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: Губеев Д.И.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 25.10.25

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 pipe(int \*fd); – создает анонимный канал, возвращает 0 при успехе, -1 при ошибке
* ssize\_t read(int fd, void \*buf, size\_t count) - читает данные из файла по дескриптору fd в буфер buf. Возвращает количество прочитанных байт
* ssize\_t write(int fd, const void \*buf, size\_t count) - записывает данные в файл по дескриптору fd. Возвращает число записанных байт
* 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 создаёт shared memory и семафоры, затем запускает клиента. Сервер записывает введённые числа в память и подаёт сигнал клиенту. Клиент читает данные, выполняет деление и записывает результат обратно. Сервер получает ответ и выводит результат.При пустом вводе или делении на ноль оба процесса завершаются и очищают ресурсы

**Код программы**

**client.c**

#include <stdint.h>

#include <stdio.h>

#include <stdbool.h>

#include <math.h>

#include <ctype.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

int divideNumbers(char \* input, int32\_t file) {

    char buf[4096];

    strncpy(buf, input, sizeof(buf) - 1);

    buf[sizeof(buf) - 1] = '\0';

    char \* saveptr;

    char \* token = strtok\_r(buf, " ", &saveptr);

    float numbers[100];

    int count = 0;

    while (token != NULL && count < 100) {

        char \*endptr;

        float val = strtof(token, &endptr);

        if (endptr != token) {

            numbers[count++] = val;

        }

        token = strtok\_r(NULL, " ", &saveptr);

    }

    if (count < 2) {

        const char msg[] = "error: need at least two numbers\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        return 0;

    }

    float dividend = numbers[0];

    for (int i = 1; i < count; i++) {

        if (fabs(numbers[i]) < 1e-6) {

            const char msg[] = "error: division by zero\n";

            write(STDERR\_FILENO, msg, sizeof(msg));

            return 1;

        }

        float result = dividend / numbers[i];

        char resultStr[50];

        int len = snprintf(resultStr, sizeof(resultStr), "%f ", result);

        if (write(file, resultStr, len) == -1) {

            const char msg[] = "error: failed to write to file\n";

            write(STDERR\_FILENO, msg, sizeof(msg));

            return 1;

        }

    }

    if (write(file, "\n", 1) == -1) {

        const char msg[] = "error: failed to write to file\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        return 1;

    }

    return 0;

}

char \* get\_shm\_buf(char \*SHM\_NAME, int \*shm\_out) {

    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);

    }

    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);

    }

    \*shm\_out = shm;

    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[] = "error: failed to open semaphore\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        \_exit(EXIT\_FAILURE);

    }

    return sem;

}

int main(int argc, char \*argv[])

{

    char \*SHM\_S2C\_NAME = argv[1];

    char \*SHM\_C2S\_NAME = argv[2];

    char \*SEM\_S2C\_NAME = argv[3];

    char \*SEM\_C2S\_NAME = argv[4];

    // NOTE: `O\_WRONLY` only enables file for writing

    // NOTE: `O\_CREAT` creates the requested file if absent

    // NOTE: `O\_TRUNC` empties the file prior to opening

    // NOTE: `O\_APPEND` subsequent writes are being appended instead of overwritten

    int32\_t file = open(argv[5], O\_WRONLY | O\_CREAT | O\_TRUNC | O\_APPEND, 0600);

    if (file == -1) {

        const char msg[] = "error: failed to open requested file\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        exit(EXIT\_FAILURE);

    }

    int shm\_s2c;

    int shm\_c2s;

    char \*shm\_s2c\_buf = get\_shm\_buf(SHM\_S2C\_NAME, &shm\_s2c);

    char \*shm\_c2s\_buf = get\_shm\_buf(SHM\_C2S\_NAME, &shm\_c2s);

    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 = (uint32\_t \*)shm\_s2c\_buf;

        char \*text = shm\_s2c\_buf + sizeof(uint32\_t);

        if (\*length == UINT32\_MAX) {

            running = false;

        } else if (\*length > 0) {

            text[\*length] = '\0';

            int status = divideNumbers(text, file);

            uint32\_t \*length\_to\_server = (uint32\_t \*)shm\_c2s\_buf;

            char \*text\_to\_server = shm\_c2s\_buf + sizeof(uint32\_t);

            if (status != 0) {

                \*length\_to\_server = UINT32\_MAX;

                sem\_post(sem\_c2s);

                exit(EXIT\_FAILURE);

            } else {

                memset(text\_to\_server, 0, SHM\_SIZE - sizeof(uint32\_t));

                \*length\_to\_server = \*length;

                memcpy(text\_to\_server, text, \*length);

                text\_to\_server[\*length] = '\0';

            }

            const char msg[] = "CLIENT RECEIVED: ";

            write(STDOUT\_FILENO, msg, sizeof(msg) - 1);

            \*length = 0;

        }

        sem\_post(sem\_c2s);

    }

    sem\_close(sem\_s2c);

    sem\_close(sem\_c2s);

    munmap(shm\_s2c\_buf, SHM\_SIZE);

    munmap(shm\_c2s\_buf, SHM\_SIZE);

    close(shm\_s2c);

    close(shm\_c2s);

    close(file);

