Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу

«Операционные системы»

РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ И ОБМЕН ДАННЫХ МЕЖДУ ПРОЦЕССАМИ ПОСРЕДСТВОМ ТЕХНОЛОГИИ «FILE MAPPING»

Студент: Суханов Е.А.
Группа: М8О–206Б–19
Вариант: 17
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Постановка задачи

Цель работы

Приобретение практических навыков в:

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

Задание

Требуется составить программу на языке Си, которая:

- Создает два дочерних процесса, которые считывают из стандартного потока ввода и записывают в стандартный поток вывода;
- Отправляет дочерним процессам данные с помощью file mapping;
- Считывает строки из стандартного потока ввода и отправляет их дочерним процессам по правилу.

Правило задано следующим образом: если длина считанной строки больше 10, то она записывается в файл для больших строк. Иначе она записывается в файл для маленьких строк. Кроме этого необходимо отфильтровать гласные буквы. Дочерние процессы должны быть представлены другой программой, у которых выполнена подмена стандартных потоков ввода и вывода.

Для проверки программы нужно составить итоговые тесты. В конечном итоге, программа должна состоять из следующих частей:

- Основная программа, которая запускает дочерние процессы и фильтрует строки из потока ввода;
- Программа дочернего процесса, которая считывает данные из потока ввода, а затем записывает их в поток вывода.

Общие сведения о программе

Для успешной компиляции нужно использовать дополнительные библиотеки: pthread - для использования семафоров и mmap; rt - для общей памяти (shm). Для упрощения процесса компиляции я использовал make.

Исходный код включает следующие файлы:

- parent.c - основной процесс, который запускает дочерний и ожидает ввода строк;
- child.c - дочерний процесс, который обрабатывает полученные строки по правилу и записывает в файл;
- shared.c - содержит функции и структуры данных, обеспечивающих обмен информацией между процессами с помощью общей памяти.
- io.c, io.h - так как запрещено использовать абстракции над системными вызовами ввода и вывода, здесь реализованы более высокоуровневые функции ввода-вывода;

Системные вызовы, которые были использованы:

• open – открывает файл и возвращает связанный с ним файловый дескриптор. Принимает три аргумента: путь, флаги и модификатор доступа. Возвращает -1, если произошла ошибка. С помощью флагов нужно указать каким способом работать с файлом. Файл так же можно создать, используя флаг O_CREAT. В случае создания файла, нужно указать модификаторы доступа;

• close – закрывает файл, связанный с файловым дескрпитором. Принимает в качестве аргумента файловый дескриптор. Возвращает 0, если вызов был успешно выполнен. Иначе -1;

• fork – создает дочерний процесс. Возвращает -1 в случае неудачи. Иначе 0, если это дочерний процесс. PID дочернего процесса, если это родительский. Дочерний процесс является копией родительского: имеет копию памяти, а так же те же самые открытые файловые дескрипторы;

• dup2 – создает дубликат файлового дескриптора. Принимает два аргумента: старый дескриптор и новый дескриптор. При успешном выполнении новый дескриптор станет синонимом старого дескриптора. Возращют номер нового файлового дескриптора, если вызов выполнен успешно. Иначе -1;

• execve – процесс начинает выполнять указанную программу. Данный вызов имеет несколько оболочек с немного разными аргументами для удобства. Принимиает 3 аргумента: путь до исполняемого файла; массив аргументов для программы, который должен заканчиваться NULL; массив параметров среды выполнения, который так же должен заканичваться NULL. При успешном вызове execve не возращает управление. При ошибке возращается -1;

• waitpid – процесс ждет завершения указанного процесса. Принимает три аргумента: id процесса, который нужно ждать; указатель на число, куда будет возращен статус завершенного процесса; Options, которые указывают возращать ли управление сразу или ждать до завершения процесса. Если первый аргумент равен 0, то будет происходить ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса. Если первый аргумент равен -1, то будет происходить ожидание любого дочернего процесса. Если же первый аргумент меньше -1, то будет происходить ожидание любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению pid. Если второй аргумент не равен NULL, то возможно получить полезную информацию о завершенном процессе с помощью макросов. Возвращает -1 в случае ошибки, 0, если ни один из процессов не был завершен (если установлен соотвествующий флаг), либо возращает PID завершенного процесса;

• read – считывает count байт из fd в буффер buf. Возращает количество считанных байт, или -1, в случае ошибки. Количество считанных байт может быть меньше, чем число count;

• write – записывает count байт в fd из буффера buf. Работает аналогично read.

