Łącza nienazwane(potoki)

- Przykład:

```
$ ls | more
```

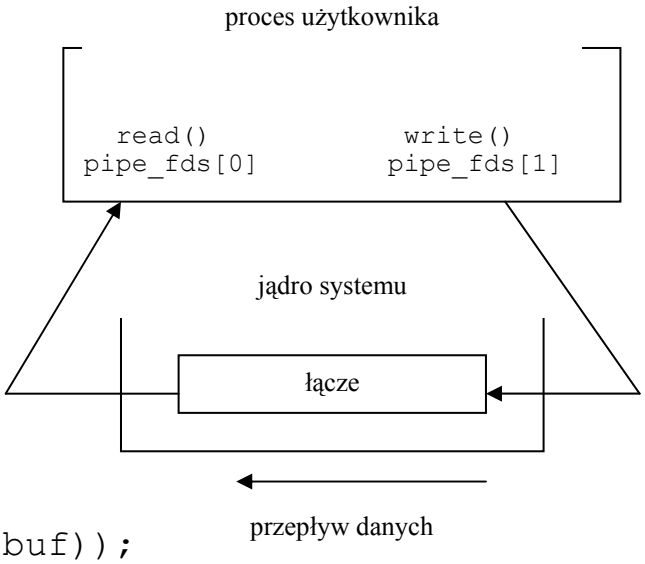
- *Łącza nienazwane mogą być używane tylko pomiędzy procesami ze sobą powiązanymi.*

Tworzenie łącza

```
#include <unistd.h>
int pipe(int filedes[2]);
```

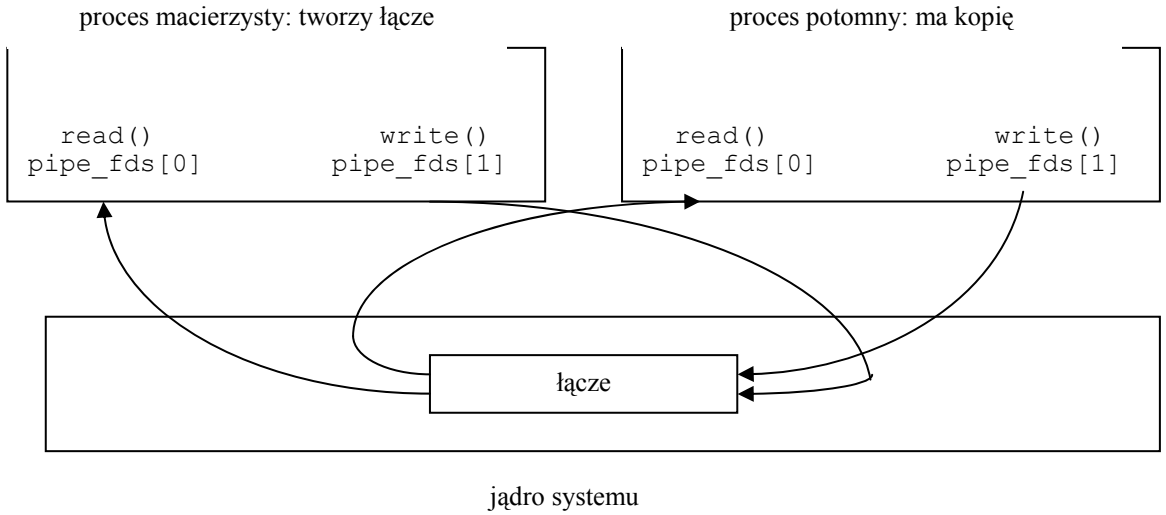
- Przykład:

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

if (pipe(fd) !=0 ) {
  perror("pipe()");
  ...
}
r_fd=fd[0];
w_fd=fd[1];
...
read(r_fd,buf,sizeof(buf));
```

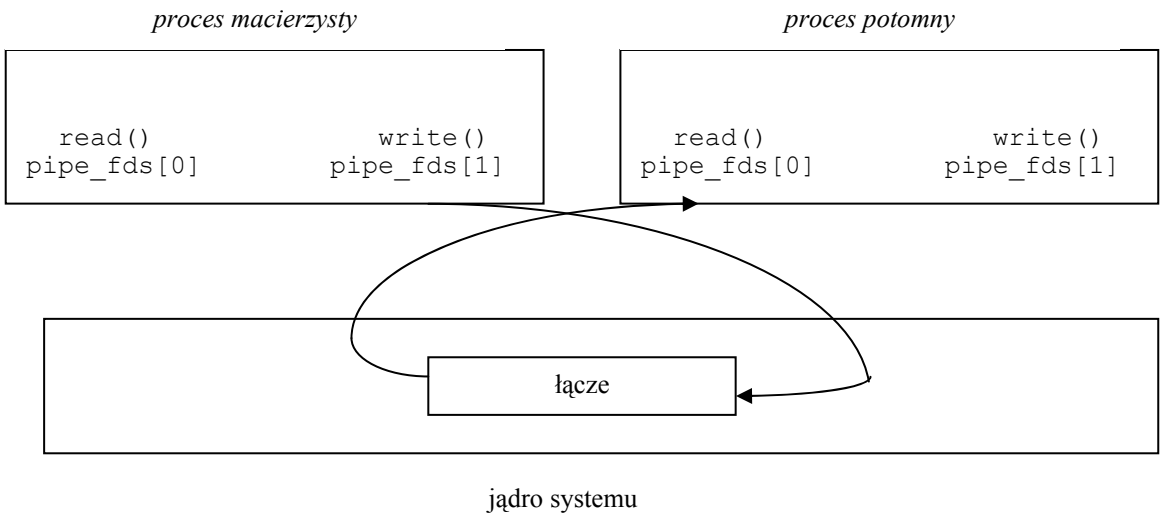


- Stała PIPE_BUF
- Sygnał SIGPIPE
- Błąd EPIPE

Wykorzystanie łącza do komunikacji między procesem macierzystym i potomnym



- Przykład: Przesyłanie komunikatów od procesu macierzystego do procesu potomnego.



- Przykład: Proces macierzysty pisze do potoku, proces potomny czyta i wyświetla na ekranie. Obydwa 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) {
    FILE* strumien;
    close (fd[1]);
    strumien = fdopen(fd[0], "r");
    odczytywanie(strumien);
    close(fd[0]);
  }
/ * proces macierzysty */
  else {
    FILE* strumien;
    close(fd[0]);
    strumien = fdopen(fd[1], "w");
    zapisywanie("Witam!", 5, 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`.

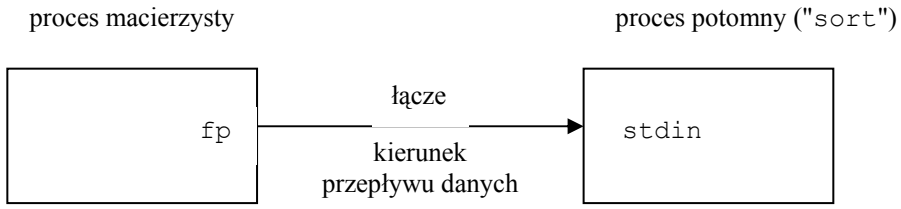
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main ()
{
  int fd[2];
  pid_t pid;
  pipe(fd);
  pid = fork ();
  if (pid == (pid_t) 0) {
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    close(fd[0]);
    execlp("sort", "sort", 0);
  }
  else {
    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;
}
```

Funkcje popen i pclose – biblioteka standardowa we-wy

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

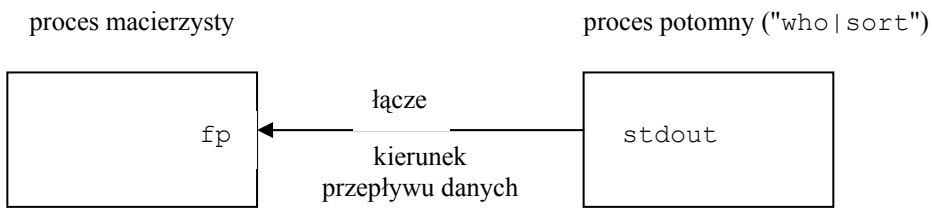
- Przykład:

```
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);
}
```

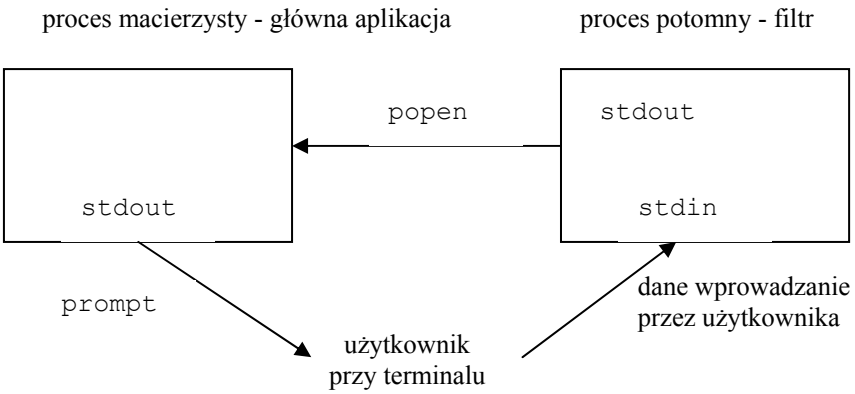
- Przykład:
  `fp=popen("who|sort","r");`




```
#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)
    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);
}
```

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.

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
```

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.