нетология

Операционные системы

лекция 1

Александр
Крылов

Александр Крылов

Lead DevOps services в ПАО СК Росгосстрах

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. POSIX и *nix
2. Задачи ОС
3. Системные вызовы, библиотечные вызовы, strace, eBPF
4. Процессы и треды
5. Итоги
6. Домашнее задание

POSIX и *nix

POSIX-совместимая Unix-подобная ОС Linux

Bell Labs и **AT&T Research** создали оригинальную **ОС UNIX®** в 1970-х годах.

ОС UNIX® сделала популярными:

- иерархические файловые системы с директориями глубокой вложенности;
- единообразное представление устройств для пользователя на уровне файловой системы;
- интерпретатор с возможностью использования одних и тех же команд в интерактивном режиме и скриптах;
- регулярные выражения и многое другое.

POSIX-совместимая Unix-подобная ОС Linux

Изначальная модель распространения UNIX® привела **к появлению множества не полностью совместимых** между собой ответвлений этой ОС:

- BSD;
- Solaris;
- AIX;
- и других.

Непереносимость кода между этими родственных ОС показала **необходимость создания стандарта**. Такой стандарт был разработан и получил название: **POSIX** – **P**ortable (переносимый) **O**perating **S**ystem **I**nterface for Uni**x**.

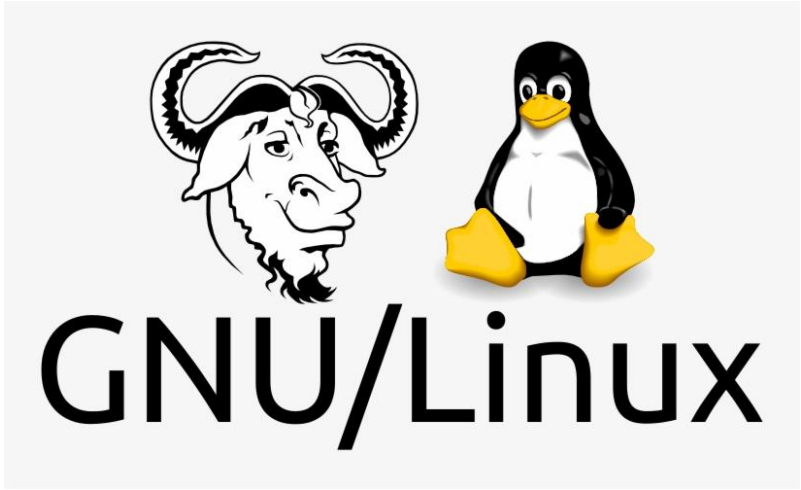
Кстати, утверждением занималась рабочая групп института IEEE, который и по сей день отвечает за более близкие нам стандарты как 802.11a/b/g/n/ac – WiFi.

Развитие *nix

В настоящее время ни одна ОС, включая последнюю из UNIX® – System V, не имеет общего исходного кода с оригинальным ПО 70-ых годов.

Однако следование концепции дизайна, заданной общим предком, позволяет называть многие системы Unix-подобными, а реализация стандарта POSIX приводит к их схожести с точки зрения пользователя и разработчика.

macOS Big Sur является полностью сертифицированной UNIX®-совместимой ОС, а не просто Unix-like :)



GNU/Linux (от GNU is not Unix, мощнейший проект по реализации свободной Unix-like ОС с GPL лицензией) на сегодняшний день является стандартом де-факто в серверном сегменте.

Илюстрации взяты с сайтов: pngkit.com, apple.com

Linux vs. BSD

Различные варианты BSD не утратили своей актуальности. Их широко используют такие гиганты, как Netflix или WhatsApp.

Однако подавляющее большинство интернет-сервисов базируются на одном из многообразных дистрибутивов Linux.

Мы не будем уделять времени сравнению разных дистрибутивов, а **сконцентрируемся на общих для дистрибутивов основах**.

