Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Прямой доступ к памяти (DMA)»

Студент _____ИУ7-75Б_____ _____ _____С. А. Мирзоян_____
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ _____Н. Ю. Рязанова_____
(Подпись, дата) (И.О.Фамилия)

2020 г.

Оглавление

Введение

При работе с периферией независимо от наличия или отсутствия у центрального процессора ввода-вывода, отображаемого на пространство памяти, ему необходимо обращаться к контроллерам устройств, чтобы осуществлять с ними обмен данными. Центральный процессор может запрашивать данные у контроллера ввода-вывода побайтно, однако при этом будет нерационально расходоваться его рабочее время. В качестве решения данной проблемы можно воспользоваться схемой, называемой прямым доступом к памяти (Direct Memory Access (DMA)).

Цель данной работы – реализовать загружаемый модуль ядра, который имитирует работу DMA. Рассмотреть способы распределения памяти при работе с современными и устаревшими устройствами.

1. Аналитическая часть

1.1 Ядро Linux

Ядро Linux относится к категории так называемых *монолитных* – это означает, что большая часть функциональности операционной системы называется *ядром* и запускается в привилегированном режиме. Можно было бы подумать, что ядро Linux очень статично, но на действительности это не так.

Ядро Linux *динамически изменяемое* – это означает, что можно загружать в ядро дополнительную функциональность, выгружать функции из ядра и даже добавлять новые модули, использующие другие модули ядра. Преимущество загружаемых модулей заключается в возможности сократить расход памяти для ядра, загружая только необходимые модули.

1.1.1 Анатомия модуля ядра

Загружаемые модули ядра имеют ряд фундаментальных отличий от элементов, интегрированных непосредственно в ядро, а также от обычных программ. Обычная программа содержит главную процедуру (main) в отличие от загружаемого модуля, содержащего функции входа и выхода (в версии 2.6 эти функции можно именовать как угодно). Функция входа вызывается, когда модуль загружается в ядро, а функция выхода – соответственно при выгрузке из ядра. Поскольку функции входа и выхода являются пользовательскими, для указания назначения этих функций используются макросы *module_init* и *module_exit*. Загружаемый модуль содержит также набор обязательных и дополнительных макросов. Они определяют тип лицензии, автора и описание модуля, а также другие параметры. Пример очень простого загружаемого модуля приведен на рисунке 1.

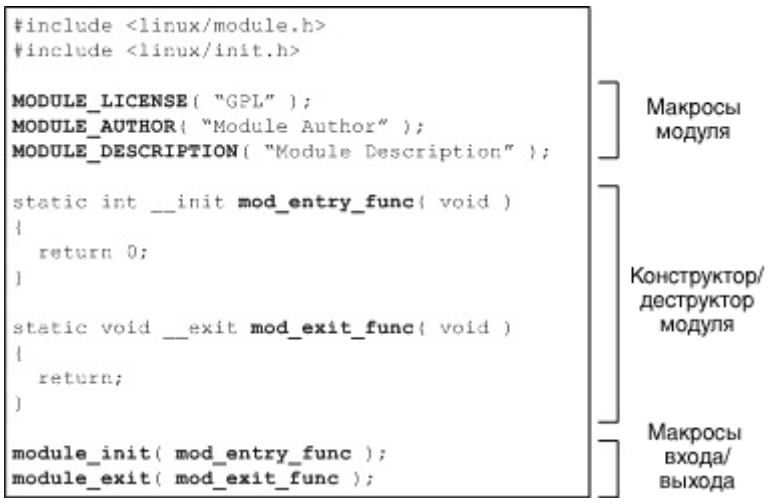

Рисунок 1.1. Код простого загружаемого модуля.

Версия 2.6 ядра Linux предоставляет новый, более простой метод создания загружаемых модулей. После того как модуль создан, можно использовать обычные пользовательские инструменты для управления модулями (несмотря на изменения внутреннего устройства):

- insmod (устанавливает модуль),
- rmmod (удаляет модуль),
- modprobe (контейнер для insmod и rmmod),
- depmod (для создания зависимостей между модулями),
- modinfo (для поиска значений в модулях макроса).

1.1.2 Анатомия объектного кода модуля ядра

Загружаемый модуль представляет собой просто специальный объектный файл в формате ELF (Executable and Linkable Format). Обычно объектные файлы обрабатываются компоновщиком, который разрешает символы и формирует исполняемый файл. Однако в связи с тем, что загружаемый модуль не может разрешить символы до загрузки в ядро, он остается ELF-объектом. Для работы с загружаемыми модулями можно использовать стандартные средства работы с объектными файлами (которые в версии 2.6 имеют суффикс *.ko,* от kernel object). данные).

В модуле также обнаружатся дополнительные разделы, ответственные за поддержку его динамического поведения. Раздел .init.text содержит код `module_init`, а раздел .exit.text – код `module_exit` code (рисунок 2). Раздел .modinfo содержит тексты макросов, указывающие тип лицензии, автора, описание и т. д.

Рисунок 1.2. Пример загружаемого модуля с разделами ELF

1.1.3 Жизненный цикл загружаемого модуля ядра

Процесс загрузки модуля начинается в пользовательском пространстве с команды `insmod` (вставить модуль). Команда `insmod` определяет модуль для загрузки и выполняет системный вызов уровня пользователя `init_module` для начала процесса загрузки. Команда `insmod` для ядра версии 2.6 стала чрезвычайно простой (70 строк кода) за счет переноса части работы в ядро. Команда `insmod` не выполняет никаких действий по разрешению символов (вместе с командой `kerneld`), а просто копирует двоичный код модуля в ядро при помощи функции `init_module`; остальное делает само ядро.

