Chapter 8

PREPARATION FOR EXECUTION

8.1 FUNCTIONS

An object-code segment produced by a compiler or by a module assembler cannot be executed without modification. As many as five further functions must be performed first. Their logical sequence is the following. (1) If the segment is one of several which have been translated independently but are to be executed together, there will be symbols in one segment which refer to entities defined in another. Establishment of the proper correspondences and use of the correct attributes is called *resolution* of such intersegment symbolic references. (2) A symbol not resolved by comparison with the other segments may be a reference to the execution-time library of system-provided service routines and user-written subroutines. Means are required for managing the library. (3) Storage must be allocated for each segment's private needs, for any area accessed jointly, and for the required library routines. (4) Address fields must be relocated to match the allocation. (5) The segments must be loaded into the allocated storage. These five functions will now be examined, although in a slightly different order.

8.1.1 *Loading*

A translator which generates machine code in the locations from which it is to be executed, such as a load-and-go compiler or a load-and-go assembler, performs its own loading function. No separate loader is required. This organization requires the program to be translated each time it is used. Moreover, because there is no provision for modifying the program after translation, it is essentially limited to a single segment. Furthermore, storage space is required not only for the program but for the translator as well. Nevertheless, small programs which are frequently retranslated can be handled conveniently by such a translator.

Two situations exist in which the segment to be loaded is in absolute form ready for execution, but does not occupy the storage locations to which it is destined. One arises when the translator, usually an assembler, produces *absolute* machine code, in which all addresses are valid only if the segment is placed in specified locations. The other arises after a program has been copied to backing storage, from which it is later read in to resume execution. An *absolute loader* (or "binary loader") is used to load a machine-code segment. It need only place the code in the specified locations, perhaps checking the accuracy of the information transferred, and then branch to the starting location. Because this function is performed just prior to execution and does not involve translation, such a loader may quite properly be viewed as a control program rather than a translator.

8.1.2 *Relocation*

Most translators do not bind the locations which must be occupied by the code which they generate. Instead, they produce one or more relocatable segments, identifying in each which address fields are absolute and which relative. If multiple location counters are used, the origin to which each relative address field is relative must also be identified. The task of relocation is to add the applicable relocation constant (address of the segment origin) to each relative address in the segment. Different methods of performing relocation apply to machines with direct addressing and with base addressing. Although relocation can be performed prior to loading, this may bind the execution-time location before storage is allocated. Consequently, relocation is usually performed in conjunction with loading, by a program called a *relocating loader*, which is in part a translator. The actual adjustment of an address field often occurs after the field has been loaded, rather than before. It should be noted that the term "relocation" refers to adjustment of address fields and not to movement of a program segment. Movement of a segment requires relocation, but relocation does not require movement.

8.1.3 *Resolution of External References*

If a segment is to be executed in combination with others which have been translated separately, the translator prepares for each segment a list of symbols to be resolved. This list includes (1) internal references to externally defined symbols and (2) internally defined symbols which may be referenced externally. The two types of list entries may or may not be segregated. The resolution is usually effected by a program called a *linker*, which performs the following functions. It establishes the correspondences between symbol references and definitions, determines the relocatability attribute of symbols and address expressions, and substitutes for them addresses in the form required by the loader. It typically assumes that symbols still undefined are names of library routines. In many systems resolution is combined with loading (and usually relocation) in a *linking loader*. If intersegment references are limited to calls of other segments, whether library routines or not, resolution can be accomplished more simply than with a linker, by use of a *transfer vector* of collected branch instructions to the different segments.

8.1.4 *Library Management*

Strictly speaking, management of the library of routines available at execution time is not a function of translation, but performing it does affect several aspects of translation. The determination of which library routines are needed and of how much storage to allocate for them is an example. Although library routines used to be in absolute code, it is almost universal practice now to provide them in relocatable form. The size of the library can often be reduced by replacing similar portions of two or more routines with another routine, which is then called by the routines which it serves. In that event, execution-time calls will be generated not only to the routines named in the user's program but to yet other routines as well. If an index of calls made by library routines is available to the linker, it can determine the entire set of routines which will be needed, and link the references accordingly. If the library is on a linear medium such as paper or magnetic tape, it may be wise to have the copy of each calling routine precede copies of routines which it calls, even at the cost of space for duplicate copies. Then the necessary routines can be loaded in a single pass even if an index of calls is not constructed beforehand, provided that each routine names the others which it needs. If the library routines are on a rotating medium such as disk or drum, a dictionary of their placement should be maintained.

8.1.5 *Storage Allocation*

Although storage allocation within each segment is performed by the translator which generates the segment, there remains the problem of ensuring that all the pieces will fit at execution time. Space is required for each of the segments, for the largest of the common data areas used for intersegment communication, and for the library routines which may be called. The linker is the first program in a position to determine the total storage requirement, but it may lack information concerning space needed for library routines. If calls by one library routine to another are not discovered until the former is loaded, space information is not available until loading time. This provides one motive for deferring linking until loading. A more important motive for combining linking with loading is to reduce the total number of passes over the text. If any of the required library routines is to be generated at execution time rather than merely selected, even loading time is too early to specify the exact size, although it may be possible to determine an upper bound.

