Nivel de Linguagem de Montagem (Assembly)

Orlando Loques
setembro 2006

Referências:

- Structured Computer Organization (capítulo 7), A.S. Tanenbaum, (c) 2006 Pearson Education Inc
- Computer Organization and Architecture, W. Stallings, Prentice Hall

Motivação (i)

- Tradução X interpretação; Java usa dois níveis:
  - O compilador traduz Java para bytecode
  - A máquina virtual interpreta o bytecode
- Tradutores:
  - Compilador: Java, C, Fortran
  - Montador/Assembler – representação simbólica da linguagem da máquina; inclui instruções e endereços; cada instrução corresponde exatamente a uma instrução do nível ISA
- Resultado da Tradução da Linguagem de Montagem:
  - Programa Objeto / Programa Binário Executável

Motivação (ii)

- Programar em Assembler é mais difícil e toma mais tempo que usando linguagem de alto-nível
- Por quê usar Assembler ?
  - Desempenho
    - Expert pode produzir código mais rápido e menor; essencial em aplicações embutidas, etc, eg Smartcards
  - Acesso a detalhes
    - Interrupção, trap handlers, E/S; gerenciamento de memória virtual

Motivação (iii)

Comparação de linguagem de montagem e linguagem de alto nível, considerando um ajuste fino (tuning)

	Programmer-years to produce the program	Program execution time in seconds
Assembly language	50	33
High-level language	10	100
Mixed approach before tuning		
Critical 10%	1	90
Other 90%	9	10
Total	10	100
Mixed approach after tuning		
Critical 10%	6	30
Other 90%	9	10
Total	15	40

Motivação (iv)

1. O sucesso do projeto pode depender do desempenho
2. Algumas aplicações têm limitações de memória
3. Acesso a detalhes de baixo nível
4. Projeto de compiladores
5. Estudo de arquitetura de computadores

   …
6. Etc

Formatos de Comandos (i)

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J
I	DD	3	; reserve 4 bytes initialized to 3
J	DD	4	; reserve 4 bytes initialized to 4
N	DD	0	; reserve 4 bytes initialized to 0

(a)

Computação de $N = I + J$ - Pentium 4

Formato - MASM (microsoft macro assembler)

Campos:

- Rótulo / Label: representam posições/endereços na memória
- Opcode
- Operandos
- Comentários

Formatos de Comandos (ii)

Label	Opcode	Operands	Comments
FORMULA	MOVE.L	I, D0	; register D0 = I
	ADD.L	J, D0	; register D0 = I + J
	MOVE.L	D0, N	; N = I + J
I	DC.L	3	; reserve 4 bytes initialized to 3
J	DC.L	4	; reserve 4 bytes initialized to 4
N	DC.L	0	; reserve 4 bytes initialized to 0

(b)

Computação de $N = I + J$ - Motorola 680x0.

Formatos de Comandos (iii)

Computação de $N = I + J$ - SPARC.

Label	Opcode	Operands	Comments
FORMULA:	SETHI	%HI(I),%R1	! R1 = high-order bits of the address of I
	LD	[%R1+%LO(I)],%R1	! R1 = I
	SETHI	%HI(J),%R2	! R2 = high-order bits of the address of J
	LD	[%R2+%LO(J)],%R2	! R2 = J
	NOP		! wait for J to arrive from memory
	ADD	%R1,%R2,%R2	! R2 = R1 + R2
	SETHI	%HI(N),%R1	! R1 = high-order bits of the address of N
	ST	%R2,[%R1+%LO(N)]	
I:	.WORD 3		! reserve 4 bytes initialized to 3
J:	.WORD 4		! reserve 4 bytes initialized to 4
N:	.WORD 0		! reserve 4 bytes initialized to 0

(c)

Pseudo-Instruções (i)

Pseudoinstruction	Meaning
SEGMENT	Start a new segment (text, data, etc.) with certain attributes
ENDS	End the current segment
ALIGN	Control the alignment of the next instruction or data
EQU	Define a new symbol equal to a given expression
DB	Allocate storage for one or more (initialized) bytes
DW	Allocate storage for one or more (initialized) 16-bit (word) data items
DD	Allocate storage for one or more (initialized) 32-bit (double) data items
DQ	Allocate storage for one or more (initialized) 64-bit (quad) data items
PROC	Start a procedure
ENDP	End a procedure
MACRO	Start a macro definition

Pseudo-instruções (comandos para o próprio assembler) disponíveis no assembler do Pentium 4 (MASM)

Pseudo-Instruções (ii)

Pseudo-instruções disponíveis no assembler do Pentium 4 (MASM)

