

Заметки о написании операционной системы. От чайника чайникам.

Евсиенко Даниил. 22 февраля 2020.

Введение

Мне стал интересен вопрос ос дева. Я пошел гуглить, многие статьи начинались со слов о том, что уже есть много статей, но нет законченных, поэтому авторы решили создать свою серию. Почти все эти серии были в итоге не закончены))

Это не еще одна серия. Эта просто мои заметки, конспект. Здесь я не буду никого и ничему учить, я описываю вещи, которые помогли бы мне освежить мою память в случае необходимости за минимальное время. Я прикладной программист, который хочет поделиться своим опытом изучения данного вопроса, а также ресурсами, которые мне в этом очень помогли. До погружения в данный вопрос у меня не было опыта программирования и знаний ассемблера, си и практически всего того о чем ниже пойдет речь.

Эти строки я пишу имея перед собой нечто рабочее, что загружается, переходит в защищенный режим, обрабатывает необходимый минимум прерываний даже! запускает несколько кусков user mode кода в режиме вытесняющей многозадачности. И вишенкой на этом торге праздника жизни - один системный вызов.

Время от времени я буду сыпать терминами, которые вы либо уже знаете и для вас это само собой разумеющееся либо идете гуглить.

Многие технические решения мягко говоря далеки от идеала, продиктованы ленью. Из-за нее же большая часть кода и текста была скопирована из других ресурсов. И хвала небесам, что в них было много ошибок или они не собирались бесшовно. Именно этим моментам я благодарен за знания, которые остались у меня в голове. Несколько раз я заходил в казалось бы безнадежные тупики, приходилось гуглить, читать, думать наконец!

Основные источники моего копипаста в порядке уменьшения объема заимствований:

<https://subscribe.ru/catalog/comp.soft.myosdev>

<http://www.brokenthorn.com/Resources/OSDevIndex.html>

http://www.jamesmolloy.co.uk/tutorial_html/1.-Environment%20setup.html

Ресурсы, которые мне хорошо помогли:

<https://www.ics.uci.edu/~aburtsev/143A/2017fall/>

https://wiki.osdev.org/Main_Page
<https://forum.osdev.org/>
http://skelix.net/skelixos/index_en.html
<https://frolov-lib.ru/books/bsp.old/v01b/>
<https://manybutfinite.com/post/how-the-kernel-manages-your-memory/>
<https://arjunsreedharan.org/post/82710718100/kernel-101-lets-write-a-kernel>
<https://dev64.wordpress.com/>
<https://bsodtutorials.wordpress.com/2013/12/14/virtual-to-physical-address-translation-part-1/>
https://codemachine.com/articles/prototype_ptes.html
<http://www.independent-software.com/operating-system-development.html>
https://pdos.csail.mit.edu/6.828/2005/readings/i386/s07_05.htm
<https://systo.ru/prog/theor/task-x86.html>
http://skelix.net/skelixos/tutorial06_en.html

и еще некоторые другие.

Можете ознакомиться с тем, что у меня получилось.

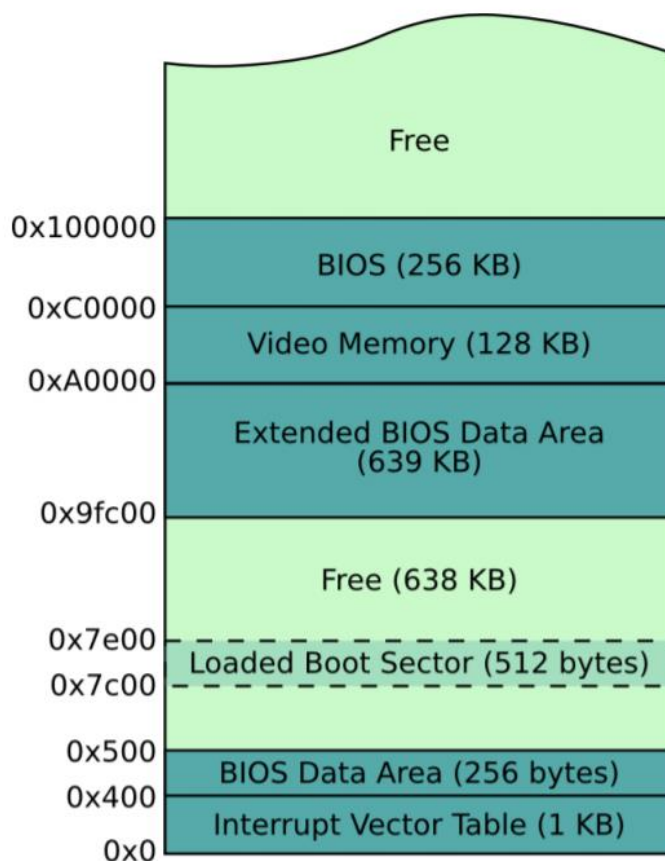


Начальный загрузчик

Начать можно было бы с определения того, что такое операционная система, для чего она нужна, истории возникновения понятия, но боюсь не выдержать конкуренции со стороны сборников сочинений Эндрю Таненбаума и Википедии.

Поэтому мы начнем с включения компьютера. Вы нажимаете кнопку питания и практически сразу после этого управление получает BIOS - Basic Input-Output System. BIOS запускает процесс первичного тестирования оборудования - POST (Power On Self Test), который проверяет что подается достаточное количество энергии, устройства установлены (например, клавиатура, мышь, USB, последовательные порты и т. д.) и проверяет, хорошо ли работает память (путем тестирования на обрыв памяти). Затем POST возвращает контроль BIOS. POST загружает BIOS в конец памяти. Сейчас процессор находится в реальном режиме, поэтому может адресовать всего 1Mb памяти (надеюсь вы знакомы с режимами работы процессора).

Память при этом будет выглядеть как показано на рисунке:



Далее BIOS выполняет еще кое какие активности и загружает 0-ой сектор загрузочного диска (512 байт) в оперативную память, передавая ему управление (загрузочный диск определяется в настройках BIOS'a).

Этот 0-ой сектор загрузочного диска в простонародии называется загрузчиком ОС. Его основной задачей является подгрузка остальных частей операционной системы в память. У многих операционных систем полноценный загрузчик не умещается в один сектор и тогда используется многоэтапная загрузка - первичный загрузчик загружает вторичный загрузчик, который уже загружает всё остальное.

Наш код оказывается в реальном режиме работы процессора по адресу 0000:7C00 (так сложилось исторически, можете поискать в Интернете почему). 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.

Последние два байта начального загрузчика должны содержать сигнатуру 0x55,0xAA, иначе многие BIOS посчитают такой загрузчик некорректным и откажутся запускать.

Попробуем написать и запустить немного кода. Я буду делать это в Windows 10, которая установлена на моем домашнем компьютере.

Нам нужно установить виртуальную машину Bochs и ассемблер fasm (<http://bochs.sourceforge.net/getcurrent.html>) и ассемблер fasm (<https://flatassembler.net/download.php>).

Создаем следующие 3 файла:

Файл boot.asm:

```
org      0x7c00          ; We are loaded by BIOS at 0x7C00

start:

    cli                ; Clear all Interrupts
    hlt                ; halt the system

times 510 - ($-$$) db 0    ; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55                ; Boot Signiture
```




Файл bochs.bxrc:

```
megs: 64
boot: floppy
floppya: 1_44=boot.bios.bin, status=inserted
```





Файл **build.bat**:

fasm boot.asm boot.bios.bin

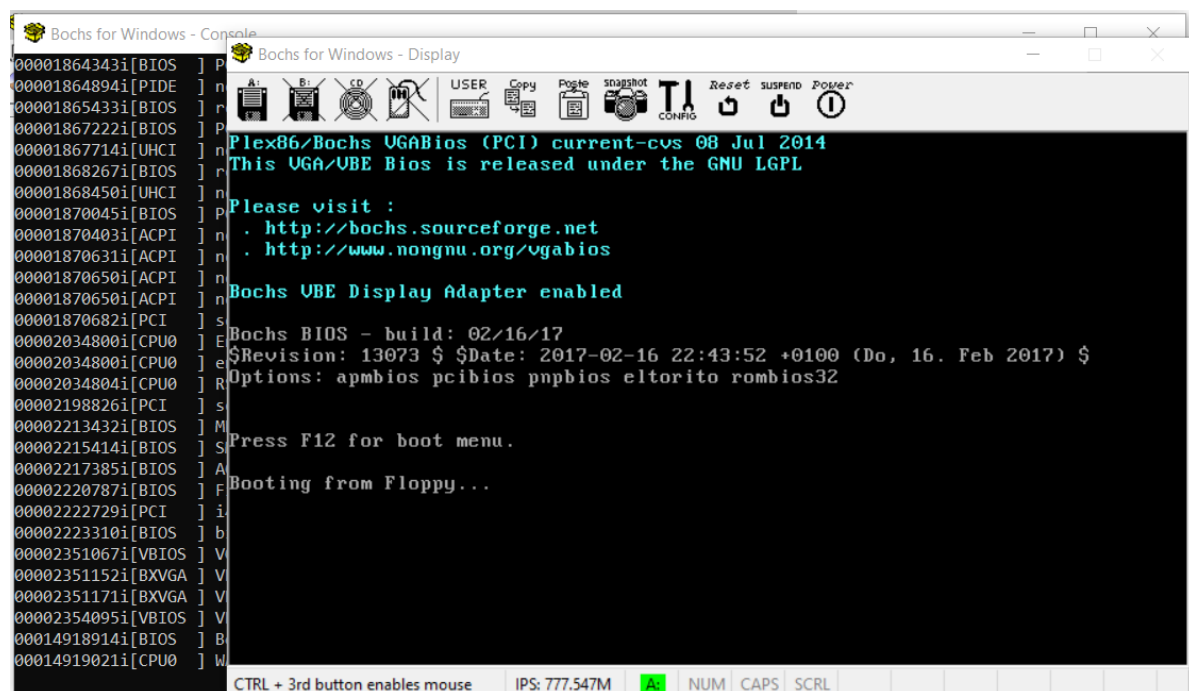
У нас получается такая картина:

Name	Date modified	Type	Size
 bochs.bxrc	4/10/2020 4:52 PM	Bochs 2.6.9 Config...	1 KB
 boot.asm	4/10/2020 5:12 PM	Assembler Source	1 KB
 build.bat	4/10/2020 4:51 PM	Windows Batch File	1 KB

Запускаем *build.bat* и появляется 4-й файл, наш “загрузчик”:

Name	Date modified	Type	Size
 bochs.bxrc	4/10/2020 4:52 PM	Bochs 2.6.9 Config...	1 KB
 boot.asm	4/10/2020 5:12 PM	Assembler Source	1 KB
 boot.bios.bin	4/10/2020 5:20 PM	BIN File	1 KB
 build.bat	4/10/2020 4:51 PM	Windows Batch File	1 KB

Запускаем файл *bochs.bxrc* и смотрим на результат:

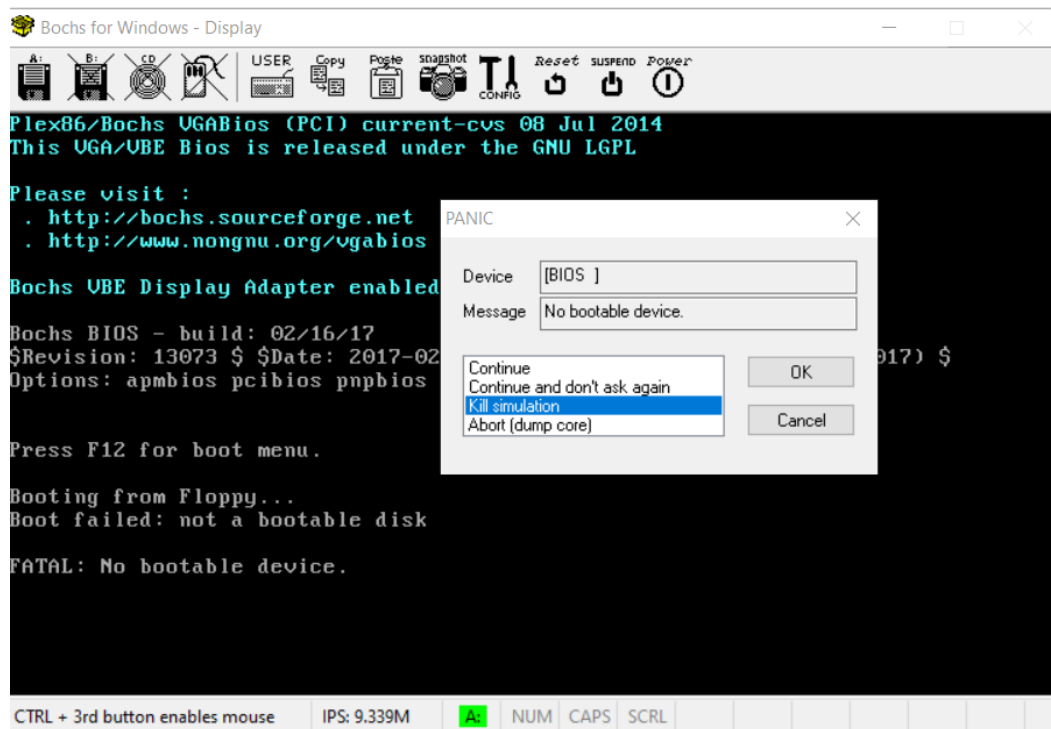


Машина стартует и останавливается как только доходит до инструкции **hlt**, которая останавливает выполнение команд и переводит процессор в режим ожидания. Как уже говорилось, BIOS загружает наш загрузчик по адресу 0x7c00. Все пространство вплоть

до сигнатуры загрузочного сектора заполняем нулями. Это можно увидеть, например, открыв получившийся бинарный файл с помощью утилиты HxD (<https://mh-nexus.de/en/hxd/>).

boot.bios.bin																	
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	FA	F4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	uδ.....
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AAU²

Если например в исходном коде исправить **dw 0xAA55**, скажем, на **dw 0xAA54**, то после перекомпиляции и запуска машины мы получим ошибку:



Исходный

код:

<https://github.com/devsienko/c4os/tree/fb6f696981cc22fc9cd6484391140d0cbc054ddc>.

Немного улучшим наш загрузчик добавив функцию вывод строки на экран.

```
write_str:
    push ax si
    mov ah, 0x0E
@@:
    lodsb
    test al, al
    jz @f
    int 0x10
    jmp @b
@@:
    pop si ax
    ret
```

Функция `write_str` 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, ассемблерной инструкцией `int`. Она принимает в качестве аргумента номер прерывания. Одно прерывание может содержать несколько функций. Номер функции указывается в регистре, также через регистры передаются остальные

параметры. Например для управления экраном служит прерывание с кодом 0x10. Оно содержит ряд функций, например функцию **0x0E** для вывода символа:

```
mov ah, 0x0E ; we choose output symbol function here
int 0x10 ; we choose interrupt
```

Или функцию для установки позиции курсора:

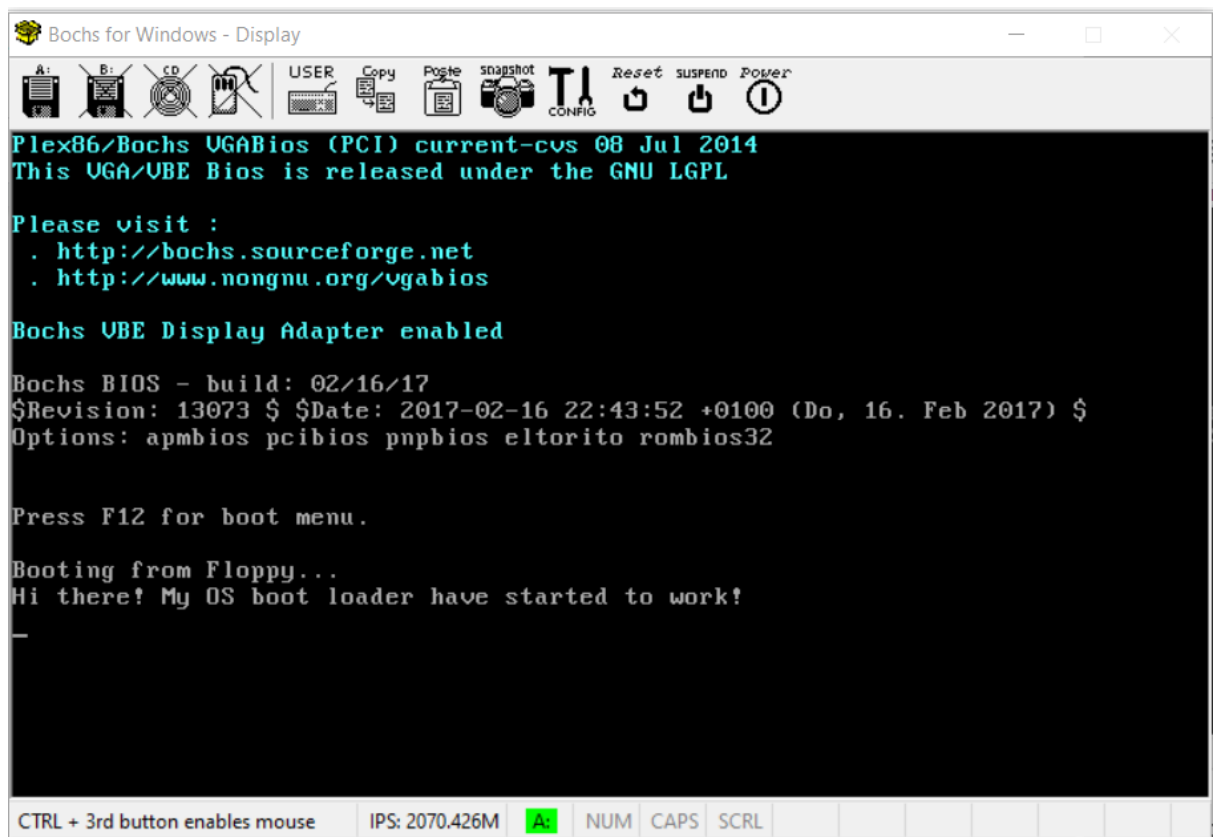
```
mov ah, 0x02 ; we choose set cursor position function
int 0x10 ; we choose interrupt
```

Мы видим, что номер нужной функции прерывания передается в регистре AH. Функция write_str использует функцию с кодом 0x0E, которая просто выводит символ из AL на экран со сдвигом курсора (функция понимает также различные управляющие коды вроде последовательности, например 13,10 - коды для перевода строки и возврата 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.

Также в код была добавлена последовательность символов для вывода на экран и код вызова функции:

```
boot_msg db "Hi there! My OS boot loader have started to work!";13,10,0
;
;
;
;
mov si, boot_msg
call write_str
```

После перекомпиляции и запуска получаем:



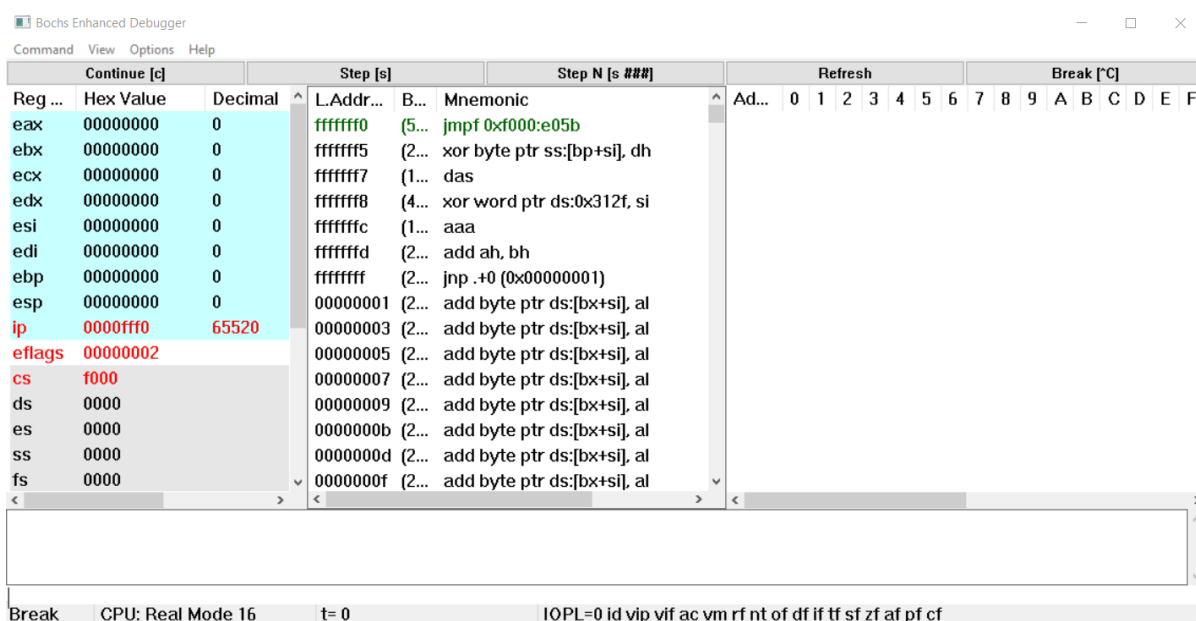
Весь исходный код здесь:
<https://github.com/devsienko/c4os/tree/896aa59ca4fb3d71d623382fba893e4534a4d633>.

Bochs имеет режим отладки. Отлаживать операционную систему можно как из консоли, так и через графическую оболочку. Чтобы воспользоваться графической оболочкой нужно добавить в конфигурационный файл `.bxrc` следующую запись:

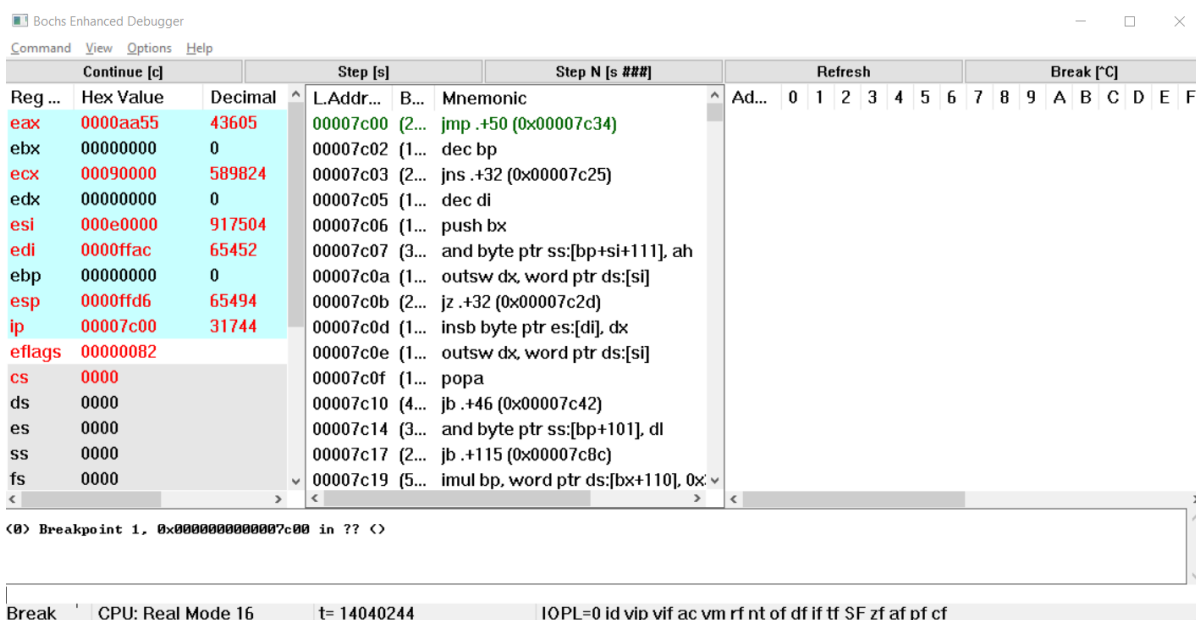
```
display_library: win32, options="gui_debug"
```

а затем запустить конфигурационный файл с помощью утилиты **bochsdbg.exe**, которая находится в той же директории что и **bochs.exe**. Для этой цели я создал в исходном коде файл **start-debug.bat**.

После запуска отладки нашего загрузчика мы увидим что-то вроде следующего окна:



Машина стартовала и остановилась в ожидании команды отладчика, можно нажать кнопку Continue или ввести в поле команд латинскую букву 'c' и тогда работа виртуальной машины будет продолжена в штатном режиме. Можно поставить точку останова, например на первой команде нашего загрузчика, вбив в поле команд **b 0x7c00** и затем продолжить выполнение командой 'c':



Этот отладчик не сильно отличается своей концепцией от тех, которые мы привыкли использовать для прикладных языков программирования. В левой части отображается перечень регистров и их содержимого, в средней части код, в правой части можно выводить регионы памяти.

Загрузка файлов с диска

Начать хотелось бы с подготовки вспомогательного кода. Этот код будет читать данные с диска и записывать в память для дальнейшего выполнения.

Как вы могли заметить по содержимому файлу конфигурации Bochs:

floppya: 1_44=boot.bios.bin, status=inserted

в качестве носителя мы используем floppy-диск, на который записываем образ нашей операционной системы.

Так как у нас нет никаких драйверов и уместить в первые 512 байт мы их не можем на помощь приходят сервисы BIOS. А точнее прерывание 0x13 и несколько его функций:

- 0x00
- 0x02.

Первая функция (**0x00**) служит для сброса дисковой подсистемы, то есть на каком месте бы сейчас Floppy-контроллер не осуществлял чтение, после вызова данной функции он немедленно возвращается на первый сектор.

INT 0x13/AH=0x0 - сброс дисковой подсистемы

AH = 0x0

DL = номер диска, который необходимо сбросить

Возвращает:

AH = Status Code

CF (Carry Flag) пустой в случае успеха, не пустой при ошибке.

Пример вызова этого прерывания:

```
.Reset:
    mov     ah, 0      ; reset floppy disk function
    mov     dl, 0      ; drive 0 is floppy drive
    int     0x13       ; call BIOS
    jc      .Reset     ; If Carry Flag (CF) is set, there was an error. Try resetting again
```

Перед чтением с диска нам необходимо вызвать эту функцию, чтобы гарантированно читать начиная с первого сектора.

Вторая функция (**0x02**) служит для чтения сектора диска и дальнейшей его записи в память.

INT 0x13/AH=0x02 - прочитать сектор(ы) в память

AH = 0x02

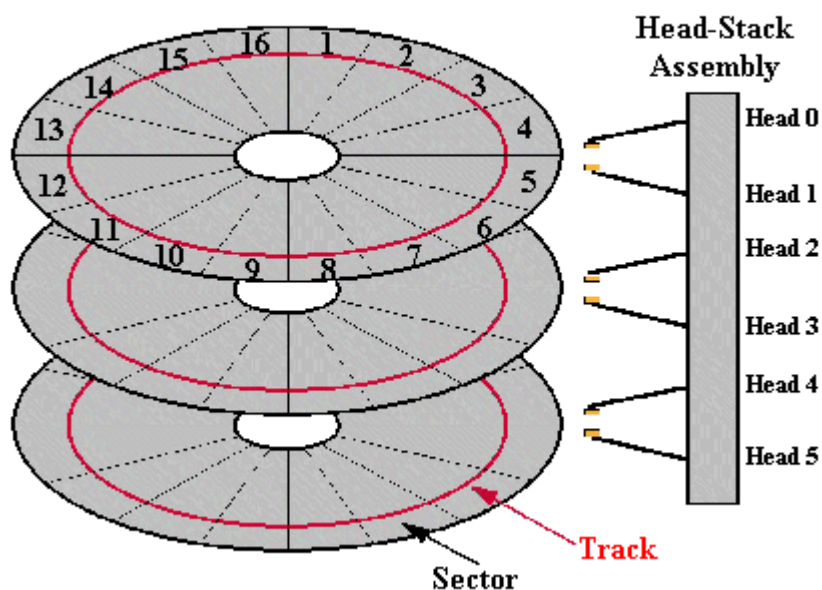
AL = Number of sectors to read
CH = Low eight bits of cylinder number
CL = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only
DH = Head Number
DL = Drive Number (Bit 7 set for hard disks)
ES:BX = Buffer to read sectors to

Возвращает:

AH = Status Code
AL = Number of sectors read
CF = set if failure, cleared is successfull

Здесь могут потребоваться дополнительные объяснения.

Drive Physical and Logical Organization



Рассмотрим классический жесткий магнитный диск. Он состоит из нескольких пластин. У пластины есть две стороны и данные записываются/считываются с каждой стороны специальной **Головкой (Head)**. Каждая сторона разбита на множество Дорожек (Track). Дорожки равноудалены от центра пластины. Множество дорожек с одинаковым радиусом на разных пластинах и их сторонах называются **Цилиндром (Cylinder)**. На рисунке выше все дорожки одного цилиндра выделены красным цветом. Каждая дорожка разбита на **Сектора (Sectors)**. Обычно на Floppy-диске 18 секторов и каждый сектор равен 512 байтам. Рассмотрим пример кода для чтения сектора диска и записи его в память.

```
mov     ax, 0x1000    ; we are going to read sector to into address 0x1000:0
mov     es, ax
xor     bx, bx
```

```

.Read:
    mov     ah, 0x02      ; function 2
    mov     al, 1         ; read 1 sector
    mov     ch, 1         ; we are reading the second sector past us,
                          ; so it's still on track 1
    mov     cl, 2         ; sector to read (The second sector)
    mov     dh, 0         ; head number
    mov     dl, 0         ; drive number. Remember Drive 0 is floppy drive.
    int     0x13          ; call BIOS - Read the sector
    jc      .Read         ; Error, so try again

    jmp     0x1000:0x0     ; jump to execute the sector!

```

Далее. Компоненты нашей операционной системы (загрузчики, ядро, конфигурационные файлы и т. д.) будем хранить в отдельных файлах. При записи в образ эти файлы надо каким-то образом разместить, чтобы загрузчик мог их идентифицировать и считывать. Для организации расположения файлов можно прибегнуть к помощи какой-либо файловой системы (NTFS, FAT, ext2 и т.д.). Некоторые из этих систем слишком сложны, некоторые ограничены, к тому же найти и использовать готовые инструменты для компоновки файлов в соответствии с их форматом не всегда тривиальная задача. В этой рассылке (<https://subscribe.ru/catalog/comp.soft.myosdev?pos=3>) я увидел альтернативное решение - ListFS, разработанная автором рассылки простая и гибкая файловая система с исходными кодами компоновщика.

Чтобы компилировать исходный код компоновщика (make_listfs.c) я установил пакет minGW (<http://www.mingw.org/>) и создал скрипт **compile-make-listfs.bat**.

Исходный код: <https://github.com/devsienko/c4os/tree/f36a4519bdef68a0a2093be6ef2e9ee3e354321a>.

Файловая система построена на концепции двунаправленных связанных списков. Рассмотрим её структуру.

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. 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 -1
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 файла. 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

Добавим соответствующие переменные к кода нашего загрузчика:

```
org 0x7C00
```

```
jmp start
```

```
; ListFS Header
```

```
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 ?
; ListFs file header
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
;...

```

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 создает 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.

Исходный

код:

<https://github.com/devsienko/c4os/tree/3b65ddeda656f0c4d860788ae0fb2eb987c068c5>.

CHS -> LBA и обратно

Ввиду исторических обстоятельств, сервисы BIOS (как мы рассмотрели выше) требуют указания номера сектора в неудобной форме ЦИЛИНДР \ ГОЛОВКА \ СЕКТОР, но так как мы будем работать с линейными номерами секторов (LBA, Logical block addressing), нам необходимо написать код преобразования адресации CHS в адресацию LBA.

LBA представляет сектора, последовательно пронумерованные начиная с нулевого индекса, второй сектор имеет индекс 1 и так далее.

$$LBA = (C \times HPC + H) \times SPT + (S - 1)$$

где

C, H и S - номера цилиндра, головки и сектора

LBA - адрес логического блока

HPC - это максимальное количество головок на цилиндр (сообщается дисководом)

SPT - это максимальное количество секторов на дорожку (сообщается дисководом).

Обратное преобразование:

sector = **(LBA % sectors per track) + 1**

head = **(LBA / sectors per track) % number of heads**

track = **LBA / (sectors per track * number of heads)**

При последовательном считывании данных с накопителя в режиме CHS процесс чтения начинается с цилиндра 0, головки 0 и сектора 1 (который является первым сектором на данном диске), после чего считываются все остальные секторы первой дорожки. Затем выбирается следующая головка и читаются все секторы, находящиеся на этой дорожке. Это продолжается до тех пор, пока не будут считаны данные со всех головок первого цилиндра. Затем выбирается следующий цилиндр, и процесс чтения продолжается в такой же последовательности.

Расположение первого сектора накопителя определяется выражением “цилиндр 0, головка 0, сектор 1 (0,0,1)”; адресом второго сектора является 0,0,2; третьего — 0,1,1; четвертого — 0,1,2 и т.д., пока мы не дойдем до последнего сектора, адрес которого 1,1,2.

При последовательном считывании данных с накопителя в режиме LBA процесс чтения начинается с сектора 0, после чего читается сектор 1, сектор 2 и т.д. В режиме CHS первым сектором жесткого диска является 0,0,1. В режиме LBA этот же сектор будет сектором 0.

Рисунок 13.

Режим	Соответствующие номера секторов							
CHS:	0,0,1	0,0,2	0,1,1	0,1,2	1,0,1	1,0,2	1,1,1	1,1,2
LBA:	0	1	2	3	4	5	6	7

Теперь перейдем к реализации. Добавим в начало нашего загрузчика следующие переменные:

Листинг 9.

```
; ...
```

```
end virtual
```

```
label sector_per_track word at $$ ; number of sectors per track
```

```
label head_count byte at $$ + 2 ; number disk heads
```

```
label disk_id byte at $$ + 3  
; ...
```

Далее с помощью функции 0x08 прерывания 0x13 (к услугам которого мы прибегали чуть выше) определяем параметры нашего диска. Рекомендуется обнулить ES:DI, для обхода багов некоторых BIOS. После вызова данной функции 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 устанавливается флаг переноса.

```
get_disk_parameters:  
    mov [disk_id], dl ; get bootable disk id  
    mov ah, 0x08  
    xor di, di ; ES is already zero  
    push es ; we don't need a pointer to BIOS parameters block but we need zero ES  
    int 0x13  
    pop es  
    jc load_sector.error ; this function is not implemented for now  
    inc dh ; first heads number is 0, to get the length we need increment this one  
    mov [head_count], dh  
    and cx, 111111b ; select low 6 bits of CX  
    mov [sector_per_track], cx
```

Исходный

код:

<https://github.com/devsienko/c4os/tree/dd0e5dcfe2e4532123ae07ad58aaf5330597c4ac>.

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.

Небольшая справка по инструкции DIV, вы можете прибегать к этой справке в случае сложностей с пониманием следующего кода. Инструкция DIV в Ассемблере выполняет деление без знака. Синтаксис команды DIV такой:

DIV ЧИСЛО

ЧИСЛОМ может быть один из следующих:

- Область памяти (MEM)
- Регистр общего назначения (REG)

Эта команда не работает с сегментными регистрами, а также не работает непосредственно с числами. То есть вот так

DIV 200 ; неправильно

делать нельзя.

А теперь алгоритм работы команды DIV:

Если ЧИСЛО - это БАЙТ, то **AL** = **AX** / ЧИСЛО

Если ЧИСЛО - это СЛОВО, то **AX** = (**DX AX**) / ЧИСЛО

остаток от деления, если таковой имеется, будет записан:

- В регистр АН, если ЧИСЛО - это байт
- В регистр DX, если ЧИСЛО - это слово

Также хотелось бы напомнить, то AX делится на АН и AL, DX на ДН и DL.

