## Пишем свою операционную систему. Предисловие

|  |
| --- |
| Доброго времени суток!  В последнее время можно наблюдать некоторый спад интереса к osdev-тематике. На самом Subscribe.ru мне удалось найти лишь три рассылки, и все они не выпускаются уже больше года, а имеет более-менее большое количество интересного материала лишь одна. Можно подумать, что рассылки доходят до своего логического завершения и не выпускаются, потому что больше нечего писать, но это не так - все рассылки обрываются на середине разработки учебной операционной системы, так и не реализовав и десятой части необходимых функций. Возможно, пропал интерес разработчиков создавать новые проекты, но я надеюсь, что ещё не пропали желающие читать про всё это и принимать посильное участие в уже готовых проектах, иначе моя рассылка не имеет смысла (куда я без подписчиков?).  Зачем вообще нужна эта рассылка? Во-первых, построение новой операционной системы как нельзя лучше подходит для изучения принципов работы уже существующих, потому что можно наблюдать постепенное, поэтапное, усложнение кода ядра и приложений, "детские" проблемы, с которыми сталкивается очередной разработчик и пути их решения, которые когда-то давно повлияли и на развитие "больших" ОС. Во-вторых, не смотря на всё чего достигли популярные операционный системы (Windows, Linux) за последние десятилетия их развитие несколько однобоко, меняется внешний вид, но используются низкоуровневые концепции заложенные ещё в прошлом веке. Достаточно сказать, что обе эти операционные системы используют самую простую архитектуру ядра - монолит, хотя уже были придуманы более интересные и функциональные подходы (например, микроядро). Если всё пройдёт удачно, то то, что у нас получится в результате этой рассылки, готовая небольшая операционная система, станет самостоятельным проектом. Надеяться на то, что она станет "убийцей Windows" было бы слишком оптимистично, но она может занять какое-то место хотя бы в области встраиваемых систем, пусть даже для любителей.  Для начала определимся с используемыми для разработки инструментами. В этом выпуске не будет строчек программ, это всего лишь вступление, предисловие.   * flat assembler - компилятор для языка Ассемблера. Конечно, при желании вы можете использовать любой другой компилятор, но по личным соображениям я предпочитаю этот как наиболее функциональный и гибкий. Все примеры программ в рассылке будут ориентированы именно на него. Скачать последнюю версию для Windows или Linux вы всегда можете на официальном сайте: [flatassembler.net](http://flatassembler.net/download.php) * GCC - компилятор для языка программирования Си. Разрабатывать ядро и другие сложные компоненты системы полностью на Assembler было бы не рационально. Во-первых, теряется наглядность и внешняя простота кода. Во-вторых, перенос системы на другую платформу (встраиваемые системы, помните?) потребует практически полного переписывания системы с нуля. Мы будем использовать компилятор этого языка от GNU. Если вы используете ОС Linux, то он уже скорее всего уже у вас установлен, либо вы знаете как его установить из стандартных репозиториев. Пользователи же Windows могут скачать установщик MinGW с его официального сайта - [www.mingw.org](http://www.mingw.org/) * Виртуальная машина Bochs - в наше время быстрых процессоров крайне не удобно перезагружаться ради каждой проверки новой операционной системы. С учётом того, что на первых этапах разработки она не будет способна выполнять никаких полезных функций. К тому же далеко не у каждого пользователя сейчас есть дисковод для гибких дисков (попросту говоря - дискет), а использовать для этого флешку - необходимо искать чистую, данные с которой не жалко потерять (запись образа нашей ОС уничтожит данные и потом флешку придётся заново форматировать). Да и работа с USB в ядре ОС гораздо более сложна. Почему именно Bochs? Он не отличается высокой производительностью, но она нам и не нужна, ведь на первых шагах, наша ОС будет мало что делать, зато он обладает по истине огромными возможностями для настройки эмулируемой среды и отладки кода системы, что очень важно для нас, как для разработчиков. Скачать последнюю версию можно по адресу [bochs.sourceforge.net](http://bochs.sourceforge.net/). Пользователи большинства дистрибутивов Linux могут установить и эту программу из своих репозиториев, либо собрать из исходников с этого же сайта. * Шестнадцатеричный редактор - иногда бывает необходимо просмотреть содержимое файлов естественном для компьютере представлении, а то и подредактировать его. Пользователям Windows могу посоветовать очень хороший бесплатный редактор [HxD](http://mh-nexus.de/en/hxd/). С его же помощью вы сможете и записать загрузочный образ системы на флешку или дискету.   Ну вот и всё для начала, пожалуй. Все перечисленные выше программы совершенно бесплатны, поэтому у вас не должно возникнуть проблем с их получением и установкой. Для того, чтобы продолжить дальше необходимо иметь их все в наличии, потому что я не буду в будущем заострять внимание на инструментах. Также я уже предполагаю некоторое знание языка Assembler и не буду рассказывать основы, потому что тогда рассылка получится слишком большой и скучной для тех, кто уже что-то знает.  Все возникшие вопросы и предложения (кроме, пожалуй, "автор убейся - твоя ОС никому не нужна") вы можете отправлять в любое время дня и ночи на мой личный адрес электронной почты [kiv.apple@gmail.com](mailto:kiv.apple@gmail.com). Я постараюсь ответить как только смогу, только, пожалуйста, указывайте, что вы обратились ко мне именно в результате чтения моей рассылки.  Благодарю за внимание. До встречи ;-) |

## Пишем свою операционную систему. Немного теории и начальный загрузчик.

|  |
| --- |
| Приветствую всех своих читателей!  В этом выпуске мы приступим к написанию не очень сложной, но достаточно важной части любой операционной системы - начального загрузчика. Именно эта часть обычно присутствует абсолютно во всех рассылках, но её не стоит пропускать, потому что без загрузчика мы не сможем производить какие-либо действия. Можно очень много рассуждать о структуре ядра ОС, однако без загрузчика мы никак не сможем проверить свои идеи на практике. Конечно, существуют универсальные загрузчики вроде GRUB, но хотелось бы рассмотреть аспект написания системы наиболее полно. Тем, кому интересует исключительно программирование самого ядра, придётся подождать пока мы научимся его запускать :-)  **Теория**  Итак, начнём с теоретической части. Весь обмен данными с носителями информации вроде дискет, жёстких дисков и флешек осуществляется только блоками фиксированными размера - секторами. Как в памяти компьютера нельзя непосредственно обратиться к единице меньшей, чем байт, так на диске все операции чтения и записи выполняются посекторно. Самый часто используемый размер сектора - 512 байт, хотя есть и немногочисленные исключения - на CD-дисках, например размер сектора 2 килобайта. Но сейчас мы не рассматриваем загрузку с последних, так что можно считать, что размер сектора ровно 512 байт, не больше и не меньше.  После включения компьютера управление получает BIOS - Basic Input-Output System. Он проводит первичное тестирование оборудования, предоставляет интерфейс для настройки некоторых компонентов (например, часов) и наконец загружает 0-ой сектор загрузочного диска (определяется в настройках), передавая ему управление. Поскольку 512 байт слишком мало, чтобы там можно было разместить полноценный драйвер для работы с диском, BIOS предоставляет все необходимые функции для работы с экраном, клавиатурой и мышью. Их более чем достаточно для любого начального загрузчика.  В последнее время внедряется альтернативная система инициализации - EFI (Extensible Firmware Interface). Возможно, когда-нибудь мы рассмотрим и её, но не раньше, чем напишем ядро ОС. С одной стороны эта система отбрасывает некоторые устаревшие технологии, но с другой стороны требует б**о**льших теоретических знаний для начала работы. К тому же подавляющее большинство версий EFI на сегодняшний день поддерживают режим эмуляции BIOS и наша система сможет работать и с ними. Да и пока достаточно мало виртуальных машин поддерживают загрузку EFI-совместимых систем.  **Практика**  Приступим к написанию начального загрузчика (boot loader) для нашей системы. Поскольку мне отнюдь не симпатизирует идея написания жёсткого монолита (скорее я предпочитаю микроядра), в задачу загрузчика будет входить не только загрузка ядра, но и модулей, необходимых для дальнейшей инициализации (например, драйвер жёсткого диска), которые содержатся в отдельных файлах.  Пока для простоты будем рассматривать загрузку с диска без таблицы разделов. Скажем, дискеты, хотя теоретически ничто не мешает реализовать такую же структуру и на других носителях (хотя это и не совсем правильно).  Как я уже сказал выше, BIOS загружает в оперативную память первые 512 байт диска и передаёт им управление. Наш код оказывается в реальном режиме работы процессора (те, кто не знают что это, идут читать какой-нибудь учебник по Assembler) по адресу 0000:7C00. Прерывания запрещены, в регистре DL находится номер загрузочного диска (например, 0 и 1 для дискет, начиная с 0x80 идут все прочие виды дисков). Некоторые сегментные регистры указывают на область данных BIOS, другие регистры также могут содержать дополнительную информацию, но на это лучше не рассчитывать, потому что многое зависит от деталей реализации конкретного BIOS. Также стоит отметить, что последние два байта начального загрузчика должны содержать сигнатуру 0x55,0xAA, иначе многие BIOS посчитают такой загрузчик некорректным и откажутся запускать.  Начнём с заготовки загрузчика, который способен просто запуститься, настроить всё окружение, вывести пару строчек, подождать нажатия на клавиатуру и перезагрузить машину при любом нажатии. На этом пока и остановимся, иначе этот выпуск рассылки получится слишком длинным. Загрузчик будет выполнять некоторые ненужные сейчас действия (например, запоминание номера диска, с которого мы стартовали), но они нам пригодятся, когда в следующей части мы напишем его полноценную (способную что-то загрузить) версию.  org 0x7C00  jmp word boot  ; Данные начального загрузчика  label disk\_id byte at $$  boot\_msg db "MyOS boot loader. Version 0.04",13,10,0  reboot\_msg db "Press any key...",13,10,0  ; Вывод строки DS:SI на экран  write\_str:  push ax si  mov ah, 0x0E  @:  lodsb  test al, al  jz @f  int 0x10  jmp @b  @:  pop si ax  ret  ; Критическая ошибка  error:  pop si  call write\_str  ; Перезагрузка  reboot:  mov si, reboot\_msg  call write\_str  xor ah, ah  int 0x16  jmp 0xFFFF:0  ; Точка входа в начальный загрузчик  boot:  ; Настроим сегментные регистры  jmp 0:@f  @:  mov ax, cs  mov ds, ax  mov es, ax  ; Настроим стек  mov ss, ax  mov sp, $$  ; Разрешим прерывания  sti  ; Запомним номер загрузочного диска  mov [disk\_id], dl  ; Выводим приветственное сообщение  mov si, boot\_msg  call write\_str  ; Завершение  jmp reboot  ; Пустое пространство и сигнатура  rb 510 - ($ - $$)  db 0x55,0xAA  Опишу некоторые аспекты работы этого кода.  Обычно, после первых 3-4 байт загрузчика размещает заголовок некоторых файловых систем (например, FAT), поэтому первым делом мы обходим все данные и прыгаем на истинную точку входа начального загрузчика. Специальное слово word говорит flat assembler не пытаться оптимизировать размер перехода и в любом случае положить адрес перехода в 2 байта, даже если можно в 1. Таким образом наш jmp должен гарантированно занять 3 байта.  Также в результате этого действия у нас первые 3 байта загрузчика содержат ненужные для дальнейшей работы данные, поэтому мы можем сэкономить 3 байта (512 это очень мало и любую возможность оптимизировать размер без потери функционала не следует упускать), разместив там какие-нибудь неинициализированные переменные. Пока она одна: номер загрузочного диска - байтовая переменная disk\_id.  Также в области данных я разместил две строки, которые будут использоваться в нашем загрузчике - приветственное сообщение с его названием и версией и предложение нажать любую клавишу для перезагрузки. Вы можете легко изменить выводимый текст, но помните, что у нас в распоряжении нет русских букв.  Первая из описанных функций предназначена для вывода текстовых сообщений на экран. Она принимает единственный параметр в паре регистров DS:SI. Строка должна оканчиваться нуль-символом (использование null-terminated строк сейчас является стандартом в подавляющем большинстве операционных систем и языков программирования, и это вполне оправданно - символ с кодом 0 не нужен для простых текстовых данных). Вывод осуществляется с помощью сервиса BIOS.  BIOS предоставляет достаточно много функций, все они вызываются с помощью программных прерываний, для осуществления которых служит ассемблерная инструкция int. Она принимает в качестве аргумента номер прерывания. За каждым прерыванием закреплён адрес функции обработки. Она принимает параметры в регистрах процессора, выполняет действие и возвращает управление нашему коду. В данном случае нас интересует сервис с кодом 0x10. Это прерывание служит для управления экраном. Номер нужной функции прерывания передаётся в регистре AH. Функция write\_str использует функцию с кодом 0x0E, которая просто выводит символ из AL на экран со сдвигом курсора (воспринимаются также различные управляющие коды вроде последовательности 13,10 - перевод строки и возврат каретки). Наш код сохраняет оба модифицируемых во время работы регистра - AX и SI, поэтому его можно вызывать из любых мест кода не беспокоясь о последствиях.  Вторая функция - error. Она не используется в текущем коде, не пригодится в будущем. Это один из примеров оптимизации размера. Как известно, команда call помещает в стек адрес возврата, то есть адрес, следующий за этой инструкцией. Но если возврат из функции не предполагается, то там могут находиться произвольные данные, к которым можно получить доступ вытолкнув адрес из стека. В этом случае мы экономим на команде запихивания в стек этого адреса, потому что этим занимается процессор. Функция error нужна для обработки критических ошибок загрузки (а других на этом этапе и не бывает :-) ), например ошибки чтения диска. Использовать эту функцию следует так:  call error  db "DISK READ ERROR!",13,10,0  В итоге будет выведено сначала сообщение об ошибке, а потом предложение нажать любую клавишу для перезагрузки. За последнее действие отвечает функция reboot. Она выводит сообщение "Press any key...", а затем ждёт нажатия на любую клавишу (этим занимается нулевая функция прерывания BIOS 0x16). После того, как нажата клавиша, происходит прыжок на точку входа в BIOS - FFFF:0000. Возможно, было бы корректнее вызывать прерывание 0x18, но на многих машинах оно лишь приводит к попытке загрузки со следующего устройства, я же хочу эффект аналогичный нажатию Reset.  Основной код располагается за меткой boot. Первым делом он обнуляет все сегментые регистры, чтобы правильно работала адресация данных. Затем он настраивает стек и разрешает прерывания. Стеком будет считаться область от данных BIOS (0000:0400) до нашего кода. Это более 30 килобайт, что вполне достаточно.  Далее с помощью write\_str выводится название загрузчика и наконец работа завершается переходом на reboot (нам больше нечего делать - мы пока ничего не умеем загружать).  **Компиляция и запуск**  Для компиляции достаточно просто выполнить команду:  fasm boot.asm boot.bin  В итоге должен получиться файл boot.bin размером 512 байт. Некоторые эмуляторы (например, Bochs) позволяют подключать неполные образы дискет. Просто отсутствующие сектора не будут загружаться.  Затем можно запускать всё это в Bochs и, если не было ошибок, вы увидите две строки.    Конфиг для Bochs:  megs: 64 boot: floppy floppya: 1\_44=bin/boot.bios.bin, status=inserted  Могу вас поздравить: вы написали свой первый начальный загрузчик!  **Заключение**  В этом выпуске мы рассмотрели основные теоретические аспекты начальной загрузки компьютера, а также написали простейший загрузчик. В следующем выпуске я расскажу про файловые системы и мы научим наш загрузчик действительно что-нибудь загружать.  Если у вас есть какие-либо вопросы ко мне насчёт содержания этого выпуска или последующих - вы всегда можете написать мне электронное сообщение на адрес [kiv.apple@gmail.com](mailto:%20kiv.apple@gmail.com).  До встречи! |

## Пишем свою операционную систему. Основы Assembler

|  |
| --- |
| Отзывы подписчиков после предыдущего выпуска показали, что я ошибался, насчёт уровня знания Assembler моей аудиторией. Поэтому я немного нарушаю свои планы и в этом выпуске расскажу об основах программирования на Assembler, а о материал про загрузчик будет в следующем выпуске.  Те, кто уже и так не плохо умеет программировать на Assembler могут пропустить этот выпуск. Я не претендую я полное изложение всех аспектов, какой-то материал может быть намеренно упрощён или даже искажён для простоты понимания. Так что знающим людям не следует писать мне гневные письма "у тебя в рассылке ошибка!" - всё так и запланировано. Информации приведённой ниже должно быть достаточно для понимания работы загрузчика из второго выпуска. По мере разработки я буду дополнять выпуски новой информацией. Мы не планируем всей ОС целиком на Assembler, а того что я скажу вполне хватит.  Также, я не планировал рассказывать про языки программирования, а сконцентирироваться на разработке ОС, поэтому, возможно, выпуск получился немного странным по формулировкам и фразам. Приношу свои извинения. Как впрочем и за возможные орфографические и пунктуационные ошибки - текст получился просто огромным...  Assembler - достаточно сложный язык для новичков, но в деле осеписательства без него совсем обойтись нельзя. Не пугайтесь, потом будет проще :-) **Память. Сегменты. Регистры**  Сейчас мы будем рассматривать реальный режим работы процессора. То есть самый старый для процессоров x86. Начинаем с него, потому что загрузчик после BIOS попадаёт именно в него, а все остальные переходы совершаются уже специальными командами. Когда придёт время переводить процессор в защищённый режим, я дополню сведения об Assembler.  Память представляет собой массив байт. Отсчёт ведётся от нуля. Доступ к памяти возможен на уровне байтов (8 бит), слов (16 бит), двойных слов (32 бита, доступно только на процессорах, поддерживающих защищённый режим). При этом доступ осуществляется быстрее, если переменная выровнена на свой размер. То есть, при обращении к слову лучше, если адрес кратен 2 байтам, к двойному слову - 4 байтам. В отличии от некоторых других архитектур, это лишь рекомендуемое, но не обязательное условие и к переменной любого размера можно обращаться, даже если её адрес не кратен ничему. Адреса принято записывать в 16-ричной системе счисления, потому что очень часто они выравнены на какую-нибудь степень двойки и выглядят более наглядно.  У процессора есть своя внутренняя память - регистры. Это небольшая, но очень быстрая память. Она предназначена для хранения состояния процессора (например, адрес выполняемой в данной момент инструкции, текущий режим процессора и т. д.) и для хранения программой промежуточных результатов вычислений. В отличии от обычной памяти, для обращения к регистрам используется не адрес, а имя. Да и обратиться к произвольной части регистра нельзя (например, нельзя взять 3-ий байт 32-битного регистра - только весь регистр целиком, а потом с помощью логических операций выделить часть).  У процессора 8086 (все его потомки унаследовали эти регистры, хотя добавили и свои, новые) все регистры делятся на регистры данных, сегментые регистры и указатель команд (IP - Instruction pointer), последний недоступен программно. Все регистры имеют размер 16 бит. Начнём с регистров данных - их 8 штук: AX, BX, CX, DX, SI, DI, SP, BP. Первые четыре регистра позволяют обращаться к своим 8-битным половинкам - для AX это AH (старшие 8 бит) и AL (младшие 8 бит), для BX - BH и BL, для CX - CH и CL, для DX - DH и DL. То есть, например, если мы поместим в AX число 0x55AA, то в AH будет 0x55, а в AL в 0xAA.  Регистры SI, DI ещё иногда называют "индексные", BX, SP и BP - "регистры-указатели".  4 регистра (кроме SP, но он тоже по сути используется для адресации памяти) обладают одной особенностью - они позволяют хранить не только данные, но и адреса. То есть, процессору можно сказать не только "прочитать слово из памяти по адресу 0x1234 и поместить в регистр AX", но и "прочитать слово из памяти по адресу из регистра BX и поместить в регистр AX". Во втором случае адрес не известен на этапе компиляции, а может вычисляться на ходу. SP - необычный регистр-указатель. Его нельзя использовать (в реальном режиме, в защищённом можно) для адресации в командах, но сам процессор использует его для поддержки аппаратного [стека](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA) - в SP хранится указатель на его вершину.  Как я уже сказал выше, все регистры имеют размер 16 бит, то есть с их помощью получится адресовать лишь 64 КБ (2 ^ 16 байт) памяти. Это слишком мало, поэтому были введены сегментые регистры, которые расширяют адресуемое пространство до 1 МБ (этих проблем нет в защищённом режиме, где ширина адреса 32 или 64 бита на самых новых системах). Теперь адрес вычисляется путём сдвига значения сегментного регистра на 4 бита и прибавления значения адреса (смещения). Это можно записать в виде формулы: адрес = сегмент \* 16 + смещение. В итоге получается ширина адреса 20 бит, что и даёт 1 МБ адресуемой памяти.  У 8086 есть 4 сегментные регистра (в последствии были добавлены ещё 2, но они нам не нужны) - CS, DS, ES, SS.  CS в паре с IP используется для выбора текущей инструкции для исполнения. Так же как и IP, он не доступен для программной записи (изменение значения этих регистров может происходить только в ходе естественного выполнения программы, либо команды перехода на другой адрес), хотя в отличии от него доступен для чтения. DS - регистр сегмента данных. Большинство команд подразумевают его использование для получения сегментной части адреса, если не указано иное. ES - альтернативный сегмент данных. Два сегментных регистра данных упрощают адресацию, позволяя реже изменять значения сегментов. SS в паре с SP служит для адресации вершины стека.  Ещё у процессора 8086 есть регистр флагов - FLAGS. Он недоступен для прямого чтения и записи, но может влиять на исполнение некоторых команд, также как и некоторые команды могут изменять его значение. **Несколько простых команд** Команды ассемблера начинаются с кодового названия команды длиной от 2 до 5 букв. Затем может следовать от 0 до 3 (3 встречается очень редко) аргументов. Количество аргументов и допустимые значения (константа, адрес, регистр) зависят от команды.  Пожалуй, самая часто используемая команда - MOV (MOVe) - пересылки значения из одного места в другое. Можно провести аналогию с командой присваивания из языков высокого уровня. Пример использования:  mov ax, 0x1234 ; AX :=  0x1234  mov dx, ax ; DX := AX  mov ax, [0x1234] ; Поместить в AX слово из памяти по адресу DS:0x1234 (DS используется для данных по умолчанию)  mov [es:0x1234], dl ; Поместить младший байт DL в память по адресу ES:0x1234  mov word[0x1234], 1 ; Поместить единицу в слово памяти по адресу DS:0x1234  ; (явное указание размера byte или word нужно потому, что без регистра компилятор не может угадать какой размер мы имеем ввиду)  Как можно заметить с помощью точки с запятой в Assembler отделяются комментарии. Прямая работа с сегментными регистрами (то есть, не для адресации в других командах) ограничена - с ними могут работать только команды работы с аппаратным стеком и mov. Причём последний в качестве второго аргумента может иметь только регистр (нельзя сохранять или восстанавливать сегментные регистр прямо из памяти или же присваивать ему непосредственное значение).  Перечислю ещё несколько простых команд:  add ax, bx ; AX := AX + BX  sub dx, cx ; DX := DX - CX  inc si ; SI := SI + 1  dec di ; DI := DI - 1  or ax, bx ; AX := AX or BX  and ax, bx ; AX := AX and BX  xor ax, bx ; AX := AX xor BX  xor cx, cx ; CX := 0 - одна из простейших оптимизаций. В отличии от mov cx, 0 эта инструкция занимает лишь 1 байт.  ; Очевидно, что результат исключающего ИЛИ числа с самим собой равен нулю.  not dx ; DX := not DX  В качестве операндов каждой из этих инструкций могут выступать 16- и 8-битные регистры, непосредственные значения (например, add ax, 10), адреса памяти (например, add [0x1234], 10 или add ax, [es:0x4321]) с соблюдением одного условия - в команде может быть только одно явное обращение к памяти. То есть нельзя, например, складывать значение из памяти со значением из памяти и т. д. Это ограничение распространяется на подавляющее большинство команд процессора x86 (включая и вышеописанный MOV), поэтому я не буду о нём больше специально напоминать при описании команд. Если действие с двумя аргументами из памяти всё же необходимо, один из них придётся временно скопировать в какой-нибудь регистр.  Адрес памяти не обязательно должен быть жёстко задан, его можно вычислять во время работы программы. Допустимо сложение непосредственного значения, одного регистра-указателя и одного индексного регистра. То есть в самой длинной форме адрес может выглядеть как-то так:  mov byte[ds:0x1234  + bx + si], 0  mov word[es:0x1234 + bx + di], 1  mov [ss:0x1234 + bx + bp], ax  Любая из частей адреса может отсутствовать, то есть для адресации можно использовать лишь какой-нибудь один из регистров, а то и вообще ничего к его значению не прибавлять. **Переходы. Ветвления. Циклы** Для безусловного перехода есть специальная команда - jmp (JuMP). Она принимает один аргумент - адрес, куда надо перейти. Адрес может быть без указания сегмента - в таком случае CS не меняется, так и с указанием, тогда этот переход называется "дальним" и меняется вся пара CS:IP. В качестве адреса перехода может использоваться непосредственное значение, регистр-указатель, индексный регистр или переменная в памяти. В случае использования значения из регистра, переход возможен только ближний (в пределах сегмента), потому что размер регистра лишь 16 бит.  jmp 0x1234 ; Переход к адресу CS:0x1234  jmp bx ; Переход к адресу CS:BX  jmp 0:0x7C00 ; CS := 0, IP := 0x7C00  jmp word[es:0x7E00 + bx + si] ; Адрес куда следует прыгать хранится в 16-битной переменной  Переходы бывают не только безусловные, но и условные. То есть, выполняющиеся лишь в определённом состоянии процессора. Это позволяет реализовывать ветвления и циклы. Команды условного перехода анализируют значение регистра FLAGS. Обычно команды перехода используются после инструкции CMP (CoMPare - сравнить), которая сравнивает два числа. Эта инструкция принимает два аргумента и производит вычитание одного из другого. После этого некоторые биты регистра флагов (флаг нулевого результата, флаг отрицательного результата, флаг переноса, флаг переполнения) изменяют своё значение. Анализируя их можно сделать вывод об отношении чисел, которые мы сравнили.  jz, je - Установлен флаг нулевого результата, в результате арифметической операции получился ноль. В случае cmp это значит, что числа равны. jb - При беззнаковом сравнении означает, что второй аргумент больше первого. ja - При беззнаковом сравнении означает, что второе аргумент меньше первого. jl - При знаковом сравнении означает, что второй аргумент больше первого. jg - При знаковом сравнении означает, что второй аргумент меньше первого. jc - Установлен флаг переноса. jo - Установлен флаг переполнения. js - Установлен флаг отрицательного результата.  Действие операции можно изменить на противоположное добавив N после J - jne (переход, если не равно), jnz, jnb, ja, jnl, jng, jnc, jno. Также можно комбинировать 2 условия - jae (переход, если больше или равно), jbe, jle и т. д.  В качестве аргумента команды условного перехода можно использовать только непосредственное значение.  Изменяет значение регистров флагов не только cmp, но и практически все арифметические и логические операции (add, sub, inc, dec, and, or, xor и т. д.; а вот mov, кстати, не изменяет, как и операции работы со стеком). Это следует учитывать как с положительной (можно проанализировать результат выполнения операции по флагам), так и с отрицательной (между cmp и условным переходом нельзя выполнять многие операции, потому что они затрут результат сравнения).  Существует команда test, которая выполняет проверку битов. По сути это and, только без сохранения результатов. Например, test ax, 0xF проверит установлен ли в регистре AX хоть один из 4 младших битов. Его же используют для очередной оптимизации - вместо cmp ax, 0, можно написать test ax, ax. Если AX = 0, то флаг нулевого результата будет установлен и сработает переход jz/je, а такая команда занимает лишь 1 байт. **Метки. Описание данных** Разумеется, вычислять все адреса вручную было бы слишком сложно, поэтому это вполне естественно переложить на компилятор.  Для создания метки надо записать сначала её идентификатор, а потом символ двоеточия. После этого может сразу же следовать команда, а может и не следовать (для читабельности).  Метки можно использовать в любом месте, где ожидается адрес в памяти. Ассемблер вычисляет все адреса и заменяет имена меток на непосредственные значения на этапе компиляции - процессор ничего о метках не знает.  mov di, 0x100  mov ax, 0x55AA  fill\_memory:  mov [di], ax  add di, 2  cmp di, 0x200  jb fill\_memory  Пример выше заполняет память от адреса DS:0x100 до адреса DS:0x200 словами 0x55AA. Таким образом это простейший цикл. С помощью меток и условных переходов можно проверять условия любой сложности.  Для описания данных служат псевдоинструкции db, dw, dd и dq (define byte, define word, define dword, define qword). Эти команды не обозначают никаких инструкций процессора, а лишь говорят компилятору поместить в этом месте без какой-либо обработки указанные через запятую значения. Инструкция может претворятся меткой, чтобы задать имя для переменной. При этом двоеточие не обязательно и даже не желательно (если его не использовать, то flat assembler сможет запомнить размер переменной и при использовании в других командах не придётся его явно указывать). Количество аргументов может быть от 1 до бесконечности (это не обычная инструкция). Все они будут просто скопированы в выходной файл. В случае объявления байтовой переменной можно указать в качестве значения строку в кавычках, вся она будет записана в выходной файл. Примеры:  number dw 0xB800  string db "Hello world!",13,10,0  byte\_array db 1,2,3,4,5,6  word\_array dw 1,2,3,4  ...  mov ax, [number]  mov [number], 0 ; Размер переменной известен. Указывать word не обязательно.  mov cl, byte[number] ; Но если мы захотим выполнить приведение типов, то можно указать другой размер.  mov si, string ; А тут мы записываем в SI не значение переменной, а её адрес  mov al, [byte\_array + bx] ; Получаем из массива элемент с индексом BX  Есть специальные псевдо-инструкции, служащие лишь для резервирования места, но не описания самих данных. В некоторых ситуациях они позволяют сьэкономить место в исполняемом файле. По умолчанию значения этих переменных равных нулю. Это инструкции rb (reserve byte), rw (reserve word), rd (reserve double word) и rq (reserve quad word). В качестве аргумента они принимают количество элементов, место для которых надо зарезервировать. Например, rw 100 создаст пустой массив из 100 слов. Для одиночных переменных такого же поведения можно добиться, если вместо значения указать знак вопроса.  Все метки, которые мы описывали выше были глобальные, а бывают ещё и локальные. Их имя начинается с точки. В этом случае к ним можно обращаться напрямую во всём блоке до описания следующей глобальной метки, а вне блока надо писать ИмяГлобальнойМетки.ИмяЛокальнойМетки. Например:  global\_label:  .local\_label db 1  .addr dw .local\_label  next\_global\_label:  dw global\_label.local\_label  Все переменные видимы как после их описания, так и до.  Существует специальный вид меток, которые являются глобальными, но никак не влияют на локальные. Имена этих меток начинаются с двух точек.  Чтобы вычислить адреса меток необходимо знать, где в памяти будет располагаться код программы. Часто программа располагается в памяти далеко не с нулевого адреса. Чтобы задать базовый адрес программы служит псевдоинструкция org.  label1: ; label1 = 0  org 0x1000  label2: ; label2 = 0x1000  dw 0  label3: ; label3 = 0x1002  Менять базовый адрес можно несколько раз, если того требует особенности выполнения приложения, но важно помнить - инструкция org меняет лишь базовый адрес для вычисления адресов меток, при этом данные в файле никуда не перемещаются. Хотя в памяти между label1 и label2 должен быть пробел в 4 килобайта, в файле эти метки будут расположены вплотную. Правильно разместить код в памяти - задача загрузчика приложения (например, BIOS загружает наш код загрузчика по адресу 0:0x7C00 и мы можем на это рассчитывать).  Кстати, команды условных и коротких безусловных переходов являются относительными (типа "прыгнуть на 10 байт вперёд"), поэтому переходы в рамках одного сегмента будут работать вне зависимости от базового адреса (главное, чтобы метки располагались в пределах одного org), по которому он действительно загружен.  Также, flat assembler объявляет пару псевдометок (их значение зависит от того в каком месте они используются) - $ и $$. Первая всегда равна текущему адресу (то есть jmp $ - бесконечный цикл, каждый раз прыжок будет совершаться на эту же команду), вторая - последнему заданному с помощью org базовому. **Стек** Процессор предоставляет доступ к аппаратному стеку. Для работы с ним служат две команды - push и pop. Обе они имеют один аргумент, который может быть чем-угодно (константа, регистр, сегментные регистр, адрес памяти). Первая команда уменьшает значение SP на 2 и записывает аргумент по адресу SS:SP. Вторая команда наоборот, копирует значение из SS:SP в свой аргумент и увеличивает значение SP на 2. Помимо прочего эти команды позволяют менять значения сегментных регистров без использования основных.  push ax  mov ax, 10  pop ax  ...  push ds  pop es ; ES := DS  ...  push 0xB800  pop es ; ES := 0xB800  ...  push [variable]  pop [other\_variable]  Как можно заметить, указатель стека всегда выровнен на 2 байта. Поэтому нет смысла только ради экономии памяти помещать туда байт место слова. Это имеет смысл, только если приёмник имеет однобайтовый размер. **Подпрограммы** Процессор на аппаратном уровне поддерживает использование подпрограмм. Для этого есть две команды - call и ret. Первая помещает в стек адрес следующей инструкции и выполняет безусловный переход по адресу из аргумента. Вторая извлекает из стека адрес возврата и выполняет переход по нему. Таким образом откуда бы не был произведён вызов подпрограммы выполнение вернётся в вызывающий код.  procedure:  ...  ret  ...  call procedure  call procedure  Если аргументом call является адрес с сегментной частью, то в стек запихивается не 1 слово, а 2 (не только IP, но и CS). Это называется "дальним вызовом". Подпрограмма, которую так вызывают должна использовать не ret, а retf, который умеет возвращаться по такому длинному адресу возврата. Таким образом "дальние" и "ближние" подпрограммы несовместимы между собой и программист должен помнить на какой тип вызова рассчитана каждая его процедура или функция.  Ближние вызовы как и переходы могут быть относительными и корректно работать даже если базовый адрес кода в памяти неизвестен. **Строковые операции**  Процессор x86 обладает достаточно широким набором команд. На мой взгляд стоит рассмотреть одно из их подмножества - "строковые". Эти команды предназначены для работы с массивами данных.  Опишу несколько команд:  lods - Загрузка байта/слова из памяти по адресу DS:SI в AL/AX. Увеличение SI на размер элемента stos - Сохранение байта/слова из AL/AX в память по адресу ES:DI. Увеличение DI на размер элемента. movs - Копирование байта/слова из DS:SI в ES:DI. Увеличение SI и DI на размер элемента. cmps - Сравнение байта/слова из DS:SI и ES:DI. Увеличение SI и DI на размер элемента. Регистр флагов изменяется аналогично команде cmp. scas - Сравнение байта/слова из ES:DI с AL/AX. Увеличение DI на размер элемента. Регистр флагов изменяется аналогично команде cmp.  Каждая строковая команда имеет постфикс размера - b (byte), w (word). Например, lodsb, movsw и т. п.  Совместно с этими командами часто применяют префиксы повтора операции - rep, repe, repne. Эти префиксы позволяют повторить следующую за ними строковую операцию CX раз (каждую итерацию CX будет уменьшаться на единицу, пока не достигнет нуля. если CX равен нулю изначально, то команда не выполнится ни разу). repe помимо основного условия может прервать выполнение, если после выполнения флаг нулевого результата не будет установлен ("выполнять пока равно"), а repne в противоположной ситуации (выполнять пока не равно).  Например, с помощью rep movsw можно быстро скопировать CX слов из DS:SI в ES:DI. С помощью других префиксов можно также выполнять сравнение строк (repe cmpsb), поиск символа в строке (repne scasb). Эти операции будут выполняться значительно быстрее, чем простой цикл с помощью mov, dec, cmp и jmp, да и выглядят очень наглядно. **Разбор фрагмента загрузчика и заключение** Рассмотрим, подпрограмму нашего загрузчика из предыдущего выпуска:  write\_str:  push ax si  mov ah, 0x0E  @:  lodsb  test al, al  jz @f  int 0x10  jmp @b  @:  pop si ax  ret  Перед разбором, стоит обратить внимание на @@ (в предыдущем выпуске Subscribe покорёжил код и заменил две "собаки" на одну) - это так называемая анонимная метка. Их можно описывать сколько угодно в одной программе (если описать две обычных метки с одинаковом именем, то будет ошибка). @f ссылается на ближайшую анонимную метку дальше по коду, @b - на ближайшую предыдущую. Глобальные метки никак не влияют на видимость анонимных, а анонимные не ограничивают видимость локальных, как глобальные.  Первая строка объявляет глобальную метку для подпрограммы, которая позволяет обратиться к ней из любой точки программы. Вторая и предпоследняя строка обеспечивает сохранение и восстановление регистров AX и SI, которые будут затронуты кодом процедуры. flat assembler позволяет записать сколько угодно аргументов для push и pop через пробел - в итоге будет сгенерировано по одному push/pop на каждый аргумент. Это позволяет сэкономить строчки программы. Заметьте, что при восстановлении регистров из стека я указываю их в обратном порядке, потому что действует правило стека "первым пришёл - последним ушёл".  В третьей строке я  записываю значение 0x0E в старший байт AX. Далее между двумя анонимными метками выполняется цикл по всем символам строки. Первая команда цикла подгружает очередной символ из строки. Вторая проверяет не равен ли он нулю (ноль у нас - признак конца строки). Если равен, то третья выведет нас из цикла. И наконец команда безусловного перехода повторяет цикл вновь.  Последняя команда выполняет возврат из процедуры в вызывающую программу. Поскольку мы используем ret, а не retf нашу подпрограмму можно вызывать только из того же сегмента кода (но нам большего и не нужно в загрузчике, а ближние вызовы занимают меньше места).  Пусть разбор остальных частей загрузчика будет вашим домашним заданием. В случае возникновения трудностей задавайте вопросы ;-)  На этом этот выпуск рассылки можно считать завершённым. До встречи! |

## Пишем свою операционную систему. Простейший загрузчик

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Приветствую всех своих подписчиков!  Как вы уже знаете, BIOS загружает первые 512 байт диска и передаёт коду, который там расположен, управление. Разумеется, этого слишком мало, чтобы уместить туда код операционной системы, поэтому основной задачей этого загрузчика является подгрузка остальных частей операционной системы в память. У многих операционных систем даже сам полноценный загрузчик не умещается в один сектор и тогда используется многоэтапная загрузка - boot sector загружает вторичный загрузчик, который уже загружает всё остальное. Я постараюсь писать более-менее универсальный загрузчик, когда же он перестанет помещаться в 1 сектор буду догружать второй, а уж в целый килобайт, думаю, уместимся... **Функции BIOS для дискового ввода-вывода** Драйвер для какого-либо устройства хранения данных слишком сложная программа, чтобы уместиться в начальный загрузчик, поэтому BIOS предоставляет набор сервисов для работы с дисками. Нас будет интересовать только чтение, ибо начальному загрузчику нет нужды изменять данные. За все дисковые операции отвечает прерывание под номером 0x13. Как обычно, код вызываемой функции указывается в регистре AH. Большинство функций этого прерывания принимают в DL идентификатор диска. Узнать его не составит проблем, ведь предыдущая версия нашего загрузчика сохраняет номер, переданный BIOS, в переменную disk\_id. Пока нас интересуют три функции дискового сервиса:   |  |  | | --- | --- | | Номер функции (значение для AH) | Описание и параметры | | 0x00 | Сброс дисковой подсистемы. Некоторые носители, такие как дискеты, могут даже в случае исправной работы выдавать ошибки чтения. В этом случае следует пытаться читать несколько раз, прежде чем считать это настоящей ошибкой. Между попытками чтения следует сбрасывать статус с помощью этой функции. | | 0x02 | Чтение сектора. AL содержит количество секторов для чтения,  ES:BX - адрес буфера куда следует поместить прочитанные данные. CH:DH:CL - адрес сектора в формате ЦИЛИНДР:ГОЛОВКА:СЕКТОР. В случае ошибки чтения после  вызова этого прерывания будет установлен флаг переноса, который можно  проверить командами условного перехода JC и JNC. | | 0x08 | Определение параметров диска. Рекомендуется обнулить ES:DI, для обхода  багов некоторых BIOS. После вызова DL содержит количество дисков в  системе, DH - максимальный номер головки, CX - максимальный номер  цилиндра и сектора (данные о количестве секторов находятся в 6 младших  битах), BL - тип устройства, ES:DI - указатель на таблицу параметров  устройства. В случае ошибки, также устанавливается флаг переноса. |   Ввиду исторических обстоятельств, сервисы BIOS требуют указания номера сектора в такой неудобной форме (вдобавок адресация ограничена 8 ГБ), нам остаётся только преобразовывать нормальный, линейный номер (называется LBA) сектора в адресацию CHS, используя данные от последней функции. **Реализация подпрограммы чтения сектора** Начнём с описания дополнительных переменных:  ; Данные начального загрузчика  align 4  label sector\_per\_track word at $$ ; Количество секторов на одной дорожке  label head\_count byte at $$ + 2 ; Количество головок у дисковода  label disk\_id byte at $$ + 3  ...  Прежде, чем мы сможем что-либо загружать следует заполнить эти переменные параметрами, полученными от BIOS.  ; Запомним номер загрузочного диска  mov [disk\_id], dl  ; Определим параметры загрузочного диска  mov ah, 0x08 ; Номер функции  xor di, di ; ES и так равен нулю  push es ; Указатель на блок параметров BIOS нам не нужен, а вот нулевой ES - нужен  int 0x13  pop es  jc load\_sector.error ; Обработка дисковых ошибок находится в функции, которую мы напишем позднее  inc dh ; Отсчёт номера головки начинается от нуля - чтобы получить количество необходимо прибавить единицу  mov [head\_count], dh ; Запоминаем количество головок  and cx, 111111b ; Выделяем 6 младших бит CX  mov [sector\_per\_track], cx ; Запоминаем количество секторов на дорожке  Теперь у нас достаточно данных для преобразования LBA to CHS. Напишем функцию загрузки одного сектора. Для передачи номера сектора используем пару регистров DX:AX (DX - старшая часть, AX - младшая часть), потому что адрес может быть больше 16 бит. Прочитать сектор будем пытаться 3 раза, если все три раза неудачны, скорее всего, имеет место аппаратная проблема - выводим сообщение об ошибке и ожидаем перезагрузки.  ; Загрузка  сектора DX:AX в буфер ES:DI  load\_sector:  push ax bx cx dx si ; Сохраняем все используемые регистры  div [sector\_per\_track] ; Делим DX:AX на количество секторов на дорожке (частное от деления будет записано в AX, остаток в DX)  mov cl, dl ; Остатком является номер сектора - запоминаем в CL  inc cl ; Сектора в отличии от всего остального отсчитываются от единицы  div [head\_count] ; Делим частное на количество головок у дисковода (в этот раз мы делим на byte, а не word)  mov dh, ah ; Остатком (который в AH) является номер головки - сохраняем в DH  mov ch, al ; Частным (которое в AL) является номер дорожки - сохраняем в CH  mov dl, [disk\_id] ; Загружаем в DL сохранённый номер диска  mov bx, di ; Помещаем в BX смещение буфера. Его сегментная часть и так в ES  mov al, 1 ; Помещаем в AL количество секторов для чтения - 1.  mov si, 3 ; Количество попыток чтения  @@: ; Цикл чтения  mov ah, 2 ; Номер функции чтения  int 0x13 ; Пробуем прочитать  jnc @f ; Если всё хорошо, выходим из подпрограммы  xor ah, ah ; Номер функции сброса  int 0x13 ; Сбрасываем дисковую подсистему  dec si ; Уменьшаем счётчик попыток  jnz @b ; Если попытки ещё есть, переходим к началу цикла  .error: ; Попыток больше не осталось - ошибка чтения  call error ; Вызываем аварийное завершение загрузчика  db "DISK ERROR",13,10,0 ; Сообщение об ошибке  @@:  pop si dx cx bx ax ; Восстанавливаем регистры  ret ; Выходим из подпрограммы  Вот мы и написали подпрограмму для удобного чтения секторов диска. Эта процедура применима как к дискетам, так и к жёстким дискам. Возможно, она может прочитать и CD (только будет прочитано соответственно 2 КБ вместо 512 байт), но в этом не уверен, потому что не проверял. Мы не используем возможность чтения нескольких секторов подряд, потому что это приведёт к излишнему усложнению кода начального загрузчика (последовательное чтение возможно лишь в пределах одной дорожки), в котором размер гораздо важнее производительности (грузить в любом случае нам надо не много, разница заметна не будет). **Расширенный дисковый сервис BIOS** Код выше обладает одним значительным недостатком - он умеет загружать только сектора в пределах первых 8 ГБ диска. Для дискет это ограничение не актуально, но жёсткие диски, да и многие флешки давно перевалили за этот порог. Если мы будем работать с файловой системой, то нам необходим доступ ко всему объёму диска. Спустя какое-то время производители BIOS дополнили набор функций прерывания 0x13 так называемым "расширенным дисковым сервисом". Он позволяет указывать при чтении 64-битный линейный адрес. Однако полностью перейти на его использование нельзя - он не поддерживает работу с дискетами, поэтому мы реализуем оба способа чтения, а загрузчик будет сам выбирать подходящий.  Для начала дополним код определения параметров загрузочного диска. Чтобы определить поддержку сервиса существует функция 0x41 прерывания 0x13. Если она устанавливает флаг переноса, то это означает отсутствие поддержки для указанного диска.  ; Определим параметры загрузочного диска  mov ah, 0x41 ; Номер функции  mov bx, 0x55AA ; Сигнатура для вызова  int 0x13 ; Вызов дискового сервиса  jc @f ; Если сервис недоступен, то переходим к старому определению параметров  mov byte[sector\_per\_track], 0xFF ; Помещаем в sector\_per\_track невозможное значение  jmp .disk\_detected ; Пропускаем старое определение параметров  @@:  mov ah, 0x08  xor di, di  push es  int 0x13  pop es  jc load\_sector.error  inc dh  mov [head\_count], dh  and cx, 111111b  mov [sector\_per\_track], cx  .disk\_detected:  Новый код в случае поддержки расширенного дискового сервиса присваивает переменной sector\_per\_track значение 0xFF. Такого значения при обычной работе там быть не может - это и будет признаком для load\_sector, что надо использовать другой метод для чтения.  За чтение секторов отвечает функция 0x42 прерывания 0x13. Помимо номера диска в DL эта функция принимает в DS:SI указатель на "пакет дискового адреса", который имеет следующую структуру:   |  |  |  | | --- | --- | --- | | Смещение | Размер | Описание | | 00h | byte | Размер структуры (должен быть не меньше 16) | | 01h | byte | Не используется и должно быть равно нулю | | 02h | word | Количество секторов для чтения | | 04h | dword | Адрес буфера для чтения. Указывается в формате СЕГМЕНТ:СМЕЩЕНИЕ (смещение записывается перед сегментом) | | 08h | qword | 64-битный номер сектора |   Модифицируем код load\_sector для поддержки расширенного дискового сервиса BIOS:  ; Загрузка сектора DX:AX в буфер ES:DI  load\_sector:  cmp byte[sector\_per\_track], 0xFF ; Проверяем признак использования расширенного чтения  je .use\_EDD ; Переходим в другую часть подпрограммы  push ax bx cx dx si  ...  pop si dx cx bx ax  ret  .use\_EDD:  push ax dx si ; Сохранение используемых регистров  mov byte[0x600], 0x10 ; Разместим структуру после области данных BIOS - по адресу 0000:0600  mov byte[0x601], 0  mov word[0x602], 1 ; Читаем всё так же 1 сектор  mov [0x604], di ; Смещение  push es  pop word[0x606] ; Сегмент  mov [0x608], ax ; Младшая часть номера сектора  mov [0x60A], dx ; Старшая часть номера сектора  mov word[0x60C], 0 ; Верхние 32 бита  mov word[0x60E], 0 ; мы не используем  mov ah, 0x42 ; Код функции  mov dl, [disk\_id] ; Номер диска  mov si, 0x600 ; Начальный адрес структуры  int 0x13 ; Обращение к BIOS  jc .error ; Для жёстких дисков и флешек нет необходимости производить многократные попытки чтения  pop si dx ax ; Восстанавливаем регистры  ret ; Возврат из подпрограммы   Итак, теперь у нас есть универсальная подпрограмма для чтения секторов с любых носителей с возможностью адресации до 2 ТБ данных. Её мы сможем использовать, когда заходим загружать остальные части системы. Например, вот так можно загрузить дополнительные 512 байт загрузчика, если они следуют в следующем секторе.  xor dx, dx  mov ax, 1  mov di, 0x7E00  call load\_sector **Полный код загрузчика и заключение** ; Начальный загрузчик ядра для архитектуры x86  format Binary as "bin"  org 0x7C00  jmp boot  ; Данные начального загрузчика  align 4  label sector\_per\_track word at $$  label head\_count byte at $$ + 2  label disk\_id byte at $$ + 3  boot\_msg db "MyOS boot loader. Version 0.04",13,10,0  reboot\_msg db "Press any key...",13,10,0  ; Вывод строки DS:SI на экран  write\_str:  push ax si  mov ah, 0x0E  @@:  lodsb  test al, al  jz @f  int 0x10  jmp @b  @@:  pop si ax  ret  ; Критическая ошибка  error:  pop si  call write\_str  ; Перезагрузка  reboot:  mov si, reboot\_msg  call write\_str  xor ah, ah  int 0x16  jmp 0xFFFF:0  ; Загрузка сектора DX:AX в буфер ES:DI  load\_sector:  cmp byte[sector\_per\_track], 0xFF  je .use\_EDD  push ax bx cx dx si  div [sector\_per\_track]  mov cl, dl  inc cl  div [head\_count]  mov dh, ah  mov ch, al  mov dl, [disk\_id]  mov bx, di  mov al, 1  mov si, 3  @@:  mov ah, 2  int 0x13  jnc @f  xor ah, ah  int 0x13  dec si  jnz @b  .error:  call error  db "DISK ERROR",13,10,0  @@:  pop si dx cx bx ax  ret  .use\_EDD:  push ax dx si  mov byte[0x600], 0x10  mov byte[0x601], 0  mov word[0x602], 1  mov [0x604], di  push es  pop word[0x606]  mov [0x608], ax  mov [0x60A], dx  mov word[0x60C], 0  mov word[0x60E], 0  mov ah, 0x42  mov dl, [disk\_id]  mov si, 0x600  int 0x13  jc .error  pop si dx ax  ret  ; Точка входа в начальный загрузчик  boot:  ; Настроим сегментные регистры  jmp 0:@f  @@:  mov ax, cs  mov ds, ax  mov es, ax  ; Настроим стек  mov ss, ax  mov sp, $$  ; Разрешим прерывания  sti  ; Выводим приветственное сообщение  mov si, boot\_msg  call write\_str  ; Запомним номер загрузочного диска  mov [disk\_id], dl  ; Определим параметры загрузочного диска  mov ah, 0x41  mov bx, 0x55AA  int 0x13  jc @f  mov byte[sector\_per\_track], 0xFF  jmp .disk\_detected  @@:  mov ah, 0x08  xor di, di  push es  int 0x13  pop es  jc load\_sector.error  inc dh  mov [head\_count], dh  and cx, 111111b  mov [sector\_per\_track], cx  .disk\_detected:  ; Завершение  jmp reboot  ; Пустое пространство и сигнатура  rb 510 - ($ - $$)  db 0x55,0xAA  В этом выпуске вы разработали процедуру, которая позволяет нам загружать выбранные сектора с диска в память компьютера. В дальнейшем этот код можно использовать для загрузки дополнительных секторов загрузчика и для доступа к файловой системе диска. Уже сейчас вы можете попробовать дописать загрузку второго сектора и что-нибудь сделать.  В следующем выпуске нам предстоит разобраться с файловыми системами. Все вопросы и предложения, как обычно, можно направлять на мой [личный адрес](mailto:%20kiv.apple@gmail.com). До встречи! |