POSIX функции:

- int shm_open(const char *name, int oflag, mode_t mode) - открывает объект разделяемой памяти posix. Где name - имя объекта; oflag и mode - по аналогии с mode. Возвращает файловый дескриптор, по которому можно обратиться к разделяемой памяти. В случае ошибки возвращает -1. Обычно используется вместе с mmap, что бы получить параллельный доступ к памяти;
- int shm_unlink(const char *name) - Закрывает объект разделяемой памяти. Где name - имя объекта. Возвращает -1 в случае ошибки;
- int ftruncate(int fd, off_t length) - устанавливает длину size у заданного файла fd. Возвращает -1 в случае ошибки
- void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset) - создаёт отображение файла на область в памяти. Где start - рекомендуемая область отображения; length - размер области; prot - режим защиты памяти, похоже на режим работы с файлом, только константы по другому называются; flags - режим работы с файлом, мы можем указать MAP_SHARED для того, чтобы изменения были видны другим процессам; fd - файловый дескриптор отображаемого файла, можно указать значение -1 для анонимной области памяти, но это не

будет работать для других процессов после вызова exec; offset - указывает отступ от начала файла. Возвращает указатель на отображение, в случае ошибки возвращает MAP_FAILTURE;

- int munmap(void *start, size_t length) - Освобождает выделенную область памяти. Где start - указатель на отображение файла; length - размер отображения. Возвращает -1 в случае ошибки;
- int sem_init(sem_t *sem, int pshared, unsigned int value) - создаёт семафор, сохраняя его в переменной sem; pshared - указывает возможность работы с другими процессами; value - значение семафора по умолчанию;
- int sem_destroy(sem_t *sem) - уничтожает семафор sem. Возвращает -1 в случае ошибки;
- int sem_wait(sem_t *sem) - уменьшает значение счетчика семафора sem. Если счетчик равен нулю, то поток блокируется, пока кто-то не увеличит его. Возвращает -1 в случае ошибки;
- int sem_post(sem_t *sem) - увеличивает значение счетчика семафора sem. Возвращает -1 в случае ошибки.

Общий метод и алгоритм решения

Основная логика программы не изменилась. Изменился только способ взаимодействия между процессами.

Создание разделяемой области памяти происходит в три этапа:

1. Создание shm объекта;
2. Установка размера общей памяти;
3. Отображение shm с помощью mmap на процессе родителя и ребенка.

Общение обеспечивается с помощью двух семафоров read и write.
Изначально семафор read заблокирован, а write разблокирован:

1. При записи в буфер, родительский процесс блокирует семафор write и, в конце работы, разблокирует семафор read;
2. При чтении из буфера, дочерний процесс блокирует семафор read и , в конце своей работы, разблокирует семафор write.

Обмен информацией происходит с помощью буфера.

Основные файлы программы

parent.c:

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

#include "io.h"
#include "shared.h"

// Имплементирует работу с дочерним процессом
typedef struct
{
    enum
    {
        FP_OK,       // Процесс создан успешно
        FP_ERR_FORK, // Ошибка создания нового процесса
        FP_ERR_PIPE, // Ошибка создания канала
    } status;

    pid_t pid; // PID дочернего процесса
    SharedContainer shared_container; // Указатель на общую память
} FileProc;

// Запускает процесс для работы с файлом file
FileProc run_file_process(const char *file)
{
    FileProc ret;

    // Нужно создать общий канал
    ret.shared_container = shared_init(file);

    // Открытие файла, куда будет писать процесс
```

```
    int fd = open(file, O_WRONLY | O_APPEND | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1)
        log_error_and_exit("Ошибка открытия файла процесса");

    // Создание нового процесса
    if ((ret.pid = fork()) == -1)
    {
        ret.status = FP_ERR_FORK;
        return ret;
    }
    // Логика дочернего процесса
    if (ret.pid == 0)
    {

        if (dup2(fd, STDOUT_FILENO) == -1)
            log_error_and_exit("Ошибка создания дубликата файлового
дескриптора");
        close(fd);

        if (execl("child", "child", file, NULL) == -1)
            log_error_and_exit("Ошибка подмены программы");
    }

    close(fd);
    ret.status = FP_OK;
    return ret;
}

// Отпроваляет msg длины len процессу.
int write_file_process(FileProc *fp, const char *msg, size_t len)
{
    while(len > 0)
    {
        int writed = shared_buf_write(&fp->shared_container, msg, len);
        if(writed < 0)
            return -1;
        len -= writed;
```

```
    }

    return 0;
}

// Завершает работу дочернего процесса
void close_file_process(FileProc *fp, const char* file)
{
    shared_destroy(&fp->shared_container, file);
    waitpid(fp->pid, NULL, 0);
}

