Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

**Процессы операционных систем**

Студент: Шевлякова София Сергеевна

Группа: М8О–208Б–21

Вариант: 16

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2022

**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 строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись. Перенаправление стандартных потоков ввода-вывода показано на картинке выше. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1. Процесс child проверяет строки на валидность правилу. Если строка соответствует правилу, то она выводится в стандартный поток вывода дочернего процесса, иначе в pipe2 выводится информация об ошибке. Родительский процесс полученные от child ошибки выводит в стандартный поток вывода. Правило проверки: строка должна оканчиваться на «.» или «;».

**Общие сведения о программе**

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

1. **write () -** переписывает count байт из буффера в файл. Возвращает количество записанных байт или -1;
2. **mmap ()** – отображает файл на память. Возвращает указатель на начало файла, при ошибке возвращает MAP\_FAILED;
3. **munmap ()** – отменяет отображение файла на память. В случае ошибки возвращает -1;
4. **shm\_open () –** открывает или создает при необходимости объект разделяемой памяти. Возвращает дескриптор открытого файла или -1;
5. **shm\_unlink () –** обратная к shm\_open;
6. **sem\_open () –** инициализирует и открывает именованный семафор;
7. **sem\_close () –** обратная к sem\_open;
8. **sem\_post () –** увеличивает (разблокирует) семафор. Возвращает 0 при успехе и -1 при неудаче;
9. **sem\_wait () –** уменьшает (блокирует) семафор. Возвращает 0 при успехе и -1 при неудаче;
10. **ftruncate ()** – устанавливает файлу заданную длину в байтах.
11. **open () -** открывает или создаёт файл при необходимости. Возвращает дескриптор открытого файла или -1;
12. **close() -** закрывает файловый дескриптор, который больше не ссылается ни на один файл, возвращает 0 или -1;
13. **fork () -** порождается процесс-потомок. Весь код после fork() выполняется дважды, как в процессе-потомке, так и в процессе-родителе. Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родители возвращает идентификатор pid потомка или -1. Процесс-потомок возвращает 0 или -1;

**Общий метод и алгоритм решения**

Используя системный вызов mmap будет отображать файл на память. Теперь наш файл представляет собой массив символов, в него родительский процесс будет записывать, введенные пользователем строчки, а дочерний – читать и проверять на валидность правилу. Чтобы синхронизовать их работу используем семафоры, благодаря им дочерний процесс сможет понимать, что родитель записал в файл строчку, а родительский процесс поймет, что ребенок проверил ее. Если строчка не удовлетворяет условию, то дочерний процесс запишет в конец файла константу, каждый раз родительский процесс проверяет последний элемент файла, в случае, если в конце файла оказалась константа, то мы ждем завершения работы дочернего процесса и программа останавливает свою работу.

**Основные файлы программы**

======================== main.c ========================

#include <string.h>

#include <stdio.h>

#include <sys/mman.h>

#include <unistd.h>

#include <fcntl.h>

#include <semaphore.h>

#include <wait.h>

#include <sys/stat.h>

#include <stdbool.h>

#define check\_ok(VALUE, OK\_VAL, MSG) if (VALUE != OK\_VAL) { printf("%s", MSG); return 1; }

#define check\_wrong(VALUE, WRONG\_VAL, MSG) if (VALUE == WRONG\_VAL) { printf("%s", MSG); return 1; }

const int FILENAME\_LIMIT = 255;

const int BUFFER\_SIZE = FILENAME\_LIMIT;

const int SHARED\_MEMORY\_SIZE = BUFFER\_SIZE + 1;

const int STOP\_FLAG = BUFFER\_SIZE;

const char\* SHARED\_FILE\_NAME = "meow";

bool check(const char\* s, int len) {

if (len < 2) return false;

if (s[len - 2] != ';' && s[len - 2] != '.') return false;

return true;

}

int last(const char\* s) {

for (int i = 0; i < BUFFER\_SIZE; ++i) {

if (s[i] == '\0') return i;

}

return BUFFER\_SIZE;

}

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

check\_ok(argc, 2, "Specify the file name as the first argument\n")

if (strlen(argv[1]) > FILENAME\_LIMIT) {

check\_ok(1, 0, "Filename is too long\n")

