

# Systemy operacyjne

## Sprawozdanie - laboratorium 6 „Komunikacja międzyprocesowa 1”

Andrzej Kołakowski  
296586

### 1) Tworzenie prostych łączy jednokierunkowych

#### 1. Funkcja `pipe(2)`. Analiza działania funkcji `write`, `read` i `close` w odniesieniu do operacji na łączach.

Funkcja `pipe` służy do tworzenia potoku – jednokierunkowego kanału danych, który może zostać wykorzystany do komunikacji międzyprocesowej. Funkcja zapisuje w tablicy podanej jako argument dwa deskryptory plików – końce potoku (jeden do odczytu, drugi do zapisu).

Jeżeli proces spróbuje odczytać z pustego potoku, to `read` zablokuje wykonanie wątku dopóki nie napłyną nowe dane.

Jeżeli proces spróbuje zapisać do pełnego potoku, to `write` zablokuje wykonanie wątku dopóki wystarczająca ilość danych nie zostanie odczytana z potoku, tak aby udał się zapis.

Jeżeli wszystkie deskryptory plików opisujące końcówkę do zapisu zostaną zamknięte, to wywołanie `read` na przeciwnej końcówce zwróci **EOF**.

Jeżeli wszystkie deskryptory plików opisujące końcówkę do odczytu zostaną zamknięte, to wywołanie `write` na przeciwnej końcówce spowoduje wysłanie sygnału **SIGPIPE**.

Aplikacja wykorzystująca `pipe(2)` powinna używać odpowiednich wywołań `close(2)` aby zamknąć niepotrzebne duplikaty deskryptorów plików. Tylko wtedy **EOF** oraz **SIGPIPE** zostaną poprawnie dostarczone.

*Opracowane na bazie man 7 pipe*

#### 2. Modyfikacja programu `simple_pipe.c`

- Podział na 2 procesy używając `fork`
- Zamknięcie zbędnych łączy za pomocą `close`
- Przekazanie komunikatu między procesami
- Sprawdzenie wielkości bufora potoków w systemie

```
root@localhost:~/Desktop/so/lab6
File Edit View Search Terminal Help
[root@localhost lab6]# ./simple_pipe
Ur beautiful pipe example (25 bytes)
fpathconf for read:4096 write:4096
[root@localhost lab6]#
```

### *simple\_pipe.c*

Ważnym elementem jest zamykanie zbędnych końcówek – zarówno od odczytu jak i od zapisu.

Tak długo jak dowolny proces ma otwartą końcówkę do zapisu, nawet jeżeli jedyny taki proces to ten który próbuje z końcówki odczytać, system założy że wciąż może nastąpić zapis i nie zgłosi **EOF**.

Podobnie, jeżeli potok jest pełny, i istnieje dowolny proces z otwartą końcówką do odczytu, nawet jeżeli jedyny taki proces to ten który próbuje do końcówki zapisać, to zapis zostanie zatrzymany tak długo, aż nie zwolni się miejsce.

Wielkość bufora mówi o maksymalnej ilości danych z gwarancją atomicznego zapisu do potoku. Zapis więcej niż `PIPE_BUF` bajtów może być nieatomiczny, tzn. jądro może przepłścić te dane z danymi zapisywanymi przez inne procesy.

Jako ciekawostkę dodam, że robiąc to zadanie znalazłem błąd w manualu.

```
FPATHCONF(3)                                Linux Programmer's Manual                                FPATHCONF(3)

NAME
    fpathconf, pathconf - get configuration values for files

SYNOPSIS
    #include <unistd.h>

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

_PC_PIPE_BUF
    The maximum number of bytes that can be written atomically to a
    pipe or FIFO. For fpathconf(), fd should refer to a pipe or
    FIFO. For fpathconf(), path should refer to a FIFO or a direc-
    tory; in the latter case, the returned value corresponds to
    FIFOs created in that directory. The corresponding macro is
    _POSIX_PIPE_BUF.
```

Dwa razy wymieniona jest funkcja `fpathconf`, zamiast za drugim razem `pathconf`.

## 2) Praca z łączami komunikacyjnymi

### 1. Funkcje `dup` i `dup2`

Funkcje `dup` i `dup2` duplikują deskryptor pliku.

Funkcja `dup` tworzy kopię deskryptora pliku podanego w argumencie używając najniższego wolnego numeru deskryptora dla nowego deskryptora.

