

Modern Disk Scheduling

- Disk used to be simple devices and disk scheduling used to be performed by OS (file system or device driver) only...
- ... but, new disks are more complex
 - hide their true layout, e.g.,
 - only logical block numbers
 - different number of surfaces, cylinders, sectors, etc.

1:

OS view **real view**

Modern Disk Scheduling

- Disk used to be simple devices and disk scheduling used to be performed by OS (file system or device driver) only...
- ... but, new disks are more complex
 - hide their true layout
 - transparently move blocks to spare cylinders
 - e.g., due to bad disk blocks

2:

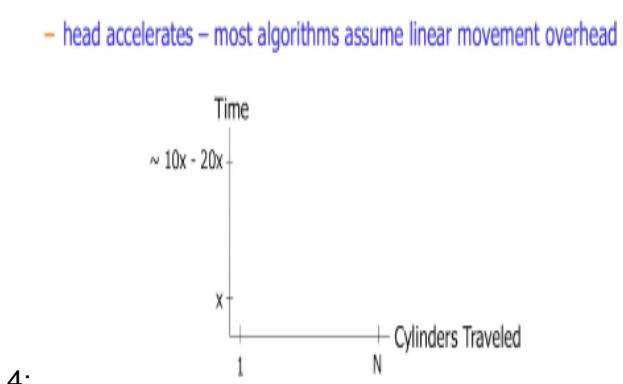
Modern Disk Scheduling

- Disk used to be simple devices and disk scheduling used to be performed by OS (file system or device driver) only...
- ... but, new disks are more complex
 - hide their true layout
 - transparently move blocks to spare cylinders
 - have different zones

3:

OS view **real view**

Zone	Cylinders per Zone	Sectors per Track	Zone Transfer Rate (MB/s)	Sectors per Zone	Efficiency	On-Disk Cache (MB)
1	31,544	672	170,040,000	2,048,000	76,0%	9013,248
2	13862	652	878,43	17604000	76,0%	
3	3079	624	635,76	15304000	76,5%	7984,293
4	2579	592	576,00	14004000	76,5%	7460,000
5	2798	576	575,00	12897792	75,5%	7145,000
6	2676	512	728,43	11474612	75,5%	5875,000
7	2354	512	687,00	10440704	75,5%	5345,641
8	2437	488	649,41	9338864	75,5%	4781,356
9	2315	466	632,41	8049696	75,5%	4426,266
10	2543	456	604,29	8188864	75,5%	4197,690



3:

- on device buffer caches may use read-ahead prefetching

5:

OS view **real view**

6:

- on device buffer caches may use read-ahead prefetching

OS view **real view**

Bildene over beskriver ulike faktorer til hvorfor moderne disker er mer komplekse. Dette påvirker utviklere til å bygge en "moderne" skeduler siden man ikke vet hvordan OS skal ta hensyn til disse faktorene som man bygger skeduleren. Under er det vist beskrivelser for de ulike faktorene:

- **De gjemmer den ekte "layouten"**
 - Dvs, at i forrige forelesning så observerte vi ulike disker som hadde 512 bytes og x antall plater osv, men dette er ikke nødvendigvis noe OS får vite. Det er derfor mulig at man implementerer et ukjent interface. I bildet over så ser vi da to ulike layouts som da beskriver dette tilfellet hvor OS får en annen layout enn hva som er beskrevet i virkeligheten
- **Bevege datablokker for å spare cylindere**
 - Si vi har et tilfelle hvor datamaskinen ble utsatt for vann og noen av datablokkene ble rammet av dette. Da vil OS prøve å gjenopprette disse dataene og bevege dem til andre ledige datablokker. Så selvom datamaskinen din har 512 byte av datablokker, så er det ikke slik at alt sammen blir benyttet for "bruk". Noe blir spart opp når tilfeller som nevnt over.
- **Ha forskjellige soner**

- Noen disker kan ha forskjellige soner som beskriver distansen av spor. Altså vi kan ha lange spor, eller korte spor hvor bildet "3" så kan vi se at "Real view" og "OS view" har forskjellige soner. Betydningen til dette, er at vi kan ha forskjellige mengder data hvor noen soner kan ha mer data og andre vil da ha færre. Dette gjør at "Zone-transfer-rate" i "Real view" vil være raskere enn "OS-view" siden sonene er blitt delt opp i flere "biter", i dette tilfelle var det dobbelt så stort
- **Disk-hode aksellerasjon**
 - De fleste algoritmer baserer en linear bevegelse "overhead" når det kommer til disk-hode akselleraasjonen, hvor den disk-hode vil gradvis bli linær når det kommer til aksellerasjon i form av hvor mange cylindere som er blitt traversert igjennom. Grunnen er at man ikke kan vite på forhånd, hvor lenge diskhode vil bruke på å lese de ulike cylinderne så da har man satt en tilnærming til "linear".
- **Cache**
 - Når en diskhode skal finne sporet som skal leses, så må den gå igjennom ulike spor. Når den har nådd sporet, så må den lese sporet og rotasjonen som sporet befinner seg i. Når dette er blitt skjedd, så vil den kopien av rotasjonen, altså dataene som er lagret i den rotasjonen, bli sendt til Cache hvor vi kan benytte cache til å hente disse sporene skulle det bli forespurgt.
 - Tatt alle disse ulike faktorene nevnt over, så blir det vanskelig for oss å designe en moderne "Disk-schedulers" siden det er mye vi ikke har kontroll over som vi ser gjennom faktorene over. Under skal vi se moderne skedulere som diverse OS bruker idag, spesielt "Linux" og "Windows".

