**Linux - syscalls. Системные вызовы в Linux.**

*О многом - молвил Морж,- пришла пора поговорить.  
Л. Кэролл (Цитата по книге Б. Страустрапа)*

*Как бы у нас ни сложилось дальше, я благодарен тебе за то прекрасное время,  
что у нас было и что навсегда останется со мною.  
В появлении этой статьи есть и твоя заслуга...*

**Вместо введения.**

По теме внутреннего устройства ядра Linux в общем, его различных подсистемах и о системных вызовах в частности, было писано и переписано уже порядком. Наверно, каждый уважающий себя автор должен хоть раз об этом написать, подобно тому, как каждый уважающий себя программист обязательно должен написать свой собственный файл-менеджер :) Хотя я и не являюсь профессиональным IT-райтером, да и вообще, делаю свои пометки исключительно для своей пользы в первую очередь, чтобы не забыть изученное слишком быстро. Но, если уж кому-то пригодятся мои путевые заметки, конечно, буду только рад. Ну и вообще, кашу маслом не испортишь, поэтому, может даже мне и удастся написать или описать что-то такое, о чём никто не удосужился упомянуть.

**Теория. Что такое системные вызовы?**

Когда непосвящённым объясняют, что такое ПО (или ОС), то говорят обычно следующее: компьютер сам по себе - это железяка, а вот софт - это то, благодаря чему от этой железяки можно получить какую-то пользу. Грубовато, конечно, но в целом, в чём-то верно. Так же я сказал бы наверно об ОС и системных вызовах. На самом деле, в разных ОС системные вызовы могут быть реализованы по-разному, может розниться число этих самых вызовов, но так или иначе, в том или ином виде механизм системных вызовов есть в любой ОС. Каждый день пользователь явно или не явно работает с файлами. Он конечно может явно открыть файл для редактирования в своём любимом MS Word'е или Notepad'е, а может и просто запустить игрушку, исполняемый образ которой, к слову, тоже хранится в файле, который, в свою очередь, должен открыть и прочитать загрузчик исполняемых файлов. В свою очередь игрушка также может открывать и читать десятки файлов в процессе своей работы. Естественно, файлы можно не только читать, но и писать (не всегда, правда, но здесь речь не о разделении прав и дискретном доступе :) ). Всем этим заведует ядро (в микроядерных ОС ситуация может отличаться, но мы сейчас будем ненавязчиво клониться к объекту нашего обсуждения - Linux, поэтому проигнорируем этот момент). Само по себе порождение нового процесса - это также услуга, предоставляемая ядром ОС. Всё это замечательно, как и то, что современные процессоры работают на частотах гигагерцевых диапазонов и состоят из многих миллионов транзисторов, но что дальше? Да то, что если бы не было некого механизма, с помощью которого пользовательские приложения могли выполнять некоторые достаточно обыденные и, в то же время, нужные вещи (*на самом деле, эти тривиальные действия при любом раскладе выполняются не пользовательским приложением, а ядром ОС - авт.*), то ОС была просто вещью в себе - абсолютно бесполезной или же напротив, каждое пользовательское приложение само по себе должно было бы стать операционной системой, чтобы самостоятельно обслуживать все свои нужды. Мило, не правда ли?

Таким образом, мы подошли к определению системного вызова в первой аппроксимации: системный вызов - это некая услуга, которую ядро ОС оказывает пользовательскому приложению по запросу последнего. Такой услугой может быть уже упомянутое открытие файла, его создание, чтение, запись, создание нового процесса, получение идентификатора процесса (pid), монтирование файловой системы, останов системы, наконец. В реальной жизни системных вызовов гораздо больше, нежели здесь перечислено.

Как выглядит системный вызов и что из себя представляет? Ну, из того, что было сказано выше, становится ясно, что системный вызов - это подпрограмма ядра, имеющая соответственный вид. Те, кто имел опыт программирования под Win9х/DOS, наверняка помнят прерывание int 0x21 со всеми (или хотя бы некоторыми) его многочисленными функциями. Однако, есть одна небольшая особенность, касающаяся всех системных вызовов Unix. По соглашению функция, реализующая системный вызов, может принимать N аргументов или не принимать их вовсе, но так или иначе, функция должна возвращать значение типа int. Любое неотрицательное значение трактуется, как успешное выполнение функции системного вызова, а стало быть и самого системного вызова. Значение меньше нуля является признаком ошибки и одновременно содержит код ошибки (коды ошибок определяются в заголовках include/asm-generic/errno-base.h и include/asm-generic/errno.h). В Linux шлюзом для системных вызовов до недавнего времени было прерывание int 0x80, в то время, как в Windows (вплоть до версии XP Service Pack 2, если не ошибаюсь) таким шлюзом является прерывание 0x2e. Опять же, в ядре Linux, до недавнего времени все системные вызовы обрабатывались функцией system\_call(). Однако, как выяснилось позднее, классический механизм обрабатки системных вызовов через шлюз 0x80 приводит к существенному падению производительности на процессорах Intel Pentium 4. Поэтому на смену классическому механизму пришёл метод виртуальных динамических разделяемых объектов (DSO - динамический разделяемый объектный файл. Не ручаюсь за правильный перевод, но DSO, это то, что пользователям Windows известно под названием DLL - динамически загружаемая и компонуемая библиотека) - VDSO. В чём отличие нового метода от классического? Для начала разберёмся с классическим методом, работающим через гейт 0x80.

