Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

**ОТОБРАЖАЕМЫЕ ФАЙЛЫ**

Студент: Киреев Александр Константинович

Группа: М8О–206Б–19

Вариант: 4

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2020.

**Постановка задачи**

**Цель работы**

Приобретение практических навыков в:

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

**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 через отображаемые файлы (memory-mapped files).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Вариант 4:

Родительский процесс создает дочерний процесс. Первой строчкой пользователь в консоль родительского процесса пишет имя файла, которое будет передано при создании дочернего процесса. Родительский и дочерний процесс должны быть представлены разными программами.

Родительский процесс передает команды пользователя через pipe1, который связан с стандартным входным потоком дочернего процесса. Дочерний процесс при необходимости передает данные в родительский процесс через pipe2. Результаты своей работы дочерний процесс пишет в созданный им файл. Допускается просто открыть файл и писать туда, не перенаправляя стандартный поток вывода. Пользователь вводит команды вида: «число число число<endline>». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс производит деление первого числа, на последующие, а результат выводит в файл. 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 float.

**Общие сведения о программе**

Программа компилируется из файла main.c вместе с файлов io.c. Также используется заголовочные файлы: stdio.h, unistd.h, stdlib.h, stdbool.h, sys/types.h, sys/stat.h, fcntl.h, sys/mman.h. В программе используются следующие системные вызовы:

1. **mmap** ­­–– отражает *length* байтов, начиная со смещения *offset*файла (или другого объекта), определенного файловым дескриптором *fd*, в память, начиная с адреса *start*. При удачном выполнении mmap возвращает указатель на область с отраженными данными. При ошибке возвращается значение MAP\_FAILED (-1), а переменная *errno* приобретает соответствующее значение.
2. **munmap** ­­–– удаляет все отражения из заданной области памяти, после чего все ссылки на данную область будут вызывать ошибку "неправильное обращение к памяти". При удачном выполнении munmap возвращаемое значение равно нулю. При ошибке возвращается -1, а переменная *errno* приобретает соответствующее значение.
3. **fork ––** создает новый процесс, который является копией родительского процесса, за исключением разных process ID и parent process ID. В случае успеха fork() возвращает 0 для ребенка, число больше 0 для родителя – child ID, в случае ошибки возвращает -1.
4. **read ––** предназначена для чтения какого-то числа байт из файла, принимает в качестве аргументов файловый дескриптор, буфер, в который будут записаны данные и число байт. В случае успеха вернет число прочитанных байт, иначе -1.
5. **write ––** предназначена для записи какого-то числа байт в файл, принимает в качестве аргументов файловый дескриптор, буфер, из которого будут считаны данные для записи и число байт. В случае успеха вернет число записанных байт, иначе -1.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы mmap.
2. Написать парсер для типа float, который бы превращал строковое представление числа в естественное (учитывая не валидные данные).
3. Организовать считывание названия файла и строк чисел типа float.
4. Разработать архитектуру проекта так, чтобы избежать *race condition* при обращении к общей разделяемой памяти.
5. Реализовать функции для процесса-родителя и процесса-ребенка.
6. Реализовать сообщение между процессами при помощи отраженных данных.
7. Реализовать обработку системных ошибок согласно заданию.

**Основные файлы программы**

**io.h:**

#pragma once

#include <stdio.h>

#include <ctype.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdbool.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define BUF\_SIZE 64

#define RF\_ENDL 3

#define RF\_EOF 1

#define RF\_INVALID 2

#define RF\_VALID 0

// проигнорировать невалидную строку

void skip\_str();

// проверка числа на валидность

bool is\_valid(char\* buf);

// считать float с клавиатуры при помощи read

int read\_float(float\* n);

// считать имя файла с клавиатуры при помощи read

void read\_file\_name(char file\_name[BUF\_SIZE]);

**main.c:**

#include <stdio.h>

#include <ctype.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdbool.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <fcntl.h>

#include "io.h"

#define PAGESIZE 4096

#define CHILD\_BLOCKED 0

#define PARENT\_BLOCKED 1

#define KILL 2

#define OK 0

#define FILE\_OPEN\_ERR 1

#define FORK\_ERR 2

#define MMAP\_ERR 3

#define CLOSE\_FILE\_ERR 4

#define NULL\_DIV 5

int mmap\_clear(float\* mf, int\* sig) {

int to\_return = OK;

if (munmap(mf, PAGESIZE) == -1) {

to\_return = MMAP\_ERR;

}

if (munmap(sig, sizeof(int)) == -1) {

to\_return = MMAP\_ERR;

}

return to\_return;

}

int child(char\* file\_name, float\* mf, int\* sig) {

FILE\* fp;

if ((fp = fopen(file\_name, "w")) == NULL) {

perror("unable to open file");

exit(FILE\_OPEN\_ERR);

