Лабораторная работа № 5. Процессы

Как уже упоминалось на прошлых занятиях, загрузка Linux завершается тем, что на всех виртуальных консолях (на самом деле -- на всех терминалах системы), предназначенных для работы пользователей, запускается программа `getty`. Программа выводит приглашение и ожидает активности пользователя, который может захотеть работать именно на этом терминале. Введённое входное имя `getty` передаёт программе `login`, которая вводит пароль и определяет, разрешено ли работать в системе с этим входным именем и этим паролем. Если `login` приходит к выводу, что работать можно, он запускает **стартовый командный интерпретатор**, посредством которого пользователь и командует системой.

Выполняющаяся программа называется в Linux **процессом**. Все процессы система регистрирует в **таблице процессов**, присваивая каждому уникальный номер -- **идентификатор процесса** (process identificator, PID). Манипулируя процессами, система имеет дело именно с их идентификаторами, другого способа отличить один процесс от другого, по большому счёту, нет. Для просмотра *своих* процессов можно воспользоваться утилитой `ps` ("process status"):

```
[methody@localhost methody]$ ps -f
 UID        PID  PPID  C STIME TTY          TIME CMD
 methody   3590  1850  0 13:58 tty3     00:00:00 -bash
 methody   3624  3590  0 14:01 tty3     00:00:00 ps -f
```

Просмотр таблицы собственных процессов

Здесь Мефодий вызвал `ps` с ключом "`-f`" ("**full**"), чтобы добыть побольше информации. Представлены оба принадлежащих ему процесса: стартовый командный интерпретатор, `bash`, и выполняющийся `ps`. Оба процесса запущены с терминала `tty3` (третьей системной консоли), и имеют идентификаторы `3590` и `3624` соответственно. В поле `PPID` ("**p**arent **p**rocess **id**entificator") указан идентификатор **родительского процесса**, т. е. процесса, *породившего* данный. Для `ps` это -- `bash`, а для `bash`, очевидно, `login`, так как именно он запускает стартовый shell. В выдаче не оказалось строки для этого `login`, равно как и для большинства других процессов системы, так как они не принадлежат пользователю `methody`.

процесс	Выполняющаяся программа в Linux. Каждый процесс имеет уникальный **идентификатор процесса**, PID. Процессы получают доступ к ресурсам системы (оперативной памяти, файлам, внешним устройствам и т. п.) и могут изменять их содержимое. Доступ регулируется с помощью **идентификатора пользователя** и **идентификатора группы**, которые система присваивает каждому процессу.

Запуск дочерних процессов

Запуск одного процесса *вместо* другого устроен в Linux с помощью **системного вызова** `exec()`. Старый процесс из памяти удаляется навсегда, вместо него загружается новый, при этом настройка *окружения* не меняется, даже PID остаётся прежним. *Вернуться* к выполнению старого процесса невозможно, разве что запустить его по новой с помощью того же `exec()` (от "execute" -- "исполнить"). Кстати, *имя файла* (программы), из которого запускается процесс, и *собственное имя* процесса (в таблице процессов) могут и не совпадать. Собственное имя процесса -- это такой же параметр командной строки, как и те, что передаются ему пользователем: для `exec()` требуется и путь к файлу, и *полная* командная строка, *нулевой* (стартовый) элемент которой -- как раз название команды

Нулевой параметр -- `argv[0]` в терминах языка Си и `$0` в терминах shell

---

Вот откуда "-" в начале имени *стартового* командного интерпретатора (`-bash`): его "подсунула" программа `login`, чтобы была возможность отличать его от других запущенных тем же пользователем оболочек.

Для работы командного интерпретатора недостаточно одного `exec()`. В самом деле, shell не просто запускает утилиту, а дожидается её завершения, обрабатывает результаты её работы и продолжает диалог с пользователем. Для этого в Linux служит системный вызов `fork()` ("вилка, развилка"), применение которого приводит к возникновению ещё одного, *дочернего*, процесса -- точной копии породившего его *родительского*. **Дочерний процесс** ничем не отличается от родительского: имеет такое же окружение, те же стандартный ввод и стандартный вывод, одинаковое содержимое памяти и продолжает работу с той же самой точки (возврат из `fork()`). Отличия два: во-первых, эти процессы имеют *разные* PID, под которыми они зарегистрированы в таблице процессов, а во-вторых, различается **возвращаемое значение** `fork()`: родительский процесс получает в качестве результата `fork()` идентификатор процесса-потомка, а процесс-потомок получает "`0`".

