

Московский Физико-Технический Институт  
Физтех-школа Аэрокосмических технологий  
Институт Аэромеханики и Летательной Техники



# **Архитектура Компьютера и Операционные Системы. Часть 2. Основы операционных систем**

## **Лекция 3: Процессы в Unix**

Новиков Андрей Валерьевич  
д.ф.-м.н.

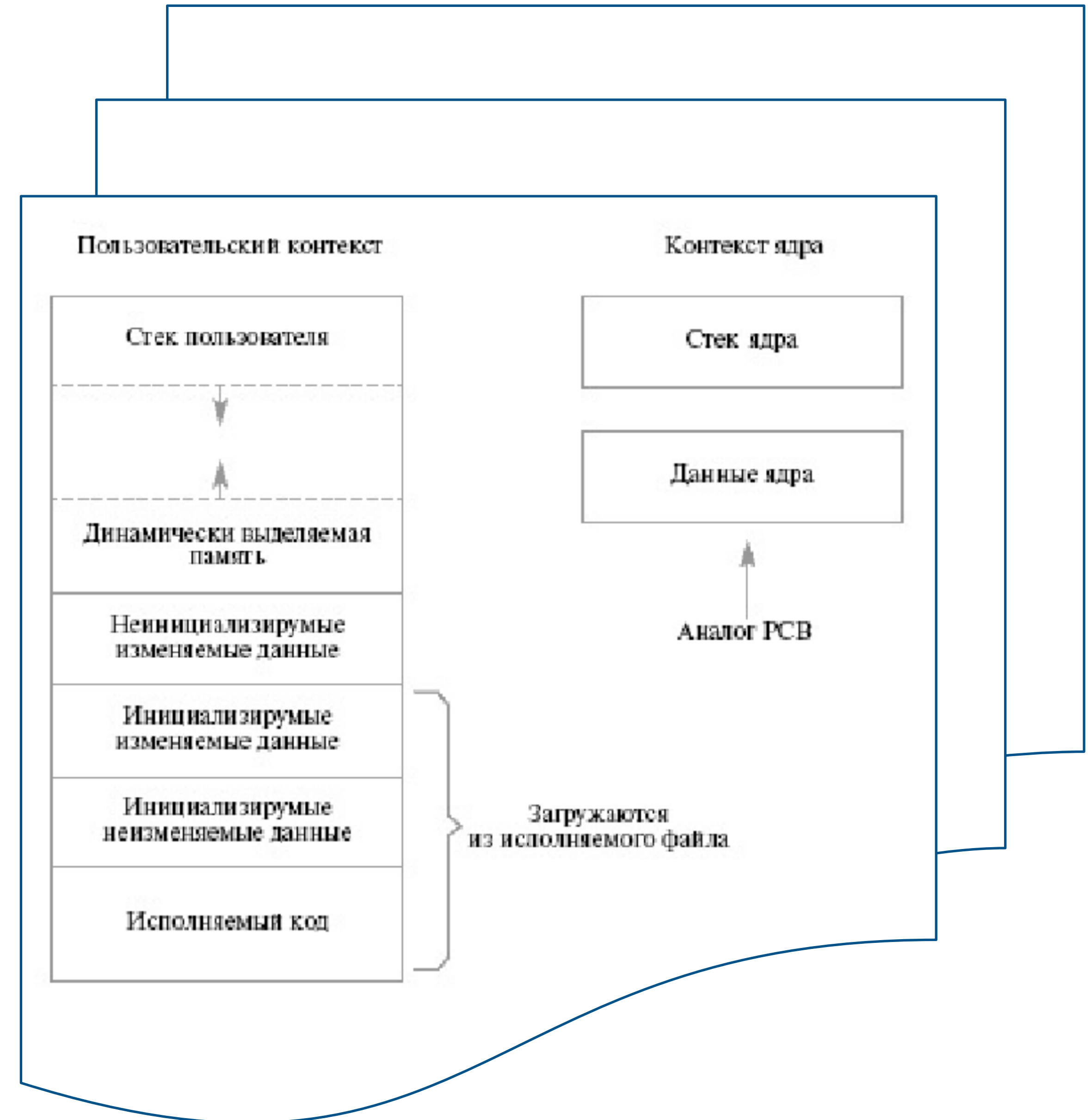
**Жуковский**

# Процесс

- ❑ Программа + входные данные + момент исполнения
  
- ❑ Процесс — совокупность
  - набора исполняющихся команд,
  - ассоциированных ресурсов (выделенная память, адресное пространство, стеки, файлы, устройства ввода-вывода и т.д.),
  - текущего момента выполнения (значения регистров, программного счётчика, состояния стека, значения переменных и т.д.)под управлением операционной системы
  
- ❑ Всё что выполняется в ОС – процесс!