Задачи ОС

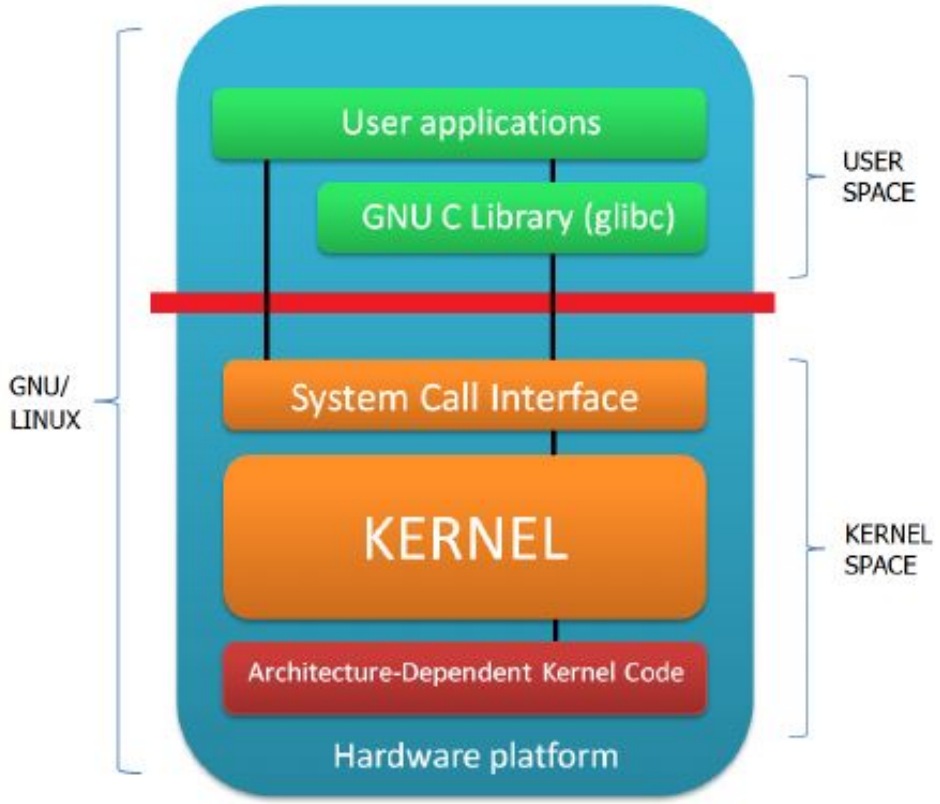
Базовые задачи ОС

- **Организация процесса загрузки, предоставление runtime** для приложений в виде системных библиотек, интерпретируемых языков, виртуальных машин и т.д. после первоначальной инициализации оборудования.
- **Интерфейс работы с оборудованием** – программы не хотят знать об особенностях доступа к жесткому диску или сетевому контроллеру, они хотят просто записать данные локально или отправить по сети.
- **Виртуализация и планирование использования ресурсов** – программы не только не хотят знать о физических особенностях доступа к ресурсу, но и думать о том, что за этот ресурс могут одновременно конкурировать другие запущенные приложения.
- **Передача данных локально между приложениями**.
- **Поддержка многопользовательского режима работы** - поддержка большого количества терминалов и соответствующее разграничение прав доступа к ресурсам.

Дополнительные задачи ОС

- **Адаптация к широкому спектру оборудования и возможность расширения функциональности**.
- **Контроль над программами** – принудительное завершение сбойных приложений, при наличии выделенных ресурсов – слежение за их расходом.
- **Виртуализация устройств** – возможность работы не только с физическим оборудованием хоста, но и предоставление интерфейса для создания псевдо-устройств.

ОС с точки зрения системного администратора



- Разделение на пользовательскую и ядерную части (юзер-спейс и кернел-спейс).
- Такое разделение является реальным - в Linux есть область памяти, доступная ядру (+ модули ядра, драйверы), работающему с ЦП в привилегированном режиме, и пользовательская часть памяти, которая делится между прикладными программами.
- Управление памятью, планирование (процессы, ввод-вывод локальный и сетевой), большинство файловых систем, доступ к оборудованию работает на уровне ядра и недоступно приложениям из юзер-спейса.

взято с сайта: researchgate.net

Системные вызовы, библиотечные вызовы, strace, eBPF

Системные вызовы – метод общения приложений и ОС

Описанное ранее **на практике реализовано механизмом системных вызовов** – это интерфейс взаимодействия пользовательских приложений и ядра.

