Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

Потоки в операционных системах

Студент	Гришин Алексей Юрьевич
Группа	М8О-208Б-21
Вариант	17
Преподаватель	Соколов Андрей Алексеевич
Оценка	5
Дата	07.11.2022
Подпись	

Москва, 2022.

Постановка задачи

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 C, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решения 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.

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 компилируется из файла `scheduler.c`. Основная логика родительского и дочернего процессов содержится в файлах `manager.c`, `worker.c`. Файл `scheduler.c` связан с файлами `manager.c` и `worker.c` через контракты, организованные в виде заголовочных файлов `manager.h` и `worker.h` соответственно.

Также, используются вспомогательные модули `buffer`, `io_line`, `remove_vowels`. Файл `manager.c` использует заголовочные файлы `io_line/io_line.h`, а файл `worker.c` – `io_line/io_line.h` и `remove_vowels/remove_vowels.h`.

Из системных библиотек, программа использует заголовочные файлы `unistd.h`, `string.h`, `stdlib.h`, `fcntl.h`, `sys/wait.h`, `sys/types.h`, `sys/stat.h`, `sys/mman.h`, `semaphore.h`, `stdbool.h`, `errno.h`.

Системные вызовы

В программе используются следующие системные вызовы:

1. `mmap(addr, length, prot, flags, fd, offset)` – создает отображению на область в логической памяти. В параметре `addr` указывается желаемый адрес расположения отображения. Если область по данному адресу уже занята, то операционная система ищет другое. Параметр `length` обозначает размер занимаемой области. В качестве параметра prot передаются специальные флаги защиты, обозначающие права доступа к файлу: на запись, на чтение, на исполнение. В качестве параметра flags указываются параметры для отображения, среди которых я использовал `MAP_SHARED` и `MAP_ANONYMOUS`. Первый параметр обозначает, что создаваемое отображение будет общее между процессами, второй – что отображаемый файл анонимный. В случае, если не был поставлен флаг `MAP_ANONYMOUS`, в параметр `fd` передается файловый дескриптор отображаемого файла, а параметр `offset` обозначает смещение в файле, после которого вся последующая часть файла отображается в логическую память.

2. `munmap(addr, length)` – освобождает занятую для отображения область логической памяти, находящуюся по адресу и размером `length`. В случае успешного выполнения возвращает `0`, в случае неуспеха – `MAP_FAILED`.
3. `fork()` – создает копию родительского процесса (который вызывает `fork`), которая называется дочерним процессом. После создания нового дочернего процесса оба процесса будут выполнять следующую инструкцию после системного вызова `fork()`. Для дочернего процесса возвращаемым значением будет `0`, для родительского – `id` дочернего процесса. В случае ошибки возвращается отрицательное число.
4. `waitpid(pid, status, options)` – ожидает завершения дочернего процесса. В параметр `pid` передается `id` дочернего процесса, в `status` запишется статус, с которым завершился дочерний процесс. В параметр `options` передаются опции для ожидания такие, как `WNOHANG` (возвращает управление сразу же, если дочерний процесс пока не завершился).
5. `exit(status)` – завершает текущую программу. В параметр status передается статус, с которым завершается программа (обычно, при успешном выполнении записывается значение `0`)
6. `sem_init(sem, pshared, value)` – инициализирует анонимный семафор. Параметр `pshared` является флагом, определяющим, будет ли семафор общим для дочерних процессов при клонировании. В параметр `value` передается начальное значение семафора.
7. `sem_wait(sem)` – уменьшает значение в семафоре на `1`. Если при выполнении операции в семафоре лежит нулевое значение, то процесс “встает в очередь” до тех пор, пока в семафоре не будет положительное значение.
8. `sem_post(sem)` – увеличивает значение в семафоре на `1`

9. `sem_destroy(sem)` – удаляет анонимный семафор в текущем процессе.
10. `write(fd, addr, length)` – записывает в файл, которому соответствует файловый дескриптор `fd`, данные, хранящиеся в логической памяти по адресу `addr` и имеющую размеры в `length` байт.
11. `read(fd, addr, length)` – считывает из файла, которому соответствует файловый дескриптор fd, `length` байт. Считанные данные заносятся в область, находящуюся по адресу `addr`.

