2

TIEDOSTO-JÄRJESTELMÄ

ks. StRa13, luku 3 ja 4

Tämän osan sisältö

- Tiedostokuvaaja, int fd
- Tiedostonkäsittelyn systeemikutsut
- Tiedostojen käsittelyn tietorakenteita
  - tiedostokuvaajataulu, avoimet tiedostot taulu, indeksisolmutaulu
- Tiedostojen yhteiskäyttö
  - atomisuus
  - käyttöoikeudet ja niiden käsittely UNIXissa
- Tiedostolukot
- Tiedoston attribuutit = indeksisolmu
- Hakemiston käsittely
- Tiedoston lohkot
  - lohkohakemisto

TIEDOSTOKUVAAJA SYSTEEMIKUTSUT

Systeemikutsut vs. kirjastofunktiot

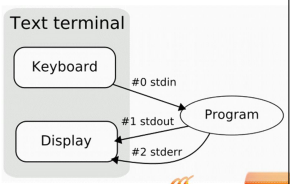
- Tiedostoja voi käsitellä kirjastofunktioiden kautta
  - `#include <stdio.h>`
  - tiedosto-osoitin `FILE *`
  - Nippu f-alkuisia kirjastofunktioita:
    fopen(), fclose(), fgetc(), fgets(), fputc(), fputs(), fprintf(), jne..

Nämä opeteltu kurssilla C/C++ -ohjelmointi

- Kirjastofunktioiden toteutus perustuu tällä kurssilla käsiteltävien systeemikutsujen käyttöön
  - `#include <fcntl.h>`
  - `#include <unistd.h>`
  - tiedostokuvaaja `int fd`
- Huom:
  - ratko kurssin tehtävät systeemikutsuja käyttäen
  - poikkeuksena lähinnä vain tulostus näytölle ja näppäilyltä lukeminen

Tiedostokuvaajat (file descriptor)

- Tiedostoon viitataan nimellä vain avattaessa, muissa systeemikutsuissa käytetään avattaessa saatua tiedostokuvaajan numeroa
  - `int fd;`
  - numero viittaa UNIXin ylläpitämään tiedostokuvaajatauluun
- Tiedosto voi tarkoittaa perinteisen tiedoston lisäksi
  - laitetta, esim. näyttö, näppäimistö (/dev/nrst0)
  - ”bittisankoa”, ”mustaa aukkoa” (/dev/null)
  - putkea tai nimettyä putkea (ns. fifot)
  - verkkoliittymää (pistokkeet)
- Kaikille samat perussysteemikutsut, luomisessa ja avaamisessa eroja
- Valmiiksi avatut tiedostokuvaajat
  - `#include <unistd.h>`
  - `0 STDIN_FILENO`
  - `1 STDOUT_FILENO`
  - `2 STDERR_FILENO`
- Seuraavaksi avattaville tdstoille numerot
  - `3 ... OPEN_MAX`



Tiedoston avaaminen

- Tiedosto on avattava ennen käyttöä, jotta
  - KJ voi tarkistaa käyttöoikeudet
  - KJ voi luoda tiedostojärjestelmän tarvitsemat kirjanpitorakenteet
- Tiedosto avataan systeemikutsulla

```
#include <sys/types.h>          käyttötapalipukkeet (flags)
#include <sys/stat.h>           käyttöoikeusbittimaskit (status)
#include <fcntl.h>
int open(const char *pathname, int oflag);
int open(const char *pathname, int oflag, mode_t mode);
```

  - Palauttaa tiedostokuvaajan numeron
- Systeemikutsut palauttavat virhetilanteissa -1
  - virheen syy `errno`-muuttujassa
  - katso mahdolliset virhenumerot ja selitykset manuaalisivulta, esim. man 2 open

Annettava, kun luodaan uusi

- Lipukkeessa (oflag) kerrottava aina käyttötapa

`O_RDONLY`	vain lukemista varten
`O_WRONLY`	vain kirjoittamista varten
`O_RDWR`	sekä lukemista että kirjoittamista varten

- Valinnaisia lisälipukkeita

`O_APPEND`	kirjoita aina tiedoston loppuun
`O_CREAT`	luo ellei jo olemassa, annettava myös luotavan tiedoston rwx-bitit (mode)
`O_EXCL`	käytetään ed. kanssa, open palauttaa virheen, jos tiedosto on jo olemassa
`O_TRUNC`	jos tiedosto olemassa ja avataan kirjoittamista varten, aseta aluksi kooksi 0
`O_NONBLOCK`	älä odota I/O -kutsun valmistumista
`O_SYNC`	odota write():ssä, kunnes tieto todella kirjoitettu levylle

- Uutta tiedostoa luotaessa pitää määritellä käyttöoikeudet (mode)

```
fd = open("ULOY/puppua.txt", O_CREAT | O_WRONLY,
          S_IRUSR | S_IWUSR | S_IRGRP);

if (fd < 0){
   perror("Eipä auennut");
...
}
```

- Oikeudet eivät kuitenkaan jää sellaisenaan voimaan, sillä myös prosessin voimassaoleva umask-arvo huomioidaan
  - Esimerkiksi, jos
    `mode = S_IRWXU | S_IRWXG | S_IRWXO` (eli 0777)
    `umask = 0007` *kertoo mitkä otetaan pois*
    niin luotavalle tiedostolle tulee oikeudet
    `-rwxrwx---`
- Uuden tiedoston omistaja (uid) ja ryhmä (gid) kopioituvat ohjelman suorittaneen prosessin kuvaajasta

<sys/stat.h> *file status*

```
#define S_IRWXU   00700 read, write, execute: owner
#define S_IRUSR   00400 read permission: owner
#define S_IWUSR   00200 write permission: owner
#define S_IXUSR   00100 execute permission: owner
#define S_IRWXG   00070 read, write, execute: group
#define S_IRGRP   00040 read permission: group
#define S_IWGRP   00020 write permission: group
#define S_IXGRP   00010 execute permission: group
#define S_IRWXO   00007 read, write, execute: other
#define S_IROTH   00004 read permission: other
#define S_IWOTH   00002 write permission: other
#define S_IXOTH   00001 execute permission: other
jne.
```


Tiedoston luominen

- Tiedoston voi luoda (ja avata kirjoittamista varten) myös systeemikutsulla

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

  - Palauttaa tiedostokuvaajan numeron
- Tämä vastaa kutsua

```
fd = open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);
```



```
int main(void) {
  int fd,fd2;
  if (fd = open("Tiedosto1", O_CREAT|O_WRONLY,
                S_IRUSR|S_IWUSR|S_IRGRP) ) < 0)
      perror("Tiedoston Tiedosto1 avaaminen ei onnistunut");
      exit(EXIT_FAILURE);
  }

  if (fd2 = creat("Tiedosto2", S_IRUSR|S_IWUSR|S_IRGRP)) < 0 )
      perror("Tiedoston Tiedosto2 luominen ei onnistunut");
      exit(EXIT_FAILURE);
  }
  close(fd);
  close(fd2);
  exit(EXIT_SUCCESS);
}
```


