Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

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

Студент: Мариничев И. А.

Группа: М8O-208Б-19

Преподаватель: Миронов Е. С.

Дата: 16.12.2020

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2020

1. **Постановка задачи. Вариант 21**

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 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.

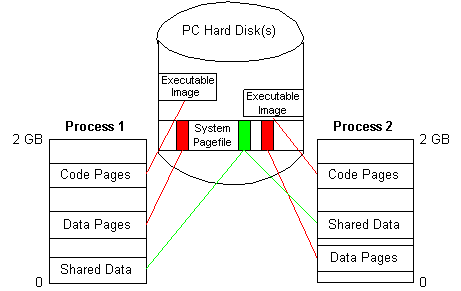
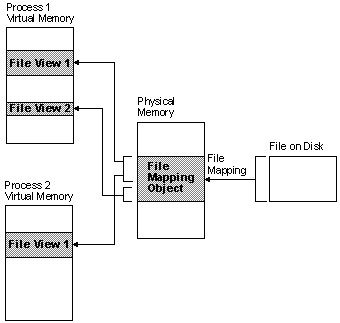
Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в отображаемый файл1 или в отображаемый файл2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод. Родительский и дочерний процесс должны быть представлены разными программами.

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

1. **Общие сведения о программе**

Программа написана на языке Си в UNIX-подобной операционной системе (Ubuntu). В программе создается два дочерних процесса child1 и child2.

Имена отображаемых файлов известны в заранее. Отображение файлов выполняется при запуске как родительского, так и дочерних процессов.



Программа принимает на вход неограниченное количество строк произвольной длины.

Один из двух дочерних процессов выполняет инверсию данной строки и выводит её на экран. Программа для дочерних процессов запускается при помощи функции execv().

Программа завершает свою работу при нажатии Ctrl+D.

Программа обрабатывает все возможные системные ошибки и выводит соответствующие сообщения в случае их возникновения.

1. **Общий метод и алгоритм решения**

При запуске программы пользователю предлагается ввести имя файла для первого и для второго дочернего процесса. В эти файлы будет записываться вывод соответствующих процессов. Если пользователь ввёл имя несуществующего файла, он будет создан.

После запуска программы выполняется отображение двух файлов, имена которых известны заранее. Так как операционная система не позволяет выполнить отображение пустого файла, то перед отображением в файлы записываются «пустые» строки. В качестве «пустой» строки используется строка, состоящая из одного системного символа.

Затем создаются два дочерних процесса. Родительский процесс считывает строки с консольного ввода при помощи функции GetString(). Данная функция считывает строку произвольной длины из стандартного ввода. Затем при помощи четности/нечетности счетчика counter определяется дочерний процесс, которому отправится эта строка на обработку. Счетчик изначально равен 1 (по номеру первой строки), соответственно и первая строка будет оправлена в первый дочерний процесс, затем счетчик инкрементируется, поэтому следующая строка будет отпралена во второй, так строка с нечетным счетчиком будет передана первому дочернему процессу, в противном случае – второму.

Передача строки дочерим процессам осуществляется посредством ее копирования в отображенный файл.

Дочерние процессы перенаправляют свой стандартный вывод в созданный файл. Затем они заменяют свой образ памяти и выполняют программу child, в которой они считывают строки и выполняют её инверсию.

В качестве сигнала используется «пустая» строка. Если дочерний процесс считал «пустую» строку, то ему не нужно ничего выполнять. Если же считана другая строка, то её необходимо обработать. После обработки в отображённый файл вновь записывается «пустая» строка.

Инверсия строки производится в процедуре ReverseString(). Она принимает строку и выполняет ее реверс «на месте», используя технику «двух указателей». После обработки новая строка направляется в стандартный вывод.

Если пользователь нажал Ctrl+D, то родительский процесс посылает обоим дочерним процессам сигнал о завершении работы, закрывает все файлы и завершается сам. Отображаемые файлы, использованные для взаимодействия процессов, удаляются.

1. **Основные файлы программы**

**parent.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h> // strlen(), strcmp(), strcpy()

#include <fcntl.h> // open() and O\_XXX flags

#include <unistd.h> // close(), dup2(), execl()

#include <sys/stat.h> // S\_IXXX flags

#include <sys/mman.h> // mmap(), munmap()

#define MAP\_SIZE 4096

// Files for mapping

char\* file1\_name = "file1\_mapped";

char\* file2\_name = "file2\_mapped";

// Empty string as a signal

char isEmpty = 1;

char\* empty\_string = &isEmpty;

// Function to scan a string with unknown length

char\* GetString() {

    int length = 0, capacity = 10;

    char\* s = (char\*)malloc(10 \* sizeof(char));

    if (s == NULL) {

        perror("ERROR: unable to read string.");

        exit(6);