Описание

Так как задание лабораторной работы №4 основывается на задании лабораторной работы №2, то решение, соответственно, тоже основано на коде из 2-й лабораторной работы. Изменился только способ общения между родительским и дочерними процессами. Для обеспечения общения между процессами с использованием `memory mapped` файла я реализовал модуль `buffer`, который добавляет тип данных `MappedBuffer` и методы, для работы с буфером: `init_buffer`, `send_data`, `receive_data`, `close_buffer`. По своему описанию буфер практически ничем не отличается от анонимного канала, однако, он основывается на других технологиях. Метод `init_buffer` создает анонимный `memory mapped` файл, который является общим для родительского и дочерних процессов. Для синхронизации операций используется 2 семафора: для обозначения заполненности буфера и его пустоты. Если при вызове метода `send_data` буфер оказался полным вызывающий процесс блокируется до тех пор, пока буфер не очистится. Аналогично, если при вызове операции `receive_data` буфер оказался пустым, вызывающий процесс блокируется до тех пор, пока буфер не заполнится.

Исходный код

`buffer/buffer.c`

```
#include "buffer.h"
#include <string.h>
#include <sys/mman.h>
#include <semaphore.h>

typedef struct __BufferMetadata {
    sem_t empty;
    sem_t full;
    void * buff;
} BufferMetadata;

int init_buffer(MappedBuffer * buffer, size_t size) {
    BufferMetadata * md = mmap(
        NULL, sizeof(BufferMetadata),
```

```
        PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS,
        0, 0
    );

    md -> buff = mmap(
        NULL, size,
        PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS,
        0, 0
    );

    buffer -> pointer = md;
    buffer -> size = size;

    sem_init( & md -> full, 1, 0);
    sem_init( & md -> empty, 1, 1);

    return BUFFER_OK;
}

int send_data(MappedBuffer * buffer, void * data, size_t size) {
    BufferMetadata * md = buffer -> pointer;
    size_t copy_size;

    while (size > 0) {
        copy_size = (size < buffer -> size) ? size : buffer -> size;

        sem_wait( & md -> empty);
        memcpy(md -> buff, data, copy_size);
        sem_post( & md -> full);

        size -= copy_size;
        data += copy_size;
    }

    return BUFFER_OK;
}

int receive_data(MappedBuffer * buffer, void * buff, size_t size) {
    BufferMetadata * md = buffer -> pointer;
    size_t copy_size;

    while (size > 0) {
        copy_size = (size < buffer -> size) ? size : buffer -> size;

        sem_wait( & md -> full);
        memcpy(buff, md -> buff, copy_size);
        sem_post( & md -> empty);

        size -= copy_size;
        buff += copy_size;
```

```
    }

    return BUFFER_OK;
}

void close_buffer(MappedBuffer * buffer) {
    BufferMetadata * md = buffer -> pointer;

    munmap(md -> buff, buffer -> size);
    sem_destroy( & md -> full);
    sem_destroy( & md -> empty);
    munmap(buffer -> pointer, sizeof(BufferMetadata));
}
```

buffer/buffer.h

```
#ifndef BUFFER_H
#define BUFFER_H

#include <stdlib.h>

#define BUFFER_OK 0
#define BUFFER_ERROR_INIT 1
#define BUFFER_ERROR_RECEIVE 2
#define BUFFER_ERROR_SEND 3

struct __MappedBuffer {
    void * pointer;
    size_t size;
};

typedef struct __MappedBuffer MappedBuffer;

int init_buffer(MappedBuffer * buffer, size_t buf_size);

int send_data(MappedBuffer * buffer, void * data, size_t size);

int receive_data(MappedBuffer * buffer, void * buff, size_t size);

void close_buffer(MappedBuffer * buffer);

#endif
```

io_line/io_line.c

```
#include "io_line.h"
#include <stdio.h>
```

```
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

bool __is_endline_symbol(char symbol) {
    return (symbol == EOF || symbol == '\n');
}

bool __try_to_write(Descriptor d, void * data, size_t data_size) {
    return write(d, data, data_size) == data_size;
}

bool __try_to_read(Descriptor d, void * data, size_t data_size) {
    return read(d, data, data_size) == data_size;
}

int __handle_file_error(int error) {
    if (error == EACCES)
        return IO_LINE_ACCESS_DENIED;

    if (error == EBADF)
        return IO_LINE_INVALID_DESCRIPTOR;
}

int __get_char(Descriptor d, char * symbol) {
    if (!__try_to_read(d, symbol, 1 * sizeof(char)))
        return __handle_file_error(errno);

    return IO_LINE_OK;
}

int write_line(Descriptor d, String line) {
    StringSize line_length = strlen(line);

    if (!__try_to_write(d, line, line_length * sizeof(char)))
        return __handle_file_error(errno);

    if (!__try_to_write(d, "\n", 1 * sizeof(char)))
        return __handle_file_error(errno);

    return IO_LINE_OK;
}

int read_line(Descriptor d, String buffer, StringSize buffer_size) {
    char symbol;
    size_t index = 0;
    int result;

    while (1) {
        result = __get_char(d, & symbol);
```

```
        if (result != IO_LINE_OK)
            return result;

        if (__is_endline_symbol(symbol))
            break;

        if (symbol == '\r')
            continue;

        if (index >= buffer_size - 1)
            return IO_LINE_NOT_ENOUGH_BUFFER_SIZE;

        buffer[index++] = symbol;
    }

    buffer[index] = '\0';
    return IO_LINE_OK;
}
```

