3

PROSESSIT

ks. StRa13: luvut 8.1-8.10, Sta12: luku 3

Tämän osan sisältö

- Prosessi
- Prosessin kuvaaja
- Prosessinhallinnan rutiinit
  - fork(), exec(), wait(), exit()
- Prosessin tilat
- Prosessin vaihto, vuorottaminen
- Prosessien välinen kommunikointi
  - Putket ja FIFOt

PROSESSI

Prosessi

= Suoritettavaksi otettu ohjelma
- koodi muistissa (voi olla yhteiskäytössä)
- oma data-alue ja pino muistissa (muuttujat, parametrinvälitys)
- kaikki ei välttämättä yhtäaikaa muistissa (virtualimuisti, heittovaihto)

+ KJ:n ylläpitämät hallinnolliset rakenteet
- prosessin kuvaaja
  - tunnistus (pid, ppid), omistaja (uid, gid, euid, egid)
  - prosessin tila, prioriteetti
  - aikalaskureita
  - tallealue CPU:n rekistereiden arvoille
  - tietoja prosessille varatuista muistialueista
  - tietoja prosessin avaamista tiedostoista
  - työhakemiston polkunimi
  - yms. . .
- KJ suorituttaa moniajojärjestelmän prosesseja vuorotellen CPU:ssa
  - vuorottaminen (scheduling, dispatching)
  - prosessin (ohjelmoijan) itse ei tarvitse miettiä muita prosesseja

Prosessin kuvaaja

- Tällä rakkaalla lapsella on kirjallisuudessa monta nimeä
  - Process Table Entry (PTE), Process Control Block (PCB), Process Context, task_structure
- Prosessin kuvaajaan kirjataan tyypillisesti

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Tan09 Fig 2-4

Prosessin tunnistus

- Prosessin numeron saa systeemikutsulla
```
#include <unistd.h>
pid_t getpid(void)
```
  - *Palauttaa prosessin oman prosessinumeron*
```
pid_t getppid(void)
```
  - *Palauttaa prosessin luoneen emoprosessin (parent) numeron*
- Prosessin omistajan ja ryhmän tunnisteet saa systeemikutsuilla
```
uid_t getuid(void)
```
  - *Palauttaa alkuperäisen omistajan tunnisteen (real user ID)*
```
uid_t geteuid(void)
```
  - *Palauttaa voimassaolevan omistajan tunnisteen (effective user ID)*
```
gid_t getgid(void)
```
  - *Palauttaa alkuperäisen omistajan ryhmätunnisteen (real group ID)*
```
gid_t getegid(void)
```
  - *Palauttaa voimassaolevan omistajan ryhmätunnisteen (effective group ID)*

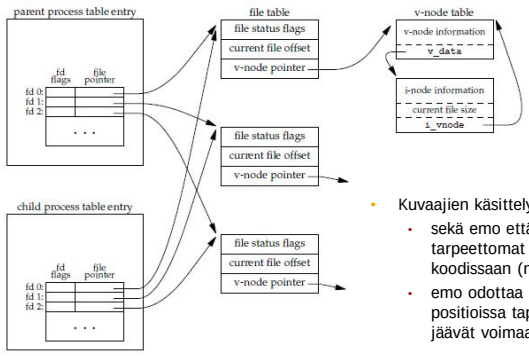
Prosessin luonti

- Prosessi luodaan systeemikutsulla
```
#include <sys/types.h>
#include <unistd.h>
pid_t *fork(void);
```
  - *Palauttaa emoprosessille luodun prosessin numeron, lapselle aina 0*
- fork()-kutsun tuloksena syntyy uusi prosessi, joka on 'klooni' emoprosessista
  - alussa samat prosessinkuvaajan perustiedot
  - yhteinen koodi
  - alussa samat muuttujien arvot
  - alussa samat avoimet tiedostokuvaajat
- Molemmilla on kuitenkin
  - oma data-alue ja pino
  - oma prosessinkuvaaja
- fork()-kutsun jälkeen ei voi olla varma siitä kumpi prosessi (emo vai lapsi) jatkaa aiemmin suoritustaan
  - jos järjestys tärkeää, on ohjelmoitava itse synkronointi


