Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Д. Е. Оношко, В. В. Бахтизин

ОСНОВЫ РАЗРАБОТКИ ОПЕРАЦИОННЫХ СИСТЕМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для специальности 1-40 01 01 «Программное обеспечение информационных технологий»

Минск БГУИР 2022

УДК 004.451(076)
ББК 32.972.11я73
О-59

Р е ц е н з е н т ы:

кафедра информатики и веб-дизайна учреждения образования
«Белорусский государственный технологический университет»
(протокол №3 от 14.10.2020);

доцент кафедры многопроцессорных систем и сетей
Белорусского государственного университета
кандидат физико-математических наук, доцент Т. В. Соболева

Оношко, Д. Е.

О-59 Основы разработки операционных систем : учеб.-метод. пособие / Д. Е. Оношко, В. В. Бахтизин. – Минск : БГУИР, 2022. – 123 с. : ил.

ISBN 978-985-543-611-0.

Содержит теоретический материал, посвящённый основам разработки операционных систем, примеры разработки отдельных модулей операционных систем и примерные задания для лабораторных работ и курсовых проектов по дисциплине «Языки программирования». Может быть использовано для организации самостоятельной работы студентов.

УДК 004.451(076)
ББК 32.972.11я73

ISBN 978-985-543-611-0

Содержание

Введение....4
Методические указания....6
Шаг 1. Простейший загрузчик....8
Что происходит при включении компьютера?....8
Зачем нужна *BIOS*?....10
Чем *BIOS* отличается от *UEFI* и что такое *CSM*?....11
Как *BIOS* загружает операционную систему?....13
Пример 1.1. «*Hello, world!*»....15
Что такое *CHS* и *LBA*?....24
Пример 1.2. Простейший загрузчик....28
Пример 1.3. Доработка простейшего загрузчика....36
Подведение итогов....39
Задания....40
Шаг 2. Загрузка с поддержкой файловых систем....41
Зачем загрузчику поддерживать файловые системы?....41
Как устроена файловая система *FAT*?....42
Пример 2.1. Образ пустой *FAT*12-дискеты....44
Пример 2.2. *FAT*12-дискета с файлами....53
Пример 2.3. Загрузочная дискета *MS-DOS*....59
Как написать свой загрузчик для *FAT*?....70
Подведение итогов....72
Задания....72
Шаг 3. Основные модули ядра операционной системы....73
На какие вопросы нужно ответить, приступая к разработке ОС?....73
Какую архитектуру использовать для операционной системы?....75
С чего начинает работу ядро операционной системы?....78
Пример 3.1. Разработка простого менеджера памяти....79
Как придумать формат исполняемого файла?....93
Пример 3.2. Проектирование формата исполняемого файла....94
Как загрузить с диска драйвер для работы с диском?....100
Пример 3.3. Загрузка и запуск программы....102
Подведение итогов....106
Задания....106
Шаг 4. Поддержка носителей с несколькими разделами....107
Что такое *MBR*?....107
Чем *MBR* отличается от *GPT*?....111
Пример 4.1. Загрузка с *USB*-флэш-накопителя....115
Подведение итогов....120
Задания....121
Список использованных источников....122

Введение

Разработка операционной системы – непростая, но увлекательная задача. Написать свою операционную систему, пусть даже совсем примитивную, – предмет гордости любого программиста, ведь для этого требуется не только глубокое понимание принципов работы современного программного и аппаратного обеспечения, но и широкий кругозор, умение разрабатывать не только алгоритмы, но и архитектуру сложных систем, принимать решения и расставлять приоритеты и, наконец, работать с технической документацией.

В наше время, когда сложнейшие задачи могут решаться в три строчки кода благодаря высокоуровневым языкам и библиотекам, а кроссплатформенность для многих стала едва ли не главной ценностью, погружение в такую низкоуровневую область, как разработка операционных систем, может показаться бесполезным занятием, тем более, что составить конкуренцию существующим операционным системам вряд ли удастся.

Тем не менее именно сейчас такое погружение может быть особенно полезным.

Во-первых, это помогает очень детально разобраться с принципами функционирования выбранной платформы. «Для кроссплатформенной разработки это не нужно и даже вредно», – скажет кто-то. И будет неправ. Как бы хорошо современные инструменты ни скрывали от программиста механизмы работы нижележащих платформ, следует помнить, что у каждой из них есть свои особенности и ограничения. Именно в них и заключается главная сложность кроссплатформенной разработки: любой неучтённый нюанс может привести к резкому снижению производительности, неработоспособности отдельных функций программы, а то и вовсе ошибке, которая проявит себя когда-нибудь потом, спустя пять лет и восемь версий платформы. Изучение платформ в этом смысле похоже на изучение языков: хорошо зная английский язык, освоить, например, французский или итальянский будет намного проще, потому что можно делать это по аналогии, в сравнении. И чем глубже изучена одна платформа, тем больше отличий, особенностей удастся разглядеть при освоении другой.

Во-вторых, даже самая простая операционная система – это достаточно сложный проект, который позволяет выработать и развить навыки проектирования архитектуры программных средств. Особенно ценно то, что здесь необходимость декомпозиции на модули и проектирования их взаимодействия возникает уже при относительно небольшом объёме исходных кодов: даже в сравнительно небольшом проекте появляется возможность развить навыки, необходимые опытному программисту, причём ввиду специфичности самой задачи в рамках одного проекта удаётся продемонстрировать решение самых разнообразных проблем, редко встречающихся одновременно в других проектах.

В-третьих, при разработке операционной системы приходится взаимодействовать с программными и аппаратными средствами, созданными сторонними специалистами высокой квалификации. Это позволяет не только попрактиковаться в чтении документации, но и на примере чужих разработок увидеть интересные технические решения, расширить собственный арсенал приёмов.

Наконец одна из самых важных причин – возможность на конкретных примерах увидеть, как происходит развитие технологий, какие решения получают распространение и развитие, а от каких отказываются. Знание таких закономерностей позволяет специалисту грамотно осуществлять выбор инструментария для своих проектов, прогнозировать, какие технологии, языки программирования и библиотеки будут популярны в будущем, а какие потеряют актуальность через несколько лет, что в свою очередь позволяет правильно расставлять приоритеты в их изучении.

Данное учебно-методическое пособие содержит пошаговое описание процесса разработки учебной операционной системы ***«OSE!»***. Авторами сознательно принято решение описывать разработку операционной системы, которая обеспечивает поддержку в том числе и ранних моделей персональных компьютеров. Это позволяет создать дополнительные ограничения и тем самым сделать очевиднее пользу обсуждаемых приёмов, а также более полно продемонстрировать некоторые технические решения, детали реализации которых в случае поддержки только современных компьютеров оказались бы скрытыми за абстракциями.

Вместе с исходными кодами учебной операционной системы ***«OSE!»*** данное пособие образует учебно-методический комплекс, позволяющий не только начать разработку собственной операционной системы с нуля, но и получить сопутствующие знания и навыки на примере готового проекта, сразу же применяя их в разработке предназначенных для него дополнительных модулей и прикладного ПО. Целевая платформа ***«OSE!»*** – компьютеры, совместимые с *IBM PC*, на базе *Intel*-совместимых процессоров, т. е. процессоров архитектур *Intel IA*-32, *AMD*64 и др.

Пособие разделено на шаги – возможные этапы разработки, результатом каждого из которых является законченный прототип операционной системы или её модулей. Использованная в пособии последовательность этих шагов не является единственно возможной, однако, по мнению авторов, наиболее эффективна в учебных целях. Каждый шаг включает в себя необходимые теоретические сведения, а также практическую часть, демонстрирующую их применение.

Методические указания

Учебно-методический комплекс, в состав которого входит настоящее учебно-методическое пособие, разработан для использования в рамках дисциплины «Языки программирования» специальности 1-40 01 01 «Программное обеспечение информационных технологий» и предназначается для успевающих студентов специальности, проявляющих интерес к углубленному изучению тем повышенной сложности.

Для успешного освоения материала студент должен быть предварительно ознакомлен с дисциплинами «Основы алгоритмизации и программирования» и «Языки программирования». В частности, при изложении материала предполагается, что студент:

– свободно владеет *Pascal*-подобным языком программирования (например, *Delphi*);

– свободно владеет языком ассемблера для реального режима архитектуры *IA*-32;

– знаком с принципами и базовыми архитектурными приёмами разработки программного обеспечения;

– знаком с внутренним устройством современных ЭВМ.

Учебно-методический комплекс может быть использован в учебном процессе несколькими способами в зависимости от преследуемых целей.

Руководство по курсовому проектированию. Студенту или группе студентов может быть предложена разработка в рамках курсового проектирования прототипа операционной системы. Настоящее учебно-методическое пособие при этом может быть использовано как пошаговое руководство, позволяющее решить поставленную задачу с минимальным риском срыва сроков выполнения проекта, а также как сборник основной информации, необходимой для выполнения такой разработки.

Кодовая база для курсового проектирования. Помимо полностью самостоятельной разработки прототипа операционной системы «с нуля», студенты могут использовать отдельные части исходных кодов для того, чтобы сократить трудозатраты на решение типовых задач, например, таких, как загрузка ядра операционной системы или подготовка образов загрузочных накопителей.

Самостоятельный учебный проект. Студентам может быть предложена разработка отдельных компонентов или прикладного программного обеспечения для ***«OSE!»*** в рамках лабораторного практикума или курсового проектирования. В этом случае проект дорабатывается силами студентов под руководством преподавателя, а учебно-методическое пособие используется в качестве расширенной документации к операционной системе, способствующей более быстрому погружению в проект новых участников.

«Песочница» для работы с аппаратным обеспечением. При выполнении заданий лабораторного практикума по дисциплинам, посвящённым работе с аппаратным обеспечением, ***«OSE!»*** может выступать в роли среды, обеспечи-

вающей более простой доступ к аппаратному обеспечению, чем в популярных операционных системах.

Учебный пример (наглядное пособие). При изучении дисциплин, посвящённых внутреннему устройству операционных систем, исходные коды ***«OSE!»*** могут быть использованы в качестве примера одной из возможных, максимально простых, реализаций. Учебно-методическое пособие в этом случае становится дополнительным источником материала по дисциплине.

Шаг 1. Простейший загрузчик

Любое программное обеспечение разрабатывается для того, чтобы однажды быть запущенным и приступить к решению возложенных на него задач, – без этого оно так и осталось бы бесполезным набором байтов. Но как инициировать запуск своей (пока ещё будущей) операционной системы? И как вообще указать, какой именно код будет выполнять роль операционной системы?

Что происходит при включении компьютера?

При включении компьютера производится начальная инициализация процессора. Она включает в себя:

– выполнение встроенной процедуры самотестирования (*BIST, Built-In Self-Test*);

– запись начальных значений в регистры;

– сброс кэшей и других внутренних буферов (например, используемых для аппаратных оптимизаций);

– решение иных задач по инициализации.

Конечной целью является переход процессора в заранее известное и подробно описанное в документации состояние. В частности, задокументированным является физический адрес, по которому процессор начнёт считывать первую инструкцию. В ранних моделях процессоров *Intel* это был физический адрес `FFFF0h`, в более поздних – `FFFFFFF0h`. Более подробную информацию о начальном состоянии различных моделей процессоров *Intel* можно найти в документации производителя.

Содержимое оперативной памяти после включения компьютера не определено. Например, если с момента отключения питания прошло совсем немного времени, ячейки оперативного запоминающего устройства (ОЗУ) могут продолжать хранить записанные в них значения. В иных случаях конкретные значения будут зависеть от вида оперативной памяти, времени, прошедшего с момента отключения, и т. п. Как бы то ни было, кода, способного выполнять полезные действия, в ОЗУ не будет.

Ключом к пониманию того, как «осмысленный» код оказывается доступным для выполнения процессором, является схема, изображённая на рис. 1.1.

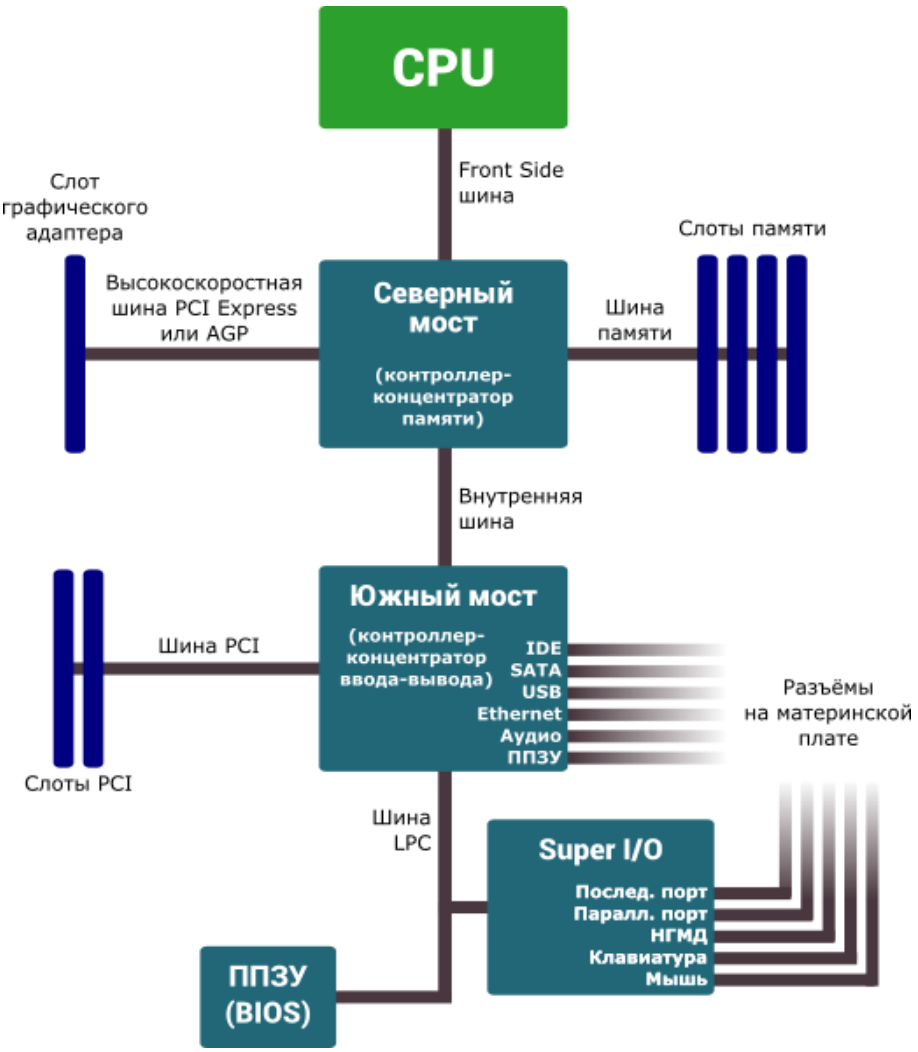

Рис. 1.1. Логическая структура материнской платы

Как следует из схемы, взаимодействие между процессором и ОЗУ происходит не напрямую, а опосредованно, через так называемый северный мост (англ. *northbridge*). В настоящее время северный мост часто встраивается в сам процессор, однако логически по-прежнему является самостоятельным узлом. Компания *Intel* для собирательного обозначения подобных узлов ввела понятие «*uncore*», в противовес «*core*» – традиционным вычислительным ядрам. Объединение логически различных узлов в общем корпусе позволяет повысить эффективность их взаимодействия, поэтому северный мост – главное связующее звено между процессором и остальными аппаратными узлами – оказался закономерным кандидатом для такого объединения.

При попытке процессора получить доступ по адресам из определённого диапазона северный мост перенаправляет эти запросы в программируемое постоянное запоминающее устройство (ППЗУ), куда записывается так называемая *BIOS* – базовая система ввода-вывода.

Зачем нужна *BIOS*?

Теоретически операционную систему можно было бы записывать прямо в ППЗУ, но при этом возник бы целый ряд проблем. Во-первых, изначально ПЗУ не были перезаписываемыми, что серьёзно затруднило бы обновление или замену операционной системы. Во-вторых, даже при наличии перезаписываемого (программируемого) ПЗУ процесс перезаписи может быть достаточно медленным и сложным с технической точки зрения. В-третьих, объём ПЗУ традиционно невелик, в т. ч. ввиду высокой стоимости производства, поэтому размещение в нём операционной системы серьёзно ограничило бы её разработчика.

Есть и ещё одна причина. Поставим себя на место производителей аппаратного обеспечения. С одной стороны, для того чтобы успешно конкурировать с другими компаниями, необходимо добавлять в свои устройства новые, уникальные функциональные возможности. С другой стороны, добавление новых возможностей требует поддержки со стороны программного обеспечения: кому нужна веб-камера с поддержкой запахов, если ни одна программа не умеет с ними работать?

Разумеется, производители оборудования стараются договариваться между собой о том, каким образом предоставлять доступ к возможностям устройств программному обеспечению. Тем не менее, иногда добавление расширенных возможностей невозможно без нарушения этих договорённостей. История персональных компьютеров знает примеры, когда правила игры на аппаратном уровне изменялись достаточно существенно: ярчайший пример – переход от шины *ISA* к шине *PCI*. Не следует также забывать и о том, что разные компьютеры оснащаются разными устройствами, для работы которых приходится поразному распределять аппаратные ресурсы.

В этих условиях появление промежуточного звена (его ещё называют встроенным программным обеспечением, *firmware* или прошивкой) между аппаратурой и программным обеспечением было неизбежным – таким промежуточным звеном для персональных компьютеров стала *BIOS*.

Идея заключается в том, что производитель материнской платы записывает в ППЗУ некоторый набор подпрограмм (это и есть *BIOS*), которые «знают», как взаимодействовать с аппаратурой конкретной модели компьютера, т. е. обращаются к ней именно так, как этого требует конкретное оборудование.

В итоге эти подпрограммы решают три основные задачи:

– выполняют проверку и начальную настройку устройств, подключённых к материнской плате (известна также под названием *POST – Power-On Self-Test*);

– выполняют поиск и загрузку операционной системы;

– предоставляют унифицированый (общий для компьютеров с различным оборудованием) программный интерфейс для взаимодействия с имеющимся оборудованием.

За решение первых двух задач отвечает код, получающий управление при включении или перезагрузке компьютера, – именно его первая инструкция должна оказаться по адресу `FFFFFFF0h` (или `FFFF0h` для старых моделей про-

цессоров *IA*-32). Подпрограммы же, отвечающие за третью задачу, обеспечивают работоспособность операционной системы до момента загрузки специализированных драйверов (если таковые будут использоваться).

Следует понимать, что *BIOS*, как правило, не разрабатывается отдельно для каждой модели и конфигурации. Вместо этого обычно применяется одна из универсальных *BIOS*, подходящих для целого класса компьютеров. В связи с этим программный интерфейс, т. е. набор функций, предоставляемый *BIOS*, достаточно ограничен и позволяет использовать лишь самые базовые возможности оборудования, имеющиеся в большинстве подобных устройств.

Чтобы задействовать все возможности того или иного устройства, операционная система, как правило, загружает соответствующий драйвер – программный компонент, который «умеет» работать с конкретным устройством, – и в дальнейшем прекращает использование функций *BIOS*. Тем не менее, сама загрузка драйвера требует взаимодействия с устройствами хранения данных, поэтому на начальных этапах работы ОС использование *BIOS* – необходимость.

Чем *BIOS* отличается от *UEFI* и что такое *CSM*?

Когда разрабатывались первые реализации *BIOS*, процессоры *Intel* поддерживали только реальный режим, вследствие чего и функции *BIOS* изначально использовали для приёма параметров, возврата результатов и собственно вызова те механизмы, которые поддерживались первыми процессорами семейства *x*86. В частности, в классических *BIOS*:

– параметры передаются через 16-битные регистры, для параметров больших размеров в регистрах передаётся адрес участка памяти;

– вызов функции осуществляется генерацией программного прерывания с определённым номером;

– результаты работы функций, как правило, возвращаются через 16-битные регистры.

Со временем в *Intel*-совместимых процессорах появился защищённый режим, а затем и его 64-битный подрежим, получивший название *long mode*. При этом механизм обработки прерываний стал привилегированной возможностью, доступ к которой напрямую из обычных прикладных программ был закрыт, 16-битные регистры оказались недостаточно большими на фоне объёмов обрабатываемых данных, а в моду вошли стековые соглашения вызова. И хотя при включении новые процессоры, как и раньше, начинали работу в реальном режиме, возможности классических *BIOS* стали упираться в его ограничения.

В то время когда разрабатывались первые реализации *BIOS*, вряд ли кто-то мог подумать, что в будущем размеры дисковых накопителей вместо сотен килобайт начнут измеряться терабайтами: для функций, которые изначально использовались для работы с дисками, способ передачи параметров был выбран так, что максимальный размер накопителя мог составлять всего лишь около 8 Гбайт. По мере того как диски наращивали ёмкость, расширялись возможности старых моделей и появлялись новые функции для работы с ними. В итоге

для выполнения одних и тех же задач сформировалось несколько разных наборов функций, причём в течение некоторого времени после введения новых функций они могли быть доступны не во всех реализациях *BIOS* или поддерживаться лишь частично.

Другое ограничение – адресация в реальном режиме: при 16-битных номерах сегментов и 16-битных смещениях внутри сегментов размер адресного пространства составляет лишь 1 Мбайт. И это при том, что тот же самый процессор в защищённом режиме может адресовать значительно больше памяти. Разумеется, если ОС использует функции *BIOS* только для начальной загрузки, а затем переключает процессор в защищённый режим и работает с аппаратурой напрямую (с помощью драйверов), ограничения реального режима – не большая проблема, но всё же избавиться от разработки части кода в условиях этих ограничений не отказались бы многие разработчики операционных систем.

В *Intel* осознавали проблему и во второй половине 1990-х годов вели разработку нового стандарта, который позволил бы снять не только ограничения классической *BIOS*, но и решить проблемы обратной совместимости. Это решение планировалось представить вместе с новой архитектурой *Itanium*, которую в *Intel* рассматривали как 64-битную замену архитектуре *IA*-32, однако новая платформа не нашла широкой поддержки. В результате нишу 64-битных процессоров для потребительского сегмента заняла архитектура *AMD*64, которая в отличие от *Itanium* представляла собой развитие *IA*-32, а не принципиально новую архитектуру.

Несмотря на это наработки *Intel* по замене классической *BIOS* никуда не делись и были опубликованы в начале 2000-х годов, а уже в 2005 году была основана организация *Unified EFI Forum*, которая впоследствии стала отвечать за разработку спецификаций *UEFI* – новой *BIOS*.

Строго говоря, *UEFI* – это спецификация программных интерфейсов, т. е. правил взаимодействия, а не сам код. Тем не менее, изменения оказались достаточно существенными, поэтому на сегодняшний день, как правило, под *BIOS* подразумевают старые, классические *BIOS*, а новые обозначают аббревиатурой *UEFI*.

Для разработчиков операционных систем основными отличиями *UEFI* от *BIOS* можно считать следующие:

– при запуске операционной системы процессор находится не в реальном, а в защищённом режиме (или его 64-битном подрежиме);

– используется объектно-ориентированный программный интерфейс: вместо процедур с регистровым соглашением вызова, вызываемых через механизм обработки прерываний, используются методы интерфейсов;

– упрощён процесс загрузки операционной системы: прямо в *UEFI* реализована (пусть и на примитивном уровне) поддержка популярных файловых систем, графических видеорежимов с высоким разрешением и т. п.

Набор предоставляемых функций стал существенно шире, и теперь они не имеют с функциями классической *BIOS* практически ничего общего, кроме

назначения – быть минимальным набором возможностей, необходимых операционной системе до момента, пока не будут загружены специфические драйверы для имеющихся устройств и ОС не сможет управлять их работой самостоятельно, – и стоящих за этими функциями принципов, подходов и технологий.

Впрочем, функции классических *BIOS* также остались доступными. Большинство современных реализаций *UEFI* включают в себя так называемый модуль поддержки совместимости – *CSM*, который представляет собой не что иное, как классический набор функций *BIOS*, но уже с современной «начинкой», т. е. кодом, внутри. Спецификации *UEFI* определяют, какие именно из классических функций и каким образом должны быть реализованы.

Выделяют четыре класса компьютеров в зависимости от поддержки ими интерфейсов *UEFI* и *BIOS*:

1) класс 0 – компьютеры с классической *BIOS*;

2) класс 1– компьютеры, которые всегда работают только в режиме *CSM* (т. е. реализация потенциально поддерживает новые возможности, но загрузка операционной системы осуществляется только классическим способом);

3) класс 2 – компьютеры, поддерживающие запуск так называемых *UEFI*-приложений и имеющие режим *CSM*;

4) класс 3 – компьютеры с поддержкой *UEFI*-приложений, но без поддержки *CSM*.

Большинство современных компьютеров относится к классу 2. При этом иногда *UEFI*-приложения и *CSM* реализуются как два взаимоисключающих режима, а иногда возможно совместное использование и *UEFI*-приложений, и модуля совместимости.

Компьютеры класса 3 – преимущественно либо модели со специфической маркетинговой политикой (например, *Microsoft Surface Pro*), либо бюджетные модели. В перспективе *Intel* планирует перейти к производству исключительно продуктов класса 3, однако с учётом текущего положения дел в этом сегменте рынка потребность в *CSM* будет сохраняться ещё довольно продолжительное время.

UEFI нередко подвергается критике. Основная претензия заключается в том, что «новая *BIOS*» оказалась значительно сложнее своей предшественницы, не предоставляя при этом существенных преимуществ, которых нельзя было бы получить расширением классических *BIOS*. Также немало нареканий вызвала и технология *Secure Boot*, которая должна была повысить безопасность систем за счёт проверки цифровых подписей, однако на деле создаёт ряд препятствий, связанных со сложностью сопутствующих процедур, для разработки операционных систем (в том числе любительских и учебных) энтузиастами.

Как *BIOS* загружает операционную систему?

Процесс загрузки операционной системы претерпел немало изменений с момента выпуска первых моделей *IBM PC*, однако они касались скорее того, что происходит «за кулисами» функций, предоставляемых *BIOS*, а также расширения набора устройств и способов загрузки операционной системы. Хоть

сколько-нибудь значимые для разработчика операционной системы изменения произошли только при переходе от классических *BIOS* к *UEFI*, но даже в этом случае концептуально всё осталось по-прежнему, просто были сняты некоторые ограничения.

В 1981 году были выпущены первые *IBM PC*, в частности – самая первая их модель *IBM* 5150. Комплектация их по нынешним меркам была весьма скромной: объём оперативной памяти начинался с 16 Кбайт. Доступный объём ОЗУ можно было наращивать за счёт продаваемых отдельно модулей по 16 Кбайт, но их могло быть не больше трёх, что позволяло получать модели с 32 Кбайт, 48 Кбайт и 64 Кбайт памяти. После этого становилось возможным использовать более объёмные модули по 32 или 64 Кбайт, которые устанавливаться в специальные слоты расширения на материнской плате, что не всегда было удобно. Всего ранние модели *IBM* 5150 были ограничены 256 Кбайт ОЗУ.

Рис. 1.2. Компьютер *IBM* 5150

В минимальной комплектации *IBM PC* не имели ни жёстких дисков, ни даже приводов для дискет (*FDD*), а роль операционной системы выполняла входившая в состав их *BIOS* так называемая *Cassette BASIC* – реализация транслятора для языка *Basic*, позволявшая использовать для хранения и загрузки программ магнитофон, подключённый через специальный разъём, и магнитную ленту. Загрузка операционной системы с магнитной ленты не поддерживалась. Для загрузки какой-либо операционной системы требовалось наличие привода для дискет 5,25″ и не менее 32 Кбайт ОЗУ: подключение жёстких дисков, помимо покупки самого накопителя, требовало наличия дополнительного оборудования и было большой роскошью.

Современные компьютеры могут похвастаться и значительно большими объёмами памяти, и возможностью загрузки не только с дискет, но и с жёстких и оптических дисков, с *USB*-флэш-накопителей и даже по сети, однако сами правила загрузки, используемые *BIOS*, существенно изменились только при использовании *UEFI*, но даже в этом случае, по сути, произошла лишь смена декораций. При использовании же *CSM*-модуля или классической *BIOS* загрузка

любым из перечисленных способов осуществляется почти одинаково, примерно так же, как это происходило при загрузке с дискеты в *IBM* 5150, лишь с поправкой на специфику конкретного способа загрузки.

Для того чтобы определить, откуда должна быть загружена операционная система, *BIOS* после выполнения *POST* начинает перебирать все поддерживаемые способы загрузки. Порядок этого перебора, как правило, может быть настроен пользователем с использованием специальной утилиты, входящей в состав *BIOS* (рис. 1.3).

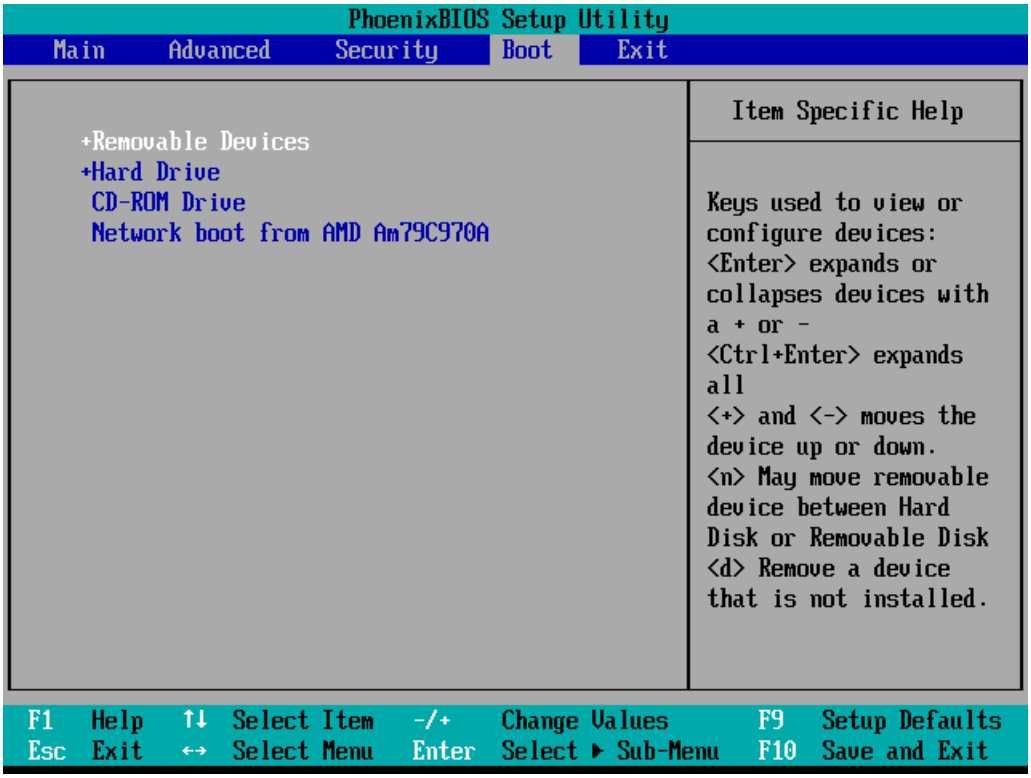

Рис. 1.3. Настройка порядка загрузки в *PhoenixBIOS*

Для каждого из доступных способов *BIOS* считывает в память некоторое количество данных. Если в них присутствует специальная сигнатура, источник считается загрузочным и управление передаётся на начало загруженных в память данных. Конкретные условия и правила незначительно варьируются в зависимости от способа загрузки.

Пример 1.1. «*Hello, world!*»

Освоение любой новой области в программировании принято начинать с написания простейшей программы, которая выводит на экран сообщение «*Hello, world!*». Разработаем фрагмент кода, который сможет вывести такое сообщение, будучи запущенным в ходе поиска *BIOS* источника загрузки.

Обычно задачей кода, записанного в загрузочном секторе, является загрузка ядра операционной системы и других её компонентов, а сам этот код называют соответственно загрузчиком. Но для начала, прежде чем браться

непосредственно за проблему считывания данных с диска, полезно «осмотреться» и разобраться, в каком окружении будет выполняться загрузчик.

В качестве источника загрузки будем использовать дискету. Несмотря на то что современные компьютеры редко комплектуются *FDD*, осуществить загрузку с дискеты можно при помощи виртуальных машин и эмуляторов. Кроме того, существуют *USB*-приводы для дискет, которые также можно использовать для этих целей, если такая возможность поддерживается *BIOS*.

Минимальная единица данных, которую можно считать или записать при работе с диском, называется сектором. Изначально это понятие применялось к дискетам и жёстким дискам, где данные записывались на круглые пластины путём намагничивания, позже оно было применено к круглым оптическим дискам, а затем стало использоваться и в отношении тех накопителей, для которых понятие сектора не имеет геометрического смысла.

Размеры секторов со временем претерпевали некоторые изменения. Так, например, у 8-дюймовых дискет, которые к моменту выхода первого *IBM PC* уже были практически вытеснены более новыми и компактными 5-дюймовыми, встречались размеры секторов и в 128, и в 256, и даже в 1024 байта. В более поздних моделях дискет (в том числе в ровесницах *IBM PC* – дискетах формата 3,5″) тоже наблюдалось разнообразие, но для *IBM*-совместимых компьютеров в итоге устоялся размер в 512 байт на сектор, который впоследствии был использован и для жёстких дисков.