io_line/io_line.h

```
#ifndef IO_LINE_H
#define IO_LINE_H

#include <stdlib.h>

#define IO_LINE_OK 0
#define IO_LINE_INVALID_DESCRIPTOR 11
#define IO_LINE_ACCESS_DENIED 12
#define IO_LINE_NOT_ENOUGH_BUFFER_SIZE 13

typedef int Descriptor;
typedef char * String;
typedef size_t StringSize;

int write_line(Descriptor d, String line);

int read_line(Descriptor d, String buffer, StringSize buffer_size);

#endif
```

remove_vowels/remove_vowels.c

```
#include "remove_vowels.h"

#include <string.h>
#include <stdbool.h>
```

```
char * __vowels = "eyuioaEYUIOA";

bool __is_vowel(char symbol) {
    for (char * c = __vowels;* c != '\0'; c++)
        if ( * c == symbol)
            return true;

    return false;
}

int remove_vowels(char * s, char * buff, size_t buff_size) {
    int index = 0;
    size_t s_length = strlen(s);
    int ret_value = REMOVE_VOWELS_OK;

    for (int j = 0;
        (j < s_length) && (index < buff_size - 1); j++) {
        if (__is_vowel(s[j]))
            continue;

        buff[index++] = s[j];
    }

    buff[index] = '\0';

    if (index == buff_size - 1)
        return REMOVE_VOWELS_ERROR_NOT_ENOUGH_BUFFER_LEN;
    return REMOVE_VOWELS_OK;
}
```

remove_vowels/remove_vowels.h

```
#ifndef REMOVE_VOWELS_H
#define REMOVE_VOWELS_H

#include <stdlib.h>

#define REMOVE_VOWELS_OK 0
#define REMOVE_VOWELS_ERROR_NOT_ENOUGH_BUFFER_LEN 31

int remove_vowels(char * s, char * buff, size_t buff_size);

#endif
```

manager.c

```
#include "manager.h"
#include "io_line/io_line.h"
```

```
#include <unistd.h>
#include <string.h>

#define MAX_INPUT_STRING_LEN 256

void manager_logic(MappedBuffer workers[2]) {
    char input_string[MAX_INPUT_STRING_LEN];

    while (1) {
        int res = read_line(
            STDIN_FILENO, input_string, MAX_INPUT_STRING_LEN);

        if (res != IO_LINE_OK)
            exit(res);

        if (strlen(input_string) == 0)
            break;

        int i = strlen(input_string) > 10;
        send_data( & workers[i], input_string, strlen(input_string));
    }
}
```

manager.h

```
#ifndef MANAGER_H
#define MANAGER_H

#include "buffer/buffer.h"

void manager_logic(MappedBuffer workers[2]);

#endif
```

worker.c

```
#include "worker.h"
#include "io_line/io_line.h"
#include "remove_vowels/remove_vowels.h"
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define MAX_INPUT_LINE_LEN 256
#define FAILED_TO_OPEN_RESULT_FILE 52

int protected_open_result_file_descriptor(char * filepath) {
```

```
    int descriptor;

    if ((descriptor = open(filepath, O_WRONLY | O_TRUNC)) == -1)
        exit(FAILED_TO_OPEN_RESULT_FILE);

    return descriptor;
}

void protected_get_data(
    MappedBuffer * src,
    char * buff,
    size_t buff_size)
{
    receive_data(src, buff, buff_size);
}

void protected_write_line(int descriptor, char * line) {
    int result = write_line(descriptor, line);

    if (result != IO_LINE_OK)
        exit(result);
}

void protected_format_line(char * input_line, char * buff, size_t
buff_size) {
    int result = remove_vowels(input_line, buff, buff_size);

    if (result != REMOVE_VOWELS_OK)
        exit(result);
}

void worker_logic(MappedBuffer * buff, char * result_filepath) {
    int result_file_descriptor;
    char input_line[MAX_INPUT_LINE_LEN];
    char formatted_line[MAX_INPUT_LINE_LEN];

    result_file_descriptor =
protected_open_result_file_descriptor(result_filepath);
    protected_get_data(buff, input_line, MAX_INPUT_LINE_LEN);

    while (strlen(input_line) != 0) {
        protected_format_line(
            input_line, formatted_line, MAX_INPUT_LINE_LEN);
        protected_write_line(result_file_descriptor, formatted_line);
        protected_get_data(buff, input_line, MAX_INPUT_LINE_LEN);
    }

    close(result_file_descriptor);
    exit(0);
}
```

worker.h

```
#ifndef WORKER_H
#define WORKER_H

#include "buffer/buffer.h"

void worker_logic(MappedBuffer* buff, char* result_filepath);

#endif
```