Pseudoinstruction	Meaning
ENDM	End a macro definition
PUBLIC *	Export a name defined in this module
EXTERN *	Import a name from another module
INCLUDE	Fetch and include another file
IF	Start conditional assembly based on a given expression
ELSE	Start conditional assembly if the IF condition above was false
ENDIF	End conditional assembly
COMMENT	Define a new start-of-comment character
PAGE	Generate a page break in the listing
END	Terminate the assembly program

Pseudo-Instruções (iii)

- BASE EQU 1000; o nome BASE pode ser usado em todo lugar ao invés de 1000
- LIMIT EQU 4*BASE + 2000; pode ser definido através de expressões
- DB, DW, DD, DQ alocam memória para variáveis
  - TABLE DB 11, 23, 49 nomeia e inicializa 3 bytes

Definição de Macro, Chamada e Expansão (i)

- Representam trechos de código que têm que ser repetidos dentro do programa
- Codigo para intercambiar P e Q duas vezes

```
MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX

MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX
```

(a) sem macro

```
SWAP

SWAP
```

(b) com macro

```
SWAP    MACRO
        MOV EAX,P
        MOV EBX,Q
        MOV Q,EAX
        MOV P,EBX
        ENDM
```

definição da macro

Definição de Macro, Chamada e Expansão (i)

Comparação de “chamadas” macros com chamadas de procedimentos

Item	Macro call	Procedure call
When is the call made?	During assembly	During program execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	One

Macros com Parâmetros

```
MOV     EAX,P                 CHANGE    MACRO P1, P2
MOV     EBX,Q                           MOV EAX,P1
MOV     Q,EAX                           MOV EBX,P2
MOV     P,EBX                           MOV P2,EAX
                                        MOV P1,EBX
MOV     EAX,R                           ENDM
MOV     EBX,S
MOV     S,EAX                           CHANGE P, Q
MOV     R,EBX
                                        CHANGE R, S
```

(a) sem macro

(b) com macro

Montagem Condicional

```
M1      MACRO
        IF WORDSIZE GT 16
M2              MACRO
                  •••
                ENDM
        ELSE
M2              MACRO
                  •••
                ENDM
        ENDIF
ENDM
```

Assemblers de dois Passos (i)

- Quais são os endereços usados nos labels (desvios e referências)?
  - Forward reference problem
- No primeiro passo os símbolos são coletados na Tabela de Símbolos; as macros também são expandidas, três tabelas são usadas:
  - Tabela de símbolos
  - Tabela de Códigos de Operação
  - Tabela de Pseudo-instruções

Assemblers de dois Passos (i)

Label	Opcode	Operands	Comments	Length	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	branch to DONE	5	129

Instruction location counter (ILC): mantém o controle dos endereços onde as instruções serão carregadas na memória. Neste exemplo, os comandos antes de MARIA ocupam 100 bytes.

Assemblers de dois Passos (ii)

Tabela de Símbolos para o programa anterior

- Campo Outras Informações:
  - Tamanho do dado associado ao campo
  - Visibilidade do simbolo
  - Bits de realocação – se necessário

Symbol	Value	Other information
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

Assemblers de dois Passos (iii)

Tabela de códigos de operação (parcial) de um montador para o Pentium 4

Opcode	First operand	Second operand	Hexadecimal opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

A classe da instrução define a tradução da instrução, cada classe sendo associada a procedimentos específicos de tradução do montador

Passo I (i)

```
public static void pass_one( ) {
   // This procedure is an outline of pass one of a simple assembler.
   boolean more_input = true;                  // flag that stops pass one
   String line, symbol, literal, opcode;       // fields of the instruction
   int location_counter, length, value, type;  // misc. variables
   final int END_STATEMENT = –2;               // signals end of input

   location_counter = 0;                       // assemble first instruction at 0
   initialize_tables( );                       // general initialization

   while (more_input) {                        // more_input set to false by END
      line = read_next_line( );                // get a line of input
      length = 0;                              // # bytes in the instruction
      type = 0;                                // which type (format) is the instruction
```

Passo I (ii)

• • •

```
if (line_is_not_comment(line)) {
    symbol = check_for_symbol(line);        // is this line labeled?
    if (symbol != null)                     // if it is, record symbol and value
        enter_new_symbol(symbol, location_counter);
    literal = check_for_literal(line);      // does line contain a literal?
    if (literal != null)                    // if it does, enter it in table
        enter_new_literal(literal);

    // Now determine the opcode type.  –1 means illegal opcode.
    opcode = extract_opcode(line);          // locate opcode mnemonic
    type = search_opcode_table(opcode);     // find format, e.g. OP REG1,REG2
    if (type < 0)                           // if not an opcode, is it a pseudoinstruction?
        type = search_pseudo_table(opcode);
    switch(type) {                          // determine the length of this instruction
        case 1: length = get_length_of_type1(line);  break;
        case 2: length = get_length_of_type2(line);  break;
        // other cases here
    }
}
```

