Модуль 9. Данные

Типы данных и структуры ядра

Мы должны сделать несколько замечаний о вопросах переносимости. Современные версии ядра Linux весьма переносимы, работают на большой числе различных архитектур. Учитывая мультиплатформенный характер Linux, драйверы, предназначенные для серьёзного использования также должны быть переносимыми.

Но основными проблемами с кодом ядра являются возможность доступа к объектам данных известной длины (например, структуры данных файловой системы или регистры на плате устройства) и эксплуатация возможностей разных процессоров (32-х разрядных и 64-х разрядной архитектур и, возможно, также 16-ти разрядных).

Некоторые из проблем, с которыми столкнулись разработчики ядра при переносе кода x86 на новые архитектуры, были связаны с неправильной типизацией данных. Соблюдением строгой типизации данных и компиляцией с флагами -Wall -Wstrict-prototypes можно предотвратить большинство ошибок.

Типы данных, используемые данными ядра, разделены на три основных класса: стандартные типы Си, такие как int, типы с явным размером, такие как u32 и типы, используемые для определённых объектов ядра, такие как pid_t. Мы собираемся показать когда и каким образом должен быть использован каждый из трёх типовых классов. Также мы поговорим о некоторых других типичных проблемах, с которыми можно столкнуться при переносе кода драйвера с x86 на другие платформы.

Если вы будете следовать предлагаем принципам, ваш драйвер должен компилироваться и работать даже на платформах, на которых вы не сможете его протестировать.

Специфика использования стандартных типов данных

Хотя большинство программистов привыкли свободно использовать стандартные типы, такие как int и long, написание драйверов устройств требует некоторой осторожности, чтобы избежать конфликтов типов и неясных ошибок.

Проблема в том, что вы не можете использовать стандартные типы, когда необходим "2-х байтовый наполнитель" или "что-то, представляющее 4-х байтовые строки", потому что обычные типы данных Си не имеют одинакового размера на всех архитектурах. В таблице показаны размеры данных различных типов Си на разных архитектурах:

\Size arch	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386	1	2	4	4	4	8	1	2	4	8
alpha	1	2	4	8	8	8	1	2	4	8
armv4l	1	2	4	4	4	8	1	2	4	8
ia64	1	2	4	8	8	8	1	2	4	8
m68k	1	2	4	4	4	8	1	2	4	8
mips	1	2	4	4	4	8	1	2	4	8
ppc	1	2	4	4	4	8	1	2	4	8
sparc	1	2	4	4	4	8	1	2	4	8
sparc64	1	2	4	4	4	8	1	2	4	8
x86_64	1	2	4	8	8	8	1	2	4	8

Интересно отметить, что архитектура SPARC 64 работает с 32-х разрядным пространством пользователя, имея там указатели размерностью 32 бита, хотя они и 64 бита в пространстве ядра.

Хотя вы должны быть осторожны при смешивании различных типов данных, иногда для этого бывают веские причины. Одной из таких ситуаций является адресация памяти, которая особо касается ядра. Хотя концептуально адреса являются указателями, управление памятью зачастую лучше выполняется с использованием типа беззнакового целого; ядро обрабатывает физическую память, как огромный массив, и адрес памяти является просто индексом внутри массива. Кроме того, указатель легко разыменовывается; при непосредственной работе с адресами памяти вы почти никогда не хотите их разыменовывать таким образом. Использование целочисленного типа мешает такому разыменованию, что позволяет

избежать ошибок. Таким образом, обычные адреса памяти в ядре, как правило, unsigned long, эксплуатируя тот факт, что указатели и целые long всегда одного и того же размера, по крайней мере, на всех платформах в настоящее время поддерживаемых Linux.

Кому это интересно, стандарт C99 определяет для целочисленной переменной типы intptr_t и uintptr_t, которая может содержать значение указателя.

Назначение типам данных явного размера

Иногда код ядра требует элементов данных определённого размера, может быть, чтобы соответствовать предопределённым бинарным структурам (это происходит при чтении таблицы разделов, когда запускается бинарный файл, либо при декодировании сетевого пакета), для общения с пользовательским пространством, или выравнивания данных в структурах включением "добивочных" полей (рассмотрим ниже).

Когда вам необходимо знать размер ваших данных, ядро предлагает для использования следующие типы данных. Все типы объявлены в <asm/types.h>, который, в свою очередь, подключается через <linux/types.h>:

```
u8; /* беззнаковый байт byte (8 бит) */
u16; /* беззнаковое слово (16 бит) */
u32; /* беззнаковая 32-х битовая переменная */
u64; /* беззнаковая 64-х битовая переменная */
```

Существуют соответствующие знаковые типы, но они требуются редко; если они вам необходимы, просто заменить в имени u на s.

