Ядро ОС Linux.

Руководство системного программиста

.

Содержание

.

Введение.

Эта книга вдохновляет вас, начинающих исследователей ядер, не достаточно знающих UNIX-системы, для изучения ядра Linux, когда она впервые появилась у вас и еще тяжела для полного понимания. Это пособие создано для того, чтобы помочь вам быстрее изучить основные концепции и выделить из внутренней структуры Linux то, что может понадобиться вам, чтобы, не читая полностью исходный текст ядра, определить, что же случилось с какой-либо конкретной переменной. Почему Linux ? Linux - это первая свободно доступная система типа UNIX для 386 компьютеров. Она была полностью переписана в уменьшенном объеме так, не имеет большого количества функций, работающих с режимом реального времени, как в других операционных системах (386BSD), и, следовательно, проста в понимании и доступна для изменений.

UNIX появился около 20 лет назад, но только недавно появились столь мощные микрокомпьютеры, способные поддерживать работу операционных систем с многозадачным, многопользовательским защищенным режимом. Кроме того, описания UNIX труднодоступны, лишь документация о внутренностях ядра распространялась свободно. UNIX, кажущийся в начале простым, со временем увеличивался в размерах и превратился в объемную систему, понятную лишь профессионалу. С Linux, однако, мы можем решить часть описанных выше проблем в связи с тем, что:

- У Linux довольно простое ядро с хорошо структурированным интерфейсом;
- Контроль за написанием ядра вел один человек - Linus Torvalds, что не позволило появиться в ядре раздробленным участкам;
- Исходные тексты ядра свободно распространяются, так что начинающие программисты могут свободно понимать и изучать их, становясь выше в собственных глазах.

Мы надеемся, что эта книга поможет начинающим исследователям ядер

разобраться в ядре Linux, поняв его структуру.

Сведения об авторских правах.

Глава 1
" Прежде чем вы начали..."

1.1 Типографские соглашения.

- Выделенный шрифт используется для обозначения определений, предупреждений и ключевых слов в языке.

- Курсив используется для обозначения вставок и введений для новых статей.

- Наклонный шрифт используется для обозначения мета-переменных в тексте, особенно в командной строке:

ls - l [foo]

где [foo] - имя файла,как /bin/cp.Иногда довольно сложно в тексте заметить наклонный шрифт, и соответствующее выражение берется в [---].

- Шрифт печатной машинки используется для отображения ответной информации компьютера:

ls -l /bin/cp

[-rwxr-xr-x 1 root wheel 12104 Sep 25 15:53 /bin/cp]

также он используется для примеров в кодах Си для обозначения системных команд и для описания конфигурационных файлов. Иногда для наглядности эти примеры помещаются в рамку.

- В <---> скобки берется нажатая клавиша:
Для продолжения нажмите .

- Звездочка на полях выделяет место, требующее особого внимания.

1.2 Необходимые знания для изучения книги.

Для того, чтобы понять эту книгу, вы должны хотя бы поверхностно знать Си. Это означает, что вы можете читать программы на Си, не уделяя внимания справочникам. Вы должны уметь писать простейшие программы на Си и понимать структуры, указатели и макросы, атак же прототипы ANSI C.

Вы должны также представлять тексты стандартной библиотеки ввода/вывода, так как стандартные библиотеки не доступны в ядре. Некоторые часто используемые функции ввода/вывода были переписаны внутри ядра и они по возможности описываются далее. Также вам нужен хороший текстовый редактор, перекомпилирование ядра Linux и умение выполнять простейшие задачи системного администратора, такие как включение информации в /dev/.

1.3 Наставление читателю.

В этой части приводятся некоторые полезные при прочтении вещи.

Статические переменные.

Всегда определяйте статические переменные. Масса почти случайных ошибок возникает из - за игнорирования статических переменных. Т.к. ядро на самом деле не является запускаемой программой, сегмент bss не всегда обнуляется в зависимости от метода загрузки.

Невозможность использования libc.

Библиотека libc не доступна в ядре, однако некоторые функции из нее были продублированы. Смотрите разделы книги, в которых эти функции документированы. В основном, это разделы 3 и 9.

/\ Linux - это не UNIX.
\/ Linux - система, написанная не для коммерческого распространения.

Глава 2.

Драйверы устройств.

Что такое драйвер устройства.

Создание драйвера устройства - дело достаточно трудоемкое. Запись на жесткий диск требует помещения определенных цифровых данных в определенное место, ожидания ответа на запрос о готовности жесткого диска, затем аккуратной пересылки информации. Запись на флопповод проходит еще сложнее - нужен постоянный контроль на текущим состоянием дискеты.

Вместо помещения кода каждого отдельного приложения управляющего устройством, вы разделяете код между приложениями. Вам следует защитить этот код от других пользователей и использующих его программ.

Если вы верно сделали это, то вы можете без смены приложений подключать или убирать устройства. Более того, вы должны иметь возможности ОС - загрузить вашу программу в память и запустить ее. Так что ОС, в сущности, - это набор привилегированных, общих и частных функций или функций аппаратного обеспечения низкого уровня,функций работы с памятью и функций контроля.

Все версии UNIX имеет абстрактный способ считывания и записи

на устройство. Действующие устройства представляются в виде файлов, так что одинаковые вызовы (read(), write() и т.п.) могут быть использованы и как устройства и как файлы.

Внутри ядра существует набор функций, отмеченных как файлы, вызываемые при запросе для ввода/вывода на файлы устройств, каждый из которых представляет свое устройство.

Всем устройствам, контролируемым одним драйвером, дается один и тот же основной номер, и различные подномера.

Эта глава описывает, как написать любой из допускаемых в Linux типов драйверов устройств : символьных, блочных, сетевых и драйверов SCSI. Она описывает, какие функции вы должны написать, как инициализировать драйверы и эффективно выделять под них память, какие функции встроены в Linux для упрощения деятельности такого рода.

Создание драйвера устройств для Linux оказывается более простым чем мнится на первый взгляд, ибо оно включает в себя написание новой функции и определение ее в системе переключения файлов(VFS).

Тем самым, когда доступно устройство, присущее вашему драйверу, VFS вызывает вашу функцию.

Однако, вы должны помнить, что драйвер устройства является частью ядра. Это означает, что ваш драйвер запускается на уровне ядра и обладает большими возможностями : записать в любую область памяти, повредить ваш монитор или разбить вам унитаз в случае,

если ваш компьютер управляет сливным баком.

Также ваш драйвер будет запущен в режиме работы с ядром, а ядро Linux, как и большинство ядер UNIX, не имеет средств принудительного сброса. Это означает, что если ваш драйвер будет долго работать, не давая при этом работать другим программам, ваш компьютер может "зависнуть ". Нормальный пользовательский режим с последовательным запуском не обращается к вашему драйверу.

Если вы решили написать драйвер устройства, вы должны внимательно прочитать всю эту главу, однако, нет гарантий, что эта глава не содержит ошибок, и вы не сломаете ваш компьютер, даже если будете следовать всем инструкциям. Единственный совет - сохраняйте информацию перед запуском драйвера.

Драйверы пользовательского уровня.

Не всегда нужно писать драйвер для устройства, особенно если за устройством следит всего одно приложение. Наиболее полезным примером этому является устройство карты памяти, однако вы можете сделать карту памяти с помощью устройств ввода/вывода (доступ к устройствам осуществляется с помощью функций inpb() и outpb()).

Если вы работаете в режиме superuser, вы можете использовать функцию mmap для того, чтобы поместить вашу функцию в какую-то область памяти. С помощью этой процедуры вы сможете весьма просто работать с адресами памяти, как с обычными переменными.

Если ваш драйвер использует прерывание, то вам придется работать внутри ядра, так как не существует других путей для прерываний обычных пользовательских процессов. В проекте DOSEMU однако, есть Простейший Генератор прерываний - SIG, но он работает недостаточно быстро, как это можно было ожидать от последней версии DOSEMU.

Прерывание - это жестко определенная процедура. Также вы при установке своего аппаратного обеспечения вы определяете линию IRQ для физического сигнала прерываний, возникающего, когда устройство обращается к драйверу. Это происходит, когда устройство пересылает или запрашивает информацию, а также при обнаружении каких-либо исключительных ситуаций, о которых должен знать драйвер. Для обработки прерываний в ядре и для обработки сигналов на пользовательском уровне используется одна и та же структура данных - sigaction. Таким образом, где сигналы аппаратных прерываний доставляются ядру точно так же, как

системные сигналы на уровне пользовательского обеспечения.

Если ваш драйвер должен обращаться к нескольким процессам сразу или управлять общими ресурсами, тогда вы должны написать драйвер устройства, и драйвер пользовательского уровня вам не подходит.

2.2.1 Пример - vgalib.

Хорошим примером драйвера пользовательского уровня является библиотека vgalib. Стандартные функции read() и write() не подходят для написания действительно быстрого графического

драйвера, и поэтому существует библиотека функций, которая концептуально работает как драйвер устройства, но на пользовательском уровне. Все функции, которые используют ее, должны запускать setuid, так как она использует системную функцию ioperm(). Функции, которые не запускают setuid, обладают возможностью записи в /DEV/MEM, если у вас есть группы mem или kmem, которые позволяют это, но только корневые процессы могут запускать ioperm().

Есть несколько портов ввода/вывода, относящихся к графике VGA. Vgalib дает им символические имена с помощью #define, и далее используют ioperm() для разрешения функции правильного прочтения и записи в эти порты.

```
if (ioperm(CRT_IC, 1, 1)) {
   printf("VGAlib: can't get I/O permission \n");
   exit(-1);
 }
ioperm(CRT_IM, 1, 1);
ioperm(ATT_IW, 1, 1);
[--]
```

Это требует лишь однократной проверки, так как единственной причиной нефункционирования ioperm() может быть обращение к ней не в статусе superuser или во время смены статуса.

/\
\/ После вызова этой функции разрешается использование inb и outb инструкций, однако лишь с определенными портами. Эти инструкции могут быть доступны без использования прямого ассемблерного кода , но работают они лишь в случае компиляции с параметром optimization on и с ключом -0?. Для более подробных сведений читай .
После обращения в порты ввода вывода vgalib засылает информацию в область ядра следующим образом :

```
/* open /dev/mem */
if ((mem_fd = open("/dev/mem", 0_RDWR) ) < 0) {
    prntf( "VGAlib: can' t open /dev/mem \n");
    exit (-1);
}

/* mmap graphics memory */
if ((graph_mem = malloc(GRAPH*SIZE + (PAGE-SIZE-1))) == NULL) {
    printf( " VGAlib: allocation error \n ");
    exit (-1);
}
if ((unsigned long)graph_mem % PAGE_SIZE)
    graph_mem += PAGE_SIZE - ((unsigned long)graph_mem % PAGE_SIZE);
graph_mem = (unsigned char *)mmap(
    (caddr_t)graph_mem,
    GRAPH_SIZE,
    PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_FIXED,
    mem_fd,
    GRAPH_BASE
);
if ((long)graph_mem < 0) {
    printf(" VGAlib: mmap error \n");
    exit (-1);
}
```

В начале программа открывает /dev/mem, затем выделят достаточное количество памяти для распределения на страницу, затем меняет карту памяти.

GRAPHSIZE - размер памяти vga.
GRAPHBASE - адрес начала памяти VGA в /dev/mem.

Затем, записывая в адрес возвращаемый mmap(), программа осуществляет запись в память экрана.

2.2.2 Пример : Преобразование мыши.

Если вы хотите написать драйвер, работающий так же, как и драйвер на уровне ядра, но не находящийся в его области, то вы можете создать fifo (буфер - first in, first out). Обычно он помещается в директорию /dev (во время нефункционирования) и ведет себя как подключенное устройство.

В частности, это используется когда вы используете мышь типа PS/2 и хотите запустить XFree86. Вы должны создать fifo, называемый /dev/mouse, и запустить программу mconv, которая, читая сигналы мыши PS/2 из /dec/psaux, пишет эквивалентные сигналы microsoft mouse в /dev/mouse.

В этом случае XFree86 будет читать сигналы из /dev/mouse и функционировать также как и при подключенной microsoft mouse.

2.3 Основы драйверов устройств.

Мы будем полагать, что вы не хотите писать драйвер на пользовательском уровне, а желаете работать непосредственно в области ядра.

В таком случае вам придется иметь дело с файлами .c и .h. Мы будем условно обозначать ваши труды как foo.c и foo.h.

2.3.1 Область имени (именная область).

Первое что вы должны сделать при написании драйвера - назвать устройство. Имя должно выть кратким - строка из двух - трех символов. К примеру, параллельные устройства - "lp", дисководы "fd", диски SCSI - "sd".

Создавая ваш драйвер, называйте функции в нем с первыми тремя буквами избранной строки в имени. Так как мы называем его foo - функции в нем соответственно - foo_read и foo_write.

2.3.2 Выделение памяти.

Выделение памяти в ядре отличается от выделения памяти на пользовательском уровне. Вместо функции malloc() выделяющее почти неограниченное количество памяти, существует kmalloc(), которая имеет некоторые отличия:

- Память выделяется кусками размером степени 2, за

исключением кусков больше 128 байтов, размер коих равен степени 2 за вычетом части под метку о размере. Вы можете запросить произвольный размер, однако это будет неэффективно, так как 31 байтового об'екта, к примеру, выделяется 32 байтовый кусок. Общий предел выделяемой памяти 131056 байт.

- В качестве второго аргумента kmalloc() использует приоритет. Он используется в качестве аргумента функции get_free_page(), где он используется в качестве числа определяющего момент возврата. Обычно используемый приоритет - GFP_KERNEL. Если функция может быть вызвана с помощью прерывания используйте GFP_ATOMIC и приготовьтесь к тому, что функция может не работать. Это происходит из-за того, что при использовании GFP_KERNEL kmalloc() может не быть активным в любой момент времени, что не возможно при прерывании. Можно так же использовать опцию GFP_BUFFER, которая используется для выделения ядром области буфера. В драйверах устройств она не используется.

Для очистки памяти, выделенной kmalloc(), используйте функции kfree() и kfree_s(). Они также несколько отличаются от функции free() :

- kfree() - это макрос, вызывающий kfree_s() и работающая как free() вне ядра.

- Если вы знаете размеры об'екта, удаляемого из памяти, вы можете ускорить процесс, запуская сразу kfree_s(). У него существуют два аргумента : первым является аргумент kfree(), вторым - размер удаляемого об'екта.

См 2.6 для получения более подробной информации о kmalloc(), kfree() и о других полезных функциях.

Другой способ сохранить память - выделение ее во время инициализации. Ваша инициализационная функция foo_init() в качестве аргумента использует указатель на текущий конец памяти.Она может взять столько памяти, сколько хочет сохранить указатель/указатели на эту память и возвратить указатель на новый конец памяти.Преимуществом этого метода является то, что при выделении большого буфера в случае, если foo - драйвер не находит foo- устройства, подключенного к компьютеру, память не тратится. Функция инициализации подробно обсуждается в части 2.3.6. Будьте предельно аккуратны при использовании kmalloc(), используйте его только в случае крайней необходимости. Помните, что память в ядре не своппится. Аккуратно выделяйте ее, а затем каждый раз очищайте ее функцией free().

! Существует возможность выделения виртуальной памяти с помощью vmalloc(), однако это будет описано лишь в главе VMM во время ее написания. В данный момент вам придется изучать это самостоятельно.!

2.3.3 Символьные и блочные устройства.

Существует два типа устройств в системах UN*X - символьные и блочные устройства. Для символьных устройств не предусмотрено

буфера, в то время как блочные устройства имеют доступ лишь через буферную память. Блочные устройства должны быть равнодоступными, а для символьных это не обязательно, хотя и возможно. Файловая

система может работать лишь в случае, если она является блочным устройством.

Общение с символьными устройствами осуществляется с помощью функций foo_read() и foo_write(). Функции foo_read() и foo_write() не могут останавливаться в процессе деятельности, поэтому блочные устройства даже не требуют использования этих функций, а вместо этого используют специальный механизм, называемый "strategy routine" - стратегическая подпрограмма. Обмен информацией происходит при помощи функций bread(), breada(), bwrite(). Эти функции, просматривая буферную память, могут вызывать "strategy routine" в зависимости от того, готово устройство или нет к приему информации (в случае записи - буфер переполнен), или же присутствует ли информация в буфере (в случае чтения).Запрос текущего блока из буфера может быть асинхронен чтению - breada() может вначале определить график передачи информации, а затем заняться непосредственно передачей. Далее мы представим полный обзор буферной памяти(кэш). Исходные тексты для символьных устройств содержатся в /kernel/chr_drv, исходники для блочных - /kernel/blk_drv. Для простоты чтения интерфейсы у них довольно просты, за исключением функций записи и чтения. Это происходит из за определения вышеописанной "strategy routine" в случае блочных устройств и соответствующего ему определения foo_read и foo_write() для символьных устройств. Более подробно об этом в 2.4.1 и 2.5.1.

2.3.4. Прерывание или поочередное опрашивание устройств ?

Аппаратное обеспечение работает достаточно медленно. Это определяется временем получения информации, в момент получения которой процессор не занят, и находится в состоянии ожидания. Для того чтобы вывести процессор из режима работа - ожидание, вводятся ! прерывания ! - процессы, предназначенные для прерывания конкретных операций и предоставления ОС задачи по выполнению

которой последняя без потерь возвращается в исходное положение.

В идеале все устройства должны обрабатываться с использованием прерываний, однако на PC и совместимых прерывания используются лишь в некоторых случаях, так что некоторые драйверы вынуждены проверять аппаратное обеспечение на готовность к приему информации.

Так же существуют аппаратные средства (дисплей с распределенной памятью) работающие быстрее остальных частей компьютера. В таком случае драйвер, управляемый прерываниями будет выглядеть нелепо.

В Linux существуют как драйверы, управляемые прерываниями так и драйверы, не использующие прерываний, и оба типа драйверов могут отключаться или включаться во время работы подпрограммы. В частности, "lp" устройство ждет готовности принтера к принятию информации и, в случае отказа, отключается на какой-то промежуток времени, чтобы затем попытаться вновь.

Это улучшает показатели системы. Однако, если вы имеете параллельную карту, поддерживающую прерывания, драйвер, используя ее, увеличит скорость работы. Существуют несколько программных отличий между драйвером, управляемым прерываниями и ждущими драйверами. Для осознания этих отличий вы должны представлять себе устройство системных вызовов UN*X. Ядро - неразделяемая задача под UN*X. В таком случае в каждом процессе находится копия ядра.

Когда процесс запускает системный запрос, он не передает управление другому процессу, а скорее меняет режим исполнения на режим ядра. В этом режиме он запускает он запускает защищенный от ошибок код ядра.

В режиме ядра процесс все еще имеет доступ к пространству памяти пользователя, как и до смены режима, что достигается с помощью макросов: get_fs_*() и memcpy_fromfs(), осуществляющих чтение из памяти, и put_fs_*() и memcpy_tofs(), осуществляющих запись. Так как процесс переходит из одного режима в другой,

вопроса о помещении информации в определенную область памяти не возникает.

Однако во время работы прерывания может работать любой процесс и вышеназванные макросы не могут быть использованы - они либо запишут информацию в случайную область памяти,либо повергнут ядро в ужас.

! Об'ясните, как работает verify_area(), который используется лишь в случае необусловленной защиты от записи во время работы в режиме ядра для проверки области памяти, принимающей информацию.!

Вместо отслеживания прерываний драйвер может выделять временную область для информации.Когда часть драйвера, управляемая прерыванием, заполняет эту область, она замораживает процесс,списывает информацию в пространство памяти пользователя.В блочных устройствах драйвер, создающий эту временную область, снабжен механизмом кеширования, что не предусмотрено в символьных устройствах.

2.3.5. Механизмы замораживания и активизации.

Начнем с об"яснения механизма заморозки и его использования. Это включает в себя то, что процесс, будучи в замороженном состоянии (не функционирует), в какой - то момент времени можно активизировать, а затем опять заморозить (приостановить)!

Возможно, лучший способ понять механизм замораживания и активизации в Linux - изучение исходного текста функции __sleep_on(), использующейся для описания функций sleep_on() и interruptible_sleep_on().

```
static inline void __sleep_on(struct wait_queue **p, int state)
{
    unsigned long flags;
    struct wait_queue wait = { current, NULL };
```



```
    if (!p)
        return;
    if (current == task[0])
        panic ( "task[0] trying to sleep");
    current->state = state;
    add_wait_queue(p, &wait);
    save_flags(flags);
    sti();
    schedule();
```

```
    remove_wait_queue(p, &wait);
    restore_flags(flags);
  }
```

wait_queue - циклический список указателей на структуры задач, определенные в как

```
  struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
  };
```

Меткой состояния процесса в данном случае является или TASK_INTERRAPTIBLE, или TASK_UNINTERRAPTIBLE, в зависимости от того, может ли заморозка процесса прерываться такими вещами, как системные вызовы.Вообще говоря, механизм заморозки необходимо прерывать лишь в случае медленных устройств, так как такое устройство может приостановить на достаточно длительный срок работу всей системы. add_wait_queue() отключает прерывание, создает новый элемент структуры wait_queue, определенной в начале функции как список p.Затем она восстанавливает в исходное положение метку о состоянии процесса.

save_flags() - макрос, сохраняющий флаги процессов, задаваемых в виде аргументов. Это делается для фиксации предыдущего положения метки состояния процесса. Таким образом, функция restore_flags() может восстанавливать положение метки.

Функция sti() затем разрешает прерывания, а schedule() выбирает для выполнения следующий процесс. Задача не может быть избранной для выполнения, пока метка не будет находиться в состоянии TASK_RUNNING.

Это достигается с помощью функции wake_up(),примененной к задаче, ждущей в структуре p своей очереди.

Затем процесс исключает себя из wait_queue,восстанавливает состояние положения прерывания с помощью restore_flags() и завершает работу.

Для определения очередности запросов на ресурсы в структуру wait_queue введены указатели на задачи, использующие этот ресурс. В таком случае, когда несколько задач запрашивают один и тот же ресурс одновременно, задачи, не получившие доступ к ресурсу, замораживаются в wait_queue.По окончании работы текущей задачи активизируется следующая задача из wait_queue,относящаяся к этому ресурсу с помощью функций wake_up() или wake_up_interruptible().

Если вы хотите понять последовательность разморозки задач или более детально изучить механизм заморозки, вам нужно купить одну из книг, предложенных в приложении A и просмотреть !mutual exclusion! и !deadlock!.

2.3.5.1.Усложненный механизм заморозки.

Если механизм sleep_on()/wake_up() в Linux не удовлетворяет вашим требованиям, вы можете усовершенствовать его. В качестве примера тому можете посмотреть серийный драйвер устройства (/kernel/chr_drv/serial.c), функцию
block_til_ready(), которая представляет собой несколько измененные add_wait_queue() и schedule().

2.3.6. VFS.

VFS - Virtual Filesystem Switch (Система виртуального переключения файловой системы) - механизм, позволяющий Linux поддерживать сразу несколько файловых систем. В первой версии Linux доступ к файловой системе осуществляется через подпрограммы, работающие с файловой системой minix. Для обеспечения возможности работы с другой файловой системой ее вызовы переопределяются как функции знакомой Linux системы файлов. Это делается с помощью программы, содержащей структуру указателей на функции, представляющие все возможные действия с файловой системой. Вызывает интерес структура file_operations :

From /usr/include/linux/fs.h:

```
struct file_operations {
    int  (*lseek)   (struct inode *, struct file *, off_t, int);
    int  (*read)    (struct inode *, struct file *, char *, int);
    int  (*write)   (struct inode *, struct file *, char *, int);
    int  (*readdir) (struct inode *, struct file *, struct dirent *,
                int count);
    int  (*select)  (struct inode *, struct file *, int,
                select_table *);
    int  (*ioctl)   (struct inode *, struct file *, unsigned int,
                unsigned int);
    int  (*mmap)    (struct inode *, struct file *, unsigned long,
                size_t, int, unsigned long);
    int  (*open)    (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
};
```

Эта структура содержит список функций, нужных для создания драйвера.

2.3.6.1. Функция lseek().

Функция вызывается, когда в специальном файле, представляющем устройство, появляется системный вызов lseek().Это функция перехода текущей позиции на заданное смещение.Ей задается четыре аргумента :

struct inode * inode
- Указатель на структуру inode для этого устройства.

struct file * file
- Указатель на файловую структуру для данного устройства.

off_t offset
- Смещение от ! origin !.

int origin 0 = смещение от начала.
1 = смещение от текущей позиции.
2 = смещение от конца.

lseek() возвращает -errno в случае ошибки или положительное смещение после выполнения.

Если lseek() отсутствует, ядро автоматически изменяет элемент file -> f_pos.При origin = 2 в случае file -> f_inode = NULL ему присваивается значение -EINVAL,иначе file -> fpos принимает

значение file -> f_inode -> i_size + offset.Поэтому в случае возврата ошибки устройства системным вызовом lseek() вы должны использовать функцию lseek для определения этой ошибки.

2.3.6.2. Функции read() и write().

Функции read() и write() осуществляют обмен информацией с устройством, посылая на него строку символов.Если функции read() и write() отсутствуют в структуре file_operatios, определенной в ядре, то в случае символьного устройства одноименные вызовы будут возвращать -EINVAL.B случае блочных устройств функции не определяются, так как VFS будет общаться с устройством через механизм обработки буфера, вызывающий "strategy routine". См. 2.5.2 для более подробного изучения устройства механизма работы с буфером.

Функции read() и write() используют следующие аргументы :

struct inode * inode
- Указатель на структуру inode специального файла устройства, доступного для использования непосредственно пользователем.

В частности, вы можете найти подномер файла при помощи конструкции unsigned int minor = MINOR(inode -> i_rdev); Определение макроса MINOR находится в < linux/fs.h >, так же, как и масса других нужных определений. Для получения более подробной информации см. fs.h. Более подробное описание представлено в 2.6. Для определения типа файла может быть использована inode -> i_mode.

struct file * file
- Указатель на файловую структуру этого устройства.

char * buf
- Буфер символов для чтения и записи. Он расположен в пространстве памяти пользователя, и доступ к нему осуществляется с помощью макросов get_fs*(), put_fs*() и memcpu*fs(), описанных в 2.6. Пространство памяти пользователя не доступно во время прерывания, так что если ваш драйвер управляется прерываниями, вам придется списывать содержание буфера в очередь (queue).

int count
- Число символов, записанных или читаемых из buf.
count - размер буфера, так что с помощью него легко определить последний символ buf, даже если буфер не заканчивается NULL.

