

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<newline>». 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)` - закрывает 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 = strtod(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

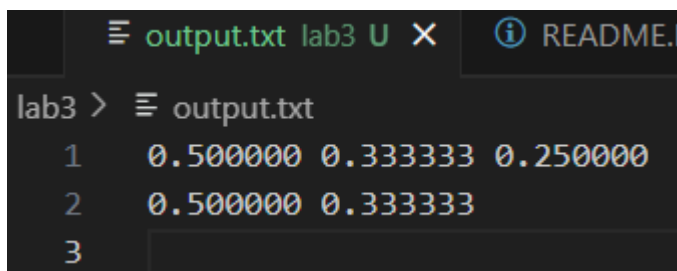
Входные данные:

```

1 2 3 4
CLIENT RECEIVED: 1 2 3 4
1 2 3
CLIENT RECEIVED: 1 2 3
1 0
error: division by zero

```

Выходные данные:



```

lab3 > output.txt
1 0.500000 0.333333 0.250000
2 0.500000 0.333333
3

```

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.