Produção do código executável

Ivan Ricarte

2008

Sumário

Interação do compilador com outros aplicativos

- Compilador atua como *front-end* para demais aplicativos
- Com as opções apropriadas, é possível interromper o processo de compilação antes de uma dessas etapas
- Nos compiladores gcc:

  -S Não faça a montagem, produto é um arquivo de texto com o código em linguagem simbólica

  -c Não faça a ligação, produto é um arquivo binário com o código-objeto

Montadores

- Instruções em linguagem simbólica refletem a arquitetura de cada processador
  - Quais operações podem ser realizadas
  - Que registradores estão disponíveis
  - Quais modos de endereçamento são suportados
- O programa montador reconhece essas instruções
  - E sabe como realizar a tradução do formato simbólico (texto) para o formato de máquina (binário)
- Quando montador que executa num processador permite a geração de código de máquina para um outro processador, ele é denominado montador multiplataforma (*cross-assembler*)

Montadores

- Traduzem código em linguagem simbólica para linguagem de máquina
  - Específicos para cada processador
- Facilidades presentes em montadores:

  Pseudo-instruções Diretivas que não geram código de máquina
    - Facilidades para simplificar codificação em linguagem simbólica

  Rótulos Identificação simbólica de endereços

  Macroinstruções Seqüências de instruções associadas a um nome

Estrutura de programas em linguagem simbólica

- Programas em linguagem simbólica têm estrutura rígida
  - Campo 1 Rótulo (opcional)
    - Associado a identificadores
  - Campo 2 Código de operação
    - Código da instrução do processador ou da pseudo-instrução do montador
  - Campo 3 Operandos
- Comentários também podem estar presentes

Exemplo de código em linguagem simbólica

```
        .file   "hello.cpp"
        .text
        .align 2
.LCFI2:
        movl    %eax, -4(%ebp)
        movl    %edx, -8(%ebp)
        cmpl    $1, -4(%ebp)
        jne     .L5
```

Exemplo de código em linguagem simbólica

```
POS       DS.W      1
; Busca 0 na sequencia de inteiros
; DATUM definido alhures
SRCH0     MOVEA.L   #DATUM,A0
          MOVE.L    #DATUM,D0    ; guarda inicio
          CLR.W     D1
LOOP      CMP.W     (A0)+,D1
          BNE       LOOP
          SUB.L     A0,D0
          MOVE.W    D0,POS
          RTS
          END
```

Facilidades presentes em montadores

Representação de constantes (exemplos)

Inteiro decimal 48

Inteiro hexadecimal $30

Inteiro octal @60

Inteiro binário %110000

Caractere simples ou seqüência ’0’, ’ABC’

Facilidades presentes em montadores

Pseudo-instruções (exemplos)

EQU

Associação de valores a nomes simbólicos

DC

Alocação de espaço para variáveis com valores inicializados

DS

Reserva de espaço para variáveis sem valores inicializados

Facilidades presentes em montadores

Pseudo-instruções (exemplos)

ORG

Definição de segmento (origem)

END

Sinalização do fim do programa em linguagem simbólica

GLOB

Indicação de símbolo que pode ser referenciado por outros módulos

Exemplos de uso de pseudo-instruções

EQU

Definição

```
SIZE      EQU      100
```

...

Uso

```
          MOVE     #SIZE,D0
```

Exemplos de uso de pseudo-instruções

DC, DS

```
CONTADOR   DC.L      100
ARR1       DC.W      0,1,1,2,3,5,8,13
MENSAGEM   DC.B      'Alo, pessoal!'
VALUE      DS.W       1
```

Exemplos de uso de pseudo-instruções

ORG, END

```
SEG1        EQU         $1000
            ...
            ORG         SEG1
            MOVE.W      DATA,D0
            MOVE.W      D0,DATA+2
            RTS
            ...
            END
```

Macroinstrução

- Seqüência de instruções que recebe um nome
- Futuras referências a esse nome serão substituídas pela seqüência de instruções
- Podem ter argumentos

Definição de macro Em geral, delimitada por pseudo-instruções de início e fim de definição de macro

Facilidades na definição da macro Comandos dentro do corpo da macro, para expansão condicional

Invocação da macro Substituição do nome referenciado pela sua expansão (instruções e argumentos)

Macro em montadores

Exemplo de definição

```
TOLOWER   MACRO    &IN,&OUT
          MOVE     &IN,D0
          ORI      32,D0
          MOVE     D0,&OUT
          ENDM
```

- Argumento `IN` é um caráter ASCII
- Se caráter é uma letra maiúscula, resultado é a letra minúscula correspondente
  - `OR` imediato com valor 32 — sexto bit é setado
  - `D0` é um registrador de dados do processador
- Resultado é colocado no argumento `OUT`

