# Системное программное обеспечение вычислительных машин (СПОВМ)

## Лекция № 02 — Процессы

**Преподаватель: Поденок Леонид Петрович, 505а-5**

**+375 17 293 8039 (505а-5)**
**+375 17 320 7402 (ОИПИ НАНБ)**
**prep@lsi.bas-net.by**

**ftp://student:2ok*uK2@Rwox@lsi.bas-net.by**

**Кафедра ЭВМ, 2021**

2022.02.17

# Оглавление

## Аппаратные возможности компьютеров

Один, отдельно взятый, процессор (процессорное ядро), в один момент времени, может исполнять только одну программу (XX-DOS).

Но к компьютерам предъявляются более широкие требования. В связи с этим разработчики процессоров предусмотрели **мультизадачные возможности** — процессор выполняет какую-то одну программу (процесс или задача).

По истечении некоторого времени (микросекунды), операционная система переключает процессор на другую программу. При этом все регистры текущей программы (состояние, контекст) сохраняются. Через некоторое время вновь передается управление этой программе. Программа при этом не замечает каких либо изменений — для нее процесс переключения незаметен.

Для того чтобы программа не могла каким либо образом нарушить работоспособность системы или других программ, разработчики процессоров предусмотрели механизмы защиты на основе уровней привилегий.

**Уровень привилегий** — это степень использования ресурсов процессора. Всего таких уровней четыре и они имеют номера от 0 до 3.

**Уровень 0** - самый привилегированный. Когда программа работает на этом уровне привилегий, ей «можно всё».

**Уровень 1** - менее привилегированный, и запреты, установленные на уровне 0, действуют для уровня 1.

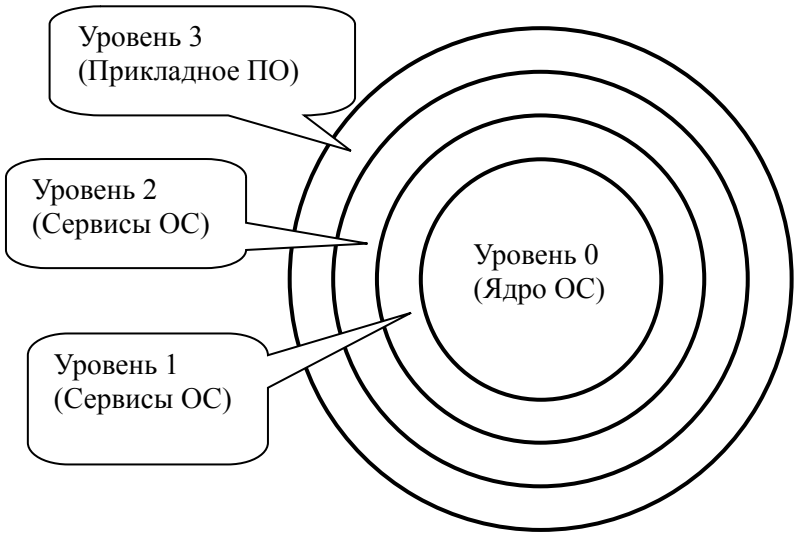
**Уровень 2** - ещё менее привилегированный.

**Уровень 3** - имеет самый низкий приоритет.

## Классическая схема работы программ по уровням привилегий

Классическая схема работы программ по уровням привилегий имеет вид:

уровень 0: ядро операционной системы;
уровень 1: драйверы ОС;
уровень 2: интерфейс ОС;
уровень 3: прикладные программы.




Использование всех четырех уровней привилегий не является необходимым. Существующие системы, спроектированные с меньшим количеством уровней, могут просто игнорировать другие допустимые уровни.

***NIXы** и **Windows**, например, используют только два уровня привилегий:
0 -- ядро системы;
3 -- все остальное.

**IBM OS/2** использует уровни:
0 -- ядро системы;
2 -- процедуры ввода-вывода;
3 -- прикладные программы.

# Процессы

Многозадачность — это один из основных параметров всех современных ВС.

Фундаментальным понятием для изучения работы современных ВС является понятие **процесса** — основного динамического объекта, над которыми ОС выполняет определенные действия.

По ходу работы программы ВС обрабатывает различные команды и преобразует значения переменных. Для этого ОС должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ВВ или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей ВС.

Количество и конфигурация ресурсов могут изменяться с течением времени.

Для описания таких активных объектов внутри вычислительной системы используется термин **«процесс»** — программа, загруженная в память и выполняющаяся.

Стандарт ISO 9000:2000 определяет процесс как совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

Компьютерная программа — только пассивная совокупность инструкций.

Процесс — непосредственное выполнение программных инструкций.

Иногда процессом называют выполняющуюся программу и все её элементы:

- адресное пространство;
- глобальные переменные;
- регистры;
- стек;
- открытые файлы и прочие ресурсы.

Понятие процесса охватывает:

- некоторую совокупность исполняющихся команд;

- совокупность ассоциированных с процессом ресурсов — выделенная для исполнения память или адресное пространство, стеки, используемые файлы, устройства ввода-вывода, и т. д (прикладной контекст, системный контекст).

- текущий момент выполнения — значения регистров и программного счетчика, состояние стека, значения переменных (регистровый контекст).

Взаимно однозначного соответствия между процессами и программами нет. Обычно в рамках программы можно организовывать более одного процесса.

Процесс находится под управлением операционной системы и поэтому в нем может выполняться часть кода ее ядра, которая не находится в исполняемом файле. Это происходит как в случаях, специально запланированных авторами программы, например, при использовании системных вызовов, так и в непредусмотренных ими ситуациях, например, при обработке внешних прерываний).

**Процесс — некоторая совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и текущего момента его выполнения, находящуюся под управлением ОС.**

**Все, что выполняется в вычислительных системах (программы пользователей и определенные части операционных систем), организовано в виде набора процессов.**

## Состояния процесса

На однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс.

