Dick Grune · Kees van Reeuwijk
Henri E. Bal · Ceriel J.H. Jacobs
Koen Langendoen

Modern Compiler Design

Second Edition

Springer

Chapter 8
Assemblers, Disassemblers, Linkers, and Loaders

An assembler, like a compiler, is a converter from source code to target code, so many of the usual compiler construction techniques are applicable in assembler construction; they include lexical analysis, symbol table management, and back-patching. There are differences too, though, resulting from the relative simplicity of the source format and the relative complexity of the target format.

8.1 The tasks of an assembler

Assemblers are best understood by realizing that even the output of an assembler is still several steps away from a target program ready to run on a computer. To understand the tasks of an assembler, we will start from an execution-ready program and work our way backwards.

8.1.1 The running program

A running program consists of four components: a code segment, a stack segment, a data segment, and a set of registers. The contents of the **code segment** derive from the source code and are usually immutable; the code segment itself is often extendible to allow dynamic linking. The contents of the **stack segment** are mutable and start off empty. Those of the **data segment** are also mutable and are prefilled from the literals and strings from the source program. The contents of the registers usually start off uninitialized or zeroed.

The code and the data relate to each other through addresses of locations in the segments. These addresses are stored in the machine instructions and in the prefilled part of the data segment. Most operating systems will set the registers of the hardware memory manager unit of the machine in such a way that the address spaces

D. Grune et al., *Modern Compiler Design*, DOI 10.1007/978-1-4614-4699-6_8,

of the code and data segments start at zero for each running program, regardless of where these segments are located in real memory.

8.1.2 The executable code file

A run of a program is initiated by loading the contents of an executable code file into memory, using a **loader**. The loader is usually an integrated part of the operating system, which makes it next to invisible, and its activation is implicit in calling a program, but we should not forget that it is there. As part of the operating system, it has special privileges. All initialized parts of the program derive from the executable code file, in which all addresses should be based on segments starting at zero. The loader reads these segments from the executable code file and copies them to suitable memory segments; it then creates a stack segment, and jumps to a predetermined location in the code segment, to start the program. So the executable code file must contain a code segment and a data segment; it may also contain other indications, for example the initial stack size and the execution start address.

8.1.3 Object files and linkage

The executable code file derives from combining one or more program object files and probably some library object files, and is constructed by a **linker**. The linker is a normal user program, without any privileges. All operating systems provide at least one, and most traditional compilers use this standard linker, but an increasing number of compiling systems come with their own linker. The reason is that a specialized linker can check that the proper versions of various object modules are used, something the standard linker, usually designed for FORTRAN and COBOL, cannot do.

Each object file carries its own code and data segment contents, and it is the task of the linker to combine these into the one code segment and one data segment of the executable code file. The linker does this in the obvious way, by making copies of the segments, concatenating them, and writing them to the executable code file, but there are two complications here. (Needless to say, the object file generator and the linker have to agree on the format of the object files.)

The first complication concerns the addresses inside code and data segments. The code and data in the object files relate to each other through addresses, the same way those in the executable code file do, but since the object files were created without knowing how they will be linked into an executable code file, the address space of each code or data segment of each object file starts at zero. This means that all addresses inside the copies of all object files except the first one have to be adjusted to their actual positions when code and data segments from different object files are linked together.

Suppose, for example, that the length of the code segment in the first object file a.o is 1000 bytes. Then the second code segment, deriving from object file b.o, will start at the location with machine address 1000. All its internal addresses were originally computed with 0 as start address, however, so all its internal addresses will now have to be increased by 1000. To do this, the linker must know which positions in the object segments contain addresses, and whether the addresses refer to the code segment or to the data segment. This information is called **relocation information**. There are basically two formats in which relocation information can be provided in an object file: in the form of bit maps, in which some bits correspond to each position in the object code and data segments at which an address may be located, and in the form of a linked list. Bit maps are more usual for this purpose. Note that code segments and data segments may contain addresses in code segments and data segments, in any combination.