• • •

Passo I (iii)

• • •

```
        write_temp_file(type, opcode, length, line);    // useful info for pass two
        location_counter = location_counter + length;       // update loc_ctr
        if (type == END_STATEMENT) {            // are we done with input?
            more_input = false;                 // if so, perform housekeeping tasks
            rewind_temp_for_pass_two( );        // like rewinding the temp file
            sort_literal_table( );              // and sorting the literal table
            remove_redundant_literals( );       // and removing duplicates from it
        }
    }
}
```

Passo II (i)

```
public static void pass_two( )  {
    // This procedure is an outline of pass two of a simple assembler.
    boolean more_input = true;                    // flag that stops pass two
    String line, opcode;                          // fields of the instruction
    int location_counter, length, type;           // misc. variables
    final int END_STATEMENT = –2;                 // signals end of input
    final int MAX_CODE = 16;                      // max bytes of code per instruction
    byte code[ ] = new byte[MAX_CODE];            // holds generated code per instruction

    location_counter = 0;                         // assemble first instruction at 0

    while (more_input) {                          // more_input set to false by END
        type = read_type( );                      // get type field of next line
        opcode = read_opcode( );                  // get opcode field of next line
        length = read_length( );                  // get length field of next line
        line = read_line( );                      // get the actual line of input
```

• • •

Passo II (i)

- Geração do programa objeto e da listagem

- Geração de informação para o Linker
  - Usado na composição de programas a partir de módulos

- Identificação de Erros

Passo II (ii)

. . .

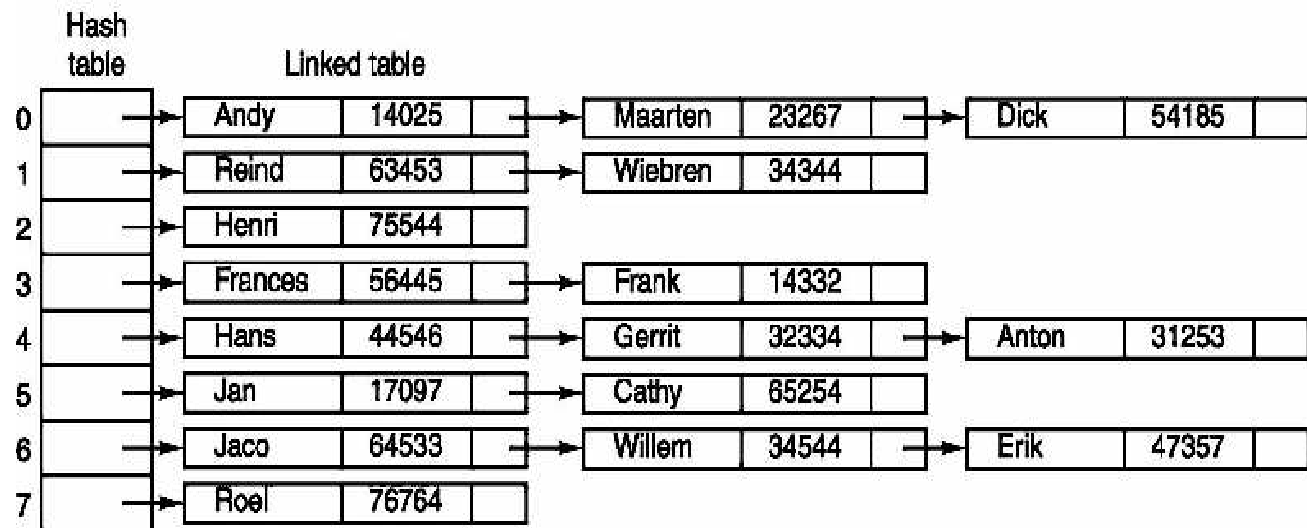
```
        if (type != 0) {                          // type 0 is for comment lines
            switch(type) {                        // generate the output code
                case 1: eval_type1(opcode, length, line, code);  break;
                case 2: eval_type2(opcode, length, line, code);  break;
                // other cases here
            }
        }

        write_output(code);                       // write the binary code
        write_listing(code, line);                // print one line on the listing
        location_counter = location_counter + length;      // update loc_ctr
        if (type == END_STATEMENT) {              // are we done with input?
            more_input = false;                   // if so, perform housekeeping tasks
            finish_up( );                         // odds and ends
        }
    }
}
```