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

int  glob = 6;
char buf[] = "a write to stdout\n";

int main(void) {
  int var;
  pid_t pid;

  var = 88;
  if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
     err_exit("write error");

  printf("before fork\n");
  if ((pid = fork()) < 0)
     err_exit("fork error");
  if (pid == 0) { /* lapsiprosessi */
     glob++;
     var++;
  } else /* emoprosessi */
     sleep(2);

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

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88
```

- Lapsiprosessi perii emoprosessilta kaikki avoimet tiedostokuvaajat
- Sekä emo että lapsi käyttävät yhteistä avoimet tiedostot taulun alkiota
  - niillä on yhteinen luku / kirjoituspositio



- Kuvaajien käsittely esimerkiksi
  - sekä emo että lapsi sulkevat tarpeettomat kuvaajansa omassa koodissaan (muista tämä!)
  - emo odottaa lapsen valmistumista, positioissa tapahtuvat muutokset jäävät voimaan lapsen päättyessä

StRa13 Fig 8.2

Lapsiprosessi luodaan, kun

- halutaan suorittaa emo- 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 päättyminen

- Prosessin suoritus päättyy normaalisti, kun suoritetaan
  - funktiossa main() return(status)
  - funktio exit(status) tai
  - funktio _exit(status)
- Prosessin kuvaaja jää vielä olemaan KJ:n tekemiä lopputoimia varten ("zombie"-prosessina)
  - tiedostojen sulkeminen
  - muistitilan vapauttaminen
  - emoprosessin signalointi
  - paluuarvon välittäminen emoprosessille
  - (aika)resurssien kulutus- / laskutustietojen kokoaminen
- Jos emoprosessi päättynyt ennen lapsiprosessia, merkitsee KJ lapsiprosessin emoksi prosessin 1 (init-prosessi)
  - kokoaa resurssi-/laskutustiedot ja vapauttaa prosessinkuvaajan
- Suoritus voi päättyä myös 'epänormaalisti'
  - poikkeus koodissa: nollallajako, ...
  - kutsutaan funktiota abort()
  - prosessi saa signaalin, johon se ei varautunut tai ei voi varautua
  - ð KJ generoi paluustatuksen

Lapsiprosessin odottaminen

- Emo voi pysähtyä odottamaan lapsen päättymistä systeemikutsuihin

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

  - *Palauttaa: päättyneen prosessin pid, parametrissa status on mm. päättyneen lapsen palauttama statustieto*
- wait() odottaa minkä tahansa lapsen päättymistä
- waitpid() odottaa jonkun tietyn prosessin päättymistä
- Jos lapsiprosessi on jo päättynyt, pääsee emoprosessi heti jatkamaan
- Kun prosessi päättyy, saa emo aina signaalin SIGCHLD
  - oletusarvo on, että emo ei välitä tästä signaalista
  - ei siis pakko odottaa waitissä, lapsiprosessin päättymisen voi käsitellä myös signaalinkäsittelijässä

- pid

> 0	odota lapsiprosessia, jolla ko. pid
0	odota mitä tahansa emon kanssa saman prosessiryhmän lasta
-1	odota mitä tahansa lapsiprosessia
< -1	odota prosessiryhmään |pid| kuuluvaa prosessia

- status
  - lapsiprosessin palauttama statustieto, mukana myös KJ:n lisäämä tieto
  - tulkintaa varten valmiit makrot, esim. WEXITSTATUS(status)
  - ks. man 2 waitpid
- options

WNOHANG	tarkasta vain onko kysytty lapsi päättynyt, älä odota
muut	ks. man 2 waitpid

- Systeemikutsulla

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop,
           int options);
```

  voi yksilöidä vieläkin tarkemmin millaisista lapsen tilan muutoksista emoprosessi on kiinnostunut

  ks. man 2 waitid

Prosessin koodin vaihto

- Prosessi voi vaihtaa suoritettavaa koodia kutsumalla

```
#include <unistd.h>
int execl(char *pathname, char *arg0, ...);
int execv(char *pathname, char *argv[]);
int execlp(char *filename, char *arg0, ...);
int execvp(char *filename, char *argv[]);
int execle(char *pathname, char *arg0, ..., char *envp[]);
int execve(char *pathname, char *argv[], char *envp[]);
int fexecve(int fd, char *argv[], char *envp[]);
```

merkintä tarkoittaa, että parametrien lukumäärä vaihtelee, laita viimeiseksi NULL

