Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

**Отображение файлов в памяти**

Студент: Шараковский Юрий Дмитриевич

Группа: М8О–206Б–19

Вариант: 21

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 21: Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами.

Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Правило фильтрации: нечетные строки отправляются в pipe1, четные в pipe2. Дочерние процессы инвертируют строки.

**Общие сведения о программе**

Программа компилируется из файла main.c. Помимо стандартных также используется заголовочные файлы: sys/mman.h, semaphore.h. В программе используются следующие системные и библиотечные вызовы:

1. **getpid** – для получения PID процесса.
2. **read, write, fork, open, close** – аналогично системным вызовам в лабораторной работе №2.
3. **mmap –** для отображения файла в память, при этом, в качестве аргумента с файловым дескриптором передается -1, и указывается флаг MAP\_ANONYMOUS и MAP\_SHARED, что означает создание отображения без самого файла, используется для связи между процессами.
4. **munmap –** закрыть отображение, ранее созданное вызовом **mmap**, в отличии от вызова **free**, мы должны также передать размер отображения в байтах
5. **sem\_open –** используется для создания именованного семафора – простейшего примитива синхронизации, при совместно создается файл объекта разделяемой памяти в директории /dev/shm.
6. **sem\_close** – закрытие ранее открытого семафора в текущем процессе.
7. **sem\_unlink –** удаление файла семафора из каталога /dev/shm, вызывается в главном процессе перед выходом из програмы.
8. **sem\_wait –** уменьшает значение семафора на 1, если значение не больше нуля, то вызывающий поток выполнения блокируется до момента, когда значение вновь станет положительным.
9. **sem\_post –** увеличивает значение семафора на 1, совместно с **sem\_wait** используется для синхронизации доступа к разделяемой памяти между родительским и дочерним процессом.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы с семафорами pthread и системным вызовом mmap.
2. Написать инициализирующую и финальную функцию программы для корректного создания и освобождения ресурсов – семафоров и разделяемой памяти.
3. Написать функции содержащие логику работы программы по заданному варианту: **parent** и **child**. При этом нужно учитывать возможность ошибок при одновременной попытке чтения/записи в разделяемую часть памяти, и использовать примитивы синхронизации чтобы избежать таких ситуаций.
4. Отладить программу и протестировать на основных тестах.

**Основные файлы программы**

**main.c**

#define \_GNU\_SOURCE

#include <string.h>

#include <stdio.h>

#include <errno.h>

#include <signal.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/syscall.h>

#include <sys/mman.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <semaphore.h>

#define BUFSIZE 256

#define SYSEOF 0

#define FAILURE -1

#define SUCCESS 0

#define TRUE 1

#define FALSE 0

#define NUM\_OF\_CHILDS 2

char\* parent\_name[2] = { "/parent0", "/parent1" };

char\* child\_name[2] = { "/child0", "/child1" };

sem\_t\* parent\_sem[2];

sem\_t\* child\_sem[2];

int out\_files[2];

char filename[2][BUFSIZE];

void\* mmap\_ptrs[2];

int signals[BUFSIZE];

int readString(int *fd*, char\* *str*, int *maxsize*) {

    char ch;

    int rbytes = 0;

    int ret;

    if ((ret = read(*fd*, &ch, 1)) < 1)

        return SYSEOF;

    while ((ch == ' ') || (ch == '\n') || (ch == '\t'))

        if (read(*fd*, &ch, 1) < 1)

            return SYSEOF;

    int i = 0;

    for (; i < *maxsize* - 1; i++) {

*str*[i] = ch;

        rbytes++;

        if (read(*fd*, &ch, 1) < 1)

            return rbytes;

        if (ch == '\r')

            continue;

        if (ch == '\n')

            break;

    }

*str*[++i] = '\0';

    return rbytes;

}

static inline void reverseString(char\* *dst*, char\* *src*, int *size*) {

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

*dst*[*size* - i - 1] = *src*[i];

    return;

}

void open\_sems() {

    for (int i = 0; i < NUM\_OF\_CHILDS; ++i) {

        parent\_sem[i] = sem\_open(parent\_name[i], O\_CREAT, S\_IRUSR | S\_IWUSR, 1);

        if (parent\_sem[i] == SEM\_FAILED) {

            perror("Error: ");

        }

        child\_sem[i] = sem\_open(child\_name[i], O\_CREAT, S\_IRUSR | S\_IWUSR, 0);

        if (child\_sem[i] == SEM\_FAILED) {

            perror("Error: ");

        }

    }

}

void close\_sems() {

    for (int i = 0; i < NUM\_OF\_CHILDS; ++i) {

        if (sem\_close(parent\_sem[i])) {

            perror("Error: ");

        }

        if (sem\_close(child\_sem[i])) {

            perror("Error: ");

        }

    }

}

void unlink\_sems() {

    for (int i = 0; i < NUM\_OF\_CHILDS; ++i) {

        if (sem\_unlink(parent\_name[i]) < 0)

            perror("Error: ");

        if (sem\_unlink(child\_name[i]) < 0)

            perror("Error: ");

        }

    }

}

int parent() {

    char string\_buf[BUFSIZE];

    int size;

    int idx = 0;

    while ((size = readString(STDIN\_FILENO, string\_buf, BUFSIZE)) != SYSEOF) {

        sem\_wait(parent\_sem[idx]);

        memcpy((char\*)(mmap\_ptrs[idx]), (char\*)(&size), sizeof(size));

        memcpy((char\*)((char\*)(mmap\_ptrs[idx]) + sizeof(size)), string\_buf, size);

        sem\_post(child\_sem[idx]);

        idx ^= 1;

