T-110.6120: UNIX-sovellusohjelmointi

Jukka Manner
Teknillinen korkeakoulu
Tietoliikenneohjelmistojen ja multimedian laboratorio
jmanner@tml.hut.fi
http://www.tml.hut.fi/Opinnot/T-110.6120/
http://www.tml.hut.fi/~jmanner/

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

UNIX-sovellusohjelmointi

- Luennot torstaisin 14-17
- Sali T2
- Luennoija: minä (Jukka Manner)
- Harjoitustöiden ohjaus ja tarkastus: Miika Komu
- Esitietovaatimukset (ehdottomat)
  - Käyttöjärjestelmät, rinnakkaisohjelmointi, C-ohjelmointi
- Suoritus:
  - Tentti 40 pistettä
  - Harjoitustyöt x 2 á 10 pistettä
  - Yhteensä 60 pistettä

Yleisesittely

- Portable Operating System Interface (POSIX) on IEEE:n standardi, jonka tarkoitus on määritellä yhteinen rajapinta helpottamaan ohjelmien siirtoa alustalta toiselle
- UNIX03 muodostaa ns. "Single Unix Specification" määrittelyn, jonka tarkoitus on antaa tarkka muoto UNIX-nimiselle käyttöjärjestelmälle.
- UNIX03 sisältää POSIX-standardin, mutta lisää siihen erilaisia uusia ja edistyneempiä ominaisuuksia

Yleisesittely

- UNIX03 on laaja määrittely ja tällä kurssilla käydään läpi osa "System Interfaces and Headers".
- UNIX03 on jatkoa ns. X/-standardille.
- Käytännössä POSIX ja UNIX03-standardeissa on kyse C-kielisistä funktio-kutsuista ja niiden oikeaoppisesta käytöstä.
- Kurssilla perehdytään UNIX03-määrittelyn alaisiin funktiokutsuihin ja opiskellaan niiden käyttöä omissa ohjelmissa.
- Linux-käyttöjärjestelmä toteuttaa UNIX03-määrittelyn lähes täysin.

Sisältö

- **Osa I (luennot 14.9. - 19.10.):**
  - Työkaluja ja niiden käyttö
  - Virhetilanteiden hallinta
  - Ohjelman suoritusympäristö
  - Tiedostojärjestelmä
  - Signaalit ja niiden käsittely
  - Prosessien luonti ja päättyminen, syvällisempi prosessien hallinta
  - Siirräntää eri muodoissaan (estymätön, asynkroninen, ym.)
- **Osa II (luennot 2.11. - 7.12.):**
  - Prosessien välinen vuorovaikutus ja kommunikointi
  - Säikeet ja säikeiden käsittely, synkronointi, ym.
  - Muuta kivaa

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Kurssin suoritus

- Kurssi muodostuu luennoista ja harjoituksista, sekä kurssikokeesta. Kurssilla ei ole tavallisia laskuharjoituksia.
- Kurssi on jaettu kahteen osaan. Kukin osa sisältää luentoja ja niiden päätteeksi tehdään pienimuotoinen harjoitustyö.
- Harjoitustyönä palautetaan C-kielinen lähdekoodi sekä oppimispäiväkirja.
- Kurssin maksimipisteet ovat **60** pistettä, joista **40** voi saada tentistä ja **20** pistettä harjoitustöistä (**10** pistettä kustakin harjoitustyöstä).
- Harjoitustyöt on suoritettava hyväksytysti.

Harjoituksista

- Harjoitus on C-kielinen ohjelma, jossa käytetään luennoilla esiteltyjä funktiokutsuja.
- Tarkoitus on harjoitella luentojen asioita omakohtaisesti.
- Harjoitukset tehdään yksin.
- Harjoituksena tuotetun ohjelman pitää sisältää tietty määrä luennoiduista menetelmistä.
- Ohjelman ei varsinaisesti tarvitse tehdä mitään kovin järkevää, kunhan työssä on selvästi harjoiteltu luennoitujen funktoiden käyttöä.
- Harjoituksen toteutukseen pitää myös kuulua oma kirjasto, ts. osa työn funktioista pitää olla käytettävissä siististi muistakin ohjelmista.
- Kunnollinen Makefile kuuluu myös vaatimuksiin.

Aiheista ja palautuksesta

- Aiheita annetaan pääosin kurssin vetäjien toimesta.
- On myös mahdollista toteuttaa oma aihe, kunhan siitä erikseen sovitaan, ja syntynyt ohjelma käyttää riittävässä määrin luennoituja menetelmiä.
- Erikseen on myös mahdollista palauttaa aikaisemmin tehty ohjelma, joka täyttää edellämainitut kriteerit. Tällöin opiskelijan on osoitettava, että työ on hänen itsensä tekemä. Mukaan pitää kuitenkin kirjoittaa oppimispäiväkirja työstä.
- Kustakin harjoitustyöstä palautetaan C-kielinen lähdekoodi sekä oppimispäiväkirja.
- Ohjelman saa kehittää millä tahansa alustalla (Linux, Solaris, FreeBSD, Windows/Cygwin, Mac, mutta sen pitää kääntyä myös Linux-ympäristössä.

Harjoitustöiden aikataulu

- 1. harjoitustyö 12.11. mennessä
- 2. harjoitustyö 31.12. mennessä
- Nämä ajat ovat ehdottomat, sillä aikaa työn tekemiseen on runsaasti, ja kurssilla monta taukoa.

Kurssikirjoja

- W. Richard Stevens and Stephen A. Rago: Advanced Programming in the UNIX Environment, Addison-Wesley, 2005, ISBN: 0201433079

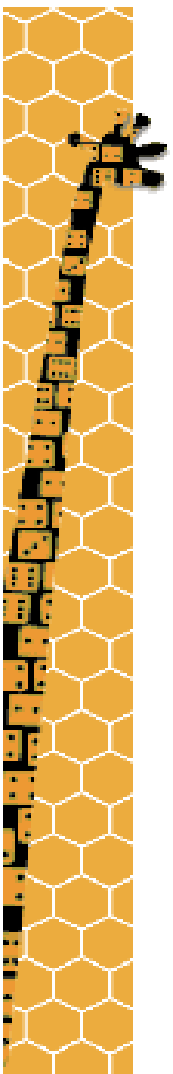
- Marc J. Rochkind: Advanced UNIX Programming (2nd Edition) , Pearson Education; 2 edition (April 29, 2004), ISBN: 0131411543.

Taustakirjallisuutta

- Manuaalisivut
- UNIX03 käyttöjärjestelmärajapinnan määrittely
- Robins and Robins, UNIX Systems Programming, Prentice Hall, 2003.
- Stevens, R: Advanced Programming in the UNIX(R) Environment, Addison-Wesley Professional; 1st edition (May 1, 1992), ISBN: 0201563177. (Wanha versio, silti erittäin ajankohtainen)
- Gallmeister B.: POSIX.4: Programming for the real word, O'Reilly, 1995, ISBN: 1-56592-074-0.
- Butenhof D.: Programming with POSIX Threads, Addison-Wesley, 1997, ISBN: 0-201-63392-2.
- Stevens, R: UNIX Network Programming, Volume 2: Interprocess Communications (2nd Edition), Prentice Hall PTR; 2nd edition (August 25, 1998), ISBN: 0130810819.
- Sarwar and Al-Saqabi, Linux and Unix Programming Tools, Addison Wesley, 2003.

Työkaluja

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY



Työkaluja

- gcc
- gdb
- gprof
- time
- make
- ar
- ranlib
- nm

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

GCC:n käyttö

- GNU Compiler Collection (gcc) on kääntäjäperhe, joka tuntee mm. C, C++, Objective-C, Fortran, Java, ja Ada ohjelmointikielet
- Tyypillisimmät komentoriviparametrit
  - - g: lisää virheenjäljitystä varten lisäinformaatiota
  - - O2 tai -O3: optimoi koodia
  - -Wall ja -pedantic: anna varoituksia koodista (ei fataalit)
  - -l*kirjasto (esim -lm, -lpthread):* linkkaa mukaan kirjasto
  - -L*polku*,-I*polku*: epästandardin kirjaston ja headerin sijainti
  - -D*jotain*: lisää “define jotain” käännösvaiheessa
  - -o *nimi*: luo ohjelma nimeltään “nimi”
  - -c: luo vain objektitiedosto
  - -mtune/-mcpu/-march: kääntää koodin eri prosessoriarkkitehtuureille, mahdollisesti optimoiden
- Katso Makefile-esimerkki

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Gnu Debugger (gdb)

- gdb mahdollistaa ohjelman suorituksen tutkimisen, esimerkiksi virheiden löytämiseksi
- Ohjelmaan pitää kääntää mukaan ylimääräistä informaatiota (“gcc *-g*”)
- Keskeisimmät komennot (“help” komennon lisäksi):
  - break [[file:]line|func]
  - backtrace
  - [[file:]line|func]
  - run ja finish
  - info [break|func|frame|locals|var]
  - list [-|a,b|func]
  - next ja nexti, step ja stepi
  - print, whatis, ja x

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Graafisia työkaluja C-ohjelmointiin

- *ddd* ja *xxgdb* ovat graafisia käyttöliittymiä gdb:lle
- *Anjuta* (KDE:n valikko ja /opt/anjuta-1.2.2/) on patevä työkalu
- KDE:n päälle on myös oma työkalunsa, Kdevelop
- Eclipse on plugin-pohjianen työkalu, jossa oletusasennuksessa on Java-tuki, mutta C/C++-tuki löytyy softan kotisivulta.

- ... ja onhan se perinteinen “monta ikkunaa ja uEmacs/VIM/PICO editori parissa terminaalissa sekä sikana vaan printf-lauseita sekaan”-menetelmä

Suorituskyky: gprof, time ja valgrind

- gprof tulostaa ohjelman funktioiden suoritukseen kuluvan ajan
- Nyrkkisääntö ohjelmien suorituksessa on, että 80% ajasta menee suoritettaessa 20% koodia
- Ohjelman suorituksen seuranta antaa vihjeita siitä, mitä osia ohjelmasta voisi ajatella optimoitavan
- Tarvitsee kääntövaiheessa “-pg” parametrin, ohjelma ajetaan tavalliseen tapaan ja syntyy gmon.out
- Tämän jälkeen ajetaan “gprof ohjelmannimi”
- Komento “time ohjelmannimi” kertoo ohjelman, eli prosessin, suoritusajat: seinäkello, prosessoriaika ja KJ:n palvelukutsuissa kulunut aika
- *Valgrind* on työkalu, jolla voi esim. tarkistaa ohjelman muistinkäytön ja säikeiden toiminnan
- oprofile...

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Make

- Mahdollistaa suurten ohjelmointiprojekien konfiguroinnin, kääntämisen ja asentamisen automaattisesti
- Runkona on ns. Makefile-niminen tekstitiedosto, joka määrittelee ohjelmoiniprojektin ja sen eri toiminnot
- Katso esimerkki ja “man make”

Makefile esimerkki

```
CC = gcc
CFLAGS= -O2 -g -Wall -pedantic -pg
LDFLAGS= -lesim -L.
#CFLAGS+= -DTESTAALYHYT

SUFFIXES= .c .o

OBJECTS=libesim.o esim.o esim_utils.o
PROGS= libesim.a esim

all: $(PROGS)

.c.o:
        $(CC) $(CFLAGS) -c $<

esim: esim.o esim_utils.o
        $(CC) $(CFLAGS) esim.o esim_utils.o $(LDFLAGS) -o esim

libesim.a: libesim.o
        ar rv libesim.a $?
        ranlib libesim.a
        @echo "Kirjasto käännetty!"
        @echo ""

clean:
        rm -f $(OBJECTS) $(PROGS)
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ar, ranlib ja nm

- Näillä ohjelmilla voidaan luoda esimerkiksi ohjelmointikirjastoja
- ar-työkalun avulla voidaan luoda ja manipuloida arkistoja tiedostoista, esim. funktiokirjastoja
- Ranlib luo arkiston funktioista ja muuttujista indeksin, joka nopeuttaa arkiston käyttöä
- nm-työkalulla voi tulostaa arkiston, ts. kirjaston, symbolit, funktiot ja muuttujat

Kääntäjän makrot

- Ns. makroilla voi yksinkertaistaa koodia ja helpottaa sen luettavuutta
- Makroilla voi ohjata kääntäjää korvaamaan lähdekoodin käännösvaiheessa tietyt makro-nimet annetuilla syötteillä, esim. vakiot, merkkijonot, jopa funktiot (makrojen nimet tyypillisesti isoin kirjaimin):
  - *#define VAKIOLUKU 128*
  - *#define MERKKIJONO "merkkijono"*
  - *#define FUNKTIO(param) do {*
    Jotain
    Jotain
    ...
    } while (0)
  - Huom. rivinvaihdot pitää lopettaa "\"-merkkiin
  - do-while(0) silmukka on hyvä olla

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Kääntäjän makrot

```
POSITIIVISEKSI (x) do {\
        if(x < 0) x = x * -1; \
        } while (0)

if (kaikki_ok())
      POSITIIVISEKSI ;
else tee_jotain();
...
```

syntyy:

```
if (kaikki_ok())
      do{
           if (x < 0) x = x * -1;
           }while (0) ;
else tee_jotain();
```

```
POSITIIVISEKSI (x) if(x < 0) x = x * -1;

if (kaikki_ok())
      POSITIIVISEKSI ;
else tee_jotain();
```

syntyy:

```
if (kaikki_ok())
      if(x < 0) x = x * -1; ;
      else tee_jotain();
```

Ei mene kääntäjästä läpi ja kun menee, lopputulos ei ole haluttu...

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Virhetilanteiden käsittely

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Virhetilanteiden käsittely

- Periaatteena että jokaisen systeemikutsun onnistuminen on tarkistettava.
- Virhetilanteessa systeemikutsut ja kirjastofunktiot palauttavat yleensä arvon -1 tai NULL ja asettavat muuttujalle *errno* virheen numeron.
- Onnistunut systeemikutsu ei muuta muuttujaa *errno*.
- Numeroa vastaavan ilmoituksen saa funktiolla strerror().

Virheiden tulostus

- Tulostuksen voi hoitaa funktiolla *perror*(msg), joka tulostaa parametrinsa, kaksoispisteen ja järjestelmän virheilmoituksen, esim.:
  perror("Virhe luettaessa tiedostoa") esim.

  Virhe luettaessa tiedostoa: Operation not permitted

- Järjestelmän virhenumerot ja niihin samaistetut tunnukset pitäisi löytyä otsaketiedostosta /usr/include/sys/errno.h
- Linuxissa ne löytyvät tiedostosta /usr/include/asm/errno.h

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki errno.c

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<sys/errno.h>
#include<string.h>

int main(int argc, char *argv[])
{
    fprintf(stderr,"EACCES: %s\n",strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);  /*viesti: ohjelman nimi, tms. */

    exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki virhetulostuksesta (mydbg.h)

```
#define CHECK(EXPR) do {\
   if (!((EXPR) + 0)) {\
     perror(#EXPR);\
     fprintf(stderr,
        "Error on process %d file %s line  %d\n",\
        getpid(), __FILE__,__LINE__);\
     fputs("Myprog: Failing assertion: " #EXPR "\n",
       stderr);\
     exit(1);\
   }\
} while (0)
```

- Jos vertailun tulos on epätosi, makro keskeyttää ohjelman suorituksen ja tulostaa, missä virhe tapahtui: prosessi, tiedosto, rivinumero ja ehto.
- Makron saa pois tekemällä esim. seuraavasti:

```
#undef CHECK
#define CHECK(EXPR) if(EXPR)
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki kaytto.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include"mydbg.h"

int main(void)
{
  int p = 0;

  CHECK(p == 1); /* Pakota virhe */
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

assert-funktio

- Toinen tapa tuottaa tarkkoja virheilmoituksia on käyttää CHECK-makron tilalla funktiota:
  - **void** *assert*(scalar *expression*)
- Funktio ei palauta mitään, mutta jos vertailu on epätosi, funktio tulostaa CHECK-makroa vastaavan tulokset ja keskeyttää prosessin toiminnan.
- *assert*-funktiota käytetään makron NDEBUG avulla:
  - Jos makrolla on arvo, kääntäjä synnyttää mainitun toiminnallisuuden ohjelmakoodin sekaan
  - Jos makro ei ole asetettu, *assert*-funktio ei saa aikaan mitään, oli vertailun tulos mikä hyvänsä, ts. sen mukana ei generoida ohjelmakoodia

Virheiden laatu

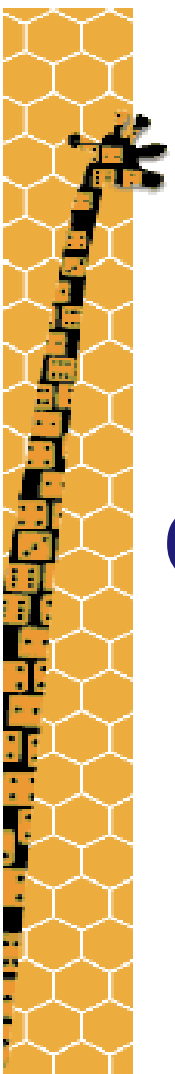
- Virheitä on kahdenlaisia:
  - Kriittisiä, jotka rikkovat jotain pahasti eikä niistä voi toipua
  - Poikkeustilanteita, joille voi tehdä jotain
- Kriittisiin virheisiin voi käyttää esitettyä CHECK-makroa tai muuta vastaavaa
- Poikkeustilanteisiin EI SAA käyttää CHECK-makroa tai muuta vastaavaa, vaan pitää rakentaa funktion ympärille tilanteen korjaava ratkaisu, esim. yritetään uudelleen tai tehdään jotain toisin.
- Jos esim. tiedoston avaaminen ei onnistu, ei kai koko ohjelmaa tarvitse välttämättä ajaa alas...

