



**Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)**

---

ФАКУЛЬТЕТ Информатика и системы управления  
КАФЕДРА Информационная безопасность (ИУ8)

**Отчёт**  
по лабораторной работе № 3  
по дисциплине «Безопасность систем баз данных»

Выполнил: Аббасалиев Э.Н.,  
студент группы ИУ8-61

Проверил: Зенькович С. А.,  
ассистент каф. ИУ8

## Оглавление

ВСТУПЛЕНИЕ .....	3
1. ОПИСАНИЕ POSIX.....	4
2. ИСПОЛЬЗОВАНИЕ SHELL-СКРИПТА.....	5
3. ИСПОЛЬЗОВАНИЕ ПРИЛОЖЕНИЯ НА C++11 .....	6
ЗАКЛЮЧЕНИЕ .....	9

# ВСТУПЛЕНИЕ

**Цель работы:** Разработать приложение, которое должно уметь:

1. Запускать произвольное приложение.
2. Получать его поток stdout, и:
  - записывать в файл;
  - выводить в консоль.
3. Получать его поток stderr, и:
  - записывать в файл;
  - выводить в консоль.
4. Получать его код завершения.

При этом каждая “опция” п. 2-3 должна быть настраиваемой. Вывод информации из п. 4 должен происходить в консоль, и, если есть открытые на запись файлы - дублироваться в них. Приложение должно быть реализовано в 2-х вариантах:

1. POSIX-совместимый shell скрипт.
2. Приложение на C99/C++11.

# 1. ОПИСАНИЕ POSIX

POSIX (англ. Portable Operating System Interface — переносимый интерфейс операционных систем) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix систем.

Задачи POSIX:

- Содействовать облегчению переноса кода прикладных программ на иные платформы.
- Способствовать определению и унификации интерфейсов заранее при проектировании, а не в процессе их реализации.
- Сохранять по возможности и учитывать все главные, созданные ранее и используемые прикладные программы.
- Определять необходимый минимум интерфейсов прикладных программ для ускорения создания, одобрения и утверждения документов.
- Развивать стандарты в направлении обеспечения коммуникационных сетей, распределенной обработки данных и защиты информации.
- Рекомендовать ограничение использования бинарного (объектного) кода для приложений в простых системах.

Стандарт состоит из четырёх основных разделов.


- **Основные определения** (англ. *Base definitions*) — список основных определений и соглашений, используемых в спецификациях, и список заголовочных файлов языка Си, которые должны быть предоставлены соответствующей стандарту системой.
- **Оболочка и утилиты** (англ. *Shell and utilities*) — описание утилит и командной оболочки sh, стандарты регулярных выражений.
- **Системные интерфейсы** (англ. *System interfaces*) — список системных вызовов языка Си.
- **Обоснование** (англ. *Rationale*) — объяснение принципов, используемых в стандарте.

Интерфейс пользователя с POSIX-системой обеспечивается в большинстве случаев классом программ, именуемых командными интерпретаторами, командными процессорами, командными оболочками или по-простому – shell.

## 2. ИСПОЛЬЗОВАНИЕ SHELL-СКРИПТА

Первый вариант приложения реализован на shell. В качестве аргументов данному скрипту подаются пары ключ-значение (название файла), для переопределения потоков stdin, stdout, stderr и после «--» путь к запускаемому приложению. Если какого-то из аргументов не будет, то ввод и вывод будет производиться через консоль.

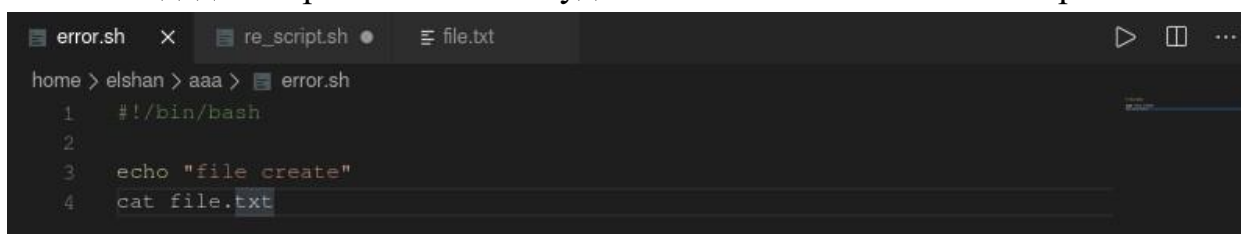
```
$ ./re_script arg1 arg2 arg3
```



```
error.sh  re_script.sh  file.txt
home > elshan > aaa > re_script.sh
1  if [ $1 = "out_console" ]
2  then
3  |   ./$2
4  fi
5
6  if [ $1 = "out_file" ]
7  then
8  |   ./$2 1> file.txt
9  fi
10
11 if [ $1 = "err_file" ]
12 then
13 |   ./$2 2> file.txt
14 fi
15
```

