Bootlib: uma linguagem para definição de setores de inicialização

Roland Teodorowitsch
roland@ulbra.tche.br

ULBRA - Universidade Luterana do Brasil
Centro de Ciências Naturais e Exatas - Departamento de Informática
Av. Miguel Tostes, 101 - Bairro São Luís - CEP 92420-280 - Canoas-RS - BRASIL

Abstract

This paper presents the first version of a programming language, called Bootlib, that was specifically developed for the implementation of boot sectors. The main purpose of that language is to supply a friendly way to create a program that can take over the boot process in computers compatible with the IBM/PC standard. In its first version, the Bootlib language provides a set of commands that allow definition of data structures, video control, keyboard reading and disk sectors reading. Along this paper the concepts related to the boot process are presented. The definition of the version 0.1 of Bootlib and examples that illustrate its use and reinforce its potential are also presented. At last, the conclusions and some extensions that will be implemented in the next version of Bootlib are presented.
Key-words: Operating Systems, Initialization, Boot

1 Introdução

Os sistemas operacionais modernos têm quase todo o seu código implementado em linguagem de alto nível, o que facilita tanto o entendimento do código quanto a sua manutenção. O setor de inicialização (*boot sector*) é, no entanto, um dos poucos componentes dos sistemas operacionais que é invariavelmente implementado em linguagem de baixo nível (linguagem de montagem - *assembly*) [3].

O setor de inicialização corresponde ao primeiro setor de um disco ou de uma partição, que será carregado para a memória e executado, quando a máquina for ligada. Sua principal função é transferir a imagem do sistema operacional do disco para a memória e iniciar a sua execução. No entanto, para garantir que este setor de inicialização seja executado corretamente em todas as máquinas de uma determinada arquitetura, alguns quesitos devem ser atendidos: ele deve ter um tamanho bem definido e deve ser carregado sempre na mesma posição de memória. Para as máquinas padrão IBM/PC, o setor de inicialização deve ter apenas 512 bytes, sendo carregado sempre a partir do endereço de memória 0000:7C00 (ou 07C0:0000) [1,7,10,11].

Um dos principais fatores que contribuem para que o código do setor de inicialização seja codificado em linguagem de montagem é o fato de que ele está obrigatoriamente limitado a 512 bytes. O que significa que o código deve ser extremamente eficiente em termos de espaço ocupado. Também é importante observar que não poderão ser utilizadas quaisquer chamadas de sistema do sistema operacional, uma vez que o mesmo ainda não foi carregado. A única alternativa disponível é utilizar as rotinas disponíveis no BIOS (*Basic Input Output System*).

Caso a inicialização seja feita a partir de um disco rígido, ela será executada em duas etapas. Isto se deve ao fato de que os discos rígidos podem conter mais de uma partição, cada uma acomodando um sistema operacional diferente. Na primeira etapa, o primeiro setor do disco rígido é carregado e executado da mesma forma que em um disco flexível. No entanto, além do código de inicialização, este setor contém uma tabela, chamada de tabela de partições, com quatro entradas, que contém informações

sobre as quatro possíveis partições primárias. O programa contido neste setor de inicialização procura a partição marcada como ativa e carrega o primeiro setor desta partição (segunda etapa). Este setor será responsável pelo carregamento do sistema operacional armazenado nesta partição. O primeiro setor de um disco rígido, que contém a tabela de partições, recebe o nome de Registro Mestre de Inicialização (MBR - *Master Boot Record*).

Tanto para discos rígidos quanto para discos flexíveis, após o carregamento do setor de inicialização, o BIOS executa um desvio intersegmento para o endereço 0000:7C00. Em geral, os sistemas operacionais colocam um setor de inicialização nos discos flexíveis ou rígidos independentemente se eles são inicializáveis ou não. Isto evita que o usuário tente executar uma partida usando um disco sem sistema.

O setor de inicialização é responsável diretamente pelo carregamento de um determinado sistema operacional e deve buscar no disco a respectiva imagem deste sistema operacional e executá-la. O setor de inicialização do MS-DOS, por exemplo, foi projetado para verificar se o disco contém sistema verificando a presença de dois arquivos no diretório raiz: IO.SYS e MSDOS.SYS (ou IBMBIO.SYS e IBMDOS.SYS em determinadas versões de MS-DOS e PC-DOS) [2,9,10]. Em um disco flexível, se o setor de inicialização não encontrá-los, uma mensagem equivalente a "disco sem sistema" é exibida e o setor de inicialização espera pela substituição do disco para uma nova tentativa. Em um disco rígido, geralmente, será apresentada uma mensagem e o sistema ficará travado ou uma nova inicialização será tentada.

