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: Мицкевич А.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 13.11.25

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:

- int ftruncate(int fd, off_t length); – устанавливает размер объекта разделяемой памяти (или файла). Возвращает 0 при успехе, -1 при ошибке.
- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset); – отображает разделяемую память в адресное пространство процесса. Возвращает указатель на отображённую область или MAP_FAILED в случае ошибки.
- int munmap(void *addr, size_t length); – удаляет ранее созданное отображение памяти. Возвращает 0 при успехе, -1 при ошибке.
- int shm_unlink(const char *name); – удаляет именованный объект разделяемой памяти из системы. Возвращает 0 при успехе, -1 при ошибке.

Системные вызовы для работы с именованными семафорами:

- sem_t *sem_open(const char *name, int oflag, ...); – открывает или создаёт именованный семафор. Возвращает указатель на семафор или SEM_FAILED при ошибке.
- int sem_wait(sem_t *sem); – ожидает освобождения семафора (уменьшает его значение на 1). Возвращает 0 при успехе, -1 при ошибке.
- int sem_post(sem_t *sem); – освобождает семафор (увеличивает его значение на 1). Возвращает 0 при успехе, -1 при ошибке.
- int sem_close(sem_t *sem); – закрывает дескриптор семафора в текущем процессе. Возвращает 0 при успехе, -1 при ошибке.
- int sem_unlink(const char *name); – удаляет именованный семафор из системы. Возвращает 0 при успехе, -1 при ошибке.

Системные вызовы для управления процессами:

- pid_t fork(void); – создаёт новый процесс-потомок.
- int execv(const char *path, char *const argv[]); – заменяет текущий процесс новым, запуская указанную программу.
- pid_t waitpid(pid_t pid, int *wstatus, int options); – ожидает завершения конкретного дочернего процесса.

Функции для работы с файлами и вводом-выводом:

• ssize_t write(int fd, const void *buf, size_t count); – записывает данные из буфера по указанному файловому дескриптору.

• ssize_t read(int fd, void *buf, size_t count); – читает данные из файла по файловому дескриптору и сохраняет в буфер.

• int close(int fd); – закрывает файловый дескриптор. Возвращает 0 при успехе, -1 при ошибке.

Межпроцессное взаимодействие реализовано с использованием именованной разделяемой памяти и именованных семафоров. Родительский процесс создаёт два дочерних: первый преобразует все символы текста в верхний регистр, второй – удаляет лишние пробелы. Обмен данными между процессами происходит через общую область памяти, созданную с помощью shm_open и отображённую в адресное пространство каждого процесса через mmap. Синхронизация доступа к разделяемой памяти обеспечивается с помощью семафоров, создаваемых через sem_open. Взаимодействие с пользователем (ввод текста и вывод результата) осуществляется исключительно родительским процессом.

Код программы

client.c

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


#define SHM_SIZE 4096


char SHM_NAME[1024] = "shm-name";
char SEM_CLIENT[1024] = "sem-client";
char SEM_PROCESSOR_1[1024] = "sem-processor1";
char SEM_PROCESSOR_2[1024] = "sem-processor2";


int main() {
    char unique_suffix[64] = "\0";
    snprintf(unique_suffix, sizeof(unique_suffix), "%d", getpid());



    snprintf(SHM_NAME, sizeof(SHM_NAME), "/shm-name_%s", unique_suffix);
    snprintf(SEM_CLIENT, sizeof(SEM_CLIENT), "/sem-client_%s", unique_suffix);
```

```
    snprintf(SEM_PROCESSOR_1, sizeof(SEM_PROCESSOR_1), "/sem-processor1_%s",
unique_suffix);
    snprintf(SEM_PROCESSOR_2, sizeof(SEM_PROCESSOR_2), "/sem-processor2_%s",
unique_suffix);



    int shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_TRUNC, 0600);
    if (shm < 0) {
        const char error[] = "Error: Failed to open SHM\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    if (ftruncate(shm, SHM_SIZE) < 0) {
        const char error[] = "Error: Failed to resize SHM\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    char * shm_buffer = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm, 0);
    if (shm_buffer == MAP_FAILED) {                              // отображает в виртуальное
адресное пространство процесса, чтобы
        const char error[] = "Error: Failed to map SHM\n";      // можно было работать как
с обычным участком памяти (через указатель)
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }



    sem_t * sem_wait_for_processor2 = sem_open(SEM_CLIENT, O_RDWR | O_CREAT | O_TRUNC,
0600, 1);
    if (sem_wait_for_processor2 == SEM_FAILED) {
        const char error[] = "Error: Failed to create semaphore (client)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    sem_t * sem_signal_to_processor1 = sem_open(SEM_PROCESSOR_1, O_RDWR | O_CREAT |
O_TRUNC, 0600, 0);
    if (sem_signal_to_processor1 == SEM_FAILED) {
        const char error[] = "Error: Failed to create semaphore (processor 1)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    sem_t * sem_processor_2 = sem_open(SEM_PROCESSOR_2, O_RDWR | O_CREAT | O_TRUNC, 0600,
0);
    if (sem_processor_2 == SEM_FAILED) {
        const char error[] = "Error: Failed to create semaphore (processor 2)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }
```

```
    pid_t child1 = fork();
    if (child1 == 0) {
        char * args[] = {"processor1", SHM_NAME, SEM_CLIENT, SEM_PROCESSOR_1,
SEM_PROCESSOR_2, NULL};
        execv("./processor1", args);

        const char error[] = "Error: Failed to exec (processor 1)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    } else if (child1 == -1) {
        const char error[] = "Error: Failed to fork (processor 1)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }



    pid_t child2 = fork();
    if (child2 == 0) {
        char * args[] = {"processor2", SHM_NAME, SEM_CLIENT, SEM_PROCESSOR_1,
SEM_PROCESSOR_2, NULL};
        execv("./processor2", args);

        const char error[] = "Error: failed to exec (processor 2)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    } else if (child2 == -1) {
        const char error[] = "Error: Failed to fork (processor 2)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }



    bool running = true;
    while (running) {
        if (sem_wait(sem_wait_for_processor2) < 0) {
            const char error[] = "Error: sem_wait failed\n";
            write(STDERR_FILENO, error, sizeof(error));
            _exit(EXIT_FAILURE);
        }

        int * length = (int *)shm_buffer;
        char * text = shm_buffer + sizeof(int);

        if (*length == -1) {
            running = false;
        } else if (*length > 0) {                              // длина изменилась => это
результат работы processor2
            const char message[] = "Result: ";
            write(STDOUT_FILENO, message, sizeof(message) - 1);
```

```
            write(STDOUT_FILENO, text, *length);

            *length = 0;
            sem_post(sem_wait_for_processor2);
        } else {                                                // длина 0 => данных нет =>
надо получить их от пользователя
            const char message[] = "Enter text (Ctrl+D for exit): ";
            write(STDOUT_FILENO, message, sizeof(message) - 1);

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

            if (bytes < 0) {
                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_signal_to_processor1);
            } else {
                *length = -1;
                running = false;
                sem_post(sem_signal_to_processor1);
            }
        }
    }

    waitpid(child1, NULL, 0);
    waitpid(child2, NULL, 0);

    sem_unlink(SEM_CLIENT);
    sem_unlink(SEM_PROCESSOR_1);
    sem_unlink(SEM_PROCESSOR_2);

    sem_close(sem_wait_for_processor2);
    sem_close(sem_signal_to_processor1);
    sem_close(sem_processor_2);

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

    return 0;
}
```

processor1.c

```
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdbool.h>
```

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


#define SHM_SIZE 4096


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


    const char * SHM_NAME = argv[1];
    const char * SEM_CLIENT = argv[2];
    const char * SEM_PROCESSOR_1 = argv[3];
    const char * SEM_PROCESSOR_2 = argv[4];


    int shm_fd = shm_open(SHM_NAME, O_RDWR, 0600);
    if (shm_fd < 0) {
        const char error[] = "Error: Failed to open SHM\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    char * shm_buffer = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd,
0);
    if (shm_buffer == MAP_FAILED) {
        const char error[] = "Error: Failed to map shared memory\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    sem_t * sem_wait_for_client = sem_open(SEM_PROCESSOR_1, O_RDWR);
    if (sem_wait_for_client == SEM_FAILED) {
        const char error[] = "Error: Failed to open semaphore (processor 1)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    sem_t * sem_signal_to_processor2 = sem_open(SEM_PROCESSOR_2, O_RDWR);
    if (sem_signal_to_processor2 == SEM_FAILED) {
        const char error[] = "Error: Failed to open semaphore (processor 2)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    bool running = true;
    while (running) {
        if (sem_wait(sem_wait_for_client) < 0) {
            const char error[] = "Error: sem_wait failed\n";
```

```
            write(STDERR_FILENO, error, sizeof(error));
            _exit(EXIT_FAILURE);
        }

        int * length_ptr = (int *)shm_buffer;
        int length = *length_ptr;
        char * text = shm_buffer + sizeof(int);

        if (length == -1) {
            running = false;
        } else if (length > 0) {
            for (int i = 0; i < length; i++) {
                text[i] = toupper(text[i]);
            }
        }

        if (sem_post(sem_signal_to_processor2) < 0) {
            const char error[] = "Error: sem_post failed\n";
            write(STDERR_FILENO, error, sizeof(error));
            _exit(EXIT_FAILURE);
        }
    }

    sem_close(sem_wait_for_client);
    sem_close(sem_signal_to_processor2);

    munmap(shm_buffer, SHM_SIZE);
    close(shm_fd);

    return 0;
}
```

