Datagrenser: Siden TCP sender en kontinuerlig strøm av bytes, må applikasjonen selv definere grensene mellom separate dataenheter. Dette kan gjøres ved å bruke spesielle markører, protokollhodere eller lengdeindikatorer for å identifisere starten og slutten på hver melding eller pakke.
3. Dataintegritet: TCP tilbyr pålitelig dataoverføring, noe som betyr at det automatisk håndterer tap av pakker og gjenoppbygging av tapte data. Dette betyr at programmere ikke trenger å bekymre seg for å implementere disse funksjonene på applikasjonsnivå. Men programmere må likevel være oppmerksomme på at TCP ikke garanterer for tidsriktig levering av data.
4. Feilhåndtering: Programmerere må også håndtere feil og unntak som kan oppstå under TCP-overføringen, for eksempel nettverksfeil, tidsavbrudd eller forbindelsesavbrudd. Dette kan inkludere å gjenopprette forbindelser, håndtere timeouts og implementere mekanismer for å gjenopprette feilaktige eller tapte data.

Hvis en applikasjon bruker TCP, men trenger pakker, hvordan må utvikleren designe applikasjonslagsprotokollen?

Hvis en applikasjon bruker TCP, men trenger pakkebasert levering av data, må utvikleren designe en protokoll på applikasjonslaget. Dette innebærer å segmentere datastrømmen i mindre pakker, legge til et pakkehode med nødvendig informasjon, håndtere buffering og rekonstruksjon av pakkene på mottakersiden, og implementere egne mekanismer for håndtering av feil og protokollhandshaking. Denne protokollen legger et lag av abstraksjon over TCP og tillater applikasjonen å arbeide med pakker i stedet for byte-strømmer.

Er det enkelt å lage en ny transportprotokoll PacketTP med små endringer i TCP, men som leverer en forbindelsesorientert, pålitelig, ende-til-ende

datagramtjeneste? Foreslå minst 3 endringer som er nødvendige og forklar hvorfor de er viktige. Endringene kan være små.

Det er ikke enkelt å lage en ny transportprotokoll basert på små endringer i TCP for å levere en forbindelsesorientert, pålitelig, ende-til-ende datagramtjeneste. TCP er spesifikt designet for byte-orientert strømoverføring. Noen foreslalte endringer kan inkludere segmentering av data, definisjon av datagrenser og implementering av bekreftelsesmekanismer for hvert datagram. Imidlertid kan slike endringer føre til kompleksitet og tap av TCPs innebygde fordeler. I stedet kan det være mer hensiktsmessig å vurdere eksisterende protokoller som UDP som allerede er optimalisert for datagramlevering.

Prosesser (2021)

Forklar hva som skjer når et kontekstbytte mellom to prosesser skjer fordi tiden (timeslice) til den kjørende prosessen er brukt opp.

Når et kontekstbytte mellom to prosesser skjer fordi tiden, også kjent som timeslice, til den kjørende prosessen er brukt opp, betyr det at operativsystemet tar kontrollen fra den nåværende prosessen og gir den til en annen prosess som venter på å bli utført. Dette skjer som en del av tidsplanleggingen for å gi rettferdig og effektiv bruk av prosessoren.

1. Den nåværende prosessen, som har brukt opp sin tildelte timeslice, blir satt i en midlertidig pausestatus. Den nåværende prosessens tilstand, inkludert registerverdier, instruksjonspeker og annen kontekstinformasjon, blir lagret i prosesskontrollblokken (PCB) knyttet til prosessen.
2. Operativsystemet velger deretter en annen prosess som skal kjøres neste. Valget kan være basert på forskjellige tidsplanleggingsalgoritmer, for eksempel Round Robin eller prioritetsplanlegging.
3. Den valgte prosessen, også kjent som neste prosess, får kontrollen over prosessoren. Operativsystemet laster kontekstinformasjonen fra neste prosess sin PCB. Dette inkluderer registerverdier, instruksjonspeker og andre nødvendige data for å gjenoppta kjøringen av prosessen.
4. Den nye prosessen begynner å kjøre, og utførelsen fortsetter fra det punktet der prosessen ble sist avbrutt eller suspendert.
5. Kontekstbyttet er fullført når prosesskontrollinformasjonen er korrekt lastet og den nye prosessen fortsetter sin kjøring. Den nåværende prosessen og den nye prosessen bytter plass i prosessortabellen.

