Модуль 6. Управление памятью в ядре Linux

Специфика управления памятью в ядре Linux

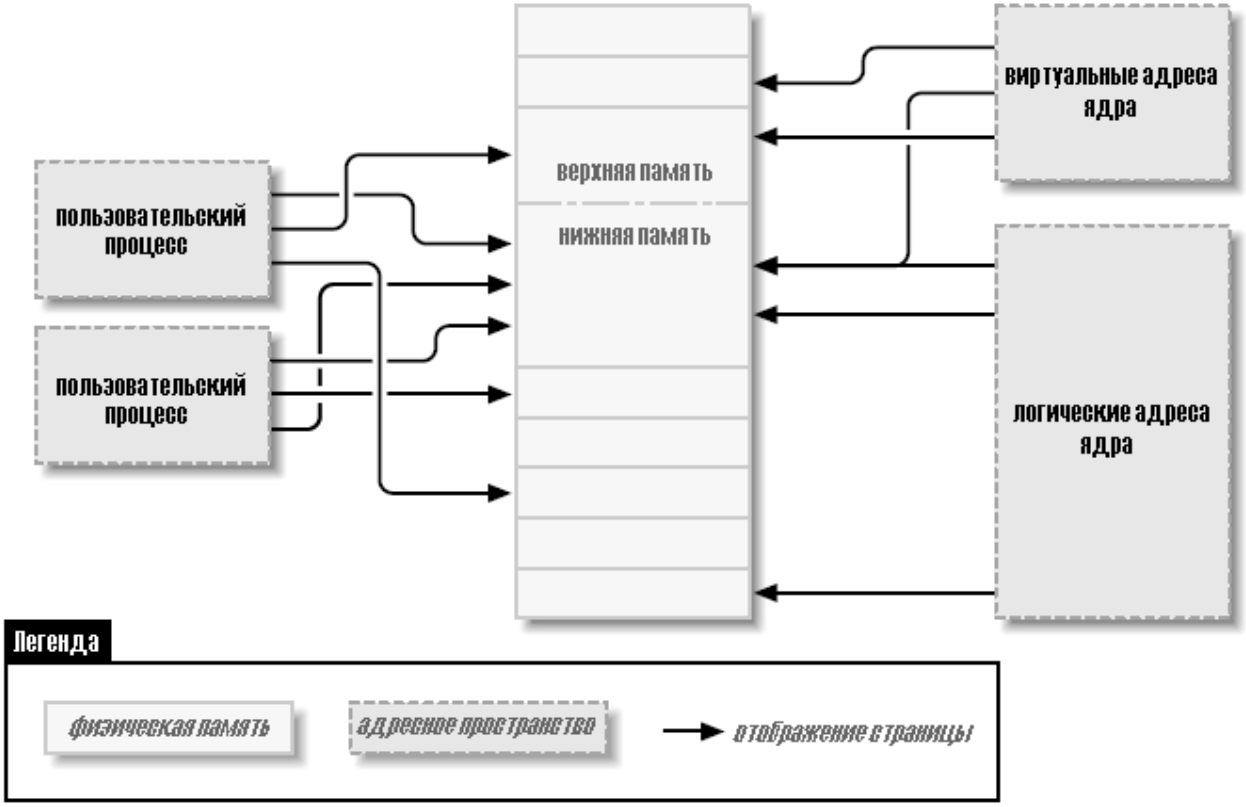
Вместо того, чтобы описывать теорию управления памятью в операционных системах, этот раздел попытается выявить основные особенности её реализации в Linux. Хотя вам не требуется быть гуру в виртуальный памяти Linux для выполнения mmap, основной обзор того, как всё это работает, полезен. Далее следует довольно пространное описание структур данных, используемых ядром для управления памятью. После получения необходимых предварительных знаний мы сможем приступить к работе с этими структурами.

