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: Кудрявцев Е.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 29.09.25

Москва, 2025

Постановка задачи

Вариант 13.

Родительский процесс создает два дочерних процесса. Перенаправление стандартных потоков ввода-вывода показано на картинке выше. Child1 и Child2 можно «соединить» между собой дополнительным каналом. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1. Процесс child1 и child2 производят работу над строками. Child2 пересылает результат своей работы родительскому процессу. Родительский процесс полученный результат выводит в стандартный поток вывода Child1 переводит строки в нижний регистр. Child2 превращает все пробельные символы в символ «_»

При выполнении работы опираться на техническое задание (далее приведена прямая цитата).

Нужно взять свою первую лабу и переделать её с использованием shared memory и memory mapping. Варианты остаются те же, что и у первой лабораторной.

Так как блокирующего чтения из каналов у вас больше не будет, то для синхронизации чтения и записи из shared memory будем использовать семафор.

Для тех, кто будет писать под Windows, прилагаю список системных функций, которые вам понадобятся: CreateFileMapping, MapViewOfFile, UnmapViewOfFile, OpenSemaphore, WaitForSingleObject, ReleaseSemaphore

Аналогично, для Linux: shm_open, shm_unlink, ftruncate, mmap, munmap, sem_open, sem_wait, sem_post, sem_unlink, sem_close

Отдельно отмечу, что shared memory и memory mapping это не одно и то же. В этой лабораторной работе вам нужно создать именованый shared memory объект, он будет находиться именно в оперативной памяти, а не на вашем диске в виде файлика

Поскольку вам нужно будет создавать именованые shared memory и семафоры, то их название должно отличаться для экземпляров пар программ server и client. Не делайте название именованных shared memory и семафоров константным!

Общий метод и алгоритм решения

Цель лабораторной работы

Целью программы являлось изучение механизмов межпроцессного взаимодействия в POSIX (общая именованная shared memory, именованные семафоры), управление дочерними процессами и организация простого протокола передачи сообщений между процессами. Программа реализует процесс-родитель, запускающий два дочерних процесса (child1 и child2), и обеспечивает последовательный обмен строковыми сообщениями через три сегмента разделяемой памяти и три семафора.

Краткое описание работы программы

Программа создаёт три именованных сегмента разделяемой памяти и три именованных семафора, затем порождает два дочерних процесса (child1 и child2) с помощью fork() + execl() (ожидая, что соответствующие бинарники ./child1 и ./child2 присутствуют). После этого родитель читает строки с stdin и пересылает их в child1 через shm+sem; child1 по протоколу пересылает данные child2 и т.д., а результат от child2 поступает обратно родителю, который выводит его на stdout. Пустая строка (нажатие Enter без текста) служит сигналом завершения — родитель отправляет нулевую длину и инициирует завершение цепочки.

Реализованные режимы / поведение (логика)

1. Инициализация:
    - Формируются уникальные имена для shm и семафоров с суффиксом PID родителя.
    - Создаются и отображаются в память три shm-сегмента размером SHM_SIZE (8192 байт).
    - Создаются три именованных семафора с начальным значением 0: sem_p_c1, sem_c1_c2, sem_c2_p.
2. Порождаются дочерние процессы:
    - child1 запускается с аргументами: имена shm p->c1 и c1->c2 и имена семафоров p->c1 и c1->c2.
    - child2 запускается с аргументами: имена shm c1->c2 и c2->p и имена семафоров c1->c2 и c2->p.
3. Протокол обмена (в цикле):
    - Родитель читает строку с stdin (read_line()).
    - Если строка пустая (len == 0) — в shm записывается uint32_t msg_len = 0 и sem_post(sem_p_c1) — это сигнал завершения; цикл прерывается.
    - Иначе: в сегмент shm_p_c1 записывается 4-байтовая длина (uint32_t) и за ней — msg_len байт текста; затем sem_post(sem_p_c1) сообщает child1.
    - Родитель ждёт sem_wait(sem_c2_p), затем читает ответ из shm_c2_p: сначала uint32_t rlen, затем rlen байт данных, которые выводит в stdout.

- Процесс повторяется до завершения.

4. Завершение:
    - Родитель ждёт завершения дочерних (waitpid), удаляет shm и семафоры (munmap, close, shm_unlink, sem_close, sem_unlink) и выходит.

