Manual Lab 5 by Skynet

Komunikacja międzyprocesowa - cz. 1

Zakres przygotowania do zajęć:

- Łącza nienazwane i nazwane oraz ich obsługa.
- Mechanizmy powłoki: przekierowanie strumieni, łącza, potoki.
- Polecenia systemowe: *mknod*, *mkfifo*, *cat*, *tail*, *tee*
- Funkcje:
  - *open, read, write, close,*
  - *popen, pclose, pipe, mkfifo*
  - *dup, dup2, fcntl, select, mknod, fpathconf*

Pozdro dla Paziego :D I Karola za pomoc :D

Łącza komunikacyjne

Łącza w systemie UNIX są plikami specjalnymi, służącymi do komunikacji pomiędzy procesami. Łącza mają kilka cech typowych dla plików zwykłych, czyli posiadają swój i-węzeł, posiadają bloki z danymi (chociaż ograniczoną ich liczbę), na otwartych łączach można wykonywać operacje zapisu i odczytu. Łącza od plików zwykłych odróżniają następujące cechy:

- ograniczona liczba bloków — łącza mają rozmiar 4KB - 8KB w zależności od konkretnego systemu,
- dostęp sekwencyjny — na łączach można wykonywać tylko operacje zapisu i odczytu, nie można natomiast przemieszczać wskaźnika bieżącej pozycji (nie można wykonywać funkcji **lseek**),
- sposób wykonywania operacji zapisu i odczytu — dane odczytywane z łącza są zarazem usuwane (nie można ich odczytać ponownie), proces jest blokowany w funkcji **read** na pustym łączu i w funkcji **write**, jeśli w łączu nie ma wystarczającej ilości wolnego miejsca, żeby zmieścić zapisywany blok (wyjątkiem od tej zasady jest przypadek gdy jest ustawiona flaga O_NDELAY)

W systemie UNIX wyróżnia się dwa rodzaje łączy: **łącza nazwane** i **łącza nienazwane**. Zwyczajowo przyjęło się określać łącza nazwane terminem **kolejki FIFO**, a łącza nienazwane terminem **potoki**. Różnica pomiędzy łączem nazwanym i nienazwanym polega na tym, że pierwsze z nich ma dowiązanie w systemie plików (czyli istnieje jako plik w jakimś katalogu) i może być identyfikowane przez nazwę a drugie nie ma dowiązania i istnieje tak długo, jak długo jest otwarte. Po zamknięciu wszystkich deskryptorów łącze nienazwane przestaje istnieć i zwalniany jest jego i-węzeł oraz wszystkie bloki. Łącze nazwane natomiast po zamknięciu wszystkich deskryptorów w dalszym ciągu ma przydzielony i-węzeł, zwalniane są tylko bloki dyskowe. Jeżeli dwa procesy mają odpowiednie deskryptory łącza, to dla komunikacji między nimi nie ma znaczenia, czy są to deskryptory łącza nazwanego czy nienazwanego. Różnica jest natomiast w sposobie uzyskania deskryptorów łącza, która wynika z różnic w tworzeniu i otwieraniu łączy.

Ponieważ łącze nienazwane nie ma dowiązania w systemie plików, nie można go identyfikować przez nazwę. Jeśli procesy chcą się komunikować za pomocą takiego łącza, muszą znać jego deskryptory. Oznacza to, że procesy muszą uzyskać deskryptory tego samego łącza, nie znając jego nazwy. Jedynym sposobem przekazania informacji o łączu nienazwanym jest przekazanie jego deskryptorów procesom potomnym dzięki dziedziczeniu tablicy otwartych plików od swojego procesu macierzystego. Za pomocą łącza nienazwanego mogą się zatem komunikować procesy, z których jeden otworzył łącze nienazwane, a następnie utworzył pozostałe komunikujące się procesy, które w ten sposób otrzymają w tablicy otwartych plików deskryptory istniejącego łącza.

Operacje zapisu i odczytu na łączu nazwanym wykonuje się tak samo, jak na łączu nienazwanym, inaczej natomiast się je tworzy i otwiera. Łącze nazwane tworzy się poprzez wywołanie funkcji **mkfifo** w programie procesu lub przez wydanie polecenia **mkfifo** na terminalu. Funkcja **mkfifo** tworzy plik specjalny typu łącze

podobnie, jak funkcja **creat** tworzy plik zwykły. Funkcja **mkfifo** nie otwiera jednak łącza i tym samym nie przydziela deskryptorów. Łącze nazwane otwierane jest funkcją **open** podobnie jak plik zwykły, przy czym łącze musi zostać otwarte jednocześnie w trybie do zapisu i do odczytu przez dwa różne procesy. W przypadku wywołania funkcji **open** tylko w jednym z tych trybów proces zostanie zablokowany aż do momentu, gdy inny proces nie wywoła funkcji **open** w trybie komplementarnym.

Funkcje operujące na łączach nienazwanych zdefiniowane są w pliku unistd.h, natomiast funkcje używane w celu tworzenia łączy nazwanych zdefiniowane są w plikach `sys/types.h` oraz `sys/stat.h.`

Funkcje systemowe służące do tworzenia i komunikacji poprzez łącza nienazwane.

```
int pipe ( int pdesk[2] )
```

Wartości zwracane:

- poprawne wykonanie funkcji: 0
- zakończenie błędne: -1

Możliwe kody błędów (*errno*) w przypadku błędnego zakończenie funkcji:

- EMFILE - w procesie używanych jest zbyt wiele deskryptorów pliku
- ENFILE - tablica plików systemu jest pełna
- EFAULT - deskryptor *pdesk* jest nieprawidłowy

Argumenty funkcji:

- *pdesk*[0]- deskryptor potoku do odczytu
- *pdesk*[1]- deskryptor potoku do zapisu

UWAGI:

Funkcja tworzy parę sprzężonych deskryptorów pliku, wskazujących na inode potoku i umieszcza je w tablicy *pdesk*. Komunikacja przez łącze wymaga aby dwa procesy znały deskryptory tego samego łącza. Zatem proces, który utworzył potok może się przez niego komunikować tylko ze swoimi potomkami (niekoniecznie bezpośrednimi)lub przekazać im odpowiednie deskryptory, umożliwiając w ten sposób wzajemną komunikację. Dwa procesy z kolei mogą komunikować się przez potok wówczas, gdy mają wspólnego przodka (lub jeden z nich jest przodkiem drugiego), który utworzył potoką następnie odpowiednie procesy potomne, przekazując im w ten sposób deskryptory potoku. Jest to pewnym ograniczeniem zastosowania potoków.