Tyypillisiä virheitä ja/tai kääntäjän herjoja

- C-kielessä pitää muisti varata ja vapauttaa itse – varaamisen kanssa tulee usein tehtyä virheitä.
- Funktiot yleensä tarvitsevat parametrina muistialueen koon
- Huomioi ero viiteparametrin (int *) ja muuttujaparametrin (int) välillä – viiteparametri osoittaa muistialueeseen, josta tieto on
- Jos tietorakenne on viiteparametri, sen yksittäisiin kenttiin viitataan “->” esim. “tietorakenne->nimi”
- Jos tietorakenne on arvoparametri, sen yksittäisiin kenttiin viitataan “.” esim. “tietorakenne.nimi”
- Paluuarvoja ei tarkasteta, josta seuraa että ohjelma kaatuu joskus vasta paljon myöhemmin ja “hämäristä” syistä
- Merkkijonot päättyvät yleensä NUL ('\0') merkkiin, mutta joskus ei – monet funktiot kuitenkin olettavat löytävänsä '\0':n
- Toisinaan ohjelmat pitää osata sulkea hallitusti
- Onko erilaiset käyttöoikeudet varmasti kunnossa?

Muistinvapautus esimerkki

```
int foo() {
  int err = 0;
  char *a = NULL, b = NULL; /* have to be set to NULL, see
                                          out_err */
  a = malloc();
  if (!a) {
    err = -1;
    goto out_err;
  }

  b = malloc();
  if (!b) {
    err = -1;
    goto out_err;
  }

  /* do something with a and b */

 out_err:

   /* free the memory     1) in case of successful operation
                          2) in case of errors */

 if (b) free(b);

 if (a) free(a);

 return err;
}
```



Ohjelman suoritusympäristö

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Sisältö

- Komentoriviargumentit
- Ohjelman päättyminen
- Ympäristömuuttujat
- Käyttäjätiedot
- Ryhmät
- Muita ympäristökyselyjä

Komentoriviparametrit

- Ohjelmaa käynnistettäessa ohjelmalle voidaan antaa parametrejä.
- Esimerkiksi:
  $ prog -kc puppu.dat
- Ohjelman suoritus alkaa funktiosta
  int *main*(**int** argc, **char** *argv[])
- Komentotulkki välittää argumentit ohjelmalle pinossa. Niihin voi viitata muuttujilla argc ja argv[].

Komentoriviparametrien muoto

```
/*
argc  argumenttien lukumäärä
argv[]      ohjelman nimi ja argumentit merkkijonoina

Yo. esimerkissä

            argc    = 3
            argv[0] = "prog"
            argv[1] = "-kc"
            argv[2] = "puppu.dat"
            argv[3] = NULL

*/
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Komentoriviparametrien tulostus (param.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include”mydbg.h”

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

    for (i=0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Komentorivien tutkimista

- UNIX03-määrittelystä löytyy paljon hyödyllisiä funktoita, esimerkiksi:
  int *getopt*(**int** argc, **char** * const argv[],const **char** *optstring)
- Funktio mahdollistaa helpon tavan tutkia ohjelman “-” alkuisia komentoriviparametreja
- Funktiolle annetaan yhtenä parametrina lista mahdollisista komentoriviparametreista
- Funktion käyttöön liittyvät läheisesti muuttujat
  - *optarg:* komentoriviparametrin arvo, jos annettu
  - *optind:* ensimmäinen ei '-' alkuinen parametri
  - *opterr:* määrittelee tulostetaanko virheilmoitusta
  - *optopt:* virheellinen komentoriviparametri (kirjain)
- *getopt* järjestelee argv-taulukon sisältöä
- *getopt_long* mahdollistaa “--” alkuiset parametrit

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki (getopt_esim.c)

```
int main (int argc, char *argv[])
{
    int c, vikat, aflg=0, bflg=0, Eflg=0;
    char *cparam=NULL, *dparam=NULL;
    extern char *optarg;
    extern int optind, optopt,opterr;
    opterr=1;
    while ((c = getopt(argc, argv, "abc:d:Eh")) != -1) {
        switch (c) {
        case 'a':
            aflg=1;
            break;
        case 'b':
            bflg=1;
            break;
        case 'c':
            cparam = malloc(strlen(optarg)+1);
            strcpy(cparam,optarg);
            break;
        case 'd':
            dparam = malloc(strlen(optarg)+1);
            strcpy(dparam,optarg);
            break;
        case 'E':
           Eflg++;
           break;
        case 'h':
                printf("Usage: %s ...\n”); return 0;
        default: return 0;
        }
    }
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ohjelman päättyminen

- Prosessin suorituksen päättyessä "normaalisti", suoritetaan main() funktiossa jokin seuraavista
  - return(status)
  - *exit(status)*
  - *_exit(status)*
- exit() tekee standardikirjastoihin liittyvää siivousta ja kutsuu edelleen funktiota _exit().
- Normaalisti päättyvän ohjelman tulisi palauttaa 0.
- Käytä myös main-funktiossa funktiota exit().

Ohjelman päättyminen

- Prosessin suoritus päättyy myös
  - kun kutsutaan funktiota *abort()*
  - saadaan signaali, johon ei olla varauduttu
- Prosessin päättyessä suoritetaan funktiolla *atexit()* rekisteröidyt funktiot, viimeiseksi rekisteröity ensin.
- Rekisteröityjen funktioiden avulla voidaan huolehtia ohjelman varaamien resurssien vapautus.

Esimerkki atexit.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include”mydbg.h”

static void my_exit1(void), my_exit2(void);

int main(void)
{
        atexit(my_exit2);
        atexit(my_exit1);
        atexit(my_exit1);

        printf("main is done\n");
        return(0);
}
static void my_exit1(void)
{
        printf("first exit handler\n");
}
static void my_exit2(void)
{
        printf("second exit handler\n");
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ympäristömuuttujat

- Tietoa suoritusympäristöstä voi välittää prosessille myös ympäristömuuttujien avulla.
- Komentoriviargumentit on tarkoitettu yksittäisen sovelluksen omien parametrien välittämiseen.
- Ympäristömuuttujien arvot pysyvät kauan samoina ja ne ovat usealle sovellukselle yhteisiä.
- Ympäristömuuttujalla on nimi ja arvo. Käytännössä se on merkkijono, joka on muotoa nimi=arvo.
- Huom. ympäristömuuttujien muutokset ovat terminaalikohtaiset!

Ympäristömuuttujat komentotulkissa

- Ympäristömuuttujan asettaminen riippuu komentotulkista, esim.
  - $ setenv KURSSI UNSO (tcsh)
  - $ export KURSSI=UNSO (bash)
- Jos ympäristömuuttuja halutaan asetaa vain yhden ohjelman suoritusajaksi, voi käyttää komentoa env.
  - $ env KURSSI=UNSO prog -kc puppu.dat
- Ympäristömuuttujaan viitattaessa on käytettävä tunnuksen edessä $-merkkiä
  - $ echo $KURSSI
- Kaikkien ympäristömuuttujien arvot saa tcsh:ssä listattua komennolla
  - $ printenv

Ympäristömuuttujat C-kielessä

```
int
main(int argc,char *argv[],char *envp[])

envp[]    ympäristömuuttujat merkkijonoina
          envp[0] = "ARCH=sun4"
          envp[1] = "DISPLAY=:0.0"
          ...
          envp[n] = NULL
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ympäristömuuttujien asetus

- Ympäristömuuttujia voi käsitellä myös esittelemällä koodissa muuttujan **extern char **environ**;
- Tai käyttämällä funktioita:
  - **char** **getenv*(**const char** **name*): palautaa ympäristömuuttujan name arvon.
  - **int** *putenv*(**const char** **str*): asettaa ympäristömuuttujan arvon.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki env.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include"mydbg.h"

extern char **environ;

int main(void) {
  printf("Tunnus on %s\n",getenv("USER");

  putenv("KURSSI=Unso");

  printf("Muuttuja asetettu\n");
  printf("Kurssi on %s\n",getenv("KURSSI");
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Huomioita

- putenv() ei muuta main-funktion "envp"-argumenttia, mutta extern-muuttuja environ muuttuu.
- getenv() näkee uudet muutetut arvot.

Käyttäjätiedot

- Unix on monen käyttäjän moniajojärjestelmä ja siksi tarvitaan resurssien hallittua jakelua, kiintiöintiä, tiedostojen yhteiskäyttöä ja tiedon suojausta luvattomalta käytöltä.
- Resurssien jaon toteutus perustuu käyttäjätunnuksiin, salasanoihin ja käyttäjäryhmiin.
- Käyttäjän yksikäsitteinen tunniste on :
  - käyttäjätunnus (username)
  - käyttäjänumero (user ID, uid)

Käyttäjätietojen käyttö

- Käyttäjätunnusta tarvitaan sisäänkirjoittautumisessa ja käyttäjien välisessä kommunikoinnissa.
- Käyttäjänumeroa käytetään prosessien suorituksessa ja tiedostojen käyttöoikeuksien määräämisessä.
- Käyttäjän perustiedot ovat yhdellä rivillä tavallisesti tiedostossa /etc/passwd.

Salasanatiedoston rakenne

- Rivin rakenne on tavallisesti:
  - kjätunnus:salasana:uid:gid:gecos:hakemisto:ohjelma
- Esimerkki
  - jmanner:x:10:10:Jukka Manner:/home/jmanner:/bin/bash
  - uid: käyttäjänumero
  - gid: ryhmätunnus
  - gecos: ylimääräistä tietoa, jota ei varsinaisesti käytetään mihinkään, esim. puhelinnumero, nimi, tms.
  - hakemisto: käyttäjän kotihakemisto
  - ohjelma: ohjelma, joka käynnistetään käyttäjän sisäänkirjoittautuessa, yleensä komentotulkki.

Käyttäjätietojen käsittely ohjelmassa

- Uid-numeron saa selville funktiolla
  uid_t *getuid*(**void**)
- käyttäjätunnus selviää funktiolla
  char **getlogin*(**void**)
- Muut tiedot voi hakea käyttäjätunnuksen tai -numeron perusteella funktioilla:
  struct passwd **getpwnam*(**const char** *name)
  struct passwd **getpwuid*(**uid_t** uid)

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

struct passwd

```
#include<pwd.h>

struct passwd {
    char    *pw_name;    /* user name */
    char    *pw_passwd;  /* user password */
    uid_t   pw_uid;      /* user id */
    gid_t   pw_gid;      /* group id */
    char    *pw_gecos;   /* real name */
    char    *pw_dir;     /* home directory */
    char    *pw_shell;   /* shell program */
};
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ryhmät

- Ryhmän yksikäsitteinen tunniste on:
  - ryhmätunnus (groupname)
  - ryhmänumero (group ID)
- Ryhmätunnusta käytetään prosessien suorituksessa ja tiedostojen käyttöoikeuksien määräämisessä.
- Ensisijainen ryhmä on kirjattu salasanatiedostoon.
- Sen lisäksi käyttäjä voi kuulua toissijaisiin ryhmiin. Toissijaisten ryhmien tiedot ovat tiedostossa:
  - /etc/group

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ryhmätiedot

- Rivin rakenne on:
  - ryhmätunnus:salasana:gid:jäsenten kjätunnukset
- Esimerkki
  - games:x:10:jmanner,paakki,kutvonen
- Komentotulkissa omat ryhmätunnukset saa komennolla
  - $ groups

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ryhmätiedot

- Sovelluksessa ensisijainen ryhmätunnus selviää funktiolla **gid_t** *getgid*(**void**) ja toisijaiset ryhmänumerot funktiolla **int** *getgroups*(**int** gidsetsize, **gid_t** grouplist[])
- Funktio täyttää taulukon grouplist toissijaisilla ryhmänumeroilla.
- Funktion ens. parametri on taulukon koko. Jos kooksi annetaan 0, palauttaa ryhmien lkm:n eli saa selville taulukon tilantarpeen.
- Muut tiedot voi hakea ryhmätunnuksen tai -numeron perusteella funktioilla:
  struct group **getgrnam*(**const char** *name
  struct group **getgrgid*(**gid_t** gid)

Esimerkki groups.c

```
#include<stdio.h>
#include<stdlib.h>
#include<pwd.h>
#include<grp.h>
#include<unistd.h>

int main(void)
{
  struct passwd *pwd;
  struct group *grp;
  uid_t user_id;
  gid_t group_id,*groups;
  int size,i;
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkin jatkoa

```
user_id = getuid();
group_id = getgid();
pwd = getpwuid(user_id);
grp = getgrgid(group_id);

if (pwd == NULL || grp == NULL ) exit(1);

printf("User id: %d, name: %s, group: %d\n",user_id,
      pwd->pw_name,group_id);
printf("Shell: %s, home: %s\n",pwd->pw_shell,pwd-
      >pw_dir);
printf("Primary group id:%d, name %s\n", group_id,
      grp->gr_name);
```

Esimerkin jatkoa

```
if ((size = getgroups(0,NULL)) > 0) {
    if(!(groups = calloc(size,sizeof(gid_t))))
       exit(1);

    if((size = getgroups(size,groups)) == -1) exit(1);

    for(i=0;i < size;i++) {
      if((grp = getgrgid(groups[i])) == NULL) exit(1);

      printf("Group id: %d, name %s\n",groups[i],
        grp->gr_name);
    }
  }
  free(groups);
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ryhmätietojen tietorakenne

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

struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);

struct group {
    char *gr_name;        ryhmätunnus
    char *gr_passwd;      koodattu salasana
    gid_t gr_gid;         ryhmänumero
    char **gr_mem;        käyttäjätunnukset
 };
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Koneen tietoja

- Sovelluksessa voi selvittää koneen tietoja funktiolla

  int *uname*(**struct utsname** *buf)

```
#include < sys/utsname.h >
struct utsname {
    char sysname[SYS_NMLN];      KJ:n nimi
    char nodename[SYS_NMLN];     Koneen solmunimi
    char release[SYS_NMLN];      KJ:n releasenumero
    char version[SYS_NMLN];      KJ:n versionumero
    char machine[SYS_NMLN];      Laitteiston tyyppi
 };
```

- Koneen nimen voi selvittää myös funktiolla

  int *gethostname*(**char** *name, **size_t** len)

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki uname.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/utsname.h>
#include"mydbg.h"

int main(void) {
  struct utsname muname;
  char cname[256+1];

  CHECK( gethostname(cname,256) == 0);

  CHECK(uname(&muname) == 0 );

  printf("Hostname: %s, sysname: %s, machine: %s\n",
         cname,muname.sysname,muname.machine);

  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ajan käsittelyä

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Aika ja kalenteri

- Unix tallettaa esim. tiedoston attribuutteihin kalenteriajan sekunteina omassa sisäisessä esitysmuodossaan. Nollakohta on 00:00:00, 1.1.1970 GMT.
- Päiväys ja kellonaika saadaan komennolla:
  - $ date
    - Wed Jan 22 10:21:58 1996
- Aikaviipaleita talletetaan kellokeskeytysten lukumääränä (engl. ticks). Sekunti on järjestelmästä riippuen 50, 60 tai 100 "tiksausta".

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Kellonajat

- Tässä muodossa on talletettu prosessin kuvaajaan:
  - *Seinäkelloaika*: aika, joka on kulunut prosessin suorituksen alkamisesta.
  - *Prosessoriaika*: aika, jonka prosessi on ollut suoritettavana prosessorissa. Se jaetaan vielä osiin:
    - aika, jonka prosessori on käyttänyt suorittaessaan prosessia käyttäjätilassa
    - aika, jonka prosessori on käyttänyt suorittaessaan prosessia. systeemitilassa (ts. suorittaessaan prosessin systeemikutsuja).

