06. Работа с процессами

Егор Орлов

Курс: UNIX-DEV-SYS. Системное программирование в среде UNIX (Linux/FreeBSD). ВИШ СПбПУ, 2021

Содержание

1. Процессы в ОС

1.1. Иерархия процессов в системе

- Процесс **init**

  Ядро после загрузки запускает процесс **init** с идентификатором (PID) 1

  В современных дистрибутивах роль процесса **init** выполняет система **systemd**

  Этот “первый” процесс является прародителем всех остальных процессов в системе, он запускает: системные/сетевые службы, Систему **XWindow** - графический интерфейс процессы, обеспечивающие вход пользователя в систему, командный интерпретатор пользователя, из которого запускаются уже пользовательские приложения

`/proc/1`

Информацию об этом процессе можно посмотреть в **/proc/**

```
$ $ ls -l $(which init)
lrwxrwxrwx 1 root root 22 окт 26 20:51
        /sbin/init -> ../lib/systemd/systemd
```

1.2. Идентификаторы процессов

- **PID**

  Каждый процесс получает уникальный идентификатор (PID) со значениями от 2 (1 - init) до значения

- **PPID**

  Идентификатор родительского процесса - PPID

`/proc/sys/kernel/pid_max`

При достижении максимального значения (система работает долго), начинают задействоваться свободные идентификаторы с начала

`$ ls /proc/<pid>`

Информация о процессах доступна в **/proc**

1.2.1. Пример proc-pid.c

```c
#include <stdio.h>
#include <unistd.h>
int main() {
        printf("PID: %d PPID: %d\n", (int)getpid(), (int)getppid());
        return 0;
}
```

- **getpid(),getppid()** - обюертки над системными вызовами, определены в **unistd.h**

1.3. Утилита pstree

```
$ pstree
init-+-cron
     |-login---bash---pstree
```

```
|-named---18*[{named}]
|-rsyslogd---2*[{rsyslogd}]
`-sshd
```

Иерархию процессов можно посмотреть при помощи **pstree**

1.4. Взаимодействие процесса со средой выполнения

1.4.1. Передача параметров (argc.c)

- Передача параметров вызываемой программе

```
int main(int argc, char* argv[]);
```

- **argc** - количество аргументов

  argc == 1 - аргументов нет, т.к. первый аргумент - сама программа

- **argv** - массив указателей на строки, заканчивающиеся 0

```
#include <stdio.h>
int main(int argc, char *argv[]) {
        printf("Program name: %s\n", argv[0]);
        printf("Parameters count: %d\n", argc-1);
        if (argc > 1)
                for (int i = 1; i < argc; ++i)
                        printf("Param # %d: %s\n",i,argv[i]);
        return 0;
}
```

- **getopt.h**

1.4.2. Коды завершения процессов

- То, что возвращает **main()**

```
$ echo $?
```

1.4.3. Среда выполнения

- **Переменные интерпретатора** - переменные командного интерпретатора, область видимости которых ограничена данным сеансом, локальные переменные. Влияют только на текущий сеанс.

```
переменная=значение
```

Установка значения

```
$ echo $VAR
```

Получение значения

- **Переменные окружения** - определены во всех сеансах и в порожденных процессах, т.е. могут использоваться для их параметризации.

  Переменные окружения являются переменными сеанса интерпретатора, но не наоборот.

```
$ printenv
$ env
```

Получение всех переменных окружения

```
$ export VAR1=value1
$ VAR2=value2
$ export VAR2
```

Создание переменной окружения

- Переменные окружения процесса

```
/proc/<id>/environ
```

- Структура в области памяти процесса

```
#include <stdio.h>
extern char **environ;
```

Глобальная переменная со всеми переменными среды. Массив указателей на символьные строки, заканчивающиеся NULL

Каждая строка имеет формат переменная=значение

```
#include <stdio.h>
extern char **environ;
int main() {
        char **var;
        for (var = environ; *var != NULL; ++var)
                printf("%s\n",*var);
        return 0;
}
```

- Работа с отдельными переменными окружения

```
#include <stdlib.h>
char *getenv(const char *name);
```