При загрузке с дискеты *BIOS* считывает самый первый её сектор в память по физическому адресу `7C00h`. Поскольку для секторов принято начинать нумерацию с нуля, этот сектор ещё называется нулевым, а исходя из его назначения – быть промежуточным этапом в загрузке операционной системы – его также называют загрузочным сектором. Для того чтобы дискета считалась загрузочной, нужно, чтобы байты со смещениями 510 и 511 в этом нулевом секторе содержали значения `55h` и `AAh` соответственно.

Если дискета признана загрузочной, происходит передача управления на начало считанного в память загрузочного сектора. О состоянии процессора в этот момент известно немного. В частности, с уверенностью можно говорить о выполнении только нескольких условий, перечисленных в табл. 1.1.

Таблица 1.1

Состояние системы в момент передачи управления загрузочному сектору дискеты

Характеристика	**Состояние**
Режим процессора	Реальный режим
Регистр `DL`	Содержит номер диска, с которого выполнена загрузка
Регистры `CS:IP`	Задают физический адрес `7C00h`

Первое, на что следует обратить внимание, – это информация о значениях регистров `CS:IP`. Ввиду особенностей адресации в реальном режиме физический адрес `7C00h` может быть задан множеством способов. Различные *BIOS* поразному подходят к решению этого вопроса: одни передают управление со значениями `0000:7C00` в этих регистрах, другие – со значениями `07C0:0000`. Не исключено, что существуют и такие, которые используют ещё более экзотические комбинации. Таким образом, информацию из таблицы следует воспринимать буквально: `CS:IP` задают физический адрес `7C00h`, но каким именно способом – не уточняется.

Вторая проблема заключается в том, что отсутствуют какие-либо гарантии того, что стек готов к использованию: значения регистров `SS:SP` не уточняются, а их настройка на участок памяти, который можно использовать в качестве стековой памяти, не гарантируется. Разумеется, до запуска загрузчика какой-то код выполнялся и наверняка использовал стек, но сколько именно места доступно – неизвестно. Фактически это означает, что перед использованием любых инструкций, работающих со стеком, следует самостоятельно выбрать место в памяти для его размещения. В противном случае эти инструкции будут работать с заранее не известным участком памяти, запись в который, возможно, нежелательна.

Ещё одна проблема, которая может существенно усложнить разработку, – существование версий *BIOS*, которые оставляют процессор в защищённом режиме. Впрочем, такое поведение следует считать скорее ошибочным, а количество таких компьютеров стремится к нулю, поэтому имеет смысл либо отказываться от их поддержки, либо разрабатывать для них отдельную версию загрузчика.

Наконец, следует помнить, что в момент передачи управления загрузчику операционная система отсутствует. Это означает, что никаких функций никакой операционной системы использовать не получится. В частности, нет функций *MS-DOS* для работы с файлами, для ввода текста, для получения даты и времени и решения многих других задач – все их предстоит реализовать разработчику операционной системы. Но есть и хорошие новости: *BIOS* никуда не делась и предоставляет целый ряд готовых функций, на которые можно опереться на начальном этапе или даже на протяжении всего времени работы операционной системы. Доступ к этим

функциям предоставляется посредством механизма обработки прерываний. В общем случае соглашение вызова следующее:

– в регистр `AH` помещается номер функции;

– в остальные регистры (в зависимости от того, какая функция вызывается) помещаются значения параметров;

– вызов функции осуществляется генерацией программного прерывания с определённым номером (такое прерывание ещё называется сервисом) с помощью инструкции `int` (табл. 1.2).

Таблица 1.2

Назначение основных сервисов *BIOS*

Номер сервиса	**Предоставляемые возможности**
`10h`	Работа с видеоадаптером
`11h`	Получение информации об оборудовании
`12h`	Получение информации об объёме оперативной памяти
`13h`	Работа с дисками
`14h`	Работа с последовательными портами (*serial ports*)
`15h`	Вспомогательные функции
`16h`	Работа с клавиатурой
`17h`	Работа с параллельными портами (*parallel ports*)
`18h`	Обработка ситуации отсутствия загрузочного накопителя
`19h`	Поиск источника загрузки
`1Ah`	Работа с датой и временем

Для вывода текста на экран будем использовать функцию `int 10h/0Eh`, которая позволяет отобразить заданный символ с автоматическим перемещением курсора на следующую позицию.

Функция: AH = 0Eh
Вывод в режиме телетайпа на активную страницу
Отображает заданный символ в текущей позиции курсора на активную видеостраницу и соответствующим образом перемещает курсор (при необходимости – с прокруткой).

Исходные данные:	AH = 0Eh
	AL = код выводимого символа
	BL = цвет текста (только для графических режимов)
	BH = номер активной страницы
Результаты:	нет

В общем случае функции *BIOS* гарантируют, что значения всех регистров, кроме используемых для возврата значений и AX, останутся неизменными после вызова. Но для функций, предоставляемых сервисом 10h, такая гарантия предоставляется только для регистров BX, CX и DX, а значения других регистров общего назначения (особенно SI и DI) могут быть изменены произвольно. Это следует иметь в виду при написании кода загрузчика.

Для удобства договоримся все смещения отсчитывать от сегмента 0 – это удобно, т. к. смещения относительно этого сегмента совпадают с физическими адресами. В этом случае наш «загрузчик» в памяти будет начинаться по смещению 7C00h, на что мы укажем ассемблеру директивой org, а заодно попросим, чтобы создаваемый в результате ассемблирования файл имел расширение .img:

```
        format binary as 'img'
        org $7C00
```

Первым делом проинициализируем значения сегментных регистров, используемых при работе с данными:

```
EntryPoint:
        xor     ax, ax
        mov     ds, ax
        mov     es, ax
```

Следующее, что желательно сделать, – перенастроить стек таким образом, чтобы гарантировать, что оставшегося в нём места хватит для целей загрузчика. Для этого необходимо записать в пару регистров SS:SP значения, соответствующие адресу дна стека. В момент передачи управления загрузчику память используется, как показано в табл. 1.3.

Таблица 1.3

Примерная карта памяти в момент передачи управления загрузчику

Диапазон физических адресов	**Размер**	**Использование**	**Размещение**
00000h–003FFh	1 Кбайт	Таблица векторов прерываний	ОЗУ
00400h–004FFh	256 байт	Область данных *BIOS* (*BDA, BIOS Data Area*)	
00500h–07BFFh	≈ 29 Кбайт	Свободно (кроме нескольких байт в начале)	
07C00h–07DFFh	512 байт	Загрузочный сектор (считанный в память)	
07E00h–07FFFh	512 байт	Свободно	
08000h–7FFFFh	480 Кбайт	Свободно	ОЗУ
80000h–9FFFFh	128 Кбайт	Расширенная область данных *BIOS* (*EBDA, Extended BIOS Data Area*)	
A0000h–BFFFFh	128 Кбайт	Видеопамять	Другие устройства
C0000h–C7FFFh	32 Кбайт	*BIOS* видеоадаптера	
C8000h–EFFFFh	160 Кбайт	Расширения *BIOS*	
F0000h–FFFFFh	64 Кбайт	*BIOS* материнской платы	

Следует понимать, что в таблице приведены физические адреса, которые процессор подаёт на северный мост. Далеко не все они соответствуют оперативной памяти. В частности, например, для модификации *IBM PC* с 32 Кбайт ОЗУ оперативной памяти будут соответствовать только адреса от 0000h до 8000h, а результат обращения ко всем прочим областям, обозначенным как оперативная память, не определён и зависит от реализации задействованных в работе с памятью устройств (в том числе и северного моста). Если планируется разработка операционной системы, способной работать даже на компьютерах с малыми объёмами памяти, подобных ситуаций, безусловно, следует избегать.

Для размещения стека можно использовать любое свободное место в памяти, но для того, чтобы обеспечить поддержку даже компьютеров с минимальным объёмом оперативной памяти (32 Кбайт), отведём для него 512 байт сразу за загрузчиком. Этого должно быть достаточно для того, чтобы обеспечить несколько вложенных вызовов подпрограмм с сохранением значений всех регистров общего назначения. Заодно такое размещение позволит в дальнейшем рассматривать всю память, использованную кодом из загрузочного сектора (512 байт для самого загрузчика и 512 байт для его стека), как один блок памяти. В случае настоящего загрузчика после выполнения им своей работы и пере-

дачи управления дальше эту память можно будет снова считать свободной и использовать для других нужд.

```
        mov     ss, ax
        mov     sp, $8000
```

При перенастройке стека важно помнить о существовании аппаратных прерываний, которые могут возникать в произвольные моменты времени. В архитектуре *Intel IA*-32 прерывания возникают только между инструкциями, т. е. когда одна из инструкций уже выполнена, а другая выполняться ещё не начала. Но перенастройка стека заключается в изменении значений двух разных регистров – `SS` и `SP`, – что в нашем случае требует использования двух отдельных инструкций. Если вызов обработчика прерывания произойдёт в тот момент, когда значение одного из регистров будет уже обновлено, а другого – ещё нет, будет использовано некорректное значение указателя на вершину стека.

К счастью, процессор архитектуры *IA*-32 гарантирует, что после записи значения в регистр `SS` инструкциями `mov` или `pop` обработка прерываний не будет производиться, пока не завершится выполнение ещё одной, следующей инструкции. Таким образом, для приведённого фрагмента кода необходимая нам атомарность его выполнения гарантируется. Можно было бы выполнить перенастройку в одну инструкцию `lss`, однако она появилась только в процессоре *Intel* 80386, а значит, её использование может быть нежелательным, если планируется поддержка более ранних моделей.

После выполнения приведённых выше инструкций в три сегментных регистра из четырёх – `DS`, `ES` и `SS` – записаны нули. Для удобства адресации необходимо обнулить также и регистр `CS`. Сделать это можно с помощью дальнего прыжка:

```
        jmp     $0000:RealEntryPoint

RealEntryPoint:
```

Метка `RealEntryPoint` располагается непосредственно за инструкцией безусловного перехода. Смещение метки ассемблер вычисляет в соответствии со значением, указанным ранее в директиве `org`, а значит – относительно сегмента 0. Поэтому, выполняя прыжок так, как показано в приведённом фрагменте кода, мы попадаем именно туда, куда и планировали, но при этом обнуляем регистр `CS`. До этого момента была вероятность, что в `CS` записано ненулевое значение: в этом случае смещения меток, рассчитанные ассемблером, не соответствовали бы реальному положению дел. Именно по этой причине ни одна из инструкций до этого меток не использовала. Теперь же значение `CS` известно, а значит, далее рассчитанные ассемблером смещения будут правильными.

Теперь можно перейти непосредственно к выводу сообщения. Для удобства будем использовать *Pascal*-строки. В этом случае вывести строку на экран можно будет следующим фрагментом кода:

```
        mov     si, strHello
        lodsb
        movzx   cx, al
        xor     bx, bx
.WriteLoop:
        mov     ah, $0E
        lodsb
        push    si
        int     10h
        pop     si
        loop    .WriteLoop
```

Данный код использует тот факт, что функции `int 10h` не изменяют значений регистров `BX`, `CX` и `DX`. Остальные регистры (в т. ч. регистр `SI`) могут быть изменены. Объявление строки в этом случае будет выглядеть так:

```
strHello        db      13, 'Hello, world!'
```

На практике подобные объявления удобно реализовать, используя макросы и возможности ассемблера по автоматическому вычислению длины строки. Пример реализации подобного макроса можно найти в модуле `Macros\Strings.inc` в исходных кодах ***«OSE!»***.

Следующий вопрос, который нужно решить, – это завершение работы загрузчика. В обычной программе для этого достаточно было бы вызвать специальную функцию операционной системы или передать ей управление обычной инструкцией `ret` (как, например, в *COM*-программах для *MS-DOS*). Однако в случае загрузчика никакой операционной системы нет, а значит, передавать управление по сути некуда. Традиционно для подобного учебного примера предлагается четыре варианта остановки дальнейшего выполнения кода загрузочного сектора. Один из них заключается в том, чтобы организовать бесконечный цикл:

```
        jmp     $
```

Недостатком этого способа является то, что процессор фактически продолжает выполнять одну и ту же инструкцию перехода до тех пор, пока не будет произведена перезагрузка компьютера (или его отключение). Более «экологичным» является второй способ:

```
        cli
        hlt
```

Инструкция `cli` отключает обработку процессором аппаратных прерываний, а инструкция `hlt` останавливает работу процессора до возникновения аппаратного прерывания. Вместе эти две инструкции приводят к полной останов-

ке вычислений: процессор перестаёт выполнять инструкции, а единственный способ возобновить их выполнение в этом случае – перезагрузка компьютера. В результате данный способ характеризуется меньшим энергопотреблением.

Третий способ, которым можно завершить работу нашего «загрузчика» – передать управление *BIOS*. Сервис `int 18h` вызывается *BIOS*, когда ей не удаётся найти загрузочный накопитель, однако его может сгенерировать и сам загрузчик. В ранних моделях *IBM PC*-совместимых компьютеров при этом происходил запуск уже упоминавшегося ранее *Cassette BASIC*. В современных *BIOS* за этим прерыванием вместо целого интерпретатора языка *Basic* скрывается просто вывод сообщения об отсутствии загрузочного накопителя и последующее ожидание.

Наконец четвёртый способ заключается в использовании прерывания `int 19h`. Это прерывание сама *BIOS* генерирует для того, чтобы осуществить поиск загрузочного накопителя. Вызов соответствующего сервиса приводит к повторному запуску этого процесса. Можно увидеть использование этого прерывания в действии, если в настройках *BIOS* сделать загрузку с дискет приоритетнее, чем с других накопителей, и оставить обычную дискету с данными в дисководе во время перезагрузки: код её загрузочного сектора выведет сообщение о том, что дискета не содержит операционной системы, которую можно было бы запустить, и инициирует повторный поиск загрузочного накопителя с помощью `int 19h` после нажатия любой клавиши (рис. 1.4).

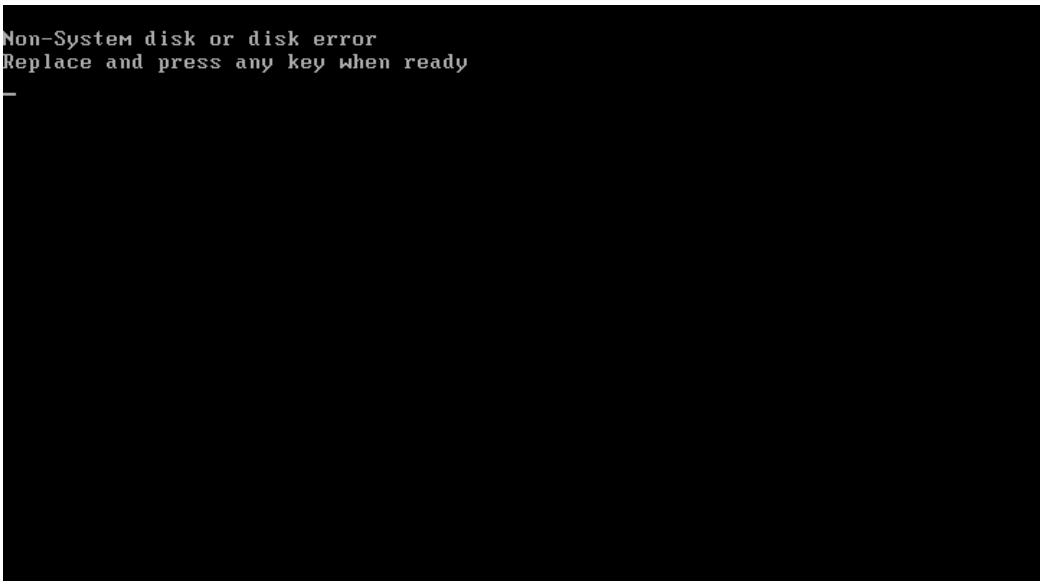

Рис. 1.4. Использование `int 19h` загрузочным сектором дискеты

Для целей этого примера воспользуемся способом с остановкой вычислений, однако в дальнейшем будем иметь в виду и альтернативные решения.

Последнее, о чём следует позаботиться, – специальная сигнатура, по которой *BIOS* определит, что дискета является загрузочной. Для этого необходимо дополнить генерируемый ассемблером файл до размера в 512 байт, причём последние два байта должны будут иметь значения `55h` и `AAh`:

```
db 510 - ($ - $$) dup(0)
db $55, $AA
```

Полученный в результате компиляции файл можно указать в настройках виртуальной машины или эмулятора в качестве образа дискеты. Полный листинг этого примера будет следующим:

```
        format binary as 'img'
        org $7C00

EntryPoint:
        xor     ax, ax
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, $8000
        jmp     $0000:RealEntryPoint

RealEntryPoint:
        mov     si, strHello
        lodsb
        movzx   cx, al
        xor     bx, bx
.WriteLoop:
        mov     ah, $0E
        lodsb
        push    si
        int     10h
        pop     si
        loop    .WriteLoop

        cli
        hlt

strHello        db      13, 'Hello, world!'

db 510 - ($ - $$) dup(0)
db $55, $AA
```

Что такое *CHS* и *LBA*?

Для того чтобы загрузчик действительно что-нибудь загружал, необходимо осуществить чтение с загрузочного накопителя, в нашем случае – дискеты. Разумеется, для этого есть соответствующие функции *BIOS*, однако для их использования следует разобраться с тем, как происходит адресация секторов при использовании подобных накопителей.

Основной конструктивный элемент, скрывающийся внутри защитного корпуса дискеты, – это гибкий пластиковый диск, покрытый специальным ферромагнитным слоем и позволяющий записывать информацию путём намагничивания определённых участков. Для чтения и записи данных с такого диска используются так называемые магнитные головки, которые располагаются над

поверхностью диска и могут перемещаться с использованием специального механизма позиционирования головок (рис. 1.5).

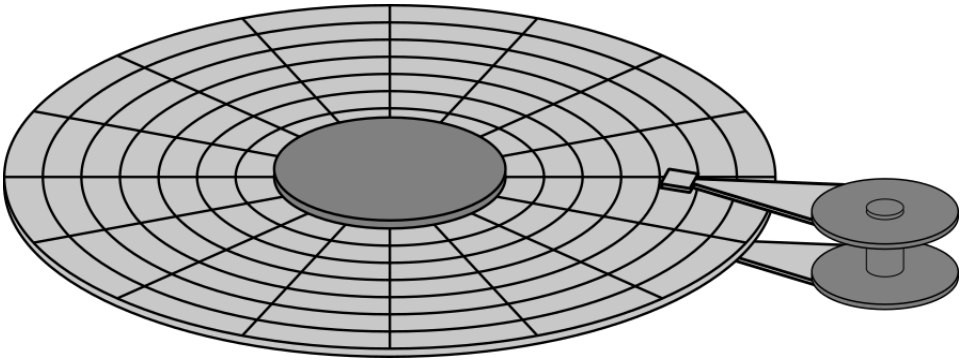

Рис. 1.5. Принцип работы накопителей на магнитных дисках

Жёсткие диски (*HDD*) устроены аналогичным образом и отличаются лишь тем, что, во-первых, состоят из нескольких таких магнитных дисков, а во-вторых, блок магнитных головок является частью самого *HDD*.

Как было отмечено ранее, минимальной единицей обмена данными с различными накопителями является сектор. Однако для того, чтобы считать или записать данные, нужно иметь способ указать, какой именно сектор использовать. Один из таких способов адресации получил название *CHS*.

Идея *CHS*-адресации заключается в том, чтобы выбирать требуемый сектор, задавая три числа: номер цилиндра (***C****ylinder*), номер магнитной головки (***H****ead*) и собственно номер сектора (***S****ector*) на выбранной первыми двумя числами дорожке. Цилиндром называется группа секторов, находящихся на одинаковом расстоянии от центра диска. Причём, если диск двухсторонний, к цилиндру относятся сектора обеих сторон (рис. 1.6).

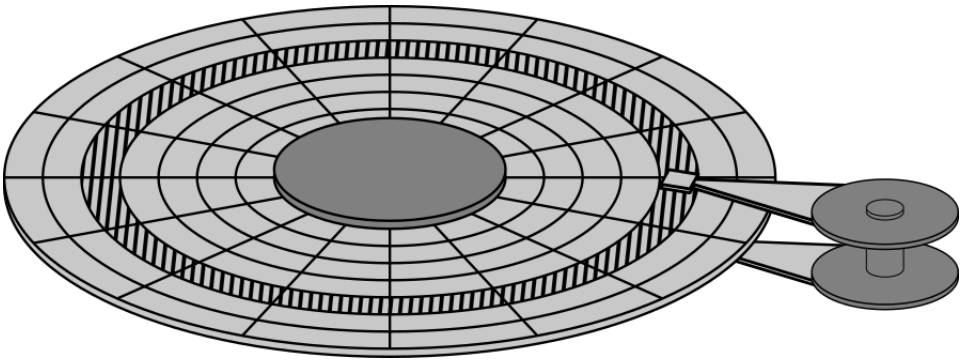

Рис. 1.6. Выбор цилиндра магнитного диска

В *CHS*-адресации номер цилиндра является первым из трёх чисел, используемых для идентификации сектора. Второе число задаёт номер магнитной головки, под которой находится требуемый сектор (рис. 1.7).

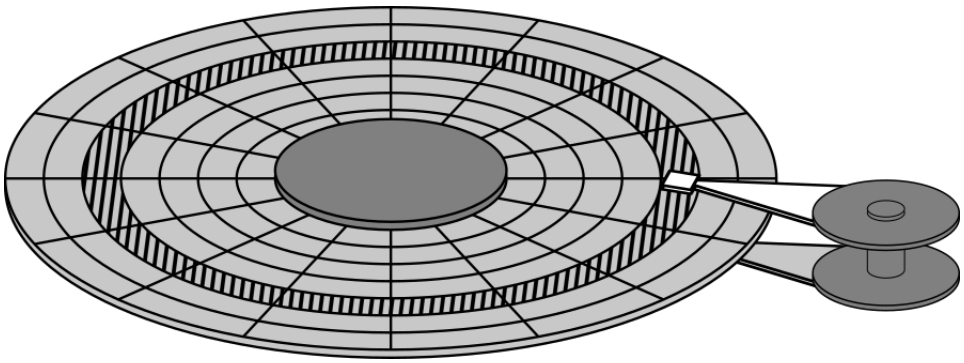

Рис. 1.7. Выбор головки магнитного диска

Набор секторов, задаваемых двумя числами – номерами цилиндра и головки, – называется дорожкой (*track*) и включает в себя все сектора, расположенные на одинаковом расстоянии от центра с одной стороны диска. Теперь для того чтобы выбрать конкретный сектор, нужно задать его номер в пределах дорожки – по сути угол, на который нужно повернуть диск, чтобы головка оказалась над этим сектором (рис. 1.8).

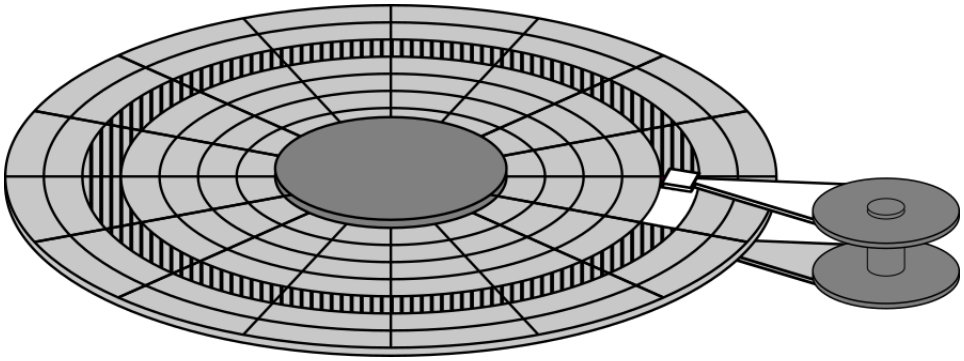

Рис. 1.8. Выбор сектора магнитного диска

Нетрудно заметить, что при всей логичности такой схемы нумерации она всё равно оказывается крайне неудобной. Дело в том, что геометрия диска – информация о том, сколько в нём цилиндров, головок и секторов – отличается от диска к диску. Это означает, что после сектора (0, 0, 24) может идти сектор (0, 0, 25), а может и сектор (0, 1, 1). По сути сектор задаётся трёхзначным числом (*C*, *H*, *S*), вот только основание системы счисления оказывается плавающим: между *C* и *H* используется один множитель, а между *H* и *S* – другой. Более того, для разных дисков сами эти множители тоже могут быть разными. Стоит также обратить внимание, что цилиндры и головки нумеруются с нуля, а сектора – начиная с 1. Назвать такую арифметику простой вряд ли возможно.

Для решения этой проблемы была предложена другая схема адресации – *LBA*, *Linear Block Address*. Её идея заключается в том, чтобы пронумеровать все сектора одним-единственным числом – порядковым номером сектора на диске. В этом случае весь диск представляет собой один большой массив секторов, индексом в котором как раз и будет *LBA*, или номер, сектора.

Но избежать *CHS*-адресации разработчику операционной системы всё равно не удастся. Дело в том, что функции *BIOS* изначально предназначались для работы с дискетами (единственным поддерживаемым на тот момент носителем информации), и лишь спустя некоторое время были расширены и для поддержки жёстких дисков, поэтому сектор для чтения/записи этими функциями нужно задавать именно в формате *CHS*. По всей видимости, преобразование из *LBA* в *CHS* на уровне аппаратуры было неприемлемым на тот момент решением, поэтому эта задача была делегирована программному обеспечению. Отчасти это могло быть связано, например, с тем, что, имея контроль над физическим размещением данных на диске, программное обеспечение могло оптимизировать работу с ним: стараться записывать данные в сектора, расположенные на диске ближе друг к другу, чтобы минимизировать количество механических (а значит, медленных) операций по позиционированию магнитных головок и вращению дисков.

Чуть позже, когда жёсткие диски стали более «умными», был разработан новый набор функций *BIOS* – *BIOS Enhanced Disk Drive Interface*, или просто *EDD*. Однако случилось это только в конце 1997 года, спустя более чем 15 лет после появления первых реализаций *BIOS*. Но эти функции, как правило, доступны только для работы с жёсткими дисками – для работы с дискетами по-прежнему приходится возвращаться к классическим, работающим с *CHS*. Более того, даже в современных *BIOS* при работе с более продвинутыми устройствами для загрузки (например, *USB*-флэш-накопителями) в ряде случае приходится ограничиваться именно *CHS*-адресацией.

Впрочем, при написании загрузчика преобразование, как правило, приходится выполнять только в одну сторону – из *LBA* в *CHS*. Соответствующие формулы достаточно просты, достаточно лишь знать геометрию конкретного накопителя. *BIOS* предоставляет функцию, позволяющую получить эти сведения, правда, только для жёстких дисков. Для дискет эта функция может работать некорректно или не работать вовсе, но проблема частично решается тем, что количество форматов дискет сравнительно невелико, а их параметры известны заранее.

Для того чтобы преобразовать *LBA* сектора в числа для *CHS*-адресации, нужно помнить, что *CHS*-адрес – это трёхзначное число, причём старший разряд – *C*, затем идёт *H* и наконец младший разряд – *S*.

```
S    = (LBA mod SectorsPerTrack) + 1
Temp = (LBA div SectorsPerTrack)
H    = Temp mod NumHeads
C    = Temp div NumHeads
```

Обратите внимание: нумерация секторов начинается с 1, поэтому к остатку от деления при вычислении номера сектора нужно прибавить единицу. Для выполнения этого преобразования фактически достаточно знать два параметра геометрии диска: количество секторов в одной дорожке (*SectorsPerTrack*) и количество головок (*NumHeads*). Третья характеристика – количество цилиндров – будет нужна только для проверки, существует ли сектор с заданным *LBA* на этом диске.

В табл. 1.4 справочно приведены характеристики некоторых популярных форматов дискет (размер сектора во всех случаях составляет 512 байт).

Таблица 1.4

Характеристики некоторых форматов дискет

Формат и объём	Кол-во цилиндров	Кол-во головок	Кол-во секторов в одной дорожке	Общее кол-во секторов
5,25″; 160 Кбайт	40	1	8	320
5,25″; 180 Кбайт	40	1	9	360
5,25″; 320 Кбайт	40	2	8	640
5,25″; 360 Кбайт	40	2	9	720
3,5″; 720 Кбайт	80	2	9	1440
5,25″; 1,2 Мбайт	80	2	15	2400
3,5″; 1,44 Мбайт	80	2	18	2880
3,5″; 2,88 Мбайт	80	2	36	5760

Наибольшую популярность в последние годы активного использования дискет приобрёл формат 3,5″; 1,44 Мбайт, поэтому будем ориентироваться на него. К тому же этот формат гарантированно поддерживается большинством виртуальных машин и эмуляторов.

Пример 1.2. Простейший загрузчик

Размеры загрузочного сектора сравнительно невелики: обычно это всего лишь 512 байт, причём, как было показано ранее, последние два из них должны иметь строго определённые значения – `55h` и `AAh`. Разумеется, в такой объём крайне сложно поместить всю операционную систему, поэтому обычно туда помещается лишь загрузчик – код, который обеспечивает загрузку в память ядра операционной системы и, возможно, других её компонентов.

Иногда для решения этой задачи места в загрузочном секторе оказывается недостаточно. В этом случае загрузка реализуется в два этапа:

– первичный загрузчик считывает в память код вторичного загрузчика и передаёт ему управление;

– вторичный загрузчик производит загрузку собственно ядра операционной системы.

Идея загрузки в два этапа основана на том, что размером сектора ограничен только первичный загрузчик. Количество же данных, которые он может загрузить в память, ограничено только тем, сколько места займёт код, который эту загрузку выполняет, но не объёмом этих данных. Таким образом, использование загрузки в

два этапа позволяет обойти ограничения, связанные с размером сектора, правда, ценой некоторого замедления запуска операционной системы.

Выбор того или иного подхода определяется целым рядом факторов. Во-первых, он зависит от степени сложности поиска загружаемых данных: чем сложнее организовано их хранение на диске, тем больше инструкций потребуется, чтобы описать алгоритм их поиска и загрузки. Во-вторых, влияние может оказывать то, из какого источника происходит загрузка: например, для жёстких дисков обычно доступна загрузка с использованием *EDD*-функций, что позволяет сэкономить место на преобразовании из *CHS* в *LBA*. Третий фактор – то, какое аппаратное обеспечение должна поддерживать операционная система: как правило, код для старых моделей процессоров получается более объёмным, т. к. в новых моделях имеются инструкции, позволяющие выполнять те же действия, но с помощью менее объёмного машинного кода. Наконец, немалую роль в принятии решения играют и навыки разработчика: как правило, код начинающего программиста менее оптимален с точки зрения соотношения размера и количества выполняемых полезных действий.

Разработаем простейший загрузчик. В качестве загрузочного накопителя будем использовать дискету формата 3,5″; 1,44 Мбайт и договоримся, что её пространство будет задействовано, как показано в табл. 1.5.

Таблица 1.5

Размещение данных на дискете для учебного примера

LBA	**Кол-во секторов**	**Содержимое**
0	1	Загрузочный сектор
1	*N*	Загружаемый код («ядро»)
N + 1	…	Неиспользуемое (свободное) пространство

В качестве «ядра» (конечно, пока ещё не настоящего) операционной системы будет выступать код из предыдущего примера, выводящий сообщение «*Hello, world!*». Для большей наглядности разделим код на три отдельно компилируемых файла:

– `BootSector.asm` будет содержать код загрузочного сектора и компилироваться в файл `BootSector.bin`;

– `Kernel.asm` будет содержать код «ядра» и компилироваться в файл `Kernel.bin`;

– `FloppyDisk.asm` будет содержать загрузочный сектор и «ядро» и собирать их в образ загрузочной дискеты – файл `FloppyDisk.img`.

Прежде всего выберем место в памяти, куда будет происходить загрузка «ядра». Из табл. 1.3 следует, что довольно большой свободный участок памяти начинается по физическому адресу `500h`. Там же отмечено, что некоторое количество байт в начале этого участка всё же может быть занято *BIOS*, которая использует их для своих нужд как продолжение *BIOS Data Area* (*BDA*). Точно

сказать, сколько именно будет таких «проблемных» байт, сложно, поэтому будем загружать «ядро» начиная с физического адреса `600h`, с запасом. До следующей занятой области памяти – кода загрузчика (`7C00h–7E00h`) – будет целых 29 Кбайт, и этого более чем достаточно для наших целей.

```
        org $0600

        mov     si, strHello
        lodsb
        movzx   cx, al
        xor     bx, bx
.WriteLoop:
        mov     ah, $0E
        lodsb
        push    si
        int     10h
        pop     si
        loop    .WriteLoop

        cli
        hlt

strHello        db      13, 'Hello, world!'
```

Код для вывода сообщения существенных изменений не претерпел, единственное отличие по сравнению с предыдущим примером – теперь он будет загружаться в память по другому адресу, что отражено в параметре директивы `org`. Директива `format` не используется, поскольку поведение ассемблера при её отсутствии соответствует нашим целям, а именно – эквивалентно записи:

```
        format binary as 'bin'
```

Основные изменения коснутся кода загрузочного сектора. Работу с диском, с которого происходит загрузка, следует начать с переинициализации дискового контроллера, поскольку состояние, в котором он будет оставлен *BIOS*, неизвестно. Кроме того, выполнение этой процедуры необходимо, если при попытке чтения секторов возникают ошибки, поэтому выполнить её до начала работы с диском не будет лишним. Для этих целей предусмотрена функция `int 10h/00h`.

Функция: `AH = 00h`
Переинициализация дискового контроллера
Переинициализирует контроллер и привод заданного диска (головки перемещаются к нулевой дорожке). Если при вызове других функций работа с дисками возникает ошибка, необходимо вызвать эту функцию, а затем повторить проблемный вызов.

