Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Операционные системы" №1

Студент: Пирязев М.А.

Преподаватель: Миронов Е.С.

Группа: М8О-207Б-22

Дата: 28.09.2022

Оценка:

Подпись:

Москва 2023 г.

Оглавление

Цель работы

Приобретение практических навыков в:

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

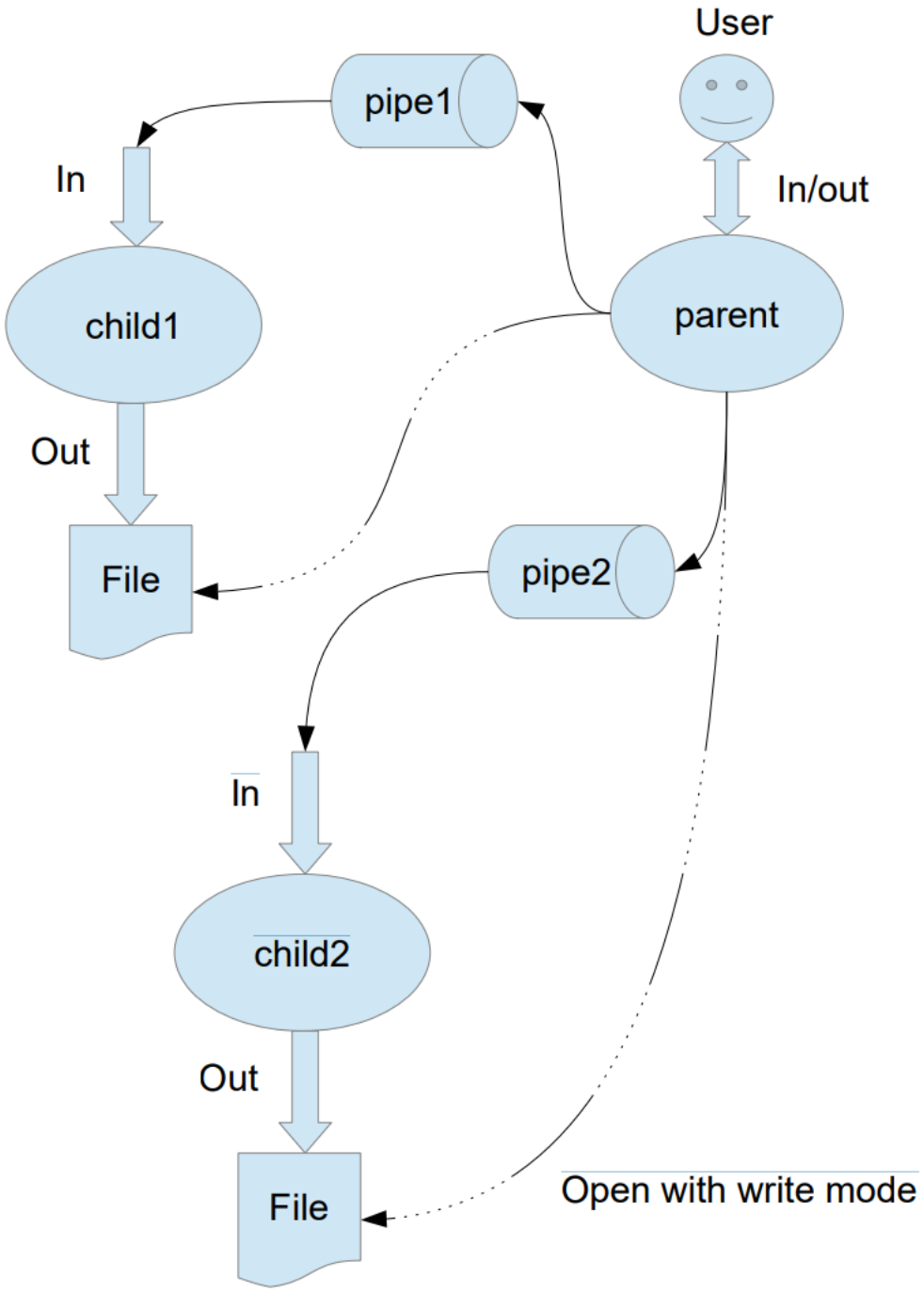
Постановка задачи

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 решения 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.

Группа вариантов 2

Группа вариантов 5



Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Вариант 21.

Перед началом работы программы программа ожидает названия двух файлов чтобы создать их и вывести в них результат работы. После этого программа ожидает входные строки, которые инвертирует и запишет в файл. Пустые строки программа игнорирует и переходит к следующим.

