ГЛАВА 10

Системные вызовы

Операционные системы предлагают процессам, работающим в режиме пользователя, набор интерфейсов для взаимодействия с аппаратными устройствами, такими как процессор, диски или принтеры. Создание дополнительного программного слоя между приложением и аппаратной частью компьютера имеет ряд преимуществ. Во-первых, это облегчает программирование, потому что программисты избавлены от необходимости изучать низкоуровневые характеристики аппаратных устройств. Во-вторых, такой подход существенно повышает безопасность системы, поскольку ядро может проверить корректность запроса на уровне интерфейса до выполнения этого запроса. Наконец, что не менее важно, подобные интерфейсы делают программы более переносимыми, позволяя компилировать и корректно выполнять их в каждом ядре, предлагающем такой же набор интерфейсов.

В Unix-системах большинство интерфейсов между процессами режима пользователя и аппаратными устройствами реализовано с помощью *системных вызовов*, направляемых ядру. В этой главе подробно рассматривается, как Linux реализует системные вызовы, передаваемые ядру пользовательскими программами.

API-интерфейсы стандарта POSIX и системные вызовы

Мы начнем с обсуждения различия между API (Application Programmer Interface, интерфейс прикладного программирования) и системным вызовом. Первый представляет собой определение функции, описывающее, как полу-

чить данную услугу, а второй является явным запросом к ядру, выполненным с помощью программного прерывания.

Unix-системы включают в себя несколько библиотек функций, предоставляющих программистам API-интерфейсы. Некоторые из интерфейсов, определенных в стандартной библиотеке C, называемой libc, обращаются к *интерфейсным процедурам* (процедурам, единственным назначением которых является выдача системного вызова). Обычно каждый системный вызов имеет интерфейсную процедуру, определяющую API-интерфейс, которому должна следовать прикладная программа.

Между прочим, обратное неверно. Интерфейс API не обязательно соответствует какому-либо системному вызову. Во-первых, API может предлагать свои услуги непосредственно в режиме пользователя. (Для таких абстрактных функций, какими являются математические, нет смысла делать системные вызовы.) Во-вторых, одна API-функция может сделать несколько системных вызовов. Более того, несколько API-функций могут делать один и тот же системный вызов, но "нагружать" его дополнительными особенностями. Например, в Linux API-функции `malloc()`, `calloc()` и `free()` реализованы в библиотеке libc. Код в этой библиотеке отслеживает запросы на выделение и освобождение памяти и использует системный вызов `brk()` для увеличения или уменьшения кучи процесса *(см. главу 9)*.

Стандарт POSIX описывает API-интерфейсы, но не системные вызовы. Система может быть сертифицирована как удовлетворяющая стандарту POSIX, если она предоставляет прикладным программам надлежащий набор API-интерфейсов, независимо от того, как реализованы соответствующие функции. На практике несколько систем, не являющихся Unix-подобными, были сертифицированы как удовлетворяющие стандарту POSIX, поскольку они предлагали все традиционные службы Unix в своих библиотеках режима пользователя.

С точки зрения программиста, различие между API и системным вызовом не имеет значения. Единственное, что ему нужно знать, — это имя функции, типы параметров и смысл возвращаемого значения. Зато с точки зрения разработчика ядра это различие принципиально, ведь системные вызовы принадлежат ядру, а библиотеки режима пользователя — нет.

Большинство интерфейсных процедур возвращает целое значение, смысл которого зависит от системного вызова. Код возврата −1, как правило, указывает на то, что ядро не смогло удовлетворить запрос процесса. Неудача обработчика системного вызова могла быть следствием некорректных параметров, недостатка системных ресурсов, аппаратных проблем и т. п. Конкретный код ошибки содержится в переменной `errno`, которая определена в библиотеке libc.

