CI064 - Software Básico

Bruno Müller Junior

21 de Setembro de 2009

Prefácio

Este texto corresponde à versão inicial de um material de apoio para a disciplina CI064-Software Básico, ministrada por mim no Curso de Bacharelado em Informática da UFPR, e certamente contém erros léxicos, sintáticos e semânticos. Sempre que o aluno que está matriculado na disciplina encontrar erros (ou não entender alguma coisa), deve me procurar para que eu possa melhorar o texto, tornando-o mais claro e preciso.

Conteúdo

Capítulo 1

Objetivos da Disciplina

- Fornecer conhecimentos básicos sobre sistemas computacionais e organização de computadores:
  - Descrever o relacionamento entre a máquina, sua linguagem de baixo nível e as linguagens de alto nível usadas para programá-la.
  - Diferenciar os programas de software básico: compilador, ligador, montador e carregador.
  - Diferenciar programas objeto e programas executáveis.
  - Diferenciar bibliotecas estáticas, compartilhadas e dinâmicas.
  - Programação em linguagem de baixo nível onde serão explicados conceitos sobre:
    - conversão de programas em linguagem de alto nível para linguagens de baixo nível.
    - diferenciar área de código (*text*), área de dados estáticos (*bss*), área de dados dinâmicos (*heap*) e área de pilha (*stack*).
    - explicar conceitos sobre a alocação de variáveis estáticas e dinâmicas.
    - detalhar o mecanismo de chamadas de procedimento.
    - técnicas de endereçamento.
    - explicar o uso de chamadas de sistema (system calls).

Capítulo 2

Organização da Disciplina

Para atingir os objetivos, a disciplina está organizada em duas partes:

Primeira Parte: Linguagem assembly da família x86. A linguagem assembly é utilizada para exemplificar alguns dos conceitos de linguagens de programação, de sistemas operacionais e de arquitetura de computadores como, por exemplo, uso de registradores, memória, interrupções, system calls e chamadas de procedimento. As ferramentas (software) que serão utilizados são:

- "as" (montador assembly do gnu)
- "ald" (assembly language debugger)

Ambos estão disponíveis nas máquinas do departamento de informática da UFPR, porém aqueles que preferem utilizar os seus computadores pessoais: o primeiro normalmente vem junto com qualquer distribuição linux, enquanto que o segundo deve ser baixado à parte. Além deste texto, é recomendável a leitura do livro [Bar04], no qual eu me baseei para vários exemplos. Atualmente (1^o Semestre de 2006), há uma versão gratuita disponível em "http://savannah.nongnu.org/projects/pgubook/".

Segunda Parte: software de base (compiladores, ligadores, montadores e carregadores), os formatos intermediários dos arquivos que eles geram (arquivos objeto, arquivos executáveis, arquivos fonte).

Parte I

Tradução e Execução de Programas

Cada sistema operacional apresenta um mecanismo próprio para a execução de programas. A idéia mais simples para colocar um programa em execução é copiar todo o arquivo a ser executado em alguma região da memória física e habilitar a sua execução. Este é o modelo adotado pelos primeiros sistemas operacionais MS-DOS (que foi baseado em outro ainda mais antigo que era utilizado em sistemas operacionais CP/M), mais especificamente para os arquivos executáveis que tinham a extensão ".com". Este modelo de execução ficou conhecido como modelo COM (que dizem que são as iniciais de "Copy On Memory").

Via de regra, os arquivos executáveis tem um cabeçalho, que informa ao sistema operacional alguns aspectos importantes do arquivo a ser executado, como por exemplo quanta memória física ele vai utilizar. No caso do COM, o cabeçalho do arquivo indica o endereço **físico** de memória em que o código e alguns dados devem ser copiados. O COM funciona bem em ambientes mono-processo (onde só um processo pode ser executado de cada vez), porém quando mais de um processo é permitido, há a possibilidade (nada remota) de que o segundo processo queira usar exatamente o mesmo espaço de memória física do primeiro. Neste caso, o Sistema Operacional deveria encontrar uma forma de permitir que o segundo processo fosse colocado em execução sem afetar o primeiro. Uma solução é copiar em disco a imagem em memória do primeiro processo e copiar o segundo processo em memória. Enquanto o segundo processo não finalizar a sua execução, o primeiro processo fica "suspenso". Para colocar o primeiro processo de volta para a execução, basta copiar a imagem armazenada em disco para a memória novamente. Esta solução funciona (e foi usada por muito tempo), mas se fosse encontrada uma forma de carregar os programas em endereços diferentes, vários poderiam ocupar a memória simultaneamente.

Para isto foi criado o modelo EXE, onde o sistema operacional pode inserir o novo processo em qualquer lugar vago da memória física (onde o procesos caiba, claro). Neste caso, algumas mudanças no arquivo são necessárias, como por exemplo informações para auxiliar no mapeamento do endereço físico de cada variável na memória.

Apesar de simples, estes modelos são frágeis em vários aspectos, principalmente no aspecto segurança. Todos os processos em execução recebem "super-poderes" para ler e escrever em qualquer lugar da memória, e em alguns casos ele pode até de se comunicar diretamente com os dispositivos do computador (vídeo, disco, etc). Desta forma, qualquer processo pode, por exemplo, escrever dados em uma região crítica do disco (tipicamente , na trilha zero) e com isso danificar irremediavelmente o equipamento (ou o acesso aos dados). Suponha que um programador malvado consegue incluir um trecho de código "maldoso" em um programa executável "bondoso". Se este novo programa for distribuído entre os amigos (ou inimigos), ele pode funcionar bem durante algum tempo, porém quando o trecho maldoso for acionado (e o usuário pode nem perceber que isso ocorre), ele pode fazer uma grande maldade de pouco em pouco. Se este trecho de código "maldoso" souber como se incluir em outros arquivos armazenados em disco, então o computador estará infectado com o trecho maldoso (que é também conhecido como vírus).

Este trecho maldoso é composto por um pequeno conjunto de instruções em assembly que não muda quando infecciona outros arquivos. Como ele não muda, basta procurar por estas seqüências de instruções em cada arquivo do sistema para verificar a presença ou ausência de vírus. Porém, como novos vírus aparecem regularmente,

é imprescindível que o programa anti-vírus seja atualizado com os novos "padrões" a procurar[1].

Em contrapartida, sistemas operacionais seguros dão uma área restrita para cada processo atuar (uma "caixa de areia" para brincar), e se o processo quiser qualquer coisa fora daquela área, tem de pedir ao SO. Isto inibe, mas não impede, a criação de processos que causem danos ao computador (como por exemplo vírus). A criação de vírus também é possível em sistemas operacionais seguros, porém como os programas não tem "super-poderes", os vírus normalmente não causam um dano tão grande. Para tal, falhas de implementação podem e são usadas para ataques. Exemplos destes ataques são relatados em [JDD+04]. Para aqueles que se interessam pelo tema, é interessante ler sobre *rootkits*, ferramentas desenvolvidas para auxiliar intrusos a subverter a segurança da máquina. Um dos livros mais importantes na área é[GJ02], que trata de *rootkits* para windows. Apesar de não estar atualizado, é um excelente ponto de partida.

Os modelos COM e EXE são simples, porém não permitem a inclusão de aspectos interessantes como a iniciação de objetos em linguagens orientadas a objetos, entre outros. Para isto, é necessária a criação de outros modelos de arquivo executáveis e de formatos de execução em memória. Por esta razão, um arquivo criado para um SO dificilmente será executado diretamente em outro (só através de interpretadores ou de conversores).

O modelo de execução mais utilizado em linux (e em todo o mundo Unix) é o ELF[2], onde um programa recebe uma grande porção de memória para executar. Esta porção de memória é dividida em pedaços, chamados seções, e cada uma destas seções pode ser iniciada a partir de valores que constam nos arquivos que contêm os programas a serem executados. É importante destacar que os arquivos contêm somente os valores **iniciais** de cada seção (e às vezes nem isso). Ao longo da execução, estes valores em memória podem ser alterados (nem todos), enquanto que o arquivo em disco não é mais usado.

Por aspectos de disponibilidade, transparência e documentação, esta parte do livro lida exclusivamente com ELF.

Este formato é adequado para sistemas que usam memória virtual. Com isso, é possível disponibilizar um espaço de memória imenso para cada programa (4 Gigabytes para cada programa em execução no ELF), apesar de a memória física não precisar ser tão grande.

A figura 2.1 mostra um programa em execução em memória (virtual). Observe que o programa em execução é composto por várias partes (chamadas seções): seção *text*, *data*, *bss*, *stack*, e áreas de uso exclusivo do sistema operacional (se um programa tentar endereçar esta área, o programa será cancelado por tentativa de acesso a um segmento inválido[3]).

Esta parte do livro aborda cada uma das seções individualmente. O capítulo 3 descreve alguns aspectos da seção de código (*text*), e também alguns aspectos de variáveis globais. O capítulo 4 descreve como o programa em execução utiliza a pilha (*stack*)

[1] Ao longo dos anos, muitas pessoas se perguntaram se o sistema operacional uindous é um vírus ou um bug. Para responder isso, lembre-se que um vírus é um programa pequeno e eficiente :o)

[2] Executable and Linking Format

[3] Onde aparece a simpática mensagem *segmentation fault*

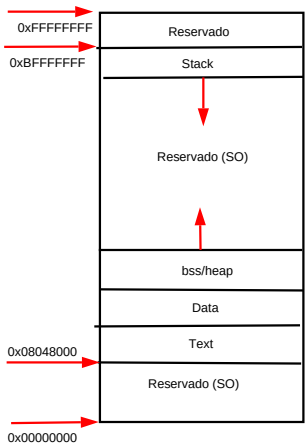

Figura 2.1: Programa em execução na memória virtual (formato ELF)

para fazer chamadas de procedimento. O capítulo 5 descreve a utilidade da área da bss, onde são alocadas as variáveis dinâmicas. Por fim, o capítulo 6 descreve algumas chamadas de sistema.

O método de apresentação baseia-se na tradução de pequenos programas que estão na linguagem C para assembly do ix86 e está fundamentado nos conceitos de execução do sistema operacional linux.

Capítulo 3

A Seção de Código e de Dados

Este capítulo descreve primeiramente a seção do programa em execução que contém as instruções do programa. Esta seção é chamada de seção de texto ou *text*. Incidentalmente também descreve alguns aspectos de variáveis globais (seção de dados ou *data*), finalizando com vetores.

A figura 3.1 mostra que parte do programa em execução será abordado. Como pode ser visto, a área de código começa sempre no endereço virtual `0x08048000`. Como os diagramas deste texto consideram que endereço 0x00000000 está "em baixo" e o endereço 0xFFFFFFFF está em cima, o programa em execução tem seu menor endereço em `0x08048000`, e as instruções são copiadas do arquivo para este endereço "para cima". Assim que a última instrução é copiada, começa a área de dados (cujo endereço de início depende do número de instruções do programa). É importante destacar que o montador consegue determinar qual é o endereço de início dos dados, uma vez que para tal basta contar o número de bytes usados pelas instruções e somar à base (`0x08048000`). Logo em seguida inicia a área de dados, cujo tamanho varia de acordo com a quantidade de variáveis globais contidas no programa. Também é importante destacar que é possível determinar o endereço de cada uma destas variáveis em tempo de montagem do programa (basta somar o endereço de início dos dados com o tamanho de cada variável).

O capítulo está organizado da seguinte forma: a seção 3.1 apresenta o modelo de um programa em assembly, que aqui chamamos de "esqueleto de programas", e as demais seções incluem comandos sobre este esqueleto. A seção 3.2 apresenta alguns aspectos de variáveis globais e como traduzir expressões aritméticas. A seção sec:repetitivos descreve como traduzir os comandos repetitivos da linguagem C para assembly (formato geral), enquanto que a seção 3.4 aplica os conceitos gerais para traduzir os comandos do tipo `while`. A seção 3.5 descreve como traduzir comandos condicionais (na verdade só o comando `if`). Para finalizar, a seção 3.6 contém um exemplo que usa comandos repetitivos e condicionais, onde se descreve como trabalhar com vetores em assembly.

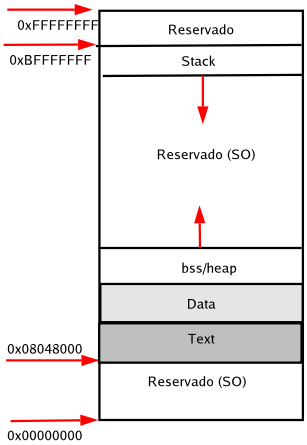

Figura 3.1: Programa em execução na memória virtual - área de texto e dados globais (formato ELF)

3.1 Esqueleto de programas em assembly

Este capítulo apresenta um programa escrito em linguagem C, semelhante ao algoritmo 33), cuja única função é terminar graciosamente. O programa tem poucos comandos, e o objetivo é mostrar o esqueleto de um programa traduzido de C para assembly, e posteriormente alguns aspectos da execução de programas. A tradução do algoritmo 4 é apresentada em 7.

```
1 main ( int argc, char** argv);
2 {
3     return 13;
4 }
```

Algoritmo 1: Arquivo prog1.c

Neste programa, o único comando é o da linha 3, `return 13`. Este comando finaliza o programa, e retorna o número 13, que pode ser visto pelo comando `echo $?`.

A tradução acima não é literal, uma vez que o `main` é uma função e deveria ter sido traduzida como tal (como será visto no capítulo sobre tradução de funções). Porém, é uma tradução válida, uma vez que a funcionalidade do programa assembly equivale à funcionalidade do programa em C.

Programas em assembly são divididos em seções, e o programa traduzido contém duas seções. A seção `.data` contém as variáveis globais do programa (como este programa não usa variáveis globais, esta seção está vazia) enquanto que a seção `.text` contém os comandos a serem executados quando o programa for colocado em execução.

```
1 .section .data
2 .section .text
3 .globl _start
4 _start:
5     movl $1, %eax
6     movl $13, %ebx
7     int $0x80
```

Algoritmo 2: Arquivo prog1.s

A linha 4 contém um rótulo, ou seja, um nome associado a um endereço. Na sintaxe do assembly que estamos usando (como em vários outros), o rótulo é sempre um conjunto de caracteres e letras terminadas por dois pontos (`:`).

O rótulo `_start` é especial e deve sempre estar presente. Ele corresponde ao endereço da primeira instrução do programa que será executada, e deve ser declarada como global (linha 3). As demais instruções serão executadas na seqüência.

Os comandos necessários para gerar o arquivo executável estão descritos a seguir:

```
> as  prog1.s -o prog1.o
> ld prog1.o -o prog1
> ./prog1
> echo \$?
13
```

A execução descrita acima não esclarece como o programa é executado. Para tal, utilizaremos o programa "ald" (assembly language debugger), para executar o programa, passo a passo.

O primeiro passo é, na linha de prompt, digitar "ald", e depois das mensagens iniciais digitar `load prog1`. Isto fará com que o programa indicado seja colocado para iniciar a simulação. Para iniciar simulação, digite "s" (step), e a primeira instrução será impressa, apesar de não ser executada. Outras informações impressas são o conteúdo de cada registrador (eax, ebx, ecx, edx, esp, ebp, esi, edi, ds, es, fs, gs, ss, cs, eip, eflags) da CPU.

A primeira instrução é `mov eax, 0x1`[1].

Existem outras informações incluídas, que em minha máquina são: `08048079` e `B801000000`. A primeira indica o endereço da instrução e a segunda corresponde ao padrão binário da instrução. Para deixar este ponto mais claro, digite `examine 0x08048070`, que nos permite examinar o conteúdo da memória a partir do endereço indicado. Procure o padrão `B8 01 00 00 00` na primeira linha. Os valores hexadecimais que seguem correspondem às demais instruções do programa.

[1]Observe que a ordem dos parâmetros está invertida quando comparada com o programa. Isto ocorre porque há dois tipos de formado de programas assembly para o x86: o formato Intel e o formato AT&T. O programa foi escrito obedecendo o formato AT&T (próprio para o montador `as`, desenvolvido pelo mesmo grupo que implementou o programa gcc (gnu), enquanto que o ald apresenta comandos no formato Intel. Ambos os formatos são equivalentes, e a maior diferença entre eles é a ordem dos parâmetros. Os programas que apresentaremos neste texto estão sempre no formato AT&T.)

3.2 Expressões Aritméticas

Este capítulo mostra como traduzir para assembly os programas escritos em linguagem C que contém somente expressões aritméticas que usam variáveis globais.

```
1 int a, b;
2 main ( int argc, char** argv);
3    {
4       a=6;
5       b=7;
6       a = a+b;
7       return a;
8    }
```

Algoritmo 3: Arquivo prog2.c

```
1  .section .data
2     a: .int 0
3     b: .int 0
4  .section .text
5  .globl _start
6  _start:
7     movl $6, a
8     movl $7, b
9     movl a, %eax
10    movl b, %ebx
11    addl %eax, %ebx
12    movl $1, %eax
13    int $0x80
```

Algoritmo 4: Arquivo prog2.s

O primeiro aspecto a ser levantado é como as variáveis globais “a” e “b” foram declaradas no programa assembly (linhas 2 e 3 do arquivo prog2.s), ou seja, como rótulos (veja a definição de rótulo na página 19).

Estes dois rótulos foram declarados na seção “.data”, o que indica que quando o programa for colocado em execução, devem ser reservados espaços de inteiro (.int - quatro bytes no x86) para cada um deles, e iniciado com zero.

Para deixar este ponto mais claro, vamos verificar como o programa será executado. Após montar e ligar o programa, obteremos o arquivo executável prog2. Vamos utilizar novamente o simulador ald e analisar a segunda instrução (linha 8).

```
0804807E C705A090040807000000 mov dword [+0x80490a0], 0x7
```

O primeiro valor impresso, `0804807E`, corresponde ao endereço onde está localizada esta instrução enquanto que `C705A090040807000000` corresponde ao código da instrução, e operandos, decodificados ao lado (`mov dword [+0x80490a0], 0x7`). Esta é a instrução que corresponde a colocar a constante `0x7` no endereço `0x80490a0`. Este instrução corresponde à instrução da linha 8 em nosso arquivo "prog2.s": `movl $7, b`. Desta forma, deduz-se que, em tempo de execução, o endereço de "b" é `0x80490a0`.

É importante destacar que quem define o endereço de "a" e de "b" é o ligador, e não o carregador. Para comprovar isso, observe o resultado abaixo, do conteúdo do arquivo objeto.

```
Lixo@fradim$ objdump prog2.o -S

prog2.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0: c7 05 00 00 00 00 06  movl   $0x6,0x0
   7: 00 00 00
   a: c7 05 04 00 00 00 07  movl   $0x7,0x4
  11: 00 00 00
  14: a1 00 00 00 00        mov    0x0,%eax
  19: 8b 1d 04 00 00 00     mov    0x4,%ebx
  1f: 01 c3                 add    %eax,%ebx
  21: b8 01 00 00 00        mov    $0x1,%eax
  26: cd 80                 int    $0x80
```

Compare este resultado com o conteúdo do arquivo executável:

```
> objdump -S prog2
prog2:     file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074: c7 05 9c 90 04 08 06  movl   $0x6,0x804909c
 804807b: 00 00 00
 804807e: c7 05 a0 90 04 08 07  movl   $0x7,0x80490a0
 8048085: 00 00 00
 8048088: a1 9c 90 04 08        mov    0x804909c,%eax
 804808d: 8b 1d a0 90 04 08     mov    0x80490a0,%ebx
 8048093: 01 c3                 add    %eax,%ebx
 8048095: b8 01 00 00 00        mov    $0x1,%eax
 804809a: cd 80                 int    $0x80
```

Em especial, compare os resultados da instrução `movl $6, a`. No arquivo objeto, temos `movl $0x6,0x0` enquanto que no arquivo executável, temos `movl $0x6,0x804909c`. No arquivo objeto, a referência ao símbolo `"a"` é zero (`0x0`), que é só uma referência. Algo como um aviso ao ligador: este símbolo deve ser alocado no primeiro endereço disponível da região de dados. Quando o ligador gera o executável, ele observa isso e aloca este símbolo no endereço virtual disponível na seção de dados, ou seja,

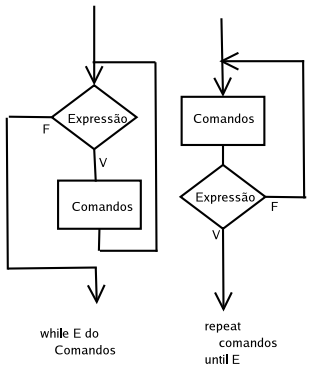

Figura 3.2: Funcionalidade das construções while e repeat.

`0x804909c`. Já o símbolo `"b"` no arquivo objeto está indicado para ser alocado em `$0x4`, ou seja, quatro bytes depois de `"a"`. O ligador então aloca-o em `0x80490a0 = 0x804909c + 0x4`.

Os processadores da família intel definem os seguintes tipos fundamentais de dados: bytes (8 bits), words (16 bits), double words (32 bits) quadwords (64 bits) e double quadwords (128 bits).

Aliás, o termo "variável" em assembly é diferente do termo utilizado em linguagens de programação fortemente tipadas (como Pascal), onde cada variável é de um tipo e deve passar por alguma operação especial para ser convertida para outro tipo. Aqui, as duas variáveis ocupam quatro bytes, e não estão associados a nenhum tipo (.int indica que deve ser alocado espaço necessário para um inteiro, ou seja, 32 bits). Isto significa que tanto "a" quanto "b" podem ser usadas com instruções sobre bytes, words, etc. Nem o montador nem o ligador fazem checagem de tipo - isto é responsabilidade do programador!

Outro ponto importante é que a instrução de soma utilizada, `addl`, soma o conteúdo de dois registradores (neste caso `%eax` e `%ebx`, jogando o resultado em `%ebx`. Esta operação soma dois inteiros de 32 bits com sinal (complemento de dois).

Para outros tipos, formatos de somas, e outras instruções para operações aritméticas, veja [Int04b].

3.3 Comandos Repetitivos

Comandos repetitivos são aqueles que permitem que um conjunto de instruções seja repetido até que uma determinada condição ocorra. Em linguagens de alto nível, normalmente são divididos em comandos do tipo "repeat" e comandos do tipo "while". Um "repeat" indica que o conjunto de instruções pode ser repetido de um a várias vezes, enquanto que um "while" indica que o conjunto de instruções pode ser repetido de zero a várias vezes. A forma de operação destas construções está apresentada na figura 3.2.

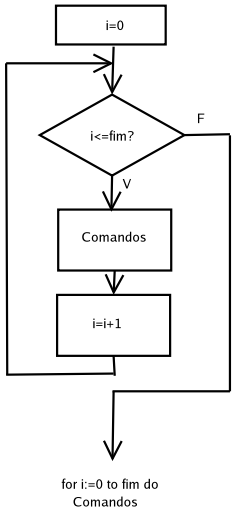

Figura 3.3: Funcionalidade da construção for.

Há também a construção do tipo “for”, que indica que um conjunto de instruções deve ser repetido um determinado número de vezes (figura 3.3). Esta construção é semelhante ao comando while, porém acrescida de uma variável de controle e de:

1. um comando para iniciar a variável de controle antes do primeiro teste;
2. um comando para acrescentar a variável de controle de um após a execução da seqüência de comandos especificada;
3. que a expressão verifica se o valor da variável de controle ultrapassou o limite.

Estas construções presentes em linguagens de alto nível são traduzidas para linguagem assembly pelos compiladores. O nosso próximo passo é descrever como esta tradução ocorre.

Em linguagens assembly, estas construções não estão presentes. Os comandos como `addl`, `movl`, `int`, etc. indicam implicitamente que a instrução seguinte é a próxima a ser executada[2]. Existem várias instruções de desvio, porém esta seção se concentra em dois tipos de comandos: os comandos de desvio incondicional e as de desvio condicional

Os dois tipos de desvios são implementados em associação com rótulos, ou seja, os comandos de desvio indicam que o fluxo de execução deve ser desviado para o rótulo

[2]Aqui cabe uma pequena explanação sobre o hardware. Em todas as CPUs (que adotam o modelo von Neumann) existe um registrador especial que contém o endereço da próxima instrução a ser executada. Nos x86, é o registrador `%eip`. Basta somar o número de bytes ocupados pela instrução com o valor (endereço) contido neste registrador para saber qual o endereço da próxima instrução. Para desviar o fluxo, basta colocar o endereço da instrução desejada no `%eip`. Intuitivamente, uma instrução `movl %eip, ...` seria uma alternativa para o desvio incondicional, porém o programa não pode escrever diretamente sobre este registrador.

indicado. Como exemplo, considere o algoritmo 5, que incrementa o valor em `%eax` de um até ultrapassar o valor de `%ebx`. Este programa tem três rótulos (`_start`, `loop`, `fim_loop`) e duas instruções de desvio (`jg` e `jmp`), que estão para desvia se maior (*jump if greater*) e desvio incondicional (*jump*).

O rótulo `_start`, como já vimos, indica o local onde a execução do programa deve iniciar. O rótulo `loop` indica o ponto de início do laço e o rótulo `fim_loop` indica o ponto de saída do laço, que será executada assim que a condição de término for atingida.

```
1  .section .text
2  .globl _start
3  _start:
4      movl $0, %eax
5      movl $10, %ebx
6  loop:
7      cmpl %ebx, %eax
8      jg fim_loop
9      add $1, %eax
10     jmp loop
11 fim_loop:
12     movl $1, %eax
13     int $0x80
```

Algoritmo 5: Arquivo rot_e_desvios.s

Além das instruções de desvio, o programa apresenta um outro comando novo: `cmpl`. Esta instrução compara o SEGUNDO argumento com o primeiro (no caso, `%eax com %ebx`) colocando o resultado em um bit de um registrador especial (EFLAGS). Este registrador é afetado por vários tipos de instrução, e contém informações sobre a última instrução executada, como por exemplo se ocorreu overflow (que afeta o bit de carry).

Os bits deste registrador podem ser testados individualmente, e no caso da operação jump if greater, verifica se o bit ZF (zero flag) é igual a zero e se SF=OF (SF=Sign Flag e OF=Overflow Flag). Para mais detalhes sobre o funcionamento do EFLAGS, veja [Int04a, pp 3-12].

Observe que se uma instrução for colocada entre a comparação (`cmpl`) e o desvio, os bits do EFLAGS provavelmente serão afetados. Isto implica dizer que a instrução de desvio `jg` deve ser colocada imediatamente após a comparação.