; load sector from DX:AX to buffer from ES:DI

load_sector:

push ax bx cx dx si

div [sector_per_track] ; divide DX:AX by sector per track numbers (quotient will be written into AX, remainder of division into DX)

mov cl, dl ; remainder of division is **sector index**, we put it into CL

inc cl ; because first sector index is 1

div [head_count] ; we divide quotient (from AX) by disk head numbers (in this case we divide by byte instead of word)

mov dh, ah ; remainder of division (it's in AH) is **head number**, we put it into DH

mov ch, al ; quotient (it's in AL) is **track number (aka cylinder number)** - we put it into CH

mov dl, [disk_id] ; put disk id into DL

mov bx, di ; put the buffer offset into BX. It's segment offset is already in ES

mov al, 1 ; number of sectors for the reading - 1.

mov si, 3 ; number of writing attempts

@@: ; reading loop

mov ah, 2 ; routine number

int 0x13 ; try to read

jnc @f ; in success case go out from function

xor ah, ah ; reset routine number (0x00)

int 0x13 ; reset disk system

dec si ; decrement attempts count

jnz @b ; if we have not spent attempts then go to begin of the loop

.error: ; reading attempts are exceeded

call error

db "DISK ERROR",13,10,0

@ @:

```
pop si dx cx bx ax
ret
```

Подпрограмма для чтения секторов диска готова. 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 это не точно. 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 немного, разница заметна не будет).

Исходный код:
<https://github.com/devsienko/c4os/tree/78b50c2109785c298e3ca5ac811c3cf6ee4168d7>.

Загрузка продолжения загрузчика

Продолжим обсуждение подгрузки частей нашей операционной системы. Работа с каждым файлом будет происходить в два этапа - поиск файла и его загрузка.

Начнём с подпрограммы для поиска файла в каталоге (**find_file**). 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, вывести сообщение "NOT FOUND" и система переходит в ожидание перезагрузки (подпрограмма error), потому что на этом этапе любая ошибка является критической.

Небольшая справка. Сперва по команде **repne scasb**. Оператор **scas** (scan string) позволяет нам работать со строками. У него есть несколько разновидностей предназначенных для различных размеров данных. В нашем случае это **scasb**. **scasb** служит для поиска первого попавшегося байта. Адрес строки для поиска нужно записать в **ES:DI**, ее длину заносим в регистр **CX**. В **AL** заносим символ, который нам нужно найти в строке. После сравнения автоматически производится декремент (при **DF (Direction Flag) = 1**) или инкремент (при **DF = 0**) содержимого регистра **DI**. Если команда оперирует байтами, индексный регистр изменяется на 1, если словами - на 2. Теперь пример.

```
str_test db 1,2,3,4,5,6,7,8,9,0
str_test_length equ $-str_test
```

```

mov di, str_test
mov cx, str_test_length
mov al 4
repne scasb

```

После выполнения команды **repne scasb** регистр **DI** будет указывать на адрес байта, следующего после найденного символа.

Команда **scasb** без префикса выполняется один раз. Размещение префикса **repne** перед командой **scasb** заставляет ее выполняться в цикле. Префикс **repne** (repeat if not equal) позволяет сканировать строку до тех пор, пока символ расположенный в **AL**, не будет в этой строке найден.

При наличии префикса после каждого выполнения команды производится автоматическое уменьшение регистра **CX** на 1, поэтому его необходимо инициализировать на требуемое число повторений. Команда **repne** перестанет повторяться, если **ZF=1**.

Команда **cmpsb** сравнивает один байт из памяти по адресу **DS:SI** с байтом по адресу **ES:DI**. После выполнения команды, регистры **SI** и **DI** увеличиваются на 1, если флаг **DF = 0**, или уменьшаются на 1, если **DF = 1**.

С префиксом **repe** команда используется для поиска отличающихся элементов строк. При наличии префикса после каждого выполнения команды производится автоматическое уменьшение регистра **CX** на 1, поэтому его необходимо инициализировать на требуемое число повторений. **repe** повторяет сравнение до тех пор, пока сравниваемые элементы равны (флаг **ZF** равен 1), или регистр **CX** не равен нулю.

```

; find a file by name DS:SI in directory DX:AX
find_file:
    push cx dx di
    .find:
        cmp ax, -1 ; is it the end of the list?
        jne @f
        cmp dx, -1
        jne @f
    .not_found: ;if it's the end of the list and we didn't find the file
        call error
        db "NOT FOUND",13,10,0
    @@:
        mov di, f_info
        call load_sector ; load a sector with file info
        push di ; get file name length (DI = f_info = f_name)

```

```

mov cx, 0xFFFF ; we don't know string length, consider max (0xFFFF) length
xor al, al ; 0 is the end of string sign
repne scasb
neg cx ; get length of the string, some binary math magic
dec cx
pop di
push si ; compare file names
repe cmpsb
pop si
je .found ; if the file name are equal then we return
mov ax, word[f_next] ; load next file info sector
mov dx, word[f_next + 2]
jmp .find
.found:
pop di dx cx
ret

```

Исходный

код:

<https://github.com/devsienko/c4os/tree/0a02d566f12861888d2376adabe54ac6e16eb940>.

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.

Небольшая справка. Команда **lodsw** копирует слово из памяти по адресу **DS:SI** в регистр **AX**. После выполнения команды, регистр **SI** увеличивается на 2, если флаг **DF** = 0, или уменьшается на 2, если **DF** = 1.

```

; load current file to memory to BX:0 address. we load number of loaded sectors to AX
load_file_data:
    push bx cx dx si di
    mov ax, word[f_data] ; load to DX:AX index of the first file sectors list
    mov dx, word[f_data + 2]
.load_list:
    cmp ax, -1 ; is it the end of the list?
    jne @f
    cmp dx, -1
    jne @f
.file_end: ; file loading is done
    pop di si dx cx ; restore all registers except BX

```

```

    mov ax, bx ; remember BX
    pop bx ; restore previous BX value
    sub ax, bx ; find difference it will be the file size in 16 byte blocks
    shr ax, 9 - 4 ; translate blocks to sectors
    ret

@@:
    mov di, f_info ; we will load the list to temp buffer
    call load_sector
    mov si, di ; SI := DI
    mov cx, 512 / 8 - 1 ; number of sectors in the list

.load_sector:
    lodsw ; load next sector
    mov dx, [si]
    add si, 6
    cmp ax, -1 ; is it the end of the list?
    jne @f
    cmp dx, -1
    je .file_end ; if it's the end then we return

@@:
    push es
    mov es, bx ; load next sector
    xor di, di
    call load_sector
    add bx, 0x200 / 16 ; load the next sector further 512 bytes
    pop es
    loop .load_sector ; this command decrement CX and if CX is greater than 0 then go to
.load_sector
    lodsw ; load next list index to DX:AX
    mov dx, [si]
    jmp .load_list

```

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).

Исходный

код:

<https://github.com/devsienko/c4os/tree/9487aa1ce4ace0b808e9034a6679c604e87f0a92>.

Теперь продолжим писать код. Реализуем полноценную функцию загрузки файла, которая поддерживает подкаталоги. Располагаться она будет, как и весь остальной код во второй половине загрузчика. Мы 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". Полнофункциональную подпрограмму загрузки файла:

```
; additional bootloader data
load_msg_prefix db "Loading ",0
load_msg_suffix db "...",0
ok_msg db "OK",13,10,0
; split string from DS:SI by '/'
split_file_name:
    push si
@@:
    lodsb
    cmp al, "/"
    je @f
    jmp @b
@@:
    mov byte[si - 1], 0
    mov ax, si
    pop si
    ret
; load file with name DS:SI to the buffer DI:0. file size (in sectors) we return in AX
load_file:
    push si
    mov si, load_msg_prefix
    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] ; start to search with root dir
    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 ; is it dir flag?
    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

```

Теперь добавим немного кода для загрузки продолжения начального загрузчика:

```

; load stage 2 of the bootloader
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
; go to stage 2 of the bootloader
jmp 0x7E00

```

А в области данных опишем:

```

boot_file_name db "boot.bin",0

```

В директория с исходным кодом я добавил утилиту `dd` (<https://aeroquartet.com/movierepair/dd%20for%20windows.en.html>). Она позволяет копировать файлы побайтово. После компиляции нашего исходного кода у нас на выходе мы получаем исполняемый файл размером больше чем 512 байт. Перед тем как компоновать образ диска мы разбиваем наш скомпилированный код следующей парой команд (в файле **build.bat**):

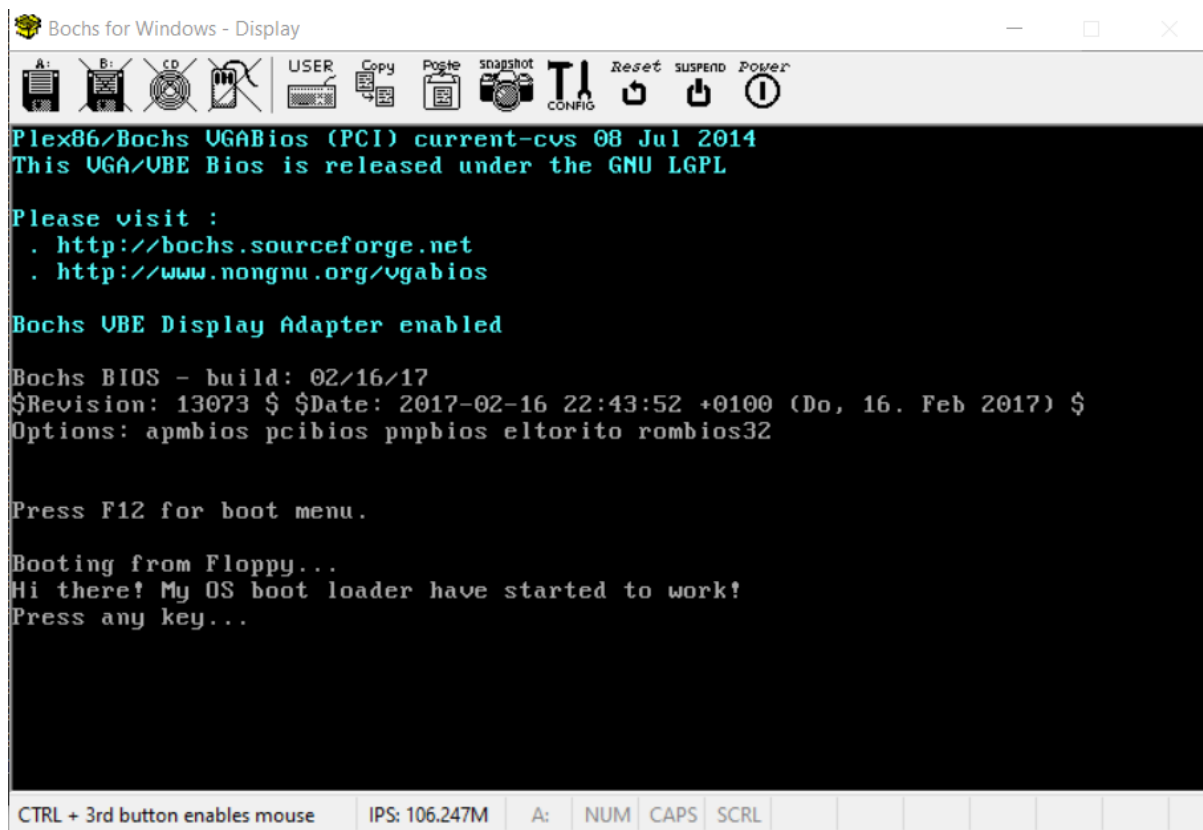
```

dd if=bin\boot.bios.bin of=bin\boot_sector.bin bs=512 count=1
dd if=bin\boot.bios.bin of=bin\disk\boot.bin bs=1 skip=512

make_listfs\make_listfs of=bin\disk.img bs=512 size=2880 boot=bin\boot_sector.bin src=.\bin\disk

```

Теперь можем скомпилировать наш код и запустить его на виртуальной машине:

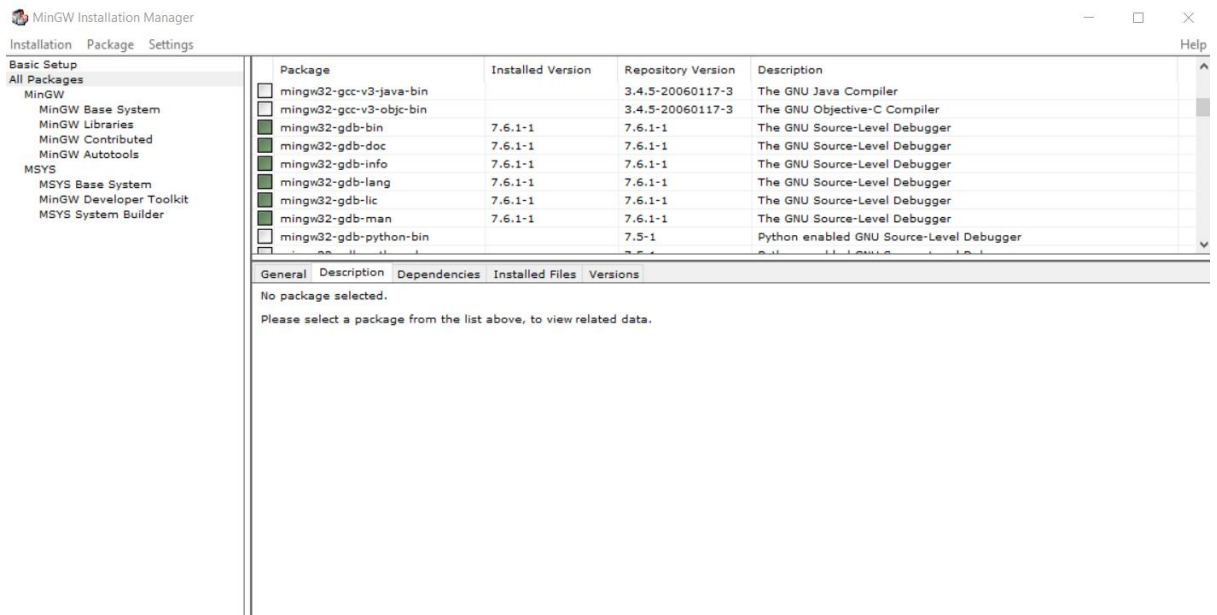


Исходный

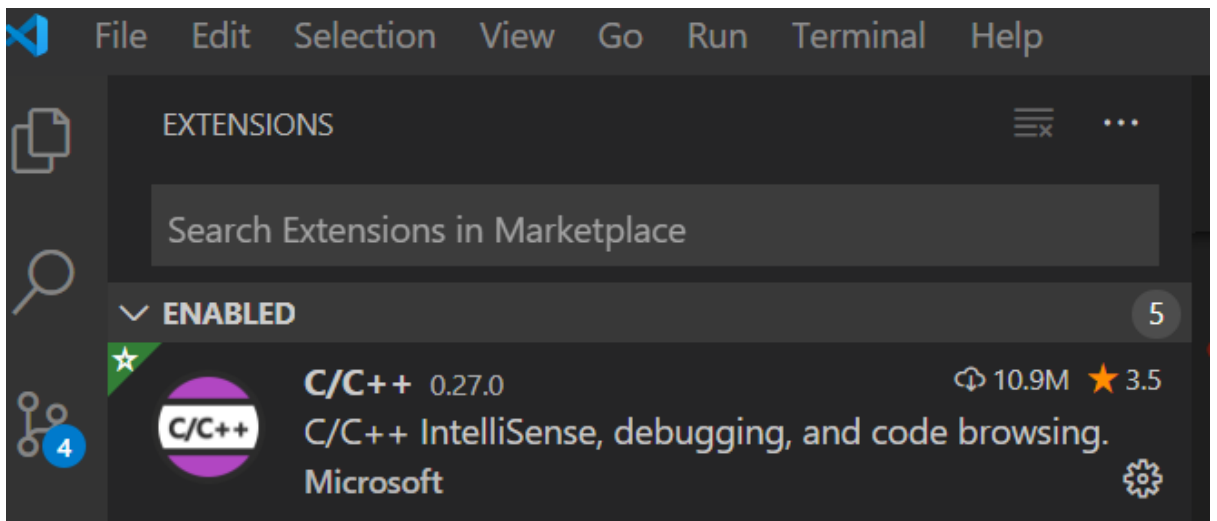
код:

<https://github.com/devsienko/c4os/tree/2ce4c97181d10c45341f474a9d15ac424227af1f>.

Если у вас будет желание вникнуть глубже в работу утилиты make_listsf или может быть в устройство или структуру listfs я могу порекомендовать установить отладчик gdb (это можно сделать через менеджер установки minGw):



Затем установить рекомендованное расширение для VS Code:



После чего нажать F5 (для старта отладки), выбрать **C++ (GDB/LLDB)**, затем выбрать **g++.exe build and debug active file**. После выполнения этих действий в папке .vscode должны появиться два автоматически созданных файла launch.json и tasks.json:



Проследите за тем, чтобы пути до компилятора и отладчика в этих файлах были прописаны корректно! В моем случае были следующие пути:

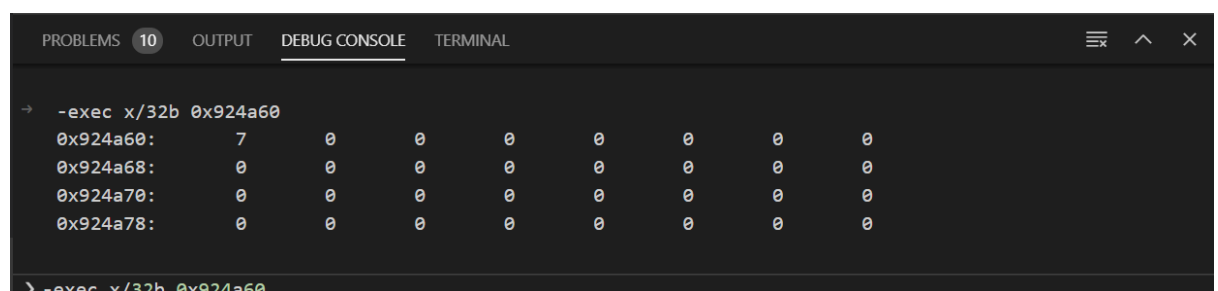
```
"miDebuggerPath": "C:\\mingw\\bin\\gdb.exe"
```

```
"command": "C:\\mingw\\bin\\gcc.exe"
```

Также в файле **launch.json** нужно прописать аргументы командной строки утилиты `make_listfs`:

```
"args": [  
  "of=bin\\disk.img",  
  "bs=512",  
  "size=2880",  
  "boot=bin\\boot_sector.bin",  
  "src=\\.bin\\disk"  
]
```

Отладчик языка достаточно удобный, но на момент написания этих строк дампы памяти можно просматривать только в консоле, для этого нужно на вкладке **DEBUG CONSOLE** ввести команду отладчика, например **-exec x/32b 0x924a60**. Эта команда выведет содержимое 32 байт начиная с адреса 0x924a60 (формат вывода и его размер можно менять, смотрите документацию):



```
→ -exec x/32b 0x924a60  
0x924a60:  7      0      0      0      0      0      0      0  
0x924a68:  0      0      0      0      0      0      0      0  
0x924a70:  0      0      0      0      0      0      0      0  
0x924a78:  0      0      0      0      0      0      0      0  
➤ -exec x/32b 0x924a60
```

Переход в защищенный режим

До этого момента мы находились в реальном режиме работы процессора и сейчас готовы к переходу в защищенный режим. Реальный режим присутствует в процессоре для целей совместимости и имеет ряд ограничений из-за которых работать с ним неудобно.

Защищенный режим по сравнению с ним предоставляет следующие возможности:

- увеличение адресуемого пространства до 4 Гбайт;
- возможность работать в виртуальном адресном пространстве, превышающем максимально возможный объем физической памяти и составляющем огромную величину 64 Тбайт;
- организация многозадачного режима с параллельным выполнением нескольких программ (процессов). Собственно говоря, многозадачный режим организует многозадачная операционная система, однако микропроцессор предоставляет необходимый для этого режима мощный и надежный механизм защиты задач друг от друга с помощью четырехуровневой системы привилегий;
- страничная организация памяти, повышающая уровень защиты задач друг от друга и эффективность их выполнения.

При включении микропроцессора в нем автоматически устанавливается режим реального адреса. Переход в защищенный режим осуществляется программно путем выполнения специальной последовательности команд. Поскольку многие детали функционирования микропроцессора в реальном и защищенном режимах существенно различаются, программы, предназначенные для защищенного режима, должны быть написаны особым образом. При этом различия реального и защищенного режимов настолько велики, что программы реального режима не могут выполняться в защищенном режиме и наоборот. Другими словами, реальный и защищенный режимы несовместимы.

Наиболее важным отличием защищенного режима от реального является иной принцип формирования физического адреса. Вспомним, что в реальном режиме физический адрес адресуемой ячейки памяти состоит из двух компонентов - сегментного адреса и смещения. Оба компонента имеют размер 16 бит, и процессор, обращаясь к памяти, пользуется следующим правилом вычисления физического адреса:

физический адрес = сегментный адрес * 16 + смещение

И сегментный адрес, и смещение не могут быть больше FFFFh, откуда следуют два важнейших ограничения реального режима: объем адресного пространства составляет всего 1 Мбайт, а сегменты не могут иметь размер, превышающий 64 Кбайт.

В защищенном режиме программа по-прежнему состоит из сегментов, адресуемых с помощью сегментных регистров, однако местоположение сегментов в физической памяти определяется другим способом. В **сегментные регистры** в защищенном режиме записываются не сегментные адреса, а так называемые

селекторы, биты **3...15!!!** которых рассматриваются, как **номера (индексы)** ячеек специальной таблицы (**Global Descriptor Table, GDT**), содержащей дескрипторы сегментов памяти.

GDT представляет собой глобальную карту памяти. Она описывает какая память может быть выполнимой (**Code Descriptor**), и какая содержит данные (**Data Descriptor**).

Каждая запись этой таблицы, которая называется **дескриптором (descriptor)**, описывает регион памяти и его свойства (см. ниже). Дескриптор представляет собой 8-байтовое значение следующего формата (потратив около получаса на перевод я осознал, что лучше все ревертнуть и оставить на английском):

- **Bits 56-63:** Bits 24-32 of the base address
- **Bit 55:** Granularity
 - **0:** None
 - **1:** Limit gets multiplied by 4K
- **Bit 54:** Segment type
 - **0:** 16 bit
 - **1:** 32 bit
- **Bit 53:** Reserved-Should be zero
- **Bits 52:** Reserved for OS use
- **Bits 48-51:** Bits 16-19 of the segment limit
- **Bit 47:** Segment is in memory (Used with Virtual Memory)
- **Bits 45-46:** Descriptor Privilege Level
 - **0:** (Ring 0) Highest
 - **3:** (Ring 3) Lowest
- **Bit 44:** Descriptor Bit
 - **0:** System Descriptor
 - **1:** Code or Data Descriptor
- **Bits 41-43:** Descriptor Type
 - **Bit 43:** Executable segment
 - **0:** Data Segment
 - **1:** Code Segment
 - **Bit 42:** Expansion direction (Data segments), conforming (Code Segments)
 - **Bit 41:** Readable and Writable
 - **0:** Read only (Data Segments); Execute only (Code Segments)
 - **1:** Read and write (Data Segments); Read and Execute (Code Segments)
- **Bit 40:** Access bit (Used with Virtual Memory)

- **Bits 16-39:** Bits 0-23 of the Base Address
- **Bits 0-15:** Bits 0-15 of the Segment Limit

Или:

31				16				15				0							
Base 0:15								Limit 0:15											
63		56		55		52		51		48		47		40		39		32	
Base 24:31				Flags				Limit 16:19				Access Byte				Base 16:23			

Как и в реальном режиме, адрес адресуемой ячейки вычисляется процессором, как сумма базового адреса сегмента и смещения:

линейный адрес = базовый адрес сегмента + смещение

В 32-разрядных процессорах смещение имеет размер 32 бит, поэтому максимальная длина сегмента составляет $2^{32} = 4$ Гбайт.

Таблица дескрипторов сегментов

Индексы дескрипторов	0	Первый пустой дескриптор
	1	Адрес = 0 Длина = 1 Мбайт
	2	Адрес = 1Мбайт Длина = 100 Кбайт
	3	Адрес = 8,5 Мбайт Длина = 256 Кбайт

На рисунке выше приведен гипотетический пример программы, состоящей из трех сегментов, первый из которых имеет длину 1 Мбайт и расположен в начале адресного пространства, второй, размером 100 Кбайт, вплотную примыкает к первому, а третий, имеющий размер всего 256 байт, расположен в середине девятого по счету мегабайта.

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 вообще 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 мы будем использовать этот подход. Не следует пытаться использовать сегментацию в своей операционной системе - это сильно усложняет код ядра, языки высокого уровня (например, C или C++) 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 было сказано выше мы создадим таблицу в которой будет один дескриптор кода и один дескриптор данных, оба этих дескриптора будут охватывать память от 0 до 0xFFFFFFFF и разрешать запись, чтение и выполнение любого участка кода:

```
; This is the beginning of the GDT. Because of this, its offset  
is 0.
```

```
; null descriptor
```

```
dd 0 ; null descriptor--just fill 8 bytes with zero  
dd 0
```

```
; Notice that each descriptor is exactly 8 bytes in size. THIS  
IS IMPORTANT.
```

```
; Because of this, the code descriptor has offset 0x8.
```

```
; code descriptor:
```

```
; code descriptor. Right after null descriptor
```

```
dw 0xFFFFh ; limit low  
dw 0 ; base low  
db 0 ; base middle  
db 10011010b ; access  
db 11001111b ; granularity
```

```
db 0 ; base high
```

; Because each descriptor is 8 bytes in size, the Data descriptor is at offset 0x10 from

; the beginning of the GDT, or 16 (decimal) bytes from start.

; data descriptor: ; data descriptor

```
dw 0FFFFh ; limit low (Same as code)
```

```
dw 0 ; base low
```

```
db 0 ; base middle
```

```
db 10010010b ; access
```

```
db 11001111b ; granularity
```

```
db 0 ; base high
```

Наша таблица будет содержать три дескриптора, каждый по 8 байт. Null дескриптор, дескриптор кода и дескриптора данных.

Давайте проанализируем вышеприведенный код. Null дескриптор это просто набор нулей, поэтому сосредоточимся на двух оставшихся.

Разберемся с дескриптором кода

Взглянем еще раз:

; code descriptor:

; code descriptor. Right after null descriptor

```
dw 0FFFFh ; limit low
```

```
dw 0 ; base low
```

```
db 0 ; base middle
```

```
db 10011010b ; access
```

```
db 11001111b ; granularity
```

```
db 0 ; base high
```

Инструкции объявления байта, слова, двойного слова и т. д. идут здесь друг за другом. То есть вышеприведенный код в памяти будет выглядеть как следующая бинарная последовательность (с учетом того, что 0FFFFh это последовательность единиц):

```
11111111 11111111 00000000 00000000 00000000 10011010 11001111 00000000
```

Биты 0-15 (11111111 11111111), а также 48-51 (1100) являются размером сегмента памяти (итого 11111111 11111111 1100, что является 1 048 572 в десятичной системе счисления или 0xFFFFC), который описывает данный дескриптор. Это значит, что мы не можем использовать адрес больше чем 1Мб. Если мы все же попытаемся, то это приведет нас к ошибке GPF (General protection fault). Однако, если бит гранулярности (бит 55) выставлен в 1, то мы должны интерпретировать размер сегмента не в байтах, а в страницах (4096 байта или 0x1000). А это уже 0xFFFFC * 0x1000 = 4Гб.

Под адрес сегмента в дескрипторе выделяется 32 бит (биты 16-39 и биты 56-63), и, таким образом, сегмент может начинаться в любой точке адресного пространства объемом $2^{32} = 4$ Гбайт. Это адресное пространство носит название линейного. В простейшем случае, когда **выключено страничное преобразование**, о котором речь будет идти позже, линейные адреса отвечают физическим. Таким образом, процессор может работать с оперативной памятью объемом до 4 Гбайт. В нашем случае это 0x0. Так как базовый адрес 0x0 и размер сегмента $0xFFFFC * 0x1000 = 4Гб$, следовательно в нашем распоряжении вся память в этом промежутке.

Далее идет байт номер 6. Рассмотрим его побитово:

`db 10011010b ; access`

Опять же переводить это не поднялась рука. Да и лень.

- **Bit 0 (Bit 40 in GDT):** Access bit (Used with Virtual Memory). Because we don't use virtual memory (Yet, anyway), we will ignore it. Hence, it is 0
- **Bit 1 (Bit 41 in GDT):** is the readable/writable bit. Its set (for code selector), so we can read and execute data in the segment (From 0x0 through 0xFFFF) as code
- **Bit 2 (Bit 42 in GDT):** is the "expansion direction" bit. We will look more at this later. For now, ignore it.
- **Bit 3 (Bit 43 in GDT):** tells the processor this is a code or data descriptor. (It is set, so we have a code descriptor)
- **Bit 4 (Bit 44 in GDT):** Represents this as a "system" or "code/data" descriptor. This is a code selector, so the bit is set to 1.
- **Bits 5-6 (Bits 45-46 in GDT):** is the privilege level (i.e., Ring 0 or Ring 3). We are in ring 0, so both bits are 0.
- **Bit 7 (Bit 47 in GDT):** Used to indicate the segment is in memory (Used with virtual memory). Set to zero for now, since we are not using virtual memory yet

Проще говоря, последовательность 10011010b говорит нам о том, что данный дескриптор это дескриптор кода, в наш сегмент, который он описывает, можно писать и читать из него, доступ к нему разрешен только коду режима ядра (Ring 0).

Посмотрим следующий байт:

```
db 11001111b ; granularity
```

Биты 0-3 (они же 48-51 в GDT) мы уже рассмотрели, это часть значения размера сегмента. Бит 7 (он же бит 55 в GDT) также рассмотрели. Остаются:

- **Bit 4 (Bit 52 in GDT):** Reserved for our OS's use--we could do whatever we want here. Its set to 0.
- **Bit 5 (Bit 53 in GDT):** Reserved for something. Future options, maybe? Who knows. Set to 0.
- **Bit 6 (Bit 54 in GDT):** is the segment type (16 or 32 bit). we want 32 bits. After all-we are building a 32 bit OS! So set to 1.

Оставшийся байт является битами 24-32 базового адреса, логично, что в нашем случае они заполнены нулями. Все, с дескриптором кода разобрались.

Дескриптор данных

Наш дескриптор данных один в один с дескриптором кода, кроме бита 43. Этот бит установлен в дескрипторе кода и сброшен в дескрипторе данных.

Теперь, когда мы имеет готовую таблицу GDT нам остается лишь как то указать процессору где она находится. Это делается с помощью инструкции **lgdt**. В качестве параметра она принимает указатель на место в памяти, где содержится структура из размера таблицы GDT минут один и начального адреса таблицы:

```
gdt_data:  
    dw $ - dt_data - 1  
    dd dt_data
```

gdt_data это адрес начала GDT. Такой формат обязателен для команды **lgdt**. **lgdt** загружает виртуальный адрес начала таблицы загружается в 48-битный регистр GDTR:

```
lgdt [gdt_data] ; load GDT into GDTR
```

Важный момент! Gdtr содержит **линейный** а не физический адрес!

После того, как мы разобрались с глобальной таблицей дескрипторов мы переходим к установке первого бита регистра CR0.

- **Bit 0 (PE) : Puts the system into protected mode**
- **Bit 1 (MP) : Monitor Coprocessor Flag** This controls the operation of the **WAIT** instruction.
- **Bit 2 (EM) : Emulate Flag.** When set, **coprocessor instructions will generate an exception**
- **Bit 3 (TS) : Task Switched Flag** This will be set when the processor switches to another **task**.
- **Bit 4 (ET) : ExtensionType Flag.** This tells us what type of coprocessor is installed.
 - 0 - 80287 is installed
 - 1 - 80387 is installed.
- **Bit 5 : Unused.**
- **Bit 6 (PG) : Enables Memory Paging.**

Устанавливаю бит 0 в значение 1 мы указываем процессору продолжить его работу в 32-битном защищенном режиме (**protected mode**):

; go to protected mode:

```
mov eax, cr0  
or  eax, 1  
mov cr0, eax
```

Обратите внимание! Перед переходом в защищенный режим необходимо отключить прерывания. Если этого не сделать процессор выдаст ошибку **triple fault**.

Сразу после входа в защищенный режим мы сталкиваемся с проблемой. В реальном режиме мы использовали адресацию сегмент:смещение, но в защищенном режиме должны перейти на модель дескриптор:адрес.

Также необходимо иметь в виду, что реальный режим ничего не знает о GDT, в то время как для режима адресации защищенного режима GDT является обязательной для использования. Из-за этого в реальном режиме регистр CS все еще содержит последний используемый адрес сегмента, а в защищенном режиме он должен содержать индекс дескриптора таблицы GDT. Чтобы устранить эту проблему нам необходимо сделать так называемый **far jump**, используя дескриптор кода из созданной нами таблицы GDT.

Так как индекс нашего дескриптора кода равен 0x8 (то есть 8-байтовое смещение относительно начала GDT, именно там лежит наш дескриптор), код нашего far jump'a будет выглядеть следующим образом:

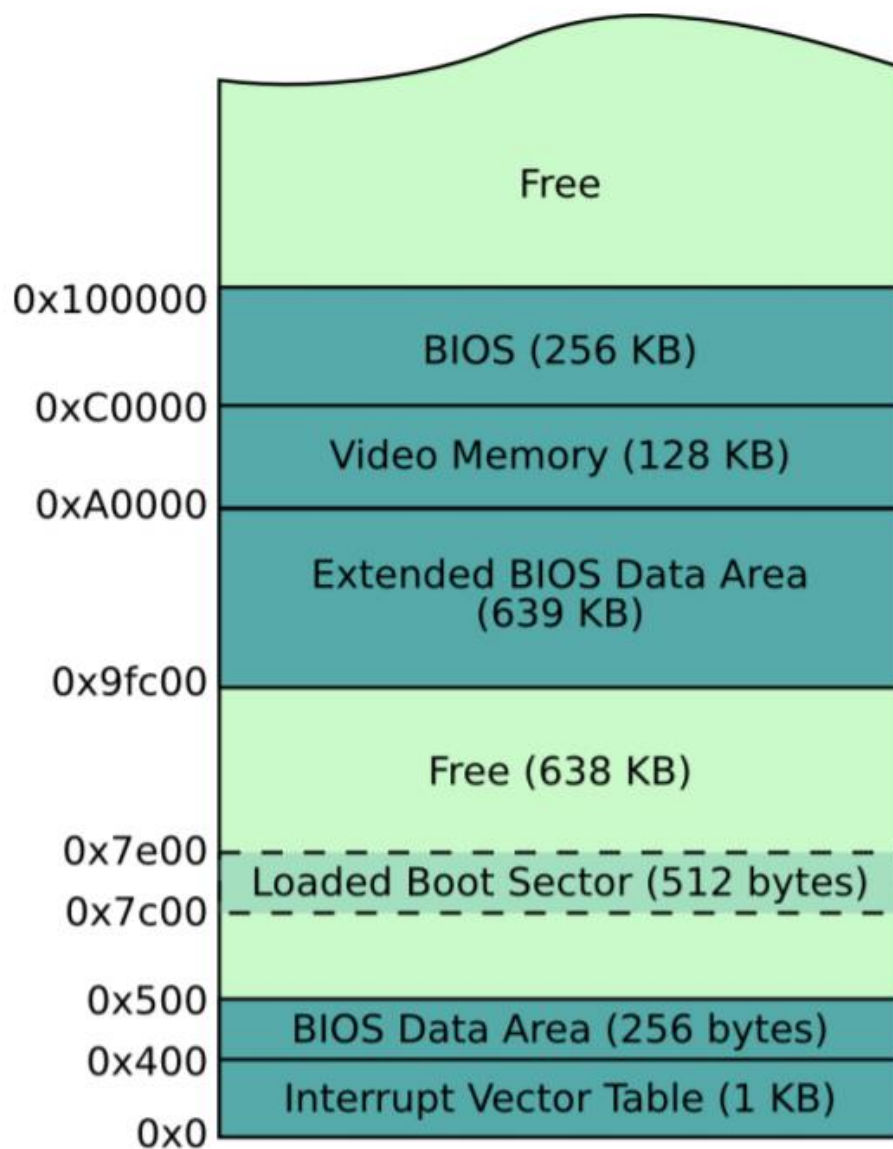
```
; go to 32-bit code  
jmp 8:start32
```

Также в защищённом режиме мы должны сбросить все сегментные регистры в новое корректное состояние:

```
use32  
start32:  
; prepare segment registers  
mov eax, 16  
mov ds, ax  
mov es, ax  
mov fs, ax  
mov gs, ax  
mov ss, ax  
  
.
```

После всех произведенных манипуляций мы должны оказаться в защищенном режиме. Чтобы проверить там ли мы, давайте выведем на экран в правый нижний угол символ '!'. Выведем его по всем правилам работы в защищенном режиме.

