нетология

Работа в терминале

лекция 2

Александр
Крылов

Александр Крылов

Lead DevOps services в ПАО СК Росгосстрах

Александр Крылов

План модуля

1. Работа в терминале, лекция 1
2. **Работа в терминале, лекция 2**
3. Операционные системы, лекция 1
4. Операционные системы, лекция 2
5. Файловые системы
6. Компьютерные сети, лекция 1
7. Компьютерные сети, лекция 2
8. Компьютерные сети, лекция 3
9. Элементы безопасности информационных систем

План занятия

1. Стандартные потоки ввода-вывода: stdin, stdout, stderr
2. Терминал TTY, эмулятор терминала TTY, псевдо-терминал PTY
3. Итоги
4. Домашнее задание

Стандартные потоки ввода-вывода: stdin, stdout, stderr

Как shell понимает, откуда ждать ввод и куда направлять вывод?

На прошлой лекции мы познакомились с несколькими командами, которые предполагают ввод...

```
rgersh@netology:~$ read new_var
Hello world!
```

... и вывод данных:

```
rgersh@netology:~$ echo $new_var
Hello world!
```

Когда вы работаете в терминале за **локальным** компьютером, может показаться само собой разумеющимся, что ввод и вывод в сессии происходит с физически подключенных устройств (*ваша* клавиатура, экран *вашего* компьютера).

Но всегда ли это так, и как именно оболочка принимает решение об обращении с **потоками данных**?

Потоки данных – одна из абстракций ОС

Использовать в программах прямой доступ к устройствам через драйвер – не универсально. Например, у *удаленного* сервера нет физической клавиатуры, но он как-то должен получить команды, переданные с *вашей* клавиатуры.

- **stdin** или **standard input** – стандартный ввод, условный номер **0**
  - ввод команды и ее аргументов в shell: `ls -l`, `man ls`
- **stdout** или **standard output** – стандартный вывод, условный номер **1**
  - вывод результата работы команды: листинг директорий `ls`, документация `man`
- **stderr** или **standard error** – стандартный вывод ошибок, условный номер **2**
  - вывод ошибок при работе команды: сообщение о вызове `ls` на несуществующую директорию, `man` на отсутствующую команду

Потоки могут не содержать данных:

- в случае успеха `cp` без дополнительных опций не сообщит об успешном копировании,
- не о чем будет сообщать и в потоке ошибок.

Процессы

Процесс – экземпляр запущенной программы.

Находясь в bash и вызывая внешние по отношению к оболочке программы (вспоминая первую лекцию – не *builtin* или не *keyword*, они как раз выполнятся внутри bash), мы порождаем новые процессы.

Хотя многие простые команды могут отработать и завершиться быстро, во время своего существования эти короткоживущие процессы также полноправны в системе как и, скажем, сам bash, или долгие ”тяжелые” процессы вроде виртуальных машин Java.

Нумерация файловых дескрипторов

Цифры 0, 1 и 2 называются **файловыми дескрипторами** (file descriptors, FD) и в общем случае стандартный *процесс* Linux содержит эти потоки данных по умолчанию.

Как увидеть данные потоки, если известны лишь их абстрактные номера?

Здесь нам поможет **lsof** (list open files), или список открытых файлов.

Отношение файлов, файловых дескрипторов и стандартных потоков

Вы спросите, причем здесь список открытых *файлов*, и почему вдруг *файловые дескрипторы*, если мы хотим посмотреть на *стандартные потоки* процесса?

Нередко встречается выражение “все в Linux является файлом”, и в данном случае эта фраза поможет ответить на заданный вопрос:

у потоков **std{in,out,err}** существует виртуальное *файловое* представление.

Открытие любых файлов приводит к выделению им файловых дескрипторов, все они получают номера.

Псевдо-файловые системы /dev и /proc

Мы уже рассматривали операции над реальными файлами, находящимися на существующей файловой системе: научились их просматривать `cat` или `less`, перемещать `mv`, удалять `rm` и т.д.

