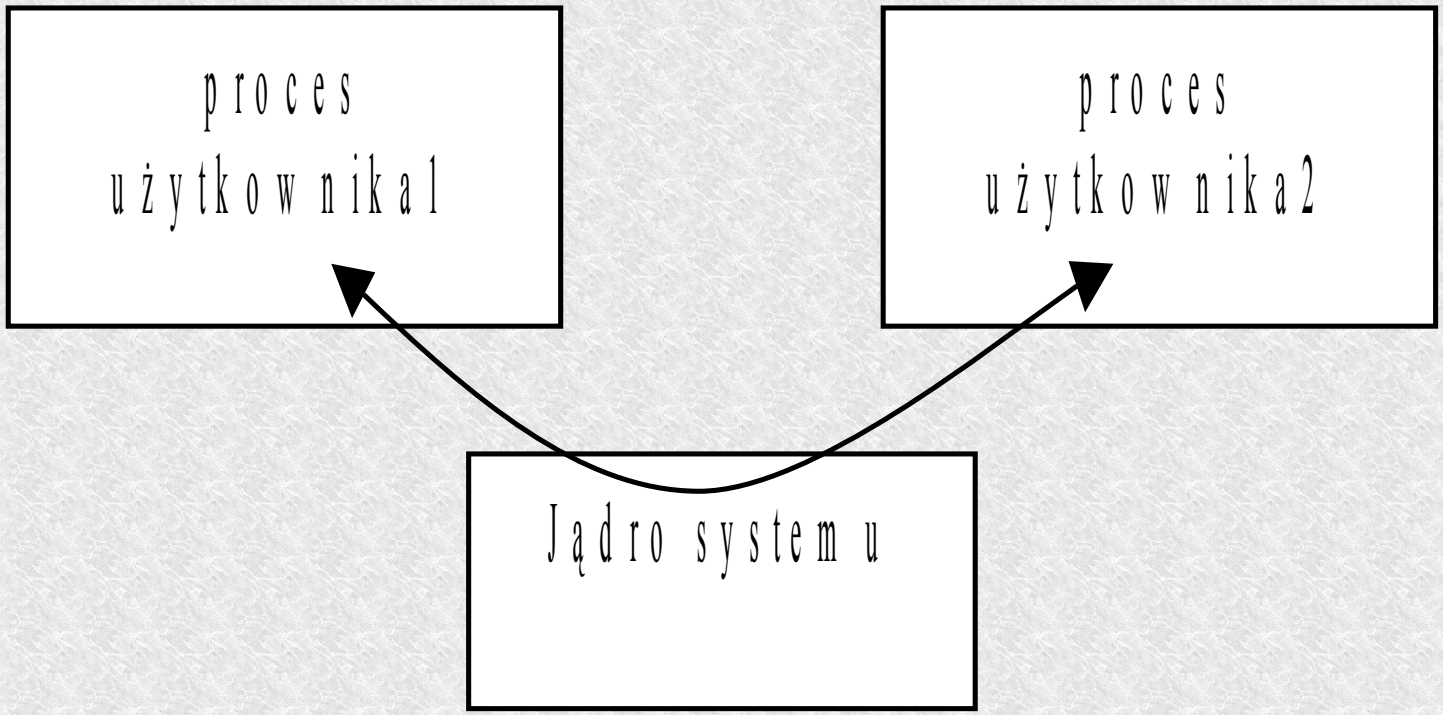
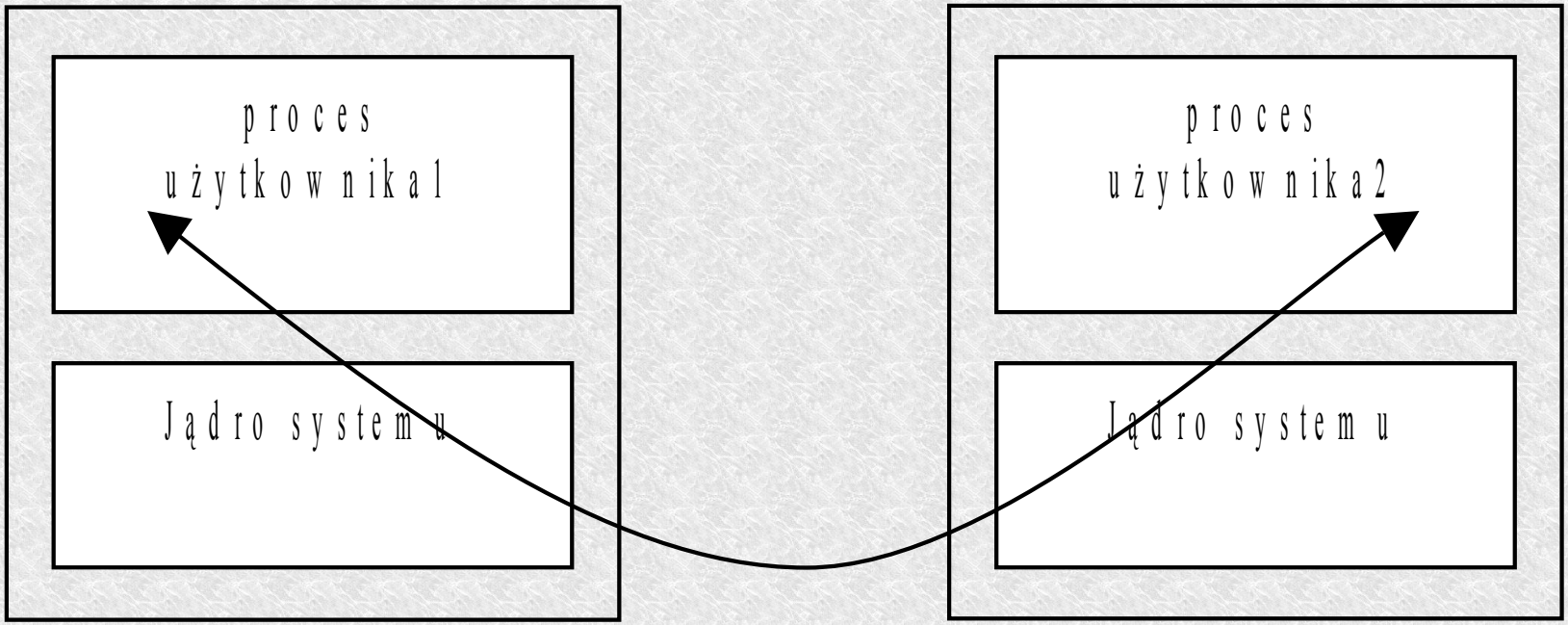
Programowanie Współbieżne