Processamento de macros

- Se todo uso de macro é precedido da correspondente definição, processamento da macro ocorre em um único passo
  1. Quando pseudo-instrução `MACRO` é encontrada, as instruções seguintes (até a pseudo-instrução `ENDM`, de fim de macro) são armazenadas em uma Tabela de Definição de Macro
     - Associada a essa tabela é armazenada também a lista de parâmetros da macro
  2. Quando uma operação utilizada no código é encontrada na Tabela de Definição de Macro, essa referência é substituída pelo conjunto de instruções lá armazenada
     - Parâmetros referenciados na tabela são substituídos pelos argumentos usados na referência à macro

O processo de montagem

Características gerais:

- Pré-processamento para eliminação de comentários e substituição de macros
- Varreduras por linha, com extração e processamento dos campos

Estruturas auxiliares

Tabela de instruções de máquina informação necessária para traduzir instruções simbólicas do processador para o correspondente código de máquina

- Depende do processador

Tabela de pseudo-instruções Informação necessária para processar pseudo-instruções do montador

- Não depende do processador, mas sim do montador

Um montador em dois passos

Passo 1 Processamento dos símbolos definidos no código

- Qual o valor associado a cada símbolo?
- Resultado armazenado numa tabela de símbolos

Passo 2 Geração do código de máquina

- Utiliza tabela criada no Passo 1 para codificar operandos das instruções de máquina
- Resultado armazenado em arquivo binário (módulo objeto)

Passo 1 do montador

Objetivo do primeiro passo é identificar a posição associada a cada identificador utilizado no código

- Uma instrução por linha, processada independentemente
- Identificadores no código assembly: sempre no campo de rótulo
- Processamento do rótulo: cria entrada na Tabela de Símbolos
  - Para cada identificador, o valor do endereço associado a ele
  - Informação obtida na MOT: quanto espaço é ocupado pela instrução
  - Informação obtida na POT: qual impacto da pseudo-instrução no valor do endereço
    - Altera valor do endereço? (`ORG`)
    - Precisa reservar espaço no módulo objeto? (`DS`, `DC`)
  - Se pseudo-instrução é `EQU`, entrada na Tabela de Símbolos é criada com o valor do argumento e não do endereço

Montador

Passo 2: criação do módulo objeto

- No segundo passo, cada linha do código é novamente processada
- Como todas as referências simbólicas foram resolvidas no passo anterior, código de máquina para cada instrução pode ser gerado
  - Para cada instrução do processador, código de operação é obtido da MOT
  - Operandos das instruções:

    Constante valor para operando obtido do próprio código da instrução (modo imediato)

    Simbólico valor para operando obtido da Tabela de Símbolos criada no primeiro passo
- Código binário gerado para cada instrução é agregado a um arquivo com o módulo objeto

Exemplo: código-fonte

```
DATA      EQU       $6000
PROGRAM   EQU       $4000
          ORG       DATA
VALUE     DS.W      1
RESULT    DS.W      1
          ORG       PROGRAM
PGM       MOVE.W    VALUE,D0
          MOVE.W    D0,RESULT
          RTS
          END       PGM
```

Exemplo: Tabela de símbolos

Símbolo	**Valor**
DATA	$6000
PGM	$4000
PROGRAM	$4000
RESULT	$6002
VALUE	$6000

Exemplo: Código gerado

Segmento de dados

Posição	$6000	$6002
Conteúdo	$0000	$0000

Segmento de texto (código)

Posição	$4000	$4002	$4004	$4006	$4008	$400A	$400C
Conteúdo	$3038	$0000	$6000	$31C0	$0000	$6002	$4E75

Carregadores e ligadores

Módulo objeto criado pelo montador ainda não é executável

- Pode ter referências a símbolos em outros módulos
  - Variáveis externas
  - Rotinas
- Precisa ser carregado em memória para execução
  - Pode necessitar ajustes nas referências à memória

Programas que executam essas tarefas

Ligador Resolve referências entre símbolos de diferentes módulos

Carregador transfere código objeto para a memória e dá início à sua execução

Formato do módulo objeto

Informação usualmente presente

Cabeçalho: contém a identificação do tipo de arquivo e dados sobre o tamanho do código e eventualmente o arquivo que deu origem ao arquivo objeto.

Código gerado: contém as instruções e dados em formato binário, apropriado ao carregamento.

Relocação: contém as posições no código onde deverá ocorrer mudanças de conteúdo quando for definida a posição de carregamento.

Símbolos: contém os símbolos globais definidos no módulo e símbolos cujas definições virão de outros módulos.

Depuração: contém referências para o código-fonte, tais como número de linha, nomes originais dos símbolos locais e estruturas de dados definidas.