Напрямую системные вызовы используются редко, а формат обращения к ним сложен для прикладного программирования. Над системными вызовами написаны библиотеки, которые удобнее для прямого вызова программистами.

В Linux подобной стандартной библиотекой является **glibc (GNU Lib C)**. Зависимость от общей библиотеки libc присутствует практически во всех местах операционной системы от простейших приложений:

```
vagrant@netology1:~$ ldd $(which ls) | grep libc
     libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9ab2e5f000)
```

и до сложных интерпретаторов:

```
vagrant@netology1:~$ ldd $(which python3) | grep libc
     libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3f75d6a000)
```

glibc – не единственная стандартная библиотека C. Например, в популярном в контейнерах Alpine Linux вместо нее musl libc – легковесная альтернатива для компактных сборок.

Приложение > библиотечный вызов > системный вызов

Пример CPython:

1. функция print языка Python делает **библиотечный вызов libc write**:

```
vagrant@netology1:~$ sotruss python3 -c 'print("Hi, Netology!")' 2>&1 | grep write
        python3 -> libc.so.6      :*write(0x1, 0xb96880, 0xe)
```

2. библиотечный вызов libc, в свою очередь, делает **системный вызов write**:

```
vagrant@netology1:~$ strace -e trace=write python3 -c 'print("Hi, Netology!")' >
/dev/null
write(1, "Hi, Netology!\n", 14)         = 14
+++ exited with 0 +++
```

Скрипт для отслеживания библиотечных вызовов **sotruss из комплекта libc**:

```
vagrant@netology1:~$ dpkg -S $(which sotruss)
libc-dev-bin: /usr/bin/sotruss
```

strace – программа, использующая подсистему ядра ptrace для отслеживания системных вызовов. Основной недостаток strace – замедление работы приложений, в десятки или даже сотни раз, поэтому в production среде применяйте аккуратно.

strace на примерах, write

В ядре на сегодня более 380 системных вызовов. Практически все сервисы, которые предоставляет приложениям ОС, – системные вызовы:

- **установить сетевое соединение** (создать сокет и подключить его к удаленному серверу);
- **отправить или получить данные сетевого соединения**;
- **открыть, прочитать или записать файл** (который может располагаться на локальной или удаленной файловой системе любого типа);
- **создать новый процесс** (фактически, запустить приложение);
- **выделить/освободить память** (malloc/free) и т.д.

strace на примерах, write

Информация по системным вызовам доступна в man 2 после установки пакета `manpages-dev`. Еще раз посмотрим на write:

```
write(1, "Hi, Netology!\n", 14)         = 14
```

man 2 write расскажет, что конкретно мы видим:

```
write() writes up to count bytes from the buffer starting at buf to the
file referred to by the file descriptor fd.
On success, the number of bytes written is returned.
```

stdout – стандартный вывод с файловым дескриптором номер **1**.

strace на примерах, open

Запишем и откроем файл:

```
root@netology1:~# echo -n 'Hello Netology!' > /tmp/netology
root@netology1:~# cat read_something.py
#!/usr/bin/env python3
with open('/tmp/netology', 'r') as tmp_file:
    print(tmp_file.read())
```

Считываем содержимое из записанного ранее файла, а затем выводим его:

```
openat(AT_FDCWD, "/tmp/netology", O_RDONLY|O_CLOEXEC) = 3
read(3, "Hello Netology!", 16)          = 15
close(3)                                = 0
write(1, "Hello Netology!\n", 16)       = 16
```

openat присваивает 3 номер fd открываемому файлу /tmp/netology, read прочитал 15 байт, close закрыл 3 дескриптор, write написал содержимое в stdout как в прошлом примере.

strace, полезные опции

- Часто strace необходимо запустить на уже работающие программы, чтобы понять, почему они не ведут себя должным образом. Для этого есть флаг **-p**:

```
root@netology1:~# strace -p $(pgrep -f 'nginx: worker')
strace: Process 25294 attached
epoll_wait(9, [{EPOLLIN, {u32=4108079352, u64=140097351340280}}], 512, -1) =
1
accept4(7, {sa_family=AF_INET6, sin6_port=htons(37288),
sin6_flowinfo=htonl(0), inet_pton(AF_INET6, "::1", &sin6_addr)...
```

- Программы при работе часто порождают потомков, то есть новые процессы. Чтобы подключившись с -p к процессу и включить при этом трейсинг его потомков, добавьте **-f**.

```
root@netology1:~# strace sh -c date
vs.
root@netology1:~# strace -f sh -c date
```

strace, полезные опции

- Бывает, что в системных вызовах вы обнаружите важную для отладки информацию, которая отсутствует в логах.