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Kellonaika sovelluksessa

- Sovellus saa kalenteriajan Unixin sisäisessä esitysmuodossa funktiolla:
  time_t *time*(**time_t** *t)
- Kalenteriajan saa pilkottua helpommin käsiteltäviin osiin funktioilla:
  struct tm **gmtime*(**const time_t** *timep)
  struct tm **localtime*(**const time_t** *timep)

Ajan käsite: time

- Ajanhetkestä “Epoch” kuluneen ajan saa selville funktiolla:
  time_t time(**time_t** *t)
- Paluuarvo on sekuntteja ajanhetkestä:
  00:00:00 UTC, January 1, 1970
- Jos **t* ei ole NULL, funktio tallettaa ajan myös annetun osoitteen päässä olevaan muuttujaan
- Ainoa virhe on, että **t* ei osoita prosessin osoiteavaruuteen (tosin ei UNIX03:ssa)
- Funktiota voi käyttää esimerkiksi kuluneen ajan mittaamiseen

Esimerkki

```
#include <time.h>

int main(...)
{
    time_t eka_aika, toka_aika, erotus;
    ...
    eka_aika=time(NULL);
    ...
    /* Jotain tekemistä */
    ...
    toka_aika=time(NULL);
    erotus=toka_aika – eka_aika;
    ...
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ajan käsite: gettimeofday

- Ajanhetkestä “Epoch” kuluneen ajan sekunteina ja microsekunteina saa selville funktiolla:
  int *gettimeofday*(**struct timeval** *tv, **struct timezone** *tz)

```
struct timeval {
    long tv_sec;         /* seconds */
    long tv_usec;  /* microseconds */
};

struct timezone {
 int  tz_minuteswest; /* minutes W of Greenwich */
 int  tz_dsttime;     /* type of dst correction */
};
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ajan käsite: gettimeofday

- Kellon tarkkuutta ei ole määritelty, ts. paluuarvon tarkkuus on määrittelemätön
- Esimerkiksi kahden peräkkäisen funktiokutsun palauttamien arvojen erotus voi olla kaukanakin todellisesta kuluneesta ajasta
- Linuxilla arvon pitäisi yleensä olla hyvin tarkka
- Funktio myös olettaa **tz* olevan NULL-osoitin, muuta toimintaa ei ole määritelty
- Kellon voi asettaa komennolla
  int *settimeofday*(const **struct timeval** *tv , const **struct timezone** *tz)
- Kellon voi asettaa ainoastaan superuser, luonnollisesti

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki

```
#include <sys/time.h>

int main(...)
{
    struct timeval eka_aika, toka_aika
    unsigned long erotus_us;

    ...
    gettimeofday(&eka_aika,NULL);

    ...
    /* Jotain tekemistä, jota mitataa */
    ...

    gettimeofday(&toka_aika,NULL);
    erotus_us=(toka_aika.tv_sec - eka_aika.tv_sec)*1000000 +
              toka_aika.tv_usec – eka_aika.tv_usec;
    printf(“Aikaa kului %.6f sekuntia\n”,
            erotus_ms/1000000.0);
    ...
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ajan käsittelyä

- Ajan merkkijonona saa funktioilla

```
char *ctime(const time_t *timep) ja
char *ctime_r(const time_t *timep, char *buf)
```

- Ajan saa myös purettuna päiviksi ja vuosiksi

```
struct tm *gmtime(const time_t *timep)
struct tm *gmtime_r(const time_t *timep,
  struct tm *result)
```

```
struct tm {
 int     tm_sec;         /* seconds */
 int     tm_min;         /* minutes */
 int     tm_hour;        /* hours */
 int     tm_mday;        /* day of the month */
 int     tm_mon;         /* month */
 int     tm_year;        /* year */
 int     tm_wday;        /* day of the week */
 int     tm_yday;        /* day in the year */
 int     tm_isdst;       /* daylight saving time */
};
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ajan käsittelyä

- Ajan merkkijonona saa myös struct tm:stä:

```
char *asctime(const struct tm *tm) tai
char *asctime_r(const struct tm *tm,char *buf)
```

- Funktio time_t mktime(struct tm *tm) palauttaa ajan sekunteina
- Lisäksi löytyy vielä seuraavat globaalit muuttujat:

```
extern char *tzname[]
extern long int timezone
extern int daylight
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Ajan käsite: times

- Prosessin käyttämä aika saadaan selville funktiolla:
  clock_t *times*(**struct tms** *buf)
- Funktio palauttaa tietystä ajanhetkestä kuluneet kellosyklit (huom, ei CPU syklejä!)
- Linuxissa tämä on järjestelmän käynnistymisen hetki
- Lisäksi funktio täyttää tietorakenteen:

```
struct tms {
  clock_t tms_utime;  /* all user processor time */
  clock_t tms_stime;  /* system time */
  clock_t tms_cutime; /* user time of children */
  clock_t tms_cstime; /* system time of children*/
};
```

- Kellosyklit sekunnissa saa selville funktiolla
  long *sysconf(***int** *name)*
- Jossa *name* on makro "*_SC_CLK_TCK*"

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki

```
clock_t start, end;
struct tms tms_start,tms_end
long sykleja_per_sec;

sykleja_per_sec=sysconf(_SC_CLK_TCK);

start = times(&tms_start);
...
/* Tee jotain */
...
end = times(&tms_end);


/* Kuluneet ajat:
   kellosyksejä: end-start
   reaaliaikaa: (end-start)/sykleja_per_sec;
   user-aikaa: (tms_end->tms_utime - tms_start->tms_utime) / clktck;
   sys-aikaa: (tmsend->tms_stime - tmsstart->tms_stime) / clktck;
*/
```

Esimerkkejä: aika[1-4].c

- aika1.c: haetaan aika kahdella eri menetelmällä ja tulostetaan ne
- aika2.c: haetaan aika ja päivämäärä ja tulostellaan niitä
- aika3.c: suoritetaan komentoriviparametrina annettu määrä yksinkertaista silmukkaa ja tulostetaan käytetty prosessori- ja seinäkelloaika (huomaa silmukassa oleva gettimeofday-kutsu, ota kutsu käyttöön ja katso miten ajat muuttuvat).
- aika4.c: suoritetaan komentoriviparametrina annetut komennot ja tulostetaan komentojen käyttämä prosessori- ja seinäkelloaika (kokeile esim. ' aika4 “aika3 1000000” ').

Sisäisen kellonajan muotoilu

- time_t muodossa olevan kellonajan voi tulostaa merkkimuodossa funktiolla:
  char **ctime*(**const time_t** *calptr);
- Palauttaa ajan muodossa (26 merkkiä):
  - “Wed Jan 22 10:21:58 1997\n\0”
- struct tm-rakenteessa olevan ajan voi muuttaa vapaasti muotoilluksi merkkijonoksi funktiolla
  size_t *strftime*(**char** *buf, **size_t** buflen, **const char** *format, **const struct** tm *tmptr)
- Funktio palauttaa taulukkoon buf talletettujen merkkien lukumäärän.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki time.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
#include"mydbg.h"

int main(void) {
  time_t ltime;
  struct tm *ltmtime;
  char htime[256];

  ltime=time((time_t *)NULL);

  ltmtime = localtime(&ltime);

  strftime(htime,256,"%A %d %B %Y",ltmtime);
  printf("Linux time: %d, local time:
       %s\n",(int)ltime,htime);
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostojärjestelmä

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Sisällys

- Tiedoston avaus
- Tiedoston luonti
- Tiedoston sulkeminen
- Tiedostosta lukeminen
- Tiedostoon kirjoittaminen
- Luku/kirjoitusposition asettaminen
- Tiedostojärjestelmän tietorakenteet
- Kuvaajien duplikointi ja uudelleenjärjestely
- Lipukkeiden kysely ja asettaminen
- Tiedostojen ominaisuudet
- Tiedostojen käyttöoikeudet
- Tiedoston uudelleennimeäminen ja poisto
- Tiedoston aikaleimat
- Hakemiston käsittely

Tiedostoon viittaaminen

- Tiedostoon viitataan nimellä lähinnä vain avattaessa, muissa kutsuissa käytetään avattaessa saatua tiedostokuvaajaa.
- Tiedostokuvaaja on kokonaisluku, joka toimii avaimena avointen tiedostojen taulukkoon.
- Tiedostokuvaaja voi osoittaa myös:
  - "bittisankoa", "mustaa aukkoa" (/dev/null)
  - laitetta (/dev/audio)
  - putkea tai nimettyä putkea (ns. fifot)
  - tiedonsiirtoyhteyttä (pistokkeet)

Tiedostojen käyttö

- Kaikille samat read(), write() ja close(), luomisessa ja avaamisessa eroja.
- Valmiiksi avatut tiedostokuvaajat:
  - 0 = STDIN_FILENO
  - 1 = STDOUT_FILENO
  - 2 = STDERR_FILENO

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedoston avaaminen

- Tiedosto avataan funktiolla
  int (**const char** *pathname, **int** oflag, . . . /* , **mode_t** mode */)
- Palauttaa tiedostokuvaajan numeron
- Käyttötapalipukkeet (oflag):
  - O_RDONLY vain lukemista varten
  - O_WRONLY vain kirjoittamista varten
  - O_RDWR sekä lukemista että kirjoittamista varten
  - O_APPEND kirjoita tiedoston loppuun
  - O_CREAT luo uusi tiedosto
  - 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 writessä, kunnes tieto todella kirjoitettu levylle

Tiedoston omistaja ja ryhmä

- Uusi tiedosto saa omistajan ja ryhmän tiedot prosessin kuvaajasta. st_uid = euid ja st_gid = egid tai st_gid = isähakemiston st_gid.
- Uudelle tiedostolle on annettava käyttöoikeudet luonnin yhteydessä (eli open() / creat() kutsussa).
- Ne eivät tule kuitenkaan sellaisenaan voimaan, vaan prosessin umask-arvo vaikuttaa i-solmuun tallettuviin arvoihin.

Tiedoston luonti ja umask

- umask-arvoa voi muuttaa funktiolla:
  mode_t *umask*(**mode_t** cmask): asettaa uuden maskin ja palauttaa edellisen umask-arvon
- Esimerkiksi, jos
  - mode = S_IRWXU | S_IRWXG | S_IRWX0
  - umask = 007
- niin luotavalle tiedostolle tulee oikeudet
  - -rwxrwx---
    - omistajalle luku-, kirjoitus- ja suoritusoikeus
    - ryhmälle samoin
    - muille ei mitään oikeuksia
- Umask siis estää tiettyjen oikeuksien asettamisen
- Eli tiedoston lopulliset oikeudet ovat
  - mode AND NOT umask

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedoston luonti (creat() tai O_CREAT)

- Tiedoston voidaan luoda myös funktiolla:
  int *creat*(**const char** *pathname, **mode_t** mode)
- Tämä vastaa kutsua:
  - fd=*open*(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
- Molemmat funktiot palauttavat tiedostokuvaajan numeron tai -1 virhetilanteessa.
- Huom. paluuarvo voi olla myös 0, jos käyttöjärjestelmässä ei ole avattuna yhtään standardivirtaa (stdin/out/err).

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Oikeudet sys/stat.h

- S_IRWXU = 00700 omistajalla luku-, kirjoitus-, ja suoritusoikeus
- S_IRUSR (S_IREAD) = 00400 omistajalla lukuoikeus
- S_IWUSR (S_IWRITE) = 00200 omistajalla kirjoitusoikeus
- S_IXUSR (S_IEXEC) = 00100 omistajalla suoritusoikeus
- S_IRWXG = 00070 ryhmällä luku-, kirjoitus- ja suoritusoikeus
- S_IRGRP = 00040 ryhmällä lukuoikeus
- S_IWGRP = 00020 ryhmällä kirjoitusoikeus
- S_IXGRP = 00010 ryhmällä suoritusoikeus
- S_IRWXO = 00007 muilla luku-, kirjoitus- ja suoritusoikeus
- S_IROTH = 00004 muilla lukuoikeus
- S_IWOTH = 00002 muilla kirjoitusoikeus
- S_IXOTH = 00001 muilla suoritusoikeus

Tiedoston sulkeminen

- Tiedosto suljetaan funktiolla **int** *close*(**int** fildes);
- Unix sulkee kaikki avoimet tiedostot, kun ohjelma päättyy.
- Sääntö on silti: Sulje tiedosto heti, kun et sitä enää tarvitse.
- Miksikö ?
  - Prosessin avointen tiedostojen lukumäärä on rajoitettu.

Esimerkki file1.c

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

int main(void)
{
  int fd,fd2;

  CHECK((fd = open("Tiedosto1",O_CREAT | O_WRONLY,
                  S_IRUSR | S_IWUSR | S_IRGRP) ) >= 0);

  CHECK(( fd2 = creat("Tiedosto2",
                      S_IRUSR | S_IWUSR | S_IRGRP)) >= 0 );

  close(fd);
  close(fd2);
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki file2.c

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

int main(void)
{
  int fd,fd2;

  umask(0);

  CHECK((fd = open("Tiedosto3",O_CREAT | O_WRONLY,
                  S_IRUSR | S_IWUSR | S_IRGRP) ) >= 0);

  umask(S_IWUSR);

  CHECK(( fd2 = creat("Tiedosto4",
                      S_IRUSR | S_IWUSR | S_IRGRP)) >= 0 );

  close(fd);
  close(fd2);
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostosta lukeminen

- Avatusta tiedostosta luetaan merkkejä funktiolla: **ssize_t** *read*(**int** fildes, **void** *buff, **size_t** nbytes)
- Palauttaa luettujen merkkien lukumäärän tai 0 kun **EOF**.
- Muista: *read()* ei aina palauta arvonaan samaa arvoa kuin on pyydettyjen merkkien lukumäärä (nbytes): tiedoston viimeinen lukupyyntö jää yleensä vajaaksi!
- Jos *read()* keskeytyi kesken lukuoperaation ennenkuin yhtään tavua ehdittiin lukea, on errno = EINTR.
- Tällöin sovelluksen on itse osattava jatkaa lukuoperaatiota siitä mihin jäätiin.

Tiedostoon kirjoittaminen

- Avattuun tiedostoon kirjoitetaan funktiolla: **ssize_t** *write*(**int** fildes, **const void** *buff, **size_t** nbytes)
- Palauttaa kirjoitettujen merkkien lukumäärän.
- Yleensä paluuarvo on aina sama kuin kirjoitettavaksi pyydettyjen merkkien lukumäärä (nbytes).
- *write()* asettaa errno = EINTR jos kirjoitusoperaatio keskeytyi ennenkuin yhtään tavua ehdittiin kirjoittaa.
- Tällöin sovelluksen on itse osattava jatkaa lukuoperaatiota siitä mihin jäätiin.

Tiedostosta lukuesimerkki readn.c

```
/* Read "n" bytes from a descriptor. */

ssize_t
readn(int fd, void *vptr, size_t n)
{
        size_t  nleft;
        ssize_t nread;
        char    *ptr;

        ptr = vptr;
        nleft = n;
        while (nleft > 0) {
                if ( (nread = read(fd, ptr, nleft)) < 0) {
                        if (errno == EINTR)
                                nread = 0;/* and call read()
                                             again */
                        else
                                return(-1);
                } else if (nread == 0)
                        break;                  /* EOF */

                nleft -= nread;
                ptr   += nread;
        }
        return(n - nleft);                      /* return >= 0 */
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostoon kirjoitusesimerkki writen.c

```
/* Write "n" bytes to a descriptor. */
ssize_t
writen(int fd, const void *vptr, size_t n)
{
        size_t          nleft;
        ssize_t         nwritten;
        const char      *ptr;

        ptr = vptr;
        nleft = n;
        while (nleft > 0) {
                if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
                        if (errno == EINTR)
                            nwritten = 0;/* and call write()
                                            again */
                        else
                            return(-1);    /* error */
                }

                nleft -= nwritten;
                ptr   += nwritten;
        }
        return(n);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki: tiedoston kopiointi (copy.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/errno.h>

#define     BUFFSIZE     8192

int main(void)
{
  int       n;
  char buf[BUFFSIZE];

  memset(buf,0,BUFFSIZE);

  while ((n=readn(STDIN_FILENO,buf,BUFFSIZE))>0)
    if (writen(STDOUT_FILENO, buf, n) != n) {
      perror("write error");
       exit(-1);
    }

  if (n < 0) perror("read error");
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Luku- ja kirjoitusposition asettaminen

- Luku/ kirjoitusposition voi asettaa funktiolla **off_t** *lseek*(**int** fildes, **off_t** offset, **int** whence);
- Onnistunut kutsu palauttaa position tiedoston alusta laskettuna.
- Uusi positio määräytyy parametrin whence arvon perusteella seuraavasti:
  - SEEK_SET positio = offset
  - SEEK_CUR positio = positio + offset
  - SEEK_END positio = tiedoston koko + offset
- Nykyposition saa selville kutsulla:
  - currpos = lseek(fd, 0, SEEK_CUR);
- Tiedoston koon saa selville kutsulla
  - size = lseek(fd, 0, SEEK_END);

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki: tiedoston kopiointi (copy2.c)

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

int main(int argc, char *argv[])
{
  int fd;
  off_t size;
  char * buf;

  if (argc != 3) { perror("Usage: copy2 infile outfile "); exit(1); }

  CHECK((fd = open(argv[1],O_RDONLY)) >= 0);

  size = lseek(fd,0,SEEK_END);
  CHECK((buf = malloc(size)));

  lseek(fd,0,SEEK_SET);
  CHECK((readn(fd,buf,size) == size));
  close(fd);

  CHECK((fd = open(argv[2], O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR)) >= 0);

  CHECK((writen(fd,buf,size) == size));

  close(fd);
  free(buf);
  exit(0);
}
```

Tiedostojen tietorakenteita

- Tiedostojärjestelmän toteutukseen liittyy kolme tietorakennetta, joiden avulla voidaan järjestää mm. tiedostojen yhteiskäyttö:
  - *Tiedostokuvaajataulu*: jokaisella prosessilla omansa eli on osa PCB:tä. Tästä käy ilmi avoimet tiedostot. Tiedostokuvaajaa (se pieni kok.luku) vastaavassa lokerossa on linkki avoimet tiedostot tauluun sekä lipukkeita (FD_CLOEXEC).
  - *Avoimet tiedostot taulu*: globaali, ts. sisältää kaikkien prosessien tietoja eli tiedoston luku- ja kirjoitusposition, viitelaskurin (monestako paikasta tulee viite), linkin indeksisolmutauluun ja lipukkeita.
  - *Indeksisolmutaulu*: globaali, ts. yksi 'alkio' kutakin avattua tiedostoa kohden. Levyltä muistiin tuotu indeksisolmu ja lisätietoa: viitelaskuri, lipukkeita.

Seuraukset

- Sama tiedosto voi olla auki usealla prosessilla, koska vain yksi alkio indeksisolmutaulussa per tiedosto.
- Kahdella prosessilla voi olla yhteinen alkio avoimet tiedostot taulussa vain, kun lapsi on perinyt avoimet tiedostot äidiltään *fork*() -kutsussa.
- Saman prosessin kaksi tiedostokuvaajaa voi osoittaa samaan avoimet tiedostot taulun alkioon, jos kuvaajia on duplikoitu systeemikutsulla *dup*() tai *dup2*().

Systeemikutsujen toiminta

- *read*() kasvattaa lukupositiota luettujen tavujen lkm:llä. Jos positio = tiedoston koko, palauttaa arvon 0 (= eof).
- *write*() kasvattaa positiota 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 eli jokainen kirjoitus menee tiedoston loppuun.
- *lseek*() muuttaa vain positiota avoimet tiedostot taulussa. Se ei aiheuta koskaan siirräntää.

Tiedostokuvaajien kopiointi ja järjestely

- Avatun tiedostokuvaajan voi kopioida toisen tiedostokuvaajan arvoksi funktiolla
  int *dup*(**int** filedes) tai
  int *dup2*(**int** filedes1, **int** filedes2)
- Palauttavat uuden tiedostokuvaajan numeron.
- *dup*() kopioi parametrina annetun kuvaajan numeroltaan pienimpään vapaaseen tiedostokuvaajaan.
- *dup2*() sulkee kuvaajan filedes2 ja kopioi sen paikalle kuvaajan filedes1. Atominen toiminto.