Функция `init_module` работает на уровне системных вызовов и вызывает функцию ядра `sys_init_module` (рисунок 3). Это основная функция для

загрузки модуля, обращающаяся к нескольким другим функциям для решения специальных задач. Аналогичным образом команда `rmmod` выполняет системный вызов функции `delete_module`, которая обращается в ядро с вызовом `sys_delete_module` для удаления модуля из ядра.

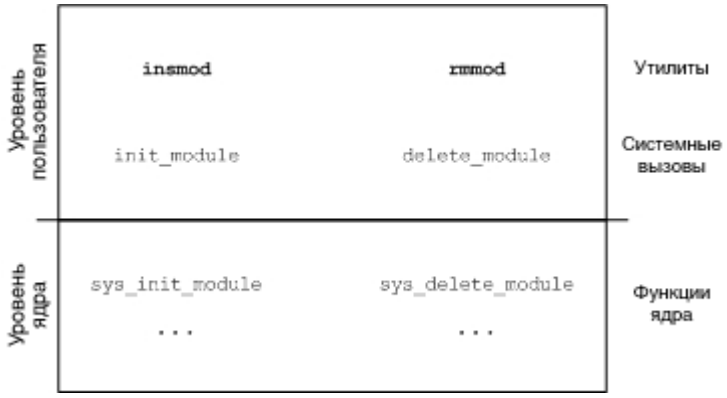

Рисунок 1.3. Основные команды и функции, участвующие в загрузке и выгрузке модуля

Во время загрузки и выгрузки модуля подсистема модулей поддерживает простой набор переменных состояния для обозначения статуса модуля. При загрузке модуля он имеет статус `MODULE_STATE_COMING`. Если модуль загружен и доступен, его статус – `MODULE_STATE_LIVE`. Если модуль выгружен – `MODULE_STATE_GOING`.

1.1.4 Подробности загрузки модуля

Теперь взглянем на внутренние функции для загрузки модуля (рисунок 4). При вызове функции ядра `sys_init_module` сначала выполняется проверка того, имеет ли вызывающий соответствующие разрешения (при помощи функции `capable`). Затем вызывается функция `load_module`, которая выполняет механическую работу по размещению модуля в ядре и производит необходимые операции. Функция `load_module` возвращает ссылку, которая указывает на только что загруженный модуль. Затем он вносится в двусвязный список всех модулей в системе, и все потоки, ожидающие изменения состояния

модуля, уведомляются при помощи специального списка. В конце вызывается функция `init()` и статус модуля обновляется, чтобы указать, что он загружен и доступен.

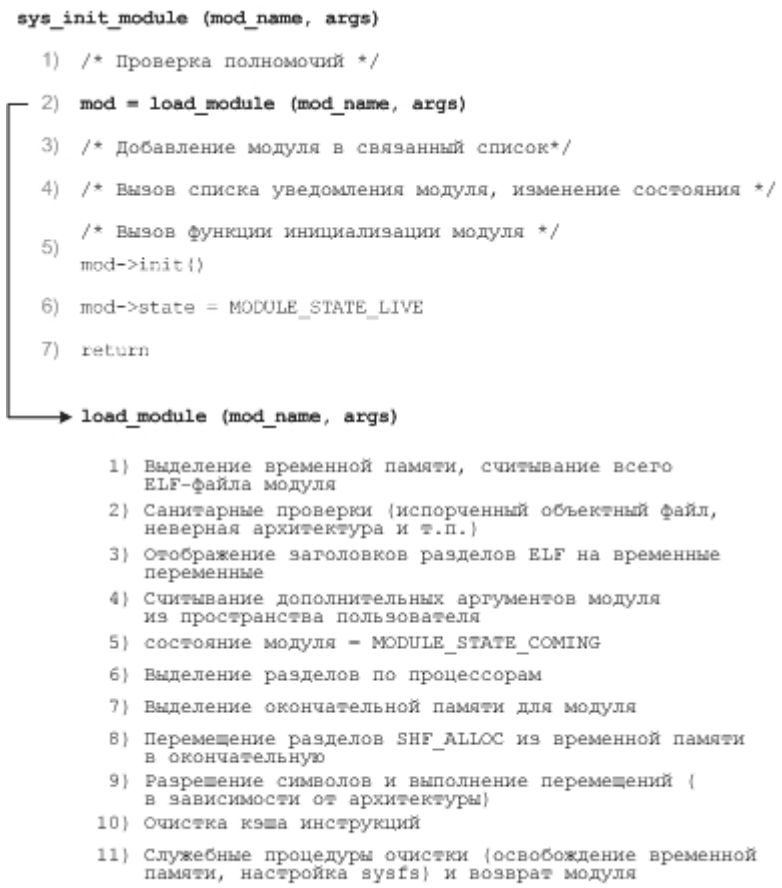

Рисунок 4. Внутренний процесс загрузки модуля (упрощенно)

Внутренние процессы загрузки модуля представляют собой анализ и управление модулями ELF. Функция `load_module` (которая находится в ./linux/kernel/module.c) начинает с выделения блока временной памяти для хранения всего модуля ELF. Затем модуль ELF считывается из пользовательского пространства во временную память при помощи `copy_from_user`. Являясь объектом ELF, этот файл имеет очень специфичную структуру, которая легко поддается анализу и проверке.

