8 Łącza nienazwane(potoki)

- **Łącze (potok, ang. *pipe*)** jest to urządzenie komunikacyjne pozwalające na przesyłanie informacji w jedną stronę. Jeden proces wysyła dane do łącza za pomocą funkcji `write`, zaś inny odczytuje dane z łącza za pomocą funkcji `read`. Procesy zapisujące dopisują dane na końcu łącza, zaś procesy odczytujące odczytują je z początku łącza. ***Odczytane dane są usuwane z łącza*.**

- Przykład:

```
$ ls | more
```

- *Łącza nienazwane mogą być używane tylko pomiędzy procesami ze sobą powiązanymi.*

8.1 Tworzenie łącza – funkcja `pipe`

```
#include <unistd.h>

int pipe(int filedes[2]);
```

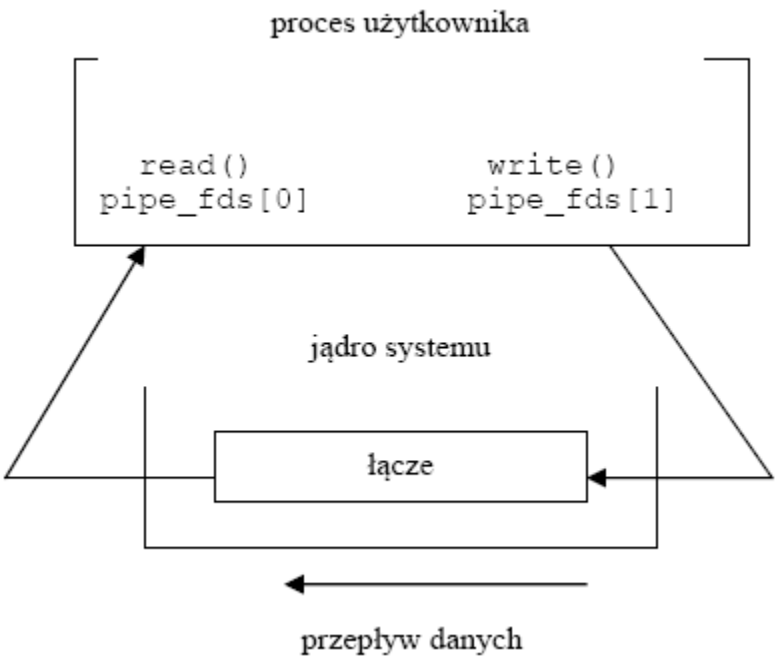
Funkcja `pipe` umieszcza w tablicy parę deskryptorów reprezentujących potok (jego dwa końce). Pierwszy z nich jest otwarty do czytania z potoku, zaś drugi do zapisu.
Funkcja zwraca 0, jeśli potok został utworzony, -1 w przeciwnym wypadku.

- Przykład:

```
int pipe_fds[2];
int r_fd;
int w_fd;
char buf[512];

if (pipe(pipe_fds) !=0 ) {
   perror("pipe()");
   ...
}

r_fd= pipe_fds [0];
w_fd= pipe_fds [1];
...
write(w_fd,buf,sizeof(buf));
...
read(r_fd,buf,sizeof(buf));
```


proces użytkownika
read()
pipe_fds[0]
write()
pipe_fds[1]
jądro systemu
łącze
przepływ danych

8.2 Właściwości łącza

- **Potok buforuje dane**: dane przesyłane są do potoku i trzymane w nim, aż zostaną przeczytane. (Łącze automatycznie synchronizuje dwa procesy).

- **Pojemność łącza jest ograniczona.** Stała `PIPE_BUF` określa pojemność łącza w jądrze systemu (Posix wymaga, aby było to co najmniej 512 bajtów, na przykład w Linuksie często jest to 4096 bajtów).