Каждый код ошибки определен в виде макроконстанты, возвращающей положительное значение. Стандарт POSIX устанавливает имена макросов для некоторых кодов ошибок. В Linux, на платформе 80×86 , эти макросы определены в заголовочном файле include/asm-i386/errno.h. Для обеспечения переносимости программ на языке C между системами Unix заголовочный файл include/asm-i386/errno.h, в свою очередь, включен в стандартный заголовочный файл библиотеки C, называемый /usr/include/errno.h. Другие системы имеют свои собственные специализированные подкаталоги для заголовочных файлов.

Обработчик системного вызова и служебные процедуры

Когда процесс режима пользователя делает системный вызов, процессор переключается в режим ядра и приступает к выполнению функции ядра. Как мы увидим в следующем разделе, в архитектуре 80×86 системный вызов Linux может быть сделан двумя разными способами. Впрочем, в обоих случаях результатом будет переход на некоторую ассемблерную функцию, называемую *обработчиком системного вызова*.

Поскольку в ядре реализовано много разных системных вызовов, процесс режима пользователя должен передавать параметр, содержащий *номер системного вызова*, который служит для идентификации вызова. В Linux для этой цели используется регистр eax. Как мы увидим в *разд. "Передача параметров" далее в этой главе*, при совершении системного вызова обычно передаются и дополнительные параметры.

Все системные вызовы возвращают целые значения. Соглашения, принятые для этих возвращаемых значений, отличны от тех, что приняты для интерфейсных процедур. В ядре положительное или нулевое значение свидетельствует об успешном завершении системного вызова, а отрицательное — об ошибке. В последнем случае абсолютная величина значения является кодом ошибки, который должен быть возвращен прикладной программе в переменной errno. Ядро никак не использует эту переменную, зато интерфейсные процедуры записывают в нее значение после возврата управления от системного вызова.

Обработчик системного вызова, имеющий структуру, аналогичную структуре других обработчиков прерываний, выполняет следующие действия:

- сохраняет содержимое большинства регистров в стеке режима ядра (эта операция является общей для всех системных вызовов, и она закодирована на языке ассемблера);

- обрабатывает системный вызов с помощью специальной функции, написанной на языке C, которая называется *служебной процедурой системного вызова*;
- выполняет корректный выход: загружает в регистры значения, сохраненные в стеке режима ядра, и переключает процессор из режима ядра в режим пользователя (эта операция является общей для всех системных вызовов, и она закодирована на языке ассемблера).

Имя служебной процедуры, ассоциированной с системным вызовом `xyz()`, обычно имеет формат `sys_xyz()`; впрочем, из этого правила есть несколько исключений.

На рис. 10.1 проиллюстрированы взаимоотношения между прикладной программой, сделавшей системный вызов, соответствующей интерфейсной процедурой, обработчиком системного вызова и служебной процедурой системного вызова. Стрелки обозначают передачу управления от одной функции к другой. Конструкции "`SYSCALL`" и "`SYSEXIT`" являются условными заменителями реальных команд на языке ассемблера, которые переключают процессор, соответственно, из режима пользователя в режим ядра и обратно.

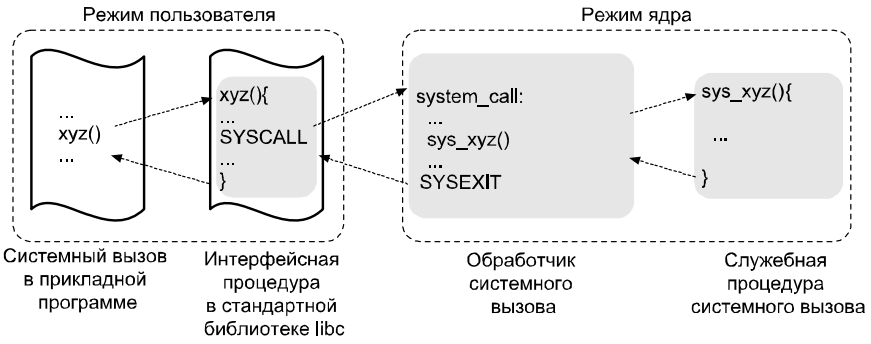

Рис. 10.1. Выполнение системного вызова

Чтобы ассоциировать номер каждого системного вызова с соответствующей служебной процедурой, ядро пользуется *таблицей диспетчеризации системных вызовов*, которая хранится в массиве `sys_call_table` и содержит `NR_syscalls` записей (289 в ядре Linux 2.6.11). В таблице *n*-я запись содержит адрес служебной процедуры для системного вызова с номером *n*.

