**Проектирование ОС и компонентов**

Системные вызовы

**Теоретическая часть**

Системный вызов в программировании и вычислительной технике — обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции. Современные операционные системы (ОС) предусматривают разделение времени между выполняющимися вычислительными процессами (многозадачность) и разделение полномочий, препятствующее обращению исполняемых программ к данным других программ и оборудованию. Ядро ОС исполняется в привилегированном режиме работы процессора. Для выполнения межпроцессной операции или операции, требующей доступа к оборудованию, программа обращается к ядру, которое, в зависимости от полномочий вызывающего процесса, исполняет либо отказывает в исполнении такого вызова. С точки зрения программиста, системный вызов обычно выглядит как вызов подпрограммы или функции из системной библиотеки. Однако системный вызов, как частный случай вызова такой функции или подпрограммы, следует отличать от более общего обращения к системной библиотеке, поскольку последнее может и не требовать выполнения привилегированных операций.

Режим ядра

Когда процессор находится в режиме ядра, код может быть выполнен с любым адресом памяти и любым аппаратным ресурсом. Следовательно, режим ядра - очень привилегированный и мощный режим. Если программа выйдет из строя в режиме ядра, вся система будет остановлена.

Пользовательский режим

Когда CPU находится в пользовательском режиме, программы не имеют прямого доступа к памяти и аппаратным ресурсам. В режиме пользователя, если какая-либо программа выходит из строя, останавливается только эта конкретная программа. Это означает, что система будет находиться в безопасном состоянии, даже если программа в пользовательском режиме сработает.

Следовательно, большинство программ в ОС запускаются в пользовательском режиме.

**Системные вызовы Syslog, Ptrace, Uname, Sysinfo**

**Syslog**

Краткий обзор

|  |
| --- |
| int syslog(int type, char \*bufp, int len);  /\* No wrapper provided in glibc \*/  /\* The glibc interface \*/  #include <sys/klog.h>  int klogctl(int type, char \*bufp, int len); |

Ядро имеет циклический буфер длиной LOG\_BUF\_LEN, в котором сообщения

       Заданные в качестве аргументов функции ядра printk (), сохраняются (

       Меньше их loglevel). В ранних ядрах значение LOG\_BUF\_LEN имело значение

       4096; Из ядра 1.3.54 это было 8192; Из ядра 2.1.113 это было

       16384; Поскольку 2.4.23 / 2.6 это параметр конфигурации ядра. В

       Последние ядра размер может быть запрошен с типом команды 10.

* Вызов syslog ожидает, пока этот буфер журнала ядра не будет

       Непустым, а затем считывает не более len байтов в буфер buf. Это

       Возвращает количество прочитанных байтов. Байты, считанные из журнала, исчезают

       Из буфера журнала: информация может быть прочитана только один раз. Это

       Функция, выполняемая ядром при чтении пользовательской программы

       / Proc / kmsg.

* В вызове syslog) будут прочитаны последние len байты из журнала

       Буфера (неразрушающе), но не будет читать больше, чем было записано в

       Буфер с последней команды «clear ring buffer» (которая не

       Полностью очистить буфер). Он возвращает количество прочитанных байтов.

* Вызов syslog делает то же самое, но также выполняет

       Команда «clear ring buffer».

* Вызов syslog выполняет только «очищающий буфер звонка»,

       команда. (В каждом вызове, где buf или len отображается как «фиктивный», значение

       Аргумента игнорируется вызовом.)

* Вызов syslog устанавливает минимальный уровень журнала консоли,

       Так что на консоль не печатаются сообщения.

* Syslog вызова) устанавливает уровень журнала консоли по умолчанию,

       Так что сообщения печатаются на консоль.

* Вызов syslog) устанавливает уровень журнала консоли на уровень, который должен быть целым числом от 1 до 8 (включительно).

       Для подробностей.

* В вызове syslog возвращается количество байтов в настоящее время. Доступный для чтения в буфере журнала ядра.
* Вызов syslog возвращает общий размер ядра. Буфер журнала.

ОШИБКИ

        EINVAL Плохие аргументы (например, плохой тип, или для типа 2, 3 или 4, buf -

NULL или len меньше нуля; Или для типа 8,

На стороне от 1 до 8).

        EPERM Была сделана попытка изменить console\_loglevel или очистить кера-

Буфера кольцевого буфера сообщений процессом без достаточных

Lege (точнее: без возможности CAP\_SYS\_ADMIN).

ERESTARTSYS

Сигнал системы был прерван сигналом; Ничего не было прочитано.