Исходные данные:	`AH` = `00h`
	`DL` = номер диска
Результаты:	`AH` = 0, если функция выполнилась успешно, иначе – код ошибки
	`CF` = 0, если функция выполнилась успешно, иначе – 1

В случае возникновения ошибки будем просто останавливать работу процессора комбинацией инструкций `cli` и `hlt`. Код загрузочного сектора будет примерно таким:

```
        org $7C00

EntryPoint:
        xor     ax, ax
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, $8000
        jmp     $0000:RealEntryPoint

RealEntryPoint:
        mov     [bDrvNum], dl

        int     13h
        jc      .Error

        ; Здесь будет чтение секторов с диска

.Error:
        cli
        hlt

bDrvNum         db      ?

db 510 - ($ - $$) dup(0)
db $55, $AA
```

Обратите внимание, что специальной подготовки параметров для вызова функции `int 10h/00h` не потребовалось, т. к. в регистре `DL` номер диска, с которого производится загрузка, уже записан *BIOS*, а в регистре `AH` нулевое значение осталось после обнуления сегментных регистров.

Для чтения данных с диска будем использовать функцию `int 13h/02h`.

Функция: `AH = 02h`
Посекторное чтение с диска (для дискет)
Считывает заданное количество секторов с диска.

Исходные данные:
`AH` = `02h`
`AL` = количество секторов, от 1 до 18 в зависимости от вида носителя
`CH` = номер трека (цилиндра), от 0 до 79 в зависимости от вида носителя
`CL` = номер сектора, от 1 до 18 в зависимости от вида носителя
`DH` = номер головки, от 0 до 1
`DL` = номер диска
`ES:BX` = указатель на буфер, в который должны быть прочитаны данные

Результаты:
`AL` = количество прочитанных секторов
`AH` = 0, если функция выполнилась успешно, иначе – код ошибки
`CF` = 0, если функция выполнилась успешно, иначе – 1

На количество считываемых секторов накладывается ряд ограничений:

– количество должно быть строго меньше 128;

– считываемые данные не должны пересекать границ сегмента, номер которого задан регистром `ES`;

– операция чтения не должна пересекать границы цилиндра, т. е. не должна охватывать сектора, относящиеся к разным цилиндрам.

В некоторых *BIOS* ошибка чтения может быть вызвана тем, что в момент запроса на чтение двигатель, вращающий диск, был отключён, а *BIOS* не ожидает достижения требуемой скорости вращения. В этом случае необходимо произвести переинициализацию устройства (функцией `int 13h/00h`) и повторить попытку чтения три раза, чтобы убедиться, что ошибка неустранима.

Ограничения, накладываемые этой функцией на диапазон считываемых секторов, довольно существенны. Больше всего неудобств может доставить запрет на пересечение границ цилиндров, т. к. это означает, что при необходимости чтения большого количества секторов придётся выполнять более одной операций чтения, причём при разбиении на несколько вызовов нужно учитывать геометрию конкретного носителя. С учётом ограниченности размеров загрузочного сектора самым простым решением будет использование этой функции для чтения секторов по одному за вызов.

При преобразовании номеров секторов из *LBA* в *CHS* придётся выполнять целочисленное деление, а предназначенная для этого инструкция `div` в качестве единственного операнда-делителя может принимать либо регистр, либо значение в памяти, поэтому отведём в загрузочном секторе место для хранения величин, на которые будет производиться деление, – количества секторов в дорожке и количества головок. Удобнее всего это будет сделать между дальним прыжком к метке `RealEntryPoint` и самой меткой. Туда же запишем и количество секторов, которые должен будет прочитать загрузчик:

```
        ...
        jmp     $0000:RealEntryPoint

wNumSectors     dw      1
wNumHeads       dw      2
wSecPerTrack    dw      18

RealEntryPoint:
        ...
```

Это не единственное возможное решение, но, вероятно, одно из наиболее удачных. Преимущество перед размещением этих данных рядом с объявлением `bDrvNum`, ближе к концу кода загрузчика, заключается в том, что при необходимости перекомпиляции его для дискеты с другой геометрией или для «ядра» другого размера удобнее искать их ближе к началу исходного кода. По той же причине такое решение практичнее, чем прописывание конкретных величин непосредственно в коде загрузки. Более того, поскольку предшествующий код вряд ли будет изменяться, появляется возможность перенастройки загрузчика без перекомпиляции: достаточно в уже скомпилированном загрузчике изменить значения байтов, расположение которых заранее известно.

Приступим к написанию кода, отвечающего непосредственно за загрузку. Для начала организуем цикл чтения:

```
        mov     cx, [wNumSectors]
        jcxz    .Error
        mov     si, 1
        mov     bx, $0600
.ReadLoop:
        push    cx

        ; Чтение сектора с LBA = SI

        pop     cx
        inc     si
        add     bx, $0200
        loop    .ReadLoop
```

Регистр `BX` при вызове функции чтения используется для адресации памяти, куда выполняется считывание. В примере предполагается, что размер сектора всегда равен 512 байтам. Поддержка произвольного размера сектора может быть выполнена по аналогии с поддержкой различных геометрий дисков.

Есть два регистра общего назначения, которые не будут задействованы ни при целочисленном делении, ни при вызове функции чтения, – это `SI` и `DI`. Для того чтобы хранить *LBA* (т. е. номер) очередного считываемого сектора, задействуем регистр `SI`. На каждой итерации нужно будет преобразовывать это значение в *CHS* и размещать номера цилиндра, головки и сектора в соответствующие регистры:

```
        mov     ax, si
        xor     dx, dx
        div     [wSecPerTrack]
        mov     cx, dx
        inc     cx
        xor     dx, dx
        div     [wNumHeads]
        mov     ch, al
        mov     dh, dl
        mov     dl, [bDrvNum]
```

Остаётся только выполнить вызов функции чтения, однако следует помнить, что его завершение с ошибкой не означает, что чтение выполнить невозможно. В разных источниках указывается разное количество попыток, которые следует сделать, прежде чем считать ошибку неустранимой, – от 3 до 5. Ниже продемонстрирован один из способов реализовать чтение с учётом этих особенностей.

```
        mov     di, 5
.Retry:
        mov     ax, $0201
        int     13h
        jnc     .Continue
        dec     di
        jz      .Error

        xor     ax, ax
        int     13h
        jc      .Error
        jmp     .Retry

.Continue:
```

Регистр `DI` задействован в качестве счётчика количества попыток. Кроме того, используется тот факт, что функции `int 13h` не изменяют значений регистров общего назначения, за исключением используемых для возврата результатов. Это позволяет избежать повторного преобразования *LBA* в *CHS* перед каждой попыткой.

Если цикл чтения завершается успешно, после его выполнения, начиная с физического адреса `0600h`, будет записано содержимое `wNumSectors` секторов, следующих за загрузочным.

```
        org $7C00

EntryPoint:
        xor     ax, ax
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, $8000
        jmp     $0000:RealEntryPoint

wNumSectors     dw      1
wNumHeads       dw      2
wSecPerTrack    dw      18

RealEntryPoint:
        mov     [bDrvNum], dl

        int     13h
        jc      .Error

        mov     cx, [wNumSectors]
        jcxz    .Error
        mov     si, 1
        mov     bx, $0600
.ReadLoop:
        push    cx

        mov     ax, si
        xor     dx, dx
        div     [wSecPerTrack]
        mov     cx, dx
        inc     cx
        xor     dx, dx
        div     [wNumHeads]
        mov     ch, al
        mov     dh, dl
        mov     dl, [bDrvNum]

        mov     di, 5
.Retry:
        mov     ax, $0201
        int     13h
        jnc     .Continue
        dec     di
        jz      .Error

        xor     ax, ax
        int     13h
        jc      .Error
        jmp     .Retry

.Continue:
        pop     cx
        inc     si
        add     bx, $0200
        loop    .ReadLoop
```

```
        jmp     $0600
.Error:
        cli
        hlt

bDrvNum         db      ?

db 510 - ($ - $$) dup(0)
db $55, $AA
```

Собрать код загрузочного сектора и код «ядра» в образ дискеты можно следующим кодом:

```
        format binary as 'img'

file 'BootSector.bin'
file 'Kernel.bin'

db 2880 * 512 - ($ - $$) dup(0)
```

Пример 1.3. Доработка простейшего загрузчика

Реализация примера 1.2 предполагает, что компиляция файлов `BootSector.asm` и `Kernel.asm` уже произведена. На практике подобное разбиение на три отдельных файла необязательно и даже отчасти неудобно, т. к. для получения образа дискеты требуется отдельно запустить компиляцию для каждого из них.

Самый простой способ устранения этих неудобств – объединить отдельные файлы на уровне исходного кода:

```
        format binary as 'img'

include 'BootSector.asm'
include 'Kernel.asm'

db 2880 * 512 - ($ - $$) dup(0)
```

К сожалению, несмотря на кажущуюся простоту, у этого решения есть недостаток: последняя строка кода, которая должна дополнить скомпилированный файл до размера дискеты, будет работать не так, как ожидается, т. к. символ `$$` отражает смещение, с которого начался текущий блок адресации. Начать новый такой блок могут директивы `org` или `virtual`, и одна из них – `org` – как раз встречается в обоих подключаемых файлах.

Решить эту проблему можно различными способами. Самый простой – задействовать символ `$%`, который доступен при использовании `format binary` и всегда равен размеру сгенерированных к этому моменту байтов скомпилированного файла:

```
        format binary as 'img'

include 'BootSector.asm'
include 'Kernel.asm'

db 2880 * 512 - $% dup(0)
```

Ещё одно неудобство, которое имеет место в примере 1.2, – необходимость самостоятельно рассчитывать размер «ядра» в секторах и обновлять значение, записанное по адресу `wNumSectors`. Чтобы автоматизировать это действие, добавим в конец файла `Kernel.asm` следующую строку:

```
KernelSize = $ - $$
```

Теперь в файле `BootSector.asm` можно изменить объявление данных с меткой `wNumSectors`:

```
wNumSectors     dw      (KernelSize + 511) / 512
```

Записанная после директивы `dw` формула обеспечивает вычисление количества секторов, необходимых для записи «ядра». Проверить работоспособность внесённых изменений можно искусственно увеличив размер «ядра», например, добавив перед объявлением строки «*Hello, world!*» некоторое количество байтов:

```
db 1024 dup(0)
```

На этом этапе полный исходный код, как его видит ассемблер, будет иметь следующий вид:

```
        format binary as 'img'



        ; Содержимое файла BootSector.asm

        org $7C00

EntryPoint:
        xor     ax, ax
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, $8000
        jmp     $0000:RealEntryPoint

wNumSectors     dw      (KernelSize + 511) / 512
wNumHeads       dw      2
wSecPerTrack    dw      18

RealEntryPoint:
        mov     [bDrvNum], dl

        int     13h
        jc      .Error

        mov     cx, [wNumSectors]
        jcxz    .Error
        mov     si, 1
        mov     bx, $0600
.ReadLoop:
        push    cx

        mov     ax, si
        xor     dx, dx
        div     [wSecPerTrack]
        mov     cx, dx
        inc     cx
        xor     dx, dx
        div     [wNumHeads]
        mov     ch, al
        mov     dh, dl
        mov     dl, [bDrvNum]

        mov     di, 5
.Retry:
        mov     ax, $0201
        int     13h
        jnc     .Continue
        dec     di
        jz      .Error

        xor     ax, ax
        int     13h
        jc      .Error
        jmp     .Retry
```

```
.Continue:
        pop     cx
        inc     si
        add     bx, $0200
        loop    .ReadLoop

        jmp     $0600
.Error:
        cli
        hlt

bDrvNum         db      ?

db 510 - ($ - $$) dup(0)
db $55, $AA



        ; Содержимое файла Kernel.asm

        org $0600

        mov     si, strHello
        lodsb
        movzx   cx, al
        xor     bx, bx
.WriteLoop:
        mov     ah, $0E
        lodsb
        push    si
        int     10h
        pop     si
        loop    .WriteLoop

        cli
        hlt

db 1024 dup(0)

strHello        db      13, 'Hello, world!'

KernelSize = $ - $$



db 2880 * 512 - $% dup(0)
```

Подведение итогов

Первый модуль операционной системы, получающий управление, – загрузчик, который размещается в так называемом загрузочном секторе. Его задача – обеспечить чтение с загрузочного диска ядра операционной системы и, возможно, других её компонентов. Решение этой задачи может происходить в один или в два этапа. В первом случае код, записанный в загрузочный сектор,

передаёт управление непосредственно ядру операционной системы, во втором – обеспечивает запуск вторичного загрузчика.

Минимальной единицей обмена данными с дисками является сектор. При загрузке с дискеты для нумерации секторов используется схема *CHS* (*Cylinder, Head, Sector*), которая отражает физическое устройство магнитного диска. При этом у функций *BIOS*, используемых для работы с дискетами, имеется ряд ограничений, которые необходимо учитывать при разработке загрузчика.

Задания

1. Оформите код чтения секторов с диска как отдельную процедуру.

2. Доработайте загрузчик так, чтобы при возникновении ошибки он выводил её код и по нажатию клавиши инициировал повторный поиск загрузочного накопителя.

3. Реализуйте вывод текстового сообщения в загрузчике в виде процедуры и передайте её адрес «ядру», чтобы избежать дублирования кода.

4. Разработайте игру размером до 510 байт и организуйте её запуск из загрузочного сектора дискеты.

5. Добавьте возможность сохранения результатов и/или текущего состояния для игры, разработанной в п. 4.

Шаг 2. Загрузка с поддержкой файловых систем

Одна из главных задач операционной системы – обеспечивать запуск различных программ, в том числе не входящих в её состав. Для этого необходимо иметь механизм, позволяющий записывать данные различного назначения на один носитель информации и обращаться к ним выборочно и по мере необходимости. Таким механизмом являются файловые системы, и об их поддержке стоит задумываться уже на ранних этапах разработки операционной системы.

Зачем загрузчику поддерживать файловые системы?

У простейшего загрузчика, рассмотренного ранее, есть ряд недостатков. Один из них заключается в том, что информацию о размерах и расположении загружаемых данных приходится так или иначе прописывать непосредственно в коде самого загрузчика. Второй недостаток становится очевидным, если попробовать подключить полученный образ загрузочной дискеты к виртуальной машине, на которую установлена одна из «настоящих» операционных систем, или записать его на физический носитель – дискету: работа с такой дискетой как с обычным носителем информации оказывается невозможной (рис. 2.1).

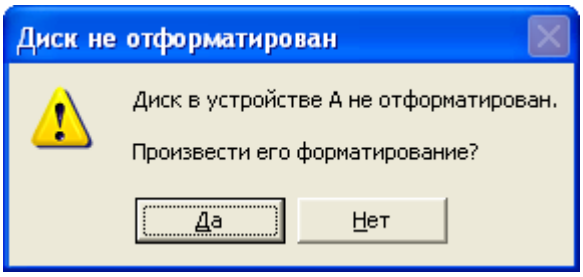

Рис. 2.1. Реакция Проводника *Windows XP* на дискету с простейшим загрузчиком

Было бы весьма удобно, если бы ядро разрабатываемой операционной системы можно было записывать на дискету (или другой загрузочный накопитель) просто копированием файла. Особенно ценна эта возможность при отладке операционной системы на реальном компьютере: посекторная запись на накопитель сопряжена с множеством неудобств. Обычно она выполняется вручную с помощью специального программного обеспечения, что значительно повышает вероятность допущения ошибок по невнимательности.

Всё перечисленное касается также и других компонентов операционной системы, от драйверов и служебных утилит до поставляемого вместе с ней прикладного программного обеспечения (ПО) и документации (или справочной системы): до тех пор, пока операционная система не имеет собственного инструментария для решения типовых задач – текстового редактора, средств разработки программ и т. п., – приходится обходиться инструментами одной из уже существующих ОС. Наличие удобного способа получения доступа к своим наработкам из обеих систем существенно упрощает жизнь разработчику.

Ключом к достижению этой цели является размещение данных на дискете по определённым правилам – в определённом формате. Эти правила, или формат, и называются файловой системой.

Как устроена файловая система *FAT*?

Среди существующих файловых систем особо выделяется *FAT*, отличающаяся своей простотой и в то же время широкой поддержкой различными операционными системами. Строго говоря, *FAT* – это целое семейство файловых систем, объединённых общей идеей того, как распределить дисковое пространство между файлами и служебными структурами данных, однако ввиду значительного сходства получившие наибольшее распространение *FAT*12, *FAT*16 и *FAT*32 уместнее будет считать различными версиями одной и той же файловой системы.

Структура диска, отформатированного в *FAT*, в общем виде представлена на рис. 2.2.

Рис. 2.2. Обобщённая структура *FAT*-раздела

В зависимости от версии файловой системы назначение и содержимое перечисленных блоков (кроме, разумеется, области данных) могут слегка отличаться. Так, например, на дисках, отформатированных в *FAT*12 и *FAT*16, зарезервированная область должна состоять только из загрузочного сектора, а в *FAT*32 она содержит большее количество секторов, но в ней отсутствует явно выделенная область для корневой директории. Тем не менее общая структура сохраняется во всех перечисленных версиях.

Все блоки, кроме области данных, изображённые на рис. 2.2, носят служебный характер и используются для хранения информации о размещении файлов на диске. Сами же файлы размещаются в области данных.

Хранение информации о занятых или доступных для использования секторах из области данных связано с понятием «кластер». Кластер – это группа смежных секторов области данных, которая рассматривается в рамках файловой системы как минимальная единица хранения данных. Другими словами, это некоторое количество идущих подряд секторов, условно объединённых в груп-

пу и всегда используемых вместе. Решение о том, из какого количества секторов будет состоять один кластер, принимается при форматировании диска.

Простейший случай, когда размер кластера равен одному сектору, встречается в основном на небольших накопителях наподобие дискет, но для дисков большего объёма такой подход непрактичен. Во-первых, чем больше количество секторов на диске, тем бóльшие числа приходится использовать для их нумерации, что увеличивает размеры служебных структур данных, сокращая полезное (доступное для хранения данных) дисковое пространство. Во-вторых, крайне неудобно работать с числами, превышающими разрядность процессора: например, для 16-битных процессоров сложности возникают уже при работе с числами, для представления которых требуется больше 16 бит. Следовательно, чем меньше числа, которыми должно оперировать программное обеспечение, тем проще применять файловую систему на более простых устройствах. Наконец, размеры большинства файлов превышают размеры сектора, поэтому имеет смысл сразу выделять место на диске более крупными блоками.

При записи файла на диск ему в соответствии с его размером выделяется некоторое количество кластеров. Кластеры необязательно располагаются на диске последовательно, вместо этого они образуют так называемую цепочку кластеров, в которой для каждого кластера известно, где искать следующий за ним, т. е. хранящий следующую часть данных файла. Такое решение позволяет избежать ситуации, когда на диске достаточно свободных кластеров, но нет непрерывного блока достаточного размера: любой свободный кластер независимо от его положения на диске может быть в случае необходимости задействован для записи данных файла.

Информация о цепочках кластеров хранится отдельно – в таблице размещения файлов (*File Allocation Table, FAT*). Как правило, эта таблица записывается на диск в нескольких экземплярах: это одна из самых активно используемых частей диска, поэтому в случае повреждения носителя полезно иметь её резервную копию. *FAT* представляет собой массив, индексом в котором является номер кластера, а значением по этому индексу – номер следующего кластера в цепочке.

Корневая директория является отправной точкой для перечисления хранящихся на диске файлов. От всех прочих директорий (т. е. каталогов, папок) она отличается только фиксированным расположением на диске (в *FAT*12 и *FAT*16). Директории в *FAT* представляют собой особый вид файлов: вместо каких-либо пользовательских данных в них записана информация об именах и расположении других файлов.

Подробное описание структур данных, используемых в *FAT*, приведено в спецификации *FAT*.

Пример 2.1. Образ пустой *FAT*12-дискеты

Знакомство с принципами работы файловых систем *FAT* проще всего начать с изучения образа пустой дискеты, не содержащей ни одного файла. Создать такой образ можно с помощью специализированных программ наподобие *WinImage* (рис. 2.3).

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   EB 58 90 57 49 4E 49 4D  41 47 45 00 02 01 01 00   ëX.WINIMAGE.....
00000010   02 E0 00 40 0B F0 09 00  12 00 02 00 00 00 00 00   .à.@.ð..........
00000020   00 00 00 00 00 00 29 4E  33 70 39 20 20 20 20 20   ......)N3p9
00000030   20 20 20 20 20 20 46 41  54 31 32 20 20 20 00 00         FAT12   ..
00000040   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000050   00 00 00 00 00 00 00 00  00 00 FA 33 C0 8E D0 BC   ..........ú3ÀŽÐ¼
00000060   00 7C B8 B0 07 8E D8 8E  C0 B9 00 01 8B F1 BF 00   .|¸°.ŽØŽÀ¹..‹ñ¿.
00000070   03 F3 A5 B8 D0 07 50 8E  D8 8E C0 B8 80 01 50 CB   .ó¥¸Ð.PŽØŽÀ¸€.PË
00000080   FB BE 13 02 E8 3A 00 B8  01 02 B9 01 00 BA 80 00   û¾..è:.¸..¹..°€.
00000090   33 DB 8E C3 BB 00 7C 06  53 CD 13 72 0A 26 81 3E   3ÛŽÃ».|.SÍ.r.&.>
000000A0   FE 7D 55 AA 75 01 CB BE  D0 01 E8 14 00 B4 01 CD   þ}Uªu.Ë¾Ð.è..´.Í
000000B0   16 74 06 32 E4 CD 16 EB  F4 32 E4 CD 16 33 D2 CD   .t.2äÍ.ëô2äÍ.3ÒÍ
000000C0   19 FC AC 0A C0 74 08 56  B4 0E CD 10 5E EB F3 C3   .ü¬.Àt.V´.Í.^ëóÃ
000000D0   43 61 6E 6E 6F 74 20 6C  6F 61 64 20 66 72 6F 6D   Cannot load from
000000E0   20 68 61 72 64 64 69 73  6B 2E 0D 0A 49 6E 73 65    harddisk...Inse
000000F0   72 74 20 53 79 73 74 65  6D 64 69 73 6B 20 61 6E   rt Systemdisk an
00000100   64 20 70 72 65 73 73 20  61 6E 79 20 6B 65 79 2E   d press any key.
00000110   0D 0A 00 44 69 73 6B 20  66 6F 72 6D 61 74 74 65   ...Disk formatte
00000120   64 20 77 69 74 68 20 57  69 6E 49 6D 61 67 65 20   d with WinImage
00000130   36 2E 35 30 20 28 63 29  20 31 39 39 33 2D 32 30   6.50 (c) 1993-20
00000140   30 34 20 47 69 6C 6C 65  73 20 56 6F 6C 6C 61 6E   04 Gilles Vollan
00000150   74 0D 0A 73 65 65 20 68  74 74 70 3A 2F 2F 77 77   t..see http://ww
00000160   77 2E 77 69 6E 69 6D 61  67 65 2E 63 6F 6D 0D 0A   w.winimage.com..
00000170   42 6F 6F 74 73 65 63 74  6F 72 20 66 72 6F 6D 20   Bootsector from
00000180   43 2E 48 2E 20 48 6F 63  68 73 74 61 74 74 65 72   C.H. Hochstatter
00000190   0D 0A 0D 0A 4E 6F 20 53  79 73 74 65 6D 64 69 73   ....No Systemdis
000001A0   6B 2E 20 42 6F 6F 74 69  6E 67 20 66 72 6F 6D 20   k. Booting from
000001B0   68 61 72 64 64 69 73 6B  2E 0D 0A 00 00 00 00 00   harddisk........
000001C0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000001D0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000001E0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000001F0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 AA   ..............Uª
```

Рис. 2.3. Содержимое загрузочного сектора дискеты, созданной с помощью программы *WinImage*

Классические *BIOS* не предпринимают попыток анализа используемой на загрузочном накопителе файловой системы: все правила запуска загрузчика остаются неизменными. Таким образом, содержимое загрузочного сектора, приведённое на рис. 2.3, можно рассматривать как машинный код для реального режима.

Первые 2 байта соответствуют короткому (*short*) безусловному переходу на `58h` байт вперёд (здесь и далее в левой колонке указан физический адрес):

```
07C00   EB 58           jmp +58h        ; -> 07C5A
```

Следующий байт – инструкция `nop`:

```
07C02   90              xchg ax, ax     ; nop
```

Чтобы понять, зачем здесь нужен `nop`, обратимся к спецификации *FAT*. Первые 3 байта загрузочного сектора – поле `BS_jmpBoot` – используются в этой файловой системе для записи инструкции безусловного перехода к коду загрузчика. Обычно здесь применяется короткий переход, цель которого задаётся относительно текущего значения `IP`. Эта инструкция занимает в машинном коде 2 байта (по одному байту для кода операции и для расстояния прыжка), и тогда третий байт оказывается ненужным. Однако возможны ситуации, когда дальность короткого перехода (от –128 до +127 байт) оказывается недостаточной. В этом случае приходится использовать ближний переход, где задаётся непосредственно новое значение регистра `IP`, – такая инструкция занимает как раз 3 байта: один байт для кода операции и ещё два – для значения смещения. Чтобы иметь возможность вписать сюда оба варианта перехода, отводится 3 байта, а «ненужный» байт, если он есть, заполняется `nop`'ом.

```
07C03                   db 'WINIMAGE'   ; BS_OEMName
```

Следующие 8 байт занимает поле `BS_OEMName`. В соответствии со спецификацией *FAT* здесь записывается значение, которое каким-либо образом идентифицирует систему, которая выполняла форматирование раздела. Утверждается, что использование каких-либо значений, кроме `MSWIN4.1`, может помешать некоторым программным средствам корректно работать с таким разделом, однако на практике это крайне маловероятно. К тому же содержащаяся в этом поле информация не требуется для определения расположения файлов на диске.

Далее следует целая группа полей, которая называется *BPB – BIOS Parameter Block*. Здесь содержится информация, которая может понадобиться для работы с разделом, в том числе – сведения о размере сектора, геометрии диска, расположении раздела на диске, количестве копий *FAT* и т. п.

```
07C0B   00 02           dw 512          ; BPB_BytsPerSec
07C0D   01              db 1            ; BPB_SecPerClus
07C0E   01 00           dw 1            ; BPB_RsvdSecCnt
07C10   02              db 2            ; BPB_NumFATs
07C11   E0 00           dw 224          ; BPB_RootEntCnt
07C13   40 0B           dw 2880         ; BPB_TotSec16
07C15   F0              db $F0          ; BPB_Media
07C16   09 00           dw 9            ; BPB_FATSz16
07C18   12 00           dw 18           ; BPB_SecPerTrk
07C1A   02 00           dw 2            ; BPB_NumHeads
07C1C   00 00 00 00     dd 0            ; BPB_HiddSec
07C20   00 00 00 00     dd 0            ; BPB_TotSec32
```

Как следует из значений полей для разбираемого нами образа, этот *FAT*-раздел располагается на диске с размером сектора 512 байт (`BPB_BytsPerSec`), 18 секторами на трек (`BPB_SecPerTrk`) и 2 головками (`BPB_NumHeads`). Раздел размещён в самом начале диска (`BPB_HiddSec`) и занимает 2880 секторов (поле `BPB_TotSec16` при нулевом значении поля `BPB_TotSec32`).

В пределах раздела только один сектор является зарезервированным (`BPB_RsvdSecCnt`) – сам загрузочный сектор. После него располагается область таблиц размещения файлов: 2 таблицы (`BPB_NumFATs`) по 9 секторов каждая (`BPB_FATSz16`). Далее следует корневая директория, в которой может быть не более 224 элементов (`BPB_RootEntCnt`), что соответствует 14 секторам, т. к. размер одного элемента директории составляет 32 байта. Область данных состоит из кластеров размером в 1 сектор (`BPB_SecPerClus`).

Поле `BPB_Media` осталось в *BPB* по историческим причинам и в процессе работы с *FAT*-разделом на сегодняшний день интереса не представляет.

```
07C24   00              db $00          ; BS_DrvNum
07C25   00              db 0            ; BS_Reserved1
07C26   29              db $29          ; BS_BootSig
07C27   4E 33 70 39     dd $3970334E    ; BS_VolID
07C2B   20 ... 20       db 11 dup(' ')  ; BS_VolLab
07C36   46 ... 20       db 'FAT12   '   ; BS_FilSysType
```

Значения, записанные в эти поля, в основном также не представляют ценности при написании загрузчика. Для разработчика других компонентов операционной системы некоторую пользу могут иметь поля `BS_VolID` (для идентификации разделов) и `BS_VolLab` (для хранения отображаемого имени раздела).

Реализации загрузчиков ***«OSE!»*** для *FAT* в начале своей работы перезаписывают прочитанное в оперативную память значение поля `BS_DrvNum` значением регистра `DL`, полученным от *BIOS*, чтобы упростить передачу этой величины между частями загрузчика.

```
07C3E   00 ... 00       db 28 dup(0)
```

В соответствии со спецификацией *FAT* далее может записываться код загрузчика. Тем не менее реализация, которую использует *WinImage*, передаёт управление физическому адресу `07C5A`, что на 28 байт дальше, чем заканчива-

ются служебные поля *FAT*12. Причиной, вероятнее всего, является то, что используется один и тот же код загрузчика для всех версий *FAT*, а в *FAT*32 служебных полей в загрузочном секторе больше и они занимают суммарно как раз 90 байт (`5A`, если перевести в шестнадцатеричную систему счисления).

```
07C5A   FA              cli
07C5B   33 C0           xor ax, ax
07C5D   8E D0           mov ss, ax
07C5F   BC 00 7C        mov sp, $7C00
```

Работа загрузчика, предлагаемого *WinImage*, начинается с настройки стека. Дно стека размещается по адресу `0000:7C00`, т. е. непосредственно перед кодом загрузочного сектора в памяти. Перед тем как настраивать стек, загрузчик отключает обработку прерываний, что, как было отмечено ранее, является излишним из-за особенностей обработки прерываний процессором после записи в регистр `SS`. Ошибкой это не является, к тому же не исключено, что это потребуется для последующих действий загрузчика.

```
07C62   B8 B0 07        mov ax, $07B0
07C65   8E D8           mov ds, ax
07C67   8E C0           mov es, ax
07C69   B9 00 01        mov cx, $0100
07C6C   8B F1           mov si, cx
07C6E   BF 00 03        mov di, $0300
07C71   F3 A5           rep movsw
```

Далее происходит настройка сегментных регистров `DS` и `ES` на сегмент `07B0h`, а затем копирование `100h` слов (по два байта каждое) из области памяти, начинающейся по адресу `07B0:0100` (физический адрес `07C00`), в область по адресу `07B0:0300` (физический адрес `07E00`). Таким образом, в результате выполнения этих действий в памяти окажется две копии загрузочного сектора: одна – загруженная с диска BIOS – располагается в диапазоне физических адресов с `07C00` по `07DFF`, вторая – по физическим адресам с `07E00` по `07FFF`, т. е. непосредственно за исходной копией и в самом конце первых 32 Кбайт оперативной памяти.

```
07C73   B8 D0 07        mov ax, $07D0
07C76   50              push ax
07C77   8E D8           mov ds, ax
07C79   8E C0           mov es, ax
07C7B   B8 80 01        mov ax, $0180
07C7E   50              push ax
07C7F   CB              retf
```

Следующий фрагмент кода перенастраивает сегментные регистры `DS` и `ES` на сегмент `07D0h` и, поместив на вершину стека значения `07D0h` и `0180h`, выполняет дальний возврат из процедуры, что приводит к передаче управления по адресу `07D0:0180` (физический адрес `07E80`). Важно отметить, что при этом

происходит также перезапись значения регистра CS, которое до этого не было достоверно известным.

Адрес, куда происходит передача управления, соответствует смещению 80h внутри ранее созданной копии загрузочного сектора. При этом следующая за retf инструкция как раз располагается по смещению 80h от начала загрузочного сектора, прочитанного в память *BIOS*.

Таким образом, управление фактически передаётся следующей за retf инструкции, но теперь она и последующие инструкции будут считываться из другой области памяти. В таком случае говорят, что произошло перемещение загрузочного сектора в памяти. Обычно так делают, если планируется память, изначально занятую кодом загрузочного сектора, переиспользовать для других целей. Как правило, такое решение применяют при загрузке с жёстких дисков, где загрузочный сектор перемещает себя в памяти, а затем находит первый сектор одного из разделов и загружает его вместо себя всё по тому же физическому адресу 07C00. Для загрузки с дискеты это смысла не имеет, но, по всей видимости, *WinImage* использует универсальный фрагмент кода для различных накопителей и то, что мы видим, – следствие этой универсальности.

```
07E80   FB              sti
07E81   BE 13 02        mov si, $0213
07E84   E8 3A 00        call +$003A      ; -> 07EC1
```

После перемещения происходит включение обработки прерывания и вызов процедуры. В этот момент регистры CS:IP содержат значения 07D0:0187, а для инструкции call используется относительная форма, следовательно, вызываемая процедура находится на 003Ah байт дальше в памяти, т. е. по адресу 07D0:01C1 (физический адрес 07EC1). Проанализируем её код.

