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-209Б-23

Студент: Калинин И.Н.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 15.12.24

Москва, 2024

Постановка задачи

Вариант 20.

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 pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Правило фильтрации: строки длины больше 10 символов отправляются в pipe2, иначе в pipe1. Дочерние процессы инвертируют строки.

Взаимодействие между процессами осуществляется через системные сигналы/события и/или через отображаемые файлы (memory-mapped files).

Общий метод и алгоритм решения

Использованные системные вызовы:

- **pid_t fork(void)** — создаёт дочерний процесс. В программе используется для создания двух дочерних процессов, каждый из которых будет выполнять свою часть задачи.
- **int execl(const char *path, const char *arg, ...)** — загружает и исполняет новый образ программы. Дочерние процессы запускают программу child, передавая имя объекта разделяемой памяти в качестве аргумента.
- **int open(const char *pathname, int flags, mode_t mode)** — открытие/создание файла. В родительском процессе открываются два файла для записи результатов обработки данных дочерними процессами.
- **close(int fd)** — закрывает файл. Используется для закрытия всех открытых файлов в родительском и дочерних процессах после их использования.
- **int shm_open(const char *name, int oflag, mode_t mode)** — создаёт или открывает разделяемую память. Родительский процесс создаёт два объекта разделяемой памяти для обмена данными с каждым из дочерних процессов.
- **int shm_unlink(const char *name)** — удаляет разделяемую память по имени. Используется для очистки системных ресурсов после завершения работы всех процессов.
- **int ftruncate(int fd, off_t length)** — изменяет размер открытого файла. Применяется для установки размера разделяемой памяти, достаточного для размещения структуры данных.
- **void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)** — сопоставляет область памяти с файлом. Используется для отображения разделяемой памяти в адресное пространство процессов, чтобы они могли взаимодействовать с ней.
- **int munmap(void *addr, size_t length)** — отменяет сопоставление области памяти. Применяется для отмены отображения разделяемой памяти после завершения работы.
- **int sem_post(sem_t *sem)** — сигнализирует (разблокирует) семафор. Родительский процесс использует sem_post для подачи сигнала дочернему процессу, что данные готовы для обработки.
- **int sem_wait(sem_t *sem)** — ожидает (блокируется) на семафоре. Дочерний процесс использует sem_wait, чтобы дождаться сигнала от родителя перед началом обработки данных.

Алгоритм работы программы

Программа создает два дочерних процесса, используя fork(). Родительский процесс устанавливает общую память (shared memory) с помощью shm_open и отображает её в адресное пространство с помощью mmap. Для синхронизации между процессами используются семафоры. Родительский процесс считывает строки ввода, распределяет их между дочерними процессами в зависимости от длины строки, а дочерние процессы обрабатывают данные (реверсируют строки) и записывают результат в соответствующие файлы.