W Linuxie/Unixie

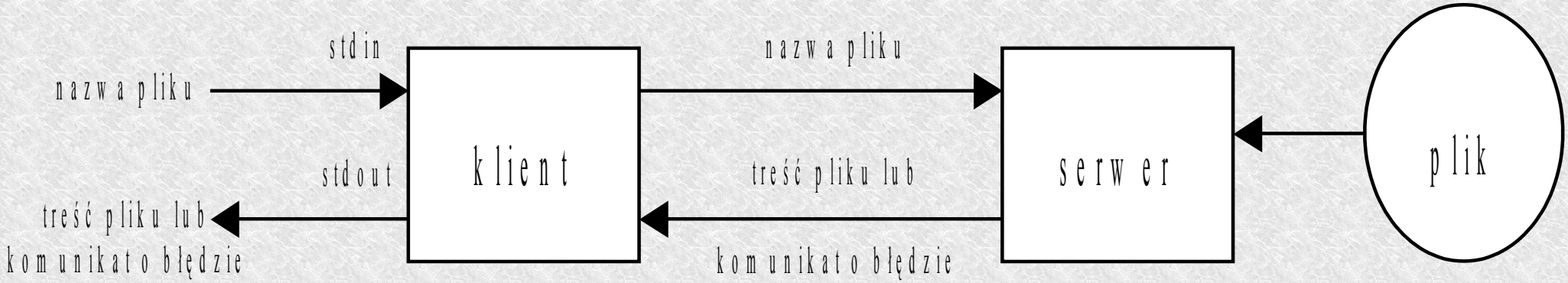
Komunikacja



Komunikacja



Komunikacja



- klient pobiera nazwę pliku ze standardowego wejścia
- przesyła ją do serwera
- serwer otwiera plik
- przesyła go klienta (lub info o błędzie)
- klient wyświetla zawartość na stdout