- Eroavat komentoriviargumenttien ja ympäristömuuttujien välityksessä
  - execve() on se systeemikutsu, muut ovat sitä käyttäviä kirjastofunktioita
- Sekä listan että vektorin (taulukko) viimeisenä arvona oltava NULL
  - listan / taulukon koko ei muuten tiedossa

- Koodia etsitään annetun suhteellisen tai absoluuttisen polkunimen perusteella
  - execl(), execv(), execle(), execve()
- tai tiedostonimen perusteella ympäristömuuttujassa PATH luetelluista hakemistoista (p)
  - execlp(), execvp()
- Koodille voi välittää komentoriviargumentteja joko listana (l) tai vektorina (v)
  - execl(), execv()
- Koodille voi välittää edellisten lisäksi myös ympäristömuuttujia (e)
  - aina vektorina
  - execle() tai execve()
- Muissa kutsuissa emoprosessin ympäristömuuttujat periytyvät lapselle sellaisenaan muuttujassa

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

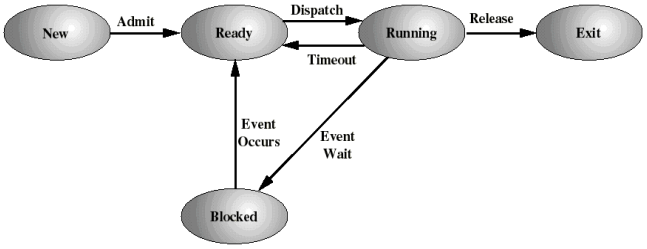


```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void) {
  pid_t pid;

  if ((pid = fork()) < 0) err_exit("fork error");

  if (pid == 0) { /* specify pathname, specify environment */
    if (execle("/home/stevens/bin/echoall",
      "echoall", "myarg1", "MY ARG2", NULL, env_init) < 0)
       err_exit("execle error");
  }

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

  if ((pid = fork()) < 0) err_exit("fork error");

  if (pid == 0) { /* specify filename, inherit environment */
     if (execlp("echoall","echoall","only 1 arg", NULL) < 0)
        err_exit("execlp error");
  }
  exit(EXIT_SUCCESS);
}
```

PROSESSIN TILAT

Prosessin tilakaavio (5 tilaa)

Sta12 Fig 3.6



- Moniajojärjestelmässä kukin prosessi on välillä suorituksessa CPU:ssa ja välillä odottamassa uutta suoritusvuoroa
- Syitä sille, että prosessi joutuu pois suorituksesta
  - prosessi saanut CPU-aikaa niin paljon, että KJ päättää siirtää suoritukseen uuden prosessin (aikaviipale täyttyi)
  - prosessin tekemä palvelupyyntö aiheuttaa odotustilanteen (esim. luku levyltä) eikä se voi jatkaa ennen kuin pyyntö palveltu
  - suuremman prioriteetin prosessi tullut Blocked-tilasta Ready-tilaan

Prosessin tilat

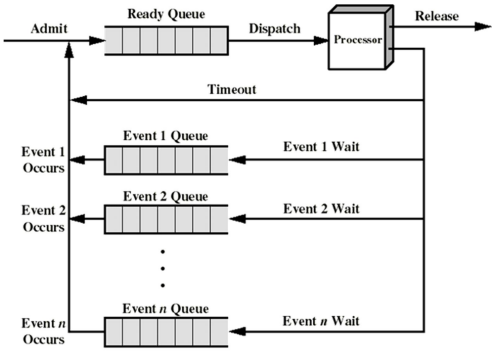
- New
  - KJ luonut prosessin, mutta ei kelpuuta sitä vielä suoritettavaksi
    - esim. vapaata muistia ei riittävästi
    - liian monta prosessia käynnissä
- Running
  - prosessi käyttää parhaillaan prosessoria
- Ready
  - prosessi voisi edetä, jos saisi CPU:n käyttöönsä
  - odottaa Ready-jonossa esim. prioriteetin mukaan
- Blocked
  - prosessi odottaa tapahtuman valmistumista, ei voisikaan vielä jatkaa
    - esim. I/O, synkronointi, ajastus, sleep, ...
  - kullakin laitteella / tapahtumalla oma jono
- Exit
  - suoritus päättynyt, mutta 'saattohoito' tekemättä
    - hallinnolliset rakenteet (lähinnä prosessin kuvaaja) olemassa muita sovelluksia varten, esim: kirjanpitoa, laskutus, tilastot