Tiedoston sulkeminen

- Tiedosto suljetaan systeemikutsulla

```
#include <unistd.h>
int close(int fd);
```

- Kun tiedosto suljetaan, KJ osaa vapauttaa käsittelyssä vaaditut resurssit
- KJ sulkee kaikki avoimet tiedostot, kun ohjelma päättyy
- Suositus on silti, että
  - sulje tiedosto heti, kun et sitä enää tarvitse

Tiedostosta lukeminen

- Avatusta tiedostosta luetaan merkkejä systeemikutsulla

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
```

  - Palauttaa luettujen merkkien lkm, 0 kun EOF
- Varattava itse etukäteen puskuritila luettavalle tiedolle
  - joko kiinteän pituinen merkkijonotaulukko tai dynaaminen varaus malloc()-kutsulla
- nbytes on tilavarauksen koko
- Muista
  - read() ei aina palauta arvonaan samaa arvoa kuin on pyydettyjen merkkien lkm (nbytes, eli puskurin koko)
  - tiedoston viimeinen lukupyyntö jää yleensä vajaaksi

Tiedostoon kirjoittaminen

- Avattuun tiedostoon kirjoitetaan systeemikutsulla

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t nbytes);
```

  - Palauttaa kirjoitettujen merkkien lukumäärän
- Yleensä paluuarvo on aina sama kuin kirjoitettavaksi pyydettyjen merkkien lukumäärä (nbytes)
  - Välikysymys: Milloin ei ole?

```
   #define BUFFSIZE 1024
   char buf[BUFFSIZE];

   while ((n=read(STDIN_FILENO, buf, BUFFSIZE))>0) {
       if (write(STDOUT_FILENO, buf, n) != n) {
          perror("Could not write all chars");
          exit(EXIT_FAILURE);
       }
}
```


Luku- / kirjoitusposition asettaminen

- Avattuun tiedostoon liittyy käsittelykohtaa osoittava positio
  - current file offset
  - aluksi sen arvona on 0
  - paitsi jos lipuke O_APPEND, niin arvona on tiedoston koko
- Käsittelykohtaa voi muuttaa systeemikutsulla

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

  - Palauttaa uuden position
- Uusi positio määräytyy parametrin whence perusteella seuraavasti

SEEK_SET	positio = offset
SEEK_CUR	positio = positio + offset
SEEK_END	positio = tiedoston koko + offset

- offset voi olla negatiivinen
  - tiedostoa voi siis käsitellä myös lopusta alkuun
- Nykyposition saa selville kutsulla

```
curpos = lseek(fd, 0, SEEK_CUR);
```

- Tiedoston koon saa selville kutsulla

```
size = lseek(fd, 0, SEEK_END);
```

- lseek():llä voi siirtää kirjoituspositiota reippaasti viimeisen kirjoitetun tavun ohikin
  - tiedostoon voi jättää aukkoja
  - luettaessa aukosta saadaan nolla-arvoa (merkkiä ' \0')

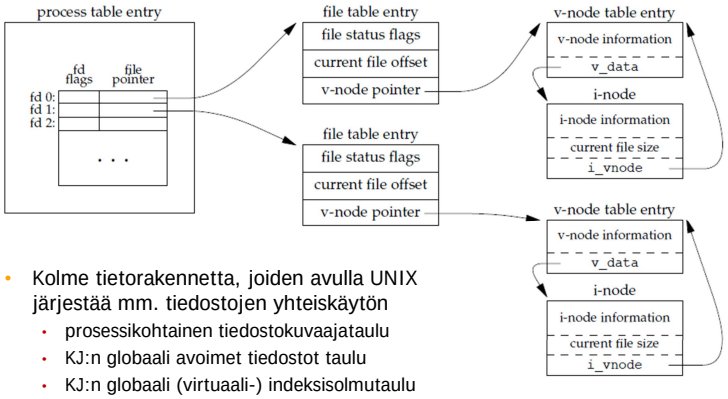
```c
#include <unistd.h>
#include <fcntl.h>
void main(int argc,char *argv[]) { // gulp.c
   int fd;  off_t size; char *buf;

   if (argc != 3)
        err_exit("Usage: gulp infile outfile");

   if ((fd = open(argv[1],O_RDONLY)) < 0)
        err_exit("open");

   size = lseek(fd,0,SEEK_END);  // varaa riittävän iso puskuri
   if ((buf = (char *)malloc(size)) == NULL)
        err_exit("malloc");

   lseek(fd,0,SEEK_SET);  // aseta positio takaisin alkuun
   if (read(fd,buf,size) != size)  // lue koko tiedosto kerralla
        err_exit("read");
   close(fd);

   if ((fd = open(argv[2], O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR)) < 0)
        err_exit("open");

   if (write(fd,buf,size) != size) // tallenna kaikki kerralla tiedostoon
        err_exit("write");
   close(fd);
   free(buf);  // vapauta dyn. varattu muisti
   exit(EXIT_SUCCESS);
}
```

Jatkossa oma apufunktio
- tulostaa msg:n ja errno-virheilmoituksen

```c
void err_exit(char *msg) {
   perror(msg);  // stderr-tiedostoon
   exit(EXIT_FAILURE)
}
```


TIEDOSTOJEN YHTEISKÄYTTÖ

Tietorakenteet

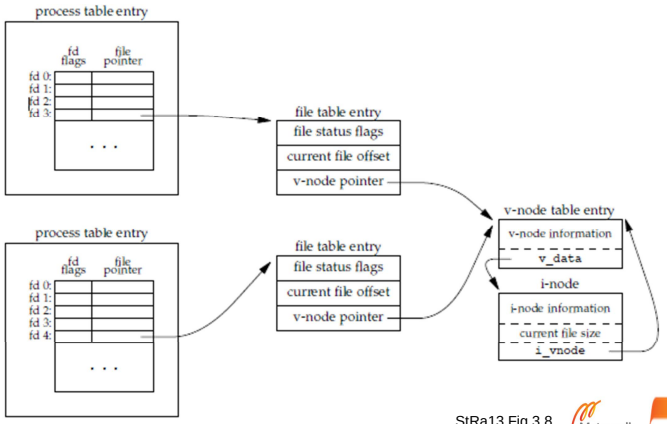


- Kolme tietorakennetta, joiden avulla UNIX järjestää mm. tiedostojen yhteiskäytön
  - prosessikohtainen tiedostokuvaajataulu
  - KJ:n globaali avoimet tiedostot taulu
  - KJ:n globaali (virtuaali-) indeksisolmutaulu