```
07EC1   FC              cld
07EC2   AC              lodsb
07EC3   0A C0           or al, al
07EC5   74 08           jz +$08          ; -> 07ECF
07EC7   56              push si
07EC8   B4 0E           mov ah, $0E
07ECA   CD 10           int 10h
07ECC   5E              pop si
07ECD   EB F3           jmp -$0D         ; -> 07EC2
07ECF   C3              ret
```

Данная процедура осуществляет посимвольный вывод *C*-строки, начинающейся по адресу DS:SI, на экран с помощью функции int 10h/0Eh. Перед её вызовом в SI было помещено значение 0213h, а в регистре DS содержалось значение 07D0h, следовательно, выводимая строка начинается по адресу 07D0:0213 (физический адрес 07F13). Это соответствует смещению 0113h от начала загрузочного сектора, а по этому смещению в образе диска как раз записана *C*-строка, которая выводится при попытке загрузки со сформированной *WinImage* дискеты:

```
Disk formatted with WinImage 6.50 (c) 1993-2004 Gilles Vollant
see http://www.winimage.com
Bootsector from C.H. Hochstatter

No Systemdisk. Booting from harddisk.
```

Вернёмся в точку вызова этой процедуры и продолжим анализ кода.

```
07E87   B8 01 02        mov ax, $0201
07E8A   B9 01 00        mov cx, $0001
07E8D   BA 80 00        mov dx, $0080
07E90   33 DB           xor bx, bx
07E92   8E C3           mov es, bx
07E94   BB 00 7C        mov bx, $7C00
07E97   06              push es
07E98   53              push bx
07E99   CD 13           int 13h
07E9B   72 0A           jc +$0A          ; -> 07EA7
```

Здесь происходит вызов функции `int 13h/02h`. По адресу `0000:7C00` (регистры `ES:BX`) считывается один сектор (регистр `AL`) с диска номер `80h` (регистр `DL`). Номера дисков с единичным старшим битом соответствуют жёстким дискам. При работе с ними *CHS*-номер сектора задаётся несколько сложнее, чем при работе с дискетами: в регистр `CL` записывается не только номер сектора на дорожке, но и два старших бита номера цилиндра. Тем не менее в данном случае эта особенность ни на что не влияет. Обращение происходит к сектору (0, 0, 1), т. е. *LBA* этого сектора равен 1. Это сектор, следующий сразу за загрузочным.

```
07E9D   26 81 ... AA    cmp [es:$7DFE], $AA55
07EA4   75 01           jne +$01          ; -> 07EA7
07EA6   CB              retf
```

После считывания сектора его последние 2 байта сравниваются с сигнатурой `55 AA`. В данном случае код загрузчика фактически выполняет ту же проверку, которую использует при поиске загрузочного сектора *BIOS*. Если сигнатура обнаружена, выполняется дальний возврат из процедуры. На вершине стека при этом находятся ранее помещённые туда значения регистров `ES` и `BX`, задающие адрес перехода `0000:7C00`.

Если сигнатура `55 AA` отсутствует, инструкция `retf` пропускается и управление передаётся следующему фрагменту кода. То же самое происходит и при ошибочном завершении вызова функции `int 13h/02h`.

```
07EA7   BE D0 01        mov si, $01D0
07EAA   E8 14 00        call +$0014      ; -> 07EC1
```

В этом фрагменте кода происходит вызов уже проанализированной выше процедуры, отвечающей за отображение *C*-строки. В качестве адреса строки в регистрах `DS:SI` передаётся `07D0:01D0`, что соответствует физическому

адресу `07ED0` или смещению `D0h` от начала загрузочного сектора. Там размещена *C*-строка, соответствующая следующему сообщению:

```
Cannot load from harddisk.
Insert Systemdisk and press any key.
```

После вывода этого сообщения будет выполнено ещё несколько действий для обеспечения удобства пользователя.

```
07EAD   B4 01           mov ah, $01
07EAF   CD 16           int 16h
07EB1   74 06           je +$06          ; -> 07EB9
07EB3   32 E4           xor ah, ah
07EB5   CD 16           int 16h
07EB7   EB F4           jmp -$0C         ; -> 07EAD

07EB9   32 E4           xor ah, ah
07EBB   CD 16           int 16h
07EBD   33 D2           xor dx, dx
07EBF   CD 19           int 19h
```

Данный фрагмент кода очищает клавиатурный буфер *BIOS* путём вызова функции `int 16h/00h` до тех пор, пока функция `int 16h/01h` не сообщит об отсутствии в буфере необработанных нажатий клавиш, после чего однократно вызывает функцию `int 16h/00h` при пустом буфере, что приводит к гарантированному ожиданию нажатия клавиши. После нажатия клавиши обнуляется регистр `DX` и генерируется прерывание `int 19h`, запускающее повторный поиск загрузочного накопителя.

Далее следует код уже рассмотренной процедуры вывода *C*-строки и используемые загрузчиком строковые данные. Часть загрузочного сектора от окончания кода и данных загрузчика до конца сектора (кроме последних 2 байт) заполнена нулями и не используется. Последние 2 байта содержат сигнатуру `55 AA`, позволяющую *BIOS* рассматривать дискету как загрузочную.

Следует понимать, что рассмотренный код загрузчика *WinImage* имеет ряд недостатков. Первая проблема заключается в том, что попытка загрузки с диска `80h` может выглядеть для пользователя нелогичной, если порядок загрузки в настройках *BIOS* выбран так, чтобы после дискеты просматривался не жёсткий диск (первый, если их в системе несколько), а другой источник. Загрузчик *WinImage* в этом случае фактически нарушит запрошенный пользователем компьютера порядок загрузки.

Вторая проблема связана с самой загрузкой с жёсткого диска. Сектор с *LBA* 1 может не быть началом первого раздела на этом диске, а даже если первый раздел действительно будет начинаться в этом секторе, он может оказаться не загрузочным. Поиск загрузочного раздела на жёстких дисках принято производить иначе, поэтому такая попытка выполнить загрузку с жёсткого диска не может считаться правильной.

Обратите внимание на то, что структура данных в начале загрузочного сектора *FAT*-раздела по своей сути является развитием идеи, которая была использована в примере 1.2 для повышения гибкости загрузчика: в частности, вместо `wNumHeads` и `wSecPerTrack` используются поля `BPB_NumHeads` и `BPB_SecPerTrk`. На смену конкретному значению `wNumSectors` из этого примера – количеству секторов, которые необходимо прочитать, – приходит разбор структур данных файловой системы, поиск требуемого файла на диске и его загрузка.

Продолжим анализ образа созданной *WinImage* дискеты. В соответствии со спецификацией *FAT* первые `BPB_RsvdSecCnt` секторов (для данной дискеты – 1) являются зарезервированными. После этого `BPB_NumFATs` групп по `BPB_FATSz16` секторов (для данной дискеты – 2 группы по 9 секторов) используются для записи таблицы размещения файлов, которая и дала название семейству файловых систем *FAT*.

Таблица размещения файлов (*FAT*) представляет собой массив целых чисел – номеров кластеров. Размер элемента определяется версией файловой системы: в *FAT*12 это 12 бит, в *FAT*16 – 16 бит (2 байта), в *FAT*32 – 32 бита (4 байта). Для определения того, какая версия используется, необходимо проанализировать количество кластеров: поля в загрузочном секторе наподобие `BS_FilSysType` не считаются достоверным источником такой информации. Подробности можно найти в спецификации *FAT*.

Для дискет, как правило, применяют *FAT*12. Использование этой версией файловой системы 12-битных элементов *FAT* создаёт некоторые неудобства: работать с элементами, размеры которых не кратны байту, приходится с использованием побитовых операций и манипуляций с индексами и смещениями. Каждые два элемента *FAT* занимают по 3 байта, причём первому элементу из пары соответствуют младшие 12 бит, а второму – старшие. Значения самих битов записываются в порядке *little-endian* – от младшего к старшему.

```
                                        ...

000001F0   00 00 00 00 00 00 00 00   00 00 00 00 00 00 55 AA

00000200   F0 FF FF 00 00 00 00 00   00 00 00 00 00 00 00 00
00000210   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00000220   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

                                        ...
```

В образе дискеты сектор с *LBA* 1 располагается по смещению `200h`. Первые три байта содержат элементы `FAT[0]` и `FAT[1]`. Чтобы понять, какие значения имеют эти элементы, запишем значения битов в «математическом» порядке, от старшего к младшему:

```
FF FF F0
```

Младшие 12 бит, соответствующие элементу `FAT[0]`, – это три 16-ричные цифры `FF0`, старшие, соответствующие `FAT[1]`, – `FFF`. Значения этих двух элементов устанавливаются в соответствии с требованиями спецификации *FAT*. Элемент `FAT[0]` должен содержать значение поля `BPB_Media` из загрузочного сектора со зна-

ковым расширением до требуемого количества битов. В нашем случае в BPB_Media записано значение F0h. В результате знакового расширения до 12 бит получим как раз величину FF0. Элемент же FAT[1] должен содержать специальное значение, которое используется для обозначения окончания цепочки кластеров. В случае *FAT*12 это значение FFFh, для *FAT*16 и *FAT*32 это значение получается немного сложнее.

Все последующие элементы *FAT* используются для хранения информации о цепочках кластеров: значение элемента FAT[I] должно содержать номер кластера, следующего за кластером с номером I, или одно из специальных значений. Значения элементов в таблице размещения файлов для *FAT*12 приведены в табл. 2.1.

Таблица 2.1

Значения элементов таблицы размещения файлов для *FAT*12

Значение	Описание
000	Свободный кластер
001	Значение не используется
002–FF6	Номер следующего кластера
FF7	Повреждённый кластер
FF8–FFF	Последний кластер в цепочке

Следует понимать, что значения первых двух элементов носят чисто технический, вспомогательный характер. Нумерация кластеров в области данных начинается с 2. В образе пустой дискеты все элементы *FAT*, начиная с индекса 2, содержат значение 000h. Это означает, что все кластеры в области данных не заняты, диск с точки зрения файловой системы пуст.

В *FAT*-разделах, как правило, применяется не менее двух копий таблицы размещения файлов. При повседневном использовании для работы с файлами используют только одну из таблиц, а при записи данных на диск обновляют обе. Предполагается, что в случае выхода из строя той части диска, где записана одна из них, утилиты восстановления данных смогут использовать резервную копию – вторую таблицу. Нетрудно убедиться в том, что в образе пустой дискеты, созданной *WinImage*, секторы, занятые двумя копиями таблицы размещения файлов, действительно совпадают: первая копия занимает сектора с *LBA* 1–9, вторая – сектора с *LBA* 10–18.

В секторе с *LBA* 19 (смещение 2600h в образе дискеты) начинается корневая директория. В ней содержится 224 элемента (см. поле BPB_RootEntCnt загрузочного сектора) по 32 байта каждый, что соответствует 14 секторам. Поскольку рассматриваемая дискета пуста, в этих 14 секторах записаны нули.

Наконец, с сектора с *LBA* 33 (смещение 4200h в образе дискеты) начинается область данных. Поскольку на дискету не записано никаких файлов, эта область также заполнена нулями. Следует понимать, что при форматировании реального диска сектора в области данных могут не заполняться нулями в целях экономии времени, поэтому, если ранее в них содержались какие-то дан-

ные, их можно увидеть и на отформатированном носителе (в частности, если использовалось так называемое быстрое форматирование).

Пример 2.2. *FAT*12-дискета с файлами

Следующим шагом для понимания семейства файловых систем *FAT* является анализ образа дискеты, на которую записаны файлы. В этом примере рассматривается дискета, содержимое которой представлено на рис. 2.4.

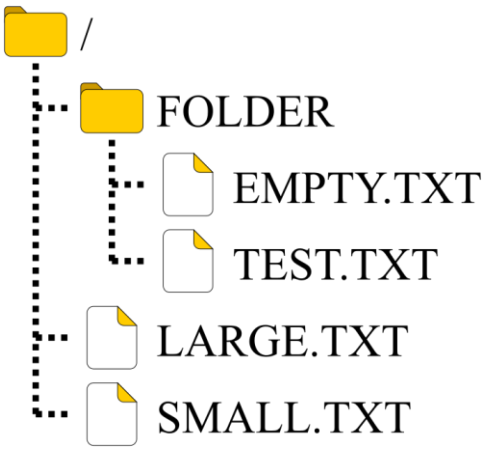

Рис. 2.4. Содержимое дискеты с файлами, используемой в примере

Загрузочный сектор такой дискеты ничем не отличается от рассмотренного в примере 2.1, поэтому общая структура раздела – размеры и расположение таблиц размещения файлов, корневой директории и области данных – идентична рассмотренной ранее, изменения коснулись только содержимого этих областей. Поскольку информация о разделе, записанная в загрузочном секторе, уже была разобрана в предыдущем примере, сразу перейдём к перечислению записанных на дискету файлов и определению их местоположения на диске.

Отправной точкой для перечисления содержимого диска, отформатированного в *FAT*, является корневая директория. Ранее уже было определено, что она начинается с сектора с *LBA* 19, т. е. по смещению `2600h` в файле с образом дискеты (рис. 2.5).

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00002600   4C 41 52 47 45 20 20 20  54 58 54 00 00 46 EF 8C   LARGE   TXT..FïŒ
00002610   45 51 00 00 00 00 DA 99  45 51 02 00 53 08 00 00   EQ....Ú™EQ..S...
00002620   53 4D 41 4C 4C 20 20 20  54 58 54 00 00 4C A4 8C   SMALL   TXT..L¤Œ
00002630   45 51 00 00 00 00 3D 8E  45 51 07 00 09 01 00 00   EQ....=ŽEQ......
00002640   46 4F 4C 44 45 52 20 20  20 20 20 10 00 00 00 00   FOLDER     .....
00002650   00 00 00 00 00 00 53 08  46 51 08 00 00 00 00 00   ......S.FQ......
00002660   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```

Рис. 2.5. Содержимое корневой директории дискеты с файлами

Корневая директория представляет собой массив из записей размером 32 байта каждая. Их формат приведён в табл. 2.2.

Таблица 2.2

Формат записи (элемента) директории

Имя поля	Смещение, байт	Размер, байт	Описание
DIR_Name	0	11	Имя файла
DIR_Attr	11	1	Атрибуты файла
DIR_NTRes	12	1	Зарезервировано для использования *Windows NT*
DIR_CrtTimeTenth	13	1	Часть информации о времени создания файла – количество десятых долей секунды
DIR_CrtTime	14	2	Время создания файла
DIR_CrtDate	16	2	Дата создания файла
DIR_LstAccDate	18	2	Дата последнего обращения к файлу
DIR_FstClusHI	20	2	Старшие 2 байта номера первого кластера (всегда 0 для *FAT*12 и *FAT*16)
DIR_WrtTime	22	2	Время последнего изменения файла
DIR_WrtDate	24	2	Дата последнего изменения файла
DIR_FstClusLO	26	2	Младшие 2 байта номера первого кластера
DIR_FileSize	28	4	Размер файла, в байтах

Первый байт каждой записи в директории имеет двойное назначение. Помимо того, что он содержит первый символ имени файла, по нему определяется статус этой записи.

Если DIR_Name[0] содержит значение 00h, эта и все последующие записи в директории свободны, т. е. не соответствуют ни одному файлу и не используются. Как только в директории встречается запись с нулевым значением первого байта, её чтение можно прекращать: больше используемых записей в ней не будет.

Второй особый случай – значение DIR_Name[0], равное E5h. В этом случае запись также считается свободной, однако после неё могут быть используемые записи и чтение директории прерывать не следует. Такие записи могут появляться при выборочном удалении файлов из каталога: вместо полного перестраивания директории первые байты записей, которые соответствовали удаляемым файлам, заменяются значением E5h. Впоследствии при создании нового файла такие записи могут быть использованы заново.

Третье специальное значение – 05h. Если оно встречается в первом байте имени файла, то должно интерпретироваться как символ с кодом E5h. В специ-

фикации *FAT* утверждается, что это сделано в целях совместимости с кодировкой, использовавшейся для японского языка.

В корневой директории рассматриваемого нами образа дискеты используются три записи: для файлов `LARGE.TXT` и `SMALL.TXT`, а также для каталога `FOLDER`. Для записи имён файлов используется формат 8.3: имя файла условно делится на две части – имя и расширение, – причём для имени отводится 8 символов, а для расширения – 3. Если какая-либо из этих частей короче своей максимальной длины, она дополняется пробелами. Поддержка более длинных имён файлов в *FAT* изначально отсутствовала и была добавлена при разработке *Windows* 95, однако в данный момент не представляет для нас интереса.

Байт `DIR_Attr` содержит комбинацию флагов, определяющих атрибуты данного файла. В табл. 2.3 приведены флаги, которые могут использоваться в этом поле.

Таблица 2.3

Флаги атрибутов файла

Имя флага	Значение	Описание
`ATTR_READ_ONLY`	`01h`	Только для чтения
`ATTR_HIDDEN`	`02h`	Скрытый
`ATTR_SYSTEM`	`04h`	Системный
`ATTR_VOLUME_ID`	`08h`	Метка тома (раздела)
`ATTR_DIRECTORY`	`10h`	Директория (каталог, папка)
`ATTR_ARCHIVE`	`20h`	Архивный
`ATTR_LONG_NAME`	`0Fh`	Часть длинного имени файла

Первые две записи в корневой директории рассматриваемого диска имеют значение атрибута `00h` (обычные файлы), третья – `10h` (директория). Таким образом, с точки зрения *FAT* директория является разновидностью файла с тем отличием, что в нём хранятся не произвольные данные, а 32-байтные записи в формате, приведённом в табл. 2.2.

Три поля, которые потребуются для доступа к данным файла, – `DIR_FstClusHI`, `DIR_FstClusLO` и `DIR_FileSize`. Первые два содержат номер кластера, в котором хранится первая часть содержимого файла, третье – размер файла в байтах. В табл. 2.4 приведена информация из этих полей в удобном для восприятия виде.

Таблица 2.4

Информация об элементах корневой директории

Имя файла	Номер первого кластера	Размер, байт
LARGE.TXT	2	2131
SMALL.TXT	7	265
FOLDER	8	0

Попробуем получить доступ к данным файла LARGE.TXT. Первый кластер в цепочке, относящейся к этому файлу, имеет индекс 2. Обратимся к таблице размещения файлов, чтобы определить номера всех кластеров в цепочке (рис. 2.6).

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000200   F0 FF FF 03 40 00 05 60  00 FF FF FF FF FF FF 00   ðÿÿ.@..`.ÿÿÿÿÿÿ.
00000210   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```

Рис. 2.6. Содержимое таблицы размещения файлов

Элемент FAT[2] содержится в группе из 3 байт, начинающейся в файле образа по смещению 203h, причём в младших 12 битах. Значение этих трёх байт – 004003h, следовательно, элемент FAT[2] содержит число 3 – оно и будет номером следующего кластера. Элемент FAT[3] записан в старших 12 битах этой же тройки байт и равен 4. Элементу FAT[4] соответствуют младшие 12 бит следующей группы из трёх байт – и это число 5.

Продолжая разбор таблицы размещения файлов по тому же принципу, получим следующую цепочку: 002–003–004–005–006–FFF. Значение FFF является не номером кластера, а признаком окончания цепочки: кластер, для которого в таблице размещения файлов указывается такое значение, содержит последнюю часть файла. Таким образом, для хранения файла LARGE.TXT задействовано 5 кластеров. С учётом того, что размер кластера для данной дискеты составляет ровно один 512-байтный сектор, это и есть минимально необходимое количество кластеров для записи файла размером 2131 байт.

Теперь найдём эти кластеры на диске. Нумерация кластеров начинается с 2, т. е. кластер, располагающийся в самом начале области данных, имеет индекс 2. Формула для вычисления *LBA* кластера с индексом N выглядит так:

```
ClusterLBA(N) = BPB_RsvdSecCnt
              + BPB_NumFATs * BPB_FATSz16
              + BPB_RootEntCnt * 32 / BPB_BytsPerSec
              + (N - 2) * BPB_SecPerClus
```

В приведённой формуле не учитывается случай, когда количество записей в корневой директории делает её размер не кратным размеру сектора, поскольку спецификация *FAT* такую ситуацию запрещает. Тем не менее следует

иметь в виду этот нюанс на случай, если *FAT*-раздел будет отформатирован программным обеспечением, нарушающим это требование.

Цепочка номеров кластеров для файла `LARGE.TXT` состоит из последовательно возрастающих чисел. Это означает, что содержимое файла на диске также будет находиться в секторах, расположенных последовательно. На практике номера кластеров в цепочке могут быть произвольными, особенно если свободного места на диске мало и он активно использовался до этого. В нашем случае речь идёт об образе дискеты, что эквивалентно записи на только что отформатированную дискету, поэтому цепочка кластеров оказалась именно такой.

Для рассматриваемого нами раздела *LBA* кластера 2 будет равно 33, что соответствует смещению `4200h` от начала диска. Последовательное выделение кластеров для хранения этого файла приведёт к тому, что фактически его данные будут храниться в пяти секторах начиная с сектора 33. Нетрудно убедиться, что это действительно соблюдается для рассматриваемого в примере диска.

С файлом `SMALL.TXT` всё будет несколько проще, т. к. его размер – 265 байт – меньше, чем размер кластера, а значит, для хранения такого файла достаточно будет одного кластера. Для рассматриваемого диска таковым является кластер с индексом 7. В таблице размещения файлов, приведённой ранее (см. рис. 2.6), элемент `FAT[7]` содержит маркер конца цепочки `FFF`, т. е. кластер 7 действительно является одновременно первым и последним кластером данного файла. LBA этого кластера будет равно 38 (смещение `4C00h` от начала диска).

Согласованность цепочки кластеров в таблице размещения файлов и информации о размере файла, записанной в директории, – слабое место файловых систем *FAT*. Если размер блока памяти для считывания данных из файла определяется по информации из директории, а цикл покластерного чтения опирается только на маркер конца цепочки, возможно возникновение уязвимости к атаке переполнением буфера.

Поскольку, как правило, размеры файлов не кратны размеру кластера, в последнем кластере чаще всего остаётся некоторое количество неиспользуемых байтов. Для определения того, какая часть такого кластера реально используется, можно опираться на данные о размере файла.

Неосторожное обращение с неиспользуемыми байтами в последнем кластере может стать причиной утечки данных. При копировании файла или уменьшении его размера перед записью соответствующих секторов на диск имеет смысл принудительно заполнять неиспользуемые байты каким-либо значением, не имеющим отношения к ранее хранившимся там данным (например, нулями). В противном случае фрагменты ранее записанных в них данных могут оказаться скопированными на другой носитель и стать доступными для анализа злоумышленником.

Особым видом файлов являются директории. Помимо соответствующего атрибута, они отличаются тем, что для них размер файла всегда указывается равным 0. Определить фактический размер директории несложно: во-первых, в таблице размещения файлов для неё по-прежнему создаётся цепочка кластеров, а во-вторых, сам формат директории позволяет определить, когда достигнута последняя её запись.

Для директории `FOLDER` в рассматриваемом примере первым кластером в цепочке является кластер с индексом 8, что соответствует смещению `4E00h` в файле с образом диска. При этом элемент `FAT[8]` равен `FFF`, т. е. эта директория занимает только один кластер (рис. 2.7).

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00004E00   2E 20 20 20 20 20 20 20  20 20 20 10 00 00 00 00   .          .....
00004E10   00 00 00 00 00 00 53 08  46 51 08 00 00 00 00 00   ......S.FQ......
00004E20   2E 2E 20 20 20 20 20 20  20 20 20 10 00 00 00 00   ..         .....
00004E30   00 00 00 00 00 00 53 08  46 51 00 00 00 00 00 00   ......S.FQ......
00004E40   54 45 53 54 20 20 20 20  54 58 54 00 00 3F 5C 08   TEST    TXT..?\.
00004E50   46 51 00 00 00 00 8F 08  46 51 09 00 11 00 00 00   FQ......FQ......
00004E60   45 4D 50 54 59 20 20 20  54 58 54 00 00 B7 28 04   EMPTY   TXT..·(.
00004E70   47 51 00 00 00 00 29 04  47 51 00 00 00 00 00 00   GQ....).GQ......
00004E80   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00004E90   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```

Рис. 2.7. Содержимое директории `FOLDER`

В табл. 2.5 приведена информация о содержимом директории в соответствии с данными, записанными в её единственном кластере.

Таблица 2.5

Информация об элементах директории `FOLDER`

Имя файла	Номер первого кластера	Размер, байт
`.`	8	0
`..`	0	0
`TEST.TXT`	9	17
`EMPTY.TXT`	0	0

Во всех директориях, кроме корневой, первыми двумя элементами должны быть специальные служебные записи «`.`» (*dot*-элемент) и «`..`» (*dotdot*-элемент). Несмотря на атрибут `ATTR_DIRECTORY` они не являются полноценными вложенными директориями. Поля `DIR_FstClusHI` и `DIR_FstClusLO` для *dot*-элемента содержат номер первого кластера той директории, в которой этот элемент находится, т. е. *dot*-элемент, по сути, представляет собой ссылку на текущую директорию. В свою очередь *dotdot*-элемент является точно такой же ссылкой, но на директорию, которая является родительской для текущей. Если

dotdot-элемент записывается в директорию, непосредственно вложенную в корневую, поля `DIR_FstClusHI` и `DIR_FstClusLO` содержат число 0.

Помимо двух служебных элементов, директория `FOLDER` содержит также записи о файлах `TEST.TXT` и `EMPTY.TXT`. Файл `TEST.TXT` особого интереса для нас не представляет, т. к. обрабатывается аналогично ранее рассмотренным файлам: его цепочка состоит из единственного кластера с индексом 9. Файл же `EMPTY.TXT` интересен тем, что является пустым, т. е. имеет размер 0 байт. В этой ситуации выделять для него даже один кластер было бы неэффективно, поэтому в качестве номера первого кластера в полях `DIR_FstClusHI` и `DIR_FstClusLO` задано число 0, что является признаком пустого файла наравне с нулевым значением в поле `DIR_FileSize`.

Пример 2.3. Загрузочная дискета *MS-DOS*

Практичным и логичным выглядит решение возложить на код загрузочного сектора задачу загрузки в память произвольного файла, записанного на этом же разделе. В этом случае файл, содержащий вторичный загрузчик, ядро операционной системы или её модуль, который должен быть загружен первым, можно будет заменять при помощи обычного файлового менеджера (особенно при использовании накопителей, отличных от дискеты).

Как следует из рассмотренных ранее примеров, для загрузки файла в память в общем случае необходимо проанализировать ряд структур данных: информацию из загрузочного сектора, содержимое корневой директории и всех вложенных директорий на пути к искомому файлу, а также содержимое таблицы размещения файлов как для самого файла, так и для всех промежуточных директорий. При этом необходимо учитывать особенности записи информации в этих структурах – правила интерпретации значений элементов в *FAT*, записей в директориях и т. п., – что в сочетании с небольшими размерами загрузочного сектора представляет собой серьёзную проблему.

Так, например, на *FAT*12-разделе данные, записанные в загрузочном секторе и задающие основные характеристики диска и файловой системы, занимают 62 байта. Ещё 2 байта тратится на загрузочную сигнатуру в байтах 510 и 511. Таким образом, остаётся всего лишь 448 байт непосредственно для кода загрузчика. В них, помимо анализа структур файловой системы, необходимо поместить хотя бы минимальный код для отображения сообщений о возникающих ошибках. Не следует также забывать о том, что при передаче управления загрузчику значения большинства регистров могут быть произвольными и для их инициализации также потребуется некоторое количество инструкций.

Даже если средний размер используемых инструкций будет составлять 2 байта, в загрузочный сектор поместится примерно 224 инструкции. Это очень серьёзное ограничение, делающее «честную» загрузку файла почти невозможной. При этом следует понимать, что приведённая оценка может оказаться излишне оптимистичной и в реальности средний размер инструкции может в зависимости от особенностей написания кода оказаться больше.

Один из возможных способов обхода этой проблемы заключается в использовании поля `BPB_RsvdSecCnt` в загрузочном секторе. В нём фактически задаётся количество секторов (включая загрузочный нулевой сектор), которые предшествуют *FAT* и другим структурам данных файловой системы. Указывая здесь значение большее 1, можно задействовать произвольное количество секторов в начале *FAT*-раздела для хранения произвольной информации, в том числе – загрузчика, который не помещается в загрузочный сектор. В этом случае в коде загрузочного сектора достаточно осуществить чтение нескольких последовательно расположенных на диске секторов и в нужный момент передать туда управление.

К сожалению, такой способ решения проблемы следует признать нежелательным. В спецификации *FAT* явно указано, что для разделов, отформатированных в *FAT*12 и *FAT*16, значение поля `BPB_RsvdSecCnt` всегда должно быть равно 1. Хотя технически другие значения являются допустимыми, большое количество существующего программного обеспечения игнорирует записанное в этом поле значение, считая его равным единице. Таким образом, *FAT*-раздел с другим значением в этом поле может быть воспринят некоторыми программами и/или операционными системами как некорректно отформатированный.

На практике разработчикам операционных систем, которые должны загружаться с *FAT*-разделов, приходится упрощать задачу, возлагаемую на код загрузочного сектора. В качестве примера такого упрощения рассмотрим код, используемый загрузочной дискетой с операционной системой *MS-DOS 6.22* (рис. 2.8).

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   EB 3C 90 4D 53 44 4F 53  35 2E 30 00 02 01 01 00   ë<.MSDOS5.0.....
00000010   02 E0 00 40 0B F0 09 00  12 00 02 00 00 00 00 00   .à.@.ð..........
00000020   00 00 00 00 00 00 29 E7  1A 17 0A 4E 4F 20 4E 41   ......)ç...NO NA
00000030   4D 45 20 20 20 20 46 41  54 31 32 20 20 20 FA 33   ME    FAT12   ú3
00000040   C0 8E D0 BC 00 7C 16 07  BB 78 00 36 C5 37 1E 56   ÀŽÐ¼.|..»x.6Å7.V
00000050   16 53 BF 3E 7C B9 0B 00  FC F3 A4 06 1F C6 45 FE   .S¿>|¹..üó¤..ÆEþ
00000060   0F 8B 0E 18 7C 88 4D F9  89 47 02 C7 07 3E 7C FB   .‹..|ˆMù‰G.Ç.>|û
00000070   CD 13 72 79 33 C0 39 06  13 7C 74 08 8B 0E 13 7C   Í.ry3À9..|t.‹..|
00000080   89 0E 20 7C A0 10 7C F7  26 16 7C 03 06 1C 7C 13   ‰. | .|÷&.|...|.
00000090   16 1E 7C 03 06 0E 7C 83  D2 00 A3 50 7C 89 16 52   ..|...|ƒÒ.£P|‰.R
000000A0   7C A3 49 7C 89 16 4B 7C  B8 20 00 F7 26 11 7C 8B   |£I|‰.K|¸ .÷&.|‹
000000B0   1E 0B 7C 03 C3 48 F7 F3  01 06 49 7C 83 16 4B 7C   ..|.ÃH÷ó..I|ƒ.K|
000000C0   00 BB 00 05 8B 16 52 7C  A1 50 7C E8 92 00 72 1D   .»..‹.R|¡P|è'.r.
000000D0   B0 01 E8 AC 00 72 16 8B  FB B9 0B 00 BE E6 7D F3   °.è¬.r.‹û¹..¾æ}ó
000000E0   A6 75 0A 8D 7F 20 B9 0B  00 F3 A6 74 18 BE 9E 7D   ¦u... ¹..ó¦t.¾ž}
000000F0   E8 5F 00 33 C0 CD 16 5E  1F 8F 04 8F 44 02 CD 19   è_.3ÀÍ.^....D.Í.
00000100   58 58 58 EB E8 8B 47 1A  48 48 8A 1E 0D 7C 32 FF   XXXëè‹G.HHŠ..|2ÿ
00000110   F7 E3 03 06 49 7C 13 16  4B 7C BB 00 07 B9 03 00   ÷ã..I|..K|»..¹..
00000120   50 52 51 E8 3A 00 72 D8  B0 01 E8 54 00 59 5A 58   PRQè:.rØ°.èT.YZX
00000130   72 BB 05 01 00 83 D2 00  03 1E 0B 7C E2 E2 8A 2E   r»...ƒÒ....|ââŠ.
00000140   15 7C 8A 16 24 7C 8B 1E  49 7C A1 4B 7C EA 00 00   .|Š.$|‹.I|¡K|ê..
00000150   70 00 AC 0A C0 74 29 B4  0E BB 07 00 CD 10 EB F2   p.¬.Àt)´.»..Í.ëò
00000160   3B 16 18 7C 73 19 F7 36  18 7C FE C2 88 16 4F 7C   ;..|s.÷6.|þÂˆ.O|
00000170   33 D2 F7 36 1A 7C 88 16  25 7C A3 4D 7C F8 C3 F9   3Ò÷6.|ˆ.%|£M|øÃù
00000180   C3 B4 02 8B 16 4D 7C B1  06 D2 E6 0A 36 4F 7C 8B   Ã´.‹.M|±.Òæ.6O|‹
00000190   CA 86 E9 8A 16 24 7C 8A  36 25 7C CD 13 C3 0D 0A   Ê†éŠ.$|Š6%|Í.Ã..
000001A0   4E 6F 6E 2D 53 79 73 74  65 6D 20 64 69 73 6B 20   Non-System disk
000001B0   6F 72 20 64 69 73 6B 20  65 72 72 6F 72 0D 0A 52   or disk error..R
000001C0   65 70 6C 61 63 65 20 61  6E 64 20 70 72 65 73 73   eplace and press
000001D0   20 61 6E 79 20 6B 65 79  20 77 68 65 6E 20 72 65    any key when re
000001E0   61 64 79 0D 0A 00 49 4F  20 20 20 20 20 20 53 59   ady...IO      SY
000001F0   53 4D 53 44 4F 53 20 20  20 53 59 53 00 00 55 AA   SMSDOS   SYS..Uª
```