Получение значения переменной, или NULL, если переменной с таким именем нет.

```
#include <stdlib.h>
#include <stdio.h>
int main() {
        char *server = getenv("SERVER");
        if (server != NULL)
                printf("Getting access to: %s\n",server);
        else
                printf("Server not set\n");
        return 0;
}
```

- Установка и сброс значения переменных

```
#include <stdlib.h>
int setenv(const char *envname, const char *envval, int overwrite);
int unsetenv(const char *name);
```

- **overwrite** - разрешение изменять значение переменной, если она уже есть

1.5. Создание процессов

- Два способа создания процесса

1. Функция **system()** стандартной библиотеки C (stdlib.h)
2. fork() & exec()

1.5.1. system()

```
#include <stdlib.h>
int system(const char *command);
```

Запускает командный интерпретатор и в нем выполняет указанную в качестве параметра командную конструкцию.

1.5.2. Пример system.c

```
#include <stdlib.h>
int main() {
        return system("yes");
}
```

1.5.3. fork & exec

Клонирует текущий процесс и в клоне заменяет выполняемый образ на содержимое исполняемого файла

```
#include <unistd.h>
pid_t fork(void);
```

- Создает точную копию родительского процесса, за исключением:
  - дочерний процесс имеет уникальный PID
  - дочерний процесс имеет PPID равный родительскому
- Родительский процесс продолжает выполняться со следующей за **fork()** инструкции, то же самое делает и дочерний процесс

```
#include <stdio.h>
#include <unistd.h>
int main() {
        fork();
        fork();
        fork();
        printf("3 times fork\n");
        return 0;
}
```

Делает ветвление 3 раза.

- Функция fork() возвращает **0** в дочернем процессе и **PID** потомка в родительском
  - именно так их и можно отличать

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
        pid_t child_pid;
        printf("*** Before fork *** PID: %d\n",(int)getpid());
        child_pid = fork();
        if (child_pid != 0)
                printf("In parent. My PID: %d Child PID: %d\n",(int)getpid(),(int)child_pid);
        else
                printf("In child. PID: %d PPID: %d\n",(int)getpid(),(int)getppid());
        return 0;
}
```

- Для файловых дескрипторов родительского процесса создаются копии в дочернем, указывающем на те же самые потоки.При этом сами потоки, как структура ядра остаются теми же и в частности все их свойства, например такие, как указатель позиции в файле (lseek).
- MMF выполненные в родительском процессе остаются доступными в дочернем
  - в случае **MAP_PRIVATE** подразумевается поведение **Copy-on-Write**

```
$ man 3p fork
```

- Семейство функций **exec()**

  Заменяют текущий образ выполняемого процесса образом из исполняемого файла, сохраняя при этом основные его свойства, такие как полномочия, открытые потоки ввода вывода.

```c
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execle(const char *path, const char *arg0, ... /*, (char *)0, char *const envp[]*/);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

- Суффиксы;
  - **p** - принимают имя исполняемого файла и ищут его по переменной PATH. Другим функциям надо передавать полный путь к программе.
  - **v** - принимают список аргументов в виде массива строковых указателей **argv[]**
  - **l** - принимают список аргументов переменного размера
  - **e** - в качестве дополнительного аргумента принимают массив строковых указателей на переменные среды в формате **переменная=значение**

  Функции могут вернуть только значение **-1** в случае ошибки. Код при этом в переменной **errno**. Если все в порядке, то управление не возвращается.

- Запуск программы по пути

```
char *cmdline[] = {"ls", "-l", "/var", NULL};
execvp("ls", cmdline);
```

- Или вот так

```
execlp("ls", "ls" , "-a", "/var", NULL);
```

1.5.4. Пример forkexec.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int spawn(char *prog, char** args) {
        pid_t child_pid;
        child_pid = fork();
        if (child_pid != 0)
                return child_pid;
        else {
                execvp(prog,args);
                perror(prog);
                exit(1);
        }
}
int main() {
        char* args[] = { "ls", "-l", "/", NULL };
        spawn("ls", args);
        printf("main() finished\n");
        return 0;
}
```

1.5.5. Корректная работа с высокоуровневым вводом-выводом

- Функция **exit()** перед завершением будет вытеснять буферы высокоуровневого ввода-вывода (stdio.h) на диск.

  Если на момент вызова **fork()** высокоуровневые потоки содержат неотправленные данные, эти данные будут скопированы и в дочерний процесс и в итоге могут быть выведены дважды.

- Поэтому стоит использовать **_exit()**б которая не занимается вытеснением буферов.