Макрос `NR_syscalls` является всего лишь статическим ограничением количества системных вызовов, реализуемых в системе; он не показывает фактическое количество реализованных системных вызовов. В реальности любая

запись таблицы может содержать адрес функции `sys_ni_syscall()`, которая представляет собой служебную процедуру "нереализованных" системных вызовов и просто возвращает код ошибки `-ENOSYS`.

Вход в системный вызов и выход из него

Приложения, "родные" для Linux[1], могут делать системные вызовы двумя различными способами:

- выполнив ассемблерную инструкцию `int $0x80`; в ранних версиях ядра Linux это был единственный способ переключения из режима пользователя в режим ядра;
- выполнив ассемблерную инструкцию `sysenter`, появившуюся в микропроцессорах Intel Pentium II; эта инструкция теперь поддерживается ядром Linux 2.6.

Аналогичным образом, ядро может выйти из системного вызова (и тем самым вернуть процессор в режим пользователя) двумя способами:

- выполнив ассемблерную инструкцию `iret`;
- выполнив ассемблерную инструкцию `sysexit`, появившуюся в микропроцессорах Intel Pentium II, вместе с инструкцией `sysenter`.

Однако поддержка двух разных способов входа в ядро является не таким простым делом, как может показаться:

- ядро должно поддерживать как старые библиотеки, в которых используется только инструкция `int $0x80`, так и новые, в которых используется и инструкция `sysenter`;
- стандартная библиотека, в которой применяется только инструкция `sysenter`, должна уметь работать со старыми версиями ядра, которые поддерживают только инструкцию `int $0x80`;
- ядро и стандартная библиотека должны уметь работать как на старых процессорах, не имеющих инструкции `sysenter`, так и на новых, у которых она есть.

Мы увидим, как Linux справляется с этими трудностями, в *разд. "Выполнение системного вызова с помощью инструкции sysenter" далее в этой главе*.

---

[1] Как мы увидим в *разд. "Области выполнения" главы 20*, Linux может выполнять программы, откомпилированные под другие операционные системы. Следовательно, ядро обеспечивает режим совместимости для входа в системный вызов. Процессы режима пользователя, выполняющие программы операционных систем iBCS и Solaris /x86, могут перейти в режим ядра с помощью подходящего шлюза вызовов из принятой по умолчанию локальной таблицы дескрипторов *(см. разд. "The Linux LDTs" в главе 2)*.

Выполнение системного вызова с помощью инструкции *int $0x80*

"Традиционным" способом выполнения системного вызова является выполнение ассемблерной инструкции `int`, которая была представлена в *главе 4*.

Вектор 128 (`0x80` в шестнадцатеричной системе счисления) ассоциирован с точкой входа в ядро. Функция `trap_init()`, вызываемая на этапе инициализации ядра, устанавливает запись таблицы дескрипторов прерываний, соответствующую вектору 128, следующим образом:

```
set_system_gate(0x80, &system_call);
```

Этот код загружает в поля дескриптора шлюза *(см. главу 4)* следующие значения:

- селектор сегмента — селектор сегмента кода ядра `__KERNEL_CS`;
- смещение — указатель на обработчик системного вызова `system_call()`;
- тип — число 15, означающее, что исключение имеет тип `Trap`, а соответствующий обработчик не отключает маскируемые прерывания;
- DPL (Descriptor Privilege Level, уровень привилегий дескриптора) — число 3, означающее, что процессам режима пользователя разрешено вызывать обработчик исключений *(см. главу 4)*.

Таким образом, когда процесс режима пользователя выдает инструкцию `int $0x80`, процессор переключается в режим ядра и приступает к выполнению инструкций, начиная с адреса `system_call`.

