# Системные вызовы для управления процессами

## или
## Как размножаются процессы

## или
## Немного некромантии



# Определение и состав процесса

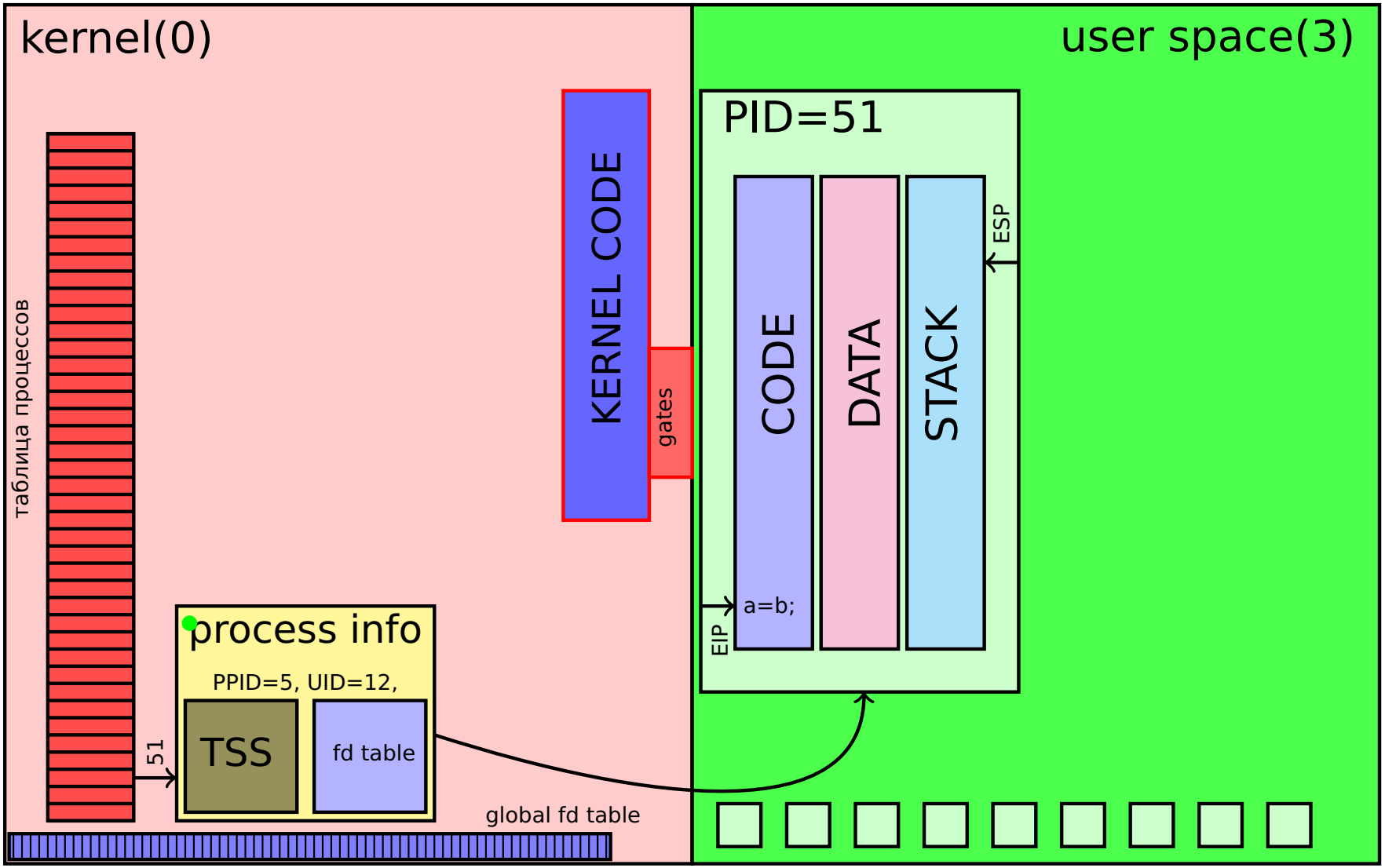
## Определение

Программа, загруженная в оперативную память и выполняющаяся (или способная выполняться).

## Состав

- сегменты кода, данных стека;
- состояние задачи (TSS, files);
- информация о процессе (PID, PPDI, *UID, *GID, times);