Следующим шагом является ряд "санитарных проверок" загруженного образа. После того как проверка пройдена, образ ELF анализируется и создается набор

вспомогательных переменных для заголовка каждого раздела, чтобы облегчить дальнейший доступ к ним. Поскольку базовый адрес объектного файла ELF равен 0 (до перемещения), эти переменные включают соответствующие смещения в блок временной памяти. Во время создания вспомогательных переменных также проверяются заголовки разделов ELF, чтобы убедиться, что загружаемый модуль корректен.

Дополнительные параметры модуля, если они есть, загружаются из пользовательского пространства в другой выделенный блок памяти ядра (шаг 4), и статус модуля обновляется, чтобы обозначить, что он загружен (`MODULE_STATE_COMING`). Если необходимы данные для процессоров (согласно результатам проверки заголовков разделов), для них выделяется отдельный блок.

В предыдущих шагах разделы модуля загружались в память ядра (временную), и было известно, какие из них используются постоянно, а какие могут быть удалены. На следующем шаге для модуля в памяти выделяется окончательное расположение, и в него перемещаются необходимые разделы (обозначенные в заголовках `SHF_ALLOC` или расположенные в памяти во время выполнения). Затем производится дополнительное выделение памяти размера, необходимого для требуемых разделов модуля. Производится проход по всем разделам во временном блоке ELF,, и те из них, которые необходимы для выполнения, копируются в новый блок. Затем следуют некоторые служебные процедуры. Также происходит разрешение символов, как расположенных в ядре (включенных в образ ядра при компиляции), так и временных (экспортированных из других модулей).

Затем производится проход по оставшимся разделам и выполняются перемещения. Этот шаг зависит от архитектуры и соответственно основывается на вспомогательных функциях, определенных для данной архитектуры (./linux/arch/<arch>/kernel/module.c). В конце очищается кэш инструкций

(поскольку использовались временные разделы .text), выполняется еще несколько служебных процедур (очистка памяти временного модуля, настройка sysfs) и, в итоге, модуль возвращает `load_module`.

1.1.5 Подробности выгрузки модуля

Выгрузка модуля фактически представляет собой зеркальное отражение процесса загрузки за исключением того, что для безопасного удаления модуля необходимо выполнить несколько "санитарных проверок". Выгрузка модуля начинается в пользовательском пространстве с выполнения команды `rmmod` (удалить модуль). Внутри команды `rmmod` выполняется системный вызов `delete_module`, который в конечном счете инициирует `sys_delete_module` внутри ядра (рисунок 3). Основные операции удаления модуля показаны на рисунке 5.

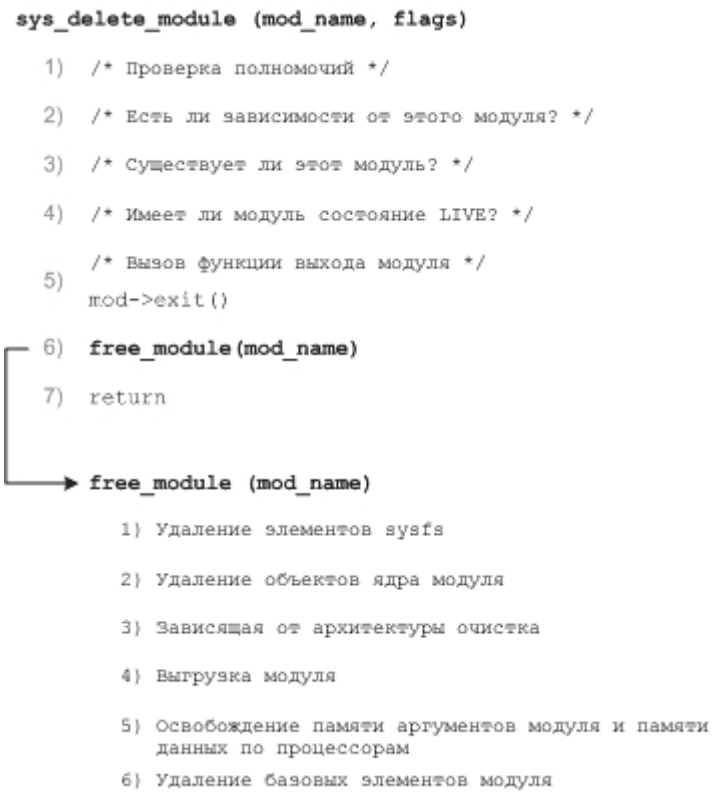

Рисунок 5. Внутренний процесс выгрузки модуля (упрощенно).

При вызове функции ядра `sys_delete_module` (с именем удаляемого модуля в качестве параметра) сначала выполняется проверка того, имеет ли вызывающий соответствующие разрешения. Затем по списку проверяются зависимости других модулей от данного модуля. При этом используется список `modules_which_use_me`, содержащий по элементу для каждого зависимого модуля. Если список пуст, т.е. зависимостей не обнаружено, то модуль становится кандидатом на удаление (иначе возвращается ошибка). Затем проверяется, загружен ли модуль. Ничто не запрещает пользователю запустить команду `rmmod` для модуля, который в данный момент устанавливается, поэтому данная процедура проверяет, активен ли модуль. После нескольких дополнительных служебных проверок предпоследним шагом вызывается функция выхода данного модуля (предоставляемая самим модулем). В заключение вызывается функция `free_module`.

