



**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 № 2
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. СИСТЕМНЫЕ КОМАНДЫ.....	4
1.1. Man	4
1.2. Chmod.....	5
1.3. Chown.....	6
1.4. Fstab.....	7
1.5. Proc	8
1.6. Signal	9
1.7. Sh	9
2. ПРОГРАММНЫЕ КОМАНДЫ.....	10
2.1. Stdio	10
2.2. Stdin/stdout/stderr	11
2.3. Pipe.....	12
2.4. Dup	13
2.5. Fork	14
2.6. Exec.....	15
ЗАКЛЮЧЕНИЕ	18

ВСТУПЛЕНИЕ

Цель работы: ознакомиться с man-pages и её идеологиями. Привести примеры на C/C++ с использованием функций: `stdio`, `stdin/stdout/stderr`, `pipe`, `dup`, `fork`, `exec`.

Man-pages (от слова *manual* — *руководство*) — это целая библиотека в системе Linux, содержащая руководства по командам, утилитам, программированию и другим областям системы и не только.

1. СИСТЕМНЫЕ КОМАНДЫ

1.1. Man

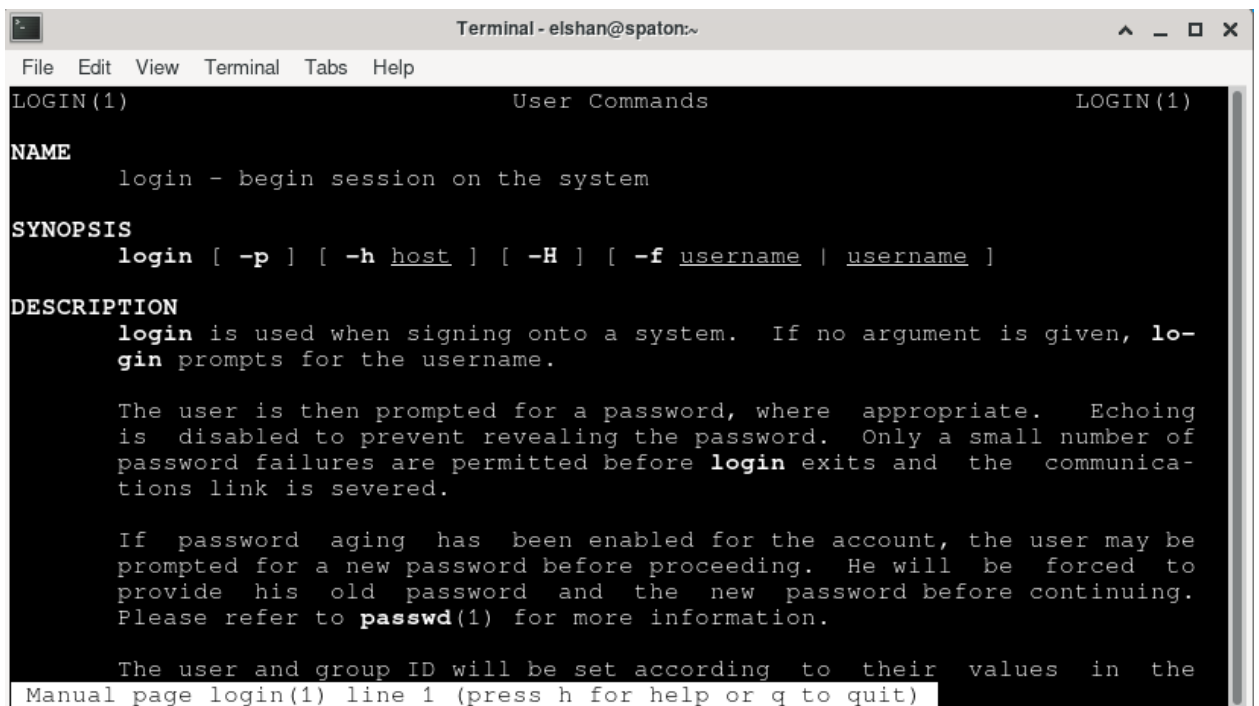
Чтобы получить руководство по использованию какой-либо команды нужно выполнить команду: `man <имя_страницы>`

На `man`-страницы принято ссылаться по имени, с указанием номера категории в скобках. Часто существуют сразу несколько `man`-страниц с одинаковыми именами, но в разных категориях, например `man(1)` и `man(7)`. В таком случае, команде `man` необходимо передать номер конкретной категории перед именем `man`-страницы.

Если не указать раздел при выполнении команды `man`, то сначала будет выполнен поиск руководства в первом разделе, если его там нет, то во втором и так далее.

