1

# Sovelluksen toimintaympäristö



## Tämän osan sisältö

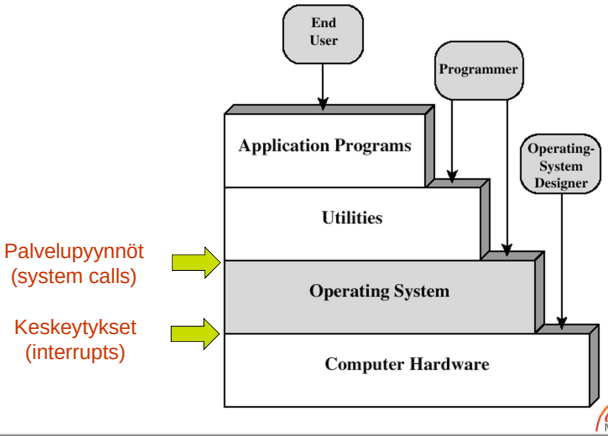
- Tietokonejärjestelmän hierarkia
  - Laitteisto, KJ, sovellukset
  - Etuoikeutettu tila vs. käyttäjätila
- KJ:n tehtäviä ja palveluja
- Palvelupyynnöt a.k.a systeemikutsut
- Virhetilanteista
  - #include <errno.h>
  - perror(), strerror()
  - stderr
- Prosessin suoritusympäristö
  - Komentoriviargumentit, argv[]
  - Ympäristömuuttujat, envp[]



# TIETOKONEJÄRJESTELMÄN HIERARKIA



## Hierarkkinen rakenne






## Laitteisto

- CPU, tarjoaa joukon laitetoimintoja
  - kello ja kellokeskeytys
  - keskeytysten huomaaminen
  - etuoikeutettu / käyttäjätila (bitti CPU:n tilarekisterissä)
  - ohjelman sisäisen osoitteen muunnos fyysiseksi muistiosoitteeksi ja suojaustarkistus
  - nämä ovat perusvaatimukset moniajojärjestelmän toteuttamiseksi
    - Mietippä joutessais miksi!
- Keskusmuisti (memory)
  - suoritettavana olevat ohjelmat ja niiden data
- Väylät (bus)
  - yhdistävät laitteiston osat toisiinsa
- Laiteohjaimet (controller, adapter) (~ laitteet)
  - toimivat KJ:hin kuuluvien laiteajurien (driver) ohjauksessa
  - fiksuja ja vähemmän fiksuja ohjaimia
  - DMA, direct memory access



## KJ + Etuoikeutettu tila (kernel mode, supervisor mode)

- Kun koodia suoritetaan CPU:n etuoikeutetussa tilassa, sillä on kaikki oikeudet kaiken laitteiston käyttöön
  - Siirtyminen etuoikeutettuun tilaan aina kontrolloitua: aina keskeytyskäsittelyn aluksi (laitetoiminto)
  - CPU suostuu suorittamaan kaikkia käskykantaan kuuluvia käskyjä
- Keskeiset KJ:n osat vaativat toimiakseen etuoikeutetun tilan
  - = oikeudet kaikkeen laitteistoon
  - mutta suuri osa KJ:stä toimii "tavallisen" ohjelman oikeuksin
- KJ:n ohjelmallista toimintaperustaa kutsutaan ytimeksi (kernel)
  - keskeytyskäsittely (interrupts)
  - laiteajurit (drivers)
  - muistinhallinta (memory management)
  - prosessien hallinta ja vuorottaminen (process management, scheduling)
  - tiedostojärjestelmä (file system)
  - siirräntäjärjestelmä (input/output system)



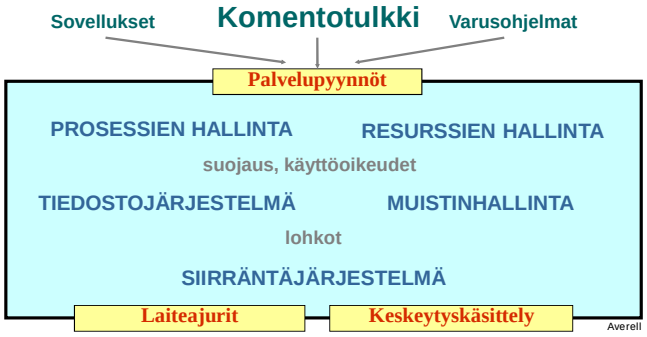
## Sovellukset + Käyttäjätila (user mode)

