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: М80-211Бв-24

Студент: Скворцов Ю.И.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата:5.12.25

Москва, 2025

Постановка задачи

Вариант 6.

Родительский процесс создает дочерний процесс. Первой строчкой пользователь в консоль родительского процесса пишет имя файла, которое будет передано при создании дочернего процесса. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс передает команды пользователя через pipe1, который связан с стандартным входным потоком дочернего процесса. Дочерний процесс при необходимости передает данные в родительский процесс через pipe2. Результаты своей работы дочерний процесс пишет в созданный им файл. Допускается просто открыть файл и писать туда, не перенаправляя стандартный поток вывода.

Пользователь вводит команды вида: «число число число». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс производит деление первого числа, на последующие, а результат выводит в файл. Если происходит деление на 0, то тогда дочерний и родительский процесс завершают свою работу. Проверка деления на 0 должна осуществляться на стороне дочернего процесса. Числа имеют тип float. Количество чисел может быть произвольным

Общий метод и алгоритм решения

ftruncate() - Изменяет размер файла

shm_open() - Создаёт/открывает именованную shared memory

shm_unlink() - Удаляет именованную shared memory

mmap() - Отображает файл/shared memory в память процесса

munmap() - Убирает отображение памяти

sem_open() - Создаёт/открывает именованный семафор

sem_wait() - Уменьшает семафор (ожидает если 0)

sem_post() = Увеличивает семафор (будит ожидающих)

sem_close() / sem_unlink() - Закрывает/удаляет именованный семафор

Работа программы:

Родитель сначала готовит рабочее пространство: создаёт shared memory и два семафора, давая им уникальные имена на основе своего PID. Потом он создаёт дочерний процесс через fork(), и ребёнок запускает свою программу child, получая имена ресурсов как аргументы. Цикл начинается с пользовательского ввода в родителе. Родитель копирует команду (например, "10 2 5") в shared memory и будит ребёнка семафором. Сам засыпает, ждя ответа. Ребёнок, проснувшись, читает данные из shared memory, парсит числа, проверяет их валидность. Если всё хорошо — записывает в файл выражение типа "10.00 / 2.00 / 5.00", выполняет деление (первое число делит на все остальные), проверяя деление на ноль. Результат пишет в файл. Потом возвращает в shared memory "OK" и будит родителя. Родитель проверяет ответ: если "DIVISION_BY_ZERO" — выводит ошибку и заканчивает работу; если "OK" — ждёт следующую команду. При вводе "exit" родитель отправляет команду завершения, ждёт окончания работы ребёнка, потом аккуратно освобождает все ресурсы (shared memory и семафоры) и завершается.