Łącza PIPE

- Łącza nie nazwane (PIPE) umożliwiają przepływ danych w jednym kierunku
- tworzymy za pomocą funkcji

```
int pipe(int *filedes);
```

- filedes jest tablica dwuelementową
- filedes[0] jest deskryptorem pliku otwartym do czytania
- filedes[1] jest deskryptorem pliku otwartym do pisana

Łącza PIPE

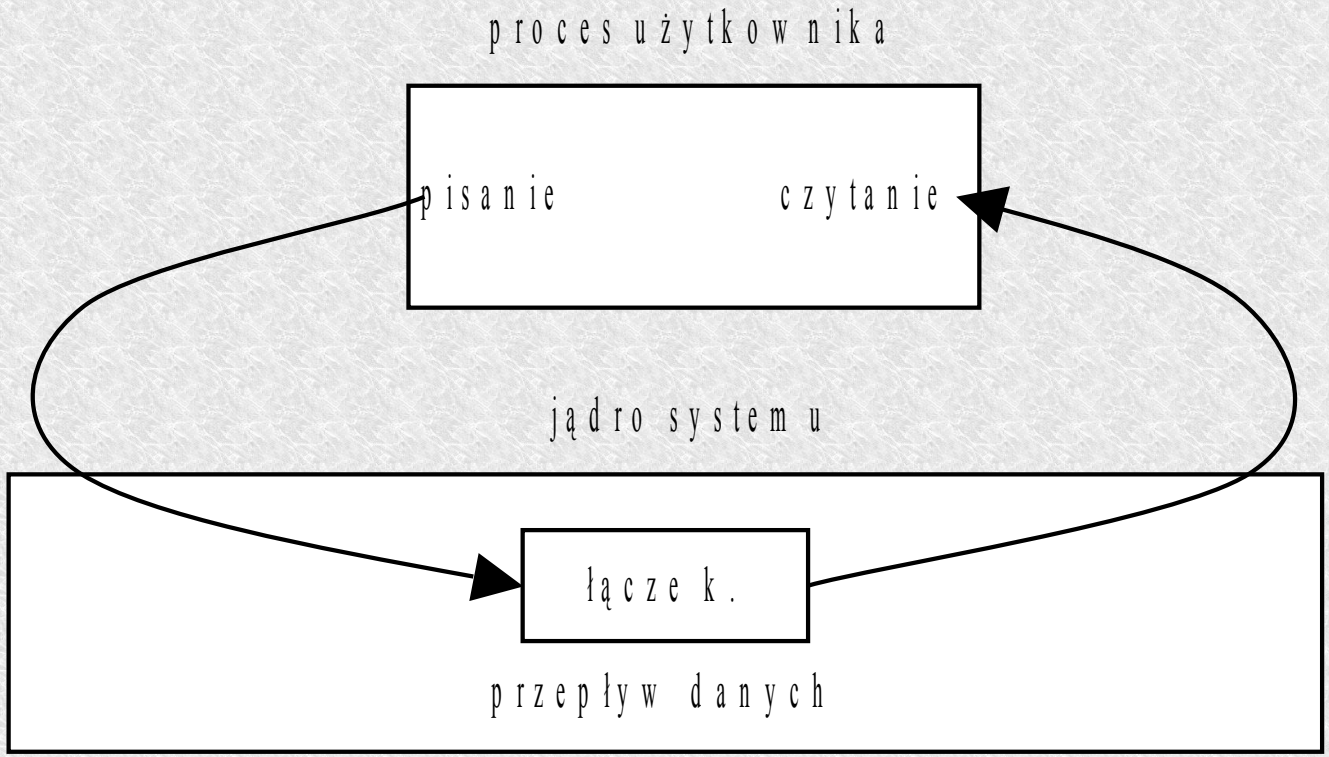
łącza komunikacyjne zwykle używamy do komunikowania się dwóch procesów, ale poniższy przykład będzie na jednym

```
main()
{
int pipefd[2],n;
char buff[100];
if (pipe(pipefd) <0)
    perror("blad pipe\n");
printf("read fd = %d, write fd = %d\n",pipefd[0],pipefd[1]);
if (write(pipefd[1],"hello world\n",12) !=12)
    perror("blad zapisu\n");
if ((n = read(pipefd[0],buff,sizeof(buff))) <=0)
    perror("blad odczytu\n");
write(1, buff, n); /* deskryptor pliku=1 wyjscie standardowe
*/
exit(0);
}
```

Łącza PIPE

może być tak że hello world będzie przed napisem read fd = ... itd. Ponieważ printf korzysta z bufora i jest on opróżniany pod koniec programu.