Кроме файловых систем, предназначенных для работы с физическими накопителями, в Linux существуют и **виртуальные псевдо-файловые системы**, две из которых мы сегодня затронем:

- **/dev** (файловое представление устройств)
- **/proc** (файловое представление структур ядра)

Read-only действия с псевдо-файловыми системами

Базовые операции с виртуальными служебными файловыми системами **не отличаются от реальных файлов и директорий**:

```
rgersh@netology:~$ ls /proc/up*
/proc/uptime
rgersh@netology:~$ cat /proc/uptime # время со включения в секундах, время в idle
2465.32 1911.94
```

Каталог /proc содержит множество сервисной информации, которая является актуальной на момент просмотра и постоянно изменяется.

Чтение из /dev на примере клавиатуры

Среди прочего в **/dev** представлены и физические устройства, такие как мышь и клавиатура. Это “виртуальные” файлы, представление которых реализуют драйверы. Почитаем из устройства клавиатуры:

```
evtest /dev/input/event2
```

Этот выходной поток данных может являться входным для ожидающей ввода программы.

Атрибуты процессов: PID

Чтобы воспользоваться **lsof** и исследовать потоки данных, нам необходимо знать базовые атрибуты процесса – ведь к нему, как минимум, надо как-то обратиться.

PID, Process IDentifier - числовой идентификатор процесса, однозначно его определяющий. Находясь в shell, мы можем узнать собственный PID через зарезервированную переменную **$:**

```
echo $$
```

Для задач: `jobs -l`.

Поиск по **Special Parameters** в **man bash** расскажет о других подобных переменных. Он же отвечает и на 9 вопрос домашнего задания прошлой лекции.

Атрибуты процессов: PPID

PPID, Parent PID - атрибут процесса, определяющий идентификатор его родительского процесса. Все процессы в Linux выстроены в дерево и должны иметь “родителя”. Так, для `sleep 1h` родителем будет bash, из которого sleep вызвали.
Для PPID в shell так же есть одноименная переменная: **`echo $PPID`**.

Воспользуйтесь **pstree -p**, если хотите посмотреть графическое представление всех существующих в ОС процессов в виде дерева.

Часть информации в **/proc** представлена с группировкой по **PID**: в **/proc/<PID>** находятся относящиеся к конкретному процессу данные.

lsof -p $$

Теперь мы обладаем достаточными знаниями, чтобы выполнить данную команду и понять ее вывод.

Запросим список открытых файлов для процесса с PID нашего shell:

```
rgersh@netology:~$ lsof -p $$
COMMAND  PID   USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
...
bash    2020 rgersh    0u   CHR  136,0      0t0      3 /dev/pts/0
bash    2020 rgersh    1u   CHR  136,0      0t0      3 /dev/pts/0
bash    2020 rgersh    2u   CHR  136,0      0t0      3 /dev/pts/0
```

Альтернативно:

```
rgersh@netology:~$ ls -l /proc/$$/fd/{0,1,2}
lrwx------ 1 rgersh rgersh 64 21:08 /proc/2020/fd/0 -> /dev/pts/0
lrwx------ 1 rgersh rgersh 64 21:08 /proc/2020/fd/1 -> /dev/pts/0
lrwx------ 1 rgersh rgersh 64 21:08 /proc/2020/fd/2 -> /dev/pts/0
```

Так как же shell понимает, откуда ждать ввод и куда направлять вывод?

Таким образом, ответ на изначальный вопрос раздела:

> shell и его потомки *понимают, откуда ждать ввод и куда направлять вывод, благодаря стандартным потокам,* которые в нашем случае указывают на некий **/dev/pts/X**.

В следующем разделе мы разберем, что это за виртуальное устройство.

