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: Николич С.Р.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 12.11.25

Москва, 2025

Постановка задачи

Вариант 20.

Группа вариантов 5: Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в shared memory с использованием семафоров для синхронизации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в файлы.

Вариант 20) Правило фильтрации: строки длины больше 10 символов отправляются в область shared memory для child2, иначе в область для child1. Дочерние процессы инвертируют строки.

Общий метод и алгоритм решения

Использованные системные вызовы:

• pid_t fork(void) – создает дочерний процесс. После вызова оба процесса выполняют один и тот же код, но в родительском процессе возвращается PID ребенка, а в дочернем процессе возвращается 0.

• int shm_open(const char name, int oflag, mode_t mode) – создает или открывает объект разделяемой памяти. Флаги O_CREAT | O_RDWR | O_EXCL обеспечивают создание нового уникального объекта с доступом на чтение и запись.

• int ftruncate(int fd, off_t length) – устанавливает размер shared memory объекта.

• void mmap(void addr, size_t length, int prot, int flags, int fd, off_t offset) – отображает shared memory в адресное пространство процесса. Флаг MAP_SHARED обеспечивает видимость изменений между процессами.

• int munmap(void addr, size_t length) – удаляет отображение памяти.

• int shm_unlink(const char name) – удаляет объект разделяемой памяти из системы.

• sem_t sem_open(const char name, int oflag, mode_t mode, unsigned int value) – создает или открывает именованный семафор. Начальное значение 1 обеспечивает мьютекс.

• int sem_wait(sem_t sem) – атомарно уменьшает значение семафора. Если значение равно 0, блокирует процесс до его увеличения.

• int sem_post(sem_t sem) – атомарно увеличивает значение семафора и разблокирует ожидающие процессы.

• int sem_close(sem_t sem) – закрывает семафор.

• int sem_unlink(const char name) – удаляет именованный семафор из системы.

• int execv(const char path, char const argv[]) – заменяет текущий образ процесса новым образом из указанного исполняемого файла.

• pid_t waitpid(pid_t pid, int status, int options) – ожидает завершения указанного дочернего процесса и получает информацию о его завершении.

• FILE *fopen(const char pathname, const char mode) – открывает файл для записи.

• char *fgets(char s, int size, FILE stream) – читает строку из стандартного ввода.

В рамках лабораторной работы программа реализует многопроцессную обработку строк с использованием межпроцессного взаимодействия через разделяемую память (shared memory) и семафоры для синхронизации. Родительский процесс создает два дочерних процесса, объект разделяемой памяти и именованный семафор. Пользователь вводит имена файлов для каждого дочернего процесса. Родительский процесс читает строки из стандартного ввода, анализирует их длину и отправляет в соответствующую область разделяемой памяти: короткие строки (≤10 символов) в text1 для child1, длинные строки (>10 символов) в text2 для child2. Для синхронизации доступа к разделяемой памяти используется семафор, который захватывается перед записью или чтением и освобождается после операции. Дочерние процессы открывают существующие объекты разделяемой памяти и семафора, читают строки из своих областей памяти, инвертируют их и записывают результаты в указанные файлы. Когда пользователь завершает ввод (Ctrl+D), родительский процесс устанавливает флаг parent_finished, что сигнализирует дочерним процессам о завершении данных. Дочерние процессы завершают работу после обработки всех данных, а родительский процесс ожидает их завершения с помощью waitpid перед своим завершением и освобождением системных ресурсов.