Prosessin tilasiirtymät

- New ð Ready
  - resursseja riittävästi käytettävänä
    - esim. CPU:n käyttöaste laskenut alle sopivan rajan, muistissa vapaata tilaa
- Ready ð Running
  - vuorottaja valitsee suoritukseen Ready-jonon ensimmäisen prosessin
  - saa CPU:n käyttöönsä seuraavan aikaviipaleen ajaksi
- Running ð Ready
  - prosessin aikaviipale täynnä
  - suuremman prioriteetin prosessi Ready-tilassa
- Running ð Blocked
  - prosessi pyytää KJ:ltä palvelua, jonka valmistumista joutuu odottamaan
    - tarvittava resurssi varattu
    - odottaa siirrännän valmistumista
    - odottaa toisen prosessin etenemistä sopivaan vaiheeseen (prosessin välinen kommunikointi ja synkronointi)
  - virtuaalimuistin laitteisto aiheuttaa sivunpuutoskeskeytyksen ð siirräntää
    - tarvittava koodinpätkä tai data (muuttujat) ei muistissa

- Blocked ð Ready
  - prosessin odotus päättyy
    - tarvittu resurssi vapautui
    - siirräntä valmistui
    - toinen prosessi saavutti synkronointikohdan
    - ajastus valmistui
- Running ð Exit
  - prosessin suoritus päättyy
    - normaali / virhetilanne
  - KJ vapauttaa resurssit prosessin kuvaajaa lukuun ottamatta
  - odotettava, että joku toinen prosessi kokoaa kirjanpidolliset tiedot prosessin kuvaajasta
- Mikä tahansa tila ð Exit
  - KJ tai omistaja voi tappaa (kill-signaali)
  - emoprosessi päättyy
- Exit ð
  - kun 'saattohoito' tehty
  - KJ vapauttaa prosessin resurssit (sulkee tiedostot, vapauttaa muistin, ...)
  - KJ vapauttaa prosessin kuvaajan

Prosessijonot



- Jos vain yksi CPU, vain yksi prosessi suorituksessa
- Muut odottavat jonoissa - kullekin tapahtumatyypille oma jononsa
  - siis prosessin kuvaajat ovat jonossa

Sta12 Fig 3.8

PROSESSIN VAIHTO
vuorottaminen

Milloin prosessinvaihto?

- Vain ja ainoastaan keskeytyksen jälkeen (ei kuitenkaan aina)
  - esim. KJ vain merkitsee siirrännän valmistumista odottaneen Ready-tilaan, ja hetkeksi keskeytynyt prosessi saa jatkaa
- eli kun CPU siirtynyt suorittamaan KJ:tä
  - Palvelupyyntö(keskeytys)
    - prosessi pyytää esim. siirräntää, jonka seurauksena joutuu odottamaan
  - Poikkeus(keskeytys)
    - prosessin suorituksessa virhe: sunnuntaiohjelmoijan koodissa nollallajako, ..
    - prosessi joutuu exit-tilaan ja tapetaan
  - Laitteiston aiheuttama keskeytys
    - esim. kellokeskeytys + prosessin aikaviipale täyttyi, siirräntä valmistui
- Kj:hin kuuluva vuorottaja valitsee CPU:lle suoritettavan prosessin Ready-jonosta, tavallisimmin Round-Robin periaatteella
  - prosessit prioriteetin mukaan jonossa, tärkeimmät jonon keulilla
    - tavallisesti kullekin prioriteetille oma jononsa
  - vuorottaja vie poistuvan Ready-jonon viimeiseksi (prioriteetin mukaan)
  - vuorottaja antaa CPU:n jonon ensimmäiselle (uusi aikaviipale)
    - korkeimman prioriteetin prosessit pääsevät siten ensin suoritukseen