Anta at datamaskinen bare har en enkel prosesseringskjerner. Når en ny prosess startes ved å bruke fork(), bør foreldreprosessen fortsette å kjøre, eller skal den nyopprettede barneprosessen kjøre først? Gi argumenter for avgjørelsen din.

Når en ny prosess startes ved å bruke fork() i en datamaskin med bare en enkelt prosesseringskjerne, er det vanlig praksis at både foreldreprosessen og barneprosessen kjører parallelt. Denne tilnærmingen kalles "først-til-mølla" (first-come, first-served). Argumenter for at både foreldreprosessen og barneprosessen bør kjøre parallelt er:

- Optimal utnyttelse av prosessorkapasitet: Ved å la både foredre- og barneprosessen kjøre samtidig kan man utnytte prosessorkapasiteten mer effektivt. Selv om det bare er en enkelt prosessorkjerne tilgjengelig, kan det likevel være perioder der den ene prosessen er inaktiv, for eksempel når den venter på I/O-operasjoner. Ved å tillate parallel kjøring, kan den aktive prosessen fortsette å utføre beregninger mens den andre prosessen venter, og dermed maksimere utnyttelsen av prosessorkraften.
- Raskere gjennomføring av oppgaver: Ved å la både foredre- og barneprosessen kjøre parallelt, kan man potensielt oppnå raskere gjennomføring av oppgavene. Mens barneprosessen utfører sin egen sekvens av instruksjoner, kan foreldreprosessen også fortsette med sin egen sekvens av instruksjoner. Dette kan redusere den totale tiden det tar å fullføre de to oppgavene i forhold til å vente på at barneprosessen skal fullføre før foreldreprosessen kan gjenoppta sin kjøring.
- Bedre brukeropplevelse: Dersom foreldreprosessen er et interaktivt program, kan det være viktig å opprettholde responsiviteten for brukeren. Ved å la både foredre- og barneprosessen kjøre samtidig, kan foreldreprosessen svare på brukerens inndata og gi en jevn brukeropplevelse mens barneprosessen utfører sine oppgaver i bakgrunnen.

Det er imidlertid verdt å merke seg at rekkefølgen der foredre- og barneprosessen kjører parallelt kan variere, og det er ikke garantert at den ene alltid vil kjøre først. Dette kan avhenge av faktorer som operativsystemets planleggingsalgoritme og hvordan ressursene fordeles mellom prosessene.

5.4 Virtuelt minne – m utregning

Virtuelt minne brukes av operativsystemer for å tillate at userspace-prosesser bruker mer minne enn datamaskinen har fysisk RAM. Minnesider som tilhører en prosess, kan enten være plassert på disk eller i RAM. Når de er plassert i RAM, har de som oftest en fysisk adresse som er ulik adressen som brukes av userspace-prosessen de tilhører. Page tables brukes til å oversette fra en logisk adresse som userspace-prosessene bruker til fysiske adresser i RAM, samt for å finne ut om en side for øyeblikket er lastet inn i RAM.

- Prosessoren din har 32-bit adresser
- Du har et adresseringsrom på 4 GB
- Sidestørrelsen på operativsystemet ditt er 1 kilobyte (1024 byte)
- Hver byte kan adresseres
- Du bruker multi-level paging for å la prosesser være opptil 4 GB store

Oppgave

- Forklar (generelt sett) hvordan virtuelle adresser blir oversatt til fysiske adresser med multi-level paging. Forklaringen må inkludere problemer som minneoppslag, page cache, innlasting av sider og begrepet "dirty page". Det er ikke nødvendig å forklare strategier for utskiftning av sider.
- Hvor mange nivåer må din virtuelle adresseoversettelse minst ha med den oppgitte sidestørrelsen? Forklar hvorfor. Ikke glem at page tables og page maps også er begrenset av 1 kilobyte sidestørrelse.