**Классический механизм обслуживания системных вызовов в Linux.**

**Прерывания в архитектуре х86.**

Как было сказано выше, ранее для обслуживания запросов пользовательских приложений использовался шлюз 0x80 (int 0x80). Работа системы на базе архитектуры IA-32 управляется прерываниями (строго говоря, это касается вообще всех систем на базе x86). Когда происходит некое событие (новый тик таймера, какая-либо активность на некотором устройстве, ошибки - деление на ноль и пр.), генерируется прерывание. Прерывание (interrupt) названо так, потому что оно, как правило, прерывает нормальный ход выполнения кода. Прерывания принято подразделять на аппаратные и программные (hardware and software interrupts). Аппаратные прерывания - это прерывания, которые генерируются системными и периферическими устройствами. При возникновении необходимости какого-то устройства привлечь к себе внимание ядра ОС оно (устройство) генерирует сигнал на своей линии запроса прерывания (IRQ - Interrupt ReQuest line). Это приводит к тому, что на определённых входах процессора формируется соответствующий сигнал, на основе которого процессор и принимает решение прервать выполнение потока инструкций и передать управление на обработчик прерывания, который уже выясняет, что произошло и что необходимо сделать. Аппаратные прерывания асинхронны по своей природе. Это значит, что прерывание может возникнуть в любое время. Кроме периферийных устройств, сам процессор может вырабатывать прерывания (или, точнее, аппаратные исключения - Hardware Exceptions - например, уже упомянутое деление на ноль). Делается это с тем, чтобы уведомить ОС о возникновении нештатной ситуации, чтобы ОС могла предпринять некое действие в ответ на возникновение такой ситуации. После обработки прерывания процессор возвращается к выполнению прерванной программы. Прерывание может быть инициировано пользовательским приложением. Такое прерывание называют программным. Программные прерывания, в отличие от аппаратных, синхронны. Т.е., при вызове прерывания, вызвавший его код приостанавливается, пока прерывание не будет обслужено. При выходе из обработчика прерывания происходит возврат по дальнему адресу, сохранённому ранее (при вызове прерывания) в стеке, на следующую инструкцию после инструкции вызова прерывания (int). Обработчик прерывания - это резидентный (постоянно находящийся в памяти) участок кода. Как правило, это небольшая программа. Хотя, если мы будем говорить о ядре Linux, то там обработчик прерывания не всегда такой уж маленький. Обработчик прерывания определяется вектором. Вектор - это ни что иное, как адрес (сегмент и смещение) начала кода, который должен обрабатывать прерывания с данным индексом. Работа с прерываниями существенно отличается в реальном (Real Mode) и защищённом (Protected Mode) режиме работы процессора (напомню, что здесь и далее мы подразумеваем процессоры Intel и совместимые с ними). В реальном (незащищённом) режиме работы процессора обработчики прерываний определяются их векторами, которые хранятся всегда в начале памяти выборка нужного адреса из таблицы векторов происходит по индексу, который также является номером прерывания. Перезаписав вектор с определённым индексом можно назначить прерыванию свой собственный обработчик.

В защищённом режиме обработчики прерываний (шлюзы, гейты или вентили) больше не определяются с помощью таблицы векторов. Вместо этой таблицы используется таблица вентилей или, правильнее, таблица прерываний - IDT (Interrupt Descriptors Table). Эта таблица формируется ядром, а её адрес хранится в регистре процессора idtr. Данный регистр недоступен напрямую. Работа с ним возможна только при помощи инструкций lidt/sidt. Первая из них (lidt) загружает в регистр idtr значение, указанное в операнде и являющееся базовым адресом таблицы дексрипторов прерываний, вторая (sidt) - сохраняет адрес таблицы, находящийся в idtr, в указанный операнд. Так же, как происходит выборка информации о сегменте из таблицы дескрипторов по селектору, происходит и выборка дескриптора сегмента, обслуживающего прерывание в защищённом режиме. Защита памяти поддерживается процессорами Intel начиная с CPU i80286 (не совсем в том виде, в каком она представлена сейчас, хотя бы потому, что 286 был 16-разрядным процессором - поэтому Linux не может работать на этих процессорах) и i80386, а посему процессор самостоятельно производит все необходимые выборки и, стало быть, сильно углубляться во все тонкости защищённого режима (а именно в защищённом режиме работает Linux) мы не будем. К сожалению, ни время, ни возможности не позволяют нам остановиться надолго на механизме обработки прерываний в защищённом режиме. Да это и не было целью при написании данной статьи. Все сведения, приводимые здесь касательно работы процессоров семейства х86 довольно поверхностны и приводятся лишь для того, чтобы помочь немного лучше понять механизм работы системных вызовов ядра. Кое-что можно узнать непосредственно из кода ядра, хотя, для полного понимания происходящего, желательно всё же ознакомиться с принципами работы защищённого режима. Участок кода, который заполняет начальными значениями (но не устанавливает!) IDT, находится в arch/i386/kernel/head.S:

/\*

\* setup\_idt

\*

\* sets up a idt with 256 entries pointing to

\* ignore\_int, interrupt gates. It doesn't actually load

\* idt - that can be done only after paging has been enabled

\* and the kernel moved to PAGE\_OFFSET. Interrupts

\* are enabled elsewhere, when we can be relatively

\* sure everything is ok.

\*

\* Warning: %esi is live across this function.

\*/

1.setup\_idt:

2. lea ignore\_int,%edx

3. movl $(\_\_KERNEL\_CS << 16),%eax

4. movw %dx,%ax /\* selector = 0x0010 = cs \*/

5. movw $0x8E00,%dx /\* interrupt gate - dpl=0, present \*/

6. lea idt\_table,%edi

7. mov $256,%ecx

8.rp\_sidt:

9. movl %eax,(%edi)

10. movl %edx,4(%edi)

11. addl $8,%edi

12. dec %ecx

13. jne rp\_sidt

14..macro set\_early\_handler handler,trapno

15. lea \handler,%edx

16. movl $(\_\_KERNEL\_CS << 16),%eax

17. movw %dx,%ax

18. movw $0x8E00,%dx /\* interrupt gate - dpl=0, present \*/

19. lea idt\_table,%edi

20. movl %eax,8\*\trapno(%edi)

21. movl %edx,8\*\trapno+4(%edi)

22..endm

23. set\_early\_handler handler=early\_divide\_err,trapno=0

24. set\_early\_handler handler=early\_illegal\_opcode,trapno=6

25. set\_early\_handler handler=early\_protection\_fault,trapno=13

26. set\_early\_handler handler=early\_page\_fault,trapno=14

28. ret

Несколько замечаний по коду: приведённый код написан на разновидности ассемблера AT&T, поэтому Ваше знание ассемблера в его привычной интеловской нотации может только сбить с толку. Самое основное отличие в порядке операндов. Если для интеловской нотации определён порядок - "аккумулятор" < "источник", то для ассемблера AT&T порядок прямой. Регистры процессора, как правило, должны иметь префикс "%", непосредственные значения (константы) префиксируются символом доллара "$". Синтаксис AT&T традиционно используется в Un\*x-системах.

В приведённом примере в строках 2-4 устанавливается адрес обработчика всех прерываний по умолчанию. Обработчиком по умолчанию является функция ignore\_int, которая ничего не делает. Наличие такой заглушки необходимо для корректной обработки всех прерываний на данном этапе, так как других ещё просто нет (правда, ловушки (traps) устанавливаются немного ниже по коду - о ловушках см. Intel Architecture Manual Reference или что-то подобное, здесь мы не будем касаться ловушек). В строке 5 устанавливается тип вентиля. В строке 6 мы загружаем в индексный регистр адрес нашей таблицы IDT. Таблица должна содержать 255 записей, по 8 байт каждая. В строках 8-13 мы заполняем всю таблицу одними и теми же значениями, установленными ранее в регистрах eax и edx - т.е., это вентиль прерывания, ссылающийся на обработчик ignore\_int. Чуть ниже мы определяем макрос для установки ловушек (traps) - строки 14-22. В строках 23-26 используя вышеопределённый макрос мы устанавливаем ловушки для следующих исключений: early\_divide\_err - деление на ноль (0), early\_illegal\_opcode - неизвестная инструкция процессора (6), early\_protection\_fault - сбой защиты памяти (13), early\_page\_fault - отказ страничной трансляции (14). В скобках приведены номера "прерываний", генерируемые при возникновении соответствующей нештатной ситуации. Перед проверкой типа процессора в arch/i386/kernel/head.S таблица IDT устанавливается вызовом setup\_idt:

/\*

\* start system 32-bit setup. We need to re-do some of the things done

\* in 16-bit mode for the "real" operations.

\*/

1. call setup\_idt

...

2. call check\_x87

3. lgdt early\_gdt\_descr

4. lidt idt\_descr

После выяснения типа (со)процессора и проведения всех подготовительных действий в строках 3 и 4 мы загружаем таблицы GDT и IDT, которые будут использоваться на самых первых порах работы ядра.