Дальнейшие действия shell при запуске какой-либо программы очевидны. Shell-потомок немедленно вызывает эту программу с помощью `exec()`, а shell-родитель дожидается завершения работы процесса-потомка (PID которого ему известен) с помощью ещё одного системного вызова, `wait()`. Дождавшись и проанализировав результат команды, shell продолжает работу.

```
[methody@localhost methody]$ cat > loop
 while true; do true; done
^D
[methody@localhost methody]$ sh loop
^C
[methody@localhost methody]$
```

Создание бесконечно выполняющегося сценария

По совету Гуревича Мефодий создал **сценарий** для `sh` (или `bash`, на *таком* уровне их команды совпадают), который ничего не делает. Точнее было бы сказать, что этот сценарий делает *ничего*, бесконечно повторяя в цикле команду, вся работа которой состоит в том, что она завершается без ошибок (на следующих занятиях будет сказано о том, что "`> файл`" в командной строке просто *перенаправляет* стандартный вывод команды в `файл`). Запустив этот сценарий с помощью команды вида `sh имя_сценария`, Мефодий ничего не увидел, но услышал, как загудел вентилятор охлаждения центрального процессора: машина трудилась! Управляющий символ "`^C`", как обычно, привёл к завершению активного процесса, и командный интерпретатор продолжил работу.

Если бы в описанной выше ситуации родительский процесс не ждал, пока дочерний завершится, а сразу продолжал работать, получилось бы, что оба процесса выполняются "параллельно": пока запущенный процесс что-то делает, пользователь продолжает командовать оболочкой. Для того, чтобы запустить процесс параллельно, в shell достаточно добавить "`&`" в конец командной строки:

```
[methody@localhost methody]$ sh loop&
 [1] 3634
[methody@localhost methody]$ ps -f
 UID        PID  PPID  C STIME TTY          TIME CMD
 methody   3590  1850  0 13:58 tty3     00:00:00 -bash
 methody   3634  3590 99 14:03 tty3     00:00:02 sh loop
 methody   3635  3590  0 14:03 tty3     00:00:00 ps -f
```

Запуск фонового процесса

В результате стартовый командный интерпретатор (PID `3590`) оказался отцом сразу двух процессов: `sh`, выполняющего сценарий `loop` и `ps`.

Процесс, запускаемый параллельно, называется **фоновым** (background). Фоновые процессы не имеют возможности *вводить* данные с того же терминала, что и породивший их shell (только из файла), зато *выводить* на это терминал могут (правда, когда на одном и том же терминале вперемежку появляются сообщения от нескольких фоновых процессов, начинается сущая неразбериха). При каждом терминале в каждый момент времени может быть не больше одного **активного** (foreground) процесса, которому разрешено с этого терминала вводить. На время, пока команда (например, `cat`) работает в активном режиме, породивший её командный интерпретатор "уходит в фон", и там, в фоне, выполняет свой `wait()`.

активный процесс	Процесс, имеющий возможность вводить данные с терминала. В каждый момент у каждого терминала может быть не более одного активного процесса.
фоновый процесс	Процесс, не имеющий возможность вводить данные с терминала. Пользователь может запустить любое, не превосходящее заранее заданного в системе, число фоновых процессов.

Стоит заметить, что параллельность работы процессов в Linux -- *дискретная*. Здесь и сейчас выполняться может столько процессов, сколько

центральных процессоров есть в компьютере (например, один). Дав этому одному процессу немного поработать, система запоминает всё, что тому для работы необходимо, приостанавливает его, и запускает *следующий* процесс, потом следующий и так далее. Возникает *очередь* процессов, ожидающих выполнения. Только что поработавший процесс помещается в конец этой очереди, а следующий выбирается из её начала. Когда очередь вновь доходит до того, первого процесса, система вспоминает необходимые для его выполнения данные (они называются **контекстом процесса**), и он продолжает работать, как ни в чём не бывало. Такая схема *разделения времени* между процессами носит названия **псевдопараллелизма**.

