Лабораторная работа № 6. Права доступа

Работая с Linux, Мефодий заметил, что некоторые файлы и каталоги недоступны ему ни на чтение, ни на запись, ни на использование. Зачем такие нужны? Оказывается, другие пользователи *могут* обращаться к этим файлам, а у Мефодия просто *не хватает прав*.

Идентификатор пользователя

Говоря о *правах* доступа пользователя к файлам, стоит заметить, что в действительности манипулирует файлами не сам пользователь, а запущенный им **процесс** (например, утилита `rm` или `cat`). Поскольку и файл, и процесс создаются и управляются системой, ей нетрудно организовать какую угодно политику доступа одних к другим, основываясь на любых свойствах процессов как *субъектов* и файлов как *объектов* системы.

В Linux, однако, используются не какие угодно свойства, а результат **идентификации** пользователя -- его **UID**. Каждый процесс системы обязательно *принадлежит* какому-нибудь пользователю, и **идентификатор пользователя** (**UID**) -- обязательное свойство любого процесса Linux. Когда программа `login` запускает **стартовый командный интерпретатор**, она приписывает ему **UID**, полученный в результате диалога. Обычный запуск программы (`exec()`) или порождение нового процесса (`fork()`) не изменяют UID процесса, поэтому *все* процессы, запущенные пользователем во время терминальной сессии, будут иметь его идентификатор.

Поскольку **UID** однозначно определяется **входным именем**, оно нередко используется *вместо* идентификатора -- для наглядности. Например, вместо выражения "идентификатор пользователя, соответствующий входному имени `methody`", говорят "**UID** `methody`" (в примере ниже этот идентификатор равен `503`),

```
[methody@localhost methody]$ id
 uid=503(methody) gid=503(methody) группы=503(methody)
[methody@localhost methody]$ id shogun
 uid=400(shogun) gid=400(shogun)
группы=400(shogun),4(adm),10(wheel),19(proc)
```

Как узнать идентификаторы пользователя и членство в группах

Утилита `id`, которой воспользовался Мефодий, выводит **входное имя** пользователя и соответствующий ему **UID**, а также **группу по умолчанию** и полный список **групп**, членом которых он является.

Идентификатор группы

Как было рассказано на предыдущих занятиях, пользователь может быть членом нескольких **групп**, равно как и несколько пользователей может быть членами одной и той же группы. Исторически сложилось так, что одна из групп -- **группа по умолчанию** -- является для пользователя основной: когда

(не вполне точно) говорят о "**GID** пользователя", имеют в виду именно идентификатор группы по умолчанию. На следующих занятиях будет рассказано, что **GID** пользователя вписан в **учётную запись** и хранится в `/etc/passwd`, а информация о соответствии *имён* групп их идентификаторам, равно как и о том, в какие *ещё* группы входит пользователь -- в файле `/etc/group`. Из этого следует, что пользователь *не может не быть* членом как минимум одной группы, как снаряд не может не попасть в эпицентр взрыва[1].

Часто процедуру *создания* пользователя проектируют так, что имя группы по умолчанию совпадает с **входным именем** пользователя, а **GID** пользователя -- с его **UID**. Однако это совсем не обязательно: не всегда нужно заводить для пользователя отдельную группу, а если заводить, то не всегда *удаётся* сделать так, чтобы *желаемый* идентификатор группы совпадал с *желаемым* идентификатором пользователя.

