# **Потоки**

В каждом языке есть свой способ напечатать результат на экран.

Javascript:

console.log('hello!');

PHP:

**<?php**

**echo** 'hello!';

Python:

**print**('hello!')

Java:

**System.**out**.**print**(**"hello!"**);**

Ruby:

puts 'hello!'

Несмотря на разнообразие языков и способов печати, с точки зрения операционной системы все программы работают абсолютно идентично. При старте любой программы операционная система связывает с ней три потока:

* **STDIN** (Standard Input)
* **STDOUT** (Standard Output)
* **STDERR** (Standard Error)

Для языка программирования эти потоки выглядят как файлы, и взаимодействие с ними происходит как с файлами.

Начнем с потока STDOUT. Он отвечает за вывод на экран. Во время каждой печати на экран в программе на любом языке происходят следующие два шага:

* Функция печати записывает данные в поток STDOUT с помощью функции write
* Операционная система решает, куда вывести результат. По умолчанию вывод происходит на экран терминала

Чтобы хорошо понять эту тему, нужно подробно изучить устройство операционных систем, в частности подсистемы, отвечающей за процессы и файловую систему. Но мы попробуем кратко рассказать самое важное для понимания.

В двух словах, языки программирования не взаимодействуют с монитором и железом в целом. За взаимодействие с железом целиком и полностью отвечает операционная система. Программы могут только лишь попросить операционную систему выполнить ту или иную задачу.

При таком разделении реализация языков программирования сильно упрощается. Достаточно знать про существование STDOUT и уметь писать в этот поток, а дальше все сделает операционная система.

Именно благодаря такому разделению и потокам, можно написать программу на одном компьютере и без проблем запустить ее на другом компьютере с другой конфигурацией и монитором.

Самое удивительное начинается дальше. Операционные системы позволяют подменять потоки при старте системы, что открывает интересные возможности.

Например, вывод любой команды в bash можно не выводить на экран, а записать в файл:

ls -la **>** output

Запустив эту команду, вы не увидите на экране ничего нового. Зато в текущей директории появится файл *output*:

cat output

total 44

drwxr-xr-x 5 kirill.m kirill.m 4096 Aug 29 09:39 .

drwxr-xr-x 8 root root 4096 Apr 26 10:38 ..

-rw------- 1 kirill.m kirill.m 1822 Aug 29 08:45 .bash\_history

-rw-r--r-- 1 kirill.m kirill.m 220 Aug 31 2015 .bash\_logout

-rw-r--r-- 1 kirill.m kirill.m 3771 Aug 31 2015 .bashrc

drwx------ 2 kirill.m kirill.m 4096 Mar 30 18:10 .cache

-rw------- 1 kirill.m kirill.m 55 Aug 28 18:49 .lesshst

drwxrwxr-x 2 kirill.m kirill.m 4096 Aug 29 08:35 .nano

-rw-rw-r-- 1 kirill.m kirill.m 0 Aug 29 09:39 output

-rw-r--r-- 1 kirill.m kirill.m 655 May 16 2017 .profile

drwx------ 2 kirill.m kirill.m 4096 Jan 22 2018 .ssh

-rw------- 1 kirill.m kirill.m 513 Aug 29 08:06 .viminfo

Операция, которую мы сделали выше, называется **перенаправление потоков**.

Символ > означает, что нужно взять вывод из команды слева и отправить его в файл, указанный справа. Этот символ > всегда **перезаписывает** файл.

Такое перенаправление работает с абсолютно любой командой, которая выводит результаты своей работы в консоль:

grep alias .bash\_profile **>** result

cat result

alias fixssh**=**'eval $(tmux showenv -s SSH\_AUTH\_SOCK)'

Если нужно не перезаписывать, а **добавлять**, то используйте >>.

Для экспериментов с выводом удобно использовать встроенную команду echo. Она принимает на вход строчку и выдает ее в STDOUT, который уже можно перенаправлять:

*# > Заменяет содержимое файла*

echo 'hi' **>** result *# Вывод команды echo направляем в файл result*

cat result *# С помощью команды cat читаем файл result*

hi *# Содержимое файла result в окне терминала*

echo 'hello' **>** result *# Вывод команды echo заменяет содержимое файла result*

cat result *# С помощью команды cat читаем файл result*

hello *# Содержимое файла result в окне терминала# >> Добавляет содержимое в конец файла*

echo 'hello' **>>** result *# Вывод команды echo добавляем в конец файла result*

cat result *# С помощью команды cat читаем файл result*

*# Так выглядит содержимое файл result в окне терминала:*

hello

hello

Кроме стандартного вывода, с каждым процессом ассоциируются два дополнительных потока:

* STDERR — вывод ошибок
* STDIN — стандартный ввод

STDIN работает в обратную сторону: через него программа может получать данные на вход.