2.3.6.3 Функция readdir().

Еще один элемент структуры file_operations, используемый для описания файловых систем так же, как драйверы устройств. Функция не нуждается в предопределении. Ядро возвращает -ENOTDIR в случае вызова readdir() из специального файла устройства.

2.3.6.4 Функция select().

Функция select() полезна в основном в работе с символьными устройствами. Обычно она используется для многократного чтения без использования последовательного вызова функций. Приложение делает системный вызов select(), задавая ему список дескрипторов файлов, затем ядро сообщает программе, при просмотре какого дескриптора

она была активизирована. Также select() иногда используется как таймер. Однако функция select() в драйвере устройства не вызывается непосредственно системным вызовом, так что file_operations select() выполняет небольшое количество примитивных операций. Ее аргументы:

struct inode * inode
- Указатель на структуру inode устройства.

struct file * file
- Указатель на файловую структуру устройства.

int sel_type
- Тип совершаемого действия
SEL_IN - чтение
SEL_OUT - запись
SEL_EX - удаление

select_table * wait
- Если wait = NULL, функция select() проверяет, готово ли устройство, и возвращается в случае отсутствия готовности. Если wait не равен NULL, select() замораживает процесс и ждет, пока устройство не будет готово. Функция select_wait() делает то же, что и select() при wait = NULL.

2.3.6.5 Функция ioctl().

Функция ioctl() осуществляет функцию передачи контроля ввода/вывода. Структура вашей функции должна быть следующей: первичная проверка ошибок, затем переключение, дающее вам право контролировать все ioctl. Номер ioctl находится в аргументе cmd, аргумент контролируемой команды находится в arg. Для работы с ioctl() вы должны иметь подробное представление о контроле над вводом/выводом. Если вы сомневаетесь в правильности использования ioctl(), спросите кого-нибудь, так как эта функция в текущий момент может оказаться ненужной. Так как ioctl() является частью интерфейса драйверов, вам придется уделить ей внимание.

struct inode * inode
- Указатель на inode структуру данного устройства;

struct file * file
- Указатель на файловую структуру устройства;

unsigned int cmd
- Команда, над которой осуществляется контроль;

unsigned int arg
- Это аргумент для команды, определяется пользователем. В случае, если он вида (void *), он может быть использован как указатель на область пользователя, обычно находящуюся в регистре fs.

Возвращаемое значение :
-errno в случае ошибки, все другие значения определяются пользователем.

Если слот ioctl() в file_operations не заполнен, VFS возвращает значение -EINVAL, однако в любом другом случае, кесли cmd принимант одно из значений - FIOCLEX, FIONCLEX,FIONBIO, FIOASYNC, будет происходить следующее:

FIOCLEX 0x5451
Устанавливает бит "закрытие для запуска"

FIONCLEX 0x5450
Очищает бит "закрытие для запуска"

FIONBIO 0x5421
Если аргумент не равен 0, устанавливает O_NONBLOCK, иначе очищает O_NONBLOCK.

FIOASYNC 0x5421
Если аргумент не равен 0, устанавливает O_SYNC, иначе очищает O_SYNC. Пока еще не описано, но для полноты

вставлено в ядро.

Помните, что вам надо учитывать эти четыре номера при написании своих ioctl(), так как они могут быть несовместимы между собой, откуда в программе может возникнуть тяжело обнаруживаемая ошибка.

2.3.6.6.Функция mmap().

struct inode *inode
- Указатель на inode

struct file *file
- Указатель на файловую структуру

unsigned long addr
- Начальный адрес блока, используемого mmap()

size_t len - Общая длина блока.

int prot - Принимает значения:
PROT_READ читаемый кусок
PROT_WRITE перезаписываемый кусок
PROT_EXEC кусок, доступный для запуска
PROT_NONE недоступный кусок

unsigned long off
- Внутрифайловое смещение, от которого производится перестановка. Этот адрес будет переставлен на адрес addr.

[В описании распределения памяти описано, как функции интерфейса Менеджера виртуальной памяти могут быть использованы mmap().]

2.3.6.7. Функции open() и release().

struct inode *inode
- Указатель на inode

struct file *file
- Указатель на файловую структуру

Функция вызывается после открытия специальных файлов

устройств. Она является механизмом слежения за последовательностью выполняемых действий. Если устройством пользуется лишь один процесс, функция open() закроет устройство любым доступным в данный момент способом, обычно устанавливая нужный бит в положение "занято". Если процесс уже использует устройство (бит уже установлен), open() возвращает -EBUSY.

Если же устройство необходимо нескольким процессам, эта функция обладает возможностью любой очередности.

Если устройство не существует, open() вернет -ENODEV.

Функция release() вызывается лишь тогда, когда процесс закрывает последний файловый дескриптор. release() может переустанавливать бит "занято". После вызова release(), вы можете очистить куски выделенной kmalloc() памятью под очереди процессов.

2.3.6.8 Функция init().

Эта функция не входит в file_operations но вам придется использовать ее, так как именно она регистрирует file_operations с содержащейся там VFS - - без нее запросы на драйвер будут находится в беспорядочном состоянии. Эта функция запускается во время загрузки и самоконфигурирования ядра. init() получает переменную с адресом конца используемой памяти. Затем она обнаруживает все устройства, выделяет память, исходя из их общего числа, сохраняет полезные адреса и возвращает новый адрес конца используемой памяти. Функцию init() вы должны вызывать из определенного места. Для символьных устройств это /kernel/cdr_dev/mem.c. В общем случае функции надо задавать лишь переменную memory_start.

Во время работы функции init(), она регистрирует ваш драйвер

с помощью регистрирующих функций. Для символьных устройств это register_chrdev(). register_chrdev использует три аргумента :

а. int major - основной номер устройства.
б. srtring name - имя устройства.
в. адрес #DEVICE#_fops структуры file_operations.

После окончания работы функции, файлы становятся доступными для VFS, и она по надобности переключает устройство с одного вызова на другой.

Функция init() обычно выводит сведения о найденном аппаратном обеспечении и информацию о драйвере.Это делается с использованием функции printk().

2.4 Символьные устройства.

2.4.1. Инициализация.

Кроме функций описанных в file_operations, есть еще одна функция, кото- рую вам надо вписать в функцию foo_init(). Вам придется изменить функцию chr_dev_init() в chr_drv/mem.c для вызова вашей функции foo_init(). foo_init() вначале должна вызывать register_chrdev() для определения самой себя и установки номеров устройств. Аргументы register_chrdev() :

int major - основной номер драйвера.

char *name - имя драйвера оно может быть изменено, но не имеет практического применения.

struct file_operations *fops - адрес определенной вами
file_operations.

Возвращаемые значения : 0 - в случае если указанным основным номером
ни одно устройство более не обладает.
не 0 в случае некорректного вызова.

2.4.2 Прерывания или последовательный вызов ?

В драйверах, не использующих прерывания, легко пишутся функции foo_read() и foo_write() :

```
static int foo_write(struct inode * inode, struct file * file,
                     char * buf, int count)
{
   unsigned int minor = MINOR(inode->i_rdev);
   char ret;
   while (count > 0) {
      ret = foo_write_byte(minor);
if (ret < 0) {
         foo_handle_error(WRITE, ret, minor);
         continue;
      }
      buf++ = ret; count--
   }
   return count;
}
```

foo_write_byte() и foo_handle_error() - функции, также определенные в foo.c или псевдокоде.

WRITE - константа или определена #define.

Из примера также видно как пишется функция foo_read(). Драйверы, управ- ляемые прерываниями, более сложны :

Пример foo_write для драйвера, управляемого прерываниями :

```
static int foo_write(struct inode * inode, struct file * file,
                     char * but, int count)
{
   unsigned int minor = MINOR(inode->i_rdev);
   unsigned long copy_size;
```



```
   unsigned long total_bytes_written = 0;
   unsigned long bytes_written;
   struct foo_struct *foo = &foo_table[minor];

   do {
      copy_size = (count <= FOO_BUFFER_SIZE ? count : FOO_BUFFER_SIZE);
      memcpy_fromfs(foo->foo_buffer, buf, copy_size);

      while (copy_size) {
           /* запуск прерывания */

           if (some_error_has_occured) {
              /* обработка ошибочного состояния */                    }
```

```
            current->timeout = jiffies +FOO_INTERRUPT_TIMEOUT;
               /* set timeout in case an interrupt has been missed */
            interruptible_sleep_on(&foo->foo_wait_queue);
            bytes_written = foo->bytes_xfered;
            foo->bytes_written = 0;
            if (current->signal & ~current->blocked) {
               if (total_bytes_written + bytes_written)
                  return total_bytes_written + bytes_written;
               else
                  return -EINTR; /* nothing was written, system
                           call was interrupted, try again */
            }
         }
         total_bytes_written += bytes_written;
         buf += bytes_written;
         count -= bytes-written;

      } while (count > 0);

      return total_bytes_written;
   }

   static void foo_interrupt(int irq)
   {
```



```
      struct foo_struct *foo = &foo_table[foo_irq[irq]];

      /* Here, do whatever actions ought to be taken on an interrupt.
         Look at a flag in foo_table to know whether you ought to be
         reading or writing. */

      /* Increment foo->bytes_xfered by however many characters were
         read or written */
      if (buffer too full/empty)
          wake_up_ interruptible(&foo->foo_wait_queue);
   }
```

Здесь функция foo_read также аналогична. foo_table[] - массив структур, каждая из которых имеет несколько элементов, в том числе foo_wait_queue и bytes_xfered, которые используются и для чтения, и для записи. foo_irq[] - - массив из 16 целых использующийся для контроля за приоритетами элементов foo_table[] засылаемыми в foo_interrupt().

Для указания обработчику прерываний вызвать foo_interrupt() вы должны использовать либо request_irq(), либо irqaction(). Это делается либо при вызове foo_open(), либо для простоты в foo_init(). request_irq() работает проще нежели irqaction и напоминает работу сигнального переключателя. У нее существует два аргумента:

- номер irq, которым вы располагаете

- указатель на процедуру управления прерываниями, имеющую аргумент типа integer.

request_irq() возвращает -EINVAL, если irq > 15, или в случае указателя на программу равного NULL, EBUSY если прерывание уже используется или 0 в случае успеха.

irqaction() работает также как функция sigaction() на пользовательском уровне и фактически использует структуру sigaction. Поле sa_restorer() в структуре не используется, остальное - же осталось неизменным. См. раздел "Функции поддержки"

для более полной информации о irqaction().

2.5 Драйверы для блочных устройств.

При поддержке файловой системы устройства, она должна быть разбита на блоки самим устройством. Это означает что устройство не должно принимать информацию посимвольно, а значит должно быть равнодоступно. Иными словами вы, в любой момент времени должны имеет доступ к любому состоянию физического устройства.

Вам не придется в случае блочных устройств пользоваться функциями read() и write(). Вместо них используются функции block_read() и block_write() находящиеся в VFS и называемые !strategy routine! или функцию request() которую вы пишете в позиции функций read() и write() в вашем драйвере. strategy routine вызывается также механизмом кэширования буфера, который запускается подпрограммами VFS, которые представлены в виде обычных файлов.

Запросы ввода-вывода поступают через механизм кэширования буффера в подпрограмму называется ll_rw_block, которая создает список запросов упорядоченных алгоритмом !elevator!, который сортирует списки для более быстрого доступа и повышения эффективности работы устройств.

Затем она вызывает фнкцию request() для осуществления ввода - вывода. Отметим что диски SCSI и CDROM также относятся к блочным устройствам но управляются более особым образом. Часть 2.7 "Написание драйвера SCSI" описывает это более подробно.

2.5.1 Инициализация

Инициализация блочного устройства имеет более общий вид, нежели инициализация символьного устройства, т.к. часть "инициализации" происходит во время компиляции. Также существует

вызов register_blkdev() аналогичный register_chrdev() определяющий какой из драйверов может быть назван актив- ным, работающим, присутствующим.

2.5.1.1 Файл blk.h

Вначале текста вашего драйвера после описания.h файлов вы должны написать две строки:

```
#define MAJOR_NR DEVICE MAJOR
#include
```

где DEVICE_MAJOR - основной номер вашего устройства.drivres/block/blk.h требует основной номер для установки других определений и макросов драйвера.

Теперь вам нужно изменить файл blk.h.После #ifdef MAJOR_NR есть часть программы в которой определены некоторые основные номера, защищенные

```
#elif (MAJOR_NR = DEVICE_MAJOR).
```

В конце списка вы запишете раздел для вашего драйвера :

```
#define DEVICE_NAME "device"
#define DEVICE_REQUEST do_dev_request
#define DEVICE_ON( device ) /* usully blank, see below */
#define DEVICE_OFF( device ) /* usully blank, see below */
#define DEVICE_NR( device ) (MINOR(device))
```

DEVICE_NAME - имя устройства.В качестве примера посмотрите предыдущие записи в blk.h.

DEVICE_REQUEST - ваша "strategy routine", которая будет осуществлять ввод/вывод в вашем устройстве.См 2.5.3 для более полного изучения.

DEVICE_ON и DEVICE_OFF - для устройств, которые включаются/выключаются во время работы.

DEVICE_NR(device) - используется для определения номера физического устройства с помощью подномера устройства. В частности, драйвер hd, в то время как второй жесткий диск работает с подномером 64, DEVICE_NR(device) определяется (MINOR(device) >> 6).

Если ваш драйвер управляется прерываниями, также установить

```
#define DEVICE_INTR do_dev
```

что автоматически становится переменной и используется даже в blk.h, в основном макросами SET_INTR и CLEAR_INTR.

Также вы можете присовокупить такие определения :

```
#define DEVICE_TIMEOUT DEV_TIMER
#define TIMEOUT_VALUE n,
```

где n - число тиков часов (в Linux/386 - сотые секунды)для паузы в случае незапуска прерывания. Это делается для того,чтобы драйвер не ждал прерывания, которое может никогда не случиться. Если вы делаете эти установки, они автоматически используются

SET_INTR для установки драйвера в положение ожидания. Конечно, в таком случае ваш драйвер должен будет иметь возможность отмены ожидания.

2.5.1.2. Опознание комплектующих PS.

![Вам следует изучить текст подпрограмм genhel.c и include для понимания их использования.]

2.5.2. Механизм кеширования буфера.

Здесь следовало бы об'яснить, как вызывается ll_rw_block(), рассказать о getblk(), bread() и breada(), bwrite(). Подробное об'яснение механизма кеширования буфера отложено до создания

описания VFS.

Читателю предлагается изучить его самостоятельно. Если у вас возникнут трудности, обращайтесь за помощью к автору этой книги.

2.5.3. Strategy Routine.

Обработка блочных данных осуществляется strategy routine. Эта подпрограмма не имеет аргументов и ничего не возвращает, однако ей известно, где найти список запросов ввода/вывода (CURRENT определена как blk_dev[MAJOR_NR].current_request),а также как получать данные от устройства и формировать блоки. Она вызывается при !запрещенных ! прерываниях, так что для разрешения прерываний вам надо вызвать функцию sti() до возврата "strategy routine".

"Strategy routine" сначала вызывает макрос INIT_REQUEST,который убеждается в принадлежности запроса списку запросов. add_request() сортирует запросы в определенном порядке с помощью алгоритма elevator, вызываемого в связи с каждым запросом, так что "strategy routine" должна лишь удовлетворить текущий запрос, затем вызвать end_request(1) для удаления запроса и так далее,пока запросов в списке не останется.

В случае, если ваш драйвер управляется прерываниями, он, вызывая "strategy routine", передает ей конкретный запрос, прерывая работу компьютера, затем, после выполнения задачи, поставленной запросом, он исключает последний из списка с помощью end_request(), после чего в нужный момент, определяемый обработчиком прерываний, драйвер опять вызывает "strategy routine" со следующим процессом.

Если во время удовлетворения текущего запроса происходит сбой ввода/вывода, для снятия запроса также вызывается end_request().

Запрос может быть на чтение и запись. Драйвер определяет тип запроса, просматривая CURRENT -> cmd.

```
CURRENT -> cmd == READ - чтение,
CURRENT -> cmd == WRITE - запись.
```

Если устройство имеет раздельные управляемые прерываниями подпрограммы чтения и записи, то драй вер должен использовать SET_INTR(n) для определения типа запроса.

!Здесь нужно привести пример strategy routine процедуры, не использующей прерывания и использующей их.Драйвер, управляемый прерываниями, будет заключать в себе раздельные процедуры ввода/вывода для указания, как использовать SET_INTR. !

2.6. Функции поддержки.

Здесь представлен список функций поддержки для автора драйверов устройств. Приведенный далее список не полон, однако, он окажется вам полезен.

```
add_request()
static void add_request(struct blk_dev_struct *dev,
                        struct request *req )
```

Эта функция статическая, находящаяся в ll_rw_block.c, и она

может быть вызвана в тексте другой программы. Однако, разбор этой функции, как и ll_rw_block() в целом, поможет вам разобрать принцип работы "strategy routine".

Установленный порядок алгоритма сортировки elevator:

a) Операции чтения имеют более высокий приоритет, чем записи.
b) Устройства с меньшими подномерами ставятся в очередь перед устройствами с большими.
c) Условие с подномерами распространяется на номера блоков.

Алгоритм elevator описан в макросе IN_ORDER(), который определен в drivers.block/blk.h

Определена в drivers/block/ll_rw_block.c См. также make_request(), ll_rw_block()

add_timer()
void add_timer(struct timer_list *timer)
#include

Устанавливает структуру таймера в список timer.

Структура timer_list определена как:

```
struct timer_list {
    struct timer_list *next;
    struct timer_list *prev;
    unsigned long data;
    void (*function) (unsigned long)
```

Для каждого вызова add_timer() вам надо создать в памяти структуру timer_list, а затем вызвать init_timer(), передав ей указатель на вашу timer_list. Она обнулит последующий(next) и предшествующий(prev) элементы. По мере надобности вы можете создать одновременно несколько структур timer_list и сформировать из них список.

Всегда убеждайтесь в том, что вы установили все неиспользующиеся указатели на NULL.

Для каждой структуры списка вы устанавливаете три переменные:

expires - число "тиков" (100 - е секунды в Linux/86) при достижении которого происходит приостановка процесса.

function - Функция в области ядра запускаемая во время приоста- новки.

data - Используется как аргумент во время вызова функции.

Список в программе следует представлять в виде указателя на первый элемент, являющийся также аргументом add_timer(). Также вам придется создать копию этого указателя для продвижения по списку. Примечание: Эта функция не представляет собой идейно новый процесс. Если вы хотите работать с процессом находящимся в режиме приостановки, вам в любом случае придется использовать конструкции активизации и заморозки. Функции используемые этим механизмом будут использоваться в одинаковом контексте с функциями обработчика прерываний.

Определена в kernel/sched.c
См. также timer_table в include/linux/timer.h
init_timer,
del_timer.

cli()
#define cli() __asm__ __volatile__ ("cli"::)
#include

Пресекает неопознанные прерывания процессов. cli - "CLear Interrupt enable" - (очистка от запрещенных прерываний)

См. также sti()

del_timer
void del_timer(struct timer_list *timer)
#include

Уничтожает структуры таймера в списке timer.

Элемент списка таймера, который вы желаете удалить должен быть созданным ранее с помощью add_timer(). Этим вызовом вы также одновременно очищаете память выделенную под удаляемый элемнт.
Определен в kernel/sched.c
См. также: timer_table в include/linux/timer.h, init_timer(),
add_timer().

end_request()
static void end_request(int uptodate)

#include "blk.h"

Вызывается после удовлетворения запроса. Имеет один аргумент:

uptodate Если не равен нулю - запрос удовлетворен
Не равен - обратная ситуация.

Если запрос удовлетворен, end_request() просматривает список запросов, открывает доступ в буфер, подбирает время включения механизма перестановки задач (sheduler), замороженный в make_request(),ll_rw_page() и ll_rw_swap_file(), до активизации всех процессов замороженных в wait_for_request.

Примечание: Это - статическая функция, определенная в drivers/block/blk.h для каждого устройства не включая SCSI. (Устройства SCSI выполняют вышеуказанную процедуру несколько иначе; программы SCSI на высоком уровне, непосредственно обеспечивают функционирование драйверов устройств SCSI на низком уровне). Она включает в себя несколько определений статических характеристик устройства, таких как номер. Эта функция значительно быстрее своего более общего Си-го аналога.

Определена в kernel/blk_drv/blk.h
См. также ll_rw_block(), add_request(),make_request().

free_irq()
void free_irq(unsigned int irq)
#include

Освобождает приоритет прежде зарезервированный request_irq()
или irqaction(). Имеет один аргумент:

irq - приоритет нуждающийся в освобождении.

Определена в kernel/irq.c
См. также request_irq(),irqaction().

get_user*()
inline unsigned char get_user_byte(const char *addr)
inline unsigned short get_user_word(const short *addr)
inline unsigned long get_user_long(const int *addr)
#include

Позволяет драйверу иметь доступ к пространству памяти
пользователя отличающееся по адресам от пространства
ядра.

Предупреждение: Эта функция может неявно повлиять на ввод/вывод,
если доступная память была своппирована и в пространстве
памяти используемой вами могут происходить непредвиденные
изменения. Никогда не пишите в ответственных местах
ваших программ эту функцию, даже если эти части защищены парой
cli()/sti(). Если вы хотите использовать данные в пространстве
пользователя спишите их сначала в ядровое, затем уже хачите.
Функция имеет один аргумент:

addr Адрес из которого берется дата.

Возвращаемое значение: Дата из пространства памяти пользова-
теля находящаяся по этому смещению.

inb(), inb_p()
inline unsigned int inb(unsigned short port)
inline unsigned int inb_p(unsigned short port)

#include

Чтение одного байта из порта. inp_b() перед возвратом делает
паузу (некоторые устройства не воспринимают быстрого обмена
информацией), inb() работает без задержек.

У обеих функций один аргумент:
port - Порт из которого получается информация.

Возвращаемое значение: Возвращаемый байт находится в нижних
байтах 32-битного целого, 3 верхних
байта не используются.

Определена в include/asm/io.h
См. также outb(),outb_p().

init_timer()
Встроенная функция для инициализации структур timer_list для
использования add_timer()

Определена в include/linux/timer.h
См. также add_timer().

irq_action()
int irqaction(unsigned int irq, struct sigaction *new)
#include

Прерывания аппаратного обеспечения действительно сильно похожи на сигналы. Следовательно мы можем представлять прерывания как сигналы. Поле struct sigaction, sa_restorer() не используется, но оно одинаково. Аргумент целого типа функции sa.handler() может иметь разный смысл в зависимости от того установлен-ли приоритет(IRQ) с помощью флага SA_INTERRUPT. Если нет то аргумент функции поступает к обработчику в виде указателя на текущую структуру, если да поступает как номер приоритета. Для примера установки обработчика для использования SA_INTERRUPT разберите как установлена rs_interrupt() в.../kernel/chr_drv/serial.c.
Флаг SA_INTERRUPT используется для определения будет-ли прерывание "коротким". Обычно во время отключения прерывания, проверяется глобальный флаг need_reshed. Если он не равен 0, то shedule() запускает следующий на очереди процесс. Также она вызывается при полном запрете прерываний. Однако установив в структуре sigaction, поле sa_flags как SA_INTERRUPT, мы выберем работу с "короткими"

прерываниями, которые исключают некоторые процессы не используя при этом schedule().

irqaction задается два аргумунта:

irq - Номер приоритета на который претендует драйвер.
new - Указатель на структуру sigaction.
Возвращаемые значения :
-EBUSY - прерывание уже перехвачено.
-EINVAL - если sa.handler = NULL.
0 - в случае успеха.

Определена в kernel/irq.c
См.также request_irq(),free_irq()

IS_*(inode)
IS_RDONLY(inode) ((inode)->i_flags & MS_RDONLY)
IS_NOSUID(inode) ((inode)->i_flags & MS_NOSUID)
IS_NODEV(inode) ((inode)->i_flags & MS_NODEV)
IS_NOEXEC(inode) ((inode)->i_flags & MS_NOEXEC)
IS_SYNC(inode) ((inode)->i_flags & MS_SYNC)

#include

Пять тестов на принадлежность inode к файловой системе устанавливающей соответствующий флаг.

kfree*()
#define kfree(x) kfree_s((x), 0)
void kfree_s(void *obj, int size)

#include

Очищает память выделенную прежде kmalloc(). Существуют два возможных аргумента:

obj указатель на память ядра для чистки.
size Для ускорения процесса, в случае если вы точно знаете

размер удаляемого куска, используйте сразу kfree_s() с

указанием этого размера. В таком случае механизму управления памяти не придется определять к какой области памяти принадлежит обьект.

Определена в mm/kmalloc.c, include/linux/malloc.h
См. также: kmalloc()

kmalloc()
void *kmalloc(unsigned int len, int priority)
#include

Максимальный обьем памяти выделяемый kmalloc() - 131056 байт ((32*4096)-16) в пакетах размерами степени двойки за вычетом некоего небольшого числа, за исключением чисел меньше или равных 128. Более подробно в определении в mm/kmalloc.c

Использует два аргумента:
len - длина выделяемой памяти. Если размер будет превышать допустимый kmalloc() выдаст сообщение об ошибке : "kmalloc of too large a block (%d bytes)" и вернет NULL.
priority- приимает значения GFP_KERNEL или GFP_ATOMIC. В случае выбора GFP_KERNEL kmalloc() может находится в замороженном состоянии в ожидании освобождения блока памяти нужного размера. Это является нормальным режимом работы kmalloc(), однако бывают случаи, когда более удобен быстрый взврат. Одним из примеров этому служит своппируемое пространство в котором могли возникнуть несколько запросов на одно и то же место, или сетевое пространство в котором события могут происходить намного быстрее своппинга диска в связи с поиском свободного места. GFP_ATOMIC как раз и служит для отключения клонящегося ко сну kmalloc().

Возвращаемые значения: В случае провала - NULL.
В случае успеха - указатель на начало выделенного куска.

Определен в mm/kmalloc.h
См. также: kfree()

ll_rw_block
void ll_rw_block(int rw, int nr, struct buffer_head *bh[])
#include

Ни один драйвер устройства никогда к этой функции непосредственно не обращается - обращение идет исключительно через механизм кэширования буфера, однако разбор этой функции поможет вам познать принципы работы strategy routine.

После проверки на наличие ожидающих запросов в очереди запросов устройства, ll_rw_block() запирает очередь, так чтобы ни один запрос не покинул ее. Затем функция make_request() по одному вызывает запросы отсортированные в очереди алгоритмом elevator. strategy routine для устройсва, в случае запертой очереди, неактивна, так что функция вызывает ее с !запрещенными прерываниями!, Однако strategy routine имеет возможность разрешения последних.

