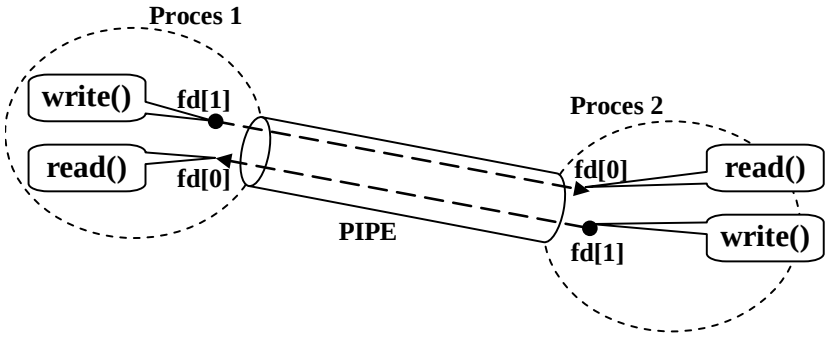
Mecanisme de comunicare intre procese (IPC[1])

1. Comunicatia prin PIPE

Acest mecanism ofera suport pentru comunicatii bidirectionale intre mai multe procese aflate in relatie tata-fiu sau care au un stramos comun (ptr ca prin fork(), acestea pot partaja aceleasi intrari in tabela de fisiere din kernel).
Conceptual comunicatia este realizata prin intermediul unui fisier de tip PIPE, conform politicii FIFO. De regula, fisierul PIPE este creat de catre procesul tata/stramos prin apelul functiei **pipe(),** inainte de apelul functiei fork(). Odata creat un PIPE, se pot realiza operatii de scriere/citire prin intermediul functiilor **read()/write()** si a doi descriptori de fisier:

fd[0] = descriptor de fisier in citire;
fd[1] = descriptor de fisier in scriere



Conceptul de comunicare prin PIPE

Sincronizarea intre procesul care scrie si cel care citeste dintr-un PIPE este minimala, conform principiului producator-consumator: scrierea este blocata daca nu exista spatiu in PIPE; procesul care citeste se blocheaza daca nu exista date in PIPE.
Mecanismul de sincronizare poate fi dezactivat data este specificata optiunea O_NONBLOCK atunci cand sunt apelate functiile read()/write().

Observatii:

- pentru ca ambii descriptori de fisier fac referinta la acelasi i-node, apar probleme locale de sincronizare. O solutie este de a inchide unul dintre acesti descriptori; ca urmare, orice operatie realizata pe unul dintre descriptori pentru care corespondentul sau a fost inchis, se soldeaza cu un rezultat de eroare;
- in general, mecanismul PIPE este utilizat pentru comunicatii intre doua procese;

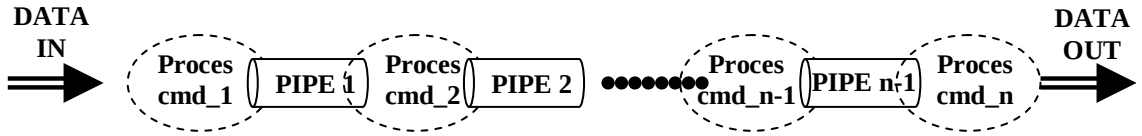
[1] en. Inter-Process Communication

- odata datele citite, acestea sunt extrase din fisierul PIPE;
- dimensiunea unui PIPE este de maxim 10 blocuri de date

Interfata Shell ofera suport pentru a realiza comunicatii prin PIPE intre comenzi(procese). In acest mod, este posibil ca o problema complexa sa fie rezolvata prin descompunere in mai multe procese simple care opereaza secvential, asfel:

```
$ cmd_1 | cmd_2 | cmd_3 | ... | cmd_n
```

Conceptual este echivalent cu:



Procesarea in cascada a datelor IN folosind mecanismul PIPE

Unde, **PIPE i** reprezinta fisier de iesire pentru procesul (i) si fisier de intrare pentru procesul (i+1).

Exercitiul 1.
Se vor afisa atributele contului utilizatorului "root".

```
$ cat /etc/passwd | grep root
```

Def: se mai numeste filtru, un program care citeste intrarea standard si afiseaza rezultatul la iesirea standard.

Obs: in general, utilizarea unui filtru in scrierea unui PIPE determina aparitia unui blocaj. Solutii: punerea in asteptare a procesului parinte pana la terminarea procesului fiu; alternativ, utilizarea a doua PIPE-uri, unul pentru scriere, celalalt pentru citire.