- 1) KJ:n tarjoamat varus- / palveluohjelmat (utility programs)
  - komentotulkki, graafinen KÄLI
  - tiedostojen käsittely ja käyttöoikeudet
  - editorit: tiedostojen luonti, ohjelmankehitys, tiedon/tekstin muokkaus...
  - käyttäjien välinen kommunikointi, sähköposti, ...
  - töiden ja prosessien kontrollointi
  - jne...
- 2) muiden tarjoamat "omat" sovellukset (application programs)
  - toimisto-ohjelmat, Internet-selain, pelit ...
  - oma komentotulkki, oma ikkunointiympäristö, ...
- Sovellukset käyttävät ytimen palveluja palvelupyyntöjen välityksellä
  - sovelluksilla ei oikeutta käyttää suoraan laitteistoa
    - KJ:n voitava tarkastaa käyttöoikeudet
  - CPU suorittaa tietyt konekielen käskyt vain etuoikeutetussa tilassa, (etuoikeutetut käskyt)



# KJ:N TEHTÄVIÄ JA PALVELUJA

# KJ:n tehtäviä ja palveluja




**Laiteohjaimet ja laitteet**



# KJ:n tehtäviä ja palveluja

- KJ on ohjelmisto, joka huolehtii sovellusten suorituttamisesta CPU:ssa
  - antaa laitteiston sovelluksen käyttöön (CPU, muisti, oheislaitteet)
  - huolehtii, että sovellus ei yksin valloita koko laitteistoa
  - tarjoaa turvallisen suoritusympäristön
- KJ tarjoaa laitteiston käytössä tarvittavat palvelunsa sovellukselle
  - laitteistopiirteiden hallinta KJ:n heiniä, sovelluksen ulkopuolella
  - sovellus esittää tarpeensa palvelupyynnöin
- Siirtyminen KJ:n koodiin keskeytysmekanismin kautta
  - palvelupyyntö aiheuttaa keskeytyksen
  - myös laitteisto esittää ohjaustarpeensa keskeytyksillä
  - CPU tutkii jokaisen konekielisen käskyn jälkeen onko tullut keskeytyksiä



# Komentotulkki / Käyttöliittymä

- Alkujaan UNIX-käyttäjä kommunikoi koneen kanssa merkkipohjaisen komentotulkin (shell) avulla
  - lukee käyttäjän komennot
  - pyytää KJ:tä käynnistämään käyttäjän haluamat ohjelmat
  - sh (Bourne shell), bash (Bourne again shell)
  - csh (C-shell), tcsh (Tenex C-shell), ksh (Korn shell), ...
- Myös Windowsissa perinteinen komentotulkkiohjelma
  - Command Prompt ja PowerShell
- Nykyään graafinen käyttöliittymä ajaa saman asian
  - ohjelmia käynnistellään valikoista tai klikkailemalla
  - Gnome, KDE, Cinnamon, Mate, Unity, ...
- Komentotulkki tai graafinen KÄLI ei oikeastaan kuulu UNIX-käyttöjärjestelmään
  - voisit korvata ne halutessasi vaikka omilla versioillasi



# KJ:n tarjoamia palveluja

- Prosessien hallinta (process management), eli ohjelmien suorituttaminen
  - prosessien käynnistäminen
    - kirjanpito, lataaminen muistiin ...
  - prosessien vuorottaminen (scheduling)
    - CPU:n käyttövuorot, resurssien varaus ja käyttö
  - prosessien tappaminen
    - resurssien vapauttaminen
- Muistinhallinta (memory management)
  - varaa muistitilaa sovelluksille
  - sovellus saa viitata vain tiettyihin muistiosoitteisiin
  - laitteisto huolehtii ajonaikana suojaustarkistuksista (protection)
    - sen tekee CPU:n MMU (Memory Managent Unit)



# KJ:n tarjoamia palveluja

- Siirräntä (I/O, input - output)
  - sovellus pyytää I/O-palvelua KJ:ltä
  - KJ 'komentaa' I/O-ohjaimia töihin
  - Kun ohjain saa hommansa valmiiksi, se aiheutta keskeytyksen
  - KJ tarkastaa, että Ok ja päästää sovelluksen jatkamaan
- Tiedostojen hallinta (file management)
  - kirjanpito tiedostoista ja hakemistoista
  - tiedostosta lukeminen / tallettaminen
  - käyttöoikeuksien hallinta (access rights)



