Маленькая книга о разработке операционных систем

Авторы: Эрик Хелин, Адам Ренберг
Дата: 19.01.2015 | Коммит: fe83e27dab3c39930354d2dea83f6d4ee2928212

Содержание

1. Введение

Эта книга представляет собой практическое руководство по написанию собственной операционной системы для архитектуры x86. Она дает достаточно технических деталей, но при этом не раскрывает слишком много в примерах и фрагментах кода. Мы постарались собрать материалы из огромного (и часто превосходного) множества доступных источников и добавить собственные идеи о проблемах, с которыми мы столкнулись.

Эта книга не о теории операционных систем или о том, как работают конкретные ОС. Для изучения теории мы рекомендуем книгу "Современные операционные системы" Эндрю Таненбаума [1]. Списки и детали современных операционных систем доступны в Интернете.

Начальные главы очень подробны и конкретны, чтобы быстро ввести вас в процесс кодирования. В последующих главах дается больше общих указаний, так как все больше реализации и проектирования остается на усмотрение читателя, который к этому моменту уже должен быть более знаком с миром разработки ядер.

В главах 2 и 3 мы настраиваем среду разработки и загружаем ядро нашей ОС в виртуальной машине, постепенно переходя к написанию кода на C. В главе 4 мы продолжаем работу с выводом на экран и последовательный порт, затем углубляемся в сегментацию (глава 5) и прерывания с вводом (глава 6).

После этого у нас будет вполне функциональное, но минимальное ядро ОС. В главе 7 мы начинаем путь к пользовательским приложениям, с виртуальной памятью через страничную организацию (главы 8 и 9), распределением памяти (глава 10) и, наконец, запуском пользовательского приложения в главе 11.

В последних трех главах обсуждаются более продвинутые темы: файловые системы (глава 12), системные вызовы (глава 13) и многозадачность (глава 14).

1.1 О книге

Ядро ОС и эта книга были созданы в рамках индивидуального курса в Королевском технологическом институте [2] в Стокгольме. Авторы ранее изучали теорию операционных систем, но имели небольшой практический опыт разработки ядер. Чтобы глубже понять, как теория из предыдущих курсов работает на практике, авторы решили создать новый курс, сосредоточенный на разработке небольшой ОС. Другой целью курса было написание подробного руководства по разработке небольшой ОС практически с нуля, и результатом стала эта книга.

Архитектура x86 долгое время была одной из самых распространенных. Выбор x86 в качестве цели для ОС был очевиден благодаря большому сообществу, обширным справочным материалам и зрелым эмуляторам. Однако документация и информация о деталях оборудования не всегда были легкодоступны или понятны, несмотря на (или, возможно, из-за) возраст архитектуры.

ОС разрабатывалась около шести недель полного рабочего дня. Реализация выполнялась небольшими шагами, и после каждого шага ОС тестировалась вручную. Такой инкрементальный и итеративный подход часто упрощал поиск ошибок, так как только небольшая часть кода изменялась с момента последнего рабочего состояния. Мы рекомендуем читателю работать аналогичным образом.

Почти каждая строка кода была написана авторами вместе (этот стиль работы также называется парным программированием). Мы считаем, что такой подход помог избежать многих ошибок, хотя это трудно доказать научно.

1.2 Для кого эта книга

Читатель этой книги должен быть знаком с UNIX/Linux, системным программированием, языком C и компьютерными системами в целом (например, с шестнадцатеричной нотацией [3]). Эта книга может быть способом начать изучение этих вещей, но это будет сложнее, поскольку разработка операционной системы сама по себе является непростой задачей. Поисковые системы и другие руководства часто помогают, если вы застряли.

1.3 Благодарности

Мы благодарим сообщество OSDev [4] за их отличную вики и полезных участников, а также Джеймса Маллоя за его замечательный учебник по разработке ядер [5]. Мы также благодарим нашего руководителя Торбьёрна Гранлунда за его проницательные вопросы и интересные обсуждения.

Большая часть CSS-форматирования книги основана на работе Скотта Чакона для книги "Pro Git", http://progit.org/.

1.4 Участники

Мы очень благодарны за патчи, которые присылают нам люди. Следующие пользователи внесли свой вклад в эту книгу:

alexschneider
Avidanborisov
nirs
kedarmhaswade
vamanea
ansjob

1.5 Изменения и исправления

Эта книга размещена на Github - если у вас есть предложения, комментарии или исправления, просто форкните книгу, внесите изменения и отправьте нам pull request. Мы с радостью включим все, что сделает эту книгу лучше.

1.6 Проблемы и где получить помощь

Если у вас возникли проблемы при чтении книги, проверьте issues на Github: https://github.com/littleosbook/littleosbook/issues.

1.7 Лицензия

Весь контент распространяется под лицензией Creative Commons Attribution Non Commercial Share Alike 3.0, http://creativecommons.org/licenses/by-nc-sa/3.0/us/. Примеры кода находятся в общественном достоянии - используйте их как хотите. Отзывы об этой книге всегда принимаются с благодарностью.

2. Первые шаги

Разработка операционной системы (ОС) - непростая задача, и вопрос "Как вообще подступиться к этой проблеме?" будет возникать несколько раз в ходе проекта. Эта глава поможет вам настроить среду разработки и загрузить очень маленькую (и примитивную) операционную систему.

2.1 Инструменты

2.1.1 Быстрая настройка

Мы (авторы) использовали Ubuntu [6] в качестве операционной системы для разработки ОС, запуская ее как физически, так и виртуально (с помощью виртуальной машины VirtualBox [7]). Быстрый способ получить все необходимое - использовать ту же настройку, что и мы, поскольку мы знаем, что эти инструменты работают с примерами, приведенными в этой книге.

После установки Ubuntu (физически или виртуально) следующие пакеты должны быть установлены с помощью apt-get:

```
sudo apt-get install build-essential nasm genisoimage bochs bochs-sdl
```

2.1.2 Языки программирования

Операционная система будет разрабатываться на языке программирования C [8][9] с использованием GCC [10]. Мы используем C, потому что разработка ОС требует очень точного контроля над генерируемым кодом и прямого доступа к памяти. Можно использовать и другие языки, предоставляющие те же возможности, но эта книга будет рассматривать только C.

Код будет использовать один атрибут типа, специфичный для GCC:

```
__attribute__((packed))
```

Этот атрибут позволяет гарантировать, что компилятор использует layout структуры точно так, как мы определили его в коде. Это подробнее объясняется в следующей главе.

Из-за этого атрибута примеры кода может быть сложно скомпилировать с помощью компилятора, отличного от GCC.

Для написания ассемблерного кода мы выбрали NASM [11] в качестве ассемблера, так как нам больше нравится синтаксис NASM по сравнению с GNU Assembler.

Bash [12] будет использоваться в качестве языка сценариев на протяжении всей книги.

2.1.3 Операционная система хоста

Все примеры кода предполагают, что код компилируется в UNIX-подобной операционной системе. Все примеры кода были успешно скомпилированы с использованием Ubuntu [6] версий 11.04 и 11.10.

2.1.4 Система сборки

Make [13] использовался при создании примеров Makefile.

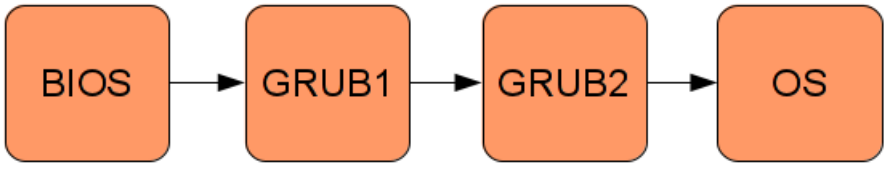
2.1.5 Виртуальная машина

При разработке ОС очень удобно запускать код в виртуальной машине, а не на физическом компьютере, поскольку запуск ОС в виртуальной машине происходит гораздо быстрее, чем перенос ОС на физический носитель и запуск на физической машине. Bochs [14] - это эмулятор для платформы x86 (IA-32), который хорошо подходит для разработки ОС благодаря своим возможностям отладки. Другие популярные варианты - QEMU [15] и VirtualBox [7]. В этой книге используется Bochs.

Используя виртуальную машину, мы не можем гарантировать, что наша ОС будет работать на реальном физическом оборудовании. Среда, эмулируемая виртуальной машиной, разработана так, чтобы быть очень похожей на физические аналоги, и ОС можно протестировать на физической машине, просто скопировав исполняемый файл на CD и найдя подходящую машину.

2.2 Загрузка

Загрузка операционной системы состоит из передачи управления по цепочке небольших программ, каждая из которых более "мощная", чем предыдущая, где операционная система является последней "программой". Смотрите следующий рисунок для примера процесса загрузки:



Пример процесса загрузки. Каждый прямоугольник представляет программу.

2.2.1 BIOS

Когда компьютер включается, он запускает небольшую программу, соответствующую стандарту Basic Input Output System (BIOS) [16]. Эта программа обычно хранится в микросхеме постоянной памяти на материнской плате компьютера. Изначальная роль программы BIOS заключалась в предоставлении некоторых библиотечных функций для вывода на экран, чтения ввода с клавиатуры и т.д. Современные операционные системы не используют функции BIOS, они используют драйверы, которые взаимодействуют с оборудованием напрямую, минуя BIOS. Сегодня BIOS в основном выполняет некоторые ранние диагностики (POST - power-on self-test), а затем передает управление загрузчику.

2.2.2 Загрузчик

Программа BIOS передаст управление компьютера программе, называемой загрузчиком. Задача загрузчика - передать управление нам, разработчикам операционной системы, и нашему коду. Однако из-за некоторых ограничений оборудования и обратной совместимости загрузчик часто разделен на две части: первая часть загрузчика передает управление второй части, которая, в свою очередь, передает управление ОС.

Написание загрузчика подразумевает написание большого количества низкоуровневого кода, взаимодействующего с BIOS. Поэтому мы будем использовать существующий загрузчик: GNU GRand Unified Bootloader (GRUB) [17].

Используя GRUB, операционная система может быть собрана как обычный исполняемый файл ELF [18], который будет загружен GRUB в правильное место в памяти. Компиляция ядра требует, чтобы код был размещен в памяти определенным образом (как компилировать ядро, будет обсуждаться позже в этой главе).

2.2.3 Операционная система

GRUB передаст управление операционной системе, перейдя по адресу в памяти. Перед переходом GRUB проверит наличие "волшебного числа", чтобы убедиться, что он переходит к ОС, а не к случайному коду. Это "волшебное число" является частью спецификации multiboot [19], которой соответствует GRUB. Как только GRUB выполнит переход, ОС получит полный контроль над компьютером.

2.3 Hello Cafebabe

В этом разделе описана реализация минимально возможной ОС, которую можно использовать вместе с GRUB. Единственное, что будет делать ОС, - это записывать значение 0xCAFEBABE в регистр eax (большинство людей, вероятно, даже не назвали бы это ОС).

2.3.1 Компиляция операционной системы

Эта часть ОС должна быть написана на ассемблере, так как C требует наличия стека, который пока недоступен (глава "Переход к C" описывает, как его настроить). Сохраните следующий код в файле loader.s:

```
global loader                   ; символ входа для ELF

MAGIC_NUMBER equ 0x1BADB002     ; определяем магическое число
FLAGS        equ 0x0            ; флаги multiboot
CHECKSUM     equ -MAGIC_NUMBER  ; вычисляем контрольную сумму
                                ; (магическое число + контрольная сумма + флаги должны равняться 0)

section .text:                  ; начало секции кода
align 4                         ; код должен быть выровнен по 4 байтам
    dd MAGIC_NUMBER             ; записываем магическое число в машинный код,
    dd FLAGS                    ; флаги,
    dd CHECKSUM                 ; и контрольную сумму

loader:                         ; метка загрузчика (определена как точка входа в скрипте линкера)
    mov eax, 0xCAFEBABE         ; помещаем число 0xCAFEBABE в регистр eax
.loop:
    jmp .loop                   ; бесконечный цикл
```

Единственное, что делает эта ОС, - записывает очень специфическое число 0xCAFEBABE в регистр eax. Очень маловероятно, что это число окажется в регистре eax, если ОС не поместила его туда.

Файл loader.s можно скомпилировать в 32-битный объектный файл ELF [18] следующей командой:

```
nasm -f elf32 loader.s
```

2.3.2 Линковка ядра

Теперь код необходимо слинковать для создания исполняемого файла, что требует некоторых дополнительных размышлений по сравнению с линковкой большинства программ. Мы хотим, чтобы GRUB загрузил ядро по адресу памяти, большему или равному 0x00100000 (1 мегабайт (МБ)), поскольку адреса ниже 1 МБ используются самим GRUB, BIOS и памятью, отображенной на устройства ввода-вывода. Поэтому необходим следующий скрипт линкера (написанный для GNU LD [20]):

```
ENTRY(loader)                /* имя метки входа */

SECTIONS {
    . = 0x00100000;          /* код должен быть загружен по адресу 1 МБ */

    .text ALIGN (0x1000) :   /* выравнивание по 4 КБ */
    {
        *(.text)             /* все текстовые секции из всех файлов */
    }

    .rodata ALIGN (0x1000) : /* выравнивание по 4 КБ */
    {
        *(.rodata*)          /* все секции только для чтения из всех файлов */
    }

    .data ALIGN (0x1000) :   /* выравнивание по 4 КБ */
    {
        *(.data)             /* все секции данных из всех файлов */
    }

    .bss ALIGN (0x1000) :    /* выравнивание по 4 КБ */
    {
        *(COMMON)            /* все COMMON секции из всех файлов */
        *(.bss)              /* все bss секции из всех файлов */
    }
}
```

Сохраните скрипт линкера в файл link.ld. Исполняемый файл теперь можно слинковать следующей командой:

```
ld -T link.ld -melf_i386 loader.o -o kernel.elf
```

Итоговый исполняемый файл будет называться kernel.elf.

2.3.3 Получение GRUB

Версия GRUB, которую мы будем использовать, - это GRUB Legacy, поскольку образ ISO может быть сгенерирован на системах, использующих как GRUB Legacy, так и GRUB 2. В частности, мы будем использовать загрузчик stage2_eltorito из GRUB Legacy. Этот файл может быть собран из GRUB 0.97, загрузив исходный код с ftp://alpha.gnu.org/gnu/grub/grub-0.97.tar.gz. Однако скрипт configure плохо работает с Ubuntu [21], поэтому бинарный файл можно загрузить с http://littleosbook.github.com/files/stage2_eltorito. Скопируйте файл stage2_eltorito в папку, где уже находятся loader.s и link.ld.

2.3.4 Создание ISO-образа

Исполняемый файл должен быть размещен на носителе, который может быть загружен виртуальной или физической машиной. В этой книге мы будем использовать образы ISO [22] в качестве носителя, но можно также использовать образы дискет, в зависимости от того, что поддерживает виртуальная или физическая машина.