Типы адресов

Linux, конечно же, система с виртуальной памятью, а это означает, что адреса, видимые пользовательскими программами, не соответствуют напрямую физическим адресам, используемым оборудованием. Виртуальная память вводит слой косвенности, который позволяет ряд приятных вещей. С виртуальной памятью программы, выполняющиеся в системе, могут выделить гораздо больше памяти, чем доступно физически; более того, даже один процесс может иметь виртуальное адресное пространство больше физической памяти системы. Виртуальная память позволяет также программе использовать разные ухищрения с адресным пространством процесса, в том числе отображение памяти программы в память устройства.

До сих пор мы говорили о виртуальных и физических адресах, но подробности умалчивались. Система Linux имеет дело с несколькими типами адресов, каждый со своей собственной семантикой. К сожалению, в коде ядра не всегда чётко понятно, какой именно тип адреса используется в каждой ситуации, так что программист должен быть осторожным.

Ниже приведён список типов адресов используемых в Linux. Следующий рисунок показывает, как эти типы адресов связаны с физической памятью.



Пользовательские виртуальные адреса

Это обычные адреса, видимые программами пространства пользователя. Пользовательские адреса имеют размерность 32 или 64 бита, в зависимости от архитектуры используемого оборудования, и каждый процесс имеет своё собственное виртуальное адресное пространство.

Физические адреса

Адреса, используемые между процессором и памятью системы. Физические адреса 32-х или 64-х разрядные; даже 32-х разрядные системы могут использовать большие физические адреса в некоторых ситуациях.

Адреса шин

Адреса, используемые между периферийными шинами и памятью. Зачастую они такие же, как физические адреса, используемые процессором, но это не обязательно так. Некоторые архитектуры могут предоставить блок управления памятью ввода/вывода (I/O memory management unit, IOMMU), который переназначает адреса между шиной и оперативной памяти. IOMMU может сделать жизнь легче несколькими способами (делая разбросанный в памяти буфер выглядящим непрерывным для устройства, например), но программирование IOMMU является дополнительным шагом, который необходимо выполнить при настройке DMA операций. Конечно, адреса шин сильно зависят от архитектуры.

Логические адреса ядра

Они составляют обычное адресное пространство ядра. Эти адреса отображают какую-то часть (возможно, всю) основной памяти и часто рассматриваются, как если бы они были физическими адресами. На большинстве архитектур логические адреса и связанные с ними физические адреса отличаются только на постоянное смещение. Логические адреса используют родной размер указателя оборудования и, следовательно, могут быть не в состоянии адресовать всю физическую память на 32-х разрядных системах, оборудованных в большей степени. Логические адреса обычно хранятся в переменных типа unsigned long или void *. Память, возвращаемая kmalloc, имеет логический адрес ядра.

Виртуальные адреса ядра

Виртуальные адреса ядра похожи на логические адреса в том, что они являются отображением адреса пространства ядра на физический адрес. Однако, виртуальные адреса ядра не всегда имеют линейную, взаимно-однозначную связь с физическими адресами, которая характеризует логическое адресное пространство. Все логические адреса являются виртуальными адресами ядра, но многие виртуальные адреса ядра не являются логическими адресами. Так, например, память, выделенная vmalloc, имеет виртуальный адрес (но без прямого физического отображения). Функция kmap также возвращает виртуальные адреса. Виртуальные адреса обычно хранятся в переменных указателях.

Если у вас есть логический адрес, макрос __pa( ) (определённый в <asm/page.h>) возвращает соответствующий ему физический адрес. Физические адреса могут быть преобразованы обратно в логические адреса с помощью __va( ), но только для нижних страниц памяти.

Различные функции ядра требуются разные типы адресов. Было бы неплохо, если бы были определены различные типы Си, так, чтобы необходимый тип адреса был бы явным, но этого нет. Дальше мы постараемся ясно указать, какой тип адресов где используется.

Преобразование адреса в ядре

Физические адреса и страницы

Физическая память разделена на отдельные модули, называемые страницами. Значительная часть внутренней системной обработки памяти производится на постраничной основе. Размер страницы варьируется от одной архитектуры к другой, хотя большинство систем в настоящее время использует страницы по 4096 байт. Постоянная PAGE_SIZE (определённая в <asm/page.h>) задаёт размер страницы для любой архитектуры.