К моменту вызова `free_module` уже известно, что модуль может быть безопасно удален. Зависимостей не обнаружено, и для данного модуля можно начать процесс очистки ядра. Этот процесс начинается с удаления модуля из различных списков, в которые он был помещен во время установки (sysfs, список модулей и т.д.). Потом инициируется команда очистки, зависящая от архитектуры (она расположена в ./linux/arch/<arch>/kernel/module.c). Затем обрабатываются зависимые модули, и данный модуль удаляется из их списков. В конце, когда с точки зрения ядра очистка завершена, освобождаются различные области памяти, выделенные для модуля, в том числе память для параметров, память для данных по процессорам и память модуля ELF (`core` и `init`).[5]

1.2 Принцип работы прямого доступа к памяти

Предположим, что центральный процессор (ЦП) обращается ко всем устройствам и к памяти посредством единой системной шины, соединяющей центральный процессор, память и устройства ввода-вывода (рис. 1).

Операционная система может использовать DMA только при наличии аппаратного DMA-контроллера, присутствующего у большинства систем. Иногда этот контроллер встроен в контроллеры дисков и другие контроллеры, но такая конструкция требует отдельного DMA-контроллера для каждого устройства. Чаще всего для упорядочения обмена данными с несколькими устройствами, проводимого нередко в параллельном режиме, доступен только один DMA-контроллер (размещенный, к примеру, на системной плате).

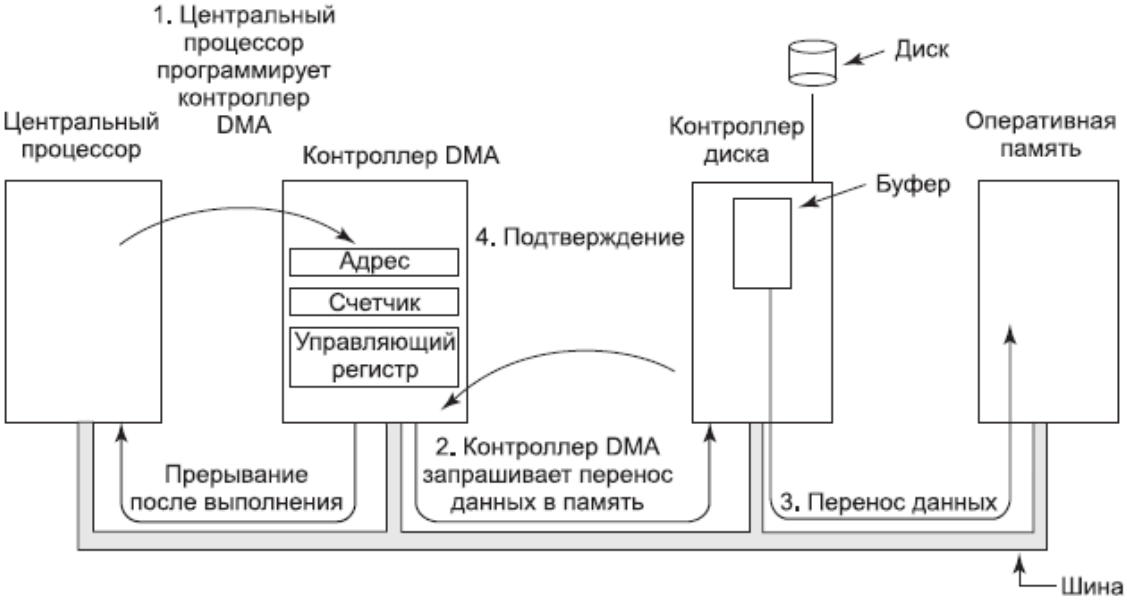

Рисунок 1.3. Операции, осуществляемые при передаче данных с использованием DMA [1]

На рисунке 1 можно наблюдать следующее. DMA-контроллер он имеет доступ к системной шине независимо от центрального процессора. В нем имеется несколько регистров. Регистры контроллера DMA доступны ЦП для чтения и записи, а также используются для задания параметров передачи:

Регистры DMA-контроллера:

- Адрес памяти
- Счетчик байтов
- Регистры управления
    - номер порта, который должен быть использован для передачи данных;
    - вид операции (чтение или запись);
    - единица переноса (побайтно или пословно);
    - размера данных, которые следует перенести, в байтах.

1.3 Чтение диска

Рассмотрим чтение диска без использования DMA. Сначала контроллер диска последовательно побитно считывает блок (один или несколько секторов) с диска, пока весь блок не окажется во внутреннем буфере контроллера. Затем он вычисляет контрольную сумму, чтобы убедиться в отсутствии ошибок чтения. Затем контроллер инициирует прерывание. Когда операционная система приступает к работе, она может в цикле побайтно или пословно считать дисковый блок из буфера контроллера, считывая при каждом проходе цикла один байт или слово из регистра контроллера устройства и сохраняя его в оперативной памяти.

