Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу
«Операционные системы»

Тема работы
“Межпроцессорное взаимодействие через memory-mapped files”

Студент: Слободин Никита Алексеевич
Группа: М8О-203Б-23
Вариант: 21

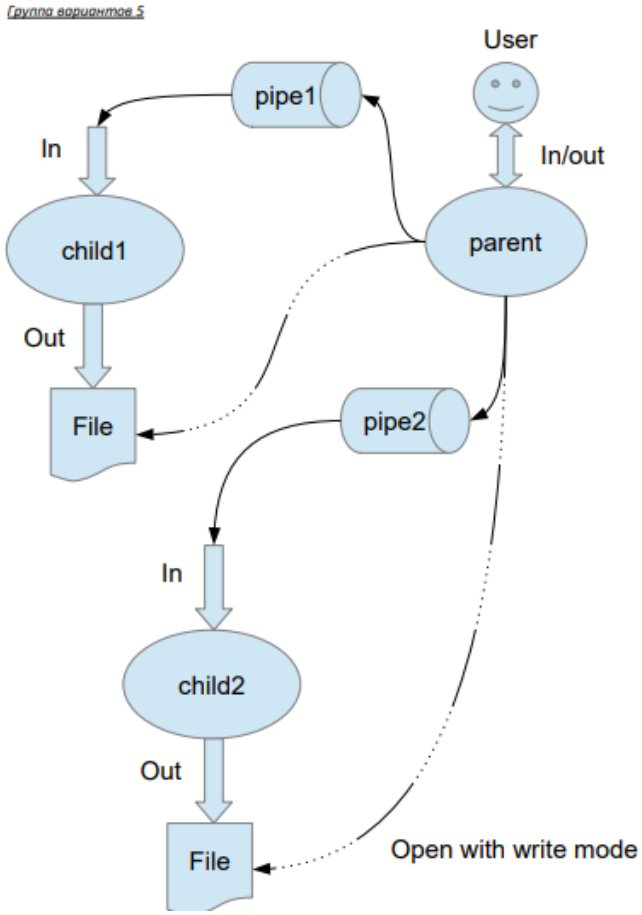
Преподаватель: Миронов Евгений Сергеевич

Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

Постановка задачи

Задача: реализовать программу, в которой родительский процесс создает два дочерних процесса. Межпроцессорное взаимодействие осуществляется посредством отображаемых файлов (memory-mapped files).



Вариант 21) Правило фильтрации: нечетные строки отправляются в pipe1, четные в pipe2. Дочерние процессы инвертируют строки.

Общие сведения

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 первый процесс или во второй процесс в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Описание программы:

Родительский процесс создает разделяемую область памяти и инициирует именованные семафоры, обеспечивающие синхронизацию процессов. После этого пользователь вводит два имени файлов, куда дочерние процессы будут записывать результаты обработки.

Родительский процесс передает вводимые строки в разделяемую память. В зависимости от длины строки управление передается одному из дочерних процессов. Дочерние процессы получают строки, обрабатывают их, удаляя гласные буквы, и записывают результат в соответствующий файл.

После завершения обработки родительский процесс передает сигнал завершения дочерним процессам, а затем очищает созданные ресурсы, включая разделяемую память и семафоры. Эта программа демонстрирует эффективное использование системных вызовов для межпроцессного взаимодействия и управления потоками данных.

Исходный код программы представлен в приложении.

Вывод

В рамках данной лабораторной работы я освоил методы взаимодействия с дочерними процессами, включая создание, управление и синхронизацию процессов. Особое внимание было уделено использованию сигналов для координации действий между процессами и обеспечения их корректного взаимодействия. Кроме того, я изучил способы работы с файлами, отображёнными в оперативную память (mmap), что позволило углубить понимание эффективного обмена данными между процессами. Эти навыки

укрепили моё понимание механизмов межпроцессного взаимодействия и работы с системными ресурсами.