Для мультипрограммных вычислительных систем **псевдопараллельная** обработка нескольких процессов достигается с помощью **переключения процессора с одного процесса на другой** — пока один процесс выполняется, остальные ждут своей очереди на доступ к процессору.

Каждый процесс может находиться, как минимум, в двух состояниях:

**- процесс исполняется;**

**- процесс не исполняется.**

Процесс, находящийся в состоянии «**процесс исполняется**», может через некоторое время завершиться или быть приостановлен операционной системой и снова переведен в состояние «**процесс не исполняется**».

Приостановка процесса происходит по одной из двух причин:

1) для его дальнейшей работы потребовалось возникновение какого-либо события (например, завершения операции ввода-вывода);

2) истек временной интервал, отведенный операционной системой для работы этого процесса.

После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии «**процесс не исполняется**», и переводит его в состояние «**процесс исполняется**».

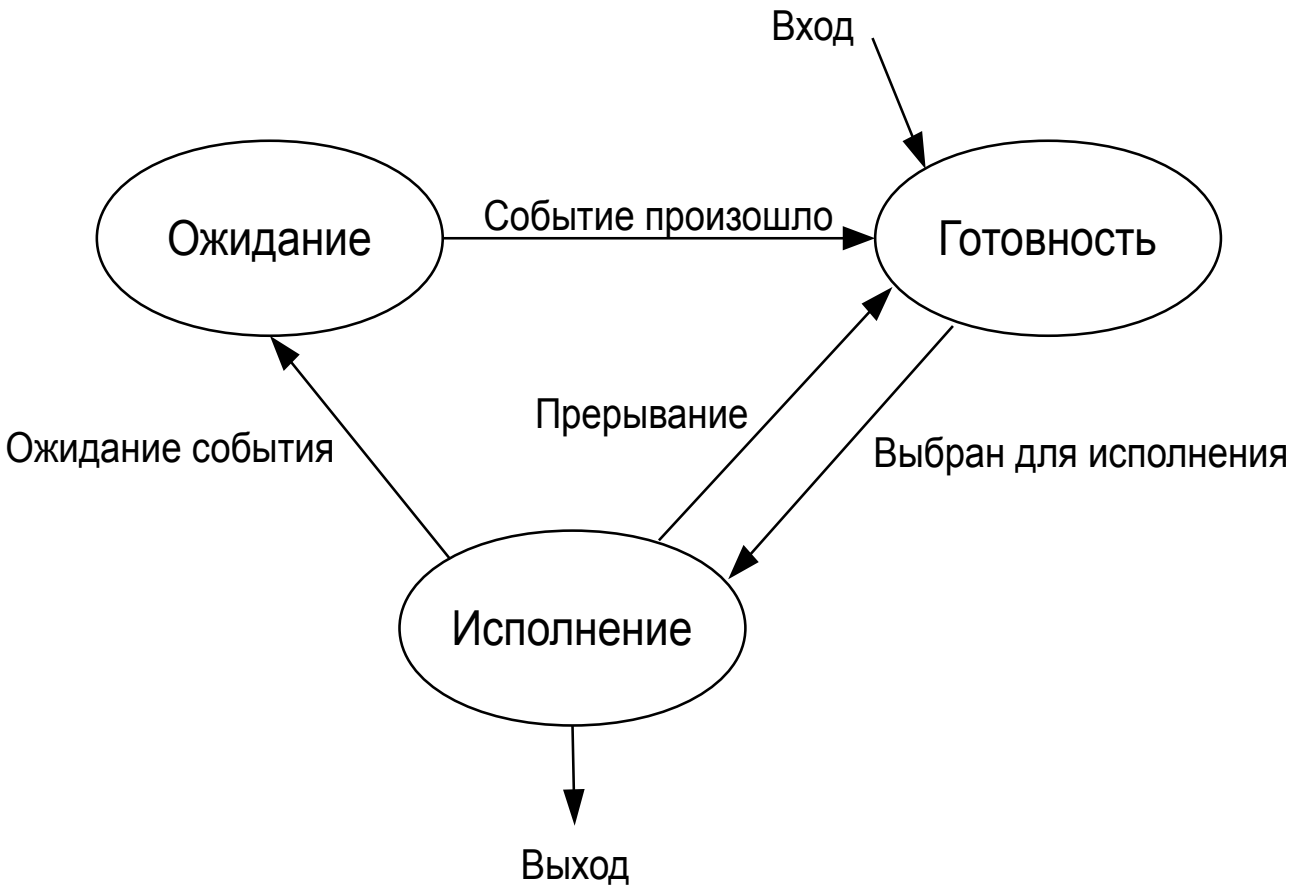
Процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов.

Чтобы это учесть, состояние «процесс не исполняется» разбивается на два новых:

**- готовность;**

**- ожидание**.

Всякий новый процесс, появляющийся в системе, попадает в состояние **«готовность»**. ОС, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние **«исполнение»**.

