

# MINIX 3 API вызовов ядра

Jorrit N. Herder  
<jnherder@cs.vu.nl>  
20.10.2005

## Аннотация

Главным образом, вызовы ядра позволяют системным<sup>1</sup> процессам запрашивать сервисы ядра, например, для осуществления привилегированных операций. Этот документ сжатой обобщает организацию вызовов ядра в MINIX3 и предоставляет обзор всех вызовов ядра.

## Организация вызовов ядра

Вызов ядра означает, что запрос посылается ядру, где он обслуживается одной из задач (tasks) ядра. Детали асемблерного кода сообщения запроса, посылки его ядру, и ожидание ответа удобно сокрыты в системной библиотеке. Заголовочный файлы этой библиотеки - `src/include/minix/syslib.h`, а файлы реализации находятся в каталоге `src/lib/syslib`.

Действительные реализации вызовов ядра определены в одной из задач ядра. В противоположность MINIX2, задача CLOCK более не принимает системных вызовов. Вместо этого, все вызовы теперь направляются к задаче SYSTEM. Предполагается, что программа делает `sys_call()` системный вызов<sup>2</sup>. Согласно соглашениям, этот вызов трансформируется в сообщение микроядра с типом `SYS_CALL`, которое посылается задаче ядра SYSTEM. Задача SYSTEM обслуживает запрос в функции с именем `do_call()` и возвращает результат.

Карта отображения номеров вызовов ядра на функции обработчики делается во время инициализации задачи SYSTEM в `src/kernel/system.c`. Прототипы функций обработчиков декларированы в `src/kernel/system.h`. Их реализации содержатся в отдельных файлах в каталоге `src/kernel/system/`. Эти файлы компилируются в библиотеку `src/kernel/system/system.a`, которая компонуется с ядром.

Номера вызовов ядра, их параметры запроса и параметры ответа, определены в `src/include/minix/com.h`. К сожалению, MINIX2 не следует строгой схеме именования. Поэтому, нумерация типов сообщений и параметров были переименованы в MINIX3. Вызовы ядра сейчас все начинаются с `SYS_` и все параметры, которые принадлежат к одному и тому же вызову ядра, теперь разделяют общий префикс.

## Обзор вызовов ядра MINIX3

Сжатый обзор вызовов ядра в MINIX3 приведен ниже, на рисунке 1. Статус каждого вызова

---

1 Точнее: привилегированным процессам, обычно запускаемым посредством утилиты `service`; к таким процессам относятся системные задачи, драйверы, сервера (прим. перев.).

2 Здесь оригинал не точен. Каждая функция вызова ядра вида `sys_*`, на самом деле ретранслируется в вызов функции отправки сообщения запроса `_taskcall()` с соответствующими параметрами (а не `syscall()`, как указано в тексте). Тип запроса `_taskcall()` при этом (2-й параметр) будет соответствовать запрошенной операции (например `SYS_FORK`), а не `SYS_CALL`. Код функции `_taskcall()` вы можете найти: `/usr/src/lib/syslib/taskcall.c` (прим. перев.)

относительно MINIX2<sup>3</sup> даётся в последней колонке.

Вызов ядра	Цель	Статус
управление процессами		
SYS_FORK	Fork a process; copy parent process	
SYS_EXEC	Execute a process; initialize registers	
SYS_EXIT	Exit a user process; clear process slot	U
SYS_NICE	Change priority of a user process	N
SYS_PRIVCTL	Change system process privileges	N
SYS_TRACE	Trace or control process execution	
обслуживание сигналов		
SYS_KILL	Send a signal to a process	U
SYS_GETKSIG	Check for pending kernel signals	N
SYS_ENDKSIG	Tell kernel signal has been processed	
SYS_SIGSEND	Start POSIX-style signal handler	
SYS_SIGRETURN	Return from POSIX-style signal	
управление памятью		
SYS_NEWMAP	Install new or updated memory map	
SYS_SEGCTL	Add extra, remote memory segment	N
SYS_MEMSET	Write a pattern into physical memory area	N
копирование данных		
SYS_UMAP	Map virtual to physical address	U
SYS_VIRCOPY	Copy data using virtual addressing	U
SYS_PHYSCOPY	Copy data using physical addressing	U
SYS_VIRVCOPY	Handle vector with virtual copy requests	U
SYS_PHYSVCOPY	Handle vector with physical copy requests	N
ввод/вывод на устройствах		
SYS_DEVIO	Read or write a single device register	N
SYS_SDEVIO	Input or output an entire data buffer	N
SYS_VDEVIO	Process vector with multiple requests	N
SYS_IRQCTL	Set or reset an interrupt policy	N
SYS_INT86	Make a real-mode BIOS call	N
SYS_IOPENABLE	Give process I/O privilege	N
управление системой		
SYS_ABORT	Abort MINIX: shutdown the system	U
SYS_GETINFO	Get a copy system info or kernel data	N
функции часов		
SYS_SETALARM	Set or reset a synchronous alarm timer	U
SYS_TIMES	Get process times and uptime since boot	U

**Рисунок 1.** Этот рисунок производит обзор вызовов ядра MINIX3. Обозначения для статуса:

<sup>3</sup> Здесь автор сравнивает с предыдущей линией версий MINIX (прим. перев.).