В выдаче `ps`, которую получил Мефодий, можно заметить, что PID стартовой оболочки равен `3590`, а PID запущенных из-под него команд (одной фоновой и одной активной) -- `3634` и `3635`. Это значит, что за время, прошедшее с момента входа Мефодия в систему до момента запуска `sh loop&`, в системе было запущено ещё `3634-3590=44` процесса. Что ж, в Linux могут одновременно работать несколько пользователей, да и самой системе иногда приходит в голову запустить какую-нибудь утилиту (например, выполняя действия по расписанию). А вот `sh` и `ps` получили *соседние* PID, значит, пока Мефодий нажимал `Enter` и набирал `ps -f`, никаких других процессов не запускалось.

В действительности далеко не всем процессам, зарегистрированным в системе, *на самом деле* необходимо давать поработать наравне с другими. Большинству процессов работать *прямо сейчас* не нужно: они ожидают какого-нибудь события, которое им нужно обработать. Чаще всего процессы ждут завершения операции ввода-вывода. Чтобы посмотреть, как потребляются ресурсы системы, можно использовать утилиту `top`. Но сначала Мефодий решил запустить *ещё один* бесконечный сценарий: ему было интересно, как два процесса конкурируют за ресурсы между собой:

```
[methody@localhost methody]$ bash loop&
 [2] 3639
[methody@localhost methody]$ top
  14:06:50  up  3:41,  5 users,  load average: 1,31, 0,76, 0,42
4 processes: 1 sleeping, 3 running, 0 zombie, 0 stopped
CPU states: 99,4% user,  0,5% system,  0,0% nice,  0,0% iowait,  0,0% idle
Mem:    514604k av,  310620k used,  203984k free,       0k shrd,   47996k buff
        117560k active,              148388k inactive
Swap: 1048280k av,       0k used, 1048280k free                   184340k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 3639 methody  20   0  1260 1260  1044 R    50,3  0,2   0:12 bash
 3634 methody  18   0   980  980   844 R    49,1  0,1   3:06 sh
 3641 methody   9   0  1060 1060   872 R     0,1  0,2   0:00 top
 3590 methody   9   0  1652 1652  1264 S     0,0  0,3   0:00 bash
```

Разделение времени между процессами

Оказалось, что дерутся даже не два процесса, а три: `sh` (первый из запущенных интерпретаторов `loop`), `bash` (второй) и сам `top`. Правда, по сведениям из поля `%CPU`, львиную долю процессорного времени отобрали `sh` и `bash` (они без устали вычисляют!), а `top` довольствуется десятой долей процента (а то и меньшей: ошибки округления). Стартовый `bash` вообще не

хочет работать, он *спит* (значение "S", Sleep, поля STAT, status): ждёт завершения активного процесса, top.

Увидев такое разнообразие информации, Мефодий кинулся читать руководство по top, однако скоро понял, что без знания *архитектуры* Linux большая её часть не имеет смысла. Впрочем, некоторая часть всё же понятна: объём оперативной памяти (всей, используемой и свободной), время работы машины, объём памяти, занимаемой процессами и т. п.

Последний процесс, запущенный из оболочки в фоне, можно из *этой* оболочки сделать активным при помощи команды fg ("foreground" -- "передний план").

```
[methody@localhost methody]$ fg
 bash loop
^C
```

Перевод фонового процесса в активное состояние с помощью команды fg (foreground)

Услужливый bash даже написал командную строку, какой был запущен этот процесс: "bash loop". Мефодий решил "убить" его с помощью управляющего символа "^C". Теперь *последним* запущенным в фоне процессом стал sh, выполняющий сценарий loop.

Сигналы

Чтобы завершить работу фонового процесса с помощью "^C", Мефодию пришлось сначала сделать его активным. Это не всегда возможно, и не всегда удобно. На самом деле, "^C" -- это не волшебная кнопка-убийца, а предварительно установленный символ (с ascii-кодом 3), при получении которого с терминала Linux передаст активному процессу **сигнал** 2 (по имени INT, от "interrupt" -- "прервать").

