Projeto 1 - Bootloader

IF677 - Infra-Estrutura de Software
Centro de Informática - UFPE

Autor: Thyago Porpino (tnp)

Objetivos

- Desmistificar o processo de inicialização de um computador.
- Entender como um Sistema Operacional vem a ser executado.
- Introduzir programação em assembly.

Programação no Modo Real

O IBM PC original utilizava uma CPU Intel 8088 com um único modo de operação, modo esse que é mantido até hoje nos processadores, tudo para não quebrar a compatibilidade com os softwares da época.

Esse modo de operação, originário no 8086, é conhecido como *modo real* (16 bits), e contrasta com o *modo protegido* (32 ou 64 bits) que é usado por sistemas operacionais atuais. Todos os descendentes do IBM PC original realizam o boot no modo real, e é normalmente, trabalho do bootloader entrar no modo protegido e transferir controle para o kernel.

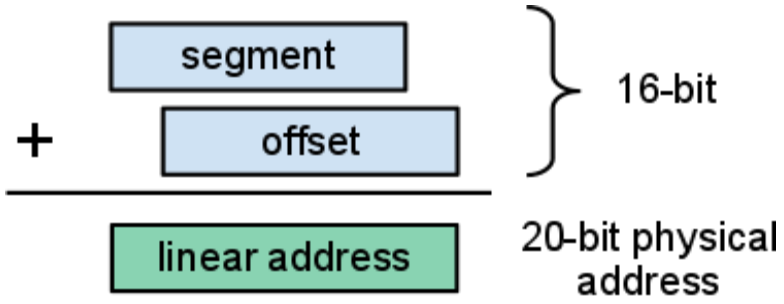
Em um ambiente sem um SO, como é o caso do ambiente encontrado por um bootloader, é necessário utilizar-se de serviços da BIOS para realizar as mais variadas funções (e.g. acessar o disco, imprimir na tela, etc).

Segmentação no Modo Real

O processador 8086 foi projetado para suportar um espaço de endereçamento de 20 bits, isto é, é capaz de endereçar 2^20 bits (1 MB). Já que o 8086 é formado por registradores de 16 bits, todos os acessos de memória suportam apenas operandos de 16 bits.

Para acessar todo o espaço de endereçamento (1 MB), em tese são necessários registradores de 20 bits, porém o 8086 só possuía registradores de 16 bits, então a técnica de segmentação foi utilizada, que combinava 2 registradores de 16 bits para que fosse possível o endereçamento de 1 MB de memória.

No modo real, os registradores de segmento contêm um endereço base de um segmento, e são usados para achar o endereço linear (físico) na memória, da seguinte forma:




```
(segment reg.)*16 + offset = linear address
```

```
OBS: Lembrando que multiplicar por 16 (decimal) é a mesma coisa que
fazer 4 shifts para a esquerda, ou 1 shift (em hexadecimal)
Exemplos:

      0x07c0 * 16 + 0x0 = 0x7c00
      0x0 * 16 + 0x7c00 = 0x7c00
```

Como uma alternativa de se especificar o endereço linear, alguns textos se referem aos endereços de memória no modo real usando a notação *segment:offset*. Por exemplo, o endereço linear (0x179b8) pode ser escrito como o par segment:offset 1234:5678.

