

ПО сетевых устройств

Трещановский Павел Александрович, к.т.н.

14.03.20

Состояния процесса

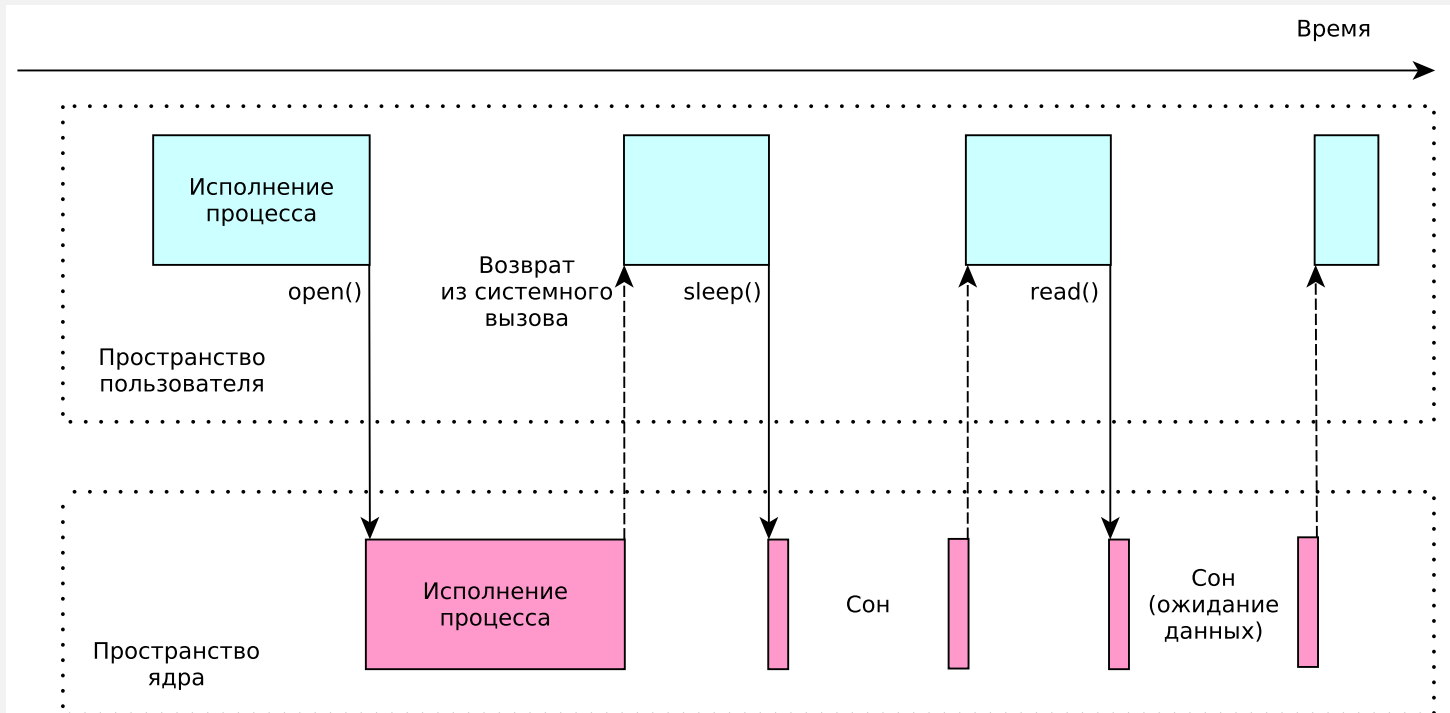
Основные состояния:

- TASK_RUNNING (буква R в выводе ps) - процесс готов к исполнению на процессоре,
- TASK_INTERRUPTIBLE (S) - прерываемое ожидание события,
- TASK_UNINTERRUPTIBLE (D) - непрерываемое ожидание события.

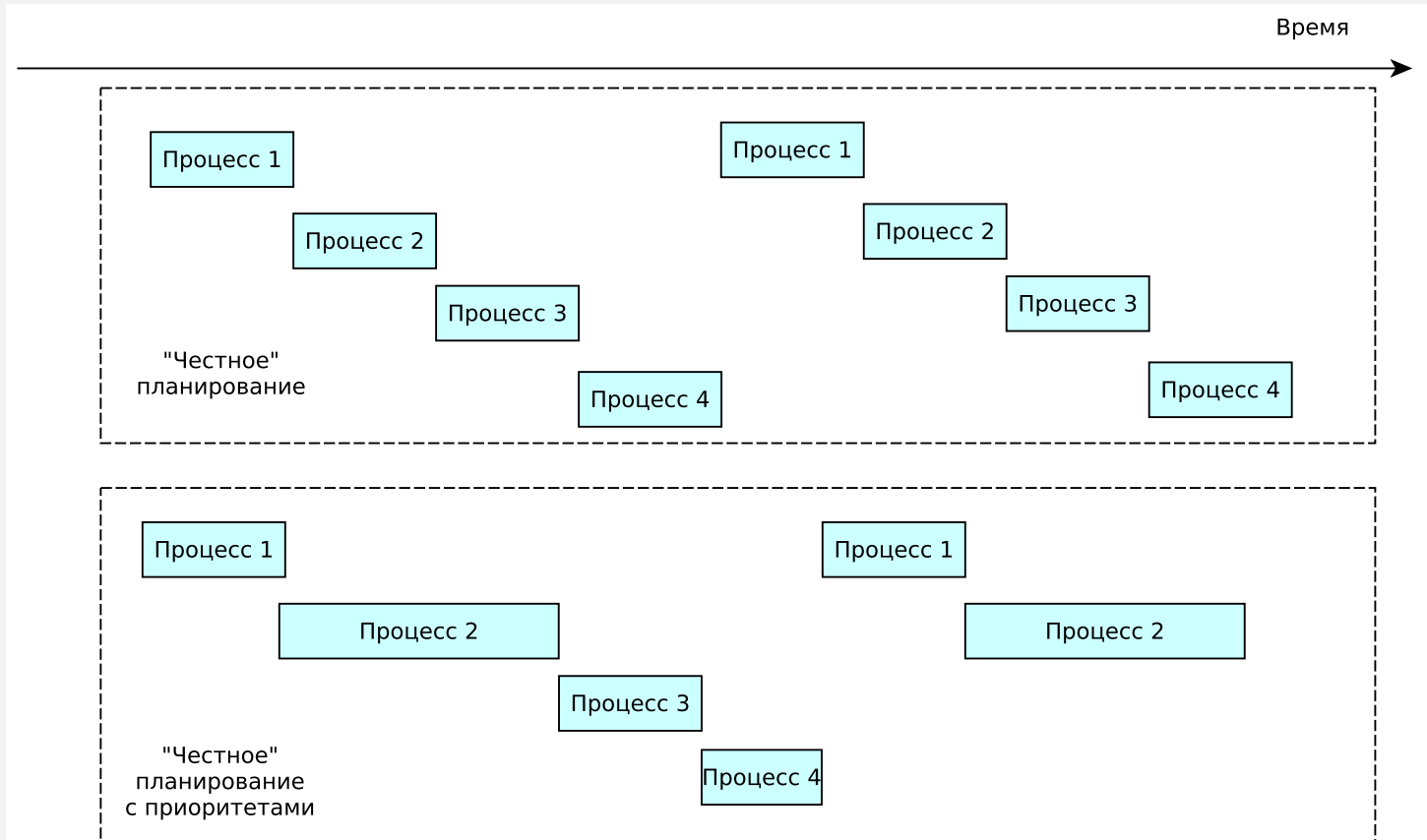
Независимо от состояния процесс может в текущий момент исполняться в

- пространстве ядра (kernel space) или
- пространстве пользователя (user space).