# Контекст процесса

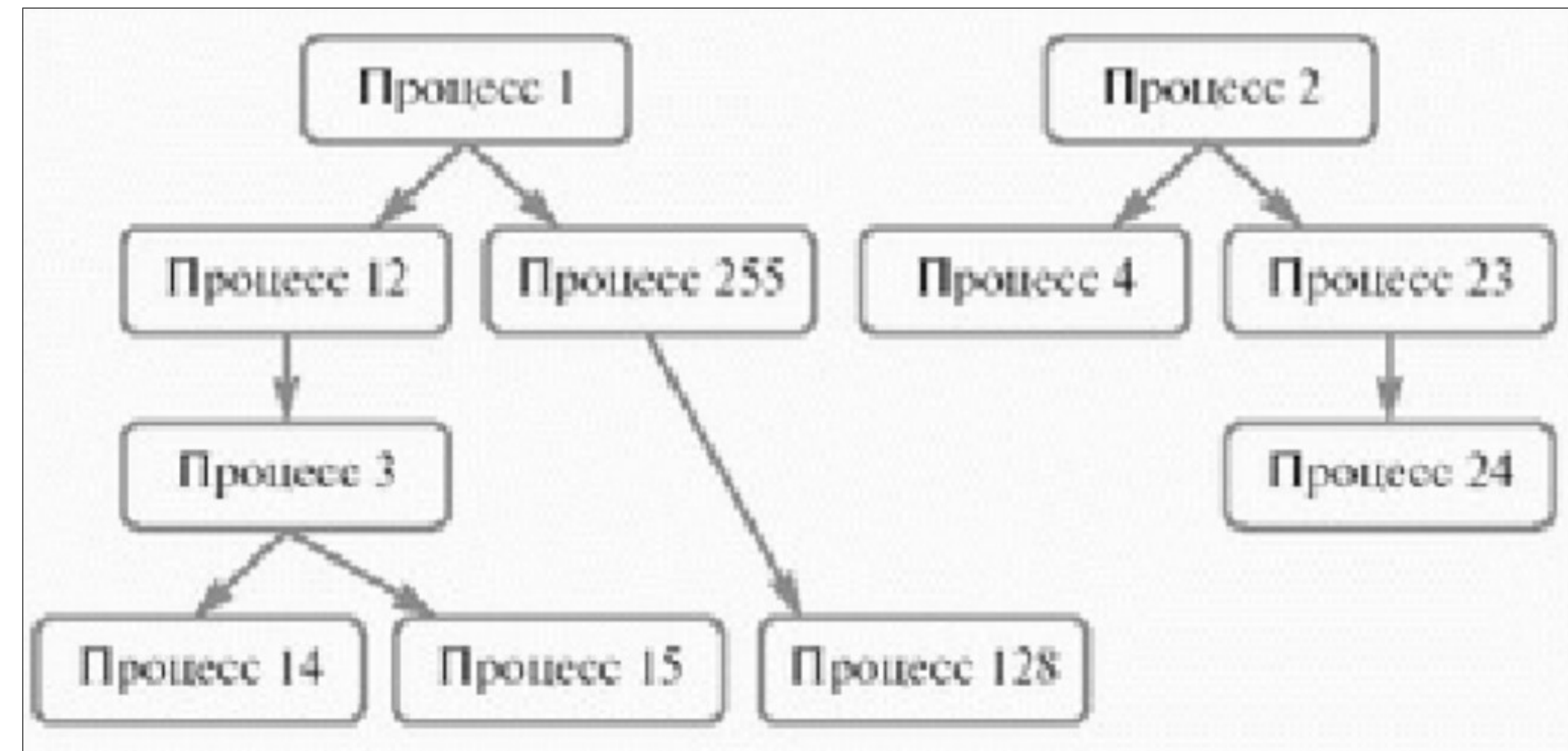
- ❑ Контекст = пользовательский + ядерный
- ❑ Контекст ядра = регистровый контекст + системный; стек ядра + данные ядра
- ❑ Данные ядра (PCB Process Control Block):
  - идентификатор процесса PID
  - идентификатор родительского процесса PPID
  - идентификатор пользователя UID
  - идентификатор группы GID
  - ...