# Иллюстрация — 1 процесс



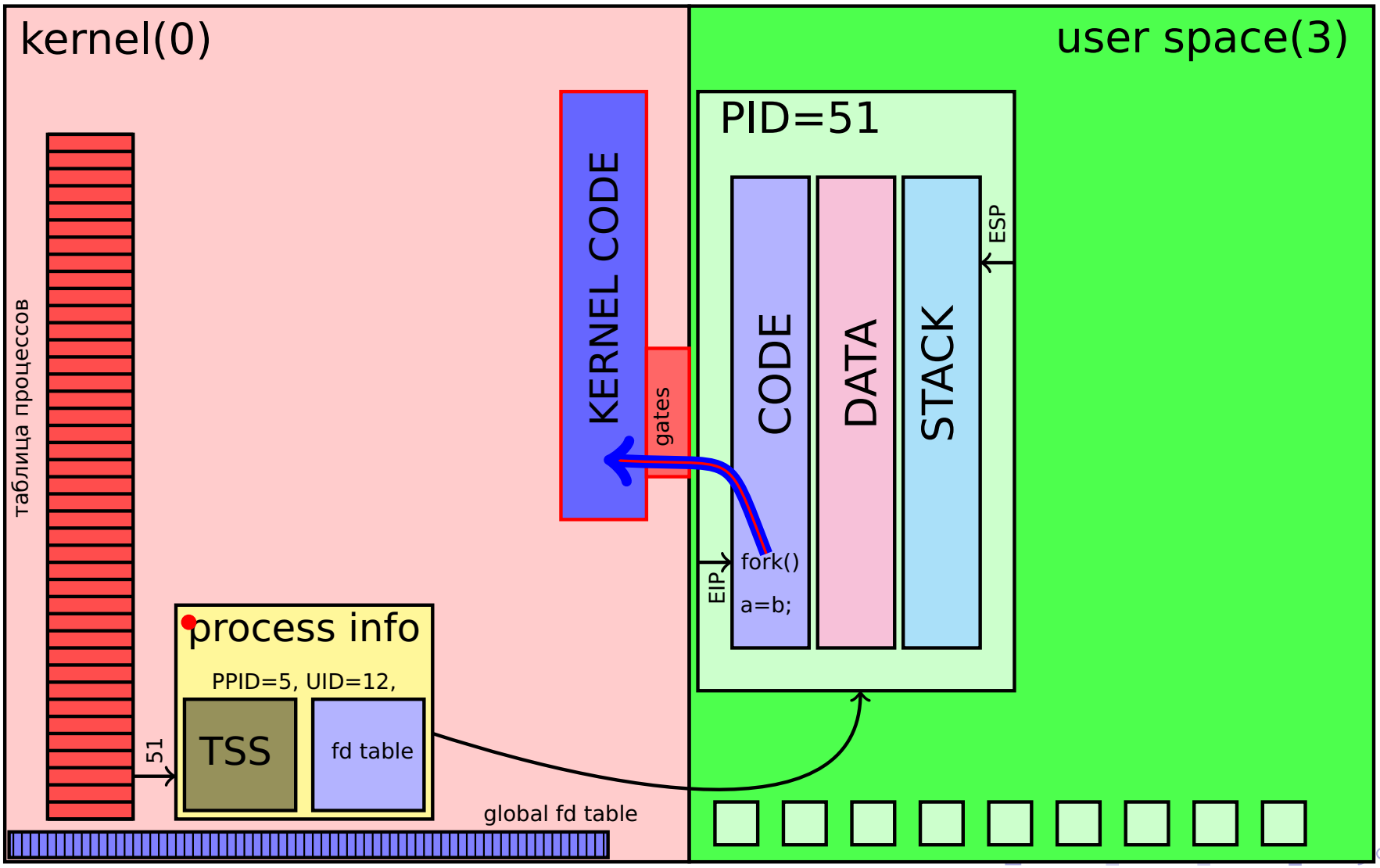
# Системный вызов fork
## Создание нового процесса

Для создания нового процесса используется системный вызов **fork**.

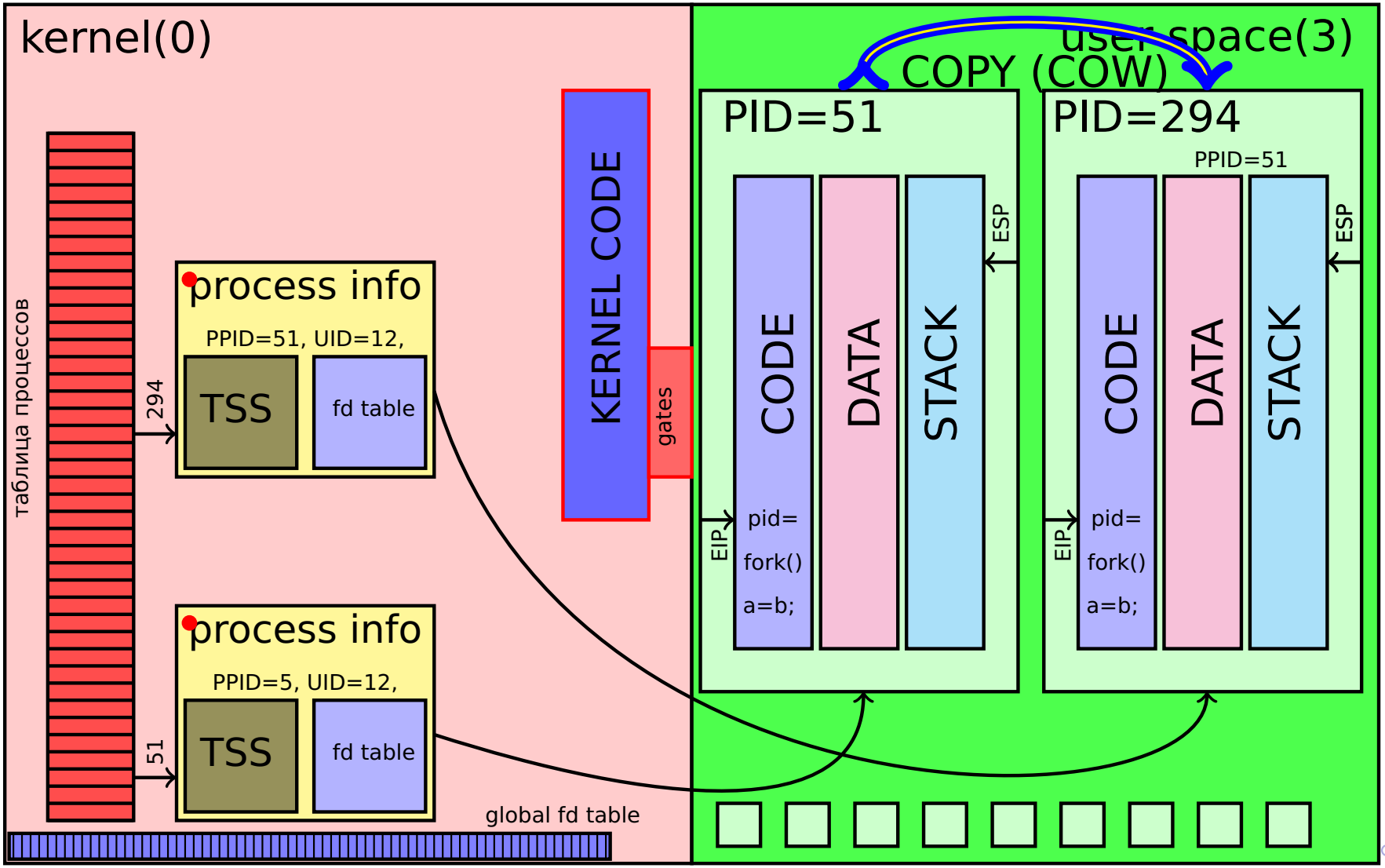
```
#include <unistd.h>

  pid_t fork(void);
```

В случае успеха создаёт новый (дочерний) процесс. В случае ошибки возвращает $-1$.
Синоним типа **pid_t** — это другое имя для целочисленного типа, подходящего для хранения идентификатора процесса (например, unsigned short).