**Системные вызовы и int 0x80.**

От прерываний вернёмся обратно к системным вызовам. Итак, что необходимо, чтобы обслужить процесс, который запрашивает какую-то услугу? Для начала, необходимо перейти из кольца 3 (уровень привелегий CPL=3) на наиболее привелигированный уровень 0 (Ring 0, CPL=0), т.к. код ядра расположен в сегменте с наивысшими привелегиями. Кроме того, необходим код обработчика, который обслужит процесс. Именно для этого и используется шлюз 0x80. Хотя системных вызовов довольно много, для всех них используется единая точка входа - int 0x80. Сам обработчик устанавливается при вызове функции arch/i386/kernel/traps.c::trap\_init():

void \_\_init trap\_init(void)

{

...

set\_system\_gate(SYSCALL\_VECTOR,&system\_call);

...

}

Нас в trap\_init() больше всего интересует эта строка. В этом же файле выше можно посмотреть на код функции set\_system\_gate():

static void \_\_init set\_system\_gate(unsigned int n, void \*addr)

{

\_set\_gate(n, DESCTYPE\_TRAP | DESCTYPE\_DPL3, addr, \_\_KERNEL\_CS);

}

Здесь видно, что вентиль для прерывания 0х80 (а именно это значение определено макросом SYSCALL\_VECTOR - можете поверить наслово :) ) устанавливается как ловушка (trap) с уровнем привелегий DPL=3 (Ring 3), т.е. это прерывание будет отловлено при вызове из пространства пользователя. Проблема с переходом из Ring 3 в Ring 0 т.о. решена. Функция \_set\_gate() определена в заголовочном файле include/asm-i386/desc.h . Для особо любопытных ниже приведён код, без пространных объяснений, впрочем:

static inline void \_set\_gate(int gate, unsigned int type, void \*addr, unsigned short seg)

{

\_\_u32 a, b;

pack\_gate(&a, &b, (unsigned long)addr, seg, type, 0);

write\_idt\_entry(idt\_table, gate, a, b);

}

Вернёмся к функции trap\_init(). Она вызывается из функции start\_kernel() в init/main.c . Если посмотреть на код trap\_init(), то видно, что эта функция переписывает некоторые значения таблицы IDT заново - обработчики, которые использовались на ранних стадиях инициализации ядра (early\_page\_fault, early\_divide\_err, early\_illegal\_opcode, early\_protection\_fault), заменяются на те, что будут использоваться уже в процессе работы ядра. Итак, мы практически добрались до сути и уже знаем, что все системные вызовы обрабатываются единообразно - через шлюз int 0x80. В качестве обработчика для int 0x80, как опять же видно из приведённого выше куска кода arch/i386/kernel/traps.c::trap\_init(), устанавливается функция system\_call().

**system\_call().**

Код функции system\_call() находится в файле arch/i386/kernel/entry.S и выглядит следующим образом:

# system call handler stub

ENTRY(system\_call)

RING0\_INT\_FRAME # can't unwind into user space anyway

pushl %eax # save orig\_eax

CFI\_ADJUST\_CFA\_OFFSET 4

SAVE\_ALL

GET\_THREAD\_INFO(%ebp)

# system call tracing in operation / emulation

/\* Note, \_TIF\_SECCOMP is bit number 8, and so it needs testw and not testb \*/

testw $(\_TIF\_SYSCALL\_EMU|\_TIF\_SYSCALL\_TRACE|\_TIF\_SECCOMP|\_TIF\_SYSCALL\_AUDIT),TI\_flags(%ebp)

jnz syscall\_trace\_entry

cmpl $(nr\_syscalls), %eax

jae syscall\_badsys

syscall\_call:

call \*sys\_call\_table(,%eax,4)

movl %eax,PT\_EAX(%esp) # store the return value

...

Код приведён не полностью. Как видно, сперва system\_call() настраивает стек для работы в Ring 0, сохраняет значение, переданное ей через eax в стек, сохраняет все регистры также в стек, получает данные о вызывающем потоке и проверяет, не выходит ли переданное значение-номер системного вызова за пределы таблицы системных вызовов и затем, наконец, пользуясь значением, переданным в eax в качестве аргумента, system\_call() осуществляет переход на настоящий обработчик системного вывода, исходя из того, на какой элемент таблицы ссылается индекс в eax. Теперь вспомните старую добрую таблицу векторов прерываний из реального режима. Ничего не напоминает? В реальности, конечно, всё несколько сложнее. В частности, системный вызов должен скопировать результаты из стека ядра в стек пользователя, передать код возврата и ещё некоторые вещи. В том случае, когда аргумент, указанный в eax не ссылается на существующий системный вызов (значение выходит за диапазон), происходит переход на метку syscall\_badsys. Здесь в стек по смещению, по которому должно находиться значение eax, заносится значение -ENOSYS - системный вызов не реализован. На этом выполнение system\_call() завершается.

