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: Захарченко М.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 13.11.25

Москва, 2025

Постановка задачи

Вариант 8.

Родительский процесс получает на вход имя входного файла. Стандартный поток ввода дочернего процесса переопределяется файлом. Дочерний процесс обрабатывает файл, состоящий из команд вида “число1 число2 число3/n” и подает это родительском процессу в виде “число1/чилсо2 число1/число3”. Взаимодействие между процессами реализовать с помощью shared memory и memory mapping.

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 родительском процессе осуществлялся только вывод получаемых данных от процесса-ребенка. Стандартный поток ввода процесса-ребенка заменялся файлом с командами. Взаимодействие между процессами было реализовано с помощью shared memory, mmap и семафоров, для предотвращения гонки данных. Родительский процесс ждал команды от ребенка, что он выполнил часть своей работы, первый с свою очередь выполнял вывод, и также отправлял ребенку команду, что он выполнил свою часть работы, чтобы ребенок продолжал работу. Взаимодействие было реализовано с помощью sem_wait и sem_post**.**

Код программы

parent.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>
#include <stdio.h>

#define SHM_SIZE 4096

char SHM_NAME[1024] = "shm-name";
char SEM_NAME_PARENT[1024] = "sem-parent";
char SEM_NAME_CHILD[1024] = "sem-child";

int main(int argc, char* argv[]) {
   if(argc < 2){
       const char error_msg[] = "error: incorrect input\nUsage:
<programmname> <file_name>\n";
       write(STDERR_FILENO, error_msg, sizeof(error_msg));
       _exit(EXIT_FAILURE);
   }

   char pid[64] = "\0";
   snprintf(pid, sizeof(pid), "%d", getpid());

   snprintf(SHM_NAME, sizeof(SHM_NAME), "shm-%s", pid);
   snprintf(SEM_NAME_PARENT, sizeof(SEM_NAME_PARENT), "sem-parent-%s", pid);
```

```
    snprintf(SEM_NAME_CHILD, sizeof(SEM_NAME_CHILD), "sem-child-%s", pid);

    int shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_TRUNC, 0600);
    if (shm == -1) {
        const char error_msg[] = "error: failed to open SHM\n";
        write(STDERR_FILENO, error_msg, sizeof(error_msg));
        _exit(EXIT_FAILURE);
    }

    if (ftruncate(shm, SHM_SIZE) == -1) {
        const char error_msg[] = "error: failed to resize SHM\n";
        write(STDERR_FILENO, error_msg, sizeof(error_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 error_msg[] = "error: failed to map SHM\n";
        write(STDERR_FILENO, error_msg, sizeof(error_msg));
        _exit(EXIT_FAILURE);
    }

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

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

    pid_t child = fork();

    if (child == 0) {
        char *args[] = {"child", SHM_NAME, SEM_NAME_PARENT, SEM_NAME_CHILD,
argv[1], NULL};
        execv("./child", args);

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

    bool running = true;
    while (running) {
        sem_wait(sem_child); // ждем пока ребенок не даст знак, что он
отработал

        uint32_t *length = (uint32_t *)shm_buf;
        char *text = shm_buf + sizeof(uint32_t);
        if (*length == UINT32_MAX) {
```

```
            running = false;
        }
        else if (*length > 0) {
            write(STDOUT_FILENO, text, *length);
        }

        sem_post(sem_parent); // подаем знак ребенку, что родитель выполнил
свою часть работы
    }

    waitpid(child, NULL, 0);

    sem_close(sem_parent);
    sem_close(sem_child);
    sem_unlink(SEM_NAME_PARENT);
    sem_unlink(SEM_NAME_CHILD);
    munmap(shm_buf, SHM_SIZE);
    shm_unlink(SHM_NAME);
    close(shm);