I Schedulers today (Linux)?

- Elevator – SCAN
- 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

```
$> more /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

I bildet over, så er dette oversikt over de ulike skedulerne som blir brukt i Linux. Det er noen som er blitt forklart i tidligere forelesninger når det kommer til algoritmer som blir brukt, men skedulerne har noen modifiseringer med disse algoritmene.

NOOP

- Benytter av "FCFS, First-Come-First-Serve", men har inkludert "request merging". Det fungerer slik at om vi har to forespørsler av datablokker som skal aksessere disk, og disse ligger etter hverandre, så skal disse slås sammen til en forespørsel for å spare søketid.

Deadline I/O

- Benytter av "C-scan" men har også modifiseringer. Denne type skedulering er hva som blir brukt på IFI-maskinene ved UIO. Vi har 3 køer hvor 2 av køene benyttes for "lesing" og "skriving". Disse 2 køene har "deadlines" som vil si at man har en tidsbegrensning for hvor lenge en "leseforespørsel" eller en "skriveforespørsel" skal vare. Og hvis vi har at "deadline" for "leseforespørsel" er satt til 1,5 sekunder, men denne "leseforespørselen" varer mer enn dette, så vil denne bli prioritert.

Anticipatory

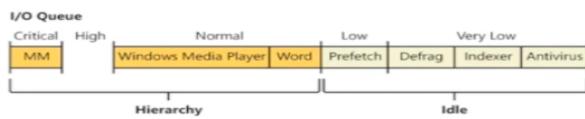
- Fungerer akkurat som ved "Deadline I/O", men denne gangen så inkluderer man også tankegangen ved "NOOP" når det kommer til merging. Dvs, om man finner en "leseforespørsel" som varer mer enn et 1,5 sekund for den en fil, så leter man etter en annen "leseforespørsel" som varer mer enn ett 1,5 sekund for den samme filen. Da kan man slå sammen disse forespørslene for å spare tid.

Completely Fair Queuing (CFQ)

- Anticipatory var ganske bra når det kom til håndtering av forespørsler som nevnt over gjennom "merge", men fungerte ikke bra for database-forespørsler. Derfor lagde man denne for å forbedre "Anticipatory". Denne fungerer slik at man tar litt av ideen bak "CPU-scheduling" i forbindelse med "time-slots", hvor hver kø vil få en "time-slot". Tidsrammen vil bli gitt avhengig av antall forespørsler som finnes hver kø. I tillegg så inkluderer man prioritet i den forstand at, de forespørslene som har høye prioriteter, vil få lengre tid til å benytte av CPU'en. I tillegg er den "Work-conserving" som vil si at, om vi har en prosess som ikke ble ferdig med sin prosess gjennom CPU'en så skal man igangsette neste prosess med engang. Dette er for å hindre at tid blir bortkasta.

Schedulers today (Windows)?

- Well, a bit hard to say, but...
- I/O priorities
 - individual I/O operations
 - FIFO within each queue
 - to avoid low-priority starvation, a timer enforces ONE I/O per time unit (.5 sec)
 - reservations
 - many special functions: fast I/O, I/O boosts and bumps, ...



Slik fungerer "Windows" sin skeduler hvor man har en "Hierarchy-gruppe" og en "Idle-gruppe". I "Hierarchy-gruppe", så utføres de prosessene som er nært bundet til OS'et, altså de som er rangert høyest. I "Idle-gruppen" så utføres de prosessene som er rangert lavest hvor de over, er prosesser som rydder opp i ting slik Pål beskrev det. Poenget er at du har en FIFO i hver kø, men får å unngå at vi får "starvation" i "Idle-gruppen" siden vi utfører prosessene i "Hierarchy-gruppen", så inkluderer man en prosess fra "Idle-gruppen" til enhver tid som kan være et halvt sekund.