Например, откроем руководство по команде `login`. В данном случае будет использоваться первый раздел.

man login



```
Terminal - elshan@spaton:~
File Edit View Terminal Tabs Help
LOGIN(1) User Commands LOGIN(1)
NAME
    login - begin session on the system
SYNOPSIS
    login [ -p ] [ -h host ] [ -H ] [ -f username | username ]
DESCRIPTION
    login is used when signing onto a system.  If no argument is given, login prompts for the username.

    The user is then prompted for a password, where appropriate.  Echoing is disabled to prevent revealing the password.  Only a small number of password failures are permitted before login exits and the communications link is severed.

    If password aging has been enabled for the account, the user may be prompted for a new password before proceeding.  He will be forced to provide his old password and the new password before continuing.  Please refer to passwd(1) for more information.

    The user and group ID will be set according to their values in the
Manual page login(1) line 1 (press h for help or q to quit)
```

1.2. Chmod

chmod (от англ. *change mode*) — программа для изменения прав доступа к файлам и каталогам.

```
chmod [options] mode[,mode] file1 [file2 ...]
```

Опции:

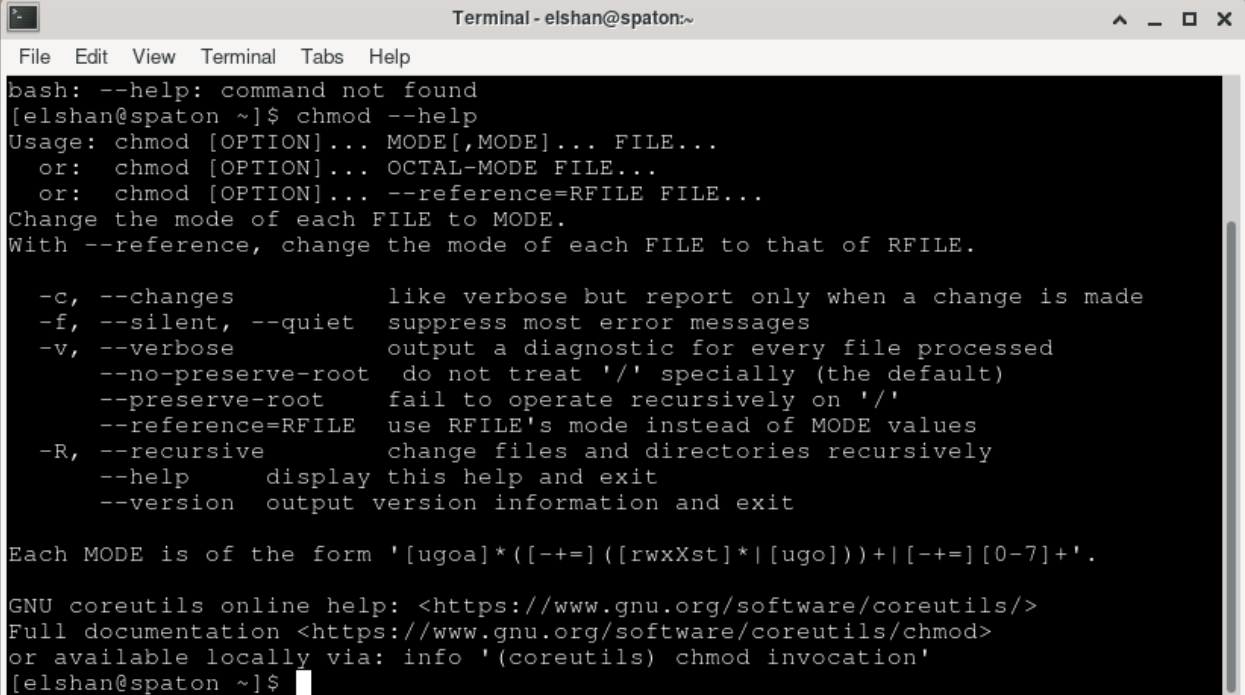
- -R рекурсивное изменение прав доступа для каталогов и их содержимого
- -f не выдавать сообщения об ошибке для файлов, чьи права не могут быть изменены.
- -v подробно описывать действие или отсутствие действия для каждого *файла*.

chmod никогда не изменяет права на символичные ссылки. Однако для каждой символической ссылки, заданной в командной строке, **chmod** изменяет права доступа связанного с ней файла. При этом **chmod** игнорирует символичные ссылки, встречающиеся во время рекурсивной обработки каталогов.

Аргумент команды **chmod**, задающий разрешения, может быть записан в двух форматах: в числовом и в символьном.

