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-211БВ-24

Студент: ПРОХОРОВ В.И.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 17.02.26

Москва, 2026

Постановка задачи

Вариант 2.

Пользователь вводит команды вида: «число число число<endline>». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс считает их сумму и выводит её в файл. Числа имеют тип float. Количество чисел может быть произвольным.

Общий метод и алгоритм решения

Использованные вызовы:

int shm_open(const char *name, int oflag, mode_t mode); - создание или открытие именованного объекта shared memory

int shm_unlink(const char *name); - Удаляет именованный объект shared memory по имени

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset); - Отображает файл или объект shared memory в виртуальную память процесса

int ftruncate(int fd, off_t length); - Устанавливает размер объекта по файловому дескриптору (нужен перед mmap)

int munmap(void *addr, size_t length); - Удаляет отображение (unmap) из адресного пространства

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value); - Создаёт или открывает именованный семафор

int sem_unlink(const char *name); - Удаляет именованный семафор по имени

int sem_close(sem_t *sem); - Закрывает дескриптор семафора в текущем процессе

int sem_wait(sem_t *sem); - Уменьшает значение семафора (блокируется, если значение = 0)

int sem_post(sem_t *sem); Увеличивает значение семафора (разблокирует ждущие процессы)

Клиент и сервер общаются через shared memory в виртуальной памяти. Родитель создаёт shared memory и симофор который управляет доступом к shm после memory mapping, устанавливая размер shm в 4096 – 4 байта на “длину”, отсальные на данные. Родитель запускает дочерний процесс, перенаправляя ему запрос (строку). Из буфера shared memory дочерний процесс читает данные, и складывает вещественные числа (float). После ребёнок готовит буфер вывода, и увеличивая длину в буфере (первые 4 байта) на 4096, так родитель понимает что

ребёнок отправил ответ/ошибку, сигнализируем дочернему процессу об остановке *length = uint32_max, завершаем работу с shared memory и memory mapping и симофором.