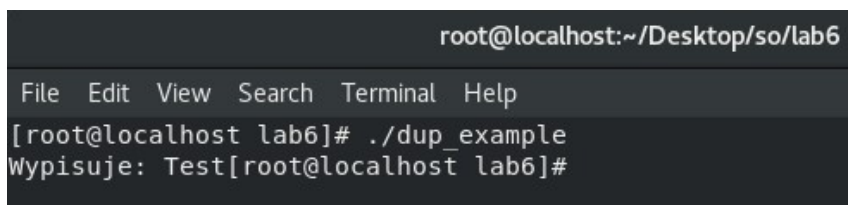
Funkcja `dup2` działa tak samo jak `dup`, ale zamiast używać najniższego wolnego numeru deskryptora, używa numer deskryptora podany w argumencie. Jeżeli deskryptor podany w argumencie jest otwarty, to przed wykorzystaniem w funkcji zostaje zamknięty.

Operacja zamykania i ponownego wykorzystania deskryptora w funkcji `dup2` wykonywana jest atomicznie, w przeciwieństwie do użycia `close(2)` i `dup`, gdzie deskryptor może zostać w międzyczasie wykorzystany przez inny wątek.

### 2. Modyfikacja programu `dup_example.c`

- Dodanie obsługi błędów dla funkcji `pipe`
- Dodanie funkcji `close` zamykających otwarte deskryptory
- Wykorzystanie funkcji `dup2` zamiast `dup`
- Do czego w procesie potomnym użyta jest funkcja `close(0)`?

Funkcja `close` wykorzystana jest do zamknięcia deskryptora pliku 0, czyli standardowego wejścia. Po wywołaniu funkcji `dup/dup2` jego miejsce zajmie końcówka potoku.



```
root@localhost:~/Desktop/so/lab6
File Edit View Search Terminal Help
[root@localhost lab6]# ./dup_example
Wypisuje: Test[root@localhost lab6]#
```

*dup\_example.c*

## 3) Ćwiczenia dotyczące łącz nienazwanych

### 1. Komunikacja z kilkoma procesami – program `3_1.c`

- Tworzy proces czytający dane z pliku (słownika) i dwa podprocesy
- Jeden liczy liczbę linii (liczbę słów w słowniku)

- Drugi liczy liczbę linii zawierających podciąg „pipe” (liczbę słów zawierających słowo „pipe”)
- Po policzeniu podprocesy przekazują dane do procesu macierzystego, który wypisuje wyniki na ekran

```

root@localhost:~/Desktop/so/lab6
File Edit View Search Terminal Help
[root@localhost lab6]# cat dictionary.txt | wc -l
80368
[root@localhost lab6]# cat dictionary.txt | grep pipe | wc -l
32
[root@localhost lab6]# ./3_1
Number of lines: 80368
Number of lines with "pipe": 32
[root@localhost lab6]#

```

*3\_1.c*

## 2. Przekazywanie danych przez kilka łączy – program 3\_2.c

- Tworzy dwa procesy potomne
- Pierwszy generuje kolejne liczby od 1 do 10 wykorzystując seq i za pomocą łącza nienazwanego przekazuje do drugiego
- Drugi mnoży każdą z otrzymanych liczb przez 5 i przekazuje liczby do procesu macierzystego, który wyświetla otrzymane liczby

```

root@localhost:~/Desktop/so/lab6
File Edit View Search Terminal Help
[root@localhost lab6]# ./3_2
5
10
15
20
25
30
35
40
45
50
[root@localhost lab6]#

```

*3\_2.c*

## 3. Przykładowy potok z Unixa – program 3\_3.c

- Realizuje następujący potok:

```
who | cut -d' ' -f1 | sort | uniq -c | sort -r
```

```
root@localhost:~/Desktop/so/lab6
File Edit View Search Terminal Help
[root@localhost lab6]# who | cut -d' ' -f1 | sort | uniq -c | sort -r
    1 root
[root@localhost lab6]# ./3_3
    1 root
[root@localhost lab6]#
```

*3\_3.c*

- Ciekawsza wersja z `ls -l` zamiast `who`:

```
root@localhost:~/Desktop/so/lab6
File Edit View Search Terminal Help
[root@localhost lab6]# ls -l | cut -d' ' -f1 | sort | uniq -c | sort -r
    9 -rw-r--r--.
    6 -rwxr-xr-x.
    1 total
[root@localhost lab6]# ./3_3_ls
    9 -rw-r--r--.
    6 -rwxr-xr-x.
    1 total
[root@localhost lab6]#
```

*3\_3\_ls.c*