TDIU11 – Föreläsning 5

Filsystem

Filip Strömbäck

LINKÖPING UNIVERSITY

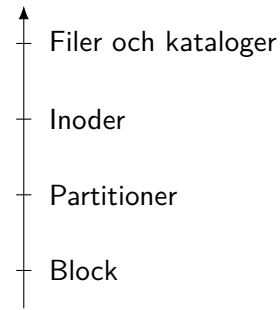
Planering

Vecka	Föreläsning	Seminarie
3	Processer och trådar	—
4	Minneshantering	Challenge 1: Schemaläggning
5	Virtuellt minne	Artikel 1: Schemaläggning
6	Filsystem och lagring	Challenge 2: Virtuellt minne
7	Säkerhet	Challenge 3: Filsystem
8	Repetition/utblickar	Artikel 2: Filsystem
9	—	Challenge 4: Säkerhet
10	Tentaförberedelse	Challenge 5: Repetition
11	(omtenta-p)	Artikel 3: Säkerhet (reserv)

LINKÖPING UNIVERSITY

Mål med föreläsningen

- Operativsystem ger oss:
  filer (*files*) och
  kataloger (*directories*)
- En disk är en array av *block* där OS kan lagra data
- Hur fungerar det?



Hur fungerar en disk?

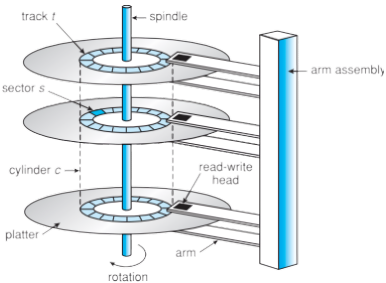
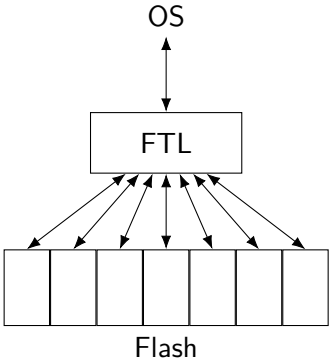


Bild från material till
Operating System Concepts

- Minsta enhet: sektor
  Historiskt: 512 B, numera ca 4 KiB
- Åtkomstid: byt spår (*track*) + vänta på sektor
- Söktid: 3–12 ms
- Vänta på sektor: $\frac{60}{\text{RPM}}$
  För 5400 RPM: 11.2 ms max, 5.5 ms medel.

Hur fungerar en SSD?



- Minsta enhet: block, ca 4 KiB
- Flash Translation Layer (FTL):
  - Wear-levelling
  - Rensar använda block
  - Emulering av mindre blockstorlek
- Åtkomsttid: svarstid hos FTL och flash-minne, kan hantera flera operationer samtidigt

Abstraktion för att hantera disk

Möjligt gränssnitt:

```
int get_block_size(int disk);

void read_block(int disk, int block, byte *data);

void write_block(int disk, int block, const byte *data);
```

Hur lagrar vi filer?

Schemaläggning av diskåtkomst

För mekaniska diskar:

- Söktid (accesstid) tar mest tid
- Läs block ”nära” varandra om möjligt!

Några alternativ:

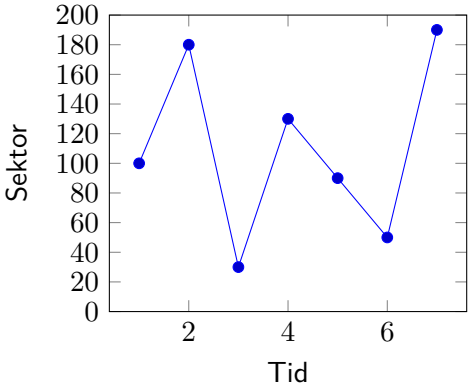
- FCFS
- SSTF
- SCAN, C-SCAN
- LOOK, C-LOOK

