Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу

«Операционные системы»

Тема работы

Студент: Волков Евгений Андреевич
Группа: М8О-207Б-21
Вариант: 17
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

Содержание

Репозиторий

Постановка задачи

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.

Группа вариантов 5

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. parent child1 pipe1 In/out User In Out child2 In Out File Open with write mode pipe2 File 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.

Вариант 17

Правило фильтрации: строки длины больше 10 символов отправляются в pipe2, иначе в pipe1. Дочерние процессы удаляют все гласные из строк.

Общие сведения о программе

Программа представлена двумя файлами: main.cpp, child.cpp.

Общий метод и алгоритм решения

Опишу новые для себя системные вызовы:

shm_open

<sys/stat.h> + <fcntl.h>

Создает и открывает объект общей памяти POSIX, который эффективен для работы с несвязанными процессами, которые хотят использовать единый объект памяти. С флагом O_RDWR - открывает объект на чтение и запись. O_CREAT - создает объект, если он не существует. Аргумент mode означает права доступа, я их установил в переменной accessPerm, установив 644. В случае ошибки возвращает -1.

sem_open

<semaphore.h> + <fcntl.h>

Создает новый семафор POSIX, или открывает уже существующий. Семафор - число, не меньше 0. Семафоры можно уменьшать (sem_wait) и увеличивать (sem_post). При этом если применить операцию sem_wait к семафору, когда его значение 0, то sem_wait блокирует работу, пока значение не увеличится (для чего они и создавались). Именованные семафоры, также, как и объекты общей памяти, лежат на диске в директории /dev/shm. Если устанавлен аттрибут O_CREAT и семафор при этом существует, то аттрибуты значения и прав доступа игнорируются.

ftruncate

<unistd.h>

Устанавливает необходимую длину файла в байтах.

fstat

<sys/stat.h> + <sys/types.h>

Содержит информацию о файле, например, размер st_size, и заполняет буфер.

mmap

<sys/mman.h>

Создает отображение файла на память в пространстве процесса.

Алгоритм решения:

Алгоритм работы с процессами аналогичен ЛР2 кроме способа обмена сообщениями. Вместо pipes использовался memory map и семафоры для синхронизации процессов. У нас есть 2 объекта разделяемой памяти и 2 семафора для каждого процесса. Для начала в файле main.cpp удалим старые файлы семафоров и объектов разделяемой памяти. Потом создадим семафоры. Передадим в дочерние процессы названия файлов семафоров и объектов разделяемой памяти.

В main.cpp устанавливаем семафорам значение 1, если входная строка <=10, записываем ее в 1 объект, уменьшаем 1 семафор, иначе 2 объект и втрой семафор.

В child.cpp мы ждем, когда значение семафора станет равно 0 (уменьшаем в main) и когда дождались удаляем гласные и пишем в файл, затем увеличиваем семафор чтобы main мог дальше вводить строки.

Исходный код

main.cpp

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

using namespace std;
```

```
int main(int argc, char const *argv[])
{
    string current_str;
    int child_tag;
    string child1, child2;
    cout << "Enter the name for first child file: ";
    cin >> child1;
    cout << "Enter the name for second child file: ";
    cin >> child2;

    shm_unlink("1.back"); // удаляем старые объекты разделяемой памяти из директории /dev/shm
    shm_unlink("2.back");
    sem_unlink("_sem1"); // также удаляем старые семафоры
    sem_unlink("_sem2");
    // создаем семафоры по названию, даем им права на доступ, присваиваем им значение 1
    sem_t *sem1 = sem_open("_sem1", O_CREAT, S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH, 1);
    sem_t *sem2 = sem_open("_sem2", O_CREAT, S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH, 1);
    int state = 0;        // переменная для получения значения семафора
    pid_t f_id1 = fork(); // создаем 1 процесс
    if (f_id1 == -1)
    {
        cout << "Fork error with code -1 returned in the parent, no child_1 process is created" << endl;
        exit(EXIT_FAILURE);
    }
    else if (f_id1 == 0) // в 1 процессе
    {
        sem_close(sem1); // закрываем семафор 1 в основном процессе
        string child = child1;
        // передаем в 1 дочерний процесс название объекта разделяемой памяти, семафора и файла
        execlp("./child", child.c_str(), "1.back", "_sem1", NULL);
        perror("Execlp error");
        return 0;
    }
    else
    {
        pid_t f_id2 = fork();
        if (f_id2 == -1)
```