Таблица системных вызовов находится в файле arch/i386/kernel/syscall\_table.S и имеет достаточно простой вид:

ENTRY(sys\_call\_table)

.long sys\_restart\_syscall /\* 0 - old "setup()" system call, used for restarting \*/

.long sys\_exit

.long sys\_fork

.long sys\_read

.long sys\_write

.long sys\_open /\* 5 \*/

.long sys\_close

.long sys\_waitpid

.long sys\_creat

...

Иными словами, вся таблица являет собой ничто иное, как массив адресов функций, расположенных в порядке следования номеров системных вызовов, которые эти функции обслуживают. Таблица - обычный массив двойных машинных слов (или 32-разрядных слов - кому как больше нравится). Код части функций, обслуживающих системные вызовы, находится в платформно-зависимой части - arch/i386/kernel/sys\_i386.c, а часть, не зависящая от платформы - в kernel/sys.c .

Вот так обстоит дело с системными вызовами и вентилем 0x80.

**Новый механизм обработки системных вызовов в Linux. sysenter/sysexit.**

Как упоминалось, достаточно быстро выяснилось, что использование традиционного способа обработки системных вызовов на основе гейта 0х80 пиводит к потере производительности на процессорах Intel Pentium 4. Поэтому Линус Торвальдс реализовал в ядре новый механизм, основанный на инструкциях sysenter/sysexit и призванный повысить производительность ядра на машинах, оснащённых процессором Pentium II и выше (именно с Pentium II+ процессоры Intel поддерживают упомянутые инструкции sysenter/sysexit). В чём суть нового механизма? Как ни странно, но суть осталась та же. Изменилось исполнение. Согласно документации Intel инструкция sysenter является частью механизма "быстрых системных вызовов". В частности, эта инструкция оптимизирована для быстрого перехода с одного уровня привелегий на другой. Если точнее, то она ускоряет переход в кольцо 0 (Ring 0, CPL=0). При этом, операционная система должна подготовить процессор к использовании инструкции sysenter. Такая настройка осуществляется единожды при загрузке и инициализации ядра ОС. При вызове sysenter устанавливает регистры процессора согласно машинно-зависимых регистров, ранее установленных ОС. В частности, устанавливаются сегментный регистр и регистр указателя инструкций - cs:eip, а также сегмент стека и указатель вершины стека - ss, esp. Переход на новый сегмент кода и смещение осуществляется из кольца 3 в 0.

Инструкция sysexit выполняют обратные действия. Она производит быстрый переход с уровня привелегий 0 на 3-й (CPL=3). При этом регистр сегмента кода устанавливается в 16 + значение сегмента cs, сохранённое в машинно-зависимом регистре процессора. В регистр eip заносится содержимое регистра edx. В ss заносится сумма 24 и значения cs, занесённое ОС ранее в машинно-зависимый регистр процессора при подготовке контекста для работы инструкции sysenter. В esp заносится содержимое регистра ecx. Значения, необходимые для работы инструкций sysenter/sysexit хранятся по следующим адресам:

1. SYSENTER\_CS\_MSR    0х174 - сегмент кода, куда заносится значение сегмента, в котором находится код обработчика системного вызова.
2. SYSENTER\_ESP\_MSR   0х175 - указатель на вершину стека для обработчика системного вызова.
3. SYSENTER\_EIP\_MSR    0х176 - указатель на смещение внутри сегмента кода. Указывает на начало кода обработчика системных вызовов.

Данные адреса ссылаются на модельно-зависимые регистры, которые не имеют имён. Значения записываются в модельно зависимые регистры с помощью инструкции wrmsr, при этом edx:eax должны содержать страшую и младшую части 64-битного машинного слова соответственно, а в ecx должен быть занесён адрес ригистра, в который будет произведена запись. В Linux адреса модельно-зависимых регистров определяются в заголовочном файле include/asm-i368/msr-index.h следующим образом (до версии 2.6.22 как минимум они определялись в заголовочном файле include/asm-i386/msr.h, напомню, что мы рассматриваем механизм системных вызовов на примере ядра Linux 2.6.22):

#define MSR\_IA32\_SYSENTER\_CS 0x00000174

#define MSR\_IA32\_SYSENTER\_ESP 0x00000175

#define MSR\_IA32\_SYSENTER\_EIP 0x00000176

Код ядра, ответственный за установку модельно-зависимых регистров, находится в файле arch/i386/sysenter.c и выглядит следующим образом:

1. void enable\_sep\_cpu(void)

{

2. int cpu = get\_cpu();

3. struct tss\_struct \*tss = &per\_cpu(init\_tss, cpu);

4. if (!boot\_cpu\_has(X86\_FEATURE\_SEP)) {

5. put\_cpu();

6. return;

}

7. tss->x86\_tss.ss1 = \_\_KERNEL\_CS;

8. tss->x86\_tss.esp1 = sizeof(struct tss\_struct) + (unsigned long) tss;

9. wrmsr(MSR\_IA32\_SYSENTER\_CS, \_\_KERNEL\_CS, 0);

