Komunikacja międzyprocesowa - łącza nienazwane		
Dominik Wróbel	11 IV 2019	Czw. 17:00

Spis treści

1 Tworzenia prostych łączy jednokierunkowych

Zadanie 1.1 1.2 1.3

Zmodyfikuj kod programu:

- Podziel program na 2 procesy (użyj fork()).
- Korzystając z funkcji close zamknij zbędne łącza (WAŻNE: które?/dlaczego?).
- Przekaż komunikat pomiędzy procesami.

Pytanie 1.3

Korzystając z funkcji close zamknij zbędne łącza (WAŻNE: które?/dlaczego?).

Po rozdziale programu na dwa procesy musimy zamknąć deskryptory:

- Odpowiadający operacji czytania w procesie, który zapisuje dane, ponieważ powinien istnieć tylko jeden deskryptor do czytania we wszystkich procesach
- Odpowiadający operacji zapisywania w procesie, który czyta dane, ponieważ jeden proces nie powinien mieć możliwości czytania i zapisywania jednocześnie

Listing 1: simple_pipe.c

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

int main(int argc, char *argv[]){

    int pfd[2]; // file descriptors
    size_t nread;
    char buf[100];
    pid_t newProcessPID;

    pipe(pfd); // create pipe

    char * str ="Ur beautiful pipe example";

    newProcessPID = fork(); // create child process

    if(newProcessPID == 0){ // child process
        close(pfd[1]); // close writing descriptor
        nread=read(pfd[0],buf,sizeof(buf));
        (nread!=0)?printf("%s (%ld bytes)\n",buf,(long)nread):printf("No data\n");
        _exit(EXIT_SUCCESS);
    } else{ // parent process
        close(pfd[0]); // close reading descriptor
        write(pfd[1], str, strlen(str));
        _exit(EXIT_SUCCESS);
    }

    return 0;
}
```

Zadanie 1.4

Korzystając z funkcji fpathconf sprawdź wielkość bufora dla łączy komunikacyjnych na Twoim systemie. Odp. na pytanie: do czego potrzebna jest ta informacja?

Listing 2: zadanie1_4.c

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

int main(){

    int pfd[2];
    long v;

    pipe(pfd);

    v = fpathconf(pfd[0], _PC_PIPE_BUF);
    printf("PIPE_BUF = %ld\n", v);

}

// WYNIK DZIALANIA:
// dominik@dominik-VirtualBox:~/OperatingSystems/Lab6$ ./check_buf
// PIPE_BUF = 4096
```

Pytanie 1.4

Korzystając z funkcji fpathconf sprawdź wielkość bufora dla łączy komunikacyjnych na Twoim systemie. Odp. na pytanie: do czego potrzebna jest ta informacja?

Jest to stała definiująca ile danych można zapisać do łącza w operacji atomicznej. Wartość ta jest nie mniejsza niż 512, oznacza to, że przy zapisywaniu zawsze 512 bajtów danych nie będzie przerwane przez inne dane. Jest to ważne z punktu widzenia działania funkcji read, write na łączach, ponieważ zbyt duża ilość danych może doprowadzić do zablokowania funkcji write.

2 Praca z łączami komunikacyjnymi

Zadanie 2

- Funkcja pipe w przypadku błędu ustawia zmienną errno. Zaimplementuj obsługę błędów wyświetlającą informację dla użytkownika dot. błędu.
- Do powyższego programu dodaj niezbędne funkcje close zamykające w odpowiednich miejscach otwarte deskryptory plików, a zamiast funkcji dup użyj dup2.

Pytanie 2

Do czego w procesie potomnym użyta jest funkcja close(0).

W tym przypadku 0 odnosi się do deskryptora standardowego wejścia. Deskryptor ten jest zamykany i jest dzięki temu zwalniany. Wywołanie close dla deskryptorów łączy zapisujących powoduje, że read otrzymuje znak EOF, a w wypadku zamknięcia deskryptora czytającego zwracany jest błąd dla deskryptorów zapisujących.

