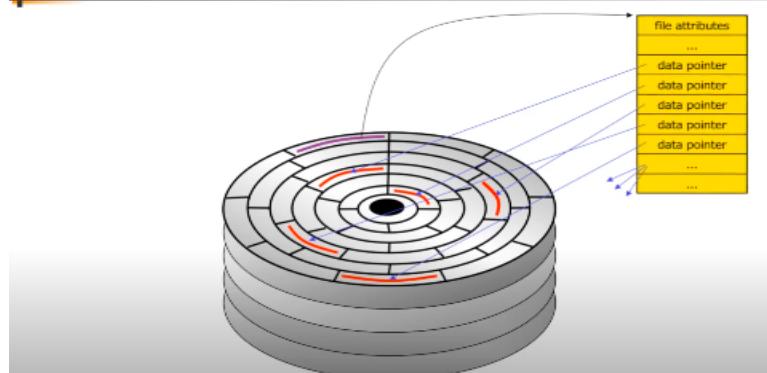


Håndtering av fil-blokker

Management of File Blocks



Nå skal vi se på utfordringen vi har ved at, blokkene som tilhører en gitt fil er blitt spredt rundt cylinderne og sporene i disken. Så f.eks, hvordan skal jeg vite hvor blokk "3" som er knyttet til filen "3", befinner seg i. Den gule tabellen håndterer denne situasjonen, men er annerledes for hvert operativsystem.

For å håndtere slike fil-blokker så må vi vite følgende:

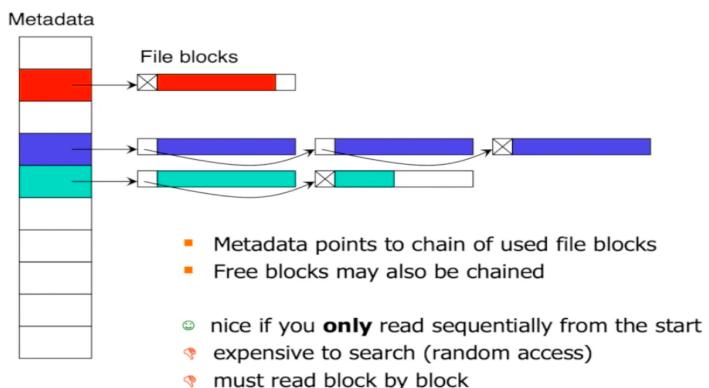
- Vite hvordan vi skal relatere blokkene til filene deres
- Vite hvordan vi skal lokalisere en gitt blokk
- Ivareta ordningen av disse blokkene

Det finnes ulike vendinger for hvordan vi kan gjøre dette som vi skal se under. Vi går da fra de eldste til de nyeste. Disse er da følgende vendinger:

- Chaining in media (Eldste)
- Chaining in a map
- Table of pointers
- Extent-based allocation (Nyeste)