N — новый, U — обновлённый (т. е. полностью обновлённый — так как все вызовы получили минимальные обновления) со времени MINIX2.

## Интерфейс вызовов ядра

Интерфейс вызовов ядра детализируется ниже. Для каждого вызова ядра специфицируется тип сообщения, назначение, параметры запроса и/или ответа, и возвращаемое значение. Стенографические дополнительные ремарки о будущем предназначении вызова также могут быть приведены.

### Обозначения

**CONSTANT**: так обозначается определение константы; числовой индикатор типа запроса, или статуса;

**PARAMETER**: параметры сообщения; поле в сообщениях запроса или ответа.

**void sys call(arguments)**: функция из системной библиотеки; заметка о выполнении вызова ядра;

### Алфавитный обзор<sup>4</sup>

**SYS\_ABORT**: Завершить систему MINIX и возвратиться в монитор загрузки — если возможно. Эта возможность используется PM, FS и TTY. Нормально завершение инициируется обычно пользователем, например, обозначая команду shutdown, или <Ctrl><Alt><Del>. MINIX также будет осуществлять это действие, если обнаружится фатальная ошибка в PM или FS<sup>5</sup>.

- параметры запроса:

**ABRT\_NOW**: как остановить, одно из константных значений, определённых в файле  
src/include/unistd.h:

**RBT\_HALT** — остановить MINIX и вернуться в монитор загрузки.

**RBT\_REBOOT** — перезагрузить MINIX.

**RBT\_PANIC** — наблюдается 'kernel panic'.

**RBT\_MONITOR** — выполнить специфицированный код в мониторе загрузки.

**RBT\_RESET** — аппаратный сброс системы.

**ABRT\_MON\_PROC**: процесс, из которого получить параметры для монитора загрузки.

**ABRT\_MON\_LEN**: длина списка параметров монитора загрузки.

**ABRT\_MON\_ADDR**: виртуальный адрес параметров.

- возвращаемый результат:

**OK**: последовательность завершения стартовала.

**ENIVAL**: ошибочный номер процесса.

**EFAULT**: недопустимый адрес параметра монитора.

**E2BIG**: параметры монитора превышают максимальную длину.

- библиотечные функции:

```
int sys_abort(int shutdown status, ...);
```

---

<sup>4</sup> Вначале, в алфавитном порядке, указывается параметр сообщения системного вызова, функции API, его порождающие, приведены после как «библиотечные вызовы»: одному системному вызову может соответствовать несколько вызовов API (прим. перев.).

<sup>5</sup> И некоторое, достаточно большое, число сложно диагностируемых ошибочных ситуаций в коде ядра (прим. перев.).

**SYS\_DEVIO:** Осуществить ввод/вывод на устройстве от имени драйвера устройства из пользовательского пространства. Драйвер может запросить одиночный порт, который будет прочитан или записан этим вызовом. Смотри также вызовы **SYS\_SDEVIO** и **SYS\_VDEVIO**.

- параметры запроса:

*DIO\_REQUEST* : ввод или вывод.

*DIO\_INPUT* прочитать значение из *DIO\_PORT*.

*DIO\_OUTPUT* записать *DIO\_VALUE* в *DIO\_PORT*.

*DIO\_TYPE* : флаг, индицирующий тип значения.

*DIO\_BYTE* байт.

*DIO\_WORD* слово.

*DIO\_LONG* long тип.

*DIO\_PORT* : порт для чтения или записи.

*DIO\_VALUE* : записываемое значение, только для *DIO\_OUTPUT*.

- параметры ответа:

*DIO\_VALUE* : значение, сосчитанное из указанного порта, только для *DIO\_INPUT*.

- возвращаемый результат:

OK: порт ввода/вывода успешно обслужен.

EINVAL: недопустимые *DIO\_REQUEST* или *DIO\_TYPE* было установлено.

- библиотечные функции:

```
int sys_in(port_t port, unsigned long value, int io_type);  
int sys_inb(port_t port, u8_t *byte);  
int sys_inw(port_t port, u16_t *word);  
int sys_inl(port_t port, u32_t *long);  
int sys_out(port_t port, unsigned long *value, int io_type);  
int sys_outb(port_t port, u8_t byte);  
int sys_outw(port_t port, u16_t word);  
int sys_outl(port_t port, u32_t long);
```

**SYS\_ENDKSIG:** Завершить обработку сигнала ядром. РМ использует этот вызов, чтобы обозначить этим, что он обработал то, что ядро сигнализировало в отображении через **SYS\_GETKSIG** вызов.

- параметры запроса:

*SIG PROC* : процесс, которого это касается.

- возвращаемый результат:

EINVAL: процесс не имеет не обработанных сигналов, или завершается.

OK: ядро очистило все ожидающие сигналы.