Переход между состояниями



Планировщик процессов



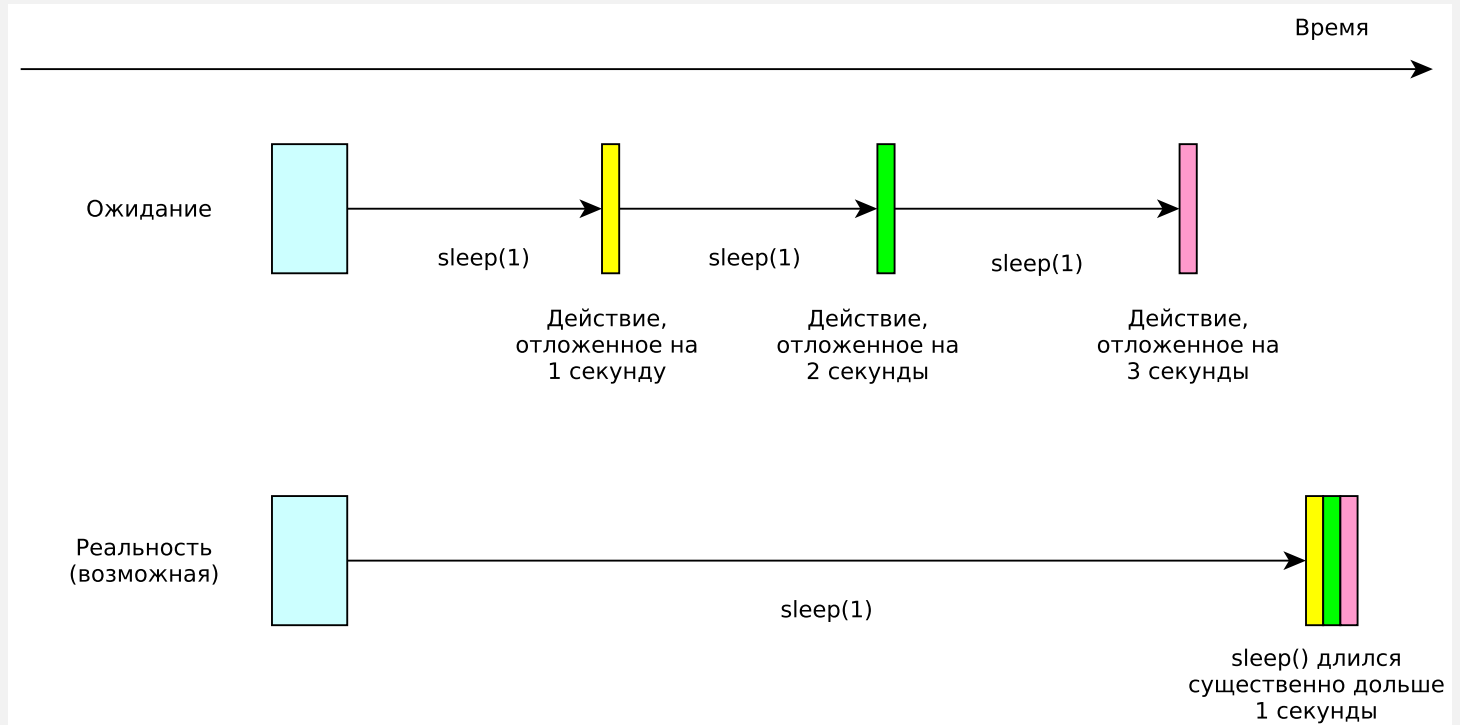
Вытесняющая многозадачность

Когда процесс перестает исполняться:

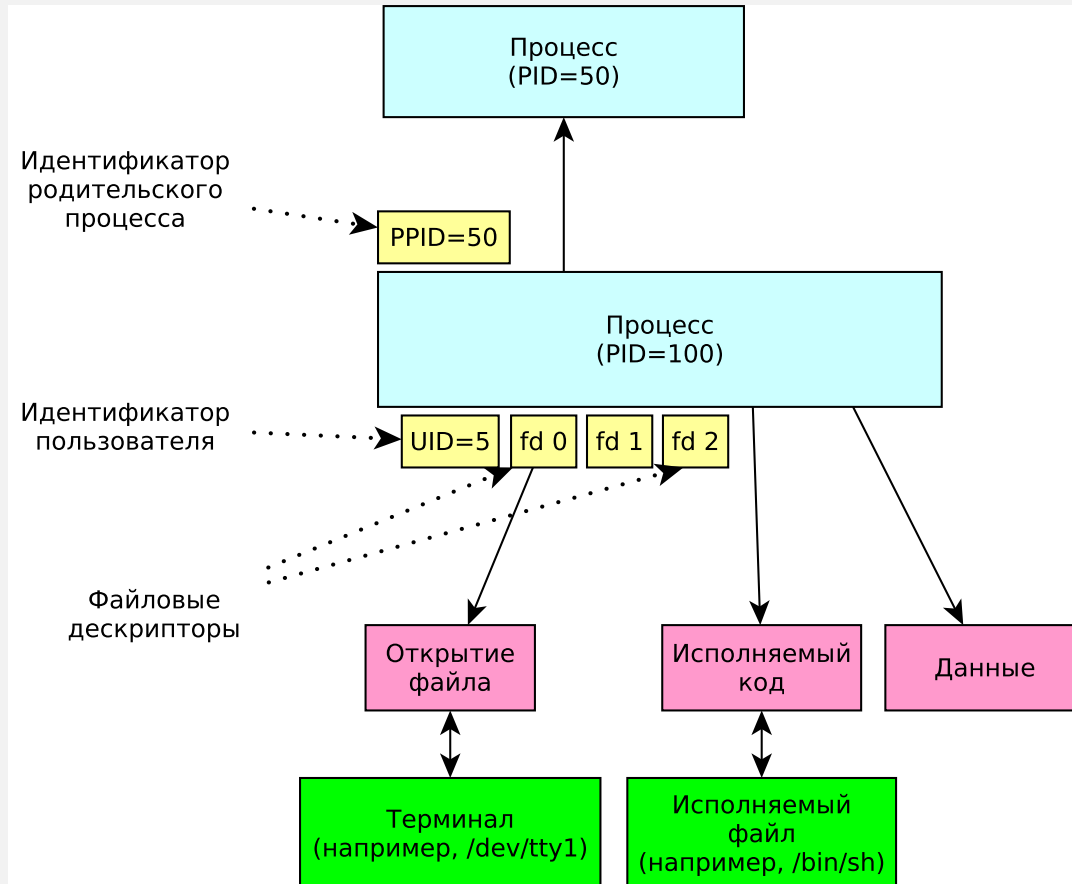
- процесс запрашивает у ядра задержку (например, `sleep()`),
- процесс ожидает завершения операции ввода-вывода (например, `read()`),
- у процесса заканчивается доля времени, выделенная планировщиком,
- в системе появляется более приоритетный процесс.

