Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

Процессы в операционных системах

Студент	Гришин Алексей Юрьевич
Группа	М8О-208Б-21
Вариант	17
Преподаватель	Соколов Андрей Алексеевич
Оценка	5
Дата	03.10.2022
Подпись	

Москва, 2022.

Постановка задачи

Цель работы

Приобретение практических навыков в:

- Управлении процессами в ОС
- Обеспечении обмена данных между процессами посредством каналов

Задание

Составить и отладить программу на языке С, 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 один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы / события / или каналы (`pipe`). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Общие сведения о программе

Программа состоит из двух частей: основной и дочерней.

Основная программа компилируется из файла `manager.c`. Задача основной программы заключается в общении с пользователем, считывании входных значений и делегировании дальнейшей работы одной из двух дочерних программ. Также, основная программа использует заголовочные файлы `string.h`, `stdlib.h`, `unistd.h`, `fcntl.h`, `sys/wait.h`, `sys/types.h`, `sys/stat.h` и вспомогательные модули `io_line` и `pipe`.

Дочерняя программа компилируется из файла `worker.c`. Она запускается основной программой через системный вызов execv. Общение основной программы с дочерними происходят через каналы. Также, дочерняя программа использует заголовочные файлы: `string.h`, `stdlib.h`, `fcntl.h`, `unistd.h`. Для своей работы дочерняя программа использует вспомогательные модули `io_line` и `remove_vowels`.

Модуль `io_line` создан с целью упрощения считывания и печатания строк. В него входят такие методы, как `write_line` и `read_line`, которые реализуют вывод и считывание строки соответственно.

Модуль `pipe` создан с целью абстрагирования от конкретной реализации каналов операционной системой, чтобы в будущем было легче переписать программы под операционную систему Windows. Модуль включает в себя методы `init_pipe`, `set_mode`, `get_input_descriptor`, `get_output_descriptor`. Метод `init_pipe` предназначен для инициализации канала, `set_mode` – для установки режима, в котором будет использоваться канал (получение или отправка сообщений), `get_input_descriptor` – для получения дескриптора входного файла, `get_output_descriptor` – для получения дескриптора выходного файла.

Метод `remove_vowels` решает само задание – удаление из входной строки всех гласных букв. Имеет один метод `remove_vowels`, который и выполняет поставленную задачу.

В программе используются следующие системные вызовы:

1. `execv` – выполняет программу, относительный путь к которой указан в `filename`. Аргументы командной строки передаются вторым аргументом в качестве массива С-строк (указатель `char*`), последний элемент которого обязательно должен быть `NULL`. При успешном выполнении возвращает `0`, в противном случае - `-1`.
2. `dup2` – создает копию дескриптора, переданного в качестве первого аргумента. Дескриптор, который представляет собой копию, передается в качестве второго аргумента.
3. `exit` – немедленно останавливает текущую программу. В качестве аргумента передается статус, с которым завершается программа.
4. `close` – закрывает файл, используя дескриптор, переданный в качестве аргумента
5. `open` – открывает файл. Путь к файлу передается в качестве первого аргумента в виде строки. При успешном выполнении возвращает число, являющееся файловым дескриптором открытого файла, в противном случае возвращает `-1`. Существует 2 версии: с двумя аргументами и с трема. Первая версия ожидает в качестве второго аргумента флаги, с которыми нужно открыть файл (если необходимо использовать множество флагов, то используется побитовое «или». Вторая версия позволяет указать атрибуты доступа к файлу.
6. `read` – предназначен для считывания данных с открытого файла, используя дескриптор, переданный в качестве первого аргумента. Вторым и третьим аргументом должны передаться указатель на область памяти, куда необходимо

занести считанную информацию, и размер этой области (в байтах). Возвращаемое значение – количество считанных байт.

