Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

УПРАВЛЕНИЕ ПРОЦЕССАМИ И ОБМЕН ДАННЫХ МЕЖДУ НИМИ

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

Москва, 2020.

Постановка задачи

Цель работы

Приобретение практических навыков в

- Управление процессами в ОС;
- Обеспечение обмена данных между процессами посредством каналов.

Задание

Требуется составить и отладить 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 каналов;
- 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 записыватеся в файл для маленьких строк.

Дочерние процессы должны быть представленны другой программой, у которых выполненна 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

Программа компилируется из файла main.c. Также используется заголовочные файлы: sys/types.h; sys/wait.h; sys/stat.h; unistd.h; fcntl.h; stdio.h; stdlib.h. В программе используются следующие системные вызовы:

- pipe – создает канал. В качестве аргумента принимает указатель на массив из двух элемнетов типа я int. Возвращает 0, если вызов выполнен успешно и -1, если нет. При успешном вызове, первый элемент массива является файловым дескприптором, из которого можно считать данные канала, а второй – в который можно записать;

- 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. Возращает 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.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить системные вызовы, которые указаны выше.
2. Разбить задачу на подзадачи:
    a. Попробовать fork и exec
    b. Открытие файлов
    c. Наладить pipe между процессами
    d. Отправка строки по pipe
    e. Считывание строки
    f. Фильтрация строки
    g. Тестирование
3. Выполнить поставленные подзадачи.
4. Рефаторинг кода.
5. Провести тестирование программы. Если были обнаружены ошибки – вернуться к 3-му пункту.

Основные файлы программы

parent.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"

// 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
    int fd[2]; // Файловые дескрипторы канала
} FileProc;

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

    if (pipe(ret.fd) == -1)
    {
        ret.status = FP_ERR_PIPE;
        return ret;
    }

    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");

    if ((ret.pid = fork()) == -1)
    {
        ret.status = FP_ERR_FORK;
        return ret;
    }
    if (ret.pid == 0)
    {
        close(ret.fd[1]);

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

        if (dup2(ret.fd[0], STDIN_FILENO) == -1)
            log_error_and_exit("Ошибка создания дубликата файлового
дескриптора");
        close(ret.fd[0]);

        if (execl("child", "child", NULL) == -1)
```

```
            log_error_and_exit("Ошибка подмены программы");
    }

    close(fd);
    close(ret.fd[0]);
    ret.status = FP_OK;
    return ret;
}

// Отпроваляет msg длины len процессу.
int write_file_process(const FileProc *fp, const char *msg, size_t len)
{
    return write_str(fp->fd[1], msg, len);
}

// Завершает работу дочернего процесса
void close_file_process(const FileProc *fp)
{
    close(fp->fd[1]);
    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]);
    FileProc fp_large = run_file_process(argv[2]);
    handle_create_error_file_process(&fp_small);
    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_large);
    close_file_process(&fp_small);
    return 0;
}
```

child.c:

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

#define BUFSIZE 1024

char is_vowel(char ch)
{
   static const char* vowel = "eyuio";
   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()
{
   char bufin[BUFSIZE];
   char bufout[BUFSIZE];
   ssize_t read_bytes;
   while ((read_bytes = read(STDIN_FILENO, bufin, sizeof(char) * BUFSIZE)) > 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");
        }
    }
    return 0;
}
```

io.h:

```
// Записывает len байт из msg в fd.

// Возвращает 0, если успешно, иначе -1.
int write_str(int fd, const char *msg, int len);

// Считывает строку, которая будет иметь длину len и возращает ее.
// Не забудь освободить память, выделенную под строку.
char* get_line(int *len);

// Выводит msg (perror) и выходит из программы.
void log_error_and_exit(const char *msg);
// Выводит msg (perror).
void log_error(const char *msg);
// Выводит сообщение msg
void log_message(const char *msg);
```

io.c:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "io.h"

int write_str(int fd, const char *msg, int len)
{
    ssize_t write_bytes = 0;
    while(write_bytes != len)
    {
        ssize_t w = write(fd, msg, len-write_bytes);
        if(w == -1)
            return -1;
        write_bytes += w;
    }
    return 0;
}

char* get_line(int *len)
{
    *len = 0;
    int cap = 4;
    char *line = (char*)malloc(sizeof(char)*cap);
    if(line == NULL)
    {
     log_error("Не удалось выделить память под строку");
     return NULL;
    }

    char ch;
```

```
    while(read(STDIN_FILENO, &ch, 1) == 1)
    {
        line[(*len)++] = ch;
        if(*len >= cap)
        {
            cap *= 2;
            char *new_line = (char*)realloc(line, cap);
            if(new_line == NULL)
            {
                log_error("Не удалось увеличить размер строчки");
                *len = cap-1;
                break;
            }
        line = new_line;
        }
        if(ch == '\n')
        break;
    }
    line[*len] = '\0';
    return line;
}

void log_error_and_exit(const char *msg)
{
    perror(msg);
    exit(1);
}

void log_error(const char *msg)
{
    perror(msg);
}

void log_message(const char* msg)
{
    write_str(STDERR_FILENO, msg, strlen(msg));
}
```

Пример работы

```
reterer@retcom:~/os_lab_2/build$ ./parent less more
Hello world!
small
looooooooooooong
reterer@retcom:~/os_lab_2/build$ cat less
small
reterer@retcom:~/os_lab_2/build$ cat more
Hello world!
looooooooooooong

reterer@retcom:~/os_lab_2/build$ ./parent ./ someting
Ошибка открытия файла процесса: Is a directory

reterer@retcom:~/os_lab_2/build$ ./parent
ИСПОЛЬЗОВАНИЕ:
    Программа перенаправляет поток ввода в два указанных файла по правилу:
        Если длина строки больше 10, то она дописывается в файл large;
        Иначе в файл small.
    Программа запускается с двумя параметрами:
        <small file> <large file>
        small file -- сюда будут записываться короткие строчки;
        large file -- сюда будут записываться длинные строчки.
        Если указанных файлов не существует -- они будут созданны.
```

Вывод

Выполняя данную лабораторную работу я узнал об некоторых системных вызовах. А так же приобрел опыт работы с каналами и процессами. Например, что бы корректно работать с каналом, необходимо закрыть не нужный файловый дескриптор (если нужно только считывать – закрыть fd для записаи и наоборот). Познакомился с семейством вызовов exec. С помощью процессов можно делегировть выполнение задач. Например браузеры используют разные процессы для разных вкладок. Это позволяет обеспечить межпроцессорную защиту, а так же «параллельное» выполнение задач. В *nix подобных системах процессы имеют иерархию. У каждого процесса должен быть родитель.