Приложение

src/child1.cpp

```
#include "utils.h"
#include <atomic>
#include <signal.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <cstdio>
#include <cstdlib>


#define SHM_SIZE 4096

std::atomic<bool> dataReady(false);
std::atomic<bool> terminateFlag(false);

// Обработчики сигналов
void dataSignalHandler(int signum) {
    (void)signum;
    dataReady.store(true, std::memory_order_relaxed);
}

void terminateSignalHandler(int signum) {
    (void)signum;
    terminateFlag.store(true, std::memory_order_relaxed);
}

int main(int argc, char** argv) {
    if (argc != 3) {
        fprintf(stderr, "Child1: Required arguments are shared memory name and output
file name.\n");
        return 1;
    }

    const char* shmName = argv[1];
    const char* outputFileName = argv[2];

    // Открытие и маппинг разделяемой памяти
    int shmFd;
    char* shmPtr = CreateAndMapSharedMemory(shmName, shmFd, SHM_SIZE);

    // Регистрация обработчиков сигналов
    RegisterSignalHandler(SIGUSR1, dataSignalHandler);
    RegisterSignalHandler(SIGTERM, terminateSignalHandler);

    // Установка флага готовности
    shmPtr[0] = 1;

    // Открытие файла для вывода
    FILE* outputFile = fopen(outputFileName, "w");
    if (!outputFile) {
```

```
        perror("Failed to open output file");
        munmap(shmPtr, SHM_SIZE);
        close(shmFd);
        exit(EXIT_FAILURE);
    }

    // Основной цикл обработки
    while (!terminateFlag.load(std::memory_order_relaxed)) {
        if (dataReady.load(std::memory_order_relaxed)) {
            dataReady.store(false, std::memory_order_relaxed);

            std::string input(shmPtr + 2); // Пропуск флагов

            if (input.empty()) {
                break;
            }

            std::string modified = Modify(input);

            fprintf(outputFile, "%s", modified.c_str());

            // Установка флага завершения обработки
            shmPtr[1] = 1;
        } else {
            usleep(1000); // Небольшая задержка
        }
    }

    fclose(outputFile);
    munmap(shmPtr, SHM_SIZE);
    close(shmFd);

    return 0;
}
```

src/child2.cpp

```
#include "utils.h"
#include <atomic>
#include <signal.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <cstdio>
#include <cstdlib>

#define SHM_SIZE 4096

std::atomic<bool> dataReady(false);
std::atomic<bool> terminateFlag(false);

// Обработчики сигналов
void dataSignalHandler(int signum) {
```

```
    (void)signum;
    dataReady.store(true, std::memory_order_relaxed);
}

void terminateSignalHandler(int signum) {
    (void)signum;
    terminateFlag.store(true, std::memory_order_relaxed);
}

int main(int argc, char** argv) {
    if (argc != 3) {
        fprintf(stderr, "Child2: Required arguments are shared memory name and output
file name.\n");
        return 1;
    }

    const char* shmName = argv[1];
    const char* outputFileName = argv[2];

    // Открытие и маппинг разделяемой памяти
    int shmFd;
    char* shmPtr = CreateAndMapSharedMemory(shmName, shmFd, SHM_SIZE);

    // Регистрация обработчиков сигналов
    RegisterSignalHandler(SIGUSR2, dataSignalHandler);
    RegisterSignalHandler(SIGTERM, terminateSignalHandler);

    // Установка флага готовности
    shmPtr[0] = 1;

    // Открытие файла для вывода
    FILE* outputFile = fopen(outputFileName, "w");
    if (!outputFile) {
        perror("Failed to open output file");
        munmap(shmPtr, SHM_SIZE);
        close(shmFd);
        exit(EXIT_FAILURE);
    }

    // Основной цикл обработки
    while (!terminateFlag.load(std::memory_order_relaxed)) {
        if (dataReady.load(std::memory_order_relaxed)) {
            dataReady.store(false, std::memory_order_relaxed);

            std::string input(shmPtr + 2); // Пропуск флагов

            if (input.empty()) {
                break;
            }

            std::string modified = Modify(input);

            fprintf(outputFile, "%s", modified.c_str());
```

```
            // Установка флага завершения обработки
            shmPtr[1] = 1;
        } else {
            usleep(1000); // Небольшая задержка
        }
    }

    fclose(outputFile);
    munmap(shmPtr, SHM_SIZE);
    close(shmFd);

    return 0;
}
```