7. `write` – предназначен для записи информации в открытый файл, дескриптор которого указан в качестве первого аргумента. Также, как и в системном вызове read, вторым и третьим аргументами служат указатель на области памяти, которую необходимо записать в файл и ее размер в байтах соответственно. Возвращаемое значение – количество записанных байт.
8. `fork` – клонирует текущий процесс. При этом создается иерархия в виде дочернего и родительского процесса. Возвращает число, которое имеет разную семантику в зависимости от его значения. Если возвращаемое число равно `0`, то текущий процесс – дочерний, если больше нуля – родительский и обозначает идентификатор дочернего процесса. Если же возвращаемое число отрицательное, то при клонировании процесса произошла ошибка.
9. `waitpid` – ожидает дочерний процесс или группу процессов по его идентификатору, переданному в качестве первого аргумента. В качестве второго аргумента передается указатель на область в памяти, куда занесется статус, с которым завершился процесс. Если было передано значение `NULL`, то запись не производится. В качестве третьего аргумента можно передать дополнительные опции для выполнения ожидания. При успешном выполнении возвращает идентификатор процесса, состояние которого изменилось, в противном случае возвращает `-1`.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы `open`, read, `write`, `close`.
2. Изучить принципы работы `fork`, `waitpid`
3. Написать библиотеку `io_line` для работы со считыванием и записью строк
4. Написать библиотеку pipe для абстрагирования работы с каналами
5. Написать библиотеку `remove_vowels` для удаления гласных букв в строке

Основные файлы программы

io_line/io_line.h

```
#ifndef IO_LINE_H
#define IO_LINE_H

#include <stdlib.h>

#define IO_LINE_OK 0
#define IO_LINE_INVALID_DESCRIPTOR 11
#define IO_LINE_ACCESS_DENIED 12
#define IO_LINE_NOT_ENOUGH_BUFFER_SIZE 13

typedef int Descriptor;
typedef char* String;
typedef size_t StringSize;

int write_line(Descriptor d, String line);
int read_line(Descriptor d, String buffer, StringSize buffer_size);

#endif
```

io_line/io_line.c

```
#include "io_line.h"
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>


bool __is_endline_symbol(char symbol) {
    return (symbol == EOF || symbol == '\n');
}

bool __try_to_write(Descriptor d, void * data, size_t data_size) {
    return write(d, data, data_size) == data_size;
}

bool __try_to_read(Descriptor d, void * data, size_t data_size) {
    return read(d, data, data_size) == data_size;
}

int __handle_file_error(int error) {
    if (error == EACCES)
        return IO_LINE_ACCESS_DENIED;

    if (error == EBADF)
        return IO_LINE_INVALID_DESCRIPTOR;
}
```

```
int __get_char(Descriptor d, char * symbol) {
    if (!__try_to_read(d, symbol, 1 * sizeof(char)))
        return __handle_file_error(errno);

    return IO_LINE_OK;
}

int write_line(Descriptor d, String line) {
    StringSize line_length = strlen(line);

    if (!__try_to_write(d, line, line_length * sizeof(char)))
        return __handle_file_error(errno);

    if (!__try_to_write(d, "\n", 1 * sizeof(char)))
        return __handle_file_error(errno);

    return IO_LINE_OK;
}

int read_line(Descriptor d, String buffer, StringSize buffer_size) {
    char symbol;
    size_t index = 0;
    int result;

    while (1) {
        result = __get_char(d, & symbol);

        if (result != IO_LINE_OK)
            return result;

        if (__is_endline_symbol(symbol))
            break;

        if (symbol == '\r')
            continue;

        if (index >= buffer_size - 1)
            return IO_LINE_NOT_ENOUGH_BUFFER_SIZE;

        buffer[index++] = symbol;
    }

    buffer[index] = '\0';
    return IO_LINE_OK;
}
```

pipe/pipe.h

```
#ifndef __PIPE_H__
#define __PIPE_H__

#include <stdbool.h>
```

```
#define PIPE_OK 0
#define PIPE_ERROR_INIT_PIPE 21
#define PIPE_ERROR_MODE_ALREADY_DEFINED 22
#define PIPE_ERROR_ACCESS_DENIED 23

typedef enum {
    WRITE,
    READ
} PipeMode;

struct __Pipe {
    int input_dp;
    int output_dp;

    PipeMode mode;
    bool mode_defined;
};

typedef struct __Pipe Pipe;

int init_pipe(Pipe * p);
int set_mode(Pipe * p, PipeMode mode);
int get_input_descriptor(Pipe * p, int * out_value);
int get_output_descriptor(Pipe * p, int * out_value);
void close_pipe(Pipe * p);

#endif
```