Стандартные потоки – атрибут процесса

Представление **/dev** четко демонстрирует, что стандартные потоки являются атрибутом процесса. Используя **/dev/stderr**, можно сослаться только на свой собственный поток ошибок, **/dev/stderr** будет “свой” у каждого процесса.

```
rgersh@netology:~$ ls -l /dev/std* # self – ссылка на “самого себя”
lrwxrwxrwx 1 root root 15 21:07 /dev/stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root 15 21:07 /dev/stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root 15 21:07 /dev/stdout -> /proc/self/fd/1
```

Зато эти потоки можно перенаправлять:

```
rgersh@netology:~$ bash 2>/dev/null
lsof -p $$
COMMAND  PID   USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
…
bash    2568 rgersh    0u   CHR  136,0      0t0      3 /dev/pts/0
bash    2568 rgersh    1u   CHR  136,0      0t0      3 /dev/pts/0
bash    2568 rgersh    2w   CHR    1,3      0t0      6 /dev/null
```

> – перенаправление выходных потоков

Без **id** подразумевает **stdout**, т.е. `echo $$ > /tmp/bash_pid` эквивалентно `echo $$ 1> /tmp/bash_pid`

Частый пример использования: `>/tmp/log 2>&1`, что означает:

- *перенаправить stdout в файл /tmp/log,*
- *перенаправить stderr в stdout,*

то есть собрать в **/tmp/log** одновременно и **stdout**, и **stderr**.

>> – режим открытия файла.

Почему не будет работать `2>&1 >/tmp/log`?

Почему не будет работать sed 's/foo/bar/' file >file?

Порядок важен!

1. Если сначала перенаправить **stderr**, он будет указывать на текущее значение **stdout (/dev/pts/X)**, то есть для **stderr** ничего не изменится.

2. Все перенаправление устанавливается shell до начала выполнения программы

< – перенаправление входного потока

Перенаправление входного потока подразумевает **stdin**.

```
rgersh@netology:~$ echo DevOps@Netology > learning
rgersh@netology:~$ read new_var < learning
rgersh@netology:~$ echo $new_var
DevOps@Netology
```

Если попытаться выполнить некорректную операцию, например, записать что-то в stdout, ОС не даст этого сделать:

```
vagrant@netology1:~$ ls 1< /tmp/1
ls: write error: Bad file descriptor
```

| – pipe

Позволяет связать **stdout** одного процесса с **stdin** другого: `ls -R /usr | less`.

Можно делать цепочки из нескольких pipe.

Не применяйте pipe без необходимости.

Часто встречаются конструкции вида `cat some_file | grep some_string`, тогда как можно обойтись одиночной `grep some_string some_file`.

Наследование файловых дескрипторов

При создании новых процессов в Linux дочерний процесс автоматически получает от родительского определенный набор свойств.

В этот список входят и **открытые файловые дескрипторы**. *Именно поэтому процессы, которые мы создаем из bash, по-умолчанию используют те же самые стандартные потоки, что и оболочка:*

```
rgersh@netology:~$ sleep 1h & ls -l /proc/$!/fd
[1] 4859
total 0
lrwx------ 1 rgersh rgersh 64 апр 26 23:25 0 -> /dev/pts/0
lrwx------ 1 rgersh rgersh 64 апр 26 23:25 1 -> /dev/pts/0
lrwx------ 1 rgersh rgersh 64 апр 26 23:25 2 -> /dev/pts/0
```

Перенаправим вручную **stdout** на соседнюю вкладку:

```
rgersh@netology:~$ echo Follow the white rabbit >/dev/pts/1
```

Но что же все-таки это за устройство **/dev/pts/0,** и кто запустил **bash** со связанными с ним потоками?

Терминал TTY, эмулятор терминала TTY, псевдо-терминал PTY

Терминал TTY

Чтобы понять, что за устройство **/dev/pts/X**, нужно обратиться к истории.

