Introdução à programação em Assembly x86 no Linux

Blabos de Blebe

<blabos@blabos.org>

Centro Universitário da FEI

São Bernardo do Campo

9 de maio de 2010

Resumo

Este pequeno texto é baseado na **minha própria interpretação** do excelente livro *Programming from the Ground Up* de Jonathan Bartlett e tenta apresentar brevemente uma introdução aos principais conceitos relacionados à programação Assembly em ambiente Linux, abrangendo um pouco da teoria e das ferramentas necessárias.

Palavras-chave: Programação, Assembly, Linux

Sumário

1 Introdução

Este texto é baseado principalmente no livro Programming from the Ground Up de Jonathan Bartlett [Bar04]. A leitura do livro é bastante recomendada. Além uma linguagem simples e fácil de ler os assuntos são abordados mais profundamente do que serão aqui. Ele é distribuído sob a licença Gnu FDL [FSF08] e pode ser baixado livremente através deste link.

O livro disponibiliza exercícios, alguns dos quais serão reproduzidos aqui. Há também uma lista de emails disponibilizada pelo autor do livro que pode ser acessada clicando aqui.

Os créditos pelo conteúdo são inteiramente do autor do livro original, enquanto qualquer erro certamente é minha culpa [1].

1.1 Ferramentas

Este texto trata de programação em Assembly para x86 em ambiente Linux e será feito com base em um Ubuntu 10.04. Na medida do possível tentarei apresentar um breve sumário dos comandos utilizados ou links para a documentação.

Os pacotes de desenvolvimento podem ser instalados no Ubuntu a partir do metapacote **build-essentials** que vai automaticamente instalar o gcc, bibliotecas, cabeçalhos, etc. Recomendo também a instalação dos pacotes **manpages-dev** e **manpages-posix-dev** que são pacotes de documentação muito úteis.

1.1.1 GCC

GCC significa Gnu Compiler Collection e é uma coleção de compiladores bibliotecas e ferramentas para várias linguagens [GCC10].

[1] Eu ando torcando letras frequentemnete :)

1.1.2 Linux

Linux é o nome do *Kernel* de uma distribuição GNU/Linux. Ele é o núcleo do sistema operacional, responsável por disponibilizar uma interface homem-máquina, gerenciar os recursos do hardware e prover uma API unificada de programação [2], facilitando o acesso pelo usuário e protegendo-o dele mesmo. Entretanto o kernel por si próprio não faz muita coisa, ele precisa de um conjunto de ferramentas para ter uso prático.

Combinando o kernel (Linux) com as ferramentas de usuário do Projeto GNU, temos um sistema operacional completo [3].

1.1.3 Linguagens de Programação

Na maior parte do texto usaremos linguagem de baixo nível, Assembly, existindo basicamente três tipos de linguagem:

Linguagem de Máquina É a linguagem que o computador entende e manipula, tratando-se de uma sequência de comandos binários essencialmente ilegíveis para humanos normais.

Linguagem Assembly [4] É essencialmente o mesmo que a linguagem de máquina, exceto que os comandos binários são substituídos por mnemônicos [5] mais fáceis de serem memorizados. Outras coisas são adicionadas para tornar a programação mais fácil [6].

Linguagem de Alto Nível Uma linguagem de alto nível existe para tornar a programação fácil. Quando você programa em Assembly, você precisa lidar com a máqina você mesmo, enquanto que ao utilizar uma linguagem de alto nível, você pode descrever o seu algoritmo através de expressões mais naturais. Em geral, uma simples instrução em linguagem de alto nível é equivalente a várias instruções em Assembly [7].

O texto abordará principalmente linguagem assembly e eventualmente alguns aspectos de linguagens de alto nível. Entendendo bem os conceitos básicos de programação de baixo nível, você estará apto a entender melhor a programação de alto nível.

[2] Eu não resisti ;)

[3] Diga-se de passagem, o GNU/Linux deve o que ele é hoje a um conjunto enorme de ferramentas criadas por várias pessoas e empresas ao redor do mundo, principalmente pelo projeto GNU, e alguns autores afirmam que chamar toda uma distribuição somente de Linux, seria não dar crédito ao projeto GNU.

[4] **Assembly** é o nome da linguagem, enquanto **assembler** é o nome do programa que realiza a montagem. Você programa em Assembly e monta com o assembler.

[5] Do Grego μνημονικός μνēμονικός ("da memória"), é uma técnica de memorização que utiliza siglas para lembrar de números por exemplo.

[6] Mais fácil do que escrever em binário, e só!

[7] Isso explica porque instruções em C ou Java podem ser interrompidas "no meio" de sua execução.

2 *Arquitetura Básica de Computadores*

Antes de aprender como programar, você precisa primeiro entender como um computador interpreta programas. Você não precisa ter um diploma em Engenharia Elétrica, mas você precisa enteder o básico.

