CRIANDO UM SISTEMA OPERACIONAL BÁSICO

Por: Fernando Birck aka Fergo

SUMÁRIO

1. INTRODUÇÃO

Neste tutorial, vou ensinar os passos básicos para criar um mini sistema operacional de 16 bits para x86, inteiramente programado do zero, em Assembly. Não vou ensinar nada sobre a linguagem de programação em si, então é recomendável ter algum conhecimento sobre ela.

Vamos fazer algo bem básico, mas suficiente para você entender como é o funcionamento de um SO. Faremos apenas uma mensagem de texto ser exibida na tela. Parece pouco, mas verá que a coisa não é tão simples assim.

Veja os aplicativos necessários no próximo capítulo e boa sorte.

2. APLICATIVOS

Para programar o sistema, vamos usar o Emu8086, um ótimo emulador de 8086, com capacidade de compilar, fazer debug e, claro, emular o sistema (para não ter que ficar reiniciando o computador toda hora para testar). Baixe-o endereço abaixo:

Emu086
http://www.emu8086.com

Ele não é gratuito, tem um prazo de 90 dias para testar, mas é suficiente para o nosso tutorial. Em seguida, baixe mais 2 aplicativos, que vamos usar para gravar nosso SO em um disquete e dar o boot por ele:

Fergo RawImage Maker
http://www.fergonez.net/download.php?file=frim.zip

RawWriteWin
http://www.chrysocome.net/rawwrite

O local de instalação desses aplicativos fica a sua escolha, não tem um local específico para instalar. Vamos em frente, para uma breve explicação sobre o processo de boot.

3. PROCESSO DE BOOT

Geralmente, após o término da checagem de Hardware, o computador busca pelos 512 bytes gravados no primeiro setor do disquete (*Cabeça: 0, Trilha: 0, Setor: 1*). Caso não encontre, ele busca por um sistema operacional na MBR (*Master Boot Record*) do seu HD.
É importante que, para testar o SO, você configure a BIOS para "bootar" o disquete antes de qualquer outro dispositivo (HD, CDROM, USB, etc...).

Se ele encontrar algum sistema nesses 512 bytes do disquete, ele o carrega na memória no endereço *0000:7C00h*. Vamos ver como funciona esses endereços de memória.
Quando se usa o formato *xxxx:yyyy*, você está trabalhando com endereços relativos, sendo que a primeira seqüência representa o que chamamos de *Segment* e a segunda, *Offset*. Para calcular o endereço físico, real, é feito o seguinte cálculo (lembre-se que estamos trabalhando com números hexadecimais, indicado pelo 'h' após o último algarismo):

*Segment * 16h + Offset = Endereço físico*

Tomando como exemplo o local onde a *BIOS* carrega o sistema operacional, podemos calcular o endereço físico real através do cálculo:

*0000h * 16h + 7C00h = 7C00h*

Nosso sistema é carregado no endereço físico da memória *7C00h*. É bom lembrar que diferentes *segments* e *offsets* podem gerar o mesmo endereço físico. Por exemplo:

0000:7C00h = 07C0:0000h

Qual a importância de saber sobre esses endereços? Bom, é por eles que você vai controlar seu programa. Você pode usar endereços físicos diretamente, mas usando no formato de *segment/offset* você consegue organizar melhor as posições de memória.

O endereço físico também é importante para saber onde eu posso e onde eu não posso gravar os dados na memória. A *BIOS* reserva um trecho de memória baixa (*640KB*) para que você use-o livremente. Esse trecho vai do endereço físico *00500h* até *A0000h*. Ou seja, você não deve gravar nada antes do endereço *00500h* e nem após *A0000* (são locais reservados para memória de vídeo, bios, vetores de interrupts (veja adiante), etc...).

Voltando ao boot. Eu mencionei que ele carrega os 512 bytes do primeiro setor na memória. Certo, e se meu SO tiver mais de 512 bytes? Aí nos vamos precisar de um *Loader*, que é basicamente uma seqüência de instruções (com no máximo 512 bytes para caber no primeiro setor), que é responsável por carregar o *Kernel* do disco para a memória. Vamos usar um para o nosso SO (apesar de não precisar nesse caso).

4. INTERRUPTS

Eu disse na introdução que não iria explicar sobre a linguagem *Assembly*, mas acho que *Interrupts* é algo importante para ressaltar.

Interrupt, no caso do Assembly, é uma instrução que paralisa o código atual para que alguma ação seja realizada (chamamos isso de *IRQ – Interruption Request*). Se for para comparar com algo nas linguagens de alto nível, podemos comparar ele com uma função, que é chamada, executa suas instruções, e depois retorna para o código onde ela foi chamada.