Если вы посмотрите на адрес памяти, виртуальный или физический, он делится на номер страницы и смещение внутри этой страницы. Например, если используются страницы по 4096 байт, 12 младших значащих бит являются смещением, а остальные, старшие биты, указывают номер страницы. Если отказаться от смещения и сдвинуть оставшуюся часть адреса вправо, результат называют номером страничного блока (page frame number, PFN). Сдвиг битов для конвертации между номером страничного блока и адресами является довольно распространённой операцией; макрос PAGE_SHIFT сообщает, сколько битов должны быть смещены для выполнения этого преобразования.

Верхняя и нижняя память

Разница между логическими и виртуальными адресами ядра хорошо видна на 32-х разрядных системах, которые оборудованы большими объёмами памяти. Используя 32 бита можно адресовать 4 ГБ памяти. Однако, Linux на 32-х разрядных системах до недавнего времени был ограничен значительно меньшей памятью, чем это, из-за способа инициализации виртуального адресного пространства.

Ядро (на архитектуре x86, в конфигурации по умолчанию) разделяет 4 ГБ виртуальное адресное пространство между пространством пользователя и ядром; в обоих контекстах используется один и тот же набор отображений. Типичное разделение выделяет 3 Гб для пространства пользователя и 1 ГБ для пространства ядра. (* Многие не-x86 архитектуры способны эффективно обходиться без описанного здесь разделения на ядро/ пользовательское пространство, так что они могут работать с адресным пространством ядра до 4 ГБ на 32-х разрядных системах. Однако, ограничения, описанные в этом разделе, до сих пор относятся к таким системам, где установлено более 4 ГБ оперативной памяти.) Код ядра и структуры данных должны вписываться в это пространство, но самым большим потребителем адресного пространства ядра является виртуальное отображение на физическую память. Ядро не может напрямую управлять памятью, которая не отображена в адресное пространство ядра. Ядру, другими

словами, необходим свой виртуальный адрес для любой памяти, с которой оно должно непосредственно соприкасаться. Таким образом, на протяжении многих лет, максимальным объёмом физической памяти, которая могла быть обработана ядром, было значение, которое могло быть отображено в часть для ядра виртуального адресного пространства, минус пространство, необходимое для самого кода ядра. В результате, базирующиеся на x86 системы Linux могли работать максимально с немногим менее 1 ГБ физической памяти.

В ответ на коммерческое давление, чтобы поддержать больше памяти, не нарушая в то же время работу 32-х разрядных приложений и совместимость системы, производители процессоров добавили в свои продукты функцию "расширение адреса". Результатом является то, что во многих случаях даже 32-х разрядные процессоры могут адресовать более 4 ГБ физической памяти. Однако, ограничение на объём памяти, которая может быть непосредственно связана с логическими адресами, остаётся. Только самая нижняя часть памяти (до 1 или 2 ГБ, в зависимости от оборудования и конфигурации ядра) имеет логические адреса; остальная (верхняя память) не имеет. Перед доступом к заданной странице верхней памяти ядро должно установить явное виртуальное соответствие, чтобы сделать эту страницу доступной в адресном пространстве ядра. Таким образом, многие структуры данных ядра должны быть размещены в нижней памяти; верхняя память имеет тенденцию быть зарезервированной для страниц процесса пространства пользователя.

Термин "верхняя память" может ввести некоторых в заблуждение, особенно поскольку он имеет другое значение в мире персональных компьютеров. Таким образом, чтобы внести ясность, мы определим здесь эти термины:

Нижняя память

Память, для которой существуют логические адреса в пространстве ядра. Почти на каждой системе, с которой вы скорее всего встретитесь, вся память является нижней памятью.

Верхняя память

Память, для которой логические адреса не существуют, потому что она выходит за рамки диапазона адресов, отведённых для виртуальных адресов ядра.

На системах i386 граница между нижней и верхней памятью обычно установлена на уровне только до 1 Гб, хотя эта граница может быть изменена во время конфигурации ядра. Эта граница не связана никаким образом со старым ограничением 640 Кб, имеющимся на оригинальном ПК, и её размещение не продиктовано оборудованием. Напротив, это предел, установленный в самом ядре, так как он

разделяет 32-х разрядное адресное пространство между ядром и пространством пользователя.