Функция *system_call()*

Функция `system_call()` начинает работу с того, что сохраняет в стеке номер системного вызова и все регистры процессора, которые могут понадобиться обработчику исключений, кроме регистров `eflags`, `cs`, `eip`, `ss` и `esp`, автоматически сохраненных блоком управления. Макрос `SAVE_ALL`, описанный в *главе 4*, загружает селектор сегмента данных ядра в регистры `ds` и `es`:

```
system_call:
  pushl %eax
  SAVE_ALL
  movl $0xffffe000, %ebx /* or 0xfffff000 for 4-KB stacks */
  andl %esp, %ebx
```

Затем функция сохраняет адрес структуры `thread_info` текущего процесса в регистре `ebx` *(см. главу 3)*. С этой целью значение указателя на стек ядра округляется до числа, кратного 4 или 8 Кбайт.

После этого функция system_call() проверяет, установлен ли хотя бы один из флагов TIF_SYSCALL_TRACE и TIF_SYSCALL_AUDIT в поле flags структуры thread_info, т. е. отслеживает ли отладчик системные вызовы, которые делает выполняемая программа. В случае положительного результата проверки функция system_call() дважды вызывает функцию do_syscall_trace(): непосредственно перед и сразу после выполнения служебной процедуры данного системного вызова. Эта функция останавливает процесс current и позволяет отлаживающему процессу собрать информацию о нем.

Затем проверяется допустимость номера системного вызова, переданного процессом режима пользователя. Если он больше или равен количеству записей в таблице передачи системных вызовов, обработчик системного вызова завершает работу:

```
  cmpl $NR_syscalls, %eax
  jb nobadsys
  movl $(-ENOSYS), 24(%esp)
  jmp resume_userspace
nobadsys:
```

Если номер системного вызова оказался недопустимым, функция записывает значение -ENOSYS в ячейку стека, в который был сохранен регистр eax, т. е. в ячейку со смещением 24 от текущей верхушки стека. Затем функция переходит по адресу resume_userspace (см. далее). Таким образом, когда процесс возобновит свое выполнение в режиме пользователя, он обнаружит в регистре eax отрицательный код возврата.

Наконец, функция вызывает служебную процедуру, ассоциированную с номером системного вызова, хранящимся в регистре eax:

```
call *sys_call_table(0, %eax, 4)
```

Поскольку в таблице системных вызовов каждая запись имеет длину 4 байта, ядро находит адрес нужной служебной процедуры, умножая номер системного вызова на 4, прибавляя к произведению начальный адрес таблицы sys_call_table и извлекая указатель на служебную процедуру из соответствующей записи в таблице.

Выход из системного вызова

Когда служебная процедура системного вызова завершает работу, функция system_call() получает код возврата из регистра eax и записывает его в то место стека, где было сохранено содержимое регистра eax в режиме пользователя:

```
movl %eax, 24(%esp)
```

В результате процесс режима пользователя найдет код возврата системного вызова в регистре eax.

Затем функция system_call() отключает локальные прерывания и проверяет флаги в структуре thread_info процесса current:

```
cli
movl 8(%ebp), %ecx
testw $0xffff, %cx
je restore_all
```

Поле flags имеет смещение 8 в структуре thread_info, а маска 0xffff выделяет биты, соответствующие всем флагам, перечисленным в табл. 4.15, кроме флага TIF_POLLING_NRFLAG. Если ни один из этих флагов не установлен, функция переходит на метку restore_all. Как сказано в *главе 4*, код, расположенный по этому адресу, восстанавливает содержимое регистров, сохраненное в стеке режима ядра, и выполняет ассемблерную инструкцию iret, чтобы возобновить выполнение процесса в режиме пользователя (см. блок-схему на рис. 4.6).

Если хотя бы один из флагов установлен, функция должна проделать определенную работу до возвращения в режим пользователя. Если установлен флаг TIF_SYSCALL_TRACE, функция system_call() второй раз вызывает функцию do_syscall_trace(), а затем переходит на метку resume_userspace. Если же флаг TIF_SYSCALL_TRACE сброшен, функция переходит на метку work_pending.

