Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

# Лабораторная работа №3 по курсу

# «Операционные системы»

Студент: Ветошкина София Владимировна
Группа: М8О-203Б-23
Вариант: 6
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

**Содержание**

## Репозиторий

https://github.com/sofiavetoshkina/os_labs/tree/main

## Постановка задачи

Составить и отладить программу, осуществляющую работу с процессами и взаимодействие между ними. 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.

## Общие сведения о программе

Родительский процесс создает дочерний процесс. Первой строчкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия файла с таким именем на чтение. Родительский и дочерний процесс представлены разными программами. Обеспечение обмена данных между процессами происходит посредством технологии «File mapping».

В файле записаны команды вида: «число число число<endline>». Дочерний процесс считает их сумму и выводит результат в стандартный поток вывода. Числа имеют тип int. Количество чисел может быть произвольным.

Программа собирается системой сборки CMake.

Реализованы тесты для проверки корректности программы с помощью Google Test.

## Метод и алгоритм решения задачи

Родительский процесс открывает файл на чтение. Затем создает отображаемый файл (shm_open) и отображает содержимое исходного файла в память (mmap). Инициализирует семафор для синхронизации с дочерним процессом. Создает дочерний процесс с помощью fork. Если процесс дочерний: запускает его через execl. Ожидает завершения дочернего процесса через семафор. Читает результат суммы из отображаемой памяти и выводит результат в консоль.

Дочерний процесс подключается к отображаемой памяти, созданной родительским процессом. Парсит числа из памяти, суммирует их. Записывает результат обратно в отображаемую память. Использует семафор для уведомления родительского процесса о завершении.

После завершения работы освобождаются все ресурсы: дескрипторы файлов, отображаемая память, семафор.

Программа проверяется библиотекой Google Test.

## Исходный код

lab3/main.cpp:

```
#include <iostream>
#include <string>
#include <csignal>
#include "parent.hpp"

int main() {
   std::string fileName;
   std::cout << "Введите название файла, где необходимо посчитать сумму
чисел: ";
   std::cin >> fileName;

   try {
      ParentRoutine(fileName, std::cout);
   } catch (const std::exception& e) {
      std::cerr << "Ошибка: " << e.what() << std::endl;
      return 1;
   }

   return 0;
}
```

lab3/include/patrent.hpp:

```
#ifndef OS_LABS_PARENT_H
#define OS_LABS_PARENT_H

#include <iostream>
#include <string>

void ParentRoutine(const std::string& fileName, std::ostream& output);

#endif //OS_LABS_PARENT_H
```

lab3/src/parent.cpp:

```
#include <fcntl.h>
#include <iostream>
#include <sys/mman.h>
#include <sys/stat.h>
```

```
#include <sys/wait.h>
#include <unistd.h>
#include <cstring>
#include <semaphore.h>
#include <string>
#include <stdexcept>
#include "parent.hpp"

void ParentRoutine(const std::string& fileName, std::ostream& output) {
    // Открываем файл на чтение
    int fileFd = open(fileName.c_str(), O_RDONLY);
    if (fileFd < 0) {
        throw std::runtime_error("Не удалось открыть файл");
    }

    struct stat fileStat {};
    if (fstat(fileFd, &fileStat) == -1) {
        close(fileFd);
        throw std::runtime_error("Не удалось получить информацию о файле");
    }

    size_t fileSize = fileStat.st_size;

    if (fileSize == 0) {
        output << 0 << std::endl;
        close(fileFd);
        return;
    }

    // Создаем отображаемый файл
    int shmFd = shm_open("/shared_memory", O_CREAT | O_RDWR, 0666);
    if (shmFd < 0) {
        close(fileFd);
        throw std::runtime_error("Ошибка создания отображаемого файла");
    }

    if (ftruncate(shmFd, fileSize) == -1) {
        close(fileFd);
        close(shmFd);
        throw std::runtime_error("Ошибка выделения памяти в отображаемом
файле");
    }

    // Отображаем файл в память
```

```
    void* mappedMemory = mmap(nullptr, fileSize, PROT_READ |
PROT_WRITE, MAP_SHARED, shmFd, 0);
    if (mappedMemory == MAP_FAILED) {
        close(fileFd);
        close(shmFd);
        throw std::runtime_error("Ошибка отображения файла в память");
    }

    // Читаем данные из файла
    if (read(fileFd, mappedMemory, fileSize) != static_cast<ssize_t>(fileSize)) {
        close(fileFd);
        close(shmFd);
        munmap(mappedMemory, fileSize);
        throw std::runtime_error("Ошибка чтения данных из файла");
    }

    close(fileFd);

    // Создаем и открываем семафор
    sem_t* sem = sem_open("/semaphore", O_CREAT, 0666, 0);
    if (sem == SEM_FAILED) {
        close(shmFd);
        munmap(mappedMemory, fileSize);
        throw std::runtime_error("Ошибка создания семафора");
    }

    // Создаем дочерний процесс
    pid_t pid = fork();
    if (pid < 0) {
        close(shmFd);
        munmap(mappedMemory, fileSize);
        sem_unlink("/semaphore");
        throw std::runtime_error("Ошибка создания дочернего процесса");
    }

    if (pid == 0) {
        // Дочерний процесс
        const char* pathToChild = getenv("PATH_TO_EXEC_CHILD");
        if (pathToChild == nullptr) {
            perror("Переменная PATH_TO_EXEC_CHILD не установлена");
            exit(1);
        }
        execl(pathToChild, pathToChild, std::to_string(fileSize).c_str(), nullptr);
        perror("Ошибка запуска дочернего процесса");
        exit(1);
```

```
    }

    // Ожидаем завершения дочернего процесса
    sem_wait(sem);

    // Читаем результат из памяти
    long int result = 0;
    memcpy(&result, mappedMemory, sizeof(result));

    output << result << std::endl;

    // Очистка
    close(shmFd);
    munmap(mappedMemory, fileSize);
    shm_unlink("/shared_memory");
    sem_unlink("/semaphore");
}
```