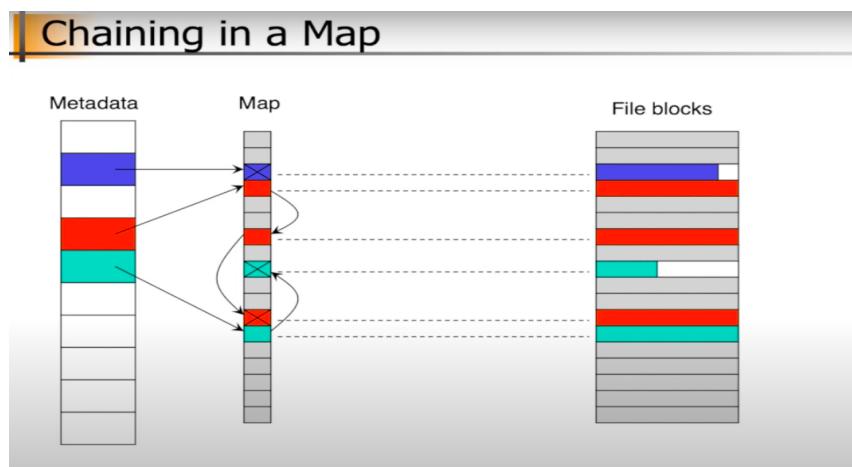
Chaining in media

Chaining in the Media



Metadata, vil si “data om data” altså at beskriver informasjon om dataene som er blitt lagret. I dette tilfelle vil metadata beskrive informasjonen til fil-blokkene. Chaining media fungerer slik at vi benytter av metadata til å peke på en kjede av fil-blokker som er sekvensielt satt opp etter hverandre. Ett eksempel er å se på den blå-metadataen som peker til de blå fil-blokkene etter hverandre. Dette er en bra løsning hvis vi ønsker å lese sekvensielt fra start til slutt. Men problemet er om vi har en type “random access” hvor vi ønsker å lese en gitt fil-blokk fra en gitt metadatabeskrivelse. For problemet ved denne måten er at, vi må lese blokk for blokk. F.eks, la oss si at vi skulle lese den siste “blå-blokken”, vi kan ikke bare direkte gå til den. Siden det er satt opp som en “lenket-liste” så må vi iterere gjennom pekere og pekere til å komme til det siste blokken som vi skal hente. Dette er problemet som ligger ved denne metoden.

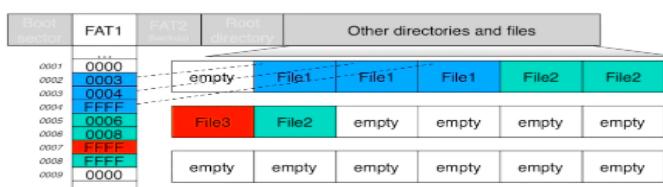
Chaining in a Map



Ideen er at vi har en “Map”/tabell som består av pekere som da peker på de ulike fil-blokkene. Dette er en optimalisering av forrige metode siden vi nå slipper å måtte lese blokk-for-blokk, men kan heller traversere gjennom pekere i “mappen” for å spare tid og finner den blokken vi er ute etter. Ulempen ved dette er at størrelsen på den “mappen” kan bli stor, som gjør at søket for gitte datablokker kan ta tid med tanke på avstanden mellom pekerne i mappen. Da har vi neste løsning, som går ut på “table of pointers”.

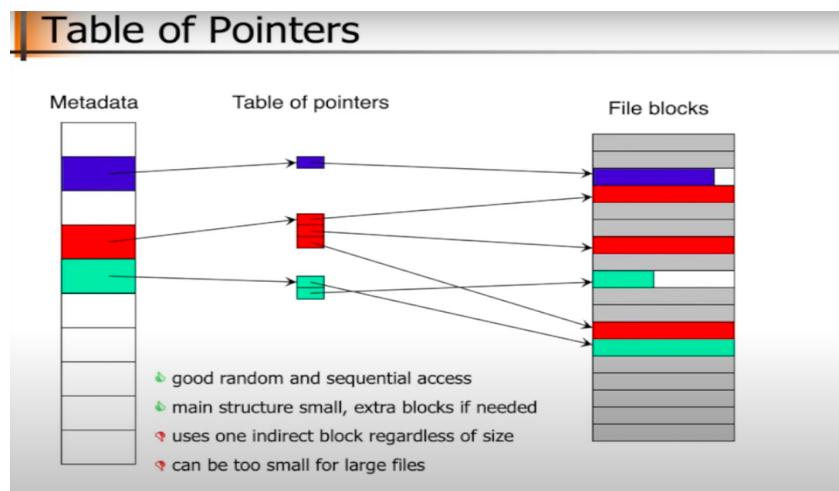
Chaining in a Map: FAT Example

- FAT: File Allocation Table
- Versions FAT12, FAT16, FAT32
 - number indicates number of bits used to identify blocks in partition ($2^{12}, 2^{16}, 2^{32}$)
 - FAT12: Block sizes 512 bytes – 8 KB: max 32 MB partition size
 - FAT16: Block sizes 512 bytes – 64 KB: max 4 GB partition size
 - FAT32: Block sizes 512 bytes – 64 KB: max 2 TB partition size