Если программе пространства пользователя необходимо использовать эти типы, она может прибавить префикс к именам с двойным подчеркиванием: __u8 и другие типы определяются независимо от __KERNEL__. Если, например, драйверу необходимо обменяться бинарными структурами с программой, работающей в пользовательском пространстве посредством ioctl, файлы заголовков должны объявить 32-х разрядные поля в структурах как __u32.

Важно помнить, что эти типы специфичны для Linux, и их использование препятствует переносимости программ на другие виды Unix. Системы с последними компиляторами поддерживают типы стандарта C99, такие как uint8_t и uint32_t; если переносимость вызывает обеспокоенность, вместо разнообразных специфичных для Linux могут быть использованы эти типы.

Можно также отметить, что иногда ядро использует обычные типы, такие, как unsigned int, для элементов, размерность которых зависит от архитектуры. Это обычно сделано для обратной совместимости. Когда в версии 1.1.67 были введены u32 и друзья, разработчики не могли изменить существующие структуры данных в соответствие с новыми типами, потому что компилятор выдаёт предупреждение, когда есть несоответствие между типом поля структуры и присваиваемой ему переменной. На самом деле, компилятор сигнализирует о несоответствии типа, даже если два типа просто разные названия одного и того же объекта, такие как unsigned long и u32 на ПК. Линус не ожидал, что операционная система (ОС),

которую он написал для собственного пользования, станет мультиплатформенной; как следствие, старые структуры иногда типизированы небрежно.

Специфичные типы данных

Некоторые из наиболее часто используемых типов данных в ядре имеют свои собственные операторы typedef, предотвращая таким образом любые проблемы переносимости. Например, идентификатор процесса (pid) обычно pid_t, вместо int. Использование pid_t маскирует любые возможные различия в фактическом типе данных. Мы используем выражения, специфичные для интерфейса, для обозначения типов, определённых библиотекой, с тем, чтобы предоставить интерфейс к определённой структуре данных.

Отметим, что в последнее время были определено сравнительно мало новых типов, специфичных для интерфейса. Использование оператора typedef вошло в немилость у многих разработчиков ядра, которые предпочли бы видеть реальную информацию об используемом типе непосредственно в коде, а не скрывающейся за определённым пользователем типом. Однако, многие старые специфичные для интерфейса типы остаются в ядре и они не уйдут в ближайшее время.

Даже тогда, когда нет определённого интерфейсного типа, всегда важно использовать правильный тип данных для совместимости с остальной частью ядра. Счётчик тиков, например, всегда unsigned long, независимо от его фактического размера, так что при работе с тиками всегда должен быть использован тип unsigned long. В этом разделе мы сосредоточимся на использовании типов _t.

Многие _t типы определены в <linux/types.h>, но этот список редко бывает полезен. Когда вам потребуется определённый тип, выбудете находить его в прототипе функции, которую вам необходимо вызвать, или в структурах используемых данных.

Каждый раз, когда драйвер использует функции, которые требуют такого "заказного" типа, и вы не следуете соглашению, компилятор выдаёт предупреждение; если вы используете флаг компилятора -Wall и тщательно удаляете все предупреждения, вы можете быть уверены, что ваш код является переносимым.

Основной проблемой с элементами данных _t является то, что когда вам необходимо их распечатать, не всегда просто выбрать правильный формат printk или printf и предупреждения, которые вы убираете на одной архитектуре, снова появляются на другой. Например, как бы вы напечатали size_t, который является unsigned long на одних платформах и unsigned int на других?

Если вам необходимо распечатать некоторые специфичные для интерфейса данные, наилучшим способом сделать это является приведение типа значения к наибольшему возможному типу (как правило, long или unsigned long) и затем печать их через соответствующий формат. Такой вид корректировки не будет

генерировать ошибки или предупреждения, потому что формат соответствует типу, и вы не потеряете биты данных, потому что приведение либо ничего не делает, либо увеличивает объект до большего типа данных.

На практике, объекты данных, о которых мы говорим, обычно не предназначены для печати, так что вопрос распространяется только на сообщения об отладке. Чаще всего коду необходимо только запоминать и сравнивать типы, специфичные для интерфейса, в дополнение к передаче их в качестве аргумента в библиотечные функции или функции ядра.

Хотя _t типы являются правильным решением для большинства ситуаций, иногда правильного типа не существует. Это происходит на старых интерфейсах, которые ещё не были очищены.

Одним неоднозначным моментом, который мы нашли в заголовках ядра, является типизация данных для функций ввода/вывода, которые определены небрежно. Небрежные типы в основном существуют по историческим причинам, но это может создать проблемы при написании кода. Например, можно попасть в беду, меняя аргументы функций, таких как outb; если бы существовал тип port_t, компилятор бы нашёл этот тип ошибки.