O objetivo deste artigo é apresentar uma linguagem de alto nível especialmente desenvolvida para a implementação de setores de inicialização. Esta linguagem se chama Bootlib. A filosofia do ambiente de programação desta linguagem é gerar, a partir de um programa fonte, um programa em linguagem de montagem correspondendo ao setor de inicialização. O programa em linguagem de montagem, que deve ser processado por um montador, contém basicamente um conjunto de rotinas que compõem uma biblioteca (daí a origem do nome Bootlib) e chamadas a estas rotinas.

O desenvolvimento desta linguagem e de seu compilador justificam-se pela praticidade e facilidade na especificação do primeiro código do sistema operacional a ser executado: o setor de inicialização. Constitui, portanto, uma ferramenta valiosa tanto no desenvolvimento de um novo sistema operacional, quanto no aperfeiçoamento de um sistema operacional já existente. O primeiro item poderia, sem dúvida, ser bastante útil em disciplinas envolvidas com o projeto de sistemas operacionais.

O fato do programador poder visualizar o código gerado em linguagem de montagem constitui também um fator que aumenta o valor didático do trabalho desenvolvido, uma vez que se pode acompanhar em detalhes todos os passos necessários à inicialização.

O desenvolvimento da linguagem Bootlib prevê diversas etapas, sendo que em cada etapa as funcionalidades da linguagem serão ampliadas. Nesta primeira etapa, versão 0.1, a definição da linguagem prevê uma série de funções de manipulação do vídeo, de leitura do disco, de leitura do teclado, de definição de estruturas de dados e de uso geral. Para uma próxima etapa, está prevista a introdução de variáveis, expressões e estruturas de controle, o que permitirá a especificação de expressões lógicas e aritméticas, testes e laços.

Visando ainda aumentar o uso potencial da linguagem e dos setores de inicialização gerados a partir dela, permite-se a criação de mais de um estágio de inicialização. Como o setor de inicialização primário é limitado a 512 bytes, não seria possível executar uma quantidade grande de código a partir dele. A solução é fazer com que o setor de inicialização primário carregue um segundo estágio de inicialização. Sendo que este segundo estágio, com mais de 512 bytes, pode executar tarefas mais

complexas, como, por exemplo, o gerenciamento da inicialização através de um menu para seleção do sistema operacional que será carregado.

A seção 2 apresenta a definição da linguagem Bootlib e a seção 3 apresenta exemplos de programas. Em seguida, a seção 4 descreve sucintamente o desenvolvimento do compilador para Bootlib. Por fim, são apresentadas as conclusões sobre o trabalho desenvolvido.

2 Definição da Linguagem

A linguagem Bootlib, possui, até certo ponto, algumas semelhanças com programas em C, tais como: uso dos mesmos caracteres para delimitar blocos, a mesma sintaxe para chamada de funções, os mesmos caracteres para definição de cadeias de caracteres e de caracteres constantes, e o mesmo tipo de comentários. A definição completa da versão 0.1 da Bootlib está descrita no anexo A.

A estrutura dos programas desenvolvidos em Bootlib é basicamente a seguinte:

```
boot ( <parâmetros> )
{
      <comandos>
}
```

Os comandos e funções a serem executados são definidos em um bloco de uma primitiva `boot()`. Esta primitiva também permite especificar alguns parâmetros que orientam a geração de código.

2.1 Parâmetros da Primitiva boot()

Os parâmetros espeficados através da primitiva `boot()` são os seguintes:

- `segment`: define o segmento de memória onde o programa será colocado. O valor do registrador DS (segmento de dados) será automaticamente inicializado com este valor, portanto, é importante definir corretamente este valor para que ele realmente coincida com a posição de memória onde o programa executável será carregado. Caso não seja especificado, assume-se o valor 0x07C0;
- `size`: define o tamanho máximo do programa executável. Deve-se escolher um valor compatível com o tamanho do código gerado. Ou seja, quanto maior o programa, mais memória ele ocupa e, portanto, tanto maior deverá ser o valor especificado através deste parâmetro. Caso o código ultrapasse o tamanho especificado, será gerada uma mensagem de erro durante o processo de montagem. Não há valor padrão;
- `signature`: é utilizado para definição dos dois últimos bytes do programa de inicialização, que são utilizados pelo BIOS e por outros utilitários para validação do setor de inicialização. Caso seja especificado algum valor, os dois últimos bytes do setor de inicialização receberão este valor. Se nenhum valor for especificado, os dois últimos bytes do setor de inicialização receberão, respectivamente, os valores 0x55 e 0xAA.