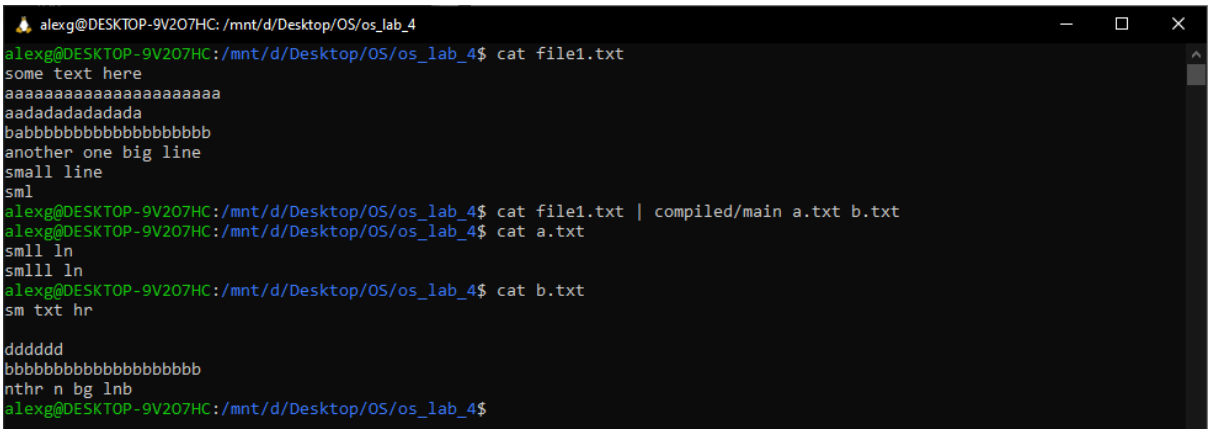
scheduler.c

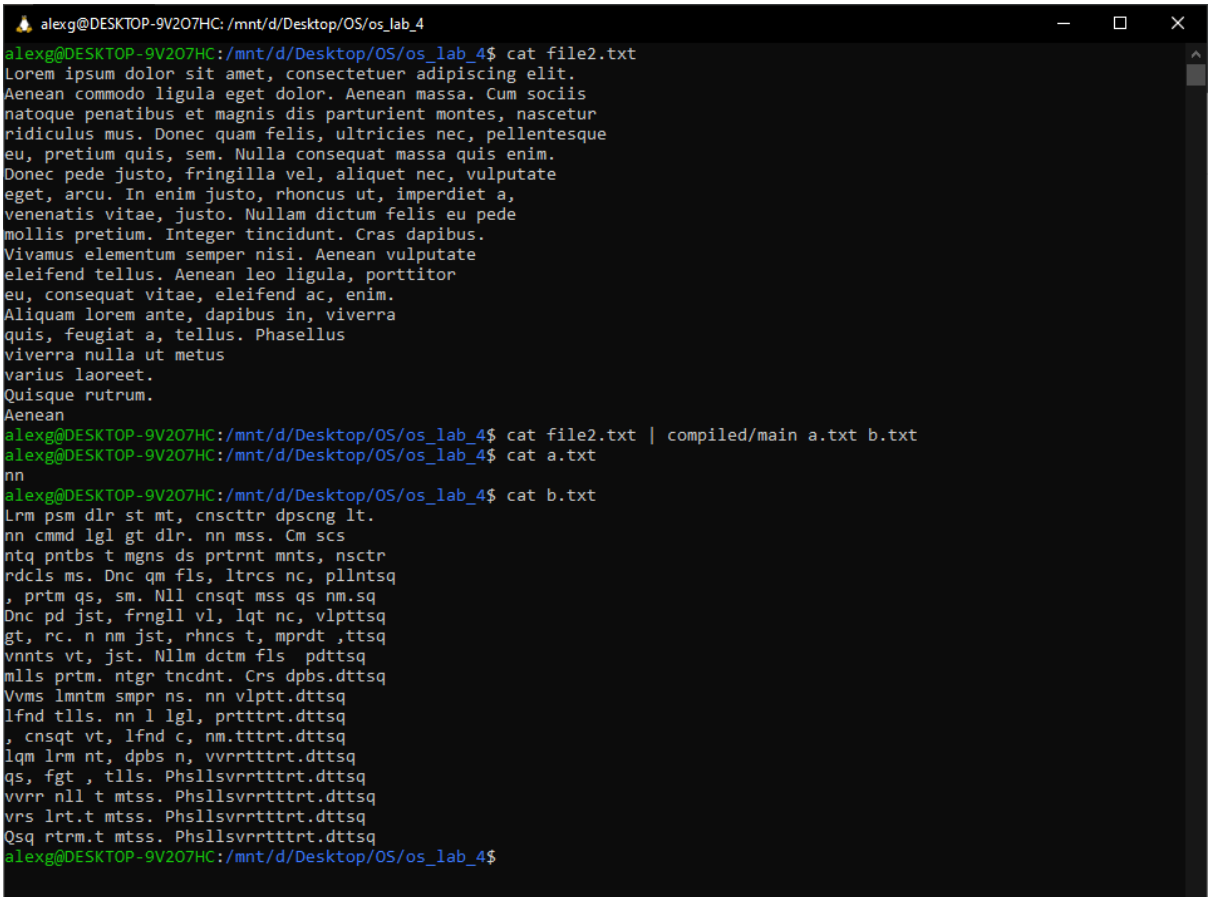
```
#include "manager.h"
#include "worker.h"
#include "buffer/buffer.h"
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>

#define BUFFER_SIZE 256
#define MAX_INPUT_STRING_LEN 256
#define WORKER_PATH "worker"

#define CREATE_PROCESS_ERROR 41
#define MANAGER_ERROR_CREATE_RESULT_FILE 44
#define MANAGER_ERROR_RESULT_FILE_NOT_SPECIFIED 45
#define MANAGER_ERROR_OPEN_RESULT_FILE 46

void protected_create_file(char * filename) {
    int descriptor = open(filename, O_CREAT, S_IWRITE | S_IREAD);

    if (descriptor == -1)
        exit(MANAGER_ERROR_CREATE_RESULT_FILE);

    close(descriptor);
}

char * protected_get_result_file(int argc, char * argv[], int index) {
    int descriptor;

    if (index >= argc)
        exit(MANAGER_ERROR_RESULT_FILE_NOT_SPECIFIED);

    if (access(argv[index], F_OK) != 0)
        protected_create_file(argv[index]);
```

```
    descriptor = open(argv[index], O_WRONLY);

    if (descriptor == -1)
        exit(MANAGER_ERROR_OPEN_RESULT_FILE);

    close(descriptor);
    return argv[index];
}

pid_t create_worker(MappedBuffer * buff, char * result_filepath) {
    pid_t pid = fork();

    if (pid < 0)
        exit(CREATE_PROCESS_ERROR);

    if (pid == 0)
        worker_logic(buff, result_filepath);

    return pid;
}

void close_worker(pid_t worker_id, MappedBuffer * buff) {