- библиотечные функции:

```
int sys_endksig(int proc_nr);
```

**SYS\_EXEC:** Обновить регистры процесса после успешного `exec()` POSIX-вызова. После того, как FS копировал бинарный образ в память, РМ информирует ядро о новой детализации регистров.

- параметры запроса:

*PR\_PROC\_NR* : процесс который выполняет программу.

*PR\_STACK\_PTR* : новый указатель стека.

*PR\_IP\_PTR* : новый счётчик команд.

*PR\_NAME\_PTR* : указатель на имя программы.

- возвращаемый результат:

OK: этот вызов всегда успешен.

- библиотечные функции:

```
int sys_exec(int proc, char *stack ptr,  
             char *prog name, vir bytes pc);
```

**SYS\_EXIT:** Очистить слот в таблице процессов. Это обычно исходит от PM после завершения пользовательского процесса. Системные процессы, включая PM, могут также непосредственно вызывать эту функцию по их завершению.

- параметры запроса:

*PR\_PROC\_NR* : номер слота завершаемого процесса, если вызывающей стороной  
есть PM. Используйте **SELF**, если завершается сам PM.

- возвращаемый результат:

OK : очистка успешна.

EINVAL : некорректный номер процесса.

EDONTREPLY : этот вызов не возвращается, если системный процесс завершается.

- библиотечные функции:

```
int sys_exit(int proc_nr);
```

**SYS\_FORK:** Инициировать новый (дочерний) процесс в таблице процессов ядра, и инициализировать его, базируясь на прототипе (родителе) процесса. PM находит свободный процессорный слот для дочернего процесса в своей таблице процессов, и тут же запрашивает ядро обновить таблицу процессов ядра.

- параметры запроса:

*PR\_PROC\_NR* : дочернего процесса слот в таблице.

*PR\_PPROC\_NR* : родительский процесс, выполняющий `fork()`.

- возвращаемый результат:

OK : новый процессорный слот успешно присвоен.

EINVAL : неверный номер родительского процесса, или дочерний слот занят.

- библиотечные функции:

```
int sys_fork(int parent_proc_nr, int child_proc_nr);
```

**SYS\_GETINFO:** Получить копию структуры данных ядра. Этот вызов поддерживает драйверы и сервера пользовательского пространства, нуждающиеся в достоверной системной информации.

- параметры запроса:

*I\_REQUEST* : тип системной информации, которая запрашивается:

GET\_IMAGE - копировать таблицу загрузочного образа.

GET\_IRQHOOKS - копировать таблицу с перехватчиками прерываний.

GET\_KINFO - копировать информационную структуру ядра.

GET\_KMESSAGES - копировать буфер с диагностическими сообщениями ядра.

GET\_LOCKTIMING - копировать lock times — если установлено  
DEBUG\_TIME\_LOCKS.

GET\_MACHINE - копировать системное окружение.

GET\_MONPARAMS - копировать набор параметров монитора загрузки.

GET\_PRIVTAB - копировать таблицу системных привилегий.

GET\_PROCTAB - копировать полностью таблицу процессов ядра.

GET\_PROC - копировать одиночный слот таблицы процессов.

GET\_RANDOMNESS - копировать неупорядоченность (randomness) собранную

событиями ядра.

GET\_SCHEDINFO - копировать очереди готовности и процессорную таблицу.

*I\_VAL\_PTR* : виртуальный адрес куда должна быть скопирована информация.

*I\_VAL\_LEN* : максимальная длина с которой вызывающий может оперировать.

*I\_VAL\_PTR2* : опционально, второй адрес. Используется когда копируются диспетчера данные.

*I\_VAL\_LEN2* : опционально, вторая длина. Перезагружается номером процесса.

- возвращаемый результат:

OK : успешный информационный запрос.

EFAULT : был диагностирован недопустимый адрес памяти.

E2BIG : запрашиваемые данные превышают максимум, обеспечиваемый вызывающим.

- библиотечные функции:

```
int_sys_getinfo(int request, void *ptr, int len,
                void *ptr2, int len2);
int_sys_getirqhooks(struct irq_hook *ptr);
int_sys_getimage(struct boot_image *ptr);
int_sys_getkinfo(struct kinfo *ptr);
int_sys_getkmessages(struct kmessages *ptr);
int_sys_getlocktimings(struct lock_timingdata *ptr);
int_sys_getmachine(struct machine *ptr);
int_sys_getmonparams(char *ptr, int max_len);
int_sys_getprivtab(struct priv *ptr);
int_sys_getproctab(struct proc *ptr);
int_sys_getproc(struct proc *ptr, int proc nr);
int_sys_getrandomness(struct randomness *ptr);
int_sys_getschedinfo(struct proc* ptr, struct proc *ptr2);
```

SYS\_GETKTSIG: Проверяет, этот ли есть процесс, который должен получить сигнал. Это многократно выполняется РМ после того, как он получит уведомление, что присутствуют задержанные сигналы ядра.