If more space is required than will be available, different segments or routines whose presence is not required simultaneously can share storage sequentially, each using a given portion of space in turn. This mode of storage use is called *overlaying*. If the writer of the program can predict in what sequence different program segments will be required, the segments can be replaced dynamically in a predetermined sequence. This *dynamic loading* is programmed in most systems by the user. Some linkers, however, can accept a specification of what dynamic loading is to be performed and generate the necessary execution-time calls to the loader. Although the loading is dynamic, the overlay structure is static, being unchanged from one execution of the program to the next. Sometimes, however, the selection of required segments cannot be determined prior to execution, because it is data-dependent. Such dynamic overlay structures are characteristic of transaction-oriented interactive systems, such as airline reservations or banking. The nature of the transaction dictates which processing programs are required, and they are loaded as appropriate and linked with the using programs. Because the translators are no longer available, the control program is called upon to perform this *dynamic linking*.

8.2 IMPLEMENTATIONS

8.2.1 *Absolute Loader*

The task of an absolute loader is virtually trivial. It reads a stream of bits from a specified source into specified storage locations and transfers control to a designated address. Loader input prepared by an absolute assembler is in the form of records, often card images, each containing the text (instructions, data, or both) to be loaded, the length of the text, and the starting address. The loader reads the record into a fixed location, examines the starting address and length, and copies the text as specified. The last text record is followed by a branch record, which contains the address to which the loader branches. The three main storage accesses required for the text (one for reading, two for copying) can be reduced to one by first reading only the length and starting address. The loader then reads the text directly into its final location.

An absolute loader provided for reloading programs which have been rolled out will surely be designed to effect the foregoing saving. The length and starting address of the entire program will have been recorded at the beginning of the program. If the program length exceeds the amount which can be fetched in a single read operation, the loader can fetch the program in pieces, incrementing the starting address and decrementing the remaining length as it proceeds.

8.2.2 *Relocating Loader*

A translator may generate object code intended to be executed when resident in storage locations whose addresses will be specified only after translation. Address references made by the object code to main storage locations which it will occupy after loading must be adjusted by the addition of a relocation constant before execution. Instruction and data fields which require this adjustment are termed *relative*; those which do not, *absolute*. Within an instruction, the operation code is absolute, as are register addresses. Operand fields may or may not be absolute. Immediate data and shift amounts are absolute; many address fields, although not all, are relative. An address specified numerically in the source program is absolute, but a symbol defined by its appearance in a label field of the same segment is relative. So is a literal, and so is the redefinable current line symbol *. Constants generated by the translator are absolute, except for address constants. The value of an address constant, typically written in source language as A(PLACE), is defined to be the execution-time storage address corresponding to the symbol named. Clearly such an address is relative.

The relocatability attribute of an address expression depends upon those of its components. If an address expression is limited to an algebraic sum of signed symbols (i.e. no products), such as PLACE+7 or HERE−LOOP +STAND, then its relocatability attribute is easily determined. Ignoring each absolute component and replacing each relative component by R reduces the address expression to the form nR, where n is a signed integer. If $n=1$, the expression is relative; if $n=0$, the expression is absolute. Thus if HERE and LOOP are relative and STAND is absolute, then we ignore STAND and replace HERE by R and −LOOP by −R. This yields zero and the address expression is found to be absolute.

If n has any other value than 0 or 1, the expression is ill-formed. This can be seen most readily from an example. The execution-time location A+B, where A and B are relative symbols, cannot be fixed even relative to the origin. Suppose A and B have location counter values 20 and 30, respectively. If the origin is 100, then the execution-time locations A and B are

120 and 130, and A+B is 250, or 150 beyond the origin. But if the origin is 300, then the execution-time location A+B is 650, or 350 beyond the origin. The address expression A+B does not have a reproducible meaning. Similar problems arise if *n* is greater than 2, or is negative.

If the relocatability attribute of each symbol is known to the translator, as it is in the absence of internal references to externally defined symbols, then the translator can determine the relocatability attribute of every field in the object code which it generates. Until we discuss linkers, we shall assume that this condition holds.

We examine relocation for a direct-addressing computer first. The relocating loader, having no access to the source text, cannot determine from inspection of the generated text whether a field is absolute or relative. It cannot even distinguish instructions from data. The translator must therefore specify for each field whether it is relative. One way to specify these relocatability attributes, easily implemented in an assembler or in the assembly step of a compiler, is to emit with each line of text one *relocation bit* for each field. In Fig. 8.1(a), which shows object code generated for the sample assembler-language program of Fig. 2.2, the convention is that bit value unity indicates that the associated field is relative. The relocation constant to be added is the address of the origin of the segment, and is normally specified by the operating system.