Права записываются одной строкой сразу для трёх типов пользователей:

- владельца файла (u);
- других пользователей, входящих в группу владельца (g);
- всех прочих пользователей (o);
-



```
Terminal - elshan@spaton:~
File Edit View Terminal Tabs Help
bash: --help: command not found
[elshan@spaton ~]$ chmod --help
Usage: chmod [OPTION]... MODE[,MODE]... FILE...
       or:  chmod [OPTION]... OCTAL-MODE FILE...
       or:  chmod [OPTION]... --reference=RFILE FILE...
Change the mode of each FILE to MODE.
With --reference, change the mode of each FILE to that of RFILE.

-c, --changes      like verbose but report only when a change is made
-f, --silent, --quiet suppress most error messages
-v, --verbose      output a diagnostic for every file processed
--no-preserve-root do not treat '/' specially (the default)
--preserve-root    fail to operate recursively on '/'
--reference=RFILE  use RFILE's mode instead of MODE values
-R, --recursive    change files and directories recursively
--help            display this help and exit
--version         output version information and exit

Each MODE is of the form '[ugoa]*([+|=]([rwxXst]*|[ugo]))+|[-|=][0-7]+'

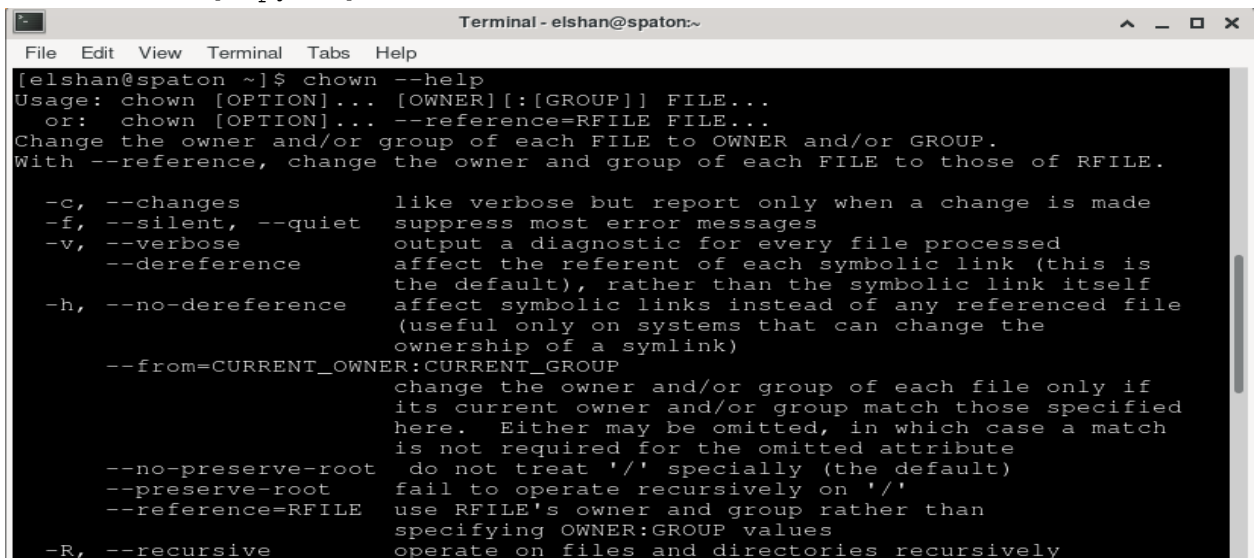
GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Full documentation <https://www.gnu.org/software/coreutils/chmod>
or available locally via: info '(coreutils) chmod invocation'
[elshan@spaton ~]$
```

1.3. Chown

chown (от англ. *change owner*) — UNIX-утилита, изменяющая владельца и/или группу для указанных файлов. В качестве имени владельца/группы берётся первый аргумент, не являющийся опцией. Если задано только имя пользователя (или числовой идентификатор пользователя), то данный пользователь становится владельцем каждого из указанных файлов, а группа этих файлов не изменяется. Если за именем пользователя через двоеточие следует имя группы (или числовой идентификатор группы), без пробелов между ними, то изменяется также и группа файла. При стандартной настройке сервера команда вызывает сброс накопленных кэшей (событие touch).