Outras instruções de desvio que serão utilizadas ao longo deste texto são:

- `jge` (*jump if greater or equal*),
- `jl` (*jump if less*),
- `jle` (*jump if less or equal*),
- `je` (*jump if equal*),

- `jne` (*jump if not equal*).

Estas instruções também funcionam analisando os bits de EFLAGS.

Outro aspecto importante a ser destacado neste programa é que ele segue a funcionalidade do comando "for". Compare o programa com a figura 3.3, e observe que no desenho, o fim do loop é atingido quando a condição for falsa (ou seja, quando a variável de controle não é mais menor ou igual ao limite), e que no programa acima o teste é se a condição de fim de loop foi atingida (ou seja, que a variável de controle é maior do que o limite).

Como exercício, simule a execução do programa acima, prestando atenção ao valor que o registrador EIP (extended instrucion pointer) assume. Fiz isto em meu simulador, e obtive o seguinte resultado:

	%eip	instrução (hexa)	instrução
1	08049074	B800000000	movl 0x0, %eax
2	08049079	BB0A000000	movl 0xa, %ebx
3	0804907E	39D8	cmpl %eax, %ebx
4	08049080	7F05	jg fim_loop
5	08049082	83C001	addl %eax, 0x1
6	08049085	EBF7	jmp loop
7	0804907E	39D8	cmpl %eax, %ebx
8	08049080	7F05	jg fim_loop
9	08049082	83C001	addl %eax, 0x1
10	08049085	EBF7	jmp loop
	...	...	...

Cada linha contém exatamente uma instrução. A coluna da esquerda indica a ordem de execução das instruções. A coluna %eip indica o endereço da instrução, e a terceira coluna indica o conteúdo daquele endereço. A última coluna mostra o código mnemônico da instrução hexadecimal indicada. Desta forma, a primeira instrução executada estava no endereço `0x08049074` (endereço do rótulo `_start`). A instrução contida naquele endereço é `movl 0x0,%eax`, cujo código de máquina (em hexadecimal), é `0xB800000000`.

Como esta instrução ocupa cinco bytes, a próxima instrução deverá estar localizada em `0x08049074 + 0x00000005 = 0x08049079`. Este é o valor de `%eip` na instrução seguinte.

Desta forma, a instrução seguinte está localizada alguns bytes adiante da instrução atual, e o endereço da nova instrução pode ser facilmente calculado somando o endereço atual com o tamanho da instrução atual.

Esta lógica de funcionamento é quebrada quando ocorre um desvio. Observe o que ocorre após a instrução `jmp loop`. A instrução seguinte está no endereço `0x0804907E`, que não coincidentemente é o mesmo endereço do rótulo `loop`. Este endereço é determinado em tempo de ligação (ou seja, pelo ligador), e pode ser observado no código executável (verifique com o comando objdump).

Com este exemplo é possível verificar na prática que rótulos são utilizados para indicar endereços que tem alguma importância para o programa.

3.4 Tradução da construção While

A tradução de uma construção while segue o fluxo de execução apresentado na figura 3.2, onde os desvios no fluxo são substituídos por comandos de desvio em assembly e os "pontos de entrada" dos desvios são substituídos por rótulos.

Como exemplo, considere os algoritmos 6 e 7.

```
1 main ( int argc, char** argv)
2    {
3    ...
4    while ( E )
5       {
6          C_1, C_2, ... C_n;
7       }
8    ...
9    }
```

Algoritmo 6: Comando while (Linguagem de alto nível).

```
1  .section .text
2  .globl _start
3  _start:
4  ...
5  while:
6     Tradução da Expressão (E)
7     jFALSO fim_while
8     Tradução dos Comandos (C_1, C_2, ... C_n)
9     jmp while
10 fim_while:
11 ...
```

Algoritmo 7: Tradução do comando while da figura 6 para assembly.

Os comandos que precedem o while (os pontilhados da linha 3 do algoritmo 6) são traduzidos, um a um, e estão representados na linha 4 do algoritmo 7. De forma análoga se traduz os comandos de sucedem o while.

Quando encontramos o início do while (linha 4 do algoritmo 6), o arquivo assembly recebe um rótulo `while:`, que indica o início da construção. Em seguida, traduz-se a Expressão, e caso ela seja falsa, o fluxo é desviado para o rótulo `fim_while` (linhas 5-7 do algoritmo 7). O desvio da linha 6 (jFALSO) deve ser substituída por algum dos comandos de desvio (página 24).

Cada comando em linguagem de alto nível é traduzida individualmente entre o início e o fim da construção. Após o último comando do while, o arquivo assembly apresenta uma instrução de desvio incondicional (linha 9 do algoritmo 7). Isto indica que a instrução seguinte a ser executada é aquela indicada pelo rótulo `while`.

As instruções do algoritmo 7 que estão em negrito correspondem às instruções que dão a funcionalidade do comando while.

Como exemplo de tradução de um programa completo, considere os algoritmos 8 e 9.

```
1  int i, a;
2  main ( int argc, char** argv)
3  {
4     i=0; a=0;
5     while ( i<10 )
6        {
7           a+=i;
8           i++;
9        }
10    return (a);
11 }
```

Algoritmo 8: Programa while.c

Há um aspecto importante ser levantado no programa traduzido (algoritmo 9): o acesso às variáveis.

As variáveis globais “i” e “a” do programa em linguagem C foram mapeados para rótulos no arquivo assembly.

Quando ocorre uma atribuição a alguma variável de um programa de alto nível, esta atribuição é mapeada para o endereço da variável correspondente (neste caso, variável global). Porém, como o acesso à memória é mais lento do que o acesso a registradores, é mais eficiente mapear as variáveis em registradores e minimizar os acessos à memória.

Desta forma, podemos construir um programa equivalente ao programa contido no algoritmo 9 ao considerar que “i” está mapeado no registrador `%eax` e que “a” está mapeado no registrador `%ebx`. Desta forma, podemos eliminar as linhas 2, 3, 6, 7, 8, 12, 14 e 16, para obter o algoritmo 10. Este programa é equivalente ao programa do algoritmo 9, porém como ele só acessa registradores, é mais rápido.

Um dos desafios da tradução de programas escritos em linguagens de alto nível para programas em linguagem assembly é maximizar a quantidade de variáveis mapeadas em registradores, e com isso melhorar o desempenho do programa. Os compiladores são capazes de fazer esta tarefa muito bem, porém o resultado final depende muito da quantidade de registradores que estão disponíveis na arquitetura alvo. Por esta razão, a maior parte dos processadores modernos são projetados com um grande número de registradores, uma vez que quando o número de registradores é pequeno (como por exemplo nos processadores da família ix86 que só tem oito registradores de propósito geral), a tendência é que programas que usam muitas variáveis tenham um desempenho inferior daquele obtido em processadores com muitos registradores. É importante observar que é cada vez mais incomum encontrar programas úteis que usam poucas variáveis.

Uma solução freqüente para melhorar o desempenho dos processadores é anexar

```
1  .section .data
2  i: .int 0
3  a: .int 0
4  .section .text
5  .globl _start
6  _start:
7  movl $0, i
8  movl $0, a
9  movl i, %eax
10 while:
11     cmpl $10, %eax
12     jge fim_while
13     movl a, %ebx
14     addl %eax, %ebx
15     movl %ebx, a
16     addl $1, %eax
17     movl %eax, i
18     jmp while
19 fim_while:
20 movl $1, %eax
21 int $0x80
```

Algoritmo 9: Tradução do programa while.c (algoritmo 8) para assembly.

```
1  .section .text
2  .globl _start
3  _start:
4  movl $0, %eax
5  while:
6      cmpl $10, %eax
7      jge fim_while
8      movl a, %ebx
9      addl %eax, %ebx
10     addl $1, %eax
11     jmp while
12 fim_while:
```

Algoritmo 10: Tradução do programa while.c (algoritmo 9) sem acessos à memória.

uma memória super-rápida, bem próxima à CPU (ou até dentro), chamada de memória cache, cuja capacidade de armazenamento de dados é muito superior à capacidade de armazenamento de dados do processador e o tempo de acesso não é muito maior do que o acesso a registradores. Porém o inconveniente é o custo, que é **muito** superior à memória convencional. Por esta razão, as memórias cache são normalmente pequenas.

3.5 Comandos Condicionais

As linguagens de programação apresentam normalmente pelo menos dois tipos de comandos condicionais: `if-then` e `if-then-else`. Se a expressão testada for verdadeira, a seqüência de comandos contida nos comandos assembly relativos ao "then" deve ser executada, enquanto que se a condição for falsa, a seqüência de comandos do "else" deve ser executada (se houver "else", obviamente). Os comandos do "then" e "else" não podem ser executados consecutivamente - ou executa um ou outro, mas nunca os dois.

O modelo de um comando condicional em C é apresentado no algoritmo 11 e o arquivo assembly correspondente é apresentado no algoritmo 12.

```
1  main ( int argc, char** argv)
2  {
3  ...
4  if ( E )
5     {
6        C_then1, C_then2, ... C_thenn;
7     }
8  else
9     {
10       C_else1, C_else2, ... C_elsem;
11       }
12    ...
13    }
```

Algoritmo 11: Comando If-Then-Else (Linguagem de alto nível).

A tradução segue os moldes apresentados na estrutura repetitiva. A execução depende do resultado da expressão. Se a expressão for verdadeira, a seqüência de instruções incluirá as linhas 4,5,6,7,8,13. Se a expressão for falsa, a seqüência seguirá as linhas 4,5,9,10,11,12,13. No primeiro caso, serão executados os comandos "then" e no segundo os comandos relativos a "else". Não há nenhuma forma de se executar os comandos relativos a "then" e "else" ao mesmo tempo.

Como exercício, traduza o algoritmo 13 para assembly, monte e execute o programa através do `ald`. Em seguida, altere os valores das variáveis para que a execução siga a outra alternativa. A idéia com este exercício é se familiarizar com todo o processo de geração do programa executável, com o fluxo de execução, e principalmente com o modelo de memória virtual que é adotada no linux (endereço dos rótulos da seção .data

```
1  .section .text
2  .globl _start
3  _start:
4  . . .
5     Tradução da Expressão (E)
6     jFALSO else
7     Tradução dos Comandos do then (C_then1, C_then2, . . . C_thenn; )
8     jmp fim_if
9  else:
10    Tradução dos Comandos do else (C_else1, C_else2, . . . C_elsen; )
11 fim_if:
12 . . .
```

Algoritmo 12: Tradução do comando if-then-else do algoritmo 11 para assembly.

e da seção .text, endereço de início do programa, etc.).

```
1  int a, b;
2  main ( int argc, char** argv)
3  {
4     a=4; b=5;
5     if (a>b) {
6        a=a+b;
7        return a;
8     }
9     else
10    {
11       a=a-b;
12       return a;
13    }
14 }
```

Algoritmo 13: Exemplo para tradução.

3.6 Vetores

Para finalizar o capítulo, esta seção apresenta um exemplo que combina comandos condicionais e repetitivos. Para aprofundar o conhecimento da seção de dados, este exemplo utiliza um vetor, e descreve as formas de acessar dados em vetores.

O programa é apresentado no algoritmo 14, e contém um vetor (fixo) de dados. Este programa retorna em `$?` o valor do maior elemento do vetor. A variável `maior` armazena sempre este valor a cada iteração.

A tradução deste programa é apresentada no algoritmo 15, e as novidades são:

```
1  int data_items[]={3, 67, 34, 222, 45, 75, 54, 34, 44, 33, 22, 11, 66, 0};
2  int i, int maior;
3  main ( int argc, char** argv)
4  {
5     maior = data_items[0];
6     for (i=1; data_items[i] != 0; i++)
7     {
8        if (data_items[i] > maior)
9        maior = data_items[i];
10    }
11    return (maior);
12 }
```

Algoritmo 14: Arquivo vetor.c

1. a forma de criar um vetor com valores fixos. Basta listá-los lado a lado na seção `data`, ao lado do rótulo associado àquele vetor (veja a linha 4 do algoritmo 15.

2. a forma de referenciar cada elemento do vetor: `movl data_items(, %edi, 4), %ebx`. Este comando indica que o endereço do dado a ser movido para `%ebx` é o endereço de `data_items` somado com $4 \times \%edi$. Como `%edi` é incrementado de um a cada passo da iteração, o endereço do dado procurado é incrementado de quatro a cada iteração (ou seja, de um elemento do vetor de inteiros). Para detalhes sobre os tipos de endereçamento utilizados, veja o apêndice A.1.

```
1  .section .data
2  i: .int 0
3  maior: .int 0
4  data_items: .int 3, 67, 34, 222, 45, 75, 54, 34, 44, 33, 22, 11, 66, 0
5  .section .text
6  .globl _start
7  _start:
8  movl $0, %edi
9  movl data_items(, %edi, 4), %ebx
10 movl $1, %edi
11 loop:
12 movl data_items(, %edi, 4), %eax
13 cmpl $0, %eax
14 je fim_loop
15 cmpl %ebx, %eax
16 jle fim_if
17 movl %eax, %ebx
18 fim_if:
19 addl $1, %edi
20 jmp loop
21 fim_loop:
22 movl $1, %eax
23 int $0x80
```

Algoritmo 15: Arquivo vetor.s

Capítulo 4

A Seção da Pilha

A seção da pilha (*stack*) armazena informações sobre chamadas de procedimento (parâmetros, informações sobre contexto e variáveis locais).

A idéia de armazenar estas informações em uma pilha é originária da linguagem de programação Algol 60[1]. Antes desta linguagem todas as variáveis eram globais e não havia procedimentos. Expoentes desta época foram as linguagens Fortran e Cobol, que posteriormente foram adaptadas para comportar procedimentos.

Antes de detalhar o tema, é necessário compreender o que ocorre em uma chamada de procedimento, ou seja, o modelo utilizado para implementar chamadas de procedimento em praticamente todas as linguagens de programação atuais.

A figura 4.1 mostra a parte do programa em execução que será abordado neste capítulo, que corresponde ao intervalo `0xbfffffff` e o valor atual do registrador `%esp`. Existem instruções assembly para valores na pilha e para retirá-los de lá que serão abordados na seqüência.

4.1 Modelo de Chamadas de Procedimento

Em termos abstratos, o modelo de chamadas de procedimento (independente de sistema operacional e de arquitetura de computador), assemelha-se ao modelo de folhas de papel como descrito a seguir (baseado em [Kow83]).

Quando um programa é colocado em execução, uma grande folha de papel (digamos uma folha em formato A2) é colocada sobre uma mesa. Nesta folha, escrevemos (por exemplo no canto superior esquerdo) o nome de todas as variáveis globais do programa. O valor de iniciação da variável é normalmente indefinido, e qualquer atribuição a esta variável se sobrepõe ao valor que consta naquela folha.

Cada vez que um procedimento é chamado, uma folha de papel menor (por exemplo, formato A5) é colocada sobre as demais (ou somente sobre a primeira) e nela é

[1] As referências não são definitivas, mas a idéia aparentemente é oriunda dos estudos do pesquisador alemão Friedrich L. Bauer, que também trabalhou no desenvolvimento da linguagem, no uso da pilha para cálculos de expressões aritméticas e lógicas. Ele até ganhou um prêmio de pioneirismo da IEEE (IEEE Computer Society Pioneer Award) em 1988 pelo seu trabalho.

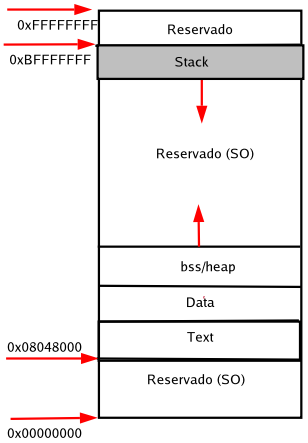

Figura 4.1: Programa em execução na memória virtual - área da pilha (formato ELF)

escrito o nome do procedimento e todas a variáveis (parâmetros e variáveis locais) relativas àquele procedimento. Assim como ocorre nas variáveis globais, as atribuições a variáveis descritas naquela folha se sobrepõe, ou seja, não alteram o valor anterior (ou em outras folhas).

Através deste modelo fica mais fácil compreender o que ocorre quando da execução de um procedimento recursivo. Em cada chamada recursiva, as variáveis a serem acessadas são somente aquelas da folha de papel no topo da pilha (parâmetros e variáveis locais daquele procedimento) ou à primeira folha de papel (variáveis globais).

Cada nova chamada de procedimento (ou seja, cada nova instância da recursão), uma nova folha de papel é colocada sobre as anteriores, de tal forma que quando a execução de uma determinada instância da recursão chegar ao fim, a folha de papel associada a ele (a folha de papel no topo) deverá ser eliminada e a folha de papel anterior (ou seja, da instância recursiva anterior) serão utilizados, e as variáveis voltam a ter o valor que tinham antes de iniciar a instância que foi finalizada (excessão feita a variáveis passadas por referência, claro).

Considere a linguagem C. Quando uma variável é referenciada no programa, o primeiro lugar onde esta variável deve ser procurada é na folha de papel do topo. Se não estiver lá, deve ser procurada na folha de variáveis globais e se não estiver lá também, a variável não existe. Este trabalho não é realizado em tempo de execução, mas sim em tempo de compilação, e isso explica o erro comum que gera uma mensagem de erro indicando que uma determinada variável não foi declarada.

4.2 Implementação do Modelo em Uma Arquitetura

O modelo genérico apresentado na seção anterior é implementado de formas diferentes dependendo do conjunto Linguagem de programação/Sistema Operacional/CPU. Esta seção descreve a implementação para o conjunto Linguagem C/Linux/x86.

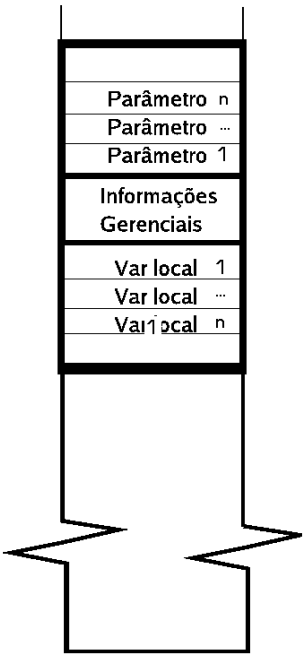

Figura 4.2: Registro de Ativação.

O primeiro aspecto importante a ser lembrado é que os endereços das variáveis e dos procedimentos são definidas em tempo de compilação. Os endereços dos procedimentos podem ser fixos (são associados a rótulos), o que não pode ser feito para as variáveis de cada folha de papel. Estas variáveis devem ser acessadas indicando a folha de papel onde ela está contida. Desta forma, cada folha de papel deve ser numerada e quando nos referirmos a uma determinada variável devemos dizer o número da folha de papel onde ela está embutida e qual a variável em questão.

Aliás, já está na hora de utilizarmos o nome correto para as "folhas de papel". Para um processo em execução, cada folha de papel é chamada de **registro de ativação**[2].

Um registro de ativação (figura 4.2) é instanciado sempre na área da pilha, e é basicamente composto por três partes:

- parâmetros: local onde é reservado espaço para as variáveis que correspondem aos parâmetros de um procedimento.
- informações gerenciais: esta área armazena basicamente o endereço da instrução de retorno e o valor anterior do registrador que indica o registro de ativação anterior, mas pode armazenar ainda outras informações.
- variáveis locais: espaço para as variáveis do procedimento.

[2]Em inglês *activation register*. Porém é também referenciado como *frame register*, *stack register*, entre outros.

A seguir descrevemos como montar (e desmontar) os registros de ativação para cada procedimento em tempo de execução. Observe que os comandos devem ser calculados em tempo de compilação, porém o efeito destas instruções (que implementam o registro de ativação) só poderá ser visto em tempo de execução.

A seção 4.2.1 descreve os registros de ativação sem parâmetros e sem variáveis locais. A seção 4.2.2 acrescenta as variáveis locais ao modelo da seção anterior e a seção 4.2.3 acrescenta os parâmetros ao modelo. A seção4.2.4 descreve parâmetros passados por referência e a seção 4.2.5 descreve alguns aspectos da função main da linguagem C. Para finalizar, a seção 4.2.6 apresenta um exemplo de uma função recursiva, que usa todos os conceitos apresentados e a seção 4.2.7 mostra como utilizar as funções disponíveis em bibliotecas, em especial na libc.

4.2.1 Sem Parâmetros e sem Variáveis Locais

De acordo com o modelo da figura 4.2, o registro de ativação de um programa que não tem parâmetros e nem variáveis locais só contêm as informações gerenciais. Estas informações, no caso da nossa arquitetura-alvo corresponde ao endereço de retorno (ou seja, o endereço que o programa deve retornar quando finalizar a execução do procedimento) e o valor de um determinado registrador (%ebp) que é usado para acessar as variáveis locais e os parâmetros, mas que neste exemplo não é usado para nada.

Para exemplificar como montar o registro de ativação neste caso, considere o algoritmo 16, traduzido para assembly no algoritmo 17.

```
1  int a, b;
2  int soma ( )
3  {
4      return (a+b);
5  }
6  main ( int argc, char** argv)
7  {
8      a=4;
9      b=5;
10     b = soma();
11     return (b);
12 }
```

Algoritmo 16: Programa sem parâmetros e sem variáveis locais.

Este programa apresenta dois novos pares de instruções assembly: `pushl` e `popl`, `call` e `ret`. O par `pushl` e `popl` insere e retira valores genéricos da pilha, enquanto que `call` e `ret` insere e retira endereços da pilha.

Começaremos descrevendo o par `pushl` e `popl`. A instrução `pushl <registrador>` empilha o valor contido no registrador indicado, e é equivalente às seguintes operações: (1) `subl $4, %esp` e (2) `movl registrador, (%esp)`. Já a instrução `popl <registrador>` faz a operação inversa, ou seja: `movl (%esp), <registrador>` e `addl $4, %esp`. Observe que, como a pilha cresce para

```
1  .section .data
2  A: .int 0
3  B: .int 0
4  .section .text
5  .globl _start
6  soma:
7      pushl %ebp
8      movl %esp, %ebp
9      movl A, %eax
10     movl B, %ebx
11     addl %eax, %ebx
12     movl %ebx, %eax
13     popl %ebp
14     ret
15 _start:
16     movl $4, A
17     movl $5, B
18     call soma
19     movl %eax, %ebx
20     movl $1, $eax
21     int $0x80
```

Algoritmo 17: Tradução do programa do algoritmo 16.

baixo, é necessário decrementar %esp para inserir um elemento e incrementar para retirá-lo.

O par `call` e `ret` é análogo, porém ao invés de empilhar/desempilhar o valor de um registrador qualquer, ele empilha/desempilha o valor do registrador %eip (instruction pointer), que indica qual o endereço da próxima instrução a ser executada.

Desta forma, a instrução `call <rotulo>` tem efeito equivalente a executar as instruções `pushl %eip` seguido de `jmp <rotulo>`, enquanto que ret tem efeito equivalente a `popl %eip`. Estas duas instruções são a base para a chamada de procedimento, uma vez que o endereço da instrução que seria executada após a chamada é empilhado com a instrução `call` e desempilhado e jogado em %eip na última instrução do procedimento, e este será o endereço da próxima instrução a ser executada.

A função retorna um número inteiro, e este valor é retornado em `%eax`. Este é o padrão para o ix86, porém outras arquiteturas podem indicar outras formas. Para entender melhor como funciona todo o processo, acompanhe a execução do programa assembly usando o simulador.

4.2.2 Sem Parâmetros e com Variáveis Locais

Um aspecto que ainda não foi levantado corresponde às instruções das linhas 7, 8, e 13 do algoritmo 17. Estas instruções salvam e atualizam o valor do registrador `%ebp`. O endereço contido neste registrador sempre aponta para o mesmo local do registro de

ativação corrente. Ao entrar em um novo procedimento, sempre é necessário salvar o valor atual de `%ebp` e ao sair é necessário restaurá-lo. A razão disso é que este registrador é usado para acessar as variáveis locais e as variáveis que correspondem aos parâmetros.

Observe que:

1. são sempre empilhadas duas informações gerenciais
2. `%ebp` sempre aponta para uma mesma posição no registro de ativação (logo abaixo do endereço de retorno e logo acima da primeira variável local, se esta existir).

Como as variáveis locais estão sempre logo abaixo do valor indicado por `%ebp`, indica-se o endereço delas usando `%ebp` como referência. Como exemplo, suponha que um programa contenha três variáveis locais inteiras. Então o endereço destas variáveis será: `%ebp-4`, `%ebp-8` e `%ebp-12`. Para tal, é necessário primeiro abrir espaço para elas. Como exemplo, considere o algoritmo 18, cujo código asssembly está indicado no algoritmo 19.

```
1  int a, b;
2  int soma ( )
3  {
4     int x, y;
5     x=a;
6     y=b;
7     return (x+y);
8  }
9  main ( int argc, char** argv)
10 {
11    a=4;
12    b=5;
13    b = soma();
14    return (b);
15 }
```

Algoritmo 18: Programa sem parâmetros e com variáveis locais.

No algoritmo 19, os endereços das variáveis locais `x` e `y` são `-4(%ebp)` e `-8(%ebp)` respectivamente, e o espaço para estas variáveis é aberto logo no início do procedimento ao subtrair 8 de `%esp`, quatro bytes para `x` e quatro para `y` (lembre-se que a pilha cresce para baixo). O valor de `%esp` é restaurado ao fim do procedimento, e é importante destacar que como ele foi alocado DEPOIS de `%ebp` ter sido empilhado, ele é liberado ANTES de restaurar `%ebp`.

É importante destacar que todas as CPUs modernas (que eu conheço) tem registradores específicos para lidar com o acesso de variáveis em registros de ativação. As CPUs x86 chamam este registrador de “base pointer”, enquanto que as CPUs MIPS chamam de “frame register”. Por vezes, o nome não é tão significativo quanto estas duas CPUs.

```
1  .section .data
2  A: .int 0
3  B: .int 0
4  .section .text
5  .globl _start
6  _start:
7  soma:
8      pushl %ebp
9      movl %esp, %ebp
10     subl $8, %esp
11     movl A, %eax
12     movl %eax, -4(%ebp)
13     movl B, %eax
14     movl %eax, -8(%ebp)
15     addl -4(%ebp), %ebx
16     movl -8(%ebp), %eax
17     addl $8, %esp
18     popl %ebp
19     ret
20 _start:
21     movl $4, A
22     movl $5, B
23     call soma
24     movl %eax, %ebx
25     movl $1, $eax
26     int $0x80
```

Algoritmo 19: Tradução do programa do algoritmo 18.