Сигнал -- это способность процессов обмениваться стандартными короткими сообщениями непосредственно с помощью системы. Сообщение-сигнал не содержит никакой информации, кроме номера сигнала (для удобства вместо номера можно использовать предопределённое системой имя). Для того, чтобы передать сигнал, процессу достаточно задействовать системный вызов kill(), а для того, чтобы *принять* сигнал, не нужно ничего. Если процессу нужно как-то по-особенному реагировать на сигнал, он может зарегистрировать *обработчик*, а если обработчика нет, за него отреагирует система. Как правило, это приводит к немедленному завершению процесса, получившего сигнал. Обработчик сигнала запускается *асинхронно*, немедленно после получения сигнала, что бы процесс в это время ни делал.

сигнал	Короткое сообщение, посылаемое системой или процессом другому процессу. Обрабатывается асинхронно специальной подпрограммой-обработчиком. Если процесс не обрабатывает сигнал самостоятельно, это делает система.

Два сигнала -- 9 (KILL) и 19 (STOP) -- всегда обрабатывает система. Первый из них нужен для того, чтобы убить процесс *наверняка* (отсюда и название). Сигнал STOP *приостанавливает* процесс: в таком состоянии процесс не удаляется из таблицы процессов, но и не выполняется до тех пор, пока не получит сигнал 18 (CONT) -- после чего продолжит работу. В Linux сигнал STOP можно передать активному процессу с помощью управляющего символа "^Z":

```
[methody@localhost methody]$ sh loop
^Z
 [1]+  Stopped                 sh loop
[methody@localhost methody]$ bg
 [1]+ sh loop &
[methody@localhost methody]$ fg
 sh loop
^C
[methody@localhost methody]$
```

Перевод процесса в фон с помощью "^Z" и bg

Мефодий сначала запустил вечный цикл в качестве активного процесса, затем передал ему сигнал STOP с помощью "^Z", после чего дал команду bg (back ground), запускающую в фоне последний остановленный процесс. Затем он снова перевёл этот процесс в активный режим, и, наконец, убил его.

Передавать сигналы из командной строки можно любым процессам с помощью команды `kill -сигнал PID` или просто `kill PID`, которая передаёт сигнал 15 (TERM).

```
[methody@localhost methody]$ sh
sh-2.05b$ sh loop & bash loop &
 [1] 3652
 [2] 3653
sh-2.05b$ ps -fH
 UID        PID  PPID  C STIME TTY          TIME CMD
 methody   3590  1850  0 13:58 tty3     00:00:00 -bash
 methody   3634  3590 87 14:03 tty3     00:14:18   sh loop
 methody   3651  3590  0 14:19 tty3     00:00:00   sh
 methody   3652  3651 34 14:19 tty3     00:00:01     sh loop
 methody   3653  3651 35 14:19 tty3     00:00:01     bash loop
 methody   3654  3651  0 14:19 tty3     00:00:00     ps -fH
```

Запуск множества фоновых процессов

Мефодий решил поназапускать процессов, а потом выборочно поубивать их. Для этого он, вдобавок к уже висящему в фоне `sh loop`, запустил в качестве активного процесса новый командный интерпретатор, `sh` (при этом изменилась **приглашение командной строки**). Из этого `sh` он запустил в фоне ещё один `sh loop` и новый `bash loop`. Сделал он это одной командной строкой (при этом команды *разделяются* символом "&", т. е. "И"; выходит так, что запускается **и** та, **и** другая команда). В `ps` он использовал новый ключ -- "-H" ("Hierarchy", "иерархия"), который добавляет в выдачу `ps` отступы, показывающие отношения "родитель--потомок" между процессами.

```
sh-2.05b$ kill 3634
 [1]+  Terminated              sh loop
sh-2.05b$ ps -fH
 UID        PID  PPID  C STIME TTY          TIME CMD
 methody   3590  1850  0 13:58 tty3     00:00:00 -bash
 methody   3651  3590  0 14:19 tty3     00:00:00   sh
 methody   3652  3651 34 14:19 tty3     00:01:10     sh loop
 methody   3653  3651 34 14:19 tty3     00:01:10     bash loop
 methody   3658  3651  0 14:23 tty3     00:00:00     ps -fH
```