Мы создадим образ ISO ядра с помощью программы genisoimage. Сначала необходимо создать папку, которая будет содержать файлы, находящиеся на ISO-образе. Следующие команды создают структуру папок и копируют файлы в нужные места:

```
mkdir -p iso/boot/grub              # создаем структуру папок
cp stage2_eltorito iso/boot/grub/   # копируем загрузчик
cp kernel.elf iso/boot/             # копируем ядро
```

Необходимо создать конфигурационный файл menu.lst для GRUB. Этот файл сообщает GRUB, где находится ядро, и настраивает некоторые параметры:

```
default=0
timeout=0

title os
kernel /boot/kernel.elf
```

Поместите файл menu.lst в папку iso/boot/grub/. Содержимое папки iso теперь должно выглядеть следующим образом:

```
iso
|-- boot
  |-- grub
  | |-- menu.lst
  | |-- stage2_eltorito
  |-- kernel.elf
```

ISO-образ можно сгенерировать следующей командой:

```
genisoimage -R                              \
            -b boot/grub/stage2_eltorito    \
            -no-emul-boot                   \
            -boot-load-size 4               \
            -A os                           \
            -input-charset utf8             \
            -quiet                          \
            -boot-info-table                \
            -o os.iso                       \
            iso
```

Для получения дополнительной информации о флагах, используемых в команде, см. руководство по genisoimage.

ISO-образ os.iso теперь содержит исполняемый файл ядра, загрузчик GRUB и конфигурационный файл.

2.3.5 Запуск Bochs

Теперь мы можем запустить ОС в эмуляторе Bochs, используя ISO-образ os.iso. Bochs нужен конфигурационный файл для запуска, и пример простого конфигурационного файла приведен ниже:

```
megs:            32
display_library: sdl
romimage:        file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage:     file=/usr/share/bochs/VGABIOS-lgpl-latest
ata0-master:     type=cdrom, path=os.iso, status=inserted
boot:            cdrom
log:             bochslog.txt
clock:           sync=realtime, time0=local
cpu:             count=1, ips=1000000
```

Возможно, вам потребуется изменить путь к romimage и vgaromimage в зависимости от того, как вы установили Bochs. Дополнительную информацию о конфигурационном файле Bochs можно найти на сайте Bochs [23].

Если вы сохранили конфигурацию в файле с именем bochsrc.txt, то Bochs можно запустить следующей командой:

```
bochs -f bochsrc.txt -q
```

Флаг -f указывает Bochs использовать данный конфигурационный файл, а флаг -q говорит Bochs пропустить интерактивное стартовое меню. Теперь вы должны увидеть, как Bochs запускается и отображает консоль с некоторой информацией от GRUB.

После выхода из Bochs отобразите журнал, созданный Bochs:

```
cat bochslog.txt
```

Где-то в выводе вы должны увидеть содержимое регистров CPU, эмулируемого Bochs. Если вы найдете RAX=00000000CAFEBABE или EAX=CAFEBABE (в зависимости от того, запускаете ли вы Bochs с поддержкой 64 бит или без), значит, ваша ОС успешно загрузилась!

2.4 Дополнительное чтение

Густаво Дуарте написал подробную статью о том, что на самом деле происходит при загрузке компьютера x86, http://duartes.org/gustavo/blog/post/how-computers-boot-up
Густаво продолжает описывать, что делает ядро на самых ранних этапах, http://duartes.org/gustavo/blog/post/kernel-boot-process
Вики OSDev также содержит хорошую статью о загрузке компьютера x86: http://wiki.osdev.org/Boot_Sequence

3. Переход к C

Эта глава покажет вам, как использовать C вместо ассемблера в качестве языка программирования для ОС. Ассемблер очень хорош для взаимодействия с CPU и обеспечивает максимальный контроль над каждым аспектом кода. Однако, по крайней мере для авторов, C - гораздо более удобный язык. Поэтому мы хотели бы использовать C как можно больше и использовать ассемблер только там, где это имеет смысл.

3.1 Настройка стека

Одно из требований для использования C - наличие стека, так как все нетривиальные программы на C используют стек. Настройка стека не сложнее, чем заставить регистр esp указывать на конец области свободной памяти (помните, что стек растет в сторону меньших адресов на x86), которая правильно выровнена (рекомендуется выравнивание по 4 байтам для производительности).

Мы могли бы указать esp на случайную область памяти, так как пока в памяти есть только GRUB, BIOS, ядро ОС и некоторая память, отображенная на устройства ввода-вывода. Это не очень хорошая идея - мы не знаем, сколько памяти доступно или используется ли область, на которую указывает esp, чем-то другим. Лучшая идея - зарезервировать кусок неинициализированной памяти в секции bss файла ELF ядра. Лучше использовать секцию bss вместо секции data, чтобы уменьшить размер исполняемого файла ОС. Поскольку GRUB понимает ELF, GRUB выделит любую память, зарезервированную в секции bss, при загрузке ОС.

Псевдоинструкция NASM resb [24] может использоваться для объявления неинициализированных данных:

```
KERNEL_STACK_SIZE equ 4096                  ; размер стека в байтах

section .bss
align 4                                     ; выравнивание по 4 байтам
kernel_stack:                               ; метка указывает на начало памяти
    resb KERNEL_STACK_SIZE                  ; резервируем стек для ядра
```

Не нужно беспокоиться об использовании неинициализированной памяти для стека, так как невозможно прочитать ячейку стека, в которую не было записи (без ручного манипулирования указателями). (Правильная) программа не может извлечь элемент из стека, не поместив его туда сначала. Поэтому ячейки памяти стека всегда будут записаны перед тем, как их прочитают.

Указатель стека затем настраивается путем указания esp на конец kernel_stack памяти:

```
mov esp, kernel_stack + KERNEL_STACK_SIZE   ; указываем esp на начало стека (конец области памяти)
```

3.2 Вызов кода на C из ассемблера

Следующий шаг - вызов функции на C из ассемблерного кода. Существует множество различных соглашений о том, как вызывать код на C из ассемблера [25]. В этой книге используется соглашение cdecl, поскольку оно используется GCC. Соглашение cdecl гласит, что аргументы функции должны передаваться через стек (на x86). Аргументы функции должны быть помещены в стек в порядке справа налево, то есть сначала помещается самый правый аргумент. Возвращаемое значение функции помещается в регистр eax. Следующий код показывает пример:

```
/* Функция на C */
int sum_of_three(int arg1, int arg2, int arg3)
{
    return arg1 + arg2 + arg3;
}
```

```
; Ассемблерный код
extern sum_of_three   ; функция sum_of_three определена где-то еще

push dword 3            ; arg3
push dword 2            ; arg2
push dword 1            ; arg1
call sum_of_three       ; вызываем функцию, результат будет в eax
```

3.2.1 Упакованные структуры

В остальной части этой книги вы часто будете встречать "байты конфигурации", которые представляют собой набор битов в очень специфическом порядке. Ниже приведен пример с 32 битами:

```
Бит:     | 31     24 | 23          8 | 7     0 |
Содержимое: | index     | address       | config  |
```

Вместо использования беззнакового целого, unsigned int, для работы с такими конфигурациями гораздо удобнее использовать "упакованные структуры":

```
struct example {
    unsigned char config;   /* биты 0 - 7   */
    unsigned short address; /* биты 8 - 23  */
    unsigned char index;    /* биты 24 - 31 */
};
```

При использовании структуры в предыдущем примере нет гарантии, что размер структуры будет ровно 32 бита - компилятор может добавить некоторое заполнение между элементами по разным причинам, например, для ускорения доступа к элементам или из-за требований оборудования и/или компилятора. Когда структура используется для представления конфигурационных байтов, очень важно, чтобы компилятор не добавлял никакого

заполнения, поскольку структура в конечном итоге будет рассматриваться оборудованием как 32-битное беззнаковое целое. Атрибут packed может быть использован, чтобы заставить GCC не добавлять заполнение:

```
struct example {
    unsigned char config;   /* биты 0 - 7   */
    unsigned short address; /* биты 8 - 23  */
    unsigned char index;    /* биты 24 - 31 */
} __attribute__((packed));
```

Обратите внимание, что **attribute**((packed)) не является частью стандарта C - он может не работать со всеми компиляторами C.

3.3 Компиляция кода на C

При компиляции кода на C для ОС необходимо использовать множество флагов GCC. Это связано с тем, что код на C не должен предполагать наличие стандартной библиотеки, так как для нашей ОС стандартная библиотека недоступна. Для получения дополнительной информации о флагах см. руководство GCC.

Флаги, используемые для компиляции кода на C:

```
-m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles -nodefaultlibs
```

Как всегда при написании программ на C мы рекомендуем включить все предупреждения и рассматривать предупреждения как ошибки:

```
-Wall -Wextra -Werror
```

Теперь вы можете создать функцию kmain в файле kmain.c, которую вы вызываете из loader.s. На этом этапе kmain, вероятно, не потребуется никаких аргументов (но в последующих главах это изменится).

3.4 Инструменты сборки

Сейчас также хорошее время для настройки некоторых инструментов сборки, чтобы упростить компиляцию и тестовый запуск ОС. Мы рекомендуем использовать make [13], но доступно множество других систем сборки. Простой Makefile для ОС может выглядеть следующим образом:

```
OBJECTS = loader.o kmain.o
CC = gcc
CFLAGS = -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector \
         -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c
LDFLAGS = -T link.ld -melf_i386
AS = nasm
ASFLAGS = -f elf

all: kernel.elf

kernel.elf: $(OBJECTS)
    ld $(LDFLAGS) $(OBJECTS) -o kernel.elf

os.iso: kernel.elf
    cp kernel.elf iso/boot/kernel.elf
    genisoimage -R                              \
                -b boot/grub/stage2_eltorito    \
                -no-emul-boot                   \
                -boot-load-size 4               \
                -A os                           \
                -input-charset utf8             \
                -quiet                          \
                -boot-info-table                \
                -o os.iso                       \
                iso

run: os.iso
    bochs -f bochsrc.txt -q

%.o: %.c
    $(CC) $(CFLAGS)  $< -o $@

%.o: %.s
    $(AS) $(ASFLAGS) $< -o $@

clean:
    rm -rf *.o kernel.elf os.iso
```

Содержимое вашего рабочего каталога теперь должно выглядеть следующим образом:

```
.
|-- bochsrc.txt
|-- iso
|   |-- boot
|     |-- grub
|       |-- menu.lst
|       |-- stage2_eltorito
|-- kmain.c
|-- loader.s
|-- Makefile
```

Теперь вы должны иметь возможность запустить ОС простой командой make run, которая скомпилирует ядро и загрузит его в Bochs (как определено в Makefile выше).

3.5 Дополнительное чтение

Книга Кернигана и Ричи "Язык программирования C, второе издание" [8] отлично подходит для изучения всех аспектов C.

4. Вывод данных

Эта глава расскажет, как выводить текст на консоль, а также записывать данные в последовательный порт. Кроме того, мы создадим наш первый драйвер, то есть код, который действует как прослойка между ядром и оборудованием, предоставляя более высокий уровень абстракции, чем прямое взаимодействие с оборудованием. Первая часть этой главы посвящена созданию драйвера для фреймбуфера [26], чтобы иметь возможность выводить текст на консоль. Вторая часть показывает, как создать драйвер для последовательного порта. Bochs может сохранять вывод из последовательного порта в файл, что эффективно создает механизм логирования для операционной системы.

4.1 Взаимодействие с оборудованием

Обычно существует два разных способа взаимодействия с оборудованием: memory-mapped I/O и I/O порты.

Если оборудование использует memory-mapped I/O, то вы можете записать в определенный адрес памяти, и оборудование обновится новыми данными. Примером этого является фреймбуфер, который будет подробно обсуждаться позже. Например, если вы запишете значение 0x410F по адресу 0x000B8000, вы увидите букву A белого цвета на черном фоне (подробнее см. раздел о фреймбуфере).

Если оборудование использует I/O порты, то для взаимодействия с оборудованием необходимо использовать ассемблерные инструкции out и in. Инструкция out принимает два параметра: адрес I/O порта и данные для отправки. Инструкция in принимает один параметр, адрес I/O порта, и возвращает данные от оборудования. Можно думать о I/O портах как о взаимодействии с оборудованием так же, как вы взаимодействуете с сервером с помощью сокетов. Курсор (мигающий прямоугольник) фреймбуфера - это пример оборудования, управляемого через I/O порты на PC.

4.2 Фреймбуфер

Фреймбуфер - это аппаратное устройство, способное отображать буфер памяти на экране [26]. Фреймбуфер имеет 80 столбцов и 25 строк, причем индексы строк и столбцов начинаются с 0 (поэтому строки помечены как 0 - 24).

4.2.1 Вывод текста

Вывод текста на консоль через фреймбуфер осуществляется с помощью memory-mapped I/O. Начальный адрес memory-mapped I/O для фреймбуфера - 0x000B8000 [27]. Память разделена на 16-битные ячейки, где 16 бит определяют как символ, так и цвет переднего плана и фона. Старшие восемь бит - это ASCII [28] значение символа, биты 7 - 4 - фон, а биты 3 - 0 - передний план, как показано на следующем рисунке:

```
Бит:     | 15 14 13 12 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
Содержимое: | ASCII                | FG      | BG      |
```

Доступные цвета показаны в следующей таблице:

Цвет	Значение	Цвет	Значение	Цвет	Значение	Цвет	Значение
Черный	0	Красный	4	Темно-серый	8	Светло-красный	12
Синий	1	Пурпурный	5	Светло-синий	9	Светло-пурпурный	13
Зеленый	2	Коричневый	6	Светло-зеленый	10	Светло-коричневый	14
Голубой	3	Светло-серый	7	Светло-голубой	11	Белый	15

Первая ячейка соответствует строке нуль, столбец нуль на консоли. Используя таблицу ASCII, можно увидеть, что A соответствует 65 или 0x41. Поэтому, чтобы записать символ A с зеленым передним планом (2) и темно-серым фоном (8) в позиции (0,0), используется следующая ассемблерная инструкция:

```
mov [0x000B8000], 0x4128
```

Вторая ячейка соответствует строке нуль, столбец один, и ее адрес:

```
0x000B8000 + 16 = 0x000B8010
```

Запись во фреймбуфер также может быть выполнена на C, рассматривая адрес 0x000B8000 как указатель на char, char *fb = (char *) 0x000B8000. Тогда запись A в позиции (0,0) с зеленым передним планом и темно-серым фоном становится:

```
fb[0] = 'A';
fb[1] = 0x28;
```

Следующий код показывает, как это можно обернуть в функцию:

```
/** fb_write_cell:
     *  Записывает символ с заданными цветами переднего плана и фона в позицию i
     *  во фреймбуфере.
     *
     *  @param i  Позиция во фреймбуфере
     *  @param c  Символ
     *  @param fg Цвет переднего плана
     *  @param bg Цвет фона
     */
void fb_write_cell(unsigned int i, char c, unsigned char fg, unsigned char bg)
{
    fb[i] = c;
    fb[i + 1] = ((fg & 0x0F) << 4) | (bg & 0x0F)
}
```

Функция может быть использована следующим образом:

```
 #define FB_GREEN     2
#define FB_DARK_GREY 8

fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
```

4.2.2 Перемещение курсора

Перемещение курсора фреймбуфера осуществляется через два разных I/O порта. Позиция курсора определяется 16-битным целым: 0 означает строку нуль, столбец нуль; 1 означает строку нуль, столбец один; 80 означает строку один, столбец нуль и так далее. Поскольку позиция 16-битная, а инструкция out может отправлять только 8 бит за раз, позиция должна отправляться в два приема: сначала старшие 8 бит, затем младшие 8 бит. Фреймбуфер имеет два I/O порта, один для принятия данных, и один для описания принимаемых данных. Порт 0x3D4 [29] - это порт, который описывает данные, а порт 0x3D5 [29] - для самих данных.

