Systemy Operacyjne 1
Laboratorium 3
„Potoki i łącza nazwane w Linuksie”
(jeden tydzień)

dr inż. Arkadiusz Chrobot

16 października 2020

Wstęp

W tej instrukcji zawarte są informacje na temat jednych z podstawowych środków komunikacji między procesami, które dostępne są w systemie Linux. Pierwsza cześć przedstawia koncepcję przesyłania informacji z użyciem strumieni. W drugiej części zawarto opis użycia potoków. Trzecia część poświęcona jest łączom nazwanym, które znane są także jako kolejki FIFO. Czwarta część zawiera opis funkcji związanych z obsługą wymienionych łącz. Instrukcja kończy się listą zadań do samodzielnego wykonania na laboratorium.

1. Komunikacja z wykorzystaniem strumieni

W systemie Linux istnieje możliwość stworzenia przez proces nowego procesu realizującego wybrane polecenie powłoki oraz łącza komunikacyjnego, nazwanego strumieniem, które będzie łączyło go z procesem rodzicielskim. Umożliwia to funkcja `popen()`. Łącze utworzone przy jej pomocy jest jednokierunkowe, tzn. proces rodzicielski może je wyłącznie odczytywać lub wyłącznie zapisywać, nie może wykonywać obu tych czynności. Strumień, po zakończeniu korzystania z niego, należy zamykać za pomocą funkcji `pclose()`. Jest on powiązany ze standardowym wejściem lub wyjściem polecenia realizowanego w ramach procesu potomnego, a więc, w zależności od tego czy jest czytany, czy zapisywany, „zastępuje" klawiaturę lub ekran.

2. Łącza nienazwane - potoki

Innym środkiem komunikacji jednokierunkowej jest łącze nienazwane, czyli inaczej potok (ang. *pipe*). Służy on do wymiany informacji między dwoma spokrewnionymi procesami, czyli np. między potomkiem i rodzicem, lub dwoma potomkami. Istnieje także mało praktyczna możliwość korzystania z potoku w obrębie jednego procesu. Potok tworzony jest przez system operacyjny na rzecz procesu użytkownika, który wywołuje funkcję `pipe()`. Dostęp do tego łącza proces użytkownika uzyskuje za pomocą funkcji umożliwiających niskopoziomowe operacje na plikach - `read()` i `write()`. Pojemność potoku jest ograniczona, a wielkość tego ograniczenia zależy od konfiguracji systemu. Dla utworzonego potoku można ustawić znacznik (flagę) `O_NONBLOCK`, który powoduje, że program nie będzie przechodził w stan oczekiwania w określonych sytuacjach, związanych z obsługą potoku (szczegóły w następnym punkcie).

Listing 1 przedstawia program, który tworzy dwa procesy (rodzic i potomek), które komunikują się ze sobą przy pomocy potoku. Procesem nadającym informacje, jest potomek, a odbierającym - rodzic.

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

enum names_of_pipe_descriptors {READ, WRITE};


void do_parent_work(int descriptors[])
{
        char message[100] = "";
        if(close(descriptors[WRITE])<0)
                perror("parent - close write");
        if(read(descriptors[READ],message,sizeof(message))<0)
                perror("read");
        printf("Odebrana wiadomość: %s\n",message);
        if(close(descriptors[READ])<0)
                perror("parent - close read");
        if(wait(0)<0)
                perror("wait");
}

void do_child_work(int descriptors[])
{
        char *message = "Systemy Operacyjne";
        if(close(descriptors[READ])<0)
                perror("child - close read");
        if(write(descriptors[WRITE],message,strlen(message))<0)
                perror("write");
        if(close(descriptors[WRITE])<0)
                perror("child - close write");
        exit(0);
}