Exempel:
Disk med 200 sektorer
Start: 100, från 10
Kö: 180, 30, 130, 90, 50, 190

FCFS – First Come First Served

Exempel:
Disk med 200 sektorer
Start: 100, från 10
Kö: 180, 30, 130, 90, 50, 190

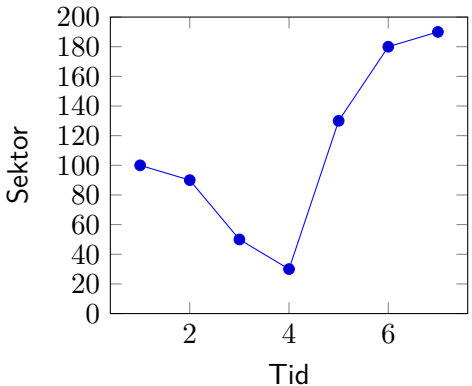
Totalt: 550



SSTF – Shortest Search Time First

Exempel:
Disk med 200 sektorer
Start: 100, från 10
Kö: 180, 30, 130, 90, 50, 190

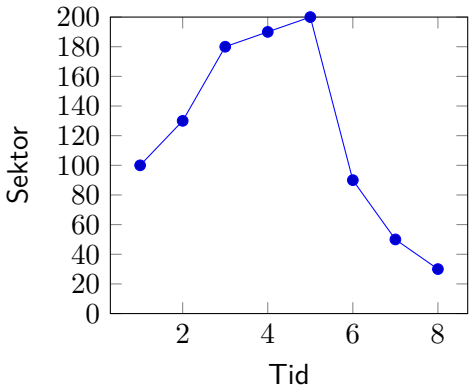
Totalt: 230
Risk för *starvation*



SCAN

Exempel:
Disk med 200 sektorer
Start: 100, från 10
Kö: 180, 30, 130, 90, 50, 190

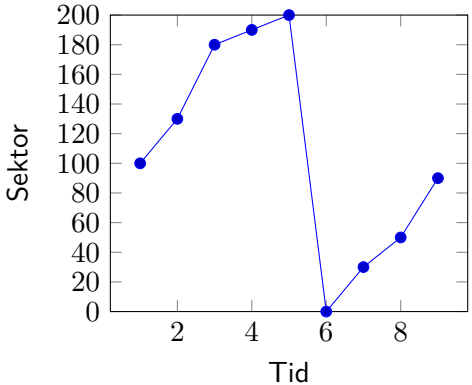
Totalt: 270



C-SCAN

Exempel:
Disk med 200 sektorer
Start: 100, från 10
Kö: 180, 30, 130, 90, 50, 190

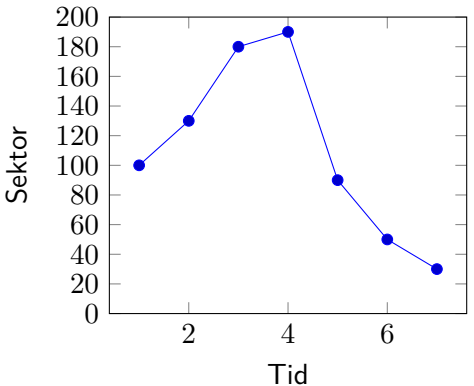
Totalt: 390
Jämnare accesstid



LOOK

Exempel:
Disk med 200 sektorer
Start: 100, från 10
Kö: 180, 30, 130, 90, 50, 190

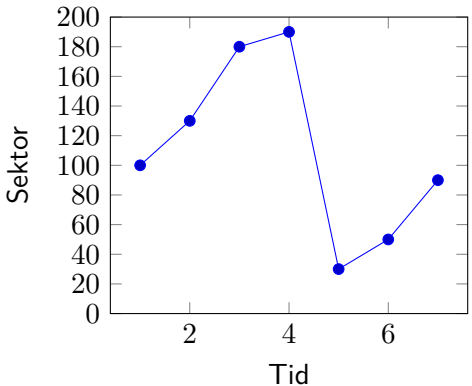
Totalt: 250



C-LOOK

Exempel:
Disk med 200 sektorer
Start: 100, från 10
Kö: 180, 30, 130, 90, 50, 190

Totalt: 310
Jämnare accesstid



Räkna total söktid – I Emacs-lisp!

