Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу
«Операционные системы»

Студент: Армишев Кирилл Константинович
Группа: М8О–208Б–21
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

Постановка задачи

Задание

Необходимо написать 3-и программы. Далее будем обозначать эти программы A, B, C. Программа A принимает из стандартного потока ввода строки, а далее их отправляет программе C. Отправка строк должна производится построчно. Программа C печатает в стандартный вывод, полученную строку от программы A. После получения программа C отправляет программе A сообщение о том, что строка получена. До тех пор пока программа A не примет «сообщение о получение строки» от программы C, она не может отправлять следующую строку программе C. Программа B пишет в стандартный вывод количество отправленных символов программой A и количество принятых символов программой C. Данную информацию программа B получает от программ A и C соответственно. Способ организация межпроцессорного взаимодействия выбирает студент.

Общие сведения о программе

Программа состоит из трех файлов: a.c, b.c, c.c, используются заголовочные файлы: stdio.h, stdlib.h, unistd.h, sys/wait.h, fcntl.h, string.h. В ходе работы были применены следующие системные вызовы:

1. **write () -** переписывает count байт из буффера в файл. Возвращает количество записанных байт или -1;
2. **read () -** считывает count байт из файла в буффер. Возвращает количество считаных байт (оно может быть меньше count) или -1;
3. **pipe () -** создаёт канал между двумя процессами. Создаёт и помещает в массив 2 файловых дескриптора для чтения и для записи. Возвращает 0 или -1;
4. **close() -** закрывает файловый дескриптор, который больше не ссылается ни на один файл, возвращает 0 или -1;

5. **fork ()** - порождается процесс-потомок. Весь код после fork() выполняется дважды, как в процессе-потомке, так и в процессе-родителе. Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родители возвращает идентификатор pid потомка или -1. Процесс-потомок возвращает 0 или -1;
6. **execv ()** - заменяет текущий образ процесса новым образом процесса. Новая программа наследует от вызывавшего процесса идентификатор и открытые файловые дескрипторы;

Общий метод и алгоритм решения

С помощью двух вызовов fork(), создаются два дочерних процесса B и C. A получает от пользователя строку, длинна которой ограничивается 100 символами, и передает ее в C через pipe. C получает строку и выводит ее в стандартный поток вывода, после этого отправляет программе A через канал длину строки (т.к. по условию, пока программа A не примет «сообщение о получение строки» от программы C, она не может отправлять следующую строку программе C). A и C отправляют через каналы в программу B размеры строк и B выводит их. Программа прекращает свою работу в случае введения пустой строки.

Основные файлы программы

========== a.c ==========

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int A_C[2];
    if (pipe(A_C)==-1){
        perror("error pipe1");
    }
    int A_B[2];
    if (pipe(A_B)==-1){
        perror("error pipe2");
    }
    int C_B[2];
    if (pipe(C_B)==-1){
        perror("error pipe3");
    }
```

```
    int C_A[2];
    if (pipe(C_A)==-1){
        perror("error pipe3");
    }
    int id = fork();
    if (id == -1)
    {
        perror("fork error");
        return 0;
    }else if(id == 0) {
        close(A_C[1]);  close(A_C[0]);  close(C_B[1]);  close(A_B[1]);
close(C_A[1]);  close(C_A[0]);
        char A_read[10];
        char C_read[10];
        sprintf(A_read, "%d", A_B[0]);
        sprintf(C_read, "%d", C_B[0]);
        char *B_argv[] = {"b", A_read, C_read, NULL};
        execv("b", B_argv);
    }
    int id2;
    if(id > 0) {
        id2 = fork();
        if (id2 == -1)
        {
            perror("fork2 error");
            return 0;
        }else if(id2 == 0) {
            close(A_C[1]); close(C_B[0]); close(A_B[1]); close(C_A[0]);
close(A_B[0]);
            char A_read[10];
            char A_write[10];
            char B_write[10];
            sprintf(A_read, "%d", A_C[0]);
            sprintf(A_write, "%d", C_A[1]);
            sprintf(B_write, "%d", C_B[1]);

            char *C_argv[] = {"c", A_read, A_write, B_write, NULL};