    }

    char ch;

    while ((ch = getchar()) != '\n') {

        s[length++] = ch;

        if (ch == EOF) {

            break;

        }

        if (length == capacity) {

            capacity \*= 2;

            s = (char\*)realloc(s, capacity \* sizeof(char));

            if (s == NULL) {

                perror("ERROR: unable to read string.");

                exit(6);

            }

        }

    };

    s[length] = '\0'; // Null-terminate the completed string

    return s;

}

// Function that reverses user string

void ReverseString(char \*str) {

    int length = strlen(str);

    char \*front = str;

    char \*back = str + length - 1;

    while (front < back) {

        char tmp = \*front;

        \*front = \*back;

        \*back = tmp;

        ++front;

        --back;

    }

}

int main() {

    // Creating files for output of child processes

    printf("Enter file's name for child process 1 (gets odd lines): ");

    char\* output\_file1\_name = GetString();

    printf("Enter file's name for child process 2 (gets even lines): ");

    char\* output\_file2\_name = GetString();

    // Flags for open():

    //O\_WRONLY - write only, O\_CREAT - create file if it doesn't exist,

    //S\_IWRITE, S\_IREAD - give user rights to write and read accordingly,

    int output\_file1 = open(output\_file1\_name, O\_WRONLY | O\_CREAT, S\_IWRITE | S\_IREAD); // System call to open file

    int output\_file2 = open(output\_file2\_name, O\_WRONLY | O\_CREAT, S\_IWRITE | S\_IREAD);

    if (output\_file1 < 0 || output\_file2 < 0) {

        perror("ERROR: unable to open file.");

        exit(1);

    }

    // Сreating files for mapping

    int fd1 = open(file1\_name, O\_RDWR | O\_CREAT, S\_IWRITE | S\_IREAD);

    int fd2 = open(file2\_name, O\_RDWR | O\_CREAT, S\_IWRITE | S\_IREAD);

    if (fd1 < 0 || fd2 < 0) {

        perror("ERROR: unable to open file."); // open() returns '-1'

        exit(1);

    }

    // Empty files can't be mapped, so we'll put our empty\_stirng there

    if (write(fd1, empty\_string, sizeof(empty\_string)) < 0) {

        perror("ERROR: unable to write to file");

        exit(1);

    }

    if (write(fd2, empty\_string, sizeof(empty\_string)) < 0) {

        perror("ERROR: unable to write to file");

        exit(1);

    }

    // Mapping files

    char\* file1 = mmap(NULL, MAP\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd1, 0); // maps file into memory

    char\* file2 = mmap(NULL, MAP\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd2, 0);

    if (file1 == MAP\_FAILED || file2 == MAP\_FAILED) {

        perror("ERROR: unable to map a file");

        exit(2);

    }

    // Creating child processes

    pid\_t ID\_1 = fork(); // System call for creating child processes

    if (ID\_1 < 0) { // fork() returns negative value if error occured

        perror("ERROR: unable to create child process.");

        exit(3);

    }

    if (ID\_1 > 0) { // fork() returns default value for parent process

        pid\_t ID\_2 = fork();

        if (ID\_2 < 0) {

            perror("ERROR: unable to create child process");

            exit(3);

        }

        if (ID\_2 > 0) { // Does everything parent process related

            int counter = 1; // Counter to differ odd and even lines

            while (1) { // To stop input send EOF (for example using 'Ctrl + D' in \*nix terminal)

                char\* s = GetString();

                if (counter % 2 == 0) { // Writes to file1 odd lines

                    counter += 1;

                    strcpy(file2, s);

                    if (s[0] == EOF) {

                        strcpy(file2, s);

                        break;

                    }

                }

                else { // Writes to file2 even lines

                    counter += 1;

                    strcpy(file1, s);

                    if (s[0] == EOF) {

                        strcpy(file1, s);

                        break;

                    }

                }

            }

            // Unmaps files from memory, closes  and removes them. Also checks for errors

            if (munmap(file1, MAP\_SIZE) < 0 || munmap(file2, MAP\_SIZE) < 0) {

                perror("ERROR: unable to unmap files");

                exit(4);

            }

            if (close(fd1) < 0 || close(fd2) < 0) {

                perror("ERROR: unable to close files");

                exit(5);

            }

            if (remove(file1\_name) < 0 || remove(file2\_name) < 0) {

                perror("ERROR: unable to delete files");

                exit(6);

            }

        }

        else { // Does everything child2 process related

            // dup2 () redirects standart input and output for child processes

            if (dup2(output\_file2, STDOUT\_FILENO) < 0) {

                perror("ERROR: unable to redirect stdout for child process.");

                exit(7);

            }

            char\* myargs [] = {"2", NULL};

            execv("child", myargs); // System call to execute  file 'child'

            // It won't go here if 'child' executes

            perror("ERROR: unable to execute child process.");

            exit(8);