# Иллюстрация — начало системного вызова fork



# Иллюстрация — Работа системного вызова fork



# Свойства процесса-потомка

## Образ процесса

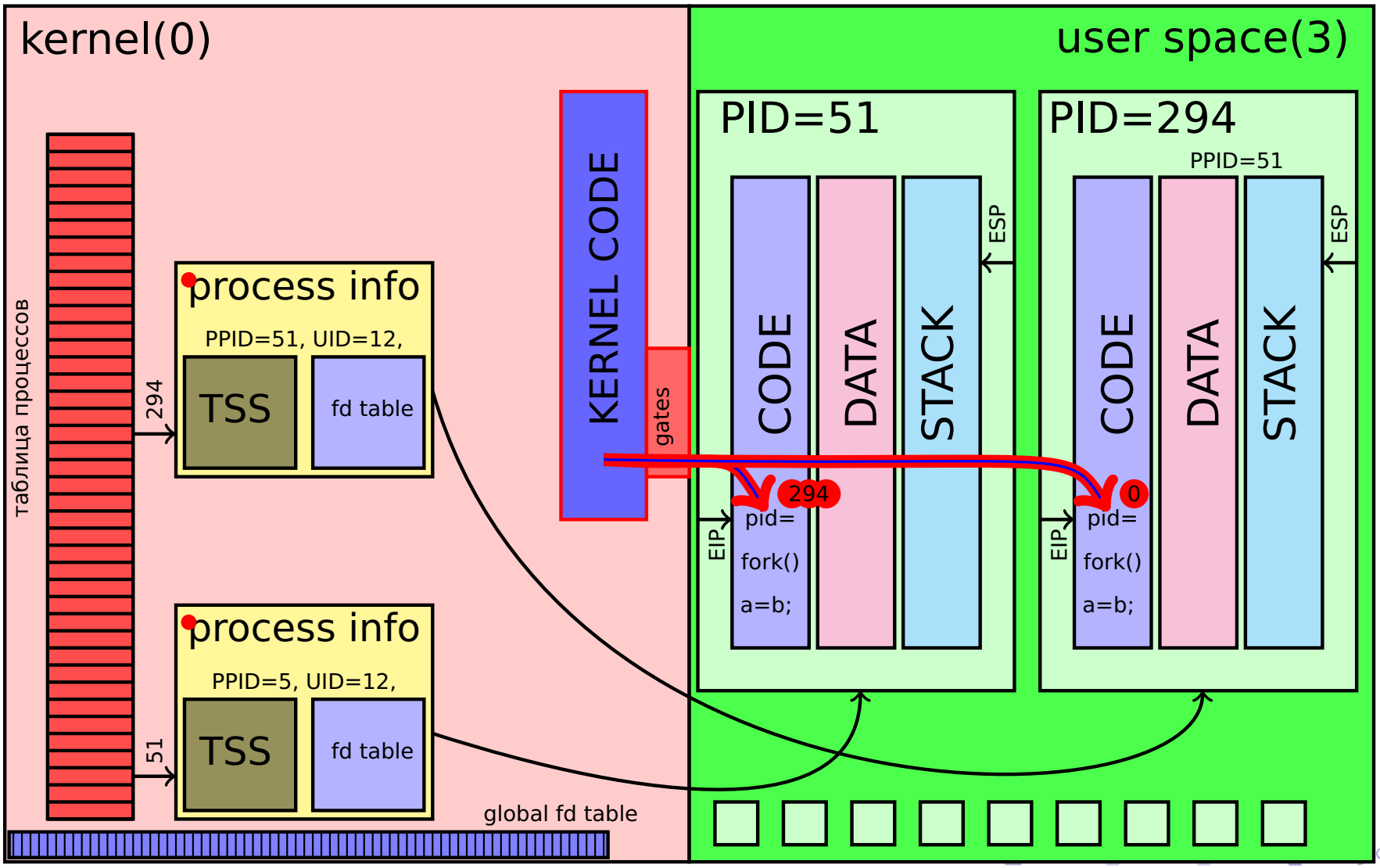
Потомок получает **копию** образа (всех сегментов) процесса-родителя. Таким образом, потомок выполняет ту же программу, что и родитель, и в том же месте. Изменения в сегментах потомка не влияют на родителя, и наоборот.

Потомок получает новый, уникальный PID. Значение PPID (parent PID) устанавливается в значение PID родителя. У потомка сбрасываются значения счётчиков ресурсов, использованного времени, блокировки памяти, семафоры, список необработанных сигналов ...