Cooperative user-kernel space scheduling

- Some times the kernel does not have enough information to make an efficient schedule
- ↳ File tree traversals
 - processing one file after another
 - tar, zip, ...
 - recursive copy (cp -r)
 - search (find)
 - ...
- Only application knows access pattern
 - use ioctl FIEMAP (FIBMAP) to retrieve extent/block locations
 - sort in user space
 - send I/O request according to sorted list

⇒ GNU/BSD Tar vs. QTAR

Vi har sett gjennom tidligere, at OS har ulike faktorer å forholde seg til, når det kommer til å bygge en effektiv skeduler. Dette synet støttes også i bildet over hvor det står at, noen ganger kan det forekomme tilfeller hvor kjernen ikke har nok informasjon til å lage en effektiv skeduler. Det man kan gjøre, er å samarbeide mellom kjernen og applikasjonsnivået. F.eks, hvis vi har Fil-tre hvor vi må da traversere gjennom utrolig mange filer og kompile disse, så

vet ikke OS hvilken rekkefølge som er mest effektiv ettersom det finnes utrolig mange filer. Da kan utviklerne bygge opp skeduleren til å hjelpe OS med dette problemet slik at OS vet at "Jeg skal lese fil 1, komprimere denne også må vi hente fil 2 og komprimere dette". Slik at man kan da anvende "FIBMAP" som gjør det enkelt å hente disse datablokk-lokasjonene gjennom av at OS har da kontroll over hvilke rekkefølge som skal gjøres. Da kan man forhåndslagre de ulike posisjonene til filene som OS skal da håndtere gjennom applikasjonen. Ved en slik samhandling så blir det lettere for OS å skedulere filer.

Cooperative user-kernel space scheduling

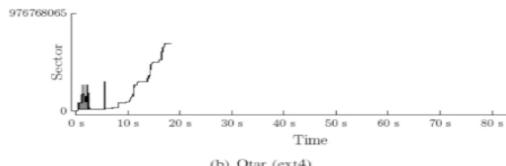
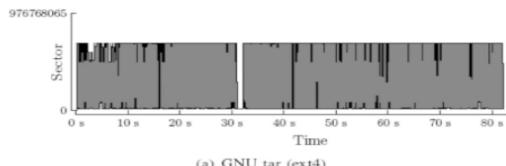
- Some times the kernel does not have enough information to make an efficient schedule

↳ File tree traversals

- processing one file after another
- tar, zip, ...
- recursive copy (`cp -r`)
- search (`find`)
- ...

▪ Only application knows access pattern

- use ioctl FIEMAP (FIBMAP) to retrieve extent/block locations
- sort in user space
- send I/O request according to sorted list



⇒ GNU/BSD Tar vs. QTAR

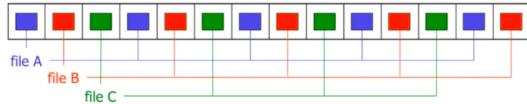
Bildet over er ment til å beskrive forskjellen av en disk hvor vi har "GNU" som ansees som en naiv skedulerer hvor vi ser at diskhodet traverserer ganske ofte får å hente sitt gitte spor. Dette skyldes av at OS'et ikke er bevisst på forhånd om hvilke filer som skal komprimeres og lager skeduleren etter hva OS mener er den mest effektive måten og hente datablokkene på. QTAR derimot, vet på forhånd hvilke filer som skal settes i rekkefølgen hvor OS'et for da vite på forhånd hvordan rekkefølgen bør settes opp av applikasjonen. Dermed ser vi her at diskhodet ikke traverserer like ofte som i "GNU" siden diskhodet vet hva den skal gjøre på forhånd. Poenget med denne forskjellen, er å vise at å utnytte ressursene ganske godt som i dette tilfellet var "applikasjonen", er ett av grunnene til at vi kan spare mye tid ved skedulering. Nå skal vi se på ulike måter diskblokkene rundt på platene ved harddisken, hvor vi tenker at diskblokkene er som filer.