        }

    }

    else { // Does everything child1 process related

        // dup2 () redirects standart input and output for child processes

        if (dup2(output\_file1, STDOUT\_FILENO) < 0) {

            perror("ERROR: unable to redirect stdout for child process.");

            exit(7);

        }

        char\* myargs [] = {"1", NULL};

        execv("child", myargs); // System call to execute  file 'child'

        // It won't go here if 'child' executes

        perror("ERROR: unable to execute child process.");

        exit(8);

    }

}

**child.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h> // strlen(), strcmp(), strcpy()

#include <fcntl.h> // open() and O\_XXX flags

#include <sys/mman.h> // mmap(), munmap()

#include <sys/stat.h> // S\_IWRITE, S\_IREAD flags

#define MAP\_SIZE 4096

// Files for mapping

char\* file1\_name = "file1\_mapped";

char\* file2\_name = "file2\_mapped";

// Empty string as a signal

char isEmpty = 1;

char\* empty\_string = &isEmpty;

// Function that reverses user string

void ReverseString(char \*str) {

    int length = strlen(str);

    char \*front = str;

    char \*back = str + length - 1;

    while (front < back) {

        char tmp = \*front;

        \*front = \*back;

        \*back = tmp;

        ++front;

        --back;

    }

}

int main(int argc, char\* argv[]) {

    char\* file\_name;

    if (argv[0][0] == '1') {

        file\_name = file1\_name;

    }

    else if (argv[0][0] == '2') {

        file\_name = file2\_name;

    }

    else {

        perror("Unknown file");

        exit(8);

    }

    // Opening a file for mapping

    int fd = open(file\_name, O\_RDWR | O\_CREAT, S\_IWRITE | S\_IREAD);

    if (fd < 0) {

        perror("ERROR: unable to open file");

        exit(1);

    }

    // Mapping file

    /\*MAP\_SHARED - Share this mapping.  Updates to the mapping are visible to

                   other processes mapping the same region, and (in the case of

                   file-backed mappings) are carried through to the underlying

                   file.\*/

    char\* file = mmap(NULL, MAP\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0); // maps file into memory

    if (file == MAP\_FAILED) {

        perror("ERROR: unable to map a file");

        exit(2);

    }

    while (1) { // To stop input send EOF (for example using 'Ctrl + D' in \*nix terminal)

        // Wainting for a string

        while (strcmp(file, empty\_string) == 0) {}

        // Terminating if 'Ctrl + D' was pressed

        if (file[0] == EOF) {

            if (munmap(file, MAP\_SIZE) < 0) { // unmaps file from memory

                perror("ERROR: unable to unmap file");

                exit(4);

            }

            exit(0);

        }

        char\* string = (char\*)malloc(strlen(file) \* sizeof(char)); // allocates memory for string

        strcpy(string, file);

        ReverseString(string);

        printf("%s\n", string);

        fflush(stdout); // Makes the OS flush any buffers to the underlying file

        strcpy(file, empty\_string);

        free(string); // deallocates memory for string

    }

}

1. **Демонстрация работы программы**

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** gcc child.c -o child

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** gcc parent.c -o parent

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** ./parent

Enter file's name for child process 1 (gets odd lines): test\_file1

Enter file's name for child process 2 (gets even lines): test\_file2

oddline1

evenline1

oddline2

evenline2

oddline3

evenline3

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** cat test\_file1

1enilddo

2enilddo

3enilddo

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** cat test\_file2

1enilneve

2enilneve

3enilneve

1. **Strace**

Утилита, отслеживающая системные вызовы, которые являются механизмом трансляции, обеспечивающим взаимодействие между процессом и операционной системой. Эти вызовы могут быть перехвачены и прочитаны, что позволяет лучше понять, какую задачу процесс пытается выполнить в заданное время. Перехватывая эти вызовы, мы можем добиться лучшего понимания поведения процесов, особенно если что-то пошло не так. Команда ОС, позволяющая отслеживать системны вызовы, называется ptrace. Strace же вызывает ptrace и читает данные о поведении процесса, а затем выводит отчет.