В \*nix-системах встроена утилита wc (сокращение от *word count*). Она умеет считать количество слов, строк или символов в файле. Когда мы говорим о файле, то в \*nix-системах это почти всегда означает, что данные можно передать и в стандартный поток ввода:

*# Флаг l (буква l, а не цифра 1) указывает, что надо считать количество строк*

wc -l < result *# Содержимое файла result отправляем в стандартный ввод команды wc*

2

Выглядит довольно логично — стрелка меняет свое направление в другую сторону и содержимое файла отправляется в поток STDIN запускаемой программы wc.

Теперь объединим перенаправление ввода и вывода:

wc -l < result **>** output *# Содержимое файла result отправляем в стандартный ввод команды wc, а вывод направляем в файл output*

cat output *# С помощью команды cat читаем файл output*

2

Поток STDERR позволяет отделить нормальный вывод программы от возникающих ошибок.

Будьте осторожны, потому что перенаправление вывода в файл перенаправляет только STDOUT. Убедиться в этом очень просто.

Если попробовать отобразить содержимое несуществующей директории, то команда ls выдаст ошибку:

ls lala *# Команда ls выводит список файлов и директорий, которые содержит директория lala*

ls: cannot access 'lala': No such file or directory *# Получаем сообщение об ошибке в окне терминала*

Теперь попробуем перенаправить вывод в файл *output*:

ls lala **>** output *# Направляем вывод команды ls в файл output*

ls: cannot access 'lala': No such file or directory *# Вывод на экран*

Перенаправление есть, но сообщение вывелось на экран. Это произошло именно по той причине, что STDERR остался привязан к экрану, а внутри файла *output* — пустота. Решить эту задачу можно несколькими способами.

### **Перенаправление STDERR в STDOUT**

В unix-системах за каждым потоком закреплен определенный номер, который называется файловым дескриптором. С помощью него можно получить доступ к потокам ввода и вывода.

Существуют следующие стандартные потоки ввода и вывода:

* STDIN — 0
* STDOUT — 1
* STDERR — 2

Первый способ — перенаправить STDERR в STDOUT или отправить оба потока в файл.

Часто стандартный поток ошибок объединяют со стандартным потоком вывода, чтобы можно было обрабатывать ошибки и результат выполнения вместе:

*# Сначала STDOUT перенаправляется в файл output, затем STDERR перенаправляется в STDOUT, запись в файл продолжается*

ls lala **>** output 2>&1

cat output *# С помощью команды cat читаем содержимое файла output*

ls: cannot access 'lala': No such file or directory *# Содержимое файла output в окне терминала*

В примере выше обратите внимание на & после символа перенаправления >. По правилам синтаксиса символ & нужно ставить, чтобы указать поток, в который осуществляется перенаправление.

## **Перенаправление конкретного потока в файл**

Чтобы перенаправить конкретный поток, нужно указать его номер перед >:

*# Так можно сразу перенаправить STDERR в файл*

cd lala 2> output

cat output

-bash: cd: lala: No such file or directory

## **Перенаправление обоих потоков в файл**

Этот вариант используется чаще всего. Он помогает проводить отладку и подсказывает, почему вообще возникла та или иная ошибка:

*# Оба потока (STDERR и STDOUT) перенаправляются в файл*

cd lala &> output

cat output

-bash: cd: lala: No such file or directory

# **Пайплайн**

Вы уже знаете, что у одного процесса есть вход, а у другого — выход. При этом их можно подменять. Логично предположить, что их можно и соединить. Этот подход носит название **пайплайн** (от англ. *pipeline*).

Благодаря пайплайну можно соединять программы и протаскивать данные сквозь них. Получается цепочка функций, в которой каждое звено выступает в роли преобразователя или фильтра.

Когда мы учились грепать, то делали это по какому-то одному слову. Но часто возникает задача погрепать по нескольким словам. Не важно, как они расположены внутри строки, главное, что они встречаются там вместе.