Tiedoston ominaisuudet

- Avatun tiedoston lipukkeita voi kysellä ja asettaa funktiolla

  int *fcntl*(**int** filedes, **int** cmd, . . . /* **int** arg */)

  - F_DUPFD - tee duplikaatti tiedostokuvaajasta
  - F_GETFD, F_SETFD - kysy / aseta tdostokuvaajan lipuke FD_CLOEXEC (ts. 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 synkroninen I/O
  - O_ASYNC asynkroninen I/O + signaali
  - F_GETLK, F_SETLK, F_SETLKW - kysy / aseta tiedostolukko

Kirjoittaminen levylle

- Moderneissa järjestelmissä on kirjoituspuskuri, joka saa aikaan viivytetyn kirjoittamisen: kirjoituspyyntöjä puskuroidaan ja lähetetään kerralla levylle.
- Jotkin sovellukset tarvitsevat välttämättä tiedon siitä, että tieto on todella kirjoitettu levylle, esim. tietokantaoperaatiosta on hyvä tietää, että tieto on mennyt levylle asti
- Tämän mahdollistavat kutsut

  int *fsync*(**int** fd): pakota tietty tiedosto levylle asti, sisältö ja mahdolliset ominaisuuksien muutokset

  int *fdatasync*(**int** fd): pakota tietyn tiedoston pelkkä sisältö levylle asti

  void *sync*(**void**): lähetä kaikki KJ:n puskuroimat kirjoitukset levylle (ei odota varsinaista kirjoitusta!)

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki (mode.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include "mydbg.h"

int check_mode(int fd)
{
  int accmode, val;

  CHECK((val=fcntl(fd,F_GETFL,0)) >= 0);

  accmode = val & O_ACCMODE;
  if (accmode==O_RDONLY) printf("read only");
  if (accmode==O_WRONLY) printf("write only");
  if (accmode == O_RDWR) printf("read write");

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

  putchar('\n');
  return(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Lipukkeiden asetus (set_fl.c)

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

int set_fl(int fd, int flags)
{
  int val;

  if ((val = fcntl(fd, F_GETFL, 0)) < 0)
    return val;

  val |=flags;

  val = fcntl(fd, F_SETFL, val);

  return val;
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedoston ominaisuudet

- Tiedosto on jono tavuja.
- Siihen liittyy nimi, attribuutit ja datalohkot.
- Unix tallettaa nämä erilleen toisistaan.
- Hakemisto on tiedosto, jossa on peräkkäin pareja: tiedostonimi ja i-solmunumero.
- Hakemisto voi edelleen sisältää hakemistotiedostojen nimiä, jolloin muodostuu puurakenne.
- Tiedoston attribuutit on talletettu indeksisolmuun.
- Indeksisolmusta käy ilmi missä tiedoston datalohkot sijaitsevat.

Tiedostojen ominaisuuksien selvittäminen

- Tiedostojen ominaisuuksia voi kysellä funktioilla:
  int *stat*(**const char** *file_name, **struct stat** *buf)
  int *fstat*(**int** filedes, **struct stat** *buf)
  int *lstat*(**const char** *file_name, **struct stat** *buf)
- *stat*() osaa kulkea symbolista linkkiä pitkin todelliseen kohdetiedostoon.
- *lstat*() on muuten kuin *stat*(), mutta jos nimi on symbolinen linkki, se antaa linkkitiedoston ominaisuudet (ei kuulu UNIX03-speksiin).
- Onnistuneen kutsun paluuarvo riippuu pyydetystä operaatiosta.
- Epäonnistunut kutsu palauttaa -1.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

struct stat

```
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    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 type */
    off_t         st_size;     /* total size, in bytes */
    blksize_t     st_blksize;  /* optim. blocksize for I/O */
    blkcnt_t      st_blocks;   /* # of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* last modification */
    time_t        st_ctime;    /* last status change */
};
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostotyypit (sys/stat.h)

- Tiedostotyypit ls -la
  - - tavallinen (teksti- tai binääri-)tiedosto
  - d hakemisto(tiedosto)
  - l symbolinen linkki(tiedosto)
  - c erikoistiedosto, merkkilaite
  - b erikoistiedosto, lohkolaite
  - p putki(tiedosto), FIFO
  - s pistoke(tiedosto)
- Tiedostotyypit (kentässä st_mode):
  - S_ISFIFO(mode) : FIFO
  - S_ISCHR(mode) : merkkilaite
  - S_ISDIR(mode) : hakemisto
  - S_ISBLK(mode) : lohkolaite
  - S_ISREG(mode) : normaali tiedosto
  - S_ISLNK(mode) : symbolinen linkki
  - S_ISSOCK(mode): pistoke

Esimerkki (filemod.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include"mydbg.h"

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

  for (i = 1; i < argc; i++) {
    (printf("%s: ", argv[i]);
    CHECK((lstat(argv[i], &buf)) == 0);

    if(S_ISREG(buf.st_mode))          ptr = "regular";
    else if (S_ISDIR(buf.st_mode))    ptr = "directory";
    else if (S_ISCHR(buf.st_mode))    ptr = "character special";
    else if (S_ISBLK(buf.st_mode))    ptr = "block special";
    else if (S_ISFIFO(buf.st_mode))   ptr = "fifo";
    else if (S_ISLNK(buf.st_mode))    ptr = "symbolic link";
    else if (S_ISSOCK(buf.st_mode))   ptr = "socket";
    else                              ptr = "** unknown mode **";

    printf("%s\n", ptr);
  }
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostojen käyttöoikeudet

- Suoritettavaan prosessiin liittyy aina: real user id (uid), real group id (gid), effective user id (euid), effective group id (egid) ja supplementary group list.
- Lisäksi voi olla: saved user id ja saved group id.
  - uid ja gid: Todellinen prosessin omistaja ja hänen ensisijainen ryhmänsä. Saatu istunnon alussa salasanatiedostosta.
  - euid ja egid: Käytetään tiedostojen käyttöoikeuksien tarkistuksessa.
- Aluksi euid = uid ja egid = gid
- euid muuttuu koodin suoritusajaksi, jos kooditiedostolla on omistajan antama s-oikeus (set-user-id).
- egid muuttuu koodin suoritusajaksi, jos kooditiedostolla on ryhmän S-oikeus (set-group-id).
- Esimerkkejä Linuxissa esim. ping ja traceroute

Tiedostojen käyttöoikeudet

- supplementary group list: Muodostetaan ryhmätiedoston /etc/group perusteella prosessia käynnistettäessä.
- Käytetään tiedostojen käyttöoikeuksien tarkistamisessa.
- saved uid ja saved gid: Näihin kopioidaan euid ja egid, kun prosessi on käynnistymässä.
- Käyttöä set-uid ja set-gid ohjelmien yhteydessä.

Tiedoston käyttöoikeuksien määräytyminen

```
jos euid = root niin
    saa kaikki oikeudet
muuten
    jos euid = st_uid niin
        tarkista kohdasta 'user'
    muuten
        jos egid = st_gid | egid IN grouplist niin
            tarkista kohdasta 'group'
        muuten
            tarkista kohdasta 'other'
```

Tiedoston käyttöoikeuksien tarkastus

- Tiedoston käyttöoikeus voidaan tarkastaa funktiolla
  int *access*(**const char** *pathname, **int** mode)
- Palauttaa 0, jos oikeus olemassa muuten -1
- mode:
  - R_OK onko lukuoikeus
  - W_OK onko kirjoitusoikeus
  - X_OK onko suoritus / läpikulkuoikeus
  - F_OK onko tiedosto olemassa

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki (access.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/types.h>
#include"mydbg.h"

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

  if (argc != 2) { printf("usage: access <file>\n"); exit(1); }

  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);
}
```

Tiedoston käyttöoikeuksien muuttaminen

- Prosessi voi muuttaa omistamansa tiedoston käyttöoikeuksia funktioilla:
  int *chmod*(**const char** *pathname, **mode_t** mode)**int** *fchmod*(**int** filedes, **mode_t** mode)
- Palauttavat 0, jos funktion suoritus onnistuu.
- Käyttäjä voi muuttaa vain omistamiensa tiedostojen käyttöoikeuksia.
- Vain super-user voi muuttaa muiden omistamien tiedostojen käyttöoikeuksia.

Esimerkki (chmod.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include"mydbg.h"

int main(void)
{
  struct stat statbuf;

   /* Haetaan foo:n oikeudet */
  CHECK((stat("foo", &statbuf)) == 0);

  /* Poistetaan foo:lta ryhmän ja muiden kirjoitusoikeus */
  CHECK((chmod("foo", (statbuf.st_mode & ~S_IWGRP & ~S_IWOTH)))
        == 0);
  /* Asetetaan bar:lle tietyt oikeudet vanhoista arvoista
        huolimatta*/

  CHECK((chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) ==
        0);

  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedoston omistajan ja ryhmän muuttaminen

- Prosessi voi muuttaa eräin edellytyksin omistamansa tiedoston omistajaa ja ryhmää funktioilla:
  int *chown*(**const char** *pathname, **uid_t** owner, **gid_t** group)
  int *fchown*(**int** filedes, **uid_t** owner, **gid_t** group)
  int *lchown*(**const char** *pathname, **uid_t** owner, **gid_t** group);
- Tiedoston omistajaa voi vaihtaa vain super-user.
- Omistamansa tiedoston ryhmän voi vaihtaa sellaiseen ryhmään johon kuuluu.
- *lchown()* muuttaa viitatessa linkkitiedostoon linkkitiedoston omistajaa tai ryhmää.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedoston uudelleennimeäminen ja poistaminen

- Tiedoston nimeä voi vaihtaa funktiolla:
  int *rename*(**const char** *oldname, **const char** *newname)
- Toiminta vaihtelee sen mukaan onko oldname ja/tai newname tiedosto vai hakemisto.
- Tiedoston voi merkitä poistettavaksi funktiolla:
  int *unlink(***const char** *pathname)
- Jotta tiedoston voisi poistaa, on hakemistoon oltava sekä w-oikeus että x- oikeus. Tiedosto poistuu, kun viimeinenkin siihen osoittava linkki katkaistaan.
- Jos ns. *sticky bit* (S_ISVTX) on asetettu, tiedoston voi poistaa vain hakemiston tai tiedoston omistaja, tai superuser.

Tiedoston poistamisesta

- *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.
- Aputiedostolle voi tehdä *unlink*() heti, kun se on luotu, jolloin aputilan siivoaminen ei pääse unohtumaan.

Esimerkki (unlink.c)

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

int main(void)
{
    int fd;

    printf("Luodaan tempfile\n");
    CHECK((fd=creat("tempfile", S_IRWXU)) > 0);

    printf("Avataan se lukemista varten\n");
    printf("Tiedoston koko: %d\n",(int)lseek(fd,0,SEEK_END));

    sleep(10);

    printf("Poistetaan tiedosto\n");

    CHECK((unlink("tempfile")) == 0);
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki (unlink.c)

```
    printf("Kirjoitetaan tiedostoon vaikka se on poistettu \n");
    CHECK(write(fd,"jepulis\n",8)==8);
    printf("Tiedoston koko nyt: %d\n",(int)lseek(fd,0,SEEK_END));
    close(fd);
    printf("Se siitä\n");
    exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostojen aikaleimat

- Indeksisolmussa on kolme aikaleimaa:
  - st_atime milloin tdstoa viimeeksi käytetty
  - st_mtime milloin tdstoa viimeeksi muutettu
  - st_ctime milloin i-solmua viimeeksi muutettu
- Komento ls näyttää oletusarvoisesti milloin tiedostoa on viimeeksi muutettu eli kentän st_mtime.
- Ko. kenttää hyödynnetään esim. varmuuskopioinnissa (täydennyskopiot).
- Indeksisolmun tietojen kysely funktiolla stat() tms. ei muuta aikaleimoja.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Aikaleimojen asettaminen

- Omistamansa tiedoston aikaleimoja st_atime ja st_mtime voi muuttaa funktiolla:
  int *utime*(**const char** *filename, **const struct utimbuf** *buf);

```
#include < utime.h>
struct utimbuf {
        time_t  actime;
        time_t  modtime;
}
Jos buf = NULL
        st_atime <- current time
        st_mtime <- current time
muuten
        st_atime <- times.actime
        st_mtime <- times.modtime
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Hakemistojen käsittely

- Hakemisto luodaan funktiolla:
  int *mkdir*(**const char** *pathname, **mode_t** mode)
- Tyhjä hakemisto poistetaan funktiolla:
  int *rmdir*(**const char** *pathname)
- Työhakemistonsa nimen voi selvittää funktiolla:
  char **getcwd*(**char** *buf, **size_t** bufsize)
- Työhakemistoa voi vaihtaa funktioilla:
  int *chdir*(**const char** *pathname)
  int *fchdir*(**int** fildes)

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Hakemistotiedoston lukeminen

- Vain käyttöjärjestelmä voi kirjoittaa hakemistotiedostoon.
- Hakemistotiedosto avataan lukemista varten funktiolla:
  DIR **opendir*(**const char** *name);
- Seuraava hakemistotiedoston alkio luetaan funktiolla:
  struct dirent ***readdir*(**DIR** *dir);
- *readdir()* palauttaa NULL arvon hakemistotiedoston lopussa tai virhetilanteessa.
- Hakemistotiedosto suljetaan funktiolla:
  int *closedir*(**DIR** *dir);
- **DIR** on oikeasti *struct __dirstream*

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

struct __dirtream ja struct dirent

```
struct __dirstream {
    int fd;                     /* File descriptor.  */
    char * data;                /* Directory block.  */
    size_t allocation;          /* Space allocated for the block.  */
    size_t size;                /* Total valid data in the block.  */
    size_t offset;              /* Current offset into the block.  */
    off_t filepos;              /* Position of next entry to read.  */
    pthread_mutex_t lock;       /* Mutex lock for this structure.  */
};


struct dirent {
#if defined _FILE_OFFSET_BITS && _FILE_OFFSET_BITS == 64
    ino64_t             d_ino;    /* inode number */
    off64_t             d_off;    /* offset to the next dirent */
#else
    ino_t               d_ino;    /* inode number */
    off_t               d_off;    /* offset to the next dirent */
#endif
    unsigned short int  d_reclen; /* length of this record */
    unsigned char       d_type;   /* type of file */
    char                d_name[256];    /* filename: UNIX03 */
};
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki: Hakemistotiedoston listaus (dir.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<dirent.h>
#include<errno.h>
#include"mydbg.h"

int main(int argc,char **argv)
{
  DIR *dir;
  struct dirent *de;

  if ( argc < 2) {
    printf("Usage: dir <directory>\n"); exit(1);
  }

  CHECK((dir=opendir(argv[1])) != NULL);
  printf("Hakemisto %s\n",argv[1]);

  while( de = readdir(dir) ) {
    CHECK((printf("+ Tiedoston nimi: %s\n",de->d_name)) > 0);
  }

  CHECK(errno == 0);
  CHECK((closedir(dir)) == 0);
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostojenkäsittelystä: tiedostokuvaajat ja FILE-osoittimet

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostojen käsittely

- Tiedostoja voidaan käsitellä periaatteessa kahdella tavalla/tasolla:
  - Alhaisella tasolla: käyttämällä suoraan käyttöjärjestelmän palvelupyyntöjä ja tiedostokuvaajaa, ts. “int fd”, tavupohjaisesti
  - Korkealla tasolla: Käyttämällä FILE-osoitinta ja siihen liittyviä monipuolisia funktioita

Alhaisen tason funktiot

- Tiedostojen käsittelyyn riittää periaatteessa viisi perusfunktiota: *open, read, write, lseek* ja *close*
- Näillä tehdään ns. puskuroimatonta siirräntää (*unbuffered I/O)*
- Kukin funktiokutsu synnyttää suoraan KJ:n palvelupyynnön ja siis saa aikaan palvelupyyntökeskeytyksen
- Nämä kutsut käsittelevät tiedostokuvaajaa (se *int fd*), jonka KJ liittää prosessiin
- Tiedostokuvaajien yläraja on OPEN_MAX (Linux: 256)
- Tiedostonimen pituuden yläraja on NAME_MAX (Linux: 255)
- Uudella prosessilla on valmiiksi kolme tiedostokuvaajaa:
  - STDIN_FILENO (tyypillisesti numeroarvo 0)
  - STDOUT_FILENO (tyypillisesti numeroarvo 1)
  - STDERR_FILENO (tyypillisesti numeroarvo 2)
- Katso */dev/fd*

Alhaisen tason funktiot

- Tiedostokuvaaja on osoitin prosessin avoimien tiedostojen tauluun
- Tässä taulussa on tyypillisesti
  - Lipukkeita (FD_CLOEXEC, O_SYNC, O_APPEND, jne.)
  - Tiedosto-osoitin, joka osoittaa kohtaan tiedostossa, josta seuraavaksi luetaan tai johon seuraavaksi kirjoitetaan
  - Viite itse tiedoston tietorakenteeseen (*v*-node tai *i-node*)
- Lukeminen ja kirjoittaminen on tehokkainta, jos operaatiot tapahtuvat levylohkojen kokoisina kutsuina
- Levylohkon koko löytyy *stat*-funktiolla ja tietorakenteen *struct stat* muuttujasta *st_blksize* (katso hieman aiemmat kalvot)
- Kirjoituksessa KJ tyypillisesti tekee jotain omaa puskurointiaan, tiedon voi pakottaa levylle kutsuilla *sync, fsync* tai *fdatasync*

Alhaisen tason funktiot

- *write*- ja *read*-funktiot kirjoittavat tai lukevat tiedosto-osoittimen osoittamasta kohdasta
- Lipukkeella *O_CREAT* suoritetaan atomisesti tiedoston luonti ja avaaminen, jos tiedostoa ei ollut
- Mitä tapahtuu, jos prosessin kaksi säiettä lukee samaa tiedostoa?
  - Jos ohjelma haluaisi ensin siirtää tiedosto-osoitinta ja sitten kirjoittaa tähän uuteen paikkaa, perustilanteessa tämä toiminta vaatii kaksi eri funktiokutsua eikä siis ole atominen.
  - Voi käydä seuraavasti:
    - 1. säie A tekee lseek(): tiedosto-osoitin siirtyy
    - 2. säie B tekee lseek(): tiedosto-osoitin siirtyy
    - 3. säie B lukee read(): tiedosto-osoitin siirtyy luetun verran
    - 4. säie A lukee read(): lukupositio on 2 * lseek + read

Alhaisen tason funktiot

- Tähän ongelmaan on kuitenkin kolme ratkaisua
- Jos ohjelma haluaa kirjoittaa tiedoston loppuun, asettamalla ”O_APPEND” lipukkeen, KJ pitää huolen, että kirjoitus tapahtuu aina oikeaan paikkaan
- Atominen lukeminen/kirjoittaminen tietystä paikasta:

  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)
- Operaatiot suorittavat lukemisen tai kirjoittamisen tietystä paikasta
- Huom. tiedosto-osoittimen paikka ei muutu
- Pread ja pwrite ovat nopeampia kuin vastaava “lseek + read/write”: miksi?

Ylemmän tason kutsut: standard I/O kirjasto (stdio)

- Ns. *standard I/O*-kirjaston kutsut peittävät alleen osan siirrännän detaljeista, kuten puskurien allokoinnin ja oikean koon määrittelyn sekä todelliset siirrännät levyn kanssa
- stdio-kirjaston kirjoitti alun perin Dennis Ritchie n. vuonna 1975
- stdio-kirjaston keskeinen elementti on ns. s*tream*
- Kaikki siirräntä tapahtuu näiden ”virtojen” kautta
- Kun alhaisen tason funktiolla siirrettiin puhtaasti tavuja, stdio-kirjastolla voi myös käsitellä kansainvälisiä merkistöjä, joissa kirjain koodataan useammalla kuin yhdellä tavulla
- Kirjasto ei välttämättä ole käytännössä tehokas, joustava kylläkin on

stdio-kirjastosta

- stdio-kirjaston idea on minimoida todellisten read- ja write-kutsujen määrä
- Tämä tapahtuu puskuroimalla tietoa, yleensä kolmella tavalla:
  - Täysin puskuroitu (*fully buffered*) (Linux 8192 tavua)
  - Rivi kerrallaan (*line buffered*)
  - Puskuroimaton (*unbuffered*)
- Lukeminen ja kirjoittaminen voi tapahtua merkki, rivi tai objekti kerrallaan
- Merkki kerrallaan: *getc, fgetc, getchar, ungetc, putc, fputc, putchar*
- Rivi kerrallaan: *fgets, gets, fputs, puts*
- Objekteja:

size_t *fread*(**void** *ptr, **size_t** size, **size_t** nobj, **FILE** *fp)

size_t *fwrite*(**void** *ptr, **size_t** size, **size_t** nobj, **FILE** *fp)

FILE-tietorakenne (libio.h)