}

int fd = shm\_open(SHARED\_FILE\_NAME, O\_RDWR | O\_CREAT, S\_IRWXU);

check\_wrong(fd, -1, "Error creating shared file!\n")

check\_ok(ftruncate(fd, SHARED\_MEMORY\_SIZE), 0, "Error truncating shared file!\n")

char\* map = (char\*)mmap(NULL, SHARED\_MEMORY\_SIZE, PROT\_WRITE | PROT\_READ, MAP\_SHARED, fd, 0);

check\_wrong(map, NULL, "Cant map file\n")

const char\* in\_sem\_name = "/input\_semaphor";

const char\* out\_sem\_name = "/output\_semaphor";

sem\_unlink(in\_sem\_name);

sem\_unlink(out\_sem\_name);

sem\_t\* in\_sem = sem\_open(in\_sem\_name, O\_CREAT, S\_IRWXU, 0);

check\_wrong(in\_sem, SEM\_FAILED, "Cannot create 'in' semaphore\n")

sem\_t\* out\_sem = sem\_open(out\_sem\_name, O\_CREAT, S\_IRWXU, 0);

check\_wrong(out\_sem, SEM\_FAILED, "Cannot create 'out' semaphore\n")

strcpy(map, argv[1]);

map[BUFFER\_SIZE] = (char) strlen(argv[1]);

int pid = fork();

if (pid == -1) {

check\_wrong(pid, -1, "Fork failure\n")

} else if (pid == 0) {

//child

int output\_file = open(argv[1], O\_RDWR | O\_TRUNC | O\_CREAT, S\_IREAD | S\_IWRITE);

if (output\_file == -1) {

map[BUFFER\_SIZE] = (char) STOP\_FLAG;

sem\_post(out\_sem);

check\_ok(1, -1,"Cannot create output file\n")

}

sem\_post(out\_sem);

while (true) {

sem\_wait(in\_sem);

int l = (int) map[BUFFER\_SIZE];

if (check(map, l) == false ) {

map[BUFFER\_SIZE] = (char) STOP\_FLAG;

sem\_post(out\_sem);

break;

}

check\_wrong(write(output\_file, map, map[BUFFER\_SIZE]), -1, "Cannot write fo the file\n")

sem\_post(out\_sem);

}

close(output\_file);

} else {

//parent

sem\_wait(out\_sem);

if (map[BUFFER\_SIZE] != (char) STOP\_FLAG){

memset(map, 0, BUFFER\_SIZE);

check\_wrong(fgets(map, BUFFER\_SIZE, stdin), NULL, "Unexpectedly EOF\n")

int read\_count = last(map);

map[BUFFER\_SIZE] = (char) read\_count;

sem\_post(in\_sem);

while (true) {

sem\_wait(out\_sem);

if (map[BUFFER\_SIZE] == (char) STOP\_FLAG) {

break;

}

memset(map, 0, BUFFER\_SIZE);

check\_wrong(fgets(map, BUFFER\_SIZE, stdin), NULL, "Unexpectedly EOF\n")

read\_count = last(map);

map[BUFFER\_SIZE] = (char) read\_count;

sem\_post(in\_sem);

}

int stat\_lock;

wait(&stat\_lock);

if (stat\_lock != 0) {

printf("Child failure\n");

}

} else {

int stat\_lock;

wait(&stat\_lock);

if (stat\_lock != 0) {

printf("Child failure\n");

}

}

sem\_close(in\_sem);

sem\_close(out\_sem);

check\_wrong(munmap(map, SHARED\_MEMORY\_SIZE), -1, "Error unmapping\n")

check\_wrong(shm\_unlink(SHARED\_FILE\_NAME), -1, "Error unlinking shared cond file!\n")

}

}

======================== test1.txt ========================

sdfds;

saaaaaaaaaa;

aaaa

111111.l

======================== test2.txt ========================

meow

qqqqqqqq;qqq;.q

qqq

11112232321FHNGJ......

======================== test3.txt ========================

.

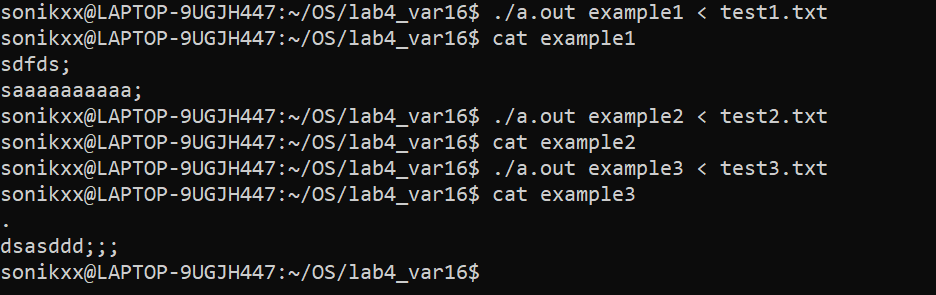
dsasddd;;;

dsassdd

meowmeow

1343212321346788

**Пример работы**

****

**Вывод**

В ходе выполнения работы я изучила основы работы с файлами, отображаемыми в память, составил программу, в которой синхронизировала работу двух процессов с помощью общих файлов, узнала, что в ОС Ubuntu общие файлы располагаются в /dev/shm, а также использовала в работе семафоры. В современных реалиях пользователю приходится открывать сразу много приложений. Поместить в память все данные может быть невозможным, поэтому при разработке ОС важно предусмотреть выгрузку фоновых процессов на диск и вовремя подгрузить их.