Такую функциональность можно было бы сделать, усложнив саму программу grep. Но пайплайн позволяет добиться такого же поведения без необходимости писать сложную программу:

grep alias .bashrc | grep color

*# enable color support of handy aliases*

alias grep**=**'grep --color=auto'

alias fgrep**=**'fgrep --color=auto'

alias egrep**=**'egrep --color=auto'

Обратите внимание на | — этот символ называется **пайп**. Он указывает, что командная оболочка должна взять поток STDOUT одного процесса и соединить его с потоком STDIN другого процесса.

Все утилиты, которые читают файлы, могут принимать данные через поток STDIN. Поскольку утилита grep принимает на вход и возвращает текст, то его можно комбинировать бесконечно.

cat source | grep Dog | uniq | sort

Посмотрим, как этот пример работает по шагам:

1. Читается файл *source*
2. Грепаются входные данные по подстроке Dog
3. Убираются дубли (в исходном файле две одинаковых строки Dog)
4. Сортируются входные данные и выводятся на экран

Пайплайн стал основой философии Unix, которая звучит так:

* Пишите программы, которые делают что-то одно и делают это хорошо
* Пишите программы, которые бы работали вместе
* Пишите программы, которые бы поддерживали текстовые потоки, поскольку это универсальный интерфейс

Именно поэтому большинство утилит работают с сырым текстом — принимают его на вход и возвращают в поток STDOUT.

Такой подход позволяет получать сложное поведение из крайне простых составных блоков. Такая концепция называется **стандартные интерфейсы** и хорошо отражена в конструкторах Lego.

# **Манипулирование файловой структурой**

Для создания файлов принято использовать утилиту touch. Основная задача этой утилиты — поменять время последнего доступа к файлу, но она обладает побочным эффектом.

Если файла не существует, то он будет создан — именно поэтому ее используют для создания файлов, хотя это не основное предназначение:

*# В текущей директории создается пустой файл*

touch empty-file

Удалить файл можно командой rm (сокращение от *remove*):

rm empty-file

В \*nix-системах не существует понятия «переименовать файл». Переименование всегда равносильно перемещению, которое выполняется командой mv (*move*):

touch file

mv file renamed-file

Для копирования файлов и директорий используется утилита cp (*copy*).

У этой утилиты два аргумента:

* Имя источника (откуда копируем)
* Имя приемника (куда копируем)

Посмотрим, как эта утилита работает на практике:

cp renamed-file renamed-file-copy

Для копирования директории нужно добавить флаг -r (*recursive*).

Все эти и последующие утилиты работают с файлами и директориями, расположенными в любом месте файловой системы. Поэтому вы всегда можете передать любой путь: touch /tmp/tempfile.

Утилиты для работы с директориями частично отличаются. Создание директории выполняется командой mkdir (*make directory*):

mkdir my-dir

По умолчанию эта команда не создает вложенных директорий:

mkdir one/two/three

mkdir: cannot create directory ‘one/two/three’: No such file or directory

В такой ситуации придется создавать каждую директорию отдельно. Но есть и другой способ — воспользоваться флагом -p (--parents), который создает директории рекурсивно:

mkdir -p one/two/three

Удаление директорий выполняется той же командой, что и удаление файлов, но без флагов оно выдает предупреждение:

rm my-dir/

rm: cannot remove 'my-dir/': Is a directory

Чтобы не было ошибки, нужно добавить флаг -r (*recursion*). Он включает режим рекурсивного удаления содержимого директорий. Другими словами, идет просмотр содержимого во всех вложенных директориях и поддиректориях до самого конца:

rm -r my-dir

Теперь представим такую ситуацию: внутри директории содержатся файлы или директории с ограниченными правами доступа, например, доступные только для чтения. В таком случае команда rm начнет задавать вопрос по каждому из них, нужно ли удалять файл.

Если вы точно уверены, что удалить нужно все, добавьте флаг -f (--force). Этот флаг позволяет игнорировать несуществующие файлы и не запрашивать подтверждение на удаление. В таком случае rm удалит всю директорию без вопросов:

rm -rf one

## **Что такое переменные окружения**

Переменные окружения похожи на переменные в языках программирования. Они существуют в рамках запущенной сессии командной оболочки, то есть, переменные пропадут, когда терминал закроется. Они подгружаются туда во время его инициализации, хотя это не единственный путь их появления.