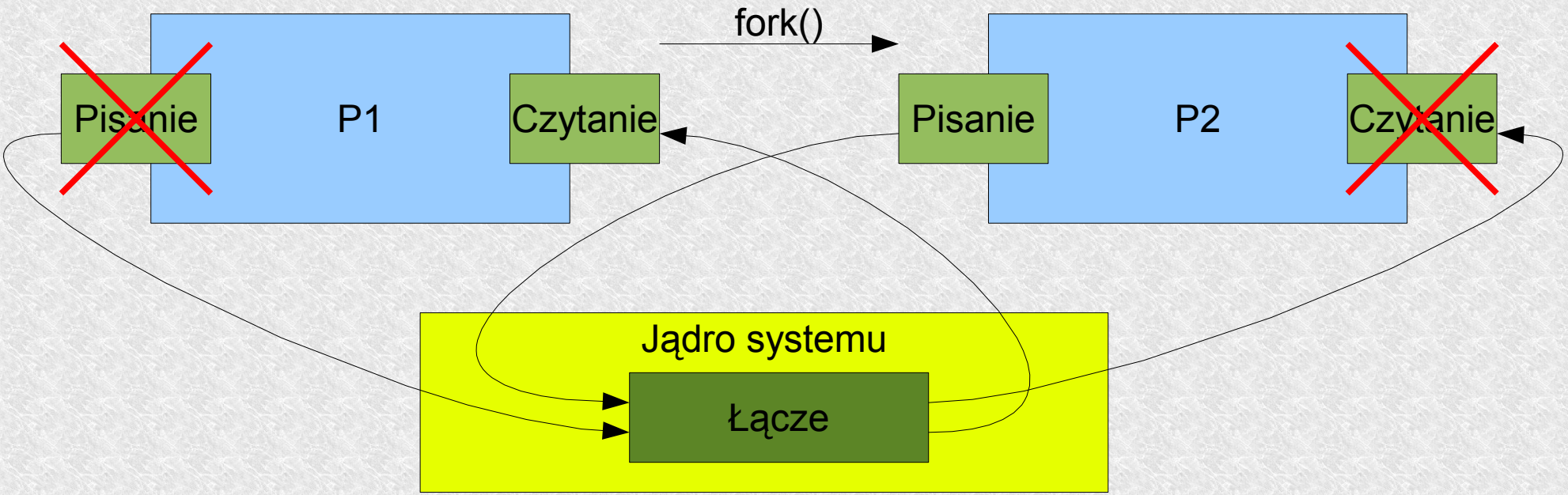
Stworzyliśmy mniej więcej takie łącze:



Łącza PIPE

By stworzyć łącze pomiędzy dwoma procesami musimy podążać następującym algorytmem

