Fazendo um kernel simples em C com as funções printf e clearscreen

by Joachim Nock and K.J.

Original em :

http://www.osdever.net/tutorials/basickernel.php?the_id=12

Traduzido por : Ataliba Teixeira < ataliba@ataliba.eti.br >

Este tutorial tem como objetivo apresentar como desenvolver um *kernel simples. Comecemos com um exemplo de entrada de kernel, que está no arquivo kernel_start.asm.*

```
[BITS 32]
[global start]
[extern _k_main] ; this is in the c file

start:
  call _k_main

  cli  ; stop interrupts
  hlt ; halt the CPU
```

Okay, este é um código 32 bits (a entrada [BITS 32] mostra isto) e chama um função k_main, que estará definida em um arquivo C chamado kernel.c . Você pode estar se perguntando porque a função chama-se k_main no arquivo C e tem o nome de _kmain no assembly. Isto aconte porque os compiladores C/C++ acrescentam um underscore (_) em frente ao nome das funções C/C++, a menos que você as una a um arquivo ELF. O arquivo ELF não precisa do underscore. Quando k_main é chamado, a instrução *cli* é executada. *Cli* desliga as interrupções (apesar das mesmas nunca estarem “habilitadas” neste exemplo). Então, *hlt* é executada e diz a CPU para parar as execuções. Nós poderiamos executar um jmp ao invés do *hlt* mas isto iria gerar uma carga muito alta a CPU. Note que uma interrupção pode anular o comando *hlt* , reavivando a CPU. Isto não acontece, devido a termos desabilitado as interrupções antes de executarmos o *hlt* , para termos uma parada completa da CPU.

Agora, vamos dar uma olhada nas definições e os protótipos de funções no início do arquivo kernel.c .

```
#define WHITE_TXT 0x07 // white on black text

void k_clear_screen();
unsigned int k_printf(char *message, unsigned
int line);
void update_cursor(int row, int col);
```

Note que não há nada de especial aqui, exceto por #define WHITE_TXT 0x07. Em breve voltaremos aqui, por hora, só é necessário saber que isto está aqui.

Agora, vamos dar uma olhada na função k_main.

```
k_main() // like main in a normal C program
{
        k_clear_screen();
        k_printf("Hi!\nHow's this for a starter OS?", 0);
};
```

O entry point de nosso kernel (ponto de entrada) é a função k_main, que é chamada no arquivo *kernel_start.asm*.
A função *k_clear_screen* faz o que já se imagina pelo nome, limpa a tela. k_printf("Hi!\nHow's this for a starter OS?", 0); mostra na tela o seguinte texto :

Hi!
How's this for a starter OS?

O texto se inicia na primeira linha da memória de vídeo (0 é a primeira linha, 1 é a segunda linha, 2 é a terceira linha e assim por diante). O símbolo *\n* especifica uma nova linha, tal qual no *printf* em C/C++.

No modo protegido, não podemos chamar as interrupções de BIOS para limpar a tela, portanto, temos que fazer isto nós mesmos escrevendo diretamente na memória de vídeo.

```
void k_clear_screen() // clear the entire text screen
{
        char *vidmem = (char *) 0xb8000;
        unsigned int i=0;
        while(i < (80*25*2))
        {
                vidmem[i]=' ';
                i++;
                vidmem[i]=WHITE_TXT;
                i++;
        };
};
```

Nesta função (*k_clear_screen*), o ponteiro *vidmem* aponta para o início da memória de vídeo em modo protegido, que é 0xb8000. Nós declaramos o ponteiro como um char, assim poderemos escrever um byte de cada vez na memória de vídeo. O modo texto na arquitetura x86 possui 80x25 caracteres e cada caracter necessita de 2 bytes. O primeiro byte é o caracter, e o segundo é o atributo que controla a cor e se vai piscar ou não. Então, nós multiplicamos 80*25 (para se ter o número de caracteres que pode ser mostrado na tela) e multiplicamos por 2 para que possamos ter acesso a memória byte a byte. O loop se auto-explica, the *vidmem[i]=' ';* escreve uma espaço na memória de vídeo (*i* aponta para um campo específico da string). Adicionamos 1 a *i,* através de *i++,* para pegar o próximo byte na memória de vídeo (*o byte de atributo*) e nele colocamos *0x07*. *0X07* especifica um fundo de tela preto, com letras brancas que não piscam.

