

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-211Б-23

Студент: Бугренков В.П.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_

Дата: 17.12.24

Москва, 2024

# Постановка задачи

Вариант 9.

Цель работы

Приобретение практических навыков в:

- Управление процессами в ОС
- Обеспечение обмена данных между процессами посредством shared memory и memory mapping

Задание

В файле записаны команды вида: «число число число<newline>». Дочерний процесс производит деление первого числа команда, на последующие числа в команде, а результат выводит в стандартный поток вывода. 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 shm_open(const char *__name, int __oflag, mode_t __mode)` — открывает сегмент shm
- `void *mmap(void *__addr, size_t __len, int __prot, int __flags, int __fd, off_t __offset)` — создает новый маппинг в виртуальном адресном пространстве
- `sem_t *sem_open (const char *__name, int __oflag, ...)` - открывает именованный семафор
- `int sem_unlink (const char *__name)` — удаляет именованный семафор
- `int sem_wait(sem_t *sem)` - уменьшает (блокирует) семафор
- `int sem_post(sem_t *sem)` - увеличивает (разблокирует) семафор
- `int open(const char *pathname, int flags, mode_t mode)` - открытие\создание файла
- `int close(int fd)` - закрыть файл
- `void exit(int status)` - завершения выполнения процесса и возвращение статуса
- `int execv(const char *filename, char *const argv[])` - замена образа памяти процесса
- `pid_t getpid(void)` — получение ID процесса
- `ssize_t read(int __fd, void* __buf, size_t __nbytes)` — чтение из fd в буфер
- `ssize_t write(int __fd, const void* __buf, size_t __n)` — запись байтов в буфер

В решении задачи используется взаимодействие между родительским и дочерним процессами через общую память (shared memory) и синхронизацию через семафоры. Родительский процесс выполняет следующие шаги:

1. Открывает файл с помощью системной функции `fopen`.
2. Создает объект общей памяти с помощью `shm_open`, задает его размер с помощью `ftruncate` и мапирует его в пространство памяти с помощью `mmap`.
3. Создает семафор с помощью `sem_open` для синхронизации доступа к общей памяти между процессами.
4. Читает строки из файла и записывает их в общую память с использованием системного вызова `memcpy` или прямым обращением к области памяти.
5. Сигнализирует дочернему процессу с помощью `sem_post`, что данные готовы для обработки.

Дочерний процесс выполняет следующие шаги:

1. Создает локальную ссылку на общую память с помощью `mmap`, где он будет получать данные от родительского процесса.
2. Ожидает, пока родительский процесс запишет данные в общую память, с помощью `sem_wait` (блокировка до получения данных).
3. Читает строку из общей памяти, разбивает ее на токены и выполняет операцию деления чисел, проверяя деление на ноль.
4. После выполнения вычислений выводит результат с помощью системных вызовов `write` или `writev`.
5. После завершения работы с данными, дочерний процесс отправляет сигнал родительскому процессу о завершении с помощью `sem_post`.

Завершающие шаги:

1. Родительский процесс получает уведомление от дочернего, что данные обработаны, и повторяет цикл до конца файла.
2. Когда все данные обработаны, родительский процесс завершает выполнение, используя системную функцию `munmap` для освобождения памяти и `sem_close` для закрытия семафоров.
3. Обе стороны очищают ресурсы с помощью `shm_unlink` и `sem_unlink`.

Таким образом, взаимодействие между процессами и синхронизация через семафоры обеспечивают правильную последовательность операций и завершение работы программы.