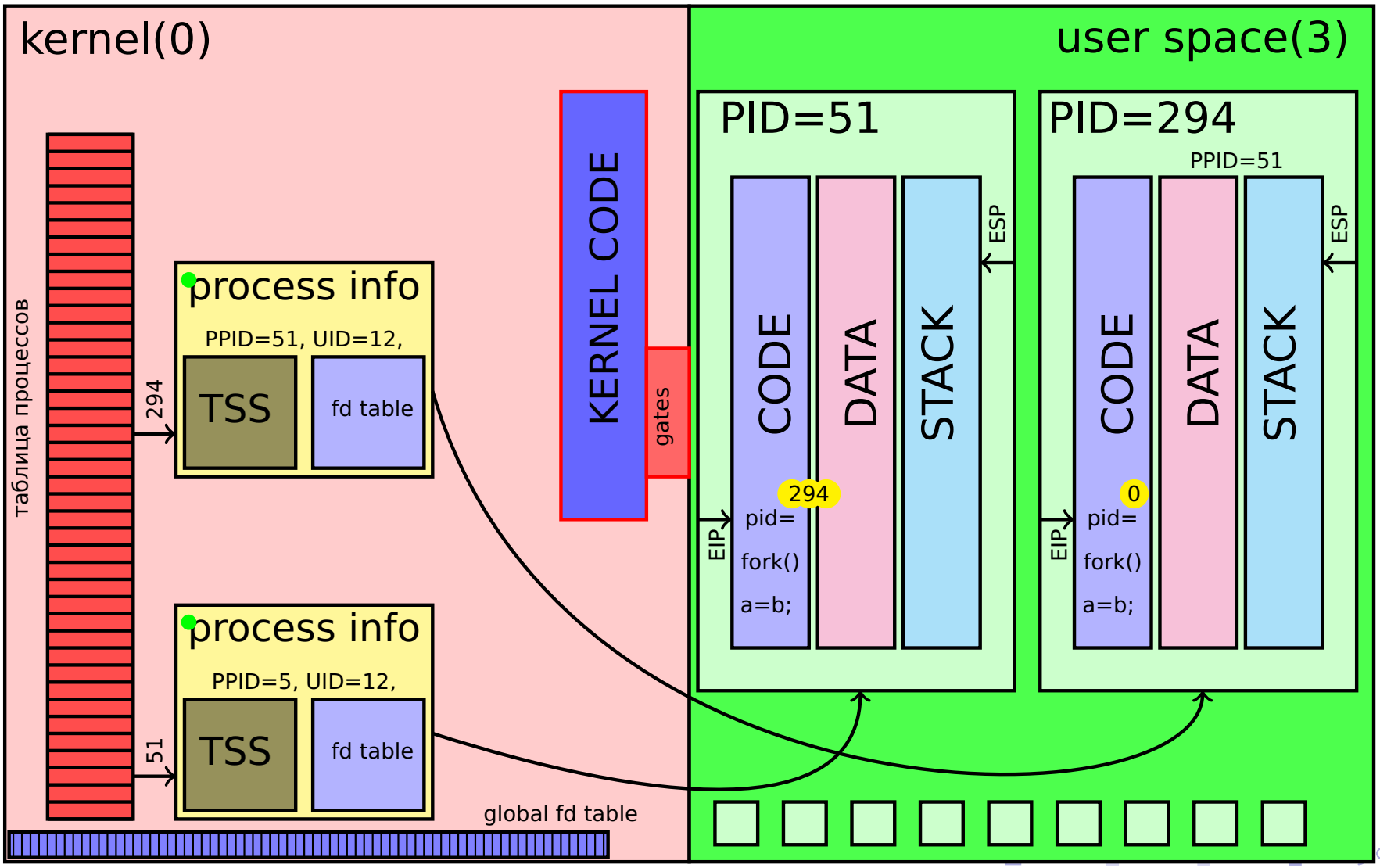
## Унаследованные ресурсы и параметры

По наследству, помимо образа процесса, достаются открытые файлы и каталоги. Наследуются параметры *UID, *GID, текущий каталог и корень ФС, ограничения и возможности.

# Иллюстрация — Возврат из системного вызова fork



# Иллюстрация — продолжение работы процессов



# Возвращаемое значение fork

В случае ошибки возвращает $-1$. Код причины — в errno.

## Два возвращаемых значения

В случае успеха происходит возврат из системного вызова fork сразу в **2** процесса: и в родитель, и в потомок. Причём состояние сегментов кода, данных и стека — одинаковое.

Определять, кто родитель, а кто потомок, следует по **возвращаемому значению** fork:

- потомок получает 0;
- родитель получает PID потомка.

# Завершение процесса
## Системный вызов _exit

Процесс может завершиться двумя способами:

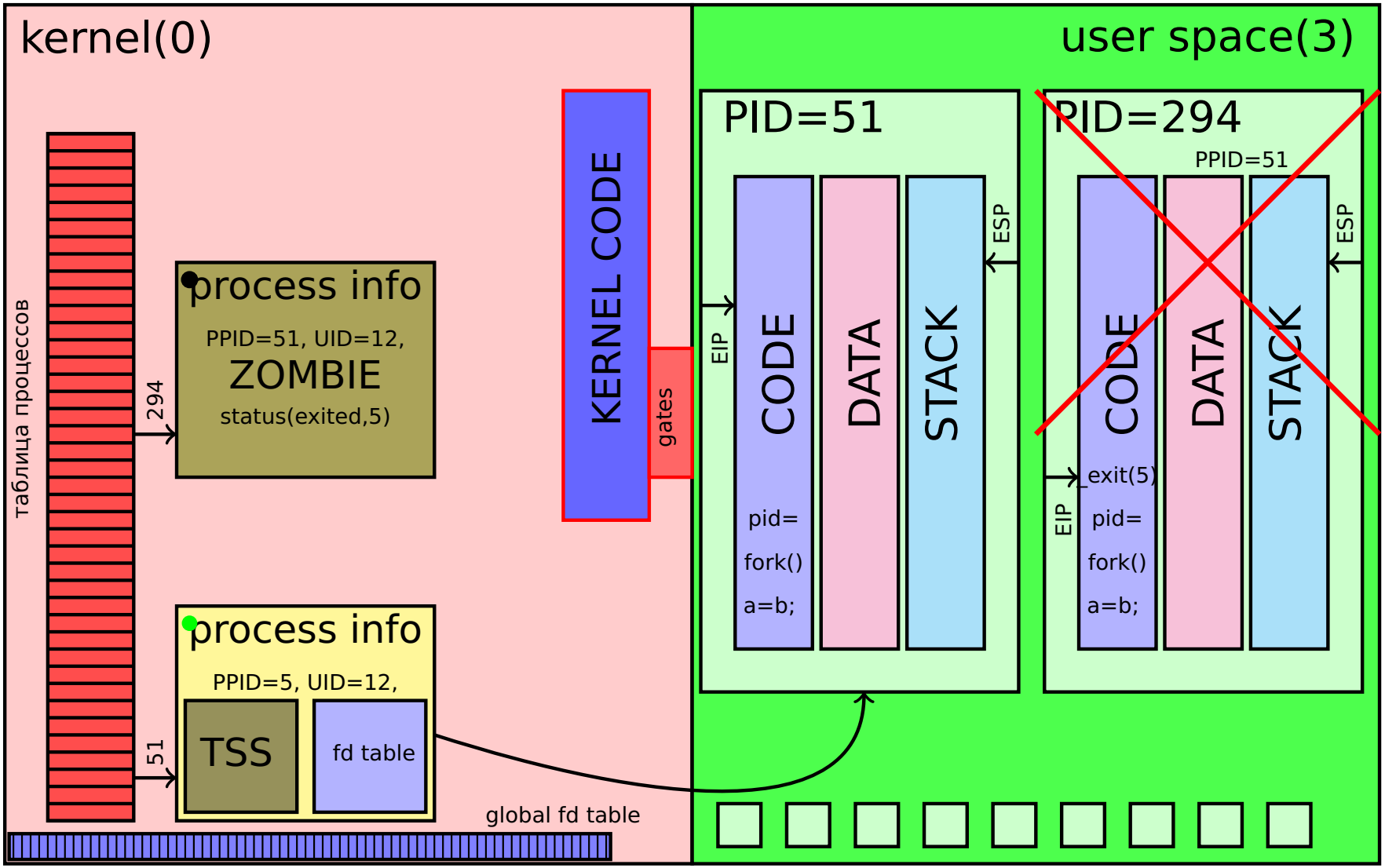
- умереть сам, вызвав системный вызов **_exit** ;
- может быть убит сигналом.