Аспекты, связанные с совместимостью

В дополнение к типам данным есть несколько других программных вопросов, чтобы иметь ввиду при написании драйвера, если вы хотите, чтобы он был переносим между платформами Linux. Общее правило состоит в том, чтобы относиться с подозрением к явным постоянным значениям. Обычно код параметризован помощью макросов препроцессора. Всякий раз, когда вы сталкиваетесь другими значениями, которые были параметризованы, вы можете найти подсказки в файлах заголовков и драйверах устройств, распространяемых с официальным ядром.

Измерение временных интервалов

Когда речь идёт о временных интервалах, не думайте, что есть 1000 тиков в секунду. Хотя в настоящее время для архитектуры i386 это справедливо, не каждая платформа Linux работает на этой скорости. Предположение может быть ложным даже для x86, если вы играете со значением HZ (как делают некоторые люди), и никто не знает, что произойдёт в будущих ядрах. Всякий раз, когда вы рассчитываете интервалы времени используя тики, масштабируйте ваши времена с помощью HZ (число прерываний таймера в секунду). Например, чтобы проверить ожидание в пол-секунды, сравнивайте прошедшее время с HZ/2. Более широко, число тиков, соответствующее msec миллисекунд, всегда msec*HZ/1000.

Размер страницы

При играх с памятью помните, что память страницы - PAGE_SIZE байт, а не 4 Кб. Предполагая, что размер страницы составляет 4 Кбайт и явное указание этого значения является распространённой ошибкой среди программистов ПК, вместо этого, поддерживаемые платформы показывают размер страницы от 4 Кб до 64 Кб и иногда они различаются между разными реализациями одной и той же платформы. Соответствующими макросами являются PAGE_SIZE и PAGE_SHIFT. Последний содержит число битов для сдвига адреса, чтобы получить номер страницы. В настоящее время число составляет 12 или больше для страниц, которые 4 Кб и более. Макросы определены в <asm/page.h>; программы пространства пользователя могут использовать библиотечную функцию getpagesize, если им когда-нибудь потребуется такая информация.

Давайте посмотрим на нетривиальную ситуацию. Если драйверу необходимо 16 Кб для временных данных, не следует указывать order как 2 для get_free_pages. Вам необходимо переносимое решение. Такое решение, к счастью, был написано разработчиками ядра и называется get_order:

```
#include <asm/page.h>
int order = get_order(16*1024);
buf = get_free_pages(GFP_KERNEL, order);
```

Помните, что аргумент get_order должно быть степенью двойки.

Порядок следования байт в словах

Будьте внимательны, чтобы не делать предположений о порядке байт. Если ПК сохраняют многобайтовые величины начиная с младшего байта (сначала младший конец, то есть little-endian), некоторые высокоуровневые платформы делают это другим способом (big-endian). Когда это возможно, ваш код должен быть написан так, чтобы не заботиться о порядке байт в данных, которыми он манипулирует. Однако, иногда драйверу необходимо построить целое число из раздельный байтов или сделать обратное, или он должен взаимодействовать с устройством, которое ожидает определённый порядок.

Подключаемый файл <asm/byteorder.h> определяет либо __BIG_ENDIAN, либо __LITTLE_ENDIAN, в зависимости от порядка байт в процессоре. Имея дело с вопросами порядка байт, вы могли бы написать кучу условий #ifdef __LITTLE_ENDIAN, но есть путь лучше. Ядро Linux определяет набор макросов, которые занимаются переводом между порядком байтов процессора и теми данными, которые необходимо сохранять или загружать с определённым порядком байтов. Например:

```
u32 cpu_to_le32 (u32);
u32 le32_to_cpu (u32);
```

Эти два макроса преобразуют значение от любого используемого процессором в unsigned, little-endian, 32-х разрядное и обратно. Они работают независимо от того, использует ли ваш процессор big-endian или little-endian и, если на то пошло, является ли он 32-х разрядным процессором или нет. Они возвращают их аргумент неизменным в тех случаях, где нечего делать. Использование этих макросов позволяет легко писать переносимый код без необходимости использовать большое количество конструкций условной компиляции.

Существуют десятки подобных процедур; вы можете увидеть их полный список в <linux/byteorder/big_endian.h> и <linux/byteorder/little_endian.h>. Спустя некоторое время шаблону не трудно следовать. be64_to_cpu преобразует unsigned, big-endian, 64-х разрядное значение во внутреннее представление процессора. le16_to_cpus вместо этого обрабатывает signed, little-endian, 16-ти разрядное значение. При работе с указателями вы можете также использовать функции, подобные cpu_to_le32p, которые принимают указатель на значение, которое будет преобразовано, вместо самого значения указателя. Для всего остального смотрите подключаемый файл.

