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: Рыбин В.В.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 23.10.25

Москва, 2025

Постановка задачи

Вариант 12.

Child1 переводит строки в верхний регистр. Child2 убирает все задвоенные пробелы.

Общий метод и алгоритм решения

Использованные системные вызовы:

Системные вызовы для работы с разделяемой памятью

- int shm_open(const char *name, int oflag, mode_t mode)
  Создает или открывает объект разделяемой памяти. Возвращает файловый дескриптор или -1 при ошибке.
- int ftruncate(int fd, off_t length)
  Изменяет размер файла или разделяемой памяти. Возвращает 0 при успехе, -1 при ошибке.
- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
  Отображает файл или разделяемую память в адресное пространство процесса. Возвращает указатель на отображенную область или MAP_FAILED.
- int munmap(void *addr, size_t length)
  Удаляет отображение памяти. Возвращает 0 при успехе, -1 при ошибке.
- int shm_unlink(const char *name)
  Удаляет объект разделяемой памяти. Возвращает 0 при успехе, -1 при ошибке.

Системные вызовы для работы с семафорами

- sem_t *sem_open(const char *name, int oflag, ...)
  Открывает или создает именованный семафор. Возвращает указатель на семафор или SEM_FAILED.
- int sem_wait(sem_t *sem)
  Ожидает семафор (уменьшает значение на 1). Возвращает 0 при успехе, -1 при ошибке.
- int sem_post(sem_t *sem)
  Освобождает семафор (увеличивает значение на 1). Возвращает 0 при успехе, -1 при ошибке.
- int sem_close(sem_t *sem)
  Закрывает семафор. Возвращает 0 при успехе, -1 при ошибке.
- int sem_unlink(const char *name)
  Удаляет именованный семафор. Возвращает 0 при успехе, -1 при ошибке.

Системные вызовы для управления процессами

- pid_t fork(void)
  Создает новый процесс-потомок. Возвращает 0 в потомке, PID потомка в родителе, -1 при ошибке.
- int execv(const char *path, char *const argv[])
  Заменяет текущий процесс новым процессом. Возвращает -1 только при ошибке.

- pid_t waitpid(pid_t pid, int *wstatus, int options)
  Ожидает завершения указанного процесса. Возвращает PID завершенного процесса или -1.

Функции для работы с файлами и вводом-выводом

- ssize_t write(int fd, const void *buf, size_t count)
  Записывает данные в файловый дескриптор. Возвращает количество записанных байт или -1.
- ssize_t read(int fd, void *buf, size_t count)
  Читает данные из файлового дескриптора. Возвращает количество прочитанных байт или -1.
- int close(int fd)
  Закрывает файловый дескриптор. Возвращает 0 при успехе, -1 при ошибке.

В ходе лабораторной работы я создавал объекты разделяемой памяти, отображал их в адресное пространство процессов. Также я создавал несколько процессов, чтобы они могли общаться между собой благодаря той самой разделяемой памяти для обработки входных данных. Чтобы не было гонки данных, я использовал семафор.