Примеры вызова скрипта:

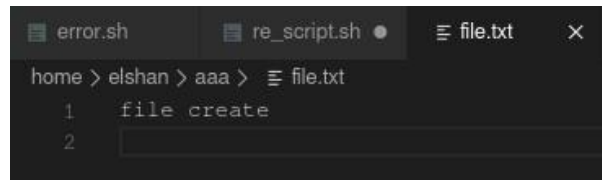
Создадим скрипт error.sh и будем вызывать его в качестве приложения.



```
error.sh  re_script.sh  file.txt
home > elshan > aaa > error.sh
1  #!/bin/bash
2
3  echo "file create"
4  cat file.txt
```

```
[elshan@spaton aaa]$ ./re_script.sh v_console error.sh
cat: file.txt: input file is output file
[elshan@spaton aaa]$ ./re_script.sh out_console error.sh
[elshan@spaton aaa]$ ./re_script.sh out_file error.sh
[elshan@spaton aaa]$ ./re_script.sh err_file error.sh
```

Запишем в файл text.txt:



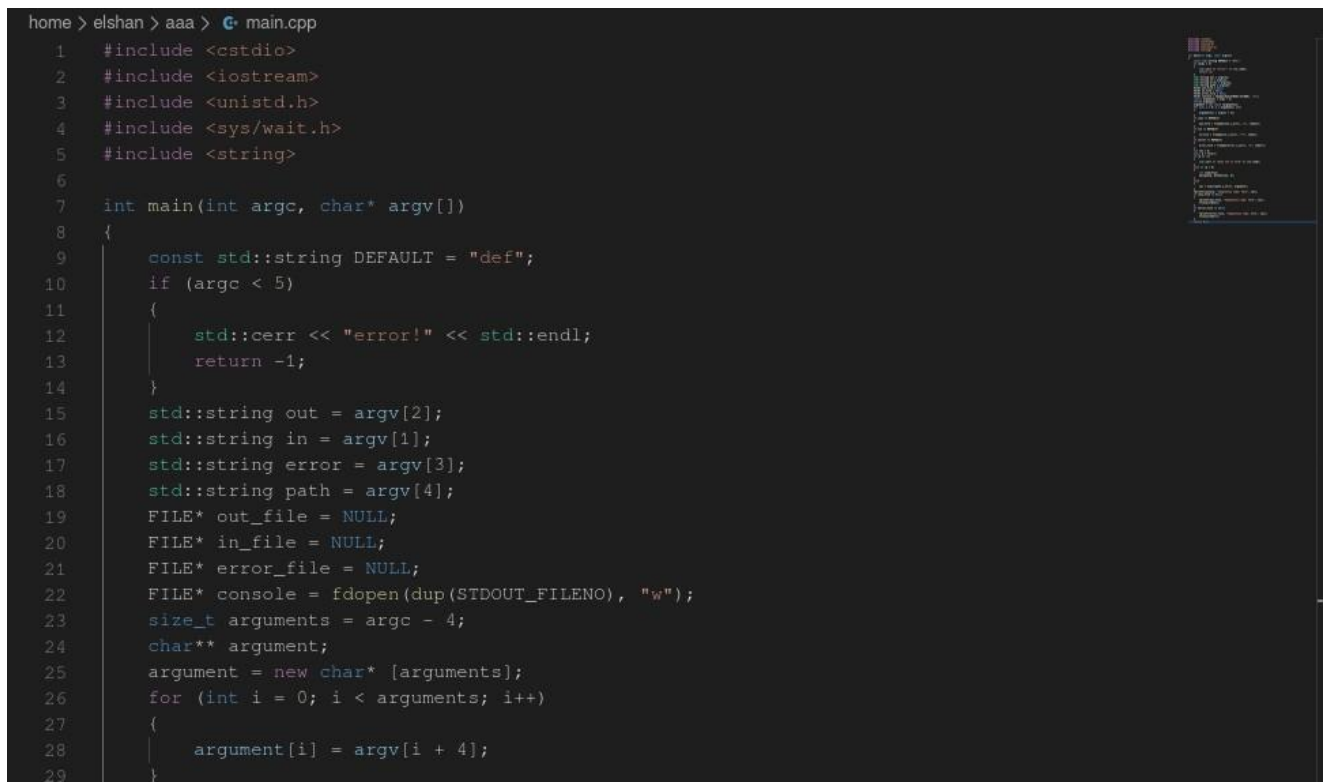
```
error.sh  re_script.sh  file.txt x
home > elshan > aaa > file.txt
1 file create
2
```

### 3. ИСПОЛЬЗОВАНИЕ ПРИЛОЖЕНИЯ НА C++11

Второй вариант приложения, реализован на C++. В качестве аргументов данному скрипту подаются пары ключ-значение (название файла), для переопределения потоков stdin, stdout, stderr и последним аргументом путь к запускаемому приложению. Если какого-то из аргументов не будет, то ввод и вывод будет производиться через консоль.