Data Placement

Interleaved placement

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)

- fine for predictable workloads reading multiple files
- no gain if we have unpredictable disk accesses

- **Non-interleaved** (or even **random**) placement can be used for highly unpredictable workloads

Dette er en måte som diskblokkene blir plassert på platene i disken. I dette eksempelet så har vi da filene “A,B,C” hvor vi skal ha leseoperasjonene mellom disse filene. Man starter først med å lese en liten del av fil “A” også “B” også “C”. Dette er fint hvis vi kan forutse leseoperasjoner på diverse filer, altså at vi vet hvordan rekkefølgen skal være for leseoperasjoner mellom filene. Hvis rekkefølgen ikke er bevisst og vi begynner å endre på den, så kan det hende man ikke for noe nytte. Altså om jeg gjør om rekkefølgen fra “A,B,C” til “C,A,B”, så er det mulig at jeg ikke får den samme leseoperasjonsnytten som jeg hadde ved “A,B,C”. Da ville en “Random” eller “Non-interleaved” vært aktuell å benytte fremfor “Interleaved”.

Contiguous placement

Data Placement on Disk

- **Contiguous** placement stores disk blocks contiguously on disk



– minimal disk arm movement reading the whole file (no intra-file seeks)

- 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
- no inter-operation gain if we have unpredictable disk accesses (but still not worse than random placement)

Bildet over, beskriver hvordan denne plasserings-algoritmen funker hvor man kontinuerlig oppbevarer datablokker etter hverandre som er da knyttet til enhver fil.

Fordelene ved dette kan være følgende

- Når man skal lese en fil, så blir det lite diskhode-bevegelse ettersom diskblokkene knyttet til filen, ligger nærmere hverandre.
- Kan oppnå teoretisk transfer-rate, vil anta at dette vil si at, ettersom vi har blokkene sekvensielt etter hverandre for en fil, så vil vi da overføre datablokkene ferdig for den filen. Det er altså ikke som "Interleaved-placement" hvor overførselsesraten for en gitt fil, vil være lang siden datablokkene er plassert forskjellig.

Ulempene ved dette

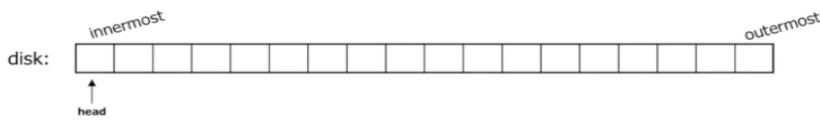
- Ulempen kan være at vi pleier å lese andre filer også, og hvis vi har mange filer, så kan det føre til at søketiden blir lengre for filer som er langt i køen. Forestill at fil "A" er ferdig lest, så skal vi lese fil "B", som er et stykke unna "A". Da kan det bli lang søketid.

Men likevel så er denne "data-placementen" en god ettersom vi utfører en gitt leseforespørsel i en "go" siden datablokkene er plassert i nærheten av hverandre uavhengig av størrelsen til filen.

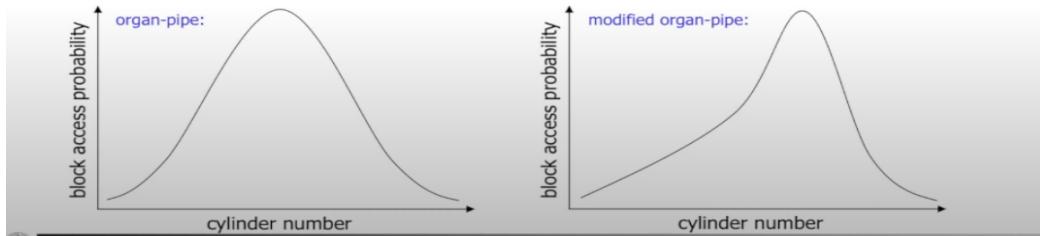
Organ-pipe

Data Placement on Disk

- **Organ-pipe** placement consider the 'average' disk head position
 - place most popular data where head is most often



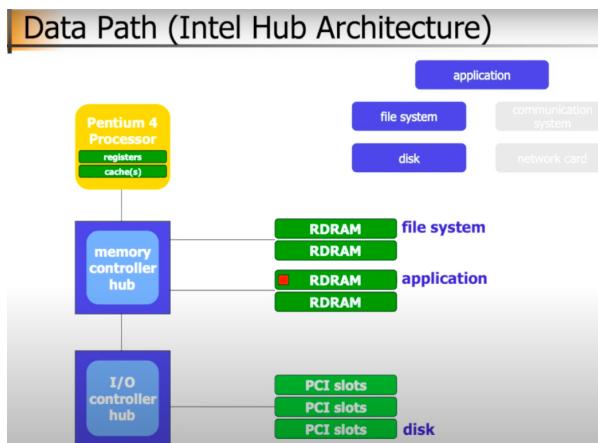
- center of the disk is in average "closest" to the head
 - but, a bit outward for **zoned** disks (**modified organ-pipe**)



For denne "Data-placement" så vil datablokkene plasseres i henhold til hvor disk-hodet er plassert gjennomsnittlig. Ved en "Organ-pipe" så vil diskhodet befinner seg i sentrum av disken altså mellom "innermost" og "outermost". Men i en modifisert "Organ-pipe" så vil vi