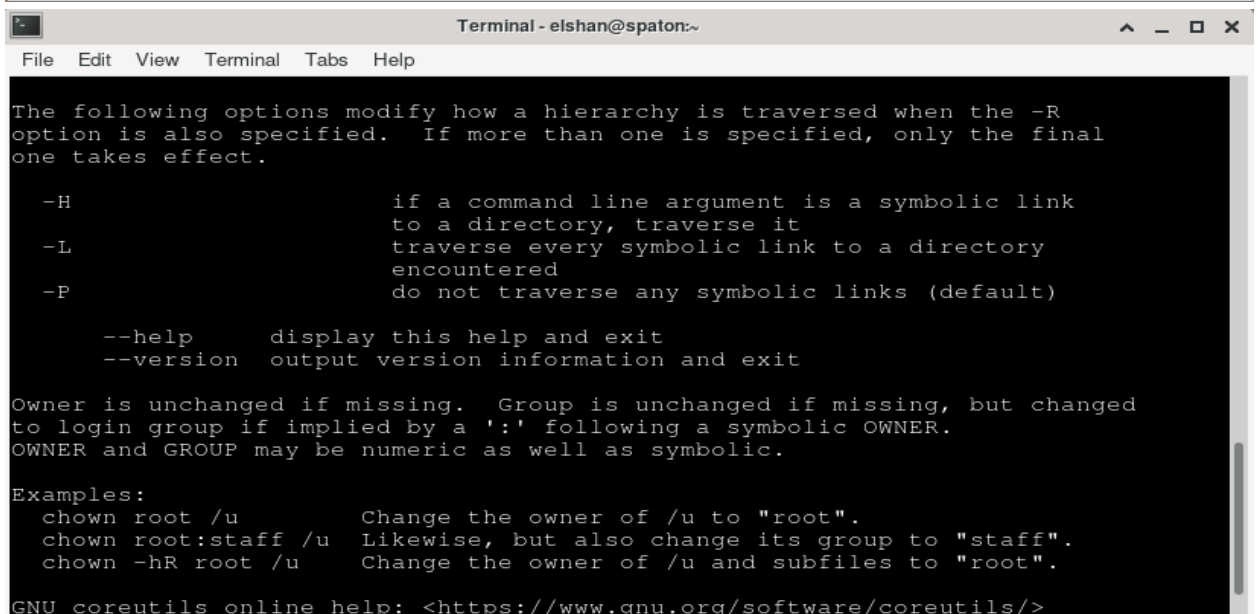
Использование:

```
chown      [-cfhvR]      [--dereference]      [--reference=rfile]
пользователь [:группа]
```



```
Terminal - elshan@spaton:~
File Edit View Terminal Tabs Help
[elshan@spaton ~]$ chown --help
Usage: chown [OPTION]... [OWNER]][:[GROUP]] FILE...
       or: chown [OPTION]... --reference=RFILE FILE...
Change the owner and/or group of each FILE to OWNER and/or GROUP.
With --reference, change the owner and group of each FILE to those of RFILE.

  -c, --changes           like verbose but report only when a change is made
  -f, --silent, --quiet  suppress most error messages
  -v, --verbose           output a diagnostic for every file processed
                        --dereference  affect the referent of each symbolic link (this is
                        the default), rather than the symbolic link itself
  -h, --no-dereference   affect symbolic links instead of any referenced file
                        (useful only on systems that can change the
                        ownership of a symlink)
                        --from=CURRENT_OWNER:CURRENT_GROUP
                        change the owner and/or group of each file only if
                        its current owner and/or group match those specified
                        here. Either may be omitted, in which case a match
                        is not required for the omitted attribute
                        --no-preserve-root  do not treat '/' specially (the default)
                        --preserve-root   fail to operate recursively on '/'
                        --reference=RFILE  use RFILE's owner and group rather than
                        specifying OWNER:GROUP values
  -R, --recursive        operate on files and directories recursively
```



```
Terminal - elshan@spaton:~
File Edit View Terminal Tabs Help

The following options modify how a hierarchy is traversed when the -R
option is also specified. If more than one is specified, only the final
one takes effect.

  -H           if a command line argument is a symbolic link
               to a directory, traverse it
  -L           traverse every symbolic link to a directory
               encountered
  -P           do not traverse any symbolic links (default)

  --help      display this help and exit
  --version   output version information and exit

Owner is unchanged if missing. Group is unchanged if missing, but changed
to login group if implied by a ':' following a symbolic OWNER.
OWNER and GROUP may be numeric as well as symbolic.

Examples:
  chown root /u           Change the owner of /u to "root".
  chown root:staff /u      Likewise, but also change its group to "staff".
  chown -hR root /u       Change the owner of /u and subfiles to "root".

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
```

Примеры:

- Поменять владельца для strace.log в 'rob' и идентификатор группы в 'developers'.

```
# chown rob:developers strace.log
```

- Поменять идентификатор группы на newgroup для /home

```
# chown :newgroup /home
```

Команда **chown** позволяет только менять владельца и группу, если необходимо более подробно настроить права для владельца, группы и всех остальных, понадобится команда **chmod**.

1.4. Fstab

Файл **fstab** - это текстовый файл, который содержит информацию о различных файловых системах и устройствах хранения информации в вашем компьютере. Это всего лишь один файл, определяющий, как диск и/или раздел будут использоваться и как будут встроены в остальную систему. Полный путь к файлу - **/etc/fstab**. Этот файл можно открыть в любом текстовом редакторе, но редактировать его возможно только от имени суперпользователя, т.к. файл является важной, неотъемлемой частью системы, без него система не загрузится. Файл **FSTab** представляет собой обычный текстовый файл со строками по определённому формату. Строки, начинающиеся со знака '#' не обрабатываются и могут содержать комментарии.