Tabela de Simbolos - Codificação Hash

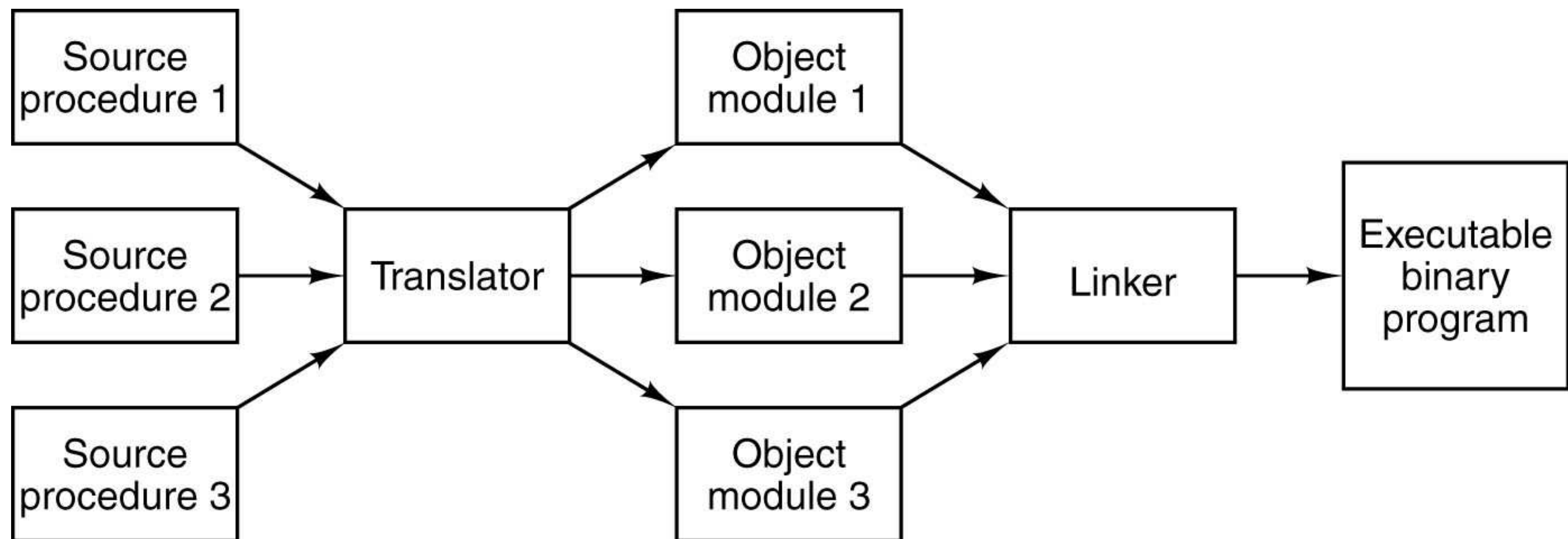
(a)símbolos, valores e códigos de hash derivados dos símbolos
(b) tabela hash, com lista