- параметры запроса:

*SIG\_PROC* : вернуть следующий процесс с задержанными сигналами или NONE.

*SIG\_MAP* : битовая карта с задержанными сигналами ядра.

- возвращаемый результат:

OK : этот вызов всегда успешен.

- библиотечные функции:

```
int sys_getksig(int *proc_nr, sigset_t *sig_map);
```

SYS\_INT86: выполнить BIOS операцию реального режима от имени и по поручению драйвера устройства. Этот вызов временно переключает процессор из 32-бит защищённого режима в 16-бит реальный режим для доступа к вызову операции BIOS. Он присутствует для поддержки драйвера устройства BIOS\_WINI.

- параметры запроса:

*INT86\_REG86* : адрес запроса вызывающей стороны.

- возвращаемый результат:

OK : BIOS вызов успешно произведен.

EFAULT : недопустимый адрес запроса.

- библиотечные функции:

-

**SYS\_IOPENABLE**: разрешить CPU биты I/O уровня привилегий для заданного процесса, что позволяет ему непосредственно осуществлять I/O операции в пользовательском пространстве.

- параметры запроса:

*PROC\_NR* : процесс которому предоставляются привилегии.

- возвращаемый результат:

ОК : этот вызов всегда успешен.

- библиотечные функции:

-

**SYS\_IRQCTL**: установить или сбросить стратегию аппаратного прерывания для заданной IRQ линии, и разрешить или запретить прерывание от этой линии. Этот вызов позволяет драйверу пользовательского пространства захватывать ловушку для использования с типовым обработчиком прерываний ядра. Обработчик прерывания ядра просто уведомляет драйвер о прерывании сообщением **HARD\_INT**, и переразрешает прерывание с IRQ линии, если стратегия требует этого. Уведомляющее сообщение будет содержать *id*, предоставляемое вызывающей стороной как аргумент. Как только стратегия введена в действие, драйвера могут разрешать и запрещать прерывания.

- параметры запроса:

*IRQ\_REQUEST* : управление прерыванием, которое необходимо осуществить.

*IRQ\_SETPOLICY* - установить стратегию прерывания для типового обработчика прерывания.

*IRQ\_RMPOLICY* - удалить ранее установленную стратегию прерывания.

*IRQ\_ENABLE* - разрешить прерывания для заданной IRQ линии.

*IRQ\_DISABLE* - запретить прерывания для заданной IRQ линии.

*IRQ\_VECTOR* : IRQ линия подлежащая управлению.

*IRQ\_POLICY* : битовая карта с флагами, индицирующими стратегию IRQ.

*IRQ\_HOOK\_ID* : когда устанавливается стратегия, здесь предоставляется индекс, который посылается вызывающей стороне при возникновении прерывания. Для всех других запросов это идентификатор ловушки ядра, возвращаемый ядром.

- параметры ответа:

*IRQ\_HOOK\_ID* : идентификатор ловушки ядра, ассоциированный с драйвером.

- возвращаемый результат:

**EINVAL** : ошибка запроса: IRQ линия, *IRQ\_HOOK\_ID*, или номер процесса<sup>6</sup>.

**EPERM** : только владелец *IRQ\_HOOK\_ID* может переключать прерывания или освободить ловушку.

**ENOSPC** : не может быть найдено свободных ловушек.

ОК : запрос успешно обслужен.

- библиотечные функции:

```
int sys_irqctl(int request, int irq_vec, int policy,  
               int *hook_id);
```

```
int sys_irqsetpolicy(int irq_vec, int policy, int *hook_id);
```

```
int sys_irqrmpolicy(int irq_vec, int *hook_id);
```

```
int sys_irqenable(int hook_id);
```

```
int sys_irqdisable(int hook_id);
```

---

<sup>6</sup> Так в оригинале, хотя номер процесса в этом запросе вообще не фигурирует (прим. перев.).

**SYS\_KILL**: послать сигнал процессу от лица системного сервера. Системный процесс может послать сигнал другому процессу этим вызовом. Ядро уведомляет РМ о задержанном сигнале для последующей обработки. (Отметим, что POSIX вызов `kill()` напрямую обрабатывается РМ.) РМ использует этот вызов, чтобы опосредованно послать сигнальное сообщение системному процессу (получателю). Такое случается, когда сигнал поступает для системного процесса, который установил специальный `SIG_MESS` сигнальный обработчик с помощью POSIX вызова `sigaction()`.

- параметры запроса:

*SIG\_PROC\_NR* : процесс, которому будет послан сигнал.

*SIG\_NUMBER* : номер сигнала. Диапазон от 0 до `_NSIG`.

- возвращаемый результат:

ОК : вызов успешный.

`EINVAL` : недопустимый процесс или номер сигнала.

`EPERM` : невозможно послать сигнал задаче ядра; РМ не может послать сигнал пользовательскому процессу посредством уведомляющего сообщения.