The algorithm executed by the relocating loader is simple. It reads lines of object code one at a time. It copies the text of each line to addresses formed by adding the relocation constant to the indicated locations. For each relocation bit equal to unity it also adds the relocation constant to the corresponding text field. This second addition can be performed either before or after the text is copied. If the relocation constant is 40, the resulting content of storage is as shown in Fig. 8.1(b). The location counter values can be omitted from the object code if the lines are presented in serial order with no gaps, as they are in the figure. In that event the loader runs an actual location counter initialized to the origin location. If multiple location counters are provided, the object code must include with each relative address a designation of the location counter which applies, and the translator must either indicate the total amount of storage associated with each location counter or segregate the object code associated with each.

Interleaving relocation bits and length fields with the program text precludes reading the text directly into the storage locations it is to occupy. Moreover, it requires that the text be handled in small units, often of variable length. These disadvantages can be obviated by collecting all the relocation bits into a single contiguous *relocation map* which follows the text of the object code. Such a map is readily prepared by the assembler

	Source program			Object code			
Label	**Opcode**	**Opd1**	**Opd2**	**Locn**	**Len**	**Reloc**	**Text**
	COPY	ZERO	OLDER	00	3	011	13 33 35
	COPY	ONE	OLD	03	3	011	13 34 36
	READ	LIMIT		06	2	01	12 38
	WRITE	OLD		08	2	01	08 36
FRONT	LOAD	OLDER		10	2	01	03 35
	ADD	OLD		12	2	01	02 36
	STORE	NEW		14	2	01	07 37
	SUB	LIMIT		16	2	01	06 38
	BRPOS	FINAL		18	2	01	01 30
	WRITE	NEW		20	2	01	08 37
	COPY	OLD	OLDER	22	3	011	13 36 35
	COPY	NEW	OLD	25	3	011	13 37 36
	BR	FRONT		28	2	01	00 10
FINAL	WRITE	LIMIT		30	2	01	08 38
	STOP			32	1	0	11
ZERO	CONST	0		33	1	0	00
ONE	CONST	1		34	1	0	01
OLDER	SPACE						
OLD	SPACE						
NEW	SPACE						
LIMIT	SPACE						

(a) Before relocation

Locn	**Machine code**
40	13 73 75
43	13 74 76
46	12 78
48	08 76
50	03 75
52	02 76
54	07 77
56	06 78
58	01 70
60	08 77
62	13 76 75
65	13 77 76
68	00 50
70	08 78
72	11
73	00
74	01

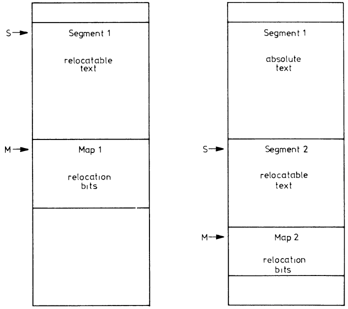
(b) After relocation

Figure 8.1 Use of Relocation Bits

(or other translator), which produces a segment of relocatable text followed immediately by the associated relocation map. The segment and map are read as a unit into storage locations beginning at the segment's assigned origin S, resulting in the status depicted in Fig. 8.2(a). The assembler is required to furnish the lengths of the segment and of the map. Consequently the loader can determine the location M at which the map begins. The loader scans the map and adjusts the addresses which correspond to bits which signify relocation. The segment origin S for the first segment is specified by the operating system. For each succeeding segment it is defined by performing the assignment S←M prior to reading. Figure 8.2(b) shows the status after the first segment has been relocated and the second one loaded.

A rather efficient compromise between line relocation and segment relocation is to divide the text into fixed-length chunks having precisely as many potentially relative fields as the machine word has bits. One chunk of relocatable text is loaded, together with one word of relocation bits. The relative addresses in the chunk are relocated before the next chunk is loaded. This approach captures much of the I/O efficiency of the map method, without requiring extra storage for the last segment map, and permits the use of fixed-length units throughout.

For a machine with explicit-base addressing, any program normally



(a) Before relocation of Segment 1 (b) After relocation of Segment 1

Figure 8.2 Use of Relocation Map

incorporates instructions to load appropriate segment origin locations into base registers at execution time. Relocation is then performed even later during execution time by the machine's calculation of effective addresses for each instruction which references storage. Although this relocation applies to address fields of instructions, it is not performed upon address constants, which must be relocated in much the same manner as are all addresses in a direct-addressing machine. Because address constants typically constitute a very small portion of the text of a program, it is wasteful to supply relocation bits, all but a few of them zero, for all fields of the program text. Instead, the assembler appends a *relocation list* which specifies the adjustments to be made. For the program fragment

```
 0    PROG3     ---     ---
      ---       LOAD    POINTER
      ---       ---     ---
38    LOCN      ---     ---
      ---       ---     ---
71    POINTER   CONST   A(LOCN)
72    TWO       CONST   2
```

(to which the location counter values have been appended), the object code assembled at address 71 for the address constant A(LOCN) is the number 38. The relocation list entry specifies the address (71) to be modified and the appropriate location counter (the one whose origin is associated with the symbol PROG3). The loader merely scans the relocation list after loading the text and makes the indicated adjustments. Address constants are normally not used when programming machines with implicit-base addressing.