Todo computador na arquitetura x86 possui diversos interrupts, controlados pela *BIOS*. Os interrupts são compostos por uma função e uma subfunção. Por exemplo, para trabalhar com o vídeo, é usado o interrupt *10h*. E a operação a ser realizada no vídeo (subfunção), depende do valor de algum registrador (normalmente *AH*). Veja o exemplo abaixo que imprime um caractere na tela:

```
mov ah, 0Eh ;subfunção que indica para imprimir texto
mov al, 'A' ;caractere a ser impresso
int 10h     ;interrupção de vídeo
```

Como eu sei o que cada interrupt faz, quais os argumentos e quais registradores estão envolvidos? Eu uso este site:

http://www.htl-steyr.ac.at/~morg/pcinfo/hardware/interrupts/inte1at0.htm

Vamos usar Interrupts para imprimir caracteres na tela, para buscar por teclas pressionadas, para alterar o modo de vídeo, etc. Dá pra notar que entender o seu funcionamento é fundamental para prosseguir com este tutorial.

Vamos por a mão na massa agora.

5. CRIANDO O LOADER

O *Loader* basicamente consiste num conjunto de instruções que devem caber no primeiro setor do disquete (512b) e que lê os outros setores do disquete (onde está o *Kernel* e o resto do código) para a memória.

O nosso código precisa configurar algumas coisas básicas para que tudo funcione corretamente. Precisamos ter uma pilha de dados (*Stack*). Se você programa em Assembly, deve saber o que é uma pilha, como ela funciona, e para que é usada.
Os registradores envolvidos com a pilha são os seguintes:

SS -> Stack Segment -> Aponta para o segmento onde está a pilha
SP -> Stack Pointer -> Aponta para determinada região da pilha (normalmente o topo)

Além da pilha, é necessário indicar onde está o nosso segmento de dados

DS -> Data Segment -> Aponta para a base de dados (usado sempre que for acessar algum endereço de memória).

Vamos organizar nossa memória da seguinte maneira:

Memória	Descrição
07C0:0000 até 07C0:01FF	Local onde foi carregado o bootloader
07C0:0200 até 07C0:03FF	Pilha
0800:0000 em diante	Nosso Kernel

Essa estrutura da memória é a mesma utilizada no site do *Emu8086*, pois é bem didática e segue uma seqüência lógica. Claro que vai ficar um bom trecho sobrando (de 0500 até 7C00), mas por enquanto é suficiente.

Então eis a seqüência que vamos usar no nosso *Loader*

- Determina a pilha e seus registradores
- Indica o segmento de dados
- Altera o formato de vídeo para 80x25 (80 colunas, 25 linhas)
- Lê o setor do disquete onde está o *Kernel*
- Escreve os dados lidos no endereço 0800:0000
- Pula para este endereço e passa o controle para o *Kernel*

Abra o *Emu8086*, selecione *'New'* e marque a opção *'Empty Workspace'*. Em seguida, cole o código da página seguinte (vou fazer os comentários sobre o que cada instrução faz no próprio código)

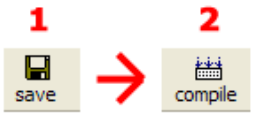
```
org 7C00h                          ;organiza o offset

;inicialização da pilha
mov     ax, 07C0h
mov     ss, ax                     ;seta o SS para 07C0h
mov     sp, 03FEh                  ;aponta para o topo da pilha

;seta segmento de dados
xor     ax, ax                     ;zera AX
mov     ds, ax                     ;seta o segmento de dados par 0000h

;altera o modo de vídeo
mov     ah, 00h                    ;subfucao para setar modo de vídeo
mov     al, 03h                    ;03h = 80x25, 16 cores
int     10h                        ;interrupt de vídeo

;le dados do disquete
mov     ah, 02h                    ;subfunção de leitura
mov     al, 1                      ;numero de setores para ler
mov     ch, 0                      ;trilha ( cylinder )
mov     cl, 2                      ;setor
mov     dh, 0                      ;cabeça
mov     dl, 0                      ;drive ( 00h = A: )
mov     bx, 0800h                  ;ES:BX aponta para o local da memória_
mov     es, bx                     ;onde vai ser escrito os dados_
mov     bx, 0                      ;0800:0000h ( ES = 0800h, BX = 0000h )
int     13h                        ;interrupt de disquete

jmp     0800h:0000h                ;pula para o local onde está o kernel
                                   ;e passa a execução para ele
```

Observações:

O número de setores a serem lidos varia com o tamanho do *Kernel*. Cada setor tem 512 bytes. Então se o *Kernel* tiver 512 bytes ou menos, basta ler 1 setor. Se tiver 700 bytes, precisa ler 2 setores e assim por diante.