## Код программы

```
parent.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <sys/stat.h>

#define BUFFER_SIZE 512
#define END_MARKER "END"

typedef enum Errors {
    E_SUCCESS = 0,
    E_INVALID_INPUT,
    E_CANNOT_OPEN_FILE,
    E_PIPE_FAILED,
    E_SEMAPHORE_FAILED
} ERRORS_EXIT_CODES;

void write_error(const char *error_string) {
    if (error_string == NULL) {
        write(STDERR_FILENO, "ERROR", 6);
    }
    write(STDERR_FILENO, error_string, strlen(error_string));
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        write_error("ERROR: Missing filename argument\n");
        exit(EXIT_FAILURE);
    }
}
```

```

FILE *file = fopen(argv[1], "r");
if (file == NULL) {
    write_error("ERROR: Cannot open file\n");
    exit(EXIT_FAILURE);
}

// Создаем общую память
int shm_fd = shm_open("/shared_mem", O_CREAT | O_RDWR, 0666);
if (shm_fd == -1) {
    fclose(file);
    write_error("ERROR: shm_open failed\n");
    exit(EXIT_FAILURE);
}

// Устанавливаем размер общей памяти
if (ftruncate(shm_fd, BUFFER_SIZE) == -1) {
    close(shm_fd); // Закрываем дескриптор памяти
    fclose(file);
    write_error("ERROR: ftruncate failed\n");
    exit(EXIT_FAILURE);
}

// Отображаем общую память
char *shm_ptr = mmap(0, BUFFER_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
if (shm_ptr == MAP_FAILED) {
    close(shm_fd); // Закрываем дескриптор памяти
    fclose(file); // Освобождаем файл
    write_error("ERROR: mmap failed\n");
    exit(EXIT_FAILURE);
}

// Создаем семафоры для синхронизации
sem_t *sem_parent_write = sem_open("/sem_parent_write", O_CREAT, 0666, 0);
sem_t *sem_child_read = sem_open("/sem_child_read", O_CREAT, 0666, 0);
if (sem_parent_write == SEM_FAILED || sem_child_read == SEM_FAILED) {
    munmap(shm_ptr, BUFFER_SIZE); // Освобождаем память
    close(shm_fd); // Закрываем дескриптор памяти
    fclose(file); // Освобождаем файл
    write_error("ERROR: sem_open failed\n");
    exit(EXIT_FAILURE);
}

pid_t pid = fork();
if (pid < 0) {
    sem_close(sem_parent_write);
    sem_close(sem_child_read);
    sem_unlink("/sem_parent_write");
    sem_unlink("/sem_child_read");

    munmap(shm_ptr, BUFFER_SIZE);
    close(shm_fd);
    shm_unlink("/shared_mem");
    fclose(file);
    write_error("ERROR: fork failed\n");
    exit(EXIT_FAILURE);
}

if (pid == 0) {
    // Дочерний процесс

```

```

    close(shm_fd);
    execl("./child", "", NULL);
    write_error("ERROR: execl failed\n");
    exit(EXIT_FAILURE);
} else {
    // Родительский процесс
    char file_buffer[BUFFER_SIZE];
    while (fgets(file_buffer, sizeof(file_buffer), file) != NULL) {
        // Записываем команду в общую память
        int i = 0;
        while (file_buffer[i] != '\0' && i < BUFFER_SIZE - 1) {
            shm_ptr[i] = file_buffer[i];
            i++;
        }
        shm_ptr[i] = '\0'; // Добавляем нулевой символ в конец строки

        // Сигнализируем дочернему процессу, что данные готовы для обработки
        sem_post(sem_parent_write);

        // Ждем, пока дочерний процесс завершит обработку
        sem_wait(sem_child_read);
    }

    // Отправляем специальный маркер для завершения работы дочернего процесса
    int i = 0;
    while (END_MARKER[i] != '\0' && i < BUFFER_SIZE - 1) {
        shm_ptr[i] = END_MARKER[i];
        i++;
    }
    shm_ptr[i] = '\0';
    sem_post(sem_parent_write); // Дочерний процесс получит маркер завершения

    // Закрываем семафоры
    sem_close(sem_parent_write);
    sem_close(sem_child_read);

    // Закрываем файл
    fclose(file);

    // Ожидаем завершения дочернего процесса
    wait(NULL);
}

sem_close(sem_parent_write);
sem_close(sem_child_read);
sem_unlink("/sem_parent_write");
sem_unlink("/sem_child_read");

munmap(shm_ptr, BUFFER_SIZE);
close(shm_fd);
shm_unlink("/shared_mem");
fclose(file);

return E_SUCCESS;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <fcntl.h>

#define BUFFER_SIZE 512
#define END_MARKER "END"

typedef enum Errors {
    E_SUCCESS = 0,
    E_INVALID_INPUT,
    E_DEVIDE_BY_ZERO
} ERRORS_EXIT_CODES;

void write_error(const char *error_string) {
    if (error_string == NULL) {
        write(STDERR_FILENO, "ERROR", 6);
    }
    write(STDERR_FILENO, error_string, strlen(error_string));
}

int process_command(const char *command) {
    char *token;
    float result = 0.0;
    int first = 1;

    // Копируем строку, чтобы не изменять оригинал
    char buffer[BUFFER_SIZE];
    int i = 0;
    while (command[i] != '\0' && i < BUFFER_SIZE - 1) {
        buffer[i] = command[i];
        i++;
    }
    buffer[i] = '\0'; // Нулевой символ для завершения строки

    token = strtok(buffer, " ");

    while (token != NULL) {
        float num = atof(token);

        if (first) {
            result = num;
            first = 0;
        } else {
            if (num == 0) {
                write_error("ERROR: Division by zero\n");
                return E_DEVIDE_BY_ZERO;
            }
            result /= num;
        }

        token = strtok(NULL, " ");
    }

    write(STDOUT_FILENO, "Division result is: ", 20);
    char result_str[BUFFER_SIZE];
    int length = snprintf(result_str, sizeof(result_str), "%f\n", result);