StRa13 Fig 3.7

- Tiedostokuvaajataulu (file descriptor table)
  - jokaisella prosessilla omansa => osa prosessinkuvaajaa
  - tästä käy ilmi yhden prosessin avoimet tiedostot
  - tiedostokuvaajaa (se pieni kok.luku) vastaavassa lokerossa linkki avoimet tiedostot tauluun sekä lipukkeita (mm. `O_CLOEXEC`)
- Avoimet tiedostot taulu (file table, open file table)
  - globaali, ts. sisältää kaikkien prosessien tietoja
  - tiedoston käsittelypositio (current file offset)
  - viitelaskuri (monestako paikasta tulee viite)
  - linkki indeksisolmutauluun
  - lipukkeita
- Indeksisolmutaulu / Virtuaalisolmutaulu (i-node table / v-node table)
  - globaali
  - yksi 'alkio' kutakin avattua tiedostoa kohden
  - levyltä muistiin tuodut tiedostokohtaiset indeksisolmut
    - = tiedostoattribuutit: nimi, koko, omistaja, käyttöoikeudet, aikaleimat, ...
    - = datalohkojen numerot
  - lisätietoa: viitelaskuri, lipukkeita

Yksi tiedosto - monta prosessia



StRa13 Fig 3.8

Yksi tiedosto - monta prosessia

- Sama tiedosto voi olla auki usealla prosessilla
  - vain yksi alkio indeksisolmutaulussa per tiedosto
    - tiedoston attribuuttitiedot yhdessä paikassa, yhteiskäytössä
  - erilliset alkiot tai yhteinen alkio avoimet tiedostot taulussa
    - erilliset luku/kirjoituspositiot tai
    - yhteinen luku/kirjoituspositio
- Kahdella prosessilla voi olla yhteinen alkio avoimet tiedostot taulussa
  - lapsi perinyt tiedostokuvaajataulun äidiltään fork() -kutsussa
  - lapsiprosessi on äitiprosessina klooni
- Saman prosessin kaksi tiedostokuvaajaa voi osoittaa samaan avoimet tiedostot taulun alkioon
  - prosessi itse duplikoinut kuvaajia systeemikutsulla dup() tai dup2()

- read() kasvattaa käsittelypositiota luettujen tavujen lkm:llä
  - Jos pyydettiin enemmän kuin tiedostossa oli jäljellä tavuja, palauttaa luettujen tavujen määrän ja positio=tiedoston koko
  - Jos positio = tiedoston koko, palauttaa arvon 0 (= eof)
- write() kasvattaa käsittelypositiota kirjoitettujen tavujen lkm:llä
  - Jos tiedoston koko kasvaa, arvo kopioidaan i-solmutauluun.
  - Jos lipuke O_APPEND on asetettu, kopioidaan positiolle arvo ennen kirjoittamista i-solmutaulusta
    - joku muu prosessi on voinut myös kirjoittaa
    - jokainen kirjoitus menee varmasti tiedoston loppuun
- lseek() muuttaa vain positiota avoimet tiedostot taulussa
  - Se ei aiheuta koskaan siirräntää

Kuvaajan duplikointi

- Avatun tiedostokuvaajan voi kopioida toisen tiedostokuvaajan arvoksi systeemikutsulla

```c
#include <unistd.h>
int dup(int fd);
int dup2(int fd1, int fd2);
```

  - Palauttavat uuden tiedostokuvaajan
- dup() kopioi parametrina annetun kuvaajan numeroltaan pienimpään vapaaseen tiedostokuvaajaan

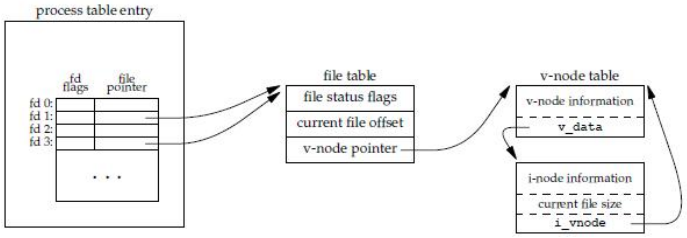
```c
if ((fd = open("AvenPuppuA.dat", O_WRONLY)) < 0)
    err_exit("Can't open AvenPuppuaA.dat");

close(STDOUT_FILENO); // tiedostokuvaaja numero 1 jää vapaaksi
dup(fd);
close(fd);
n = write(STDOUT_FILENO, buf, BUFLEN);
```

Ahaa, numero 1 ei olekaan enää näyttö, vaan...

- dup2() sulkee ensin jälkimmäisenä parametrina annetun annetun kuvaajan ja kopioi sitten ekan parametrin sen tilalle
- Edellisen esimerkin rivit close() ja dup() voi korvata rivillä

```
dup2(fd, STDOUT_FILENO);
```

- Tulostus stdoutiin (näytölle) on uudelleenohjattu menemään tiedostoon AvenPuppuA.dat



StRa13 Fig 3.9

Lipukkeiden kysely ja asettaminen

- Avatun tiedoston lipukkeita voi kysellä ja asettaa systeemikutsulla

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* int arg */);
```

  - Parametrien lukumäärä ja paluuarvo riippuu parametrista cmd
- Kutsun palauttamasta int-luvusta voi tutkia sopivilla bittitason loogisilla operaatioilla, onko tietty lipuke asetettu vai ei

cmd

- F_DUPFD
  - tee duplikaatti tiedostokuvaajasta
- F_GETFD, F_SETFD
  - kysy / aseta tdstokuvaajan lipuke FD_CLOEXEC (suljetaanko tiedosto exec:ssä)
- F_GETFL, F_SETFL
  - kysy / aseta käyttötapalipukkeita
  - O_RDONLY, O_RDWR, O_WRONLY (näille vain get)
  - O_APPEND kirjoitus aina loppuun
  - O_NONBLOCK estymätön I/O
  - O_SYNC kirjoitus aina levylle
  - O_ASYNC asynkroninen I/O + signaali
- F_GETOWN, F_SETOWN
  - kysy / aseta estymättömän I/O:n omistaja
  - ts. prosessi, joka saa SIGIO- ja SIGURG-signaalin
- F_GETLK, F_SETLK, F_SETLKW
  - kysy / aseta tiedostolukko
- F_FREESP
  - vapauta tiedostolle varattua tilaa


```
#include <sys/types.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
  int accmode, val;

  if (argc != 2) err_exit("usage: a.out <descriptor#>");

  if ((val=fcntl(atoi(argv[1]),F_GETFL,0))< 0)
      err_exit("fcntl error for fd %d",atoi(argv[1]));

  accmode = val & O_ACCMODE;
  if (accmode==O_RDONLY)      printf("read only");
  else if (accmode==O_WRONLY) printf("write only");
  else if (accmode == O_RDWR) printf("read write");
  else
      err_exit("unknown access mode");

  if (val & O_APPEND)   printf(", append");
  if (val & O_NONBLOCK) printf(", nonblocking");
  if (val & O_SYNC)     printf(", synchronous writes");

  putchar('\n');
  exit(EXIT_SUCCESS);
}
```


- Yksittäisen lipukkeen voi asettaa päälle tähän tyyliin

```
void set_fl(int fd, int flags) {/* file status flags to turn on */
   int val;

   if ((val = fcntl(fd, F_GETFL, 0)) < 0)
      err_exit("fcntl F_GETFL error");

   val = val | flags;  /* turn on flags. Bitwise OR. */

   if (fcntl(fd, F_SETFL, val) < 0)
      err_exit("fcntl F_SETFL error");
}
```

```
val    00110101
flags  00010010
     | 00110110
```

- Lipukkeen kääntö pois päältä vastaavasti

```
val = val  & ~flags;   /* turn off flags. Bitwise AND. */
```

```
val    00110101
~flags 11101101
     & 00100101
```


ATOMISUUS

- Tarkoituksena on lisätä tiedoston loppuun 100 uutta merkkiä. Mitäpä vikaa voisi olla alla olevassa koodinpätkässä?