Рис. 2.8. Содержимое загрузочного сектора загрузочной дискеты *MS-DOS 6.22*

Структура данных в начале загрузочного сектора содержит типовые для *FAT*12 значения.

```
07C00   EB 3C           jmp +3Ch          ; -> 07C3E
07C02   90              xchg ax, ax       ; nop

07C03                   db 'MSDOS5.0'     ; BS_OEMName
07C0B   00 02           dw 512            ; BPB_BytsPerSec
07C0D   01              db 1              ; BPB_SecPerClus
07C0E   01 00           dw 1              ; BPB_RsvdSecCnt
07C10   02              db 2              ; BPB_NumFATs
07C11   E0 00           dw 224            ; BPB_RootEntCnt
07C13   40 0B           dw 2880           ; BPB_TotSec16
07C15   F0              db $F0            ; BPB_Media
07C16   09 00           dw 9              ; BPB_FATSz16
07C18   12 00           dw 18             ; BPB_SecPerTrk
07C1A   02 00           dw 2              ; BPB_NumHeads
07C1C   00 00 00 00     dd 0              ; BPB_HiddSec
07C20   00 00 00 00     dd 0              ; BPB_TotSec32
07C24   00              db $00            ; BS_DrvNum
07C25   00              db 0              ; BS_Reserved1
07C26   29              db $29            ; BS_BootSig
07C27   E7 1A 17 0A     dd $0A171AE7      ; BS_VolID
07C2B   4E ... 20       db 'NO NAME    ' ; BS_VolLab
07C36   46 ... 20       db 'FAT12   '     ; BS_FilSysType
```

Единственное существенное отличие в этой части загрузочного сектора от ранее рассмотренной дискеты, созданной в *WinImage*, – в более правильном обозначении диска без метки: при отсутствии метки у диска в поле `BS_VolLab` должны находиться не пробелы, как это было на дискете *WinImage*, а именно значение `'NO␣NAME␣␣␣␣'`.

```
07C3E   FA              cli
07C3F   33 C0           xor ax, ax
07C41   8E D0           mov ss, ax
07C43   BC 00 7C        mov sp, $7C00
07C46   16              push ss
07C47   07              pop es
07C48   BB 78 00        mov bx, $0078
07C4B   36 C5 37        lds si, dword [ss:bx]
07C4E   1E              push ds
07C4F   56              push si
07C50   16              push ss
07C51   53              push bx
07C52   BF 3E 7C        mov di, $07C3E
07C55   B9 0B 00        mov cx, $000B
07C58   FC              cld
07C59   F3 A4           rep movsb
```

После настройки стека загрузчик *MS-DOS* загружает в регистры `DS:SI` значение, находящееся по адресу `0000:0078`. Это элемент таблицы векторов прерываний с индексом `1Eh`, который используется для хранения адреса структуры данных, которая называется *Diskette Parameter Table* (таблица параметров дискет) и инициализируется *BIOS*. Приведённый фрагмент кода копирует эту таблицу (11 байт) в память по адресу `0000:7C3E`, перезаписывая таким образом часть уже выполнившегося кода.

```
07C5B   06              push es
07C5C   1F              pop ds
07C5D   C6 45 FE 0F     mov byte [di - 2], $0F
07C61   8B 0E 18 7C     mov cx, [$7C18]
07C65   88 4D F9        mov [di - 7], cl
07C68   89 47 02        mov [bx + 2], ax
07C6B   C7 07 3E 7C     mov word [bx], $7C3E
07C6F   FB              sti
```

Байт по смещению `di - 2` соответствует полю, задающему время ожидания установки головок дисковода. Для дискет формата 3,5″ (образ именно такой дискеты рассматривается в примере) это значение обычно равно `0Fh`. По смещению `di - 7` находится поле, задающее количество секторов в одной дорожке. Сюда записывается значение из регистра `CL`, который инициализируется в результате чтения значения поля `BPB_SecPerTrk` в регистр `CX`. Далее, поскольку `BX` по-прежнему содержит смещение вектора `1Eh` в таблице векторов прерываний, а `DS` уже равен `0000h`, происходит подмена записанного там адреса адресом модифицированной копии таблицы параметров дискет.

```
07C70   CD 13           int 13h
07C72   72 79           jc +$79          ; -> 07CED
```

Далее производится переинициализация дисковой подсистемы *BIOS* с переходом по физическому адресу `07CED` в случае ошибки.

```
07C74   33 C0           xor ax, ax
07C76   39 06 13 7C     cmp [$7C13], ax
07C7A   74 08           je +$08          ; -> 07C84
07C7C   8B 0E 13 7C     mov cx, [$7C13]
07C80   89 0E 20 7C     mov [$7C20], cx
```

По адресу `0000:7C13`, куда обращается данный фрагмент кода, находится поле `BPB_TotSec16`. Поля `BPB_TotSec16` и `BPB_TotSec32` в *FAT* используются в паре: в одном из них записывается 0, в другом – количество секторов на диске. Нулевое значение в поле `BPB_TotSec16` означает, что количество секторов больше, чем может быть записано 16-битным числом, и находится в поле `BPB_TotSec32`, располагающемся в данный момент по адресу `0000:7C20`.Таким образом, в результате выполнения данного фрагмента кода общее количество секторов независимо от размеров этого числа можно будет найти в поле `BPB_TotSec32`.

```
07C84   A0 10 7C        mov al, [$7C10]
07C87   F7 26 16 7C     mul word [$7C16]
07C8B   03 06 1C 7C     add ax, [$7C1C]
07C8F   13 16 1E 7C     adc dx, [$7C1E]
07C93   03 06 0E 7C     add ax, [$7C0E]
07C97   83 D2 00        adc dx, 0
07C9A   A3 50 7C        mov [$7C50], ax
07C9D   89 16 52 7C     mov [$7C52], dx
07CA1   A3 49 7C        mov [$7C49], ax
07CA4   89 16 4B 7C     mov [$7C4B], dx
```

Значение поля `BPB_NumFATs` (адрес `0000:7C10`) загружается в `AX`, после чего умножается на значение поля `BPB_FATSz16` (адрес `0000:7C16`). К полученному размеру одной таблицы размещения файлов (всего их на данном диске две), оказавшемуся в `DX:AX`, прибавляются значения полей `BPB_HiddSec` (4-байтовое значение по адресу `0000:7C1C`) и `BPB_RsvdSecCnt` (2 байта по адресу `0000:7C0E`). Результатом вычислений является количество секторов от начала диска до начала корневой директории, т. е. её *LBA*. Это число записывается по адресам `0000:7C50` и `0000:7C49` (до этого там находились уже выполненные ранее инструкции).

```
07CA8   B8 20 00        mov ax, 32
07CAB   F7 26 11 7C     mul word [$7C11]
07CAF   8B 1E 0B 7C     mov bx, [$7C0B]
07CB3   03 C3           add ax, bx
07CB5   48              dec ax
07CB6   F7 F3           div bx
07CB8   01 06 49 7C     add [$7C49], ax
07CBC   83 16 4B 7C 00  adc [$7C4B], 0
```

Следующим этапом в работе загрузчика *MS-DOS* является вычисление *LBA* области данных. Для этого размер элемента директории (32 байта) умножается на количество элементов в корневой директории, записанное в поле `BPB_RootEntCnt` (адрес `0000:7C11`). Полученный размер корневой директории в байтах переводится в количество занимаемых ею секторов. При этом, если размер корневой директории вопреки спецификации *FAT* не кратен размеру сектора, производится округление в бóльшую сторону. Результат прибавляется к ранее записанному по адресу `0000:7C49` 4-байтовому *LBA* корневой директории.

Следует отметить, что данный фрагмент кода написан в расчёте, что в результате умножения в регистре `DX` окажется 0, т. е. что размер корневой директории составляет менее 64 Кбайт. В противном случае возможно получение некорректных значений при округлении. Спецификация *FAT* таких гарантий не даёт, поэтому в данном случае можно говорить об упрощении (и уменьшении в объёме) кода загрузчика за счёт использования недокументированных особенностей используемых на практике параметров *FAT*-дисков.

```
07CC1   BB 00 05        mov bx, $0500
07CC4   8B 16 52 7C     mov dx, [$7C52]
07CC8   A1 50 7C        mov ax, [$7C50]
07CCB   E8 92 00        call +$0092      ; -> 07D60
07CCE   72 1D           jc +$1D          ; -> 07CED
```

После инициализации регистра `BX` значением `500h` и загрузки в `DX:AX`, ранее сохранённого по адресу `0000:7C50` *LBA* корневой директории, происходит вызов процедуры по физическому адресу `07D60`.

```
07D60   3B 16 18 7C     cmp dx, [$7C18]
07D64   73 19           jae +$19         ; -> 07D7F
07D66   F7 36 18 7C     div word [$7C18]
07D6A   FE C2           inc dl
07D6C   88 16 4F 7C     mov [$7C4F], dl
07D70   33 D2           xor dx, dx
07D72   F7 36 1A 7C     div word [$7C1A]
07D76   88 16 25 7C     mov [$7C25], dl
07D7A   A3 4D 7C        mov [$7C4D], ax
07D7D   F8              clc
07D7E   C3              ret

07D7F   F9              stc
07D80   C3              ret
```

Данная процедура осуществляет преобразование *LBA*, записанного в `DX:AX`, в *CHS* соответствующего сектора. При этом номер сектора (остаток от деления на значения поля `BPB_SecPerTrk` по адресу `0000:7C18`) записывается как 1-байтовая величина по адресу `0000:7C4F`, а номера головки и цилиндра (получаемые в результате деления ранее полученного частного на значение поля `BPB_NumHeads` по адресу `0000:7C1A`) – соответственно как 1-байтовая величина по адресу `0000:7C25` и 2-байтовая величина по адресу `0000:7C4D`.

Если преобразование успешно выполнено, процедура возвращает управление с обнулённым флагом `CF`. Если же преобразование невозможно (значение *LBA* слишком велико, чтобы приведённый фрагмент кода мог выполнить его корректно), управление возвращается с установленным в единицу флагом `CF` и без изменения каких-либо значений в памяти.

```
07CD0   B0 01           mov al, $01
07CD2   E8 AC 00        call +$00AC      ; -> 07D81
07CD5   72 16           jc +$16          ; -> 07CED
```

При успешном преобразовании загрузчик осуществляет вызов ещё одной процедуры с физическим адресом `07D81`, предварительно помещая в регистр `AL` значение 1.

```
07D81   B4 02           mov ah, $02
07D83   8B 16 4D 7C     mov dx, [$7C4D]
07D87   B1 06           mov cl, 6
07D89   D2 E6           shl dh, cl
07D8B   0A 36 4F 7C     or dh, [$7C4F]
07D8F   8B CA           mov cx, dx
07D91   86 E9           xchg cl, ch
07D93   8A 16 24 7C     mov dl, [$7C24]
07D97   8A 36 25 7C     mov dh, [$7C25]
07D9B   CD 13           int 13h
07D9D   C3              ret
```

В этой процедуре осуществляется вызов функции `int 13h/02h` с использованием ранее вычисленных *CHS*-значений. Количество считываемых секторов берётся из регистра `AL`.

В качестве номера диска указывается значение поля `BS_DrvNum`, записанное на дискете. Строго говоря, это не самое удачное решение, т. к. в теории номер диска, с которого осуществляется загрузка, может отличаться от записываемого в это поле значения (`00h` для дискет). На практике же, как правило, *BIOS* назначает наименьший номер именно загрузочному накопителю, поэтому чаще всего подобный код оказывается работоспособным. Впрочем, записав значение `DL` при передаче управления загрузчику, можно было бы существенно повысить надёжность его работы.

```
07CD7   8B FB           mov di, bx
07CD9   B9 0B 00        mov cx, 11
07CDC   BE E6 7D        mov si, $7DE6
07CDF   F3 A6           repe cmpsb
07CE1   75 0A           jne +$0A          ; -> 07CED
07CE3   8D 7F 20        lea di, [bx + 32]
07CE6   B9 0B 00        mov cx, 11
07CE9   F3 A6           repe cmpsb
07CEB   74 18           je +$18           ; -> 07D05
```

В случае успешного чтения первого сектора корневой директории производится сравнение её первых 11 байт с 11 байтами по адресу `0000:7DE6`. Таким образом проверяется, соответствует ли первая запись корневой директории файлу `IO.SYS`. Если это так, следующая (вторая по счёту) запись корневой директории подвергается аналогичной проверке: на этот раз ожидается файл `MSDOS.SYS`.

```
07CED   BE 9E 7D        mov si, $7D9E
07CF0   E8 5F 00        call +$005F      ; -> 07D52
07CF3   33 C0           xor ax, ax
07CF5   CD 16           int 16h
07CF7   5E              pop si
07CF8   1F              pop ds
07CF9   8F 04           pop [si]
07CFB   8F 44 02        pop [si + 2]
07CFE   CD 19           int 19h
```

На следующий фрагмент кода загрузчик передаёт управление каждый раз, когда возникает какая-либо ошибка: при переинициализации дисковой подсистемы *BIOS*, преобразовании *LBA* в *CHS*, чтении секторов с диска и т. п. После вызова процедуры, располагающейся по физическому адресу `07D52`, следует ожидание нажатия клавиши (в отличие от загрузчика *WinImage* здесь не производится очистка клавиатурного буфера), затем из стека извлекаются значения, сохранённые туда в самом начале работы загрузчика. Первые два значения на вершине стека – полный адрес элемента `1Eh` в таблице векторов прерываний, вследствие чего в `DS:SI` оказывается адрес `0000:0078`. Следующие два элемента в стеке содержат адрес, который был записан в этом элементе таблицы векторов прерываний в момент передачи управления загрузчику, – это значение восстанавливается.

Следует отметить, что в начале работы загрузчика обработка прерываний остаётся отключённой до окончания записи в таблицу векторов прерываний. В целом большого смысла это не имеет, т. к. маловероятно, что какой-либо код, кроме самого загрузчика, станет перезаписывать вектор `1Eh`, а сам вектор будет использоваться не для обработки аппаратных прерываний. При завершении работы загрузчик выполняет перезапись этого вектора без отключения прерываний. Такое поведение может указывать на то, что отключение обработки прерываний в начале производилось с другой целью – для перенастройки стека. Однако, поскольку использованные для этого инструкции гарантируют атомарность, это также является избыточным. Вероятнее всего, манипуляции с флагом `IF` нужны для совместимости с моделями *x*86-совместимых процессоров, которые подобных гарантий не обеспечивают.

В случае возникновения ошибок работа загрузчика завершается вызовом сервиса `int 19h`.

```
07D52   AC              lodsb
07D53   0A C0           or al, al
07D55   74 29           jz +$29          ; -> 07D80
07D57   B4 0E           mov ah, $0E
07D59   BB 07 00        mov bx, $0007
07D5C   CD 10           int 10h
07D5E   EB F2           jmp -$0E         ; -> 07D52

...     ...             ...

07D80   C3              ret
```

Процедура по адресу `07D52` осуществляет вывод на экран *C*-строки, адрес которой находится в `DS:SI`. При этом вместо отдельной инструкции `ret` переиспользуется та же, что и в процедуре преобразования *LBA* в *CHS*.

В случае завершения работы загрузчика из-за ошибки этой процедуре передаётся строка по адресу `0000:7D9E`:

```
Non-System disk or disk error
Replace and press any key when ready
```

Если же в корневой директории оба требуемых файла обнаружены на своих позициях, управление передаётся физическому адресу `07D05`. При этом пропускается небольшой фрагмент кода:

```
07D00   58              pop ax
07D01   58              pop ax
07D02   58              pop ax
07D03   EB E8           jmp -$18          ; -> 07CED
```

Как можно заметить, эта часть загрузчика представляет собой переход по тому же адресу, куда передавалось управление при возникновении ошибок, однако теперь – с извлечением трёх элементов с вершины стека.

```
07D05   8B 47 1A        mov ax, [bx + 26]
07D08   48              dec ax
07D09   48              dec ax
07D0A   8A 1E 0D 7C     mov bl, [$7C0D]
07D0E   32 FF           xor bh, bh
07D10   F7 E3           mul bx
07D12   03 06 49 7C     add ax, [$7C49]
07D16   13 16 4B 7C     adc dx, [$7C4B]
```

К началу выполнения этого фрагмента кода в `BX` содержится смещение прочитанного в память первого сектора корневой директории, т. е. фактически – смещение первой записи в корневой директории. Передача управления сюда означает, что первая запись в корневой директории соответствует файлу `IO.SYS`. Следовательно, обращение к памяти по смещению `bx + 26` загружает в `AX` значение поля `DIR_FstClusLO` (для *FAT*12 и *FAT*16 в поле `DIR_FstClusHI` всегда записан 0).

Из номера первого кластера файла `IO.SYS` вычитается 2, затем полученный результат умножается на значение поля `BPB_SecPerClus` (адрес `0000:7C0D`). К результату умножения – смещению в секторах от начала области данных до начала кластера – прибавляется *LBA* области данных, сохранённый ранее в четырёх байтах по адресу `0000:7C49`. Таким образом, в `DX:AX` оказывается *LBA* первого кластера файла `IO.SYS`.

```
07D1A   BB 07 00        mov bx, $0700
07D1D   B9 03 00        mov cx, $0003

07D20   50              push ax
07D21   52              push dx
07D22   51              push cx
07D23   E8 3A 00        call +$003A      ; -> 07D60
07D26   72 D8           jc +$D8          ; -> 07D00
07D28   B0 01           mov al, $01
07D2A   E8 54 00        call +$0054      ; -> 07D81
07D2D   59              pop cx
07D2E   5A              pop dx
07D2F   58              pop ax
07D30   72 BB           jc -$45          ; -> 07CED
07D32   05 01 00        add ax, 1
07D35   83 D2 00        adc dx, 0
07D38   03 1E 0B 7C     add bx, [$7C0B]
07D3C   E2 E2           loop -$1E        ; -> 07D20
```

Далее следует цикл из трёх итераций, в котором по адресу `0000:0700` последовательно считывается три сектора. Так как для данного диска размер кластера равен одному сектору, загрузчик фактически исходит из предположения, что по меньшей мере первые 1,5 Кбайт файла `IO.SYS` записаны в последовательных кластерах. Это достаточно смелое предположение, которое легко может быть нарушено, если удалить указанный файл, а затем попробовать повторно записать его на заполненный данными и сильно фрагментированный диск. Впрочем, такое использование загрузочной дискеты нетипично, поэтому обычно этот подход срабатывает.

```
07D3E   8A 2E 15 7C     mov ch, [$7C15]
07D42   8A 16 24 7C     mov dl, [$7C24]
07D46   8B 1E 49 7C     mov bx, [$7C49]
07D4A   A1 4B 7C        mov ax, [$7C4B]
07D4D   EA 00 00 70 00  jmp $0070:$0000
```

Последнее, что делает загрузчик в случае успешного считывания трёх секторов, – загрузка нескольких значений в регистры общего назначения и передача управления на начало загруженного с диска содержимого файла `IO.SYS`. В табл. 2.6 указаны значения регистров в момент передачи управления коду из этого файла.

Таблица 2.6

Исходные данные для запуска `IO.SYS` в *MS-DOS* 6.22

Регистр	Описание
`CH`	Значение поля `BPB_Media`
`DL`	Значение поля `BS_DrvNum`
`AX:BX`	*LBA* области данных

Таким образом, загрузчик *MS-DOS* 6.22 выполняет ограниченный разбор структур данных файловой системы. В частности, предполагается, что загружаемый файл `IO.SYS` находится непосредственно в корневой директории и имеет специфическое расположение на диске, а информация о нём содержится в самой первой записи корневой директории. В вычислениях и при работе с диском также используется ряд допущений, которые не всегда могут соответствовать действительности. Так, например, игнорируется возможность повторения операции чтения в случае ошибки, никак не используется таблица размещения файлов.

Как написать свой загрузчик для *FAT*?

Разработка загрузчика, который бы производил чтение содержимого определённого файла на диске «честно», т. е. не опираясь на допущения, которые могут оказываться ошибочными, – нетривиальная задача. Как следует из рассмотренных ранее примеров, даже при разработке коммерческих продуктов её значительно упрощают.

При разработке загрузчиков для учебной операционной системы ***«OSE!»*** приняты три допущения, которые по сути представляют собой требования к загружаемому файлу:

– файл должен находиться в корневой директории;

– файл должен иметь размер, позволяющий загрузить его целиком в первые 32 Кбайт ОЗУ;

– файл должен быть расположен на диске так, чтобы его можно было загрузить с использованием доступного набора функций *BIOS*.

При соблюдении этих трёх условий и требований спецификации *FAT* загрузчики ***«OSE!»*** будут работать корректно, обеспечивая запуск операционной системы. Фактически можно говорить о том, что в загрузчике ***«OSE!»*** удалось реализовать простейший драйвер файловой системы, обеспечивающий только чтение с диска и только из корневой директории. Впрочем, запись на диск из загрузчика вряд ли может понадобиться, а ограниченность только корневой директорией – не слишком существенная проблема, т. к. речь идёт лишь об одном файле.

Следует также заметить, что в загрузочных секторах ***«OSE!»*** не используются приёмы сжатия кода: потенциально с их помощью можно сделать загрузчики ещё более гибкими, однако накладные расходы на распаковку кода могут стать серьёзным ограничением на этом пути. Для того чтобы поместить весь необходимый код в загрузочный сектор, используется ряд приёмов, с которыми можно подробнее ознакомиться в исходных кодах операционной системы.

Также в учебных целях код загрузчика ***«OSE!»*** использует только инструкции, доступные в процессорах *Intel* 8086 и *Intel* 8088, что приводит к некоторому увеличению объёма кода. В частности, в ранних *x*86-процессорах отсутствовала инструкция `push` для константного значения, величину побитовых сдвигов можно было задавать либо равной 1, либо через регистр `CL`, но не опе-

рандом-константой, инструкциями условного перехода поддерживались только ближние переходы (на расстояние от –128 до +127 байт) и т. п.

При разработке собственного загрузчика в качестве минимальных системных требований можно выбрать более позднюю модель процессора, например, 80386 или 80586. Также не стоит пытаться поддержать в одном загрузчике несколько различных файловых систем или их версий. Кроме того, имеет смысл пойти на ряд других упрощений. Некоторые из них перечислены далее в порядке от менее критичных к наиболее нежелательным по мнению авторов данного пособия.

Использование фиксированного размера сектора. Размеры сектора, поддерживаемые *FAT*, являются степенями двойки, что позволяет вместо громоздких инструкций умножения, затрагивающих ряд регистров общего назначения, использовать инструкции побитового сдвига. Заранее известный размер сектора позволяет уйти от вычисления величины сдвигов и задавать её константами.

Использование фиксированной геометрии диска, размеров кластера и т. п. Отказ от учёта значений соответствующих полей в загрузочном секторе позволяет, во-первых, избежать обращений к памяти, машинный код для которых, как правило, длиннее, чем при работе с операндами-регистрами и операндами-константами, а во-вторых, в ряде случаев упростить и саму логику работы загрузочного сектора.

Использование ограничений для расположения информации о загружаемом файле. По аналогии с загрузчиком *MS-DOS* можно потребовать, чтобы загружаемый файл был представлен первым элементом корневой директории.

Использование заранее известной информации о загружаемом файле. Сюда можно отнести как используемое загрузчиком *MS-DOS* допущение о последовательном размещении кластеров файла, так и, например, отказ от использования значения в поле `DIR_FileSize`, загрузку фиксированного количества данных независимо от реального размера файла по данным файловой системы, использование фиксированного номера первого кластера и т. п.

Отказ от повторных попыток чтения данных с диска в случае ошибки. При небольшом уменьшении объёма кода этот приём существенно снижает надёжность работы загрузчика при использовании изношенных дисков и устройств для их чтения.

Использование фиксированных номеров и количества секторов. Поскольку загружаемый файл, как правило, записывается на чистый диск одним из первых, чаще всего он оказывается нефрагментированным (занимающим непрерывную группу секторов), а его положение на диске известно заранее. Опираясь на эти допущения, можно свести работу загрузчика к чтению группы секторов. Впрочем, при этом любые изменения в содержимом диска могут нарушить работу загрузчика. По сути, в данном случае загрузчик перестаёт привязываться к той или иной файловой системе, а сама файловая система используется лишь для того, чтобы защитить занятые загружаемым файлом сектора от случайной перезаписи при работе с диском.

Подведение итогов

Для повышения удобства разработки, а также для обеспечения возможности обмена данными с существующими операционными системами целесообразно поддерживать в своей операционной системе одну или несколько популярных файловых систем, в том числе и в коде загрузочного сектора. Наиболее широко используемым и простым в поддержке семейством файловых систем является семейство *FAT*.

Разработка загрузчика является искусством поиска компромиссов: реализовать в рамках загрузочного сектора полноценную поддержку той или иной файловой системы крайне проблематично, однако чем точнее загрузчик будет следовать спецификации и чем больше возможностей файловой системы он будет поддерживать, тем выше будет вероятность успешной загрузки операционной системы при различных конфигурациях компьютеров и тем удобнее будет проводить её отладку.

Задания

1. Разработайте собственный загрузчик для одной из файловых систем семейства *FAT*. Для тестирования используйте файл, в котором записан код вывода сообщения об успешном запуске операционной системы.
2. Напишите программу, которая позволит создавать и изменять образы *FAT*-дисков (аналог *WinImage*).
3. Напишите программу, которая позволит восстанавливать удалённые с *FAT*-диска файлы (аналог утилиты `undelete` в *MS-DOS*).
4. Разработайте набор макросов *FASM* для создания образов *FAT*-дисков.
5. Разработайте обучающую программу, позволяющую пользователю в игровой форме изучить принципы работы файловых систем семейства *FAT*.

Шаг 3. Основные модули ядра операционной системы

После того как разработан загрузчик, можно переходить к разработке непосредственно ядра – модуля, который реализует в себе основную функциональность операционной системы. Однако в отличие от загрузчика – модуля операционной системы, который при необходимости сравнительно легко заменить, – ядро во многом определяет, каким образом будет происходить вся дальнейшая разработка. Поэтому именно на этом этапе особенно важно составить представление о том, какой будет операционная система изначально и как она будет развиваться в дальнейшем.

На какие вопросы нужно ответить, приступая к разработке ОС?

Важным этапом в разработке любого программного средства является составление спецификации требований – документа, в котором должны быть чётко сформулированы цели разработки и перечислены конкретные характеристики разрабатываемого продукта, составляющие его ценность. При разработке учебной операционной системы детальной формализацией своей задумки можно пренебречь, однако для того, чтобы минимизировать количество ошибок проектирования и избежать многократного переписывания отдельных частей системы, крайне желательно как можно раньше определиться с тем, какой должна быть будущая операционная система.

1. С какой целью разрабатывается операционная система?

Часто целью разработки являются просто повышение собственной квалификации в целом как разработчика и удовлетворение любопытства – в такой ситуации можно принимать решения о способе реализации тех или иных частей ОС произвольным образом. Если же преследуются более конкретные цели, они должны быть учтены. Например, для понимания самых общих принципов достаточно будет разработки ОС для реального режима, которая просто сможет запускать программы, а для того, чтобы поэкспериментировать с многозадачностью, скорее всего стоит присмотреться к программированию таймеров, которые могут быть в системе, и, может быть, даже рассмотреть вопрос об использовании защищённого режима. Ещё один пример – ОС для управления конкретным аппаратным устройством или решения какой-либо узкой прикладной задачи, здесь многое будет зависеть от типа устройства или решаемой задачи.

Например, операционная система ***«OSE!»*** разрабатывается для того, чтобы выступать в роли учебного примера и готового каркаса для экспериментов, причём изначально она предназначена для студентов, знакомых с программированием для реального режима *IA*-32. Исходя из этого первоначальная версия ***«OSE!»*** представляет собой операционную систему наподобие *MS-DOS*, однако с более жёстким следованием спецификациям и более простыми техническими решениями. При этом, с одной стороны, игнорируются устройства, которые

должны были поддерживаться *MS-DOS*, но работали с ошибками, которые этой ОС приходилось «исправлять» самостоятельно, скрывая их от прикладных программ, а с другой – ставится цель охватить все модели процессоров начиная с *Intel* 8086 и *Intel* 8088.

2. Планируется ли развитие ОС и какое?

Если для учебных операционных систем «на один раз» можно обойтись более поверхностным проектированием и в целом вести разработку более «наивным» способом, то для ОС, которая должна стать полноценной альтернативой существующим, крайне желательно уделить внимание таким аспектам, как документирование, наработка вспомогательного инструментария, детальная проработка архитектуры. Немаловажно и то, планируется ли доработка системы в случае появления новых устройств или предоставляемых аппаратурой возможностей, например, новых наборов инструкций в процессорах.

Учебная операционная система ***«OSE!»*** разрабатывалась исходя из предположения, что со временем она будет переориентирована на защищённый режим *IA*-32 и получит поддержку его 64-битного подрежима, сохранив при этом и свой статус наглядного пособия по разработке ОС для реального режима. Это означает, что было необходимо уделить достаточно внимания таким вещам, как возможность расширения форматов исполняемых файлов, механизмов вызова системных функций и т. п., а также ответственно подойти к документированию, чётко выделив те аспекты поведения, которые будут частью контракта между ОС и разработчиками драйверов и прикладного ПО, и те, которые должны рассматриваться как детали реализации.

3. Какие аппаратные средства должна поддерживать ОС?

При разработке ОС в качестве учебного проекта, как правило, достаточно поддержки клавиатурного ввода, вывода информации на экран и работы с дисками. При этом можно ориентироваться на определённый перечень устройств, установленных в конкретном компьютере, или вести разработку исключительно для виртуальной машины, эмулируемой продуктами наподобие *VMWare* или *Bochs*, а прямую работу с устройствами заменить, где возможно, вызовами функций *BIOS*.

Если разработка ОС рассматривается как способ получения прямого доступа к какому-либо конкретному устройству, следует учитывать используемый им интерфейс подключения и, вероятно, предусмотреть прозрачные механизмы взаимодействия с аппаратурой, при использовании которых влияние ОС будет минимальным. Для профессиональной системы, которая должна поддерживать широкий круг устройств, на первый план выходит грамотное проектирование её архитектуры, а также продуманные программные интерфейсы для встраивания драйверов в систему.

В случае ***«OSE!»*** предполагается, что первоначальная версия для реального режима будет выполнять роль учебного примера ОС для реального режима, однако в перспективе система должна обзавестись поддержкой защищённого режима. При этом ожидается, что разработка драйверов для отдельных устройств может быть предложена в качестве учебного задания студентам. По этим причинам, с одной

стороны, основные компоненты ***«OSE!»*** поддерживают ранние модели процессоров, с другой – делается акцент на модульности системы.

4. Нужна ли поддержка выполнения программ для других ОС?

Как правило, ответ на этот вопрос утвердительный. Ценность операционной системы обычно во многом определяется доступностью для неё прикладных программ, позволяющих решать реальные задачи пользователя. Поддержка исполняемых файлов, скомпилированных для других операционных систем, позволяет быстро получить такой набор программ. Сложность при этом заключается в необходимости реализации в ОС некоторых алгоритмов и структур данных, аналогичных используемым в других ОС. Зачастую это довольно трудозатратно и даже накладывает отпечаток на разрабатываемую ОС, в том числе ограничивая круг технических решений, которые могут быть в ней применены.

Для первоначальной версии ***«OSE!»*** целью является частичная поддержка программ, написанных для *MS-DOS*. По этой причине в её состав входят функции, работающие аналогично функциям *MS-DOS*, а формат исполняемых файлов спроектирован так, чтобы программы могли корректно завершать выполнение, если будут по ошибке запущены под управлением *MS-DOS*.

5. Какие возможности ОС должна предоставлять прикладным программам?

Важно определиться, какие именно технические возможности следует реализовать на уровне операционной системы, а какие оставить для реализации прикладным программам. Например, если планируется разработка профессиональной ОС, имеет смысл задуматься о проработке функций, связанных с мультимедиа, а также о поддержке пользователей с ограниченными возможностями, причём заложить это уже на ранних этапах разработки. Если предполагаемое применение будущей системы – предоставление сетевых сервисов (например, запуск на ней *web*-серверов), стоит особое внимание уделить проработке стека сетевых протоколов и предоставить прикладным программам доступ к разным его уровням. Немаловажно также определиться и с тем, должна ли операционная система поддерживать многозадачность и в каком виде, т. к. это оказывает существенное влияние не только на набор предоставляемых приложениям функций, но и на структуры данных, используемые самой операционной системой.

Применительно к ***«OSE!»*** каких-либо особых требований первоначально не выдвигается, однако, поскольку планируется дальнейшее развитие, везде, где это технически осуществимо, оставляется пространство для расширения возможностей.

Какую архитектуру использовать для операционной системы?

Традиционно выделяют два основных вида архитектуры:

- микроядро;
- монолитное ядро.

Данная классификация основана на том, как организовано взаимодействие основной части ядра операционной системы, обеспечивающей общую

координацию, с теми частями, которые отвечают за организацию работы с устройствами (как физическими, так и виртуальными) – драйверами.

В случае выбора монолитного ядра все его части работают в одном общем адресном пространстве и, как правило, на одном уровне привилегий. Классические монолитные ядра представляли собой полностью неделимый программный компонент и требовали перекомпиляции при изменении состава оборудования, однако в настоящее время принято считать монолитными и те ядра, в которых драйверы загружаются по мере необходимости.

Микроядерная архитектура напротив предполагает максимальное отделение основной части ядра от драйверов. Фактически в этом случае ядро решает только самые базовые задачи – занимается координацией работы отдельных модулей и содержит лишь необходимый минимум кода для работы с аппаратурой, – а вся остальная логика выносится в явным образом обособленные драйверы. При этом предполагается, что адресное пространство ядра недоступно драйверам напрямую.