Код программы

main.c

```c
#include <fcntl.h>
#include <stdint.h>
#include <stdbool.h>

#include <stdlib.h>
#include <string.h>
#include <errno.h>

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


#define SHM_SIZE 4096

const char SHM_NAME[] = "shm-name";
const char SEM_NAME_SERVER[] = "example-1-sem-server";
const char SEM_NAME_CHILD_1[] = "example-1-sem-child-1";
const char SEM_NAME_CHILD_2[] = "example-1-sem-child-2";


int main() {
    int shm = shm_open(SHM_NAME, O_RDWR, 0600);
    if (shm == -1 && errno != ENOENT) {
        // enoent - ошибка: нет такого файла или каталога
        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);
    // O_RDWR - открыть на запись и на чтение
    // O_CREAT - создать, если не существует
    // O_TRUNC - если существовует, то стереть
    // 0600 - права доступа, 6 = 110
    // владелец может читать и писать

    if (shm == -1) {
        // возникла ошибка при создании разделяемой памяти
        const char msg[] = "error: failed to open SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    if (ftruncate(shm, SHM_SIZE) == -1) {
        // shm_open просто создал разделяемую память нулевой длины
        // функция ftruncate изменяет размер этой памяти до SHM_SIZE
        // возвращает -1, если возникла ошибка
        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);
    // отображает разделяемую память в адресное пространство процесса
    if (shm_buf == MAP_FAILED) {
        const char msg[] = "error: failed to map SHM\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    sem_t *sem_server = sem_open(SEM_NAME_SERVER, O_RDWR | O_CREAT | O_TRUNC, 0600,
1);
    if (sem_server == SEM_FAILED) {
        const char msg[] = "error: failed to create semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    sem_t *sem_client_1 = sem_open(SEM_NAME_CHILD_1, O_RDWR | O_CREAT | O_TRUNC,
0600, 0);
    if (sem_client_1 == SEM_FAILED) {
        const char msg[] = "error: failed to create semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    sem_t *sem_client_2 = sem_open(SEM_NAME_CHILD_2, O_RDWR | O_CREAT | O_TRUNC,
0600, 0);
    if (sem_client_2 == SEM_FAILED) {
```

```
        const char msg[] = "error: failed to create semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    pid_t child_1 = fork();

    if (child_1 == 0) {
        // значит мы находимся в дочернем процессе
        char *args[] = {"child_1", NULL};
        execv("./child_1", args);

        const char msg[] = "error: failed to exec\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }
    else if (child_1 == -1) {
        const char msg[] = "error: failed to fork\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    pid_t child_2 = fork();

    if (child_2 == 0) {
        // находимся в дочернем процессе
        char *args[] = {"child_2", NULL};
        execv("./child_2", args);

        const char msg[] = "error: failed to exec\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }
    else if (child_2 == -1) {
        // ошибка при создании процесса
        const char msg[] = "error: failed to fork\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    // находимся в родительском процессе
    bool running = true;
    while (running) {
        sem_wait(sem_server);

        uint32_t *length = (uint32_t *)shm_buf;
        char *text = shm_buf + sizeof(uint32_t);

        if (*length == UINT32_MAX) {
            // завершение программы
            running = false;
        }
        else if (*length > 0) {
            // значит данные получены от child_2
```

```
            const char msg[] = "Result: ";
            write(STDOUT_FILENO, msg, sizeof(msg) - 1);
            write(STDOUT_FILENO, text, *length);

            *length = 0;
            sem_post(sem_server);
        }

        else {
            // length == 0
            // значит обработанных данных нет, надо получить данные от пользователя
            const char msg[] = "Enter text (Ctrl+D for exit): ";
            write(STDOUT_FILENO, msg, sizeof(msg) - 1);

            char buf[SHM_SIZE - sizeof(uint32_t)];
            ssize_t bytes = read(STDIN_FILENO, buf, sizeof(buf));

            if (bytes == -1) {
                const char error[] = "error: failed to read from standart input\n";
                write(STDERR_FILENO, error, sizeof(error));
                _exit(EXIT_FAILURE);
            }

            if (bytes > 0) {
                *length = bytes;
                memcpy(text, buf, bytes);
                sem_post(sem_client_1);
            }
            else {
                *length = UINT32_MAX;
                running = false;
                sem_post(sem_client_1);
            }
        }
    }

    waitpid(child_1, NULL, 0);
    waitpid(child_2, NULL, 0);

    sem_unlink(SEM_NAME_SERVER);
    sem_unlink(SEM_NAME_CHILD_1);
    sem_unlink(SEM_NAME_CHILD_2);

    sem_close(sem_server);
    sem_close(sem_client_1);
    sem_close(sem_client_2);

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

    return 0;
}
```