Algumas CPUs chamam este registrador de “frame pointer” (fp), e outros usam nomes mais ou menos significativos.

4.2.3 Com Parâmetros e com Variáveis Locais

Quando um procedimento tem parâmetros, eles devem ser empilhados ANTES de chamar o procedimento (antes do `call`), porém a ordem em que os parâmetros devem ser empilhados pode variar de uma linguagem para outra. Na linguagem Pascal, o padrão é que a ordem de empilhamento deve ser a mesma em que os parâmetros aparecem, enquanto que na linguagem C, a ordem é invertida, ou seja, o primeiro parâmetro (aquele que aparece primeiro após o “(“) é o último a ser empilhado, o segundo é o penúltimo e assim por diante. O motivo para esta ordem (que não é natural, diga-se de passagem) está relacionado com a implementação de funções que tem um número variável de argumentos, como por exemplo `printf` e `scanf`, onde o primeiro argumento (o string) indica quantos parâmetros foram (ou deveriam ter sido) empilhados.

Como estamos seguindo o padrão utilizado na linguagem C, adotaremos o segundo modelo. Como exemplo, considere o algoritmo 20, e sua versão assembly no algoritmo 21.

```
1  int a, b;
2  int soma ( int x, int y)
3  {
4     int z;
5     z = x + y;
6     return (z);
7  }
8  main ( int argc, char** argv)
9  {
10    a=4;
11    b=5;
12    b = soma(a, b);
13    return (b);
14 }
```

Algoritmo 20: Programa com parâmetros e com variáveis locais.

A diferença deste programa com relação aos anteriores é que os parâmetros foram empilhados antes da instrução `call`, nas linhas 20 e 21, na ordem inversa àquela em que que aparecem na linha de comando (primeiro foi empilhado o segundo parâmetro, depois o primeiro). Os parâmetros aparecem acima das informações gerenciais. Como `4(%ebp)` corresponde ao endereço de retorno, os endereços dos parâmetros `x` e `y` são `8(%ebp)` e `12(%ebp)` respectivamente. O programa também apresenta uma variável global que está localizada em `-4(%ebp)`.

A tabela 4.2.3 relaciona os endereços das variáveis em um registro de ativação com as suas posições na pilha. Como exemplo, considere o procedimento apresentado no algoritmo **??**.

```
1  .section .data
2  A: .int 0
3  B: .int 0
4  .section .text
5  .globl _start
6  _start:
7  soma:
8      pushl %ebp
9      movl %esp, %ebp
10     subl $4, %esp
11     movl 8(%ebp), %eax
12     addl 12(%ebp), %eax
13     movl %eax, -4(%ebp)
14     addl $4, %esp
15     popl %ebp
16     ret
17 _start:
18     movl $4, A
19     movl $5, B
20     pushl B
21     pushl A
22     call soma
23     addl $8, %esp
24     movl %eax, %ebx
25     movl $1, $eax
26     int $0x80
```

Algoritmo 21: Tradução do programa do algoritmo 20.

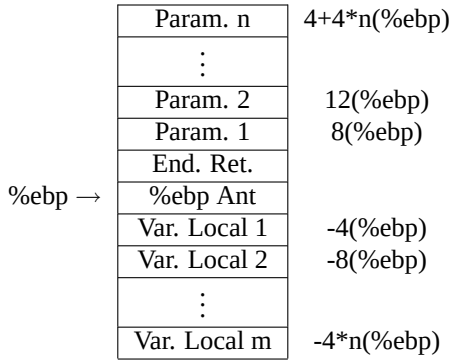

Tabela 4.1: Endereços Léxicos de um Registro de Ativação

```
1 Proc (int p1, int p2, int p3) {
2    int l1, l2, l3;
3    ...
4 }
5 ...
```

Algoritmo 22: Procedimento com parâmetros e variáveis locais

Observe que a "primeira" variável que aparece na lista de parâmetros (o que no caso de `Proc (int p1, int p2, int p3)`, corresponde a `p1`) é a variável que ocupa o primero espaço acima do endereço de retorno. Por isso mesmo, deve ser a última a ser empilhada.

Os endereços das variáveis `p1, p2, p3, l1, l2, l3` são relativos ao registrador `%ebp`. Por exemplo, o endereço de `p1` é `%ebp + 8`, ou `8(%ebp)`.

Quando o endereço de uma variável é indicado por dois números, uma base (o registrador `%ebp` e um deslocamento (para p1 o deslocamento é 8, para p2 é 12 e assim por diante), dizemos que este é o endereço léxico da variável[3].

4.2.4 Parâmetros passados por Referência e com Variáveis Locais

Os parâmetros da seção anterior foram passados por valor, ou seja, o valor foi copiado para a pilha. Isto faz com que a variável utilizada na chamada do procedimento não tenha o seu valor alterado quando do retorno.

Porém, por vezes é interessante que a variável que corresponde ao parâmetro seja alterada durante a execução da função. Para satisfazer este tipo de necessidade, muitas linguagens de programação usam o conceito de variável passada por referência. Na linguagem Pascal, um parâmetro passado por referência é precedido pela palavra reservada "var".

A linguagem C, usada neste texto, não contém este tipo de construção, porém provê mecanismos para que o programador crie uma construção análoga.

A idéia básica é que o valor a ser empilhado na chamada não é uma cópia do valor a ser passado, porém uma cópia do **endereço** da variável. Toda vez que a variável for acessada, deve ser indicado o deseja-se acessar o valor *apontado* por aquele parâmetro e não o parâmetro em si. Como exemplo, considere o algoritmo 24.

Neste algoritmo, a função `troca` recebe dois parâmetros, `x` e `y` e os troca. Ao final do programa principal, `a` terá o valor 2 e `b` terá o valor 1. Alguns aspectos merecem destaque:

1. na chamada do procedimento `troca (&a, &b)` o símbolo `&` antes de uma variável indica que deve ser empilhado o *endereço* desta variável e não o seu valor.

2. a função `void troca ( int* x, int* y)` indica que os dois parâmetros são *apontadores*, ou seja, contêm os endereços das variáveis a serem acessa-

[3] na bibliografia, o registrador de base é o que aponta para o registro de ativação, e por isso a base também é chamada de *frame pointer* ou *frame number*.

```
1  int a, b;
2  void troca ( int* x, int* y)
3  {
4     int z;
5     z = *x;
6     x = *y;
7     y = z;
8  }
9  main ( int argc, char** argv)
10 {
11    a=1;
12    b=2;
13    troca (&a, &b);
14    exit (0);
15 }
```

Algoritmo 23: Programa com parâmetros passados por referência e com variáveis locais.

das. Por esta razão, o comando `z = *x;` diz que o valor indicado pelo endereço contido em `x` deve ser copiado para `z`.

Dadas estas explicações, fica mais simples indicar a tradução do programa acima (algoritmo 24).

O programa assembly reflete as idéias apresentadas anteriormente:

1. A chamada da função (`troca (&a, &b)`) empilha os endereços das variáveis (linhas 25 e 26).

2. Os acessos aos parâmetros são traduzidos da seguinte forma: A construção `*<var>`, corresponde a duas instruções: `movl <paramPilha>, %registrador1`, movl (%registrador1), %registrador2. Como exemplo de tradução de `*x`, temos as instruções 15 e 16 do algoritmo 24.

Esperamos que a presente seção tenha esclarecido como e quando usar as construções `&<var>` e `*<var>`, que são de uso misterioso para todos os programadores iniciantes na linguagem C.

4.2.5 A função main da linguagem C

Cada linguagem de programação define um local para iniciar o programa. Na linguagem C, este local é o procedimento `main`. Até este momento, associamos este procedimento ao rótulo `start`, que indica o local onde um programa assembly tem início. Porém as coisas não ocorrem desta forma, e esta seção irá apresentar alguns dos detalhes que envolvem a função `main` da linguagem C.

```
1  .section .data
2  A: .int 0
3  B: .int 0
4  .section .text
5  .globl _start
6  troca:
7  pushl %ebp
8  movl %esp, %ebp
9  subl $4, %esp
10 movl 8(%ebp), %eax
11 movl (%eax), %ebx
12 movl %ebx, -4(%ebp)
13 movl 12(%ebp), %eax
14 movl (%eax), %ebx
15 movl 8(%ebp), %eax
16 movl %ebx, (%eax)
17 movl -4(%ebp), %ebx
18 movl 12(%ebp), %eax
19 movl %ebx, (%eax)
20 addl $4, %esp
21 pop %ebp
22 ret
23 _start:
24 movl $1, A
25 movl $2, B
26 pushl $B
27 pushl $A
28 call troca
29 addl $8, %esp
30 movl $0, %ebx
31 movl $1, %eax
32 int $0x80
```

Algoritmo 24: Tradução do algoritmo 23.

Para começar, a função `main` é uma função que tem dois argumentos: um inteiro (`argc`) e um vetor de endereços (`argv`). O parâmetro `argc` indica quantos argumentos foram incluídos na linha de comando que chamou o programa. Como exemplo, considere o comando `programa arg1 arg2 arg3`. Quando este programa entrar em execução, `argc` será empilhado com o valor 4 (pois são quatro palavras digitadas), e os valores de `argv` serão os seguintes: `argv[0]=programa`, `argv[1]=arg1`, `argv[2]=arg2`, `argv[3]=arg3`.

Conforme o modelo de execução, estes parâmetros estão na pilha e podem ser acessados a partir de `%ebp`, e assim o é. Porém, isto leva a duas outras perguntas:

P Quem foi que colocou estes valores na pilha?

R o sistema operacional, mais especificamente o carregador de programas. Ele é quem organiza o espaço virtual de execução de programas, e entre várias outras tarefas, coloca os argumentos da linha de comando na pilha antes de "liberar" o programa para execução. É importante destacar que em linux, o carregador faz isto para TODOS os programas, independente da linguagem em que foi desenvolvido (a linguagem C tem o mecanismo descrito acima para acessar estas informações, e outras linguagens tem outros mecanismos).

P Onde ficam e como são organizados estes parâmetros?

R Ficam na pilha, e para acessar `argc` a partir de um programa assembly basta usar 8(%ebp), como fizemos em todos os demais procedimentos

Para exemplificar, considere o algoritmo 25. Vamos executá-lo no simulador ald, onde podemos analisar cuidadosamente alguns aspectos de sua execução.

```
1  .section .text
2  .globl _start
3  pushl %ebp
4  movl %esp, %ebp
5  movl 4(%ebp), %eax
6  movl 8(%ebp), %ebx
7  movl 12(%ebp), %ebx
8  movl 16(%ebp), %ebx
9  movl $1, %eax
10 int $0x80
```

Algoritmo 25: Programa que acessa os parâmetros da função main

O programa basicamente armazena o valor de `argc` em `%eax` e depois armazena os parâmetros `argv[0]` (`8(%ebp)`), `argv[1]` (`12(%ebp)`) e `argv[2]` (`16(%ebp)`) em `%ebx`.

Para colocar argumentos no programa, após entrar no simulador, digite `set args XXX YY ZZZ` e load <nome do programa>. Ao longo da execução, verifique os valores que são colocados em `%ebx` e examine aquelas posições de memória (comando `examine`). Em minha máquina, os valores são, na ordem: `0xBFFFF976`,

`0xBFFFF97F` e `0xBFFFF983`. Estes endereços podem variar de máquina para máquina. Quando executei o programa no simulador, e examinei o conteúdo de `argv[0]`, obtive o seguinte resultado:

```
ald> examine 0xBFFFF976
Dumping 64 bytes of memory starting at 0xBFFFF976 in hex
BFFFF976:  69 6D 70 72 41 72 67 73 00 58 58 58 00 59 59 59    imprArgs.XXX.YYY
BFFFF986:  00 5A 5A 5A 00 4D 41 4E 50 41 54 48 3D 3A 2F 75    .ZZZ.MANPATH=:/u
```

O endereço inicial a ser impresso é 0xBFFFF976 (`argv[0]`). Este endereço contém o primeiro byte do nome do meu programa (imprArgs) seguido de um byte zero, que indica fim desta cadeia de caracteres. O endereço seguinte é `0xBFFFF97F`(`argv[1]`), que corresponde ao primeiro byte do primeiro argumento (XXX), e assim por diante. Observe que todas as cadeias de caracteres indicadas por `argv` terminam em zero.

Um fato curioso no *dump* de memória acima é o que vem depois do último argumento do programa: `MANPATH=`. . .. Esta é uma variável de ambiente, e todas as variáveis de ambiente são listadas em seqüência após o último parâmetro (por exemplo, `HOME`, `PWD`, etc.. Com este mecanismo, qualquer programa, quando em execução, contém uma "fotografia" do valor das variáveis de ambiente tinham quando ele foi executado. Para encontrar o valor de uma variável de ambiente, basta procurar pelo nome dela a partir fim da lista de argumentos (neste caso, `ZZZ`). Os programadores C podem obter o valor destas variáveis de ambiente através da função `getenv`.

4.2.6 Chamadas Recursivas

Para finalizar o estudo sobre o funcionamento de chamadas de procedimento, esta seção apresenta um exemplo de um programa recursivo. A idéia é fixar todos os conceitos apresentados sobre chamadas de procedimento, em especial a forma com que os registros de ativação são empilhados para um mesmo procedimento (ou seja, como é a implementação de várias folhas de papel sobrepostas que correspondem a um mesmo procedimento).

O algoritmo 26 contém todos estes elementos. Ele calcula o fatorial de um número (no caso, de 4). Existem maneiras muito mais elegantes e muito mais eficientes para implementar um programa que calcula o fatorial, porém este programa foi implementado desta maneira (até certo ponto confusa) para incluir parâmetros passados por referência, parâmetros passados por valor e variáveis locais (observe que a variável "r" é desnecessária).

A tradução do algoritmo 26 é apresentada no algoritmo 27.

Os aspectos importantes a serem destacados neste algoritmo são os seguintes:

- a forma de inserir o endereço de *x* na pilha (linhas 44 a 46).
- a forma de acessar o valor de x (através do endereço que está na pilha (*x): linhas 27 e 28.
- os passos para a chamada recursiva (linhas 21 a 25).

```
1  void fat ( int* res, int n)
2  {
3     int r;
4     if (n<=1)
5        *res=1;
6     else {
7        fat (res, n-1);
8        r = *res * n;
9        *res=r;
10    }
11 }
12 main (int argc, char** argv)
13 {
14    int x;
15    fat (&x, 4);
16    return (x);
17 }
```

Algoritmo 26: Programa recursivo.

A figura 4.2.6 detalha o conteúdo de um registro de ativação. Observe que `%ebp` aponta para o endereço de memória que contém o valor de `%ebp` do registro de ativação anterior. Nesta figura, fica mais fácil de perceber qual o endereço relativo a `%ebp` dos parâmetros e das variáveis locais.

A figura 4.2.6 mostra uma série de registros de ativação empilhados, como seria na execução do programa recursivo tratado nesta seção. Cada registro de ativação está representado com uma cor diferente. Da esquerda para a direita, a figura mostra como os registros de ativação são colocados na pilha a cada chamada de procedimento. A configuração mais à equerda é obtida no procedimento "main", a configuração à direita corresponde à chamada de fat(4), e à esquerda desta, correspode a fat(3), e assim por diante até fat(1).

O objetivo desta figura é ressaltar os valores de `%ebp` ao longo do tempo. Ele sempre aponta para o registro de ativação corrente, no endereço exato onde está guardado o registro de ativação do procedimento anterior a este. As setas indicam a lista encadeada de valores de `%ebp`.

A figura mostra que o registrador `%ebp` aponta para último registro de ativação colocado na pilha. O endereço apontado por ele é o local onde o valor anterior de `%ebp` foi salvo (veja figura 4.2.6). O valor antigo de `%ebp` aponta para outro registro de ativação (imediatamente acima). Este, por sua vez, aponta para o registro de ativação anterior, e assim por diante. O último registro de ativação corresponde ao procedimento `main` conforme destacado na figura.

Como exercício, indique:

1. quais instruções assembly são responsáveis pelo empilhamento e pelo desempilhamento de cada campo do registro de ativação.

```
1  .section .data
2  .section .text
3  .globl _start
4  fat:
5      pushl %ebp
6      movl %esp, %ebp
7      subl $4, %esp
8      movl 12(%ebp), %eax
9      movl $1, %ebx
10     cmpl %eax, %ebx
11     jle else
12     movl 8(%ebp), %eax
13     movl $1, (%eax)
14     jmp fim_if
15 else:
16     movl 12(%ebp), %eax
17     subl $1, %eax
18     pushl %eax
19     pushl 8(%ebp)
20     call fat
21     addl $8, %esp
22     movl 8(%ebp), %eax
23     movl (%eax), %eax
24     movl 12(%ebp), %ebx
25     imul %ebx, %eax
26     movl %eax, -4(%ebp)
27     movl -4(%ebp), %eax
28     movl 8(%ebp), %ebx
29     movl %eax, (%ebx)
30 fim_if:
31     addl $4, %esp
32     popl %ebp
33     ret
34 _start:
35     pushl %ebp
36     movl %esp, %ebp
37     subl $4, %esp
38     pushl $4
39     movl %ebp, %eax
40     subl $4, %eax
41     pushl %eax
42     call fat
43     addl $8, %esp
44     movl -4(%ebp), %ebx
45     movl $1, %eax
46     int $0x80
```

Algoritmo 27: Tradução do Algoritmo 26

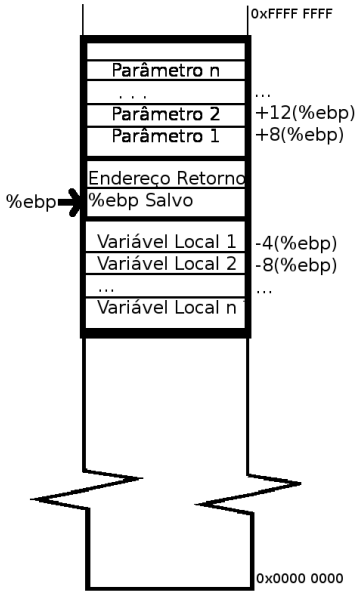

Figura 4.3: Registro de ativação com indicação dos campos.

2. preencha os valores das variáveis do programa em “C” nos campos relacionados com as variáveis na figura 4.2.6.

4.2.7 Uso de Bibliotecas

Os programas apresentados até o momento ou não imprime os resultados, ou os coloca em uma variável de ambiente. Agora descreveremos como trabalhar com funções desenvolvidas externamente, mais especificamente com as funções disponíveis na libc, em especial as funções printf e scanf.

Para utilizar estas duas funções, são necessários dois cuidados:

1. Dentro do programa assembly, empilhar os procedimentos como descrito neste capítulo.

2. Ao ligar o programa, é necessário incluir a própria libc. Como faremos a ligação dinâmica, é necessário incluir a biblioteca que contém o ligador dinâmico.

Para exemplificar o processo, considere o algoritmo 28, e sua tradução, o programa29.

Como pode ser observado, a tradução é literal. Empilha-se os parâmetros e em seguida há a chamada para as funções printf e scanf.

```
> as psca.s -o psca.o
> ld psca.o -o psca -lc -dynamic-linker /lib/ld-linux.so.2
> ./psca
```

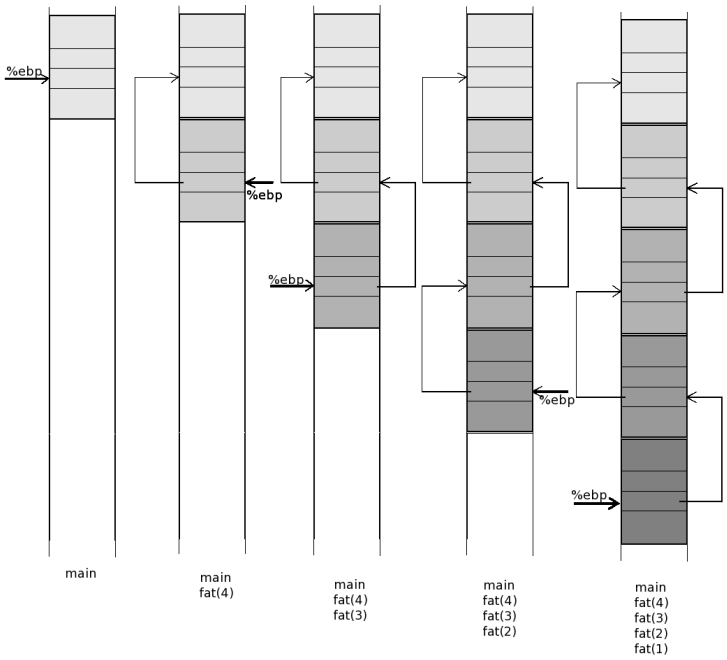

Figura 4.4: Registros de ativação do procedimento “fat” ao longo das chamadas recursivas.

```
1 main (int argc, char** argv)
2 {
3    int x, y;
4    scanf ("Digite dois numeros %d %d ", &x, &y);
5    printf("Os numeros digitados foram %d %d \n ", x, y);
6 }
```

Algoritmo 28: Uso de printf e scanf.

```
1  .section .data
2  str1: .string "Digite dois números: "
3  str2: .string "%d %d"
4  str3: .string "Os numeros digitados foram %d %d \n"
5  .section .text
6  .globl _start
7  _start:
8      pushl %ebp
9      movl %esp, %ebp
10     subl $8, %esp
11     pushl $str1
12     call printf
13     addl $4, %esp
14     movl %ebp, %eax
15     subl $8, %eax
16     pushl %eax
17     movl %ebp, %eax
18     subl $4, %eax
19     pushl %eax
20     pushl $str2
21     call scanf
22     addl $12, %esp
23     pushl -8(%ebp)
24     pushl -4(%ebp)
25     pushl $str3
26     call printf
27     addl $12, %esp
28     movl $1, %eax
29     int $0x80
```

Algoritmo 29: Tradução do Algoritmo 28

```
> Digite dois números: 1 2
> Os numeros digitados foram 1 2
```

As funções `printf` e `scanf` não estão implementadas aqui, e sim na biblioteca libc, disponível em /usr/lib/libc.a (versão estática) e /usr/lib/libc.so (versão dinâmica). Como é mais conveniente utilizar a biblioteca dinâmica, usaremos esta para gerar o programa executável.

A opção `-dynamic-linker /lib/ld-linux.so.2` indica qual o ligador dinâmico que será o responsável por carregar a biblioteca `libc`, indicada em `-lc`, em tempo de execução.

Capítulo 5

A Seção BSS

Suponha um programa que lê uma matriz de 1024 linhas e 1024 colunas de inteiros. Em tempo de execução, este programa utilizará $1024 \times 1024 = 4194304$, 4Mbytes de memória só para conter esta matriz.

Se os dados forem alocados como uma variável global, isto significa também que o arquivo executável teria, no mínimo, 4Mbytes, ocupando inutilmente muito espaço em disco (afinal, estes 4Mbytes só são úteis em tempo de execução).

Como disco é normalmente um recurso escasso, e formas de diminuir o espaço usado em disco (neste caso diminuir o tamanho do arquivo executável) são sempre bem vindas. O formato ELF (assim como quase todos os formatos de execução existentes) foi projetado para esta economia através da seção bss. A idéia básica é que o arquivo executável contenha, na seção .bss, uma diretiva dizendo que, quando o programa for colocado em execução, o carregador deve alocar 4Mbytes de dados para aquela matriz. Esta diretiva ocupa poucos bytes e com isso reduz drasticamente o espaço em disco ocupado pelos arquivos executáveis. Já em tempo de execução não há nenhuma mudança (somente o local da memória virtual que a memória é alocada).

Para exemplificar o funcionamento desta diretiva e a forma de utilizar a bss, o algoritmo 30 apresenta um programa C que lê um arquivo colocado no primeiro argumento e o copia para o arquivo indicado no segundo argumento. A tradução deste algoritmo é mais longa, e foi dividida em duas partes: 31 e 32.

O efeito da compilação e da execução deste programa está apresentado abaixo.

```
> gcc copia.c -o copia
> ./copia SB.ps temp
> diff SB.ps temp
>
```

Dois aspectos devem ser observados:

1. Este programa utiliza a entrada e saída não formatadas (`read` e `write`) para a cópia. Estas operações são mapeadas diretamente para chamadas de sistema com o mesmo nome. É importante destacar que o procedimento printf utiliza esta mesma chamada de sistema write após formatar os dados para impressão.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #define TAM_BUFFER 256
7  char buffer[TAM_BUFFER];
8  main ( int argc, char** argv )
9  {
10    int fd_read, fd_write;
11    int bytes_lidos, bytes_escritos;
12    printf("Copiando %s em %s \n ", argv[1], argv[2]);
13    fd_read = open (argv[1], O_RDONLY);
14    if (fd_read < 0 )
15    {
16       printf ("Erro ao abrir arquivo %s \n ", argv[1]);
17       return -1;
18    }
19    fd_write = open (argv[2], O_CREAT | O_WRONLY );
20    if (fd_write < 0 )
21    {
22       printf ("Erro ao abrir arquivo %s \n ", argv[2]);
23       return -1;
24    }
25    bytes_lidos=read(fd_read, buffer, TAM_BUFFER);
26    while (bytes_lidos > 0 )
27    {
28       bytes_escritos = write (fd_write, buffer, bytes_lidos);
29       bytes_lidos = read(fd_read, buffer, TAM_BUFFER);
30    }
31    return 1;
32 }
```

Algoritmo 30: Programa copia.c

2. Como alocar o vetor `buffer`. Se este vetor fosse declarado como variável global, o arquivo executável deveria alocar `TAM_BUFFER` bytes para armazenar o vetor (lembre-se que as variáveis globais ocupam espaço físico dentro do arquivo executável). Para economizar espaço, é possível declarar este vetor dentro da área bss e dizendo quantos bytes devem ser alocados para ele quando ele for colocado em execução.

O programa assembly equivalente (algoritmos 31 e 32) apresenta uma série de novidades:

1. Constantes: A diretiva `equ` indica uma macro, onde o primeiro conjunto de caracteres deve ser substituído pelo argumento que vem depois da vírgula. Como exemplo, todos os lugares que existe o nome `SYSCALL` serão substituídos por `0x80` para efeito de montagem. A idéia é que o programa se torne mais "legível".

2. A seção .bss contém uma macro (`TAM_BUFFER`) equivalente ao programa em C e em seguida o comando `.lcomm BUFFER, TAM_BUFFER` que indica que **em tempo de execução**, deverão ser alocados 256 bytes ao rótulo BUFFER. Dentro do arquivo executável existe somente uma referência à necessidade de criação de 256 bytes no topo da seção bss (veja com `objdump -t`).

5.1 Heap

O nome *heap* está associado a alocação dinâmica. É neste local em que é alocado espaço através das funções `malloc` e seus assemelhados. Porém, o modelo de execução não tem uma seção heap. Este espaço é alocado imediatamente acima da área da bss.

Devido a esta proximidade, os dois nomes são freqüentemente usados indistintamente, apesar de serem conceitualmente diferentes: a área da bss é estática (em nosso exemplo, um bloco de 256 bytes foi alocado no início da execução, e não será liberado até o fim da execução do programa). Por outro lado, a heap pode crescer e decrescer durante a execução do programa.

Considere mais uma vez o modelo de execução de um programa. As áreas de texto, dados e bss são estáticas (não mudam de tamanho ao longo da execução), enquanto que a pilha pode crescer ou decrescer. A área da pilha em que um programa pode acessar é definida pelo registrador `%esp`, ou seja, o programa tem acesso à área definida entre o topo da memória virtual até o endereço contido em `%esp`. Tentar acessar o endereço `-12(%esp)` ocasiona um erro de execução com simpática mensagem "segmentation fault"[1].

O topo da heap não é armazenado em registrador, mas sim em uma estrutura de dados interna do Sistema Operacional (por vezes chamada de `brk`, e somente ele pode

[1]Teoricamente, é isso mesmo, mas nem sempre é assim na prática, pois o custo para implementar este teste normalmente é "caro". Por isso, a maioria dos Sistemas Operacionais implementam esta regra com mecanismos mais simples, o que pode permitir a violação em alguns casos. Esta é uma das razões para que alguns programas funcionem (ou deixem de funcionar) ao acrescentar ou retirar uma variável (ou mesmo um byte) à região de dados.