```

```

write(STDOUT_FILENO, result_str, length);

return E_SUCCESS;
}

int main() {
    // Открываем общую память
    int shm_fd = shm_open("/shared_mem", O_RDWR, 0666);
    if (shm_fd == -1) {
        write_error("ERROR: shm_open failed\n");
        exit(EXIT_FAILURE);
    }

    // Отображаем общую память
    char *shm_ptr = mmap(0, BUFFER_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (shm_ptr == MAP_FAILED) {
        close(shm_fd); // Закрываем дескриптор памяти
        write_error("ERROR: mmap failed\n");
        exit(EXIT_FAILURE);
    }

    // Открываем семафоры для синхронизации
    sem_t *sem_parent_write = sem_open("/sem_parent_write", 0);
    sem_t *sem_child_read = sem_open("/sem_child_read", 0);
    if (sem_parent_write == SEM_FAILED || sem_child_read == SEM_FAILED) {
        munmap(shm_ptr, BUFFER_SIZE); // Освобождаем память
        close(shm_fd); // Закрываем дескриптор памяти
        write_error("ERROR: sem_open failed\n");
        exit(EXIT_FAILURE);
    }

    while (1) {
        // Ждем, пока родительский процесс напишет команду
        sem_wait(sem_parent_write);

        // Проверяем маркер завершения
        if (strcmp(shm_ptr, END_MARKER) == 0) {
            break; // Завершаем работу, если получен маркер
        }

        // Обрабатываем команду из общей памяти
        if (process_command(shm_ptr) == E_DEVIDE_BY_ZERO) {
            break;
        }

        // Сигнализируем родительскому процессу, что обработка завершена
        sem_post(sem_child_read);
    }

    // Закрываем семафоры
    sem_close(sem_parent_write);
    sem_close(sem_child_read);

    // Отключаем общую память
    munmap(shm_ptr, BUFFER_SIZE);
    close(shm_fd);

    return E_SUCCESS;
}

```

## Протокол работы программы

```
qwental@DESKTOP-NKF1EUK:~/workspace/OS_LABS/lab3/src$ gcc child.c -o child
```

```
qwental@DESKTOP-NKF1EUK:~/workspace/OS_LABS/lab3/src$ gcc parent.c -o parent
```

```
qwental@DESKTOP-NKF1EUK:~/workspace/OS_LABS/lab3/src$ cat test.txt
```

100 2 2 5

50 5 2

120 4 2 3 2

200 2 2 2 2 2

15 3 3

36 6 6

1000 5 5 5 2

8 2 2

90 10 3 3

60 2 3 5

45 5 5

25 5 1

300 3 5 2 2

100 2 4 5

40 4 2

64 8 2

128 8 2

500 10 2 5

100 2 2 2 2

```
qwental@DESKTOP-NKF1EUK:~/workspace/OS_LABS/lab3/src$ ./parent test.txt
```