# KJ:n tarjoamia palveluja

- Kirjanpito / tilinpito (accounting)
  - tilastointi resurssien käytöstä
  - suorituskyvyn seuranta (esim. vastausaika)
  - järjestelmäparametrien optimointi hyvän suorituskyvyn saamiseksi
  - koneen käyttäjien laskuttaminen
- Virhetilanteiden käsittelyä
  - laitteistovirheet
  - ohjelmistovirheet
  - resurssipula
- Virheistä toipumista
  - palauttaa statustietoa sovellukselle
  - uudelleenyritykset
  - prosessin tappaminen

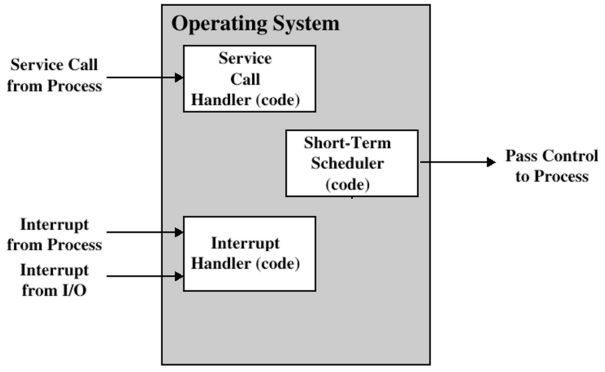
*Välikysymys:*
*Mitä yleistermi resurssi tarkoittaa?*



# PALVELUPYYNNÖT
## a.k.a
## systeemikutsut



# Palvelupyyntö



# Palvelupyynnöt ja vuorottaminen

- Sovellus pyytää KJ:n palvelua käskykantaan kuuluvan konekielisen käskyn välityksellä, esim. käsky SVC (supervisor call), INT tms.
- Palvelupyyntö aiheuttaa keskeytyksen (interrupt)
  - CPU etuoikeutettuun tilaan
  - CPU suorittamaan KJ:tä
- Myös laitteisto pyytää KJ:n palvelua aiheuttamalla keskeytyksen
  - Kello
  - Ohjain, kun annettu I/O-tehtävä valmis
- Keskeytys saattaa aiheuttaa sen, että suorituksessa olleen prosessin täytyy jäädä odottamaan palvelun valmistumista
  - KJ siirtää prosessin pois suorituksesta, ja valitsee suoritettavaksi uuden prosessin
  - Kun palvelu valmistuu, voi KJ siirtää odottavan prosessin taas suoritukseen

  => vuorottaminen (scheduling)



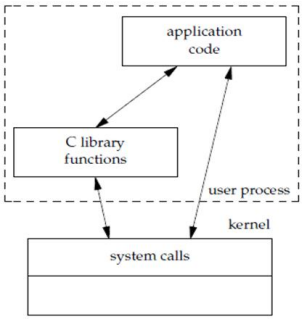
# Montako on riittävän paljon?

- Palvelupyyntörajapinta riippuu KJ:stä
  - vaikka rajapinnoissa eroja, tarjoavat lähes kaikki KJ:t suunnilleen samat palvelut, mm.
    - Prosessit ja niiden välinen kommunikointi
    - Muistinhallinta
    - Tiedostot ja tiedostojärjestelmä
    - Siirräntä (I/O)
- Linuxissa systeemikutsuja noin 380 kpl
  - `man syscalls`
  - määrää kasvanut huomattavasti viime vuosina
- Standardoitu POSIX-rajapinta
  - Näin eri UNIXin versioista on saatu yhteensopivampia
  - `#include <unistd.h>`
  - POSIX = Portable Operating System Interface (for UNIX)



# Palvelupyynnöt vs. kirjastorutiinit

- Ohjelmoija ei käytä välttämättä suoraan palvelupyyntöjä
- Esim. Javassa ohjelmoija käyttää KJ:n tarjoamia palveluja epäsuoraan Java API:n kautta
- Samoin C tarjoaa ohjelmoijalle korkeamman tason kirjastorutiineja
- Kirjastorutiinien toteutus käyttää KJ:n palvelupyyntöjä




bash-komentotulkin kehoite

```
$ man 2 intro        Introduction to system calls
$ man 3 intro        Introduction to library functions
```



# POSIX –rajapinta (esimerkkejä)

**Process management**

| Call | Description |
|---|---|
| pid = fork( ) | Create a child process identical to the parent |
| pid = waitpid(pid, &statloc, options) | Wait for a child to terminate |
| s = execve(name, argv, environp) | Replace a process' core image |
| exit(status) | Terminate process execution and return status |