(Это можно увидеть только во время трассировки).

        ENOSYS Этот системный вызов syslog () недоступен, поскольку ядро

Был скомпилирован с параметром конфигурации ядра CONFIG\_PRINTK

отключен.

Исходный код:

|  |
| --- |
| /\*\* Test program for the syslog() system call.  \* From the syslog(2) man page:  \* If you need the libc function syslog() (which talks to syslogd(8)),  \* then look at syslog(3). The system call of this name is about control-  \* ling the kernel printk() buffer, and the glibc version is called  \* klogctl().  \*/  #include "../../config.h"  #include <stdio.h>  #if defined(HAVE\_SYS\_KLOG\_H)  #include <sys/klog.h>  #endif  int main(int argc, char\*\* argv)  {  int number\_of\_unread\_characters;  #if defined HAVE\_KLOGCTL  number\_of\_unread\_characters = klogctl(9, 0, 0);  #endif  fprintf(stderr, "Done.\n");  return 0 \* number\_of\_unread\_characters;  } |

**Ptrace**

Системный вызов ptrace () предоставляет средство, с помощью которого один процесс (

«Tracer») может наблюдать и контролировать выполнение другого процесса

(«Трассировка»), а также изучить и изменить память трассировки и

регистры. Он в основном используется для реализации отладки точки останова

И отслеживание системных вызовов.

Сначала трассировка должна быть привязана к индикатору. Приложение и

Последующие команды для потока: в многопоточном процессе каждый

Нить может быть индивидуально прикреплена к (потенциально другому)

Трассировщик, или левый, не прикрепленный и, следовательно, не отлаженный. Следовательно,

«Trace» всегда означает «(один) поток«, никогда »(возможно

Многопоточность). Команды Ptrace всегда отправляются на

Конкретный трассировка с использованием вызова формы.   
Ptrace (PTRACE\_foo, pid, ...)

Pid - это идентификатор потока соответствующего потока Linux.

ОШИБКИ

EBUSY (только i386) - Произошла ошибка при распределении или освобождении регистр отладки.

EFAULT - Была попытка прочитать или записать недопустимую область в памяти трассировщика или трассировки, вероятно, потому, что область не была отображена или недоступна. К сожалению, под Linux, различные варианты этой ошибки возвратят EIO или

EFAULT - Более или менее произвольно.

EINVAL - Была сделана попытка установить недопустимый вариант. Запрос EIO недействителен, или была сделана попытка прочитать или в недопустимую область в трассировке или в трассировке памяти, или произошло нарушение выравнивания по словам или недействительное. Сигнал был указан во время запроса перезапуска.

EPERM - Указанный процесс не может быть прослежен. Это может быть связано с тем, что у индикатора недостаточно привилегий (требуется возможность - CAP\_SYS\_PTRACE); Непривилегированные процессы не могут отслеживать процессы, которые они не могут отправлять запущенные программы set-user-ID / set-group-ID, для очевидных причины. Альтернативно, процесс может уже прослеживается или (на ядрах до 2.6.26) - init (1) (PID 1).

ESRCH - Указанный процесс не существует или в настоящее время отсутствует отслеживается вызывающим абонентом или не останавливается (для запросов которые требуют остановленного трассировки).

**Uname**

Функция uname () должна хранить информацию, идентифицирующую текущую систему, в структуре, на которую указывает имя.

Функция uname () использует структуру utsname, определенную в <sys / utsname.h>.

Функция uname () возвращает строку, набирая текущую систему в массиве sysname. Аналогично, имя узла должно содержать имя этого узла в сети связи, определенной реализацией. Выпуск и версия массивов должны дополнительно идентифицировать операционную систему. Машина массива должна содержать имя, которое идентифицирует аппаратное обеспечение, на котором работает система.

Формат каждого участника определяется реализацией.

В ядре 2.6.32 функция sys\_uname реализована в 6 файлах. Эта функция переписывает пользователю структуру utsname, если пользователь передал некорректную входную структуру, то возвращается ошибка -EFAULT.

В ядре 2.6.32 функция sys\_uname реализована в файлах:

arch/m32r/kernel/sys\_m32r.c (функция, строки 160-169)

arch/powerpc/kernel/syscalls.c (функция, строки 242-253)

arch/sh/kernel/sys\_sh.c (функция, строки 201-210)

arch/um/kernel/syscall.c (функция, строки 54-63)

arch/x86/kernel/sys\_i386\_32.c (функция, строки 166-175)

arch/um/sys-x86\_64/syscalls.c (функция, строки 15-27)

Хорошо, что в ядре 4.4.1 разработчики навели порядок и uname теперь реализован только в файле /kernel/sys.c (строки 1161-1178), правда там тоже есть насколько реализаций, но они связаны с различиями в структуре utsname.

