# Урок 7. Работа в Linux

1. Что делает системный вызов fork()?
2. Что такое процесс зомби?
3. Что такое процесс сирота?
4. Как убить процесс зомби?

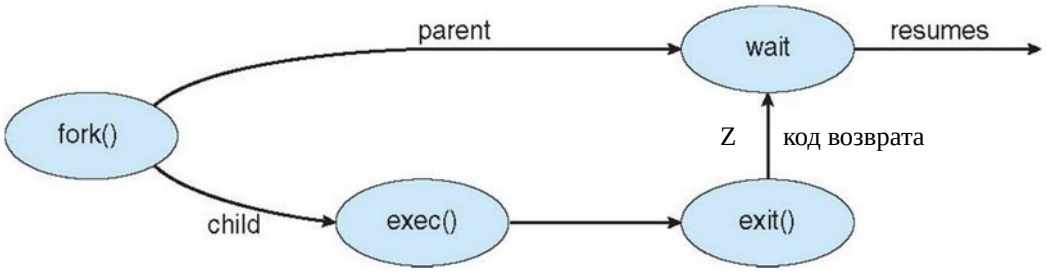
***Практическое задание (по желанию):**
1) С помощью команды top определить количество процессоров в системе
2) С помощью команды top проанализировать нагрузку на сервер (используя показатели load average первых трех строк)
3) При возможности подключиться к ВМ через SSH (в Virtual Box поставить настройки сетевого адаптера "Мост" и с помощью команды "ip a s" узнать IP-адрес ВМ, после чего попробовать зайти на ВМ через любой SSH-клиент), найти свою сессию в списке процессов и узнать PID процесса. "Прибить" процесс, PID которого получили. Что в таком случае произошло или должно произойти?

Как подключиться по SSH к ВМ:
https://youtu.be/Ugek2bOLRRk

## 1. Что делает системный вызов fork()?

*fork() в*ызов создаёт новый процесс посредством копирования вызывающего процесса.
Новый процесс считается дочерним процессом.
Вызывающий процесс считается родительским процессом.
Дочерний и родительский процессы находятся в отдельных пространствах памяти. Сразу после fork() эти пространства имеют одинаковое содержимое. Запись в память, отображение файлов *mmap(2)* и снятие отображения *munmap(2)*, выполненных в одном процессе, ничего не меняя в другом.

***Parent* (родитель) – *Child* (дочерний)**



| | | |
|---|---|---|
| 1- Kernel space | kthread | PID2 |
| 2- User space | init (systemd) | PID1 |

- fork() 1) скопируем процесс
- exec() 2) загружает и запускает код
- exit() 3) потомок завершил работу и отправил код возврата
- wait() 4) родитель ожидает позицию возврата

*Комманды:*дентификатор родительского процесса находится в четвёртой колонк
- *ps – efl | head* *PID (process ID); PPID (Parent process ID); kill -9 2025 (PID No.)*
- *ls- l /sbin/init*
- *pstree*

**Дочерний процесс** является точной копией родительского процесса за исключением следующих моментов:
- Потомок имеет свой уникальный идентификатор процесса, и этот *PID* (идентификатор процесса) не совпадает ни с одним существующим идентификатором группы процессов (setpgid(2)).
- Идентификатор родительского процесса у потомка равен идентификатору родительского процесса.
- Потомок не наследует блокировки памяти родителя (mlock(2), mlockall(2)).
- Счётчики использования ресурсов (getrusage(2)) и времени ЦП у потомка сброшены в 0.
- Набор ожидающих сигналов потомка изначально пуст (sigpending(2)).
- Потомок не наследует значения семафоров родителя (semop(2)).
- Потомок не наследует связанные с процессом блокировки родителя (fcntl(2)) (с другой стороны, он наследует блокировки файловых описаний fcntl(2) и блокировки flock(2)).
- Потомок не наследует таймеры родителя (setitimer(2), alarm(2), timer_create(2)).
- Потомок не наследует ожидающие выполнения операции асинхронного ввода-вывода (aio_read(3), aio_write(3)) и контексты асинхронного ввода-вывода родителя (см. io_setup(2)).