pipe/pipe.c

```
#include "pipe.h"
#include <unistd.h>


int init_pipe(Pipe * p) {
    int dp[2];

    if (pipe(dp) != 0)
        return PIPE_ERROR_INIT_PIPE;

    p -> input_dp = dp[0];
    p -> output_dp = dp[1];
    p -> mode_defined = false;

    return PIPE_OK;
}

int set_mode(Pipe * p, PipeMode mode) {
    if (p -> mode_defined)
        return PIPE_ERROR_MODE_ALREADY_DEFINED;

    if (mode == WRITE)
```

```
        close(p -> input_dp);

    if (mode == READ)
        close(p -> output_dp);

    p -> mode = mode;
    p -> mode_defined = true;

    return PIPE_OK;
}

int get_input_descriptor(Pipe * p, int * out_value) {
    if (!p -> mode_defined || p -> mode != READ)
        return PIPE_ERROR_ACCESS_DENIED;

    * out_value = p -> input_dp;
    return PIPE_OK;
}

int get_output_descriptor(Pipe * p, int * out_value) {
    if (!p -> mode_defined || p -> mode != WRITE)
        return PIPE_ERROR_ACCESS_DENIED;

    * out_value = p -> output_dp;
    return PIPE_OK;
}

void close_pipe(Pipe * p) {
    if (!p -> mode_defined || p -> mode == WRITE)
        close(p -> output_dp);

    if (!p -> mode_defined || p -> mode == READ)
        close(p -> input_dp);

    p -> mode_defined = false;
}
```

remove_vowels/remove_vowels.h

```
#ifndef __REMOVE_VOWELS_H__
#define __REMOVE_VOWELS_H__

#include <stdlib.h> // size_t

#define REMOVE_VOWELS_OK 0
#define REMOVE_VOWELS_ERROR_NOT_ENOUGH_BUFFER_LEN 31

int remove_vowels(char * s, char * buff, size_t buff_size);

#endif
```

remove_vowels/remove_vowels.c

```
#include "remove_vowels.h"
#include <string.h>
#include <stdbool.h>

char * __vowels = "eyuioaEYUIOA";

bool __is_vowel(char symbol) {
    for (char * c = __vowels;* c != '\0'; c++)
        if ( * c == symbol)
            return true;

    return false;
}

int remove_vowels(char * s, char * buff, size_t buff_size) {
    int index = 0;
    size_t s_length = strlen(s);
    int ret_value = REMOVE_VOWELS_OK;

    for (int j = 0;
        (j < s_length) && (index < buff_size - 1); j++) {
        if (__is_vowel(s[j]))
            continue;

        buff[index++] = s[j];
    }

    buff[index] = '\0';

    if (index == buff_size - 1)
        return REMOVE_VOWELS_ERROR_NOT_ENOUGH_BUFFER_LEN;
    return REMOVE_VOWELS_OK;
}
```