Код программы

parent.c

```c
#include <stdint.h>

#include <stdbool.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <semaphore.h>


#define SHM_SIZE 4096


static char CHILD_PROGRAM_NAME[] = "child";


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

    if (argc == 1) {

        char msg[1024];

        uint32_t len = snprintf(msg, sizeof(msg), "usage: %s filename\n",
argv[0]);

        write(STDERR_FILENO, msg, len);

        exit(EXIT_SUCCESS);

    }


    char progpath[1024];

    ssize_t len = readlink("/proc/self/exe", progpath, sizeof(progpath) - 1);
```

```
    if (len == -1) {
        const char msg[] = "error: failed to read full program path\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }
    progpath[len] = '\0';
    char *last_slash = strrchr(progpath, '/');
    if (last_slash) *last_slash = '\0';

    pid_t pid = getpid();
    char shm_name[64], sem_parent_name[64], sem_child_name[64];
    snprintf(shm_name, sizeof(shm_name), "/shm_%d", pid);
    snprintf(sem_parent_name, sizeof(sem_parent_name), "/sem_parent_%d", pid);
    snprintf(sem_child_name, sizeof(sem_child_name), "/sem_child_%d", pid);

    int shm_fd = shm_open(shm_name, O_CREAT | O_RDWR, 0600);
    if (shm_fd == -1) {
        const char msg[] = "error: shm_open failed\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }
    if (ftruncate(shm_fd, SHM_SIZE) == -1) {
        const char msg[] = "error: ftruncate failed\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }
    char *shm_ptr = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);
    if (shm_ptr == MAP_FAILED) {
```

```
        const char msg[] = "error: mmap failed\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }

    sem_t *sem_parent = sem_open(sem_parent_name, O_CREAT, 0600, 0);
    sem_t *sem_child = sem_open(sem_child_name, O_CREAT, 0600, 0);
    if (sem_parent == SEM_FAILED || sem_child == SEM_FAILED) {
        const char msg[] = "error: sem_open failed\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }

    pid_t child = fork();
    switch (child) {
    case -1: {
        const char msg[] = "error: failed to spawn new process\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    } break;

    case 0: {
        char child_path[2048];
        snprintf(child_path, sizeof(child_path), "%s/%s", progpath,
CHILD_PROGRAM_NAME);

        char *args[] = {
            CHILD_PROGRAM_NAME,
            argv[1],
```

```
            shm_name,
            sem_parent_name,
            sem_child_name,
            NULL
        };

        execv(child_path, args);

        const char msg[] = "error: failed to exec into new executable image\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    } break;

    default: {
        char msg[128];
        int32_t length = snprintf(msg, sizeof(msg),
            "%d: Parent process, child PID: %d\n", pid, child);
        write(STDOUT_FILENO, msg, length);

        const char prompt1[] = "Enter command type: num num num ...\n";
        write(STDOUT_FILENO, prompt1, sizeof(prompt1) - 1);
        const char prompt2[] = "Print 'exit' to out\n";
        write(STDOUT_FILENO, prompt2, sizeof(prompt2) - 1);

        char buf[4096];
        ssize_t bytes;

        while ((bytes = read(STDIN_FILENO, buf, sizeof(buf))) > 0) {
            if (bytes >= 4 && strncmp(buf, "exit", 4) == 0) {
```

```
            break;
        }

        size_t copy_size = (bytes < SHM_SIZE - 1) ? bytes : SHM_SIZE - 1;
        memcpy(shm_ptr, buf, copy_size);
        shm_ptr[copy_size] = '\0';

        sem_post(sem_child);
        sem_wait(sem_parent);

        if (strstr(shm_ptr, "DIVISION_BY_ZERO") != NULL) {
            const char error_msg[] = "Zero division found!\n";
            write(STDOUT_FILENO, error_msg, sizeof(error_msg) - 1);
            break;
        } else if (strstr(shm_ptr, "OK") != NULL) {
        } else if (strlen(shm_ptr) == 0) {
            const char child_exit_msg[] = "Child process ended.\n";
            write(STDOUT_FILENO, child_exit_msg, sizeof(child_exit_msg) - 1);
            break;
        }
    }

    snprintf(shm_ptr, SHM_SIZE, "exit");
    sem_post(sem_child);

    wait(NULL);
    const char parent_exit_msg[] = "Child process ended.\n";
    write(STDOUT_FILENO, parent_exit_msg, sizeof(parent_exit_msg) - 1);
```

```
        munmap(shm_ptr, SHM_SIZE);
        close(shm_fd);
        shm_unlink(shm_name);
        sem_close(sem_parent);
        sem_close(sem_child);
        sem_unlink(sem_parent_name);
        sem_unlink(sem_child_name);
    } break;
    }

    return 0;
}
```

child.c

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

#define MAX_NUMBERS 100
#define SHM_SIZE 4096
```

```c
int main(int argc, char **argv) {
    if (argc < 5) {
        char msg[256];
        uint32_t len = snprintf(msg, sizeof(msg),
            "usage: %s filename shm_name sem_parent_name sem_child_name\n",
argv[0]);
        write(STDERR_FILENO, msg, len);
        exit(EXIT_FAILURE);
    }

    int file = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0600);
    if (file == -1) {
        const char msg[] = "error: failed to open requested file\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }

    int shm_fd = shm_open(argv[2], O_RDWR, 0);
    if (shm_fd == -1) {
        const char msg[] = "error: shm_open failed\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }
    char *shm_ptr = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);
    if (shm_ptr == MAP_FAILED) {
        const char msg[] = "error: mmap failed\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }
```

```
sem_t *sem_parent = sem_open(argv[3], 0);
sem_t *sem_child = sem_open(argv[4], 0);
if (sem_parent == SEM_FAILED || sem_child == SEM_FAILED) {
    const char msg[] = "error: sem_open failed\n";
    write(STDERR_FILENO, msg, sizeof(msg) - 1);
    exit(EXIT_FAILURE);
}

float numbers[MAX_NUMBERS];
while (1) {
    sem_wait(sem_child);

    if (strcmp(shm_ptr, "exit") == 0) {
        break;
    }

    char local_buf[SHM_SIZE];
    strncpy(local_buf, shm_ptr, sizeof(local_buf) - 1);
    local_buf[sizeof(local_buf) - 1] = '\0';

    int count = 0;
    char *token = strtok(local_buf, " \t\n");
    while (token != NULL && count < MAX_NUMBERS) {
        int valid = 1;
        int dot_count = 0;
        for (int i = 0; token[i] != '\0'; i++) {
            if (!isdigit(token[i]) && token[i] != '.' &&
                !(i == 0 && token[i] == '-')) {
```