Все перечисленные атрибуты указаны в POSIX.1.
**Родитель и потомок** также отличаются по следующим атрибутам процесса, которые есть только в Linux:
- Потомок не наследует уведомления об изменении каталога (dnotify) родителя (смотрите описание F_NOTIFY в fcntl(2)).
- Настройка PR_SET_PDEATHSIG у prctl(2) сбрасывается, и поэтому потомок не принимает сигнал о завершении работы родителя.
- Резервное значение по умолчанию устанавливается равным родительскому текущему резервному значению таймера. Смотрите описание PR_SET_TIMERSLACK в prctl(2).
- Отображение памяти, помеченное с помощью флага MADV_DONTFORK через madvise(2), при fork() не наследуется.
- Сигнал завершения работы потомка всегда SIGCHLD (см. Clone(2)).
- Биты прав доступа к порту, установленные с помощью ioperm(2), не наследуются потомком; потомок должен установить все нужные ему биты с помощью ioperm(2).

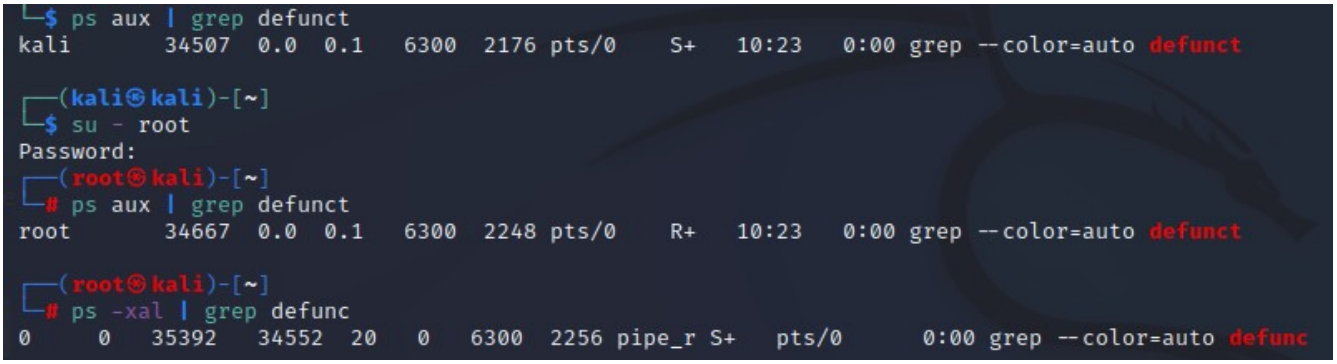
- Процесс потомка создаётся с одиночной нитью — той, которая вызвала fork(). Всё виртуальное адресное пространство родителя копируется в потомок, включая состояние мьютексов, условных переменных и других объектов pthreads; в случае проблем с этим может помочь pthread_atfork(3).
- В многонитивой программе после fork(2) потомок может безопасно вызывать только безопасные-асинхронные-сигнальные функции (смотрите signal(7)) до тех пор, пока не вызовет execve(2).
- Потомок наследует копии набора открытых файловых дескрипторов родителя. Каждый файловый дескриптор в потомке ссылается на то же описание файла что и родитель (смотрите open(2)). Это означает, что два файловых дескриптора совместно используют флаги состояния открытого файла, текущее смещение файла и атрибуты ввода-вывода, управляемые сигналами (смотрите описание F_SETOWN и F_SETSIG в fcntl(2)).
- Потомок наследует копии набора файловых дескрипторов открытых очередей сообщений родителя (смотрите mq_overview(7)). Каждый файловый дескриптор в потомке ссылается на то же описание открытой очереди сообщений что и родитель. Это означает, что два файловых дескриптора совместно используют флаги (mq_flags).
- Потомок наследует копии набора потоков открытых каталогов родителя (смотрите opendir(3)). В POSIX.1 сказано, что соответствующие потоки каталогов в родителе и потомке могут совместно использовать позицию в потоке каталога; в Linux/glibc они не могут этого делать.