```
if (lseek(fd, 0, SEEK_END) < 0)
   err_exit("lseek error");

if (write(fd, buff, 100) != 100)
   err_exit("write error");
```

- Vihje:
  - Entäpä, jos useampi prosessi haluaa lisätä samaan tiedostoon?

- Position asetus ja kirjoittaminen on voitava tehdä atomisesti
  - ts. peräkkäin muiden prosessien keskeyttämättä
- Ratkaisu
  - käytä avattaessa käyttötapalipuketta O_APPEND
  - erillistä lseek() -operaatiota ei tarvita, järjestelmä takaa lisäämisen tiedoston sen hetkiseen loppuun
- Ratkaisu
  - käytä systeemikutsuja (parallel read / write)

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t nbytes,
              off_t offset);
ssize_t pwrite(int fd, void *buf, size_t nbytes,
              off_t offset);
```

  - niissä voi antaa myös position, eikä erillistä lseek()-operaatiota tarvita

- Entä mikä voi mennä väärin tässä?

```
if ((fd = open(pathname, O_WRONLY)) < 0)
   if (errno == ENOENT) { // No such file or directory
       if ((fd = creat(pathname, mode))< 0)
            err_exit("creat error");
   } else
        err_exit("open error");
```

- Ratkaisu: open() ja käyttötapalipukkeet O_CREATE ja O_EXCL
  - Testi "onko tiedosto olemassa" ja "luo tiedosto" tapahtuvat atomisesti
  - Jos tiedosto olemassa open() palauttaa virheen
- Myös tiedostokuvaajien duplikointi dup2()-kutsulla on atominen operaatio

```
dup2(fd, fd2);
```

- Vaikka kutsut

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

tuottavat yleensä saman tuloksen, jää silti eräissä tapauksissa virhemahdollisuus!

TIEDOSTOLUKOT

- Saman tiedoston lukeminen useassa sovelluksessa ei ole ongelma
- Jos useampi prosessi päivittää samaa tiedostoa, voi syntyä sotkua
  - kahdesta tiliviennistä kirjautuu vain myöhempi
  - lentokoneen paikka varataan kahdelle
  - jne.
- Lukituksessa
  - prosessi varaa oikeuden tiedoston tai sen osan käsittelemiseen
  - muut prosessit jäävät odottamaan tarvittaessa lukon vapautumista
- UNIX ei huolehdi automaattisesti lukituksista, vaan ne on ohjelmoitava sovellukseen itse
  - oletuksena neuvoa-antava lukitus (advisory lock)
  - lukitus vain vihje, tiedostoa voi silti käsitellä vapaasti
  - prosessi voisi siis käyttää toisen lukitsemaa tiedostoa
  - ohjelmat kirjoitettava lukituksen suhteen "hyvätapaisiksi"

Lukkotiedosto

- Ylimääräinen tiedosto voi olla vihje ('lipuke') muille siitä, että joku käyttää parhaillaan yhteistä tiedostoa
- Kaikkien käyttäjäprosessien tulee tutkia ensin lukkotiedoston olemassaolo
  - jos lukkotiedosto on olemassa, prosessi odottaa tai tekee muuta
- Kun varsinaista tiedostoa ei enää tarvita, käyttäjä vapauttaa lukkotiedoston
- Ongelmat
  - tiedettävä lukkotiedostonkin nimi
  - jos ohjelma kaatuu, lukkotiedosto jää
  - aktiivinen odotus?
  - atomisuus?

```
do { /* varaa lukko */
  sleep(5);
  lock = open("tdsto.lock~", O_CREAT | O_EXCL, FILE_MODE);
} while (lock < 0 && errno == EEXIST);
/* avaa muutettava tiedosto */
fd = open("tdsto", O_RDWR);
...
close(fd);
close(lock);
unlink("tdsto.lock~"); /* poista lukko */
```


Pyydä KJ:ltä koko tiedoston lukitus

- KJ pitää kirjaa lukosta indeksisolmutaulun yhteydessä
  - se on kaikille samaa tiedostoa käyttäville prosesseille yhteinen
- Tiedoston voi lukita kokonaisena funktiolla

```
#include <sys/file.h>
int flock(int fd, int operation)
```

  - Palauttaa: 0, jos onnistuu, muuten -1

operation
  - LOCK_SH aseta yhteiskäytön salliva lukko (shared)
  - LOCK_EX aseta poissulkeva lukko (exclusive)
  - LOCK_UN vapauta lukko
- Yhteiskäytön salliva lukko voi olla usealla prosessilla yhtä aikaa
  - sallii usean lukea yhtä aikaa
- Poissulkeva lukko voi olla vain yhdellä kerrallaan
  - vain yksi saa muuttaa sisältöä kerrallaan, eikä tällöin saa olla yhtään lukijaa
- Jos haluaa, että kutsu ei jää odottamaan lukkoa, operaatioon voi liittää non-blocking-lipukkeen LOCK_NB
  - Paluuarvo tällöin -1 ja errno == EAGAIN

Pyydä KJ:ltä tiedoston osan lukitus

- Tavallisesti riittää, että vain käsiteltävä (muutettava) osa lukitaan
- Tiedoston osan voi lukita funktiolla

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, struct flock *lock)
```

- UNIXeissa oletuksena vain neuvoa-antava lukko (advisory lock)
  - KJ estää muita lukitsemasta jo lukittua osaa fcntl()-kutsulla
  - KJ ei estä toisten prosessien read() / write() –kutsuja (mieti miksi ei?)
    - tiedoston käyttöoikeudet (mode) sallii tai kieltää
    - ohjelmoijan vastuulla
- Vaihtoehto: pakottava lukko (mandatory lock)
  - KJ estää muita lukitsemasta jo lukittua osaa fcntl()-kutsulla
  - KJ estää myös ristiriidassa olevat read() / write() -kutsut
  - voi johtaa lukkiutumistilanteisiin (deadlock)

Lukituskomennot - cmd

- F_GETLK
  - kysy onko alueella lukkoa
  - jos alueella on lukko, palauttaa estävän lukon tiedot rakenteessa lock, muuten lock.l_type = F_UNLCK
- F_SETLK
  - varaa / vapauta lukko
  - jos varaaminen / vapauttaminen ei onnistu, palauttaa -1 ja errno == EAGAIN
- F_SETLKW
  - varaa / vapauta lukko + mahd. odotus
  - jos ei onnistu, jää odottamaan

Lukittavan alueen tiedot - struct flock_t

- struct flock_t {