При старте компьютера BIOS запускается в текстовом режиме (MDA, Monochrome Display Adapter), который поддерживает 80 колонок и 25 строк. Видеоконтроллер использует регион отраженный (mapped) на физическую памяти **0xA0000 - 0xBFFFF**.



Регион 0xA0000 - 0xBFFFF в свою очередь делится на 2 части:

0xB0000 - 0xB7777 монохромный текстовый режим

0xB8000 - 0xBFFFF цветной текстовый режим и CGA совместимая графика.

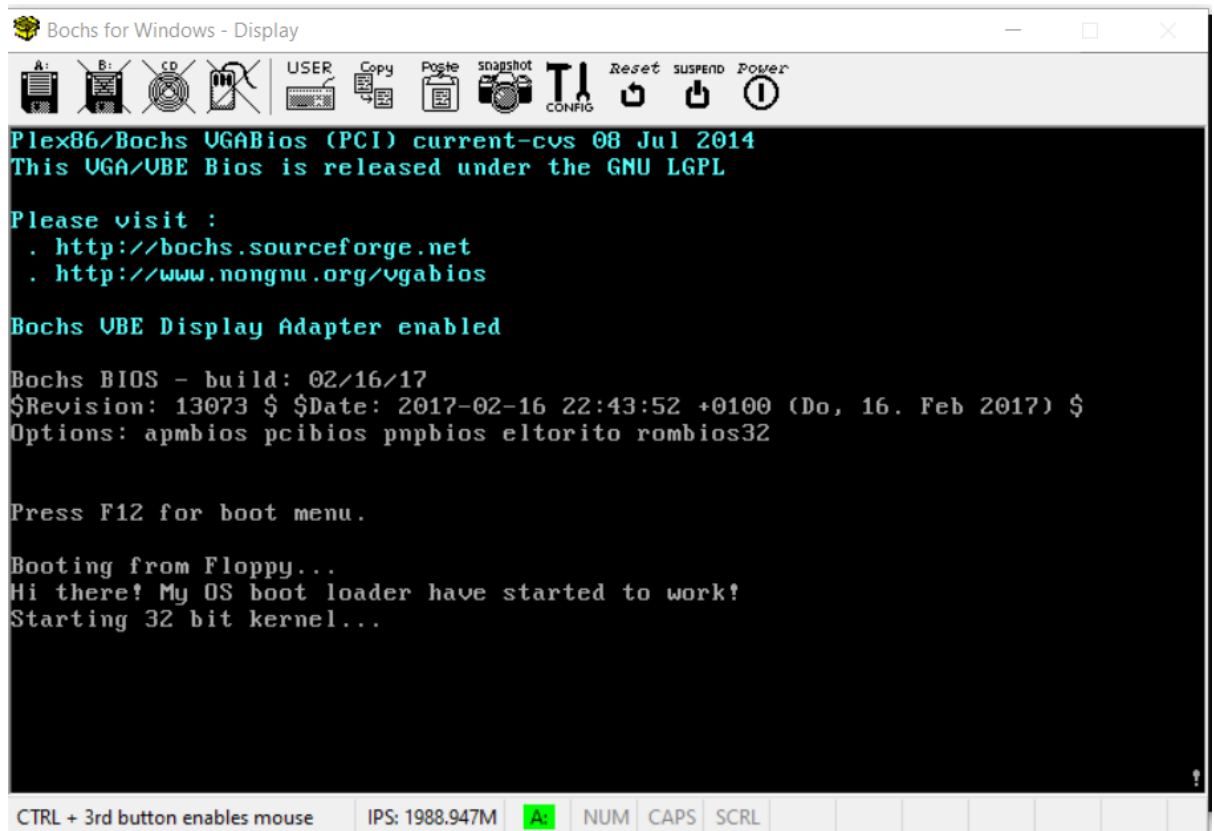
Запись в эти регионы памяти изменяет то, что отображается на экране.

```
mov byte[0xB8000 + (25 * 80 - 1) * 2], "!"
```

В вышеприведенном коде мы обращаемся по абсолютному 32-битному адресу к видео-памяти текстового режима, выводя символ "!" в правый нижний угол экрана (каждый символ занимает в памяти два байта - код символа и его атрибуты цвета).

Еще стоит обратить внимание на команду **movzx**, которая расширяет второй аргумент до первого. То есть из 16 битного значения SP получается 32-битное. Старшие биты обнуляются (мало ли, что там было до нас).

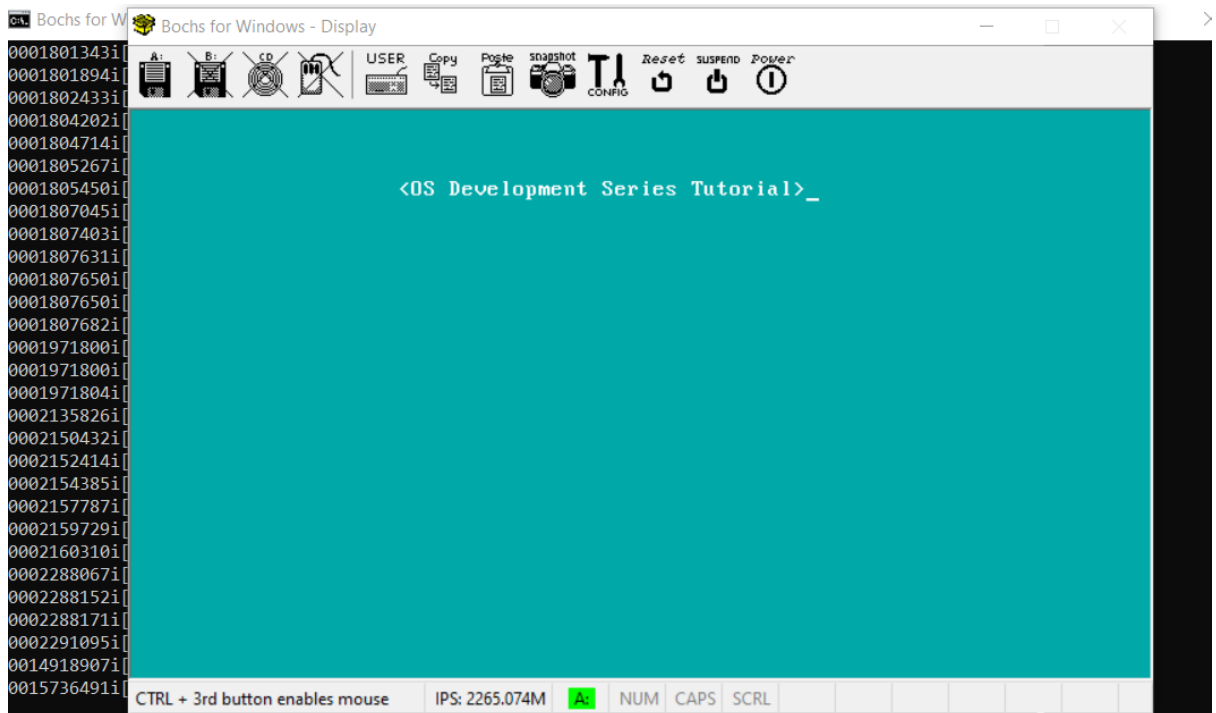
Теперь, если мы скомпилируем и запустим нашу операционную систему мы увидим следующее:



Исходный код:

<https://github.com/devsienko/c4os/tree/d7dde5c0c59635f1fe6f043c494b0fab2813c231>.

Вывод на экран можно немного улучшить, сделав его как на изображении:



Исходный код (файл `stdio.inc`) я взял из курса <http://www.brokenthorn.com/Resources/OSDev11.html>, исправив ошибку переполнения регистра **AL** (из-за чего курсор некорректно работал при положении дальше 128 символов, не исключая что это специфично для `fasm`). Здесь происходит очистка экрана, то есть мы заполняем его пробелами с установленными атрибутами заднего фона. Затем выводим строку с также предустановленными атрибутами цвета шрифта и его заднего фона. После вывода строки мы перемещаем курсор.

Исходный код: <https://github.com/devsienko/c4os/tree/b7f8eec9e7de3a78cb46008588b42427b29ad037>.

После того, как мы перешли в защищенный режим нам необходимо активировать линию A20, 20-й линию шины адреса. При проектировании микропроцессора 80286 инженеры Intel допустили ошибку, позволившую из реального режима обращаться к части памяти за пределами младшего мегабайта — так называемой области верхней памяти (НМА). Чтобы компенсировать эту ошибку и гарантировать совместимость со старыми программами, в состав компьютера пришлось включить специальную схему, блокирующую 20-й разряд шины адреса (или все старшие разряды, начиная с 20-го — это зависит от особенностей чипсета) — Gate A20 (вентиль или шлюз линии A20). При использовании старых программ реального режима этот вентиль может быть закрыт, что обеспечит присутствие на линии A20 логического нуля, а при работе новых

программ, поддерживающих защищённый режим, этот вентиль должен быть обязательно открыт.

Существует несколько способов активации линии A20. Код каждого из таких случаев включен в файл A20.inc, который я взял из курса <http://www.brokenthorn.com/Resources/OSDev9.html>.

Исходный код:

<https://github.com/devsienko/c4os/tree/68a74e35d3a8d928ccc4bf4e30021aa127751404>.

Страничная адресация

Материал

ВЗЯТ

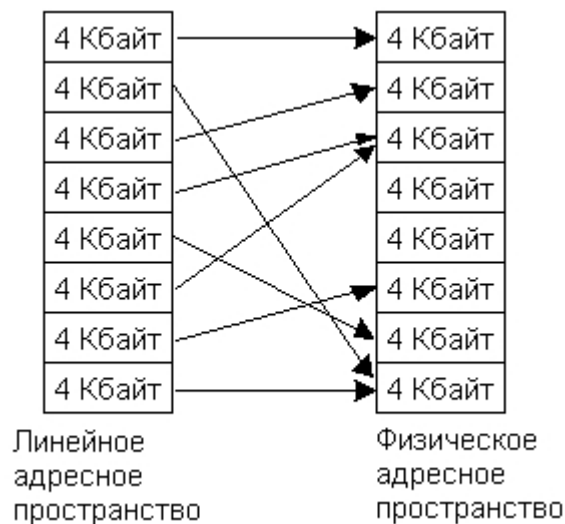
с

http://softcreate.narod.ru/asm/articles/base_manual/GL4/Index4.htm.

Если страничная трансляция включена, то линейные адреса преобразуются в физические в соответствии с содержимым страничных таблиц:



Страницей называется связный участок линейного или физического адресного пространства объемом 4 Кбайт. Программа работает в линейном адресном пространстве, не подозревая о существовании страничного преобразования или даже самих страниц. Механизм страничной трансляции отображает логические страницы на физические в соответствии с информацией, содержащейся в страничных таблицах. В результате отдельные 4х-килобайтовые участки программы могут реально находиться в любых несвязных друг с другом 4х-килобайтовых областях физической памяти:



Порядок размещения физических страниц в памяти может не соответствовать (и обычно не соответствует) порядку следования логических страниц. Более того, некоторые логические страницы могут перекрываться, фактически сосуществуя в одной и той же области физической памяти.

Страничная трансляция представляет собой довольно сложный механизм, в котором принимают участие аппаратные средства процессора и находящиеся в памяти таблицы преобразования. Назначение и взаимодействие элементов системы страничной трансляции схематически изображено на рисунке:



Система страничных таблиц состоит из двух уровней. На первом уровне находится каталог таблиц страниц (или просто каталог страниц) - резидентная в памяти таблица, содержащая **1024 4х-байтовых** поля с адресами таблиц страниц. На втором уровне находятся таблицы страниц, каждая из которых содержит так же **1024 4х-байтовых** поля с адресами физических страниц памяти. Максимально возможное число таблиц страниц определяется числом полей в каталоге и может достигать до **1024**. Поскольку размер страницы составляет **4 Кбайт**, **1024 таблицы по 1024 страницы** покрывают все адресное пространство (**4 Гбайт**).

Не все 1024 таблицы страниц должны обязательно иметься в наличии (кстати, они заняли бы в памяти довольно много места - 4 Мбайт). Если программа реально использует лишь часть возможного линейного адресного пространства, а так всегда и бывает, то неиспользуемые поля в каталоге страниц помечаются, как отсутствующие. Для таких полей система, экономя память, не выделяет страничные таблицы.

При включенной страничной трансляции линейный адрес рассматривается как совокупность трех полей: **10-битового** индекса в каталоге страниц, **10-битового** индекса в выбранной таблице страниц и **12-битового** смещения в выбранной странице. Напомним, что линейный адрес образуется путем сложения базового адреса сегмента, взятого из дескриптора сегмента, и смещения в этом сегменте, предоставленного программой.

Старшие 10 бит линейного адреса образуют номер поля в каталоге страниц. Базовый адрес каталога хранится в одном из управляющих регистров процессора, конкретно, в регистре **CR3**. Из-за того, что каталог сам представляет собой страницу и выровнен в памяти на границу 4 Кбайт, в регистре **CR3** для адресации к каталогу используются лишь старшие 20 бит, а младшие 12 бит зарезервированы для будущих применений.

Поля каталога имеют размер **4 байт**, поэтому индекс, извлеченный из линейного адреса, сдвигается влево на 2 бита (т.е. умножается на 4, так как адрес будет кратен 4, что 0100 в двоичной системе счисления, то есть два первых бита всегда будут нулями, поэтому можно сэкономить и не выделять под них память) и полученная величина складывается с базовым адресом каталога, образуя адрес конкретного поля каталога (примечание автора - раз размер 4 байта, то он всегда). Каждое поле каталога содержит физический базовый адрес одной из таблиц страниц, причем, поскольку таблицы страниц сами представляют собой страницы и выровнены в памяти на границу 4 Кбайт, в этом адресе значащими являются только старшие 20 бит.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
Page-Table Base Address (Базовый адрес таблицы страниц)																				AVAIL		G	P	S	0	A	P	C	D	P	W	T	U	/	S	R	/	W	P

Свободно для использования операционной системой _____

Глобальная страница (Global Page) _____

Размер страницы 4Кб/4Мб (Page Size = 0 для страниц 4Кб) _____

Доступ (Accessed) _____

Запрещение кэширования (Cach disabled) _____

Сквозная запись (Write-through) _____

Пользователь/Супервизор (User/Supervisor) _____

Чтение/Запись (Read/Write) _____

Присутствие (Present) _____

■ Зарезервированные биты. Должны устанавливаться только в указанные значения.

Bit 0 (P): Present flag

- 0: Page is not in memory
- 1: Page is present (in memory)

Bit 1 (R/W): Read/Write flag

- 0: Page is read only
- 1: Page is writable

Bit 2 (U/S): User mode/Supervisor mode flag

- 0: Page is kernel (supervisor) mode
- 1: Page is user mode. Cannot read or write supervisor pages

Bit 3 (PWT): Write-through flag

- 0: Write back caching is enabled
- 1: Write through caching is enabled

Bit 4 (PCD): Cache disabled

- 0: Page table will not be cached
- 1: Page table will be cached

Bit 5 (A): Access flag. Set by processor

- 0: Page has not been accessed

- 1: Page has been accessed

Bit 6 (D): Reserved by Intel

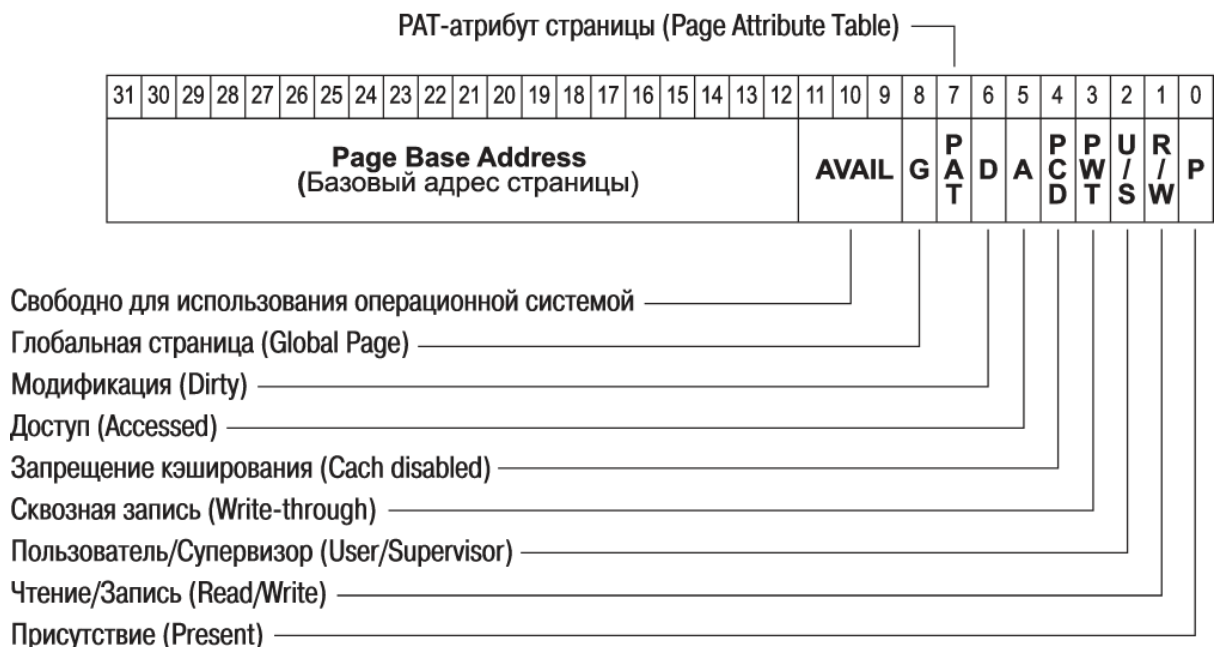
Bit 7 (PS): Page Size

- 0: 4 KB pages
- 1: 4 MB pages

Bit 8 (G): Global Page (Ignored)

Bits 9-11 (AVAIL): Available for use

Bits 12-31 (FRAME): Page Table Base address



■ Зарезервированные биты. Должны устанавливаться только в указанные значения.

- **Bit 0 (P):** Present flag
 - 0: Page is not in memory
 - 1: Page is present (in memory)
- **Bit 1 (R/W):** Read/Write flag
 - 0: Page is read only
 - 1: Page is writable
- **Bit 2 (U/S):**User mode/Supervisor mode flag
 - 0: Page is kernel (supervisor) mode
 - 1: Page is user mode. Cannot read or write supervisor pages
- **Bits 3-4 (RSVD):** Reserved by Intel

- **Bit 5 (A):** Access flag. Set by processor
 - 0: Page has not been accessed
 - 1: Page has been accessed
- **Bit 6 (D):** Dirty flag. Set by processor
 - 0: Page has not been written to
 - 1: Page has been written to
- **Bits 7-8 (RSVD):** Reserved
- **Bits 9-11 (AVAIL):** Available for use
- **Bits 12-31 (FRAME):** Frame address

Далее из линейного адреса извлекается средняя часть (биты 12...21), сдвигается влево на 2 бита и складывается с базовым адресом, хранящимся в выбранном поле каталога. В результате образуется физический адрес страницы в памяти, в котором опять же используются только старшие 20 бит. Этот адрес, рассматриваемый, как старшие 20 бит физического адреса адресуемой ячейки, носит название страничного кадра. Страничный кадр дополняется с правой стороны младшими 12 битами линейного адреса, которые проходят через страничный механизм без изменения и играют роль смещения внутри выбранной физической страницы.

Рассмотрим пример, позволяющий проследить цепочку преобразования виртуального адреса в физический. Пусть программа выполняет команду

mov EAX,DS:[EBX]

пусть при этом содержимое DS (селектор) составляет **1167h**, а содержимое EBX (смещение) **31678h**.

Старшие **13** бит селектора (число **116h**) образуют индекс дескриптора в системной дескрипторной таблице. Каждый дескриптор включает в себя довольно большой объем информации о конкретном сегменте и, в частности, его линейный адрес. Пусть в ячейке дескрипторной таблицы с номером **116h** записан линейный адрес (базовый адрес сегмента) **01051000h**. Тогда полный линейный адрес адресуемой ячейки определится, как сумма базового адреса и смещения:

Базовый адрес сегмента **01051000h**

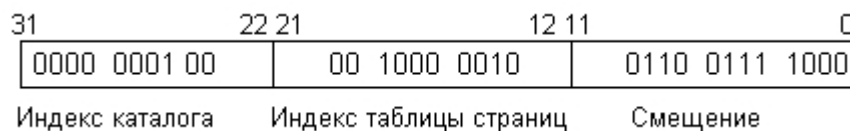
Смещение **00031678h**

Полный линейный адрес **01082678h**

При *выключенной* табличной трансляции величина **01082678h** будет представлять собой абсолютный физический адрес ячейки, содержимое которой

должно быть прочитано приведенной выше командой mov. Легко сообразить, что эта ячейка находится в самом начале 17-го мегабайта оперативной памяти.

Посмотрим, как будет образовываться физический адрес при использовании страничной трансляции адресов. Полученный линейный адрес надо разделить на три составляющие для выделения индексов и смещения:



Индекс каталога составляет **4h**. Умножение его на 4 даст смещение от начала каталога. Это смещение равно **10h**.

Индекс таблицы страниц оказался равным **82h**. После умножения на 4 получаем смещение в таблице страниц, равное в данном случае **210h**.

Предположим, что регистр **CR3** (базовый адрес каталога) содержит число 8000h. Тогда физический адрес ячейки в каталоге, откуда надо получить адрес закрепленной за данным участком программы таблицы страниц, составит **8000h + 10h = 8010h**.

Пусть по этому адресу записано число **46021h**. Его 12 младших битов составляют служебную информацию (в частности, бит 1 свидетельствует о присутствии этой таблицы страниц в памяти, а бит 5 говорит о том, что к этой таблице уже были обращения), а старшие биты, т.е. число **46000h** образуют физический базовый адрес таблицы страниц. Для получения адреса требуемой ячейки этой таблицы к базовому адресу надо прибавить смещение **210h**.

Результирующий адрес составит **46210h**. Будем считать, что по адресу **46210h** записано число **01FF5021h**. Отбросив служебные биты, получим адрес физической страницы в памяти **01FF5000h**. Этот адрес всегда оканчивается тремя нулями, так как страницы выровнены в памяти на границу 4 Кбайт. Для получения физического адреса адресуемой ячейки следует заполнить 12 младших бит полученного адреса битами смещения из линейного адреса нашей ячейки, в которых в нашем примере записано число **678h**. В итоге получаем физический адрес памяти **01FF5678h**, расположенный в конце 32-го Мбайта.

Начнем реализацию вышеизложенной теории в коде. Планируется, что нижние 2 Гб адресного пространства (**0x00000000 - 0x7FFFFFFF**) будут отведены под использование процессов, а верхние (**0x80000000 - 0xFFFFFFFF**) для ядра и будут общими для всех процессов. Там будут находиться код ядра, данные для межпроцессного взаимодействия и т. д. Сам код ядра и его самые важные данные -

последние 4 МБ **0xFFC00000** - **0xFFFFFFFF**, потому что ровно столько памяти переадресует одна таблица страниц самого нижнего уровня.

Когда я впервые прочитал о том, что верхние два Гб отводятся для ядра у меня было недоумение, зачем размещать в таблицах страниц процесса таблицы страниц ядра. Оказывается все просто. Из-за аппаратной-программной защиты код приложения не сможет получить доступ к таблицам страниц ядра, но будет выигрыш в производительности. Если надо переключиться на код ядра не нужно будет переключаться на новый каталог страниц, обновлять кэш преобразований адресов и прочее. Все уже находится в памяти и без лишних манипуляций может быть использовано.

Создадим таблицу каталогов и две таблицы страниц. Первая таблица будет байт в байт монтировать первый мегабайт памяти, чтобы после перехода на страничную адресацию наш код продолжил свое выполнение без ошибок, в противном случае это будет невозможно. Другая таблица страниц будет предназначена для кода ядра. И монтировать последние 4 Mb.

```
.prepare_page_tables:
```

```
    ; clear page tables
```

```
    xor ax, ax
```

```
    mov cx, 3 * 4096 / 2 ; count of words (stosw), / 2 because one word is 2 bytes
```

```
    mov di, 0x1000 ; address (stosw)
```

```
    rep stosw
```

```
    ; fill the page directory that starts at 4096 (0x1000)
```

```
    mov word[0x1000], 0x2000 + 111b ; points to first page table
```

```
    mov word[0x1FFC], 0x3000 + 111b ; last page table record, 8188 (0x1FFC) = 4096 (page  
directory base) + 4096 (page directory size) - 4 (size of one page directory record)
```

```
    ; fill the first page table
```

```
    mov eax, 11b
```

```
    mov cx, 0x100000 / 4096 ; map first 1 Mb
```

```
    mov di, 0x2000 ; address (stosd)
```

```
@ @:
```

```
    stosd
```

```
    add eax, 0x1000
```

```
    loop @b
```

```
    ; fill the last page table
```

```
    mov di, 0x3000
```

```
    mov eax, dword[0x6000]
```

```

    or eax, 11b
    mov ecx, 1024 ; map whole table
@@:
    stosd
    add eax, 0x1000
    loop @b
    mov word[0x3FF4], 0x4000 + 11b ; Kernel stack
    mov word[0x3FF8], 0x3000 + 11b ; Kernel page table

; load page directory to CR3
    mov eax, 0x1000
    mov cr3, eax
    ret

```

Для справки. Команда **rep stosw** заполнить блок из **(E)CX** слов по адресу **ES:(E)DI** содержимым **AX**. Команда **stosd** сохраняет регистр **EAX** в ячейке памяти по адресу **ES:DI**. После выполнения команды, регистр **DI** увеличивается на 4, если флаг **DF = 0**, или уменьшается на 4, если **DF = 1**. Если команда используется в 32-разрядном режиме адресации, то используется регистр **EDI**.

Теперь необходимо включить страничную трансляцию. Для этого нужно установить бит 31 регистра **CR0**. Это можно сделать одновременно с установкой бита для перехода в защищенный режим:

```

    mov eax, cr0
    or eax, 0x80000001
    mov cr0, eax

```

На этом все. Теперь мы находимся в защищенном режиме с включенной страничной адресацией. Для демонстрации выведем в правый нижний угол сообщение "OK". Адрес **0xB8000** (был описан выше) расположен в первом мегабайте памяти, поэтому для вывода на экран мы можем использовать его без изменений. Выведем символ 'K':

```

    mov byte[0xB8000 + (25 * 80 - 1) * 2], "K"

```

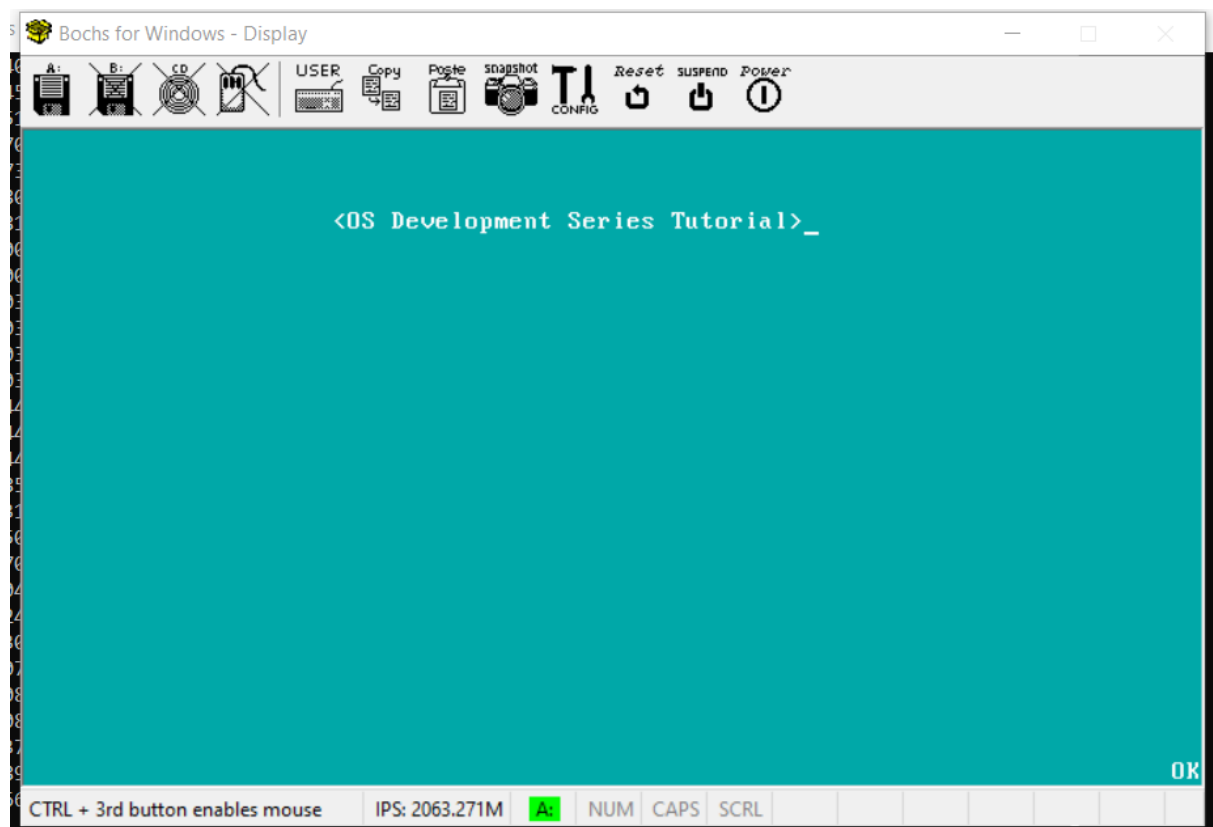
А вот символ 'O' выведем задействуя механизм страничной адресации:

```

    mov dword[0x3FFC], 0xB8000 + 11b
    mov byte[0xFFFFF000 + (25 * 80 - 2) * 2], "O"

```

Последняя запись (**Page Table Entry, PTE**) второй таблицы страниц, которую мы создали выше, начинается по адресу **0x3FFC**. Она отвечает за отображение адреса **0xFFFFF000** на физическую память. В коде выше мы смонтировали этот адрес, чтобы он смотрел на физический адрес **0xB8000** и после этого записали по виртуальному адресу **0xFFFFF000 + (25 * 80 - 2) * 2** символ 'O'. В результате он появится на экране рядом с буквой 'K' в правом нижнем углу.



Исходный код:

<https://github.com/devsienko/c4os/tree/789ce5483d9afcdfa68e2555ae85269494cfed2f>.

Переходим на Си

Загрузчик готов, в защищенный режим мы перешли, страничную адресацию включили. Что дальше? Перейдем к написанию ядра. Начнем с самой примитивной его реализации, а именно:

```
void kernel_main() {  
    char *screen_buffer = (void*)0xB8000;  
    char *msg = "Hello world!";  
    unsigned int i = 24 * 80;
```

```

while (*msg) {
    screen_buffer[i * 2] = *msg;
    msg++;
    i++;
}
}

```

Мы создаем указатель на регион памяти и выводим “Hello, World!” на последнюю строку, то есть в левый нижний угол экрана.

Если мы скомпилируем этот исходный код как есть (я делаю это из под Windows 10), то в бинарный файл будут добавлены заголовки формата PE, код стандартных библиотек операционной системы и выбранный случайно адрес входа. Нас это совсем не устраивает. Нам нужно чтобы код стартовал с адреса **0xFFC00000**, желательно, шел сразу с самого начала файла без всяких заголовков и не содержал никаких стандартных библиотек операционной системы, потому что как таковой операционной системы у нас сейчас и нет, мы можем пользоваться только кодом, который написали до сих пор, это все чем мы располагаем.

Одним из способов добиться желаемого в операционной системе Windows 10 можно создав скрипт линковщика, указать в нем базовый адрес, точку входа и смещения секций кода ядра.

```

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 = .;
        }
    }
}

```

Также мы будем использовать промежуточный между загрузчиком и ядром код на ассемблере. Он нам потребуется, чтобы передать ядру данные через входные параметры функции **kernel_main**.

format ELF

```

public _start
extrn _kernel_main

section ".text" executable

_start:
    movzx edx, dl
    push edx
    push esi
    push ebx
    call _kernel_main

@@:
    jmp @@

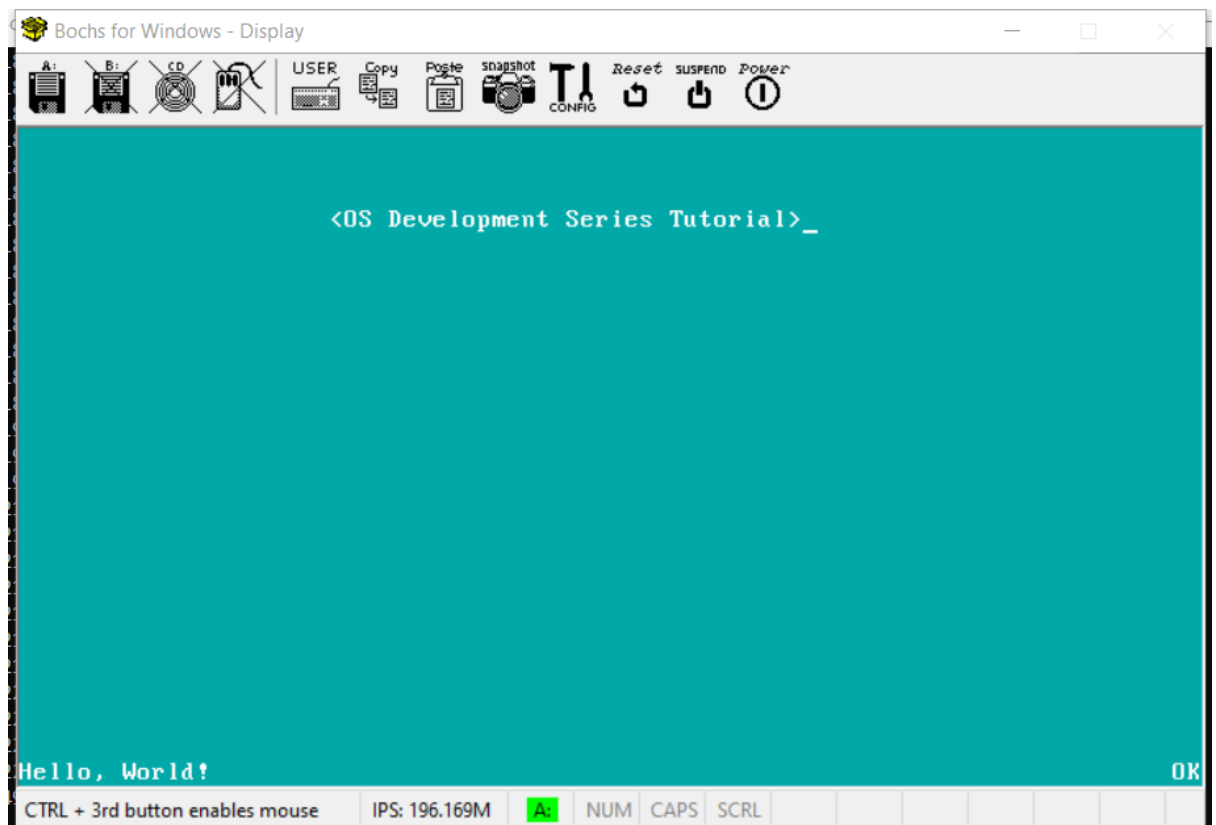
;section ".data" writable

```

Весь исходный код можно посмотреть здесь

<https://github.com/devsienko/c4os/tree/3ca62f7b2e9eda110bcf81ed001e4ac485fa6acf>.