Код программы

parent.c

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

#define SHM_SIZE 4096 //Размер для виртуальной памяти
#define BUFFER_SIZE 1024 //Размер строк для ввода

struct shared_data{
    uint32_t length1;// длина строки для child1
    uint32_t length2; // длина строки для child2
    bool parent_finished; // флаг завершения родителя
    char text1[BUFFER_SIZE];// строка для child1
    char text2[BUFFER_SIZE];// строка для child2
};

//Уникальное имя для системных ресурсов
void generate_resource_names(char *shm_name, char *sem_name, size_t size){
    pid_t pid = getpid();
    snprintf(shm_name, size, "/lab3-shm-%d", pid);
    snprintf(sem_name, size, "/lab3-sem-%d", pid);
}

int main(){
    char shm_name[64], sem_name[64];
```

```
    generate_resource_names(shm_name, sem_name, sizeof(shm_name));

    printf("SHM name: %s\n", shm_name);
    printf("SEM name: %s\n", sem_name);


    char filename1[256], filename2[256];
    printf("Введите имя файла для child1: ");
    if(fgets(filename1, sizeof(filename1), stdin) == NULL){
        perror("fgets failed");
        exit(EXIT_FAILURE);
    }

    filename1[strcspn(filename1, "\n")] = 0;

    printf("Введите имя файла для child2: ");
    if(fgets(filename2, sizeof(filename2), stdin) == NULL){
        perror("fgets failed");
        exit(EXIT_FAILURE);
    }

    filename2[strcspn(filename2, "\n")] = 0;


    // Создание shared memory (shm_open файловый дескриптор или -1 при ошибке)
    int shm_fd = shm_open(shm_name, O_RDWR | O_CREAT | O_EXCL, 0600);
    if(shm_fd == -1){
        perror("shm_open failed");
        exit(EXIT_FAILURE);
    }
    //Устанавливаем размер shared memory
    if(ftruncate(shm_fd, SHM_SIZE) == -1){
        perror("ftruncate failed");
        shm_unlink(shm_name);//Удаление объекта
        exit(EXIT_FAILURE);
    }

    //mmap - отображает shared memory в адресное пространство процесса
    struct shared_data *shm = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);
    //MAP_SHARED - изменения видны другим процессам, 0 - смещение от начала
    if (shm == MAP_FAILED){
        perror("mmap failed");
        shm_unlink(shm_name);
        exit(EXIT_FAILURE);
    }

    // Инициализация shared memory
    shm->length1 = 0;
    shm->length2 = 0;
    shm->parent_finished = false;

    // Семафор
    // 1 - начальное значение семафора (доступен)
```

```
// Возвращает: указатель на семафор или SEM_FAILED
sem_t *sem = sem_open(sem_name, O_CREAT | O_EXCL, 0600, 1);
if(sem == SEM_FAILED){
    perror("sem_open failed");
    munmap(shm, SHM_SIZE); //убирает отображение памяти
    shm_unlink(shm_name);
    exit(EXIT_FAILURE);
}


pid_t child1 = fork();
if (child1 == -1){
    perror("fork failed");
    sem_close(sem);
    sem_unlink(sem_name);
    munmap(shm, SHM_SIZE);
    shm_unlink(shm_name);
    exit(EXIT_FAILURE);
}

if (child1 == 0){
    //Массив аргументов командной строки для execv
    char *args[] = {"./child1", shm_name, sem_name, filename1, NULL};
    execv(args[0], args);
    perror("execv failed");
    _exit(EXIT_FAILURE);
}

pid_t child2 = fork();
if (child2 == -1){
    perror("fork failed");
    sem_close(sem);
    sem_unlink(sem_name);
    munmap(shm, SHM_SIZE);
    shm_unlink(shm_name);
    exit(EXIT_FAILURE);
}

if (child2 == 0){
    char *args[] = {"./child2", shm_name, sem_name, filename2, NULL};
    execv(args[0], args);
    perror("execv failed");
    _exit(EXIT_FAILURE);
}

printf("Вводите строки (Ctrl+D для завершения):\n");
char buffer[BUFFER_SIZE];

while (fgets(buffer, sizeof(buffer), stdin) != NULL){
    buffer[strcspn(buffer, "\n")] = 0;
    size_t len = strlen(buffer);

    sem_wait(sem);// Захват семафора
```

```
        if (len > 10){
            shm->length2 = len;
            memcpy(shm->text2, buffer, len + 1);
            printf("Parent: отправлена длинная строка в child2: '%s'\n", buffer);
        }
        else{
            shm->length1 = len;
            memcpy(shm->text1, buffer, len + 1);
            printf("Parent: отправлена короткая строка в child1: '%s'\n", buffer);
        }

        sem_post(sem);// Освобождение семафора

        usleep(10000);//Это время для того чтобы дети смогли обработать данные
    }

    sem_wait(sem);
    shm->parent_finished = true;
    sem_post(sem);

    printf("Parent: ожидание завершения дочерних процессов...\n");

