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: Брюханов З. Д.

Преподаватель: Миронов Е.С.

Группа: М8О-207Б-22

Дата: 15.09.2022

Оценка:

Подпись:

Москва 2023 г.

Оглавление

Цель работы

Приобретение практических навыков в:

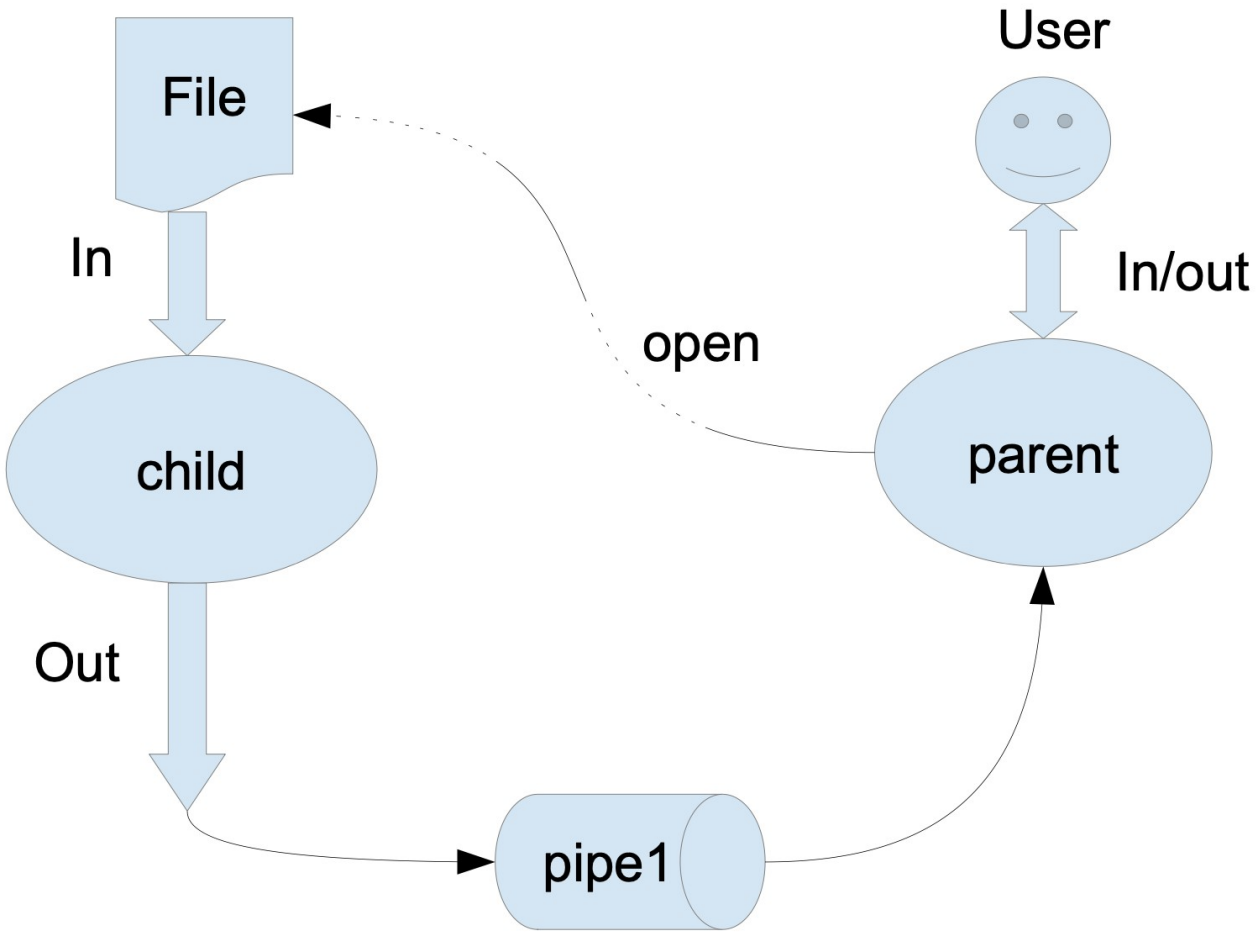
- Управлении процессами в ОС
- Обеспечении обмена данных между процессами посредством каналов

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



Родительский процесс создает дочерний процесс. Первой строчкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия файла с таким именем на чтение. Стандартный поток ввода дочернего процесса переопределяется открытым файлом. Дочерний процесс читает команды из стандартного потока ввода. Стандартный поток вывода дочернего процесса перенаправляется в pipe1. Родительский процесс читает из pipe1 и прочитанное выводит в свой стандартный поток вывода. Родительский и дочерний процесс должны быть представлены разными программами.

Вариант 8.

В файле записаны команды вида:«число число число<endline>». Дочерний процесс производит деление первого числа команда, на последующие числа в команде, а результат выводит в стандартный поток вывода. Если происходит деление на 0, то тогда дочерний и родительский процесс завершают свою работу. Проверка деления на 0 должна осуществляться на стороне дочернего процесса. Числа имеют тип int. Количество чисел может быть произвольным.

Общие сведения о программе

Программа представлена двумя файлами – main.c и child.c.

