Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Операционные системы" №3

Студент: Пирязев М.А.

Преподаватель: Миронов Е.С.

Группа: М8О-207Б-22

Дата: 8.11.2023

Оценка:

Подпись:

Москва 2023 г.

Оглавление

Цель работы

Приобретение практических навыков в:

- Управлении потоками в ОС
- Обеспечении синхронизации между потоками

Постановка задачи

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

Вариант 2.

См. лабораторная работа №1.

Варианты выбираются такие же как и в лабораторной работе №1.

Правило фильтрации: нечетные строки отправляются в pipe1, четные в pipe2.

Дочерние процессы инвертируют строки

Общие сведения о программе

Цель этой программы - синхронизировать два дочерних процесса, чтобы они могли параллельно записывать данные в два разделяемых сегмента памяти. Родительский процесс контролирует доступ к разделяемым сегментам памяти с помощью семафора. Когда оба дочерних процесса завершают запись данных, родительский процесс освобождает память и завершает программу

Общий алгоритм решения

Эта программа выполняет следующие действия: 1. Открывает два файла для записи. 2. Создает два разделяемых сегмента памяти (mmap) для хранения данных. 3. Создает семафор для синхронизации доступа к разделяемым сегментам памяти. 4. Создает два дочерних процесса. 5. В каждом дочернем процессе перенаправляет стандартный вывод в соответствующий файл и выполняет другую программу (child.out) с передачей имени семафора и имени разделяемого сегмента памяти. 6. В родительском процессе проверяет значение семафора и, если оно равно 2, выполняет следующие действия: - Отображает разделяемые сегменты памяти в память. - Считывает строки с клавиатуры и записывает их в соответствующий разделяемый сегмент памяти. - Устанавливает значение семафора в 1. - Получает размеры разделяемых сегментов памяти и освобождает память. 7. Закрывает файлы и семафоры, завершает программу

Реализация

Родительский процесс:

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

const char* FIRST_MMAP_NAME = "f1";
const char* SECOND_MMAP_NAME = "f2";
const char* SEMAPHORE_NAME = "semaphore";

int get_semaphore_value(sem_t* semaphore)
{
    int value;
    sem_getvalue(semaphore, &value);
    return value;
}

void set_semaphore_value(sem_t* semaphore, int value)
{
    while (get_semaphore_value(semaphore) < value)
    {
        sem_post(semaphore);
    }
    while (get_semaphore_value(semaphore) > value)
    {
        sem_wait(semaphore);
    }
}

void error_checking(int result, char* error)
{
    if (result == -1)
    {
        printf("%s\n", error);
        exit(-1);
    }
}

char* get_string()
{
    int received_length = 0;
    int capacity = 1;
    char* string_for_output = (char*) malloc(sizeof(char));
```

```
    char input_symbol = getchar();

    if (input_symbol == EOF) {
        return NULL;
    }
    if (input_symbol == '\n') {
        input_symbol = getchar();
    }

    while (input_symbol != '\n') {
        if (received_length >= capacity) {
            capacity *= 2;
            string_for_output = (char*) realloc(string_for_output, capacity *
sizeof(char));
        }

        string_for_output[received_length] = input_symbol;
        ++received_length;
        input_symbol = getchar();
    }

    string_for_output[received_length] = '\0';
    return string_for_output;
}


int main()
{
    char* first_filename;
    first_filename = get_string();

    char* second_filename;
    second_filename = get_string();

    int first_child_file, second_child_file;

    error_checking(first_child_file = open(first_filename, O_WRONLY | O_CREAT, 0777),
"File openning error");
    error_checking(second_child_file = open(second_filename, O_WRONLY | O_CREAT, 0777),
"File openning error");

    int first_mmap_file, second_mmap_file;

    error_checking(first_mmap_file = shm_open(FIRST_MMAP_NAME, O_RDWR | O_CREAT, 0777),
"File openning error"); // открывает файл в разделяемой памяти
    error_checking(second_mmap_file = shm_open(SECOND_MMAP_NAME, O_RDWR | O_CREAT,
0777), "File openning error");

    sem_unlink(SEMAPHORE_NAME); // здесь удаляем семафор чтобы открыть наш новый с нуле-
выми параметрами
```

```
    sem_t* semaphore = sem_open(SEMAPHORE_NAME, O_CREAT, 0777, 2);

    pid_t first_process_id = fork();
    error_checking(first_process_id, "Fork error");

    if (first_process_id == 0)
    {
        pid_t second_process_id = fork();
        error_checking(second_process_id, "Fork error");

        if (second_process_id == 0)
        {
            printf("Second child process\n");

            error_checking(dup2(second_child_file, fileno(stdout)), "dup2 error"); //
перенаправление стандартного в second child file
            error_checking(execl("child.out", SEMAPHORE_NAME, SECOND_MMAP_NAME, NULL),
"Execl error");
        }

        else
        {
            printf("First child process\n");

            error_checking(dup2(first_child_file, fileno(stdout)), "dup2 error");
            error_checking(execl("child.out", SEMAPHORE_NAME, FIRST_MMAP_NAME, NULL),
"Execl error");