Wielkość potoku zależy od konkretnej implementacji, faktyczną maksymalną liczbę bajtów można uzyskać przez funkcję fpathconf: **fpathconf(pdesk[0], _PC_PIPE_BUF).**

Gdy deskryptor pliku reprezentujący jeden koniec potoku zostaje zamknięty

- Dla deskryptora zapisu :
    - jeśli istnieją inne procesy mające potok otwarty do zapisu nie dzieje się nic
    - gdy nie ma więcej procesów a potok jest pusty, procesy, które czekały na odczyt z potoku zostają obudzone a ich funkcje **read** zwrócą 0 (wygląda to tak jak osiągnięcie końca pliku)

- Dla deskryptora odczytu :
    - jeśli istnieją inne procesy mające potok otwarty do odczytu nie dzieje się nic
    - gdy żaden proces nie czyta, do wszystkich procesów czekających na zapis zostaje wysłany sygnał SIGPIPE.

`int read(inf fd, void *buf, size_t count)`

Wartości zwracane:

- poprawne wykonanie funkcji: *rzeczywista liczba bajtów, jaką udało się odczytać*
- zakończenie błędne: -1

Argumenty funkcji:

- *fd* - deskryptor potoku z którego mają zostać odczytane dane
- *buf* - adres bufora znajdującego się w segmencie danych procesu, do którego zostaną przekazane dane odczytane z potoku w wyniku wywołania funkcji **read**
- *count* - ilość bajtów do odczytania

UWAGI:

Jeśli wszystkie deskryptory do zapisu są zamknięte i łącze jest puste, to zostaje zwrócona wartość 0.

Różnica w działaniu funkcji **read** na łączu i na pliku zwykłym polega na tym, że dane odczytane z łącza są z niego zarazem usuwane, wobec czego mogą być one odczytane tylko przez jeden proces i tylko jeden raz, podczas gdy z pliku można je odczytywać wielokrotnie.

W przypadku pliku funkcja **read** zwróci 0 (co oznacza dojście do końca pliku), wówczas, gdy zostaną odczytane wszystkie dane. Po odczytaniu wszystkich danych z łącza, czyli przy próbie odczytu z pustego łącza proces będzie blokowany w funkcji **read** (potencjalnie w potoku mogą pojawić się jakieś dane a 0 zostanie zwrócone przez funkcję **read** dopiero wówczas, gdy zamknięte zostaną wszystkie deskryptory do zapisu.

Odczytanie mniejszej liczby bajtów z pliku, niż rozmiar bufora przekazany jako trzeci parametr oznacza dojście od końca pliku. Jeżeli w łączu jest mniej danych, niż rozmiar bloku, który ma zostać odczytany, funkcja systemowa **read** zwróci wszystkie dane z łącza. Będzie to oczywiście liczba mniejsza, niż rozmiar bufora, przekazany jako trzeci parametr funkcji **read**, co nie oznacza jednak zakończenia komunikacji przez łącze.

`int write(inf fd, void *buf, size_t count)`

Wartości zwracane:

- poprawne wykonanie funkcji: *rzeczywista liczba bajtów, jaką udało się zapisać*
- zakończenie błędne: -1

Argumenty funkcji:

- *fd* - deskryptor potoku do którego mają zostać zapisane dane
- *buf* - adres bufora znajdującego się w segmencie danych procesu, z którego zostaną pobrane dane zapisane przez funkcję write
- *count* - ilość bajtów do zapisania

UWAGI:

Funkcja zapisuje w potoku *count* bajtów w całości (nie przeplatają się one z danymi pochodzącymi z innych zapisów) Różnica w działaniu funkcji **write** na łączu i na pliku zwykłym polega na tym, że jeżeli nie jest możliwe zapisanie bloku danych ze względu na brak miejsca w łączu, proces blokowany jest w funkcji **write** tak długo aż pojawi się odpowiednia ilość wolnego miejsca, tzn. aż inny proces odczyta i tym samym usunie dane z łącza

```
int close( int fd )
```

<unistd.h>

Wartości zwracane:

- poprawne wykonanie funkcji: 0
- zakończenie błędne: -1

Możliwe kody błędów (*errno*) w przypadku błędnego zakończenie funkcji:

*EBADF - wartość *fd* nie jest prawidłowym deskryptorem otwartego pliku

Argumenty funkcji:

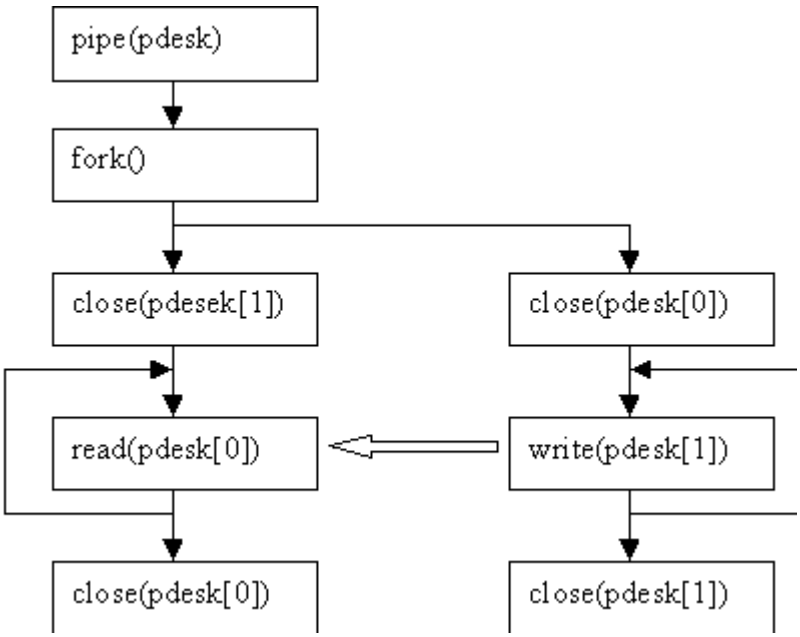
- *fd* - deskryptor zamykanego pliku

UWAGI:

Zamknięcie deskryptora pliku. Funkcja zamyka deskryptor pliku przekazany przez parametr *fd*. Po zamknięciu pliku zwalniana jest pozycja w tablicy deskryptorów i może ona zostać ponownie wykorzystana przy otwarciu kolejnego pliku, czyli nowo otwarty plik może otrzymać ten sam deskryptor, który miał plik wcześniej zamknięty. Ponadto zmniejszany jest o 1 licznik deskryptorów w tablicy otwartych plików. Jeśli *fd* jest ostatnią kopią deskryptora pliku, to zasoby z nim związane zostają zwolnione, natomiast jeśli deskryptor był ostatnia referencją do pliku, który usunięto komendą **unlink**, plik jest kasowany.
Funkcja ta nie opróżnia bufora pamięci podręcznej!