Файл **FSTab** содержит строки вида:

```
# <file system> <dir> <type> <options> <dump> <pass>
```

```
Terminal - elshan@spaton:~
File Edit View Terminal Tabs Help
GNU nano 4.9.3 /etc/fstab
# Static information about the filesystems.
# See fstab(5) for details.

# <file system> <dir> <type> <options> <dump> <pass>
# /dev/sda2
UUID=95aeaad7-6214-4021-956b-9ebba7abb21e / ext4
# /dev/sda1
UUID=8C8D-64E0 /boot vfat rw,relatime,fmask=0022,
# /dev/sda3
UUID=b183f0c2-50c5-48c8-a74c-15baf2345575 /var ext4
# /dev/sda4
UUID=48d86c83-cf37-4448-9959-2e6c19975f96 /opt ext4
# /dev/sda5
UUID=b45e72e2-6e63-41d6-8386-7b76bc38d556 /tmp ext4
# /dev/sda6

[ Read 23 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell ^_ Go To Line
```

1.5. Proc

Программы пространства пользователя в Linux не могут обращаться к ядру системы напрямую. Но для получения информации от ядра были созданы несколько специальных директорий с помощью которых любая программа или пользователь могут получить данные о состоянии компьютера и ядра. Это файловая система proc и sys.

Из этих папок можно получить любую информацию о вашей системе. Например, сколько памяти подкачки сейчас используется, насколько велик размер кеша процессора, какие модули ядра загружены, сколько дисков или разделов доступно и т.д. Все это можно получить в обычном текстовом виде из папки proc linux.

Все поддиректории, файлы и хранящаяся в них информация генерируется ядром на лету, как только вы ее запрашиваете.

С помощью такой системы разработчики придерживаются главной концепции Unix - все есть файл. Все файлы доступны для редактирования любым редактором, и все они в простом текстовом формате, но для того чтобы проанализировать весь каталог вам понадобятся права суперпользователя. Почти все файлы доступны только для чтения, с них мы можем только получать информацию. Но есть и доступные для записи, в частности это /proc/sys с помощью которого вы можете настраивать различные параметры ядра.

1.6. Signal

Signal — асинхронное уведомление процесса о каком-либо событии, один из основных способов взаимодействия между процессами. Когда сигнал послан процессу, операционная система прерывает выполнение процесса, при этом, если процесс установил собственный *обработчик сигнала*, операционная система запускает этот обработчик, передав ему информацию о сигнале, если процесс не установил обработчик, то выполняется обработчик по умолчанию.

Сигналы посылаются:

- из терминала, нажатием специальных клавиш или комбинаций (например, нажатие Ctrl-C генерирует SIGINT, Ctrl-\ SIGQUIT, а Ctrl-Z SIGTSTP);
- ядром системы;
- одним процессом другому (или самому себе), с помощью системного вызова kill(), в том числе:

1.7. Sh

Sh - Вызывает командный интерпретатор shell.

Shell - это стандартный командный язык программирования, который выполняет чтение команд с терминала или из файла. В такой файл мы можем вписать все команды, которые выполняем в терминале, то есть, которые исполняются командной оболочкой нашей системы.

Чтобы запустить скрипт, надо зайти в каталог, где расположен скрипт, набрать название интерпретатора sh и первым параметром указать файл.

```
sh hello.sh
```

Чтобы каждый раз не указывать интерпретатор в терминале, можно сделать скрипт исполняемым (для этого используется команда chmod +x hello.sh) и необходимо указать интерпретатор внутри файла скрипта (в первой строке после #! прописывается путь к bash-интерпретатору).

Содержимое скрипта hello.sh:

```
#!/bin/bash  
  
echo "Hi"
```

2. ПРОГРАММНЫЕ КОМАНДЫ

2.1. Stdio

stdio - стандартные библиотечные функции ввода/вывода (I/O).

```
#include <stdio.h>
FILE *stdin, *stdout, *stderr;
```

Макросы и функции предоставляют пользователю эффективные средства буферизованного ввода/вывода. Макросы `getc(3S)` и `putc(3S)` служат для быстрого ввода/вывода символов. Макросы `getchar` и `putchar` и функции более высокого уровня `fgetc`, `fgets`, `fprintf`, `fputc`, `fputs`, `fread`, `fscanf`, `fwrite`, `gets`, `getw`, `printf`, `puts`, `putw` и `scanf` ведут себя так, как если бы они использовали `getc` и `putc`. Обращения к макросам и функциям из данного пакета можно чередовать произвольным образом.

Файл и ассоциированный с ним механизм буферизации называются потоком. Поток описывается как указатель на переменную типа `FILE`. Функция `fopen(3S)` создает описатель потока и возвращает указатель на него. Этот указатель идентифицирует поток во всех последующих операциях. Обычно имеются три открытых потока с постоянными указателями, они описаны во включаемом файле `<stdio.h>` и связаны со стандартными открытыми файлами:

```
stdin    стандартный ввод
stdout   стандартный вывод
stderr   стандартный протокол
```

Константа `NULL (0)` обозначает пустой указатель.

Целая константа `EOF (-1)` возвращается по достижении конца файла или в случае ошибки большинством из целочисленных функций, работающих с потоками (для получения детальной информации см. описания отдельных функций).