The foregoing relocation methods all assume that the original program is free of external references. Relocation of addresses which incorporate external references must be postponed until after those references have been resolved.

8.2.3 *Transfer Vector*

If references to nonlocal data are made through a data area global to all segments (e.g. Fortran COMMON), then truly external references can be restricted to subroutine calls. The translator and loader together can resolve those remaining references by use of a simple mechanism. The translator (say, an assembler) prefixes the program text with a *transfer vector* of symbolic names of the subroutines. After loading the transfer vector and program text, the loader then loads each subroutine specified in the transfer vector, replacing its symbolic name in the transfer vector by the actual address of the subroutine. The assembled code then uses the modified transfer vector in calling the subroutines indirectly. If indirect addressing is available, a call of SUBR can be generated as a branch to the address stored at the transfer vector location associated with SUBR. If only direct addressing is provided, the loader can replace each name in the assembled transfer vector by a branch instruction to the actual subroutine address. In that situation a call of SUBR is generated as a direct branch to the associated transfer vector location, which will branch in turn to the desired subroutine.

Figure 8.3 illustrates the transfer vector mechanism for the direct-addressing machine defined in Chapter 2. The symbols SQRT and SORTFLD are externally defined subroutine names, which appear as branch addresses in (a). The assembler adjoins to the object code a transfer vector, shown at the top of (b), and generates branches to that transfer vector. The loader distinguishes the transfer vector elements from other text by means of a flag bit, or perhaps a special record to delimit the transfer vector. The result of replacing the transfer vector elements is shown in (c).

If the symbolic names used are longer than the space required to hold the address or complete branch instruction in the transfer vector, space can be wasted. This waste can be eliminated at the cost of extra processing by the assembler, which prepares a separate table of subroutine names and the associated locations within the transfer vector. The loader uses that table rather than the transfer vector itself to determine which subroutines to load and where to insert their addresses.

8.2.4 *Linker*

If external references are not limited to subroutine calls, then the definition of a symbol in one segment must be applied in translating each instance of its use in another segment. This resolution is performed by a *linker*, using information supplied to it by the assembler (or compiler). In translating a segment of source program, either the assembler assumes that symbols not defined within the segment are defined externally, or the programmer identifies such symbols explicitly. Symbols which are defined and are expected by the programmer to be referenced by other segments cannot be identified as such by the assembler. If they are not explicitly identified by the programmer, it will be necessary, whenever an externally defined symbol is encountered in some other segment, to search for its definition over all segments external to that other segment.

The detection of errors is easiest if both types of external references are signalled in the source-language program. Many different syntactic forms

```
                              ---                   ---
                           16 SORTFLD            16 BR       376
                           18 SQRT               18 BR       200
---                           ---                   ---
---                           ---                   ---
BR      SQRT               52 BR        18       52 BR        18
---                           ---                   ---
---                           ---                   ---
BRNEG   SORTFLD            93 BRNEG     16       93 BRNEG     16
---                           ---                   ---
---                           ---                   ---
  (a) Source text          (b) After assembly    (c) After loading
```

Figure 8.3 Use of a Transfer Vector

occur, some of which use statement fields in a manner which is inconsistent with their use in other statements. For our illustrative assembler language we adopt two assembler instructions, INTUSE and INTDEF. The INTUSE instruction requires a label, which is a symbol internally used but externally defined. There is no operand. The INTDEF instruction, which is not labeled, takes one operand, which is a symbol which must appear as the label on another line. That symbol is internally defined and expected to be externally used. These instructions usually precede any other occurrences of the symbols which they contain.

The assembler prepares a *definition table* which lists each internally defined global symbol. There is one entry for each symbol, including the name of the program segment. Each entry includes the symbol, its address, and its relocatability mode. The assembler also prepares a *use table* which lists each internally used global symbol. There is one entry for each occurrence, not merely one for each symbol. Each entry includes the symbol itself, the location counter value (i.e. relative address) of the operand field in which it occurs, and the sign with which it occurs. The use table is preferably in address order. The first of two segments to be linked is shown in Fig. 8.4, both in source language and as translated independently. The relocatability mode M ("a" for absolute, "r" for relative) is shown after each word of object code. Although the definition and use tables are shown following the text, either or both can precede the text.