Os computadores modernos são baseados na arquitetura chamada de Von Neuman, nome de seu criador. A arquitetura Von Neuman divide o computador em duas partes principais, a CPU (Unidade Central de Processamento) e a memória. Essa arquitetura é usada em todos os computadores modernos, inclusive em PCs, supercomputadores, mainframes e ainda em celulares [Bar04].

2.1 Estrutura da memória

Para facilitar o entendimento da estrutura da memória, imagine que você tem um armário com várias gavetas do mesmo tamanho e cada uma delas é numerada [1].

A memória do computador é organizada de forma análoga. A principal diferença é que nas gavetas você pode armazenar qualquer coisa, mas na memória somente um único número por vez. A razão disso é que é mais fácil e barato implementar assim.

A memória do computador é utilizada para os mais diversos fins. Tudo que é calculado é armazenado na memória, aliás, tudo que é "armazenado", é guardado na memória. Por exemplo:

- A posição do cursor na tela
- O tamanho de cada janela
- As imagens de cada ícone
- E por ai vai ...

[1] O autor utiliza o exemplo de caixas postais no correio, mas eu achei que o meu exemplo seria mais ilustrativo.

Adicionalmente, a arquitetura define que não somente dados são armazenados na memória, mas os programas que são executados pelo computador também. Em computação, não há diferença entre programas e dados, exceto como eles são usados pelo computador. Ambos são armazenados na memória e acessados do mesmo jeito.

2.2 A CPU

Só armazenar dados não é muito útil. Além disso é preciso acessá-los, manipulá-los, movê-los, etc. Esse é o trabalho da CPU.

A CPU lê as instruções da memória uma de cada vez e as executa. Isto é conhecido com ciclo *fetch-execute* (obter e executar). Para isso a CPU conta com os seguintes componentes:

Contador de Programa é um registrador que aponta para o endereço onde está a próxima instrução a ser executada.

Decodificador de Instrução é o responsável por identificar a instrução e os dados que ela utiliza.

Barramento de Dados é a ligação entre a memória e a CPU, utilizada para obter os dados necessários para executar a instrução.

Registradores de propósito geral são aŕeas de memória ultra-rápidas que existem dentro do processador. Na verdade o processador só pode executar instruções que estejam carregadas nos registradores.

Unidade Lógica e Aritmética é o componente que finaliza o processamento, executando as instruções.

Na realidade existem outros componentes trabalhando durante o processo de execução de uma instrução, mas eles são irrelevantes ao objetivo deste texto.

2.3 Um pouco de vocabulário

Para tornar as coisas mais claras possível, vamos relembrar e/ou explanar alguns termos de uso comum em computação:

Endereço um número que identifica e localiza cada unidade da memória.

Bit é a unidade mínima de dados, podendo assumir dois valores: 1 ou 0.

Byte é o nome da unidade mínima de alocação. Na arquitetura x86 possui 8 bits de tamanho podendo aramazenar os números de 0 a 255.

Byte, Word, Double Word e Quad Word Por razões históricas admite-se que essas palavras significam os tamanhos de dados em computadores, referindo-se 8, 16, 32 e 64 bits respectivamente. Note que o significado da palavra "word" neste caso é ligeiramente diferente do significado na expressão "word machine".

char, short, int, long, float e double são nomes de abstrações criadas por linguagens de alto nível para representar tipos de dados. Alguns desses tipos são definidos por normas, enquanto outros não, ficando a cargo da especificação da linguagem

ASCII é uma das tabelas utilizadas para converter números em caracteres para serem mostrados na tela [2]. Como a tabela ASCII é a mais famosa, em geral, quando utilizamos o temo "ASCII" estamos nos referindo ao número que representa um determinado caractere nesta tabela.

Tamaho da palavra da máquina é a quantidade de bits que uma máquina consegue processar por vez. Esse é também o tamanho dos registradores. Quando falamos "máquina de X bits" estamos querendo dizer "máquina cujo tamanho da palavra é X". Na arquitetura x86 isso define também a quantidade de memória que uma máquina pode endereçar [3] (32 bits = 2^{32} = 4294967296 endereços).

Note que o computador armazena endereços da mesma forma que um número ordinário. De fato, o computador não sabe a diferença entre um número, um endereço ou um código ASCII. Um número torna-se um caracter quando você o envia par a tela e torna-se um endereço quando você tenta acessar o byte que está naquela posição da memória.

Reflita um pouco sobre isso, pois este conceito é crucial para entender commo os programas de computador funcionam.

Ponteiro é um endereço armazenado na memória. Esse nome é dado porque ao invés de possuir um "dado", nesse caso a memória possui um endereço que "aponta" para o dado.