Estratégias de carregamento e ligação

Esquemas absolutos

- Nesses esquemas, o programador é responsável pela realização dos ajustes
- Código já contém referências às posições efetivas de memória de variáveis e rotinas
- Esquemas adequados para sistemas de pequeno porte e monoprogramados

Exemplos:

Montador combinado com carregador Código gerado já é colocado na memória e não no disco

Carregador absoluto Módulo objeto contém referências efetivas à memória e precisa ser carregado em uma posição específica

Montagem e carregamento combinados

Desvantagens

- Responsabilidade do programador de manter endereços
- Programa montador ocupa memória que poderia ser usada na execução do programa de aplicação
- Cada execução demanda nova montagem, mesmo que nenhuma alteração tenha ocorrido

Carregamento absoluto

- Também esquema absoluto
- Separa montagem do carregamento
  - Montador cria módulo objeto em disco
- Carregador transfere módulo objeto do disco para a memória principal
  - Outro programa do sistema
  - Módulo objeto deve conter informação para sua operação

Carregamento absoluto

Conteúdo do módulo objeto

Módulo objeto deve conter informação sobre endereço de memória principal no qual deve ser carregado

- Um registro para cada segmento contíguo de código
- Um outro tipo de registro contém o endereço de início de execução
  - Controle da execução é transferido para esse endereço ao final do carregamento

Carregamento absoluto

Limitações

- Programador é responsável por manter o registro dos endereços
  - Localização na memória de variáveis e subrotinas
- Não há flexibilidade para execução do código
  - Se posição de memória não estiver disponível, programa não pode ser executado

Estratégias de carregamento e ligação

Esquemas relocáveis

- Nesses esquemas, o módulo objeto gerado não tem referências às posições efetivas de memória de variáveis e rotinas, mas relativas a algum endereço-base
- Quando módulo é carregado, as referências relativas precisam ser resolvidas
- Módulo objeto precisa manter informação para realização desses ajustes

Dois tipos de ajustes

relocação: ajuste interno a um módulo

ligação: ajuste entre módulos distintos

Esquemas relocáveis

Informação presente no módulo objeto

Dicionário de Símbolos Externos (ESD): contém todos os símbolos que podem estar envolvidos no processo de resolução de referências entre segmentos: símbolos associados a referências externas (ER), a definições locais (LD) ou a definições de segmentos (SD)

Diretório de Relocação e Ligação (RLD): para cada segmento indica que posições deverão ter seus conteúdos atualizados de acordo com o posicionamento deste e de outros segmentos na memória

Informação para ajustes

Dicionário de Símbolos Externos (ESD)

- Quais símbolos precisam ser conhecidos entre segmentos para resolução das referências
  - ER Referência externa, símbolo que é usado no presente módulo mas definido em outro
  - LD Definição local, símbolo definido no presente módulo que pode ser referenciado em outros

Informação para ajustes

Diretório de Relocação e Ligação (RLD)

- Quais posições de memória no código precisam ser ajustadas
  - em relação aos valores de símbolos externos
  - em relação à posição inicial do próprio segmento
    - Para resolver essa informação, o ESD também deve manter informação sobre a posição de início do segmento
    - SD, definição de segmento

Formato do módulo objeto

Para realizar os ajustes de ligação e relocação, o módulo objeto deve conter registros de quatro tipos:

- **ESD** Informação sobre nome do símbolo, sua posição relativa no segmento e sua dimensão em bytes
- **TXT** Dimensão e o código de máquina do segmento
- **RLD** Posição relativa e dimensão das referências que precisam ser ajustadas no código de máquina
  - Qual valor (símbolo) usado como base para esse ajuste
- **END** Endereço para início da execução
  - Se o módulo contém programa principal

Exemplo de carregador com ajustes

Carregador de ligação direta

Execução em duas etapas:

1. Varredura de todos os módulos que serão ligados para definir
   - Posição de memória na qual o módulo será carregado
   - Definição dos valores dos símbolos externos em uma tabela global de símbolos externos
2. Carregamento do código com realização dos ajustes nas posições indicadas no diretório de relocação e ligação

Carregador de ligação direta

Forma mais simples de operação: em dois passos

1. Resolução dos endereços
   - Aloca espaço em memória
     - Define endereço inicial de memória para cada módulo
   - Lê registros ESD
     - Cria Tabela de Símbolos Externos Globais (GEST)
2. Transferência e ajustes
   - Realiza a transferência dos registros TXT para a memória
   - Altera as posições de memória indicadas nos registros RLD
     - Valores na GEST
   - Inicia execução com endereço obtido em um registro END

Desvantagens

- Código do carregador torna-se mais complexo
- Como carregador compartilha memória com a execução da aplicação, quanto mais simples for, melhor