- По умолчанию вывод строк strace ограничен 32 байтам. Используйте **-s** с требуемой длиной строки, чтобы увидеть полные сообщения, например **-s 65000**. В таком случае может пригодиться и -o для записи в файл вместо вывода на экран.

- **-y** – аннотации к файловым дескрипторам.

- Очень часть системным администраторам помогает **strace -e open**: показывает чтение конфигурационных файлов из разных мест, недоступные ресурсы и т.д.

Используйте strace!

Упомянутые утилиты помогут вам в разных реальных ситуациях не раз и полезны для обучения. Большинство вещей, которые мы упоминаем в теории, через strace можно увидеть вживую.

Ранее мы упоминали **$PATH**:

```
vagrant@netology1:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:...
```

Запустите bash под strace и увидите, как оболочка действительно перебирает эти директории при поиске исполняемого файла:

```
stat("/usr/local/sbin/bash", 0x7fff921ea4e0) = -1 ENOENT (No such file or
directory)
stat("/usr/local/bin/bash", 0x7fff921ea4e0) = -1 ENOENT (No such file or
directory)
stat("/usr/sbin/bash", 0x7fff921ea4e0)  = -1 ENOENT (No such file or directory)
stat("/usr/bin/bash", {st_mode=S_IFREG|0755, st_size=1183448, ...}) = 0
```

eBPF

eBPF - еще один инструмент (не использует ptrace), позволяющий отследить, что происходит в ОС.

Фактически, это целый язык программирования, который генерирует микропрограммы для исполнения ядром и может манипулировать или следить за различными подсистемами.

eBPF не ограничен привязкой к системным или библиотечным вызовам, а также не влияет на производительность. Рекомендуем ознакомиться с полезным набором утилит iovisor BCC. Изучите список готовых eBPF программ – многие вам понадобятся.

Например, можно посмотреть **все новые процессы, появляющиеся в системе:**

```
root@netology1:~# execsnoop-bpfcc
PCOMM            PID    PPID   RET ARGS
systemctl        174640 24777    0 /usr/bin/systemctl status nginx
systemctl        174641 24777    0 /usr/bin/systemctl status nginx
```

Процессы и треды

Почему в примере с *strace open* когда мы открывали в Python файл с файловой системы, был виден вызов open с флагом r для чтения, однако не было подобного open для записи в stdout?

fork/clone, наследование

На прошлой лекции, рассматривая файловые дескрипторы, мы узнали, что они наследуются новым процессом от своего родителя.

Иными словами, **запущенный из bash Python пишет на тот же стандартный выход, что был назначен для bash и открыт им же**.

fork/clone, наследование

Как вы могли догадаться, в реальности за этот механизм тоже отвечают системные вызовы. В частности, семейства вызовов **fork** (вилка) и **clone**.

Из man strace про ключ -f:

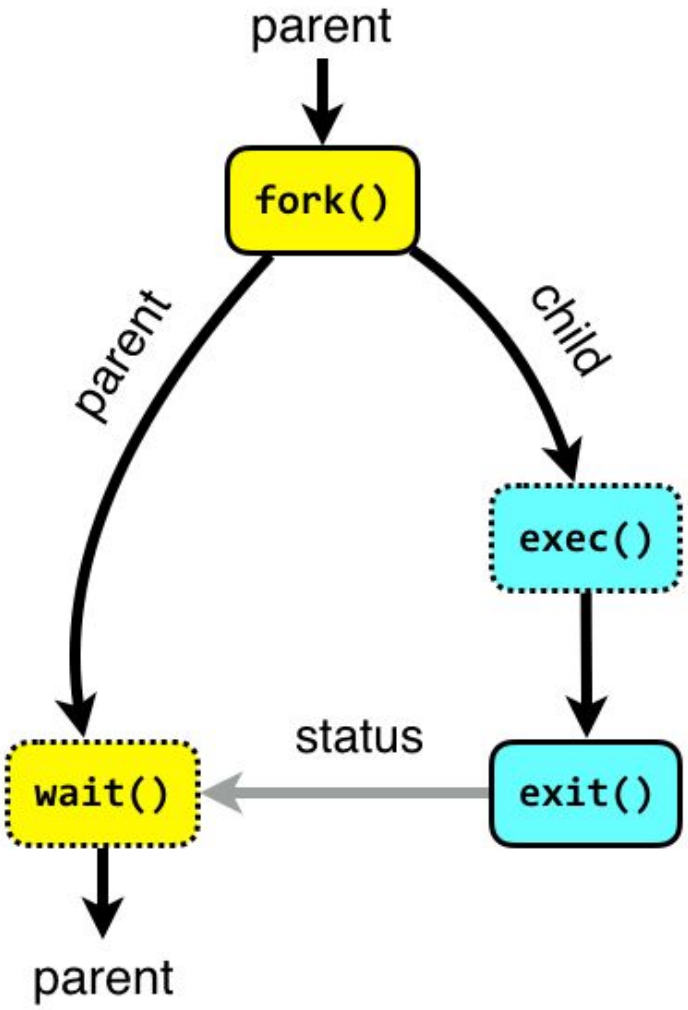
*Trace child processes as they are created by currently traced processes as a result of the **fork**(2), **vfork**(2) and **clone**(2) system calls.*

man 2 fork:

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. At the time of fork() both memory spaces have the same content.

При запуске нового процесса первым действием ОС дублирует тот процесс, который инициировал fork.

wait родительского процесса



После того как родитель вызвал fork, в памяти находится 2 идентичных процесса в одной точке исполнения. Разница между ними – код возврата системного вызова, по которому процесс должен понять, является ли он родителем (получает число – PID для потомка), или новым дочерним процессом (получает 0 из fork).

Родительский процесс после вызова fork/clone должен сделать wait (waitpid) на PID нового процесса, дождавшись его завершения.

В случае, когда по какой-либо причине родительский процесс не смог обработать код возврата от дочернего процесса, такой дочерний процесс становится ***зомби*** и, завершившись, остается висеть строкой в таблице процессов ядра.

Процессы ***сироты*** тоже существует – те, родительский процесс которых был завершен до того как дочерний отработал.

взято с сайта: it.uu.se

exec – замещение клонированного процесса

Резонный вопрос – зачем нам в памяти 2 экземпляра одного процесса. Поэтому третий пункт при запуске нового процесса – вызов exec (execve):

```
vagrant@netology1:~$ strace -f sh -c ls
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7fae4827b850) = 161515
wait4(-1, strace: Process 161515 attached
 <unfinished ...>
[pid 161515] execve("/usr/bin/ls", ["ls"], 0x55bb2b539b48 /* 29 vars */) = 0
…
[pid 161515] +++ exited with 0 +++
<... wait4 resumed>[{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 161515
```

При **вызове fork физического копирования данных в памяти не происходит** – благодаря механизму виртуальной памяти ядро просто назначает одни и те же участки разным процессам, а copy-on-write уже записывает при необходимости только изменяющиеся данные для нового процесса.

Код возврата

Мы посмотрели как создаются новые процессы, но не менее важно и то, как они завершают свою работу. Штатный способ завершения – системный вызов exit (exit_group). Получив его, ОС понимает, что нужно освободить все ресурсы, с которыми процесс работал. **Возвращаемый exit code – важный признак, по которому можно сделать вывод об успешности завершенного процесса**.

Код возврата **0** сигнализирует об успешности. Отличные от 0 значения зависят от приложения, но так или иначе они указывают, что процесс не отработал штатно.

Код возврата собирает с дочерних процессов wait. В bash узнать exit code можно через служебную переменную **?** ($?).
Например, ls удалось успешно сделать листинг директорий в /tmp:

```
vagrant@netology1:~$ ls /tmp > /dev/null; echo $?
0
```

А вот доступа в домашний каталог /root у обычного пользователя нет:

```
vagrant@netology1:~$ ls /root; echo $?
ls: cannot open directory '/root': Permission denied
2
```

Особенно полезен exit code в сценариях для обработки ошибок.