Med multi-level paging kan virtuelle adresser bli oversatt til fysiske adresser i flere steg. Her er en generell forklaring på hvordan dette fungerer:

1. Page Directory:

Det første nivået av oversettelse i multi-level paging er Page Directory. Page Directory er en tabell som inneholder oppføringer for hver stor side (en side som består av flere små sider) i det virtuelle adresserommet. Hver oppføring i Page Directory peker til en Page Table.

2. Page Table:

Page Table er den andre nivået av oversettelse. Den er en tabell som inneholder oppføringer for hver liten side i det virtuelle adresserommet. Hver oppføring i Page Table peker til en fysisk side i RAM.

3. Fysisk adresse:

Den fysiske adressen blir dannet ved å kombinere fysisk sidenummer (indeks i Page Table) og sidens offset.

Når en prosess ønsker å lese eller skrive til en virtuell adresse, følger operativsystemet disse trinnene:

1. Minneoppslag:

Operativsystemet tar den virtuelle adressen og bruker de øverste bitsene til å finne den relevante oppføringen i Page Directory. Denne oppføringen gir adressen til den tilsvarende Page Table.

2. Page Cache:

Hvis siden som prosessen ønsker å aksessere er i page cache, kan oversettelsen fullføres. Page cache er en del av RAM som inneholder nylig brukte sider for rask tilgang. Hvis siden er i page cache, returneres den tilsvarende fysiske adressen direkte.

3. Innlasting av sider:

Hvis siden ikke er i page cache, blir den lastet inn i RAM fra disk. Dette innebærer å finne en ledig fysisk side i RAM, kopiere innholdet fra disk til denne siden, oppdatere Page Table-oppføringen til å peke til den nye fysiske siden og deretter fullføre oversettelsen.

4. Dirty Page:

Når en side i RAM er endret (skriveoperasjon), blir den markert som "dirty". Dette indikerer at siden er forskjellig fra innholdet på disken og må lagres til disk senere for å oppdatere diskens kopi.

Antall nivåer i den virtuelle adresseoversettelsen bestemmes av sidestørrelsen. I dette tilfellet er sidestørrelsen 1 kilobyte (1024 byte). Hver side er på 1024 byte, og siden hver oppføring i Page Table eller Page Directory peker til en side, vil hver oppføring være på 1024 byte. Derfor vil antall nivåer være 2 (Page Directory og Page Table) for å oppnå en virtuell adresseoversettelse som støtter en 4 GB stor prosess.

6 Hvordan regne ut, med eksempler

6.1 Virtuelt minne – 1-nivås minneoppslag

I et virtuelt minnesystem på 32-bits, der hver side bruker 2^{12} (4096) bytes i offset, kan vi finne antall oppføringer (entries) i tabellen ved å bruke følgende beregning:

Antall oppføringer = Totalt antall virtuelle minneadresser / Antall adresser per side

Her er hvordan vi kan komme frem til svaret:

Totalt antall virtuelle minneadresser: I et 32-bits minnesystem er det totalt 2^{32} (4 294 967 296) unike virtuelle minneadresser tilgjengelig.

Antall adresser per side: Hver side bruker 2^{12} (4096) bytes i offset. Siden størrelsen på sidene er den samme som offset-størrelsen, vil antall adresser per side være 2^{12} .

Beregning av antall oppføringer: Deler totalt antall virtuelle minneadresser med antall adresser per side:

$$\text{Antall oppføringer} = (2^{32}) / (2^{12})$$

Forenkler uttrykket ved å trekke fra eksponentene i nevneren:

$$\text{Antall oppføringer} = 2^{(32-12)}$$

Antall oppføringer = 2^{20} Svaret er 2^{20} (1 048 576) oppføringer i tabellen.

Dermed vil det være 1 048 576 entries i tabellen for et virtuelt minnesystem på 32-bits der hver side bruker 2^{12} (4096) bytes i offset.

6.2 Utregning av antall sider som kan adresseres i lang- vs beskyttet-modus

På x86-64-arkitekturen tillater langmodus (64- bits) et mye større virtuelle adresserom enn beskyttet modus (32-bits), selv om de 16 høyordensbitene ikke er i bruk, slik at kun 48 bits faktisk brukes for adressering. Gitt at sidestørrelsen er 4 KiB (4096

bytes), hvor mange sider kan adresseres i hver av de to modusene? Du kan angi svaret ditt som en potens av to. Forklar kort hvordan du kom frem til svaret ditt.