# Иерархия процессов

- ❑ Каждый процесс порождается другим
- ❑ Создающий процесс – родитель (parent process)
- ❑ Создаваемый процесс – ребёнок, «дочерний процесс» (child process)
- ❑ Идентификатор процесса в Unix – **PID** – уникальное целое число  $2^{31} - 1$



# Команды управления процессами в Linux



**ps**  
мгновенный список процессов, запущенных текущим пользователем в текущем терминале



**ps -eF** (или **ps aux**)  
мгновенный список всех процессов в системе в расширенном формате



**top**  
динамический список работающих процессов



**htop**  
динамический список процессов с интерфейсом

0[|

1[|

Mem[|||||||||||||

Swp[

1.3%

1.3%

476M/1.92G

0K/0K

Tasks: 77, 137 thr, 66 kthr; 1 running

Load average: 0.17 0.06 0.02

Uptime: 01:21:19

Main

I/O

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1	root	20	0	127M	12996	9776	S	0.0	0.6	0:02.33	/usr/lib/systemd/systemd --switched-root --system --deseria
478	root	20	0	49912	9396	8196	S	0.0	0.5	0:00.24	/usr/lib/systemd/systemd-journald
497	root	20	0	58832	9720	7452	S	0.0	0.5	0:00.30	/usr/lib/systemd/systemd-udev
528	root	16	-4	13472	1752	1532	S	0.0	0.1	0:00.04	/sbin/auditd
540	root	20	0	8472	7880	1932	S	0.0	0.4	0:00.98	/usr/sbin/haveged -w 1024 -v 0 -F
561	avahi	20	0	18156	3720	3368	S	0.0	0.2	0:00.04	avahi-daemon: running [suse-vm.local]
563	messagebus	20	0	24996	5696	4556	S	0.0	0.3	0:00.63	/usr/bin/dbus-daemon --system --address=systemd: --nofor
572	root	20	0	90044	1652	1484	S	0.0	0.1	0:00.13	/usr/sbin/irqbalance --foreground
589	nscd	20	0	770M	2952	2452	S	0.0	0.1	0:00.06	/usr/sbin/nscd
590	root	20	0	277M	1136	1008	S	0.0	0.1	0:01.35	/usr/bin/VBoxDRMClient
597	root	20	0	20420	6272	5020	S	0.0	0.3	0:00.06	/usr/lib/wicked/bin/wickedd-auto4 --systemd --foreground
610	root	20	0	20424	5900	4652	S	0.0	0.3	0:00.05	/usr/lib/wicked/bin/wickedd-dhcp4 --systemd --foreground
612	root	20	0	20424	6092	4844	S	0.0	0.3	0:00.03	/usr/lib/wicked/bin/wickedd-dhcp6 --systemd --foreground
631	root	20	0	267M	8624	7532	S	0.0	0.4	0:00.29	/usr/lib/systemd/systemd-logind
653	root	20	0	20540	6436	5120	S	0.0	0.3	0:00.06	/usr/sbin/wickedd --systemd --foreground
711	root	20	0	20448	6364	5112	S	0.0	0.3	0:00.03	/usr/sbin/wickedd-nanny --systemd --foreground
727	root	20	0	354M	3216	2824	S	0.0	0.2	0:00.54	/usr/sbin/VBoxService --pidfile /var/run/vboxadd-service
1312	root	20	0	13748	2832	2456	S	0.0	0.1	0:00.00	/usr/sbin/cron -n