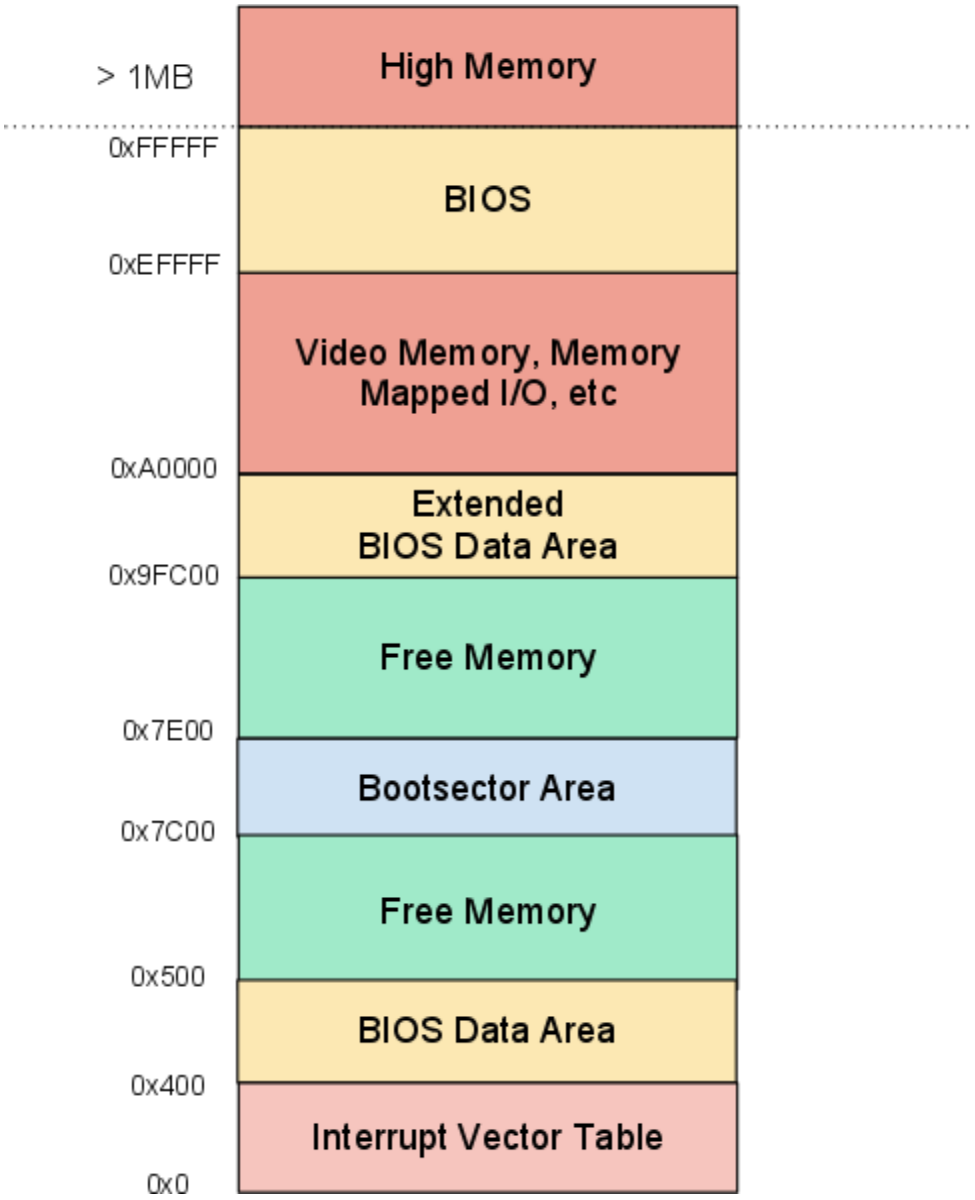
Como 12 bits da base do segmento e do endereço de offset se sobrepõe, é possível representar qualquer endereço linear (< 1 MB) no modo real através de mais de uma combinação segment:offset (e.g. 3000:1337 e 3123:0107 representam o endereço linear 0x31337).

Layout da memória física

O espaço de endereçamento do PC é dividido entre 2 regiões principais. A primeira região está abaixo de 1MB e corresponde à todo o espaço de endereçamento de 20 bits do IBM PC original e pode ser totalmente endereçada no modo real. A segunda região, é a memória acima dos primeiros 1 MB e é conhecida como high (ou extended) memory, e que só pode ser endereçada completamente no modo protegido.

Já que a lower memory corresponde à todo o espaço de endereçamento do IBM PC original, ela própria é dividida em subregiões. Historicamente, os primeiros 640KB da lower memory eram usados pelo SO (e.g. DOS), e por programas do usuário (se o modo real estiver sendo usado). Em contraste, os últimos 384KB da lower memory era usados para vídeo, I/O mapeado em memória, e a ROM (read-

only memory) da BIOS.