Listing 3: dup-example.c

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

int main(void)
{
    int     pfd[2];
    pid_t   pid;
    char    string[] = "Test";
    char    buf[10];

    if(pipe(pfd) == -1){ // 1. error handling
        perror("Pipe call failed.");
        exit(EXIT_FAILURE);
    }

    pid = fork();

    if(pid == 0) {
        close(0);
        // dup(pfd[0]);
        dup2(pfd[0],0);
        read(STDIN_FILENO, buf, sizeof(buf));
        close(pfd[0]);
        printf("Wypisuje: %s", buf);
    } else {
        close(pfd[0]);
        write(pfd[1], string, (strlen(string)+1));
        close(pfd[1]);
    }

    return 0;
}
```

3 Komunikacja z kilkoma procesami

3.1. Komunikacja z kilkoma procesami

Zadanie 3.1

Napisz program, który utworzy proces czytający dane z pliku (słownika), a następnie utworzy dwa podprocesy, z których jeden policzy liczbę linii (liczbę słów w słowniku), a drugi policzy liczbę linii, zawierających podciąg pipe (liczbę słów, które zawierają słowo pipe).

Komentarze opisujące wykonane w programie działania zamieszczono na listingu.

Listing 4: Komunikacja z kilkoma procesami.

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>

#define BUFFSIZE 4096

int main()
{
    // ------------------------------------------------
    // parent --> child1
    // communication
    int pfd1[2];

    // child1 --> parent
    // communication
    int pfd2[2];

    pipe(pfd1);
    pipe(pfd2);
    // ------------------------------------------------

    // ------------------------------------------------
    // parent --> child2
    // communication
    int ppfd1[2];

    // child2 --> parent
    // communication
    int ppfd2[2];

    pipe(ppfd1);
    pipe(ppfd2);
    // ------------------------------------------------

    int childOne;
    int childTwo;

    char childOneReceived[BUFFSIZE];
    char parentSent[BUFFSIZE];
    char childOneToParent[BUFFSIZE];
    char parentFromChildOneReceived[BUFFSIZE];

    char childTwoReceived[BUFFSIZE];
    char childTwoToParent[BUFFSIZE];
    char parentFromChildTwoReceived[BUFFSIZE];
```

```
48
49      const char * lookForThisWord = "pipe";
50
51      int linesNumber = 0;
52
53      int pipesWordNumber = 0;
54
55      // open file
56      int fdDictionary = open("dictionary.txt", O_RDONLY);
57      if(fdDictionary == -1){
58          printf("Error opening dictionary.txt");
59          exit(EXIT_FAILURE);
60      }
61
62      // printf("BreakPoint1");
63
64      childOne = fork();
65
66      if(childOne == 0) // from parent to childOne
67      {
68          int j = 0;
69
70          close(pfd1[1]);
71          close(pfd2[0]);
72
73          // parent --> childOne
74          // receive and count lines
75          while(read(pfd1[0], childOneReceived, BUFFSIZE) != 0) {
76              for(j=0; j < BUFFSIZE; j++){
77                  if(childOneReceived[j]=='\n'){
78                      linesNumber++;
79                  }
80              }
81          }
82
83          // childOne --> parent
84          // send number of counted lines
85          printf("There are %d lines", linesNumber);
86
87          sprintf(childOneToParent, "%d", linesNumber);
88
89          write(pfd2[1],childOneToParent,BUFFSIZE);
90          close(pfd2[1]);
91
92          // printf("\nchild recieved is %s and lines are %d", childOneReceived,
    linesNumber);
93
94          exit(EXIT_SUCCESS);
95      }
96
97      childTwo = fork();
98
99      // printf("Child two is %d", childTwo);
100
101     if(childTwo == 0) { // from parent to childTwo
102
103         printf("BreakPoint2.1");
104
105         close(ppfd1[1]);
106         close(ppfd2[0]);
107
108         char * s;
109         const char * start = NULL;
```

```
110
111         // parent --> childTwo
112         // receive and count lines
113         while(read(ppfd1[0], childTwoReceived, BUFFSIZE) != 0) {
114                 while ( (s = strstr(start, lookForThisWord)) != NULL ) {
115                     pipesWordNumber++;
116                 }
117         }
118
119         // childTwo --> parent
120         // send number of counted lines
121         printf("Lines are %d", pipesWordNumber);
122
123         sprintf(childTwoToParent, "%d", pipesWordNumber);
124
125         write(ppfd2[1], childTwoToParent, BUFFSIZE);
126         close(ppfd2[1]);
127
128         exit(EXIT_SUCCESS);
129
130         // printf("\nchild recieved is %s and lines are %d", childOneReceived,
    linesNumber);
131     }
132
133
134     if(childOne > 0 && childTwo > 0) { // parent
135
136         close(pfd1[0]);
137         close(pfd2[1]);
138         close(ppfd1[0]);
139         close(ppfd2[1]);
140
141     // printf("BreakPoint4");
142     // send from parent to children
143         while(read(fdDictionary, parentSent, BUFFSIZE) != 0){
144             // printf("BreakPoint5");
145             write(pfd1[1], parentSent, BUFFSIZE);
146             write(ppfd1[1], parentSent, BUFFSIZE);
147             // printf("BreakPoint7");
148         }
149         close(pfd1[1]);
150         close(ppfd1[1]);
151
152     // printf("BreakPoint10");
153     // receive from children
154         read(pfd2[0], parentFromChildOneReceived, BUFFSIZE);
155
156         printf("\nFrom childOne recieved is %s", parentFromChildOneReceived);
157
158         read(ppfd2[0], parentFromChildTwoReceived, BUFFSIZE);
159
160         printf("\nFrom childTwo recieved is %s", parentFromChildTwoReceived);
161
162         exit(EXIT_SUCCESS);
163
164     }
165
166
167 }
```

