Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

Студент: Орусский Вячеслав Русланович

Группа: М8О-206Б-21

Вариант: 20

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2023

Содержание

[Содержание 2](#_Toc133018226)

[**Репозиторий** 3](#_Toc133018227)

[**Цель работы** 3](#_Toc133018228)

[**Задание** 3](#_Toc133018229)

[**Общие сведения о программе** 3](#_Toc133018230)

[**Исходный код** 3](#_Toc133018231)

[Main.cpp 3](#_Toc133018232)

[Slave1.cpp 6](#_Toc133018233)

[Slave2.cpp 7](#_Toc133018234)

[**Демонстрация работы программы** 7](#_Toc133018235)

[**Выводы** 8](#_Toc133018236)

**Репозиторий**

[Ссылка](https://github.com/RoKivals/OperatingSystems/tree/master/Lab%234/src)

**Постановка задачи**

## **Цель работы**

* 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 решения 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. Выполняется тот же вариант, что и в ЛР №2 (Общий алгоритм полностью её повторяет).

**Общие сведения о программе**

Используются следующие системные вызовы:

Create\_memory – выделение памяти в ОЗУ.

Shm\_open – создаёт файл разделяемой памяти (memory-mapped file)

Ftruncate – изменяет размер файла разделяемой памяти

Mmap – отражает заданное кол-во байт в память

**Исходный код**

Main.cpp

#include <bits/stdc++.h>  
#include <unistd.h>  
#include <fcntl.h>  
#include <sys/mman.h>  
#include <signal.h>  
#include "sys/types.h"  
#include "errno.h"  
#include <sys/stat.h>  
#include <semaphore.h>  
*using namespace* std;  
  
*// Имена для разделяемой памяти*string backFile1 = "main1.back";  
string backFile2 = "main2.back";  
  
*/\*  
 \* S\_IWUSR == write access  
 \* S\_IRUSR == read access  
 \* S\_IRGRP == read access for group  
 \* S\_IROTH == read access for all  
 \* \*/  
int* accessPerm = S\_IWUSR | S\_IRUSR | S\_IRGRP | S\_IROTH;  
  
*int* create\_memory(string name, *int* flag, *int* permitions);  
  
*void* wrt\_fd(string &ans1, string &ans2, string name\_file) {  
 *// Открываем поток на чтение* ifstream file1(name\_file);  
 string line;  
 *// Если файл нормально подключился к потокуы  
 if* (file1.is\_open()) {  
 *while* (getline(file1, line)) {  
 line = line + "\n";  
 *int* lineSize = line.length();  
 *if* (lineSize > 10) {  
 ans1 += line;  
 } *else* {  
 ans2 += line;  
 }  
 }  
 }  
 file1.close();  
}  
  
*// Заготовка для создания + проверки fork (не используется)  
int* create\_fork\_and\_check() {  
 *int* result = fork();  
 *if* (result == -1) {  
 std::perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
 *return* result;  
}  
  
*int* main(*int* argc, *char* \*argv[]) {  
 *// Создаём 1-ый дочерний процесс  
 int* id = fork(), id2;  
 *int* flag = 0;  
 *// Если в родительском процессе, то создаём 2-ой дочерний  
 if* (id > 0) {  
 id2 = fork();  
 flag = 1;  
 }  
  
 *if* (id == -1 || id2 == -1) {  
 std::perror("fork");  
 exit(EXIT\_FAILURE);  
 } *else if* (id2 != 0 && id != 0) { *// Если находимся в родителе* string ans1, ans2;  
 wrt\_fd(ans1, ans2, "./file1.txt");  
 wrt\_fd(ans1, ans2, "./file2.txt");  
  
 *// Получаем размеры двух файлов  
 int* mapSize1 = ans1.size(), mapSize2 = ans2.size();  
 *// Создаём объект разделяемой памяти POSIX  
 // O\_RDRW == read-write access; O\_CREAT == create memory object  
 int* fd1 = shm\_open(backFile1.c\_str(), O\_RDWR | O\_CREAT, accessPerm);  
 *// Устанавливаемый для файла (памяти) необходимый размер* ftruncate(fd1, mapSize1);  
  
 *int* fd2 = shm\_open(backFile2.c\_str(), O\_RDWR | O\_CREAT, accessPerm);  
 ftruncate(fd2, mapSize2);  
  
 *// Проверяем ошибки создания памяти  
 if* (fd1 == -1) {  
 perror("shm open 1:");  
 exit(EXIT\_FAILURE);  
 }  
 *if* (fd2 == -1) {  
 perror("shm open 2:");  
 exit(EXIT\_FAILURE);  
 }  
  
  
 *// mmap(start, len, prot, flags, fd, offset=0) - отражает файл в память (ОЗУ)  
 /\* PROT\_READ == may be read  
 \* PROT\_WRITE == may be written  
 \* MAP\_SHARED == let other processes to get info about updating map  
 \* \*/  
 char* \*mapped1 = (*char* \*) mmap(NULL, mapSize1, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd1, 0);  
 *char* \*mapped2 = (*char* \*) mmap(NULL, mapSize2, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd2, 0);  
 *if* (mapped1 == MAP\_FAILED || mapped2 == MAP\_FAILED) {  
 perror("MMAP");  
 exit(EXIT\_FAILURE);  
 }  
  
