7

THE ASSEMBLY LANGUAGE LEVEL

	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

Figure 7-1. Comparison of assembly language and high-level language programming, with and without tuning.

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	DW	3	; reserve 4 bytes initialized to 3
J	DW	4	; reserve 4 bytes initialized to 4
N	DW	0	; reserve 4 bytes initialized to 0

(a)

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)

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)

Figure 7-2. Computation of $N = I + J$. (a) Pentium II. (b) Motorola 680x0. (c) SPARC.

Pseudoinstr	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
DD	Allocate storage for one or more (initialized) 16-bit halfwords
DW	Allocate storage for one or more (initialized) 32-bit words
DQ	Allocate storage for one or more (initialized) 64-bit double words
PROC	Start a procedure
ENDP	End a procedure
MACRO	Start a macro definition
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

Figure 7-3. Some of the pseudoinstructions available in the Pentium II assembler (MASM).

(a)			(b)	
MOV	EAX,P		SWAP	MACRO
MOV	EBX,Q			MOV EAX,P
MOV	Q,EAX			MOV EBX,Q
MOV	P,EBX			MOV Q,EAX
				MOV P,EBX
MOV	EAX,P			ENDM
MOV	EBX,Q			
MOV	Q,EAX			SWAP
MOV	P,EBX			
				SWAP

Figure 7-4. Assembly language code for interchanging P and Q twice. (a) Without a macro. (b) With a macro.

Item	Macro call	Procedure call
When is the call made?	During assembly	During 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	1

Figure 7-5. Comparison of macro calls with procedure calls.

```
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)                         (b)
```

Figure 7-6. Nearly identical sequences of statements. (a) Without a macro. (b) With a macro.

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

Figure 7-7. The instruction location counter (ILC) keeps track of the address where the instructions will be loaded in memory. In this example, the statements prior to MARIA occupy 100 bytes.

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

Figure 7-8. A symbol table for the program of Fig. 7-7.

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

Figure 7-9. A few excerpts from the opcode table for a Pentium II assembler.

```
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

        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
            }
        }

        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
        }
    }
}
```

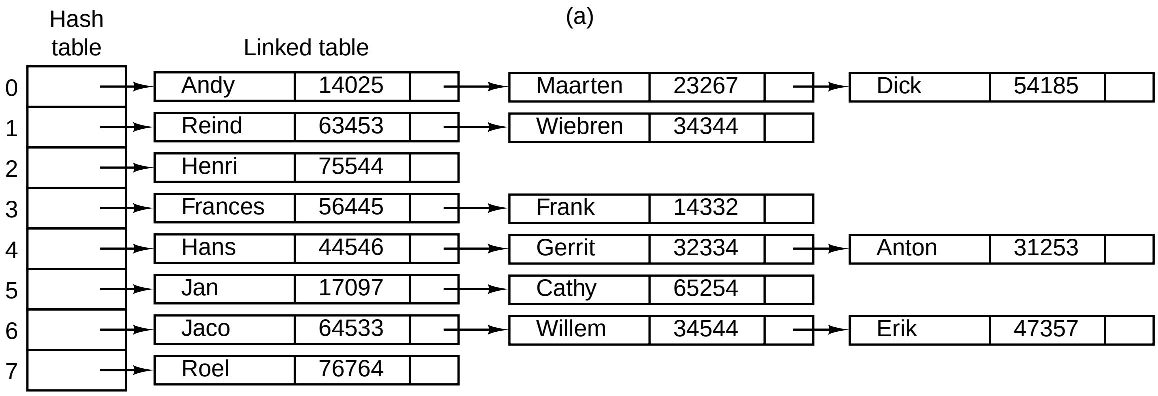
Figure 7-10. Pass one of a simple assembler.

```
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 one
    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

        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
        }
    }
}
```

Figure 7-11. Pass two of a simple assembler.

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

(a)



(b)

Figure 7-12. Hash coding. (a) Symbols, values, and the hash codes derived from the symbols. (b) Eight-entry hash table with linked lists of symbols and values.