Чтобы установить курсор в строку один, столбец нуль (позиция 80 = 0x0050), используются следующие ассемблерные инструкции:

```
 out 0x3D4, 14      ; 14 говорит фреймбуферу ожидать старшие 8 бит позиции
out 0x3D5, 0x00    ; отправляем старшие 8 бит 0x0050
out 0x3D4, 15      ; 15 говорит фреймбуферу ожидать младшие 8 бит позиции
out 0x3D5, 0x50    ; отправляем младшие 8 бит 0x0050
```

Ассемблерная инструкция out не может быть выполнена непосредственно в C. Поэтому хорошей идеей будет обернуть out в функцию на ассемблере, к которой можно получить доступ из C через соглашение о вызовах cdecl [25]:

```
 global outb             ; делаем метку outb видимой вне этого файла

; outb - отправляет байт в I/O порт
; стек: [esp + 8] байт данных
;        [esp + 4] адрес I/O порта
;        [esp    ] адрес возврата
outb:
    mov al, [esp + 8]    ; перемещаем данные для отправки в регистр al
    mov dx, [esp + 4]    ; перемещаем адрес I/O порта в регистр dx
    out dx, al           ; отправляем данные в I/O порт
    ret                  ; возвращаемся к вызывающей функции
```

Сохранив эту функцию в файле io.s и создав заголовочный файл io.h, можно удобно обращаться к ассемблерной инструкции out из C:

```
#ifndef INCLUDE_IO_H
#define INCLUDE_IO_H

/** outb:
     *  Отправляет данные в указанный I/O порт. Определена в io.s
     *
     *  @param port Адрес I/O порта
     *  @param data Данные для отправки
     */
void outb(unsigned short port, unsigned char data);

#endif /* INCLUDE_IO_H */
```

Перемещение курсора теперь можно обернуть в функцию на C:

```
#include "io.h"

/* I/O порты */
#define FB_COMMAND_PORT         0x3D4
#define FB_DATA_PORT            0x3D5

/* Команды I/O портов */
#define FB_HIGH_BYTE_COMMAND    14
#define FB_LOW_BYTE_COMMAND     15

/** fb_move_cursor:
     *  Перемещает курсор фреймбуфера в указанную позицию
     *
     *  @param pos Новая позиция курсора
     */
void fb_move_cursor(unsigned short pos)
{
    outb(FB_COMMAND_PORT, FB_HIGH_BYTE_COMMAND);
    outb(FB_DATA_PORT,    ((pos >> 8) & 0x00FF));
    outb(FB_COMMAND_PORT, FB_LOW_BYTE_COMMAND);
    outb(FB_DATA_PORT,    pos & 0x00FF);
}
```

4.2.3 Драйвер

Драйвер должен предоставлять интерфейс, который остальной код в ОС будет использовать для взаимодействия с фреймбуфером. Нет правильного или неправильного в том, какую функциональность должен предоставлять интерфейс, но предлагается иметь функцию write со следующей сигнатурой:

```
int write(char *buf, unsigned int len);
```

Функция write записывает содержимое буфера buf длиной len на экран. Функция write должна автоматически продвигать курсор после записи символа и прокручивать экран при необходимости.

4.3 Последовательные порты

Последовательный порт [30] - это интерфейс для обмена данными между аппаратными устройствами. Хотя он доступен почти на всех материнских платах, он редко предоставляется пользователю в виде разъема DE-9 в наши дни. Последовательный порт прост в использовании, и, что более важно, он может быть использован как механизм логирования в Bochs. Если компьютер поддерживает последовательный порт, то обычно поддерживается несколько последовательных портов, но мы будем использовать только один из них. Это связано с тем, что мы будем использовать последовательные порты только для логирования. Более того, мы будем использовать последовательные порты только для вывода, а не для ввода. Последовательные порты полностью управляются через I/O порты.

4.3.1 Конфигурация последовательного порта

Первые данные, которые необходимо отправить на последовательный порт, - это конфигурационные данные. Чтобы два аппаратных устройства могли обмениваться данными, они должны согласовать несколько вещей. К ним относятся:

Скорость передачи данных (битовая скорость или бод)
Нужна ли проверка на ошибки (бит четности, стоповые биты)
Количество бит, представляющих единицу данных (биты данных)

4.3.2 Настройка линии

Настройка линии означает настройку того, как данные передаются по линии. Последовательный порт имеет I/O порт, порт команд линии, который используется для конфигурации.

Сначала будет установлена скорость передачи данных. Последовательный порт имеет внутренние часы, работающие на частоте 115200 Гц. Установка скорости означает отправку делителя на последовательный порт, например, отправка 2 дает скорость 115200 / 2 = 57600 Гц.

Делитель - это 16-битное число, но мы можем отправлять только 8 бит за раз. Поэтому необходимо отправить инструкцию, сообщающую последовательному порту сначала ожидать старшие 8 бит, затем младшие 8 бит. Это делается отправкой 0x80 на порт команд линии. Пример показан ниже:

```
#include "io.h" /* io.h реализован в разделе "Перемещение курсора" */

/* I/O порты */

/* Все I/O порты вычисляются относительно порта данных. Это потому,
     * что все последовательные порты (COM1, COM2, COM3, COM4) имеют свои порты
     * в одном порядке, но начинаются с разных значений.
     */

#define SERIAL_COM1_BASE                0x3F8      /* Базовый порт COM1 */

#define SERIAL_DATA_PORT(base)          (base)
#define SERIAL_FIFO_COMMAND_PORT(base)  (base + 2)
#define SERIAL_LINE_COMMAND_PORT(base)  (base + 3)
#define SERIAL_MODEM_COMMAND_PORT(base) (base + 4)
#define SERIAL_LINE_STATUS_PORT(base)   (base + 5)

/* Команды I/O портов */

/* SERIAL_LINE_ENABLE_DLAB:
     * Сообщает последовательному порту ожидать сначала старшие 8 бит на порту данных,
     * затем младшие 8 бит
     */
#define SERIAL_LINE_ENABLE_DLAB         0x80

/** serial_configure_baud_rate:
     *  Устанавливает скорость передачи данных. Скорость по умолчанию для последовательного
     *  порта 115200 бит/с. Аргумент - делитель этого числа, поэтому
     *  результирующая скорость становится (115200 / делитель) бит/с.
     *
     *  @param com      COM порт для настройки
     *  @param divisor  Делитель
     */
void serial_configure_baud_rate(unsigned short com, unsigned short divisor)
{
    outb(SERIAL_LINE_COMMAND_PORT(com),
         SERIAL_LINE_ENABLE_DLAB);
    outb(SERIAL_DATA_PORT(com),
         (divisor >> 8) & 0x00FF);
    outb(SERIAL_DATA_PORT(com),
         divisor & 0x00FF);
}
```

Способ передачи данных также должен быть настроен. Это также делается через порт команд линии отправкой байта. Разметка 8 бит выглядит следующим образом:

```
Бит:         | 7 | 6 | 5 4 3 | 2 | 1 0 |
Содержание: | d | b | prty  | s | dl  |
```

Описание каждого имени можно найти в таблице ниже (и в [31]):

Имя	Описание
d	Включает (d = 1) или отключает (d = 0) DLAB
b	Если включено управление break (b = 1) или отключено (b = 0)
prty	Количество бит четности для использования

Имя	Описание
s	Количество стоповых бит для использования (s = 0 равно 1, s = 1 равно 1.5 или 2)
dl	Описывает длину данных

Мы будем использовать стандартное значение 0x03 [31], означающее длину 8 бит, без бита четности, один стоповый бит и отключенное управление break. Это отправляется на порт команд линии, как показано в следующем примере:

```
/** serial_configure_line:
     *  Настраивает линию заданного последовательного порта. Порт настраивается на
     *  длину данных 8 бит, без битов четности, один стоповый бит и отключенное
     *  управление break.
     *
     *  @param com  COM порт для настройки
     */
void serial_configure_line(unsigned short com)
{
    /* Бит:     | 7 | 6 | 5 4 3 | 2 | 1 0 |
         * Содержимое: | d | b | prty  | s | dl  |
         * Значение:   | 0 | 0 | 0 0 0 | 0 | 1 1 | = 0x03
         */
    outb(SERIAL_LINE_COMMAND_PORT(com), 0x03);
}
```

Статья на OSDev [31] содержит более глубокое объяснение значений.

4.3.3 Настройка буферов

Когда данные передаются через последовательный порт, они помещаются в буферы, как при приеме, так и при отправке. Таким образом, если вы отправляете данные на последовательный порт быстрее, чем он может передать их по проводу, данные будут буферизованы. Однако, если вы отправите слишком много данных слишком быстро, буфер заполнится и данные будут потеряны. Другими словами, буферы - это очереди FIFO. Конфигурационный байт очереди FIFO выглядит следующим образом:

```
Бит:        | 7 6 | 5  | 4 | 3   | 2   | 1   | 0 |
Содержимое: | lvl | bs | r | dma | clt | clr | e |
```

Описание каждого имени можно найти в таблице ниже:

Имя	Описание
lvl	Сколько байт должно храниться в буферах FIFO
bs	Должны ли буферы быть 16 или 64 байт
r	Зарезервировано для будущего использования
dma	Как следует получать доступ к данным последовательного порта
clt	Очистить буфер передачи FIFO
clr	Очистить буфер приема FIFO
e	Включить или отключить буфер FIFO

Мы используем значение 0xC7 = 11000111, которое:

Включает FIFO
Очищает обе очереди FIFO (прием и передачу)
Использует 14 байт в качестве размера очереди

WikiBook по последовательному программированию [32] объясняет значения более подробно.

4.3.4 Настройка модема

Регистр управления модемом используется для очень простого аппаратного управления потоком через выводы Ready To Transmit (RTS) и Data Terminal Ready (DTR). При настройке последовательного порта мы хотим, чтобы RTS и DTR были равны 1, что означает, что мы готовы к отправке данных.

Конфигурационный байт модема показан на следующем рисунке:

```
Бит:        | 7 | 6 | 5  | 4  | 3   | 2   | 1   | 0   |
Содержимое: | r | r | af | lb | ao2 | ao1 | rts | dtr |
```

Описание каждого имени можно найти в таблице ниже:

Имя	Описание
r	Зарезервировано
af	Автоуправление потоком включено
lb	Режим loopback (используется для отладки последовательных портов)
ao2	Вспомогательный выход 2, используется для получения прерываний
ao1	Вспомогательный выход 1
rts	Ready To Transmit
dtr	Data Terminal Ready

Нам не нужно включать прерывания, так как мы не будем обрабатывать принимаемые данные. Поэтому мы используем конфигурационное значение 0x03 = 00000011 (RTS = 1 и DTS = 1).

4.3.5 Запись данных в последовательный порт

Запись данных в последовательный порт осуществляется через порт данных I/O. Однако перед записью очередь передачи FIFO должна быть пуста (все предыдущие записи должны быть завершены). Очередь передачи FIFO пуста, если бит 5 порта состояния линии I/O равен единице.

Чтение содержимого I/O порта осуществляется с помощью ассемблерной инструкции in. Невозможно использовать ассемблерную инструкцию in из C, поэтому ее необходимо обернуть (так же, как ассемблерную инструкцию out):

```
global inb

; inb - возвращает байт из заданного I/O порта
; стек: [esp + 4] адрес I/O порта
;        [esp    ] адрес возврата
inb:
    mov dx, [esp + 4]       ; перемещаем адрес I/O порта в регистр dx
    in  al, dx              ; читаем байт из I/O порта и сохраняем в регистре al
    ret                     ; возвращаем прочитанный байт
```

```
/* в файле io.h */

/** inb:
     *  Читает байт из I/O порта.
     *
     *  @param  port Адрес I/O порта
     *  @return      Прочитанный байт
     */
unsigned char inb(unsigned short port);
```

Проверить, пуста ли очередь передачи FIFO, можно из C:

```
#include "io.h"

/** serial_is_transmit_fifo_empty:
     *  Проверяет, пуста ли очередь передачи FIFO для заданного COM
     *  порта.
     *
     *  @param  com COM порт
     *  @return 0 если очередь передачи FIFO не пуста
     *          1 если очередь передачи FIFO пуста
     */
int serial_is_transmit_fifo_empty(unsigned int com)
{
    /* 0x20 = 0010 0000 */
    return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x20;
}
```

Запись в последовательный порт означает ожидание, пока очередь передачи FIFO не станет пустой, а затем запись данных в порт данных I/O.

4.3.6 Настройка Bochs

Для сохранения вывода из первого последовательного порта конфигурационный файл Bochs bochsrc.txt должен быть обновлен. Конфигурация com1 указывает Bochs, как обрабатывать первый последовательный порт:

```
com1: enabled=1, mode=file, dev=com1.out
```

Вывод из последовательного порта один теперь будет сохранен в файле com1.out.

4.3.7 Драйвер

Мы рекомендуем реализовать функцию write для последовательного порта, аналогичную функции write в драйвере фреймбуфера. Чтобы избежать конфликтов имен, хорошей идеей будет назвать функции fb_write и serial_write, чтобы отличать их.

Мы также рекомендуем попробовать написать функцию, подобную printf, см. раздел 7.3 в [8]. Функция printf может принимать дополнительный аргумент для выбора устройства вывода (фреймбуфер или последовательный порт).

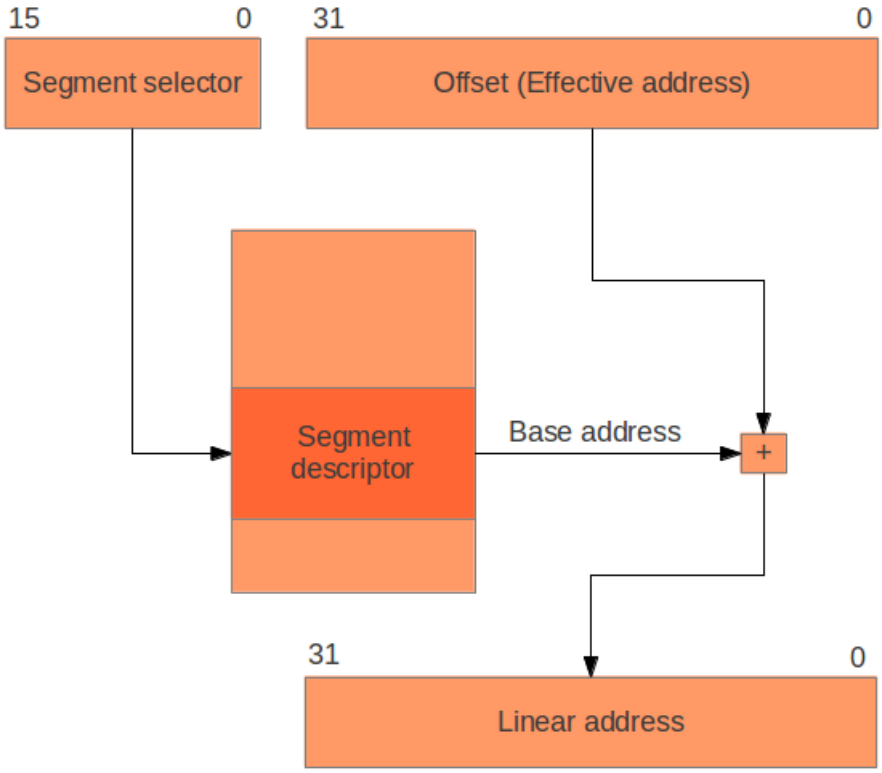
Последняя рекомендация - создать способ различения серьезности сообщений журнала, например, добавляя к сообщениям префиксы DEBUG, INFO или ERROR.