```
struct _IO_FILE {
  int _flags; /* High-order word is _IO_MAGIC; rest isflags. */

#define _IO_file_flags _flags

/*The following pointers correspond to the C++ streambuf protocol. */
/* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields
   directly.*/

  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */

/* The following fields are used to support backing up and undo. */

  char *_IO_save_base; /* Pointer to start of non-current get area.*/
  char *_IO_backup_base;  /* Pointer to first valid character of
   backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

FILE-tietorakenne (libio.h)

```
  struct _IO_marker *_markers;
  struct _IO_FILE *_chain;

  int _fileno;

#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too
   small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

FILE-tietorakenne (libio.h)

```
#ifdef _IO_USE_OLD_IO_FILE
};

struct _IO_FILE_complete
{
  struct _IO_FILE _file;

#endif

#if defined _G_IO_IO_FILE_VERSION && _G_IO_IO_FILE_VERSION == 0x20001
  _IO_off64_t _offset;
# if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
  /* Wide character stream stuff.  */
  struct _IO_codecvt *_codecvt;
  struct _IO_wide_data *_wide_data;
# else
  void *__pad1;
  void *__pad2;
# endif
  int _mode;
  /* Make sure we don't get into trouble again.  */
  char _unused2[15 * sizeof (int) - 2 * sizeof (void *)];
#endif
};
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki: ftest.c

```
#include <stdio.h>

/*
   Käsitellään tiedostoa samaan aikaan int fd ja FILE *
   tyyppisten muuttjien kautta.
*/

int main(int argc, char *argv[])
{
    int fd, fd2, merkki;
    FILE *fi = fopen(argv[1],"r");
    if(fi == NULL) { printf("Usage: %s <file>\n",argv[0]); return 0;}
    fd=fileno(fi);
    fd2=fi->_fileno;
    merkki=fgetc(fi);    /* Tämä lukupyyntö on tarpeen,
                            sillä FILE-osoittimen taustalla
                            olevaa osoitinta ja puskureita ei
                            alusteta ennen kuin ensimmäisellä
                            lukupyynnöllä. */

                         /* Kokeile mitä tapahtuu, jos kommentoit
                            tuon fgetc-rivin pois... */
    printf("%d %d\n",fd,fd2);
    printf("%s\n",fi->_IO_read_ptr);
    return 0;
}
```

Jotain funktioita

- int fwide(FILE *fp, int mode)
- void setbuf(FILE *fp, char *buf)
- int setvbuf(FILE *fp, char *buf, int mode, size_t size)
- int fflush(FILE *fp)
- FILE *fopen(char *pathname, char *type)
- FILE *freopen(char *pathname, char *type, FILE *fp)
- int fclose(FILE *fp)
- FILE * fdopen(int fd, char *type)
- int ferror(FILE *fp), int feof(FILE *fp), void clearerr(FILE *fp)
- long ftell(FILE *fp)
- int fseek(FILE *fp, long offset, int whence)
- void rewind(FILE *fp)
- int fileno(FILE *fp)
- fprintf, fscanf, tmpnam, tmpfile, ...

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Signaalit

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Sisällys

- Signaalin olemus
- Signaalin lähettäminen
- Signaaling odottaminen eli signaalikäsittelijän asettaminen
- Signaalijoukot
- Viritettyjen signaalien kysely
- Signaalimaskit
- Signaalien käyttö kommunikoinnissa - haitat
- Signaalijonot (Real-Time Signals Extension)

Signaali

- Signaali on ohjelmallinen keskeytys, joka virittää prosessinkuvaajassa signaalilipukkeen
- Signaali voi tulla asynkronisesti käyttöjärjestelmältä, toiselta prosessilta tai prosessilta itseltään
- Kaikille signaaleille on määritelty oletustoiminto, esim. prosessin päättyminen tai että signaalia ei huomioida mitenkään
- Prosessi voi asettaa lähes kaikille signaaleille myös oman käsittelyfunktion
- Poikkeus: signaali SIGKILL, SIGSTOP

Määritellyt signaalit (bits/signum.h)

```
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGILL          4       /* Illegal instruction (ANSI).  */
#define SIGTRAP         5       /* Trace trap (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGIOT          6       /* IOT trap (4.2 BSD).  */
#define SIGBUS          7       /* BUS error (4.2 BSD).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
#define SIGPIPE         13      /* Broken pipe (POSIX).  */
#define SIGALRM         14      /* Alarm clock (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGSTKFLT       16      /* Stack fault.  */
#define SIGCLD          SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD         17      /* Child status has changed (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).  */
#define SIGTSTP         20      /* Keyboard stop (POSIX).  */
#define SIGTTIN         21      /* Background read from tty (POSIX).  */
#define SIGTTOU         22      /* Background write to tty (POSIX).  */
#define SIGURG          23      /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU         24      /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ         25      /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM       26      /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF         27      /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH        28      /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL         SIGIO   /* Pollable event occurred (System V).  */
#define SIGIO           29      /* I/O now possible (4.2 BSD).  */
#define SIGPWR          30      /* Power failure restart (System V).  */
#define SIGSYS          31      /* Bad system call.  */
#define SIGUNUSED       31
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Signaalin lähettäminen

- Signaalin voi lähettää prosessille funktiolla:
  int *kill*(**pid_t** pid, **int** signo)
  - pid > 0 : prosessille pid
  - pid == 0 : samaan prosessiryhmään kuuluville
  - pid < 0 : prosessiryhmään |pid| kuuluville prosesseille
- Käyttäjän prosessi voi lähettää signaaleja vain prosesseille, joiden uid tai euid on sama kuin itsellä
- Super-user voi lähettää signaaleja rajoituksetta
- Prosessi voi pyytää signaalin SIGABRT itselleen funktiolla
  void *abort*(**void**): ei voi hylätä, mutta voidaan asettaa oma käsittelijä

Signaalin tilaaminen ja odottaminen

- Prosessi voi tilata signaalin SIGALRM funktiolla
  unsigned int *alarm*(**unsigned int** seconds)
- Käyttöjärjestelmä lähettää signaalin SIGALRM prosessille aikaisintaan seconds sekunnin kuluttua
- Funktio palauttaa edellisestä ajastuksesta jäljellä olevan ajan, edellinen ajastin korvataan uudelle
- Jos prosessi ei hylkää tai käsittele tätä tilaamaansa signaalia, sen suoritus päättyy
- *alarm*()-kutsu kumoaa edellisen tilauksen
- Signaalia voi pysähtyä odottamaan funktiolla:
  int *pause*(**void**)
- Prosessi herää, kun se saa minkä tahansa signaalin
- Signaali käsitellään normaalisti käsittelijäasetusten mukaan

Toiminta signaalin saapuessa

- Signaali voi olla:
  - estetty (masked, blocked), jolloin lipuke prosessinkuvaajassa virittyy, mutta signaalia ei toimiteta prosessille (signal pending)
  - sallittu (enabled), jolloin lipuke virittyy ja käsittely mahd. pian
- Kun signaali käsitellään, lipuke laukeaa ja signaali voidaan:
  - unohtaa (SIG_IGN)
  - hoitaa oletuksen mukaisesti (SIG_DFL),esimerkiksi unohtaa tai lopettaa prosessi
  - käsitellä prosessin omassa funktiossa

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Signaalikäsittelijän asettaminen

- Signaalinkäsittelijä asetetaan funktiolla
  int *sigaction*(**int** signum, **const struct sigaction** *act, **struct sigaction** *oldact);

```
struct sigaction {
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restorer)(void); /* ei UNIX03 */
}
```

- Älä aseta sekä sa_handler että sa_sigaction funktioita, koska joissain arkkitehtuureissa rakenne on union.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

sigaction parametreistä

- *signum* = signaali, jonka käsittelijää kysellään tai jolle asetetaan käsittelijä.
- *act*: uusi toiminta tai NULL, jos kysytään vanhoja asetuksia
- *oact*: vanha toiminta tai NULL, jos vanhoja arvoja ei haluta ottaa talteen
- *sa_handler*
  - SIG_IGN unohda signaali
  - SIG_DFL toimi oletuksen mukaan
  - funktio() kutsuttava käsittelijäfunktio
- *sa_mask* = Käsittelyn ajaksi voidaan estää muita signaaleja (huom. SA_NODEFER)

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Lipukkeita

- sa_flags-kenttä:
  - SA_RESTART signaali saa katkaista systeemikutsuja
  - SA_RESETHAND signaali käsitellään asetetussa käsittelijässä, jonka jälkeen sa_hander <- SIG_DFL
  - SA_ONSTACK käsittelijäfunktio käyttää omaa erillistä pinoa
  - SA_NOCHLDSTOP signaali SIGCHLD lähetetään äitiprosessille vain prosessin päättyessä, ei sen pysähtyessä
  - SA_SIGINFO signaali vastaanottaa 3 parametria eikä yhtä. Tähän ominaisuuteen palataan myöhemmin.

Esimerkki signaaleista (signal1.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
#include<string.h>
#include"mydbg.h"

static void sig_usr(int signo)
{
  if (signo == SIGUSR1)        printf("received SIGUSR1\n");
  else if (signo == SIGUSR2)   printf("received SIGUSR2\n");
  else                         printf("received signal
                                        %d\n",signo);
  return;
}

int  main(void)
{
  struct sigaction sig;
  sigemptyset(&sig.sa_mask);
  sig.sa_flags=0;
  sig.sa_handler = sig_usr;

  CHECK((sigaction(SIGUSR1,&sig,NULL)) == 0);
  CHECK((sigaction(SIGUSR2,&sig,NULL)) == 0);

  for ( ; ; )
    pause();
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Signaalit ja systeemikutsut

- Signaali voi katkaista hitaan systeemikutsun suorituksen. Hitaita systeemikutsuja:
  - read(), write(): 'hitaille' tiedostoille (pääte, putki, verkkoyhteydet). Levytiedostoa käsittelevä kutsu ei voi katketa.
  - wait(), pause(): voivat katketa
  - sleep(): voi katketa (SysV), ei voi katketa (BSD)
  - Eräät ioctl() kutsut
  - Eräät prosessien välisen kommunikoinnin systeemikutsut
- Huom: Signaali voi tulla minkä tahansa kahden käskyn välillä.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Funktion suorituksen jatkaminen

- Kesken jääneen funktion uudelleenkutsuminen voi aiheuttaa ongelmia, jos funktio ei ole vapaakäyntinen (*re-entrant*), ts. ellei se ole toteutettu siten, että useita sen suorituksia voi olla meneillään yhtä aikaa.
- Ohjelman omia muuttujia ja omia vapaakäyntisiä funktioita saa käyttää vapaasti
- Ongelmia aiheuttavat:
  - malloc() ja free() sekä kaikki niitä käyttävät funktiot
  - funktiot, jotka tallettavat tuloksen staattiseen muistitilaan (esim. getpwent())
  - stdio-kirjastoon kuuluvat funktiot (osa)

Mitä koodia signaalinkäsittelijään ?

- Ohje: mahdollisimman vähän koodia käsittelijään. Vain lipuke pystyyn ja käsittely 'normaalissa' koodissa.
- Standardeista löytyy lista turvallisista funktioista, joita voi kutsua signaalinkäsittelijästä.

```
static volatile sig_atomic_t mysignow = 0;

int my_sighandler(int signo)
{
    mysignow = signo;
}
```

- Katso esimerkki *signal1-parempi.c*

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Signaalijoukot

- Signaalijoukko on bittimaski, jonka avulla voi estää tai sallia yhdellä kertaa suuren joukon signaaleja.
- Signaalijoukkoa käsitellään funktioilla
  int *sigemptyset*(**sigset_t** *set): luo tyhjän joukon
  int *sigfillset*(**sigset_t** *set): täyttää joukon
  int *sigaddset*(**sigset_t** *set, **int** signo): lisää joukkoon signaalin
  int *sigdelset*(**sigset_t** *set, **int** signo): poistaa joukosta signaalin
  int *sigismember*(**sigset** *set, **int** signo): tarkistaa onko kyseinen signaali asetettu joukkoon

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Signaalijoukon kysely/asettaminen

- Prosessin kuvaajassa on bittimaski, josta käy ilmi mitkä signaalit on toimitettava prosessille käsiteltäviksi ja mitkä signaalit vain viritetään (ts. merkitään tulleiksi).
- Maskin voi kysyä ja asettaa funktiolla:
  int *sicprocmask*(**int** how, **const sigset_**t set, **sigset_t** *oldset);

```
Jos oldset != NULL niin
   oldset = nykyinen maski
Jos set != NULL niin
   jos how == SIG_BLOCK niin
                maski = maski + set
   jos how == SIG_UNBLOCK niin
                maski = maski - set
   jos how == SIG_SETMASK niin
                maski = set
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki: signaalijoukon kysely (siggroup.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<unistd.h>
#include"mydbg.h"

int main(void)
{
  sigset_t    sigset;

  CHECK((sigprocmask(SIG_UNBLOCK, NULL, &sigset)) == 0);

  if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
  if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
  if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
  if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");

  /* remaining signals can go here */

  printf("\n");
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki: signaalien estäminen ja kysely (signal2.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<unistd.h>
#include<string.h>
#include"mydbg.h"

static void sig_quit(int);
struct sigaction sig;

int main(void)
{
  sigset_t newmask, oldmask, pendmask;

  sigemptyset(&sig.sa_mask);
  sig.sa_flags=0;
  sig.sa_handler = sig_quit;

  CHECK((sigaction(SIGQUIT,&sig,NULL)) == 0);
  sigemptyset(&newmask);
  sigaddset(&newmask, SIGQUIT);
  CHECK((sigprocmask(SIG_BLOCK,&newmask,&oldmask)) == 0);
  sleep(15);    /* SIGQUIT here will remain pending */
  CHECK((sigpending(&pendmask)) == 0);

  if (sigismember(&pendmask, SIGQUIT))
    printf("\nSIGQUIT pending\n");

  CHECK((sigprocmask(SIG_SETMASK,&oldmask,NULL)) == 0);
  printf("SIGQUIT unblocked\n");
  sleep(5);    /* SIGQUIT here will terminate with core file */
  exit(0);
}
```

Esimerkin jatkoa

```
static void sig_quit(int signo)
{
  printf("caught SIGQUIT\n");

  sig.sa_handler = SIG_DFL;

  CHECK((sigaction(SIGQUIT,&sig,NULL)) == 0);

  return;
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Huomioita

- Viritetyt signaalit voi selvittää funktiolla:
  int *sigpending*(**sigset_t** *set)
- Prosessi saa tiedon siitä, että signaali on tullut, mutta ei sitä kuinka monta kertaa se on tullut
- Esimerkki: tehdään jotain kriittistä ja sen jälkeen odotetaan SIGINT-signaalia ennen kuin jatketaan
- Funktion pause() käyttö ei aina riitä

```
sigset_t newmask, oldmask;
sigemptyset(&newmask);
sigaddset(&newmask,SIGINT);

CHECK((sigprocmask(SIG_BLOCK,&newmask,&oldmask)) == 0);
/* kriittinen osa koodia alkaa */
 ...
 /* Kriittinen osa loppuu */
CHECK((sigprocmask(SIG_SETMASK,&oldmask,NULL)) == 0);

pause();
      /* jatka prosessointia */
```