Определена в devices/block/ll_rw_block.c
См. также make_request(), add_request()

MAJOR()

#define MAJOR(a) (((unsigned)(a))>>8)
#include

Функция берет в качестве аргумента 16-ти битный номер устройства
и возвращает основной номер.
См. также MINOR().

make_request()
static void make_request(int major, int rw, struct buffer_head *bh)

Эта функция является статической, принадлежит к ll_rw_block.c и не
может быть вызвана другой прграммой. Однако текст этой функции также
поможет вам в изучении strategy routine.

make_request() вначале проверяет принадлежность запроса к типу
чтения или записи, затем просматривает буфер на предмет доступа.
Если буфер закрыт она игнорирует запрос и завершается. Иаче она
закрывает буфер и, за исключением драйвера SCSI, проверяет очередь на
заполненность (в случае записи) или на присутствие запроса (чтение).
Если в очереди нет свободного места, то make_request() замораживается
в состоянии wait_for_request и пытается снова поместить запрос в
очередь, когда размораживатся. Когда в очереди находится место для
запроса, он помещается туда с помощью add_request().

Определена в devices/block/ll_rw_block.c
См.также add_request(), ll_rw_block()

MINOR()
#define MINOR(a) ((a)&0xff)
#include

По 16-ти битному номеру устройства определяет подномер маскированием
основного номера.

См. также MAJOR().

memcpy_*fs()
inline void memcpy_tofs(void *to,const void *form, unsigned long n)
inline void memcpy_fromfs(void *to,const void *from, unsigned long n)
#include

Служит для обмена памятью пользовательского уровня и уровня ядра
копируя кусками не более одного байта, слова. Будте осторожны в
указании правильного порядка аргументов.

****
Эти функции требуют три аргумента:

to Адрес, куда перенести дату.
from Адрес, откуда.
n Количество переписываемых байтов.

Определена в include/asm/segment.h
См. также: get_user*(),put_user*(),cli(),sti().

outb(), outb_p()
inline void outb(char value,unsigned short port)
inline void outb_p(char value, unsigned short port)
#include

Записывает в порт одие байт. outb() работает без задержки,
в то время как outb_p() перед возвратом делает паузу, так
как некоторые устройства не воспринимают быстрого обмена
информацией. Обе функции используют два аргумента:

value Записываемый байт.
port Порт в который он записывается.

Определены в include/asm/io.h
См. также inb(), inb_p().

printk()
int printk(const char* fmt,...)
#include

printk() - это ядровая модификация printf()с некоторыми ограничениями
такими, как запрещение использования типа float и
несколько других изменений описанных подробно в kernel/vsprintf.c
Количество переменных функции может меняться:

fmt Строка формата (аналогична printf())
... Остальные аргументы (аналогично printf())

Возвращаемое значение : Число записанных байтов.

Примечание: Никогда не используйте функцию printfk() в коде
защищенном cli(), так как из за постоянного своппинга
задействуемой памяти, обращение ф-ции к ней
может вызвать неявный ввод-вывод с последующей

выгрузкой.

Определено в kernel/printk.c

put_user*()
inline void put_user_byte(char val,char *addr)
inline void put_user_word(short val,short *addr)
inline void put_user_long(unsigned long val,
unsigned long *addr)
#include

Позволяет драйверу писать информацию в пространство пользователя,
с сегментом отличающимся от ядра. Во время обращения к ядру с
помощью системного вызова, селектор сегмента пользовательской
области заносится в сегментный регистр fs.

Примечание: см Примечание get_user*()

Функция имеет два аргумента:

val записываемое.
addr адрес для записи мнформации.

Определена в asm/segment.h
См. также: memcpy_*fs(), get_user*(), cli(), sti().

register_*dev()
int register_chrdev(unsigned int major, const char *name,
struct file_operations *fops)

int register_blkdev(unsigned int major, const char *name,

struct file_operations *fops)

#include
#include

Регистрирует устройство ядром, дав последнему возможность проверки

на занятость основного номера устройства иным драйвером. Имеет три аргумента:

major основной номер регистрируемого устройства
name строка идентифицирующая драйвер. Используется при выводе в файл в /proc/devices
fops Указатель на структуру file_operations. Во избежании ошибки не должен быть равен NULL.

Возвращаемые значения: -EINVAL если основной номер >= MAX_CHRDEV или MAX_BLKDEV (определены в) для символьных или блочных устройств соответственно.
-EBUSY если основной номер уже занят.
0 - в случае успеха.

Определена в fs/devices.c
См. также: unregister_*dev().

request_irq()
int request_irq(unsigned int irq, void (*handler)(int),
unsigned long flags, const char *device)
#include
#include

Запрашивает в ядре IRQ и устанавливает приоритетный обработчик прерываний в случае удовлетворения запроса. Имеет четыре аргумента:

irq запрашиваемый приоритет.
handler обработчик прерываний вызываемый во время поступления сигнала с IRQ.
flags устанавливаются в SA_INTERRUPT для запроса "быстрого" прерывания или в случае значения 0 "ждущего".
device Строка содержащая имя драйвера устройства.

Возвращаемые значения: -EINVAL если irq > 15, или handler = NULL.
-EBUSY если irq уже используется.

См. также: free_irq(), irqaction().

select_wait()
inline void select_wait(struct wait_queue **wait_address,
select_table *p)
#include

Помещает процесс в определенную очередь select_wait. Имеет два аргумента:

wait_address Адрес указателя на wait_queue для помещения в циклический список запросов.
p Если p=NULL, select_wait бездействует, иначе текущий процесс замораживается.
wait переносится из функции select().

Определена в: linux/sched.h
См. также: *sleep_on(), wake_up*().

*sleep_on()
void sleep_on(stuct wait_queue **p)
void interruptible_sleep_on(struct waitqueue **p)
#include

Замораживает процесс до определенного события, помещая информацию, требуемую для активизации, в wait_queue. sleep_on() используется в случае запрещенных прерываний, так что процесс может быть запущен исключительно функцией wake_up(). interruptible_sleep_on() используется в случае заморозки с разрешенными прерываниями, когда процесс может быть активизирован определенными сигналами, перерывами работы других процессов. Используя wake_up_interruptible() вы можете активизировать процесс с дальнейшим его исключением по отработке. Используют один аргумент.

p Указатель на заданную структуру wait_queue, в которую записывается информация для пробуждения процесса.

Определена в: kernel/sched.c
См. также: select_wait(), wake_up*().

sti()
#define sti()__asm__ __volatile__("sti"::)
#include

Разрешает неопознанные прерывания. sti - "SeT Interrupt enable"

Определена в asm/system.h
См. также: cli().

sys_get*()
int sys_getpid(void)
int sys_getuid(void)
int sys_getgid(void)
int sys_geteuid(void)
int sys_getegid(void)
int sys_getppid(void)
int sys_getpgrp(void)

Эти системные вызовы могут быть использоваы для получения информации находящейся в таблице ниже или информации, которую можно получить прямо из таблицы процесса:

foo=current->pid;

pid ..... ID процесса.
uid ..... ID пользователя.
gid ..... ID группы.
euid..... ID "эффективного" пользователя.
egid..... ID "эффективной" группы.
ppid..... ID пророднтеля процесса.
pgid..... ID пророднтеля группы.

Системные вызовы не находят широкого применения, так как они не достаточно быстры и требуют большого количества памяти. Поэтому они более не экспортируются как символы через все ядро.
Определена в: kernel/sched.c

unregister_*dev()
int unregister_chrdev(unsigned int major,const char *name)
int unregister_blkdev(unsigned int major,const char *name)

#include
#include

Аннулирует регистрацию драйвера устройства ядром, позволяя последнему передать основной номер другому устройству. Имеет два аргумента.

major Основной номер зарегестрируемого ранее устройства. Должен быть идентичен номеру заданному register_*dev().
name Уникальная строка идентифицирующая устройство. Должно быть также идентична заданной в register_*dev().

Возвращаемые значения:

-EINVAL если основной номер >= MAX.CHRDEV или MAX_BLKDEV (определены в), для символьных и блочных устройств соответственно, если не имя или основной номер не совпвдают с заданными при регистрации.
0 в случае успеха.

Определена в fs/devices.c
См. также: register_*dev().

wake_up*()
void wake_up(struct wait_queue **p)
void wake_up_interruptible(struct wait_queue **p)

#include

Активизируют процесс, замороженный соответственной функцией *sleep_on(). wake_up() служит для активизации процессов находящихся в очереди, где они могут быть помечены как TASK_INTERRUPTIBLE или TASK_UNINTERRUPTIBLE, в то время как wake_up_interruptible() может активизировать процессы лишь помеченные второй меткой, однако работает на порядок быстрее wake_up(). Имеют один аргумент:

q указатель на структуру wait_queue, активизируемого процесса.

Помните что wake_up() не осуществляет переключение задач, она лишь делает процесс запускаемым для того, чтобы далее вызванная функция schedule() использовала его как претендента на выполнение.

Определена в kernel/sched.c
См. также: select_wait(), *sleep_on().

2.7. Написание драйвера SCSI.

2.7.1. Зачем нужны драйверы SCSI.

Ядро Linux содержит драйверы для следующих основных адаптеров SCSI: Adaptec 1542, adaptec 1740, Future Domain TMS-1660/TMS-1680, Segate ST-01/ST-02, Ultrastor 14F и Western Digital WD-7000.вы можете написать ваш собственный драйвер для неподдерживаемого адаптера. Также вы можете изменять готовые драйверы.

>

Transfer interrupted!

тупление к стандартному описанию SCSI-2 дает подробнейшее определение Small Computer System Interfase (Интерфейс Малых Компьютерных Систем) и об'ясняет, как SCSI-2 соотносится с SCSI-1 и CCS.

Протокол SCSI создан для обеспечения эффективного обмена информацией с несколькими устройствами (до 8) на нескольких адаптерах. Данные

могут передаваться асинхронно со скоростью, определяемой характеристиками устройства и длиной кабеля.

Синхронный обмен информацией может поддерживать скорость до 10 млн. передач в секунду. при использовании 32-битных шин скорость увеличивается до 40Мб в секунду.

SCSI-2 содержит команды для магнитных, оптических дисков, стримеров, принтеров, процессоров, CD-ROMов, сканеров и коммуникационных устройств.

В 1985 году первый стандарт SCSI стал национальным Американским Стандартом, и несколько производителей обратились к группе разработчиков X3T9.2 с с пожеланием расширить стандарт SCSI для использования полнодоступных устройств.
В процессе расширения SCSI группа X3T9.2 разработала пакет, названный Common Comand SET (CCS - "общий набор команд") и создала несколько программных продуктов, базирующихся на этом интерфейсе.
Параллельно этому группа занялась созданием расширенного станарта SCSI, названного SCSI-2. Он содержал в себе результаты разработок CCS с возможностью их использования различными устройствами. Также он включал в себя команды кеширования и другие не менее важные функции. Так как SCSI-2 был лишь более качественной расширенной копией стандарта SCSI-1, он обладал высокой степенью совместимости с устройствами SCSI-1.

2.7.2.1. Термины SCSI.

"SCSI bus" - протокол обмена информацией с подключенными внешними устройствами SCSI. Одиночный обмен инициатора("initiator") с целью("target") может содержать до 8 слов ("phases"). Эти слова определяются целью (т.е. жестким диском). Текущее слово может быть определено путем просмотра пяти сигналов SCSI bus так, как это показано в таблице 1.1.
Некоторые контроллеры (в частности, недорогой контроллер Seagate) требуют переделки сигналов, переданных SCSI bus, другие автоматически используют эти низкоуровневые сигналы. Каждое из 8 слов будет подробно описано.

-SEL	-BSY	-MSG	-C/D	-I/O	PHASE
HI	HI	?	?	?	BUS FREE
HI	LO	?	?	?	ARBITRATION

I	I&T	?	?	?	SELECTION

T	I&T	?	?	?	RESELECTION
HI	LO	HI	HI	HI	DATA OUT
HI	LO	HI	HI	LO	DATA IN
HI	LO	HI	LO	HI	COMMAND
HI	LO	HI	LO	LO	STATUS
HI	LO	LO	LO	HI	MESSAGE OUT
HI	LO	LO	LO	LO	MESSAGE IN

I = сигнал инициатора; T = сигнал цели;
? = HI или LO

Таблица 1.1. Определение слов SCSI Bus.

Слово BUS FREE

Определяет SCSI bus как незанятый.

Слово ARBITRATION

Подается в случае, если устройство SCSI пытается установить контроль над SCSI bus.В этот момент устройство вносит свой SCSI ID в DATA BUS (установки SCSI bus).Например, если ID = 2, устройство задает дате 0x04.

В случае попытки обращения нескольких устройств одновременно, над целью устанавливает контроль устройство с наиболее высоким ID.Слово ARBITRATION использовалось также в стандарте SCSI-1.

Слово SELECTION

После установки контроля устройство, ставшее инициатором, заносит в дату протокола передачи SCSI ID цели. Если цель обнаруживается, она определяется, как занятая с помощью строки -BSY. Эта строка остается активной все то время, пока цель соединена с инициатором.

Слово RESELECTION

Протокол SCSI позволяет устройству отключаться от протокола передачи во время работы запроса. Когда устройство готово к продолжению обмена, оно вновь подключается к адаптеру. Слово RESELECTION идентично слову SELECTION, за исключением того, что оно используется отключенной целью для подключения к исходному инициатору. Драйверы, не поддерживающие RESELECTION, не имеют возможности раз'единения с целью SCSI.

Однако RESELECTION поддерживается почти всеми драйверами, так что многозадачные многозадачные устройства SCSI выполнять одновременно несколько задач, что уменьшает время обмена при запросах ввода/вывода.

Слово COMMAND

После этого слова отинициатора к цели может передаваться 6-ти, 10-ти и 12-ти байтная команда.

Слова DATA OUT и DATA IN

После этих слов осуществляется непосредственная передача информации между целью и инициатором. В случае DATA OUT, например, информация передается от адаптера к диску. DATA IN в таком случае осуществляет обратную передачу. Если команда SCSI требует передачи информации, слово не используется.

Слово STATUS

Это слово задается после завершения всех команд и дает возможность послать инициатору статусный байт. Существует 9 вариантов статусного байта (таблица 1.2). Заметим, что так как для статусного кода используются биты 1-5, статусный байт перед использованием маскируется 0x3e. Значения важнейших статусных кодов:

GOOD - операция выполнена успешно.

CHECK CONDITION - сообщение о случившейся ошибке.Команда REQUEST SENSE

может быть использована для получения более подробной информации об ошибке.

BUSY - устройство не может выполнить комаду. Это может случиться во время самотестирования или сразу после включения устройства.

Слова MESSAGE OUT и MESSAGE IN

Дополнительная информация передается между инициатором и целью. Этой информацией может быть статус посторонней команды или запрос

Value	Status
0x00	GOOD
0x02	CHECK CONDITION

0x04	CONDITION MET
0x08	BUSY
0x10	INTERMEDIATE
0x14	INTERMEDIATE-CONDITION MET
0x18	RESERVATION CONFLICT
0x22	COMMAND TERMINATED
0x28	QUEUE FULL

(После наложения маски 0x3e)

Таблица 1.2. Статусные коды SCSI.

для смены протокола. Слова MESSAGE OUT и MESSAGE IN могут неоднократно встречаться во время одной передачи.Если во время передачи доступно использование RESELECTION, драйвер должен поддерживать также слова SAVE DATA POINTERS, RESTORE POINTERS и DISCONNECT (сохранение и загрузка указателей, раз'единение). В SCSI-2 не все драйверы сохраняют указатели перед раз'единением.

2.7.3. Команды SCSI.

Каждая команда SCSI имеет длину 6,10 или 12 байт. нижеперечисленные команды должны быть качественно изучены будущими разработчиками драйверов SCSI:

REQUEST SENSE

Когда команда возвращает статус CHECK KONDITION, предусмотренная в Linux подпрограмма высокого уровня автоматически запрашивает более подробную информацию об ошибке, подавая команду REQUEST SENSE. Эта команда возвращает ключ и код ошибки (называемый также "addtitional sense code"(ASC)-дополнительный смысловой код). 16 возможных ключей описаны в таблице 1.3. Для получения информации о ASC, а также об ASCQ ("additional sense code qualiter"-дополнительный спецификатор смыслового значения кода), возвращаемом некоторыми драйверами, обращайтесь к стандарту SCSI[ANS] или к техническому руководству SCSI.

Ключ	Описание

0x00	NO SENSE (НЕТ ОТВЕТА)
0x01	RECOVERED ERROR (ВСКРЫТАЯ ОШИБКА)
0x02	NOT READY (НЕ ГОТОВ)
0x03	MEDIUM ERROR (СРЕДНЯЯ ОШИБКА)
0x04	HARDWARE ERROR (ОШИБКА АППАРАТНОГО ОБЕСПЕЧЕНИЯ)
0x05	ILLEGAL REQUEST (НЕПРАВИЛЬНЫЙ ЗАПРОС)

0x06 UNIT ATTENTION (ПРЕДУПРЕЖДЕНИЕ)
0x07 DATA PROTECT (ЗАЩИЩЕННАЯ ИНФОРМАЦИЯ)
0x08 BLANK CHECK (ПРОВЕРКА НА ОТСУТСТВИЕ ИНФОРМАЦИИ)
0x09 (Vendor specific error) (Ошибка инициатора)
0x0a COPY ABORTED (ПРЕКРАЩЕННОЕ КОПИРОВАНИЕ)
0x0b ABORTED COMMAND (ПРЕКРАЩЕННАЯ КОМАНДА)
0x0c EQUAL (ЭКВИВАЛЕНТНОСТЬ)
0x0d VOLUME OVERFLOV (ПЕРЕПОЛНЕНИЕ)
0x0e MISCOMPARE (НЕСООТВЕТСТВИЕ)
0x0f RESERVED (ЗАРЕЗЕРВИРОВАНО)

Таблица 3.1. Значения смысловых ключей.

TEST UNIT READY

Эта команда для тестирования статуса цели. Если цель может воспринимать команды среднего доступа (READ, WRITE),команда возвращает статус GOOD, в ином случае возвращается статус CHECK CONDITION и смысловой ключ NOT READY. Последнее обычно говорит о происходящем в настоящий момент самотестировании цели.

INQUIRY

Эта команда возвращает модель, производителя и тип устройства цели. Высокоуровневый Linux использует эту команду для определения разницы между оптическими, магнитными дисками и стримерами (высокоуровневый Linux не управляет принтерами, процессорами, или автоматическими устройствами).

READ и WRITE

Эти команды передачи информации от и к цели. До использования READ и WRITE вы должны убедиться в том, что ваш драйвер обладает возможностью поддержки простейших команд, таких, как TEST UNIT READY и INQUIRY.

2.7.4. С чего начинать ?

Авторы низкоуровневых драйверов устройств должны представлять себе, как управляет прерываниями ядро. Как минимум, вами должны быть изучены функции, которые разрешают (sti()) и запрещают(cli()) прерывания. Также для некоторых драйверов нужны функции определения времени вызова функций schedule(), sleepon() и wakeup(). В разделе 2.6 вы можете встретить более подробное описание этих функций.

2.7.5. Введение: сбор инструментов.

До того, как вы начнете писать драйвер SCSI для Linux, вам придется достать некоторые инструменты (ресурсы).

Самое важное - системный диск с системой Linux, желательно, жесткий диск с интерфейсом IDE, RLL или MFM. Во время разработки вашего SCSI придется много раз перестраивать ядро и перезапускать систему. Ошибки программирования могут привести к уничтожению информации на вашем диске SCSI, а также на посторонних носителях. Сохраняйте информацию на дисках!

Установленная система Linux может быть минимизирована: вы можете ограничиться библиотеками и утилитами компилятора GCC, текстовым редактором и текстом ядра. Также будут полезны дополнительные инструменты od, hexdump и less. Все эти программы свободно помещаются на диске размером 20 -30Мб.

Также вам потребуется подробная документация. Как минимум,

вам нужно описание используемого вами адаптера. Так как Linux распространяется свободно, и так как вы тоже пожелаете поделиться с другими вашими разработками, существуют нагласные соглашения, согласно которым, если если вы хотите обнародовать вашу подпрограмму, к ней должен быть приложен об'ектный код; однако на данном этапе это не всегда случается.

Вам будет полезно описание стандарта SCSI. Описание жесткого

диска обычно не требуется.

Прежде, чем начать, сохраните копии файлов hosts.h и scsi.h, а также одного из существующих драйверов ядра Linux. Это будет полезной рекомендацией во время написания.

2.7.6. Интерфейс SCSI в Linux.

Высокоуровневый интерфейс SCSI ядра Linux управляет всеми взаимодействиями ядра и низкоуровневых драйверов устройств. Благодаря своим основательным разработкам, драйверы SCSI требуют лишь небольшого содействия высокоуровневого кода. Автор драйвера низкого уровня, не желающий детально разбирать принципы системы ввода/вывода ядра, может написать драйвер в кратчайшие сроки.

Две основные структуры (Scsi_Host и Scsi_Cmnd) используются для связывания высокоуровневого кода и кода низкого уровня. Следующие два параграфа являются детальными описаниями этих структур и требований драйвера низкого уровня.

2.7.6. Структура Scsi_Host.

Структура Scsi_Host служит для описания драйвера низкого уровня коду высокого. Обычно это описание помещается в главный файл драйвера устройства в препроцессорные определения, как показано на рис. 1.1.

Структура Scsi_Host представлена на рис. 1.2 Каждое из полей будет дале подробно об'яснено.

```
#deflne FDOMAIN_16X0 { "Future Domain TMC-16x0",          \
                    fdomain-16x0_detect,              \
                    fdomain_16x0_info,               \
                    fdomain_16x0_command,             \
                    fdomain_16x0_queue,               \
                    fdomain_16x0_abort,               \

                    - 60 -

                    fdomain_16x0_reset,               \
                    NULL,                        \
                    fdomain_16x0_biosparam,           \
                    1, 6, 64, 1,0, 0}
#endif
```

Рис 1.1: Основной файл драйвера устройства.

```
typedef struct
{
 char             *name;
```

```
    int             (* detect) (int);
    const char      *(* info)(void);
    int             (* queuecommand)(Scsi_Cmnd *,
                    void (*done)(Scsi_Cmnd *));
    int             (* command) (Scsi_Cmnd *);
    int             (* abort) (Scsi_Cmnd *, int);
    int             (* reset) (void);
    int             (* slave_attach) (int, int);
    int             (* bios_param)(int, int, int []);
    int             can_queue;
    int             this_id;
    short unsigned int sg_tablesize;
    short           cmd_per_lun;
    unsigned        present:1;
    unsigned        unchecked_isa_dma:1;
} Scs i_Host;
```

Рис.1.2: Структура Scsi_Host.

2.7.7.1. Переменные в структуре Scsi_Host.

В общем случае переменные в структуре Scsi_Host не используются до вызова функции detect(), так как некоторым переменным может присваиваться значение лишь во время определения

(обнаружения) адаптера. Это происходит в случае, если драйвер может управлять несколькими устройствами с похожими свойствами, так что некоторые параметры структуры зависят от обнаруженного адаптера.

2.7.7.1.1. name

name содержит указатель на краткое описание host адаптера SCSI.

2.7.7.1.2. can_queue

can_queue содержит число невыполненных команд, которые может выполнить главный адаптер. В случае, если ваш драйвер поддерживает слово RESELECTION и использует прерывания, этой переменной присваивается значение 1.

2.7.7.1.3. this_id

Большинство главных адаптеров имеют особые, приписанные им SCSI ID. Эти SCSI ID, обычно равные 6 или 7, используются для реализации RESELECTION. this_id содержит SCSI ID адаптера. Если адаптеру не соответствует ID, этой переменной присваивается значение -1 (RESELECTION в таком случае не поддерживается).

2.7.7.1.4. sg_tablesize

Высокоуровневый код поддерживает метод "scatter-gather" (компановка - раз'единение) повышения эффективности обмена информацией с помощью комбинирования многих маленьких запросов в несколько больших. Так как большинство накопителей SCSI форматированы с прослойкой 1:1, что означает, что все сектора на одной дорожке располагаются последовательно, время, требуемое для выполнения слов ARBITRATION и SELECTION, не превышает времени

чередования секторов.Так что за один оборот диска может сработать

лишь один процесс, что приводит к скорости передачи 50Кб в секунду, в то время, как метод "scatter-gather" дает скорость около 500Кб в секунду.

sg_tablesize содержит максимально возможное число запросов в списке метода компановки-раз'единения. Если драйвер не поддерживает метод "scatter-gather", этой переменной присваивается значение SG_NONE. Если драйвер поддерживает неограниченное число групповых запросов, эта переменная принимает значение SG_ALL. В некоторых драйверах это число ограничивается предельным значением sg_tablesize, поддерживаемым адаптером. Некоторые адаптеры Adaptec требуют значение не более 16.

2.7.7.1.5. cmd_per_lun

SCSI стандарт поддерживает понятие "компановка команд". Компановка команд позволяет нескольким командам выстраиваться в порядке очередности к подаче на одно устройство. Эта переменная равна 1 в случае поддержки компановки команд. Однако на данный момент высокоуровневый код SCSI не использует преимуществ, предоставляемых этой возможностью.

Скомпанованные команды имеют фундаментальные отличия от команд одиночных (что описывается в переменной can_queue). Скомпанованные команды всегда предназначаются одной и той же цели и не обязательно используют слово RESELECTION.

Также компанованные команды исключают слова ARBITRATION, SELECTION и MESSAGE OUT после прохождения первой установленной в списке. В то же время одиночные команды могут посылаться на контролируемую цель и требуют слова ARBITRATION, SELECTION, MESSAGE OUT и RESELECTION.

2.7.7.1.6. present

Бит present устанавливается в случае обнаружения устройства.

2.7.7.1.7. unchecked_isa_dma

Некоторые host - адаптеры используют доступ к указанной памяти (Direct Memory Acess(DMA)) для чтения и записи блочной информации прямо в основаную память компьютера. Linux - система виртуальной памяти,имеющая возможность использовать более 16Мб физической памяти. На машинах с шиной ISA DMA ограничен шестнадцатью Мб физической памяти.

Если установлен бит unchecked_isa_dma, высокоуровневый код будет поддерживать информационный буфер адресацией ниже 16Мб физической памяти. Драйверы, не использующие DMA, устанавливают бит в 0. Драйверы, работающие с шиной EISA, всегда устанавливают этот бит также в 0, так как машины с EISA не позволяют доступа к DMA.