Блокирующий (системный) вызов - вызов, который переводит процесс в состояние ожидания до завершения операции ввода-вывода. Примеры: `read()`, `write()`.

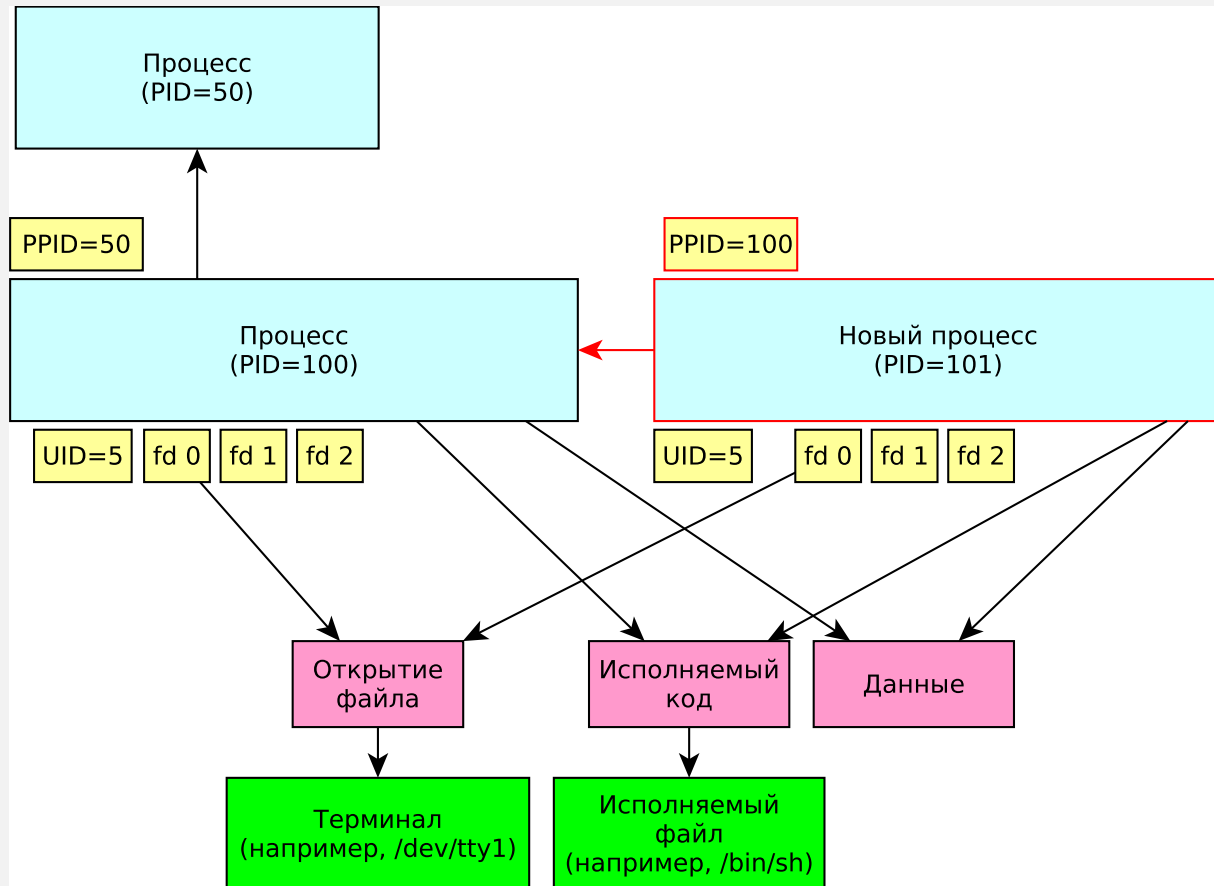
Планирование отложенных действий



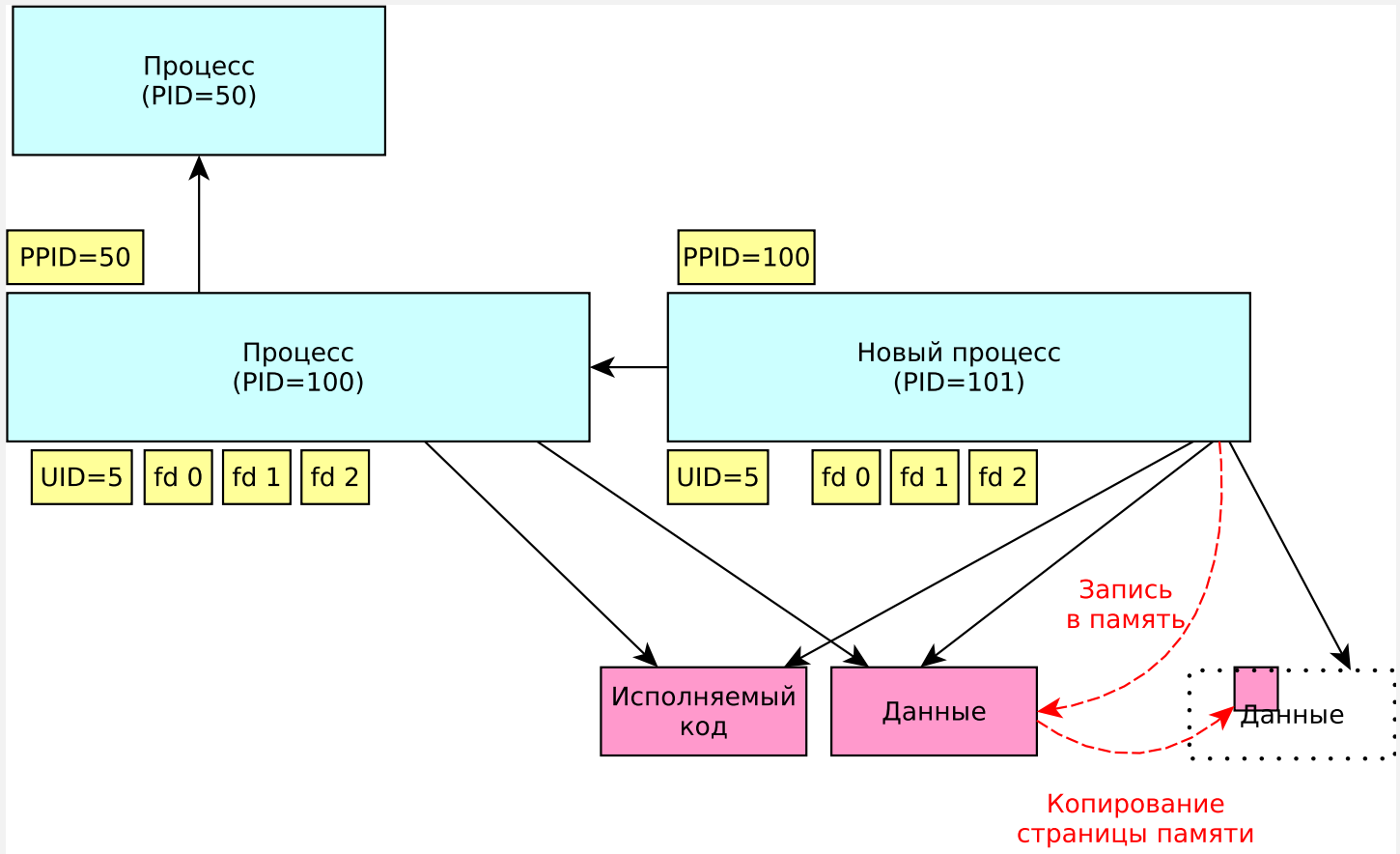
Программное окружение процесса



Создание процесса



Copy on write



Создание процесса, API

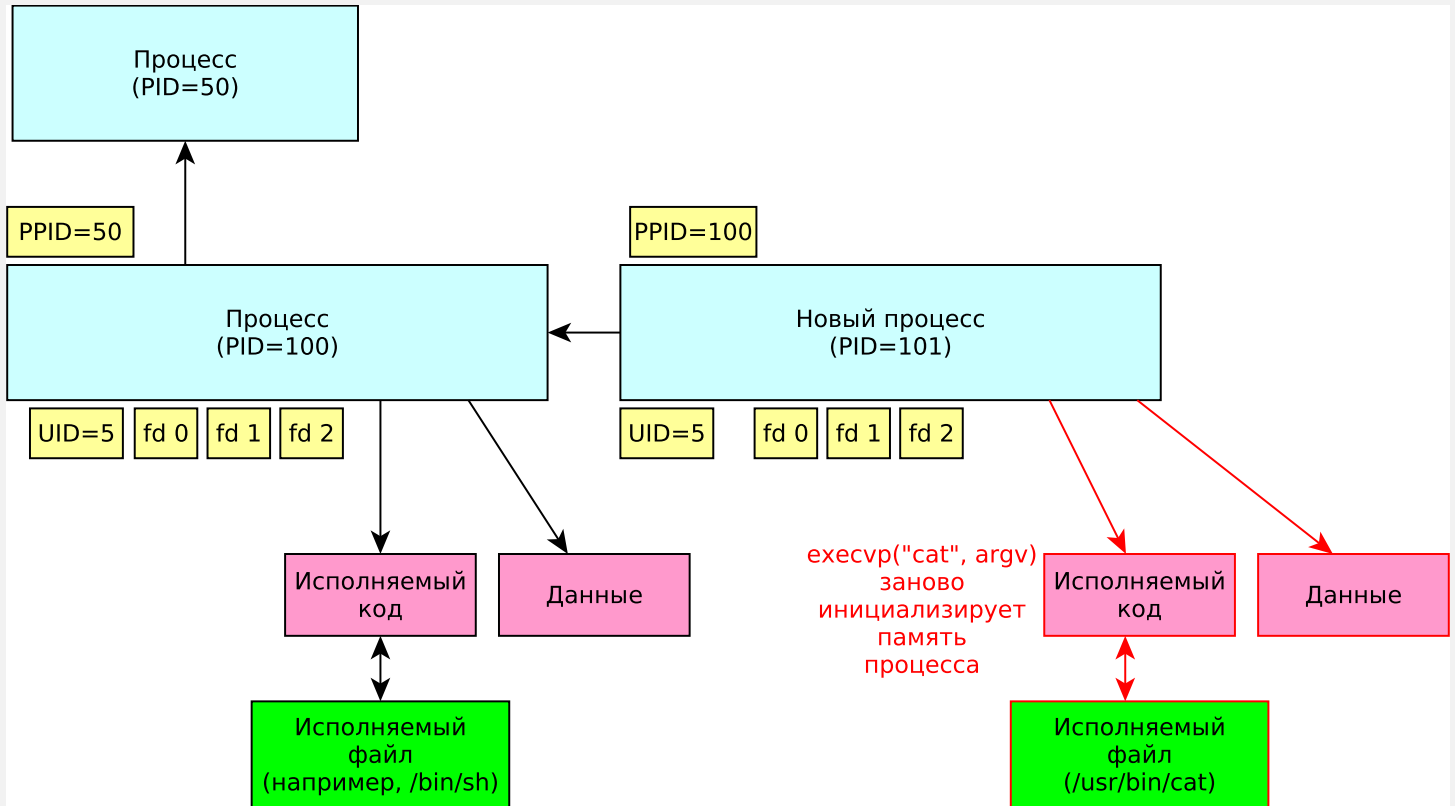
```
pid_t pid; /* pid_t - целое число. */

/* В этой точке выполняется только родительский процесс. */
pid = fork();
if (pid < 0)
    goto on_error;

/* В этой точке выполняется 2 процесса. Текст выведется дважды. */
printf("Hello\n");

if (pid == 0) {
    /* Дочерний процесс. */
} else if (pid > 0) {
    /* Родительский процесс. pid - идентификатор дочернего
    * процесса.
    */
}
}
```

Запуск приложения



Запуск приложения, API

```
char *argv[] = {"/bin/cat", "/home/root/file1", NULL};
int ret;

/* int execvp (const char *file, char *const argv[]); */
ret = execvp(argv[0], argv);

/* В случае успеха программа не возвращается из execvp. */

goto on_error;
```