The translation of a symbol defined externally to the segment must necessarily be incomplete. The assembler assigns to it address zero and mode absolute. This permits uniform processing of address expressions as well as of lone symbols. The address corresponding to each global symbol can later be added or subtracted, as appropriate. Its relocatability mode is

```
PROG1     START  0
PROG2     INTUSE
TABLE     INTUSE
          INTDEF TEST
          INTDEF RESUMEPT
          INTDEF HEAD               Addr  Word  M   Word  M
          ---                       ---
          ---                       ---
TEST      BRPOS  PROG2              10    01    a   00    a
RESUMEPT  LOAD   LIST               12    03    a   30    r
          ---                       14
          ---                       ---
          LOAD   TABLE+2            20    03    a   02    a
HEAD      ---                       22
          ---                       ---
          STOP                      29    11    a
LIST      CONST  37                 30    37    a
          END                       31
```

(a) Source program   (b) Object code

Symbol	Addr	Sign
PROG2	11	+
TABLE	21	+

(c) Use table

Symbol	Addr	Mode
PROG1	00	r
TEST	10	r
RESUMEPT	12	r
HEAD	22	r

(d) Definition table

Figure 8.4 First Segment Ready for Linking

also handled straightforwardly, in the manner explained after Fig. 8.8. Thus PROG2 at address 11 is translated into address zero and mode absolute. Likewise, TABLE+2 at address 21 is translated into address 2 (zero for TABLE plus the stated 2) and mode absolute (for the sum of two absolute quantities). The second segment, translated similarly, is given in Fig. 8.5.

The design of the linker is simplified if it is assumed that the independently translated segments are eventually to be loaded into consecutive areas of storage, hence subject to a single relocation constant. We shall relax that restriction later. A two-pass algorithm collects the global symbol definitions during the first pass and applies them during the second. As in assembly, a symbol table is used, called the *global symbol table* (also "external symbol table" or "linkage symbol table"). During Pass 1 the linker merges the definition tables of the several segments into one, taking ap-

```
PROG2     START  0
          INTDEF TABLE
TEST      INTUSE
RESUMEPT  INTUSE
HEAD      INTUSE                   Addr  Word   M   Word   M
          ---                      ---
          ---                      ---
          STORE  TABLE+HEAD-TEST   15     07    a    27    r
          ---                      17
          ---                      ---
          BR     RESUMEPT          25     00    a    00    a
TABLE     SPACE                    27     XX    a
          SPACE                    28     XX    a
          SPACE                    29     XX    a
TWO       CONST  2                 30     02    a
ADDRTEST  CONST  A(TEST)           31     00    a
          END                      32
```

(a) Source program (b) Object code

Symbol	Addr	Sign
HEAD	16	+
TEST	16	-
RESUMEPT	26	+
TEST	31	+

(c) Use table

Symbol	Addr	Mode
PROG2	00	r
TABLE	27	r

(d) Definition table

Figure 8.5 Second Segment Ready for Linking

propriate error action if any symbol has more than one definition. For the first segment processed, the definition table entries are copied unchanged into the global symbol table. In processing the second segment, however, the length of the first segment is added to the address of each relative symbol entered from the second definition table. This ensures that its address is now relative to the origin of the first segment in the about-to-be-linked collection of segments. Either during this pass or during the second, the same adjustment must be made to relative addresses within the program and to location counter values of entries in the use table. In processing the third and subsequent segments, the linker adds the sum of the lengths of the two or more previously processed segments.

Figures 8.6–8.8 show the result of Pass 1 processing for the two segments previously presented, PROG1 having been processed first. It is assumed that the adjustment of addresses in the program text and use table, al-

```
PROG1     START  0
PROG2     INTUSE
TABLE     INTUSE
          INTDEF TEST
          INTDEF RESUMEPT
          INTDEF HEAD
          ---
          ---
TEST      BRPOS  PROG2
RESUMEPT  LOAD   LIST
          ---
          ---
          LOAD   TABLE+2
HEAD      ---
          ---
          STOP
LIST      CONST  37
          END
```

(a) Source program

Addr	Word	M	Word	M

10	01	a	00	a
12	03	a	30	r
14				

20	03	a	02	a
22				

29	11	a		
30	37	a		

(b) Object code

Symbol	Addr	Sign
PROG2	11	+
TABLE	21	+

(c) Use table

Figure 8.6 First Segment after Pass 1 of Linker

though they could have been deferred to Pass 2, were performed during Pass 1.

