

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

**Лабораторная работа №1 по курсу
«Операционные системы»**

ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ

Студент: Железнов Илья Васильевич

Группа: М8О–210Б–20

Вариант: 16

Преподаватель: Соколов Андрей Алексеевич

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 Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение задачи один или 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.out. Также используются заголовочные файлы: iostream, string, stdio.h, unistd.h, cstdlib, sys/wait.h, fstream, fcntl.h, sys/stat.h. 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 bufu 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 из файлов 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.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы fork, pipe, read, write, close, exec*, dup2.
2. Написать две программы для родительского и дочернего процесса, а так же написать библиотеку common.h, для работы со стандартными потоками ввода и вывода через read и write.
3. Использовать в parent.c fork, чтобы запустить дочерний процесс.
4. При помощи конструкции if/else организовать работу с дочерним и родительским процессом.
5. В дочернем процессе 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.out.

Основные файлы программы

parent.cpp:

```
#include <iostream>
#include <string>
#include "stdio.h"
#include "unistd.h"
#include <cstdlib>
#include "sys/wait.h"
#include <fstream>
#include <fcntl.h>
#include <sys/stat.h>

#include "src/common.h"

int main() {
    int pipeFD1[2];
    int pipeFD2[2];

    if (pipe(pipeFD1) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
```

```

if (pipe(pipeFD2) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}

int fileFD;
mode_t mode = S_IRWXU;
int flags = O_CREAT | O_WRONLY | O_APPEND;

writeString(STDOUT_FILENO, _USER_ALERT_FILE_INPUT);

if ((fileFD = open(readString(STDIN_FILENO).c_str(), flags, mode)) < 0) {
    writeString(STDOUT_FILENO, _USER_ALERT_ERROR_FILE);
    perror("file");
}

pid_t pid = fork();

if (pid < 0) {
    perror("pid");
    exit(EXIT_FAILURE);
}

if (pid == 0) { // child process
    close(pipeFD2[RD]);
    close(pipeFD1[WR]);

    dup2(pipeFD2[WR], STDOUT_FILENO);
    dup2(pipeFD1[RD], STDIN_FILENO);
    dup2(fileFD, STDOUT_FILENO);
    execl("child.out", "child.out", NULL);
} else { // parent process
    close(pipeFD1[RD]);
    close(pipeFD2[WR]);
    std::string input;

    writeString(STDOUT_FILENO, _USER_ALERT_STRING_INPUT);
    writeString(pipeFD1[WR], readString(STDIN_FILENO));
    writeString(STDOUT_FILENO, readString(pipeFD2[RD]));
    wait(NULL);

    close(pipeFD1[WR]);
    close(pipeFD2[RD]);
}
}

```

child.cpp

```
#include <iostream>
#include <string>
#include "stdio.h"
#include "unistd.h"
#include <cstdlib>
#include "sys/wait.h"
#include <fstream>
#include <fcntl.h>
#include <sys/stat.h>
```

```
#include "src/common.h"
```

```
bool checkPatternStr(std::string string)
```

```
{
    int len = string.length();
    if (string[len - 2] == '.' || string[len - 2] == ';') {
        return true;
    }

    return false;
}
```

```
int main() {
```

```
    std::string stringLine = readString(STDIN_FILENO);
```

```
    if (checkPatternStr(stringLine)) {
        write(STDOUT_FILENO, stringLine.c_str(), stringLine.size() - 1);
        write(STDERR_FILENO, _USER_ALERT_VALID_OUT, sizeof(char) *
_UAVO_SIZE);
    } else {
        write(STDERR_FILENO, _USER_ALERT_INVALID_OUT, sizeof(char) *
_UAIO_SIZE);
    }

    return 0;
}
```

common.h

```
#pragma once
```

```
#include <iostream>
#include <string>
#include "stdio.h"
```

```

#include "unistd.h"
#include <cstdlib>
#include "sys/wait.h"
#include <fstream>
#include <fcntl.h>
#include <sys/stat.h>

#define RD 0
#define WR 1

#define _USER_ALERT_FILE_INPUT "Enter filename to work:\n"
#define _USER_ALERT_ERROR_FILE "\tFile is not opening\n"
#define _USER_ALERT_STRING_INPUT "Enter string to check:\n"
#define _USER_ALERT_INVALID_OUT "\tString isn't valid\n"
#define _UAIO_SIZE 21
#define _USER_ALERT_VALID_OUT "\tString is valid.\n"
#define _UAVO_SIZE 19

std::string readString(int fd);
void writeString(int fd, std::string line);

```

Пример работы

```
keinpop@DESKTOP-T6SLHUS:/mnt/c/oc_lab1/src$ ./parent.out
```

Enter filename to work:

out.txt

Enter string to check:

aaad

String isn't valid

```
keinpop@DESKTOP-T6SLHUS:/mnt/c/oc_lab1/src$ ./parent.out
```

Enter filename to work:

dddd.

Enter string to check:

^C

```
keinpop@DESKTOP-T6SLHUS:/mnt/c/oc_lab1/src$ ./parent.out
```

Enter filename to work:

out.txt

Enter string to check:

asdad.

String is valid.

keinpop@DESKTOP-T6SLHUS:/mnt/c/oc_lab1/src\$./parent.out

Enter filename to work:

out.txt

Enter string to check:

sadsadsd;

String is valid.

keinpop@DESKTOP-T6SLHUS:/mnt/c/oc_lab1/src\$./parent.out

Enter filename to work:

out.txt

Enter string to check:

213123123vvvvv.d

String isn't valid

keinpop@DESKTOP-T6SLHUS:/mnt/c/oc_lab1/src\$./parent.out

Enter filename to work:

out.txt

Enter string to check:

.....;ddd

String isn't valid

Вывод

В первой лабораторной работе я научился работать с процессами программ. Изучив работу каждого системного вызова, путем изучения их мануалов и информации из интернета и разобрав работу стандартных потоков, я понял, что умение и понимание этого позволит в будущем понимать более глубоко устройство программ и их процессов в работе. Любая современная функция работы с вводом/выводом в наше время, работает на основе read и write. А такие низкоуровневые функции, как `exes*` используются по сей день в улучшенных оболочках. Управление процессами путем `dup2`, `close` и `wait` помогут в будущем более умело пользоваться многопроцессорными программами.