После компиляции кода и запуска нашей операционной системы мы должны увидеть следующее:



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 перейдет 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 уйдет в 0x100, а старшая в 0x101. Для работы с портами существует две ассемблерных команды - `in` и `out`. Они существуют в единственной форме - **`in регистр, номер_порта`** и **`out ном_порта, регистр`**. Эта справка нам скоро пригодится.

Стандартная библиотека и драйвер текстового экрана

Перед тем как двигаться дальше подготовим вспомогательные функции и типы данных, которые разместим в файлах **stdlib.h** и **stdlib.c**, а также драйвер текстового экрана в файлах **tty.c** и **tty.h**.

В файле **stdlib.h** мы скрываем стандартные типы данных за именами синонимов чтобы сделать код более переносимым, определяем некоторые полезные макросы, функции работы с памятью, строками, а также макросы для работы с портами ввода-вывода. В файле **tty.h** мы определяем ряд функций упрощающих вывод информации на экран.

```
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, ...);
```

Код достаточно прост, внимание может заслуживать работа с мигающим курсором, получение его позиции и ее изменение. 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);
    }
}
```

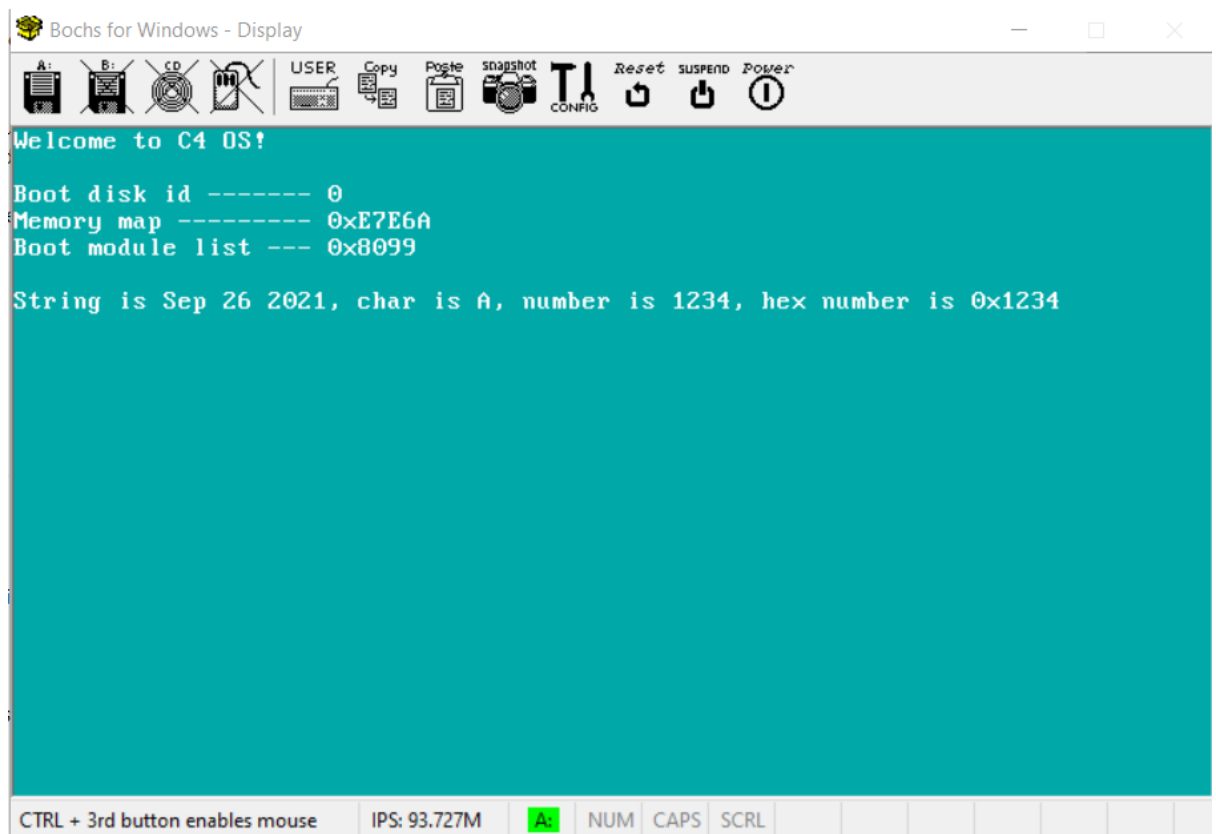
Ну и наконец самая сложная функция - move_cursor. Эта функция обновляет значение переменной cursor, а также следит за тем, чтобы его позиция не вышла за границу экрана. Если такое случается курсор перемещается на начало последней строки, всё содержимое экрана сдвигается на 1 строку вверх, а последняя строка очищается.

Для взаимодействия с контроллером дисплея используются порты ввода-вывода. Контроллер дисплея имеет два порта - tty_io_port и tty_io_port + 1. Первый задает 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.

```
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);
}
```

После компиляции и запуска вы должны увидеть такой экран:



Исходный

код:

<https://github.com/devsienko/c4os/tree/8d460dadf90908066faf5bb3bee32b370499c0e4>.

Обработка аппаратных прерываний

Если опустить детали, схематично обработку прерываний можно представить следующим образом:

- внешнее устройство посылает сигнал программируемому контроллеру прерываний (Programmable Interrupt Controller - PIC), например при нажатии кнопки клавиатуры;
- контроллер прерываний сигнализирует о прерывании процессору через специальную линию о прерывании и передает ему **номер прерывания**;
- процессор используя **таблицу IDT (таблицу прерываний)** и **номер прерывания** вызывает **обработчик прерывания**.

Теперь более подробно. 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 сегмента дескриптор содержит различные атрибуты прерывания. Эти дескрипторы располагаются в таблице последовательно без каких-либо промежутков. Самый первый дескриптор, расположенный по самому младшему адресу в начале таблицы соответствует прерыванию 0, а самый последний дескриптор, расположенный по самому старшему адресу в таблице, соответствует прерыванию 255. 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;
```

Offset	Size	Name	Description
0	2 bytes	Offset (low)	Low 2 bytes of interrupt handler offset
2	2 bytes	Selector	A code segment selector in the GDT
4	1 byte	Zero	Zero - unused
5	1 byte	Type/attributes	Type and attributes of the descriptor
6	2 bytes	Offset (high)	High 2 bytes of interrupt handler offset

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 отличие 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.

Обработчик прерываний это обыкновенная функция, которая будет вызвана в ответ на прерывание. Она может быть написана как на СИ так и на ассемблере, ничего необыкновенного, разве что для выхода из этой функции вместо команды RET следует использовать IRET, которая в отличие от RETF, восстанавливает не только CS и IP, но и FLAGS (а **если происходит смена уровня привилегий!** то еще ESP и SS, но об этом позже).

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 (каскадирование). Один обрабатывает IRQ с 0 по 7, а другой с 8 по 15. Второй подключен к выводу IRQ2 первого. Нам нет необходимости беспокоиться 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).

Так как контроллер прерываний программируемый, мы можем перенастроить маппинг запросов от устройств на номера прерываний. Делается это с помощью специальных инициализационных команд (**Initialization Control Words, ICW**).

Перейдем к реализации - загрузим значение в IDTR и разрешим прерывания от PIC (команда STI). Обработаем IRQ0, на котором висит таймер, генерирующий прерывания примерно 18,5 раз в секунду (он нужен для реализации вытесняющей многозадачности и подсчёта времени). Также перенастроим 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

void init_interrupts();
void set_int_handler(uint8 index, void *handler, uint8 type);

#endif
```

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 недопустимо, чтобы смещения полей отклонялись от заданных. Например, без этого атрибута структура `IDTR` будет занимать не 6, а 8 байт, потому что `base` будет выровнен на 4 байта (сейчас он выровнен лишь на 2, хотя занимает 4). В результате структура не будет корректной. К сожалению, каждый компилятор предоставляет возможность отключить выравнивание разными способами, вышеприведенный вариант для GCC и MinGW, но он не подойдёт, например, для компилятора MSVC, для которого существует свой синтаксис.

Инициализируем PIC. Это делается последовательной отсылкой **ICW (Initialization Control Words)** через порты ввода-вывода. 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`.

ICW 1

Это первое ICW для инициализации PIC. Его значение должно быть передано командному регистру первичного PIC. Его формат:

Initialization Control Word (ICW) 1				
Bit Number	Value	Description		
0	IC4	If set(1), the PIC expects to receive IC4 during initialization.		
1	SNGL	If set(1), only one PIC in system. If cleared, PIC is cascaded with slave PICs, and ICW3 must be sent to controller.		
2	ADI	If set (1), CALL address interval is 4, else 8. This is usually ignored by x86, and is default to 0		
3	LTIM	If set (1), Operate in Level Triggered Mode. If Not set (0), Operate in Edge Triggered Mode		

4	1	Initialization bit. Set 1 if PIC is to be initialized		
5	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0		
6	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0		
7	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0		

Чтобы инициализировать первичный PIC, все что нам нужно сделать это создать ICW и выставить нужным образом его биты.

- **Bit 0** - Set to 1 so we can sent ICW 4
- **Bit 1** - PIC cascading bit. x86 architectures have 2 PICs, so we need the primary PIC cascaded with the slave. Keep it 0
- **Bit 2** - CALL address interval. Ignored by x86 and kept at 8, so keep it 0
- **Bit 3** - Edge triggered/Level triggered mode bit. By default, we are in edge triggered, so leave it 0
- **Bit 4** - Initialization bit. Set to 1
- **Bits 5...7** - Unused on x86, set to 0.

Учитывая вышесказанное мы получим набор битов **00010001** или 0x11. Инициализируем PIC, отправив значение 0x11 в регистр контроллера PIC смалленного на порт 0x20.

; Setup to initialize the primary PIC. Send ICW 1

mov **al**, 0x11

out 0x20, **al**

; Remember that we have 2 PICs.

; Because we are cascading with this second PIC,

; send ICW 1 to second PIC command register

out 0xA0, **al** *; slave PIC command register*

Так как мы будем использовать каскадирование, нам надо также отправить контроллеру ICW 3. А так как мы выставили бит 0, нам необходимо еще и отправить ICW4. Сделаем это чуть позже, сперва разберемся с ICW 2.

ICW 2

Это контрольное слово используется, чтобы отобразить базовый адрес IVT, который будет использоваться PIC. **Это важно!**

Initialization Control Word (ICW) 2		
Bit Number	Value	Description
0-2	A8/A9/A10	Address bits A8-A10 for IVT when in MCS-80/85 mode.
3-7	A11(T3)/A12(T4)/A13(T5)/A14(T6)/A15(T7)	Address bits A11-A15 for IVT when in MCS-80/85 mode. In 80x86 mode, specifies the interrupt vector address. May be set to 0 in x86 mode.

During initialization, we need to send ICW 2 to the PICs to tell them where the base address of the IRQ's to use. If an ICW1 was sent to the PICs (With the initialization bit set), you must send ICW2 next. **Not doing so can result in undefined results.** Most likely the incorrect interrupt handler will be executed.

Unlike ICW 1, which is placed into the PIC's data registers, ICW 2 is sent to the data Registers, as software ports 0x21 for the primary PIC, and port 0xA1 for the secondary PIC. (Please see the **8259A Software Port Map** table for a complete listing of PIC software ports).

Okay, so assuming we have just sent an ICW 1 to both PICs (Please see the above section), let's send an ICW 2 to both PICs. This will map a base IRQ address to both PICs.

This is very simple, but we must be careful at where we map the PICs to. Remember that the first 31 interrupts (0x0-0x1F) are reserved (Please see the above **x86 Interrupt Vector Table (IVT)** table). As such, we have to insure we do not use any of these IRQ numbers.

Instead, let's map them to IRQs 32-47, right after these reserved interrupts. the first 8 IRQ's are handled by the primary PIC, so we map the primary PIC to the base address of 0x20 (32 decimal), and the secondary PIC at 0x28 (40 decimal). Remember there are 8 IRQ's for each PIC.

```
; send ICW 2 to primary PIC
```

```

mov  al, 0x20 ; Primary PIC handled IRQ 0..7. IRQ 0 is now
mapped to interrupt number 0x20
out  0x21, al

; send ICW 2 to secondary controller
mov  al, 0x28 ; Secondary PIC handles IRQ's 8..15. IRQ 8 is now
mapped to use interrupt 0x28
out  0xA1, al

```

ICW 3

This is an important command word. It is used to let the PICs know what IRQ lines to use when communicating with each other.

ICW 3 Command Word for Primary PIC

Initialization Control Word (ICW) 3 - Primary PIC		
Bit Number	Value	Description
0-7	S0-S7	Specifies what Interrupt Request (IRQ) is connected to slave PIC

ICW 3 Command Word for Secondary PIC

Initialization Control Word (ICW) 3 - Secondary PIC		
Bit Number	Value	Description
0-2	ID0	IRQ number the master PIC uses to connect to (In binary notation)
3-7	0	Reserved, must be 0

We must send an ICW 3 whenever we enable cascading within ICW 1. this allows us to set which IRQ to use to communicate with each other. Remember that the 8259A Microcontroller relies on the IR0-IR7 pins to connect to other PIC devices. With this, it uses the CAS0-CAS2 pins to communicate with each other.

We need to let each PIC know about each other and how they are connected. We do this by sending the ICW 3 to both PICs containing which IRQ line to use for both the master and associated PICs.

Remember: The 80x86 architecture uses IRQ line 2 to connect the master PIC to the slave PIC.

Knowing this, and remembering that we need to write this to the data registers for both PICs, we need to follow the formats shown above.

Note that, in the ICW 3 for the primary PIC, **Each bit represents an interrupt request.** That is...

IRQ Lines for ICW 2 (Primary PIC)	
Bit Number	IRQ Line
0	IR0
1	IR1
2	IR2
3	IR3
4	IR4
5	IR5
6	IR6
7	IR7

Notice that IRQ 2 is Bit 2 within ICW 3. So, in order to set IRQ 2, we need to set bit 2 (Which is at 0100 binary, or 0x4).

Here is an example of sending ICW 3 to the primary PIC:

```
; Send ICW 3 to primary PIC
```

```
mov    al, 0x4          ; 0x4 = 0100 Second bit (IR Line 2)
```

```
out    0x21, al        ; write to data register of primary PIC
```

To send this to the secondary PIC, we must remember that we must send this in **binary notation**. Please refer to the table above. Note that only Bits 0...2 are used to represent the IRQ line. By using binary notation, we can refer to the 8 IRQ lines to choose from:

IRQ Lines for ICW 2 (Secondary PIC)				
Binary	IRQ Line			
000	IR0			
001	IR1			
010	IR2			
011	IR3			
100	IR4			
101	IR5			
110	IR6			
111	IR7			

Simple enough. Notice that this just follows a binary<->decimal conversion in the above table.

Because we are connected by IRQ line 2, we need to use bit 1 (Shown above).

Here is a complete example, that sends a ICW 2 to both primary and secondary PIC controllers:

```
; Send ICW 3 to primary PIC
```

```
mov    al, 0x4          ; 0x04 => 0100, second bit (IR line 2)
```

```
out    0x21, al        ; write to data register of primary PIC
```

```

; Send ICW 3 to secondary PIC
mov    al, 0x2          ; 010=> IR line 2
out    0xA1, al         ; write to data register of secondary PIC

```

Thats all there is to it ;)

Okay, so now both PICs are connected to use IR line 2 to communicate with each other. We have also set a base interrupt number for both PICs to use.

This is great, but we are not done yet. Remember that, when building up ICW 1, **if bit 0 is set, the PIC will be expecting us to send it ICW 4**. As such, we need to send ICW 4, the final ICW, to the PICs.

ICW 4

Yey! This is the final initialization control word. This controls how everything is to operate.

Initialization Control Word (ICW) 4				
Bit Number	Value	Description		
0	uPM	If set (1), it is in 80x86 mode. Cleared if MCS-80/86 mode		
1	AEOI	If set, on the last interrupt acknowledge pulse, controller automatically performs End of Interrupt (EOI) operation		
2	M/S	Only use if BUF is set. If set (1), selects buffer master. Cleared if buffer slave.		
3	BUF	If set, controller operates in buffered mode		
4	SFNM	Special Fully Nested Mode. Used in systems with a large amount of cascaded controllers.		
5-7	0	Reserved, must be 0		

This is a pretty powerful function. Bits 5..7 are always 0, so lets focus on the other bits and peices (pun entended ;))

The PIC was originally designed to be a generic microcontroller, even before the 80x86 existed. As such, it contains a lot of different operation modes designed for different systems. One of these modes is the **Special Fully Nested Mode**.

The x86 family does not support this mode, so you can safely set bit 4 to 0.

Bit 3 is used for buffered mode. For now, set this to 0. We will cover modes of operation later. Bit 2 is only used when bit 3 is set, so set this to 0. With this, Bit 1 is rarely used either.

As such, we only need to set bit 0, which enables the PIC for 80x86 mode.

Simple enough. So, to send ICW 4, all we need to do is this:

```
mov al, 1 ; bit 0 enables 80x86 mode
```

```
; send ICW 4 to both primary and secondary PICs
```

```
out 0x21, al
```

```
out 0xA1, al
```

```
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);
```

Спроецируем таблицу прерываний в виртуальное адресное пространство и загрузим правильное значение в IDTR.

```
void init_interrupts() {
    *((size_t*)0xFFFFFFF0) = 0x8000 | 3;
    memset(idt, 0, 256 * sizeof(IntDesc));
    IDTR idtr = {256 * sizeof(IntDesc), idt};
    asm("lidt (,%0,)"::"a"(&idtr));
}
```

Назначим обработчик прерывания таймера и разрешить прерывания:

```
set_int_handler(irq_base, timer_int_handler, 0x8E);  
    asm("sti");  
}
```

Функция `set_int_handler`. Помимо собственно заполнения элемента таблицы прерываний эта функция отключает прием прерываний на время своей работы. Инструкция `PUSHF` сохраняет в стек старое значение регистра флагов, 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 запрещенных прерываниях, он не разрешит их.

```
void set_int_handler(uint8 index, void *handler, uint8 type) {  
    asm("pushf\ncli");  
    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");  
}
```

На этом инициализация подсистемы обработки прерываний завершена, осталось написать обработчик прерывания таймера `timer_int_handler`. Но мы используем язык высокого уровня. 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
```

Для обработчика прерываний нам нужно возвращать инструкцию `iret`. К тому же не сохраняют значения многих регистров. Сейчас решим проблему специальным макросом.

```
#define IRQ_HANDLER(name) void name(); \
```

```

asm("_#name ": pusha \n call _wrp_ " #name " \n movb $0x20, %al \n outb %al, $0x20 \n outb
%al, $0xA0 \n popa \n iret"); \
void wrp_##name()

```

Этот макрос сначала описывает функцию обработки IRQ на чистом Assembler - обработчик сохраняет все регистры процессора в стек с помощью команды PUSHА, затем выполняет вызов функции на Си, потом отправляет байт 0x20 в порты контроллеров PIC (контроллер ждет 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 POPА и IRET соответственно.

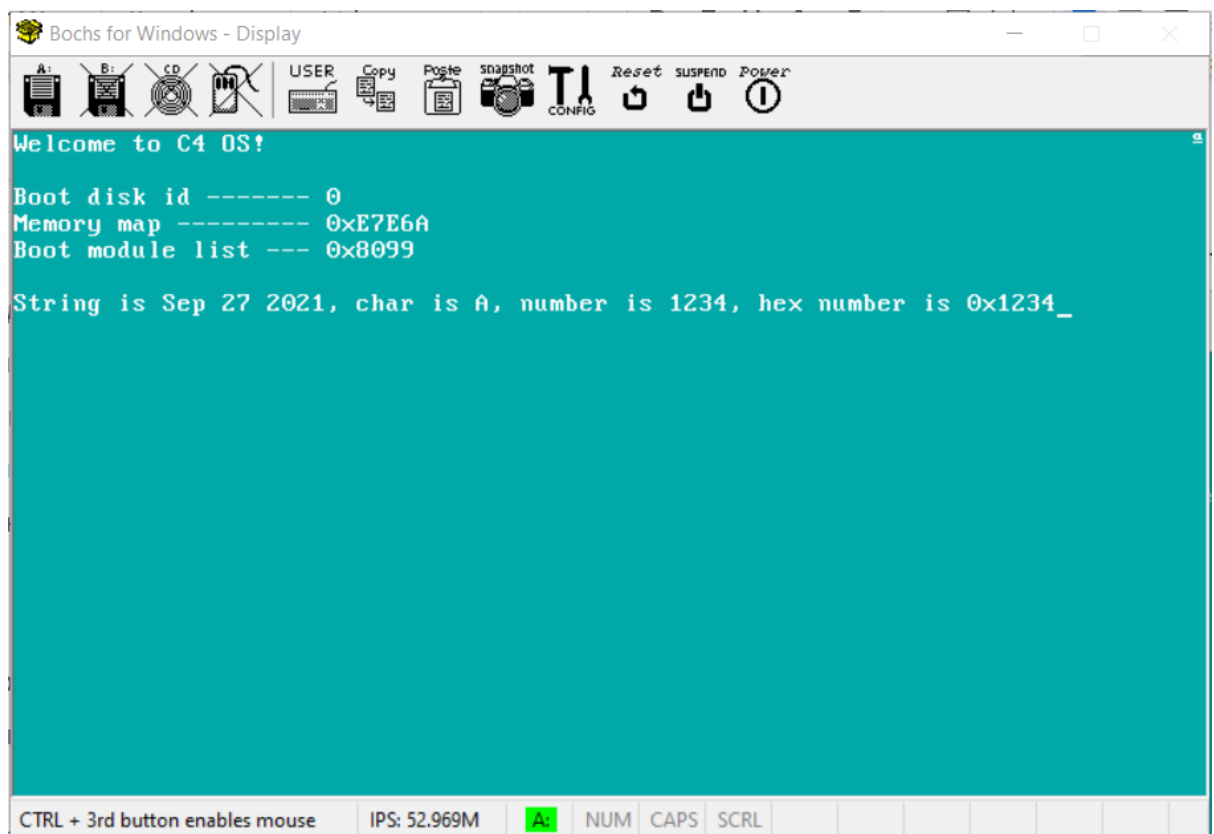
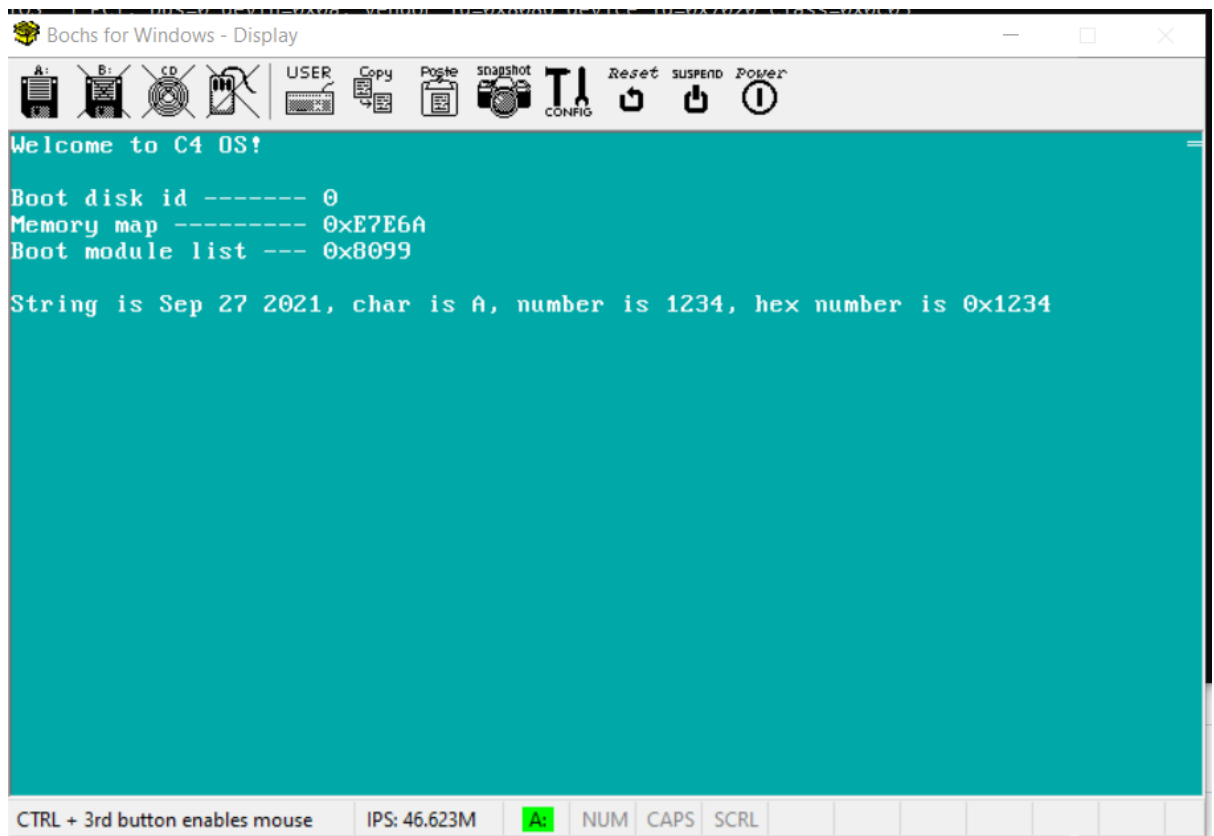
Теперь мы можем описать обработчик таймерного прерывания в interrupts.c.

```

IRQ_HANDLER(timer_int_handler) {
    (*((char*)(0xB8000 + 79 * 2)))++;
}

```

Обработчик увеличивает на 1 код символа в правом верхнем углу текстового экрана для наглядности работы. 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. 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 ресурсоемкие процедуры, потому что это скажется на производительности всей системы не лучшим образом.

Исходный код:

<https://github.com/devsienko/c4os/tree/2bcebd81277c87403034d99800ad63504476b728>.

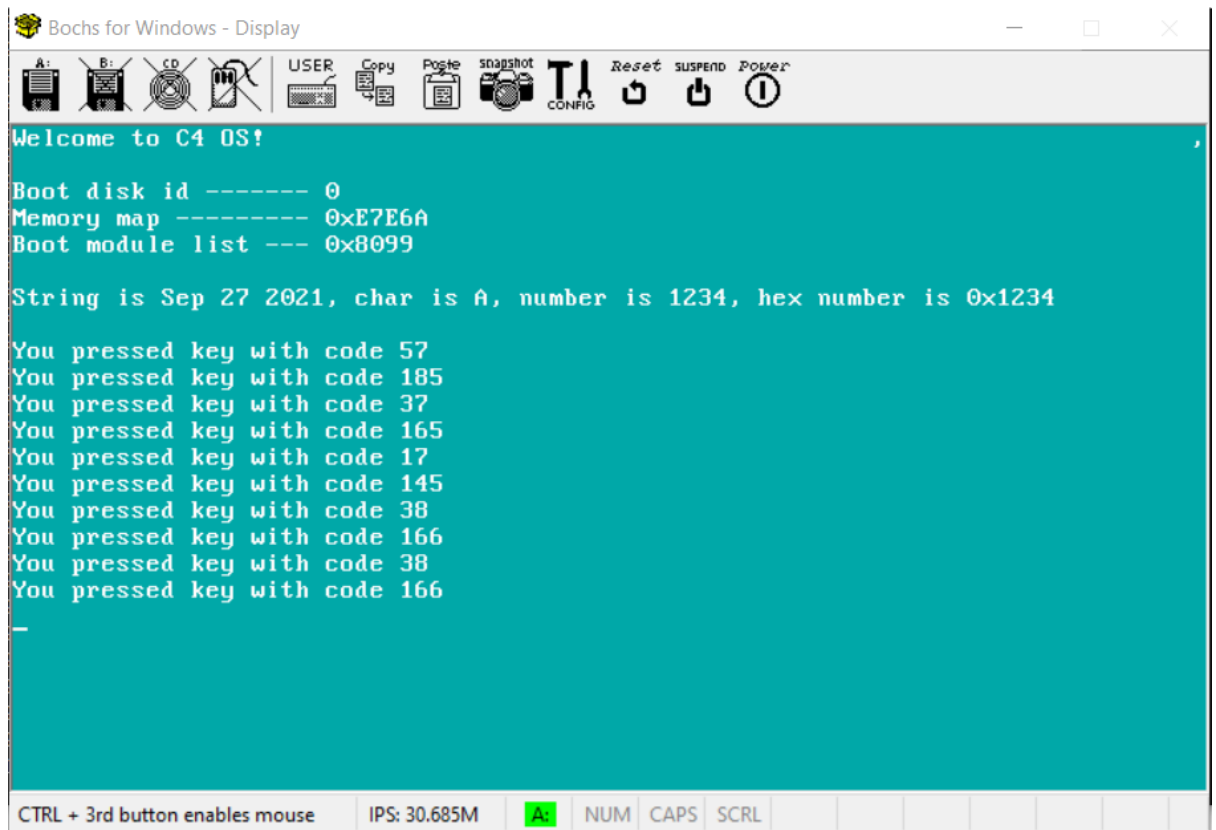
Взаимодействие с клавиатурой

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 нее 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 не 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.). Преобразование кодов мы реализуем немного позже, а сейчас лишь напишем базовый обработчик. Сначала добавим установку обработчика в `init_tty` в `tty.c` и реализуем сам обработчик, который будет выводить на экран сообщение с кодом символа.

```
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);
}
```

Отпускание клавиши тоже порождает прерывания, только прочитанный код клавиши на 0x80 больше, чем код нажатия.



Исходный

код:

<https://github.com/devsienko/c4os/tree/86e1c050887ebd83f0c33533e1ee4132a320a290>.

Чтение символов с клавиатуры

На данный момент мы получаем скан-коды при нажатии клавиш клавиатуры и сейчас нам предстоит преобразовать их в символы. Для этого создадим файл **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', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\', '\n',
    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', 'D', '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, // F16 - F24
0, 0, 0, 0, 0, 0, 0, 0
};

```

```

#endif

```

Здесь есть два набора для случая когда клавиша **shift** нажата и не нажата.

Обычно драйвер клавиатуры создает программный буфер нажатых клавиш. Если приложение хочет получить ввод с клавиатуры, оно читает содержимое буфера или ждет 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.

```

#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. 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**.

Изменим обработчик прерывания IRQ1, убрав вывод сообщений и добавив запись символа в буфер.

```
IRQ_HANDLER(keyboard_int_handler) {
    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);
}
```

Добавим несколько новых функций в **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.

```
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 позволяет читать одиночный символ с клавиатуры, умеет ждать нажатий при необходимости, а также преобразовывать скан-коды в символы с учётом нажатости клавиши Shift.

И последняя функция для считывания целой строки символов.

```

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;
}

```

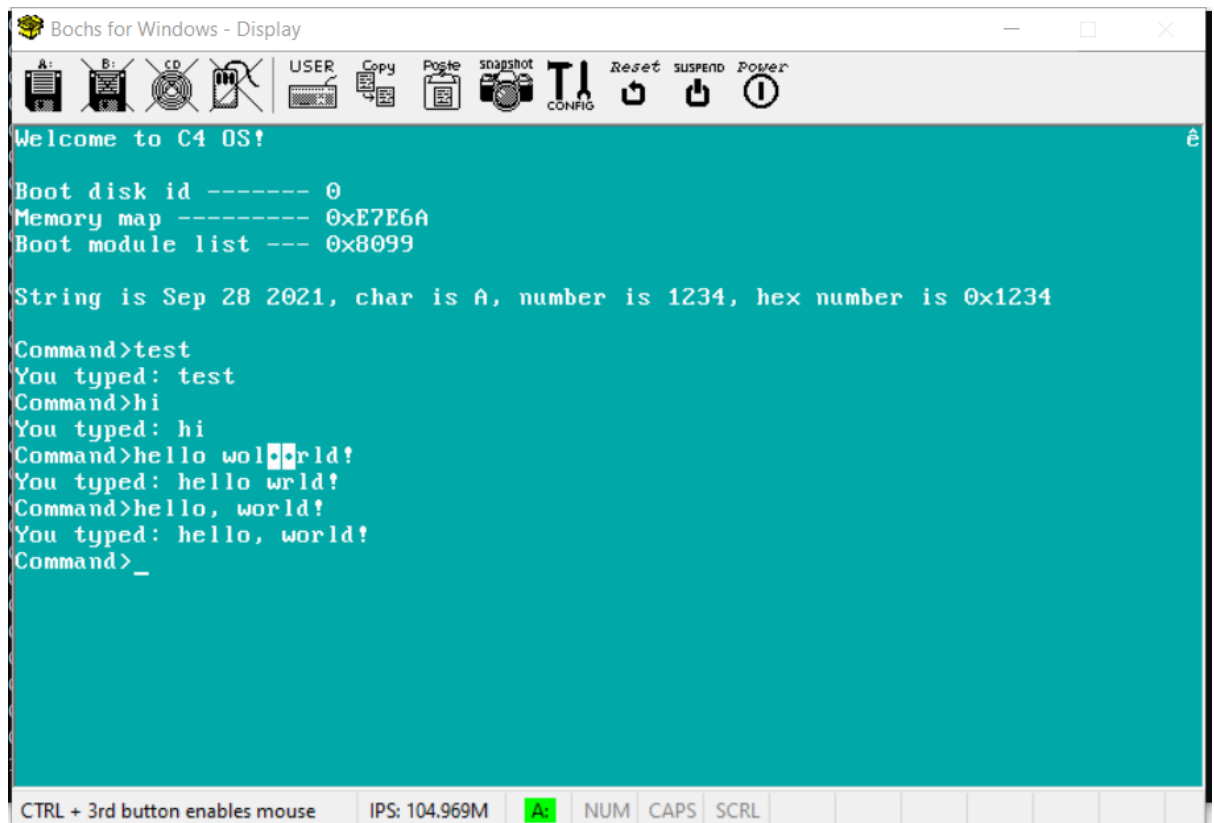
Изменим файл **kernel.c** для обеспечения диалога системы и пользователя добавив в него следующий код.