Division result is: 5.000000

Division result is: 5.000000

Division result is: 2.500000

Division result is: 6.250000

Division result is: 1.666667

Division result is: 1.000000

Division result is: 4.000000

Division result is: 2.000000



Division result is: 1.000000

Division result is: 2.000000

Division result is: 1.800000

Division result is: 5.000000

Division result is: 5.000000

Division result is: 2.500000

Division result is: 5.000000

Division result is: 4.000000

Division result is: 8.000000

Division result is: 5.000000

Division result is: 6.250000

### **Strace:**

```
qwental@DESKTOP-NKF1EUK:~/workspace/OS_LABS/lab3/src$ strace ./parent test.txt
execve("./parent", [".parent", "test.txt"], 0x7ffdb85b8 /* 21 vars */) = 0
brk(NULL)                               = 0x5625a6bbe000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd7afc4000) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f54629f4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=30263, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 30263, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f54629ec000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0@\0\0\0\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0I\17\357\204\3$\f221\2039x\324\224\323\236S"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0@\0\0\0\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f54627c3000
mprotect(0x7f54627eb000, 2023424, PROT_NONE) = 0
```

```

mmap(0x7f54627eb000, 1658880, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f54627eb000
mmap(0x7f5462980000, 360448, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f5462980000
mmap(0x7f54629d9000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7f54629d9000
mmap(0x7f54629df000, 52816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f54629df000
close(3) = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f54627c0000
arch_prctl(ARCH_SET_FS, 0x7f54627c0740) = 0
set_tid_address(0x7f54627c0a10) = 2024
set_robust_list(0x7f54627c0a20, 24) = 0
rseq(0x7f54627c10e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f54629d9000, 16384, PROT_READ) = 0
mprotect(0x5625a5f9a000, 4096, PROT_READ) = 0
mprotect(0x7f5462a2e000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f54629ec000, 30263) = 0
getrandom("\x7d\x3c\xc8\x6b\x76\x2e\x00\x0e", 8, GRND_NONBLOCK) = 8
brk(NULL) = 0x5625a6bbe000
brk(0x5625a6bdf000) = 0x5625a6bdf000
openat(AT_FDCWD, "test.txt", O_RDONLY) = 3
openat(AT_FDCWD, "/dev/shm/shared_mem",
O_RDWR|O_CREAT|O_NOFOLLOW|O_CLOEXEC, 0666) = 4
ftruncate(4, 512) = 0
mmap(NULL, 512, PROT_READ|PROT_WRITE, MAP_SHARED, 4, 0) = 0x7f5462a2d000
openat(AT_FDCWD, "/dev/shm/sem.sem_parent_write", O_RDWR|O_NOFOLLOW) = 5
newfstatat(5, "", {st_mode=S_IFREG|0644, st_size=32, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 32, PROT_READ|PROT_WRITE, MAP_SHARED, 5, 0) = 0x7f54629f3000
close(5) = 0
openat(AT_FDCWD, "/dev/shm/sem.sem_child_read", O_RDWR|O_NOFOLLOW) = 5
newfstatat(5, "", {st_mode=S_IFREG|0644, st_size=32, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 32, PROT_READ|PROT_WRITE, MAP_SHARED, 5, 0) = 0x7f54629f2000
close(5) = 0
clone(child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7f54627c0a10) = 2025
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=176, ...}, AT_EMPTY_PATH) = 0
read(3, "100 2 2 5\n50 5 2\n120 4 2 3 2\n200"..., 4096) = 176

```

Division result is:

**futex** (0x7f54629f2000, FUTEX\_WAIT\_BITSET|FUTEX\_CLOCK\_REALTIME, 0, NULL, FUTEX\_BITSET\_MATCH\_ANY5.000000

) = -1 EAGAIN (Resource temporarily unavailable)

**futex** (0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
5.000000

**futex** (0x7f54629f3000, FUTEX\_WAKE, 1Division result is: 2.500000  
) = 1

**futex** (0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
6.250000

**futex** (0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
1.666667

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
1.000000

**futex**(0x7f54629f2000, FUTEX\_WAIT\_BITSET|FUTEX\_CLOCK\_REALTIME, 0, NULL, FUTEX\_BITSET\_MATCH\_ANY) = -1 EAGAIN (Resource temporarily unavailable)

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
4.000000

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: 2.000000  
) = 1

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: 1.000000  
) = 1

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
2.000000

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
1.800000

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
5.000000

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
5.000000

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: 2.500000  
) = 1

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
5.000000

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
4.000000

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: 8.000000  
) = 1

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: 5.000000  
) = 1

**futex**(0x7f54629f3000, FUTEX\_WAKE, 1Division result is: ) = 1  
6.250000

```
read(3, "", 4096) = 0
futex(0x7f54629f3000, FUTEX_WAKE, 1) = 1
munmap(0x7f54629f3000, 32) = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=2025, si_uid=1000,
si_status=0, si_utime=0, si_stime=1} ---
munmap(0x7f54629f2000, 32) = 0
close(3) = 0
wait4(-1, NULL, 0, NULL) = 2025
exit_group(0) = ?
+++ exited with 0 +++
```

## Вывод

В ходе лабораторной работы я приобрел практические навыки в управлении процессами ОС и обеспечении обмена данных между процессами посредством shared memory и mmap. Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. Проблем в ходе выполнения не возникло.