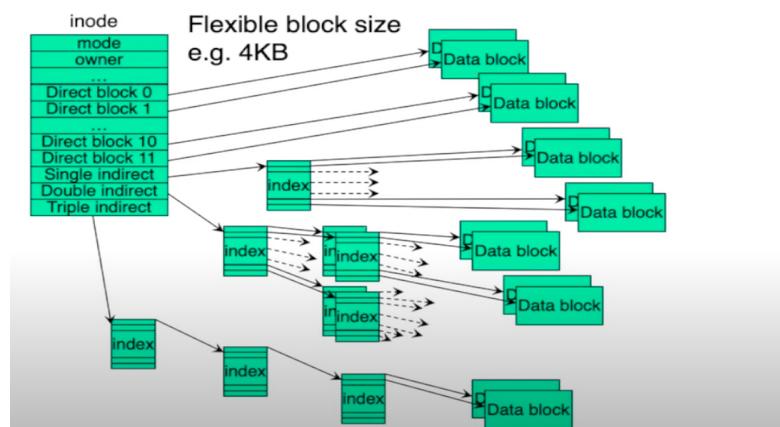
I eksempelet over, så kan vi forstå hvordan dette fungerer enda bedre. Så si at vi skal lese fil-blokkene som er av fargen "lyse-blå". Da har vi at startpekeren er "0002" med en indeks "0003" hvor denne indeksen peker på "File 1" av de lyseblå. Dette gjentas også for "0004" hvor den vil peke på neste blokk. Tilslutt har vi "FFFF" som er da den siste pekeren for blokken for den "lyse-blå".

Table of pointers



Ideen er at vi samler alle pekerne for de samme filene i et sett. Fordelen er at søkingen for en gitt datablokk, blir ganske enkelt ettersom pekerne er i et sett som gjør det enklere å hente pekeren for den datablokk vi skal finne.

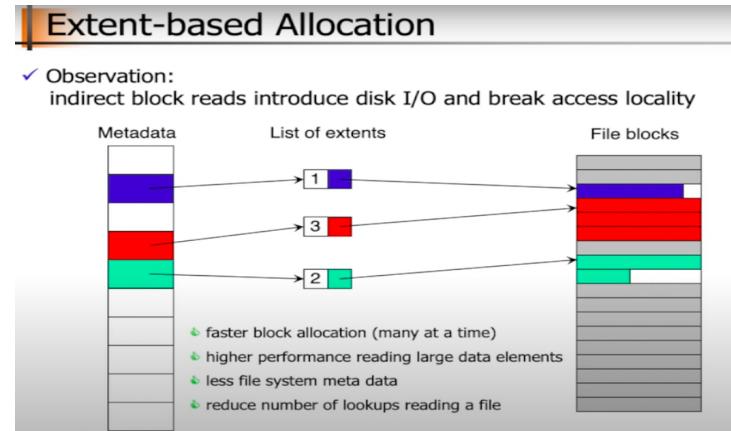
Unix/Linux Example: FFS, UFS, ...



Problemet ved denne løsningen kan forklares gjennom eksempelet over. Si vi har en videofil på mange KB. Da er det mange blokker som skal til for å kunne dekke hele videofilen. Løsningen blir da å ha såkalte "indireksjoner" hvor vi har indeks som peker på nye diskblokker igjen. Siden vi har at en blokkstørrelse er satt til 4kb, så har vi da at indeksen kan peke på 1024-datablokker. Desto flere indirekts, desto flere filer av mange bytes, kan vi da peke på. Så ved "triple indirect" så kan vi da lagre filene som har utrolig mange

datablokker av stor størrelse. Men problemet er at vi må hente diskblokker som gir oss nye pekerne som peker til de nye blokkene igjen, så innebærer dette flere I/O operasjoner som burde unngås. Dermed blir en løsning “Extent-based allocation”.

