Процессы и потоки

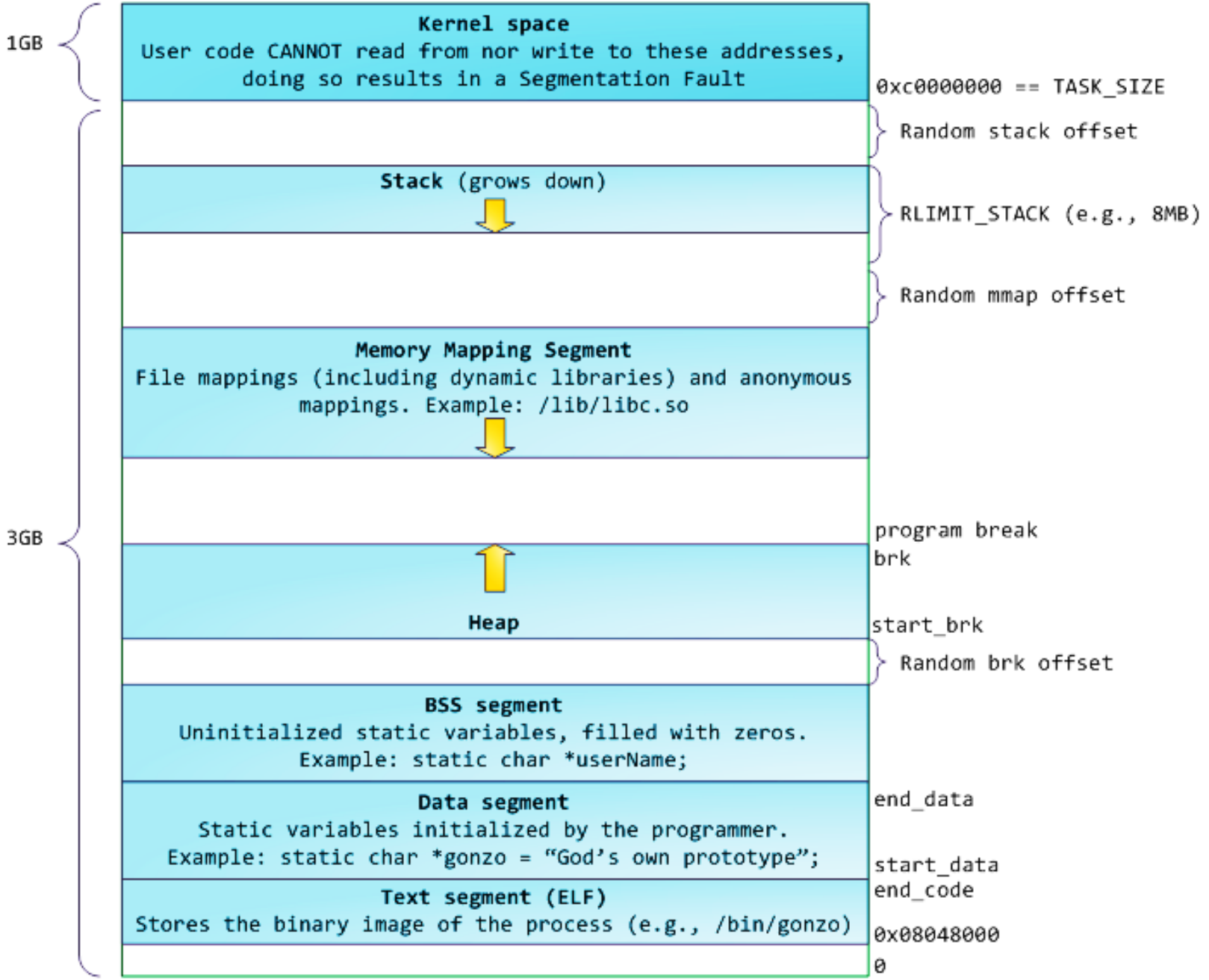
Понятие процесса

Процесс – это программа, находящаяся в состоянии выполнения, вместе с необходимыми ей ресурсами (открытые файлы, сигналы, ожидающие обработки, состояние процессора и т.п.)