Agora, vamos analisar a funçao *k_printf* !!!

```
unsigned int k_printf(char *message, unsigned int line)
// the message and then the line #
{
        char *vidmem = (char *) 0xb8000;
        unsigned int i=0;

        i=(line*80*2);

        while(*message!=0)
        {
                if(*message=='\n') // check for a new line
                {
                        line++;
                        i=(line*80*2);
                        *message++;
                } else {
                        vidmem[i]=*message;
                        *message++;
                        i++;
                        vidmem[i]=WHITE_TXT;
                        i++;
                };
        };

        return(1);
};
```

A função *k_printf* é bem parecida com a função *k_clear_screen* em suas funcionalidades. O loop *while(*message!=0)* acontece enquanto não achamos o fim da string que foi passada para a função. O comando *if(*message='\n')* testa se a string é um caracter de nova linha ... e se for, adiciona 1 a linha e os caracteres depois do *n* irão ser impressos na próxima linha. Se o caracter não for o *n,* simplesmente o caracter é colocado na memória de vídeo e o atributo é setado para 0x07 (fundo de tela preto com letras brancas e que não piscam).

Compilando o kernel

Primeiro, você deve fazer o download do código fonte do kernel[1]. Nós iremos precisar de um assembler (NASM [2]), um compilador C (DJGPP[3] ou gcc[4]) e um linker (ld[5]).

Agora, no início do arquivo de linker, você verá a seguinte linha :
.text 0x100000
Este número hexadecimal deve ser setado quando o kernel for carregado na memória. Neste caso, ele está na marca de 1M da memória (0x100000 em hexadecimal).

Agora, vamos compilar nosso código assembly :

```
nasm -f aout kernel_start.asm -o ks.o
```

Isto compila o *kernel_start.asm* em um arquivo *ks.o* em um formato *aout* compatível. Agora, para o nosso arquivo C :

```
gcc -c kernel.c -o kernel.o
```

O próximo e último passo é linkar o arquivo *ks.o* e o *kernel.o* em um só arquivo. Neste caso, iremos linkar os dois em um só arquivo com o linker *ld.* Nós iremos efetuar isto com o seguinte comando :

```
ld -T link.ld -o kernel.bin ks.o kernel.o
```

É importante que o arquivo *ks.o* seja linkado primeiramente ou o kernel não vai funcionar. O kernel agora é chamado kernel.bin e está pronto para ser executado por um bootsector/loader que chama o modo protegido e habilita o A20 (O bootsector do John Fine[6] faz isto). Se você quer usar o seu kernel com um GRUB onde seja possível carregá-lo, você pode fazer o download do mesmo aqui[7] (você terá que compilá-lo e linka-lo ao que você precisa).

Conclusão

Pronto, um kernel básico. Você poderá agora querer escrever uma função k_printf, pois a que usamos em nosso exemplo é bem simples e não controla coisa como %s, %d, %c, etc. Com isto, agora é possível a você escrever uma função melhor.

Referências

[1] http://www.osdever.net/downloadsOLD/tuts/basic_kernel.zip
[2] http://nasm.sourceforge.net/
[3] http://www.delorie.com/djgpp/
[4] http://gcc.gnu.org/
[5] http://www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html
[6] http://www.execpc.com/~geezer/johnfine/index.htm
[7] http://www.osdever.net/downloadsOLD/tuts/basic_kernel_grub.zip