I x86-64-arkitekturen tillater langmodus (64-bits) et større virtuelt adresserom enn beskyttet modus (32-bits), selv om bare 48 av de 64 bitene faktisk brukes for adressering. For å finne antall sider som kan adresseres i hver av de to modusene, må vi først beregne antall bits som brukes til å adressere en side i hvert tilfelle.

1. Beskyttet modus (32-bits):

- a. Totalt antall bits i en 32-bits adresse: 32 bits
- b. Av disse er de 12 lavordensbitene brukt til å adressere byte-indekser innenfor en side (4096 bytes per side).
- c. Antall bits brukt til å adressere sider: $32 - 12 = 20$ bits
- d. Antall sider som kan adresseres: $2^{20} = 1\ 048\ 576$ sider

2. Langmodus (64-bits):

- a. Totalt antall bits i en 64-bits adresse: 64 bits
- b. Av disse er de 12 lavordensbitene brukt til å adressere byte-indekser innenfor en side (4096 bytes per side).
- c. Antall bits brukt til å adressere sider: $64 - 12 = 52$ bits
- d. Av de 52 bitene brukes bare de 48 høyordensbitene faktisk til adressering.
- e. Antall sider som kan adresseres: $2^{48} = 281\ 474\ 976\ 710\ 656$ sider
Så, antall sider som kan adresseres i hver av de to modusene er:
Beskyttet modus (32-bits): 1 048 576 sider Langmodus (64-bits): 281 474 976 710 656 sider

Flernivå-paging legger et nytt indireksjonsnivå til virtuelt minne. Tenk deg en 32-biters arkitektur med 4 KiB (4096 bytes) sidestørrelse. Gitt at systemets totale mengde RAM er 500 MiB og at i gjennomsnitt 100 prosesser kjører på systemet til enhver tid. Hvor mange paging-nivåer vil du bruke? Gi et argument.

I dette tilfellet bruker vi en 32-biters arkitektur med 4 KiB sidestørrelse og en total mengde RAM på 500 MiB. Hvis gjennomsnittlig 100 prosesser kjører på systemet samtidig, må vi avgjøre hvor mange paging-nivåer vi vil bruke.

For å adressere en 32-biters arkitektur har vi tilgjengelig 32 bits for å representer den virtuelle adressen. Vi må derfor finne ut hvor mange bits som skal brukes til å representer sideindeks og sidemasker.

Siden sidestørrelsen er 4 KiB, har vi 12 bits (fordi $2^{12} = 4096$) som brukes til å adressere offsetet innenfor en side.

For å finne antall paging-nivåer, må vi finne ut hvor mange bits som skal brukes til sideindeks. Antall sider som trengs for å adressere den totale mengden RAM kan beregnes som:

$$\text{Antall sider} = \text{Total mengde RAM} / \text{Sidestørrelse}$$

$$\text{Antall sider} = 500 \text{ MiB} / 4 \text{ KiB}$$

$$\text{Antall sider} = 500 \text{ MiB} / 4096 \text{ B}$$

$$\text{Antall sider} = 128\,000 \text{ sider}$$

Vi kan finne antall sider som kan adresseres ved å bruke formelen $2^{(\text{Totalt antall adressebits} - \text{Offsetbits})}$. I dette tilfellet blir Totalt antall adressebits = 32 bits - 12 bits = 20 bits.

$$\text{Antall sider som kan adresseres} = 2^{20} = 1\,048\,576 \text{ sider}$$

Siden vi trenger 1 048 576 sider for å adressere den totale mengden RAM, trenger vi 20 bits for å representere sideindeks (fordi $2^{20} = 1\,048\,576$).

Basert på dette, vil vi trenge to nivåer med flernivå-paging for å representere virtuelt minne. Det første nivået vil representere de øvre bitsene i den virtuelle adressen, som identifiserer den aktuelle tabellen i det andre nivået. Det andre nivået vil representere sideindeksen som brukes til å identifisere den konkrete siden i minnet.