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebren	34344	1



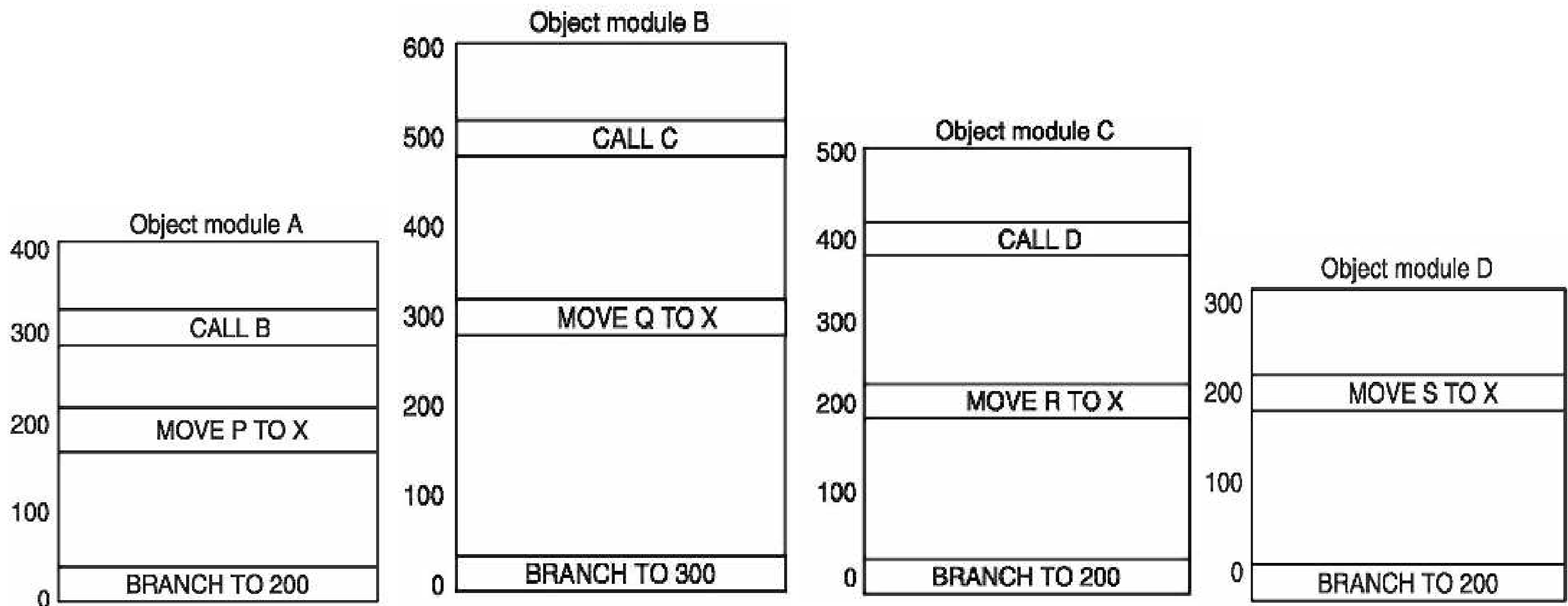
Ligação e Carregamento (Linking and Loading)

A geração de um programa binário executável, a partir de um conjunto de programas traduzidos independentemente, requer o uso de um ligador (loader)