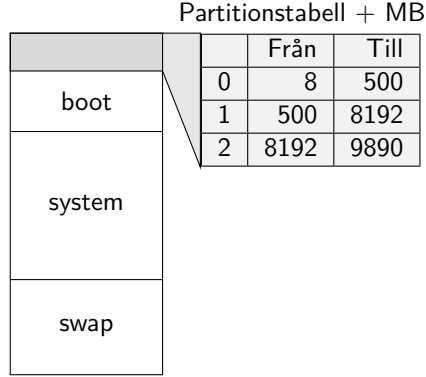
```
(defun travel (elems)
  (if (null (cdr elems))
      0
    (+ (abs (- (car elems) (car (cdr elems))))
       (travel (cdr elems)))))

(travel '(100 180 30 130 90 50 190)) ;; FCFS
(travel '(100 90 50 30 130 180 190)) ;; SSTF
(travel '(100 130 180 190 200 90 50 30)) ;; SCAN
(travel '(100 130 180 190 200 0 30 50 90)) ;; C-SCAN
(travel '(100 130 180 190 90 50 30)) ;; LOOK
(travel '(100 130 180 190 30 50 90)) ;; C-LOOK
```

LINKÖPING UNIVERSITY

Partitioner

- En disk kan delas upp i *partitioner*
- Varje *partition* är en samling block som ligger efter varandra
- Olika *partitioner* är oberoende av varandra, kan innehålla olika filsystem
- Exempelvis:
  - FAT32 för uppstartsfiler
  - ext4/NTFS för systemet
  - Partition för swap
  - Återställningspartition



Abstraktion för partitioner

Möjligt gränssnitt:

```
int get_block_size(int disk);

int get_partition_count(int disk);

void read_block(int disk, int part, int block, byte *data);

void write_block(int disk, int part, int block, const byte *data);
```

Hur lagrar vi filer?

LINKÖPING UNIVERSITY

Filsystem

- Lager ovanpå en partition
- Beskriver hur vi lagrar filer och kataloger
- Många olika med olika för- och nackdelar
  - FAT32, exFAT
  - NTFS
  - ext3, ext4, ...
  - HFS
  - ...

Blockstorlek:

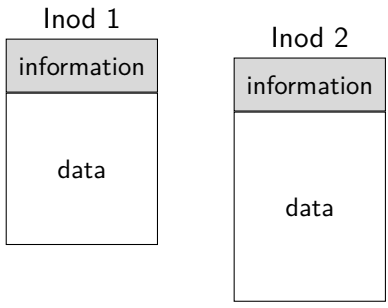
- FS arbetar med *logiska block* eller *kluster*
- Kan ha annan storlek än *fysiska block*
- Bra idé att *logiska block*/*kluster* är större än fysiska block

Vad är en inod?

Inod (UNIX) / MFT Record (NTFS)

- Sekvens av *bytes* (inte *block*)
- Godtycklig längd
- Attribut (ägare, rättigheter, ...)
- Identitet i form av ID (`ls -i`)

Tänk *fil utan namn* – vi använder tal som "namn"



Hur lagrar vi inoder? – Kontinuerlig allokering

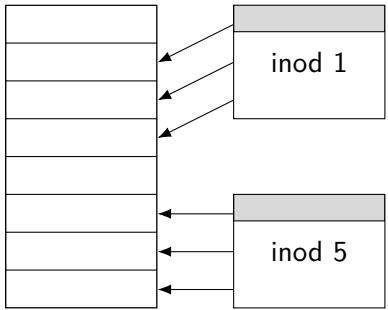
Idé: Lagra inoder sekventiellt

Fördelar:

- enkel att implementera
- sekventiell åtkomst

Nackdelar:

- fragmentering



Hur lagrar vi inoder? – Länkad allokering

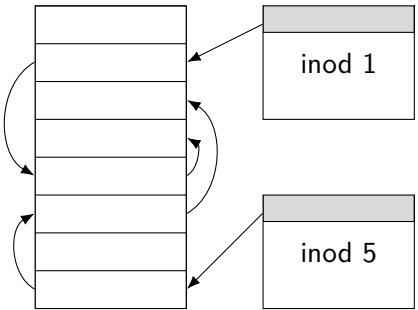
Idé: Undvik fragmentering genom att länka inoder

Fördelar:

- enkel att implementera
- ingen fragmentering

Nackdelar:

- svårt att hoppa runt i filer



Hur lagrar vi inoder? – File Allocation Table (FAT)

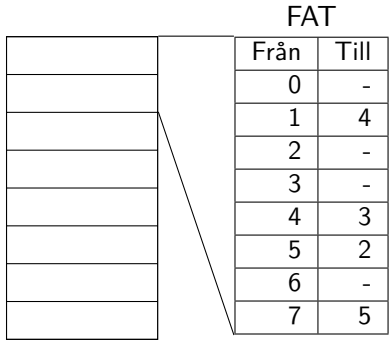
Idé: Flytta ihop länkarna!