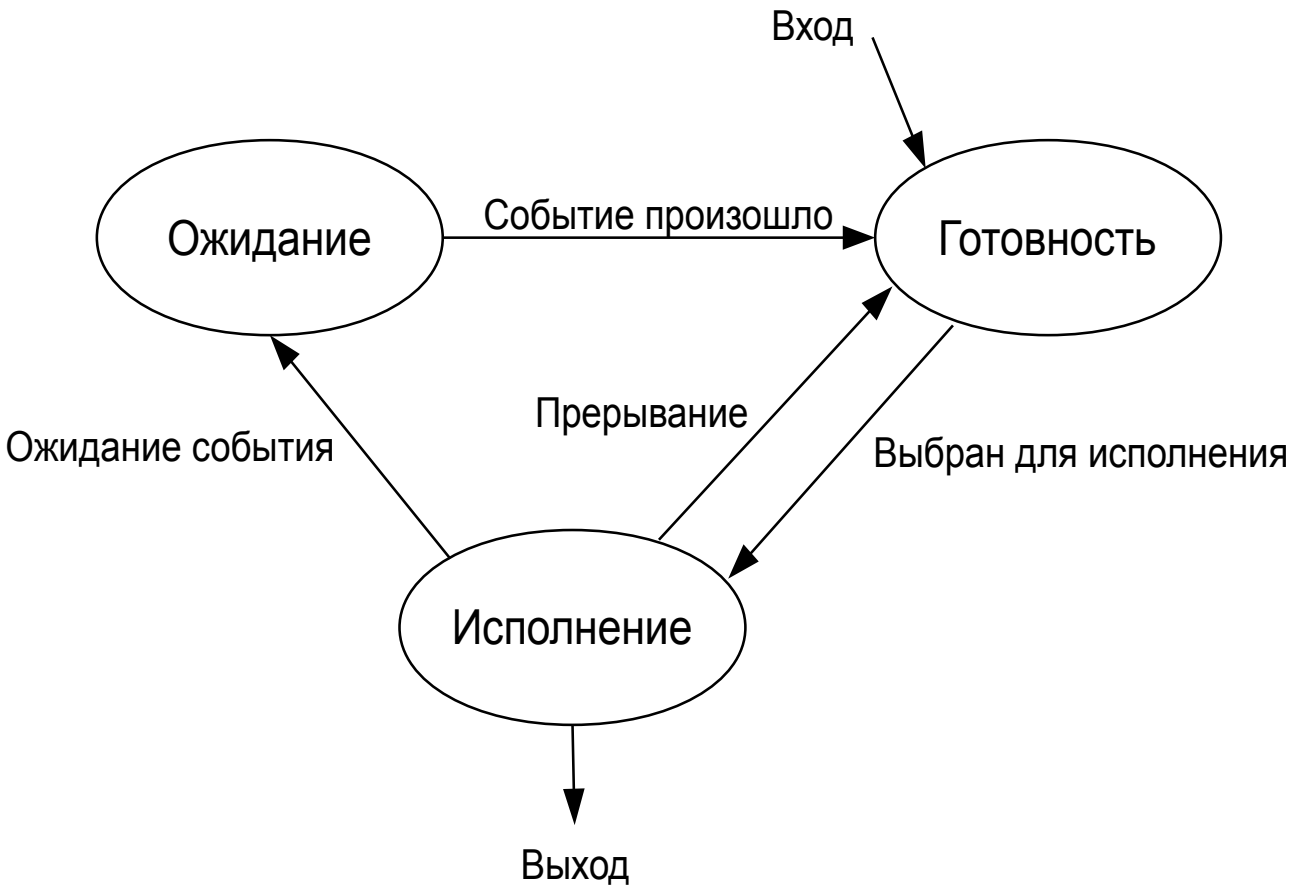


В состоянии **«исполнение»** происходит непосредственное выполнение программного кода процесса. Покинуть это состояние процесс может по трем причинам:

**1)** процесс заканчивает свою деятельность;

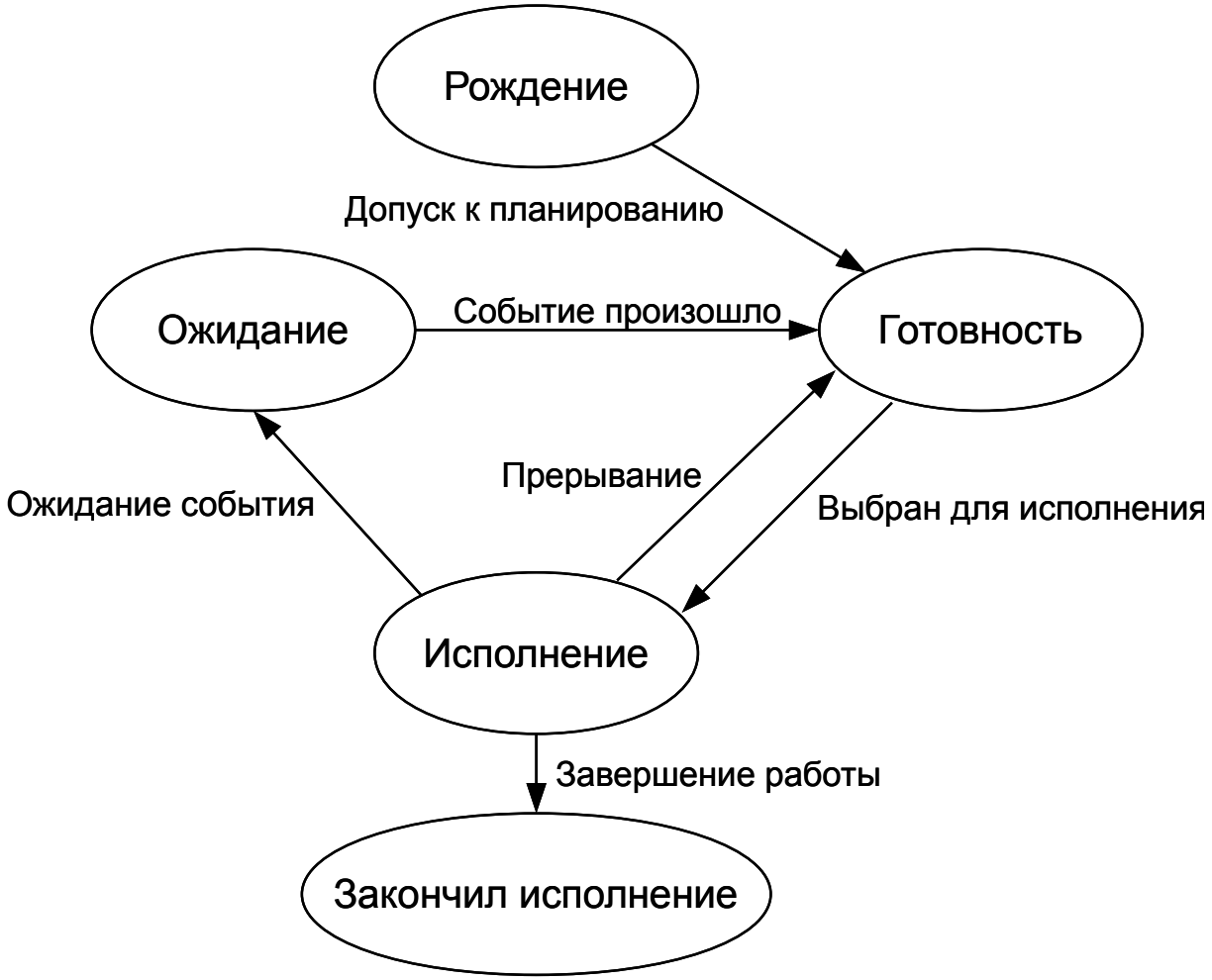
**2)** процесс не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние **«ожидание»**;