manager.c

```
#include "io_line/io_line.h"
#include "pipe/pipe.h"
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>

#define MAX_FILEPATH_LEN 256
#define MAX_INPUT_STRING_LEN 256

#define CREATE_PROCESS_ERROR 41
#define CREATE_PIPE_ERROR 42
```

```
#define WORKER_ERROR_EXECUTE_FILE 43

#define MANAGER_ERROR_CREATE_RESULT_FILE 44
#define MANAGER_ERROR_RESULT_FILE_NOT_SPECIFIED 45
#define MANAGER_ERROR_OPEN_RESULT_FILE 46

#define WORKER_PATH "worker"

void protected_create_file(char * filename) {
    int descriptor;

    descriptor = open(filename, O_CREAT, S_IWRITE | S_IREAD);

    if (descriptor == -1)
        exit(MANAGER_ERROR_CREATE_RESULT_FILE);

    close(descriptor);
}

char * protected_get_result_file(int argc, char * argv[], int index) {
    int descriptor;

    if (index >= argc)
        exit(MANAGER_ERROR_RESULT_FILE_NOT_SPECIFIED);

    if (access(argv[index], F_OK) != 0)
        protected_create_file(argv[index]);

    descriptor = open(argv[index], O_WRONLY);

    if (descriptor == -1)
        exit(MANAGER_ERROR_OPEN_RESULT_FILE);

    close(descriptor);
    return argv[index];
}

void init_worker(int input_file_descriptor, char * result_filepath) {
    char * args[] = {
        WORKER_PATH,
        result_filepath,
        NULL
    };

    dup2(input_file_descriptor, STDIN_FILENO);
    execv(WORKER_PATH, args);
    exit(WORKER_ERROR_EXECUTE_FILE);
}

void protected_init_pipe(Pipe * p) {
    if (init_pipe(p) != PIPE_OK)
        exit(res);
}
```

```
int protected_get_input_descriptor(Pipe * p) {
    int pipe_dp;
    int result;

    result = get_input_descriptor(p, & pipe_dp);

    if (result != PIPE_OK)
        exit(result);

    return pipe_dp;
}

int protected_get_output_descriptor(Pipe * p) {
    int pipe_dp;
    int result;

    result = get_output_descriptor(p, & pipe_dp);

    if (result != PIPE_OK)
        exit(result);

    return pipe_dp;
}

void send_data(int pipe_descriptor, char * data) {
    size_t string_len;

    string_len = strlen(data);
    data[string_len] = '\n';
    write(pipe_descriptor, data, string_len + 1);
}

pid_t create_worker(Pipe * p, char * result_filepath) {
    pid_t pid = fork();

    if (pid < 0)
        exit(CREATE_PROCESS_ERROR);

    if (pid == 0) {
        set_mode(p, READ);
        init_worker(protected_get_input_descriptor(p), result_filepath);
    }

    set_mode(p, WRITE);
    return pid;
}

void close_worker(pid_t worker_id, int pipe_descriptor) {
    write(pipe_descriptor, "\n", 1);
    waitpid(worker_id, NULL, 0);
}

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

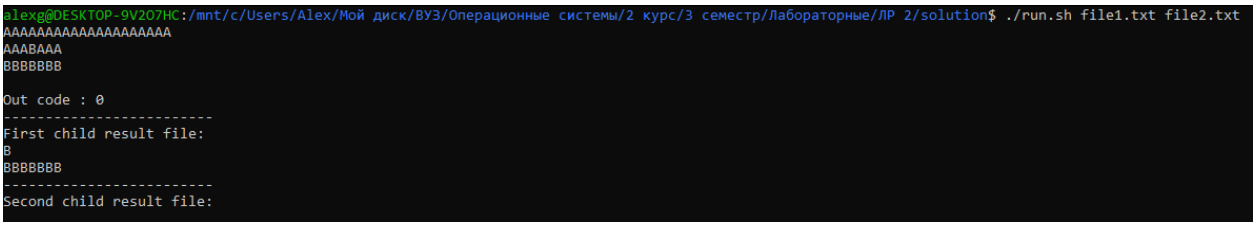
```
    char * first_filepath;
    char * second_filepath;
    char input_string[MAX_INPUT_STRING_LEN];

    pid_t first_worker_id, second_worker_id;

    Pipe pipe1, pipe2;

    first_filepath = protected_get_result_file(argc, argv, 1);
    second_filepath = protected_get_result_file(argc, argv, 2);

    protected_init_pipe( & pipe1);
    first_worker_id = create_worker( & pipe1, first_filepath);

    protected_init_pipe( & pipe2);
    second_worker_id = create_worker( & pipe2, second_filepath);

    while (1) {
        int res = read_line(
            STDIN_FILENO,
            input_string,
            MAX_INPUT_STRING_LEN);

        if (res != IO_LINE_OK)
            exit(res);

        if (strlen(input_string) == 0)
            break;

        if (strlen(input_string) > 10)
            send_data(
                protected_get_output_descriptor( & pipe2),
                input_string);

        else
            send_data(
                protected_get_output_descriptor( & pipe1),
                input_string);
    }

    close_worker(
        first_worker_id,
        protected_get_output_descriptor( & pipe1));
    close_worker(
        second_worker_id,
        protected_get_output_descriptor( & pipe2));