```
#include <unistd.h>
void _exit(int status);
```

- Но тогда в случае перенаправления потока ошибок, т.е. когда он не связан с терминалом, вывод **perror/fprintf**может быть потерян.

  Т.е. его неплохо бы вытеснять из буферов высокоуровневой библиотеки, причем делать это до **fork()* и перед выходом из дочернего процесса.

- Итого

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int spawn(char *prog, char** args) {
        pid_t child_pid;
    fflush(stderr);
        child_pid = fork();
        if (child_pid != 0)
                return child_pid;
        else {
                execvp(prog,args);
                perror(prog);
        fflush(stderr);
                _exit(1);
        }
}
int main() {
        char* args[] = { "ls", "-l", "/", NULL };
        spawn("ls", args);
        printf("main() finished\n");
        return 0;
}
```

1.6. Утилита ps - список процессов

```
$ ps
```

Получает информацию из /**proc** Вывод процессов текущего сеанса командного интерпретатора (привязанных к текущей TTY)

- **TIME**

процессорное время, потребленное процессом

```
$ ps x
```

Вывод всех процессов текущего пользовтеля, независимо от TTY

- **STAT**

текущее состояние процесса

```
$ ps aux
$ ps -ef
```

Вывод всех процессов - **a** в удобном - **u** формате

```
$ ps -u \<user\>
```

Вывод процессов конкретного пользователя

```
$ ps f (ps --forest)
```

Вывод процессов в древовидном формате, как в **pstree**

1.7. Атрибуты процессов

Атрибут	Описание
PID	идентификатор процесса
PPID	идентификатор родительского процесса
EUID	эффективный идентификатор пользователя
RUID	реальный идентификатор пользователя
EGID	эффективный идентификатор группы
RGID	реальныф идентификатор группы
GROUPS	перечень групп, в которые входит EUID
SID	идентификатор сеанса (session ID)
PGID	идентификатор группы процессов
TPGID	идентификатор терминальной группы
TTY	терминал, к которому привязан процесс
NICE	число - возможность отдавать CPU другим

1.8. Процессы, группы и сеансы

- Для удобства управления процессами при помощи сигналов процессы объединяются в **группы и сеансы**
- **Сеанс** (session) связывает процессы с **управляющим терминалом**, когда пользователь входит в систему, все создаваемые им процессы будут принадлежать сеансу, связанному с его текущим терминалом. Процесс, создавший сеанс, имеет идентификатор PID, совпадающий с идентификатором SID, называется лидером сеанса.
- **Группы** процессов внутри сеансов позволяют управлять, какие из процессов сеанса работают **на переднем фоне**. Процесс, создавший группу, имеет идентификатор PID, совпадающий с идентификатором PGID, называется лидером группы. Только одна группа сеанса называется “терминальной” TGPID, является группой переднего фона (foreground).

1.9. Состояния процессов

STAT	Описание
R	Выполняется
S	Готов - ожидает выполнения (спит)
D	Ожидает - приостановлен (ввод-вывод)
T	Остановлен (принудительно)
Z	Зомби - завершился, но не был удален

1.10. Параметры процессов

STAT	Описание
<	высокоприоритетный процесс
N	низкоприоритетный процесс
s	лидер сеанса
L	часть страниц блокирована в памяти