Ярлыки объектов файловой системы

При *создании* объектов файловой системы -- файлов, каталогов и т. п. -- каждому в обязательном порядке приписывается **ярлык**. Ярлык включает в себя **UID** -- идентификатор пользователя-*хозяина* файла, **GID** -- идентификатор группы, которой принадлежит файл, тип объекта и набор т. н. **атрибутов**, а также некоторую дополнительную информацию. **Атрибуты** определяют, кто и что с файлом имеет право делать, и описаны ниже.

```
[methody@arnor methody]$ ls -l
итого 24
drwx------  2 methody methody 4096 Сен 12 13:58 Documents
drwxr-xr-x  2 methody methody 4096 Окт 31 15:21 examples
-rw-r--r--  1 methody methody   26 Сен 22 15:21 loop
-rwxr-xr-x  1 methody methody   23 Сен 27 13:27 script
drwx------  2 methody methody 4096 Окт  1 15:07 tmp
-rwxr-xr-x  1 methody methody   32 Сен 22 13:26 to.sort
```

Права доступа к файлам и каталогам, показанные командой `ls -l`

Ключ "`-l`" утилиты `ls` определяет "длинный" (long) формат выдачи (справа налево): имя файла, время последнего изменения файла, размер в байтах, группа, хозяин, количество **жёстких ссылок** и строчка **атрибутов**. Первый символ в строчке атрибутов определяет тип файла. Тип "`-`" отвечает "обычному" файлу, а тип "`d`" -- каталогу (directory). Имя пользователя и имя группы, которым принадлежит содержимое домашнего каталога Мефодия, -- естественно, `methody`.

Быстрый разумом Мефодий немедленно заинтересовался вот чем: несмотря на то, что создание *жёстких ссылок* на каталог невозможно, поле "количество жёстких ссылок" для *всех* каталогов примера равно *двум*, а не одному. На самом деле этого и следовало ожидать, потому что *любой* каталог

[1] Здесь есть тонкость. В файле `group` указываются не идентификаторы, а **входные имена** пользователей. Формально говоря, можно создать двух пользователей с *одинаковым* UID, но *разными* GID и списками групп. Обычно так не делают: надобности -- почти никакой, а неразберихи возникнет много.

файловой системы Linux всегда имеет не менее двух имён: собственное (например, `tmp`) и имя "." в самом этом каталоге (`tmp/.`). Если же в каталоге создать подкаталог, количество жёстких ссылок на этот каталог увеличится на 1 за счёт имени ".." в подкаталоге (например, `tmp/subdir1/..`):

```
[methody@arnor methody]$ ls -ld tmp
 drwx------  3 methody methody 4096 Окт  1 15:07 tmp
[methody@arnor methody]$ mkdir tmp/subdir2
[methody@arnor methody]$ ls -ld tmp
 drwx------  4 methody methody 4096 Окт  1 15:07 tmp
[methody@arnor methody]$ rmdir tmp/subdir*
[methody@arnor methody]$ ls -ld tmp
 drwx------  2 methody methody 4096 Окт  1 15:07 tmp
```

Несколько жёстких ссылок на *каталог* всё-таки бывают!

Здесь Мефодий использовал ключ "`-d`" (directory) для того, чтобы `ls` выводил информацию не о *содержимом* каталога `tmp`, а о самом этом каталоге.

Пример от рута есть смысл приводить только в том случае, если пример на что-то *совершенно* универсальное, что обязательно будет устроено точно так же в любом Линуксе. Иначе есть опасность, что пользователь начнёт мудрить -- и что он там намудрит... В Linux определено несколько **системных групп**, задача которых -- обеспечивать доступ членов этих групп к разнообразным ресурсам системы. Часто такие группы носят говорящие названия: "`disk`", "`audio`", "`cdwriter`" и т. п. Тогда обычным пользователям доступ к некоторому файлу, каталогу или файлу-дырке Linux закрыт, но открыт членам группы, которой этот объект принадлежит.