В программе используются следующие системные вызовы:

`pipe()` – создаёт однонаправленный канал данных, 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

`execl()` – создание процесса с другой программой

`exet()` – завершение выполнения процесса и возвращение статуса

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

Общий алгоритм решения

Считываем имя файла и открываем его на чтение. Создаем **pipe** для передачи результата вычислений из дочернего процесса в родительский.

Создаем дочерний процесс, и заменяем стандартный поток ввода переданным файлом (для дочернего процесса), используя системный вызов **dup2()**. Запускаем программу **child**, используя системный вызов **execl()**, передаем ей в аргументы файловый дескриптор пайпа на запись. Читаем строку и делим ее, проверяя деление на 0. Результат вычислений передаем в **pipe**.

Для родительского процесса читаем **pipe**, пока читается. Выводим в стандартный поток вывода данные из **pipe.**

Реализация

child.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
int main(int argc __attribute__((unused)), char* argv[]) {
    const int max_size_of_string = 50;
    int out_descriptor = atoi(argv[0]);
    float division_result = 0;
    char character;
    bool is_this_first_number = true;
    while ((read(STDIN_FILENO, &character, 1)) > 0) {
        char* buffer = malloc(max_size_of_string * sizeof(char));
        int index_of_buffer = 0;
        while (character != ' ' && character != '\n' && character != '\0') {
            buffer[index_of_buffer++] = character;
            if (read(STDIN_FILENO, &character, 1) <= 0) {
                character = EOF;
                break;
            }
        }
        if (is_this_first_number) {
            division_result = strtof(buffer, NULL);
            is_this_first_number = false;
        } else {
            int number = atoi(buffer);
            if (number == 0) {
```

```
                float error = -1;
                write(out_descriptor, &error, sizeof(float));
                exit(-1);
            }
            division_result /= (float) number;
            if (character == '\n' || character == EOF) {
                write(out_descriptor, &division_result, sizeof(float));
                is_this_first_number = true;
            }
        }
    }
    close(out_descriptor);
    return 0;
}
```

---

main.c

```
#include <stdio.h>
#include <stdbool.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
void error_processing(bool exception, char* bug_report) {
    if (exception) {
        write(STDERR_FILENO, bug_report, strlen(bug_report) * sizeof(char));
        exit(-1);
    }
}
void file_name_correction(char* file_name) {
    for (unsigned long symbol_index = 0; symbol_index < strlen(file_name); +
+symbol_index) {
        if (file_name[symbol_index] == '\n') {
            file_name[symbol_index] = '\0';
            return;
        }
    }
}
int main() {
    const int max_size_of_string = 50;
    int pipe_[2];
    error_processing((pipe(pipe_) == -1), "pipe error");
    char file_name[max_size_of_string];
    error_processing(read(fileno(stdin), file_name, sizeof(file_name)) <= 0,
"error reading form stdin");
    file_name_correction(file_name);
    int file_descriptor;
    error_processing((file_descriptor = open(file_name, O_RDONLY)) == -1, "Can't
open file");
    pid_t process_id = fork();
    error_processing((process_id == -1), "process creation error");
    if (process_id == 0) {
        close(pipe_[0]);
        error_processing(dup2(file_descriptor, STDIN_FILENO) < 0, "error dub");
        char out_descriptor[max_size_of_string];
        error_processing(sprintf(out_descriptor, "%d", pipe_[1]) < 0, "error
cast");
```

```
        error_processing(execl("child", out_descriptor, NULL) < 0, "child process
startup error");
    }
    close(pipe_[1]);
    float result_of_child_program;
    char answer[max_size_of_string];
    while ((read(pipe_[0], &result_of_child_program, sizeof(float))) > 0) {
        error_processing(result_of_child_program == -1, "division by 0");
        error_processing(sprintf(answer, "%f", result_of_child_program) < 0,
"error cast");
        error_processing(write(STDOUT_FILENO, answer, strlen(answer)) == -1,
"response output error");
        error_processing(write(STDOUT_FILENO, "\n", 1) == -1, "line break output
error");
    }
    close(pipe_[0]);
    return 0;
}
```

Пример работы

Test 1

Input	Output
4 2	2.000000
10 2 5	1.000000
12 3 4	1.000000
0 1	0.000000
132 2	66.000000
100 5 4 6 8 101	0.001031
5 0	division by 0

Вывод

В ходе выполнения лабораторной работы, я познакомился с системными вызовами и межпроцессным взаимодействием. Чтобы создать новый процесс я использовал системный вызов fork, а также **execl** для запуска процесса с другой программой. Благодаря этим системным вызовам легко управлять процессами, каждый из которых выполняет свой участок кода. Я также разобрался, как используя **pipe** можно удобно передавать данные между процессами. Системный вызов **dup2** помог поменять дескриптор ввода на дескриптор файла, благодаря чему я смог читать из стандартного потока ввода в дочерней программе данные из файла. Стоит не забывать закрывать все открытые и

неиспользуемые дескрипторы, потому что если они закончатся, попытка открыть очередной дескриптор завершится неудачей. Используя системные вызовы из данной лабораторной работы, можно случайно создать «fork-бомбу», поэтому стоит быть осторожней при работе с ними.