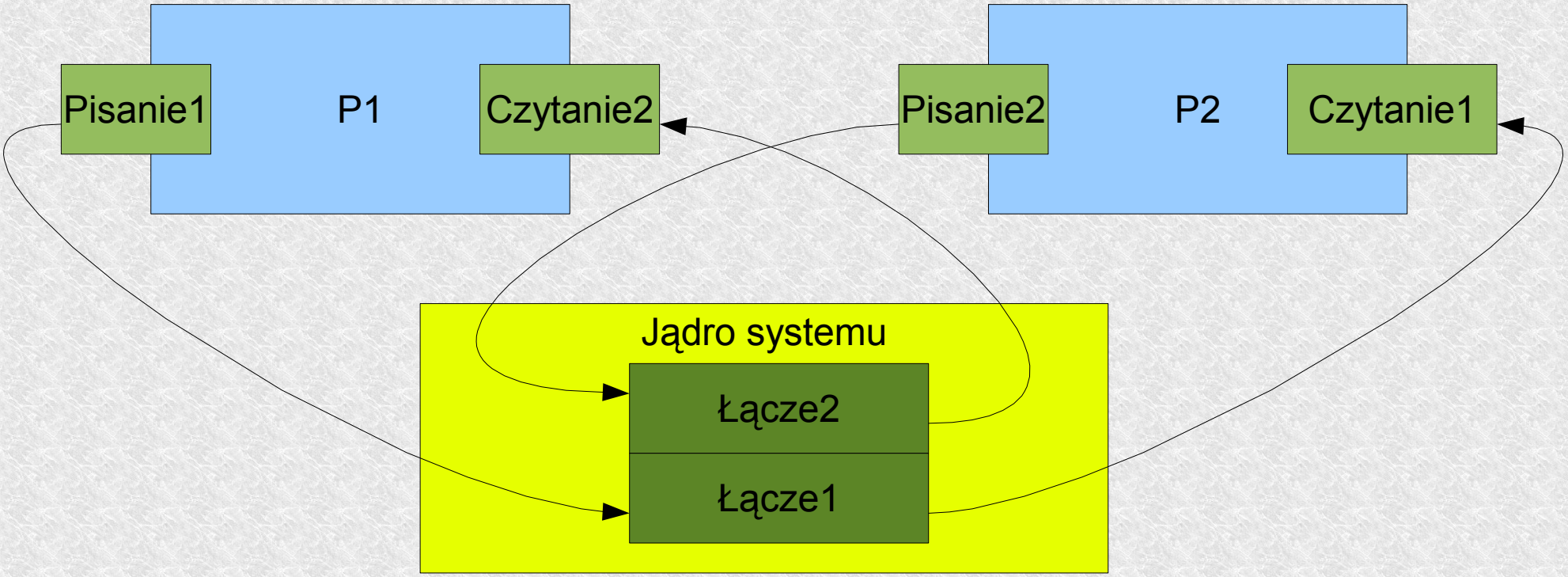
- proces macierzysty tworzy łącze
- następnie się forkuje, tu następuję skopiowanie deskryptorów
- proces macierzysty zamyka jeden deskryptor do pisania lub do czytania
- proces potomny zamyka jeden deskryptor do czytania lub do pisania



Łącza PIPE

łącza te są jednokierunkowe, jeżeli chcemy komunikację w dwóch kierunkach to trzeba wykonać następujące kroki

- utwórz dwa łącza (łącze1, łącze2)
- wywołaj fork
- przodek zamyka łącze 1 do czytania;
- przodek zamyka łącze 2 do pisania;
- potomek zamyka łącze 1 do pisania;
- potomek zamyka łącze 2 do czytania;



Łącza PIPE

standardowa bibliteka we/wy zawiera funkcje tworząca łącze komunikacyjne oraz inicjującą wykonywanie drugiego procesu:

```
#include <stdio.h>
FILE *popen(char *command, char *type);
```

- command oznacza wiersz polecenia
- typ ”r” lub ”w”
- zwraca wejście lub wyjście lub w razie nie powodzenia NULL

funkcja do zamknięcia strumienia otworzonego przez popen to:

```
#include <stdio.h>
int pclose(FILE *stream);
```

Łącza PIPE

```
#include <stdio.h>
#define MAXLINE 1024
main ()
{
char line[MAXLINE],command[MAXLINE+10];
int n;
FILE *fp;

/* pobierz nazwe pliku z wejscia standardowego */
if (fgets(line, MAXLINE, stdin) == NULL)
    perror("client: blad odczytu nazwy opliku\n");

sprintf(command, "cat %s",line);
if ((fp = popen(command, "r")) == NULL)
    perror("blad popen\n");
/* pobieraj dane z pliku i wysylaj je do wyjscia standardowego */
while ((fgets(line, MAXLINE,fp)) != NULL)
    {
    n = strlen(line);
    if (write(1, line, n) != n) /* fd 1 = stdout */
    perror("client: blad zapisu danych\n");
    }
if (ferror(fp)) /* sprawdza znacznik bledu  dla  strumienia  wskazywanego
       przez  stream, zwracaj niezerowa warto jesli jest on ustawiony */
    perror("blad fget\n");
pclose(fp);
exit(0);
}
```

Łącza PIPE

Największą wadą łącz PIPE jest to że możemy je tworzyć jedynie pomiędzy procesami spokrewnionymi.