Преимущество раздельного подхода

```
pid_t pid;

pid = fork();

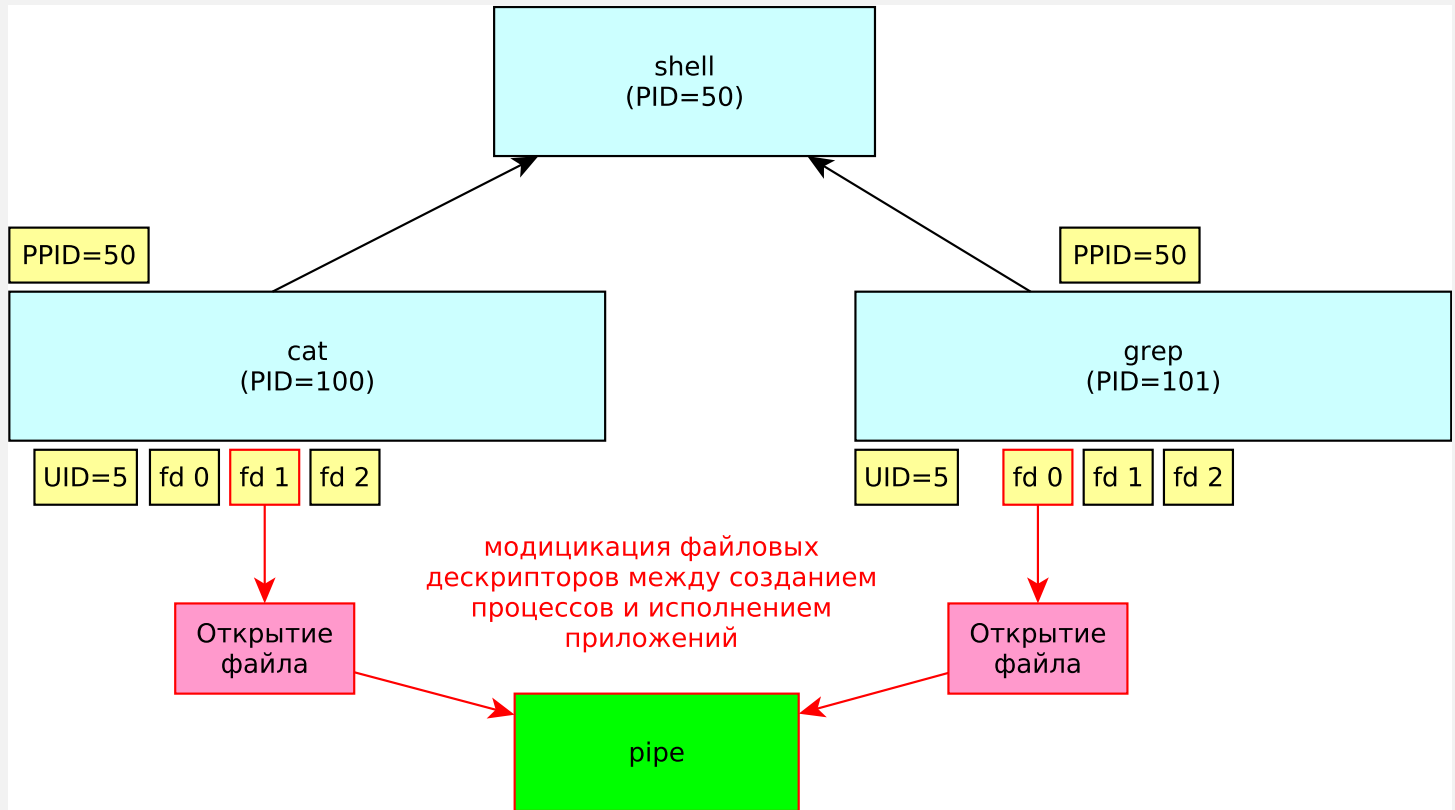
if (pid == 0) {
    /* Дочерний процесс. */

    char *argv[] = {"/bin/cat", "/home/root/file1", NULL};

    /* .....
     * Формирование программного окружения нового процесса:
     * перенаправление в файл,
     * создание конвейера (pipe),
     * открытие устройства ввода-вывода и др.
     * /

    execvp(argv[0], argv);
}
```

Конвейер в shell



Завершение процесса

Возможные причины завершения процесса.

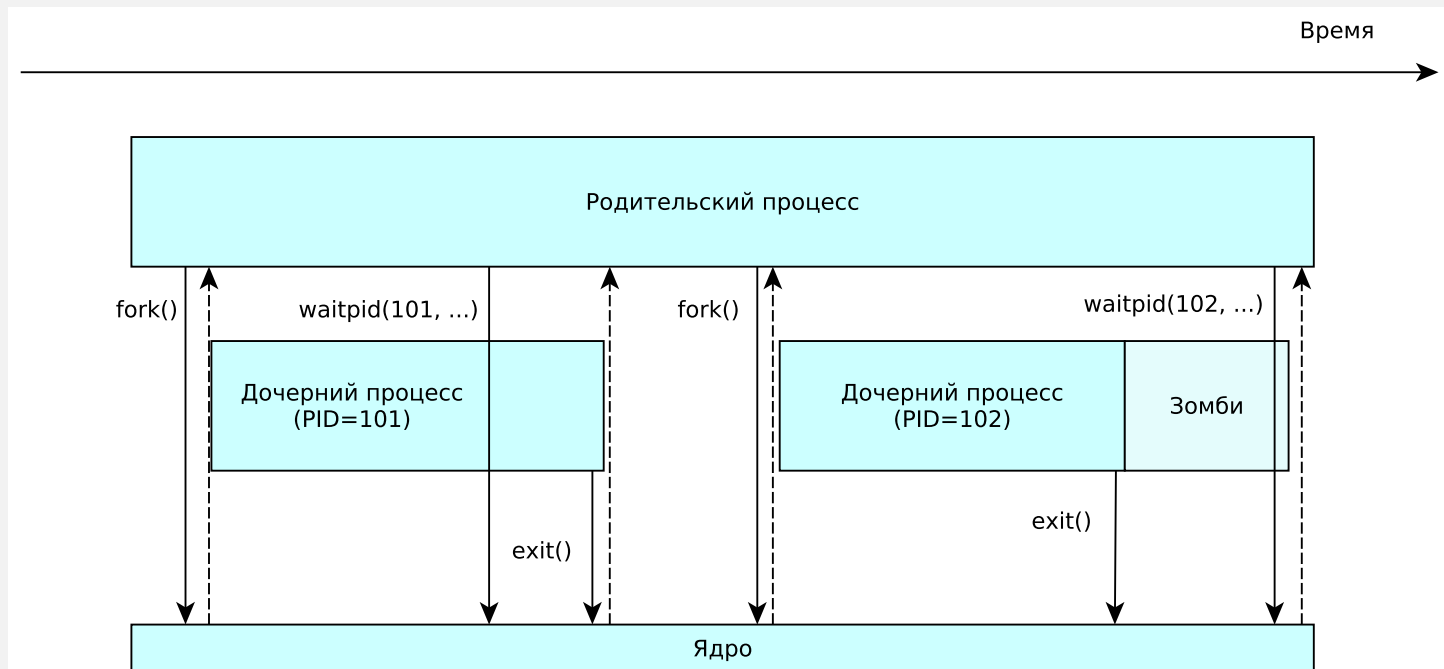
- Вызов функции `exit()` самим процессом, в том числе в результате выхода из функции `main()`.
- Получение *сигнала* от ядра или другого процесса.

