Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №3 по курсу

«Операционные системы»

Группа: М8О-210БВ-24

Студент: Дмитренко Я.С.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 2.11.25

Москва, 2025

Постановка задачи

Вариант 4.

Пользователь вводит команды вида: «число число число<endline>». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс производит деление первого числа, на последующие, а результат выводит в файл. Если происходит деление на 0, то тогда дочерний и родительский процесс завершают свою работу. Проверка деления на 0 должна осуществляться на стороне дочернего процесса. Числа имеют тип float. Количество чисел может быть произвольным.

Общий метод и алгоритм решения

Использованные системные вызовы:

- pid_t fork(void); – создает дочерний процесс.
- int open(const char *pathname, int flag, mode_t mode) - Открывает файл, возвращает файловый дескриптор
- int close(int fd) - закрываает файловый дескриптор
- int execv(const char *path, char *const argv[]) - заменяет текущий процесс новым. Возвращает -1 при ошибке
- pid_t getpid(void) - возвращает pid текущего процесса
- int shm_open(const char *name, int oflag, mode_t mode); - создаёт или открывает именованный участок разделяемой памяти (реализуется как файл в /dev/shm)
- int shm_unlink(const char *name); - удаляет именованный объект разделяемой памяти
- int ftruncate(int fd, off_t length); - изменяет размер файла или разделяемой памяти до указанной длины
- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset); - отображает файл (или shared memory) в адресное пространство процесса
- int munmap(void *addr, size_t length); - снимает отображение памяти, созданное через mmap
- sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value); - создаёт или открывает именованный семафор
- int sem_close(sem_t *sem); - закрывает именованный семафор
- int sem_unlink(const char *name); - удаляет именованный семафор из системы
- int sem_wait(sem_t *sem); - блокирует процесс, уменьшая значение семафора на 1 (ожидание доступа)
- int sem_post(sem_t *sem); - увеличивает значение семафора на 1, разблокируя ожидающий процесс

Сервер две области разделяемой памяти (shared memory) и семафоры, затем запускает клиента. Серве р записывает введённые числа в память и подаёт сигнал клиенту. Клиент читает данные, выполняет д еление и записывает результат обратно. Сервер получает ответ и выводит результат.При делении на ноль оба процесса завершаются и очищают ресурсы