}

**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 <wait.h>

#include <semaphore.h>

#include <sys/mman.h>

#define SHM\_SIZE 4096

char \* get\_shm\_buf(char \*SHM\_NAME, int \*shm\_out) {

    shm\_unlink(SHM\_NAME);

    int shm = shm\_open(SHM\_NAME, O\_RDWR, 0600);

    if (shm == -1 && errno != ENOENT) {

        const char msg[] = "error: failed to open SHM\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        \_exit(EXIT\_FAILURE);

    }

    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);

    }

    if (ftruncate(shm, SHM\_SIZE) == -1) {

        const char msg[] = "error: failed to resize SHM\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        \_exit(EXIT\_FAILURE);

    }

    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);

    }

    uint32\_t \*length = (uint32\_t \*)shm\_buf;

    \*length = 0;

    \*shm\_out = shm;

    return shm\_buf;

}

sem\_t \*open\_semaphore(char \* SEM\_NAME) {

    sem\_t \*sem = sem\_open(SEM\_NAME, O\_RDWR | O\_CREAT | O\_TRUNC, 0600, 0);

    if (sem == SEM\_FAILED) {

        const char msg[] = "error: failed to create semaphore\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        \_exit(EXIT\_FAILURE);

    }

    return sem;

}

int main(int argc, char \*\*argv)

{

    if (argc == 1) {

        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);

    }

    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), "example-shm-s2c-%d", getpid());

    snprintf(SHM\_C2S\_NAME, sizeof(SHM\_C2S\_NAME), "example-shm-c2s-%d", getpid());

    snprintf(SEM\_S2C\_NAME, sizeof(SEM\_S2C\_NAME), "example-sem-s2c-%d", getpid());

    snprintf(SEM\_C2S\_NAME, sizeof(SEM\_C2S\_NAME), "example-sem-c2s-%d", getpid());

    sem\_unlink(SEM\_S2C\_NAME);

    sem\_unlink(SEM\_C2S\_NAME);

    int shm\_s2c;

    int shm\_c2s;

    char \* shm\_s2c\_buf = get\_shm\_buf(SHM\_S2C\_NAME, &shm\_s2c);

    char \* shm\_c2s\_buf = get\_shm\_buf(SHM\_C2S\_NAME, &shm\_c2s);

    sem\_t \*sem\_s2c = open\_semaphore(SEM\_S2C\_NAME);

    sem\_t \*sem\_c2s = open\_semaphore(SEM\_C2S\_NAME);

    pid\_t client = fork();

    if (client == 0) {

        char \*args[] = {"client.out", SHM\_S2C\_NAME, SHM\_C2S\_NAME, SEM\_S2C\_NAME, SEM\_C2S\_NAME, argv[1], NULL};

        execv("./client.out", args);

        const char msg[] = "error: failed to exec\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        \_exit(EXIT\_FAILURE);

    } else if (client == -1) {

        const char msg[] = "error: failed to fork\n";

        write(STDERR\_FILENO, msg, sizeof(msg));

        \_exit(EXIT\_FAILURE);

    }

    bool running = true;

    while (running) {

        char buf[SHM\_SIZE - sizeof(uint32\_t)];

        ssize\_t bytes = read(STDIN\_FILENO, buf, sizeof(buf));

        if (bytes == -1) {

            const char msg[] = "error: failed to read from standard input\n";

            write(STDERR\_FILENO, msg, sizeof(msg));

            \_exit(EXIT\_FAILURE);

        }

        uint32\_t \*length = (uint32\_t \*)shm\_s2c\_buf;

        char \*text = shm\_s2c\_buf + sizeof(uint32\_t);

        if (buf[0] == '\n' || bytes <= 0) {

            running = false;

            \*length = UINT32\_MAX;

            sem\_post(sem\_s2c);

            break;

        } else {

            memset(text, 0, SHM\_SIZE - sizeof(uint32\_t));

            \*length = bytes;

            memcpy(text, buf, bytes);

            text[bytes] = '\0';

        }

        sem\_post(sem\_s2c);

        sem\_wait(sem\_c2s);

        uint32\_t \*length\_client = (uint32\_t \*)shm\_c2s\_buf;

        char \*text\_client = shm\_c2s\_buf + sizeof(uint32\_t);

        if (\*length\_client == UINT32\_MAX) { // division by zero

            running = false;

            sem\_post(sem\_c2s);

            break;

        } else if (\*length\_client > 0) {

            write(STDOUT\_FILENO, text\_client, \*length\_client);

        }

        \*length\_client = 0;

    }

    waitpid(client, NULL, 0);

    sem\_unlink(SEM\_S2C\_NAME);

    sem\_unlink(SEM\_C2S\_NAME);

    sem\_close(sem\_s2c);

    sem\_close(sem\_c2s);

    munmap(shm\_s2c\_buf, SHM\_SIZE);

    munmap(shm\_c2s\_buf, SHM\_SIZE);

    shm\_unlink(SHM\_C2S\_NAME);

    shm\_unlink(SHM\_S2C\_NAME);

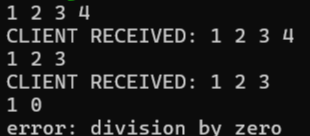
    close(shm\_s2c);

    close(shm\_c2s);

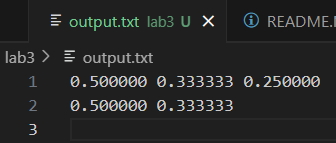
}

**Протокол работы программы**

**Входные данные:**

****

**Выходные данные:**

****

**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.