Systemy operacyjne – studia stacjonarne 2022/23
Lab 12. Łącza komunikacyjne: potoki nazwane – kolejki FIFO. Projekt nr 5.

1. Potoki nazwane.
   Kolejki FIFO są podobne do łączy pipe, zapewniają jednokierunkowy przepływ danych. Łączą w sobie cechy pliku i łącza. Podobnie jak plik łącze nazwane posiada swoja nazwę, co umożliwia komunikację procesom niepowiązanym ze sobą. Kolejka FIFO jest tworzona za pomocą funkcji mkfifo() lub polecenia mkfifo (powstaje plik typu p). Kolejki FIFO używa się tak jak zwykłego pliku. Aby możliwa była komunikacja za pomocą kolejki jeden program musi otworzyć ją do zapisu, a inny do odczytu – nie jest możliwe otwarcie łącza w trybie O_RDWR. Można korzystać z niskopoziomowych funkcji I/O (open(), write(), read(), close()) oraz z funkcji I/O biblioteki C (fopen(), fprintf(), fscanf(), fclose()).

   Do wywołania funkcji niezbędne są następujące pliki nagłówkowe:
   <sys/types.h>, <sys/stat.h>
2. Tworzenie potoku: funkcja mkfifo(). Otwarcie łącza, zapis i odczyt danych.

pliki nagłówkowe	<unistd.h>		
prototyp	int mkfifo(const char *filename, mode_t mode);		
zwracana wartość	sukces	porażka	zmiana errno
	0	-1	tak

filename - nazwa ścieżkowa; mode - prawa dostępu;
Otwieranie i zamykanie potoku FIFO funkcje: open(), close() bez flagi O_RDWR.
Dodatkowe informacje:
https://man7.org/linux/man-pages/man1/mkfifo.1.html
https://man7.org/linux/man-pages/man3/mkfifo.3.html

Przy zapisie do łącza lub kolejki FIFO bufora danych za pomocą niskopoziomowych funkcji I/O można posłużyć się następującym kodem:

```
int fd=open (fifo_path, O_WRONLY);
write (fd,data, data_lenght);
close(fd);
```

Aby odczytać tekst z kolejki FIFO za pomocą funkcji I/O biblioteki C, można użyć następującego kodu:

```
FILE* fifo=fopen (fifo_path, ”r”);
fscanf(fifo, ”%s”, bufor);
close(fifo);
```

Jeżeli należy zrealizować komunikację dwukierunkową między programami, można użyć pary FIFO lub jawnie zmienić kierunek przepływu danych poprzez zamknięcie i ponowne otwarcie FIFO.
Wiele procesów może pisać do kolejki FIFO lub z niej czytać. Bajty od każdego procesu są niepodzielnie zapisywane do maksymalnej wielkości PIPE_BUF = 4KB w systemie Linux.
Zapis do FIFO, które nie może przyjąć wszystkich bajtów może zakończyć się w dwojaki sposób:

- Spowodować błąd jeśli zażądano zapisu PIPE_BUF (limits.h - 4096 bajtów) lub mniejszej liczby bajtów, a dane nie mogą zostać przyjęte;
- Zapisać część danych, jeśli zażądano zapisu więcej niż PIPE_BUF bajtów, zwracając liczbę faktycznie zapisanych danych (może być równa 0)

System gwarantuje, że zapis PIPE_BUF lub mniejszej ilości bajtów do FIFO otwartego w trybie O_WRONLY (blokującego się), zapisze wszystkie bajty albo żadnego.
Jeśli kilka programów próbuje jednocześnie zapisać dane do FIFO istotne jest żeby bloki pochodzące z różnych programów nie uległy przemieszaniu.

Aby to zapewnić należy:

- zadania zapisu kierować do blokującego się FIFO;
- bloki muszą mieć rozmiar mniejszy lub równy PIPE_ BUF system sam zadba o to, żeby dane się nie pomieszały.

Kiedy proces Linux-a jest zablokowany, nie zużywa zasobów procesora - metoda synchronizacji procesów za pomocą blokujących się FIFO jest bardzo wydajna.

Typowe przykłady otwierania kolejki:

`open(const char *path, O_RDONLY);`

- funkcja zablokuje się, dopóki inny proces nie otworzy kolejki do zapisu;

`open(const char *path, O_RDONLY | O_NONBLOCK);`

- proces nie blokuje się

`open(const char *path, O_WRONLY);`

- funkcja zablokuje się, dopóki inny proces nie otworzy tej kolejki do odczytu;

`open(const char *path, O_WRONLY | O_NONBLOCK);`

- nie blokuje się, ale jeśli żaden proces nie otworzył FIFO do odczytu zwraca błąd (-1)

Najczęstsze zastosowanie nazwanych potoków:

proces czytający - O_RDONLY

proces piszący - O_WRONLY | O_NONBLOCK

- Czytający proces uruchamia się, czeka na powrót funkcji open(), a kiedy inny program otworzy FIFO do zapisu oba programy kontynuują działanie,
- Procesy synchronizują się przez wywołanie open(),

3. Zamknięcie łącza.

Aby zamknąć łącze (lub plik) używamy funkcji systemowej

```
int close(int fd);
```

Funkcja zwróci -1 w przypadku błędu.