Код программы

client.c

```
//gcc -o server server.c -lrt -pthread
//gcc -o client client.c -lrt -pthread -lm
//./server results.txt
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/fcntl.h>
#include <sys/mman.h>
#include <semaphore.h>

#define SHM_SIZE 4096

void write_to_file(int file, const char* message) {
    size_t len = strlen(message);
    if (write(file, message, len) == -1) {
        const char msg[] = "Ошибка записи в файл\n";
        write(STDERR_FILENO, msg, sizeof(msg));
    }
}

int process_numbers(int file, const char* input, char* response_buf, size_t response_size)
{
    char buffer[4096];
    strncpy(buffer, input, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';

    float numbers[100];
    int count = 0;

    char* token = strtok(buffer, " ");
    while (token != NULL && count < 100) {
        numbers[count++] = atof(token);
        token = strtok(NULL, " ");
    }

    if (count < 2) {
        write_to_file(file, "Ошибка: нужно минимум 2 числа\n");
        snprintf(response_buf, response_size, "ERROR: Need at least 2 numbers\n");
        return 0;
    }

    char result_str[256];
    float first = numbers[0];

    snprintf(result_str, sizeof(result_str), "Делим %.2f на:", first);
    write_to_file(file, result_str);
    write_to_file(file, "\n");

    char temp_response[1024] = "";

    for (int i = 1; i < count; i++) {
        if (fabs(numbers[i]) < 1e-6) {
```

```
            snprintf(result_str, sizeof(result_str), "  ОШИБКА: деление на ноль! %.2f /
%.2f\n", first, numbers[i]);
            write_to_file(file, result_str);

            write_to_file(file, "Сервер завершил работу из-за деления на ноль\n");

            return 1;
        }

        float result = first / numbers[i];
        snprintf(result_str, sizeof(result_str), "  %.2f / %.2f = %.2f\n", first,
numbers[i], result);
        write_to_file(file, result_str);

        char temp[50];
        snprintf(temp, sizeof(temp), "%.2f ", result);
        strcat(temp_response, temp);
    }
    write_to_file(file, "---\n");

    if (strlen(temp_response) > 0) {
        temp_response[strlen(temp_response) - 1] = '\0';
    }
    snprintf(response_buf, response_size, "OK: %s\n", temp_response);

    return 0;
}

char* open_shared_memory(char* shm_name, int* shm_fd) {
    *shm_fd = shm_open(shm_name, O_RDWR, 0);
    if (*shm_fd == -1) {
        const char msg[] = "Ошибка: не удалось открыть shared memory\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    char* shm_buf = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, *shm_fd, 0);
    if (shm_buf == MAP_FAILED) {
        const char msg[] = "Ошибка: не удалось отобразить shared memory\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    return shm_buf;
}

sem_t* open_semaphore(char* sem_name) {
    sem_t* sem = sem_open(sem_name, O_RDWR);
    if (sem == SEM_FAILED) {
        const char msg[] = "Ошибка: не удалось открыть семафор\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }
    return sem;
```

```
}

int main(int argc, char* argv[]) {
    if (argc != 6) {
        const char msg[] = "Использование: client <shm_s2c> <shm_c2s> <sem_s2c> <sem_c2s>
<файл_результатов>\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    char* shm_s2c_name = argv[1];
    char* shm_c2s_name = argv[2];
    char* sem_s2c_name = argv[3];
    char* sem_c2s_name = argv[4];
    char* filename = argv[5];

    int file = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file == -1) {
        const char msg[] = "Ошибка открытия файла\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    write_to_file(file, "Сервер запущен. Ожидание чисел...\n");

    int shm_s2c_fd, shm_c2s_fd;
    char* shm_s2c_buf = open_shared_memory(shm_s2c_name, &shm_s2c_fd);
    char* shm_c2s_buf = open_shared_memory(shm_c2s_name, &shm_c2s_fd);

    sem_t* sem_s2c = open_semaphore(sem_s2c_name);
    sem_t* sem_c2s = open_semaphore(sem_c2s_name);

    bool running = true;
    while (running) {
        sem_wait(sem_s2c);

        uint32_t* length_from_server = (uint32_t*)shm_s2c_buf;
        char* text_from_server = shm_s2c_buf + sizeof(uint32_t);

        if (*length_from_server == UINT32_MAX) {
            running = false;
            break;
        } else if (*length_from_server > 0) {
            text_from_server[*length_from_server] = '\0';

            char response[SHM_SIZE - sizeof(uint32_t)];
            int status = process_numbers(file, text_from_server, response,
sizeof(response));

            uint32_t* length_to_server = (uint32_t*)shm_c2s_buf;
            char* text_to_server = shm_c2s_buf + sizeof(uint32_t);

            if (status == 1) {
                *length_to_server = UINT32_MAX;
```

```
                running = false;
            } else {
                *length_to_server = strlen(response);
                memcpy(text_to_server, response, *length_to_server);
                text_to_server[*length_to_server] = '\0';
            }

            sem_post(sem_c2s);
        }

        // Нужно сбросить длину, иначе сервер будет читать старые данные
        *length_from_server = 0;
    }

    write_to_file(file, "Сервер завершил работу\n");

    sem_close(sem_s2c);
    sem_close(sem_c2s);

    munmap(shm_s2c_buf, SHM_SIZE);
    munmap(shm_c2s_buf, SHM_SIZE);

    close(shm_s2c_fd);
    close(shm_c2s_fd);
    close(file);

    return 0;
}
```