During Pass 2 the linker updates address fields to reflect the juxtaposition of the segments, unless this was performed during Pass 1. Necessarily deferred to Pass 2 is the actual patching of external references. The object code is copied unchanged, except for the updating just described (if it was indeed deferred), until the field is reached whose address is given in the next entry of the use table for the segment being copied. The symbol in that entry is looked up in the global symbol table, and its address therefrom added to the object code field. For the programs of Figs. 8.6 and 8.7 this first occurs when word 11 is encountered. The symbol PROG2 is looked up in the global symbol table and found to have address 31. This value is added (as directed by the use table sign field) to the 00 in word 11 to yield the correct branch address. The relocation mode indicator of word 11 is also adjusted. The original address is absolute; the address added in is relative. The resulting sum is therefore relative. A similar adjustment

```
PROG2     START  0
          INTDEF TABLE
TEST      INTUSE
RESUMEPT  INTUSE
HEAD      INTUSE                      Addr  Word   M    Word   M
          ---                         ---
          ---                         ---
          STORE  TABLE+HEAD-TEST      46    07     a    58     r
          ---                         48
          ---                         ---
          BR     RESUMEPT             56    00     a    00     a
TABLE     SPACE                       58    XX     a
          SPACE                       59    XX     a
          SPACE                       60    XX     a
TWO       CONST  2                    61    02     a
ADDRTEST  CONST  A(TEST)              62    00     a
          END
```

(a) Source program (b) Object code

Symbol	Addr	Sign
HEAD	47	+
TEST	47	-
RESUMEPT	57	+
TEST	62	+

(c) Use table

Figure 8.7 Second Segment after Pass 1 of Linker

Symbol	Addr	Mode
HEAD	22	r
PROG1	00	r
PROG2	31	r
RESUMEPT	12	r
TABLE	58	r
TEST	10	r

Figure 8.8 Global Symbol Table

occurs in processing word 21, which originally holds the absolute address 02 and is set to the relative address 60.

The processing of word 47 is somewhat more complicated. The original content is the relative address 58. Addition of the value 22 of HEAD yields 80, but HEAD is also relative, and the sum of two relative symbols is illegal. No matter. The linker continues, subtracting the relative address 10 of TEST to yield the legal relative address 70 for the address expression TABLE+HEAD−TEST. During the address adjustment, the linker must count the relative address components algebraically. If the final count is neither 0 nor 1, then an error has occurred. Intermediate values, however, may be other than 0 or 1. The branch address in word 57 and the address constant in word 62 are adjusted without difficulty, and the result of linking the two segments is shown in Fig. 8.9.

We imposed earlier a restriction that the linked program be relocated as

```
PROG1    START   0
PROG2    INTUSE
TABLE    INTUSE
         INTDEF  TEST
         INTDEF  RESUMEPT
         INTDEF  HEAD                 Addr  Word   M    Word   M
         ---                          ---
         ---                          ---
TEST     BRPOS   PROG2                10    01     a     31    r
RESUMEPT LOAD    LIST                 12    03     a     30    r
         ---                          14
         ---                          ---
         LOAD    TABLE+2              20    03     a     60    r
HEAD     ---                          22
         ---                          ---
         STOP                         29    11     a
LIST     CONST   37                   30    37     a
         END
PROG2    START   0
         INTDEF  TABLE
TEST     INTUSE
RESUMEPT INTUSE
HEAD     INTUSE
         ---                          ---
         ---                          ---
         STORE   TABLE+HEAD-TEST      46    07     a     70    r
         ---                          48
         ---                          ---
         BR      RESUMEPT             56    00     a     12    r
TABLE    SPACE                        58    XX     a
         SPACE                        59    XX     a
         SPACE                        60    XX     a
TWO      CONST   2                    61    02     a
ADDRTEST CONST   A(TEST)              62    10     r
         END
```

(a) Source programs adjoined (b) Object code

Figure 8.9 Program after Pass 2 of Linker

a single unit. A simple modification suffices to remove this restriction, and also permits the use of multiple location counters within a single segment. The relocation mode indicator of each word is not just a bit to indicate relative or absolute, but rather the number of the associated location counter. (Zero can be used if the field is absolute.) This information is included with entries in the global symbol table. The linker verifies during Pass 2 that each field is either absolute or else relative to exactly one location counter.

Although the examples of this section have assumed direct addressing, linking of programs for base-addressed machines is performed similarly. Address constants in different segments are then usually relative to different origins, which are identified in the global symbol table.

Just as assembly can be performed in what is nominally a single pass, so can linking. If there is enough storage to hold the linker and all the segments, the following scheme serves. Each segment is read in, preceded by its definition table and followed by its use table. The definition table of the first segment is copied into the initially empty global symbol table. The use table of the first segment includes references to subsequent segments only. Each symbol in that use table is also entered in the global symbol table, but marked as undefined. The remainder of each entry in the use table (address and sign) is placed on a chain linked with its symbol in the table. As each succeeding definition table is read, each symbol is compared with the global symbol table. If it is not in the table, it is entered as before with its definition and marked as defined. If it is already in the table, marked as undefined, its definition is entered in the table and is also used to patch the addresses on the associated chain, the space for which can then be released. If the symbol is already in the table, but marked as defined, a duplicate definition error has occurred.