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** strace -o strace\_log.txt ./parent

Enter file's name for child process 1 (gets odd lines): file1

Enter file's name for child process 2 (gets even lines): file2

odd1

even1

odd2

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** cat strace\_log.txt

execve("./parent", ["./parent"], 0x7fffcfa517e0 /\* 19 vars \*/) = 0

brk(NULL) = 0x7fffb8721000

access("/etc/ld.so.nohwcap", F\_OK) = -1 ENOENT (No such file or directory)

access("/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3

fstat(3, {st\_mode=S\_IFREG|0644, st\_size=60037, ...}) = 0

mmap(NULL, 60037, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7fb2cf175000

close(3) = 0

access("/etc/ld.so.nohwcap", F\_OK) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/lib/x86\_64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"..., 832) = 832

fstat(3, {st\_mode=S\_IFREG|0755, st\_size=2030544, ...}) = 0

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fb2cf170000

mmap(NULL, 4131552, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7fb2cea00000

mprotect(0x7fb2cebe7000, 2097152, PROT\_NONE) = 0

mmap(0x7fb2cede7000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1e7000) = 0x7fb2cede7000

mmap(0x7fb2ceded000, 15072, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7fb2ceded000

close(3) = 0

arch\_prctl(ARCH\_SET\_FS, 0x7fb2cf1714c0) = 0

mprotect(0x7fb2cede7000, 16384, PROT\_READ) = 0

mprotect(0x7fb2cf401000, 4096, PROT\_READ) = 0

mprotect(0x7fb2cf027000, 4096, PROT\_READ) = 0

munmap(0x7fb2cf175000, 60037) = 0

fstat(1, {st\_mode=S\_IFCHR|0660, st\_rdev=makedev(4, 8), ...}) = 0

ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0

brk(NULL) = 0x7fffb8721000

brk(0x7fffb8742000) = 0x7fffb8742000

fstat(0, {st\_mode=S\_IFCHR|0660, st\_rdev=makedev(4, 8), ...}) = 0

ioctl(0, TCGETS, {B38400 opost isig icanon echo ...}) = 0

write(1, "Enter file's name for child proc"..., 56) = 56

read(0, "file1\n", 4096) = 6

write(1, "Enter file's name for child proc"..., 57) = 57

read(0, "file2\n", 4096) = 6

openat(AT\_FDCWD, "file1", O\_WRONLY|O\_CREAT, 0600) = 3

openat(AT\_FDCWD, "file2", O\_WRONLY|O\_CREAT, 0600) = 4

openat(AT\_FDCWD, "file1\_mapped", O\_RDWR|O\_CREAT, 0600) = 5

openat(AT\_FDCWD, "file2\_mapped", O\_RDWR|O\_CREAT, 0600) = 6

write(5, "\1\0\0\0\0\0\0\0", 8) = 8

write(6, "\1\0\0\0\0\0\0\0", 8) = 8

**mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 5, 0) = 0x7fb2cf180000**

**mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 6, 0) = 0x7fb2cf160000**

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7fb2cf171790) = 402

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7fb2cf171790) = 403

read(0, "odd1\n", 4096) = 5

read(0, "even1\n", 4096) = 6

read(0, "odd2\n", 4096) = 5

read(0, "", 4096) = 0

munmap(0x7fb2cf180000, 4096) = 0

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=403, si\_uid=1000, si\_status=0, si\_utime=0, si\_stime=0} ---

munmap(0x7fb2cf160000, 4096) = 0

close(5) = 0

close(6) = 0

unlink("file1\_mapped") = 0

unlink("file2\_mapped") = 0

exit\_group(0) = ?

+++ exited with 0 +++

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** strace -e mmap -o strace\_log1.txt ./parent

Enter file's name for child process 1 (gets odd lines): s\_file1

Enter file's name for child process 2 (gets even lines): s\_file2

odd

even

odd

**ivan@Laptop-IM:/mnt/c/Users/Иван/projects/os\_labs/os\_lab4$** cat strace\_log1.txt

mmap(NULL, 60037, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7f9789523000

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f9789520000

mmap(NULL, 4131552, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f9788e00000

mmap(0x7f97891e7000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1e7000) = 0x7f97891e7000

mmap(0x7f97891ed000, 15072, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f97891ed000

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 5, 0) = 0x7f9789530000

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_SHARED, 6, 0) = 0x7f9789510000

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=411, si\_uid=1000, si\_status=0, si\_utime=0, si\_stime=0} ---

+++ exited with 0 +++

1. **Выводы**

В ходе данной лабораторной работе мной был изучен и применён на практике механизм межпроцессорного взаимодействия при помощи отображаемых файлов. Файл отображается на оперативную память, так что мы получаем доступ к его содержимому и можем обращаться с ним как с массивом.

Таким образом, вместо многократного выполнения небыстрых запросов на чтение и запись мы выполняем отображение файла на ОЗУ и получаем произвольный доступ за О(1). Это происходит, потому что нет операций копирования больших блоков данных (которые происходят, к тому же, внутри syscall'ов) и чтение с диска производится по факту необходимости данных, а не заранее, что делает поведение системы более "плавным".

В связи с этим, когда мы используем файлы, отображаемые в память, появляется и ускорение прораммы в несколько раз по сравненеию, например, с механизмом межпроцессорного взаимодействия при помощи каналов.

В качестве недостатка можно выделить тот факт, что дочерние процессы обязательно должны знать имя отображаемого файла и выполнить их отображение перед началом работы.