Пишем свою операционную систему. Файловая система

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Доброго времени суток!  Наш загрузчик готов к тому, чтобы научить его загружать что-нибудь полезное. Осталось определиться с файловой системой.  Можно было бы воспользоваться готовой ФС, однако либо они достаточно сложные (NTFS, FAT при использовании имён длиннее 12 символов), либо обладают целым набором ограничений (FAT). Поскольку немаловажным для нашей ОС критерием является наглядность разработки, мы реализуем загрузку с нашей собственной файловой системы. С одной стороны она имеет очень простую структуру, с другой с ней можно достаточно быстро работать и она не имеет ограничений вроде FAT. Её возможности можно будет легко расширить, ну а уже потом, в рабочую систему, можно добавить драйверы для работы с другими ФС, чтобы был возможен обмен данными с другими операционными системами.  Моя файловая система называется ListFS, потому что построена на концепции двунаправленных связанных списков. Рассмотрим её структуру.  Начинается файловая система с заголовка, который располагается в загрузочном секторе, начиная с 4-ого байта (байты до этого хранят команду перехода к настоящей точке входа в загрузчик).   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Название | | | | Размер | Описание | | fs\_magic | 4 байта | | Сигнатура файловой системы. Должна быть равна 0x84837376. | | | | fs\_version | 4 байта | | Версия файловой системы - для возможности будущих изменений. Сейчас, равна единице. | | | | fs\_flags | 4 байта | | Атрибуты файловой системы. | | | | fs\_base | 8 байт | | Номер начального сектора файловой системы. Служит для упрощения работы загрузчика, когда на диске несколько разделов (Сервисы BIOS ничего про разделы не знают). Все остальные адреса рассчитываются относительно этого значения. | | | | fs\_size | 8 байт | | Размер файловой системы в секторах | | | | fs\_map\_base | 8 байт | | Базовый адрес битовой карты свободных секторов. | | | | fs\_map\_size | 8 байт | | Размер битовой карты свободных секторов. Равно fs\_size / 8 / fs\_block\_size (при делении всегда округлять в большую сторону). | | | | fs\_first\_file | | 8 байт | Номер сектора, содержащего заголовок первого файла в корневом каталоге. Если диск пуст -1. | | | | fs\_uid | | 8 байт | Уникальный идентификатор файловой системы | | | | fs\_block\_size | | 4 байта | Размер сектора. Для поддержки любых блочных устройств. У нас равно 512. | | |   Таким образом заголовок файловой системы занимает целых 64 байта, зато предусматривает самые разнообразные требования к ФС и оставляет возможность расширения. Битовая карта представляет собой непрерывный массив секторов, доступ осуществляется на уровне отдельных битов (ОС, которая планирует производить операции записи на ListFS, скорее всего просто загрузит всю карту в оперативную память для удобства доступа). Каждый бит соответствует одному сектору ФС. Если бит установлен, значит сектор занят, если сброшен, то свободен. Карта необходима только для изменения данных - для поиска свободного сектора для новых и для освобождения секторов от удалённых файлов.  Все каталоги файлов имеют структуру списков. Каждый заголовок файла имеет указатель на следующий и предыдущие, таким образом зная первый элемент списка можно перебрать все файлы каталога. Каждый заголовок файла занимает ровно один сектор:   |  |  |  |  | | --- | --- | --- | --- | | Название | Размер | | Описание | | f\_name | 256 байт | | Имя файла в кодировке UTF-8. Пока мы не делаем локализованный интерфейс пользователя нас мало волнуют особенности Юникода, но без него будет плохо в будущем. | | f\_next | 8 байт | | Номер сектора со следующим заголовком файла. Если файл последний в каталоге, то -1. | | f\_prev | 8 байт | | Номер сектора с предыдущим заголовком файла. Если файл первый в каталоге, то -2 | | f\_parent | | 8 байт | Указатель на родительский каталог. -1, если файл находится в корневом каталоге | | f\_flags | 8 байт | | Атрибуты файла. Пока определён только один - 0-ой бит - признак того, что это не файл, а каталог. | | f\_data | 8 байт | | Указатель на данные файлы. Если это каталог, то это указатель на первый файл каталога или -1 для пустого каталога. В обратном случае это указатель на сектор со списком секторов файла (у пустого файла так же будет -1). Такой сектор хранит sector\_size / 8 номеров секторов. Список оканчивается номером -1. Последний элемент списка является указателем на следующий список (если он не равен -1 и до него не встретилось -1). Таким образом можно хранить файлы любой длины. | | f\_size | 8 байт | | Размер файла в байтах | | f\_create\_time | 8 байт | | Дата создания | | f\_modify\_time | 8 байт | | Дата последнего изменения | | f\_access\_time | 8 байт | | Дата последнего обращения |   Остальные байты сектора с заголовком зарезервированы для будущего использования. Например, можно добавить поля для хранения прав доступа POSIX.  Вот и вся структура ListFS. Возможно, она понравится и другим начинающим авторам операционных систем за свою простоту и гибкость. Только давайте договоримся об одном соглашении - любое расширение ListFS должно лишь добавлять новые поля в структуру заголовка ФС или файла, но не модифицировать старые. То есть должна сохраняться обратная совместимость и любая ОС, поддерживающая ListFS, сможет работать с вашей модификацией, просто без отсутствия некоторых плюшек.  Для удобного создания образов я написал кросс-платформенную утилиту на Си. Её исходники будут представлены в конце выпуска. Вы можете собрать её как в Windows (с использованием компилятора MinGW), так и в Linux (с использованием GCC). При запуске без параметров эта утилита выводит полную справку о своих аргументах.  А теперь, научим наш начальный загрузчик загружать файлы с нашей файловой системы.  Для начала введём новые переменные:  org 0x7C00  jmp boot  ; Заголовок ListFS  align 4  fs\_magic dd ?  fs\_version dd ?  fs\_flags dd ?  fs\_base dq ?  fs\_size dq ?  fs\_map\_base dq ?  fs\_map\_size dq ?  fs\_first\_file dq ?  fs\_uid dq ?  fs\_block\_size dd ?  ; Заголовок файла  virtual at 0x800  f\_info:  f\_name rb 256  f\_next dq ?  f\_prev dq ?  f\_parent dq ?  f\_flags dq ?  f\_data dq ?  f\_size dq ?  f\_ctime dq ?  f\_mtime dq ?  f\_atime dq ?  end virtual ; Данные начального загрузчика  label sector\_per\_track word at $$  label head\_count byte at $$ + 2  label disk\_id byte at $$ + 3  boot\_msg db "MyOS boot loader. Version 0.04",13,10,0  reboot\_msg db "Press any key...",13,10,0  ...  Все данные мы делаем неинициализированными (с помощью вопросительных знаков вместо значений) - Ассемблер оставит в файле пустое место для них, а заполнять уже будет утилита генерации образа ListFS. Специальная директива virtual позволяет разместить данные "как бы по указанному адресу". На выходной файл это никак не влияет, лишь создаёт метки по желаемым адресам. Заголовок файла мы будем загружать по адресу 0x800 и потом уже анализировать. Это будет специальный 512-байтный буфер для чтения служебных данных файловой системы.  Чтение файла мы разделим на несколько этапов - поиск файла, загрузка данных. Поскольку в 512 байт уместить код чтения с обработкой подкаталогов уже не получится, мы уместим туда лишь код поиска и загрузки, которого хватит, чтобы найти в корневом каталоге файл boot.bin и загрузить его, а уже он будет содержать продолжение загрузчика.  Начнём с подпрограммы для поиска файла в каталоге. Она будет принимать два параметра - имя файла в DS:SI и указатель на первый файл каталога в DX:AX. В результате в f\_info должен оказаться заголовок файла, либо, если файл не найден, вывестись сообщение "NOT FOUND" и система переходит в ожидание перезагрузки (подпрограмма error), потому что на этом этапе любая ошибка является критической.  ; Поиск файла с именем DS:SI в каталоге DX:AX  find\_file:  push cx dx di ; Сохраним регистры  .find:  cmp ax, -1 ; Проверим, не достигли ли мы конца списка  jne @f  cmp dx, -1  jne @f  .not\_found: ; Если достигли, то не остаётся ничего как вывести сообщение об ошибке  call error  db "NOT FOUND",13,10,0  @@:  mov di, f\_info  call load\_sector ; Загружаем сектор с информацией о файле  push di ; Ниже расположен код определения длины имени файла (DI = f\_info = f\_name)  mov cx, 0xFFFF  xor al, al  repne scasb  neg cx  dec cx  pop di  push si ; Сравнение имени файлов  repe cmpsb  pop si  je .found ; Если совпало, то выходим  mov ax, word[f\_next] ; Загружаем номер следующего файла  mov dx, word[f\_next + 2]  jmp .find ; Повторяем цикл  .found:  pop di dx cx ; Восстанавливаем регистры  ret  ; Выходим   Функция выше позволит нам найти файл в каталоге по имени. В качестве значения DX:AX можно указать либо f\_data каталога, либо fs\_first\_file (для поиска в корневом каталоге). После выполнения этой подпрограммы структура f\_info будет содержать всю информацию о файле. После того как файл найден, его можно загрузить в оперативную память. Этим занимается следующая подпрограмма:  ; Загрузка текущего файла в память по  адресу BX:0. Количество загруженных секторов возвращается в AX  load\_file\_data:  push bx cx dx si di ; Сохраняем регистры  mov ax, word[f\_data] ; Загружаем в DX:AX номер сектора с первым списком  mov dx, word[f\_data + 2]  .load\_list:  cmp ax, -1 ; Проверяем, не достигли ли мы конца списка  jne @f  cmp dx, -1  jne @f  .file\_end: ; Мы полностью загрузили файл  pop di si dx cx ; Восстанавливаем все регистры кроме BX  mov ax, bx ; Запоминаем BX  pop bx ; Восстанавливаем его старое значение  sub ax, bx ; Находим разницу - это и будет размер файла в блоках по 16 байт  shr ax, 9 - 4 ; Переводим в сектора  ret ; Выходим  @@:  mov di, f\_info ; Будем загружать список во временный буфер  call load\_sector  mov si, di ; SI := DI  mov cx, 512 / 8 - 1 ; Количество секторов в списке  .load\_sector:  lodsw ; Загружаем очередной номер сектора  mov dx, [si]  add si, 2  cmp ax, -1 ; Проверяем, что мы не в конце списка  jne @f  cmp dx, -1  je .file\_end ; Если в конце, то выходим  @@:  push es  mov es, bx ; Загружаем очередной сектор куда надо  call load\_sector  add bx, 0x200 / 16 ; Следующий сектор будем загружать на 512 байт дальше  pop es  loop .load\_sector ; Данная команда уменьшает CX на 1 и, если он ещё больше нуля, переходит на метку .load\_sector  lodsw ; Загружаем в DX:AX номер следующего списка  mov dx, [si]  jmp .load\_list ; Переходим в начало цикла по спискам секторов   Готово, теперь у нас есть всё, чтобы загрузить продолжение загрузчика любой длины (надо только помнить, что лишь первые 640 КБ ОЗУ можно свободно изменять, дальше идут служебные области и BIOS, которые трогать не стоит). Что мы собственно и сделаем:  ; Загрузим продолжение начального загрузчика  mov si, boot\_file\_name  mov ax, word[fs\_first\_file]  mov dx, word[fs\_first\_file + 2]  call find\_file  mov di, 0x7E00 / 16  call load\_file\_data  ; Переходим на продолжение  jmp 0x7E00   А в области данных опишем:  boot\_file\_name db "boot.bin",0  Ну вот наш загрузочный сектор и завершён :-)  Он настраивает содержимое регистров, как нам нужно, определяет параметры загрузочного устройства, находит и загружает в память файл boot.bin и, наконец, передаёт управление на него. При этом корректно обрабатывает все ошибки, кроме нехватки памяти (если boot.bin окажется больше, чем 500 с чем-то килобайт). От вывода названия загрузчика в этом месте пришлось отказаться, потому что иначе уже не получалось уложиться в один сектор.  Теперь продолжим писать код. В этом выпуске мы реализуем полноценную функцию загрузки файла, которая поддерживает подкаталоги. Располагаться она будет, как и весь остальной код во второй половине загрузчика. Мы не будем особо думать, а просто продолжим писать код после db 0x55,0xAA. При создании образа файл boot.bin следует разделить на две части - первые 512 байт подлежат записи в загрузочный сектор, всё остальное - в файл boot.bin уже на диске. Теперь мы не скованы размерами сектора и можно "разгуляться".  Приведу полнофункциональную подпрограмму загрузки файла:  ; Дополнительные данные загрузчика  load\_msg\_preffix db "Loading '",0  load\_msg\_suffix db "'...",0  ok\_msg db "OK",13,10,0  ; Разбиение строки DS:SI по символу слеша  split\_file\_name:  push si  @@:  lodsb  cmp al, "/"  je @f  jmp @b  @@:  mov byte[si - 1], 0  mov ax, si  pop si  ret  ; Загрузка файла с именем DS:SI в буфер DI:0. Размер файла в секторах возвращается в AX  load\_file:  push si  mov si, load\_msg\_preffix  call write\_str  pop si  call write\_str  push si  mov si, load\_msg\_suffix  call write\_str  pop si  push si bp  mov dx, word[fs\_first\_file + 2] ; Начинаем поиск с корневого каталога  mov ax, word[fs\_first\_file]  @@:  push ax  call split\_file\_name  mov bp, ax  pop ax  call find\_file  test byte[f\_flags], 1 ; Каталог?  jz @f  mov si, bp  mov dx, word[f\_data + 2]  mov ax, word[f\_data]  jmp @b  @@:  call load\_file\_data  mov si, ok\_msg  call write\_str  pop bp si  ret   Ну вот теперь работа с ListFS реализована в нашем загрузчике полностью. Можно читать любые файлы с диска, например, как-нибудь так:  mov si, file\_name  mov di, 0x8000 / 16  call load\_file  ...  file\_name db "system/kernel.bin",0  На этом сегодняшний выпуск можно закончить. Мы проделали большую работу, реализовав все необходимые для загрузчика функции и убрав ограничение на размер. В следующем выпуске мы закончим работу над загрузчиком, сделав загрузку файла ядра и перехода в защищённый режим. После этого начнётся более интересная часть разработки, уже на языке высокого уровня - Си.  В заключение приведу полный исходный код загрузчика:   ; Начальный загрузчик  ядра для архитектуры x86  format Binary as "bin"  org 0x7C00  jmp boot  ; Заголовок ListFS  align 4  fs\_magic dd ?  fs\_version dd ?  fs\_flags dd ?  fs\_base dq ?  fs\_size dq ?  fs\_map\_base dq ?  fs\_map\_size dq ?  fs\_first\_file dq ?  fs\_uid dq ?  fs\_block\_size dd ?  ; Заголовок файла  virtual at 0x800  f\_info:  f\_name rb 256  f\_next dq ?  f\_prev dq ?  f\_parent dq ?  f\_flags dq ?  f\_data dq ?  f\_size dq ?  f\_ctime dq ?  f\_mtime dq ?  f\_atime dq ?  end virtual  ; Данные начального загрузчика  label sector\_per\_track word at $$  label head\_count byte at $$ + 2  label disk\_id byte at $$ + 3  reboot\_msg db "Press any key...",13,10,0  boot\_file\_name db "boot.bin",0  ; Вывод строки DS:SI на экран  write\_str:  push si  mov ah, 0x0E  @@:  lodsb  test al, al  jz @f  int 0x10  jmp @b  @@:  pop si  ret  ; Критическая ошибка  error:  pop si  call write\_str  ; Перезагрузка  reboot:  mov si, reboot\_msg  call write\_str  xor ah, ah  int 0x16  jmp 0xFFFF:0  ; Загрузка сектора DX:AX в буфер ES:DI  load\_sector:  cmp byte[sector\_per\_track], 0xFF  je .use\_EDD  push bx cx dx si  div [sector\_per\_track]  mov cl, dl  inc cl  div [head\_count]  mov dh, ah  mov ch, al  mov dl, [disk\_id]  mov bx, di  mov al, 1  mov si, 3  @@:  mov ah, 2  int 0x13  jnc @f  xor ah, ah  int 0x13  dec si  jnz @b  .error:  call error  db "DISK ERROR",13,10,0  @@:  pop si dx cx bx  ret  .use\_EDD:  push dx si  mov byte[0x600], 0x10  mov byte[0x601], 0  mov word[0x602], 1  mov [0x604], di  push es  pop word[0x606]  mov [0x608], ax  mov [0x60A], dx  mov word[0x60C], 0  mov word[0x60E], 0  mov ah, 0x42  mov dl, [disk\_id]  mov si, 0x600  int 0x13  jc .error  pop si dx  ret  ; Поиск файла с именем DS:SI в каталоге DX:AX  find\_file:  push cx dx di  .find:  cmp ax, -1  jne @f  cmp dx, -1  jne @f  .not\_found:  call error  db "NOT FOUND",13,10,0  @@:  mov di, f\_info  call load\_sector  push di  mov cx, 0xFFFF  xor al, al  repne scasb  neg cx  dec cx  pop di  push si  repe cmpsb  pop si  je .found  mov ax, word[f\_next]  mov dx, word[f\_next + 2]  jmp .find  .found:  pop di dx cx  ret  ; Загрузка текущего файла в память по адресу BX:0. Количество загруженных секторов возвращается в AX  load\_file\_data:  push bx cx dx si di  mov ax, word[f\_data]  mov dx, word[f\_data + 2]  .load\_list:  cmp ax, -1  jne @f  cmp dx, -1  jne @f  .file\_end:  pop di si dx cx  mov ax, bx  pop bx  sub ax, bx  shr ax, 9 - 4  ret  @@:  mov di, 0x8000 / 16  call load\_sector  mov si, di  mov cx, 512 / 8 - 1  .load\_sector:  lodsw  mov dx, [si]  add si, 6  cmp ax, -1  jne @f  cmp dx, -1  je .file\_end  @@:  push es  mov es, bx  xor di, di  call load\_sector  add bx, 0x200 / 16  pop es  loop .load\_sector  lodsw  mov dx, [si]  jmp .load\_list  ; Точка входа в начальный загрузчик  boot:  ; Настроим сегментные регистры  jmp 0:@f  @@:  mov ax, cs  mov ds, ax  mov es, ax  ; Настроим стек  mov ss, ax  mov sp, $$  ; Разрешим прерывания  sti  ; Запомним номер загрузочного диска  mov [disk\_id], dl  ; Определим параметры загрузочного диска  mov ah, 0x41  mov bx, 0x55AA  int 0x13  jc @f  mov byte[sector\_per\_track], 0xFF  jmp .disk\_detected  @@:  mov ah, 0x08  xor di, di  push es  int 0x13  pop es  jc load\_sector.error  inc dh  mov [head\_count], dh  and cx, 111111b  mov [sector\_per\_track], cx  .disk\_detected:  ; Загрузим продолжение начального загрузчика  mov si, boot\_file\_name  mov ax, word[fs\_first\_file]  mov dx, word[fs\_first\_file + 2]  call find\_file  mov bx, 0x7E00 / 16  call load\_file\_data  ; Переходим на продолжение  jmp boot2  ; Пустое пространство и сигнатура  rb 510 - ($ - $$)  db 0x55,0xAA  ; Дополнительные данные загрузчика  load\_msg\_preffix db "Loading '",0  load\_msg\_suffix db "'...",0  ok\_msg db "OK",13,10,0  ; Разбиение строки DS:SI по символу слеша  split\_file\_name:  push si  @@:  lodsb  cmp al, "/"  je @f  jmp @b  @@:  mov byte[si - 1], 0  mov ax, si  pop si  ret  ; Загрузка файла с именем DS:SI в буфер BX:0. Размер файла в секторах возвращается в AX  load\_file:  push si  mov si, load\_msg\_preffix  call write\_str  pop si  call write\_str  push si  mov si, load\_msg\_suffix  call write\_str  pop si  push si bp  mov dx, word[fs\_first\_file + 2]  mov ax, word[fs\_first\_file]  @@:  push ax  call split\_file\_name  mov bp, ax  pop ax  call find\_file  test byte[f\_flags], 1  jz @f  mov si, bp  mov dx, word[f\_data + 2]  mov ax, word[f\_data]  jmp @b  @@:  call load\_file\_data  mov si, ok\_msg  call write\_str  pop bp si  ret  ; Продолжение начального загрузчика  boot2:  ; Завершение  jmp reboot   Утилита создания образов дисков make\_listfs.c:  #include <stdlib.h>  #include <stdio.h>  #include <stdbool.h>  #include <errno.h>  #include <dirent.h>  #include <string.h>  #include <time.h>  #include <sys/stat.h>  #define LISTFS\_MAGIC 0x84837376  #define LISTFS\_VERSION 0x0001  typedef unsigned char uint8;  typedef signed char int8;  typedef unsigned short uint16;  typedef signed short int16;  typedef unsigned int uint32;  typedef signed int int32;  typedef unsigned long long uint64;  typedef signed long long int64;  typedef struct {  char jump[4];  uint32 magic;  uint32 version;  uint32 flags;  uint64 base;  uint64 size;  uint64 map\_base;  uint64 map\_size;  uint64 first\_file;  uint64 uid;  uint32 block\_size;  } \_\_attribute((packed)) listfs\_header;  typedef struct {  char name[256];  uint64 next;  uint64 prev;  uint64 parent;  uint64 flags;  uint64 data;  uint64 size;  uint64 create\_time;  uint64 modify\_time;  uint64 access\_time;  } \_\_attribute((packed)) listfs\_file\_header;  char \*output\_file\_name = NULL;  char \*boot\_loader\_file\_name = NULL;  char \*source\_dir\_name = NULL;  long block\_size = 512;  long block\_count = 0;  FILE \*output\_file;  listfs\_header \*fs\_header;  char \*disk\_map;  long boot\_loader\_extra\_blocks = 0;  bool get\_commandline\_options(int argc, char \*\*argv) {  int i;  for (i = 1; i < argc; i++) {  if (!strncmp(argv[i], "of=", 3)) {  output\_file\_name = argv[i] + 3;  } else if (!strncmp(argv[i], "bs=", 3)) {  block\_size = atoi(argv[i] + 3);  } else if (!strncmp(argv[i], "size=", 5)) {  block\_count = atoi(argv[i] + 5);  } else if (!strncmp(argv[i], "boot=", 5)) {  boot\_loader\_file\_name = argv[i] + 5;  } else if (!strncmp(argv[i], "src=", 4)) {  source\_dir\_name = argv[i] + 4;  }  }  return (output\_file\_name && block\_count);  }  void usage() {  printf("Syntax:\n");  printf(" make\_listfs [options]\n");  printf("Options:\n");  printf(" of=XXX - output file name (required)\n");  printf(" bs=XXX - size of disk image block in bytes (optional, default - 512)\n");  printf(" size=XXX - size of disk image in blocks (required)\n");  printf(" boot=XXX - name of boot loader image file (optional, need only for bootable image)\n");  printf(" src=XXX - Dir, that contents will be copied on created disk image (optional)\n");  printf("Author:\n");  printf(" kiv (kiv.apple@gmail.com)\n");  printf("\n");  }  bool check\_options() {  if ((block\_size < 512) && !(block\_size & 7)) {  fprintf(stderr, "Error: Block size must be larger than 512 and be multiple of 8!\n");  return false;  }  if (block\_count < 2) {  fprintf(stderr, "Error: Block count must be larger than 1!\n");  return false;  }  return true;  }  bool open\_output\_file() {  char \*buf;  output\_file = fopen(output\_file\_name, "wb+");  if (!output\_file) {  fprintf(stderr, "Error: Could not create output file: %s\n", strerror(errno));  return false;  }  fseek(output\_file, block\_size \* (block\_count - 1), SEEK\_SET);  buf = calloc(1, block\_size);  if (fwrite(buf, block\_size, 1, output\_file) == 0) {  fprintf(stderr, "Error: Could not create output file: %s\n", strerror(errno));  fclose(output\_file);  remove(output\_file\_name);  free(buf);  return false;  }  free(buf);  return true;  }  void close\_output\_file() {  fclose(output\_file);  }  void write\_block(uint64 index, void \*data, unsigned int count) {  fseek(output\_file, block\_size \* index, SEEK\_SET);  fwrite(data, block\_size, count, output\_file);  }  void init\_listfs\_header() {  fs\_header = calloc(block\_size, 1);  if (boot\_loader\_file\_name) {  FILE \*f;  f = fopen(boot\_loader\_file\_name, "rb");  if (f) {  fread(fs\_header, block\_size, 1, f);  if (!feof(f)){  char \*buffer = calloc(block\_size, 1);  int i = 1;  while (!feof(f)) {  fread(buffer, block\_size, 1, f);  write\_block(i, buffer, 1);  i++;  boot\_loader\_extra\_blocks++;  }  }  fclose(f);  } else  fprintf(stderr, "Warning: Could not open boot loader file for reading. No boot loader will be installed.\n");  }  fs\_header->magic = LISTFS\_MAGIC;  fs\_header->version = LISTFS\_VERSION;  fs\_header->base = 0;  fs\_header->size = block\_count;  fs\_header->first\_file = -1;  fs\_header->flags = 0;  fs\_header->block\_size = block\_size;  fs\_header->uid = (time(NULL) << 16) | (rand() & 0xFFFF);  }  void store\_listfs\_header() {  write\_block(0, fs\_header, 1);  free(fs\_header);  }  uint64 alloc\_disk\_block() {  uint64 index = 0;  while (disk\_map[index >> 3] & (1 << (index & 7))) {  index++;  if (index >= fs\_header->size) {  fprintf(stderr, "Error: Disk image is full. Could not write more.\n");  exit(-3);  }  }  disk\_map[index >> 3] |= 1 << (index & 7);  return index;  }  void get\_disk\_block(uint64 index) {  disk\_map[index >> 3] |= 1 << (index & 7);  }  void init\_disk\_map() {  int i;  fs\_header->map\_base = /\* 1 + \*/ boot\_loader\_extra\_blocks;  fs\_header->map\_size = block\_count / 8;  if (fs\_header->map\_size % block\_size) fs\_header->map\_size += block\_size;  fs\_header->map\_size /= block\_size;  disk\_map = calloc(block\_size, fs\_header->map\_size);  for (i = 0; i < fs\_header->map\_base + fs\_header->map\_size; i++) {  get\_disk\_block(i);  }  }  void store\_disk\_map() {  write\_block(fs\_header->map\_base, disk\_map, fs\_header->map\_size);  free(disk\_map);  }  uint64 store\_file\_data(FILE \*file) {  if (feof(file)) return -1;  uint64 \*block\_list = malloc(block\_size);  uint64 block\_list\_index = alloc\_disk\_block();  char \*block\_data = malloc(block\_size);  int i;  for (i = 0; i < (block\_size / 8 - 1); i++) {  if (!feof(file)) {  memset(block\_data, 0, block\_size);  if (fread(block\_data, 1, block\_size, file) == 0) {  block\_list[i] = -1;  continue;  }  uint64 block\_index = alloc\_disk\_block();  block\_list[i] = block\_index;  write\_block(block\_index, block\_data, 1);  } else {  block\_list[i] = -1;  }  }  block\_list[block\_size / 8 - 1] = store\_file\_data(file);  write\_block(block\_list\_index, block\_list, 1);  free(block\_list);  free(block\_data);  return block\_list\_index;  }  uint64 process\_dir(uint64 parent, char \*dir\_name) {  DIR \*dir = opendir(dir\_name);  struct dirent \*entry;  uint64 prev\_file = -1;  uint64 cur\_file;  uint64 first\_file = -1;  char file\_name[260];  struct stat file\_stat;  listfs\_file\_header \*file\_header;  if (!dir) {  fprintf(stderr, "Warning: Could not read contents of directory '%s': %s\n", dir\_name, strerror(errno));  return -1;  }  file\_header = malloc(block\_size);  while (entry = readdir(dir)) {  if (!strcmp(entry->d\_name, ".") || !strcmp(entry->d\_name, "..")) continue;  cur\_file = alloc\_disk\_block();  if (prev\_file != -1) {  file\_header->next = cur\_file;  write\_block(prev\_file, file\_header, 1);  } else  first\_file = cur\_file;  memset(file\_header, 0, block\_size);  file\_header->prev = prev\_file;  file\_header->next = -1;  strncpy(file\_header->name, entry->d\_name, sizeof(file\_header->name) - 1);  file\_header->name[sizeof(file\_header->name) - 1] = 0;  file\_header->size = 0;  sprintf((char\*)&file\_name, "%s/%s", dir\_name, entry->d\_name);  stat(file\_name, &file\_stat);  file\_header->parent = parent;  file\_header->flags = 0;  file\_header->create\_time = file\_stat.st\_ctime;  file\_header->modify\_time = file\_stat.st\_mtime;  file\_header->access\_time = file\_stat.st\_atime;  if (S\_ISDIR(file\_stat.st\_mode)) {  file\_header->flags = 1;  file\_header->data = process\_dir(cur\_file, file\_name);  } else if (S\_ISREG(file\_stat.st\_mode)) {  FILE \*f = fopen(file\_name, "rb");  if (f) {  file\_header->size = file\_stat.st\_size;  file\_header->data = store\_file\_data(f);  fclose(f);  } else {  fprintf(stderr, "Warning: Could not open file '%s' for reading!\n", file\_name);  file\_header->data = -1;  }  } else {  fprintf(stderr, "Warning: File '%s' is not directory or regular file.\n", file\_name);  }  prev\_file = cur\_file;  }  write\_block(prev\_file, file\_header, 1);  free(file\_header);  closedir(dir);  return first\_file;  }  int main(int argc, char \*argv[]) {  if (!get\_commandline\_options(argc, argv)) {  usage();  return 0;  }  if (!check\_options()) return -1;  if (!open\_output\_file()) return -2;  init\_listfs\_header();  init\_disk\_map();  if (source\_dir\_name) fs\_header->first\_file = process\_dir(-1, source\_dir\_name);  store\_disk\_map();  store\_listfs\_header();  close\_output\_file();  return 0;  }   Пример скрипта создания образа:  dd if=bin/boot.bios.bin of=bin/boot\_sector.bin bs=512 count=1  dd if=bin/boot.bios.bin of=disk/boot.bin bs=1 skip=512  bin/make\_listfs of=disk.img bs=512 size=2880 boot=bin/boot\_sector.bin src=./disk |

Пишем свою операционную систему. Загрузка ядра и модулей

|  |
| --- |
| Приветствую всех!  В прошлом выпуске мы написали подпрограммы для работы с файловой системой ListFS и реализовали загрузку продолжения загрузчика из файла на диске. Теперь мы не ограничены 512 байтами и можем дописать начальный загрузчик. Я не буду его жёстко привязывать к своему ядру - загрузчик будет читать конфигурационный файл, в котором содержится информация о расположении ядра и модулей, загружать все необходимые файлы в память и передавать управление ядра, предварительно переведя процессор в нужный режим.  Для начала собственно загрузка конфигурационного файла:  ; Дополнительные данные загрузчика  ...  config\_file\_name db "boot.cfg",0  ...  ; Продолжение начального загрузчика  boot2:  ; Загрузим конфигурационный файл загрузчика  mov si, config\_file\_name  mov bx, 0x1000 / 16  call load\_file  Теперь по адресу 0000:1000 расположен текстовый конфиг загрузчика. Он имеет очень простой формат. Первый символ строки определяет её тип. Символ "#" в начале говорит о том, что вся строка далее является комментарием и должна быть пропущена. Также пропускаются пустые строки. Символ "L" (L:oad) обозначает, что за ним следует имя файла, подлежащего загрузки. Файл загружается в свободное пространство оперативной памяти. Первый загруженный таким образом файл считается ядром и на него будет передано управление, остальные файлы могут иметь произвольный формат - вся обработка этих файлов ложится на плечи ядра. Подразумевается, что эти файлы являются дополнительными модулям, которые необходимы для работы с диском, файловой системой и прочее, без чего система не сможет нормально работать. И наконец символ "S" (Start) приводит к немедленному запуску ядра. Символы следующие далее игнорируются (запустив ядро, загрузчик остаётся не у дел). После этого символа должны идти два символа - "16", "32" или "64". В зависимости от этого загрузчик может подготовить необходимое окружение. Загрузчик передаёт ядру в качестве параметров немного информации - список загруженных файлов, объём оперативной памяти, номер загрузочного диска.  Анализ загрузочного скрипта (пожалуй, конфиг можно называть и так, ведь он содержит команды) можно выполнять достаточно просто - перебираем символы файла, запомнив начало. Если натыкаемся на символ переноса строки, то время прерваться и посмотреть на первый символ. Выполнив действие, мы продолжаем перебор с предыдущей позиции.  ; Выполним загрузочный скрипт  mov dx, 0x1000  .parse\_line:  mov si, dx  .parse\_char:  lodsb  test al, al  jz .config\_end  cmp al, 10  je .run\_command  cmp al, 13  je .run\_command  jmp .parse\_char  .run\_command:  mov byte[si - 1], 0  xchg dx, si ; Обмен значений DX и SI  cmp byte[si], 0  je .parse\_line ; Пустая строка  cmp byte[si], "#"  je .parse\_line ; Комментарий  cmp byte[si], "L"  je .load\_file ; Загрузка файла  cmp byte[si], "S"  je .start ; Запуск ядра  ; Неизвестная команда  mov al, [si]  mov [.cmd], al  call error  db "Unknown boot script command '"  .cmd db ?  db "'!",13,10,0  .config\_end: ; При правильном конфигурационном файле мы не должны сюда попасть  ; Завершение  jmp reboot   Осталось реализовать пару подпрограмм - загрузки файла и запуска ядра. Начнём с более простой - загрузки файла. Список файлов будет хранится под адресу 0000:6000 (таким образом для конфига останется 20 КБ, чего вполне достаточно) . Каждый элемент списка состоит из двух 64-битных целых чисел - базовый адрес загрузки файла (физический, а не сегмент:смещение) и размер файла в байтах. Последний элемент списка содержит нули (его созданием займётся .start).  Указатель на текущий элемент списка содержит BP (перед разбором конфига BP необходимо присвоить 0x6000), указатель на сегмент, куда следует загружать очередной файл содержит BX (перед разбором конфига ему следует присвоить 0x9000. 4 КБ хватит для продолжения загрузчика). Вот такой в итого получается код разбора конфигурационного файла:  ; Продолжение начального загрузчика  boot2:  ; Загрузим конфигурационный файл загрузчика  mov si, config\_file\_name  mov bx, 0x1000 / 16  call load\_file  ; Выполним загрузочный скрипт  mov bx, 0x9000 / 16  mov bp, 0x6000  mov dx, 0x1000  .parse\_line:  mov si, dx  .parse\_char:  lodsb  test al, al  jz .config\_end  cmp al, 10  je .run\_command  cmp al, 13  je .run\_command  jmp .parse\_char  .run\_command:  mov byte[si - 1], 0  xchg dx, si  cmp byte[si], 0  je .parse\_line ; Пустая строка  cmp byte[si], "#"  je .parse\_line ; Комментарий  cmp byte[si], "L"  je .load\_file ; Загрузка файла  cmp byte[si], "S"  je .start ; Запуск ядра  ; Неизвестная команда  mov al, [si]  mov [.cmd], al  call error  db "Unknown boot script command '"  .cmd db ?  db "'!",13,10,0  .config\_end: ; При правильном конфигурационном файле мы не должны сюда попасть  ; Завершение  jmp reboot  ; Загрузка файла  .load\_file:  push dx  inc si  call load\_file  push ax  mov cx, 512  mul cx  mov word[bp + 8], ax  mov word[bp + 10], dx  mov word[bp + 12], 0  mov word[bp + 14], 0  mov ax, bx  mov cx, 16  mul cx  mov word[bp], ax  mov word[bp + 2], dx  mov word[bp + 4], 0  mov word[bp + 6], 0  pop ax  shr ax, 9 - 4  add bx, ax  add bp, 16  pop dx  jmp .parse\_line  ; Запуск ядра  .start:  ; Проверим, что загружен хотя бы один файл  cmp bx, 0x9000 / 16  ja @f  call error  db "NO KERNEL LOADED",13,10,0  @:  ; Заполняем последний элемент списка файлов  xor ax, ax  mov cx, 16  mov di, bp  rep stosw  ...   Попутно, я заметил баг в функции split\_file\_name, из-за которого load\_file не находил файлы в подкаталогах. Вот исправление:  ; Разбиение строки DS:SI по символу слеша  split\_file\_name:  push si  @:  lodsb  cmp al, "/"  je @f  **test al, al**  **jz @f**  jmp @b  @:  mov byte[si - 1], 0  mov ax, si  pop si  ret   Уже сейчас можно подготовить конфигурационный файл boot.cfg вроде такого:  # Loading kernel  Lsystem/kernel.bin  # Boot 64 bit kernel  S64  Загрузчик загрузит все указанные файлы и приступит к инициализации ядра, которую нам осталось дописать.  Для полноты реализации напишем короткий код для старта 16-битного ядра, 32-битные ядра пока загружать не будем, потому что я планирую писать сразу 64-битную систему.  ; Запуск ядра  .start:  ; Проверим, что загружен хотя бы один файл  cmp bx, 0x9000 / 16  ja @f  call error  db "NO KERNEL LOADED",13,10,0  @:  ; Заполняем последний элемент списка файлов  xor ax, ax  mov cx, 16  mov di, bp  rep stosw  ; Переходим к процедуре инициализации ядра для нужной разрядности  inc si  cmp word[si], "16"  je .start16  cmp word[si], "32"  je .start32  cmp word[si], "64"  je .start64  ; Неизвестная рязрядность ядра  call error  db "Invalid start command argument",13,10,0  ; Запуск 16-разрядного ядра  .start16:  mov si, start16\_msg  mov bx, 0x6000  mov dl, [disk\_id]  jmp 0x9000  ; Запуск 32-разрядного ядра  .start32:  call error  db "Starting 32 bit kernels is not implemented yet",13,10,0  ; Запуск 64-разрядного ядра  .start64:  ; Выводим уведомление о запуске 64-битного ядра  mov si, start64\_msg  call write\_str  ;  jmp reboot  Если ядро 16-разрядное (это может также пригодится опытным осеписателям, чтобы самостоятельно инициализировать всё). Код для запуска 64-битного ядра нам ещё нужно написать. Мы уже вышли на "финишную прямую", однако перед переходом в защищённый режим, мне необходимо рассказать больше теории, поэтому к написанию кода инициализации ядра мы приступим в следующем выпуске.  В заключение приведу полный код нашего начального загрузчика, который уже почти принял окончательный вид:  ; Начальный загрузчик ядра для архитектуры x86  format Binary as "bin"  org 0x7C00  jmp boot  ; Заголовок ListFS  align 4  fs\_magic dd ?  fs\_version dd ?  fs\_flags dd ?  fs\_base dq ?  fs\_size dq ?  fs\_map\_base dq ?  fs\_map\_size dq ?  fs\_first\_file dq ?  fs\_uid dq ?  fs\_block\_size dd ?  ; Заголовок файла  virtual at 0x800  f\_info:  f\_name rb 256  f\_next dq ?  f\_prev dq ?  f\_parent dq ?  f\_flags dq ?  f\_data dq ?  f\_size dq ?  f\_ctime dq ?  f\_mtime dq ?  f\_atime dq ?  end virtual  ; Данные начального загрузчика  label sector\_per\_track word at $$  label head\_count byte at $$ + 2  label disk\_id byte at $$ + 3  reboot\_msg db "Press any key...",13,10,0  boot\_file\_name db "boot.bin",0  ; Вывод строки DS:SI на экран  write\_str:  push si  mov ah, 0x0E  @:  lodsb  test al, al  jz @f  int 0x10  jmp @b  @:  pop si  ret  ; Критическая ошибка  error:  pop si  call write\_str  ; Перезагрузка  reboot:  mov si, reboot\_msg  call write\_str  xor ah, ah  int 0x16  jmp 0xFFFF:0  ; Загрузка сектора DX:AX в буфер ES:DI  load\_sector:  push dx  add ax, word[fs\_base]  adc dx, word[fs\_base + 2]  cmp byte[sector\_per\_track], 0xFF  je .use\_EDD  push bx cx si  div [sector\_per\_track]  mov cl, dl  inc cl  div [head\_count]  mov dh, ah  mov ch, al  mov dl, [disk\_id]  mov bx, di  mov al, 1  mov si, 3  @:  mov ah, 2  int 0x13  jnc @f  xor ah, ah  int 0x13  dec si  jnz @b  .error:  call error  db "DISK ERROR",13,10,0  @:  pop si cx bx dx  ret  .use\_EDD:  push si  mov byte[0x600], 0x10  mov byte[0x601], 0  mov word[0x602], 1  mov [0x604], di  push es  pop word[0x606]  mov [0x608], ax  mov [0x60A], dx  mov word[0x60C], 0  mov word[0x60E], 0  mov ah, 0x42  mov dl, [disk\_id]  mov si, 0x600  int 0x13  jc .error  pop si dx  ret  ; Поиск файла с именем DS:SI в каталоге DX:AX  find\_file:  push cx dx di  .find:  cmp ax, -1  jne @f  cmp dx, -1  jne @f  .not\_found:  call error  db "NOT FOUND",13,10,0  @:  mov di, f\_info  call load\_sector  push di  mov cx, 0xFFFF  xor al, al  repne scasb  neg cx  dec cx  pop di  push si  repe cmpsb  pop si  je .found  mov ax, word[f\_next]  mov dx, word[f\_next + 2]  jmp .find  .found:  pop di dx cx  ret  ; Загрузка текущего файла в память по адресу BX:0. Количество загруженных секторов возвращается в AX  load\_file\_data:  push bx cx dx si di  mov ax, word[f\_data]  mov dx, word[f\_data + 2]  .load\_list:  cmp ax, -1  jne @f  cmp dx, -1  jne @f  .file\_end:  pop di si dx cx  mov ax, bx  pop bx  sub ax, bx  shr ax, 9 - 4  ret  @:  mov di, 0x8000 / 16  call load\_sector  mov si, di  mov cx, 512 / 8 - 1  .load\_sector:  lodsw  mov dx, [si]  add si, 6  cmp ax, -1  jne @f  cmp dx, -1  je .file\_end  @:  push es  mov es, bx  xor di, di  call load\_sector  add bx, 0x200 / 16  pop es  loop .load\_sector  lodsw  mov dx, [si]  jmp .load\_list  ; Точка входа в начальный загрузчик  boot:  ; Настроим сегментные регистры  jmp 0:@f  @:  mov ax, cs  mov ds, ax  mov es, ax  ; Настроим стек  mov ss, ax  mov sp, $$  ; Разрешим прерывания  sti  ; Запомним номер загрузочного диска  mov [disk\_id], dl  ; Определим параметры загрузочного диска  mov ah, 0x41  mov bx, 0x55AA  int 0x13  jc @f  mov byte[sector\_per\_track], 0xFF  jmp .disk\_detected  @:  mov ah, 0x08  xor di, di  push es  int 0x13  pop es  jc load\_sector.error  inc dh  mov [head\_count], dh  and cx, 111111b  mov [sector\_per\_track], cx  .disk\_detected:  ; Загрузим продолжение начального загрузчика  mov si, boot\_file\_name  mov ax, word[fs\_first\_file]  mov dx, word[fs\_first\_file + 2]  call find\_file  mov bx, 0x7E00 / 16  call load\_file\_data  ; Переходим на продолжение  jmp boot2  ; Пустое пространство и сигнатура  rb 510 - ($ - $$)  db 0x55,0xAA  ; Дополнительные данные загрузчика  load\_msg\_preffix db "Loading '",0  load\_msg\_suffix db "'...",0  ok\_msg db "OK",13,10,0  config\_file\_name db "boot.cfg",0  start16\_msg db "Starting 16 bit kernel...",13,10,0  start64\_msg db "Starting 64 bit kernel...",13,10,0  ; Разбиение строки DS:SI по символу слеша  split\_file\_name:  push si  @:  lodsb  cmp al, "/"  je @f  test al, al  jz @f  jmp @b  @:  mov byte[si - 1], 0  mov ax, si  pop si  ret  ; Загрузка файла с именем DS:SI в буфер BX:0. Размер файла в секторах возвращается в AX  load\_file:  push si  mov si, load\_msg\_preffix  call write\_str  pop si  call write\_str  push si  mov si, load\_msg\_suffix  call write\_str  pop si  push si bp  mov dx, word[fs\_first\_file + 2]  mov ax, word[fs\_first\_file]  @:  push ax  call split\_file\_name  mov bp, ax  pop ax  call find\_file  test byte[f\_flags], 1  jz @f  mov si, bp  mov dx, word[f\_data + 2]  mov ax, word[f\_data]  jmp @b  @:  call load\_file\_data  mov si, ok\_msg  call write\_str  pop bp si  ret  ; Продолжение начального загрузчика  boot2:  ; Загрузим конфигурационный файл загрузчика  mov si, config\_file\_name  mov bx, 0x1000 / 16  call load\_file  ; Выполним загрузочный скрипт  mov bx, 0x9000 / 16  mov bp, 0x6000  mov dx, 0x1000  .parse\_line:  mov si, dx  .parse\_char:  lodsb  test al, al  jz .config\_end  cmp al, 10  je .run\_command  cmp al, 13  je .run\_command  jmp .parse\_char  .run\_command:  mov byte[si - 1], 0  xchg dx, si  cmp byte[si], 0  je .parse\_line ; Пустая строка  cmp byte[si], "#"  je .parse\_line ; Комментарий  cmp byte[si], "L"  je .load\_file ; Загрузка файла  cmp byte[si], "S"  je .start ; Запуск ядра  ; Неизвестная команда  mov al, [si]  mov [.cmd], al  call error  db "Unknown boot script command '"  .cmd db ?  db "'!",13,10,0  .config\_end: ; При правильном конфигурационном файле мы не должны сюда попасть  ; Завершение  jmp reboot  ; Загрузка файла  .load\_file:  push dx  inc si  call load\_file  push ax  mov cx, 512  mul cx  mov word[bp + 8], ax  mov word[bp + 10], dx  mov word[bp + 12], 0  mov word[bp + 14], 0  mov ax, bx  mov cx, 16  mul cx  mov word[bp], ax  mov word[bp + 2], dx  mov word[bp + 4], 0  mov word[bp + 6], 0  pop ax  shr ax, 9 - 4  add bx, ax  add bp, 16  pop dx  jmp .parse\_line  ; Запуск ядра  .start:  ; Проверим, что загружен хотя бы один файл  cmp bx, 0x9000 / 16  ja @f  call error  db "NO KERNEL LOADED",13,10,0  @:  ; Заполняем последний элемент списка файлов  xor ax, ax  mov cx, 16  mov di, bp  rep stosw  ; Переходим к процедуре инициализации ядра для нужной разрядности  inc si  cmp word[si], "16"  je .start16  cmp word[si], "32"  je .start32  cmp word[si], "64"  je .start64  ; Неизвестная рязрядность ядра  call error  db "Invalid start command argument",13,10,0  ; Запуск 16-разрядного ядра  .start16:  mov si, start16\_msg  mov bx, 0x6000  mov dl, [disk\_id]  jmp 0x9000  ; Запуск 32-разрядного ядра  .start32:  call error  db "Starting 32 bit kernels is not implemented yet",13,10,0  ; Запуск 64-разрядного ядра  .start64:  ; Выводим уведомление о запуске 64-битного ядра  mov si, start64\_msg  call write\_str  ;  jmp reboot |