Код программы

parent.c

```c
#include <errno.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ctype.h>
#include <fcntl.h>
#include <libgen.h>
#include <mach-o/dyld.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <unistd.h>

#define SHM_SIZE 4096

char SHM_NAME[] = "/shared-memory";
char SEM_NAME[] = "/semaphore";

static char CHILD_PROGRAM_NAME[] = "child";

int main(int argc, char **argv) {
    if (argc != 2) {
        char msg[1024];
        uint32_t len =
            snprintf(msg, sizeof(msg) - 1, "usage: %s <filename>\n", argv[0]);
        write(STDERR_FILENO, msg, len);
        exit(EXIT_SUCCESS);
    }

    // Получение пути
    char progpath[2048];
    uint32_t size = sizeof(progpath);
    if (_NSGetExecutablePath(progpath, &size) != 0) {
        const char msg[] = "ERROR: Failed to get executable path\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }
```

```
    ssize_t len = strlen(progpath);
    while (progpath[len] != '/') {
        --len;
    }
    progpath[len] = '\0';

    // Удаляем существующие shared memory и semaphore (если были)
    shm_unlink(SHM_NAME);
    sem_unlink(SEM_NAME);

    // Создаём shared memory (вирт память), возвращает fd
    int shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_TRUNC, 0600);
    if (shm == -1) {
        const char msg[] = "ERROR: Failed to create SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    // Устанавливаем размер разделяемой памяти
    if (ftruncate(shm, SHM_SIZE) == -1) {
        const char msg[] = "ERROR: Failed to resize SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    // Отображаем shared memory в виртуальную память,
    // чтобы использоват его как массив
    char *shm_buf =
        mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm, 0);
    if (shm_buf == MAP_FAILED) {
        const char msg[] = "ERROR: Failed to map SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    // Создаём семафор (его операции атомарны)
    sem_t *sem = sem_open(SEM_NAME, O_RDWR | O_CREAT | O_TRUNC, 0600, 1);
    if (sem == SEM_FAILED) {
        const char msg[] = "ERROR: Failed to create semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    // Создаём новый процесс
    const pid_t child = fork();

    if (child == -1) {
        // Не удалось создать новый процесс
```

```
        const char msg[] = "ERROR: Failed to spawn new process\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    } else if (child == 0) {
        // В дочернем процессе

        char path[4096];
        snprintf(path, sizeof(path), "%s/%s", progpath, CHILD_PROGRAM_NAME);

        char *const args[] = {CHILD_PROGRAM_NAME, argv[1], NULL};

        int32_t status = execv(path, args);

        if (status == -1) {
            const char msg[] =
                "ERROR: Failed to exec into new executable image\n";
            write(STDERR_FILENO, msg, sizeof(msg));
            exit(EXIT_FAILURE);
        }
    }

    // В родительском процессе
    char buf[SHM_SIZE - sizeof(uint32_t)];
    ssize_t bytes = read(STDIN_FILENO, buf, sizeof(buf));
    if (bytes < 0) {
        const char msg[] = "ERROR: Failed to read from stdin\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    // Отправляем данные дочернему процессу через shared memory
    sem_wait(sem);
    // Указатель на длину данных
    uint32_t *length = (uint32_t *)shm_buf;
    // Указатель на данные
    char *data = shm_buf + sizeof(uint32_t);
    if (bytes > 0) {
        // Пишем данные в виртуальную память
        *length = (uint32_t)bytes;
        memcpy(data, buf, bytes);
        sem_post(sem);

        // Ждём, пока дочерний процесс обработает данные и вернёт результат
        bool running = true;
        while (running) {
            // захватываем память
            sem_wait(sem);
```

```
            uint32_t len = *length;
            if (len >= SHM_SIZE) {
                ssize_t result_len = len - SHM_SIZE;
                *length = 0;
                // Данные прочитаны, отпускаем семафор
                sem_post(sem);
                // Проверяем, начинается ли результат с "ERROR:"
                const char error_prefix[] = "ERROR:";
                int output_fd = STDOUT_FILENO;
                if (result_len >= sizeof(error_prefix) - 1 &&
                    strncmp(data, error_prefix, sizeof(error_prefix) - 1) ==
                        0) {
                    output_fd = STDERR_FILENO;
                }
                // Выводим результат
                write(output_fd, data, result_len);
                running = false;
            } else {
                sem_post(sem);
            }
        }
        // Сигнализируем дочернему процессу о завершении
        sem_wait(sem);
        *length = UINT32_MAX;
        sem_post(sem);
    } else {
        // Нет входных данных: сигнализируем дочернему процессу о завершении
        *length = UINT32_MAX;
        sem_post(sem);
    }

    // Ждём завершения дочернего процесса
    int status;
    if (wait(&status) == -1) {
        const char msg[] = "ERROR: Failed to wait for child\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }
    if (WIFEXITED(status)) {
        if (WEXITSTATUS(status) != 0) {
            exit(EXIT_FAILURE);
        }
    } else {
        // Дочерний процесс завершился ненормально
        exit(EXIT_FAILURE);
    }

    sem_unlink(SEM_NAME);
```

```
    sem_close(sem);

    munmap(shm_buf, SHM_SIZE);
    shm_unlink(SHM_NAME);
    close(shm);
}
```

child.c

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ctype.h>
#include <fcntl.h>
#include <math.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>

#define SHM_SIZE 4096

char SHM_NAME[] = "/shared-memory";
char SEM_NAME[] = "/semaphore";

int main(int argc, char **argv) {
    // Открываем shared memory
    int shm = shm_open(SHM_NAME, O_RDWR, 0);
    if (shm == -1) {
        const char msg[] = "ERROR: Failed to open SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    // Отображаем shared memory в виртуальную память
    char *shm_buf =
        mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm, 0);
    if (shm_buf == MAP_FAILED) {
        const char msg[] = "ERROR: Failed to map SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    // Открываем семафор
```