Процесс можно представить в виде одного или нескольких потоков выполнения программного кода

Несколько процессов могут выполнять одну и ту же программу и использовать одни и те же ресурсы

Адресное пространство процесса

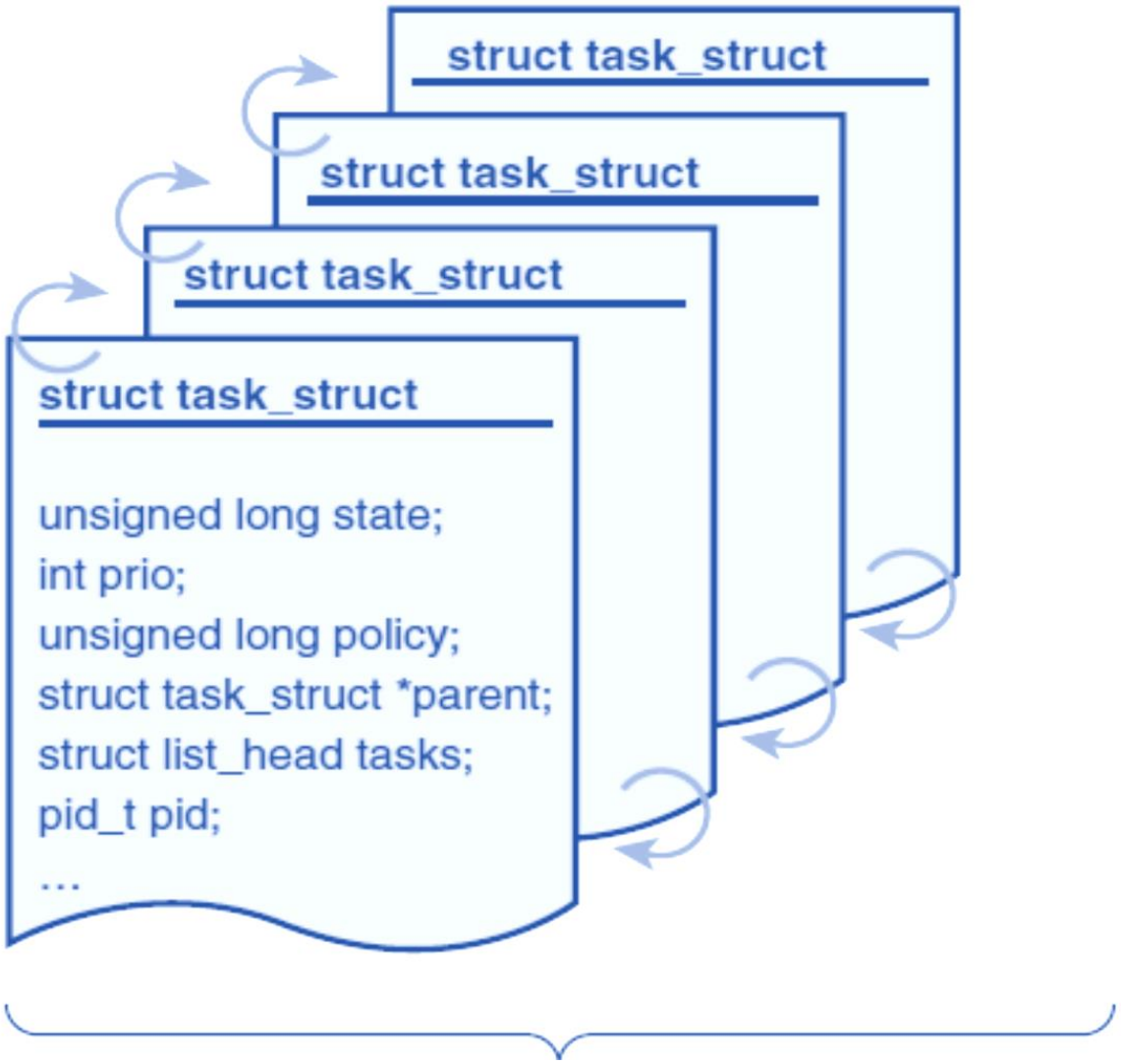


Дескриптор процесса

slab allocator – программа распределения блочного типа

структура thread_info

дескриптор процесса



Идентификатор процесса

Система идентифицирует процессы по значению уникального *идентификатора процесса*