Прародитель Linux (*nix, о котором будем говорить в следующей лекции) был изначально многопользовательской ОС, предполагавшей подключение множества устройств ввода-вывода к компьютеру.

О графическом интерфейсе тогда речи не шло, был доступен только текст, видеокарты в компьютере не существовало.

Устройство для передачи текстовых символов, подключающееся по последовательному порту к компьютеру через UART (Universal Asynchronous Receiver and Transmitter), – это и есть ***аппаратный терминал*, teletypewriter или TTY**.

взято с сайта: upweek.ru

Кен Томпсон *(сидит) и* ***Деннис Ритчи*** *за DEC PDP-11 – создатели Unix, благодаря которым мы знаем современный IT мир таким, как он есть.*

Терминал TTY

Пара интересных видео, показывающих устройства в действии:

- https://youtube.com/watch?v=WqgFK9h75eg
- https://youtube.com/watch?v=6zBvYs5Zej0

Оцените скорость работы :)

Терминал TTY

Данные с UART обрабатывались на уровне ядра операционной системы в модуле ***line discipline*** и ***драйвере TTY.***

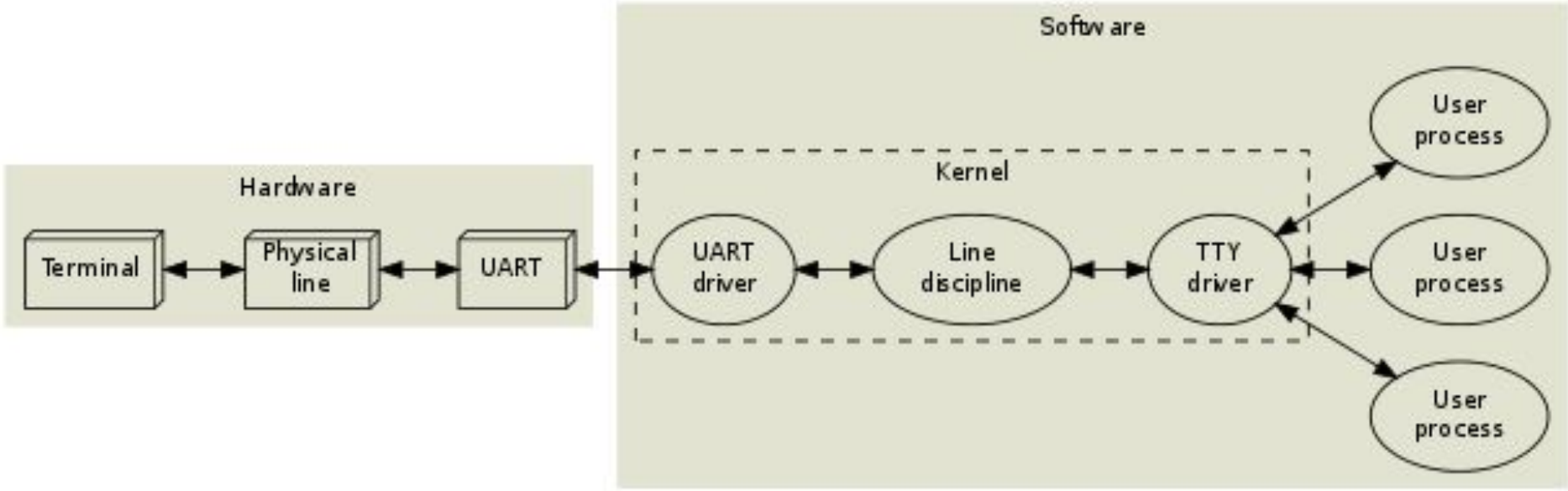
Они реализовывали:

- буферизацию вводимых символов,
- реакцию на специальные символы, такие как перевод строки/backspace или комбинации вроде Ctrl + C,
- вывод через UART переданных данных назад на TTY,
  чтобы пользователь сразу мог видеть вводимые им символы.