## Пишем свою операционную систему. Переход в защищённый режим

|  |
| --- |
| После недельного перерыва вновь могу обрадовать вас новым выпуском своей рассылки.  Итак, как вы помните, в реальном режиме работы процессора нам доступен всего лишь 1 мегабайт адресного пространства (из которых обычной памятью является всего лишь 640 килобайт). Так это и было во времена первых процессоров вроде 8086, но постепенно объёма оперативной памяти стало не хватать. В то же время требовалось сохранить полную обратную совместимость, чтобы 16-разрядные операционные системы вроде DOS смогли нормально работать. Поэтому был введён новый режим работы процессора - защищённый режим. После перехода в него для адресации используется не 16, а 32 или даже 64 бита, а сегменты в старом понимании исчезают. Также добавляются защитные механизмы (именно поэтому **защищённый** режим), чтобы ядро ОС было изолированно от приложений и могло ими свободно управлять. Это необходимо любой полноценной многозадачной системе.  Начнём с того, что добавляется в реальном режиме на процессорах , которые поддерживают 32-битный адрес (i386 и выше) - добавляются новые регистры, вернее расширяются старые: EAX, EBX, ECX, EDX, ESP, EBP, EIP, ESI, EDI, EFLAGS. Как можно догадаться, это 32-битные версии обычных регистров реального режима (к имени регистра добавляется приставка "E"). Все эти 32-битные регистры кроме EIP доступны и в реальном режиме, но в таком случае будут занимать на 1 байт больше (к ним добавляется специальный префикс). На процессоре моложе 286 эти команды будут некорректны. Мы можем, например, написать mov eax, 0x12345678 и после этого в AX будет 0x5678, потому что он как бы является "окном" в младшую часть регистра EAX (аналогично, AL младшая часть AX). Регистра-отображения старшей части 32-битных регистров не существует - можно её извлечь только с помощью арифметики (например, сдвинуть EAX на 16 бит вправо с помощью shr eax, 16, тогда в AX будет старшая половина, но содержимое младших бит будет утеряно). Что характерно, в защищённом режиме наоборот, команды работы с 16-битными регистрами (но не 8-битными) требуют префикс, поэтому несмотря на то, что разрядность в два раза больше, в защищённом режиме быстрее выполняются и занимают меньше места именно команды 32-битной арифметики.  Также, теперь у нас на 2 сегментных регистра больше - GS и FS. Работа с ними полностью аналогична DS и ES и вы можете их свободно использовать в реальном режиме. Отличие только в том, что никакие команды их явно не подразумевают (DS используется по умолчанию практически всеми командами и некоторыми строковыми операциями, ES некоторыми строковыми операциями) и надо явно указывать, что вы хотите обращаться через них. Например, mov ax, [gs:0x1234].  Помимо этого расширения регистров, добавляются новые управляющие регистры (раньше влиял на режим работы процессора только FLAGS) - CR0, CR2, CR3 и CR4. Есть и другие (например, отладочные регистры), но они нас сейчас не интересуют. Именно с помощью этих регистров производится переключение процессора в защищённый режим и настройка новых функций вроде страничной адресации. Они доступны в реальном режиме.  В защищённом режиме понятие сегмента изменяется. Теперь это не просто базовый адрес, а номер элемента (дескриптора сегмента) в специальной таблице. Таблица дескрипторов сегментов создаётся операционной системой и может содержать необходимое количество описаний сегментов защищённого режима. Каждый элемент таблицы занимает 8 байт и в специальном формате описывает базовый адрес сегмента, размер, права доступа и т. д.  Сегменты защищённого режима делятся на два типа - сегменты кода и сегменты данных (на самом деле есть ещё всякие TSS и LDT, но пока они нам не важны тоже). В CS можно загружать только номера дескрипторов, описанных как сегмент кода, в остальные сегментные регистры можно загружать любые сегменты - как данных, так и кода. Важная разница в том, что сегмент кода можно только читать и исполнять, а сегмент данных только читать и писать. К счастью, сегменты могут перекрываться в памяти, поэтому можно создать два дескриптора, ссылающиеся на один и тот же регион памяти, но один из них сделать исполняемым, а другой доступным для записи.  Несмотря на поддержку сегментации, она считается устаревшей. Ни Windows, ни Linux не используют её в полной мере, а на отличных от x86 архитектуры (например, ARM) она вовсе отсутствует. Для разграничения доступа к памяти используется гораздо более гибкий механизм страничной адресации, который мы рассмотрим далее. Чтобы избавиться от сегментации ОС просто описывает таблицу из двух дескрипторов, у каждого из которых базовый адрес 0, а размер 4 ГБ (максимальный размер адресуемой памяти в 32-битном режиме). В таком случае говорят, что мы включили режим линейных адресов - смещение соответствует физическому адресу. Это очень удобно и я пойду по тому же пути. Не следует пытаться использовать сегментацию в своей операционной системе - это сильно усложняет код ядра, языки высокого уровня (например, С или С++) не поддерживают сегментацию (то есть вы сможете полноценно программировать только на Assembler) и, наконец, вы не сможете перенести систему на другую архитектуру, потому что x86 единственная, которая умеет этот механизм (и то, в 64-битном режиме поля базового адреса и размера сегмента игнорируются, а используется лишь информация о правах доступа).  Как я уже сказал, таблица дескрипторов сегментов формируется самой операционной системой. Чтобы указать процессору, где она находится используется специальная команда - lgdt (Load Global Descriptor Table). Она принимает 6-байтовую переменную в памяти. Первые её 16 бит содержат размер таблицы в байтах (таким образом, максимальное количество дескрипторов - 65536 / 8 = 8192), последующие 32 бита - базовый линейный адрес в памяти самой таблицы (то есть без учёта всех сегментов). Имеет смысл выравнять начало таблицы на 16 байт, потому что это улучшает скорость доступа к её элементам. Первый элемент таблицы всегда должен быть равен нулю и любое использование **нулевого селектора** (указатель на элемент таблицы дескрипторов в сегментном регистре называется так) приводит к ошибке. Значит более-менее работоспособная таблица дескрипторов должна содержать хотя бы три дескриптора - пустой, дескриптор кода, дескриптор данных.  Ну что ещё стоит рассказать, прежде, чем мы попробуем перейти в защищённый режим? Пожалуй, ещё стоит упомянуть про уровни доступа. Код ядра системы и код приложений отделены друг от друга с той целью, чтобы ядро могло полностью управлять процессором, а приложения не могли вмешаться в работу ядра (ведь у нас многозадачная ОС). Код исполняется с определённым **уровнем привилегий**. В x86 их целых 4 штуки - от 0 до 3. Нулевой уровень самый привилегированный (может выполнять любые команды и менять режимы работы процессора), третий самый "бесправный". Как и в случае с сегментацией, разработчики x86 переборщили с функционалом и все ОС используют лишь два уровня из четырёх возможных, а другие архитектуры процессора поддерживают только их. У каждого сегмента в его дескрипторе указан **DPL (Descriptor privilege level)** - уровень доступа необходимый для данного сегмента. Непривилегированный код не может получить доступ к сегменту с уровнем доступа 0, а привилегированный код может получить доступ ко всем сегментам.  Селектор сегмента, который содержится в сегментном регистре, является не просто номером элемента в таблице, но и указателем уровня доступа - младшие 2 бита содержат уровень привилегий (от 0 до 3), а уже старшие номер самой таблицы. Таким образом селектор = (индекс\_дескриптора shl 2) + RPL. **RPL** - Requested privelege level - запрашиваемый уровень привилегий. При этом RPL должен быть больше или равен максимальному из DPL и CPL (Current privilege level). CPL равен RPL селектора в CS. Таким образом код не может получить доступа к сегментам, у которых уровень доступа в числовом виде ниже, чем у него самого. Я, вероятно, описал достаточно запутанно, но **вполне можно обойтись RPL = DPL**, как мы и поступим.  Пока мы пишем только ядро, мы будем работать в нулевом кольце защиты (так ещё называют уровни привилегий), чтобы иметь полный доступ к аппаратуре.  Сегментация нам не нужна, поэтому я не буду останавливаться пока что на формате дескриптора, а дам готовые значения. Если интересно, можете почитать[эту статью](http://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%81%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%80_%D1%81%D0%B5%D0%B3%D0%BC%D0%B5%D0%BD%D1%82%D0%B0). Рассмотрим простейший код перехода в защищённый режим.  ; Запуск 32-разрядного ядра  .start32:  ; Выводим уведомление о запуске 32-битного ядра  mov si, start32\_msg  call write\_str  ; Загрузим значение в GDTR  lgdt [gdtr32]  ; Запретим прерывания  cli  ; Перейдём в защищённый режим  mov eax, cr0  or eax, 1  mov cr0, eax  ; Перейдём на 32-битный код  jmp 8:start32  ; Таблица дескрипторов сегментов для 32-битного ядра  align 16  gdt32:  dq 0 ; NULL - 0  dq 0x00CF9A000000FFFF ; CODE - 8  dq 0x00CF92000000FFFF ; DATA - 16  gdtr32:  dw $ - gdt32 - 1  dd gdt32  ; 32-битный код  use32  start32:  ; Настроим сегментные регистры и стек  mov eax, 16  mov ds, ax  mov es, ax  mov fs, ax  mov gs, ax  mov ss, ax  movzx esp, sp  ; Выводим символ на экран  mov byte[0xB8000 + (25 \* 80 - 1) \* 2], "!"  ; Завершение  jmp $  Этот код следует дописать к нашему начальному загрузчику.  Перед переходом в защищённый режим необходимо запретить приём аппаратных прерываний (клавиатура, мышь, таймер и другие устройства), потому что BIOS после перехода остаётся не у дел, а свои обработчики мы ещё не написали, поэтому первое же прерывание обрушит систему.  Непосредственно переход в защищённый режим осуществляет установка нулевого бита в CR0. Именно это мы и делаем (прямой доступ к CR0,2,3,4 невозможен так же как и к сегментным регистрам, поэтому используем EAX). Несмотря на то, что мы уже перешли в защищённый режим, код продолжает исполняться по-прежнему 16-битный. Для окончательного перехода нам нужно обновить содержимое сегментных регистров. Директива Ассемблера use32 говорит ему, что дальнейший код выполняется в защищённом режиме и необходимо переключиться в режим генерации команд для него, а не 16-битного (он используется по умолчанию) .  Команда movzx расширяет второй аргумент до первого. В смысле, что из 16 битного значения SP получается 32-битное. Старшие биты обнуляются (мало ли, что там было до нас). Предпоследняя команда демонстрирует нам возможности защищённого режима - мы обращаемся по абсолютному 32-битному адресу к видео-памяти текстового режима, выводя символ "!" в правый нижний угол экрана (текстовый экран имеет разрешение 80 x 25 символов, каждый символ занимает в памяти два байта - код символа и его атрибуты цвета).  Мы больше не можем обращаться к сервисам BIOS, теперь пришло время нам стать полностью самостоятельными и самим управлять всем оборудованием. Перезагружаться и ждать нажатия на клавишу мы пока не умеем, поэтому просто зависаем с помощью команды jmp $ (переход на ту же самую команду - бесконечный цикл).  В нашем boot.cfg команду S64 пока заменим на S32. Теперь, если вы всё правильно сделали, наш загрузчик будет завершать свою работу выводом восклицательного знака в угол экрана из защищённого режима. Это только начало. Мы наконец-то практически ушли из реального режима (на самом деле там ещё осталось немного дел) в защищённый. Поскольку наш загрузчик выполняется в нулевом сегменте реального режима, все смещения соответствуют физическим адресам и при переходе в защищённый режим, нам не пришлось ничего пересчитывать.  В завершение выпуска, пожалуй, добавлю последний штрих - проверку, что процессор поддерживает защищённый режим. Суть проверки в том, что не все биты FLAGS можно изменить программно. То есть регистр не совсем 16-битный. На новых процессорах доступно для изменения больше бит и это можно обнаружить. Разберите код ниже сами, скажу только, что команда pushf помещает регистр флагов в стек, а popf выталкивает содержимое стека во FLAGS. Таким образом его можно менять целиком, а не отдельными командами. Вот полный код нашего загрузчика:  org 0x7C00  jmp boot  ; Заголовок ListFS  align 4  fs\_magic dd ?  fs\_version dd ?  fs\_flags dd ?  fs\_base dq ?  fs\_size dq ?  fs\_map\_base dq ?  fs\_map\_size dq ?  fs\_first\_file dq ?  fs\_uid dq ?  fs\_block\_size dd ?  ; Заголовок файла  virtual at 0x800  f\_info:  f\_name rb 256  f\_next dq ?  f\_prev dq ?  f\_parent dq ?  f\_flags dq ?  f\_data dq ?  f\_size dq ?  f\_ctime dq ?  f\_mtime dq ?  f\_atime dq ?  end virtual  ; Данные начального загрузчика  label sector\_per\_track word at $$  label head\_count byte at $$ + 2  label disk\_id byte at $$ + 3  reboot\_msg db "Press any key...",13,10,0  boot\_file\_name db "boot.bin",0  ; Вывод строки DS:SI на экран  write\_str:  push si  mov ah, 0x0E  @:  lodsb  test al, al  jz @f  int 0x10  jmp @b  @:  pop si  ret  ; Критическая ошибка  error:  pop si  call write\_str  ; Перезагрузка  reboot:  mov si, reboot\_msg  call write\_str  xor ah, ah  int 0x16  jmp 0xFFFF:0  ; Загрузка сектора DX:AX в буфер ES:DI  load\_sector:  push dx  add ax, word[fs\_base]  adc dx, word[fs\_base + 2]  cmp byte[sector\_per\_track], 0xFF  je .use\_EDD  push bx cx si  div [sector\_per\_track]  mov cl, dl  inc cl  div [head\_count]  mov dh, ah  mov ch, al  mov dl, [disk\_id]  mov bx, di  mov al, 1  mov si, 3  @:  mov ah, 2  int 0x13  jnc @f  xor ah, ah  int 0x13  dec si  jnz @b  .error:  call error  db "DISK ERROR",13,10,0  @:  pop si cx bx dx  ret  .use\_EDD:  push si  mov byte[0x600], 0x10  mov byte[0x601], 0  mov word[0x602], 1  mov [0x604], di  push es  pop word[0x606]  mov [0x608], ax  mov [0x60A], dx  mov word[0x60C], 0  mov word[0x60E], 0  mov ah, 0x42  mov dl, [disk\_id]  mov si, 0x600  int 0x13  jc .error  pop si dx  ret  ; Поиск файла с именем DS:SI в каталоге DX:AX  find\_file:  push cx dx di  .find:  cmp ax, -1  jne @f  cmp dx, -1  jne @f  .not\_found:  call error  db "NOT FOUND",13,10,0  @:  mov di, f\_info  call load\_sector  push di  mov cx, 0xFFFF  xor al, al  repne scasb  neg cx  dec cx  pop di  push si  repe cmpsb  pop si  je .found  mov ax, word[f\_next]  mov dx, word[f\_next + 2]  jmp .find  .found:  pop di dx cx  ret  ; Загрузка текущего файла в память по адресу BX:0. Количество загруженных секторов возвращается в AX  load\_file\_data:  push bx cx dx si di  mov ax, word[f\_data]  mov dx, word[f\_data + 2]  .load\_list:  cmp ax, -1  jne @f  cmp dx, -1  jne @f  .file\_end:  pop di si dx cx  mov ax, bx  pop bx  sub ax, bx  shr ax, 9 - 4  ret  @:  mov di, 0x8000 / 16  call load\_sector  mov si, di  mov cx, 512 / 8 - 1  .load\_sector:  lodsw  mov dx, [si]  add si, 6  cmp ax, -1  jne @f  cmp dx, -1  je .file\_end  @:  push es  mov es, bx  xor di, di  call load\_sector  add bx, 0x200 / 16  pop es  loop .load\_sector  lodsw  mov dx, [si]  jmp .load\_list  ; Точка входа в начальный загрузчик  boot:  ; Настроим сегментные регистры  jmp 0:@f  @:  mov ax, cs  mov ds, ax  mov es, ax  ; Настроим стек  mov ss, ax  mov sp, $$  ; Разрешим прерывания  sti  ; Запомним номер загрузочного диска  mov [disk\_id], dl  ; Определим параметры загрузочного диска  mov ah, 0x41  mov bx, 0x55AA  int 0x13  jc @f  mov byte[sector\_per\_track], 0xFF  jmp .disk\_detected  @:  mov ah, 0x08  xor di, di  push es  int 0x13  pop es  jc load\_sector.error  inc dh  mov [head\_count], dh  and cx, 111111b  mov [sector\_per\_track], cx  .disk\_detected:  ; Загрузим продолжение начального загрузчика  mov si, boot\_file\_name  mov ax, word[fs\_first\_file]  mov dx, word[fs\_first\_file + 2]  call find\_file  mov bx, 0x7E00 / 16  call load\_file\_data  ; Переходим на продолжение  jmp boot2  ; Пустое пространство и сигнатура  rb 510 - ($ - $$)  db 0x55,0xAA  ; Дополнительные данные загрузчика  load\_msg\_preffix db "Loading '",0  load\_msg\_suffix db "'...",0  ok\_msg db "OK",13,10,0  config\_file\_name db "boot.cfg",0  start16\_msg db "Starting 16 bit kernel...",13,10,0  start32\_msg db "Starting 32 bit kernel...",13,10,0  ; Разбиение строки DS:SI по символу слеша  split\_file\_name:  push si  @:  lodsb  cmp al, "/"  je @f  test al, al  jz @f  jmp @b  @:  mov byte[si - 1], 0  mov ax, si  pop si  ret  ; Загрузка файла с именем DS:SI в буфер BX:0. Размер файла в секторах возвращается в AX  load\_file:  push si  mov si, load\_msg\_preffix  call write\_str  pop si  call write\_str  push si  mov si, load\_msg\_suffix  call write\_str  pop si  push si bp  mov dx, word[fs\_first\_file + 2]  mov ax, word[fs\_first\_file]  @:  push ax  call split\_file\_name  mov bp, ax  pop ax  call find\_file  test byte[f\_flags], 1  jz @f  mov si, bp  mov dx, word[f\_data + 2]  mov ax, word[f\_data]  jmp @b  @:  call load\_file\_data  mov si, ok\_msg  call write\_str  pop bp si  ret  ; Продолжение начального загрузчика  boot2:  ; Загрузим конфигурационный файл загрузчика  mov si, config\_file\_name  mov bx, 0x1000 / 16  call load\_file  ; Выполним загрузочный скрипт  mov bx, 0x9000 / 16  mov bp, 0x6000  mov dx, 0x1000  .parse\_line:  mov si, dx  .parse\_char:  lodsb  test al, al  jz .config\_end  cmp al, 10  je .run\_command  cmp al, 13  je .run\_command  jmp .parse\_char  .run\_command:  mov byte[si - 1], 0  xchg dx, si  cmp byte[si], 0  je .parse\_line ; Пустая строка  cmp byte[si], "#"  je .parse\_line ; Комментарий  cmp byte[si], "L"  je .load\_file ; Загрузка файла  cmp byte[si], "S"  je .start ; Запуск ядра  ; Неизвестная команда  mov al, [si]  mov [.cmd], al  call error  db "Unknown boot script command '"  .cmd db ?  db "'!",13,10,0  .config\_end: ; При правильном конфигурационном файле мы не должны сюда попасть  ; Завершение  jmp reboot  ; Загрузка файла  .load\_file:  push dx  inc si  call load\_file  push ax  mov cx, 512  mul cx  mov word[bp + 8], ax  mov word[bp + 10], dx  mov word[bp + 12], 0  mov word[bp + 14], 0  mov ax, bx  mov cx, 16  mul cx  mov word[bp], ax  mov word[bp + 2], dx  mov word[bp + 4], 0  mov word[bp + 6], 0  pop ax  shr ax, 9 - 4  add bx, ax  add bp, 16  pop dx  jmp .parse\_line  ; Запуск ядра  .start:  ; Проверим, что загружен хотя бы один файл  cmp bx, 0x9000 / 16  ja @f  call error  db "NO KERNEL LOADED",13,10,0  @:  ; Заполняем последний элемент списка файлов  xor ax, ax  mov cx, 16  mov di, bp  rep stosw  ; Переходим к процедуре инициализации ядра для нужной разрядности  inc si  cmp word[si], "16"  je .start16  cmp word[si], "32"  je .start32  ;cmp word[si], "64"  ;je start64  ; Неизвестная рязрядность ядра  call error  db "Invalid start command argument",13,10,0  ; Запуск 16-разрядного ядра  .start16:  mov si, start16\_msg  mov bx, 0x6000  mov dl, [disk\_id]  jmp 0x9000  ; Запуск 32-разрядного ядра  .start32:  ; Выводим уведомление о запуске 32-битного ядра  mov si, start32\_msg  call write\_str  ; Проверим, что процессор не хуже i386  mov ax, 0x7202  push ax  popf  pushf  pop bx  cmp ax, bx  je @f  call error  db "Required i386 or better",13,10,0  @:  ; Загрузим значение в GDTR  lgdt [gdtr32]  ; Запретим прерывания  cli  ; Перейдём в защищённый режим  mov eax, cr0  or eax, 1  mov cr0, eax  ; Перейдём на 32-битный код  jmp 8:start32  ; Таблица дескрипторов сегментов для 32-битного ядра  align 16  gdt32:  dq 0 ; NULL - 0  dq 0x00CF9A000000FFFF ; CODE - 8  dq 0x00CF92000000FFFF ; DATA - 16  gdtr32:  dw $ - gdt32 - 1  dd gdt32  ; 32-битный код  use32  start32:  ; Настроим сегментные регистры и стек  mov eax, 16  mov ds, ax  mov es, ax  mov fs, ax  mov gs, ax  mov ss, ax  movzx esp, sp  ; Выводим символ на экран  mov byte[0xB8000 + (25 \* 80 - 1) \* 2], "!"  ; Завершение  jmp $  В следующем выпуске мы поговорим про страничную адресацию и наконец-то загрузим хоть какое-то ядро. До встречи! :-) |

Пишем свою операционную систему. Страничная адресация

|  |
| --- |
| Доброго времени суток!  В прошлом выпуске мы остановились на том, что перешли в защищённый режим и вывели один символ на экран. Сейчас пришло время рассказать об одной вещи, которая является, пожалуй, ключевым элементом в построении многозадачности и вообще сильно влияет на архитектуру любой полноценной операционной системы. Это - страничная трансляция адресов.  Для начала познакомимся с тремя видами адреса - **физический адрес**, **линейный адрес** и **виртуальный адрес**. Во-первых, физический адрес - это непосредственно адрес в оперативной памяти. Именно он будет послан контроллеру памяти при обращении к переменной, про остальные адреса знает только процессор. В реальном режиме все обращения производятся по физическим адресам. То есть, если мы обратились по адресу 1000:FFFF, то мы будем читать или писать байт в оперативной памяти именно с адресом 0x1FFFF и никакой другой. В самом простом случае физическое адресное пространство более-менее непрерывно - сначала первый непрерывный блок в 640 КБ, потом 384 КБ памяти устройств (память видео-карты для текстового режима, BIOS), потом ещё один непрерывный блок адресов оперативной памяти. Затем идёт "дыра" несуществующей памяти (запись данных туда игнорируется, а чтение возвращает 0x00 или 0xFF), в которой встречаются блоки памяти других устройств (например, память видео-карты для графического режима). Если оперативной памяти достаточно много (более 3 ГБ), то она уже не будет образовывать непрерывный блок - адрес-то 32-битный (значит можно адресовать лишь 4 ГБ), а устройства тоже хотят адресов. Поэтому "лишняя" память будет размещена за пределом 4 ГБ и доступна лишь для ОС с поддержкой 64 бит или PAE. Положение участков оперативной памяти жёстко задано и не меняется ОС.  Второй тип адресов - линейный. Без использования страничной адресации он полностью совпадает с физическим, но она всё меняет. Любой адрес в линейом адресном пространстве может на самом деле ссылаться на любой физический адрес. То есть мы можем как бы сказать процессору "адреса с 0 до 0x100000 должны является проекцией адресов 0x2000000 - 0x2100000". В это виртуальное адресное пространство можно спроецировать любое количество блоков физической памяти, в том числе и один и тот же несколько раз. Объём виртуальной памяти может быть меньше физической (не все физические адреса спроецированы в линейные), а может быть и больше (некоторые адреса спроецированы более одно раза). Зачем это нужно? Каждый процесс получает своё личное, независимое адресное пространство. Переключение между процессами осуществляется заменой таблиц преобразования. Таким образом достигается две хороших вещи:  1) Каждый процесс формирует своё адресное пространство исходя из своих предпочтений. По удобному для него адресу будет располагаться то, что ему нужно, несмотря на то, что другой процесс разметил в своём адресном пространстве совсем другие данные. Ну и наконец все структуры можно разместить по одним и тем же линейным адресам вне зависимости от объёма и расположения блоков физической памяти.  2) Каждый процесс изолирован от других и существует в своей своеобразной песочнице. Он не может случайно или умышленно изменить данные чужого процесса. Всё взаимодействие лишь с помощью сервисов ядра (которое обычно во всех адресных пространствах спроецировано по одним и тем же адресам, недоступным для приложения).  Именно этот механизм позволяет строить эффективные и защищённые многозадачные среды. Также страницы памяти (переадресовать каждый байт было бы слишком накладно, поэтому вся физическая и виртуальная память поделена на блоки по 4 КБ - **страницы**. Переадресация осуществляется на уровне целых страниц вроде - сотая страница виртуальной памяти является проекцией десятой физической) имеют свои атрибуты доступа. Например, страницы с кодом можно защитить от записи, а страницы с данными и кодом ядра - от доступа со стороны приложения.  Страничная адресация нужна и для создания механизма подкачки страница или свопа. При обращении к неспроецированной странице памяти происходит ошибка, которая может быть обработана ядром системы. Оно совсем не обязательно при любой такой ошибке должно прибить приложение - возможна ситуация, когда эта страница была выгружена в файл подкачки, в таком случае она загружается обратно, а приложение даже не замечает и продолжает работу, как будто данные всегда были на месте.  Ну и наконец виртуальный адрес. Он связан с использованием сегментации. Если базовый адрес сегмента не равен нулю, то он будет прибавлен к виртуальному адресу, таким образом получается линейный, который по таблице преобразований превращается в физический. Именно этот адрес указывается во всех ассемблерных командах. Если сегментация не используется, то он равен линейному.  **Виртуальный адрес** (аргумент ассемблерной команды) -> Прибавление базы сегмента, проверка границы и прав доступа к сегменту -> **Линейный адрес** -> Преобразование по таблицам страниц и проверка прав доступа к страницам -> **Физический адрес** (передаётся контроллеру памяти).  Страничное преобразование осуществляется с помощью специальных таблиц страниц. Каждый элемент таблицы страниц занимает 32 или 64 бита в зависимости от разрядности системы. Младшие 12 бит содержат атрибуты страницы, старшие - физический адрес, на который эта страница должна быть переадресована. При этом в атрибутах может быть указано, что страница не существует, в таком случае при попытке обращения произойдёт исключение, зато и заполнять физический адрес не надо.  Если бы таблица страниц являлась простым линейным массивом, то она заняла бы в памяти целых 4 мегабайта. Это не очень приятно для 32-битной системы. Ну а в 64-битной (даже с учётом того, что значащими являются только 48 бит виртуального адреса) системе, такая таблица страниц была бы уже на 64 ГБ, что часто превышает весь объём оперативной памяти компьютера. Эта проблема решается достаточно просто - совсем не все страницы должны быть спроецированы. Даже как правило бОльшая их часть не нужна. Многие приложения занимают в памяти лишь несколько десятков мегабайт, зачем хранить информацию о том, что кучи страниц просто?  Для этого были придуманы многоуровневые таблицы страничной трансляции. 2-х уровневая для 32 бит и 4-х уровневая для 64 бит. Рассмотрим 32-битную.  Первый уровень называется каталогом страниц. Именно его адрес мы сообщаем процессору, когда настраиваем страничную адресацию. Каталог таблиц (как, впрочем, и таблицы) занимает одну страницу памяти. Соответственно он содержит 1024 элемента (4096 / 4 = 1024). Старшие 10 бит адреса являются индексом в каталоге страниц. Полученный элемент является физическим адресом таблицы. А уже следующие 10 бит адреса являются номером страницы в этой таблице. Каждый элемент каталога имеет те же 12 бит атрибутов, поэтому любая таблица тоже может отсутствовать. Младшие 12 бит адреса используются как смещение на странице.  В 64-битном режиме в каталоге и каждой таблице не 1024, а 512 элементов (ведь они в два раза больше, а размер страницы тот же), а выше каталога страниц есть ещё каталог каталогов страниц и каталог страниц 4-ого уровня. При этом на любом уровне таблица может отсутствовать и тогда процессор не пойдёт дальше вглубь, а сразу кинет исключение.  Возможно, эта тема не самая простая для понимания, дальше, пожалуй, будет проще. Если возникнут вопросы - вы можете задавать их мне по электронной почте. Основную информации я постарался рассказать как мог.  Итак, произведём первичное планирование адресного пространства нашей ОС. Я думаю разделить его пополам. Нижние 2 ГБ (0x00000000 - 0x7FFFFFFF) у каждого процесса свои и полностью им настраиваются. Верхние 2 ГБ (0x80000000 - 0xFFFFFFFF) - принадлежат системе и общие у всех процессов. Там находится код ядра, данные для межпроцессного взаимодействия и т. д. Сам код ядра и его самые важные данные - последние 4 МБ 0xFFC00000 - 0xFFFFFFFF (последние 2 для 64-битной версии), потому что ровно столько памяти переадресует одна таблица страниц самого нижнего уровня.  Базовый физический адрес каталога страниц хранится в регистре CR3. И таблицы, и каталог выровнены на размер страницы.  Мы подготовим таблицы страниц и загрузим адрес каталога в CR3 в реальном режиме. Переход в защищённый режим мы выполним одновременно с включением страничной адресации (бит 31 регистра CR0). Главное позаботится о том, чтобы первый мегабайт был тоже примонтирован, причём в соответствии с физическими адресами, чтобы наш код продолжил правильно выполняться до прыжка на ядро.  Нам надо будет создать две таблицы страниц - первую (чтобы примонтировать первый мегабайт адрес-в-адрес) и последнюю (для ядра). Помимо проекции загруженного файла ядра, в этой таблице страниц будет ещё стек и самое главное - сама таблица. Зачем это нужно? сейчас объясню.  Страничная адресация полностью скрывает от нас физическую память. Чтобы получить доступ к новым адресам необходимо изменить таблицу (а если нужной таблицы пока нет, то и каталог) страниц, но ведь страницы тоже находятся в физической памяти. Выхода два:  1) Проецировать все таблицы всех уровней в виртуальную память. Достаточно неудобно и запутанно. Придётся отрезать кусок виртуальной памяти, следить за примонтированностью нужных данных.  2) Сделать специальную временную страницу, которую мы сможем монтировать когда нам захочется. В моём случае это реализуется за счёт того факта, что последняя таблица страниц спроецирована. Мы меняем в ней последний элемент и в итоге проецируем любой нужный физический адрес по виртуальному адресу 0xFFFFF000. Таким образом мы имеем "окно", через которое смотрим на непримонтированную физическую память. На самом деле реализовать функции монтирования при такой системе очень просто, хотя может показаться труднее. В этом мы убедимся позднее.  Итак, пришло время воплотить всё это в коде:  ; Запуск 32-разрядного ядра  .start32:  ; Выводим уведомление о запуске 32-битного ядра  mov si, start32\_msg  call write\_str  ; Проверим, что процессор не хуже i386  mov ax, 0x7202  push ax  popf  pushf  pop bx  cmp ax, bx  je @f  call error  db "Required i386 or better",13,10,0  @:  ; Очистим таблицы страниц  xor ax, ax  mov cx, 3 \* 4096 / 2  mov di, 0x1000  rep stosw  ; Заполним каталог страниц  mov word[0x1000], 0x2000 + 111b  mov word[0x1FFC], 0x3000 + 111b  ; Заполним первую таблицу страниц  mov eax, 11b  mov cx, 0x100000 / 4096  mov di, 0x2000  @:  stosd  add eax, 0x1000  loop @b  ; Заполним последнюю таблицу страниц  mov di, 0x3000  mov eax, dword[0x6000]  or eax, 11b  mov ecx, dword[0x6008]  shr ecx, 12  @:  stosd  add eax, 0x1000  loop @b  mov word[0x3FF4], 0x4000 + 11b ; Kernel stack ;dword  mov word[0x3FF8], 0x3000 + 11b ; Kernel page table ;dword  ; Загрузим значение в CR3  mov eax, 0x1000  mov cr3, eax  ; Загрузим значение в GDTR  lgdt [gdtr32]  ; Запретим прерывания  cli  ; Перейдём в защищённый режим  mov eax, cr0  or eax, 0x80000001  mov cr0, eax  ; Перейдём на 32-битный код  jmp 8:start32  ; Таблица дескрипторов сегментов для 32-битного ядра  align 16  gdt32:  dq 0 ; NULL - 0  dq 0x00CF9A000000FFFF ; CODE - 8  dq 0x00CF92000000FFFF ; DATA - 16  gdtr32:  dw $ - gdt32 - 1  dd gdt32  ; 32-битный код  use32  start32:  ; Настроим сегментные регистры и стек  mov eax, 16  mov ds, ax  mov es, ax  mov fs, ax  mov gs, ax  mov ss, ax  mov esp, 0xFFFFDFFC  ; Выводим символы на экран  mov byte[0xB8000 + (25 \* 80 - 1) \* 2], "K"  mov dword[0xFFFFEFFC], 0xB8000 + 11b  mov byte[0xFFFFF000 + (25 \* 80 - 2) \* 2], "O"  ; Завершение  jmp $   Последние строки демонстрируют возможности страничной адресации - символ "O" мы выводим через проекцию страницы 0xB8000 на адрес 0xFFFFF000.  Первый загруженный файл примонтирован по адресу 0xFFC00000. Если там есть корректный 32-битный код, то можно смело заменять команду jmp $ на jmp 0xFFC00000 и наш загрузчик будет запускать ядро. Как производить временное монтирование страниц показано в команде между выводом "K" и "O" (в итоге получится "OK" :-) ).  Ну вот и всё на сегодня... В следующем выпуске мы наконец-то попробуем написать код ядра на Си. Это, наверное, самый запутанный выпуск, поэтому не стесняйтесь задавать вопросы. Дальше будет уже попроще, пожалуй. |