По умолчанию ядро может выставить максимальную величину идентификатора, равную 32 768

Бóльшую величину можно установить во время работы системы, отредактировав файл **`/proc/sys/kernel/pid_max`**

Ядро не назначает использованные ранее идентификаторы, пока не будет достигнута величина, записанная в **`pid_max`**

Процесс бездействия (idle process), который выполняется ядром в отсутствие других процессов, имеет **`pid = 0`**

Первый процесс, который ядро выполняет во время запуска системы, называется *процессом инициализации* и имеет **`pid = 1`**

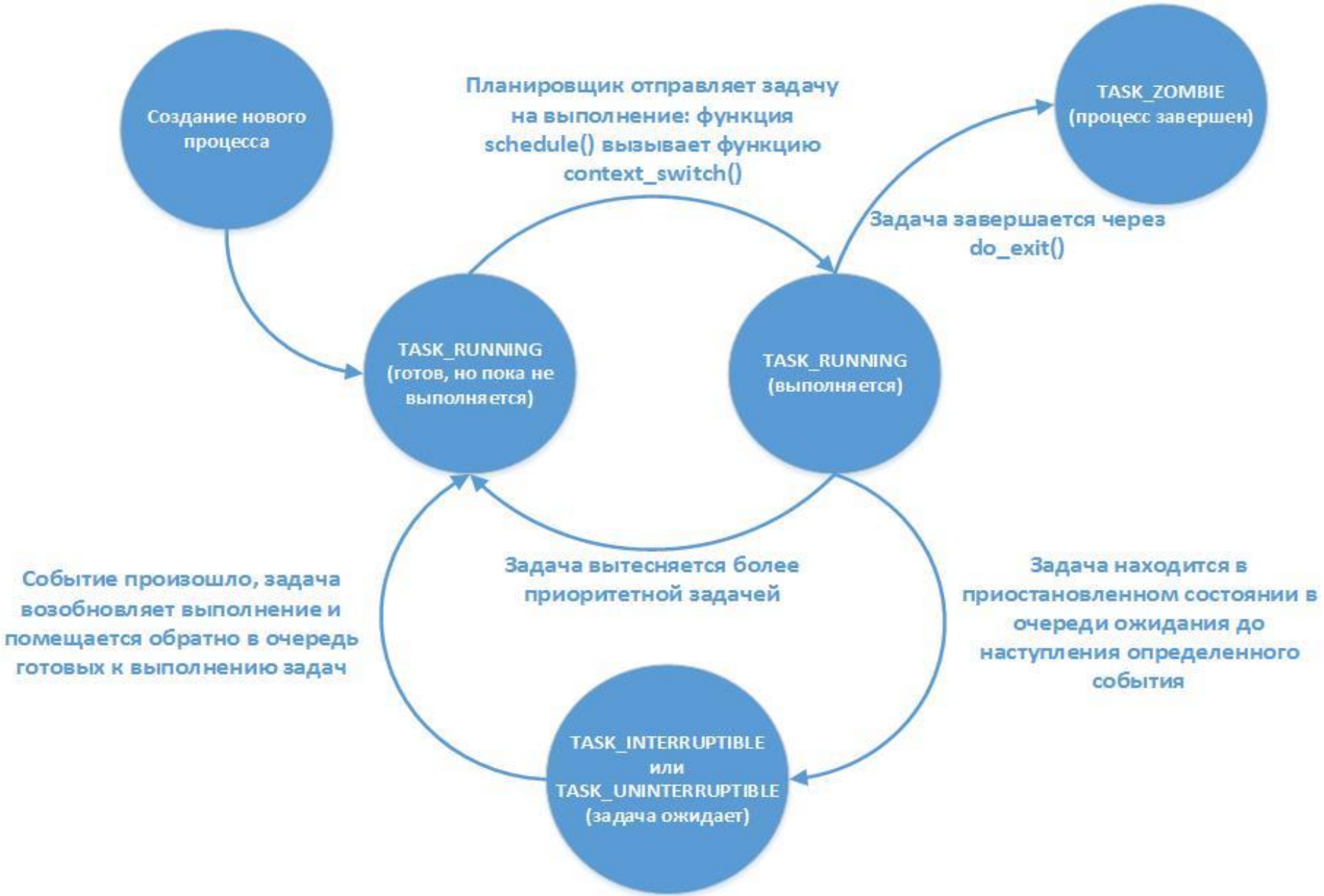
Получение идентификатора процесса

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

pid_t getpid(void);

pid_t getppid(void);
```

Состояния процесса



Процесс инициализации

1. /sbin/init
2. /etc/init
3. /bin/init
4. /bin/sh

7 уровней инициализации:

0 — остановка системы

1 — загрузка в однопользовательском режиме

2 — загрузка в многопользовательском режиме без поддержки сети

3 — загрузка в многопользовательском режиме с поддержкой сети

4 — не используется

5 — загрузка в многопользовательском режиме с поддержкой сети и графического входа в систему

6 — перезагрузка