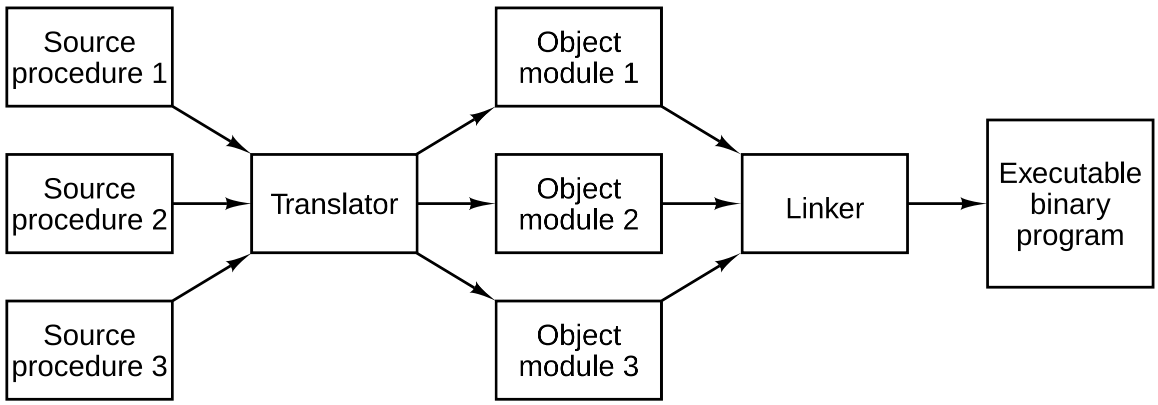


Figure 7-13. Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.

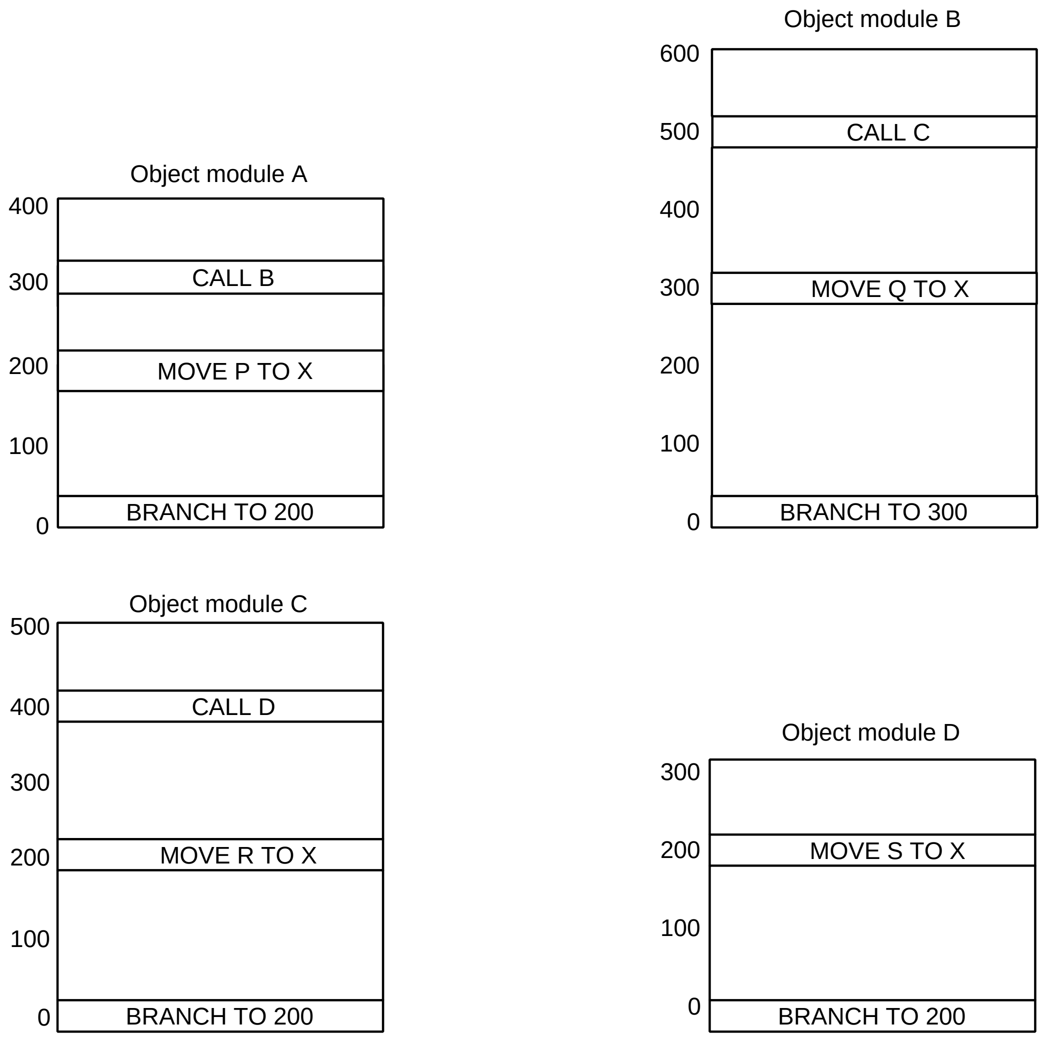


Figure 7-14. Each module has its own address space, starting at 0.

