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;
saaaaaaaaa;
aaaa
111111.l
```

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

```
meow
qqqqqqqq;qqq;.q
qqq
11112232321FHNGJ.....
```

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

```
.
dsasddd;;;
dsassdd
meowmeow
1343212321346788
```

Пример работы

```
sonikxx@LAPTOP-9UGJH447:~/OS/lab4_var16$ ./a.out example1 < test1.txt
sonikxx@LAPTOP-9UGJH447:~/OS/lab4_var16$ cat example1
sdfds;
saaaaaaaaa;
sonikxx@LAPTOP-9UGJH447:~/OS/lab4_var16$ ./a.out example2 < test2.txt
sonikxx@LAPTOP-9UGJH447:~/OS/lab4_var16$ cat example2
sonikxx@LAPTOP-9UGJH447:~/OS/lab4_var16$ ./a.out example3 < test3.txt
sonikxx@LAPTOP-9UGJH447:~/OS/lab4_var16$ cat example3
.
dsasddd;;;
sonikxx@LAPTOP-9UGJH447:~/OS/lab4_var16$
```

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.