Общие сведения о программе

Программа представлена двумя файлами – laba3.c и laba_child3.c.

В программе используются следующие системные вызовы:

`pipe()` – создаёт однонаправленный канал данных, по принципу работы похожий на очередь, который можно использовать для взаимодействия между процессами(конвейер)

`fork()` – создание дочернего процесса, в переменной id будет лежать «специальный код» процесса(-1 -ошибка, 0- дочерний процесс, >0- родительский)

`open()` – открывает/создает файл, возвращает файловый дескриптор

`read()` – чтение из канала pipe()

`write()` – запись в канал pipe()

`dup2()` – перенаправление дескриптора

`exev()` – создание процесса с другой программой

`close()` – закрытие файлового дескриптора, который после этого не ссылается ни на один и файл и может быть использован повторно.

Общий алгоритм решения

Программа получает от пользователя имена файлов и строки, которые потом инвертирует.
Функция error_check получает 2 аргумента на вход: int received и char * output. Если какой- то из системных вызовов вернет -1, то функция выведет указанное сообщение об ошибке, которое мы передали как второй аргумент.

После этого идет функция pull_string_from_file. На вход она ничего не получает, получает символы из стандартного входа. Работает как вектор. Есть несколько переменных, которые регулируют процесс приема строки на вход: полученная длина, вместительность, строка на выход, входной символ. Рассмотрим несколько случаев: если входной символ равено NULL, то функция нам его и вернет. Если \n –

то начнет считывать следующую строку. Далее запускается цикл, в процессе которого ф-ция получает символы и проверяет вместимость импровизированного вектора. Если он полон, то увеличивает его вдвое, умножая вместимость на 2, после чего с помощью realloc, увеличивая вместимость строки, которая позже отправится на выход из функции. Записываем в строку символ, увеличиваем размер длину (размер) вектора, и считывая новый символ. Если встречается символ окончания строки, то цикл завершается. Возвращаем полученную строку.

43 - 53

Далее создаем pipe1 и pipe2. Проверяем созданы ли они. После этого создаем специальные строки в которые считаем имена файлов, которые созданим несколькими строками позже. Далее 2 файловых дескриптора для файлов, в которые откроем с указанными названиями.

54 – 66

Создается первый дочерний процесс, проверяется на то, создан ли он. Если переменная индекса равна 0, значит мы находимся в дочернем процессе. Закрываем в нем pipe2 и оставляем pipe1 на чтение, чтобы приготовить процесс для чтения. После этого перенаправляем стандартный ввод в pipe1. **Теперь дочерний процесс будет считать pipe1 как stdin.** Далее закрываем pipe1 так как ввод в дочерний процесс закончен. Далее переопределяем стандартный вывод в файловый дескриптор output_file_first. Теперь дочерний процесс считает файл своим stdout. Закрываем файл, после чего передаем в дочерний процесс исполняемую программу с помощью execv. Как кроме передаваемой программы никаких аргументов мы не передаем, то просто вписываем NULL.

Во втором дочернем процессе все аналогично, разнятся только номера pipe.

81 – 101

Если еще else, то мы оказываемся в родительском процессе. Закрываем оба pipe на чтение, так как готовимся записывать в них строки. Создаем буферную переменную string_from_file, и переменную которая будет считать порядковые номера строк и определять четные они или нет, после чего передавать их в дочерний процесс, который их обработает и запишет в файл.

Реализация

child.c

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

```
void error_checking(int result, char* error)
{
    if (result == -1)
    {
        printf("%s\n", error);
        exit(-1);
    }
}


int is_vowel(char symbol)
{
    symbol = tolower(symbol);

    if (symbol == 'a' || symbol == 'e' || symbol == 'i' || symbol == 'o' || symbol ==
'u' || symbol == 'y')
    {
        return 1;
    }

    return 0;
}

char* delete_vowels(char input_string[])
{
    int input_string_size = strlen(input_string);
    int output_string_size = 0;
    char* output_string = (char*) malloc(sizeof(char));

    for (int index = 0; index < input_string_size; ++index)
    {
        if (!is_vowel(input_string[index]))
        {
            output_string[output_string_size] = input_string[index];
            ++output_string_size;
            output_string = (char*)realloc(output_string, (output_string_size + 1) *
sizeof(char));
        }
    }

    output_string[output_string_size] = '\0';
    return output_string;
}


int main()
{

    int input_size;
    while (read(fileno(stdin), &input_size, sizeof(int)) != 0)
```

```
    {
        char input_string[input_size];
        read(fileno(stdin), &input_string, sizeof(char) * input_size);
        char* new_string;
        new_string = delete_vowels(input_string);
        write(fileno(stdout), new_string, sizeof(char) * strlen(new_string));
        write(fileno(stdout), "\n", sizeof(char));
    }


    return 0;
}
```