```
1  .section .data
2
3  # Constantes
4  .equ CLOSE_SERVICE, 6
5  .equ OPEN_SERVICE, 5
6  .equ WRITE_SERVICE, 4
7  .equ READ_SERVICE, 3
8  .equ EXIT_SERVICE, 1
9  .equ SYSCALL, 0x80
10
11 .equ O_RDONLY, 00
12 .equ O_CREAT, 0100
13 .equ O_WRONLY, 01
14
15 str1: .string "Copiando %s em %s \n "
16 str2: .string "Erro ao abrir arquivo %s \n "
17
18 .section .bss
19 .equ TAM_BUFFER, 256
20 .lcomm BUFFER, TAM_BUFFER
21
22 .section .text
23 .globl _start
24 _start:
25    pushl %ebp
26    movl %esp, %ebp
27    subl $16, %esp
28    pushl 16(%ebp)
29    pushl 12(%ebp)
30    pushl $str1
31    call printf
32    addl $12, %esp
33    movl $OPEN_SERVICE, %eax
34    movl 12(%ebp), %ebx
35    movl $O_RDONLY, %ecx
36    movl $0666, %edx
37    int $SYSCALL
38    movl %eax, -4(%ebp)
39    cmpl $0, %eax
40    jge abre_argv2
41    pushl 16(%ebp)
42    pushl $str2
43    call printf
44    addl $8, %esp
45    movl $-1, %ebx
46    jmp fim_pgma
```

Algoritmo 31: Primeira Parte da Tradução do Algoritmo 30

```
47 abre_argv2:
   movl $OPEN_SERVICE, %eax
   movl 16(%ebp), %ebx
   movl $O_CREAT, %ecx
   orl $O_WRONLY, %ecx
   movl $0666, %edx
   int $SYSCALL
   movl %eax, -8(%ebp)
   cmpl $0, %eax
   jge primeira_leitura
   pushl 12(%ebp)
   pushl $str2
   call printf
   addl $8, %esp
   jmp fim_pgma
primeira_leitura:
   movl $READ_SERVICE, %eax
   movl -4(%ebp), %ebx
   movl $BUFFER, %ecx
   movl $TAM_BUFFER, %edx
   int $SYSCALL
   movl %eax, -12(%ebp)
while:
   cmpl $0, -12(%ebp)
   jle fim_while
   movl $WRITE_SERVICE, %eax
   movl -8(%ebp), %ebx
   movl $BUFFER, %ecx
   movl -12(%ebp), %edx
   int $SYSCALL
   movl %eax, -16(%ebp)
   movl $READ_SERVICE, %eax
   movl -4(%ebp), %ebx
   movl $BUFFER, %ecx
   movl $TAM_BUFFER, %edx
   int $SYSCALL
   movl %eax, -12(%ebp)
   jmp while
fim_while:
   movl $CLOSE_SERVICE, %eax
   movl -4(%ebp), %ebx
   int $SYSCALL
   movl $CLOSE_SERVICE, %eax
   movl -8(%ebp), %ebx
   int $SYSCALL
   movl $1, %ebx
fim_pgma:
   movl $EXIT_SERVICE, %eax
   int $SYSCALL
```

Algoritmo 32: Segunda Parte da Tradução do Algoritmo 30

alterar o valor. No início da execução do programa, o topo da heap é definido como o topo da bss (ou seja, está vazio). Quando o programa deseja alocar alguns bytes dinamicamente, ele basicamente precisa solicitar que o SO altere o valor da brk.

Como a única forma de se comunicar com o Sistema Operacional é através de chamadas de sistema, o programa pode fazer uso do serviço número 45, indicando o novo endereço da brk em `%ebx`. O endereço do novo valor do topo da heap é retornado em `%eax`. Se o valor em `%ebx` for zero, somente é retornado o endereço atual da brk.

É importante destacar que as funções de C que fazem uso da heap (malloc, free, etc.) são gerenciadores deste espaço, e por isso nem sempre um malloc termina com uma chamada ao sistema. Por exemplo, quando o programa solicita 100 bytes, o malloc pode (e deve) alocar mais do que isso (por exemplo, 1024 bytes ou uma página de memória). A segunda chamada a malloc não precisará usar a chamada ao sistema se o espaço alocado anteriormente não foi totalmente utilizado.

De forma análoga, um `free` nem sempre libera o espaço da heap (neste caso, o espaço é apagado logicamente através de um indicador de espaço livre). Por isso, se um programa fizer seguidamente `malloc`s, a heap cresce, mas os `free`s nem sempre fazem a heap decrescer.

Capítulo 6

Chamadas de Sistema

Como já foi destacado na seção anterior, os programas em execução que quiserem fazer uso de recursos externos ao programa (por exemplo, acesso a arquivos, tela, etc) devem solicitar que o sistema operacional intermedie esta ação. A solicitação se dá a partir de chamadas ao sistema[1] e alguns exemplos destas solicitações já foram apresentados.

Um dos motivos para a existência das chamadas de sistema é segurança: permitir que um programa acesse diretamente qualquer dispositivo pode ser perigoso. Como exemplo, considere um programa que ou por falta de conhecimento ou por má intenção do programador, repousa o cabeçote em um disco rígido. Este ato pode danificar o disco de forma irrecuperável.

Devido a este tipo de possibilidade, sistemas operacionais conscientes não permitem que um processo acesse diretamente qualquer dispositivo externo.

Para conseguir tal acesso, é necessária uma solicitação para que o sistema operacional faça o acesso por ele. Desta forma, o sistema operacional pode detectar se a requisição irá comprometer a integridade física do dispositivo (ou da execução de outro processo, violação da privacidade, etc), e com isso impedir acesso indevidos. Esta segurança tem um custo, pois o tempo entre fazer a solicitação e ela ser atendida é normalmente mais lento do que executar a ação dentro do próprio programa[2]orém, acreditamos que este preço não é alto, e certamente todos aqueles que já tiveram dispositivos "queimados" devido a sistemas operacionais que optam por desempenho em detrimento à segurança concordam com nosso ponto de vista :o)..

As chamadas de sistema em linux seguem o padrão POSIX[3], que define, entre muitas outras coisas, as API[4] dos serviços que o sistema operacional oferece para cada processo em execução. Este padrão foi definido por um grupo de trabalho da IEEE e se mantém estável desde o seu lançamento em 1985.

A forma de solicitar um serviço já foi apresentada anteriormente e pode ser resumida em dois passos:

[1] System calls ou syscalls
[2] P
[3] *Portable Operating System Interface*
[4] Application Programming Interface

1. indica os parâmetros em registradores (%eax recebe o número do serviço e os demais registradores recebem os parâmetros para aquele serviço);

2. instrução `int 0x80`.

A instrução `int 0x80` faz a mudança de modo execução usuário para modo de execução supervisor (ou sistema operacional). No modo de execução usuário o programa não tem acesso a determinadas regiões de sua própria memória virtual por questões de segurança. Já um processo no modo supervisor pode acessar toda a memória virtual e física, e inclusive interferir na execução de outros processos. O sistema operacional executa no modo supervisor, enquanto que todos os demais processos executam no modo usuário. Assim, a instrução `int 0x80` ajusta o modo de execução e começa a executar o serviço solicitado dentro do sistema operacional. O parâmetro `0x80` indica o endereço da memória (chamado vetor de interrupção) que contém o endereço do trecho do SO que é capaz de tratar interrupções. Assim que o serviço for completado, o SO devolve a execução para o processo que requisitou o serviço.

Um aspecto importante a ser levantado é que os valores que constam na CPU não são alterados quando de uma chamada de sistema, ao contrário do que ocorre quando da chamada de uma subrotina (por exemplo `printf` ou `scanf`). As primeiras instruções executadas imediatamente após a instrução `int 0x80` (ou seja, o trecho de código no sistema operacional) salvam (por exemplo empilhando) todos os valores que constam nos registradores. Assim que a execução da chamada de sistema é finalizada, os valores salvos são restaurados. Assim, ao longo da execução da chamada de sistema, os registradores podem ser alterados sem causar efeito no programa que fez a chamada de sistema.

Os serviços disponibilizados pelo sistema operacional linux seguem o padrão POSIX, sendo que o número do serviço é indicado no valor do registrador `%eax`. Até o momento descrevemos os seguintes serviços:

1 finalizar programa (exit);

2 fechar Arquivo (close);

3 ler dados de arquivo (read);

4 escrever dados em arquivo (write);

5 abrir arquivo (open);

45 ajustar altura da bss (brk)

A tabela 6 descreve alguns serviços, seus parâmetros e funcionalidade.

A título de curiosidade, descreveremos abaixo a função de algumas outras chamadas de sistema. Para maiores detalhes, veja livros especializados em sistemas operacionais linux.

fork Cria um novo processo. O novo processo herda várias características do processo criador, como descritores de arquivo, etc.). Após a execução desta chamada de sistema, dois processos idênticos estarão em execução, e exatamente no mesmo

%eax	Nome	%ebx	%ecx	%edx	Comentário
1	exit	retorno			Finaliza o programa
3	read	descritor de arquivo	início do buffer	tam. buffer	lê para o buffer indicado
4	write	descritor de arquivo	início do buffer	tam. buffer	escreve o buffer para o descritor de arquivo.
6	close	descritor de arquivo	início do buffer	tam. buffer	fecha o descritor de arquivos indicado
45	brk	Novo valor da brk (Se zero, retorna valor atual de brk em %eax)			Altera a altura da brk, o que aumenta ou diminui a área da heap.

Tabela 6.1: Exemplos de chamadas de sistema (syscalls).

ponto do programa pai. A diferença entre eles é que no processo pai, o retorno da chamada fork (em `eax`) é o número do processo filho, enquanto que para o retorno do processo filho, o retorno é zero.

exec substitui o processo corrente pelo processo indicado nos argumentos. Para criar um novo processo, é necessário primeiro fazer um fork e um exec no processo filho.

dup duplica os descritores de arquivo.

Várias chamadas de sistema têm funções associadas na linguagem C chamadas *wrapper funcions*. Exemplos incluem `write`, `read`, `open`, `close`, `fork`, `exec`, entre outras. O funcionamento destas funções é simples: coloca-se os parâmetros da função em registradores e executa a chamada de sistema associada. Todas estas funções estão documentadas nas man pages.

Um exemplo clássico do uso das chamadas `fork` e `exec` em programas é apresentado em livros de sistemas operacionais como modelo de shell script (veja [Tan01]).

Existe um comando para visualizar quais as chamadas de sistema que um programa executa, o comando `"strace"`. Abaixo o resultado (resumido) do que se obtém ao analisar o comando `ls` aplicado em um diretório vazio.

```
> ls
> strace ls
execve("/bin/ls", ["ls"], [/* 43 vars */]) = 0
brk(0)                                  = 0x805c000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or ...
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1 ...
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or  ...
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=111501, ...}) = 0
mmap2(NULL, 111501, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f63000
close(3)                                = 0
(...)
```

A primeira chamada ao sistema que é executada é `execve`. Esta chamada ao sistema é uma variação de `exec`, que basicamente indica qual programa deve ser executado com que parâmetros. Em seguida, consulta a altura da brk (`brk(0)`), e assim por diante. A saída completa tem mais de 100 linhas).

Escreva um programa com alocação dinâmica (sugestão: pilha ou fila simples) o execute com `strace`. Veja quantas vezes você aloca dados dinamicamente e confira se o número de chamadas a `brk` é igual. Se não for (o que é esperado), explique.

Parte II

Software Básicos

Este texto considera como “software básico” aquele que converte arquivos associados à execução de programas para outro tipo de arquivo, cujo conteúdo é mais próximo ao modelo de execução.

O capítulo 7 descreve sucintamente alguns destes programas. Cada programa recebe um arquivo como entrada e gera um arquivo de saída. Os formatos dos arquivos de entrada e de saída são descritos no capítulo 8.

Dos programas indicados no capítulo 8, dois se destacam, e são tratados em seções próprias: o programa ligador (capítulo 8.4) e o programa carregador (capítulo 8.5).

Capítulo 7

Software Básicos

Entende-se como software básicos aqueles que convertem arquivos associados à execução de programas de um formato para outro mais próximo ao formato executável compreendido pela máquina. Os formatos e software estão descritos esquematicamente na figura 7.1. Observe que um programa em linguagem de "alto nível" é visto como um programa executável pelo programador, porém para que a máquina consiga colocá-lo em execução, é necessário converter da linguagem entendida pelo programador (linguagem de alto nível) para linguagem de máquina (linguagem de baixo nível). Quem faz esta conversão são o compilador e o ligador.

Uma breve descrição das ferramentas é apresentada a seguir:

Compiladores: Compiladores são programas que convertem arquivos que contém programas escritos em linguagens de alto nível (formato texto) para formato objeto. Exemplos de linguagens de alto nível incluem "C", "Java", "Fortran", "Pascal", entre outros.

Montadores: Montadores são programas que convertem arquivos que contém programas em linguagem assembly (formato texto) para o formato de código objeto.

Ligadores: são programas convertem arquivos em formato objeto para formato executável.

Carregadores : Carregam os arquivos que contém formato executável para a execução. Para isto ele lê partes dos arquivos objeto e copia estas partes para a memória.

O restante desta seção detalha o funcionamento, a forma de se utilizar e a funcionalidade destas ferramentas.

Compilador, montador e ligador são programas que podem ser invocados diretamente pelo programador.

Como exemplo destes programas e do efeito que eles produzem, digite e salve o algoritmo 33 abaixo em um arquivo, por exemplo, `prog1.c`.

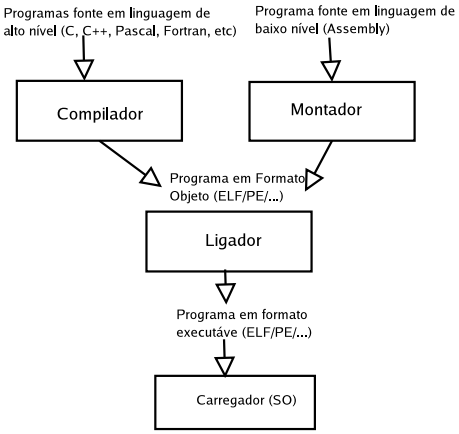

Figura 7.1: Formatos de arquivos “executáveis” e os software de conversão para o formato executável

```
1 int a, b;
2 main ( int argc, char** argv);
3 {
4 return 13;
5 }
```

Algoritmo 33: Arquivo prog1.c

Em seguida, compile o programa para obter o arquivo objeto (prog.o). Por fim, use o arquivo objeto para gerar o executável como descrito abaixo.

```
> gcc -c prog1.c -o prog1.o
> gcc prog1.o -o prog1
> ./prog1
> echo \$?
13
```

Os arquivos `prog1.c`, `prog1.o` e `prog1` são respectivamente os programas fonte, objeto e executável sendo que cada um está armazenado em um formato específico.

O arquivo `prog1.c` (programa fonte) segue o formato especificado da linguagem C.

O arquivo `prog1.o` e o arquivo `prog1` estão em um formato que não usa somente caracteres ASCII, e por isso é ilegível para a maior parte da raça humana. Porém, existem programas que imprimem o conteúdo deste tipo de arquivo para uma linguagem que utiliza caracteres ASCII (porém não por isso são mais legíveis). Um exemplo é o programa `objdump`, que pode ser usado em arquivos ELF objeto e executáveis. Exemplo:

```
> gcc -c prog1.c -o prog1.o
> objdump -s prog1.o

prog1.o:     file format elf32-i386

Contents of section .text:
 0000 8d4c2404 83e4f0ff 71fc5589 e551c705  .L$.....q.U..Q..
 0010 00000000 06000000 c7050000 00000700  ................
 0020 00008b15 00000000 a1000000 008d0402  ................
 0030 a3000000 00a10000 0000595d 8d61fcc3  ..........Y].a..
Contents of section .comment:
 0000 00474343 3a202847 4e552920 342e312e  .GCC: (GNU) 4.1.
 0010 31203230 30373031 30352028 52656420  1 20070105 (Red
 0020 48617420 342e312e 312d3531 2900      Hat 4.1.1-51).
```

Como foi dito antes, a impressão não é exatamente o que se pode chamar de "legível".

Uma outro aspecto importante é que o formato do arquivo objeto e executável é definido pelo Sistema Operacional. No linux, este formato é o ELF (Executable and Linking Format), que foi originariamente desenvolvido e publicado pelo UNIX System Laboratories (USL) como parte da *Application Binary Interface (ABI)*.

Já para o Sistema Operacional Windows foram projetados vários modelos de execução. Como exemplos, temos o formato COM, EXE e mais recente o PE (*Portable Executable File Format*.

Como quem define o formato do arquivo executável é o Sistema Operacional, programas executáveis gerados em um sistema operacional não pode ser executado diretamente em outro sistema operacional (mesmo que ambas as máquinas usem o mesmo processador). Quando tal se faz necessário, é comum usar interpretadores ou emuladores do modelo executável em questão. Uma forma bastante esperta de se fazer isso é ler

o o arquivo em formato executável "A" e colocam em execução no formato executável "B". É assim que o `wine`[1] trabalha.

Este texto irá detalhar alguns aspectos do formato de arquivos ELF. Sugerimos ao leitor mais curioso que digite `objdump -s prog1` para ver as seções em que o arquivo `prog1` está dividido (section .init, .plt, .text, .fini). Destas, destacamos a seção .text, que contém os comandos em linguagem de máquina para o referido programa.

Um carregador é uma parte do Sistema Operacional que é capaz de ler um arquivo executável e colocá-lo na memória para execução. Basicamente ele executa as seguintes tarefas:

1. identifica as seções de um arquivo executável;
2. solicita uma nova entrada na tabela de processos do SO;
3. aloca espaço na memória para que o programa seja lá colocado;
4. termina sua ação.

Ao final desta seqüência, o Sistema Operacional terá um novo processo na tabela de processos, e o escalonará para execução de acordo com as regras de escalonamento de processos.

É comum encontrar dois tipos de carregadores: os carregadores absolutos e os relocáveis. Os carregadores absolutos alocam sempre a mesma porção de memória para um mesmo processo.

Isso parece estranho, e um exemplo pode ajudar. Todos os programas executáveis tem comandos assembly. Um dos comandos assembly que permitem o desvio do fluxo de execução é o comando do tipo `jump ENDEREÇO`. Quando se usa carregadores absolutos, o compilador precisa assumir que o programa começa em um determinado endereço da memória física e assim o ENDEREÇO contido no `jump` corresponde a um endereço físico de memória conhecido ANTES de o programa ser colocado em execução. Com isso, o programa tem que ser colocado sempre no mesmo local da memória física para funcionar corretamente. Se esta região de memória estiver sendo ocupada, o Sistema Operacional adota uma das seguintes medidas:

1. retira o outro processo da memória (copia o outro processo para uma área em disco, chamada "memória swap") e assim libera a área desejada para o novo processo;
2. impede a execução do novo processo.

Como é de se imaginar, a primeira solução é a mais comum.

O segundo tipo de carregador é o **carregador relocável**. Neste caso, o programa pode ser colocado em qualquer local da memória para execução. O programa executável relocável é semelhante ao programa executável absoluto, exceto que os endereços de memória que ele referencia são ajustados para o endereço de memória correto quando ele é colocado para execução (na verdade, este ajuste pode ocorrer tanto em

[1] http://www.winehq.com/

tempo de carregamento ou em tempo de execução). Como estes endereços são ajustáveis, recebem o nome de endereços relocáveis. Esta solução é tão comum que muitos processadores incluem facilidades para determinar os endereços relativos, facilitando a relocação em tempo de execução.

Os programas DOS com extensão ".exe" são exemplos de programas executáveis relocáveis, enquanto que os que tem extensão ".com" usam relocador absoluto.

Apesar de mais flexíveis, os carregadores relocáveis também apresentam problemas:

- Os programas a serem executados não podem ser maiores do que a memória disponível. Ou seja, se o computador tiver 128M de memória RAM, ele não poderá executar programas que ocupem 129M.
- Se a memória estiver ocupada por outros programas, (ou seja, há memória mas não está livre), estes programas terão de ser descarregados antes da execução do novo programa.

Várias abordagens foram tentadas para minimizar os problemas de memória citados acima, e as soluções propostas nem sempre são convincentes[2].

É importante destacar que um dos maiores (senão o maior) problema para executar um processo[3], é alocar espaço para ele na memória física. Cada sistema operacional tem seus próprios mecanismos mas estes mecanismos não serão detalhados aqui, pois são um tema longo normalmente abordado em livros de Sistemas Operacionais.

Primeira Solução jogar o problema para o usuário: exigir que o computador tenha muita memória. Apesar de parecer pouco viável comercialmente, esta solução era muito comum, e para isto basta veja a "configuração" mínima exigida por certos programas[4]. Apesar de simples, ela pode comprometer as vendas dos produtos, principalmente se o programa precisar de uma grande quantidade de memória. Não é necessário dizer que esta solução não deixa os usuários felizes.

Segunda Solução jogar o problema para os programadores. Aqui, várias opções foram tentadas, mas a que é mais viável divide o programa em várias partes logicamente relacionadas. Como exemplo, considere os programas de cadastro que normalmente começam com uma tela que solicita que o usuário escolha por um opção entre *inclusão*, *alteração* e *exclusão*. As três opções são, normalmente, implementados em trechos diferentes de código, e não precisam estar simultaneamente na memória. Neste tipo de aplicação, o programador indica quais são os três módulos e que eles devem ser carregados de forma excludente. Esta solução também está longe de ser a ideal, mas era encontrada freqüentemente em

[2]Como aconteceu em uma estória de Asterix onde o druida Amnesix "curou" uma pessoa que pensava ser um Javali. Ele o ensinou a ficar de pezinho, e assim notava-se menos o problema (veja historia completa em Asterix e o Combate dos Chefes).

[3]Na taxonomia de sistemas operacionais, um processo é um programa pronto para execução ou em execução.

[4]Isto não quer dizer que todos os programas que apresentam uma "configuração mínima" recaem neste caso. Por vezes, ela indica o tamanho mínimo de memória necessário para que partes básicas (aquelas que tem de ficar na memória ao longo de toda a execução do programa).

programas da década de 70-80, quando os computadores tinham pouco espaço de memória (por exemplo 64K de memória RAM). Esta abordagem remonta ao conceito de **overlays**, assunto de disciplinas de Sistemas Operacionais e foi adotado em vários Sistemas Operacionais.

Esta idéia é semelhante ao modelo de bibliotecas carregadas dinamicamente, onde o programador cria um programa principal (que pode ficar o tempo todo na memória) e as bibliotecas (subrotinas chamadas somente quando necessário). Em tempo de execução, o programa indica quais bibliotecas devem ser carregadas na memória, e o carregador (por vezes chamado de carregador-ligador) traz a biblioteca requisitada para a memória. Após o uso, o programador pode dizer ao Sistema Operacional que aquela biblioteca pode ser descarregada da memória.

Terceira Solução assumir que o sistema operacional é que é o responsável pelo gerenciamento da memória, esta abordagem é a adotada pelos sistemas operacionais baseados em unix. O sistema operacional gerencia a memória por meio de um mecanismo chamado “paginação”.

A idéia básica da paginação é que um programa não acessa diretamente os endereços reais (físicos - na memória), mas sim endereços virtuais. Trechos dos programas em memória virtual são então copiados para a memória real, e, cada vez que um endereço é acessado, o SO primeiro faz a conversão de endereços para descobrir a qual endereço físico corresponde cada endereço virtual. Este processo é lento, e por isso muitas CPUs já incorporaram este mecanismo para ganhar desempenho.

Mais detalhes sobre o funcionamento deste mecanismo fogem ao escopo deste livro. Para mais informações, consulte livros de sistemas operacionais ou de arquitetura de computadores.

Capítulo 8

Formatos de Programa

Um programa de computador pode ser visto de várias formas diferentes de acordo com o software básico que pode ser utilizado sobre ele. Estes formatos são: linguagem de alto nível (C, C++, Pascal, Fortran, Cobol), linguagem assembly e linguagem de máquina (objeto e executável). Descrevemos cada uma destas a seguir.

8.1 Linguagem de alto nível

As linguagens de alto nível são as mais compreensíveis para os seres humanos, porém para a máquina são somente um conjunto de caracteres. Programas escritos em linguagens de alto nível podem ser transformados em linguagens mais próximas à linguagem que o computador compreende através do compilador.

A vantagem de escrever programas neste formato é que o programador não precisa conhecer as peculiaridades da arquitetura em que este programa será executado como, por exemplo, o conjunto de registradores, conjunto de instruções, interrupções, etc. Existem duas desvantagens em usar o compilador:

1. o compilador normalmente inclui uma série de procedimentos que muitas vezes não são utilizados;

2. o compilador por vezes pode gerar um código mais lento do que seria se fosse gerado em linguagem de máquina (não devemos ficar chateados com o compilador por isso, ele deve ser conservador por natureza, ou seja, ele deve gerar um código que funciona, não obrigatoriamente o código mais rápido que funciona).

Também é importante destacar que estas duas desvantagens são na verdade um preço muito barato a se pagar quando os programas são grandes (e em muitos casos também é barato para programas pequenos).

A seguir, apresentamos um exemplo de um programa em linguagem de alto nível e as diretivas de compilação que o transformam em um programa em linguagem executável. Para ilustrar melhor algumas peculiaridades envolvidas com o processo, apresentamos um programa composto de dois arquivos descritos em 34 e 35.