- **Zapisywanie do potoku**

    - Proces próbujący pisać do pełnego łącza zostanie zablokowany, dopóki inny proces nie odczyta danych.
    - Zapis do potoku wykonuje się *niepodzielnie*, o ile ilość bajtów nie przekracza `PIPE_BUF`, w przeciwnym przypadku system nie gwarantuje niepodzielności zapisu.
    - Jeśli piszemy do łącza, które zostało zamknięte do odczytu, generowany jest sygnał `SIGPIPE`. Domyślną obsługą tego sygnału jest zakończenie procesu. Jeśli proces obsługuje sygnał `SIGPIPE` i powraca z procedury obsługi tego błędu (lub ignoruje `SIGPIPE`), kolejne odwołanie się do funkcji `write` zwróci błąd `EPIPE` (*Broken pipe*).
- **Czytanie z potoku**
    - Proces próbujący czytać automatycznie zostaje zablokowany, dopóki w łączu nie pojawią się dane.

    - Jeśli przy odczycie żąda się więcej niż `PIPE_BUF`, jądro po prostu odczytuje dostępne dane i funkcja `read` zwraca liczbę przeczytanych bajtów.

    - Kiedy wszystkie procesy piszące zostaną zakończone, `read` zwracać będzie 0 (czyli koniec pliku).

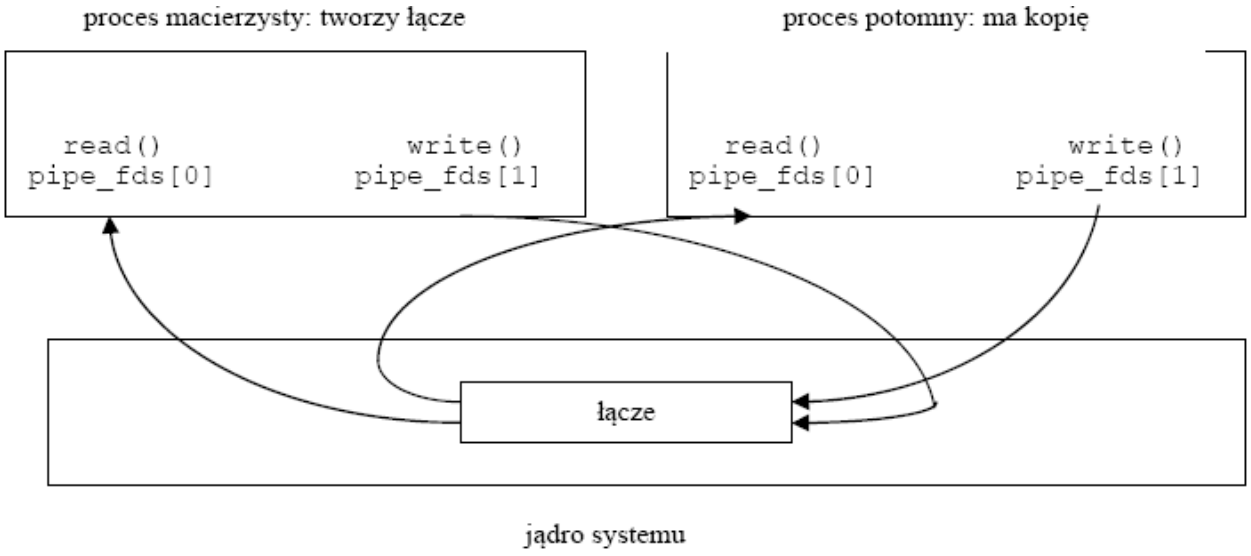
    - Potok działa podobnie do kolejki; jeśli jest wielu odbiorców, każdy z nich pobierze kolejną porcję danych - nie można skierować danych do konkretnego odbiorcy.

- Dane w łączu traktuje się jako strumień danych bez zaznaczenia granic komunikatu. Ewentualny podział tego strumienia bajtów należy do aplikacji.