Давая оценку этим двум, по сути диаметрально противоположным подходам, следует отметить, во-первых, потенциально более высокую производительность монолитных ядер, во-вторых, потенциально более высокую надёжность микроядер и, в-третьих, большой потенциал для использования всевозможных смешанных подходов.

Среди популярных и просто известных операционных систем есть представители всех трёх подходов: примерами монолитных ядер считаются операционные системы *UNIX*, *Linux*, ранние версии *Windows* и даже *MS-DOS*, примерами микроядер – *Symbian*, *Windows CE* и *Minix*. Различия между смешанными подходами зачастую совершенно незначительны: например, гибридные и модульные ядра обычно считаются подвидами монолитных ядер, но с разной степенью обособленности модулей.

На практике стремление к чёткому следованию одной из архитектур имеет смысл разве что в академических целях, для изучения преимуществ и недостатков подходов, а во всех прочих случаях на первый план выходит техническая целесообразность: какие части операционной системы должны быть обособленными, а какие – встроенными, какие имеются ограничения в части доступных ресурсов и т. д. Важно заметить, что в реальном режиме *IA*-32 отсутствует аппаратная поддержка разграничения адресных пространств и управления привилегиями: любой код выполняется с одинаковыми правами и может выполнять любые действия, доступные в этом режиме. Исходя из этого, если придерживаться строгих определений, для реального режима возможна реализация только различных разновидностей монолитных ядер.

Намного важнее условных классификаций сам характер взаимосвязей между частями операционной системы. Среди различных подходов, доступных при разработке системы для реального режима, можно выделить несколько наиболее типичных. Перечислим их в порядке возрастания сложности реализации и гибкости конечного результата.

Монолитная ОС. Считываемый загрузчиком файл содержит в себе код, реализующий всю функциональность операционной системы. Не только логика работы с устройствами и управления памятью, но и пользовательский интерфейс жёстко прописаны в файле с ядром и могут быть изменены только путём перекомпиляции. По сути, единственная функция такой ОС – запуск прикладных программ и создание им условий для выполнения. Основное преимущество такой архитектуры – относительная простота реализации. Главный недостаток – сложность в расширении возможностей операционной системы. Применение такого подхода целесообразно либо при наличии очень серьёзных ограничений в ресурсах, либо в образовательных и экспериментальных целях.

Монолитная ОС с отдельной *UI*-оболочкой. Устройство аналогично монолитной ОС, но вместо встроенного прямо в ядро пользовательского интерфейса выделяется специальная программа – оболочка, – которая автоматически запускается после того, как завершена инициализация системы. Вся прочая логика по-прежнему содержится в ядре. Такая архитектура характеризуется чуть большей гибкостью: например, при реализации интерфейса командной строки набор доступных команд можно менять, заменяя одну оболочку другой. Появляется возможность запуска нескольких программ синхронно: программа А может запустить программу Б и, дождавшись её завершения, продолжить работу. Тем не менее расширение возможностей такой системы по-прежнему затруднено.

ОС с поддержкой драйверов. Взаимодействие операционной системы с аппаратурой организовано не напрямую, а в виде загружаемых модулей – драйверов. Ядро содержит лишь минимальный объём кода, необходимый для инициализации. В ходе инициализации в память загружаются драйверы, предназначенные для фактически имеющихся в системе устройств. Драйверы реализованы как отдельные исполняемые файлы, предоставляющие операционной системе набор функций, вызывая которые она может задействовать аппаратное обеспечение и возможности, которые изначально в ней не были заложены: работать с новыми устройствами, поддерживать сторонние файловые системы и т. п.

Перечисленные подходы можно рассматривать и как этапы добавления возможностей в операционную систему для реального режима: первоначально можно реализовывать всю логику работы системы прямо в ядре, затем постепенно выносить в отдельные исполняемые файлы и загружать по запросу. Однако следует понимать, что уже на ранних этапах желательно продумывать программный интерфейс драйвера, формат исполняемого файла и прочие сопутствующие технические решения. В противном случае при внесении даже незначительных изменений много времени будет тратиться на то, чтобы адаптировать уже разработанные драйверы к обновлённым правилам взаимодействия.

Учебная операционная система ***«OSE!»*** реализует модульную архитектуру: подавляющее большинство возможностей реализовано в виде отдельных модулей, большая часть которых может быть заменена без перекомпиляции ядра. В частности, отдельными драйверами реализовано взаимодействие с дисками, работа с файловыми системами и т. п.

С чего начинает работу ядро операционной системы?

Основной задачей ядра операционной системы является координация работы модулей ОС и прикладных программ с имеющимися ресурсами. При этом она может заключаться не только в определении, каким модулям и программам следует передать в пользование какую часть ресурсов, но также и в предоставлении программного интерфейса, позволяющего абстрагироваться от конкретных способов доступа к этим ресурсам.

Следует также понимать, что организация работы с ресурсами во многом зависит от того, какой режим работы процессора выбран для операционной системы основным и какой уровень взаимодействия с аппаратурой предполагается обеспечивать. В данном пособии эта тема рассматривается применительно к работе с реальным режимом (*real-address mode*) процессоров архитектуры *IA*-32. При всей ограниченности возможностей, доступных в этом режиме, это позволяет в первую очередь продемонстрировать общие принципы и подходы, не отвлекаясь на особенности настройки процессора в защищённом режиме и его подрежимах.

Оперативная память. Ресурс, управления которым практически невозможно избежать разработчику операционной системы – оперативная память. Даже если ОС будет строго однозадачной и ориентированной на решение узкого круга проблем, необходимость реализации хотя бы простейшего механизма управления памятью сохранится. Минимальная задача при этом – определить, какое количество памяти доступно для использования операционной системой и прикладными программами, и отслеживать, какие её участки доступны для использования (свободны), а какие принадлежат тому или иному компоненту.

Дисковые накопители и файлы. В задачи даже простейших операционных систем входит организация выполнения прикладных программ. В более сложных системах помимо прикладных программ используется также особый вид исполняемых файлов – драйверы. Кроме того, для повышения гибкости и расширяемости могут использоваться файлы настроек, содержимое которых должно учитываться ядром операционной системы, другими её компонентами, драйверами и прикладными программами. Поддержка более простого (и высокоуровневого) интерфейса, чем предлагает *BIOS*, для работы с файлами, записанными на дисковых накопителях, позволяет существенно упростить разработку прикладных программ для операционной системы, а также её настройку.

Процессорное время. В случае разработки многозадачной операционной системы одним из ресурсов, подлежащих управлению, становится время, которое отводится программам для выполнения. В зависимости от выбранной модели многозадачности управление прочими ресурсами, например, оперативной памятью, может организовываться по-разному.

Другие аппаратные ресурсы. В то время как для учебной операционной системы может быть достаточно поддержки минимального набора аппаратуры – для ввода и отображения информации, – при разработке более профессиональных систем требуется поддержка самого разнообразного аппаратного

обеспечения. При этом ОС берёт на себя задачу учёта подключённых устройств и задействованных ими ресурсов, таких как линии запроса прерываний (*IRQ*), каналы для передачи данных (*DMA*) и т. п.

Таким образом, первоочередной задачей ядра операционной системы, получившего управление от загрузчика, будет инициализация структур данных, необходимых для управления ресурсами. Чем быстрее все возможности ядра будут готовы к использованию, тем в большей части ядра их можно будет переиспользовать, что позволит в том числе снизить сложность тестирования.

Следует, однако, иметь в виду, что уже сама инициализация одних структур данных будет требовать наличия других структур, готовых к работе. При этом возможно возникновение проблемы «курицы и яйца»: например, если модуль, отвечающий за управление аппаратурой, должен считывать настройки из файла (например, ранее сохранённые пользователем предпочтения относительно настройки отдельных устройств), ему потребуется механизм для работы с файлами, а также память, в которую содержимое файла настроек будет считываться для анализа. При этом файлы, как правило, располагаются на дисковых накопителях, которые сами по себе также являются внешними устройствами, за управление которыми должен отвечать тот самый модуль.

Решение проблемы может заключаться в инициализации некоторых возможностей операционной системы в несколько этапов: сначала использовать одну из возможных реализаций (например, воспользоваться специализированным фрагментом кода для чтения файла с конкретного диска, встроенным в само ядро), а затем, по мере готовности, переключаться на использование другой реализации (например, на загруженный драйвер для работы с конкретным видом накопителей).

Главное, что необходимо помнить: не существует единственного правильного подхода. Одна и та же задача может быть решена разными способами и каждый из них может оказаться оптимальным для каких-либо условий. Рассматриваемые далее идеи и подходы не являются обязательным руководством к действию, а представляют собой лишь примеры того, как можно реализовать те или иные возможности. Не следует бояться отступления от описываемых подходов в своей операционной системе.

Пример 3.1. Разработка простого менеджера памяти

Главный ресурс, управление которым должна принять на себя операционная система, – это оперативная память. Модуль, который обеспечивает учёт занятых (используемых) и свободных блоков памяти, принято называть менеджером памяти. Само по себе это понятие не привязано к разработке операционных систем: собственный менеджер памяти можно найти практически в каждой программе, написанной на языке программирования высокого уровня, и там он решает примерно ту же задачу. Тем не менее разработка менеджера памяти ОС имеет свою специфику.

Основное отличие заключается в том, менеджер памяти, встроенный в программу, как правило, работает с непрерывным блоком достаточно большого размера, в то время как менеджер памяти ОС имеет дело со всей доступной оперативной памятью, часть которой уже занята данными и кодом *BIOS*, областями, в которые спроецирована память внешних устройств, и т. п. Как правило, все подобные низкоуровневые нюансы стараются учесть в менеджере памяти ОС, чтобы впоследствии упростить разработку прикладного ПО и драйверов. Кроме того, в защищённом режиме *IA*-32 (и его 64-битном подрежиме) на менеджер памяти ОС ложится также задача управления страничной адресацией (механизмом виртуальной памяти), что представляет собой отдельную, довольно нетривиальную задачу, ввиду чего нередко в операционных системах, работающих в защищённом режиме, эта логика выносится в отдельный менеджер виртуальной памяти.

Рассмотрим процесс разработки простого менеджера памяти для операционной системы, работающей в реальном режиме, на примере учебной операционной системы ***«OSE!»***.

Начнём с решения вопроса о том, какой именно памятью нужно будет управлять. В табл. 3.1 приведена карта памяти в момент передачи управления ядру ***«OSE!»*** (при использовании параметров сборки по умолчанию).

Таблица 3.1

Примерная карта памяти в момент передачи управления ядру ***«OSE!»***

Диапазон физических адресов	**Размер**	**Использование**	**Размещение**
00000h–003FFh	1 Кбайт	Таблица векторов прерываний	ОЗУ
00400h–004FFh	256 байт	Область данных *BIOS* (*BDA, BIOS Data Area*)	
00500h–005FFh	256 байт	Свободно (кроме нескольких байт в начале)	
00600h–006FFh	256 байт	Свободно	
00700h–0XXXXh	29 Кбайт	Ядро ОС	
0XXXXh–07BFFh		Свободно	
07C00h–07DFFh	512 байт	Загрузочный сектор	
07E00h–07FFFh	512 байт	Стек загрузчика	
08000h–7FFFFh	480 Кбайт	Свободно	ОЗУ
80000h–9FFFFh	128 Кбайт	Расширенная область данных *BIOS* (*EBDA, Extended BIOS Data Area*)	
A0000h–BFFFFh	128 Кбайт	Видеопамять	Другие устройства
C0000h–C7FFFh	32 Кбайт	*BIOS* видеоадаптера	
C8000h–EFFFFh	160 Кбайт	Расширения *BIOS*	
F0000h–FFFFFh	64 Кбайт	*BIOS* материнской платы	

Ядро ***«OSE!»*** загружается по физическому адресу `00700h`, а предыдущие 256 байт, начиная с физического адреса `00600h`, отведены для хранения глобальных данных ядра, по аналогии с *BDA*. Эта область в терминологии ***«OSE!»*** называется *SDA – System Data Area*. Поскольку размер файла ядра может изменяться в ходе разработки, удобнее отводить память для *SDA* именно перед ядром, чтобы положение этих данных оставалось фиксированным: это удобно и при отладке, и при реализации процедур, которые будут с этими данными работать. Фактически в этой области используется менее 256 байт, однако по мере развития операционной системы сюда можно будет добавлять новые поля.

Таким образом, память, которую ***«OSE!»*** может использовать для других нужд – загрузки драйверов и приложений, выделения им динамической памяти, – начинается сразу после загруженного файла ядра. Следующие два участка памяти размером по 512 байт каждый содержат код загрузочного сектора и настроенный им стек. Однако после успешной загрузки ядра их содержимое также становится ненужным, поэтому, если выполнить перенастройку стека, с этого момента можно также считать эти 1024 байта свободными.

Следующая занятая область памяти – *EBDA*. Она начинается в диапазоне физических адресов от `80000h` до `9FFFFh`. В ранних реализациях *BIOS* этой структуры данных не существовало, и, при наличии достаточного количества физической памяти, все 128 Кбайт из этого диапазона потенциально были свободной памятью. В более поздних версиях *EBDA* может присутствовать, однако её точные формат и размеры – это детали реализации. Гарантируется лишь, что *EBDA* расположена в конце этой области.

Для определения расположения *EBDA* используется сервис `int 12h`, возвращающий размер основной области памяти (так называемой *conventional memory*) за вычетом размера *EBDA*.

Сервис: `int 12h`	
Определение размера основной области памяти	
Возвращает размер основной области памяти (без *EBDA*) в килобайтах.	
Исходные данные:	нет
Результаты:	`AX` = размер основной области памяти, Кбайт
	`CF` = 0 при успешном выполнении, 1 в случае ошибки

В современных *BIOS* (в особенности в *UEFI CSM*) этот сервис возвращает значение `280h`, что соответствует 640 Кбайт памяти, однако можно встретить и версии, возвращающие другие значения: например, в *Bochs* возвращается значение на единицу меньше – `27Fh` (639 Кбайт).

Таким образом, за счёт перенастройки стека при инициализации ядра можно добиться оптимальных условий для менеджера памяти, когда вся управляемая им память будет представлять собой единственную непрерывную область. Это существенно упрощает инициализацию менеджера памяти: какая бы структура данных ни использовалась для управления блоками памяти, количе-

ство элементов, которые должны быть созданы в ней при инициализации, будет минимальным.

Теперь следует определиться с тем, как именно будет вестись учёт занятых и свободных блоков памяти. Перечислим несколько популярных подходов к разработке менеджера памяти, отличающихся используемыми структурами данных.

***Watermark*-аллокатор.** Один из самых простых способов управления памятью. Его идея заключается в том, чтобы выделять блоки памяти последовательно. В этом случае достаточно хранить только информацию о том, где заканчивается последний выделенный блок памяти. Основным недостатком является невозможность реализовать полноценное освобождение ставших ненужными блоков.

Менеджер памяти с фиксированным размером блока. В случае если выделяемые блоки памяти всегда имеют одинаковый размер (или по крайней мере существует некоторый максимальный размер блока), можно в начале каждого свободного блока записывать указатель на следующий свободный блок. В этом случае свободные блоки образуют связный список и достаточно хранить лишь информацию о расположении первого из них. Поскольку размер блоков фиксированный, выделение памяти сводится к удалению первого элемента из списка, а освобождение – к добавлению освобождаемого элемента в начало списка.

Менеджер памяти со списком блоков произвольного размера. В отличие от предыдущего подхода здесь помимо объединения блоков в список необходимо для каждого блока хранить его размер. Чаще всего для этих целей перед каждым блоком памяти размещается небольшая запись, содержащая всю необходимую служебную информацию. Выделение памяти в этом случае заключается в поиске в списке свободного блока подходящего размера, а освобождение – в том, чтобы пометить ранее выделенный блок как свободный. Кроме того, необходимо предусмотреть возможность объединения смежных свободных блоков: если этого не сделать, со временем может сложиться ситуация, когда в памяти будут свободные участки достаточного размера, но они будут состоять из нескольких блоков, каждый из которых будет меньше, чем запрашивается.

Менеджер памяти на основе древовидных структур данных. Для решения проблемы объединения свободных блоков, а также повышения скорости поиска блоков подходящего размера менеджеры памяти часто реализуются на основе древовидных структур данных. Например, если имеющиеся блоки образуют двоичное дерево поиска, найти блок подходящего размера можно за логарифмическое время против линейной сложности предыдущего подхода. В то же время, если размещать блоки в дереве по принципу их смежности, можно существенно упростить задачу их объединения.

В *MS-DOS* используется подход, основанный на использовании односвязного списка. Перед каждым блоком памяти, как занятым, так и свободным, размещается 16-байтовая запись, которая называется *MCB* (*Memory Control Block*). Её формат приведён в табл. 3.2.

Таблица 3.2

Формат *MCB* (*Memory Control Block*) в *MS-DOS*

Имя поля	Смещение, байт	Размер, байт	Описание
`bSignature`	0	1	`'Z'` – последний блок в списке, `'M'` – *Middle*-блок (все, кроме последнего)
`wOwnerID`	1	2	Идентификатор программы, которой принадлежит блок
`wSizeParas`	3	2	Размер блока в 16-байтных параграфах
`reserved`	5	3	Зарезервировано
`szOwnerName`	8	8	*C*-строка, содержащая часть имени файла программы-владельца

Блоки при такой реализации образуют односвязный список: зная расположение текущего *MCB* и размер описываемого им блока, можно сложить их, учесть размер самого *MCB* – и таким образом вычислить расположение следующего *MCB*. Свободными считаются блоки, у которых `wOwnerID` равен 0.

В первоначальной версии ***«OSE!»*** используется аналогичный подход: перед каждым блоком памяти размещается 16-байтовая запись, которая называется *MBP* (*Memory Block Prefix*). Её формат приведён в табл. 3.3.

Таблица 3.3

Формат *MBP* (*Memory Block Prefix*) в *«OSE!»*

Имя поля	Смещение, байт	Размер, байт	Описание
`cpSize`	0	2	Размер блока в 16-байтных параграфах
`wFlags`	2	2	Флаги, описывающие блок
`segPrevMBP`	4	2	Номер сегмента, содержащего предыдущий *MBP*
`segOwner`	6	2	Идентификатор программы, которой принадлежит блок
`reserved`	8	8	Зарезервировано

Следует заметить, что сегментная адресация, которая используется в реальном режиме, оказывается достаточно неудобной для практического применения, когда обращения к памяти выходят за рамки какого-либо одного сегмента: во-первых, из-за того, что одни и те же места в памяти могут иметь несколько различных адресов – комбинаций номера сегмента и смещения, а во-вторых,

из-за необходимости при обращении по такому адресу загружать его по частям, в сегментный регистр и регистр общего назначения.

Для решения этой проблемы как в *MS-DOS*, так и в ***«OSE!»*** при управлении памятью минимальной единицей считается параграф – 16-байтная область памяти, имеющая физический адрес, кратный 16. При выделении блока памяти возвращается не полный адрес («сегмент-смещение») начала блока, а только номер сегмента, в котором выделенная память начинается по смещению 0. Из тех же соображений удобно и запись, предшествующую блоку памяти (*MCB* или *MBP* соответственно), сделать размером в 16 байт. При следовании этим правилам для управления памятью достаточно работать только с номерами сегментов: каждый следующий сегмент начинается на 16 байт (один параграф) позже предыдущего (с меньшим номером). По той же причине удобно измерять размеры блоков в параграфах, а для идентификации программы, которой выделен блок, использовать номер какого-либо характерного для неё сегмента.

В реализации ***«OSE!»*** имеется два основных отличия от реализации *MS-DOS*. Во-первых, вместо однобайтового поля-сигнатуры для описания типа блока используется двухбайтовое поле флагов. В первоначальной реализации поддерживается только флаг `MBP_FLAG_LASTITEM`, соответствующий младшему биту в поле `wFlags` и устанавливаемый в 1, если блок является последним в списке. При необходимости набор флагов может быть расширен в последующих версиях. Во-вторых, добавлено поле, позволяющее быстро находить не только следующий, но и предыдущий блок в списке, что особенно полезно при объединении свободных блоков.

В обеих операционных системах список блоков реализован таким образом, чтобы порядок следования блоков в списке совпадал с порядком их следования в памяти. Отчасти это обусловлено тем, что при этом нет необходимости хранить и размер блока, и указатель на следующий блок. Вместе с тем такой порядок включения блоков в список также упрощает объединение свободных блоков: при их включении в список в произвольном порядке было бы проблематично (и вычислительно более затратно) находить те из них, которые являются смежными.

Один из вопросов, на которые нужно ответить при реализации менеджера памяти, заключается в том, должна ли память, занятая ядром операционной системы, рассматриваться как особая или её следует также отдать под управление менеджеру памяти. У каждого из этих вариантов есть свои преимущества и недостатки: особый подход к памяти позволяет упростить инициализацию менеджера памяти, в то же время при управлении этим участком памяти наравне со всеми прочими появляется возможность реализовать выгрузку ненужных частей ядра с освобождением используемой ими памяти. Поскольку ***«OSE!»*** должна поддерживать даже компьютеры с ограниченным объёмом ОЗУ (от 32 Кбайт), в этой операционной системе используется второй подход.

Приступим к реализации аналогичного используемому в ***«OSE!»*** менеджера памяти. Будем исходить из предположения, что ядро загружается по адресу `0070:0000` и управление ему передаётся дальним прыжком именно по

этому адресу. Процедуры, непосредственно реализующие логику работы менеджера памяти, вынесем в отдельный файл `Memory.inc`, а основной код ядра (которое в данном примере будет просто выполнять несколько вызовов процедур менеджера памяти) оставим в главном файле `BootMe.asm`.

```
        include 'proc16.inc'

Options.Kernel.SDASegment       equ     $0060

EntryPoint:
        xor     ax, ax
        mov     ss, ax
        mov     sp, (Options.Kernel.SDASegment) shl 4
```

Подключаемый файл `proc16.inc` является адаптацией для 16-битного кода, поставляемого вместе с *FASM* файла `proc32.inc`. Номер сегмента для размещения глобальных данных операционной системы (*SDA*) объявляется как символическая константа, что позволит при необходимости изменять эту величину без внесения правок в код менеджера памяти.

Код ядра в данном примере начинается с перенастройки стека на область памяти, непосредственно предшествующую *SDA*. Для этого номер сегмента *SDA* преобразуется в смещение относительно нулевого сегмента. При этом предполагается, что результат преобразования представи́м в виде 16-битного числа. Если это не так, данный участок кода следует скорректировать.

Выбор именно этой области для размещения стека обусловлен тем, что в данном примере его активное использование не планируется: вложенность вызовов процедур не превышает 2, параметры передаются через регистры, внутри самих процедур стековая память также используется ограниченно. Для полноценного ядра операционной системы следует подойти к размещению стека более серьёзно.

Далее в коде ядра будут содержаться вызовы процедур менеджера памяти, поэтому перейдём к рассмотрению возможной их реализации, содержащейся в файле `Memory.inc`.

```
label SDA.segFirstMBP   word at $0000

label MBP.cpSize        word at $0000
label MBP.wFlags        word at $0002
label MBP.segPrevMBP    word at $0004
label MBP.segOwner      word at $0006

MBP_FLAG_LASTITEM       = $0001
```

Для повышения читаемости кода выполняется ряд объявлений, соответствующих используемому в ***«OSE!»*** формату *MBP*. При этом в *SDA* будут использоваться только первые 2 байта – для хранения номера сегмента, содержащего первый *MBP*.

```
proc Memory.GetTotalSize
     clc
     int        12h
     jnc        .EndProc
.Default:
     mov        ax, 32
.EndProc:
     ret
endp
```

Вспомогательная процедура `Memory.GetTotalSize` используется для получения размера основной области памяти (до начала *EBDA*). Параметры у процедуры отсутствуют, результат возвращается в регистре `AX`. В случае неудачного вызова возвращается значение по умолчанию – 32 Кбайт, что соответствует минимальному поддерживаемому ***«OSE!»*** объёму ОЗУ.

```
proc Memory.Initialize uses ds bx
     call       Memory.GetTotalSize
     mov        cl, 6
     shl        ax, cl
     sub        ax, bx
     sub        ax, (Kernel.cpSize) + 1
     mov        cx, (Options.Kernel.SDASegment)
```

Процедура `Memory.Initialize` отвечает за инициализацию менеджера памяти. Ожидается, что при её вызове в регистр `BX` будет помещён номер сегмента, начиная с которого расположено ядро ОС (далее – базовый сегмент ядра). Результатом работы процедуры будет проинициализированный список из двух блоков: блока памяти, занятого ядром ОС, и свободного блока, охватывающего всю остальную память.

Инициализация начинается с получения размера основной области памяти и его преобразования из килобайтов в 16-байтные параграфы. Из полученного числа вычитается номер базового сегмента (или количество параграфов до начала ядра), а также количество параграфов, занимаемых ядром (`Kernel.cpSize`). Ещё один параграф, исключаемый из общего объёма памяти – 16 байт, занятые *MBP* для блока с ядром.

```
     dec        bx
     mov        ds, cx
     mov        [SDA.segFirstMBP], bx
     mov        ds, bx
     inc        bx
     mov        [MBP.cpSize], (Kernel.cpSize)
     mov        [MBP.wFlags], 0
     mov        [MBP.segPrevMBP], $0000
     mov        [MBP.segOwner], bx
```

MBP, описывающий область памяти, занятую ядром, располагается непосредственно перед этой областью, одним параграфом ранее. Номер соответствующего сегмента (на 1 меньше, чем номер сегмента, в который загружается

ядро) записывается в *SDA*. Поля самого *MBP* заполняются в соответствии с описываемым им блоком: размер блока определяется размером ядра, блок не является последним в списке, не имеет предыдущего (значение `$0000` в поле `MBP.segPrevMBP`), а его владельцем является само ядро ОС.

```
      add         bx, (Kernel.cpSize)
      mov         cx, ds
      mov         ds, bx
      mov         [MBP.cpSize], ax
      mov         [MBP.wFlags], MBP_FLAG_LASTITEM
      mov         [MBP.segPrevMBP], cx
      mov         [MBP.segOwner], 0
      ret
endp
```

MBP второго блока должен описывать всю оставшуюся свободную память. Размер блока вычисляется в начале процедуры, блок помечается как последний в списке, предыдущим для него будет блок, содержащий ядро ОС. Так как владелец отсутствует – блок свободен.

Приступая к разработке алгоритма выделения памяти, необходимо определиться с реализуемой им стратегией управления памятью. В общем случае память, учёт которой ведёт менеджер памяти, состоит из занятых и свободных блоков различных размеров, расположенных в произвольном порядке. При этом для удовлетворения запроса на выделение памяти потенциально можно использовать любой свободный блок, размер которого больше или равен запрашиваемому: в случае, если свободный блок больше требуемого, его всегда можно разделить на две части.

Рассмотрим несколько наиболее популярных и простых стратегий выделения памяти.

«Первый подходящий» (*first-fit*). При просмотре свободных блоков памяти выбирается самый первый из них, имеющий достаточный размер. Основным преимуществом этой стратегии является простота реализации и сравнительно высокая скорость работы. Недостаток заключается в том, что память при использовании этой стратегии используется неравномерно: в основном занятые блоки будут располагаться ближе к её началу. При некоторых условиях это может снижать производительность со временем.

«Наилучший» (*best-fit*). При просмотре свободных блоков памяти выбирается тот, размер которого наиболее близок к запрашиваемому. Преимущество данной стратегии заключается в том, что удаётся избежать дробления крупных свободных блоков за счёт переиспользования ранее освободившихся участков памяти. Ценой этого является более низкая производительность: при реализации менеджера памяти на основе связных списков приходится каждый раз просматривать все элементы. Частично эту проблему можно решить, если хранить дополнительные данные, позволяющие исключать из рассмотрения заведомо неподходящие блоки, однако это делает реализацию сложнее. Ещё один недостаток этой стратегии заключается в том, что со временем образуется много

свободных блоков, размеры которых меньше, чем запрашиваемые объёмы памяти: в результате часть памяти, являясь свободной, фактически становится недоступной для использования.

«Наихудший» (*worst-fit*). При просмотре свободных блоков памяти выбирается блок наибольшего размера. В результате при выделении памяти свободные блоки всегда остаются достаточно крупными, чтобы быть полезными при последующих запросах. Как правило, это способствует более равномерному использованию доступной памяти, однако потенциальной проблемой является ситуация, когда нужно выделить блок, размер которого больше размера любого из свободных блоков: суммарно свободной памяти при этом может быть достаточно, но фрагментация делает выполнение запроса невозможным.

«Следующий подходящий» (*next-fit*). Эта стратегия похожа на стратегию «первый подходящий» (*first-fit*), отличаясь лишь тем, что при каждом запросе на выделение памяти поиск подходящего блока начинается не с начала управляемой менеджером памяти области, а с того места, где был остановлен поиск при предыдущем выделении памяти. Это требует хранения несколько большего набора метаданных – не только адреса первого блока, но и адреса блока, с которого следует начинать поиск в следующий раз, – а также делает алгоритм поиска чуть сложнее (т. к. необходимо, дойдя до конца списка, вернуться в его начало и остановиться, если в ходе поиска достигнута отправная точка), однако позволяет использовать память более равномерно.

Перечисленные стратегии применимы преимущественно к менеджерам памяти, которые организованы на основе связных списков или для которых порядок просмотра блоков-кандидатов может быть установлен иным способом. На практике можно встретить и более сложные реализации, основанные на группировке блоков одинакового размера и/или накоплении освобождаемых блоков определённого размера для последующего быстрого переиспользования.

В качестве примера рассмотрим реализацию стратегии *first-fit*. Будем исходить из того, что при вызове процедуры в регистре `AX` содержится требуемый размер блока в байтах, а в `BX` – идентификатор программы, запрашивающей выделение памяти.

```
proc Memory.AllocFirst uses ds bx
     mov        cl, 4
     add        ax, 15
     shr        ax, cl
```

Поскольку для идентификации блоков используются номера сегментов, минимальной единицей выделения памяти является параграф. Запрашиваемый объём памяти в байтах конвертируется в количество параграфов (полных или неполных), требуемых для размещения блока.

```
        mov         dx, (Options.Kernel.SDASegment)
        mov         ds, dx
        mov         dx, [SDA.segFirstMBP]
        mov         ds, dx
```

Для минимизации объёма машинного кода при поиске подходящего блока для указания на текущий элемент в списке будет использоваться регистр `DS`. До начала цикла в него помещается номер сегмента первого *MBP*.

```
.SearchLoop:
        mov         cx, [MBP.cpSize]
        cmp         [MBP.segOwner], 0
        jne         .Next
        cmp         cx, ax
        jae         .Found
        test        [MBP.wFlags], MBP_FLAG_LASTITEM
        jnz         .NotFound
.Next:
        add         dx, cx
        inc         dx
        mov         ds, dx
        jmp         .SearchLoop
```

На каждой итерации цикла каждый из рассматриваемых блоков проходит ряд проверок. В первую очередь отсеиваются занятые блоки – у них поле `MBP.segOwner` содержит ненулевое значение. Затем проверяется размер текущего блока. Если он слишком мал, необходимо перейти к следующему блоку. Исключение – если рассмотренный блок был последним: в этом случае делается вывод о невозможности выделения запрошенного количества памяти.

```
.Found:
        mov         [MBP.segOwner], bx
        sub         cx, ax
        jz          .Done
```

В случае если подходящий блок был найден, идеальной ситуацией является точное совпадение его размера с запрашиваемым: в этом случае номер его сегмента уже находится в `DS`, поэтому можно сразу увеличивать его на 1 и возвращать из процедуры. В противном случае выбранный свободный блок необходимо разделить на две части: занятый блок в начале и свободный в неиспользованной части.

```
        mov         [MBP.cpSize], ax
        mov         bx, [MBP.wFlags]
        and         [MBP.wFlags], not(MBP_FLAG_LASTITEM)
        add         dx, ax
        inc         dx
        dec         cx
```

Для ставшей занятой части блока переиспользуется уже имеющийся *MBP*, в него записывается размер выделяемого блока. Кроме того, поскольку за выделен-

ной областью будет создаваться ещё один блок, данный *MBP* перестаёт быть последним в списке, даже если был таковым до запроса на выделение памяти.

```
        push    ds
        mov     ax, ds
        mov     ds, dx
        mov     [MBP.cpSize], cx
        mov     [MBP.wFlags], bx
        mov     [MBP.segPrevMBP], ax
        mov     [MBP.segOwner], 0
        pop     ds
```

Оставшаяся часть разделённого блока превращается в свободный блок меньшего размера. Для этого в её начале создаётся новый *MBP*, при заполнении которого используются побочные результаты предыдущих действий: в регистр `BX` ранее были скопированы флаги первоначального блока, а при проверке на точное совпадение размеров использована инструкция `sub`, вследствие чего в `CX` остаётся размер в параграфах свободной области (вместе с размером *MBP*).

Следует отметить, что здесь возможна ситуация, когда размер неиспользованной части составляет ровно один параграф и создаваемый *MBP* фактически начинает описывать свободную область нулевого размера. Это неизбежные издержки управления памятью: подобные блоки нулевого размера иногда будут возникать, поскольку далее следует уже занятый блок (иначе выбранный при поиске свободный блок изначально просто был бы большего размера), а для управления памятью *MBP* необходим.

```
.Done:
        mov     ax, ds
        inc     ax
        clc
        jmp     .EndProc
```

Независимо от того, потребовалось разбиение найденного блока на две части или нет, в регистре `DS` остаётся номер сегмента с *MBP*, который ровно на единицу отличается от номера сегмента, с которого начинается сам выделенный блок памяти. Признак успешного завершения работы процедуры – обнуление флага `CF`.

```
.NotFound:
        xor     ax, ax
        stc
.EndProc:
        ret
endp
```

При невозможности выделить запрошенный объём памяти возвращается нулевой номер сегмента, а флаг `CF` устанавливается в единицу, сигнализируя об ошибке.