Sposób korzystania z łącza nienazwanego

Schemat komunikacji przez łącze nienazwane wygląda następująco:



Rysunek 2.Schemat komunikacji poprzez łącze nienazwane

Listing 1 pokazuje przykładowe użycie łącza do przekazania napisu (ciągu znaków) Hallo! z procesu potomnego do macierzystego.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
        int pdesk[2];
3
        if (pipe(pdesk) == -1){
```

```
                perror("Tworzenie potoku");
6       exit(1);
        }

9       switch(fork()){
          case -1: // blad w tworzeniu procesu
                perror("Tworzenie procesu");
12              exit(1);
          case 0: // proces potomny
                if (write(pdesk[1], "Hallo!", 7) == -1){
15                      perror("Zapis do potoku");
                        exit(1);
                }
18              exit(0);
          default: { // proces macierzysty
                char buf[10];
21              if (read(pdesk[0], buf, 10) == -1){
                        perror("Odczyt z potoku");
                        exit(1);
24              }
                printf("Odczytano z potoku: %s\n", buf);
          }
27      }
        return 0;
   }
```

Listing 1:Przykład użycia łącza nienazwanego w komunikacji przodek-potomek

Opis programu: Do utworzenia i zarazem otwarcia łącza nienazwanego służy funkcja systemowa **pipe**, wywołana przez proces macierzysty (linia 4). Następnie tworzony jest proces potomny przez wywołanie funkcji systemowej **fork** w linii 9, który dziedziczy tablicę otwartych plików swojego przodka. Warto zwrócić uwagę na sposób sprawdzania poprawności wykonania funkcji systemowych zwłaszcza w przypadku funkcji **fork**, która kończy się w dwóch procesach — macierzystym i potomnym. Proces potomny wykonuje program zawarty w liniach 14-19 i zapisuje do potoku ciąg 7 bajtów spod adresu początkowego napisu *Hallo!*. Zapis tego ciągu polega na wywołaniu funkcji systemowej **write** na odpowiednim deskryptorze, podobnie jak w przypadku pliku zwykłego. Proces macierzysty (linie 20-25) próbuje za pomocą funkcji **read** na odpowiednim deskryptorze odczytać ciąg 10 bajtów i umieścić go w buforze wskazywanym przez *buf* (linia 21). *buf* jest adresem początkowym tablicy znaków, zadeklarowanej w linii 20. Odczytany ciąg znaków może być krótszy, niż to wynika z rozmiaru bufora i wartości trzeciego parametru funkcji **read** (odczytane zostanie mniej niż 10 bajtów). Zawartość bufora, odczytana z potoku, wraz z odpowiednim napisem zostanie przekazana na standardowe wyjście.

Listing 2 zawiera zmodyfikowaną wersję przykładu przedstawionego na listingu 1. W poniższym przykładzie zakłada się, że wszystkie funkcje systemowe wykonują się poprawnie, w związku z czym w kodzie programu nie ma reakcji na błędy.

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
   int pdesk[2];
3
        pipe(pdesk);

6       if (fork() == 0){ // proces potomny
                write(pdesk[1], "Hallo!", 7);
                exit(0);
9       }
        else { // proces macierzysty
```

```
                char buf[10];
12              read(pdesk[0], buf, 10);
                read(pdesk[0], buf, 10);
                printf("Odczytano z potoku: %s\n", buf);
15      }
        return 0;
   }
```

Listing 2: Przykład odczytu z pustego łącza

Opis programu: Podobnie, jak w przykładzie na listingu 1, proces potomny przekazuje macierzystemu przez potok ciąg znaków *Hallo!*, ale proces macierzysty próbuje wykonać dwa razy odczyt zawartości tego potoku. Pierwszy odczyt (linia 12) będzie miał taki sam skutek jak w poprzednim przykładzie. Drugi odczyt (linia 13) spowoduje zawieszenie procesu, gdyż potok jest pusty, a proces macierzysty ma otwarty deskryptor do zapisu.

Listing 3 pokazuje sposób przejęcia wyniku wykonania standardowego programu systemu UNIX (w tym przypadku **ls**) w celu wykonania określonych działań (w tym przypadku konwersji małych liter na duże). Przejęcie argumentów z linii poleceń umożliwia przekazanie ich do programu wykonywanego przez proces potomny.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

 #define MAX 512

3 int main(int argc, char* argv[]) {
        int pdesk[2];

6       if (pipe(pdesk) == -1){
                perror("Tworzenie potoku");
                exit(1);
9       }

        switch(fork()){
12      case -1: // blad w tworzeniu procesu
                perror("Tworzenie procesu");
                exit(1);
15      case 0: // proces potomny
                dup2(pdesk[1], 1);
                execvp("ls", argv);
18              perror("Uruchomienie programuls");
                exit(1);
        default: { // proces macierzysty
21              char buf[MAX];
                int lb, i;

24              close(pdesk[1]);
                while ((lb=read(pdesk[0], buf, MAX)) > 0){
                  for(i=0; i<lb; i++)
27                      buf[i] = toupper(buf[i]);
                        if (write(1, buf,lb) == -1){
                                perror ("Zapis na standardowe wyjscie");
30                              exit(1);
                        }
                  }
33                if (lb == -1){
                        perror("Odczyt z potoku");
                        exit(1);
36              }
```

```
          }
        }
39      return 0;
  }
```

Listing 3: Konwersja wyniku polecenia **ls**

Opis programu: Program jest podobny do przykładu listingu 1, przy czym w procesie potomnym następuje przekierowanie standardowego wyjścia do potoku (linia 16), a następnie uruchamiany jest program **ls** (linia 17). W procesie macierzystym dane z potoku są sukcesywnie odczytywane (linia 25), małe litery w odczytanym bloku konwertowane są na duże (linie 26-27), a następnie blok jest zapisywany na standardowym wyjściu procesu macierzystego. Powyższa sekwencja powtarza się w pętli (linie 25-32) tak długo, aż funkcja systemowa **read** zwróci wartość 0 (lub -1 w przypadku błędu). Istotne jest zamknięcie deskryptora potoku do zapisu (linia 24) w celu uniknięcia zawieszenia procesu macierzystego w funkcji **read**.

Przykład na listingu 4 pokazuje realizację programową potoku **ls|tr a-z A-Z**, w którym proces potomny wykonuje polecenie **ls**, a proces macierzysty wykonuje polecenie **tr**. Funkcjonalnie jest to odpowiednik programu z listingu 3.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

 main(int argc, char* argv[]) {
        int pdesk[2];
3
        if (pipe(pdesk) == -1){
                perror("Tworzenie potoku");
6               exit(1);
        }

9       switch(fork()){
          case -1: // blad w tworzeniu procesu
                perror("Tworzenie procesu");
12              exit(1);
          case 0: // proces potomny
                dup2(pdesk[1], 1);
15              execvp("ls", argv);
                perror("Uruchomienie programuls");
                exit(1);
18        default: { // proces macierzysty
                close(pdesk[1]);
                dup2(pdesk[0], 0);
21              execlp("tr", "tr", "a-z", "A-Z", 0);
                perror("Uruchomienie programu tr");
                exit(1);
24       }
        }
        return 0;
   }
```

Listing 4: Programowa realizacja potoku **ls|tr a-z A-Z** na łączu nienazwanym

Opis programu: Program procesu potomnego (linie 16-19) jest taki sam, jak w przykładzie na listingu 3. W procesie macierzystym następuje z kolei przekierowanie standardowego wejścia na pobieranie danych z potoku (linia 22), po czym następuje uruchomienie programu **tr** (linia 23). W celu zagwarantowania, że przetwarzanie zakończy się w sposób naturalny konieczne jest zamknięcie wszystkich deskryptorów potoku do zapisu.