```
  short l_type;      lukon tyyppi
  short l_whence;    mistä siirtymä mitataan
  long  l_start;     siirtymä lukituskohdan alkuun
  long  l_len;       montako tavua lukitaan
  pid_t l_pid;       lukon omistajan pid
}
```

- Lukittava alue annetaan kolmikkona (vrt. lseek)
  - l_whence
    - mistä siirtymä mitataan
    - SEEK_SET, SEEK_CUR, SEEK_END (alusta, nykypositiosta, lopusta)
  - l_start
    - siirtymä lukituskohdan alkuun (tavuina)
  - l_len
    - montako tavua lukitaan
    - jos l_len == 0, niin lukitus tiedoston loppuun (vaikka koko kasvaisi)

Lukon tyyppi - l_type

- L_RDLCK
  - lukulukko L_RDLCK on yhteiskäytön salliva lukko (shared)
  - ”kieltää” kaiken kirjoittamisen lukitulla osalla
  - sallii muiden prosessien lukea
- L_WRLCK
  - kirjoituslukko L_WRLCK on poissulkeva lukko (exclusive)
  - ”kieltää” muilta prosesseilta lukitun osan käytön
  - lukinnut prosessi saa itse sekä lukea että kirjoittaa
- L_UNLCK
  - lukon vapautus
  - cmd = F_SETLK ja l_type = F_UNLCK

- Lukon saa muuttaa poissulkevaksi ja päinvastoin avaamatta lukkoa välillä
- Lukot säilyvät prosessilla myös koodinvaihdossa (exec-kutsu)
  - ellei ole asetettu lipuketta FD_CLOEXEC (eli tiedosto suljetaan)
- Lapsiprosessi ei peri mammaprosessin asettamia lukkoja
  - muutenhan kaksi voisi muuttaa yhtä aikaa
  - lukossa omistajakenttä l_pid
- Kun tiedosto suljetaan, lukot poistuvat
  - kun prosessi päättyy, lukot poistuvat
  - jos dup()-kutsun jälkeen suljetaan alkuperäinen tiedostokuvaaja, lukot poistuvat myös kopioissa
    - kuvaajathan osoittavat samaan paikkaan!
  - jos tiedosto avataan prosessissa uudestaan, lukot poistuvat

Esimerkki

- Tilitiedot tiedostossa, vakiokokoinen tietue per tili

```
typedef struct {
  int nro;            // 0, 1, 2, ...
  float saldo;
  /* muuta tietoa */
} Tili_tietue;
#define LEN sizeof(Tili_tietue);
```

- Vain yksi prosessi kerrallaan saa modifioida yhtä tiliä
  - saldon lukeminen, muuttaminen ja kirjoitus oltava atomista

```
lseek(fd, i*LEN, SEEK_SET);
read(fd, tili, LEN);
tili.saldo += muutos;
lseek(fd, i*LEN, SEEK_SET);
write(fd, tili, LEN);
```

Prosessi A

```
lseek(fd, i*LEN, SEEK_SET);
read(fd, tili, LEN);
tili.saldo += muutos;
lseek(fd, i*LEN, SEEK_SET);
write(fd, tili, LEN);
```

Prosessi B

Ei OK, miksi ei?

- Tarvitaan tiedostolukitus
- Eri tilien rinnakkainen käsittely onnistuu, kun lukitaan vain yksi tili kerralla


```
int lukko(short tyyppi, int alkaen, int lkm) {
  struct flock L;
  L.l_type   = tyyppi; // rd-lukko, wr-lukko tai unlock
  L.l_whence = SEEK_SET;
  L.l_start  = alkaen * sizeof(Tili_tietue);
  L.l_len    = lkm * sizeof(Tili_tietue);
  return fcntl(fd, F_SETLKW, &L); // odota (W), kunnes saat lukittua
}
void muuta_saldo(int nro, long muutos) {
  Tili_tietue tili;
  lukko(F_WRLCK,nro,1); // lukitse vain yksi tietue, tili jonka numero on nro
  lue_tili(nro,&tili);
  tili.saldo += muutos;
  kirjoita_tili(nro,&tili);
  lukko(F_UNLCK,nro,1);
}
float tase(void) {
  float summa = 0; int nro=0; Tili_tietue tili;
  lukko(F_RDLCK,0,0); // lukitse koko tiedosto, joten len-kenttä 0
  while (lue_tili(nro,&tili)) {
    summa += tili.saldo;
    nro++
  }
  lukko(F_UNLCK,0,0);
  return summa;
}
```

- Jos halutaan tehdä tilisiirto tililtä x tilille y, on molemmat tilit lukittava
- Entäpä, jos kaksi prosessi hommailee samanaikaisesti
  - Prosessi A: siirrä tililtä x tilille y
  - Prosessi B: siirrä tililtä y tilille x
- Jos A ehtii lukita tilin x ja B ehtii lukita tilin y, niin lukkiutuu
  - kumpikin jää odottamaan, että se toinen tili vapautuisi

- Ratkaisu
  - ohjelmoi niin, että kukin prosessi lukitsee tilit aina tilinumeron mukaan kasvavassa järjestyksessä
  - Prosessi A: lukitse x, lukitse y
  - Prosessi B: lukitse x, lukitse y

TIEDOSTOATTRIBUUTIT
i-solmu

Tiedostoattribuutit

- UNIXissa tiedosto on aina vain jono tavuja
  - KJ ei ylläpidä rakennetietoja
  - vain tavulukumäärä tiedossa
- Ohjelmat (ohjelmoija) tietää käyttämänsä tiedostojen rakenteen
  - mitä tietokenttiä ja missä järjestyksessä
- Hakemisto on tiedosto, jossa on lueteltu peräkkäin pareja
  - tiedostonimi ja i-solmunumero
- Tiedoston attribuutit eli ominaisuudet (file status) on talletettu muualle indeksisolmuun (i-solmu)
  - mm. omistaja, koko, käyttöoikeudet, aikaleimat, ...
  - mitkä levylohkot kuuluvat tiedostoon
- Hakemisto voi edelleen sisältää hakemistotiedostojen nimiä
  - näin muodostuu hierarkkinen puurakenne

- Attribuutteja voi kysellä systeemikutsuilla

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *file_name, struct stat *buf)
int fstat(int fd, struct stat *buf)
int lstat(const char *file_name, struct stat *buf)
```