- библиотечные функции:

```
int sys_kill(int proc_nr, int sig_nr);
```

**SYS\_MEMSET**: прописать 4-байтным шаблоном указанную область памяти. Этот вызов используется РМ для обнуления BSS сегмента при POSIX вызове `exec()`. Ядро запрашивается для выполнения этой работы из соображений производительности.

- параметры запроса:

*MEM\_PTR* : физический базовый адрес области памяти.

*MEM\_COUNT* : протяжённость в байтах области памяти.

*MEM\_PATTERN* : 4-х байтовый шаблон, который должен быть записан.

- возвращаемый результат:

ОК : этот вызов всегда успешен.

- библиотечные функции:

```
int sys_memset(long pattern, phys_bytes base,  
               phys_bytes length);
```

**SYS\_NEWMAP**: установить новую карту памяти для нового процесса, создаваемого `fork()`, или для случая, если карта памяти процесса изменяется. Ядро выбирает новую карту памяти из РМ и обновляет свои структуры данных.

- параметры запроса:

*PR\_PROC\_NR* : установит новую карту для этого процесса (endpoint процесса).

*PR\_MEM\_PTR* : указатель на карту памяти в РМ.

- возвращаемый результат:

ОК : новое отображение было успешно установлено.

`EFAULT` : некорректный адрес нового отображения памяти.

`EINVAL` : неверный номер процесса.

- библиотечные функции:

```
int sys_newmap(int proc_nr, struct mem_map *ptr);
```

**SYS\_NICE**: Изменить приоритет процесса. Этим достигается изменением значения приоритета между `PRIO_MIN` (отрицательное) и `PRIO_MAX` (положительное). Значение 0



восстанавливает приоритет в умалчиваемое значение.

- параметры запроса:

*PR\_PROC\_NR* : процесс, чей приоритет должен быть изменён.

*PR\_PRIORITY* : новое изменение приоритета для процесса.

- возвращаемый результат:

OK : новое значение приоритета установлено.

EINVAL : недопустимое значение номера процесса или приоритета.

EPERM : невозможно изменить приоритет задачи ядра.

- библиотечные функции:

```
int sys nice(int proc_nr, int priority);
```

*SYS\_PHYSCOPY*: копировать данные, используя физическую адресацию. Источник и приёмник могут быть виртуальными, подобно *SYS\_VIRCOPY*, но в дополнение произвольный физический адрес принимается в качестве *PHYS\_SEG*.

- параметры запроса:

*CP\_SRC\_SPACE* : сегмент источника.

*CP\_SRC\_ADDR* : виртуальный адрес источника.

*CP\_SRC\_PROC\_NR* : номер (endpoint) процесса источника.

*CP\_DST\_SPACE* : сегмент получателя.

*CP\_DST\_ADDR* : виртуальный адрес получателя.

*CP\_DST\_PROC\_NR* : номер (endpoint) процесса получателя.

*CP\_NR\_BYTES* : число байт для копирования.

- возвращаемый результат:

OK : копирование произведено.

EDOM : неверный счётчик копирования.

EFAULT : виртуально в физическое отображение потеряно.

EINVAL : некорректный сегмента тип, или номер процесса.

EPERM : только владелец *REMOTE\_SEG* может копировать в него или из него.

- библиотечные функции:

```
int sys_abcopy(phys_bytes src_phys, phys_bytes dst_phys,  
               phys_bytes count);
```

```
int sys physcopy(int src_proc, int src_seg, vir_bytes src_vir,  
                 int dst_proc, int dst_seg,  
                 vir_bytes dst_vir, phys_bytes count);
```

*SYS\_PHYSVCOPY*: копировать множественные блоки данных, используя физическую адресацию. Запрашиваемый вектор выбирается вызывающей стороной, и каждый элемент вектора обслуживается подобно запросу *SYS\_PHYSCOPY*. Копирование продолжается до тех пор, пока все элементы будут обслужены, или будет наблюдаться ошибка.

- параметры запроса:

*VCP\_VEC\_SIZE* : число элементов в запрошенном векторе.

*VCP\_VEC\_ADDR* : виртуальный адрес вектора, запрошенного вызывающей стороной.

- параметры ответа:

*VCP\_NR\_OK* : число элементов успешно скопированных.

- возвращаемый результат:

OK : копирование выполнено.

EDOM : недопустимый счётчик копирования.

EFAULT : отображение виртуального в физическое потеряно.

EINVAL : копируемый вектор слишком велик, некорректный сегмент,

или недопустимый процесс.

EPERM : только собственник REMOTE\_SEG может копировать в него, или из него.

- библиотечные функции:

```
int sys physvcopy(phys_cp_req *copy_vec,  
                 int vec_size, int *nr_ok);
```

SYS\_PRIVCTL: получение частных структур привилегий, и обновление привилегий процесса. Это используется при динамическом старте системных сервисов.

- параметры запроса:

CTL\_PROC\_NR : процесс, чьи привилегии должны быть обновлены.

- возвращаемый результат:

OK : вызов успешен.

EINVAL : недопустимый номер процесса.

ENOSPC : не найдено свободной структуры привилегий.