[2] A tabela ASCII traduz somente os principais caracteres existentes no inglês, onde cada caracter corresponde a um número e vice-versa. Com o advento do Unicode, essa correspondência nem sempre vale. Neste texto vamos ignorar o Unicode para simplificar.

[3] Em outras arquiteturas isso pode ser diferente.

2.4 Interpretando a memória

Computadores são exatos, portanto os programadores precisam ser igualmente exatos. Se você manda um número para a tela, ao invés de um código ASCII, o computador vai tentar interpretar esse número como ASCII e imprimeir na tela o que ele conseguir. Se você mandar o computador executar uma instrução numa área que só contém dados, ninguém sabe como ele vai interpretar isso, mas certamente ele vai tentar te obedecer.

O ponto é que o computador vai tentar fazer exatamente o que você mandou, não importa se isso faz sentido ou não. Lembre-se que computadores só sabem armazenar números, então letras, imagens, páginas web ou qualquer outra coisa são apenas grandes sequências de números, as quais alguns programas em particular conseguem interpretar como algo útil.

2.5 Métodos de acesso aos dados

Os processadores possuem várias formas de acessar os dados que são conhecidas como *modos de endereçamento*.

O modo mais simples é o *modo imediato*, onde o dado é passado na própria instrução. Por exemplo, para inicializar um registrador com 42, ao invés de dar ao processador o endereço onde procurar 42, simplemsmente especificamos o modo imediato e passamos 42.

No *modo de endereçamento de registrador*, a instrução contém o registrador onde está o dado, ao invés de um lugar na memória.

No *modo de endereçamento direto*, a instrução contém um endereço de memória onde está o dado. Podemos pedir por exemplo para o processador carregar o registrador X com o coteúdo da área de memória 0x2010. O computador vai então olhar o que tem dentro da memória 0x2010 e colocar o seu **conteúdo** no registrador X.

Já no *modo de endereçamento indexado*, a instrução contém um endereço de memória e um *registrador de índice* o qual armazena um offset ou deslocamento para o endereço dos dados. Na arquitetura x86 você pode ainda ter um *multiplicador* para o índice. Por exemplo, se quisermos o dado no quarto byte após o endereço 0x2010, podemos especificar o modo indexado, passar o endereço de base 0x2010, colocar 1 no multiplicador e 3 no índice (lembre-se que em computação contamos a partir do 0), que obteremos o dado no endereço 0x2013.

Temos ainda o *modo de endereçamento indireto* no qual a instrução contém um registrador no qual há um ponteiro (endereço) para o dado na memória.

Finalmente temos o *modo de endereçamento por ponteiro base*, semelhante ao modo indireto exceto a instrução inclui um número para o offseta ser adicionado ao ponteiro base.

2.6 Exercícios

1. Descreva o ciclo fetch-execute.
2. O que é um registrador? Qual o tamanho dos registradores nas máquians que usamos hoje?
3. Como um computador sabe como interpretar os dados na memória?
4. Qual endereço estamos tentando acessar nos casos abaixo?
   (a) Modo ponteiro base, passando um registrador contendo o endereço 3122 e offset 20.
   (b) Modo indexado, endereço base 6512, registrador de índice contendo 5 e multiplicador contendo 4.
   (c) Modo indexado, endereço base 123472, registrador de índice contendo 0 e multiplicador contendo 4.
   (d) Modo indexado, endereço base 9123478, registrador de índice contendo 20 e multiplicador contendo 1.
5. Como o pipelime pode interferir no ciclo fetch-execute?
6. Quais os prós e contras de instruções de tamanho fixo e instruções de tamanho variável?

3 *Primeiros programas*

Neste caítulo nós vamos aprender a escrever programas em Assembly utilizando o Linux, bem como a estrutura desses programas e algumas instruções em Assembly.

3.1 Escrevendo e executando um programa em Assembly

Para começar vamos considerar o código abaixo:

Listing 3.1: Primeiro programa em Assembly

```
.section .data
.section .text
.globl _start
_start:
movl $1, %eax    # Coloca o valor 1 no registrador eax
movl $0, %ebx    # Coloca o valor 2 no registrador ebx
int $0x80        # Chama a interrupcao 0x80
```

Com um editor de textos, salve esse código exatamente como está em um arquivo chamado exit.s (*.s, *.S ou *.asm são extensões comuns para códigos em Assembly). Não se preocupe com o código, nesta primeira etapa vamos falar como escrever e rodar o programa, depois explicaremos como ele funciona.

Esse arquivo que você acabou de criar, é o código fonte. ele precisa ser montado (assembled) e link-editado [1] (linked) para que ele possa ser transformado em um programa.