```
1 void a (char* s);
2 int main (int argc, char** argv)
3 {
4    static char str[] = "Hello World\n";
5    a (str);
6 }
```

Algoritmo 34: Programa exemplo: arquivo “main.c”

```
1 #include <stdio.h>
2 void a (char* s)
3 {
4    printf("%s", s);
5 }
```

Algoritmo 35: Programa exemplo: arquivo “a.c”

Para compilar os dois arquivos em um arquivo executável, usamos os seguintes comandos:

```
> gcc -o a.o -c a.c
> gcc -o main.o -c main.c
> gcc -o main main.o a.o
> ./main
Hello World
>
```

As duas primeiras linhas geram o código objeto (extensão *.o) que corresponde a cada arquivo com extensão “.c”. A terceira linha agrupa os dois arquivos objeto em um único arquivo executável.

Existem métodos alternativos:

1. Gerando um único arquivo objeto.

   ```
   > gcc -o a.o -c a.c
   > gcc -o main main.c a.o
   ```

2. Não gerando explicitamente nenhum arquivo objeto.

   ```
   > gcc -o main main.c a.c
   ```

Observe que em nenhum caso foi utilizado o ligador. O processo de compilação procura simplificar o trabalho do programador, e por isso o programa “gcc” chama automaticamente o ligador (programa “ld”) para que o programa executável seja gerado a partir do programa objeto.

Um aspecto importante é que nem o arquivo “main.o” e nem o arquivo “a.o” podem ser executados. Somente o arquivo “main” é que pode ser executado, uma vez que está no formato que é compreendido pelo programa carregador.

Os dois tipos de arquivo (objeto e executável) estão no formato “ELF” (estamos falando sobre Linux, especificamente) e são arquivos binários. Cada arquivo é dividido em seções, e as diferenças entre eles podem ser vistas através de um programa que transforma o arquivo binário em um arquivo texto com os comandos em assembly (chamado “disassembly”). Um exemplo deste tipo de programa para Linux é o “objdump”. Como exercício, digite os programas acima e em seguida:

```
> objdump -S a.o
> objdump -S main
```

O opção “-S” faz com que o código executável seja impresso em formato texto, uma instrução por linha.

8.2 Linguagem assembly

Os primeiros programas eram escrito em código de máquina, e os programadores da época deviam indicar explicitamente quais bits da memória deveriam estar “apagados” e quais deveriam estar “ligados”. Este processo era muito propenso a erros, e por isso, foi projetada a linguagem assembly. Como regra geral, cada instrução em assembly representa exatamente uma instrução em linguagem de máquina (a que é executada diretamente pela CPU). O programa que traduz o arquivo contendo instruções em assembly para linguagem de máquina é chamado de montador, que é um programa bastante simples pois a correspondência entre instruções em assembly e linguagem de máquina é quase sempre de um para um.

Um programa assembly é um conjunto de caracteres ASCII, porém cada linha corresponde a uma instrução em linguagem de máquina. Como cada arquitetura tem o seu próprio conjunto de instruções (definido no processador utilizado), para converter um programa em assembly que roda no processador X para o processador Y é necessário reescrevê-lo totalmente.

Freqüentemente os próprios fabricantes de CPU vendem também os programas montadores para as suas CPUs. Existem excessões a esta regra, mas neste caso pelo menos é fornecido um manual que explica o mapeamento para que terceiros possam escrever montadores.

Em função da complexidade e detalhamento envolvido na programação de computadores via linguagens assembly, elas começaram a cair em desuso quando surgiram as primeiras linguagem de alto nível, onde uma linha de código pode corresponder a dezenas de linhas de programas assembly.

Quando isso ocorreu, programas que eram antes escritos por um complexo código de milhares (ou milhões) de linhas passaram a ser escritos por algumas centenas de linhas de código em linguagens de alto nível, que além de serem mais fáceis de ser projetados, são também mais fáceis de serem depurados e de se dar manutenção.

É importante destacar que as linguagens de alto nível são convertidas para assembly (ou diretamente para linguagem de máquina) para que possam ser executadas em uma máquina específica. Neste caso, os programas em linguagem assembly (ou em linguagem de máquina) são equivalentes (ou seja, causam o mesmo efeito) às linguagens de programação em linguagem de alto nível

O código do algoritmo 36 é um exemplo de programa assembly.

```
1 .section .data
2 .section .text
3 .globl _start
4 _start:
5     movl $1, %eax
6     movl $13, %ebx
7     int $0x80
```

Algoritmo 36: Exemplo de programa assembly (`exit.s`).

Uma diferença importante entre programas escritos em linguagem assembly e programas escritos em linguagens de alto nível é que os primeiros lidam diretamente com os objetos de armazenamento do computador (registradores e memória). O conceito de variável é um conceito abstrato, e está associado com um local de armazenamento, mas sem explicitar qual (por exemplo, se em memória ou em registrador).

Até certo ponto, podemos considerar que programas assembly lidam com "objetos concretos", enquanto que linguagens de alto nível lidam com "objetos abstratos". Linguagens como as orientadas a objeto lidam com objetos ainda mais abstratos, como objetos, classes, métodos, entre outros.

Em linguagens de alto nível, o conceito de variável é abstrato, mas inclui outros aspectos, como por exemplo, um conjunto de atributos, como tipo, tamanho, entre outros. Como cada variável é de um tipo específico, o compilador se recusa a gerar código executável se o programa fonte misturar tipos de dados (por exemplo, se tentar somar um inteiro com um caractere). Em assembly, isto é possível, por exemplo somando um inteiro de oito bits (um caractere) com um inteiro de 32 bits (um inteiro com ou sem sinal). Além disso, em assembly se utilizam instruções onde o tipo está implícito: somar dois valores assumindo que os dois são inteiro com ou sem sinal. O programador não se preocupa com isso em linguagens de alto nível.

Voltemos agora a diferenças mais concretas entre linguagens de alto e de baixo nível. O processo de geração de programa executável a partir de um programa em assembly está descrito abaixo:

```
> as -o exit.o exit.s
> ld -o exit exit.o
> ./exit
>
```

Compare este processo com a figura 7.1). A primeira linha gera um arquivo objeto `exit.o` a partir do arquivo de entrada `exit.s` usando o programa montador (`as` neste caso). A segunda linha gera o arquivo executável `exit` a partir do arquivo objeto `exit.o` através do ligador (`ld` neste caso). Por fim, a terceira linha executa o programa. Este programa não gera nenhum resultado na tela como pode ser visto.

8.3 Linguagem de Máquina

A linguagem de máquina pode ser encontrada tanto em arquivos executáveis, abordado na seção 8.3.2.1, quanto em arquivos objeto, abordado na seção 8.3.2.1.

8.3.1 Linguagem de Máquina - Arquivo executável

O arquivo executável em linguagem de máquina contém instruções em uma linguagem que a máquina compreende, mas que é incompreensível para os seres humanos (bom, pelo menos para a grande maioria). O formato do arquivo executável (ou seja, como ele é organizado, quais as informações que ele contém, e como estão organizadas) é definido pelo sistema operacional (e não pela CPU, como muitos podem imaginar). Um programa (o carregador) é capaz de ler um arquivo neste formato e colocá-lo em execução.

Os arquivos que contém programas neste formato estão representados em binário (normalmente “compactado” na representação em hexadecimal). Um exemplo de arquivo binário (em hexadecimal), que corresponde ao programa "exit", montado acima, é mostrado a seguir:

```
> hexdump exit -C -v
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  74 80 04 08 34 00 00 00  |........t...4...|
00000020  ac 00 00 00 00 00 00 00  34 00 20 00 02 00 28 00  |........4. ...(.|
00000030  07 00 04 00 01 00 00 00  00 00 00 00 00 80 04 08  |................|
00000040  00 80 04 08 80 00 00 00  80 00 00 00 05 00 00 00  |................|
00000050  00 10 00 00 01 00 00 00  80 00 00 00 80 90 04 08  |................|
00000060  80 90 04 08 00 00 00 00  00 00 00 00 06 00 00 00  |................|
00000070  00 10 00 00 b8 01 00 00  00 bb 03 00 00 cd 80     |...............|
00000080  00 2e 73 79 6d 74 61 62  00 2e 73 74 72 74 61 62  |..symtab..strtab|
00000090  00 2e 73 68 73 74 72 74  61 62 00 2e 74 65 78 74  |..shstrtab..text|
000000a0  00 2e 64 61 74 61 00 2e  62 73 73 00 00 00 00 00  |..data..bss.....|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000d0  00 00 00 00 1b 00 00 00  01 00 00 00 06 00 00 00  |................|
000000e0  74 80 04 08 74 00 00 00  0c 00 00 00 00 00 00 00  |t...t...........|
000000f0  00 00 00 00 04 00 00 00  00 00 00 00 21 00 00 00  |............!...|
00000100  01 00 00 00 03 00 00 00  80 90 04 08 80 00 00 00  |................|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 04 00 00 00  |................|
00000120  00 00 00 00 27 00 00 00  08 00 00 00 03 00 00 00  |....'...........|
00000130  80 90 04 08 80 00 00 00  00 00 00 00 00 00 00 00  |................|
00000140  00 00 00 00 04 00 00 00  00 00 00 00 11 00 00 00  |................|
00000150  03 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  |................|
00000160  2c 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |,...............|
00000170  00 00 00 00 01 00 00 00  02 00 00 00 00 00 00 00  |................|
00000180  00 00 00 00 c4 01 00 00  b0 00 00 00 06 00 00 00  |................|
00000190  07 00 00 00 04 00 00 00  10 00 00 00 09 00 00 00  |................|
000001a0  03 00 00 00 00 00 00 00  00 00 00 00 74 02 00 00  |............t...|
000001b0  20 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  | ...............|
000001c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001d0  00 00 00 00 00 00 00 00  74 80 04 08 00 00 00 00  |........t.......|
000001e0  03 00 01 00 00 00 00 00  80 90 04 08 00 00 00 00  |................|
000001f0  03 00 02 00 00 00 00 00  80 90 04 08 00 00 00 00  |................|
00000200  03 00 03 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000210  03 00 04 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000220  03 00 05 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000230  03 00 06 00 01 00 00 00  74 80 04 08 00 00 00 00  |........t.......|
00000240  10 00 01 00 08 00 00 00  80 90 04 08 00 00 00 00  |................|
00000250  10 00 f1 ff 14 00 00 00  80 90 04 08 00 00 00 00  |................|
00000260  10 00 f1 ff 1b 00 00 00  80 90 04 08 00 00 00 00  |................|
00000270  10 00 f1 ff 00 5f 73 74  61 72 74 00 5f 5f 62 73  |.....start.__bs|
```

```
00000280  73 5f 73 74 61 72 74 00  5f 65 64 61 74 61 00 5f  |s_start._edata._|
00000290  65 6e 64 00                                       |end.|
00000294
```

A primeira coluna indica os endereços, a segunda indica o conteúdo de cada endereço em formato hexadecimal enquanto que a última coluna indica a representação ASCII para cada conjunto de oito bits. Exemplo: endereço 00000001 = (45)hexa = (E) em ASCII.

O arquivo executável apresentado está no formato ELF (Executable and Linking Format), um formato desenvolvido pela Sun. Devido à sua flexibilidade e documentação livre, este formato foi adotado em vários (se não em todos) sistemas operacionais baseados em Unix. Existem dois tipos de arquivos ELF: os que armazenam código objeto (incluindo bibliotecas) e os que armazenam código executável (que é o caso do exemplo acima).

Para executar um programa ELF, o usuário digita somente o nome do arquivo, e o sistema operacional ativará o carregador, um programa interno que lê este arquivo binário e executa uma série de passo para colocá-lo em execução. Para matar a curiosidade sobre o formato, veja [Wheb] (que apresenta bem mais do que só o formato ELF). Um arquivo executável no formato ELF está dividido em seções (.symtab, .strtab, .shstrtab, .text, .data, .bss, etc.). Uma destas seções, a seção .text, contém as instruções em linguagem de máquina que são compreendidos pela CPU.

Muitas vezes há interesse em ver os mnemônicos ASCII para cada uma das instruções em assembly, e por isso foram criados programas que "desmontam" (disassembly) os arquivos executáveis. Usaremos aqui o programa "objdump". Considere como exemplo o programa "exit", que contém somente três instruções.

```
> objdump -S exit

exit:     file format elf32-i386

Disassembly of section .text:

08048054 <_start>:
 8048054: b8 01 00 00 00        mov    $0x1,%eax
 8048059: bb 0d 00 00 00        mov    $0xd,%ebx
 804805e: cd 80                 int    $0x80
```

Esta é somente uma representação mais amigável do programa executável. A visualização mais fiel é aquela apresentada anteriormente. Cada instrução assembly (normalmente) corresponde a uma instrução em linguagem de máquina. Neste caso, a representação em linguagem de máquina da instrução `mov $0x1,%eax` é o padrão hexadecimal `b8 01 00 00 00`. Para ver mais sobre formatos de instrução assembly, modos de endereçamento dos parâmetros, etc. veja o apêndice A.1.

Através do programa objdump, também é possível selecionar partes dos arquivos binários para serem mostrados "in natura". Um exemplo é mostrar o código binário da seção .text:

```
> objdump -s exit

exit:     file format elf32-i386
```

```
Contents of section .text:
 8048054 b8010000 00bb0d00 0000cd80          ............
```

É possível identificar a instrução `mov $0x1,%eax` codificada no padrão `b801000000`.

8.3.2 Linguagem de Máquina - Arquivo objeto

O formato ELF também comporta uma representação para arquivos objeto. Um exemplo de arquivo objeto é o arquivo "a.o" apresentado acima, gerado a partir do arquivo-fonte "a.c". Basta olhar para o arquivo a.c para perceber que ele não é passível de ser executado diretamente, uma vez que não apresenta a função "main".

É interessante observar que o arquivo "main.o" contém a função "main", porém não é possível criar um arquivo executável a partir dele pois não está incluída a implementação da função "a".

Assim, os arquivos em linguagem de máquina no formato objeto são incompletos, ou seja, falta-lhes algo para serem executados. Este algo pode ser a função "main" ou a definição de alguma variável ou função externa.

Observe que pode-se gerar um programa executável usando vários arquivos objeto através do ligador.

No exemplo acima, os arquivos "main.c" e "a.c", geraram dois arquivos objeto: "main.o" e "a.o", que foram ligados e geraram um único arquivo executável. Isto foi possível porque os arquivos se completam: o arquivo "main.o" contém a função "main" enquanto que o arquivo "a.o" contém a implementação da função "a".

O uso de arquivos objeto agiliza em muito o processo de desenvolvimento de software. Se um software for composto de vários módulos, cada módulo composto de várias funções, é razoável dividir o programa em vários arquivos menores e só juntá-los para gerar o arquivo executável. Se houver necessidade de modificar um destes arquivos, por exemplo para corrigir um erro (bug), então não há necessidade de alterar os demais arquivos.

Como exemplo, suponha que um programa tenha sido dividido em cinco arquivos: a.c, b.c, c.c, d.c, e main.c. O arquivo main.c utiliza as funções implementadas nos outros arquivos, e o arquivo executável "main" é obtido através dos seguintes comandos:

```
> gcc -o a.o -c a.c
> gcc -o b.o -c b.c
> gcc -o d.o -c c.c
> gcc -o d.o -c d.c
> gcc -o main main.c a.o b.o c.o d.o
```

Se na fase de testes for detectado que há um erro em alguma das funções implementadas no arquivo "c.c", então o programador deve somente alterar este arquivo e para testar se o erro foi corrigido, deve executar os seguintes comandos:

```
> gcc -o c.o -c c.c
> gcc -o main main.c a.o b.o c.o d.o
```

Observe que não foi necessário recompilar os arquivos objeto a.o, b.o e d.o, pois estes não sofreram modificações. Uma idéia interessante seria a de agrupar os arquivos objeto em um único arquivo, chamado “biblioteca”.

8.3.2.1 Bibliotecas

O uso de arquivos objeto, reduz o tempo e a complexidade de construção de programas, especialmente se as funções contidas em arquivos objeto já tiverem sido testadas exaustivamente. Neste caso, as funções em arquivos-objeto são normalmente mais confiáveis do que projetar estas funções, além de ser muito mais rápido.

Como exemplo, suponha que alguém foi encarregado de desenvolver um programa onde é necessário calcular a raiz quadrada de um número. Se um programador souber que esta função está presente em algum arquivo objeto, ele ganhará tempo utilizando esta implementação ao invés de implementá-la.

Como isso já foi observado há muito tempo, já foram desenvolvidos trechos de programa para várias operações, e foram gerados arquivos objeto para elas. Porém, por uma questão de organização, é interessante agrupar arquivos objeto em um único arquivo para somente então incluí-los nos programas. Este arquivo que contém uma série de arquivos objeto é chamado de biblioteca. A presente seção detalha os três tipos de bibliotecas disponíveis, que se diferenciam principalmente nos seguintes aspectos:

1. na forma com que são geradas;
2. na forma com que são invocadas em tempo de execução;
3. no espaço físico que ocupam no arquivo executável e na memória.

As próximas seções descrevem cada um dos tipos de bibliotecas, destacando em especial os aspectos supracitados. A seção 8.3.2.2 descreve as bibliotecas estáticas, a seção 8.3.2.3 descreve as bibliotecas compartilhadas e a seção 8.3.2.4 descreve as bibliotecas dinâmicas.

8.3.2.2 Bibliotecas Estáticas

Uma biblioteca estática é um conjunto de arquivos objeto agrupados em um único arquivo. Normalmente se utiliza a extensão ".a" para indicar uma biblioteca estática, e em Linux elas são criadas através do programa "ar" (archive).

```
> ar -rc libMyLib.a a.o b.o c.o d.o
> gcc -o main -lMylib -L.
```

O programa "ar" simplesmente concatena os arquivos indicados e coloca alguma informação adicional (para facilitar o acesso às funções) no cabeçalho do arquivo "libMyLib.a".

A linha de compilação para gerar um aquivo executável agora é um pouco diferente:

-lMyLib : diz ao compilador que o código das funções que não forem encontradas em main.c podem ser encontradas na biblioteca libMyLib.a (observe que são omitidos a extensão (.a) e os caracteres iniciais, que sempre tem que ser "lib".

-L. : Quando houver necessidade, o compilador (mais especificamente o ligador) irá procurar pela biblioteca em alguns diretórios padrão (/usr/lib, /usr/local/lib, etc), porém não irá procurar no diretório corrente. A opção -L" avisa ao ligador para procurar pela biblioteca também no diretório indicado, que no caso é ".", ou seja, o diretório corrente.

Este mecanismo é largamente utilizado e existem mais de 100 bibliotecas (extensão ".a") em /usr/lib. Cada biblioteca contém uma série de funções relacionadas. Exemplos incluem `libm.a` (biblioteca de funções matemáticas), `libX.a` (X window), `libjpeg.a` (compressão de imagens para formato jpeg), `libgtk.a` (interface gráfica), etc.

Cada biblioteca pode ser composta de vários arquivos objeto, e cada arquivo objeto implementa dezenas ou centenas de funções, e por vezes torna-se difícil lembrar de todas as funções e da ordem dos parâmetros em cada uma delas. Por isso, para cada arquivo biblioteca está relacionado um arquivo cabeçalho (header) e este arquivo deve ser incluído para que o compilador possa verificar se os parâmetros correspondem aos parâmetros da implementação da função.

Por vezes é difícil lembrar qual o arquivo cabeçalho correto e qual a biblioteca que deve ser incluída para que um programa possa usar uma determinada função. Por isso, as *man pages* relacionam os nomes das funções, ao arquivo cabeçalho que deve ser incluído e à biblioteca desejada.

Tomemos como exemplo a função `sqrt`. Quando eu digito "man sqrt", aparecem as seguintes informações:

```
SQRT(3)            Linux Programmer's Manual       SQRT(3)

NAME
       sqrt, sqrtf, sqrtl - square root function

SYNOPSIS
       #include <math.h>

       double sqrt(double x);
       float sqrtf(float x);
       long double sqrtl(long double x);

       Link with -lm.

DESCRIPTION

       The sqrt() function returns the non-negative square root of x.
       It fails and sets errno to EDOM, if x is negative.

       (...)
```

O cabeçalho da função está em `<math.h>` e também é apresentado algumas variações `sqrtf` e `sqrtl`. Com isso, o programador não precisa consultar `<math.h>` para saber quais e de qual o tipo são os parâmetros. Também indica como fazer a ligação (`Link with -lm`) É importante mencionar que muitas vezes não é indicado como fazer a ligação. Nestes casos, a ligação é automática, uma vez que a função em questão faz parte da libc, biblioteca que é usada automaticamente para todos os programas escritos na linguagem "C". Verifique isso consultando as *man pages* das funções `malloc`, `printf`, etc.

Em nosso programa exemplo, a verificação dos parâmetros é feita na primeira linha do programa, onde um arquivo com o protótipo da função é inserido. Como não há código associado a esta função, o compilador entende que é um protótipo e o usa para

checar quais os parâmetros e o valor de retorno da função "a". Para exemplificar a importância desta linha, compile e execute o programa abaixo, sem a linha `#include (...)`. Nestes casos, como a informação não foi passada explicitamente pelo programador, o compilador assume que a função tem um parâmetro inteiro e retorna um inteiro. Apesar disso, o ligador irá incluir normalmente o código da função e vai acreditar que a checagem de tipos foi feita.

Quando o programa for executado, várias coisas podem acontecer, inclusive o programa pode gerar o resultado correto. Porém o mais comum é ocorrer algum tipo de erro de execução. Aqui o erro é identificável porque o compilador avisa (warning) que falta o protótipo e o que ele está fazendo. Esta "permissividade" é uma das deficiências da linguagem C, apesar de muitos programadores experientes entenderem que é isso é uma virtude, pois a torna ótima para “escovar bits”.

```
#include <stdio.h>

void a (char* s) {
  printf ("%s", s);
}
```

8.3.2.3 Bibliotecas Compartilhadas

Apesar de serem simples de usar e simples de criar, as bibliotecas estáticas podem resultar em mau uso da memória. Para entender por que isso ocorre, basta verificar que existem procedimentos (em especial da libc) utilizados em praticamente todos os programas. Um exemplo de uma função com estas características é a "printf". São poucos os programas que não a utilizam, e se todos estes utilizarem bibliotecas estáticas, cada um deles terá uma cópia desta função amarrada em seu código executável. Assim, quando vários programas estão na memória, cada um deles tem uma cópia da função printf. Somando-se todas as cópias, é provável que alguns megabytes de memória são alocados unicamente para executar somente esta função.

As bibliotecas compartilhadas permitem que somente uma cópia de cada biblioteca seja colocada na memória, e permite que todos os programas utlizem aquela cópia como se estivesse dentro do seu próprio espaço de endereçamento (ou seja, como se fosse parte do seu programa executável).

Todos os sistemas operacionais modernos permitem a criação deste tipo de biblioteca, e em algumas versões a própria libc é incluída automaticamente como biblioteca compartilhada.

A seqüência de comandos para gerar uma biblioteca compartilhada para os arquivos objeto "a.c" e "b.c" é a seguinte:

```
> gcc -fPIC -c -Wall a.c -o a.o
> gcc -fPIC -c -Wall b.c -o b.o
> ld -shared -o libMyLib.so a.o b.o
> gcc -o main main.c -lMyLib -L.
```

A opção -fPIC" gera um código objeto relocável, ou seja, um código cujos endereços são definidos na hora de carregar o programa na memória. A opção -Wall" pede para sejam impressos todos os avisos de erro (warnings). A terceira linha é quem

gera a biblioteca compartilhada com o uso da opção -shared". Observe que a biblioteca é gerada pelo ligador. Finalmente, para gerar o programa executável, a linha é igual àquela apresentada para bibliotecas estáticas (seção 8.3.2.2).

Porém, ao executar o programa, recebemos a seguinte mensagem:

```
> ./main
./main: error while loading shared libraries: libMyLib.so: cannot open
shared object file: No such file or directory
```

Isto indica que o carregador não conseguiu encontrar a biblioteca compartilhada "libMyLib.so" (o programa executável não armazena esta informação). Para corrigir este problema, a solução mais simples é incluir o caminho da biblioteca "libMyLib.so" dentro da variável de ambiente "LD_LIBRARY_PATH".

```
> echo $LD_LIBRARY_PATH
/public/soft/Linux/avifile/usr/local/lib
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
> echo $LD_LIBRARY_PATH
/public/soft/Linux/avifile/usr/local/lib:.
> ./main
Hello World
```

A variável `LD_LIBRARY_PATH` indica o diretório do sistema onde as bibliotecas compartilhadas residem. À medida que cada usuário for criando mais bibliocas compartilhadas, o gerenciamento da localização destas bibliotecas pode se tornar difícil. Por isso, sugere-se que cada usuário tenha um diretório "$HOME/usr", com as ramificações convencionais de "/usr" (bin,lib,include,src, entre outras) e coloque ali aquilo que for desenvolvendo. Todas as bibliotecas deveriam ser colocadas no mesmo local, e então basta colocar em seu ".bashrc" a linha

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/usr/lib
```

para que todas as bibliotecas sejam incluídas facilmente. Para o gerenciamento e atualização de versões das bibliotecas, é mais simples utilizar Makefile nos diretórios /usr/src/... convenientes.

Em tempo de execução, o carregador coloca o programa a ser executado na memória e liga o programa a todas as biblitotecas compartilhadas. Todas as biblitotecas deverão estar na memória antes do início da execução do programa.

O texto acima é somente um resumo desenvolvido para que cada um possa desenvolver uma biblioteca compartilhada, porém há muitos outros aspectos importantes, como por exemplo controle de versão. Estes aspectos são apresentados em maiores detalhes na internet[1].