STAT	Описание
l	мульти-поточный (нити)
+	работает в группе переднего плана

1.10.1. Жизненный цикл процесса

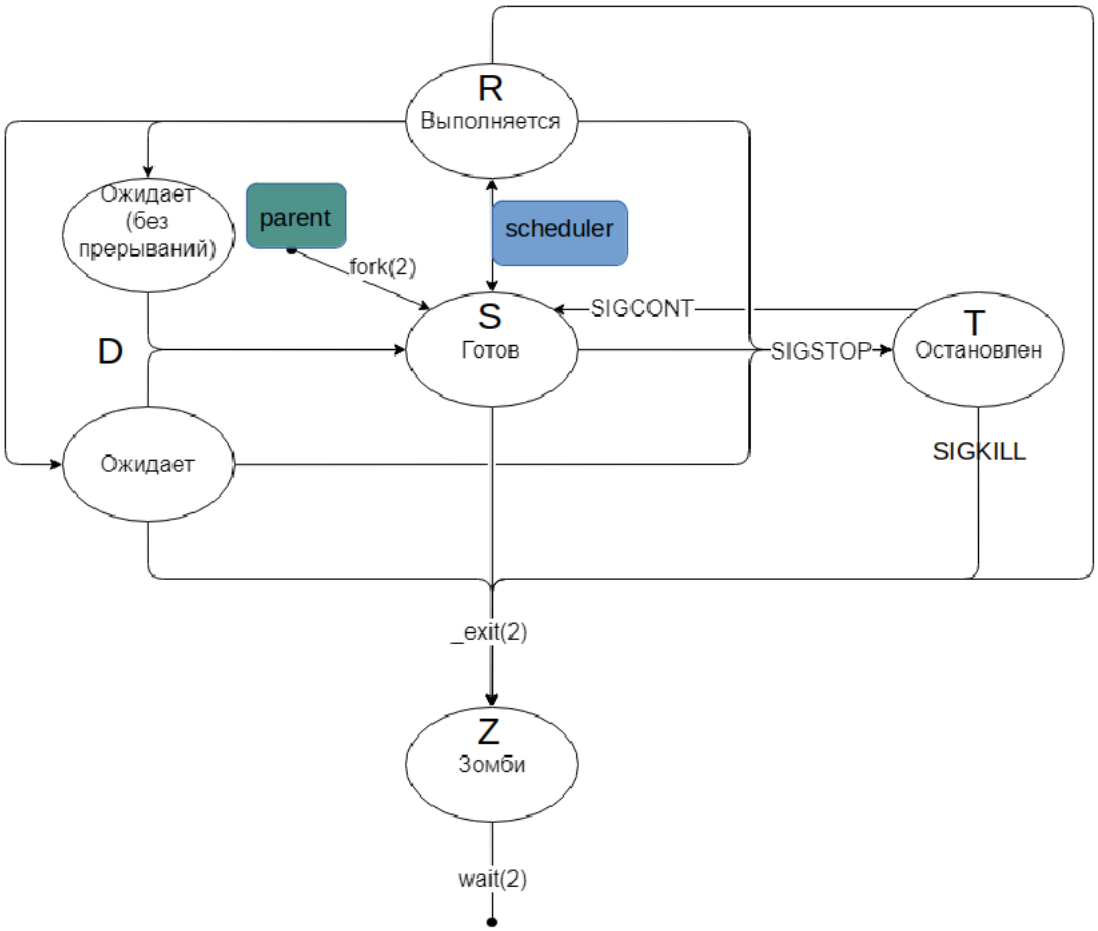

Рис. 1: Жизненный цикл процесса

1.11. Завершение процесса

- Системный вызов **_exit()** - завершает процесс.

```
#include <unistd.h>
void _exit(int status);
```

  Принимает код завершения процесса.

- Возврат из функции **main()** при помощи **return** и, что то же самое, вызов функции **exit()**

делает гораздо больше, в частности, вытесняет все из буферов высокоуровневой библиотеки ввода-вывода.

1.11.1. Примеры использования высокоуровневого ввода-вывода

```
#include <unistd.h>
#include <stdio.h>
int main() {
        printf("String");
        _exit(0);
}
```

Ничего не выведет

```
#include <unistd.h>
#include <stdio.h>
int main() {
        printf("String\n");
        _exit(0);
}
```

Вывод будет только на терминал, а при перенаправлении - не будет.

```
#include <unistd.h>
#include <stdio.h>
int main() {
        printf("String\n");
        fork();
        return 0;
}
```

Без перенаправления - вывод один раз, с перенаправлением - два.