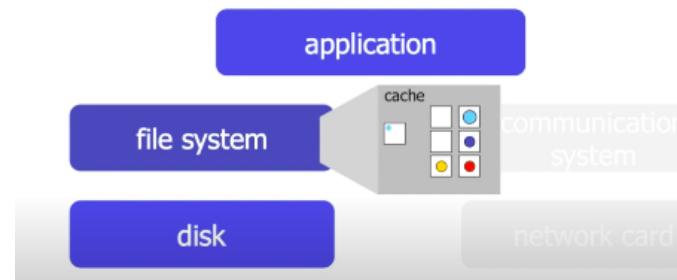
bevege oss mer mot kanten altså “outermost” siden sonene for et gitt spor, vil befinner seg i dette området. Sonene gjør det mulig å oppbevare mer data, og med mer data så vil diskhodet da traversere mer mot den siden som sonene befinner seg i. Dvs at diskhodet vil oftere traversere i “outermost” posisjon på grunn av sonene og dermed får vi at grafen blir slik.

Memory Caching



Dette er et bildet som er blitt gjentatt i forelesninger, men hensikten av å vise dette bildet, var for å repetere med at, å hente datablokker fra disk kan ta ganske lang tid som vi har sett tidligere. Dette var ment for å gi en overgang over til caching hvor Pål diskuterer om at, det er ganske kostbart å hente datablokk fra disk så hvordan kan vi gjøre dette bedre, vel løsningen er å ha et buffer som vi allokerer minne til, slik at vi unngår å måtte aksessere data gjennom disk.

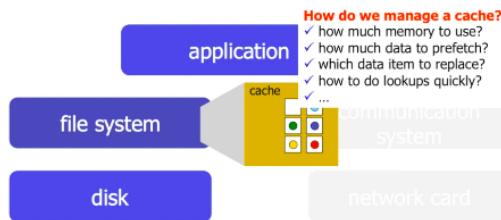
Buffer Caching



Si at applikasjonen har en forespørsel fra filsystemet som kan være en “lese” eller “skrive”-forespørsel. Filsystemet har et buffer for noen datablokker av tidligere. Hvis applikasjonen forespør om data som befinner seg i dette bufferet, så får vi raskere aksesstid sammenlignet med om den var i disk. Det var altså poenget med dette bildet og bildet over. Hvis vi har at forespørselen ikke befinner seg i bufferet, så må vi aksessere disken. Disken

vil da legge til en kopi av dataen i bufferet, slik at vi slipper å måtte aksessere diskene neste gang for den samme forespørselen siden den er i bufferet.

Buffer Caching



Nå skal vi da se på de ulike punktene som vist i bildet over når det kommer til caching.

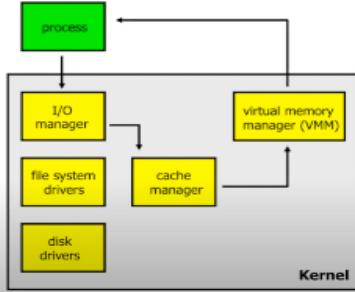
Buffer Caching: Windows (XP ++)

- An **I/O manager** performs caching
 - centralized facility to all components (not only file data)

- I/O request processing:
 1. I/O request from process
 2. I/O manager forwards to cache manager

in cache:

3. cache manager locates and copies data to process buffer via VMM
4. VMM notifies process



I Windows så har vi en I/O manager, som håndterer forespørsler fra prosessen. Deretter vil I/O manager sjekke om forespørselen er lagt inn i fil-systems-cachen, hvis så skal vi lokalisere datablokkene for den gitte forespørselen i det virtuelle minne. Det virtuelle minne vil notere seg prosessen og overføre dataen over til den. Denne prosessen beskriver bildet over. Dette var et tilfelle hvor vi hadde en forespørsel som befant seg i cachen. Men hva hvis den ikke er i cachen, da må vi aksessere disk. Tilfellet under vil da beskrive dette.

Buffer Caching: Windows (XP ++)

- An **I/O manager** performs caching
 - centralized facility to all components (not only file data)

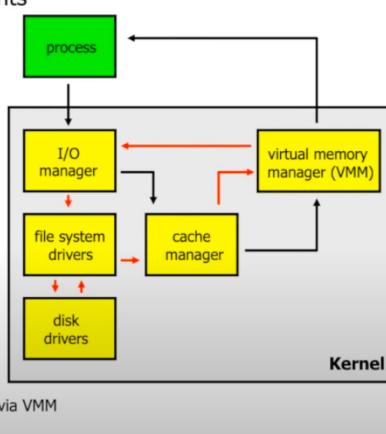
- I/O request processing:
 1. I/O request from process
 2. I/O manager forwards to cache manager

in cache:

3. cache manager locates and copies data to process buffer via VMM
4. VMM notifies process

on disk:

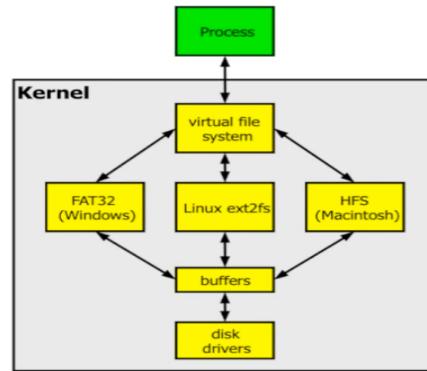
3. cache manager generates a page fault
4. VMM makes a **non-cached service request**