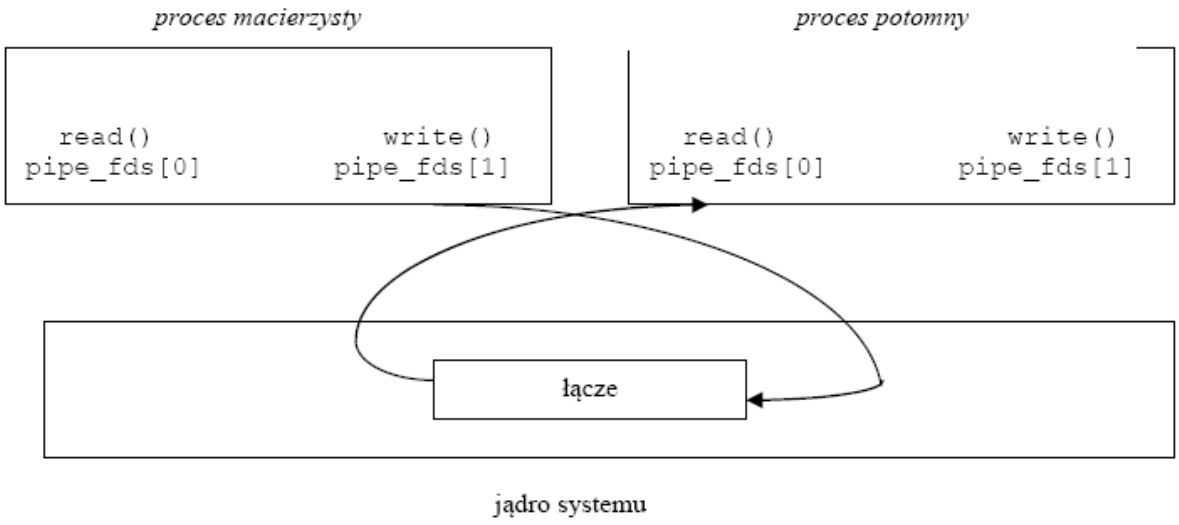
- Łącze nie może stosowane do rozgłaszania danych dla wielu odbiorców - odczytanie danych powoduje usunięcie ich z łącza). Podobnie, jeśli jest wiele procesów zapisujących nie można stwierdzić, który z nich przesłał dane. Ewentualne identyfikowanie należy do aplikacji.

8.3 Wykorzystanie łącza do komunikacji między procesem macierzystym i potomnym

- W procesie potomnym powielane są otwarte deskryptory z procesu macierzystego.



- Przykład: Chcemy utworzyć potok pozwalający przesyłać komunikaty od procesu macierzystego do procesu potomnego. Wystarczy zamknąć odpowiednie deskryptory odpowiednio w procesie macierzystym i potomnym.



- Przykład: Proces macierzysty pisze do potoku, proces potomny czyta i wyświetla na ekranie. Oba procesy korzystają ze strumieni we-wy.

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

void zapisywanie(const char* komunikat, int licznik, FILE* strumien){
for (; licznik > 0; --licznik) {
      fprintf (strumien, "%s\n", komunikat);
      fflush (strumien);
      sleep (1);
}
}

void odczytywanie (FILE* strumien) {
      char bufor[1024];
      while (!feof (strumien) && !ferror (strumien)
            && fgets (bufor, sizeof (bufor), strumien) != NULL)
        fputs (bufor, stdout);
}
int main () {
int fd[2];
pid_t pid;