- Другие варианты завершения процессов - по приему сигнала, подразумевающего завершение работы после возможно какой-то обработки. В этом случае кода завершения нет, но есть код сигнала.

1.12. Ожидание завершения процесса

- После завершения процесса в системе (в таблице процессов ОС) остается информация о том, как он завершился (с каким кодом, или каким сигналом), а так же значения счетчиков потребленных ресурсов.

  Эту информацию должен затребовать родительский процесс, если же родительский процесс завершится раньше, то эту функцию берет на себя процесс **init**.

  Значение **PPID** у осиротевшего процесса равно **1**.

- Завершенный процесс продолжает существовать в системе в виде **процесса-зомби** (состояние Z). Когда родительский процесс запрашивает информацию об обстоятельствах завершения процесса, место в в таблице процессов освобождается.

```
#include <stdlib.h>
#include <sys/types.h>
```

```c
#include <unistd.h>

int main() {
        pid_t child_pid;
        child_pid = fork();
        if ( child_pid > 0 )
                sleep(60);
        else
                exit(0);
        return 0;
}
```

- Системные вызовы семейства **wait**

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

Если у процесса нет ниодного потомка - возвращает **-1**

Если потомки есть, но ни один из них еще не завершился - ждет завершения.

По появлению среди потомков процесса-зомби извлекает из него информацию об обстоятельствах завершения и освобождает слот в таблице процессов.

Возвращает PID завершившегося процесса.

Если при вызове параметр представлял собой ненулевой указатель, то в область памяти, на которую он указывает заносится код завершения или номер сигнала, завершивший процесс.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
int spawn(char *prog, char** args) {
        pid_t child_pid;
        child_pid = fork();
        if (child_pid != 0)
                return child_pid;
        else {
                execvp(prog,args);
                fprintf(stderr, "Error launching program %s\n",prog);
                exit(1);
        }
}
int main() {
        int child_status;
        char* args[] = { "ls", "-l", "/", NULL };
        spawn("ls", args);
        wait(&child_status);
        if (WIFEXITED(child_status))
                printf("Normal exit, code %d\n",WEXITSTATUS(child_status));
```

```
        else
                printf("Abnormal exit\n");
        printf("main() finished\n");
        return 0;
}
```

- макросы
  - **WIFEXITED** - позволяет узнать, нормально ли завершился процесс
  - **WEXITSTATUS** - возвращает код возврата при нормальном завершении
  - **WIFSIGNALED** - завершился ли по сигналу
  - **WTERMSIG** - каов код сигнала
-

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Ожидание завершения конкретного процесса по PID

1.13. Сигналы

- Средство для изменения состояния процессов из-вне
- Могут передаваться целевому процессу (и тогда могут быть проигнорированы им)
- **Неигнорируемые** - обрабатываются ядром и таким образом срабатывают всегда - **SIGKILL** и **SIGSTOP**, обработчик поставить нельзя

```
$ kill [ -SIGNAL ] PID
$ killall [ -SIGNAL ] { user | name }
```

- Системный вызов

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

- Определение сигналов - целые числа от 1 до 31

```
#include <bits/signum-generic.h>
#include <bits/signum-arch.h>
```

Обычно включать не надо, для работы с сигналами используют

```
#include <signal.h>
```

1.13.1. Часто используемые сигналы

№	Имя	Описание
1	SIGHUP	обрыв связи, переинициализация
2	SIGINT	Прервать (Ctrl+C)
9	SIGKILL	Уничтожить (не игнор)
10	SIGUSR1	“Пользовательский” сигнал
11	SIGSEGV	Нарушение сегментации
12	SIGUSR2	“Пользовательский” сигнал
15	SIGTERM	Завершить (сигнал по умолчанию)
18	SIGCONT	Продолжить после STOP

№	Имя	Описание
19	SIGSTOP	Приостановить (не игнор)
20	SIGTSTP	Стоп с клавиатуры (Ctrl+Z)

```
$ kill -9 87564
```

1.13.2. Работа с заданиями (jobs)

- Запуск процесса в фоновом режиме (создание задания)