Терминал TTY

При нажатии Enter, буферизованные символы сбрасывались в **stdin** программе, которая выполнялась в этой сессии и находилась в активном режиме (**foreground**, вспомните **bg/fg** и **jobs** из предыдущей лекции). Много процессов, но активное взаимодействие только с одним.

Например, при начале работы с системой этой программой могла быть **login**, необходимая для аутентификации пользователя. Затем, как вы уже догадались, ей становился **shell**.



взято с сайта: linusakesson.net

Эмулятор терминала TTY

С развитием техники и физическим уменьшением компьютеров необходимость в аппаратных терминалах отпала, однако до появления современных интерфейсов оставалось еще много времени.

Следующим этапом в развитии стал **эмулятор терминала**, то есть ***программа*,** которая позволила пользоваться привычным окружением на оборудовании, в котором аппаратные терминалы уже отсутствовали.

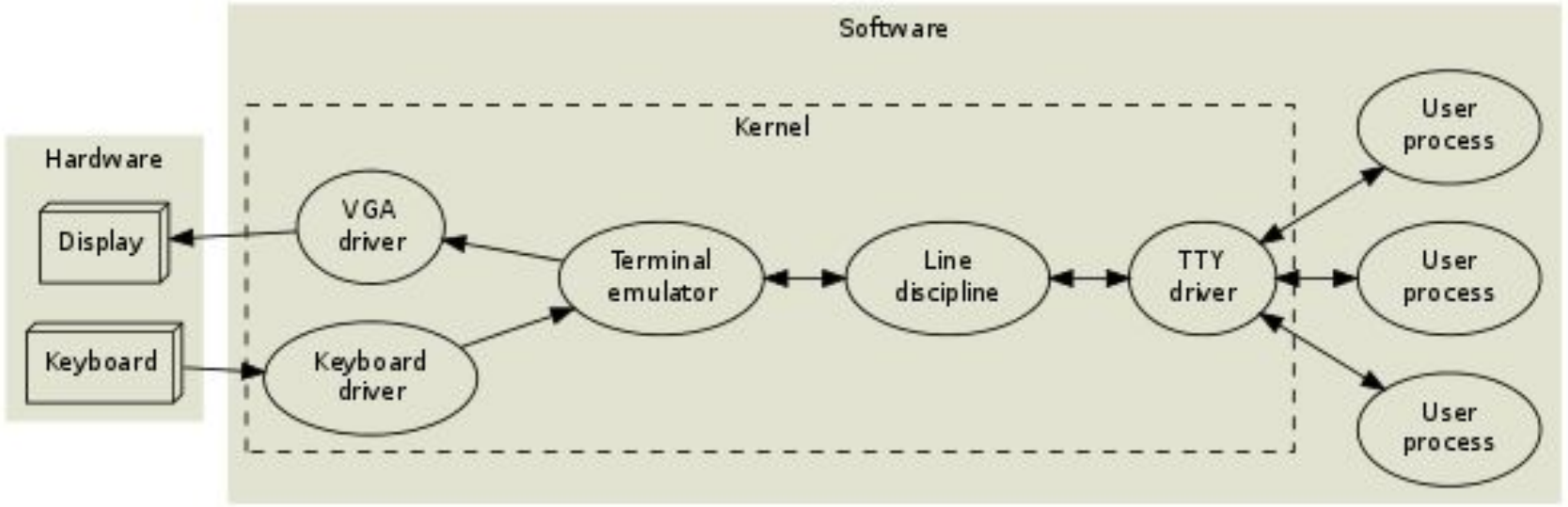
Прямой доступ к эмулятору терминала в Linux имеется до сих пор:

нажатие **Ctrl + Alt + F1..F6** позволяет запустить несколько независимых сессий, как будто к вашему компьютеру подключено несколько аппаратных терминалов.

Возврат в графический режим осуществляется через **Ctrl + Alt + F2**.