Prosessin vaihto

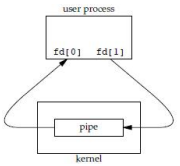
- Rekistereiden arvot talteen CPU:sta prosessin kuvaajaan
- Päivitä mm. aikalaskureita
- Päivitä prosessin tila (Ready / Blocked / Exit ...)
- Liitä prosessi tilan mukaiseen jonoon
  - aika usein Ready-jonon viimeiseksi
- Valitse seuraava prosessi suoritettavaksi
  - Ready-jonon ensimmäinen (suurin prioriteetti)
  - Ready ð Running
- Palauta valitun prosessin rekistereiden arvot CPU:hun
  - prosessi jatkaa siitä mihin jäi, kun menetti CPU:n
- Alusta muistinhallintaan liittyvät laitteistoasiat CPU:ssa
  - MMU, Memory Management Unit
  - prosessi saa viitata vain omiin muistialueisiin

Linux: Prioriteetti ja aikaviipale

- Mitä pienempi prioriteetin numeroarvo, sitä tärkeämpi prosessi
  - Reaaliaikaprosesseilla aina kiinteät prioriteetit 0...99
    - näitä saa käynnistää vain root-käyttäjä
  - Tavallisilla prosessilla vaihteleva prioriteetti 100-139 (oletus 120)
- Kullakin prioriteetilla omat Ready-jononsa
- Prioriteettiin vaikuttaa myös se, käyttikö prosessi koko aikaviipaleen
  = dynaaminen vaihtelu
  - jos prosessi käytti koko viipaleen, sen prioriteetti laskee hieman
  - jos prosessi ei käyttänyt koko viipaletta, sen prioriteetti nousee hieman
  = ratkaisu suosii hieman I/O-sidonnaisia prosesseja (miksi hyödyllistä?)
- Mitä suurempi prioriteetti, sitä pitemmän aikaviipaleen saa
  - oletus 100 ms, maksimi 800 ms
- Lapsiprosessin ensimmäinen viipale otetaan emoprosessin jäljellä olevasta viipaleesta (50 %)
  - seur. kerralla molemmat saavat viipaleensa taas normaalilla tavalla

PROSESSIEN VUOROVAIKUTUS putket

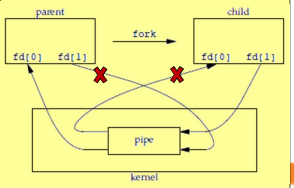
Putket (pipes)



- Putki luodaan systeemikutsulla
```
#include <unistd.h>
int pipe(int pipefd[2]);
int pipe2(int pipefd[2], int flags);
```
- Putki on erikoistiedosto, jolla on kaksi tiedostokuvaajaa
  - pipefd[1] tänne voi kirjoittaa
  - pipefd[0] täältä voi lukea mitä putkeen kirjoitettiin
  - "first-in-first-out"
- pipe()- ja fork()-kutsuilla syntyy tiedonsiirtokanava emo- ja lapsiprosessin_välille, sillä myös putken tiedostokuvaajat periytyvät lapsiprosessille
  - putkia voi käyttää vain 'saman perheen prosessien kesken'
- Yhtä putkea järkevää käyttää vain yhteen suuntaan
  - tarpeettomat putken päät suljetaan sekä emo- että lapsiprosessissa
- Kaksisuuntaiseen kommunikointiin tarvitaan kaksi putkea


```
#define N 3

int add_vector(int v[]) { // yhden rivin summa
  int i, sum = 0;
  for (i=0; i < N; i++) sum += v[i];
  return sum;
}
int main() {
  int a[N][N] = {{1, 1, 1},{2, 2, 2},{3, 3, 3}};
  int i, row_sum, sum = 0;
  int fd[2];

  pipe(fd);
  for (i = 0; i < N; i++) // luo N lapsiprosessia
     if (fork() == 0) {
         /* lapsi */
         close(fd[0]); // ei lue putkesta
         row_sum = add_vector(a[i]);
         write(fd[1], &row_sum, sizeof(int));
         exit(EXIT_SUCCESS);
     }
   /* emo */
   close(fd[1]); // ei kirjoita putkeen
   for (i=0; i < N; i++) {
       read(fd[0], &row_sum, sizeof(int));
       printf("   Row sum = %d\n",row_sum);
       sum += row_sum;
   }
   printf("Sum of the array = %d\n",sum);
}
```

```
$ ./rows
   Row sum = 3
   Row sum = 9
   Row sum = 6
Sum of the array = 18
```