No exemplo a seguir, será gerado um programa de inicialização com tamanho igual a 512 bytes, que deverá ser executado no endereço de memória 07C0:0000, com a assinatura padrão.

```
boot ( size=512, signature )  {  ...  }
```

Para criar um programa com tamanho igual a 5120 bytes, que será executado no endereço 07E0:0000, sem assinatura, basta especificar:

```
boot ( segment=0x07E0, size=5120 )  {  ...  }
```

2.2 Funções da Linguagem

As funções aceitas pela versão 0.1 da Bootlib podem ser classificados em cinco grupos: definição de estruturas de dados, manipulação de vídeo, leitura do teclado, acesso a disco e funções diversas.

2.2.1 Definição de Estruturas de Dados

Para ser reconhecido pelo sistema operacional MS-DOS, e por todos aqueles sistemas operacionais que reconhecem sistemas de arquivos baseados em FAT (*File Allocation Table*) - 12 ou 16 bits, o setor de inicialização de discos flexíveis ou partições formatadas para MS-DOS, devem iniciar por uma estrutura de 62 bytes onde são definidos os seguintes parâmetros [7,10]: salto para o início do código (3 bytes), nome do sistema que formatou o disco (8 bytes), número de bytes por setor (2 bytes), número de setores por agrupamento (1 byte), número de setores reservados (2 bytes), número de FATs (1 byte), número de entradas no diretório raiz (2 bytes), número de setores para discos pequenos (2 bytes), descritor de mídia (1 byte), número de setores da FAT (2 bytes), número de setores por trilha (2 bytes), número de cabeças (2 bytes), número de setores escondidos (4 bytes), número extendido de setores (4 bytes), número da unidade (1 byte), byte reservado (1 byte), assinatura de inicialização (1 byte), identificador de volume (4 bytes), nome do volume (11 bytes) e nome do sistema de arquivos (8 bytes).

Esta estrutura pode ser inserida em um setor de inicialização através do comando `fat()`. Ao encontrar este comando em um programa, o compilador inserirá automaticamente a estrutura de dados esperada pelo MS-DOS. A princípio, este comando só tem sentido se utilizado no início do setor de inicialização, ou seja, se o comando `fat()` for o primeiro comando do programa.

O comando `fat()` aceita as seguintes opções:

- `disk`: determina o tipo de disco para o qual os dados serão gerados. Atualmente apenas discos de 3 1/2 polegadas de 1.44Mb (`FD1440`) são suportados;
- `oem`: permite especificar o nome do fabricante ou do sistema operacional que gerou o disco (deve ter até oito caracteres). O valor padrão é "BOOTLIB";
- `volume`: permite especificar o nome do volume. O valor padrão é "BOOTLIB_VOL".

No exemplo a seguir, é gerada uma estrutura de dados para um disco de 3 1/2 polegadas de 1.44Mb, com nome de fabricante igual a "TESTE" e nome de volume igual a "TESTE_VOL".

```
fat ( disk=FD1440, oem="TESTE", volume="TESTE_VOL" );
```

2.2.2 Manipulação do Vídeo

A linguagem Bootlib disponibiliza um conjunto razoável de funções que permitem criar aberturas coloridas em modo texto. Estas funções são:

- `bar`: recebe as coordenadas do canto superior esquerdo (coluna e linha), as coordenadas do canto inferior direito (coluna e linha) e um caracter, preenchendo um quadrado na tela com este caracter. A cor utilizada no preenchimento pode ser definida com as funções `setcolor()`, `setbgcolor()` ou `setfgcolor()`. Exemplo:
  ```
  bar ( 0, 0, 79, 24, 'X' );
  ```
- `clrscr`: limpa a tela. Não recebe nenhum parâmetro. A cor utilizada pode ser definida com as funções `setcolor()`, `setbgcolor()` ou `setfgcolor()`. Exemplo:
  ```
  clrscr ( );
  ```