1168: Блокировка для чтения структуры utsname

1169: Копируем содержимое структуры utsname в переменную пользователя

1171: Освобождаем семафор

1173-1178: Проверка на ошибки и возвращение результата (0 или -EFAULT)

|  |
| --- |
| #include <sys/utsname.h>  #include <errno.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  struct utsname unameData;  int ret = 0;  // Might check return value here (non-0 = failure)  ret = uname(&unameData);  if (ret != 0) {  if (errno == EFAULT) {  printf("EFAULT - Not valid value for buf.\n");  }  else {  printf("Error with uname.\n");  }  return -1;  }  //After this, the struct will contain the info you want:  printf("sysname: %s\n", unameData.sysname);  printf("nodename: %s\n", unameData.nodename);  printf("release: %s\n", unameData.release);  printf("version: %s\n", unameData.version);  printf("machine: %s\n", unameData.machine);  #ifdef \_GNU\_SOURCE  printf("domainname: %s\n", unameData.domainname);  #endif  return 0;  } |

# Перехват системных вызовов

**LSM**

До ядра версии 2.6, для того, чтобы перехватить системный вызов, писали функцию-хук, которая её заменяла: выполняла другой код и вызывала непосредственно сам syscall (чтобы не нарушить работоспособность системы).

Так как каждый системный вызов подобно функции имеет свой адрес, а в Linux есть специальная таблица где эти адреса хранятся, задача сводилась к тому, чтобы в этой самой таблице заменить адрес системного вызова на адрес нашей функции.

Позже, разработчики Linux пытались устранить возможность такого способа, но до сих пор существуют способы, которые позволяют реализовать этот метод.

LSM — это фреймворк для разработки модулей безопасности ядра.  LSM является частью ядра начиная с Linux версии 2.6. Он был создан для того, чтобы расширить стандартную модель безопасности DAC, сделать её более гибкой. Этот фреймворк использует известный модуль безопасности SELinux, а также ещё несколько других, встроенных в ядро.  
Самое ценное для нас в данном фреймворке то, что он реализован через набор заранее предустановленных в ядро хуков (безопасный способ, потому что ядро заранее рассчитано на наличие таких хуков).

LSM позволяет вставлять в код своих хуков вызов пользовательских, что позволяет безопасно работать с системными вызовами без изменения таблицы символов.

Прежде чем вызвать ваш LSM обработчик сначала производится стандартная проверка прав доступа. Так, если у пользователя не хватает прав на создание директории, то системный вызов сразу вернёт код ошибки пользовательскому процессу. Ваш обработчик при этом не вызовется. Не для всех системных вызовов есть соответствующий LSM hook. Для системных вызовов uname, нет хуков в рассматриваемых версиях ядра.

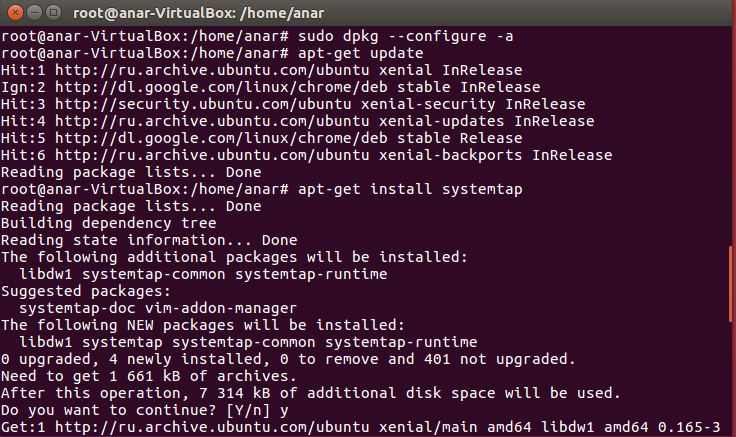
LSM предоставляет возможности реализации дополнительных проверок прав доступа поверх стандартных. Более того, c некоторых пор LSM стал стековым интерфейсом, то есть приобрёл возможность подключать сразу несколько модулей безопасности (например SELinux, Yama)

**SystemTap**

SystemTap — это средство, которое позволяет собирать и анализировать информацию о работающей Linux системе. SystemTap представляет из себя интерфейс командной строки и скриптовый язык программирования. Разработчики могут использовать SystemTap для мониторинга и анализа производительности системы, а разработчики программного обеспечения могут использовать SystemTap для анализа поведения приложения в работающей системе.