src/parent.cpp

```
#include "parent.h"
#include "utils.h"
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sstream>
#include <cstring>
#include <csignal>
#include <cstdlib>

// Размер разделяемой памяти
#define SHM_SIZE 4096

// Функция для создания и маппинга разделяемой памяти
char* CreateAndMapSharedMemory(const std::string& shmName, int& shmFd) {
    shmFd = shm_open(shmName.c_str(), O_CREAT | O_RDWR | O_EXCL, 0666);
    if (shmFd == -1) {
        perror("shm_open failed");
        exit(EXIT_FAILURE);
    }

    if (ftruncate(shmFd, SHM_SIZE) == -1) {
        perror("ftruncate failed");
        shm_unlink(shmName.c_str());
        exit(EXIT_FAILURE);
    }

    char* shmPtr = static_cast<char*>(mmap(nullptr, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shmFd, 0));
    if (shmPtr == MAP_FAILED) {
        perror("mmap failed");
        shm_unlink(shmName.c_str());
        exit(EXIT_FAILURE);
    }
```

```
    // Инициализация флагов готовности и завершения обработки
    shmPtr[0] = 0; // Флаг готовности
    shmPtr[1] = 0; // Флаг завершения обработки

    return shmPtr;
}

pid_t LaunchChildProcess(const char* childPath, const std::string& shmName, const
char* outputFileName) {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Дочерний процесс
        if (execl(childPath, childPath, shmName.c_str(), outputFileName, nullptr) == -
1) {
            perror("execl failed");
            exit(EXIT_FAILURE);
        }
    }

    return pid;
}

void ParentRoutine(const char* pathToChild1, const char* pathToChild2, std::istream&
input) {
    char filename1[256];
    char filename2[256];

    std::ostringstream oss1, oss2;
    static int counter = 0;
    oss1 << "/test_shm_" << getpid() << "_" << counter++ << "_1";
    oss2 << "/test_shm_" << getpid() << "_" << counter++ << "_2";
    std::string shmName1 = oss1.str();
    std::string shmName2 = oss2.str();

    // Игнорирование сигналов SIGUSR1 и SIGUSR2 в родительском процессе
    struct sigaction sa = {};
    sa.sa_handler = SIG_IGN;
    sigaction(SIGUSR1, &sa, nullptr);
    sigaction(SIGUSR2, &sa, nullptr);

    // Ввод имен файлов от пользователя
    std::cout << "Enter filename for 1 process: ";
    input.getline(filename1, 256);
    std::cout << "Enter filename for 2 process: ";
    input.getline(filename2, 256);

    // Создание и маппинг разделяемой памяти для child1
```

```
    int shmFd1;
    char* shmPtr1 = CreateAndMapSharedMemory(shmName1, shmFd1);

    // Создание и маппинг разделяемой памяти для child2
    int shmFd2;
    char* shmPtr2 = CreateAndMapSharedMemory(shmName2, shmFd2);

    // Запуск дочерних процессов
    pid_t child1_pid = LaunchChildProcess(pathToChild1, shmName1, filename1);
    pid_t child2_pid = LaunchChildProcess(pathToChild2, shmName2, filename2);

    // Ожидание готовности дочерних процессов
    while (shmPtr1[0] != 1 || shmPtr2[0] != 1) {
        usleep(1000); // 1 мс
    }

    size_t lineNumber = 1;
    ReadData([shmPtr1, shmPtr2, &lineNumber, child1_pid, child2_pid](const
std::string& str) {
        if (lineNumber % 2 == 1) {
            // Запись данных в shmPtr1 + 2
            strncpy(shmPtr1 + 2, str.c_str(), SHM_SIZE - 3); // +2 для флагов
            shmPtr1[1] = 0; // Сброс флага завершения обработки
            shmPtr1[SHM_SIZE -1] = '\0'; // Обеспечение нулевого завершения строки
            kill(child1_pid, SIGUSR1);
            // Ожидание завершения обработки дочерним процессом
            while (shmPtr1[1] != 1) {
                usleep(1000);
            }
        } else {
            // Запись данных в shmPtr2 + 2
            strncpy(shmPtr2 + 2, str.c_str(), SHM_SIZE - 3); // +2 для флагов
            shmPtr2[1] = 0; // Сброс флага завершения обработки
            shmPtr2[SHM_SIZE -1] = '\0'; // Обеспечение нулевого завершения строки
            kill(child2_pid, SIGUSR2);
            // Ожидание завершения обработки дочерним процессом
            while (shmPtr2[1] != 1) {
                usleep(1000);
            }
        }
        lineNumber++;
    }, input);

    // Отправка сигналов завершения дочерним процессам
    kill(child1_pid, SIGTERM);
    kill(child2_pid, SIGTERM);

    // Ожидание завершения дочерних процессов
    waitpid(child1_pid, nullptr, 0);
    waitpid(child2_pid, nullptr, 0);

    // Очистка ресурсов
    munmap(shmPtr1, SHM_SIZE);
```

```
    munmap(shmPtr2, SHM_SIZE);
    close(shmFd1);
    close(shmFd2);
    shm_unlink(shmName1.c_str());
    shm_unlink(shmName2.c_str());
}
```