processor2.c

```
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdbool.h>
#include <sys/mman.h>
#include <semaphore.h>


#define SHM_SIZE 4096


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


    const char * SHM_NAME = argv[1];
```

```
    const char * SEM_CLIENT = argv[2];
    const char * SEM_PROCESSOR_1 = argv[3];
    const char * SEM_PROCESSOR_2 = argv[4];


    int shm_fd = shm_open(SHM_NAME, O_RDWR, 0600);
    if (shm_fd < 0) {
        const char error[] = "Error: Failed to open SHM\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    char * shm_buffer = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd,
0);
    if (shm_buffer == MAP_FAILED) {
        const char error[] = "Error: Failed to map shared memory\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    sem_t * sem_wait_for_processor1 = sem_open(SEM_PROCESSOR_2, O_RDWR);
    if (sem_wait_for_processor1 == SEM_FAILED) {
        const char error[] = "Error: Failed to open semaphore (processor 2)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    sem_t * sem_signal_to_client = sem_open(SEM_CLIENT, O_RDWR);
    if (sem_signal_to_client == SEM_FAILED) {
        const char error[] = "Error: Failed to open semaphore (client)\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }

    bool running = true;
    while (running) {
        if (sem_wait(sem_wait_for_processor1) < 0) {
            const char error[] = "Error: sem_wait failed\n";
            write(STDERR_FILENO, error, sizeof(error));
            _exit(EXIT_FAILURE);
        }

        int * length = (int *)shm_buffer;
        char * text = shm_buffer + sizeof(int);

        if (*length == -1) {
            running = false;
        } else if (*length > 0) {
            int new_length = 0;
            bool prev_is_space = 0;
            for (int i = 0; i < *length; ++i) {
                if (text[i] == ' ') {
                    if (!prev_is_space) {
                        text[new_length++] = text[i];
```

```
                    prev_is_space = true;
                }
            }
            else {
                text[new_length++] = text[i];
                prev_is_space = false;
            }
        }

        *length = new_length;
    }

    if (sem_post(sem_signal_to_client) < 0) {
        const char error[] = "Error: sem_post failed\n";
        write(STDERR_FILENO, error, sizeof(error));
        _exit(EXIT_FAILURE);
    }
}

sem_close(sem_wait_for_processor1);
sem_close(sem_signal_to_client);

munmap(shm_buffer, SHM_SIZE);
close(shm_fd);

return 0;
}
```

Протокол работы программы

```
massakazu@WIN-IJEE655BLAV:~/lab$ ./client
Enter text (Ctrl+D for exit): Hello,    world!
Result: HELLO, WORLD!
```

Вывод

В ходе выполнения лабораторной работы было реализовано межпроцессное взаимодействие в Unix-подобной операционной системе с использованием именованной разделяемой памяти и именованных семафоров. Родительский процесс создаёт два дочерних, каждый из которых выполняет отдельную обработку текста: первый преобразует все символы в верхний регистр, второй удаляет лишние пробелы. Обмен данными осуществляется через общую область памяти, созданную с помощью shm_open и отображённую в адресное пространство процессов через mmap, а синхронизация доступа к ней обеспечивается семафорами, созданными с помощью sem_open. Такой подход позволяет эффективно координировать работу процессов и безопасно разделять данные без использования каналов или копирования.