

Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа № 2
по курсу «Операционные системы»

Студент:	Голубев В.С.
Группа:	М8О-206Б-18
Вариант:	26
Преподаватель:	Миронов Е.С.
Оценка:	
Дата:	

Москва, 2019

1. Постановка задачи

Дочерний процесс при создании принимает имя файла. При работе дочерний процесс получает числа от родительского процесса и пишет их в файл. Родительский процесс создает n дочерних процессов и передает им поочередно числа из последовательности от $1 \dots m$.

Операционная система: Unix.

Целью лабораторной работы является:

Приобретение практических навыков в:

- Управление процессами в ОС
- Обеспечение обмена данными между процессами посредством каналов

Задание

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 решения задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (pipe). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

2. Решение задачи

Используемые системные вызовы:

- **ssize_t write(int fd, const void *buf, size_t count)** - пишет до *count* байт из буфера, на который указывает *buf*, в файле, на который ссылается файловый дескриптор *fd*.

- **ssize_t read(int fd, void *buf, size_t count)** - пытается прочитать *count* байт из файлового дескриптора *fd* в буфер, начинающийся по адресу *buf*. Для файлов, поддерживающих смещения, операция чтения начинается с текущего файлового смещения, и файловое смещение увеличивается на количество прочитанных байт. Если текущее файловое смещение находится за концом файла, то ничего не читается и **read()** возвращает ноль. Если значение *count* равно 0, то **read()** может обнаружить ошибки, описанные далее. При отсутствии ошибок, или если **read()** не выполняет проверки, то **read()** с *count* равным 0 возвращает 0 и ничего не меняет.

- **pid_t fork(void)** - создаёт новый процесс посредством копирования вызывающего процесса. Новый процесс считается *дочерним* процессом. Вызывающий процесс считается *родительским* процессом. Дочерний и родительский процессы находятся в отдельных пространствах памяти. Сразу после **fork()** эти пространства имеют одинаковое содержимое.

- **int close(int fd)** - закрывает файловый дескриптор.

- **int pipe2(int pipefd[2], int flags)** - создаёт однонаправленный канал данных, который можно использовать для взаимодействия между процессами. Массив *pipefd* используется для возврата двух файловых дескрипторов, указывающих на концы канала. *pipefd[0]* указывает на конец канала для чтения. *pipefd[1]* указывает на конец канала для записи. Данные, записанные в конец канала, буферизируются ядром до тех пор, пока не будут прочитаны из конца канала для чтения.

Каждая из функций - родитель и ребёнок - принимают идентификаторы потоков, на чтение и на запись.

Функция `pipe()` создаёт собственный поток обмена данными. При этом в передаваемый ей массив записываются числовые идентификаторы (дескрипторы) двух "концов" потока: один на чтение, другой на запись. Поток (`pipe`) работает по принципу "положенное первым - первым будет считано".

`int pid = fork();` Начиная с этого момента, процессов становится два. У каждого своя память. в процессе-родителе `pid` хранит идентификатор ребёнка. в ребёнке в этой же переменной лежит 0. Далее в каждом случае надо закрыть "лишние" концы потоков. поскольку сама программа теперь существует в двух экземплярах, то фактически у каждого потока появляются вторые дескрипторы.

ДЕМОНСТРАЦИЯ РАБОТЫ ПРОГРАММЫ:

```
vsgolubev@vsgolubev-VirtualBox:~/OS$ ./a.out
```

```
FORK: Success
```

```
Enter name file
```

```
1.txt
```

```
Enter n
```

```
9
```

```
Enter m
```

```
4
```

```
vsgolubev@vsgolubev-VirtualBox:~/OS$ cat <1.txt
```

```
1 2 3 4
```

```
1 2 3 4
```

```
1 2 3 4
```

```
1 2 3 4
```

```
1 2 3 4
```

```
1 2 3 4
```

```
1 2 3 4
```

```
1 2 3 4
```

1 2 3 4

vsgolubev@vsgolubev-VirtualBox:~/OS\$

3. Руководство по использованию программы

Компиляция и запуск программного кода в *Ubuntu* :

```
gcc lab2.c
```

```
./a.out
```

```
Cat < *.txt
```

4. Листинг программы

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

void parentProcess(int* pipe_fd, int m, char *fname) {
    int d;
    close(pipe_fd[0]);
    int i = 1;
    for (i ; i < m; ++i)
        write(pipe_fd[1], &i, sizeof(i));

    close(pipe_fd[1]);
}

void childProcess(int* pipe_fd, char *fname) {
    int d;
    int fd;
    close(pipe_fd[1]);
    fd = open(fname, O_CREAT | O_APPEND | O_WRONLY, S_IWUSR | S_IRUSR);
    dup2(fd, 1);
    while(read(pipe_fd[0], &d, sizeof(d)) > 0) {
        printf("%d ", d);
    }
    printf("\n");
    close(fd);
}

int main(int argc, char const *argv[]) {
    int pipe_fd[2];
    pid_t pid;
    char name_file[20];
    int count_process;
    int m;
    int err = 0;
    err = -1;
    perror("FORK");

    printf("\n=====Enter name file\n");
    scanf ("%s", name_file);
    printf("Enter n\n");
    scanf ("%d", &count_process);
    printf("Enter m\n");
    scanf ("%d", &m);
```

```

int i = 0;

for (i; i < count_process; ++i) {
    if(pipe(pipe_fd) == -1) {
        perror("PIPE");
        err = -2;
    }

    pid = fork();

    if(pid == -1) {
        perror("FORK");
        err = -1;
    }
    else if (pid == 0) {
        childProcess(pipe_fd, name_file);
        break;
    } else
        parentProcess(pipe_fd, m, name_file);

}

return err;
}

```

5. Вывод

При работе с потоками и процессами необходимо быть осторожным, так как после вызова `fork()` все допускаемые ошибки будет вдвойне сложно и неприятно отлавливать и исправлять.