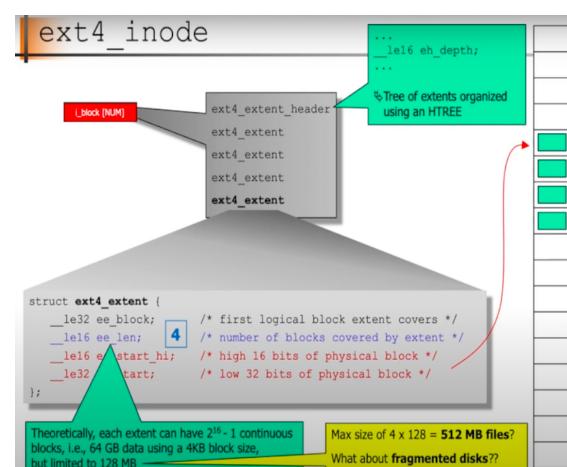
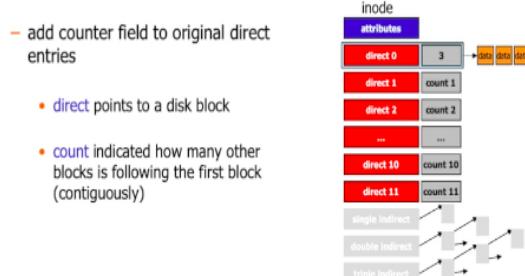
Extent-based Allocation



For at denne metoden skal fungere, må filblokkene være satt opp slik at de er sekvensielt etter hverandre. Ved bruk av “extents”, kan vi da løse problemene til metodene over. Extents er et element som består av både en teller og en peker som peker til det første data-blokk til en gitt fil. F.eks så har vi at den røde-extenten peker på de røde-datablokkene hvor disse tilsammen tilsvarer 3. Vi trenger bare en “entry” for å adressere til en gitt datablokk i motsetning til forrige metode. Fordelene er som vist i bildet, hvor vi har raskere blokk-allokering i og med at vi kan allokere mange blokker etter hverandre fremfor i tilfeldige steder. Blir på en måte som en stack hvor du fyller inn blokkene avhengig av hvilken fil de tilhører. En annen fordel er knyttet til større-datablokker hvor om man skal lese slike blokker, så får man færre oppslag som er ganske bra. Vi for også færre metadata å forholde oss til som fører til å redusere oppslag for å lese en fil sammensatt av mange datablokker.

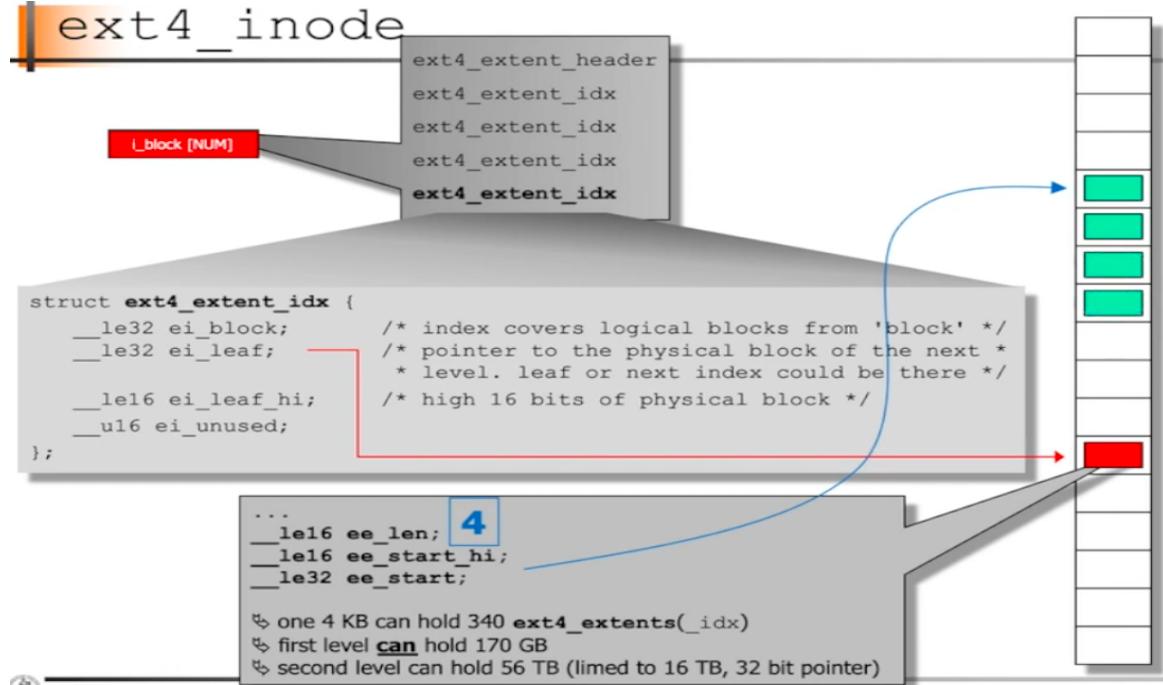
Linux Example: XFS, JFS, EXT4...