Maskin palauttaminen atomisesti

- Vanhan maskin palauttamisen ja pause()-kutsun välissä tuleva SIGINT jää huomaamatta
- Funktiolla:

  int *sigsuspend*(**const sigset_t** *mask)

  signaalimaskin asettaminen ja prosessin nukuttaminen tapahtuvat atomisesti.
- *sisgsuspend*()-kutsussa annettu maski on voimassa vain ko. funktion aikana
- Kun funktio palaa, asettuu kutsua edeltänyt maski takaisin
- Oikea ratkaisu eo. tapauksessa on siis

Ratkaisu

```
sigset_t newmask, oldmask, zeromask;
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask,SIGINT);

CHECK((sigprocmask(SIG_BLOCK,&newmask,&oldmask))
== 0);
/* kriittinen osa koodia alkaa */
...
/* kriittinen osa koodia loppuu */
CHECK((sigsuspend(&zeromask))  == 0);

/* jatka prosessointia */
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Sovelluksen yleiset signaalit (handle_signals.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<signal.h>
#include"mydbg.h"

void handle_signals(int (*handler)(int))
{
  sigset_t set;
  struct sigaction act;

  CHECK((sigfillset(&set)) == 0);
  CHECK((sigprocmask(SIG_SETMASK,&set,NULL)) == 0);
  act.sa_flags=0;
  CHECK((sigfillset(&act.sa_mask)) == 0);
  act.sa_handler = SIG_IGN;
  CHECK((sigaction(SIGHUP,&act,NULL)) == 0);
  CHECK((sigaction(SIGINT,&act,NULL)) == 0);
  CHECK((sigaction(SIGQUIT,&act,NULL)) == 0);
  CHECK((sigaction(SIGPIPE,&act,NULL)) == 0);
  act.sa_handler = handler;
  CHECK((sigaction(SIGTERM,&act,NULL)) == 0);
  CHECK((sigaction(SIGBUS,&act,NULL)) == 0);
  CHECK((sigaction(SIGFPE,&act,NULL)) == 0);
  CHECK((sigaction(SIGILL,&act,NULL)) == 0);
  CHECK((sigaction(SIGSEGV,&act,NULL)) == 0);
  CHECK((sigaction(SIGSYS,&act,NULL)) == 0);
  CHECK((sigaction(SIGXCPU,&act,NULL)) == 0);
  CHECK((sigaction(SIGXFSZ,&act,NULL)) == 0);
  CHECK((sigemptyset(&set)) == 0);
  CHECK((sigprocmask(SIG_SETMASK,&set,NULL)) == 0);
}
```

Signaalien käyttö kommunikointiin - haitat

- Sovelluksen käyttöön soveltuvia vapaita signaaleja ei ole riittävästi (käytännössä vain SIGUSR1 ja SIGUSR2)
- Signaalit eivät ole luotettava kommunikointitapa
  - Oletetaan että sovellus käynnistää 5 kyselyä ja jää odottamaan jokaiselta signaalia kyselyn päättymisestä
  - Pahimmassa tapauksessa sovellus vastaanottaa vastauksista vain osan
  - Signaalin katoaminen on mahdollista silloin kun kaksi samaa signaalia saapuu prosessille niin nopeasti ettei ensimmäistä ehditä prosessoida
  - Ongelma johtuu siitä että kaikki järjestelmät eivät toteuta signaalien jonotusta

Haittojen jatkoa

- Signaalien toimituksella ei ole järjestystä
  - Oletetaan että sovellus vastaanottaa useita eri signaaleja ennenkuin yhdenkään signaalin signaalinkäsittelijää kutsutaan
  - Standardi ei määrittele missä järjestyksessä signaalit toimitetaan prosessille
  - Sovelluksen kannalta olisi hyödyllistä jos pystyisi määrittelemään tietyt signaalit korkeammalle prioriteetille
- Signaalien informaatioarvo
  - Sovellus saa tietoonsa vain minkä signaalin vastaanotti
  - Sovellus hyötyisi esimerkiksi tiedoista: miksi signaali tuli, keneltä se tuli, jne

Haittojen jatkoa

- Asynkronisyys
  - Signaali voi saapua prosessille missä tahansa suorituksen vaiheessa
  - Ohjelmassa pitää siis varautua signaalin saapumiseen kaikessa koodissa
  - Tämä lisää ohjelman monimutkaisuutta

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

POSIX.4 Real-Time Signals Extension (RTS)

- Lisää ohjelmassa käytettävissä olevia signaaleja
- Sallii signaalien jonotuksen siten että signaalit eivät katoa vaikka sama signaali on jo odottamassa
- Määrittelee signaalien välitysjärjestyksen
- Määrittelee lisää informaatiota, jota signaali toimittaa:
  - Kuka lähetti
  - Miksi lähetti
  - Mahdollisuus omaan dataan

RTS Signaalit

- RTS standardi määrittelee joukon signaaleja, joiden numerot ovat välillä SIGRTMIN ja SIGRTMAX
- Jos järjestelmä toteuttaa standardin on vähintään RTSIG_MAX signaalia käytettävissä
- Järjestelmässä toteutetun määrän saa selville käyttämällä *sysconf* kutsua
- Linuxissa arvo on 32

Signaalijonot

- Normaalisti ei voi luottaa siihen mitä tapahtuu, jos signaali saapuu ja sama signaali on jo odottamassa käsittelyä
- RTS määrittelee kuitenkin signaalijonon, jonka minipituus on vähintään 32, todellisen arvon voi selvittää *sysconf* funktiolla
- Signaali asetetaan jonoon funktiolla
  int *sigqueue*(**pid_t** pid, **int** signum, **const union sigval** value);
- Palautttaa 0 onnistuessaan, muuten -1 ja asettaa errno-arvon

Toteuttaako järjestelmä RTS:n (rtsok.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include"mydbg.h"

int main(void)
{
    int val;

    CHECK((val = sysconf(_SC_REALTIME_SIGNALS)) != -1);

    if ( val > 0)
        printf("Realtime signals found\n");
    else
        printf("No realtime signals found\n");

    CHECK((val = sysconf(_SC_RTSIG_MAX)) != -1);

    if ( val > 0)
        printf("Realtime signals max %d\n",val);
    else
        printf("No real-time signals\n");

    exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

RTS signaalinkäsittelijät

- Standardi ei kerro voiko uusia ominaisuuksia käyttää vanhoille signaaleille vaikka monissa järjestelmissä niin voi
- RTS signaalinkäsittelijän saa määriteltyä *sigaction* funktiossa antamalla parametrina SA_SIGINFO lipuke struct sicaction kentässä sa_flags
- Tällöin voi käyttää sa_sigaction kenttää sa_handler kentän sijasta
- Signaalinkäsittelinjän prototyyppi muuttuu tällöin muotoon:

  void *rts_handler*(**int** signum, **siginfo_t** *info,
  void *context);

siginfo_t

```
typedef struct {
        int      si_signo;  /* Signal number */
        int      si_errno;  /* An errno value */
        int      si_code;   /* Signal code */
        pid_t    si_pid;    /* Sending process ID */
        uid_t    si_uid;    /* Real user ID of sending process */
        int      si_status; /* Exit value or signal */
        clock_t  si_utime;  /* User time consumed */
        clock_t  si_stime;  /* System time consumed */
        sigval_t si_value;  /* Signal value */
        int      si_int;    /* POSIX.1b signal */
        void *   si_ptr;    /* POSIX.1b signal */
        void *   si_addr;   /* Memory location which caused fault */
        int      si_band;   /* Band event */
        int      si_fd;     /* File descriptor */
} siginfo_t;
```

Esimerkki (rtssignal.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
#include<string.h>
#include"mydbg.h"

int main(void) {
  struct sigaction act;
  union sigval val;

  sigemptyset(&act.sa_mask);
  act.sa_flags = SA_SIGINFO;
  act.sa_sigaction = rts_handler;
  CHECK((sigaction(SIGUSR1, &act, NULL)) == 0 );
  CHECK((sigaction(SIGRTMIN, &act, NULL)) == 0 );
  CHECK((kill(getpid(), SIGUSR1)) == 0 );
  val.sival_int = 1234;
  CHECK((sigqueue(getpid(), SIGRTMIN, val)) == 0 );
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkin jatkoa

```
static void rts_handler(int signum, siginfo_t *info, void *context)
{
    printf("signal number: %d\n", info->si_signo);
    printf("sending process ID: %ld\n", (long)info->si_pid);
    printf("real user ID of sending process: %ld\n", (long)
             info->si_uid);
    switch (info->si_code) {
      case SI_USER:
          printf("Signal from user\n"); break;
      case SI_QUEUE:
          printf("Signal from sigqueue; value = %d\n",
                  info->si_value.sival_int); break;
      case SI_TIMER:
          printf("Signal from timer expiration; value = %d\n",
                  info->si_value.sival_int); break;
      case SI_ASYNCIO:
          printf("Signal from asynchronous I/O completion; value =
    %d\n",
                  info->si_value.sival_int); break;
      case SI_MESGQ:
          printf("Signal from message arrival; value = %d\n",
                  info->si_value.sival_int); break;
      default:
          printf("Other signal\n"); break;
    }
}
```

RTS signaalin odottaminen

- Signaalia voi odottaa funktiolla:
  int *sigwaitinfo*(**const sigset_t*** set,**siginfo_t** *info)
  - Parametrinä annetaan odotettavat signaalit ja paluuarvona saadaan signaali, joka päätti funktion
  - info kenttään tallettuu mahdollinen lisäinformaatio
- Signaalia voi odottaa myös tietyn ajan funktiolla:
  int *sigtimedwait*(**const sigset_t** *set,**siginfo_t** *info,**const struct timespec** *ts);
  - Palauttaa -1 ja errno==EAGAIN jos aikaraja täyttyy ennen signaalia
  - Muuten palauttaa saadun signaalin numeron

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Prosessienhallinnan perusteet

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Sisältö

- Prosessin luonti ja päättyminen
- Prosessin koodin vaihto
- Prosessin ominaisuudet ja niiden tutkiminen
- Toisen ohjelman kutsuminen
- daemon-ohjelmointi

Prosessinhallinnan pääsysteemikutsut

- Uuden prosessin luonti fork()
- Prosessin koodinvaihto exec()
- Prosessin lopettaminen exit()
- Prosessin päättymisen odotus wait(), waitpid()

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Prosessin luonti

- Prosessi luodaan funktiolla:
  pid_t *fork*(**void**): palauttaa kutsuvalle prosessille lapsiprosessin numeron ja lapsiprosessille 0.
- *fork*()-kutsun tuloksena syntyy uusi prosessi, joka on 'klooni' äitiprosessista.
  - yhteinen koodi
  - samat muuttujien arvot
  - samat prosessinkuvaajan perustiedot
  - yhteiset avoimet tiedostot
- Kummallakin on kuitenkin:
  - oma data-alue
  - oma prosessinkuvaaja
- fork()-kutsun jälkeen ei voi olla varma siitä kumpi prosessi (äiti vai lapsi) jatkaa aiemmin suoritustaan. Jos järjestys tärkeää, on ohjelmoitava itse synkronointi.

Esimerkki (fork.c)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include"mydbg.h"

int  glob = 6;              /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int main(void)
{
  int      var;    /* automatic variable on the stack */
  pid_t pid;

  var = 88;
  CHECK((write(STDOUT_FILENO, buf, sizeof(buf)-1)) == sizeof(buf)-1);
  printf("before fork\n");

  CHECK((pid = fork()) >= 0);

  if (pid == 0) {   /* child */
    glob++;          /* modify variables */
    var++;
  } else
    sleep(2);          /* parent */

  printf("ppid=%d,pid=%d,glob=%d,var=%d\n",getppid(),getpid(), glob, var);
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki (fork2.c)

```
/* Samaa koodi kuin fork.c:ssä */
...
int status=0;
...
/* Samaa koodi kuin fork.c:ssä */

  else          /* parent */
  {
    printf("Waiting for child %d\n",pid);

    if(waitpid(pid,&status,WUNTRACED) == -1)
    {
      perror("waitpid");
      exit(-1);
    }

    if(WIFEXITED(status) != 0)
      printf("Child exited normally\n");

    printf("Child returned: %d\n",WEXITSTATUS(status));

    if(WIFSIGNALED(status))
      printf("Child returned due to a signal which was not caucht: %d\n",
               WTERMSIG(status));
  }

  printf("ppid=%d,pid=%d,glob=%d,var=%d\n",getppid(),getpid(), glob, var);
  exit(0);
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Lapsiprosessin luonti

- Lapsiprosessi luodaan, kun:
  - halutaan suorittaa äiti- ja lapsiprosessissa erillinen osa samassa tiedostossa olevasta koodista. Esim. verkkosovelluksissa on tavallista, että palvelija luo lapsiprosessin antamaan palvelua ja jää itse odottamaan uusia palvelupyyntöjä.
  - halutaan suorittaa kokonaan toinen ohjelma. Tällöin fork()-kutsun jälkeen on lapsiprosessissa myös exec()-kutsu, eli se vaihtaa suoritettavaa koodia. Esim. komentotulkit käyttävät tätä menetelmää.

Prosessin loppuminen

- Unix-spesifisiin lopputoimiin kuuluu tiedostojen sulkeminen, muistitilan vapauttaminen sekä äitiprosessin signalointi, mutta prosessinkuvaaja jää vielä olemaan ("zombie"), koska paluuarvon välittäminen äitiprosessille ja laskutustietojen kokoaminen on vielä kesken.
- Jos äitiprosessi on päättynyt ennen lapsiprosessia, merkitsee ydin zombien äidiksi prosessin 1 (init). Se kokoaa laskutustiedot ja vapauttaa prosessinkuvaajan.

Lapsiprosessin päättyminen

- Kun lapsi prosessi päättyy, saa äiti aina signaalin SIGCHLD.
- Oletusarvo on, että äiti ei välitä tästä signaalista.
- Äiti voi pysähtyä odottamaan lapsen päättymistä funktioilla:
  pid_t *wait(***int** *status*)*
  pid_t *waitpid(***pid_t** pid*, **int** *status*, **int** options*)*
- Jos lapsiprosessi on jo päättynyt, pääsee äitiprosessi heti jatkamaan.
- Funktiolla *waitpid*() voi määrätä odotettavaksi jonkin tietyn prosessin päättymistä, kun taas funktiolla *wait*() odotetaan minkä tahansa lapsen päättymistä.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Koodinvaihto

- Prosessi voi myös suorittaa toisen ohjelman, ts. vaihtaa koodinsa
- Tämä tapahtuu funktiolla **int** *exec(...)*
- Exec-funktiosta löytyy eri variaatioita tarpeen mukaan

int *execl*(const **char** *pathname, const **char** * arg0, ...,NULL

int *execv*(const **char** *pathname, const **char** *argv[])

int *execle*(const **char** *pathname, const **char** *arg0, ...
NULL, const **char** *envp[])

int *execve*(const **char** *pathname, const **char** *argv[], const
char *envp[])

int *execlp*(const **char** *filename, const **char** *arg0,...,NULL)

int *execvp*(const **char** *filename, const **char** *argv[])

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Koodinvaihto

- Koodia voi etsiä polkunimen mukaan kutsuilla *execl(), execv(), excele()* ja *execve()*
- Tiedostonimen mukaan ympäristömuuttujan PATH määrittelemistä hakemistoista kutsuilla *execlp()* ja *execvp()*
- Koodille voi välittää argumentit joko listana (*l*) tai vektorina (*v*), vrt. *execl()* vs. *execv()*
- Koodille voi myös antaa haluamansa ympäristömuuttujat vektorina kutsuissa *execle()* ja *execve()*
- Äidin ympäristömuuttujat periytyvät lapselle sellaisenaan environ-muuttujasta

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Koodinvaihto

- Kaikki edelliset funktiot kutsuvat funktiota:
  int *execve*(const **char** *filename, const **char** *argv [], **char** *const envp[])
- Tässä *filename* pitää osoittaa ohjelmaan tai shell-skriptiin
- Ensimmäinen argumentti on aina ohjelman nimi
- Signaalit nollataan
- Onnistuessaan, kutsu ei palauta paluuarvoa, kun ei ole tarkoitus olla ketään, jolle palauttaa
- Virhetilanteessa paluuarvo on -1 ja *errno* on asetettu

Koodinvaihto

- Käynnistyvä prosessi perii mm. kutsuvan prosessin tunnuksen (process ID) ja avoimet tiedostokuvaajat
- *fcntl*-funktiokutsulla voidaan määritellä jokaiselle tiedostokuvaajalle, jätetäänkö kuvaaja auki vai suljetaanko se *exec*-kutsun yhteydessä
- Katso esimerkki *open-exec1.c*, komentorivi-parametreista riippuen:
  - Uusi ohjelma lukee aiemmin avatusta tiedostokuvaajasta ilman omaa *open*-kutsua
  - Uusi ohjelma lukee omasta *stdin* virrastaan, joka viittaa edellisen ohjelman avaamaan tiedostoon

Periydyvät tiedot

OMINAISUUS	*fork()*	*exec()*	KYSELY	ASETTAMINEN
prosessinumero pid	ei	+	*getpid()*	
vanhemman nro ppid	ei	+	*getppid()*	
prosessiryhmä	+	+	*getpgrp()*	*setpgrp()*
prosessin istunto	+	+	*/dev/tty*	*setsid()*
hallintapääte	+	+	*/dev/tty*	*setsid()*
omistajan numero uid euid	+ +	+ + / ei	*getuid()* *geteuid()*	*setreuid()* *setreuid()*
yhmänumero gid egid	+ +	+ + / ei	*getgid()* *getegid()*	*setregid()* *setgregid()*
muut ryhmänumerot	+	+	*getgroups()*	*setgroups()*
komentoriviargumenti t	+	ei	*argc, argv[]*	
ympäristömuuttujat	+	+ / ei	*getenv(), environ*	*putenv()*
resurssirajat	+	+	*getrlimit()*	*setrlimit()*
resurssien käyttö mm. utime, stime	ei	+	*getrusage()*	
prioriteetti	+	+	*getpriority()*	*setpriority(), nice()*
työhakemisto	+	+	*getcwd()*	*chdir()*
juurihakemisto	+	+	*hakemisto /*	*chroot()*
käyttöoikeusmaski	+	+	*umask()*	*umask()*
tiedostokuvaajat	+	+ / ei	*fstat()*	*open(), dup(), pipe()*
tiedostolukot	ei	+	*fcntl()*	*fcntl()*
muistikuvaukset	+	ei		*mmap(), shmat()*
reaaliaika-ajastin	ei	+	*getitimer()*	*setitimer(), alarm()*
muut ajastimet	+	+	*getitimer()*	*setitimer(), alarm()*
signaalimaski	+	+	*sigprocmask()*	*sigprocmask()*
signaalin käsittelijät	+	ei	*sigaction()*	*sigaction()*
odottavat signaalit	ei	+	*sigpending()*	*kill(), killpg(), ...*

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki (exec1.c)