Принудительное завершение процесса с помощью `kill`

Мефодий принялся убивать! Для начала он остановил работу давно запущенного `sh`, выполнявшего сценарий с вечным циклом (PID `3634`). Как видно из предыдущего примера, этот процесс за 16 минут работы системы съел не менее 14 минут процессорного времени, и конечно, ничего полезного не сделал. *Сигнал* о том, что процесс-потомок умер, дошёл до обработчика в стартовом `bash` (PID `3590`, и на терминал вывелось сообщение "`[1]+ Terminated sh loop`", после чего стартовый `bash` продолжил ждать завершения активного процесса -- `sh` (PID `3651`).

```
sh-2.05b$ exit
[methody@localhost methody]$ ps -fH
 UID        PID  PPID  C STIME TTY          TIME CMD
 methody   3590  1850  0 15:17 tty3     00:00:00 -bash
 methody   3663  3590  0 15:23 tty3     00:00:00   ps -fH
 methody   3652     1 42 15:22 tty3     00:00:38 bash loop
 methody   3653     1 42 15:22 tty3     00:00:40 sh loop
[methody@localhost methody]$ kill -HUP 3652 3653
[methody@localhost methody]$ ps
   PID TTY          TIME CMD
  3590 tty3     00:00:00 bash
  3664 tty3     00:00:00 ps
```

Завершение процесса естественным путём с помощью сигнала "Hang Up"

Ждать ему оставалось недолго. Этот `sh` завершился естественным путём, от команды `exit`, оставив после себя двух детей-сирот (PID `3652` и `3653`), которые тотчас же усыновил "отец всех процессов" -- `init` (PID `1`). Когда кровожадный Мефодий расправился и с ними -- с помощью сигнала 1 (`HUP`, то есть "Hang UP", "повесить"[1]) -- некому было даже сообщить об их кончине (если бы процесс-родитель был жив, на связанный с ним терминал вывелось бы что-нибудь вроде "`[1]+ Hangup sh loop`").

Доступ к файлу и каталогу

Довольно насилия. Пора Мефодию задуматься и о другой стороне работы с Linux: о правах и свободах. Для начала -- о свободах. Таблица процессов содержит список важнейших объектов системы -- процессов.

[1] Имя этого сигнала происходит не от казни через повешение, а от повешенной телефонной трубки

Однако не менее важны и объекты другого класса, те, что доступны в **файловой системе**: файлы, каталоги и специальные файлы (символьные ссылки, устройства и т. п.). По отношению к объектам файловой системы процессы выступают в роли *действующих субъектов*: именно процессы пользуются файлами, создают, удаляют и изменяют их. *Факт* использования файла процессом называется **доступом** к файлу, а *способ* воспользоваться файлом (каталогом, ссылкой и т. д.) -- *видом* доступа.

Чтение, запись и использование

Видов доступа в файловой системе Linux три. Доступ на **чтение** (read) разрешает получать информацию из объекта, доступ на **запись** (write) -- изменять информацию в объекте, а доступ на **использование** (execute) -- выполнить операцию, специфичную для данного типа объектов. Доступ к объекту можно изменить командой `chmod` (**ch**ange **mod**e, сменить режим (доступа)). В простых случаях формат этой команды таков: `chmod` *`доступ объект`*, где *`объект`* -- это имя файла, каталога и т. п., а *`доступ`* описывает вид доступа, который необходимо разрешить или запретить. Значение "`+r`" разрешает доступ к объекту на чтение (read), "`-r`" -- запрещает. Аналогично "`+w`", "`-w`", "`+x`" и "`-x`" разрешают и запрещают доступ на запись (write) и использование (execute).