F1Help

F2Setup

F3Search

F4Filter

F5List

F6SortBy

F7Nice -

F8Nice +

F9Kill

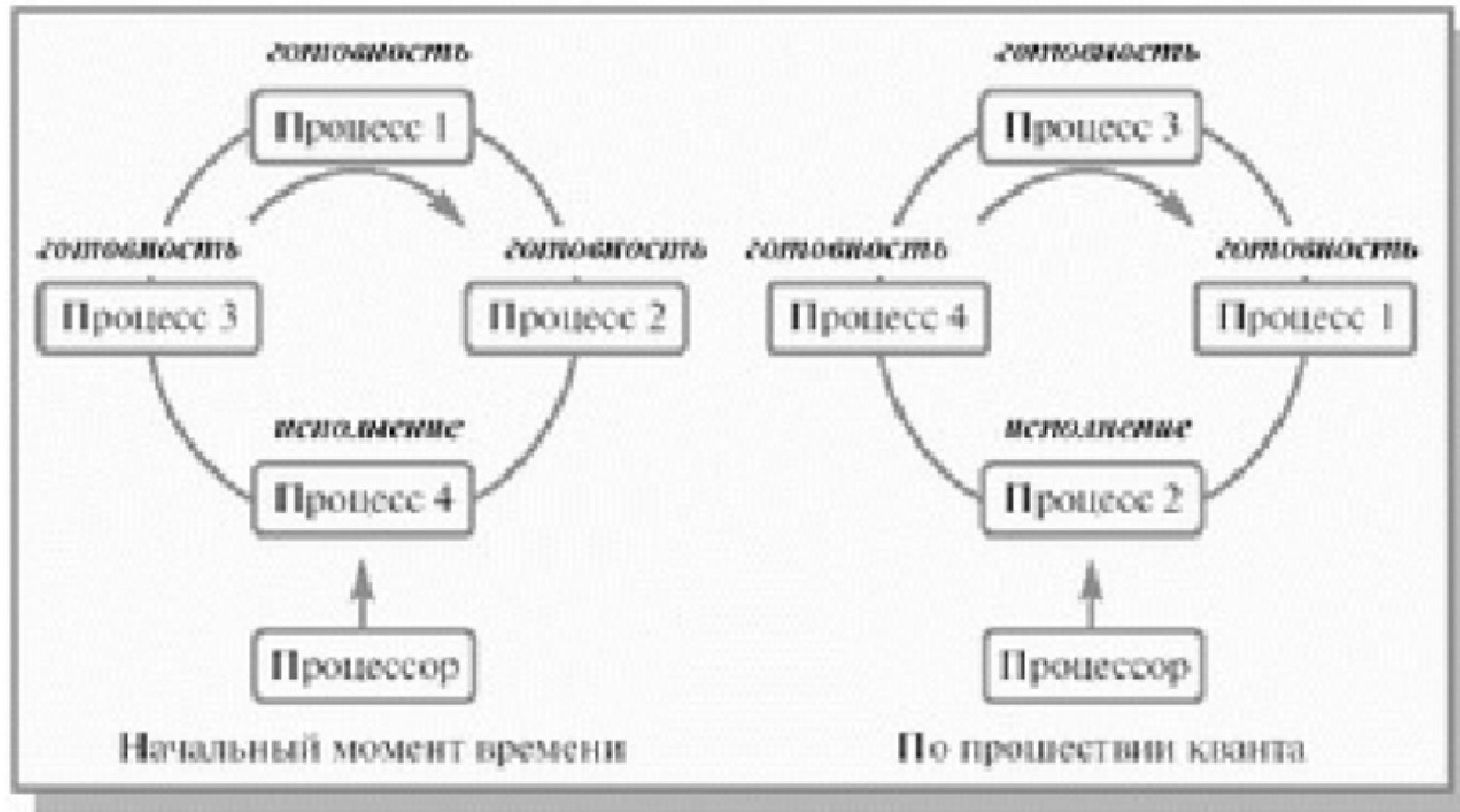
F10Quit

# Диаграмма состояний процесса





# Планирование



# Системные вызовы для PID

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

pid_t getpid(void);
pid_t getppid(void);
```

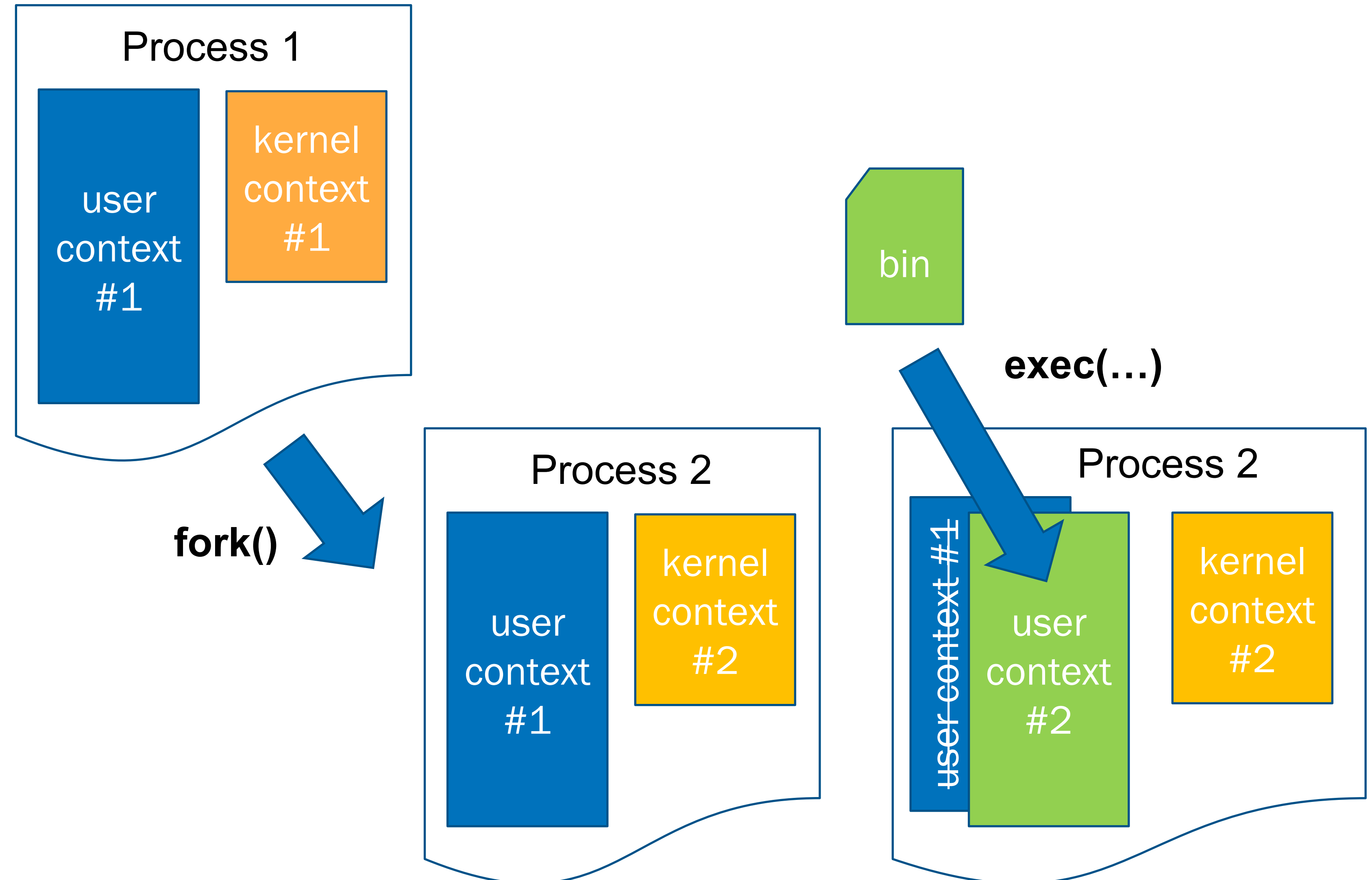
❑ Возвращаемое значение

- >0 – идентификатор процесса (родительского процесса)
- -1 – ошибка



# Создание процесса в Unix

1. Скопировать весь контекст  
(по методу **copy-on-write**)
2. Возможно настроить окружение
3. Заменить  
ПОЛЬЗОВАТЕЛЬСКИЙ  
КОНТЕКСТ



# Системный вызов создания процесса

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

pid_t fork(void);
```

- ❑ Создаёт копию текущего процесса
- ❑ Возвращается **дважды!**
  - в родительском процессе
    - > 0 – PID ребёнка
    - 1 - ошибка
  - в процессе-ребёнке
    - = 0 – успешное порождение

# Использование `fork()`

```
pid_t pid = fork();
if( pid > 0 ){
    // родитель
} else if( pid == 0 ){
    // ребёнок
} else {
    // ошибка
}
```

- ❑ После выхода из `fork()` продолжается исполнение следующего за вызовом кода (одинакового!), как в родителе, так и в ребёнке.
- ❑ Ветви исполнения кода в родителе и ребёнке можно разделить по возвращаемому значению