2.7.7.2. Функции структуры Scsi_Host.

2.7.7.2.1. detect()

Единственный аргумент функции detect() - "главный номер"(host number), индекс к переменным Scsi_hosts (массив типа struct Scsi_Host). Функция detect() возвращает ненулевое значение в случае обнаружения адаптера и нулевое в обратном случае.

Определение главного (host) адаптера должно производиться очень аккуратно. Обычно процесс начинается с просмотра области ROM в поисках "описания BIOS" главного адаптера.

В PS/AT и совместимых компьютерах адресное пространство с адреса 0xc0000 по 0fffff полностью распределено. Видео-BIOS компьютера расположена начиная с адреса 0xc0000, BIOS жесткого диска, если таковой существует, начинается с адреса 0xc8000. Во время загрузки PS/AT - совместимых компьютеров каждый 2-х килобайтный блок с адреса 0xc0000 до 0xf8000 проверяется на 2-х байтовую запись 0x55aa, которая свидетельствует о существовании

расширенного BIOS.

Описание BIOS обычно содержит серию из нескольких байт, идентифицирующих BIOS. Future Domain Bios, например, имеет описание:

FUTURE DOMAIN CORP. (C)
1986 - 1990 1800 - V2.07/28/89

Оно начинается с пятого байта от начала блока BIOS.

После обнаружения описания BIOS можно оттестировать функциональные качества адаптера особыми способами. Так как описания BIOS жестко закодированы в ядре, смена BIOS может привести драйвер к сбою. У пользователей адаптера SCSI исключительно в Linux может возникнуть желание отключить BIOS для ускорения начальной загрузки. По этим причинам должен существовать альтернативный метод определения адаптера.

Обычно каждый адаптер имеет несколько адресов ввода/вывода, использующихся для обеспечения связи. Иногда эти адреса жестко определены в драйвере, заставляя пользователей Linux, имеющих подобный адаптер, использовать определенную установку адресов. Другие драйверы сами определяют эти адреса, просматривая все возможные.

Обычно адаптер позволяет использовать 3 - 4 набора, руководствуясь переключателями на карте.

После определения адресов портов ввода/вывода адаптер может сам заявлять о себе. Эти тесты особенны для каждого адаптера, но имеют общие методы определения основного адреса BIOS (который затем может быть сравнен с адресм BIOS, найденным во время поиска определения BIOS)для проверки уникального номера, присущего карте. На машинах с шиной MCA каждому типу карты дается уникальный номер, благодаря которому ни один посторонний производитель не может использовать некоторые адаптеры. Future Domain, например, используют эту технологию на машинах ISA.

2.7.7.2.1.1. Запрос IRQ.

После определения detect() должен запросить канал DMA и пириоритет прерывания. Всего существует 16 приоритетов, называемых IRQ - от 0 до 15. Ядро поддерживает два метода установки обработчика IRQ: irqaction() и request_irq().

Функция request_irq() запрашивает два аргумента: номер IRQ и указатель на подпрограмму-обработчика. Часто устанавливаются параметры структуры sigaction с использованием irqaction(). Текст request_irq() показан на рисунке 1.3.

Определение функции irqaction():

int irqaction(unsigned int irq, struct sigaction *new) где первый параметр, irq, номер запрошенного IRQ, второй, new, структура, определение которой показано на рис. 1.4.

```
int request_irq( unsigned int irq, void (*handler)( int ))
{
  struct sigaction sa;

  sa.sa_handler  = handler;
  sa.sa_flags    = 0;
  sa.sa_mask     = 0;
  sa.sa_restorer = NULL;
  return irqaction( lrq, &sa );
}
```

Рис. 1.3: Функция request-irq().

```
struct sigaction
 {
   __sighandler_t sa_handler;
   sigset_t       sa_mask;
   int            sa_flags;

                    - 66 -

   void           (*sa_restorer) (void);
 };
```

Рис. 1.4: Структура sigaction

sa_handler в этой структуре указывает на подпрограмму обработчика прерываний, определяемую

```
void fdomain_16x0_intr( int irq )
```

где irq - номер IRQ, указывающий обработчику на пробуждение.

Переменная sa_mask используется как глобальный флаг подпрограммы irqaction().

Переменная sa_flags может быть установлена либо в 0, либо в SA_INTERRUPT. Если выбран 0, обработчик прерываний запускается при разрешенных посторонних прерываниях и возвращает значение через сигнальные функции обработчика. Эта установка используется для низких IRQ, таких, как таймер и клавиатура.

SA_INTEERUPT используется при больших ("быстрых") IRQ, например, при использовании упраляемых прерываниями драйверов жестких дисков. В последнем случае обработчик вызывается с запрещенными прерываниями.

Переменная sa_restorer в данный момент не задействована и

традиционно установлена в NULL.

Функции request_irq() и irqaction() будут возвращать нуль, если IRQ успешно поставлен в соответствие определенному обработчику прерываний. Ненулевые возвращаемые значения могут быть следующими:

- EINVAL Запрошенный IRQ больше 15, или обработчику прерываний был подан указатель на NULL.
- EBUSY Запрошенный IRQ уже занят другим обработчиком прерываний. Эта ситуация не возникает в случае использования

panic().

Ядро использует Intel "распределение" для установки IRQ, запрашиваемых функцией irqaction().

2.7.7.2.2. Запрос канала DMA.

Некоторые адаптеры SCSI используют DMA лдя помещения больших информационных блоков в памяти. Так как процессор не управляет передачей информации в блоки DMA, передача осуществляется быстрее передачи, контролируемой процессором и позволяет последнему работать в это время над другой задачей.

Адаптеры используют определенные каналы DMA. Эти каналы определяются функцией detect() и запрашиваются ядром с помощью request_dma(). Эта функция получает номер канала DMA как свой единственный параметр и возвращает нуль, если канал DMA успешно подключен. Другие возможные возвращаемые значения:

- EINVAL Запрошенный канал DMA имеет номер больше 7.
- EBUSY Запрошенный канал DMA уже используется. Этой ситуация может привести к неудовлетворения запроса SCSI. В этом случае также можно использовать panic().

2.7.7.2.3. info()

Функция info() возвращает указатель на статическую область, содержащую описание драйвера низкого уровня. Это описание содержится в переменной-указателе name и выводится во время загрузки.

2.7.7.2.4. queuecommand()

Функция queuecommand() осуществляет запуск команды SCSI адаптером, затем завершает работу. По завершению команды вызывается функция done() с указателем на структуру Scsi_Cmnd в

качестве параметра. Это позволяет команде SCSI запуститься в режиме прерывания. Перед завершением работы функция queuecommand() должна выполнить следующие операции:

1. Сохранить указатель на структуру Scsi_Cmnd.
2. Сохранить указатель на функцию done() в качестве поля Scsi_done() в структуре Scsi_Cmnd. См. раздел 2.7.7.2.5 для более подробной информации.
3. Установить специальные переменные в Scsi_Cmnd, требуемые драйвером.
4. Запустить команду SCSI. Для расширенных host-адаптеров это может

быть простейшая засылка команды в "mailbox" host-адаптера. Для менее "мудрых" адаптеров используется сначала слово ARBITRATION.

Функция queuecommand() вызывается лишь в случае ненулевой переменной can_queue (см. 2.7.7.1.2). В ином случае для всех запросов используется функция command(). В случае успеха функция queuecommand() возвращает 0. (Высокоуровневый код SCSI игнорирует это возвращаемое значение).

2.7.7.2.5. done()

Функция done() вызывается после завершения команды SCSI. Единственный параметр, этой функции - указатель на структуру Scsi_Cmnd, используемую прежде функцией queuecommand(). Перед вызовом функции done() должна быть правильно установлена переменная result. Она имеет тип 32-битного целого, каждый байт которого имеет свое значение:

Байт 0 - Содержит код SCSI STATUS, как описано в 2.7.2.1.
1 - Содерит SCSI MESSAGE, как описано в 2.7.2.1
2 - Содержит возвращаемый код host адаптера. Этим кодам присваивается значения в scsi.h:

DID_OK Ошибок не обнаружено
DID_NO_CONNECT SCSI SELECTION не может передаться из-за отсутствия устройства по указанному адресу.
DID_BUS_BUSY Ошибка SCSI ARBITRATION

DID_TIME_OUT Произошла приостановка работы процесса по неизвестной причине, возможно во время SELECTION или в ожидании RESELECTION.
DID_BAD_TARGET SCSI ID цели такой-же как ID адаптера
DID_ABORT Высоко-уровневый код вызывает низко-уровневую функцию abort().
DID_PARITY Ошибка SCSI PARITY
DID_ERROR Ошибка, не поддающаяся распознанию (к примеру ошибка самого адаптера)
DID_RESET Высоко-уровневый код вызывает низко-уровневую функцию reset()
DID_BAD_INTR Возникновение непредвиденного прерывания, которым не возожно управлять.

Возврат DID_BUS_BUSY будет пытаться запустить команду еще раз, в то время как DID_NO_CONNECT сбросит команду.

Байт 3 Этот байт предназначен для возвращения кода высокого уровня и устанавливается низким уровнем в 0.

В настоящий момент драйвера низкого уровня не описывают сообщения об ошибках, поэтому легче всего для вас найти их определения в scsi.c вместо того чтобы исследовать существующие драйвера.

2.7.7.2.6 command()

Функция command() запускает команду SCSI и возвращается после ее завершения. Когда был создан оригинал кода SCSI, в нем не осуществлялась поддержка драйверов управляемых прерываниями. Старые драйвера менее эфеективны чем созданные на данный момент драйвера управляемые прерываниями, но более просты в написании. Для новых драйверов эта функция заменена на queuecommand(), как

описано в следующей программе:

```
ststic volatile int internal_done_flag    = 0;
static volatile int internal_done_errcode = 0;
```



```
static void        internal_done(Scsi_Cmnd *SCpnt);
{
  internal_done_errcode = SCpnt->result;
  ++internal_done_flag;
}
int aha1542_command(Scsi_Cmnd *SCpnt)
{
  aha1542_queuecommand (SCpnt, internal_done );

  while(!internal_done_flag);
  internal_done_flag = 0;
  return internal_done_errcode;
}
```

Возвращаемое значение - то же, что и в переменной result в структуре Scsi_Cmnd. См 2.7.7.2.5 и 2.7.8.

2.7.7.2.7 abort()

Высокоуровневый код SCSI управляет всеми преостановками. Это освобождает драйвер низкого уровня от распределения времени между запросами на периоды исполнения для различных устройств (преостановка работы стримера может быть на много дольше, нежели преостановка жесткого диска).

Функция abort() используется отключения запроса текущей команды SCSI определенной указателем Scsi_Cmnd. После установки переменной result в структуре Scsi_Cmnd функция abort() возвращает нулевое значение.

Если code, второй параметр функции abort(), равен нулю, тогда result устанавливается в DID_ABORT. В ином случае result равн code (обычно это DID_TIM_OUT и DID_RESET).

На данный момент ни один из драйверов низкого уровня не может правильно отключать комманды SCSI. Инициатор должен запрашивать словом MESSEGE OUT цель, для решения этой задачи. Затем инициатор посылает ABORT цели.

2.7.7.2.8 reset()

Функция reset() служит для выгрузки шины SCSI. После выгрузки ни комманда SCSI не будет выполняться, возвращая код DID_RESET.

В настоящий момент ни один из драйверов низкого уровня не может правильно пользоваться этой операцией. Для правильной выгрузки инициатор запрашивает (посылая -ATN) MESSAGE OUT, и подает цели команду BUS DEVICE RESET. Можно также дать команду SCSI RESET, спослав -RST, заставляющую все цели отключиться.

После выгрузки будет полезно удалить также протокол связи.

2.7.7.2.9 slave_attach()

Функция на данный момент не описана. Используется для установки связи между host адаптером и целью. Связь подразумевает обмен парой SYNCHRONOUS DATA TRANSFER REQUEST между целью и инитатором. Обмен возникает при условиях:

- Устройство SCSI поддерживающее обмен не соединяется с устройством, после получения сигнала отбоя (RESET).
- Устройство SCSI также не может быть соединено с другим в случае если оно получило сообщени BUS DEVICE RESET.

2.7.7.2.10 bios_param()

Linux поддерживает систему деления жеского диска MS-DOS. Каждый диск содержит "таблицу частей" в которой определено как диск разбит на логические диски. Обработка информации в таблице требует знания о размере диска в циллиндрах, головках и секторах. Диски SCSI скрывают свои физические параметры и логически представляются списком секторов.

Для получения совместимости с MS-DOS, host адаптер SCSI "лжет" о своих физических параметрах. Так что вместо параметров физических устройство SCSI посавляет "логические параметры".

Linux нуждается в определении "логических параметров" для правильного изменеия таблицы. В сущности метода конвертации логических параметров в физические не существует. Функция bios_param() представляет собой осуществление доступа к параметрам.

Параметр size содержит размер диска в секторах. Некоторые host адаптеры располагают формулой для подсчета логических параметров исходя из этой цифры, иным приходится хранить информацию в таблицах доступных драйверу.

Для обеспечения этого доступа, параметр dev хранит информацию о номере устройства. Два макроса описанные в linux/fs.h осуществляют определение этого значения: MAJOR(dev) - основного номера устройства и MINOR(dev) - определение подномера. Это те-же номера, используемые при выполнении стандартной команды Linux mknod, служащей для создания устройства в каталоге /dev. Параметр info указывает на массив целых, заполняемый функцией bios_param() до возвращения:

info[0] Количество головок
info[1] Количество секторов на циллиндр
info[2] Количество циллиндров

Информация в info является "логиескими парвметрами" устройстваб, используемые методами MS-DOS как физические.

2.7.8 Структура Scsi_Cmnd

Структура Scsi_Cmnd, как показано на рисунке 1.6 использует код высокого уровня для спецификации комманды SCSI для запуска низко-уровневым кодом. Множество переменных в структуре Scsi_Cmnd могут не использоваться в драйвере низкого уровня.

2.7.8.1 Зарезервированная область

2.7.8.1.1 Информационные переменные.

host - индекс массива scsi_hosts.

target - содержит ID цели команды SCSI. Эта информация важна в случае поддержки целью многозадачности.

cmnd - массив байт, содержащий текущую команду SCSI. Эти байты посылаются цели посте строки COMMAND. cmnd[0] - код команды SCSI. Макро COMMAND_SIZE, определенный в scsi.h используется для определения длины команды.

result - код результата запроса SCSI. См. 2.7.7.2.5 для более подробной информации об этой переменной. Она должна быть верно установлена до возврата низкоуровневых подпрограмм.

2.7.8.1.2 Список Разветвления - компановки. (Scatter-gather)

use_sg содержит количество кусков обрабатываемых scatter-gather. Если use_sg = 0, тогда request_buffer указывает на буфер данных команды SCSI, и размер буфера содержится в request_bufferlen. В ином случае request_buffer указывает на массив структур scatterlist и use_sg идентифицирует количество структур в массиве. Использование request_buffer довольно тяжело.

Каждый элемент массива scatterlist содержит компоненты address и length. Если флаг unchecked_isa_dma в структуре scsi_Host установлен в 1, адрес гарантированно попадает в область первых 16Мб физической памяти. Одной SCSI командой можно в таком случае передать большое количество информации, при этом длина большого куска равна сумме длин всех малых.


```
typedef struct scsi_cmnd
{
  int             host;
  unsigned char   target;
                  lun;
                  index;
  struct scsi_cmnd *next,
                  *prev;

  unsigned char   cmnd[10];
  unsigned        request_bufflen;
  void            *request_buffer;

  unsigned char   data_cmnd[10];
  unsigned short  use_sg;
  unsigned short  sglist_len;
  unsigned        bufflen;
  void            *buffer;

  struct request  request;
  unsigned char   sense_buffer[16];
  int             retries;
  int             allowed;
  int             timeout_per_command,
```

```
                timeout-total,
                timeout;
    unsigned char    internal_timeout;
    unsigned         flags;

    void (*scsi_done)(struct scsi_cmnd *);
    void (*done)(struct scsi_cmnd *);

    Scsi_Pointer     Scp;
    unsigned char    *host_schribble;
    int              result;

} Scsi_Cmnd;
```


Рис. 1.6: Структура Scsi_Cmnd.

2.7.8.2. Рабочие области.

В зависимости от возможостей и требований host адаптера, список scatter- gather может управляться различными способами. Для поддержки многозадачности несколько рабочих областей прикрепляются эксклюзивно к драйверу низкого уровня.

2.7.8.2.1 Указатель scsi_done().

Указатель должен быть установлен на функцию done() в функции queuecommand(). Других использований этому указатенлю не предусмотрено.

2.7.8.2.2 Указатель host_scribble

Код высокого уровня поддерживает пару функций распределения памяти - scsi_malloc() и scsi_free(), которые гарантируют возврат физической памяти из первых 16Мб. Эта память также подходит для использования DMA.

Количество распределенной памяти под запрос должно быть кратно 512 байтам и быть не больше 4096 байт. Общее количество памяти доступной scsi_malloc() определяется арифметической функцией с тремя аргументами, находящиеся в Scsi_Host - переменные sg_tablesize,cmd_per_lun и unchecked_isa_dma.

Указатель host_scribble указывает на область досупной памяти выделенной scsi_malloc(). Драйвер SCSI низкого уровня обладает возможностью управления этим указателем и соответствующей ему памяти, а также возможностью очистки ненужной информации в памяти.

2.7.8.2.3 Структура Scsi_Pointer.

Переменная SCp, структура типа Scsi_Pointer, описана на рисунке ниже. Переменные этой структуры могут быть использованы любыми средствами в драйверах низкого уровня. Как обычно buffer здесь указывает на текущую позицию scatterlist, buffer_residual показывает количество элементов находящихся в scatterlist, ptr - указатель на буффер, а this_residual - число символов для

передачи. Некоторые host адаптеры требуют эту информацию, некоторые игнорируют ее.

Второй набор переменных содержит информацию о статусе SCSI, различные указатели и флаги.

```
typedef struct scsi_pointer
{
  char          *ptr;
  int           this_residual;
  struct scatterlist *buffer;
  int           buffers_residual;

  volatile int   Status;
  volatile int   Message;
  volatile int   have_data_in;
  volatile int   sent_command;
  volatile int   phase;
}Scsi_Pointer;
```

Глава 3.

Файловая система /proc.

Файловая система proc представляет собой интерфейс к нескольким структурам данных ядра, которые работают также как и файловая система. Вместо того, чтобы каждый раз обращаться в

/dev/kmem и искать путь к определению местонахождения какой-либо информации, все приложения читают файлы и каталоги из /proc. Таким образом все адреса структур данных ядра заносятся в /proc во время компиляции ядра, и программы использующие proc не могут перекомпилироваться после этого.

Существует возможность поддерживать файловую систему proc вне /proc, но при этом она теряет эффективность, поэтому в данном труде эта возможность не рассматривается.

3.1 Каталоги и файлы /proc.

Эта часть довольно сильно урезана, однако на данный момент авторы не могут предложить ничего более существенного.

В /proc существует подкаталог для каждого запускаемого процесса, названый по номеру pid процесса. Эти директории более подробно описаны ниже. Также в /proc присутствует несколько других каталогов и файлов:

self Этот файл имеет отношение к процессам имеющим доступ к файловой системе proc, и идентифицированным в директоиях названных по id процессов осуществляющих контроль.

kmsg Этот файл используется системным вызовом syslog() для регистрации сообщений ядра. Чтение этого файла может осуществляться лишь одним процессом имеющим привилегию superuser. Этот файл не доступен для чтения при регистрации с помощью вызова syslog().

loadavg Этот файл содержит числа подобно:

0.13 0.14 0.05
Эти числа являются результатом комманд uptime и подобных, показывающих среднее число процессов пытающихся запуститься в одно и то же врямя за последнюю минуту, последние пять минут и последние пятнадцать.

meminfo Файл содержит обзор выходной информации программы free. Содержание

его имеет следующий вид:

	total:	used:	free:	shared:	buffers:
Mem:	7528448	7344128	184320	2637824	1949696
Swap:	8024064	1474560	6549504		

Помните что данные числа представлены в байтах! Linus написала версию free осуществляющую вывод как в байтах, так и в кидобайтах в зависимости от ключа (-b или -k). Она находится в пакете procps в tsx-11.mit.edu. Также помните, что что своп-файлы используются нераздельно - все пространство памяти доступное для своппинга суммируется.

uptime Файл содержит время работы систмы вцелом и идеализированное время затрачивоемое системой на один процесс. Оба числа представлены в виде десятичных дробей с точностью до сотых секунды. Точность до двух цифр после запятой не гарантируется на всех архитектурах, однако на всех подпрограммах Linux даются достаточно точно используя удобные 100-Гц цасы. Этот файл выглядит следующим образом:

604.33 205.45

В этом случае система функционирует 604.33 секунды, а время затрачиваемое на идеальный прцесс равно 204.45 секунд.

kcore Этот файл представляет физическую память данной системы, в формате аналогичном "основному файлу"(core file). Он может быть использован отладчиком для проверки значений переменных ядра. Длина файла равна длине физической памяти плюс 4кб под заголовок.

stat Файл stat отображает статистику данной системы в формате ASCII.
Пример:

```
cpu  5470 0 3764 193792
disk 0 0 0 0
page 11584 937
swap 255 618
intr 239978
ctxt 20932
btime 767808289
```

Значения строк:

cpu Четыре числа сообщают о количестве тиков за время

работы системы в пользовательском режиме, в пользовательском режиме с низким приоритетом, в системном режиме, и с идеальной задачей. Последнее число является стократным увеличением второго значения в файле uptime.

disk Четыре компонеты dk_drive в структуре kernel_stat в данный момент незаняты.

page Количество страниц введенных и исключенных системой.

swap Количество своп-страниц введенных и исключенных системой.

intr Количество прерываний установленных при загрузке системы.

ctxt Номер подтекста выключающий систему.

btime Время в секундах отсчитываемое сначала суток.

modules Список модулей ядра в формате ASCII. Формат файла изменяется

от версии к версии, поэтому пример здесь неприводится. Окончательно формат установится, видимо со стабилизацией интерфейса самих модулей.

malloc Этот файл присутствует в случае, если во время компиляции ядра была описана строка CONFIG_DEBUG_MALLOC.

version Файл содержит строку идентифицирующую версию работающего в данный момент Linux.

Linux version 1.1.40 (johnson@nigel) (gss version 2.5.8) #3 Sat Aug 6

Строка содержит версию Linux, имя пользователя и владельца осуществлявшего компиляцию ядра, версию gcc, количество предыдущих компиляций владельцем, дата последней компиляции.

net Этот каталог содержит три файла, каждый из которых представляет статус части уровня работы с сетями в Linux. Эти файлы представляют двоичные структуры и они визуально нечитабельны, однако стандартный набор сетевых прграмм использует их. Двоичные структуры читаемые из этих файлов определены в . Файлы называются следующим образом:

unix
arp

route
dev
raw
tcp
udp - К сожалению, автор не располагает подробной информацией об устройстве файлов, поэтому в данной книге оно не описывается.

Каждый из подкаталогов прцессов (прнумерованных и имеющих собственный каталог) имеет свой набор файлов и подкаталогов. В подобном подкаталоге присутствует следующий набор файлов:

cmdline Содержит полную коммандную строку процесса, если он полнось не выгружен или убит. В любом из последних двух случаев файл пуст и чтение его поводит к тому-же результату, что и чтение пустой строки. Этот файл содержит в коце нулевой символ.

cwd Компановка текущего каталога данного процесса. Для обнаружения cwd процесса 20, сделайте следующее:
(cd /proc/20/cwd; pwd)

environ Файл содержит требования процесса. В файле отсутствуют переводы строки: в конце файла и между записями находятся нулевые символы. Для вывода требоаний процесса 10 вы должны сделать:
cat /proc/10/environ | tr "\000" "\n"

exe Компановка запускаемого прцесса. Вы можете набрать:
/proc/10/exe
для перезапуска процесса 10 с любыми изменениями.

fd Подкаталог содержащий запись каждого файла открытого процесса, названого именем дескриптора, и скомпанованного как фактический файл. Программы работающие с файлами, но не использующие стандартный ввод-вывод, могут быть переопределены с использованием флагов -i (определение входного файла), -o (определение выходного файла):
... | foobar -i /proc/self/fd/0 -o /proc/self/fd/1 |...
Помните, что это не будет работать в программах осуществляющих поиск файлов, так как файлы в каталоге fd поиску не поддаются.

maps Файл содержащий список распределенных кусков памяти, используемых процессом. Общедоступные библиотеки распределены в памяти таким образом, что на каждую из них отводится один отрезок памяти. Некоторые процессы также используют память для других целей.
Пример:

```
00000000 - 00013000 r-xs 00000400 03:03 12164
00013000 - 00014000 rwxp 00013400 03:03 12164
00014000 - 0001c000 rwxp 00000000 00:00 0
bffff000 - c0000000 rwxp 00000000 00:00 0
```

Первое поле записи определяет начало диапазона распределенного куска памяти.
Второе поле определяет конец диапазона отрезка.
Третье поле содержит флаги:

```
r - читаемый кусок, - нет.
w - записываемый,   - нет.
x - запускаемый,    - нет.
s - общедоступный, p - частного пользования.
```

Четвертое поле - смещение от которого происходит распределение.
Пятое поле отображает основной номер:подномер устройства распределяемого файла.
Пятое поле показывает число inode распределяемого файла.

mem Этот файл не идентичен устройству mem, несмотря на то, что они имет одинаковый номер устройств. Устройство /dev/mem - физическая память перед выполнением переадресации, здесь mem - память доступная процессу. В данный момент она не может быть перераспределена (mmap()), поскольку в ядре нет функции общего перераспределения.

root указатель на корневой каталог процесса. Полезен для программ использующих chrroot(), таких как ftpd.

stat Файл содержит массу статусной информации о процессе. Здесь в порядке представления в файле описаны поля и их формат чтения функцией scanf():

```
pid %d      id процесса.
comm (%s)   Имя запускаемого файла в круглых скобках. Из него
```



```
            видно использует-ли процесс своппинг.
state %c    один из символов из набора "RSDZT", где:
              R - запуск
              S - заморозка в ожидании прерывания
              W - заморозка с запрещением прерывания (в частности
                  для своппинга)
              Z - исключение процесса
              T - приостановка в определенном состоянии
ppid %d     pid процесса
pgrp %d     pgrp процесса
session %d
tty %d      используемая процессом tty.
tpgid %d    pgrp процесса который управляет tty соединенным
            с текущим процессом.
flags %u    Флаги процесса. Каждый флаг имеет набор битов
min_flt %u  Количество малых сбоев работы процесса, которые не
            требуют загрузки с диска страницы памяти.
cmin_flt %u Количество малых сбоев в работе процесса и его сыновей
maj_flt %u  Количество существенных сбоев в работе процесса,
            требующих подкачки страницы памяти.
cmaj_flt %u Количество существенных сбоев процесса и его сыновей.
```

utime %d Количество тиков, со времени распределения работы процесса
в пространстве пользователя.
stime %d Количество тиков, со времени распределения работы процесса
в пространстве ядра.
cutime %d Количество тиков, со времени распределения работы
процесса и его сыновей в пространстве пользователя.
cstime %d Количество тиков, со времени распределения работы проццесса
и его сыновей в пространстве ядра.
counter %d Текущий максимальный размер в тиках следующего периода
работы процесса, в случае его непосредственной деятельности,
количество тиков до завершения деятельности.
priority %d стандартное UN*X-е значение плюс пятнадцать. Это
число не может быть отрицательным в ядре.
timeout %u Время в тиках, следующего перерыва в работе процесса.
it_real_value %u
Период времени в тиках, по истечении которого процессу
передается сигнал SIGALARM (будильник).

start_time %d
Время отсчитываемое от момента загрузки системы, по
истечении которого начинает работу процесс.
vsize %u Размер виртуальной памяти.
rss %u Установленный размер резидентной памяти - количество
страниц используемых процессом, содержащихся в реальной
памяти минус три страницы занятые под управление.
Сюда входят стековые страницы и информфционные.
Своп-страницы, страницы загрузки запросов не входят в
данное число.
rlim %u Предел размера процесса. По усмотрению 2Гб.
start_code %u
Адрес выше которого может выполняться текст программы.
end_code %u Адрес ниже которого может выполняться текст программы.
start_stack %u
Адрес начала стека.
kstk_esp %u Текущее значение указателя на 32-битный стек, получаемый
в стековой странице ядра для процесса.
kstk_eip %u Текущее значение указателя на 32-битную инструкцию,
получаемую в стековой странице ядра для процесса.
signal %d Побитовая таблица задержки сигналов (обычно 0)
blocked %d Побитовая таблица блокируемых сигналов (обычно 0,2)
sigignore %d
Побитовая таблица игнорируемых сигналов.
sigcatch %d Побитовая таблица полученных сигналов.
wchan %u "Канал" в котором процесс находится в состоянии
ожидания. Это адрес системного вызова, который
можно посмотреть в списке имен, если вам нужно
получить строковое значение имени.

statm Этот файл содержит специальную статусную информацию, занимающую немного больше места, нежели информация в stat, и используемую достаточно редко, чтобы выделить ее в отдельный файл. Для создания каждого поля в этом файле, файловая система proc должна просматривать каждый из 0x300 составляющих в каталоге страниц и вычислять их текущее состояние.