```
    {
        cout << "Fork error with code -1 returned in the parent, no child_2 process is created" << endl;
        exit(EXIT_FAILURE);
    }
    else if (f_id2 == 0)
    {
        sem_close(sem2);
        string child = child2;
        execlp("./child", child.c_str(), "2.back", "_sem2", NULL);
        perror("Execlp error");
        return 0;
    }

    else
    {
        while (getline(std::cin, current_str))
        {
            int s_size = current_str.size() + 1;    // получаем размер строки
            char *buffer = (char *)current_str.c_str(); // получаем строку
            if (current_str.size() <= 10)
            {
                // открываем объект разделяемой памяти
                int fd = shm_open("1.back", O_RDWR | O_CREAT, S_IWUSR | S_IRUSR | S_IRGRP |
S_IROTH);
                ftruncate(fd, s_size); // устанавливаем размер файла на размер строки
                // отображаем данные из файла в оперативную память
                char *mapped = (char *)mmap(NULL, s_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
                memset(mapped, '\0', s_size); // заполняем объект '\0'
                sprintf(mapped, "%s", buffer);  // пишем туда строку
                munmap(mapped, s_size); // удаляем отображение
                close(fd); // закрываем файл
                sem_wait(sem1); // уменьшаем семафор на 1
            }
            else
            {
                int fd = shm_open("2.back", O_RDWR | O_CREAT, S_IWUSR | S_IRUSR | S_IRGRP |
S_IROTH);
```

```
                ftruncate(fd, s_size);
                char *mapped = (char *)mmap(NULL, s_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
                memset(mapped, '\0', s_size);
                sprintf(mapped, "%s", buffer);
                munmap(mapped, s_size);
                close(fd);
                sem_wait(sem2);
            }
        }
    }
  }
  sem_close(sem1);
  sem_close(sem2);
  return 0;
}
```

child.cpp

```
#include <iostream>
#include <string>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <fcntl.h>
#include <string.h>
#include <fstream>
#include <set>

using namespace std;

int main(int argc, char const *argv[])
{
  char *semFile = (char *) argv[2];
  sem_t *sem = sem_open(semFile,  O_CREAT, S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH, 0);
  std::string vovels = "aoueiy";
  std::set<char> volSet(vovels.begin(), vovels.end());
  string filename = argv[0];
```

```
    fstream cur_file;
    cur_file.open(filename, fstream::in | fstream::out | fstream::app);
    char *backfile = (char *) argv[1];
    int state = 1; // переменная для получения значения семафора
    while (1)
    {
        sem_getvalue(sem, &state); // получаем значение семафора в переменную state
        if (state == 0) {    // ждем пока state != 0
            int fd = shm_open(backfile, O_RDWR | O_CREAT, S_IWUSR | S_IRUSR | S_IRGRP |
S_IROTH);
            struct stat statBuf;
            fstat(fd, &statBuf); // получаем данные о файле
            int size_of_str = statBuf.st_size; // получаем размер файла из структуры
            ftruncate(fd, size_of_str); // устанавливаем размер файла
            char *mapped = (char *) mmap(NULL, size_of_str, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0); // отображаем файл в память
            std::string allocated = mapped; // исходная строка
            string result_str;
            for (int i = 0; i < size_of_str; i++) {
                if (volSet.find(std::tolower(allocated[i])) == volSet.cend()) {
                    result_str.push_back(allocated[i]);
                }
            }
            cur_file << result_str << endl;
            close(fd);
            munmap(mapped, size_of_str);
            sem_post(sem);
        }
    }
    sem_close(sem);
    return 0;
}
```

Демонстрация работы программы

ggame@ggame:~/hubs/newos/evgeny/OC-labs/lab4/build$ ./main

Enter the name for first child file: 1.txt

Enter the name for second child file: 2.txt

efwegfweg

aass

ewfgwef

s

a

resdgyujesdtgyhujiestfghujiesdrgyhjisedrgy\

fqtewfewtfwqfeefwe

qtwftqdfqtwftd

sdfg

ggame@ggame:~/hubs/newos/evgeny/OC-labs/lab4/build$ cat 2.txt

rsdgjsdtghjstfghjsdrghjsdrg\

fqtwfwtfwqffw

qtwftqdfqtwftd

ggame@ggame:~/hubs/newos/evgeny/OC-labs/lab4/build$ cat 1.txt

fwgfwg

ss

wfgwf

s

sdfg

Выводы

Был изучен механизм memory map, который позволяет разделять память между процессами. Это позволяет реализовать обмен данными между процессами и сделать его более эффективным. Также были изучены семафоры, которые используются для синхронизации работы нескольких процессов. С их помощью можно контролировать доступ к разделяемым ресурсам.