Łącza (kolejki) FIFO

Inaczej łącza nazwane.

- Są podobne do łącz komunikacyjnych
- umożliwiają przepływ jednokierunkowy
- pierwszy bajt przeczytany z kolejki fifo będzie pierwszym który był tam wpisany.
- kolejka FIFO w odróżnieniu od łącza nienazwanego ma nazwę
- dzięki temu do tego samego łącza mogą mieć dostęp procesy niespokrewnione

Łącza (kolejki) FIFO

Do tworzenia kolejki FIFO służy funkcja

```
int mknod(char *pathname, int mode, int dev);
```

- pathname – normalna nazwa ścieżki
- mode – słowo trybu dostępu które będzie zsumowane logicznie ze znacznikiem S_IFIFO z pliku <sys/stat.h>
- dev – urządzenie, przy kolejkach FIFO można pominąć

można też za pomocą polecenia

```
/etc/mknod name p
```

gdy już powstanie trzeba ją otworzyć do czytania lub pisania np. przez open czy fopen, freopen.
Można ustawić odpowiedni znacznik O_NDELAY
Wynik ustawienia znacznika O_NDELAY mamy w tabelce:

Łącza (kolejki) FIFO

Sytuacja	Nie ustawiono znacznika O_NDELAY	ustawiono znacznik O_NDELAY
Otwórz kolejkę FIFO tylko do czytania, a żaden proces nie otworzył tej kolejki do pisania	Czekaj, aż proces otworzy kolejkę FIFO do pisania	powróć natychmiast bez sygnalizowania błędu
otwórz kolejkę FIFO tylko do pisania żaden proces nie otworzył tej kolejki do czytania	czekaj, aż proces otworzy kolejkę FIFO do czytania	powróć natychmiast, sygnalizuj błąd, w errno umieść stałą ENXIO
czytaj z łącza komunikacyjnego (lub kolejki FIFO), brak danych	czekaj, aż pojawią się dane w łączu (kolejce) lub nie będzie ono (ona) otarte do pisania dla żadnego procesu; jako wartość funkcji przekaż zero, jeżeli żaden proces nie otworzył łącza (kolejki) do pisania, inaczej przekaż liczbę danych	powróć natychmiast, przekaż zero jako wartość funkcji
pisz, łącze komunikacyjne (lub kolejka FIFO) zapełnione	czekaj, aż będzie miejsce, następnie pisz dane	powróć natychmiast, przekaż zero jako wartość funkcji

Łącza (kolejki) FIFO

Reguły:

- jeżeli proces zażąda przeczytania mniejszej ilość danych niż jest w łączu to przeczyta tyle ile chciał a resztę przy okazji
- jeżeli proces zażąda więcej danych niż jest to dostanie tyle ile jest w łączu.
- jeżeli w łączu nie ma danych, a żaden proces nie otworzył go do pisania to wartość read będzie zero, oznaczające koniec pliku.
- jeżeli proces zapisze mniejszą porcję danych niż wynosi pojemność łącza lub kolejki FIFO (zwykle 4096B) to niepodzielność danych będzie zagwarantowana
- jeżeli więcej to nie jest to zagwarantowane bo drugi proces piszący do tego samego łącza może przepleść dane
- jeżeli proces wywołuje write do łącza którego nie otworzył żaden inny proces do czytania
    - to dostanie on SIGPIPE
    - write zwróci 0
    - errno będzie miało wartość EPIPE
    - jeżeli proces nie obsłuży sygnału SIGPIPE to zostanie zakończony

Łącza (kolejki) FIFO

Wyobraźmy sobie demona czekającego na dane od klientów na jakimś łączu otwartym do czytania, gdy klient skończy pisanie i zakończy się przez np. exit to łącze trzeba było by drugi raz otwierać i czekać.