int main(void)
{
        int pipe_descriptors[2];
        if(pipe(pipe_descriptors)<0)
                perror("pipe");
        int pid = fork();
        if(pid==-1)
                perror("fork");
        if(pid==0)
                do_child_work(pipe_descriptors);
        else
                do_parent_work(pipe_descriptors);
        return 0;
}
```

Listing 1: Przykład prostej komunikacji za pomocą potoku

Aby uprościć identyfikację deskryptorów potoku zdefiniowany został typ wyliczeniowy. Indeks w tablicy deskryptorów identyfikujący deskryptor do odczytu (`0`) został oznaczony elementem `READ`, a identyfikujący deskryptor do zapisu, elementem `WRITE`.

Proces potomka nadaje do rodzica wiadomość „Systemy Operacyjnej". Aby to uczynić najpierw zamyka deskryptor do odczytu, bo nie będzie wykonywał na potoku tej operacji, a następnie, za pomocą funkcji `write()` zapisuje do potoku, korzystając z deskryptora do zapisu, wspomnianą wiadomość, po czym zamyka również deskryptor do zapisu i kończy swoje działanie. Proces rodzicielski dysponuje buforem na `100` znaków, który po zadeklarowaniu jest inicjowany pustym łańcuchem, a wiec wszystkie jego elementy będą zawierały znak `'\0'`. Rodzic zamyka najpierw deskryptor potoku do zapisu, następnie przy pomocy funkcji `read()` i deskryptora do odczytu próbuje z potoku odczytać `100` znaków. Jednakże w potoku będzie mniej znaków i funkcja tylko tyle umieści w buforze `message` i zakończy swe działanie, nie sygnalizując wyjątku. Rodzic następne wypisze te znaki na ekran, poczeka na zakończenie potomka i również się zakończy.

3. Łącza nazwane - kolejki FIFO

Kolejka FIFO jest podobna w działaniu do potoku, ale w przeciwieństwie do niego ma nazwę, a więc mogą z niej korzystać procesy niespokrewnione, w podobny sposób, jak mogą korzystać z pliku. Kolejki FIFO pojawiły się oficjalnie w systemach uniksowych zgodnych z oryginalną wersją Uniksa o nazwie System III. Pierwotnie tworzono je za pomocą funkcji `mknod()`, ale obecnie istnieje wygodniejsza w użyciu funkcja `mkfifo()`. Aby skorzystać z tak utworzonego łącza nazwanego jeden z procesów musi otworzyć je albo wyłącznie do zapisu, albo wyłącznie do odczytu (to samo ograniczenie co w przypadku potoku). Dodatkowo można zastosować flagę `O_NONBLOCK`. Powoduje to, że proces nie będzie oczekiwał na zakończenie pewnych operacji związanych z obsługą kolejki. Dokładniej objaśnia to tabela 1, której dwa ostatnie wiersze odnoszą się także do potoków.

Sytuacja	**Nie ustawiono `O_NONBLOCK`**	**Ustawiono `O_NONBLOCK`**
Próba otwarcia kolejki tylko do czytania, żaden proces nie otworzył jej do pisania.	Oczekiwanie, aż drugi proces otworzy kolejkę do pisania.	Natychmiastowe zakończenie działania funkcji otwierającej bez sygnalizowania błędu.
Próba otwarcia kolejki tylko do pisania, żaden proces nie otworzył jej do czytania.	Oczekiwanie, aż drugi proces otworzy kolejkę do czytania.	Natychmiastowe zakończenie działania funkcji otwierającej z sygnalizacją błędu.
Próba czytania z kolejki, w której nie ma danych.	Oczekiwanie, aż pojawią się dane. Jeśli żaden proces nie otworzył kolejki do pisania, to funkcja natychmiast zwraca wartość zero. Jeśli w kolejce pojawią się dane, to funkcja zwróci liczbę oczytanych bajtów.	Funkcja odczytująca natychmiast kończy działanie, zwracając wartość `-1`. Jeśli kolejka nie jest otwarta do zapisu, to funkcja również natychmiast kończy działanie, ale zwraca wartość `0`.
Próba zapisania do zapełnionej kolejki.	Oczekiwanie, aż będzie miejsce do zapisania nowych danych.	Funkcja `write()` natychmiast kończy działanie zwracając wartość `0`.

Tabela 1: Zachowanie procesu dla kolejki FIFO z ustawioną i nieustawioną flagą `O_NONBLOCK`.

Dodatkowo, zapis i odczyt danych z potoków i kolejek `FIFO` podlega następującym regułom:

- Jeśli proces żąda przeczytania mniejszej porcji danych niż bieżąco znajduje się w łączu, to otrzyma dokładnie tyle danych, ile zażądał. Reszta danych nie ulega zniszczeniu i może zostać pobrana podczas kolejnej operacji odczytu.
- Jeśli proces będzie usiłował odczytać więcej danych niż znajduje się bieżąco w łączu, to odczytanych i tak zostanie tylko tyle danych, ile jest w potoku lub kolejce.

- Jeśli proces próbuje czytać z łącza, które nie zostało przez żaden inny proces otwarte do pisania, to funkcja `read()` zwróci zero. W przypadku kiedy ustawiony jest znacznik `O_NONBLOCK` zachowanie funkcji `read()` jest takie samo.
- Zapis danych jest operacją niepodzielną, o ile proces zapisuje dane do łącza porcjami mniejszymi od jego pojemności.
- Jeśli proces próbuje zapisywać do łącza, które nie zostało otwarte przez inny proces do odczytu, to otrzyma sygnał `SIGPIPE`, którego domyślna obsługa polega na zakończeniu procesu.

Po skorzystaniu z kolejki FIFO należy ją usunąć, aby nie zajmowała miejsca na dysku.

Listingi 2 i 3 zawierają kody źródłowe programów komunikujących się przy pomocy kolejki FIFO.

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

void read_from_fifo(int descriptor)
{
        char message[100]="";
        if(read(descriptor,message,sizeof(message))<0)
                perror("read");
        printf("Odebrana wiadomość: %s\n",message);
}

int main(void)
{
        if(mkfifo("fifo",0600)<0)
                perror("mkfifo");
        int descriptor = open("fifo",O_RDONLY);
        if(descriptor<0)
                perror("open");
        read_from_fifo(descriptor);
        if(close(descriptor)<0)
                perror("close");
        if(unlink("fifo")<0)
                perror("unlink");
        return 0;
}
```