8.3.2.4 Bibliotecas Dinâmicas

Apesar de economizarem memória quando comparadas com as bibliotecas estáticas, pode ocorrer que algumas bibliotecas compartilhadas sejam incluídas no espaço de

[1]Em especial[Whea] Aliás, este link é de leitura obrigatória, ou seja, quando eu elaborar a prova, considerarei que *todos* leram e entenderam o texto. Apesar do texto ser bastante didático, eventuais dúvidas podem ser levantadas em sala ou por e-mail.

memória virtual de um processo, mas que sejam pouco utilizadas ao longo da execução. Isto também acarreta gasto desnecessário de memória.

Em ambientes onde o recurso memória é muito escasso (como por exemplo em uma servidora que executa centenas ou milhares de processos), pode ser interessante exigir que somente as bibliotecas que serão efetivamente usadas sejam carregadas na memória, e que isto ocorra de acordo com as necessidades impostas pelo programador. Por exemplo, se o programa precisar de uma biblioteca no início e no fim da execução do programa, é razoável carregá-la no início, liberá-la durante a execução do programa e talvez re-carregá-la ao final para terminar o programa.

As bibliotecas que se comportam desta forma são chamadas de bibliotecas dinâmicas (Dynamic Link Libraries - DLLs), e para usá-las é necessário um esforço maior do programador, uma vez que o programa principal deve sofrer alterações significativas. Como habitual, vamos exemplificar para uso na linguagem C em Linux.

1. Deve ser incluído o cabeçalho `<dlfcn.h>`, que contém os protótipos das funções dlopen, dlclose e dlsym.

2. A biblioteca deve ser aberta explicitamente através da função dlopen, que irá retornar um handle para esta biblioteca:

```
(...)
void *handle;
char *pathLib="/home/.../libMyDynamicLib.so";
(...)
handle = dlopen ( pathLib, RTLD_LAZY );
(...)
```

3. A função desejada na biblioteca deve ser associada a um apontador para ponteiro da seguinte forma:

```
(...)
void (*localSym)(parametros da funcao na bib);
(...)
localSym = dlsym (handle, "nome da funcao na bib");
```

4. O símbolo `localSym` agora é um sinônimo da função indicada na biblioteca. Para executar a função da biblioteca, basta usar `localSym` com os mesmos parâmetros da função.

```
localSym ( parâmetros ... );
```

5. Ao terminar de usar a biblioteca, deve-se fechá-la com a função `dlclose`:

```
dlclose ( handle );
```

   Suponha que temos dois arquivos fonte (a.c e b.c) com os quais deve ser criada a biblioteca libMyDynamicLib.so. Para criar a biblioteca e para gerar o programa executável, deve ser digitada a seguinte seqüência de comandos:

```
> gcc -fPIC -o a.o -c a.c
> gcc -fPIC -o b.o -c b.c
> ld -shared -o libMyDynamic.so a.o b.o
> gcc -o main main.c -L. -lMyDynamic -ldl
```

Observe que a única diferença com relação ao processo de compilação de bibliotecas compartilhadas é a inclusão da biblioteca -ldl", que contém as funções "dl*".

Este tópico está bem documentado em dois artigos disponíveis na internet, em especial em [Whea, Che] (que devem ser lidos).

```
1 #include <stdio.h>
2
3 void a (char* s) {
4    printf ("(a): %s\n", s);
5 }
```

Algoritmo 37: Arquivo a.c

```
1 #include <stdio.h>
2
3 void b (char* s) {
4    printf ("(b): %s\n", s);
5 }
```

Algoritmo 38: Arquivo b.c

Como exemplo, considere os algoritmos 37, 38, e 39.

O laço contido entre as linhas 13 e 16 espera que o usuário digite a letra `a` ou `b`. As linhas 18 e 19 colocam no vetor `s` o caminho completo (desde a raiz do sistema de arquivos) para o arquivo onde está a biblioteca que iremos incluir dinamicamente. Para tal é utilizada a função getenv (get environment variable), que coloca um ponteiro com o string da variável de ambiente HOME na variável local path. Mais adiante veremos que esta variável de ambiente (na verdade todas as variáveis de ambiente) está inserida no arquivo executável.

As linhas 21 até 26 abrem a biblioteca dinâmica e verificam se houve algum erro (por exemplo, biblioteca não encontrada). Se tudo correr bem, a variável error será `NULL`, ou seja, zero. A opção `RTLD_LAZY` pode ser entendida como indicador para que a biblioteca seja carregada “preguiçosamente”, ou seja, quando alguma das funções contidas nela for utilizada. Observe que a biblioteca é aberta e associada a uma variável (`handle`). Daquele ponto em diante, todas as vezes que for utilizada a variável `handle`, estaremos fazendo referência ao arquivo biblioteca aberto.

As linhas 28 até 31 indicam qual das duas funções contida na biblioteca deve ser utilizada. O comando `funcao = dlsym(handle, "a");` pode ser lida da seguinte forma: Associe a função `a()` contida na biblioteca dinâmica indicada em `handle` à variável `funcao`. Isto se faz basicamente copiando o endereço da função `a` para a variável `funcao`.

```
1  #include <stdlib.h>
2  #include <dlfcn.h>
3
4  int main ( int argc, char** argv ){
5  char opcao;
6  void* handle;
7  void* (*funcao)(char*);
8  char* error;
9  char* nomeBib="/lixo/libMyDynamicLib.so";
10 char* path;
11 char s[100];
12
13    do{
14       printf("Digite (a) para uma bib e (b) para a outra \n");
15       scanf ("%c", &opcao );
16    } while (opcao!='a' && opcao!='b');
17
18    path = getenv ("HOME");
19    sprintf(s, "%s%s", path, nomeBib);
20
21    handle = dlopen(s, RTLD_LAZY);
22    error = dlerror();
23    if ( error ){
24       printf("Erro ao abrir %s", s );
25       exit (1);
26    }
27
28    if ( opcao == 'a' )
29       funcao = dlsym(handle, "a");
30    else
31       funcao = dlsym(handle, "b");
32
33    funcao ("texto\n");
34
35    dlclose (handle);
36 }
```

Algoritmo 39: Arquivo main.c

A execução está na linha 33. Observe que aqui o conteúdo da variável `funcao` é ou o endereço da função `a` ou o endereço da função `b`. Como as duas funções tem a mesma seqüência de parâmetros, não ocorrerá nenhum problema.

Por fim, a linha 35 fecha a biblioteca.

A forma de gerar o programa executável, e o resultado obtido pela execução do programa é a seguinte:

```
> gcc -fPIC -o a.o -c a.c
> gcc -fPIC -o b.o -c b.c
> ld -shared -o libMyDynamicLib.so a.o b.o
> gcc -o main main.c -L. -ldl
> ./main
Digite (a) para uma bib e (b) para a outra
a
(a):  texto
> ./main
Digite (a) para uma bib e (b) para a outra
b
(b):  texto
>
```

Como pode ser visto, a geração da biblioteca (`ld ...`) é igual ao que ocorre em bibliotecas compartilhadas. Porém, há algo de novo na linha `gcc -o main main.c -L. -ldl`. A opção `-ldl`, diz para incluir estaticamente a biblioteca `libdl.a`. Esta biblioteca é que contém as funções `dlopen`, `dlclose`, `dlsym` e `dlerror`. Ela contém muitas outras funções do tipo `dl*`, mas o nosso exemplo só usou estas.

Para maiores detalhes de uso de bibliotecas, consulte[Che, Whea]. A terceira parte do presente texto mostra como estas bibliotecas se comportam em tempo de execução.

8.4 O Programa Ligador

Até o momento, consideramos o uso de programa ligador em várias formas. Agora vamos explicar o que ele faz. Em linhas gerais um programa ligador recebe como entrada uma série de arquivos executáveis e gera um único arquivo como saída. Este arquivo é executável.

Porém, por questões didáticas, vamos dividir o tema. A seção 8.4.1 trata da ligação de arquivos objeto com uma única seção, enquanto que a seção 8.4.2 trata de arquivos objeto com várias seções. A seção 8.4.3 apresenta alguns detalhes que também são relevantes ao processo, e a seção 8.4.4 fecha a seção mostrando como os conceitos apresentados se unem na prática.

Estes primeiros aspectos são direcionados para explicar como o ligador trabalha com arquivos objetos ligados estaticamente. As seções 8.4.5 e 8.4.6 estendem a explicação como o ligador trabalha com arquivos objetos compartilhados e arquivos objeto dinâmicos. Daqui para frente, usaremos os termos objeto estático, compartilhado e dinâmico para referenciar inclusive as bibliotecas deste tipo (que nada mais são do que uma coleção de arquivos objeto).

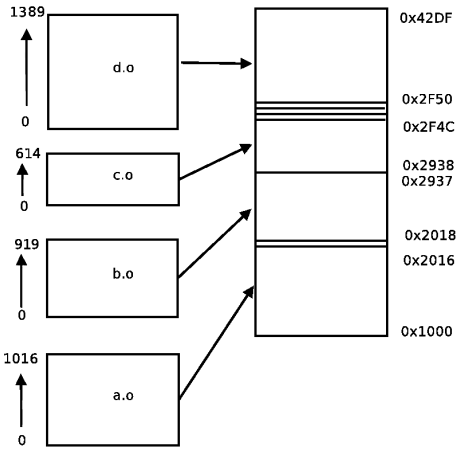

Figura 8.1: Concatenação de arquivos objeto de um único segmento para a composição de um arquivo executável

As seções descrevem o tema são fortemente baseados em [Lev00]. Para maiores detalhes, consulte o livro (que tem uma versão gratuita disponível na internet, fornecida pelo próprio autor).

8.4.1 Arquivos objeto com um único segmento

Considere uma situação simples, onde um conjunto de arquivos objeto `O1`, `O2`, ... `On`, devem ser ligados para gerar uma arquivo executável. Cada arquivo contém um único segmento (seção texto, por exemplo), cujos endereços variam de acordo com o tamanho do arquivo. Se o primeiro arquivo tiver $T1$ bytes, então os endereços deste arquivo variam de 0 até $T1 - 1$. Os endereços do aquivo objeto $O2$ variam de 0 até $T2 - 1$ e assim por diante.

Em nosso modelo simplificado, o arquivo executável `E` seria composto pela concatenação simples dos arquivos O1, O2, ..., On. conforme descrito na figura 8.1.

Vamos agora inserir alguns detalhes a mais neste exemplo. Considere que temos exatamente quatro arquivos objeto: `O1`, `O2`, `O3` e `O4`, cada um deles com o tamanho descrito na tabela 8.1.

Se o arquivo executável iniciar no endereço `0x1000`, então a concatenação dos arquivos será como descrito na tabela 8.2. É importante observar que este exemplo assume um alinhamento de quatro em quatro bytes no arquivo executável. Por isso, os bytes dos endereços 2017 e 2F4D, 2F4E, 2F4F estão vagos.

Arquivo	Tamanho
01	1017
02	615
03	920
04	1390

Tabela 8.1: Arquivos objeto e seus tamanhos

Arquivo	Endereços
01	1000-2016
02	2018-2937
03	2938-2F4C
04	2F50-42DF

Tabela 8.2: Composição do arquivo executável E

Vamos analisar mais cuidadosamente este exemplo, em especial para esclarecer alguns aspectos sobre a relocação de endereços. Para tal, vamos aumentar a quantidade de detalhes do exemplo da seção anterior. Vamos supor que:

1. o arquivo 01 corresponde ao arquivo que contém a função main;
2. cada um dos arquivos objeto restantes tem um único procedimento, digamos P2, P3, P4;
3. o arquivo 01 chama os procedimentos P2, P3, P4.

A figura 8.2 ilustra esta situação. Os arquivos objeto estão representados na coluna da esquerda, enquanto que o arquivo executável é representado à direita. O arquivo objeto 01 contém as chamadas aos procedimentos P1, P2 e P3. Fica claro no exemplo que os endereços dos procedimentos não podem ser atribuídos neste arquivo, e por isso foi anotada somente uma referência aos respectivos nomes.

Os arquivos objeto contém, cada um, a implementação de um destes procedimentos. Este é o quadro antes da geração do arquivo executável. Após passar pelo ligador, o arquivo executável é gerado. Neste arquivo, as referências aos procedimeentos foram substituídas pelos endereços deles dentro do arquivo executável (aqui representados esquematicamente por setas ao invés dos endereços efetivos).

Vamos agora analisar como isto ocorre no mundo real. Uma chamada de procedimento é essencialmente uma instrução `call <endereço>`. Porém, como o arquivo 01 chama três procedimentos cujos endereços não são conhecidos antes de ligá-lo com os outros arquivos objeto, o endereço do procedimento chamado dentro do arquivo 01 fica indeterminado (por exemplo colocando `call 0x00000000` no arquivo objeto). Somente quando os arquivos objeto forem concatenados no arquivo E é que o endereço

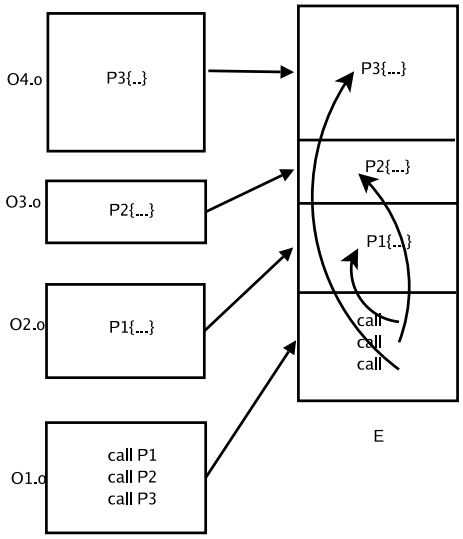

Figura 8.2: Relocação de endereços de procedimentos

dos procedimentos será conhecido, e cada um instrução `call 0x00000000` será substituída por `call <endereço do procedimento em E>`.

Observe que o ligador tem de saber qual o endereço que deve ser colocado em cada uma das várias chamadas, e por isso o arquivo objeto armazena esta informação em uma tabela de endereços relocáveis. Esta tabela pode ser vista através do programa `objdump`:

```
> objdump O1.o -r

O1.o:     file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000012 R_386_PC32        P2
00000017 R_386_PC32        P3
0000001c R_386_PC32        P4
```

A primeira coluna (OFFSET) indica o local dentro do segmento texto que devem ser substituídos. A segunda coluna (TYPE) diz quantos bytes devem ser substituídos. Neste caso, o tipo `R_386_PC32` corresponde a quatro bytes[2]. Finalmente a terceira coluna indica o símbolo (o nome do objeto no programa) cujo endereço no arquivo executável deve ser substituído. Assim, quando o programa executável for construído,

[2] A primeira impressão é que todos os endereços relocáveis seriam de quatro bytes, porém existem outros objetos relocáveis como por exemplo variáveis globais (entre outros) também são relocáveis. Por isso, foram criados identificadores de tipo e tamanho para serem usado em cada caso.

o ligador deve substituir o conteúdo do endereço `0x00000012` do segmento text pelo endereço efetivo do procedimento `P2` dentro de `E` e o mesmo para `P3` e `P4`.

Para completar a explicação, usaremos o programa objdump para mostrar os endereços efetivos gerados pelo nosso exemplo.

Primeiramente, o arquivo `O1`.

```
> objdump O1.o -S

O1.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
(...)
   0: 8d 4c 24 04           lea    0x4(%esp),%ecx
   4: 83 e4 f0              and    $0xfffffff0,%esp
   7: ff 71 fc              pushl  0xfffffffc(%ecx)
   a: 55                    push   %ebp
   b: 89 e5                 mov    %esp,%ebp
   d: 51                    push   %ecx
   e: 83 ec 04              sub    $0x4,%esp
  11: e8 fc ff ff ff        call   12 <main+0x12>
  16: e8 fc ff ff ff        call   17 <main+0x17>
  1b: e8 fc ff ff ff        call   1c <main+0x1c>
  20: 83 c4 04              add    $0x4,%esp
  23: 59                    pop    %ecx
  24: 5d                    pop    %ebp
  25: 8d 61 fc              lea    0xfffffffc(%ecx),%esp
  28: c3                    ret
>
```

A opção "-S" faz o "disassembly", ou seja, lê o segmento text e mostra as instruções assembly que correspondem aos códigos lá inseridos.

Observe as linhas que contêm a instrução `call`. Todas contém o mesmo padrão hexadecimal `0xe8fcffffff`, e correspondem à instrução `call <endereço a ser relocado>`. A instrução assembly associada é `call <deslocamento a partir de main>`, que facilita encontrar o símbolo associado se usarmos o `objdump -r`.

Vamos agora analisar o arquivo executável.

```
> objdump E -S
(...)
 8048354 <main>:
(...)
 8048365: e8 16 00 00 00        call   8048380 <P2>
 804836a: e8 1d 00 00 00        call   804838c <P3>
 804836f: e8 24 00 00 00        call   8048398 <P4>
(...)

08048380 <P2>:
(...)

0804838c <P3>:
```

```
(...)

08048398 <P4>:
(...)

>
```

Como o resultado da operação `objdump E -S` é grande, nos concentramos em mostrar somente a parte que nos interessa para o exemplo em questão.

Observe agora as instruções `call`. Cada uma delas foi substituída pelo endereço efetivo dos procedimentos dentro do arquivo executável. Assim, a chamada `P2()` contida no arquivo `O1`, que era associada à instrução assembly `call 12 <main+0x12>` em $O1.o$ foi agora associada à instrução `call 8048380 <P2>`, ou seja, para o endereço onde o código do procedimento `P2` está mapeado dentro de `E`. Este endereço corresponde ao endereço virtual quando `E` for posto em execução.

Isto nos apresenta outro detalhe interessante. No arquivo objeto, estes endereços são calculados relativo ao endereço de `main`, enquanto que no arquivo executável o endereço é o virtual, que será utilizado ao longo da execução do programa. Verifique isso ao simular um programa, por exemplo com o `ald`.

8.4.2 Arquivos objeto com mais de um segmento

Na seção anterior, apresentamos um exemplo simples, onde vários arquivos objeto tinham somente o segmento texto e eram concatenados para gerar um arquivo executável. Na prática, porém, esta não é uma situação comum. Normalmente os arquivos objeto e os executáveis são compostos de vários segmentos.

O ligador deve concatenar os segmentos equivalentes de cada arquivo objeto em um único segmento no arquivo executável. Um exemplo simples, porém bastante significativo é apresentado na figura 8.3, onde os quatro arquivos objeto tem exatamente dois segmentos. Cada segmento é concatenado a seus pares no arquivo executável. Assim, todos os segmentos texto são concatenados juntos, enquanto que todos os segmentos de dados também o são. As regras de relocação e alinhamento de bytes apresentadas na seção anterior também se aplicam.

8.4.3 Detalhes Importantes

Além do que já foi descrito acima, os ligadores também podem atuar para tornar o programa executável mais ágil em função da arquitetura onde o programa deverá ser executado. É evidente que para tal, eles devem conhecer algo sobre esta arquitetura (por exemplo, CPU, modelo de paginação, tamanho da palavra, etc.). A forma de se explorar estes aspectos está normalmente relacionada com o fato de ser o ligador quem define os endereços virtuais dos segmentos (e de todos os rótulos) que serão usados em tempo de execução.

O primeiro detalhe já foi apresentado, que é o alinhamento de bytes. Em arquiteturas cujo tamanho da palavra é de 32 bits (ou seja, quatro bytes), é razoável iniciar

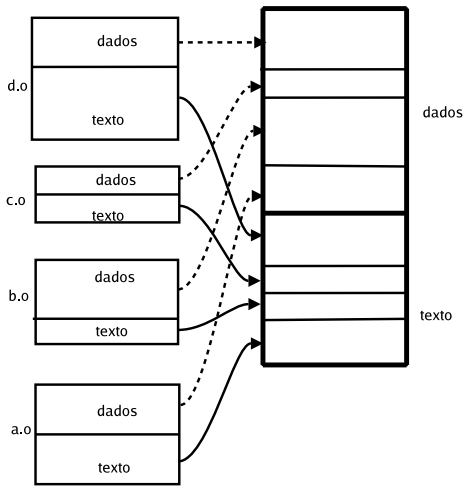

Figura 8.3: Arquivos objeto com mais de uma seção e sua relação com o arquivo executável

um segmento sempre em um endereço múltiplo de quatro bytes. Uma das vantagens é que desta forma é possível carregar todos os quatro primeiros bytes do segmento para a CPU em uma única leitura. Existem outras vantagens, mas estão mais relacionadas com arquitetura de computadores e não iremos descrevê-las aqui.

Outro detalhe é se o sistema operacional utiliza paginação. Neste caso, é razoável atribuir endereços de segmento de tal forma que cada segmento resida em uma página diferente. Por exemplo, se a página é de 4K, então a seção de código pode ser iniciada em um endereço pré-determinado, múltiplo de 4K enquanto que a seção de dados começaria 4K acima deste endereço (neste exemplo, a seção de texto deve ser menor do que 4K). Porém quando o tamanho da seção de texto e de dados somados forem menores ou iguais a 4K, então é razoável colocá-los juntos na mesma página.

8.4.4 Exemplo Prático

As seções anteriores mostraram como funciona o ligador de forma conceitual, e em especial as alterações que ele aplica aos arquivos objeto para gerar um arquivo executável.

Para fechar o assunto sobre o trabalho dos ligadores, vamos detalhar estas alterações usando um exemplo no Linux, com arquivos objeto e executável no formato `ELF`.

Estes exemplos se destinam a analisar em mais detalhes os endereços atribuídos pelo ligador. Para tal, vamos estender os programas apresentados na seção 8.4.1. Usaremos ao todo quatro programas descritos nos algoritmos 40,41, 42, 43. A difereça básica é que os programas desta seção usam quatro variáveis globais, uma declarada em cada um dos quatro arquivos objeto.

Observe também como foram declaradas as variáveis externas ao arquivo `01.c`. Elas foram declaradas com o comando `extern`, que basicamente diz que as variáveis

```
1  #include <stdio.h>
2  int G1;
3  extern int G2, G3, G4;
4
5  int main () {
6     G1=15;
7     P2(); G2=2;
8     P3(); G3=3;
9     P4(); G4=4;
10 }
```

Algoritmo 40: arquivo O1.c

descritas a seguir foram declaradas em outros módulos, e que este modulo não precisa alocar espaço para elas. Se esta diretiva for retirada, o arquivo `O1.o` irá indicar a necessidade de alocar espaço para, por exemplo, `G2`. Porém, o arquivo `O2.o` **também** faz isso. Como resultado, o ligador deve acusar um erro e não será possível gerar o arquivo executável.

```
1  int G2;
2
3  int P2 () {
4     return(1);
5  }
```

Algoritmo 41: arquivo O2.c

Sugerimos que o leitor digite os programas e siga os passos aqui indicados.

```
1  int G3;
2
3  int P3 () {
4     return(2);
5  }
```

Algoritmo 42: arquivo O3.c

O primeiro passo é compilar os quatro programas para gerar os arquivos `O1.o`, `O2.o`, `O3.o`, `O4.o`. Com estes arquivos, gere o arquivo executável `E`.

Reveja a seção 8.4.1 e digite os comandos `objdump` correspondentes. Analise os valores dos endereços. Se quiser, retire as referências às variáveis e compare os resultados com os obtidos na seção 8.4.1.

Execute o `E` usando um simulador. Observe que os endereços que estão indicados no arquivo executável correspondem aos mesmos endereços em tempo de execução. Isto mostra que o ligador é quem define estes endereços.

```
1 int G4;
2
3 int P4 () {
4    return(3);
5 }
```

Algoritmo 43: arquivo O4.c

Agora estamos prontos para analisar as seções geradas. As seções podem ser vistas com o comando `readelf -a O1`. Este programa é equivalente ao `objdump`, porém nos fornece mais informações sobre o arquivo.

O resultado deve ser algo semelhante ao seguinte:

```
> readelf -a O1.o

ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          256 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         10
  Section header string table index: 7

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000051 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 0003a0 000038 08      8   1  4
  [ 3] .data             PROGBITS        00000000 000088 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000088 000000 00  WA  0   0  4
  [ 5] .comment          PROGBITS        00000000 000088 00002d 00      0   0  1
  [ 6] .note.GNU-stack   PROGBITS        00000000 0000b5 000000 00      0   0  1
  [ 7] .shstrtab         STRTAB          00000000 0000b5 000049 00      0   0  1
  [ 8] .symtab           SYMTAB          00000000 000290 0000f0 10      9   7  4
  [ 9] .strtab           STRTAB          00000000 000380 000020 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.
```

```
There are no program headers in this file.

Relocation section '.rel.text' at offset 0x3a0 contains 7 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000013  00000801 R_386_32          00000004   G1
0000001c  00000902 R_386_PC32        00000000   P2
00000022  00000a01 R_386_32          00000000   G2
0000002b  00000b02 R_386_PC32        00000000   P3
00000031  00000c01 R_386_32          00000000   G3
0000003a  00000d02 R_386_PC32        00000000   P4
00000040  00000e01 R_386_32          00000000   G4

There are no unwind sections in this file.

Symbol table '.symtab' contains 15 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS O1.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000000     0 SECTION LOCAL  DEFAULT    5
     7: 00000000    81 FUNC    GLOBAL DEFAULT    1 main
     8: 00000004     4 OBJECT  GLOBAL DEFAULT  COM G1
     9: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND P2
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND G2
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND P3
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND G3
    13: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND P4
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND G4

No version information found in this file.
>
```

O programa `readelf` nos fornece muitas informações interessantes, mas vamos analisar somente uma parte delas.

Primeiramente a divisão em seções. Observe mais atentamente a parte `Sections Headers:`, que enumera todas as seções contidas no arquivo. Talvez muitos esperassem ver somente duas seções (`.text` e `.data`), porém após décadas de experiências, todos os formatos de arquivos objeto e executáveis passaram a acrescentar mais seções, cada uma delas contendo informações específicas. Estas informações nem sempre significam alguma coisa para o programador, porém são extremamente úteis para os programas ligador e carregador.

Vamos agora analisar o tamanho das seções. Observe que a seção `.text` começa no endereço `0x34` do arquivo[3]. Do endereço `0x0` até `0x33` estão contidas informações sobre a organização do arquivo. No endereço `0x34` começam as informações sobre a seção `.text`. Como pode ser visto, a seção `.text` tem o tamanho de `0x51` bytes, o que significa que a seção `.text` está contida entre os bytes `0x34` e `0x84` inclusive (lembre-se que o primeiro byte está no endereço `0x34`).