Код, расположенный по адресам resume_userspace и work_pending, проверяет наличие запроса на перепланирование процессов, режим виртуального 8086, наличие сигналов, ожидающих доставки, и режим пошагового выполнения. Затем в любом случае совершается переход на метку restore_all, чтобы возобновилось выполнение процесса в режиме пользователя.

Выполнение системного вызова с помощью инструкции *sysenter*

Ассемблерная инструкция int является медленной по своей природе, потому что она выполняет ряд проверок на непротиворечивость и безопасность. (Эта инструкция подробно описана в *главе 4*.)

Инструкция sysenter, получившая в документации Intel название "Быстрый системный вызов", предоставляет более быстрый способ переключения из режима пользователя в режим ядра.

Функция *sysenter*

Ассемблерная инструкция `sysenter` использует три специальных регистра, в которые должна быть загружена следующая информация[2]:

- `SYSENTER_CS_MSR` — селектор сегмента кода ядра;
- `SYSENTER_EIP_MSR` — линейный адрес точки входа;
- `SYSENTER_ESP_MSR` — указатель стека ядра.

Когда выполняется инструкция `sysenter`, блок управления процессора:

- копирует содержимое регистра `SYSENTER_CS_MSR` в регистр `cs`;
- копирует содержимое регистра `SYSENTER_EIP_MSR` в регистр `eip`;
- копирует содержимое регистра `SYSENTER_ESP_MSR` в регистр `esp`;
- прибавляет 8 к значению в регистре `SYSENTER_CS_MSR` и загружает результат в регистр `ss`.

Таким образом, процессор переключается в режим ядра и начинает выполнять первую инструкцию в точке входа ядра. Как мы видели в *главе 2*, сегмент стека ядра совпадает с сегментом данных ядра, а соответствующий дескриптор следует за дескриптором сегмента кода ядра в глобальной таблице дескрипторов. Следовательно, на шаге 4 в регистр `ss` загружается правильный селектор сегмента.

Три регистра, специфичных для модели, инициализируются функцией `enable_sep_cpu()`, которая выполняется один раз каждым процессором в системе на этапе инициализации ядра. Эта функция выполняет следующие действия:

1. Записывает селектор сегмента кода ядра (`__KERNEL_CS`) в регистр `SYSENTER_CS_MSR`.
2. Записывает линейный адрес функции `sysenter_entry()` в регистр `SYSENTER_CS_EIP`.
3. Вычисляет линейный адрес конца локального сегмента TSS и записывает это значение в регистр `SYSENTER_CS_ESP`[3].

Здесь необходимо прокомментировать запись значения в регистр `SYSENTER_CS_ESP`. Когда начинается выполнение системного вызова, стек ядра пуст, и,

---

[2] "MSR" является сокращением для "Model-Specific Register" (регистр, специфичный для модели) и обозначает регистр, имеющийся только в некоторых моделях микропроцессоров 80x86.

[3] Кодирование адреса локального сегмента TSS, записанного в регистр `SYSENTER_ESP_MSR`, происходит вследствие того факта, что регистр должен указывать на реальный стек, который растет в направлении уменьшения адресов. На практике сработает инициализация регистра любым значением, при условии, что по этому значению можно будет получить адрес локального сегмента TSS.

следовательно, регистр esp должен указывать на конец четырех- или восьмикилобайтовой области памяти, включающей в себя стек ядра и дескриптор текущего процесса (см. рис. 3.2). В режиме пользователя интерфейсная процедура не может корректно установить этот регистр, поскольку она не знает адреса этой области памяти. С другой стороны, значение должно быть записано в регистр до переключения в режим ядра. Поэтому ядро инициализирует регистр, чтобы закодировать адрес сегмента состояния задачи (TSS) локального процессора. Как было сказано в описании шага 3 функции __switch_to() *(см. главу 3)*, при каждом переключении процесса ядро сохраняет указатель стека ядра, которым пользуется текущий процесс, в поле esp0 локального сегмента TSS. Таким образом, обработчик системного вызова считывает содержимое регистра esp, вычисляет адрес поля esp0 локального сегмента TSS и загружает "правильный" указатель стека ядра в регистр esp.