    exit(0);
}
```

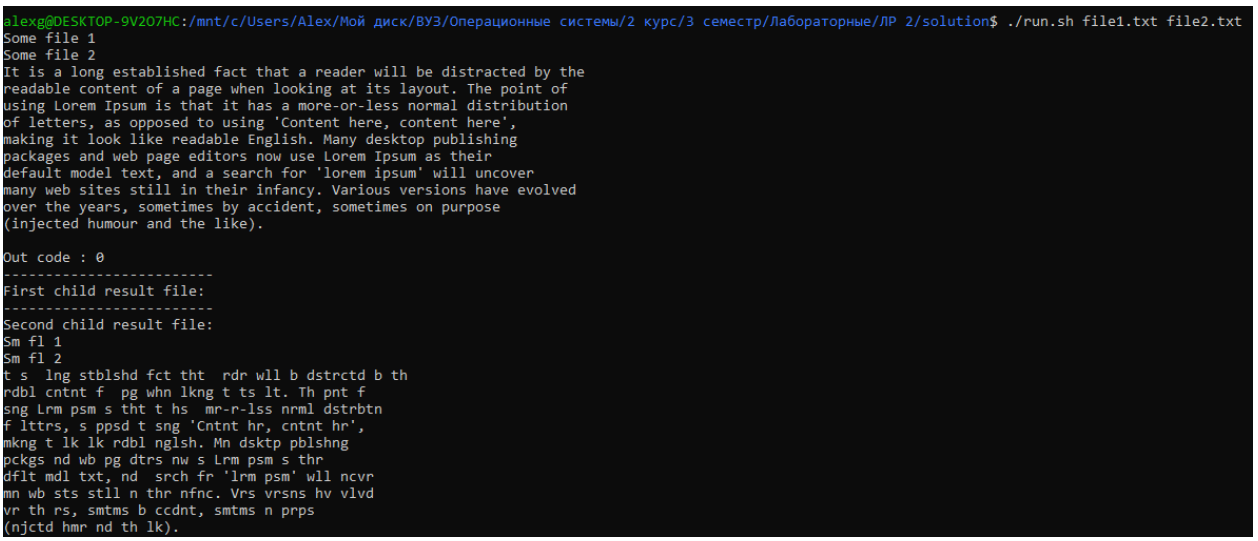
worker.c

```
#include "io_line/io_line.h"
#include "remove_vowels/remove_vowels.h"
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define RESULT_FILE_NOT_SPECIFIED 51
#define FAILED_TO_OPEN_RESULT_FILE 52

#define MAX_INPUT_LINE_LEN 256

int protected_open_result_file_descriptor(int argc, char * argv[]) {
    int descriptor;

    if (argc < 2)
        exit(RESULT_FILE_NOT_SPECIFIED);

    descriptor = open(argv[1], O_WRONLY | O_TRUNC)
    if (descriptor == -1)
        exit(FAILED_TO_OPEN_RESULT_FILE);

    return descriptor;
}

void protected_get_data(int descriptor, char * buff, size_t buff_size) {
    int result = read_line(descriptor, buff, buff_size);

    if (result != IO_LINE_OK)
        exit(result);
}

void protected_write_line(int descriptor, char * line) {
    int result = write_line(descriptor, line);

    if (result != IO_LINE_OK)
        exit(result);
}

void protected_format_line(char * input_line, char * buff, size_t buff_size)
{
    int result = remove_vowels(input_line, buff, buff_size);

    if (result != REMOVE_VOWELS_OK)
        exit(result);
}

int main(int argc, char * argv[]) {
    int result_file_descriptor;
    char input_line[MAX_INPUT_LINE_LEN];
    char formatted_line[MAX_INPUT_LINE_LEN];
```

```
    result_file_descriptor = protected_open_result_file_descriptor(
        argc, argv);
    protected_get_data(STDIN_FILENO, input_line, MAX_INPUT_LINE_LEN);

    while (strlen(input_line) != 0) {
        protected_format_line(
            input_line, formatted_line, MAX_INPUT_LINE_LEN);
        protected_write_line(result_file_descriptor, formatted_line);
        protected_get_data(STDIN_FILENO, input_line, MAX_INPUT_LINE_LEN);
    }

    close(result_file_descriptor);
    return 0;
}
```

Пример работы





Вывод

В ходе выполнения данной лабораторной работы я получил навыки в разработке много процессных программ, изучил инструменты для межпроцессного взаимодействия. Ознакомился с

основными системными вызовами Unix для работы с файлами, каналами, заменой образа программы.

Также, в ходе выполнения данной лабораторной работы я познакомился с особенностями организации памяти в программах, написанных на языке С, столкнулся с рядом проблем при передаче данных с использованием статических объектов.