Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу
«Операционные системы»

Взаимодействие между процессами

Студент: Хлебникова Елизавета Сергеевна
Группа: М8О-209Б-23
Вариант: 20

Преподаватель:

Оценка:_____
Дата:_____
Подпись:_____

Москва, 2023

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

Управление процессами в ОС

Обеспечение обмена данных между процессами посредством каналов

Задание

Составить и отладить программу на языке C/C++, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решения задачи несколько дочерних процессов.

Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (pipe). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Общие сведения о программе

Программа компилируется при помощи утилиты CMake и запускается путем запуска ./parent. Также используется заголовочные файлы: iostream, string, stdio.h, unistd.h, sys/wait.h, fcntl.h, vector. В программе используются следующие системные вызовы:

1. **read** – функция read() считывает count байт из файла, описываемого аргументом fd, в буфер, на который указывает аргумент buf Указателю положения в файле дается приращение на количество считанных байт. Если файл открыт в текстовом режиме, то может иметь место транслирование символов.
2. **write** – функция переписывает count байт из буфера, на который указывает bufy в файл, соответствующий дескриптору файла handle. Указателю положения в файле дается приращение на количество записанных байт. Если файл открыт в текстовом режиме, то символы перевода строки автоматически дополняются символами возврата каретки.
3. **pipe** – создаёт механизм ввода вывода, который называется конвейером. Возвращаемый файловый дескриптор можно использовать для операций

чтения и записи. Когда в конвейер что-то записывается, то буферизуется до 504 байтов данных, после чего процесс записи приостанавливается.

4. **fork** - вызов создаёт новый процесс посредством копирования вызывающего процесса. Новый процесс считается дочерним процессом. Вызывающий процесс считается родительским процессом.
5. **close** - закрывает файловый дескриптор, который после этого не ссылается ни на один и файл и может быть использован повторно. Все блокировки, находящиеся на соответствующем файле, снимаются (независимо от того, был ли использован для установки блокировки именно этот файловый дескриптор).
6. **dup2** - системная функция используется для создания копии существующего файлового дескриптора.

Общий метод и алгоритм решения

1) Изучить принципы работы fork, pipe, read, write, close, exec*, dup2.
2) Написать две программмы для родительского и дочерних процессов.
3) Использовать в parent.cpp fork, чтобы запустить дочерние процессы.
4) При помощи конструкции if/else организовать работу с дочерними и родительским процессом.
5) В дочерних процессах скопировать файловые дескрипторы пайпов в stdin и stdout и запустить child.c при помощи execl.
6) Скомпилировать обе программы при помощи CMake и запустить ./parent

Основные файлы программы

parent.cpp

```
#include <iostream>
#include <unistd.h>
#include <string.h>
#include <string>
#include <sys/wait.h>
#include <fcntl.h>
#include <vector>

int main() {
    std::string file_name1;
    std::string file_name2;
    char c = 1;
```

```
write(STDOUT_FILENO, "Enter file name 1: ", 20);
    while (c != '\n') {
        read(STDIN_FILENO, &c, sizeof(char));
        if (c != '\n') {
            file_name1 += c;
        }
    }
    c = 1;

    write(STDOUT_FILENO, "Enter file name 2: ", 20);
    while (c != '\n') {
        read(STDIN_FILENO, &c, sizeof(char));
        if (c != '\n') {
            file_name2 += c;
        }
    }


    int fd1[2], fd2[2];
    int tmp = pipe(fd1);
    if (tmp == -1) {
        write(STDERR_FILENO, "An error occured with creating pipe1", 37);
        return 1;
    }
    tmp = pipe(fd2);
    if (tmp == -1) {
        write(STDERR_FILENO, "An error occured with creating pipe2", 37);
        return 1;
    }


    pid_t process_id1 = fork();
    pid_t process_id2 = fork();
    if (process_id1 == -1) {
        write(STDERR_FILENO, "An error occured with creating child process 1", 47);
        return 1;
    }
    if (process_id2 == -1) {
        write(STDERR_FILENO, "An error occured with creating child process 2", 47);
        return 1;
    }
    if (process_id1 == 0 && process_id2 > 0) {  //child process 1
        close(fd1[1]);
        close(fd2[0]);
        close(fd2[1]);
        tmp = dup2(fd1[0], STDIN_FILENO);
        if (tmp == -1) {
            write(STDERR_FILENO, "An error occured with redirecting input 1", 40);
            return 1;
```

```
}

        tmp = execl("child1", "child1", file_name1.c_str(), NULL);
        if (tmp == -1) {
            write(STDERR_FILENO, "An error occured with runing program from a child process 1", 58);
            return 1;
        }
        exit(EXIT_FAILURE);
    }

    if (process_id1 > 0 && process_id2 == 0) {  //child process 2
        close(fd1[0]);
        close(fd1[1]);
        close(fd2[1]);
        tmp = dup2(fd2[0], STDIN_FILENO);
        if (tmp == -1) {
            write(STDERR_FILENO, "An error occured with redirecting input 2", 40);
            return 1;
        }

        tmp = execl("child2", "child2", file_name2.c_str(), NULL);
        if (tmp == -1) {
            write(STDERR_FILENO, "An error occured with runing program from a child process 2", 58);
            return 1;
        }
        exit(EXIT_FAILURE);
    }

    if (process_id1 > 0 && process_id2 > 0) {   //parent process
        close(fd1[0]);
        close(fd2[0]);
        char c;
        char prev = '?';
        std::vector<char> vec;
        while(read(STDIN_FILENO, &c, 1)) {
            vec.push_back(c);
            if (c == '\n') {
                if (vec.size() > 11){
                    for (int i = 0; i < vec.size(); ++i) {
                        write(fd2[1], &vec[i], 1);
                    }
                } else {
                    for (int i = 0; i < vec.size(); ++i) {
                        write(fd1[1], &vec[i], 1);
                    }
                }
                vec.clear();
            }
            prev = c;
        }
```

```
        close(fd1[1]);
        close(fd2[1]);
        wait(nullptr);
    }
    return 0;
}
```