3.2. Przekazywanie danych przez kilka łączy

Zadanie 3.2

Proszę napisać program, który utworzy dwa procesy potomne p1 i p2. Proces potomny p1 generuje kolejne liczby od 1 do 10 (można użyć seq). Proces p1 powinien za pomocą łącza nienazwanego przekazać je do procesu p2, który pomnoży każdą z liczb razy 5. Następnie proces p2 przekaże liczby do procesu macierzystego p0, który wyświetli otrzymane liczby. .

W kodzie proces childOne odpowiedzialny jest za generację liczb w pętli od 1 do 10. Liczby są wysyłane 'po jednej' do procesu childTwo, który odbiera dane w postaci string, konwertuje je na int i mnoży przez 5, a następnie wysyła do procesu rodzica w postaci string, rodzic wyświetla otrzymane dane. Program działa poprawnie, ponieważ funkcję read czekają na zamknięcie deskryptorów plików zapisujących oraz na dane do odczytania, co umożliwia poprawną kolejność przepływu danych.

Listing 5: Przekazywanie danych przez kilka łączy.

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>

#define BUFFSIZE 4096

int main()
{

    // ------------------------------------------------
    // child1 --> child2
    int ch1ch2Pipe[2];

    // child2 --> parent
    int ch2pPipe[2];

    pipe(ch1ch2Pipe);
    pipe(ch2pPipe);
    // ------------------------------------------------

    int childOne;
    int childTwo;

    char childOneToChildTwo[BUFFSIZE];
    char childTwoReceived[BUFFSIZE];

    // char childTwoToParent[BUFFSIZE];
    char parentReceived[BUFFSIZE];

    childOne = fork();

    if(childOne == 0){ // child one

        printf("Inside child one ! \n");

        int i = 0;
        close(ch1ch2Pipe[0]);

        for(i=0; i<10; i++){
            sprintf(childOneToChildTwo, "%d", (i+1));
            write(ch1ch2Pipe[1],childOneToChildTwo,BUFFSIZE);
        }
```

```
45
46          close(ch1ch2Pipe[0]);
47          exit(EXIT_SUCCESS);
48      }
49
50
51      childTwo = fork();
52
53      if(childTwo == 0){ // child two
54
55          printf("Inside child two ! \n");
56          int numReceived = 0;
57
58          close(ch1ch2Pipe[1]);
59          close(ch2pPipe[0]);
60
61          while(read(ch1ch2Pipe[0], childTwoReceived, BUFFSIZE) != 0) {
62              printf("Child two received %s \n", childTwoReceived);
63              numReceived = (atoi(childTwoReceived) * 5);
64              sprintf(childTwoReceived, "%d", numReceived);
65              write(ch2pPipe[1], childTwoReceived, BUFFSIZE);
66          }
67
68          close(ch2pPipe[1]);
69
70          exit(EXIT_SUCCESS);
71      }
72      else { // parent
73         printf("I am inside parent ! \n");
74
75         close(ch2pPipe[1]);
76
77          while(read(ch2pPipe[0], parentReceived, BUFFSIZE) != 0) {
78              printf("Parent Received %s \n", parentReceived);
79          }
80
81          exit(EXIT_SUCCESS);
82      }
83
84      return 0;
85
86  }
```