Эмулятор терминала TTY

В примере **stdin**, **stdout** и **stderr** shell будут установлены в **/dev/ttyX** вместо **/dev/pts/X** в этом режиме, так как используется драйвер TTY, а не PTY.



взято с сайта: linusakesson.net

Псевдо-терминал PTY

Важной особенностью эмулятора TTY является то, что он реализован в ядре ОС, а значит должен быть максимально универсален, не требователен к ресурсам и т.д.

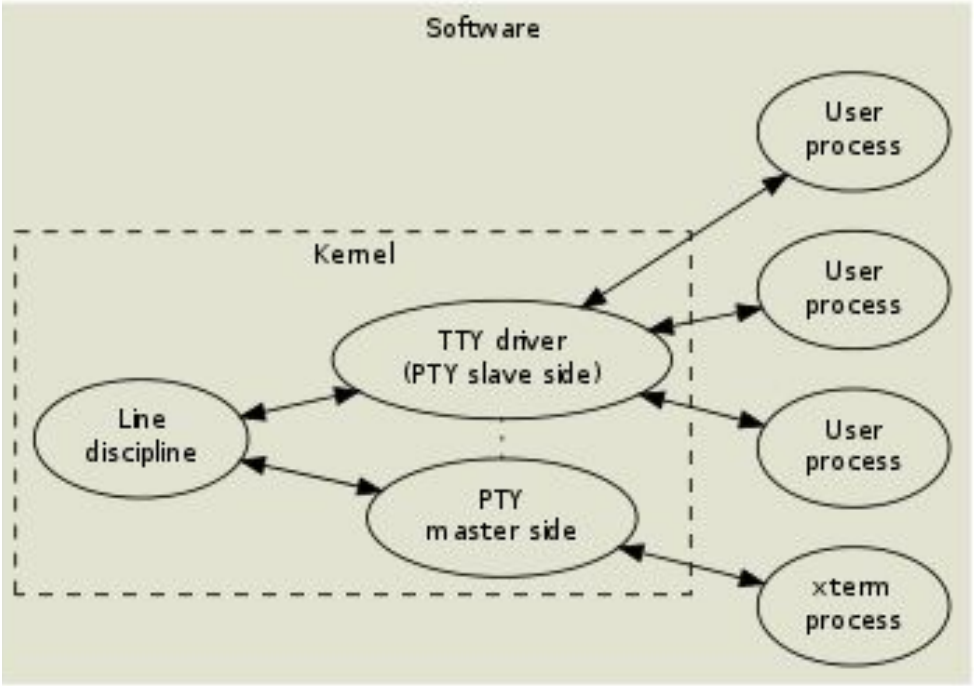
В современной реальности, когда доступно несравнимо больше аппаратных ресурсов, а в графическом интерфейсе пользователи хотят удобства, было принято решение для этих целей не расширять имеющийся эмулятор, а **создать новую концепцию псевдо-терминала** (**pseudo-terminal, PTY**).

Псевдо-терминал PTY

PTY – пара виртуальных устройств:

- **/dev/ptmx** для PTY-master
- **/dev/pts/X** для PTY-slave

Сами драйверы этих виртуальных устройств, как и драйвер TTY, работают на уровне ядра, однако если раньше и сам эмулятор терминала был частью ядра, теперь это пользовательское приложение, лишь взаимодействующее с мастером PTY.



взято с сайта: linusakesson.net

Псевдо-терминал PTY

Фактически, **PTY-slave** продолжает выполнять те же функции, которые выполнял ранее **драйвер TTY**, взаимодействуя с модулем **line discipline**.

Изменилась только схема доставки данных в и из него.

Если раньше они поставлялись от **аппаратного терминала** или от **ядерного модуля эмулятора терминала**, теперь ядро занимается доставкой данных между PTY slave и master.