```
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>

int main(int argc, char *argv[])
{
        if(argc == 1)
        {
                printf("Usage: %s command par1 par2 ...\n",argv[0]);
                return 0;
        }

        printf("**********Suorittamassa exec-kutsun...\n");

        /* specify pathname, specify environment */

        if (execv(argv[1],&argv[1]) < 0)
                perror("execle error");

        else printf(“tänne ei koskaan päästä\n”);


        return 0;
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki (exec2.c)

```
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>

int main(int argc, char *argv[])
{
    pid_t pid;
    if(argc == 1)
    {
        printf("Usage: %s command param1 param2 ...\n",argv[0]);
        return 0;
    }
    printf("**********Suorittamassa exec-kutsun...\n");
    if ( (pid = fork()) < 0)
        perror("fork error");
    else if (pid == 0) /* This is the child */
    {
        if (execvp(argv[1],&argv[1]) < 0)
          perror("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        perror("wait error");

    printf("**********exec suoritettu\n");
    return 0;
}
```

Toisen ohjelman kutsuminen

- Funktiolla:
  int *system*(**char** *string) voi suorittaa toisen ohjelman ilman suoranaista koodin vaihtoa
- Kutsuva ohjelma jatkaa suoritustaan, kun system-kutsu palaa
- Funktio kutsuu komentotulkkia “sh” ja antaa tälle ohjelman “string” parametrina
- Paluuarvo on suoritetun ohjelman paluuarvo, mutta muoto seuraa wait-funktiota, esim. paluuarvon saa selville makrolla *WEXITSTATUS(***int** *status)*
- Virhetilanteessa paluuarvo on -1

Esimerkki (system.c)

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<signal.h>
#include<sys/wait.h>

int main(int argc, char *argv[])
{
   int i;
   int ret;

   for (i = 1; i < argc; i++)
   {               /* once for each command-line arg */
        ret=system(argv[i]);

        printf("Paluuarvo: %d (ret %d)\n",WEXITSTATUS(ret),ret);

        if(WIFSIGNALED(ret) &&
          (WTERMSIG(ret) == SIGINT || WTERMSIG(ret) == SIGQUIT))
        {
                printf("painettu CTRL-C tai QUIT\n");
        }
   }
    return 0;
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

System-kutsu exec-kutsuna?

- Voiko system-kutsun toteuttaa käyttäen exec-kutsua?
- Entä päinvastoin?

Esimerkki: system-kutsu exec-kutsun avulla

```
int ret;
char *komento;
char params[]; /[0]=”sh”, [1]=komento, [2]=eka parametri, jne...*/

pid=fork();
if(pid==0) /* child */
{
   ret=exec(komento,params);

   return ret;
}

wait(&ret); /* odotetaan lapsen poistumista */
...
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki: exec-kutsu system-kutsun avulla

```
Mitään ei pitäisi palautua kutsujalle.

...

Kopioidaan params[] yhdeksi merkkijonoksi param

...

/* sitten: */

ret=system(param);

return WEXITSTATUS(ret);

...


/* tai suoraan */

return WEXITSTATUS(system(params));
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Daemon-ohjelmointi

- ns. “daemon” on taustalla toimiva prosessi, joka tyypillisesti ei koskaan pääty
- Käyttöjärjestelmässä pyörii lukuisia daemon-prosesseja, jotka suorittavat erilaisia tehtäviä, esim.
  - atd: mahdollistaa komentojen suorittamisen ajatettuna
  - kjournald: ylläpitää tietoa tiedostojärjestelmän muutoksista
  - klogd: kernel logging daemon
  - lpd: tulostuksen hoitava daemon
  - sshd: ssh-palvelin
  - swapd: sivutuksen mahdollistava daemon
  - syslogd: system log-daemon, lokiin kirjoitus

Daemon-ohjelmointi

- Daemon-prosessi suoriutuu siis “piilossa” taustalla
- Sillä ei ole ohjaavaa terminaalia: ei *STDIN/OUT/ERR*
- Yksinkertainen taustakäynnistys ei riitä näille pitkäkestoisille ohjelmille: ei irroita prosessia lopullisesti terminaali-istunnosta, joka sen käynnisti
- “Temppu” tehdään siten, että käynnistetään uusi(a) lapsiprosesseja
- Äiti-prosessi lopettaa suoraan ja lapset mm. hankkivat uuden istunnon, sulkevat kaikki tiedostot ja vaihtavat työhakemistonsa
- Lisäksi uuteen daemoniin tarvittaneen jokin kommunikointiyhteys ja sen pitäisi myös pystyä kertomaan toiminnastaan
- EI UNIX03-määrittelyssä: *daemon(int,int)*

Daemon-ohjelmoinnin vaiheet

I *fork()*, jotta äiti voi suoritaa exit() kutsun ja kontrolli palaa komentorivile. Tarvitaan, koska uusi prosessi ei ole prosessiryhmän johtaja.

II setsid(), jotta päästään prosessiryhmän ja sessioryhmän johtajaksi. Koska kontrollipääte on yhdistetty sessioon, tämä uusi sessio ei vielä ole hankkinut kontrolliprosessia.

III *fork()* uudestaan, jos ei haluata edes mahdollisuutta kontrollipäätteeseen joskus tulevaisuudessa, äiti suorittaa exit():n.

IV *chdir("/")* varmistamaan ettei prosessi sido nykyistä työhakemistoaan käyttöönsä ikuisesti.

V umask(0), jotta on kontrolli kirjoittamiseen.

VI Suljetaan kaikki tiedostokuvaajat, myös STDIN, STDOUT, ja STDERR.

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki

```
int main()
{
    if (jukan_daemon() < 0)   /* alustustemput */
    {
        perror("daemon");
        exit(2);
    }
    process();  /* Varsinainen demoni */
    return 0;
}

void process()
{
  /* avaa jokin loki, jonne voit kertoa tekemisistäsi */

  /* avaa jokin paikka, josta otat vastaan tekemistä */
  /* odottele jotain tekemistä? */

  /* lopeta vain, jos niin käsketään */
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki

```
int jukan_daemon()
{
    switch (fork())  {            /* 1. fork */
        case 0:  break;
        case -1: return -1;
        default: _exit(0);          /* alkuperäinen prosessi poistuu */
    }

    if (setsid() < 0) return -1;  /* hankitaan oma istuntomme */

    switch (fork())  {       /* 2. fork */
        case 0:  break;
        case -1: return -1;
        default: _exit(0);
    }

    chdir("/");     /* vaihdetaan työhakemisto */

    closeall();     /* suljetaan kaikki tiedostot ml. STD_IN/OUT/ERR */

    open("/dev/null",O_RDWR);  /* avataan /dev/null  */

    dup(0);                /* Ohjataan STD_OUT /dev/null:iin. Miten? */
    dup(0);                /* Ohjataan STD_ERR /dev/null:iin. Miten? */

    return 0;
}
```

Monipuolista siirräntää

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Sisältö

- Estymätön siirräntä
- Monen tiedostokuvaajan käsittely rinnakkain
- Asynkroninen siirräntä
- Tiedostojen lukitseminen
- Muistiinkuvatut tiedostot

Estymätön ja asynkroninen siirräntä

- Eri palvelimet tyypillisesti lukevat useampaa tietovirtaa (putket, tietoliikennepistokkeet, tiedostot,...)
- Palvelin ei voi jäädä odottamaan, että tietovirta on valmis luettavaksi (tai kirjoitettavaksi) esimerkiksi,
  - Tiedoston avaus lopulta onnistuu (putken toinen pää avattu tai tiedosto vapautuu toiselta prosessilta)
  - Saataisiin syötettä asiakkaalta (putkesta, pistokkeesta, näppäimistöltä)
  - Tulisi tilaa kirjoittamista varten (putkeen, pistokkeeseen)
  - Joku vapauttaisi tiedostolukituksen (*fnctl*)
  - Tietyt *ioctl*-kutsut valmistuisivat
  - Prosessien välinen tiedonsiirto valmistuisi

Estymätön ja asynkroninen siirräntä

- Jos tietää etukäteen, mikä tiedostokuvaaja on valmis, voi käyttää estyviä systeemikutsuja
- Jos aktiivisuus on satunnaista, voi sovelluksen toteuttaa kahdella tavalla
  - Luo kullekin asiakkaalle oma prosessi, joka käyttää estyviä systeemikutsuja
  - Tee kaikille yhteinen prosessi, joka käyttää estymättömiä systeemikutsuja

Siirrännän eri muodot

- Siirräntää voi tehdä hyvin erilaisilla tavoilla:
  1. *Synkronoitua*: suoritettu, kun fyysinen siirräntä on päättynyt
  2. *Synkronoimatonta*: suoritettu, kun siirräntä prosessin ja käyttöjärjestelmän puskurin välillä tapahtunut
  3. *Synkronista*: funktiokutsu palaa, kun siirräntä on tapahtunut (kuten yllä)
  4. *Asynkronista*: funktiokutsu palaa, kun siirräntä on alustettu, tulos tarkastetaan erikseen muilla kutsuilla
- Kaksi ensimmäistä viittaa siihen, miten määritellään "onnistuminen"
- Kaksi jälkimmäistä määrittelee, milloin funktiokutsu palaa, ts. mikä riittää onnistuneeseen funktiokutsuun
- Näiden lisäksi siirräntä voi olla *estyvää* tai *estymätöntä*

Synkronoitu siirräntä

- Yleensä tehtäessä write-operaatio, KJ puskuroi tietoa ja kirjoittaa sen vasta myöhemmin
- Voidaan kuitenkin määritellä, että halutaan siirräntä tapahtuvaksi heti, esimerkiksi *open*-kutsussa:

```
fd=open("tiedosto",O_WRONLY,O_SYNC)
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Estymätön siirräntä

- Toistaiseksi kaikki siirräntä on oletettu tapahtuvan estyvänä, ts. funktiokutsu palaa vasta, kun siirräntä on todella tapahtunut
- Siirräntä voidaan määritellä estymättömäksi, jolloin kutsu palauttaa virheilmoituksen, jos siirräntä aiheuttaisi prosessin pysähtymisen
- Tällöin siirrännän systeemikutsut *read* ja *write* eivät jätä prosessia odottamaan

Estymätön siirräntä

- *open*- tai *fcntl*-kutsuissa voidaan asettaa käyttötavaksi *O_NONBLOCK*
- Jos kutsua ei voi saattaa toimintoa loppuun asti, se palauttaa arvon -1 ja errno == EAGAIN
- Toimintoa voidaan kokeilla uudelleen hetken päästä
- Jatkuva “kokeilu” voi kuitenkin olla epätehokasta ja monimutkaistaa usein toteutusta
- Olisi hyvä saada vinkki, kun jotain olisi oikeasti siirrettävissä

Estymätön siirräntä

- Prosessi tarkkailtee montaa kuvaajaa, mistä tiedetään mitä voi lukea tai kirjoittaa ?
  - Yritetään lukea tai kirjoittaa vuorotellen kutakin kuvaajaa eli aktiivinen tarkkailu
  - Kysellään systeemikutsuilla *select* tai *poll* mitä kuvaajia voi käsitellä odottamatta
  - Saadaan signaali, kun on käsiteltävää
- Kahden jälkimmäisen kohdalla selvitettävä, mikä tiedostokuvaaja on tullut aktiiviseksi (ts. prosessi ei esty)

Estymätön siirräntä: select

- select-funktiokutsu on tärkeä mm. palvelimissa
  int *select*(**int** n, **fd_set** *readfds, **fd_set** *writefds, **fd_set** *exceptfds, **struct timeval** *timeout)
- Kutsussa
  - *int n:*suurimman tiedostokuvaajan numero + 1
  - fd_set readfds: lista niistä tiedostokuvaajista, joista halutaan lukea
  - fd_set writefds: lista niistä tiedostokuvaajista, joihin halutaan kirjoittaa
  - fd_set exceptfds: lista niistä tiedostokuvaajista, joiden poikkeustilanteesta ollaan kiinnostuneita
  - struct timeval *timeout: antaa maksimiajan, jonka jälkeen kutsu palaa

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Estymätön siirräntä: select

- *select*-funktiokutsu palaa heti, kun yksikin tiedostukuvaaja on valmis, ts.
  - Yhdestäkin annetuista readfds-tiedostokuvaajista voidaan lukea
  - Yhteenkin annetuista writefds-tiedostokuvaajista voidaan kirjoittaa
  - Yksikin tiedostokuvaaja on poikkeustilassa
  - Annettu aika kului umpeen
- Kun kutsu palaa, useampi tiedostokuvaaja voi olla valmis käsiteltäväksi:
  - Paluuarvo 0: aika kului umpeen
  - 1-n: 1-n tiedostokuvaajaa valmiina
  - -1: virhe ja *errno* asetettu
- Tyhjä lista kuvaajia annetaan arvona *NULL*

Estymätön siirräntä: select

- *fd_set* tietotyyppi on taulukko bittikenttiä
- Linux rajaa taulukon alkioiden lukumääräksi 1024, ts. näin monta kuvaajaa voi olla tarkkailtavana tiettyä operaatiota varten
- Tiedostokuvaajien käsittelyyn select-kutsun kanssa liittyy useampi makro-funktio
  - FD_CLR(int fd, fd_set *set): poistaa tiedostokuvaajan maskista
  - FD_ISSET(int fd, fd_set *set): palauttaa true, jos tiedotokuvaaja on asetettu maskiin, ts. valmis
  - FD_SET(int fd, fd_set *set): lisää tiedostokuvaajan maskiin
  - FD_ZERO(fd_set *set): tyhjentää maskin

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Estymätön siirräntä: select

- Kutsuun kuuluva aika määritellään tietorakenteella:

```
struct timeval {
  unsigned long tv_sec;  /* seconds */
  unsigned long tv_usec;/*microseconds*/ };
```

- Kun kutsu palaa, tietorakenteen sisältöön ei kannata luottaa
  - Esimerkiksi Linux tallettaa tuohon jäljelle jääneen ajan, mutta kaikki KJ:t eivät tuota tee
- Kutsua voi myös käyttää nukkumiseen: aseta kaikki eri fds-kentät *NULL*-arvoiksi
- Esimerkki: select.c (ja select_test)

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Estymätön siirräntä: select-kutsun logiikka

```
int fd1, fd2, fd3;
fd_set rset;
int ret,max_fd;
struct timeval timeout;
...
/* avaa kaikki tiedostokuvaajat */
/* max_fd on suurin tiedostokuvaajan numero */
...
/* sitten vaikka ikuinen silmukka */
while(1)
{
    FD_ZERO(&rset); /* tyhjennetään bittikenttä */
    FD_SET(fd1,&rset); /* Lisätään fd1 kuunneltaviin */
    FD_SET(fd2,&rset); /* Lisätään fd2 kuunneltaviin */
    FD_SET(fd3,&rset); /* Lisätään fd3 kuunneltaviin */
    timeout.tv_sec=1; timeout.tv_usec=500000; /* 1.5s */

    ret=select(max_fd+1,&rset,NULL,NULL,&timeout);
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Estymätön siirräntä: select-kutsun logiikka

```
    if(ret > 0)  /* Jokin tiedostokuvaaja laukesi? */
    {
       if(FD_ISSET(fd1,&rset))
       {
         /* lue tavara ja tee sillä jotain */
       }
       if(FD_ISSET(fd2,&rset))
       {
         /* lue tavara ja tee sillä jotain */
       }
       if(FD_ISSET(fd3,&rset))
       {
         /* lue tavara ja tee sillä jotain */
       }
    }
    else /* select laukesi aikaan tai signaaliin
    {
         /* tee jotain
    }
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Estymätön siirräntä

- Muita samanlaisia funktiokutsuja:

  int *pselect*(**int** n, **fd_set** *readfds, **fd_set** *writefds, **fd_set** *exceptfds, **const struct timespec** *timeout, **const sigset_t** *sigmask)

  - Muuten sama kuin select, mutta mahdollistaa signaalien estämisen odotuksen ajaksi: sigmask määrittelee estetyt signaalit

  int *poll*(**struct pollfd** *ufds, **unsigned int** nfds, **int** timeout)

```
struct pollfd {
    int fd;         /* file descriptor */
    short events;   /* requested events */
    short revents; /* returned events */ }
;
```

  - Parametreina annetaan lista tiedostoista ja tapahtumista, joista ollaan kiinnostuneita

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Asynkroninen siirräntä

- Asynkronisen siirrännän juoni on siinä, että prosessi voi tehdä omia töitään ja KJ ilmoittaa, kun siirräntä voi tapahtua tai on valmistunut
- Näin prosessin ei esimerkiksi tarvitse odotella *select*-kutsussa, vaan saa tietää, kun siirräntä on valmis
- Asynkronisen siirrännän toteuttamiseen on kaksi tapaa, uusi ja vanha, jotka myös eroavat toiminnallisuudessa:
  - Vanha: tiedostokuvaajaa luotaessa, esim. *open*-kutsussa annetaa O_ASYN-parametri: *open("tiedosto",O_RDWR,O_ASYNC)*
  - Uusi: käytetään uusia aio-funktiokutsuja