- `cursor`: recebe um valor inteiro que define o formato do cursor. O valor 0 torna o cursor invisível, o valor 2 deixa o cursor do tamanho de um caracter, qualquer outro valor dexia o cursor no seu estado normal. Exemplo:
```
cursor ( 0 );
```
- `gotoxy`: desloca o cursor até as coordenadas (coluna e linha) especificadas. Exemplo:
```
gotoxy ( 10, 10 );
```
- `mode`: recebe um valor inteiro que define o modo de vídeo. São aceitos dois modos: 40 colunas ou 80 colunas. O valor 40 é utilizado para inicializar o vídeo em 40 colunas. Qualquer outro valor inicializa o vídeo em 80 colunas. Exemplo:
```
mode ( 40 );
```
- `newline`: desloca o cursor para o início da próxima linha, executando uma rolagem de tela se o cursor estiver na última linha da tela. Não recebe nenhum parâmetro. Exemplo:
```
newline ( );
```
- `putchar`: recebe e imprime um caracter no vídeo, atualizando a posição corrente do cursor. A cor utilizada na impressão pode ser definida com as funções `setcolor()`, `setbgcolor()` ou `setfgcolor()`. Exemplo:
```
putchar ( 'A' );
```
- `putnchars`: imprime um caracter no vídeo várias vezes, atualizando a posição do cursor. Esta função recebe o caracter e o número de vezes que o mesmo deverá ser impresso. A cor utilizada na impressão pode ser definida com as funções `setcolor()`, `setbgcolor()` ou `setfgcolor()`. Exemplo:
```
putnchars ( 'B', 20 );
```
- `putstr`: imprime uma cadeia de caracteres no vídeo, atualizando a posição do cursor. Esta função recebe uma cadeia de caracteres que pode conter alguns caracteres de escape: \n (nova linha), \\ (contra-barra), \" (aspas) e \cff (altera o atributo de cor de impressão para 0xff). A cor inicial utilizada na impressão pode ser definida com as funções `setcolor()`, `setbgcolor()` ou `setfgcolor()`. Exemplo:
```
putstr ( "cor normal\c70branco em fundo preto\n" );
```
- `rectangle`: recebe as coordenadas do canto superior esquerdo (coluna e linha) e as coordenadas do canto inferior direito (coluna e linha), desenhando uma moldura de linha simples. A cor utilizada no desenho pode ser definida com as funções `setcolor()`, `setbgcolor()` ou `setfgcolor()`. Exemplo:
```
rectangle ( 0, 0, 79, 24 );
```
- `rectangle2`: recebe as coordenadas do canto superior esquerdo (coluna e linha) e as coordenadas do canto inferior direito (coluna e linha), desenhando uma moldura de linha dupla. A cor utilizada no desenho pode ser definida com as funções `setcolor()`, `setbgcolor()` ou `setfgcolor()`. Exemplo:
```
rectangle2 ( 1, 1, 78, 23 );
```
- `setcolor`: define a cor que será utilizada pelas outras funções de manipulação de vídeo. Esta função recebe um único valor que corresponde à composição dos valores das cores de frente e fundo, conforme o formato utilizado pelas controladoras gráficas. O valor recebido deverá ser um valor inteiro de 0 a 255. Exemplo:
```
setcolor ( 0x70 );
```
- `setbgcolor`: define a cor de fundo que será utilizada pelas outras funções de manipulação de vídeo. O valor recebido deverá ser um valor inteiro de 0 a 7. Exemplo:
```
setbgcolor ( 0x7 );
```

- `setfgcolor`: define a cor de frente que será utilizada pelas outras funções de manipulação de vídeo. O valor recebido deverá ser um valor inteiro de 0 a 15. Exemplo:
```
setfgcolor ( 0xF );
```

2.2.3 Leitura do Teclado

Bootlib fornece uma única função para leitura do teclado. Esta função é `getch()`. Apesar de a sua implementação prever o retorno do código do caracter que foi pressionado, na versão 0.1, ainda não existem comandos e estruturas de controle para testar este valor. Basicamente ele serve para que o usuário espere pelo pressionamento de uma tecla. Exemplo:

```
getch ( );
```

2.2.4 Acesso a Disco

Para implementação do acesso a disco, Bootlib disponibiliza quatro funções básicas:

- `bootexec`: executa a carga e execução de um setor inicialização. Esta função recebe como parâmetro um valor inteiro que indica qual dos setores de inicialização deve ser carregado e executado. O valor zero corresponde ao Registro Mestre de Inicialização (MBR - *Master Boot Record*); 1, 2, 3 e 4, designam as respectivas partições. Esta função não retorna de sua execução, a não ser que ocorra algum erro. O disco a ser utilizado será aquele definido através da função `setdisk()`. Exemplo:
```
bootexec ( 0 );
```
- `readdisk`: executa a leitura de setores do disco. Esta função recebe como parâmetros: o número absoluto do setor a ser lido, o número de setores a ser lido, bem como o segmento e o deslocamento de uma área de memória onde os setores serão colocados. O disco a ser utilizado deverá ser definido através da função `setdisk()`. Devido à sua complexidade, esta função apresenta um código relativamente grande para ser colocado em um setor de 512 bytes. Para este caso recomenda-se o uso da função `readdisk_chs()`. Exemplo:
```
readdisk ( 0x21, 10, 0x1000, 0x0000 );
```
- `readdisk_chs`: executa a leitura de setores do disco. Esta função recebe como parâmetros: o número do disco, o número do cilindro, o número da cabeça, o número do setor, o número de setores a ser lido, bem como o segmento e o deslocamento de uma área de memória onde os setores serão colocados. Observe que esta função não necessita da função `setdisk()`. Exemplo:
```
readdisk_chs ( 0x00, 0, 1, 0x10, 10, 0x1000, 0x0000 );
```
- `setdisk`: seleciona o disco sobre o qual as funções `readdisk()` ou `bootexec()` serão executadas. O valor que esta função recebe como parâmetro tem o seguinte significado: 0x00, primeiro disco flexível; 0x01, segundo disco flexível; etc.; 0x80, primeiro disco rígido; 0x81, segundo disco rígido; etc. Exemplo:
```
setdisk ( 0x80 );
```

2.2.5 Funções Diversas

Entre as funções diversas estão:

- `bootstrap`: esta função ativa a rotina de partida do BIOS. Exemplo:
```
bootstrap ( );
```
- `jump`: executa um salto intersegmento. Para tanto, recebe dois parâmetros, respectivamente: o valor do segmento e o valor do deslocamento do código que deve ser executado. Esta função

é utilzada para executar o código de um setor que tenha sido carregado pelas rotinas de acesso a disco. Exemplo:

```
jump ( 0x0000, 0x7C00 );
```

- `setstack`: define a pilha do sistema. Recebe dois parâmetros, respectivamente: o valor do segmento e o valor do deslocamento do ponteiro de pilha. Esta deve ser uma das primeiras funções que o programa de inicialização deverá chamar. Exemplo:

```
setstack ( 0x07C0, 0x0000 );
```

Na próxima seção será apresentado um exemplo de programa usando a linguagem Bootlib.

3 Exemplo e Uso

Esta seção apresenta um exemplo de uso da linguagem e do compilador Bootlib. O compilador para a linguagem Bootlib está disponível para tanto para MS-DOS (BOOTLIB.EXE) quanto para Unix ou Linux [5] (bootlib). Basicamente ele recebe como parâmetro o nome do programa codificado em Bootlib e gera um programa em linguagem de montagem. Recomenda-se a utilização da extensão ".bl" (ou ".BL") para os programas codificados em Bootlib.

O exemplo que será apresentado, corresponde a um programa de inicialização de dois estágios. O primeiro estágio, ao ser carregado e executado, simplesmente imprime uma mensagem e carrega o segundo estágio. O segundo estágio, por sua vez, imprime uma tela formatada, espera pelo pressionamento de uma tecla e carrega e executa o setor de inicialização do primeiro disco rígido.

A figura 1 apresenta um o programa correspondente ao primeiro estágio, enquanto a figura 2 descreve o programa correspondente ao segundo estágio.

```
boot(size=512,signature,segment=0x07C0)
{
  fat(oem="SOS",volume="VOLUME",disk=FD1440);
  setstack(0x0000,0x7C00);
  setcolor(0x07);
  putstr("Carregando segundo estagio de inicializacao...");
  readdisk_chs(0x00,0x00,0x00,0x02,10,0x07E0,0x0000);
  jump(0x07E0,0x0000);
}
```

Figura 1 - Código do Primeiro Estágio - boot1.bl

Para compilar estes programas e gerar os arquivos binários correspondentes, ou seja, gerar os arquivos com a imagem dos programas de inicialização, deve-se inicialmente usar o compilador Bootlib:

```
bootlib -o boot1.asm boot1.bl
bootlib -o boot2.asm boot2.bl
```

A seguir, os arquivos `boot1.asm` e `boot2.asm`, gerados no passo anterior, devem ser montados e lincados, de forma que o resultado final sejam dois arquivos em formato binário puro, ou seja, sem o cabeçalho comum à arquivos executáveis.

