МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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»

Студент: Кириллова Е.К.
Группа: М8О-203Б-23
Вариант: 12
Преподаватель: Миронов Е. С.

Дата: _____

Оценка: _____

Подпись: _____

Москва, 2024

Содержание

Репозиторий

https://github.com/ElenaKirillova05/osLabs

Постановка задачи

Цель работы

Приобретение практических навыков в:

Управление процессами в ОС

Обеспечение обмена данных между процессами посредством каналов

Задание

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (pipe). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Родительский процесс создает дочерний процесс. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись. Перенаправление стандартных потоков ввода-вывода показано на картинке выше. Родительский и дочерний процесс должны быть представлены разными программами.Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1. Процесс child проверяет строки на валидность правилу. Если строка соответствует правилу, то она выводится в стандартный поток вывода дочернего процесса, иначе в pipe2 выводится информация об ошибке. Родительский процесс полученные от child ошибки выводит в стандартный поток вывода.

12 вариант) Child1 переводит строки в верхний регистр. Child2 убирает все задвоенные пробелы

Общие сведения о программе

Разработанная система реализует межпроцессное взаимодействие с использованием общей памяти и сигналов для координации процессов программа состоит из трех компонентов: родительского процесса и двух дочерних процессов первый дочерний процесс преобразует все буквы строки в заглавные второй дочерний процесс удаляет лишние пробелы между словами данные передаются через общую память синхронизация осуществляется с использованием сигналов SIGUSR1 и SIGUSR2 родительский процесс управляет вводом и выводом обменивается данными с дочерними процессами и контролирует завершение программы

Общий метод и алгоритм решения

Общий метод и алгоритм решения родительский процесс создает объект общей памяти и отображает его в адресное пространство затем порождаются два дочерних процесса с помощью fork первый дочерний процесс ожидает сигнала SIGUSR1 затем преобразует строку в верхний регистр и отправляет сигнал родителю второй дочерний процесс ожидает сигнала SIGUSR2 затем удаляет лишние пробелы и уведомляет родителя о завершении родительский процесс передает введенную строку в общую память активирует первый дочерний процесс ожидает его завершения затем активирует второй дочерний процесс и выводит результат завершение программы осуществляется с помощью передачи сигнала SIGINT обоим дочерним процессам

Исходный код

child.c:

```

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <unistd.h>

#include <signal.h>


#define SHARED_MEMORY_FILE "/shared_memory"
```

```
#define MEMORY_SIZE 1024

volatile sig_atomic_t ready = 0;
volatile sig_atomic_t exit_flag = 0;

void signal_handler(int sig) {
    ready = 1;
}

void exit_handler(int sig) {
    exit_flag = 1;
}

int main() {
    signal(SIGUSR1, signal_handler);
    signal(SIGINT, exit_handler);

    int shared_fd = shm_open(SHARED_MEMORY_FILE, O_RDWR, 0666);
    if (shared_fd == -1) {
        perror("shm_opeфn failed");
        exit(EXIT_FAILURE);
    }

    char *shared_memory = mmap(NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shared_fd, 0);
    if (shared_memory == MAP_FAILED) {
        perror("mmap failed");
        exit(EXIT_FAILURE);
    }
```

```
    while (1) {
        while (!ready && !exit_flag) pause();

        if (exit_flag) break;

        for (int i = 0; shared_memory[i]; i++) {
            shared_memory[i] = toupper(shared_memory[i]);
        }

        kill(getppid(), SIGUSR1);

        ready = 0;
    }

    munmap(shared_memory, MEMORY_SIZE);
    close(shared_fd);
    return 0;
}
```

child2.c:

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

```
#include <signal.h>

#define SHARED_MEMORY_FILE "/shared_memory"
#define MEMORY_SIZE 1024

volatile sig_atomic_t ready = 0;
volatile sig_atomic_t exit_flag = 0;

void signal_handler(int sig) {
    ready = 1;
}

void exit_handler(int sig) {
    exit_flag = 1;
}

int main() {
    // Устанавливаем обработчик сигнала SIGUSR2
    signal(SIGUSR2, signal_handler);
    // Устанавливаем обработчик сигнала SIGINT для завершения программы
    signal(SIGINT, exit_handler);

    // Открываем файл общей памяти
    int shared_fd = shm_open(SHARED_MEMORY_FILE, O_RDWR, 0666);
    if (shared_fd == -1) {
        perror("shm_open failed");
        exit(EXIT_FAILURE);
    }
```

```
    // Отображаем файл общей памяти в адресное пространство процесса
    char *shared_memory = mmap(NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shared_fd, 0);
    if (shared_memory == MAP_FAILED) {
        perror("mmap failed");
        exit(EXIT_FAILURE);
    }

    while (1) {
        // Ждем сигнала готовности
        while (!ready && !exit_flag) pause();

        if (exit_flag) break;

        // Указатели для копирования строки
        char *src = shared_memory;
        char *dest = shared_memory;
        int last_was_space = 0;

        while (*src) {
            if (*src == ' ') {
                if (!last_was_space) {
                    *dest++ = ' ';
                    last_was_space = 1;
                }
            } else {
                *dest++ = *src;
                last_was_space = 0;
```

```
            }
            src++;
        }
        *dest = '\0';

        // Отправляем сигнал родительскому процессу
        kill(getppid(), SIGUSR2);

        // Сбрасываем флаг готовности
        ready = 0;
    }

    // Освобождаем память
    munmap(shared_memory, MEMORY_SIZE);
    close(shared_fd);
    return 0;
}
```

parent.c:
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
#include <signal.h>