Для самостоятельного завершения процесс вызывает системный вызов **_exit** :

```
void _exit(int status);
```

Параметр **status** — статус завершения программы (8 бит: 0–255).
0 — признак успешного завершения,
не 0 — признак ошибки.

# Иллюстрация — системный вызов _exit



# Системный вызов _exit

При системном вызове _exit

- файлы и каталоги процесса закрываются (буферы высокого уровня теряются);
- память, принадлежащая процессу, освобождается;
- потомки этого процесса получают назначенного родителя с PID=1
- родитель получает сигнал SIGCHLD;
- PID умершего процесса и соответствующая структура ядра остаются – для хранения статуса возврата и значений счётчиков. Такое состояние процесса называется зомби.

# Функция exit и return из main

При выходе по _exit() у процесса нет возможности выполнить завершающие “предсмертные действия”. Поэтому, для выхода чаще используется функция exit():

```
#include <stdlib.h>

void exit(int status);
```

Эта функция сначала вызывает функции, зарегистрированные с помощью функций **atexit** и **on_exit**, необходимые завершающие действия из стандартной библиотеки, и только потом вызывает системный вызов _exit с тем же значением аргумента. Аналогичные действия происходят при возврате из функции main. Поэтому main **всегда должна возвращать int**, причём 0 означает — нет ошибки.

# Системные вызовы wait и waitpid

Для того, что бы дождаться смерти потомка, узнать причину смерти, и уничтожить зомби, используется системый вызов **wait** или одна из его разновидностей:

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

Вызов wait ожидает смерти любого из потомков (блокируя родителя), уничтожает зомби, и записывает статус возврата в целую переменную, адрес которой указан в аргументе status.
Первым аргументом waitpid можно указать, завершения какого из потомков мы ожидаем (>1=PID, 0 — потомок из моей группы, $-1$ — любой из потомков, $< -1$ — потомок из заданной группы). Третий аргумент может быть 0 (ждём), или WNOHANG — вернуть 0, если никто не умер.

# Макросы для разбора статуса завершения

Статус возврата представляет собой набор битовых полей. Для правильного и переносимого разбора статуса следует использовать следующие макросы:

- **WIFEXITED**(status) — возвращает истину (!=0), если процесс завершился сам, вызвав _exit().
- **WEXITSTATUS**(status) — код возврата, переданный вызову _exit (имеет смысл, только если WIFEXITED вернул истину).
- **WIFSIGNALED**(status) — возвращает истину (!=0), если процесс завершился из-за необработанного сигнала (убит.)
- **WTERMSIG**(status) — номер сигнала, которым был убит процесс.

Дополнительную информацию о смерти процесса предоставляют системные вызовы wait3, wait4, waitid.

# Системный вызов exec
## Необходимость

### Потомок — копия родителя

Так как потомок наследует образ процесса-родителя, открытые файла, то он выполняет **ту же программу, что и родитель**.

Иногда это правильно. Например, WEB-сервер размножается до множества экземпляров, и каждый из них обслуживает свой набор клиентов. Но чаще всего потомка создают для того, что бы “направить его на другую работу” – выполнять **другую** программу.

### Системный вызов для выполнения другой программы

Для того, что бы выполнить другую программу в рамках того же процесса, используется системный вызов **exec** , доступный программисту в виде функции **execve**.