## Пишем свою операционную систему. Переходим на Си

|  |
| --- |
| Приветствую всех своих читателей!  Предыдущие выпуски могли быть несколько запутанными. Начальная загрузка, Assembler, BIOS. Сегодня мы наконец переходим к более интересной и понятной части - мы начинаем писать ядро. И писать мы его будем на языке высокого уровня Си.  В начальный загрузчик осталось внести всего пару дополнений и он будет полностью готов грузить любые 32-битные ядра. Определение объёма оперативной памяти Конечно, можно подсчитать объём памяти вручную в ядре - перебирать адреса от 0x100000 и пытаться записать туда значение отличное от нуля и 0xFF. Если при чтении мы получаем полученное значение, то всё хорошо, иначе память кончилась - запоминаем адрес последнего удачного чтения, это и будет объёмом оперативной памяти. Однако такой способ имеет два недостатка:  1) Его следует использовать до включения страничной адресации, чтобы иметь доступ ко всей физической памяти, либо устраивать запись через "окно" временной страницы. Лишняя трата времени, при условии, что тестирование памяти BIOS и так выполняет при начальной инициализации, а мы делаем двойную работу.  2) Всё хорошо пока память представляет собой непрерывный участок адресов, но на современных системах с большим объёмом памяти это правило может быть нарушено. К тому же BIOS пишет в самую обычную память таблицы ACPI, которые пригодятся операционной системе и не стоит их затирать до прочтения.  Из этого следует, что лучше спросить про объём оперативной памяти у BIOS, благо он предоставляет все необходимые функции.  Исторически первой функцией определения объёма оперативной памяти было прерывание 0x12. Оно не принимает никаких входных параметров, в на выходе в регистре AX содержится размер базовой памяти в килобайтах. Базовая память - те самые 640 КБ доступные в реальном режиме. Сейчас вы уже не сможете найти компьютер, где бы было менее 640 КБ памяти, но мало ли. Использовать её нам смысла нет - если процессор поддерживает защищённый режим, то вряд ли у него будет меньше нескольких мегабайт памяти.  Объёмы памяти росли и 640 КБ стало мало. Тогда появилась новая функция - прерывание 0x15 AH=0x88. Она возвращает в AX размер расширенной памяти (свыше 1 МБ) в килобайтах в AX. Эта функция не может возвращать значения больше 15 МБ (15 + 1 итого 16 МБ).  Когда и 16 МБ стало недостаточно появилась новая функция - прерывание 0x15, AX=0xE801. Она возвращает результаты аж в 4 регистрах:  AX - размер расширенной памяти до 16 МБ в килобайтах BX - размер расширенной памяти сверх 16 МБ к блоках по 64 КБ CX - размер сконфигурированной расширенной памяти до 16 МБ в килобайтах DX - размер сконфигурированной расширенной памяти сверх 16 МБ в блоках по 64 КБ  Что такое "сконфигурированная" память производители BIOS судя по всему не договорились, поэтому надо просто, если в AX и BX нули, брать значение из CX и DX.  Но и этого оказалось мало. Ведь все перечисленные выше функции имеют ограничение объёма памяти в 4 ГБ, к тому же не учитывают то, что память может быть не непрерывным блоком. Поэтому в новых BIOS появилась ещё одна функция - прерывание 0x15, AX=0xE820. Она возвращает не просто число, а карту памяти. Входные параметры:  EAX=0xE820 EDX=0x534D4150 ("SMAP") EBX - смещение от начала карты памяти (для начала 0) ECX - размер буфера (как правило 24 байта - размер одного элемента) ES:DI - адрес буфера, куда надо записать очередной элемент  Выходные параметры:  EAX=0x534D4150 ("SMAP") EBX - новое смещение для следующего вызова функции. Если 0, то вся карта памяти прочитана ECX - количество реально возвращённых байт (20 или 24 байта) В указанном буфере содержится очередной элемент карты памяти.  Каждый элемент карты памяти имеет следующую структуру (напишу в синтаксисе Си, потому что разбор данных мы будем делать уже в ядре):  struct {  unsigned long long base; //Базовый физический адрес региона  unsigned long long length; //Размер региона в байтах  unsigned long type; // Тип региона  unsigned long acpi\_attrs; //Расширенные атрибуты ACPI  };  Последний элемент структуры не обязателен. Ещё в одном источнике видел, что перед запросом элемента стоит поместить туда единичку. Конечно, сейчас мы не поддерживаем ACPI, но лучше заранее позаботится о том, чтобы получить как можно больше данных. В отличии от параметров памяти, всё остальное можно легко узнать и из защищённого режима напрямую, без BIOS.  Регионы памяти, описываемые картой, могут быть нескольких типов:  1 - Обычная память. Может быть свободно использована ОС для своих целей. Пока мы только к ней и будем обращаться, а всё остальное пропускать. 2 - Зарезервировано (например, код BIOS). Эта память может быть как физически недоступна для записи, так и просто запись туда нежелательна. Такую память лучше не трогать. 3 - Доступно после прочтения таблиц ACPI. Вероятно, именно в этих блоках эти таблицы и хранятся. Пока драйвер ACPI не прочитает таблицы, эту память лучше не трогать. Потом можно использовать так же, как и память типа 1. 4 - Эту память следует сохранять между NVS сессиями. Такую память мы трогать не будем, пока не узнаем, что такое NVS сессии :-)  Не все BIOS могут поддерживать эту функцию. Если какая-то функция не поддерживается, то при выходе из неё установлен флаг переполнения и следует обращаться к более старой. Мы будем использовать формат карты памяти функции 0xE820. Если саму эту функцию вызвать не получилось - получать объём памяти обычными средствами и создавать свою собственную карту памяти из одного элемента. Поскольку определение объёма памяти задача нужная и для запуска 32-битного и для запуска 64-битного ядра, лучше оформить её в виде подпрограммы. Карту памяти разместим по адресу 0x7000. Не думаю, что она может быть больше пары килобайт. Последний элемент вручную сделаем типа 0 - такого типа не возвращает BIOS и это и будет признаком конца.  ; Получение карты памяти  get\_memory\_map:  mov di, 0x7000  xor ebx, ebx  @:  mov eax, 0xE820  mov edx, 0x534D4150  mov ecx, 24  mov dword[di + 20], 1  int 0x15  jc @f  add di, 24  test ebx, ebx  jnz @b  @:  cmp di, 0x7000  ja .ok  mov dword[di], 0x100000  mov dword[di + 4], 0  mov dword[di + 12], 0  mov dword[di + 16], 1  mov dword[di + 20], 0  mov ax, 0xE801  int 0x15  jnc @f  mov ah, 0x88  int 0x15  jc .ok  mov cx, ax  xor dx, dx  @:  test cx, cx  jz @f  mov ax, cx  mov bx, dx  @:  movzx eax, ax  movzx ebx, bx  mov ecx, 1024  mul ecx  push eax  mov eax, ebx  mov ecx, 65536  mul ecx  pop edx  add eax, edx  mov [di + 8], eax  add di, 24  jmp .ok  .ok:  xor ax, ax  mov cx, 24 / 2  rep stosw  ret   Ну вот и готов наш начальный загрузчик для 32-битных ядер. В заключение привожу его полный код и мы перейдём к ядру.  ; Начальный загрузчик ядра для архитектуры x86  format Binary as "bin"  org 0x7C00  jmp boot  ; Заголовок ListFS  align 4  fs\_magic dd ?  fs\_version dd ?  fs\_flags dd ?  fs\_base dq ?  fs\_size dq ?  fs\_map\_base dq ?  fs\_map\_size dq ?  fs\_first\_file dq ?  fs\_uid dq ?  fs\_block\_size dd ?  ; Заголовок файла  virtual at 0x800  f\_info:  f\_name rb 256  f\_next dq ?  f\_prev dq ?  f\_parent dq ?  f\_flags dq ?  f\_data dq ?  f\_size dq ?  f\_ctime dq ?  f\_mtime dq ?  f\_atime dq ?  end virtual  ; Данные начального загрузчика  label sector\_per\_track word at $$  label head\_count byte at $$ + 2  label disk\_id byte at $$ + 3  reboot\_msg db "Press any key...",13,10,0  boot\_file\_name db "boot.bin",0  ; Вывод строки DS:SI на экран  write\_str:  push si  mov ah, 0x0E  @:  lodsb  test al, al  jz @f  int 0x10  jmp @b  @:  pop si  ret  ; Критическая ошибка  error:  pop si  call write\_str  ; Перезагрузка  reboot:  mov si, reboot\_msg  call write\_str  xor ah, ah  int 0x16  jmp 0xFFFF:0  ; Загрузка сектора DX:AX в буфер ES:DI  load\_sector:  push dx  add ax, word[fs\_base]  adc dx, word[fs\_base + 2]  cmp byte[sector\_per\_track], 0xFF  je .use\_EDD  push bx cx si  div [sector\_per\_track]  mov cl, dl  inc cl  div [head\_count]  mov dh, ah  mov ch, al  mov dl, [disk\_id]  mov bx, di  mov al, 1  mov si, 3  @:  mov ah, 2  int 0x13  jnc @f  xor ah, ah  int 0x13  dec si  jnz @b  .error:  call error  db "DISK ERROR",13,10,0  @:  pop si cx bx dx  ret  .use\_EDD:  push si  mov byte[0x600], 0x10  mov byte[0x601], 0  mov word[0x602], 1  mov [0x604], di  push es  pop word[0x606]  mov [0x608], ax  mov [0x60A], dx  mov word[0x60C], 0  mov word[0x60E], 0  mov ah, 0x42  mov dl, [disk\_id]  mov si, 0x600  int 0x13  jc .error  pop si dx  ret  ; Поиск файла с именем DS:SI в каталоге DX:AX  find\_file:  push cx dx di  .find:  cmp ax, -1  jne @f  cmp dx, -1  jne @f  .not\_found:  call error  db "NOT FOUND",13,10,0  @:  mov di, f\_info  call load\_sector  push di  mov cx, 0xFFFF  xor al, al  repne scasb  neg cx  dec cx  pop di  push si  repe cmpsb  pop si  je .found  mov ax, word[f\_next]  mov dx, word[f\_next + 2]  jmp .find  .found:  pop di dx cx  ret  ; Загрузка текущего файла в память по адресу BX:0. Количество загруженных секторов возвращается в AX  load\_file\_data:  push bx cx dx si di  mov ax, word[f\_data]  mov dx, word[f\_data + 2]  .load\_list:  cmp ax, -1  jne @f  cmp dx, -1  jne @f  .file\_end:  pop di si dx cx  mov ax, bx  pop bx  sub ax, bx  shr ax, 9 - 4  ret  @:  mov di, 0x8000 / 16  call load\_sector  mov si, di  mov cx, 512 / 8 - 1  .load\_sector:  lodsw  mov dx, [si]  add si, 6  cmp ax, -1  jne @f  cmp dx, -1  je .file\_end  @:  push es  mov es, bx  xor di, di  call load\_sector  add bx, 0x200 / 16  pop es  loop .load\_sector  lodsw  mov dx, [si]  jmp .load\_list  ; Точка входа в начальный загрузчик  boot:  ; Настроим сегментные регистры  jmp 0:@f  @:  mov ax, cs  mov ds, ax  mov es, ax  ; Настроим стек  mov ss, ax  mov sp, $$  ; Разрешим прерывания  sti  ; Запомним номер загрузочного диска  mov [disk\_id], dl  ; Определим параметры загрузочного диска  mov ah, 0x41  mov bx, 0x55AA  int 0x13  jc @f  mov byte[sector\_per\_track], 0xFF  jmp .disk\_detected  @:  mov ah, 0x08  xor di, di  push es  int 0x13  pop es  jc load\_sector.error  inc dh  mov [head\_count], dh  and cx, 111111b  mov [sector\_per\_track], cx  .disk\_detected:  ; Загрузим продолжение начального загрузчика  mov si, boot\_file\_name  mov ax, word[fs\_first\_file]  mov dx, word[fs\_first\_file + 2]  call find\_file  mov bx, 0x7E00 / 16  call load\_file\_data  ; Переходим на продолжение  jmp boot2  ; Пустое пространство и сигнатура  rb 510 - ($ - $$)  db 0x55,0xAA  ; Дополнительные данные загрузчика  load\_msg\_preffix db "Loading '",0  load\_msg\_suffix db "'...",0  ok\_msg db "OK",13,10,0  config\_file\_name db "boot.cfg",0  start16\_msg db "Starting 16 bit kernel...",13,10,0  start32\_msg db "Starting 32 bit kernel...",13,10,0  label module\_list at 0x6000  label memory\_map at 0x7000  ; Разбиение строки DS:SI по символу слеша  split\_file\_name:  push si  @:  lodsb  cmp al, "/"  je @f  test al, al  jz @f  jmp @b  @:  mov byte[si - 1], 0  mov ax, si  pop si  ret  ; Загрузка файла с именем DS:SI в буфер BX:0. Размер файла в секторах возвращается в AX  load\_file:  push si  mov si, load\_msg\_preffix  call write\_str  pop si  call write\_str  push si  mov si, load\_msg\_suffix  call write\_str  pop si  push si bp  mov dx, word[fs\_first\_file + 2]  mov ax, word[fs\_first\_file]  @:  push ax  call split\_file\_name  mov bp, ax  pop ax  call find\_file  test byte[f\_flags], 1  jz @f  mov si, bp  mov dx, word[f\_data + 2]  mov ax, word[f\_data]  jmp @b  @:  call load\_file\_data  mov si, ok\_msg  call write\_str  pop bp si  ret  ; Получение карты памяти  get\_memory\_map:  mov di, memory\_map  xor ebx, ebx  @:  mov eax, 0xE820  mov edx, 0x534D4150  mov ecx, 24  mov dword[di + 20], 1  int 0x15  jc @f  add di, 24  test ebx, ebx  jnz @b  @:  cmp di, 0x7000  ja .ok  mov dword[di], 0x100000  mov dword[di + 4], 0  mov dword[di + 12], 0  mov dword[di + 16], 1  mov dword[di + 20], 0  mov ax, 0xE801  int 0x15  jnc @f  mov ah, 0x88  int 0x15  jc .ok  mov cx, ax  xor dx, dx  @:  test cx, cx  jz @f  mov ax, cx  mov bx, dx  @:  movzx eax, ax  movzx ebx, bx  mov ecx, 1024  mul ecx  push eax  mov eax, ebx  mov ecx, 65536  mul ecx  pop edx  add eax, edx  mov [di + 8], eax  add di, 24  jmp .ok  .ok:  xor ax, ax  mov cx, 24 / 2  rep stosw  ret  ; Продолжение начального загрузчика  boot2:  ; Загрузим конфигурационный файл загрузчика  mov si, config\_file\_name  mov bx, 0x1000 / 16  call load\_file  ; Выполним загрузочный скрипт  mov bx, 0x9000 / 16  mov bp, module\_list  mov dx, 0x1000  .parse\_line:  mov si, dx  .parse\_char:  lodsb  test al, al  jz .config\_end  cmp al, 10  je .run\_command  cmp al, 13  je .run\_command  jmp .parse\_char  .run\_command:  mov byte[si - 1], 0  xchg dx, si  cmp byte[si], 0  je .parse\_line ; Пустая строка  cmp byte[si], "#"  je .parse\_line ; Комментарий  cmp byte[si], "L"  je .load\_file ; Загрузка файла  cmp byte[si], "S"  je .start ; Запуск ядра  ; Неизвестная команда  mov al, [si]  mov [.cmd], al  call error  db "Unknown boot script command '"  .cmd db ?  db "'!",13,10,0  .config\_end: ; При правильном конфигурационном файле мы не должны сюда попасть  ; Завершение  jmp reboot  ; Загрузка файла  .load\_file:  push dx  inc si  call load\_file  push ax  mov cx, 512  mul cx  mov word[bp + 8], ax  mov word[bp + 10], dx  mov word[bp + 12], 0  mov word[bp + 14], 0  mov ax, bx  mov cx, 16  mul cx  mov word[bp], ax  mov word[bp + 2], dx  mov word[bp + 4], 0  mov word[bp + 6], 0  pop ax  shr ax, 9 - 4  add bx, ax  add bp, 16  pop dx  jmp .parse\_line  ; Запуск ядра  .start:  ; Проверим, что загружен хотя бы один файл  cmp bx, 0x9000 / 16  ja @f  call error  db "NO KERNEL LOADED",13,10,0  @:  ; Заполняем последний элемент списка файлов  xor ax, ax  mov cx, 16  mov di, bp  rep stosw  ; Переходим к процедуре инициализации ядра для нужной разрядности  inc si  cmp word[si], "16"  je .start16  cmp word[si], "32"  je .start32  ;cmp word[si], "64"  ;je .start64  ; Неизвестная рязрядность ядра  call error  db "Invalid start command argument",13,10,0  ; Запуск 16-разрядного ядра  .start16:  mov si, start16\_msg  mov bx, module\_list  mov dl, [disk\_id]  jmp 0x9000  ; Запуск 32-разрядного ядра  .start32:  ; Выводим уведомление о запуске 32-битного ядра  mov si, start32\_msg  call write\_str  ; Проверим, что процессор не хуже i386  mov ax, 0x7202  push ax  popf  pushf  pop bx  cmp ax, bx  je @f  call error  db "Required i386 or better",13,10,0  @:  ; Получим карту памяти  call get\_memory\_map  ; Очистим таблицы страниц  xor ax, ax  mov cx, 3 \* 4096 / 2  mov di, 0x1000  rep stosw  ; Заполним каталог страниц  mov word[0x1000], 0x2000 + 111b  mov word[0x1FFC], 0x3000 + 111b  ; Заполним первую таблицу страниц  mov eax, 11b  mov cx, 0x100000 / 4096  mov di, 0x2000  @:  stosd  add eax, 0x1000  loop @b  ; Заполним последнюю таблицу страниц  mov di, 0x3000  mov eax, dword[module\_list]  or eax, 11b  mov ecx, dword[module\_list + 8]  shr ecx, 12  @:  stosd  add eax, 0x1000  loop @b  mov word[0x3FF4], 0x4000 + 11b ; Kernel stack  mov word[0x3FF8], 0x3000 + 11b ; Kernel page table  ; Загрузим значение в CR3  mov eax, 0x1000  mov cr3, eax  ; Загрузим значение в GDTR  lgdt [gdtr32]  ; Запретим прерывания  cli  ; Перейдём в защищённый режим  mov eax, cr0  or eax, 0x80000001  mov cr0, eax  ; Перейдём на 32-битный код  jmp 8:start32  ; Таблица дескрипторов сегментов для 32-битного ядра  align 16  gdt32:  dq 0 ; NULL - 0  dq 0x00CF9A000000FFFF ; CODE - 8  dq 0x00CF92000000FFFF ; DATA - 16  gdtr32:  dw $ - gdt32 - 1  dd gdt32  ; 32-битный код  use32  start32:  ; Настроим сегментные регистры и стек  mov eax, 16  mov ds, ax  mov es, ax  mov fs, ax  mov gs, ax  mov ss, ax  mov esp, 0xFFFFDFFC  ; Поместим в DL номер загрузочного диска  mov dl, [disk\_id]  ; Поместим в EBX адрес списка загруженных файлов  mov ebx, module\_list  ; Поместим в ESI адрес карты памяти  mov esi, memory\_map  ; Переходим на ядро  jmp 0xFFC00000 Первое ядро Ядро пока у нас будет состоять из двух файлов - startup.asm и main.c. startup.asm нужен для того, чтобы быть уверенными, что управление попадёт на функцию kernel\_main. Ведь она может быть не в начале файла, а содержимое startup.o мы полностью контролируем и если укажем его первым линкеру, то будем управлять и первыми байтами двоичного файла.  format ELF  public \_start  extrn kernel\_main  section ".text" executable  \_start:  movzx edx, dl  push edx  push esi  push ebx  lgdt [gdtr]  call kernel\_main  @:  ;cli  ;hlt  jmp @b  section ".data" writable  gdt:  dq 0  dq 0x00CF9A000000FFFF  dq 0x00CF92000000FFFF  gdtr:  dw $ - gdt  dd gdt   Ну вот и последний наш код на чистом Assembler :-). Он выполняет простейшую задачу - уложить в стек три аргумента для функции kernel\_main и передать на неё управление. После возврата из неё ядро уходит в бесконечный цикл. По соглашению вызова функций Си параметры следует пихать в стек в образом порядке. Также этот код инициализации загружает новое значение в GDTR - теперь таблица дескрипторов сегментов находится в пространстве ядра и даже если мы отмонтируем первый мегабайт не произойдёт никаких ошибок.  А теперь самое вкусное - простейшее ядро на языке высокого уровня:  typedef struct {  unsigned long long base;  unsigned long long size;  } BootModuleInfo;  void kernel\_main(char boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  char \*screen\_buffer = (void\*)0xB8000;  char \*msg = "Hello world!";  unsigned int i = 24 \* 80;  while (\*msg) {  screen\_buffer[i \* 2] = \*msg;  msg++;  i++;  }  }  Это ядро не делает ничего особенного - просто выводит строку "Hello world!" на последнюю строчку текстового экрана. Структура описанная в начале будет нужна для доступа к списку загруженных модулей.  Важно помнить, что никакой стандартной библиотеки у нас нет - нам доступны только те функции, которые мы сделаем сами. Все printf, strcpy, memcpy и т. п. придётся реализовывать самостоятельно, не пытайтесь обратиться к ним. В следующем выпуске мы займёмся созданием нашего собственного жутко урезанного аналога libc, чтобы программировать было удобнее. Тут начинается самая интересная часть, а принятые решения во многом повлияют на всю структуру системы. Сборка ядра Исполняемые файлы собираются в два этапа - компиляция, а потом линковка. На первом этапе компилятор преобразует исходный код в команды процессора и сохраняет всё это в объектный файл. Каждый модуль системы сохраняется в отдельном файле. В этом файле так же содержится информация о функциях, описанных в модули, поэтому из одного файла можно свободно вызывать функцию из другого. Весь код в объектных файлах не привязан к конкретным адресам. На втором этапе линкер собирает все объектные файлы в один бинарный. При этом код привязывается к конкретным адресам (если, конечно, мы не собираем динамически загружаемую библиотеку), вместо ссылок на функции подставляются нужные адреса. Нам нужно получить на выходе особый двоичный файл. Это просто код и данные, без каких-либо заголовков (то есть это не PE и не ELF). В качестве базового адреса используется адрес 0xFFC00000. Для упрощения этого мы опишем всё, что нам нужно в специальном формате скрипта ld:  OUTPUT\_FORMAT("binary")  ENTRY(\_start)  SECTIONS {  .text 0xFFC00000 : {  \*(.text)  \*(.code)  \*(.rodata\*)  }  .data ALIGN(0x1000) : {  \*(.data)  }  .bss ALIGN(0x1000) : {  \*(.bss)  }  .empty ALIGN(0x1000) - 1 : {  BYTE(0)  }  }  Этот скрипт говорит, что наш файл будет лежать в памяти непрерывным блоком начиная с адреса 0xFFC00000. В самом начале будет идти секция кода, потом секция read-only данных, затем обычных данных, потом неинициализированных. Все секции выровнены на размер страницы 4 КБ (вдруг мы потом захотим защитить на уровне таблицы страниц код от записи). Последнее описание секции .empty необходимо для того, чтобы даже неинициаилизорованные переменные занимали место в файле (там будут нули). Ведь начальный загрузчик выделяет память для ядра руководствуясь размером файла.  Собрать всё ядро можно следующими командами:  fasm startup.asm startup.o  gcc -c -m32 -ffreestanding -o main.o main.c  ld --oformat=binary -melf\_i386 -T script.ld -o kernel.bin startup.o main.o  Параметр GCC -ffreestanding указывает ему отключить все стандартные библиотеки. Ведь они привязаны к конкретной операционной системе, а мы пишем новую. Сборка образа диска Обойдусь без лишних комментариев и просто приведу линуксовый скрипт сборки образа:  dd if=bin/boot.bios.bin of=bin/boot\_sector.bin bs=512 count=1  dd if=bin/boot.bios.bin of=disk/boot.bin bs=1 skip=512  cp bin/kernel.bin disk/kernel.bin  bin/make\_listfs of=disk.img bs=512 size=2880 boot=bin/boot\_sector.bin src=./disk  Он предполагает, что все скомпилированные файлы лежат в bin в текущем каталоге, а ещё имеется каталог disk, в котором лежит boot.cfg следующего содержания:  # Loading kernel  Lkernel.bin  # Boot 32 bit kernel  S32  Если вы всё сделали правильно, полученный образ можно запустить в эмуляторе или даже на реальном железе и вы получите подобную картину:   Загрузчик считывает конфигурационный файл, загружает ядро, переходит в защищённый режим и передаёт ему управление. Получив его, наше ядро выводит последнюю строку на экран. Это лишь начало долгого пути, мы переходим к самой интересной части разработки. Теперь выпуски будут гораздо более простым для восприятия, благодаря использованию языка высокого уровня, который как я надеюсь все и так знают. Если вы не хотите разбираться с Assembler, можете просто взять мой готовый загрузчик и startup.asm и изменять уже только содержимое main.c, поскольку весь код до этого не диктует жёстко какие-либо параметры ядра (кроме ФС с которой мы загружаемся) и позволяет построить на своей базе что угодно. Автоматизация сборки или Makefile Вы могли заметить, что вручную набивать столько команд достаточно утомительно. К тому же не всегда есть необходимость перекомпилировать все файлы. Например, если startup.asm не был изменён, можно не вызывать fasm. Специально для упрощения компиляции приложений была придумана утилита make, которая входит в стандартную поставку GCC и MinGW.  Любой Makefile стоит из набора правил с такой структурой:  ИмяЦелиИлиФайла: ИмяПервогоИсходногоФайла ИмяВторогоИсходногоФайла  ...  КомандыКомпиляции  Первое правило, которое должно быть в любом Makefile - цель all. make смотрит на зависимости цели all и компилирует их, а затем выполняет команды и этой цели. Для каждой другой цели сначала собираются её зависимости. При этом имя цели и имя зависимостей могут совпадать с именами реальных файлов. В таком случае пересборка цели произойдёт только если исходники были изменены.  Ещё одна цель, которая часто используется в Makefile - clean. Её задача удалить все бинарные файлы, чтобы начать сборку "с чистого листа". Вот так может выглядеть Makefile для ядра:  all: startup.o main.o script.ld  ld --oformat=binary -melf\_i386 -T script.ld -o kernel.bin startup.o main.o  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  main.o: main.c  gcc -c -m32 -ffreestanding -o main.o main.c  clean:  rm -v \*.o kernel.bin  Этот текст необходимо сохранить в файл с именем Makefile (без расширения) в каталог с исходными текстами ядра. Теперь достаточно выполнить команду make без параметров, находясь в этом каталоге и мы получим файл kernel.bin (либо сообщения об ошибках, если что-то пошло не так).  А вот так я собираю загрузчик:  all: boot.bios.bin  boot.bios.bin: boot.bios.asm  fasm boot.bios.asm boot.bios.bin  clean:  rm -v boot.bios.bin   ... и make\_listfs:  all: compile  compile: make\_listfs.c  gcc -o make\_listfs make\_listfs.c  clean:  rm -f make\_listfs make\_listfs.exe  Ну и наконец расскажу про вызов других Makefile из одного. Я достаточно ленив, чтобы даже заходить в каталоги с каждым компонентом системы, поэтому создал 1 Makefile, который собирает сразу всю систему. У меня есть папка src, в ней подкаталоги: boot, kernel, make\_listfs. В самой src находится вот такой Makefile:  all:  make -C boot/  make -C kernel/  make -C make\_listfs/  clean:  make -C boot/ clean  make -C kernel/ clean  make -C make\_listfs clean   Теперь, находясь в каталоге src я просто пишу make и получаю полностью собранную систему, а если написать make clean, то все двоичные файлы будут удалены и останутся только исходники.  Ну и в довершение последний скрипт, который выполняет полную компиляцию и сборку всех компонентов и образа диска. В одном каталоге с ним надо разместить src, пустой каталог bin и каталог disk с файлом boot.cfg.  #!/bin/sh  make -C src  cp src/boot/boot.bios.bin bin/  cp src/kernel/kernel.bin bin/  cp src/make\_listfs/make\_listfs bin/  dd if=bin/boot.bios.bin of=bin/boot\_sector.bin bs=512 count=1  dd if=bin/boot.bios.bin of=disk/boot.bin bs=1 skip=512  cp bin/kernel.bin disk/kernel.bin  bin/make\_listfs of=disk.img bs=512 size=2880 boot=bin/boot\_sector.bin src=./disk  read -p "Press Enter to continue..." dummy  С таким набором скриптов сборка система становится предельно простой, особенно если учесть, что последний скрипт можно запускать двойным кликом из файлового менеджера. Различные команды вроде dd, cp, rm не существуют под Windows, поэтому её пользователям пригодится пакет MSYS или Cygwin. Однако простая сборка всех компонентов будет работать даже если у вас есть только GCC и fasm (make\_listfs легко скомпилируется и запустится в виде Windows-приложения). Примечание для пользователей ОС Windows ld для Windows не совсем полноценный - он не поддерживает вывод сразу в бинарный файл, только в EXE. Исправить это можно создав сначала EXE (ld не обратит внимание, что базовые адреса секций невозможные для вендовых бинарников), а потом вытащить оттуда чистые данные с помощью objcopy. Если вы столкнётесь с тем, что ld отказывается создавать файл kernel.bin, воспользуйтесь вот таким Makefile для ядра:  all: startup.o main.o script.ld  ld -melf\_i386 -T script.ld -o kernel.bin startup.o main.o  objcopy kernel.bin -O binary  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  main.o: main.c  gcc -c -m32 -ffreestanding -o main.o main.c  clean:  rm -v \*.o kernel.bin   Заодно уберите строку OUTPUT\_FORMAT("binary") из script.ld. Теперь и под Windows получится собрать ядро системы. Загрузка системы на реальной машине После таких успехов у некоторых может возникнуть желание опробовать новую ОС на реальном железе. Это не представляет проблем. С помощью HxD в Windows откройте дискету или флешку, выбрав вариант "Открыть диск". При открытии флешки важно открыть именно саму флешку, а не её раздел. В другой вкладке откройте disk.img, выделите его содержимое полностью и скопируйте на диск с его самого начала. После этого можно нажать "Сохранить" и дождаться окончания записи. Все данные на флешке или дискете при этом будут уничтожены, а для того, чтобы её использовать снова по назначению, её придётся заново отформатировать!  Пользователи Linux могут поступить проще - выполнить специальную команду в терминале. Для дискеты:  dd if=disk.img of=/dev/fd0  Для флешки:  dd  if=disk.img of=/dev/sdX  Вместо sdX надо подставить настоящее имя устройства (sda, sdb, sdc, sdd и т. д.). Главное при этом не перепутать и не записать образ на системный диск, уничтожив все данные. Разумеется, обе команды должны выполняться от имени root или с помощью sudo.  После этого надо настроить в BIOS загрузку с дискеты или флешки (старые BIOS не поддерживают флешки) и наслаждаться видом "Hello world". Заключение Ну вот собственно и всё на сегодня. Мы наконец-то закончили программирование на Assembler (хотя в С всё равно придётся иногда делать ассемблерные вставки для работы с оборудованием) и перешли на язык высокого уровня. Ещё очень много предстоит сделать. Мы можете уже проводить различные эксперименты, изменяя мой main.c, только учтите, что любая ошибка (доступ к неспроецированной памяти, деление на ноль) приведёт к перезагрузке или зависанию системы (мы пока не обрабатываем исключения, поэтому процессор не может продолжить работу после ошибки). До встречи!  Любые вопросы вы можете задать на мой адрес: [kiv.apple@gmail.com](mailto:apple@gmail.com). И да, сейчас самое время для различных идей по концепции ОС и предложений. |

## Пишем свою операционную систему. Стандартная библиотека

|  |
| --- |
| Заголовочный файл стандартной библиотеки В прошлом выпуске мы наконец-то перешли с Ассемблера на язык высокого уровня Си и написали простейшее ядро. Поскольку мы пишем свою ОС, в нашем распоряжении лишь аппаратные возможности и конструкции языка Си, никакой стандартной библиотеки и функций - ведь они рассчитаны на работу под существующей системой (если вы компилируете программу в Linux, функции рассчитаны на работу с системными вызовами Linux, если под Windows, то функции уже другие и т. д.). Сегодня мы напишем базовые функции, являющиеся аналогами соответствующих функциям из libc, чтобы упростить разработку. Также наша стандартная библиотека будет содержать определения типов специфичные для низкоуровневого программирования. Для начала приведу заголовочный файл stdlib.h:  #ifndef STDLIB\_H  #define STDLIB\_H  typedef enum {  false = 0,  true = 1  } bool;  #define NULL ((void\*)0)  typedef unsigned char uint8;  typedef signed char int8;  typedef unsigned short uint16;  typedef signed short int16;  typedef unsigned long uint32;  typedef signed long int32;  typedef unsigned long long uint64;  typedef signed long long int64;  #ifdef \_\_x86\_64\_\_  typedef uint64 size\_t;  #else  typedef uint32 size\_t;  #endif  #define min(a, b) (((a) > (b)) ? (b) : (a))  #define max(a, b) (((a) > (b)) ? (a) : (b))  void memset(void \*mem, char value, size\_t count);  void memset\_word(void \*mem, uint16 value, size\_t count);  void memcpy(void \*dest, void \*src, size\_t count);  int memcmp(void \*mem1, void \*mem2, size\_t count);  void \*memchr(void \*mem, char value, size\_t count);  size\_t strlen(char \*str);  void strcpy(char \*dest, char \*src);  void strncpy(char \*dest, char\*src, size\_t max\_count);  int strcmp(char \*str1, char \*str2);  char \*strchr(char \*str, char value);  #endif  В этом заголовочном файле описан булевный тип, типы заданной разрядности (uint8 - целое 8 бит без знака, int32 - целое 32 бита со знаком и т. д.), которых так не хватает в обычном Си, полезные макросы min и max (возвращают минимальный или максимальный свой аргумент), прототипы функций работы с памятью, прототипы строковых функций. Описанные функции теперь надо ещё реализовать в файле stdlib.c. Начнём с подключения заголовочного файла:  #include "stdlib.h"  Важно указать имя файла в кавычках, а не знаках больше-меньше, как мы привыкли (#include <stdlib.h>), чтобы файл в первую очередь искался к текущем каталоге, а не в каталоге с библиотечными include'ами. Ассемблерные вставки GCC как и многие другие компиляторы позволяет включать в программу фрагменты ассемблерных инструкций. Это просто необходимая возможность для разработчиков ОС. Синтаксис у ассемблерных  вставок таков:  asm("код на языке Assembler":выходные параметры:входные  параметры);   Входные и выходные параметры - список переменных, значение которых следует поместить в регистры перед исполнением кода, а потом переместить обратно в переменные. Важно также отметить, что используется не Intel, а AT&T синтаксис:     * Параметры большинства инструкций указываются в обратном порядке: вместо "mov приёмник, источник" надо писать "mov источник, приёмник". * В конце имени команды добавляется суффикс "b" (байт), "w" (слово), "l" (двойное слово), "q" (четверное слово, только для 64-битной системы) в зависимости от размера операндов. * Все константы имеют префикс "$", все регистры "%". Если были указаны входные-выходные параметры, то параметры выглядят как "%0", "%1" в зависимости от порядка их следования, а настоящие регистры претворяются двумя символами процента.      Пример ассемблерной вставки в GCC:  asm("movl  %1, %%edx \n add $10, %%edx \n movl %%edx, %0":"=a"(dest):"b"(src));  asm("lgdt (,%0,)"::"a"(&gdtr));  asm("cli");   Как можно заметить, для того чтобы указать несколько команд в одной конструкции asm используется вставка символа переноса строки "\n".  Не трудно догадаться, что имя аргумента "a" обозначает EAX, "b" - EBX, "c" - ECX, "d" - EDX, но лучше на это не надеяться, потому что всё зависит от особенностей компилятора и целевой архитектуры. Функции работы с памятью Теперь мы можем реализовать функции работы с памятью:  void memset(void \*mem, char value, size\_t count) {  asm("movl %0, %%eax \n movl %1, %%edi \n movl %2, %%ecx \n rep stosl"  ::"a"((uint32)value | ((uint32)value << 8) | ((uint32)value << 16) | ((uint32)value << 24)),"b"(mem),"c"(count >> 2));  asm("movb %b0, %%al \n movl %1, %%ecx \n rep stosb"::"a"(value),"b"(count & 3));  }  void memset\_word(void \*mem, uint16 value, size\_t count) {  asm("movl %0, %%eax \n movl %1, %%edi \n movl %2, %%ecx \n rep stosl"  ::"a"((uint32)value | ((uint32)value << 16)),"b"(mem),"c"(count >> 1));  }    void memcpy(void \*dest, void \*src, size\_t count) {  asm("movl %0, %%edi \n movl %1, %%esi \n movl %2, %%ecx \n rep movsl"::"a"(dest),"b"(src),"c"(count >> 2));  asm("movl %0, %%ecx \n rep movsb"::"a"(count & 3));  }    int memcmp(void \*mem1, void \*mem2, size\_t count) {  char above, below;  asm("movl %0, %%esi \n movl %1, %%edi \n movl %2, %%ecx \n repe cmpsb"::"a"(mem1),"b"(mem2),"c"(count));  asm("seta %0 \n setb %1":"=a"(above),"=b"(below));  return above - below;  }    void \*memchr(void \*mem, char value, size\_t count) {  void \*result;  asm("movb %b0, %%al \n movl %1, %%edi \n movl %2, %%ecx \n repe cmpsb"::"a"(value),"b"(mem),"c"(count));  asm("movl %%edi, %0":"=a"(result));  if (result < mem + count) {  return result;  } else {  return NULL;  }  }  Эти функции, написанные с использованием Assembler, послужат основой для строковых функций:  size\_t strlen(char \*str) {  return (char\*)memchr(str, '\0', -1) - str;  }  void strcpy(char \*dest, char \*src) {  memcpy(dest, src, strlen(src) + 1);  }  void strncpy(char \*dest, char \*src, size\_t max\_count) {  size\_t len = min(max\_count - 1, strlen(src));  memcpy(dest, src, len);  dest[len] = '\0';  }  int strcmp(char \*str1, char \*str2) {  return memcmp(str1, str2, strlen(str1) + 1);  }  char \*strchr(char \*str, char value) {  return memchr(str, value, strlen(str));  }  Теперь можно переписать main.c с учётом появления новых функций:  #include "stdlib.h"  typedef struct {  uint64 base;  uint64 size;  } BootModuleInfo;  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  char \*screen\_buffer = (void\*)0xB8000;  memset\_word(screen\_buffer, 0x0E00, 2000);  char msg[] = {'H',0x0E,'e',0x0E,'l',0x0E,'l',0x0E,'o',0x0E,' ',0x0E,'w',0x0E,'o',0x0E,'r',0x0E,'l',0x0E,'d',0x0E,'!',0x0E};  memcpy(screen\_buffer, msg, sizeof(msg));  }  Данный код сначала очищает экран, а потом выводит надпись "Hello world!" жёлтым цветом на чёрном фоне в левом верхнем углу. Заключение Теперь у нас есть немного функций, которые упростят нам жизнь. В следующем выпуске мы напишем своеобразный аналог printf для более удобного вывода текста на экран, а также узнаем про порты ввода-вывода. Привожу полные листинги модулей:  **stdlib.c:**  #include "stdlib.h"  void memset(void \*mem, char value, size\_t count) {  asm("movl %0, %%eax \n movl %1, %%edi \n movl %2, %%ecx \n rep stosl"  ::"a"((uint32)value | ((uint32)value << 8) | ((uint32)value << 16) | ((uint32)value << 24)),"b"(mem),"c"(count >> 2));  asm("movb %b0, %%al \n movl %1, %%ecx \n rep stosb"::"a"(value),"b"(count & 3));  }  void memset\_word(void \*mem, uint16 value, size\_t count) {  asm("movl %0, %%eax \n movl %1, %%edi \n movl %2, %%ecx \n rep stosl"  ::"a"((uint32)value | ((uint32)value << 16)),"b"(mem),"c"(count >> 1));  }  void memcpy(void \*dest, void \*src, size\_t count) {  asm("movl %0, %%edi \n movl %1, %%esi \n movl %2, %%ecx \n rep movsl"::"a"(dest),"b"(src),"c"(count >> 2));  asm("movl %0, %%ecx \n rep movsb"::"a"(count & 3));  }  int memcmp(void \*mem1, void \*mem2, size\_t count) {  char above, below;  asm("movl %0, %%esi \n movl %1, %%edi \n movl %2, %%ecx \n repe cmpsb"::"a"(mem1),"b"(mem2),"c"(count));  asm("seta %0 \n setb %1":"=a"(above),"=b"(below));  return above - below;  }  void \*memchr(void \*mem, char value, size\_t count) {  void \*result;  asm("movb %b0, %%al \n movl %1, %%edi \n movl %2, %%ecx \n repe cmpsb"::"a"(value),"b"(mem),"c"(count));  asm("movl %%edi, %0":"=a"(result));  if (result < mem + count) {  return result;  } else {  return NULL;  }  }  size\_t strlen(char \*str) {  return (char\*)memchr(str, '\0', -1) - str;  }  void strcpy(char \*dest, char \*src) {  memcpy(dest, src, strlen(src) + 1);  }  void strncpy(char \*dest, char \*src, size\_t max\_count) {  size\_t len = min(max\_count - 1, strlen(src));  memcpy(dest, src, len);  dest[len] = '\0';  }  int strcmp(char \*str1, char \*str2) {  return memcmp(str1, str2, strlen(str1) + 1);  }  char \*strchr(char \*str, char value) {  return memchr(str, value, strlen(str));  }  **Makefile:**  all: startup.o stdlib.o main.o script.ld  ld --oformat=binary -melf\_i386 -T script.ld -o kernel.bin startup.o stdlib.o main.o  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  stdlib.o: stdlib.c stdlib.h  gcc -c -m32 -ffreestanding -o stdlib.o stdlib.c  main.o: main.c stdlib.h  gcc -c -m32 -ffreestanding -o main.o main.c  clean:  rm -v \*.o kernel.bin  До встречи! |

## Пишем свою операционную систему. Драйвер текстового экрана

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| До этого момента мы выводили текст на экран лишь с помощью прямого копирования байт в видео-память. Было бы не плохо реализовать функции вроде printf для более удобного вывода на экран.  Вообще-то я хочу получить в итоге микроядро, и драйверу консоли там не место, однако поскольку эта рассылка имеет цель не только задокументировать ход работы, но и наглядно показать процесс разработки, я пока немного нарушу последовательность и напишу драйвер для ядра. Ничто не мешает в будущем удалить лишние модули из ядра.  Консоль в некоторых её проявлениях называется телетайпом, я буду придерживаться этой же терминологии. Код драйвера будет располагаться в tty.c. Для начала опишем заголовочный файл tty.h, который определяет доступные другим модулям функции:  #ifndef TTY\_H  #define TTY\_H  void init\_tty();  void out\_char(char chr);  void out\_string(char \*str);  void clear\_screen();  void set\_text\_attr(char attr);  void move\_cursor(unsigned int pos);  #endif  Имена функций говорят сам за себя: init\_tty - инициализация драйвера, необходимо вызвать до обращения к любым другим функциям. out\_char - вывод одиночного символа, поддерживается символ переноса строки '\n' out\_string - вывод целой строки символов clear\_screen - очистка экрана и перевод курсора в левый верхний угол set\_text\_attr - смена текущего цвета и фона символов, описание формата цвета будет чуть ниже. move\_cursor - перемещение курсора на заданную позицию.  Видео-память в текстовом режиме представляет собой массив из 2 тысяч слов (для разрешения экрана 80x25 символов). Первый байт слова содержит сам код символа, а второй - его атрибуты (цвет фона и текста). Младшие 4 бита атрибутов символа содержат код цвета текста. Таким образом доступно 16 цветов. Следующие 3 бита содержат код цвета фона (значит существует 8 цветов фона для текста), а самый старший бит включает мерцание символа.  Вот начало файла tty.c:  #include "stdlib.h"  #include "tty.h"  typedef struct {  uint8 chr;  uint8 attr;  } TtyChar;  unsigned int tty\_width;  unsigned int tty\_height;  unsigned int cursor;  uint8 text\_attr;  TtyChar \*tty\_buffer;  uint16 tty\_io\_port;  Мы подключаем стандартную библиотеку и описываем несколько внутренних переменных драйвера, а также структуру одиночного символа. Про назначение tty\_io\_port я расскажу чуть позже. В первую очередь необходимо инициализировать телетайп. Было бы не плохо подхватить старую позицию курсора от BIOS. Также, для перемещения аппаратного курсора (мерцающая горизонтальная черта) нам необходимо узнать tty\_io\_port. Все эти данные можно извлечь из области данных BIOS, которая расположена по адресу 0x400 - 0x600. Поскольку 1-ый мегабайт примонтирован, мы можем использовать обычное чтение.  Нас интересуют следующие данные из области данных BIOS:   |  |  | | --- | --- | | Адрес | Комментарий | | 0x44A | 2 байта. Количество столбцов на экране. Если количество строк постоянно - 25, то количество столбцов теоретически может быть не только 80, но и 40. Будет красиво прочитать это отсюда. | | 0x463 | 2 байта. Базовый порт ввода-вывода для управления контроллером дисплея. | | 0x450 | 1 байт. Координата курсора Y | | 0x451 | 1 байт. Координата курсора X |   Задачей init\_tty будет как раз прочитать эти параметры и сохранить их в нужные переменные:  void init\_tty() {  tty\_buffer = (void\*)0xB8000;  tty\_width = \*((uint16\*)0x44A);  tty\_height = 25;  tty\_io\_port = \*((uint16\*)0x463);  cursor = (\*((uint8\*)0x451)) \* tty\_width + (\*((uint8\*)0x450));  text\_attr = 7;  }  Вывод символа с обработкой переноса строки. Для сдвига курсора используется move\_cursor, чтобы и обновить положение аппаратного курсора, и прокрутить экран в случае, если кончилось место.  void out\_char(char chr) {  switch (chr) {  case '\n':  move\_cursor((cursor / tty\_width + 1) \* tty\_width);  break;  default:  tty\_buffer[cursor].chr = chr;  tty\_buffer[cursor].attr = text\_attr;  move\_cursor(cursor + 1);  }  }  Как многие уже могли догадаться, вывод строки будет выглядеть максимально примитивно:  void out\_string(char \*str) {  while (\*str) {  out\_char(\*str);  str++;  }  }  Функция очистки экрана просто заполняет экран пробелами с текущим цветом и переводит курсор в нулевую позицию:  void clear\_screen() {  memset\_word(tty\_buffer, (text\_attr << 8) + ' ', tty\_width \* tty\_height);  move\_cursor(0);  }  Смена текущего цвета текста приводит лишь к обновлению переменной text\_attr. По-настоящему цвет будет использоваться уже при выводе символов или очистке экрана:  void set\_text\_attr(char attr) {  text\_attr = attr;  }  Ну и наконец самая сложная функция - move\_cursor. Для начала приведу упрощённый вариант, который не перемещает аппаратный курсор:  void move\_cursor(unsigned int pos) {  cursor = pos;  if (cursor >= tty\_width \* tty\_height) {  cursor = (tty\_height - 1) \* tty\_width;  memcpy(tty\_buffer, tty\_buffer + tty\_width, tty\_width \* tty\_height \* sizeof(TtyChar));  memset\_word(tty\_buffer + tty\_width \* (tty\_height - 1), (text\_attr << 8) + ' ', tty\_width);  }  }  Эта функция обновляет значение переменной cursor, а также следит за тем, чтобы его позиция не вышла за границу экрана. Если такое случается курсор перемещается на начало последней строки, всё содержимое экрана сдвигается на 1 строку вверх, а последняя строка очищается.  Такой драйвер уже можно использовать для вывода текста на экран, но не хватает одной мелочи - аппаратный курсор остаётся на том же месте, где его оставил BIOS. Надо бы доработать нашу функцию move\_cursor.  Для начала расскажу про порты ввода-вывода. В архитектуре x86 общение с устройствами может осуществляться двумя способами - через память и через порты ввода-вывода. Обращение через память обозначает, что какой-то блок физических адресов не является на самом деле ОЗУ, а все запросы чтения-записи уходят к устройству, которое реагирует на них согласно своим функциям (например, что-то вроде "при записи единицы по такому-то физическому адресу такое-то устройство перейдёт в активный режим"). Помимо этого способа программисту доступно 65536 портов ввода-вывода. Из каждого из них можно прочитать или записать 1 байт, однако можно объединять их в группы - например, если записать слово в порт 0x100, то его младшая половина уйдёт в 0x100, а старшая в 0x101. Для работы с портами существует две ассемблерных команды - in и out. Они существуют в единственной форме - in al/ax/eax, dx и out dx, al/ax/eax. Аргументы указывать обязательно, но в итоге номер порта хранится непременно в DX, а значение для ввода или вывода в AL/AX/EAX (в зависимости от того сколько данных мы хотим записать или прочитать).  Контроллер дисплея имеет два порта - tty\_io\_port и tty\_io\_port + 1. Первый задаёт номер его внутреннего регистра. После записи туда значения, второй порт содержит значение этого регистра. Если произвести запись во второй порт, значение соответствующего регистра изменится.  Нас интересует два внутренних регистра контроллера - 0x0E и 0x0F. Первый из них хранит старший байт позиции курсора, второй младший байт.  Доработаем move\_cursor:  void  move\_cursor(unsigned int pos) {  cursor = pos;  if (cursor >= tty\_width \* tty\_height) {  cursor = (tty\_height - 1) \* tty\_width;  memcpy(tty\_buffer, tty\_buffer + tty\_width, tty\_width \* tty\_height \* sizeof(TtyChar));  memset\_word(tty\_buffer + tty\_width \* (tty\_height - 1), (text\_attr << 8) + ' ', tty\_width);  }  asm("movw %w0, %%dx \n movl %1, %%ecx \n movb $0x0E, %%al \n movb %%ch, %%ah \n outw %%ax, %%dx \n incb %%al \n movb %%cl, %%ah \n outw %%ax, %%dx"  ::"d"(tty\_io\_port),"c"(cursor));  }  Вот и всё. Наш драйвер телетайпа готов! Теперь можно будет удобно и просто выводить текстовую информацию на экран. Например, вот так:  #include "stdlib.h"  #include "tty.h"  typedef struct {  uint64 base;  uint64 size;  } BootModuleInfo;  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  init\_tty();  out\_string("Hello world!\n");  }  При этом вывод продолжится с того места, где остановился загрузчик. Или так:  #include "stdlib.h"  #include "tty.h"  typedef struct {  uint64 base;  uint64 size;  } BootModuleInfo;  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  init\_tty();  set\_text\_attr(0x1F);  clear\_screen();  out\_string("Hello world!\n");  }  В этом случае ядро заполняет экран синим цветом и выводит белую надпись "Hello world".  Как обычно, в заключение выпуска привожу полный код написанного модуля tty.c:  #include "stdlib.h"  #include "tty.h"  typedef struct {  uint8 chr;  uint8 attr;  } TtyChar;  unsigned int tty\_width;  unsigned int tty\_height;  unsigned int cursor;  uint8 text\_attr;  TtyChar \*tty\_buffer;  uint16 tty\_io\_port;  void init\_tty() {  tty\_buffer = (void\*)0xB8000;  tty\_width = \*((uint16\*)0x44A);  tty\_height = 25;  tty\_io\_port = \*((uint16\*)0x463);  cursor = (\*((uint8\*)0x451)) \* tty\_width + (\*((uint8\*)0x450));  text\_attr = 7;  }  void out\_char(char chr) {  switch (chr) {  case '\n':  move\_cursor((cursor / tty\_width + 1) \* tty\_width);  break;  default:  tty\_buffer[cursor].chr = chr;  tty\_buffer[cursor].attr = text\_attr;  move\_cursor(cursor + 1);  }  }  void out\_string(char \*str) {  while (\*str) {  out\_char(\*str);  str++;  }  }  void clear\_screen() {  memset\_word(tty\_buffer, (text\_attr << 8) + ' ', tty\_width \* tty\_height);  move\_cursor(0);  }  void set\_text\_attr(char attr) {  text\_attr = attr;  }  void move\_cursor(unsigned int pos) {  cursor = pos;  if (cursor >= tty\_width \* tty\_height) {  cursor = (tty\_height - 1) \* tty\_width;  memcpy(tty\_buffer, tty\_buffer + tty\_width, tty\_width \* tty\_height \* sizeof(TtyChar));  memset\_word(tty\_buffer + tty\_width \* (tty\_height - 1), (text\_attr << 8) + ' ', tty\_width);  }  asm("movw %w0, %%dx \n movl %1, %%ecx \n movb $0x0E, %%al \n movb %%ch, %%ah \n outw %%ax, %%dx \n incb %%al \n movb %%cl, %%ah \n outw %%ax, %%dx"  ::"d"(tty\_io\_port),"c"(cursor));  }  И новый Makefile:  LDFLAGS = -melf\_i386  CFLAGS = -m32 -ffreestanding  all: startup.o stdlib.o main.o tty.o script.ld  ld $(LDFLAGS) -T script.ld -o kernel.bin startup.o stdlib.o main.o tty.o  objcopy kernel.bin -O binary  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  stdlib.o: stdlib.c stdlib.h  gcc -c $(CFLAGS) -o stdlib.o stdlib.c  main.o: main.c stdlib.h tty.h  gcc -c $(CFLAGS) -o main.o main.c  tty.o: tty.c tty.h stdlib.h  gcc -c $(CFLAGS) -o tty.o tty.c  clean:  rm -v \*.o kernel.bin Ещё немного информации для пользователей Windows Оказывается ld для win32 не умеет собирать исполняемые ELF файлы, поэтому для успешной компиляции придётся изменить строку LDFLAGS = -melf\_i386 на LDFLAGS = -mi386pe.  Один из подписчиков моей рассылки - DragoN - предложил специальный bat-файл для упрощения сборки ОС под Windows:  make -C src  cp src/boot/boot.bios.bin bin/  cp src/kernel/kernel.bin bin/  cp src/make\_listfs/make\_listfs bin/  dd if=bin\boot.bios.bin of=bin\boot\_sector.bin bs=512 count=1  dd if=bin\boot.bios.bin of=disk\boot.bin bs=1 skip=512  cp bin/kernel.bin disk/kernel.bin  bin\make\_listfs of=disk.img bs=512 size=2880 boot=bin/boot\_sector.bin src=./disk  Вот его комментарии касательно особенностей сборки:  Сборка ОС MyOS под Win32.  Инструменты:  1. FASM - чтобы собрать загрузчик  2. MinGW - из него GCC чтобы собрать ядро  3. MSYS - понадобятся утилиты sh, cp, rm, dd  3. DD - утилита понадобится отдельно, так как не идћт с MinGW, MSYS  4. Bochs  Процесс:  1. Устанавливаем FASM, MinGW, MSYS, DD (я взял отсюда http://www.chrysocome.net/dd), Bochs  2. Настраиваем пути к FASM, MinGW, MSYS, DD, Bochs (переменная окружения PATH)  3. Перезагружаем шелл или тотал командер, чтобы новые пути вступили в силу  4. Запускаем в папке MyOS команду makew, получим:  MyOS\src\boot\boot.bios.bin  MyOS\src\kernel\main.o  MyOS\src\kernel\startup.o  MyOS\src\kernel\stdlib.o  MyOS\src\kernel\kernel.bin  MyOS\src\make\_listfs\make\_listfs.exe  MyOS\bin\boot.bios.bin  MyOS\bin\kernel.bin  MyOS\bin\make\_listfs.exe  MyOS\bin\boot\_sector.bin  MyOS\disk\kernel.bin  MyOS\disk\boot.bin  MyOS\disk.img  5. Запускаем bochsrc.bxrc (бочс при установке создаст ассоциацию .bxrc с собой)  Если видим жћлтый Hello, World!, значит ОС запустилась успешно.  Если что то пошло не так, сначала проверяем список файлов по пункту 4, скорее всего чего то не хватает,  и дальше действуем по обстоятельствам.   От себя добавлю, что надо обязательно убрать строку OUTPUT\_FORMAT из script.ld. Заключение Итак, сегодня мы разработали драйвер текстового экрана. Теперь можно выводить различные информационные сообщения на экран.  В следующем выпуске мы разберём обработку прерываний и ввод с клавиатуры. На этом я планирую закончить эту демонстрацию возможностей и, если ничего не изменится, уже через выпуск мы начнём разрабатывать менеджер памяти.  Задавайте вопросы на мой e-mail [kiv.apple@gmail.com](mailto:apple@gmail.com)! До встречи ;-) |