Por fim, os arquivos em formato binário, `boot1.bin` e `boot2.bin`, devem ser copiados, respectivamente, para um disco flexível. O arquivo `boot1.bin` deve ser copiado para o primeiro setor do disco e o arquivo `boot2.bin` deve ter seus dez setores copiados a partir do segundo setor deste mesmo disco.

```
boot(size=5120,segment=0x07E0)
{
  setstack(0x0000,0x7C00);
  setcolor(0x04);
  clrscr();
  gotoxy(45,0);  putstr("ÚÄÄ¿ÚÄÄÂÄÄ¿  ÚÄÄÄÄÄ¿ÚÄÄÄÄÄ¿ ÚÄÄÄÄ¿");
  gotoxy(45,1);  putstr("³  ³³  ³  ³  ³  Ä  Á´  Ä  ÀÂÙ ÄÄ Á¿");
  gotoxy(45,2);  putstr("³  ÀÙ  ³  ÀÄÄ´  ÄÄ  ³  Ú¿  ³  Ú¿  ³");
  gotoxy(45,3);  putstr("ÀÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÙÀÄÄÄÁÄÄÙÀÄÄÄÙ");
  gotoxy(45,4);  putstr("- UNIVERSIDADE LUTERANA DO BRASIL -");
  setcolor(0x0C);
  gotoxy(0,0);
  putstr("         ÛÜÜ    ÜÜÜÛÛÛÜ     ÜÛÛÛÜÜÜ    ÜÜÛ\n");
  putstr("          ÛÛÛÛÛÛÛÛÛÛßßßÛÜ ÜÛßßßÛÛÛÛÛÛÛÛÛÛÛ\n");
  putstr("          ÛÛÛ ßßß       Û Û       ßßß ÛÛÛ\n");
  putstr("         ÜÛÛÛß          Û Û           ßÛÛÜ\n");
  putstr("       ÜÛÛÛÛß           Û Û            ßÛÛÛÜ\n");
  putstr("      ÜÛÛÛÛß            Û Û             ßÛÛÛÜ\n");
  putstr("      ßÛÛ          ÜÜÛÛÛÛÛ ÛÛÛÛÛÜÜ         ÛÛß\n");
  putstr("       ßÛÜÜ      ÜÛÛÛß   ßÜß   ßÛÛÜ      ÜÜÛß\n");
  putstr("       ÜÜ ßßÛÛÜÜÜÛÛ       Û       ÛÛÜÜÛÛßß ÜÜ\n");
  putstr("   ÜÛÛßßßßßßßÛÜÜÛÛÛÜÜÜÜÜÜÜÛÛÜÜÜÜÜÜÜÛÛÛÜÜÛßßßßßßÛÛÜ\n");
  putstr("  ÜÛÛ          ÛÛ         Û         ÛÛ          ÛÛÜ\n");
  putstr("  ÛÛÛ          ßÛÛ        Û        ÛÛß          ÛÛÛ\n");
  putstr("  ÛÛÛ           ÛÛÜ       Û       ÜÛÛ          ÛÛÛ\n");
  putstr("  ÛÛÛ            ÛÛ       Û       ÛÛ           ÛÛÛ\n");
  putstr("ÜÛÛÛÛÜÜ           ÛÛÜÜÜ  Û  ÜÜÜÛÛ          ÜÜÛÛÛÛÜ\n");
  putstr("  ßßÛÛÛÛÜ        ÜÛßßÛßßÛÛÛÛÛÛÛÛßßÛßßÛÜ       ÜÛÛÛÛßß\n");
  putstr("      ßÛÛÛÛÜ    Üß ßÛ            ßÛ ßÜÜ   ÜÛÛÛÛß\n");
  putstr("       ßÛÛÛÛÜÜÜÛßÜÛ               ÛÜ ÛÜÜÜÛÛÛÛß\n");
  putstr("         ßßßß  ÜÜ                  ÛÜ  ßßßß\n");
  putstr("               ÛÛÜ                 ÜÛÛ\n");
  putstr("               ßÛÛÛÛÛÛÜÜÜ   ÜÜÜÜÛÛÛÛÛß\n");
  putstr("                 ßßßÛÛÛÛÛÛÛÛÛÛÛÛßßß\n");
  putstr("                      ßÛÛÛÛÛÛß\n");
  putstr("                         Û\n");
  setcolor(0x04);
  gotoxy(50,24);
  putstr("Pressione uma tecla...");
  getch();
  bootexec(0);
}
```

Figura 2 - Código do Segundo Estágio - boot2.bl

4 Desenvolvimento do Compilador

O compilador para a linguagem Bootlib foi desenvolvido utilizando a linguagem C [5] e as ferramentas FLEX [8] e BISON [3], respectivamente, um gerador de analisadores lexicos e um gerador de analisadores sintáticos. Estas ferramentas são atualizações dos já consagrados LEX e YACC, disponíveis para diversas plataformas.