Ved å bruke to nivåer med flernivå-paging kan vi effektivt håndtere virtuelt minne for de gjennomsnittlige 100 prosessene som kjører samtidig på systemet, samtidig som vi utnytter sidestørrelsen og den tilgjengelige adresseplassen på en effektiv måte.

6.3 Velge riktig filsystem

Du har til oppgave å designe et system som skal lagre store datamengder, og som også krever høyst tilgjengelig og redundant lagring. Velg en teknologi som tjener disse formålene og forklar hvordan den gjør det.

En teknologi som kan tjene formålene med å lagre store datamengder og samtidig tilby høy tilgjengelighet og redundans er distribuert lagring med bruk av RAID (Redundant Array of Independent Disks) teknologi.

RAID er en metode for å kombinere flere fysiske harddisker til en enkelt logisk enhet. Det finnes forskjellige RAID-nivåer som tilbyr ulike nivåer av redundans og ytelse, avhengig av behovene til systemet.

En vanlig RAID-konfigurasjon er RAID 5, som bruker striping og paritet.

Her er hvordan RAID 5 fungerer:

1. Striping: Data blir delt opp i mindre biter og distribuert over flere harddisker. Dette gir økt ytelse ved at lesing og skriving kan skje parallelt over flere diskene.
2. Paritet: I RAID 5 blir paritetsinformasjon beregnet og distribuert over de ulike diskene. Paritet er en form for redundans som gjør det mulig å gjenopprette data hvis en av diskene feiler. Paritetsinformasjonen inneholder informasjon om de andre dataene på de tilsvarende posisjonene på de andre diskene.

Fordelene med RAID 5 og distribuert lagring generelt er:

1. Høy tilgjengelighet: Hvis en harddisk svikter, kan systemet fortsette å fungere uten tap av data. Den manglende informasjonen kan gjenoppbygges ved å bruke paritetsinformasjonen på de gjenværende diskene.
2. Skalerbarhet: Distribuert lagring gjør det mulig å legge til flere harddisker etter behov. Dette gjør det enkelt å øke lagringskapasiteten når det er nødvendig.
3. Redundans: Ved bruk av paritet kan RAID 5 gjenopprette data hvis en harddisk svikter. Dette gir en ekstra beskyttelsesmekanisme mot datatap.
4. Ytelse: Striping gjør det mulig å fordele lesing og skriving over flere disker, noe som resulterer i bedre ytelse og raskere dataoverføringshastigheter.

Samlet sett gir distribuert lagring med RAID 5 en pålitelig, redundant og høyt tilgjengelig løsning for å lagre store datamengder. Ved å kombinere flere harddisker til en enhet, kan systemet oppnå både kapasitet og pålitelighet som kreves for slike scenarier.

6.4 Virtuelt minne – Page replacement

What is virtual memory? Why do we use it and how does it work?

- 3: Virtual memory can exceed the physical RAM of a computer and use other storage resources as well. We want to use it to allow processes to use a large amount of memory without a lot of effort for the programmer. It requires hardware support (MMU) that can (a) translate transparently the addresses used by a running process from virtual address to physical address, (b) detect that a virtual address does not have a physical address right now (page fault), and start the page replacement procedure, (c) update the MMU data.