child_1.c

```c
#include <fcntl.h>
#include <stdint.h>
#include <stdbool.h>

#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

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


#define SHM_SIZE 4096

const char SHM_NAME[] = "shm-name";
const char SEM_NAME_SERVER[] = "example-1-sem-server";
const char SEM_NAME_CHILD_1[] = "example-1-sem-child-1";
const char SEM_NAME_CHILD_2[] = "example-1-sem-child-2";

int main() {
    int shm = shm_open(SHM_NAME, O_RDWR, 0600);
    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\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    sem_t *sem_client_1 = sem_open(SEM_NAME_CHILD_1, O_RDWR);
    if (sem_client_1 == SEM_FAILED) {
        const char msg[] = "error: failed to open semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }
    sem_t *sem_client_2 = sem_open(SEM_NAME_CHILD_2, O_RDWR);
    if (sem_client_2 == SEM_FAILED) {
        const char msg[] = "error: failed to open semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }
```

```
    bool running = true;
    while (running) {
        sem_wait(sem_client_1);

        uint32_t *length = (uint32_t *)shm_buf;
        char *text = shm_buf + sizeof(uint32_t);

        if (*length == UINT32_MAX) {
            running = false;
        }
        else if (*length > 0) {
            // переводим в верхний регистр
            for (uint32_t i = 0; i < *length; ++i) {
                text[i] = toupper(text[i]);
            }
        }
        sem_post(sem_client_2);
    }

    sem_close(sem_client_1);
    sem_close(sem_client_2);

    munmap(shm_buf, SHM_SIZE);
    close(shm);

    return 0;
}
```

child_2.c

```
#include <fcntl.h>
#include <stdint.h>
#include <stdbool.h>

#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

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


#define SHM_SIZE 4096

const char SHM_NAME[] = "shm-name";
const char SEM_NAME_SERVER[] = "example-1-sem-server";
const char SEM_NAME_CHILD_1[] = "example-1-sem-child-1";
const char SEM_NAME_CHILD_2[] = "example-1-sem-child-2";

int main() {
```

```
    int shm = shm_open(SHM_NAME, O_RDWR, 0600);
    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\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    sem_t *sem_client_2 = sem_open(SEM_NAME_CHILD_2, O_RDWR);
    if (sem_client_2 == SEM_FAILED) {
        const char msg[] = "error: failed to open semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }
    sem_t *sem_server = sem_open(SEM_NAME_SERVER, O_RDWR);
    if (sem_server == SEM_FAILED) {
        const char msg[] = "error: failed to open semaphore\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    bool running = true;
    while (running) {
        sem_wait(sem_client_2);

        uint32_t *length = (uint32_t *)shm_buf;
        char *text = shm_buf + sizeof(uint32_t);

        if (*length == UINT32_MAX) {
            running = false;
        }
        else if (*length > 0) {
            // переводим в верхний регистр
            uint32_t new_length = 0;
            bool prev_symbol_is_space = 0;
            for (uint32_t i = 0; i < *length; ++i) {
                if (text[i] == ' ') {
                    if (!prev_symbol_is_space) {
                        text[new_length++] = text[i];
                        prev_symbol_is_space = true;
                    }
                }
                else {
                    text[new_length++] = text[i];
                    prev_symbol_is_space = false;
                }
            }
```

```
            *length = new_length;
        }
        sem_post(sem_server);
    }

    sem_close(sem_client_2);
    sem_close(sem_server);

    munmap(shm_buf, SHM_SIZE);
    close(shm);

    return 0;
}
```

Протокол работы программы

```
./server
Enter text (Ctrl+D for exit): vova   vova  lab_03
Result: VOVA VOVA LAB_03
Enter text (Ctrl+D for exit): lab3      vovaaaaaa lab_3  lab_3     lab_3
Result: LAB3 VOVAAAAAA LAB_3 LAB_3 LAB_3
Enter text (Ctrl+D for exit): %
```

Вывод

В ходе лабораторной работы я приобрел практические навыки при работе с разделяемой памятью, с ее отображением в адресное пространство процесса, а также потренировался в использовании семафоров.