- demon powinien otworzyć to samo łącze zarówno do czytania jak i do pisania
- do pisania nigdy nie użyje ale dzięki temu nie dostanie EOF z powodu braku klienta który by pisał.

Łącza (kolejki) FIFO

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;
#define FIFO1 "tmp/fifo.1"
#define FIFO2 "tmp/fifo.2"
#define PERMS 0666

main()
{
int readfd, writefd;
/* otworz kolejki FIFO przyjmujemy ze serwer juz je stworzyl */
if ((writefd = open(FIFO1, 1)) < 0)
    perror("Klient: nie moze otworzyc fifo1 do pisania");
if ((readfd = open(FIFO2, 0)) < 0)
    perror("Klient: nie moze otworzyc fifo2 do czytania");
client(readfd,writefd);
close(readfd);
close(writefd);
/* usun kolejki FIFO */
if (unlink(FIFO1) < 0)
    perror("Rodzic nie moze usunac FIFO1 %s",FIFO1);
if (unlink(FIFO2) < 0)
    perror("Rodzic nie moze usunac FIFO2 %s",FIFO2);
exit(0);
}
```

Łącza (kolejki) FIFO

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;
#define FIFO1 "tmp/fifo.1"
#define FIFO2 "tmp/fifo.2"
#define PERMS 0666
main()
{
int readfd, writefd;
/* utworz kolejki FIFO i nastepnie je otworz - jedna do czytania  */
/* druga do pisania */
if ((mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno !=EEXIST))
  perror("nie moge utworzyc fifo 1: %s\n", FIFO1);
if ((mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno !=EEXIST))
    {
    unlink(FIFO1);
    perror("nie moge utworzyc fifo 2: %s\n", FIFO2);
    }
if ((readfd = open(FIFO1 ,0)) < 0)
  perror("Serwer: nie moze otworzyc fifo1 do czytania");
if ((writefd = open(FIFO2,1)) < 0)
  perror("Serwer: nie moze otworzyc fifo2 do pisania");
server(readfd,writefd);
close(readfd);
close(writefd);
exit(0);
}
```

Strumienie danych a komunikaty

- W dotychczasowych przykładach używaliśmy pojęcia strumienia danych
- nie ma w nim żadnych wydzielonych rekordów czy struktur
- system w żaden sposób nie interpretuje jakie dane otrzymuje
- jeżeli chcemy interpretować to procesy czytający i piszący muszą ustalić wspólny sposób przekazywania danych
- najprostszą strukturą są wiersze zakończone ‘\n’
- można też wymyślić bardziej złożone struktury
- Struktura z typem i długością komunikatu. Dla dalszych przykładów strukturę tą umieścimy w pliku mesg.h:

Strumienie danych a komunikaty

```
#define MAXMESGDATA (4096-16)
#define MESGHDRSIZE (sizeof(Mesg) - MAXMESGDATA)

typedef struct {
   int mesg_len;
   long mesg_type;
   char mesg_data[MAXMESGDATA];
} Mesg;
```

Strumienie danych a komunikaty

```
#include „mesg.h” /* nasza struktura */

/* wyślij komunikat używając deskryptora pliku, pola
struktury musi wypełnić wcześniej proces wywołujący
*/
void mesg_send(int fd, Mesg *mesgptr)
{
int n;
/* przygotuj nagłówek */
n = MESGHDRSIZE + mesgptr->mesg_len;
if (write(fd, (char*) mesgptr, n ) !=n)
   perror(„blad zapisu komunikatu\n”);
}
```

Strumienie danych a komunikaty

```
/* pobierz komunikat, korzystając z deskryptora plików
wypełnij pola struktury Mesg i zwróć wartość pola mesg_len */

int mesg_recv(int fd, Mesg *mesgptr)
{
int n;
/* pobieramy nagłówek i sprawdzamy ile jeszcze do pobrania */
/* jeżeli EOF to zwracamy 0 */
if ((n = read(fd, (char*)mesgptr, MESGHDRSIZE)) == 0)
   return(0); /*koniec pliku */
else if (n != MESGHDRSIZE)
   perror(„blad odczytu nagłówka komunikatu \n”);
if ((n = mesgptr->mesg_len) > 0)
   if (read(fd, mesgptr->mesg_data, n) != n)
      perror(„blad odczytu danych komunikatu\n”);
return(n);
}
```