Сигнал - специальный механизм межпроцессного взаимодействия. Обычно используется для управления жизненным циклом процесса.

Примеры сигналов:

- нажатие `Ctrl+C` в терминале отправляет сигнал `SIGINT` активному процессу,
- недопустимое обращение к памяти приводит к отправке сигнала `SIGSEGV`,
- исполнение недопустимой машинной инструкции приводит к отправке сигнала `SIGILL`.

Ожидание завершения, процессы-зомби



Ожидание завершения, API

```
pid_t pid;
int child_status;

/* pid_t waitpid(pid_t pid, int *status, int options);
 * Если pid равен -1, функция будет следить за всеми дочерними процессами.
 * Если опции не заданы, функция будет ждать завершения
 * одного из дочерних процессов.
 * Если задана опция WNOHANG, функция завершится сразу.
 */

pid = waitpid(-1, &child_status, WNOHANG);
if (pid == 0) {
    /* Все дочерние процессы продолжают работать. */
    return;
}
if (WIFEXITED(child_status))
    printf("Процесс %d завершился со статусом %d\n",
           pid, WEXITSTATUS(child_status));
else if (WIFSIGNALED(child_status))
    printf("Процесс %d был прерван сигналом %d\n",
           pid, WTERMSIG(child_status));
```

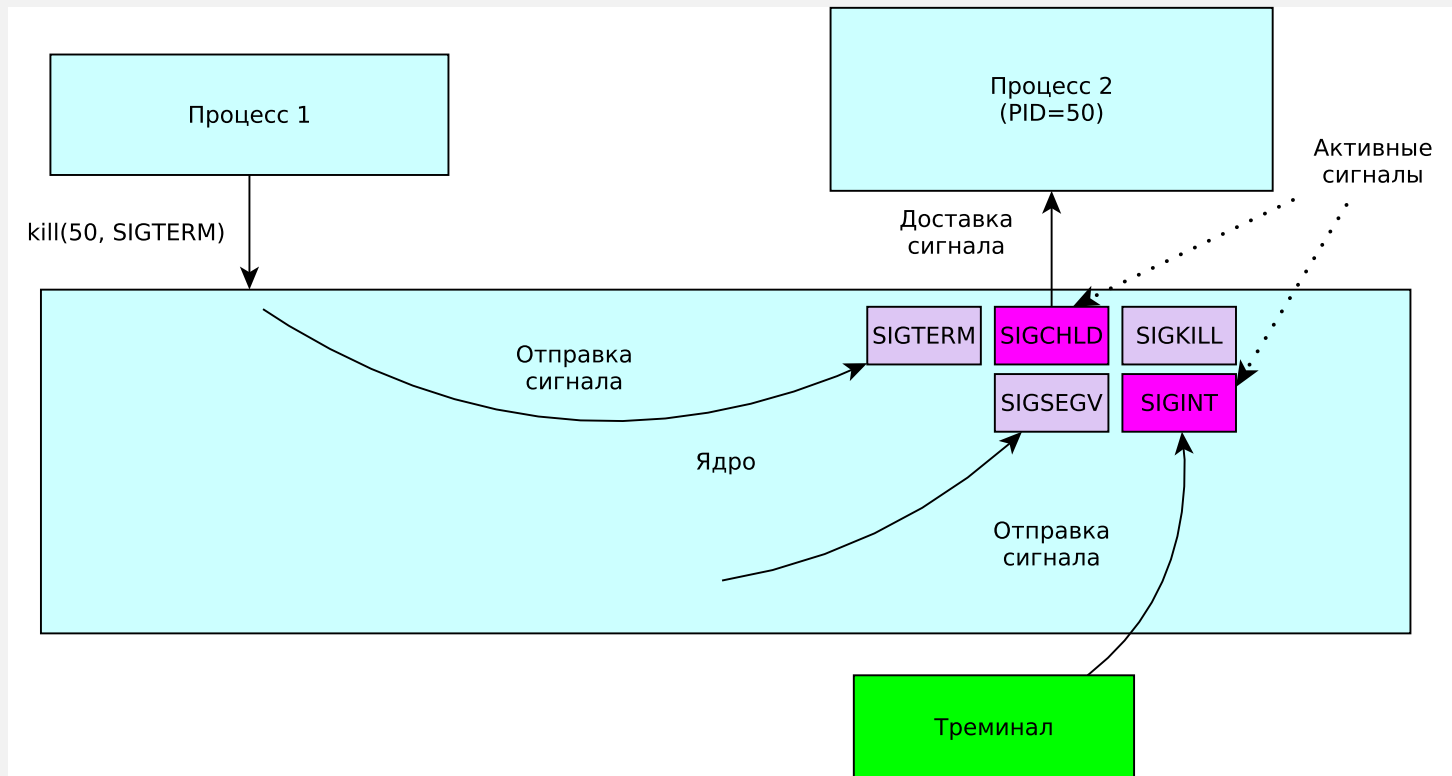
Сигналы, определение, типы

Сигнал - механизм межпроцессного взаимодействия, который прерывает нормальное исполнение процесса для выполнения *обработчика сигнала*.