Interrupções no Modo Real

Em programação de sistemas, ao se utilizar algum serviço de baixo nível,

como leitura e escrita de arquivo, é sempre necessário se comunicar com o SO, pois ele é o único código com permissão de acessar diretamente o hardware (disco). Para isso, se faz uso de chamadas de sistema, que nada mais são do que a interface que o SO oferece aos programas de usuário. Mesmo quando se utiliza uma biblioteca como a libc (biblioteca padrão de C), e se chama funções como fopen (leitura de arquivo), na verdade se está implicitamente fazendo uma chamada de sistema (ou system call), pois a libc internamente faz essas chamadas.

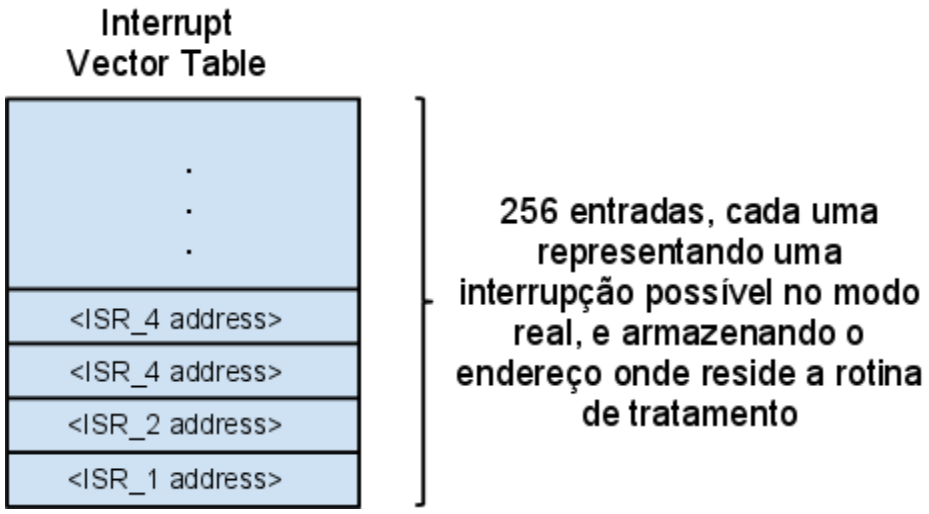
Na programação do modo real, especificamente no bootloader, ou seja quando não há um SO que ofereça essas chamadas de sistema, é preciso usar as interrupções da BIOS.

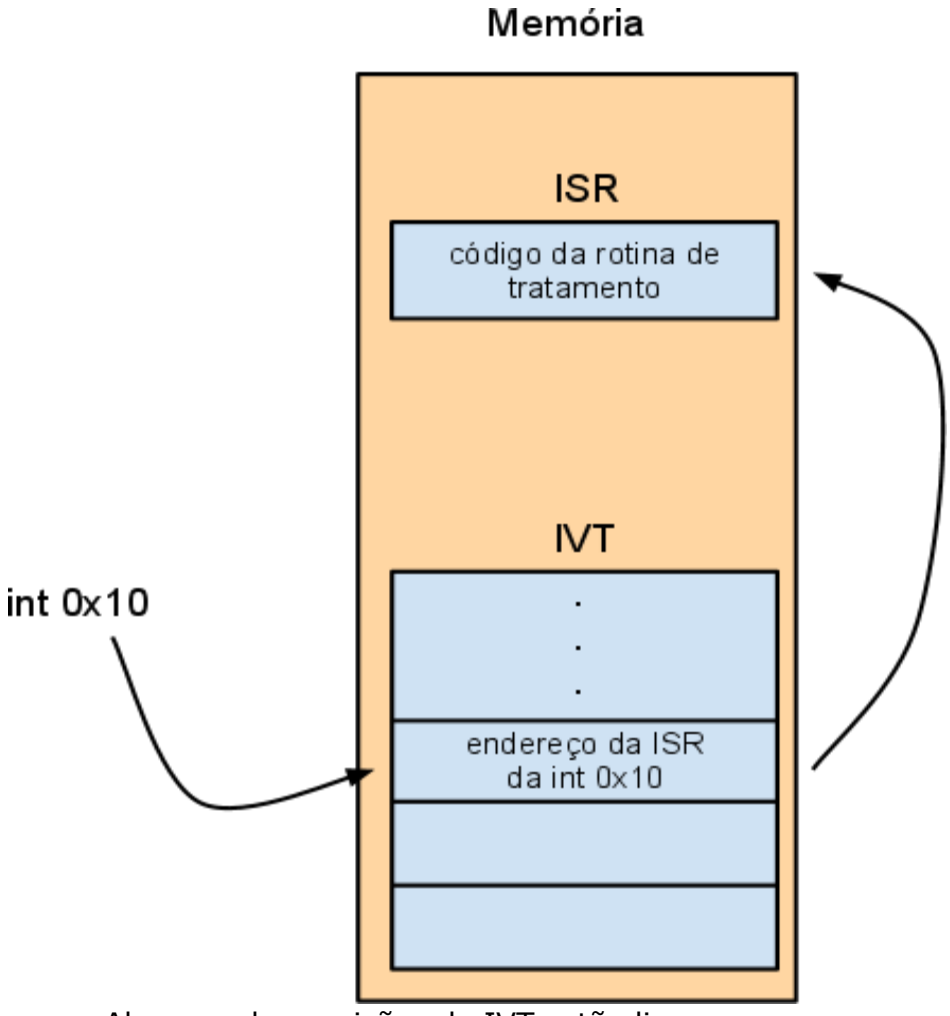
Como já foi mostrado, dentro da memória, existe uma região que se chama IVT (Interrupt Vector Table), e que guarda os endereços da onde se encontram as ISR (Interrupt Service Routine), ou seja, as rotinas de tratamento para cada interrupção. A IVT, possui tanto interrupções de hardware (e.g. divisão por zero), como interrupções de software (da BIOS). Ao se chamar uma interrupção da BIOS:

```
int 0x10
```

