Systemy operacyjne – studia stacjonarne 2022/23

Lab 11. Łącza komunikacyjne: potoki nienazwane, przykłady przetwarzania potokowego. Projekt nr 4.

1. Potoki nienazwane.

   Potok (pipe) można uznać za plik specjalnego typu, który służy do przechowywania ograniczonej ilości danych i do którego dostęp można uzyskać jedynie w trybie FIFO. Maksymalna liczba bajtów, którą można zapisać w potoku jest określona stałą PIPE_BUF, której deklaracja znajduje się w pliku nagłówkowym <limits.h> lub <sys/param.h> (maksymalną liczbę bajtów można odczytać przez funkcję fpathconf(file_descriptor[0], _PC_PIPE_BUF)).

   Potoki zapewniają synchroniczny sposób wymiany danych między procesami. Dane zapisywane są na jednym końcu potoku i odczytywane na jego drugim końcu. Odczytane dane są usuwane z potoku. System zapewnia synchronizację między procesem zapisującym i odczytującym. Jeśli proces spróbuje zapisać dane do pełnego potoku zostanie przez system automatycznie zablokowany do czasu, gdy potok będzie w stanie je odebrać. Podobnie proces odczytujący, który podejmie próbę pobrania danych z pustego potoku zostanie zablokowany do czasu, kiedy pojawią się jakieś dane. Do zablokowania dojdzie również wtedy, gdy potok zostanie otwarty przez jeden proces do odczytu, a nie zostanie otwarty przez inny proces do zapisu. Potoki nienazwane mogą łączyć tylko procesy pokrewne np.: macierzysty i potomny, dwa potomnie itd. Do ich tworzenia służy funkcja systemowa pipe(). Funkcja tworzy parę sprzężonych deskryptorów pliku, wskazujących na inode potoku i umieszcza je w tablicy file_descriptor[2].

   Do wywołania funkcji niezbędne są następujące pliki nagłówkowe:

   <unistd.h>

2. Tworzenie potoku: funkcja pipe(). Zapis i odczyt danych.

pliki nagłówkowe	<unistd.h>		
Prototyp	int pipe(int file_descriptor[2]);		
zwracana wartość	sukces	porażka	zmiana errno
	0	-1	tak

   file_descriptor - tablica z deskryptorami plików;

   Funkcja wypełnia tablicę file_descriptor[] dwoma deskryptorami plików. Wszystkie dane zapisane do file_descriptor[1] mogą być odczytane z file_descriptor[0] (wg FIFO).

   Zapis i odczyt funkcje: write(), read().

   ```
   ssize_t write(int fd, const void *buf, size_t count);
   ```

   write() zapisuje maksymalnie count bajtów do pliku wskazywanego przez deskryptor fd. Zapis następuje z bufora wskazywanego przez buf.

   ```
   ssize_t read(int fd, void *buf, size_t count);
   ```

   read() próbuje odczytać maksymalnie count bajtów z deskryptora plików fd do bufora, którego początek znajduje się w buf.

   Gdy zostanie zamknięty deskryptor do 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)

   Gdy zostanie zamknięty deskryptor do 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.

Łącza zapewniają komunikację w jednym kierunku. Kiedy potrzebujemy przepływu w obu kierunkach musimy utworzyć dwa łącza i korzystać z jednego dla każdego kierunku.

3. Zamknięcie łącza.

Aby zamknąć łącze (lub plik) używamy funkcji systemowej

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

Funkcja przekaże -1 w przypadku błędu.

O zamykaniu otwartych, a niepotrzebnych już deskryptorów często się zapomina. Jednak jest to bardzo ważna operacja i to z dwóch powodów:

- *Zamykając zbędne deskryptory natychmiast, gdy przestają być potrzebne zapobiegamy zbędnemu kopiowaniu deskryptorów (przy fork()) i nadmiernemu wzrostowi tablic deskryptorów.*
- *Zamknięcie wszystkich deskryptorów służących do zapisu do danego łącza jest często jedynym sposobem poinformowania procesu czytającego z tego łącza, że nie ma już więcej danych do odczytu w łączu.*

Nie zamykamy sami deskryptorów 0, 1, 2, chyba że jest do tego istotny powód. Zawsze zamykamy to, co sami otworzyliśmy, choć tak naprawdę proces wykonujący funkcję systemową exit() zamyka sam wszystkie otwarte łącza.

Ćwiczenie 1.

Skopiuj do swojego katalogu domowego pliki pipe.c oraz pipe_1.c znajdujący się w katalogu:
/home/inf-prac/wojtas.jan/Dydaktyka/SO/Projekty/PIPE
W trybie pracy krokowej przeanalizuj sposób działania funkcji związanych wykorzystaniem łącza nienazwanego.