Описание полей:

size %d Общее число страниц, распределенное под процесс
в виртуальной памяти, вне зависимости физическая
она или логическая.

resident %d Общее число страниц физической памяти используемых процессом. Это поле должно быть численно равно полю rss в файле stat, однако метод подсчета значения отличается от примитивного чтения структуры процесса.

trs %d Размер текста в резидентной памяти - общее количество страниц текста(кода), принадлежащих процессу, находящихся в области физической памяти. Не включает в себя страницы с общими библиотеками.

lrs %d Размер резидентной памяти выделенный под библиотеки - общее количество страниц, содержащих библиотеки, находящихся в верхней памяти.

drs %d Размер резидентной области используемой процессом в физической памяти.

dt %d Количество доступных страниц памяти.

3.2 Структура файловой системы /proc.

Файловая система proc интересна тем, что в реальной структуре каталогов не существует файлов. Функцияии, которые поводят гигантское количество операции по чтению файла, получению страницы и заполнеию ее, выводу результата в пространство памяти пользователя, помещаются в определенные vfs-структуры.

Одним из интереснейших свойств файловой системы proc, является описание каталогов процессов. По существу, каждый каталог процесса имеет свой номер inode своего PID помещеннающий 16 бит в 32 - битный номер больше 0x0000ffff.

Внутри каталогов номер inode перезаписывается, так как верхние 16 бит номера маскируется выбором каталога.

Другим не менее интересным свойством, отличающим proc от других файловых систем в которых используется одна структура file_operations для всей файловой системы, введены различные

структуры file_operations записываемые в компонент файловой структуры f_ops вбирающий в себя функции нужные для просмотра конкретного каталога или файла.

3.3 Програмирование файловой системы /proc.

Предупреждение: Текст фрагментов программ, представленных здесь, может отличаться от исходников вашего ядра, так как файловая система /proc видоизменилась со времени создания этой книги, и видимо, будет видоизменяться далее. Структура root_dir со времени написания данного труда увеличилась вдвое.

В отличие от других файловых систем, в proc не все номера inode уникальны. Некоторые файлы определены в структурах

```
static struct proc_dir_entry root_dir[] = {
  { 1,1,"." },
  { 1,2,".." },
  { 2,7,"loadavg" },
  { 3,6,"uptime" },
  { 4,7,"meminfo" },
  { 5,4,"kmsg" },
  { 6,7,"version" },
  { 7,4,"self" }, /* смена номера inode */
  { 8,4,"net" }
};
```

Некоторые файлы динамически создаются во время чтения файловой системы. Все каталоги процесса имеют номера inode, чей идентификационный номер помещается в 16 бит, но файлы в этих каталогах переиспользуют малые номера inode (1-10), помещаемые во время работы процесса в pid процесса. Это происходит в inode.c с помощью аккуратного переопределения структур inode_operations.

Большинство файлов в корневом каталоге и в кадом подкаталоге процесса, доступных только для чтения используют простейший интерфейс поддерживаемый структурой array_inode_operations,

находящейся в array.c.

Такие каталоги, как /proc/net, имеют свой номер inode. К примеру сам каталог net имеет номер 8. Файлы внутри этих каталогов имеют номера со 128 по 160, определенные в inode.c и для просмотра и записи таких файлов нужно специальное разрешение.

Внесение файла является несложной задачей, и остается в качестве упражнения читателю. Если предположить, что каталог в который вносится файл не динамический, как к примеру каталоги процессов, приведем следующий алгоритм:

1.Выберите уникальный диапазон номеров inode, дающий вам приемлимый участок памяти для помещения. Зытем справа от строки:

```
if (!pid) {/* в каталоге /proc/ */
```

сделайте запись идентичную следующей

```
if ((ino>=128) && (ino<=160) { /*Файлы внутри /proc/net*/
  inode->i_mode = S_IFREG | 0444
  inode->i_op = &proc_net_inode_operations;
  return;
}
```

изменив ее для опрерации нужной вам. В частности, если вы работаете в диапазоне 200-256 и ваши файлы имеют номера inode 200,201,202, ваши каталоги имеют номера 204 и 205, а номер inode 206 имеет имеющийся у вас файл читаемый лишь из корневрго каталога, ваша запись будет выглядеть следующим образом:

```
if ((ino >= 200)&&(ino <= 256)) { /* Файлы в /poc/foo */
  switch (ino) {
    case 204:
    case 205:
        inode->i_mode = S_IFDIR | 0555;
        inode->i_op = &proc_foo_inode_oprations;
        break;
    case 206:
        inode->imode = S_IFREG | 0400;
        inode->i_op = &proc_foo_inode_operations;
        break;
    default:
```



```
        inode->i_mode = S_IFREG | 0444;
        inode->i_op = &proc_foo_inode_operations;
        break;
  }
  return;
}
```

2.Найдите место определения файлов. Если ваши файлы помещаются в подкатаог каталога /proc, вам надо найти следующие строки в файле root.c:

```
static struct proc_dir_entry root_dir[] = {
{ 1,1,"." },
```

```
        { 1,2,".." },
        { 2,7,"loadavg" },
        { 3,6,"uptime" },
        { 4,7,"meminfo" },
        { 5,4,"kmsg" },
        { 6,7,"version" }
        { 7,4,"self" }, /* смена inode */
        { 8,4,"net" }
      };
```

Затем вам следут подставить в эту запись после строки:

```
        { 8,4,"net" }
```

подставиь строку:

```
        { 9,3,"foo"}
```

Таким образом, вы предусматриваете новый каталог в inode.c, и текст:

```
      if (!pid) { /* not a process directory but in /proc/ */
        inode->i_mode = S_IFREG | 0444;
        inode->i_op = &proc_array_inode_operations;
        switch (ino)
          case 5:
            inode->i_op = &proc_array_inode_operations;
            break;
          case 8: /* for the net directory */
            inode->i_mode = S_IFDIR | 0555;
            inode->i_op = &proc_net_inode_operations;
            break;
```



```
          default:
            break;
        return;
      }
```

становится

```
      if (!pid) { /* not a process directory but in /proc/ */
        inode->i_mode = S_IFREG | 0444;
        inode->i_op = &proc_array_inode_operations;
        switch (ino)
          case 5:
            inode->i_op = &proc_array_inode_operations;
            break;
          case 8: /* for the net directory */
            inode->i_mode = S_IFDIR | 0555;
            inode->i_op = &proc_net_inode_operations;
            break;
          case 9: /* for the foo directory */
            inode->i_mode = S_IFDIR | 0555;
            inode->i_op = &proc_foo_inode_operatlons;
            break;
          default:
            break;
        return;
      }
```

3.Затем вам нужно обеспечить содержание файла в каталоге foo. Создайте файл proc/foo.c следуя указанной модели.

```
    /*
     * linux/fs/proc/foo.c
     *
     * Copyright (C) 1993 Lunus Torvalds, Michael K. Johnson, and Your N. Here
     *
```

```
* proc foo directory handling functions
*
* inode numbers 200 - 256 are reserved for this directory
```



```
* (/proc/foo/ and its subdirectories)
*/

#include
#include
#include
#include
#include

static int proc_readfoo(struct inode *, struct file *, struct dirent *, int);
static int proc-lookupfoo(struct inode *,const char *,int,struct inode **);
static int proc_read(struct inode * inode, struct file * file,
                     char * buf, int count),
static struct file_operations proc_foo_operations = {
                        NULL,                /* lseek - default */
                        proc_read,             /* read */
                        NULL,                /* write - bad */
                        proc_readfoo,           /* readdir */
                        NULL,                /* select - default */
                        NULL,                /* ioctl - default */ /* danlap */
                        NULL,                /* mmap */
                        NULL,                /* no special open code */
                        NULL                 /* no special release code */
   };

/*
 * proc directories can do almost nothing..
 */
 struct inode_operations proc_foo_inode_operations = {
                        &proc_foo_operations,  /* default foo directory file-ops */
                        NULL,                /* create */
                        proc_lookupfoo,        /* lookup */
                        NULL,                /* link */
                        NULL,                /* unlink */
                        NULL,                /* symlink */
                        NULL,                /* mkdir */
                        NULL,                /* rmdir */
                        NULL,                /* mknod */
```



```
                        NULL,                /* rename */
                        NULL,                /* readlink */
                        NULL,                /* follow_link */
                        NULL,                /* bmap */
                        NULL,                /* truncate */
                        NULL                 /* permission */
    } ;

 static struct proc_dir_entry foo_dir[] = {
                          { 1,2,".." },
                          { 9,1,"." },
                          { 200,3,"bar" },
                          { 201,4,"suds" },
                          { 202,5,"xyzzy" },
                          { 203,3,"baz" },
                          { 204,4,"dir1" },
                          { 205,4,"dir2" },
                          { 206,8,"rootfile" }
```

```
};

#define NR_FOO-DIRENTRY ((sizeof (foo_dir))/(sizeof (foo_dir[0])))

unsigned int get_bar(char * buffer);
unsigned int get_suds(char * buffer);
unsigned int get_xyzzy(char * buffer);
unsigned int get_baz(char * buffer);
unsigned int get_rootfile(char * buffer);


static int proc_read(struct inode * inode, struct file * file,
                     char * buf, int count)
 {
   char * page;
   int length;
   int end;
   unsigned int ino;
```



```
   if (count < 0)
          return -EINVAL;
   page = (char *) get_free_page(GFP-KERNEL);
   if (!page)
           return -ENOMEM;
   ino = inode->i_ino;
   switch (ino) {
            case 200:
                  length = get_bar(page);
                  break;
            case 201:
                  length = get_suds(page);
                  break;
            case 202:
                  length = get_xyzzy(page);
                  break;
            case 203:
                  length = get_baz(page);
                  break;
            case 206:
                  length = get_rootfile(page);
                  break;
            default:
                  free_page((unsigned long) page);
                  return -EBADF;
              }
           if (file->f_pos >= length) {
                  free_page ((unsigned long) page);
                  return 0;
                }
           if (count + file->t_pos > length)
                    count = length - file->f_pos;
           end = count + file->f_pos;
           memcpy_tofs(buf, page + file->f_pos, count);
           free_page((unsigned long) page);
           file->f_pos = end;
           return count;
```



```
     }
```

```
static int proc_ lookupfoo(struct inode * dir, const char * name, int len,
        struct inode ** result)
 {
        unsigned int pid, ino;
        int i;

        *result = NULL;
        if (!dir)
              return -ENOENT;
        if (!S_ISDIR(dir->i_mode)) {
              iput(dir);
              return -ENOENT;
            }
        ino = dir->i_ino;
        i = NR_FOO_DIRENTRY;
        while (i-- > 0 && !proc_match(len,name,foo_dir+i))
             /* nothing */;
        if (i < 0) {
                iput(dir);
                return -ENOENT;
              }
        if (!(*result = iget(dir->i_sb,ino))) {
                iput(dir);
                return -ENOENT;
              }
        iput(dir);
        return 0;
      }

static int proc_readfoo(struct inode * inode, struct file * flie,
   struct dirent * dirent, int count)

   {
        struct proc_dir_entry * de;
        unsigned int pid, ino;
        Int i,j;

                - 93 -


        if (!inode || !S_ISDIR(inode->i_mode))
              return -EBADF;
        ino = inode->i_ino;
        if (((unsigned) filp->f_pos) < NR_FOO_DIRENTRY) {
                de = foo_dir + filp->f_pos;
                filp->f_pos++;
                i = de->namelen;
                ino = de->low_ino;
                put_fs _long(ino, &dirent->d_ino);
                put_fs_word(i, &dirent->d_reclen);
                put_fs_byte(0, i+dirent->d_name);
                j = i;
                while (i--)
                put_fs_byte(de->name[i], i+dirent->d_name);
                return j;
              }
        return 0;
   }


unsigned int get_foo(char * buffer)

   {
     /* code to find everything goes here */
```

```
        return sprintf(buffer, "format string ", variables);
    }


unsigned int get_suds(char * buffer)
    {
      /* code to find everything goes here */

        return sprintf(buffer, "format string", variables);
    }

unsigned int get_xyzzy(char * buffer)
```



```
    {
      /* code to find everything goes here */

        return sprintf(buffer, "format string", valriables);
    }

unsigned int get_baz(char * buffer)
    {
      /* code to find everything goes here */

        return sprintf(buffer, "format string", variables);
    }

unsigned int get_rootfile(char * buffer)
    {
      /* code to find everything goes here */

        return sprintf(buffer, "format string", variables);
    }
```

Прмечание: Текст функций proc_lookupfoo() и proc_readfoo() абстактный, так как они могут использоваться в разных местах.

4.Заполнеие каталогов dir1 и dir2 остается в качестве упражнения. В большинстве случаев эти каталоги не используются, однако алгоритм представленный здесь может быть перестроен в рекрсивный алгоритм заполнения более глубоких каталогов. Заметим, что в программе сохранены номера inode с 200 по 256 для каталога /proc/foo/ и всех его подкаталогов, так что вы можете использовать незанятые номероа inode в этом диапазоне для ваших собственных файлов в dir1 и dir2. Программа резервирует диапазон под каждый каталог для ваших будующих расширений. Автор также предпочел собрать всю информацию и требуемые функции в foo.c нежели создавать другой файл, если файлы не в dir1 и в dir2 не сильно концептуально отличаются от foo.

5.Сделайте соответствующие изменения в fs/proc/имя_файла. Это также будет достойным упражнением для читателя.

Примечание: вышенаписанная программа (/proc/net/supprt)была написана по памяти и может оказаться неполной. Если вы обнаружите в ней какие-то несоответствия пожалуйста пришлите аннотацию по адресу johnsonm@sunsite.unc.edu.

Глава 4.

Планировщик Linux.

Планировщик Linux представлен функцией schedule(), определяемой время переключения задач, и задачу представляемую к активизации. Планировщик работает совместно с функцией do_timer() вызюваемой 100 раз за одну секунду (в Linux/x86) на каждое прерывание таймера, с частью драйвера системного вызова ret_from_sys_call(), вызываемой при возврате системных вызовов.

Когда завершают работу симтемный вызов или "медленное" прерывание, вызывается ret_from_sys_call(). Эта функция делает небольшую работу, и нас в ней интересуют две строки:

```
cmpl $0_need_reshed
jne reshedule
```

Эта часть проверяет флаг need_reshed, и в случае если он установлен, вызывается функция schedule(), которая выбирает следующий процесс. После выбора процесса, ret_from_sys_call() выбирает условия работы процесса (которые часто зависят от процессов уже активизированных) и возвращается в пространство пользователя. Возврат в пользовательскую область вызывает новый прцесс, выбранный для запуска.

В sched_init() в kernel/sched.c, request_irq() используется для получения прерывания таймера. request_irq() устанавливается в положение ожидания до и после обслуживания прерываний, как видно в . Прерывания требуют быстрого обслуживания и случаются достаточно часто, так что распространенные прерывания по

возможности не используют ret_from_sys_call() после их выполнения, для уменьшение непроизводительных затрат. В частности они лишь сохраняют регистры, затертые C, и проверяют не собирается-ли обработчик использовать новые регистры. Эти обработчики "быстрых" прерываний могут быть установлены с помощью функции irqaction(), описанной в главе 2.6. Планировщик Linux сильно отличается от других планировщиков систем типа UN*X. Особенно это различие видно в "преданности" к приоритетам "nice-level". Вместо планирования запуска прцессов с высоким приоритетом в первую очередь, Linux использует круговое планирование, однако позволяет процессам с высоким приоритетом запускаться скорее и на более долгие сроки.

Стандартный планировщик UN*X использует очереди процессов. Обычно используются две приоритетные очереди: стандартная очередь и очередь "реального времени". Обычно процессы в очереди "реального времени" запускаются раньше процессов в стандартной очереди, в случае если они не заблокированы. Внутри каждой очереди высокоприоритетные процессы "nice-level" активизируются раньше менее приоритетных. Планировщик Linux более эффективен с точки зрения производительности.

4.1 Исходный текст.

Здесь представлена закомментированная и сокращенная копия исходника из /usr/src/linux/kernel/sched.c:

```
void schedule(void)
{
    int i, next, c;
    struct task_struct **p;
```

```
/* проверка на условия прбуждения, активизирует задачу, */
/* управляемую прерыванием, получившую сигнал          */

  need_reshed = 0;
  for(p=&LAST_TASK; p>&FIRST_TASK; --p) {
```


Таблица процессов находится в массиве указателей на структуры struct task_struct. См. определение этой структуры в /usr/include/linux/sched.h.

```
        if (!*p || ((*p)->state != TASK_INTERRUPTIBLE))
              continue;
        if ((*p)->timeout && (*p)->timeout < jiffies) {
```

Если прцесс имеет блокировку по времени и достигает ее, jiffies (число сотых секунды со времени старта системы) принимает значение timeout. timeout обычно установлена как jiffies+desired_timeout.

```
           (*p)->timeout = 0;
           (*p)->state = TASK_RUNNING;
        }else if ((*p)->signal & ~(*p)->blocked)
```

Если процессу подается сигнал отключения блокировки, процессу снова разрешается активизироваться, когда придет его очередь.

```
           (*p)->state = TASK_RUNNING;
        }
```

В этот момент все процессы готовы к работе и их флаги установлены на разрешение запуска. Прграмма готова выбрать один из них для запуска, просматривая таблицу процессов. В данный момент осуществляется поиск процесса с самой большой численной величиной на счетчике(counter). Счетчик прцесса прибавляется каждый раз во время вызова планировщика с помощью приоритета численно равного значению "nice" в ядре.

```
/* соответствующий планировщик */
  while (1) {
      c = -1;
      next = 0;
      i = NR_TASKS;
      p = &task[NR_TASKS]
      while (--i) {
```



```
        if (!*--p)
```

Если процесс в этом слоте отсутствует, не беспокойтесь...

```
           continue;
        if((*p)->state == TASK_RUNNING && (*p)>counter > c)
           c = (*p)->counter, next = i;
```

Если счетчик (counter) больше чем предыдущий просмотренный счетчик, осуществляется переход к следующему процессу, конечно, в случае если в дальнейшем в цикле не обнаружится еще более большое значение.

```
        }
        if (c)
```

```
                break;
            for(p = &LAST_TASK; p > &FIRST_TASK; --p)
                if (*p)
                    (*p)->counter = ((*p)->counter >> 1) +
                            (*p)->priority;
```

Здесь представлена установка счетчика. Сначала он делится на два затем устанавливается приоритет. Заметим, что из-за строки break это происходит лишь в случае отсутствия прцесса на который можно переключиться.

```
        }
        sti();
        switch_to(next);
    }
```

sti() снова запрещает прерывания, а switch_to() обуславливает переход к новому процессу сразу после выполнения ret_to_sys_call().

Я урезал функцию do_timer(), демонстрируя лишь куски относящиеся к schedule(). Для просмотра остальной части смотрите соответствующий раздел. В частности для ознакомления с механизмом

itimer смотрите раздел itimers. [Я думаю мне стоит написать этот раздел... Иногда мне придется ссылаться на него здесь]

Я специально выкинул весь учет наполнения, учет времени, и гибкий таймер.

```
    static void do_timer(struct pt_regs *regs)
    {
        unsigned long mask;
        struct timer_struct *tp = timer_table+0;
        struct task_struct **task_p;

        jiffies++;
```

Здесь представлено осуществление увеличения числа тиков. Эта процедура представляет ценность для всего ядра, так как все подсчеты времени (за исключением циклов задержки) основываются на ней.

```
        if (current == task[0] || (--current->counter)<=0) {
            current->counter = 0;
            need_reshed = 1;
        }

    }
```

Не позволяйте запускаться задаче 0, так как эта задача не делает ничего. В случае ее работы машина неактивна. Не позволяйте этого, при вероятности происхождения какого-либо события - по возможности чаще запускайте schedule().

Глава 5.

Как работают системные вызовы.

Эта глава рассматривает первые механизмы поддерживаемые 386 процессором и то как Linux использует эти механизмы. Здесь нет ссылок на конкретные системные вызовы - их слишком много, среди них постоянно появляются новые, и они документированы на страницах описания Linux.

5.1 Что поддерживет 386 процессор?

386 процессор разделяет события на два класса: прерывания и исключения. Оба типа событий предназначены для ускорения процесса переключения между задачами. Прерывания могу случаться в любое время работы программы, так как являются ревкцией на сигналы аппаратного обеспечения. Исключения вызываются определенными програмными инструкциями.

386 процессор распознает два типа прерываний: маскируемые и немаскируемые. Также определяются два типа исключений: определяемые процессором и програмные исключения.

Каждое исключение и прерывание имеет свой номер, который в литературе называется вектором. Немаскируемым прерываниям и исключениям определяемым процессором приписываются вектора с 0-го по 32, включительно. Вектора маскируемых прерываний определяются аппаратным обеспечением. Внешние прерывания помещают вектор в шину во время цикла определения прерывания. Любой вектор входящий в диапазон от 32 до 255 может быть использован маскируемым прерыванием или програмируемым исключением. См рис 4.1 для просмотра всех возможных прерываний и исключений:

0 ошибка деления
1 исключение отладки
2 немаскируемое прерывание
3 контрольная точка (breakpoint)
4 переполнение при вводе
5 достижение границы
6 неверный код операции
7 недоступен сопроцессор
8 двойная ошибка

9 перезапущен сегмент сопроцессора
10 неправильный сегмент метки задачи
11 отсутствие сегмента
12 неисправность стека
13 общая защита
14 неисправность страницы
15 не используется
16 ошибка сопроцессора
17-31 не используются
32-255 маскируемые прерывания

Рис 4.1: Значения прерываний и исключений.

ВЫСШИЙ | Неисправности включающие в себя неисправность отладчика
| Неисправность инструкций INTO, INT n, INT 3.
| Отладка неисправностей этих инструкций.
| Отладка последующих инструкций.
| Немаскируемое прерывание
НИЗШИЙ | Прерывание INTR

Рис 4.2: Приоритет прерываний и исключений.

5.2 Как Linux использует прерывания и исключения.

В Linux, запуск системного запроса вызывается маскируемым прерыванием, или-же передачей класса исключения, обусловленной инструкцией int 0x80. Мы используем вектор 0x80 для передачи контроля ядру. Этот вектор устанавливается во врем инициализации, среди других важнейших векторов таких, как вектор таймера.

В версии Linux 0.99.2 присутствует 116 системных вызовов. Документацию по ним можно найти непосредственно в самой документации по Linux. Во время обращения пользователя к системному вызову, происходит следующее:

- Каждый вызов определяется в libc. Каждый вызов внутри

библиотеки libc в общем-то представляет собой макрос syscallX, где X - число параметров текущей подпрограммы. Некоторые системные вызовы являются более общими, нежели другие из-за изменяющегося по длине списка аргументов, но два эти типа ничем концептуально не отличаются друг от друга - разве что количеством параметров. Примерами общих системных вызовов могут служить вызовы open() и ioctl().

- Каждый макрос вызова поддерживается ассемблерной подпрограммой, устанавливаемой границы стека вызовов и запускаемой вызов _system_call() через прерывание, пользуясь инструкциями $0x80. К примеру вызов setuid представлен как:

```
_syscall1(int,setuid,uid_t,uid);
Что расширяется в :
_setuid
 subl $4,%exp
 pushl %ebx
 movzwl 12(%esp),%eax
 movl %eax,4(%esp)
 movl $23,%eax
 movl 4(%esp),%ebx
 int $0x80
 movl %eax,%edx
 test1 %edx,%edx
 jge L2
 negl %edx
 movl %edx,_errno
 movl $-1,%eax
 popl %ebx
 addl $4,%esp
 ret
L2:
 movl %edx,eax
 popl %ebx
 addl $4,esp
 ret
```


Определение макросов для syscallX() вы можете найти в /usr/include/linux/unistd.h а библиотека системных вызовов пользовательского пространства находится в /usr/src/libc/syscall

- С этой точки зрения системный код вызова не запущен. Он не

запускается до запуска int $0x80 осуществляющего переход на ядровую _system_call(). Эта процедура общая для всех системных вызовов. Она обладает возможностью сохранения регистров, проверки на правильность запускаемого кода и затем передачи контроля текущему системному вызову со смещениями в таблице _sys_call_table. Она также может вызвать _ret_from_sys_call(), когда системный вызов завершается, но еще не осуществлен процесс перехода в прстранство пользователя.