Посмотреть установленные переменные можно командой env (*environment*).

env

TERM**=**xterm-256color

SHELL**=**/bin/bash

LC\_ALL**=**en\_US.UTF-8

USER**=**kirill.m

PATH**=**/home/kirill.m/bin:/home/kirill.m/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin

PWD**=**/home/kirill.m

LANG**=**en\_US.UTF-8

SHLVL**=**1

HOME**=**/home/kirill.m

LOGNAME**=**kirill.m

Без переменной окружения пришлось бы делать одну из двух вещей:

* При вызове cd каждый раз указывать, где находится домашняя директория. Например, так: cd --home-dir /home/kirill.m. Такой способ убивает весь смысл быстрого перехода в домашнюю директорию
* Договариваться, что где-то будет храниться специальный файл с настройками, в том числе домашней директории. Настройки из этого файла будут читаться командой cd при каждом запуске

Существует некоторый базовый набор переменных, которые всегда устанавливаются bash при старте. Они используются большим количеством утилит и нужны для нормального функционирования системы. Одну из таких переменных мы уже знаем — это HOME:

*# Для вывода переменной используется echo,# а перед самой переменной ставится знак $*

echo $HOME

/home/kirill.m

*# Вокруг знака `=` не должно быть пробелов*

HOME**=**/tmp

echo $HOME

/tmp

*# Возвращаем обратно*

HOME**=**/home/kirill.m

echo $HOME

/home/kirill.m

Теперь проведем эксперимент: попробуем установить HOME и выполнить переход в эту директорию с помощью команды cd, которая должна отправить нас в домашнюю директорию:

*# Достаточно выполнить присваивание слева от запускаемой команды*

HOME**=**/tmp cd /tmp

pwd

/tmp

Команда cd выполнила переход по пути, указанном в переменной HOME. Теперь снова сделаем cd, но не добавляя определение переменной слева:

cd

pwd

/home/kirill.m

echo $HOME

/home/kirill.m

В этот раз произошел переход в настоящую домашнюю директорию. Получается, что вызов HOME=/tmp cd изменил переменную только для того конкретного запуска. И действительно, существует два способа задавать значение переменной окружения: **локальный** и **глобальный**.

Когда мы указали HOME=/tmp прямо перед командой cd, то переменная не была изменена для текущей сессии — она была изменена для запущенной команды, в данном случае cd. Такой трюк можно проделывать с любой командой.

Другой способ установки изменяет значение глобально, для всей сессии и выполняется так:

echo $HOME

/home/kirill.m

export HOME**=**/tmp

cd

pwd

/tmp

echo $HOME

/tmp

Обратите внимание, что изменение происходит в рамках текущей сессии. Другими словами, если у нас открыто две вкладки терминала, то изменение произойдет только в той вкладке, в которой мы написали команду.

# **История —Основы командной строки**

Чем больше работаешь в командной строке, тем чаще возникает необходимость повторять введенные ранее команды. Самый простой способ просматривать историю команд — нажимать клавиши «вверх» и «вниз». При каждом нажатии стрелки «вверх» в поле ввода начнет появляться предыдущая выполненная команда, если нажать «вниз» — то следующая.

История команд bash хранится в специальном файле .bash\_history, который лежит в домашней директории пользователя. Каждый раз, когда пользователь вводит команду, она попадает именно в этот файл. Запись происходит при завершении сеанса. Этот файл ничем не отличается от остальных файлов: его можно открыть, посмотреть и даже отредактировать. За то, какое количество команд хранится в истории, отвечает переменная окружения HISTFILESIZE. Если она выставлена, то берется указанное в ней число. Если переменной нет, история не обрезается и файл .bash\_history растет бесконечно:

tail .bash\_history

rm -r one/

env

HOME**=**/tmp

cd

pwd

cd

echo $HOME

export HOME**=**/tmp id

exit

id

Посмотреть историю можно и более простым способом. Достаточно выполнить команду history:

history

1 docke rps

2 free -m

3 docker ps

4 docker exec -it 8678a6520641 bash

5 ls

6 exit

7 docker ps

8 docker exec -it 1209b6e5ce6b bash ...

Эта команда выведет содержимое файла .bash\_history, добавив слева номер. Если набрать history 5, то отобразятся только пять последних введенных команд. Используя номер команды в выводе history, можно выполнить повторный запуск без необходимости набирать или копировать команду из истории:

*# Эта команда имела второй номер в списке выше*

**!**2

free -m

total used free shared buff/cache available

Mem: 1998 1532 75 21 390 227

Swap: 0 0 0

# **Пользователи и группы**

Команда whoami позволяет выяснить имя пользователя:

whoami

kirill.m

## **Права пользователей**

Абсолютно любой процесс, запускаемый в операционной системе, стартует от имени некоторого пользователя. Соответственно, его возможности по влиянию на файловую систему ограничены **правами пользователя**, от имени которого процесс запущен.