Состояния процессов

Процессы могут находиться в разных состояниях: одни активно исполняются, какие-то могли стать зомби, так как не был сделан wait, какие-то могут ожидать выполнения долгой задачи дочернего процесса.

Некоторые значимые состояния:

D - uninterruptible sleep (*непрерываемый* сон, обычно во время IO операций);

R - running и runnable (исполняется или ожидает исполнения);

S - interruptible sleep (обычный спящий процесс, который *может быть прерван*, ожидает какого-то события);

T - остановлен сигналом управления задачами;

Z - упомянутый ранее зомби-процесс.

Состояния процессов

uptime – команда, показывающая load average за последние 1, 5 и 15 минут.

Load average (LA) – среднее число процессов в состояниях **R** и **D** за рассчитываемый промежуток времени. Обратите внимание, что в Linux **D**-state влияет на LA. Нередко можно встретить утверждение, что LA - это нагрузка на процессор, но это не соответствует действительности, так как медленное IO под диску также будет влиять на LA напрямую.

LA не нормировано по ядрам. То есть, LA 12 для 12-ядерного хоста – показатель, что система нагружена слабо.

```
root@service-pod-6c48b4f965-v4s46:/# uptime
 14:17:13 up 21 days, 4 min,  0 users,  load average: 1.45, 0.92, 0.95
```

Просмотр таблицы процессов

ps – process state.

По умолчанию показывает “собственные” процессы пользователя в том терминале, из которого **ps** запущен. Имеет множество ключей, их можно сочетать.

Например, **ps aux**, где:

a – убирает ограничение о собственных процессах,
u – добавляет расширенный набор часто нужных колонок,
x – убирает ограничение о процессах, запущенных из текущего терминала,
w/ww – убирает ограничение по длине вывода.

```
root@netology1:~# ps aux | head -n3
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  1.2 170644 12612 ?        Ss   Jul05   6:51 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Jul05   0:00 [kthreadd]
```

Просмотр таблицы процессов

Посмотреть процессы конкретного пользователя:

```
root@netology1:~# ps -u www-data u
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
www-data   25294  0.0  0.3  57868  3956 ?        S    Jul05   0:00 nginx:
worker
```

Обратите внимание, *в ps многие суффиксы с минусом и без различаются по смыслу*. **Расширенный вывод с дополнительными колонками**:

```
root@netology1:~# ps -p 25294 -o pid,user,comm,etime,%mem,%cpu
    PID USER     COMMAND             ELAPSED %MEM %CPU
  25294 www-data nginx            2-19:29:30  0.3  0.0
```

Можно **добавить сортировку:**

```
root@netology1:~# ps -o pid,user,comm,etime,%mem,%cpu --sort=-%mem
```

Управление процессами, сигналы

Одна из форм межпроцессного взаимодействия – сигналы.

- Процессу можно послать специальный код, который он может обработать.
  Нажатия Ctrl + Z или Ctrl + C отправляют сигналы TSTP (temporary stop) и INT (interrupt) соответственно.
- **Команда kill** - другой способ отправить сигнал:

  ```
  root@netology1:~# kill -HUP $(cat /var/run/nginx.pid)
  ```

  Нередко HUP командует процессу перечитать свои конфигурационные файлы без перезапуска. Несмотря на свое название, kill может послать любой сигнал, но по умолчанию применяет **сигнал TERM** (terminate).
- Сигналы могут быть поданы **по имени и по номеру** – например, знаменитый **kill dash nine (-9)**: принудительное завершение на случай невозможности выхода программы по штатному сигналу -15.
- **SIGKILL** и **SIGSTOP** – два сигнала, которые не могут быть перехвачены процессом и обработаны как-то по желанию разработчика (на все остальные сигналы можно повесить подобные обработчики).

Треды (потоки/нити) исполнения

Существует много сценариев, когда распараллеливание обработки данных является эффективным. **Вызов fork** является относительно недорогой операцией, но архитектура программы, подразумевающая несколько процессов, может быть не всегда оптимальна. Такая архитектура – не редкость. Один из самых популярных веб-серверов **nginx** использует модель с несколькими процессами.