Fördelar:

- ingen fragmentering
- enkelt att veta vad som är ledigt

Nackdelar:

- måste ha FAT i RAM
- om FAT skadas är filsystemet oläsbart



Hur lagrar vi inoder? – Indexerad allokering

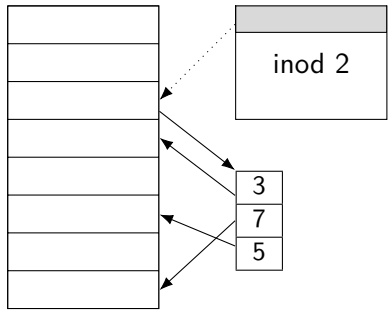
Idé: Lagra en ”pagetabell” med pekare

Fördelar:

- ingen fragmentering
- enkelt att söka

Nackdelar:

- begränsad filstorlek



Hur lagrar vi inoder? – Fler nivåer av indexering

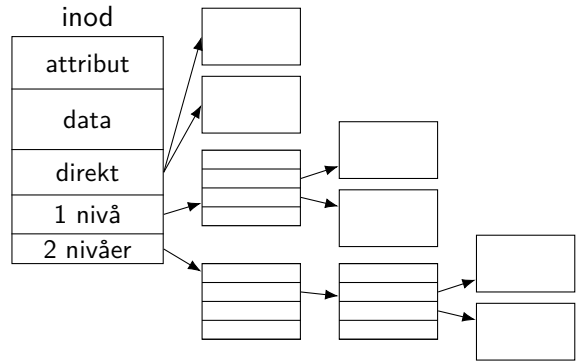
Idé: Använd multi-level "paging"!

Fördelar:

- ingen fragmentering
- ok sökning
- hanterar stora filer

Nackdel:

- komplex att implementera



Hur vet vi vad som är ledigt?

Bitmap:

- 1 bit/logiskt block
- Blir snabbt stor

Länkad:

- Länka lediga block
- Svårt att hitta sekventiella block

Länkad med räknare:

- Räkna sekventiella lediga block
- Fortfarande dyr att traversera

Indexerad:

- Lagra en ”inod” med alla lediga block
- Kan enkelt plocka indexblock till ny fil

Abstraktion för inoder

Möjligt gränssnitt:

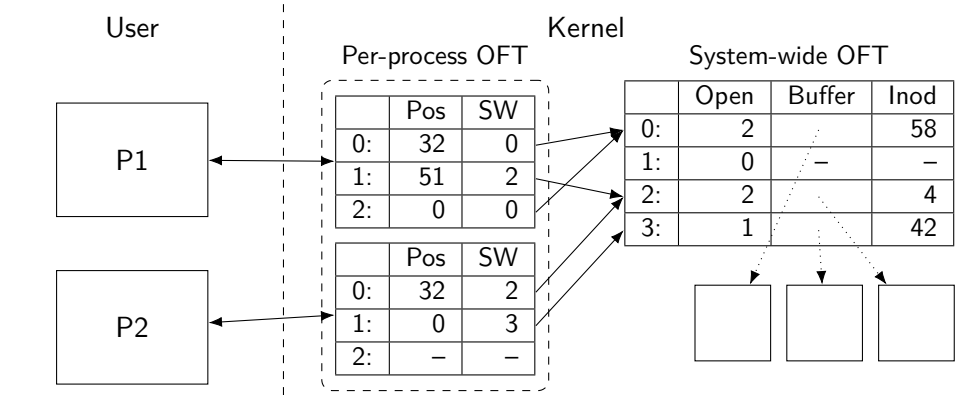
```
int inode_size(int id);

int inode_read(int id, int offset, byte *data, int size);

int inode_write(int id, int offset, const byte *data, int size);
```

Hur kan OS veta vad som borde laddas in i RAM?

Datastrukturer i kernel



Smidigare abstraktion för inoder

Idé: Vi kräver att man *öppnar* en inod först!

```
int  inode_open(int inode_id);
void inode_close(int fd);

int  inode_size(int fd);
void inode_seek(int fd, int pos);
int  inode_read(int fd, byte *data, int size);
int  inode_write(int fd, const byte *data, int size);
```

Hur namnger vi inoder?

LINKÖPING UNIVERSITY

Hur namnger vi filer?