# Завершение процесса

```
#include <stdlib.h>

void exit(int status);
```

- ❑ Нормальное завершение процесса – закрытие файлов, опустошение буферов, перевод в состояние «закончил исполнение»
- ❑ Возврата в программу НЕТ.
- ❑ status – код завершения процесса передаётся ОС
  - 0 – 255 допустимые значения (8 бит)
  - 0 – индикация успешного завершения



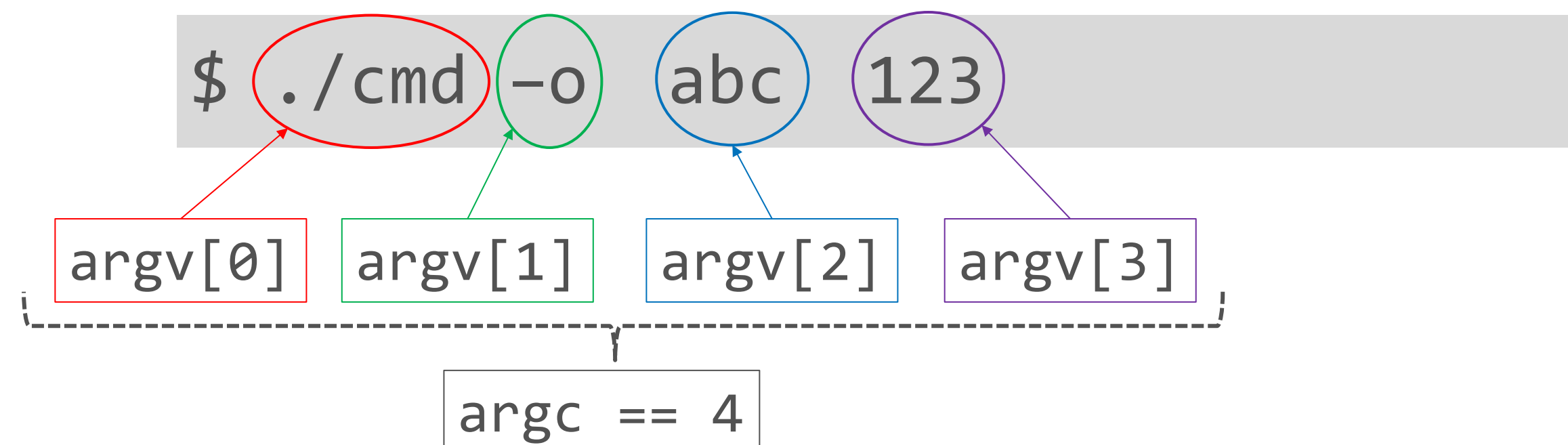
# Завершение процесса

- ❑ Если родитель завершается раньше ребёнка
  - ребёнок получает PPID=1, т.е. родителем становится PID=1 (головной процесс ``init`` или ``systemd``)
- ❑ Если родитель не запрашивает информацию о завершении
  - ребёнок остаётся в состоянии «завершил исполнение»  
= **процесс-зомби** (zombie, defunct)

# Процесс в представлении C

```
int main(  
    int argc,  
    char* argv[],  
    char* envp[]  
) {  
    ...  
    return err_code;  
}
```

- ❑ `main()` – точка входа
- ❑ `return err;` из `main()` → вызов `exit(err)`
- ❑ Параметры передаются из ОС при запуске
  - `argc` – количество слов-аргументов в командной строке (arguments count)
  - `argv[]` – массив слов в «командной строке» (arguments values)
  - `envp[]` – список переменных окружения, строк в виде “var=value”



# Изменение контекста процесса

```
#include <unistd.h>

int exec1(const char *path, const char *arg,
..., NULL);

int exec1p(const char *file, const char *arg,
..., NULL);

int execle(const char *path, const char *arg,
..., NULL, char * const envp[]);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const
argv[]);
int execve(const char *path, char *const argv[],
char* const envp[]);
```

- ❑ Уничтожает(!) текущий пользовательский контекст и **заменяет** его на загружаемый из исполняемого файла
- ❑ Возвращает
  - -1 – при ошибке
  - **НЕ возвращается** при успехе
- ❑ file – имя исполняемого бинарного(!) файла, автопоиск в стандартных каталогах, перечисленных в PATH
- ❑ path – путь к исполняемому бинарному(!) файлу (каталог + имя)