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Asynkroninen siirräntä

- Vanhassa tavassa prosessi asettaa signaalin-käsittelijän signaalille *SIGIO* (oletusarvo)
- Jokaiselle tiedostokuvaajalle voi pyytää asynkronista ilmoitusta kuvaajan aktivoitumisesta
- Toimii vain ns. tietovirroille, kuten pääte I/O, tietoliikennepistokkeet ja putket
- Prosessi voi sitten tehdä omia hommiaan, ja kun signaali tulee, tarkastetaan mikä tiedostokuvaaja oli tullut aktiiviseksi, ts. esim. lukeminen ei aiheuta blokkautumista
- Esimerkit: asyn_old2.c ja asyn_old.c

Esimerkki

```
static void sig_int(int signo)
{
  if (signo == SIGINT) lopeta=1;
  else if (signo == SIGIO) sigio=1;
  else printf("signaali %d\n",signo);
}

int main(...)
{
   struct sigaction sig;
   int fd, ret;

  /* create the signal handler */

   sigemptyset(&sig.sa_mask);

   sig.sa_flags=0;

   sig.sa_handler = sig_int;

   if(sigaction(SIGINT,&sig,NULL) != 0) exit(0);
   if(sigaction(SIGIO,&sig,NULL) != 0) exit(0);
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki

```
/* avaa tiedostokuvaaja */

fd = open(“jotain”, O_RDWR|O_NONBLOCK|O_ASYNC)

/* Aseta signaali asynkronisille tapahtumille */

if(fcntl(fd,F_SETSIG,SIGIO) != 0) perror("fcntl");

while(!lopeta) {
     tee_jotain();
     if(sigio){
       sigio=0;
       printf("Katsotaanpa tuliko jotain...");
       ret = read(fd,buf,BUFSIZE);
       if(ret>0){
            printf("täältä tuli jotain!\n");
            printf("  Viesti STDIN:sta: %s", buf);
       }
       else printf("virhe luettaessa?!?\n");
     }
}
close(fd);
return 0;}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Asynkroninen siirräntä

- Uusi menetelmä toimii siten, että funktiokutsu (luku- tai kirjoitus) palaa, kun siirräntä on alustettu
- Tieto siirtyy sitten ns. "taustalla" ja prosessi voi myöhemmin selvittää, miten siirrännässä kävi
- Prosessi voi pyytää signaalin, tai jopa säikeen luotavaksi, kun siirräntä on tapahtunut, tai siirrännän tilaa voi kysyä jossain vaiheessa

Asynkroninen siirräntä

- Uusi tapa käyttää seuraavaa tietorakennetta:

```
struct aiocb {
 int aio_fildes;               /* File desriptor.  */
 off_t aio_offset;             /* file offset */
 volatile void *aio_buf;       /* Location of buffer.  */
 size_t aio_nbytes;            /* Length of transfer.  */
 int aio_reqprio;              /* Request priority offset. */
 struct sigevent aio_sigevent; /* Signal number and value.*/
 int aio_lio_opcode;           /* Operation to be performed.*/
};
```

  - *aio_fildes*: tiedostokuvaaja, johon siirräntä liittyy
  - *aio_offset:* siirtymä tiedostossa (oletusarvo *0*)
  - *aio_buf*: puskuri johon/josta siirräntä tapahtuu
  - *aio_nbytes*: tavujen lukumäärä
  - *aio_reqprio*: liittyy siiräntöjen priosisointiin
  - *aio_sigevent*: tietoa signaalista, jos sellainen halutaan siirrännän päätyttyä
  - *aio_lio_opcode*: liittyy *lio_listio*-kutsuun (katso jäljempänä)

Asynkroninen siirräntä

- Lukuoperaatio asetetaan asynkronisesti funktiokutsulla:
  int *aio_read* (**struct aiocb** *aiocbp)
- Kirjoittamista annetuista kuvaajasta ja puskurista pyydetään funktiokutsulla:
  int *aio_write* (**struct aiocb** *aiocbp)
- Molemmat kutsut palauttavat onnistuessaan 0 ja virhetilanteessa -1 sekä asettavat *errno*-arvon
- Muista, että kutsun palaaminen ei tarkoita, että siirräntä olisi valmistunut, vaan että operaatio on annettu KJ:lle käsiteltäväksi ja se suoritetaan taustalla
- Kutsut eivät siis palauta positiivisia arvoja

Asynkroninen siirräntä

- Asynkronisen kutsun tilan saa funktiokutsulla:
  int *aio_error* (**const struct aiocb** *aiocbp**)**
- Kutsu palauttaa 0, jos siirräntä onnistui
- Muuten paluuttaa -1 ja *errno*-arvon, jonka vastaava *read*-tai *write*-kutsu olisi aiheuttanut
- Huomaa, että kutsu EI palauta positiivisia arvoja
- Siirretyt tavut saa kutsumalla funktiota:
  ssize_t *aio_return*(**struct aiocb** *aiocbp)
- Kutsu palauttaa siirretyt tavut, tai virhetilanteessa *-1*
- Huomaa, että tämä EI ole siirrännän virhe, vaan virhe, joka syntyi väärin määrittelystä funktiokutsusta

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Asynkroninen siirräntä

- Asynkronisen siirrännän voi keskeyttää funktiokutsulla:
  int *aio_cancel* (**int** fildes, **struct aiocb** *aiocbp)
- Jos *aiocbp*-parametri on *NULL*, keskeytetään kaikki tiedostokuvaajaan asetetut siirrännät
- Ennen kutsua valmistuneet siirrännät ilmoittavat valmistumisestaan tavalliseen tapaan
- Keskeytyneet kutsut palauttavat *errno*-arvon *ECANCELLED*
- Kutsu palauttaa virhetilanteessa *-1*, muussa tapauksessa paluu arvo jokin seuraavista
  - *AIO_CANCELLED*: kaikki pyydetyt operaatiot keskeytettiin
  - *AIO_NOTCANCELLED*: yhtä tai useampaa operaatiota ei saatu keskeytettyä, koska ne olivat jo käynnissä
  - *AIO_ALLDONE*: kaikki operaatiot olivat jo valmistuneet

Asynkroninen siirräntä

- Asynkronista siirtoa voi myös jäädä odottamaan *select*-kutsun tapaan

  int *aio_suspend*(**const struct aiocb** *const list[], **int** nent, **const struct timespec** *restrict timeout)
- Tällöin prosessi odottaa joko määritetyn ajan tai yhdenkin siirrännän valmistumiseen asti, jonka jälkeen kutsu palauttaa
  - *0*, jos yksikin siirräntä valmitui (mikä siirräntä, pitää selvittää erikseen)
  - *-1*, jos syntyi virhetilanne, esimerkiksi ajastimen laukeaminen saa tämän aikaan ja *errno*-arvo on tällöin *EAGAIN*

Asynkroninen siirräntä

- Asynkronisia, ja myös synkronisia, siirtoja voi myös pyytää sarjassa funktiokutsulla:
  int *lio_listio*(**int** mode, **struct aiocb** *const list[], **int** cbcnt, **struct sigevent** *sig)
  - *mode*: joko *LIO_WAIT* tai *LIO_NOWAIT*, kertoo halutaanko synkroninen tai asynkroninen siirräntä
  - *list[]*: lista *aiocb*-tietorakenteita
  - *cbcnt*: listan alkioiden lukumäärä
  - *sig*: signaali, kun KAIKKi operaatiot ovat valmistuneet
- *aiocb*-tietorakenteen *aio_lio_opcode* kertoo operaation tyypin: *LIO_READ, LIO_WRITE, LIO_NOP*

Esimerkki

```
static void sig_int(int signo) {
  if (signo == SIGINT) lopeta=1;
  else if (signo == SIGIO) sigio=1;
}

int main(int argc, char *argv[]) {
   struct sigaction sig; int ret;

    /* no parameters */
   if (argc != 1) {
      fprintf(stderr, "Usage: %s\n", argv[0]); return 1;
   }

   /* Empty data structure and create the signal handlers */
   sigemptyset(&sig.sa_mask);
   sig.sa_flags=0;
   sig.sa_handler = sig_int;

   if(sigaction(SIGINT,&sig,NULL) != 0) exit(0);
   if(sigaction(SIGIO,&sig,NULL) != 0) exit(0);

   /* set up aio for STDIN */
   if(aseta_aio(0) != 0) {
        perror("aio_read");
        return 0;
   }
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Esimerkki

```
   printf("Okei, kaikki valmista.\n");
   printf("teen jotain...");

   while(!lopeta){
     sleep(1);
     printf("."); fflush(stdout);
     if(sigio)
     {
         sigio=0;
         printf("Katsotaanpa tuliko jotain...");
         ret = aio_error(&aiocbt);
         if(ret==0)
         {
            printf("täältä tuli jotain!\n");
            printf("  Viesti STDIN:sta: %s", (char*)aiocbt.aio_buf);
            aseta_aio(0);
         }
         else perror("aio");
      }
   }
   printf("Jep, signaali tuli, lopeta=1, moi, moi\n");
   return 0;
}
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostolukitukset

- UNIX ei huolehdi automaattisesti lukituksista, vaan ne on ohjelmoitava sovellukseen itse:
  - 'neuvoa antava lukko' (advisory lock), joka estää muita lukitsemasta jo lukittua osaa, mutta ei estä toisten prosessien read() / write() - kutsuja
    - käyttöoikeudet (mode) sallii tai kieltää
    - prosessi kirjoitettava "hyvätapaiseksi" (vrt. ue)
  - pakottavaa lukko (mandatory lock): estää lukitsemasta jo lukittua osaa ja estää ristiriidassa olevat read() / write() - kutsut. Voi johtaa lukkiutumistilanteisiin.

Tiedostolukitukset

- Jos usea prosessi päivittää samaa tiedostoa, voi syntyä sotkua
- Yksi tapaa hoitaa lukitus on ohjelmoida sovellukseen oma ns. lukkotiedosto, esim. microEmacs (ue) käyttää tiedostoa “tiedoston.lock~”, joka sisältää tiedoston avanneen käyttäjän tunnuksen
- Lukituksen voi hoitaa myös KJ:n tuella.
- Tällöin prosessi varaa oikeuden tiedoston tai sen osan käsittelemiseen ja muut prosessit jäävät odottamaan tarvittaessa lukon vapautumista

Tiedostolukitukset

- Jos usea prosessi päivittää samaa tiedostoa, voi syntyä sotkua
- Yksi tapaa hoitaa lukitus on ohjelmoida sovellukseen oma ns. lukkotiedosto, esim. microEmacs (ue) käyttää tiedostoa “tiedoston.lock~”, joka sisältää tiedoston avanneen käyttäjän tunnuksen

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostolukitukset

- Yksinkertainen sovelluksen oma lukkomekanismi:

```
int fd=-1, fd2=0;
/* varaa */
do {
    fd=open("tdsto.lock~",O_CREAT | O_EXCL,FILE_MODE);
    if(fd==-1) sleep(5);
} while (fd < 0 && errno == EEXIST);

/* tee varsinaiselle tiedostolle mitä mieli tekee */
fd2=open(“tdsto”,O_RDWR);

/* vapauta */
close(fd2);
close(fd);
unlink("tdsto.lock~");
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostolukitukset

- Edellisessä ratkaisussa on paljon huonoja piirteitä, esimerkiksi
  - Jos ohjelma kaatuu, voi lukkotiedosto jäädä paikalleen
  - Muut halukkaat odottavat aktiivisesti, mikä kuluttaa resursseja turhaan
  - Muiden tiedoston käyttäjien käytettävä samaa tapaa ja tiedettävä lukkotiedoston polkunimi

Tiedostolukitukset

- Lukituksen voi hoitaa myös KJ:n tuella.
- Tällöin prosessi varaa oikeuden tiedoston tai sen osan käsittelemiseen ja muut prosessit jäävät odottamaan tarvittaessa lukon vapautumista
- Menetelmässä on yksi ongelma, lukitus on vain vihje, tiedostoa voi silti käsitellä vapaasti: toinen prosessi voi ottaa lukitun tiedoston käyttöön ihan vapaasti.

Tiedostolukitukset

- Tiedoston voi myös lukita funktiokutsulla:
  int *fcntl*(**int** fd, **int** cmd, **struct flock** *lock)
- *cmd* voi olla:
  - F_GETLK kysy onko alueella lukkoa. Jos alueella on lukko, niin palauttaa estävän lukon tiedot rakenteessa lock, muuten arg.l_type = F_UNLCK.
  - F_SETLK varaa / vapauta lukko. Jos varaaminen /vapauttaminen ei onnistu, palauttaa -1 ja errno=EAGAIN.
  - F_SETLKW varaa / vapauta lukko. Jos ei onnistu, jää odottamaan.
- Onnistuessaan kutsu palauttaa arvon 0
- Virhetilanteessa paluuarvo on -1 ja errno on asetettu
- Mahdollisia virheitä esim. EAGAIN ja EDEADLK (F_SETLKW)

Tiedostolukitukset

- *struct flock* sisältää:
  - short l_type: type of lock
  - short l_whence: flag for starting offset
  - off_t l_start: relative offset in bytes
  - off_t l_len: size of locket region
  - pid_t l_pid process ID of the process holding the lock
- *l_type*: L_RDLCK, L_WRLCK tai L_UNLCK
- *l_whence*: SEEK_SET, SEEK_CUR, SEEK_END
- *l_len*: alueen pituus, jos 0, lukitaan tiedoston loppuun

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostolukitukset

- Lukon tyypin voi muuttaa avaamatta lukkoa välillä
- Lukot säilyvät koodinvaihdossa (exec-kutsu), ellei ole asetettu lipuketta *FD_CLOEXEC*
- Lapsiprosessi ei peri äidin asettamia lukkoja (muutenhan kaksi prosessia voisi kirjoittaa samaan tiedostoon samaan aikaan)
- Kun tiedosto suljetaan, lukot avautuvat
- Jos tiedosto avataan prosessissa uudestaan, lukot avautuvat
- Esimerkki lukituksesta: *flock.c*

Tiedostolukitukset

- Esimerkki sovelluksesta:
  - Tilitiedot tiedostossa, vakiomittainen tietue
  - Yksi prosessi kerrallaan voi modifioida samaa tiliä
  - Eri tilien rinnakkainen käyttö onnistuu, kun vain yksi tili lukitaan kerralla
  - Tilin n lukitseminen esimerkiksi:

```
struct flock l;
l.l_type = F_RDLCK tai F_WRLCK /*get tai set-operaatio*/
l.l_whence = SEEK_SET;
l.l_start = n * sizeof(tili_tietue);
l.l_len = sizeof(tili_tietue);
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostolukitukset

- Esimerkkejä:

```
/* Lukitse kokonainen tiedosto omaan käyttöön: */

      struct flock lukko;

      lukko.l_type   = F_WRLCK;
      lukko.l_whence = SEEK_SET;
      lukko.l_start  = 0;
      lukko.l_len    = 0;

      fcntl(fd, F_SETLKW,&lukko);
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Tiedostolukitukset

- Esimerkkejä:

```
/* Lukitse 100 seuraavaa tavua siten, että muut eivät
saa kirjoittaa ko. Alueelle, lukea voivat */

    lukko.l_type   = F_RDLCK;
    lukko.l_whence = SEEK_CUR;
    lukko.l_start  = 0;
    lukko.l_len    = 100;
    fcntl(fd, F_SETLKW,&lukko);


/* Avaa edellä lukitusta alueesta keskeltä 50 tavua */
     lukko.l_type = F_UNLCK;
     lukko.l_whence = SEEK_CUR;
     lukko.l_start = 25;
     lukko.l_len = 50;
     fcntl(fd, F_SETLKW,&lukko);
```

TELECOMMUNICATIONS SOFTWARE AND MULTIMEDIA LABORATORY

Muistiinkuvatut tiedostot

- Levytiedosto voidaan kuvata keskusmuisti-puskuriin siten, että kun haetaan tavuja puskurista, saadaan vastaavat tiedoston tavut
- Talletus toimii vastaavasti, ts. kun kirjoitetaan muistialueeseen, tavut menevät myös levylle
- Prosessi voi siis tehdä siirräntää ilman funktioita read() ja write()
- Toiminta perustuu virtuaalimuistin hyödyntämiseen: sivutusalgoritmit huolehtivat muistiinnoudosta ja levylle tallettamisesta

Muistiinkuvatut tiedostot

- Tiedoston kuvaaminen muistiin hoituu funktiolla:
  void **mmap(***void** *addr, **size_t** len, **int** prot, **int** flags, **int** fildes, **off_t** off*)*
- Tässä:
  - *addr* kertoo virtuaalimuistipaikan, jos arvo on 0, KJ valitsee paikan
  - *len* kertoo kuvattavan alueen pituuden
  - *prot* määrittelee suojauksen: PROT_READ/WRITE/EXEC/NONE pitää vastata open-kutsun vastaavaa *mode*-märittelyä
- Jos kuvatun tiedoston koko muuttuu jonkin ulkoisen tekijän vaikutuksesta, seuraus voi olla mitä tahansa...

Muistiinkuvatut tiedostot

- Tiedostokuvaajan sulkeminen ei vielä vapauta muistiinkuvausta, vaan se on tehtävä funktiolla:
  int *munmap*(**caddr_t** addr, **size_t** len)
- Lapsiprosessi perii fork()-kutsussa äitinsä muistiinkuvatut alueet, mutta exec()-kutsussa ne vapautetaan
- Muistiinkuvattu I/O on suhteellisen nopeaa, sillä siirrossa ei käytetä KJ:n puskurointi, vaan siirretään suoraan sovelluksen käytettäväksi
- Yksi hyvä käyttökohde on palvelin, joka ylläpitää jotain tietoa tietorakenteissaan ja tämä tieto pitäisi varmistaa levylle – kuvataan kriittinen alue tiedostoon, jolloin varmistus hoituu tavallaan “taustalla”