```
                valid = 0;
                break;
            }
            if (token[i] == '.') dot_count++;
        }
        if (valid && dot_count <= 1) {
            numbers[count++] = atof(token);
        }
        token = strtok(NULL, " \t\n");
    }

    if (count < 2) {
        const char error_msg[] = "ERROR: Need at least 2 numbers\n";
        write(STDOUT_FILENO, error_msg, sizeof(error_msg) - 1);
        const char file_error[] = "Error: 2 numbers needed\n";
        write(file, file_error, sizeof(file_error) - 1);
        snprintf(shm_ptr, SHM_SIZE, "ERROR");
        sem_post(sem_parent);
        continue;
    }

    float first = numbers[0];
    char calculation[512];
    int calc_len = snprintf(calculation, sizeof(calculation),
                            "Calculation: %.2f / ", first);
    write(file, calculation, calc_len);

    for (int i = 1; i < count; i++) {
        calc_len = snprintf(calculation, sizeof(calculation),
```

```
                        "%.2f", numbers[i]);
        write(file, calculation, calc_len);

        if (i < count - 1) {
            const char divider[] = " / ";
            write(file, divider, sizeof(divider) - 1);
        }
    }
    const char newline[] = "\n";
    write(file, newline, sizeof(newline) - 1);

    float result = first;
    for (int i = 1; i < count; i++) {
        if (numbers[i] == 0.0f) {
            const char error_msg[] = "DIVISION_BY_ZERO\n";
            write(STDOUT_FILENO, error_msg, sizeof(error_msg) - 1);
            const char file_error[] = "Error: zero division!\n";
            write(file, file_error, sizeof(file_error) - 1);
            snprintf(shm_ptr, SHM_SIZE, "DIVISION_BY_ZERO");
            sem_post(sem_parent);
            close(file);
            exit(EXIT_FAILURE);
        }
        result /= numbers[i];
    }

    char result_str[128];
    int result_len = snprintf(result_str, sizeof(result_str),
                        "Result: %.6f\n\n", result);
```

```
        write(file, result_str, result_len);


        snprintf(shm_ptr, SHM_SIZE, "OK");
        sem_post(sem_parent);
    }


    close(file);
    munmap(shm_ptr, SHM_SIZE);
    close(shm_fd);
    sem_close(sem_parent);
    sem_close(sem_child);


    return 0;
}
```

Протокол работы программы

```
9283: Parent process, child PID: 9284
Enter command type: num num num ...
Print 'exit' to out
1 2 3
3 2 1
1 0
DIVISION_BY_ZERO
Zero division found!
Child process ended.
```

```
Calculation: 1.00 / 2.00 / 3.00
Result: 0.166667

Calculation: 3.00 / 2.00 / 1.00
Result: 1.500000

Calculation: 1.00 / 0.00
Error: zero division!
```

```
22960: Parent process, child PID: 22961
Enter command type: num num num ...
Print 'exit' to out
10 3 3
10000 10 10 10 10
29 7 1 3
exit
Child process ended.
```

```
Calculation: 10.00 / 3.00 / 3.00
Result: 1.111111

Calculation: 10000.00 / 10.00 / 10.00 / 10.00 / 10.00
Result: 1.000000

Calculation: 29.00 / 7.00 / 1.00 / 3.00
Result: 1.380952
```

Вывод

Программа демонстрирует организацию межпроцессного взаимодействия через shared memory с синхронизацией именованными семафорами. Родительский процесс выступает в роли интерфейса для пользователя: принимает команды с числами и передаёт их дочернему процессу. Дочерний процесс выполняет вычисления (последовательное деление чисел) и сохраняет

результаты в файл, при этом корректно обрабатывая ошибки (деление на ноль, некорректный ввод). Программа успешно заменяет классические pipes на более гибкий механизм shared memory, сохраняя при этом надёжную синхронизацию и обработку ошибок.