При использовании DMA все происходит по-другому. Существует два похода к передаче данных: программное обеспечение запрашивает данные или аппаратное обеспечение асинхронно передает данные в систему.

1.3.1 Запрашивание данных.

Сначала центральный процессор программирует DMA-контроллер, устанавливая значения его регистров таким образом, чтобы он знал, что и куда нужно передать (шаг 1 на рис. 1). Он также выдает команду контроллеру диска на чтение данных с диска во внутренний буфер контроллера и на проверку контрольной суммы. После того как в буфере контроллера окажутся достоверные данные, к работе может приступать DMA.

DMA-контроллер инициирует передачу данных, выдавая по шине контроллеру диска запрос на чтение (шаг 2). Этот запрос на чтение выглядит так же, как и любой другой запрос на чтение, и контроллер диска не знает и даже не интересуется, откуда он пришел — от центрального процессора или от DMA-контроллера. Обычно адрес памяти, куда нужно вести запись, выставлен на адресных линиях шины, поэтому, когда контроллер диска извлекает очередное слово из своего внутреннего буфера, он знает, куда его следует записать. Запись в память — это еще один стандартный цикл шины (шаг 3). Когда запись завершается, контроллер диска также по шине посылает подтверждающий сигнал DMA-контроллеру (шаг 4). Затем DMA-контроллер дает приращение используемому адресу памяти и уменьшает значение счетчика байтов. Если счетчик байтов все еще больше нуля, то шаги со 2-го по 4-й повторяются до тех пор, пока значение счетчика не станет равно нулю. Как только это произойдет, DMA-контроллер выставляет прерывание, чтобы центральный процессор узнал о завершении передачи данных. И когда к работе приступает операционная система, ей уже не нужно копировать дисковый блок в память, потому что он уже там.

1.3.2 Аппаратная асинхронная передача

Второй случай возникает, когда DMA используется асинхронно. Это происходит, например, с устройствами сбора данных, которые продолжают передавать данные, даже если их никто не читает. В этом случае драйвер должен

поддерживать буфер таким образом, чтобы последующий вызов чтения возвращал все накопленные данные в пространство пользователя. Шаги, связанные с этим видом передачи, немного отличаются.:

1. Устройство вызывает прерывание, чтобы объявить о поступлении новых данных.
2. Обработчик прерываний выделяет буфер и сообщает аппаратному обеспечению, куда передавать свои данные.
3. Периферийное устройство записывает данные в буфер и вызывает другое прерывание, когда это сделано.
4. Обработчик отправляет новые данные, будит любой соответствующий процесс и берет машину е на себя ведение домашнего хозяйства.

Этапы обработки во всех этих случаях подчеркивают, что эффективная обработка DMA зависит от отчетов о прерываниях. Хотя и можно реализовать DMA с помощью драйвера опроса, это не имеет смысла, потому что драйвер опроса будет тратить впустую преимущества производительности, которые предлагает DMA по сравнению с более простым процессорным вводом-выводом.

Еще один важный элемент, представленный здесь, - это буфер DMA. Чтобы использовать прямой доступ к памяти, драйвер устройства должен иметь возможность выделить один или несколько специальных буферов, подходящих для DMA. Нужно обратить внимание на то, что многие драйверы выделяют свои буферы во время инициализации и используют их до завершения работы.[2]

1.4 Выделение буфера DMA

В этом разделе рассматривается распределение буферов DMA на низком уровне.

Основная проблема с буфером DMA заключается в том, что, когда он больше одной страницы, он должен занимать смежные страницы в физической памяти,

поскольку устройство передает данные с помощью системной шины ISA или PCI, обе из которых несут физические адреса. Интересно отметить, что это ограничение не распространяется на SBUS, который использует виртуальные адреса на периферийной шине. Некоторые архитектуры также могут использовать виртуальные адреса на шине PCI, но портативный драйвер не может рассчитывать на такую возможность.

Хотя буферы DMA могут быть выделены либо при загрузке системы, либо во время выполнения, модули могут выделять свои буферы только во время выполнения. Во время загрузки выделения говорили о выделении при загрузке системы, в то время как ‘реальная история то резервируется память вызовом kmalloc’ и ‘get_free_page и друзья’ описано распределение во времени выполнения. Авторы драйверов должны позаботиться о том, чтобы выделить правильный тип памяти, когда она будет использоваться для операций DMA—не все зоны памяти подходят. В частности, высокая память не будет работать для DMA в большинстве систем—периферийные устройства просто не могут работать с такими высокими адресами.

Буфера DMA могут выделяться только в строго определённых областях памяти:

- эта память должна распределяться в физически непрерывной области памяти, поэтому выделение посредством vmalloc() неприменимо, память под буфера должна выделяться kmalloc() или __get_free_pages();
- для многих архитектур выделение памяти должно быть специфицировано с флагом GFP_DMA, для x86 PCI устройств это будет выделение ниже адреса MAX_DMA_ADDRESS=16MB;
- память должна выделяться, начиная с границы страницы физической памяти, и в объёме целых страниц физической памяти; Для распределения памяти под буфера DMA предоставляются несколько альтернативных групп API (в зависимости от того, что мы хотим получить), их реализации

полностью архитектурно зависимы, но вызовы создают уровень абстракций:

1.5 Выводы из аналитического раздела

Выделив любым подходящим способом блок памяти для обмена по DMA, драйвер выполняет последовательность операций (обычно это проделывается в цикле, в чём и состоит работа драйвера):