10. wrmsr(MSR\_IA32\_SYSENTER\_ESP, tss->x86\_tss.esp1, 0);

11. wrmsr(MSR\_IA32\_SYSENTER\_EIP, (unsigned long) sysenter\_entry, 0);

12. put\_cpu();

}

Здесь в переменную tss мы получаем адрес структуры, описывающей сегмент состояния задачи. TSS (Task State Segment) используется для описания контекста задачи и является частью механизма аппаратной поддержки многозадачности для архитектуры x86. Однако, Linux практически не использует аппаратное переключение контекста задач. Согласно документации Intel переключение на другую задачу производится либо путём выполнения инструкции межсегментного перехода (jmp или call), ссылающейся на сегмент TSS, либо на дескриптор вентиля задачи в GDT (LDT). Специальный регистр процессора, невидимый для программиста - TR (Task Register - регистр задачи) содержит селектор дескриптора задачи. При загрузке этого регистра также загружаются программно-невидимые регистры базы и лимита, связанные с TR.

Несмотря на то, что Linux не использует аппаратное переключение контекстов задач, ядро вынуждено отводить запись TSS для каждого процессора, установленного в системе. Это связано с тем, что когда процессор переключается из пользовательского режима в режим ядра, он извлекает из TSS адрес стека ядра. Кроме того, TSS необходим для управления доступом к портам ввода/вывода. TSS содержит карту прав доступа к портам. На основе этой карты становится возможным осуществлять контроль доступа к портам для каждого процесса, использующего инструкции in/out. Здесь tss->x86\_tss.esp1 указывает на стек ядра. \_\_KERNEL\_CS естественно указывает на сегмент кода ядра. В качестве смещения-eip указывается адрес функции sysenter\_entry().

Функция sysenter\_entry() определена в файле arch/i386/kernel/entry.S и имеет такой вид:

/\* SYSENTER\_RETURN points to after the "sysenter" instruction in

the vsyscall page. See vsyscall-sysentry.S, which defines the symbol. \*/

# sysenter call handler stub

ENTRY(sysenter\_entry)

CFI\_STARTPROC simple

CFI\_SIGNAL\_FRAME

CFI\_DEF\_CFA esp, 0

CFI\_REGISTER esp, ebp

movl TSS\_sysenter\_esp0(%esp),%esp

sysenter\_past\_esp:

/\*

\* No need to follow this irqs on/off section: the syscall

\* disabled irqs and here we enable it straight after entry:

\*/

ENABLE\_INTERRUPTS(CLBR\_NONE)

pushl $(\_\_USER\_DS)

CFI\_ADJUST\_CFA\_OFFSET 4

/\*CFI\_REL\_OFFSET ss, 0\*/

pushl %ebp

CFI\_ADJUST\_CFA\_OFFSET 4

CFI\_REL\_OFFSET esp, 0

pushfl

CFI\_ADJUST\_CFA\_OFFSET 4

pushl $(\_\_USER\_CS)

CFI\_ADJUST\_CFA\_OFFSET 4

/\*CFI\_REL\_OFFSET cs, 0\*/

/\*

\* Push current\_thread\_info()->sysenter\_return to the stack.

\* A tiny bit of offset fixup is necessary - 4\*4 means the 4 words

\* pushed above; +8 corresponds to copy\_thread's esp0 setting.

\*/

pushl (TI\_sysenter\_return-THREAD\_SIZE+8+4\*4)(%esp)

CFI\_ADJUST\_CFA\_OFFSET 4

CFI\_REL\_OFFSET eip, 0

/\*

\* Load the potential sixth argument from user stack.

\* Careful about security.

\*/

cmpl $\_\_PAGE\_OFFSET-3,%ebp

jae syscall\_fault

1: movl (%ebp),%ebp

.section \_\_ex\_table,"a"

.align 4

.long 1b,syscall\_fault

.previous

pushl %eax

CFI\_ADJUST\_CFA\_OFFSET 4

SAVE\_ALL

GET\_THREAD\_INFO(%ebp)

/\* Note, \_TIF\_SECCOMP is bit number 8, and so it needs testw and not testb \*/

testw $(\_TIF\_SYSCALL\_EMU|\_TIF\_SYSCALL\_TRACE|\_TIF\_SECCOMP|\_TIF\_SYSCALL\_AUDIT),TI\_flags(%ebp)

jnz syscall\_trace\_entry

cmpl $(nr\_syscalls), %eax

jae syscall\_badsys

call \*sys\_call\_table(,%eax,4)

movl %eax,PT\_EAX(%esp)

DISABLE\_INTERRUPTS(CLBR\_ANY)

TRACE\_IRQS\_OFF

movl TI\_flags(%ebp), %ecx

testw $\_TIF\_ALLWORK\_MASK, %cx

jne syscall\_exit\_work

/\* if something modifies registers it must also disable sysexit \*/

movl PT\_EIP(%esp), %edx

movl PT\_OLDESP(%esp), %ecx

xorl %ebp,%ebp

TRACE\_IRQS\_ON

1: mov PT\_FS(%esp), %fs

ENABLE\_INTERRUPTS\_SYSEXIT

CFI\_ENDPROC

.pushsection .fixup,"ax"