Теперь рассмотрим пример реализации процедуры освобождения памяти. Освобождаемый блок будет задаваться значением в регистре `AX` – номером сегмента, получаемым при выделении памяти.

```
proc Memory.Free uses ds bx si
     dec        ax
     mov        ds, ax
```

Соответствующий блоку *MBP* располагается в параграфе, предшествующем началу этого блока. Для того, чтобы отслеживать ошибки в работе с динамической памятью, можно договориться записывать в *MBP* также специальную сигнатуру, наличие которой позволяло бы с высокой долей вероятности говорить о том, что процедуре освобождения передан корректный адрес (номер сегмента) ранее выделенного блока. Тем не менее полностью исключить ошибку этот приём не поможет, поэтому в приводимой реализации он не используется.

По той же причине не предпринимается попыток отследить повторное освобождение блока: блок может оказаться выделенным заново до попытки повторного освобождения, и выявить проблему в этом случае станет невозможно.

При необходимости дополнительные проверки могут быть добавлены читателем самостоятельно.

```
     mov        cx, [MBP.cpSize]
     mov        bx, ax
     mov        si, [MBP.wFlags]
     and        si, MBP_FLAG_LASTITEM
```

Главная сложность процедуры освобождения памяти заключается в необходимости объединять смежные свободные блоки, чтобы избежать излишней фрагментации. Для любого освобождаемого блока имеется только два кандидата на объединение: непосредственно предшествующий ему и непосредственно следующий за ним, причём возможна и ситуация, когда свободны одновременно оба эти блока. Тем не менее при освобождении будет образовываться не более одного свободного блока, поэтому для объединения достаточно отслеживать значения трёх величин: размер блока (в приведённом фрагменте кода начальное значение скопировано в регистр `CX`), начало блока (регистр `BX`) и флаги (регистр `SI`).

```
        jnz         .NextMergeDone
        mov         dx, ax
        add         dx, cx
        inc         dx
        mov         ds, dx
        cmp         [MBP.segOwner], 0
        jne         .NextMergeDone
        mov         dx, [MBP.wFlags]
        and         dx, MBP_FLAG_LASTITEM
        or          si, dx
        add         cx, [MBP.cpSize]
        inc         cx
.NextMergeDone:
```

Слияние блоков удобнее проводить начиная с блока, следующего за освобождаемым: во-первых, в этом случае проще сразу определить, будет ли результирующий блок последним в списке, а во-вторых, в регистр `BX`, используемый для хранения информации о начале объединённого блока, должен попасть наименьший из номеров сегментов блоков, участвующих в слиянии, поэтому такой порядок их просмотра оптимален.

```
        mov         ds, ax
        mov         dx, [MBP.segPrevMBP]
        test        dx, dx
        jz          .PrevMergeDone
        mov         ds, dx
        cmp         [MBP.segOwner], 0
        jne         .PrevMergeDone
        add         cx, [MBP.cpSize]
        inc         cx
        mov         bx, dx
.PrevMergeDone:
```

Слияние с предшествующим блоком осуществляется аналогичным образом, за исключением того, что он не может быть последним в списке. После того, как оба смежных блока учтены, можно производить слияние:

```
        mov         ds, bx
        mov         [MBP.cpSize], cx
        mov         [MBP.wFlags], si
        mov         [MBP.segOwner], 0
.EndProc:
        ret
endp
```

Приведённый пример реализации менеджера памяти ограничивается только операциями выделения и освобождения блоков памяти. На практике, как правило, добавляют поддержку операции изменения размера ранее выделенного блока, а также возможность определения размера выделенного блока. Тем не менее для простейшей операционной системы приведённую реализацию менеджера памяти можно считать необходимым минимумом.

Как придумать формат исполняемого файла?

После того как в ядре операционной системы реализован менеджер памяти, можно приступать к решению других задач, в том числе – организации запуска программ. Без поддержки запуска сторонних программ ценность операционной системы оказывается под большим вопросом, т. к. её возможности ограничиваются той функциональностью, которая заложена разработчиком.

Задача запуска программ может быть разделена на несколько подзадач:

– выбор или проектирование формата исполняемого файла;

– загрузка исполняемого файла в память и подготовка его копии в памяти к выполнению;

– предоставление запущенной программе доступа к функциям операционной системы.

Формат исполняемых файлов, который будет использоваться операционной системой, как правило, в значительной степени взаимосвязан с её архитектурой, решаемыми ею задачами и т. д., поэтому каждая из перечисленных задач в конечном счёте решается не особобленно, а в комплексе и зачастую в несколько итераций, по мере развития соответствующих модулей системы.

В целом решение вопроса о формате исполняемых файлов – задача творческая, однако несколько базовых подходов всё же можно выделить. Выбор подхода во многом определяется теми задачами, которые предстоит решать с использованием ОС, а также ресурсами, доступными разработчику операционной системы.

Использование «дампов памяти» в качестве исполняемых файлов. При этом подходе исполняемый файл представляет собой копию содержимого, которое должно оказаться в памяти сразу после запуска программы. В каком-то смысле этот подход аналогичен тому, как производится запуск ядра операционной системы простейшим загрузчиком, с тем отличием, что, как правило, загружается не последовательность секторов, а содержимое некоторого файла. Загрузка выполняется по определённым (выбранным разработчиком операционной системы) правилам, после чего управление сразу передаётся на начало загруженного в память файла. Основное преимущество этого подхода – простота реализации. Главный недостаток – ограниченные возможности управления процессом загрузки для разработчика программы. Вместе с тем данный подход успешно использовался в *CP/M* и, позднее, в *MS-DOS* в виде т. н. *COM*-файлов.

Использование формата с заголовком и фиксированным набором областей. В рамках данного подхода исполняемый файл начинается с небольшого блока полей, описывающих, как содержимое исполняемого файла должно загружаться для выполнения. При этом, как правило, всё остальное содержимое файла рассматривается как один непрерывный блок, загружаемый в память без разделения на части. Данный подход применяется, например, в файлах формата *MZ EXE* в *MS-DOS*: заголовок файла этого формата содержит желаемые начальные значения некоторых регистров, несколько полей, описывающих размеры и положение в файле самой программы, а также набор указаний о том, как необходимо изменить копию программы в памяти перед передачей ей

управления, чтобы она могла корректно работать, будучи загруженной в выбранную для неё область памяти.

Использование сложного формата. Примером этого подхода является формат *COFF* и аналогичный ему *PE EXE*. В этих форматах исполняемый файл состоит из множества составных частей, называемых секциями, каждая из которых имеет своё специфическое внутреннее устройство. При этом некоторые секции являются обязательными, другие – опциональными, а при загрузке их взаимное размещение может отличаться от использованного в исполняемом файле. Главное преимущество этого подхода заключается в том, что при грамотном проектировании возможно практически неограниченное расширение формата с сохранением обратной совместимости. Очевидный недостаток заключается в сложности таких форматов, а также некоторой избыточности возможностей для большинства программ.

Необходимость обеспечивать совместимость с уже существующими системами также может оказывать влияние на форматы исполняемых файлов, поддерживаемые операционной системой. Особенно это актуально для любительских операционных систем с небольшой командой разработчиков: разработка всего программного обеспечения, которое сделало бы такую операционную систему полезной или хотя бы просто интересной для обычного пользователя, может потребовать многих тысяч человеко-часов, а хотя бы частичная поддержка запуска программ, разработанных для уже существующей аналогичной операционной системы, позволяет довольно быстро предоставить пользователю минимальный набор инструментов для работы.

Следует также учитывать, что операционной системе, как правило, нужно поддерживать исполняемые файлы нескольких видов. Как минимум, это обычные прикладные программы и драйверы. Дополнительно в качестве отдельных видов можно рассматривать загружаемые модули для прикладных программ (аналог *DLL* в *Windows*) и собственно файл с ядром операционной системы. Из соображений единообразия и упрощения самой операционной системы обычно отдают предпочтение одному общему формату, однако возможен и вариант реализации с несколькими различными.

Пример 3.2. Проектирование формата исполняемого файла

В качестве примера рассмотрим используемый в ***«OSE!»*** формат исполняемый файлов. Для этого проанализируем изначальные требования к нему и проследим за ходом рассуждений, приводящих к той его реализации, которая используется в первоначальной версии этой учебной операционной системы.

Одной из задач при разработке ***«OSE!»*** была частичная поддержка приложений, написанных для *MS-DOS*. Это автоматически означает обязательную поддержку двух форматов исполняемых файлов: *COM* и *MZ EXE*. Вместе с тем ***«OSE!»*** разрабатывалась не только как аналог *MS-DOS*, но и как «песочница» для экспериментов с различными техническими решениями, поэтому системе было крайне желательно иметь ещё и свой собственный формат исполняемых файлов.

Естественным образом возникает необходимость различать эти три вида исполняемых файлов. Сравнительная характеристика форматов, поддерживаемых *MS-DOS*, приведена в табл. 3.4.

Таблица 3.4

Сравнительная характеристика форматов исполняемых файлов *MS-DOS*

Свойство	**Формат *COM***	**Формат *MZ EXE***
Расширение имени файла	Обычно `.com`	Обычно `.exe`
Сигнатура	Отсутствует	Первые два байта содержат символы `'MZ'`
Размер файла	До 64 Кбайт	Любой

Основным критерием для определения того, к какому из форматов относится исполняемый файл, является наличие или отсутствие в нём сигнатуры. В частности, в случае, если файл с расширением `.com` на самом деле представляет собой *EXE*-файл, его запуск пройдёт корректно. Тем не менее, если такой файл попытаться запустить на системе, которая поддерживает только *COM*-программы, произойдёт выполнение бессмысленного кода:

```
0100    4D              dec bp
0101    5A              pop dx
```

Дальнейшие инструкции, выполнение которых последует в этом случае, зависят от содержимого заголовка *EXE*-файла. Вероятность такого события довольно невысока, однако всё же желательно учитывать возможность его возникновения.

При проектировании формата исполняемых файлов для ***«OSE!»*** предполагалось, что один и тот же исполняемый файл должен поддерживать несколько различных сценариев использования.

Запуск в качестве прикладной программы. Этот случай является основным назначением исполняемых файлов, однако для ***«OSE!»*** как учебной операционной системы предусматривается возможность расширения формата в дальнейшем с целью поддержки новых добавляемых в систему возможностей. Причём, поскольку подобная доработка может производиться в рамках учебного эксперимента, расширение возможностей формата должно быть осуществимым максимально простым способом, с минимальным влиянием на уже имеющуюся реализацию.

Загрузка в качестве вспомогательного модуля или драйвера. Частным случаем расширения возможностей формата является поддержка им указания нескольких точек входа, каждая из которых представляет собой одну из функций, доступных для использования другими программами. В этом отношении как вспомогательный модуль, так и драйвер представляют один и тот же сценарий использования. Поддержка же такой возможности не только со стороны

формата, но и со стороны операционной системы, должна упростить для разработчиков прикладных программ поддержку так называемых плагинов. Этот механизм по своему назначению аналогичен работе с *DLL*-файлами в *Windows*.

Запуск в качестве ядра или вторичного загрузчика. Естественным применением формата стало также его использование для ядра системы или, в случае поддержки более сложных файловых систем, чем *FAT*, – вторичного загрузчика. Это означает, в частности, что первичный загрузчик (т. е. код загрузочного сектора) должен быть в состоянии загрузить такой файл и передать ему управление. Разумеется, в этом случае состояния регистров и памяти в момент запуска будут отличаться от условий при запуске в качестве обычной прикладной программы, а значит, речь идёт об отдельной точке входа для этого сценария использования. С точки зрения пользователя идея заключается ещё и в том, что при попытке запуска файла с ядром системы происходит не повторный запуск ядра (что нежелательно), а выполнение отдельного участка кода, который может, например, выводить сообщение о назначении файла.

Запуск под управлением других операционных систем. В случае если файл будет по ошибке запущен под управлением операционной системы, которая не поддерживает такой формат (например, *MS-DOS* или *Windows*), это не должно приводить к неконтролируемому выполнению произвольных инструкций. Поскольку корректная работа программы для ***«OSE!»*** в одной из таких операционных систем не может быть гарантирована, желательным поведением является как можно более быстрое завершение её выполнения.

Распознавание файла как исполняемого сторонними утилитами. Существуют различные виды программного обеспечения, которые решают задачу определения принадлежности файлов к тому или иному формату: например, подобные технологии используются при криминалистическом анализе файловых систем, а также для восстановления удалённых файлов, для которых утеряна информация об имени и других атрибутах. Возможность простого распознавания файлов разрабатываемого формата при этом становится его желательным свойством, а используемая сигнатура по возможности должна указывать на операционную систему, к которой имеет отношение данный формат.

Простое решение заключается в том, чтобы выбрать сигнатуру, которая не встречается в корректных *COM*-программах и не совпадает с заголовком *MZ EXE*. Тем не менее в этом случае не удастся выполнить требование о корректной работе под управлением других операционных систем, т. к. в них такой файл по-прежнему рассматривался бы как обычная *COM*-программа.

Исполняемые файлы ***«OSE!»*** в своих первых байтах фактически представляют собой три параллельных формата. Если исключить из рассмотрения конкретные возможности, добавленные по мере развития операционной системы, то заголовок в этих файлах имеет формат, представленный в табл. 3.5.

Таблица 3.5

Формат заголовка исполняемых файлов в первоначальной версии ***«OSE!»***

Имя поля	Формат	Описание
`oefMagic`	`Byte[8]`	Используется в качестве сигнатуры. Как правило, эти байты должны содержать следующие значения: `4F 53 45 21 EB 5A C3 00  «OSE»ëZÃ.`
`oefPad-ding`	`Byte[88]`	Используется для расширения возможностей формата

Как видно из таблицы, первые 4 байта исполняемого файла ***«OSE!»*** представляют собой посимвольную запись названия операционной системы: `'«OSE»'`. Это в полной мере решает задачу идентификации формата файла: при совпадении первых 4 байт с этой сигнатурой можно с высокой долей вероятности говорить о его использовании. Значения следующих 4 байт могут использоваться для подтверждения этого предположения, однако, как будет показано далее, их значения могут незначительно варьироваться.

Следующая задача – обеспечение корректной работы исполняемого файла этого формата в том случае, если он будет запущен в качестве *COM*-программы. Так как ***«OSE!»*** поддерживает запуск таких программ, написанных для *MS-DOS*, существует высокая вероятность того, что по ошибке произойдёт и обратный перенос файлов. В этом случае первые 4 байта будут выполнены как машинные команды:

```
0100    4F              dec di
0101    53              push bx
0102    45              inc bp
0103    21 ??           and ??, ??
```

Нетрудно заметить, что первые три инструкции хоть в целом и довольно бессмысленны, однако не приведут к аварийному завершению программы или выполнению нежелательных действий. Тем не менее четвёртый байт является опкодом для инструкции размером не менее 2 байт: следующий за сигнатурой байт будет восприниматься процессором как *ModR/M*-байт, а в зависимости от его значения инструкция может как ограничиться этими двумя байтами, так и иметь дополнительное поле для задания смещения операнда в памяти.

Поскольку исполняемый файл этого формата не является корректной *COM*-программой, желательно как можно быстрее, т. е. минимальным количеством байт, обеспечить корректное завершение его выполнения. Необходимый минимум включает в себя два действия:

- удаление с вершины стека одного значения (`pop reg` занимает 1 байт);
- выполнение инструкции `ret` (также занимает 1 байт).

Следует однако учитывать также случай запуска исполняемого файла первичным загрузчиком. Ограниченность загрузочного сектора не позволяет организовать сложный разбор формата, поэтому оптимальным решением будет организация передачи управления по заранее известному смещению в файле, однако при этом ход выполнения должен отличаться от всех прочих случаев. Первые 4 байта уже заняты сигнатурой, поэтому естественным решением видится размещение точки входа для первичного загрузчика по смещению +4 от начала файла. При этом следует учесть, что здесь же будет продолжаться код, выполняемый при запуске в качестве *COM*-программы.

Таким образом, по смещению +4 тоже должен начинаться корректный машинный код, однако его объём должен быть минимальным, т. к. далее должен следовать остальной заголовок исполняемого файла с информацией, необходимой для поддержки возможностей операционной системы по запуску программ. Избыточный размер заголовочной части исполняемого файла нежелателен сам по себе, т. к. чем больше места займёт этот стартовый код, тем меньше места останется для полезных полей.

Самым простым решением в этой ситуации будет размещение по смещению +4 обычной инструкции безусловного перехода. Аналогичный приём используется, например, в начале загрузочного сектора в файловых системах семейства *FAT*. В этом случае единственной сложностью останется согласование этой инструкции с теми, которые будут распознаваться процессором при передаче управления на самое начало файла.

В табл. 3.6 приведена интерпретация указанных ранее значений для первых 8 байт исполняемого файла ***«OSE!»*** с точки зрения основных сценариев использования.

Таблица 3.6

Интерпретация значений первых 8 байт заголовка исполняемых файлов ***«OSE!»***

<table>
<tr><th rowspan="2"></th><th colspan="3">Сценарии использования</th></tr>
<tr><th>MS-DOS</th><th>Загрузчик «OSE!»</th><th>Исполняемый файл «OSE!»</th></tr>
<tr><td>4F</td><td>dec di</td><td rowspan="3">Пропускается</td><td rowspan="3">Сигнатура '«OSE»'</td></tr>
<tr><td>53</td><td>push bx</td></tr>
<tr><td>45</td><td>inc bp</td></tr>
<tr><td>21</td><td rowspan="2">and bp, bx</td><td rowspan="2">jmp +$5A</td><td rowspan="4">Игнорируется</td></tr>
<tr><td>EB</td></tr>
<tr><td>5A</td><td>pop dx</td></tr>
<tr><td>C3</td><td>ret</td><td rowspan="2">Игнорируется</td></tr>
<tr><td>00</td><td>Игнорируется</td></tr>
</table>

Рассмотрим более детально смысл первых 8 байт исполняемых файлов ***«OSE!»*** в каждом из сценариев использования.

При запуске под управлением операционных систем, которые не поддерживают данный формат, программа будет выполняться начиная со смещения +0. В этом случае она изменит значения трёх регистров – `DI`, `BP` и `DX` – и корректно завершит своё выполнение. Так как операционные системы в целом должны быть готовы к тому, что программа изменит значения любых регистров общего назначения, эта ситуация будет обработана корректно. Логически это почти эквивалентно завершению работы программы сразу после запуска.

Первичный загрузчик ***«OSE!»*** передаёт управление по смещению +4 от начала файла. В этом случае первая инструкция, которая встретится процессору, – безусловный переход на `5Ah` байт вперёд. Это позволяет пропустить поля в заголовке исполняемого файла, которые могут следовать почти сразу за этой инструкцией, начиная со смещения +7 в файле.

Значение `5Ah` выбрано из двух соображений: во-первых, чтобы заголовок исполняемого файла заканчивался на 16-байтной границе, что согласуется с рассмотренной ранее реализацией менеджера памяти, а во-вторых, чтобы не откладывать извлечение лишнего элемента из стека. Значения менее `50h` при использовании в качестве первого байта инструкции могут означать следующее:

– арифметические инструкции: `add`, `or`, `adc`, `sbb`, `and`, `sub`, `xor`, `cmp`, `inc`, `dec`;

– инструкции для работы с двоично-десятичными числами: `daa`, `aaa`, `das`, `aas`;

– префиксы переопределения сегментов;

– первый байт инструкций с 2-байтовыми опкодами;

– инструкции `push` и `pop`, работающие с сегментными регистрами.

Нетрудно заметить, что ничего из перечисленного не позволит приблизиться к завершению программы: для изменения значения в вершине стека потребовалась бы адресация через регистр `SP`, которая в реальном режиме не поддерживается, а извлечение произвольных значений в сегментные регистры нежелательно. Не подойдут также и значения с `50h` по `57h`: они соответствуют инструкциям вида `push reg16`, которые будут делать стартовый код длиннее, чего также хотелось бы избежать.

Хорошим выбором оказываются значения с `58h` по `5Fh`, которые соответствуют инструкциям вида `pop reg16`. Единственное нежелательное значение из этого диапазона – `5Ch`, что соответствует инструкции `pop sp`, которая произведёт в данной ситуации перенастройку вершины стека на непредсказуемое место в памяти, сделав тем самым корректное завершение программы крайне затруднительным.

С учётом желаемой кратности размера заголовка 16 байтам оптимальным оказывается значение `5Ah`. При этом заголовок уже получается довольно объёмным, поэтому рассматривать большие значения не имеет смысла. К тому же инструкции с большими значениями опкодов – это либо более сложные инструкции наподобие `iret` или `imul`, либо инструкции, отсутствовавшие в *Intel* 8086, поддержка которого заявлена в ***«OSE!»***.

Во всех прочих случаях, когда файл должен рассматриваться как обычная ***«OSE!»***-программа, ориентиром является сигнатура в первых 4 байтах, после чего можно переходить к анализу полей заголовка и остальному содержимому файла.

Как загрузить с диска драйвер для работы с диском?

Независимо от того, спроектирован для операционной системы собственный формат исполняемых файлов или использован один из уже имеющихся, следующий шаг на пути к запуску программы – чтение соответствующего файла с диска. Сама по себе эта задача достаточно проста и сводится к анализу структур данных, используемых в той или иной файловой системе. Особенно легко она решается в том случае, когда операционная система поддерживает только одну файловую систему или семейство файловых систем, как это было при разработке *MS-DOS*.

На практике же, особенно если планируется развитие операционной системы, может возникнуть необходимость поддерживать как различные файловые системы, так и различные виды накопителей, в том числе те, работа с которыми выполняется несколько иначе, чем с обычными дисками, доступными из *BIOS*. Во всех перечисленных случаях решением, позволяющим обеспечить гибкость и расширяемость операционной системы, является разработка драйверов, т. е. вынесение части логики, которая может меняться в зависимости от используемых программных или аппаратных решений, в отдельные исполняемые файлы и их последующая загрузка по мере необходимости.

В случае если операционная система должна будет загружаться с различных видов накопителей, использующих различные файловые системы, идеальным решением было бы сделать так, чтобы весь код, выполняющий специфическую для конкретного накопителя работу, находился в первичном загрузчике, а код ядра (или вторичного загрузчика) лишь вызывал уже реализованные в первичном загрузчике процедуры. К сожалению, на практике это редко осуществимо из-за ограниченности первичного загрузчика в объёме.

При грамотной, модульной организации операционной системы ядро должно работать с дисками не напрямую, а посредством одного или нескольких драйверов (например, один отвечает за взаимодействие с самим устройством, а второй – драйвер файловой системы – отвечает за работу с хранящимися на диске файлами). В то же время драйверы должны загружаться в память по мере необходимости, а для загрузки драйвера с диска нужны драйверы, обеспечивающие работу с этим диском. Ещё сложнее эта ситуация становится из-за того, что работа с файлами понадобится ядру уже на ранних этапах: например, при перечислении доступных аппаратных устройств или при запуске первой программы, которая будет выполнять роль т. н. оболочки, т. е. предоставлять пользовательский интерфейс для решения базовых задач с использованием операционной системы.

Рассмотрим несколько возможных решений этой проблемы.

Использование вторичного загрузчика. При использовании этого подхода загрузка ядра производится не первичным, а вторичным загрузчиком. При этом оба загрузчика содержат код, специфический для конкретного вида накопителей и/или файловой системы, и отличаются перечнем решаемых задач. Первичный загрузчик вместо ядра считывает в память вторичный загрузчик, при необходимости выбирая из нескольких доступных подходящий. Вторичный же загрузчик в этом случае должен обеспечить загрузку не только собственно ядра, но и минимального набора драйверов, который позволит ядру продолжить работу уже с их помощью.

Использование различных ядер для разных видов накопителей. Это решение заключается в том, чтобы для каждого вида загрузочного накопителя предоставлять отдельную версию файла ядра. В этом случае на ранних этапах ядро может использовать загруженные вместе с ним процедуры, ориентированные на работу с конкретным накопителем, а затем, когда будут загружены соответствующие драйверы, переключиться на их использование.

Использование специального загрузочного раздела. Сущность этого подхода заключается в том, что на загрузочном накопителе создаётся специальный раздел, который не виден пользователю при повседневной работе и, как правило, использует более простую файловую систему. Например, за основу можно взять *FAT*, но, т. к. такой раздел не будет использоваться для хранения пользовательских данных, можно ослабить ограничения, накладываемые этими файловыми системами, потребовать, чтобы искомые файлы располагались определённым образом и т. д. Содержимое такого раздела может в принципе не подчиняться требованиям ни одной из популярных файловых систем, что позволяет в том числе вернуться к простейшему загрузчику, отказавшись от деления данных на файлы на ранних этапах работы системы.

В первоначальной версии учебной операционной системы ***«OSE!»*** использован смешанный подход. Формат исполняемых файлов, спроектированный для этой системы, применяется как для прикладных программ, так и для драйверов, а также позволяет любому из этих исполняемых файлов быть запущенным первичным загрузчиком. Реализованная идея заключается в том, что файл ядра является исполняемым файлом двойного назначения, в нём фактически собраны вместе три компонента:

– непосредственно код ядра;

– код примитивного драйвера для работы с дисками средствами *BIOS*;

– код драйвера для той файловой системы, которая будет использована на конкретном накопителе.

Формат исполняемых файлов спроектирован таким образом, чтобы позволить выгружать их из памяти по частям. На ранних этапах ядро использует встроенные в его исполняемый файл драйверы для инициализации системы и загрузки других драйверов. Если обнаруживаются драйверы, способные заме-

нить встроенные, производится подмена, а части файла ядра, которые содержали встроенные драйверы, выгружаются из памяти.

Пример 3.3. Загрузка и запуск программы

Независимо от того, какой формат исполняемых файлов и какой подход для работы с дисками решено использовать, рано или поздно эта логика оказывается реализованной в коде и доступной для использования при запуске программ.

В общем случае запуск программы заключается в решении нескольких подзадач:

– выделении места в памяти для загрузки исполняемого файла;

– создании в памяти структур данных, необходимых для работы программы;

– предоставлении программе доступа к функциям операционной системы;

– передаче управления программе.

Нередко ещё одной подзадачей оказывается внесение изменений в загруженный в память образ программы: чаще всего это корректировка некоторых адресов в коде с учётом того, куда программа была загружена фактически, – *relocation*, перемещение.

Неразрывно связано с запуском программы и её завершение, поэтому при подготовке к выполнению наряду с созданием в памяти структур данных, необходимых для работы программы, выполняется также запись информации, требуемой для последующего корректного завершения её выполнения. Как правило, операционные системы не накладывают особых ограничений относительно того, какими должны быть значения большинства регистров, содержимое большей части выделенной для программы памяти и т. п. к моменту завершения работы программы, а это означает, что при получении управления операционной системой может потребоваться дополнительная информация для продолжения её работы.

Рассмотрим пример кода для запуска и завершения простейших программ, аналогичных *COM*-программам в *MS-DOS*. Будем считать, что эти программы представляют собой готовый к выполнению образ с точкой входа в самом начале файла. По аналогии с *COM*-программами в *MS-DOS* выход можно будет производить вызовом прерывания `int 20h` или выполнением инструкции `ret`. Построение полноценной *PSP*, как в *MS-DOS*, выполнять не будем, однако реализуем её упрощённый аналог. Для предоставления функций операционной системы прикладным программам будем (по аналогии с *MS-DOS*) использовать механизм обработки прерываний.

Каждую из функций операционной системы, которые будут доступны прикладным программам, удобно реализовать в форме обычной процедуры: в этом случае её можно будет вызывать из ядра напрямую, в том числе при реализации обработчика прерывания.

```
proc System.Exec uses ds es bx si di,\
     segFileName, ofsFileName

     locals
        fiFile  TFileInfo
     endl

     push       cs
     pop        ds
     lea        ax, [fiFile]
     stdcall    FS.FindFile, [segFileName], [ofsFileName], ss, ax
     jc         .NotFound
```

Код процедуры начинается с копирования значения из регистра `CS` в регистр `DS`. Это распространённая практика в том числе при написании обработчиков прерываний: если в момент возникновения прерывания выполнялась не та программа, которая установила обработчик, значения регистров, в том числе `DS`, могут отличаться от ожидаемых. В результате, если не перенастроить `DS` на правильный сегмент, первое же обращение к памяти, задействующее этот регистр, произойдёт по неправильному адресу.

Логика поиска файла с заданным именем на диске вынесена в процедуру `FS.FindFile`. В зависимости от реализации это может быть как прямая работа с диском и файловой системой, так и вызов функций, предоставляемых соответствующим драйвером. Предполагается, что признаком успеха или ошибки выступает флаг `CF`, а информация о файле записывается в структуру `TFileInfo`, полный адрес которой передаётся третьим и четвёртым параметрами. Внутреннее устройство этой структуры может быть выбрано разработчиком операционной системы произвольно, в данном примере лишь предполагается, что содержащейся там информации достаточно, чтобы впоследствии выполнить чтение из найденного файла.

```
     mov        bx, cs
     mov        ax, [fiFile.cbFileSize]
     stdcall    Kernel.GetRequiredSize, ax
     call       Memory.Alloc
```

В случае успешного обнаружения файла на диске у менеджера памяти запрашивается блок, размер которого позволяет считать этот файл в память. Логика вычисления размера блока вынесена в процедуру `Kernel.GetRequiredSize` и зависит от того, какие правила работы с памятью для программ определены в конкретной операционной системе.

```
        mov         bx, ax
        add         ax, $10
        lea         dx, [fiFile]
        stdcall     FS.ReadFile, ax, ss, dx
        jc          .ReadError
```

Первые 256 байт отведённой программе памяти будет занимать структура данных, аналогичная *PSP* в *MS-DOS*. По этой причине считывание содержимого файла осуществляется на `10h` параграфов дальше по памяти. Логика считывания содержимого файла вынесена в процедуру `FS.ReadFile`.

```
        mov         ds, bx
        mov         es, bx
        mov         word [es:$0000], $20CD
        mov         [es:$0002], sp
        mov         [es:$0004], ss
        mov         ss, bx
        mov         sp, $0000

        push        0
        push        bx
        push        $0100
        retf
```

В случае успешной загрузки исполняемого файла основные сегментные регистры – `DS`, `ES` и `SS` – настраиваются, как того требуют правила запуска *COM*-программ, на сегмент, в начале которого располагается *PSP*. В рассматриваемом примере совпадение с этой структурой данных наблюдается только в первых двух байтах, куда записывается машинный код инструкции `int 20h`.

В последующих 4 байтах сохраняется полный адрес вершины стека, который используется ядром операционной системы. Делается это для того, чтобы предоставить запускаемой программе отдельную область для её стека. Такой подход позволяет, во-первых, минимизировать вероятность случайного повреждения данных, помещаемых в стек ядром, а во-вторых, позволяет иметь возможность настраивать размеры стека независимо, как для каждой программы, так и для самого ядра что, в свою очередь, способствует более эффективному использованию памяти.

Последние четыре строки используются непосредственно для передачи управления загруженной программе. Как и при запуске *COM*-программ в *MS-DOS*, на вершину стека помещается значение 0, что позволяет программе завершать своё выполнение обычной инструкцией `ret`: по смещению 0 в соответствующем сегменте окажется инструкция `int 20h`, записанная в начале *PSP*. Далее осуществляется дальний переход на начало программы с использованием инструкции `retf`. Инструкция `jmp` для этой цели менее удобна, т. к. требует либо указания фиксированного адреса, либо записи адреса перехода в память.

В момент завершения выполнения программы будет осуществлён вызов прерывания `20h`. Его обработчик может быть реализован, например, так:

```
Int20h:
        pop     ax
        pop     cx
        pop     dx
        push    cx
        push    dx
        push    cs
        push    System.Exec.ReturnFromProgram
        iret
```

В данном случае обработка прерывания сводится к тому, чтобы скорректировать содержимое стека и передать управление на метку `System.Exec.ReturnFromProgram`. В частности, при завершении программы неважно, по какому именно смещению находилась инструкция `int 20h`, единственная полезная информация – это номер сегмента, с которого начиналась память, отведённая программе.

В результате приведённых манипуляций в вершине стека формируются три элемента, позволяющие вернуться при выполнении инструкции `iret` в заранее известное место в коде ядра, причём с оставленным в вершине стека номером сегмента завершившейся программы. При таком подходе становится возможным собрать в одной процедуре как код запуска, так и код завершения программы.

```
.ReturnFromProgram:
     pop        es
     mov        ss, [es:$0004]
     mov        sp, [es:$0002]
     mov        ax, es
     call       Memory.Free
     xor        ax, ax
     jmp        .EndProc

.NotFound:
.ReadError:
     mov        ax, -1
     stc
.EndProc:
     ret
endp
```

При возврате из обработчика прерывания `20h` регистры `SS:SP` по-прежнему настроены на стек завершившейся программы, однако в вершине записан номер её стартового сегмента. По смещению +2 в этом сегменте сохранён полный адрес вершины стека, используемого ядром, который теперь можно восстановить. Память, которая была выделена программе, также может быть освобождена.