Мы укажем на ограничения на использование верхней памяти, когда мы подойдём к ним в этой главе.

Карта памяти и структура page

Исторически сложилось, что для обращения к страницам физической памяти ядро использует логические адреса. Однако, добавление поддержки верхней памяти выявило очевидную проблему с таким подходом - логические адреса не доступны для верхней памяти.

Таким образом, функции ядра, которые имеют дело с памятью, вместо этого всё чаще используют указатели на struct page (определённой в <linux/mm.h>). Эта структура данных используется для хранения информации практически обо всём, что ядро должно знать о физической памяти; для каждой физической страницы в системе существует одна struct page. Некоторые из полей этой структуры включают следующее:

atomic_t count;

Число существующих ссылок на эту страницу. Когда количество падает до 0, страница возвращается в список свободных страниц.

void *virtual;

Виртуальный адрес страницы ядра, если она отображена; в противном случае NULL. Страницы нижней памяти отображаются всегда; страницы верхней памяти - обычно нет.

Это поле появляется не на всех архитектурах; оно обычно компилируется только тогда, когда виртуальный адрес страницы ядра не может быть легко вычислен. Если вы хотите посмотреть на это поле, правильный метод заключается в использовании макроса page_address, описанного ниже.

unsigned long flags;

Набор битовых флагов, характеризующих состояние этой странице. К ним относятся PG_locked, который указывает, что страница была заблокирована в памяти, и PG_reserved, который препятствует тому, чтобы система управления памятью вообще работала с этой страницей.

Внутри struct page существует гораздо больше информации, но она является частью более глубокой чёрной магии управления памятью и не представляет интерес для авторов драйверов.

Ядро поддерживает один или несколько массивов записей struct page, которые позволяют отслеживать всю физическую память системы. На некоторых системах имеется единственный массив, называемый mem_map. На других системах, однако, ситуация более

сложная. Системы с неоднородным доступом к памяти (nonuniform memory access, NUMA) и другие с сильно разделённой физической памятью могут иметь более одного массива карты памяти, поэтому код, который предназначен для переносимости, должен избегать прямого доступа к массиву, когда это возможно. К счастью, как правило, довольно легко просто работать с указателями struct page не беспокоясь о том, откуда они берутся.

Некоторые функции и макросы, определённые для перевода между указателями struct page и виртуальными адресами:

```
struct page *virt_to_page(void *kaddr);
```

Этот макрос, определённый в <asm/page.h>, принимает логический адрес ядра и возвращает связанный с ним указатель struct page. Так как он требует логического адрес, он не работает с памятью от vmalloc или верхней памятью.

```
struct page *pfn_to_page(int pfn);
```

Возвращает указатель struct page для заданного номера страничного блока. При необходимости он проверяет номер страничного блока на корректность с помощью pfn_valid перед его передачей в pfn_to_page.

```
void *page_address(struct page *page);
```

Возвращает виртуальный адрес ядра этой страницы, если такой адрес существует. Для верхней памяти этот адрес существует, только если страница была отображена. Эта функция определена в <linux/mm.h>. В большинстве случаев вы захотите использовать версию kmap, а не page_address.

```
#include <linux/highmem.h>
void *kmap(struct page *page);
void kunmap(struct page *page);
```

kmap возвращает виртуальный адрес ядра для любой страницы в системе. Для страниц нижней памяти она просто возвращает логический адрес страницы; для страниц верхней памяти kmap создаёт специальное отображение в предназначенной для этого части адресного пространства ядра. Отображения, созданные kmap, всегда должны быть освобождены с помощью kunmap; доступно ограниченное число таких отображений, так что лучше не удерживать их слишком долго. Вызовы kmap поддерживают счётчик, так что если две или более функции вызывают kmap на той же странице, всё работает правильно. Отметим также, что kmap может заснуть, если отображение недоступно.