Доступ к файлу

Доступ к *файлу* на чтение и запись -- довольно очевидные понятия:

```
[methody@localhost methody]$ date > tmpfile
[methody@localhost methody]$ cat tmpfile
 Срд Сен 22 14:52:03 MSD 2004
[methody@localhost methody]$ chmod -r tmpfile
[methody@localhost methody]$ cat tmpfile
 cat: tmpfile: Permission denied
[methody@localhost methody]$ date -u > tmpfile
[methody@localhost methody]$ chmod +r tmpfile; chmod -w tmpfile
[methody@localhost methody]$ cal > tmpfile
 -bash: tmpfile: Permission denied
[methody@localhost methody]$ cat tmpfile
 Срд Сен 22 10:52:35 UTC 2004
[methody@localhost methody]$ rm tmpfile
 rm: удалить защищённый от записи обычный файл `tmpfile'? y
```

Что можно и что нельзя делать с файлом, доступ к которому ограничен

Следует заметить, что Мефодию известна операция **перенаправления вывода** -- "`>`", с помощью которой он создаёт файлы в своём домашнем каталоге. Добавление "`> файл`" в командную строку приводит к тому, что всё, что вывелось бы на экран терминала[2], попадает в *`файл`*. Мефодий создаёт файл,

[2] Точнее, на **стандартный вывод** программы, такое перенаправление не касается **стандартного вывода ошибок**

проверяет, можно ли из него читать, командой `cat`, запрещает доступ на чтение и снова проверяет: на этот раз `cat` сообщает об отказе в доступе ("Permission denied"). Тем не менее *записать* в этот файл, перенаправив выдачу `date -u` оказывается возможным, потому что доступ на запись не закрыт. Если же закрыть доступ на запись, а доступ на чтение открыть (Мефодий сделал это в одной командной строке, разделив команды символом "`;`"), невозможным станет *изменение* этого файла: попытка перенаправить вывод программы `cal` будет неуспешной, а чтение снова заработает. Сработает и *удаление* этого файла, хотя `rm`, на всякий случай, предупредит о том, что файл защищён от записи.

Доступ к файлу на *использование* означает возможность запустить этот файл в качестве программы, выполнить его. Например, все файлы из каталога `/bin` (в том числе `/bin/ls`, `/bin/rm`, `/bin/cat`, `/bin/echo` и `/bin/date`) -- **исполняемые**, т. е. доступны на использование, и оттого их можно применять в командной строке в качестве команд. В общем случае необходимо указать **путь** к программе, например, `/bin/ls`, однако программы, находящиеся в каталогах, перечисленных в **переменной окружения** `PATH`, можно вызывать просто по имени: `ls` (подробнее о переменных окружения будет рассказано на следующих занятиях).

Сценарий

Исполняемые файлы в Linux бывают ровно двух видов. Первый -- это файлы в собственно исполняемом (executable) формате. Как правило, такие файлы -- результат *компиляции* программ, написанных на классических языках программирования, вроде Си. Попытка прочитать такой файл с помощью, например, `cat` не приведёт ни к чему полезному: на экран полезут разнообразные бессмысленные символы, в том числе -- управляющие. Это -- так называемые *машинные коды* -- язык, понятный только компьютеру. В Linux используется несколько форматов исполняемых файлов, состоящих из машинных кодов и служебной информации, необходимой операционной системе для запуска программы: согласно этой информации, ядро Linux выделяет для запускаемой программы оперативную память, загружает программу из файла и передаёт ей управление. Большинство утилит Linux -- программы именно такого, "двоичного" формата.

Второй вид исполняемых файлов -- **сценарии**. **Сценарий** -- это *текстовый* файл, предназначенный для обработки какой-нибудь утилитой. Чаще всего такая утилита -- это *интерпретатор* некоторого языка программирования, а содержимое такого файла -- программа на этом языке. Мефодий уже написал один сценарий для `sh`: бесконечно выполняющуюся программу `loop`. Поскольку к тому времени он ещё не знал, как пользоваться `chmod`, ему всякий раз приходилось *явно* указывать интерпретатор (`sh` или `bash`), а сценарий передавать ему в виде параметра.

сценарий	Исполняемый текстовый файл. Для выполнения сценария требуется программа-интерпретатор, путь к которой может быть указан в начале сценария в виде "`#!`*путь_к_интерпретатору*". Если интерпретатор не задан, им считается `/bin/sh`.