    return 0;
}
```

child.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>
#include <stdio.h>

#define SHM_SIZE 4096

void intToStr(char* number, int num) {
    char temp[12];
    bool negative = false;
    int len = 0;

    if(num < 0) {
        num = -num;
        negative = true;
    }

    do {
        temp[len++] = (num % 10) + '0';
        num /= 10;
    } while(num > 0);

    if(negative) {
        temp[len++] = '-';
    }

    for(int i = 0; i < len; ++i) {
        number[i] = temp[len - i - 1];
    }
    number[len] = '\0';
}
```

```
int strToInt(char* number) {
    int result = 0;
    int start = 0;
    if(number[0] == '-') {
        start = 1;
    }

    for(int i = start; i < strlen(number); ++i) {
        result = result * 10 + (number[i] - '0');
    }

    if(start) {
        return -result;
    }
    return result;
}

void compare(char* res, int num1, int num2) {
    char stringNumberOne[12];
    char stringNumberTwo[12];
    intToStr(stringNumberOne, num1);
    intToStr(stringNumberTwo, num2);

    strcpy(res, stringNumberOne);
    strcat(res, " ");
    strcat(res, stringNumberTwo);
    strcat(res, "\n");
}

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

    const char *SHM_NAME = argv[1];
    const char *SEM_NAME_SERVER = argv[2];
    const char *SEM_NAME_CHILD = argv[3];
    const char *filename = argv[4];

    int shm = shm_open(SHM_NAME, O_RDWR, 0600);
    if (shm == -1) {
        const char error_msg[] = "error: failed to open SHM\n";
        write(STDERR_FILENO, error_msg, sizeof(error_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 error_msg[] = "error: failed to map\n";
        write(STDERR_FILENO, error_msg, sizeof(error_msg));
        _exit(EXIT_FAILURE);
    }

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

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

    int file = open(filename, O_RDONLY);
    if (file == -1) {
        const char error_msg[] = "error: failed to open file\n";
        write(STDERR_FILENO, error_msg, sizeof(error_msg));
        _exit(EXIT_FAILURE);
    }

    if (dup2(file, STDIN_FILENO) == -1) {
        const char error_msg[] = "error: failed to redirect stdin\n";
        write(STDERR_FILENO, error_msg, sizeof(error_msg));
        _exit(EXIT_FAILURE);
    }
    close(file);

    char buf[4096];
    ssize_t bytes;
    bool running = true;

    while (running && (bytes = read(STDIN_FILENO, buf, sizeof(buf)))) {
        if (bytes < 0) {
            const char error_msg[] = "error: failed to read from file\n";
            write(STDERR_FILENO, error_msg, sizeof(error_msg));
            _exit(EXIT_FAILURE);
        }

        int numbers[3];
        int count = 0, strIndex = 0;
        char strNumber[12];
        char outputString[25];
        size_t msg_len;
        for (uint32_t i = 0; i < bytes && running; ++i) {
            if(buf[i] == '-' || (buf[i] >= '0' && buf[i] <= '9')) {
                strNumber[strIndex++] = buf[i];
            } else if(buf[i] == ' ' || buf[i] == '\n') {
                strNumber[strIndex] = '\0';
                numbers[count++] = strToInt(strNumber);

                strNumber[0] = '\0';
                strIndex = 0;

                if(count == 3 && buf[i] == '\n') {
                    if(numbers[1] == 0 || numbers[2] == 0) {
                        const char error_msg[] = "error: in line division by
zero\n";
                        msg_len = strlen(error_msg);

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

                        *length = msg_len;
                        memcpy(text, error_msg, *length);
                        sem_post(sem_child);
                        count = 0;
                        continue;
                    }
```

```
                    int result1 = numbers[0] / numbers[1];
                    int result2 = numbers[0] / numbers[2];

                    count = 0;
                    outputString[0] = '\0';
                    compare(outputString, result1, result2);

                    sem_wait(sem_parent); // ждем знака от родителя, что он
выполнил свою работу
                    uint32_t *length = (uint32_t *)shm_buf;
                    char *text = shm_buf + sizeof(uint32_t);

                    *length = strlen(outputString);
                    memcpy(text, outputString, *length);
                    sem_post(sem_child); // говорим родителю, что ребенок
выполнил свою работу
                } else if(count < 3 && buf[i] == '\n') {
                    const char error_msg[] = "error: in line must be 3
numbers\n";
                    msg_len = strlen(error_msg);

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

                    *length = msg_len;
                    memcpy(text, error_msg, *length);
                    sem_post(sem_child);
                }
            } else {
                const char error_msg[] = "error: in line numbers must be
natural\n";
                msg_len = strlen(error_msg);

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

                *length = msg_len;
                memcpy(text, error_msg, *length);
                sem_post(sem_child);
            }
        }
    }

    sem_wait(sem_parent);
    uint32_t *length = (uint32_t *)shm_buf;
    *length = UINT32_MAX;
    sem_post(sem_child);

    sem_close(sem_parent);
    sem_close(sem_child);
    munmap(shm_buf, SHM_SIZE);
    close(shm);

    return 0;
}
```

Протокол работы программы

```
matthew@matthews-pc:~/LABS_OPERATING_SYSTEM/LAB_3$ cat input.txt
12 2 0
15 5 5
9.2 10 92
100 100 4
27 3 9
12 3
56 78 32
matthew@matthews-pc:~/LABS_OPERATING_SYSTEM/LAB_3$ ./parent input.txt
error: in line division by zero
3 3
error: in line numbers must be natural
9 1
1 25
9 3
error: in line must be 3 numbers
```

Вывод

В ходе лабораторной работы были созданы два процесса, взаимодействие которых происходило с помощью shared memory, memory mapping, также были использованы семафоры, которые предотвращали гонку данных между процессами. Для вывода данных родительский процесс ждал “знака” от процесса-ребенка (sem_wait(sem_child)), после его получения (с помощью sem_post(sem_child)) родители выводил результат обработки, в это время процесс-ребенок ждал “знака” от родителя (sem_wait(sem_parent)) и получал его после вывода родителем результат (с помощью sem_post(sem_parent)). Таким образом, были успешно освоены навыки по работе с shared memory, memory mapping.