```

while (true) {
    char buffer[256];
    out_string("Command>");
    in_string(buffer, sizeof(buffer));
    printf("You typed: %s\n", buffer);
}

```

После компиляции и запуска мы должны увидеть что-то вроде этого:



```
Welcome to C4 OS!

Boot disk id ----- 0
Memory map ----- 0xE7E6A
Boot module list --- 0x8099

String is Sep 28 2021, char is A, number is 1234, hex number is 0x1234

Command>test
You typed: test
Command>hi
You typed: hi
Command>hello wolrd!
You typed: hello wrld!
Command>hello, world!
You typed: hello, world!
Command>_

CTRL + 3rd button enables mouse  IPS: 104.969M  A:  NUM  CAPS  SCRL
```

Исходный

код:

<https://github.com/devsienko/c4os/tree/246825246fbd732e4e8e8e8af66e21733d83af8c>.

Менеджер памяти. Карта памяти

До этого момента работа с памятью велась немного хаотично, в теории мы могли выделять память для одних нужд и по ошибке перетирать ее другими данными или оставлять слишком большие пустые промежутки или что-то вроде этого. Для отслеживания состояния памяти, фрагментов которые заняты и свободны мы будем использовать менеджер памяти. Начнем с менеджера физической памяти.

Материал взят с <https://dev64.wordpress.com/2012/06/08/ibm-pc-at-memory-detection/>

Начнем с определения того, сколько и какой памяти нам доступно для работы. Современный способ определения доступной памяти основан на использовании функции BIOS **0x15, EAX = 0xE820**. Этой функции достаточно на современном железе, на старом, могут потребоваться другие функции.

Вкратце, память определяется циклическим обращением к функции 0x15, EAX = 0xE820. В регистр EBX при первом обращении указывается 0, при последующих

обращениях в EBX оставляется то, что возвратило предыдущее обращение к функции. Результат возвращается в виде 24-байтных записей. Адрес для возвращаемых записей передается в ES:DI.

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 возвращенных 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. 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. 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. Пока такую память мы трогать не будем.

Не все BIOS могут поддерживать эту функцию. Если какая-то функция не поддерживается, то при выходе из нее **установлен флаг переполнения (cf=1)** и следует обращаться к более старой.

Карту памяти разместим по адресу **0x7000**. Скорее всего она будет больше пары килобайт. **Последний элемент вручную сделаем типа 0 - такого типа не возвращает BIOS и это и будет признаком конца.**

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.

```
get_memory_map:
    mov di, 0x7000
    xor ebx, ebx
@@:
    mov eax, 0xE820
    mov edx, 0x534D4150
    mov ecx, 24
    mov dword[di + 20], 1 ; set
    int 0x15
    jc @f ; jump if carry (cf=1), it means 0x15 0xE820 function isn't supported
    add di, 24
    test ebx, ebx
    jnz @b
@@:
    ; ...
```

Set EDX to the magic number 0x534D4150 - вот это лучшее что я нашел спустя 15 минут поиска по интернету по поводу того, что же значит этот магический номер. Также я узнал что это ASCII код строки 'SMAP'. Думаю, что сейчас эти знания погоды не сделают и можно двигаться дальше.

На старых компьютерах прерывания 0x15, AX=0xE820 может не быть, тогда нужно воспользоваться более старым сервисами. 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. Использовать её нам смысла нет - если процессор поддерживает защищённый режим, то вряд ли у него будет меньше нескольких мегабайт памяти.

Далее 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).

```
mov ah, 0x88
int 0x15
; ...
```

Когда и 16 МБ стало недостаточно появилась новая функция - прерывание 0x15, AX=0xE801. Она возвращает результаты в 4 регистрах:

AX - размер расширенной памяти до 16 МБ в килобайтах

BX - размер расширенной памяти сверх 16 МБ к блокам 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 производителя BIOS судя по всему не договорились, поэтому надо просто, если в AX и BX нули, брать значение из CX и DX.

```
mov ax, 0xE801
int 0x15
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
; eax contains size of memory ...
```

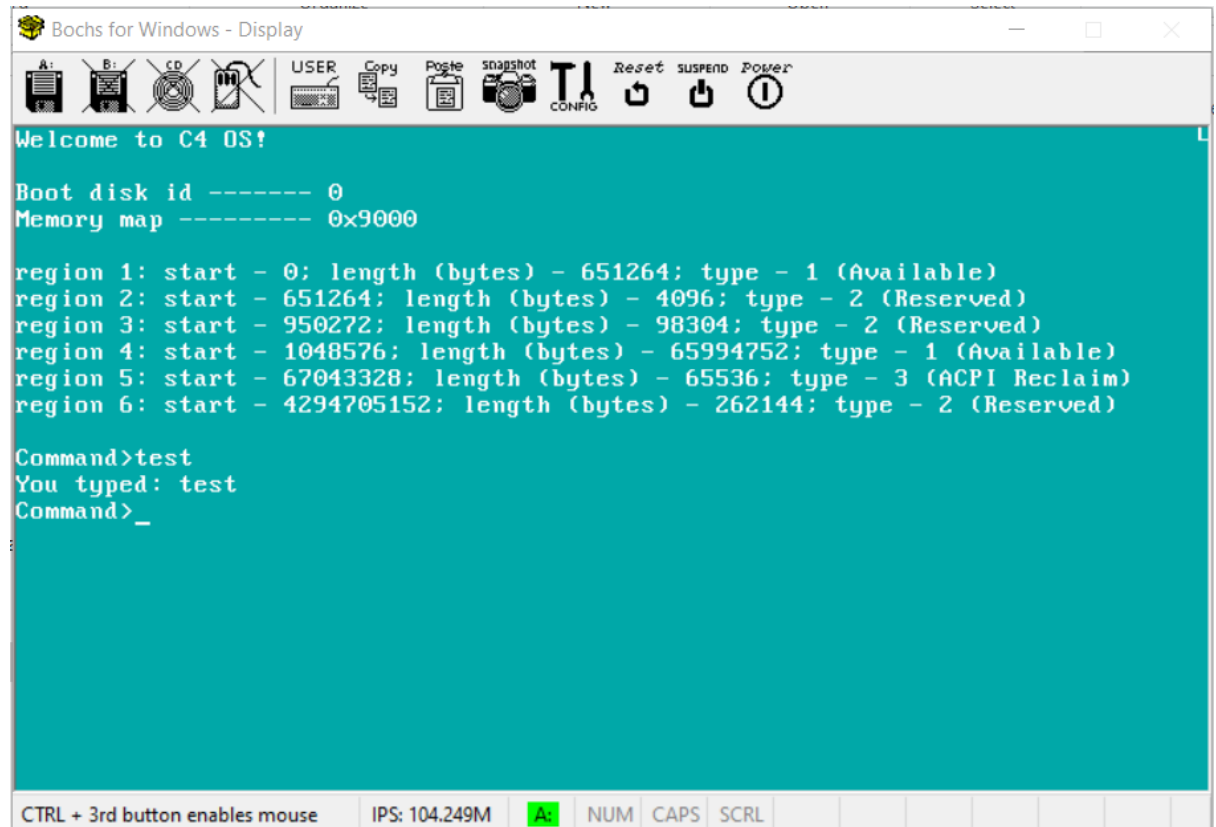
В нашем коде мы будем использовать только современный метод, основанный на прерывании 0x15, AX=0xE820.

Исходный

код:

<https://github.com/devsienko/c4os/tree/cfd7102a5b780284016e6b1f3ef5b674f0f47e78>.

Скомпилировав и запустив код мы увидим что-то вроде следующего (зависит от конфигурации вашей памяти):



```
Bochs for Windows - Display
Welcome to C4 OS!
Boot disk id ----- 0
Memory map ----- 0x9000

region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

Command>test
You typed: test
Command>_

CTRL + 3rd button enables mouse  IPS: 104.249M  A:  NUM  CAPS  SCRL
```

Хотелось бы отметить разбиение вывода **printf** в коде на несколько строк. Кажется в эту функцию прокрался баг, поэтому если выводить все одной строкой получается некорректная информация.

Исходный код исправленной (пишу это слово скрепя сердце) версии:

<https://github.com/devsienko/c4os/tree/32a5d8819ed166ff846f2242f124d1d3b5cd931f>.

Дело было в том, что **printf** для спецификатора **%x**, **%d** и прочих ожидает число типа **size_t**, а мы передавали туда **unsigned long long (uint64)**, поэтому я сделал версию для поддержки **unsigned long** (более чем достаточно для хранения информации о фрагментах памяти и нет проблем с линковкой типов с большим размером).

Менеджер памяти

Тему управления памятью можно разделить на 2 подпункта. 1 - управлению физической памятью и 2 - управление виртуальной памятью.

Для управления физической памятью нам нужно как-то помечать занятые и свободные участки, при запросах памяти отдавать ссылку на свободные участки после чего помечать их как занятые. При освобождении используемой памяти обратно помечать участки как свободные и готовые к использованию.

Создадим заголовочный файл *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_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 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 небольших 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;
    }

    unsigned int get_page_directory_index (void *vaddr) {
        char page_directory_shift = 22;
        unsigned int result = ((size_t)vaddr >> page_directory_shift) &
PAGE_TABLE_INDEX_MASK;
        return result;
    }

```

```

unsigned int get_page_table_index (void * vaddr) {
    char page_table_shift = 12;
    unsigned int result = ((size_t)vaddr >> page_table_shift) &
PAGE_TABLE_INDEX_MASK;
    return result;
}

phyaddr get_page_info(phyaddr page_dir, void *vaddr) {
    unsigned int page_directory_index = get_page_directory_index(vaddr);
    temp_map_page(page_dir);
    phyaddr pde = ((phyaddr*)TEMP_PAGE)[page_directory_index];
    if (!(pde & PAGE_VALID)) {
        return 0;
    }

    unsigned int page_table_index = get_page_table_index(vaddr);
    phyaddr page_table_base = pde & ~PAGE_OFFSET_MASK;
    temp_map_page(page_table_base);
    phyaddr pte = ((phyaddr*)TEMP_PAGE)[page_table_index];
    return pte;
}

```

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).

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");
    }

```

Добавим код инициализации в kernel_main (менеджер памяти следует инициализировать самым первым, до всех остальных подсистем ядра):

```

#include "tty.h"
#include "stdlib.h"
#include "memory_manager.h"
#include "interrupts.h"

void kernel_main(uint8 boot_disk_id, void *memory_map) {
    init_memory_manager(memory_map);
    init_interrupts();
    init_tty();
    set_text_attr(63);
    clear_screen();

    printf("Welcome to SUN OS!\n\n");
}

```

```

printf("Boot disk id ----- %d\n", boot_disk_id);
printf("Memory map ----- 0x%x\n\n", memory_map);

display_memory_map(memory_map);

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\n",
get_page_info(kernel_page_dir, (void*)0xB8000));

while (true) {
    char buffer[256];
    out_string("Command>");
    in_string(buffer, sizeof(buffer));
    printf("You typed: %s\n", buffer);
}
}

bool is_last_memory_map_entry(struct memory_map_entry *entry);

void display_memory_map(void *memory_map) {
    char* memory_types[] = {
        {"Available"},           //memory_region.type==0
        {"Reserved"},           //memory_region.type==1
        {"ACPI Reclaim"},        //memory_region.type==2
        {"ACPI NVS Memory"}      //memory_region.type==3
    };

    struct memory_map_entry *entry = memory_map;
    int map_entry_size = 24; //in bytes
    int region_number = 1;
    while(true) {
        if(is_last_memory_map_entry(entry))
            break;

        printf("region %d: start - %u; length (bytes) - %u; type - %d (%s)\n",
            region_number,
            (unsigned long)entry->base,
            (unsigned long)entry->length,
            entry->type,
            memory_types[entry->type-1]);
        entry++;
        region_number++;
    }
}

```

```

        printf("\n");
    }

bool is_last_memory_map_entry(struct memory_map_entry *entry) {
    bool result = entry->length == 0
        && entry->length == 0
        && entry->length == 0
        && entry->length == 0;

    return result;
}

```

После компиляции и запуска мы увидим примерно следующее:

```

Welcome to C4 OS!
Boot disk id ----- 0
Memory map ----- 0x9000

region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8063

Command>_

```

Исходный код:

<https://github.com/devsienko/c4os/tree/44b5a2323fc25ea337d84516954ea785c9abe6b9>.

Менеджер физической памяти

Для целей отслеживания свободной физической памяти мы будем использовать **двунаправленный связанный список, располагающийся на свободных страницах**

памяти. Этот вариант не самый быстрый из возможных, зато обладает достаточной гибкостью и простотой, а также не требует дополнительной памяти для хранения служебных структур данных.

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 пометить блок как свободный, но и дефрагментировать блоки памяти - если перед и/или после свободного блока памяти вплоты существует ещё один их следует объединить. Для упрощения задачи можно поддерживать упорядоченность блоков по возрастанию адресов, тогда сложность **free_phys_pages** будет также O(N).

Физический адрес первого свободного 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) в файле *memory_manager.c* и реализацию функции *alloc_phys_pages*:

```
...
typedef struct {
    phyaddr next;
    phyaddr prev;
    size_t size; // pages count
} PhysMemoryBlock;
...

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);
            }
        } while (next != cur_block);
        result = cur_block;
        free_phys_memory_pointer = next;
    }
    return result;
}
```

```

        ((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 длины), 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 окружен двумя. Следует удалить следующий 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;
}

```

После реализации этих функций изменим код выделения 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");
}

```

Исходный код:

<https://github.com/devsienko/c4os/tree/0af58ee95a726615c02d926efb344acee446f504>.

Защита памяти ядра

Прежде чем перейти к реализации менеджера виртуальной памяти настроим атрибуты памяти ядра, запретим запись в область кода и выставим 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

```

Изменим 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);
}
```

PAGE_PRESENT - Страница присутствует в оперативной памяти, т.е. спроецирована. 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.

Буфер ассоциативной трансляции (англ. translation lookaside buffer, **TLB**) — это специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти.

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.

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.

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.

Исходный код:

<https://github.com/devsienko/c4os/tree/e0325141c5325fd832fb0392861d057d6f3b5ec2>.

Менеджер виртуальной памяти

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 будем использовать список занятых, организованный в виде массива, который способен расширяться по мере необходимости.

Для начала добавим несколько определений в коде:

```
typedef struct {
    phyaddr page_dir;
    void *start;
    void *end;
    size_t block_table_size;
    size_t block_count;
    VirtMemoryBlock *blocks;
} 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);
```

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* ($as \rightarrow start \leq vaddr < as \rightarrow end$) 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 параметру `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.

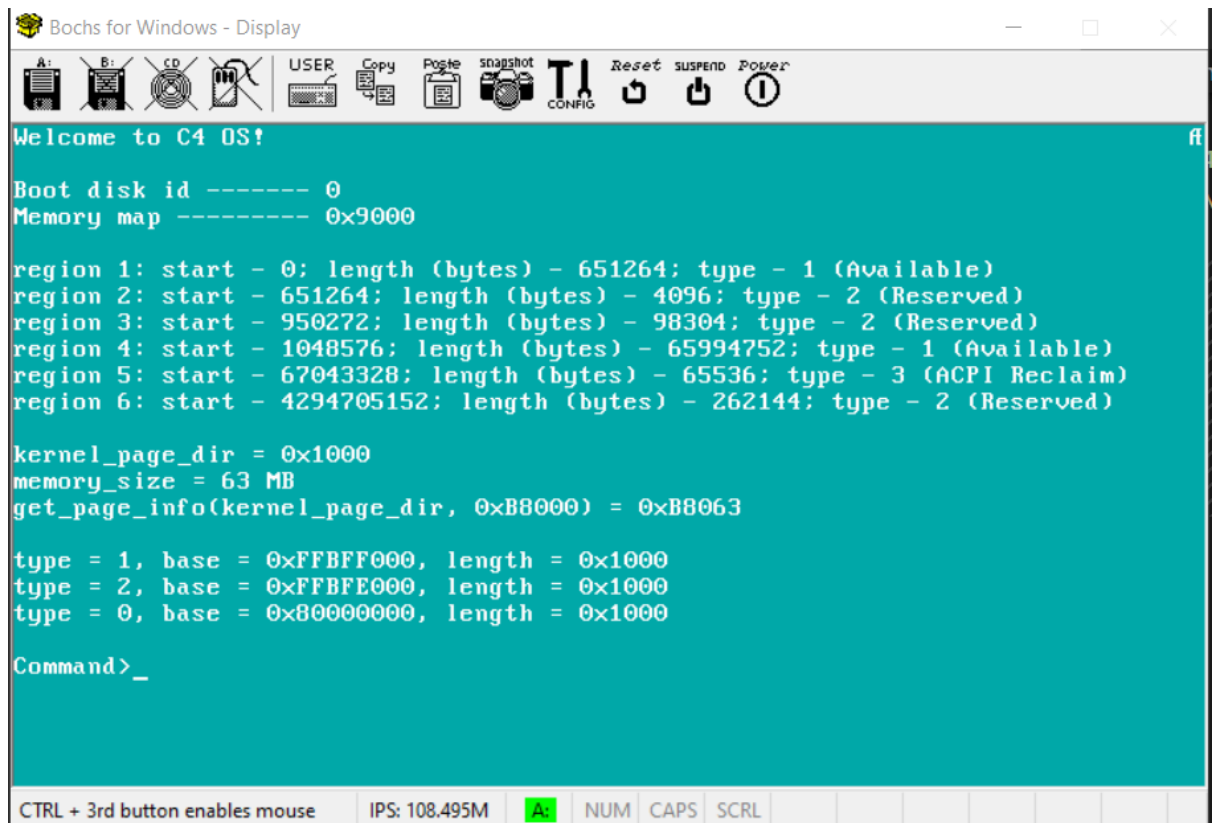
После реализации этих функций у нас появится относительно полноценный менеджер памяти (не хватает только реализации подкачки памяти и выделения блоков произвольного размера, а не только кратного размеру страницы, однако первое нам пока делать рано, а второе можно переложить на плечи прикладных программ, если писать микроядро).

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).

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 `init_memory_manager`, `init_interrupts` и `init_tty` проверка успешности выделения оперативной памяти не производится, потому что вряд ли нашу ОС запустят на компьютере с объёмом памяти менее двух мегабайт. 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).



Исходный

код:

<https://github.com/devsienko/c4os/tree/b1d6483337282f961fffc07f39d44bbbac3ed8eb>.

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. Рассмотрим эту 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 divide by zero\n");
    }
}

```

Функции thread1 и thread2 выполняются параллельно. Переменная variable доступна обоим **нителям** (параллельные задачи в рамках одного адресного пространства). Допустим, первой получает управление нить thread2. Она производит сравнение variable с нулём (далее будет деление на это значение, а на ноль делить нельзя). Сравнение проходит успешно ($10 \neq 0$) 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 divide 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.

Исходный

код:

<https://github.com/devsienko/c4os/tree/f779584fa6b5dbd4b98e8e603c745fc8f3f6f755>.

Улучшение обработки IRQ прерываний и многозадачности

До этого момента 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 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
}

; common interrupt handler
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

; interrupt handlers table
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 - записать 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 беспокоиться, 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.

Модифицируем interrupts.c:

```
#include "stdlib.h"
#include "memory_manager.h"
#include "interrupts.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:
            // timer int handler:
            (*((char*)(0xB8000 + 79 * 2)))++;
            break;
        case 1:
            keyboard_interrupt();
            break;
    }
}

```

Теперь обработчик в interrupts.c решает кому обрабатывать какие прерывания. В нём, например, задано, что IRQ0 это должно уйти системе многозадачности (в будущем, сейчас там по таймеру только меняется верхний правый символ экрана), а IRQ1 в обработчик драйвера клавиатуры.

Далее в interrupts.h удалим макрос IRQ_HANDLER и добавим следующий код:

```

#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;

```

Оставшиеся изменения можно посмотреть в исходном коде:

<https://github.com/devsienko/c4os/tree/01d2c02c4c290f53e48a865b6b105fe1f542ff51>.

DMA

Перед тем как приступить к теме многозадачности и ее реализации рассмотрим тему работы с DMA контроллером и Floppy контроллером. Это вспомогательная тема, которая будет нам полезна для загрузки исполняемых файлов в память.

Итак. Предположим, у нас есть необходимость считать достаточно большой объем данных с флоппи-диска в память. Если перемещать данные побайтно, то эта операция будет занимать процессорное время, а если параллельно необходимо выполнять еще несколько подобных операций, то нагрузка на процессор может оказаться существенной. Если посмотреть на ситуацию сверху, то процессор может вообще показаться лишним звеном между флоппи-диском и памятью. Как следствие такой идеи появилась технология DMA, или Direct Memory Access – технология **прямого доступа к памяти (ПДП)**, минуя центральный процессор.

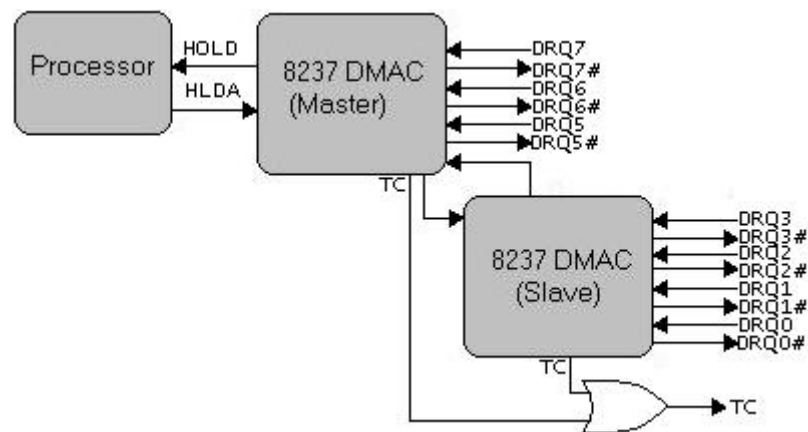
Для того, чтобы указать откуда, куда и сколько данных нужно переместить используется контроллер прямого доступа к памяти **Direct Memory Access Controller (DMAC)**. В современные ПК устанавливаются два контроллера DMA (каскад по аналогии с контроллером прерываний):

- 8-битный (каналы 0, 1, 2, 3).
- 16-битный (каналы 4, 5, 6, 7).

В данном случае разрядность контроллера указывает режим передачи данных (8-битный, 16-битный).

Микросхема каждого контроллера DMA содержит четыре независимых друг от друга канала. Среди этих восьми каналов наиболее часто используются:

- Канал 2 - для работы с гибкими дисками (FDC).
- Канал 3 - для работы с жёсткими дисками.
- Канал 4 - для каскадирования двух контроллеров **DMA** (к 4 каналу контроллера **DMA-2** подключается контроллер **DMA-1**).



Каждый канал имеет свой набор регистров, доступ к которым осуществляется через порты ввода-вывода.

Общий алгоритм ПДП.

Для осуществления прямого доступа к памяти контроллер должен выполнить ряд последовательных операций:

- принять запрос (DREQ) от устройства ввода-вывода;
- сформировать запрос (HRQ) в процессор на захват шины;
- принять сигнал (HLDA), подтверждающий захват шины;
- сформировать сигнал (DACK), сообщающий устройству о начале обмена данными;
- выдать адрес ячейки памяти, предназначенной для обмена;
- выработать сигналы (MEMR, IOW или MEMW, IOR), обеспечивающие управление обменом;
- по окончании цикла DMA либо повторить цикл DMA, изменив адрес, либо прекратить цикл.

DMAC соединен с CPU для прямой коммуникации (в оба направления). Соединен DMAC через пины **HACK** и **HREQ**. Пин **HACK (Hold Acknowledge)** включен когда CPU дает контроллеру DMAC полный контроль над системной шиной. Высокий уровень сигнала на этом пине дает контроллеру DMAC знать, когда можно безопасно передавать данные контроллеру памяти. Как следствие невозможен одновременный доступ со стороны CPU и DMAC к системной шине, только кто-то один может ее использовать в данный момент времени.

Работы с контроллером DMA

Работу с контроллером DMA я предлагаю начать с нижеприведенной функции. Эта функция занимается инициализацией DMAC для считывания блока данных с флорпи диска.

```
bool dma_initialize_floppy(uint8* buffer, unsigned length) {
    //todo: calc buffer address and extended page address register

    union {
        uint8 byte[4]; //Lo[0], Mid[1], Hi[2]
        unsigned long l;
    } a, c;

    a.l = (unsigned) buffer;
    c.l = (unsigned) length - 1;

    //Check for buffer issues
    if ((a.l >> 24) || (c.l >> 16) || (((a.l & 0xffff) + c.l) >> 16)) {
        return false;
    }

    dma_reset(1);
    dma_mask_channel(FDC_DMA_CHANNEL); //Mask channel 2

    dma_reset_flipflop(FDC_DMA_CHANNEL); //Flipflop reset on DMA 1

    dma_set_read(FDC_DMA_CHANNEL); //set the DMA for read transfer

    dma_set_address(FDC_DMA_CHANNEL, a.byte[0], a.byte[1]); //Buffer address

    dma_set_external_page_register(2, a.byte[2]); //Page number

    dma_set_count(FDC_DMA_CHANNEL, c.byte[0], c.byte[1]); //Set count

    dma_unmask_all(1); //Unmask channel 2

    return true;
}
```

Функция **dma_reset** вызывает сброс контроллера. Для дальнейшего использования контроллер должен быть заново проинициализирован.

ISA DMAC Reset Ports	
Port	Description
0x0D	DMAC 0 (16 bit) Slave (write)
0xD8	DMAC 1 (8-bit) Master (write)

Связка функций **dma_mask_channel/dma_unmask_all**. Во время настройки канала контроллера необходимо игнорировать запросы DREQ, для этого сперва делают его маскировку, а затем, когда контроллер готов к работе размаскировывают его.

Регистр маскирования канала CMR (Channel Mask Register) позволяет временно блокировать сигнал запроса на обслуживание (DREQ) для определённого канала. Содержимое регистра представляется следующим образом:

- биты 7-3 - не используются (должны быть равны нулю).
- бит 2 - запрос на **DMA** (0 - сбросить, 1 - установить).
- биты 1-0 - номер канала.

DMA-1	DMA-2	Описание
0x0e	0xdc	сброс регистра маски контроллера

Функция **dma_reset_flipflop**, необходимая для сброса триггера байтов. Это нужно для чтения или записи 16-битных значений из/в 8-битные порты 0x00-0x08. То есть триггер используется только с 8-битным DMAC при работе с 16-битными данными. Если бы мы работали с 16-битным DMAC, нам не нужно было бы его вызывать. Когда вы сбрасываете триггер, вы сообщаете DMAC, что следующим байтом данных будет младший байт, а следующий за ним - старшим. Если триггер не находится в положении по умолчанию, будет выбран старший байт.

DMAC 0 Port (Slave)	DMAC 1 Port (Master)	Descripton
0x0C	0xD8	Clear Byte Pointer Flip-Flop (Write)

Функция **dma_set_address** устанавливает адрес буфера, куда **DMAC** следует поместить данные или откуда надо взять данные. Адрес можно установить двумя способами. Первый способ подразумевает использование только 16-битного регистра адреса (порт 0x4), то есть мы имеем ограничение, мы можем адресовать первые 64Кб памяти. Мы загружаем младший бит, а затем старший бит в через порт 0x4. Второй способ - в дополнение к регистру адреса мы можем использовать 8-битный регистр смещения (Page Registers) и тогда, имея в распоряжении 24 бита, мы можем

оперировать адресами до 16Мб. В этом случае адрес необходимо представить в виде номера страницы и смещения. Например, для адреса 23456 номер страницы - 2, смещение - 3456. То есть размер страницы в данном случае 64Кб. Программа должна сначала вывести младший байт смещения в порт с адресом 0x4, затем вывести в этот же порт старший байт смещения и, наконец, вывести байт номера страницы в порт с адресом 0x81.

Канал DMA	Адрес порта регистра страниц
0	087h
1	083h
2	081h
3	082h
4	-
5	08Bh
6	089h
7	08Ah

Функция **dma_set_count** аналогично функции **dma_set_address** устанавливает длину передаваемых данных.

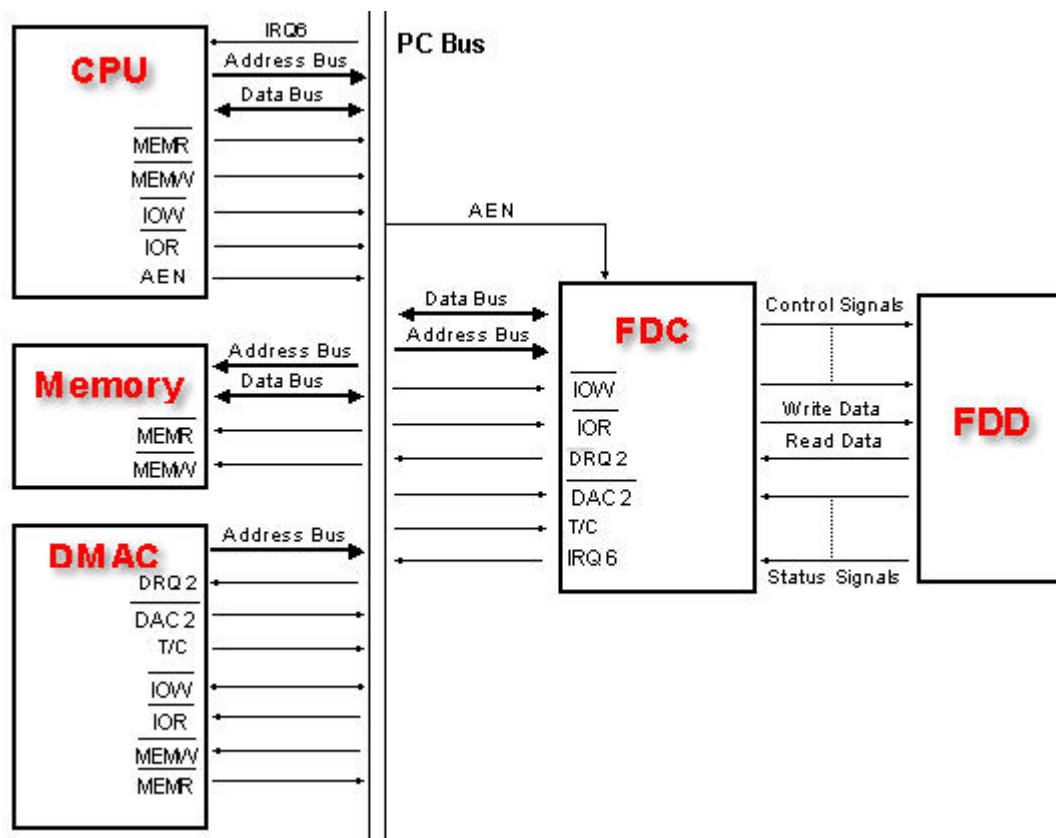
DMAC 0 Port (Slave)	Descripton
0x4	Channel 2 Address
0x5	Channel 2 Counter

Остается рассмотреть функцию **dma_set_read**. Эта функция указывает, что мы собираемся использовать канал 2, собираемся использовать его для чтения в режиме Single Transfer. Для этого мы устанавливаем соответствующие биты регистра Mode Register через порт **0x0B**.

DMAC 0 Port (Slave)	DMAC 1 Port (Master)	Descripton
0x0B	0xD6	Mode Register (Write)

Работа с флоппи-диском

Все взаимодействия с флоппи-диском (**Floppy Disk Drive, FDD**) происходит через связанный с FDD контроллер флоппи-диска (**Floppy Disk Controller, FDC**), который в свою очередь связан с **DMAC**.



Существует несколько режимов работы FDC. Мы будем использовать DMA mode, режим, при котором данные передаются с использованием DMA/DMAC.

Управление FDC производится через взаимодействия с его регистрами через порты ввода вывода и через отправку команд, также через порты ввода вывода.

Регистры FDC

Digital Output Register (DOR)

Это регистр предназначенный только для записи. С его помощью можно управлять различными функциями FDC, такими как двигатель FDD, режим работы (DMA и IRQ), сброс и привод. Имеет формат:

- **Bits 0-1** DR1, DR2
 - 00 - Drive 0
 - 01 - Drive 1
 - 10 - Drive 2
 - 11 - Drive 3
- **Bit 2** REST
 - 0 - Reset controller

- 1 - Controller enabled
- **Bit 3** Mode
 - 0 - IRQ channel
 - 1 - DMA mode
- **Bits 4 - 7** Motor Control (Drives 0 - 3)
 - 0 - Stop Motor for drive
 - 1 - Start Motor for drive

This is an easy one! Basically when sending a command to control the functionality of the FDC, just build up a bit pattern to select what drive this is for (Remember that a single FDC can communicate with four FDD's!), the controller reset status, mode of operation (Remember that the FDC can operate in both DMA and IRQ modes?) and the status of that particular FDD internal motor.

Here is an example. Lets say we want to start up the motor for the first floppy drive (FDD 0). **Starting the motor for the FDD is needed before performing any read or write operations to it!** To start it, just set the bit (4-7) that corresponds to the drive you want to start or stop the motor. Keeping all other bits at 0 will be a normal operation (IRQ mode, reset controller.) Knowing that the DOR is mapped to the processors i/o address space at port 0x3f2, this becomes very simple. First, we will create bit masks for the register to increase readability. Remember that all of this code is also in the demo at the end of this tutorial.

```
enum FLPYDSK_DOR_MASK {
    FLPYDSK_DOR_MASK_DRIVE0          = 0,      //00000000
    = here for completeness sake
    FLPYDSK_DOR_MASK_DRIVE1          = 1,      //00000001
    FLPYDSK_DOR_MASK_DRIVE2          = 2,      //00000010
    FLPYDSK_DOR_MASK_DRIVE3          = 3,      //00000011
    FLPYDSK_DOR_MASK_RESET            = 4,      //00000100
    FLPYDSK_DOR_MASK_DMA              = 8,      //00001000
    FLPYDSK_DOR_MASK_DRIVE0_MOTOR     = 16,     //00010000
    FLPYDSK_DOR_MASK_DRIVE1_MOTOR     = 32,     //00100000
    FLPYDSK_DOR_MASK_DRIVE2_MOTOR     = 64,     //01000000
    FLPYDSK_DOR_MASK_DRIVE3_MOTOR     = 128,    //10000000
};
```

Using the above bit masks, we can just bitwise OR the different bits that we would like to set. So, to start the motor for floppy drive 0:

```
outportb (FLPYDSK_DOR, FLPYDSK_DOR_MASK_DRIVE0_MOTOR |
FLPYDSK_DOR_MASK_RESET);
```

Remember that FLPYDSK_DOR was defined earlier as 0x3f2, which is the i/o address of the DOR FDC register. The above also resets the controller.

To turn this same motor off, just send the same command but without the motor bit set:

```
outportb (FLPYDSK_DOR, FLPYDSK_DOR_MASK_RESET);
```

Warning: Give the motor some time to start up! Remember that the internal FDD motor is mechanical. Like all mechanical devices, they tend to be slower than the speed of the running software. Because of this, whenever starting up a FDD motor, always give it a little time to spin up before attempting to read or write to it.

The DOR is a write only register. To enforce this, let's create a routine for it:

```
void flpydisk_write_dor (uint8_t val ) {

    /*! write the digital output register

    outportb (FLPYDSK_DOR, val);

}
```

Main Status Register (MSR)

The **Main Status Register (MSR)** follows a *gasp!* specific bit format! Bet you did not see that one coming! Okay, okay, let's get back on track here (pun intended). Here is the format of the MSR:

- **Bit 0** - FDD 0: 1 if FDD is busy in seek mode
- **Bit 1** - FDD 1: 1 if FDD is busy in seek mode
- **Bit 2** - FDD 2: 1 if FDD is busy in seek mode
- **Bit 3** - FDD 3: 1 if FDD is busy in seek mode

- 0: The selected FDD is not busy
- 1: The selected FDD is busy
- **Bit 4** - FDC Busy; Read or Write command in progress
 - 0: Not busy
 - 1: Busy
- **Bit 5** - FDC in Non DMA mode
 - 0: FDC in DMA mode
 - 1: FDC not in DMA mode
- **Bit 6** - DIO: direction of data transfer between the FDC IC and the CPU
 - 0: FDC expecting data from CPU
 - 1: FDC has data for CPU
- **Bit 7** - RQM: Data register is ready for data transfer
 - 0: Data register not ready
 - 1: Data register ready

This MSR is a simple one. It contains the current status information for the FDC and disk drives. Before sending a command or reading from the FDD, we will need to always check the current status of the FDC to insure it is ready.

Here is an example of reading from this MSR to see if its busy. We first define the bit masks that will be used in the code. Notice how it follows the format shown above.