Формат сообщений в shm

Каждый сегмент использует простой кадрированный протокол:

- Первые 4 байта — uint32_t длина сообщения (в байтах).
- Следующие length байт — полезная нагрузка (не NUL-terminated). Ограничение: length + sizeof(uint32_t) не должно превышать SHM_SIZE (8192).

Описание ключевых функций

- write_all(int fd, const void *buf, size_t len)
  — безопасная запись в файловый дескриптор: циклически вызывает write, обрабатывает EINTR, на ошибку вызывает _exit(2).
- fatal_errno_exit(const char *prefix)
  — печатает prefix: strerror(errno) в stderr и вызывает _exit(EXIT_FAILURE).
- fatal_msg_exit(const char *msg)
  — печатает сообщение и завершает процесс _exit(EXIT_FAILURE).
- create_and_map_shm(const char *name, size_t size, void **out_addr, int *out_fd)
  — создаёт именованный shm через shm_open(O_CREAT|O_EXCL), задаёт размер ftruncate, отображает mmap с MAP_SHARED. Возвращает 0 при успехе, -1 при ошибке (в этом случае вызывающий обработает fatal_errno_exit).
- read_line(int fd, char *buf, size_t maxlen)
  — читает символы по одному из fd до \n или EOF. Гарантирует, что не перепишет буфер; если строка длиннее maxlen-1, пропускает остаток до \n. Возвращает длину (без \n) или -1 при ошибке.
- main()
  — главная логика: создание имен, shm, семафоров, fork + execl дочерних процессов, цикл чтения и обмена сообщениями, ожидание дочерних, освобождение ресурсов.

Важные системные вызовы и API

В программе используются следующие POSIX-интерфейсы:

- shm_open, ftruncate, mmap, munmap, shm_unlink
- sem_open, sem_post, sem_wait, sem_close, sem_unlink
- fork, execl, waitpid
- низкоуровневые I/O: read, write
- стандартные функции snprintf, malloc, free, strerror

Пояснения по синхронизации

- Семофоры используются для синхронизации передачи: родитель сигналит sem_p_c1, чтобы child1 прочитал вход; child1 сигналит sem_c1_c2 к child2; child2 сигналит sem_c2_p родителю с ответом.
- Такой строгий handshake предотвращает гонки и гарантирует, что каждый сегмент shm используется одним отправителем и одним получателем в конкретный момент.

Поведение при ошибках и границы

- Если любая из критических операций (shm/sem/create/fork/execl/read/write/mmap) терпит неудачу — программа печатает сообщение об ошибке и корректно завершает выполнение.
- Если пользователь вводит строку, превышающую SHM_SIZE - sizeof(uint32_t), родитель выводит сообщение об ошибке и пропускает строку.
- Конечный признак — пустая строка: отправляется msg_len = 0, что сигнализирует о завершении цепочки.