Ocorre o seguinte:

1. 0x10 é a décima interrupção, então esse número é multiplicado por 4, pois cada elemento no vetor possui 4 bytes.
2. O endereço 0x40 é acessado na memória, e é obtido o endereço para a rotina de tratamento dessa interrupção específica.
3. Pula-se para esse endereço.
4. A rotina de tratamento é executada.
5. O controle é retornado para o programa.





Algumas das posições da IVT estão livres caso o programador queira

implementar sua própria interrupção, e pode-se até mesmo modificar interrupções já existentes, bastando-se alterar o endereço que se encontra numa das entradas da IVT para apontar para um código definido pelo programador.

BIOS

A BIOS (Basic Input/Output System) consiste de código armazenado em memória não-volátil na placa-mãe de um PC. A BIOS é responsável por iniciar os serviços básicos de um computadorAlém disso, a BIOS oferece um conjunto de interrupções (parecidas com chamadas de sistema de um SO) que estão disponíveis aos programas no modo real. Essas interrupções oferecem uma maneira de acessar dispositivos comuns de um PC, como o hardware de vídeo e o controlador de disco. Sistemas operacionais de modo real como o DOS, utilizam esses serviços para se comunicar com o hardware. Como, no geral, a BIOS não está disponível no modo protegido, SOs de modo protegido precisam

Bootloader

O projeto consiste em desenvolver um bootloader de 2 estágio, que chame um kernel feito em assembly, tudo desenvolvido no modo real.

Primeiro Estágio

A BIOS carrega o primeiro setor do disco (se ele tiver a assinatura de boot 0xAA55), no endereço linear de memória 0x7c00, e pula para esse endereço. Como um setor no disco geralmente possui 512 bytes, muitas vezes divide-se o bootloader em mais de um estágio, de modo que a BIOS carregue o primeiro estágio (512 bytes), e este carregue o próximo estágio. Portanto nesse primeiro estágio os alunos devem escrever um código que carregue o segundo estágio do bootloader do disco para a memória, escolhendo qualquer endereço livre da memória para carregar o mesmo.

Segundo Estágio

O segundo estágio de um bootloader normalmente carrega algumas estruturas que serão úteis ao kernel na memória, carrega o kernel na memória, configura um ambiente adequado ao kernel (e.g. passar para o modo protegido), e passa o controle para o kernel. No segundo estágio os alunos devem escrever um código que imprima na tela as tarefas comuns do segundo estágio (mesmo que na prática elas não sejam implementadas no projeto), exemplo:

```
Loading structures for the kernel...
Setting up protected mode...
Loading kernel in memory...
Running kernel...
```

Esse código deve também, carregar o kernel na memória e passar o controle para ele.