Например, в Linux почти всегда используется **виртуальная файловая система** `/proc` -- каталог, в котором в виде подкаталогов и файлов представлена информация из **таблицы процессов**. Имя подкаталога `/proc` совпадает с **PID** соответствующего процесса, а содержимое этого подкаталога отражает свойства процесса. *Хозяином* такого подкаталога будет хозяин процесса (с правами на чтение и использование), поэтому *любой* пользователь сможет посмотреть информацию о *своих* процессах. Именно каталогом `/proc` пользуется утилита `ps`:

```
[methody@arnor methody]$ ls -l /proc
 . . .
 dr-xr-x---   3 methody proc          0 Сен 22 18:17 4529
 dr-xr-x---   3 shogun  proc          0 Сен 22 18:17 4558
 dr-xr-x---   3 methody proc          0 Сен 22 18:17 4589
 . . .
[methody@localhost methody]$ ps -af
 UID         PID  PPID  C STIME TTY          TIME CMD
 methody    4529  4523  0 13:41 tty1     00:00:00 -bash
 methody    4590  4529  0 13:42 tty1     00:00:00 ps -af
```

Ограничение доступа к полной таблице процессов

Оказывается, запущено немало процессов, в том числе один -- пользователем `shogun` (**PID** `4558`). Однако, несмотря на ключ "`-a`" (all), `ps` выдала Мефодию только сведения о его процессах: `4529` -- это входной shell, а `4589` -- видимо, сам `ls`.

Другое дело -- Гуревич. Он, как видно из примера, входит в группу `proc`, членам которой разрешено читать и использовать *каждый* подкаталог `/proc`.

```
shogun@localhost ~ $ ps -af
 UID        PID  PPID  C STIME TTY          TIME CMD
 methody   4529  4523  0 13:41 tty1     00:00:00 -bash
 shogun    4556  1626  0 13:41 tty3     00:00:00 -zsh
 shogun    4598  4556  0 13:41 tty3     00:00:00 ps -af
```

Доступ к полной таблице процессов: группа `proc`

Гуревич, опытный пользователь Linux, предпочитает `bash`-у "The Z Shell", `zsh`. Отсюда и различие приглашения в командной строке. Во всех shell-ах, кроме самых старых, символ "`~`" означает домашний каталог. Этим сокращением удобно пользоваться, если текущий каталог -- не домашний, а оттуда (или туда) нужно скопировать файл. Получается команда наподобие "`cp ~/`*`нужный_файл`* `.`" или "`cp `*`нужный_файл`* `~`" соответственно.

Команда `ps -a` выводит информацию обо *всех* процессах, запущенных "живыми" (а не системными) пользователями[2]. Для просмотра *всех* процессов Гуревич пользуется командой `ps -efH`.

Разделяемые каталоги

Проанализировав систему прав доступа к каталогам в Linux, Мефодий пришёл к выводу, что в ней имеется существенный недочёт. Тот, кто имеет право *изменять* каталог, может *удалить любой файл* оттуда, даже такой, к которому совершенно не имеет доступа. Формально всё правильно: удаление файла из каталога -- всего лишь изменение содержимого каталога. Если у файла было больше одного имени (существовало несколько жёстких ссылок на этот файл), никакого удаления *данных* не произойдёт, а если ссылка была последней -- файл в самом деле удалится. Вот это-то, по мнению Мефодия, и плохо.

Чтобы доказать новичку, что право на удаление любых файлов *полезно*, кто-то создал прямо в домашнем каталоге Мефодия файл, совершенно ему недоступный, да к тому же с подозрительным именем, содержащим пробел:

```
[methody@localhost methody]$ ls
 4TO-TO Mep3koe  Documents  examples  loop  script  tmp  to.sort
[methody@localhost methody]$ ls -l 4*
 -rw-------  1 root root 0 Сен 22 22:20 4TO-TO Mep3koe
[methody@localhost methody]$ rm -i 4*
 rm: удалить защищённый от записи пустой обычный файл `4TO-TO Mep3koe'? y
```

Удаление чужого файла с неудобным именем

Подозревая, что от хулиганов всего можно ожидать, Мефодий не решился *набрать* имя удаляемого файла с клавиатуры, а воспользовался *шаблоном* и ключом "`-i`" (interactive) команды `rm`, чтобы та ожидала

---

[2] Более точно -- обо всех процессах, имеющих право выводить на какой-нибудь терминал, а значит, запущенными из терминального сеанса.