Инструкции по сборке и запуску

Код программы

parent.c

```
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <sys/wait.h>
#include <errno.h>
#include <sys/types.h>

#define SHM_SIZE 8192
#define LINE_BUF 4096

static void write_all(int fd, const void *buf, size_t len) {
    const char *p = (const char*)buf;
    while (len > 0) {
        ssize_t w = write(fd, p, len);
        if (w == -1) {
            if (errno == EINTR) continue;
            _exit(2);
        }
        p += w;
        len -= (size_t)w;
    }
}

static void fatal_errno_exit(const char *prefix) {
    char tmp[512];
    int n = snprintf(tmp, sizeof(tmp), "%s: %s\n", prefix, strerror(errno));
    if (n < 0) _exit(2);
    write_all(STDERR_FILENO, tmp, (size_t)n);
    _exit(EXIT_FAILURE);
```

```
}

static void fatal_msg_exit(const char *msg) {
    size_t len = strlen(msg);
    write_all(STDERR_FILENO, msg, len);
    write_all(STDERR_FILENO, "\n", 1);
    _exit(EXIT_FAILURE);
}

static int create_and_map_shm(const char *name, size_t size, void **out_addr, int
*out_fd) {
    int fd = shm_open(name, O_RDWR | O_CREAT | O_EXCL, 0600);
    if (fd == -1) return -1;
    if (ftruncate(fd, size) == -1) {
        close(fd);
        shm_unlink(name);
        return -1;
    }
    void *addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) {
        close(fd);
        shm_unlink(name);
        return -1;
    }
    *out_addr = addr;
    *out_fd = fd;
    return 0;
}

static ssize_t read_line(int fd, char *buf, size_t maxlen) {
    size_t pos = 0;
    char c;
    for (;;) {
        ssize_t r = read(fd, &c, 1);
        if (r == 0) {
            return (ssize_t)pos;
        }
        if (r == -1) {
            if (errno == EINTR) continue;
            return -1;
        }
        if (c == '\n') {
            return (ssize_t)pos;
        }
        if (pos + 1 < maxlen) {
            buf[pos++] = c;
        } else {
            while (c != '\n') {
                ssize_t r2 = read(fd, &c, 1);
                if (r2 <= 0) break;
                if (c == '\n') break;
            }
            return (ssize_t)pos;
        }
    }
}

int main(void) {
    pid_t pid = getpid();
    char shm_p_c1_name[128], shm_c1_c2_name[128], shm_c2_p_name[128];
    char sem_p_c1_name[128], sem_c1_c2_name[128], sem_c2_p_name[128];

    int n;
    n = snprintf(shm_p_c1_name, sizeof(shm_p_c1_name), "/lab13_shm_p_c1_%d", pid);
    if (n < 0) fatal_msg_exit("snprintf failed");
```

```
    n = snprintf(shm_c1_c2_name, sizeof(shm_c1_c2_name), "/lab13_shm_c1_c2_%d", pid);
    if (n < 0) fatal_msg_exit("snprintf failed");
    n = snprintf(shm_c2_p_name, sizeof(shm_c2_p_name), "/lab13_shm_c2_p_%d", pid);
    if (n < 0) fatal_msg_exit("snprintf failed");

    n = snprintf(sem_p_c1_name, sizeof(sem_p_c1_name), "/lab13_sem_p_c1_%d", pid);
    if (n < 0) fatal_msg_exit("snprintf failed");
    n = snprintf(sem_c1_c2_name, sizeof(sem_c1_c2_name), "/lab13_sem_c1_c2_%d", pid);
    if (n < 0) fatal_msg_exit("snprintf failed");
    n = snprintf(sem_c2_p_name, sizeof(sem_c2_p_name), "/lab13_sem_c2_p_%d", pid);
    if (n < 0) fatal_msg_exit("snprintf failed");

    void *shm_p_c1 = NULL, *shm_c1_c2 = NULL, *shm_c2_p = NULL;
    int fd1, fd2, fd3;

    if (create_and_map_shm(shm_p_c1_name, SHM_SIZE, &shm_p_c1, &fd1) == -1)
        fatal_errno_exit("shm_open p->c1");
    if (create_and_map_shm(shm_c1_c2_name, SHM_SIZE, &shm_c1_c2, &fd2) == -1)
        fatal_errno_exit("shm_open c1->c2");
    if (create_and_map_shm(shm_c2_p_name, SHM_SIZE, &shm_c2_p, &fd3) == -1)
        fatal_errno_exit("shm_open c2->p");

    sem_t *sem_p_c1 = sem_open(sem_p_c1_name, O_CREAT | O_EXCL, 0600, 0);
    sem_t *sem_c1_c2 = sem_open(sem_c1_c2_name, O_CREAT | O_EXCL, 0600, 0);
    sem_t *sem_c2_p = sem_open(sem_c2_p_name, O_CREAT | O_EXCL, 0600, 0);
    if (sem_p_c1 == SEM_FAILED || sem_c1_c2 == SEM_FAILED || sem_c2_p == SEM_FAILED)
        fatal_errno_exit("sem_open");

    pid_t c1 = fork();
    if (c1 == -1) fatal_errno_exit("fork c1");
    if (c1 == 0) {
        execl("./child1", "child1",
              shm_p_c1_name, shm_c1_c2_name,
              sem_p_c1_name, sem_c1_c2_name,
              (char*)NULL);
        char msg[256];
        int ln = snprintf(msg, sizeof(msg), "execl child1 failed: %s\n",
strerror(errno));
        if (ln > 0) write_all(STDERR_FILENO, msg, (size_t)ln);
        _exit(127);
    }

    pid_t c2 = fork();
    if (c2 == -1) fatal_errno_exit("fork c2");
    if (c2 == 0) {
        execl("./child2", "child2",
              shm_c1_c2_name, shm_c2_p_name,
              sem_c1_c2_name, sem_c2_p_name,
              (char*)NULL);
        char msg[256];
        int ln = snprintf(msg, sizeof(msg), "execl child2 failed: %s\n",
strerror(errno));
        if (ln > 0) write_all(STDERR_FILENO, msg, (size_t)ln);
        _exit(127);
    }

    char *buf = malloc(LINE_BUF);
    if (!buf) fatal_msg_exit("malloc failed");

    const char *prompt = "Введите строки. Пустая строка (просто Enter) завершает.\n";
    write_all(STDOUT_FILENO, prompt, strlen(prompt));

    while (1) {
        ssize_t len = read_line(STDIN_FILENO, buf, LINE_BUF);
        if (len == -1) {
```

```
            free(buf);
            fatal_errno_exit("read");
        }
        if (len == 0) {
            uint32_t msg_len = 0;
            memcpy(shm_p_c1, &msg_len, sizeof(msg_len));
            if (sem_post(sem_p_c1) == -1) fatal_errno_exit("sem_post p->c1");
            break;
        }

        if ((uint32_t)len + sizeof(uint32_t) > SHM_SIZE) {
            char emsg[128];
            int ln = snprintf(emsg, sizeof(emsg), "Строка слишком длинная (ьфч %d)\n",
SHM_SIZE - (int)sizeof(uint32_t));
            if (ln > 0) write_all(STDERR_FILENO, emsg, (size_t)ln);
            continue;
        }

        uint32_t msg_len = (uint32_t)len;
        memcpy(shm_p_c1, &msg_len, sizeof(msg_len));
        memcpy((char*)shm_p_c1 + sizeof(msg_len), buf, msg_len);
        if (sem_post(sem_p_c1) == -1) fatal_errno_exit("sem_post p->c1");

        if (sem_wait(sem_c2_p) == -1) fatal_errno_exit("sem_wait c2->p");
        uint32_t rlen;
        memcpy(&rlen, shm_c2_p, sizeof(rlen));
        if (rlen == 0) {
            break;
        }
        if (rlen + sizeof(uint32_t) > SHM_SIZE) {
            const char *msg2 = "Получено слишком много данных\n";
            write_all(STDERR_FILENO, msg2, strlen(msg2));
            continue;
        }
        char *res = malloc((size_t)rlen + 1);
        if (!res) fatal_msg_exit("malloc failed");
        memcpy(res, (char*)shm_c2_p + sizeof(rlen), rlen);
        res[rlen] = '\0';

        char outbuf[LINE_BUF + 64];
        int outn = snprintf(outbuf, sizeof(outbuf), "%s\n", res);
        if (outn > 0) write_all(STDOUT_FILENO, outbuf, (size_t)outn);

        free(res);
    }

    waitpid(c1, NULL, 0);
    waitpid(c2, NULL, 0);

    munmap(shm_p_c1, SHM_SIZE); close(fd1); shm_unlink(shm_p_c1_name);
    munmap(shm_c1_c2, SHM_SIZE); close(fd2); shm_unlink(shm_c1_c2_name);
    munmap(shm_c2_p, SHM_SIZE); close(fd3); shm_unlink(shm_c2_p_name);

    sem_close(sem_p_c1); sem_unlink(sem_p_c1_name);
    sem_close(sem_c1_c2); sem_unlink(sem_c1_c2_name);
    sem_close(sem_c2_p); sem_unlink(sem_c2_p_name);

    free(buf);
    return 0;
}
```

