Лабораторные работы по курсу
Операционные системы

Лабораторная работа 8
Загрузка операционной системы

Москва, 2024

Теоретическая часть

Загрузка операционной системы проходит в несколько этапов.

Первое, что запускает процессор при включении компьютера — это код BIOS (или же UEFI, но здесь я буду говорить только про BIOS), который "зашит" в памяти материнской платы (конкретно — по адресу 0xFFFFFFF0).

Сразу после включения BIOS запускает Power-On Self-Test (POST) — самотестирование после включения. BIOS проверяет работоспособность памяти, обнаруживает и инициализирует подключенные устройства, проверяет регистры, определяет размер памяти и так далее и так далее.

Следующий шаг — определение загрузочного диска, с которого можно загрузить ОС. Загрузочный диск — это диск (или любой другой накопитель), у которого последние 2 байта первого сектора (под первым сектором подразумевается первые 512 байт накопителя, т.к. 1 сектор = 512 байт) равны 55 и AA (в шестнадцатеричном формате). Как только загрузочный диск будет найден, BIOS загрузит первые его 512 байт в оперативную память по адресу 0x7c00 и передаст управление процессору по этому адресу.

Само собой, в эти 512 байт не выйдет уместить полноценную операционную систему. Поэтому обычно в этот сектор кладут первичный загрузчик, который загружает основной код ОС в оперативную память и передает ему управление.

С самого начала процессор работает в Real Mode (= 16-битный режим). Это означает, что он может работать лишь с 16-битными данными и использует сегментную адресацию памяти, а также может адресовать только 1 Мб памяти. Но вторым мы пользоваться здесь не будем. Картинка ниже показывает состояние оперативной памяти при передаче управления нашему коду.

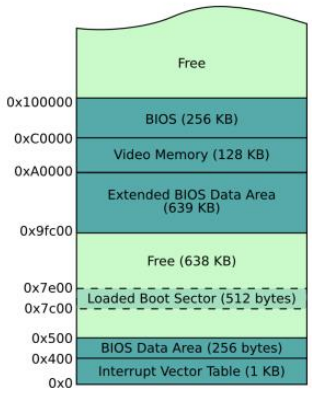


Figure 3.4: Typical lower memory layout after boot.

Последнее, о чем стоит сказать перед практической частью — прерывания. Прерывание — это особый сигнал (например, от устройства ввода, такого, как клавиатура или мышь) процессору, который говорит, что нужно немедленно прервать исполнение текущего кода и выполнить код обработчика прерывания. Все адреса обработчиков прерывания находятся в Interrupt Descriptor Table (IDT) в оперативной памяти. Каждому прерыванию соответствует свой обработчик прерывания. Например, при нажатии клавиши клавиатуры вызывается прерывание, процессор останавливается, запоминает адрес прерванной инструкции, сохраняет все значения своих регистров (на стеке) и переходит к выполнению обработчика прерывания. Как только его выполнение заканчивается,

процессор восстанавливает значения регистров и переходит обратно, к прерванной инструкции и продолжает выполнение.

Например, чтобы вывести что-то на экран в BIOS используется прерывание 0x10 (шестнадцатеричный формат), а для ожидания нажатия клавиши — прерывание 0x16. По сути, это все прерывания, что нам понадобятся здесь.

Также, у каждого прерывания есть своя подфункция, определяющая особенность его поведения. Чтобы вывести что-то на экран в текстовом формате (!), нужно в регистр AH занести значение 0x0e. Помимо этого, у прерываний есть свои параметры. 0x10 принимает значения из ah (определяет конкретную подфункцию) и al (символ, который нужно вывести). Таким образом,

```
mov ah, 0x0e
mov al, 'x'
int 0x10
```

выведет на экран символ 'x'. 0x16 принимает значение из ah (конкретная подфункция) и загружает в регистр al значение введенной клавиши. Мы будем использовать функцию 0x0.

Основы ассемблера

Микропроцессор содержит набор регистров:

Пользовательские регистры

- Регистры общего назначения
- Сегментные регистры
- Регистр флагов
- Регистр управления

Программная модель микропроцессора содержит 32 регистра, которые можно разделить на две группы:

- 16 пользовательских регистров;
- 16 системных регистров.