4.4 Дополнительное чтение

Книга "Serial programming" (доступна на WikiBooks) содержит отличный раздел о программировании последовательного порта, http://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming#UART_Registers
Вики OSDev содержит страницу с большим количеством информации о последовательных портах, http://wiki.osdev.org/Serial_ports

5. Сегментация

Сегментация в x86 означает доступ к памяти через сегменты. Сегменты - это части адресного пространства, возможно перекрывающиеся, заданные базовым адресом и пределом. Для адресации байта в сегментированной памяти используется 48-битный логический адрес: 16 бит, которые указывают сегмент, и 32 бита, которые указывают смещение внутри этого сегмента. Смещение добавляется к базовому адресу сегмента, и полученный линейный адрес проверяется относительно предела сегмента - см. рисунок ниже. Если все работает нормально (включая проверки прав доступа, которые мы пока игнорируем), результатом является линейный адрес. Когда страничная организация отключена, линейное адресное пространство отображается 1:1 на физическое адресное пространство, и можно получить доступ к физической памяти. (См. главу "Страничная организация" для информации о том, как включить страничную организацию.)



Преобразование логических адресов в линейные адреса.
Чтобы включить сегментацию, необходимо настроить таблицу, описывающую каждый сегмент — таблицу дескрипторов сегментов.

В архитектуре x86 существует два типа таблиц дескрипторов:

```
Глобальная таблица дескрипторов (GDT)

Локальные таблицы дескрипторов (LDT)
```

LDT создаётся и управляется процессами пользовательского пространства, причём у каждого процесса может быть своя собственная LDT. Локальные таблицы дескрипторов полезны, если требуется более сложная модель сегментации — но в нашем случае они не понадобятся.

GDT является общей для всех (она глобальная), и именно её мы будем использовать.

Как обсуждается в разделах, посвящённых виртуальной памяти и страничной организации, сегментация редко применяется в более сложных конфигурациях, чем минимальная настройка, подобная той, что описана ниже.

5.1 Доступ к памяти

Большую часть времени при доступе к памяти нет необходимости явно указывать сегмент для использования. Процессор имеет шесть 16-битных сегментных регистров: cs, ss, ds, es, gs и fs. Регистр cs - это регистр сегмента кода и указывает сегмент для использования при выборке инструкций. Регистр ss используется при доступе к стеку (через указатель стека esp), а ds - для других доступов к данным. ОС может свободно использовать регистры es, gs и fs как угодно.

Ниже приведен пример неявного использования сегментных регистров:

```
func:
    mov eax, [esp+4]
    mov ebx, [eax]
    add ebx, 8
    mov [eax], ebx
    ret
```

Приведенный выше пример можно сравнить со следующим, где явно используются сегментные регистры:

```
func:
    mov eax, [ss:esp+4]
    mov ebx, [ds:eax]
    add ebx, 8
    mov [ds:eax], ebx
    ret
```

Вам не обязательно использовать ss для хранения селектора сегмента стека или ds для селектора сегмента данных. Вы можете хранить селектор сегмента стека в ds и наоборот. Однако, чтобы использовать неявный стиль, показанный выше, вы должны хранить селекторы сегментов в предназначенных для них регистрах.

Дескрипторы сегментов и их поля описаны на рисунке 3-8 в руководстве Intel [33].

5.2 Глобальная таблица дескрипторов (GDT)

GDT/LDT - это массив 8-байтных дескрипторов сегментов. Первый дескриптор в GDT всегда является нулевым дескриптором и никогда не может быть использован для доступа к памяти. Как минимум, необходимы два дескриптора сегментов (плюс нулевой дескриптор), поскольку дескриптор содержит больше информации, чем просто базовый адрес и предел. Два наиболее важных для нас поля - это поле Type и поле Descriptor Privilege Level (DPL).

Таблица 3-1 в главе 3 руководства Intel [33] определяет значения для поля Type. Таблица показывает, что поле Type не может быть одновременно доступным для записи и исполняемым. Поэтому необходимы два сегмента: один сегмент для исполнения кода, который помещается в cs (Type равен Execute-only или Execute-Read), и один сегмент для чтения и записи данных (Type равен Read/Write), который помещается в другие сегментные регистры.

DPL определяет уровни привилегий, необходимые для использования сегмента. x86 позволяет использовать четыре уровня привилегий (PL), от 0 до 3, где PL0 является наиболее привилегированным. В большинстве операционных систем (например, Linux и Windows) используются только PL0 и PL3. Однако некоторые операционные системы, такие как MINIX, используют все уровни. Ядро должно иметь возможность делать все, поэтому оно использует сегменты с DPL, установленным в 0 (также называемый режимом ядра). Текущий уровень привилегий (CPL) определяется селектором сегмента в cs.

Необходимые сегменты описаны в таблице ниже.

Индекс	Смещение	Имя	Диапазон адресов	Тип	DPL
0	0x00	нулевой дескриптор			
1	0x08	сегмент кода ядра	0x00000000 - 0xFFFFFFFF	RX	PL0
2	0x10	сегмент данных ядра	0x00000000 - 0xFFFFFFFF	RW	PL0

Обратите внимание, что сегменты перекрываются - они охватывают все линейное адресное пространство. В нашей минимальной настройке мы будем использовать сегментацию только для получения уровней привилегий. См. руководство Intel [33], главу 3, для подробностей о других полях дескриптора.

5.3 Загрузка GDT

Загрузка GDT в процессор осуществляется с помощью ассемблерной инструкции lgdt, которая принимает адрес структуры, указывающей начало и размер GDT. Проще всего закодировать эту информацию с помощью "упакованной структуры", как показано в следующем примере:

```
struct gdt {
    unsigned int address;
    unsigned short size;
} __attribute__((packed));
```

Если содержимое регистра eax - это адрес такой структуры, то GDT можно загрузить с помощью следующего ассемблерного кода:

```
lgdt [eax]
```

Может быть проще сделать эту инструкцию доступной из C, так же, как это было сделано с ассемблерными инструкциями in и out.

После загрузки GDT сегментные регистры должны быть загружены их соответствующими селекторами сегментов. Содержимое селектора сегмента описано на рисунке и в таблице ниже:

```
Бит:        | 15                            3 | 2  | 1 0 |
Содержимое: | offset (index)                  | ti | rpl |
```

Разметка селекторов сегментов.

Имя	Описание
rpl	Requested Privilege Level - мы хотим исполнять код в PL0 пока.
ti	Table Indicator. 0 означает, что это указывает на сегмент GDT, 1 означает сегмент LDT.

Имя	Описание
offset (index)	Смещение внутри таблицы дескрипторов.

Смещение селектора сегмента добавляется к началу GDT для получения адреса дескриптора сегмента: 0x08 для первого дескриптора и 0x10 для второго, так как каждый дескриптор занимает 8 байт. Requested Privilege Level (RPL) должен быть 0, так как ядро ОС должно исполняться на уровне привилегий 0.

Загрузка селекторов сегментов для регистров данных проста - просто скопируйте правильные смещения в регистры:

```
mov ds, 0x10
mov ss, 0x10
mov es, 0x10
.
.
.
```

Чтобы загрузить cs, мы должны выполнить "далекий переход":

```
; код здесь использует предыдущий cs
jmp 0x08:flush_cs   ; указываем cs при переходе к flush_cs

flush_cs:
    ; теперь мы изменили cs на 0x08
```

Далекий переход - это переход, где мы явно указываем полный 48-битный логический адрес: селектор сегмента для использования и абсолютный адрес для перехода. Он сначала установит cs в 0x08, а затем перейдет к flush_cs, используя его абсолютный адрес.

5.4 Дополнительное чтение

Глава 3 руководства Intel [33] заполнена низкоуровневыми и техническими деталями о сегментации.
Вики OSDev имеет страницу о сегментации: http://wiki.osdev.org/Segmentation
Страница Wikipedia о сегментации x86 может быть полезна: http://en.wikipedia.org/wiki/X86_memory_segmentation

6. Прерывания и ввод

Теперь, когда ОС может выводить данные, было бы неплохо, если бы она также могла получать ввод. (Операционная система должна уметь обрабатывать прерывания, чтобы читать информацию с клавиатуры). Прерывание происходит, когда аппаратное устройство, такое как клавиатура, последовательный порт или таймер, сигнализирует CPU, что состояние устройства изменилось. Сам CPU также может генерировать прерывания из-за программных ошибок, например, когда программа обращается к памяти, к которой у нее нет доступа, или когда программа делит число на ноль. Наконец, существуют также программные прерывания, которые вызываются ассемблерной инструкцией int и часто используются для системных вызовов.

6.1 Обработчики прерываний

Прерывания обрабатываются через таблицу дескрипторов прерываний (IDT). IDT описывает обработчик для каждого прерывания. Прерывания пронумерованы (0 - 255), и обработчик для прерывания i определен в i-й позиции таблицы. Существует три разных типа обработчиков для прерываний:

*Обработчик задач
*Обработчик прерываний
*Обработчик ловушек

Обработчики задач используют функциональность, специфичную для версии x86 от Intel, поэтому они не будут рассматриваться здесь (см. руководство Intel [33], главу 6, для получения дополнительной информации). Единственное различие между обработчиком прерываний и обработчиком ловушек заключается в том, что обработчик прерываний отключает прерывания, что означает, что вы не можете получить прерывание во время обработки другого прерывания. В этой книге мы будем использовать обработчики ловушек и отключать прерывания вручную, когда это необходимо.

6.2 Создание записи в IDT

Запись в IDT для обработчика прерываний состоит из 64 бит. Старшие 32 бита показаны на рисунке ниже:

```
Бит:        | 31              16 | 15 | 14 13 | 12 | 11 | 10 9 8 | 7 6 5 | 4 3 2 1 0 |
Содержимое: | offset high        | P  | DPL   | 0  | D  | 1  1 0 | 0 0 0 | reserved  |
```

Младшие 32 бита представлены на следующем рисунке:

```
Бит:        | 31              16 | 15              0 |
Содержимое: | segment selector   | offset low        |
```

Описание каждого имени можно найти в таблице ниже:

Имя	Описание

Имя	Описание
offset high	Старшие 16 бит 32-битного адреса в сегменте.
offset low	Младшие 16 бит 32-битного адреса в сегменте.
p	Находится ли обработчик в памяти или нет (1 = присутствует, 0 = отсутствует).
DPL	Descriptor Privilege Level, уровень привилегий, с которого можно вызвать этот обработчик (0, 1, 2, 3).
D	Размер шлюза (1 = 32 бита, 0 = 16 бит).
segment selector	Смещение в GDT.
r	Зарезервировано.

Смещение - это указатель на код (предпочтительно на метку ассемблера). Например, чтобы создать запись для обработчика, код которого начинается с 0xDEADBEEF и который выполняется на уровне привилегий 0 (поэтому использует тот же селектор сегмента кода, что и ядро), будут использоваться следующие два байта:

```
0xDEAD8E00
0x0008BEEF
```

Если IDT представлен как массив беззнаковых целых idt[512], то для регистрации приведенного выше примера в качестве обработчика для прерывания 0 (деление на ноль) будет использоваться следующий код:

```
idt[0] = 0xDEAD8E00
idt[1] = 0x0008BEEF
```

Как написано в главе "Переход к C", мы рекомендуем вместо байтов (или беззнаковых целых) использовать упакованные структуры, чтобы сделать код более читаемым.

6.3 Обработка прерывания

Когда происходит прерывание, CPU поместит некоторую информацию о прерывании в стек, затем найдет соответствующий обработчик прерываний в IDT и перейдет к нему. Стек на момент прерывания будет выглядеть следующим образом:

```
[esp + 12] eflags
[esp + 8]  cs
[esp + 4]  eip
[esp]      error code?
```

Причина знака вопроса после error code в том, что не все прерывания создают код ошибки. Конкретные прерывания CPU, которые помещают код ошибки в стек, - это 8, 10, 11, 12, 13, 14 и 17. Код ошибки может быть использован обработчиком прерываний для получения дополнительной информации о том, что произошло. Также обратите внимание, что номер прерывания не помещается в стек. Мы можем определить, какое прерывание произошло, только зная, какой код выполняется - если выполняется обработчик, зарегистрированный для прерывания 17, то произошло прерывание 17.

Как только обработчик прерываний завершит свою работу, он использует инструкцию iret для возврата. Инструкция iret ожидает, что стек будет таким же, как в момент прерывания (см. рисунок выше). Поэтому любые значения, помещенные в стек обработчиком прерываний, должны быть извлечены. Перед возвратом iret восстанавливает eflags, извлекая значение из стека, и затем переходит к cs:eip, как указано значениями на стеке.

Обработчик прерываний должен быть написан на ассемблере, поскольку все регистры, которые использует обработчик прерываний, должны быть сохранены путем помещения их в стек. Это связано с тем, что код, который был прерван, не знает о прерывании и поэтому ожидает, что его регистры останутся неизменными. Написание всей логики обработчика прерываний на ассемблере будет утомительным. Создание обработчика на ассемблере, который сохраняет регистры, вызывает функцию на C, восстанавливает регистры и затем выполняет iret, - это хорошая идея!

Функция на C должна получать состояние регистров, состояние стека и номер прерывания в качестве аргументов. Могут быть использованы следующие определения:

```
struct cpu_state {
    unsigned int eax;
    unsigned int ebx;
    unsigned int ecx;
    .
    .
    .
    unsigned int esp;
} __attribute__((packed));

struct stack_state {
    unsigned int error_code;
    unsigned int eip;
    unsigned int cs;
    unsigned int eflags;
} __attribute__((packed));

void interrupt_handler(struct cpu_state cpu, struct stack_state stack, unsigned int interrupt);
```

6.4 Создание универсального обработчика прерываний

Поскольку CPU не помещает номер прерывания в стек, написать универсальный обработчик прерываний немного сложно. В этом разделе будут использованы макросы для показа того, как это можно сделать. И поскольку не все прерывания создают код ошибки, значение 0 будет добавлено как "код ошибки" для прерываний без кода ошибки. Следующий код показывает пример того, как это можно сделать:

```
 %macro no_error_code_interrupt_handler %1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword 0                     ; помещаем 0 как код ошибки
    push    dword %1                    ; помещаем номер прерывания
    jmp     common_interrupt_handler    ; переходим к общему обработчику
%endmacro

%macro error_code_interrupt_handler %1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword %1                    ; помещаем номер прерывания
    jmp     common_interrupt_handler    ; переходим к общему обработчику
%endmacro

common_interrupt_handler:               ; общие части универсального обработчика прерываний
    ; сохраняем регистры
    push    eax
    push    ebx
    .
    .
    .
    push    ebp

    ; вызываем функцию на C
    call    interrupt_handler

    ; восстанавливаем регистры
    pop     ebp
    .
    .
    .
    pop     ebx
    pop     eax

    ; восстанавливаем esp
    add     esp, 8

    ; возвращаемся к прерванному коду
    iret

no_error_code_interrupt_handler 0       ; создаем обработчик для прерывания 0
no_error_code_interrupt_handler 1       ; создаем обработчик для прерывания 1
.
.
.
error_code_handler              7       ; создаем обработчик для прерывания 7
.
.
.
```

Общий обработчик common_interrupt_handler делает следующее:

1. Помещает регистры в стек.
2. Вызывает функцию interrupt_handler на C.
3. Извлекает регистры из стека.
4. Добавляет 8 к esp (из-за кода ошибки и номера прерывания, помещенных ранее).
5. Выполняет iret для возврата к прерванному коду.