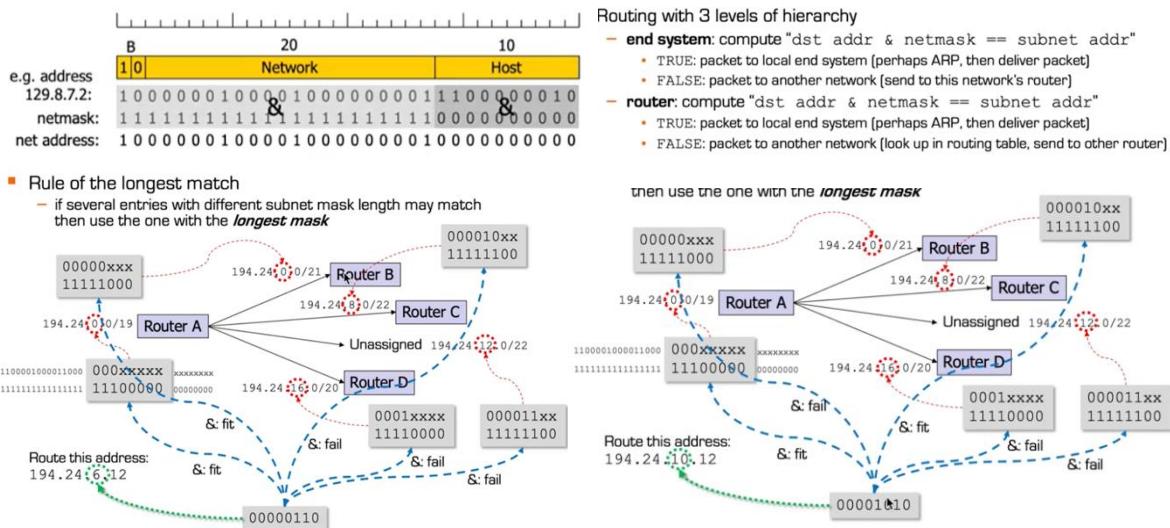
Assume you have a 32-bit memory system, you are using 4 KB pages, and have a request for a particular virtual memory address. Explain how a traditional 1-level memory look-up works, and how the physical address is found.

- You need 12 bits for the offset in the page, so the page table is 2^{20} entries long (approx 1 mio). Each entry contains at least the base address in physical memory (if the address is in physical memory) or info about its location on physical address is found. addressen blir funnet. disk, a dirty bit and a present bit. If written to, dirty is set. If looked up and not present, start the replacement procedure. If the replacement procedure expunges a dirty page, write it to swap space on disk first. If looked up and the page is present, take the 20 bits from the page table entry and add the least significant bits from the requested virtual address, and you have the physical memory address.

When our memory is full, we need a way to replace elements so that we can fetch new requested data. Can you give an example of such an algorithm, and briefly discuss what properties such an algorithm should have?

- A page replacement algorithm should be simple (efficient, low overhead), make optimal decisions (read from and write to disk very rarely). A possible replacement algorithm is LRU, which reshuffles the list of pages in physical memory every time a read or write operation to a page is performed. When a page is addressed, it is moved to the top of the list. When a different page needs to be swapped in, the page that is at the tail end of the sorted list is replaced (ie. written to disk if dirty, and its physical memory overwritten by the requested page from disk).
- According to the task, it is sufficient to write the algorithm's name and the properties "simple"+"good" decisions. The reason is that the task uses the word "such an algorithm" instead of "this algorithm".

6.5 CIDR – Classless InterDomainRouting



For eksempel, en IP-adresse kan være 192.168.1.0/24. Her er '192.168.1.0' IP-adressen, og '24' (etter skråstrekken) er antall bits som er satt til 1 i subnettmasken.

Subnettmasken er en 32-bits verdi som brukes til å skille nettverksdelen av IP-adressen fra verts- eller maskindelen. I eksempelet over er de første 24 bitene (dvs. de tre første oktettene) brukt til å identifisere nettverket. De resterende bitene ($32 - 24 = 8$ bits) er for vertene i nettverket.

Hvis vi skulle sette dette ut i binær form:

192.168.1.0 => 11000000.10101000.00000001.00000000 (binært)

/24 => 11111111.11111111.11111111.00000000 (dette er subnettmasken)

Denne subnettmasken indikerer at de første 24 bitene er reservert for nettverksadressering, og de siste 8 bitene er for individuelle enheter (verter) på nettverket. Så med denne masken, kan du ha $2^8 - 2 = 254$ verter på nettverket (vi trekker 2 fordi adressene der alle värtsbitene är 0 och alla värtsbitene är 1, är reserverade för henholdsvis nätverksadress och broadcastadress).