Выравнивание данных

Последней проблемой, заслуживающей рассмотрения при написании переносимого кода, является то, как получить доступ к невыровненным данным, например, как прочитать 4-х байтовое значение, хранящееся по адресу, который не кратен 4-м байтам. Пользователи i386 часто адресуют невыровненные элементы данных, но не все архитектуры позволяют это. Многие современные архитектуры генерируют исключение каждый раз, когда программа пытается передавать невыровненные данные; передача данных обрабатывается обработчиком исключения с большой потерей производительности. Если вам необходимо получить доступ невыровненным данным, вам следует использовать следующие макросы:

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

Эти макросы безтиповые и работают для каждого элемента данных, будь они один, два, четыре, или восемь байт длиной. Они определяются в любой версии ядра.

Другой проблемой, связанная с выравниванием, является переносимость структур данных между разными платформами. Такая же структура данных (как определена в исходном файлена языке Си) может быть скомпилирована по-разному на разных платформах. Компилятор передвигает поля структуры, для соответствия соглашениям, которые отличаются от платформы к платформе.

Для записи структур данных для элементов данных, которые могут перемещаться между архитектурами, вы должны всегда следовать естественному выравниванию элементов данных в дополнение к стандартизации на определённый порядок байтов. Естественное выравнивание означает хранение объектов данных по адресу, кратному их размеру (например, 8-ми байтовые объекты располагаются по адресам, кратным 8). Для принудительного естественного выравнивания, чтобы не допустить организацию полей компилятором непредсказуемым образом, вы должны использовать поля-заполнители, во избежание оставления пустот в структуре данных.

В таблице показано, как компилятором выполняется выравнивание нескольких платформах:

\Align arch	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386	1	2	4	4	4	4	1	2	4	4
i686	1	2	4	4	4	4	1	2	4	4
alpha	1	2	4	8	8	8	1	2	4	8
armv4l	1	2	4	4	4	4	1	2	4	4
ia64	1	2	4	8	8	8	1	2	4	8
mips	1	2	4	4	4	8	1	2	4	8
ppc	1	2	4	4	4	8	1	2	4	8
sparc	1	2	4	4	4	8	1	2	4	8
sparc64	1	2	4	4	4	8	1	2	4	8
x86_64	1	2	4	8	8	8	1	2	4	8

Интересно отметить, что не на всех платформах 64-х разрядные значения выровнены по 64-х битной границе, так что вам необходимо заполнить поля для обеспечения выравнивания и обеспечения переносимости.

Наконец, необходимо учитывать, что компилятор может спокойно вставить заполнитель в структуру сам, чтобы обеспечить выравнивание каждого поля для хорошей производительности на целевом процессоре. Если вы определяете структуру, которая призвана соответствовать структуре, ожидаемой устройством, это автоматическое заполнение может помешать вашей попытке. Способом преодоления этой проблемы является сказать компилятору, что структура должна быть "упакована" без добавления наполнителей. Например, файл заголовка ядра <linux/edd.h> определяет несколько структур данных, используемых для взаимодействия с BIOS x86, и включает в себя следующие определения:

```
struct {
  u16 id;
  u64 lun;
  u16 reserved1; u32 reserved2;
} __attribute__ ((packed)) scsi;
```

Без такого __attribute__ ((packed)) полю lun предшествовало бы два заполняющих байта или шесть, если бы мы компилировали структуру на 64-х разрядной платформе.

Размер указателя

Многие внутренние функции ядра возвращают вызывающему значение указателя. Многие из этих функций также могут закончится неудачно. В большинстве случаев отказ указывается возвращением указателя со значением NULL. Этот метод работает, но он не может сообщить точную природу проблемы. Некоторым интерфейсам действительно необходимо вернуть собственно код ошибки, с тем, чтобы вызвавший мог сделать правильное решение, основанное на том, что на самом деле не так.

Некоторые интерфейсы ядра возвращают эту информацию кодируя код ошибки в значении указателя. Такие функции должны использоваться с осторожностью, поскольку их возвращаемое значение нельзя просто сравнить с NULL. Чтобы помочь в создании и использовании подобного вида интерфейса, предоставлен небольшой набор функций (в <linux/ err.h>).

Функция, возвращающая тип указателя может, вернуть значение ошибки с помощью:

```
void *ERR_PTR(long error);
```

где error является обычным отрицательным кодом ошибки. Для проверки, является ли возвращённый указатель кодом ошибки или нет, вызвавший может использовать IS_ERR:

```
long IS_ERR(const void *ptr);
```

Если вам необходим настоящий код ошибки, он может быть извлечён с помощью:

```
long PTR_ERR(const void *ptr);
```

Вы должны использовать PTR_ERR только для значения, для которого IS_ERR возвращает значение "истина"; любое другое значение является правильным указателем.