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

kmap_atomic является высокопроизводительной формой kmap. Каждая архитектура поддерживает небольшой список слотов (специализированные записи таблицы страниц) для атомарных kmap-ов; вызывающий kmap_atomic должен сообщить системе в аргументе type, какой из этих слотов использовать. Единственными слотами, которые имеют смысл для драйверов, являются KM_USER0 и KM_USER1 (для кода, работающего непосредственно из вызова из пользовательского пространства), и KM_IRQ0 и KM_IRQ1 (для обработчиков прерываний). Обратите внимание, что атомарные kmap-ы должны быть обработаны атомарно; ваш код не может спать, удерживая её. Отметим также, что ничто в ядре не предохраняет две функции от попытки использовать тот же слот и помешать друг другу (хотя для каждого процессора имеется уникальный набор слотов). На практике, конкуренция для атомарных слотов kmap, кажется, не будет проблемой.

Таблицы страниц

В любой современной системе процессор должен иметь механизм для трансляции виртуальных адресов в соответствующие физические адреса. Этот механизм называется таблицей страниц; по существу, это многоуровневый древовидный массив, содержащий отображения виртуального к физическому и несколько связанных флагов. Ядро Linux поддерживает набор таблиц страниц даже на архитектурах, которые не используют такие таблицы напрямую.

Многие операции, обычно выполняемые драйверами устройств, могут включать манипуляции таблицами страниц. К счастью для автора драйвера, ядро ликвидировало всю необходимость непосредственно работать с таблицами страниц.

Области виртуальной памяти

Область виртуальной памяти (virtual memory area, VMA) представляет собой структуру данных ядра, используемую для управления различными регионами адресного пространства процесса. VMA представляет собой однородный регион в виртуальной памяти процесса: непрерывный диапазон виртуальных адресов, которые имеют одинаковые флаги разрешения и созданы одним и тем же объектом (скажем, файлом, или пространством для своппинга). Она примерно соответствует концепции "сегмента", хотя это лучше описывается как "объект памяти со своими свойствами". Карта памяти процесса состоит (по крайней мере) из следующих областей:

- Область для исполняемого кода программы (часто называемого текстом).
- Несколько областей для данных, в том числе проинициализированные данные (те,которые имеют явно заданные

значения в начале исполнения), неинициализированные данных (BSS), (* имя BSS является историческим пережитком старого ассемблерного оператора, означающего "блок, начатый символом" (“block started by symbol”). Сегмент BSS исполняемых файлов не сохраняется на диске и ядро отображает нулевую страницу в диапазоне адресов BSS.) и программный стек.

- По одной области для каждого активного отображения памяти.

Области памяти процесса можно увидеть, посмотрев в /proc/<pid>/maps (где pid, конечно, заменяется на ID процесса). /proc/self является особым случаем /proc/pid, потому что он всегда обращается к текущему процессу.

Управление отображением в память

Отображение памяти является одним из наиболее интересных особенностей современных систем UNIX. Что касается драйверов, отображение памяти может быть реализовано для предоставления пользовательским программам прямого доступа к памяти устройства.

Структура vm_area_struct

Когда процесс пользовательского пространства вызывает mmap, чтобы отобразить память устройства в его адресное пространство, система реагирует, создавая новую VMA для предоставления этого отображения. Драйвер, который поддерживает mmap (и, таким образом, который реализует метод mmap), должен помочь такому процессу завершая инициализацию этой VMA. Автор драйвера должен, следовательно, иметь по крайней мере минимальное понимание VMA для поддержки mmap.