Команда ps (сокращение от *process status*) выводит отчет о работающих процессах. Информацию о том, какой процесс и под каким пользователем запущен, можно получить из вывода ps aux:

ps aux

*# Левый столбец — это имя пользователя*

root 7717 0.0 0.0 4244 1504 ? S 10:52 0:00 mpstat 1 3

kirill.m 7718 0.0 0.1 36084 3236 pts/0 R+ 10:52 0:00 ps aux

alexand+ 10542 0.0 0.1 21500 2892 pts/1 Ss+ 10:10 0:00 -bash

root 11113 0.0 0.1 92796 2596 ? Ss 08:50 0:00 sshd: kirill.m **[**priv]

kirill.m 11116 0.0 0.0 45276 1408 ? Ss 08:50 0:00 /lib/systemd/systemd --user

kirill.m 11119 0.0 0.0 61148 1860 ? S 08:50 0:00 **(**sd-pam**)**

kirill.m 11194 0.0 0.0 92796 1800 ? S 08:50 0:00 sshd: kirill.m@pts/0

kirill.m 11195 0.0 0.2 21388 4448 pts/0 Ss 08:50 0:00 -bash

root 12195 0.0 0.0 0 0 ? S 10:13 0:00 **[**kworker/u30:1]

root 12880 0.0 0.1 92796 2748 ? Ss 08:55 0:00 sshd: alexander.v **[**priv]

alexand+ 12883 0.0 0.0 45276 1924 ? Ss 08:55 0:00 /lib/systemd/systemd --user

alexand+ 12884 0.0 0.0 61148 1860 ? S 08:55 0:00 **(**sd-pam**)**

alexand+ 12920 0.0 0.1 92796 2420 ? S 08:55 0:00 sshd: alexander.v@pts/1,pts/2

Модификация существующих файлов не влияет на владельца — для его смены нужно воспользоваться специальной утилитой. В домашней директории пользователя все принадлежит пользователю:

ls -la

total 44

drwxr-xr-x 5 kirill.m kirill.m 4096 Aug 29 11:34 .

drwxr-xr-x 8 root root 4096 Apr 26 10:38 ..

-rw------- 1 kirill.m kirill.m 2540 Aug 30 07:26 .bash\_history

-rw-r--r-- 1 kirill.m kirill.m 220 Aug 31 2015 .bash\_logout

-rw-r--r-- 1 kirill.m kirill.m 3771 Aug 31 2015 .bashrc

drwx------ 2 kirill.m kirill.m 4096 Mar 30 18:10 .cache

-rw------- 1 kirill.m kirill.m 55 Aug 28 18:49 .lesshst

drwxrwxr-x 2 kirill.m kirill.m 4096 Aug 29 08:35 .nano

-rw-r--r-- 1 kirill.m kirill.m 655 May 16 2017 .profile

-rw-rw-r-- 1 kirill.m kirill.m 0 Aug 29 11:27 renamed-file

drwx------ 2 kirill.m kirill.m 4096 Jan 22 2018 .ssh

-rw------- 1 kirill.m kirill.m 513 Aug 29 08:06 .viminfo

Имя пользователя в системе должно быть уникальным, но его можно менять. Если посмотреть под капот этой системы, то мы увидим, что имя пользователя связано с идентификатором, называемым **UID**. Это число, которое и определяет пользователя.

Если поменяется имя пользователя, но идентификатор UID останется прежним, то все доступы останутся. Если сменится идентификатор, то фактически сменится и пользователь. Соответственно, новый пользователь потеряет доступы к старому аккаунту.

Посмотреть свой идентификатор можно разными способами. Первый способ — с помощью команды id:

id

uid**=**1002**(**kirill.m**)** gid**=**1002**(**kirill.m**)** groups**=**1002**(**kirill.m**)**,999**(**docker**)**

Второй способ связан с просмотром одного важного файла, который выступает основным хранилищем пользователей в \*nix-системах. Да, это обычный текстовый файл, как и все остальное:

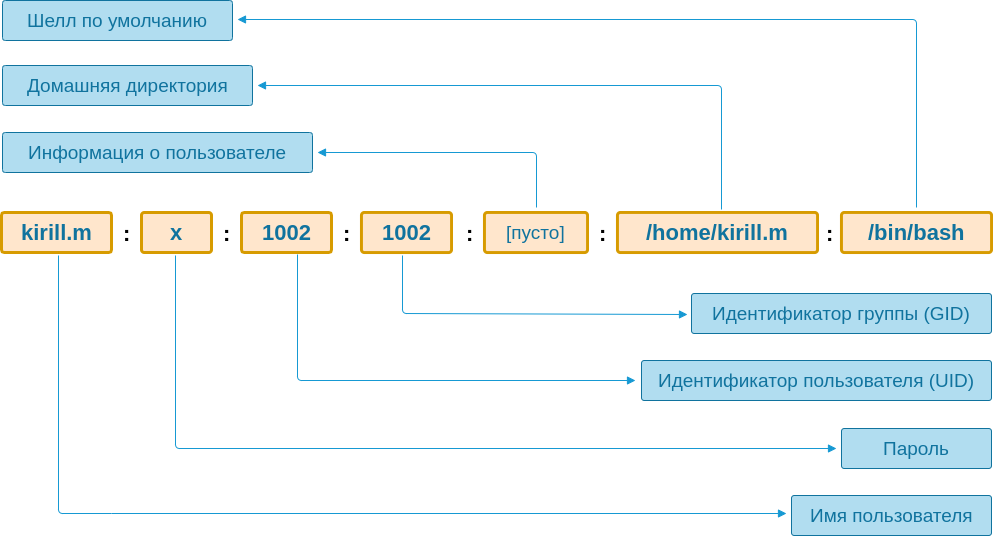
cat /etc/passwd

root:x:0:0:root:/root:/bin/bash

daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin

sync:x:4:65534:sync:/bin:/bin/sync



## **Права групп пользователей**

Кроме имени, у пользователей \*nix-систем есть связанное с ним понятие — **группа**. Она создана для группового доступа к разделяемому (общему) ресурсу — например, файлу.

Например, у нас есть группа разработчиков, которые регулярно заходят на сервер. Им нужно дать одинаковые возможности по управлению определенными файлами. Владелец у файла ровно один, поэтому мы не можем решить этот вопрос через смену владельца. Для этого нужно создать группу и привязать ее к самому пользователю.

Группы, ассоциированные с текущим пользователем, показываются в выводе команды id:

id

uid**=**1002**(**kirill.m**)** gid**=**1002**(**kirill.m**)** groups**=**1002**(**kirill.m**)**,999**(**docker**)**

# **Sudo**

В работе с командной строкой мы часто сталкиваемся с ситуациями, в которых необходимо повышать привилегии и выполнять команды от имени суперпользователя root.

Использовать sudo очень просто, достаточно написать эту команду слева от любой другой и выполнить. По умолчанию она пытается повысить привилегии до суперпользователя:

*# Нет прав на выполнение*

touch /etc/myfile

touch: cannot touch '/etc/myfile': Permission denied

*# С `sudo` все работает*

sudo touch /etc/myfile

В зависимости от настроек, утилита sudo попросит ваш пароль для входа или вообще откажется работать, сказав, что у вас нет права ее использовать. Как правило, в Ubuntu sudo спрашивает пароль и запоминает его на пять минут. На протяжении этого времени вы можете использовать sudo, не вводя пароль каждый раз.

Иногда нужно выполнить команду из-под пользователя, отличного от root. Тогда придется добавить флаг -u:

sudo -u nobody mkdir /tmp/test *# Директория создана от имени пользователя nobody*

stat /tmp/test

File: '/tmp/test'

Size: 4096 Blocks: 8 IO Block: 4096 directory

Device: ca01h/51713d Inode: 4577 Links: 2

Access: **(**0755/drwxr-xr-x**)** Uid: **(**65534/ nobody**)** Gid: **(**65534/ nogroup**)**

Если нужно произвести сразу пачку действий от имени другого пользователя, можно запустить новую оболочку внутри текущей. Другими словами, мы стартуем новую сессию:

*# Команда `sudo` запросит пароль текущего пользователя*

sudo -i

*# Проверяем, что сессия запущена от суперпользователя*

id

uid**=**0**(**root**)** gid**=**0**(**root**)** groups**=**0**(**root**)**

Главное — не забыть переключиться обратно после завершения необходимых манипуляций. Для этого наберите exit.

# **Права доступа**

Часто разработчики сталкиваются с **ошибкой доступа**:

touch /etc/myfile

touch: cannot touch '/etc/myfile': Permission denied

Она указывает, что текущий пользователь не имеет прав на создание файлов в каталоге /etc. Почему? Все дело в правах доступа.