- Count-augmented address indexing in the extent sections
- Introduce a new inode structure

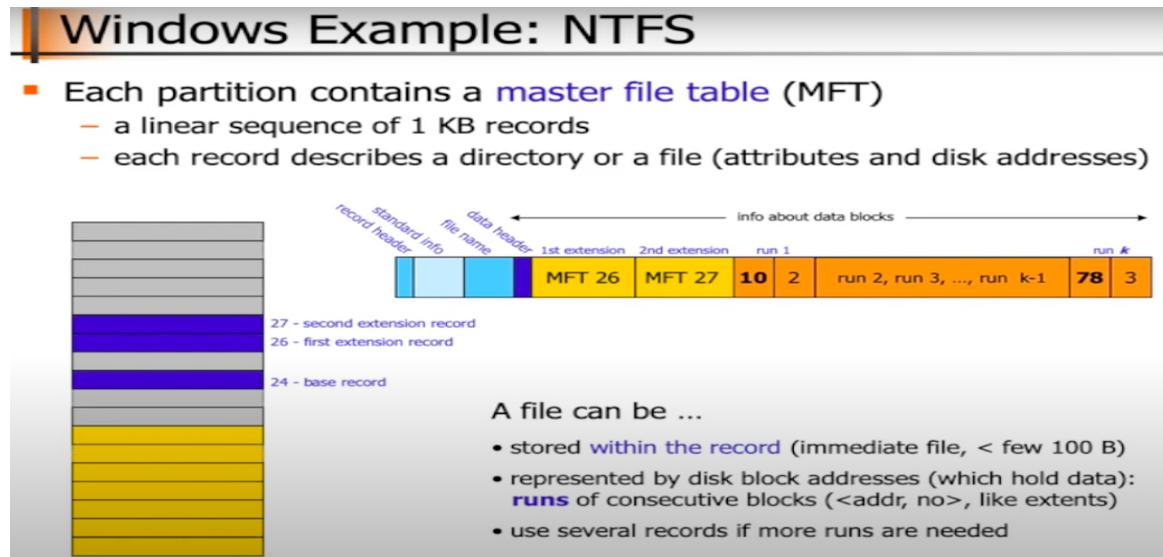


Dette er et eksempel som Pål viste hvor vi da bruker den metoden over. Som vi ser i bildet til venstre, så har vi da en nodestruktur som fungerer da slik som metoden over. I bildet til

høyre så har vi da en array av blokker hvor hver blokk består av disse “extentene”. Hver “ekstent” er beskrevet med den structen nederst i bildet til høyre, hvor du har en lengde som beskriver antall blokker lagret etter hverandre som er dekket av den gitte ekstensen. Men det er begrenset hvor mye en størrelse av en gitt datablokk som en gitt ekstent kan peke på som beskrevet over. Tilsvarer da 512 mb files som ikke er så mye. Istedetfor å ha ekstens her, så kan vi benytte av en tre-struktur hvor treeet har da en løvnode som vi skal se i bildet under.



Treenoden vil da bestå av extents, som peker til ulike nivåer hvor det første nivået av en extent, kan holde på 170 gb, andre level kan ha 56 terabyte osv.



Fungerer slik at vi har en directory-record som består av “record header”, “standard info”, “file name”, “data header” også de neste punktene som involverer info om datablokkene. Hver record beskriver en direktori eller en fil. Disse recordsene har da “runs” som er det

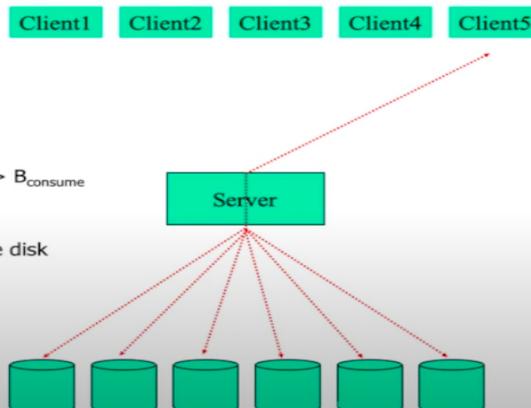
samme som extents. Extensions får vi av at, vi kan ikke ha alle “runs” i en gitt record. Derfor har vi extension som da peker på nye records med andre runs.