Если разработчик не хочет тратить время на межпроцессное взаимодействие для синхронизации работы, желает использовать более простую модель доступа к общим данным в памяти, ОС предоставляет другой механизм - тредов, то есть нескольких потоков исполнения внутри одного процесса.

Основное различие fork и clone – в возможности clone создавать треды. С точки зрения планирования процессорного времени для ОС процессы и потоки идентичны, поэтому может оказаться полезно знать, что какой-то единичный процесс может внутри плодить множество тредов и сильно нагружать CPU (один PID, разные thread id):

```
root@netology1:~# ps aux | grep mongo[db]
mongodb   220309  1.2  7.5 977304 76112 ?        Ssl  11:20   0:01 /usr/bin/mongod
--unixSocketPrefix=/run/mongodb --config /etc/mongodb.conf
root@netology1:~# ps aux -T | grep mongodb
mongodb   220309       1  220309  0   23 11:20 ?        00:00:00 /usr/bin/mongod
--unixSocketPrefix=/run/mongodb --config /etc/mongodb.conf
mongodb   220309       1  220318  0   23 11:20 ?        00:00:00 /usr/bin/mongod
--unixSocketPrefix=/run/mongodb --config /etc/mongodb.conf
...
```

Атрибуты процессов

Мы познакомились с немалым числом атрибутов процесса:

- PID, PPID;
- файловые дескрипторы;
- переменные окружения (также наследуются от родителя);
- обработчик сигналов (системный по-умолчанию или собственный);
- область данных в памяти (мы опускаем подробности в данной лекции, но для корректности: stack, heap, data, text);
- один или более тредов.

Дополнительные атрибуты процессов

Какие еще есть важные атрибуты, которые мы пока не затронули?

- принадлежность пользователю и группе
- параметры запуска:

```
root@netology1:~# cat /proc/219695/cmdline
nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
```

- принадлежность пространству имен (рассмотрим чуть позже)
- capabilities (расширение концепции привилегированных/непривилегированных процессов)

```
root@netology1:~# getcap $(which ping)
/usr/bin/ping = cap_net_raw+ep
```

Файловые дескрипторы, удаленные файлы

На прошлой лекции мы познакомились с файловыми дескрипторами. Одно из свойств, полезных системным администраторам - возможность взаимодействия с открытыми процессом файлами с помощью fd, даже если файл был удален с файловой системы:

```
vagrant@netology1:~$ python3 -c "import time;f=open('/tmp/do_not_delete_me',
'r');time.sleep(600);" &
[1] 181248
vagrant@netology1:~$ lsof -p 181248 | grep do_not_delete_me
python3 181248 vagrant    3r   REG  253,0  14 1572876 /tmp/do_not_delete_me
vagrant@netology1:~$ cat /tmp/do_not_delete_me
valuable_data
vagrant@netology1:~$ rm /tmp/do_not_delete_me
vagrant@netology1:~$ cat /tmp/do_not_delete_me
cat: /tmp/do_not_delete_me: No such file or directory
vagrant@netology1:~$ lsof -p 181248 | grep do_not_delete_me
python3 181248 vagrant    3r   REG  253,0  14 1572876 /tmp/do_not_delete_me (deleted)
vagrant@netology1:~$ cat /proc/181248/fd/3 > /tmp/do_not_delete_me
vagrant@netology1:~$ cat /tmp/do_not_delete_me
valuable_data
```

Будьте осторожны. Если вы удаляете что-то, не зная об использовании файла процессом, то, несмотря на кажущееся отсутствие на файловой системе, файл продолжит занимать место на ней. Нередко такое происходит с логами при некорректно настроенном внешнем процессе ротации, сигнал USR1 может помочь в этом случае (отпустить удаленные файлы) и пересоздать актуальные.

Итоги

- Ознакомились с краткой историей развития Unix и POSIX систем.
- Рассмотрели основные функции ОС в целом и затронули каждую из них на примере Linux.
- Познакомились с инструментами отладки: strace для системных вызовов, sotruss для библиотечных, eBPF на примере набора утилит BCC.
- Узнали, как создаются и завершаются процессы.
- Узнали о состояниях процессов.
- Узнали, как можно отправить процессам сигналы.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера .
- 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