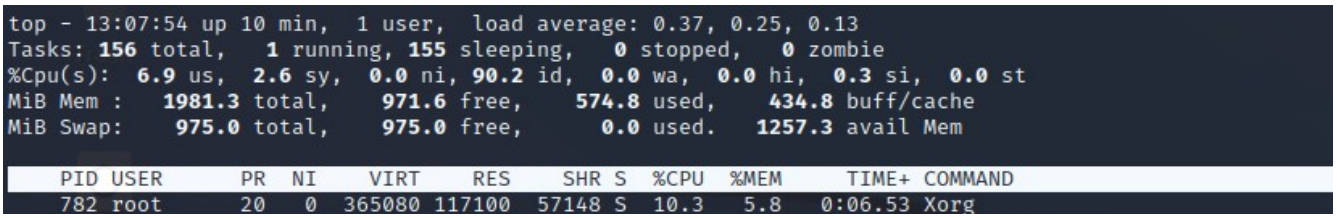
## *2. Что такое процесс зомби?*

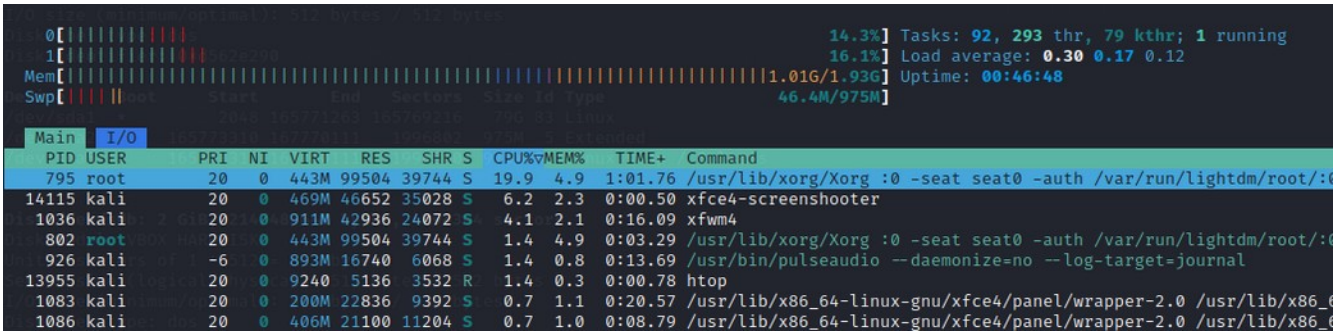
Каждый процесс может запускать дочерние процессы с помощью функции *fork()*. Каждая программа, которая выполняется в Linux - это системный процесс, у которого есть свой идентификатор и такие процессы остаются под контролем родительского процесса и не могут быть завершены без его ведома.
Процесс при завершении освобождает все свои ресурсы (за исключением *PID* - идентификатора процесса) и становится *«зомби»* - пустой записью в таблице процессов, хранящей код завершения для родительского процесса.
Система уведомляет родительский процесс о завершении дочернего с помощью сигнала *SIGCHLD*. Предполагается, что после получения *SIGCHLD* он считает код возврата с помощью системного вызова *wait()*, после чего запись зомби будет удалена из списка процессов.
Если родительский процесс игнорирует *SIGCHLD* (а он игнорируется по умолчанию), то зомби остаются до завершения родительского процесса.
Так, если один из дочерних процессов всё же завершился, а его родительский процесс не смог получить об этом информацию, то такой дочерний процесс становится зомби.
Зомби процессы Linux не выполняются и убить их нельзя, даже с помощью *sigkill*, они продолжают висеть в памяти, пока не будет завершён их родительский процесс.

Посмотреть такие процессы можно с помощью утилит *ps* и др**.**, здесь они отмечаются как defunct:

- *ps aux | grep defunct*
- *top*
- *htop*




## *3. Что такое процесс сирота?*

*Процесс-сирота (orphan process)* — в семействе операционных систем Unix дочерний процесс, чей родительский процесс был завершён.
Все процессы-сироты немедленно усыновляются специальным системным процессом init. Эта операция ещё называется переподчинением (англ. *reparenting*) и происходит автоматически.
Хотя технически процесс *init* признаётся родителем этого процесса, его всё равно считают *«осиротевшим»*, поскольку первоначально создавший его процесс более не существует.
В современных Linux (начиная с ядра 3.4) поддерживается механизм *subreaper*. Другой процесс, кроме init, может объявить себя «усыновителем». В качестве нового родителя для процесса-сироты становится первый subreaper среди предков данного процесса или, если таких нет, то процесс с *PID*, равным единице. Это нужно для того, чтобы менеджеры сервисов и супервизоры, работающие в *userspace*, могли отслеживать свои дочерние процессы и получать *SIGCHLD*.