Dodatkowe informacje:

https://man7.org/linux/man-pages/man2/close.2.html

4. Usunięcie łącza nazwanego.

Aby usunąć łącze (lub plik) używamy funkcji systemowej

```
int unlink(const char *path);
```

Jeżeli z łącza korzystają procesy zostaje usunięta nazwa łącza z dysku (nowe procesy nie mogą z niego korzystać). Łącze zostanie usunięte, gdy wszystkie procesy korzystające z niego zamkną deskryptory z nim związane.

Funkcja zwróci -1 w przypadku błędu.

Dodatkowe informacje:

https://man7.org/linux/man-pages/man2/unlink.2.html

Ćwiczenie 1.

Skopiuj do swojego katalogu domowego pliki fifo.c i fifo_1.c znajdujące się w katalogu:
/home/inf-prac/wojtas.jan/Dydaktyka/SO/Projekty/FIFO

W trybie pracy krokowej przeanalizuj sposób działania funkcji związanych wykorzystaniem łącza nazwanego.

5. Przykład wykorzystania łącza nazwanego do realizacji potoku who | wc -l (plik fifo_1.c).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char* argv[]) {
        int pdesk;
        if (mkfifo("./moje_fifo", 0777) == -1){
                printf("blad\n");
                exit(1);
        }
        switch (fork()){
                case -1:
                        exit(1);
                case 0:
                        fprintf(stderr,"jestem potomny\n");
                        close(1);
                        pdesk=open("./moje_fifo",O_WRONLY);
                        if (pdesk != 1){
                                printf("blad deskryptora do zapisu\n");
                                exit(1);
                        }
                        fprintf(stderr,"robie who\n");
//                      sleep(5);
                        execlp("who","who",NULL);
                        exit(1);
                default:
                        close(0);
                        pdesk=open("./moje_fifo",O_RDONLY);
                        if (pdesk != 0){
                                printf("blad deskryptora do odczytu\n");
                                exit(1);
                        }
//                      sleep(5);
                        unlink("./moje_fifo");
                        printf("robie wc -l\n");
                        execlp("wc","wc","-l",NULL);
                        exit(1);
        }
}
```

Ćwiczenie 2.
Napisz program realizujący potok:
who | cut -d' ' -f1 | nl

Ćwiczenie 3.
Skopiuj do swojego katalogu domowego pliki p.c i k.c znajdujące się w katalogu:
/home/inf-prac/wojtas.jan/Dydaktyka/SO/Projekty/FIFO
W trybie pracy krokowej przeanalizuj sposób działania funkcji związanych wykorzystaniem kolejki fifo w zadaniu producent - konsument.

Projekt nr 5.

Wykorzystując potoki nazwane (fifo) należy zaimplementować zadanie typu klient-serwer z możliwością obsługi wielu klientów jednocześnie.

Projekt składa się z dwóch programów uruchamianych niezależnie: serwera i klienta:

- [s] serwer tworzy swoje FIFO, otwiera je w trybie tylko do odczytu i blokuje się;
- [s] pozostaje w tym stanie do momentu aż połączy się z nim klient, otwierając to samo FIFO do zapisu;
- [s] serwer odblokuje się i wykona funkcję sleep();
- [k] po otwarciu FIFO serwera każdy klient tworzy własne FIFO o unikatowej nazwie, przeznaczone do odczytywania danych zwracanych przez serwer;
- [k] klient przesyła dane do serwera (blokując się jeżeli potok jest pełny albo serwer nadal uśpiony), dane umieszczane np. w strukturze o składowych jak w przykładzie poniżej;
- [k] klient blokuje się na odczycie własnego potoku, oczekując na odpowiedź serwera;
- [s] po otrzymaniu danych od klienta serwer przetwarza je (zamienia wszystkie litery w wiadomości na duże), otwiera FIFO klienta do zapisu (odblokowując w ten sposób klienta);
- [s] zapisuje przetworzone dane;
- [k] po odblokowaniu klient może odczytać ze swojego potoku dane zapisane przez serwer;
- cały proces powtarza się dopóki ostatni klient nie zamknie potoku serwera, wówczas funkcja read() w serwerze zwróci 0;

Przykładowa struktura (do przekazania przez łącze nazwane):

```
struct dane_do_przekazania {
                pid_t pid_klienta;
                char dane[MAX];
        };
```

Fragmenty programu klienta:

```
struct dane_do_przekazania moje_dane;
.....
fifo_serwera_dp = open(NAZWA_FIFO_SERWERA, O_WRONLY);
write(fifo_serwera_dp, &moje_dane, sizeof(moje_dane));
.....
mkfifo(fifo_klienta, 0777)
fifo_klienta_dp = open (fifo_klienta, O_RDONLY);
read (fifo_klienta_dp, &moje_dane, sizeof(moje_dane))
```

Fragmenty programu serwera:

```
struct dane_do_przekazania moje_dane;
.....
mkfifo(NAZWA_FIFO_SERWERA, 0777);
fifo_serwera_fd = open(NAZWA_FIFO_SERWERA, O_RDONLY);
.....
odczyt_res = read(fifo_serwera_fd, &moje_dane, sizeof(moje_dane));
.....
fifo_klienta_fd = open(fifo_klienta, O_WRONLY);
write(fifo_klienta_fd, &moje_dane, sizeof(moje_dane));
```

*Treści oznaczone kursywą pochodzą z różnych źródeł internetowych.