            execv("c", C_argv);
        }
    }
    if(id != 0 && id2 != 0) {
        close(A_C[0]); close(A_B[0]); close(C_B[0]); close(C_B[1]);
close(C_A[1]);
        char string[100];
        while (1) {
            if(gets(string) == NULL) {
                perror("gets error");
                return 0;
            }
            int length = strlen(string);

            if(length == 0) {
                if(write(A_C[1], &length, sizeof(length)) == -1) {
                    perror("Write in A_C error");
                    return 0;
                }

                if(write(A_B[1], &length, sizeof(length)) == -1) {
                    perror("Write in A_B error");
                    return 0;
                }
                break;
            }
```

```
            if(write(A_C[1], &length, sizeof(length)) == -1) {
                perror("Write in A_C error");
                return 0;
            }
            if(write(A_C[1], &string, sizeof(char) * length) == -1) {
                perror("Write in A_C error");
                return 0;
            }

            if(write(A_B[1], &length, sizeof(length)) == -1) {
                perror("Write in A_B error");
                return 0;
            }

            int check = 0;
            if(read(C_A[0], &check, sizeof(check)) == -1) {
                perror("Read from C_A error");
                return 1;
            }
        }
    }
    return 0;
}
```

========== b.c ==========

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



int main(int argc, const char *argv[]) {
    int length;
    int file_descriptors[2];
    file_descriptors[0] = atoi(argv[1]); //read from A
    file_descriptors[1] = atoi(argv[2]); //read from C

    while (1){
        if (read(file_descriptors[0], &length, sizeof(int)) == -1) {
            perror("Read from A error");
        }

        if(length == 0) {
            break;
        }

        printf("from A = %d\n", length);

        if(read(file_descriptors[1], &length, sizeof(int)) == -1) {
            perror("Read from C error");
            return 1;
        }
        printf("from C = %d\n", length);
    }

    return 0;
}
```

========== c.c ==========

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

```c
int main(int argc, const char *argv[]) {
    int file_descriptors[3];
    file_descriptors[0] = atoi(argv[1]);  //read from A
    file_descriptors[1] = atoi(argv[2]);  //write to A
    file_descriptors[2] = atoi(argv[3]);  //write to B

    int length;
    char string[100];
    memset(string, 0, 100);

    while (1) {
        if(read(file_descriptors[0], &length, sizeof(int)) == -1) {
            perror("Read from A error");
        }

        if(length == 0) {
            break;
        }

        if(read(file_descriptors[0], &string, sizeof(char) * length) == -1) {
            perror("Read from A error");
        }

        printf("C: %s\n", string);

        if(write(file_descriptors[1], &length, sizeof(int)) == -1) {
            perror("Write to A error");
        }

        if(write(file_descriptors[2], &length, sizeof(int)) == -1) {
            perror("Write to B error");
        }
        memset(string, 0, 100);
        length = 0;
    }
    return 0;
}
```

Пример работы

```
kirillarmishev@2 untitled20 % ./a.out
warning: this program uses gets(), which is unsafe.
cdsoc
from A = 5
C: cdsoc
from C = 5
eoirgjowrphito
from A = 14
C: eoirgjowrphito
from C = 14
```

Вывод

В ходе данного курсового проекта я попрактиковался в управлении процессами при помощи системных вызовов и обеспечении обмена данными между процессами при помощи неименованных каналов.

Системные вызовы необходимы для управления процессами, файлами и каталогами, а также каналами ввода и вывода данных. Одним из способов создания дочернего процесса является системный вызов fork(), он создает точную копию исходного процесса, включая все дескрипторы файлов, регистры и т. п. После выполнения вызова fork() исходный процесс и его копия (родительский и дочерний процессы) выполняются независимо друг от друга. Благодаря систему вызову pipe можно создать канал между двумя процессами, в которой один процесс сможем писать поток байтов, а другой процесс сможет его читать, так мы переопределяем потоки ввода-вывода.

Благодаря системным вызовам можно упростить программу или выполнить действия, запрещенные в пользовательском режиме.

При написании курсового проекта я закрепил свои знания и навыки, полученные во время прохождения курса операционных систем.