```
[user@localhost ~]$ runlevel
N 5
[user@localhost ~]$ who -r
         уровень выполнения 5 2015-09-23 10:35
```

Процесс инициализации

systemd — демон инициализации других демонов в Linux, пришедший на замену используемого ранее скрипта инициализации /sbin/init

Дистрибутивы GNU/Linux, в которых systemd установлен по умолчанию:

Ubuntu 15.04 и далее
Fedora 15 и далее
Mageia 2
Mandriva 2011
Rosa
openSUSE 12.1 и далее
Arch Linux 12.11
Sabayon 13.08

Дерево процессов

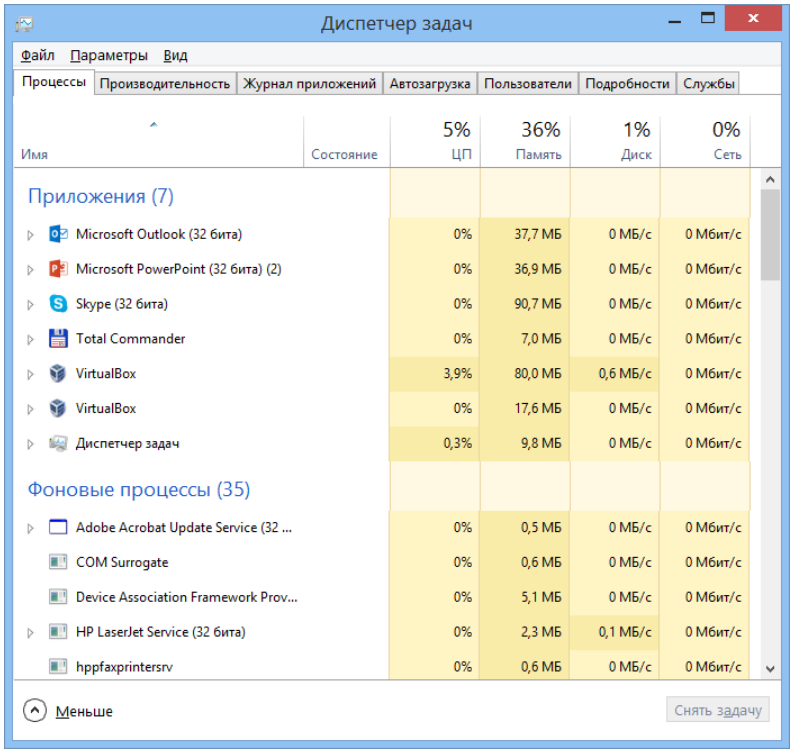
```
[user@localhost ~]$ pstree
```

Создание процесса

- ✓Инициализация системы
- ✓Выполнение работающим процессом системного вызова, предназначенного для создания процесса
- ✓Запрос пользователя на создание нового процесса

Создание процесса при инициализации системы

- ✓Интерактивные процессы, взаимодействующие с пользователями и выполняющие для них определенную работу
- ✓Фоновые процессы (демоны)



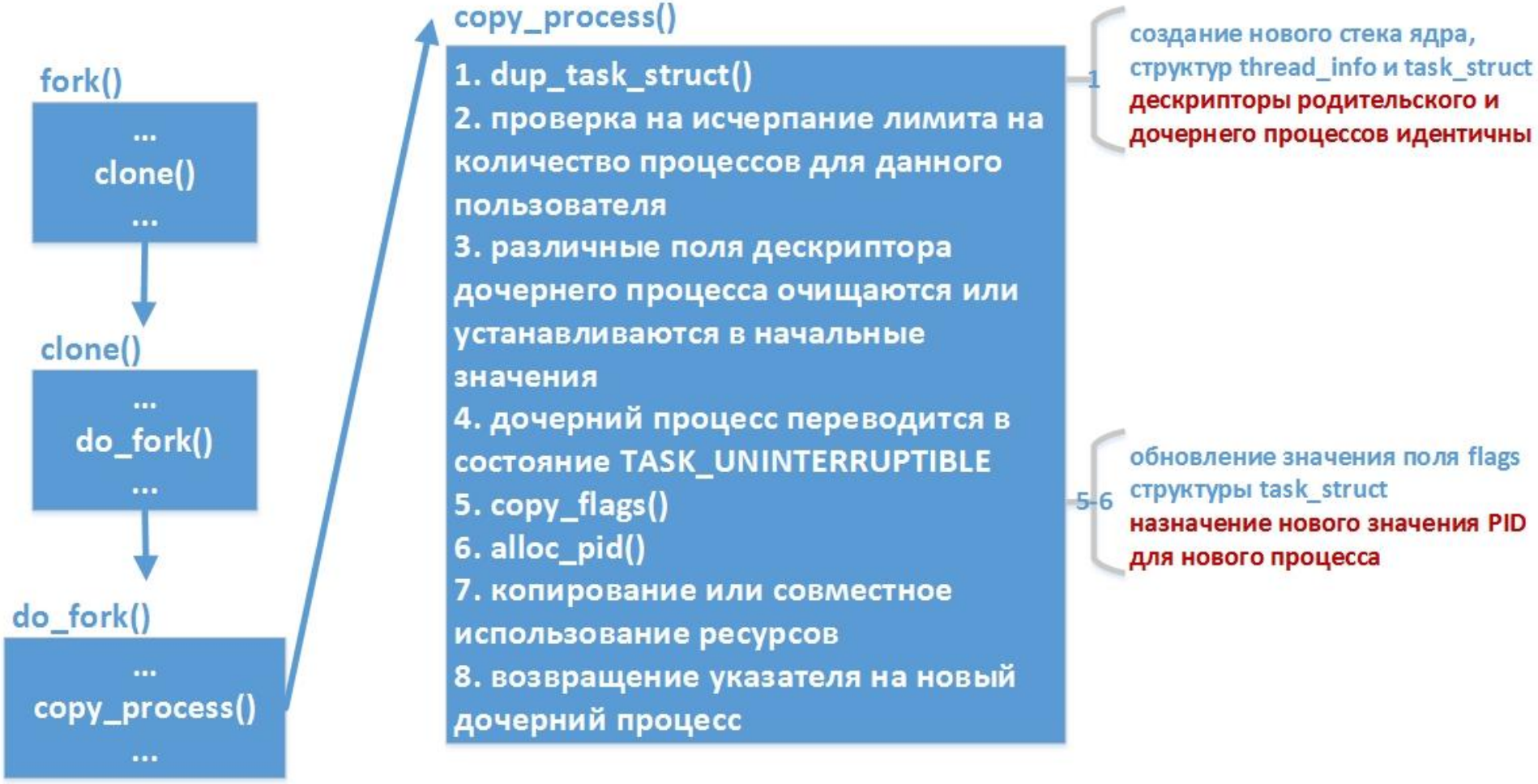