подтверждения перед тем, как удалять очередной файл. Несмотря на отсутствие доступа к самому файлу, удалить его оказалось возможно.

```
[methody@localhost methody]$ ls -dl /tmp
 drwxrwxrwt  4 root root 1024 Сен 22 22:30 /tmp
[methody@localhost methody]$ ls -l /tmp
 итого 4
 -rw-r--r--  1 root    root    13 Сен 22 17:49 read.all
 -rw-r-----  1 root    methody 23 Сен 22 17:49 read.methody
 -rw-------  1 methody root    25 Сен 22 22:30 read.Methody
 -rw-r-----  1 root    wheel   21 Сен 22 17:49 read.wheel
[methody@localhost methody]$ rm -f /tmp/read.*
 rm: невозможно удалить `/tmp/read.all': Operation not permitted
 rm: невозможно удалить `/tmp/read.methody': Operation not permitted
 rm: невозможно удалить `/tmp/read.wheel': Operation not permitted
[methody@localhost methody]$ ls /tmp
 read.all  read.methody  read.wheel
```

Работа с файлами в разделяемом каталоге

Убедившись, что *любой* доступ в каталог `/tmp` открыт *всем*, Мефодий захотел удалить оттуда все файлы. Затея гораздо более хулиганская, чем *заведение* файла: а вдруг они кому-нибудь нужны? Удивительно, но удалить удалось только файл, принадлежащий самому Мефодию...

Дело в том, что Мефодий проглядел особенность атрибутов каталога `/tmp`: вместо "`x`" в тройке "для посторонних" `ls` выдал "`t`". Это *ещё* один атрибут каталога, наличие которого как раз и запрещает пользователю удалять оттуда файлы, которым он не хозяин. Таким образом, права записи в каталог с ярлыком "`drwxrwxrwt` *группа хозяин*" и для членов группы *группа*, и для посторонних ограничены их собственными файлами, и только *хозяин* имеет право изменять список файлов в каталоге, как ему вздумается. Такие каталоги называются **разделяемыми**, потому что предназначены они, как правило, для *совместной* работы всех пользователей в системе, обмена информацией и т. п.

При установке атрибута "`t`" доступ на использование для посторонних ("`t`" в строчке атрибутов стоит на месте последнего "`x`") не отменяется. Просто они так редко используются друг без друга, что `ls` выводит их в одном и том же месте. Если кому-нибудь придёт в голову организовать разделяемый каталог *без* доступа посторонним на использование, `ls` выведет на месте девятого атрибута не "`t`", а "`T`".

```
[methody@localhost methody]$ ls -l loop
 -rw-r--r--  1 root root 26 Сен 22 22:10 loop
[methody@localhost methody]$ chown methody loop
 chown: изменение владельца `loop': Operation not permitted
[methody@localhost methody]$ cp loop loopt
[methody@localhost methody]$ ls -l loop*
 -rw-r--r--  1 root    root    26 Сен 22 22:10 loop
 -rw-r--r--  1 methody methody 26 Сен 22 22:15 loopt
[methody@localhost methody]$ mv -f loopt loop
[methody@localhost methody]$ ls -l loop*
 -rw-r--r--  1 methody methody 26 Сен 22 22:15 loop
```

Что можно делать с чужим файлом в своём каталоге

Оказывается, мелкие пакости продолжаются. Кто-то сменил файлу `loop` хозяина, так что теперь Мефодий может только читать его, но не изменять. Удалить этот файл -- проще простого, но хочется "вернуть всё как было": чтобы получился файл с тем же именем и тем же содержанием, принадлежащий Мефодию, а не `root`-у. Это несложно: чужой файл можно переименовать (это действие над каталогом, а не над файлом), скопировать переименованный файл в файл с именем старого (доступ по чтению открыт) и, наконец, удалить чужой файл с глаз долой. Ключ "`-f`" (force, "силком") позволяет утилите `mv` делать своё дело, не спрашивая подтверждений. В частности, увидев, что файл с именем, в которое необходимо переименовывать, существует, даже чужой, даже недоступный на запись, `mv` преспокойно удалит его и выполнит операцию переименования.