Поскольку макросы объявляют глобальные метки, адреса обработчиков прерываний могут быть доступны из C или ассемблера при создании IDT.

6.5 Загрузка IDT

IDT загружается с помощью ассемблерной инструкции lidt, которая принимает адрес первого элемента таблицы. Проще всего обернуть эту инструкцию и использовать ее из C:

```
global  load_idt

; load_idt - Загружает таблицу дескрипторов прерываний (IDT).
; стек: [esp + 4] адрес первой записи в IDT
;        [esp    ] адрес возврата
load_idt:
    mov     eax, [esp+4]    ; загружаем адрес IDT в регистр eax
    lidt    eax             ; загружаем IDT
    ret                     ; возвращаемся к вызывающей функции
```

6.6 Программируемый контроллер прерываний (PIC)

Чтобы начать использовать аппаратные прерывания, необходимо сначала настроить Programmable Interrupt Controller (PIC). PIC позволяет сопоставлять сигналы от оборудования с прерываниями. Причины для настройки PIC следующие:

1. Переназначение прерываний. PIC использует прерывания 0 - 15 для аппаратных прерываний по умолчанию, что конфликтует с прерываниями CPU. Поэтому прерывания PIC должны быть переназначены на другой интервал.
2. Выбор прерываний для приема. Вероятно, вы не хотите получать прерывания от всех устройств, так как у вас все равно нет кода для обработки этих прерываний.
3. Настройка правильного режима для PIC.

Вначале был только один PIC (PIC 1) и восемь прерываний. По мере добавления большего количества оборудования восьми прерываний стало недостаточно. Выбранное решение заключалось в том, чтобы соединить еще один PIC (PIC 2) с первым PIC (см. прерывание 2 на PIC 1).

Аппаратные прерывания показаны в таблице ниже:

PIC 1	Оборудование	PIC 2	Оборудование
0	Таймер	8	Часы реального времени
1	Клавиатура	9	Общий ввод/вывод
2	PIC 2	10	Общий ввод/вывод
3	COM 2	11	Общий ввод/вывод
4	COM 1	12	Общий ввод/вывод
5	LPT 2	13	Сопроцессор
6	Дискета	14	Шина IDE
7	LPT 1	15	Шина IDE

Отличное руководство по настройке PIC можно найти на сайте SigOPS [35]. Мы не будем повторять эту информацию здесь.

Каждое прерывание от PIC должно быть подтверждено - то есть необходимо отправить сообщение PIC, подтверждающее, что прерывание было обработано. Если этого не сделать, PIC не будет генерировать больше прерываний.

Подтверждение прерывания PIC осуществляется отправкой байта 0x20 в PIC, который вызвал прерывание. Реализация функции pic_acknowledge может быть выполнена следующим образом:

```
#include "io.h"

#define PIC1_PORT_A 0x20
#define PIC2_PORT_A 0xA0

/* Прерывания PIC были переназначены */
#define PIC1_START_INTERRUPT 0x20
#define PIC2_START_INTERRUPT 0x28
#define PIC2_END_INTERRUPT   PIC2_START_INTERRUPT + 7

#define PIC_ACK     0x20

/** pic_acknowledge:
     *  Подтверждает прерывание от PIC 1 или PIC 2.
     *
     *  @param num Номер прерывания
     */
void pic_acknowledge(unsigned integer interrupt)
{
    if (interrupt < PIC1_START_INTERRUPT || interrupt > PIC2_END_INTERRUPT) {
      return;
    }

    if (interrupt < PIC2_START_INTERRUPT) {
      outb(PIC1_PORT_A, PIC_ACK);
    } else {
      outb(PIC2_PORT_A, PIC_ACK);
    }
}
```

6.7 Чтение ввода с клавиатуры

Клавиатура не генерирует символы ASCII, она генерирует скан-коды. Скан-код представляет кнопку - как нажатие, так и отпускание. Скан-код только что нажатой кнопки можно прочитать из порта данных I/O клавиатуры, который имеет адрес 0x60. Как это можно сделать, показано в следующем примере:

```
#include "io.h"

#define KBD_DATA_PORT   0x60

/** read_scan_code:
     *  Читает скан-код с клавиатуры
     *
     *  @return Скан-код (НЕ символ ASCII!)
     */
unsigned char read_scan_code(void)
{
    return inb(KBD_DATA_PORT);
}
```

Следующий шаг - написать функцию, которая преобразует скан-код в соответствующий символ ASCII. Если вы хотите сопоставить скан-коды с символами ASCII так, как это сделано на американской клавиатуре, то у Андриса Брауэра есть отличное руководство [36].

Помните, что поскольку прерывание клавиатуры вызывается PIC, вы должны вызвать pic_acknowledge в конце обработчика прерываний клавиатуры. Кроме того, клавиатура не будет отправлять вам больше прерываний, пока вы не прочитаете скан-код с клавиатуры.

6.8 Дополнительное чтение

Вики OSDev имеет отличную страницу о прерываниях, http://wiki.osdev.org/Interrupts
Глава 6 руководства Intel 3a [33] описывает все, что нужно знать о прерываниях.

7. Путь к пользовательскому режиму

Теперь, когда ядро загружается, выводит данные на экран и читает с клавиатуры - что нам делать дальше? Обычно ядро не должно само выполнять логику приложений, а оставлять это приложениям. Ядро создает правильные абстракции (для памяти, файлов, устройств) для упрощения разработки приложений, выполняет задачи от имени приложений (системные вызовы) и планирует процессы.

Пользовательский режим, в отличие от режима ядра, - это среда, в которой выполняются пользовательские программы. Эта среда менее

привилегирована, чем режим ядра, и предотвращает (плохо написанные) пользовательские программы от вмешательства в другие программы или ядро. Плохо написанные ядра могут делать все, что угодно.

Предстоит еще долгий путь, прежде чем ОС, созданная в этой книге, сможет выполнять программы в пользовательском режиме, но эта глава покажет, как легко выполнить небольшую программу в режиме ядра.

7.1 Загрузка внешней программы

Откуда мы берем внешнюю программу? Нам каким-то образом нужно загрузить код, который мы хотим выполнить, в память. Более полнофункциональные операционные системы обычно имеют драйверы и файловые системы, которые позволяют им загружать программное обеспечение с CD-ROM, жесткого диска или других постоянных носителей.

Вместо создания всех этих драйверов и файловых систем мы будем использовать функцию GRUB под названием модули для загрузки программы.

7.1.1 Модули GRUB

GRUB может загружать произвольные файлы в память из ISO-образа, и эти файлы обычно называются модулями. Чтобы GRUB загрузил модуль, отредактируйте файл iso/boot/grub/menu.lst и добавьте следующую строку в конец файла:

```
module /modules/program
```

Теперь создайте папку iso/modules:

```
mkdir -p iso/modules
```

Программа приложения будет создана позже в этой главе.

Код, который вызывает kmain, должен быть обновлен для передачи информации в kmain о том, где можно найти модули. Мы также хотим сказать GRUB, что он должен выравнивать все модули по границам страниц при их загрузке (см. главу "Страничная организация" для подробностей о выравнивании страниц).

Чтобы указать GRUB, как загружать наши модули, "multiboot заголовок" - первые байты ядра - должен быть обновлен следующим образом:

```
; в файле `loader.s`

MAGIC_NUMBER    equ 0x1BADB002      ; определяем магическое число
ALIGN_MODULES   equ 0x00000001      ; указываем GRUB выравнивать модули

; вычисляем контрольную сумму (все опции + контрольная сумма должны равняться 0)
CHECKSUM        equ -(MAGIC_NUMBER + ALIGN_MODULES)

section .text:                      ; начало секции кода
align 4                             ; код должен быть выровнен по 4 байтам
    dd MAGIC_NUMBER                 ; записываем магическое число
    dd ALIGN_MODULES                ; записываем инструкцию выравнивания модулей
    dd CHECKSUM                     ; записываем контрольную сумму
```

GRUB также сохранит указатель на структуру в регистре ebx, которая, среди прочего, описывает, по каким адресам загружены модули. Поэтому, вероятно, вы захотите поместить ebx в стек перед вызовом kmain, чтобы сделать его аргументом для kmain.

7.2 Выполнение программы

7.2.1 Очень простая программа

Программа, написанная на этом этапе, может выполнять только несколько действий. Поэтому достаточно очень короткой программы, которая записывает значение в регистр. Остановка Bochs через некоторое время и проверка того, что регистр содержит правильное число, путем просмотра журнала Bochs подтвердит, что программа выполнилась. Вот пример такой короткой программы:

```
; устанавливаем eax в легко узнаваемое число, чтобы прочитать его из журнала позже
mov eax, 0xDEADBEEF

; входим в бесконечный цикл, больше нечего делать
; $ означает "начало строки", т.е. ту же инструкцию
jmp $
```

7.2.2 Компиляция

Поскольку наше ядро не может анализировать продвинутые форматы исполняемых файлов, нам нужно скомпилировать код в плоский бинарный файл. NASM может сделать это с флагом -f:

```
nasm -f bin program.s -o program
```

Это все, что нам нужно. Теперь вы должны переместить файл program в папку iso/modules.

7.2.3 Поиск программы в памяти

Прежде чем переходить к программе, мы должны найти, где она находится в памяти. Предполагая, что содержимое ebx передается как аргумент в kmain, мы можем сделать это полностью на C.

Указатель в ebx указывает на структуру multiboot [19]. Загрузите файл multiboot.h с http://www.gnu.org/software/grub/manual/multiboot/html_node/multiboot.h.html, который описывает структуру.

Указатель, переданный в kmain в регистре ebx, может быть приведен к типу multiboot_info_t. Адрес первого модуля находится в поле mods_addr. Следующий код показывает пример:

```
int kmain(/* дополнительные аргументы */ unsigned int ebx)
{
    multiboot_info_t *mbinfo = (multiboot_info_t *) ebx;
    unsigned int address_of_module = mbinfo->mods_addr;
}
```

Однако, прежде чем слепо следовать указателю, вы должны проверить, что модуль был загружен правильно GRUB. Это можно сделать, проверив поле flags структуры multiboot_info_t. Вы также должны проверить поле mods_count, чтобы убедиться, что оно равно ровно 1. Для получения более подробной информации о структуре multiboot см. документацию multiboot [19].

7.2.4 Переход к коду

Единственное, что осталось сделать, - это перейти к коду, загруженному GRUB. Поскольку проще анализировать структуру multiboot на C, чем на ассемблере, вызов кода из C более удобен (это, конечно, также можно сделать с помощью jmp или call на ассемблере). Код на C может выглядеть так:

```
typedef void (*call_module_t)(void);
/* ... */
call_module_t start_program = (call_module_t) address_of_module;
start_program();
/* мы никогда не попадем сюда, если только код модуля не вернется */
```

Если мы запустим ядро, подождем, пока оно выполнится и войдет в бесконечный цикл в программе, а затем остановим Bochs, мы должны увидеть 0xDEADBEEF в регистре eax через журнал Bochs. Мы успешно запустили программу в нашей ОС!

7.3 Начало пользовательского режима

Программа, которую мы написали сейчас, выполняется на том же уровне привилегий, что и ядро, - мы просто вошли в нее несколько необычным способом. Чтобы позволить приложениям выполняться на другом уровне привилегий, нам нужно, помимо сегментации, реализовать страничную организацию и распределение кадров страниц.

Это довольно большой объем работы и технических деталей, но через несколько глав у вас будут работающие программы в пользовательском режиме.

8. Краткое введение в виртуальную память

Виртуальная память - это абстракция физической памяти. Цель виртуальной памяти - обычно упростить разработку приложений и позволить процессам адресовать больше памяти, чем фактически физически присутствует в машине. Мы также не хотим, чтобы приложения вмешивались в память ядра или других приложений из соображений безопасности.

В архитектуре x86 виртуальная память может быть реализована двумя способами: сегментация и страничная организация. Страничная организация является наиболее распространенной и универсальной техникой, и мы реализуем ее в следующей главе. Некоторое использование сегментации все еще необходимо для выполнения кода под разными уровнями привилегий.

Управление памятью - это большая часть того, что делает операционная система. Страничная организация и распределение кадров страниц занимаются этим.

Сегментация и страничная организация описаны в [33], главах 3 и 4.

8.1 Виртуальная память через сегментацию?

Вы могли бы полностью пропустить страничную организацию и просто использовать сегментацию для виртуальной памяти. Каждый процесс пользовательского режима получил бы свой собственный сегмент, с правильно установленными базовым адресом и пределом. Таким образом, ни один процесс не сможет увидеть память другого процесса. Проблема с этим подходом в том, что физическая память для процесса должна быть непрерывной (или, по крайней мере, это очень удобно). Либо нам нужно знать заранее, сколько памяти потребуется программе (маловероятно), либо мы должны перемещать сегменты памяти в места, где они могут расти, когда предел достигнут (дорого, вызывает фрагментацию - может привести к "нехватке памяти", даже если памяти достаточно). Страничная организация решает обе эти проблемы.

Интересно отметить, что в x86_64 (64-битной версии архитектуры x86) сегментация почти полностью удалена.

8.2 Дополнительное чтение

LWN.net имеет статью о виртуальной памяти: http://lwn.net/Articles/253361/
Густаво Дуарте также написал статью о виртуальной памяти: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

9. Страничная организация

Сегментация преобразует логический адрес в линейный адрес. Страничная организация преобразует эти линейные адреса в физическое адресное пространство и определяет права доступа и то, как память должна кэшироваться.

9.1 Зачем нужна страничная организация?

Страничная организация - это наиболее распространенная техника, используемая в x86 для включения виртуальной памяти. Виртуальная память через страничную организацию означает, что каждый процесс получит впечатление, что доступный диапазон памяти - это 0x00000000 - 0xFFFFFFFF, даже если фактический размер памяти может быть намного меньше. Это также означает, что когда процесс обращается к байту памяти, он использует виртуальный (линейный) адрес вместо физического. Код в пользовательском процессе не заметит никакой разницы (кроме задержек выполнения). Линейный адрес преобразуется в физический адрес MMU и таблицей страниц. Если виртуальный адрес не отображен на физический адрес, CPU вызовет прерывание page fault.

Страничная организация необязательна, и некоторые операционные системы не используют ее. Но если мы хотим пометить определенные области памяти доступными только для кода, выполняющегося на определенном уровне привилегий (чтобы иметь процессы, выполняющиеся на разных уровнях привилегий), страничная организация - самый аккуратный способ сделать это.