2: movl $0,PT\_FS(%esp)

jmp 1b

.section \_\_ex\_table,"a"

.align 4

.long 1b,2b

.popsection

ENDPROC(sysenter\_entry)

Как и в случае с system\_call() основная работа выполняется в строке call \*sys\_call\_table(,%eax,4). Здесь вызывается конкретный обработчик системного вызова. Итак, видно, что принципиально изменилось мало. То обстоятельство, что вектор прерывания теперь забит в железо и процессор помогает нам быстрее перейти с одного уровня привелегий на другой меняет лишь некоторые детали исполнения при прежнем содержании. Правда, на этом изменения не заканчиваются. Вспомните, с чего начиналось повествование. В самом начале я упоминал уже о виртуальных разделяемых объектах. Так вот, если раньше реализация системного вызова, скажем, из системной библиотеки libc выглядела, как вызов прерывания (при том, что библиотека брала некоторые функции на себя, чтобы сократить число переключений контекстов), то теперь благодаря VDSO системный вызов может быть сделан практически напрямую, без участия libc. Он и ранее мог быть осуществлён напрямую, опять же, как прерывание. Но теперь вызов можно затребовать, как обычную функцию, экспортируемую из динамически компонуемой библиотеки (DSO). При загрузке ядро определяет, какой механизм должен и может быть использован для данной платформы. В зависимости от обстоятельств ядро устанавливает точку входа в функцию, выполняющую системный вызов. Далее, функция экспортируется в пользовательское пространство ввиде библиотеки linux-gate.so.1 . Библиотека linux-gate.so.1 физически не существует на диске. Она, если можно так выразиться, эмулируется ядром и существует ровно столько, сколько работает система. Если выполнить останов системы, подмонтировать корневую ФС из другой системы, то Вы не найдёте на корневой ФС остановленной системы этот файл. Собственно, Вы не сможете его найти даже на работающей системе. Физически его просто нет. Именно поэтому linux-gate.so.1 - это нечто иное, как VDSO - т.е. Virtual Dynamically Shared Object. Ядро отображает эмулируемую таким образом динамическую библиотеку в адресное пространство каждого процесса. Убедиться в этом несложно, если выполнить следующую команду:

f0x@devel0:~$ cat /proc/self/maps

08048000-0804c000 r-xp 00000000 08:01 46 /bin/cat

0804c000-0804d000 rw-p 00003000 08:01 46 /bin/cat

0804d000-0806e000 rw-p 0804d000 00:00 0 [heap]

...

b7fdf000-b7fe1000 rw-p 00019000 08:01 2066 /lib/ld-2.5.so

bffd2000-bffe8000 rw-p bffd2000 00:00 0 [stack]

ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]

Здесь самая последняя строка и есть интересующий нас объект:

ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]

Из приведённого примера видно, что объект занимает в памяти ровно одну страницу - 4096 байт, практически на задворках адресного пространства. Проведём ещё один эксперимент:

f0x@devel0:~$ ldd `which cat`

linux-gate.so.1 => (0xffffe000)

libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e87000)

/lib/ld-linux.so.2 (0xb7fdf000)

f0x@devel0:~$ ldd `which gcc`

linux-gate.so.1 => (0xffffe000)

libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e3c000)

/lib/ld-linux.so.2 (0xb7f94000)

f0x@devel0:~$

Здесь мы просто навскидку взяли два приложения. Видно, что библиотека отображается в адресное пространство процесса по одному и тому же постоянному адресу - 0xffffe000. Теперь попробуем посмотреть, что же такое хранится на этой странице памяти на самом деле...

Сделать дамп страницы памяти, где хранится разделяемый код VDSO, можно с помощью следующей программы:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main () {

char\* vdso = 0xffffe000;

char\* buffer;

FILE\* f;

buffer = malloc (4096);

if (!buffer)

exit (1);

memcpy (buffer, vdso, 4096);

if (!(f = fopen ("test.dump", "w+b"))) {

free (buffer);

exit (1);

}

fwrite (buffer, 4096, 1, f);

fclose (f);

free (buffer);

return 0;

}

Строго говоря, раньше это можно было сделать проще, с помощью команды **dd if=/proc/self/mem of=test.dump bs=4096 skip=1048574 count=1**, но ядра начиная с версии 2.6.22 или, быть может, даже более ранней, больше не отображают память процесса в файл /proc/`pid`/mem. Этот файл, сохранён, очевидно, для совместимости, но не содержит более информации.