**File management**

| Call | Description |
|---|---|
| fd = open(file, how, ...) | Open a file for reading, writing or both |
| s = close(fd) | Close an open file |
| n = read(fd, buffer, nbytes) | Read data from a file into a buffer |
| n = write(fd, buffer, nbytes) | Write data from a buffer into a file |
| position = lseek(fd, offset, whence) | Move the file pointer |
| s = stat(name, &buf) | Get a file's status information |

**Directory and file system management**

| Call | Description |
|---|---|
| s = mkdir(name, mode) | Create a new directory |
| s = rmdir(name) | Remove an empty directory |
| s = link(name1, name2) | Create a new entry, name2, pointing to name1 |
| s = unlink(name) | Remove a directory entry |
| s = mount(special, name, flag) | Mount a file system |
| s = umount(special) | Unmount a file system |



# Käyttöesimerkki

```
#define TRUE 1                          "Riisuttu" komentotulkki

while (TRUE) {                          /* repeat forever */
    type_prompt( );                     /* display prompt on screen */
    read_command(command, parameters);  /* read input from terminal */

    if (fork( ) != 0) {                 /* fork off child process */
        /* Parent code. */
        waitpid(-1, &status, 0);        /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0); /* execute command */
    }
}
```

- fork() luo uuden prosessin (klooni)
- waitpid() jää odottamaan, että toinen prosessi päättyy
- execve() vaihtaa suoritettavaa koodia



# VIRHETILANTEET



- Koodissa pääsääntöisesti tarkistettava kutsujen onnistuminen
- Virhetilanteessa systeemikutsut ja C-kirjastofunktiot
  - palauttavat yleensä arvon -1 tai NULL
  - ja asettavat muuttujalle **`errno`** virheen numeron

  lue man-sivulta
- Onnistunut systeemikutsu ei muuta muuttujaa `errno`
  - muuttujan arvona saattaa siis olla joku ikivanha virhekoodi
  - tutki aina ensin kutsun paluuarvo ja vasta sitten errno-muuttuja
- Virheiden numerokoodit ja tunnukset löytyvät otsaketiedostosta

  `/usr/include/sys/errno.h`

  `/usr/include/asm/errno.h` Linuxeissa
- Virheilmoitusten vakionimet ja -tekstit löytyvät esim. man-sivuilta

  `$ man 3 errno`
- Tiettyyn rutiiniin liittyvät virhetilanteet on kuvattu ko. rutiinin manuaalisivulla, esim.

  `$ man -s 2 fork`

  lue man-sivulta



```
/* Error codes */

#define EPERM   1       /* Operation not permitted    */
#define ENOENT  2       /* No such file or directory  */
#define ESRCH   3       /* No such process            */
#define EINTR   4       /* Interrupted system call    */
#define EIO     5       /* I/O error                  */
#define ENXIO   6       /* No such device or address  */
#define E2BIG   7       /* Argument list too long     */
#define ENOEXEC 8       /* Exec format error          */
#define EBADF   9       /* Bad file number            */
#define ECHILD  10      /* No children process        */
#define EAGAIN  11      /* Try again                  */
#define ENOMEM  12      /* Out of memory              */
#define EACCES  13      /* Permission denied          */
#define EFAULT  14      /* Bad address                */
#define ENOTBLK 15      /* Block device required      */
#define EBUSY   16      /* Device or resource busy    */
#define EEXIST  17      /* File exists                */
#define EXDEV   18      /* Cross-device link          */
#define ENODEV  19      /* No such device             */
#define ENOTDIR 20      /* Not a directory            */
#define EISDIR  21      /* Is a directory             */
#define EINVAL  22      /* Invalid argument           */
... jne
```

- Numeroa vastaavan merkkijonon saa funktiolla
```
extern int errno.h
#include <string.h>
char *strerror(int errnum);
```
- Virheilmoitustekstin voi tulostaa funktiolla
```
#include <stdio.h>
void perror(const char *msg);
```
  - tulostaa parametrina saamansa merkkijonon, kaksoispisteen ja errno-numeroa vastaavan järjestelmän virheilmoituksen stderr-tiedostoon

Huom!