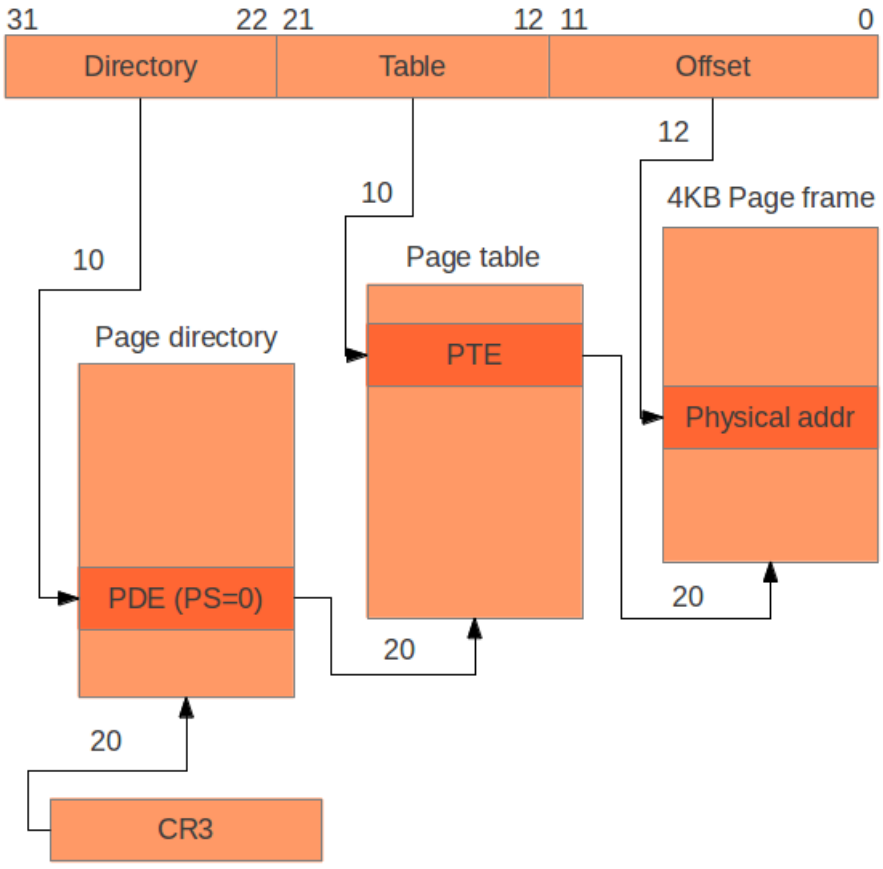
9.2 Страничная организация в x86

Страничная организация в x86 (глава 4 в руководстве Intel [33]) состоит из каталога страниц (PDT), который может содержать ссылки на 1024 таблицы страниц (PT), каждая из которых может указывать на 1024 участка физической памяти, называемых кадрами страниц (PF). Каждый кадр страницы имеет размер 4096 байт. В виртуальном (линейном) адресе старшие 10 бит указывают смещение записи каталога страниц (PDE) в текущем PDT, следующие 10 бит - смещение записи таблицы страниц (PTE) в таблице страниц, на которую указывает этот PDE. Младшие 12 бит в адресе - это смещение внутри кадра страницы, к которому осуществляется обращение.

Все каталоги страниц, таблицы страниц и кадры страниц должны быть выровнены по адресам, кратным 4096 байтам. Это делает возможным адресовать PDT, PT или PF, используя только старшие 20 бит 32-битного адреса, так как младшие 12 бит должны быть нулевыми.

Структуры PDE и PTE очень похожи друг на друга: 32 бита (4 байта), где старшие 20 бит указывают на PTE или PF, а младшие 12 бит управляют правами доступа и другими настройками. 4 байта, умноженные на 1024, равны 4096 байтам, поэтому каталог страниц и таблица страниц помещаются в кадр страницы сами.

Преобразование виртуальных адресов в физические адреса описано на рисунке ниже.



Хотя страницы обычно имеют размер 4096 байт, также возможно использовать страницы размером 4 МБ. PDE тогда указывает непосредственно на кадр страницы 4 МБ, который должен быть выровнен по границе 4 МБ. Преобразование адресов почти такое же, как на рисунке, за исключением того, что шаг таблицы страниц удален. Можно смешивать страницы 4 МБ и 4 КБ.

Преобразование виртуальных адресов (линейных адресов) в физические адреса.

20 бит, указывающих на текущий PDT, хранятся в регистре cr3. Младшие 12 бит cr3 используются для конфигурации.

Для получения более подробной информации о структурах страничной организации см. главу 4 в руководстве Intel [33]. Наиболее интересные биты - это U/S, которые определяют, какие уровни привилегий могут обращаться к этой странице (PL0 или PL3), и R/W, который делает память в странице доступной для чтения-записи или только для чтения.

9.2.1 Тождественное отображение страниц

Самый простой вид страничной организации - когда мы отображаем каждый виртуальный адрес на тот же физический адрес, называемый тождественным отображением страниц. Это можно сделать во время компиляции, создав каталог страниц, где каждая запись указывает на соответствующий кадр 4 МБ. В NASM это можно сделать с помощью макросов и команд (%rep, times и dd). Это, конечно, также можно сделать во время выполнения с помощью обычных ассемблерных инструкций.

9.2.2 Включение страничной организации

Страничная организация включается сначала записью адреса каталога страниц в cr3, а затем установкой бита 31 (бит PG "включение страничной организации") регистра cr0 в 1. Чтобы использовать страницы 4 МБ, установите бит PSE (Page Size Extensions, бит 4) регистра cr4. Следующий ассемблерный код показывает пример:

```
 ; eax содержит адрес каталога страниц
mov cr3, eax

mov ebx, cr4        ; читаем текущий cr4
or  ebx, 0x00000010 ; устанавливаем PSE
mov cr4, ebx        ; обновляем cr4

mov ebx, cr0        ; читаем текущий cr0
or  ebx, 0x80000000 ; устанавливаем PG
mov cr0, ebx        ; обновляем cr0

; теперь страничная организация включена
```

9.2.3 Несколько деталей

Важно отметить, что все адреса внутри каталога страниц, таблиц страниц и в cr3 должны быть физическими адресами структур, а не виртуальными. Это станет более актуальным в следующих разделах, где мы динамически обновляем структуры страничной организации (см. главу "Пользовательский режим").

Инструкция, полезная при обновлении PDT или PT, - это invlpg. Она аннулирует запись в буфере ассоциативной трансляции (TLB) для виртуального адреса. TLB - это кэш для преобразованных адресов, отображающий физические адреса, соответствующие виртуальным адресам. Это требуется только при изменении PDE или PTE, которые ранее были отображены на что-то другое. Если PDE или PTE ранее были помечены как отсутствующие (бит 0 был установлен в 0), выполнение invlpg не требуется. Изменение значения cr3 приведет к аннулированию всех записей в TLB.

Пример аннулирования записи TLB показан ниже:

```
 ; аннулируем любые ссылки TLB на виртуальный адрес 0
invlpg [0]
```

9.3 Страничная организация и ядро

Этот раздел описывает, как страничная организация влияет на ядро ОС. Мы рекомендуем вам запускать вашу ОС, используя тождественное отображение страниц, прежде чем пытаться реализовать более продвинутую настройку страничной организации, так как может быть сложно отлаживать неисправную таблицу страниц, настроенную с помощью ассемблерного кода.

9.3.1 Причины не использовать тождественное отображение для ядра

Если ядро размещено в начале виртуального адресного пространства - то есть виртуальное адресное пространство (0x00000000, "размер ядра") отображается на расположение ядра в памяти - возникнут проблемы при линковке кода процесса пользовательского режима. Обычно при линковке компоновщик предполагает, что код будет загружен в память по адресу 0x00000000. Поэтому при разрешении абсолютных ссылок 0x00000000 будет базовым адресом для вычисления точного положения. Но если ядро отображено на виртуальное адресное пространство (0x00000000, "размер ядра"), процесс пользовательского режима не может быть загружен по виртуальному адресу 0x00000000 - он должен быть размещен где-то еще. Поэтому предположение компоновщика о том, что процесс пользовательского режима загружен в память по адресу 0x00000000, неверно. Это можно исправить, используя скрипт компоновщика, который говорит компоновщику предполагать другой начальный адрес, но это очень громоздкое решение для пользователей операционной системы.

Это также предполагает, что мы хотим, чтобы ядро было частью адресного пространства процесса пользовательского режима. Как мы увидим позже, это удобная функция, так как во время системных вызовов нам не нужно изменять какие-либо структуры страничной организации для получения доступа к коду и данным ядра. Страницы ядра, конечно, потребуют уровня привилегий 0 для доступа, чтобы предотвратить чтение или запись в память ядра пользовательским процессом.

9.3.2 Виртуальный адрес для ядра

Предпочтительно разместить ядро по очень высокому виртуальному адресу памяти, например 0xC0000000 (3 ГБ). Процесс пользовательского режима вряд ли будет размером 3 ГБ, что сейчас является единственным способом, которым он может конфликтовать с ядром. Когда ядро использует

виртуальные адреса начиная с 3 ГБ и выше, это называется higher-half ядром. 0xC0000000 - это всего лишь пример, ядро может быть размещено по любому адресу выше 0, чтобы получить те же преимущества. Выбор правильного адреса зависит от того, сколько виртуальной памяти должно быть доступно для ядра (проще всего, если вся память выше виртуального адреса ядра должна принадлежать ядру) и сколько виртуальной памяти должно быть доступно для процесса.

Если процесс пользовательского режима больше 3 ГБ, некоторые страницы должны быть выгружены ядром. Выгрузка страниц не является частью этой книги.

9.3.3 Размещение ядра по адресу 0xC0000000

Для начала лучше разместить ядро по адресу 0xC0100000, чем 0xC0000000, так как это позволяет отобразить (0x00000000, 0x00100000) на (0xC0000000, 0xC0100000). Таким образом, весь диапазон (0x00000000, "размер ядра") памяти отображается на диапазон (0xC0000000, 0xC0000000 + "размер ядра").

Размещение ядра по адресу 0xC0100000 несложно, но требует некоторых размышлений. Это снова проблема линковки. Когда компоновщик разрешает все абсолютные ссылки в ядре, он предполагает, что наше ядро загружено в физическую память по адресу 0x00100000, а не 0x00000000, так как используется перемещение в скрипте компоновщика (см. раздел "Линковка ядра"). Однако мы хотим, чтобы переходы разрешались с использованием 0xC0100000 в качестве базового адреса, так как иначе прыжок ядра перейдет прямо в код процесса пользовательского режима (помните, что процесс пользовательского режима загружен в виртуальную память 0x00000000).

Однако мы не можем просто сказать компоновщику, что ядро должно начинаться (загружаться) по адресу 0xC01000000, так как мы хотим, чтобы оно загружалось по физическому адресу 0x00100000. Причина загрузки ядра на 1 МБ в том, что оно не может быть загружено по адресу 0x00000000, так как там находятся BIOS и код GRUB. Более того, мы не можем предполагать, что можем загрузить ядро по адресу 0xC0100000, так как машина может не иметь 3 ГБ физической памяти.

Это можно решить, используя как перемещение (.=0xC0100000), так и инструкцию AT в скрипте компоновщика. Перемещение указывает, что ссылки на память должны использовать адрес перемещения как базу при вычислении адресов. AT указывает, куда должно быть загружено ядро в память. Перемещение выполняется во время линковки GNU ld [37], адрес загрузки, указанный AT, обрабатывается GRUB при загрузке ядра и является частью формата ELF [18].

9.3.4 Скрипт линкера для higher-half

Мы можем изменить первый скрипт линкера для реализации этого:

```
ENTRY(loader)           /* имя символа входа */

. = 0xC0100000          /* код должен быть перемещен на 3 ГБ + 1 МБ */

/* выравнивание по 4 КБ и загрузка на 1 МБ */
.text ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.text)            /* все текстовые секции из всех файлов */
}

/* выравнивание по 4 КБ и загрузка на 1 МБ + . */
.rodata ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.rodata*)         /* все секции только для чтения из всех файлов */
}

/* выравнивание по 4 КБ и загрузка на 1 МБ + . */
.data ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.data)            /* все секции данных из всех файлов */
}

/* выравнивание по 4 КБ и загрузка на 1 МБ + . */
.bss ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(COMMON)           /* все COMMON секции из всех файлов */
    *(.bss)             /* все bss секции из всех файлов */
}
```

9.3.5 Вход в higher-half

Когда GRUB переходит к коду ядра, таблицы страниц отсутствуют. Поэтому все ссылки на 0xC0100000 + X не будут отображены на правильный физический адрес и вызовут исключение общей защиты (GPE) в лучшем случае, в противном случае (если компьютер имеет более 3 ГБ памяти) компьютер просто зависнет.

Поэтому необходимо использовать ассемблерный код, который не использует относительные переходы или относительную адресацию памяти, чтобы сделать следующее:

1. Настроить таблицу страниц.
2. Добавить тождественное отображение для первых 4 МБ виртуального адресного пространства.
3. Добавить запись для 0xC0100000, которая отображается на 0x0010000.

Если мы пропустим тождественное отображение для первых 4 МБ, CPU вызовет ошибку страницы сразу после включения страничной организации при попытке получить следующую инструкцию из памяти. После создания таблицы можно выполнить переход к метке, чтобы заставить eip указывать на виртуальный адрес в higher-half:

```
; ассемблерный код, выполняющийся примерно по адресу 0x00100000
; включаем страничную организацию для фактического расположения ядра
; и его виртуального расположения в higher-half

lea ebx, [higher_half] ; загружаем адрес метки в ebx
jmp ebx                ; переходим к метке

higher_half:
    ; код здесь выполняется в higher-half ядре
    ; eip больше 0xC0000000
    ; можно продолжать инициализацию ядра, вызов кода на C и т.д.
```

Регистр eip теперь будет указывать на ячейку памяти где-то сразу после 0xC0100000 - весь код теперь может выполняться так, как если бы он был расположен по адресу 0xC0100000, higher-half. Начальное отображение первых 4 МБ виртуальной памяти на первые 4 МБ физической памяти теперь можно удалить из таблицы страниц, и соответствующую запись в TLB аннулировать с помощью invlpg [0].

9.3.6 Работа в higher-half

Есть несколько дополнительных деталей, с которыми мы должны разобраться при использовании higher-half ядра. Мы должны быть осторожны при использовании memory-mapped I/O, которое использует определенные адреса памяти. Например, фреймбуфер расположен по адресу 0x000B8000, но поскольку в таблице страниц больше нет записи для адреса 0x000B8000, необходимо использовать адрес 0xC00B8000, так как виртуальный адрес 0xC0000000 отображается на физический адрес 0x00000000.

Любые явные ссылки на адреса внутри multiboot структуры также должны быть изменены, чтобы отражать новые виртуальные адреса.

Отображение страниц 4 МБ для ядра просто, но расходует память (если только у вас не очень большое ядро). Создание higher-half ядра, отображенного как страницы 4 КБ, экономит память, но сложнее в настройке. Память для каталога страниц и одной таблицы страниц может быть зарезервирована в секции .data, но необходимо настроить отображения виртуальных адресов в физические во время выполнения. Размер ядра может быть определен путем экспорта меток из скрипта компоновщика [37], что нам все равно понадобится сделать позже при написании распределителя кадров страниц (см. главу "Распределение кадров страниц").

9.4 Виртуальная память через страничную организацию

Страничная организация позволяет сделать две вещи, полезные для виртуальной памяти. Во-первых, она позволяет осуществлять детальный контроль доступа к памяти. Вы можете пометить страницы как только для чтения, для чтения-записи, только для PL0 и т.д. Во-вторых, она создает иллюзию непрерывной памяти. Процессы пользовательского режима и ядро могут обращаться к памяти так, как если бы она была непрерывной, и непрерывная память может быть расширена без необходимости перемещать данные в памяти. Мы также можем разрешить программам пользовательского режима доступ ко всей памяти ниже 3 ГБ, но если они фактически не используют ее, нам не нужно выделять кадры страниц для этих страниц. Это позволяет процессам иметь код, расположенный около 0x00000000, и стек чуть ниже 0xC0000000, и при этом не требовать более двух фактических страниц.

