Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Операционные системы" №3

Студент: Брюханов З. Д.

Преподаватель: Миронов Е.С.

Группа: М8О-207Б-22

Дата: 27.10.2023

Оценка:

Подпись:

Москва 2023 г.

Оглавление

Цель работы

Приобретение практических навыков в:

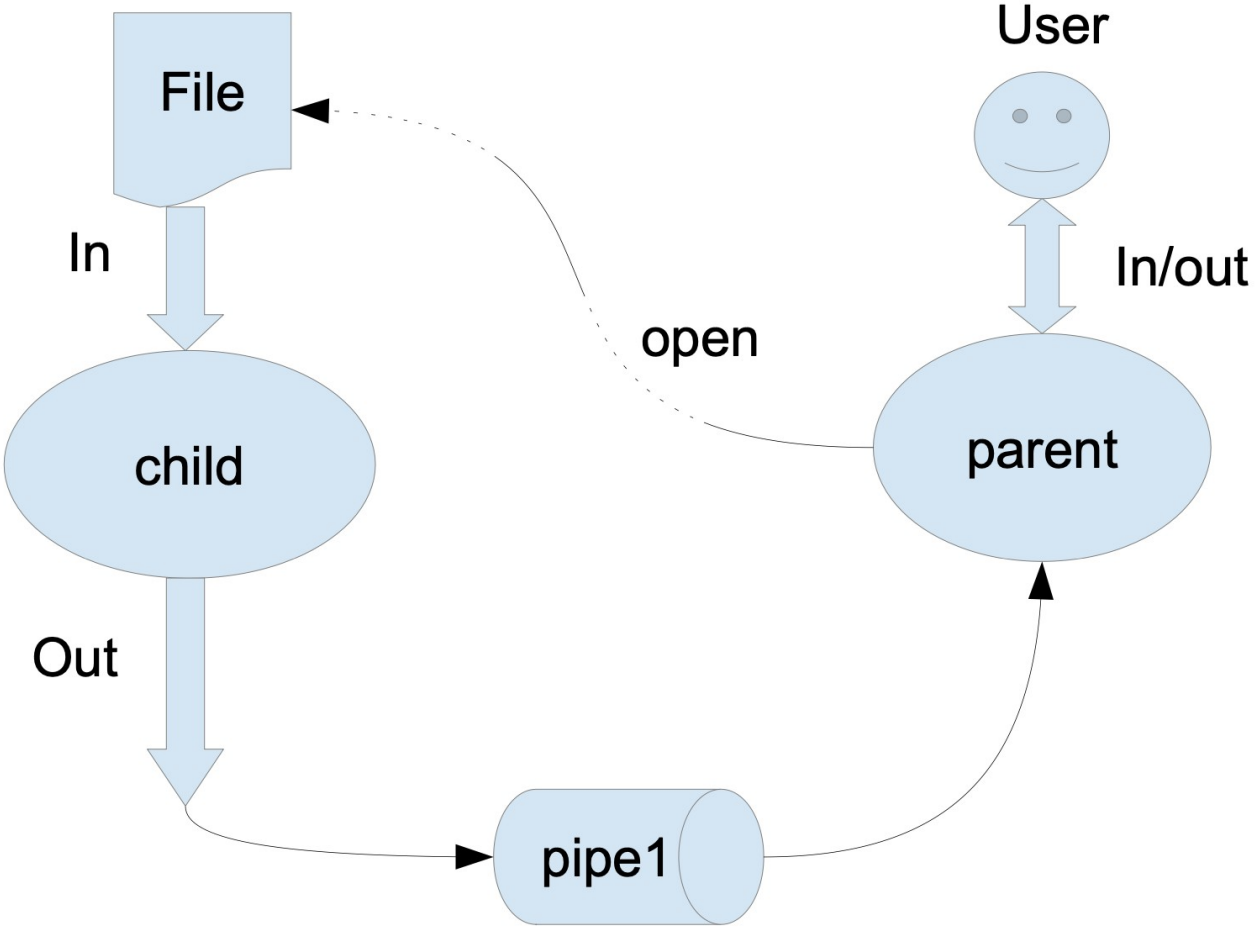
- Освоение принципов работы с файловыми системами
- Обеспечение обмена данных между процессами посредством технологии «File mapping»

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 решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или через отображаемые файлы (memory-mapped files).

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.

Общий алгоритм решения

Используя **shm_open** и **mmap** создаем отображение файла в память, я также создаю семафор используя **sem_open** для синхронизации работы родительской и дочерней программы. Далее открываю файл на чтение, перенаправляю поток входных данных из файла для дочерней программы используя **dup2.** Запускаю дочернюю программу. Ждем когда дочерняя программа разблокирует семафор, после чего проверяем деление на 0 и в случае чего завершаем программу. Выводим информацию о делении на экран и разблокируем семафор чтобы запустить дальнейшую работу дочерней программы. Дочерняя программа открывает туже память что и дочерняя и тот же семафор. Она читает данные из стандартного потока ввода и обрабатывает их. Результаты своей работы она кладет в общую память, откуда эти данные прочитает родительская программа.