// Если есть ошибка, то выводит сообщение о ней и выходит.
// Если ошибки нет, то ничего не делает.
void handle_create_error_file_process(const FileProc *fp)
{
    if (fp->status == FP_ERR_FORK)
        log_error_and_exit("Ошибка создания процесса");
    if (fp->status == FP_ERR_PIPE)
        log_error_and_exit("Ошибка создания канала");
}

void validate_args(int argc, char* argv[])
{
    if(argc != 3)
    {
        log_message(
            "ИСПОЛЬЗОВАНИЕ:\n"
            "\tПрограмма перенаправляет поток ввода в два указанных файла по
правилу:\n"
            "\t\tЕсли длина строки больше 10, то она дописывается в файл
large;\n"
            "\t\tИначе в файл small.\n"
            "\tПрограмма запускается с двумя параметрами:\n"
            "\t\t<small file> <large file>\n"
            "\t\tsmall file -- сюда будут записываться короткие строчки;\n"
            "\t\tlarge file -- сюда будут записываться длинные строчки.\n"
```

```
            "\t\tЕсли указанных файлов не существует -- они будут созданны.\
n"
        );
        exit(1);
    }
}

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

    FileProc fp_small = run_file_process(argv[1]);
    handle_create_error_file_process(&fp_small);

    FileProc fp_large = run_file_process(argv[2]);
    handle_create_error_file_process(&fp_large);

    int len;
    char* line;
    while(line = get_line(&len), len != 0)
    {
        if(len > 10)
        {
            if(write_file_process(&fp_large, line, len) == -1)
                log_error("Не могу записать в файл");
        }
        else
        {
            if(write_file_process(&fp_small, line, len) == -1)
                log_error("Не могу записать в файл");
        }
        free(line);
    }
    close_file_process(&fp_small, argv[1]);
    close_file_process(&fp_large, argv[2]);
    return 0;
}
```

child.c:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <ctype.h>
#include "io.h"
#include "shared.h"

#define BUFSIZE 1024

char is_vowel(char ch)
{
    static const char* vowel = "aeyuio";
    ch = tolower(ch);
    for(int i = 0; i < sizeof(vowel); i++)
        if(ch == vowel[i])
            return 1;

    return 0;
}

ssize_t filter(char *bufin, char *bufout, ssize_t bufin_bytes)
{
    ssize_t bufout_bytes = 0;
    for(ssize_t i = 0; i < bufin_bytes; i++)
    {
        if(!is_vowel(bufin[i]))
            bufout[bufout_bytes++] = bufin[i];
    }
    return bufout_bytes;
}

int main(int argc, char* argv[])
{
    if(argc != 2)
    {
```

```
        //TODO
    }

    SharedContainer shared_container = shared_connect(argv[1]);

    char bufin[SHARED_BUF_SIZE];
    char bufout[SHARED_BUF_SIZE];
    int read_bytes;
    while ((read_bytes = shared_buf_read(&shared_container, bufin)) > 0)
    {
        ssize_t filted_bytes = filter(bufin, bufout, read_bytes);
        if(write_str(STDOUT_FILENO, bufout, filted_bytes) == -1)
            log_error_and_exit("Ошибка записи в файл");
    }
    shared_deconnect(&shared_container);
    return 0;
}
```

shared.h:

```
#pragma once
#include <semaphore.h>
#include <stdbool.h>

#define SHARED_BUF_SIZE 1024


typedef struct
{
    sem_t write_mutex;
    sem_t read_mutex;

    char buf[SHARED_BUF_SIZE];     // Буфер для передачи текста
    int buf_len;                   // Длина текста в буфере
}  Shared;

typedef struct
```

```
{
    int fd;
    Shared* shared;

} SharedContainer;

// Создает SharedContainer, а так же инициализирует его
// Если произошла ошибка, возвращает NULL
SharedContainer shared_init(const char* name);

// Возвращает указатель на общий shared
SharedContainer shared_connect(const char* name);

// Отключает данный shared
void shared_deconnect(SharedContainer* shared_container);

// Уничтожает shared, должен быть вызван на стороне производителя
void shared_destroy(SharedContainer* shared_container, const char* name);

// Запись в сообщения msg длиной len в shared,
// Если длина сообщения len больше, чем
// SHARED_BUF_SIZE, тогда будет записано только SHARED_BUF_SIZE символов.
// Возвращает кол-во записаных символов
// -1, если произошла ошибка, связанная с семафором
// -2 - остальные
int shared_buf_write(SharedContainer* shared_container, const char* msg, int len);

// Запись буфера из shared в buf. Будет считано не более shared->buf_size элементов
// Возращает кол-во считанных символов
// -1, если произошла ошибка, связанная с семафором
// -2 - остальные
// Buf должен иметь как минимум SHARED_BUF_SIZE элементов
int shared_buf_read(SharedContainer* shared_container, char* buf);

/*
1. Открываем общий файл (посмотри, можно ли его сделать безымянным)
```