- адрес начала блока записывается в соответствующий регистр одной из 6-ти областей ввода-вывода PCI устройства, как обсуждалось выше — конкретные адреса таких регистров здесь и далее определяются исключительно спецификацией устройства...
- ещё в один специфический регистр заносится длина блока для обмена...
- наконец, в регистр команды заносится значение (чаще это выделенный бит) команды начала операции по DMA...
- когда внешнее PCI устройство сочтёт, что оно готово приступить к выполнению этой операции, оно аппаратно захватывает шину PCI, и под собственным управлением записывает (считывает) указанный блок данных...
- по завершению выполнения операции устройство освобождает шину PCI под управление процессора, и извещает систему прерыванием по выделенной устройству линии IRQ о завершении операции.

Из сказанного выше легко понять, что принципиальной операцией при организации DMA-обмена в модуле является только создание буфера DMA, всё остальное должно исполнять периферийное устройство.[3]

2. Конструкторская часть

В данном разделе будут рассмотрены требования к программе и методы реализации загружаемого модуля ядра.

2.1 Требования к программе

Организация обмена по DMA это основной способ взаимодействия со всеми высокопроизводительными устройствами. С другой стороны, обмен по DMA полностью зависим от деталей аппаратной реализации, поэтому в общем виде может быть рассмотрен только достаточно поверхностно.

Программа должна:

- В своем составе иметь загружаемый модуль ядра, имитирующий драйвер прямого доступа к памяти.
- В реализации должны быть представлены следующие способы распределения памяти под буфера DMA:
    - Для новых API (версии ядра Linux старше 2.4)
        - Когерентное распределение
        - Потоковое распределение
        - Пул
    - Для устаревших API (PCI-специфичные интерфейсы)
        - Последовательное распределение
        - Одиночное распределение

2.2 Используемые структуры и модули

<linux/module.h> - Динамическая загрузка модулей в ядро.

<linux/pci.h> - определение PCI и прототипы функций

<linux/slab.h> - Распределение памяти

<linux/dma-mapping.h> - Список возможных атрибутов, связанных с отображением DMA.

#include <linux/dmapool.h>

Из <limux/device.h>

```
struct device {
    ...
    struct list_head    dma_pools;  /* dma pools (if dma'ble) */
    ...
};
```

Из <linux/dma-mapping.h>[4]

==========

DMA_NONE	без направления (используется для отладки)
DMA_TO_DEVICE	данные идут из памяти на устройство
DMA_FROM_DEVICE	данные поступают с устройства в память
DMA_BIDIRECTIONAL	данные идут в обоих направлениях

==========

Из <linux/slab.h>

1. GFP_KERNEL - выделить нормальную память ядра. Может блокироваться.

2.3 Способы распределения памяти под буфера DMA.

Когерентное распределение:

```
void *dma_alloc_coherent( struct device *dev, size_t size,
                dma_addr_t *dma_handle, gfp_t flag );
```

Эта процедура выделяет область постоянной памяти размером <размер> байт.

Он возвращает указатель на выделенную область (в виртуальном адресном пространстве) или NULL, если выделение не удалось.

Он также возвращает <dma_handle>, который может быть приведен к целому числу без знака той же ширины, что и шина, и передается устройству как база адресов DMA.

Примечание: постоянная память может быть дорогой на некоторых платформах, и минимальная длина выделения может достигать размера страницы, поэтому следует максимально консолидировать запросы на постоянную память.

Самый простой способ сделать это - использовать вызовы dma_pool (см. Ниже).

Параметр flag (только dma_alloc_coherent ()) позволяет вызывающему указать флаги ***GFP_*** (в случае данного проекта ***GFP_KERNEL***) для выделения памяти (реализация может игнорировать флаги, которые влияют на расположение возвращаемой памяти, например, GFP_DMA).

```
1.void *dma_alloc_coherent( struct device *dev, size_t size,
2.                dma_addr_t *dma_handle, gfp_t flag );
```

Освобождение ранее выделенной области постоянной памяти. dev, size и dma_handle должны быть такими же, как переданные в dma_alloc_coherent. Следует обратить внимание на то, что в отличие от вызовов выделения ресурсов одного уровня, эти процедуры может быть вызван только с включенными IRQ.

Потоковое распределение:

```
1.dma_addr_t dma_map_single( struct device *dev, void *ptr, size_t size,
2.                    enum dma_data_direction direction );
3.void dma_unmap_single( struct device *dev, dma_addr_t dma_hand
```

Применяется для выделения буфера под однократные операции DMA, где direction это планируемое направление передачи данных.