child1.c

```
#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <stdint.h>
```

```
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <errno.h>

#define SHM_SIZE 8192

static void err_exit(const char *msg) {
    char buf[256];
    int n = snprintf(buf, sizeof(buf), "%s: %d\n", msg, errno);
    if (n > 0) write(2, buf, (size_t)n);
    _exit(1);
}

static void write_str(const char *s) {
    write(2, s, strlen(s));
}

int main(int argc, char **argv) {

    if (argc != 5) {
        write_str("Usage: child1 <shm_p_c1> <shm_c1_c2> <sem_p_c1> <sem_c1_c2>\n");
        return 2;
    }

    const char *shm_in_name  = argv[1];
    const char *shm_out_name = argv[2];
    const char *sem_in_name  = argv[3];
    const char *sem_out_name = argv[4];

    int fd_in  = shm_open(shm_in_name,  O_RDWR, 0);
    int fd_out = shm_open(shm_out_name, O_RDWR, 0);
    if (fd_in == -1 || fd_out == -1)
        err_exit("shm_open child1");

    void *shm_in = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd_in, 0);
    void *shm_out = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd_out,
0);
    if (shm_in == MAP_FAILED || shm_out == MAP_FAILED)
        err_exit("mmap child1");

    sem_t *sem_in  = sem_open(sem_in_name, 0);
    sem_t *sem_out = sem_open(sem_out_name, 0);
    if (sem_in == SEM_FAILED || sem_out == SEM_FAILED)
        err_exit("sem_open child1");

    while (1) {
        if (sem_wait(sem_in) == -1)
            err_exit("sem_wait child1");

        uint32_t len;
        memcpy(&len, shm_in, sizeof(len));

        if (len == 0) {
            uint32_t zero = 0;
            memcpy(shm_out, &zero, sizeof(zero));
            sem_post(sem_out);
            break;
        }

        if (len + sizeof(len) > SHM_SIZE) {
```

```
            write_str("child1: too large\n");
            continue;
        }

        char *tmp = malloc(len + 1);
        if (!tmp)
            err_exit("malloc child1");

        memcpy(tmp, (char*)shm_in + sizeof(len), len);
        tmp[len] = '\0';

        for (uint32_t i = 0; i < len; ++i)
            tmp[i] = (char)tolower((unsigned char)tmp[i]);

        memcpy(shm_out, &len, sizeof(len));
        memcpy((char*)shm_out + sizeof(len), tmp, len);

        sem_post(sem_out);
        free(tmp);
    }

    munmap(shm_in, SHM_SIZE);
    munmap(shm_out, SHM_SIZE);
    close(fd_in);
    close(fd_out);
    sem_close(sem_in);
    sem_close(sem_out);

    return 0;
}
```