 *// Заполняем выделенную память* memset(mapped1, '\0', mapSize1);  
 *for* (*int* i = 0; i < mapSize1; ++i) {  
 mapped1[i] = ans1[i];  
 }  
 memset(mapped2, '\0', mapSize2);  
 *for* (*int* i = 0; i < mapSize2; ++i) {  
 mapped2[i] = ans2[i];  
 }  
 } *else if* (flag) { *// Один из потомков  
 // Создаём в нём указатель на тот же раздел памяти  
 int* fd1 = shm\_open(backFile1.c\_str(), O\_RDWR, accessPerm);  
 *if* (fd1 == -1) {  
 perror("shm open 1 by child");  
 exit(EXIT\_FAILURE);  
 }  
  
 *struct* stat statBuf1;  
 fstat(fd1, &statBuf1);  
 *int* mapSize1 = statBuf1.st\_size;  
 *char* \*mapped1 = (*char* \*) mmap(NULL, mapSize1, PROT\_READ, MAP\_SHARED, fd1, 0);  
 *if* (mapped1 == MAP\_FAILED) {  
 perror("mmap1\_child");  
 exit(EXIT\_FAILURE);  
 }  
  
 *// null-vector аргументов  
 char* \*argv1[] = {NULL, NULL};  
 *// Выделяем память для нашей строки (аргумента)* argv1[0] = (*char* \*) malloc(mapSize1);  
 *// Копируем строку в аргумент* strcpy(argv1[0], mapped1);  
 execv("child1", argv1);  
 } *else* {  
  
 *int* fd2 = shm\_open(backFile2.c\_str(), O\_RDWR, accessPerm);  
 *if* (fd2 == -1) {  
 perror("shm open 2 by child");  
 exit(EXIT\_FAILURE);  
 }  
  
 *struct* stat statBuf2;  
 fstat(fd2, &statBuf2);  
 *int* mapSize2 = statBuf2.st\_size; *// полный размер в байтах  
 char* \*mapped2 = (*char* \*) mmap(NULL, mapSize2, PROT\_READ, MAP\_SHARED, fd2, 0);  
 *if* (mapped2 == MAP\_FAILED) {  
 perror("mmap2\_child");  
 exit(EXIT\_FAILURE);  
 }  
  
 TODO:  
 "Переписать под один массив argv";  
 *// char \*argv2[] = {nullptr, nullptr};  
 char* \*argv2[] = {(*char* \*) "child2", *nullptr*, *nullptr*};  
 argv2[0] = (*char* \*) malloc(mapSize2);  
 strcpy(argv2[0], mapped2);  
 execv("child2", argv2);  
 }  
 *return* 0;  
}

Slave1.cpp

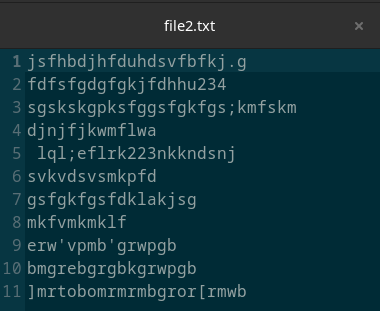
#include <bits/stdc++.h>  
*using namespace* std;  
  
*int* main(*int* argc, *char* \*argv[]) {  
 ofstream fout("./pipe1.txt", ios\_base::out | ios\_base::trunc);  
  
 string mapped1 = argv[0];  
 size\_t mapSize1 = mapped1.length();  
  
 *for* (*int* i(mapSize1 - 2); i >= 0; --i) {  
 fout << mapped1[i];  
 }  
 fout.close();  
}

Slave2.cpp

#include <bits/stdc++.h>  
*using namespace* std;  
  
*int* main(*int* argc, *char* \*argv[]) {  
 ofstream fout("./pipe2.txt", ios\_base::out | ios\_base::trunc);  
  
 string mapped2 = argv[0];  
 size\_t mapSize2 = mapped2.length();  
  
 *for* (*int* i(mapSize2 - 2); i >= 0; --i) {  
 fout << mapped2[i];  
 }  
 fout.close();  
}

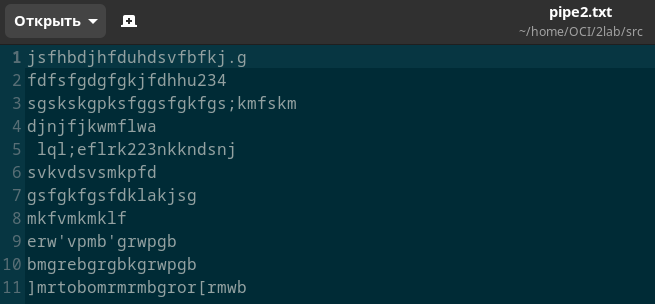
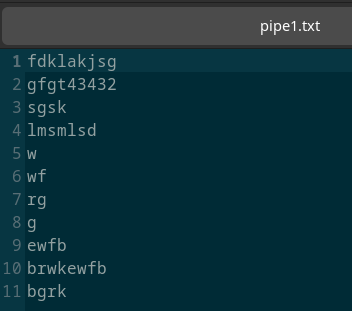
**Демонстрация работы программы**

**Ввод:**





**Вывод:**



**Выводы**

С помощью системных вызовов в ОС Linux с помощью ЯП C++ (Си) можно создавать программы, в которых работает несколько процессов, что крайне положительно может влиять на производительность, а также логику поведения программы. Для общения между процессами используются файлы разделяемой памяти (mmap).