  - Palauttavat tiedot stat-rakenteessa (struct)
- stat() ja fstat() vievät aina todelliseen kohdetiedostoon
  - osaavat edetä symbolista linkkiä pitkin
- lstat() on muuten kuin stat()
  - mutta jos nimi on symbolinen linkki, antaa linkkitiedoston ominaisuudet

```
struct stat {  /* indeksisolmun alkuosa */
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* file type, access rights = rwx-bits*/
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* optim. blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

- Levyllä olevaan indeksisolmuun kuuluu myös lohkohakemisto
  - se ei ole osa stat-rakennetta -> sovellus ei pääse siihen käsiksi

Tiedoston sisältämän datan tyypit

- Tiedoston sisältämän datan tyypit ovat kentässä `st_mode`
  - `-` tavallinen tiedosto (teksti- tai binääri-)
  - `d` hakemisto(tiedosto)
  - `l` symbolinen linkki(tiedosto)
  - `c` erikoistiedosto, merkkilaite
  - `b` erikoistiedosto, lohkolaite
  - `p` putki(tiedosto), FIFO
  - `s` pistoke(tiedosto)

  Komento ls -l näyttää kirjaimina
- Makrot tyypin tsekkaamiseen <sys/stat.h>

`S_ISFIFO(st_mode)`	FIFO (first in first out)
`S_ISCHR(st_mode)`	merkkilaite
`S_ISDIR(st_mode)`	hakemisto
`S_ISBLK(st_mode)`	lohkolaite
`S_ISREG(st_mode)`	normaali tiedosto
`S_ISLNK(st_mode)`	symbolinen linkki
`S_ISSOCK(st_mode)`	pistoke (socket)


```
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  int i; struct stat buf;

  for (i = 1; i < argc; i++) {
    printf("%s: ", argv[i]);
    if (lstat(argv[i], &buf) < 0) {
        strerror("lstat error");
        continue;
    }
    if(S_ISREG(buf.st_mode))        printf("%s\n","regular");
    else if (S_ISDIR(buf.st_mode))  printf("%s\n","directory");
    else if (S_ISCHR(buf.st_mode))  printf("%s\n","character special");
    else if (S_ISBLK(buf.st_mode))  printf("%s\n","block special");
    else if (S_ISFIFO(buf.st_mode)) printf("%s\n","fifo");
    else if (S_ISLNK(buf.st_mode))  printf("%s\n","symbolic link");
    else if (S_ISSOCK(buf.st_mode)) printf("%s\n","socket");

    else  printf("%s\n", "** unknown mode **";
  }
  exit(0);
}
```


Tiedostojen käyttöoikeudet (access rights)

- Kentässä `st_mode` myös tiedoston käyttöoikeustiedot (ns. rwx-bitit)
- Indeksisolmussa tieto tiedoston omistajasta ja ryhmästä
  - kentät `st_uid` ja `st_gid`
- Tiedoston käyttäjään eli suorituksessa olevaan prosessiin liittyy (prosessin kuvaajassa)
  - uid ja gid (real user id, real group id)
    - todellinen prosessin omistaja ja hänen ensisijainen ryhmänsä
    - saatu istunnon alussa salasanatiedostosta (esim. /etc/passwd)
  - euid ja egid (effective uid, effective gid)
    - käytetään tiedostojen käyttöoikeuksien tarkistuksessa
    - aluksi euid = uid ja egid = gid
    - jos koditiedostolla on
      - omistajan s-oikeus (set-user-id), niin koodin suoritusajan euid=st_uid
      - ryhmän S-oikeus (set-group-id), niin koodin suoritusajan egid=st_gid
  - Esimerkki

```
$ ls -l /usr/bin/passwd     tämä on ohjelmatiedosto
-rwsr-xr-x. 1 root root ... /usr/bin/passwd
$ ls -l /etc/passwd         tämä on datatiedosto
-rw-r--r--. 1 root root ... /etc/passwd
```

    - passwd-ohjelma suoritetaan root-käyttäjänä - se saa muuttaa passwd-datatiedostoa

Suorituksessa olevaan prosessiin liittyy aina myös

- supplementary group list
  - muodostetaan ryhmätiedoston /etc/group perusteella prosessia käynnistettäessä
    - /etc/group kertoo mihin muihin ryhmiin käyttäjä kuuluu
  - käyttöä tiedostojen käyttöoikeuksien tarkistamisessa
- saved uid ja saved gid
  - alussa näihin kopioidaan euid ja egid
  - kun prosessi suorittaa set-uid ja set-gid -ohjelmia, näihin otetaan talteen euid:n ja egid:in arvot
  - myöhemmin voidaan palauttaa aiemmat arvot takaisin

Käyttöoikeuksien tarkistus

```
jos euid == root niin
  saa kaikki oikeudet
muuten
  jos euid == st_uid niin
       tarkista st_mode:n kohdasta 'user'
muuten
  jos egid == st_gid tai
      egid IN supplementary group list niin
             tarkista st_mode:n kohdasta 'group'
muuten
  tarkista st_mode:n kohdasta 'other'
```

- Muilla käyttäjillä voi olla jopa enemmän oikeuksia kuin ryhmällä
  - jos käyttäjä kuuluu tiedoston ryhmään, ei oikeuksia tarkasteta kohdasta 'other'

- Tiedoston käyttöoikeuksia voi tutkia systeemikutsulla

```
#include <unistd.h>
int access(const char *pathname, int mode)
```

  - Palauttavat 0, jos oikeus olemassa
- mode

F_OK	onko tiedosto olemassa	
R_OK	onko lukuoikeus	*uid:n tai gid:n perusteella*
W_OK	onko kirjoitusoikeus	
X_OK	onko suoritus- / läpikulkuoikeus	

```
#include <unistd.h>
int main(int argc, char *argv[]) {
   if (argc != 2) err_exit("usage: access <file>\n");
   if (access(argv[1], F_OK) == 0) printf("file exists\n");
   if (access(argv[1], R_OK) == 0) printf("read access OK\n");
   if (access(argv[1], W_OK) == 0) printf("write access OK\n");
   if (access(argv[1], X_OK) == 0) printf("execute OK\n");
   exit(0);
}
```


Käyttöoikeuksien muuttaminen

- Käyttöoikeuksia voi muuttaa systeemikutsuilla

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode)
int fchmod(int fd, mode_t mode)
```

  - Palauttavat 0, jos funktion suoritus onnistuu

- Käyttäjä voi muuttaa vain omistamiensa tiedostojen käyttöoikeuksia
- Vain root voi muuttaa muiden omistamien tiedostojen käyttöoikeuksia

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

int main(void) {
  struct stat statbuf;
  if (access("yks", R_OK) < 0)
      err_exit("no read access for yks");
  else
      printf("read access for yks OK\n");

  /* turn on set group ID and turn off group execute */

  if (stat("kaks", &statbuf) < 0)
      err_exit("stat error for kaks");

  if (chmod("kaks", (statbuf.st_mode & ~S_IXGRP) | S_IRGRP) <0 )
      err_exit("chmod error for kaks");

  /* set absolute mode to "rw-r--r-- */

  if (chmod("koli", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) < 0)
      err_exit("chmod error for koli");

  exit(0);
}
```

bittitason AND ja OR

pois

päälle

- Prosessi voi muuttaa omistamansa tiedoston omistajaa ja ryhmää systeemikutsuilla