- примечания:

Этот системный вызов будет расширять и обеспечивать улучшенную, с двух сторон, и поддержку и проверку защищённости, для серверов или драйверов, которые должны загружаться динамически. Это работа на будущее.

- библиотечные функции:

-

SYS\_SDEVIO: осуществить I/O на устройстве, по поручению драйвера устройства пользовательского пространства. Заметим, что этот запрос поддерживает только байтовую или словную гранулярность операции. Драйвер может запросить ввод или вывод полного буфера. Дополнительно смотри вызовы ядра SYS\_DEVIO и SYS\_VDEVIO.

- параметры запроса:

DIO\_REQUEST : ввод или вывод.

DIO\_INPUT - считать значение из DIO\_PORT.

DIO\_OUTPUT - записать DIO\_VALUE в DIO\_PORT.

DIO\_TYPE : флаг, индицирующий тип значения.

DIO\_BYTE - байт.

DIO\_WORD - слово.

DIO\_PORT : порт для чтения или записи.

DIO\_PROC\_NR : номер (endpoint) процесса где находится буфер.

DIO\_VEC\_ADDR : виртуальный адрес буфера.

DIO\_VEC\_SIZE : число элементов ввода или вывода.

- параметры ответа:

DIO\_VALUE : значение, считанное с заданного порта, для DIO\_INPUT только.

- возвращаемый результат:

OK : операция I/O на порту успешно выполнена.

EINVAL : недопустимый запрос или для порта гранулярность.

EPERM : не возможно выполнение I/O для задачи ядра.

EFAULT : неверный виртуальный адрес буфера.

- библиотечные функции:

```
int sys insb(port_t port, u8_t buffer, int count);  
int sys insw(port_t port, u16_t buffer, int count);  
int sys outsb(port_t port, u8_t buffer, int count);  
int sys outsw(port_t port, u16_t buffer, int count);  
int sys sdevio(int req, long port, int io_type,
```

```
void *buffer, int count);
```

**SYS\_SEGCTL**: добавить сегмент памяти к LDT процесса, и его удалённой картой памяти. Этот вызов возвращает селектор и смещение, которые могут быть использованы для непосредственного доступа к удалённому сегменту, так же, как индекс в удалённой карте памяти, который может быть использован в вызове ядра **SYS\_VIRCOPY**.

- параметры запроса:

*SEG\_PHYS* : физический базовый адрес сегмента.

*SEG\_SIZE* : размер сегмента.

- параметры ответа:

*SEG\_INDEX* : индекс в удалённой карте памяти.

*SEG\_SELECT* : селектор сегмента для LDT входа.

*SEG\_OFFSET* : смещение в пределах сегмента; ноль - в противном случае  
гранулярность 4К используется.

- возвращаемый результат:

**ENOSPC** : нет свободного слота в удалённой карте памяти и LDT.

**OK** : дескриптор сегмента успешно добавлен.

- библиотечные функции:

```
int sys_segctl(int *index, u16_t *seg, vir_bytes *off,  
               phys_bytes phys, vir_bytes size);
```

**SYS\_SIGRETURN**: возврат из обработчика сигнала стиля POSIX. PM запрашивает ядро привести все атрибуты в порядок, прежде, чем сигнализировать что процесс может продолжить выполнение. Также смотри вызов ядра **SYS\_SIGSEND**, который заталкивает фрейм сигнального контекста в стек.

- параметры запроса:

*SIG\_PROC* : индицирует процесс, который уведомляется.

*SIG\_CTX\_PTR* : указатель на контекстную структуру обработчика сигнала стиля  
POSIX.

- параметры ответа:

*SIG\_PROC* : возвращает следующий процесс с отложенными сигналами, или **NONE**  
возвращаемое значение.

- возвращаемый результат:

**OK** : действие по обработке сигнала успешно осуществлено.

**EINVAL** : неверные номер процесса, или контекстная структура.

**EFAULT** : ошибочный адрес контекстной структуры, или невозможно  
скопировать сигнальный фрейм.

- библиотечные функции:

```
int sys_sigreturn(int proc_nr, struct sigmsg *sig_context);
```

**SYS\_SIGSEND**: сигнализировать процессу от лица PM о помещении контекстной структуры в стек. Ядро выбирает структуру, инициализирует её, и копирует её в пользовательский стек.

- параметры запроса:

*SIG\_PROC* : индицирует процесс, который уведомляется.

*SIG\_CTX\_PTR* : указатель на контекстную структуру обработчика сигнала стиля  
POSIX.

- параметры ответа:

*SIG\_PROC* : возвращает следующий процесс с отложенными сигналами,

или NONE возвращаемое значение.

- возвращаемый результат:

OK : действие по обработке сигнала успешно осуществлено.

EINVAL : неверный номер процесса.

EPERM : нельзя сигнализировать задаче ядра.

EFAULT : ошибочный адрес контекстной структуры, или невозможно скопировать сигнальный фрейм.

