Potoki (Łącza nienazwane)

W systemie UNIX wyróżnia się dwa rodzaje łączy: *łącza nazwane* — kolejki FIFO — i *łącza nienazwane*, określane często 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. 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, a wynika ona z różnic w tworzeniu i otwieraniu łączy.

Potok nienazwany jest plikiem specjalnym, służącym do komunikacji pomiędzy procesami.

- Komunikacja za pośrednictwem potoków polega na tym, że jeden proces zapisuje informacje do łącza podobnie jak do pliku, jest jednak usypiany, jeśli łącze jest pełne[1], a inny proces ją odczytuje podobnie jak z pliku lub jest usypiany, jeśli łącze jest puste.
- Informacja jest **odczytywana w tej samej kolejności, w jakiej została zapisana**, a **po odczycie jest usuwana z potoku** (nie może być odczytana ponownie),
- Na łączu wykonujemy tylko operacje odczytu / zapisu, nie można przemieszczać wskaźnika bieżącej pozycji (nie można wykonywać funkcji `fcntl`)
- Potoki nie mogą być identyfikowane przez nazwę: procesy żeby móc komunikować się poprzez łącze muszą znać jego deskryptory (za pomocą potoku mogą się więc komunikować procesy z których jeden utworzył potok, a następnie utworzył pozostałe komunikujące się procesy, które w ten sposób otrzymają w tablicy otwartych plików deskryptory istniejącego łącza)
- Potok jest identyfikowany przez dwa deskryptory : do odczytu i do zapisu

Przeadresowanie standardowego wejścia i wyjścia:

1. Kiedy dane wejściowe mają być pobierane z określonego pliku, a nie ze standardowego wejścia to plik ten musi mieć deskryptor o numerze 0.
2. Kiedy dane wyjściowe mają zostać zapisane w pliku, to plik ten musi mieć deskryptor o numerze 1
3. Funkcje przydzielające deskryptor (`open, pipe, dup`) wykorzystują wolny deskryptor o najmniejszej wartości

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

[1] Wyjątek gdy jest ustawiona flaga O_NDELAY

Funkcja PIPE

```
PROTOTYPE: int pipe(int pdesk[2]);
RETURNS: success : 0
          error: -1
errno = EMFILE (w procesie używanych jest zbyt wiele
               deskryptorów pliku)
        ENFILE (tablica plików systemu jest pełna)
        EFAULT (pdesk jest nieprawidłowy)
```

PARAMETRY:
1. `pdesk[0]` – deskryptor potoku do odczytu
2. `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. Jedynym sposobem przekazania deskryptorów łącza jest odziedziczenie kopii tablicy deskryptorów przez proces potomny od procesu macierzystego. 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 potok, a 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.

Funkcja READ

```
PROTOTYPE: int read( inf fd, void *buf, size_t count );
  RETURNS: success : rzeczywista liczba bajtów, jaką udało
                     się odczytać
           error: -1
```

PARAMETRY:

1. `fd`- deskryptor potoku z którego mają zostać odczytane dane
1. `buf` – adresem bufora znajdującego się w segmencie danych procesu, do którego zostaną przekazane dane odczytane z potoku w wyniku wywołania funkcji `read`
2. `count` – ilość bajtów do odczytania

UWAGI:

1. Jeśli wszystkie deskryptory do zapisu są zamknięte i łącze jest puste, to zostaje zwrócona wartość 0.
2. 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.
3. 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 zostanie zatrzymany 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**.
4. 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.

Funkcja WRITE

```
PROTOTYPE: int write( inf fd, void *buf, size_t count );
  RETURNS: success : rzeczywista liczba bajtów, jaką udało
                      się odczytać
           error: -1
```

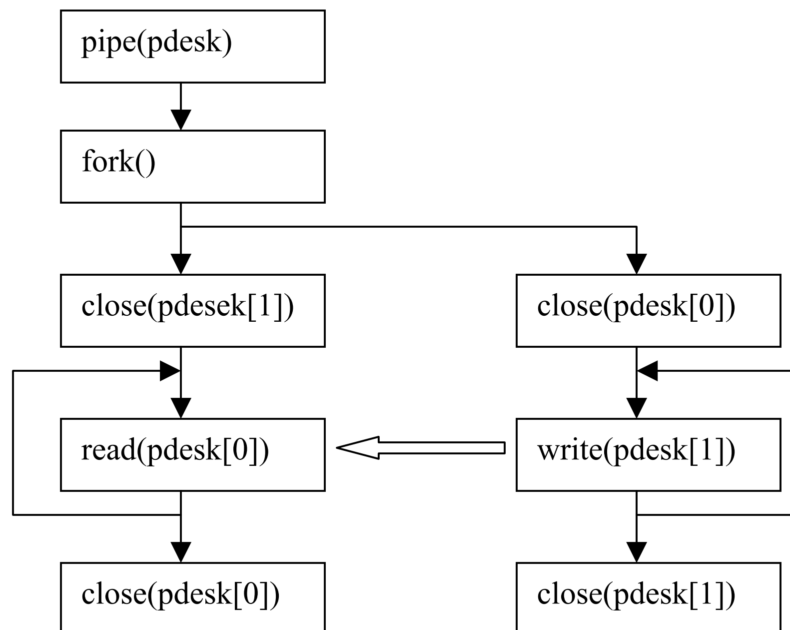
PARAMETRY:

1. `fd`- deskryptor potoku do którego mają zostać zapisane dane

2. `buf` – adresem bufora znajdującego się w segmencie danych procesu, z którego zostaną pobrane dane zapisane przez funkcję `write`
3. `count` – ilość bajtów do zapisania

UWAGI:
1. Funkcja zapisuje w potoku `count` bajtów w całości. (nie przeplatają się one z danymi pochodzącymi z innych zapisów)
2. 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[2], tzn. aż inny proces odczyta i tym samym usunie dane z łącza

Schemat komunikacji przez potok:



[2] Proces nie zostanie zablokowany wówczas, gdy dla odpowiedniego deskryptora zostanie ustawiona flaga O_NDELAY

Kolejki FIFO (Łącza nazwane)

- Łącze nazwane ma dowiązanie w systemie plików (istnieje jako plik w jakimś katalogu)
- Łącze nazwane może być identyfikowane poprzez nazwę
- Procesy nie spokrewnione ze sobą mogą przekazywać dane poprzez łącze nazwane

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

Funkcja MKFIFO

```
PROTOTYPE: int mkfifo( char * path, mode_t mode );
  RETURNS: success :0
           error: -1
```

PARAMETRY:

1. `path` – nazwa ścieżkowa pliku specjalnego będącego kolejką fifo
2. `mode` – prawa dostępu do łącza

UWAGI:

Funkcja tworzy (**ALE NIE OTWIERA**) plik typu kolejka FIFO

Funkcja OPEN

```
PROTOTYPE: int open ( char *path, int flags);
  RETURNS: success : deskryptor kolejki FIFO
           error: -1
```

PARAMETRY:

1. `path` – nazwa ścieżkowa pliku specjalnego będącego kolejką fifo
2. `mode` – prawa dostępu do łącza
3. `flags` – określenie trybu w jakim jest otwierana kolejka:
   O_RDONLY – tryb tylko do odczytu
   O_WRONLY– tryb tylko do zapisu

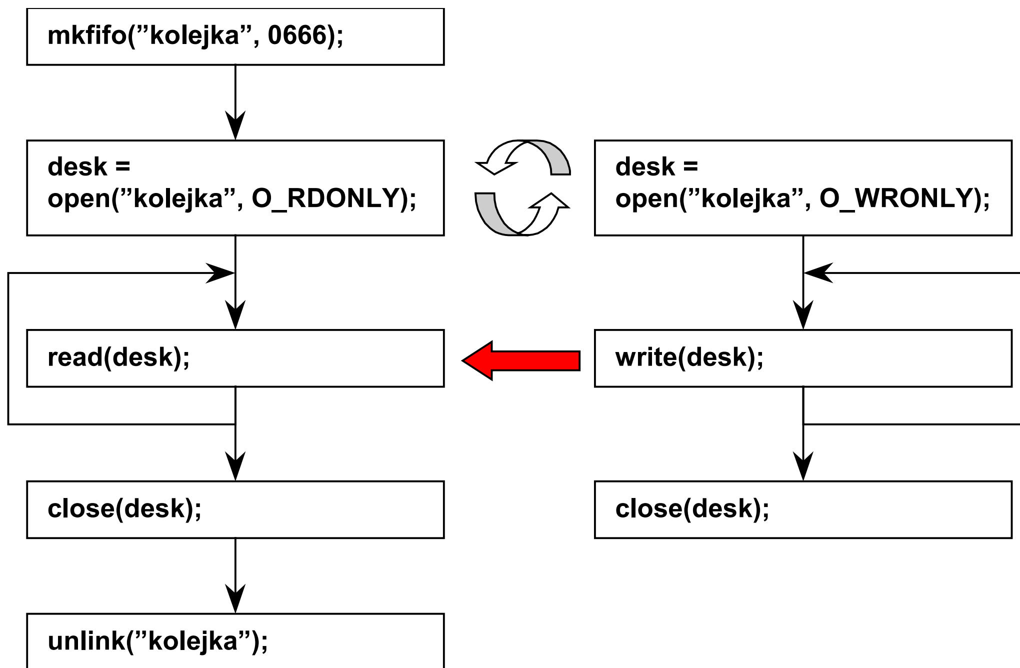
UWAGI:

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. Proces jest zatem tak

długo blokowany w funkcji open(), aż inny proces nie wywoła funkcji open() w sposób komplementarny[1].

Odczyt i zapis danych za pomocą funkcji: READ, WRITE, jak dla plików

Schemat komunikacji przez kolejkę FIFO:



[1] Proces nie jest blokowany, jeżeli łącze jest otwierane w trybie O_NDELAY.