Usando FLEX, foi desenvolvido um reconhecedor léxico capaz de identificar todos os componentes que aparecem em um programa Bootlib: nomes de funções e comandos, valores inteiros, caracteres constantes, cadeias de caracteres e caracteres de pontuação. Após a identificação destes componentes, eles são passados na forma de sinalizadores (*tokens*) para o analisador sintático gerado

por BISON. A medida que o analisador sintático reconhece as construções da linguagem no programa, ele cria uma lista de comandos e funções que ao final da análise é usada para gerar um programa em linguagem de montagem.

5 Conclusão

Este artigo apresentou o desenvolvimento de uma linguagem especificamente projetada para a construção de setores de inicialização. A primeira especificação da linguagem contém uma série de funções que executam ações comuns em setores de inicialização. Com esta linguagem e seu compilador é possível criar programas capazes de assumir o processo de inicialização de um computador compatível com a arquitetura IBM/PC, exibindo mensagens, esperando pelo pressionamento de teclas, carregando e executando setores de um disco, flexível ou rígido.

A segunda versão para a linguagem Bootlib deverá prever não só um número maior de funções e comandos, mas também a inclusão de variáveis, expressões aritméticas e lógicas, bem como estruturas de controle como laços e instruções do tipo se-então-senão.

O valor didático da linguagem projetada e do compilador implementado é grande na medida em que apresenta uma forma simplificada para criar setores de inicialização - provavelmente um dos primeiros problemas com os quais um projetista de sistemas operacionais terá que se preocupar. A possibilidade de criar um setor de inicialização e carregar um segundo estágio de inicialização caracteriza bem o início da execução de um sistema operacional. A disponibilidade do código gerado em linguagem de montagem também contribui para aumentar o seu valor didático.

O código fonte do compilador para Bootlib está disponível gratuitamente a partir do endereço `http://www.ulbra.tche.br/~roland/bootlib`.

Bibliografia

[1] BURGESS, Richard A. **Developing Your Own 32-Bit Operating System**. SAMS Publishing, Indianapolis, 1995.

[2] CHAPPELL, Geoff. **DOS Internals**. Addison-Wesley, Readings, 1994.

[3] DONNELLY, Charles, STALLMAN, Richard. **Bison: The YACC-compatible Parser Generator**. Edição 1.25. Free Software Foundation: Novembro de 1995.

[4] INTEL Corporation. **Pentium Pro Family Developer's Manual - Volume 2: Programmer's Reference Manual**. Intel, Mount Prospect, 1996.

[5] KERNIGHAN, Brian W., RITCHIE, Dennis M. **A Linguagem de Programação C**. 4ª. ed. Rio de Janeiro: Campus, 1988.

[6] LINUX Systems Labs. **LINUX: The Complete Reference**. 4th edition. Walnut Creek CDROM, 1996.

[7] NORTON, Peter, AITKEN, Peter, WILTON, Richard. **Peter Norton, A Bíblia do programador**. Rio de Janeiro: Campus, 1994.

[8] PAXSON, Vern. **Flex: A fast scanner generator**. Edição 2.5. University of California: Março de 1995.

[9] PODANOFFSKY, Michael. **Dissecting DOS**. Reading: Addison-Wesley, 1995.

[10] VILLANI, Pat. **FreeDOS Kernel**. Lawrence: R&D Books, 1996.

[11] TANENBAUM, Andrew S. **Operating Systems: Design and Implementation**. Prentice-Hall, Englewood Cliffs, 1987.