```
[ovm@ovm-pc ~]$ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.6 122892  6884 ?        Ss   11:27   0:01 /usr/lib/systemd/systemd --switched
root         2  0.0  0.0      0     0 ?        S    11:27   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    11:27   0:01 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   11:27   0:00 [kworker/0:0H]
root         7  0.0  0.0      0     0 ?        S    11:27   0:01 [rcu_sched]
root         8  0.0  0.0      0     0 ?        S    11:27   0:00 [rcu_bh]
root         9  0.0  0.0      0     0 ?        R    11:27   0:00 [rcuos/0]
root        10  0.0  0.0      0     0 ?        S    11:27   0:00 [rcuob/0]
root        11  0.0  0.0      0     0 ?        S    11:27   0:00 [migration/0]
root        12  0.0  0.0      0     0 ?        S    11:27   0:00 [watchdog/0]
root        13  0.0  0.0      0     0 ?        S<   11:27   0:00 [khelper]
root        14  0.0  0.0      0     0 ?        S    11:27   0:00 [kdevtmpfs]
root        15  0.0  0.0      0     0 ?        S<   11:27   0:00 [netns]
root        16  0.0  0.0      0     0 ?        S<   11:27   0:00 [perf]
root        17  0.0  0.0      0     0 ?        S<   11:27   0:00 [writeback]
root        18  0.0  0.0      0     0 ?        SN   11:27   0:00 [ksmd]
root        19  0.0  0.0      0     0 ?        SN   11:27   0:00 [khugepaged]
root        20  0.0  0.0      0     0 ?        S<   11:27   0:00 [crypto]
```

Системный вызов для создания процесса

`fork()` + `exec()`

- **`fork()`** создает процесс, который является копией текущего процесса
- новый процесс отличается от родительского значениями PID, PPID, статистикой использования ресурсов
- в системной функции **`fork()`** используется *механизм копирования страниц памяти при их записи* (copy-on-write)
- **`exec()`** загружает исполняемый файл в адресное пространство процесса и передает ему управление

Системный вызов **fork()**



Системный вызов **fork()**

```
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char ** argv)
{
   int pid;
   printf ("Создаю дочерний процесс\n");
   pid = fork();
   if(pid == -1) abort();
   if(pid == 0)
   {
      printf("Работает дочерний процесс\n");
   }else{
      printf("Работает родительский процесс\n");
   }
   printf("Работают оба процесса\n");
   return(0);
}
```

В системе существует один процесс

Два процесса работают параллельно

Системный вызов **exec()**

```
#include <unistd.h>

int execl(const char *path, const char *arg, …);
int execlp(const char *file, const char *arg, …);
int execle(const char *path, const char *arg, …,
           char * const envp[]);
int execv(const char *path, char * const argv[]);
int execvp(const char *file, char * const argv[]);
int execve(const char *filename, char * const argv[],
           char * const envp[]);
```

l – аргументы передаются списком
v – аргументы передаются массивом
p – поиск файла по полному пользовательскому пути
e – создание нового окружения

Системный вызов **exec()**

```
#include <unistd.h>

const char * args[] = {“vi”, “/home/user/test.txt”, NULL};
int ret;

ret = execv(“/bin/vi”, args);
if(ret == -1)
{
      perror(“execv”);
}
```

Завершение процесса

- ✓Выполнение работающим процессом системного вызова, предназначенного для завершения процесса
- ✓Уничтожение другим процессом
- ✓Возникновение исключительной ситуации

Системный вызов **exit()**

```
#include <stdlib.h>
void exit(int status);
```

```
exit(EXIT_SUCCESS); // 0
exit(EXIT_FAILURE); // -1 или 1
```

- **status** – статус процесса завершения
- компилятор языка Си помещает вызов функции **exit()** в код, который выполняется после возврата из **main()**

Системный вызов **exit()**