    }

    size = SYSEOF;

    sem\_wait(parent\_sem[0]);

    memcpy((char\*)(mmap\_ptrs[0]), (char\*)(&size), sizeof(size));

    sem\_post(child\_sem[0]);

    sem\_wait(parent\_sem[1]);

    memcpy((char\*)(mmap\_ptrs[1]), (char\*)(&size), sizeof(size));

    sem\_post(child\_sem[1]);

    return SUCCESS;

}

int child(int *id*) {

    char buffer[BUFSIZE];

    char reversed[BUFSIZE];

    char newline[] = { '\n' };

    char\* input = mmap\_ptrs[*id*];

    int output = out\_files[*id*];

    int size = 0;

    while (TRUE) {

        sem\_wait(child\_sem[*id*]);

        memcpy(&size, input, sizeof(size));

        if (size == SYSEOF) {

            sem\_post(parent\_sem[*id*]);

            break;

        }

        memcpy(buffer, input + sizeof(size), size);

        reverseString(reversed, buffer, size);

        if (write(output, reversed, size) != size) {

            return FAILURE;

        }

        if (write(output, newline, sizeof(newline)) != sizeof(newline)) {

            return FAILURE;

        }

        sem\_post(parent\_sem[*id*]);

    return SUCCESS;

}

int work() {

    readString(STDIN\_FILENO, filename[0], BUFSIZE);

    readString(STDIN\_FILENO, filename[1], BUFSIZE);

    open\_sems();

    int is\_parent = FALSE;

    int ret;

    int pid[2];

    for (int i = 0; i < NUM\_OF\_CHILDS; ++i) {

        pid[i] = fork();

        if (pid[i] < 0) {

            perror("Error: ");

            return FAILURE;

        }

        else if (pid[i] == 0) {

            out\_files[i] = open(filename[i], O\_CREAT | O\_WRONLY | O\_TRUNC, S\_IRUSR|S\_IWUSR);

            if (out\_files[i] == -1) {

                perror("Error: ");

            } else {

                printf("CHILD: %d HELLO!\n", getpid());

                ret = child(i);

                close(out\_files[i]);

                printf("CHILD: %d GOODBYE!\n", getpid());

            }

            break;

        }

        else if (i == NUM\_OF\_CHILDS - 1) {

            is\_parent = TRUE;

            printf("PARENT: %d HELLO!\n", getpid());

            ret = parent();

            int status;

            waitpid(pid[0], &status, 0);

            waitpid(pid[1], &status, 0);

            printf("PARENT: %d GOODBYE!\n", getpid());

            break;

        }

    }

    close\_sems();

    if (is\_parent) {

        unlink\_sems();

    }

    return ret;

}

int main() {

    mmap\_ptrs[0] = mmap(NULL,BUFSIZE,PROT\_READ|PROT\_WRITE, MAP\_SHARED|MAP\_ANONYMOUS, -1, 0);

    mmap\_ptrs[1] = mmap(NULL,BUFSIZE,PROT\_READ|PROT\_WRITE, MAP\_SHARED|MAP\_ANONYMOUS, -1, 0);

    if (mmap\_ptrs[0] == MAP\_FAILED) {

        perror("Error: ");

        return FAILURE;

    }

    if (mmap\_ptrs[1] == MAP\_FAILED) {

        perror("Error: ");

        if (munmap(mmap\_ptrs[0], BUFSIZE)) {

            perror("Error: ");

        }

        return FAILURE;

    }

    int ret = work();

    if (munmap(mmap\_ptrs[0], 32)) {

        perror("Error: ");

}

    if (munmap(mmap\_ptrs[1], 65536)) {

        perror("Error: ");

}

    return ret;

}

**Пример работы**

$ make

gcc -Werror -pedantic main.c -o main -lpthread

$ ./main

f1

f2

PARENT: 544 HELLO!

CHILD: 545 HELLO!

CHILD: 546 HELLO!

test1: hello world

test2: goodbye!

test3: asasdasdasdasdsdasda asdasd asd asda

CHILD: 545 GOODBYE!

CHILD: 546 GOODBYE!

PARENT: 544 GOODBYE!

$ cat ./f1

dlrow olleh :1tset

adsa dsa dsadsa adsadsdsadsadsadsasa :3tset

$ cat ./f2

!eybdoog :2tset

$ ./main

$ ./main < testfile05

PARENT: 558 HELLO!

CHILD: 559 HELLO!

CHILD: 560 HELLO!

CHILD: 560 GOODBYE!

CHILD: 559 GOODBYE!

PARENT: 558 GOODBYE!

$ cat f1

lilhhljhkjhkhkhgjgjfgh fghfd ngf hf fhfd hd dhfdh gmvngfd cfjgjgcfjvkjgtfkjh.hhgykhg,hjgfmghdshgjfrdhdhesfdhrdjhgm,hgjgmcfhfdgdxbxvx

,ghmgfd

$ cat f2

q

$ cat testfile05

**Репозиторий**

https://github.com/YurokSov/os\_lab\_4

**Вывод**

Выполнив эту лабораторную работу, я укрепил свои знания системных вызовов unix, узнал о таком способе межпроцессного взаимодействия, как разделяемая память, и применил свои навыки программирования для написания кода с использованием семафоров (примитивов синхронизации), для предотвращения ошибок одновременного доступа нескольких процессов к одному блоку данных