As each succeeding use table entry is read, its symbol is checked against the table. If the symbol is not in the table, it is entered, marked as undefined, and a new reference chain started. If the symbol is indeed in the table, but undefined, an element is added to the existing chain. If the symbol has already been defined, its table definition is used to patch the specified word in the program, which has already been read in. Alternatively, a true one-pass linker issues to the loader a directive to effect the patch. This organization is not unlike that of the load-and-go assembler, which combines one full pass with a number of mini-passes.

8.2.5 *Linking Loader*

Although linking is conceptually distinct from relocation, the two functions are often implemented concurrently. This is particularly convenient, because the segments must be read for either linking or loading, and both functions require inspection of the relocation bits. Duplication of these efforts is obviated if linking can be deferred until loading time. A program which loads, links, and relocates segments is called a *linking loader*.

The over-all organization of a linking loader is similar to that of either the two-pass or the one-pass linker previously described. It is assumed, of course, that sufficient storage is provided to hold all the segments. The operating system makes a gross allocation of storage, which is known to the linking loader before the first segment is read. The relocation constant associated with each location counter can therefore be established before reading either the definition table or the text of any segment assembled with respect to that location counter. As the definition of each global relative symbol is read into the global symbol table, the appropriate relocation constant is added to its address. The relocation constant for each local relative address is added after the program text has been loaded into its allocated storage.

One important difference between a linker and a linking loader is that the former prepares relocatable code with interspersed relocation information, whereas the latter produces executable code with no insertions. In a two-pass linking loader it is therefore not until the second pass that the program code can be loaded into the locations from which it will be executed. For either the two-pass or the one-pass organization, this loading occurs as each program word is separated from its relocation information and packed next to the previously loaded and adjusted word.

During Pass 1, a two-pass linking loader allocates storage, determines the relocation constants, relocates global relative symbols, and builds the global symbol table. It can defer inspection of program texts and use tables. During Pass 2 it loads the text, relocating local relative symbols as it proceeds, and then adjusts and relocates fields with externally defined symbols. A one-pass linking loader performs all of these operations in a single pass, augmented by following chains of forward references.

The foregoing descriptions have ignored features such as products of global symbols, which are permitted in some assembler languages, and preparation for execution in the absence of enough storage. Mechanisms for handling the former are rather specialized, and will not be discussed here. The latter is the subject of the next two sections.

8.2.6 *Static Overlay Generator*

A program whose storage requirement exceeds its allocation can still be prepared for execution if not all of its segments need to be resident in main storage concurrently. If a reference is made to a segment which is not in main storage, execution must be delayed to allow that segment to be loaded. The resulting decrease in execution efficiency can often be held to a tolerable level by careful selection of the segments to be resident simultaneously. The translator has no practical way of identifying the groups of segments which should be resident simultaneously to prevent unacceptable loss of efficiency. Consequently, this information must be specified by the programmer. The specification customarily takes the form of a static *overlay structure* in the form of a tree, which may be multiply-rooted. Each node of the tree represents a segment. The tree has the property that two segments which are to be in main storage simultaneously lie either on the same path from a root to a leaf or on disjoint paths from a root to a leaf. An example is presented in Fig. 8.10. The unorthodox representation of that tree facilitates the visualization of storage use. Each node is drawn as a vertical line of length proportional to the size of the corresponding segment. The arcs to descendant nodes are drawn as horizontal lines from the bottom end of the parent node. Thus vertical position anywhere in the diagram is in direct correspondence with location in storage. Each segment is labeled by its number and by its storage requirement. Segments 1 and 5 are both root segments; of the others, only segment 3 is not a leaf. Segments 2 and 6 may not be co-resident because they lie on different paths (1,2 and 1,3,6) which are not disjoint. Segments 1, 3, 4, 5, and 8 may be co-resident because segments 1,3, and 4 lie together on one path, segments 5 and 8 lie together on another path, and the two paths are disjoint. The storage required for those five segments is 78K, the maximum needed for any set of segments which may be co-resident. It would require 132K, on the other hand, to store all nine segments simultaneously.

Segments with a common parent are never co-resident, and in fact are assigned the same origin just beyond the end of their parent's allocation. One such segment *overlays* another when it is loaded. A root segment, of course, is never overlaid.

The use of an overlay structure to save space requires the linker and relocating loader together (whether they are combined or not) to ensure that references external to a segment result in the correct execution-time accesses. In examining this requirement it is helpful to distinguish two classes of external references, those to a segment which is permitted to be co-resident (*inclusive* references in IBM parlance) and those to a segment which is prohibited from being co-resident (*exclusive* references). For example, an instruction in segment 1 which loads a word from segment 6 makes an inclusive reference; a call of segment 8 by segment 7 is an exclusive reference. An inclusive reference upward in a path presents no

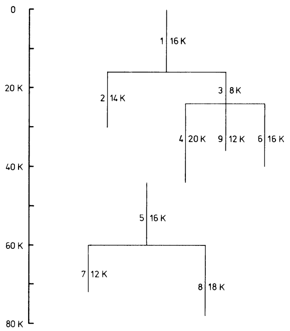


Figure 8.10 Static Overlay Structure

problem, because the referenced segment is necessarily in storage. This is true, for example, of a reference from segment 6 to segment 3. Other inclusive references, such as from segment 3 to segment 8, may require that the referenced segment first be loaded. This action can be effected at execution time only if a description of the tree structure is available and if the loader can be invoked as needed.