Если же сделать файл исполняемым, то ту же самую процедуру -- запуск интерпретатора и передачу ему сценария в качестве параметра командной строки -- будет выполнять система:

```
[methody@localhost methody]$ cat > script
 echo 'Hello, Methody!'
^D
[methody@localhost methody]$ ./script
 -bash: ./script: Permission denied
[methody@localhost methody]$ sh script
 Hello, Methody!
[methody@localhost methody]$ chmod +x script
[methody@localhost methody]$ ./script
 Hello, Methody!
```

Создание простейшего исполняемого сценария

С первого раза Мефодию не удалось запустить сценарий `script`, потому что по умолчанию файл создаётся доступным на запись и чтение, но не на использование. После `chmod +x` файл стал исполняемым. Поскольку домашний каталог Мефодия не входил в `PATH`, пришлось использовать **путь** до созданного сценария, благо путь оказался недлинным: "*текущий_каталог/имя_сценария*", т. е. `./script`.

Если системе не намекнуть специально, в качестве интерпретатора она запускает стандартный shell -- `/bin/sh`. Однако есть возможность написать сценарий для *любой* утилиты, в том числе и написанной самостоятельно. Для этого первыми двумя байтами сценария должны быть символы "`#!`", тогда всю его первую строку, начиная с третьего байта, система воспримет как *команду* обработки. Исполнение такого сценария приведёт к запуску указанной после "`#!`" команды, последним параметром которой добавится имя самого файла сценария.

```
[methody@localhost methody]$ cat > to.sort
 #!/bin/sort
 some
 unsorted
 lines
[methody@localhost methody]$ sort to.sort
 #!/bin/sort
 lines
 some
 unsorted
[methody@localhost methody]$ chmod +x to.sort
[methody@localhost methody]$ ./to.sort
 #!/bin/sort
 lines
 some
 unsorted
```

Создание sort-сценария

Утилита `sort` сортирует -- расставляет их в алфавитном, обратном алфавитном или другом порядке -- строки передаваемого ей файла. Совершенно то же самое произойдёт, если сделать этот файл исполняемым, вписав в начало `/bin/sort` в качестве интерпретатора, а затем выполнить получившийся сценарий: запустится утилита `sort`, а сценарий (файл с неотсортированными строками) передастся ей в качестве параметра. Оформлять файлы, которые необходимо отсортировать, в виде sort-сценариев довольно бессмысленно, просто Мефодий хотел ещё раз убедиться, что сценарий можно написать для чего угодно.

Доступ к каталогу

В отличие от файла, новый каталог создаётся (с помощью `mkdir`) доступным и на чтение, и не запись, и на использование. Суть всех трёх видов доступа к каталогу менее очевидна, чем суть доступа к файлу. Вкратце она такова: доступ по чтению -- это возможность *просмотреть содержимое* каталога (список файлов), доступ по записи -- это возможность *изменить содержимое* каталога, а доступ на использование -- возможность *воспользоваться* этим содержимым: во-первых, сделать этот каталог **текущим**, а во-вторых, *обратиться* за доступом к содержащемуся в нём файлу.

```
[methody@localhost methody]$ mkdir dir
[methody@localhost methody]$ date > dir/smallfile
[methody@localhost methody]$ /bin/ls dir
 smallfile
[methody@localhost methody]$ cd dir
[methody@localhost dir]$ pwd
 /home/methody/dir
[methody@localhost dir]$ cd
[methody@localhost methody]$ pwd
 /home/methody
[methody@localhost methody]$ cat dir/smallfile
 Срд Сен 22 13:15:20 MSD 2004
```

Доступ к каталогу на чтение и использование

Мефодий создал каталог `dir` и файл в `smallfile` в нём. При смене текущего каталога `bash` автоматически изменил строку-подсказку: последнее слово этой подсказки -- имя текущего каталога. Описанная в той же лекции команда `pwd` (**p**rint **w**ork **d**irectory) подтверждает это. Команда `cd` *без* параметров, как ей это и полагается, делает текущим **домашний каталог** пользователя.

```
[methody@localhost methody]$ chmod -x dir
```

```
[methody@localhost methody]$ ls dir
 ls: dir/smallfile: Permission denied
[methody@localhost methody]$ alias ls
 alias ls='ls --color=auto'
[methody@localhost methody]$ /bin/ls dir
 smallfile
[methody@localhost methody]$ cd dir
 -bash: cd: dir: Permission denied
[methody@localhost methody]$ cat dir/smallfile
 cat: dir/smallfile: Permission denied
```