Multiple Disks

Striping

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
- Drawbacks
 - can't serve multiple clients in parallel
 - positioning time increases (i.e., reduced efficiency)



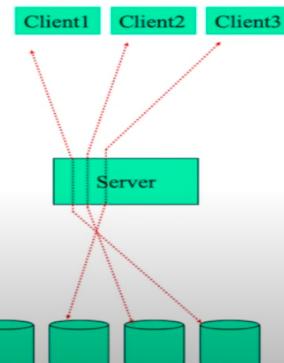
En grunn til å bruke flere disker, kan skyldes av at man muligen benytter av en disk som ikke greier å levere dataen i den “transfer-raten” som applikasjon behøver at den skal leveres på. Feks at vi har en avspillingsvideo og vi ønsker at dataene skal leveres slik at opplevelsen for brukeren blir god for den videoen, ikke at den skal “lagge” siden raten dataen blir overført er treg. I en slik senario så kom man med ideen “striping” som går ut på å fordele datablokker utover forskjellige disker og lese disse i parallell. Et eksempel er da tegningen til høyre for bildet som beskriver hvordan det funker. Dette gjør at man får en høyere transfer-rate sammenlignet med en disk. Utfordringen med dette er at en enkel prosess okkupere alle diskene samtidig. Altså hvis vi har at en klient(jeg) ønsker å lese mine videofiler, så vil jeg okkupere alle diskene. Ingen andre vil få muligheten til å utnyttet dem siden jeg allerede utnytter dem. En annen utfordring er at man kan få redusert effektivitet siden det kan hende at noen av diskblokkene er tregere enn andre, og da må man vente til de trege diskblokkene også er blitt ferdig lest.

Interleaving

Interleaving (Compound Striping)

- Full striping usually not necessary today:
 - faster disks
 - better compression algorithms

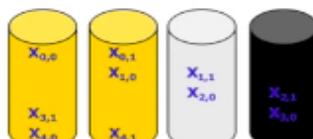
- Interleaving lets each client be serviced by only a set of the available disks
 - make groups
 - "stripe" data in a way such that a consecutive request arrive at next group
 - one disk group example:



Ideen går ut på at man ikke har en klient som nødvendigvis leser alle diskblokkene samtidig som i striping, hvor man lager rom for at andre klienter vil få muligheten til å utføre et sett av ledige disker i en sekvens. F.eks så kan "Client1" lese diskene fra venstre til høyre, men kun en av gangen. I mellomtiden kan en annen klient som "Client2" utføre andre diskblokker i parallel med "Client1" hvor "Client2" kan lese diskene fra høyre til venstre.

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 disks operations
 - potentially shorter response time
- Potential drawback/challenge
 - load balancing (all clients access same group)

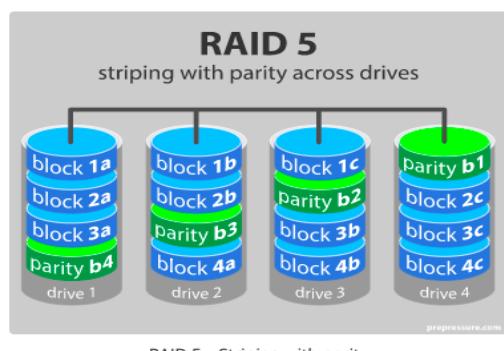
Dette er en blanding av både "Striping" og "Interleaving" hvor vi nå får muligheten for at flere klienter kan utnytte diskene i parallel. Man får potensielt kortere responstid og får gjort mer effektive disk-operasjoner. Utfordring er at alle klienter kan aksessere samme gruppe, hvor man kan få overbelastet den gruppen imens andre grupper ikke er benyttet.