**3)** процесс возвращается в состояние **«готовность»** в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении дозволенного времени выполнения).



В реальной ОС вводится еще два состояния:

- **рождение**;
- **закончил исполнение**.




Для появления в вычислительной системе процесс должен пройти через состояние **«рождение»**.

При рождении процессу выделяются адресное пространство, в которое загружается программный код процесса, стек и системные ресурсы, устанавливается начальное значение программного счетчика этого процесса и т. д.

После этого родившийся процесс переводится в состояние «готовность».

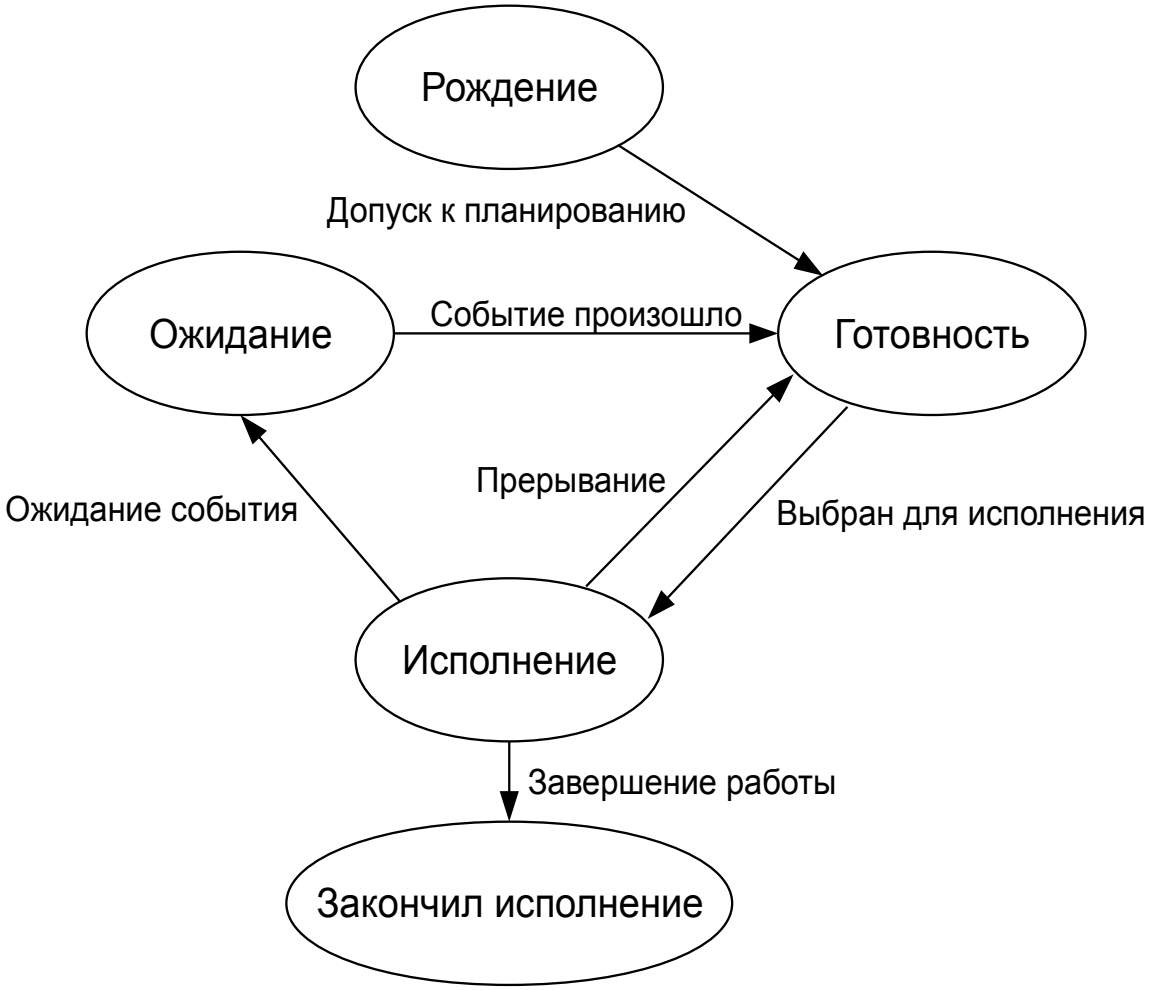
При завершении своей деятельности процесс из состояния исполнение попадает в состояние **«закончил исполнение»**.

## Операции над процессами

Процесс не может сам перейти из одного состояния в другое.

Изменением состояния процессов занимается ОС, совершая над ними операции:

**Создание — завершение**
**Приостановка — запуск**
**Блокирование — разблокирование**
**Изменение приоритета**




Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы никогда не завершаются при работе вычислительной системы).

Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми.

## Process Control Block и контекст процесса

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных.

Эта структура содержит информацию, специфическую для данного процесса:

- состояние, в котором находится процесс;
- программный счетчик (адрес команды, которая будет выполнена следующей);
- содержимое регистров процессора;
- данные, необходимые для планирования использования процессора и управления памятью (приоритет , размер и расположение адресного пространства и т. д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
- информацию об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Во многих ОС информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации (аппаратный вариант — TSS).

Блок управления процессом содержит всю информацию, необходимую для выполнения операций над процессом — **он является моделью процесса для ОС**.

Любая операция, производимая ОС над процессом, вызывает определенные изменения в РСВ.

Информацию, для хранения которой предназначен блок управления процессом, можно разделить на две части:

**- регистровый контекст** — содержимое всех регистров процессора (включая значение программного счетчика);

**- системный контекст** — все остальное.

Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его поведением в операционной системе, совершая над ним операции, однако недостаточно, чтобы полностью характеризовать процесс.

## Пользовательский контекст

Операционную систему не интересует, какими именно вычислениями занимается процесс, т. е. какой код и какие данные находятся в его адресном пространстве.

С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно наряду с регистровым контекстом, определяющее последовательность преобразования данных и полученные результаты.

**Пользовательский контекст** — код и данные, находящиеся в адресном пространстве процесса.

**Контекст процесса** — совокупность регистрового, системного и пользовательского контекстов.

В любой момент времени процесс полностью характеризуется своим контекстом.

В контексте архитектуры x86 (IA32) используется термин задача (**task**), который в некоторой степени, является эквивалентом понятия процесс.

**Задача — это некоторая самостоятельная последовательность команд (процесс), которая выполняется в своём окружении.**

Основные параметры, которыми характеризуется окружение задачи:

- состояние регистров общего назначения (E[ABCD]X, E[SD]I);
- состояние сегментных регистров (DS,CS, SS,ES, FS, GS);
- адресное пространство, которое характеризуется регистром CR3;
- состояние регистров математического сопроцессора и расширений (MMX, SSE, XMM, ...);
- карта портов ввода-вывода.

Поскольку в большинстве случаев количество задач (процессов) намного больше количества процессоров (ядер), многозадачность реализуется путём быстрого **переключения задач**. Именно благодаря этому создаётся ощущение, что все задачи работают одновременно.

# Устройство ядра ОС

1) «Собственно ядро»
2) Драйверы устройств
3) Системные вызовы

**«Собственно ядро»** — функции управления памятью и процессами. Переключение процессов — это важнейший момент нормального функционирования системы.

**Драйвера** — это специальные программы, обеспечивающие работу устройств компьютера. В некоторых OS (FreeBSD) предусматриваются механизмы прерывания работы драйверов по истечении какого-то времени. Тем не менее, можно написать драйвер под FreeBSD или Linux, который полностью заблокирует работу системы.

Избежать этого при двухуровневой защите не представляется возможным, поэтому драйвера надо тщательно программировать.

От драйверов в основном зависит общая производительность системы.

**Системные вызовы** — это интерфейс между процессами и ядром. Никаких других методов взаимодействия процессов с устройствами компьютера быть не должно.

Системных вызовов достаточно много, в Linux 5.10 их более 400, во FreeBSD их порядка 500, причем большей частью они совпадают, соответствуя стандарту POSIX (стандарт, описывающий системные вызовы в UNIX). Разница заключается в передаче параметров.

Прикладным программам абсолютно безразлично, как системные вызовы реализуются в ядре. Это облегчает обеспечение совместимости с существующими системами.

Основные системные вызовы:

- системные вызовы для работы с каталогами;
- системные вызовы для работы с файлами (ввод/вывод);
- системные вызовы для работы с памятью.

В прикладных программах обработки данных наиболее часто используются системные вызовы данных типов. В linux 5.15 системные вызовы объявляются в заголовке:

`/usr/include/asm/unistd.h`

```
$ cat unistd.h                          # с-комментарии удалены
#ifndef _ASM_X86_UNISTD_H
#define _ASM_X86_UNISTD_H
...
#define __X32_SYSCALL_BIT  0x40000000

# ifdef __i386__
#  include <asm/unistd_32.h>
# elif defined(__ILP32__)
#  include <asm/unistd_x32.h>
# else
#  include <asm/unistd_64.h>
# endif

#endif /* _ASM_X86_UNISTD_H */
```

```
$ cat /usr/include/asm/unistd_32.h | grep '__NR_' | wc
    438    1314   11793
$ cat /usr/include/asm/unistd_32.h
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod 15
#define __NR_lchown 16
#define __NR_break 17
#define __NR_oldstat 18
#define __NR_lseek 19
#define __NR_getpid 20
...
#endif /* _ASM_UNISTD_32_H */
```

```
$ cat /usr/include/unistd_64.h | grep '__NR_' | wc
    360    1080    9604
$ cat /usr/include/unistd_64.h
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
#define __NR_rt_sigaction 13
#define __NR_rt_sigprocmask 14
#define __NR_rt_sigreturn 15
#define __NR_ioctl 16
define __NR_pread64 17
#define __NR_pwrite64 18
#define __NR_readv 19
#define __NR_writev 20
...
#endif /* _ASM_UNISTD_64_H */
```

# Системные вызовы управления процессами

**`fork()`** — создает дочерний процесс
**`execve()`** — выполнить программу
**`exit()`** — функция, завершающая работу программы
**`wait()`** — ожидает завершения процесса
**`waitpid()`** — ожидает завершения процесса