Данное приложение аналогично предыдущему, но нам необходимо создавать дочерний процесс, так-как вызов `exec1` прерывает выполнение основного процесса.

```
$ ./main def out.txt err.txt ./test
```



```
home > elshan > aaa > main.cpp
1 #include <cstdio>
2 #include <iostream>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <string>
6
7 int main(int argc, char* argv[])
8 {
9     const std::string DEFAULT = "def";
10    if (argc < 5)
11    {
12        std::cerr << "error!" << std::endl;
13        return -1;
14    }
15    std::string out = argv[2];
16    std::string in = argv[1];
17    std::string error = argv[3];
18    std::string path = argv[4];
19    FILE* out_file = NULL;
20    FILE* in_file = NULL;
21    FILE* error_file = NULL;
22    FILE* console = fdopen(dup(STDOUT_FILENO), "w");
23    size_t arguments = argc - 4;
24    char** argument;
25    argument = new char* [arguments];
26    for (int i = 0; i < arguments; i++)
27    {
28        argument[i] = argv[i + 4];
29    }
```

```
#include <cstdio>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <string>
```

```

int main(int argc, char* argv[])
{
    const std::string DEFAULT = "def";
    if (argc < 5)
    {
        std::cerr << "error!" << std::endl;
        return -1;
    }
    std::string out = argv[2];
    std::string in = argv[1];
    std::string error = argv[3];
    std::string path = argv[4];
    FILE* out_file = NULL;
    FILE* in_file = NULL;
    FILE* error_file = NULL;
    FILE* console = fdopen(dup(STDOUT_FILENO), "w");
    size_t arguments = argc - 4;
    char** argument;
    argument = new char* [arguments];
    for (int i = 0; i < arguments; i++)
    {
        argument[i] = argv[i + 4];
    }
    if (out != DEFAULT)
    {
        out_file = freopen(out.c_str(), "w", stdout);
    }
    if (in != DEFAULT)
    {
        in_file = freopen(in.c_str(), "rt", stdin);
    }
    if (error != DEFAULT)
    {
        error_file = freopen(error.c_str(), "w", stderr);
    }
    int val = 0;
    pid_t p = fork();
    if (p == -1)
    {
        std::cerr << "does not to fork" << std::endl;
    }
    else if (p > 0)
    {
        int condition;
        waitpid(p, &condition, 0);
    }
    else
    {
        val = execv(path.c_str(), argument);
    }
    fprintf(console, "Completion code: %d\n", val);
    if (out_file != NULL)
    {
        fprintf(out_file, "Completion code: %d\n", val);
        fclose(stdout);
    }
    if (error_file != NULL)
    {
        fprintf(error_file, "Completion code: %d\n", val);
        fclose(stderr);
    }
    return 0;
}

```

Примеры вызова скрипта:

Создадим скрипт test.cpp и будем вызывать его в качестве приложения.

```
home > elshan > aaa > test.cpp > main(int, char * [])
1  # include <cstdio>
2  # include <string>
3  # include <iostream>
4
5  int main(int argc, char* argv[])
6  {
7      std::string text;
8      std::getline(std::cin, text);
9      std::cout << "This is output. String: " << text << " " << argv[1] << std::endl;
10     std::cerr << "This is error! " << std::endl;
11     return 0;
12 }
```

```
[elshan@spaton aaa]$ g++ -Wall -o main test.cpp
[elshan@spaton aaa]$ ./main in.txt out.txt err.txt ./test
Hello
```

```
home > elshan > aaa > in.txt
1  Hello
```

Запустим программу переопределив потоки *stdout* и *stderr*

```
home > elshan > aaa > out.txt
1  This is output. String: Hello
2  Application completion code: 0
```

```
home > elshan > aaa > err.txt
1  This is error!
2  Application completion code: 0
```

```
[elshan@spaton aaa]$ ./main def out.txt err.txt ./test
Hello
```

```
home > elshan > aaa > out.txt
1  This is output. String: Hello
2  Application completion code: 0
```



В этом случае ввод происходит из консоли, а потоки *stdout* и *stderr* переопределены

```
home > elshan > aaa > ≡ err.txt
1   This is error!
2   Application completion code: 0
```

Переопределим один поток *stderr*:

`./main def def err.txt ./test`

```
home > elshan > aaa > ≡ err.txt
1   This is error!
2   Application completion code: 0
```

Код совершения программ дублируется в открытые файлы и консоль

## ЗАКЛЮЧЕНИЕ

**Вывод:** в ходе выполнения лабораторной работы, был изучен стандарт POSIX, а также реализованы два приложения для получения потоков запускаемого приложения (stderr/stdout) на POSIX-совместимый shell-скрипте и приложение на C++11.