Redundant Array of Inexpensive Disks

- ### Redundant Array of Inexpensive Disks
- The various **RAID levels** define different disk organizations to achieve higher performance and more reliability
 - RAID 0 - striped disk array without fault tolerance (non-redundant)
 - RAID 1 - mirroring
 - RAID 2 - memory-style error correcting code (Hamming Code ECC)
 - RAID 3 - bit-interleaved parity
 - RAID 4 - block-interleaved parity
 - RAID 5 - block-interleaved distributed-parity
 - RAID 6 - independent data disks with two independent distributed parity schemes (P+Q redundancy)
 - RAID 10 - striped disk array (RAID level 0) whose segments are mirrored (RAID level 1)
 - RAID 0+1 - mirrored array (RAID level 1) whose segments are RAID 0 arrays
 - RAID 03 - striped (RAID level 0) array whose segments are RAID level 3 arrays
 - RAID 50 - striped (RAID level 0) array whose segments are RAID level 5 arrays
 - RAID 53, 51, ...

Dette er den mest kjente metoden for datalagring som blir benyttet idag. Ideen går ut på at man bruker en serie av enkeltstående harddisker som et samordnet datalager.

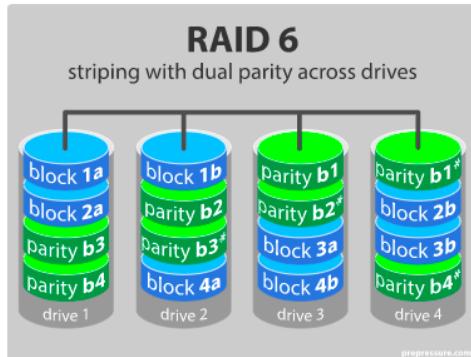
- Raid 0 går ut på at man “striper” utover alle diskene uten redundans. Dvs at man fokuserer kun på diskutnyttelse og ikke sikkerheten.
- Raid 1 går ut på at man lager to eksakte kopier slik at man får sikkerhet og potensielt høyere ytelse siden man kan lese fra to disker samtidig. Sikkerhet vil si i form av det å forsikre at dataene ikke blir mistet hvor man ivaretar sikkerheten ved denne type Raid.
- Raid 2, 3 blir ikke benyttet siden disse opererer på Bit-nivå og ikke blokk-nivå.
- Raid 4 går ut på at man har “n-1” disker som er aktive og brukes for leseforespørslar mens den siste disken blir benyttet for “paritet”. Dvs, om en av diskene blir skadd og dataene er blitt påvirket, så bruker man “paritets-disken” til å gjenopprette og fikse disken som ble skadd. Det kan også være for å oppdatere diskene.



RAID 5 – Striping with parity

- Raid 5 er samme som “4”, men hvor “paritets-disken” blir fordelt mellom alle diskene slik at, vi kan da oppdatere de ulike diskene ettersom man har fordelt pariteten mellom alle. Skulle en feil oppstå i en disk, så vil ikke det utgjøre noen fare siden

pariteten kan rekalkulere data som ikke lenger er tilgjengelig. Men dette vil da alle diskene få oppnådd.



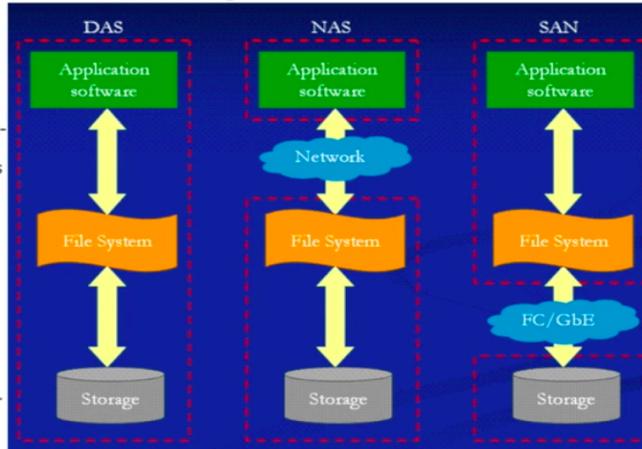
RAID 6 – Striping with double parity

- Raid 6 har samme ide som Raid 5, men nå fordeler vi en ekstra paritets-disk for hver disk. Forskjellen er om to diskene feiler, så har vi fremdeles tilgang til alle dataene. Denne er altså mer sikrere enn Raid 5 på grunn av dobbel-paritet.