The linker must therefore generate a description of the static overlay structure. This description is often called a *segment table*. To ensure that the segment table will be available at any time during execution, it is placed in a root segment, thereby increasing the length of that segment. The segment table also indicates for each segment whether it is loaded or not. At execution time, a control program called the *overlay supervisor* interrogates the segment table for every external reference other than an upward inclusive reference. The overlay supervisor calls the loader if the referenced segment is not resident. It also updates the segment table entries of each overlaid segment and of each newly loaded segment.

The process of loading a segment is known as *dynamic loading*, because the decision to load is made dynamically at execution time. If a segment about to be overlaid has been modified since it was loaded, it is necessary first to copy that segment to backing storage, unless it is known that the segment will never subsequently be reloaded. This rollout of an overlaid segment is obviated if the segment is serially reusable and the overlaying occurs between successive uses, or if the segment is reenterable.

The linker must generate code for execution-time calls to the overlay supervisor. The moments at which these calls will be issued are, of course, not predictable. It is therefore hardly worth preparing and using an overlay structure if the overlay supervisor is not permanently resident in storage during execution of the overlaid program. This, too, somewhat reduces the space saving.

An exclusive external reference presents the further problem that the segment which makes the reference is overlaid before the segment to which reference is made becomes available. For this reason, some systems do not permit exclusive references. Those which do usually limit them to procedure calls and returns, because unrestricted communication between segments which overlay each other is intolerably inefficient. The linker generates a table of all symbols to which an exclusive reference is made. The overlay supervisor can interrogate this exclusive reference table to discover to which segment an exclusive reference is made, and then consult the segment table as before. For simplicity, the exclusive reference table can be incorporated in the root segment, to be available at any time. This may waste storage, however, particularly if the tree has several levels and many exclusive references are to nearby nodes. Although the total space required by the table cannot be reduced, the table itself can be overlaid. For a singly-rooted tree, the table entry for a symbol can be placed in a segment as low as the lowest common ancestor of the segment which contains the referent and those which refer to it. Thus the exclusive reference table may be distributed among as many as all the non-leaf nodes.

8.2.7 *Dynamic Linker*

The static linking of an overlay structure is possible only if the programmer is able and willing to specify at linkage time which sets of segments are to be co-resident at execution time. There exist situations, however, in which the programmer may prefer to, or need to, postpone the specification until execution time. An error routine, for example, may very well not be invoked in the course of a particular execution of the program. Although the programmer knows with which other segments the error routine is to be co-resident if invoked, he may prefer to allocate storage for it dynamically only if and when it is invoked. This precludes the binding, at linkage time, of references to the error routine. The programmer must choose between allocating a maximum amount of space which *might* be required and deferring the resolution of external references until execution time. In many transaction-oriented systems, the determination of which segments must be co-resident is highly data-dependent. There is often a large collection of processing routines, almost any small subset of which may be needed at a given moment, but hardly enough main storage to hold all. Dependence on a static overlay structure could cause intolerable service delays.

A solution is provided by performing dynamically not only the loading, but also the linking. The linker does not attempt to resolve external references beyond translating each into an ordered pair (s, d), where s identifies a segment and d is a displacement from the segment origin. A segment table is used, as before, to indicate for each segment whether it is in storage and, if so, where its origin is. If segment s is not in main storage when the reference is encountered at execution time, a control program causes the segment to be loaded. Dynamic linking is provided by adding the displacement to the origin to obtain a physical storage address. If no reference is made to a particular segment, no effort is wasted in linking it nor storage in holding it.

Some computers have segmentation hardware to perform the table lookup and addition as part of calculating the effective address. For these computers, all addresses can be provided as segment-displacement pairs, and dynamic linking becomes standard practice rather than a complex expedient.

Because program loading is effected just prior to execution, and because storage space is one of the resources managed by the operating system, the implementation of linking, loading, and relocation, particularly dynamic loading and linking, is closely tied to that of the control programs. This is a major example of the way in which translators and control programs work cooperatively to provide a programming system.

FOR FURTHER STUDY

Linking and loading are particularly well presented in the brief Section 8.4 of Brooks and Iverson [1969], the tutorial by Presser and White [1972], which is specific to the IBM 360, and in the following book chapters: Barron [1972, ch. 5], Graham [1976, ch. 6], Arms, Baker, and Pengelly [1976, ch. 13], and Ullman [1976, ch. 5].