Deskryptory potomka zostaną zamknięte wraz z jego zakończeniem, a deskryptor procesu macierzystego zamykany jest w linii 21.

Funkcje systemowe służące do tworzenia i komunikacji poprzez łącza nazwane.

```
int mkfifo ( char * path, mode_t mode )
```

Wartości zwracane:

- poprawne wykonanie funkcji: 0
- zakończenie błędne: -1

Możliwe kody błędów (*errno*) w przypadku błędnego zakończenie funkcji:

- EEXIST - plik o podanej nazwie już istnieje, użyto flag O_CREAT i O_EXCL
- EFAULT - nazwa *pathname* wskazuje poza dostępną przestrzeń adresową
- EACCES - żądany dostęp do pliku nie jest dozwolony
- ENFILE - osiągnięto limit otwartych plików w systemie
- EMFILE - proces już otworzył dozwoloną maksymalną liczbę plików
- EROFS - żądane jest otwarcia w trybie zapisu pliku będącego plikiem tylko do odczytu

Argumenty funkcji:

- *path* - nazwa ścieżkowa pliku specjalnego będącego kolejką fifo
- *mode* - prawa dostępu do łącza

UWAGI:

Funkcja tworzy (ALE NIE OTWIERA) plik typu kolejka FIFO

```
int open (char *path, int flags)
```

Wartości zwracane:

- poprawne wykonanie funkcji: *deskryptor kolejki FIFO*
- zakończenie błędne: -1

Argumenty funkcji:

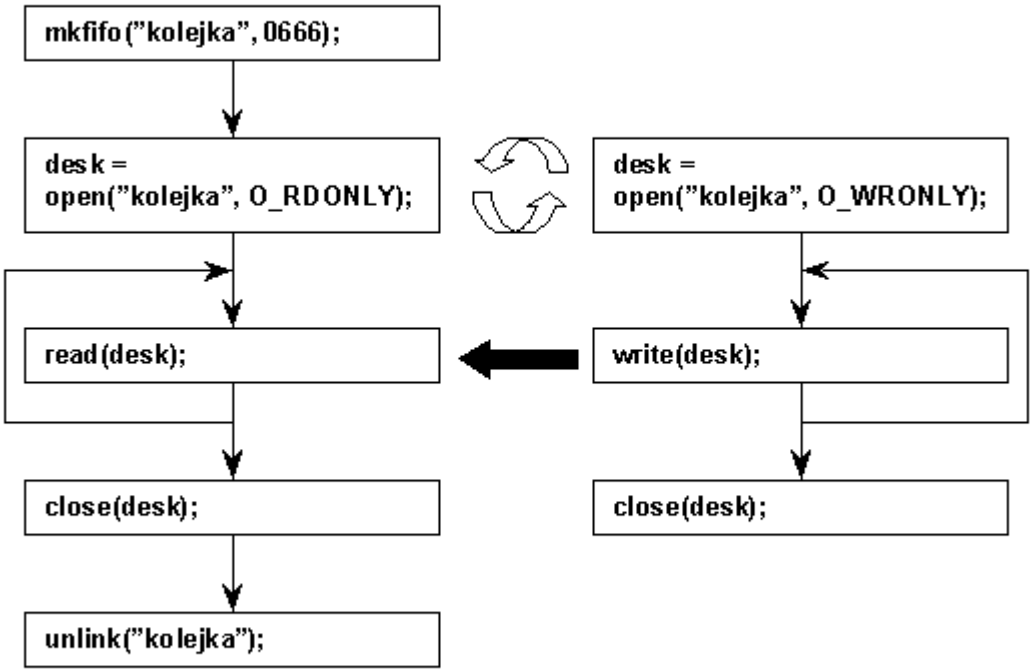
- *path* - nazwa ścieżkowa pliku specjalnego będącego kolejką fifo
- *mode* - prawa dostępu do łącza
- *flags* - określenie trybu w jakim jest otwierana kolejka:
    - O_RDONLY - tryb tylko do odczytu
    - O_WRONLY- tryb tylko do zapisu

UWAGI:

Utworzone łącze musi zostać następnie otwarte przez użycie funkcji **open**. Funkcja ta musi zostać wywołana przynajmniej przez dwa procesy w sposób komplementarny, tzn. jeden z nich musi otworzyć łącze do zapisu, a drugi do odczytu. Odczyt i zapis danych z łącza nazwanego odbywa się za pomocą funkcji: **READ**, **WRITE**, jak dla plików

Sposób korzystania z łącza nazwanego

Schemat komunikacji przez kolejkę FIFO:



Rysunek 2.Schemat komunikacji poprzez łącze nazwane

Program na listingu 5 pokazuje przykładowe tworzenie łącza i próbę jego otwarcia w trybie do odczytu.

```
#include <fcntl.h>

3  main(){
       mkfifo("kolFIFO", 0600);
       open("kolFIFO", O_RDONLY);
6  }
```

Listing 5: Przykład tworzenie i otwierania łącza nazwanego

Opis programu: Funkcja **mkfifo** (linia 4) tworzy plik specjalny typu łącze o nazwie kolFIFO z prawem zapisu i odczytu dla właściciela. W linii 5 następuje próba otwarcia łącza w trybie do odczytu. Proces zostanie zawieszony w funkcji **open** do czasu, aż inny proces będzie próbował otworzyć tę samą kolejkę w trybie do zapisu.

Listing 6 pokazuje realizację przykładu z listingu 1, w której wykorzystane zostało łącze nazwane.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>


3 int main() {
        int pdesk;

6       if (mkfifo("/tmp/fifo", 0600) == -1){
                perror("Tworzenie kolejki FIFO");
                exit(1);
9       }

        switch(fork()){
12        case -1: // blad w tworzeniu procesu
                perror("Tworzenie procesu");
                exit(1);
15        case 0:
                pdesk = open("/tmp/fifo", O_WRONLY);
                if (pdesk == -1){
18                      perror("Otwarcie potoku do zapisu");
                        exit(1);
                }
21              if (write(pdesk, "Hallo!", 7) == -1){
                         perror("Zapis do potoku");
                        exit(1);
24              }
                exit(0);
          default: {
27              char buf[10];