# Системный вызов exec
## Прототип

Системный вызов **exec** описывается следующим образом:

```
#include <unistd.h>

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

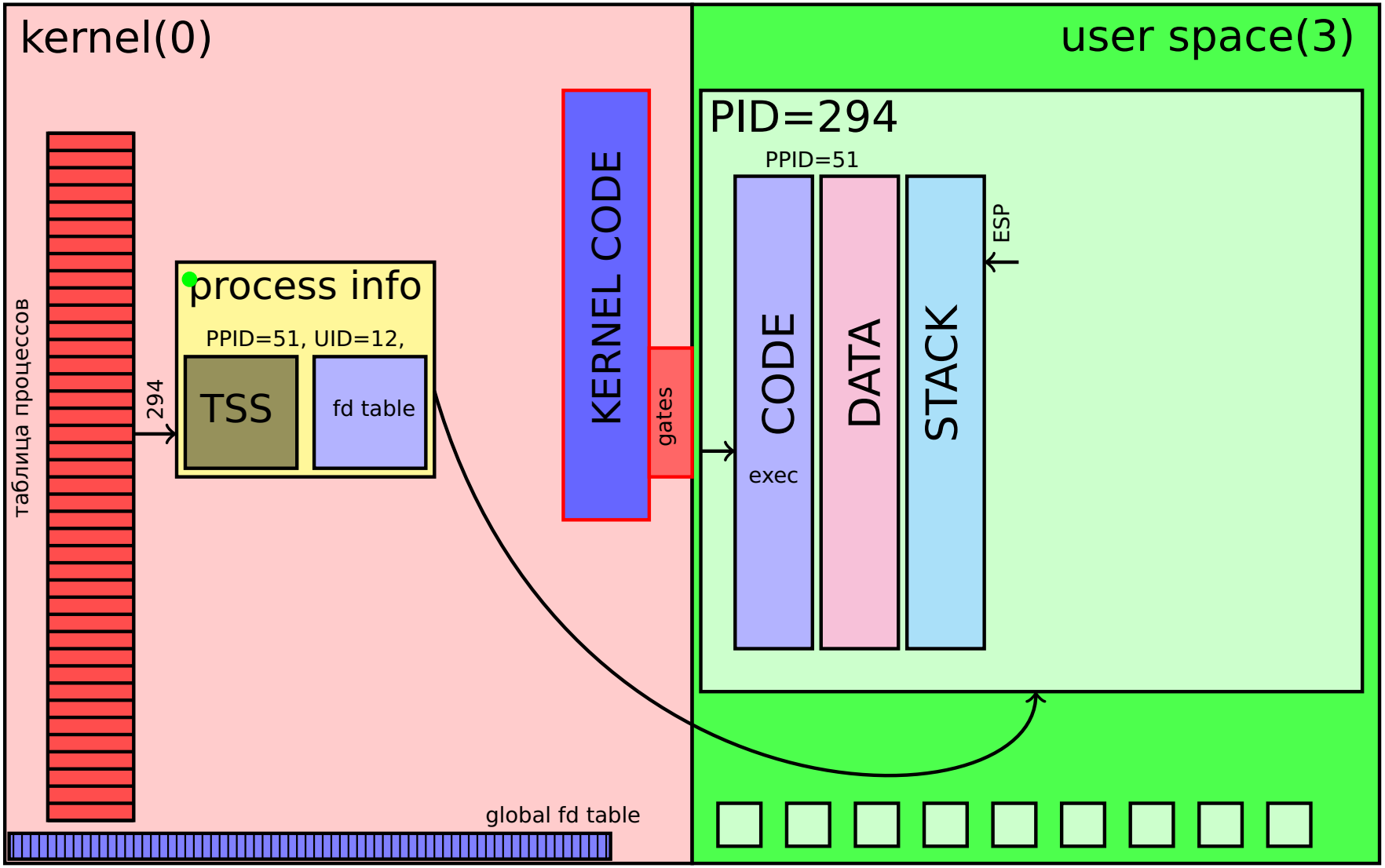
filename — файл программы, которую надо выполнить.

argv — массив аргументов, передаваемых программе. Именно этот массив программа получает в качестве своего argv. По соглашению, нулевой элемент (argv[0]) — имя программы. За последним аргументом надо передать нулевой указатель (NULL) – по нему вычисляется argc.

envp — список переменных окружения, например "TERM=vt100", "PATH=/usr/bin:/bin:/sbin", NULL

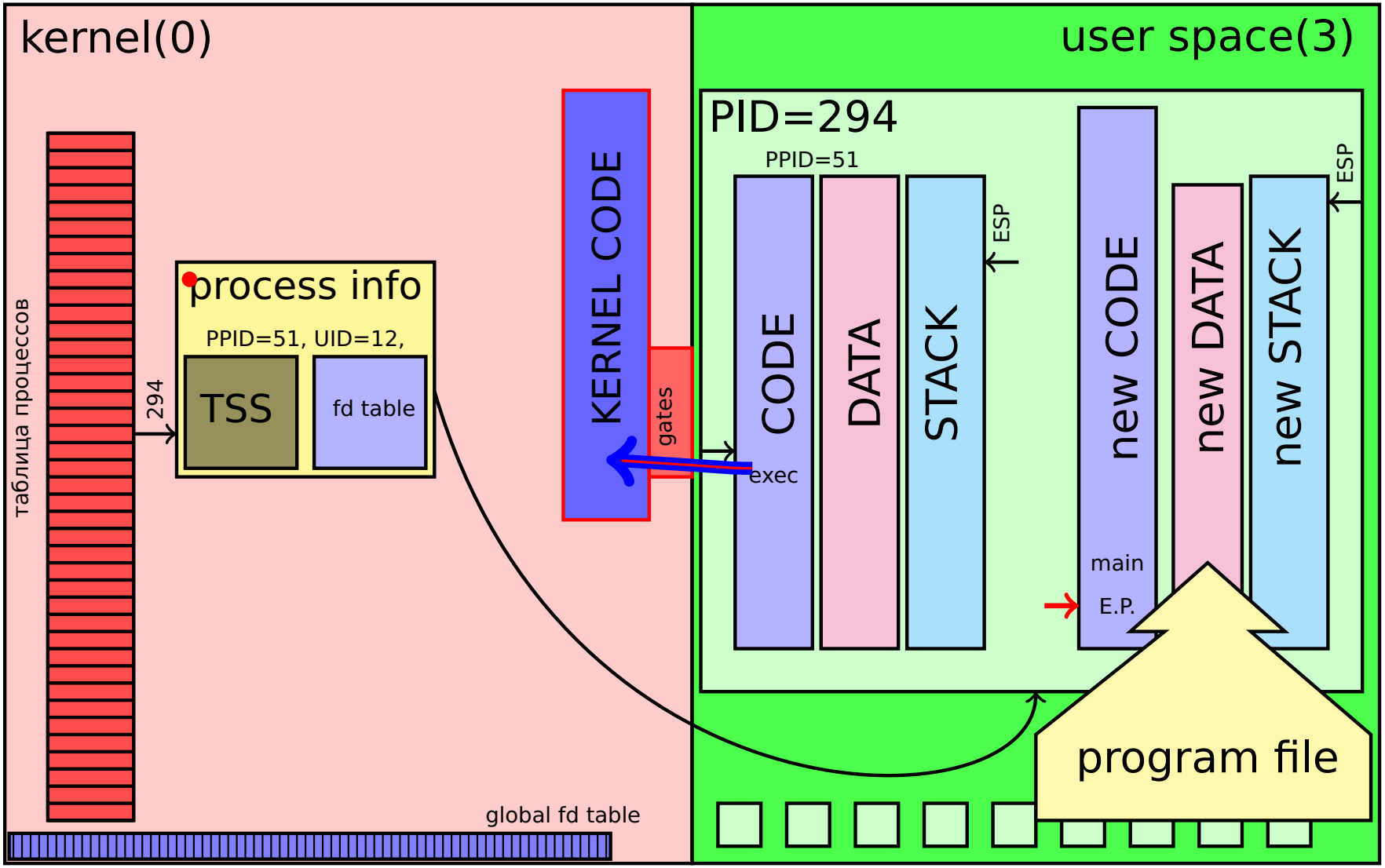
# Системный вызов exec

## Состояние до вызова

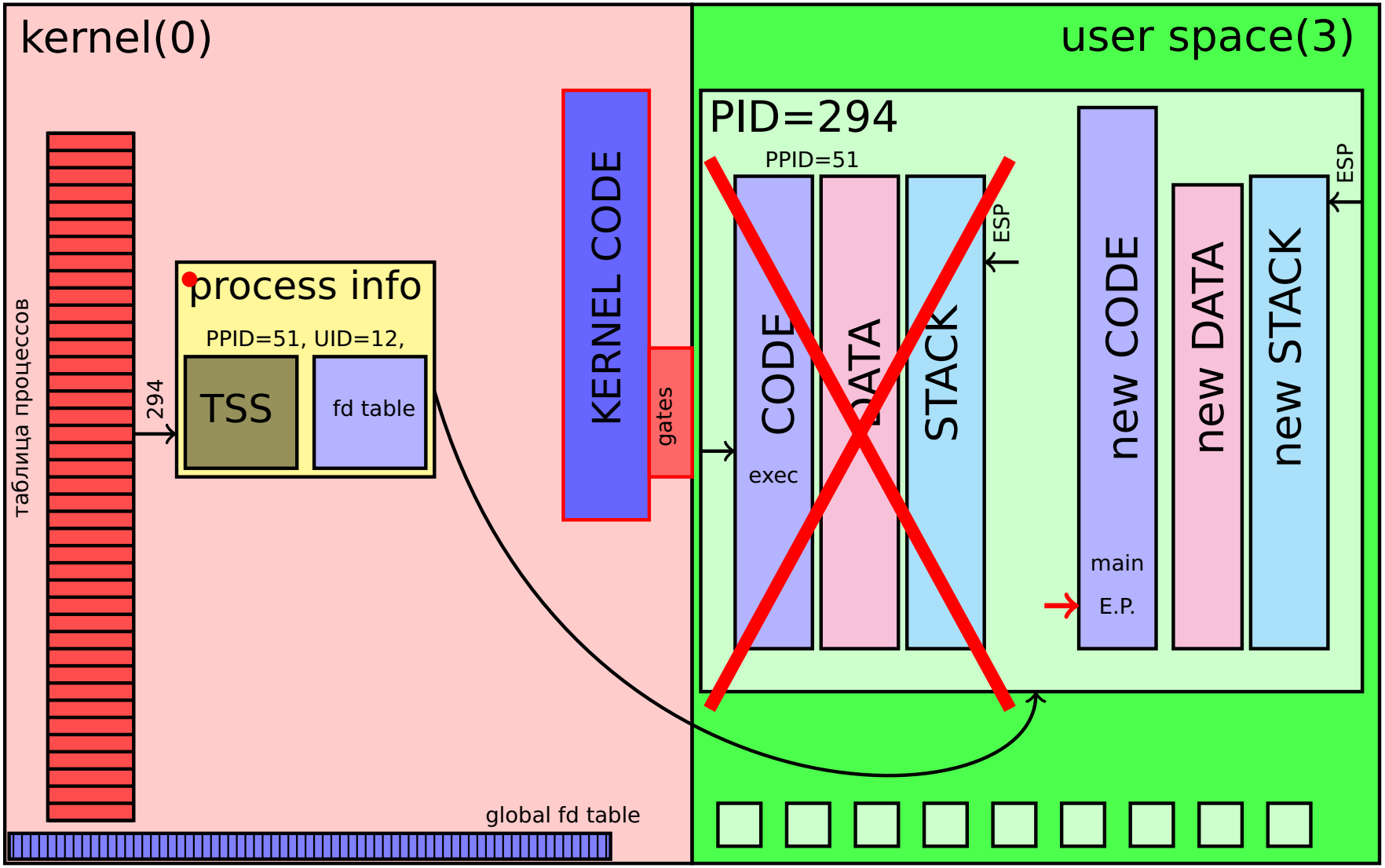


# Системный вызов exec

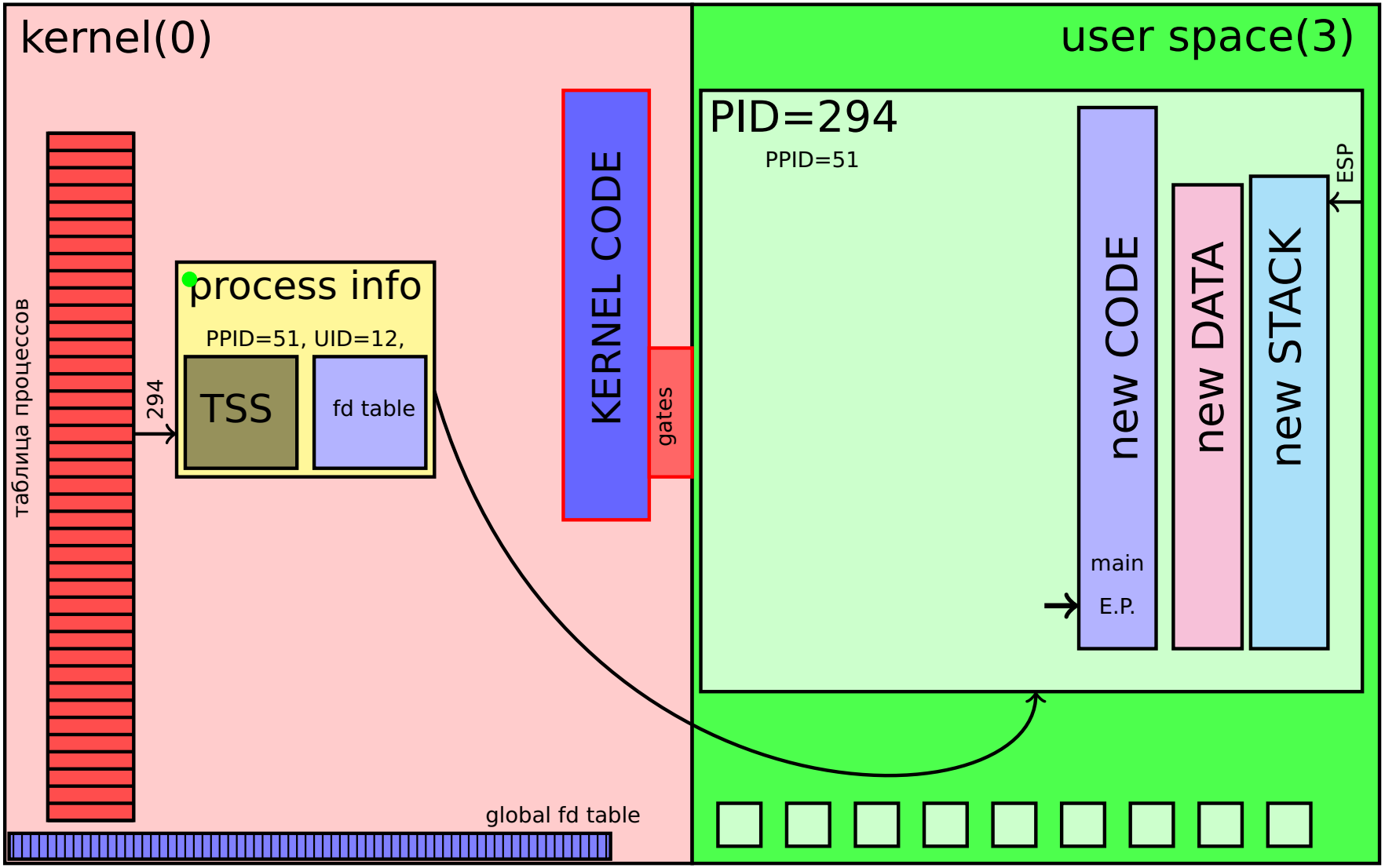
## Работа вызова 1



# Системный вызов exec
## Работа вызова 2



# Системный вызов exec
## Завершение вызова



# Системный вызов exec
## Возвращаемое значение

Если происходит ошибка (нет файла, не исполняемый файл, неправильный интерпретатор), то **exec** возвращает $-1$, а в errno помещается код ошибки.

### Возвращаемое значение в случае успеха

В случае успеха старый код, куда можно было бы выполнить возврат, **НЕ СУЩЕСТВУЕТ** — вместо него выполняется код другой программы.
Поэтому в случае успеха exec **НИЧЕГО НЕ ВОЗВРАЩАЕТ** — некуда.