Страница vsyscall

Интерфейсная функция из стандартной библиотеки libc может применять инструкцию sysenter, только если эту инструкцию поддерживает как процессор, так и ядро Linux.

Эта проблема совместимости требует довольно сложного решения. Фактически, на этапе инициализации функция sysenter_setup() строит страничный кадр, называемый *страницей vsyscall*, который содержит небольшой совместно используемый ELF-объект (то есть маленькую динамическую ELF-библиотеку). Когда процесс делает системный вызов execve(), чтобы выполнить ELF-программу, код на странице vsyscall динамически компонуется с адресным пространством процесса *(см. разд. "Функции exec" главы 20)*. Код на странице vsyscall использует наиболее подходящую инструкцию для выдачи системного вызова.

Функция sysenter_setup() выделяет новый страничный кадр для страницы vsyscall и связывает его физический адрес с фиксированно отображаемым линейным адресом FIX_VSYSCALL *(см. главу 2)*. Затем функция копирует на страницу один из двух заранее определенных совместно используемых ELF-объектов:

- если процессор не поддерживает инструкцию sysenter, функция строит страницу vsyscall, включающую в себя такой код:

```
__kernel_vsyscall:
  int
$0x80
  ret
```

- если же процессор поддерживает инструкцию `sysenter`, функция строит страницу vsyscall, включающую в себя такой код:

```
__kernel_vsyscall:
  pushl %ecx
  pushl %edx
  pushl %ebp
  movl %esp, %ebp
  sysenter
```

Когда интерфейсная процедура из стандартной библиотеки должна сделать системный вызов, она вызывает функцию `__kernel_vsyscall()`, независимо от того, каков код последней.

Еще одна проблема совместимости возникает из-за того, что старые версии ядра Linux не поддерживают инструкцию `sysenter`. В этом случае, конечно, ядро не строит страницу vsyscall, а функция `__kernel_vsyscall()` не компонуется с адресным пространством процессов в режиме пользователя. Когда функции из новых стандартных библиотек распознают этот факт, они просто выполняют инструкцию `int $0x80` для выполнения системного вызова.

Вход в системный вызов

Последовательность действий, выполняемых, когда системный вызов делается с помощью инструкции `sysenter`, такова:

1. Интерфейсная процедура из стандартной библиотеки загружает номер системного вызова в регистр `eax` и вызывает функцию `__kernel_vsyscall()`.
2. Функция `__kernel_vsyscall()` сохраняет в стеке режима пользователя содержимое регистров `ebp`, `edx` и `ecx` (эти регистры будут использованы обработчиком системного вызова), копирует указатель пользовательского стека в регистр `ebp` и выполняет инструкцию `sysenter`.
3. Процессор переключается из режима пользователя в режим ядра, и ядро приступает к выполнению функции `sysenter_entry()` (на которую указывает регистр `SYSENTER_EIP_MSR`).
4. Функция `sysenter_entry()`, написанная на ассемблере, выполняет следующие действия:

   - устанавливает указатель стека ядра:

     ```
     movl -508(%esp), %esp
     ```

   - изначально регистр `esp` указывает на адрес, непосредственно следующий за локальным сегментом TSS, имеющим длину 512 байтов. Следовательно, инструкция загружает в регистр `esp` содержимое поля,

Примечание [TR1]: Далее в оригинале идут вложенные нумерованные списки, т.к. описана последовательность действий.