Strumienie danych a komunikaty

```
Mesg mesg;
void client(int ipcreadfd, int ipcwritefd)
{
int n;
if (fgets(mesg.mesg_data, MAXMESGDATA, stdin) == NULL)
    perror("blad odczytu nazwy pliku\n");
n = strlen(mesg.mesg_data);
if (mesg.mesg_data[n-1] == '\n')
    n--; /*pomin znak nowego wiersza pobierany przez fgets()
*/
mesg.mesg_len = n;
mesg.mesg_type = 1L;
mesg_send(ipcwritefd, &mesg);

while (( n = mesg_recv(ipcreadfd, &mesg)) > 0)
    if (write(1, mesg.mesg_data, n) != n)
   perror("blad zapisu nanych\n");
if (n<0)
    perror("blad odczytu danych\n");
}
```

Strumienie danych a komunikaty

```
extern int errno;

void server(int ipcreadfd, int ipcwritefd)
{
int n, filefd;
char errmesg[256];
mesg.mesg_type = 1L;
if ((n = mesg_recv(ipcreadfd,&mesgt)) <= 0)
    perror("server: blad odczytu nazwy pliku\n");
mesg.mesg_data[n] = '\0';
if ((filefd = open(mesg.mesg_data, 0)) < 0)
    {   /* blad, przygotuj komunikat o bledzie */
    sprintf(errmesg,":nie moge otworzyc %s\n",
     strerror(errno));
    strcat(mesg.mesg_data,errmesg);
    mesg.mesg_len = strlen(mesg.mesg_data);
    mesg_send(ipcwritefd,&mesg);
     }
else //...cdn
```

Strumienie danych a komunikaty

```
//...cd

    {
    while (( n = read(filefd, mesg.mesg_data, MAXMESGDATA)) > 0)
   {
   mesg.mesg_len = n;
   mesg_send(ipcwritefd, &mesg);
   }
    close(filefd);
    if (n < 0)
   perror("server: blad odczytu");
    }
/* wyslij pusty komunikat ktory oznacza zakonczenie przetwarzania */
mesg.mesg_len = 0;
mesg_send(ipcwritefd,&mesg);
}
```

Przestrzenie nazw

- łącza nie mają nazw ale kolejki można identyfikować za pomocą unixowych nazw ścieżek.
- zbiór możliwych nazw dla danego rodzaju komunikacji międzyprocesowej nazywamy **przestrzenią nazw**.
- wszystkie rodzaje komunikacji międzyprocesowej (z wyjątkiem łączy) odbywają się za pośrednictwem nazw.
- Zestawienie konwencji nazewniczych przyjętych dla różnych rodzajów komunikacji międzyprocesowej:

Przestrzenie nazw

Rodzaj komunikacji międzyprocesowej	Przestrzeń nazw	Identyfikacja
łącze komunikacyjne	(nie ma nazwy)	deskryptor pliku
kolejka FIFO	nazwa ścieżki	deskryptor pliku
kolejka komunikatów	klucz key_t	identyfikator
pamięć wspólna	klucz key_t	identyfikator
semafor	klucz key_t	identyfikator
gniazdo – dziedzina Unixa	nazwa ścieżki	deskryptor pliku
gniazdo – inna dziedzina	(w zależności od dziedziny)	deskryptor pliku

Klucze key_t

- są to identyfikatory zazwyczaj 32b określające jednoznacznie kolejki komunikatów, pamięć wspólną lub semafory.
- Można wymyślać samemu np 1234L
- Ale w poważniejszych zastosowaniach lepiej użyć:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

- na podstawie nazwy ścieżki i numeru projektu (proj jest liczbą 8b), tworzy nam prawie unikalny klucz. *(jest to liczba 32 bitowa a jest tworzona na podstawie i-węzła (32b) nr projektu (8b) i tzw. małego numeru urządzenia systemu pliku (8b))*

Klucze key_t

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

main()
{
key_t klucz;
klucz = ftok("/tmp/alamakota",4); //plik musi byc
bo jak nie to ftok zwroci -1
printf(" ftok %d\n",klucz);
}
```