Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

Студент: Епанешников Владислав Сергеевич

Группа: М8О–206Б–19

Вариант: 1

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2021

**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.

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 принеобходимости 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.

1 вариант) Пользователь вводит команды вида: «число числочисло<endline>». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс считает их сумму и выводит её в файл. Числа имеют тип int.

**Общие сведения о программе**

Программа компилируется из файла main.c Также используется заголовочные файлы: unistd.h, stdio.h , stdlib.h, fcntl.h, errno.h, sys/mman.h, sys/stat.h, string.h, stdbool.h, ctype.h, sys/wait.h, semaphore.h. В программе используются следующие системные вызовы:

1. **mmap** – создает отображение файла в память.
2. **munmap** – снимает отображение.
3. **open** – открывает файл.
4. **close –** закрывает файл.
5. **sem\_init –** инициализация семафора.
6. **sem\_wait** – ожидание доступа, если значение семафора отрицательное, то вызывающий поток блокируется до тех пор, пока один из потоков не вызовет sem\_post.
7. **sem\_post** – увеличивает значение семафора и разблокирует ожидающие потоки.
8. **sem\_destroy** – уничтожает семафор.
9. **read** – чтение из файла в буфер.
10. **write** – запись из буфера в файл.
11. **sleep** – переход в режим ожидания на указанное количество секунд.
12. **exit** – завершение работы программы с некоторым статусом.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить работу с отображением файла в память(mmap и munmap).
2. Изучить работу с процессами(fork).
3. Создать родительский и дочерний процессы.
4. В каждом процессе отобразить файл в память, преобразовать в соответствии с вариантом и снять отображение(mmap, munmap).

**Основные файлы программы**

**main.c:**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#include <ctype.h>

#include <stdbool.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <semaphore.h>

#define STDIN 0

#define STDOUT 1

#define MIN\_CAP 4

char\* read\_string(int fd) {

size\_t str\_size = 0;

size\_t cap = MIN\_CAP;

char \*str = (char\*) malloc(sizeof(char) \* cap);

if (str == NULL) {

perror("Malloc error");

exit(-1);

}

char c;

while (read(fd, &c, sizeof(char)) == 1) {

if (c == '\n') {

break;

}

str[(str\_size)++] = c;

if (str\_size == cap) {

str = (char\*) realloc(str, sizeof(char) \* cap \* 3 / 2);

cap = cap \* 3 / 2;

if (str == NULL) {

perror("Realloc error");

exit(-2);

}

}

}

str[str\_size] = '\0';

return str;

}

int str\_length(char \*str) {

int length = 0;

for (int i = 0; str[i] != '\0'; ++i) {

++length;

}

return length;

}

char\* int\_to\_string(int sum) {

int size = 0;

int temp\_sum = sum;

while (temp\_sum != 0) {

temp\_sum /= 10;

++size;

}

bool is\_positive = true;

if (sum < 0) {

is\_positive = false;

sum \*= -1;

size++;

}

char\* str = (char\*) malloc(sizeof(char) \* (size + 1));

str[size] = '\0';

for (int i = size - 1; i >= 0; --i) {

if (!is\_positive && i == 0) {

str[0] = '-';

} else {

str[i] = sum % 10 + '0';

sum /= 10;

}

}

return str;

}

void child\_work(char str[], char path[]) {

mode\_t mode = S\_IRUSR | S\_IWUSR;

int flags = O\_WRONLY | O\_CREAT | O\_TRUNC;

int file = open(path, flags, mode);

if (file == -1) {

perror("Open error");

exit(7);

}

int sum = 0, number = 0, is\_positive = 1;

for (size\_t i = 0; str[i] != '\0'; ++i) {

if (str[i] == '-') {

is\_positive = -1;

continue;

}

if (isdigit(str[i])) {

number = number \* 10 + str[i] - '0';

}

else if (str[i] == ' ') {

sum += number \* is\_positive;

number = 0;

is\_positive = 1;

}

}

sum += number \* is\_positive;

char\* answer = int\_to\_string(sum);

int answer\_size = str\_length(answer);

printf("Child: ");

fflush(stdout);

answer[answer\_size] = '\n';

write(STDOUT, answer, sizeof(char) \* answer\_size + 1);

write(file, answer, sizeof(char) \* answer\_size);

close(file);

free(answer);

}

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

if (argc != 2) {

printf("Usage: ./a.out <filename>\n");

exit(1);

}

char \*file = argv[1];

sem\_t \*sem = sem\_open("semaphore", O\_CREAT | O\_EXCL, 0777, 0);

if (sem == SEM\_FAILED) {

perror("Sem error");

exit(10);

}

if (sem\_unlink("semaphore") == -1) {

perror("Sem\_unlink error");

exit(11);

}

int id = fork();

if (id == -1) {

perror("Can\'t fork child");

exit(2);

} else if (id > 0) {

int input = open("input.txt", O\_RDWR | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR);

if (input < 0) {

perror("Open error");

exit(2);

}

struct stat buff;

if (fstat(input, &buff) < 0) {

perror("Stat error");

exit(3);

}

char\* str = read\_string(STDIN);

size\_t str\_size = str\_length(str);

if (write(input, str, sizeof(char) \* str\_size) != str\_size) {

perror("write error");

exit(4);

}

close(input);

free(str);

sem\_post(sem);

int child\_status;

if (wait(&child\_status) == -1) {

perror("wait error");

exit(5);

}

int exit\_code = WEXITSTATUS(child\_status);

if (!exit\_code) {

printf("Parent: The child process exited normally with exit code %d\n", exit\_code);

} else {

printf("Parent: The child process exited abnormally with exit code %d\n", exit\_code);

}

if (remove("input.txt") == -1) {

perror("Remove error");

exit(111);

}

printf("Parent: exit 0\n");

}

else {

int fd = open("input.txt", O\_RDWR);

if (fd < 0) {

perror("Open error");

exit(5);

}

sem\_wait(sem);

struct stat buff;

if (fstat(fd, &buff) < 0) {

perror("Stat error");

exit(6);

}

char \*str = mmap(NULL, buff.st\_size, PROT\_READ, MAP\_SHARED, fd, 0);

if (str == MAP\_FAILED) {

perror("Mmap error");

exit(7);

}

close(fd);

child\_work(str, file);

if (munmap(str, buff.st\_size) != 0) {

perror("Munmap error");

exit(8);

}

printf("Child: exit 0\n");

}

return 0;

}

**Пример работы**

MacBook:src vladislove$ cat ../tests/test1

1 2 3 4 5 6 7 8 9 10

MacBook:src vladislove$ ./a.out file < ../tests/test1

Child: 55

Child: exit 0

Parent: The child process exited normally with exit code 0

Parent: exit 0

MacBook:src vladislove$ cat file

55MacBook:src vladislovecat ../tests/test2

-80 40 -20 10 0 1 -1

MacBook:src vladislove$ ./a.out file < ../tests/test2

Child: -50

Child: exit 0

Parent: The child process exited normally with exit code 0

Parent: exit 0

MacBook:src vladislove$ cat file

-50MacBook:src vladislove$

**Вывод**

В данной лабораторной работе мнойй был изучен и применён на практике механизм межпроцессорного взаимодействия при помощи отображаемых файлов (технология «File Mapping»).

В СИ помимо механизма общения между процессами через pipe, также существуют и другие способы взаимодействия, например отображение файла в память, такой подход работает быстрее, засчет отстутствия постоянных вызово read, write и тратит меньше памяти под кэш. После отображения возвращается void\*, который можно привести к своему указателю на тип и обрабатывать данные как массив, где возвращенный указатель – указатель на первый элемент.