lab3/src/child.cpp:

```
#include <fcntl.h>
#include <iostream>
#include <sys/mman.h>
#include <unistd.h>
#include <cstring>
#include <semaphore.h>

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "Неверное количество аргументов" << std::endl;
        return 1;
    }

    size_t fileSize = std::stoul(argv[1]);

    int shmFd = shm_open("/shared_memory", O_RDWR, 0666);
    if (shmFd < 0) {
        perror("Ошибка открытия отображаемого файла");
        return 1;
    }

    void* mappedMemory = mmap(nullptr, fileSize, PROT_READ |
PROT_WRITE, MAP_SHARED, shmFd, 0);
    if (mappedMemory == MAP_FAILED) {
        close(shmFd);
```

```
        perror("Ошибка отображения файла в память");
        return 1;
    }

    char* data = static_cast<char*>(mappedMemory);
    long int sum = 0;
    char* endPtr = data + fileSize;

    while (data < endPtr) {
        long int number = std::strtol(data, &data, 10);
        sum += number;
    }

    // Записываем результат в начало памяти
    memcpy(mappedMemory, &sum, sizeof(sum));

    sem_t* sem = sem_open("/semaphore", 0);
    if (sem == SEM_FAILED) {
        close(shmFd);
        munmap(mappedMemory, fileSize);
        perror("Ошибка sem_open в дочернем процессе");
        return 1;
    }
    sem_post(sem);

    // Очистка
    close(shmFd);
    munmap(mappedMemory, fileSize);

    return 0;
}
```

tests/lab3_test.cpp:

```
#include <gtest/gtest.h>
#include <fstream>
#include <string>
#include "parent.hpp"

TEST(ParentRoutineTest, CalculatesSumCorrectly) {
    std::ostringstream outputStream;
    const char* fileName = getenv("PATH_TO_TEST_FILE");
    if (fileName == nullptr) {
        perror("Переменная PATH_TO_TEST_FILE не установлена");
        exit(1);
```

```
    }

    //Содержимое test.txt:
    //100 10 50
    //40 -10 10

    const int expectedOutput = 200;

    ParentRoutine(fileName, outputStream);

    std::string output = outputStream.str();
    std::istringstream iss(output);
    int realOutput = 0;
    iss >> realOutput;

    EXPECT_EQ(realOutput, expectedOutput);
}

TEST(ParentRoutineTest, EmptyFile) {
    std::ostringstream outputStream;
    const char* fileName = getenv("PATH_TO_EMPTY_TEST_FILE");
    if (fileName == nullptr) {
        perror("Переменная PATH_TO_EMPTY_TEST_FILE не установлена");
        exit(1);
    }

    const int expectedOutput = 0;

    ParentRoutine(fileName, outputStream);

    std::string output = outputStream.str();
    std::istringstream iss(output);
    int realOutput = 0;
    iss >> realOutput;

    EXPECT_EQ(realOutput, expectedOutput);
}

TEST(ParentRoutineTest, CalculatesSumCorrectly2) {
    std::ostringstream outputStream;
    const char* fileName = getenv("PATH_TO_TEST_FILE2");
    if (fileName == nullptr) {
        perror("Переменная PATH_TO_TEST_FILE2 не установлена");
        exit(1);
    }
```

```
    //Содержимое test2.txt:
    //100 10 50
    //40 -10 10 100000 0

    const int expectedOutput = 100200;

    ParentRoutine(fileName, outputStream);

    std::string output = outputStream.str();
    std::istringstream iss(output);
    int realOutput = 0;
    iss >> realOutput;

    EXPECT_EQ(realOutput, expectedOutput);
}
```

**Демонстрация работы программы**

Содержимое файла test.txt: 100 10 50\n40 -10 10

getz66@getz1165-nettop:~/OS/os_labs/build$ export PATH_TO_EXEC_CHILD='/home/getz66/OS/os_labs/build/lab3/child1'

getz66@getz1165-nettop:~/OS/os_labs/build$ ./lab3/parent1

Введите название файла, где необходимо посчитать сумму чисел: test.txt

200

Тестирование Google Test также выполняется успешно.

**Выводы**

Memory-mapped files — это механизм, позволяющий отобразить содержимое файла напрямую в адресное пространство процесса. Это позволяет работать с файлами как с обычной оперативной памятью, без необходимости явного чтения и записи данных, что существенно упрощает и ускоряет работу с большими файлами. Такой подход особенно удобен для обработки больших объемов данных, совместного доступа нескольких процессов к одному файлу, а также для реализации эффективных кэшей. Memory-mapped files — это мощный инструмент, который объединяет удобство использования и высокую производительность. Memory-mapped files в C++ можно использовать с помощью функций операционной системы.