## *4. Как убить процесс зомби?*

Чтобы завершить процесс зомби, нужно найти "родителя" этого процесса. Для этого можно использовать команду:

- *ps aux | grep defunct*
- *ps -xal | grep defunct*
- *kill* ***-KILL*** *34552* или *kill* ***-9*** *34552* сперва попробовать *kill* ***-15*** *PID или* ***-TERM***

```
└─$ ps aux | grep defunct
kali       34507  0.0  0.1   6300  2176 pts/0    S+   10:23   0:00 grep --color=auto defunct

┌──(kali㉿kali)-[~]
└─$ su - root
Password:
┌──(root㉿kali)-[~]
└─# ps aux | grep defunct
root       34667  0.0  0.1   6300  2248 pts/0    R+   10:23   0:00 grep --color=auto defunct

┌──(root㉿kali)-[~]
└─# ps -xal | grep defunc
0     0   35392   34552  20   0   6300  2256 pipe_r S+   pts/0      0:00 grep --color=auto defunc

┌──(root㉿kali)-[~]
└─# kill -KILL 34552
Killed

┌──(kali㉿kali)-[~]
└─$ ps aux | grep defunct
kali       35775  0.0  0.1   6300  2292 pts/0    S+   10:28   0:00 grep --color=auto defunct

┌──(kali㉿kali)-[~]
└─$ ps -xal | grep defunc
0  1000   36024    1371  20   0   6300  2184 pipe_r S+   pts/0      0:00 grep --color=auto defunc
```

## *Практическое задание (по желанию):

1) С помощью команды *top* определить количество процессоров в системе

**Количество процессоров** в „top“, нажав на „1“.

В нашем случае **два процессора.**

Сверху слева - данные о нагрузке каждого ядра процессора, объем занятой памяти, *сведения о количестве процессов*, значения ***load avearage*** **(средней нагрузки)**, отображаются значения за 1, 5 и 15 минут назад.

**Tasks** — общее количество запущенных процессов в разных статусах (*running* — выполняемые; *sleeping* — в ожидании; *stopped* — остановленные; *zombie* — «зомби», дочерние процессы, ожидающие завершения родительского процесса).

2) С помощью команды top проанализировать нагрузку на сервер (используя показатели load average первых трех строк)

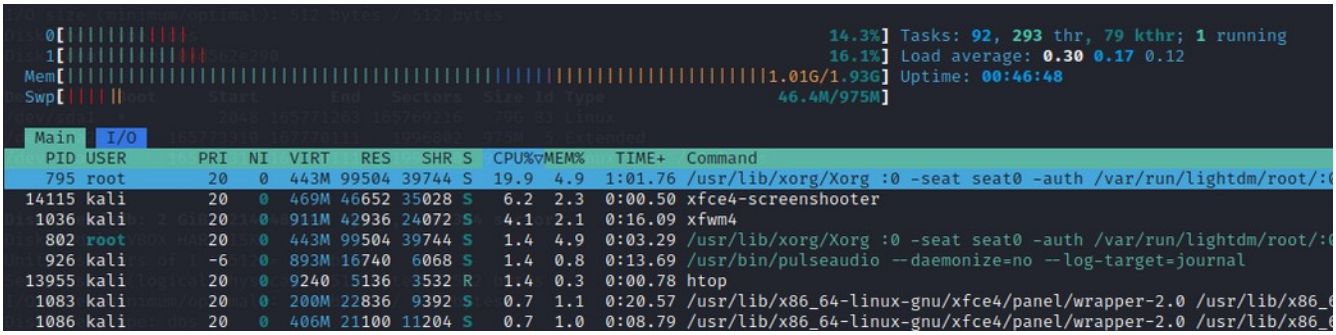
Значения ***load avearage*** **(средней нагрузки)**, отображаются значения за 1, 5 и 15 минут назад.
*Ср. нагрузка* **не должна превышать кол-во CPU**