child.cpp

```
#include <iostream>
#include <unistd.h>
#include <vector>
#include <fcntl.h>
#include <string>

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

    int file = open(argv[1], O_CREAT | O_WRONLY, S_IRWXU);
    if (file == -1) {
        write(STDERR_FILENO, "An error occured with opening file", 35);
        return 1;
    }
    int temp = ftruncate(file, 0);
    if (temp == -1) {
        write(STDERR_FILENO, "An error occured with clearing file", 36);
        return 1;
    }

    int tmp = dup2(file, STDOUT_FILENO);
    if (tmp == -1) {
        write(STDERR_FILENO, "An error occured with redirecting stdout to file", 48);
        close(file);
        return 1;
    }

    std::vector<char> vec;
    char c;
    while (read(STDIN_FILENO, &c, 1)) {
        vec.push_back(c);
        if (c == '\n') {
                for (int i = vec.size() - 2; i >= 0; --i) {
                    write(STDOUT_FILENO, &vec[i], 1);
                }
                write(STDOUT_FILENO, &vec.back(), 1);
                vec.clear();
            }
    }
    close(file);
    return 0;
}
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(OS_Lab1)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsanitize=address")
set(CMAKE_LINKER_FLAGS "${CMAKE_LINKER_FLAGS} -fsanitize=address")

add_executable(parent parent.cpp)
add_executable(child1 child.cpp)
add_executable(child2 child.cpp)
```

Пример работы

Ввод:

```
chava@Deli:~/Desktop/OS_folder/primer/Lab_1/src/build$ ./parent
Enter file name 1: file_name_1
Enter file name 2: file_name_2
1234567890
12345678901
ghh
lllfffllll
l


vf
v
675246284gftwvugbivywugyfuivwygfvsjkbhfeusivbjakbdjvkaegfuvagkjxhbvjkadvbfu
egfvkajv
```

Результатом работы являются два файла, лежащие в одной дирректории с Программой:

file_name_1:

```
0987654321
hhg
lllfffllll
l


fv
v
```

file_name_2:

```
10987654321
Vjakvfgeufbvdakjvbhxjkgavufgeakvjdbkajbvisuefhbkjsvfgywviufyguwyvibguvwtfg482
642576
```

Вывод

В первой лабораторной работе я научилась работать с процессами программ.

Изучив работу системных вызовов, я поняла, что навык работы с процессами позволит в будущем более глубоко понимать устройство программ.
Любая современная функция работы с вводом/выводом в наше время работает на основе read и write. Такие низкоуровневые функции, как exec* используются по сей день в улучшенных оболочках. Управление процессами путем dup2, close и wait поможет в будущем более умело пользоваться многопроцессорными программами.