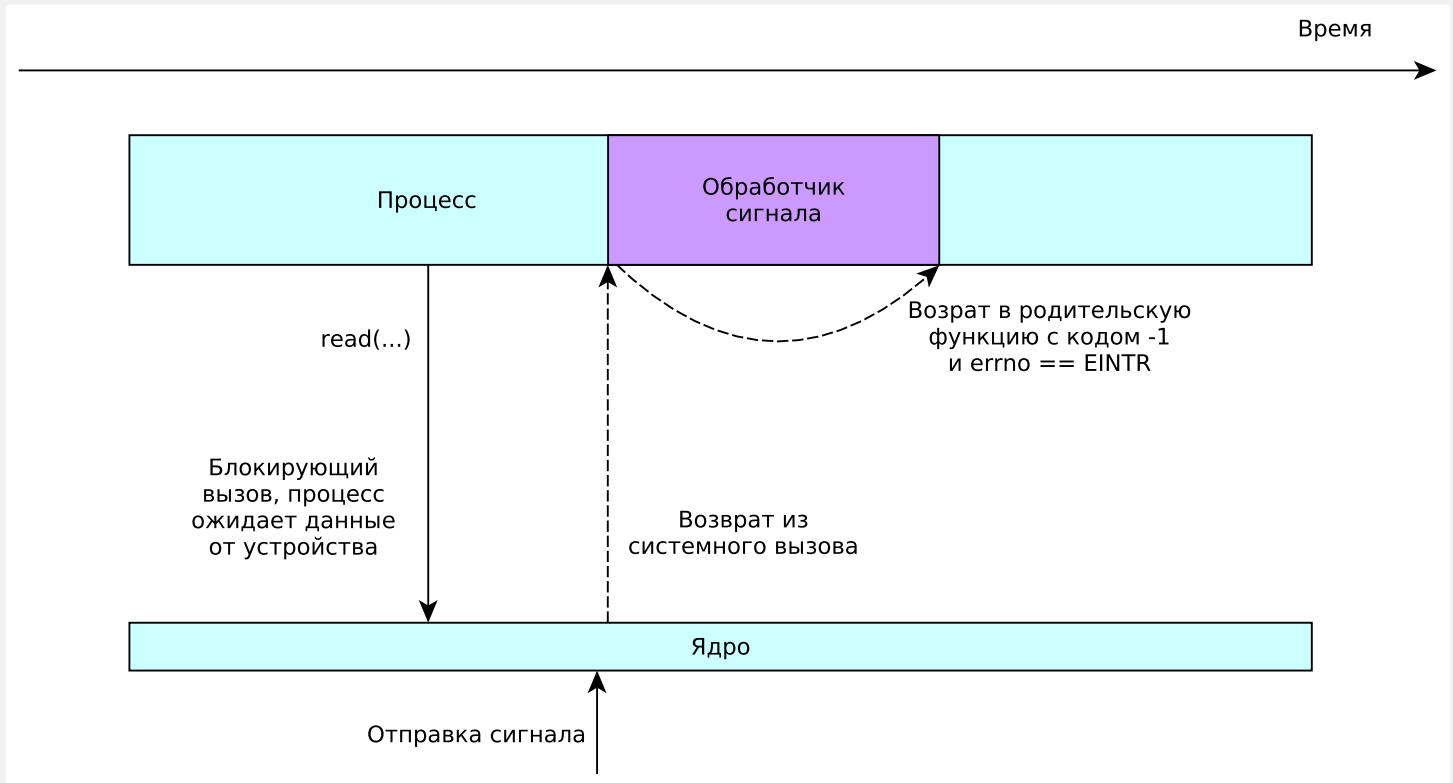
Назначение некоторых важных сигналов:

- SIGTERM - завершение процесса по запросу другого процесса с освобождением всех выделенных ресурсов,
- SIGKILL - немедленное завершение процесса,
- SIGINT - завершение процесса со стороны управляющего терминала,
- SIGSEGV, SIGILL, SIGABRT - завершение процесса в результате некорректных действий самого процесса,
- SIGCHLD - оповещение о событии в дочернем процессе.

Жизненный цикл сигналов



Доставка сигналов



Отправка сигналов через shell

Процесс запускается в фоновом режиме с помощью символа &.

```
$ sleep 100 &  
[1] 7584  
$ ps -A | grep sleep  
7584 pts/42    00:00:00 sleep  
$ kill 7584  
$  
[1]+  Complété          sleep 100
```

Отправка сигналов, API

```
pid_t pid;

pid = fork();

if (pid > 0) {
    int child_status;
    /* Родительский процесс. */

    /* Отправка сигнала SIGTERM дочернему процессу. */
    kill(pid, SIGTERM);

    /* Ожидание завершения дочернего процесса. */
    waitpid(pid, &child_status, 0);
}
```

Обработчики сигналов

У всех сигналов есть обработчик по умолчанию:

- SIGTERM, SIGKILL, SIGINT - процесс немедленно завершается,
- SIGCHLD - сигнал игнорируется.

Многие сигналы могут быть перехвачены процессом за счет регистрации своего специального обработчика. Некоторые сигналы (например, SIGKILL) не перехватываются.

Обработчик сигнала - функция на C. Но не все функции можно использовать в обработчике сигналов.

Обработчики сигналов, API

```
static void sigterm_handler(int sig)
{
    char *text = "Обрабатываю сигнал SIGTERM\n";
    write(1, text, strlen(text));
}

int main(int argc, char *argv[])
{
    struct sigaction sa;

    sa.sa_handler = sigterm_handler;

    /* int sigaction(int sig, const struct sigaction *act,
                     struct sigaction *old_act); */
    sigaction(SIGTERM, &sa, NULL);
}
```


Реентерабельные функции

Не допускает повторное вхождение:

```
int t;  
  
void swap(int *x, int *y)  
{  
    t = *x;  
    *x = *y;  
    *y = t;  
}
```

Допускает повторное вхождение:

```
void swap(int *x, int *y)  
{  
    int t;  
    t = *x;  
    *x = *y;  
    *y = t;  
}
```

Сигналы, повторные вхождения

```
static void sigterm_handler(int sig)
{
    /* Повторное вхождение в функцию printf нарушает внутреннее состояние
    потока вывода stdout. */
    printf("Обрабатываю сигнал SIGTERM\n");
}

int main(int argc, char *argv[])
{
    struct sigaction sa;

    sa.sa_handler = sigterm_handler;

    sigaction(SIGTERM, &sa, NULL);

    /* Функция printf() выполняет вывод данных через системные вызовы
    write(). Один из вызовов прерывается сигналом SIGTERM. */
    printf("Hello world\n");
}
```

Отложенная обработка сигналов, код EINTR

```
bool something_happened;

static void signal_handler(int sig)
{
    something_happened = true;
}

static int deferred_handler(void) {}

int main(int argc, char *argv[])
{
    /* ... Регистрация обработчика signal_handler(). */

    while (1) {
        char buf[100];
        int ret;

        ret = read(STDIN_FILENO, buf, sizeof(buf));
        if (ret < 0 && errno == EINTR) {
            deferred_handler();
        }
    }
}
```