DAS vs. NAS vs. SAN??

- How will the introduction of **network attached disks** influence storage?

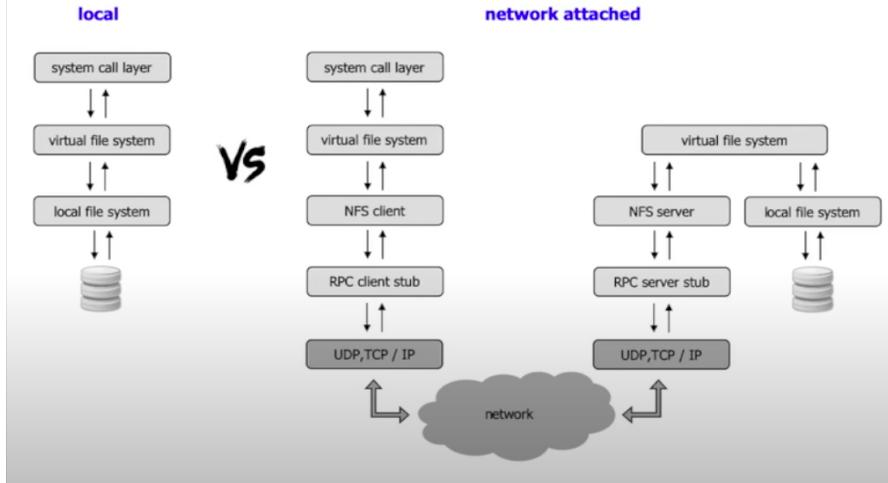
- Direct attached storage
- Network attached storage
 - uses some kind of file-based protocol to attach remote devices non-transparently
 - NFS, SMB, CIFS
- Storage area network
 - transparently attach remote storage devices
 - iSCSI (SCSI over TCP/IP), iFCP (SCSI over Fibre Channel), HyperSCSI (SCSI over Ethernet), ATA over Ethernet



Det vi har sett til nå når det kommer til lagring er da type "DAS" hvor lagringen foretas kun gjennom din egen datamaskin og dets minne. Men hva hvis man har nettverkslagring, hvordan vil det påvirke alt vi har sett til nå. Altså dette med søkeraten, overførselsraten, effektiviteten når en bruker f.eks skal utføre en lese-forespørsel gjennom nettverk? Dette er bare noe Pål visste til for å få oss til å undre over problemer som kan oppstå her og skal bli besvart når Carsten tar over.

Example: Network File System (NFS)

- Distributed file system – allowing a client to access a file over a network



Dette er problemene vi står ovenfor nå. Vi må forholde oss til TCP/IP-laget, og overføre dataene blant to mange rutere og switchere mellom klient og server. Dette skal Carsten snakke mer om, men ved å se på bildene så ser vi hvor stor forskjell det er mellom "lokal" og "nettverk".

Mechanical Disks vs. Solid State Disks???

- How will the introduction of **SSDs** influence storage?



	Storage capacity (GB)	Average (seek) time / latency (ms)	Sustained transfer rate (MBps)	Interface (Gbps)
Seagate Cheetah X15.6 (3.5 inch)	450	3.4 (track to track 0.2)	110 - 171	SAS (3) FC (4)
Seagate Savvio 15K (2.5 inch)	73	2.9 (track to track 0.2)	29 - 112	SAS (3)
Sagate Barracuda Pro	14000	8.5 (track to track 1.0)	250	SATA (6)
Intel X25-E (extreme)	64	0.075	250	SATA (3)
Intel Drive 910 Series	800	< 0.065	2000	SAS (6)
Intel DC S3700 Series	2000	0.020	2000	PCIe, v3
Intel DC P3608	4000	0.020	5000	PCIe, v3

Dette er ment for å stille problemstillingen på om man burde foretrekke SSD fremfor harddisk. Vi ser da forskjellen over hvor man kan få mindre søketid og overførselsrate i SSD sammenlignet med harddiskene. Men SSD'er er noe dyrere enn harddisk som er viktig å ta hensyn til. I tillegg til at lagringsmengden er større i harddisk enn ved SSD hvor man kan få en standard HDD med "12 Terabyte" for 5 norske kroner mot en SSD som kun har "2 TB" med samme pris.