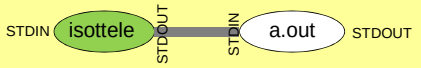
- Putkeen mahtuu kerrallaan dataa vain rajallinen määrä
  - riippuu järjestelmäasetuksista
  - Linuxissa oletus 64 KB
- Synkronointi ja odotus
  - read + tyhjä putki odota kunnes tulee luettavaa
  - write + täysi putki odota kunnes putkessa tilaa (ts. kunnes joku lukee)
  - write + osa sopii kirjoita se mikä sopii ja odota kunnes tilaa
- Jos putken write-pää on suljettu
  - read palauttaa putkessa olevat tavut
  - seuraava read palauttaa EOF (eli 0 luettua merkkiä)
- Jos putken read-pää on suljettu
  - write palauttaa -1 ja errno=EPIPE
  - lisäksi prosessi saa signaalin SIGPIPE

- Kirjastofunktio
```
#include <stdio.h>
FILE *popen(const char *cmd, const char * mode)
```
  luo putken, käynnistää lapsiprosessin sekä vaihtaa lapsiprosessin koodin, mode määrää mitenpäin putki "kuljettaa"
- On siis yhdistelmä pipe-, fork-, exec- ja dup2-kutsuista
- cmd
  - laita lapsiprosessi suorittamaan tätä kooditiedostoa
  - cmd:n jokerimerkit evaluoidaan ennen suoritusta, esim.
    `fp = popen("ls *.c", "r");`
- mode kertoo kirjoittaako vai lukeeko itse (toinen pää suljetaan itseltä)
  - "r" emo avaa itselle lukemista varten
  - "w" emo avaa itselle kirjoittamista varten
- Järjestelee lapsen tiedostokuvaajia
  - "r": kun lapsi kirjoittaa stdoutiin, niin merkit menevät putkeen (emolle)
  - "w": kun lapsi lukee stdinistä, tieto tuleekin putkesta (emolta)
- putken voi sulkea funktiolla `pclose()`
  - odottaa, kunnes lapsiprosessi on päättynyt, ja palauttaa sen paluuarvon


```
/* isottele.c */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {

  char c; int n;

  while ( (n = read(STDIN_FILENO, &c, 1)) != 0) {
     if (islower(c))
        c = toupper(c);
     if (write(STDOUT_FILENO, &c, n) != n) {
        perror("can't write to STDOUT_FILENO");
        exit(EXIT_FAILURE);
     }
  }
  exit(EXIT_SUCCESS);
}
```

```
$ gcc isottele.c -o isottele
$ ./isottele
pikkukirjaimet isoksi
PIKKUKIRJAIMET ISOKSI
toinen syöterivi
TOINEN SYÖTERIVI
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MAXLEN 80
void isottele(void) {
  FILE *fpin;
  // popen(): luo putki, luo lapsiprosessi, vaihda sen koodi, "r"=emo lukee,
  // joten järjestele tdstokuvaajia s.e kun lapsi kirjoittaa STDOUTiin, se meneekin putkeen
  if ((fpin = popen("./isottele", "r")) == NULL) {
     perror("popen error");
     exit(EXIT_FAILURE);
  }
  dup2(fileno(fpin),STDIN_FILENO);
  close(fileno(fpin));
}
int main(int argc, char *argv[]) {
  char line[MAXLEN]; int n;

  isottele();

  for ( ; ; ) {
    write(STDOUT_FILENO, "prompt> ",8));
     // kun äiti lukee, STDIN:stä, tieto tuleekin putkesta
    if ((n = read(STDIN_FILENO, line, MAXLEN)) == 0)
       break;
    write(STDOUT_FILENO, line, n);
  }
  exit(EXIT_SUCCESS);
}
```