[3]Considerando que o primeiro byte do arquivo está no endereço `0x0`, o segundo byte está no de endereço `0x1`, e assim por diante.

A próxima seção é `rel.text`, texto relocável. Ela é a segunda a ser descrita, mas está posicionada no endereço `0x3a0` e tem tamanho `0x38`. Ela cotém informações sobre os nomes relocáveis, como variáveis globais e funções. A listagem mostra também uma descrição sobre o conteúdo desta seção. Abaixo, uma cópia daquele trecho.

```
Relocation section '.rel.text' at offset 0x3a0 contains 7 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000013  00000801 R_386_32          00000004   G1
0000001c  00000902 R_386_PC32        00000000   P2
00000022  00000a01 R_386_32          00000000   G2
0000002b  00000b02 R_386_PC32        00000000   P3
00000031  00000c01 R_386_32          00000000   G3
0000003a  00000d02 R_386_PC32        00000000   P4
00000040  00000e01 R_386_32          00000000   G4
```

São ao todo sete entradas: quatro para as variáveis globais e três para funções. A única entrada que tem tamanho especificado é `G1`, que foi declarada neste módulo (`O1.c`). As demais entradas não foram declaradas neste módulo (`Sym.Value=0`).

Em seguida, vem a seção `.data`. Ela deveria começar no byte seguinte ao fim da seção `.text`, ou seja, `0x85`, mas começa no endereço `0x88`. Isto ocorre porque esta seção deve começar em um byte múltiplo de quatro (alinhamento de quatro bytes). Isto significa dizer que não há informação útil nos bytes `0x85`, `0x86` e `0x87`.

Veja que a seção `.bss` está vazia (tem tamanho zero). A seção `.data` também, o que pode parecer estranho, pois deveria conter a informação sobre a variável global G1. Porém como este arquivo está no formato objeto, não é necessário alocar espaço para cada variável global, basta deixar indicado que o arquivo executável deve fazê-lo. Por isso, esta informação foi colocada na seção `.rel.text` (texto relocável, mas que aqui seria melhor se fosse traduzida para símbolos relocáveis). Veja onde são colocadas as variáveis globais e os nomes de procedimentos (também chamados de "símbolos") no arquivo executável. Compare também os tamanhos das seções `.data` nos arquivos objeto e no arquivo executável.

Apesar de existirem muitas outras seções, estas não tem interesse para este livro. Para mais informações, consulte [Com95].

8.4.5 Ligadores para objetos compartilhados

A seção anterior apresentou o modelo básico de funcionamento dos ligadores, onde os arquivos objeto são incluídos no arquivo executável. Este modelo corresponde ao que se denomina "objetos ligados estaticamente". Este modelo foi o primeiro modelo de ligadores que foram criados, porém apresenta dois problemas fundamentais:

1. quando são incluídos muitos arquivos objetos (ou se os arquivos objeto forem grandes), o arquivo executável tende a ser muito grande, gastando muito espaço em disco.

2. em tempo de execução é possível que programas diferentes usem memória física para armazenar informações sobre um mesmo arquivo objeto. Um exemplo clássico é o da função `printf` em um ambiente acadêmico. Normalmente todos os

alunos tendem a usar esta função, e se por exemplo, 100 alunos estiverem trabalhando em uma mesma máquina, então é provável que só o código da função printf ocupe vários megabytes de memória física.

Estes dois problemas motivaram a busca de alternativas para diminuir o tamanho de arquivos executáveis e do tamanho do programas em execução.

Atualmente estão disponíveis duas alterantivas, utilizar objetos compartilhados e utilizar objetos dinâmicos. O primeiro será abordado nesta seção, enquanto que o segundo será abortado na próxima seção (8.4.6.

No caso de objetos compartilhados, o sistema operacional é quem gerencia o processo. O usuário deve criar os auivos objeto compartilhado, e agrupá-los em bibliotecas para incluí-los no arquivo executável. Porém, diferente do que ocorre com os arquivos objetos estáticos, o arquivo objeto não é copiado para o arquivo executável, mas sim o nome do arquivo que contém o objeto.

Como exemplo, considere que os arquivos objeto usados na seção anterior sejam gerados como arquivos objeto compartilhados, e com este seja gerado o arquivo executável:

```
> gcc -fPIC -c  O2-semVars.c -o O2-semVars.o
> gcc -fPIC -c  O3-semVars.c -o O3-semVars.o
> gcc -fPIC -c  O4-semVars.c -o O4-semVars.o
> ld -shared -o libbibComp.so O2-semVars.o  O3-semVars.o  O4-semVars.o
> gcc  O1-semVars.c -o O1-semVars -lbibComp
```

Suponha que ainda não ajustamos a variável de ambiente `LD_LIBRARY_PATH`, e queremos saber quais os arquivos (agrupados em bibliotecas) fazem parte deste arquivo executável. O programa `ldd` traz esta informação. Vamos mostrar o efeito de executar este comando antes de atualizar a variável de ambiente:

```
> ldd -r O1-semVars
undefined symbol: P2 (./O1-semVars)
undefined symbol: P4 (./O1-semVars)
undefined symbol: P3 (./O1-semVars)
Linux-gate.so.1 =>  (0xb7f9e000)
libbibComp.so => not found
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e55000)
/lib/ld-Linux.so.2 (0xb7f9f000)
>
```

Observe os símbolos indefinidos: são as funções referenciadas em O1-semVars. Em seguida, ele apresenta alguns nomes:

Linux-gate.so.1 Esta biblioteca será associada ao endereço `0xb7f9e000` quando o programa for carregado para a execução.

libbibComp.so Esta biblioteca não foi encontrada porque não atualizamos a variável de ambiente.

libc.so.6 Esta biblioteca está em `/lib/tls/i686/cmov/libc.so.6`, e será associada ao endereço `0xb7e55000` quando o programa for carregado para a execução.

/lib/ld-Linux.so.2 É a biblioteca que faz a ligação compartilhada em tempo de execução, e será associada ao endereço `0xb7f9f000` em tempo de execução.

Agora, vamos ver como a referência a `libbibComp.so` se comporta quando mudamos a variável de ambiente:

```
> export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:.
> ldd -r O1-semVars
Linux-gate.so.1 =>  (0xb7f36000)
libbibComp.so => ./libbibComp.so (0xb7f32000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7deb000)
/lib/ld-Linux.so.2 (0xb7f37000)
> ldd -r O1-semVars
Linux-gate.so.1 =>  (0xb7f1e000)
libbibComp.so => ./libbibComp.so (0xb7f1a000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7dd3000)
/lib/ld-Linux.so.2 (0xb7f1f000)
```

Observe agora a referêcia a `libbibComp.so` "magicamente" foi ajustada **sem mexer no arquivo executável**.

Como exercício, faça um `hexdump` no arquivo `O1-semVars`. Verá que o arquivo não contém nenhuma informação de onde procurar a biblioteca `libbibComp.so`. Em tempo de carregamento é que esta biblioteca é procurada nos diretórios indicados em `LD_LIBRARY_PATH`.

Como já foi citado antes, uma das vantagens deste tipo de biblioteca é que gera um arquivo executável menor. Porém, se compararmos os tamanhos dos arquivos executáveis gerado nas seções 8.4.2 e 8.4.5, veremos que a versão compartilhada é maior. Isto ocorreu porque na versão compartilhada foi necessário incluir bibliotecas para gerenciar o funcionamento e carregamento das bibliotecas, enquanto que na versão estática isso não foi necessário. Porém, para bibliotecas grandes, a versão compartilhada vera arquivos **muito** menores do que a estática.

8.4.6 Ligadores para objetos dinâmicos

O objetivo dos objetos dinâmicos é fazer um melhor uso da memória. Por vezes, um programa chama uma biblioteca para executar um pequeno pedaço de código durante sua iniciação, e não é mais usado posteriormente. Porém, se a biblioteca for gerada estaticamente, o código será copiado para dentro do arquivo executável, e ocupará espaço de memória ao longo da execução do programa. Se a biblioteca for gerada de forma compartilhada, então a biblioteca não ocupará espaço no arquivo executável, mas ocupará espaço em tempo de execução.

Vamos considerar o cenário ende programa são grandes. Considerando execução no Linux, assuma 4Gbytes de memória virtual não são suficientes para incluir todo o programa executával. Neste caso, não é possível colocar o programa todo em execução. Porém, pode-se usar um artifício onde se incluem e se retiram bibluiotecas durante a execução do programa.

Existem sistemas operacionais que tem este problema, não usam o modelo de memória virtual, e por isso dependem da quantidade de memória física disponível para executar.

Nestes casos, uma solução interessante é a de colocar o problema na mão do programador. Se o programa dele não cabe em, por exemplo, 4Gbytes, ele pode selecionar quais as bibliotecas que devem ficar disponíveis a cada momento, e com isso ele consegue calcular qual o tamanho máximo de memória usada seguindo as várias possibilidades de alocação das bibliotecas.

Considerando-se os objetos dinâmicos, a tarefa do ligador é simples se comparada com a tarefa do carregador. O ligador deve basicamente incluir uma biblioteca estática que tem os módulos para carregar e liberar bibliotecas em tempo de execução (no caso apresentado, "-ldl" na linha de compilação/ligação para incluir a biblioteca `libdl.a` (veja seção 8.3.2.4). E só.

O resto do trabalho é do programa carregador, e da biblioteca `libdl.a`.

8.5 O Programa Carregador

A função do programa carregador é trazer um programa para a memória e colocá-lo em execução. O carregador é um programa associado ao sistema operacional, e cada sistema operacional trabalha normalmente com um único formato de arquivo executável, apesar de que é possível criar carregadores que podem trabalhar com mais de um formato (como no caso do Windows para os formatos COM, EXE e PE).

Explicado da maneira mais genérica possível, o programa carregador lê um arquivo que está no formato executável, cria um novo processo, aloca espaço na memória para este processo, e copia as informações do arquivo executável para a memória virtual.

Algumas observações importantes:

- O programa não obrigatoriamente iniciará a execução assim que for diponibilizado. Em Linux, por exemplo, "quando um programa é colocado para a execução", ele basicamente é colocado na lista de processos, e o sistema operacional selecionará aquele processo para a execução de acordo com alguma política do escalonador de processos.

- Em princípio, um arquivo executável gerado para operar em uma determinada arquitetura (CPU+SO), não pode ser executado em outra arquitetura ou em outro SO. Porém existem formas de fazer isso através de emuladores, interpretadores ou ainda adaptadores. Estas ferramentas serão analisadas na seçãoẽfsec:interpret.

- É possível gerara código executável para uma arquitetura CPU diferente daquela na qual se está trabalhando.

Existem várias classes de carregadores. Este texto apresentará três classes: carregadores que copiam os programas em memória física sem relocação (seção 8.5.1), carregadores que copiam os programas em memória física com relocação (seção 8.5.2), carregadores que copiam os programas em memória virtual (seção 8.5.3).

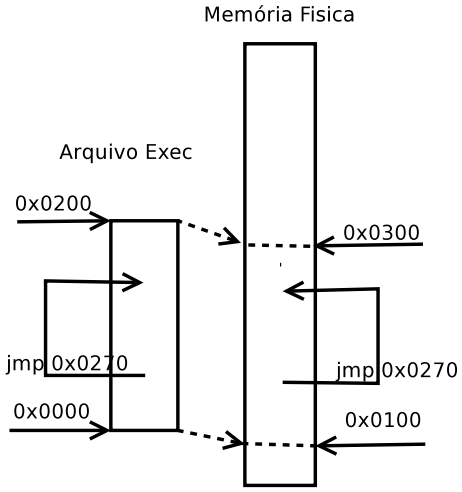

Figura 8.4: Modelo funcionamento do carregador COM

8.5.1 Carregadores que copiam os programas em memória física sem relocação

Esta classe de relocador corresponde ao modelo mais simples, e foi vastamente utilizada nos primeiros programas em PC. O seu membro mais conhecido é o carregador que trabalha com o formato COM do MS-DOS.

Antes de surgir o modelo EXE, este era único tipo de carregador fornecido pelo MS-DOS, e ele é bastante simples. Quando um programa é colocado para a execução, o carregador lê o arquivo que contém o programa a ser executado. Este arquivo, como já sabemos, deve estar organizado em um formato especial para que o carregador possa entendê-lo.

Assim, quando o carregador confirma que o arquivo em questão está no formato apropriado, ele copia o arquivo a partir do endereço de memória `0x0100` (na região da memória entre os endereços `0x0000` e `0x0100`, chamada PSP, onde são armazenadas outras informações, como por exemplo a linha de comando digitada).

Quando o programa estava totalmente copiado na memória física, o controle de execução era passado para o endereço `0x0100`, ou seja, o programa começava a execução.

A figura 8.4 mostra esquematicamente o trabalho do carregador. O exemplo desta figura contém um programa de `0x0200` bytes no formato COM. Como já foi visto, este programa será copiado para a memória a partir do endereço `0x0100`. Desta forma, o endereço `0x0000` do arquivo executável será copiado para o endereço `0x0100` da memória física e assim por diante até o endereço `0x0200` do arquivo executável que será copiado para o endereço `0x0300` da memória física.

Um detalhe importante que está colocado na figura é o comando `jmp`. O parâmetro

deste comando é o endereço para qual desviar. Este endereço corresponde ao endereço em memória física. Este endereço é conhecido pelo ligador porque o ligador sabe que o programa será carregado a partir do endereço `0x0100`. Sendo assim, o arquivo executável gerado pelo ligador já contém o endereço correto. Aliás, além de correto, este endereço é imutável, o que limita sensivelmente este modelo.

Como exemplo, suponha que este programa seja carregado a pertir de outro endereço, digamos `0x1000`. Então, quando o programa chegar no comando `jmp`, o fluxo será desviado para `0x0270`, e vai seguir a execução a partir daquele ponto (seja lá o que ele encontrar).

Após este exemplo já é possível examinar algumas características deste modelo:

1. O MS-DOS era um sistema operacional monousuário e monoprocesso. Isto significa dizer que só um processo podia estar em execução em cada momento. Outro processo só podia entrar em execução se o processo atual fosse cancelado (ou copiado temporariamente para outro local).

2. O carregador COM não fazia relocação de endereços. Como o ligador sabe que este será o único processo a ser executado e que ele começa sempre no endereço `0x0100`, então os endereços de todos os objetos (aqueles normalmente associados a rótulos, como endereços de desvio, variáveis e procedimentos) já podiam ser calculados em tempo de ligação.

3. Em versões posteriores do MS-DOS, tentou-se criar uma maneira de colocar mais de um processo em execução. O mecanismo consistia em chamar o sistema operacional quando determinadas teclas fossem acionadas pelo usuário (por exemplo, ALT e TAB) para que toda a imagem do processo em execução fosse copiado para uma área de salvamento em disco, e que outro processo fosse colocado em execução em seu lugar. Quando o usuário quisesse colocar o outro processo de volta para a execução, bastava usar as duas teclas mágicas para que a imagem do processo atual fosse salvo em disco e que a imagem salva do outro processo fosse copiado novamente para a memória. Com isso, criava-se a ilusão de um ambiente multiprocessos onde o mecanismo de troca de processos era gerenciado pelo usuário.

4. Para que um processo pudesse ser colocado em execução, deveria haver memória física suficiente para comportá-lo. Curiosamente, esta primeira versão do MS-DOS trabalhava com um dos primeiros processadores intel `x86`, onde podia-se endereçar até 64Kbytes. Sendo assim, nenhum programa executável podia ser maior do que 64K. Posteriormente este limitação caiu, porém algumas características importantes (como tamanho do segmento de 64k) continuaram por ainda alguma tempo.

O formato COM é o mais simples de todos os formatos apresentados neste texto. Alguns aspectos dele foram projetados para trabalhar no computador IBM-PC, e baseiam-se em várias características de hardware, em especial do processador intel. A melhoria dos processadores permitiram melhorias no modelo de execução. Para maiores referências a estas melhorias, consulte [Paz07]

8.5.2 Carregadores que copiam os programas em memória física sem relocação

É evidente que as deficiências no modelo COM eram graves, e algumas tinham raízes no próprio processador intel da época (que eram basicamente de 8 bits). Quando evoluiram para processadores de 16 bits, a intel encontrou uma forma de endereçar até 20 bits usando registradores de segmento, o que permitiu um endereçamento total de até 1Mbytes de memória física.

Com esta melhoria do hardware, era necessária uma melhoria do sistema operacional também, especialmente uma melhoria no modelo no formato de execução. O modelo COM, monoprocesso, continuou existindo, porém a Microsoft criou o modelo EXE (que também é bastante semelhante ao modelo de mesmo nome do CP/M). Este modelo debutou no MS-DOS 2.0.

Este modelo usa várias características interessantes, algumas destas característics baseavam-se em aspectos do hardware, e fogem ao escopo da discussão. Porém uma característica importante foi a de usar uma tabela de relocação, que apresentava os endereços de todos os símbolos que deveriam ser relocados (alterados) quando o programa fosse colocado em execução.

O carregador lia o arquivo executável, guardava os endereços de relocação e então colocava o programa para executar em qualquer endereço físico disponível, digamos `0x0700`. Todos os endereços reloáveis eram procurados na memória e substituídos pelos novos endereços relativos a `0x0700`. Quando este procedimento terminava, o programa podia ser disparado a partir do endereço `0x0700`, pois todos os endereços relocáveis já foram ajustados.

Este modelo permitiu um sistama realmente multiprocessos, porém a questão de gerenciamento de procesos e de memória ainda era bastante rudimentar.

Uma outra característica importante desta classe de carregadores é que toda a área de memória era considerada área válida para execução. O usuário que, sem querer (ou por querer) tinha acesso a locais onde hoje se prega que não devem ter acesso, como ao sistema operacional, sendo que poderiam alterá-lo. Um exemplo é a quantidade de livros que foram lançados na época explicando como alterar os destinos das interrupções contidas no vetor de interrupções.

O comprometimento da segurança permitiu o rápido ataque de diversos programas mal-intencionados (e por vezes divertidos), chamados vírus. Quando o MS-DOS foi estendido para uso em redes de computadores, foi possível escrever vírus que se reproduziam em todos os computadores da rede.

Porém, também é importante destacar que o MS-DOS foi projetado para operar em um computador, com um único processo para um único usuário. Neste ambiente, ao acreditar que o usuário é bem intencionado (afinal, porque criar um vírus para atacar o próprio computador?), não há motivos para preocupações muito grandes. Porém, quando este sistema foi estendido para multiusuários e para redes de computadores, o tímido modelo de segurança simplesmente ruiu. Desde esta ruptura, o número de vírus aumentou significativamente, sempre atacando falhas deixadas a partir da própria concepção do sistema operacional.

Apesar de Microsoft alegar que as versões atuais de seus sistemas operacionais são mais resistentes a falhas e vírus, a quantidade de vírus detectados semanalmente

mostram que estas falhas ainda existem.

8.5.3 Carregadores que copiam os programas em memória virtual

Como foi visto nas duas seções anteriores, o mapeamento de um programa diretamente em memória física pode ocasionar problemas. O principal destes é o acesso a toda a memória física através de qualquer programa.

O modelo de memória virtual soluciona este problema, uma vez que cada programa recebe uma grande quantidade de memória virtual linear para trabalhar (uma grande caixa de areia onde ele pode fazer o que quiser). Pedaços desta memória virtual são mapeados para a memória física, e o cada processo só tem acesso às suas páginas físicas. O gerenciamento de memória virtual é feito diretamente em hardware e pelo sistema operacional fazendo com que o mapeamento seja transparente para o usuário e rápido. Evitentemente não tão rápido quanto os modelos anteriores, uma vez que o tempo de carregar novas páginas da memória virtual para a física pode gerar pequenos atrasos.

Aqui, o carregador deve copiar o programa contido no arquivo para o espeço de memória virtual alocado para este programa. Depois de copiar todas as seções (text, data, bss, etc.), basta habilitar o programa para a execução.

Aqui também existe relocação, porém ela depende do tipo de biblioteca (estática, compartilhada e dinâmica) estão contidos no arquivo executável. Trataremos cada caso em uma seção diferente.

8.5.3.1 Executáveis com Bibliotecas Estáticas

Quando as bibliotecas são estáticas, o ligador gera endereços virtuais fixos, sem relocação. A tarefa do carregador é basicamente copiar as seções para a memória virtual. É bastante semelhante ao relocador do modelo COM, porém como trabalha-se com memória virtual e não com memória física, não tem os inconvenientes daquele modelo.

8.5.3.2 Executáveis com Bibliotecas Compartilhadas

Quando um arquivo executável faz referência a bibliotecas compartilhadas, o carregador tem de fazer muito trabalho.

Depois de copiar o arquivo executável para a área de memória virtual daquele processo, o carregador copia as bibliotecas compartilhadas para outra área da memória virtual. Aqui é importante observar que somente depois disto é que o carregador sabe quais são os endereços relocáveis (pois só agora sabemos quais os endereços virtuais de cada objeto relocável).

O problema do carregador então passa a ser encontrar todos os endereços contidos no arquivo carregado que exigem relocação, para então atribuir a eles os endereços corretos.

Para facilitar esta tarefa, o ligador cria uma seção contendo todos os objetos do arquivo executável que devem ser relocados. Existem várias formas de se fazer a ligação

entre o arquivo executável e as bibliotecas que ele usa. A idéia mais simples (e ineficiente) é listar todas os enderços que devem ser relocados e esperar que o carregador determine quais os endereços corretos em tempo de execução, substituíndo cada uma das entradas. Isso causa algumas complicações relacionadas com endereçamento, e por isso não é adotada.

Quando se fala somente em desvios, uma solução mais inteligente é usar uma tabela de desvios, onde cada entrada corresponde a um símbolo. Assim, cada procedimento a ser relocado tem uma entrada, e cada entrada tem um pequeno trecho de código. O programa executável desvia para esta entrada e esta entrada desviará para o destino apropriado. Neste caso, o carregador só precisa relocar o destino deste trecho de código, deixando todo o resto do código intacto.

Esta idéia é implementada em vários modelos de execução, inclusive no ELF, onde a tabela é chamada `plt`[4]. Assim, ao invés de o arquivo executável conter chamadas aos procedimentos que serão ligados em tempo de execução, ele chama os procedimentos contidos na seção `.plt`, que por sua vez irá desviar o fluxo para as bibliotecas compartilhadas que forem carregadas. A vantagem deste método é que não é necessário substituir todos os destinos das chamadas de procedimento que estão contidas no arquivo executável. Nenhuma é alterada na seção `texto`, e o carregador deve somente alterar os endereços contidos na seção `plt`. Desta forma, se um programa fizer duzentas chamadas ao procedimento `P1`, então o carregador não precisa alterar o endereço destas duzentas chamadas, uma vez que elas desviam para uma entrada na seção `plt`. Basta ao carregador alterar a entrada da seção `plt` para o destino correto.

Uma entrada simplificada na seção `plt` é apresentada a seguir para o programa `O1`(seção 8.4.4).

```
> objdump -S O1
  (...)
Disassembly of section .plt:
  (...)
 080483b0 <P4@plt>:
 80483b0: ff 25 a4 96 04 08     jmp    *0x80496a4
 80483b6: 68 18 00 00 00        push   $0x18
 80483bb: e9 b0 ff ff ff        jmp    8048370 <_init+0x18>

 080483c0 <P2@plt>:
 80483c0: ff 25 a8 96 04 08     jmp    *0x80496a8
 80483c6: 68 20 00 00 00        push   $0x20
 80483cb: e9 a0 ff ff ff        jmp    8048370 <_init+0x18>

Disassembly of section .text:
080483d0 <_start>:
(...)
 80484ad: e8 fe fe ff ff        call   80483b0 <P4@plt>
 80484b2: c7 05 bc 96 04 08 04  movl   $0x4,0x80496bc
 80484b9: 00 00 00
```

Há pelo menos dois aspectos a serem analisados. Primeiro, observe que a chamada ao procedimento P4 (`call 80483b0 <P4@plt>`). A referência é `P4plt`, ou seja, o procedimento P4 que está na seção `plt`. O endereço do procedimento

[4]Procedure Linkage Table.

(`0x80483b0`) está listado um pouco acima. Isto significa que o programa executável contém um desvio para `P4`, porém não aquele que será ligado em tempo de execução, mas um outro, na seção `plt`, que será futuramente alterado para desviar para o procedimento da biblioteca dinâmica carregada.

O segundo aspecto corresponde ao formato do endereço de desvio `jmp *0x80496a4`. Este modo de endereçamento correponde a desvio indireto, ou seja, o desvio é para o endereço contido em `0x80496a4`, e não para `0x80496a4`. A tarefa do carregador é colocar o endereço correto de `P4` no endereço `0x80496a4`.

8.5.3.3 Executáveis com Bibliotecas Dinâmicas

8.5.4 Emuladores e Interpretadores.

Emuladores e interpretadores são termos que muitas vezes são usados

Emulador O termo emulador está associado a sistemas de hardware ou de software básico. Um emulador é capaz de duplicar o comportamento das funções de um sistema "A" em um sistema "B" de tal forma que o sistema "B" se comporte exatamente como se fosse o sistema "A".

Interpretador Um termo interpretador é normalmente associado a linguagens de programação. Um interpretador é um sistema capaz de executar instruções de uma linguagem interpretada.

Desta forma, existem linguagens de programação interpretadas e emuladores de hardware. A seção 8.5.4.1 detalha como funcionam os emuladores enquanto que a seção 8.5.4.2 detalha como funcionam os interpretatores.

8.5.4.1 Emuladores

O termo emulador foi cunhado para descrever um hardware que se comporta como outro. Porém, com o tempo, o termo foi estendido para descrever sistemas de sofware básicos também. Um exemplo interessante é o wine[5] (Wine Is Not Emulator - um emulador de aplicações windows para linux).

Este programa lê um arquivo executável no formato usado pelo windows (COM, EXE, PE) e converte para o formato ELF. Após esta conversão, o programa é colocado em execução em linux. A idéia é simples: a seção de intruções do formato origem é copiada para seção "text" do ELF. A seção de dados globais é copiada para a seção "data", e assim por diante. Com isso, o formato original é totalmente convertido para ELF e o programa pode executar.

Porém, ainda resta um problema. Em tempo de execução, um programa pode (e deve) usar chamadas ao sistema. O programa origem foi projetado para fazer uso das chamadas de sistema da Win32 (o equivalente ao POSIX do linux), e da mesma forma: indica parâmetros nos registradores e chama a instrução `int 0x????` (os endereços são diferentes do linux).