9.5 Дополнительное чтение

Глава 4 (и в некоторой степени глава 3) руководства Intel [33] - ваш окончательный источник подробностей о страничной организации.
Wikipedia имеет статью о страничной организации: http://en.wikipedia.org/wiki/Paging
Вики OSDev имеет страницу о страничной организации: http://wiki.osdev.org/Paging и руководство по созданию higher-half ядра: http://wiki.osdev.org/Higher_Half_bare_bones
Статья Густаво Дуарте о том, как ядро управляет памятью, стоит прочтения: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory
Подробности о языке команд компоновщика можно найти на сайте Стива Чемберлена [37].
Более подробная информация о формате ELF доступна в этой презентации: http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf

10. Распределение кадров страниц

При использовании виртуальной памяти, как ОС узнает, какие части памяти свободны для использования? Это роль распределителя кадров страниц.

10.1 Управление доступной памятью

10.1.1 Сколько памяти доступно?

Сначала нам нужно знать, сколько памяти доступно на компьютере, на котором работает ОС. Самый простой способ сделать это - прочитать это из структуры multiboot [19], переданной нам GRUB. GRUB собирает информацию, которая нам нужна о памяти - что зарезервировано, отображено на ввод-вывод, только для чтения и т.д. Мы также должны убедиться, что мы не отмечаем часть памяти, используемую ядром, как свободную (поскольку GRUB не отмечает эту память как зарезервированную). Один из способов узнать, сколько памяти использует ядро, - это экспортировать метки в начале и конце ядра из скрипта компоновщика:

```
ENTRY(loader)           /* имя символа входа */

. = 0xC0100000          /* код должен быть перемещен на 3 ГБ + 1 МБ */

/* эти метки экспортируются в файлы кода */
kernel_virtual_start = .;
kernel_physical_start = . - 0xC0000000;

/* выравнивание по 4 КБ и загрузка на 1 МБ */
.text ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.text)            /* все текстовые секции из всех файлов */
}

/* выравнивание по 4 КБ и загрузка на 1 МБ + . */
.rodata ALIGN (0x1000) : AT(ADDR(.rodata)-0xC0000000)
{
    *(.rodata*)         /* все секции только для чтения из всех файлов */
}

/* выравнивание по 4 КБ и загрузка на 1 МБ + . */
.data ALIGN (0x1000) : AT(ADDR(.data)-0xC0000000)
{
    *(.data)            /* все секции данных из всех файлов */
}

/* выравнивание по 4 КБ и загрузка на 1 МБ + . */
.bss ALIGN (0x1000) : AT(ADDR(.bss)-0xC0000000)
{
    *(COMMON)           /* все COMMON секции из всех файлов */
    *(.bss)             /* все bss секции из всех файлов */
}

kernel_virtual_end = .;
kernel_physical_end = . - 0xC0000000;
```

Эти метки могут быть непосредственно прочитаны из ассемблерного кода и помещены в стек, чтобы сделать их доступными для кода на C:

```
extern kernel_virtual_start
extern kernel_virtual_end
extern kernel_physical_start
extern kernel_physical_end

; ...

push kernel_physical_end
push kernel_physical_start
push kernel_virtual_end
push kernel_virtual_start

call kmain
```

Таким образом, мы получаем метки как аргументы kmain. Если вы хотите использовать C вместо ассемблерного кода, один из способов сделать это - объявить метки как функции и взять адреса этих функций:

```
void kernel_virtual_start(void);

/* ... */

unsigned int vaddr = (unsigned int) &kernel_virtual_start;
```

Если вы используете модули GRUB, вам нужно убедиться, что память, которую они используют, также помечена как зарезервированная.

Обратите внимание, что доступная память не обязательно должна быть непрерывной. В первых 1 МБ есть несколько секций памяти, отображенных на устройства ввода-вывода, а также память, используемая GRUB и BIOS. Другие части памяти могут быть аналогично недоступны.

Удобно разделить секции памяти на полные кадры страниц, так как мы не можем отобразить часть страниц в память.

10.1.2 Управление доступной памятью

Как мы узнаем, какие кадры страниц используются? Распределитель кадров страниц должен отслеживать, какие кадры свободны, а какие нет. Есть несколько способов сделать это: битовые карты, связные списки, деревья, система Buddy (используемая Linux) и т.д. Для получения дополнительной информации о различных алгоритмах см. статью на OSDev [38].

Битовые карты довольно легко реализовать. Один бит используется для каждого кадра страницы, и один (или более) кадров страниц выделяется для хранения битовой карты. (Обратите внимание, что это всего лишь один из способов сделать это, другие проекты могут быть лучше и/или интереснее для реализации.)

10.2 Как мы можем получить доступ к кадру страницы?

Распределитель кадров страниц возвращает физический начальный адрес кадра страницы. Этот кадр страницы не отображен - ни одна таблица страниц не указывает на этот кадр страницы. Как мы можем читать и записывать данные в кадр?

Нам нужно отобразить кадр страницы в виртуальную память, обновив PDT и/или PT, используемые ядром. Что делать, если все доступные таблицы страниц заполнены? Тогда мы не сможем отобразить кадр страницы в память, потому что нам понадобится новая таблица страниц - которая занимает целый кадр страницы - и чтобы записать в этот кадр страницы, нам нужно отобразить его кадр страницы... Как-то эту циклическую зависимость нужно разорвать.

Одно из решений - зарезервировать часть первой таблицы страниц, используемой ядром (или какой-либо другой higher-half таблицы страниц), для временного отображения кадров страниц, чтобы сделать их доступными. Если ядро отображено по адресу 0xC0000000 (запись в каталоге страниц с индексом 768), и используются кадры страниц 4 КБ, то ядро имеет по крайней мере одну таблицу страниц. Если мы предположим (или ограничимся) ядром размером не более 4 МБ минус 4 КБ, мы можем выделить последнюю запись (запись 1023) этой таблицы страниц для временных отображений. Виртуальный адрес страниц, отображенных с использованием последней записи PT ядра, будет:

```
(768 << 22) | (1023 << 12) | 0x000 = 0xC03FF000
```

После того как мы временно отобразили кадр страницы, который мы хотим использовать как таблицу страниц, и настроили его для отображения нашего первого кадра страницы, мы можем добавить его в каталог страниц и удалить временное отображение.

10.3 Куча ядра

До сих пор мы могли работать только с данными фиксированного размера или напрямую с сырой памятью. Теперь, когда у нас есть распределитель кадров страниц, мы можем реализовать malloc и free для использования в ядре.

Керниган и Ритчи [8] имеют пример реализации в своей книге [8], из которого мы можем черпать вдохновение. Единственная модификация, которую нам нужно сделать, - это заменить вызовы sbrk/brk на вызовы распределителя кадров страниц, когда требуется больше памяти. Мы также должны убедиться, что отображаем кадры страниц, возвращаемые распределителем кадров страниц, на виртуальные адреса. Правильная реализация также должна возвращать кадры страниц распределителю кадров страниц при вызове free, когда освобождаются достаточно большие блоки памяти.

10.4 Дополнительное чтение

Страница OSDev wiki о распределении кадров страниц: http://wiki.osdev.org/Page_Frame_Allocation

11. Пользовательский режим

Пользовательский режим теперь почти в пределах нашей досягаемости, осталось сделать всего несколько шагов, чтобы достичь его. Хотя эти шаги могут показаться простыми так, как они представлены в этой главе, их реализация может быть сложной, так как есть много мест, где небольшие ошибки могут вызвать ошибки, которые трудно найти.

11.1 Сегменты для пользовательского режима

Чтобы включить пользовательский режим, нам нужно добавить еще два сегмента в GDT. Они очень похожи на сегменты ядра, которые мы добавили, когда настраивали GDT в главе о сегментации:

Необходимые дескрипторы сегментов для пользовательского режима.

Индекс	Смещение	Имя	Диапазон адресов	Тип	DPL
3	0x18	сегмент кода пользователя	0x00000000 - 0xFFFFFFFF	RX	PL3
4	0x20	сегмент данных пользователя	0x00000000 - 0xFFFFFFFF	RW	PL3

Разница в DPL, который теперь позволяет коду выполняться на уровне PL3. Сегменты все еще могут использоваться для адресации всего адресного пространства, просто использование этих сегментов для кода пользовательского режима не защитит ядро. Для этого нам нужна страничная организация.

11.2 Настройка для пользовательского режима

Есть несколько вещей, которые нужны каждому процессу пользовательского режима:

Кадры страниц для кода, данных и стека. На данный момент достаточно выделить один кадр страницы для стека и достаточно кадров страниц для кода программы. Не беспокойтесь о настройке стека, который может расти и уменьшаться на этом этапе, сосредоточьтесь на том, чтобы сначала получить

работающую базовую реализацию.
Двоичный файл из модуля GRUB должен быть скопирован в кадры страниц, используемые для кода программы.
Необходимы каталог страниц и таблицы страниц для отображения кадров страниц, описанных выше, в память. Как минимум, необходимы две таблицы страниц, потому что код и данные должны быть отображены по адресу 0x00000000 и увеличиваться, а стек должен начинаться сразу ниже ядра, по адресу 0xBFFFFFFB, и расти в сторону меньших адресов. Флаг U/S должен быть установлен, чтобы разрешить доступ на уровне PL3.
Может быть удобно хранить эту информацию в структуре, представляющей процесс. Эту структуру процесса можно динамически выделить с помощью функции malloc ядра.

11.3 Вход в пользовательский режим

Единственный способ выполнить код с более низким уровнем привилегий, чем текущий уровень привилегий (CPL), - это выполнить инструкцию iret или lret - возврат из прерывания или длинный возврат соответственно.

Чтобы войти в пользовательский режим, мы настраиваем стек так, как если бы процессор вызвал межпривилегионное прерывание. Стек должен выглядеть следующим образом:

```
 [esp + 16]  ss      ; селектор сегмента стека, который мы хотим для пользовательского режима
[esp + 12]  esp     ; указатель стека пользовательского режима
[esp +  8]  eflags  ; флаги управления, которые мы хотим использовать в пользовательском режиме
[esp +  4]  cs      ; селектор сегмента кода
[esp +  0]  eip     ; указатель инструкции кода пользовательского режима для выполнения
```

См. руководство Intel [33], раздел 6.2.1, рисунок 6-4 для получения дополнительной информации.

Инструкция iret затем прочитает эти значения из стека и заполнит соответствующие регистры. Перед выполнением iret нам нужно переключиться на каталог страниц, который мы настроили для процесса пользовательского режима. Важно помнить, что для продолжения выполнения кода ядра после переключения PDT ядро должно быть отображено. Один из способов достичь этого - иметь отдельный PDT для ядра, который отображает все данные по адресу 0xC0000000 и выше, и объединять его с пользовательским PDT (который отображает только ниже 0xC0000000) при выполнении переключения. Помните, что физический адрес PDT должен использоваться при установке регистра cr3.

Регистр eflags содержит набор различных флагов, указанных в разделе 2.3 руководства Intel [33]. Наиболее важный для нас - это флаг прерываний (IF). Ассемблерная инструкция sti не может быть использована в уровне привилегий 3 для включения прерываний. Если прерывания отключены при входе в пользовательский режим, то прерывания не могут быть включены после входа в пользовательский режим. Установка флага IF в записи eflags на стеке включит прерывания в пользовательском режиме, так как ассемблерная инструкция iret установит регистр eflags соответствующим значением на стеке.

На данный момент у нас должны быть отключены прерывания, так как требуется немного больше работы для правильной работы межпривилегионных прерываний (см. раздел "Системные вызовы").

Значение eip на стеке должно указывать на точку входа для пользовательского кода - 0x00000000 в нашем случае. Значение esp на стеке должно быть там, где начинается стек - 0xBFFFFFFB (0xC0000000 - 4).

Значения cs и ss на стеке должны быть селекторами сегментов для пользовательского кода и пользовательских данных соответственно. Как мы видели в главе о сегментации, два младших бита селектора сегмента - это RPL - Requested Privilege Level. При использовании iret для входа в PL3, RPL для cs и ss должен быть 0x3. Следующий код показывает пример:

```
 USER_MODE_CODE_SEGMENT_SELECTOR equ 0x18
USER_MODE_DATA_SEGMENT_SELECTOR equ 0x20
mov cs, USER_MODE_CODE_SEGMENT_SELECTOR | 0x3
mov ss, USER_MODE_DATA_SEGMENT_SELECTOR | 0x3
```

Регистр ds и другие регистры сегментов данных должны быть установлены в тот же селектор сегмента, что и ss. Их можно установить обычным способом, с помощью ассемблерной инструкции mov.

Теперь мы готовы выполнить iret. Если все было настроено правильно, у нас теперь должно быть ядро, которое может входить в пользовательский режим.

11.4 Использование C для программ пользовательского режима

Когда C используется в качестве языка программирования для программ пользовательского режима, важно подумать о структуре файла, который будет результатом компиляции.

Причина, по которой мы можем использовать ELF [18] в качестве формата файла для исполняемого файла ядра, заключается в том, что GRUB умеет анализировать и интерпретировать формат файла ELF. Если бы мы реализовали анализатор ELF, мы могли бы компилировать программы пользовательского режима в ELF-бинарники. Мы оставляем это в качестве упражнения для читателя.

Одна вещь, которую мы можем сделать, чтобы упростить разработку программ пользовательского режима, - это разрешить программам быть написанными на C, но компилировать их в плоские бинарники вместо ELF-бинарников. В C layout генерируемого кода менее предсказуем, и точка входа, main, может не находиться по смещению 0 в бинарнике. Один из распространенных способов обойти это - добавить несколько строк ассемблера, размещенных по смещению 0, которые вызывают main:

```
extern main

section .text
    ; push argv
    ; push argc
    call main
    ; main вернул, eax - возвращаемое значение
    jmp  $    ; бесконечный цикл
```

Если этот код сохранен в файле start.s, то следующий код показывает пример скрипта линкера, который помещает эти инструкции первыми в исполняемый файл (помните, что start.s компилируется в start.o):

```
OUTPUT_FORMAT("binary")    /* вывод в плоский бинарник */

SECTIONS
{
    . = 0;                 /* перемещаем на адрес 0 */

    .text ALIGN(4):
    {
        start.o(.text)     /* включаем секцию .text из start.o */
        *(.text)           /* включаем все другие секции .text */
    }

    .data ALIGN(4):
    {
        *(.data)
    }

    .rodata ALIGN(4):
    {
        *(.rodata*)
    }
}
```

Примечание: *(.text) не будет включать секцию .text из start.o снова.

С этим скриптом мы можем писать программы на C или ассемблере (или любом другом языке, который компилируется в объектные файлы, линкуемые с ld), и легко загружать и отображать их для ядра (.rodata будет отображен как доступный для записи, хотя).

При компиляции пользовательских программ мы хотим следующие флаги GCC:

```
-m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles -nodefaultlibs
```

Для линковки следует использовать следующие флаги:

```
-T link.ld -melf_i386  # эмулировать 32-битный ELF, бинарный вывод указан
                       # в скрипте линкера
```

Опция -T указывает линкеру использовать скрипт линкера link.ld.

11.4.1 Библиотека C

Теперь может быть интересно начать думать о написании небольшой "стандартной библиотеки" для ваших программ. Некоторый функционал требует системных вызовов для работы, но некоторые, например, функции в string.h, не требуют.