2. Делаем его маппинг

3. Профит

*/

shared.c:

```
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#include "io.h"
#include "shared.h"

// Создает SharedContainer, а так же инициализирует его
// Если произошла ошибка, возвращает NULL
SharedContainer shared_init(const char* name)
{
   SharedContainer shared_container;
   shared_container.fd = shm_open(name, O_CREAT | O_EXCL | O_RDWR, S_IRUSR | S_IWUSR);
   if(shared_container.fd == -1)
      log_error_and_exit("shm_open");

   if(ftruncate(shared_container.fd, sizeof(Shared)) == -1)
      log_error_and_exit("ftruncate");

   shared_container.shared = mmap(0, sizeof(Shared), PROT_READ | PROT_WRITE, MAP_SHARED,
shared_container.fd, 0);
   if(shared_container.shared == MAP_FAILED)
      log_error_and_exit("mmap");

   if(sem_init(&shared_container.shared->write_mutex, 1, 1) == -1)
   {
      log_error("Не могу создать семафор");
```

```
        munmap(shared_container.shared, sizeof(Shared));
    }
    if(sem_init(&shared_container.shared->read_mutex, 1, 0) == -1)
    {
        log_error("Не могу создать семафор");
        munmap(shared_container.shared, sizeof(Shared));
    }

    shared_container.shared->buf_len = 0;

    return shared_container;
}

// Возвращает указатель на общий shared
SharedContainer shared_connect(const char* name)
{
    SharedContainer shared_container;
    shared_container.fd = shm_open(name, O_RDWR, S_IRUSR | S_IWUSR);
    if(shared_container.fd == -1)
        log_error_and_exit("shm_open");

    shared_container.shared = mmap(0, sizeof(Shared), PROT_READ | PROT_WRITE, MAP_SHARED,
shared_container.fd, 0);
    if(shared_container.shared == MAP_FAILED)
        log_error_and_exit("mmap");

    return shared_container;
}

// Отключает данный shared
void shared_deconnect(SharedContainer* shared_container)
{
    if(shared_container == NULL)
        return;
```

```
    if(munmap(shared_container->shared, sizeof(Shared)) == -1)
        log_error("Не могу освободить общую память");
    if(close(shared_container->fd) == -1)
        log_error("close");
}

// Уничтожает данный shared
void shared_destroy(SharedContainer* shared_container, const char* name)
{
    // Ожидание конца записи сообщения
    sem_wait(&shared_container->shared->write_mutex);
    shared_container->shared->buf_len = -1;
    sem_post(&shared_container->shared->read_mutex);

    if(sem_destroy(&shared_container->shared->write_mutex) == -1)
        log_error("Не могу уничтожить семафор");

    if(sem_destroy(&shared_container->shared->read_mutex) == -1)
        log_error("Не могу уничтожить семафор");

    if(munmap(shared_container->shared, sizeof(Shared)) == -1)
        log_error("Не могу освободить общую память");

    if(close(shared_container->fd) == -1)
        log_error("close");

    if(shm_unlink(name) == -1)
        log_error_and_exit("unlink");
}

// Запись в сообщения msg длиной len в shared,
// Если длина сообщения len больше, чем
// SHARED_BUF_SIZE, тогда будет записано только SHARED_BUF_SIZE символов.
// Возвращает кол-во записаных символов
```

```
// -1, если произошла ошибка, связанная с семафором
// -2 - остальные
int shared_buf_write(SharedContainer* shared_container, const char* msg, int len)
{
    if(shared_container == NULL)
        return -2;
    if(msg == NULL && len != 0)
        return -2;

    // Заблокирвать
    if(sem_wait(&shared_container->shared->write_mutex) == -1)
        return -1;

    if(shared_container->shared->buf_len == -1)
        return -1;

    // Записать данные
    int need_to_write = (len < SHARED_BUF_SIZE) ? len : SHARED_BUF_SIZE;
    for(int i = 0; i < need_to_write; ++i)
    {
        shared_container->shared->buf[i] = msg[i];
    }
    shared_container->shared->buf_len = need_to_write;

    // Разблокировать
    if(sem_post(&shared_container->shared->read_mutex) == -1)
        return -1;

    return need_to_write;
}

// Запись буфера из shared в buf. Будет считано не более shared->buf_size элементов
// Возращает кол-во записанных файлов
// -1, если произошла ошибка, связанная с семафором
```

```
// -2 - остальные
// Buf должен иметь как минимум SHARED_BUF_SIZE элементов
int shared_buf_read(SharedContainer* shared_container, char* buf)
{
   if(shared_container == NULL || buf == NULL)
      return -2;

   // Заблокировать
   if(sem_wait(&shared_container->shared->read_mutex) == -1)
      return -1;

   // Считать данные
   int need_to_read = shared_container->shared->buf_len;
   for(int i = 0; i < need_to_read; ++i)
   {
      buf[i] = shared_container->shared->buf[i];
   }
   shared_container->shared->buf_len = 0;

   // Разблокировать
   if(sem_post(&shared_container->shared->write_mutex) == -1)
      return -1;

   return need_to_read;
}
```