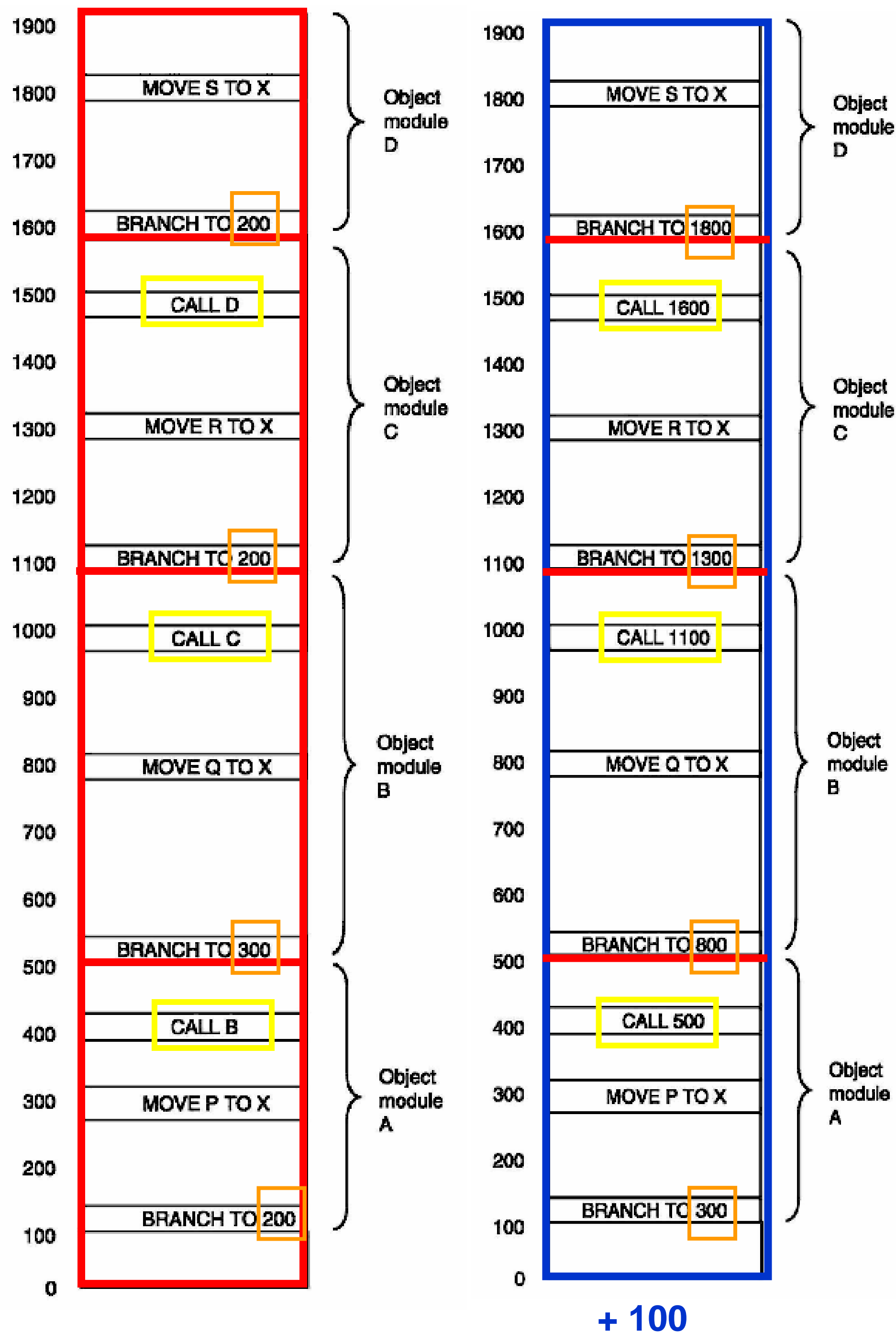


Tarefas realizadas pelo Linker (i)

Cada módulo tem o seu espaço de endereçamento próprio começando no endereço 0



Tarefas realizadas pelo Linker (ii)



- Endereço inicial de montagem = 100
- Módulos objetos após posicionados na imagem binária
- Binário executável: Módulos objetos depois da ligação e realocação, considerando a base 100

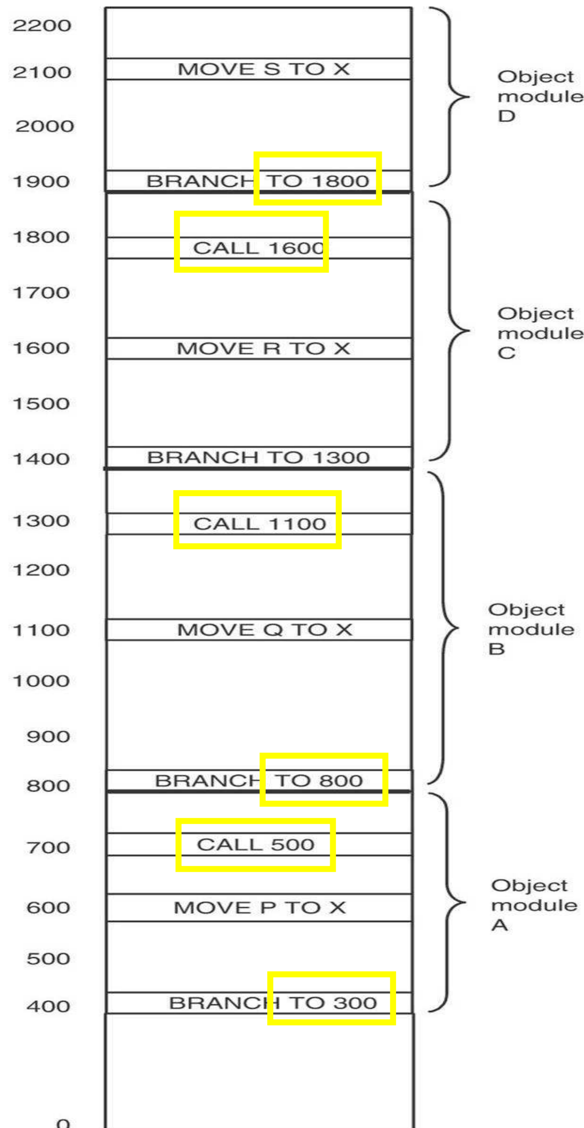
Módulo	Tamanho	Endereço Inicial
A	400	100
B	600	500
C	500	1100
D	300	1600

Estrutura de um Módulo Objeto

End of module
Relocation dictionary
Machine instructions and constants
External reference table
Entry point table
Identification

- Estrutura interna de Módulo Objeto produzida pelo tradutor
  - Identificação: nome, tamanho das diversas partes do módulo
  - Entry point table: símbolos que podem ser referenciados externamente
  - External reference table: símbolos referenciados em outros módulos
  - Código e constantes
  - Relocation Dictionary: especifica os endereços que tem que ser re-alocados; tabela ou bit map
  - End of module: check sum, endereço de início do programa