child2.c

```
#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <errno.h>

#define SHM_SIZE 8192

static void err_exit(const char *msg) {
    char buf[256];
    int n = snprintf(buf, sizeof(buf), "%s: %d\n", msg, errno);
    if (n > 0) write(2, buf, (size_t)n);
    _exit(1);
}

static void write_str(const char *s) {
    write(2, s, strlen(s));
}

int main(int argc, char **argv) {

    if (argc != 5) {
        write_str("Usage: child2 <shm_c1_c2> <shm_c2_p> <sem_c1_c2> <sem_c2_p>\n");
        return 2;
    }

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

```
        const char *shm_out_name = argv[2];
        const char *sem_in_name  = argv[3];
        const char *sem_out_name = argv[4];

        int fd_in  = shm_open(shm_in_name,  O_RDWR, 0);
        int fd_out = shm_open(shm_out_name, O_RDWR, 0);
        if (fd_in == -1 || fd_out == -1)
            err_exit("shm_open child2");

        void *shm_in = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd_in, 0);
        void *shm_out = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd_out,
0);
        if (shm_in == MAP_FAILED || shm_out == MAP_FAILED)
            err_exit("mmap child2");

        sem_t *sem_in  = sem_open(sem_in_name, 0);
        sem_t *sem_out = sem_open(sem_out_name, 0);
        if (sem_in == SEM_FAILED || sem_out == SEM_FAILED)
            err_exit("sem_open child2");

        while (1) {

            if (sem_wait(sem_in) == -1)
                err_exit("sem_wait child2");

            uint32_t len;
            memcpy(&len, shm_in, sizeof(len));

            if (len == 0) {
                uint32_t zero = 0;
                memcpy(shm_out, &zero, sizeof(zero));
                sem_post(sem_out);
                break;
            }

            if (len + sizeof(len) > SHM_SIZE) {
                write_str("child2: too large\n");
                continue;
            }

            char *tmp = malloc(len + 1);
            if (!tmp)
                err_exit("malloc child2");

            memcpy(tmp, (char*)shm_in + sizeof(len), len);
            tmp[len] = '\0';

            for (uint32_t i = 0; i < len; ++i) {
                if (isspace((unsigned char)tmp[i]))
                    tmp[i] = '_';
            }

            memcpy(shm_out, &len, sizeof(len));
            memcpy((char*)shm_out + sizeof(len), tmp, len);

            sem_post(sem_out);
            free(tmp);
        }

        munmap(shm_in, SHM_SIZE);
        munmap(shm_out, SHM_SIZE);
        close(fd_in);
        close(fd_out);
        sem_close(sem_in);
        sem_close(sem_out);
```

```
    return 0;
}
```

Протокол работы программы

Протокол работы программы

Результат работы:

> zeotq@DESKTOP-OJSJFU7:~/dev/study/mai-os-3/build$ ./parent

> Введите строки. Пустая строка (просто Enter) завершает.

< BigBoy BEEEEEEEG BOY Hello

> bigboy_beeeeeeeg_boy_hello

Вывод

Задача выполнена.