Суперпользователь

Мефодий изрядно возмутился, узнав, что кто-то может проделывать *над ним* всякие штуки, которые сам Мефодий ни над кем проделывать не может. Обоснованное подозрение пало на Гуревича, единственного администратора этой системы, обладающего правами **суперпользователя**.

суперпользователь	Единственный пользователь в Linux, на которого не распространяются ограничения прав доступа. Имеет нулевой **идентификатор пользователя**.

Суперпользователь в Linux -- это *выделенный* пользователь системы, на которого *не распространяются* ограничения прав доступа. **UID** суперпользовательских процессов равен `0`: так система отличает их от процессов других пользователей. Именно суперпользователь имеет возможность произвольно изменять владельца и группу файла. Ему открыт доступ на чтение и запись к *любому файлу* системы и доступ на чтение, запись и использование к *любому каталогу*. Наконец, суперпользовательский процесс может на время сменить *свой собственный* **UID** с нулевого на любой другой. Именно так и поступает программа `login`, когда, проведя процедуру идентификации пользователя, запускает **стартовый командный интерпретатор**.

Среди учётных записей Linux всегда есть запись по имени `root` ("корень"[3]), соответствующая нулевому идентификатору, поэтому вместо "суперпользователь" часто говорят "root". Множество системных файлов принадлежат `root`-у, множество файлов только ему доступны на чтение или запись. Пароль этой учётной записи -- одна из самых больших драгоценностей системы. Именно с её помощью системные администраторы выполняют самую ответственную работу. Свойство `root` иметь доступ ко *всем* ресурсам системы накладывает очень высокие требования на *человека*, знающего пароль `root`. Суперпользователь может всё -- в том числе и всё поломать,

[3] Вместо **полного имени** такому пользователю часто пишут "root of all evil".

поэтому *любую работу* стоит вести с правами обычного пользователя, а к правам `root` прибегать только по необходимости.

Существует два различных способа получить права суперпользователя. Первый -- это *зарегистрироваться в системе* под этим именем, ввести пароль и получить стартовую оболочку, имеющую нулевой **UID**. Это -- самый неправильный способ, пользоваться которым стоит, только если нельзя применить другие. Что в этом случае выдаст команда `last`? Что тогда-то с такой-то консоли в систему вошёл *неизвестно кто* с правами *суперпользователя* и что-то там такое *делал*. С точки зрения системного администратора, это -- очень подозрительное событие, особенно, если сам он в это время к указанной консоли не подходил... сами администраторы такой способ не любят.

Второй способ -- это воспользоваться специальной утилитой `su` (shell of user), которая позволяет выполнить одну или несколько команд от лица другого пользователя. По умолчанию эта утилита выполняет команду `sh` от лица пользователя `root`, то есть запускает командный интерпретатор с нулевым UID. Отличие от предыдущего способа -- в том, что всегда известно, *кто именно* запускал `su`, а значит, с кого спрашивать за последствия. В некоторых случаях удобнее использовать не `su`, а утилиту `sudo`, которая позволяет выполнять *только заранее заданные* команды.

Подмена идентификатора

Утилиты `su` и `sudo` имеют некоторую странность, объяснить которую Мефодий пока не в состоянии. Эта же странность распространяется и на давно известную программу `passwd`, которая позволяет редактировать собственную **учётную запись**. Запускаемый процесс *наследует* **UID** от родительского, поэтому, если этот **UID** -- не нулевой, он не в состоянии *поменять* его. Тогда как же `su` запускает для обычного пользователя *суперпользовательский* shell? Как `passwd` получает доступ к хранилищу *всех* учётных записей? Должен существовать механизм, позволяющий пользователю запускать процессы с идентификаторами *другого* пользователя, причём механизм *строго* контролируемый, иначе с его помощью можно натворить немало бед.