- библиотечные функции:

```
int sys_sigsend(int proc_nr, struct sigmsg *sig_context);
```

SYS\_SETALARM: установить или сбросить синхронный таймер тревоги. Когда выдержка таймера истечёт, это приведёт в действие то, что SYN\_ALARM уведомляющее сообщение, с текущим оперативным временем как аргумент, будет послано вызывающей стороне. Только системные процессы могут запрашивать синхронные тревоги.

- параметры запроса:

*ALRM\_EXP\_TIME* : абсолютная или относительная выдержка времени, в тиках, для этой тревоги.

*ALRM\_ABS\_TIME* : ноль, если выдержка времени, относительная к текущему оперативному времени.

- параметры ответа:

*ALRM\_TIME\_LEFT* : тиков, оставшихся от предыдущей тревоги.

- возвращаемый результат:

OK: тревога успешно установлена.

EPERM : пользовательский процесс не может запрашивать тревоги.

- библиотечные функции:

```
int sys_setalarm(clock_t expire_time, int abs_flag);
```

SYS\_TIMES: получить оперативное время ядра со времени загрузки и времена выполнения процесса.

- параметры запроса:

*T\_PROC\_NR* : процесс, для которого получить информацию по временам, или NONE.

- параметры ответа:

*T\_USER\_TIME* : время процесса в пользовательском режиме, если валидный номер.

*T\_SYSTEM\_TIME* : время процесса в режиме системы, если валидный номер.

*T\_BOOT\_TICKS* : число тиков с времени последней загрузки MINIX.

- возвращаемый результат:

OK : этот вызов всегда успешен.

- библиотечные функции:

```
int sys_times(int proc_nr, clock_t *ptr);
```

SYS\_TRACE: мониторинг или управление выполнением заданного процесса. Обслуживает отладочные команды, поддерживаемые системным вызовом ptrace( ).

- параметры запроса:

*CTL\_REQUEST* : трассировочный запрос.:

T\_STOP - остановить процесс.

T\_GETINS - вернуть значение из пространства команд.

T\_GETDATA - вернуть значение из пространства данных.

T\_GETUSER - вернуть значение из таблицы процесса.

T\_SETINS - установить значение в пространстве команд.  
T\_SETDATA - установить значение в пространстве данных.  
T\_SETUSER - установить значение в таблице процесса.  
T\_RESUME - возобновить выполнение.  
T\_STEP - установить бит трассировки.

CTL\_PROC\_NR : номер процесса, который будет трассироваться.

CTL\_ADDRESS : виртуальный адрес в пространстве трассируемого процесса.

CTL\_DATA : данные, которые должны быть записаны.

- параметры ответа:

CTL\_DATA : данные, которые возвращены.

- возвращаемый результат:

OK : успешная операция трассировки.

EIO : устанавливаемое или запрашиваемое значение утеряно.

EINVAL : не поддерживаемый запрос трассировки.

PERM : трассироваться может только пользовательский процесс.

- библиотечные функции:

```
int sys_trace(int request, int proc_nr,  
              long addr, long *data_ptr);
```

SYS\_UMAP: отобразить виртуальный адрес в физический и вернуть физический адрес. Виртуальный адрес может быть из LOCAL\_SEG, REMOTE\_SEG, или BIOS\_SEG. Смещение в байтах может быть передано, для того, чтобы убедиться, что оно действительно попадает внутрь сегмента.

- параметры запроса:

CP\_RC\_PROC NR : номер процесса, к которому относится адрес.

CP\_SRC\_SPACE : идентификатор сегмента.

CP\_SRC\_ADDR : смещение в рамках сегмента.

CP\_NR\_BYTES : число байт от начала.

- параметры ответа:

CP\_DST\_ADDR : физический адрес, если отображение успешно.

- возвращаемый результат:

OK : копирование было выполнено<sup>7</sup>.

EFAULT : отображение виртуального в физический утеряно.

EINVAL : некорректный тип сегмента или номер процесса.

- примечания:

Нулевой адрес внутри BIOS\_SEG возвращает EFAULT, тогда как нулевой BIOS вектор прерывания фактически является валидным адресом.

- библиотечные функции:

```
int sys_umap(int proc_nr, int seg, vir_bytes vir_addr,  
             vir_bytes count, phys_bytes *phys_addr);
```

SYS\_VDEVIO: осуществить серию I/O операций от лица пользовательского процесса. Вызов принимает указатель на массив пар (порт, значение) которые должны быть обслужены одновременно. Аппаратные прерывания временно запрещаются, чтобы воспрепятствовать возможности пакета I/O вызовов быть прерванным. Смотри также SYS\_DEVIO и SYS\_SDEVIO.

- параметры запроса:

---

<sup>7</sup> Так в оригинале, но, очевидно, текст сообщения явно «переполз» из другого вызова (прим. перев.).

*DIO\_REQUEST* : ввод или вывод.

*DIO\_INPUT* - читать значение из *DIO\_PORT*.

*DIO\_OUTPUT* - писать *DIO\_VALUE* в *DIO\_PORT*.

*DIO\_TYPE* : флаг, определяющий тип значений.