Instante de Ligação e Relocação Dinâmica

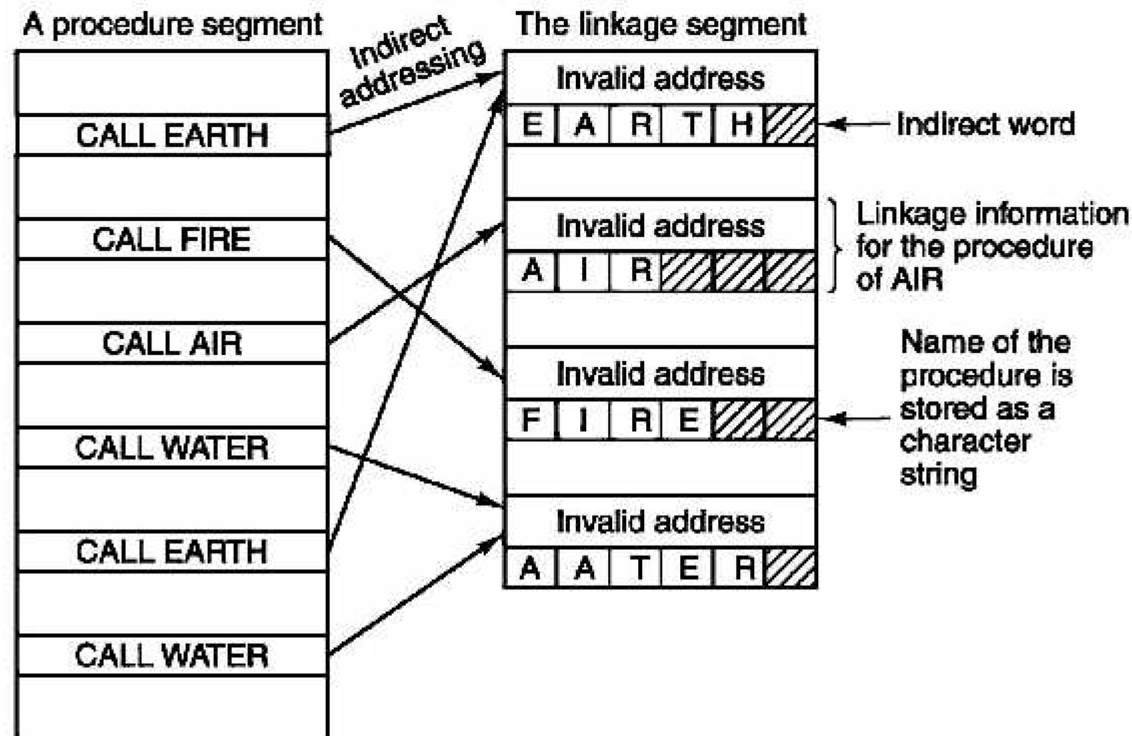


- O programa realocado anteriormente movido 300 endereços acima
- Diversas instruções usam referências a endereços incorretos
- A ligação dinâmica ajuda a resolver o problema

Instante de Ligação

- Quando o programa é escrito
- Quando o programa é traduzido
- Quando o programa é “ligado”, mas antes de ser carregado
- Quando o programa é carregado
- Quando um registro de base usado para endereçamento é carregado
- Quando a instrução contendo o endereço é carregada

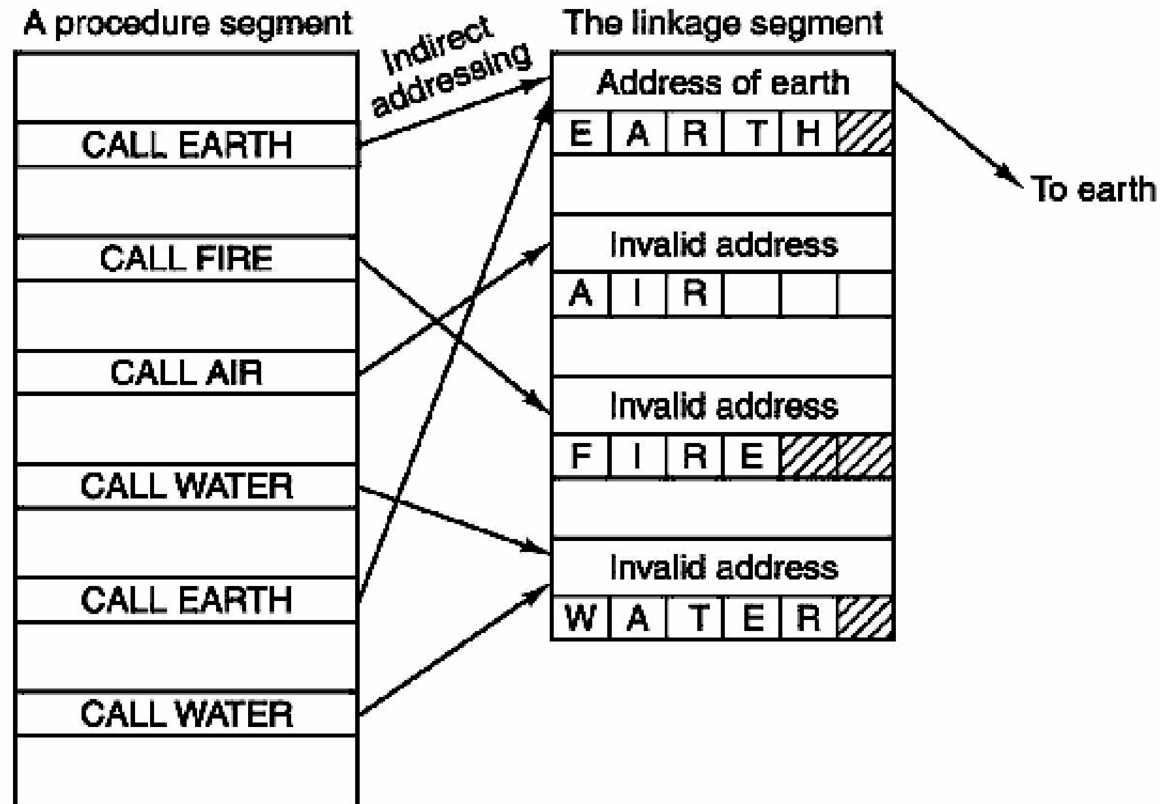
Ligação Dinâmica no MULTICS (i)