```
    sem_t *sem = sem_open(SEM_NAME, O_RDWR);
    if (sem == SEM_FAILED) {
        const char msg[] = "ERROR: Failed to open semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    // ждём данные от родителя
    bool running = true;
    while (running) {
        sem_wait(sem);
        uint32_t *length = (uint32_t *)shm_buf;
        char *data = shm_buf + sizeof(uint32_t);
        // код для выхода
        if (*length == UINT32_MAX) {
            running = false;
        }
        // это запрос от родителя
        else if (*length > 0 && *length < SHM_SIZE - sizeof(uint32_t)) {
            // Разбираем ввод исуммируем
            float sum = 0.0;
            int count = 0;
            char *p = data;
            char *endptr;
            while (*p && p < data + *length) {
                if ((*p == ' ' || *p == '\t') && *p != '\n') {
                    p++;
                    continue;
                }
                if (*p == '\n') {
                    break;
                }
                float num = strtof(p, &endptr);
                // больше чем float
                if (num == HUGE_VALF || num == -HUGE_VALF) {
                    const char msg[] = "ERROR: Number out of range\n";
                    memcpy(data, msg, sizeof(msg) - 1);
                    // Сообщаем родителю об ошибке: устанавливаем
                    // длину = длина_сообщения + SHM_SIZE (>= SHM_SIZE)
                    *length = (sizeof(msg) - 1) + SHM_SIZE;
                    sem_post(sem);
                    exit(EXIT_FAILURE);
                }
                // Не получилось ничего прочитать
                if (p == endptr) {
                    const char msg[] = "ERROR: Invalid character in input\n";
                    memcpy(data, msg, sizeof(msg) - 1);
                    *length = (sizeof(msg) - 1) + SHM_SIZE;
```

```
                sem_post(sem);
                exit(EXIT_FAILURE);
            }
            sum += num;
            count++;
            p = endptr;
        }
        // ничего не считали
        if (count == 0) {
            const char msg[] = "ERROR: No numbers provided\n";
            memcpy(data, msg, sizeof(msg) - 1);
            *length = (sizeof(msg) - 1) + SHM_SIZE;
            sem_post(sem);
            exit(EXIT_FAILURE);
        }

        // Открываем файл для записи
        int32_t file =
            open(argv[1], O_WRONLY | O_CREAT | O_TRUNC | O_APPEND, 0600);
        if (file == -1) {
            const char msg[] = "ERROR: Failed to open requested file\n";
            memcpy(data, msg, sizeof(msg) - 1);
            *length = (sizeof(msg) - 1) + SHM_SIZE;
            sem_post(sem);
            exit(EXIT_FAILURE);
        }

        // Форматируем сумму как строку
        char sum_str[64];
        int len = snprintf(sum_str, sizeof(sum_str), "%.2f\n", sum);

        // Записываем сумму в файл
        int32_t written = write(file, sum_str, len);
        if (written != len) {
            const char msg[] = "ERROR: Failed to write to file\n";
            memcpy(data, msg, sizeof(msg) - 1);
            *length = (sizeof(msg) - 1) + SHM_SIZE;
            sem_post(sem);
            exit(EXIT_FAILURE);
        }
        close(file);

        // Отправляем результат родителю через shared memory
        *length = len + SHM_SIZE;
        memcpy(data, sum_str, len);
    }
    sem_post(sem);
}
```

```
    sem_close(sem);
    munmap(shm_buf, SHM_SIZE);
    close(shm);
    return 0;
}
```

Протокол работы программы

```
→  lab_3 git:(main) ✗ make run
./parent out.txt
0.1 0.2 0.3 0.4
1.00
```

out.txt

```
1    1.00
2
```

```
→  lab_3 git:(main) ✗ ./parent out.txt
0.1 0.2 0.3 0.4 0.5 -0.1 -0.4
1.00
```

out.txt

```
1    1.00
2
```

Вывод

При выполнении данной лабораторной работы я научился работать с shared memory, симофорами и memory mapping в ОС. Попробовал создавать дочерние процессы и устанавливать каналы связи с помощью shared memory и синхронизировать доступ к виртуальной памяти с помощью симофора. Научился обрабатывать ввод, вывод как поток байтов, преобразовывать его в строки.