6.6 Vanlig til binærtall

Decimal to Binary Conversion



Step 1: Divide the given number 13 repeatedly by 2 until you get '0' as the quotient

$$\begin{array}{r} 13 \div 2 = 6 \text{ (Remainder } 1) \\ 6 \div 2 = 3 \text{ (Remainder } 0) \\ 3 \div 2 = 1 \text{ (Remainder } 1) \\ 1 \div 2 = 0 \text{ (Remainder } 1) \end{array}$$

Step 2: Write the remainders in the reverse order 1 1 0

$$\therefore 13_{10} = 1101_2$$

(Decimal) (Binary)

Division by 2	Quotient	Remainder
$13 \div 2$	6	1 (LSB)
$6 \div 2$	3	0
$3 \div 2$	1	1
$1 \div 2$	0	1 (MSB)

6.6.1 Regne antall sideoppføringer (entries) og størrelse på oppføringene

For å beregne antall sideoppføringer må man vite den totale minnestørrelsen og sidestørrelsen. For eksempel hvis minne er 4GB (2^{32}) og sidestørrelsen er 4KB (2^{12}).

Antall sider = totale minnestørrelse/sidestørrelse

Antall sider = $2^{32} / 2^{12}$

Antall sider = 2^{20}

For å beregne størrelsen på en sideoppføring trenger vi å vite hvor mange bits som trengs for å adressere alle sidene i minnet. Dette er vanligvis lik logaritmen (base 2) av antall sider i minnet.

Størrelsen på en sideoppføring (i bits) = $\log_2(\text{Antall sider})$

Størrelsen på en sideoppføring (i bits) = $\log_2(2^{20})$

Størrelsen på en sideoppføring (i bits) = 20 bits (siden $\log_2(2^n) = n$)

6.1 Regne ut nettmaske

Firmaet ditt har fått tildelt 1024 IPv4-adresser som starter på adressen 9.239.16.0. Firmaet ditt har kontorer på tre forskjellige steder, A, B og C, så du må dele adressene i tre separate, rutbare nettverk, netA, netB og netC. Hver av de tre stedene har akkurat nå behov for 120 offentlige IPv4-adresser, men du forventer at alle vil trenge mer i fremtiden. Du må velge hvordan du tildeler disse adressene.

For å finne nettmasken til adresseblokken som er tildelt selskapet ditt, må vi se på antallet IPv4-adresser som er tilgjengelige og finne en passende subnettmask.

Du har fått tildelt 1024 IPv4-adresser, og du trenger tre separate nettverk med hver 120 offentlige IPv4-adresser. Dette betyr at du trenger totalt 360 offentlige IPv4-adresser (120 per nettverk).

Når du deler nettverkene, må du finne en nettmaske som kan støtte det nødvendige antallet IPv4-adresser. For å ha nok adresser, må vi finne det minste subnettet som kan romme 360 adresser. Dette er nærmeste større antall enn 360 er 512, som er 2^9 .

For å finne nettmasken som tilsvarer 512 adresser, kan du se på antallet bits som er nødvendig for å representer 512 (2^9). 9 bits er nok til å representer 512 kombinasjoner (fra 0 til 511), og de resterende 23 bitsene vil bli brukt til å identifisere individuelle enheter i hvert nettverk.

Derfor blir nettmasken for hver av de tre separate nettverkene (netA, netB og netC):

- netA: 9 bits for nettverk + 23 bits for enheter = 32-bit nettmaske
- netB: 9 bits for nettverk + 23 bits for enheter = 32-bit nettmaske
- netC: 9 bits for nettverk + 23 bits for enheter = 32-bit nettmaske

Dette betyr at alle tre nettverkene vil ha en nettmasken på 255.255.255.0. Viktig å merke seg at denne oppdelingen gir deg nok adresser for det nåværende behovet på 120 IPv4-adresser per nettverk, men hvis du forventer at behovet vil øke i fremtiden, kan det være nødvendig å revurdere subnettmasken og adressetildelingen.

- Hvor mange adresser tildeler du hvert nettverk, netA, netB og netC? Hva er nettverksadressen og nettmasken for hvert av disse nettverkene? Forklar hvordan du kommer til denne løsningen.

Du har blitt tildelt 1024 IPv4-adresser, og du ønsker å dele dem i tre separate nettverk (netA, netB og netC) med behov for 120 offentlige IPv4-adresser på hvert sted. Du ønsker også å ta hensyn til fremtidig vekst og beholde noen adresser i reserven.