Реализация

child.c

```c
#include "../include/general.h"
int main() {
```

```
    int file_descriptor_for_exchange = shm_open(SHARED_MEMORY_NAME, O_CREAT |
O_RDWR, ACCESS_MODE);
    error_processing(file_descriptor_for_exchange == -1, "Shm_open error");
    sem_t* semaphore = sem_open(SEMAPHORE_NAME, O_CREAT, ACCESS_MODE, 0);
    error_processing(semaphore == SEM_FAILED, "Sem_open error");
    void* result_data_for_exchange = mmap(NULL, SIZE_SHARED_MEMORY, PROT_READ |
PROT_WRITE, MAP_SHARED,
                                          file_descriptor_for_exchange, 0);
    error_processing(result_data_for_exchange == MAP_FAILED, "Mmap error");
    float division_result = 0;
    char character;
    bool is_this_first_number = true;
    while ((read(STDIN_FILENO, &character, 1)) > 0) {
        char* buffer = malloc(sizeof(char) * MAX_SIZE_OF_STRING);
        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) {
                strcpy(result_data_for_exchange, "-1");
                sem_post(semaphore);
                sem_close(semaphore);
                exit(-1);
            }
            division_result /= (float) number;
            if (character == '\n' || character == EOF) {
                sprintf(result_data_for_exchange, "%f", division_result);
                is_this_first_number = true;
                sem_post(semaphore);
                sem_wait(semaphore);
            }
        }
    }
    strcpy(result_data_for_exchange, "\0");
    sem_post(semaphore);
    sem_close(semaphore);
    return 0;
}
```

---

main.c

```
#include "../include/general.h"
void string_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() {
    shm_unlink(SHARED_MEMORY_NAME);
    sem_unlink(SEMAPHORE_NAME);
    int file_descriptor_for_exchange = shm_open(SHARED_MEMORY_NAME, O_CREAT |
O_RDWR, ACCESS_MODE);
    error_processing(file_descriptor_for_exchange == -1, "Shm_open error");
    error_processing(ftruncate(file_descriptor_for_exchange, SIZE_SHARED_MEMORY)
== -1, "Truncate error");
    void* result_data_for_exchange = mmap(NULL, SIZE_SHARED_MEMORY, PROT_READ |
PROT_WRITE, MAP_SHARED,
                                          file_descriptor_for_exchange, 0);
    error_processing(result_data_for_exchange == MAP_FAILED, "Mmap error");
    strcpy(result_data_for_exchange, "\0");
    char file_name[MAX_SIZE_OF_STRING];
    error_processing(read(STDIN_FILENO, file_name, sizeof(file_name)) <= 0,
"Error reading from stdin");
    string_correction(file_name);
    int file_descriptor;
    error_processing((file_descriptor = open(file_name, O_RDONLY)) == -1, "Can't
open file");
    sem_t* semaphore = sem_open(SEMAPHORE_NAME, O_CREAT, ACCESS_MODE, 0);
    error_processing(semaphore == SEM_FAILED, "Sem_open error");
    pid_t process_id = fork();
    error_processing(process_id < 0, "Process creation error");
    if (process_id == 0) {
        error_processing(dup2(file_descriptor, STDIN_FILENO) < 0, "Error dup2");
        error_processing(execl("child", NULL) < 0, "Child process startup
error");
    }
    while (true) {
        sem_wait(semaphore);
        char* answer = (char*) result_data_for_exchange;
        if (strcmp(answer, "\0") == 0) {
            break;
        }
        error_processing(strcmp(answer, "-1") == 0, "Division by 0");
        printf("%s\n", answer);
        sem_post(semaphore);
    }
    sem_close(semaphore);
    sem_unlink(SEMAPHORE_NAME);
    shm_unlink(SHARED_MEMORY_NAME);
    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 новым способом передавать данные между различными процессами. Я воспользовался кодом из первой лабораторной работы, мне осталось только заменить **pipe** на **shm_open** и **mmap**. Стоит также заметить, что из-за разницы в применяемых технологиях мне пришлось синхронизировать работу программ используя семафоры. Из-за этого код получился логически сложнее. В этой лабораторной работе было приятнее использовать **pipe**, но не смотря на это есть случаи когда использование **mmap** дает больше плюсов, например, при работе с файлами взаимодействие через **mmap** дает прирост производительности по сравнению с обычной буферизированной работой с файлами.