Основная идея SystemTap состоит в том, чтобы обозначить события и назначить для них обработчики. Во время выполнения скрипта, SystemTap занимается мониторингом событий и, как только произойдет событие, ядро системы выполнит обработчик. Событиями могут быть начало или конец сессии SystemTap, срабатывание таймера и другие.

Получить пакет systemtap:



Общая конфигурация репозитория deb:

|  |
| --- |
| # cat > /etc/apt/sources.list.d/ddebs.list << EOF  deb http://ddebs.ubuntu.com/ precise main restricted universe multiverse  EOF  etc.  # apt-key adv --keyserver keyserver.ubuntu.com --recv-keys ECDCAD72428D7C01  # apt-get update  ………………………………………………………………………………………………………………….............  % uname -r  4.4.0-31-generic  % dpkg --list | grep linux  linux-image-4.4.0-31-generic\_3.5.0-26.41\_amd64.deb  linux-image-4.4.0-31-generic-dbgsym\_3.5.0-26.42\_amd64.ddeb |

Обработчиком является последовательность скриптовых операторов, которые будут выполнены после срабатывания события. Обычно обработчики извлекают информацию из контекста события или выводят информацию на экран. Сессия SystemTap начинается тогда, когда мы выполняем скрипт. В это время происходит следующая последовательность действий:

1. Сначала SystemTap проверяет библиотеку «тапсетов» на наличие использованных в скрипте;

2. Потом SystemTap транслирует скрипт в Си (язык программирования) и запускает системный компилятор, чтобы создать модуль ядра из скрипта;

3. SystemTap загружает модуль и активирует все события в скрипте;

4. Как только происходит событие выполняется обработчик данного события;

5. Когда все события выполнены, модуль выгружается и сессия завершается;

В SystemTap есть 2 типа событий: синхронные и асинхронные. Синхронные события привязаны к инструкции в определенном месте в коде ядра. Примеры синхронных событий:

— syscall.system\_call

— vfs.file\_operation

— kernel.function(«function»)

— module(«module»).function(«function»)

Асинхронные события не привязаны к определенной инструкции или определенному месту в коде ядра. Примеры асинхронных событий:

— begin — начало сессии SystemTap

— end — конец сессии SystemTap

— timer.event() — отсчет таймера (timer.s(4) — событие будет срабатывать каждые 4 секунды)

Рассмотрим некоторые примечательные свойства языка SystemTap:

* Как уже отмечалось, код транслируется в С, компилируется в модуль ядра и загружается;
* Точки с запятой в конце выражения *не* являются обязательными, и вообще на самом деле означают пустое выражение;
* Можно использовать локальные переменные, просто присвоив чему-нибудь значение;
* Вывод типов (строковый, числовой) производится автоматически на этапе компиляции;
* Есть тип «агрегаты» (aggregates), запись в которые производятся специальным оператором <<<, и для доступа к которым есть функции @min, @max, @avg, @count, @avg, а также функции для вывода — @hist\_log и @hist\_linear. Агрегаты не требуют тяжелых блокировок, так как данные хранятся отдельно на каждом ядре процессора, и собираются воедино только при необходимости;
* SystemTap поддерживает ассоциативные массивы, реализованные, как хэш-таблицы, чей максимальный размер фиксирован и задается при старте скрипта. Ассоциативные массивы могут быть только глобальными;
* Поддерживаются управляющие конструкции if-then-else, while, for, foreach, break, continue;
* Можно объявлять свои функции, при этом аргументы передаются по значению;
* Можно писать библиотеки (так называемые tapsets), складывая свои .stp файлы в /usr/share/systemtap/tapset, или используя аргумент -I path;
* Глобальные переменные, используемые в пробе, автоматически блокируются должным образом при входе в проб;
* Вся память выделяется один раз при инициализации скрипта и потому утечки памяти невозможны;
* Если обработчик выполняется слишком долго или уходит слишком глубоко в рекурсию, это обнаруживается и выполнение скрипта прекращается;
* Можно делить на ноль и разыменовывать null — ваш скрипт от этого грохнется, но ядру ничего не будет;
* Продвинутые пользователи могут использовать в скриптах код на Си;