                pdesk = open("/tmp/fifo", O_RDONLY);
30              if (pdesk == -1){
                        perror("Otwarcie potoku do odczytu");
                        exit(1);
33              }
                if (read(pdesk, buf, 10) == -1){
                        perror("Odczyt z potoku");
36                      exit(1);
                }
                printf("Odczytano z potoku: %s\n", buf);
39        }
        }
        return 0;
   }
```

Listing 6: Przykład tworzenie i otwierania łącza nazwanego

Opis programu: łącze nazwane (kolejka FIFO) tworzona jest w wyniku wykonania funkcji **mkfifo** w linii 6. Następnie tworzony jest proces potomny (linia 11) i łącze otwierane jest przez oba procesy (potomny i macierzysty) w sposób komplementarny (odpowiednio linia 16 i linia 29). W dalszej części przetwarzanie przebiega tak, jak w przykładzie na listingu 1.

Listing 7 jest programową realizacją potoku **ls|tr a-z A-Z**, w której wykorzystane zostało łącze nazwane podobnie, jak łącze nienazwane w przykładzie na listingu 4.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#include <fcntl.h>
3
   main(int argc, char* argv[]) {
        int pdesk;
6
        if (mkfifo("/tmp/fifo", 0600) == -1){
                perror("Tworzenie kolejki FIFO");
9               exit(1);
        }

12      switch(fork()){
          case -1: // blad w tworzeniu procesu
                perror("Tworzenie procesu");
15              exit(1);
          case 0: // proces potomny
                close(1);
18              pdesk = open("/tmp/fifo", O_WRONLY);
                if (pdesk == -1){
                        perror("Otwarcie potoku do zapisu");
21                      exit(1);
                }
                else if (pdesk != 1){
24                      fprintf(stderr, "Niewlasciwy deskryptor do
zapisu\n");
                        exit(1);
                }
27              execvp("ls", argv);
                perror("Uruchomienie programuls");
                exit(1);
30        default: { // proces macierzysty
                close(0);
                pdesk = open("/tmp/fifo", O_RDONLY);
33              if (pdesk == -1){
                        perror("Otwarcie potoku do odczytu");
                        exit(1);
36              }
                else if (pdesk != 0){
                        fprintf(stderr, "Niewlasciwy deskryptor do
odczytu\n");
39                      exit(1);
                }
                execlp("tr", "tr", "a-z", "A-Z", 0);
42              perror("Uruchomienie programu tr");
                exit(1);
          }
45      }
        return 0;
   }
```

Listing 7: Programowa realizacja potoku **ls|tr a-z A-Z** na łączu nazwanym

Opis programu: W linii 7 tworzona jest kolejka FIFO o nazwie *fifo* w katalogu */tmp* z prawem do zapisu i odczytu dla właściciela. Kolejka ta otwierana jest przez proces potomny i macierzysty w trybie odpowiednio do zapisu i do odczytu (linia 18 linia 33). Następnie sprawdzana jest poprawność wykonania operacji otwarcia (linie 19 i 34) oraz poprawność przydzielonych deskryptorów (linie 23 i 38). Sprawdzanie poprawności deskryptorów polega na upewnieniu się, że deskryptor łącza do zapisu ma wartość 1 (łącze jest standardowym wyjściem procesu potomnego), a deskryptor łącza do odczytu ma wartość 0 (łącze jest standardowym wejściem procesu macierzystego). Później następuje uruchomienie odpowiednio programów **ls** i **tr** podobnie, jak w przykładzie na listingu 4.

Przykłady błędów w synchronizacji procesów korzystających z łączy

Operacje zapisu i odczytu na łączach realizowane są w taki sposób, że procesy podlegają synchronizacji zgodnie ze modelem producent-konsument. Nieodpowiednie użycie dodatkowych mechanizmów synchronizacji może spowodować konflikt z synchronizacowsu ją na łączu i w konsekwencji prowadzić do stanów niepożądanych typu *zakleszczenie* (ang. *deadlock*).

Listing 8 przedstawia przykład programu, w którym może nastąpić zakleszczenie, gdy pojemność łącza okaże się zbyt mała dla pomieszczenia całości danych przekazywanych przez polecenie **ls**.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MAX 512

3  main(int argc, char* argv[]) {
        int pdesk[2];

6       if (pipe(pdesk) == -1){
                perror("Tworzenie potoku");
                exit(1);
9       }

        if (fork() == 0){ // proces potomny
12              dup2(pdesk[1], 1);
                execvp("ls", argv);
                perror("Uruchomienie programułs");
15              exit(1);
        }
        else { // proces macierzysty
18              char buf[MAX];
                int lb, i;

21              close(pdesk[1]);
                wait(0);
                while ((lb=read(pdesk[0], buf, MAX)) > 0){
24                 for(i=0; i<lb; i++)
                        buf[i] = toupper(buf[i]);
                   write(1, buf,lb);
27              }
        }
        return 0;
   }
```

Listing 8: Przykład programu dopuszczającego zakleszczenie w operacji na łączu nienazwanym

Opis programu: Podobnie jak w przykładzie na listingu 3 proces potomny przekazuje dane (wynik wykonania programu *ls*) do potoku (linie 12-15), a proces macierzysty przejmuje i przetwarza te dane w pętli w liniach 23-27. Przed przejściem do wykonania pętli proces macierzysty oczekuje na zakończenie potomka (linia 22). Jeśli dane generowane przez program **ls** w procesie potomnym nie zmieszczą się w potoku, proces ten zostanie zablokowany gdzieś w funkcji **write** w programie **ls**. Proces potomny nie będzie więc zakończony i tym samym proces macierzysty nie wyjdzie z funkcji **wait**. Odblokowanie potomka może nastąpić w wyniku zwolnienia miejsca w potoku przez odczyt znajdujących się w nim danych. Dane te powinny zostać odczytane przez proces macierzysty w wyniku wykonania funkcji **read** (linia 23), ale proces macierzysty nie przejdzie do linii 23 przed zakończeniem potomka. Proces macierzysty blokuje zatem potomka, nie zwalniając miejsca w potoku, a proces potomny blokuje przodka w funkcji **wait**, nie kończąc się. Wystąpi zatem zakleszczenie. Zakleszczenie nie

wystąpi w opisywanym programie, jeśli wszystkie dane, generowane przez program **ls**, zmieszczą się w całości w potoku. Wówczas proces potomny będzie mógł się zakończyć po umieszczeniu danych w potoku, w następstwie czego proces macierzysty będzie mógł wyjść z funkcji **wait** i przystąpić do przetwarzania danych z potoku.

Przykład na listingu 9 pokazuje zakleszczenie w wyniku nieprawidłowości w synchronizacji przy otwieraniu łącza nazwanego.