}

while (\*sig == CHILD\_BLOCKED) {}

while (\*sig != KILL) {

int size = mf[0];

float result = mf[1];

for (int i = 2; i < size + 1; ++i) {

if (mf[i] == 0.0) {

\*sig = KILL;

if (fclose(fp) != 0) {

perror("unable to close file");

exit(CLOSE\_FILE\_ERR);

}

exit(NULL\_DIV);

}

result /= mf[i];

}

fprintf(fp, "%f\n", result);

\*sig = CHILD\_BLOCKED;

while (\*sig == CHILD\_BLOCKED) {}

}

if (fclose(fp) != 0) {

perror("unable to close file");

exit(CLOSE\_FILE\_ERR);

}

exit(OK);

}

int parent(float\* mf, int\* sig) {

int get\_return = RF\_VALID;

while (true) {

int it = 0;

while (true) {

float num;

get\_return = read\_float(&num);

if (get\_return == RF\_INVALID) {

perror("invalid num");

break;

}

mf[it + 1] = num;

it++;

if (get\_return != RF\_VALID) {

break;

}

}

if (get\_return == RF\_EOF) {

break;

} else if (get\_return == RF\_INVALID) {

continue;

}

mf[0] = it;

\*sig = PARENT\_BLOCKED;

while (\*sig == PARENT\_BLOCKED) {}

if (\*sig == KILL) {

if (mmap\_clear(mf, sig) == MMAP\_ERR) {

perror("munmap error");

exit(MMAP\_ERR);

}

exit(NULL\_DIV);

}

}

\*sig = KILL;

if (mmap\_clear(mf, sig) == MMAP\_ERR) {

perror("munmap error");

exit(MMAP\_ERR);

}

exit(OK);

}

int main() {

char file\_name[BUF\_SIZE];

read\_file\_name(file\_name);

float\* mf = (float\*)mmap(NULL, PAGESIZE, PROT\_READ | PROT\_WRITE , MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

if (mf == MAP\_FAILED) {

perror("mmap error");

exit(MMAP\_ERR);

}

int\* sig = (int\*)mmap(NULL, sizeof(int), PROT\_READ | PROT\_WRITE , MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

if (sig == MAP\_FAILED) {

perror("mmap error");

exit(MMAP\_ERR);

}

\*sig = CHILD\_BLOCKED;

int id = fork();

if (id < 0) {

perror("fork error");

exit(FORK\_ERR);

} else if (id == 0) {

child(file\_name, mf, sig);

} else {

parent(mf, sig);

}

return 0;

}

**Пример работы**

MacBook-Air-K:test AK$ ./wrapper.sh

[info] [2020-11-16 22:40:46] Stage #1 Compiling...

gcc -pedantic -Wall -Werror -std=c11 -c io.c

gcc -pedantic -Wall -Werror -std=c11 io.o main.c -o lab4

[info] [2020-11-16 22:40:47] Stage #2 Test generating...

[info] [2020-11-16 22:40:47] Stage #3 Checking...

>>Checking test\_1.txt...

invalid num: Undefined error: 0

invalid num: Undefined error: 0

>>Checking test\_2.txt...

>>Checking test\_3.txt...

>>Checking test\_4.txt...

>>Checking test\_5.txt...

[info] [2020-11-16 22:40:47] Division by zero

MacBook-Air-K:test AK$ cd ../src/

MacBook-Air-K:src AK$ make

gcc -pedantic -Wall -Werror -std=c11 -c io.c

gcc -pedantic -Wall -Werror -std=c11 io.o main.c -o lab4

MacBook-Air-K:src AK$ ./lab4

StrangeFileName

1

2 1

4 2 2

5 10 15

64 2 2 2 2 2 2

MacBook-Air-K:src AK$ cat StrangeFileName

1.000000

2.000000

1.000000

0.033333

1.000000

MacBook-Air-K:src AK$ make clean

rm -f \*.o lab4

**Вывод**

Выполняя данную лабораторную работу я познакомился с технологией «File mapping», которая позволяет отобразить данные из файла в какой-то участок памяти, так, что при работе с данным участком памяти производится работа с файлом. Данная технология может прилично увеличить время работы с большими файлами, так как уменьшается число системных вызовов для работы с файлом, не будет происходить лишнего копирования данных в буфер.

Также при помощи «File mapping» можно разделить использование некого участка памяти между процессами, что позволит обмениваться данными между ними. Однако общий участок памяти может быть опасен возможными *race condition’*ами, так как обращение к этой области памяти не является блокирующим, что заставляет программиста пользоваться семафорами и т.п.

Суммируя полученный опыт из ЛР2 и ЛР4, я думаю, что *каналы* лучше подходят для общения “один на один”, они более просты в использовании, чем разделяемая память, они лучше подходят для синхронизации процессов, так как они блокирующие; когда как *разделяемая память* больше подходит для асинхронной работы с данными, она лучше подходит для общения между множеством процессов.