Caso tenha surgido alguma dúvida quanto aos valores, vá até o site com a lista de interrupts que eu indiquei e analise a Int em questão.

Salve este código e compile através do botão *'compile'* na barra de ferramentas. Ele vai perguntar onde você deseja salvar o arquivo (que vai ter a extensão *.bin*). Dê um nome qualquer (algo como *loader.bin*) e salve no diretório que desejar.



6. CRIANDO O KERNEL

Nosso *Kernel* vai ser o mais simples possível. Vai apenas escrever um texto na tela e aguardar que o usuário pressione alguma tecla para reiniciar o computador. Parece pouco, mas é suficiente pra você entender como um sistema básico funciona. Se precisar que o *Kernel* faça algo mais (claro que vai querer), você já vai ter conhecimento suficiente para poder usar outros interrupts e trabalhar melhor com a memória, etc.

```
org 0000h                          ;organiza o offset

push cs                            ;CS = endereço do programa atual
pop ds                             ;DS = CS

call clearscreen                  ;chama procedure de limpar a tela

lea si, Mensagem                   ;SI = endereço da mensagem
mov ah, 0Eh                        ;subfuncao para imprimir caractere

repetição:

    mov al, [si]                   ;move para AL o caractere em SI
    cmp al, 0h                     ;compara com 0 ( fim da string )
    jz terminou                    ;caso terminou, pule para 'terminou'
    int 10h                        ;interrupção de video
    inc si                         ;próximo caractere
    jmp repetição                  ;repete o processo ate achar o 0

terminou:

mov ah, 0h                         ;subfuncao de aguardar tecla
int 16h                            ;interrupção de teclado

mov ax, 0040h                      ;método de reboot consiste em setar_
mov ds, ax                         ;o valor do endereço 0040:0072h_
mov w.[0072h], 1234h               ;para 1234h e pular para o endereço_
jmp 0FFFFh:0000h                   ;FFFF:0000h

clearscreen proc                   ;procedure de limpar a tela
    pusha                          ;coloca todos os reg na pilha

    mov     ah, 06h                ;subfuncao de rolar a tela pra cima
    mov     al, 0                  ;limpa a tela
    mov     bh, 0000_1111b         ;seta as cores ( fundo_texto )
    mov     ch, 0                  ;linha do canto sup. esq.
    mov     cl, 0                  ;coluna do canto sup. esq.
    mov     dh, 19h                ;linha do canto inf. dir. ( 25 )
    mov     dl, 50h                ;coluna do canto inf. dir. ( 80 )
    int     10h                    ;interrupção de vídeo

    popa                           ;repõe os valores dos registradores
    ret                            ;retorna para o código
clearscreen endp

Mensagem db 'Meu primeiro SO',0 ;nossa string que vai ser exibida
```

Novamente, se tiver alguma dúvida quando a esse código (que não seja relativo a sintaxe e os comandos do Assembly), volte ao site com a lista de interrupts.

Depois de pronto, salve o arquivo e compile, da mesma forma como fez no *Loader*. Claro, escolha outro nome (*kernel.bin* talvez). Procure manter os arquivos em uma mesma pasta, pra manter organizado.

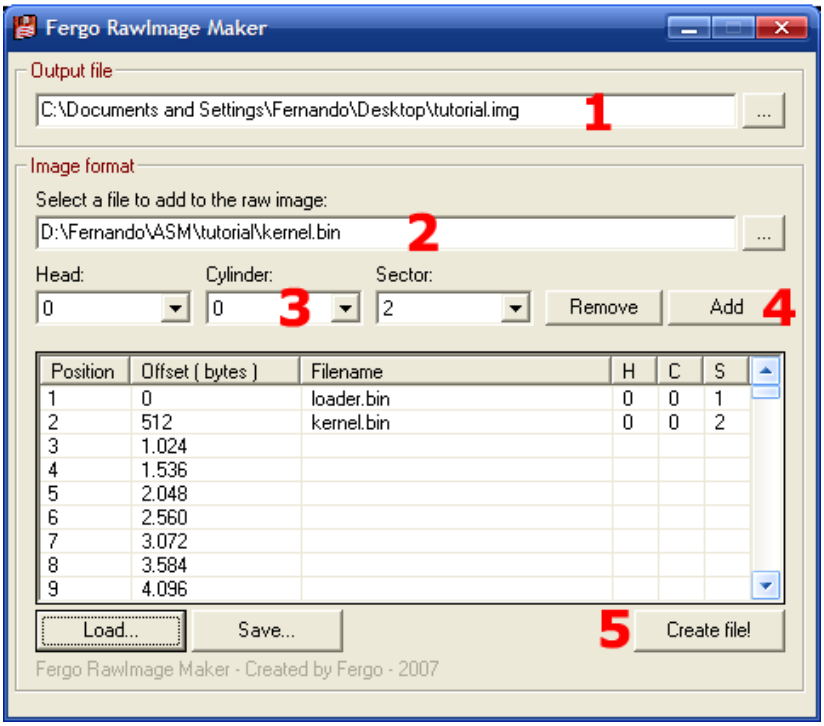
Nosso mini sistema operacional está pronto. Só falta grava-lo no disquete e testar.