Exercitiul 2.
Sa se scrie un program care sa afiseze pe doua coloane toate interfetele si adresele lor IP. Se va utiliza comanda "ifconfig -a".
Indicatie: se va utiliza functia **dup2(<old_fd>, <new_fd>)** pentru a duplica descriptorul de fisier in citire corespunzator fisierului PIPE. Se va folosi fisierul sursa "ipc_pipe.c".

Prototip functie	Semnificatie
#include <unistd.h> **int pipe (int fd[2]);**	RETURNS: 0 on success -1 on error: errno Unde errno = EMFILE (no free descriptors) EMFILE (system file table is full) EFAULT (fd array is not valid) NOTES: fd[0] is set up for reading, fd[1] is set up for writing

#include <stdio.h> **FILE *popen(const char* cmd, const char *type);**	RETURNS: new file stream on success NULL on unsuccessful fork() or pipe() call NOTES: creates a pipe, and performs fork/exec operations using "command"
int pclose(FILE *fd);	NOTES: waits on the pipe process to terminate, then closes the stream.

Functia popen()
Frecvent in practica este necesara prelucrarea intr-un mod particular a datelor de iesire a unor comenzii din distributia de baza a unui sistem de operare. Pentru a realiza acest lucru se va crea un proces fiu in care se va rula comanda, si un PIPE prin care se va realiza transferul datelor de iesire ale comenzii catre procesul parinte unde vor fi procesate. In acest scop, functia **popen()** poate automatiza procedura.

Exercitiul 3.
Se va relua exercitiul de mai sus.
Indicatie: de data aceasta folosind functia **popen()**. A se vedea exemplul din fisierul sursa "ipc_ppipe.c".

Exercitiul 4.
Sa se implementeze o aplicatie care sa realizeze o tranzactie ICMP ECHO REQ/RPY pentru un set predefinit de adrese IP, si care sa afiseze rezultatul pe consola.

2. Comunicatia prin PIPE cu nume

Acest mecanism reprezinta o combinatie dintre un fisier ordinar si mecanismul PIPE. Astfel, fisierelor PIPE cu nume li se atribuie un nume si sunt localizate intr-un director, in mod similar unui fisier ordinar. De asemenea, sunt pastrate toate caracteristicile fisierelor PIPE.
Avantajul acestui mecanism este ca descriporii de fisier nu trebuie sa fie transferati intre procesele corespondente. Acest lucru permite realizarea unor comunicatii intre procese care nu se gasesc in relatie tata-fiu sau care nu au un stramos comun.
Pentru a putea comunica procesele nu trebuie sa cunoasca decat, numele fisierului PIPE.
Obs: fisierele PIPE cu nume pot fi recunoscute avand valoarea "p" pentru atributul de tip de fisier.
Crearea fisierului PIPE cu nume este realizata prin intermediul functiei **mknod(<filename>, SFIFO | <perm_access>),** apelata odata de catre procesul server. Procesele corespondente trebuie, in prealabil comunicatiei, sa deschida fisierul PIPE cu nume, lucru realizat analog unui fisier odinar, prin intermediul functiei **open()**. In mod implicit, operatia de deschidere in scriere/citire este blocata pana cand procesul corespondent face o deschidere in citire/scriere. Alternativ, acest comportament poate fi modificat cu ajutorul optiunii **O_NONBLOCK**.

Comunicatia este realizata efectiv prin intermediul functiilor **read()/write()**. In mod implicit, comportamentul acestor operatii este blocant. Alternativ, acest comportament poate fi modificat cu ajutorul optiunii **O_NONBLOCK**.
In final, fisierul PIPE cu nume pate fi inchis, in mod analog unui fisier ordinar, prin intermediul functiei **close()**. Odata inchis, procesul care citeste primeste caracterul sfarsit de fisier (EOF). Stergerea unui fisier PIPE cu nume se poate realiza cu ajutorul functiei **unlink()**.

Prototip functie	Semnificatie
#include <sys/types.h> #include <sys/stat.h> #include <sys/fnctl.h> **int mknod (const char *path, mode_t mode, dev_t dev);**	RETURNS: 0 on success, -1 on error: errno Unde errno = EFAULT(pathname invalid) EACCES (permission denied) ENAMETOOLONG (pathname too long) ENOENT (invalid pathname) ENOTDIR (invalid pathname) (see man page for mknod for others) NOTES: Creates a filesystem node (file, device file, or FIFO)
#include <unistd.h> **ssize_t read (int fd, void *buf, size_t count);**	man 2 read
ssize_t write (int fd, const void *buf, size_t count);	man 2 read

Exercitiul 5.
Sa se scrie o aplicatie client-server pentru adunarea a doua numere intregi. Clientul va transmite o cerere catre server care contine cele doua numere de adunat, serverul le va aduna si va transmite inapoi catre client rezultatul.
Indicatie: se vor folosi cele doua fisierele sursa “ipc_nfifo_cl.c” si “ipc_nfifo_srv.c”