```
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

   #define MAX 512
3
int main(int argc, char* argv[]) {
        int pdesk;
6
        if (mkfifo("/tmp/fifo", 0600) == -1){
                perror("Tworzenie kolejki FIFO");
9                exit(1);
        }

12      if (fork() == 0){ // proces potomny
                close(1);
                open("/tmp/fifo", O_WRONLY);
15              execvp("ls", argv);
                perror("Uruchomienie programuls");
                exit(1);
18      }
        else { // proces macierzysty
                char buf[MAX];
21              int lb, i;

                wait(0);
24              pdesk = open("/tmp/fifo", O_RDONLY);
                while ((lb=read(pdesk, buf, MAX)) > 0){
                   for(i=0; i<lb; i++)
27                      buf[i] = toupper(buf[i]);
                   write(1, buf,lb);
                }
30      }
        return 0;
   }
```

Listing 9: Przykład programu dopuszczającego zakleszczenie przy otwieraniu łącza nazwanego

Opis programu: Proces potomny w linii 13 próbuje otworzyć kolejkę FIFO do zapisu. Zostanie on zatem zablokowany do momentu, aż inny proces wywoła funkcję **open** w celu otwarcia kolejki do odczytu. Jeśli jedynym takim procesem jest proces macierzysty (linia 23), to przejdzie on do funkcji **open** dopiero po zakończeniu procesu potomnego, gdyż wcześniej zostanie zablokowany w funkcji **wait**. Proces potomny nie zakończy się, gdyż będzie zablokowany w funkcji **open**, więc będzie blokował proces macierzysty w funkcji **wait**. Proces macierzysty nie umożliwi natomiast potomkowi wyjścia z **open**, gdyż nie może przejść do linii 23. Nastąpi zatem zakleszczenie.

popen, pclose - we/wy procesu

SKŁADNIA

#include <stdio.h>

FILE *popen(const char *_command_, const char *_type_);

int pclose(FILE *_stream_);

OPIS

Funkcja **popen()** otwiera proces, tworząc łącze, rozwidlając się przez fork() i wywołując powłokę. Ponieważ łącze jest z definicji jednokierunkowe, argument *type* może określać tylko odczyt albo tylko zapis, nie oba naraz. Otrzymany w wyniku tego strumień będzie tylko do odczytu albo tylko do zapisu.

Argument *command* jest wskaźnikiem do zakończonego znakiem NUL łańcucha, zawierającego wiersz poleceń powłoki. Polecenie to jest przekazywane do */bin/sh* przy użyciu opcji **-c**; wszelka interpretacja jest dokonywana przez powłokę. Argument *type* jest zakończonym znakiem NUL łańcuchem, który musi być albo `r', albo `w' (odpowiednio dla odczytu i zapisu).

Wartość zwracana przez **popen()** to normalny strumień we/wy, lecz powinien on być zamykany przy użyciu **pclose()** zamiast **fclose()**. Zapisywanie do takiego strumienia powoduje pisanie na standardowe wejście polecenia. Standardowe wyjście polecenia jest takie samo, jak procesu, który wywołał **popen()**, chyba że zostało to zmienione przez polecenie. Podobnie, odczyt z tak otwartego strumienia powoduje odczyt ze standardowego wyjścia polecenia, a standardowe wejście polecenia jest wtedy tożsame z wejściem procesu, który wywołał **popen()**.

Należy zauważyć, że strumienie wyjściowe powstałe z **popen** są domyślnie w pełni buforowane.

Funkcja **pclose** oczekuje na zakończenie stowarzyszonego procesu i zwraca jego kod zakończenia, podobnie jak to czyni **wait4**.

WARTOŚĆ ZWRACANA

Funkcja **popen** zwraca **NULL** jeśli nie powiodły się wywołania fork(2) lub pipe(2), lub jeśli nie udało się przydzielić pamięci.

Funkcja **pclose** zwraca -1 jeśli **wait4** zwróci błąd lub zostały wykryte jakieś inne błędy.

BŁĘDY

Funkcja **popen** nie ustawia *errno*, jeżeli nie uda się

tworzenie duplikatów deskryptorów plików

```
int dup( int oldfd )
```

Wartości zwracane:

- poprawne wykonanie funkcji: *nowy deskryptor*
- zakończenie błędne: -1

Możliwe kody błędów (*errno*) w przypadku błędnego zakończenie funkcji:

- EBADF - *oldfd* nie jest deskryptorem otwartego pliku
- EMFILE - proces już osiągnął maksymalną liczbę otwartych deskryptorów plików

Argumenty funkcji:

- *oldfd* - deskryptor zamykanego pliku

UWAGI:

Funkcja tworzy kopię pozycji w tablicy deskryptorów na innej, wolnej pozycji o najniższym indeksie. W ten sposób otrzymujemy nowy deskryptor związany z tym samym otwartym plikiem. Nowa pozycja w tablicy deskryptorów wskazuje na tą samą pozycję w tablicy otwartych plików, stąd stary i nowy deskryptor mogą być używane zamiennie. Deskryptory dzielą pozycję pliku i flagi, np. jeśli pozycja pliku zmieniła się po użyciu funkcji **lseek** na jednym z deskryptorów, zmieniła się ona także na drugim.

tworzenie duplikatów deskryptorów plików

```
int dup2( int oldfd, int newfd )
```

Wartości zwracane:

- poprawne wykonanie funkcji: *nowy deskryptor*
- zakończenie błędne: -1

Możliwe kody błędów (*errno*) w przypadku błędnego zakończenie funkcji

- EBADF - *oldfd* nie jest deskryptorem otwartego pliku lub *newfd* jest poza dozwolonym zasięgiem deskryptorów plików
- EMFILE - proces już osiągnął maksymalną liczbę otwartych deskryptorów plików

Argumenty funkcji:

- *oldfd* - deskryptor zamykanego pliku
- *newfd* - nowy deskryptor

UWAGI:

Utworzenie kopii deskryptora. Podobnie jak w przypadku funkcji **dup** tworzony jest nowy deskryptor otwartego pliku identyfikowanego przez *oldfd*. *Newfd* staje się nowym, dodatkowym deskryptorem, a jeśli przed wywołaniem **dup2** identyfikował on inny plik, następuje zamknięcie tego deskryptora przed powieleniem *oldfd*. Funkcja zwraca wartość nowego deskryptora.
Wykonanie operacji **close(1)**; **dup(fd)**; jest równoważne operacji **dup(1,fd)**

<unistd.h> <fcntl.h>

int fcntl(int fd, int cmd, long arg);

Steruje otwartym plikiem.

- `fd` - Deskryptor gniazda (pliku
- `cmd` - Komenda. W operacjach na gniazdach przydatne sa:
  - `F_SETFL` - Ustawia flagi deskryptora. Dla gniazda maja zastosowanie tylko dwie flagi: `O_ASYNC` - Informuje system aby wysyłał do procesu sygnał SIGIO, kiedy możliwe jest wykonywanie operacji we-wy na gnieździe. Dodatkowo, jeśli pojawią się dane typu OOB to wysłany zostanie sygnał SIGURG.
    `O_NONBLOCK` - Przełącza gniazdo w tryb nieblokujący. Tzn. wszystkie funkcje, które normalnie blokowałyby (**send()**, **recv()**, **accept()** itd.) będą zwracały błąd EAGAIN jeśli dana operacja nie może być zakończona natychmiast. Wyjątkiem jest **connect()**, które w takiej sytuacji zwraca `EINPROGRESS`. Obie flagi mają zastosowanie jeśli zamierzamy korzystać z komunikacji asynchronicznej.
  - `F_GETFL` - Odczytuje flagi deskryptora.
  - `F_SETOWN` - Wskazuje proces (PID) albo grupę procesów, które mają otrzymywać wspomniane wcześniej sygnały SIGIO/SIGURG. Grupy procesów, dla odróznienia od pojedynczych procesów, podaje się w postaci liczb ujemnych.
  - `F_GETOWN` - Odczytuje proces/grupę.
  - `F_SETSIG` - Wprowadzona w jądrach 2.2.x. Wskazuje sygnał, który będzie używany zamiast SIGIO do powiadamiania o stanie gniazd. Odpowiednie użycie tej komendy bardzo ułatwia obsługę komunikacji asynchronicznej. Przekonamy się o tym przy okazji analizy tego typu komunikacji.
  - `F_GETSIG` - Pobiera numer sygnału.
- `arg` - Argument komendy.

Przekazuje wynik zależny od argumentu op lub -1 w razie błędu (nadaje wartość zmiennej errno);

<sys/stat.h>

int mknod(const char *sciezka, mode_t tryb, dev_t nrurz)