В данном примере при обработке ошибок не выполняется дифференциация причин, по которым запуск программы не удался, а также не реализован механизм возврата кода ошибки самой программой. Предполагается, что пример можно доработать самостоятельно, адаптировав его к особенностям работы собственной операционной системы.

Также не рассматривается способ предоставления прикладным программам доступа к функциям операционной системы. Самый простой вариант, аналогичный используемому в *MS-DOS*, заключается в том, чтобы задействовать один или несколько векторов прерываний (например, *MS-DOS* использует номера начиная с `20h`, а ***«OSE!»*** для предоставления основных своих возможностей задействует вектор `5Eh`). Для реализации других подходов (например, импорта функций, как это реализовано в *Windows*) может потребоваться поддержка со стороны формата исполняемых файлов.

Подведение итогов

Одной из важнейших задач, решаемых ядром операционной системы, является управление памятью. Необходимым минимумом при реализации менеджера памяти является поддержка операций выделения и освобождения блока, причём при освобождении блоков желательно предпринимать меры по дефрагментации свободного места. Для каждого выделяемого блока помимо служебной информации, необходимой непосредственно для управления памятью, полезно сохранять информацию, которая может быть использована при отладке: например, идентификатор программы, запросившей выделение памяти.

Запуск программы сопряжён с решением ряда инженерных задач, таких как выбор существующего или проектирование собственного формата исполняемых файлов, обеспечение совместимости с существующими видами исполняемых файлов, используемыми в других операционных системах (при необходимости), подготовка среды выполнения, необходимой для работы программы, и т. п. Конкретный перечень этих задач, а также их содержание определяются реализацией операционной системы.

Задания

1. Доработайте процедуру освобождения блоков в рассмотренном менеджере памяти для отслеживания ситуации повторного освобождения блока.

2. Доработайте процедуру освобождения блоков в рассмотренном менеджере памяти для отслеживания ситуации, когда переданный номер сегмента не является корректным адресом начала блока.

3. Доработайте рассмотренный менеджер памяти, добавив процедуры для выделения памяти, реализующие стратегии *best-fit, worst-fit* и *next-fit*. Проведите их сравнительный анализ.

4. Доработайте рассмотренный пример загрузки исполняемого файла, предоставив запускающей программе возможность получить код ошибки, возвращаемый запускаемой программой.

5. Поддержка программ, написанных для *MS-DOS*, потребует реализации по крайней мере части функций этой операционной системы. Предложите способ быстрого определения перечня используемых конкретной программой функций и передаваемых при этом параметров, не используя сторонних инструментов.

Шаг 4. Поддержка носителей с несколькими разделами

Разработка хоть сколько-нибудь серьёзной операционной системы невозможна без поддержки установки на жёсткий диск компьютера и загрузки с него. Как правило, это означает необходимость работы не только с конкретным диском и файловой системой, но и с одной из схем разбиения дисков на так называемые разделы. Ещё один случай, когда поддержка накопителей с несколькими разделами может оказаться полезной, – загрузка с *USB*-флэш-накопителей, которая на сегодняшний день является одним из наиболее популярных и удобных способов установки (а иногда и запуска) ОС.

Что такое *MBR*?

С появлением дисков большого объёма возникла необходимость их разбиения на разделы – логические области, каждая из которых представляется пользователю как самостоятельный носитель. Во-первых, хранение системных файлов отдельно от данных пользователя позволяет минимизировать возможность их случайного повреждения или удаления. Во-вторых, использование для каждого из разделов отдельной файловой системы позволяет оптимизировать работу с диском за счёт использования тех файловых систем, которые оптимальны для предполагаемого характера работы с разными видами данных. Наконец, в-третьих, поддержка разбиения дисков на разделы стала механизмом, который позволил устанавливать на один и тот же компьютер (даже с одним жёстким диском) несколько различных операционных систем.

Исторически сложившимся способом разбиения дисков на разделы стало использование для этих целей *MBR – Master Boot Record*, главной загрузочной записи. Идея заключается в том, чтобы в загрузочном секторе жёсткого диска записывать не загрузчик операционной системы, а код, который:

– просматривает список имеющихся на диске разделов;

– выбирает один из разделов в качестве загрузочного;

– считывает нулевой сектор загрузочного раздела (он ещё называется *VBR – Volume Boot Record*, загрузочная запись тома) в память так же, как это делала бы *BIOS*;

– передаёт управление коду считанного сектора.

При таком подходе загрузка операционной системы с жёсткого диска становится двухуровневой: сначала *BIOS* выбирает один из физических накопителей, а затем код *MBR* накопителя выбирает один из разделов (своего рода логический накопитель).

Как и в случае с загрузочным сектором *FAT*-раздела, необходимые для работы *MBR* сведения записываются и используются во время выполнения как глобальные данные. Расположение этих данных в коде *MBR* является общепринятым, что позволяет разрабатывать программы для управления разделами дисков, вносящие изменения в эти данные, не затрагивая собственно код поиска загрузочного раздела. Впрочем, следует иметь в виду, что в отличие от струк-

тур данных файловой системы, имеющей конкретных авторов, способ описания разделов в *MBR* изначально возник как вспомогательное техническое решение, которое впоследствии неоднократно расширялось и дополнялось зачастую не совместимыми между собой возможностями. По этой причине далее речь будет идти только о минимальном наборе возможностей *MBR*, поддержка которых получила широкое распространение.

Главной частью *MBR* является таблица разделов (*partition table*), состоящая из четырёх элементов и записываемая в загрузочном секторе диска непосредственно перед 2-байтовой загрузочной сигнатурой для BIOS. Типовая структура *MBR* для общего случая представлена в табл. 4.1.

Таблица 4.1

Типовая структура *MBR*

Смещение	**Размер, байт**	**Описание поля**
`000h`	446	Код *MBR*
`1BEh`	16	Первый элемент таблицы разделов
`1CEh`	16	Второй элемент таблицы разделов
`1DEh`	16	Третий элемент таблицы разделов
`1EEh`	16	Четвёртый элемент таблицы разделов
`1FEh`	2	Загрузочная сигнатура `55 AA`

В некоторых реализациях предпринимались попытки поддерживать более четырёх разделов в одной *MBR*, однако широкого распространения этот подход не получил, вследствие чего при использовании таких расширенных форматов корректная работа различных утилит и операционных систем с разделами, начиная с пятого, не гарантируется.

Формат элементов таблицы разделов приведён в табл. 4.2.

Таблица 4.2

Формат элемента таблицы разделов

Смещение	**Размер, байт**	**Описание поля**
+0	1	Свойства раздела
+1	3	*CHS* первого сектора
+4	1	Тип раздела
+5	3	*CHS* последнего сектора
+8	4	*LBA* первого сектора
+12	4	Кол-во секторов

Поскольку раздел диска логически является непрерывной областью, т. е. состоит из секторов, *LBA* которых последовательно возрастают, для описания раздела достаточно указать, где он начинается и где заканчивается. Эта информация приводится в записи двумя разными способами: для *CHS* задаются начальный и конечный секторы, для *LBA* – начальный сектор и количество секторов, образующих раздел. Формат для *CHS*-адресации адаптирован для максимально удобного использования с функциями сервиса `int 13h`, например, `int 13h/02h`.

Функция: `AH = 02h`
Посекторное чтение с диска (для жёстких дисков)
Считывает заданное количество секторов с диска.

Исходные данные:	`AH` = `02h`
	`AL` = количество секторов, подлежащих считыванию
	`CH` = младшие 8 бит номера цилиндра
	`CL` = номера цилиндра и сектора
	биты 7–6 = старшие 2 бита номера цилиндра
	биты 5–0 = номер сектора
	`DH` = номер головки
	`DL` = номер диска
	`ES:BX` = указатель на буфер, в который должны быть прочитаны данные
Результаты:	`AL` = количество прочитанных секторов
	`AH` = 0, если функция выполнилась успешно, иначе – код ошибки
	`CF` = 0, если функция выполнилась успешно, иначе – 1

Количество считываемых секторов должно быть строго меньше 128, но не равно 0.

В некоторых *BIOS* ошибка чтения может быть вызвана тем, что в момент запроса на чтение двигатель, вращающий диск, был отключён, а *BIOS* не ожидает достижения требуемой скорости вращения. В этом случае необходимо произвести переинициализацию устройства (функцией `int 13h/00h`) и повторить попытку чтения три раза, чтобы убедиться, что ошибка неустранима.

При записи *CHS*-значений в элементах таблицы разделов значения отдельных байтов соответствуют значениям регистров `DH`, `CL` и `CH` при вызове этих функций соответственно. Следует иметь в виду, что для разделов, занимающих секторы, которые невозможно адресовать с использованием *CHS*, в эти поля, как правило, записывают значения (1023, 254, 63), а при формировании таблицы разделов для дисков, использующих более новый подход к разметке – *GPT* (будет рассмотрен позже), – значения (1023, 255, 63). Следовательно, если для какого-либо из разделов в таблице записаны подобные значения, следует использовать данные из полей, задающих расположение раздела по схеме *LBA*.

Байт по смещению 0 в элементе таблицы разделов содержит информацию о свойствах раздела. Единственная договорённость, которая поддерживается подавляющим большинством реализаций, заключается в том, что старший бит в этом байте устанавливается в 1, если соответствующий раздел является актив-

ным, и в 0 в противном случае. Как правило, значения, отличные от `$00` и `$80`, считаются ошибочными, однако предпринимались попытки использовать остальные 7 бит для хранения дополнительной информации в случае, если раздел активен. Существенной поддержки эта идея не получила.

Активный раздел – раздел, с которого должна производиться загрузка для данного диска. В большинстве реализаций кода для *MBR* предполагается, что в каждый момент времени только один из разделов может быть активным, причём в некоторых реализациях попытка сделать активными несколько разделов одновременно приведёт к выводу сообщения об ошибке, тогда как в других просто будет выбран первый из активных разделов.

Байт по смещению +4 используется для указания типа раздела. Под типом раздела обычно понимают информацию о его назначении и/или используемой файловой системе. Некоторые наиболее широко используемые значения для этого поля приведены в табл. 4.3.

Таблица 4.3

Некоторые значения для поля типа раздела

Значение	**Описание**
`00h`	Неиспользуемый элемент
`01h`, `06h`	Раздел, отформатированный в *FAT*12
`04h`, `06h`	Раздел, отформатированный в *FAT*16
`05h`	Расширенный раздел
`07h`	Раздел, отформатированный в *NTFS* или *exFAT*
`0Bh`	Раздел, отформатированный в *FAT*32 и использующий *CHS*-адресацию
`0Ch`	Раздел, отформатированный в *FAT*32 и использующий *LBA*-адресацию
`27h`	Раздел восстановления, отформатированный в *NTFS*
`7Fh`	Значение зарезервировано для использования в экспериментальных проектах и для временных нужд

Официально поддерживаемого списка значений, которого бы придерживались все разработчики, не существует, поэтому указанные значения в таблице носят исключительно справочный характер. Их можно использовать, например, для того, чтобы определить, драйвер какой файловой системы должен быть за-

гружен в первую очередь при поиске подходящего для данного раздела, однако следует иметь в виду, что возможна ситуация, когда на анализируемом разделе в действительности используется другая файловая система, для которой было переиспользовано общепринятое значение поля.

Часто некоторое количество секторов, следующих за *MBR*, не включается ни в один из разделов. Это может быть полезно, например, для записи в подобную неиспользуемую область дополнительного кода, который позволит не просто выбрать текущий активный раздел в качестве загрузочного и передать управление его *VBR*, а, например, отобразить при загрузке меню выбора одного из разделов, что может быть полезно при установке на один диск нескольких операционных систем.

Написание кода для *MBR* не является первостепенной задачей для разработчика операционной системы, т. к. нередко этот код может оказаться заменённым из-за использования специального программного обеспечения для управления разделами (наподобие *Partition Magic*) или в результате установки ещё одной операционной системы, которая выполнит перезапись *MBR* своей реализацией. По сути, единственный случай, ради которого потребуется разработать хотя бы простейший код для *MBR,* – это установка операционной системы на новый жёсткий диск, который ещё не был разбит на разделы. Однако даже в этом случае эту реализацию целесообразно включить не в состав ядра и основных компонентов операционной системы, а сделать частью утилиты (т. е. обычной прикладной программы) для управления разделами на дисках. Тем не менее, понимание того, как определяются разделы диска, потребуется для поддержки этой возможности самой операционной системой.

Чем *MBR* отличается от *GPT*?

В ходе работы над спецификацией *UEFI* помимо ограниченности программного интерфейса классических *BIOS* решалась также и проблема ограниченности *MBR*-разметки дисков. Результатом стало появление нового способа разметки – *GPT.*

Самое очевидное ограничение *MBR*-разметки – ограничение в четыре раздела на диск. Различные приёмы, которые использовались для его обхода, имели свои ограничения и в конечном счёте делали разбиение диска на количество разделов, превышающее 4, нецелесообразным. Разумеется, для среднестатистического пользователя это несущественно, однако сам факт наличия такого (достаточно жёсткого) ограничения потенциально мог создать серьёзные сложности, если бы при дальнейшем развитии информационных технологий возникла потребность в использовании большого числа разделов на дисках.

Кроме того, во второй половине 1990-х годов, пока велась подготовка спецификации *UEFI,* появилась спецификация *EDD*, позволившая использовать для жёстких дисков *LBA*-адресацию. К этому моменту стало вполне очевидным, что отказ от *CHS*-адресации для жёстких дисков рано или поздно произойдёт: во-первых, из-за сложностей в её использовании, описанных ранее, а во-

вторых, из-за того, что с развитием оборудования самих жёстких дисков их производители зачастую начали реализовывать собственную логику нумерации секторов, механизмы автоматического переназначения секторов на случай повреждения части из них и т. п., вследствие чего использование адресации, привязанной к геометрии жёсткого диска, начало терять смысл.

Закономерным следствием этого стал отказ от использования *CHS*-адресации в новой схеме разметки дисков. Кроме того, для значений *LBA* теперь отводится по 8 байт, что решило проблему ограничения максимального размера диска на довольно продолжительное время.

Третья проблема, попытку решения которой предприняли разработчики *GPT*-разметки, – сложность идентификации типа раздела. Во-первых, 256 различных значений для определения того, с какой целью создан раздел и какой способ записи данных он использует, – достаточно малая величина для подхода, который предполагается использовать по меньшей мере столько же, сколько использовалась *MBR*-разметка – десятки лет. Во-вторых, существенное влияние оказали и наметившиеся тенденции развития отрасли в целом: рост популярности решений, основанных на децентрализации (отсутствии единого центра, являющегося накопителем и эталонным источником той или иной информации), нашёл своё отражение в возможности идентификации используемых типов разделов без согласования с другими разработчиками.

В основу решения этой проблемы было положено использование т. н. *GUID* – глобально уникальных идентификаторов. Это 128-битные величины, которые генерируются таким образом, чтобы обеспечить их уникальность в пространстве и времени. Для этого биты объединяют в три группы в зависимости от назначения: часть битов содержит информацию о версии использованного алгоритма генерации *GUID* (их существует несколько, и отличаются они тем, на основе какой информации заполняются остальные поля), часть – о месте (например, оборудовании компьютера, который использовался для генерации) и времени генерации. Считается, что совпадение двух правильно сгенерированных *GUID* практически невозможно ввиду способа генерации и количества различных значений, которые они могут принимать.

UEFI при поиске загрузчика перебирает все разделы на всех подключённых к компьютеру дисках, имеющих *GPT*-разметку. Таким образом, использование этого способа разметки будет обязательным, если потребуется обеспечить загрузку операционной системы в режиме *UEFI*. И классическая *BIOS*, и *CSM* обеспечивают более простую схему загрузки, для которой достаточно *MBR*-разметки.

Размещение служебных структур данных на диске при использовании *GPT*-разметки в соответствии со спецификацией *UEFI* приведено на рис. 4.1.

Рис. 4.1. Размещение служебных структур данных при использовании *GPT*-разметки

Для совместимости с классическими *BIOS* нулевой сектор диска содержит классическую *MBR*. Применительно к *GPT*-разметке эта *MBR* называется защитной. Её назначение заключается в том, чтобы не позволить программному обеспечению, поддерживающему только *MBR*-разметку, повредить структуры данных, характерные для *GPT*-разметки. Для этого таблицу разделов рекомендуется заполнить следующим образом:

– первый элемент задаёт защитный раздел, охватывающий всё пространство на диске;

– остальные три элемента заполняются нулями, что означает отсутствие других разделов на диске.

В табл. 4.4 приведены рекомендуемые значения для описания защитного раздела.

Таблица 4.4

Рекомендуемые значения полей для описания защитного раздела

Смещение	**Размер, байт**	**Рекомендуемое значение**
+0	1	`$00` (неактивный раздел)
+1	3	`$000200` (начинается с сектора с *LBA* 1)
+4	1	`$EE` (защитный раздел для *GPT*-разметки)
+5	3	*CHS* последнего сектора на диске или `$FFFFFF`, если размеры диска превышают возможности *CHS*-адресации
+8	4	`$00000001` (начинается с сектора с *LBA* 1)
+12	4	Размер диска в секторах минус 1 или, если это число не представимо в 4-байтовой величине, `$FFFFFFFF`

В случае если загрузка производится в режиме *UEFI* без *CSM*, содержимое защитной *MBR* игнорируется, а информация о разделах считывается из последующих секторов.

Первый из этих секторов, имеющий *LBA* 1, используется для хранения структуры, которая называется *GPT header* и является заголовком таблицы разделов. Предполагаемое содержимое этого сектора и назначение отдельных его полей приведено в табл. 4.5.

Таблица 4.5

Формат заголовка таблицы разделов

Смещение	Размер, байт	Описание
+0	8	Сигнатура заголовка таблицы разделов. Это поле должно содержать значение `"EFI PART"`, что эквивалентно 64-битной константе `$5452415020494645` (*little-endian*)
+8	4	Номер ревизии (версии) заголовка таблицы разделов. Для описываемой здесь ревизии используется значение `$00010000`
+12	4	Размер заголовка таблицы разделов в байтах. Это значение должно быть больше или равно 92, но не превышать размера сектора
+16	4	Контрольная сумма *CRC*32 для заголовка таблицы разделов. При вычислении контрольной суммы это поле принимается равным `$00000000`, а сама контрольная сумма вычисляется для количества байтов, указанного в предыдущем поле, т. е. для размера заголовка, а не всего сектора
+20	4	Зарезервировано. Должно быть равно 0
+24	8	*LBA* сектора, содержащего заголовок таблицы разделов
+32	8	*LBA* альтернативного заголовка таблицы разделов (см. пояснения ниже)
+40	8	*LBA* первого сектора области данных (т. е. первого из секторов, которые могут быть использованы для создания разделов)
+48	8	*LBA* последнего сектора области данных (т. е. последнего из секторов, которые могут быть использованы для создания разделов)
+56	16	*GUID* диска, содержащего данную таблицу разделов
+72	8	*LBA* сектора, с которого начинаются записи таблицы разделов
+80	4	Количество элементов в таблице разделов
+84	4	Размер элементов таблицы разделов, в байтах
+88	4	Контрольная сумма *CRC*32 таблицы разделов. Размер данных, используемых для вычисления контрольной суммы, определяется перемножением двух предыдущих полей
+92	Размер сектора минус 92	Зарезервировано. Должно быть заполнено нулевыми значениями

Таблица разделов, как правило, начинается со следующего сектора и состоит из заданного в заголовке количества элементов. Формат отдельных элементов таблицы разделов приведён в табл. 4.6.

Таблица 4.6

Формат элемента таблицы разделов

Смещение	Размер, байт	Описание
+0	16	*GUID* типа раздела
+16	16	*GUID* раздела
+32	8	*LBA* первого сектора раздела
+40	8	*LBA* последнего сектора раздела
+48	8	Атрибуты раздела
+56	72	*C*-строка, содержащая человекочитаемое имя раздела
+128	Размер элемента минус 128	Зарезервировано. Должно быть заполнено нулевыми значениями

Резервные (альтернативные) копии таблицы разделов и её заголовка записываются в последних секторах диска. *LBA* альтернативного заголовка, как правило, равен наибольшему значению *LBA* для данного диска, однако точные значения *LBA* для каждого из его расположений записываются в самом заголовке.

При создании загрузочного накопителя потенциально возможно разметить его таким образом, чтобы таблицы разделов *MBR* и *GPT* описывали один и тот же набор разделов. В этом случае накопитель может быть использован для загрузки с него, независимо от того в каком режиме она производится. Этот приём, в частности, можно применить при создании установочного носителя.

Пример 4.1. Загрузка с *USB*-флэш-накопителя

Одними из наиболее удобных в эксплуатации устройств, которые можно использовать в качестве загрузочных, являются *USB*-флэш-накопители. В ходе разработки операционной системы бывает удобно отформатировать такой накопитель в одну из поддерживаемых файловых систем (чаще всего это одна из версий *FAT*) и реализовать загрузку ядра ОС из файла, как было показано ранее. После этого для тестирования изменений, вносимых в операционную систему, на реальном физическом устройстве (вместо виртуальных машин и эмуляторов) достаточно будет скопировать изменённые файлы на такой накопитель и перезагрузить компьютер.

Вместе с тем при работе с флэш-накопителями больших размеров отводить всё доступное пространство для загрузочного раздела операционной системы обычно нерационально, поскольку, как правило, размеры любительских и учебных операционных систем сравнительно невелики и большая часть накопителя окажется неиспользуемой. Кроме того, загрузка с *USB*-флэш-накопителя в режиме классической *BIOS* или с использованием *UEFI CSM* имеет ряд особенностей, которые также следует учитывать при подготовке накопителя.

В различных реализациях *BIOS* можно встретить следующие способы загрузки с *USB*-флэш-накопителя:

– в режиме эмуляции флоппи-диска (дискеты);
– в режиме жёсткого диска;
– в режиме эмуляции оптического диска (*CD*/*DVD*).

Самый простой для начинающего разработчика режим – режим эмуляции дискеты. Если целевая *BIOS* поддерживает этот способ загрузки, для подготовки накопителя достаточно записать образ дискеты начиная с нулевого сектора *USB*-флэш-накопителя. В зависимости от реализации *BIOS* параметры эмулируемой дискеты могут выбираться как с учетом предположения, что это дискета определённого формата (наиболее вероятный – 3,5″; 1,44 Мбайт), так и исходя из того, что дискета отформатирована в *FAT*12, – в этом случае геометрия эмулируемой дискеты может считываться из загрузочного сектора. Впрочем, несмотря на всю простоту такого решения, его поддержка, особенно в современных реализациях *CSM*, встречается нечасто. Рассмотрение ещё одного способа – эмуляции оптического диска – выходит за рамки данного пособия.

Наиболее надёжный способ загрузки с *USB*-флэш-накопителя – в режиме жёсткого диска. Он поддерживается подавляющим большинством реализаций *BIOS* и *CSM*. Для максимальной совместимости с этим способом необходимо использовать *MBR*-разметку и создать на накопителе активный раздел, который и будет содержать операционную систему.

Дополнительным преимуществом загрузки в режиме жёсткого диска является возможность создания на *USB*-флэш-накопителе небольшого по размеру раздела с тестируемой операционной системой и распределения остального пространства под раздел с данными. В этом случае накопитель можно будет использовать для удобного тестирования ОС, не теряя при этом возможность по-прежнему применять его для переноса данных между компьютерами.

Рассмотрим способ разбиения такого накопителя указанным образом с использованием встроенных средств операционной системы *Windows*.

Перед выполнением описываемых далее действий следует скопировать хранящиеся на *USB*-флэш-накопителе данные на другие носители. В ходе подготовки накопителя все записанные на нём данные станут недоступными.

Для управления разделами на *USB*-флэш-накопителе будем использовать утилиту *diskpart*, поставляемую с операционной системой *Windows*. Запустить её можно из командной строки по имени:

```
> diskpart
```

После запуска утилита выводит базовую информацию и отображает приглашение к вводу команд, например:

```
Microsoft DiskPart, версия 10.0.17763.1554

(C) Корпорация Майкрософт (Microsoft Corporation).
На компьютере: HP250G6

DISKPART>
```

Утилита *diskpart* использует свой собственный набор команд, позволяющих отображать информацию о подключённых к компьютеру дисках и доступных на них разделах, вносить изменения в разбиение дисков на разделы, изменять их атрибуты и т. п. Для получения подробной информации о командах, доступных в конкретной версии, можно воспользоваться следующей командой:

```
DISKPART> help
```

Для получения справки о конкретных командах можно указать их после `help`, например:

```
DISKPART> help list
```

В каждый момент времени утилита считает текущим один из дисков и один из разделов на нём. Все команды, вносящие те или иные изменения, работают с текущим диском или разделом. Всем объектам (как дискам, так и разделам) утилита назначает номера, которые следует использовать при выборе текущего объекта для управления. Для того чтобы сделать текущим *USB*-флэш-накопитель, следует сначала перечислить доступные диски:

```
DISKPART> list disk

  Диск ###  Состояние      Размер   Свободно Дин  GPT
  --------  -------------  -------  -------  ---  ---
  Диск 0    В сети          931 Gбайт  1024 Кбайт
  Диск 1    В сети         3824 МБайт      0 байт
```

В приведённом примере к компьютеру подключены встроенный жёсткий диск с заявленным объёмом в 1 Тбайт и *USB*-флэш-накопитель с заявленным объёмом в 4 Гбайт. Для того чтобы продолжить работу с последним, следует выбрать его:

```
DISKPART> select disk 1

Выбран диск 1.
```

После этого можно просмотреть список разделов, доступных на этом накопителе:

```
DISKPART> list partition

  Раздел    ###  Тип               Размер   Смещение
  ------------  ----------------  -------  -------
  Раздел 1    Основной          3823 МБ    64 Кб
```

На рассматриваемом накопителе создан единственный раздел, занимающий почти всё свободное пространство. Пересоздадим разделы в соответствии с ранее описанной схемой. Для начала следует удалить все имеющиеся на диске разделы:

```
DISKPART> clean

DiskPart: очистка диска выполнена успешно.
```

После выполнения этого действия на накопителе не остаётся ни одного раздела, всё доступное пространство считается неразмеченным:

```
DISKPART> list partition

Разделы на диске отсутствуют.
```

Создадим первый раздел, который будет использоваться для тестирования разрабатываемой операционной системы. Исходим из того, что 5 Мбайт для этого раздела будет достаточно (во всяком случае изначально):

```
DISKPART> create partition primary size=5

DiskPart: указанный раздел успешно создан.
```

Созданный раздел появляется в списке разделов:

```
DISKPART> list partition

  Раздел    ###  Тип               Размер   Смещение
  ------------  ----------------  -------  -------
* Раздел 1    Основной          5120 Кб    64 Кб
```

Раздел может быть сразу же отформатирован в одну из поддерживаемых файловых систем. Для получения их списка можно воспользоваться следующей командой:

```
DISKPART> filesystems
```

Тем не менее необходимости в форматировании этого раздела средствами *diskpart* нет, т. к. впоследствии придётся выполнить перезапись его загрузочного сектора другими утилитами, что само по себе эквивалентно форматированию раздела.

Создадим второй раздел, который будет использоваться для хранения пользовательских данных:

```
DISKPART> create partition primary

DiskPart: указанный раздел успешно создан.
```

Если параметр `size` не указан, создаётся раздел, занимающий всё оставшееся свободное пространство.

```
DISKPART> list partition

  Раздел    ###  Тип               Размер   Смещение
  ----------  ----------------  -------  -------
  Раздел 1    Основной          5120 Кб    64 Кб
* Раздел 2    Основной          3818 МБ  5184 Кб
```

Созданный раздел автоматически выбирается в качестве текущего. Для его форматирования, например, в *FAT*, можно воспользоваться следующей командой:

```
DISKPART> format fs=FAT quick

  Завершено (в процентах): 100

Программа DiskPart успешно отформатировала том.
```

Для того чтобы сделать возможной загрузку из первого из разделов, следует сделать его активным. Для начала выберем его:

```
DISKPART> select partition 1

Выбран раздел 1.
```

Теперь, когда раздел выбран, его можно сделать активным:

```
DISKPART> active

DiskPart: раздел помечен как активный.
```

На этом необходимый минимум действий выполнен, всю остальную подготовку удобнее выполнять с использованием других инструментов. Для завершения работы с утилитой используется следующая команда:

```
DISKPART> exit
```

Остаётся выполнить несколько действий:

– отформатировать активный раздел, записав в его загрузочный сектор (*VBR*) код своего загрузчика;

– скопировать на активный раздел необходимые файлы;

– назначить букву разделу, который будет использоваться для хранения данных.

При перезаписи загрузочного сектора следует учесть, что активный раздел начинается не с нулевого сектора на диске, а с некоторым смещением от его начала. В файловой системе *FAT* для записи этой информации предусмотрено поле `BPB_HiddSec`. Код загрузчика должен перед вызовом функций работы с диском прибавлять записанное в этом поле значение ко всем номерам секторов в пределах раздела. В приведённом примере первый созданный утилитой *diskpart* раздел располагается начиная с сектора с *LBA* 128 – именно это значение должно быть записано в поле `BPB_HiddSec`.

При разработке кода загрузчика следует также иметь в виду, что, как правило, и для *USB*-флэш-накопителей в режиме жёсткого диска, и для настоящих жёстких дисков современные *BIOS* поддерживают набор функций из спецификации *EDD*, позволяющих обращаться к секторам с использованием *LBA*-адресации, причём *CHS*-адресация ввиду ограниченности максимального размера поддерживаемых ею дисков (около 8 Гбайт при использовании 24 байт, как в *MBR* и функциях *BIOS*) может быть вообще неприменимой. Исключением является загрузка в режиме эмуляции дискеты: в этом случае, как правило, доступна только *CHS*-адресация.

Подведение итогов

Одной из полезных возможностей даже для любительских операционных систем является поддержка дисков с несколькими разделами. Исторически сложившимся способом разметки диска (указания того, какая часть доступного пространства считается каким из разделов) является *MBR*-разметка, однако всё чаще вместо неё отдают предпочтение *GPT*-разметке, снимающей ряд ограничений по сравнению с *MBR*, а также являющейся основным видом разметки для *UEFI*-систем.

С необходимостью разбиения диска на разделы можно столкнуться при попытке запуска операционной системы (или её установочного модуля) с *USB*-флэш-накопителей. Наиболее надёжным способом загрузки с них является загрузка в режиме жёсткого диска. В этом случае важно пометить загрузочный раздел в таблице разделов как активный, а также проследить, чтобы код загрузчика корректно производил вычисления, связанные с адресацией секторов, а именно – учитывал, что загрузочный раздел располагается не в самом начале диска, а с некоторым смещением.

Задания

1. Разработайте утилиту, которая позволит выполнять подготовку *USB*-флэш-накопителя в *MBR*-разметке. В качестве исходных данных на вход утилите подаётся образ загрузочного раздела, затем выбирается накопитель, который должен быть подготовлен. Предусмотрите возможность автоматической корректировки значений некоторых полей (например, `BPB_HiddSec` в случае *FAT*) при записи образа на создаваемый загрузочный раздел, а также возможность расширения набора поддерживаемых файловых систем.

2. Доработайте утилиту из п. 1 для создания гибридных загрузочных накопителей. Для *GPT*-разметки в качестве исходных данных следует указывать файл загрузчика, а также каталоги и файлы, которые должны быть созданы/скопированы на накопитель.

3. Реализуйте утилиту для учебной операционной системы ***«OSE!»*** (или своей операционной системы), позволяющую управлять разбиением дисков на разделы.

4. Доработайте утилиту из п. 3, добавив возможность форматирования создаваемых разделов.

5. Доработайте утилиту из п. 3, добавив возможность изменения размеров существующих разделов, их перемещения, изменения используемой файловой системы и т. д. без потери хранящихся в этих разделах данных. Предусмотрите возможность расширения набора поддерживаемых файловых систем.

Список использованных источников

1. Intel® 64 and IA-32 Architectures Software Developer's Manual. – [USA] : Intel Corporation, 2021. – 4778 p.

2. 8086 Family Users Manual. – Santa Clara : Intel Corporation, 1979. – 208 p.

3. IBM/2 and PS BIOS Interface Technical Reference. – 1st ed. – [USA] : International Business Machines, 1987. – 218 p.

4. System BIOS for IBM PC / XT / AT computers and compatibles. – 5th ed. – [USA] : Phoenix Technologies, 1990. – 554 p.

5. Information Technology – BIOS Enhanced Disk Drive Services – 3 (EDD-3). T13. – Rev. 3. – [USA] : American National Standards Institute, 2004. – 58 p.

6. Microsoft Extensible Firmware Initiative FAT32 File System Specification. – Ver. 1.03. – [USA] : Microsoft Corporation, 2000.

7. Unified Extensible Firmware Interface (UEFI) Specification Version 2.8 [Электронный ресурс]. – 2022. – Режим доступа : https://uefi.org/specs/UEFI/2.10.

8. TechHelp 6.0 [Электронный ресурс] : программное средство. – [USA] : Flambeaux Software, 1994. – 1 электрон. опт. диск (CD-ROM).

Св. план 2021, поз. 22

Учебное издание

Оношко Дмитрий Евгеньевич
Бахтизин Вячеслав Вениаминович

ОСНОВЫ РАЗРАБОТКИ ОПЕРАЦИОННЫХ СИСТЕМ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *С. Г. Девдера*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *О. И. Толкач*

Подписано в печать 04.10.2022. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 7,32. Уч.-изд. л. 8,00. Тираж 40 экз. Заказ 170.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014.
№2/113 от 07.04.2014, №3/615 от 07.04.2014,
Ул. П. Бровки, 6, 220013, г. Минск