Таким образом, SystemTap еще более упрощает задачу перехвата системных вызовов и обеспечивает большую безопасность. Установить SystemTap можно из официальных репозиториев Ubuntu. Так же для его работоспособности необходимо установить пакет linux-x.x.x.x-dbgsym (x.x.x.x заменить на версию ядра), который содержит символы отладки для используемого ядра. После этого необходимо выполнить инициализацию SystemTap вызвав команду stap-prep (от имени администратора).

|  |  |
| --- | --- |
| tid() | Идентификатор текущей нитки |
| pid() | Идентификатор процесса (task group) текущей нитки |
| uid() | ID пользователя |
| execname() | Имя текущего процесса |
| cpu() | Номер текущего CPU |
| gettimeofday\_s() | Сколько секунд прошло с начала эпохи |
| get\_cycles() | Снапшот хардварного счетчика циклов |
| pp() | Строка, описывающая сработавший проб (probe point) |
| ppfunc() | Имя функции, в которую был помещен сработавший проб |
| $$vars | Список всех локальных переменных в текущем пространстве |
| print\_backtrace() | Вывести ядерный стектрейс |
| print\_ubacktrace() | Вывести пользовательский стектрейс |

### Заменяется адреса функций в таблице системных вызовов

На основе решения был написан модуль ядра для замены функций системных вызовов syslog, ptrace, uname, sysinfo. Разработанный модуль одинаково работает в системах с ядром 2.6.32-21-generic и 4.4.0-31-generic.

Узнаем адрес sys\_call\_table:

|  |
| --- |
| **// В Ubuntu 10.04**  anar@ubuntu:~$ sudo cat /boot/System.map-2.6.32-21-generic | grep sys\_call\_table  c0592150 R sys\_call\_table  **// В Ubuntu 14.04**  anar@ubuntu:~/ub14.04$ sudo cat /boot/System.map-4.4.0-31-generic | grep sys\_call\_table  ffffffff818001c0 R sys\_call\_table  ffffffff81801500 R ia32\_sys\_call\_table |

Код модулей для разных версий ядра отличается только адресом таблицы с системными вызовами.

Код модуля replace.c:

|  |
| --- |
| #include <linux/kernel.h>  #include <linux/module.h>  #include <linux/moduleparam.h>  #include <linux/unistd.h>  #include <asm/cacheflush.h>  #include <linux/kallsyms.h>  #include <linux/utsname.h>  #define GPF\_DISABLE write\_cr0(read\_cr0() & (~ 0x10000))  #define GPF\_ENABLE write\_cr0(read\_cr0() | 0x10000)  // sys\_call\_table address in System.map  void \*\*sys\_call\_table = (void\*)0xffffffff818001c0;  long unsigned int addr = 0xffffffff818001c0;  asmlinkage int (\*original\_call) (const char\*, int, int);  asmlinkage int (\*original\_syslog) (int, int, int);  asmlinkage int (\*original\_uname) (struct old\_utsname \_\_user \*);  asmlinkage int (\*original\_ptrace) (int, int);  asmlinkage int (\*original\_sysinfo) (int, int);  asmlinkage int my\_uname(struct old\_utsname \_\_user \*name)  { printk("System call uname was catched!\n");  return original\_uname(name);  } asmlinkage long my\_setpriority(int which, int who, int niceval)  { printk("System call setpriority was catched: which = %d, who = %d, niceval = %d\n", which, who, niceval);  return original\_setpriority(which, who, niceval);  } asmlinkage long my\_getpriority(int which, int who)  { printk("System call getpriority was catched: which = %d, who = %d\n", which, who);  return original\_getpriority(which, who);  } int init\_module()  { original\_uname = sys\_call\_table[\_\_NR\_uname];  original\_setpriority = sys\_call\_table[\_\_NR\_setpriority];  original\_getpriority = sys\_call\_table[\_\_NR\_getpriority];  GPF\_DISABLE;  sys\_call\_table[\_\_NR\_uname] = my\_uname;  sys\_call\_table[\_\_NR\_setpriority] = my\_setpriority;  sys\_call\_table[\_\_NR\_getpriority] = my\_getpriority;  GPF\_ENABLE;  return 0;  } void cleanup\_module()  { // Restore the original calls  GPF\_DISABLE;  sys\_call\_table[\_\_NR\_uname] = original\_uname;  sys\_call\_table[\_\_NR\_setpriority] = original\_setpriority;  sys\_call\_table[\_\_NR\_getpriority] = original\_getpriority;  GPF\_ENABLE;  } |

**Список используемых источников:**