```c
#include <sys/types.h>
#include <sys/unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group)
int fchown(int fd, uid_t owner, gid_t group)
int lchown(const char *pathname, uid_t owner, gid_t group);
```

- Tiedoston omistajaa voi vaihtaa vain root
  - et siis voi "antaa" omistamaasi tiedostoa suoraan toiselle
- Tiedoston ryhmäksi voi vaihtaa sellaisen ryhmän, johon itsekin kuuluu
  - ensisijainen ryhmä
  - tosisijaiset ryhmät (supplementary groups)
- Jos viitattu tiedosto on symbolinen linkki
  - muuttavat chown() ja fchown() todellisen kohteen omistajaa tai ryhmää
  - mutta lchown() linkkitiedoston

Tiedoston koko ja pätkäisy

- Indeksisolmun kentässä `st_size` on tiedoston koko
  - tavallisella tiedostolla, hakemistolla, symbolisella linkillä
- Koko voi olla 0
  - tällöin sille ei ole varattu yhtään datalohkoa
  - luotu vain indeksisolmu
- Hakemistoalkion koko
  - ekoissa Unixeissa: i-solmunro 4B, nimi 12B
  - ext4: i-solmunro 4B, nimen pituus 2B, alkion koko pituus 2B, nimi <x>B
- Linkkitiedoston koko on siihen talletetun tiedostonimen pituus
- Kenttään `st_blksize` on merkitty suositeltavin I/O lohkon koko
  - ts. paras puskurikoko read()-kutsussa
  - Välikysymys: miksi se on i-solmussa?
- Kentässä `st_block` on tiedostolle allokoitujen lohkojen lukumäärä
  - 512B:n lohkoina

- Tiedostossa voi olla 'reikiä', esimerkiksi

```
$ ls -l core
-rw-r--r-- 1 stevens 8483248 ... core
$ du -s core                                    disk usage
272    core
```

- Luettaessa 'reiän' kohdalta, read() palauttaa muuttujassa arvoa '\0'
- Tiedoston kokoa voi muuttaa systeemikutsuilla

```c
#include <sys/types.h>
#include <unistd.h>
int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
```

- Uusi pituus on length
  - Jos `length < st_size`, niin lopusta häviää tavuja
  - Jos `length > st_size`, niin tiedostoon loppuun syntyy 'reikä'

Uudelleennimeäminen ja poisto

- Tiedoston nimeä voi vaihtaa systeemikutsulla

```c
int rename(const char *oldname, const char *newname)
```

- Toiminta vaihtelee sen mukaan onko oldname ja/tai newname tiedosto vai hakemisto
- Tiedoston voi merkitä poistettavaksi systeemikutsulla

```c
int unlink(const char *pathname)
```

- Jotta tiedoston voisi nimetä uudelleen tai poistaa, on hakemistoon oltava sekä w-oikeus että x-oikeus
  - oikeus käyttää polkunimessä (x)
  - oikeus muuttaa hakemistotiedoston sisältöä (w)
- Jos ns. sticky bit (kentässä st_mode) on asetettu, tiedoston voi poistaa
  - vain hakemiston tai tiedoston omistaja
  - tai root

- Tiedosto poistuu vasta, kun viimeinenkin siihen osoittava linkki katkaistaan
- unlink() poistaa aina hakemistoalkion ja vähentää indeksisolmussa olevaa linkkien lukumäärää yhdellä
  - Jos lukumääräksi tulee 0, vapauttaa se myös tiedostoon kuuluneet lohkot sekä indeksisolmun
- Jos parametrina annettu tiedosto on (symbolinen) linkkitiedosto, unlink() poistaa sen, ei linkin päässä olevaa tiedostoa
- unlink() poistaa hakemistoalkion heti, mutta muut vapautukset tehdään vasta ohjelman päättyessä
  - nimi katoaa, jotta muut eivät voi enää viitata tiedostoon
  - Vihje: väliaikaiselle aputiedostolle voi tehdä unlink() heti, kun se on luotu, jolloin aputilan siivoaminen ei pääse unohtumaan


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

int main(void) {
  int fd;
  printf("Luo tempfile\n");
  if (fd=creat("tempfile", S_IRWXU)) < 0)
      err_exit("creat failed");

  printf("Tiedoston koko: %d\n",(int)lseek(fd,0,SEEK_END));
  sleep(10);
  printf("Merkitse tiedosto poistettavaksi\n");
  if (unlink("tempfile") < 0)
      err_exit("unlink failed");

  printf("Kirjoita poistuvaan tiedostoon\n");
  if (write(fd,"Yksi rivi\n",10)!=10)
      err_exit("write failed");

  printf("Tiedoston koko: %d\n",(int)lseek(fd,0,SEEK_END));
  close(fd);
  exit(0);
}
```


Aikaleimat

- Indeksisolmussa (struct stat) on kolme aikaleimaa

`st_atime`	milloin tdstoa viimeksi käytetty (accessed)
`st_mtime`	milloin tdstoa viimeksi muutettu (modified)
`st_ctime`	milloin i-solmua viimeksi muutettu (changed)

- Tiedoston luontiaikaa ei siis ole tallennettuna
- Komento `ls -l` näyttää oletuksena kentän `st_mtime`

`ls -l` tai `ls -lt`	milloin tiedostoa viimeksi muutettu
`ls -lu`	milloin tiedostoa viimeksi käytetty
`ls -lc`	milloin indeksisolmua viimeksi muutettu

- Kenttää `st_mtime` hyödynnetään esim. varmuuskopioinnissa
  - kun halutaan tehdä ns. täydennyskopio
- Indeksisolmun tietojen kysely systeemikutsulla stat() ei muuta aikaleimoja
- st_atimen tai st_ctimen päivitys ei vaikuta st_ctimeen
- Aikaleimoja voi muuttaa komentotulkin komennolla `touch`

- Omistamansa tiedoston aikaleimoja `st_atime` ja `st_mtime` voi muuttaa systeemikutsulla

```
int utime(char *filename, struct utimbuf *buf);
#include <utime.h>
struct utimbuf {
  time_t actime;
  time_t modtime;
}
```

- Jos buf = NULL // aseta aika
  st_atime = current time
  st_mtime = current time
  muuten // hae aika
  st_atime = buf.actime
  st_mtime = buf.modtime

HAKEMISTOJEN KÄSITTELY

Systeemikutsut

- Hakemisto luodaan systeemikutsulla

```
int mkdir(const char *pathname, mode_t mode)
```

- Tyhjä hakemisto poistetaan systeemikutsulla

```
int rmdir(const char *pathname)
```

- Työhakemistonsa (current working directory) nimen voi selvittää systeemikutsulla

```
char *getcwd(char *buf, size_t bufsize)
```

- Työhakemistoa voi vaihtaa systeemikutsulla

```
int chdir(const char *pathname)
int fchdir(int fd)
```


Hakemistotiedoston lukeminen

- Vain KJ voi kirjoittaa hakemistotiedostoon (siksi ei kirjoitusoperaatiota)
- Hakemistotiedoston käsittelyä varten on kutsut