Целая константа `BUFSIZ` специфицирует размер буферов, используемых в конкретной реализации.

Любая программа, использующая данный пакет ввода/вывода, должна включать файл соответствующих макроопределений следующим образом:

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

Функции и константы, описанные в файле `<stdio.h>` и не требуют дальнейшего описания. Константы и следующие "функции" реализованы как макросы (переопределение этих имен опасно): `getc`, `getchar`, `putc`, `putchar`, `ferror`, `feof`, `clearerr` и `fileno`.

```
main.cpp
home > elshan > aaa > main.cpp > ...
1  #include <stdio.h>
2
3
4  int main()
5  {
6      puts("pages");
7      printf("Enter name");
8      char name[15];
9      scanf("%s", name);
10     printf("Hello, %s\n", name);
11     return 0;
12 }
```

2.2. Stdin/stdout/stderr

Stdin/stdout/stderr - стандартные потоки I/O, имеющие номер (дескриптор), зарезервированный для выполнения некоторых «стандартных» функций.

Поток номер 0 (**stdin**) зарезервирован для чтения команд пользователя или входных данных. При интерактивном запуске программы по умолчанию нацелен на чтение с устройства текстового интерфейса пользователя (клавиатуры). Командная оболочка UNIX (и оболочки других систем) позволяют изменять цель этого потока с помощью символа «<». Системные программы (демоны и т. п.), как правило, не пользуются этим потоком.

Поток номер 1 (**stdout**) зарезервирован для вывода данных, как правило (хотя и не обязательно) текстовых. При интерактивном запуске программы по умолчанию нацелен на запись на устройство отображения (монитор). Командная оболочка UNIX (и оболочки других систем) позволяют перенаправить этот поток с помощью символа «>». Средства для выполнения программ в фоновом режиме (например, `nohup`) обычно переназначают этот поток в файл.

Поток номер 2 (**stderr**) зарезервирован для вывода диагностических и отладочных сообщений в текстовом виде. Чаще всего цель этого потока совпадает с `stdout`, однако, в отличие от него, цель потока `stderr` не меняется при «>» и создании конвейеров («|»). То есть, отладочные сообщения процесса, вывод которого перенаправлен, всё равно попадут пользователю. Командная оболочка UNIX позволяет изменять цель этого потока с помощью конструкции «2>». Например, для подавления вывода этого потока нередко пишется «2>/dev/nll».

```
main.cpp
home > elshan > aaa > main.cpp > main()
1  #include <stdio.h>
2
3
4  int main()
5  {
6      puts("pages");
7      fprintf("Enter name");
8      char name[15];
9      fscanf(stdin,"%s",name);
10     fprintf(stderr,"Error name\n");
11     return 0;
12 }
```

2.3. Pipe

Pipe (конвейер) – это однонаправленный канал межпроцессного взаимодействия. Конвейеры чаще всего используются в shell-скриптах для связи нескольких команд путем перенаправления вывода одной команды (stdout) на вход (stdin) последующей, используя символ конвейера '|':

```
cmd1 | cmd2 | .... | cmdN
```

Конвейер обеспечивает асинхронное выполнение команд с использованием буферизации ввода/вывода. Таким образом все команды в конвейере работают параллельно, каждая в своем процессе.

У функции **pipe** следующее объявление:

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

```
main.cpp
home > elshan > aaa > main.cpp > main()
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7      int fildes[2];
8      const int SIZE = 50;
9      char buf[SIZE];
10     ssize_t bytes;
11     int status = pipe(fildes);
12     if(status == -1)
13     {
14         int a = 0;
15     }
16     return 0;
17 }
```

2.4. Dup

Dup - дублирование дескриптора открытого файла.

```
int dup (fildes)
int fildes
```

Аргумент *fildes* - это дескриптор файла, полученный после выполнения системных вызовов *creat*, *open*, *dup*, *fcntl* и *pipe*. Системный вызов *dup* возвращает новый дескриптор файла, имеющий следующие общие свойства с исходным дескриптором:

1. Тот же открытый файл (или канал).
2. Тот же указатель текущей позиции в файле (то есть оба дескриптора разделяют один и тот же указатель).
3. Тот же режим доступа (чтение, запись или чтение/запись).

Новый дескриптор создается таким, чтобы после выполнения системных вызовов *exec(2)* файл оставался открытым.

Возвращается наименьший из доступных дескрипторов.