имеющего смещение 4 в локальном сегменте TSS, т. е. содержимое поля esp0. Как было сказано ранее, в поле esp0 всегда находится указатель стека ядра для текущего процесса;

- включает локальные прерывания:

  ```
  sti
  ```

- сохраняет в стеке режима ядра селектор сегмента пользовательских данных, текущий указатель пользовательского стека, регистр eflags, селектор сегмента пользовательского кода и адрес инструкции, которую следует выполнить при выходе из системного вызова:

  ```
  pushl $(__USER_DS)
  pushl %ebp
  pushfl
  pushl $(__USER_CS)
  pushl $SYSENTER_RETURN
  ```

- заметьте, что эти инструкции эмулируют некоторые операции, выполняемые ассемблерной инструкцией int *(см. главу 4)*;

- восстанавливает в регистре ebp его оригинальное значение, переданное интерфейсной процедурой:

  ```
  movl (%ebp), %ebp
  ```

- эта инструкция делает именно то, что нужно, поскольку функция __kernel_vsyscall() сохранила в стеке режима пользователя оригинальное значение регистра ebp, а затем загрузила в этот регистр текущее значение указателя пользовательского стека;

- обращается к обработчику системного вызова, выполняя последовательность инструкций, идентичную той, что начинается за меткой system_call (ранее в этой главе).

Выход из системного вызова

Когда служебная процедура системного вызова завершает работу, функция sysenter_entry() выполняет практически те же действия, что и функция system_call() *(см. предыдущий раздел)*. Во-первых, она считывает код возврата служебной процедуры, хранящийся в регистре eax, и сохраняет его в той ячейке стека ядра, в которой было сохранено значение регистра eax в режиме пользователя. Затем функция отключает локальные прерывания и проверяет флаги в структуре thread_info процесса current.

Если хотя бы один из флагов установлен, значит, перед возвратом в режим пользователя должна быть проделана определенная работа. Чтобы избежать дублирования кода, этот случай обрабатывается точно так же, как и в функции `system_call()`, т. е. управление передается по адресу `resume_userspace` или `work_pending` (см. рис. 4.6).

В конечном счете ассемблерная инструкция `iret` извлекает из стека режима ядра пять аргументов, сохраненных функцией `sysenter_entry()` на шаге 4, что приводит к переключению процессора в режим пользователя и выполнению кода, расположенного за меткой `SYSENTER_RETURN`.

Если функция `sysenter_entry()` обнаруживает, что все флаги сброшены, она выполняет быстрый возврат в режим пользователя:

```
movl 40(%esp), %edx
movl 52(%esp), %ecx
xorl %ebp, %ebp
sti
sysexit
```

В регистры `edx` и `ecx` загружаются значения из стека, сохраненные функцией `sysenter_entry()` на шаге 4 *(см. предыдущий раздел)*. Регистр `edx` получает адрес метки `SYSENTER_RETURN`, а регистр `ecx` — текущий указатель пользовательского стека.

Инструкция *sysexit*

Ассемблерная инструкция `sysexit` является парной для инструкции `sysenter`: она позволяет быстро переключиться из режима ядра в режим пользователя. Когда эта инструкция выполняется, управляющий блок процессора совершает следующие действия:

1. Прибавляет 16 к значению в регистре `SYSENTER_CS_MSR` и загружает результат в регистр `cs`.
2. Копирует содержимое регистра `edx` в регистр `eip`.
3. Прибавляет 24 к значению в регистре `SYSENTER_CS_MSR` и загружает результат в регистр `ss`.
4. Копирует содержимое регистра `ecx` в регистр `esp`.

Поскольку в регистре `SYSENTER_CS_MSR` находится селектор сегмента кода ядра, в регистр `cs` загружается селектор сегмента пользовательского кода, а в регистр `ss` — селектор сегмента пользовательских данных *(см. главу 2)*.

В результате процессор переключается из режима ядра в режим пользователя и приступает к выполнению инструкции, адрес которой находится в регистре edx.

Код *SYSENTER_RETURN*

Код за меткой SYSENTER_RETURN расположен на странице vsyscall, и он выполняется, когда выполнение системного вызова, сделанного с помощью инструкции sysenter, заканчивается с помощью инструкции iret или sysexit.

Код просто восстанавливает оригинальное содержимое регистров ebp, edx и ecx, сохраненных в стеке режима пользователя, и возвращает управление интерфейсной процедуре из стандартной библиотеки:

```
SYSENTER_RETURN:
  popl %ebp
  popl %edx
  popl %ecx
  ret
```