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

    sem_close(sem);
    sem_unlink(sem_name);
    munmap(shm, SHM_SIZE);
    shm_unlink(shm_name);
    close(shm_fd);

    printf("Родительский процесс завершен.\n");
    return 0;
}
```

child1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdbool.h>
#include <stdint.h>

#define SHM_SIZE 4096
#define BUFFER_SIZE 1024

struct shared_data{
    uint32_t length1;
    uint32_t length2;
```

```
    bool parent_finished;
    char text1[BUFFER_SIZE];
    char text2[BUFFER_SIZE];
};

void invert_string(char *str){
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        char temp = str[i];
        str[i] = str[len - i - 1];
        str[len - i - 1] = temp;
    }
}

int main(int argc, char *argv[]){
    if (argc != 4){
        fprintf(stderr, "Usage: %s shm_name sem_name filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    char *shm_name = argv[1];
    char *sem_name = argv[2];
    char *filename = argv[3];

    printf("Child1: запущен для файла %s\n", filename);

    int shm_fd = shm_open(shm_name, O_RDWR, 0);
    if (shm_fd == -1){
        perror("Child1: shm_open failed");
        exit(EXIT_FAILURE);
    }

    struct shared_data *shm = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);
    if (shm == MAP_FAILED){
        perror("Child1: mmap failed");
        close(shm_fd);
        exit(EXIT_FAILURE);
    }

    sem_t *sem = sem_open(sem_name, 0);
    if (sem == SEM_FAILED){
        perror("Child1: sem_open failed");
        munmap(shm, SHM_SIZE);
        close(shm_fd);
        exit(EXIT_FAILURE);
    }

    FILE *file = fopen(filename, "w");
    if (file == NULL){
        perror("Child1: fopen failed");
        sem_close(sem);
        munmap(shm, SHM_SIZE);
        close(shm_fd);
```

```
        exit(EXIT_FAILURE);
    }

    bool running = true;
    printf("Child1: начало обработки коротких строк в файл %s\n", filename);

    while (running){
        sem_wait(sem);

        if (shm->length1 > 0){
            char inverted[BUFFER_SIZE];
            strcpy(inverted, shm->text1);
            invert_string(inverted);

            fprintf(file, "%s\n", inverted);
            // fflush - сбрасывает буфер файла (данные сразу записываются)
            fflush(file);

            printf("Child1: '%s' -> '%s'\n", shm->text1, inverted);

            // Сброс длины для сигнализации о обработке
            shm->length1 = 0;
        }

        if (shm->parent_finished && shm->length1 == 0){
            running = false;
        }

        sem_post(sem);

        usleep(50000); // Небольшая задержка для уменьшения нагрузки на CPU
    }

    fclose(file);
    sem_close(sem);
    munmap(shm, SHM_SIZE);
    close(shm_fd);

    printf("Child1: завершил работу\n");
    return 0;
}
```

child2.c

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

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

#define SHM_SIZE 4096
#define BUFFER_SIZE 1024

struct shared_data{
    uint32_t length1;
    uint32_t length2;
    bool parent_finished;
    char text1[BUFFER_SIZE];
    char text2[BUFFER_SIZE];
};

void invert_string(char *str){
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++){
        char temp = str[i];
        str[i] = str[len - i - 1];
        str[len - i - 1] = temp;
    }
}

int main(int argc, char *argv[]){
    if (argc != 4){
        fprintf(stderr, "Usage: %s shm_name sem_name filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    char *shm_name = argv[1];
    char *sem_name = argv[2];
    char *filename = argv[3];