```
char *sciezka;     /* nazwa pliku
mode_t *tryb;              /* tryb pliku
dev_t  nrurz;              /* numer urzadzenia
```

/* W przypadku powodzenia zwraca 0, bledu -1 */

Funkcja systemowa **mknod** umożliwia tworzenie plików dowolnego rodzaju. Bierze się pod uwagę wszystkich 16 bitów trybu tego pliku, określonego parametrem tryb. Tworząc plik specjalny należy określić numer urządzenia będący indeksem tablicy jadra systemu, którą zawiera informacje o podprogramach obsługi urządzeń. Numer urządzenia składa sie z drugorzędnego i głównego numeru urządzenia.
Funkcji **mknod** używa sie do stworzenia pliku specjalnego tylko wówczas, gdy w systemie jest instalowany nowy podprogram obsługi urządzeń lub gdy nowy drugorzędny numer urządzenia staje sie aktywny. Wraz z ustaleniem konfiguracji systemu funkcja **mknod** przestaje mieć znaczenie w odniesieniu do plików specjalnych. Z wyjątkiem tworzenia kolejek FIFO, wyłącznie nadzorca może wykonywać funkcje **mknod**.
W odróżnieniu do Unix'a, gdzie funkcja mknod jest wykorzystywana do tworzenia katalogów. W systemie Linux jest zdefiniowana oddzielna funkcja do tworzenia katalogów.

Wykorzystanie funkcji select

Funkcja `select` umożliwia nadzorowanie zbioru deskryptorów pod względem możliwości odczytu, zapisu bądź wystąpienia sytuacji wyjątkowych. Formalnie prototyp funkcji wygląda następująco (definicja w `sys/select.h`):

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
```

readfds – bity operacji do czytania lub NULL (sprawdzamy czy któryś deksryptor jest gotowy do czytania)
writefds – bity operacji do pisania lub NULL (sprawdzamy czy któryś deksryptor jest gotowy do zapisu)
exceptfds – bity sygnalizowania błędu lub NULL
Funkcja przyjmuje wspomniane trzy zbiory deskryptorów, jednak nie ma obowiązku określania ich wszystkich (można w miejsce odp. zbioru deskryptorów podać `NULL` - wówczas dany zbiór nie będzie nadzorowany przez `select`).
W celu umożliwienia nasłuchu na dwóch (lub więcej) gniazdach jednocześnie, należy postępować wg następującego schematu:

1. wstaw deskryptory gniazd (`g1` i `g2`) do zbioru `readfds`
2. ustaw `timeout`
3. wywołaj `select` na zbiorze `readfds`
4. jeśli `select` zwrócił wartość dodatnią, to
    1. sprawdź czy `g1` jest ustawiony w `readfds`, jeśli tak, to obsłuż odczyt na gnieździe `g1`
    2. sprawdź czy `g2` jest ustawiony w `readfds`, jeśli tak, to obsłuż odczyt na gnieździe `g2`
5. ew. powrót do 1

W przypadku gniazd TCP `select` zwróci gotowość deskryptora jeśli możliwe jest wywołanie na gnieździe funkcji `accept` (gniazdo nasłuchujące) lub `recv` (gniazdo komunikacyjne) bez blokowania aplikacji (czyli w momencie, w którym istnieje oczekujące połączenie na gnieździe nasłuchującym lub gdy czekają dane w buforze na gnieździe komunikacyjnym).
W przypadku gniazd UDP `select` zwróci gotowość deskryptora jeśli możliwe jest nieblokujące wywołanie `recvfrom` (w buforze odczytu oczekuje datagram).

Istotne informacje na temat select (dostępne w man pages):

- do czyszczenia zbioru deskryptorów służy `FD_ZERO(fd_set *fds)`
- do dodawania deskryptora do zbioru służy `FD_SET(int fd, fd_set *fds)`
- do usuwania deskryptora ze zbioru służy `FD_CLR(int fd, fd_set *fds)`
- do sprawdzania przynależności deskryptora do zbioru służy `FD_ISSET(int fd, fd_set *fds)`
- funkcja `select` jako swój wynik zwraca liczbę "gotowych" deskryptorów
- pierwszym parametrem `select` musi być **największa wartość deskryptora** ze zbiorów **powiększona o 1**, a **NIE** liczba deskryptorów (częsty błąd!)
- jeśli jako `timeout` podamy `NULL`, `select` wraca natychmiast, informując jaki jest bieżący stan deskryptorów
- jeśli jako `timeout` podamy niezerowy czas, `select` wraca po upływie tego czasu lub po wystąpieniu zdarzenia na deskryptorze (zależy co nastąpi wcześniej)

- `select` może (zależnie od implementacji) zmienić wartość `timeout`, zatem należy zawsze ustawiać czas oczekiwania na nowo przed wywołaniem funkcji
- jeśli jako `timeout` podamy zerowy czas (ale nie `NULL`), `select` wróci dopiero po wystąpieniu zdarzenia (innymi słowy zerowy czas oczekiwania oznacza czekanie w nieskończoność na wystąpienie zdarzenia)
- struktura `timeval` posiada pola `tv_sec` (sekundy) i `tv_usec` (mikrosekundy)

NAZWA

fpathconf, pathconf - pobranie konfiguracji dla plików

SKŁADNIA

#include <unistd.h>

long fpathconf(int *filedes***, int** *name***);**
long pathconf(char ****path*, int** *name***);**

OPIS

Funkcja **fpathconf()** pobiera wartość opcji konfiguracyjnej *name* dla otwartego deskryptora pliku *filedes*.

Funkcja **pathconf()** pobiera wartość opcji konfiguracyjnej *name* dla pliku o nazwie *path*.