## fork — создает дочерний процесс

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

pid_t fork(void);
```

**fork** создает процесс-потомок, который отличается от родительского только значениями **PID** (идентификатор процесса) и **PPID** (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в **0**.

**Блокировки файлов и сигналы, ожидающие обработки, не наследуются.**

Под Linux **fork** реализован с помощью "копирования страниц при записи" (copy-on-write, COW), поэтому расходы на **fork** сводятся к копирования таблицы страниц родителя и созданию уникальной структуры, описывающей задачу.

**Возвращаемое значение**

При успешном завершении родителю возвращается **PID** процесса-потомка, а процессу-потомку возвращается **0**.

При неудаче родительскому процессу возвращается **-1**, процесс-потомок не создается, а значение **errno** устанавливается должным образом.

**Ошибки**

**EAGAIN** — **fork** не может выделить достаточно памяти для копирования таблиц страниц родителя и для выделения структуры описания процесса-потомка.

**ENOMEM** — **fork** не может выделить необходимые ресурсы ядра, потому что памяти слишком мало.

**Соответствие стандартам**

Системный вызов **fork** соответствует SVr4, SVID, POSIX, X/OPEN, BSD 4.3.

**Пример**

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

int main( ) {

    pid_t pid, ppid, chpid;
    int a = 0;
    chpid = fork( );
// При успешном создании нового процесса с этого места псевдопараллельно
// начинают работать 2 процесса: старый и новый
// Перед выполнением следующего выражения a в обоих процессах равно 0
    a = a + 1;

// Узнаем идентификаторы текущего и родительского процесса в каждом из них
    pid  = getpid( );
    ppid = getppid( );

// Печатаем значения PID, PPID и вычисленное значение a в каждом из процессов
    printf("My pid = %d, my ppid = %d, result = %d\n", (int)pid, (int)ppid, a);
    return 0;
}
```

**Вывод**

My pid = **128969**, my ppid = 116473, result = 1

My pid = 128970, my ppid = **128969**, result = 1

**Пример**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

_Noreturn int child_foo(void);
int child_status;

int main() {

    fprintf(stdout, "Родительский процесс стартовал ...\n");
    pid_t pid = fork();
    if (pid == -1) {
        fprintf(stdout, "Ошибка, код ошибки - %d\n", errno);
    }
    if (pid == 0) { // дочерний процесс
        fprintf(stdout, "Это дочерний процесс. Вызываем child_foof()...\n");
        child_foo();
    }
    fprintf(stdout, "Родительский процесс продолжает выполнение\n");
    wait(&child_status);
    fprintf(stdout, "Дочерний процесс завершился с кодом завершения %d\n",
            WEXITSTATUS(child_status));
    exit(0);
}
```

```
_Noreturn int child_foo(void) {

    fprintf(stdout, "%s()\n", __func__);
    exit(123);
}
```

**Выход**

```
Родительский процесс стартовал ...
Родительский процесс продолжает выполнение
Это дочерний процесс. Вызываем child_foof()...
child_foo()
Дочерний процесс завершился с кодом завершения 123
```

## execve — выполнить программу

```
#include <unistd.h>

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

**execve()** выполняет программу, заданную параметром **filename**. Программа должна быть или двоичным исполняемым файлом, или скриптом, начинающимся со строки вида

"#! интерпретатор [аргументы]"

В последнем случае интерпретатор — это правильный путь к исполняемому файлу, который не является скриптом; этот файл будет выполнен как **интерпретатор [arg] filename**.

**argv** — это массив строк, аргументов новой программы.

**envp** — это массив строк в формате **key=value**, которые передаются новой программе в качестве окружения (*environment*).

Как **argv**, так и **envp** завершаются нулевым указателем.

К массиву аргументов и к окружению можно обратиться из функции **main()**, которая объявлена как

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

**execve( )** не возвращает управление при успешном выполнении, а код, данные, bss и стек вызвавшего процесса перезаписываются кодом, данными и стеком загруженной программы.

Новая программа также наследует от вызвавшего процесса его идентификатор и открытые файловые дескрипторы, на которых не было флага '*закрыть-при-завершении*' (close-on-exec, COE).

Сигналы, ожидающие обработки, удаляются.

Переопределённые обработчики сигналов возвращаются в значение по умолчанию.

Обработчик сигнала **SIGCHLD** (когда установлен в **SIG_IGN**) может быть сброшен или не сброшен в **SIG_DFL**.

Если текущая программа выполнялась под управлением **ptrace**, то после успешного **execve( )** ей посылается сигнал **SIGTRAP**.

Если на файле программы **filename** установлен `setuid`-бит, то фактический идентификатор пользователя вызывавшего процесса меняется на идентификатор владельца файла программы.

Точно так же, если на файле программы установлен `setgid`-бит, то фактический идентификатор группы устанавливается в группу файла программы.

Если исполняемый файл является динамически-скомпонованным файлом в формате **a.out**, содержащим заглушки для вызова совместно используемых библиотек, то в начале выполнения этого файла вызывается динамический компоновщик **ld.so**(8), который загружает библиотеки и компонует их с исполняемым файлом.

Если исполняемый файл является динамически-скомпонованным файлом в формате **ELF**, то для загрузки разделяемых библиотек используется интерпретатор, указанный в сегменте **PT_INTERP**. Обычно это **/lib/ld-linux.so.1** для программ, скомпилированных под Linux **libc** версии 5, или же **/lib/ld-linux.so.2** для программ, скомпилированных под GNU **libc** версии 2.

**Возвращаемое значение**

При успешном завершении **execve( )** не возвращает управление, при ошибке возвращается **-1**, а значение **errno** устанавливается должным образом.

**Коды ошибок**

**EACCES** — интерпретатор файла или скрипта не является обычным файлом.

**EACCES** — нет прав на выполнение файла, скрипта или ELF-интерпретатора.

**EACCES** — файловая система смонтирована с флагом **noexec**.

**EPERM** — файловая система смонтирована с флагом **nosuid**, пользователь не является супер-пользователем, а на файле установлен бит **setuid** или **setgid**.