Пишем свою операционную систему. Bugfix и printf

|  |
| --- |
| Приветствую всех своих читателей!  Недавно заметил ошибку в написанном нами ещё давно startup.asm: аргументы передавались не в том порядке, в котором их принимает kernel\_main. Это не проявлялось бы, пока мы не начали использовать значения этих аргументов. Вот правильная версия:  format ELF  public \_start  extrn kernel\_main  section ".text" executable  \_start:  movzx edx, dl  **push ebx**  push esi  **push edx**  lgdt [gdtr]  call kernel\_main  **add esp, 3 \* 4**  @@:  cli  hlt  jmp @b  section ".data" writable  gdt:  dq 0  dq 0x00CF9A000000FFFF  dq 0x00CF92000000FFFF  gdtr:  dw $ - gdt  dd gdt  Одного исправления ошибки маловато для нового выпуска, поэтому добавлю сюда и немного приятного для разработчиков функционала.  Во-первых, макросы для работы с портами: outportb, outportw, outportl, inportb, inportw, inportl. Они служат для чтения или записи значений в порты ввода-вывода и позволяет сделать основной код ядра понятнее, уменьшив количество ассемблерных вставок (хотя итоговый код будет тем же самым). Новый stdlib.h выглядит так:  #ifndef STDLIB\_H  #define STDLIB\_H  typedef enum {  false = 0,  true = 1  } bool;  #define NULL ((void\*)0)  typedef unsigned char uint8;  typedef signed char int8;  typedef unsigned short uint16;  typedef signed short int16;  typedef unsigned long uint32;  typedef signed long int32;  typedef unsigned long long uint64;  typedef signed long long int64;  #ifdef \_\_x86\_64\_\_  typedef uint64 size\_t;  #else  typedef uint32 size\_t;  #endif  #define min(a, b) (((a) > (b)) ? (b) : (a))  #define max(a, b) (((a) > (b)) ? (a) : (b))  **#define outportb(port, value) asm("outb %b0, %w1"::"a"(value),"d"(port));**  **#define outportw(port, value) asm("outw %w0, %w1"::"a"(value),"d"(port));**  **#define outportl(port, value) asm("outl %0, %w1"::"a"(value),"d"(port));**  **#define inportb(port, out\_value) asm("inb %w1, %b0":"=a"(value):"d"(port));**  **#define inportw(port, out\_value) asm("inw %w1, %w0":"=a"(value):"d"(port));**  **#define inportl(port, out\_value) asm("inl %w1, %0":"=a"(value):"d"(port));**  void memset(void \*mem, char value, size\_t count);  void memset\_word(void \*mem, uint16 value, size\_t count);  void memcpy(void \*dest, void \*src, size\_t count);  int memcmp(void \*mem1, void \*mem2, size\_t count);  void \*memchr(void \*mem, char value, size\_t count);  size\_t strlen(char \*str);  void strcpy(char \*dest, char \*src);  void strncpy(char \*dest, char\*src, size\_t max\_count);  int strcmp(char \*str1, char \*str2);  char \*strchr(char \*str, char value);  #endif  Код функции move\_cursor из tty.c стал проще и понятнее:  void move\_cursor(unsigned int pos) {  cursor = pos;  if (cursor >= tty\_width \* tty\_height) {  cursor = (tty\_height - 1) \* tty\_width;  memcpy(tty\_buffer, tty\_buffer + tty\_width, tty\_width \* tty\_height \* sizeof(TtyChar));  memset\_word(tty\_buffer + tty\_width \* (tty\_height - 1), (text\_attr << 8) + ' ', tty\_width);  }  **outportb(tty\_io\_port, 0x0E);**  **outportb(tty\_io\_port + 1, cursor >> 8);**  **outportb(tty\_io\_port, 0x0F);**  **outportb(tty\_io\_port + 1, cursor & 0xFF);**  }  Ну и наконец главное новшество этого выпуска - реализация ограниченной поддержки функции printf. До этого момента у нас не было способа выводить форматированный текст, лишь готовые строки, теперь у нас появится б**о**льшая свобода действий.  Новый заголовочный файл tty.h:  #ifndef TTY\_H  #define TTY\_H  void init\_tty();  void out\_char(char chr);  void out\_string(char \*str);  void clear\_screen();  void set\_text\_attr(char attr);  void move\_cursor(unsigned int pos);  **void printf(char \*fmt, ...);**  #endif  Для написания функции с переменным числом аргументов потребуется включаемый файл stdarg.h. Пожалуй, это единственный файл стандартной библиотеки Си, который мы можем позволить себе включить в наше ядро. Он не требует наличия libc, а содержит лишь набор макросов. Начало tty.c немного изменяется:  **#include <stdarg.h>**  #include "stdlib.h"  #include "tty.h"  typedef struct {  uint8 chr;  uint8 attr;  } TtyChar;  unsigned int tty\_width;  ...  Для начала необходимо добавить в конец этого файла ряд вспомогательных определений для организации преобразования числа в строку перед выводом:  const char digits[] = "0123456789ABCDEF";  char num\_buffer[65];  char \*int\_to\_str(size\_t value, unsigned char base) {  size\_t i = sizeof(num\_buffer) - 1;  num\_buffer[i--] = '\0';  do {  num\_buffer[i--] = digits[value % base];  value = value / base;  } while (value);  return &num\_buffer[i + 1];  }  Эта функция позволяет преобразовать число value в строковое представление в любой системе счисления от 2 до 16. Она возвращает указатель на строку с числом. Следует использовать её или же скопировать в другое место до следующего вызова int\_to\_str, потому что этот адрес находится внутри глобальной переменной.  Ну и, наконец, самая главная функция - printf (её следует описать после int\_to\_str):  void printf(char \*fmt, ...) {  va\_list args;  va\_start(args, fmt);  while (\*fmt) {  if (\*fmt == '%') {  fmt++;  size\_t arg = va\_arg(args, size\_t);  switch (\*fmt) {  case '%':  out\_char('%');  break;  case 'c':  out\_char(arg);  break;  case 's':  out\_string((char\*)arg);  break;  case 'b':  out\_string(int\_to\_str(arg, 2));  break;  case 'o':  out\_string(int\_to\_str(arg, 8));  break;  case 'd':  out\_string(int\_to\_str(arg, 10));  break;  case 'x':  out\_string(int\_to\_str(arg, 16));  break;  }  } else {  out\_char(\*fmt);  }  fmt++;  }  va\_end(args);  }  Наш printf понимает следующие форматы: %s - Вывод строки. %c - Вывод символа с указанным кодом %b - Вывод числа в двоичном представлении %o - Вывод числа в восьмеричном представлении %d - Вывод числа в десятичном представлении %x - Вывод числа в шестнадцатеричном представлении %% - Вывод символа процента  Вывод вещественных чисел, точность, выравнивание не поддерживаются, потому что кода писать придётся много, а функционал в итоге всё равно нам не пригодится.  Функция kernel\_main из main.c теперь может выглядеть так:  void kernel\_main(uint8 boot\_disk\_id,  void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  init\_tty();  **set\_text\_attr(15);**  **printf("Welcome to MyOS!\n");**  **printf("Boot disk id is %d\n", boot\_disk\_id);**  **printf("Memory map at 0x%x\n", memory\_map);**  **printf("Boot module list at 0x%x\n", boot\_module\_list);**  **printf("String is %s, char is %c, number is %d, hex number is 0x%x", \_\_DATE\_\_, 'A', 1234, 0x1234);**  }  А результат выполнения такого когда виден ниже:  Теперь вы получили больший простор для экспериментов с ядром. В следующем выпуске, как я и обещал, вы рассмотрим обработку прерываний и клавиатурный ввод. До встречи! |

## Пишем свою операционную систему. Обработка прерываний

|  |
| --- |
| Доброго времени суток!  В предыдущем выпуске мы реализовали более-менее полноценный вывод текстовой информации на экран с возможностью управления позицией курсора и цветом текста, однако до сих пор пользователь никак не мог взаимодействовать с нашей ОС. Сегодня мы исправим этот недостаток, написав простой драйвер клавиатуры PS/2 (USB клавиатуры эмулируют PS/2, так что и они будут поддерживаться). Прерывания Нам уже известно два механизма взаимодействия процессора с остальными устройствами - через специальные регионы оперативной памяти (например, тот самый буфер текстового экрана по адресу 0xB8000 является примером этого механизма, который ещё называется Memory-Mapped Input Output - **MMIO**) и через порты ввода-вывода. Управление устройствами обеими способами всегда происходит только по инициативе процессора - что бы не случилось, процессор получит данные от устройства только когда явно выполнит чтение из памяти или порта. Чтобы отреагировать на внешнее событие (например, нажатие пользователем на клавишу клавиатуры, движение мышью, приход пакета по сети, окончание операции чтения с диска и т. д.) ему придётся непрерывно в цикле проверять значение регистра устройства, тратя на это достаточно много ресурсов. Для решения этой проблемы практически на всех существующих архитектурах процессоров был введён третий механизм взаимодействия с внешними устройствами - **прерывания**.  При поступлении прерывания (как правило, процессор узнаёт об этом по повышению уровня сигнала на одном из своих специальных выводов) процессор приостанавливает выполнение кода, сохраняет текущее значение CS, IP и FLAGS (пока рассмотрим работу в реальном режиме), извлекает из **таблицы прерываний** адрес обработчика и продолжает исполнение с него. Работа этого обработчика во многом аналогична работе функции (только в данном случае обязательно нужно сохранить все регистры, чтобы не нарушить ход выполнения прерванной программы), но вместо команды RET следует использовать IRET, которая в отличии от RETF, восстанавливает не только CS и IP, но и FLAGS.  Каждое прерывание характеризуется номером. Всего в архитектуре x86 существует 256 различных прерываний. Соответственно, таблица прерываний содержит адреса 256 обработчиков и каждый раз вызывается соответствующий номеру прерывания.  Прерывания можно разделить на два вида **программные** и **аппаратные**. Это деление базируется на способе вызова обработчика. В случае программного прерывания никакого сигнала от внешнего устройства не приходило, а в исполняемом коде встретилась инструкция INT, которая имеет единственный аргумент - номер прерывания, которое следует вызвать. Программные прерывания это один из способов реализации **системных вызовов** - приложение не может напрямую обратиться к коду ядра, но может выполнить эту команду. Параметры обычно в таком случае передаются в регистрах и какие из них надо сохранить зависит лишь о соглашении вызова функции. В реальном режиме, в котором нет разделения прав доступа кода, этот метод также используется, потому что позволяет обращаться к системным функциям (BIOS, DOS) не по адресу, который может меняться от версии к версии, а по постоянному номеру. При написании начального загрузчика мы использовали несколько сервисов BIOS, которые предоставлялись прерываниями с номерами 0x10 (управление экраном), 0x13 (работа с дисковой подсистемой), 0x15 (определение конфигурации оперативной памяти) и 0x16 (работа с клавиатурой). Контроллер прерываний Сигналы от внешних устройств, прежде чем поступить в процессор, попадают в **программируемый контроллер прерываний** (Programmable Interrupt Controller - **PIC**), он транслирует номер **IRQ** (Interrupt Request) от устройства в номер прерывания процессора (не все 256 прерываний могут быть аппаратными). В случае с PIC (существует ещё его расширенная версия - Advanced Programmable Interrupt Controller - **APIC**, но мы пока его не рассматриваем) существует 16 IRQ. То есть прерывания могут приходить от 16 различных устройств.  На самом деле обработка прерывания немного более сложная - каждый PIC способен обрабатывать лишь 8 прерываний, поскольку этого мало в системе установлено два таких контроллера. Один обрабатывает IRQ с 0 по 7, а другой с 8 по 15. Второй подключен к выводу IRQ2 первого. Нам нет необходимости беспокоится о том, как они маршрутизируют прерывания, достаточно помнить, что надо настраивать сразу два PIC. Их настройка ОС обычно сводится к установке **базового вектора IRQ** (номер прерывания ещё называется вектором). Например, если у первого PIC он равен 0x20, а у второго 0x40, то IRQ0 проецируется в прерывание процессора 0x20, IRQ1 в 0x21, IRQ8 в 0x40, IRQ9 в 0x41 и т. д. Для удобства можно объединить прерывания IRQ в один блок - например, поставив базовый вектор прерывания у первого PIC в 0x20, а у второго в 0x28, тогда IRQ0-IRQ15 будут отображены в прерывания 0x20-0x2F соответственно. Чтобы не затруднять преобразование номера IRQ в номер прерывания мы так и поступим. Исключения Не все прерывания пораждаются командой INT или сигналом от устройства, у процессора есть свои внутренние прерывания, которые служат реакцией на различные события. Как правило это ошибки - деление на ноль, некорректная инструкция, доступ к несуществующей памяти (при страничной адресации), нарушение прав доступа - и т. д. Такие прерывания позволяют ОС обработать различные ошибки в коде приложения. Всего системных прерываний 32 штуки - с нулевого по 31-ое. Не все из них сейчас имеют смысл, часть зарезервированы для будущего расширение архитектуры.  Одно из самых важных исключений защищённого режима - 0x13 - **General Protection Fault**. Оно вызывается при неверных правах доступа (код приложения пытается обратиться к ядру), при обнаружении ошибки в системных структурах (например, попытке доступа к некорректному дескриптору сегмента), и наконец оно вызывается в случае, если для нужного прерывания нету обработчика.  Ещё одно важное исключение - 0x14 - **General Page Fault** - обработка ошибки страничной адресации. Возникает при обращении к непримонтированной странице, либо при обращении к странице с не теми правами доступа. При этом с CR2 помещается адрес, который вызвал исключение. Это позволяет реализовать механизм подкачки - ядро может проанализировав адрес понять, что страница была выгружена в своп файл и подгрузить её. После возврата из этого исключения выполнение продолжится с той же самой инструкции, поэтому приложение даже ничего не заметит.  Таким образом прерывания с 0 до 0x1F не следует использовать для своих собственных обработчиков (как и для IRQ). Пока мы можем не создавать обработчики исключений, если ядро таки совершит ошибку процессор будет просто перезагружен. Таблица дескрипторов прерываний В реальном режиме таблица прерываний представляет собой простой массив из 256 двойных слов (младшее слово - сегмент, старшее - смещение), располагающийся в первом килобайте оперативной памяти. В защищённом режиме эта таблица приобретает более сложный формат - это массив из 256 8-байтных дескрипторов, помимо адреса и селектора сегментна дескриптор содержит различные атрибуты прерывания. Можно представить дескриптор в виде такой структуры на Си:  typedef struct {  uint16 address\_0\_15;  uint16 selector;  uint8 reserved;  uint8 type;  uint16 address\_16\_31;  } IntDesc;  Адрес обработчика хранится в виде двух частей - младшая половина в address\_0\_15, а старшая в address\_16\_31. Селектор в selector, поле reserved зарезервировано и должно быть равно нулю, а type обозначает тип обработчика. Пока нам хватит типа 0x8E, который отлично подходит для IRQ.  В отличии от реального режима, где адрес таблицы строго фиксирован (адреса 0x0000 - 0x03FF), в защищённом режиме адрес таблицы дескрипторов прерываний задаётся системой с помощью команды LIDT (Load Interrupt Descriptor Table), аналогично адресу GDT. Размер таблицы в данном случае должен быть 256 \* 8 = 2048 байт. Простейшая обработка прерываний Реализуем простейшую обработку прерываний - загрузим значение в IDTR и разрешим прерывания от PIC (команда STI). На самом деле этого мало - нужно как минимум обрабатывать IRQ0, на котором висит таймер, генерирующий прерывания примерно 18,5 раз в секунду (он нужен для реализации вытесняющей многозадачности и подсчёта времени). Также нам не помешает перенастроить контроллеры PIC на новые адреса - по умолчанию IRQ0 приходит на 8-ое прерывание, но в защищённом режиме первые 32 прерывания заняты системными, поэтому лучше вынести IRQ повыше - пусть это будет блок адресов с 0x20 до 0x2F. Для начала напишем заголовочный файл interrupts.h с описанием прототипов функций:  #ifndef INTERRUPTS\_H  #define INTERRUPTS\_H  uint8 irq\_base;  uint8 irq\_count;  void init\_interrupts();  void set\_int\_handler(uint8 index, void \*handler, uint8 type);  #endif  Помимо пары функций с говорящими названиями, мы описываем две переменных, служащих для определения другими модулями базового адреса и количества IRQ-прерываний, чтобы потом легко добавить поддержку APIC.  Файл interrupts.c будет начинаться с подключения необходимых заголовочных файлов и описания двух полезных типов данных, а также указателя на таблицу прерываний (разместим её перед стеком ядра):  #include "stdlib.h"  #include "interrupts.h"  typedef struct {  uint16 address\_0\_15;  uint16 selector;  uint8 reserved;  uint8 type;  uint16 address\_16\_31;  } \_\_attribute\_\_((packed)) IntDesc;  typedef struct {  uint16 limit;  void \*base;  } \_\_attribute\_\_((packed)) IDTR;  IntDesc \*idt = (void\*)0xFFFFC000;  У обеих структур присутствует специальный атрибут packed, который отключает для них выравнивание. Выравнивание нужно для ускорения доступа к памяти (лучше когда адрес скалярной переменной кратен её размеру), но в случае со служебными структурами не допустимо, чтобы смещения полей отклонялись от заданных. Например, без этого атрибута структура IDTR будет занимать не 6, а 8 байт, потому что base будет выравнен на 4 байта (сейчас он выравнен лишь на 2, хотя занимает 4). В результате структура не будет корректной. К сожалению, каждый компилятор предоставляет возможность отключить выравнивание разными способами, я показал вариант для GCC и MinGW, но он не подойдёт, например, для компилятора MSVC, для которого существует свой синтаксис.  Сначала напишем более простую функцию - set\_int\_handler:  void set\_int\_handler(uint8 index, void \*handler, uint8 type) {  asm("pushf \n cli");  idt[index].selector = 8;  idt[index].address\_0\_15 = (size\_t)handler & 0xFFFF;  idt[index].address\_16\_31 = (size\_t)handler >> 16;  idt[index].type = type;  idt[index].reserved = 0;  asm("popf");  }  Помимо собственно заполнения элемента таблицы прерываний эта функция отключает приём прерываний на время своей работы - вдруг прерывание произойдёт во время изменения его дескриптора, в этом случае результат может быть непредсказуемым (от простого исключения до передачи управления по произвольному адресу). Инструкция PUSHF сохраняет в стек старое значение регистра файлов, в POPF восстанавливает. Поскольку флаг обработки прерывания содержится в нём (и именно его меняют STI и CLI) мы восстановим старое значение обработки прерываний - если вызвать set\_int\_handler при запрещённых прерываниях, он не разрешит их.  Начнём писать функции init\_interrupts - первым делом надо спроецировать таблицу прерываний в виртуальное адресное пространство и загрузить правильное значение в IDTR:  void init\_interrupts() {  \*((size\_t\*)0xFFFFEFF0) = 0x8000 | 3;  memset(idt, 0, 256 \* sizeof(IntDesc));  IDTR idtr = {256 \* sizeof(IntDesc), idt};  asm("lidt (,%0,)"::"a"(&idtr));  Когда мы напишем простейший менеджер памяти, нам больше не придётся вручную делать так. а лишь вызвать специальную функцию, которая сама найдёт свободные физические страницы (сейчас мы выделили последнюю известную нам не нужную страницу - на ней раньше был кусок начального загрузчика) и модифицирует таблицы страниц.  Итак, таблица прерываний спроецирована и очищена. Теперь можно настроить PIC на правильный базовый вектор прерываний:  irq\_base = 0x20;  irq\_count = 16;  outportb(0x20, 0x11);  outportb(0x21, irq\_base);  outportb(0x21, 4);  outportb(0x21, 1);  outportb(0xA0, 0x11);  outportb(0xA1, irq\_base + 8);  outportb(0xA1, 2);  outportb(0xA1, 1);  Работа с первым PIC производится через порты ввода-вывода 0x20 и 0x21, а со вторым 0xA0 и 0xA1. После этих строк IRQ0-15 проецируются в прерывания процессора 0x20-0x2F.  Осталось назначить обработчик прерывания таймера и разрешить прерывания:  set\_int\_handler(irq\_base, timer\_int\_handler, 0x8E);  asm("sti");  }  На этом инициализация подсистемы обработки прерываний завершена, но мы ещё не написали обработчик прерывания таймера timer\_int\_handler. Вот тут первый раз появляется проблема связанная с тем, что мы используем язык высокого уровня - как описать функцию с произвольным кодом пролога и эпилога? Обычная функция в Си транслируется в Ассемблерный код примерно так:  function\_name:  push ebp  mov ebp, esp  ...  leave ; Равносильно mov esp, ebp и pop ebp  ret  Вместо iret все функции возвращаются с помощью команды ret. К тому же не сохраняют значения многих регистров. Пока временно решим проблему специальным макросом, но когда мы дойдём до реализации многозадачности вновь столкнёмся с этой проблемой (нам будет нужна предсказуемость содержимого стека, а в зависимости от опций оптимизации компилятор может как выполнять push ebp, так и не выполнять его). Этот макрос опишем в файле interrupts.h, чтобы можно было создавать обработчики прерываний и в других модулях ядра (например, обработчик прерывания клавиатуры будет в tty.c):  #ifndef INTERRUPTS\_H  #define INTERRUPTS\_H  uint8 irq\_base;  uint8 irq\_count;  **#define IRQ\_HANDLER(name) void name(); \**  **asm(#name ": pusha \n call \_" #name " \n movb $0x20, %al \n outb %al, $0x20 \n outb %al, $0xA0 \n popa \n iret"); \**  **void \_ ## name()**  void init\_interrupts();  void set\_int\_handler(uint8 index, void \*handler, uint8 type);  #endif  На самом деле этот макрос сначала описывает функцию обработки IRQ на чистом Assembler - обработчик сохраняет все регистры процессора в стек с помощью команды PUSHA, затем выполняет вызов функции на Си, потом отсылает байт 0x20 в порты контроллеров PIC (контроллер ждёт сигнала End Of Interrupt (EOI) - до его прихода другие IRQ обработаны не будут, это сделано для того, чтобы два одновременно пришедших прерывания обслуживались по очереди) и наконец восстанавливает регистры и выходит из обработчика с помощью команды POPA и IRET соответственно.  Теперь мы можем описать обработчик таймерного прерывания в interrupts.c:  IRQ\_HANDLER(timer\_int\_handler) {  (\*((char\*)(0xB8000 + 79 \* 2)))++;  }  Можно было оставить обработчик пустым, но я добавил туда увеличение на 1 кода символа в правом верхнем углу текстового экрана для наглядности работы. Осталось добавить вызов функции инициализации обработки прерываний в kernel\_main:  #include "stdlib.h"  **#include "interrupts.h"**  #include "tty.h"  typedef struct {  uint64 base;  uint64 size;  } BootModuleInfo;  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  **init\_interrupts();**  init\_tty();  set\_text\_attr(15);  printf("Welcome to MyOS!\n");  }  Вызывать эту функцию надо до init\_tty, потому что в дальнейшем мы добавим туда установку обработчика клавиатуры, а init\_interrupts не нуждается в поддержке телетайпа.  Теперь можно скомпилировать очередную версию системы и запустить - в углу экрана мы увидим постоянно меняющийся символ - доказательство того, что таймер работает. Если нажать на любую клавишу система упадёт, потому что мы не сделали обработчик клавиатуры, но это уже хорошо. Следует отметить, что смена символа происходит как бы в фоновом режиме - мы можем выполнять в ядре другую полезную работу, но это никак не отразится на обновлении экрана. Главное не помещать в прерывания слишком ресурсоёмкие процедуры, потому что это скажется на производительности всей системы не лучшим образом.  Полный листинг файла interrupts.c:  #include "stdlib.h"  #include "interrupts.h"  typedef struct {  uint16 address\_0\_15;  uint16 selector;  uint8 reserved;  uint8 type;  uint16 address\_16\_31;  } \_\_attribute\_\_((packed)) IntDesc;  typedef struct {  uint16 limit;  void \*base;  } \_\_attribute\_\_((packed)) IDTR;  IntDesc \*idt = (void\*)0xFFFFC000;  void timer\_int\_handler();  void init\_interrupts() {  \*((size\_t\*)0xFFFFEFF0) = 0x8000 | 3;  memset(idt, 0, 256 \* sizeof(IntDesc));  IDTR idtr = {256 \* sizeof(IntDesc), idt};  asm("lidt (,%0,)"::"a"(&idtr));  irq\_base = 0x20;  irq\_count = 16;  outportb(0x20, 0x11);  outportb(0x21, irq\_base);  outportb(0x21, 4);  outportb(0x21, 1);  outportb(0xA0, 0x11);  outportb(0xA1, irq\_base + 8);  outportb(0xA1, 2);  outportb(0xA1, 1);  set\_int\_handler(irq\_base, timer\_int\_handler, 0x8E);  asm("sti");  }  void set\_int\_handler(uint8 index, void \*handler, uint8 type) {  asm("pushf \n cli");  idt[index].selector = 8;  idt[index].address\_0\_15 = (size\_t)handler & 0xFFFF;  idt[index].address\_16\_31 = (size\_t)handler >> 16;  idt[index].type = type;  idt[index].reserved = 0;  asm("popf");  }  IRQ\_HANDLER(timer\_int\_handler) {  (\*((char\*)(0xB8000 + 79 \* 2)))++;  } Работа с контроллером клавиатуры На этом выпуск в общем-то можно было закончить, но я уже давно обещал реализацию ввода с клавиатуры, поэтому придётся продолжить :-)  Клавиатура сообщает о нажатии на клавишу с помощью IRQ1, который нам и нужно обрабатывать. При этом код нажатой клавиши доступен через порт 0x60. После чтения символа надо установить младший бит в содержимом порта 0x61, чтобы сообщить клавиатуре о готовности принять от неё следующий символ (аналогично PIC клавиатура старается помочь не дать скорости поступления новых данных превысить скорость обработки их процессором образуя очередь символов. Но внутренний буфер клавиатуры не бесконечен и если пользователь будет продолжать ввод, а ОС не будет устанавливать бит, то он переполнится и самые старые символы будут потеряны).  Из порта 0x60 доступен но ASCII-код символа, а так называемый скан-код клавиши, его необходимо будет преобразовать по таблице соответствий (а также в зависимости от состояния - например, если нажат Shift надо использовать большие буквы и т. д.). Преобразованием кодов мы займёмся в следующем выпуске, а сейчас лишь напишем базовый обработчик. Сначала добавим установку обработчика в init\_tty в tty.c:  #include <stdarg.h>  #include "stdlib.h"  **#include "interrupts.h"**  #include "tty.h"  ...  **void keyboard\_int\_handler();**  void init\_tty() {  tty\_buffer = (void\*)0xB8000;  tty\_width = \*((uint16\*)0x44A);  tty\_height = 25;  tty\_io\_port = \*((uint16\*)0x463);  cursor = (\*((uint8\*)0x451)) \* tty\_width + (\*((uint8\*)0x450));  text\_attr = 7;  **set\_int\_handler(irq\_base + 1, keyboard\_int\_handler, 0x8E);**  }  Обработчик будет простым - просто выведем на экран сообщение с кодом символа. Вот так (это следует дописать в конце файла tty.c):  IRQ\_HANDLER(keyboard\_int\_handler) {  uint8 key\_code;  inportb(0x60, key\_code);  printf("You pressed key with code %d\n", key\_code);  uint8 status;  inportb(0x61, status);  status |= 1;  outportb(0x61, status);  }  Попутно стоит исправить мою опечатку в stdlib.h:  ...  #define outportw(port, value) asm("outw %w0, %w1"::"a"(value),"d"(port));  #define outportl(port, value) asm("outl %0, %w1"::"a"(value),"d"(port));  #define inportb(port, out\_value) asm("inb %w1, %b0":"=a"(**out\_**value):"d"(port));  #define inportw(port, out\_value) asm("inw %w1, %w0":"=a"(**out\_**value):"d"(port));  #define inportl(port, out\_value) asm("inl %w1, %0":"=a"(**out\_**value):"d"(port));  void memset(void \*mem, char value, size\_t count);  void memset\_word(void \*mem, uint16 value, size\_t count);  ...  Ну вот и всё. Как обычно, компилируем систему. загружаемся в Bochs или на реальном железе и тестируем. Теперь при нажатии на любую клавишу будет выводиться строка на экран. Заметьте, отпускание клавиши тоже порождает приход прерывания, только прочитанный код клавиши на 0x80 больше, чем код нажатия. Проверка старшего бита скан-кода позволяет следить за тем какие клавиши нажаты в данный момент. Примерный результаты работы представлен на иллюстрации ниже: Заключение На этом, уважаемые читатели, всё! В следующем выпуске поговорим про преобразование скан-кодов в нормальные символы, разработку функции ввода строки с клавиатуры, а также начнём писать менеджер памяти для нашей ОС.  Если у вас возникли какие-либо вопросы по этому или предыдущим выпускам, а также по разработке ОС в целом, вы всегда можете задать их мне по электронной почте на адрес [kiv.apple@gmail.com](mailto:apple@gmail.com). До встречи! |

## Пишем свою операционную систему. Работа с клавиатурой

|  |
| --- |
| Доброго времени суток!  До предыдущего выпуска пользователь не как не мог взаимодействовать с нашей операционной системой, лишь наблюдать её работу. Теперь она уже умеет как-то реагировать на нажатия клавиш, отображая текстовые сообщения, но этого мало для нормального ввода данных. Однако сначала приведу Makefile для предыдущего выпуска:  ifdef OS  LDFLAGS = -mi386pe  else  LDFLAGS = -melf\_i386  endif  CFLAGS = -m32 -ffreestanding  all: script.ld startup.o stdlib.o main.o **interrupts.o** tty.o  ld $(LDFLAGS) -T script.ld -o kernel.bin startup.o stdlib.o main.o **interrupts.o** tty.o  objcopy kernel.bin -O binary  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  stdlib.o: stdlib.c stdlib.h  gcc -c $(CFLAGS) -o stdlib.o stdlib.c  main.o: main.c stdlib.h interrupts.h tty.h  gcc -c $(CFLAGS) -o main.o main.c  **interrupts.o: interrupts.c interrupts.h stdlib.h**  **gcc -c $(CFLAGS) -o interrupts.o interrupts.c**  tty.o: tty.c tty.h stdlib.h  gcc -c $(CFLAGS) -o tty.o tty.c  clean:  rm -v \*.o kernel.bin Буфер клавиатуры Обычно драйвер клавиатуры создаёт программный буфер нажатых клавиш. Если приложение хочет получить ввод с клавиатуры, оно читает содержимое буфера или ждёт пока в нём появится код клавиши. Преобразование из скан-кода в код символа также возложим на функцию чтения символа из буфера, чтобы упростить обработчик прерывания. Начнём с описания данных:  #include <stdarg.h>  #include "stdlib.h"  #include "interrupts.h"  #include "tty.h"  ...  #define KEY\_BUFFER\_SIZE 16  char key\_buffer[KEY\_BUFFER\_SIZE];  unsigned int key\_buffer\_head = 0;  unsigned int key\_buffer\_tail = 0;  В буфер помещаются 16 последних нажатых клавиш, если их не считывать самые старые будут теряться, всё так и должно быть. Для организации списка используется указатель на "голову" и на "хвост" списка кодов клавиш. При помещении очередного символа в буфер "хвост" увеличивается на 1. При чтении символа из буфера "голова" тоже сдвигается на 1. Если голова или хвост достигают конца массива, они становятся равными нулю. Таким образом получается так называемый **кольцевой буфер**. BIOS и DOS используют такой же, и в нашем случае нет смысла придумывать что-то сложнее. Помимо прочего такая структура устойчива к той ситуации, когда запись в буфер по прерыванию происходит в середине функции чтения.  Изменим обработчик прерывания IRQ1, убрав вывод сообщений и добавив запись символа в буфер:  IRQ\_HANDLER(keyboard\_int\_handler) {  uint8 key\_code;  **if (key\_buffer\_tail >= KEY\_BUFFER\_SIZE) {**  **key\_buffer\_tail = 0;**  **}**  **key\_buffer\_tail++;**  **key\_buffer[key\_buffer\_tail - 1] = key\_code;**  uint8 status;  inportb(0x61, status);  status |= 1;  outportb(0x61, status);  } Чтение символов и строк Добавим в заголовочный файл tty.h пару новых функций:  #ifndef TTY\_H  #define TTY\_H  **#include "stdlib.h"**  void init\_tty();  void out\_char(char chr);  void out\_string(char \*str);  void clear\_screen();  void set\_text\_attr(char attr);  void move\_cursor(unsigned int pos);  void printf(char \*fmt, ...);  **char in\_char(bool wait);**  **void in\_string(char \*buffer, size\_t buffer\_size);**  #endif  Первая функция позволяет читать одиночный символ с клавиатуры, вторая - считать целую строку символов. Замечу, что вторая функция имеет защиту от переполнение буфера с помощью ограничения длины вводимой строки.  Чтобы отделить преобразование символов от работы с буфером вынесем чтение скан-кода в отдельную функцию, которую добавим в конце tty.c:  uint8 in\_scancode() {  uint8 result;  if (key\_buffer\_head != key\_buffer\_tail) {  if (key\_buffer\_head >= KEY\_BUFFER\_SIZE) {  key\_buffer\_head = 0;  }  result = key\_buffer[key\_buffer\_head];  key\_buffer\_head++;  } else {  result = 0;  }  return result;  }  Эта функция возвращает скан-код нажатой клавиши или 0, если буфер пуст.  Преобразовывать скан-код в символ придётся по специальной таблице, которую опишем в дополнительном файле scancodes.h:  #ifndef SCANCODES\_H  #define SCANCODES\_H  char scancodes[] = {  0,  0, // ESC  '1','2','3','4','5','6','7','8','9','0', '-', '=',  8, // BACKSPACE  '\t', // TAB  'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']',  '\n', // ENTER  0, // CTRL  'a', 's', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'', '`',  0, // LEFT SHIFT  '\\', 'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/',  0, // RIGHT SHIFT  '\*', // NUMPAD  0, // ALT  ' ', // SPACE  0, // CAPSLOCK  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // F1 - F10  0, // NUMLOCK  0, // SCROLLLOCK  0, // HOME  0,  0, // PAGE UP  '-', // NUMPAD  0, 0,  0,  '+', // NUMPAD  0, // END  0,  0, // PAGE DOWN  0, // INS  0, // DEL  0, // SYS RQ  0,  0, 0, // F11 - F12  0,  0, 0, 0, // F13 - F15  0, 0, 0, 0, 0, 0, 0, 0, 0, // F16 - F24  0, 0, 0, 0, 0, 0, 0, 0  };    char scancodes\_shifted[] = {  0,  0, // ESC  '!','@','#','$','%','^','&','\*','(',')', '\_', '+',  8, // BACKSPACE  '\t', // TAB  'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', '{', '}',  '\n', // ENTER  0, // CTRL  'A', 'S', 'F', 'G', 'H', 'J', 'K', 'L', ':', '"', '~',  0, // LEFT SHIFT  '|', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', '<', '>', '?',  0, // RIGHT SHIFT  '\*', // NUMPAD  0, // ALT  ' ', // SPACE  0, // CAPSLOCK  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // F1 - F10  0, // NUMLOCK  0, // SCROLLLOCK  0, // HOME  0,  0, // PAGE UP  '-', // NUMPAD  0, 0,  0,  '+', // NUMPAD  0, // END  0,  0, // PAGE DOWN  0, // INS  0, // DEL  0, // SYS RQ  0,  0, 0, // F11 - F12  0,  0, 0, 0, // F13 - F15  0, 0, 0, 0, 0, 0, 0, 0, 0, // F16 - F24  0, 0, 0, 0, 0, 0, 0, 0  };  #endif  Эту таблицу с небольшими изменениями я позаимствовал отсюда: <http://sysbin.com/files/lowlevel/osdev11.htm#keyboard>. По сути у нас две таблицы - одна обычная, другая для символов, при вводе которых был зажат Shift. Теперь мы можем написать и in\_char:  char in\_char(bool wait) {  static bool shift = false;  uint8 chr;  do {  chr = in\_scancode();  switch (chr) {  case 0x2A:  case 0x36:  shift = true;  break;  case 0x2A + 0x80:  case 0x36 + 0x80:  shift = false;  break;  }  if (chr & 0x80) {  chr = 0;  }  if (shift) {  chr = scancodes\_shifted[chr];  } else {  chr = scancodes[chr];  }  } while (wait && (!chr));  return chr;  }  Эта функция уже умеет ждать нажатий при необходимости, а также преобразовывать скан-коды в символы с учётом нажатости клавиши Shift.  Сделать функцию in\_string не составит особого труда:  void in\_string(char \*buffer, size\_t buffer\_size) {  char chr;  size\_t position = 0;  do {  chr = in\_char(true);  switch (chr) {  case 0:  break;  case 8:  if (position > 0) {  position--;  out\_char(8);  }  break;  case '\n':  out\_char('\n');  break;  default:  if (position < buffer\_size - 1) {  buffer[position] = chr;  position++;  out\_char(chr);  }  }  } while (chr != '\n');  buffer[position] = 0;  }  Ну вот и готова поддержка ввода текста, теперь можно написать в kernel\_main что-нибудь вроде этого:  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  init\_interrupts();  init\_tty();  set\_text\_attr(15);  printf("Welcome to MyOS!\n");  **while (true) {**  **char buffer[256];**  **out\_string("Enter string: ");**  **in\_string(buffer, sizeof(buffer));**  **printf("You typed: %s\n", buffer);**  **}**  }  При условии, что в stdlib.c у нас есть strcmp, ничто не мешает реализовать что-нибудь вроде командного интерпретатора и сделать ОС немного интерактивной и полезной. Заключение Теперь у нас есть достаточные возможности для поддержания диалога с пользователем, во всяком случае вы наглядно видите результат своей работы. Далее пойдут более теоретические части. В первую очередь - менеджер памяти. Кстати, дальше того, что мы сделали сейчас, ни одна рассылка не продвинулась (разве что в одной была сделана на скорую руку многозадачность), так что теперь мы не будем никого повторять и идти по своему собственному уникальному пути. До встречи!  P.S.: Пожалуйста, пишите свои замечания и предложения по поводу постройки архитектуры системы. Также было бы не плохо выбрать более красивое название. Мой адрес, как обычно, [kiv.apple@gmail.com](mailto:apple@gmail.com). |

Пишем свою операционную систему. Менеджер памяти

|  |
| --- |
| Пришло время начать писать менеджер памяти - важнейший компонент любой операционной системы. Однако до его реализации в коде почему-то не дошла ни одна из известных мне рассылок.  Напишем заголовочный файл memory\_manager.h, обозначив задачи на этот выпуск:  #ifndef MEMORY\_MANAGER\_H  #define MEMORY\_MANAGER\_H  #include "stdlib.h"  #define PAGE\_SIZE 0x1000  #define PAGE\_OFFSET\_BITS 12  #define PAGE\_OFFSET\_MASK 0xFFF  #define PAGE\_TABLE\_INDEX\_BITS 10  #define PAGE\_TABLE\_INDEX\_MASK 0x3FF  #define PHYADDR\_BITS 32  #define PAGE\_VALID 1  #define PAGE\_WRITABLE 2  #define PAGE\_USER 4  typedef size\_t phyaddr;  #define KERNEL\_BASE 0xFFC00000  #define KERNEL\_PAGE\_TABLE 0xFFFFE000  #define TEMP\_PAGE 0xFFFFF000  #define TEMP\_PAGE\_INFO (KERNEL\_PAGE\_TABLE + ((TEMP\_PAGE >> PAGE\_OFFSET\_BITS) & PAGE\_TABLE\_INDEX\_MASK) \* sizeof(phyaddr))  phyaddr kernel\_page\_dir;  size\_t memory\_size;  void init\_memory\_manager(void \*memory\_map);  void temp\_map\_page(phyaddr addr);  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr);  size\_t get\_free\_memory\_size();  phyaddr alloc\_phys\_pages(size\_t count);  void free\_phys\_pages(phyaddr base, size\_t count);  #endif  Помимо описания функций в этом файле есть и описания адресов и размеров некоторых структур работы с виртуальной памятью, а также некоторые доступные флаги для проекции страниц (PAGE\_VALID - обязательный флаг, PAGE\_WRITABLE - страница доступна для записи, PAGE\_USER - страница доступа для непривилегированного кода).  Начнём писать memory\_manager.c с простой функции инициализации:  #include "stdlib.h"  #include "memory\_manager.h"  typedef struct {  uint64 base;  uint64 length;  uint32 type;  uint32 acpi\_ext\_attrs;  } \_\_attribute\_\_((packed)) MemoryMapEntry;  size\_t free\_page\_count;  phyaddr free\_phys\_memory\_pointer;  void init\_memory\_manager(void \*memory\_map) {  asm("movl %%cr3, %0":"=a"(kernel\_page\_dir));  memory\_size = 0x100000;  free\_page\_count = 0;  free\_phys\_memory\_pointer = 0;  MemoryMapEntry \*entry;  for (entry = memory\_map; entry->type; entry++) {  if ((entry->type == 1) && (entry->base >= 0x100000)) {  free\_phys\_pages(entry->base, entry->length >> PAGE\_OFFSET\_BITS);  memory\_size += entry->length;  }  }  }  Эта функция сохраняет текущее значение CR3 в переменную kernel\_page\_dir (потом нам пригодится для вызова map\_pages), а также интерпретирует карту памяти, полученную когда-то от BIOS. Для всех блоков памяти, которые выше 1-ого мегабайта, а также доступны для использования (тип 1) вызывается функция free\_phys\_pages, которая должна помечать указанный регион физической памяти как свободный. Параллельно с этим init\_memory\_manager вычисляет полный объём оперативной памяти, который будет доступен в глобальной переменной memory\_size.  Теперь напишем пару маленьких функций, одна из которых полезна для других модулей, а вторая вообще будет одной из самых часто используемых:  void temp\_map\_page(phyaddr addr) {  ((phyaddr\*)TEMP\_PAGE\_INFO) = (page & ~PAGE\_OFFSET\_MASK) | PAGE\_VALID | PAGE\_WRITABLE;  asm("invlpg (,%0,)"::"a"(TEMP\_PAGE));  }  size\_t get\_free\_memory\_size() {  return free\_page\_count << PAGE\_OFFSET\_BITS;  }  Самая главная функция - temp\_map\_page - она позволяет спроецировать любую физическую страницу по адресу TEMP\_PAGE, через это "окно" многие другие функции обращаются к неспроецированным страницам, как это работает мы очень скоро увидим. Вторая функция нужна другим модулям, чтобы получить размер свободной незанятой памяти в байтах, а не страницах.  После того как написана функция temp\_map\_page, мы можем сделать и две других - map\_pages и get\_page\_info. Первая позволяет спроецировать указанный блок физических страниц по нужному виртуальному адресу, а вторая узнать куда же будет отображён виртуальный адрес и отображён ли он.  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {  for (; count; count--) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & PAGE\_VALID)) {  phyaddr addr = alloc\_phys\_pages(1);  if (addr) {  temp\_map\_page(addr);  memset((void\*)TEMP\_PAGE, 0, PAGE\_SIZE);  temp\_map\_page(page\_table);  ((phyaddr\*)TEMP\_PAGE)[index] = addr | PAGE\_VALID | PAGE\_WRITABLE | PAGE\_USER;  page\_table = addr;  } else {  return false;  }  }  } else {  ((phyaddr\*)TEMP\_PAGE)[index] = (paddr & ~PAGE\_OFFSET\_BITS) | flags;  asm("invlpg (,%0,)"::"a"(vaddr));  }  }  vaddr += PAGE\_SIZE;  paddr += PAGE\_SIZE;  }  return true;  }  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & PAGE\_VALID)) {  return 0;  }  } else {  return ((phyaddr\*)TEMP\_PAGE)[index];  }  }  }  Теперь у нас есть функции для проецирования страниц в виртуальное адресное пространство, а также для получения информации о виртуальной странице. Хотя пока у нас нет функций для выделения и освобождения физических страниц, map\_pages и get\_page\_info уже будут работать, пока нет необходимости создавать новую таблицу страниц (то есть мы можем проецировать страницы в первые и последние 4 МБ).  Управление физическими страницами требует отдельного обсуждения и если менеджер физической памяти описать в этом же выпуске, он получится слишком большим, поэтому пока сделаем лишь функции-заглушки, а полный код рассмотрим в следующий раз:  phyaddr alloc\_phys\_pages(size\_t count) {  return 0;  }  void free\_phys\_pages(phyaddr base, size\_t count) {    }  Теперь мы можем изменить выделение памяти под таблицу прерываний в функции init\_interrupts файла interrupts.c на более удобное:  #include "stdlib.h"  **#include "memory\_manager.h"**  #include "interrupts.h"  typedef struct {  uint16 address\_0\_15;  uint16 selector;  uint8 reserved;  uint8 type;  uint16 address\_16\_31;  } \_\_attribute\_\_((packed)) IntDesc;  typedef struct {  uint16 limit;  void \*base;  } \_\_attribute\_\_((packed)) IDTR;  IntDesc \*idt = (void\*)0xFFFFC000;  void timer\_int\_handler();  void init\_interrupts() {  **map\_pages(kernel\_page\_dir, idt, 0x8000, 1, PAGE\_VALID | PAGE\_WRITABLE);**  memset(idt, 0, 256 \* sizeof(IntDesc));  IDTR idtr = {256 \* sizeof(IntDesc), idt};  asm("lidt (,%0,)"::"a"(&idtr));  irq\_base = 0x20;  irq\_count = 16;  outportb(0x20, 0x11);  outportb(0x21, irq\_base);  outportb(0x21, 4);  outportb(0x21, 1);  outportb(0xA0, 0x11);  outportb(0xA1, irq\_base + 8);  outportb(0xA1, 2);  outportb(0xA1, 1);  set\_int\_handler(irq\_base, timer\_int\_handler, 0x8E);  asm("sti");  }  Осталось добавить код инициализации в kernel\_main (менеджер памяти следует инициализировать самым первым, до всех остальных подсистем ядра):  #include "stdlib.h"  **#include "memory\_manager.h"**  #include "interrupts.h"  #include "tty.h"  typedef struct {  uint64 base;  uint64 size;  } BootModuleInfo;  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  **init\_memory\_manager(memory\_map);**  init\_interrupts();  init\_tty();  set\_text\_attr(15);  printf("Welcome to MyOS!\n");  **printf("kernel\_page\_dir = 0x%x\n", kernel\_page\_dir);**  **printf("memory\_size = %d MB\n", memory\_size / 1024 / 1024);**  **printf("get\_page\_info(kernel\_page\_dir, 0xB8000) = 0x%x\n", get\_page\_info(kernel\_page\_dir, (void\*)0xB8000));**  }  Помимо собственно инициализации менеджера памяти этот код демонстрирует работу некоторых функций. На экране должно появится:  Welcome to MyOS!  kernel\_page\_dir = 0x1000  memory\_size = 63 MB  get\_page\_info(kernel\_page\_dir, 0xB8000) = 0xB8063  Для компиляции следует дописать Makefile:  ifdef OS  LDFLAGS = -mi386pe  else  LDFLAGS = -melf\_i386  endif  CFLAGS = -m32 -ffreestanding  all: script.ld startup.o stdlib.o main.o **memory\_manager.o** interrupts.o tty.o  ld $(LDFLAGS) -T script.ld -o kernel.bin startup.o stdlib.o main.o **memory\_manager.o** interrupts.o tty.o  objcopy kernel.bin -O binary  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  stdlib.o: stdlib.c stdlib.h  gcc -c $(CFLAGS) -o stdlib.o stdlib.c  main.o: main.c stdlib.h interrupts.h tty.h  gcc -c $(CFLAGS) -o main.o main.c  **memory\_manager.o: memory\_manager.c memory\_manager.h stdlib.h**  **gcc -c $(CFLAGS) -o memory\_manager.o memory\_manager.c**  interrupts.o: interrupts.c interrupts.h stdlib.h  gcc -c $(CFLAGS) -o interrupts.o interrupts.c  tty.o: tty.c tty.h stdlib.h  gcc -c $(CFLAGS) -o tty.o tty.c  clean:  rm -v \*.o kernel.bin  Вот и всё! В заключение приведу полный код memory\_manager.c:  #include "stdlib.h"  #include "memory\_manager.h"  typedef struct {  uint64 base;  uint64 length;  uint32 type;  uint32 acpi\_ext\_attrs;  } \_\_attribute\_\_((packed)) MemoryMapEntry;  size\_t free\_page\_count = 0;  phyaddr free\_phys\_memory\_pointer = 0;  void init\_memory\_manager(void \*memory\_map) {  asm("movl %%cr3, %0":"=a"(kernel\_page\_dir));  memory\_size = 0x100000;  MemoryMapEntry \*entry;  for (entry = memory\_map; entry->type; entry++) {  if ((entry->type == 1) && (entry->base >= 0x100000)) {  free\_phys\_pages(entry->base, entry->length >> PAGE\_OFFSET\_BITS);  memory\_size += entry->length;  }  }  }  void temp\_map\_page(phyaddr addr) {  \*((phyaddr\*)TEMP\_PAGE\_INFO) = (addr & ~PAGE\_OFFSET\_MASK) | PAGE\_VALID | PAGE\_WRITABLE;  asm("invlpg (,%0,)"::"a"(TEMP\_PAGE));  }  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {  for (; count; count--) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & PAGE\_VALID)) {  phyaddr addr = alloc\_phys\_pages(1);  if (addr) {  temp\_map\_page(paddr);  memset((void\*)TEMP\_PAGE, 0, PAGE\_SIZE);  temp\_map\_page(page\_table);  ((phyaddr\*)TEMP\_PAGE)[index] = addr | PAGE\_VALID | PAGE\_WRITABLE | PAGE\_USER;  page\_table = addr;  } else {  return false;  }  }  } else {  ((phyaddr\*)TEMP\_PAGE)[index] = (paddr & ~PAGE\_OFFSET\_BITS) | flags;  asm("invlpg (,%0,)"::"a"(vaddr));  }  }  vaddr += PAGE\_SIZE;  paddr += PAGE\_SIZE;  }  return true;  }  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & PAGE\_VALID)) {  return 0;  }  } else {  return ((phyaddr\*)TEMP\_PAGE)[index];  }  }  }  size\_t get\_free\_memory\_size() {  return free\_page\_count << PAGE\_OFFSET\_BITS;  }  phyaddr alloc\_phys\_pages(size\_t count) {  return 0;  }  void free\_phys\_pages(phyaddr base, size\_t count) {    } |

