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: Моисеенков Илья Павлович
Группа: М80 – 208Б-19
Вариант: 22
Преподаватель: Миронов Евгений Сергеевич
Дата: 16.11.2020
Оценка: отлично
Подпись: _____

Москва, 2020

# 1. Постановка задачи

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: с вероятностью 80% строки отправляются первому процессу, иначе отправляются второму процессу. Дочерние процессы инвертируют строки.

# 2. Общие сведения о программе

Программа написана на языке Си в UNIX-подобной операционной системе (Ubuntu). В программе создается два дочерних процесса child1 и child2. Каждый дочерний процесс связан с родительским при помощи отдельного канала pipe.

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.

# 3. 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 get_string(). Данная функция считывает строку произвольной длины из стандартного ввода. Затем при помощи функции rand() определяется дочерний процесс, которому отправится эта строка на обработку. Если сгенерированное случайное число по модулю 100 не больше 80, то строка будет передана первому дочернему процессу, в противном случае – второму. Таким образом, вероятность попадания строки в первый дочерний процесс составляет 80%.

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 reverse_string. 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.

## 4. Основные файлы программы

### parent.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define MAP_SIZE 4096

// files for mapping
char* file1_name = "file1_mapped";
char* file2_name = "file2_mapped";

// empty string as a signal
char empty = 1;
char* empty_string = &empty;

// scan a string with unknown length
char* get_string() {
    int len = 0, capacity = 10;
    char* s = (char*)malloc(10 * sizeof(char));
    if (s == NULL) {
        perror("Can't read a string");
```

```
        exit(6);
    }

    char c;
    while ((c = getchar()) != '\n') {
        s[len++] = c;
        if (c == EOF) {
            break;
        }
        if (len == capacity) {
            capacity *= 2;
            s = (char*)realloc(s, capacity * sizeof(char));
            if (s == NULL) {
                perror("Can't read a string");
                exit(6);
            }
        }
    };
    s[len] = '\0';
    return s;
}

void reverse_string(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() {
    srand(time(NULL));

    // creating files for output of child processes
    printf("Enter file's name for child process 1: ");
    char* output_file1_name = get_string();

    printf("Enter file's name for child process 2: ");
    char* output_file2_name = get_string();

    int output_file1 = open(output_file1_name, O_WRONLY | O_CREAT, S_IWRITE | S_IREAD)
;
    int output_file2 = open(output_file2_name, O_WRONLY | O_CREAT, S_IWRITE | S_IREAD)
;
    if (output_file1 < 0 || output_file2 < 0) {
        perror("Can't open file");
        exit(1);
    }


    // creating 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("Can't open file");
        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("Can't write to file");
    exit(1);
}
if (write(fd2, empty_string, sizeof(empty_string)) < 0) {
    perror("Can't write to file");
    exit(1);
}

// mapping files
char* file1 = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd1, 0);
char* file2 = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd2, 0);
if (file1 == MAP_FAILED || file2 == MAP_FAILED) {
    perror("Can't map a file");
    exit(2);
}

// creating child processes
pid_t pid1 = fork();
if (pid1 < 0) {
    perror("Can't create child process");
    exit(3);
}

if (pid1 > 0) { // parent
    pid_t pid2 = fork();
    if (pid2 < 0) {
        perror("Can't create child process");
        exit(3);
    }

    if (pid2 > 0) { // parent

        while (1) {
            char* s = get_string();

            if (rand() % 100 + 1 <= 80) {
                strcpy(file1, s);
                if (s[0] == EOF) {
                    strcpy(file2, s);
                    break;
                }
            }
            else {
                strcpy(file2, s);
                if (s[0] == EOF) {
                    strcpy(file1, s);
                    break;
                }
            }
        }
        if (munmap(file1, MAP_SIZE) < 0 || munmap(file2, MAP_SIZE) < 0) {
            perror("Can't unmap files");
            exit(4);
        }
        if (close(fd1) < 0 || close(fd2) < 0) {
            perror("Can't close files");
            exit(5);
        }
        if (remove(file1_name) < 0 || remove(file2_name) < 0) {
            perror("Can't delete files");
            exit(6);
        }
    }
    else { // child2
        // redirecting output
        if (dup2(output_file2, STDOUT_FILENO) < 0) {
```