***В нашем случае*** $0.10 \vee 0.21 \vee 0.18 \leqslant 2$ ***(2-процессора)*** ✔

```
top - 13:21:47 up 24 min,  1 user,  load average: 0.10, 0.21, 0.18
Tasks: 157 total,   2 running, 155 sleeping,   0 stopped,   0 zombie
%Cpu0  :  8.0 us,  2.4 sy,  0.0 ni, 89.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  8.6 us,  3.1 sy,  0.0 ni, 88.0 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
MiB Mem :   1981.3 total,    934.6 free,    595.7 used,    451.0 buff/cache
MiB Swap:    975.0 total,    975.0 free,      0.0 used.   1235.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   7430 kali      20   0  480612  46996  35336 S  10.0   2.3   0:00.35 xfce4-screensho
    782 root      20   0  365080 117228  57148 S   9.0   5.8   0:20.82 Xorg
   1023 kali      20   0  933160 107104  77088 S   2.0   5.3   0:07.09 xfwm4
   1056 kali      20   0  476388  47076  34968 S   1.3   2.3   0:01.14 xfce4-panel
```

** top: количество процессоров в „top“, нажав на* ***„1“***



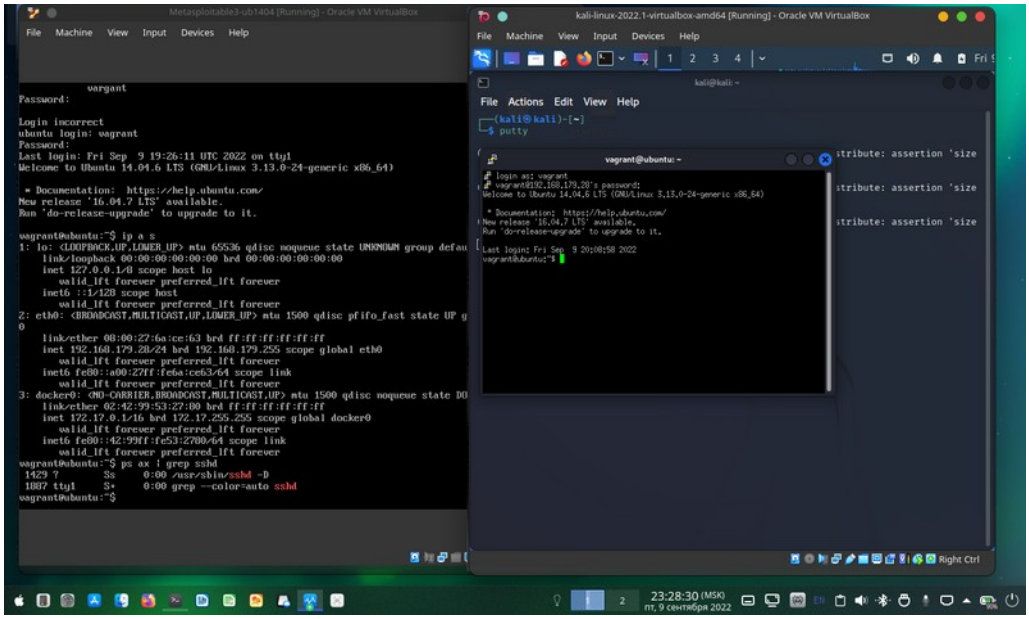

** htop: Processor no. 0, 1 ...*

3) При возможности подключиться к ВМ через SSH (в Virtual Box поставить настройки сетевого адаптера "Мост" и с помощью команды "ip a s" узнать IP-адрес ВМ, после чего попробовать зайти на ВМ через любой SSH-клиент), найти свою сессию в списке процессов и узнать PID процесса. "Прибить" процесс, PID которого получили. Что в таком случае произошло или должно произойти?

Как подключиться по SSH к ВМ:
https://youtu.be/Ugek2bOLRRk

```
└─$ w
 13:53:10 up 55 min,  1 user,  load average: 0.30, 0.23, 0.18
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
kali     tty7     :0               12:57   55:54  44.34s  0.36s xfce4-session
```

- virtualbox client> сеть („сетевой мост“ (network bridge)) ✔
- client> ps ax | grep sshd ✔
- server> putty
  - ip-client 192… port 22



- *top*
- *kill -9 PID_SSH_PuTTY* ***„Fatal Error“*** *(соединение с „client“ / PuTTY прерывается)*