11.5 Дополнительное чтение

Густаво Дуарте имеет статью об уровнях привилегий: http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection

12 Файловые системы

Наличие файловой системы в нашей операционной системе не является обязательным требованием, но это очень полезная абстракция, которая часто играет центральную роль во многих ОС, особенно UNIX-подобных. Прежде чем начать реализацию поддержки множества процессов и системных вызовов, стоит рассмотреть создание простой файловой системы.

12.1 Зачем нужна файловая система?

Как указывать, какие программы запускать в нашей ОС? Какая программа должна запускаться первой? Как программам выводить данные или считывать ввод?

В UNIX-подобных системах с их концепцией "почти всё — это файл" эти проблемы решаются файловой системой. (Также может быть интересно почитать о проекте Plan 9, который развивает эту идею дальше.)

12.2 Простая файловая система только для чтения

Простейшая файловая система — это то, что у нас уже есть: один файл, существующий только в оперативной памяти, загруженный GRUB до старта ядра. По мере роста ядра и ОС этого, вероятно, станет недостаточно.

Чуть более продвинутая файловая система — это файл с метаданными. Метаданные могут описывать тип файла, его размер и так далее. Можно создать утилиту, которая на этапе сборки добавляет эти метаданные к файлу. Таким образом, "файловую систему в файле" можно построить, объединив несколько файлов с метаданными в один большой файл. Результатом будет файловая система только для чтения, размещённая в памяти (после загрузки файла GRUB).

Программа, строящая файловую систему, может обходить каталоги на хостовой системе и добавлять все подкаталоги и файлы в целевую файловую систему. Каждый объект файловой системы (каталог или файл) может состоять из заголовка и тела, где тело файла — это сам файл, а тело каталога — список записей (имён и "адресов" других файлов и каталогов).

Каждый объект в такой файловой системе будет непрерывным, что упростит его чтение из памяти для ядра. Все объекты также будут иметь фиксированный размер (кроме последнего, который может расти), поэтому добавление новых файлов или изменение существующих будет затруднено.

12.3 Inode и файловые системы с записью

Когда возникнет необходимость в файловой системе с возможностью записи, стоит изучить концепцию inode. Рекомендуемая литература приведена в разделе "Дополнительные материалы".

12.4 Виртуальная файловая система

Какую абстракцию использовать для чтения и записи на устройства, такие как экран и клавиатура?

Виртуальная файловая система (VFS) создаёт абстракцию поверх конкретных файловых систем. VFS в основном предоставляет систему путей и иерархию файлов, делегируя операции с файлами нижележащим файловым системам. Оригинальная статья о VFS лаконична и стоит прочтения (см. раздел "Дополнительные материалы").

С помощью VFS можно смонтировать специальную файловую систему по пути /dev. Эта файловая система будет обрабатывать все устройства, такие как клавиатура и консоль. Однако можно пойти традиционным UNIX-подходом с major/minor номерами устройств и mknod для создания специальных файлов устройств. Какой подход выбрать — решать вам, при создании абстракций нет правильного или неправильного (хотя некоторые абстракции оказываются гораздо полезнее других).

12.5 Дополнительные материалы

Идеи ОС Plan 9: http://plan9.bell-labs.com/plan9/index.html

Статья в Wikipedia об inode: http://en.wikipedia.org/wiki/Inode и структуре указателей inode: http://en.wikipedia.org/wiki/Inode_pointer_structure

Оригинальная статья о концепции vnode и виртуальной файловой системе:
http://www.arl.wustl.edu/~fredk/Courses/cs523/fall01/Papers/kleiman86vnodes.pdf

Обсуждение идеи специальной файловой системы для /dev:
http://static.usenix.org/publications/library/proceedings/bsdcon02/full_papers/kamp/kamp_html/index.html

13 Системные вызовы

Системные вызовы — это способ взаимодействия пользовательских приложений с ядром: запрос ресурсов, выполнение операций и т.д. API системных вызовов — это наиболее открытая пользователям часть ядра, поэтому его проектирование требует тщательного обдумывания.

13.1 Проектирование системных вызовов

Нам, разработчикам ядра, предстоит спроектировать системные вызовы для разработчиков приложений. Можно вдохновляться стандартами POSIX или, если это кажется слишком сложным, просто взять за основу вызовы Linux, выбрав подходящие. Рекомендуемая литература приведена в конце главы.

13.2 Реализация системных вызовов

Традиционно системные вызовы выполняются с помощью программных прерываний. Пользовательские приложения помещают нужные значения в регистры или на стек, а затем инициируют предопределённое прерывание, которое передаёт управление ядру. Номер прерывания зависит от ядра; в Linux используется номер 0x80 для идентификации системного вызова.

При выполнении системных вызовов текущий уровень привилегий обычно меняется с PL3 на PL0 (если приложение работает в пользовательском режиме). Чтобы это работало, DPL записи в IDT для прерывания системного вызова должен разрешать доступ с уровня PL3.

При межпривилегийных прерываниях процессор сохраняет несколько важных регистров в стеке — те же, что использовались для перехода в пользовательский режим (см. рис. 6-4, раздел 6.12.1 в руководстве Intel [33]). Какой стек при этом используется? В том же разделе указано, что если прерывание приводит к выполнению кода с более низким уровнем привилегий, происходит переключение стека. Новые значения для регистров ss и esp загружаются из текущего сегмента состояния задачи (TSS). Структура TSS описана в разделе 7.2.1 руководства Intel [33].

Для работы системных вызовов необходимо настроить TSS перед переходом в пользовательский режим. Настройка может быть выполнена на C путём заполнения полей ss0 и esp0 "упакованной структуры", представляющей TSS. Перед загрузкой этой структуры в процессор в GDT нужно добавить дескриптор TSS. Структура дескриптора TSS описана в разделе 7.2.2 [33].

Текущий сегмент TSS задаётся загрузкой его селектора в регистр tr с помощью инструкции ltr. Если дескриптор сегмента TSS имеет индекс 5 (смещение

5 * 8 = 40 = 0x28), именно это значение нужно загрузить в tr.

При переходе в пользовательский режим в главе "Переход в пользовательский режим" мы отключали прерывания при выполнении на уровне PL3. Поскольку системные вызовы используют прерывания, в пользовательском режиме прерывания должны быть включены. Установка бита IF в значении eflags в стеке приведёт к тому, что инструкция iret включит прерывания (поскольку значение eflags из стека будет загружено в регистр eflags).

13.3 Дополнительные материалы

Статья в Wikipedia о POSIX: http://en.wikipedia.org/wiki/POSIX

Список системных вызовов Linux: http://bluemaster.iu.hio.no/edu/dark/lin-asm/syscalls.html

Статья в Wikipedia о системных вызовах: http://en.wikipedia.org/wiki/System_call

Разделы руководства Intel [33] о прерываниях (глава 6) и TSS (глава 7) содержат все необходимые детали.

14 Многозадачность

Как сделать так, чтобы несколько процессов казались выполняющимися одновременно? Сегодня есть два ответа:

Благодаря многоядерным процессорам или системам с несколькими процессорами два процесса действительно могут выполняться одновременно на разных ядрах/процессорах.

Имитировать это, то есть быстро переключаться между процессами (быстрее, чем может заметить человек). В каждый момент выполняется только один процесс, но быстрое переключение создаёт впечатление их "одновременной" работы.

Поскольку ОС, создаваемая в этой книге, не поддерживает многоядерные процессоры, остаётся только второй вариант. Часть ОС, отвечающая за быстрое переключение между процессами, называется алгоритмом планирования.

14.1 Создание новых процессов

Обычно новые процессы создаются с помощью двух системных вызовов: fork и exec. fork создаёт точную копию текущего процесса, а exec заменяет текущий процесс указанной программой из файловой системы. Рекомендуем начать с реализации exec, так как этот вызов выполняет почти те же шаги, что описаны в разделе "Настройка перехода в пользовательский режим" главы "Пользовательский режим".

14.2 Кооперативная многозадачность с добровольной передачей управления

Самый простой способ быстрого переключения между процессами — если сами процессы отвечают за переключение. Процессы работают некоторое время, а затем сообщают ОС (через системный вызов), что можно переключиться на другой процесс. Передача управления другому процессу называется "уступкой" (yielding), а когда процессы сами управляют планированием, это называется кооперативной многозадачностью (поскольку все процессы должны сотрудничать).

При уступке необходимо сохранить всё состояние процесса (все регистры), предпочтительно в структуре процесса в куче ядра. При переходе к новому процессу все регистры восстанавливаются из сохранённых значений.

Планирование можно реализовать, ведя список работающих процессов. Системный вызов yield должен запускать следующий процесс из списка и помещать текущий в конец (возможны и другие схемы).

Передача управления новому процессу выполняется с помощью инструкции iret так же, как описано в разделе "Переход в пользовательский режим".

Рекомендуем начать с реализации кооперативной многозадачности, а затем уже переходить к вытесняющей. Поскольку кооперативное планирование детерминировано, его проще отлаживать.

14.3 Вытесняющая многозадачность с прерываниями

Вместо того чтобы позволять процессам самим решать, когда передавать управление, ОС может автоматически переключать процессы через короткие промежутки времени. ОС может настроить программируемый интервальный таймер (PIT) на генерацию прерываний, например, каждые 20 мс. В обработчике прерывания PIT ОС будет переключать процессы. Такой подход называется вытесняющей многозадачностью.

14.3.1 Программируемый интервальный таймер

Для вытесняющего планирования сначала нужно настроить PIT на генерацию прерываний каждые x миллисекунд (где x должно быть настраиваемым).

Конфигурация PIT аналогична настройке других устройств: байт отправляется на порт ввода-вывода. Командный порт PIT — 0x43. Рекомендуем использовать следующие параметры:

1. Генерация прерываний (канал 0)
2. Отправка делителя как младший, затем старший байт
3. Использование меандра
4. Двоичный режим Это соответствует байту конфигурации 00110110.

Интервал между прерываниями задаётся делителем. PIT работает на частоте 1193182 Гц по умолчанию. Делитель 10 даёт частоту 1193182 / 10 = 119318 Гц. Делитель 16-битный, поэтому частоту можно настроить в диапазоне от 1193182 Гц до 18.2 Гц. Рекомендуется создать функцию, преобразующую интервал в миллисекундах в делитель.

Делитель отправляется на порт данных канала 0 PIT (0x40) сначала младший, затем старший байт.

14.3.2 Отдельные стеки ядра для процессов

Если все процессы используют один стек ядра (указанный в TSS), возникнут проблемы, если процесс будет прерван в режиме ядра. Новый процесс будет

использовать тот же стек и перезапишет данные предыдущего процесса. Чтобы избежать этого, каждый процесс должен иметь свой стек ядра, а TSS должен обновляться при переключении процессов.

14.3.3 Сложности вытесняющего планирования

При вытесняющем планировании возникает проблема, которой нет в кооперативном: процессы могут прерываться как в пользовательском, так и в режиме ядра. Если прерывание происходит без изменения уровня привилегий, CPU не сохраняет регистры ss и esp и не переключает стек. Эту проблему решают, вычисляя значение esp на момент прерывания.

Дополнительная сложность — переключение на процесс, который должен выполняться в режиме ядра. Поскольку iret выполняется без изменения уровня привилегий, CPU не обновляет esp значением из стека — это нужно делать вручную.

14.4 Дополнительные материалы

Различные алгоритмы планирования: http://wiki.osdev.org/Scheduling_Algorithms

14.4 Ссылки

[1] Andrew Tanenbaum, 2007. Modern operating systems, 3rd edition. Prentice Hall, Inc.,

[2] The royal institute of technology, http://www.kth.se,

[3] Wikipedia, Hexadecimal, http://en.wikipedia.org/wiki/Hexadecimal,

[4] OSDev, OSDev, http://wiki.osdev.org/Main_Page,

[5] James Molloy, James m's kernel development tutorial, http://www.jamesmolloy.co.uk/tutorial_html/,

[6] Canonical Ltd, Ubuntu, http://www.ubuntu.com/,

[7] Oracle, Oracle vM virtualBox, http://www.virtualbox.org/,

[8] Dennis M. Ritchie Brian W. Kernighan, 1988. The c programming language, second edition. Prentice Hall, Inc.,

[9] Wikipedia, C (programming language), http://en.wikipedia.org/wiki/C_(programming_language),

[10] Free Software Foundation, GCC, the gNU compiler collection, http://gcc.gnu.org/,

[11] NASM, NASM: The netwide assembler, http://www.nasm.us/,

[12] Wikipedia, Bash, http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29,

[13] Free Software Foundation, GNU make, http://www.gnu.org/software/make/,

[14] Volker Ruppert, bochs: The open souce iA-32 emulation project, http://bochs.sourceforge.net/,

[15] QEMU, QEMU, http://wiki.qemu.org/Main_Page,

[16] Wikipedia, BIOS, https://en.wikipedia.org/wiki/BIOS,

[17] Free Software Foundation, GNU gRUB, http://www.gnu.org/software/grub/,

[18] Wikipedia, Executable and linkable format, http://en.wikipedia.org/wiki/Executable_and_Linkable_Format,

[19] Free Software Foundation, Multiboot specification version 0.6.96, http://www.gnu.org/software/ grub/manual/multiboot/multiboot.html,

[20] GNU, GNU binutils, http://www.gnu.org/software/binutils/,

[21] Lars Nodeen, Bug #426419: configure: error: GRUB requires a working absolute objcopy, https://bugs.launchpad.net/ubuntu/+source/grub/+bug/426419,

[22] Wikipedia, ISO image, http://en.wikipedia.org/wiki/ISO_image,

[23] Bochs, bochsrc, http://bochs.sourceforge.net/doc/docbook/user/bochsrc.html,

[24] NASM, RESB and friends: Declaring uninitialized data, http://www.nasm.us/doc/nasmdoc3.htm,

[25] Wikipedia, x86 calling conventions, http://en.wikipedia.org/wiki/X86_calling_conventions,

[26] Wikipedia, Framebuffer, http://en.wikipedia.org/wiki/Framebuffer,

[27] Wikipedia, VGA-compatible text mode, http://en.wikipedia.org/wiki/VGA-compatible_text_mode,

[28] Wikipedia, ASCII, https://en.wikipedia.org/wiki/Ascii,

[29] OSDev, VGA hardware, http://wiki.osdev.org/VGA_Hardware,

[30] Wikipedia, Serial port, http://en.wikipedia.org/wiki/Serial_port,

[31] OSDev, Serial ports, http://wiki.osdev.org/Serial_ports,

[32] WikiBooks, Serial programming/8250 uART programming, http://en.wikibooks.org/wiki/Serial_Programming/ 8250_UART_Programming,

[33] Intel, Intel 64 and iA-32 architectures software developer's manual vol. 3A, http://www.intel.com/content/ www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html/,

[34] NASM, Multi-line macros, http://www.nasm.us/doc/nasmdoc4.html#section-4.3,

[35] SIGOPS, i386 interrupt handling, http://www.acm.uiuc.edu/sigops/roll_your_own/i386/irq.html,

[36] Andries Brouwer, Keyboard scancodes, http://www.win.tue.nl/,

[37] Steve Chamberlain, Using ld, the gNU linker, http://www.math.utah.edu/docs/info/ld_toc.html,

[38] OSDev, Page frame allocation, http://wiki.osdev.org/Page_Frame_Allocation,

[39] OSDev, Programmable interval timer, http://wiki.osdev.org/Programmable_Interval_Timer,

Загрузчик должен помещаться в загрузочный сектор MBR жёсткого диска, размер которого всего 512 байт.