- **stderr** on puskuroimaton, joten tulostetut merkit menevät heti sinne
  - käytä sitä esim. testauksessa tarvitsemiisi aputulostuksiin
- stdout sensijaan on puskuroitu, joten teksti ilmestyy vasta, kun sinne tuotetaan \n, tai kutsutaan fflush(stdout)

StRa13 Kuva 1.8

```
#include <errno.h>
#include <stdio.h>
int main(int argc, char *argv[]) {

    fprintf(stderr, "EACCES:  %s\n", strerror(EACCES);
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

```
$ ./a.out
EACCES: Permission denied
./a.out: No such file or directory
```



# PROSESSIN SUORITUSYMPÄRISTÖ



## Prosessin käynnistys

- Ohjelman (~prosessin) voi käynnistää komentotulkin kautta, esim.
```
$ nano -m teht1.c          (käynnistä nano-editori, -m = mouse)
$ gcc teht1.c -o teht1     (käännä C-ohjelma, -o = output)
$ ./teht1                  (käynnistä käännetty objektikoodi)
```
- Käynnistetyn C-ohjelman suoritus alkaa sen funktiosta main()
  - `int main(int argc, char *argv[])`
- Komentotulkki välittää komentorivillä annetut argumentit ohjelmalle pinossa Niihin voi viitata muuttujilla argc ja argv[]
  - `argc` argumenttien lukumäärä
  - `argv[]` ohjelman nimi ja argumentit merkkijonoina
- Yo. esimerkissä
  - `argc = 3`
  - `argv[0] = "nano"`
  - `argv[1] = "-m"`
  - `argv[2] = "teht1.c"`
  - `argv[3] = NULL`

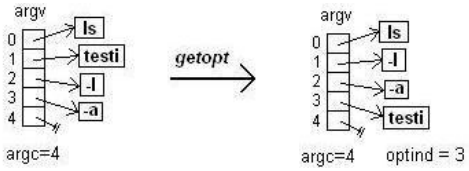


## Komentorivivalitsimien käsittely

- Valitsimet voi antaa vapaassa järjestyksessä ja niitä voi yhdistellä
  - `ls -l *.c`
  - `ls -a -l testi`
  - `ls -la`
- Valitsimella voi olla myös omia parametreja
  - `ssh munlinux.fi -l averell` -l = login name
- Kirjastofunktiolla `getopt()` voi tutkia komentoriviargumentteja
```
#include <unistd.h>
int getopt(int argc, char * argv[], char *optstring)
```
- Funktiolle annetaan kolmantena parametrina sallitut valitsinkirjaimet, ja tieto siitä liittyykö valitsinkirjaimeen parametri
  - perinteisesti valitsin on vain yksi kirjain
  - "--" -alkuisten valitsimien käsittelyä varten funktio `getopt_long()`



- getopt() siirtää toistossa valitsimet vähitellen argv[]-taulukon alkuosaan, loppuun jää vain komennon varsinaisia argumentteja
- Kun getopt() lopuksi palauttaa -1, on muuttujassa optind ensimmäisen argumentin indeksi
- Esim. `while ((opt = getopt(argc, argv, "la"))!= -1)…`




- getopt():n kolmantena parametrina merkkijono
  - lueteltu mahdolliset valitsinkirjaimet
  - kun kirjaimen perässä kaksoispiste : niin valitsimella oma argumentti heti valitsimen perässä (se ei ole siis pelkkä on/off-lipuke)
  - esim. `getopt(argc, argv, "t:n")`



- getopt()-kutsu
  - palauttaa arvonaan käsittelyvuorossa olevan valitsinkirjaimen
  - palauttaa '?', jos käsittelyvuorossa ei-sallittu valitsinkirjain
  - palauttaa -1, kun ei enää käsiteltävää
- Kutsun jälkeen tutkitaan seuraavia muuttujia

| | |
|---|---|
| `extern char *optarg` | valitsimeen liittyvän argumentin arvo, jos löytyi |
| `extern int optind` | ensimmäisen varsinaisen argumentin indeksi, argumentti on siis kohdassa argv[optind] |
| `extern int opterr` | määrittelee tulostetaanko virheilmoitus stderr:iin, jos arvoksi asetettu 0, ei tulosta |
| `extern int optopt` | virheellinen valitsinkirjain (getopt() palautti merkin ?) |

- Seuraava kutsu kohdistuu seuraavan valitsimeen jne.



Esimerkki: `prog [-n] [-t nsecs] name`