O primeiro passo é montar o código. Isso significa traduzir o que você escreveu para linguagem de máquina. Para fazer isso, abra um terminal, vá ao diretório onde slavou o arquivo exit.s e execute:

```
user@host:~ as exit.s -o exit.o
```

[1] É bastante comum em jargão técnico o aportugesamento o verbo inglês link para linkar, criando expressões como linkagem, linkar, linkado, etc. Vamos utilizar essas expressões a partir daqui.

O as é o assembler. Como resultado, deve ser gerado um arquivo chamado exit.o no diretório local. Esse arquivo é chamado de *código objeto*. Um arquivo de código objeto possui código em linguagem de máquina mas ainda não é executável. Em projetos grandes é comum existirem vários arquivos de código objeto. Para transformá-los em um programa o linker precisa juntá-los e adicionar informações necessárias para o kernel entender como executá-lo.

Para linkar o código objeto execute o seguinte comando:

```
user@host:~ ld exit.o -o exit
```

ld é o linker. Ele toma os arquivos de código objeto, realiza a linkagem e gera um programa executável.

Toda vez que o código fonte for alterado você precisa repetir a montagem e a linkagem.

Para executar o programa que acabamos de criar, execute o seguinte comando:

```
user@host:~ ./exit
```

O ./ diz ao linux para procurar o programa exit no diretório corrente (.). Todos os programas precisam do seu caminho completo para serem executados. A razão de não precisarmos fazer isso com os comandos as e ld é porque eles estão localizados em diretórios especiais, que são listados em variáveis de ambiente especiais que não abordaremos aqui. Note que se você tentar executar o programa exit sem o ./ no início, você vai na verdade executar o programa exit do sistema operacional, então preste atenção!

Você notará que nada acontece ao executar ./exit. Isso é porque o programa não faz nada mesmo, só retorna o controle para o sistema opracional. Entretanto execute o comando abaixo imediatamente após executar ./exit:

```
user@host:~ echo $?
```

Ele deverá responder 0. O que aconteceu foi que cada programa quando termina, retorna para o Linux um código de status. Em Unix, quando um programa executa corretamente ele retorna 0 ao sistema operacional, enquanto se houve algum erro ele retorna algo diferente de 0. Qual valor é retornado e qual o seu significado, fica a cargo do progrmador. O comando que você acabou de executar (`echo $?`) mostra o resultado devolvido pelo último programa executado.

3.2 Entendendo um programa em Assembly

Consideremos o programa que acabamos de criar. Todo o texto após os caracteres # são comentários. Isso significa que o assembler vai ignorá-lo. Vamos analisar os comandos linha a linha:

```
.section .data
```

Tudo que começa com um ponto não é traduzido automaticamente. Ao invés disso esse instrução é montada também. Elas são chamadas de *diretivas* ou *pseudo-operações*. O comando .section por exemplo, quebra o programa em seções.

.section .data inicia a seção de dados do programa. Esta seção é a área utilizada para armazenamento de dados dentro do executável. É aqui que as variáveis estáticas do C ficam. O nosso programa não utiliza esse seção mas ela está aí apenas para deixar o exemplo completo.

Depois temos:

```
.section .text
```

Esta linha inicia a seção de texto. É nessa seção que as instruções do program ficam.

Em seguida:

```
.globl _start
```

Este comando é utilizado para dizer ao assembler para criar um símbolo chamado _start. Símblos são nomes dados a trechos de código ou dados e são utilizados para referenciar esse código ou dado. Sem os símbolos você precisaria referenciar cada trecho de código ou dado pelo seu endereço numérico. Assim a cada vez que o program precisasse receber uma nova instrução, seria preciso alterar os endereços de todo o resto. Lembre-se que quando um programa é carregado na memória, cada instrução também tem um endereço.

O .glob diz ao assembler que _start é um símbolo global que será necessário para o linker. _start é um símblo especial que sempre precisará ser marcado com .glob, pois ele é o ponto de início do programa. Sem isso o computador vai carregar o programa mas não vai saber por onde começar a executá-lo.

A próxima linha

```
_start:
```

define o valor de uma label. Uma label é um símbolo seguido de dois pontos (:). Quando o assemlber monta o código fonte, ele asscioa cada dado ou instrução a um endereço. As labels dizem ao assembler que o valor do símbolo deve ser o endereço da primeira instrução ou dado logo depois dela. Assim, se alguma instrução ou dado mudar de lugar, você não precisa reescrever todo o código!

Agora vamos finalmente às instruções do programa:

```
movl $1, %eax
```

Quando o programa for executado, esta instrução coloca o valor 1 no registrador %eax

REFERÊNCIAS

[Bar04] Jonathan Bartlett. *Programming from the Ground Up*. Bartlett Publishing, Broken Arrow, OK, Julho 2004.

[FSF08] Gnu free documentation license. http://www.gnu.org/licenses/fdl.html, 2008.

[GCC10] Gcc, the gnu compiler collection. http://gcc.gnu.org, 2010.