    printf("Child2: запущен для файла %s\n", filename);

    int shm_fd = shm_open(shm_name, O_RDWR, 0);
    if (shm_fd == -1) {
        perror("Child2: shm_open failed");
        exit(EXIT_FAILURE);
    }

    struct shared_data *shm = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);
    if (shm == MAP_FAILED){
        perror("Child2: mmap failed");
        close(shm_fd);
        exit(EXIT_FAILURE);
    }

    sem_t *sem = sem_open(sem_name, 0);
    if (sem == SEM_FAILED){
        perror("Child2: sem_open failed");
        munmap(shm, SHM_SIZE);
        close(shm_fd);
```

```
        exit(EXIT_FAILURE);
    }

    FILE *file = fopen(filename, "w");
    if (file == NULL){
        perror("Child2: fopen failed");
        sem_close(sem);
        munmap(shm, SHM_SIZE);
        close(shm_fd);
        exit(EXIT_FAILURE);
    }

    bool running = true;
    printf("Child2: начало обработки длинных строк в файл %s\n", filename);

    while (running){
        sem_wait(sem);

        if (shm->length2 > 0){
            char inverted[BUFFER_SIZE];
            strcpy(inverted, shm->text2);
            invert_string(inverted);

            fprintf(file, "%s\n", inverted);
            fflush(file);

            printf("Child2: '%s' -> '%s'\n", shm->text2, inverted);

            shm->length2 = 0;
        }

        if (shm->parent_finished && shm->length2 == 0){
            running = false;
        }

        sem_post(sem);

        usleep(50000);
    }

    fclose(file);
    sem_close(sem);
    munmap(shm, SHM_SIZE);
    close(shm_fd);

    printf("Child2: завершил работу\n");
    return 0;
}
```

Протокол работы программы

```
savva@Honorlaptop:~/OS/lab3$ ./parent
SHM name: /lab3-shm-1001
SEM name: /lab3-sem-1001
Введите имя файла для child1: 1.txt
Введите имя файла для child2: 2.txt
Вводите строки (Ctrl+D для завершения):
Child1: запущен для файла 1.txt
Child2: запущен для файла 2.txt
Child1: начало обработки коротких строк в файл 1.txt
Child2: начало обработки длинных строк в файл 2.txt
11111111
Parent: отправлена короткая строка в child1: '11111111'
Child1: '11111111' -> '11111111'
1111111111111111111
Parent: отправлена длинная строка в child2: '1111111111111111111'
Child2: '1111111111111111111' -> '1111111111111111111'
         hgdsfgsdfj
Parent: отправлена длинная строка в child2: '         hgdsfgsdfj '
Child2: '         hgdsfgsdfj ' -> ' jfdsgfsdgh         '
jjkkhhhhhh
Parent: отправлена короткая строка в child1: 'jjkkhhhhhh'
Child1: 'jjkkhhhhhh' -> 'hkhhhhhkkjj'
55555555555555
Parent: отправлена длинная строка в child2: '55555555555555'
Child2: '55555555555555' -> '55555555555555'
00
Parent: отправлена короткая строка в child1: '00'
Child1: '00' -> '00'
Parent: ожидание завершения дочерних процессов...
Child2: завершил работу
Child1: завершил работу
Родительский процесс завершен.
savva@Honorlaptop:~/OS/lab3$
```

Вывод

В данной лабораторной работе я освоил управление процессами в операционной системе и организацию межпроцессного взаимодействия через механизм разделяемой памяти. Была реализована многопроцессная архитектура, где родительский процесс распределяет данные между дочерними процессами через общую память с использованием семафоров для синхронизации. Каждый дочерний процесс выполнял независимую обработку строк, что позволило организовать параллельное выполнение задач. В ходе работы я научился работать с системными вызовами для управления shared memory, семафорами и процессами, а также освоил методы обработки ошибок при межпроцессном взаимодействии.