Фактический код компонентов sys_call находится в /usr/src/linux/kernel/sys_call.s. Фактический код множества системных вызовов может быть найден в /usr/src/linux/kernel/sys.c. Остальную часть ищите сами.

- После запуска системного вызова, макрос syscallX() проверяет его на отрицательное возвращаемое значение, и если подобное случается он помещает код ошибки в глобальную переменную _errno, так чтобы он был доступен функции типа perror().

5.3 Как Linux устанавливает вектора системных вызовов.

Код startup_32 находящийся в /usr/src/linux/boot/head.S начинает всю работу запуская setup_idt(). Подпрограмма устанавливает IDT (таблицу описания прерываний) с 256 записями. Никаких отправных точек прерываний этой программой не загружается, и делается это лишь после разрешения пейджинга и перехода ядра по адресу 0xC0000000. В IDT находится 256 записей по 4 байта каждая, всего 1024 байта.

Когда вызывается start_kernel() (/usr/src/linux/init/main.c) она запускает trap_init() (описае в /usr/src/linux/kernel/traps.c). trap_init() устанавливает таблицу дескрипторов прерываний как показано на рис 4.3

0 - divide_error (ошибка деления)
1 - debug (отладка)
2 - nmi (немаскируемое прерывание)
3 - int3
4 - overflow (переполнение)
5 - bounds (достижение границ)
6 - invalid_op (неверный процесс)
7 - device_not_avaible (обращение к устройству невозможно)
8 - double_fault (двойная ошибка)
9 - coprocessor_segment_overrun (перезапуск сегмента сопроцкссора)
10 - invalid TTS (неверная TTS)
11 - segment_not_present (отсутствие сегмента)
12 - stack_segment (стековый сегмент)
13 - general_protection (общая защита)
14 - page_fault (ошибка чтения страницы)
15 - не используется
16 - coprocessor_error(ошибка сопроцессора)
17 - alignment_check(проверка расстановки)
18-48- не используются

На этот момент вектор прерывания системных вызовов не установлен. Он инициализируется sched_init() (находится в /usr/src/linux/kernel/sched.c). Вызов set_system_gate (0x80,&system_call) устанавливает прерывание 0x80 как вектор параметра system_call().

5.4 Как установить свой собственный системный вызов.

1. Создайте каталог в /usr/src/linux/ для вашего кода.

2. Поместите нужные вам библиотеки в /usr/include/sys/ и /usr/include/linux/

3. Поместите ваш отлинкованный модуль в ARCHIVES и подкаталог в строки SUBDIRS высокого уровня создания файла. См fs/Makefile - fs.o.

4. Поместите #define __NR_xx в unistd.h для присвоения номера вашему системному запросу, где xx - индекс описания вашего вызова. Она будет использована для установки вектора с помощью sys_call_table вызываемого ваш код.

5. Введите отправную точку для вашего системного запроса в sys_call_table в sys.h. Она будет зависеть от индекса xx в предыдущем пункте. Переменная NR_syscalls будет пересчитана автоматически.

6. Измените какой-нибудь код ядра в /fs/mm/ для установки инсрументов нужных вашему вызову.

7. Запустите процесс компановки на высшем уровне для создания вашего кода в ядре

После этого вам останется лишь занести системный вызов в ваши библиотеки, или использовать макрос _syscalln() в программе использующей ваши разработки, для разрешения им доступа к новому системному вызову.

В библиографии содержаться несколько полезных ссылок на книги охватывающие эту тему. В частности полезно будет просмотреть "The 386DX Microprocessor Programmer's Reference Manual" и "Advanced 80386 Programming Techniques" Джеймса Турли.

Глава 6

Управление памятью в Linux

6.1 Введение

Система управления памятью в Linux осуществляет подкачку страниц по обращению в соответствии с COPY-ON-WRITE стратегией, основанной на механизме подкачки, который поддерживается 386-м

процессором. Процесс получает свои таблицы страниц от родител я (при выполнении fork()) со входами, помеченными как READ-ONLY или замещаемые. Затем, если процесс пытается писать в эту область памяти, и страница является COPY-ON-WRITE страницей, она копируется и помечается как READ-WRITE. Инструкция exec() приводит к считыванию страницы или то же самое происходит при выполнении программы. В дальнейшем процессу затруднительно получить доступ к другой странице. Каждый процесс имеет директорию страниц, что подразумевает возможность доступа к 1KB таблиц страниц, указывающих на 1MB 4-х килобайтных страниц, которые размещаются в 4GB памяти. Директория страниц процесса инициализируется при выполнении распараллеливания посредством copy_page_tables().

Директория страниц холостого процесса инициализируется путем выполнения инициализирующей последовательности.

Каждый пользовательский процесс имеет локальную таблицу дескриптора, которая содержит сегмент кода и сегмент данные-стек. Сегментам пользователя отводится память от 0 до 3GB(0xc0000000). В пользовательском пространстве линейные адреса (см. сноску1) и логические адреса (см.сноску2) идентичны.

--- сноска1. В процессоре 80386 значение линейного адреса лежит в пределах 0GB - 4GB. Линейный адрес указывает на область памяти в этом пространстве. Линейный адрес отличен от физического адреса, он является виртуальным.

--- сноска 2. Логический адрес состоит из селектора и смещения. Селектор указывает на сегмент, а смещение говорит о том, где размещена область внутри сегмента.

Код ядра и сегменты данных являются привилегированными сегментами, определяются в таблице глобального дескриптора и размещаются в области от 3GB до 4GB. Установлена директория страниц программы подкачки (swapper_pg_dir) и таким образом логические и физические адреса в пространстве ядра являются идентичными.

Пространство выше 3GB появляется в директории страниц

процесса как указатели на таблицы страниц ядра. Эта область прозрачна для процессов в режиме пользователя, однако, планирование распределения памяти становится уместным при привилегированном режиме, например, при выполнении системного вызова.

Режим супервизора вводится внутри контекста текущего процесса, так что трансляция адреса происходит относительно каталога страниц процесса, но используя сегменты ядра. Это идентично тому, как происходит управление памятью с использованием swapper _pg_dir и сегменты ядра как и директории страниц используют те же таблицы страниц в этом пространстве.

Только task[0] (холостая задача (см.сноску 3) [Этот термин должен был быть определен ранее]) использует swapper_pg_dir напрямую.

---сноска 3. В силу исторических причин иногда называется задача подкачки, хотя при работе Linux с подкачкой она не связана.

* Для пользовательского процесса segment_base = 0x00, page_dir для процесса своя.

* процесс пользователя выполняет системный вызов: segment_base=0xc0000000 page_dir=та же самая page _dir пользователя

* swapper_pg_dir содержит распределение памяти для всех физических страниц от 0xc0000000 до 0xc0000000 + end_mem, так что первые 768 входов в swapper_pg_dir раны 0 и затем имеются 4 или более входов, которые указывают на таблицы страниц ядра

* Директории страниц пользователя имеют те же входы, как tt swapper_pg_dir свыше 768. Первые 768 входов представляют пространство пользователя.

В результате всегда, когда линейный адрес превышает

0xc0000000, используются те же таблицы страниц ядра.

Пользовательский стек размещается на вершине сегмента данных пользователя и увеличивается вниз. Стек ядра не является точным представлением структуры данных или сегмента, относительно которой я бы мог сказать "Вы находитесь в стеке ядра". Kernel_stack_frame (страница) связывается с каждым вновь создаваемым процессом и используется всякий раз, когда ядро выполняет действия с контекстом процесса. Неприятности могут произойти, если стек ядра будет расти ниже его текущего кадра.[Где размещен стек ядра? Я знаю, что для каждого процесса существует свой стек, но где он расположен, когда процесс не используется?]

Страницы пользователя могут заниматься или замещаться. Страница пользователя отражается ниже 3GB в таблице страниц пользователя. Эта область не содержит директорий страниц или таблиц страниц. Меняются местами (замещаются) только грязные страницы.

Необходимы незначительные изменения в некоторых местах (например, контроль ограничений на память процесса), чтобы обеспечить возможность программисту определять свои сегменты.

6.2 Физическая память

Ниже представлена карта физической памяти перед тем, как будет выполнен любой процесс. Левый столбец представляет начальный адрес инструкции, отмеченные значения являются приблизительными. Средний столбец включает в себя название инструкции. Крайний правый столбец представляет имя соответствующей процедуры или переменной или комментарий входа.

0x110000 свободна memory_end или high_memory

mem_map mem_init()

inode_table inode_init()

ннформ. устройства device_init()+

0x100000 добав. pg_tables paging_init()

0x0A0000 не используется

0x060000 свободна

low_memory_start

0x006000 код ядра + данные

floppy_track_buffer

bad_pg_table занято page_fault_handlers для
bad_page уничтожения процесса, если он
находится вне памяти.

0x002000 pg0 первая таблица страниц в ядре

0x001000 swapper_pg_dir каталог страниц ядра

0x000000 нулевая страница

+ устройство, захватывающее память (main.c): profil_buffer, con_init, psaux_init, rd_init, scsi_dev_init. Заметьте, что не вся память помечена как FREE или RESERVRVED (mem_init).

Страницы, помеченные как RESERVED принадлежат ядру и никогда не освобождаются или переставляются.

6.3 Память пользовательского процесса

0xc0000000 невидимое ядро не используется
начальный стек

место для расширения стека 4 страницы
0x60000000 стабильно записанные биьлиотеки
brk не используется
распределенная память
end_data не инициализированные данные
end_code инициализированные данные
0x00000000 текст

Как сегмент кода, так и сегмент данных в каждом случае размещаются в области от 0x00 - 3GB. Обычно программа контроля равильности использования страниц do_wp_page проверяет, чтобы процесс не производил запись в область кода. Однако, если перехватить SEGV сигнал, то появляется возможность писать в область кода, инициируя возникновение COPY-ON-WRITE. Программа управления do_no_page гарантирует, что ни одна новая страница, выделенная процессу, не будет принадлежать ни исполняемой области, ни разделяемой библиотеке, ни стеку, ни попадет внутрь brk значения. Пользовательский процесс может сбросить свое brk значение посредством вызова sbrk(). Это то, что делает malloc(), когда это необходимо. Части текста и данных размещаются в отдельных страницах, если не установлена опция -N компилятора. Адреса загрузки разделяемой библиотеки обычно сами берутся из разделяемого пространства. Такой адрес лежит между 1.5GB и 3GB за исключением особых случаев.

	своппируемая	стабильная
страницы кода	Y	Y
страницы информации	Y	N?
стек	Y	N
pg_dir	N	N
кодовая/информационная page_table	N	N
стек page_table	N	N
task_struct	N	N
kernel_stack_frame	N	N
shlib page_table	N	N

несколько shlib страниц	Y	Y?

[Что означают отметки в виде вопроса? Должны ли они означать вашу неуверенность или альтернативность решения?]

Стек, разделяемые библиотеки и данные слишком удалены друг от друга, чтобы перекрываться одной таблицей страниц. Все page_tables ядра разделяются всеми процессами и поэтому не приведены в списке. Перемещаются только грязные страницы. Чистые страницы могут заниматься, таким образом процесс может читать их снова из исполняемой области в случае такой необходимости. Обычно разделяются только чистые страницы. Грязная страница перестает разделяться в случае распараллеливания пока родитель или потомок не станет снова производить в нее запись.

6.4 Данные управления памятью в таблице процессов

Ниже приводится краткое описание некоторых данных, содержащихся в таблице процессов, которые используются для управления памятью: [Это должно быть документировано значительно лучше. Необходима значительно большая детализация]

* Ограничения на память процесса: ulong start_code, end_code, end_data, brk, start_stack;

* Определение нарушения страницы: ulong min_flt, maj_flt, cmin_flt, cmaj_flt;

* Локальная таблица дескриптора: struct desc_struct ltd[32] представляет собой локальную таблицу дескриптора задачи.

* rss количество резидентных страниц.

* swappable: если - 0, тогда страницы процесса не замещаются.

* kernel_stack_page: указатель на страницу, размещенную при

распараллеливании.

* saved_kernel_stack: V86 режим работы.

* struct tss

- Сегмент стека
  esp0 указатель на стек ядра (kernel_stack_page)
  ss0 сегмент стека ядра (0x10)
  esp1 = ss1 = esp2 = ss2 = 0
  неиспользуемые привилегированные уровни.
- Секторы сегмента: ds = es = fs = gs = ss = 0x17, cs = 0x0f
  все указатели на сегменты в текущем ltd[].
- cr3: указывает на директорию страниц для данного процесса.
- ltd: _LTD(n) селектор для LTD текущей задачи.

6.5 Инициализация памяти

В start_kernel (main.c) имеются 3 переменные, связанные с инициализацией памяти:

- memory_start начинается от 1MB. Изменяется посредством инициализации устройства.

- memory_end конец физической памяти: 8MB, 16MB и т.д.

- low_memory_start конец кода и данных ядра, загружаемых первоначально.

Каждое устройство при инициализации по-своему берет memory_stsrt и возвращает измененное значение, если оно выделяет пространство начиная с memory_stsrt (просто захватывая его). paging_init() инициализирует таблицы страниц в swapper_pg_dir (начинающиеся с 0xc0000000), чтобы накрыть всю физическую память начиная с memory_start и кончая memory_end. В действительности первые 4MB обрабатываются в startup_32 (head.s). memory_start увеличивается, если добавляется какая-либо новая page_tables. При

пребывании по обращению по пустому указателю в ядре первая страница обнуляется.

В shed_init() ltd и tss дескрипторы задачи task[0] устанавливаются в GDT, и загружаются в TR и DTR (единственный случай, когда это делается явно). TRAP GATE (0x80) устанавливается для system_call(). Флаг вложенной задачи сбрасывается при подготовке к переходу в пользовательский режим. Таймер включается. task_struct для task[0] в полном объеме появляется в
.

Далее с помощью mem_init() создается mem_map, чтобы отражать текущее использование физических страниц. Это состояние, которое отражается в карте физической памяти, описанной в предыдущем разделе.

Linux переходит в пользовательский режим посредством iret после сохранения в стеке ss, esp и т.п. Естественно, что сегменты пользователя для task[0] управляются прямо через сегменты ядра, т.о. выполнение продолжается точно с того места, где оно было прервано.

task[0]:

* pg_dir = swapper_pg_dir, что означает, что управление адресами происходит только в области от 3GB до 3GB + high_memory/

* LTD[1] = код пользователя, base=0xc0000000, size=640K

* LTD[2] = данные пользователя, base=0xc0000000, size=640K

Первый вызов exec() устанавливает входы LTD для task[1] в пользовательские значения с base=0x0, limit= TASK_SIZE = 0xc0000000. Согласно этому ни один процесс не видит сегменты ядра пока находится в пользовательском режиме.

6.5.1. Процессы и программа управления памятью

Работа, связанная с памятью, производимая посредством fork():

* Выделение памяти
  - 1 страница для task_struct.
  - 1 страница для стека ядра.
  - 1 для pg_dir и несколько для pg_tables (co[py_page_tables)

* Другие изменения
- ss0 установить на адрес сегмента стека ядра (0x10) чтобы быть уверенным?
- esp0 установит на вершину вновь созданного kernel_stack_page
- cr3 устанавливается посредством copy_page_tables() для указания на вновь размещенную директорию страниц.
- ltd=_LTD(task_nr) создает новый дескриптор ltd.
- дескриптор устанавливается в gdt для нового tss и ltd[].
- Остальные регистры наследуются от родителя.

Процессы прекращают разделение их сегментов данных и кода (хотя они имеют раздельные локальные таблицы дескрипторов, входы указывают на те же сегменты). Страницы стека и данных будут скопированы, когда родитель или наследник будет писать в них (COPY-ON-WRITE)

Работа, связанная с памятью, производимая посредством exec():

* Выделение памяти
- 1 страница для exec-заголовка полного файла для omagic
- 1 страница или больше для стека (MAX_ARG_PAGES)

* clear_page_tables() используется для удаления старых страниц.

* change_ltd() устанавливает дескрипторы в новом LTD[]

* ltd[1] = code base = 0x00, limit = TASK_SIZE

* ltd[2] = data base = 0x00, limit = TASK_SIZE

Эти сегменты представляются DPL=3,P=1,S=1,G=1 type=a(code) или 2(data)

* До MAX_ARG_PAGES грязные страницы argv и envp размещаются и сохраняются на вершине сегмента данных для вновь созданного пользовательского стека.
* Установить указатель инструкции вызывающей программы eip = ex.a_entry.

* Установить указатель стека вызывающей программы на созданный стек (esp = stack pointer) Данные будут выбраны из стека при возобновлении вызывающей программы.

* Редактирование ограничений на память
end_coe = ex.a_text
end_data = end_code + ex.a_data
brk = end_data + ex.a_bss

Программные и аппаратные прерывания управляются внутри контекста текущей задачи. Более детально, директория страниц текущего процесса используется при трансляции адреса. Сегменты, однако, являются сегментами ядра и таким образом все линейные адреса указывают в область памяти ядра. Предположим, например, что пользовательский процесс выполняет системный вызов и ядро хочет получить доступ к переменной по адресу 0x01. Линейный адрес будет равен 0xc0000001 (использование сегментов ядра) и физический адрес - 0x01. Буква здесь присутствует вследствие того, что директория страниц процесса отображает этот диапазон точно как page_pg_dir.

Область ядра (0xc0000000 + high_memory) отображается через таблицы страниц ядра, которые являются частью RESURVED памяти. Поэтому они разделяются всеми процессами. Во время

распараллеливания copy_page_tables() напрямую обращается к таблицам RESERVED страниц. Она устанавливает указатели в директориях страниц процесса, чтобы указывать на таблицы страниц ядра и не размещает в памяти новых таблиц страниц, как это обычно делается. В качестве примера kernel_stack_page (который размещен где-то в области ядра) не нуждается в связанной page_table,

размещенной в p_dir процесса, чтобы отобразить ее.

Инструкция прерывания устанавливает указатель стека и сегмент стека из привилегированных 0 значений, сохраненных в tss текущей задачи. Заметьте, что стек ядра в действительности является фрагментированным объектом - это не единственный объект, а группа стековых фреймов, каждый из которых размещается в памяти при создании процесса и освобождает память при окончании его. Стек ядра никогда не должен расти внутри контекста процесса слишком быстро, чтобы не расширится за пределы текущего фрейма.

6.6. Выделение и освобождение памяти: политика страничной организации

Когда любой процедуре ядра требуется память, она прекращает работу, вызывая get_free_page(). Это представляет собой более низкий уровень, чем kmalloc() (в действительности kmalloc() использует get_free_page(), когда возникает необходимость в расширении памяти).

get_free_mem() использует один параметр, приоритет. Допустимыми значениями являются GFP_BUFFER, GFP_KERNEL и GFP_ATOMIC. Она берет страницу из free_page_list, редактирует mem_map, обнуляет страницу и возвращает физический адрес страницы (заметте, что kmalloc() возвращает физический адрес. Логика mm зависит от идентичности отображения между логическими и физическими адресами).

Само по себе это достаточно просто. В действительности проблема состоит в том, что free_pagr_list может быть пуст. Если вы не испытывали потребности в использовании примитивных операций на этой стадии, вы попадаете в область вопросов изъятия стр аниц, которую мы будем тщательно рассматривать немного позже. Как крайнее средство (и это касается примитивных операций) страница изымалась из secondary_page_list (как вы могли догадаться, когда страницы освобождаются, в первую очередь происходит заполнение secondary_page_list).

В действительности манипуляции с page_lists и mem_map происходят в результате действий этого загадочного макроса, называемого REMOVE_FROM_MEM_QUEUE(), внутрь которого вы, возможно, никогда не захотите заглянуть. Достаточно сказать, что прерывания запрещены.

Теперь немного назад, к вопросу изъятия страниц. get_free_page() вызывает try_to_free_page, которая повторяет вызов shrink_buffers() и swap_out() в данной последовательности до тех пор, пока освобождение страниц не будет закончено успешно. Приоритет снижается при каждой успешной итерации, так что эти две процедуры чаще выполняют свои циклы по изъятию страниц.

Рассмотрим один проход swap_out():

* Пройти по таблице процесса и получить ответ Q от замещаемой задачи.

* Найти таблицу страницы пользователя (не RESERVED) в области, принадлежащей Q.

* Для каждой page в таблице выполнить try_to_swap_out(page).

* Выход, когда страница освобождена.

Отметим, что swap_out() (вызываемая из try_to_free_page()) поддерживает статические переменные, таким образом можно получить ответ на вопрос где она была освобождена при предыдущем вызове.

try_to_swap_out() сканирует таблицы страниц всех пользовательских процессов и проводит следующую политику изъятия:

1. Не относиться легкомысленно к RESERVED страницам.

2. Подвергать старению страницу, если она помечена как доступная (1 бит).

3. Не трогать страницы, выделенные ранее (last_free_pages[]).

4. Оставить грязные страницы с map_counts> 1.

5. Снизить значение map_count для чистых страниц.

6. Освободить чистые страницы, если для них не установлено соответствие.

7. Заместить грязные страницы с map_count = 1.

Результатом действий 6 и 7 будет остановка процесса при текущем освобождении физической страницы. Действие 5 вызывает потерю одним из процессов чистой неразделяемой страницы, к которой ранее не было доступа (снижение Q->rss), что вовсе неплохо, но действия накопления за несколько циклов могут значительно замедлить процесс. В данном случае происходит 6 итераций, таким образом страница, разделяемая шестью процессами, может быть изъята, если она чистая.

Входы таблицы страниц изменяются и TLB становится недействительной.[Последнее вызывает интерес. В этом, кажется, нет необходимости т.к. доступные страницы не удаляются и многие таблицы страниц могут быть просмотрены между итерациями.... возможно, в результате прерывания, если такое произошло и имела место попытка урезать наиболее раннюю страницу?]

Текущая работа по освобождению страницы выполняется free_page(), являющегося дополнением get_free_page(). Она игнорирует RESERVED страницы, редактирует mem_map, затем освобождает страницу и изменяет page_lists, если для него не установлено соответствие. Для подкачки (см. п.6 выше) write_swap_pege() принимает вызов, но не делает ничего значительного с точки зрения дальнейшего управления памятью.

Рассмотрение деталей shrink_buffers() увело бы нас далеко в сторону. В первую очередь она следит за свободными буферами, затем списывает грязные буфера, далее занимается занятыми буферами и вызывает free_page(), когда у нее появляется возможность

освободить все буфера на странице.

Заметим, что директории страниц и таблицы страниц наряду с RESERVED страниц не становятся изменяемыми, не изымаются и не стареют. Они отображаются в директории страниц процесса через зарезервированные таблицы страниц. Они освобождаются только пр и выходе из процесса.

6.7 Программы контроля корректности использования страниц

Когда процесс создается посредством распараллеливания, он стартует со своей директорией страниц и своей страницей. Таким образом программа контроля корректности использования страниц следит почти за всей памятью процесса.

Программа контроля do_page_fault() считывает некорректный адрес из регистра cr2. Код ошибки (считанный в sys_call.S) позволяет определить режим доступа - пользователя/супервизора и причину ошибки - защита записи или неправильная страница. Формирователь управляется do_wp_page() и позднее do_no_page().

Если нарушение адреса превышает TASK_SIZE, процесс получает SIGKILL. [Зачем этот контроль? Такое может произойти только в режиме ядра из-за защиты на уровне сегмента]

Эти процедуры обладают тонкостями т.к. они могут вызываться по прерыванию. Вы не можете предположить, что это текущая задача.

do_no_page() контролирует три возможные ситуации:

1. Страница замещается.
2. Страница принадлежит исполняемой или разделяемой библиотеке.
3. Страница некорректна - страница данных, которая не была загружена.

Во всех случаях в первую очередь вызывается

get_empty_pgtable() чтобы гарантировано определить существование таблицы страниц, которая накрывает некорректный адрес. В случае 3 get_empty_page() вызывается чтобы обеспечить страницу с требуемым адресом и в случае замещаемой страницы вызывается swap_in().

В случае 2 программа контроля вызывает share_page(), чтобы посмотреть является ли страница разделяемой каким либо другим процессом. В случае неудачи она считывает страницу из исполняемой программы или библиотеки (Она повторяет вызов shre_page() в случае, если другой процесс делал тем временем то же самое). Любая часть страницы сверх значения brk обнуляется.

Считывание страницы с диска вычисляется как основная ошибка (mjr_flt). Это происходит с swap_in() или когда происходит считывание из выполняемой программы или библиотеки. Другие случаи интерпретируются как второстепенные ошибки (min_flt).

Когда найдена разделяемая страница, то она защищена для записи. Процесс, который пишет в разделяемую страницу, затем должен будет пройти через do_wp_page(), которая выполняет COPY-ON-WRIGHT.

do_wp_page() выполняет следующее:

* посылает SIGSEGV, если какой-либо пользовательский процесс пишет в текущую code_space.

* Если старая страница не разделяется, она становится незащищенной. Иначе get_free_page() и copy_page(). Для страницы устанавливается грязный флаг из старой страницы. Уменьшается значение счетчика карты старой страницы.

6.8. Листание (paging)

Листание (paging) оперирует со страницей в отличии от подкачки (swapping), которая используется в отношении любых процессов. Мы будем использовать здесь термин "подкачка" для того,

чтобы ссылаться на листание, т.к. Linux только листает и не замещает, но более привычным является термин "замещать" (swap), чем "листать" (page). Страницы ядра никогда не замещаются. Очищенные страницы также не читаются для замещения. Они освобождаются и перезагружаются, когда требуется. Программа подкачки поддержи вает один бит информации старения, являющимся битом PAGE_ACCESSED во входах таблицы страниц.