Пул:

Используется для выделения буферов DMA небольшого размера, так как dma_alloc_coherent() допускает минимальное выделение только в одну физическую страницу (как видно из прототипов, пул struct dma_pool должен быть сначала создан вызовом dma_pool_create(), и только затем из него производятся выделения dma_pool_alloc());

```
1.struct dma_pool *dma_pool_create( const char *name, struct device *dev,
2. size_t size, size_t align, size_t allocation );
3.void dma_pool_destroy( struct dma_pool *pool );
4.void *dma_pool_alloc( struct dma_pool *pool, gfp_t mem_flags, dma_addr_t
   *handle );
5.void dma_pool_free( struct dma_pool *pool, void *vaddr, dma_addr_t
   handle );
```

Устаревший API:

Устаревший API, впервые появившийся в версии ядра 2.4, содержит две пары вызовов, аналогичных первому и второму API. Утверждается, что новые интерфейсы не зависят от типа аппаратной шины (в перспективе на новые расширения), а устаревший API предназначен исключительно для работы с PCI-шиной.

```
1.void *pci_alloc_consistent( struct device *dev, size_t size, dma_addr_t
2.                dma_addr_t *dma_handle );
3.void pci_free_consistent( struct device *dev, size_t size,
4.                void *vaddr, dma_addr_t dma_handle );
5.
6.dma_addr_t pci_map_single( struct device *dev, void *ptr, size_t size,
7.                int direction );
8.void pci_unmap_single( struct device *dev, dma_addr_t dma_handle,
9.                size_t size, int direction );
```

Выделив любым подходящим способом блок памяти для обмена по DMA, драйвер в цикле выполняет следующую последовательность операций:

1. адрес начала блока записывается в соответствующий регистр одной из 6-ти областей ввода-вывода PCI устройства (конкретные адреса таких регистров определяются спецификацией устройства);

2. ещё в один специфический регистр заносится длина блока для обмена;

3. в регистр команды заносится значение (чаще это выделенный бит) команды начала операции по DMA;

4. когда внешнее PCI-устройство сочтёт, что оно готово приступить к выполнению этой операции, оно аппаратно захватывает шину PCI, и под собственным управлением записывает (считывает) указанный блок данных.

Завершив выполнение операции устройство освобождает шину PCI под управление процессора, и извещает систему прерыванием по выделенной устройству линии IRQ о завершении операции

3. Технологическая часть

3.1 Выбор языка программирования и среды разработки

- Операционная система Linux версии 14.04, версия ядра 3.13.0-32-generic
- В качестве языка программирования был выбран C т.к. он лежит в основе всей операционной системы UNIX/GNU/Linux.
    - Компилируемый язык со статической типизацией.
    - Сочетание высокоуровневых и низкоуровневых средств.
    - Наличие удобной стандартной библиотеки шаблонов

4. Экспериментальная часть

В данном разделе будет проведена апробация реализованной программы для проверки корректности ее работы.

4.1 Цель эксперимента

Целью эксперимента является проверка правильности выполнения поставленной задачи.

4.2 Апробация

На рисунках 4.1 и 4.2 представлены результаты работы прямого доступа к данным для современных устройств и для старых PCI-шин соответственно.

```
[  396.438828] => kbuf=    c0150000, handle=      150000, size = 40960
[  396.438833] => (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
[  396.438835] => string written was, This is the dma_alloc_coherent() string
[  396.438839] => kbuf=    ecaf0000, handle=    2caf0000, size = 40960
[  396.438842] => (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
[  396.438845] => string written was, This is the dma_map_single() string
[  396.438850] => kbuf=    c012b000, handle=      12b000, size = 40960
[  396.438852] => (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
[  396.438855] => string written was, This is the dma_pool_alloc() string
[  597.400051] => kbuf=    c0150000, handle=      150000, size = 40960
[  597.400056] => (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
```

Рисунок 4.1. Результат работы для современных устройств

```
[  597.400051] => kbuf=    c0150000, handle=      150000, size = 40960
[  597.400056] => (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
[  597.400058] => string written was, This is the pci_alloc_consistent() string
[  597.400062] => kbuf=    eec40000, handle=    2ec40000, size = 40960
[  597.400065] => (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
[  597.400068] => string written was, This is the pci_map_single() string
sergey@sergey-VirtualBox:~/dmaS
```

Рисунок 4.2. Результат работы для современных устройств

Как можно заметить на данных рисунках, создание буферов, выделение под них памяти и прочее работает, как и ожидалось, программное обеспечение исполняет свою задачу верно.

Данные примеры интересны не только своими результатами, но и тем, что могут быть полезными как для начального обучения, так и быть шаблонами для более специализированной и глубокой разработки DMA обменов.

Заключение

Во время выполнения курсового проекта была изучены принципы работы загружаемых модулей ядра Linux, принцип работы прямого доступа к памяти (DMA). Особое внимание было уделено способам распределения памяти под буфера DMA, были проанализированы их достоинства и недостатки. Было разработано программное обеспечение, демонстрирующее принципы построения DMA модуля.

Программа реализована в виде шаблона, который можно было бы использовать как в образовательном формате, так и для дальнейшей, уже более специализированной, разработки.

Список использованной литературы

1. Э. Таненбаум, Современные операционные системы, 2015.
2. Alessandro Rubin, Linux Device Drivers 2nd Edition, June 2001.
3. Олег Цилюрик, Программирование модулей ядра Linux, 20.01.2015, 312 с.
4. Документация Linux, James E.J. Bottomley, электронный ресурс, https://www.kernel.org/doc/Documentation/DMA-API.txt
5. Джонс, М. Анатомия загружаемых модулей ядра Linux / М. Джонс. — https://www.ibm.com/developerworks/ru/library/l-lkm/index.html, 2008.