pipe(fd);
pid = fork ();
/* proces potomny */
if (pid == (pid_t) 0) { /* brak obsługi błędów */
      FILE* strumien;
      close (fd[1]);
      strumien = fdopen(fd[0], "r"); /* konwersja deskryptora
                                        do obiektu FILE */
      odczytywanie(strumien);
      close(fd[0]);
}
/* proces macierzysty */
else {
      FILE* strumien;
      close(fd[0]);
      strumien = fdopen(fd[1], "w"); /* konwersja deskryptora
                                        do obiektu FILE */
      zapisywanie("Witam!", 6, strumien);
      close(fd[1]);
}
return 0;
}
```

- Przykład: Program generuje napisy. Chcemy je wyświetlić w kolejności alfabetycznej. Możemy to zrealizować przekazując wyjście standardowe programu na wejście polecenia systemowego sort(w5p2.c).

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

int main ()
{
int fd[2];
pid_t pid;

pipe(fd);
pid = fork (); /* brak obsługi błędów */

if (pid == (pid_t) 0) {   /* proces potomny */
     close(fd[1]);
     dup2(fd[0], STDIN_FILENO);  /* posix stdin */
     close(fd[0]);   /* dlaczego zamykamy ten deskryptor? */
     execlp("sort", "sort", 0);
}
else {          /* proces macierzysty */
     FILE* strumien;
     close (fd[0]);
     strumien = fdopen(fd[1], "w");
     fprintf(strumien, "Witam.\n");
     fprintf(strumien, "Welcome.\n");
     fprintf(strumien, "Bienvenue.\n");
     fprintf(strumien, "Willkommen.\n");
     fflush(strumien);
     close(fd[1]);
     waitpid(pid, NULL, 0);
}
return 0;
}
```

- Przykład: W programie uruchamiamy polecenie: ls –l | sort –n –k5. Wykorzystujemy do tego łącze nienazwane.

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

int main(void) {

pid_t childpid;
int fd[2];

if ((pipe(fd) == -1) || ((childpid = fork()) == -1)) {
     perror("Failed to setup pipeline");
     return 1;
}

if (childpid == 0) { /* ls jest potomkiem */
     if (dup2(fd[1], STDOUT_FILENO) == -1)
               perror("Failed to redirect stdout of ls");
     else
       if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
          perror("Failed to close extra pipe descriptors on ls");
       else {
               execl("/bin/ls", "ls", "-l", NULL);
               perror("Failed to exec ls");
               }
       return 1;
    }

if (dup2(fd[0], STDIN_FILENO) == -1) /* sort jest rodzicem */
     perror("Failed to redirect stdin of sort");
else
if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
  perror("Failed to close extra pipe file descriptors on sort");
else { /* wszystko OK. */
     execl("/bin/sort", "sort", "-n", "-k5", NULL);
     perror("Failed to exec sort");
}
return 0;
}
```

Pytanie: Co będzie wyświetlane na wyjściu, jeśli deskryptory fd[0] i fd[1] nie będą zamknięte przed wywołaniem execl?

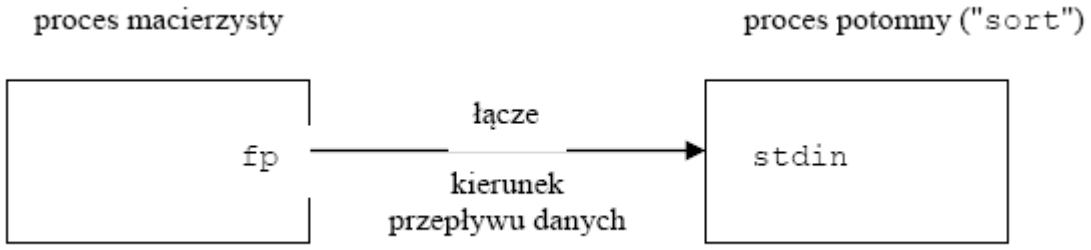
8.4 Funkcje `popen` i `pclose` – biblioteka standardowa we-wy

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);
int pclose(FILE *fp);
```

Funkcja `popen` tworzy łącze i inicjuje inny proces (*cmdstring*), który albo będzie czytał z łącza, albo do niego zapisywał. Polecenie *cmdstring* jest wykonywane jako polecenie shellowe.