```
enum FLPYDSK_MSR_MASK {

    FLPYDSK_MSR_MASK_DRIVE1_POS_MODE    =    1,    //00000001

    FLPYDSK_MSR_MASK_DRIVE2_POS_MODE    =    2,    //00000010

    FLPYDSK_MSR_MASK_DRIVE3_POS_MODE    =    4,    //00000100

    FLPYDSK_MSR_MASK_DRIVE4_POS_MODE    =    8,    //00001000

    FLPYDSK_MSR_MASK_BUSY                =   16,    //00010000

    FLPYDSK_MSR_MASK_DMA                 =   32,    //00100000

    FLPYDSK_MSR_MASK_DATAIO              =   64,    //01000000

    FLPYDSK_MSR_MASK_DATAREG             =  128    //10000000

};
```

Easy, huh? So lets test if the FDC is busy (BUSY flag is set.) Knowing that FLPYDSR_MSR is 0x3f4, the i/o port address for the MSR, all we need to do is this:

```
if ( inportb (FLPYDSK_MSR) & FLPYDSK_MSR_MASK_BUSY )  
  
    //! FDC is busy
```

When sending a read or write command, all we need to do is wait until this bit is 0. Cool, huh?

To make readability easier, I decided to hide this in a routine so here it is. This routine just returns the status of the FDC.

```
uint8_t flpydisk_read_status () {  
  
    //! just return main status register  
  
    return inportb (FLPYDSK_MSR);  
  
}
```

Data Register

This is a 8 or 16 bit read/write register. The actual size of the register is specific on the type of controller. **All command paramaters and disk data transfers are read to and written from the data register.** This register does not follow a specific bit format and is used for generic data. It is accessed through I/O port 0x3f5 (FDC 0) or 0x375 (FDC 1).

Note: Before reading or writing this register, you should always insure it is valid by first reading its status in the Master Status Register (MSR).

Remember: All command bytes and command paramaters are sent to the FDC through this register! You will see examples of this in the command section below, so dont worry to much about it yet.

If an invalid command was issued, the value returned from the data register is 0x80.

Configuration Control Register (CCR)

In PC/AT Mode, this register is known as the **Data Rate Select Register (DSR)** and only has the first two bits set (Bit 0=DRATE SEL0, Bit 1=DRATE SEL1.) This was listed in a table in the DSR register section. Lets take another look...

- **00** 500 Kbps
- **10** 250 Kbps
- **01** 300 Kbps
- **11** 1 Mbps

Bit 2 is NOPREC in Model 30/CCR modes and has no function. Other bits are undefined and may change depending on controller.

Like the other registers, I created a routine so we can write to this register.

Отправка команд

Выполнение команды разделяется на три фазы - командная фаза, фаза выполнения, фаза результата. В командной фазе программа должна передать контроллеру всю информацию, необходимую для выполнения команды. В фазе выполнения команда выполняется, и в фазе результата программа получает от контроллера информацию о состоянии контроллера.

Информация, необходимая для выполнения команды, передается контроллеру через порт данных **0x3F5** (Data Register). В соответствии с форматом команды программа должна последовательно вывести в этот порт код команды и все параметры. Например:

```
outportb(FLPYDSK_FIFO, FDC_CMD_SPECIFY);  
  
outportb(FLPYDSK_FIFO, steprate_headunload);  
  
outportb(FLPYDSK_FIFO, headload_ndma);
```

В данном случае коде мы запускаем команду Specify. Мы записываем последовательно 3 байта данных в порт ввода/вывода **0x3F5**. Первый байт - байт команды, оставшиеся байты - входные параметры команды. У некоторых команд также могут быть выходные параметры, в этом случае мы также последовательно будем считывать их из того же самого порта **0x3F5** с помощью команды макроса **inportb**.

Each command can be 1 to 9 bytes in size. The FDC knows how many bytes to expect from the first command byte. That is, the first byte is the actual command that tells the FDC what we want it to do. The FDC knows how many more bytes to expect from this command (The command parameters.)

Commands will only operate on a single head of the track. If you want to operate on both heads, you need to set the Multiple Track Bit. A lot of these commands follow bit formats (Will be shown below). This is where things get complicated.

A command byte only uses the low 4 bits of the byte for the actual command (can be more.) The high bits of these command bytes are for optional settings for the command. I call these extended command bits but it does not have an official name. There is a couple of these bits that are common for a lot of the commands that we will need to use. We will look at these bits in the command byte later.

Warning: Before sending a command or parameter byte, insure the data register is ready to receive data by testing bit 7 of the Main Status Register (MSR) first.

Extended Command Bits

Some of these commands require you to pass several bytes before the command is executed. Others return several bytes. To make things easier to read, I have listed all of the commands, formats, and parameter bytes in tables. Each command comes with an explanation and an example routine.

Okay, now remember when we mentioned extended command bits and how the commands above are only four bits? The upper four bits can be used for different things and purposes.

When describing the format of a command, we represent an extended bit with a character (like M or F.) For example, the Write Sector command has the format M F 0 0 0 1 1 0, where the first four bits (0 1 1 0) are the command byte and the top four bits, M

F 0 0 represent different settings. M is set for multitrack, F to select what density mode to operate in for the command.

Here is a list of common bits:

- M - MultiTrack Operation
 - 0: Operate on one track of the cylinder
 - 1: Operate on both tracks of the cylinder
- F - FM/MFM Mode Setting
 - 0: Operate in FM (Single Density) mode
 - 1: Operate in MFM (Double Density) mode
- S - Skip Mode Setting
 - 0: Do not skip deleted data address marks
 - 1: Skip deleted data address marks
- HD - Head Number
- DR0 - DR1 - Drive Number Bits (2 bits for up to 4 drives)

The M, F, and S bits are very common to a lot of the commands, so I decided to stick them in a nice enumeration. To set them, just bitwise OR these settings with the command that you would like to use.

```
enum FLPYDSK_CMD_EXT {  
  
    FDC_CMD_EXT_SKIP = 0x20, //00100000  
  
    FDC_CMD_EXT_DENSITY = 0x40, //01000000  
  
    FDC_CMD_EXT_MULTITRACK = 0x80 //10000000  
  
};
```

Команды, которым нам понадобятся:

This command is used to check information on the state of the FDC when an interrupt returns.

Команда Чтение состояния прерывания (SENCE INTERRUPT STATUS), формат которой показан на рис. 10.18, считывает содержимое регистра ST0 и значение текущего номера цилиндра PCN (причину возникновения прерывания можно определить по значению кода IC и флага SE регистра ST0). После выполнения команды сигнал прерывания сбрасывается.

	7	6	5	4	3	2	1	0
Фаза команды	0	0	0	0	1	0	0	0
Фаза выдачи результатов	ST0				PCN			

Рис. 10.18. Формат команды Чтение состояния прерывания

Контроллер дисководов формирует запрос прерывания в следующих случаях:

- ◆ в момент перехода из фазы выполнения в фазу выдачи результата при выполнении команд Чтение данных, Чтение удаленных данных, Чтение дорожки, Запись данных, Запись удаленных данных, Верификация, Чтение идентификатора, Форматирование дорожки;
- ◆ по окончании выполнения команд поиска и рекалибровки;
- ◆ при выполнении передачи данных в не-DMA режиме;
- ◆ при изменении линии состояния готовности накопителя (только в устаревших моделях контроллеров).

Поступление команды Чтение состояния прерывания при отсутствии запроса прерывания приводит к установке в регистре ST0 признака неверного кода команды (поле IC принимает значение 10h, а остальные разряды сбрасываются, то есть в регистре ST0 будет находиться значение 80h).

Команда Чтение состояния прерывания в обязательном порядке применяется после выполнения команд поиска и рекалибровки, чтобы сбросить сигнал прерывания и определить результат выполнения команды.

Команда Установка параметров (SPECIFY), формат которой показан на рис. 10.20, позволяет установить значения временных интервалов трех встроенных таймеров контроллера. Поле HLT задает интервал времени, в течение которого контроллер ожидает «загрузки» головки, прежде чем начать операцию записи или считывания. Поле HUT определяет временной интервал, в течение которого контроллер удерживает головку дисковод в «загруженном» состоянии после завершения фазы выполнения команды записи или считывания. Поле SRT задает временной интервал между «шаговыми» пульсациями, выдаваемыми контроллером приводу головок. Флаг ND позволяет запретить использование контроллером режима DMA (0 — режим DMA разрешен, 1 — режим DMA заблокирован).

	7	6	5	4	3	2	1	0
Фаза посылки команды	0	0	0	0	0	0	1	1
	SRT				HUT			
	HLT							ND

Рис. 10.20. Формат команды Установка параметров

По команде Рекалибровка (RECALIBRATE), формат которой показан на рис. 10.16, контроллер выполняет возврат головок дисковод на нулевую дорожку. Возврат головок выполняется путем подачи на привод головок дисковод сигнала Сделай шаг назад до тех пор, пока не сработает датчик установки головок на нулевую дорожку. После поступления сигнала от датчика нулевой дорожки контроллер устанавливает равным 1 значение разрядов SE (Поиск завершен) регистра статуса ST0 и завершает выполнение команды. Если после выполнения 79 шагов сигнал

228

10. Контроллер накопителей на гибких магнитных дисках (FDC)

от датчика так и не поступил, то контроллер завершает выполнение команды, предварительно устанавливая равным 1 значение разрядов SE (Поиск завершен) и EC (Сбой оборудования) регистра ST0.

	7	6	5	4	3	2	1	0
Фаза посылки команды	0	0	0	0	0	1	1	1
	0	0	0	0	0	0	DS1	DS0
Фаза выполнения	—							

Рис. 10.16. Формат команды Рекалибровка

ВНИМАНИЕ

При использовании дисков нестандартного формата, имеющих более 80 цилиндров, может потребоваться повторное выполнение команды Рекалибровка для возврата головок на нулевую дорожку.

При выполнении рекалибровки в течение фазы отправки команды контроллер находится в состоянии «Занят», однако во время фазы выполнения команды он находится в состоянии «Не занят». Следовательно, можно параллельно выполнять рекалибровку всех подключенных к контроллеру дисководов.

10.3 Набор команд FDC

229

Фаза выдачи результата у команды Рекалибровка отсутствует, поэтому для определения состояния дисковода после выполнения рекалибровки обычно используют команду Чтение состояния прерывания.

ВНИМАНИЕ

После включения питания компьютера программа-драйвер должна выполнить рекалибровку всех подключенных к контроллеру FDC дисководов

Интервалы времени, задаваемые полями HLT, HUT и SRT, зависят не только от записанных в эти поля значений, но и от используемой скорости передачи данных (см. табл. 10.16–10.18).

Таблица 10.16. Зависимость величины интервала времени «загрузки» головок (в миллисекундах) от установленной скорости передачи данных и значения в поле HLT

HLT	Скорость передачи данных			
	1 Мбит/с	500 кбит/с	300 кбит/с	250 кбит/с
00h	128	256	426	512
01h	1	2	3,3	4
02h	2	4	6,7	8
...
7Eh	126	252	420	504
7Fh	127	254	423	508

232

10 Контроллер накопителей на гибких магнитных дисках (FDC)

Таблица 10.17. Зависимость величины интервала времени «разгрузки» головок (в миллисекундах) от установленной скорости передачи данных и значения в поле HUT

HUT	Скорость передачи данных			
	1 Мбит/с	500 кбит/с	300 кбит/с	250 кбит/с
0h	128	256	426	512
1h	8	16	26,7	32
Eh	112	224	373	448
Fh	120	240	400	480

Таблица 10.18. Зависимость величины интервала шага головок (в миллисекундах) от установленной скорости передачи данных и значения в поле SRT

Таблица 10.18. Зависимость величины интервала шага головок (в миллисекундах) от установленной скорости передачи данных и значения в поле SRT

SRT	Скорость передачи данных			
	1 Мбит/с	500 кбит/с	300 кбит/с	250 кбит/с
0h	8,0	16	26,7	32
1h	7,5	15	25	30
Eh	1,0	2	3,33	4
Fh	0,5	1	1,67	2

Команда Поиск (SEEK), формат которой показан на рис. 10.17, обеспечивает перемещение головок дискового на заданный цилиндр диска. После получения данной команды контроллер сравнивает заданное значение номера цилиндра (NCN) с текущим значением (PCN). Если текущее и заданное значения не совпадают, то контроллер начинает выдавать приводу головок импульсы перемещения: когда текущий номер больше заданного, шаг перемещения отрицательный, когда меньше — положительный. Сделав шаг, контроллер снова сравнивает значения NCN и PCN; процесс продолжается до тех пор, пока цилиндр не будет найден. При обнаружении искомого цилиндра контроллер заканчивает выполнение команды, устанавливая равным 1 значение флага SE в регистре ST0. Скорость, с которой осуществляется перемещение головок, определяется значением интервала шага головки (SRT). Установленное по умолчанию при включении питания значение SRT может быть изменено при помощи команды Установка параметров, которую мы рассмотрим ниже.

	7	6	5	4	3	2	1	0
Фаза	0	0	0	0	1	1	1	1
посылки	0	0	0	0	0	HDS	DS1	DS0
команды	NCN							
Фаза	—							
выполнения								

Рис. 10.17. Формат команды Поиск

Read Sector

- Format: M F S 0 0 1 1 0
- Paramaters:
 - x x x x x HD DR1 DR0 = HD=head DR0/DR1=Disk
 - Cylinder
 - Head
 - Sector Number
 - Sector Size
 - Track Length
 - Length of GAP3
 - Data Length
- Return:
 - Return byte 0: ST0
 - Return byte 1: ST1
 - Return byte 2: ST2
 - Return byte 3: Current cylinder
 - Return byte 4: Current head

- Return byte 5: Sector number
- Return byte 6: Sector size

Команда **Чтение данных (READ DATA)** — наиболее часто используемая из всех команд данной группы. Выполнение данной команды начинается после завершения передачи девяти командных байтов (кода команды и ее атрибутов) от процессора к контроллеру. Когда команда принята, контроллер «загружает головки», если они находились в «незагруженном» состоянии (то есть прижимает головки к диску, если они не были прижаты). Затем контроллер ожидает завершения установки головок в течение интервала времени HLT (задаваемого командой SPECIFY) и начинает считывание адресных меток и полей идентификатора ID. Если контроллер обнаруживает на дорожке сектор, адрес которого соответствует заданному в команде, он осуществляет считывание сектора и передачу данных процессору через FIFO.

Контроллер дисководов может работать в режиме *мультисекторного считывания данных*, причем мультисекторный режим задается не какой-нибудь специальной командой FDC, а значением счетчика передаваемых данных контроллера DMA. После завершения операции считывания текущего сектора контроллер увеличивает адрес сектора на единицу и приступает к считыванию информации из следующего логического сектора. В режиме DMA контроллер дисководов будет считывать секторы и передавать хосту содержащуюся в них информацию до тех пор, пока контроллер DMA не установит в состояние 1 сигнал **Конец передачи (ТС)**, то есть пока не будет передан весь объем информации, на который контроллер DMA был запрограммирован. Если в процессе обработки очередного сектора возникает аварийная ситуация (переполнение FIFO или установка в состояние 1 сигнала ТС), то передача данных от FDC к хосту прекращается, но сектор все равно будет прочитан до конца, после чего будет осуществлена проверка контрольной суммы CRC и завершено выполнение команды **Чтение данных**.

ПРИМЕЧАНИЕ

Хостом в данном случае является не процессор, а участок памяти, заданный контроллером DMA

Максимальный объем данных, который может быть прочитан по одной команде чтения, определяется кодом размера сектора N и значением признака многодорожечной операции MT (см. табл. 10.10).

Функция многодорожечной (двусторонней) операции, задаваемая битом MT первого байта команды, позволяет контроллеру по одной команде считывать информацию с обеих сторон диска. Считывание информации при этом начинается с первого сектора стороны 0. Когда завершается обработка последнего сектора стороны 0, контроллер приступает к обработке первого сектора стороны 1. Операция завершается, когда на стороне 1 будет обработан последний сектор.

Таблица 10.10. Максимальный объем данных, который можно передать при выполнении мультисекторной операции

N	Размер сектора, байт	Число секторов на дорожке	MT	Объем передаваемых данных, байт
1	256	26	0	6656
1	256	26	1	13312
2	512	15	0	7680
2	512	15	1	15360
3	1024	8	0	8192
3	1024	8	1	16384

Если хост прерывает операцию чтения или записи, выполняемую контроллером FDC, то ID-информация, возвращаемая в конце фазы результата, будет зависеть от значения бита MT и байта EOT (см. табл. 10.11).

Таблица 10.11. Идентификационная информация, возвращаемая в фазе результата

MT	Головка	Номер последнего сектора, переданного хосту	ID-информация в фазе результата			
			C	H	R	N
0	0	Меньше, чем EOT	Без изм	Без изм	R+1	Без изм
		Равен EOT	C+1	Без изм	1	Без изм
	1	Меньше, чем EOT	Без изм	Без изм	R+1	Без изм
		Равен EOT	C+1	Без изм	1	Без изм
1	0	Меньше, чем EOT	Без изм	Без изм	R+1	Без изм
		Равен EOT	Без изм	1	1	Без изм
	1	Меньше, чем EOT	Без изм	Без изм	R+1	Без изм
		Равен EOT	C+1	0	1	Без изм

После выполнения команды чтения данных головки дисководов остаются в «загруженном» состоянии в течение некоторого интервала времени — интервала разгрузки головок (Head Unload Time Interval, сокращенно **HUT**). Это позволяет ускорить доступ к информации, если в течение HUT будет подана еще одна команда передачи данных (не нужно будет снова «загружать» головки). Величину HUT можно изменить при помощи команды SPECIFY.

Если в процессе поиска сектора контроллер дисководов дважды получает сигнал от датчика индекса IDX, но при этом не находит заданный сектор R, то в регистре статуса ST0 будет установлен код IC = 01 (аварийное завершение операции), а в регистре ST1 флаг ND (данные не найдены на диске) примет значение 1.

После считывания идентификационной информации и полей данных сектора FDC проверяет его контрольный код CRC. Если при проверке CRC обнаружена ошибка, то в регистре статуса ST0 контроллер устанавливает код IC = 01 (аварийное завершение операции), а в регистре ST1 флаг DE примет значение 1. Кроме того, если ошибка выявлена в полях ID, контроллер устанавливает равным 1 значение флага DD в регистре SR2.

В зависимости от значения бита SK в первом байте команды контроллер будет либо считывать, либо пропускать секторы, помеченные как «удаленные». Воздействие бита SK на процесс выполнения и результаты команды чтения данных показано в табл. 10.12.

Таблица 10.12. Влияние бита SK на выполнение команды Чтение данных

SK	Метка типа данных	Результаты		
		Сектор прочитан?	Значение бита CM в ST2	Описание результатов
0	Нормальные данные	Да	Нет	Нормальное завершение
	Удаленные данные	Да	Да	Адрес не увеличивается Поиск следующего сектора не выполняется
1	Нормальные данные	Да	Нет	Нормальное завершение
	Удаленные данные	Нет	Да	Нормальное завершение Сектор не прочитан (пропущен)

The Proper Way to issue a command

1. Read MSR (port 0x3F4).
2. Verify that RQM = 1 and DIO = 0 ((Value & 0xc0) == 0x80) -- if not, reset the controller and start all over.
3. Send your chosen command byte to the FIFO port (port 0x3F5).
4. In a loop: loop on reading MSR until RQM = 1. Verify DIO = 0, then send the next parameter byte for the command to the FIFO port.
5. Either Execution or Result Phase begins when all parameter bytes have been sent, depending on whether you are in PIO mode, and the command has an Execution phase. If using DMA, or the command does not perform read/write/head movement operations, skip to the Result Phase.
6. (In PIO Mode Execution phase) read MSR, verify NDMA = 1 ((Value & 0x20) == 0x20) -- if it's not set, the command has no Execution phase, so skip to Result phase.
7. begin a loop:
8. Either poll MSR until RQM = 1, or wait for an IRQ6, using some waiting method.
9. In an inner loop: transfer a byte in or out of the FIFO port via a system buffer, then read MSR. Repeat while RQM = 1 and NDMA = 1 ((Value & 0xa0) == 0xa0).
10. if NDMA = 1, loop back to the beginning of the outer loop, unless your data buffer ran out (detect underflow/overflow).
11. Result Phase begins. If the command does not have a Result phase, it silently exits to waiting for the next command.
12. If using DMA on a read/write command, wait for a terminal IRQ6.
 1. Loop on reading MSR until RQM = 1, verify that DIO = 1.

2. In a loop: read the next result byte from the FIFO, loop on reading MSR until RQM = 1, verify CMD BSY = 1 and DIO = 1 ((Value & 0x50) == 0x50).
13. After reading all the expected result bytes: check them for error conditions, verify that RQM = 1, CMD BSY = 0, and DIO = 0. **If not** retry the entire command again, several times, starting from step 2!

Note: implementing a failure timeout for each loop and the IRQ is pretty much required -- it is the only way to detect many command errors.

Какова последовательность запуска этих команд? Лучше всего это посмотреть в исходном коде, но если в двух словах:

Выполнив сброс контроллера, вам надо его проинициализировать, задав все рабочие параметры. Затем можно выдавать контроллеру команды, каждый раз проверяя регистр его основного состояния **ST** и анализируя байты результата **ST0...ST3**. Можно предложить следующую последовательность действий:

- Сброс контроллера выдачей в порт **3F2h** байта с битом **2**, установленным в **0**.
- Разрешение работы контроллера выдачей в этот же порт байта с битом **2**, установленным в **1**.
- Выдача контроллеру команды "Инициализация".
- Выдача контроллеру команды "Определить параметры".
- Включить двигатель и выждать примерно 0,5 с (время разгона двигателя).
- Установить головки в нужное положение командой "Поиск".
- Проверить результаты установки командой "Чтение состояния прерывания".
- Для машины АТ установить нужную скорость передачи данных, выдав в порт **3F7h** байт с соответствующим значением: **0** для дискет с высокой плотностью записи (HD), **1** для двойной плотности (DD) и **2** для одинарной (SD).

- Если установка головок произведена правильно, можно выдавать команды чтения/записи данных. Перед этим надо правильно запрограммировать контроллер прямого доступа к памяти, если вы собираетесь использовать режим DMA.

Регистры состояния

How to get return values from commands

Unlike functions in programming in which you can ignore return values, the FDC requires for them to be processed in some way. Granted, you can still ignore them, but you must get them from the FDC. The FDC won't allow any more commands until it is done.

If the command returns data, it will be returned -- one at a time -- in the FIFO (Data register). So, to read them, you must continually read the FIFO to get all of the returned data.

Note: If a command returns data, it will send an interrupt that you must wait for. This is how you will know when the command is done and that it is safe to read the return values from the FIFO.

A good example of return values is the read sectors command. It requires us to wait for an IRQ so we know it completes, and returns 7 bytes. So to read all of the returned data bytes, we have to read from the data register one at a time:

```
for (int j=0; j<7; j++)  
  
    flpydisk_read_data ();
```

Of course, for error checking purposes you should actually check some of the return values.

Регистрами состояния возвращаемыми большинством команд являются st0, st1, st2, st3.

Сведения о состоянии контроллера хранятся в регистрах состояния ST0–ST3. Непосредственно эти регистры для считывания недоступны — не имеют собственных адресов. Информацию из регистров состояния можно получить через регистр данных только после завершения выполнения какой-либо команды. В зависимости от кода команды через FIFO в фазе результата передается либо содержимое группы регистров ST0–ST2, либо — содержимое только регистра ST0, либо — содержимое ST3.

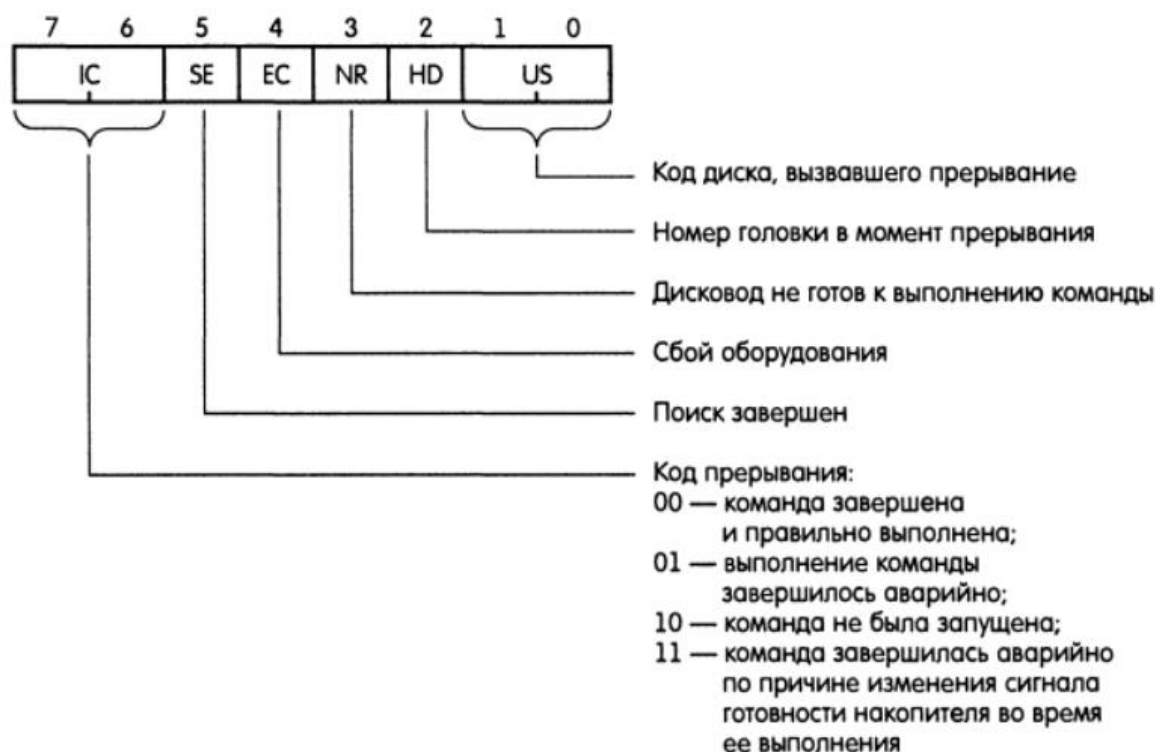


Рис. 10.8. Формат регистра состояния ST0

Формат регистра ST0 показан на рис. 10.8. Разряды регистра имеют следующее назначение:

- ◆ биты 0 и 1 (US0, US1) — номер вызвавшего прерывание дисковод, представленный в двоичном коде аналогично полю DS регистра DOR (см. табл. 10.2);
- ◆ бит 2 (HD) — номер головки вызвавшего прерывание дисковод (в момент подачи запроса прерывания);
- ◆ бит 3 (NR) — признак неготовности дисковод (принимает значение 1, если дисковод не готов к выполнению команд записи и считывания данных);
- ◆ бит 4 (EC) — признак возникновения сбоя оборудования (принимает значение 1, когда от дисковод поступает сигнал ошибки, или в том случае, когда

при выполнении рекалибровки нулевая дорожка не найдена после выполнения 77 шагов);

- ◆ бит 5 (SE) — признак завершения поиска дорожки (принимает значение 1 при завершении команды поиска дорожки SEEK независимо от того, каков результат поиска);
- ◆ биты 6 и 7 (IC) — код прерывания (00b — нормальное завершение команды; 01b — команда завершилась аварийно; 10b — команда не выполнена, поскольку имеет некорректный код; 11b — команда завершилась аварийно по причине внезапного изменения сигнала готовности накопителя во время ее выполнения).

ВНИМАНИЕ

Бит признака неготовности дисковод NR у некоторых современных контроллеров заблокирован в состоянии 0. Соответственно, не рекомендуется использовать данный признак при написании программ (чтобы не возникали проблемы с переносимостью программного обеспечения).

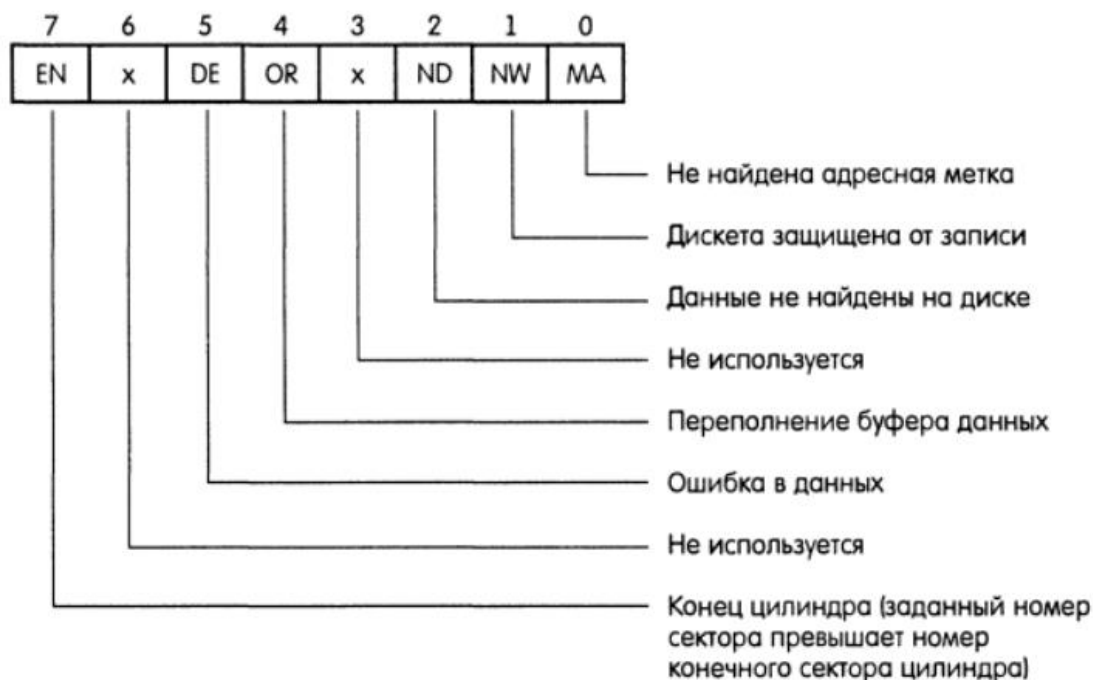


Рис. 10.9. Формат регистра состояния ST1

Назначение разрядов регистра ST1, формат которого изображен на рис. 10.9, следующее:

- ◆ бит 0 (MA) — признак отсутствия адресной метки на дорожке (принимает значение 1, если при попытке выполнения какой-либо операции контроллер не смог обнаружить маркер дорожки);
- ◆ бит 1 (NW) — признак наличия защиты от записи (принимает значение 1 при попытке выполнения записи данных на диск, если диск защищен от записи);
- ◆ бит 2 (ND) — признак отсутствия данных (принимает значение 1, если запрашиваемые данные не найдены на диске);

10.2. Регистры FDC

213

- ◆ бит 3 — не используется и всегда находится в состоянии 0;
- ◆ бит 4 (OR) — признак переполнения буфера данных (принимает значение 1, если FDC не получает необходимого обслуживания со стороны процессора или контроллера DMA в течение определенного интервала времени, зависящего от скорости передачи данных);
- ◆ бит 5 (DE) — признак наличия ошибки в данных (принимает значение 1 при обнаружении ошибки по контрольному коду CRC в идентификационном поле или поле данных);
- ◆ бит 6 — не используется и всегда находится в состоянии 0;
- ◆ бит 7 (EN) — признак конца цилиндра (принимает значение 1 при попытке обращения к сектору, номер которого превышает установленный для данного носителя информации максимальный номер сектора).

Рассмотрим более подробно признак отсутствия данных ND. Данный разряд регистра ST1 принимает значение 1 в следующих ситуациях:

- ◆ при выполнении команды Чтение данных или команды Чтение удаленных данных не был найден заданный сектор;
- ◆ при выполнении команды Чтение идентификатора контроллер не смог прочесть идентификатор или при чтении была обнаружена ошибка (по контрольному коду);
- ◆ при выполнении команды Чтение дорожки обнаружены какие-либо нарушения в считываемой последовательности секторов.

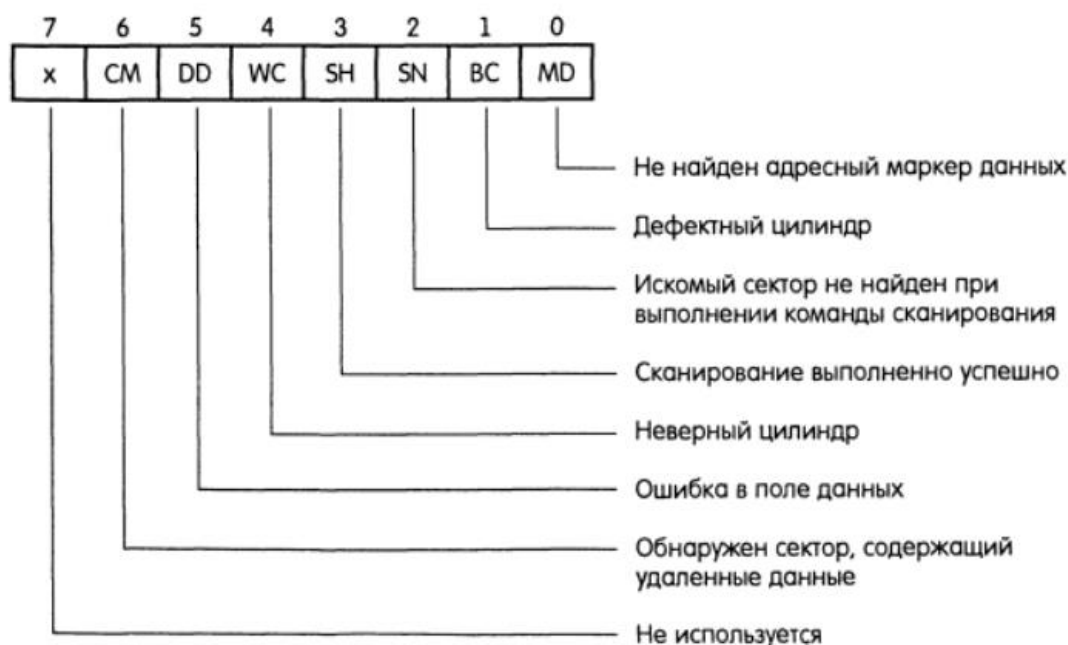


Рис. 10.10. Формат регистра состояния ST2

Формат регистра состояния ST2 изображен на рис. 10.10. Назначение разрядов регистра ST2 следующее:

- ◆ бит 0 (MD) — признак отсутствия адресной метки данных (принимает значение 1, если контроллер не может обнаружить маркер данных);
- ◆ бит 1 (BC) — признак дефектного цилиндра (принимает значение 1, если адрес дорожки в идентификаторе сектора отличается от предполагаемого контроллером, а также если дорожка помечена как дефектная, — в идентификаторах секторов в поле номера дорожки записан код FFh);
- ◆ бит 2 (SN) — признак выполнения операции сканирования секторов (находится в состоянии 1 во время фазы выполнения команд сканирования; в современных контроллерах в связи с изъятием из употребления команд сканирования данный бит не используется и всегда находится в состоянии 0);
- ◆ бит 3 (SH) — признак обнаружения эквивалентного сектора (принимает значение 1, если при выполнении команды сканирования был обнаружен эквивалентный сектор; в современных контроллерах в связи с изъятием из употребления команд сканирования данный бит не используется и всегда находится в состоянии 0);
- ◆ бит 4 (WC) — признак неверного номера цилиндра (принимает значение 1, если адрес дорожки в идентификаторе сектора отличается от предполагаемого контроллером, то есть от заданного в команде);
- ◆ бит 5 (DD) — признак наличия ошибки в поле данных (принимает значение 1, если при проверке считанных данных по контрольному коду CRC обнаружена ошибка);
- ◆ бит 6 (CM) — признак несоответствия данных используемой команде чтения (имеет значение 1, если при выполнении команды Чтение данных обнаружен сектор, содержащий удаленные данные, или при выполнении команды Чтение удаленных данных обнаружен сектор, содержащий действительные данные);
- ◆ бит 7 — не используется и всегда находится в состоянии 0.

Разряды регистра ST3, формат которого показан на рис. 10.11, имеют следующее назначение:

- ◆ биты 0 и 1 (US0, US1) — номер текущего (выбранного для работы) дисководов, записанный в двоичном коде (см. табл. 10.2);
- ◆ бит 2 (HD) — номер текущей головки;
- ◆ бит 3 (TS) — признак того, что текущий дисковод предназначен для работы с двухсторонними дисками (односторонние дисководы давно сняты с производства, поэтому данный разряд у современных контроллеров зафиксирован в состоянии 1);
- ◆ бит 4 (T0) — признак нулевой дорожки (имеет значение 1, если головки находятся на нулевом цилиндре);
- ◆ бит 5 (RDY) — признак готовности дисководов к работе (в современных контроллерах не используется, зафиксирован в состоянии 1);

10.3. Набор команд FDC

215

- ◆ бит 6 (WP) — значение сигнала датчика защиты от записи (имеет значение 1, если диск защищен от записи);
- ◆ бит 7 (FT) — признак неисправности накопителя (в современных контроллерах данный разряд не используется — зафиксирован в состоянии 0).

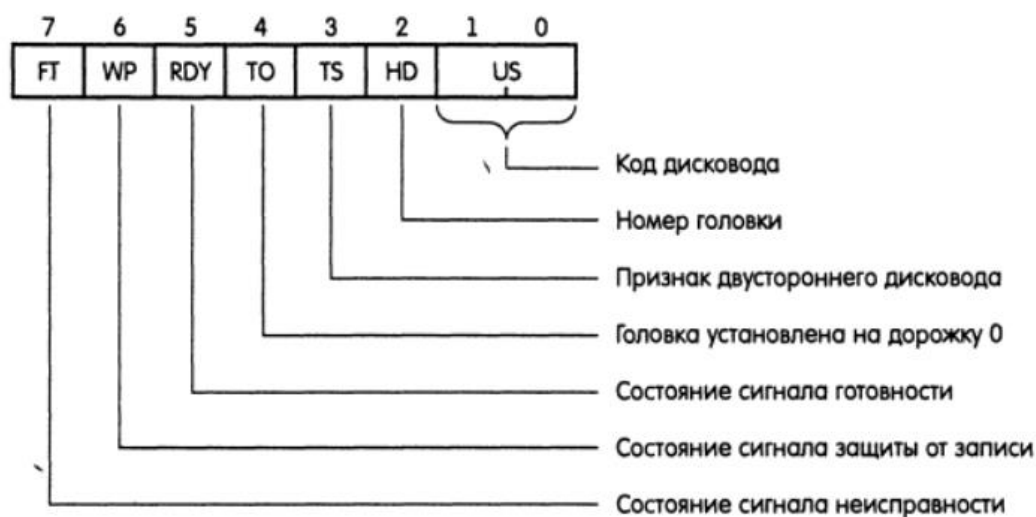


Рис. 10.11. Формат регистра состояния ST3

Как видно из приведенного выше описания, информация в регистрах состояния размещена довольно произвольным образом — регистры не имеют ярко выраженного функционального назначения; кроме того, некоторые признаки дублируются в разных регистрах. Данный недостаток, затрудняющий анализ состояния дисковой подсистемы, современные контроллеры гибких дисков получили в наследство от первых машин типа IBM PC.

The second most common failure mode for any floppy command is for the floppy controller to lock up forever. This condition is detected with a timeout, and needs to be fixed with a Reset. You also need one Reset during initialization. Hopefully, it is the only one you will ever need to do.

Bochs for Windows - Display

kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8063

type = 1, base = 0xFFBFF000, length = 0x1000
type = 2, base = 0xFFBFE000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>read

Please type in the sector number [0 is default] >

Sector 0 contents:

```

0x0E9 0xB7 0x1 0x90 0x76 0x73 0x83 0xB4 0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0 0x0 0x0 0x40 0xB 0x0 0x0 0x0 0x0 0x0 0x0 0x1 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x2 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x29 0x0
0xD6 0x6 0x0 0x0 0x0 0x0 0x0 0x2 0x0 0x0 0x48 0x69 0x20 0x74 0x68 0x65 0x72 0x6
5 0x21 0x20 0x4D 0x79 0x20 0x4F 0x53 0x20 0x62 0x6F 0x6F 0x74 0x20 0x6C 0x6F 0x6
1 0x64 0x65 0x72 0x20 0x68 0x61 0x76 0x65 0x20 0x73 0x74 0x61 0x72 0x74 0x65 0x6
4 0x20 0x74 0x6F 0x20 0x77 0x6F 0x72 0x6B 0x21 0xD 0xA 0x0 0x50 0x72 0x65 0x73 0
x73 0x20 0x61 0x6E

```

Press any key to continue...

CTRL + 3rd button enables mouse IPS: 83.821M A: NUM CAPS SCRL

КОД:

<https://github.com/devsienko/c4os/tree/565d9aa0f1f2380022d89102341054aacbf11bbb>.

Многозадачность

В очередной раз копираст теории без спроса, да простит меня ее автор, но почему то сейчас я решил об этом упомянуть)) Взято с <https://subscribe.ru/archive/comp.soft.myosdev/201208/15181714.html>.

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 за счёт 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 разработке программ под такую ОС потребуются 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 циклическим переключением задач без учёта приоритетов.

Так как переключение задач будет циклическим, списки задач/потоков удобно реализовать в виде двуправленного связанного кольцевого списка (подобный подход был использован при реализации файловой системы и менеджера физической памяти). При переключении достаточно брать следующий элемент списка. Переход из начала в конец произойдёт сам собой.

Чтобы работать со списками было удобно, реализуем три функции стандартной библиотеки специально для них (добавим их в *stdlib*):

```
typedef struct _ListItem ListItem;

typedef struct {
    ListItem *first;
    size_t count;
    Mutex mutex;
} List;

struct _ListItem {
    ListItem *next;
    ListItem *prev;
    List *list;
};

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));
}

```

Как это работает? Мы создаём новую структуру данных (например, описатель задачи или потока), 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.

Внимание! Для перебора списка захват семафора не требуется!

Исходный

код:

<https://github.com/devsienko/c4os/tree/f7c91237b0c82b12f5e8319933b7b649e0931cbd>.

План написания кода многозадачности таков. Сперва разобраться с пользовательским режимом и как в него перейти. Затем научиться загружать бинарный код user mode приложений и запускать его, а далее загружать и запускать несколько таких приложений в режиме многозадачности. Итак приступим.

Создадим файл *multikasking.h* со структурами данных для описания процесса/потока и некоторых других вещей:

```
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;
    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);
```

Так как в данный момент уже выполняется код, код ядра, хорошо бы и его представить как отдельный процесс. Это происходит в теле функции *init_multitasking* файла *multitasking.c*:

```

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_kernel_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;
}

```

Теперь у нас есть один процесс. Добавим вызов функции *switch_task* в обработчик прерывания таймера. Так как у нас всего лишь один процесс это не имеет смысла, но дальше, когда их число увеличится этот код будет нам полезен:

```

void switch_task(Registers *regs) {
    if (multitasking_enabled) {
        (*((char*)(0xB8000 + 79 * 2)))++; // just to indicate that switch task is working

        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));

return (uint32)&current_thread->state;
}
}

```

И напоследок в файл *kernel.c* добавим обработку команды **ps** в консоле, чтобы отображать список работающих в данный момент процессов, мы должны увидеть при ее запуске процесс *kernel*:

```

Bochs for Windows - Display

Boot disk id ----- 0
Memory map ----- 0x9000

region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8063

type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 2, base = 0xFFBFB000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>ps
process - kernel
Command>

CTRL + 3rd button enables mouse  IPS: 98.004M  A:  NUM  CAPS  SCRL

```

Исходный

код:

<https://github.com/devsienko/c4os/tree/97485855ad850b42b14ad3013d2ea86b37c8ed4c>.

Переход в пользовательский режим

Зачем нужен пользовательский режим (или user mode) для тех кто не знает можно попугулить. Здесь мы будем говорить о технической реализации перехода в него.

Согласно <http://www.broken Thorn.com/Resources/OSDev23.html> есть два способа перейти в режим пользователя. С помощью инструкции **SYSEXIT** или с помощью инструкции **IRET**. Мы будем использовать второй вариант. Этот метод используется во

многих операционных системах, поскольку он более переносимый, чем использование SYSEXIT. Более крупные операционные системы обычно поддерживают оба подхода и SYSEXIT, и IRET, если SYSEXIT недоступен.

IRET предназначена для возврата управления из процедуры обработчика прерывания в прерванную программу. Инструкция IRET ожидает, что стек будет содержать данные в следующем порядке (но то только при **смене уровня привилегий!** кода, если уровень привилегий не меняется ss и esp использоваться не будут, более подробно ищите здесь <https://www.ics.uci.edu/~aburtsev/143A/2017fall/>):

- SS
- ESP
- EFLAGS
- CS
- EIP

В обычной ситуации эти данные помещаются в стек при вызове инструкции INT. IRETD заставляет процессор перейти к CS:EIP, который он получает из стека. IRET также устанавливает регистр EFLAGS на значение взятое из стека. Но мы же можем манипулировать содержимым стека и следовательно настроить его так, чтобы IRET вернула нас туда, куда нам нужно, а именно передать управление коду пользовательского режима:

```
void switch_to_user_mode()
{
    // Set up a stack structure for switching to user mode.
    asm volatile(" \
        cli; \
        mov $0x23, %ax; \
        mov %ax, %ds; \
        mov %ax, %es; \
        mov %ax, %fs; \
        mov %ax, %gs; \
        \
        mov %esp, %eax; \
        pushl $0x23; \
        pushl %eax; \
        pushf; \
        pushl $0x1B; \
        push $1f; \
    "
```

```

        iret; \
1: \
        ");
printf("Hello, user world!");
while(true) {}
}

```

После того как мы перевели нашу ОС в пользовательский режим мы выводим на экран сообщение и переходим в бесконечный цикл. Скомпилировав и запустив наш обновленный код мы должны увидеть примерно следующее:

```

Bochs for Windows - Display
Welcome to C4 OS!
Boot disk id ----- 0
Memory map ----- 0x9000
region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)
kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8067
type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 2, base = 0xFFBFB000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000
Command>umode
Hello, user world!
CTRL + 3rd button enables mouse  IPS: 141.142M  A:  NUM  CAPS  SCRL

```

Но чтобы это все заработало пришлось пойти на некоторые ухищрения. Записи таблицы страниц настроены на работу в режиме ядра (что правильно), а я временно открыл доступ к ним для кода режима пользователя. В будущем я откачу эти изменения, а пока все их можно посмотреть в исходном коде: <https://github.com/devsienko/c4os/tree/a621d0bdc0ae7a48e33216b1da463ff41a507a63>.

Если вы запустили этот код, вы могли заметить, что символ в правом верхнем углу (который меняется с каждым прерыванием таймера) перестал меняться. Это все

из-за того, что мы запретили прерывания инструкцией `cli` (первая инструкция встроенной ассемблерной команды). А отключили прерывания по той причине, что после первого же прерывания наша система ребутнулась бы.

Когда происходит прерывание, начинается выполняться код его обработчика. Он находится в памяти ядра. Так как мы не меняем содержимое регистра `CR3`, смена адресного пространства не происходит и обработка прерывания будет производиться в адресном пространстве текущего процесса. Перед прерыванием указатель стека смотрит в память режима пользователя. Если его там и оставить, значит пользовательские программы могут влиять на код режима ядра, а это небезопасно и надо этого как то избежать. Возникается вопрос, как и куда перенастроить указатель стека (регистр `esp`)?

Этим занимается процессор, а процессору помогает в этом структура **TSS**. TSS (Task State Segment, сегмент состояния задачи) это специальная структура в архитектуре x86, содержащая информацию о задаче (процессе). Может использоваться ОС для диспетчеризации задач, но обычно (например в Linux) применяется только для переключения на стек ядра при обработке прерываний и исключений. В TSS содержится информация о:

- Состоянии регистров процессора;
- Разрешениях на использование портов ввода-вывода;
- Указатели на стек внутреннего уровня;
- Ссылка на предыдущую запись TSS (для задач диспетчеризации).

Грубо говоря, это аппаратный аналог нашей структуры *Registers* с некоторыми дополнительными полями, но процессор сохраняет всю информацию в TSS автоматически.

Изначально TSS задумывался как структура для аппаратной поддержки переключения задач. 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 в 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*:

```
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;

```

Теперь, чтобы воспользоваться этим механизмом, нам надо где-то в памяти выделить место под эту структуру, инициализировать ее и указать процессору где находится эта структура TSS. TSS описывается системным дескриптором, который может находиться только в GDT, поэтому это делается добавлением в нашу таблицу GDT новой специальной записи (в которой будет указано, где в памяти находится наша структура TSS) и указанием процессору на запись TSS в таблице GDT с помощью команды **ltr** (регистр текущей задачи (**TR**) является указателем на размещенный в GDT дескриптор сегмента TSS текущей задачи).

...

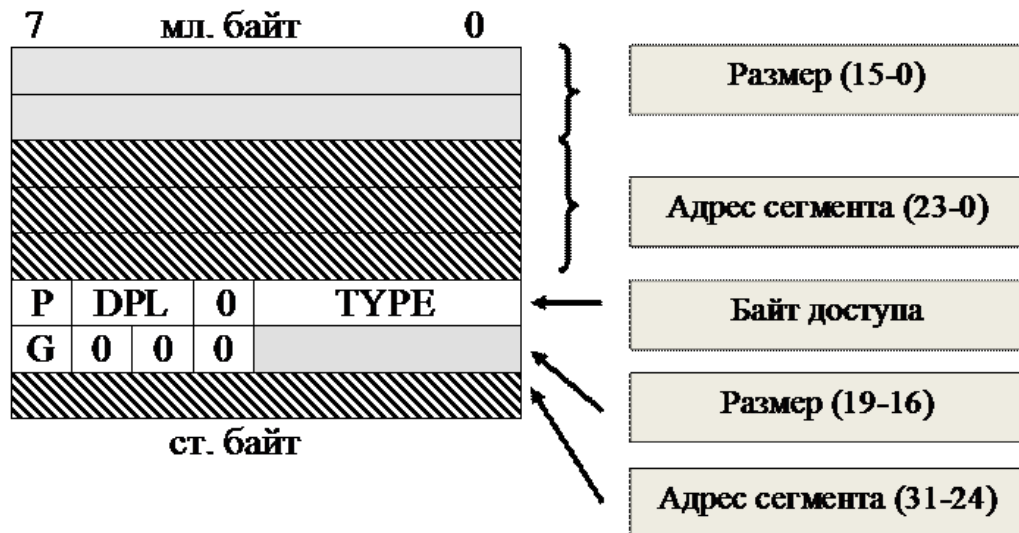
dq 0x7F4089FFD0002FFF

...

...

```
asm("ltr %w0::"a"(40));
```

...



Почему именно такое значение имеет эта запись можно будет понять чуть позже. Полей у этой структуры, как видите, очень много, но сейчас нас интересует лишь 2 из них - SS0, ESP0.

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 запикиваются 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.

SS0 и ESP0 могут быть общими для всех процессов (да так, вообще-то, и будет), но `io_map` будут для всех процессов разными (а имеется призрачная надежда на то, что однажды я доберусь до этой `io_map`), 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:

```
TSS *tss = (void*)(USER_MEMORY_END - PAGE_SIZE * 3 + 1);
```

И напоследок, нам надо обратно включить обработку прерываний. Мы отключили ее командой `cli`. Обычно ее включают командой `sti`, но мы не можем сделать это из режима пользователя и не можем вызвать ее из режима ядра до команды `iret`, это может привести нас к непредсказуемым последствиям.

Обратимся к справочной информации о командах `cli/sti`:

- команда CLI очищает флаг IF. На другие флаги или регистры она не влияет. Внешние прерывания не распознаются в конце команды CLI и начиная с этого момента до установки флага прерываний;
- команда STI устанавливает флаг IF. После выполнения следующей команды процессор может реагировать на внешние прерывания, если эта следующая команда оставляет флаг IF в состоянии, разрешающем прерывания.

Иными словами мы можем разрешить прерывания установив специальный бит регистра EFLAGS, а именно бит номер 9. Доработает нашу функцию `switch_to_user_mode`:

```
void switch_to_user_mode()
{
    tss_set_stack(0x10, get_current_esp());
    // Set up a stack structure for switching to user mode.
    asm volatile(" \
        cli; \
        mov $0x23, %ax; \
        mov %ax, %ds; \
        mov %ax, %es; \
        mov %ax, %fs; \
        mov %ax, %gs; \
        \
        mov %esp, %eax; \
    ");
}
```

```

        pushl $0x23; \
        pushl %eax; \
        pushf; \
        pop %eax; \
        or $0x200, %eax; \
        push %eax; \
        pushl $0x1B; \
        push $1f; \
        iret; \
1: \
        ");
printf("Hello, user world!");
while(true) {}
}

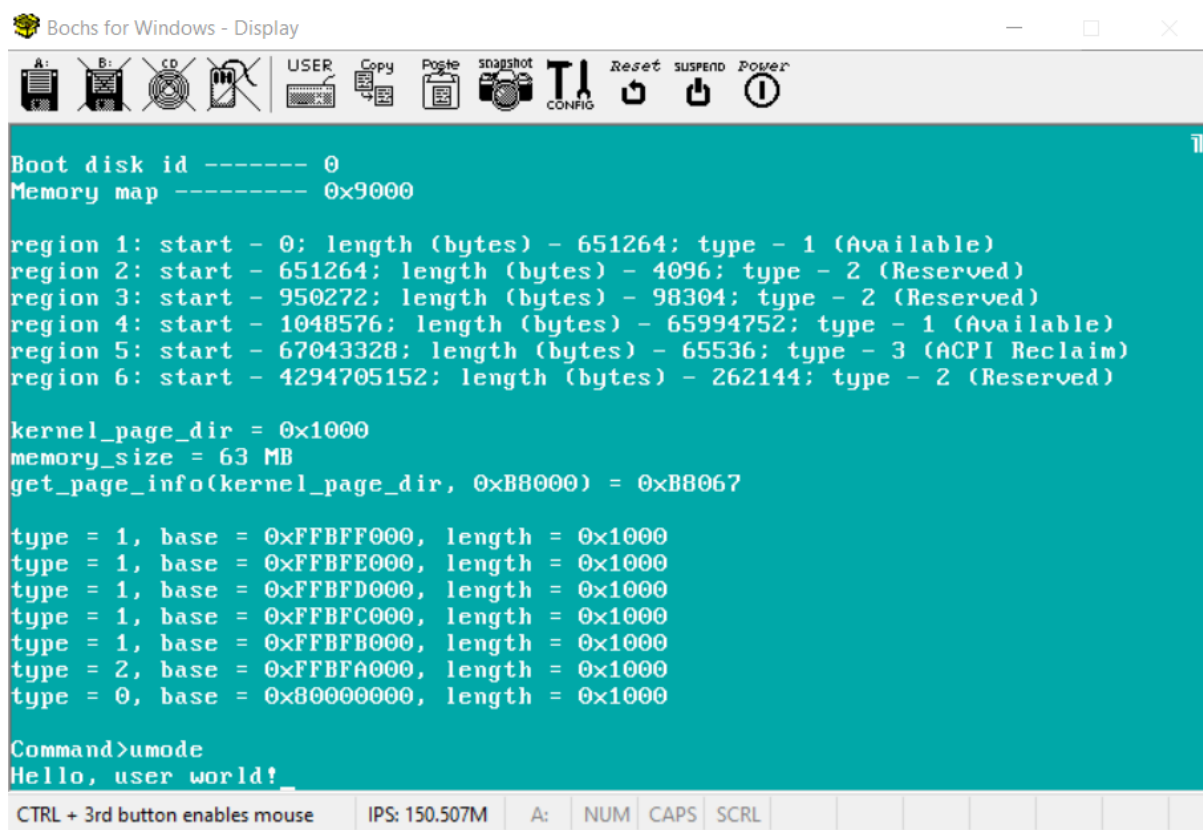
```

Исходные

код:

<https://github.com/devsienko/c4os/tree/219cf2c3fda250f5f12135bddf2878dd140b37ed>.

Запустив и скомпилировав его мы увидим ту же самую картину, что и в предыдущий раз с той лишь разницей, что символ в правом верхнем углу не перестает обновляться, а это значит, что обработка прерываний происходит как надо:



The screenshot shows the Bochs for Windows - Display window. The title bar includes standard window controls and a toolbar with icons for disk drives (A:, B:), CD-ROM, floppy disk, USER, Copy, Paste, snapshot, CONFIG, Reset, SUSPEND, and Power. The main display area has a black background with white text. It shows boot disk id as 0 and memory map starting at 0x9000. A list of six memory regions is displayed, including available, reserved, and ACPI Reclaim regions. Kernel parameters like kernel_page_dir, memory_size, and get_page_info are shown. A list of memory segments with their types, bases, and lengths follows. The command prompt shows 'umode' being entered, and the output 'Hello, user world!' is displayed. At the bottom, a status bar shows 'CTRL + 3rd button enables mouse', 'IPS: 150.507M', and keyboard status for A, NUM, CAPS, and SCRL.

```
Bochs for Windows - Display

Boot disk id ----- 0
Memory map ----- 0x9000

region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8067

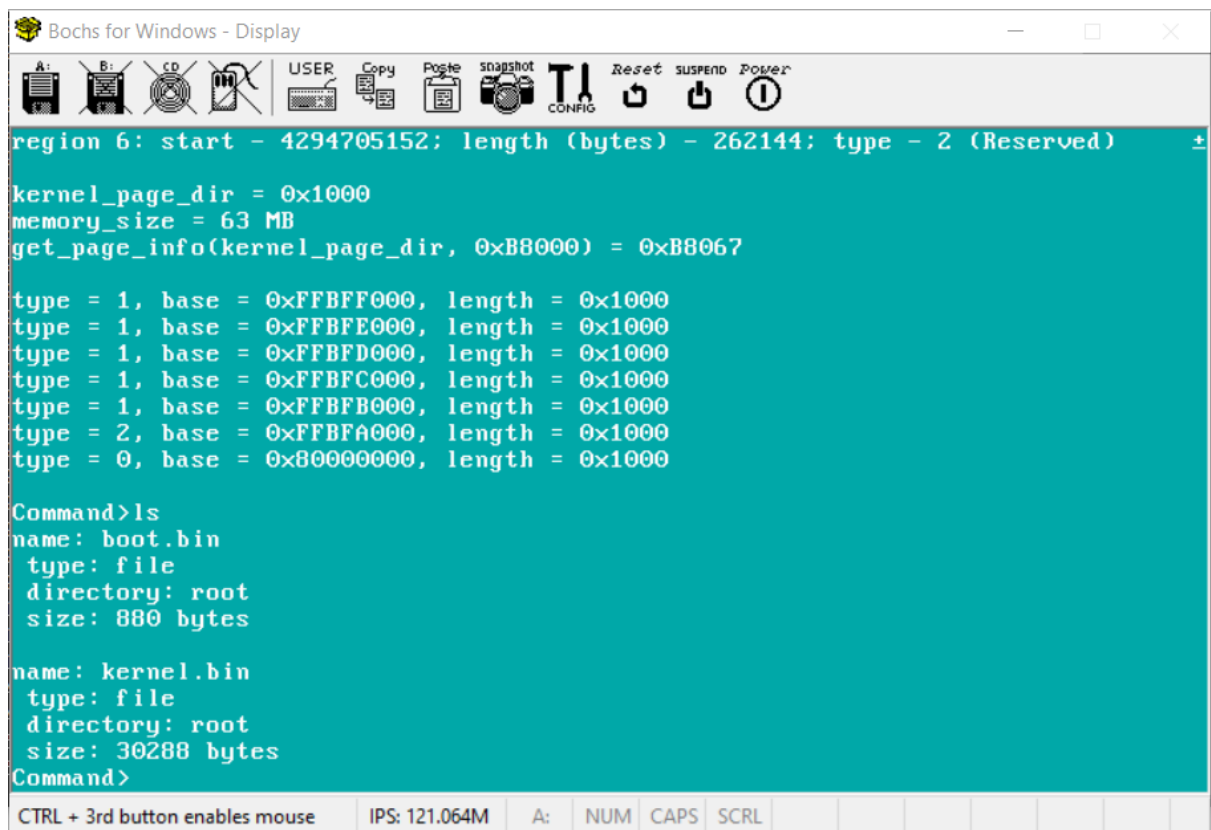
type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 1, base = 0xFFBFB000, length = 0x1000
type = 2, base = 0xFFBFA000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>umode
Hello, user world!

CTRL + 3rd button enables mouse  IPS: 150.507M  A:  NUM  CAPS  SCRL
```

В ресурсе <https://www.ics.uci.edu/~aburtsev/143A/2017fall/> хорошо описаны детали того, какие изменения и какие процессы происходят при смене уровня привилегий исполняемого кода. Советую ознакомиться с этой информацией, а также обратить внимание на то, что обращение к TSS происходит, как я понял изучая литературу, только при смене уровня привилегий. То есть, если выполняется код ядра и возникло прерывание, то для обработки прерывания будет использован тот же самый стек, если это не поменять руками.

Прежде чем вернуться к реализации многозадачности я добавлю команду **ls** для отображения файлов, которые записаны на floppy диске:



```
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8067

type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 1, base = 0xFFBFB000, length = 0x1000
type = 2, base = 0xFFBFA000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>ls
name: boot.bin
type: file
directory: root
size: 880 bytes

name: kernel.bin
type: file
directory: root
size: 30288 bytes
Command>
```

Используя ранее написанный floppy драйвер и информацию из нашей файловой системы мы отображаем все файлы, которые у нас есть. Исходный код здесь: <https://github.com/devsienko/c4os/tree/3afd58638a4844816b743eeb306792cbe357712d>.

Мы переходили в режим пользователя последовательным запуском встроенных ассемблерных команд `asm(...)`. Вынесем этот код в отдельный файл, скажем `first.asm`, и поместим его в папку `exes`, также добавим скрипт его компиляции и помещения на флоппи диск.

use32

org 0x20000 ; DMA buffer

; set tss start:

mov eax, esp

mov [0x7FFFD004], eax ; 7FFFD000 = USER_MEMORY_END - 3 pages

mov eax, 0x10

mov [0x7FFFD008], eax

cli

mov ax, 0x23

mov ds, ax

```

mov es, ax
mov fs, ax
mov gs, ax
mov eax, esp;
push 0x23
push 0x200000 ; esp of user mode code

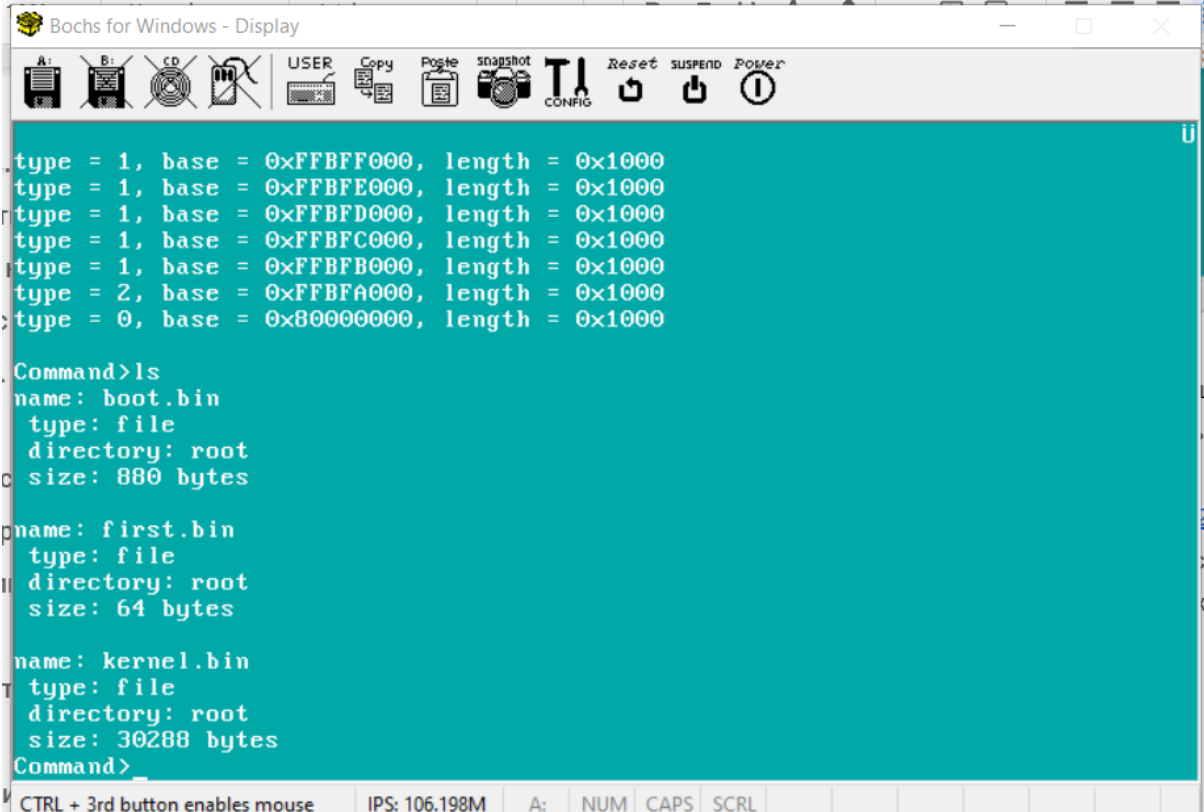
pushf
pop eax ; Get EFLAGS back into EAX. The only way to read EFLAGS is to pushf then pop.
or eax, 0x200 ; Set the IF flag.
push eax ; Push the new EFLAGS value back onto the stack.

push 0x1B
push @f
iret

@@:
mov byte[0xB8000 + (24 * 80) * 2 + 46], al
inc al
jmp @b

```

Командой **ls** мы можем посмотреть наши файлы:



The screenshot shows a window titled "Bochs for Windows - Display". The window contains a command prompt interface with a teal background. The command prompt shows the output of the 'ls' command, listing three files: boot.bin, first.bin, and kernel.bin. The output includes details such as file type, directory, and size. The command prompt is currently at the 'Command>' prompt.