Linux использует множество файлов подкачки или устройства, которые могут быть включены и выключены посредством системных вызовов swapon и swapoff. Каждый файл подкачки или устройство описано посредством struct swap_info_struct (swap.c)

```
static struct swap_info_struct {
      unsigned long flags;
      struct inode *swap_file;
      unsigned int swap_device;
      unsigned char *swap_map;
      char          *swap_lockmap;
      int           lowest_bit;
      int           highest bit;
} swap_info[MAX_SWAPFILES];
```

Поле флагов (SWP_USED или SWP_WRITEOK) используется для управления доступом к файлам подкачки. Когда флаг SWP_WRITEOK сброшен, для этого файла не будет выделяться пространство. Это используется системным вызовом swapoff, когда он делает невозможным использование файла. Когда вызов swapon добавляет новый файл подкачки, то устанавливается SWP_USED. Статическая переменная nr_swapfiles содержит количество активных файлов подкачки. Поля lowest_bit и highest_bit связывают свободные области в файле подкачки и используются, чтобы увеличить скорость поиска свободной области подкачки.

Программа пользователя mkswap инициализирует устройство подкачки или файл. Первая страница включает заголовок ("SWAP-SPACE") в последних 10 байтах и содержит область битмап. Первоначальные нулевые значения в битмап сигнализируют о плохих

страница х. "1" в битмап означает, что соответствующая страница свободна. Такой странице память никогда не выделяется, таким образом сразу необходимо провести инициализацию.

Системный вызов swapon() осуществляется из программы

пользователя обычно из /etc/rc. Пара страниц памяти выделяется для swap_map и swap_lockmap.

swap_map содержит один байт на каждую страницу файла подкачки. Инициализируется из битмап и содержит 0 для допустимых страниц и 128 для неиспользуемых страниц. Используется для поддержки счета запросов на замещение каждой страницы в файле подкачки. swap_lockmap содержит один бит на каждую страницу, который используется, чтобы гарантировать взаимное исключение при чтении или записи в файл подкачки.

Когда страница памяти готова к тому, чтобы быть замещенной, индекс области замещения получается путем вызова get_swap_page(). Этот индекс затем загружается в биты 1-31 входа таблицы страниц, таким образом замещаемая страница может быть размещена, когда это необходимо, программой контроля корректности использования страниц do_no_page().

Верхние 7 битов индекса дают файл подкачки (или устройство), а нижние 24 бита дают номер страницы на этом устройстве. Это позволяет создавать до 128 файлов, каждый из которых предоставляет область до 64GB, но пространство свыше этого требует, что бы swap_map было бы большим. Вместо этого размер файла подкачки ограничивается 16MB потому, что swap_map тогда занимает 1 страницу.

Функция swap_duplicate() используется в copy_page_tables(), чтобы разрешить дочернему процессу наследовать замещаемые страницы во время распараллеливания. Это сразу увеличивает значение счетчика для этой страницы, поддержанного в swap_map. Каждый процесс будет замещен в отдельной копии страницы при обращении к ней.

swap_free() уменьшает счетчик, поддерживаемый в swap_map. Когда счетчик сбрасывает в 0, страница может быть перезагружена с помощью get_swap_page(). Эта функция вызывается каждый раз, когда замещаемая страница читается в память (swap_in()) или когда страница готова к тому, чтобы быть сброшенной (free_one_table() и т.п.).

6.9 Управление памятью в 80386

Логический адрес, задаваемый в инструкции в первую очередь транслируется в линейный адрес посредством аппаратных средств сегментации. Этот линейный адрес затем транслируется в физический адрес модулем страничной организации (paging unit).

6.9.1 Страничная организация (paging) в 386

Существует два уровня косвенной адресации при трансляции адреса в модуле страничной организации. Директория страниц содержит указатели на 1024 таблицы страниц. Каждая таблица страниц содержит указатели на 1024 страницы. Регистр CR3 содержит физический базовый адрес директории страницы и загружается как часть TSS в task_struct и поэтому загружается при каждом переключении задачи. 32-х битный линейный адрес разделяется следующим образом:

Физический адрес затем вычисляется (аппаратно) таким образом:

```
        CR3+DIR          указатель на table_base
 table_base+TABLE         указатель на page_base
physical_address=        page_base + OFFSET
```

Директории страниц (таблицы страниц) это страница, выровненная так, что нижние 12 бит используются для загрузки

полезной информации о таблице страниц (страница), указатель на которую задается посредством входа. Формат входов директории страниц и таблицы страниц:

```
31           12 11 9 8 7 6 5 4 3  2   1  0
   ADDRESS       OS  0 0 D A 0 0 U/S R/W P
```

D - "1" означает, что страница грязная (неопределенно для входа директории страниц).

R/W - "0" означает для пользователя "только для чтения".

U/S - "1" означает страницу пользователя.

P - "1" означает, что страница находится в памяти.

A - "1" означает, что к странице был доступ (устанавливается в 0 при старении).

OS - биты могут использоваться для LRU и т.п. и определяются OS.

Соответствующие определения для Linux находятся в .

Когда страница замещается, используются биты 1-31 входа таблицы страниц, чтобы отметить, куда при замещении помещается страница (бит "0" должен иметь значение 0).

Страничная организация (paging) делается доступной путем установки старшего бита в CR0 [в head.S?]. В каждой фазе трансляции адреса проверяются разрешения доступа, страницы не присутствуют в памяти и нарушение защиты приводит к их отсутствию. Затем программа контроля корректности использования страниц (в memory.c) или вносит новую страницу, или снимает защиту страницы, или делает все необходимое, что должно быть сделано.

Информация о некорректной работе со страницей

* Регистр CR2 содержит линейный адрес, в котором было вызвано нарушение страницы.

* Коды нарушения страницы (16 бит):

бит	сброшен	установлен
0	страница не существует	защита уровня страницы
1	нарушение при чтении	нарушение при записи
2	режим супервизора	режим пользователя

Остальные биты не определены. Приведенная информация является выдержкой из sys_call.S

Translation Lookside Buffer (TLB) представляет собой аппаратную кэш-память для физических адресов, которые соответствуют ранее используемым виртуальным адресам. Когда транслируется виртуальный адрес, 386 в первую очередь просматривает TLB, чтобы узнать - является ли доступной необходимая информация. Если нет, то для того, чтобы получить страницу, он должен создать пару ссылок на память для доступа к директории страниц и затем таблице страниц. Три ссылки на физическую память для трансляции адрес а для каждой ссылки на логическую память убили бы систему и, следовательно, TLB.

TLB заполняется, если загружен CR3 или по переключению задач, в результате которого изменяетсяCR0. В Linux она заполняется путем вызова invalidate(), которая как раз и перезагружает CR3.

6.9.2 Сегменты в 80386

Сегментные регистры используются при трансляции адреса для генерации линейного адреса из логического (виртуального) адреса.

linear_address = segment_base + logical_address

Линейный адрес транслируется затем в физический адрес посредством аппаратуры страничной организации (paging)

Каждый сегмент в системе описан 8-ми байтным дескриптором сегмента, в котором содержится вся необходимая информация (база, ограничение, тип, привилегии).

Имеют место следующие сегменты:

Обычные сегменты

* сегменты кода и данных

Системные сегменты

*(TSS) сегменты состояния задачи

*(LDT) таблицы локальных дескрипторов

Характеристики системных сегментов:

* Системные сегменты являются спецификаторами задач

* Имеется TSS, связанный с каждой задачей в системе. Он содержит tss_struct (sched.h). Размер этого сегмента соответствует размеру tss_struct, исключая i387_union (232 байта). Он содержит всю информацию, необходимую для перезапуска задачи.

* Таблицы LDT содержат дескрипторы обычных сегментов, принадлежащих задаче. В Linux каждой задаче соответствует одна LDT. В Linux task_struct предусмотрено пространство для 32-х дескрипторов. Обычная LDT, созданная в Linux, имеет размер 24 байта и, следовательно, пространство для 3-х входов. Ее содержимое:

  - LDT[0] Null (принудительно)
  - LDT[1] дескриптора сегмента пользовательского кода

- LDT[2] дескриптор сегмента пользовательских данных/стека

* Все сегменты пользователя имеют базу 0x00, так что линейный адрес тот же самый, что и логический.

Для получения доступа ко всем этим сегментам 386 использует таблицу глобальных дескрипторов (GDT), которая устанавливается в памяти системой (местоположение задается регистром GDT). GDT содержит дескрипторы сегментов для каждого сегмента состоян ия задачи, каждого локального дескриптора и обычных сегментов. Linux GDT содержит входы двух обыкновенных сегментов:

- GDT[0] нулевой дескриптор
- GDT[1] дескриптор сегмента кода ядра
- GDT[2] дескриптор сегмента данных/стека ядра

Оставшаяся область GDT заполнена TSS и LDT дескрипторами системы.

- GDT[3] ???
- GDT[4] = TSS0, GDT[5] = LDT0
- GDT[6] = TSS1, GDT[7] = LDT1
  ..... и т.д.....

Заметьте LDT[n] != LDTn

- LDT[n] = n-й дескриптор в LDT текущей задачи
- LDTn = дескриптор в GDT для LDT n-й задачи

В данном случае GDT имеет 256 входов, пространство для 126 задач. Сегменты ядра имеют базу 0xc0000000, которая задает местонахождение ядра в линейном представлении. Прежде, чем сегмент может быть использован, содержимое дескриптора для этого сегмента должно быть загружено в сегментный регистр. 386 имеет множество сложных критериев, ограничивающих доступ к сегментам, так что вы не сможете просто загрузить дескриптор в сегментный регистр. Также эти сегментные регистры имеют невидимые для программиста участки. Видимые участки - это то, что обычно называется сегментными регистрами cs, ds, es, fs, gs и ss.

Программист загружает один из этих регистров 16-ти битным

значением, называемым селектором. Селектор однозначно идентифицирует дескриптор сегмента в одной из таблиц. Доступ подтверждается и соответствующий дескриптор загружается посредством аппар атных средств.

Обычно в Linux игнорируется комплексная защита на уровне сегмента (чрезмерная?), предоставляемая 386. Она базируется на основе аппаратных средств страничной организации и объединенной защитой на уровне страницы. Правила на уровне сегмента, которые применяются к пользовательским процессам, состоят в следующем:

1. Процесс не может напрямую обращаться к данным ядра или сегментам кода.

2. Всегда имеет место контроль ограничения, однако, условие, что каждый пользовательский сегмент размещается от 0x00 до 0xc0000000, неудобно для применения. [Это изменено и нуждается в редактировании]

6.9.3 Селекторы в 80386

Селектор сегмента загружается в сегментный регистр (cs, ds и т.д.), чтобы через сегментный регистр задать один из обычных сегментов в системе как один адрес.

Формат селектора сегмента:

```
15                              3   2  1   0
      индекс                        TI   RPL
```

TI - индикатор таблицы:

0 означает, что индексы селектора относятся к GDT
1 означает, что индексы селектора относятся к LDT

RPL - привилегированный уровень. Linux использует только два привилегированных уровня

0 означает ядро
1 означает пользователя

Примеры:

Сегмент кода ядра:

TI = 0, индекс = 1, RPL = 0 поэтому сектор = 0x08 (GDT[1])

Сегмент данных пользователя

TI = 1, индекс = 2, RPL = 3 поэтому сектор = 0x17 (LDT[2])

Селекторы, используемые в Linux:

TI	index	RPL	selector	segment	
0	1	0	0x08	код ядра	GDT[1]
0	2	0	0x10	данные/стек ядра	GDT[2]
0	3	0	???	???	GDT[3]
1	1	3	0x0F	код пользователя	LDT[1]
1	2	3	0x17	данные/стек пользователя	LDT[2]

Селекторы для сегментов системы не предназначены для прямой загрузки в сегментные регистры. Напротив, должны быть загружены TR или LDTR.

На входе системного вызова:

* ds и es устанавливаются на сегмент данных ядра (0x10)

* fs устанавливается на сегмент данных пользователя (0x17) и используется для доступа к данным, на которые указывают аргументы системного вызова

* Указатель на вершину и сегмент стека автоматически устанавливаются в ss0 по прерыванию и старые значения восстанавливаются при возврате из системного вызова.

6.9.4 Дескрипторы сегментов

Имеется дескриптор сегмента, используемый для описания каждого сегмента в системе. Имеются обычные и системные дескрипторы. Ниже представлен дескриптор во всей своей красоте. Такой странный формат создан специально для того, чтобы обеспечить совместимость с 286. Заметьте, что он занимает 8 байт

```
63-54  55  54  53  52  51-48  47   46  45  44-40    39-16      15-0
Base   G   D   R   U   Limit  P   DPL  S   TYPE   Segmet Base Segmet Limit
31-24                19-16                  23-0       15-0
```

Разъяснения:

R зарезервирован (0)
DPL 0 означает ядро, 3 означает пользователя
G 1 означает гарантировано 4К (В Linux установлен всегда)
D 1 означает 32-х битное значение операнда по умолчанию
U определяется программистом
P 1 означает присутствие в физической памяти
S 0 означает системный сегмент, 1 означает обычный сегмент кода или данных
TYPE Существует много возможностей. Прерывания различны для системных и обычных дескрипторов.

Системные дескрипторы в Linux

TSS: P=1,DPL=0,S=0, type=9, limit=231 пространство для 1 tss_struct

LDT: P=1,DPL=0,S=1, type=2, limit=23 пространство для 3 дескрипторов сегментов

База устанавливается при выполнении fork(). Для каждой задачи есть свой TSS и LDT.

Обычные дескрипторы ядра в Linux (head.S):

код: P=1,DPL=0,S=1,G=1,D=1, type=a, base=0xc0000000, limit=0x3ffff

данные: P=1,DPL=0,S=1,G=1,D=1, type=2, base=0xc0000000, limit=0x3ffff

Состав LDT для task[0] (sched.h):

код: P=1,DPL=3,S=1,G=1,D=1, type=a, base=0xc0000000, limit=0x9f

данные: P=1,DPL=3,S=1,G=1,D=1, type=a, base=0xc0000000, limit=0x9f

LDT, устанавливаемые по умолчанию для оставшихся задач (exec()):

код: P=1,DPL=3,S=1,G=1,D=1, type=a, base=0, limit=0xbffff

данные: P=1,DPL=3,S=1,G=1,D=1, type=2, base=0, limit=0xbffff

Размер сегментов ядра 0x40000 страниц (4КВ страниц с G=1=1GB) тип подразумевает, что для кодового сегмента разрешается read-exec, а для сегмента данных read-write.

Регистры, объединенные посредством сегментации.

Формат сегментного регистра: (для программиста видимым является только селектор)

```
16 -бит        32-бит                    32-бит
селектор   база физического адреса   ограничения сегмента атрибуты
```

Невидимую часть сегментного регистра более удобно рассматривать в формате, используемом во входах таблицы дескрипторов, которые устанавливает программист. Таблицы дескрипторов имеют связанные с ними регистры, которые используются при их размещении в памяти. GDTR (и IDTR) инициализируются при запуске и вместе с этим определяются таблицы. LDTR загружается при каждом переключении задачи.

Формат GDTR (иIDTR):

32-бит	16-бит
Линейный базовый адрес	ограничение таблицы

TR и LDTR загружаются из GDT и таким образом имеют формат других сегментных регистров. Регистр задач (TR) содержит дескриптор для TSS текущей выполняемой задачи. Выполнение перехода на селектор TSS вызывает сохранение состояния в старом TSS, в TR загружается новый дескриптор и регистры перезагружаются новым TSS. Эти действия реализуются программой-шедуллером при переключении задач пользователя. Отметим, что поле tss_struct.ltd содержит селектор для LDT этой задачи. Он используется для того, чтобы загрузить LDTR. (sched.h)

6.9.5 Макросы, используемые при установке дескрипторов

В sched.h и system.h определены несколько ассемблерных макросов для простого доступа и установки дескрипторов. Каждый вход TSS и LDT занимает 8 байт.

Манипулирование GDT дескрипторами системы:

* _TSS(n)
  _LDT(n) предоставляют индекс в GDT для n-ой задачи.

* _LDT(n) загружается в поле ldt структуры tss_struct при распараллеливании.

* _set_tssldt_desc(n, addr, limit, type)
  ulong*n указывает на вход GDT для установки (см. fork.c).

База сегмента (TSS или LDT) устанавливается в 0xc0000000 + addr. Вот специфичные экземпляры описанного выше, где ltypy ссылается на байт, содержащий P, DPL, S, и тип:

set_ldt_desc(n,addr) ltype=0x82

P=1, DPL=0, S=0, type=2 означает вход LDT. limit=23=? пространство для 3 дескрипторов сегмента

set_ldt_desc(n,addr) ltype=0x89

P=1, DPL=0, S=0, type=9 означает доступный 80386 TSS limit=231 пространство для 1 tss_struct

* load_TR(n),
  load_ldt(n) загружает дескрипторы для задачи с номером n в регистр задачи и ldt регистр.

* ulong get_base(struct desk_struct ldt) берет базу из дескриптора

* ulong get_limit (ulong segment) берет ограничение (размер) из селектора сегмента. Возвращает размер сегмента в байтах.

* set_base(struct desk_struct ldt, ulong base),
  set_base(struct desk_struct ldt, ulong limit)

  Установит базу и ограничения для дескрипторов (4К неделимых сегментов). Ограничением здесь является действительно существующий размер сегмента в байтах.

* _set_seg_desc(gate_addr, type, dpl, base, limit)

  Значения по умолчанию 0x00408000 = ? D=1,P=1,G=0 В данный момент рабочий размер 32 бита и максимальный размер 1M. gate_addr должен быть (ulong*)

Приложение А

Библиография

А.1. Аннотированная библиография.

Эта аннотированная библиография охватывает книги по теории операционных систем так же, как и по разным видам программирования

в среде UNIX. Указанная цена может быть точной, а может и нет, но будет вполне преемлема для правительственной работы. [Если у вас есть книга, которая, по-вашему, подходит для библиографии, напишите ее краткий обзор и пошлите необходимую информацию (заголовок,автор, издательство, ISBN и приблизительная цена) и обзор по адресу johnson@sunsite.unc.edu]. Эта версия постепенно отходит, в то время как появляется настоящая библиография.

Заглавие: The Design of the UNIX Operating System
Автор: Maurice J. Bach
Издательство: Prentice Hall, 1986
ISBN: 0-13-201799-7
Приближенная цена: $65.00

Это одна из книг, которые Linus использовал при разработке Linux. Это описание структур данных, используемых в ядре System V. Множество имен важных функций в исходных текстах Linux пришли из этой книги, и названы по алгоритмам, представленным в ней. Например, если вы не можете догадаться, что делают функции getblk(), brelse(), bread(), breada() и bwrite(), глава 3 об'яснит это очень хорошо.

В то время как большинство алгоритмов схожи или одинаковы, стоит отметить несколько различий:

* Буферный кеш Linux динамически перераспределяется под другой размер, так что алгоритм для действий по получению новых буферов немного другой. Поэтому об'яснение getblk(), приведенное выше, отличается от getblk() в Linux.
* Linux не использует потоков в текущий момент, и, если потоки реализуются для Linux, желательно, чтобы они имели другую семантику.
* Семантика и структура вызовов драйверов устройств другая. Концепция сходна, и еще стоит прочитать главу по драйверам устройств, но для подробностей по

драйверам лучшая ссылка - The Linux Kernel Hackers' Gude.
* Алгоритмы распределения памяти немного отличны.

Есть и другие маленькие отличия, но хорошее понимание этой книги поможет вам разобрать исходные тексты Linux.

Заглавие: Advanced Programming in the UNIX Environment
Автор: W. Richard Stevens
Издательство: Addison Wesley, 1992
ISBN: 0-201-56317-7
Приближенная цена: $50.00

Этот замечательный томик охватывает все тонкости, которые вы действительно должны знать, чтобы писать настоящие UN*X программы. Он включает обсуждение различных стандартов релизации UN*X, включая POSIX, X/Open XPG3и FIPS, и концентрируется на двух реализациях, SVR4 и предварительный выпуск 4.4 BSD, который упоминается в книге, как 4.3 + BSD.

Заглавие: Advanced 80386 Programming Techniques
Автор: James L. turley
Издательство: Osborne McGraw-Hill, 1988
ISBN: 0-07-881342-5
Прибл. Цена: $22.95

Эта книга достаточно хорошо охватывает 80386, не затрагивая других аппаратных средств. В книгу включены примеры кода. Охвачены все главные возможности, также как и множество основных понятий. Книга включает следующие главы: Основы, Сегментация памяти, Уровни привелегий, Замещение страниц, Многозадачность, Связь между задачами, Обработка сбоев и прерываний, Эмуляция 80286, Эмуляция 8086, Отладка, Математический процессор 80387, Сброс и реальный режим, Аппаратное обеспечение, и несколько приложений включая таблицы управления памятью в качестве справочника.

У автора хороший стиль изложения: если у вас технический склад ума, вы найдете захватывающим чтение этой книги. Сильная сторона этой книги в том что автор не обьясняет ни как делать что либо под DOS, ни как обращаться с конкретной аппаратурой. Фактически единственное место где он упоминает DOS и PC-совместимое аппаратное обеспечение это во введении где он обещает больше не упоминать о них.

Заглавие: The C programming Language, second edition.

Автор: Brian W. Kernighan and Dennis M. Ritchie
Издательство: Prentice Hall, 1988
ISBN: 0-13-110362-8
Прибл. Цена: $35.00

Библия по программированию на Си. Включает учебник по Си, справочник по UN*X интерфейсу, справочник по Си и справочник по стандартным библиотекам. Программируете на Си, купите эту книгу. Это просто.

Заглавие: Operating Systems: Design and Implementation
Автор: Andrew S. Tanenbaum
Издательство: Prentice Hall, 1987
ISBN: 0-13-637406-9
Прибл. Цена: $50.00

В то время как эта книга имеет немного упрощенное описание некоторых тем и опускает некоторые важные моменты, она дает достаточно четкое представление о том что надо сделать чтобы написать операционную систему. Пол книги занимает исходный код клона UN*X называемого Minix, который основывается на микроядре, в отличии от Linux, который славится монолитным дизайном. Было сказано что Minix показывает возможность написания UN*X, основанного на микроядре, но не обьясняется в достаточной степени зачем нужно это делать.

Певоначально предполагалось что Linux будет доступной заменой для Minix фактически он первоначально должен был быть совместимым на уровне двоичного кода с Minix-386. Minix-386 был средой разработки под которой был раскручен Linux. В Linux нет кода Minix, но признаки этого наследия проявляются в таких вещах как файловая система Minix в Linux.

На ранних порах существования Linux, Эндрю Таненбаум начал жаркую войну с Linus по поводу разработки ОС, которая была интересной, если не поучительной.

Однако эта книга может оказаться стоящй для тех кто ищет

доступного обьяснения основ ОС, так как в изложении Таненбаума они остаются наиболее понятными (и более занимательными, если вы не хотите скучать).

К сжалению упор делается на основы, в то время как такие вещи как виртуальная память не охвачены вообще.

Заглавие: Modern operating systems
Автор: Andrew S. Tanenbaum
Издательство: Prentice Hall, 1992
ISBN: 0-13-588187-0
Прибл. Цена: $51.75

первая половина книги это перепечатка более ранней Operating systems, но эта книга включает некоторые вещи не раскрытые в ранней, включая такие вещи как виртуальная память. Minix не упоминается, но есть обзор MS-DOS и нескольких других распостраненных систем. Эта книга вероятно более полезна для тех кто зочет углубить свои знания, чем более ранняя книга Таненбаума Operating systems: Design and Implementation. Причину этого можно видеть в заголовке.. Однако что делает DOS в книге по СОВРЕМЕННЫМ операционным системам многие не могут понять.

Заглавие: operating Systems
Автор: William Stallings
Издательство: Macmillan, 1992 (800-548-9939)
ISBN: 0-02-415481-4
Прибл. Цена: $??.??

Наиболее полнай текст по операционным системам, эта книга дает более глубокий подход к темам раскрытым в книге Таненбаума, и охватывает больше тем, имеет живой стиль мзложения. Эта книга охватывает все главные темы которые понадобятся вам для написания операционной системы, и делает это очень доступным образом. Автор использует примеры из трех главных систем сравнивая и противопоставляя их: UN*X, OS/2, и MVS. В каждом разделе эти системы используются для разяснения пунктов и приведения примеров реализации.

Темы охваченные в Operating Systems включают Нити (Связи), системы рельного времени, Планировка в Мультипроцессорах, распределенные системы, миграция процессов, и Безопасность, также как и стандартные темы как планировка управление памятью. Раздел по распределенной обработке похоже вполне современен, и я нахожу его очень полезным.

Заглавие: UNIX Network programming
Автор: W. Richard Stevens
Издательство: Prentice Hall, 1990
ISBN: 0-13-949876-1
Прибл. Цена: $48.75

Эта книга охватывает несколько видов работы в сетях под UN*X, и содержит очень полезные справки по формам сетевой обработки которые она непосредственно не охватывает. Она охватывает TCP/IP и XNS особенно полно, и довольно исчерпывающе описывает как работают все вызовы. В ней также есть описание и пример кода использующего TLI System V, и достаточно полное описание IPC System V. Книга содержит много примеров исходного кода и много полезных процедур. Один из примеров это код реализующий используемые семафоры, основанный на частично- фрагментированной реализации которая применяется в System V.

Заглавие: Programming in the UNIX environment
Автор: Brian W. Kernighan and Robert Pike
Издательство: Prentice Hall, 1984
ISBN: 0-13-937699 (hardcover) 0-13-937681-X (paperback)
Прибл. Цена: $??.??

Нет Аннотации.

Заглавие: Writing UNIX Device drivers
Автор: George Pajari
Издательство: Addison Wesley, 1992
ISBN: 0-201-52374-4

Прибл. Цена: $32.95

Эта книга написана президентом и основателем Driver Design Labs, компании специализирующейся на разработке драйверов устройств для UN*X. Эта книга отличное введение в порой суровый мир разработки драйверов устройств. Сначало кратко обсуждаются четыре основные типа драйверов (символьные, блочные, tty, STREAMS). Приведено множество полных примеров драйверов устройств разных типов, начиная с простейших и с растущей сложностью. Все примеры драйверов работают под UN*X на PC-совместимой аппаратуре. Включены следующие главы:

Character Drivers I: A Test Data Generator Character Drivers II: An A/D Converter Character Drivers III: A Line Printer Block Drivers I: A Test Data Generator Block Drivers II: A RAM Disk Driver Block Drivers III: A SCSI Disk Driver Character Drivers IV: The Raw Disk Driver Terminal Drivers I: The COM1 Port Character Drivers V: A Tape Drive STREAMS Drivers I: A Loop-Back Driver STREAMS Drivers II: The COM1 Port (Revisited) Driver Installation Zen and the Art of Device Driver Writting.

Хотя множество вызовов используемых в этой книге Linux- не

совместимы, присутствует общая идея, и большинство решений отображаются непосредственно в Linux.

Приложение B.

Обзор исходного текста ядра Linux.