В Linux этот механизм называется **подменой идентификатора** и устроен очень просто. Процесс может сменить свой **UID**, если запустит вместо себя при помощи `exec()` *другую* программу из файла, имеющего специальный атрибут **SetUID**[4]. В этом случае **UID** процесса становится равным **UID** *файла*, из которого программа была запущена.

---

[4] Строго говоря, при этом меняется не собственно идентификатор пользователя, а т. н. **исполнительный идентификатор пользователя**, EUID; это нужно для того, чтобы знать, кто *на самом деле* запустил программу.

```
[foreigner@somewhere foreigner]$ ls -l /usr/bin/passwd /bin/su
 -rws--x--x  1 root root 19400 Фев  9  2004 /bin/su
 -rws--x--x  1 root root 5704  Янв 18  2004 /usr/bin/passwd
[foreigner@somewhere foreigner]$ ls -l /etc/shadow
 -r--------  1 root root 5665 Сен 10 02:08 /etc/shadow
```

Обычная программа `passwd`, использующая `SetUID`

Как и в случае с t-атрибутом, `ls` выводит букву "`s`" *вместо* буквы "`x`" в тройке "для хозяина". Точно так же, если соответствующего x-атрибута нет (что бывает редко), `ls` выведет "`S`" вместо "`s`". Во многих дистрибутивах Linux и `/bin/su`, и `/usr/bin/passwd` имеют установленный **SetUID** и принадлежат пользователю `root`, что и позволяет `su` запускать процессы с правами этого пользователя (а значит, и любого другого), а `passwd` -- модифицировать файл `/etc/shadow`, содержащий в таких системах сведения обо *всех* учётных записях. Как правило, SetUID-ные файлы доступны обычным пользователям только на выполнение, чтобы не провоцировать этих обычных пользователей рассматривать содержимое этих файлов и исследовать их *недокументированные возможности*. Ведь если обнаружится способ заставить, допустим, программу `passwd` *выполнить* любую другую программу, то все проблемы с защитой системы от взлома будут разом решены -- нет защиты, нет и проблемы.

Однако Мефодий работает с такой системой, где `/usr/bin/passwd` вообще не имеет атрибута SetUID. Зато эта программа принадлежит *группе* `shadow` и имеет другой атрибут, **SetGID**, так что при её запуске процесс получает идентификатор группы `shadow`. Утилита `ls` выводит **SetGID** в виде "`s`" вместо "`x`" во второй тройке атрибутов ("для группы"). Замечания касательно "`s`", "`S`" и "`x`" действительны для SetGID так же, как и для SetUID.

```
[root@localhost root]# ls -l /usr/bin/passwd
 -rwx--s--x  1 root shadow 5704 Jan 18  2004 /usr/bin/passwd
[root@localhost root]# ls -al /etc/tcb/methody
 total 3
 drwx--s---   2 methody auth   1024 Sep 22 12:58 .
 drwx--x---  55 root    shadow 1024 Sep 22 18:41 ..
 -rw-r-----   1 methody auth     81 Sep 22 12:58 shadow
 -rw-------   1 methody auth      0 Sep 12 13:58 shadow-
 -rw-------   1 methody auth      0 Sep 12 13:58 shadow.lock
```

