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: Теребаев К. Д.
Группа: М8О-203Б-22
Преподаватель: Миронов Е. С.
Дата: _____
Оценка: _____
Подпись: _____

Москва, 2024

Цель работы

Приобретение практических навыков в:

- Освоение принципов работы с файловыми системами
- Обеспечение обмена данных между процессами посредством технологии «File mapping»

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 через отображаемые файлы (memory-mapped files).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы. 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 через отображаемые файлы (memory-mapped files).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

2 вариант) Пользователь вводит команды вида: «число число число». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс считает их сумму и выводит её в файл. Числа имеют тип float. Количество чисел может быть произвольным.

Общие сведения о программе

1. write () - переписывает count байт из буфера в файл. Возвращает количество записанных байт или -1;
2. mmap () – отображает файл на память. Возвращает указатель на начало файла, при ошибке возвращает MAP_FAILED;
3. munmap () – отменяет отображение файла на память. В случае ошибки возвращает -1;
4. shm_open () – открывает или создает при необходимости объект разделяемой памяти. Возвращает дескриптор открытого файла или -1;
5. shm_unlink () – обратная к shm_open;
6. sem_open () – инициализирует и открывает именованный семафор;
7. sem_close () – обратная к sem_open;
8. sem_post () – увеличивает (разблокирует) семафор. Возвращает 0 при успехе и -1 при неудаче;
9. sem_wait () – уменьшает (блокирует) семафор. Возвращает 0 при успехе и -1 при неудаче;
10. ftruncate () – устанавливает файлу заданную длину в байтах.
11. open () - открывает или создаёт файл при необходимости. Возвращает дескриптор открытого файла или -1;
12. close () - закрывает файловый дескриптор, который больше не ссылается ни на один файл, возвращает 0 или -1;
13. fork () - порождается процесс-потомок. Весь код после fork () выполняется дважды, как в процессе-потомке, так и в процессе-родителе. Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork (). Процесс-родители возвращает идентификатор pid потомка или -1. Процесс-потомок возвращает 0 или -1.

Общий алгоритм решения

Используя системный вызов mmap будет отображать файл на память. Теперь наш файл представляет собой массив символов, в него родительский процесс будет записывать, введенные пользователем строчки, а дочерний – читать и проверять на валидность правилу. Чтобы синхронизовать их работу используем семафоры, благодаря им дочерний процесс сможет понимать, что родитель записал в файл строчку, а родительский процесс поймет, что ребенок проверил ее. Если строчка не удовлетворяет условию, то дочерний процесс запишет в конец файла константу, каждый раз родительский процесс проверяет последний элемент файла, в случае если в конце файла оказался символ EOF, то мы ждем завершения работы дочернего процесса и программа останавливает свою работу.

Основные файлы программы

main.cpp

```
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/wait.h>

#include <iostream>

#include "shared_data.hpp"

using namespace std;

int main() {
    int err;
    string file;
    cout << "Enter filename: ";
    cin >> file;
    cout << "Enter commands:" << endl;

    int shm_fd = shm_open(shm_name, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    throw_if(shm_fd, "Shared memory open error");

    err = ftruncate(shm_fd, sizeof(SharedData));
    throw_if(err, "Shared memory truncate error");

    SharedData* data = (SharedData*)mmap(NULL, sizeof(SharedData), PROT_WRITE,
MAP_SHARED, shm_fd, 0);
    if (data == MAP_FAILED) {
        throw runtime_error("Error displaying shared memory");
    }

    err = sem_init(&data->sem1, 1, 0);
    throw_if(err, "Semaphore init error");

    err = sem_init(&data->sem2, 1, 0);
    throw_if(err, "Semaphore init error");
```

```
    pid_t pid = fork();
    throw_if(pid, "Fork failed");
    if (pid == 0) {
        err = execl("./child_process", "./child_process", file.c_str(), NULL);
        throw_if(err, "Child file error");
    } else {
        string s, out = "";
        while (getline(cin, s)) {
            if (s.size()) {
                s += '\n';
            }
            out += s;
        }

        data->data[15] = '\0';
        for (size_t i = 0; i < out.size(); ++i) {
            if (!(i % 15)) {
                err = sem_post(&data->sem1);
                throw_if(err, "Semaphore post error");
                err = sem_wait(&data->sem2);
                throw_if(err, "Semaphore wait error");
            }
            data->data[i % 15] = out[i];
        }
        if (!out.size() % 15) {
            data->data[out.size() % 15] = '\0';
        }
        data->end = true;
        err = sem_post(&data->sem1);
        throw_if(err, "Semaphore post error");

        wait(nullptr);
    }
    err = sem_destroy(&data->sem1);
    throw_if(err, "Semaphore destroy error");
    err = sem_destroy(&data->sem2);
    throw_if(err, "Semaphore destroy error");

    munmap(data, sizeof(SharedData));
    shm_unlink(shm_name);
}
```

child_process.cpp

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

```cpp
#include <iostream>

#include "shared_data.hpp"

using namespace std;

int main(int argc, char* argv[]) {
    int file_d = open(argv[1], O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

    if (file_d == -1) {
        cerr << "Creating file error" << endl;
        return 1;
    }

    if (dup2(file_d, fileno(stdout)) == -1) {
        cerr << "Dup2 error" << endl;
        return 1;
    }

    int shm_fd = shm_open(shm_name, O_RDWR, S_IRUSR | S_IWUSR);
    throw_if(shm_fd, "Shared memory open error");

    SharedData* data = (SharedData*)mmap(NULL, sizeof(SharedData), PROT_READ |
PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (data == MAP_FAILED) {
        throw std::runtime_error("Shared memory map error");
    }

    string in;
    while (1) {
        int err = sem_wait(&data->sem1);
        throw_if(err, "Semaphore wait error");
        in += data->data;
        err = sem_post(&data->sem2);
        throw_if(err, "Semaphore post error");
        if (data->end) {
            break;
        }
    }

    float res = 0;
    size_t pos = 0;
    for (size_t i = 0; i < in.size(); ++i) {
        if (in[i] == '\n' && (i - pos)) {
            res += stof(in.substr(pos, i - pos));
            pos = i + 1;
            cout << res << endl;
            res = 0;
        } else if (isspace(in[i]) && (i - pos)) {
            res += stof(in.substr(pos, i - pos));
            pos = i + 1;
```

```
            }
        }

        munmap(data, sizeof(SharedData));
        shm_unlink(shm_name);
        fclose(stdout);
        return 0;
    }
```

Пример работы

Ввод

```
● lw1/build [main•] » ./main
  output.txt
  1 2 3 4 5 6 6 7
  1.5 1.5 1.5
```

Вывод (файл output.txt)

```
34
4.5
```

Вывод

В ходе выполнения лабораторной работы я изучил технологию «File mapping» и применил на практике семафоры. Применение этой технологии передачи данных имеет преимущество над pipe’ом, так как к этим данным будут иметь доступ все процессы. Но тут же возникает основная проблема: синхронизация. Нужно точно понимать, как будет работать программа, чтобы правильно синхронизировать работу процессов.