Listing 2: Przykład prostej komunikacji za pomocą kolejki FIFO - odbiorca

Program odbiorcy tworzy kolejkę, otwiera ją do odczytu, następnie próbuje z niej odczytać `100` znaków, czyli tyle, ile wynosi pojemność jego bufora. Jeśli nie będzie w kolejce informacji, to będzie czekał aż się pojawią. Ostatecznie nadawca umieści w niej mniej niż `100` znaków, ale zostaną one odczytane przez odbiorcę i wyświetlone na ekranie, po czym zamknie on kolejkę i usunie ją z systemu plików. Ostatnią operację powinien wyłącznie on wykonywać, ponieważ jako ostatni będzie korzystał z kolejki. Odbiorca musi być także uruchomiony przed nadawcą, ponieważ to on tworzy kolejkę. Wyjaśnienia wymaga tajemnicza liczba ósemkowa pojawiająca się jako drugi argument wywołania `mkfifo()`. Jest to wartość opisująca prawa dostępu do kolejki. Kolejne cyfry tej liczby (poza wiodącym zerem) opisują prawa odpowiednio dla: użytkownika, który jest właścicielem (w tym wypadku także twórcą) kolejki, grupy użytkowników, do której należy właściciel oraz dla pozostałych użytkowników zarejestrowanych w systemie. Ponieważ

jedna cyfra ósemkowa odpowiada trzem bitom, to każda z tych cyfr zawiera informacje o trzech prawach: `rwx`, gdzie `r` oznacza prawo do odczytu, `w` prawo do zapisu, a `x` prawo do wykonania. Zatem w zapisie `0600` cyfra `6` oznacza prawa do odczytu i zapisu (110 w zapisie binarnym) dla właściciela kolejki, a dwa ostatnie zera oznaczają, że ani grupa, ani pozostali użytkownicy nie mają żadnych praw do tej kolejki.

```c
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<sys/fcntl.h>

void write_to_fifo(int descriptor)
{
        char *message = "Systemy Operacyjne";
        if(write(descriptor,message,strlen(message))<0)
                perror("write");
}

int main(void)
{
        int descriptor = open("fifo",O_WRONLY);
        if(descriptor<0)
                perror("open");
        write_to_fifo(descriptor);
        if(close(descriptor)<0)
                perror("close");
        return 0;
}
```

Listing 3: Przykład prostej komunikacji za pomocą kolejki FIFO - nadawca

Proces nadawca powinien być uruchomiony po odbiorcy. Otwiera on kolejkę do zapisu, a następnie zapisuje w niej wiadomość dla odbiorcy, zamyka tę kolejkę i kończy swoje działanie.

4. Opisy funkcji

`popen()` - funkcja uruchamia polecenie powłoki podane w jej argumentach wywołania oraz tworzy strumień służący do komunikacji między procesem wywołanym a wywołującym, który można obsługiwać standardowymi funkcjami `fread()` i `fwrite()`.
Szczegóły: `man 3 popen`
`fread()` - służy do odczytu buforowanego ze strumienia.
Szczegóły: `man 3 fread`
`fwrite()` - służy do buforowanego zapisu do strumienia.
Szczegóły: `man 3 fwrite`
`pclose()` - służy do zamykania strumienia stworzonego przez `popen()`.
Szczegóły: `man 3 pclose`
`open()` - otwiera plik (również kolejkę `FIFO`) i zwraca liczbę go identyfikującą, nazywaną *deskryptorem*. Możliwe jest dzięki niej ustalenie trybu otwarcia i ustawienie znaczników.
Szczegóły: `man 2 open`
`pipe()` - służy do tworzenia potoku łączącego dwa spokrewnione procesy. Jako argument wywołania przyjmuje dwuelementową tablicę, w której będą zapisane deskryptory potoku. Deskryptor zerowy jest