# Состояние процесса после вызова exec

- Образ процесса — новый, определяется заданной программой.
- Выполнение начинается за заданной в программе точки входа.
- Файлы остаются открытые, за исключением тех, на которых стоит флаг ’close-on-exec’ (O_CLOEXEC).
- Владелец и группы сохраняются, если не установлены на файле флаги SUID или SGID. Если установлены, и нет других ограничений – эффективный владелец (EUID) или эффективная группа (EGID) устанавливаются в соответственно владельца или группу файла.
- Обработчики сигналов устанавливаются в значения по умолчанию.
- Отображения файлов в память снимаются.
- Общие сегменты памяти отсоединяются.
- Таймеры, семафоры, очереди, блокировки памяти, открытые каталоги — не сохраняются.

# Исполняемые файлы

Исполняемые файлы — это те, которые разрешено выполнять данному пользователю.
Программы могут быть:

- бинарным исполняемым файлом (статической или динамической компоновки).
- бинарным файл другой О.С. (wine, dosemu)
- программой на интерпретируемом языке. В этом случае первой строкой файла специальным комментарием указывается путь к интерпретатору, например:
  #!/usr/bin/perl
  #!/bin/bash
  #!/usr/bin/python
- программой для стандартного командного интерпретатора (обычно /bin/sh).

# Другие интерфейсные функции к exec

Вызов в форме **execve** даёт максимальные возможности, но в большинстве простых случаев неудобен. Поэтому есть еще набор интерфейсных функций и этому системному вызову:

```
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[]);
```

Все они начинаются с “exec”, все выполняют одну и ту же основную задачу, но способы поиска программ и передачи аргументов — разные.
Отличия в интерфейсе этих функций зашифрованы в символах суффикса имени — “l”, “v”, “p”, “e”.

# Другие интерфейсные функции к exec
## Варианты вызовов (l/v)

Если с суффиксе программы есть “v”, то аргументы программе надо передавать в виде массива указателей на строки C. А если “l” — в виде списка аргументов. Пример:

```
execl( "/bin/ls", "/bin/ls", "-l", "/", NULL );

char *arg[] = { "/bin/ls", "/bin/ls", "-l", "/", 0 };

execv( arg[0], arg );
```

Варианты с “l” следует использовать, когда количество аргументов известно заранее, и нет готового массива строк. В противном случае следует выбирать функции с символом “v”.

# Другие интерфейсные функции к exec
## Варианты вызовов

Если с суффиксе программы есть “p”, и в первом аргументе не указан путь, то программа ищется в каталогах, указанных в переменной окружения **PATH** .
Иначе — путь должен быть указан.
Пример:

```
execl( "/bin/ls", "/bin/ls", "-l", "/", NULL );
execl( "./laba5", "./laba5", "file1",
       "123", "0x12", NULL );
execlp( "ls", "ls", "-l", "/", NULL );

execvp( "vim", my_args );
execv( "/usr/bin/vim", my_args );
```

Символ “e” обозначает, что можно задать переменные окружения. Функции без этого символа передают текущий набор переменных.

# Вспомогательные системные вызовы и функции

## getpid, getppid

Довольно часто необходимо узнать свой PID и PPID (PID родителя)

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

pid_t getpid(void);

pid_t getppid(void);
```

Вызов getpid возвращает PID процесса, а getppid — PID его родителя.

# Вспомогательные системные вызовы и функции

## Работа с файловыми дескрипторами

Перед запуском программы в рамках того же процесса можно делать манипуляции с файловыми дескрипторами:

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
#define _GNU_SOURCE
#include <unistd.h>
int dup3(int oldfd, int newfd, int flags);
```

Все эти вызовы делают создают и возвращают новый файловый дескриптор — копию старого **oldfd**.
При этом **dup** использует первый свободный номер, **dup2** — указанный (newfd), при необходимости закрывая newfd, **dup3** дополнительно позволяет установить флаги.

# Resume

- Процесс-родитель может создать потомка с помощью **fork**.
- Потомок — копия родителя, но отдельный процесс.
- fork возвращает родителю PID потомка, потомку — 0.
- Процесс может завершиться сам (_exit, exit, return из main), передав код завершения. При этом процесс становится “зомби”.
- Родитель может узнать причину завершения потомка (wait*), одновременно “упокоив зомби”.
- Процесс может выполнить другую программу (exec*), при этом новый процесс не порождается.

cow,mid