exeçl ei võimalda mugavusi, mis on normaalse käsu korral:
• automaatset otsingut mitmes kataloogis
• ei saa kasutada a-de maskeerimist
kui on soov neid kasutada, siis tuleb pöörduda shell'i poole, mis siis vastava töö ära teeb
moodusta **käsurida**
execl("/bin/sh", "sh", "-c", käsurida, NULL); }
-c näitab, et järgmist a-i tuleb vaadelda kui käsku
execl ei suurenda protsesside arvu, ta ainult muudab protsessi sisu
protsesside arvu U-s suurendab süsteemne P ***fork***

fork

loob protsessi, mis on tema vanema täpne koopia, v.a. mõned parameetrid, mis teda identifitseerivad (protsessi id-#)
loodud protsessi nim. järglaseks, tema loojat aga vanemaks (vt. pilti)
fork ei hävita vanemat, pärast ***fork***'i hakkavad mõlemad võistlema ressursside pärast
harilikult kasutatakse ***fork***'i protsesside loomiseks, mis alustavad oma tööd pöördumisega ***execl***-i
kuna ***execl*** asendab vana P-i uuega, siis vana säilitamiseks tuleb temast teha koopia (1 asendatakse, 2. ootab kui uus P lõpetab töö)
seda teeb ***fork*** (kahvel), pöördumine:
prots_id = fork();
mõlemad jätkavad tööd, ainuke erinevus on protsessi #-s (järglase prots_id on 0)
seega põhivõte P'i väljakutsumiseks ja sealt tagasipöördumiseks on:

```
if (fork() == 0)
        execl ("/bin/sh", "sh", "-c", käsk, NULL); /* järglasesse */
```

vigade puudumisel on see piisav
fork teeb koopia, "järglases" tagastab ta '0'-i, kutsub välja ***execl***'i, mis täidab 'käsu' ja siis sureb

“vanemas“ fork tagastab nullist erineva arvu ja jätab ***execl***’i vahele vea korral tagastab ***fork*** ‘-1‘

```
#include <stdio.h>
main()
{
int pid;
        pid=fork();
        printf("protsessi nr on %d\n", pid);
}
$ a.out
protsessi nr on 7010
protsessi nr on 0

#include <stdio.h>
main()
{
int pid;
        pid=fork();
        if ( pid == 0)
          printf("protsessi nr on %d\n", pid);
        if (pid != 0)
          printf("protsi nr on %d\n", pid);
}
protsi nr on 7646
protsessi nr on 0
```

24.04.97 3.

konveierid (torud,pipes)

toru on I/O kanal, mis on mõeldud kasutamiseks 2 koos töötava protsessi korral, 1 kirjutab, 2. loeb
S sünkroniseerib ja kontrollib andmevahetust
toru moodustab harilikult shell - $ ls | pr
mõnikord on vaja seda ise teha
selleks ***pipe***, kuna teda kasut. nii lugemiseks kui kirjutamiseks, siis vaja 2 fd-t: fd[0] - lugemise fd[1] - kirjutamise
neid saab kasut. funktsioonides ***read***, ***write*** ja ***close***
tegelik klots:

```
int fd[2];

stat = pipe(fd);
if (stat == -1)
        return(NULL);   /* viga  */
```

kui protsess loeb torust, mis on tühi, siis ta ootab, kuni andmed saabuvad, kui ta kirjutab torusse, mis on täis, siis ta ootab, kuni toru tühjeneb

näiteks. vaatame funktsiooni ***ppopen***(kask, olek), mis loob protsessi kask (täpselt nii, nagu ***system*** teeb) ja tagastab F'i i-node'i, mida saab kasut. sinna protsessi kirjutamiseks või sealt lugemiseks, vastavalt olek'u väärtusele; seega pöördumine

```
fout = ppopen("pr", WRITE);
```

loob protsessi, mis teostab käsku ***pr***
järgnev pöördumised ***write***-i (kasutades fout'i) saadavad oma andmed sinna protsessi läbi toru, kasutades F'i i-node'i fout

ppopen loob esiteks toru (pipe), luuakse uus protsess (fork), …
NB! close'd on olulised, et EOF-kontrollid töötaksid õigesti

24.04.97 4.

```
# include <stdio.h>
# define READ   0
# define WRITE  1
# define tst(a, b)      (olek == READ ? (b) : (a))
static   int pid;

ppopen(kask, olek)
char  *kask;
int   olek;
{ int p[2];
   if (pipe(p) < 0)
        return(NULL);

if ((pid = fork()) == 0)       {
        close(tst(p[WRITE], p[READ]));
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
        close(tst(p[READ], p[WRITE]));
        execl("/bin/sh", "sh", "-c", kask, 0);
        _exit(1);   /*  oennetus, kui sinna sattusime  */
   }

if (pid == -1)
   return(NULL);
   close(tst(p[READ], p[WRITE]));
   return(tst(p[WRITE], p[READ]));
}
```

harilikult “vanem“ tahab oodata “järglase“ töö lõppu enne jätkamist
selleks ***wait***

```
int status;
if (fork() == 0)
        execl (… );
wait (&status);
```

wait ootab protsessi lõppu
‘status‘ tagastab “järglase“ lõppenud protsessi lõpetamise oleku
0 - normaalne lõpp, …
seda ”klotsi” kasutab ***system***
on soovitav kõikide P-de puhul tagastada selge lõpetamise olek

signaalid

signaal **-** kokkulepitud väärtus, mis antakse 1-st protsessist 2.-e
näit. SIGKILL - selle signaali saanud protsess lõpetab töö
U-s kasutatakse kümmekonda standartset signaali
vaatame ainult väljast tulnud signaale (sõrmistikult, kill, liinid)
Sun’is: genereerib Ctrl+C signaali ***kill*** ja Ctrl+\ - ***quit***
nendel juhtudel saadetakse signaal kõikidele protsessidele, mis
käivitati antud terminalilt, kõik protsessid peatatakse kuni korralduseni
signal - P, mis muudab ära vaikimisi määratud tegevuse

```
#include <signal.h>
signal(signal, sigcode);
```

signal - signaal (numbriline kood)
sigcode - tegevus (funktsioon või kood)

```
signal (SIGINT, SIG_IGN);        - ignoreeri katkestusi
```

signal tagastab signaali eelmise väärtuse
harilikult kasut. nii, et kustutatakse lõpetamata tegevuse tagajärjed,
s.t. taastatakse olukord, kui asi oli OK (kustutab ajutise F)

24.04.97 6.

```
# include <time.h>
# include <sys/types.h>
# include <unistd.h>
# include <signal.h>
main()
{       int onintr();
        if(signal(SIGINT, SIG_IGN) != SIG_IGN)
          signal(SIGINT, onintr);
        /* lase edasi!  */
        exit(0);       }
onintr()
{       unlink(tempfile);
        exit(1);       }
```

signal tagastab antud signaali eelmise oleku
kontrollib katkestuste käsitlemist; kui enne katkestusi ignoreeriti, siis tehakse seda edasi; vastasel korral püütakse katkestus-signaal kinni
kill - signaalide saatmiseks 1-st protsessist teise
exit - katkestab antud protsessi
antakse automaatselt ***main***'i lõpus või mõnede signaalide korral

vigadest

kõik P-d, mis pöörduvad otse S-i poole tagastavad vea korral '-1'
vea # on external-muutujas **errno**
P perror (**errno**) väljastab vea põhjuse