Псевдо-терминал PTY

При открытии эмулятора в графическом интерфейсе происходит следующее:

1. **приложение делает вызовы к X Window (X11)** для отрисовки интерфейса, визуально схожего с оригинальным терминалом, но не ограниченного ресурсами ядра с любыми эффектами вроде цветов, прозрачности, тем, прокруткой, поддержкой системного буфера обмена и т.д.
2. **начинает "слушать" системные события от X Window**, такие как нажатия клавиатуры
3. **делает специальный вызов к /dev/ptmx**, создавая парную сессию /dev/pts/X
4. **события отправляются в мастер /dev/ptmx**, где вступает в дело line discipline
5. **полученные из /dev/ptmx в ответ данные выводятся в графический интерфейс**

То есть, само приложение терминала работает именно с ***PTY-master***:

```
rgersh@netology:~$ lsof -p $(pgrep gnome-terminal) | grep /dev/pt
gnome-ter 6802 rgersh   20u      CHR                5,2      0t0     88 /dev/ptmx
```

Псевдо-терминал PTY

Далее эмулятор терминала запускает дочерний процесс – shell того пользователя, под которым произведен вход в графическую оболочку систему.

Стандартные потоки ввода-вывода этого shell указывают на **/dev/pts/X**, что мы и видели ранее. Таким образом, прямой связи приложения терминала с std{in,out,err} shell нет.

Можно увидеть, что каждая вкладка в терминале запускает собственную сессию **/dev/pts** – **/dev/pts/0, /dev/pts/1** и т.д.

Мы отправляем команды в shell и видим результат их работы благодаря асинхронному процессу обмена данных между PTY-master и PTY-slave, которым занимается операционная система.

Следует понимать, что описанная концепция действительна только для современного Linux. Несмотря на внешнюю схожесть, вы не найдете /dev/ptmx в MacOS X, а /dev/ttyX в Android.

Псевдо-терминал PTY

Мы всегда можем узнать устройство, с которым ассоциирована наша сессия, и увидеть, кто обращался к мастеру PTY:

```
rgersh@netology:~$ tty
/dev/pts/1
rgersh@netology:~$ lsof /dev/ptmx
COMMAND     PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
gnome-ter 10425 rgersh   19u   CHR    5,2      0t0   88 /dev/ptmx
xterm     11767 rgersh    5u   CHR    5,2      0t0   88 /dev/ptmx
```

На этот раз **lsof** мы применили не к запущенному **процессу** (как делали для дескрипторов shell), а к **устройству**. Благодаря концепции дескрипторов, есть возможность использовать одно и то же устройство с разными приложениями:

драйвер PTY следит за тем, какой процесс сделал вызов к /dev/ptmx, поэтому для каждого процесса содержимое по одному и тому же пути будет разным (как со стандартными потоками).

Удаленное подключение

Прелесть абстракций стандартных потоков ввода-вывода дает большую гибкость.

Так, рассмотренный пример, в котором shell запускается локальной программой эмулятора терминала, является частным.

Эта программа может быть заменена другой. Например, той, что получает ввод и вывод не от локального оконного сервера, а через защищенный канал от удаленного клиента. Именно так работает **SSH (secure shell) сервер**.

В этой ситуации именно ssh сервер делает вызов к **/dev/ptmx** на удаленном сервере, а затем ассоциирует с сессией новый **/dev/pts/X**.

Так как по умолчанию удаленный пользователь не является аутентифицированным, ssh сервер не запускает сразу shell (так как в этом случае любой мог бы получить доступ), а сначала вызовет login.

Итоги

Сегодня мы узнали о:

- процессах и некоторых их атрибутах,
- о предоставляемых операционной системой абстракциях на примере стандартных потоков ввода-вывода,
- о служебных файловых системах /dev и /proc,
- о том, как посмотреть открытые процессом файлы с помощью lsof,
- о развитии и типах терминалов.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Александр Крылов

Александр Крылов