```

type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 1, base = 0xFFBF0000, length = 0x1000
type = 2, base = 0xFFBFA000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>ls
name: boot.bin
type: file
directory: root
size: 880 bytes

name: first.bin
type: file
directory: root
size: 64 bytes

name: kernel.bin
type: file
directory: root
size: 30288 bytes
Command>

```

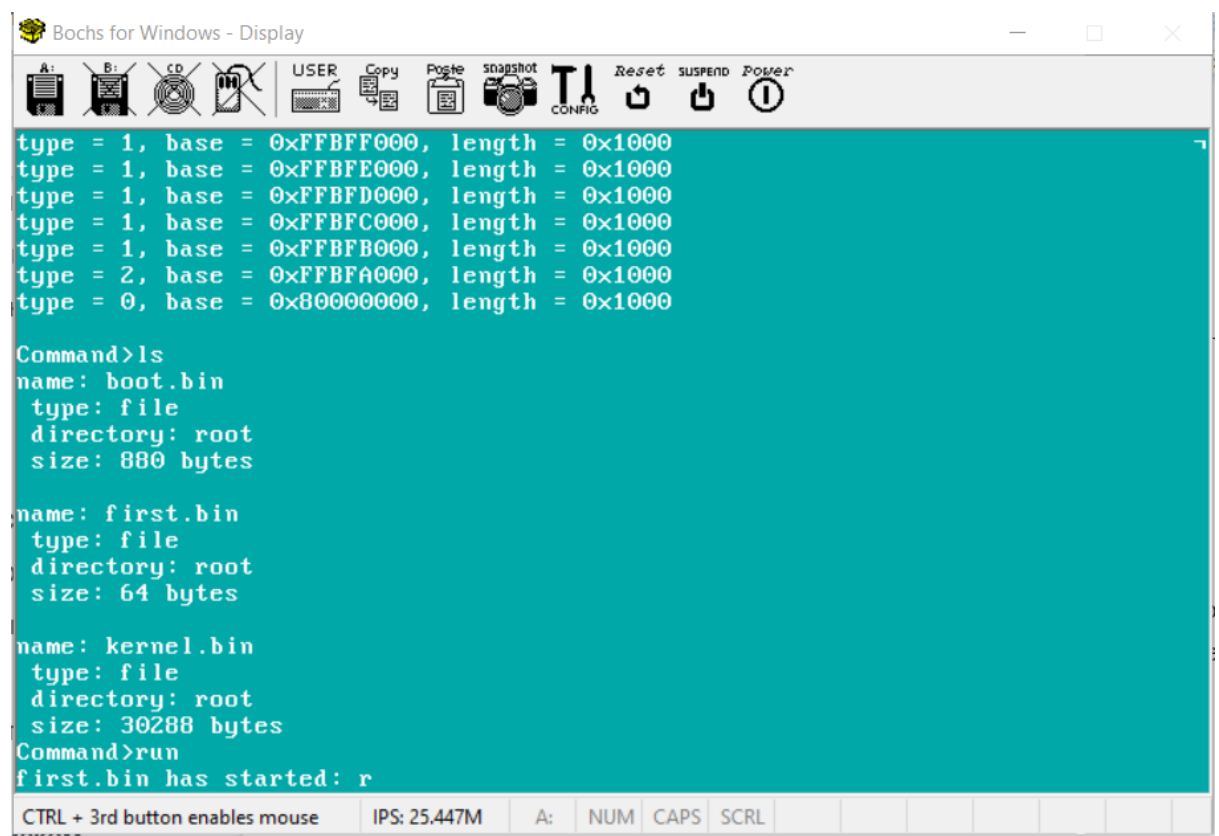
At the bottom of the window, there is a status bar with the following text: "CTRL + 3rd button enables mouse", "IPS: 106.198M", "A:", "NUM", "CAPS", "SCRL", and several empty boxes.

Напишем функцию, от которой явно неприятно пахнет и которую мы удалим при следующих изменениях кода:

```
void run_new_process(uint32 file_sector_number) {  
    char file_name[256] = "first.bin";  
    printf("first.bin has started: ", file_name);  
  
    uint32 file_data_sector_number = find_file(file_sector_number, file_name);  
    phyaddr dma_buffer = alloc_dma_buffer();  
    (uint8*)flpydisk_read_sector(file_data_sector_number);  
  
    ((func_t)dma_buffer)();  
}
```

Эта функция ищет наш файл на флоппи диске, читает его во временный буфер и после этого переводит программный поток в память этого буфера, тем самым начиная выполнение нашего файла (кода first.asm).

Наглядно это можно продемонстрировать запусив команду **run**:



Если вышепроизошедшее вам непонятно, то вот исходный код, он может ответить на многие ваши вопросы, а может и не ответить 😊:
<https://github.com/devsienko/c4os/tree/9e75acad50dabe218c8aa467f9606b0fab2543a1>.

Реализация многозадачности

Наконец мы добрались до этого момента. Что мы будем делать? Мы отслеживаем список всех процессов/потоков. Мы умеем загружать бинарные файлы в память. Теперь мы совместим это. Будем загружать в память программы, создавать для них записи в наших таблицах процессов, создавать под них адресные пространства и заставлять работать в многозадачном режиме.

Для переключения потоков можно использовать самые разные варианты, мы остановимся я на одном из них.

В обработчике прерывания от таймера все регистры процессора, а также адрес `temp_page` будут сохранены в стек, затем будет записан текущий указатель стека в структуру описателя потока. После этого будет производится поиск следующего потока для выполнения (в том числе это может оказаться и тот же самый). Когда поток найден, из его описателя восстанавливается 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 нового потока нужно заполнить его 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.

Приступим. Функция для создания нового потока:

```
Thread *create_thread(Process *process, void *entry_point, size_t stack_size_in_pages,
    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_in_pages << PAGE_OFFSET_BITS;
    thread->stack_base = alloc_virt_pages(&process->address_space, 0x200000, -1,
    stack_size_in_pages, PAGE_PRESENT | PAGE_WRITABLE | (kernel ? 0 :
    PAGE_USER));
    memset(&thread->state, 0, sizeof(Registers));
```

```

uint32 data_selector = (kernel ? 16 : 35);
uint32 code_selector = (kernel ? 8 : 27);
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;
}

```

Тут вроде бы все очевидно. Далее идет функция создания адресного пространства нового процесса. Тут немного читерства, потому что настройки таблиц ядра скопированы из процесса ядра. Образ программы загружается по адресу **0x100000**, первый мегабайт мы не трогаем, потому что так жить сильно проще:

```

phyaddr init_page_tables(phyaddr memory_location_start) {
    uint8 kernel_pte_settings = PAGE_PRESENT | PAGE_WRITABLE;
    uint8 user_mode_pte_settings = PAGE_PRESENT | PAGE_WRITABLE |
PAGE_USER;

    // new process page table addresses:
    phyaddr first_page_table_phyaddr = alloc_phys_pages(1);
    phyaddr page_table_with_tss_records = alloc_phys_pages(1);
    phyaddr kernel_page_table_phyaddr = get_kernel_page_table_addr(); // we just copy
page table of kernel process
    phyaddr last_page_table_phyaddr = get_last_page_table_addr(); // we just copy page
table of kernel process

    // new process page dir:
    phyaddr page_dir = alloc_phys_pages(1);
    temp_map_page(page_dir);
    uint32 *page_dir_p = (uint32*)TEMP_PAGE;

```



```

    page_dir_p[0] = first_page_table_phyaddr | user_mode_pte_settings;
    page_dir_p[511] = page_table_with_tss_records | kernel_pte_settings; // 511 -
because we use last three pages of user address space to store TSS segment data
    page_dir_p[1022] = kernel_page_table_phyaddr; // 1022 - the same approach like
within kernel process
    page_dir_p[1023] = last_page_table_phyaddr; // 1023 - the same approach like within
kernel process

// identity mapping of the 1st Mb
uint32 first_mb_limit = 0x100000;
temp_map_page(first_page_table_phyaddr);
uint32 *first_page_table_p = (uint32*)TEMP_PAGE;
uint32 entry_value = 0 | user_mode_pte_settings;
uint16 entries_count = first_mb_limit / PAGE_SIZE;
for (int i = 0; i < entries_count; i++) {
    first_page_table_p[i] = entry_value;
    entry_value += 0x1000;
}

// identity mapping of 2-4 Mb, pay attention on memory_location_start
phyaddr fourth_mb_limit = 0x400000;
int start_index = first_mb_limit / PAGE_SIZE;
entries_count = start_index + ((fourth_mb_limit - first_mb_limit) / PAGE_SIZE);
entry_value = memory_location_start | user_mode_pte_settings;
for (int i = start_index; i < entries_count; i++) {
    first_page_table_p[i] = entry_value;
    entry_value += 0x1000;
}

// fill page table with tss
phyaddr page_for_tss = alloc_phys_pages(1);
temp_map_page(page_table_with_tss_records);
uint32 *page_table_with_tss_records_p = (uint32*)TEMP_PAGE;
page_table_with_tss_records_p[1021] = page_for_tss | kernel_pte_settings;

return page_dir;
}

```

И наконец функция **run_image**, которая запускает нашу загруженную в память программу добавлением соответствующих записей в таблицы программ/поток:

```

void run_image(phyaddr proc_image_start, char file_name[]) {
    phyaddr page_dir = init_page_tables(proc_image_start);
    Process *new_process = alloc_virt_pages(&kernel_address_space, NULL, -1, 1,
    PAGE_PRESENT | PAGE_WRITABLE);
    init_address_space(&new_process->address_space, page_dir);
    new_process->suspend = false;
    new_process->thread_count = 0;
    strncpy(new_process->name, file_name, sizeof(new_process->name));

    list_append((List*)&process_list, (ListItem*)new_process);
    create_thread(new_process, (void*)USER_PROGRAM_BASE, 1, false, false);
}

```

Компилируем, запускаем, тестируем:

The screenshot shows the Bochs for Windows - Display window. The main display area has a teal background and shows the following text:

```

region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8063

type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 1, base = 0xFFBFB000, length = 0x1000
type = 2, base = 0xFFBFA000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>run
Command>ps
  process - kernel
  process - first.bin
Command>

```

At the bottom of the window, there is a status bar with the following information:

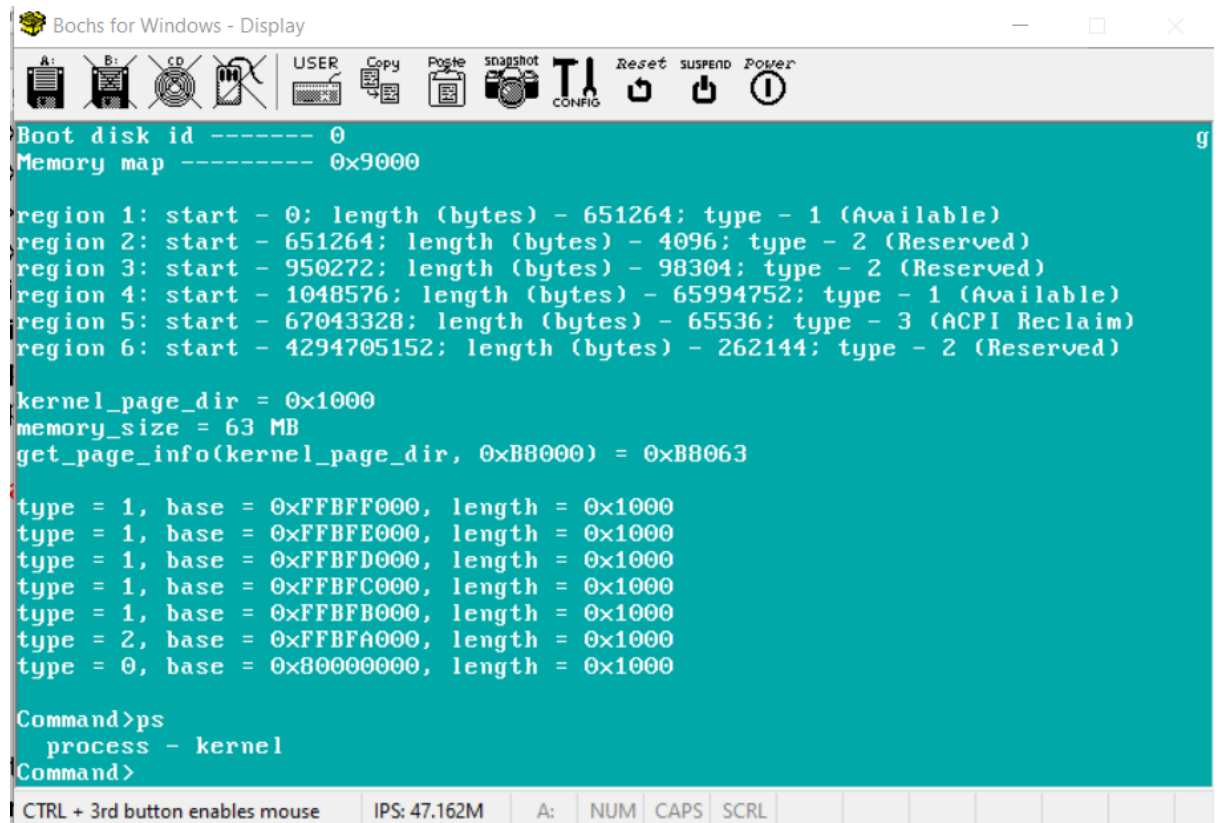
```

CTRL + 3rd button enables mouse   IPS: 50.870M   A:   NUM   CAPS   SCRL

```

Так же мы вернули на место уровни привилегий, которые меняли ранее для перехода в пользовательский режим. Теперь все по фен-шую. Исходный код: <https://github.com/devsienko/c4os/tree/db0f190fdf182389df2e3cb859babbf7506fd010>.

У нас есть приложение first.asm. Создадим еще несколько похожих, с той разницей, что они будут отображать меняющиеся символы в разных углах дисплея: ltexе, lbexе, rbexе (lt - left top, lb - left bottom, rb - right bottom). Немного поменяю способ их запуска. Чтобы запустить такое приложение нужно напечатать его имя в качестве команды. Перед тем как это сделать отобразим наш список процессов:



```
Bochs for Windows - Display
-----
Boot disk id ----- 0
Memory map ----- 0x9000

region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

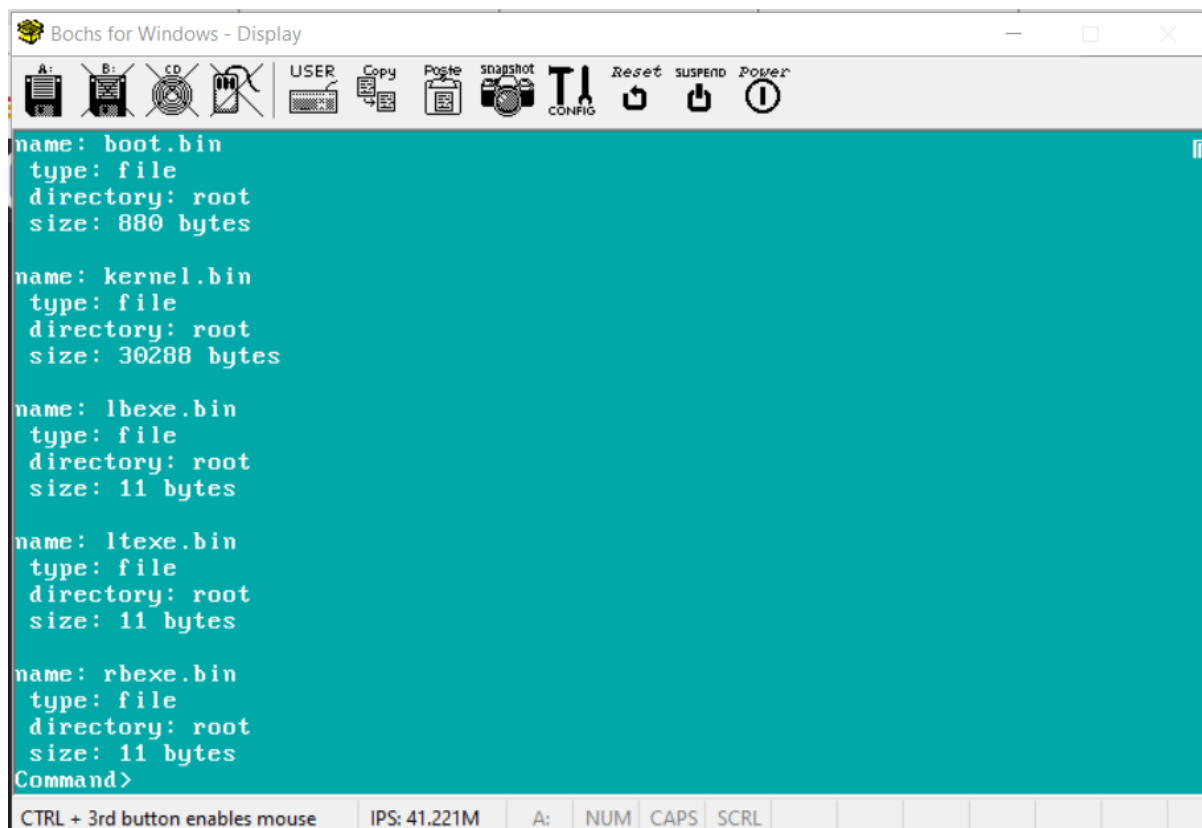
kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8063

type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 1, base = 0xFFBF0000, length = 0x1000
type = 2, base = 0xFFBFA000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>ps
  process - kernel
Command>
```

CTRL + 3rd button enables mouse IPS: 47.162M A: NUM CAPS SCRL

Пока запущен только процесс ядра. Отобразим список наших файлов:



```
name: boot.bin
type: file
directory: root
size: 880 bytes

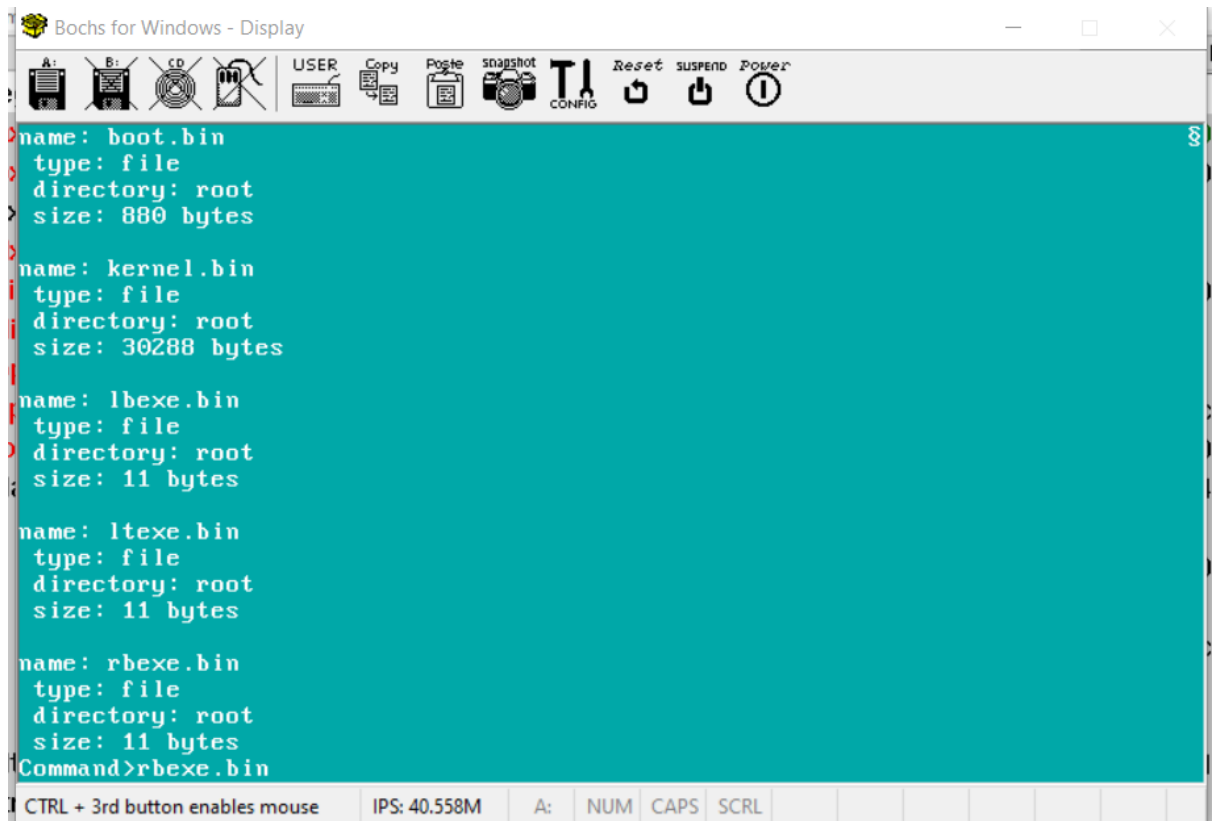
name: kernel.bin
type: file
directory: root
size: 30288 bytes

name: lbexe.bin
type: file
directory: root
size: 11 bytes

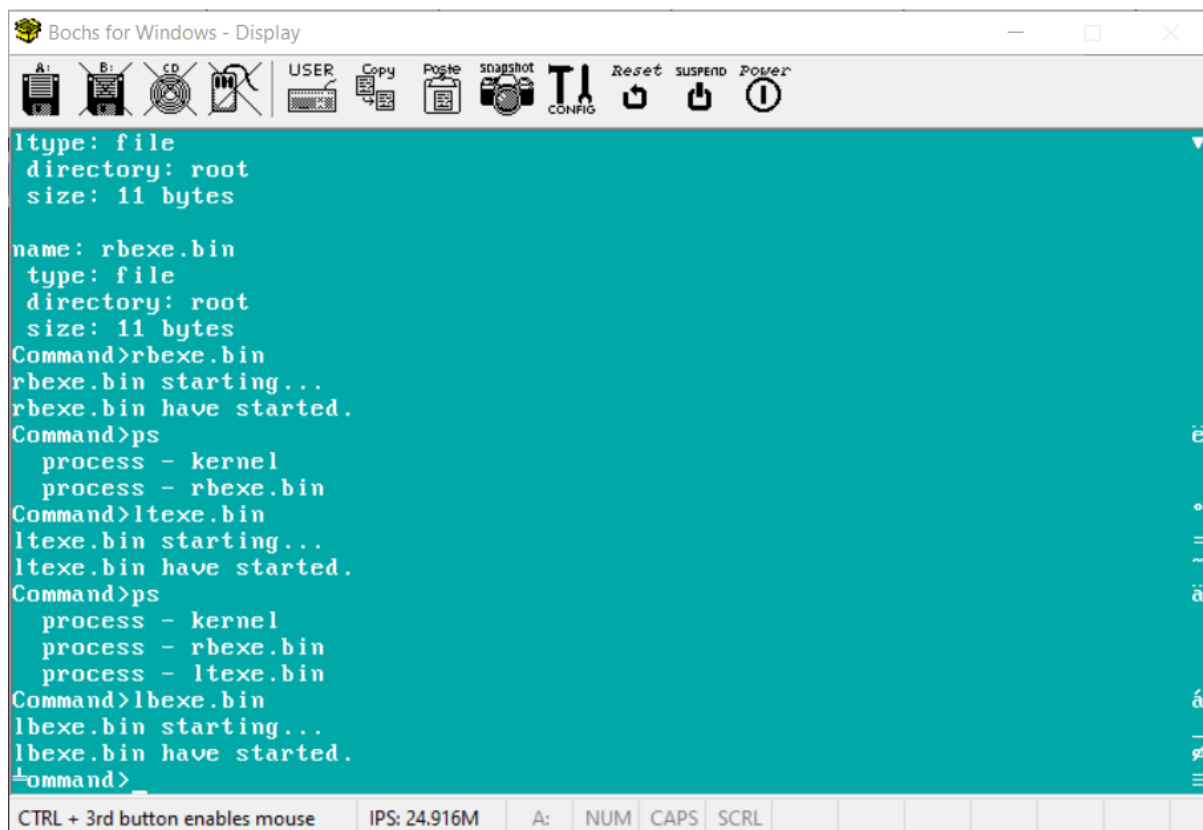
name: ltexe.bin
type: file
directory: root
size: 11 bytes

name: rbexe.bin
type: file
directory: root
size: 11 bytes
Command>
```

Для начала запустим например **rbexe.bin** (столкнулся с проблемой, что не всегда с первого раза запускаются эти программы, иногда приходится ребутать машину, иногда при последующих попытках начинают работать без ребута, сейчас оставляю все как есть, и так сойдет, как говорится)):



Символы забежали по углам дисплея виртуальной машины, можем посмотреть таблицу процессов:



```
ltype: file
directory: root
size: 11 bytes

name: rbexe.bin
type: file
directory: root
size: 11 bytes
Command>rbexe.bin
rbexe.bin starting...
rbexe.bin have started.
Command>ps
process - kernel
process - rbexe.bin
Command>ltexe.bin
ltexe.bin starting...
ltexe.bin have started.
Command>ps
process - kernel
process - rbexe.bin
process - ltexe.bin
Command>lbexe.bin
lbexe.bin starting...
lbexe.bin have started.
Command>_

CTRL + 3rd button enables mouse  IPS: 24.916M  A:  NUM  CAPS  SCRL
```

Если запустить все это под отладчиком, можно время от времени останавливать его и заметить, как меняется содержимое регистра **cr3**, а также сегментных регистров. Это свидетельствует о том, что многозадачность работает и что код выполняется как в режиме ядра так и в режиме пользователя:

Bochs Enhanced Debugger

Command View Options Help

Continue [c]			
Reg ...	Hex Value	Decimal	L
eax	0000009b	155	0
ebx	00000000	0	0
ecx	00000000	0	0
edx	00000000	0	0
esi	00000000	0	0
edi	00000000	0	0
ebp	00000000	0	0
esp	00201000	2101248	0
eip	00100007	1048583	0
eflags	00000282		0
cs	001b		0
ds	0023		0
es	0023		0
ss	0023		0
fs	0023		0
gs	0023		0
gdtr	00007fe0 (2f)		0
idtr	ffbf000 (800)		0
ldtr	0000		0
tr	d000		0
cr0	e0000011		0
cr2	00000000		0
cr3	03fdh000		0

Continue [c]			
Reg ...	Hex Value	Decimal	
eax	0000001f	31	^
ebx	00000000	0	
ecx	00000000	0	
edx	00000000	0	
esi	00000000	0	
edi	00000000	0	
ebp	00000000	0	
esp	00201000	2101248	
eip	00100009	1048585	
eflags	00000202		
cs	001b		
ds	0023		
es	0023		
ss	0023		
fs	0023		
gs	0023		
gdtr	00007fe0 (2f)		
idtr	ffbff000 (800)		
ldtr	0000		
tr	d000		
cr0	e0000011		v
cr2	00000000		
cr3	03fd2000		
<		>	

Continue [c]		
Reg ...	Hex Value	Decimal
eax	00000000	0
ebx	00001000	4096
ecx	00000000	0
edx	00000009	9
esi	80000003	2147483651
edi	ffbfafa0	4290752416
ebp	0000fe34	65076
esp	0000fe24	65060
eip	ffc02a83	4290783875
eflags	00000206	
cs	0008	
ds	0010	
es	0010	
ss	0010	
fs	0010	
gs	0010	
gdt	00007fe0 (2f)	
idt	ffbff000 (800)	
ldt	0000	
tr	d000	
cr0	e0000011	
cr2	00000000	
cr3	00001000	

Исходный

код:

<https://github.com/devsienko/c4os/tree/df29b051e89f38f7e4bb8c7c9877330e1957c233>.

Системные вызовы

Что это? Можно сказать это прослойка между кодом режима ядра и пользовательским кодом. Смотрите. Например у нас есть программа в user mode. Она хочет прочитать какой-то файл. Но чтение файла с диска это прерогатива кода режима ядра, у пользовательской программы просто нет на это привилегий. Поэтому надо каким то безопасным образом вызвать код режима ядра. Обычно для этого операционные системы имеют API, интерфейс, который может быть вызван из пользовательского кода. Одни из способов как это можно сделать являются прерывания (что-то вроде

прерываний BIOS'a). Например, прерывание с номером 0x80 (прямо как в Linux) отвечает за системные вызовы. То есть, чтобы инициировать системный вызов, нужно указать какой именно системный вызов мы хотим (указав его номер например в регистре `eax`), передать необходимые параметры через стек и инициировать прерывание.

Начнем с того, что добавим новый обработчик прерывания под номером 16 в файле `interrupts.c`:

```
void irq_handler(uint32 index, Registers *regs) {
    uint32 esp = (int)&regs;
    switch (index) {
        case 0:
            inc_pit_ticks();
            switch_task(regs, esp);
            break;
        case 1:
            keyboard_interrupt();
            break;
        case 6:
            i86_floppy_irq();
            break;
        case 16:
            syscall_handler(regs);
    }
}
```

В файле `syscall.c` определим список функций нашего интерфейса (на данный момент только одна функция, печать строки на экране):

```
static void *syscalls[1] =
{
    &printf,
};
```

Также добавим сам код обработчика (он же диспетчер):

```
void syscall_handler(Registers *regs)
{
    // Firstly, check if the requested syscall number is valid.
    // The syscall number is found in EAX.
    if (regs->eax >= num_syscalls)
```

```

    return;

    // Get the required syscall location.
    void *location = syscalls[regs->eax];

    // We don't know how many parameters the function wants, so we just
    // push them all onto the stack in the correct order. The function will
    // use all the parameters it wants, and we can pop them all back off afterwards.
    int ret;
    asm volatile (" \
        push %1; \
        push %2; \
        push %3; \
        push %4; \
        push %5; \
        call *%6; \
        pop %%ebx; \
        pop %%ebx; \
        pop %%ebx; \
        pop %%ebx; \
        pop %%ebx; \
        " : "=a" (ret) : "r" (regs->edi), "r" (regs->esi), "r" (regs->edx), "r" (regs->ecx), "r" (regs->ebx),
        "r" (location));
    regs->eax = ret;
}

```

Как написано в комментарии, мы не знаем сколько параметров будет принимать функция, которая будет обрабатывать системный вызов, поэтому ложим на стек все что можно, а функция сама разберется и возьмет, что ей надо.

Также я добавил команду **test**, что можно было поиграться с системным вызовом (из режима ядра в данном случае) и добавил системный вызов в файл **lbexe.asm** (тут уже вызов происходит из режима пользователя):

```

use32
org    0x100000 ; see USER_PROGRAM_BASE

mov    eax, 0
mov    ebx, boot_file_name
int    0x30

```

```
mov al, 75; capital K code
```

```
@ @:
```

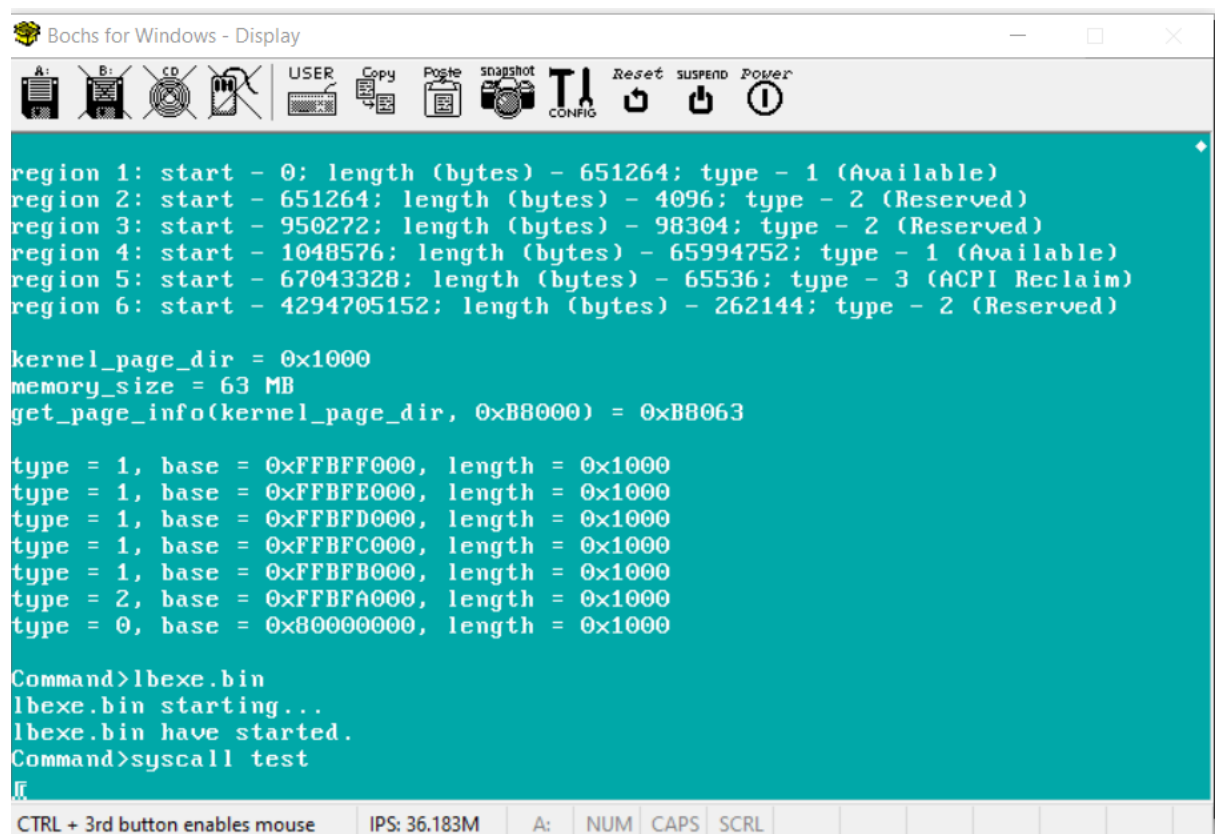
```
mov byte[0xB8000 + (24 * 80) * 2], al
```

```
inc al
```

```
jmp @b
```

```
boot_file_name db "syscall test",0
```

Компилируем, запускаем:



The screenshot shows the Bochs for Windows - Display window. The title bar includes standard window controls and a toolbar with icons for A:, B:, CD, USER, Copy, Paste, snapshot, CONFIG, Reset, SUSPEND, and Power. The main display area has a teal background and shows the following text:

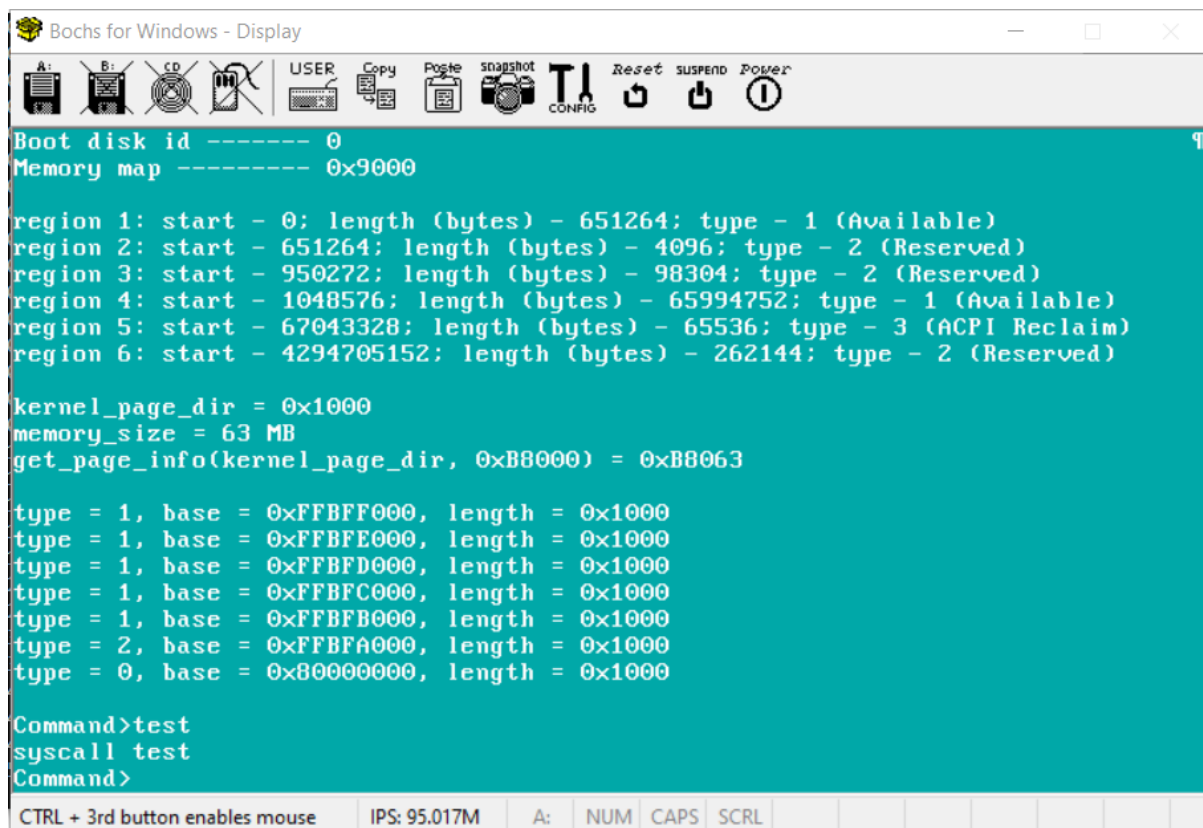
```
region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8063

type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 1, base = 0xFFBFB000, length = 0x1000
type = 2, base = 0xFFBFA000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>lbexe.bin
lbexe.bin starting...
lbexe.bin have started.
Command>syscall test
␣
```

At the bottom, a status bar displays: CTRL + 3rd button enables mouse | IPS: 36.183M | A: | NUM | CAPS | SCRL |



```
Bochs for Windows - Display
A: B: CD USER Copy Paste snapshot CONFIG Reset suspend Power
Boot disk id ----- 0
Memory map ----- 0x9000

region 1: start - 0; length (bytes) - 651264; type - 1 (Available)
region 2: start - 651264; length (bytes) - 4096; type - 2 (Reserved)
region 3: start - 950272; length (bytes) - 98304; type - 2 (Reserved)
region 4: start - 1048576; length (bytes) - 65994752; type - 1 (Available)
region 5: start - 67043328; length (bytes) - 65536; type - 3 (ACPI Reclaim)
region 6: start - 4294705152; length (bytes) - 262144; type - 2 (Reserved)

kernel_page_dir = 0x1000
memory_size = 63 MB
get_page_info(kernel_page_dir, 0xB8000) = 0xB8063

type = 1, base = 0xFFBFF000, length = 0x1000
type = 1, base = 0xFFBFE000, length = 0x1000
type = 1, base = 0xFFBFD000, length = 0x1000
type = 1, base = 0xFFBFC000, length = 0x1000
type = 1, base = 0xFFBFB000, length = 0x1000
type = 2, base = 0xFFBFA000, length = 0x1000
type = 0, base = 0x80000000, length = 0x1000

Command>test
syscall test
Command>
```

CTRL + 3rd button enables mouse IPS: 95.017M A: NUM CAPS SCRL

Исходный код:

<https://github.com/devsienko/c4os/tree/f4341d18e22dd583009281d06b69fda40396ed02>.

Вместо заключения

Все! Моя цель достигнута. Я разобрался как работает многозадачная операционная система, как работают переходы в режим ядра и обратно, системные вызовы да и с кучей всего остального. Надеюсь мой опыт поможет кому-то еще. Бай.

Евсиенко Даниил. 25 октября 2021.