        }
    }

    else
    {
        printf("Parent process\n");

        if (get_semaphore_value(semaphore) == 2)
        {
            char* first_mmap = (char*) mmap(NULL, getpagesize(), PROT_READ | PROT_WRITE,
MAP_SHARED, first_mmap_file, 0); // отображаем файлы в память
            char* second_mmap = (char*) mmap(NULL, getpagesize(), PROT_READ |
PROT_WRITE, MAP_SHARED, second_mmap_file, 0);

            if (first_mmap == MAP_FAILED)
            {
                printf("%s\n", "Error with creating first_map");
                return -1;
            }

            if (second_mmap == MAP_FAILED)
            {
```

```
                printf("%s\n", "Error with creating second_map");
                return -1;
            }

            int first_position = 0;
            int second_position = 0;
            int first_length = 0;
            int second_length = 0;

            int string_sequence_number = 1;


            char* input_string;
            while ((input_string = get_string()) != NULL)
            {
                int string_length = strlen(input_string);

                if ((string_sequence_number)%2 != 0)
                {
                    first_length += string_length + 1;

                    error_checking(ftruncate(first_mmap_file, first_length), "Ftruncate
error"); //фиксируем количество памяти, которую выделяем для представления первого файла

                    for (int char_index = 0; char_index < string_length; ++char_index)
//в цикле перегоняем строку в представление файла
                    {
                        first_mmap[first_position] = input_string[char_index];
                        ++first_position;
                    }

                    first_mmap[first_position] = '\n';
                    ++first_position;

                }

                else // здесь то же самое для второго представления фала
                {
                    second_length += string_length + 1;

                    error_checking(ftruncate(second_mmap_file, second_length), "Ftrun-
cate error");

                    for (int char_index = 0; char_index < string_length; ++char_index)
                    {
                        second_mmap[second_position] = input_string[char_index];
                        ++second_position;
                    }

                    second_mmap[second_position] = '\n';
```

```
                    ++second_position;
                }
                string_sequence_number++;
            }

            set_semaphore_value(semaphore, 1);

            struct stat first_buffer, second_buffer;//создание структур, которые будут хранить разную информацию о файлах из которых на пригодится размер
            fstat(first_mmap_file, &first_buffer);// собираем информацию о файлах
            fstat(second_mmap_file, &second_buffer);

            int first_mmap_size, second_mmap_size;
            first_mmap_size = first_buffer.st_size;//обращаемся к полю st_size ранее созданной структуры и запоминаем размер первого представления файла
            second_mmap_size = second_buffer.st_size;

            munmap(first_mmap, first_mmap_size);
            munmap(second_mmap, second_mmap_size);
        }

        close(first_child_file);
        close(second_child_file);
        close(first_mmap_file);
        close(second_mmap_file);
        //remove(FIRST_MMAP_NAME);
        //remove(SECOND_MMAP_NAME);
    }

    sem_close(semaphore);
    sem_destroy(semaphore);
    return 0;
}
```

Дочерний процесс:

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

const int MAX_STRING_SIZE = 200;

int get_semaphore_value(sem_t* semaphore)
{
```

```
    int value;
    sem_getvalue(semaphore, &value);
    return value;
}

void set_semaphore_value(sem_t* semaphore, int value)
{
    while (get_semaphore_value(semaphore) < value)
    {
        sem_post(semaphore);
    }
    while (get_semaphore_value(semaphore) > value)
    {
        sem_wait(semaphore);
    }
}

void error_checking(int result, char* error)
{
    if (result == -1)
    {
        printf("%s\n", error);
        exit(-1);
    }
}

char* reverseString(char* input) {
    int length = strlen(input);
    int first_marker = 0;
    int last_marker = length - 1;

    while (first_marker < last_marker) {
        char temp = input[first_marker];
        input[first_marker] = input[last_marker];
        input[last_marker] = temp;
        first_marker++;
        last_marker--;
    }
    input[length] = '\0';
    return input;
}

int main(int argc, char *argv[])
{
    char* semaphore_name = argv[0];
    char* mmap_filename = argv[1];
    int mmap_file;

    sem_t* semaphore = sem_open(semaphore_name, O_RDWR | O_CREAT, 0777);
```

```
    int flag = 1;

    while(flag)
    {
        while(get_semaphore_value(semaphore) == 2)
        {
            continue;
        }

        mmap_file = shm_open(mmap_filename, O_RDWR | O_CREAT, 0777);
        error_checking(mmap_file, "File open error");

        struct stat buffer;
        fstat(mmap_file, &buffer);
        int size = buffer.st_size;

        char* map = (char*) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED,
mmap_file, 0);
        char input_string[MAX_STRING_SIZE];
        char input_string_size = 0;
        for (int index = 0; index < size; ++index)
        {
            char symbol = map[index];

            if (symbol == '\n')
            {
                char* new_string;
                new_string = reverseString (input_string);
                write(fileno(stdout), new_string, sizeof(char) * strlen(new_string));
                write(fileno(stdout), "\n", sizeof(char));
                input_string_size = 0;
                continue;
            }

            input_string[input_string_size] = map[index];
            ++input_string_size;
        }
        sem_unlink(semaphore_name);
        munmap(map, size);
        close(mmap_file);
        flag = 0;
    }

    return 0;
}
```

Пример работы

Test 1

Input	Output
Yacht Zb33 academy string abc BOB Lbv 32_97 EXTRemeLY_Long string 13424374874 \(*^%$&^_ djsan	33bZ cba gnirts thcaY ymedaca vbL BOB 79_23 nasjd _^&$%^*(\ 47847342431 gnirts gnoL_YLemeRTXE

Вывод

Разделяемые сегменты памяти и семафоры являются мощными инструментами для синхронизации и обмена данными между процессами. Использование mmap для отображения разделяемых сегментов памяти в память позволяет эффективно работать с большими объемами данных. Правильная синхронизация доступа к разделяемым данным с помощью семафоров позволяет избежать состояний гонки и обеспечить корректность работы программы. Лабораторная работа помогла понять принципы работы с разделяемой памятью и семафорами, а также их применение в реальных сценариях параллельного программирования