# Изменение контекста процесса

```
#include <unistd.h>

int exec1(const char *path, const char *arg,
..., NULL);

int exec1p(const char *file, const char *arg,
..., NULL);

int execle(const char *path, const char *arg,
..., NULL, char * const envp[]);

int execv(const char *path, char *const argv[]);

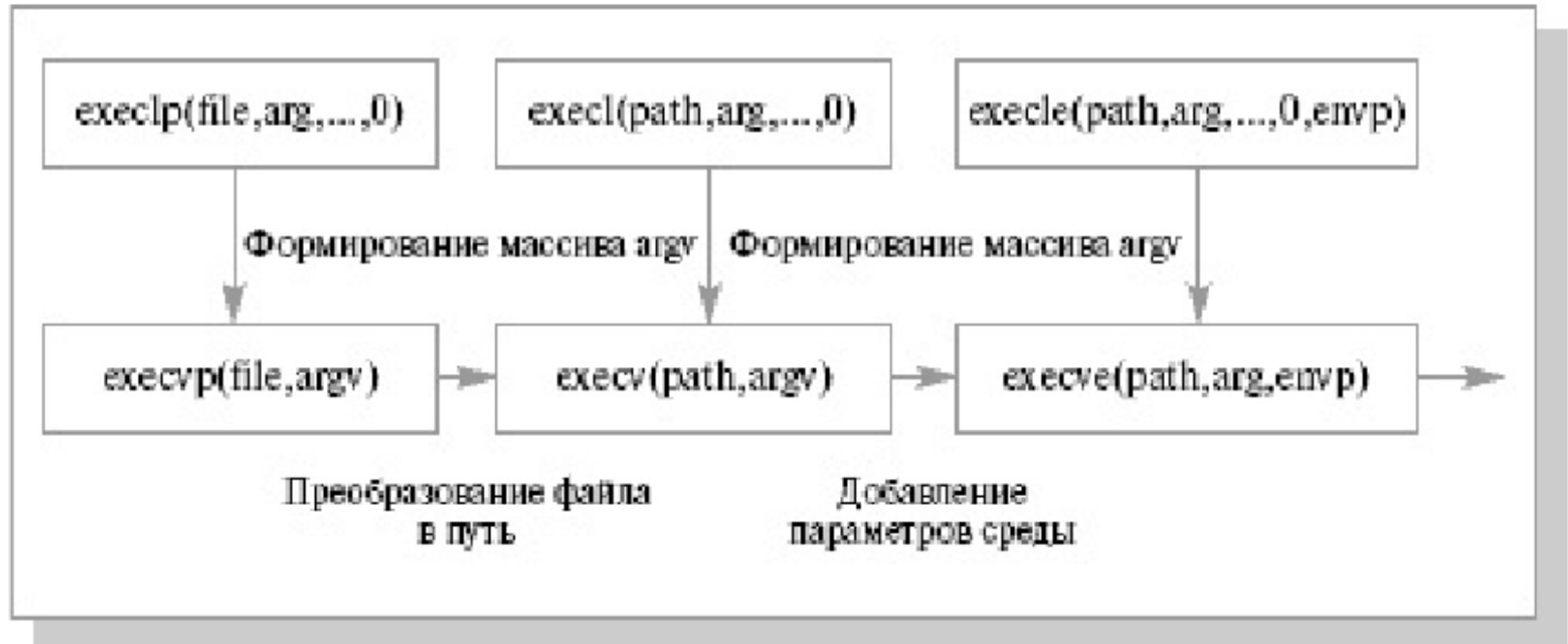
int execvp(const char *file, char *const
argv[]);

int execve(const char *path, char *const argv[],
char* const envp[]);
```

- ❑ **argv[ ]** – массив аргументов «командной строки» для исполняемой программы, который будет доступен через параметр функции `main(...argv[ ]...)`
- ❑ **arg, ..., NULL** – перечисление аргументов командной строки, из которых автоматически формируется `argv[ ]`
- ❑ В обоих случаях, аргументы командной строки должны **явно включать 0-ой**, который обычно соответствует названию запускаемой программы.



# Разновидности `exec()`



# Использование exec()

```
#include <unistd.h>

int main(int argc, char* argv[], char* envp[]){
    pid_t p = fork();
    if( p == -1 ){ perror("fork"); return -1; }
    if( p == 0 ){
        int ret = execl("/bin/ls", "ls", "-l", NULL, envp);
        // if( ret == -1 ) {
        perror("exec"); return -2;
        }
    return 0;
}
```