src/utils.cpp

```
#include "../include/utils.h"
#include <cstring>
#include <cstdio>
#include <algorithm>
#include <cstdlib>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <istream>

void ReadData(const std::function<void(const std::string&)>& handler, std::istream&
stream) {
    std::string buff;
    while (std::getline(stream, buff)) {
        if (buff.empty()) return;
        handler(buff + '\n');
    }
}

std::string Modify(const std::string& str) {
    std::string result;
    if (!str.empty() && str.back() == '\n') {
        result = str.substr(0, str.size() - 1);
        std::reverse(result.begin(), result.end());
        result += '\n';
    } else {
        result = str;
        std::reverse(result.begin(), result.end());
    }
    return result;
}

// Маппинг разделяемой памяти
char* CreateAndMapSharedMemory(const char* shmName, int& shmFd, size_t size) {
    shmFd = shm_open(shmName, O_RDWR, 0666);
    if (shmFd == -1) {
        perror("shm_open failed");
        exit(EXIT_FAILURE);
    }

    char* shmPtr = static_cast<char*>(mmap(nullptr, size, PROT_READ | PROT_WRITE,
MAP_SHARED, shmFd, 0));
```

```
    if (shmPtr == MAP_FAILED) {
        perror("mmap failed");
        close(shmFd);
        exit(EXIT_FAILURE);
    }

    return shmPtr;
}

// Регистрация обработчиков сигналов
void RegisterSignalHandler(int signum, void (*handler)(int)) {
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(signum, &sa, nullptr) == -1) {
        perror("Failed to register signal handler");
        exit(EXIT_FAILURE);
    }
}
```

Пример вывода:

```
root@c34508d80232:/workspaces/OS_MAI_Slobodin/build#
./lab3/lr3parent

Enter filename for 1 process:
/workspaces/OS_MAI_Slobodin/file1.txt

Enter filename for 2 process:
/workspaces/OS_MAI_Slobodin/file2.txt

abc

123

def

567


root@c34508d80232:/workspaces/OS_MAI_Slobodin/build#
```