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-210БВ-24

Студент: Белков А.Д.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 30.10.25

Москва, 2025

Постановка задачи

Вариант 11.

Child 1 переводит строки в верхний регистр. Child2 превращает все пробельные символы в символ ”_”.

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) - создаёт 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) - создаёт 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.

Я реализовал межпроцессное взаимодействие через разделяемую память и синхронизацию с помощью именованных семафоров. Серверный процесс создаёт shared memory объект и два дочерних процесса, которые последовательно обрабатывают данные: child_1 переводит текст в верхний регистр, child_2 заменяет пробельные символы на '_'. Три семафора обеспечивают корректную последовательность обработки и предотвращают гонку данных. Имена объектов генерируются динамически на основе PID для уникальности.

Код программы

server.c:

```c
#include <fcntl.h>

#include <stdint.h>

#include <stdbool.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <stdio.h>


#include <unistd.h>

#include <sys/fcntl.h>

#include <sys/mman.h>

#include <wait.h>

#include <semaphore.h>


#define SHM_SIZE 4096


char SHM_NAME[1024] = "shm-name";

char SEM_NAME_SERVER[1024] = "sem-server";
```

```
char SEM_NAME_CHILD_1[1024] = "sem-child-1";

char SEM_NAME_CHILD_2[1024] = "sem-child-2";


int main(void) {

    char unique_suffix[64];

    snprintf(unique_suffix, sizeof(unique_suffix), "%d", getpid());


    snprintf(SHM_NAME, sizeof(SHM_NAME), "shm_%s", unique_suffix);

    snprintf(SEM_NAME_SERVER, sizeof(SEM_NAME_SERVER), "sem_server_%s",
unique_suffix);

    snprintf(SEM_NAME_CHILD_1, sizeof(SEM_NAME_CHILD_1), "sem_child1_%s",
unique_suffix);

    snprintf(SEM_NAME_CHILD_2, sizeof(SEM_NAME_CHILD_2), "sem_child2_%s",
unique_suffix);


    int shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_TRUNC, 0600);


    // O_CREAT - создать, если не существует

    // O_TRUNC - если существует, то очистить


    if (shm == -1) {

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

        write(STDERR_FILENO, msg, sizeof(msg));

        _exit(EXIT_FAILURE);

    }


    // резервируем размер shared_memory перед mmap()

    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 память в адресное пространство процесса
    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);
    }



    // Делаем sem_server открытым, чтобы parent не блокировался на старте
    // и не завис до отправки строки
    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 sem_server\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }


    sem_t *sem_child_1 = sem_open(SEM_NAME_CHILD_1, O_RDWR | O_CREAT |
O_TRUNC, 0600, 0);
    if (sem_child_1 == SEM_FAILED) {
        const char msg[] = "error: failed to create semaphore sem_child_1\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }
```

```
    sem_t *sem_child_2 = sem_open(SEM_NAME_CHILD_2, O_RDWR | O_CREAT |
O_TRUNC, 0600, 0);

    if (sem_child_2 == SEM_FAILED) {

        const char msg[] = "error: failed to create semaphore sem_child_2\n";

        write(STDERR_FILENO, msg, sizeof(msg));

        _exit(EXIT_FAILURE);

    }


    // создаём child_1

    pid_t child1 = fork();

    if (child1 == 0) {

        char *args[] = {"child_1", SHM_NAME, SEM_NAME_SERVER,
SEM_NAME_CHILD_1, SEM_NAME_CHILD_2, NULL};

        execv("./child_1", args);

        const char msg[] = "error: failed to exec child_1\n";

        write(STDERR_FILENO, msg, sizeof(msg));

        _exit(EXIT_FAILURE);

    } else if (child1 == -1) {

        const char msg[] = "error: failed to fork child1\n";

        write(STDERR_FILENO, msg, sizeof(msg));

        _exit(EXIT_FAILURE);

    }


    // создаём child_2

    pid_t child2 = fork();

    if (child2 == 0) {

        char *args[] = {"child_2", SHM_NAME, SEM_NAME_SERVER,
SEM_NAME_CHILD_1, SEM_NAME_CHILD_2, NULL};

        execv("./child_2", args);

        const char msg[] = "error: failed to exec child_2\n";

        write(STDERR_FILENO, msg, sizeof(msg));
```

```
        _exit(EXIT_FAILURE);
    } else if (child2 == -1) {
        const char msg[] = "error: failed to fork child2\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    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) {
            running = false;
            sem_post(sem_server);
            break;
        }

        if (*length > 0) {
            // есть результат от child2 — выводим
            const char out[] = "Result: ";
            write(STDOUT_FILENO, out, sizeof(out) - 1);
            write(STDOUT_FILENO, text, *length);
            write(STDOUT_FILENO, "\n", 1);

            // сбрасываем длину для следующего ввода
            *length = 0;
```

```
            sem_post(sem_server); // освобождаем серверный семафор для ввода
            continue;
        }

        const char prom[] = "Enter text (Ctrl+Z to exit): ";
        write(STDOUT_FILENO, prom, sizeof(prom) - 1);
        char buf[SHM_SIZE - sizeof(uint32_t)];
        ssize_t bytes = read(STDIN_FILENO, buf, sizeof(buf));
        if (bytes == -1) {
            const char err[] = "error: failed to read stdin\n";
            write(STDERR_FILENO, err, sizeof(err));
            _exit(EXIT_FAILURE);
        }

        if (bytes > 0) {
            *length = (uint32_t)bytes;
            memcpy(text, buf, bytes);
            sem_post(sem_child_1); // запускаем child1 (child1 -> child2 ->
server)
        } else {
            *length = UINT32_MAX;
            sem_post(sem_child_1);
            running = false;
        }
    }

    waitpid(child1, NULL, 0);
    waitpid(child2, 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_child_1);
    sem_close(sem_child_2);

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

    return 0;
}
```

child_1.c:

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

#include <unistd.h>
#include <sys/fcntl.h>
```

```c
#include <sys/mman.h>
#include <semaphore.h>

#define SHM_SIZE 4096

int main(int argc, char **argv) {
    if (argc < 5) {
        const char msg[] = "error: not enough arguments\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    const char *SHM_NAME = argv[1];
    const char *SEM_NAME_SERVER = argv[2];
    const char *SEM_NAME_CHILD_1 = argv[3];
    const char *SEM_NAME_CHILD_2 = argv[4];

    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_child1 = sem_open(SEM_NAME_CHILD_1, O_RDWR);
sem_t *sem_child2 = sem_open(SEM_NAME_CHILD_2, O_RDWR);
if (sem_child1 == SEM_FAILED || sem_child2 == SEM_FAILED) {
    const char msg[] = "error: failed to open semaphores\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    _exit(EXIT_FAILURE);
}


bool running = true;
while (running) {
    sem_wait(sem_child1);

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

    if (*length == UINT32_MAX) {
        // чтобы child_2 разблокировался и вышел из своего цикла
        sem_post(sem_child2);
        running = false;
        break;
    }

    if (*length > 0) {
        // переводим в верхний регистр
        for (uint32_t i = 0; i < *length; ++i) {
            text[i] = (char)toupper((unsigned char)text[i]);
        }
```

```
        }


        // передаём управление child2
        sem_post(sem_child2);
    }


    sem_close(sem_child1);
    sem_close(sem_child2);
    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 <stdio.h>

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

#define SHM_SIZE 4096
```

```c
int main(int argc, char **argv) {
    if (argc < 5) {
        const char msg[] = "error: not enough arguments\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        _exit(EXIT_FAILURE);
    }

    const char *SHM_NAME = argv[1];
    const char *SEM_NAME_SERVER = argv[2];
    const char *SEM_NAME_CHILD_1 = argv[3];
    const char *SEM_NAME_CHILD_2 = argv[4];

    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_child2 = sem_open(SEM_NAME_CHILD_2, O_RDWR);
    sem_t *sem_server = sem_open(SEM_NAME_SERVER, O_RDWR);
    if (sem_child2 == SEM_FAILED || sem_server == SEM_FAILED) {
```

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

bool running = true;
while (running) {
    sem_wait(sem_child2);

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

    if (*length == UINT32_MAX) {
        // сообщаем серверу, что пора завершаться
        sem_post(sem_server);
        running = false;
        break;
    }

    if (*length > 0) {
        // заменяем пробельные символы на '_'
        for (uint32_t i = 0; i < *length; ++i) {
            if (text[i] == ' ' || text[i] == '\t') {
                text[i] = '_';
            }
        }
    }

    // передаём управление серверу
```

```
        sem_post(sem_server);
    }


    sem_close(sem_child2);
    sem_close(sem_server);
    munmap(shm_buf, SHM_SIZE);
    close(shm);
    return 0;
}
```

Протокол работы программы

Тестирование:

```
artmlink@pop-os:~/2_course_MAI/MAI-OS-Labs-2025/lab-3$ ./server
Enter text (Ctrl+Z to exit): string1
Result: STRING1

Enter text (Ctrl+Z to exit): string2
Result: STRING2_

Enter text (Ctrl+Z to exit): String with spaces
Result: STRING_WITH_SPACES_

Enter text (Ctrl+Z to exit): maimaimaimaima amaim maimaimaimaimaimai
Result: MAIMAIMAIMAIMA_AMAIM_MAIMAIMAIMAIMAIMAI____

Enter text (Ctrl+Z to exit):    zzz    zzzz
Result: ___ZZZ____ZZZZ
```

```
artmlink@pop-os:~/2_course_MAI/MAI-OS-Labs-2025/lab-3$ ./server
Enter text (Ctrl+Z to exit): string1
Result: STRING1

Enter text (Ctrl+Z to exit): string2
Result: STRING2_

Enter text (Ctrl+Z to exit): String with spaces
Result: STRING_WITH_SPACES_

Enter text (Ctrl+Z to exit): maimaimaimaima amaim maimaimaimaimaimai
Result: MAIMAIMAIMAIMA_AMAIM_MAIMAIMAIMAIMAIMAI____

Enter text (Ctrl+Z to exit):    zzz    zzzz
Result: ___ZZZ____ZZZZ
```

Вывод

В ходе работы я освоил альтернативный способ межпроцессного взаимодействия через разделяемую память вместо каналов. Реализовал создание именованных shared memory объектов и их синхронизацию с помощью семафоров. Научился динамически генерировать уникальные имена объектов на основе PID процесса для возможности одновременного запуска нескольких экземпляров программы.