Доступ к каталогу на чтение без использования

Мефодий запретил доступ на использование каталога, ожидая, что просмотреть его содержимое с помощью `ls` будет можно, а сделать его текущим и добыть содержимое находящегося в нём файла -- нельзя. Неожиданно команда `ls` выдала ошибку. Проницательный Мефодий заметил, что *имя файла* -- `smallfile`, команда `ls` всё-таки добыла, но ей зачем-то понадобился и сам файл. Гуревич посоветовал посмотреть, что выдаст команда `alias ls`. Оказывается, вместо простого `ls` умный `bash` подставляет специфичную для Linux команду `ls --color=auto`, которая раскрашивает имена файлов в разные цвета в зависимости от их типа. Для того, чтобы определить тип файла (например, запускаемый ли он) необходимо получить к нему доступ, а этого-то и нельзя сделать, когда нельзя *использовать* каталог. Если явно вызывать утилиту `ls` (`/bin/ls`), поведение каталога становится вполне предсказуемым.

```
[methody@localhost methody]$ chmod +x dir; chmod -r dir
[methody@localhost methody]$ /bin/ls dir
 /bin/ls: dir: Permission denied
[methody@localhost methody]$ cat dir/smallfile
 Срд Сен 22 13:15:20 MSD 2004
[methody@localhost methody]$ cd dir
[methody@localhost dir]$ /bin/ls
 ls: .: Permission denied
[methody@localhost dir]$ cd
[methody@localhost methody]$
```

Доступ к каталогу на использование без чтения

Если каталог, доступный на чтение, но недоступный на использование, мало когда нужен, то каталог, доступный на использование, но не на чтение, может пригодиться. Как видно из примера, получить список файлов, находящихся в таком каталоге, не удастся, но получить доступ к самим файлам, *зная* из имена -- можно. Мефодий тут же захотел положить в созданный каталог какой-нибудь нужный, но очень секретный файл, чтобы имя этого файла никто, кроме близких друзей не знал. Поразмыслив, Мефодий отказался от этой затеи: во-первых, не без оснований подозревая, что администратор (**суперпользователь**) всё равно сможет просмотреть содержимое такого каталога, а во-вторых, потому что нужного, но очень секретного файла под рукой не оказалось.

```
[methody@localhost methody]$ rm -rf dir
 rm: невозможно открыть каталог `dir': Permission denied
[methody@localhost methody]$ chmod -R +rwX dir
[methody@localhost methody]$ rm -rf dir
```

Рекурсивное удаление каталога

Потеряв интерес к секретным файлам, Мефодий решил удалить каталог `dir`. Из предыдущих занятий он знает, что это можно сделать не с помощью `rmdir`, а с помощью `rm` с ключом "`-r`" (recursive). Но сходу воспользоваться `rm` не удалось: чтение-то из каталога невозможно, и потому неизвестно, какие файлы там надо удалять. Поэтому сначала надо разрешить все виды доступа к `dir`. На всякий случай (а вдруг внутри `dir` попадётся такой же нечитаемый подкаталог?) Мефодий выполняет рекурсивный вариант `chmod` -- с ключом "`-R`" ("`R`" здесь большое, а не маленькое, потому что "`-r`" уже занято: означает запрет чтения). Команда `chmod -R +rwX dir` делает все файлы и каталоги в `dir` доступными на чтение и запись; и все каталоги -- доступными на использование. Параметр "`X`" (большое) устанавливает доступ на использование файла, только если этот доступ уже был разрешён *хоть кому нибудь* (хозяину, группе или всем остальным). Он позволяет не делать запускаемыми все файлы подряд... впрочем, кого это тревожит, если следующей командой будет `rm`?

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назначение команды ps
2. Понятие PID и PPID
3. Определение процесса
4. Понятие родительского и дочернего процесса
5. Определение активного и фонового процесса
6. Назначение утилиты top
7. Назначение команды fg
8. Назначение сигналов
9. Виды доступа в системе Linux
10. Назначение и формат команды chmod
11. Определение сценария
12. Пример создания сценария