Давайте посмотрим на наиболее важные поля в struct vm_area_struct (определённой в <linux/mm.h>). Эти поля могут быть использованы драйверами устройств в их реализации mmap. Заметим, что ядро поддерживает списки и деревья VMA для оптимизации области поиска и некоторые поля vm_area_struct используются для поддержки такой организации. Поэтому VMA не могут быть созданы по желанию драйвера, или эти структуры нарушатся. Основными полями VMA являются следующие:

```
unsigned long vm_start;
unsigned long vm_end;
```

Диапазон виртуальных адресов, охватываемый этой VMA. Эти поля являются первыми двумя полями, показываемыми в /proc/*/maps.

```
struct file *vm_file;
```

Указатель на структуру struct file, связанную с этой областью (если таковая имеется).

```
unsigned long vm_pgoff;
```

Смещение области в файле, в страницах. Когда отображается файл или устройство, это позиция в файле первой страницы, отображённой в эту область.

```
unsigned long vm_flags;
```

Набор флагов, описывающих эту область. Флагами, представляющими наибольший интерес для автора драйвера устройства, являются VM_IO и VM_RESERVED. VM_IO отмечает VMA как являющийся отображённым на память регион ввода/вывода. Среди прочего, флаг VM_IO мешает региону быть включенным в дампы процессов ядра. VM_RESERVED сообщает системе управления памятью не пытаться выгрузить эту VMA; он должен быть установлен в большинстве отображений устройства.

```
struct vm_operations_struct *vm_ops;
```
Набор функций, которые ядро может вызывать для работы в этой области памяти. Его присутствие свидетельствует о том, что область памяти является "объектом" ядра, как и struct file, которую мы рассматривали ранее.
```
void *vm_private_data;
```
Области, которые могут быть использованы драйвером для сохранения своей собственной информации.

Как и struct vm_area_struct, vm_operations_struct определена в <linux/mm.h>; она включает операции, перечисленные ниже. Эти операции являются единственными необходимыми для обработки потребностей процесса в памяти и они перечислены в том порядке, как они объявлены. Далее в этой главе реализованы некоторые из этих функций.
```
void (*open)(struct vm_area_struct *vma);
```
Метод open вызывается ядром, чтобы разрешить подсистеме реализации VMA проинициализировать эту область. Этот метод вызывается в любое время, когда создаётся новая ссылка на эту VMA (например, когда процесс разветвляется). Единственное исключение происходит, когда VMA создана впервые через mmap; в этом случае вместо этого вызывается метод драйвера mmap.
```
void (*close)(struct vm_area_struct *vma);
```
При удалении области ядро вызывает эту операцию close. Обратите внимание, что нет счётчика использований, связанного с VMA; область открывается и закрывается ровно один раз каждым процессом, который её использует.
```
struct page *(*nopage)(struct vm_area_struct *vma,
unsigned long address, int *type);
```
Когда процесс пытается получить доступ к странице, которая относится к действительной VMA, но которая в настоящее время не в памяти, для соответствующей области вызывается метод nopage (если он определён). Метод возвращает указатель struct page для физической страницы после того как, может быть, прочитал его из вторичного хранилища. Если для этой области метод nopage не определён, ядром выделяется пустая страница.
```
int (*populate)(struct vm_area_struct *vm, unsigned long
address, unsigned long len, pgprot_t prot, unsigned long
pgoff, int nonblock);
```
Этот метод позволяет ядру "повредить" страницы в памяти, прежде чем они будут доступны пользовательскому пространству. Вообще-то, драйверу нет необходимости реализовывать метод populate.

Карта памяти процесса

Последней частью головоломки управления памятью является структура карты памяти процесса, которая удерживает все другие структуры данных вместе. Каждый процесс в системе (за исключением нескольких вспомогательных потоков пространства ядра) имеет struct mm_struct (определённую в <linux/sched.h>), которая содержит список виртуальных областей памяти процесса, таблицы страниц и другие разные биты информации управления домашним хозяйством памяти вместе с семафором (mmap_sem) и спин-блокировкой (page_table_lock). Указатель на эту структуру можно найти в структуре задачи; в редких случаях, когда драйверу необходим к ней доступ, обычным способом является использование current->mm. Обратите внимание, что структура управления памятью может быть разделяемой между процессами; к примеру, таким образом работает реализация потоков в Linux.

Отображение файла в память ядра

Отображение устройства означает связывание диапазона адресов пользовательского пространства с памятью устройства. Всякий раз, когда программа читает или записывает в заданном диапазоне адресов, она на самом деле обращается к устройству. Для критичных к производительности приложений, таких как это, прямой доступ имеет большое значение.