4. Powielenie deskryptora pliku.

Często chcemy utworzyć proces potomny i ustawić jeden z końców potoku jako standardowe wejście lub wyjście. Funkcje `dup()` i `dup2()` służą do powielania istniejących deskryptorów pliku

```
#include <unistd.h>
int dup(int deskryptor1);
int dup2(int deskryptor1, deskryptor2);
```

- Funkcja `dup`: tworzy (i zwraca) nowy deskryptor pliku, gwarantując, że będzie on miał najmniejszą wartość spośród wszystkich wolnych wartości numerów dla deskryptorów. Argumentem funkcji jest istniejący już deskryptor, który chcemy powielić.
- Funkcja `dup2`: tworzy nowy deskryptor, którego wartość jest równa deskryptorowi2. Jeżeli deskryptor2 jest już otwarty, zostanie zamknięty przed wykonaniem kopiowania. Gdy deskryptory 1 i 2 są sobie równe, wtedy zostaje zwrócona wartość deskryptora2 bez uprzedniego zamknięcia (dlatego zawsze trzeba sprawdzić, jakie są deskryptory).

Przykład wykorzystania funkcji dup2() do realizacji potoku ls -la | wc -l (plik pipe_2.c).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
         int potok[2];
        if (pipe(potok) == -1){
                perror("pipe error");
```

```
                exit(1);
        }

        switch(fork()){
          case -1:
                perror("fork error");
                exit(1);
          case 0:
                close(potok[0]);
                dup2(potok[1], 1);
                execlp("ls","ls","-la",NULL);
                perror("execlp error");
                exit(2);
          default: {
                close(potok[1]);
                dup2(potok[0], 0);
                execlp("wc", "wc","-l",NULL);
                perror("execlp error");
                exit(2);
         }
        }
}
```

Dodatkowe informacje:
http://www.linux.pl/man/index.php?section=&command=dup

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

5. Tworzenie łącza komunikacyjnego przy użyciu funkcji popen().

Najczęściej tworzymy łącze komunikacyjne powiązane z innym procesem, aby czytać dane wyjściowe albo przesyłać dane wejściowe. Ułatwiają to funkcje popen() oraz pclose(), które eliminują konieczność wywoływania funkcji pipe(), fork(), dup2() oraz exec().

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

Funkcja popen() - zwraca wskaźnik (uchwyt) pliku (FILE*, odpowiadający otwartemu plikowi), gdy wszystko jest w porządku, w przypadku błędu zwraca NULL. Wywołuje funkcję fork(), a następnie exec(), która wykonuje przez powłokę polecenie cmdstring . Argument type może przyjmować dwie wartości "w" (uchwyt związany ze standardowym wejściem) oraz "r" (uchwyt związany ze standardowym wyjściem).
Funkcja pclose() - przekazuje stan zakończenia polecenia cmdstring, gdy wszystko w porządku, w razie błędu zwraca wartość -1. Zamyka standardowy strumień I/O.
Zamieszczony poniżej program (plik pipe_3.c) tworzy dwa potoki do komunikacji z procesami potomnymi: fp_in to koniec potoku do odczytu danych wygenerowanych przez proces who, fp_out to koniec potoku do zapisu danych przez proces grep . Dane wynikowe procesu who trafiają (za pośrednictwem procesu macierzystego) do procesu grep i są wypisywane na standardowe wyjście.

```
#include <stdio.h>
int main(){
   FILE *fp_in, *fp_out;
   char buf[256];

   fp_in=popen("who","r"); // zwraca koniec do odczytu - stdout
   fp_out=popen("grep wojtas", "w"); // zwraca koniec do zapisu - stdin
   while (fgets(buf,size,fp_in))
          fputs(buf,fp_out);
   pclose(fp_in);
   pclose(fp_out);
   return 0;
}
```

Dodatkowe informacje:
http://www.linux.pl/man/index.php?section=&command=popen

Projekt nr 4.

Wykorzystując potoki nienazwane uogólnić zadanie producent–konsument na wielu producentów i wielu konsumentów. W zadaniu wykorzystać jedno łącze nienazwane.
Argumentem wywołania programu jest liczba konsumentów, liczba producentów oraz liczba znaków „wyprodukowanych" przez każdego producenta (np. ./prog 10 50 1000). Program sprawdza dopuszczalny limit procesów, które może użytkownik uruchomić w danym momencie (np. przy pomocy funkcji popen()) i w przypadku, kiedy limit jest większy lub równy liczbie tworzonych procesów, uruchamia zadanie.
Każdy z producentów losuje określoną trzecim argumentem wywołania liczbę znaków. Wylosowane znaki zapisuje – po jednym znaku - do potoku oraz swojego pliku z danymi (np. we_1250.txt, gdzie podany numer to pid procesu, który utworzył dany plik). Każdy z konsumentów odczytuje po jednym znaku z potoku i zapisuje w swoim pliku z wynikami (np.: wy_1320.txt). Po zakończeniu zadana liczba znaków we wszystkich plikach we_*.txt musi się równać liczbie znaków we wszystkich plikach wy_*.txt.
Dla funkcji systemowych zaprogramować obsługę błędów w oparciu o funkcję perror() i zmienną errno.

*Treści oznaczone kursywą pochodzą z różnych źródeł internetowych.