- Käyttää ed. sivun isottele-ohjelmaa esikäsittelijäprosessina
- Näppäinsyöte menee aina isottele-ohjelmalle, sen stdout-tulostus menee putkeen
- Tämä ohjelma saakin dup2()-kutsun jälkeen stdinistä vain isoja merkkejä (jotka tulevat siis putkesta)

```
$ /a.out
prompt> pikkuisia
PIKKUISIA
prompt> isommiksi
ISOMMIKSI
prompt>
```


Nimetty putki (FIFO)

- Luodaan tavallisen tiedoston luonnin tapaan systeemikutsulla

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

  eli annetaan tiedostonimi ja käyttöoikeudet
- Nimettyä putkea edustaa tiedostojärjestelmässä nimetty fyysinen tiedosto
  - nimi ja käyttöoikeudet, kuten muillakin tiedostoilla
  - muutkin prosessit voivat lukea / kirjoittaa käyttöoikeuksien rajoissa
    - toisinkuin nimetön putki, joka vain emo- ja lapsiprossin välinen
- Käyttö ei poikkea muiden tiedostojen käytöstä
  - open(), read(), write(), close(), ...
  - hävitetään funktioilla unlink() tai remove()
- FIFO:a voi käyttää esimerkiksi erilaisten konekohtaisten asiakas-palvelin ohjelmien toteuttamisen osana
  - kaksisuuntaiseen kommunikointiin tarvitaan kaksi nimettyä putkea

Prosessi A (kirjoittaja):

```c
mkfifo("/tmp/putki", S_IRWXU|S_IRGRP|S_IROTH);

fd = open("/tmp/putki", O_WRONLY);
write(fd, "Halloota hoo", 12);
write(fd, " siellä!", 8);
close(fd);
unlink("/tmp/putki");
```

Prosessi B (lukija):

```c
fd = open("/tmp/putki", O_RDONLY);
n = read(fd, buf, BUFSIZE);
printf("%s\n", buf);
```


PROSESSI ja sen SÄIKEET

Prosessi perinteisesti (process, task)

- Resurssien omistaja, jolle allokoitu
  - suoritusympäristö muistista, eli virtuaaliosoiteavaruus
    - prosessin kuva (image): prosessin kuvaaja, koodi, data, pino
  - resursseja
    - muistia, tiedostoja, I/O-laitteita ...
- Vuorottajan hallinnoima kokonaisuus
  - prosessi on ohjelman suoritus koneessa
  - suoritus limittäin muiden prosessien kanssa
  - prosessiin liittyy tila sekä prioriteetti
  - vuorottaminen = suoritettavan prosessin vaihto

Prosessi nykyaikaisesti

- Prosessi on resurssien omistaja, kirjanpidon yksikkö
  - muistitilan varaus prosessin koodille
  - laitteiden varaus
- Prosessi on pääsyoikeuksien ja suojauksen yksikkö
  - muistinsuojaus
  - tiedostot ja niiden käyttöoikeudet
  - prosessien välinen kommunikointi

mutta

- Prosessilla on monta limittäin etenevää suoritusta, säiettä
- Vuorottamisen hallinnoima yksikkö on säie (thread, lightweight process)
  - kukin säie suorittaa omaa koodiosaansa
  - vuorottaminen = suoritettavan säikeen vaihto
- Kun yksi säie odottaa, voi CPU suorittaa jotain muuta saman prosessin säiettä
  - jokaisella säikeellä oma tila (Running, Ready, Blocked...)
  - jokaisella säikeellä oma tallealue CPU:n rekistereille

Säikeistä tarkemmin myöhemmin.

Kertauskysymyksiä

- Miten uusi prosessi syntyy?
- Milloin KJ vaihtaa suoritettavaa prosessia?
- Miten KJ vaihtaa suoritettavaa prosessia?
- Mitä tietoja on prosessin kuvaajassa?
- Mikä on säie?