5. I/O manager makes request to file system
6. file system forwards to disk
7. disk finds data
8. reads into cache
9. cache manager copies data to process buffer via VMM
10. VMM notifies process



Vi tar for oss samme tilfelle som ble beskrevd i forrige avsnitt. Vi har en prosess som har en forespørsel, I/O-manager vil sjekke i fil-system-cache om dataen befinner seg der. Det gjør den ikke og det virtuelle minne mottar et flag som heter “non-cached service request” og sender dette til I/O-manager. Da blir I/O-manager notert og da må vi aksessere disken. Vi vil da gå til disken, disken finner dataen, overfører dataen mot cache, deretter vil det virtuelle minne gjøre resten av jobben som mot slutten av forrige avsnitt.

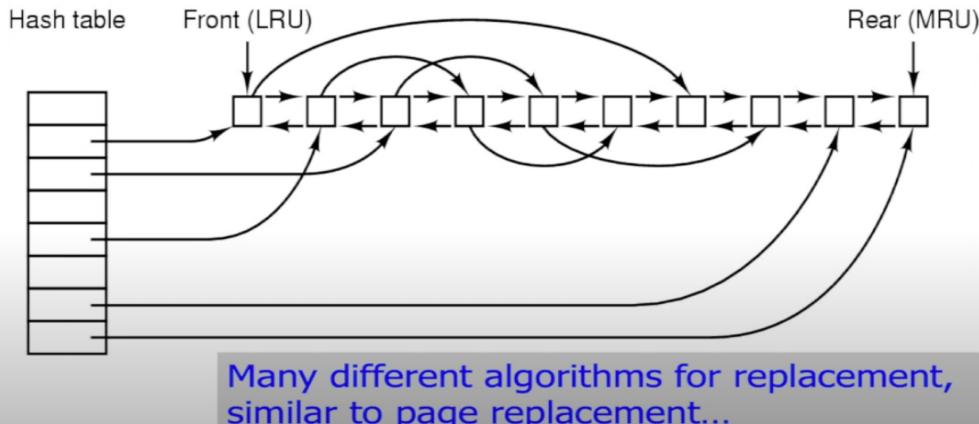
Buffer Caching: Linux / Unix

- A **file system** performs caching
 - caches disk data (blocks) only
 - may hint on caching decisions
 - prefetching
- I/O requests processing:
 1. I/O request from process
 2. virtual file system forwards to local file system
 3. local file system finds requested block number
 4. requests block from buffer cache
 5. data located...
 - ... **in cache:**
 - a. return buffer memory address
 - ... **on disk:**
 - a. make request to disk driver
 - b. data is found on disk and transferred to buffer
 - c. return buffer memory address
 6. file system copies data to process buffer
 7. process is notified



Sjekk stegene som beskriver hvordan dette fungerer.

Buffer Caching Structure



Slik ser strukturen ut ved en Buffer-Cache. Vi har alle elementene(Cache-blokkene) våre i en type “Least-recently-used” liste. De mest brukte vil da befinner seg mot slutten, imens de minst brukte befinner seg i starten. Vi har da en Hash-table som mapper et gitt oppslag til ett av disse elementene. Dette vil da fungere som hash-maps som er blitt lært tidligere hvor du mottar en form for nøkkel som mapper til den gitte elementet vi er ute etter.

Filer

- 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 (to some extent) and MVS
- In this course, we consider **unstructured files**
 - for the operating system, a file is only a sequence of bytes
 - it is up to the application/user to interpret the meaning of the bytes
 - simpler file systems

Dette er da beskrivelsen av filer hvor vi skal fokusere på ustrukturerte filer i dette emnet.

File Systems

- 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

Filsystemet er den komponenten som håndterer filene for oss brukere. Dette er delt opp i to oppgaver hvor vi har lagring(allokere plass for filer) og filhåndteringen(hvem har aksess, hvem kan lese,skrive, forsikre at integriteten ved filene er ivaretatt).

Example: open(), read() and close()

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fd, n;
    char buffer[BUFSIZE];
    char *buf = buffer;

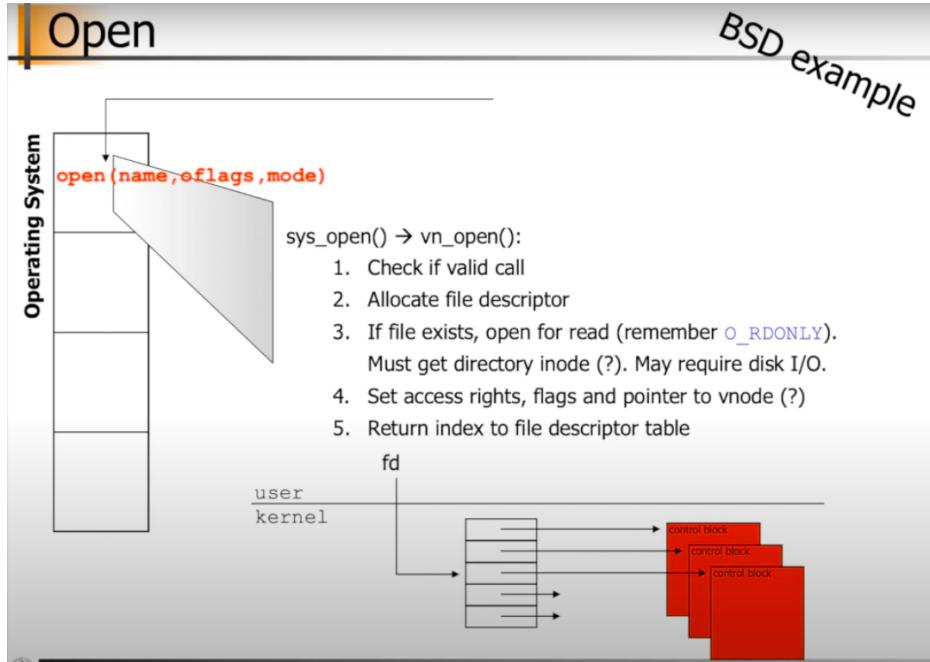
    if ((fd = open("my.file", O_RDONLY, 0)) == -1) {
        printf("Cannot open my.file!\n");
        exit(1); /* EXIT_FAILURE */
    }

    while ((n = read(fd, buf, BUFSIZE) > 0) {
        <<USE DATA IN BUFFER>>
    }

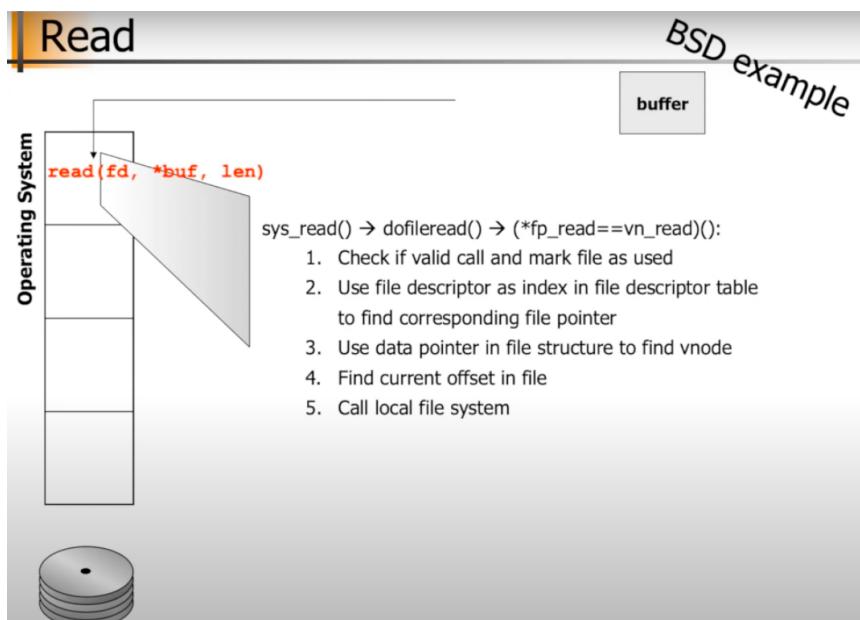
    close(fd);

    exit(0); /* EXIT_SUCCESS */
}
```