Не подверженная взлому программа `passwd`, использующая `SetGID`

Каталог `/etc/tcb` в этой системе содержит подкаталоги, соответствующие входным именам всех её пользователей. В каждом подкаталоге хранится, в числе прочего, *собственный* файл `shadow` соответствующего пользователя. Доступ к каталогу `/etc/tcb` на *использование* (а следовательно, и ко всем его подкаталогам) имеют, кроме `root`, только члены группы `shadow`. Доступ на *запись* к каталогу `/etc/tcb/methody` и к файлу `/etc/tcb/methody/shadow` имеет только пользователь `methody`. Значит, чтобы изменить что-либо в этом файле, процесс *обязан* иметь **UID** `methody` и **GID** `shadow` (или нулевой **UID**, конечно). Именно такой процесс запускается из `/usr/bin/passwd`: он наследует **UID** у командного интерпретатора и получает **GID** `shadow` из-за атрибута **SetGID**. Выходит, что даже найдя в программе

`passwd` ошибки и заставив её делать что угодно, злоумышленник всего только и сможет, что... отредактировать *собственную* учётную запись!

Оказывается, атрибут **SetGID** можно присваивать *каталогам*. В последнем примере каталог `/etc/tcb/methody` имел SetGID, поэтому все создаваемые в нём файлы получают **GID**, равный **GID** самого каталога -- `auth`. Используется это разнообразными процессами идентификации, который, не будучи суперпользовательскими, но входя в группу `auth` и `shadow`, могут прочесть информацию из файлов `/etc/tcb/`*`входное_имя`*`/shadow`.

Вполне очевидно, что подмена идентификатора распространяется на *программы*, но не работает для *сценариев*. В самом деле, при исполнении сценария *процесс* запускается не из него, а из программы-интерпретатора. Файл со сценарием передаётся всего лишь как параметр командной строки, и подавляющее большинство интерпретаторов просто не обращают внимания на атрибуты этого файла. Интерпретатор наследует UID у родительского процесса, так что если он и обнаружит SetUID у сценария, поделать он всё равно ничего не сможет.

Восьмеричное представление атрибутов

Все двенадцать атрибутов можно представить в виде *битов* двоичного числа, равных 1, если атрибут установлен, и 0, если нет. Порядок битов в числе следующий: `sU|sG|t|rU|wU|xU|rG|wG||xG|rO|wO|xO`. где `sU` -- это SetUID, `sG` -- это SetGID, `t` -- это t-атрибут, после чего следуют три тройки атрибутов доступа. Этим форматом можно пользоваться в команде `chmod` вместо конструкции "*`роли=виды_доступа`*", причём число надо записывать в *восьмеричной системе счисления*. Так, атрибуты каталога `/tmp` будут равны `1777`, атрибуты `/bin/su` -- `4711`, атрибуты `/bin/ls` -- `755` и т. д.

Тем же побитовым представлением атрибутов регулируются и права доступа по умолчанию при *создании* файлов и каталогов. Делается это с помощью команды `umask`. Единственный параметр `umask` -- восьмеричное число, задающее атрибуты, которые *не надо* устанавливать новому файлу или каталогу. Так, `umask 0` приведёт к тому, что файлы будут создаваться с атрибутами "`rw-rw-rw-`", а каталоги -- "`rwxrwxrwx`". Команда `umask 022` убирает из атрибутов по умолчанию права доступа на запись для всех, кроме хозяина (получается "`rw-r--r--`" и "`rwxr-xr-x`" соответственно), а с `umask 077` новые файлы и каталоги становятся для них полностью недоступны ("`rw-------`" и "`rwx------`")[5].

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое идентификатор пользователя
2. Назначение утилиты `id`
3. Что такое идентификатор группы

---

[5] Параметр команды `umask` должен обязательно начинаться на `0`, как это принято для восьмеричных чисел в языке Си.

4. Понятие ярлыка объектов файловой системы
5. Назначение ключей `-l, -d` команды `ls`
6. Назначение ключа `-i` команды `rm`
7. Понятие разделяемого каталога
8. Назначение ключа `-f` утилиты `mv`
9. Как удалить «чужой» файл
10. Понятие суперпользователя
11. Способы получить права суперпользователя
12. Понятие подмена идентификатора
13. Пояснить особенности восьмеричного представления атрибутов