Estratégia alternativa

Separar as atividades de ligação daquelas do carregamento

- Só o carregador (mais simples) precisa compartilhar a memória com a aplicação

Ligadores

- Programa ligador usado para criar um módulo de carga
  - Combina os módulos objetos — esforço de ligação já realizado
  - Mantém informação sobre ajustes de relocação
    - Relativos a um único símbolo: endereço inicial de memória

Exemplo Programa `ld` do Unix, invocado pelo compilador `gcc` ao final da compilação para criar o código executável

Bibliotecas

- Módulos de código objeto de uso comum num sistema devem ser mantidos para fácil integração ao código da aplicação
- Mecanismo usual para manutenção desses módulos: biblioteca
  - Arquivo cujo conteúdo é um conjunto de outros arquivos (no caso, módulos objetos) com uma tabela para localizar e extrair facilmente esses arquivos

Bibliotecas

Exemplo

Bibliotecas estáticas em Unix

- Criadas e mantidas com aplicativo `ar`
- Módulos necessários são extraídos e integrados ao código pelo aplicativo `ld`
- Bibliotecas a serem utilizadas são indicadas com a chave `-l` na linha de comando

Bibliotecas

Estrutura interna:

Nome	char name[16];
Data de modificação	char modtime[12];
Usuário	char uid[6];
Grupo	char gid[6];
Permissões de acesso	char mode[8];
Dimensão	char size[10];
Marcador de fim	char eol[2];

Limitações dos esquemas estáticos

Esquemas básicos apresentados são estáticos

- Ligadores montam o módulo de carga completo antes da execução
- Uma vez definido o endereço para o carregamento, esse não pode ser alterado até o fim da execução

Desvantagem

Num ambiente que suporta multiprogramação (com execução simultânea de vários programas), vários módulos podem estar repetidos em diferentes aplicações

- Redundância

Carregamento e ligação dinâmicos

Princípio adiar a definição do valor de um símbolo externo até o momento da execução

- Se um módulo já tiver sido carregado para a memória por uma outra aplicação, o mesmo endereço pode ser utilizado

Módulos objetos para ligação dinâmica

- Código dos módulo deve ser reentrante
  - Basicamente, sem variáveis globais ou estáticas
- Códigos mantidos em bibliotecas dinâmicas
  - Sistema Windows: DLL
  - Sistema Linux: ELF (arquivos com extensão `.so`, de *shared objects*
- Módulo de carga (o programa executável) é um módulo primário
  - Referências aos módulos externos sem resolução

Estratégias para resolução

Duas possibilidades:

1. Referências resolvidas quando módulo primário é carregado
2. Referências resolvidas quando símbolo é referenciado

Resolução em tempo de carregamento

- Quando o módulo de carga primário é carregado, todas as referências a módulos externos devem ser resolvidas
- Se módulo já está na memória, mesmo endereço é utilizado
- Caso contrário, módulo é carregado e seu endereço é definido
- Se algum módulo não for localizado, execução é abortada

Esquema utilizado no Sistema Windows

DLL *Dynamic Link Library*

Resolução em tempo de execução

- Execução inicia com o módulo de carga primário sem resolução das referências externas
- Apenas quando a referência é efetivamente necessária o sistema faz a resolução
  - Se módulo já está na memória, mesmo endereço é utilizado
  - Caso contrário, módulo é carregado e seu endereço é definido
- Se referência não for utilizada, módulo correspondente nem precisa ser carregado
  - Eventualmente, aplicação pode executar sem a instalação da correspondente biblioteca dinâmica

Esquema utilizado no Sistema Linux

ELF Executable and Linkable Format

Manipulação de símbolos em ELF

dlopen	abre o arquivo da biblioteca dinâmica
dlsym	obtém endereço do símbolo especificado
dclose	fecha a biblioteca dinâmica
dlerror	emite mensagem de diagnóstico de erro

Exemplo (sem tratamento de erros)

```
#include <dlfcn.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  void *handle;
  double (*cosine)(double);
  char *error;
  handle = dlopen("/lib/libm.so.5", RTLD_LAZY);
  cosine = dlsym(handle, "cos");
  printf("%f\n", (*cosine)(2.0));
  dlclose(handle);
}
```

O problema da dependência

Cadeia de dependências A aplicação depende da biblioteca `A`, que depende da `B`, que depende...

- Em sistemas com muitas aplicações instaladas (e desinstaladas), é possível que aplicações parem de funcionar por conta de ausência de uma biblioteca dinâmica removida inadvertidamente ou por incompatibilidade de versões
  - Problema conhecido como *DLL hell*
  - Agravado por modificações locais não documentadas e pela ausência de um padrão de numeração de versões

O problema da dependência

Soluções

- Manutenção de um controle de versões, com indicação de *major version* and *minor version*
- Atualizações de aplicações controladas por meio de um gerenciador de pacotes

O panorama geral