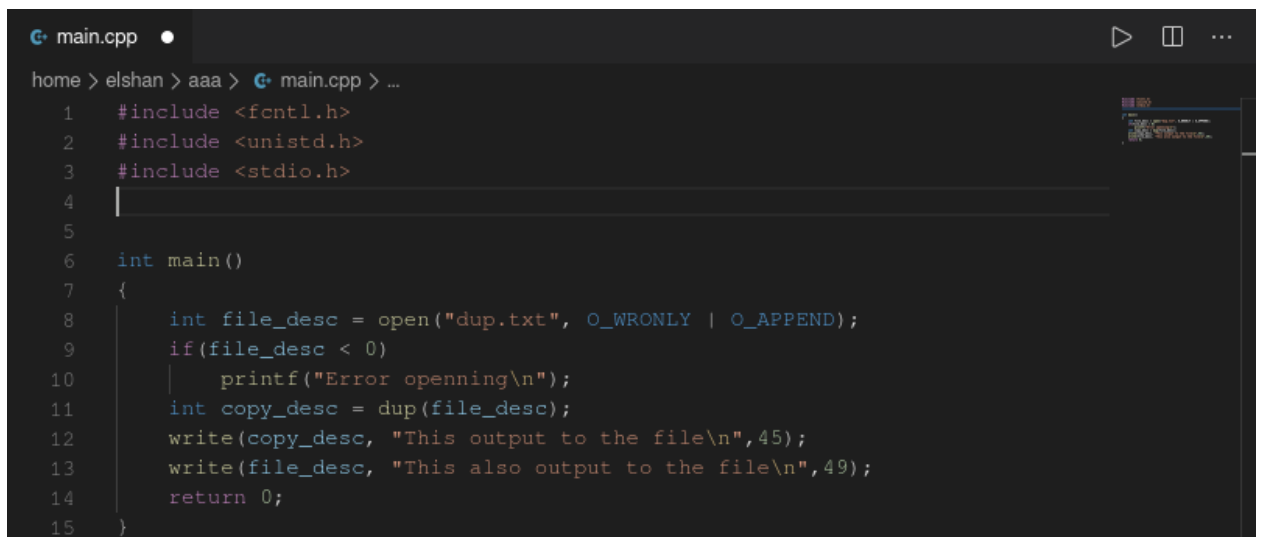
Системный вызов *dup* завершается неудачей, если выполнено хотя бы одно из следующих условий:

[EBADF] - Аргумент *fildes* не является корректным дескриптором открытого файла.

[EINTR] - Во время выполнения системного вызова перехвачен сигнал.

[EMFILE] - Превышается максимально допустимое количество файлов, открытых одновременно в одном процессе.

[ENOLINK] - Аргумент *fildes* указывает на удаленный компьютер, связи с которым в данный момент нет.



```
main.cpp
home > elshan > aaa > main.cpp > ...
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5
6  int main()
7  {
8      int file_desc = open("dup.txt", O_WRONLY | O_APPEND);
9      if(file_desc < 0)
10         printf("Error opening\n");
11     int copy_desc = dup(file_desc);
12     write(copy_desc, "This output to the file\n",45);
13     write(file_desc, "This also output to the file\n",49);
14     return 0;
15 }
```

2.5. Fork

fork - создает дочерний процесс.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

`fork` создает процесс-потомок, который отличается от родительского только значениями PID (идентификатор процесса) и PPID (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0. Блокировки файлов и сигналы, ожидающие обработки, не наследуются.

Под Linux `fork` реализован с помощью "копирования страниц при записи" (copy-on-write, COW), поэтому расходы на `fork` сводятся к копированию таблицы страниц родителя и созданию уникальной структуры, описывающей задачу.

При успешном завершении родителю возвращается PID процесса потомка, а процессу-потомку возвращается 0. При неудаче родительскому процессу возвращается -1, процесс-потомок не создается, а значение `errno` устанавливается должным образом.

```
main.cpp x
home > elshan > aaa > main.cpp > ...
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <errno.h>
5  #include <sys/wait.h>
6  #include <sys/types.h>
7
8  int main()
9  {
10     int pid = fork();
11     switch(pid)
12     {
13     case -1:
14         perror("fork");
15         return -1;
16     case 0:
17         printf("my pid = %i, returned pid = %i\n", getpid(), pid);
18         break;
19     default:
20         printf("my pid = %i, returned pid = %i\n", getpid(), pid);
21     }
22     return 0;
23 }
```

2.6. Exec

exec: execl, execv, execl, exesve, execlp, exesvr - выполнение файла.

```
int execl (path, arg0, arg1, ..., argn, (char*) 0)
```

```
char *path, *arg0, *arg1, ..., *argn;
```

```
int execv (path, argv)
```

```
char *path, *argv [];
```

```
int execl (path, arg0, arg1, ..., argn, (char*) 0, envp)
```

```
char *path, *arg0, *arg1, ..., *argn, *envp [];
```

```
int exesve (path, argv, envp)
```

```
char *path, *argv [], *envp [];
```

```
int execlp (file, arg0, arg1, ..., argn, (char*) 0)
```

```
char *file, *arg0, *arg1, ..., *argn;
```

```
int exesvp (file, argv)
```

```
char *file, *argv [];
```

Все формы системного вызова `exec` превращают вызвавший процесс в новый процесс, который строится из обычного выполняемого файла, называемого в дальнейшем новым выполняемым файлом. Выполняемый файл состоит из заголовка, сегмента команд (`.text`) и данных. Данные состоят из инициализированной (`.data`) и неинициализированной (`.bss`) частей. Если системный вызов `exec` закончился успешно, то он не может вернуть управление, так как вызвавший процесс уже заменен новым процессом.