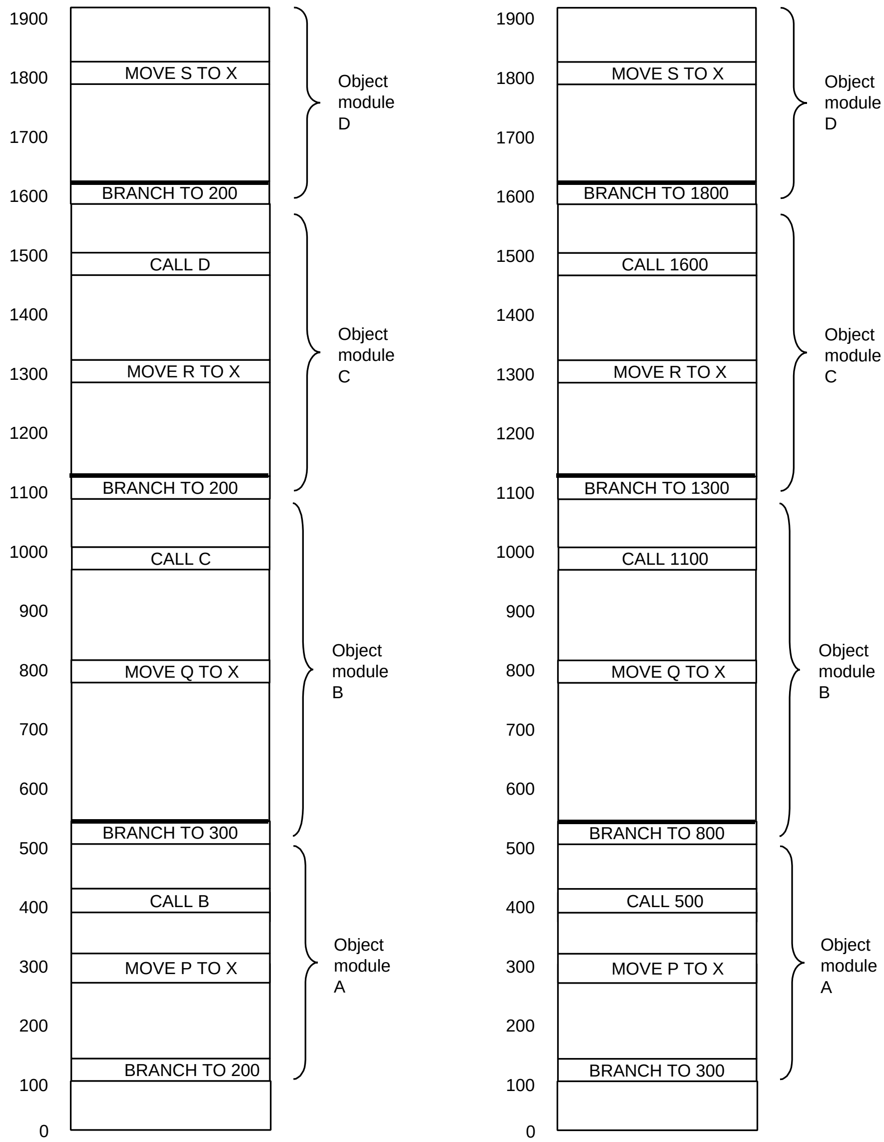


Figure 7-15. (a) The object modules of Fig. 7-14 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed. Together they form an executable binary program, ready to run.