**EPERM** — процесс работает под отладчиком, пользователь не является суперпользователем, а на файле установлен бит **setuid** или **setgid**.

**E2BIG** — список аргументов слишком велик.

**ENOEXEC** — исполняемый файл в неизвестном формате, для другой архитектуры, или же встре-чены какие-то ошибки, препятствующие его выполнению.

**EFAULT** — **filename** указывает за пределы доступного адресного пространства.

**ENAMETOOLONG** — **filename** слишком длинное.

**ENOENT** — файл **filename**, или интерпретатор скрипта или ELF-файла не существует, или же не найдена разделяемая библиотека, требуемая файлу или интерпретатору.

**ENOMEM** — недостаточно памяти в ядре.

**ENOTDIR** — компонент пути **filename**, или интерпретатору скрипта или ELF-интерпретатору не является каталогом.

**EACCES** — нет прав на поиск в одном из каталогов по пути к **filename**, или имени интерпретатора скрипта или ELF-интерпретатора.

**ELOOP** — слишком много символьных ссылок встречено при поиске **filename**, или интерпретатора скрипта или ELF-интерпретатора.

**ETXTBSY** — исполняемый файл открыт для записи одним или более процессами.

**EIO** — произошла ошибка ввода-вывода.

**ENFILE** — достигнут системный лимит на общее количество открытых файлов.

**EMFILE** — процесс уже открыл максимально доступное количество открытых файлов.

**EINVAL** — исполняемый файл в формате ELF содержит более одного сегмента **PT_INTERP** (то есть, в нем указано более одного интерпретатора).

**EISDIR** — ELF-интерпретатор является каталогом.

**ELIBBAD** — ELF-интерпретатор имеет неизвестный формат.

**Соответствие стандартам**

SVr4, SVID, X/OPEN, BSD 4.3. POSIX не документирует поведение, связанное с **#!**, но в остальном совершенно совместимо. SVr4 документирует дополнительные коды ошибок **EAGAIN**, **EINTR**, **ELIBACC**, **ENOLINK**, **EMULTIHOP**; POSIX не документирует коды ошибок **ETXTBSY**, **EPERM**, **EFAULT**, **ELOOP**, **EIO**, **ENFILE**, **EMFILE**, **EINVAL**, **EISDIR** и **ELIBBAD**.

**Замечания**

**SUID** и **SGID** процессы не могут быть оттрассированы **ptrace( )**.

Linux игнорирует **SUID** и **SGID** биты на скриптах.

Результат монтирования файловой системы с опцией **nosuid** различается в зависимости от версий ядра Linux: некоторые ядра будут отвергать выполнение **SUID/SGID** программ, когда это должно дать пользователю те возможности, которыми он уже не обладает (и возвращать **EPERM**), некоторые ядра будут просто игнорировать **SUID/SGID** биты, но успешно производить запуск программы.

Первая строка (строка с **#!**) исполняемого скрипта не может быть длиннее 127 символов.

**Пример**

**parent.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

int main() {

    int child_status;
    char *args[] = {"pr0gramm", "parm_1", (char*)0};
    pid_t pid;
    pid = fork();

    if (pid == -1) {
        fprintf(stdout, "Error occured, error code - %d\n", errno); exit(errno);
    }
    if (pid == 0) {
        fprintf(stdout, "Child process created. Please, wait...\n");
        execve("./child", args, NULL);
    }
    wait(&child_status);
    fprintf(stdout, "Child process have ended with %d exit status\n",
child_status);
    exit(0);
```

```
}
```

**child.c**

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    fprintf(stdout, "Child process begins...\n");
    for (int i = 0; i < argc; i++) {
        fprintf(stdout, "%s\n", argv[i]);
    }
    exit(0);
}
```

**makefile**

```
CC = gcc
CFLAGS = -W -Wall -Wextra -std=c11
.PHONY: clean

all: parent child
parent: parent.c makefile
        $(CC) $(CFLAGS) parent.c -o parent
child: child.c makefile
        $(CC) $(CFLAGS) child.c -o child
clean:
        rm parent child
```

**Компиляция и сборка**

```
$ make
gcc -W -Wall -Wextra -std=c11 parent.c -o parent
gcc -W -Wall -Wextra -std=c11 child.c -o child
$ls -1
child
child.c
makefile
parent
parent.c
```

**Запуск**

```
$ ./parent
Parent process begins...
Child process created. Please, wait...
Child process begins...
pr0gramm
parm_1
Child process have ended with 0 exit status
```

## wait, waitpid — ожидает завершения процесса

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Функция **wait** приостанавливает выполнение текущего процесса до тех пор, пока какой нибудь из дочерних процессов не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция **waitpid** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре **pid**, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый «*зомби*» — "zombie"), то функция немедленно возвращает управление.

Системные ресурсы, связанные с дочерним процессом, освобождаются.

Параметр **pid** может принимать несколько значений:

**< -1** — означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению **pid**.

**-1** — означает ожидание любого дочернего процесса; функция **wait** ведет себя точно так же.

**0** — означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса.

**> 0** — означает ожидание дочернего процесса, чей идентификатор равен **pid**.

**options**

Значение **options** создается путем логического сложения нескольких следующих констант:

**WNOHANG** — означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.

**WUNTRACED** — означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено. Статус для отслеживаемых остановленных подпроцессов также обеспечивается без этой опции.

**status**

Если **status** не равен **NULL**, то функции **wait** и **waitpid** сохраняют информацию о статусе в переменной, на которую указывает **status**.

Состояние **status** можно проверить с помощью макросов, которые принимают в качестве аргумента буфер (типа **int**), а не указатель на буфер!:

**WIFEXITED(status)** — не равно нулю, если дочерний процесс успешно завершился.

**WEXITSTATUS(status)** — возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс.

Эти биты могли быть установлены в аргументе функции **exit()** или в аргументе оператора **return** функции **main()**. Этот макрос можно использовать, только если **WIFEXITED** вернул ненулевое значение.