*DIO\_BYTE* - тип *byte*.

*DIO\_WORD* - тип *word*.

*DIO\_LONG* - тип *long*.

*DIO\_VEC\_SIZE* : число портов, которое должно быть обслужено.

*DIO\_VEC\_ADDR* : виртуальный адрес массива пар (порт, значение) в пространстве вызывающей стороны.

- возвращаемый результат:

OK : операции I/O были успешно выполнены.

EINVAL : ошибочный запрос или гранулярность.

E2BIG : размер вектора превышает максимум, который может быть обслужен.

EFAULT : адрес пар (порт, значение) ошибочный.

- библиотечные функции:

```
int sys_voutb(pvb_pair_t *pvb_vec, int vec_size);
```

```
int sys_voutw(pvw_pair_t *pvw_vec, int vec_size);
```

```
int sys_voutl(pvl_pair_t *pvl_vec, int vec_size);
```

```
int sys_vinb(pvb_pair_t *pvb_vec, int vec_size);
```

```
int sys_vinw(pvw_pair_t *pvw_vec, int vec_size);
```

```
int sys_vinl(pvl_pair_t *pvl_vec, int vec_size);
```

*SYS\_VIRCOPY*: копировать данные используя виртуальную адресацию. Виртуальный адрес может принадлежать трём сегментам: *LOCAL\_SEG* (сегменты кода, стека, данных), *REMOTE\_SEG* (такие как RAM диск, видео память), и *BIOS\_SEG* (*BIOS* вектора прерываний, *BIOS* область данных). Это наиболее общий системный вызов относительно копирования.

- параметры запроса:

*CP\_SRC\_SPACE* : сегмент источника.

*CP\_SRC\_ADDR* : виртуальный адрес источника.

*CP\_SRC\_PROC\_NR* : номер процесса для процесса источника.

*CP\_DST\_SPACE* : сегмент назначения.

*CP\_DST\_ADDR* : виртуальный адрес назначения.

*CP\_DST\_PROC\_NR* : номер процесса для процесса назначения.

*CP\_NR\_BYTES* : число байт для копирования.

- возвращаемый результат:

OK : копирование успешно выполнено.

EDOM : ошибочный счётчик копирования.

EFAULT : отображение виртуального в физический потеряно.

EPERM : нет прав использования *PHYS\_SEG*.

EINVAL : некорректный тип сегмента, или номер процесса.

EPERM : только собственник *REMOTE\_SEG* может копировать в него и из него.

- библиотечные функции<sup>8</sup>:

```
int sys_biosin(vir_bytes bios_vir, vir_bytes dst_vir,  
              vir_bytes bytes);
```

```
int sys_biosout(vir_bytes src_vir, vir_bytes bios_vir,  
               vir_bytes bytes);
```

```
int sys_datacopy(vir_bytes src_proc, vir_bytes src_vir,
```

---

8 Здесь в прототипах функций у автора — явная путаница в параметрах.

```

        dst_proc, dst_vir, vir_bytes bytes);
int sys_textcopy(vir_bytes src_proc, vir_bytes src_vir,
        dst_proc, dst_vir, vir_bytes bytes);
int sys_stackcopy(vir_bytes src_proc, vir_bytes src_vir,
        dst_proc, dst_vir, vir_bytes bytes);
int sys_vircopy(int src_proc, int src_seg, vir_bytes src_vir,
        int dst_proc, int dst_seg, vir_bytes dst_vir,
        phys_bytes bytes);

```

SYS\_VIRVCOPY: копировать множество блоков данных используя виртуальную адресацию. Запрашиваемый вектор выбирается на вызывающей стороне, и каждый элемент обслуживается подобно отдельному запросу SYS\_VIRCOPY. Копирование продолжается до тех пор, пока все элементы будут скопированы, или наступит ошибка.

- параметры запроса:

*VCP\_VEC\_SIZE* : число элементов запрашиваемого вектора.

*VCP\_VEC\_ADDR* : виртуальный адрес запрашиваемого вектора на вызывающей стороне.

- параметры ответа:

*VCP\_VEC\_OK* : число элементов успешно скопированных.

- возвращаемый результат:

OK : копирование успешно выполнено.

EDOM : ошибочный счётчик копирования.

EFAULT : отображение виртуального в физический потеряно.

EPERM : нет прав использования PHYS\_SEG.

EINVAL : вектор копирования слишком велик, некорректный тип сегмента, или недопустимый процесс.

EPERM : только собственник REMOTE\_SEG может копировать в него и из него.

- библиотечные функции:

```

int sys_virvcopy(vir_cp_req *copy_vec, int vec_size,
        int *nr_ok);

```

Оригинал статьи находится по адресу:

<http://www.minix3.ru/docs/kernel-api.pdf> «MINIX 3 Kernel API»

Перевод: О. И. Цилюрик

Редакция от 18.12.2009

Последняя корректура 23.06.2010 (улучшена читабельность, форматирование, исправлены опечатки — смысловая часть не претерпела изменений).