---

main.c

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


void error_checking(int result, char* error)
{
    if (result == -1)
    {
        printf("%s\n", error);
        exit(-1);
    }
}

char* get_string()
{
    int string_length = 0;
    int capacity = 1;
    char* string = (char*) malloc(sizeof(char));
    char symbol = getchar();

    if (symbol == EOF)
    {
        return NULL;
    }

    if (symbol == '\n')
    {
        symbol = getchar();
    }
```

```
    while (symbol != '\n')
    {
        string[string_length] = symbol;
        ++string_length;

        if (string_length >= capacity)
        {
            capacity *= 2;
            string = (char*) realloc(string, capacity * sizeof(char));
        }

        symbol = getchar();
    }

    string[string_length] = '\0';
    return string;
}


int main()
{
    int first_pipe_fd[2];
    int second_pipe_fd[2];

    error_checking(pipe(first_pipe_fd), "Pipe creating error");
    error_checking(pipe(second_pipe_fd), "Pipe creating error");

    char* first_filename;
    first_filename = get_string();

    char* second_filename;
    second_filename = get_string();

    int first_file, second_file;

    error_checking(first_file = open(first_filename, O_RDWR | O_CREAT, 0777), "File
openning error");
    error_checking(second_file = open(second_filename, O_RDWR | O_CREAT, 0777), "File
openning error");

    pid_t first_process_id = fork();
    error_checking(first_process_id, "Fork error");

    if (first_process_id == 0)
    {
        pid_t second_process_id = fork();
        error_checking(second_process_id, "Fork error");

        if (second_process_id == 0)
        {
```

```
            printf("Second child process\n");
            close(second_pipe_fd[1]);
            close(first_pipe_fd[0]);
            close(first_pipe_fd[1]);
            error_checking(dup2(second_pipe_fd[0], fileno(stdin)), "dup2 error");
            close(second_pipe_fd[0]);
            error_checking(dup2(second_file, fileno(stdout)), "dup2 error");
            close(second_file);
            char* const* argv = NULL;
            error_checking(execv("child.out", argv), "Execv error");
        }

        else
        {
            printf("First child process\n");
            close(first_pipe_fd[1]);
            close(second_pipe_fd[0]);
            close(second_pipe_fd[1]);
            error_checking(dup2(first_pipe_fd[0], fileno(stdin)), "dup2 error");
            close(first_pipe_fd[0]);
            error_checking(dup2(first_file, fileno(stdout)), "dup2 error");
            close(first_file);
            char* const* argv = NULL;
            error_checking(execv("child.out", argv), "Execv error");

        }
    }

    else
    {
        printf("Parent process\n");
        close(first_pipe_fd[0]);
        close(second_pipe_fd[0]);
        char* input_string;
        while ((input_string = get_string()) != NULL)
        {
            int string_length = strlen(input_string);
            int random_number = (rand() % 10) + 1;
            if (random_number <= 8)
            {
                write(first_pipe_fd[1], &string_length, sizeof(int));
                write(first_pipe_fd[1], input_string, sizeof(char) * string_length);
            }

            else
            {
                write(second_pipe_fd[1], &string_length, sizeof(int));
                write(second_pipe_fd[1], input_string, sizeof(char) * string_length);
            }
        }
```

```
        close(first_pipe_fd[1]);
        close(second_pipe_fd[1]);
        close(first_file);
        close(second_file);
    }

    return 0;
}
```

Пример работы

Test 1

Input	Output
Yacht Zb33 academy string abc BOB Lbv 32_97 EXTRemeLY_Long string 13424374874 \(*^%$&^_ djsan	33bZ cba gnirts thcaY ymedaca vbL BOB 79_23 nasjd _^&$%^*(\ 47847342431 gnirts gnoL_YLemeRTXE

Вывод

Самой главной частью лабораторной работы являются системные вызовы, которые подключаются библиотекой unistd.h. Открыват (запускать) процессы дает возможность системный вызов fork(), взаимодействие с процессами по сценарию лабораторной работы осуществляется с помощью системного вызова pipe. Перенаправление стандартного ввода и вывода между программой и файлами осуществляется с помощью системного вызова dup2(). Для того чтобы передать инструкции (программу) дочерним процессам используется системный вызов exec() и его разновидности.