```
#include <unistd.h>
extern char *optarg;
extern int optind, opterr,optopt;
int main(int argc, char *argv[]) {
   int flags=0, opt, nsecs=0, tfnd=0;
   while ((opt = getopt(argc, argv, "nt:")) != -1) {
     switch (opt) {
       case 'n': flags = 1; break;
       case 't': tfnd = 1; nsecs = atoi(optarg); break;
       default: /* '?' */
         fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n", argv[0]);
         exit(EXIT_FAILURE);
     }
   }
   printf("flags=%d; tfnd=%d; optind=%d\n", flags, tfnd, optind);
   if (optind >= argc) {
       fprintf(stderr, "Expected argument after options\n");
       exit(EXIT_FAILURE);
   }
   printf("name argument = %s\n", argv[optind]);
   /* Tähän sovelluksen varsinainen koodi */
   exit(EXIT_SUCCESS);
}
```

sallitut valitsimet -n ja -t
valitsimeen t liittyy arvo, esim. -t 10



## Prosessin päättyminen

- Prosessin suoritus päättyy "normaalisti", kun suoritetaan
  - main()-funktiossa `return(status)`, tai main()-funktio päättyy
  - `exit(status)`
  - `_exit(status)`
- `#include <stdlib.h>`
- `void exit(int status);`
  - tekee standardikirjastoihin liittyvää siivousta
  - kutsuu edelleen funktiota _exit()
- Normaalisti päättyvän ohjelman tulisi palauttaa 0 eli `EXIT_SUCCESS`
- Palauta virhetilanteissa jotain muuta, esim. -1 eli `EXIT_FAILURE`
- Prosessin suoritus päättyy myös kun
  - kutsutaan funktiota `abort()`
  - saadaan signaali, johon ei ole varauduttu
- Prosessin päättyessä suoritetaan vielä funktiolla `atexit()` rekisteröidyt funktiot, viimeiseksi rekisteröity ensin

# Ympäristömuuttujat

- Tietoa suoritusympäristöstä voi välittää prosessille myös ympäristö-muuttujien avulla
- Komentoriviargumentit on tarkoitettu yksittäisen sovelluksen omien parametrien välittämiseen
- Ympäristömuuttujien arvot pysyvät kauan samoina ja ne ovat usealle sovellukselle (prosessille) yhteisiä
- Ympäristömuuttujalla on nimi ja arvo. Se on muotoa
  `"nimi=arvo"` on siis yksi merkkijono
- Asetetaan bash-komentotulkissa komennolla `export` tai `declare`

```
$ export KURSSI=uloy
$ declare -x KURSSI=uloy
```



- Komentotulkissa viitattaessa käytettävä edessä $-merkkiä

```
$ echo $KURSSI
```

- Kaikkien ympäristömuuttujien arvot saa komentamalla

```
$ env
$ export
```

- Käynnistyvä prosessi saa ympäristömuuttujien arvot sen käynnistäneeltä prosessilta
  - komentoriviltä käynnistetty ohjelma tuntee siten komentotulkin ympäristömuuttujat
- Mammaprosessi ei näe lapsiprosessin asettamia ympäristömuuttujia
- Jos ympäristömuuttuja halutaan asettaa vain yhden ohjelman suoritusajaksi, voi sen tehdä `env`-komennolla, esim.

```
$ env KURSSI=uloy prog -kc puppu.dat
```

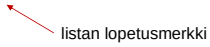
- env-komennon valitsimella `-i` voi kieltää vielä äitiprosessin ympäristön periytymisen



- Ympäristömuuttujiin voi viitata muuttujan envp[] kautta

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

  - `envp[]` kaikki ympäristömuuttujat merkkijonoina
  - `envp[0] = "HOSTNAME=edunix.metropolia.fi"`
  - `envp[1] = "TERM=xterm"`
  - `...`
  - `envp[n] = NULL`






- Ympäristömuuttujia voi käsitellä myös esittelemällä koodissa

```
extern char **environ;
```

ja käyttämällä sitten funktioita

```
#include <stdlib.h>
char *getenv(const char *name);
char *putenv(const char *str);
int setenv(char *name, char *value, int overwrite);
int unsetenv(const char *name);
int clearenv(void);
```

```
printf("Tunnus on %s\n", getenv("USER"));

if (putenv("KURSSI=uloy") == 0)
... onnistui ...
else
... muistiongelmia ...
```

- putenv() / setenv() ei muuta main()-funktion envp[]-argumenttia, sensijaan tuo environ-muuttuja muuttuu. getenv() näkee uudet arvot.
- Lapsiprosessi ei voi muuttaa sen käynnistäneen prosessin ympäristömuuttujien arvoja



# Kertauskysymyksiä

- Mitkä ovat KJ:n keskeiset tehtävät?
- Millaisia palvelupyyntöjä tarvitaan, jotta sovellus voisi käyttää laitteistoa?
- Mitä laitteistopiirteitä tarvitaan moniajon toteuttamiseksi?