Код программы

main.cpp

```cpp
#include <iostream>
#include <string>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <sys/wait.h>
#include <cstring>
#include <cstdlib>

//обмен данными через отображаемый файл
struct shared_data {
    sem_t sem_parent;
    sem_t sem_child;
    char buffer[1024];
    int terminate;
};

int main() {
    std::cout << "name for child process 1: ";
    std::string file1_name;
    std::getline(std::cin, file1_name);

    std::cout << "name for child process 2: ";
    std::string file2_name;
    std::getline(std::cin, file2_name);

    int file1 = open(file1_name.c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
    int file2 = open(file2_name.c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file1 < 0 || file2 < 0) {
        perror("Can't open file");
        exit(1);
    }

    const char *shm_name1 = "/shm_child1";
    const char *shm_name2 = "/shm_child2";

    int shm_fd1 = shm_open(shm_name1, O_CREAT | O_RDWR, 0666);
    int shm_fd2 = shm_open(shm_name2, O_CREAT | O_RDWR, 0666);

    if (shm_fd1 == -1 || shm_fd2 == -1) {
```

```
        perror("Can't create shared memory object");
        exit(1);
    }

    ftruncate(shm_fd1, sizeof(shared_data));
    ftruncate(shm_fd2, sizeof(shared_data));

    shared_data *shm_ptr1 = (shared_data *) mmap(NULL, sizeof(shared_data), PROT_READ |
PROT_WRITE, MAP_SHARED, shm_fd1, 0);
    shared_data *shm_ptr2 = (shared_data *) mmap(NULL, sizeof(shared_data), PROT_READ |
PROT_WRITE, MAP_SHARED, shm_fd2, 0);

    if (shm_ptr1 == MAP_FAILED || shm_ptr2 == MAP_FAILED) {
        perror("Can't mmap shared memory");
        exit(1);
    }

    sem_init(&shm_ptr1->sem_parent, 1, 0);
    sem_init(&shm_ptr1->sem_child, 1, 0);
    sem_init(&shm_ptr2->sem_parent, 1, 0);
    sem_init(&shm_ptr2->sem_child, 1, 0);

    shm_ptr1->terminate = 0;
    shm_ptr2->terminate = 0;

    pid_t pid1 = fork();
    if (pid1 < 0) {
        perror("Can't fork");
        exit(1);
    }

    if (pid1 == 0) {
        munmap(shm_ptr2, sizeof(shared_data));
        close(shm_fd2);

        if (dup2(file1, STDOUT_FILENO) < 0) {
            perror("Can't redirect stdout for child process 1");
            exit(1);
        }
        close(file1);
        close(file2);

        execl("./child", "./child", shm_name1, NULL);
        perror("Can't execute child process 1");
        exit(1);
    }

    pid_t pid2 = fork();
    if (pid2 < 0) {
        perror("Can't fork");
        exit(1);
    }

    if (pid2 == 0) {
```

```
        munmap(shm_ptr1, sizeof(shared_data));
        close(shm_fd1);

        if (dup2(file2, STDOUT_FILENO) < 0) {
            perror("Can't redirect stdout for child process 2");
            exit(1);
        }
        close(file2);
        close(file1);

        execl("./child", "./child", shm_name2, NULL);
        perror("Can't execute child process 2");
        exit(1);
    }

    close(file1);
    close(file2);

    while (true) {
        std::string s;
        std::getline(std::cin, s);

        if (s.empty()) {
            shm_ptr1->terminate = 1;
            shm_ptr2->terminate = 1;

            sem_post(&shm_ptr1->sem_parent);
            sem_post(&shm_ptr2->sem_parent);
            break;
        }

        if (s.size() > 10) {
            strcpy(shm_ptr2->buffer, s.c_str());
            sem_post(&shm_ptr2->sem_parent);
            sem_wait(&shm_ptr2->sem_child);
        } else {
            strcpy(shm_ptr1->buffer, s.c_str());
            sem_post(&shm_ptr1->sem_parent);
            sem_wait(&shm_ptr1->sem_child);
        }
    }

    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    sem_destroy(&shm_ptr1->sem_parent);
    sem_destroy(&shm_ptr1->sem_child);
    sem_destroy(&shm_ptr2->sem_parent);
    sem_destroy(&shm_ptr2->sem_child);

    munmap(shm_ptr1, sizeof(shared_data));
    munmap(shm_ptr2, sizeof(shared_data));
    close(shm_fd1);
    close(shm_fd2);
```

```
    shm_unlink(shm_name1);
    shm_unlink(shm_name2);

    return 0;
}
```

Child.cpp

```
#include <iostream>
#include <string>
#include <algorithm>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <cstring>
#include <cstdlib>

struct shared_data {
    sem_t sem_parent;
    sem_t sem_child;
    char buffer[1024];
    int terminate;
};

int main(int argc, char *argv[]) {
    if (argc != 2) {
        std::cerr << "Usage: ./child <shm_name>" << std::endl;
        return 1;
    }

    const char *shm_name = argv[1];

    int shm_fd = shm_open(shm_name, O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("Can't open shared memory object");
        exit(1);
    }

    shared_data *shm_ptr = (shared_data *) mmap(NULL, sizeof(shared_data), PROT_READ |
PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (shm_ptr == MAP_FAILED) {
        perror("Can't mmap shared memory");
        exit(1);
    }

    while (true) {
        sem_wait(&shm_ptr->sem_parent);
```

```
        if (shm_ptr->terminate) {
            sem_post(&shm_ptr->sem_child);
            break;
        }

        std::string str(shm_ptr->buffer);
        std::reverse(str.begin(), str.end());

        std::cout << str << std::endl;
        std::cout.flush();

        sem_post(&shm_ptr->sem_child);
    }

    munmap(shm_ptr, sizeof(shared_data));
    close(shm_fd);

    return 0;
}
```

Протокол работы программы

Тестирование:

root@1354a8b719e2:/workspaces/os_base/lab3# ./main

Enter file's name for child process 1: one

Enter file's name for child process 2: two

123456789

12345678900

Первое число запишется в one, второе в two

Вывод

При выполнении лабораторной работы возникли сложности с корректной инициализацией семафоров и синхронизацией между процессами, что приводило к дедлокам. Также было непросто отлаживать работу с общей памятью и убедиться, что все ресурсы правильно освобождаются после завершения работы. Было бы полезно иметь более подробную документацию по использованию mmap и семафоров в многопроцессных приложениях, а также примеры использования strace для отслеживания специфичных системных вызовов.