- Przykład:

```
fp=popen("sort","w");
```



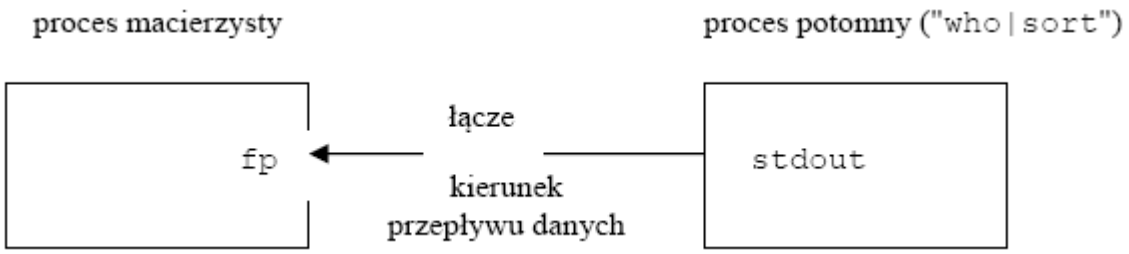

```
#include <stdio.h>
#include <unistd.h>

int main () {
 FILE* strumien = popen("sort","w");

 fprintf (strumien, "Witam.\n");
 fprintf (strumien, "Welcome.\n");
 fprintf (strumien, "Bienvenue.\n");
 fprintf (strumien, "Willkommen.\n");
 fflush (strumien);
 return pclose(strumien);
}
```

- Przykład:

```
fp=popen("who|sort","r");
```




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

int main() {
FILE *fp;
char buf[100];
int i = 0;

fp = popen( "who|sort", "r" );
while ( fgets( buf, 100, fp ) != NULL )
     printf("%3d %s", i++, buf );
pclose( fp );
return 0;
}
```

- Wady:
    - mała efektywność - utworzenie procesu potomnego to za każdym razem uruchomienie shella i zastąpienie go właściwym poleceniem

- Przykład: Dane przetwarzane przez główną aplikację są wstępnie przetwarzane przez program pomocniczy. (Stevens, Programowanie w środowisku Unix, str. 518)

```
/* program pomocniczy - filtr */
#include <ctype.h>
#include <stdio.h>

int main(void){
int c;
while ( (c = getchar()) != EOF) {
     if (isupper(c)) c = tolower(c);
     if (putchar(c) == EOF) {
          fprintf(stderr,"%s\n","blad wyjscia");
          exit(1);
                    }
     if (c == '\n')
          fflush(stdout);
     }
exit(0);
}
```

```
/* główna aplikacja */
#include <sys/wait.h>
#include <stdio.h>
#define MAXW 80

int main(){
char wiersz[MAXW];
FILE *we;

if ( (we = popen("./filtr", "r")) == NULL) {
     perror("popen: blad"); exit(1);
}
for ( ; ; ) {
     fputs("prompt> ", stdout);
     fflush(stdout);
     if (fgets(wiersz, MAXW, we) == NULL) /* czytaj z potoku */
          break;
     if (fputs(wiersz, stdout) == EOF){
          fprintf(stderr,"%d\n","blad przesłania do potoku");
     exit(1);
                                    }
          }
if (pclose(we) == -1) { perror("pclose"); exit(1); }
putchar('\n');
exit(0);
}
```

8.5 Łącza nazwane (kolejki FIFO)

- Łącze nazwane jest to plik specjalny, który zachowuje się tak jak potok. Dane są buforowane przez jądro, nie są przechowywane na dysku!

- Łącze nazwane ma nazwę, zatem może być otwarte i dostępne dla dowolnego procesu, który zna nazwę łącza i ma odpowiednie prawa dostępu do łącza.

- Łącze istnieje aż do jawnego usunięcia.

8.5.1 Tworzenie łącza

- Łącze nazwane może być utworzone na dwa sposoby:

    - za pomocą funkcji systemowej

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

Funkcja `mkfifo` tworzy łącze nazwane (kolejkę FIFO, *ang. named pipe*) o nazwie *`pathname`*, które ma prawa dostępu określone w argumencie *`mode`*. Jeśli łącze zostanie utworzone zwracane jest 0, w przeciwnym wypadku -1 i ustawiana zmienna `errno`.

    - za pomocą polecenia:

```
$ mkfifo /tmp/fifo
$ ll /tmp/fifo
prw-rw-rw- 1 user1 users 0 Nov 19 15:01 /tmp/fifo
```

8.5.2 Dostęp do plików FIFO

- Dostęp do plików FIFO jest realizowany tak, jak do zwykłego pliku. Aby można było przesyłać dane za pomocą pliku FIFO, jeden program musi otworzyć go do czytania, zaś drugi do pisania.
- Można używać:
  - funkcje poziomu niższego (`open, write, close`, itd.) ,
  - funkcje biblioteczne we-wy języka C (`fopen, fprintf, fscanf, fclose`, itd.)

```
int fd = open(nazwa_fifo,O_WRONLY);
write(fd, bufor, ile);
close (fd);

FILE* fifo=fopen(nazwa_fifo,"r");
fscanf(fifo,"%s",bufor);
fclose(fifo);
```

- Wywołanie `read()` dla łącza FIFO, które nie jest już otwarte do zapisu, zwraca koniec pliku (wartość 0).
- Wywołanie `read`() dla łącza FIFO, które jest otwarte do zapisu i puste:
  - blokujące – czeka na nadejście danych,
  - nieblokujące – zwraca -1 i `errno` przyjmuje wartość `EAGAIN`.
- Wywołanie `write()` dołącza dane na koniec łącza.

- Przykład: Prowadzenie dziennika w oparciu o model klient - serwer. Klient zapisuje informację (PID procesu, czas) do nazwanego potoku. Serwer pobiera tę informację i zapisuje do pliku (dziennika) określonego w wierszu wywołania.

- Pytania:
  - Jaką zaletę ma to rozwiązanie w stosunku do zwykłego zapisywania do określonego pliku?
  - Dlaczego w serwerze potok nazwany jest tworzony z prawami `read/write`?
  - W jaki sposób zapewnić nieskończone działanie serwera? Również wtedy, kiedy nie ma działającego klienta?
  - Jak zapewnić to, aby komunikaty pochodzące od różnych klientów pojawiały się w dzienniku w całości, tzn. nie były rozdzielana komunikatami pochodzącymi od innych klientów?

```
/* Serwer */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#define FIFOARG 1
#define FIFO_PERMS (S_IRWXU | S_IWGRP| S_IWOTH)

int main (int argc, char *argv[]) {
int fd;
if (argc != 2) {
     fprintf(stderr, "Usage: %s fifoname > logfile\n", argv[0]);
     return 1;
     }
if ((mkfifo(argv[FIFOARG], FIFO_PERMS) == -1) &&
           (errno != EEXIST)) {
     perror("Server failed to create a FIFO");
     return 1;
     }
if ((fd = open(argv[FIFOARG], O_RDWR)) == -1) {
     perror("Server failed to open its FIFO");
     return 1;
     }
/* czytaj z łącza i zapisuj na standardowe wyjście
funkcję trzeba napisać samemu */

copyfile(fd, STDOUT_FILENO);
return 1;
}
=============================================================
```

```
/* Klient */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#include <limits.h> /* PIPE_BUF */
#define FIFOARG 1

int main (int argc, char *argv[]) {
time_t curtime;
int len;
char buf[PIPE_BUF];
int fd;

if (argc != 2) {
     fprintf(stderr, "Usage: %s fifoname", argv[0]);
     return 1;
     }
if ((fd = open(argv[FIFOARG], O_WRONLY)) == -1) {
     perror("Client failed to open log fifo for writing");
     return 1;
     }
curtime = time(NULL);
snprintf(buf, PIPE_BUF, "%d: %s", (int)getpid(),
     ctime(&curtime));
len = strlen(buf);
if (r_write(fd, buf, len) != len) {
     perror("Client failed to write");
     return 1;
}
close(fd);
return 0;
}
```

- Uwaga: Funkcja `r_write()` opakowuje funkcję `write()` - jeśli zostało zapisane mniej niż *`len`* bajtów ponownie jest wywoływane `write()`.