do odczytu, a pierwszy do zapisu. Zwykle jest ona wywoływana przed `fork()`, co powoduje, że procesy powstałe w wyniku działania tej ostatniej funkcji odziedziczą tablicę deskryptorów. Każdy z tych procesów zamyka jeden z deskryptorów, np. jeśli komunikacja przebiega według schematu: rodzic → potomek, to rodzic zamyka deskryptor do odczytu, a potomek do zapisu.
Szczegóły: `man 2 pipe`
`read()` - czyta określoną liczbę bajtów z deskryptora bez buforowania.
Szczegóły: `man 2 read`
`write()` - zapisuje określoną liczbę bajtów do deskryptora bez buforowania.
Szczegóły: `man 2 write`
`close()` - zamyka deskryptor pliku. Może być użyta z potokiem lub kolejką `FIFO`. Do zamykania strumieni służy `fclose()` lub `pclose()`.
Szczegóły: `man 2 close`
`mkfifo()` - tworzy łącze nazwane o podanej nazwie i atrybutach, może być zastąpiona przez `mknod()`.
Szczegóły: `man 3 mkfifo`
`fcntl()` - służy do manipulacji deskryptorem pliku. Można dzięki niej ustawić flagę `O_NONBLOCK` dla potoku.
Szczegóły: `man 2 fcntl`
`unlink()` - funkcja ta usuwa z systemu plików nazwę, która może odnosić się do dowiązania lub pliku (ostatni przypadek odpowiada operacji usunięcia pliku). Można ją wykorzystać do automatycznego usuwania łącza nazwanego.
Szczegóły: `man 2 unlink`

5. Zadania

UWAGA! PROGRAMY NALEŻY NAPISAĆ Z PODZIAŁEM NA FUNKCJE Z PARAMETRAMI. KAŻDY PROGRAM MUSI SPRAWDZAĆ, CZY FUNKCJA, Z TYCH, KTÓRE SĄ OPISANE WYŻEJ NIE SYGNALIZUJE WYJĄTKU PODCZAS WYKONANIA, O ILE Z JEJ DOKUMENTACJI WYNIKA, ŻE MOŻE COŚ TAKIEGO ROBIĆ. PRZED ZAKOŃCZENIEM DZIAŁANIA KAŻDY PROGRAM POWINIEN ZWOLNIĆ ZASOBY, CZYLI ZAMKNĄĆ I/LUB USUNĄĆ STRUMIENIE, POTOKI I KOLEJKI FIFO.

1. Napisz program, który wywoła polecenie `sort nazwa.c`, gdzie `nazwa.c` jest nazwą pliku z jego kodem źródłowym, a następnie odczyta `50` znaków przez nie zwróconych.

2. Napisz program, który uruchomi polecenie `wc` i przekaże mu na standardowe wejście dowolny ciąg znaków.

3. Napisz program, który stworzy potok i wykorzysta go do przesyłania danych w obrębie jednego procesu.

4. Napisz program, który stworzy potok i wykorzysta go do komunikacji między dwoma spokrewnionymi procesami. Zadanie wykonaj tak, aby za pomocą argumentów wywołania programu (parametry `argc` i `argv`) można było ustalić, czy flaga `O_NONBLOCK` będzie ustawiona, czy nie.

5. Napisz program, który podzieli się na dwa procesy (rodzic i potomek) komunikujące się przy pomocy potoków. Komunikacja musi być dwukierunkowa.

6. Napisz dwa oddzielne programy (realizując oddzielny kod), które będą się komunikowały przy pomocy kolejki FIFO. Wykorzystaj argumenty wywołania tych programów (patrz zadanie 4), aby można było ustalić, czy będą one ustawiały flagę `O_NONBLOCK`, czy też nie. Po zakończeniu komunikacji jeden z procesów musi usunąć kolejkę.

7. Napisz program, w którym komunikacja między spokrewnionymi procesami będzie odbywała się w jednym kierunku za pomocą łącza nienazwanego, a w drugim za pomocą łącza nazwanego.

8. Stwórz programy do dialogu między dwoma użytkownikami pracującymi w systemie na osobnych terminalach. Do komunikacji użyj kolejki `FIFO` stworzonej za pomocą funkcji `mknod()`.

9. Napisz program, który wygeneruje cztery procesy. Każdy z tych procesów będzie się komunikował z następnym za pomocą łącza nienazwanego. Pierwszy proces wyśle przez swoje łącze liczby `1`, `2`, `3` i `4`. Każdy kolejny zwiększy każdą z nich o `1`, a ostatni z nich wypisze je na ekranie.

10. Napisz trzy programy komunikujące się przez kolejkę `FIFO`. Pierwszy program będzie wysyłał kolejne liczby parzyste, drugi kolejne liczby nieparzyste, a trzeci będzie odbierał te liczby i je sumował. Wszystkie procesy powinny wyświetlać wyniki swojej pracy na ekranie.