Vi skal nå se på hvordan de ulike fil-operasjonene utføres i et programmeringsspråk.

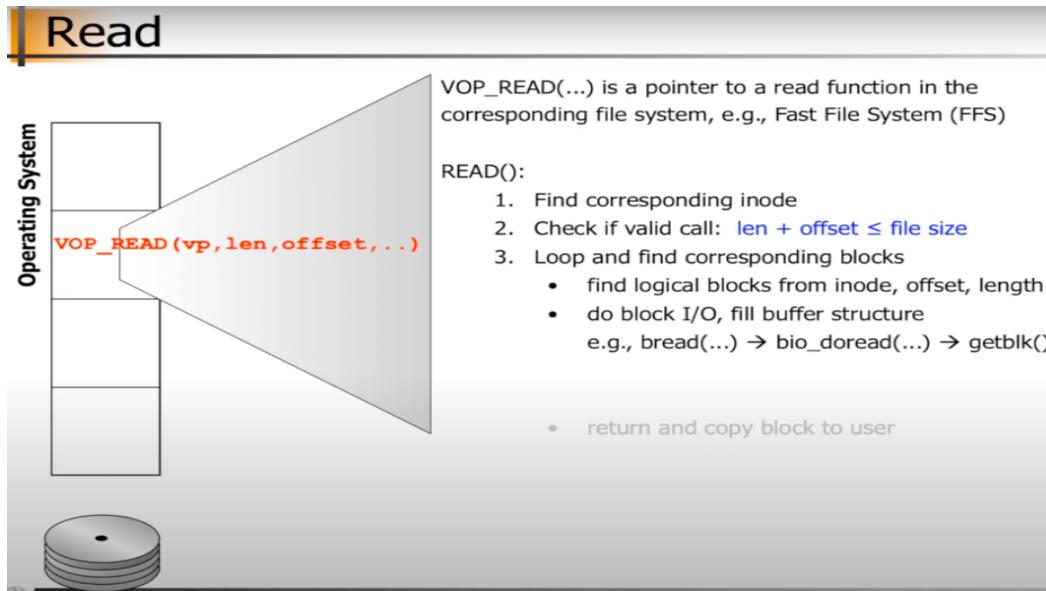


Når man skal åpne en fil, så vil sys_open() bli kalt, som er systemkall-funksjonen for å åpne en fil. Først så utføres noen sjekker før man kaller virtualnode-open(). Vi allokerer en fil-deskriptor og sjekker hvis filen eksisterer. I kjernen så har vi en tabell av fil-deskriptorer hvor hvert element i disse descriptorene, peker på en kontroll blokk som består av strøm av data av denne filen. Kontrollboksen hjelper oss med å beskrive aksessrettigheter, hvor langt vi har lest til nå, hvor lang er filen osv. Vi har en indeks som gjør det enkelt for oss hente den kontrollblokken hver gang vi skal gjøre en leseoperasjon. "fd" er da denne indeksen i bildet over som hjelper oss med å peke til kontrollblokken som vi er ute etter.

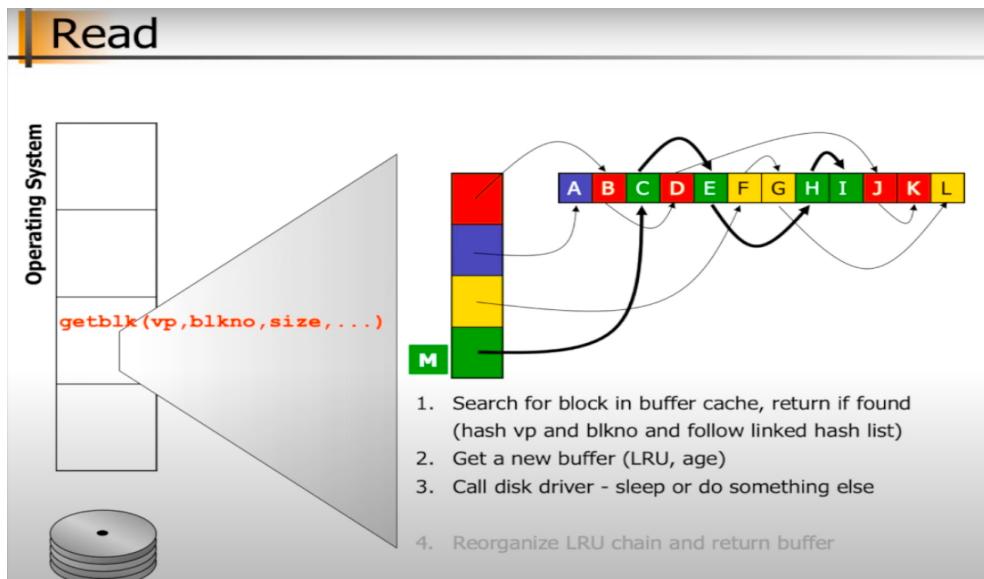


Sys_read() kalles når vi utfører read() i programmet, som vil da oversattes til en virtualnode-read() som sjekker om det er et lovlig kall og markerer at filen er i bruk. Deretter

slår vi opp i fil-deskriptoren gjennom den indeksen som ble nevnt i forrige avsnitt, hvor vi da henter kontrollblokken for den indeksen. Da får vi vite hvilke punkt i filen vi befinner oss i filen og kaller det lokale filsystemet under. Gjør da et kall på VOP_READ.