## Пишем свою операционную систему. Менеджер физической памяти

|  |
| --- |
| Доброго времени суток!  Сегодня в этом выпуске мы напишем реализацию менеджера физической памяти. Его задача - найти N свободных физических страниц, пометить их как занятые и отдать адрес первой. А также обратную операцию - пометка блока физических страниц как свободных. Мне известно 4 различных алгоритма:  1) **Стек страниц.** Создаётся специальная структура данных - массив физических адресов, количество элементов равно количеству страниц физической памяти. При 32-битном адресе это будет лишь 1/1024 всей памяти. Когда нужна очередная страница достаётся верхняя страница стека, когда какая-то страница не нужна, она кладётся на верх стека. Это самый быстрый и простой алгоритм, но он имеет существенный недостаток - невозможность выделения блока последовательных страниц. А между тем это очень нужно драйверам - устройства-то не знают ничего про страничную адресацию.  2) **Битовая карта страниц.** Аналогично битовой карте в ListFS. Битовая карта занимает 1/32768 физической памяти. Из минусов следует отметить то, что это достаточно медленный вариант, а для оперативной памяти немаловажна скорость выделения (для блоков на жёстком диске такой алгоритм вполне приемлем). К тому же появляются трудности, если память представляет собой не непрерывный блок адресов, а то и вовсе может динамически добавляться.  3) **Линейный список регионов памяти.** Можно создать массив, в котором будет содержаться начало, конец и тип (занято/свободно) участков памяти. Этот вариант обладает наибольшей скоростью, но достаточно сложен в реализации - список во время работы системы постоянно меняет свою длину и надо как-то освобождать и выделять память под него самого.  4) **Двунаправленный связанный список, располагающийся на свободных страницах памяти**. Этот вариант немного медленнее третьего, зато обладает достаточной гибкостью и простотой. Суть в том, что в начале свободного блока физической памяти создаётся структура, в которой есть адрес предыдущего свободного блока, адрес следующего и размер текущего блока. С помощью temp\_map\_page мы можем получить доступ к заголовку любого из блоков. Максимальная сложность alloc\_phys\_pages - O(N) (N - количество разрозненных свободных блоков). free\_phys\_pages должен не просто помечать блок как свободный, но и дефрагментировать блоки памяти - если перед и/или после свободного блока памяти впритык существует ещё один их следует объединить. Для упрощения задачи можно поддерживать упорядоченность блоков по возрастанию адресов, тогда сложность free\_phys\_pages будет так же O(N).  Мы будем использовать последний метод, потому что это разумный компромисс между скоростью, гибкостью и простотой. К тому же не требует дополнительной памяти на свои структуры, что тоже хорошо.  Физический адрес первого свободно блока хранится в переменной free\_phys\_memory\_pointer. Список двунаправленный для упрощения удаления и добавления элементов. Он ещё будет и кольцевым, чтобы не делать лишних проверок на NULL.  В первую очередь опишем структуру заголовка блока памяти (размер блока будем хранить в страницах, заодно, пусть некорректным физическим адресом будет -1, а не 0):  #include "stdlib.h"  #include "memory\_manager.h"  typedef struct {  uint64 base;  uint64 length;  uint32 type;  uint32 acpi\_ext\_attrs;  } \_\_attribute\_\_((packed)) MemoryMapEntry;  **typedef struct {**  **phyaddr next;**  **phyaddr prev;**  **size\_t size;**  **} PhysMemoryBlock;**  size\_t free\_page\_count = 0;  phyaddr free\_phys\_memory\_pointer = -1;  Теперь можно написать функцию выделения нового блока памяти из списка свободных:  phyaddr alloc\_phys\_pages(size\_t count) {  if (free\_page\_count < count) return -1;  phyaddr result = -1;  if (free\_phys\_memory\_pointer != -1) {  phyaddr cur\_block = free\_phys\_memory\_pointer;  do {  temp\_map\_page(cur\_block);  if (((PhysMemoryBlock\*)TEMP\_PAGE)->size == count) {  phyaddr next = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;  phyaddr prev = ((PhysMemoryBlock\*)TEMP\_PAGE)->prev;  temp\_map\_page(next);  ((PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  temp\_map\_page(prev);  ((PhysMemoryBlock\*)TEMP\_PAGE)->next = next;  if (cur\_block == free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = next;  if (cur\_block == free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = -1;  }  }  result = cur\_block;  break;  } else if (((PhysMemoryBlock\*)TEMP\_PAGE)->size > count) {  ((PhysMemoryBlock\*)TEMP\_PAGE)->size -= count;  result = cur\_block + (((PhysMemoryBlock\*)TEMP\_PAGE)->size << PAGE\_OFFSET\_BITS);  break;  }  cur\_block = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;    } while (cur\_block != free\_phys\_memory\_pointer);  if (result != -1) {  free\_page\_count -= count;  }  }  return result;  }  Функция работает достаточно просто - если список свободных блоков пуст, либо общий объём свободной памяти меньше нужного, то она тут же возвращает -1, иначе же начинает перебирать все свободные блоки. Если попадается блок, который равен по размеру запрошенному, то его адрес возвращается в качестве результата, а сам блок удаляется из списка (рассматривается вариант, когда это был первый блок в списке, тогда надо изменить free\_phys\_memory\_pointer, а также когда в списке больше не осталось элементов). Иначе, если блок был больше искомого, от его конца отрезается нужное число страниц и возвращается базовый адрес такого блока. Если в результате поиска блок нужного размера так и не был найден (хотя суммарно все регионы и подходят по размеру, но нет ни одного непрерывного блока нужной длинны), то функция завершается, возвращая -1. Для запросов небольшого количества страниц (чаще всего требуется 1 страница) эта функция вернётся уже после первой итерации цикла поиска.  Функция free\_phys\_pages сложнее, потому что её не только нужно вставить блок в нужное место в списке, но и по возможности слить его с другими. Возможно три варианта соседства блоков:  1) Новый блок после другого. Создавать новый блок не нужно, лишь увеличить size предыдущего. 2) Новый блок перед другим. Следует удалить следующий блок, а при создании предыдущего указать size больше. 3) Новый блок окружён двумя. Следует удалить следующих блок, а размер предыдущего увеличить на сумму размеров освобождаемого блока и следующего за ним.  Обработку этих ситуаций упростит упорядоченность списка блоков по базовым адресам.  void free\_phys\_pages(phyaddr base, size\_t count) {  if (free\_phys\_memory\_pointer == -1) {  temp\_map\_page(base);  ((PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  ((PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  ((PhysMemoryBlock\*)TEMP\_PAGE)->size = count;  free\_phys\_memory\_pointer = base;  } else {  phyaddr cur\_block = free\_phys\_memory\_pointer;  do {  temp\_map\_page(cur\_block);  if (cur\_block + (((PhysMemoryBlock\*)TEMP\_PAGE)->size << PAGE\_OFFSET\_BITS) == base) {  ((PhysMemoryBlock\*)TEMP\_PAGE)->size += count;  if (((PhysMemoryBlock\*)TEMP\_PAGE)->next == base + (count << PAGE\_OFFSET\_BITS)) {  phyaddr next1 = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;  temp\_map\_page(next1);  phyaddr next2 = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;  size\_t new\_count = ((PhysMemoryBlock\*)TEMP\_PAGE)->size;  temp\_map\_page(next2);  ((PhysMemoryBlock\*)TEMP\_PAGE)->prev = cur\_block;  temp\_map\_page(cur\_block);  ((PhysMemoryBlock\*)TEMP\_PAGE)->next = next2;  ((PhysMemoryBlock\*)TEMP\_PAGE)->size += new\_count;  }  break;  } else if (base + (count << PAGE\_OFFSET\_BITS) == cur\_block) {  size\_t old\_count = ((PhysMemoryBlock\*)TEMP\_PAGE)->size;  phyaddr next = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;  phyaddr prev = ((PhysMemoryBlock\*)TEMP\_PAGE)->prev;  temp\_map\_page(next);  ((PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  temp\_map\_page(prev);  ((PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  temp\_map\_page(base);  ((PhysMemoryBlock\*)TEMP\_PAGE)->next = next;  ((PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  ((PhysMemoryBlock\*)TEMP\_PAGE)->size = count + old\_count;  break;  } else if ((cur\_block > base) || (((PhysMemoryBlock\*)TEMP\_PAGE)->next == free\_phys\_memory\_pointer)) {  phyaddr prev = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;  ((PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  temp\_map\_page(prev);  ((PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  temp\_map\_page(base);  ((PhysMemoryBlock\*)TEMP\_PAGE)->next = cur\_block;  ((PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  ((PhysMemoryBlock\*)TEMP\_PAGE)->size = count;  break;  }  cur\_block = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;  } while (cur\_block != free\_phys\_memory\_pointer);  if (base < free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = base;  }    }  free\_page\_count += count;  }  Это, пожалуй, одна из самых запутанных функций нашего ядра, но я провёл различные проверки - она работает так, как нужно.  Итак, менеджер физической памяти готов, доведём до ума выделение памяти памяти под таблицу прерываний:  void init\_interrupts() {  map\_pages(kernel\_page\_dir, idt, **alloc\_phys\_pages(1)**, 1, PAGE\_VALID | PAGE\_WRITABLE);  memset(idt, 0, 256 \* sizeof(IntDesc));  IDTR idtr = {256 \* sizeof(IntDesc), idt};  asm("lidt (,%0,)"::"a"(&idtr));  irq\_base = 0x20;  irq\_count = 16;  outportb(0x20, 0x11);  outportb(0x21, irq\_base);  outportb(0x21, 4);  outportb(0x21, 1);  outportb(0xA0, 0x11);  outportb(0xA1, irq\_base + 8);  outportb(0xA1, 2);  outportb(0xA1, 1);  set\_int\_handler(irq\_base, timer\_int\_handler, 0x8E);  asm("sti");  }  На этом заканчиваю сегодняшний выпуск, разве, что приведу полный код memory\_manager.c, потому что мы его активно меняли:  #include "stdlib.h"  #include "memory\_manager.h"  typedef struct {  uint64 base;  uint64 length;  uint32 type;  uint32 acpi\_ext\_attrs;  } \_\_attribute\_\_((packed)) MemoryMapEntry;  typedef struct {  phyaddr next;  phyaddr prev;  size\_t size;  } PhysMemoryBlock;  size\_t free\_page\_count = 0;  phyaddr free\_phys\_memory\_pointer = **-1;**  void init\_memory\_manager(void \*memory\_map) {  asm("movl %%cr3, %0":"=a"(kernel\_page\_dir));  memory\_size = 0x100000;  MemoryMapEntry \*entry;  for (entry = memory\_map; entry->type; entry++) {  if ((entry->type == 1) && (entry->base >= 0x100000)) {  free\_phys\_pages(entry->base, entry->length >> PAGE\_OFFSET\_BITS);  memory\_size += entry->length;  }  }  }  /\* Physical memory manager \*/  void temp\_map\_page(phyaddr addr) {  \*((phyaddr\*)TEMP\_PAGE\_INFO) = (addr & ~PAGE\_OFFSET\_MASK) | PAGE\_VALID | PAGE\_WRITABLE;  asm("invlpg (,%0,)"::"a"(TEMP\_PAGE));  }  size\_t get\_free\_memory\_size() {  return free\_page\_count << PAGE\_OFFSET\_BITS;  }  **phyaddr alloc\_phys\_pages(size\_t count) {**  **if (free\_page\_count < count) return -1;**  **phyaddr result = -1;**  **if (free\_phys\_memory\_pointer != -1) {**  **phyaddr cur\_block = free\_phys\_memory\_pointer;**  **do {**  **temp\_map\_page(cur\_block);**  **if (((PhysMemoryBlock\*)TEMP\_PAGE)->size == count) {**  **phyaddr next = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;**  **phyaddr prev = ((PhysMemoryBlock\*)TEMP\_PAGE)->prev;**  **temp\_map\_page(next);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;**  **temp\_map\_page(prev);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->next = next;**  **if (cur\_block == free\_phys\_memory\_pointer) {**  **free\_phys\_memory\_pointer = next;**  **if (cur\_block == free\_phys\_memory\_pointer) {**  **free\_phys\_memory\_pointer = -1;**  **}**  **}**  **result = cur\_block;**  **break;**  **} else if (((PhysMemoryBlock\*)TEMP\_PAGE)->size > count) {**  **((PhysMemoryBlock\*)TEMP\_PAGE)->size -= count;**  **result = cur\_block + (((PhysMemoryBlock\*)TEMP\_PAGE)->size << PAGE\_OFFSET\_BITS);**  **break;**  **}**  **cur\_block = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;**    **} while (cur\_block != free\_phys\_memory\_pointer);**  **if (result != -1) {**  **free\_page\_count -= count;**  **}**  **}**  **return result;**  **}**  **void free\_phys\_pages(phyaddr base, size\_t count) {**  **if (free\_phys\_memory\_pointer == -1) {**  **temp\_map\_page(base);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->next = base;**  **((PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;**  **((PhysMemoryBlock\*)TEMP\_PAGE)->size = count;**  **free\_phys\_memory\_pointer = base;**  **} else {**  **phyaddr cur\_block = free\_phys\_memory\_pointer;**  **do {**  **temp\_map\_page(cur\_block);**  **if (cur\_block + (((PhysMemoryBlock\*)TEMP\_PAGE)->size << PAGE\_OFFSET\_BITS) == base) {**  **((PhysMemoryBlock\*)TEMP\_PAGE)->size += count;**  **if (((PhysMemoryBlock\*)TEMP\_PAGE)->next == base + (count << PAGE\_OFFSET\_BITS)) {**  **phyaddr next1 = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;**  **temp\_map\_page(next1);**  **phyaddr next2 = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;**  **size\_t new\_count = ((PhysMemoryBlock\*)TEMP\_PAGE)->size;**  **temp\_map\_page(next2);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->prev = cur\_block;**  **temp\_map\_page(cur\_block);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->next = next2;**  **((PhysMemoryBlock\*)TEMP\_PAGE)->size += new\_count;**  **}**  **break;**  **} else if (base + (count << PAGE\_OFFSET\_BITS) == cur\_block) {**  **size\_t old\_count = ((PhysMemoryBlock\*)TEMP\_PAGE)->size;**  **phyaddr next = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;**  **phyaddr prev = ((PhysMemoryBlock\*)TEMP\_PAGE)->prev;**  **temp\_map\_page(next);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;**  **temp\_map\_page(prev);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->next = base;**  **temp\_map\_page(base);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->next = next;**  **((PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;**  **((PhysMemoryBlock\*)TEMP\_PAGE)->size = count + old\_count;**  **break;**  **} else if ((cur\_block > base) || (((PhysMemoryBlock\*)TEMP\_PAGE)->next == free\_phys\_memory\_pointer)) {**  **phyaddr prev = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;**  **((PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;**  **temp\_map\_page(prev);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->next = base;**  **temp\_map\_page(base);**  **((PhysMemoryBlock\*)TEMP\_PAGE)->next = cur\_block;**  **((PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;**  **((PhysMemoryBlock\*)TEMP\_PAGE)->size = count;**  **break;**  **}**  **cur\_block = ((PhysMemoryBlock\*)TEMP\_PAGE)->next;**  **} while (cur\_block != free\_phys\_memory\_pointer);**  **if (base < free\_phys\_memory\_pointer) {**  **free\_phys\_memory\_pointer = base;**  **}**    **}**  **free\_page\_count += count;**  **}**  /\* Virtual memory manager \*/  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {  for (; count; count--) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & PAGE\_VALID)) {  phyaddr addr = alloc\_phys\_pages(1);  **if (addr != -1) {**  temp\_map\_page(paddr);  memset((void\*)TEMP\_PAGE, 0, PAGE\_SIZE);  temp\_map\_page(page\_table);  ((phyaddr\*)TEMP\_PAGE)[index] = addr | PAGE\_VALID | PAGE\_WRITABLE | PAGE\_USER;  page\_table = addr;  } else {  return false;  }  }  } else {  ((phyaddr\*)TEMP\_PAGE)[index] = (paddr & ~PAGE\_OFFSET\_BITS) | flags;  asm("invlpg (,%0,)"::"a"(vaddr));  }  }  vaddr += PAGE\_SIZE;  paddr += PAGE\_SIZE;  }  return true;  }  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & PAGE\_VALID)) {  return 0;  }  } else {  return ((phyaddr\*)TEMP\_PAGE)[index];  }  }  }  Теперь мы можем легко и просто управлять физической памятью - выделять блоки страниц, проецировать их, освобождать. Теперь необходимо придумать, как распоряжаться виртуальной памятью. Задачей менеджера виртуальной памяти будет являться управление виртуальным адресном пространстве на уровне страниц: Возможно, лучше понять, что я имею ввиду, поможет пара прототипов функций:  typedef struct {  phyaddr page\_dir;  void \*start;  void \*end;  } AddressSpace;  void \*alloc\_virt\_pages(AddressSpace \*as, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  void free\_virt\_pages(AddressSpace \*as, void \*vaddr, size\_t count, unsigned int flags);  Опишу параметры каждой из функций:  **alloc\_virt\_pages:** as - адресное пространство, в котором производится работа. Структура AddressSpace может содержать дополнительную информацию (нужную для поиска свободных страниц). У каждого процесса в системе свой AddressSpace (всегда нижние 2 ГБ), плюс у ядра свой AddressSpace общий для всех каталогов страниц (верхние 2 ГБ). Возвращённый функцией адрес должен принадлежать региону от as->start до as->end, либо NULL в случае ошибки. vaddr - Желаемый виртуальный адрес. Если равен NULL, функция должна найти сама свободный виртуальный адрес, куда можно примонтировать count страниц (это и есть самое сложное - как искать лучше), иначе функция должна спроецировать страницы по этому адресу при условии, что он не выходит за пределы AddressSpace (as->start <= vaddr < as->end) и на этом месте ничего не примонтировано до этого вызова. paddr - Желаемый физический адрес. Например, драйвер может захотеть примонтировать страницу 0xB8000 для работы с текстовым экраном. Если равно -1, функция должна вызвать alloc\_phys\_pages(count) и использовать его результат (если не произойдёт ошибки выделения памяти, тогда придётся вернуть NULL и ничего не проецировать). count - Количество страниц для проецирования. flags - Атрибуты доступа для проецирования. Аналогично одноимённому параметры map\_pages.  **free\_virt\_pages:** as - Аналогично alloc\_virt\_pages. vaddr - Виртуальный адрес блока страниц, который следует освободить. Должен принадлежать адресному пространству as и быть не равен NULL. count - Количество страниц для освобождения. flags - Флаги освобождения. Позволяет запретить функции освобождать страницы без флага PAGE\_USER. Нужно для того, чтобы не дать приложению уничтожить системные структуры.  Вот так. Свои предложения по механизму хранения свободных страниц и их поиска, можете присылать мне на почту: [kiv.apple@gmail.com](mailto:apple@gmail.com). До встречи! ;-) |

Пишем свою операционную систему. Немного улучшений менеджера памяти

|  |
| --- |
| В этом выпуске мы рассмотрим работу менеджера виртуальной памяти, а также улучшим некоторые другие части memory\_manager.c.  Заголовочный файл memory\_manager.h теперь будет выглядеть так:  #ifndef MEMORY\_MANAGER\_H  #define MEMORY\_MANAGER\_H  #include "stdlib.h"  #define PAGE\_SIZE 0x1000  #define PAGE\_OFFSET\_BITS 12  #define PAGE\_OFFSET\_MASK 0xFFF  #define PAGE\_TABLE\_INDEX\_BITS 10  #define PAGE\_TABLE\_INDEX\_MASK 0x3FF  #define PHYADDR\_BITS 32  **#define PAGE\_PRESENT (1 << 0)**  **#define PAGE\_WRITABLE (1 << 1)**  **#define PAGE\_USER (1 << 2)**  **#define PAGE\_WRITE\_THROUGH (1 << 3)**  **#define PAGE\_CACHE\_DISABLED (1 << 4)**  **#define PAGE\_ACCESSED (1 << 5)**  **#define PAGE\_MODIFIED (1 << 6)**  **#define PAGE\_GLOBAL (1 << 8)**  #define KERNEL\_BASE 0xFFC00000  #define KERNEL\_PAGE\_TABLE 0xFFFFE000  #define TEMP\_PAGE 0xFFFFF000  #define TEMP\_PAGE\_INFO (KERNEL\_PAGE\_TABLE + ((TEMP\_PAGE >> PAGE\_OFFSET\_BITS) & PAGE\_TABLE\_INDEX\_MASK) \* sizeof(phyaddr))  **#define USER\_MEMORY\_START ((void\*)0)**  **#define USER\_MEMORY\_END ((void\*)0x7FFFFFFF)**  **#define KERNEL\_MEMORY\_START ((void\*)0x80000000)**  **#define KERNEL\_MEMORY\_END ((void\*)(KERNEL\_BASE - 1))**  typedef size\_t phyaddr;  **typedef enum {**  **VMB\_RESERVED,**  **VMB\_MEMORY,**  **VMB\_IO\_MEMORY**  **} VirtMemoryBlockType;**  **typedef struct {**  **VirtMemoryBlockType type;**  **void \*base;**  **size\_t length;**  **} VirtMemoryBlock;**  **typedef struct {**  **phyaddr page\_dir;**  **void \*start;**  **void \*end;**  **size\_t block\_count;**  **VirtMemoryBlock \*blocks;**  **} AddressSpace;**  phyaddr kernel\_page\_dir;  size\_t memory\_size;  **AddressSpace kernel\_address\_space;**  void init\_memory\_manager(void \*memory\_map);  size\_t get\_free\_memory\_size();  phyaddr alloc\_phys\_pages(size\_t count);  void free\_phys\_pages(phyaddr base, size\_t count);  void temp\_map\_page(phyaddr addr);  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr);  **void \*alloc\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);**  **void free\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, size\_t count, unsigned int flags);**  #endif  Расскажу про назначение всех флагов для таблицы страниц:  **PAGE\_PRESENT** - Страница присутствует в оперативной памяти, т.е. спроецированна. Если этого флага нет, остальные биты даже не анализируются, а сразу генерируется исключение **Page Fault** (#14). **PAGE\_WRITABLE** - Страница доступна для записи. Если этого флага нет, то любая команда записи по этому адресу приведёт к **Page fault**. **PAGE\_USER** - Страница доступна для пользователя. Если этого флага нет, то любая команда доступа (не важно чтения или записи) к этой странице из кода с CPL = 3 спровоцирует **Page Fault**. **PAGE\_WRITE\_THROUGH** - Управлением кешем: разрешение сквозной записи. **PAGE\_CACHE\_DISABLED** - Управлением кешем: кеширование запрещено **PAGE\_ACCESSED** - К странице было обращение. Процессор лишь устанавливает этот флаг, но не сбрасывает его. Это задача самой ОС, если она использует этот бит для определения какие страницы редко используется, чтобы сбросить их в файл подкачки на диск.  Все предыдущие флаги применимы как к элементу каталога страниц (тогда это будут атрибуты целой таблицы, а не отдельной страницы), так и к элементу таблицы страниц. Следующие два флага касаются лишь элемента таблицы страниц:  **PAGE\_MODIFIED** - Содержимое страницы было изменено. **PAGE\_ACCESSED** обозначает лишь факт доступа к странице (как чтение, так и запись), а этот флаг необходимо именно для определения доступа на запись. **PAGE\_GLOBAL** - Глобальная страница. Элемент не выгружается из **TLB** при перезагрузке **CR3**. Полезный атрибут для страниц, содержащие системные структуры, общие для всех процессов системы. Например, в нашей ОС этот флаг можно применить для всех страниц из диапазона от 0x80000000 до 0xFFFFFFFF.  Пришло время так же рассказать про кеш элементов таблицы страниц - **TLB**. Без него каждое обращение к памяти при страничном преобразовании превращалось в 3 обращения (обращение к каталогу страниц, обращение к таблице страниц, обращение к нужной переменной). Для ускорения работы результаты преобразования кешируются во внутренней очень быстрой памяти процессора, чтобы в следующий раз не вычислять адрес заново. Изменение значения **CR3** приводит к очистке кеша, потому что такая команда подразумевает замену таблиц страниц на новые. Но, как правило, ядро системы во всех адресных пространствах находится по одному и тому же адресу, поэтому нет смысла удалять эти записи из кеша, для этого и существует флаг глобальности страницы. Например, можно смонтировать с ним таблицу прерываний:  void init\_interrupts() {  map\_pages(kernel\_page\_dir, idt, alloc\_phys\_pages(1), 1, **PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL**);  memset(idt, 0, 256 \* sizeof(IntDesc));  ...  set\_int\_handler(irq\_base, timer\_int\_handler, 0x8E);  asm("sti");  }  Из-за TLB изменение элемента таблицы страниц может не быть применено сразу, если до этого было обращение к нему и он до сих пор в кеше. Для принудительного удаления из кеша одного элемента существует ассемблерная инструкция **INVLPG**. Напишем специальную функцию для внутреннего использования менеджером памяти в memory\_manager.c:  static inline void flush\_page\_cache(void \*addr) {  asm("invlpg (,%0,)"::"a"(addr));  }  Обновлять кеш следует в функции temp\_map\_page и map\_pages, туда и добавим вызов этой функции:  void temp\_map\_page(phyaddr addr) {  \*((phyaddr\*)TEMP\_PAGE\_INFO) = (addr & ~PAGE\_OFFSET\_MASK) | PAGE\_VALID | PAGE\_WRITABLE;  **flush\_page\_cache((void\*)TEMP\_PAGE);**  }  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {  for (; count; count--) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & PAGE\_VALID)) {  phyaddr addr = alloc\_phys\_pages(1);  if (addr != -1) {  temp\_map\_page(paddr);  memset((void\*)TEMP\_PAGE, 0, PAGE\_SIZE);  temp\_map\_page(page\_table);  ((phyaddr\*)TEMP\_PAGE)[index] = addr | PAGE\_VALID | PAGE\_WRITABLE | PAGE\_USER;  page\_table = addr;  } else {  return false;  }  }  } else {  ((phyaddr\*)TEMP\_PAGE)[index] = (paddr & ~PAGE\_OFFSET\_BITS) | flags;  **flush\_page\_cache(vaddr);**  }  }  vaddr += PAGE\_SIZE;  paddr += PAGE\_SIZE;  }  return true;  }  Теперь можно приступить к теории менеджера виртуальной памяти.  Для него есть понятие AddressSpace - регион виртуальной памяти, в пределах которого выделяются виртуальные адреса. У ядра есть свой AddressSpace (от KERNEL\_MEMORY\_BASE до KERNEL\_MEMORY\_END), у каждого приложения свой (в личном каталоге страниц от USER\_MEMORY\_START до USER\_MEMORY\_END).  Для поиска свободного региона используется список занятых, организованный в виде массива, который способен расширяться по мере необходимости.  Для блока типа VMB\_MEMORY был выделен блок физической памяти с помощью alloc\_phys\_pages, а для VMB\_IO\_MEMORY был указан конкретный физический адрес и освобождать его с помощью free\_phys\_pages не нужно.  В следующем выпуске мы рассмотрим конкретную реализацию функций alloc\_virt\_pages и free\_virt\_pages. До встречи! |

Пишем свою операционную систему Защита памяти и работа над ошибками

|  |
| --- |
| В этом выпуске мы проставим правильные атрибуты для памяти ядра, а также исправим некоторые ошибки в коде.  Открытие адресной линии A20  Работа с виртуальной машиной Bochs немного отличается от работы с реальным железом. Например, тем, что адресная линия A20 после загрузки в Bochs по умолчанию открыта. Это приводит к тому, что разработчику ОС не требуется открывать её вручную, потому что всё и так работает, однако ошибка проявится на реальном железе. Адресная линия A20 позволяет процессору обращаться к памяти выше первого мегабайта и появилась на процессорах, которые поддерживают защищённый режим, а для обратной совместимости изначально закрыта. Открыть A20 не сложно: достаточно вставить следующий код перед переходом в защищённый режим в нашем файле boot.asm:  ; Загрузим значение в CR3  mov eax, 0x1000  mov cr3, eax  **; Откроем адресную линию A20**  **in al, 0x92**  **or al, 2**  **out 0x92, al**  ; Загрузим значение в GDTR  lgdt [gdtr32]  ; Запретим прерывания  cli  ; Перейдём в защищённый режим  mov eax, cr0  or eax, 0x80000001  mov cr0, eax  Теперь наша ОС сможет корректно обращаться к памяти свыше первого мегабайта и на реальном железе, а не только в эмуляторе Bochs.  Защита от излишней оптимизации  Сейчас все оптимизации кода GCC отключены, поэтому мы используем некие допущения, из-за которых наш код выполняет то что нужно. Но для нормальной работы, а не разработки ядро стоит компилировать с различными оптимизациями, ведь это уменьшит его размер и увеличит скорость работы. Однако в случае низкоуровневого программирования компилятор часто не знает определённых тонкостей среды, где работает код. Например, мы можем записать значение на TEMP\_PAGE, затем вызвать temp\_map\_page и переключить временную страницу на другую область физической памяти и прочитать значение. В случае разрешённых оптимизаций, компилятор просто сохранит записанное значение и сразу подставит его. Откуда ему знать, что переменные могут меняться без явного присваивания? Или компилятор может убрать инструкции, которые на его взгляд бессмысленны. Например, код, который присваивает переменной сначала одно значение, а потом сразу другое, может быть сокращён до последнего присваивания - результат тот же. Откуда компилятору знать, что мы работали с MMIO-регистром устройства? А ещё компилятор может переставлять команды местами, если они не влияют друг на друга (например, работа с независимыми переменными), но в случае работы с оборудованием совершенно обособленные переменные на самом деле могут оказаться тесно связанными.  Для того, чтобы избежать такого поведения оптимизатора, есть ключевое слово **volatile**. Его следует поставить перед объявлением переменной. Оно обозначает, что эта переменная может менять своё значение неявным образом и оптимизировать работу с ней нельзя. Злоупотреблять этим ключевым словом тоже не стоит, поэтому что оно сведёт на нет преимущества включения оптимизации.  В первую очередь добавим volatile в менеджер памяти - многие функции работают с TEMP\_PAGE, отследить изменения которой компилятор не в силах:  #include  "stdlib.h"  #include "memory\_manager.h"  typedef struct {  uint64 base;  uint64 length;  uint32 type;  uint32 acpi\_ext\_attrs;  } \_\_attribute\_\_((packed)) MemoryMapEntry;  typedef struct {  phyaddr next;  phyaddr prev;  size\_t size;  } PhysMemoryBlock;  size\_t free\_page\_count = 0;  phyaddr free\_phys\_memory\_pointer = -1;  void init\_memory\_manager(void \*memory\_map) {  asm("movl %%cr3, %0":"=a"(kernel\_page\_dir));  memory\_size = 0x100000;  MemoryMapEntry \*entry;  for (entry = memory\_map; entry->type; entry++) {  if ((entry->type == 1) && (entry->base >= 0x100000)) {  free\_phys\_pages(entry->base, entry->length >> PAGE\_OFFSET\_BITS);  memory\_size += entry->length;  }  }  }  /\* Physical memory manager \*/  size\_t get\_free\_memory\_size() {  return free\_page\_count << PAGE\_OFFSET\_BITS;  }  phyaddr alloc\_phys\_pages(size\_t count) {  if (free\_page\_count < count) return -1;  phyaddr result = -1;  if (free\_phys\_memory\_pointer != -1) {  phyaddr cur\_block = free\_phys\_memory\_pointer;  do {  temp\_map\_page(cur\_block);  if (((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size == count) {  phyaddr next = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next;  phyaddr prev = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev;  temp\_map\_page(next);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  temp\_map\_page(prev);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next = next;  if (cur\_block == free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = next;  if (cur\_block == free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = -1;  }  }  result = cur\_block;  break;  } else if (((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size > count) {  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size -= count;  result = cur\_block + (((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size << PAGE\_OFFSET\_BITS);  break;  }  cur\_block = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next;  } while (cur\_block != free\_phys\_memory\_pointer);  if (result != -1) {  free\_page\_count -= count;  }  }  return result;  }  void free\_phys\_pages(phyaddr base, size\_t count) {  if (free\_phys\_memory\_pointer == -1) {  temp\_map\_page(base);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size = count;  free\_phys\_memory\_pointer = base;  } else {  phyaddr cur\_block = free\_phys\_memory\_pointer;  do {  temp\_map\_page(cur\_block);  if (cur\_block + (((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size << PAGE\_OFFSET\_BITS) == base) {  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size += count;  if (((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next == base + (count << PAGE\_OFFSET\_BITS)) {  phyaddr next1 = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next;  temp\_map\_page(next1);  phyaddr next2 = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next;  size\_t new\_count = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size;  temp\_map\_page(next2);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev = cur\_block;  temp\_map\_page(cur\_block);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next = next2;  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size += new\_count;  }  break;  } else if (base + (count << PAGE\_OFFSET\_BITS) == cur\_block) {  size\_t old\_count = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size;  phyaddr next = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next;  phyaddr prev = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev;  temp\_map\_page(next);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  temp\_map\_page(prev);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  temp\_map\_page(base);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next = next;  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size = count + old\_count;  break;  } else if ((cur\_block > base) || (((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next == free\_phys\_memory\_pointer)) {  phyaddr prev = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next;  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  temp\_map\_page(prev);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  temp\_map\_page(base);  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next = cur\_block;  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->size = count;  break;  }  cur\_block = ((**volatile** PhysMemoryBlock\*)TEMP\_PAGE)->next;  } while (cur\_block != free\_phys\_memory\_pointer);  if (base < free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = base;  }    }  free\_page\_count += count;  }  /\* Lowlevel virtual memory manager \*/  static inline void flush\_page\_cache(void \*addr) {  asm("invlpg (,%0,)"::"a"(addr));  }  void temp\_map\_page(phyaddr addr) {  \*((phyaddr\*)TEMP\_PAGE\_INFO) = (addr & ~PAGE\_OFFSET\_MASK) | **PAGE\_PRESENT** | PAGE\_WRITABLE;  flush\_page\_cache((void\*)TEMP\_PAGE);  }  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {  for (; count; count--) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((**volatile** phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & **PAGE\_PRESENT**)) {  phyaddr addr = alloc\_phys\_pages(1);  if (addr != -1) {  temp\_map\_page(paddr);  memset((void\*)TEMP\_PAGE, 0, PAGE\_SIZE);  temp\_map\_page(page\_table);  ((**volatile** phyaddr\*)TEMP\_PAGE)[index] = addr | **PAGE\_PRESENT** | PAGE\_WRITABLE | PAGE\_USER;  page\_table = addr;  } else {  return false;  }  }  } else {  ((**volatile** phyaddr\*)TEMP\_PAGE)[index] = (paddr & ~PAGE\_OFFSET\_BITS) | flags;  flush\_page\_cache(vaddr);  }  }  vaddr += PAGE\_SIZE;  paddr += PAGE\_SIZE;  }  return true;  }  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  page\_table = ((**volatile** phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & **PAGE\_PRESENT**)) {  return 0;  }  } else {  return ((**volatile** phyaddr\*)TEMP\_PAGE)[index];  }  }  }  /\* Highlevel virtual memory manager \*/  void \*alloc\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {    }  void free\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, size\_t count, unsigned int flags) {    }  Теперь необходимо внести ещё одно незначительное исправление в функцию init\_interrupts:  map\_pages(kernel\_page\_dir, idt, alloc\_phys\_pages(1), 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);  memset(idt, 0, 256 \* sizeof(IntDesc));  **volatile** IDTR idtr = {256 \* sizeof(IntDesc), idt};  asm("lidt (,%0,)"::"a"(&idtr));  irq\_base = 0x20;  irq\_count = 16;  Без этого значение в IDTR будет загружаться некорректно. Ну и наконец осталось добавить несколько volatile в tty.c:  const char digits[] = "0123456789ABCDEF";  char num\_buffer[65];  #define KEY\_BUFFER\_SIZE 16  **volatile** char key\_buffer[KEY\_BUFFER\_SIZE];  **volatile** unsigned int key\_buffer\_head = 0;  **volatile** unsigned int key\_buffer\_tail = 0;  void keyboard\_int\_handler();  Доступ к этим переменным осуществляется как в обычном коде, так и в обработчике прерывания, поэтому компилятор не может отследить изменение переменных правильно. Простейший цикл while (key\_buffer\_head == key\_buffer\_tail) превратиться в бесконечный, если в самом начале эти переменные были равны, потому что значения будут прочитаны лишь один раз.  Ещё одна ошибка проявившаяся при включении оптимизации - работа со стеком в ассемблерной вставке в функции set\_int\_handler. Мы запихивали в стек регистр флагов, а потом его восстанавливали, но теперь для извлечения параметров из стека используется сам ESP вместо его копии в EBP, поэтому временная дестабилизация стека приводила к неправильному доступу к аргументу функции, хотя возврат из неё был корректен (до инструкции RET стек был приведён к правильному виду). Пока единственное что приходит на ум - сохранять регистр файлов не в стеке, а в локальной переменной (на самом деле она тоже оказывается в стеке, но об этом заботится уже компилятор). Так и сделаем:  void set\_int\_handler(uint8 index, void \*handler, uint8 type) {  **size\_t saved\_flags;**  **asm("pushf \n popl %0 \n cli":"=a"(saved\_flags));**  idt[index].selector = 8;  idt[index].address\_0\_15 = (size\_t)handler & 0xFFFF;  idt[index].address\_16\_31 = (size\_t)handler >> 16;  idt[index].type = type;  idt[index].reserved = 0;  **asm("pushl %0 \n popf"::"a"(saved\_flags));**  }  Ну вот и всё. Теперь наше ядро правильно работает с любым уровнем оптимизации. Чтобы включить оптимизацию следует в Makefile добавить к CFLAGS опцию -On, где n число от 0 до 3 (0 - отсутствие оптимизации, 3 - максимальная оптимизация). Обычно рекомендуется использовать 0-ой уровень для отладки и 2-ой для обычной работы. 3-ий уровень является режимом агрессивной оптимизации скорости, который может приводить к нестабильной работе приложения и не рекомендуется к применению без необходимости. Пример нового Makefile:  ifdef OS  LDFLAGS = -mi386pe  else  LDFLAGS = -melf\_i386  endif  CFLAGS = -m32 -ffreestanding **-O2**  all: script.ld startup.o stdlib.o main.o memory\_manager.o interrupts.o tty.o  ld $(LDFLAGS) -T script.ld -o kernel.bin startup.o stdlib.o main.o memory\_manager.o interrupts.o tty.o  objcopy kernel.bin -O binary  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  stdlib.o: stdlib.c stdlib.h  gcc -c $(CFLAGS) -o stdlib.o stdlib.c  main.o: main.c stdlib.h interrupts.h tty.h  gcc -c $(CFLAGS) -o main.o main.c  memory\_manager.o: memory\_manager.c memory\_manager.h stdlib.h  gcc -c $(CFLAGS) -o memory\_manager.o memory\_manager.c  interrupts.o: interrupts.c interrupts.h stdlib.h  gcc -c $(CFLAGS) -o interrupts.o interrupts.c  tty.o: tty.c tty.h stdlib.h  gcc -c $(CFLAGS) -o tty.o tty.c  clean:  rm -v \*.o kernel.bin  Защита памяти ядра  Теперь можно и правильно настроить атрибуты памяти ядра - запретить запись в область кода, поставить атрибут глобальности для всех страниц.  Для начала нужно как-то позволить реализовать определение базовых адресов секций кода и данных и их размеров. Всё это сделаем с помощью файла script.ld:  ENTRY(\_start)  **KERNEL\_BASE = 0xFFC00000;**  SECTIONS {  .text KERNEL\_BASE : {  **KERNEL\_CODE\_BASE = .;**  \*(.text)  \*(.code)  \*(.rodata\*)  }  .data ALIGN(0x1000) : {  **KERNEL\_DATA\_BASE = .;**  \*(.data)  }  .bss ALIGN(0x1000) : {  **KERNEL\_BSS\_BASE = .;**  \*(.bss)  }  .empty ALIGN(0x1000) - 1 : {  BYTE(0)  **KERNEL\_END = .;**  }  }  Теперь из любого модуля будет доступен ряд глобальных символов. Для удобства доступа опишем их в memory\_manager.h:  #ifndef MEMORY\_MANAGER\_H  #define MEMORY\_MANAGER\_H  #include "stdlib.h"  #define PAGE\_SIZE 0x1000  #define PAGE\_OFFSET\_BITS 12  #define PAGE\_OFFSET\_MASK 0xFFF  #define PAGE\_TABLE\_INDEX\_BITS 10  #define PAGE\_TABLE\_INDEX\_MASK 0x3FF  #define PHYADDR\_BITS 32  #define PAGE\_PRESENT (1 << 0)  #define PAGE\_WRITABLE (1 << 1)  #define PAGE\_USER (1 << 2)  #define PAGE\_WRITE\_THROUGH (1 << 3)  #define PAGE\_CACHE\_DISABLED (1 << 4)  #define PAGE\_ACCESSED (1 << 5)  #define PAGE\_MODIFIED (1 << 6)  #define PAGE\_GLOBAL (1 << 8)  **void KERNEL\_BASE();**  **void KERNEL\_CODE\_BASE();**  **void KERNEL\_DATA\_BASE();**  **void KERNEL\_BSS\_BASE();**  **void KERNEL\_END();**  #define KERNEL\_PAGE\_TABLE ((void\*)0xFFFFE000)  #define TEMP\_PAGE ((void\*)0xFFFFF000)  #define TEMP\_PAGE\_INFO ((size\_t)KERNEL\_PAGE\_TABLE + (((size\_t)TEMP\_PAGE >> PAGE\_OFFSET\_BITS) & PAGE\_TABLE\_INDEX\_MASK) \* sizeof(phyaddr))  #define USER\_MEMORY\_START ((void\*)0)  #define USER\_MEMORY\_END ((void\*)0x7FFFFFFF)  #define KERNEL\_MEMORY\_START ((void\*)0x80000000)  #define KERNEL\_MEMORY\_END ((void\*)(KERNEL\_BASE - 1))  typedef size\_t phyaddr;  typedef enum {  VMB\_RESERVED,  VMB\_MEMORY,  VMB\_IO\_MEMORY  } VirtMemoryBlockType;  typedef struct {  VirtMemoryBlockType type;  void \*base;  size\_t length;  } VirtMemoryBlock;  typedef struct {  phyaddr page\_dir;  void \*start;  void \*end;  size\_t block\_count;  VirtMemoryBlock \*blocks;  } AddressSpace;  phyaddr kernel\_page\_dir;  size\_t memory\_size;  AddressSpace kernel\_address\_space;  void init\_memory\_manager(void \*memory\_map);  size\_t get\_free\_memory\_size();  phyaddr alloc\_phys\_pages(size\_t count);  void free\_phys\_pages(phyaddr base, size\_t count);  void temp\_map\_page(phyaddr addr);  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr);  void \*alloc\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  void free\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, size\_t count, unsigned int flags);  #endif  Осталось только изменить атрибуты страниц с помощью map\_pages:  void init\_memory\_manager(void \*memory\_map) {  asm("movl %%cr3, %0":"=a"(kernel\_page\_dir));  memory\_size = 0x100000;  MemoryMapEntry \*entry;  for (entry = memory\_map; entry->type; entry++) {  if ((entry->type == 1) && (entry->base >= 0x100000)) {  free\_phys\_pages(entry->base, entry->length >> PAGE\_OFFSET\_BITS);  memory\_size += entry->length;  }  }  **map\_pages(kernel\_page\_dir, KERNEL\_CODE\_BASE, get\_page\_info(kernel\_page\_dir, KERNEL\_CODE\_BASE),**  **((size\_t)KERNEL\_DATA\_BASE - (size\_t)KERNEL\_CODE\_BASE) >> PAGE\_OFFSET\_BITS, PAGE\_PRESENT | PAGE\_GLOBAL);**  **map\_pages(kernel\_page\_dir, KERNEL\_DATA\_BASE, get\_page\_info(kernel\_page\_dir, KERNEL\_DATA\_BASE),**  **((size\_t)KERNEL\_END - (size\_t)KERNEL\_DATA\_BASE) >> PAGE\_OFFSET\_BITS, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);**  **map\_pages(kernel\_page\_dir, KERNEL\_PAGE\_TABLE, get\_page\_info(kernel\_page\_dir, KERNEL\_PAGE\_TABLE), 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);**  }  Теперь страниц настроены более привычно, если бы мы писали обычное приложение - доступ к страницам с кодом на запись запрещён. Также страницы ядра теперь имеют флаг PAGE\_GLOBAL, который предотвращает выгрузку этих элементов из кеша при переключении каталога страниц поскольку ядро общее для всех процессов.  Заключение  В этом выпуске мы исправили ряд предыдущих недостатков ядра и загрузчика системы, которые бы могли помешать дальнейшей разработке. Также мы сделали контроль памяти более строгим, чтобы сразу замечать ошибки вроде случайной перезаписи кода ядра. |