Anexo A: Definição Sintática da Linguagem Bootlib

A sintaxe para a linguagem Bootlib é descrita a seguir usando a representação de Backus-Naur (BNF - *Backus Naur Form*). Esta representação utiliza os seguintes metassímbolos:

::= Significa "é definido como"
| Significa "ou"
{ } Significa que o conteúdo pode ser repetido nenhuma ou várias vezes
< > Significa que o conteúdo é uma construção sintática BNF

Todos os outros símbolos, que fazem parte da linguagem, aparecem em negrito.

```
<programa>          ::=  boot ( <parâmetros> ) <bloco-comandos>
<parâmetros>        ::=  <vazio> | <opção> { , <opção> }
<opção>             ::=  <opção-segmento> | <opção-tamanho> | <opção-assinatura>
<opção-segmento>    ::=  segment = <constante-inteira>
<opção-tamanho>     ::=  size = <constante-inteira>
<opção-assinatura>  ::=  signature = <const-i> | signature
<bloco-comandos>    ::=  { <comando> }
<comando>           ::=  ;
                     |   bar ( <const-i> , <const-i> , <const-i> , <const-i> , <const-i> ) ;
                     |   bootexec ( <const-i> ) ;
                     |   bootstrap ( ) ;
                     |   clrscr ( ) ;
                     |   cursor ( <const-i> ) ;
                     |   fat ( <parâmetros-fat> ) ;
                     |   getch ( ) ;
                     |   gotoxy ( <const-i> , <const-i> ) ;
                     |   jump ( <const-i> , <const-i> ) ;
                     |   mode ( <const-i> ) ;
                     |   newline ( ) ;
                     |   putchar ( <const-i> ) ;
                     |   putnchars ( <const-i> , <const-i> ) ;
                     |   putstr ( <cadeia-de-caract> ) ;
                     |   readdisk ( <const-i> , <const-i> , <const-i> , <const-i> ) ;
                     |   readdisk_chs ( <const-i> , <const-i> , <const-i> , <const-i> ,
                                        <const-i> , <const-i> , <const-i> ) ;
                     |   rectangle ( <const-i> , <const-i> , <const-i> , <const-i> ) ;
                     |   rectangle2 ( <const-i> , <const-i> , <const-i> , <const-i> ) ;
                     |   setbgcolor ( <const-i> ) ;
                     |   setcolor ( <const-i> ) ;
                     |   setdisk ( <const-i> ) ;
                     |   setfgcolor ( <const-i> ) ;
                     |   setstack ( <const-i> , <const-i> ) ;
```

<parâmetros-fat>	::=	<vazio> | <opção-fat> **{ ,** <opção-fat> **}**
<opção-fat>	::=	<opção-disk> | <opção-oem> | <opção-volume>
<opção-disk>	::=	**disk =** <tipo-de-disco>
<opção-oem>	::=	**oem =** <cadeia-de-caract>
<opção-volume>	::=	**volume =** <cadeia-de-caract>
<tipo-de-disco>	::=	**FD1440**
<cadeia-de-caract>	::=	<aspas> { <caracter-cadeia> } <aspas>
<caracter-cadeia>	::=	<letra-mai> | <letra-min> | <digito> | <caract-esp> | <apóstrofo>
	|	<outros-caract> | <contra-barra> <aspas> | <contra-barra> **n**
	|	<contra-barra> <contra-barra>
	|	<contra-barra> **c** <digito-hexa> <digito-hexa>
<const-i>	::=	<inteiro> | <hexadecimal> | <caracter>
<inteiro>	::=	<digito> { <digito> }
<hexadecimal>	::=	**0x** <digito-hexa> { <digito-hexa> }
<caracter>	::=	<apóstrofo> <caract> <apóstrofo>
<caract>	::=	<letra-mai> | <letra-min> | <digito> | <caract-esp>
	|	<outros-caract> | <contra-barra> <apóstrofo>
<digito-hexa>	::=	<digito> | **A** | **B** | **C** | **D** | **E** | **F** | **a** | **b** | **c** | **d** | **e** | **f**
<digito>	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
<letra-mai>	::=	**A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P**
	|	**Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
<letra-min>	::=	**a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s**
	|	**t** | **u** | **v** | **w** | **x** | **y** | **z**
<caract-esp>	::=	<espaço> | **!** | **#** | **$** | **%** | **&** | **(** | **)** | ***** | **+** | **,** | **-** | **.** | **/** | **:**
	|	**;** | **<** | **=** | **>** | **?** | **@** | **[** | **]** | **^** | **_** | **`** | **{** | **|** | **}** | **~**
<espaço>	::=	
<apóstrofo>	::=	**'**
<contra-barra>	::=	\
<aspas>	::=	**"**
<vazio>	::=	

A construção sintática <outros-caract>, não definida nas regras sintáticas da Bootlib, corresponde a qualquer caracter com código ASCII variando de 127 até 255.

A linguagem Bootlib aceita também comentários, que podem aparecer em qualquer parte do programa e que são iniciados por **/*** e terminados por ***/**.