Скомпилируем и прогоним приведённую программу. Попробуем дизассемблировать полученный код:

f0x@devel0:~/tmp$ objdump --disassemble ./test.dump

./test.dump: file format elf32-i386

Disassembly of section .text:

ffffe400 <\_\_kernel\_vsyscall>:

ffffe400: 51 push %ecx

ffffe401: 52 push %edx

ffffe402: 55 push %ebp

ffffe403: 89 e5 mov %esp,%ebp

ffffe405: 0f 34 sysenter

...

ffffe40e: eb f3 jmp ffffe403 <\_\_kernel\_vsyscall+0x3>

ffffe410: 5d pop %ebp

ffffe411: 5a pop %edx

ffffe412: 59 pop %ecx

ffffe413: c3 ret

...

f0x@devel0:~/tmp$

Вот он наш шлюз для системных вызовов, весь, как на ладони. Процесс (либо, системная библиотека libc), вызывая функцию \_\_kernel\_vsyscall попадает на адрес 0хffffe400 (в нашем случае). Далее, \_\_kernel\_vsyscall сохраняет в стеке пользовательского процесса содержимое регистров ecx, edx, ebp, О назначении регистров ecx и edx мы уже говорили ранее, в ebp используется позже для восстановления стека пользователя. Выполняется инструкция sysenter, "перехват прерывания" и, как следствие, очередной переход на sysenter\_entry (см. выше). Инструкция jmp по адресу 0xffffe40e вставлена для перезапуска системного вызова с числом 6 аргументами (см. <http://lkml.org/lkml/2002/12/18/>). Код, размещаемый на странице, находится в файле arch/i386/kernel/vsyscall-enter.S (или arch/i386/kernel/vsyscall-int80.S для ловушки 0x80). Хотя я и нашёл, что адрес функции \_\_kernel\_vsyscall постоянный, но есть мнение, что это не так. Обычно, положение точки входа в \_\_kernel\_vsyscall() можно найти по вектору ELF-auxv используя параметр AT\_SYSINFO. Вектор ELF-auxv содержит информацию, передаваемую процессу через стек при запуске и содержит различную информацию, нужную в процессе работы программы. Этот вектор в частности содержит переменные окружения процесса, аргументы, и проч..

Вот небольшой пример на С, как можно обратиться к функции \_\_kernel\_vsyscall напрямую:

#include <stdio.h>

int pid;

int main ()

{

\_\_asm (

"movl $20, %eax \n"

"call \*%gs:0x10 \n"

"movl %eax, pid \n"

);

printf ("pid: %d\n", pid);

return 0;

}

Данный пример взят со страницы Manu Garg, [http://www.manugarg.com](http://www.manugarg.com/). Итак, в приведённом примере мы делаем системный вызов getpid() (номер 20 или иначе \_\_NR\_getpid). Чтобы не лазить по стеку процесса в поисках переменной AT\_SYSINFO воспользуемся тем обстоятельством, что системная библиотека libc.so при загрузке копирует значение переменной AT\_SYSINFO в блок управления потоком (TCB - Thread Control Block). На этот блок информации, как правило, ссылается селектор в gs. Предполагаем, что по смещению 0х10 находится искомый параметр и делаем вызов по адресу, хранящемуся в %gs:$0x10.

**Итоги.**

На самом деле, практически, особого прироста производительности даже при поддержке на данной платформе FSCF (Fast System Call Facility) добиться не всегда возможно. Проблема в том, что так или иначе, процесс редко обращается напрямую к ядру. И для этого есть свои веские причины. Использование библиотеки libc позволяет гарантировать переносимость программы вне зависимости от версии ядра. И именно через стандартную системную библиотеку идёт большинство системных вызовов. Если даже Вы соберёте и установите самое последнее ядро, собранное для платформы, поддерживающей FSCF, это ещё не гарантия прироста производительности. Дело в том, что Ваша системная библиотека libc.so будет попрежнему использовать int 0x80 и справиться с этим можно лишь пересобрав glibc. Поддерживается ли в glibc вообще интерфейс VDSO и \_\_kernel\_vsyscall, я, честно признаться, на данный момент ответить затрудняюсь.

**Ссылки.**

[1] Manu Garg's page, [http://www.manugarg.com](http://www.manugarg.com/)  
[2] Scatter/Gather thoughts by Johan Petersson, [http://www.trilithium.com/johan/2005/08/linux-gate/](http://www.trithium.com/johan/2005/08/linux-gate/)  
[3] Старый добрый Understanding the Linux kernel Куда же без него :)  
[4] Ну и конечно же, исходные коды Linux (2.6.22)

2008-03-18. Further revisions are possible...

Posted by [red\_f0x](https://www.blogger.com/profile/02367659412016964697" \o "author profile)at [11:19 AM](http://rflinux.blogspot.com/2008/03/linux-syscalls-linux.html" \o "permanent link)

Labels: [kernel](http://rflinux.blogspot.com/search/label/kernel), [linux](http://rflinux.blogspot.com/search/label/linux), [system calls](http://rflinux.blogspot.com/search/label/system%20calls)