```
                perror("Can't redirect stdout for child process");
                exit(7);
            }

            char* arr [] = {"2", NULL};
            execv("child", arr);

            // it won't go here if child executes
            perror("Can't execute child process");
            exit(8);
        }
    }
    else { // child1
        // redirecting output
        if (dup2(output_file1, STDOUT_FILENO) < 0) {
            perror("Can't redirect stdout for child process");
            exit(7);
        }

        char* arr [] = {"1", NULL};
        execv("child", arr);

        // it won't go here if child executes
        perror("Can't execute child process");
        exit(8);
    }
}
```

## child.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define MAP_SIZE 4096

// files for mapping
char* file1_name = "file1_mapped";
char* file2_name = "file2_mapped";

// empty string as a signal
char empty = 1;
char* empty_string = &empty;

void reverse_string(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("Can't open file");
        exit(1);
    }

    // mapping file
    char* file = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (file == MAP_FAILED) {
        perror("Can't map a file");
        exit(2);
    }

    while (1) {
        // 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) {
                perror("Can't unmap file");
                exit(4);
            }
            exit(0);
        }

        char* string = (char*)malloc(strlen(file) * sizeof(char));
        strcpy(string, file);
        reverse_string(string);
        printf("%s\n", string);
        fflush(stdout);
        strcpy(file, empty_string);
        free(string);
    }
}
```