Ожидание завершения дочернего процесса

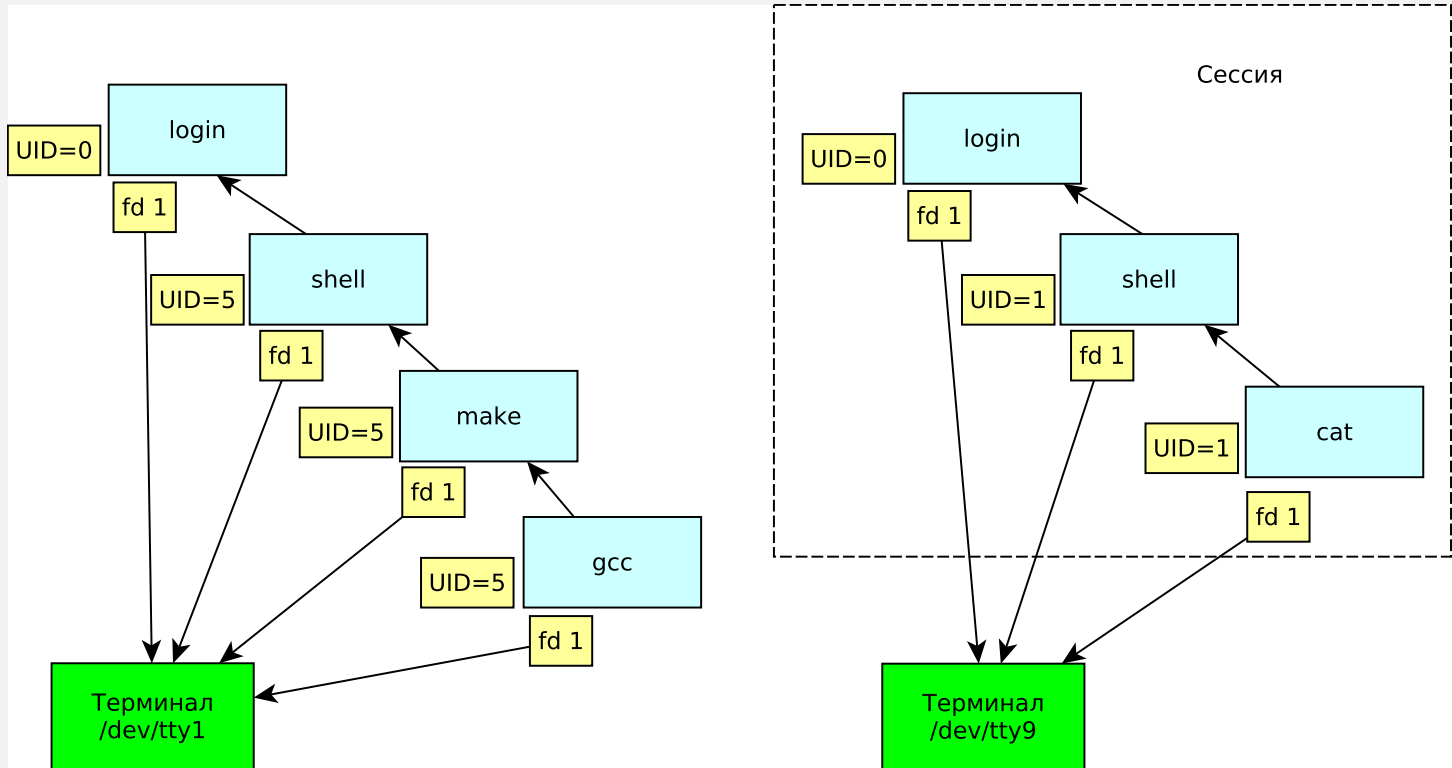
Возможности.

- Вызов `waitpid()` без опции `WNOHANG` - функция выполняется до изменения состояния указанного процесса, блокируя решение других задач.
- Обработка сигнала `SIGCHLD` - сигнал отправляется родительскому процессу в случае завершения дочернего процесса.

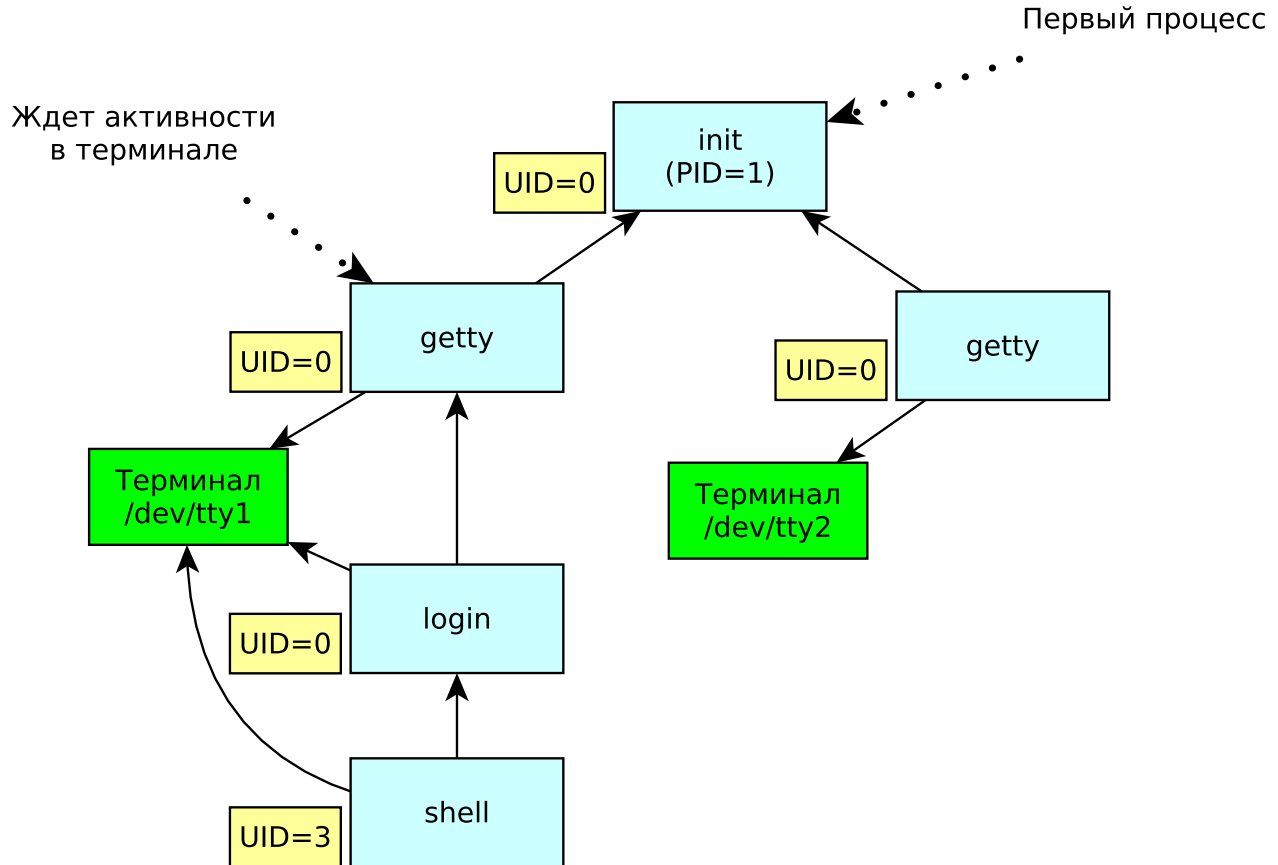
Как обрабатывать `SIGCHLD`.

- Проверить статус всех дочерних процессов с помощью `waitpid()`.
- Опция `WNOHANG` не используется.
- Необходимо иметь в виду, что несколько отправленных сигналов могут сливаться в один полученный сигнал, т.е. по одному сигналу `SIGCHLD` может завершиться несколько дочерних процессов.

Наследование программного окружения



init



Демоны (daemons)