Как вы могли бы подозревать, не каждое устройство поддаётся абстракции mmap; это не имеет смысла, например, для последовательных портов и других поточно-ориентированных устройств. Другим ограничением mmap является то, что отображение разделено на PAGE_SIZE. Ядро может управлять виртуальными адресами только на уровне таблиц страниц; таким образом, отображённая область должна быть кратной PAGE_SIZE и должна находиться в физической памяти начиная с адреса, который кратен PAGE_SIZE. Ядро управляет размером разбиения, делая регион немного больше, если его размер не является кратным размеру страницы.

Эти ограничения не являются большим препятствием для драйверов, потому что программа в любом случае обращается к устройству, зависящим от устройства способом. Поскольку программа должна знать о том, как работает устройство, программист не слишком обеспокоен необходимостью следить за деталями выравнивания страниц. Существует большое ограничение, когда на некоторых не-x86 платформах используются ISA устройства, потому что их аппаратное представление ISA может не быть непрерывным. Например, некоторые компьютеры Alpha видят ISA память как разбросанный набор 8-ми, 16-ти, или 32-х разрядных объектов без прямого отображения. В таких случаях вы не можете использовать mmap совсем. Неспособность выполнять прямое отображение адресов ISA в адреса Alpha связано с несовместимыми спецификациями передачи данных этих двух систем. В то время, как ранние процессоры Alpha могли выдавать только 32-х разрядные и 64-х разрядные обращения к памяти, ISA может делать только 8-ми разрядные и 16-ти разрядные передачи, и нет никакого способа для прозрачной связи одного протокола с другим.

Существуют веские преимущества использования mmap, когда это возможно сделать. Типичным примером является программа управления PCI устройством. Большинство PCI периферии отображает их управляющие регистры на адреса памяти и высокопроизводительные приложения, возможно, предпочтут иметь прямой доступ к регистрам, вместо того, чтобы постоянно вызывать ioctl для выполнения этой работы.

Метод mmap является частью структуры file_operations и вызывается, когда происходит системный вызов mmap. В случае с mmap, ядро выполняет много работы перед фактическим вызовом метода и, следовательно, прототип метода сильно отличается от системного вызова. Это является отличием от таких вызовов, как ioctl и poll, где до вызова метода ядро не делает ничего.

Системный вызов объявлен следующим образом (как описано на странице руководства mmap(2)):

```
mmap (caddr_t addr, size_t len, int prot, int flags,
int fd, off_t offset)
```

С другой стороны, файловая операция объявлена следующим образом:

```
int (*mmap) (struct file *filp, struct vm_area_struct
*vma);
```

Аргумент filp в методе такой же, как в примере символьного устройства, а vma содержит информацию о диапазоне виртуальных адресов, которые используются для доступа к устройству. Таким образом, большая часть работы выполнена ядром; для реализации mmap драйверу необходимо только построить подходящие таблицы страниц для диапазона адресов и, если необходимо, заменить vma->vm_ops новым набором операций.

Есть два способа построения таблиц страниц: сделать всё это единожды с помощью функции, названной remap_pfn_range, или делать по странице за раз, через метод VMA nopage. Каждый метод имеет свои преимущества и недостатки. Начнём с подхода "все сразу", который является более простым. После этого мы добавим осложнения, необходимые для настоящей реализации.

Использование remap_pfn_range

Работа по созданию новых таблиц страниц для отображения диапазона физических адресов выполняется remap_pfn_range и io_remap_page_range, которые имеют следующие прототипы:

```
int remap_pfn_range(struct vm_area_struct *vma,
unsigned long virt_addr, unsigned long pfn, unsigned
long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma,
unsigned long virt_addr, unsigned long phys_addr,
unsigned long size, pgprot_t prot);
```

Значение, возвращаемое функцией, является обычным 0 или отрицательным кодом ошибки. Давайте посмотрим на точное значение аргументов функций:

vma

Область виртуальной памяти, в которую сейчас отображается диапазон страниц.