[5] http://www.winehq.org

Assim, a grande tarefa do wine não é o de copiar o arquivo windows para o formato executável ELF, mas de projetar chamadas de sistema equivalentes ao Win32 para linux. A idéia trocar chamadas de sistema por chamadas a procedimentos. Se necessário, estas chamadas de procedimento usam as chamadas de sistema nativas do linux.

A versão da Win32 que foi projetada para o wine não é uma cópia da win32, mas sim uma versão livre baseada em engenharia reversa. Para programas bem comportados (ou seja, os que só acessam o hardware através das chamadas ao sistema), ele funciona muito bem. Porém para programas que acessam o hardware diretamente, não funciona, evidentemente.

O grande problema é como converter gráficos complexos (por exemplo simulações e jogos) do windows para linux, e a idéia natural é usar uma bibliteca gráfica para fazer o trabalho. Porém, como as bibliotecas gráficas da Microsoft (DirectX) e do linux (OpenGL) não são totalmente compatíveis, os gráficos podem ter de ser emulados, gerando perda de desempenho.

8.5.4.2 Interpretadores

Um interpretador é um programa que interpreta instruções. Para tal, ele deve ser invocado e deve ser dito para ele qual o programa que ele deve executar.

Observe que o programa a ser executado residirá na seção ".data", enquanto que o interpretador residirá na seção ".text".

O interpretador pode alocar variáveis, chamar procedimentos e qualquer outra coisa que a linguagem especificar. Por exemplo, ao ler uma intrução que diz para colocar zero em uma variável, o interpretador irá procurar esta variável em sua tabela de variáveis e ao encontrá-la, atribuir zero a ela.

A maioria das linguagens interpretadas permite alocação de variáveis em tempo de execução (e não antes como foi visto anteriormente). Assim, se a variável "Z" recebe o valor zero, e não foi declarada, o interpretador a insere em sua tabela de variáveis e atribui zero a ela. Apesar de conveniente, isso já causou sérios problemas, pois se o programador errar e digitar "X" ao invés de "Z", a variável "X" será declarada e coexitirá com Z. O problema é como detectar este erro.

Aqui é importante destacar a linguagem Java. Ela é uma linguagem de programação criada pela SUN para orientação a objetos baseada fortemente na sintaxe da linguagem C. Um compilador Java é capaz de traduzir a linguagem Java para um assembly particular chamado "bytecode". A peculiaridade do "bytecode" é que ele é um assembly para uma CPU inexistente. O código executável segue o modelo ELF apresentado aqui com várias seções (".text", ".data", etc.). A diferença é que a seção ".text" tem bytecodes ao invés de intruções assembly nativas.

A grande sacada é que a SUN também projetou interpretadores para este formato executável. Estes interpretadores vêm em vários "sabores": windows, linux e mac são os procipais. Com isso, a linguagem Java é compilada por um compilador Java que gera um arquivo executável para bytecodes. O interpretador lê este arquivo executável e pode executar o mesmo código em vários sistemas diferentes. Como o arquivo executável independe da arquitetura em que foi gerado (ou seja, os compiladores java geram o mesmo arquivo executável para um mesmo arquivo fonte Java), este arquivo executável é "multiplataforma". Um mesmo arquivo executável Java pode ser executado

(interpretado) em windows, linux e mac sem alterações!

Apêndice A

Formatos de Instrução

Este apêndice aborda alguns dos formatos de instrução nos processadores da família x86[1]. Com formato de instrução, entende-se a forma com que a instrução é dividida para conter o código de operação e os operandos (se houverem).

Este tópico é mais simples quando analisamos processadores RISC (reduced instruction set computer) , onde código de operação e operandos estão organizados de maneira bastante regular em dois ou três formatos diferentes. Nestas máquinas, o código de operação tem um tamanho fixo, e ocupa o mesmo espaço em qualquer dos formatos. Já em processadores CISC (complex instruction set computer), não há esta regularidade, e os códigos de operação podem ocupar de poucos a muitos bits e vários formatos de instrução são adotados, o que torna difícil determinar quando termina o código de operação e quando começam os operandos.

Esta é um dos aspectos que fazem com que os processadores RISC sejam considerados "melhores" do que os computadores CISC (apesar de isso nem sempre ser verdade). Nas máquinas CISC, como as instruções aparecem em grande número e são complexas, a unidade de controle deve ser maior para contemplar todas as regras e todas as excessões dos vários formatos. A CPU deve ficar maior, e por conseqüência as distâncias entre os componentes internos ficam maiores também, fazendo com que os dados demorem mais para trafegar entre os componentes (por exemplo, entre registradores). O efeito da implementação "direta" de um processador CISC é que as suas intruções ficam mais lentas.

Por outro lado, a regularidade das instruções RISC faz com que a unidade de controle seja mais simples, e conseqüentemente menor. Com isso, a CPU pode ser projetada de forma mais simples e as distâncias entre os componentes também ficam menores. A conseqüência é que as instruções demoram menos tempo para serem completadas, e o efeito é que elas passam a ser mais rápidas.

Os processadores da família x86 são exemplos de processadores CISC e honram a

[1]Originalmente estes processadores foram desenvolvidos pela Intel. Exemplos destes processadores: 80186, 80286, 80386, ... É importante destacar que a AMD fez "engenharia reversa" destes processadores e desenvolveu processadores que atuam com o mesmo conjunto de instruções. Assim, do ponto de vista do usuário, os processadores x86 da Intel e da AMD tem o mesmo comportamento, e funcionam igualmente para o mesmo conjunto de programas.

classe. Eles utilizam vários modos de endereçamento e vários tamanhos para o código de operação.

Aqui o leitor deve estar se perguntando: então porque ainda insistimos com processadores CISC (em especial da família x86) e não nos convertemos para processadores RISC? A resposta é o preço. Cada novo processador da família x86 executa o conjunto de instruções de seus predecessores de forma mais rápida que gerações anteriores (maravilhas da engenharia), acresentam algumas instruções novas de acordo com as necessidades de mercado (como por exemplo MMX e SSI) e com isso tem mercado praticamente garantido para venda. Com a massificação, os custos de projeto e de produção são minimizados (para não dizer pulverizados). Já os processadores RISC nem sempre tem mercado garantido (pelo menos não na magnitude dos x86). Com isso, tanto os custos de projeto quanto os de produção podem ser sentidos no preço final do processador.

A.1 Modos de Endereçamento

Os operandos de uma instrução em assembly podem variar de acordo com o local em que o dado se encontra. Como exemplo, observe a diferença entre as instruções:

- `movl %eax, %ebx` (endereçamento registrador)
- `movl $0, %ebx` (endereçamento imediato)
- `movl a, %ebx` (endereçamento direto)
- `movl %eax, (%ebx)` (endereçamento indireto)
- `movl a(,%edi,4), %ebx`. (endereçamento indexado)

Estes são os modos de endereçamento que esta seção abordará para processadores da família x86 (e mais freqüentemente encontrados em processadores modernos), e esta seção detalha o formato de cada modo de endereçamento nos x86.

Existem outros modos de endereçamento não descritos acima porque eles não foram implementados nesta família de processadores, como por exemplo os que lidam com dois operandos em memória. Eles não são descritos aqui porque além de eles não terem sido implementados nesta família de processadores eles estão praticamente em desuso nos processadores modernos.

Antas de iniciar a explicação, existem dois aspectos que queremos destacar:

1. Nos exemplos acima foi usada somente a instrução `movl`. Apesar de ser uma única instrução, ela foi projetada para conter um número variado de operandos. Cada um destes modos de endereçamento da instrução `movl` tem um código de operação diferente, o que os diferencia internamente para o processador. Observe agora que é possível identificar cada modo de endereçamento a partir da forma com que se escreve a instrução. Esta forma de escrever a instrução é estendida para praticamente todas as instruções aritméticas e lógicas, ou seja, compreendendo a forma de escrever o modo de endereçamento registrador com

a instrução `movl`, também se compreende a foram de endereçamento registrador para a instrução `addl`.

2. A nomenclatura dos modos de endereçamento (Registrador, Imediato, Direto, Indireto e Indexado) não é única. Os manuais da Intel usam uma taxonomia um pouco diferente: imediato, registrador, memória (que aqui está dividido em base/deslocamento e base/índice/deslocamento) e portas de I/O (que não será coberto aqui). Para maiores detalhes veja [Int04a, pp 3-20].

A.2 Registrador

Este modo de endereçamento envolve dois operandos que são registradores.

Um exemplo simples é a instrução `movl %eax, %ebx`, onde o conteúdo do registrador `%eax` será copiado para `%ebx`. A instrução contém três informações: código de operação, operando 1 (`%eax`) operando 2 (`%ebx`). O modo de endereçamento está implícito no código de operação (ou seja, o código de operação neste caso é: copie o valor de %eax para %ebx usando endereçamento registrador). Para outros modos de endereçamento, o código de operação é diferente.

Os detalhes de endereçamento da instrução `movl` pode ser encontrada em [Int04b], no comando `mov`/`move`.

Em um arquivo executável, esta instrução será associada aos símbolos hexadecimais `0x89C3` (escreva um programa assembly que contém esta instrução e veja).

A instrução contém os três operandos que são embutidos no padrão de bits indicado. O código de operação corresponde aos primeiros 10 bits (1000100111). O trecho da instrução que diz que o operando destino é `%eax` corresponde aos três bits seguintes (000), enquanto que os três últimos bits indicam o segundo operando ((011). Abaixo temos os padrão de bits para esta intrução. Considerando as siglas CO=Código de Operação, Op1=Operando 1, Op2=Operando 2, temos:

$$\underbrace{1000100111}_{CO}\ \underbrace{000}_{Op1}\ \underbrace{011}_{Op2}$$

Observe que o registrador `%eax` é $(000)_2 = 0_{10}$, enquanto que o registrador `%ebx` é $(011)_2 = 3_{10}$. Seria natural esperar que o registrador `ebx` fosse o de número 1, porém a relação é diferente, como pode ser visto na tabela A.1.

A.3 Imediato

Este modo de endereçamento envolve uma constante (também chamada de valor imediato) que é codificada na própria instrução. Nos x86, é possível codificar constantes de 8, 16 e 32 bits.

Como exemplo, considere a instrução `movl $1, %eax`, cuja representação em hexadecimal é `b8 01 00 00 00 00` (seis bytes). Em binário, temos:

Registrador	Número
`eax`	`000`
`ecx`	`001`
`edx`	`010`
`ebx`	`011`
`esp`	`100`
`ebp`	`101`
`ebi`	`110`
`edi`	`111`

Tabela A.1: Números internos associados aos registradores x86

$$\underbrace{1011}_{CO}\ \underbrace{1}_{w}\ \underbrace{000}_{OP1}\ \underbrace{0000\ 0001\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000}_{Imediato}$$

O código da instrução ocupa os primeiros quatro bits. O bit seguinte é um modificador (*w*) que indica que a instrução opera sobre uma palavra (32 bits). Os três bits seguintes indicam o registrador destino ($(000)_2$=`%eax`), enquanto que os últimos 32 bits armazenam a constante. Lembre que os x86 são *little endian*[2] por isso os bytes aperecem invertidos. Como exemplo veja que a constante `0x12345678` é armazenada como `78 56 34 12`.

A.4 Endereçamento Direto

No endereçamento direto, a instrução contém o endereço do dado a ser utilizado, como por exemplo na instrução `movl a, %eax`, onde "a" é um rótulo na seção data (lembre-se que os endereços dos rótulos são definidos em tempo de ligação). Assumindo que o endereço do rótulo "a" será `0x8049098`, então a instrução será codificada como `0xa1 98 90 04 08`, ou em binário:

$$\underbrace{1010}_{CO}\underbrace{0001\ 1001\ 1000\ 1001\ 0000\ 0000\ 0100\ 0000\ 1000}_{Op1}$$

Se alterássemos a instrução para `movl a, %ebx`, teríamos o seguinte padrão `0x8b 1d 9c 90 04 08`. É interessante observar que esta instrução é completamente diferente da anterior, pois o código de operação daquela instrução se aplica somente para mover um dado para o registrador `%eax`, enquanto o código de operação desta instrução se aplica a qualquer um dos outros registradores, e para tal, um dos operandos indica qual dos registradores está sendo envolvido. A esta altura, o leitor já

[2] Segundo a webopedia, os termos *big endian* e *little endian* são derivados dos liliputianos das viagens de Gulliver, onde o maior problema político era se os ovos cozidos deveriam ser abertos no lado maior ou no lado menor. De maneira análoga, o debate em computação está também mais relacionado com problemas políticos do que com méritos tecnológicos segundo a mesma fonte.

deve ter observado, mas não custa nada observar mais uma vez que instruções específicas como esta são muito comuns em arquiteturas CISC, e tornam o seu conjunto de intruções mais complexo, maior e irregular.

A.5 Endereçamento Indireto

No modo de endereçamento indireto, um dos parâmetros contém uma referência indireta à memória. Como exemplo, considere as seguintes instruções:

```
movl $a, %ebx
movl (%ebx), %eax
```

A primeira instrução coloca em `%ebx` o endereço do rótulo "a" (sem o $, seria colocado o conteúdo do endereço de "a"). A segunda instrução coloca em `%eax` o valor do endereço indicado por %ebx. Estas duas instruções são uma versão mais complicada para implementar a instrução `movl $a, %eax`.

A instrução `movl (%ebx), %eax` é codificada em hexadecimal como `8b 03`, e em binário, temos:

$$\underbrace{10001011}_{CO}\ \underbrace{00}_{?}\ \underbrace{000}_{OP1}\ \underbrace{011}_{OP2}$$

Observe que alguns bits estão "amarrados" em zero. Eles não fazem parte do código de operação, e talvez não devessem estar ali. Porém, estão lá para que a instrução preencha os dois bytes.

A.6 Endereçamento Indexado

O que aqui chamamos de endereçamento indexado também recebe o nome de endereçamento de memória base e deslocamento.

No endereçamento indexado, a instrução contém o endereço base de onde buscar o dado e um deslocamento. Um exemplo é a instrução `movl a(,%edi, 4), %ebx`, que usa "a" como base é $\%$`edi` $\times 4$ como deslocamento. O endereço efetivo é `a` $+$ $\%$`edi` $\times 4$, e o conteúdo deste endereço é movido para `%ebx`. Assim como no exemplo anterior, assumimos que o endereço de "a" é 0x80490a0 para obter a seguinte instrução em hexa: `0x8b 1c bd a0 90 04 08`

$$\underbrace{10001011}_{CO}\ \underbrace{00}_{}\ \underbrace{01}_{\%ebx}\ \underbrace{1}_{}\ \underbrace{100}_{4}\ \underbrace{1011\ 1101}_{?}\ \underbrace{\ldots}_{end.a}$$

Os três primeiros bytes armazenam o código de operação, o registrador destino, o registrador a ser acrescentado (no caso, `%edi` e o multiplicador `4`). Os últimos bytes contém o endereço de a.

Apêndice B

MMX

Esta seção trata de processamento paralelo. Apresentaremos a abordagem que a Intel adotou para inserir processamento paralelo nos processadores da família ix86.

O processamento paralelo é uma das alternativas mais interessantes para aumentar a velocidade de um processador. Existem várias abordagens para tal, e uma delas é a SIMD[1], onde uma instrução pode efetuar operações sobre vários dados ao mesmo tempo. O projeto de uma CPU com processamento paralelo é algo muito caro e normalmente é para um mercado específico. Assim, para manter o mercado dos processadores da família ix86, e melhorar o desempenho dos processadores desta família, a Intel optou por acrescentar um coproessador (oito registradores de 64 bits e 57 instruções que operam sobre sobre registradores). Isto pode parecer estranho (um coprocessador embutido dentro de outra CPU), porém isso é bastante comum. Antigamente existiam coprocessadores aritméticos (e espaço para eles na placa-mãe), porém atualmente a esmagadora maioria dos coprocessadores está embutida dentro da CPU principal (que normalmente lida só com inteiros). É muito mais simples (e economicamente viável) criar um coprocessador a um processador de sucesso do que criar um processador novo, mas talvez sem mercado.

Este coprocessador foi projetado para melhorar o desempenho de aplicações multimídia (como por exemplo software interativo em tempo real[2]) e comunicações. Este apêndice irá explicar resumidamente como desenvolver aplicações em assembly para este coprocessador, porém o enfoque é explicar como funciona processamento paralelo e por isso o coprocessador MMX será utilizado como o exemplo de aplicação deste conceito. O texto serve como um ponto de partida para o leitor desenvolver aplicações MMX, porém não tem a intenção de ser completo.

Este apêndice está dividida em duas partes: A seção B.1 explica como verificar se o processador contém MMX e a seção B.2 apresenta uma aplicação simples (soma um a cada elemento de um vetor de inteiros).

[1] Single Instruction Stream Multiple Data Stream

[2] mais conhecidos como joguinhos.

B.1 Como verificar a presença do MMX

Os manuais da intel[Int04a, Int04b, Int04c] apresentam o código abaixo para verificar a presença ou não do coprecessador MMX no processador. É importante que o programador implemente uma alternativa para o caso do processador não estar presente, porém a nossa implementação simplesmente cancela o programa se o MMX não for detectado.

O algoritmo 44 foi projetado para ser chamado a partir de um programa em C, e por isso primeiramente ele termina de alocar o registro de ativação. Isto pode até ser desnecessário, porém adotamos esta solução para manter um padrão. O teste da presença de MMX é feito pela instrução `CPUID`, que insere 1 no bit 23 de `%edx`. A instrução `test` compara o padrão `$0x00800000` (bit 23 aceso e todos os demais apagados) com o registrador `%edx`. Se o resultado for zero, o bit 23 de `%edx` está apagado (MMX ausente), e se for 1, o bit 23 de `%edx` está aceso (MMX presente). Ao final, o registro de ativação é liberado. O retorno da função é 1 se MMX estiver presente e zero caso contrário.

```
1  procura_MMX:
2      # termina de alocar Reg.Ativação
3      pushl %ebp
4      movl %esp, %ebp
5      mov $1, %eax
6      CPUID
7      # Analisa bit 23 de %edx. Se estiver ligado, a CPU tem MMX.
8      test $0x00800000, %edx
9      jnz MMX_Presente
10     mov $0, %eax # MMX ausente. Retorna zero.
11     jmp fim
12 MMX_Presente:
13     mov $1, %eax # MMX presente. Retorna um.
14 fim:
15     # liberacao de Reg.Ativação
16     movl %ebp, %esp
17     popl %ebp
18     ret
```

Algoritmo 44: Verifica presença de MMX

B.2 Exemplo de aplicação paralela

Esta seção exemplifica o uso do coprocessador MMX, de seus registradores e instruções. Para tal, é utilizada uma aplicação simples: somar um a vários bytes consecutivos armazendos em memória.

Esta aplicação não foi projetada para ter uma finalizade prática, porém pode ser

utilizada em processamento de imagens. Nesta área, cada pixel de uma imagem em tons de cinza é normalmente representado por um byte (enquanto que em imagens coloridas são usados mais bytes). Um vetor de `unsigned int` armazena todos os pixels. Se a imagem tiver X pixels de largura por Y pixels de altura, então o vetor deve ter espaço para pelo menos $X \times Y$ bytes.

Cada byte do vetor corresponde a um pixel. Assim, o byte 1 corresponde ao pixel do canto superior esquerdo da imagem. Se o byte tiver o valor 0, então o pixel é preto. Se o byte tive o valor 255 (o maior inteiro sem sinal que pode ser representado em oito bits), o pixel é branco. Quanto maior o valor, mais claro é o pixel.

Suponha que desejamos somar um a cada um dos bytes do vetor. A primeira solução, a mais "direta", é implementar um laço que lê um byte de cada vez, coloca-o em um registrador, soma um a este registrador e armazena o byte de volta no vetor.

Apesar de funcionar, esta solução é lenta pois o processador fica parado a maior parte do tempo, uma vez que o acesso à memória é relativamente mais lento do que o acesso a registradores. Além disso, o barramento entre o processador e a memória é capaz de transportar 32 bits por vez, e somente oito estão sendo transportados a cada iteração do ciclo.

Uma solução mais prática é trazer quatro pixels por vez para a CPU (32 bits), o que usa o máximo possível do barramento. O problema agora é separá-los em bytes e somá-los individualmente.

Como este tipo de aplicação é comum, a tecnologia MMX permite que uma operação (como por exemplo "somar um") seja efetuada em paralelo em todos os quatro bytes de um registrador de 32 bits. Um registrador MMX contém 32 bits e pode ser somado **byte a byte** com um outro registrador. A soma de um byte não afeta a soma do byte vizinho.

Com MMX, o algoritmo passa a carregar os quatro bytes de uma imagem em um registrador MMX. Um outro regitrador mmx recebe o valor `$0x01010101` (que são quatro bytes com os valores 1) e os dois registradores são somados. O resultado é colocado de volta no vetor. Não é necessário pensar muito para perceber que este algoritmo é praticamente quatro vezes mais rápido do que o anterior. O algoritmo 45, e a função `somaIm` pode ser invocada de um programa em C com o seguinte protótipo: `somaIm (endInicioVetor, tamVetor)`.

As linhs 3-4 completam o registro de ativação e as linhas 21-22 liberam o registro de ativação. Lembre-se: esta função foi projetada para ser usada a partir de um programa em C.

Na linha 5, o registrador `%eax` recebe o endereço de início do vetor (que é o parâmetro armazendo em `8(%ebp)`). Na linha 6, o registrador `%edx` recebe o tamanho do vetor (que é o parâmetro armazendo em `12(%ebp)`). A linha 7 calcula o endereço do fim do vetor e armazena-o em `%edx`.

A cada iteração do laço, o valor de `%eax` é somado de um. Quando o valor de `%eax` for igual ao valor de `%edx`, o laço termina.

Na linha 9, a constante $0x01010101 é colocada em `%ebx` e depois em `%mm1`. Este valor corresponde a quatro bytes iguais a um. A idéia é carregar um conjunto de quatro bytes por vez da memória e somá-los em paralelo com `%mm1`. Ao final da soma em

```
1  somaIm:
2      # termina de alocar Reg.Ativação
3      pushl %ebp
4      movl %esp, %ebp
5      movl 8(%ebp), %eax
6      movl 12(%ebp), %edx
7      addl %eax, %edx
8      movl $0x01010101, %ebx # quero somar um a cada um dos bytes.
9      movd %ebx, %mm1
10 loop:
11     movl (%eax), %ecx
12     movq (%eax), %mm0
13     paddusb %mm1, %mm0
14     movq %mm0, (%eax)
15     addl $4, %eax
16     cmpl %eax, %edx
17     jge loop
18 fimSomaIm:
19     # inicia liberacao de Reg.Ativação
20     movl %ebp, %esp
21     popl %ebp
22     ret
```

Algoritmo 45: Soma um a um vetor

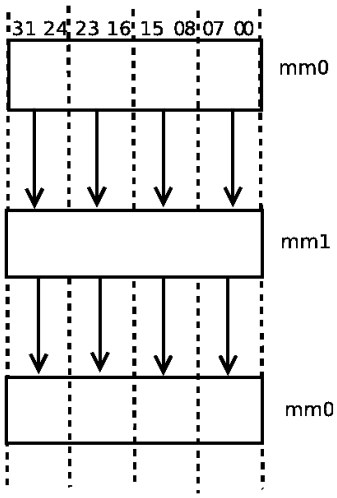

Figura B.1: Soma paralela em registradores MMX.

paralelo, o resultado será que os quatro bytes lidos terão sido somados de um[3].

O laço ocorre entre as linhas 10-17. Para colocar os primeiros quatro bytes do vetor em `%mm0`, usamos o registrador `%ecx` como intermediário (observe que `%eax` indica o endereço do próximo conjunto de quatro bytes a ser somado).

A soma em paralela é executada pela instrução `paddusb %mm0, %mm1`, e o resultado da soma é colocado direto na memória`movq %mm0, (%eax)`. A execução desta instrução está demonstrada esquematicamente na figura B.2.

Para exemplificar o que ocorre, suponha que `$mm0 = 0x00010203` e que `$mm1 = 0x01010101` (que é o valor que foi colocado lá, mas que o leitor pode modificar como quiser). Então, a instrução `paddusb %mm0, %mm1` realizará as seguintes somas em paralelo: `00+01, 01+01, 02+01, 03+01`, resultando em `$mm0 = 0x01020304`.

A seqüência de somas paralelas continua em cada nova iteração. O laço prossegue até o fim do vetor (ou seja, quando `%eax` $\geq$ `%edx` (lembre-se que `%edx` indica o endereço de fim do vetor.

O exemplo apresentado usa somente a instrução que soma quatro bytes em paralelo (quatro grupos de oito bits), porém existem outras instruções paralelas, e não só para grupos de oito bits. Também foram projetadas instruções para somar grupos de dezesseis bits e extensões para registradores de 64 bits. O elenco completo de instruções está disponível disponível nos manuais dos processadores. Existem também vários sítios na internet dedicados à programação usando mmx.

[3]pode ser que tenha um jeito mais simples de colocar uma constante em um registrador mmx, porém eu só encontrei a instrução `movq`. Se você encontrar uma alternativa, me avise

Bibliografia

[Bar04] Jonathan Bartlett. *Programming From The Ground Up*. Bartlett Publishing, 2004.

[Che] Benjamin Chelf. Building and using shared libraries.

[Com95] Tool Interface Standards Committee. *Executable and Linking Format (ELF)*. Maio 1995. Pode ser encontrado a partir de `http://refspecs.freestandards.org/`.

[GJ02] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2002.

[Int04a] Intel, editor. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, 2004. Order number 253665.

[Int04b] Intel, editor. *IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*. Intel, 2004. Order number 253666.

[Int04c] Intel, editor. *IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*. Intel, 2004. Order number 253667.

[JDD+04] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.

[Kow83] Tomasz Kowaltowski. *Implementação de Linguagens de Programação*. Guanabara, 1983.

[Lev00] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000. Versão gratuita na internet em `http://www.iecc.com/linker`.

[Paz07] Bjorn Martins Paz. *Uma Viagem ao Centro dos Executáveis*. Trabalho de Graduação - Depto. Informática - UFPR, 2007.

[Tan01] Andrew Tannenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[Whea] David A. Wheeler. Program library howto. http://www.linux.se/doc/HOWTO/Program-Library-HOWTO/index.html.

[Wheb] David A. Wheeler. System v application binary interface - draft. http://www.caldera.com/developers/gabi/latest/contents.html.