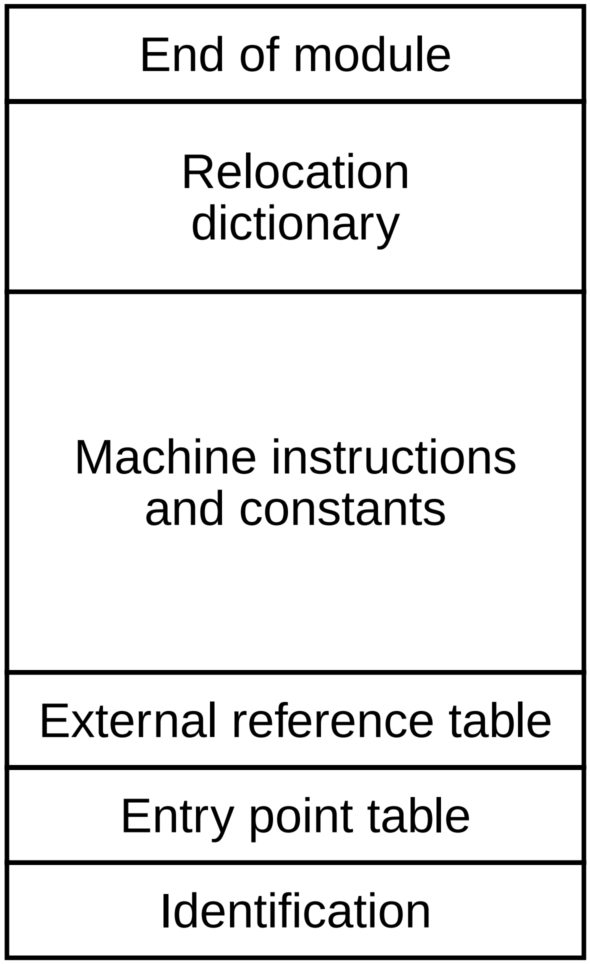


Figure 7-16. The internal structure of an object module produced by a translator.

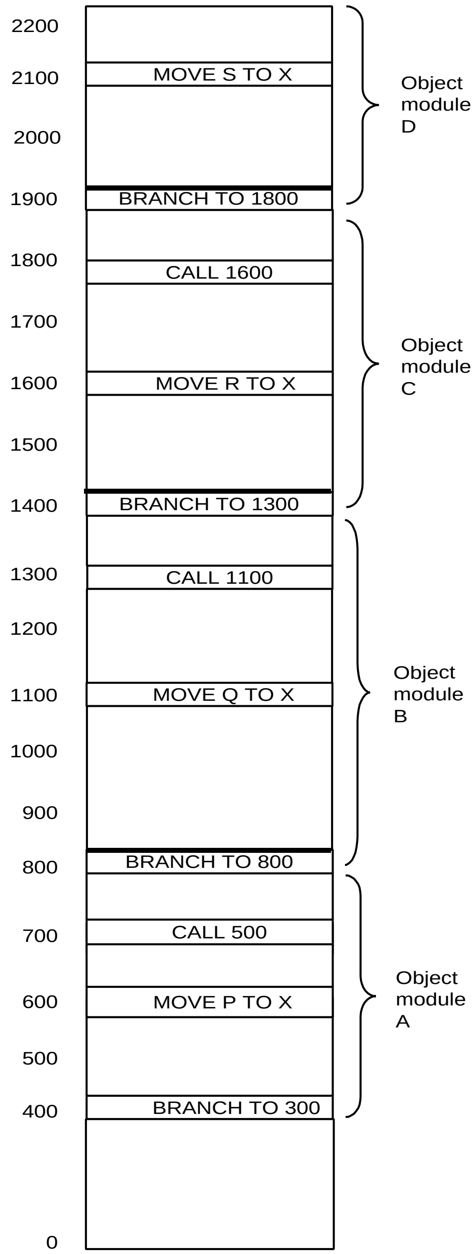


Figure 7-17. The relocated binary program of Fig. 7-15(b) moved up 300 addresses. Many instructions now refer to an incorrect memory address.