В этой главе мы пытаемся по порядку об'яснить исходный текст ядра Linux, пытаясь помочь читателям понять, как структурирован исходный текст, и как описаны соответственные части Linux.Мы преследуем цель близко познакомить начинающего программиста на языке Си с общим устройством Linux.

В качестве отправной точки обзора возьмем загрузку системы.

Для осознания этого материала требуются хорошее знание языка Си и практически полное представление о концепциях UN*X и архитектуре ПК.

В этой главе, однако, вы не встретите текстов на Си, а увидите лишь ссылки на истинный текст. Подробная информация о ядре находится в предыдущих главах, здесь же мы можем увидеть лишь обзорный материал.

Любой путь, встречающийся в этой главе, относится к основному кодовому дереву каталогов, обычно это /usr/src/linux.

******
* Большинство информации, представленной в главе, взята из Linux 1.0, однако
* здесь встречаются ссылки на более поздние версии. Любой параграф в
* этой главе, выделенный так же, как этот, означает описание изменений
* в ядре по отношению к Linux 1.0. Если в параграфе нет подобных ссылок,
* это означает, что исходный текст не претерпел изменения в версиях 1.0.9 - 1.1.76.
* Иногда выделенные места появляются в тексте в качестве ссылок на исходный текст.
******

B.1. Загрузка системы.

Во время загрузки ПК процессор 80x86 запускается в режиме реального времени и запускает код ROM-BIOS по адресу 0xFFFF0. PS BIOS проводит тестирование системы и инициализирует вектор прерывания на 0-й физический адрес. После этого она загружает сектор загрузочного устройства по адресу 0x7C00 и обращается по этому адресу. Это устройство обычно представляет собой жесткий диск или накопитель в дисководе. Данная выкладка сильно упрощена, однако она дает представление о инициализации ядра.

Основная (первая) часть ядра Linux была написана на ассемблере 8086 (boot/bootsect.s). Во время запуска она помещает себя по абсолютному адресу 0x90000, считывая следующие 2Кб кода с загрузочного устройства по адресу 0x90200 и часть ядра по адресу 0x10000. Во время загрузки системы появляется сообщение

"Loading...". Далее контроль передается коду в boot/Setup.S (другой исходник режима реального времени на ассемблере).

Установленная часть определяет остальные компоненты системы и тип карты vga. Если нужно, она может дать право выбора

видеорежима. Затем она переносит всю систему с адреса 0x10000 по адресу 0x1000, включает защищенный режим и обращается к остальной части системы (по адресу 0x1000).

Следующим шагом является распаковка ядра. Код по адресу 0x1000 берется из zBoot/head.S, которая устанавливает регистры и вызывает decompress_kernel(), которая создает zBoot/inflate.c, zBoot/unzip.c и zBoot/misc.c. Распакованная информация помещается по адресу 0x1000000 (1Мб), поэтому Linux не может быть запущена на компьютере с ОЗУ, меньшим 1Мб.

****
* Сокрытие ядра в файле gzip делается Makefile и утилитами в каталоге zBoot.
* Среди них есть занимательные программы.
*
* Ядро версии 1.1.75 помещает каталоги boot и zBoot в arch/i386/boot. Это
* изменение позволяет ядру подстраиваться под разные архитектуры.
****

Распакованный код запускается по адресу 0x1010000, где делаются все 32- битные установки: загружаются IDT, GDF и LTD, производятся установки процессору и сопроцессору, устанавливаются страницы и вызывается подпрограмма start_kernel. Исходные тексты предыдущих операций находятся в boot/head.S. Это наиболее изощренный код во всем ядре.

Помните, что в случае возникновения ошибки во время выполнения предыдущих шагов компьютер остановит работу. ОС не может справляться с ошибками, если она не полностью установлена.

start_kernel помещается в init/main и никогда не прекращает работу.

Единственное, что до этого момента написано на Си - это управление прерываниями и системный вызов enter/leave (однако и здесь большинство макросов нвписано на ассемблере).

B.2

После обработки самых тонких вопросов, start_kernel инициализирует все по отдельности части ядра.

- Устанавливает границы памяти и вызывает paging_init().
- Устанавливает каналы IRQ и планирование.
- Грамматически разбирает командную строку.
- Если надо, распределяет памть под буффер.
- Устанавливает драйвера устройств и буферизацию диска, также как и други неосновные компоненты.
- Определяет циклическую задержку.
- Проверяет работает-ли прерывание 16 с сопроцессором.

После этого ядро готово к move_to_user_mode() (перемещение в пользовательский режим. Затем 0-й процесс, так называемая идеальная задача, продолжает функционировать в бесконечном цикле.

Процесс init петается запустить /exec/init, или /bin/init, или /sbin/init. Если ни один из перечисленных методов не запускается, система запускает "/bin/sh /etc/rc" и ведает основную оболочку на первый терминал. Эта процедура была написана в Linux 0.01, когда ОС состояла из одного ядра и не поддерживала операцию login.

После запуска функцией exec() программы init с одной из стандартных позиций (предположим что мы находимся в одной из них), ядро не контролирует процесс работы программы. Его ролью в этот момент становится поддерживать процессы с помощью системных вызовов и обслуживать асинхронные события, такие как прерывания аппаратного обеспечения. Многозадачность также устанавливается до этого, поэтому управлением доступа задач с помощью fork() и login занимается программа init.

Данный обзор рассмотрит обслуживание ядром асинхронные события, также подробно как размещение информации и организацию кода.

B.3 Как ядро рассматривает процесс.

С точки зрения ядра процесс есть ни что иное, как запись в таблице процессов. Таблица процессов - одна из важнейших структур данных внутри системы совместно с таблицей распределения памяти и механизмом кэширования буфера. Особое место в таблице процессов занимает довольно об'емная структура task_structure, определенная в include/linux/sched.h. Внутри структуры task_struct содержится информация как высокого, так и низкого уровня - от некоторых регистров аппаратного обеспечения до inode работающей директории процесса.

Таблица процессов является одновременно массивом и двусвязным списком в виде дерева. Физическое описание представляет собой статический массив указателей, длина которого NR_TASKS, является константой, определенной в include/linux/tasks.h, так что размер структуры может быть переопределен лишь на определенной зарезервированной странице. Структура списка определена двумя указателями next_task и prev_task, а структура дерева общеизвестна и мы не будем на ней здесь останавливаться. Вы можете изменить величину NR_TASKS со 128, как установлено по умолчанию, однако вам придется перекомпилировать все исходные файлы измененные при этом. После загрузки ядро работает от имени какого-либо процесса, используя глобальную переменную current и указатель на структуру task_struct для запуска прцессов. current изменятся только планировщиком в kernel/sched.c.

Когда надо закрыть все процессы, используется макрос for_each_task. Это намного быстрее нежели послеовательное считывание массива, когда система загружена неполностью.

Процесс работает одновременно и в режиме ядра, и в режиме пользователя. Оновная часть процесса запускается в

пользовательском режиме, системные вызовы запускаются в режиме ядра. Стеки, используемые работающими в разных режимах процессами, различны - определенный сегментный стек используется в пользовательском режиме, а режим ядра использует стек определенной величины.

Стековая страница ядра никогда не своппится, так как она должна быть доступна в любое время работы системы.

Системные вызовы внутри ядра существуют как функции на языке Си и их имена начинаются с префикса "sys_". Системный вызов burnout, к примеру, содержится в ядре в качествефункции

Sys_burnout().

```
*********
*  Механизм обработки системных вызовов описан в главе 3 этой книги.
*  Просмотр for_each_task и SET_LINKS в include/linux/shed.h может помочь
*  вам в понимании структуры списка и дерева в таблице процессов.
*********
```

B.4. Создание и удаление процесса.

Система unix создает процесс с помощью системного вызова fork(), удаление процесса может осуществляться с помощью exit() или с помощью передачи ядру сигнала. Описание этих функций в Linux расположено в kernel/fork.c и в kernel/exit.c.

Разветвление процессов устроено довольно просто, так как файл fork.c небольшой и хорошо читаемый. его главная задача - заполнение структуры данных нового процесса. Здесь представлены основные шаги процесса заполнения, исключая заполнение полей:

- Получение пустой страницы для помещения туда task_struct;
- Нахождение пустого слота для процесса (find_empty_process());
- Получение другой пустой страницы для kernel_stack_page;
- Копирование LDT родителя наследнику;
- Засылка копии информации из mmap родителю;

sys_fork() управляет дескрипторами и inode файлов.

```
************
*  В ядре версии 1.0 предлагается весьма несовершенная поддержка наследования
*  и системный вызов fork() хорошо демонстрирует это.
************
```

Выход из программы осуществляется в системе изощренным методом, так как каждый родительский процесс получает информацию ото всех своих существующих наследников. Кроме того, процесс может завершиться при kill() (уничтожении) другого процесса (позаимствовано из UN*X). Файл exit.c содержит sys_kill(), различные версии sys_wait() и sys_exit().

Текст exit.c не описывается здесь - он неинтересен. Он оперирует большим количеством инструментов выхода из системы в рабочем состоянии.

Стандарт POSIX управляет сигналами.

B.5. Запуск программы.

После разделения (fork()) запускаются две одинаковые программы. Одна из них обычно запускает (exec()) другую. Системный вызов exec() должен создать двоичный образ запускаемого файла, загрузить и запустить его. Слово "загрузить" не обязательно означает "запись в память двоичный образ", так как Linux поддерживает загрузку по требуемым частям.

Описание вызова exec() в Linux поддерживает разные форматы двоичного кода. Это содержится в структуре linux_binfmt, которая устанавливает два указателя на функции: один - на функцию запускаемого кода, второй - на загрузку библиотеки, каждый двоичный формат может представлять и запускаемые файлы, и библиотеки.

Загрузка нужных библиотек описана в том же исходном файле, что и exec(), но они позволяют подключать себя функции exec().

Системы UN*X позволяют программисту работать с шестью различными версиями функции exec(). Одна из них может быть описана, как библиотечная функция.

Также ядро Linux подключает отдельно функцию execve(). Она исполняет достаточно простую задачу - чтение заголовка запускаемого файла и попытку запустить его. Если первые два байта "#!", делается грамматический разбор и включается интерпретатор, иначе последовательно применяются другие двоичные форматы.

Родной формат Linux поддерживается прямо внутри fs/exec.c вместе с соответствующими функциями load_aout_binary и load_aout_library.

Что касается двоичных кодов, функция, загружаемая как запускаемый файл a.out, заканчивает работу после mmap() дискового файла, или после вызова read_exec(). Формальный метод, используемый Linux, требует загружаемый механизм для обнаружения ошибок в программных страницах, когда к ним открывается доступ, тогда как более новый метод используется, когда в основной файловой системе не подчеркивается распределение памяти (к примеру, файловая система "msdos").

****
* После версии 1.1 в ядра включались переделанные файловые системы msdos,
* поддерживающие mmap(). Более того, структура linux_binfmt как список
* для поддержки новых двоичных форматов в качестве модулей ядра. В итоге
* структура была расширена для доступа к подпрограммам конвертации форматов.
****

B.6. Доступные файловые системы.

Всем известно, что файловая система - основной ресурс системы UN*X, настолько основной и общей, что она должна иметь удобное сокращение имени. Далее в тексте будем называть файловую систему "фс".

Я предполагаю, что читатель уже знаком с с основными концепциями фс UNIX - разрешение доступа, inode, superblock, mounting и umounting. Эти концепции об'яснены в других книгах по UNIX, так что я не буду повторяться, а остановлюсь на особых компонентах Linux.

Первые UNIX-системы поддерживали одну файловую систему, структуру, которая была занесена прямо в ядро. На данный момент используется нестандартный интерфейс для создания коммуникации между ядром и файловой системой в порядке непринужденного обмена информацией между архитектурами. Сам Linux поддерживает стандартный метод обмена информацией между ядром и каждым модулем. Этот метод называется VFS.

Текст файловой системы тем самым разбивается на две части: верхняя часть, связанная с распределением таблиц ядра и структур данных, и нижняя часть, созданная для установки функций, зависящих от фс и вызываемых через структуры данных VFS.

Весь материал, не зависящий от фс, хранится в файлах fs/*.c. Они выполняют следующие операции:

- Управление кешированием буфера;
- Ответ на системные вызовы fcntl() и ioctl() (fcntl.c и ioctl.c);
- Распределение pipe и fifo на inode и буферах (fifo.c и pipe.c);
- Управление файловыми и inode таблицами (file_table.c и inode.c);
- Закрытие и открытие файлов и записей (locks.c);
- Распределение имен в inode (namei.c, open.c);
- Описание функции select() (select.c);
- Информационная база (stat.c);
- mounting и umounting фс (super.c);
- Запуск (exec()) запускаемых файлов и загрузка библиотек (exec.c);
- Загрузка различных двоичных форматов (bin_fmt*.c, как описано выше).

Интерфейс VFS содержит набор операций высокого уровня, запускаемых независимым от фс кодом, и представляется нужном формате для фс. Наиболее важными структурами являются

file_operations и inode_operations, однако, они далеко не единственны. Все они описаны в include/linux/fs.h.

Отправной точкой в ядре в обращении к файловой системе является структура file_system_type. Массив file_system_types помещен в fs/filesystems.c, и во время запуска mount на него происходит ссылка. Затем функция read_super для соответствующего типа фс заполняет элемент структуры struct super_block,который, в свою очередь, заносит информацию в struct super_operations и в struct type_sb_info.

Создатель устанавливает указатели на главные операции для данного типа фс, последняя также указывает специальную информацию для типа файловой системы.

```
**********
*  Массив типов файловой системы помещен в скомпанованный список для организации
*  новых типов фс как модулей ядра. Функция, делающая это - (un-)register_filesystem,
*  описана в fs/super.c.
**********
```

B.7. Краткий обзор сущности типа файловой системы.

Роль типа файловой системы состоит в выполнении низкоуровневых задач, используемых для распределения высокоуровневых операций VFS на физических устройствах.

Интерфейс VFS достаточно универсален для поддержки обеих встроенных фс UN*X и более экзотичных, таких, как msdos и umsdos.

Каждая фс создается из следующих компонентов, принадлежащих ее собственным каталогам:

- Запись в массиве file_streams[] (fs/filesystems.c);
- Суперблочный файл include (include/linux/type_fs_sb.h);
- inode include файл (include/linux/type_fs_i.h);
- Основной include файл (include/linux/type_fs.h);

- Две строки #include внутри include/linux/fs.h, также как и запись в структуры super_block и inode.

Собственная директория типа фс содержит исходные тексты, обладающие inode и осуществляющие управление обработкой информации.

****
* Глава про фс proc в этой книге содержит все подробности о низкоуровневом коде
* и интерфейсе VFS для этого типа фс. Исходный текст в fs/procfs достаточно
* доступен после прочтения этой главы.
****

Разберем внутреннее устройство механизма VFS и фс minix в качестве примера. Я выбрал в качестве примера minix, так как она небольшая, но полная, кроме того, все фс linux берут начало от minix. Читателю предлагается разобрать в качестве упражнения тип ext2, встречающийся в инсталляции Linux.

Во время поддержки системой фс minix minix_read_super заполняет структуру super_block информацией, полученной с поддерживаемого устройства. Поле s_op структуры содержит указатель на minix_sops, используемый основным кодом фс для быстрого выполнения операции суперблока.

Соединение новой поддерживаемой фс с системой основывается на изменении следующих компонент (помещение sb в super_block и dir_i в место обращения):
- sb->s_mounted указывает на inode корневого каталога поддерживаемой фс (MINIX_ROOT_INO);
- dir_i->i_mount, содержащий sb->s_mounted;
- sb->s_covered, содержащий dir_i;

Umount происходит с помощью do_umount, включающим запуск minix_put_super. Когда разрешен доступ к файлу, minix_read_inode заполняет общую системную inode структуру полями из minix_inode. Поле inode->i_op заполняется, исходя из inode->i_mode и отвечает за все будующие операции над файлом.

Описание исходных текстов функций minix вы можете найти в fs/minix/inode.c. Структура inode_operations используется для засылки inode операций в специальные функции ядра; первая запись в структуре - указатель на file_operations, которая информационно эквивалентна i_op. Фс minix позволяет выбрать три образца наборов inode - операций (для каталогов, для файлов и для скомпанованных символов) и два образца установки file_operations.

Операции над каталогами (одна minix_readdir) могут быть найдены в fs/minix/dir.c, операции над файлами - в fs/minix/file.c, и операции над скомпанованными символами - в fs/minix/symlink.c.

Остальная часть каталога minix предназначена для следующих задач:

- bitmap.c управляет распределением и очисткой inode и блоков (ext2 имеет два разных файла).
- fsynk.c ответственнен за системные вызовы fsync() - управление прямыми, непрямыми и сдвоенными непрямыми блоками. (Я надеюсь, вы имеете о них представление из UN*X).
- namei.c включает в себя inode-операции, связанные с именами, такие как создание и удаление node, переименование, компановка.
- truncate.c выполняет усечение файлов.

B.8. Пультовый драйвер.

Будучи драйвером ввода/вывода в большинстве компонентов Linux, пультовый драйвер заслуживает внимания. Исходный текст имеет такое же отношение к управлению, как и любой другой символьный драйвер, находящийся в /drivers/chart, и мы будем мы будем использовать эту директорию при ссылке на имена файлов.

Инициализация управления происходит с помощью функции tty_init() в tty_io.c. Эта функция предназначена для получения основных номеров устройств и вызова инициализации каждого

установленного устройства. con_init() - одна из функций, относящихся к управляющему драйверу, инициализирующая его, находится в console.c.

```
**********
*  Инициализация управляющего устройства сильно изменилась после выпуска версии
*  1.1, была убрана из tty_init() и вызывается прямо из../../main.c. Виртуальные
*  пульты на данный момент динамически распределяемы, и в них изменена большая
*  часть исходного текста.
**********
```

B.8.1. Как файловые операции посылаются пульту.

Этот параграф довольно низкого уровня, и может быть исключена из прочитываемого. Доступ к устройству UN*X осуществляется через файловую систему. Этот параграф описывает все шаги файла устройства к функциям пульта. Кроме того, эта информация взята из исходных текстов версии 1.1.73, и может отличаться от исходников 1.0.

Когда открывается inode устройства, запускается функция chrdev_open() (или blkdev_open(), мы будем рассматривать символьные устройства). Эта функция полна компонентами структуры def_chr_fops, на которую ссылаются chrdev_inode_operations, используемой всеми типами фс.

chrdev_open заботится о спецификации операций над устройством,помещая собственную таблицу file_operations в текущий flip и вызывая специфицированную open(). Специфицированные таблицы устройства содержатся в массиве chrdevs[], индексированном по основным номерам устройств и заполняемом тем же

../../fs/devices.c.

Если мы рассматриваем tty устройство (нужен ли нам в таком случае пульт ?), мы переходим к драйверам tty, чьи функции находятся в tty_io.c, индексированные tty_fops. tty_open()

вызывает init_dev(), которая выделяет любую структуру данных, нужную устройству, базируемую на подномере устройства.

Подномер также используется для поиска фактического драйвера для устройства, который был зарегистрирован через tty_register_driver(). Драйвер в таком случае представляет собой иную структуру, используемую для определения подмодулей, таких, как file_ops; он напямую связан с записью и контролем над устройством. Последняя структура, используемая в управлении tty,

это линейная дисциплина, описываемая позже.

Линейная дисциплина для пульта (или любого другого устройства tty) устанавливается с помощью функции initialize_tty_struct(), запускаемой init_dev.

Все, что мы рассматривали в этом параграфе, не зависит от самих драйверов. Только специальная пультовая часть, расположенная в console.c, регистрирует свой собственный драйвер во время работы con_init(). В целом, линейная дисциплина также не зависит от устройства.

```
*************
*  Структура tty_driver полностью определена внутри .
*    Предыдущая информация была взята из исходного текста версии 1.1.73. Он не
*    похож на используемое вами ядро.
*************
```

B.8.2. Передача информации пульту.

Когда происходит запись в пультовое устройство, вызывается функция con_write(). Эта функция управляет всеми контрольными символами и esc-последовательностями, используемыми для поддержки приложений, связанных со всем управлением экрана. Эти esc-последовательности определены в коммуникационной подпрограмме vt102. Это означает, что вы должны установить TERM=vt102, когда вы хотите передать информацию не Linux-овскому host адаптеру, однако для локальных целей лучше устанавливать TEMP=console, так как пульт Linux позволяет оптимально установить vt102.

con_write() на большую часть состоит из вложенных установок переключателей, используемых для посимвольной интерпритации esc-последовательностей. В нормальном режиме, символ выводитсяя на экран, будучи записанным в видео память, используя определенные атрибуты. Внутри console.c, все поля структуры struct vc становятся доступными лишь через макросы, так что любая ссылка на attr (к примеру), на самом деле приходится на поле в структуре vc_cons[currcons], до тех пор пока система не перестает ссылаться на номер данного пульта.

```
************
*  В новых ядрах, vc_cons представляет собой массив указателей, содержание
*  которых распределено в памяти kmalloc(). Использование макросов
*  сильно упростило изменения, так как в переписывании нуждалась лишь
*  небольшая часть кода.
************
```

Непосредственное распределение памяти пульта на экран осуществляется функциями set_scrmem() (копирование информации из буфера пульта в видео память% и get_scrmem() (копирование информации обратно в буфер). Личный буфер конкретного пульта физически расположен прямо в видео RAM, для уменьшения количества передач информации. Это означает, что функции get- и set- stream() являются static(статическими) для console.c и вызываются только во время переключения пульта.

B.8.3 Чтение из пульта.

Чтение из пульта устроено через линейную дисциплину. По умолчанию Linux пользуется линейной дисциплиной tty_ldisc_N_TTY. Линейная дисциплина - это метод разбора компонентов передаваемых в строке. Она является еще одной таблицей функций, которая работает

при чтении устройства. С помощью флагов termios, линейная дисциплина контролирует ввод из tty в режимах raw,cbreak и cooked, а также работу функциий select(), ioctl() и подобных.

Функция чтения в линейной дисциплине называется read_chan().

Она осуществляет чтение из буфера tty в зависимости от того, что он представляет. Причина по которым символы передаются через tty обусловлена асинхронными прерываниями аппаратного обеспечения.

```
**************
* Линейная дисциплина N_TTY находится в том-же tty_io.c, более
* поздние ядра также подключают исходник n_tty.c
**************
```

Самым низкоуровневым передатчиком информации пульту, является менеджер клавиатуры, описанный в keyboard.c, в функции keyboard_interrupt().

B.8.4 Управление клавиатурой.

Управление клавиатурой реализовано крайне необдуманно. В keyboard.c определены все десятичные значения различных кодов клавиатуры различных производителей.

В keyboard.c истинный хакер не найдет для себя никакой полезной информации.

```
**************
*  Читателям действительно интересующихся образом клавиатуры
*  в Linux, советую просматривать файл keyboard.c с конца, так как
*  подробности управления на низком уровне могут находится лишь в
*  первой половине файла.
**************
```

B.8.5 Переключение пультов.

Текущий пульт переключается через запуск функции change_console(), которая переопределяет размер tty_io.c, осуществляемый либо keyboard.c, либо vt.c (Пользователь переключает их нажатием клавиши, программа вызовом ioctl())

Переключение происходит в два этапа, и функция

complete_change_console() отвечает за второй. Разрыв переключения обусловлен окончанием работы задачи, с предварительным сообщением об этом процессу контролируемому покилаемую нами tty. Если пульт не подчиняется контролирующему процессу, функция change_console() вызывает complete_change_console() самостоятельно. Для смены графического пульта на текстовый нужен процесс конверции, при этом отдельный сервер может продолжать работать с графическим пультом.

B.8.6 Механизм выбора пульта.

"selection" - это устройство службы вырезания и копирования для текстовых пультов. Этот механизм поддерживается процессом пользовательского уровня который может быть запущен selection или gpm. Программа пользовательского уровня использует ioctl() в работе с пультом, для сообщения ядру точного места подсветки

текста на экране. Затем выбранный текст помещается в буфер. Этот буфер статически определен в console.c. Копирование текста связывается с обычным помещением символов из буфера в входную очередь tty. Весь механизм выбора защищен #ifdef, так что пользователь может запретить его во время сохранения конфигурации ядра в несколько килобайт памяти.

Выбор является низко уровневым методом, поэтому он его деятельность не доступна другим процессам. Это означает, что большинство прстейших операций #ifdef удаляющих выделение текста в любом случае изменяется.

```
**********
*  В версиях Linux код прцесса выбора не улучшился со времени создания.
*  Единственное изменение произошло после внедрения динамического
*  буфера вместо статического, делающее ядро меньше на 4 Кб.
**********
```

B.8.7 Контроль над вводом-выводом устройства (ioctl()).

Системный вызов ioctl(), является отправной точкой

пользовательских процессов, контролирующих поведение файла устройства. Управление передачи контроля находится в ../../fs/ioctl.c, где расположен sys_ioctl(). Стандартные запросы на передачу контроля удовлетворяются прямо здесь, иные запросы, связанные с файлами довлетворяются с помощью file_ioctl() (находится в том-же исходнике), до тех пор пока следующий запрос не обратится к особой функции ioctl() устройства.

Информация о контроле над пультовыми устройствами находится в vt.c, так как пультовое устройство удовлетворяет ioctl - запросы функцией vt_ioctl().

```
**********
* Вышеописанная информация взята из версии 1.1.7x. Ядро 1.0 не
* имела таблицы драйверов, и vt_ioctl() находился прямо в
* таблице file_operations().
**********
```

В серии 1.1.7x обозначены следующие вещи: tty_ioctl.c описывает только запросы на линейные дисциплины (за исключением функции n_tty_ioctl(), являющейся единственной функцией n_tty вне n_tty.c), в то время как поле file_operations указывает на tty_ioctl() в tty_io.c. Если номер запроса не не определяется в tty_ioctl(), он передается в tty->driver.ioctl или в случае провала в tty->ldisc.ioctl.

Материал по линейным дисциплинам находится в tty_ioctl.c, в то время как информация о пультовых драйверах в vt.c.

В Ядре 1.0, tty_ioctl() находится в tty_ioctl.c и указывает на него общая file_operations. Нераспознанные запросы проходят через определенный контроль или через код линейной дисциплины похожий на версию 1.1.7x.

Помните что в обоих случаях запрос TIOCLINUX не зависит от устройства. Это говорит о том, что выбор пульта может быть установлен ioctlом любого tty.

Вы можете встреть множество разнообразных устройств, относящихся к пультовому устройству, и лучший способ познать их - изучить исходный текст vt.c.