## Пишем свою операционную систему Менеджер виртуальной памяти

|  |
| --- |
| В этом выпуске мы наконец-то реализуем менеджер виртуальной памяти, который знает не только "что спроецировать", но и "куда спроецировать". Его реализацию на текущий момент можно считать черновой (в отличии от менеджера физической памяти) и безупречную работу я гарантировать не могу.  Для начала исправим небольшую ошибку в map\_pages:  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {  for (; count; count--) {  phyaddr page\_table = page\_dir;  char shift;  for (shift = PHYADDR\_BITS - PAGE\_TABLE\_INDEX\_BITS; shift >= PAGE\_OFFSET\_BITS; shift -= PAGE\_TABLE\_INDEX\_BITS) {  unsigned int index = ((size\_t)vaddr >> shift) & PAGE\_TABLE\_INDEX\_MASK;  temp\_map\_page(page\_table);  if (shift > PAGE\_OFFSET\_BITS) {  **size\_t prev\_page\_table = page\_table;**  page\_table = ((volatile phyaddr\*)TEMP\_PAGE)[index];  if (!(page\_table & PAGE\_PRESENT)) {  phyaddr addr = alloc\_phys\_pages(1);  if (addr != -1) {  temp\_map\_page(**addr**);  memset((void\*)TEMP\_PAGE, 0, PAGE\_SIZE);  temp\_map\_page(**prev\_page\_table**);  ((volatile phyaddr\*)TEMP\_PAGE)[index] = addr | PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_USER;  page\_table = addr;  } else {  return false;  }  }  } else {  ((volatile phyaddr\*)TEMP\_PAGE)[index] = (paddr & ~PAGE\_OFFSET\_BITS) | flags;  flush\_page\_cache(vaddr);  }  }  vaddr += PAGE\_SIZE;  paddr += PAGE\_SIZE;  }  return true;  }  Теперь, финальная версия memory\_manager.h:  #ifndef MEMORY\_MANAGER\_H  #define MEMORY\_MANAGER\_H  #include "stdlib.h"  #define PAGE\_SIZE 0x1000  #define PAGE\_OFFSET\_BITS 12  #define PAGE\_OFFSET\_MASK 0xFFF  #define PAGE\_TABLE\_INDEX\_BITS 10  #define PAGE\_TABLE\_INDEX\_MASK 0x3FF  #define PHYADDR\_BITS 32  #define PAGE\_PRESENT (1 << 0)  #define PAGE\_WRITABLE (1 << 1)  #define PAGE\_USER (1 << 2)  #define PAGE\_WRITE\_THROUGH (1 << 3)  #define PAGE\_CACHE\_DISABLED (1 << 4)  #define PAGE\_ACCESSED (1 << 5)  #define PAGE\_MODIFIED (1 << 6)  #define PAGE\_GLOBAL (1 << 8)  void KERNEL\_BASE();  void KERNEL\_CODE\_BASE();  void KERNEL\_DATA\_BASE();  void KERNEL\_BSS\_BASE();  void KERNEL\_END();  #define KERNEL\_PAGE\_TABLE ((void\*)0xFFFFE000)  #define TEMP\_PAGE ((void\*)0xFFFFF000)  #define TEMP\_PAGE\_INFO ((size\_t)KERNEL\_PAGE\_TABLE + (((size\_t)TEMP\_PAGE >> PAGE\_OFFSET\_BITS) & PAGE\_TABLE\_INDEX\_MASK) \* sizeof(phyaddr))  #define USER\_MEMORY\_START ((void\*)0)  #define USER\_MEMORY\_END ((void\*)0x7FFFFFFF)  #define KERNEL\_MEMORY\_START ((void\*)0x80000000)  #define KERNEL\_MEMORY\_END ((void\*)(KERNEL\_BASE - 1))  typedef size\_t phyaddr;  typedef enum {  VMB\_RESERVED,  VMB\_MEMORY,  VMB\_IO\_MEMORY  } VirtMemoryBlockType;  typedef struct {  VirtMemoryBlockType type;  void \*base;  size\_t length;  } VirtMemoryBlock;  typedef struct {  phyaddr page\_dir;  void \*start;  void \*end;  size\_t block\_table\_size;  size\_t block\_count;  VirtMemoryBlock \*blocks;  } AddressSpace;  phyaddr kernel\_page\_dir;  size\_t memory\_size;  AddressSpace kernel\_address\_space;  void init\_memory\_manager(void \*memory\_map);  size\_t get\_free\_memory\_size();  phyaddr alloc\_phys\_pages(size\_t count);  void free\_phys\_pages(phyaddr base, size\_t count);  void temp\_map\_page(phyaddr addr);  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr);  void \*alloc\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  bool free\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, unsigned int flags);  #endif  Теперь следует усовершенствовать код memcpy, чтобы он полностью корректно обрабатывал перекрывающиеся блоки памяти (ошибки в коде нет, просто реализация была упрощённой):  void memcpy(void \*dest, void \*src, size\_t count) {  **if (dest < src) {**  asm("movl %0, %%edi \n movl %1, %%esi \n movl %2, %%ecx \n rep movsl"::"a"(dest),"b"(src),"c"(count >> 2));  asm("movl %0, %%ecx \n rep movsb"::"a"(count & 3));  **} else {**  **asm("std");**  **asm("movl %0, %%edi \n movl %1, %%esi \n movl %2, %%ecx \n rep movsb"::"a"(dest + count),"b"(src + count),"c"(count & 3));**  **asm("movl %0, %%ecx \n rep movsl"::"c"(count >> 2));**  **asm("cld");**  **}**  }  А теперь собственно реализация функций alloc\_virt\_pages и free\_virt\_pages:  /\* Highlevel virtual memory manager \*/  static inline bool is\_blocks\_overlapped(void \*base1, size\_t size1, void \*base2, size\_t size2) {  return ((base1 >= base2) && (base1 < base2 + size2)) || ((base2 >= base1) && (base2 < base1 + size1));  }  void \*alloc\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {  VirtMemoryBlockType type = VMB\_IO\_MEMORY;  size\_t i;  if (vaddr == NULL) {  vaddr = address\_space->end - (count << PAGE\_OFFSET\_BITS) + 1;  for (i = 0; i < address\_space->block\_count; i++) {  if (is\_blocks\_overlapped(address\_space->blocks[i].base, address\_space->blocks[i].length, vaddr, count)) {  vaddr = address\_space->blocks[i].base - (count << PAGE\_OFFSET\_BITS);  } else {  break;  }  }  } else {  if ((vaddr >= address\_space->start) && (vaddr + (count << PAGE\_OFFSET\_BITS) < address\_space->end)) {  for (i = 0; i < address\_space->block\_count; i++) {  if (is\_blocks\_overlapped(address\_space->blocks[i].base, address\_space->blocks[i].length, vaddr, count << PAGE\_OFFSET\_BITS)) {  vaddr = NULL;  break;  } else if (address\_space->blocks[i].base < vaddr) {  break;  }  }  } else {  vaddr = NULL;  }  }  if (vaddr != NULL) {  if (paddr == -1) {  paddr = alloc\_phys\_pages(count);  type = VMB\_MEMORY;  }  if (paddr != -1) {  if (map\_pages(address\_space->page\_dir, vaddr, paddr, count, flags)) {  if (address\_space->block\_count == address\_space->block\_table\_size) {  size\_t new\_size = address\_space->block\_table\_size \* sizeof(VirtMemoryBlock);  new\_size = (new\_size + PAGE\_OFFSET\_MASK) & ~PAGE\_OFFSET\_MASK;  new\_size += PAGE\_SIZE;  new\_size = new\_size >> PAGE\_OFFSET\_BITS;  if (&kernel\_address\_space != address\_space) {  VirtMemoryBlock \*new\_table = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, new\_size,  PAGE\_PRESENT | PAGE\_WRITABLE);  if (new\_table) {  memcpy(new\_table, address\_space->blocks, address\_space->block\_table\_size \* sizeof(VirtMemoryBlock));  free\_virt\_pages(&kernel\_address\_space, address\_space->blocks, 0);  address\_space->blocks = new\_table;  } else {  goto fail;  }  } else {  phyaddr new\_page = alloc\_phys\_pages(1);  if (new\_page == -1) {  goto fail;  } else {  VirtMemoryBlock \*main\_block = &(address\_space->blocks[address\_space->block\_count - 1]);  if (map\_pages(address\_space->page\_dir, main\_block->base + main\_block->length, new\_page, 1,  PAGE\_PRESENT | PAGE\_WRITABLE)) {  main\_block->length += PAGE\_SIZE;  } else {  free\_phys\_pages(new\_page, 1);  }  }  }  address\_space->block\_table\_size = (new\_size << PAGE\_OFFSET\_BITS) / sizeof(VirtMemoryBlock);  }  memcpy(address\_space->blocks + i + 1, address\_space->blocks + i,  (address\_space->block\_count - i) \* sizeof(VirtMemoryBlock));  address\_space->block\_count++;  address\_space->blocks[i].type = type;  address\_space->blocks[i].base = vaddr;  address\_space->blocks[i].length = count << PAGE\_OFFSET\_BITS;  } else {  fail:  map\_pages(address\_space->page\_dir, vaddr, 0, count, 0);  free\_phys\_pages(paddr, count);  vaddr = NULL;  }  }  }  return vaddr;  }  bool free\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, unsigned int flags) {  size\_t i;  for (i = 0; i < address\_space->block\_count; i++) {  if ((address\_space->blocks[i].base <= vaddr) && (address\_space->blocks[i].base + address\_space->blocks[i].length > vaddr)) {  break;  }  }  if (i < address\_space->block\_count) {  if (address\_space->blocks[i].type = VMB\_MEMORY) {  free\_phys\_pages(get\_page\_info(address\_space->page\_dir, vaddr) & ~PAGE\_OFFSET\_MASK, address\_space->blocks[i].length >> PAGE\_OFFSET\_BITS);  }  address\_space->block\_count--;  memcpy(address\_space->blocks + i, address\_space->blocks + i + 1, (address\_space->block\_count - i) \* sizeof(VirtMemoryBlock));  return true;  } else {  return false;  }  }  Все таблицы блоков хранятся в адресном пространстве ядра. При необходимости они могут расширяться (по сути это динамический массив). Выделение адресов начинается с конца адресного пространства. Почти полностью реализована обработка ошибок выделения памяти (для личных адресных пространств процессов полностью, для адресного пространства ядра - нет).  Ну и наконец добавим в конец функции init\_memory\_manager инициализацию структуры адресного пространства ядра:  void init\_memory\_manager(void  \*memory\_map) {  asm("movl %%cr3, %0":"=a"(kernel\_page\_dir));  memory\_size = 0x100000;  MemoryMapEntry \*entry;  for (entry = memory\_map; entry->type; entry++) {  if ((entry->type == 1) && (entry->base >= 0x100000)) {  free\_phys\_pages(entry->base, entry->length >> PAGE\_OFFSET\_BITS);  memory\_size += entry->length;  }  }  map\_pages(kernel\_page\_dir, KERNEL\_CODE\_BASE, get\_page\_info(kernel\_page\_dir, KERNEL\_CODE\_BASE),  ((size\_t)KERNEL\_DATA\_BASE - (size\_t)KERNEL\_CODE\_BASE) >> PAGE\_OFFSET\_BITS, PAGE\_PRESENT | PAGE\_GLOBAL);  map\_pages(kernel\_page\_dir, KERNEL\_DATA\_BASE, get\_page\_info(kernel\_page\_dir, KERNEL\_DATA\_BASE),  ((size\_t)KERNEL\_END - (size\_t)KERNEL\_DATA\_BASE) >> PAGE\_OFFSET\_BITS, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);  map\_pages(kernel\_page\_dir, KERNEL\_PAGE\_TABLE, get\_page\_info(kernel\_page\_dir, KERNEL\_PAGE\_TABLE), 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);  **kernel\_address\_space.page\_dir = kernel\_page\_dir;**  **kernel\_address\_space.start = KERNEL\_MEMORY\_START;**  **kernel\_address\_space.end = KERNEL\_MEMORY\_END;**  **kernel\_address\_space.block\_table\_size = PAGE\_SIZE / sizeof(VirtMemoryBlock);**  **kernel\_address\_space.blocks = KERNEL\_MEMORY\_START;**  **kernel\_address\_space.block\_count = 1;**  **map\_pages(kernel\_page\_dir, kernel\_address\_space.blocks, alloc\_phys\_pages(1), 1, PAGE\_PRESENT | PAGE\_WRITABLE);**  **kernel\_address\_space.blocks[0].type = VMB\_RESERVED;**  **kernel\_address\_space.blocks[0].base = kernel\_address\_space.blocks;**  **kernel\_address\_space.blocks[0].length = PAGE\_SIZE;**  }  Ну вот и готово. В качестве демонстрации будем выделять память для видео-буфера и таблицы прерываний динамически.  interrupts.c:  ...  **IntDesc \*idt;**  void timer\_int\_handler();  void init\_interrupts() {  **idt = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);**  memset(idt, 0, 256 \* sizeof(IntDesc));  volatile IDTR idtr = {256 \* sizeof(IntDesc), idt};  asm("lidt (,%0,)"::"a"(&idtr));  irq\_base = 0x20;  irq\_count = 16;  ...  tty.c:  #include <stdarg.h>  #include "stdlib.h"  **#include "memory\_manager.h"**  #include "interrupts.h"  #include "tty.h"  #include "scancodes.h"  ...  void init\_tty() {  **tty\_buffer = alloc\_virt\_pages(&kernel\_address\_space, NULL, 0xB8000, 1, PAGE\_PRESENT | PAGE\_WRITABLE);**  tty\_width = \*((uint16\*)0x44A);  tty\_height = 25;  tty\_io\_port = \*((uint16\*)0x463);  cursor = (\*((uint8\*)0x451)) \* tty\_width + (\*((uint8\*)0x450));  text\_attr = 7;  set\_int\_handler(irq\_base + 1, keyboard\_int\_handler, 0x8E);  }  А в kernel\_main добавим необязательную демонстрацию внутренностей управления памятью:  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  #include "tty.h"  typedef struct {  uint64 base;  uint64 size;  } BootModuleInfo;  uint32 \*idt;  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  init\_memory\_manager(memory\_map);  init\_interrupts();  init\_tty();  set\_text\_attr(15);  printf("Welcome to MyOS!\n");  **int i;**  **for (i = 0; i < kernel\_address\_space.block\_count; i++) {**  **printf("type = %d, base = 0x%x, length = 0x%x\n", kernel\_address\_space.blocks[i].type, kernel\_address\_space.blocks[i].base,**  **kernel\_address\_space.blocks[i].length);**  **}**  char string[10];  in\_string(string, sizeof(string));  out\_string(string);  }  Вот и всё! Теперь у нас есть относительно полноценный менеджер памяти (не хватает только реализации подкачки памяти и выделения блоков произвольного размера, а не только кратного размеру страницы, однако первое нам пока делать рано, а второе можно переложить на плечи прикладных программ, если писать микроядро).  Замечание по использованию менеджера памяти:  1) Как я уже сказал обработка нехватки виртуальных адресов для адресного пространства ядра реализована не полностью. Список блоков растёт с начала адресного пространства, а адреса выделяются с конца. То, что они встретятся крайне маловероятно, но всё же возможно. В данном случае ошибки выделения памяти не случится - список блоков "наедет" на самый нижний блок памяти и это приведёт к непредсказуемым последствиям. А вот защита от обратной ситуации реализована - менеджер памяти никогда не выделит блок памяти, который бы пересекался со списком блока, проблема лишь в том, когда последний начинает расти. В общем, будте осторожны, хотя и вряд ли наткнётесь на эту проблему.  2) В init\_memory\_manager, init\_interrupts и init\_tty я не произвожу проверку успешности выделения оперативной памяти, потому что вряд ли нашу ОС запустят на компьютере с объёмом памяти менее двух мегабайт. Да и если такие извращенцы найдутся, то не вижу ничего плохого в том, чтобы система упала - всё равно у нас даже драйвер экрана без верхней (выше 1 МБ) памяти не сможет работать. Однако вы, особенно если будете манипулировать большими блоками памяти, должны проверять, что alloc\_virt\_pages вернул не NULL (признак того, что по какой-то причине выделить память не удалось).  На этом заканчиваю этот выпуск. Все вопросы вы можете задать на мой электронный адрес - [kiv.apple@gmail.com](mailto:apple@gmail.com). |

Пишем свою операционную систему. Многозадачность

|  |
| --- |
| В этом выпуске мы рассмотрим теорию многозадачности и подготовим наш код к её поддержке. До конца в этом выпуске мы многозадачность не сделаем, потому что это достаточно сложная тема.  Теория  Многозадачность - это способность ОС выполнять несколько программ параллельно. В идеальной ситуации каждое приложение выполняется на отдельном ядре процессора, независимо и полностью параллельно. Однако, это идеальная ситуация и в реальности как правило недостижимая, поэтому сразу много процессов вынуждены делить один процессор. Иллюзия параллельной работы достигается засчёт быстрого переключения контекстов задач. То есть, допустим, в течении 1 миллисекунды процессор выполняет код одной программы, затем её работа прерывается (например, по прерыванию таймера), состояние (регистры процессора, стек) сохраняется в системную область данных, затем оттуда извлекается состояние другой задачи и работа возобновляется, но выполняется уже другой процесс. Когда и его **квант времени** истечёт, ядро точно так же заберёт у него управление, сохранит его состояние и восстановит состояние первой задачи (если больше процессов нет, иначе управление перейдёт к третьему процессу).  Есть два вида реализации многозадачности: кооперативная и вытесняющая.  При **кооперативной многозадачности** требуется поддержка приложения - оно само решает, когда прерывать своё выполнение и запустить следующую задачу. То есть, приложение выполняет некоторую работу, которую считает достаточной для одного кванта времени, а затем обращается к ОС: "я закончила, можешь передавать управление следующей задаче". Очевидно, что данный способ может быть применим лишь для небольших специфических систем (где все компоненты написаны и отлажены одной группой разработчиков и приложения считаются доверенными), но никак не для массовых очень сложных программных комплексов. Во-первых, при разработки программ под такую ОС потребуется учитывать все особенности её многозадачности, чтобы она по-прежнему работала эффективно, это требует учёта множества факторов и определённых знаний от программиста. Во-вторых, сбой в любом приложении повлечёт за собой сильное падение производительности или вовсе полное зависание системы. Поэтому такой алгоритм сейчас практически нигде в массовом порядке не применяется и мы тоже не будем его реализовывать.  **Вытесняющая многозадачность** используется во всех современных многозадачных десктопных и серверных операционных системах. Более того, нет никаких причин, чтобы не использовать её везде, где вообще нужна полноценная многозадачность и, как правило, так и делают. Её суть в том, что приложение не влияет на переключение задач, оно просто непрерывно выполняется, а ОС сама решает, когда отобрать у него управление (обычно это делается по прерыванию таймера). С одной стороны разработка приложений под такую систему становится очень простой (надо просто писать обычное приложение, не думая о том в какой момент будет переключение), с другой ОС получается очень надёжной (её весьма проблематично подвесить или затормозить), потому что её стабильность зависит лишь от ядра, а не от ошибок в приложениях. Мы будем реализовывать именно этот подход.  При написании функций многозадачности следует уделить особое внимание такому компоненту, как **планировщик задач**. Современные ОС переключают задачи не просто по кругу, а с учётом множества параметров - загруженности процессора, приоритетов, состояния приложения и т. д. В результате этого какие-то задачи получают больше квантов времени, а какие-то меньше. Такая "несправедливость" вполне оправдана: допустим, запущен процесс текстового редактора и одновременно с ним процесс кодирования видео. Первый большую часть времени проводит в ожидании нажатий клавиш, второй занят непрерывными вычислениями. Если делить кванты времени поровну, то половина времени процессора будет расходоваться впустую - на цикл проверки наличия новых событий в текстовом редакторе. Грамотная ОС же не будет давать кванты времени текстовому редактору, пока пользователь не начал с ним работать (ну или не произошло какое-то другое событие, на которое программа должна отреагировать). В таком случае практически все кванты времени будут использованы с пользой - большая часть на кодирование видео и немного на обработку событий редактором. Это лишь один из критериев планировки выполнения процессов - состояние процесса (процессы в состоянии "ожидание события" пропускаются при переключении). Вторым немаловажным параметром является приоритет процесса. Разные задачи требуют различной скорости реакции. Например, человек не в состоянии заметить задержки изображения вплоть до 20 миллисекунд, а задержка звука даже в 1-2 миллисекунды уже будет неприятна. Следовательно, процесс вывода звука должен иметь гораздо больший приоритет, чем процесс воспроизведения видео. При этом обработка звука значительно более простая задача, чем обработка видео, поэтому несмотря на высокий приоритет первый процесс не понизит отзывчивость системы, потому что большую часть времени будет спать. При грамотной разработке системы наиболее приоритетные задачи имеют крайне низкую вычислительную сложность, а действительно ресурсоёмкие задачи имеют пониженный приоритет. В таком случае система будет работать быстро и с малым временем отклика.  Мы пока не будем писать полноценный планировщик, потому что это достаточно сложная задача и даже в современных ОС порой находят, что улучшить и где его оптимизировать, то есть эта задача не решена до конца. Пока ограничимся примитивным цикличным переключением задач без учёта приоритетов. Когда мы реализуем механизм межпроцессного взаимодействия, добавим пропуск спящих задач (пока у нас нет критериев, по которым задачу можно считать ожидающей события). Когда-нибудь, возможно, я и дойду до поддержки приоритетов, но пока есть задачи поважнее.  Семафоры и синхронизация  Пока приложение взаимодействует лишь само с собой, его можно писать как обычную программу для однозадачной системы. Однако, когда приложения начинают обращаться к общим данным (хотя бы когда обмениваются сообщениями)  необходим некий механизм синхронизации. Поясню возникшую проблему на примере:  int variable = 10;  void thread1() {  variable = 0;  }  void thread2() {  if (variable != 0) {  printf("100 / variable = %d\n", 100 / variable);  } else {  printf("Cannot divive by zero!\n");  }  }  Функции thread1 и thread2 выполняются параллельно. Переменная varaible доступна обоим **нитям** (параллельные задачи в рамках одного адресного пространства). Допустим, первой получает управление нить thread2. Она производит сравнение variable с нулём (дальше будет деление на это значение, а на ноль делить нельзя). Сравнение проходит успешно (10 != 0) и функция заходит в блок if. Но тут, прямо в этот момент, ОС отбирает у thread2 управление (откуда ей знать, что нить была занята чем-то важным? а если система многоядерная и задачи выполняются вообще физически параллельно?) и отдаёт его thread1, которая обнуляет переменную. thread1 закончила свою работу и управление вернулось к thread2. Она думает, что variable не равна нулю (только что же проверили) и выполняет деление. В итоге происходит ошибка и программа завершается некорректно, хотя защита от деления на ноль в функции была! Переключение задач невозможно, либо очень предсказать, поэтому получается, что нельзя надеяться ни на какие проверки. Как же тогда работать с общими данными?  Для удобной работы были придуманы **семафоры**. Их суть в том, что приложение перед работой с общим ресурсом ставит флаг "я работаю с variable, никому не трогать!", выполняет свои действия, а потом убирает флаг ("я закончил"). В таком случае программист может быть уверен, что никто кроме его программы в данный момент времени не работает с общим ресурсом. Другие задачи, которым нужен доступ к нему будут ждать, а те, которым он не нужен, будут работать как ни в чём не бывало (то есть многозадачность не теряется).  Наш код в потокобезопасном варианте выглядит так:  int variable = 10;  Mutex mutex;  void thread1() {  get\_mutex(&mutex);  variable = 0;  release\_mutex(&mutex);  }  void thread2() {  get\_mutex(&mutex);  if (variable != 0) {  printf("100 / variable = %d\n", 100 / variable);  } else {  printf("Cannot divive by zero!\n");  }  release\_mutex(&mutex);  }  get\_mutex присваивает переменной mutex значение true, если оно было false. Если оно было true, то ждёт, пока станет false (как сделать это так быстро, чтобы никто другой не успел вмешаться, обсудим чуть позже). release\_mutex это простое обнуление переменной. Теперь в каком бы порядке планировщик не переключал задачи thread1 и thread2 каждая из них отработает именно так, как задумывал автор и ошибок не произойдёт.  Злоупотреблять семафорами тоже не стоит, потому что их неоправданное применение приведёт к падению производительности и отзывчивости системы.  Дополнение стандартной библиотеки  После теории можно приступить к началу практики. В этом выпуске мы добавим синхронизацию везде, где она нужна. Но сначала надо написать функции работы с семафорами. Их мы разместим в stdlib.c.  Модификации заголовочного файла stdlib.h:  #ifndef STDLIB\_H  #define STDLIB\_H  typedef enum {  false = 0,  true = 1  } bool;  #define NULL ((void\*)0)  typedef unsigned char uint8;  typedef signed char int8;  typedef unsigned short uint16;  typedef signed short int16;  typedef unsigned int uint32;  typedef signed int int32;  typedef unsigned long long uint64;  typedef signed long long int64;  #ifdef \_\_x86\_64\_\_  typedef uint64 size\_t;  #else  typedef uint32 size\_t;  #endif  **typedef bool Mutex;**  #define min(a, b) (((a) > (b)) ? (b) : (a))  #define max(a, b) (((a) > (b)) ? (a) : (b))  #define outportb(port, value) asm("outb %b0, %w1"::"a"(value),"d"(port));  #define outportw(port, value) asm("outw %w0, %w1"::"a"(value),"d"(port));  #define outportl(port, value) asm("outl %0, %w1"::"a"(value),"d"(port));  #define inportb(port, out\_value) asm("inb %w1, %b0":"=a"(out\_value):"d"(port));  #define inportw(port, out\_value) asm("inw %w1, %w0":"=a"(out\_value):"d"(port));  #define inportl(port, out\_value) asm("inl %w1, %0":"=a"(out\_value):"d"(port));  void memset(void \*mem, char value, size\_t count);  void memset\_word(void \*mem, uint16 value, size\_t count);  void memcpy(void \*dest, void \*src, size\_t count);  int memcmp(void \*mem1, void \*mem2, size\_t count);  void \*memchr(void \*mem, char value, size\_t count);  size\_t strlen(char \*str);  void strcpy(char \*dest, char \*src);  void strncpy(char \*dest, char\*src, size\_t max\_count);  int strcmp(char \*str1, char \*str2);  char \*strchr(char \*str, char value);  **bool mutex\_get(Mutex \*mutex, bool wait);**  **void mutex\_release(Mutex \*mutex);**  #endif  Тип Mutex объявлен не только для читабельности, но и для возможности в будущем расширения структуры семафора (например, добавить дескриптор нити, которая его захватила и т. д.), если это потребуется.  Использовать только средства языка Си не получится, иначе мы столкнёмся с той же проблемой, что и в примере выше - нельзя быть уверенным, что после проверки семафора на захваченность, это состояние не изменится. Но нам на помощь вновь приходят ассемблерные вставки.  bool mutex\_get(Mutex  \*mutex, bool wait) {  bool old\_value = true;  do {  asm("xchg (,%1,), %0":"=a"(old\_value):"b"(mutex),"a"(old\_value));  } while (old\_value && wait);  return !old\_value;  }  void mutex\_release(Mutex \*mutex) {  \*mutex = false;  }  Инструкция xchg обменивает местами значения двух регистров, либо переменной и регистров. При этом можно быть уверенными в том, что операция **атомарна** - то есть ничто не прервёт её и не изменит результат.  Потокобезопасный менеджер памяти  Ну а теперь можно наконец внести поддержку семафоров в менеджер памяти.  memory\_manager.h:  #ifndef MEMORY\_MANAGER\_H  #define MEMORY\_MANAGER\_H  #include "stdlib.h"  #define PAGE\_SIZE 0x1000  #define PAGE\_OFFSET\_BITS 12  #define PAGE\_OFFSET\_MASK 0xFFF  #define PAGE\_TABLE\_INDEX\_BITS 10  #define PAGE\_TABLE\_INDEX\_MASK 0x3FF  #define PHYADDR\_BITS 32  #define PAGE\_PRESENT (1 << 0)  #define PAGE\_WRITABLE (1 << 1)  #define PAGE\_USER (1 << 2)  #define PAGE\_WRITE\_THROUGH (1 << 3)  #define PAGE\_CACHE\_DISABLED (1 << 4)  #define PAGE\_ACCESSED (1 << 5)  #define PAGE\_MODIFIED (1 << 6)  #define PAGE\_GLOBAL (1 << 8)  void KERNEL\_BASE();  void KERNEL\_CODE\_BASE();  void KERNEL\_DATA\_BASE();  void KERNEL\_BSS\_BASE();  void KERNEL\_END();  #define KERNEL\_PAGE\_TABLE ((void\*)0xFFFFE000)  #define TEMP\_PAGE ((void\*)0xFFFFF000)  #define TEMP\_PAGE\_INFO ((size\_t)KERNEL\_PAGE\_TABLE + (((size\_t)TEMP\_PAGE >> PAGE\_OFFSET\_BITS) & PAGE\_TABLE\_INDEX\_MASK) \* sizeof(phyaddr))  #define USER\_MEMORY\_START ((void\*)0)  #define USER\_MEMORY\_END ((void\*)0x7FFFFFFF)  #define KERNEL\_MEMORY\_START ((void\*)0x80000000)  #define KERNEL\_MEMORY\_END ((void\*)(KERNEL\_BASE - 1))  typedef size\_t phyaddr;  typedef enum {  VMB\_RESERVED,  VMB\_MEMORY,  VMB\_IO\_MEMORY  } VirtMemoryBlockType;  typedef struct {  VirtMemoryBlockType type;  void \*base;  size\_t length;  } VirtMemoryBlock;  typedef struct {  phyaddr page\_dir;  void \*start;  void \*end;  size\_t block\_table\_size;  size\_t block\_count;  VirtMemoryBlock \*blocks;  **Mutex mutex;**  } AddressSpace;  phyaddr kernel\_page\_dir;  size\_t memory\_size;  AddressSpace kernel\_address\_space;  void init\_memory\_manager(void \*memory\_map);  size\_t get\_free\_memory\_size();  phyaddr alloc\_phys\_pages(size\_t count);  void free\_phys\_pages(phyaddr base, size\_t count);  void temp\_map\_page(phyaddr addr);  bool map\_pages(phyaddr page\_dir, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  phyaddr get\_page\_info(phyaddr page\_dir, void \*vaddr);  void \*alloc\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags);  bool free\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, unsigned int flags);  #endif  memory\_manager.c:  #include "stdlib.h"  #include "memory\_manager.h"  typedef struct {  uint64 base;  uint64 length;  uint32 type;  uint32 acpi\_ext\_attrs;  } \_\_attribute\_\_((packed)) MemoryMapEntry;  typedef struct {  phyaddr next;  phyaddr prev;  size\_t size;  } PhysMemoryBlock;  size\_t free\_page\_count = 0;  phyaddr free\_phys\_memory\_pointer = -1;  **Mutex phys\_memory\_mutex;**  void init\_memory\_manager(void \*memory\_map) {  asm("movl %%cr3, %0":"=a"(kernel\_page\_dir));  memory\_size = 0x100000;  MemoryMapEntry \*entry;  for (entry = memory\_map; entry->type; entry++) {  if ((entry->type == 1) && (entry->base >= 0x100000)) {  free\_phys\_pages(entry->base, entry->length >> PAGE\_OFFSET\_BITS);  memory\_size += entry->length;  }  }  map\_pages(kernel\_page\_dir, KERNEL\_CODE\_BASE, get\_page\_info(kernel\_page\_dir, KERNEL\_CODE\_BASE),  ((size\_t)KERNEL\_DATA\_BASE - (size\_t)KERNEL\_CODE\_BASE) >> PAGE\_OFFSET\_BITS, PAGE\_PRESENT | PAGE\_GLOBAL);  map\_pages(kernel\_page\_dir, KERNEL\_DATA\_BASE, get\_page\_info(kernel\_page\_dir, KERNEL\_DATA\_BASE),  ((size\_t)KERNEL\_END - (size\_t)KERNEL\_DATA\_BASE) >> PAGE\_OFFSET\_BITS, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);  map\_pages(kernel\_page\_dir, KERNEL\_PAGE\_TABLE, get\_page\_info(kernel\_page\_dir, KERNEL\_PAGE\_TABLE), 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);  kernel\_address\_space.page\_dir = kernel\_page\_dir;  kernel\_address\_space.start = KERNEL\_MEMORY\_START;  kernel\_address\_space.end = KERNEL\_MEMORY\_END;  kernel\_address\_space.block\_table\_size = PAGE\_SIZE / sizeof(VirtMemoryBlock);  kernel\_address\_space.blocks = KERNEL\_MEMORY\_START;  kernel\_address\_space.block\_count = 1;  map\_pages(kernel\_page\_dir, kernel\_address\_space.blocks, alloc\_phys\_pages(1), 1, PAGE\_PRESENT | PAGE\_WRITABLE);  kernel\_address\_space.blocks[0].type = VMB\_RESERVED;  kernel\_address\_space.blocks[0].base = kernel\_address\_space.blocks;  kernel\_address\_space.blocks[0].length = PAGE\_SIZE;  }  /\* Physical memory manager \*/  size\_t get\_free\_memory\_size() {  return free\_page\_count << PAGE\_OFFSET\_BITS;  }  phyaddr alloc\_phys\_pages(size\_t count) {  if (free\_page\_count < count) return -1;  phyaddr result = -1;  **mutex\_get(&phys\_memory\_mutex, true);**  if (free\_phys\_memory\_pointer != -1) {  phyaddr cur\_block = free\_phys\_memory\_pointer;  do {  temp\_map\_page(cur\_block);  if (((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size == count) {  phyaddr next = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next;  phyaddr prev = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev;  temp\_map\_page(next);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  temp\_map\_page(prev);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next = next;  if (cur\_block == free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = next;  if (cur\_block == free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = -1;  }  }  result = cur\_block;  break;  } else if (((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size > count) {  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size -= count;  result = cur\_block + (((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size << PAGE\_OFFSET\_BITS);  break;  }  cur\_block = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next;  } while (cur\_block != free\_phys\_memory\_pointer);  if (result != -1) {  free\_page\_count -= count;  }  }  **mutex\_release(&phys\_memory\_mutex);**  return result;  }  void free\_phys\_pages(phyaddr base, size\_t count) {  **mutex\_get(&phys\_memory\_mutex, true);**  if (free\_phys\_memory\_pointer == -1) {  temp\_map\_page(base);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size = count;  free\_phys\_memory\_pointer = base;  } else {  phyaddr cur\_block = free\_phys\_memory\_pointer;  do {  temp\_map\_page(cur\_block);  if (cur\_block + (((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size << PAGE\_OFFSET\_BITS) == base) {  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size += count;  if (((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next == base + (count << PAGE\_OFFSET\_BITS)) {  phyaddr next1 = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next;  temp\_map\_page(next1);  phyaddr next2 = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next;  size\_t new\_count = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size;  temp\_map\_page(next2);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev = cur\_block;  temp\_map\_page(cur\_block);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next = next2;  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size += new\_count;  }  break;  } else if (base + (count << PAGE\_OFFSET\_BITS) == cur\_block) {  size\_t old\_count = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size;  phyaddr next = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next;  phyaddr prev = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev;  temp\_map\_page(next);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  temp\_map\_page(prev);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  temp\_map\_page(base);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next = next;  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size = count + old\_count;  break;  } else if ((cur\_block > base) || (((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next == free\_phys\_memory\_pointer)) {  phyaddr prev = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next;  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev = base;  temp\_map\_page(prev);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next = base;  temp\_map\_page(base);  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next = cur\_block;  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->prev = prev;  ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->size = count;  break;  }  cur\_block = ((volatile PhysMemoryBlock\*)TEMP\_PAGE)->next;  } while (cur\_block != free\_phys\_memory\_pointer);  if (base < free\_phys\_memory\_pointer) {  free\_phys\_memory\_pointer = base;  }    }  free\_page\_count += count;  **mutex\_release(&phys\_memory\_mutex);**  }  /\* Lowlevel virtual memory manager \*/  ...  /\* Highlevel virtual memory manager \*/  static inline bool is\_blocks\_overlapped(void \*base1, size\_t size1, void \*base2, size\_t size2) {  return ((base1 >= base2) && (base1 < base2 + size2)) || ((base2 >= base1) && (base2 < base1 + size1));  }  void \*alloc\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, phyaddr paddr, size\_t count, unsigned int flags) {  VirtMemoryBlockType type = VMB\_IO\_MEMORY;  size\_t i;  **mutex\_get(&(address\_space->mutex), true);**  if (vaddr == NULL) {  vaddr = address\_space->end - (count << PAGE\_OFFSET\_BITS) + 1;  for (i = 0; i < address\_space->block\_count; i++) {  if (is\_blocks\_overlapped(address\_space->blocks[i].base, address\_space->blocks[i].length, vaddr, count)) {  vaddr = address\_space->blocks[i].base - (count << PAGE\_OFFSET\_BITS);  } else {  break;  }  }  } else {  if ((vaddr >= address\_space->start) && (vaddr + (count << PAGE\_OFFSET\_BITS) < address\_space->end)) {  for (i = 0; i < address\_space->block\_count; i++) {  if (is\_blocks\_overlapped(address\_space->blocks[i].base, address\_space->blocks[i].length, vaddr, count << PAGE\_OFFSET\_BITS)) {  vaddr = NULL;  break;  } else if (address\_space->blocks[i].base < vaddr) {  break;  }  }  } else {  vaddr = NULL;  }  }  if (vaddr != NULL) {  if (paddr == -1) {  paddr = alloc\_phys\_pages(count);  type = VMB\_MEMORY;  }  if (paddr != -1) {  if (map\_pages(address\_space->page\_dir, vaddr, paddr, count, flags)) {  if (address\_space->block\_count == address\_space->block\_table\_size) {  size\_t new\_size = address\_space->block\_table\_size \* sizeof(VirtMemoryBlock);  new\_size = (new\_size + PAGE\_OFFSET\_MASK) & ~PAGE\_OFFSET\_MASK;  new\_size += PAGE\_SIZE;  new\_size = new\_size >> PAGE\_OFFSET\_BITS;  if (&kernel\_address\_space != address\_space) {  VirtMemoryBlock \*new\_table = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, new\_size,  PAGE\_PRESENT | PAGE\_WRITABLE);  if (new\_table) {  memcpy(new\_table, address\_space->blocks, address\_space->block\_table\_size \* sizeof(VirtMemoryBlock));  free\_virt\_pages(&kernel\_address\_space, address\_space->blocks, 0);  address\_space->blocks = new\_table;  } else {  goto fail;  }  } else {  phyaddr new\_page = alloc\_phys\_pages(1);  if (new\_page == -1) {  goto fail;  } else {  VirtMemoryBlock \*main\_block = &(address\_space->blocks[address\_space->block\_count - 1]);  if (map\_pages(address\_space->page\_dir, main\_block->base + main\_block->length, new\_page, 1,  PAGE\_PRESENT | PAGE\_WRITABLE)) {  main\_block->length += PAGE\_SIZE;  } else {  free\_phys\_pages(new\_page, 1);  }  }  }  address\_space->block\_table\_size = (new\_size << PAGE\_OFFSET\_BITS) / sizeof(VirtMemoryBlock);  }  memcpy(address\_space->blocks + i + 1, address\_space->blocks + i,  (address\_space->block\_count - i) \* sizeof(VirtMemoryBlock));  address\_space->block\_count++;  address\_space->blocks[i].type = type;  address\_space->blocks[i].base = vaddr;  address\_space->blocks[i].length = count << PAGE\_OFFSET\_BITS;  } else {  fail:  map\_pages(address\_space->page\_dir, vaddr, 0, count, 0);  free\_phys\_pages(paddr, count);  vaddr = NULL;  }  }  }  **mutex\_release(&(address\_space->mutex));**  return vaddr;  }  bool free\_virt\_pages(AddressSpace \*address\_space, void \*vaddr, unsigned int flags) {  size\_t i;  **mutex\_get(&(address\_space->mutex), true);**  for (i = 0; i < address\_space->block\_count; i++) {  if ((address\_space->blocks[i].base <= vaddr) && (address\_space->blocks[i].base + address\_space->blocks[i].length > vaddr)) {  break;  }  }  if (i < address\_space->block\_count) {  if (address\_space->blocks[i].type = VMB\_MEMORY) {  free\_phys\_pages(get\_page\_info(address\_space->page\_dir, vaddr) & ~PAGE\_OFFSET\_MASK, address\_space->blocks[i].length >> PAGE\_OFFSET\_BITS);  }  address\_space->block\_count--;  memcpy(address\_space->blocks + i, address\_space->blocks + i + 1, (address\_space->block\_count - i) \* sizeof(VirtMemoryBlock));  **mutex\_release(&(address\_space->mutex));**  return true;  } else {  **mutex\_release(&(address\_space->mutex));**  return false;  }  }  Теперь, менеджер памяти сможет корректно обрабатывать ситуацию, когда к нему обращаются одновременно сразу несколько нитей.  Вот и всё на сегодня! В следующем выпуске можно будет начать реализацию переключения задач.  Кстати, в функции map\_pages содержится ошибка (ну не совсем ошибка, а отсутствие volatile в нужном месте или ещё что) из-за которой при уровне оптимизации -O2 она не способна корректно примонтировать страницу, если при этом приходится создавать новую таблицу страниц. Если вы сможете её найти, я буду вам благодарен :-) |