При запуске С-программы ее вызывают следующим образом:

```
main (argc, argv, envp)

int argc;

char **argv, **envp;
```

где `argc` равен количеству аргументов, `argv` - массив указателей собственно на аргументы и `envp` - массив указателей на цепочки символов, образующие окружение. Принято соглашение, по которому значение `argc` не меньше 1, а первый элемент массива `argv` указывает на цепочку символов, содержащую имя нового выполняемого файла.

Аргументам системных вызовов группы `exec` приписан следующий смысл.

Аргумент `path` указывает на маршрутное имя нового выполняемого файла.

Как и `path`, аргумент `file` указывает новый выполняемый файл, но маршрут этого файла определяется в результате просмотра каталогов, переданных через переменную окружения `PATH`. Окружение поддерживается `shell`'ом.

Аргументы `arg0`, `arg1`, ..., `argn` - это указатели на цепочки символов, ограниченные нулевыми байтами. Эти цепочки образуют доступный новому процессу список аргументов.

Массив `argv` содержит указатели на цепочки символов, ограниченные нулевыми байтами.

Массив `envp` содержит указатели на цепочки символов, ограниченные нулевыми байтами. Эти цепочки образуют окружение нового процесса. За последним занятым элементом массива `envp` должен следовать пустой указатель.

Перед началом выполнения любой программы во внешнюю переменную `environ`, описание которой выглядит как

```
extern char **environ;
```


помещается адрес массива указателей на цепочки символов, образующие окружение процесса. С помощью этой переменной (как и с помощью аргумента `env` функции `main`) в новом процессе всегда можно получить доступ к окружению, независимо от использовавшегося варианта системного вызова `exec`. Разница лишь в том, что в случае вызовов `execle` и `execve` окружение нового процесса задается явно, а в остальных случаях наследуется у вызвавшего процесса.

Использование `execl()`

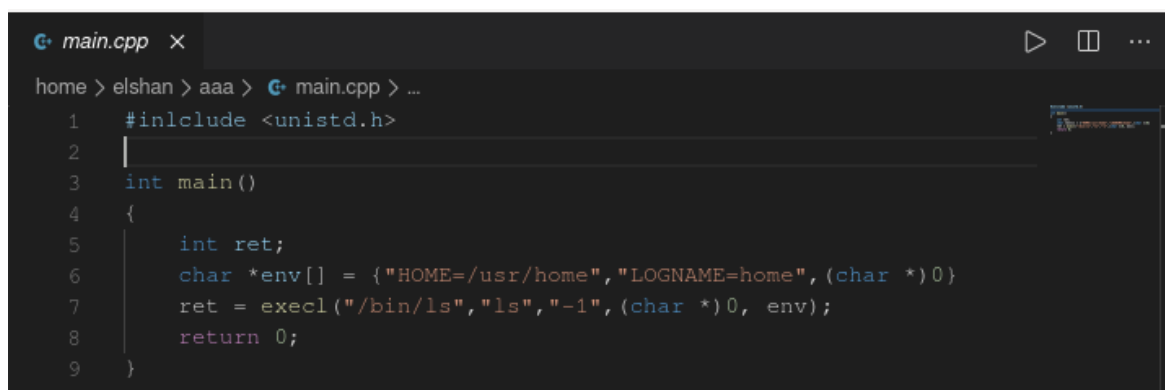
В следующем примере выполняется команда `ls` с указанием пути к исполняемому файлу (`/bin/ls`) и использованием аргументов, переданных непосредственно в команду, для вывода в один столбец.



```
main.cpp
home > elshan > aaa > main.cpp > main()
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  int main()
5  {
6      int ret;
7      ret = execl("/bin/ls", "ls", "-l", (char *) 0);
8      return 0;
9  }
```

Использование `execle()`

Следующий пример похож на Использование `execl()`. Кроме того, он определяет среду для нового образа процесса с помощью аргумента `env`.



```
main.cpp x
home > elshan > aaa > main.cpp > ...
1  #include <unistd.h>
2
3  int main()
4  {
5      int ret;
6      char *env[] = {"HOME=/usr/home", "LOGNAME=home", (char *) 0};
7      ret = execle("/bin/ls", "ls", "-l", (char *) 0, env);
8      return 0;
9  }
```

ЗАКЛЮЧЕНИЕ

Вывод: в ходе выполнения лабораторной работы, было произведено знакомство с man-pages и её идеологиями, а также были преведены примеры на C/C++ с использованием функций: `stdio`, `stdin/stdout/stderr`, `pipe`, `dup`, `fork`, `exec`.