    send_data(buff, "\0", 1);
    waitpid(worker_id, NULL, 0);
}

int main(int argc, char * argv[]) {
    char input_string[MAX_INPUT_STRING_LEN];

    char * result_filepaths[2];
    MappedBuffer worker_buffers[2];
    pid_t worker_ids[2];

    for (int i = 0; i < 2; i++)
        result_filepaths[i] = protected_get_result_file(argc, argv, i +
1);

    for (int i = 0; i < 2; i++)
        init_buffer( & worker_buffers[i], BUFFER_SIZE);

    for (int i = 0; i < 2; i++)
        worker_ids[i] = create_worker( & worker_buffers[i],
result_filepaths[i]);

    manager_logic(worker_buffers);

    for (int i = 0; i < 2; i++)
        close_worker(worker_ids[i], & worker_buffers[i]);

    for (int i = 0; i < 2; i++)
        close_buffer( & worker_buffers[i]);
```

```
    exit(0);
}
```

Пример работы





Вывод

В ходе выполнения данной лабораторной работы я ознакомился с технология memory mapped файлов. Они гораздо удобнее анонимных каналов, так как работать с отображением мы можем, как с обычной областью памяти (например, через вызов `malloc`). Memory mapped файлы удобны в случаях, когда необходимо иметь общие данные между процессами. Однако, для задач вида Producer - Consumer, на мой взгляд, эта технология не подходит, так как работа с отображением происходит не синхронизировано, в то время как в `pipe` операции чтения и записи являются синхронизированными.