io.h:

```
#pragma once
#include "unistd.h"

// Считывает из потока ввода строку, максимальной длиной max_size,
// Строка должна заканчиваться знаком end.
// Возвращает длину считанной строки.
```

```
int get_line(char* str, int max_size, char end);

// Записывает в поток fd строку str. Возвращает кол-во записанных знаков
// Длина отформатированной строки не должна превышать 1024 символа!
// Возвращает длину записанной строки. Если -1, то возникла какая-то ошибка
int write_str(int fd, char* fomrat, ...);
```

io.c:

```
#include <stdio.h>
#include <stdarg.h>

#include "io.h"

int read_from_readed_buffer(char* out_str, int nbytes) {
    static const int BUF_MAX_SIZE = 4096;
    static char buf[4096]; // Здесь значение BUF_MAX_SIZE // Он думает, что BUF_MAX_SIZE не константное значение
    static int size = 0;
    static int pos = 0;

    int res_len = nbytes;
    while(nbytes > 0) {
        if(size == pos) {
            pos = 0;
            size = read(STDIN_FILENO, buf, BUF_MAX_SIZE);
            if(size < 0)
                return size;
            if(size == 0)
                break;
        }
        *out_str++ = buf[pos++];
        nbytes--;
    }

    return res_len - nbytes;
}
```

```
int get_line(char* str, int max_size, char end) {
    int str_len = 0;
    int max_size_str = max_size - 1;

    for(;str_len < max_size_str; str_len++) {
        if(read_from_readed_buffer(&str[str_len], 1) == 0)
            break;
        if(str[str_len] == end)
            break;
    }
    str[str_len] = '\0';

    return str_len;
}

int write_str(int fd, char* format, ...) {
    const int BUF_MAX_SIZE = 1024;
    char buf[BUF_MAX_SIZE];

    va_list args;
    va_start(args, format);
    int len = vsnprintf(buf, BUF_MAX_SIZE, format, args);
    va_end(args);

    int writed_bytes = 0;
    while(len > 0) {
        int writed = write(fd, buf + writed_bytes, len);
        if(write < 0)
            return -1;
        writed_bytes += writed;
        len -= writed;
    }

    return writed_bytes;
}
```

Пример работы

```
reterer@serv:~/OS/os_exercise_04/src$ make
cc  -c -Wall -pedantic -O2  -c -o parent.o parent.c
cc  -c -Wall -pedantic -O2  -c -o io.o io.c
cc  -c -Wall -pedantic -O2  -c -o shared.o shared.c
gcc -O2 parent.o io.o shared.o -o parent -lpthread -lrt
cc  -c -Wall -pedantic -O2  -c -o child.o child.c
gcc -O2 child.o io.o shared.o -o child -lpthread -lrt
reterer@serv:~/OS/os_exercise_04/src$ ./parent small large
Hello world!
Aloha
reterer@serv:~/OS/os_exercise_04/src$ cat small
lh
reterer@serv:~/OS/os_exercise_04/src$ cat large
Hll wrld!
```

Вывод

Разделяемая память очень быстрый способ взаимодействия между процессами, так как не надо производить лишние копирования и обращения к операционной системе. Однако стоит помнить, что в данном случае возможны гонки данных, поэтому нужно позаботится о синхронизации процессов для чтения и изменения информации.

Функция mmap позволяет делать отображение обычных файлов это может ускорить считывание и запись. Со случаем записи необходимо заранее установить размер окончательного файла.

В конечном итоге можно сказать, что отображение файлов в оперативную память является довольно полезной технологией.