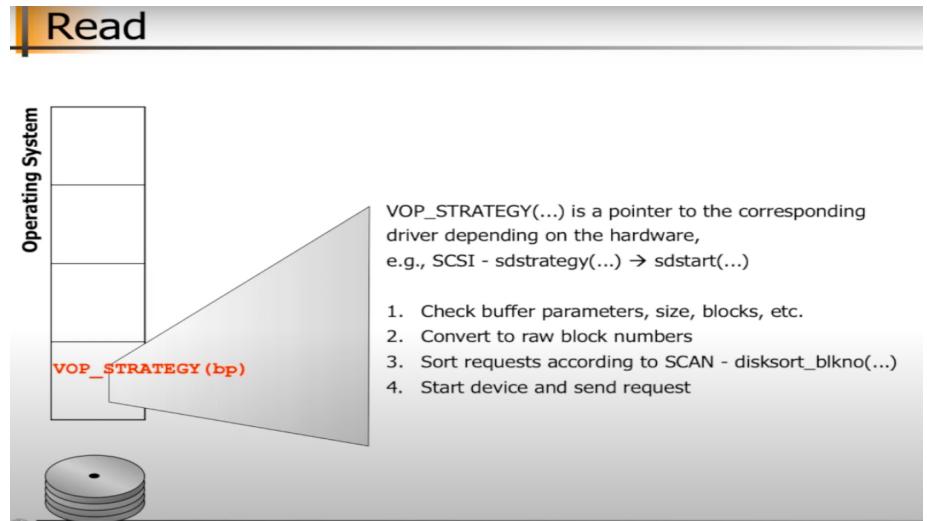


Her finner vi den faktiske metadata-beskrivelsen til filen. Først må vi forsikre at vi kan utføre denne funksjonen, hvor vi da sjekker om lengden på filen og hvor vi befinner oss nå, ikke er større enn filstørrelsen, altså at vi ikke er mot slutten av filen vi ønsker å lese inn. Basert på den operasjonen, så finner vi den logiske blokken som vi ønsker å lese inn. Vi finner altså den lengden vi ønsker å starte på som er offsettet, og lengden av antall blokker som vi skal lese inn. Vi utfører da et blokk I/O kall, som fyller inn buffer-strukturen som gjør at vi utfører da getblk().

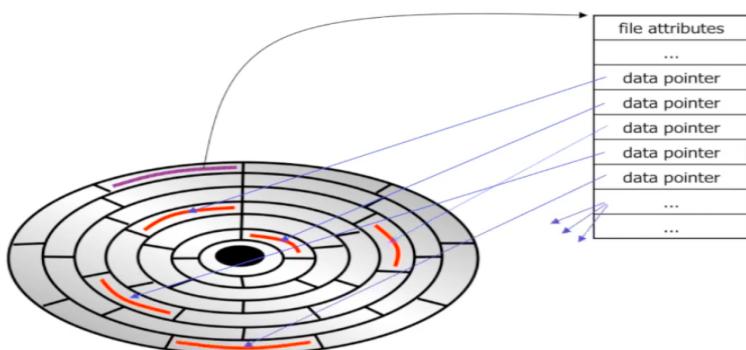


Dette er identisk til hva vi så tidligere når det kom til strukturen til buffer-cachen(LRU). Først så sjekker vi i buffer-cachen for å sjekke om hash-verdien for den blokken vi er ute etter, befinner seg her. Her ser vi at hash-verdi "M" ikke befinner seg i cachen så nå vi finne et

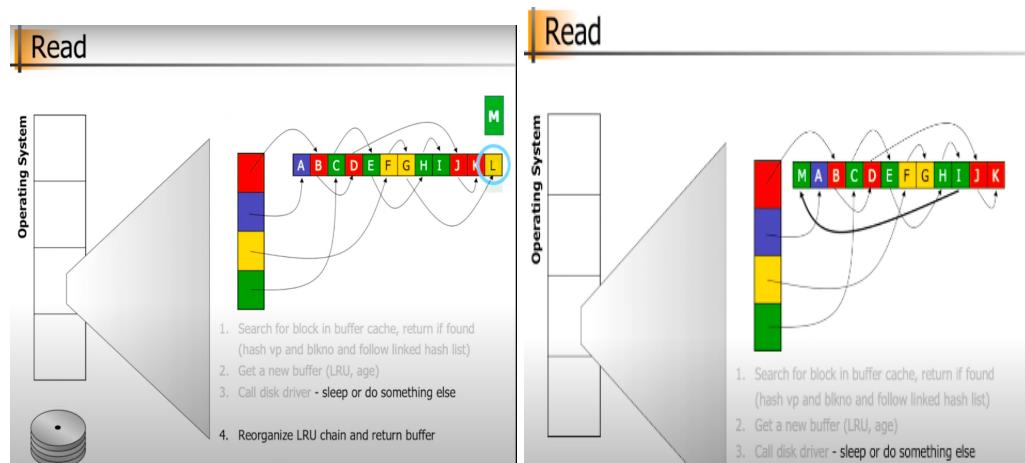
offer hvor vi da skal plukke ut offeret som skal byttes med "M". Dette blir da blokk "L" siden den er brukt for lenge siden, men først må vi lagre "M" i disken gjennom VOP_STRATEGY.



Hensikten med denne funksjonen, er at vi konverterer blokken vi sendte med(i dette tilfellet "M") og konverterer den til en fysisk adresse inn i disken.



Her vil vi da lagre blokkene til "M", i disken og returnere denne forespørselen tilbake igjen buffer-Cachen.



Tilslutt reorganiserer LRU-strukturen og setter den blokken vi ønsker å utføre som var "M" og plasserer denne helt på starten. Til syvende sist så har vi close(fd), som rydder opp og tar ut kontrollblokken fra fildeskriptor-tabellen, slik at vi rydder opp det vi har allokeret av ressurser.