Odpowiednie makra, zdefiniowane w **<unistd.h>**, są wartościami minimalnymi. Jeśli aplikacja chce korzystać z wartości, które mogą się zmieniać, to może wywołać **fpath conf()** lub **pathconf()**, które zwracają bardziej liberalne wyniki.

Ustawianie jednej z poniższych stałych jako wartości *name*, zwraca następujące opcje konfiguracji:

_PC_LINK_MAX
zwraca maksymalną liczbę dowiązań do pliku. Jeśli *filedes* lub *path* odnoszą się do katalogu, to wartość dotyczy całego katalogu. Odpowiadające temu makro to **_POSIX_LINK_MAX**.

_PC_MAX_CANON
zwraca maksymalną długość sformatowanej linii wejściowej, przy czym *filedes* lub *path* musi odnosić się do terminala. Odpowiadające temu makro to **_POSIX_MAX_CANON**.

_PC_MAX_INPUT
zwraca maksymalną długość linii wejściowej, przy czym *filedes* lub *path* musi odnosić się do terminala. Odpowiadające temu makro to **_POSIX_MAX_INPUT**.

_PC_NAME_MAX
zwraca maksymalną długość nazwy pliku w katalogu *path* lub *filedes*, jaką proces może utworzyć. Odpowiadające temu makro to **_POSIX_NAME_MAX**.

_PC_PATH_MAX
zwraca maksymalną długość względnej ścieżki, gdy *path* lub *filedes* jest katalogiem bieżącym. Odpowiadające temu makro to **_POSIX_PATH_MAX**.

_PC_PIPE_BUF
zwraca rozmiar bufora łącza komunikacyjnego (pipe), przy czym *filedes* musi odnosić się do FIFO lub łącza, a *path* musi odnosić się do FIFO. Odpowiadające temu makro to **_POSIX_PIPE_BUF**.

_PC_CHOWN_RESTRICTED

zwraca wartość niezerową jeśli wywołanie chown(2) nie może być zastosowane do tego pliku. Jeśli *filedes* lub *path* odnoszą się do katalogu, to dotyczy to wszystkich plików w tym katalogu. Odpowiadające temu makro to **_POSIX_CHOWN_RESTRICTED**.

_PC_NO_TRUNC
zwraca wartość niezerową jeśli dostęp do plików o nazwach dłuższych od **_POSIX_NAME_MAX** powoduje błąd. Odpowiadające temu makro to **_POSIX_NO_TRUNC**.

_PC_VDISABLE
zwraca wartość niezerową jeśli przetwarzanie znaków specjalnych może być wyłączone, przy czym *filedes* lub *path* muszą odnosić się do terminala.

WARTOŚĆ ZWRACANA
Jeśli istnieje ograniczenie, to jest ono zwracane. Jeśli system dla danego zasobu nie ma ograniczenia, zwracane jest -1 a *errno* pozostaje niezmienione. Jeśli wystąpi błąd, zwracane jest -1 a ustawienie *errno* określa charakter błędu.

ZGODNE Z
POSIX.1

UWAGI
W danym katalogu mogą istnieć pliki o nazwach dłuższych niż wartość zwrócona dla *name* równego **_PC_NAME_MAX**.
Niektóre ze zwracanych wartości mogą być olbrzymie, nie nadają się one do alokowania pamięci.

Napisać dwa programy: 'producent' i 'konsument'. Program 'producent' czyta informacje tekstowe ze standardowego wejścia i zapisuje je do łścza nazwanego (FIFO). Maksymalny czas pracy programu jest parametrem wywołania. Program 'konsument' czyta dane z tego samego łścza nazwanego i wyświetla je na standardowym wyjściu, ale nie dłużej niż przez czas określony przez parametr wywołania.

PRODUCENT:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>

#define KOMUNIKAT(x) printf("Blad wywolania funkcji " #x "()!.\n");
#define MAX 64

int main(int argc, int *argv[])
{
        if( argc != 2 )
                fprintf(stderr , "Bledne wywolanie programu!\n");

        int time_exit;
        if( (time_exit = atoi(argv[1]) ) == 0 ){
                KOMUNIKAT(atoi);
                exit(1);
                }

        system("rm FIFO");

        if( mkfifo("FIFO", 0666) == -1 ){
                KOMUNIKAT(mkfifo);
                exit(2);
                }
```

```
	int fd;

	if ( (fd = open("FIFO", O_WRONLY) ) < 0){
		KOMUNIKAT(open);
		exit(3);
		}

	alarm(time_exit);

	char buf[MAX];

	while(1){
		printf("Producent > \n");
		if( fgets(buf, MAX, stdin) == NULL ){
			KOMUNIKAT(fgets);
			exit(4);
		}
		if( (write(fd, buf, sizeof(buf) ) ) < 0 ){
			KOMUNIKAT(write);
			exit(5);
			}
	}
	return 0;
}
```

KONSUMENT:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>

#define KOMUNIKAT(x) printf("Blad wywolania funkcji " #x "()!.\n");
#define MAX 64

int main(void)
{

	int fd;

	if ( (fd = open("FIFO", O_RDONLY) ) < 0){
		KOMUNIKAT(open);
		exit(3);
		}

	char buf[MAX];

	while(1){
		if( (read(fd, buf, MAX) ) <= 0 ){
			KOMUNIKAT(write);
			exit(5);
			}
		printf("Konsument > %s\n", buf);

	}
	return 0;
}
```

Napisać program **potok** wywoływany następująco:
potok cmd1 cmd2 wynik

Program ten tworzy dwa procesy potomne za pomocą poleceń **cmd1** oraz **cmd**, przekierowując standardowe wyjście pierwszego procesu na standardowe wejście drugiego, oraz standardowe wyjście drugiego procesu do pliku (tu: **wynik**). Przykładowo, wywołanie:
potok ls less lista
powinno działać tak, jak polecenie powłoki (np. zsh):
ls | less > lista

```
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#define MAX 512

int main(int argc, char* argv[])
{
	pid_t pid1, pid2;
	int desk;
	int pfd[2];
	char buf[MAX];

	if (pipe(pfd) == -1)
	{
	perror("Blad tworzenie lacza nienazwanego.\n");
		exit(1);
	}

	if( (pid1 = fork()) == 0 )
	{
		close( pfd[0] );
		dup2( pfd[1], 1);
		execlp(argv[1], NULL);
		perror("Blad wywolania execcl w potomku 1\n");
		exit (2);
	}

	desk = open(argv[3], O_CREAT | O_WRONLY, 0666);

	if( (pid2 = fork()) == 0 )
	{
		dup2(pfd[0], 0);
		dup2(desk, 1);
		execlp(argv[2], NULL);
		perror("Blad wywolania execcl w potomku 2\n");
		exit (2);
	}

	close(desk);
	close(pfd[0]);
	close(pfd[1]);
	wait();
	wait();

	return 0;
}
```

UWAGA: Program Był testowany na Ubuntu 7.10 i openSUSE 10.3. Na pierwszym systemie działało na drugim nie. A mi się już nie chce wnikać o co biega :)