3. Comunicatii prin coada de mesaje

Coada de mesaje este gestionata de kernel, are o structura de lista inlantuita si i se asociaza un identificator. Procesele vor identifica coada prin intermediul unei chei.
Sincronizarea intre procesele care scriu si cele care citesc mesajele se face pe principiul producator-consumator, analog mecanismului PIPE.
Mecanismul de sincronizare poate fi dezactivat data este specificata optiunea IPC_NOWAIT atunci cand sunt apelate functiile de scriere/citire.
Daca coada de mesaje nu este creata inca, procesul care vrea sa transmita un mesaj va apela mai intai functia **msgget(KEY, IPC_CREAT | <perm_access>)** avand ca argument cheia asociata cozii, apoi va putea scrie mesajul in coada prin intermediul functiei **msgsnd()**. Procesul care va trebui sa receptioneze mesajele va apela mai intai aceasi functie msgget(KEY, 0), dupa care va citi mesajele prin apelul functiei **msgrcv()**. Citirea mesajelor dintr-o coada se poate face si altfel decat dupa regula FIFO; astfel, se pot citi numai mesajele a caror valoare tip corespunde valorii precizate prin intermediul argumentului functiei msgrcv().

Mesajele transferate intre procese trebuie sa aiba urmatoare structura:

```
struct msg {
        long type;
        mesaj [256];
};
```

Unde, campul **type** precizeaza tipul mesajului (valoarea zero specifica preluarea mesajelor in ordinea FIFO), iar campul **mesaj** contine mesajul propriu-zis (a carui structura nu este specificata).

Pentru a sterge o coada de mesaje este utilizata functia **msgctl().**

Prototip functie	Semnificatie
#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h> **int msget (key_t cheie, int ind);**	RETURNS: message queue identifier on success -1 on error: errno Unde errno = EACCESS (permission denied) EEXIST (Queue exists, cannot create) EIDRM (Queue is marked for deletion) ENOENT (Queue does not exist) ENOMEM (Not enough memory to create queue) ENOSPC (Maximum queue limit exceeded)
int msgctl (int msgid, int cmd, struct msgid_ds *buffer);	RETURNS: 0 on success -1 on error: errno Unde errno = EAGAIN (queue is full, and IPC_NOWAIT was asserted) EACCES (permission denied, no write permission) EFAULT (msgp address isn't accessable - invalid) EIDRM (The message queue has been removed) EINTR (Received a signal while waiting to write) EINVAL (Invalid message queue identifier, nonpositive message type, or invalid message size) ENOMEM (Not enough memory to copy message buffer)
int msgsndt (int msgid, const void *p, size_t nbytes, int ind);	http://tldp.org/LDP/lpg/node35.htm
int msgrcv(int msgid, void *p size_t nbyes, long tip, int ind);	RETURNS: Number of bytes copied into message buffer -1 on error: errno Unde errnod = E2BIG (Message length is greater than msgsz, no MSG_NOERROR) EACCES (No read permission) EFAULT (Address pointed to by msgp is invalid) EIDRM (Queue was removed during retrieval) EINTR (Interrupted by arriving signal)

	EINVAL (msgqid invalid, or msgsz less than 0) ENOMSG (IPC_NOWAIT asserted, and no message exists in the queue to satisfy the request)

Exercitiul 6.
Sa se scrie o aplicatie client-server pentru adunarea a doua numere intregi. Clientul va transmite o cerere catre server care contine cele doua numere de adunat, serverul le va aduna si va transmite inapoi catre client rezultatul.
Indicatie: se vor folosi cele doua fisierele sursa “ipc_mq_cl.c” si “ipc_mq_srv.c”

4. Comunicatia prin memoria comuna partajata

Prototip functie	Semnificatie
#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h> **int shmget (key_t cheie, int dim, int ind);**	http://tldp.org/LDP/lpg/node35.htm
int shmctl (int shmid, int cmd, struct shmid_ds *buffer);	http://tldp.org/LDP/lpg/node35.htm
char * shmat (int shmid, void *adr_mem, int ind);	http://tldp.org/LDP/lpg/node35.htm
int shmdt(void *adr_zona);	http://tldp.org/LDP/lpg/node35.htm

Exercitiul 7.
Sa se scrie o aplicatie client-server pentru adunarea a doua numere intregi. Clientul va transmite o cerere catre server care contine cele doua numere de adunat, serverul le va aduna si va transmite inapoi catre client rezultatul.
Indicatie: se vor folosi cele doua fisierele sursa “ipc_shm_cl.c” si “ipc_shm_srv.c”