```
$ dd if=/dev/zero of=/dev/null &
[1] 6453
```

- Просмотр списка заданий

```
$ jobs
[1]+ Running   dd if=/dev/zero of=/dev/null &
```

- Возврат процесса на передний план

```
$ fg %1
```

- Приостановка процесса переднего плана (без его завершения) - **Ctrl+Z** - осылка сигнала TSTP

```
$ dd if=/dev/zero of=/dev/null
^Z
[2]+  Остановлен    dd if=/dev/zero of=/dev/null
$
```

- Перевод приостановленного процесса в работу в фоновом режиме

```
$ bg %3
[3]+ dd if=/dev/zero of=/dev/null &
```

1.13.3. Обработка сигналов процессами

- Процессы могут определять свои обработчики для отправляемых процессу сигналов, кроме неигнорируемых.

```
void handler(int s);
```

- Сигнал может придти в любой момент, в том числе и при обработке другого сигнала.

  Установка значений глобальным переменным может занимать несколько тактов, поэтому в обработчике сигнала лучше работать с переменными специального типа **sig_atomic_t**, которые гарантированно меняют значение за такт.

  Следует избегать операций ввода-вывода и использования “тяжелых” библиотечных функций в обработчиках сигналов. Необходимо быстро вернуть управление

- Запрос и установка обработчика сигнала

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);
```

Определяет правила обработки поступающего сигнала

- **sig** - номер сигнала
- **act** - указатель на структуру **sigaction**, используется для регистрации нового обработчика сигнала
- **oact** - указатель на структуру **sigaction**, используется для описания старого обработчика сигнала

  Структуру **sigaction** перед использованием следует обнулить. Установка обработчика производится присваиванием значения полю **sa_handler**

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

sig_atomic_t sig_count = 0;

void usr1_handler (int signum) {
        ++sig_count;
}

int main () {
        struct sigaction sa;
        memset(&sa, 0, sizeof(sa));
        sa.sa_handler = &usr1_handler;
        sigaction(SIGUSR1, &sa, NULL);
        printf("Start counting SIGUSR1 in PID %d\n", (int)getpid());
        sleep(100);
        printf("SIGUSR1 was raised %d times\n", (int)sig_count);
}
```

1.14. Приоритезация процессов

- **NICE** - фактор уступчивости
- **Niceness (уступчивость)** - свойство процесса оставлять процессорное время другим процессам - процессы с **высоким приоритетом** называют менее уступчивыми - **меньше nice**
  - с **низким приоритетом** - более уступчивые - **больше nice**
- Значения параметра **nice** - от **-20*** - минимальное nice, **макисмальный** приоритет
  - до **19** - максимальное nice, **минимальный** приоритет
  - по умолчанию - **0**

1.14.1. Изменение фактора уступчивости

- Пользователь может только увеличивать значение **nice**, для уменьшения необходимы привелегии суперпользователя
- Запуск процесса с определенным значением **nice** - **nice -n 10 low-prg**
  - **nice -n -20 high-prg**

- Изменение значения **nice** для работающего процесса - **renice +5 -p 3245**

```
$ renice +5 17489
17489 (process ID) старый приоритет 0, новый приоритет 5
$ renice +2 $(pgrep firefox)
2990 (process ID) old priority 0, new priority 2
2993 (process ID) old priority 0, new priority 2
```

```
#include <unistd.h>
int nice(int incr);
```

Позволяет увеличить фактор уступчивости текущего процесса

1.14.2. Связь фактора уступчивости и приоритета

- Чем больше **nice**, тем меньше приоритет и наоборот
- Вот это не про приоритет, это про **nice**

```
$ renice +2 $(pgrep firefox)
2990 (process ID) old priority 0, new priority 2
2993 (process ID) old priority 0, new priority 2
```

- Приоритет(pri) и nice(ni) в выводе **ps**
  - **pri=19-ni**

```
$ ps x -o user,pri,ni,stat,command | grep firefox
egor      17   2 SN   firefox
egor      17   2 SNl  /usr/lib64/firefox/firefox
```

- В выводе **top** - параметр **pr** - не приоритет, он растет вместе с nice
  - **pr=20+ni**