virt_addr

Пользовательский виртуальный адрес, по которому должно начаться переназначение. Функция строит таблицы страниц для виртуального диапазона адресов между virt_addr и virt_addr+size.

pfn

Номер страничного блока, соответствующий физическому адресу, с которым должен быть связан виртуальный адрес. Номер страничного блока - это просто физический адрес, сдвинутый вправо на PAGE_SHIFT бит. Для большинства применений поле vm_pgoff структуры VMA содержит в точности необходимое вам значение. Функция оказывает влияние на физические адреса от (pfn<<PAGE_SHIFT) до (pfn<<PAGE_SHIFT)+size.

size

Размер в байтах области для переназначения.

prot

"Защита", требуемая для новой VMA. Драйвер может (и должен) использовать значение, находящееся в vma->vm_page_prot.

Аргументы для remap_pfn_range довольно просты и большинство из них уже вам предоставлены в VMA при вызове вашего метода mmap. Однако, вам может быть интересно, почему есть две функции. Первая (remap_pfn_range) предназначена для ситуаций, когда pfn ссылается на фактическое ОЗУ системы, а io_remap_page_range следует использовать, когда phys_addr указывает на память ввода/вывода. На практике эти две функции идентичны на всех архитектурах, кроме SPARC, и в большинстве ситуаций вы увидите использование remap_pfn_range. Однако, в интересах написания переносимых драйверов, вы должны использовать тот вариант remap_pfn_range, который подходит для вашей индивидуальной ситуации.

Другая сложность связана с кэшированием: как правило, ссылки на память устройства не следует кэшировать процессором. Часто системная BIOS настраивает всё должным образом, но также возможно запретить кэширование определённых VMA через поле защиты. К сожалению, отключение кэширования на этом уровне весьма зависимо от процессора.

Отображение памяти с помощью nopage

Хотя remap_pfn_range работает хорошо во многих, если не в большинстве, реализаций в драйвере mmap, иногда бывает необходимо быть немного более гибким. Втаких ситуациях может быть востребована реализация с использованием метода VMA nopage.

Ситуация, в которой подход nopage является полезным, может быть создана системным вызовом mremap, который используется приложениями для изменения границ адресов отображаемого региона.

Случается, что ядро не уведомляет драйверы непосредственно, когда отображённая VMA изменяется с помощью mremap. Если VMA уменьшается в размерах, ядро может тихо избавиться от ненужных страниц не уведомляя драйвер. Если, наоборот, VMA расширяется, драйвер в конце концов узнает об этом через вызовы nopage, когда для новых страниц потребуется создать отображение, поэтому нет необходимости выполнять отдельное уведомление. Поэтому, если вы хотите поддерживать системный вызов mremap, должен быть реализован метод nopage.

Напомним, что метод nopage имеет следующий прототип:

```
struct page *(*nopage)(struct vm_area_struct *vma,
unsigned long address, int *type);
```

Когда пользовательский процесс пытается получить доступ к странице в VMA, которая не присутствует в памяти, вызывается связанная с ней функция nopage. Параметр address содержит виртуальный адрес, который вызвал ошибку, округлённый вниз к началу страницы. Функция nopage должна найти и вернуть указатель struct page, который относится к той странице, которую хочет пользователь. Эта функция также должна заботиться об увеличении счётчика использования для страницы, которую она возвращает, вызывая макрос get_page:

```
get_page(struct page *pageptr);
```

Этот шаг необходим, чтобы на отображённых страницах сохранить счётчики ссылок верными. Ядро поддерживает этот счётчик для каждой страницы; когда счётчик уменьшается до 0, ядро знает, что страница может быть помещена в список свободных страниц. Когда VMA становится неотображённой, ядро уменьшает счётчик использования для каждой страницы в этой области. Если драйвер не увеличил счётчик при добавлении страницы в эту область, счётчик использования станет 0 преждевременно и целостность системы нарушится.

Методу nopage следует также хранить тип ошибки в месте, указываемом аргументом type, но только если этот аргумент не NULL. В драйверах устройств подходящим значением для type неизменно будет VM_FAULT_MINOR.