Antes da chamada a EARTH

- Os endereços estão indefinidos

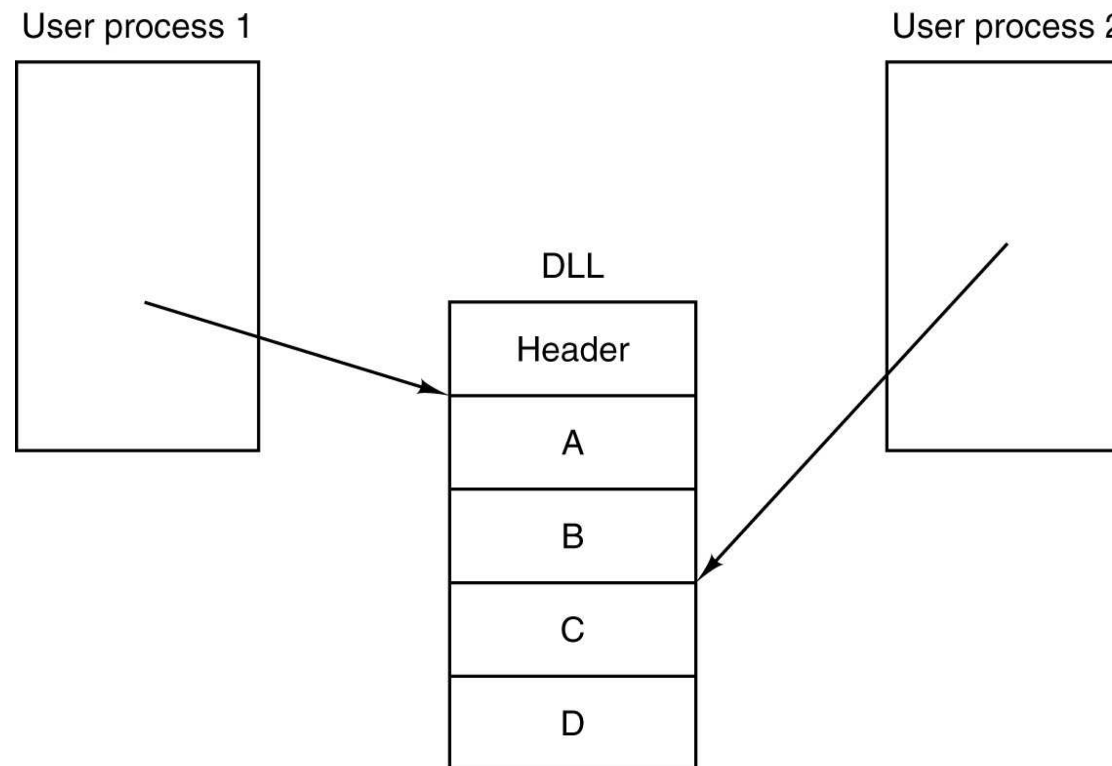
Ligação Dinâmica no MULTICS (ii)



Depois da chamada e ligação de EARTH

O endereço é atualizado na tabela

Ligação Dinâmica no Windows



- **Uso de uma DLL por dois processos**
- **DLL: Dynamic Link Library**
  - **uma coleção de procedimentos que pode ser carregada na memória**
  - **Servem para definir interfaces de módulos: jogos, bibliotecas, drives, etc**