#define SHARED_MEMORY_FILE "/shared_memory"
#define MEMORY_SIZE 1024

volatile sig_atomic_t child1_finished = 0;
volatile sig_atomic_t child2_finished = 0;

void signal_handler_child1(int sig) {
    child1_finished = 1;
}

void signal_handler_child2(int sig) {
    child2_finished = 1;
}

int main() {
    signal(SIGUSR1, signal_handler_child1);
    signal(SIGUSR2, signal_handler_child2);

    int shared_fd = shm_open(SHARED_MEMORY_FILE, O_CREAT | O_RDWR, 0666);
    if (shared_fd == -1) {
        perror("shm_open failed");
        exit(EXIT_FAILURE);
    }
    ftruncate(shared_fd, MEMORY_SIZE);

    char *shared_memory = mmap(NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shared_fd, 0);
```

```
    if (shared_memory == MAP_FAILED) {
        perror("mmap failed");
        exit(EXIT_FAILURE);
    }

    pid_t child1_pid = fork();
    if (child1_pid == -1) {
        perror("fork for child1 failed");
        exit(EXIT_FAILURE);
    }

    if (child1_pid == 0) {
        execl("./child1", "./child1", NULL);
        perror("execl for child1 failed");
        exit(EXIT_FAILURE);
    }

    pid_t child2_pid = fork();
    if (child2_pid == -1) {
        perror("fork for child2 failed");
        exit(EXIT_FAILURE);
    }

    if (child2_pid == 0) {
        execl("./child2", "./child2", NULL);
        perror("execl for child2 failed");
        exit(EXIT_FAILURE);
    }
```

```
    while (1) {
        printf("Enter the line (or 0 to end the program): ");
        if (fgets(shared_memory, MEMORY_SIZE, stdin) == NULL) break;

        if (shared_memory[0] == '0' && (shared_memory[1] == '\n' || shared_memory[1] == '\0'))
break;

        child1_finished = 0;
        child2_finished = 0;

        kill(child1_pid, SIGUSR1);
        while (!child1_finished) pause();

        kill(child2_pid, SIGUSR2);
        while (!child2_finished) pause();

        printf("Result: %s\n", shared_memory);
    }

    kill(child1_pid, SIGINT);
    kill(child2_pid, SIGINT);

    waitpid(child1_pid, NULL, 0);
    waitpid(child2_pid, NULL, 0);

    munmap(shared_memory, MEMORY_SIZE);
    shm_unlink(SHARED_MEMORY_FILE);
```

```
    return 0;
}
```

ТЕСТЫ

tests.cpp:
```
#include <gtest/gtest.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <cstring>

#define SHARED_MEMORY_FILE "/shared_memory"
#define MEMORY_SIZE 1024

class SharedMemoryTest : public ::testing::Test {
protected:
    char *shared_memory;
    int shared_fd;

    void SetUp() override {
        shared_fd = shm_open(SHARED_MEMORY_FILE, O_CREAT | O_RDWR, 0666);
        if (shared_fd == -1) {
            perror("shm_open failed");
            exit(EXIT_FAILURE);
```

```
        }
        ftruncate(shared_fd, MEMORY_SIZE);

        shared_memory = (char *)mmap(NULL, MEMORY_SIZE, PROT_READ |
PROT_WRITE, MAP_SHARED, shared_fd, 0);
        if (shared_memory == MAP_FAILED) {
            perror("mmap failed");
            exit(EXIT_FAILURE);
        }
    }

    void TearDown() override {
        munmap(shared_memory, MEMORY_SIZE);
        close(shared_fd);
        shm_unlink(SHARED_MEMORY_FILE);
    }
};

TEST_F(SharedMemoryTest, MemoryOperationsTest) {
    const char* test_input = "hello world";
    strncpy(shared_memory, test_input, MEMORY_SIZE);

    EXPECT_STREQ(shared_memory, test_input);
}

TEST_F(SharedMemoryTest, SignalHandlingTest) {
    const char* input_data = " hello  world ";
    strncpy(shared_memory, input_data, MEMORY_SIZE);
```

```
    for (int i = 0; shared_memory[i]; i++) {
        shared_memory[i] = toupper(shared_memory[i]);
    }

    EXPECT_STREQ(shared_memory, " HELLO  WORLD ");

    char *src = shared_memory;
    char *dest = shared_memory;
    int last_was_space = 0;

    while (*src) {
        if (*src == ' ') {
            if (!last_was_space) {
                *dest++ = ' ';
                last_was_space = 1;
            }
        } else {
            *dest++ = *src;
            last_was_space = 0;
        }
        src++;
    }
    *dest = '\0';

    EXPECT_STREQ(shared_memory, " HELLO WORLD ");
}

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

```
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Выводы

Программа демонстрирует использование разделяемой памяти и сигналов для межпроцессного взаимодействия. Реализована проверка корректности ввода данных (строка должна начинаться с заглавной буквы). Программа корректно завершает работу, освобождая все ресурсы (память, файловые дескрипторы). Приложение может быть расширено для выполнения более сложных задач, таких как обработка больших объемов данных или параллельные вычисления.