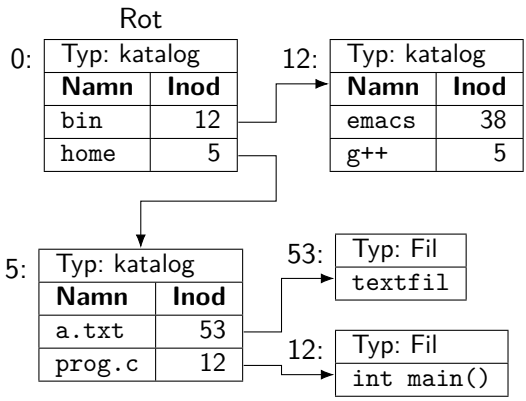
Idé: Lagra namn + id i inoder, vi kallar dem *katalog*-inoder

Tabell med:

- Namn
- Inod

Inod innehåller:

- Om katalog/fil
- Rättigheter



Länkar i systemet

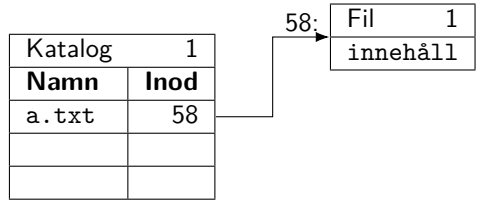
Idé: Tillåt flera referenser till samma fil

- Hårda länkar
- Symboliska länkar

I inod:

- Antal referenser

Varför inte hård länk till katalog?



Länkar i systemet

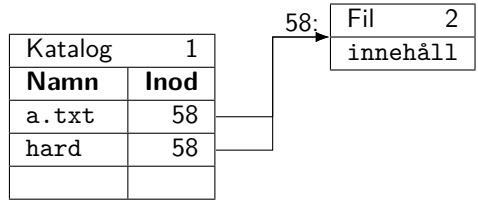
Idé: Tillåt flera referenser till samma fil

- Hårda länkar
- Symboliska länkar

I inod:

- Antal referenser

Varför inte hård länk till katalog?



Länkar i systemet

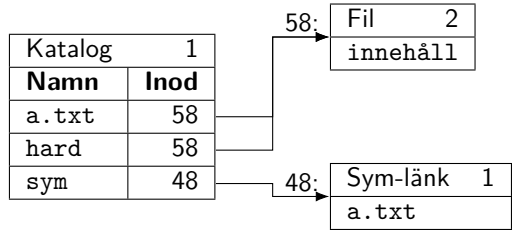
Idé: Tillåt flera referenser till samma fil

- Hårda länkar
- Symboliska länkar

I inod:

- Antal referenser

Varför inte hård länk till katalog?



Abstraktion för filer

Nu kan vi använda filnamn och sökvägar!

```
int  open(const char *name);
void close(int fd);

int  size(int fd);
void seek(int fd, int pos);
int  read(int fd, byte *data, int size);
int  write(int fd, const byte *data, int size);
```

Abstraktion för filer (forts.)

Nu kan vi använda filnamn och sökvägar!

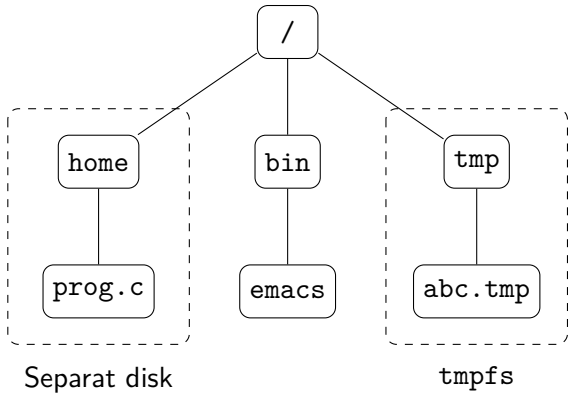
```
DIR *opendir(const char *name);
void closedir(DIR *dir);
dirent *readdir(DIR *dir);

void unlink(const char *name);
void rename(const char *old, const char *new);
```

Virtuellt filsystem

Många system har ett *virtuellt filsystem*:

- Program ser ett filsystem
- Finns egentligen olika filsystem



LINKÖPING UNIVERSITY

Nästa föreläsning

Säkerhet:

- Gör OS för att skydda sig själv mot ”elaka” program?
- Vad tillhandahåller OS för att hjälpa oss att skydda systemet?
- Hur använder vi detta för att säkra vårt system?

Filip Strömbäck

www.liu.se

LINKÖPING UNIVERSITY