**WIFSIGNALED(status)** — возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.

**WTERMSIG(status)** — возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если **WIFSIGNALED** вернул ненулевое значение.

**WIFSTOPPED(status)** — возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг **WUNTRACED** или когда подпроцесс отслеживается (см. **ptrace(2)**).

**WSTOPSIG(status)** — возвращает номер сигнала, из-за которого дочерний процесс был остановлен. Этот макрос можно использовать, только если **WIFSTOPPED** вернул ненулевое значение.

Некоторые версии Unix (например Linux, Solaris, но не AIX, SunOS) также определяют макрос **WCOREDUMP(status)** для проверки того, не вызвал ли дочерний процесс ошибку ядра.

Использовать его следует только в структуре

```
#ifdef WCOREDUMP
...
#endif
```

**Возвращаемые значения**

Возвращает идентификатор дочернего процесса, который завершил выполнение, или ноль, если использовался **WNOHANG** и ни один дочерний процесс пока еще недоступен, или **-1** в случае ошибки (в этом случае переменной **errno** присваивается соответствующее значение).

**Ошибки**

**ECHILD** — процесс, указанный в **pid**, не существует или не является дочерним процессом текущего процесса. (Это может случиться и с собственным дочерним процессом, если обработчик сигнала **SIGCHLD** установлен **в SIG_IGN**. См. главу ЗАМЕЧАНИЯ по поводу многозадачности процессов.)

**EINVAL** — аргумент **options** неверен.

**EINTR** — использовался флаг **WNOHANG**, и был получен необработанный сигнал или **SIGCHLD**. Стандарт Single Unix Specification описывает флаг **SA_NOCLDWAIT** (не поддерживается в Linux), если он установлен или обработчик сигнала **SIGCHLD** устанавливается в **SIG_IGN**, то завершившиеся дочерние процессы не становятся зомби, а вызов **wait()** или **waitpid()** блокируется, пока все дочерние процессы не завершатся, а затем устанавливает переменную errno равной **ECHILD**.

Стандарт POSIX оставляет неопределенным поведение при установке **SIGCHLD** в **SIG_IGN**. поздние стандарты, включая SUSv2 и POSIX 1003.1-2001, определяют поведение, только что описанное как опция совместимости с XSI.

Linux не следует второму варианту — если вызов **wait()** или **waitpid()** сделан в то время, когда **SIGCHLD** игнорируется, то вызов ведет себя, как если бы **SIGCHLD** не игнорировался, то есть вызов блокирует до завершения работы следующего подпроцесса и возврата идентификатора процесса PID и статуса этого подпроцесса.

**Замечания по linux**

В ядре Linux задачи, управляемые ядром, по внутреннему устройству не отличаются от процесса. Задача (*thread*) — это простой процесс, который создан специфичным для Linux системным вызовом **clone(2)**; другие процедуры, типа портируемой **pthread_create(3)**, также реализованы с помощью **clone(2)**.

До Linux 2.4, задачи были частным случаем процесса, и последовательность одной группы задач не могла ожидать дочерний процесс или другую группу задач, даже если воследняя принадлежит к этой-же группе задач. Однако, POSIX предопределяет такие особенности, и с Linux 2.4 группа задач может, и по умолчанию будет ожидать дочерний процесс другой группы задач в этой же группе задач.

Следующие специфичные для Linux параметры существуют для использования с дочерними процессами и **clone(2)**.

**__WCLONE** — только ожидает дочерние процессы. Если отменяется, то ожидает только дочерних неклонированных процессов. («клонированным» дочерним процессом является подпроцесс, не отправляющий сигналов, или выдающий сигналы, отличающиеся от **SIGCHLD** своему родителю до завершения.) Эта опция игнорируется, если определено **__WALL**.

**__WALL** — (с Linux 2.4) Ожидает все дочерние процессы, независимо от их типа («клон» или «не-клон»).

**__WNOTHREAD** — (С Linux 2.4) Не ожидать дочерние процессы или остальные задачи в идентичной группе задач. Было параметром по умолчанию до Linux 2.4.

**Соответствие стандартам**

SVr4, POSIX.1

## exit — функция, завершающая работу программы

```
#include <unistd.h>
#include <stdlib.h>

void _Exit(int status);
```

**_exit** "немедленно" завершает работу программы. Все дескрипторы файлов, принадлежащие процессу, закрываются; все его дочерние процессы начинают управляться процессом 1 (init), а родительскому процессу посылается сигнал **SIGCHLD**.

Значение **status** возвращается родительскому процессу как статус завершаемого процесса; он может быть получен с помощью одной из функций семейства **wait**.

Функция **_Exit** эквивалентна функции **_exit**.

**Возвращаемые значения**

Эти функции никогда не возвращают управление вызвавшей их программе.

**Соответствие стандартам**

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. Функция **_Exit()** была представлена C99.

**Замечания**

Для рассмотрения эффектов завершения работы, передачу статуса выхода, зомби-процессов, сигналов и т.п., следует смотреть документацию по **exit(3)**.

Функция **_exit** аналогична **exit()**, но не вызывает никаких функций, зарегистрированных с функцией C **atexit**, а также не вызывает никаких зарегистрированных обработчиков сигналов. Будет ли выполняться сброс стандартных буферов ввода-вывода и удаление временных файлов, созданных **tmpfile(3)**, зависит от реализации. С другой стороны, **_exit** закрывает открытые дескрипторы файлов, а это может привести к неопределенной задержке для завершения вывода данных. Если задержка нежелательна, то может быть полезным перед вызовом **_exit()** вызывать функции типа **tcflush()**. Будет ли завершен ввод-вывод, а также какие именно операции ввода-вывода будут завершены при вызове **_exit()**, зависит от реализации.