Kernel

Em um SO moderno, o kernel possui um código gigantesco (e.g. 10 milhões de linhas), e oferece várias funcionalidades, como:

- Gerenciamento de Recursos (memória, processador, etc).
- Gerar uma camada de abstração para os software de usuário.

Nessa parte do projeto os alunos deverão fazer um código que imprima na tela o nome do SO desenvolvido pelo grupo, exemplo:

```
Starting CheetOS!
```

Os alunos podem exercitar sua criatividade nessa parte do projeto, escolhendo cores para essa tela de abertura, ou talvez algo mais complexo. Qualquer trabalho extra que o grupo faça, será levado em consideração na hora da avaliação.

FAQ - Frequently Asked Questions

1) Pra que serve a diretiva ORG no nasm?

Ela é usada em programas no modo real, para somar um número fixo ao offset, na

hora do cálculo do endereçamento segment:offset, de forma que fique mais fácil gerenciar os registradores de segmento. Por exemplo, caso se utilize o ORG no começo de todos os programas no modo real, usando seu endereço de carregamento na memória como parâmetro (e.g. org 0x7c00), pode-se simplesmente zerar o registrador de segmento DS, pois ao tentar acessar strings presentes no programa, ela será encontrada na memória, sem problemas. Caso contrário, se a diretiva ORG não for utilizada, deve-se setar o DS para um valor que aapós o shift realizado no cálculo do endereço, ele equivale ao endereço onde o programa foi carregado.

2) Qual a diferença entre instrução, diretiva e macro?

Uma instrução é algo que pode ser interpretado pelo processador:

```
mov ax, 1
add bx, 1
```

Uma diretiva é uma espécie de declaração que pode ser interpretada por um assembler específico:

```
org 0x7c00
bits 16
```

Macro é uma forma de substituição de código que pode ser definida pelo usuário ou pelo próprio assembler, e que pode ser usado como uma espécie de função, ou como forma de dar outro nome à algo. As macros são resolvidas na passagem do pré-processador. (O uso de macros é desencorajado em projetos pequenos feitos em assembly. E, em linguagens de alto nível como C, macros são consideradas uma má prática de programação, exceto em casos onde elas realmente são necessárias).

```
%define param(a,b) ((a)+(a)*(b)
```

3) Quais são e para que servem os registradores de segmento?

Os registradores de segmento importantes para esse projeto são o CS (code segment) e o DS (data segment). Sempre que se faz um fetch de instrução, se utiliza o registrador CS no cálculo do endereço, e sempre que se faz algum acesso à dados (acessar algo na memória), o DS é utilizado. Portanto, é importante setar corretamente esses 2 registradores, por exemplo, o CS pode ser setado através de um far jump no início do programa da seguinte forma:

```
org 0x7c00
jmp 0x0000:start
start:
    ...
```

Após o far jump, nesse caso, pode-se ter certeza que CS = 0x0. Setar o CS é uma boa prática no desenvolvimento de bootloaders, pois algumas BIOS antigas ao pular para o bootloader, utilizam o endereço 0x07c0:0x0000, que é avaliado corretamente para o endereço linear 0x7c00, porém faz com que CS seja igual à 0x07c0.

Para setar DS, deve-se utilizar registrador AX (acumulador), exemplo:

```
xor ax, ax
mov ds, ax
```

4) Para que serve a diretiva BITS 16? Preciso usá-la?

A diretiva BITS 16, serve para informar ao nasm que o código deverá ser usado no modo real, e consequentemente o binário gerado deve estar no formato bin (sem estrutura). Essa diretiva é desnecessária no projeto pois como default o nasm gera binários no formato bin, e só não o faz quando outro formato é especificado na linha de comando:

```
nasm -f elf boot.asm
```

Referências

Desenvolvimento de SO e bootloaders
http://wiki.osdev.org/
http://www.brokenthorn.com/Resources/

Endereçamento Segment:Offset
http://thestarman.pcministry.com/asm/debug/Segments.html

Nasm (assembler)
www.nasm.us