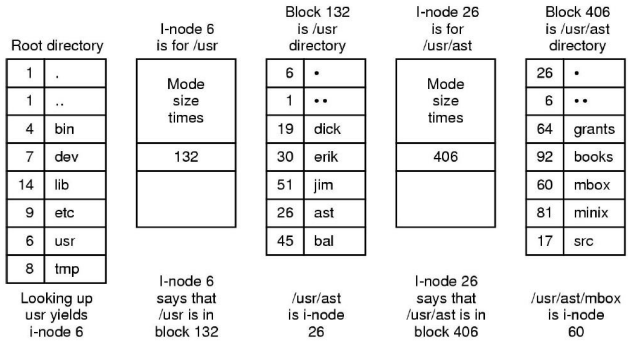
```
#include <sys/types.h>
#include <dirent.h>
struct dirent {
  ino_t d_ino;
  /* tässä välissä myös 3 toteutukseen liittyvää kenttää */
  char d_name[NAME_MAX + 1];      // usein NAME_MAX == 255
}
DIR *opendir(const char *name);      palauttaa: osoitin syötevirtaan
struct dirent *readdir(DIR *dirp); palauttaa: yksi hakemistoalkio
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
```

- readdir() palauttaa seuraavan hakemistoalkion, ja lopussa tai virhetilanteessa NULL
- rewinddir() aloittaa uudelleen alusta

(DIR on toteutuksen sisäinen rakenne - ei tarvitse tuntea tarkemmin, vrt. FILE *)


```
#include ...
#include<sys/types.h>
#include<dirent.h>

#int main(int argc, char *argv[]) {
  DIR *dirp;
  struct dirent *dent;
  if (argc < 2)
      err_exit("Usage: dir <directory>\n");
  if ((dirp=opendir(argv[1])) == NULL)
     err_exit("opendir failed");
  printf("Hakemisto %s\n",argv[1]);
  while(dent=readdir(dirp)) {
     printf("+ Tiedoston nimi: %s\n", dent->d_name)));
  }
  closedir(dirp);
  exit(0)
}
```


Hakemistopolun läpikäynti



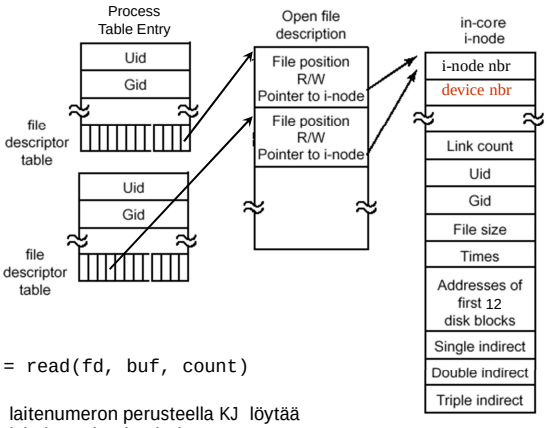
- Mistä löytyy tiedoston /usr/ast/mbox tiedot?
  - juurihakemisto / on aina vakiopaikassa levyllä (i-solmun numero aina 2)

Tane09 Fig 4-35

TIEDOSTOATTRIBUUTIT
Lohkohakemisto

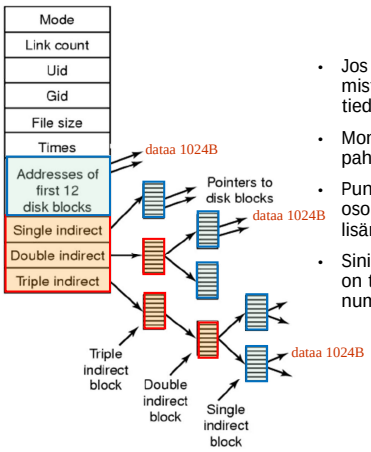
Lohkohakemisto

- i-solmussa myös 15 alkion lohkohakemisto
  - 12 suoraa tdston lohkonroa
  - lohkonro, jossa 256 tdston lohkonroa
  - lohkonro, jossa 256 lohkonroa, joissa 256 tdston lohkonroa
  - lohkonro, jossa 256 lohkonroa, joissa 256 lohkonroa, joissa 256 tdston lohkonroa
- Lohkon koko 1 KB ja lohkon numerolle 4 B
- Pääosa UNIX-tiedostoista kooltaan alle 12 KB
  - lohkonumerot nopeasti selvillä
- Uudet tiedostojärjestelmän toteutukset selviävät suuristakin tiedostoista
  - Linuxissa extended filesystem (ext2, ext3 ja ext4)
  - ext4: taltio ~ 1 EB (exabyte, 10^{18}), tiedosto ~ 16 TB (terabyte 10^{12})




```
n = read(fd, buf, count)
```

- laitenumeron perusteella KJ löytää laitekuvaajan ja ajurin

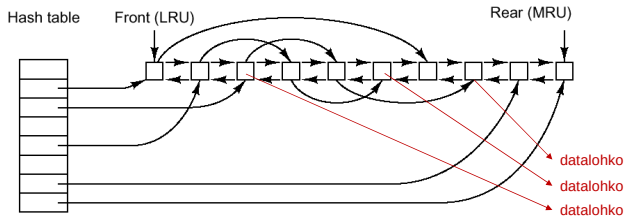


- Jos lohkon koko on 1024 B, niin mistä löytyy lohko, jossa on tiedoston 1 024 000:s tavu?
- Montako levyhakua tarvitaan pahimmassa tapauksessa?
- Punaisella värillä merkityt lohkot osoittavat kirjanpitoon tarvittaviin lisärakenteisiin
- Sinisellä värillä merkityissä lohkoissa on tiedostoon kuuluvien lohkojen numeroita

Tane09 Fig 10-34

Lohkopuskuri (buffer cache, disk cache)

Tane09 Fig 4-28



- Käsiteltävät levylohkot säilytetään keskusmuistissa lohkopuskurissa
  - jos tarvittava lohko on siellä, ei tarvita levyhakua
  - ei väliä minkä prosessin pyynnöstä tuotu / luotu
- Hajautustaulu etsinnän nopeuttamiseksi
  - avaimena laitenro, lohkonro
- Tunnussolmu (kuvan neliöt)
  - laitenro, lohkonro, linkkejä, Modified, Free
- Puskurit kokonaisina erillisellä alueella
  - tunnussolmussa viite (linkki) varsinaiseen puskuriin

Kertauskysymyksiä

- Mitä ongelmia tiedostojen yhteiskäyttö pakottaa ratkomaan?
- Miten tiedoston käyttöoikeudet tarkastetaan?
- Mitä tarkoitetaan neuvoa-antavalla (advisory) tiedostolukituksella?
- Miksi i-solmuja? Miksi ei attribuutit ja nimi samassa paikassa?
- Mitä tietoa i-solmussa?
- Tiedostojenhallinnan tietorakenteet?
- Miksi lohkopuskureita?