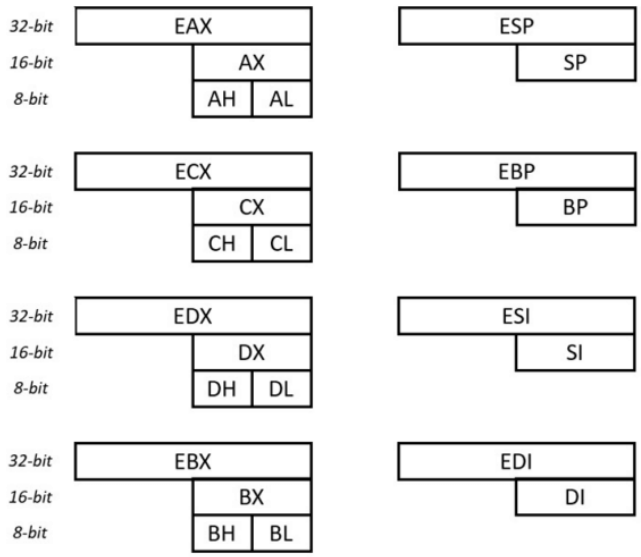
Пользовательскими регистры называются потому, что программист может их использовать при разработке программ.

К пользовательским регистрам относятся:

1. восемь 32-битных регистров, которые могут использоваться программистами для хранения данных и адресов (регистры общего назначения)
    a. `eax/ax/ah/al`
    b. `ebx/bx/bh/bl`
    c. `edx/dx/dh/dl`
    d. `ecx/cx/ch/cl`
    e. `esi/si`
    f. `edi/di`
    g. `ebp/bp`
    h. `esp/sp`
2. шесть регистров сегментов: `cs, ds, ss, es, fs, gs`

3. регистр состояния (регистр флагов) `eflags/flags`;
4. регистр управления (регистр указателя команды) `eip/ip`.

Обратим внимание, что к возможно обращаться к частям регистра.



Основные команды ассемблера:

Команда	Описание
`MOV <операнд₁>, <операнд₂>`	По команде MOV значение второго операнда записывается в первый операнд.
`ADD <операнд₁>, <операнд₂>`	Команда ADD складывает операнды и записывает их сумму на место первого операнда.
`SUB <операнд₁>, <операнд₂>`	Команда SUB вычитает из первого операнда второй и записывает полученную разность на место первого операнда.
`INC <операнд>`	Увеличить операнд на 1
`DEC <операнд>`	Уменьшить операнд на 1
`JMP <метка>`	Команда безусловного перехода
`CMP <операнд₁>, <операнд₂>`	Команда сравнения операндов
`JE <метка>`	Переход если равно
`JNE <метка>`	Переход если не равно

Видеоадаптер

При выводе текста различные видеосистемы работают одинаково. Для экрана (25 строк по 80 символов) отводится 4000 байт (по 2 байта на каждый символ). Первый (четный) байт содержит код ASCII, который аппаратно преобразуется в связанный с ним символ и посылается на экран. Второй (нечетный) байт содержит атрибуты. Это информация о том, как должен быть выведен символ.

Структура байта атрибутов:

биты 0-1-2	код цвета символа
бит 3	бит интенсивности
биты 4-5-6	код цвета фона знакоместа
бит 7	- 0 (мигание - 1)

Коды цветов:

0	черный
1	синий
2	зеленый
3	циан (голубой)
4	красный
5	магента
6	коричневый (с битом интенсивности - желтый)
7	белый

Практическая часть

1. Изучите программный код, приведенный ниже.

```
;
; Простой загрузочный сектор, который выводит сообщение на экран с помощью процедуры BIOS.
;
mov ah , 0x0e ; При значениях ah = 0x0E, int 0x10 печать происходит на текущей активной странице.
mov al , 'H'
int 0x10
mov al , 'e'
int 0x10
mov al , 'l'
int 0x10
mov al , 'l'
int 0x10
mov al , 'o'
int 0x10
jmp $ ; Бесконечный цикл
;
; Заполнение и магический номер BIOS.
;
times 510 -( $ - $$ ) db 0 ; Заполнить загрузочный сектор нулями
dw 0xaa55 ; последние два байта - магический номер, чтобы биос определил что это загрузочный сектор
```

Скомпилируйте код с помощью *nasm* и запустите его в эмуляторе qemu

```
nasm -f bin hello.asm -o os.img
```

Убедитесь, что на экране появилось сообщение «Hello»

Измените программу таким образом, чтобы на экран выводились ваши ФИО

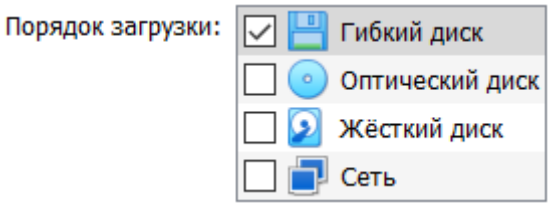
Запустите собранный образ диска на эмуляторе VirtualBox