server.c

```
#include <fcntl.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/fcntl.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <sys/mman.h>

#define SHM_SIZE 4096

char* create_shared_memory(char* shm_name, int* shm_fd) {
    shm_unlink(shm_name);

    *shm_fd = shm_open(shm_name, O_RDWR | O_CREAT | O_TRUNC, 0600);
    if (*shm_fd == -1) {
        const char msg[] = "Ошибка: не удалось создать shared memory\n";
        write(STDERR_FILENO, msg, sizeof(msg));
```

```
        exit(EXIT_FAILURE);
    }

    if (ftruncate(*shm_fd, SHM_SIZE) == -1) {
        const char msg[] = "Ошибка: не удалось установить размер shared memory\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    char* shm_buf = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, *shm_fd, 0);
    if (shm_buf == MAP_FAILED) {
        const char msg[] = "Ошибка: не удалось отобразить shared memory\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    uint32_t* length = (uint32_t*)shm_buf;
    *length = 0;

    return shm_buf;
}

sem_t* create_semaphore(char* sem_name) {
    sem_unlink(sem_name);
    sem_t* sem = sem_open(sem_name, O_RDWR | O_CREAT | O_TRUNC, 0600, 0);
    if (sem == SEM_FAILED) {
        const char msg[] = "Ошибка: не удалось создать семафор\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }
    return sem;
}

int main(int argc, char** argv) {
    if (argc != 2) {
        const char msg[] = "Использование: server <имя_файла>\n";
        write(STDERR_FILENO, msg, strlen(msg));
        exit(EXIT_FAILURE);
    }

    char* filename = argv[1];

    char shm_s2c_name[64];
    char shm_c2s_name[64];
    char sem_s2c_name[64];
    char sem_c2s_name[64];

    snprintf(shm_s2c_name, sizeof(shm_s2c_name), "lab3-shm-s2c-%d", getpid());
    snprintf(shm_c2s_name, sizeof(shm_c2s_name), "lab3-shm-c2s-%d", getpid());
    snprintf(sem_s2c_name, sizeof(sem_s2c_name), "lab3-sem-s2c-%d", getpid());
    snprintf(sem_c2s_name, sizeof(sem_c2s_name), "lab3-sem-c2s-%d", getpid());

    int shm_s2c_fd, shm_c2s_fd;
    char* shm_s2c_buf = create_shared_memory(shm_s2c_name, &shm_s2c_fd);
```

```
    char* shm_c2s_buf = create_shared_memory(shm_c2s_name, &shm_c2s_fd);

    sem_t* sem_s2c = create_semaphore(sem_s2c_name);
    sem_t* sem_c2s = create_semaphore(sem_c2s_name);

    pid_t client_pid = fork();

    if (client_pid == 0) {
        char* args[] = {
            "./client",
            shm_s2c_name,
            shm_c2s_name,
            sem_s2c_name,
            sem_c2s_name,
            filename,
            NULL
        };

        execv("./client", args);

        const char msg[] = "Ошибка: не удалось запустить клиент\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);

    } else if (client_pid == -1) {
        const char msg[] = "Ошибка: не удалось создать процесс клиента\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    printf("Сервер запущен. Дочерний процесс PID: %d\n", client_pid);
    printf("Вводите числа (первое делится на остальные), 'exit' для выхода:\n");

    bool running = true;
    while (running) {
        char input_buf[SHM_SIZE - sizeof(uint32_t)];
        printf("Введите числа: ");
        fflush(stdout);

        if (fgets(input_buf, sizeof(input_buf), stdin) == NULL) {
            break;
        }

        size_t input_len = strlen(input_buf);
        if (input_len > 0 && input_buf[input_len - 1] == '\n') {
            input_buf[input_len - 1] = '\0';
            input_len--;
        }

        if (strcmp(input_buf, "exit") == 0) {
            printf("Завершение работы...\n");
               uint32_t* length_to_client = (uint32_t*)shm_s2c_buf;
                *length_to_client = UINT32_MAX;  //  сигнал завершения
                 sem_post(sem_s2c);                // разбудить клиента
```

```
            running = false;
            break;
            }

    if (input_len <= 0) {
        continue;
    }

    uint32_t* length_to_client = (uint32_t*)shm_s2c_buf;
    char* text_to_client = shm_s2c_buf + sizeof(uint32_t);

    *length_to_client = input_len;
    memcpy(text_to_client, input_buf, input_len);
    text_to_client[input_len] = '\0';

    sem_post(sem_s2c);

    sem_wait(sem_c2s);

    uint32_t* length_from_client = (uint32_t*)shm_c2s_buf;
    char* text_from_client = shm_c2s_buf + sizeof(uint32_t);

    if (*length_from_client == UINT32_MAX) {
        printf("Обнаружено деление на ноль! Завершаем работу.\n");
        running = false;
    } else if (*length_from_client > 0) {
        text_from_client[*length_from_client] = '\0';
        printf("Сервер: %s", text_from_client);
    }

    *length_from_client = 0;
}

if (running) {
    uint32_t* length_to_client = (uint32_t*)shm_s2c_buf;
    *length_to_client = UINT32_MAX;
    sem_post(sem_s2c);
}

waitpid(client_pid, NULL, 0);

sem_close(sem_s2c);
sem_close(sem_c2s);
sem_unlink(sem_s2c_name);
sem_unlink(sem_c2s_name);

munmap(shm_s2c_buf, SHM_SIZE);
munmap(shm_c2s_buf, SHM_SIZE);

shm_unlink(shm_s2c_name);
shm_unlink(shm_c2s_name);

close(shm_s2c_fd);
```

```
    close(shm_c2s_fd);

    printf("Сервер завершил работу.\n");

    return 0;
}
```

Протокол работы программы

Входные данные:

```
yaroslav@DESKTOP-Q6D5K84:/mnt/c/OS/lab3$ ./server result.txt
Сервер запущен. Дочерний процесс PID: 22883
Вводите числа (первое делится на остальные), 'exit' для выхода:
Введите числа: 1 2 3
Сервер: OK: 0.50 0.33
Введите числа: exit
Завершение работы...
Сервер завершил работу.
```

Выходные данные:

lab3 > results.txt

```
1  Сервер запущен. Ожидание чисел...
2  Делим 1.00 на:
3    1.00 / 2.00 = 0.50
4    1.00 / 3.00 = 0.33
5  ---
6  Сервер завершил работу
7
```

Вывод

При выполнении данной лабораторной работы я научился работать с процессами в ОС, используя разделяемую память и семафоры. Освоил принципы синхронизации работы процессов при помощи семафоров. Реализовал двухстороннее взаимодействие между сервером и клиентом, используя shared memory.