## 5. Демонстрация работы программы

```
mosik@LAPTOP-69S778GL:~/os_lab4$ gcc child.c -o child -Wall
mosik@LAPTOP-69S778GL:~/os_lab4$ gcc parent.c -o parent -Wall
mosik@LAPTOP-69S778GL:~/os_lab4$ ./parent
Enter file's name for child process 1: file1
Enter file's name for child process 2: file2
Hey
Here are
some string
for my
program
it should work
enough strings
let's check the result
mosik@LAPTOP-69S778GL:~/os_lab4$ cat file1
era ereH
gnirts emos
ym rof
margorp
```

```
krow dluohs ti
sgnirts hguone
tluser eht kcehc s'tel
mosik@LAPTOP-69S778GL:~/os_lab4$ cat file2
yeH

mosik@LAPTOP-69S778GL:~/os_lab4$ strace -f -e trace="%process,read,write,dup2,mmap" -o
strace_log.txt ./parent
Enter file's name for child process 1: f1
Enter file's name for child process 2: f2
here are
some string
for strace
mosik@LAPTOP-69S778GL:~/os_lab4$ cat f1
era ereh
gnirts emos
mosik@LAPTOP-69S778GL:~/os_lab4$ cat f2
ecarts rof
mosik@LAPTOP-69S778GL:~/os_lab4$ cat strace_log.txt
535   execve("./parent", ["./parent"], 0x7fffee34b5d8 /* 30 vars */) = 0
535   mmap(NULL, 42756, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ffea19c4000
535   read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0\0"...,
832) = 832
535   mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7ffea19c0000
535   mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7ffea1200000
535   mmap(0x7ffea15e7000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7ffea15e7000
535   mmap(0x7ffea15ed000, 15072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ffea15ed000
535   arch_prctl(ARCH_SET_FS, 0x7ffea19c14c0) = 0
535   write(1, "Enter file's name for child proc"..., 39) = 39
535   read(0, "f1\n", 1024)             = 3
535   write(1, "Enter file's name for child proc"..., 39) = 39
535   read(0, "f2\n", 1024)             = 3
535   write(5, "\1\0\0\0\0\0\0\0", 8)   = 8
535   write(6, "\1\0\0\0\0\0\0\0", 8)   = 8
535   mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, 5, 0) = 0x7ffea19b0000
535   mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, 6, 0) = 0x7ffea19a0000
535   clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7ffea19c1790) = 536
536   dup2(3, 1 <unfinished ...>
535   clone( <unfinished ...>
536   <... dup2 resumed> )              = 1
536   execve("child", ["1"], 0x7fffeda13568 /* 30 vars */ <unfinished ...>
535   <... clone resumed> child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7ffea19c1790) =
537
537   dup2(4, 1 <unfinished ...>
535   read(0,  <unfinished ...>
537   <... dup2 resumed> )              = 1
537   execve("child", ["2"], 0x7fffeda13568 /* 30 vars */ <unfinished ...>
536   <... execve resumed> )            = 0
536   mmap(NULL, 42756, PROT_READ, MAP_PRIVATE, 7, 0) = 0x7f82f5b9a000
536   read(7,  <unfinished ...>
537   <... execve resumed> )            = 0
536   <... read resumed>
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0\0"..., 832) = 832
536   mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f82f5b90000
536   mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 7, 0) =
0x7f82f5400000
536   mmap(0x7f82f57e7000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 7, 0x1e7000) = 0x7f82f57e7000
```

```
536   mmap(0x7f82f57ed000, 15072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0 <unfinished ...>
537   mmap(NULL, 42756, PROT_READ, MAP_PRIVATE, 7, 0 <unfinished ...>
536   <... mmap resumed> )              = 0x7f82f57ed000
537   <... mmap resumed> )              = 0x7f781f8c7000
536   arch_prctl(ARCH_SET_FS, 0x7f82f5b914c0) = 0
537   read(7, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0\0"...,
832) = 832
537   mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f781f8c0000
537   mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 7, 0) =
0x7f781f200000
537   mmap(0x7f781f5e7000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 7, 0x1e7000) = 0x7f781f5e7000
537   mmap(0x7f781f5ed000, 15072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f781f5ed000
537   arch_prctl(ARCH_SET_FS, 0x7f781f8c14c0 <unfinished ...>
536   mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, 7, 0 <unfinished ...>
537   <... arch_prctl resumed> )        = 0
536   <... mmap resumed> )              = 0x7f82f5ba0000
537   mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, 7, 0) = 0x7f781f8d0000
535   <... read resumed> "here are\n", 1024) = 9
535   read(0,  <unfinished ...>
536   write(1, "era ereh\n", 9)         = 9
535   <... read resumed> "some string\n", 1024) = 12
535   read(0,  <unfinished ...>
536   write(1, "gnirts emos\n", 12)     = 12
535   <... read resumed> "for strace\n", 1024) = 11
535   read(0,  <unfinished ...>
537   write(1, "ecarts rof\n", 11)      = 11
535   <... read resumed> "", 1024)      = 0
536   exit_group(0 <unfinished ...>
537   exit_group(0 <unfinished ...>
536   <... exit_group resumed>)         = ?
537   <... exit_group resumed>)         = ?
536   +++ exited with 0 +++
537   +++ exited with 0 +++
535   --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=536, si_uid=1000,
si_status=0, si_utime=0, si_stime=0} ---
535   exit_group(0)                     = ?
535   +++ exited with 0 +++
```

## 6. Выводы

Взаимодействие между процессами можно организовать при помощи очереди сообщений, СУБД, а также при помощи низкоуровневых механизмов: каналов, сокетов и отображаемых файлов.

В данной лабораторной работе мной был изучен и применён на практике механизм межпроцессорного взаимодействия при помощи отображаемых файлов (технология «File Mapping»). 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 O(1). По этой причине при использовании технологии «File Mapping» можно добиться ускорения работы программы в несколько раз по сравнению с использованием, например, механизма 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.