7. GRAVANDO E TESTANDO

Com os binários do *Loader* e do *Kernel* em mãos, vamos criar uma imagem para ser gravada no disquete. Vamos usar uma ferramenta que eu programei, chamada *Fergo RawImage Maker.*

OBS.: Se precisar, baixe as VB6 Runtime Libraries

Execute o programa. A interface é bem simples e intuitiva. Siga estes passos para criar a imagem (os números em parênteses fazem relação com a imagem).

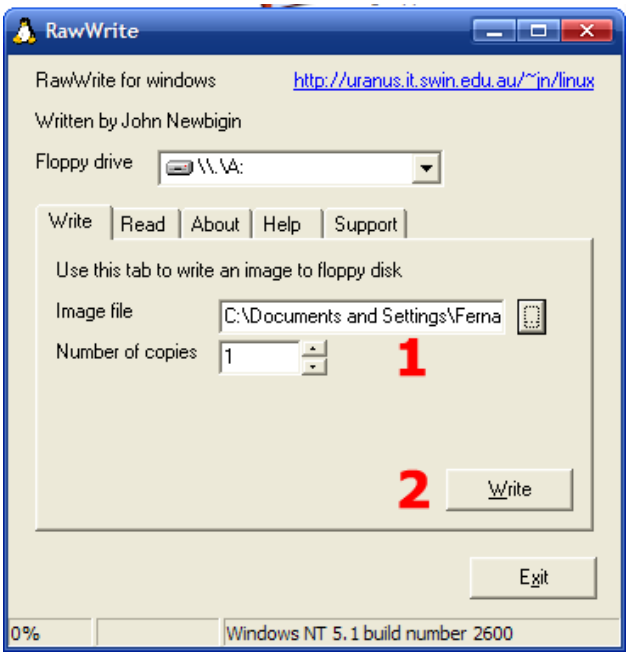


- Escolha o arquivo de destino (1)
- Selecione o *loader.bin* na segunda caixa de texto (2)
- Marque *Head* = 0, *Cylinder* = 0 e *Sector* = **1** (3)
- Clique em *Add* (4)
- Selecione o *kernel.bin* na segunda caixa de texto (2)
- Marque *Head* = 0, *Cylinder* = 0 e *Sector* = **2** (3)
- Clique em *Add* (4)
- E em seguida clique em '*Create File!*' (5)

Preste bastante atenção nos setores que você for gravar. Verifique se o resultado final ficou semelhante ao da lista na imagem.

Se tudo ocorreu bem, deve ter aparecido uma mensagem indicando que o arquivo foi criado com sucesso.

Agora temos que gravar a nossa imagem no disquete. Insira um disquete de 3.5” no drive *A:* e abra o programa *RawWriteWin*. Configure o caminho para a imagem que você acabou de criar (*tutorial.img* no meu caso) e clique em ‘*Write*’.



Se o disquete estiver em bom estado e você seguiu os passos corretamente, deverá receber novamente uma mensagem indicando que a imagem foi gravada com sucesso no disquete (certifique de que ele não está protegido contra gravação).

Agora basta testar. Coloque o disquete no drive, reinicie o micro, configure para que a *BIOS* dê o boot pelo disquete e se deu tudo certo, você vai ver a seguinte mensagem após o boot:

Se esta mensagem apareceu, parabéns, você fez tudo corretamente. Caso ela não tenha aparecido, releia o tutorial que com certeza você encontrará o local onde está o erro.

8. CONSIDERAÇÕES FINAIS

É isso aí. Espero ter ajudado aqueles iniciantes (assim como eu) que sempre tiveram vontade de saber como fazer um sistema operacional e como ele se comporta. Claro que este foi o exemplo mais básico, apenas o ponta pé inicial, mas com certeza isso já é suficiente para você avançar mais um pouco.
Sugiro tentar implementar agora alguns comandos no seu SO (assim como eu fiz nos meus experimentos) como '*Clear Screen'*, *'Reboot'*, *'Help'*, etc.

Boa sorte e até a próxima!

Fernando Birck aka Fergo
Website: www.fergonez.net