Приложения

Приложение А (Листинг DMA для современных устройств)

```
1.#include <linux/module.h>                          //Динамическая загрузка
   модулей в ядро.
2.#include <linux/pci.h>                             //определение PCI и
   прототипы функций
3.#include <linux/slab.h>                            //Распределение памяти
4.#include <linux/dma-mapping.h>                    //Список возможных атрибутов,
   связанных с отображением DMA.
5.#include <linux/dmapool.h>
6.#include "out.c"
7.MODULE_LICENSE( "GPL" );
8.#define pool_size 1024
9.#define pool_align 8
10.
11.     /* обмен данными будет осуществляться в обоих направлениях */
12.     static int direction = PCI_DMA_BIDIRECTIONAL;
13.     // int direction = PCI_DMA_TODEVICE ;        //Обмен данными в
   направлении устройства
14.     // int direction = PCI_DMA_FROMDEVICE ;      //Обмен данными в
   направлении от устройства
15.     //int direction = PCI_DMA_NONE;             //Блокировка обмена
   данными
16.
17.     static int __init my_init( void )
18.     {
19.         char *kbuf;                             //буфер DMA
20.
21.         dma_addr_t handle;                      /* Dma_addr_t может
   содержать любой действительный адрес DMA или шины для платформы. Оно
   может
22.                                                  * передаваться
   устройству для использования в качестве источника или цели DMA. Это
   характерно для
23.                                                  * данное устройство и
   может быть передано между физическим адресом ЦП и адресным пространством
   шины
24.
```

```
25.                                                     dma-
   mapping.h :#define DMA_MAPPING_ERROR        (~(dma_addr_t)0)
26.                                                 */
27.
28.         size_t size = ( 10 * PAGE_SIZE );       //Размер страницы
29.         struct dma_pool *mypool;                //devic.h :
   dma_pools:  Dma pools (if dma'ble device).
30.
31.         /* использование метода dma_alloc_coherent */
32.         kbuf = dma_alloc_coherent( NULL, size, &handle, GFP_KERNEL );
33.         output( kbuf, handle, size, "This is the dma_alloc_coherent()
   string" );
34.         dma_free_coherent( NULL, size, kbuf, handle );
35.
36.         /* использование метода dma_map/unmap_single */
37.         kbuf = kmalloc( size,
   GFP_KERNEL );                                   // GFP_KERNEL (__GFP__WAIT
   | __GFP__IO | __GFP__FS) - выделение производится от имени процесса,
38.
       //который выполняет системный запрос в пространстве ядра — такой
   запрос может быть временно переводиться
39.
       //в пассивное состояние (блокирован).
40.
       //slab.h
41.         handle = dma_map_single( NULL, kbuf, size, direction );
42.         output( kbuf, handle, size, "This is the dma_map_single()
   string" );
43.         dma_unmap_single( NULL, handle, size, direction );
44.         kfree( kbuf );
45.
46.         /* использование метода dma_pool */
47.         mypool = dma_pool_create( "mypool", NULL, pool_size, pool_align,
   0 );
48.         kbuf = dma_pool_alloc( mypool, GFP_KERNEL, &handle );
49.         output( kbuf, handle, size, "This is the dma_pool_alloc()
   string" );
50.         dma_pool_free( mypool, kbuf, handle );
51.         dma_pool_destroy( mypool );
52.      return -1;
```

```
53.     }
```

Приложение Б (Листинг DMA для устаревших устройств)

```
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/slab.h>
MODULE_LICENSE( "GPL" );

#include "out.c"

/* обмен данными будет осуществляться в обоих направлениях */
static int direction = PCI_DMA_BIDIRECTIONAL;
     // int direction = PCI_DMA_TODEVICE ;
     // int direction = PCI_DMA_FROMDEVICE ;
     //int direction = PCI_DMA_NONE;


     static int __init my_init( void ) {
      char *kbuf;
      dma_addr_t handle;
      size_t size = ( 10 * PAGE_SIZE );
      /* использование метода pci_alloc_consistent */
      kbuf = pci_alloc_consistent( NULL, size, &handle );
      output( kbuf, handle, size, "This is the pci_alloc_consistent()
string" );
      pci_free_consistent( NULL, size, kbuf, handle );
      /* использование метода pci_map/unmap_single */
      kbuf = kmalloc( size, GFP_KERNEL );
      handle = pci_map_single( NULL, kbuf, size, direction );
      output( kbuf, handle, size, "This is the pci_map_single()
string" );
      pci_unmap_single( NULL, handle, size, direction );
      kfree( kbuf );
      return -1;
     }
```

Приложение В (Листинг: Общая часть тестов)

```
1.static int __init my_init( void );
2.module_init( my_init );
3.
4.MODULE_AUTHOR( "Sergey Mirzoyan" );
5.
6.#define MARK "=> "
7.
8.static void output( char *kbuf, dma_addr_t handle, size_t size, char
   *string ) {
9.   unsigned long diff;
10.        diff = (unsigned long)kbuf - handle;
11.        printk( KERN_INFO MARK "kbuf=%12p, handle=%12p, size = %d\n",
12.                kbuf, (void*)(unsigned long)handle, (int)size );
13.        printk( KERN_INFO MARK "(kbuf-handle)= %12p, %12lu,
   PAGE_OFFSET=%12lu, compare=%lu\n",
14.                (void*)diff, diff, PAGE_OFFSET, diff - PAGE_OFFSET );
15.        strcpy( kbuf, string );
16.        printk( KERN_INFO MARK "string written was, %s\n", kbuf );
17.     }
```