Кроме имени пользователя и группы, с каждым файлом ассоциированы права доступа:

* **r** — чтение (*read*)
* **w** — запись (*write*)
* **x** — исполнение (*execute*)

Причем эти права задаются для трех типов пользователей:

* Владельца (User)
* Пользователей, входящих в ту же группу (Group)
* Остальных (Other) — тех, кто не попал в предыдущие две

Разберем на примере:

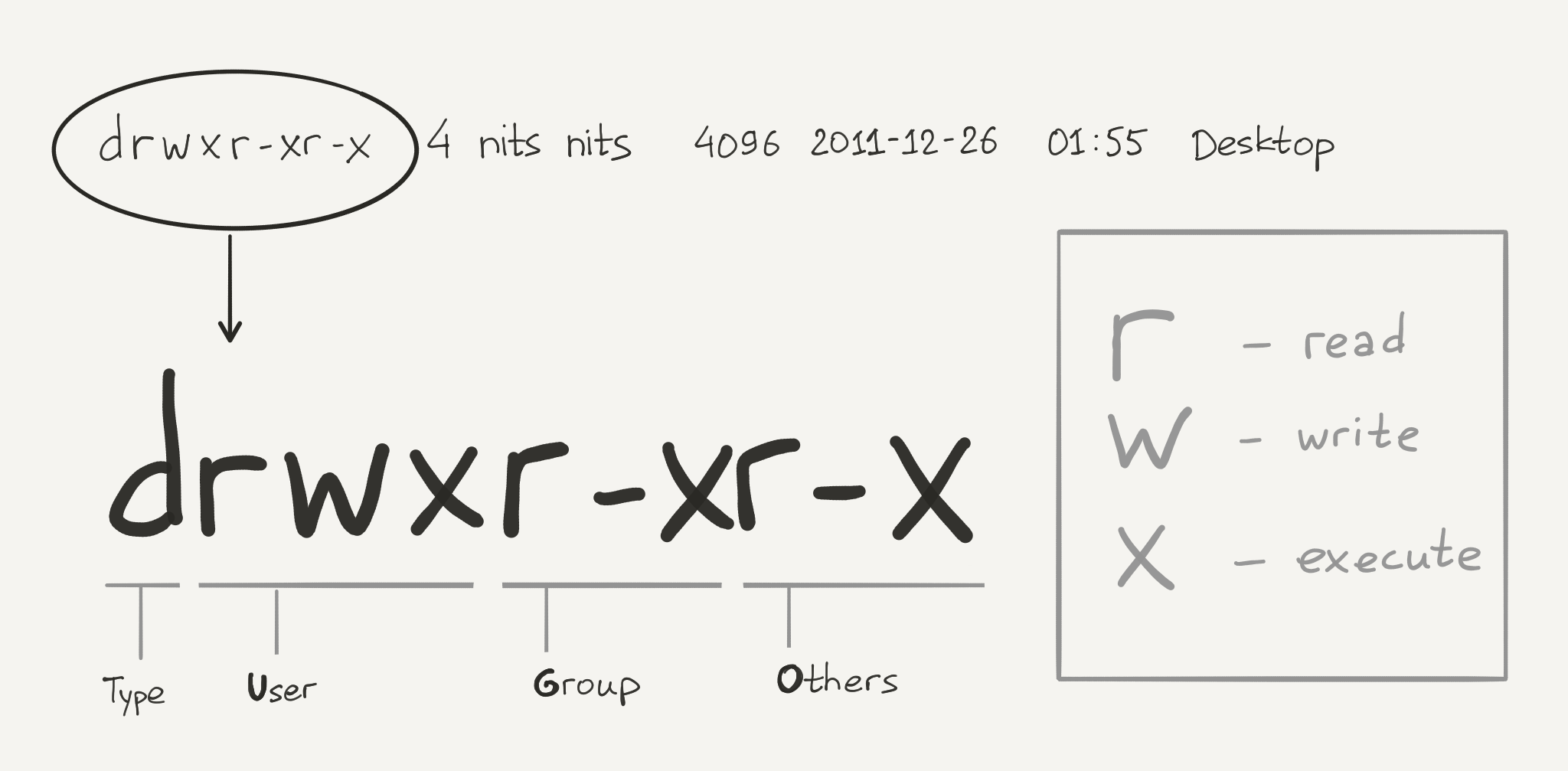
*# Пример строчки из вывода команды ls -la*

-rw-r--r-- 1 kirill.m kirill.m 3771 Aug 31 2015 .bashrc

Запись слева представляет собой один из вариантов описания прав доступа (*permissions*). Для удобства чтения разделим эту запись на группы символов:

* - — этот символ обозначает, что перед нами файл
* rw- — права первого типа (для владельца файла). Владелец может читать этот файл (r) и писать в него (w). Прочерк в конце означает, что этот файл нельзя исполнять
* r-- — права второго типа (для группы владельца). В нашем случае такие права есть у тех, кто входит в группу kirill.m. Запись r-- говорит о наличии доступа только для чтения, а изменение и исполнение запрещены.
* r-- — права третьей группы (все остальные). В нашем случае права совпадают со второй группой, поэтому файл доступен только для чтения

Не важно, какой файл или директорию мы смотрим — порядок прав в этой группе всегда один и тот же — чтение-запись-исполнение, а прочерк означает отсутствие этого права:



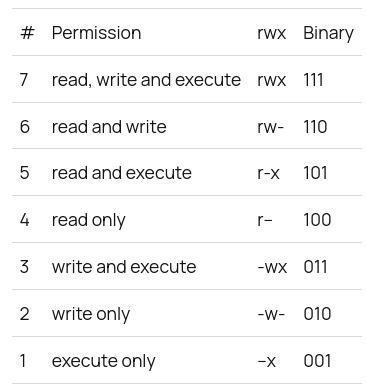
Домашняя директория имеет другие права:

* В самом начале вместо - стоит d, которая обозначает директорию
* Права для владельца — rwx, а для всех остальных — r-x. Из этого описания видно, что только пользователь может писать внутрь своей домашней директории

Что такое x в отношении директорий? Это право позволяет перемещаться в директорию и обращаться ко всем расположенным в ней файлам и каталогам. Обращаться можно при условии, что эти файлы доступны на чтение, запись или выполнение. Например, если положить доступный на чтение файл в директорию с правом x, то вы сможете прочитать этот файл. Если убрать с этой директории право x, то вы лишитесь доступа к файлу.

Но что тогда такое чтение? Здесь все более интуитивно понятно. По сути, директория — это список файлов, поэтому право на чтение позволяет прочитать этот список, а именно вывести список имен файлов, содержащихся в директории.

В статьях, книгах и руководствах иногда используется другой способ описания прав доступа: 755. Это не одно число, а три числа. Каждое из которых представляет собой группу rwx для наших типов пользователя в том же порядке: для владельца, входящих в группу и всех остальных. Число 0 означает, что нет никаких прав для данного типа пользователей:



Попробуем перевести несколько примеров из буквенной формы в числовую:

* drwxr-xr-x соответствует 755
* -rw-r--r-- соответствует 644

Однако пользователь root находится вне этой системы. Для него не имеет никакого значения наличие любых прав — суперпользователь может все.

Для изменения владельца файлов или директорий можно использовать утилиту chown. В самом простом использовании команда принимает имя пользователя и файл (или директорию) для которого нужно сменить пользователя:

chown kirill .bashrc

Для изменения прав доступа к файлам и директориям можно использовать утилиту chmod. Утилита принимает права доступа и путь к файлу. Права доступа можно записывать обоими способами. С помощью знаков + и - права добавляются или удаляются соответственно:

chmod +r file.txt

chmod 754 file.txt

# **Пакетный менеджер**

Привычный способ ставить программы для Windows-пользователя — бродить по интернету в поисках нужных приложений. В \*nix-системах такой способ используется значительно реже. Для установки программ придумали **пакетные менеджеры**.

В некоторых операционных системах есть пакетный менеджер по умолчанию, в других его надо устанавливать самостоятельно. Например, для macOS есть пакетный менеджер brew.

Далее, для всех примеров будет использоваться пакетный менеджер **apt**, который поставляется с Ubuntu. Например, установка PHP в Ubuntu выглядит так:

*# Требует sudo, потому что мы устанавливаем в системную директорию*

sudo apt install php

*# Здесь будет много текста и вопрос, устанавливать ли? Если да, то PHP установится*

php -v

PHP 7.4.3 **(**cli**)** **(**built: Feb 20 2020 08:51:50**)** **(** NTS **)**

Установленные пакеты можно удалять. Для этого используется команда apt remove:

apt remove php

*# Тут тоже много вывода и вопрос, удалять ли?*

Иногда это может приводить к ситуациям, что пакет в реальности существует, но apt утверждает обратное. Если вы чувствуете подвох, то запустите ручное обновление локального индекса:

sudo apt update

*# Теперь можно устанавливать*

sudo apt install php