Для этого выполните следующие действия:

1. Откройте программу VirtualBox
2. Создайте новую виртуальную машину (Кнопка «Создать»)
    a. Задайте имя машины – ваши инициалы.
    b. Тип: Other
    c. Версия: Other/Unknown
    d. Основная память: 4 Mb
    e. Процессоры: 1
    f. Виртуальный жесткий диск: не подключать
3. Зайдите в настройки созданной машины и перейдите в раздел «Носители»

a. Удалите все существующие носители и добавьте контроллер Floppy ( )
b. Выберите гибкий диск – созданный образ из предыдущего пункта

4. В разделе система установите Гибкий диск первым в порядке загрузки. Остальные галочки уберите.



5. Выйдите из настроек и запустите вашу операционную систему.

Если все сделано верно, то на экране появится сообщение.

2. Изучите программный код, приведенный ниже.

```
[BITS 16]
[ORG 0x7c00]
_start:
    cli
    mov ax, cs
    mov ds, ax
    mov ss, ax
    mov sp, _start

    ;; Загрузка регистра GDTR:
    lgdt [gd_reg]

    ;; Включение A20:
    in al, 0x92
    or al, 2
    out 0x92, al

    ;; Установка бита PE регистра CR0
    mov eax, cr0
    or al, 1
    mov cr0, eax

    ;; С помощью длинного прыжка мы загружаем
    ;; селектор нужного сегмента в регистр CS
    ;; (напрямую это сделать нельзя)
    ;; 8 (1000b) - первый дескриптор в GDT, RPL=0
    jmp 0x8: _protected



[BITS 32]
_protected:
    ;; Загрузим регистры DS и SS селектором
    ;; сегмента данных
    mov ax, 0x10
    mov ds, ax
    mov ss, ax

    mov ebx , Message
    call print_string_pm
    mov eax, $1
    mov ebx, $1
    add eax, ebx

    ;; Завесим процессор
    hlt
    jmp short $

Message db "Hello World", 0x0

%define VIDEO_MEMORY  0xb8000
%define WHITE_ON_BLACK  0x0f
```

```
; устанавливаем атрибуты символа.

; печатает строку, оканчивающуюся символом 0x0
print_string_pm :
    pusha
    mov edx , VIDEO_MEMORY ; Устанавливает в edx значение начала памяти видеобуфера

print_string_pm_loop :
    mov al , [ ebx ] ; Сохраняет значение символа из EBX в AL
    mov ah , WHITE_ON_BLACK ; Устанавливает атрибуты в AH
    cmp al , 0 ; если ( al == 0) , то жто конец строки ->
    je print_string_pm_done ; прыжок на окончание
    mov [ edx ] , ax ; Переместите символ и атрибуты в edx
    add ebx , 1 ; Увеличте ebx, чтобы обратиться к следующему символу
    add edx , 2 ; Перейдите к следующей ячейке в видеопамяти
    jmp print_string_pm_loop ; продолжаем цикл, пока не выведены все символы

print_string_pm_done :
    popa
    ret ; выход из функции

gdt:
    dw 0, 0, 0, 0   ; Нулевой дескриптор

    db 0xFF     ; Сегмент кода с DPL=0
    db 0xFF     ; Базой=0 и Лимитом=4 Гб
    db 0x00
    db 0x00
    db 0x00
    db 10011010b
    db 0xCF
    db 0x00

    db 0xFF     ; Сегмент данных с DPL=0
    db 0xFF     ; Базой=0 и Лимитом=4Гб
    db 0x00
    db 0x00
    db 0x00
    db 10010010b
    db 0xCF
    db 0x00



    ;; Значение, которое мы загрузим в GDTR:
gd_reg:
    dw 8192
    dd gdt

    times 510-($-$$) db 0
    db 0xaa, 0x55
```

Скомпилируйте и запустите программу и убедитесь, что на экране появилось сообщение.

Измените программу таким образом, чтобы на экран выводились ваши ФИО и номер группы. Установите цвет символа, чей номер равен вашему номеру в списке mod 8

Контрольные вопросы

1. Опишите процесс загрузки ОС
2. В чем отличие защищенного режима от реального?
3. Как работает видеоадаптер?

Список рекомендованной литературы

1. Карпов В.Е., Коньков К.А. Основы операционных систем. Москва: Физматкнига, 2019. 326 pp.
2. Таненбаум Э., Бос Х. Современные операционные системы. Санкт-Петербург: Питер, 2021. 1119 pp.
3. https://natalia.appmat.ru/c&c++/assembler.html