## Пишем свою операционную систему. Улучшение стандартной библиотеки

|  |
| --- |
| Причина падения ОС при включении оптимизации была найдена - memset (как и memcpy) меняет значение регистра EDI, который считается неизменным по соглашению вызова C (вызываемая функция обязана сохранить его значение).  Я принял достаточно радикальное решение - часть функций стандартной библиотеки будет вынесена в ассемблерный файл для лучшей оптимизации (строковые операции компилятор делает неэффективно) и простоты написания. Теперь в нашем проекте появляется файл stdlib.i386.asm:  format ELF  public memset  public memsetw  public memcpy  public memcmp  public memchr  section ".text" executable  ; void memset(void \*mem, char value, size\_t count);  memset:  push edi  mov edi, [esp + 8]  mov edx, [esp + 12]  mov al, dl  shl eax, 8  mov al, dl  shl eax, 8  mov al, dl  shl eax, 8  mov al, dl  mov edx, [esp + 16]  mov ecx, edx  shr ecx, 2  rep stosd  mov ecx, edx  and ecx, 3  rep stosb  pop edi  ret  ; void memsetw(void \*mem, uint16 value, size\_t count);  memsetw:  push edi  mov edi, [esp + 8]  mov edx, [esp + 12]  mov ax, dx  shl eax, 16  mov ax, dx  mov edx, [esp + 16]  mov ecx, edx  shr ecx, 1  rep stosd  mov ecx, edx  and ecx, 1  rep stosw  pop edi  ret  ; void memcpy(void \*dest, void \*src, size\_t count);  memcpy:  push esi edi  mov edi, [esp + 12]  mov esi, [esp + 16]  mov edx, [esp + 20]  cmp edi, esi  jb .reversed  mov ecx, edx  shr ecx, 2  rep movsd  mov ecx, edx  and ecx, 3  rep movsb  .exit:  pop edi esi  ret  .reversed:  add esi, edx  add edi, edx  dec esi  dec edi  std  mov ecx, edx  and ecx, 3  rep movsb  mov ecx, edx  shr ecx, 2  rep movsd  cld  jmp .exit  ; int memcmp(void \*mem1, void \*mem2, size\_t count);  memcmp:  push esi edi  mov edi, [esp + 12]  mov esi, [esp + 16]  mov ecx, [esp + 20]  repe cmpsb  seta al  setb dl  sub al, dl  movzx eax, al  pop edi esi  ret  ; void \*memchr(void \*mem, char value, size\_t count);  memchr:  push edi  mov edi, [esp + 8]  mov eax, [esp + 12]  mov ecx, [esp + 16]  repne scasb  jne .not\_found  dec edi  mov eax, edi  .exit:  pop edi  ret  .not\_found:  xor eax, eax  jmp .exit  Заголовочный файл stdlib.h остаётся неизменным, а вот stdlib.c укорачивается:  #include "stdlib.h"  size\_t strlen(char \*str) {  return (char\*)memchr(str, '\0', -1) - str;  }  void strcpy(char \*dest, char \*src) {  memcpy(dest, src, strlen(src) + 1);  }  void strncpy(char \*dest, char \*src, size\_t max\_count) {  size\_t len = min(max\_count - 1, strlen(src));  memcpy(dest, src, len);  dest[len] = '\0';  }  int strcmp(char \*str1, char \*str2) {  return memcmp(str1, str2, strlen(str1) + 1);  }  char \*strchr(char \*str, char value) {  return memchr(str, value, strlen(str));  }  bool mutex\_get(Mutex \*mutex, bool wait) {  bool old\_value = true;  do {  asm("xchg (,%1,), %0":"=a"(old\_value):"b"(mutex),"a"(old\_value));  } while (old\_value && wait);  return !old\_value;  }  void mutex\_release(Mutex \*mutex) {  \*mutex = false;  }  К тому же я избавился от ряда не очень "красивых" ассемблерных вставок оставив лишь полностью корректные (в том числе и в других файлах - memory\_manager.c, interrupts.c).  Теперь наша ОС работает полностью корректно и не падает при любом уровне оптимизации (O0, O1, O2, O3, Os). Ещё немного о многозадачности Оставшееся время расскажу ещё немного о реализации многозадачности. Для переключения нитей можно использовать самые разные варианты, мой выбор остановился на одном из них.  В обработчике прерывания от таймера я собираюсь сохранить в стек все регистры процессора, а также адрес temp\_page, затем записать текущий указатель стека в структуру описателя нити. После этого производится поиск следующей нити для выполнения (в том числе это может оказаться и та же самая). Когда она найдена, из её описателя восстанавливается её указатель стека, а затем все регистры из стека. В итоге весь контекст задачи хранится в её стеке и не нужны дополнительные структуры данных. Для создания новой нити нужно заполнить её стек так, как будто только что случилось прерывание таймера (запихнуть туда адрес возврата, регистр флагов, регистры общего назначения и т. д.). Именно поэтому нам нужна полная предсказуемость содержимого стека. Возможно, обработчик прерывания таймера придётся писать на Assembler. Двунаправленные связанные списки Как список процессов, так и список их нитей нужно как-то хранить. Причём в форме удобной для быстрого переключения, добавления и удаления нитей. Я собираюсь использовать для этого двунаправленные связанные кольцевые списки (на примере файловой системы и менеджера физической памяти вы уже могли догадаться, что я к ним крайне не равнодушен). Каждое переключение достаточно брать следующий элемент списка. Переход из начала в конец произойдёт сам собой.  Чтобы работать со списками было удобно, реализую три функции стандартной библиотеки специально для них:  #ifndef STDLIB\_H  #define STDLIB\_H  typedef enum {  false = 0,  true = 1  } bool;  #define NULL ((void\*)0)  typedef unsigned char uint8;  typedef signed char int8;  typedef unsigned short uint16;  typedef signed short int16;  typedef unsigned int uint32;  typedef signed int int32;  typedef unsigned long long uint64;  typedef signed long long int64;  #ifdef \_\_x86\_64\_\_  typedef uint64 size\_t;  #else  typedef uint32 size\_t;  #endif  typedef bool Mutex;  **typedef struct \_ListItem ListItem;**  **typedef struct {**  **ListItem \*first;**  **size\_t count;**  **Mutex mutex;**  **} List;**  **struct \_ListItem {**  **ListItem \*next;**  **ListItem \*prev;**  **List \*list;**  **};**  #define min(a, b) (((a) > (b)) ? (b) : (a))  #define max(a, b) (((a) > (b)) ? (a) : (b))  #define outportb(port, value) asm("outb %b0, %w1"::"a"(value),"d"(port));  #define outportw(port, value) asm("outw %w0, %w1"::"a"(value),"d"(port));  #define outportl(port, value) asm("outl %0, %w1"::"a"(value),"d"(port));  #define inportb(port, out\_value) asm("inb %w1, %b0":"=a"(out\_value):"d"(port));  #define inportw(port, out\_value) asm("inw %w1, %w0":"=a"(out\_value):"d"(port));  #define inportl(port, out\_value) asm("inl %w1, %0":"=a"(out\_value):"d"(port));  void memset(void \*mem, char value, size\_t count);  void memsetw(void \*mem, uint16 value, size\_t count);  void memcpy(void \*dest, void \*src, size\_t count);  int memcmp(void \*mem1, void \*mem2, size\_t count);  void \*memchr(void \*mem, char value, size\_t count);  size\_t strlen(char \*str);  void strcpy(char \*dest, char \*src);  void strncpy(char \*dest, char\*src, size\_t max\_count);  int strcmp(char \*str1, char \*str2);  char \*strchr(char \*str, char value);  bool mutex\_get(Mutex \*mutex, bool wait);  void mutex\_release(Mutex \*mutex);  **void list\_init(List \*list);**  **void list\_append(List \*list, ListItem \*item);**  **void list\_remove(ListItem \*item);**  #endif  А вот и сам код функций. Он не использует какие-либо особенности процессора и может быть реализован даже в обычных пользовательских приложениях.  void list\_init(List \*list) {  list->first = NULL;  list->count = 0;  list->mutex = false;  }  void list\_append(List \*list, ListItem \*item) {  if (item->list == NULL) {  mutex\_get(&(list->mutex), true);  if (list->first) {  item->list = list;  item->next = list->first;  item->prev = list->first->prev;  item->prev->next = item;  item->next->prev = item;  } else {  item->next = item;  item->prev = item;  list->first = item;  }  list->count++;  mutex\_release(&(list->mutex));  }  }  void list\_remove(ListItem \*item) {  mutex\_get(&(list->mutex), true);  if (item->list->first == item) {  item->list->first = item->next;  if (item->list->first == item) {  item->list->first = NULL;  }  }  item->next->prev = item->prev;  item->prev->next = item->next;  list->count--;  mutex\_release(&(list->mutex));  }  **Внимание!** Для перебора списка захват семафора не требуется!  Как это работает? Мы создаём новую структуру данных (например, описатель процесса или нити), самым первым полем структуры должно являться поле с любым именем и типом ListItem. После этого мы можем передавать указатель на новую структуру функциям list\_add и list\_remove в качестве параметра item, совершая явное приведение типов. Новый Makefile Мы добавили новый файл - [stdlib.i386.asm.](http://stdlib.i386.asm/) Следует описать правило для его сборки:  ifdef OS  LDFLAGS = -mi386pe  else  LDFLAGS = -melf\_i386  endif  CFLAGS = -m32 -ffreestanding -O3  all: script.ld startup.o **stdlib\_asm.o** stdlib.o main.o memory\_manager.o interrupts.o tty.o  ld $(LDFLAGS) -T script.ld -o kernel.bin startup.o **stdlib\_asm.o** stdlib.o main.o memory\_manager.o interrupts.o tty.o  objcopy kernel.bin -O binary  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  stdlib.o: stdlib.c stdlib.h  gcc -c $(CFLAGS) -o stdlib.o stdlib.c  **stdlib\_asm.o: stdlib.i386.asm**  **fasm stdlib.i386.asm stdlib\_asm.o**  main.o: main.c stdlib.h interrupts.h tty.h  gcc -c $(CFLAGS) -o main.o main.c  memory\_manager.o: memory\_manager.c memory\_manager.h stdlib.h  gcc -c $(CFLAGS) -o memory\_manager.o memory\_manager.c  interrupts.o: interrupts.c interrupts.h stdlib.h  gcc -c $(CFLAGS) -o interrupts.o interrupts.c  tty.o: tty.c tty.h stdlib.h  gcc -c $(CFLAGS) -o tty.o tty.c  clean:  rm -v \*.o kernel.bin  Ну вот и всё. У нас есть практически полноценный менеджер памяти и стандартная библиотека с поддержкой многопоточности, то есть всё, чтобы приступить к реализации многозадачности. Кстати, в качестве "домашнего задания" можете добавить потокобезопасность в драйвер экрана (tty.c).  Мой e-mail для всех вопросов: [kiv.apple@gmail.com](mailto:apple@gmail.com).  До встречи! |

## Пишем свою операционную систему. Начало реализации многозадачности

|  |
| --- |
| Во-первых, вот исправление функции list\_remove из stdlib.c, чтобы она нормально компилировалась:  void list\_remove(ListItem \*item) {  mutex\_get(&(**item->**list->mutex), true);  if (**item->**list->first == item) {  **item->**list->first = item->next;  if (**item->**list->first == item) {  **item->**list->first = NULL;  }  }  item->next->prev = item->prev;  item->prev->next = item->next;  **item->**list->count--;  mutex\_release(&(**item->**list->mutex));  }  Во-вторых, сегодня мы начнём писать функции для поддержки многозадачности. Как обычно вначале определимся с прототипами функций в заголовочном файле multitasking.h:  #ifndef MULTITASKING\_H  #define MULTITASKING\_H  #include "stdlib.h"  typedef struct {  ListItem list\_item;  AddressSpace address\_space;  bool suspend;  size\_t thread\_count;  char name[256];  } Process;  typedef struct {  ListItem list\_item;  Process \*process;  bool suspend;  void \*stack\_base;  size\_t stack\_size;  void \*stack\_pointer;  } Thread;  List process\_list;  List thread\_list;  Process \*current\_process;  Thread \*current\_thread;  Process \*kernel\_process;  Thread \*kernel\_thread;  void init\_multitasking();  void switch\_task();  Thread \*create\_thread(Process \*process, void \*entry\_point, size\_t stack\_size, bool kernel, bool suspend);  #endif  Структуры описателей процессов и потоков предусматривают множество полей, которые понадобятся нам в будущем, а вот набор функций пока очень скуден - не реализовано даже завершение нитей, работы с процессами нет вообще. Но для начала хватит, остальное допишем в следующих выпусках.  Многозадачность как таковую мы сегодня не сделаем - функция create\_thread будет реализована не до конца, но мы вплотную приблизимся к ней.  Начинается файл multitasking.c с простой функции init\_multitasking, которая создаёт структуры процесса и нити ядра, а также устанавливает обработчик прерывания таймера:  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  #include "multitasking.h"  void task\_switch\_int\_handler();  void init\_multitasking() {  list\_init(&process\_list);  list\_init(&thread\_list);  kernel\_process = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  kernel\_process->address\_space.page\_dir = kernel\_page\_dir;  kernel\_process->suspend = false;  kernel\_process->thread\_count = 1;  strncpy(kernel\_process->name, "Kernel", sizeof(kernel\_process->name));  list\_append((List\*)&process\_list, (ListItem\*)kernel\_process);  kernel\_thread = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  kernel\_thread->process = kernel\_process;  kernel\_thread->suspend = false;  kernel\_thread->stack\_size = PAGE\_SIZE;  list\_append((List\*)&thread\_list, (ListItem\*)kernel\_thread);  current\_process = kernel\_process;  current\_thread = kernel\_thread;  set\_int\_handler(irq\_base, task\_switch\_int\_handler, 0x8E);  }  Эта функция в настоящий момент упрощена - мы не создаём нормального личного адресного пространства для процесса ядра, а также не задаём базовый адрес стека для главной нити ядра. Второе впрочем нужно только для будущей функции завершения нити (чтобы освободить память из-под стека), а главная нить ядра по моей задумке никто завершена не будет (но может уходить в спячку).  Дальше следует новый обработчик прерывания таймера:  IRQ\_HANDLER(task\_switch\_int\_handler) {  asm("movl %%esp, %0":"=a"(current\_thread->stack\_pointer));  do {  current\_thread = (Thread\*)current\_thread->list\_item.next;  current\_process = current\_thread->process;  } while (current\_thread->suspend || current\_process->suspend);  asm("movl %0, %%cr3"::"a"(current\_process->address\_space.page\_dir));  asm("movl %0, %%esp"::"a"(current\_thread->stack\_pointer));  }  Вот так просто переключаются задачи. Запоминаем текущий указатель стека, ищем следующую нить для выполнения, переключаем каталог страниц, переключаем стек. Всё! Мы теперь находимся в контексте новой нити. Однако на самом деле такой код использовать не получится - как создавать новые нити? Функция create\_thread должна заполнить стек новой нити как-будто он находится в середине обработчика прерывания. То, что помещается в стек процессором (адрес возврата, регистр флагов), а также ассемблерной вставкой (все регистры процессора) известно, но вот поведение самого task\_switch\_int\_handler - нет. При разных настройках компилятора он может как создавать стековый фрейм, так и нет. В итоге в стеке будет на одно двойное слово больше или меньше, а это всё портит. До следующего выпуска предстоит придумать как с этим бороться. Я пока не знаю красивого решения на Си (если бы мы писали ОС на чистом Assembler решение проблемы было бы элементарно и я так уже много раз делал).  Ну а пока идём дальше - функция switch\_task. Её вызывает процесс, если ему больше нечего делать - система должна отдать остаток времени другой задаче. Это обозначает, что нить хочет дать возможность выполнится другим, потому что она сможет продолжить лишь получив данные от кого-то другого. Никакого механизма обмена данными между нитями у нас пока нет, поэтому просто остановим процессор инструкцией HLT. Она заставляет его перейти в режим пониженного энергопотребления и продолжить работу лишь после прихода прерывания.  void switch\_task() {  asm("hlt");  }  Сразу же приведу пример, где такая команда уместна - в функции in\_char tty.c:  #include <stdarg.h>  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  **#include "multitasking.h"**  #include "tty.h"  #include "scancodes.h"  ...  char in\_char(bool wait) {  static bool shift = false;  uint8 chr;  do {  chr = in\_scancode();  switch (chr) {  case 0x2A:  case 0x36:  shift = true;  break;  case 0x2A + 0x80:  case 0x36 + 0x80:  shift = false;  break;  }  if (chr & 0x80) {  chr = 0;  }  if (shift) {  chr = scancodes\_shifted[chr];  } else {  chr = scancodes[chr];  }  **if ((!chr) && wait) {**  **asm("hlt");**  **}**  } while (wait && (!chr));  return chr;  }  Такое изменение в будущем повысит быстродействие системы, а уже сейчас - энергопотребление.  Ну вот осталась последняя функция файла multitasking.c - create\_thread:  Thread \*create\_thread(Process \*process, void \*entry\_point, size\_t stack\_size, bool kernel, bool suspend) {  Thread \*thread = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  thread->process = process;  thread->suspend = suspend;  // ... вот тут должно быть создание и заполнение стека нити ...  list\_append((List\*)&thread\_list, (ListItem\*)thread);  process->thread\_count++;  return thread;  }  Это заготовка, но не готовая функция. Её вызов в том виде, в котором она сейчас есть с suspend != false приведёт к краху системы на следующем тике таймера, потому что мы не создали для нити нормальный стек. Нам ещё многое предстоит написать.  Пока добавим в kernel\_main инициализацию многозадачности:  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  **#include "multitasking.h"**  #include "tty.h"  typedef struct {  uint64 base;  uint64 size;  } BootModuleInfo;  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  init\_memory\_manager(memory\_map);  init\_interrupts();  **init\_multitasking();**  init\_tty();  set\_text\_attr(15);  printf("Welcome to MyOS!\n");  char string[10];  in\_string(string, sizeof(string));  out\_string(string);  }  Ну и наконец новый Makefile:  ifdef OS  LDFLAGS = -mi386pe  else  LDFLAGS = -melf\_i386  endif  CFLAGS = -m32 -ffreestanding -O3  all: script.ld startup.o stdlib\_asm.o stdlib.o main.o memory\_manager.o interrupts.o multitasking.o tty.o  ld $(LDFLAGS) -T script.ld -o kernel.bin startup.o stdlib\_asm.o stdlib.o main.o memory\_manager.o interrupts.o multitasking.o tty.o  objcopy kernel.bin -O binary  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  stdlib.o: stdlib.c stdlib.h  gcc -c $(CFLAGS) -o stdlib.o stdlib.c  stdlib\_asm.o: stdlib.i386.asm  fasm stdlib.i386.asm stdlib\_asm.o  main.o: main.c stdlib.h interrupts.h multitasking.h tty.h  gcc -c $(CFLAGS) -o main.o main.c  memory\_manager.o: memory\_manager.c memory\_manager.h stdlib.h  gcc -c $(CFLAGS) -o memory\_manager.o memory\_manager.c  interrupts.o: interrupts.c interrupts.h memory\_manager.h stdlib.h  gcc -c $(CFLAGS) -o interrupts.o interrupts.c  multitasking.o: multitasking.c multitasking.h interrupts.h memory\_manager.h stdlib.h  gcc -c $(CFLAGS) -o multitasking.o multitasking.c  tty.o: tty.c tty.h multitasking.h interrupts.h stdlib.h  gcc -c $(CFLAGS) -o tty.o tty.c  clean:  rm -v \*.o kernel.bin  Ну вот и всё. В поведении ядра пока ничего не изменилось, но мы подготовили всё к тому, чтобы начать создавать новые нити. Заодно по новому обработчику таймера вы можете наглядно понять новый принцип работы нашего переключателя задач. Заключение У меня сразу две хороших новости - во-первых, моя рассылка получила статус "Серебряной", во-вторых, у неё уже 152 подписчика (вероятно, столь быстрый рост их числа связан с появлением рассылки в общем каталоге из-за изменения статуса).  Теперь я могу прикладывать к выпускам файлы, поэтому сразу же воспользуюсь этим, приложив полный архив исходных текстов нашей ОС, который точно работоспособен. Я не стал убирать скомпилированный образ диска (всё равно ZIP его очень не плохо сжал), чтобы можно было легко проверить систему.  Раз вас стало больше, я могу вновь задать свои вопросы:  1) Как должна называться наша ОС? 2) Как по вашему мнению лучше реализовать переключение задач (сделать стек при обработке прерывания более предсказуемым)?  И вообще, пишите любые свои замечания, предложения, пожелания, вопросы мне на адрес [kiv.apple@gmail.com](mailto:apple@gmail.com). До встречи! |

## Пишем свою операционную систему. Начало реализации многозадачности

|  |
| --- |
| Во-первых, вот исправление функции list\_remove из stdlib.c, чтобы она нормально компилировалась:  void list\_remove(ListItem \*item) {  mutex\_get(&(**item->**list->mutex), true);  if (**item->**list->first == item) {  **item->**list->first = item->next;  if (**item->**list->first == item) {  **item->**list->first = NULL;  }  }  item->next->prev = item->prev;  item->prev->next = item->next;  **item->**list->count--;  mutex\_release(&(**item->**list->mutex));  }  Во-вторых, сегодня мы начнём писать функции для поддержки многозадачности. Как обычно вначале определимся с прототипами функций в заголовочном файле multitasking.h:  #ifndef MULTITASKING\_H  #define MULTITASKING\_H  #include "stdlib.h"  typedef struct {  ListItem list\_item;  AddressSpace address\_space;  bool suspend;  size\_t thread\_count;  char name[256];  } Process;  typedef struct {  ListItem list\_item;  Process \*process;  bool suspend;  void \*stack\_base;  size\_t stack\_size;  void \*stack\_pointer;  } Thread;  List process\_list;  List thread\_list;  Process \*current\_process;  Thread \*current\_thread;  Process \*kernel\_process;  Thread \*kernel\_thread;  void init\_multitasking();  void switch\_task();  Thread \*create\_thread(Process \*process, void \*entry\_point, size\_t stack\_size, bool kernel, bool suspend);  #endif  Структуры описателей процессов и потоков предусматривают множество полей, которые понадобятся нам в будущем, а вот набор функций пока очень скуден - не реализовано даже завершение нитей, работы с процессами нет вообще. Но для начала хватит, остальное допишем в следующих выпусках.  Многозадачность как таковую мы сегодня не сделаем - функция create\_thread будет реализована не до конца, но мы вплотную приблизимся к ней.  Начинается файл multitasking.c с простой функции init\_multitasking, которая создаёт структуры процесса и нити ядра, а также устанавливает обработчик прерывания таймера:  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  #include "multitasking.h"  void task\_switch\_int\_handler();  void init\_multitasking() {  list\_init(&process\_list);  list\_init(&thread\_list);  kernel\_process = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  kernel\_process->address\_space.page\_dir = kernel\_page\_dir;  kernel\_process->suspend = false;  kernel\_process->thread\_count = 1;  strncpy(kernel\_process->name, "Kernel", sizeof(kernel\_process->name));  list\_append((List\*)&process\_list, (ListItem\*)kernel\_process);  kernel\_thread = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  kernel\_thread->process = kernel\_process;  kernel\_thread->suspend = false;  kernel\_thread->stack\_size = PAGE\_SIZE;  list\_append((List\*)&thread\_list, (ListItem\*)kernel\_thread);  current\_process = kernel\_process;  current\_thread = kernel\_thread;  set\_int\_handler(irq\_base, task\_switch\_int\_handler, 0x8E);  }  Эта функция в настоящий момент упрощена - мы не создаём нормального личного адресного пространства для процесса ядра, а также не задаём базовый адрес стека для главной нити ядра. Второе впрочем нужно только для будущей функции завершения нити (чтобы освободить память из-под стека), а главная нить ядра по моей задумке никто завершена не будет (но может уходить в спячку).  Дальше следует новый обработчик прерывания таймера:  IRQ\_HANDLER(task\_switch\_int\_handler) {  asm("movl %%esp, %0":"=a"(current\_thread->stack\_pointer));  do {  current\_thread = (Thread\*)current\_thread->list\_item.next;  current\_process = current\_thread->process;  } while (current\_thread->suspend || current\_process->suspend);  asm("movl %0, %%cr3"::"a"(current\_process->address\_space.page\_dir));  asm("movl %0, %%esp"::"a"(current\_thread->stack\_pointer));  }  Вот так просто переключаются задачи. Запоминаем текущий указатель стека, ищем следующую нить для выполнения, переключаем каталог страниц, переключаем стек. Всё! Мы теперь находимся в контексте новой нити. Однако на самом деле такой код использовать не получится - как создавать новые нити? Функция create\_thread должна заполнить стек новой нити как-будто он находится в середине обработчика прерывания. То, что помещается в стек процессором (адрес возврата, регистр флагов), а также ассемблерной вставкой (все регистры процессора) известно, но вот поведение самого task\_switch\_int\_handler - нет. При разных настройках компилятора он может как создавать стековый фрейм, так и нет. В итоге в стеке будет на одно двойное слово больше или меньше, а это всё портит. До следующего выпуска предстоит придумать как с этим бороться. Я пока не знаю красивого решения на Си (если бы мы писали ОС на чистом Assembler решение проблемы было бы элементарно и я так уже много раз делал).  Ну а пока идём дальше - функция switch\_task. Её вызывает процесс, если ему больше нечего делать - система должна отдать остаток времени другой задаче. Это обозначает, что нить хочет дать возможность выполнится другим, потому что она сможет продолжить лишь получив данные от кого-то другого. Никакого механизма обмена данными между нитями у нас пока нет, поэтому просто остановим процессор инструкцией HLT. Она заставляет его перейти в режим пониженного энергопотребления и продолжить работу лишь после прихода прерывания.  void switch\_task() {  asm("hlt");  }  Сразу же приведу пример, где такая команда уместна - в функции in\_char tty.c:  #include <stdarg.h>  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  **#include "multitasking.h"**  #include "tty.h"  #include "scancodes.h"  ...  char in\_char(bool wait) {  static bool shift = false;  uint8 chr;  do {  chr = in\_scancode();  switch (chr) {  case 0x2A:  case 0x36:  shift = true;  break;  case 0x2A + 0x80:  case 0x36 + 0x80:  shift = false;  break;  }  if (chr & 0x80) {  chr = 0;  }  if (shift) {  chr = scancodes\_shifted[chr];  } else {  chr = scancodes[chr];  }  **if ((!chr) && wait) {**  **asm("hlt");**  **}**  } while (wait && (!chr));  return chr;  }  Такое изменение в будущем повысит быстродействие системы, а уже сейчас - энергопотребление.  Ну вот осталась последняя функция файла multitasking.c - create\_thread:  Thread \*create\_thread(Process \*process, void \*entry\_point, size\_t stack\_size, bool kernel, bool suspend) {  Thread \*thread = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  thread->process = process;  thread->suspend = suspend;  // ... вот тут должно быть создание и заполнение стека нити ...  list\_append((List\*)&thread\_list, (ListItem\*)thread);  process->thread\_count++;  return thread;  }  Это заготовка, но не готовая функция. Её вызов в том виде, в котором она сейчас есть с suspend != false приведёт к краху системы на следующем тике таймера, потому что мы не создали для нити нормальный стек. Нам ещё многое предстоит написать.  Пока добавим в kernel\_main инициализацию многозадачности:  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  **#include "multitasking.h"**  #include "tty.h"  typedef struct {  uint64 base;  uint64 size;  } BootModuleInfo;  void kernel\_main(uint8 boot\_disk\_id, void \*memory\_map, BootModuleInfo \*boot\_module\_list) {  init\_memory\_manager(memory\_map);  init\_interrupts();  **init\_multitasking();**  init\_tty();  set\_text\_attr(15);  printf("Welcome to MyOS!\n");  char string[10];  in\_string(string, sizeof(string));  out\_string(string);  }  Ну и наконец новый Makefile:  ifdef OS  LDFLAGS = -mi386pe  else  LDFLAGS = -melf\_i386  endif  CFLAGS = -m32 -ffreestanding -O3  all: script.ld startup.o stdlib\_asm.o stdlib.o main.o memory\_manager.o interrupts.o multitasking.o tty.o  ld $(LDFLAGS) -T script.ld -o kernel.bin startup.o stdlib\_asm.o stdlib.o main.o memory\_manager.o interrupts.o multitasking.o tty.o  objcopy kernel.bin -O binary  startup.o: startup.i386.asm  fasm startup.i386.asm startup.o  stdlib.o: stdlib.c stdlib.h  gcc -c $(CFLAGS) -o stdlib.o stdlib.c  stdlib\_asm.o: stdlib.i386.asm  fasm stdlib.i386.asm stdlib\_asm.o  main.o: main.c stdlib.h interrupts.h multitasking.h tty.h  gcc -c $(CFLAGS) -o main.o main.c  memory\_manager.o: memory\_manager.c memory\_manager.h stdlib.h  gcc -c $(CFLAGS) -o memory\_manager.o memory\_manager.c  interrupts.o: interrupts.c interrupts.h memory\_manager.h stdlib.h  gcc -c $(CFLAGS) -o interrupts.o interrupts.c  multitasking.o: multitasking.c multitasking.h interrupts.h memory\_manager.h stdlib.h  gcc -c $(CFLAGS) -o multitasking.o multitasking.c  tty.o: tty.c tty.h multitasking.h interrupts.h stdlib.h  gcc -c $(CFLAGS) -o tty.o tty.c  clean:  rm -v \*.o kernel.bin  Ну вот и всё. В поведении ядра пока ничего не изменилось, но мы подготовили всё к тому, чтобы начать создавать новые нити. Заодно по новому обработчику таймера вы можете наглядно понять новый принцип работы нашего переключателя задач. Заключение У меня сразу две хороших новости - во-первых, моя рассылка получила статус "Серебряной", во-вторых, у неё уже 152 подписчика (вероятно, столь быстрый рост их числа связан с появлением рассылки в общем каталоге из-за изменения статуса).  Теперь я могу прикладывать к выпускам файлы, поэтому сразу же воспользуюсь этим, приложив полный архив исходных текстов нашей ОС, который точно работоспособен. Я не стал убирать скомпилированный образ диска (всё равно ZIP его очень не плохо сжал), чтобы можно было легко проверить систему.  Раз вас стало больше, я могу вновь задать свои вопросы:  1) Как должна называться наша ОС? 2) Как по вашему мнению лучше реализовать переключение задач (сделать стек при обработке прерывания более предсказуемым)?  И вообще, пишите любые свои замечания, предложения, пожелания, вопросы мне на адрес [kiv.apple@gmail.com](mailto:apple@gmail.com). До встречи! |

Пишем свою операционную систему. Улучшение обработки IRQ прерываний и многозадачности

|  |
| --- |
| Давненько не писал новых выпусков... пришло время это исправить.  Сегодня мы сильно изменим структуру обработки IRQ-прерываний. Раньше, каждое прерывание описывалось отдельно. Теперь мы сведём обработку всех прерываний в одну функцию. Если сейчас выигрыш от этого не очевиден, то потом он будет заметнее. Ведь в конечном счёте ядро должно при возникновении IRQ-прерывания отправить сообщение программе-драйверу (у нас же микроядро). И меняться в этом сообщении будет только номер прерывания. Не будем же мы писать 16 разных функций, различающихся лишь 1 цифрой?   Для начала создадим файл interrupts.asm, который будет содержать кое-какие полезные для нас описания, которые нельзя написать на Си ввиду их низкоуровневости.  format ELF  public irq\_handlers  extrn irq\_handler  section ".text" executable  macro IRQ\_handler index {  IRQ # index # \_handler:  push eax  mov eax, index - 1  jmp common\_irq\_handler  }  rept 16 i {  IRQ\_handler i  }  ; Обработчик всех IRQ прерываний  common\_irq\_handler:  push ebx ecx edx esi edi ebp  push ds es fs gs  mov ecx, 16  mov ds, cx  mov es, cx  mov fs, cx  mov gs, cx  mov edx, esp  push edx  push eax  call irq\_handler  add esp, 2 \* 4  pop gs fs es ds  pop ebp edi esi edx ecx ebx  mov al, 0x20  out 0x20, al  out 0xA0, al  pop eax  iretd  section ".data" writable  ; Таблица обработчиков IRQ прерываний  irq\_handlers:  rept 16 i {  dd IRQ # i # \_handler  }  С помощью макроса создаётся обработчик для каждого из 16 IRQ-прерываний (быстро программно узнать какой номер у текущего прерывания, насколько мне известно, способа нет), который очень простой - запихнуть в стек регистр EAX, поместить в EAX номер прерывания, перейти на основной обработчик (этот код занимает всего 10 байт для одного IRQ прерывания, или 160 байт для всех прерываний, так что можете не беспокоится, что ядро слишком растолстеет).  Основной обработчик прерываний на самом деле тоже не окончательный - его задача подготовить окружение для функции на Си, которая уже сделает всё, что нужно. Это заключается в сохранении уже всех регистров, а не только EAX в стек, переключение сегментных регистров на сегмент данных ядра, передача в функцию на Си номера прерывания и указателя на структуру, хранящую значения регистров.  Теперь модифицируем interrupts.c:  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  **#include "multitasking.h"**  **#include "tty.h"**  **void (\*irq\_handlers[])();**  **void irq\_handler(uint32 index, Registers \*regs);**  typedef struct {  uint16 address\_0\_15;  uint16 selector;  uint8 reserved;  uint8 type;  uint16 address\_16\_31;  } \_\_attribute\_\_((packed)) IntDesc;  typedef struct {  uint16 limit;  void \*base;  } \_\_attribute\_\_((packed)) IDTR;  IntDesc \*idt;  void init\_interrupts() {  idt = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);  memset(idt, 0, 256 \* sizeof(IntDesc));  volatile IDTR idtr = {256 \* sizeof(IntDesc), idt};  asm("lidt (,%0,)"::"a"(&idtr));  irq\_base = 0x20;  irq\_count = 16;  outportb(0x20, 0x11);  outportb(0x21, irq\_base);  outportb(0x21, 4);  outportb(0x21, 1);  outportb(0xA0, 0x11);  outportb(0xA1, irq\_base + 8);  outportb(0xA1, 2);  outportb(0xA1, 1);  **int i;**  **for (i = 0; i < 16; i++) {**  **set\_int\_handler(irq\_base + i, irq\_handlers[i], 0x8E);**  **}**  asm("sti");  }  void set\_int\_handler(uint8 index, void \*handler, uint8 type) {  size\_t saved\_flags;  asm("pushf \n popl %0 \n cli":"=a"(saved\_flags));  idt[index].selector = 8;  idt[index].address\_0\_15 = (size\_t)handler & 0xFFFF;  idt[index].address\_16\_31 = (size\_t)handler >> 16;  idt[index].type = type;  idt[index].reserved = 0;  asm("pushl %0 \n popf"::"a"(saved\_flags));  }  **void irq\_handler(uint32 index, Registers \*regs) {**  **switch (index) {**  **case 0:**  **switch\_task(regs);**  **break;**  **case 1:**  **keyboard\_interrupt();**  **break;**  **}**  **}**  Теперь обработчик в interrupts.c решает кому обрабатывать какие прерывания. В нём, например, задано, что IRQ это должно уйти системе многозадачности, а IRQ1 в обработчик драйвера клавиатуры. В будущем лишь IRQ0 будет иметь специальную обработку, а все остальные прерывания будут преобразовываться в сообщения пользовательским программам-драйверам.  После interrupts.c меняется и interrupts.h:  #ifndef INTERRUPTS\_H  #define INTERRUPTS\_H  **#include "stdlib.h"**  **typedef struct {**  **uint32 gs, fs, es, ds;**  **uint32 ebp, edi, esi, edx, ecx, ebx, eax;**  **uint32 eip, cs;**  **uint32 eflags;**  **uint32 esp, ss;**  **} Registers;**  uint8 irq\_base;  uint8 irq\_count;  void init\_interrupts();  void set\_int\_handler(uint8 index, void \*handler, uint8 type);  #endif  Тут ничего особенного, лишь добавилось описание нового типа - структуры Registers и исчез ненужный макрос IRQ\_HANDLER.  Драйвер клавиатуры также нуждается в доработке. Во-первых, необходимо убрать вызов set\_int\_handler из init\_tty, во-вторых, заменить обработчик IRQ timer\_int\_handler на следующий код:  void keyboard\_interrupt() {  uint8 key\_code;  inportb(0x60, key\_code);  if (key\_buffer\_tail >= KEY\_BUFFER\_SIZE) {  key\_buffer\_tail = 0;  }  key\_buffer\_tail++;  key\_buffer[key\_buffer\_tail - 1] = key\_code;  uint8 status;  inportb(0x61, status);  status |= 1;  outportb(0x61, status);  }  Как можно заметить, обработчик стал обычной функций, что упростит код ядра и улучшит его читабельность.  Ну и наконец самое интересное. Наконец-то, у нас заработает многозадачность как надо. При IRQ0 вызывается функция switch\_task, которой передаётся указатель на структуру regs. Функции остаётся лишь заменить эту структуру на новую и обработчик прерывания выйдет в другую задачу.  Подправим multitasking.h:  #ifndef MULTITASKING\_H  #define MULTITASKING\_H  #include "stdlib.h"  **#include "interrupts.h"**  typedef struct {  ListItem list\_item;  AddressSpace address\_space;  bool suspend;  size\_t thread\_count;  char name[256];  } Process;  typedef struct {  ListItem list\_item;  Process \*process;  bool suspend;  void \*stack\_base;  size\_t stack\_size;  **Registers state;**  } Thread;  List process\_list;  List thread\_list;  Process \*current\_process;  Thread \*current\_thread;  Process \*kernel\_process;  Thread \*kernel\_thread;  void init\_multitasking();  **void switch\_task(Registers \*regs);**  Thread \*create\_thread(Process \*process, void \*entry\_point, size\_t stack\_size, bool kernel, bool suspend);  #endif  Изменился описатель нити - указатель стека нам больше не нужно хранить, зато нужно хранить структуру с регистрами.  Также изменился заголовок функции switch\_task (я помню, что мы хотели использовать её немного для другого, но теперь будем так, а для старого действия позднее создадим новую функцию с более подходящим названием).  А вот и новая реализация переключения задач:  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  #include "multitasking.h"  **bool multitasking\_enabled = false;**  void init\_multitasking() {  list\_init(&process\_list);  list\_init(&thread\_list);  kernel\_process = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  **init\_address\_space(&kernel\_process->address\_space, kernel\_page\_dir);**  kernel\_process->suspend = false;  kernel\_process->thread\_count = 1;  strncpy(kernel\_process->name, "Kernel", sizeof(kernel\_process->name));  list\_append((List\*)&process\_list, (ListItem\*)kernel\_process);  kernel\_thread = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  kernel\_thread->process = kernel\_process;  kernel\_thread->suspend = false;  kernel\_thread->stack\_size = PAGE\_SIZE;  list\_append((List\*)&thread\_list, (ListItem\*)kernel\_thread);  current\_process = kernel\_process;  current\_thread = kernel\_thread;  **multitasking\_enabled = true;**  }  **void switch\_task(Registers \*regs) {**  **if (multitasking\_enabled) {**  **memcpy(&current\_thread->state, regs, sizeof(Registers));**  **do {**  **current\_thread = (Thread\*)current\_thread->list\_item.next;**  **current\_process = current\_thread->process;**  **} while (current\_thread->suspend || current\_process->suspend);**  **asm("movl %0, %%cr3"::"a"(current\_process->address\_space.page\_dir));**  **memcpy(regs, &current\_thread->state, sizeof(Registers));**  **}**  **}**  Переключение задач стало проще. Также, наконец то адресное пространство ядра в описателе ядерного процесса настраивается до конца. Появился вызов новой функции - init\_address\_space. Её следует разместить в конце файла memory\_manager.c:  void init\_address\_space(AddressSpace \*address\_space, phyaddr page\_dir) {  address\_space->page\_dir = page\_dir;  address\_space->start = USER\_MEMORY\_START;  address\_space->end = USER\_MEMORY\_END;  address\_space->block\_table\_size = PAGE\_SIZE / sizeof(VirtMemoryBlock);  address\_space->blocks = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);  address\_space->block\_count = 0;  }  Ну вот и осталось только написать новую, работающую функцию create\_thread:  Thread \*create\_thread(Process \*process, void \*entry\_point, size\_t stack\_size, bool kernel, bool suspend) {  Thread \*thread = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  thread->process = process;  thread->suspend = suspend;  thread->stack\_size = stack\_size;  thread->stack\_base = alloc\_virt\_pages(&process->address\_space, NULL, -1, (stack\_size + PAGE\_SIZE - 1) & ~PAGE\_OFFSET\_MASK, PAGE\_PRESENT | PAGE\_WRITABLE |  (kernel ? 0 : PAGE\_USER));  memset(&thread->state, 0, sizeof(Registers));  uint32 data\_selector = (kernel ? 16 : 27);  uint32 code\_selector = (kernel ? 8 : 35);  thread->state.eflags = 0x202;  thread->state.cs = code\_selector;  thread->state.eip = (uint32)entry\_point;  thread->state.ss = data\_selector;  thread->state.esp = (uint32)thread->stack\_base + thread->stack\_size;  thread->state.ds = data\_selector;  thread->state.es = data\_selector;  thread->state.fs = data\_selector;  thread->state.gs = data\_selector;  list\_append((List\*)&thread\_list, (ListItem\*)thread);  process->thread\_count++;  return thread;  }  Ну вот, многозадачность почти готова. Если сейчас попытаться создать новую нить с помощью create\_thread система упадёт. Это вызвано тем, что сейчас при прерывании в стек не записывается SS и ESP (и это если прерывание произошло в привилегированном коде, если нет, то система опять же упадёт). Чтобы всё происходило так, как нужно, следует создать структуру TSS.  Изначально TSS задумывался как структура для аппаратной поддержки переключения задач. Грубо говоря, это аппаратный аналог нашей структуры Registers с некоторыми дополнительными полями. Почему же мы сразу не воспользовались им? Потому что он не очень удобен (подразумевается для каждой задачи создавать отдельных сегмент TSS) и существует лишь на архитектуре Intel. Из-за этого он не используется по прямому назначению ни в одной крупной современной ОС, а в 64-битном режиме был урезан, сохранив с себе только указатели привилегированных стеков и карту ввода-вывода (как раз эти поля этой структуры активно используются), утратив функцию хранения контекстов задач.  Опишем эту структуру в multitasking.h:  #ifndef INTERRUPTS\_H  #define INTERRUPTS\_H  #include "stdlib.h"  typedef struct {  uint32 gs, fs, es, ds;  uint32 ebp, edi, esi, edx, ecx, ebx, eax;  uint32 eip, cs;  uint32 eflags;  uint32 esp, ss;  } Registers;  **typedef struct {**  **uint32 reserved\_1;**  **uint32 esp0;**  **uint32 ss0;**  **uint32 esp1;**  **uint32 ss1;**  **uint32 esp2;**  **uint32 ss2;**  **uint32 cr3;**  **uint32 eip;**  **uint32 eflags;**  **uint32 eax;**  **uint32 ecx;**  **uint32 edx;**  **uint32 ebx;**  **uint32 esp;**  **uint32 ebp;**  **uint32 esi;**  **uint32 edi;**  **uint32 es;**  **uint32 cs;**  **uint32 ss;**  **uint32 ds;**  **uint32 fs;**  **uint32 gs;**  **uint32 ldtr;**  **uint16 reserved\_2;**  **uint16 io\_map\_offset;**  **uint8 io\_map[8192 + 1];**  **} \_\_attribute\_\_((packed)) TSS;**  uint8 irq\_base;  uint8 irq\_count;  void init\_interrupts();  void set\_int\_handler(uint8 index, void \*handler, uint8 type);  #endif  Полей у этой структуры, как видите, очень много, но нас интересует лишь 4 из них - SS0, ESP0, io\_map\_offset и io\_map, остальные мы никогда использовать не будем. io\_map мы рассмотрим несколько позднее, про io\_map\_offset скажу лишь, что он должен быть равен смещению io\_map относительно начала структуры. А вот на SS0 и ESP0 остановимся подробнее, потому что именно они нам сейчас и нужны.  Допустим, процессор выполняет непривилегированный код. Тут происходит прерывание, причём его обработчик находится в сегменте привилегированного кода. Непривилегированный стек использовать небезопасно, поэтому производится переключение на стек ядра (его вершина - SS0:ESP0), уже туда запихиваются старые SS, ESP, EFLAGS и CS с EIP. При выходе из прерывания эти значения будут восстановлены в соответствующие регистры и следовательно указатель стека станет прежним.  Адрес стека ядра как раз и хранится в структуре TSS (там есть и вершины стеков для ring1 и ring2, но мы используем лишь 2 уровня привилегий, поэтому они нам не нужны). К тому же, такое переключение стека при настроенном TSS происходит всегда, а не только, если прерывается непривилегированный код, что упрощает работу обработчика прерывания, если ему интересен стек прерванной задачи.  Наша структура Registers как раз рассчитана на такую ситуацию, но чтобы так было, нужно обязательно создать TSS.  Если SS0 и ESP0 могут быть общими для всех процессов (да так, вообще-то, и будет), то io\_map должны быть разные, поэтому TSS имеет смысл разместить в адресном пространстве приложения, чтобы он переключался при переключении задач. Поэтому, пусть TSS будет располагаться на 3-х последних страницах адресного пространства приложения. Создадим его для процесса ядра:  #include "stdlib.h"  #include "memory\_manager.h"  #include "interrupts.h"  #include "multitasking.h"  **TSS \*tss = (void\*)(USER\_MEMORY\_END - PAGE\_SIZE \* 3 + 1);**  bool multitasking\_enabled = false;  void init\_multitasking() {  list\_init(&process\_list);  list\_init(&thread\_list);  kernel\_process = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  init\_address\_space(&kernel\_process->address\_space, kernel\_page\_dir);  **alloc\_virt\_pages(&kernel\_process->address\_space, tss, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);**  **tss->esp0 = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);**  **tss->ss0 = 16;**  **tss->io\_map\_offset = (uint32)((uint32)tss->io\_map - (uint32)tss);**  kernel\_process->suspend = false;  kernel\_process->thread\_count = 1;  strncpy(kernel\_process->name, "Kernel", sizeof(kernel\_process->name));  list\_append((List\*)&process\_list, (ListItem\*)kernel\_process);  kernel\_thread = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  kernel\_thread->process = kernel\_process;  kernel\_thread->suspend = false;  kernel\_thread->stack\_size = PAGE\_SIZE;  list\_append((List\*)&thread\_list, (ListItem\*)kernel\_thread);  current\_process = kernel\_process;  current\_thread = kernel\_thread;  multitasking\_enabled = true;  }  ...  Осталось лишь указать процессору, где у нас находится TSS. Это делается с помощью сегмента в GDT. Добавим его (напомню, что таблица дескрипторов сегментов у нас описана в startup.asm).  format ELF  public \_start  extrn kernel\_main  section ".text" executable  \_start:  movzx edx, dl  push ebx  push esi  push edx  lgdt [gdtr]  call kernel\_main  add esp, 3 \* 4  @:  ;cli  ;hlt  jmp @b  section ".data" writable  gdt:  dq 0  dq 0x00CF9A000000FFFF  dq 0x00CF92000000FFFF  dq 0x00CFFA000000FFFF  dq 0x00CFF2000000FFFF  **dq 0x7F4089FFD0002FFF**  gdtr:  dw $ - gdt  dd gdt  Таблица объявлена, надо лишь загрузить в регистр TR её селектор. Это будет логично сделать в init\_multitasking.  void init\_multitasking() {  list\_init(&process\_list);  list\_init(&thread\_list);  kernel\_process = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  init\_address\_space(&kernel\_process->address\_space, kernel\_page\_dir);  alloc\_virt\_pages(&kernel\_process->address\_space, tss, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  tss->esp0 = (uint32)alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE | PAGE\_GLOBAL);  tss->ss0 = 16;  tss->io\_map\_offset = (uint32)((uint32)tss->io\_map - (uint32)tss);  kernel\_process->suspend = false;  kernel\_process->thread\_count = 1;  strncpy(kernel\_process->name, "Kernel", sizeof(kernel\_process->name));  list\_append((List\*)&process\_list, (ListItem\*)kernel\_process);  kernel\_thread = alloc\_virt\_pages(&kernel\_address\_space, NULL, -1, 1, PAGE\_PRESENT | PAGE\_WRITABLE);  kernel\_thread->process = kernel\_process;  kernel\_thread->suspend = false;  kernel\_thread->stack\_size = PAGE\_SIZE;  list\_append((List\*)&thread\_list, (ListItem\*)kernel\_thread);  current\_process = kernel\_process;  current\_thread = kernel\_thread;  **asm("ltr %w0"::"a"(40));**  multitasking\_enabled = true;  }  Многозадачность готова. Ну вот на сегодня и хватит... |

Пишем свою операционную систему. Доступ к ListFS из обычной ОС

|  |
| --- |
| Всем доброго времени суток!  Последний выпуск рассылки выходил очень давно. Приношу свои извинения, но у меня, к сожалению, стало слишком мало времени. Да и теперь, спустя такой длительный интервал времени, возникает желание переписать ОС с чистого листа, исправив старые ошибки (и добавив новых, кхе-кхе). Наверное, стоит оформить ОС как OpenSource проект, чтобы всегда был доступен полный исходный код, а все желающие могли вносить исправления (мне неоднократно приходили письма с просьбами выслать полный исходный код ОС).  На этом не всё, у меня для вас есть маленький подарок. А именно - FUSE-драйвер для ListFS. В настоящий момент он позволяет пользователям ОС Linux примонтировать образ ListFS и успешно работать с ним в режиме read-only. В дальнейшем планируется добавить поддержку изменения ФС.  На мой взгляд, ListFS оптимальный вариант для начинающих разработчиков ОС, потому что она очень проста по сравнению с "большими" ФС вроде NTFS и EXT4, с другой стороны она обладает бОльшим функционалом, чем устаревшая FAT. Ну и наконец её можно свободно расширять, добавляя новый функционал, которого не хватает именно вашей ОС. В принципе, в перспективе можно будет добавить атрибуты файлов и журналирование, и тогда ФС будет вполне конкурентноспособна.  Вот репозиторий исходного кода на github - https://github.com/kiv-apple/ListFS  Пока там только заголовочный файл со структурами и утилита форматирования, а также драйвер FUSE. Всё это полностью совместимо с образами, создаваемыми утилитой make\_listfs, опубликованной в данной рассылке (собственно, она сейчас является единственным путём создания не пустого образа) и начальным загрузчиком.  Спасибо за внимание! |