For å fordele adressene jevnt, kan vi bruke CIDR-notasjon og subnetting-teknikker. Siden du har 1024 adresser, som er 2^{10} , kan vi bruke 10 bits for å identifisere hvert nettverk.

Hvert nettverk vil da ha $2^{10} = 1024$ adresser tilgjengelig. Av disse adressene trenger du 120 offentlige IPv4-adresser på hvert sted. Siden nettverks- og kringkastingsadressene ikke kan brukes, vil hvert nettverk ha 1022 brukbare adresser.

La oss finne nettverksadressen og nettmasken for hvert nettverk:

netA:

- Nettverksadresse: 9.239.16.0
- Nettmasken: 255.255.252.0 (/22)

netB:

- Nettverksadresse: 9.239.20.0
- Nettmasken: 255.255.252.0 (/22)

netC:

- Nettverksadresse: 9.239.24.0
- Nettmasken: 255.255.252.0 (/22)

Ved å bruke /22-nettmasken får hvert nettverk 1022 brukbare adresser, noe som er mer enn nok for de nåværende behovene på 120 adresser per sted.

- Forklar hvorfor du har valgt å dele adresseområdet ditt i tre nettverk som dette. Vurderte du fremtidig vekst i oppgaven din? Beholdt du noen adresser i reserven? Og hvis ja, hvordan planlegger du å bruke dem?

Når det gjelder fremtidig vekst, har du reservert en stor adresseblokk på 1024 adresser. Dette betyr at du har mange ledige adresser tilgjengelig for eventuell utvidelse eller behov som oppstår i fremtiden. Disse adreslene kan tildeles til de ulike nettverkene etter behov.

Ved å dele adresseområdet i tre nettverk på denne måten, sikrer du at hvert sted har tilstrekkelig adressekapasitet for nåværende behov og fremtidig vekst. Du har også tatt hensyn til subnetting-prinsipper for effektiv bruk av IP-adresser og reservert noen adresser i reserven for fremtidige behov.

6.2 Regne ut pagingnivåer

Gitt et adresseringsrom 4GB.

Pagesize på 1024 bytes.

1 page = 1024 bytes, med offset/peker 4 bytes.

Da tar vi pagesize / offset

Da tar vi $1024/4 = 2^8$.

Vi ganger da $2^{8 \cdot \text{antall pagingnivåer} + 10}$ til man finner noe som dekker 4GB. Tallet 10 kommer fra 2^{10} som pagesize.

$4\text{GB} = 2^{32}$ bits

$1\text{ GB} = 1\text{ milliard, altså } 2^{10} * 2^{10} * 2^{10}$

Derfor får vi 3 som indireksjonsnivå fordi $2^{24+10} = 2^{34}$

6.3 Binær vs decimal

BINARY SYSTEM			DECIMAL SYSTEM		
NAME	FACTOR	VALUE IN BYTES	NAME	FACTOR	VALUE IN BYTES
kibibyte (KiB)	2^{10}	1,024	kilobyte (KB)	10^3	1,000
mebibyte (MiB)	2^{20}	1,048,576	megabyte (MB)	10^6	1,000,000
gibibyte (GiB)	2^{30}	1,073,741,824	gigabyte (GB)	10^9	1,000,000,000
tebibyte (TiB)	2^{40}	1,099,511,627,776	terabyte (TB)	10^{12}	1,000,000,000,000
pebibyte (PiB)	2^{50}	1,125,899,906,842,624	petabyte (PB)	10^{15}	1,000,000,000,000,000
exbibyte (EiB)	2^{60}	1,152,921,504,606,846,976	exabyte (EB)	10^{18}	1,000,000,000,000,000,000
zebibyte (ZiB)	2^{70}	1,180,591,620,717,411,303,424	zettabyte (ZB)	10^{21}	1,000,000,000,000,000,000,000
yobibyte (YiB)	2^{80}	1,208,925,819,614,629,174,706,176	yottabyte (YB)	10^{24}	1,000,000,000,000,000,000,000,000

©2018 TECHTARGET. ALL RIGHTS RESERVED. TechTarget