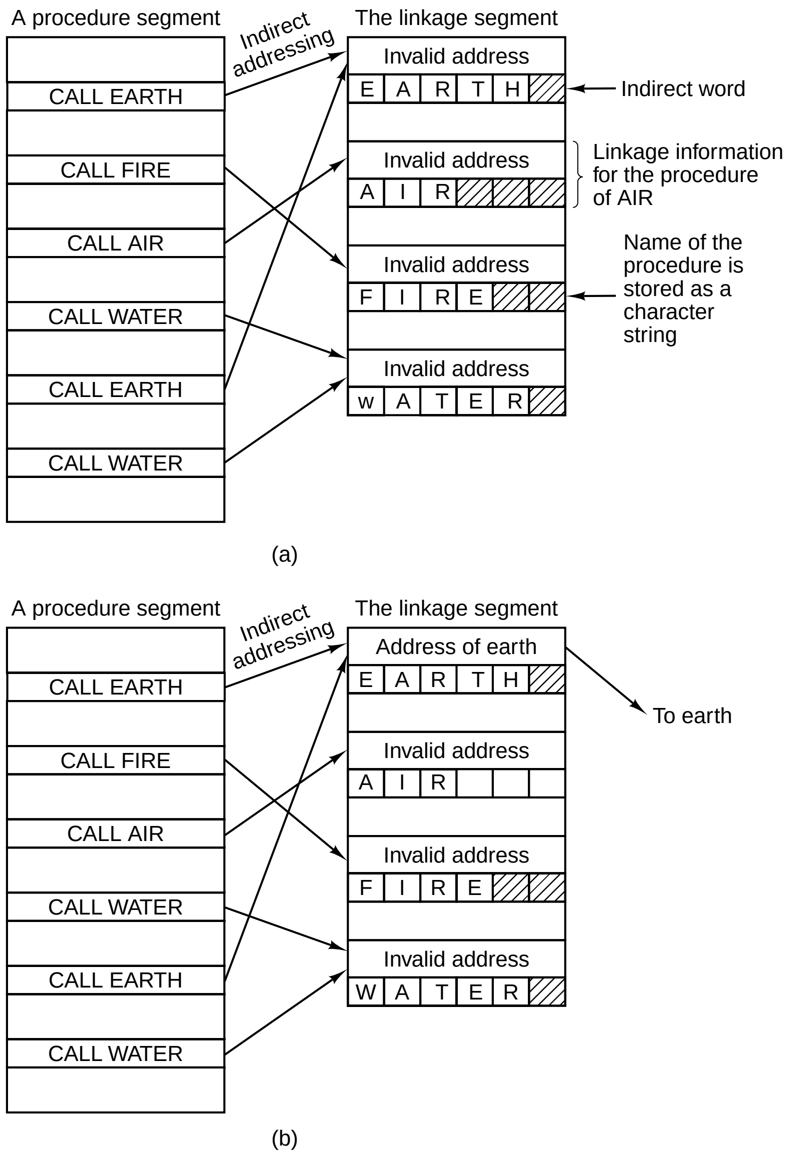


Figure 7-18. Dynamic linking. (a) Before *EARTH* is called. (b) After *EARTH* has been called and linked.

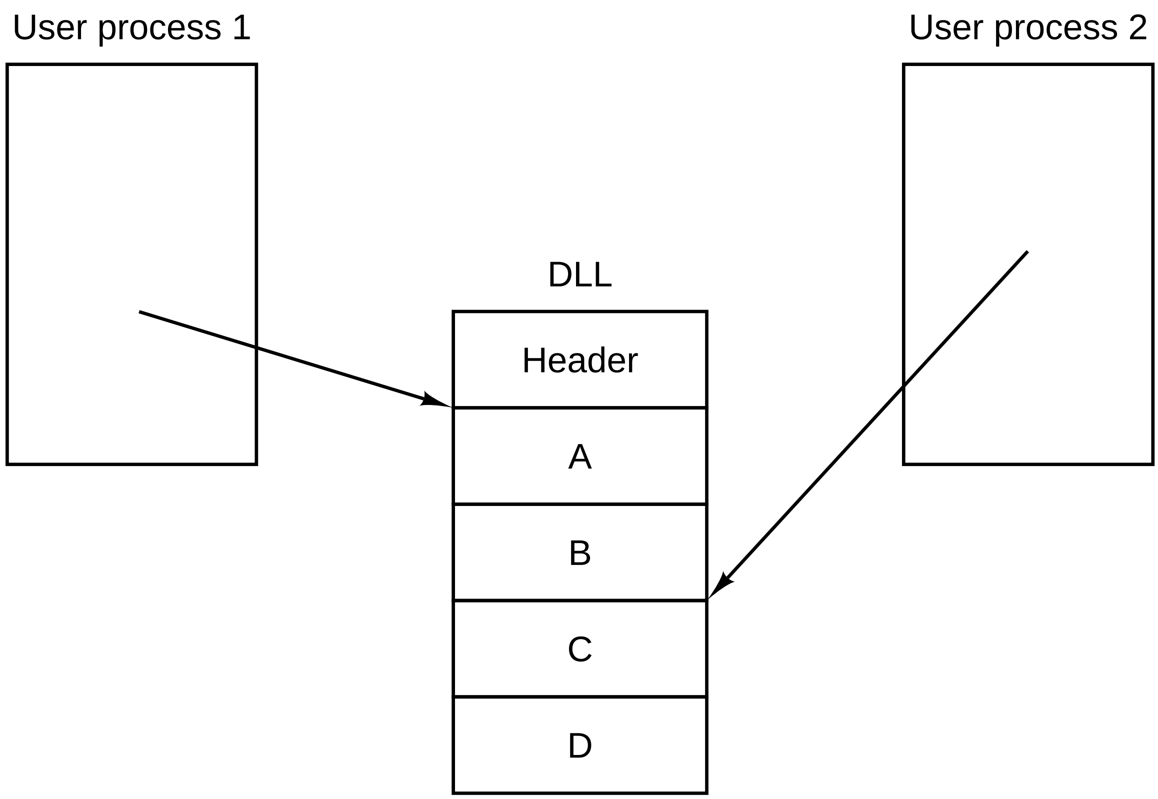


Figure 7-19. Use of a DLL file by two processes.