The second complication is that code and data segments in object files may contain addresses of locations in other program object files or in library object files. A location *L* in an object file, whose address can be used in other object files, is marked with an **external symbol**, also called an **external name**; an external symbol looks like an identifier. The location *L* itself is called an **external entry point**. Object files can refer to *L* by using an **external reference** to the external symbol of *L*. Object files contain information about the external symbols they refer to and the external symbols for which they provide entry points. This information is stored in an **external symbol table**.

For example, if an object file a.o contains a call to the routine printf at location 500, the file contains the explicit information in the external symbol table that it refers to the external symbol printf at location 500. And if the library object file printf.o has the body of printf starting at location 100, the file contains the explicit information in the external symbol table that it features the external entry point printf at address 100. It is the task of the linker to combine these two pieces of information and to update the address at location 500 in the copy of the code segment of file a.o to the address of location 100 in the copy of printf.o, once the position of this copy with respect to the other copies has been established.

The linking process for three code segments is depicted in Figure 8.1; the segments derive from the object files a.o, b.o, and printf.o mentioned above. The length of the code segment of b.o is assumed to be 3000 bytes and that of printf.o 500 bytes. The code segment for b.o contains three internal addresses, which refer to locations 1600, 250, and 400, relative to the beginning of the segment; this is indicated in the diagram by having relocation bit maps along the code and data segments, in which the bits corresponding to locations 1600, 250, and 400 are marked with a C for "Code". The code segment for a.o contains one external address, of the external symbol printf as described above. The code segment for printf.o contains one external entry point, the location of printf. The code segments for a.o and printf.o will probably also contain many internal addresses, but these have been ignored here.

Segments usually contain a high percentage of internal addresses, much higher than shown in the diagram, and relocation information for internal addresses requires only a few bits. This explains why relocation bit maps are more efficient than

linked lists for this purpose.

The linking process first concatenates the segments. It then updates the internal addresses in the copies of a.o, b.o, and printf.o by adding the positions of those segments to them; it finds the positions of the addresses by scanning the relocation maps, which also indicate if the address refers to the code segment or the data segment. Finally it stores the external address of printf, which computes to 4100 (=1000+3000+100), at location 100, as shown.

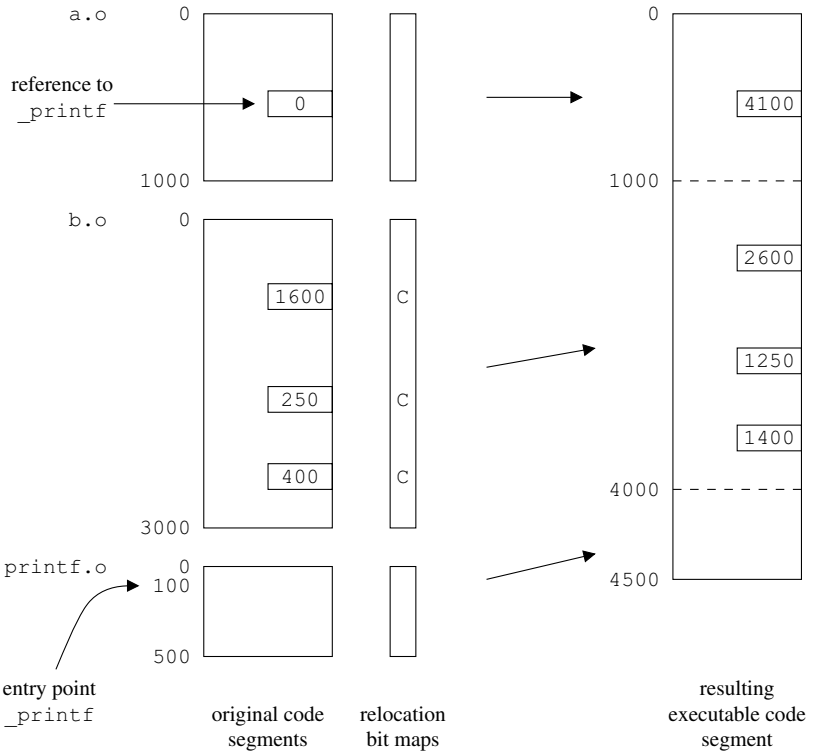

Fig. 8.1: Linking three code segments

We see that an object file needs to contain at least four components: the code segment, the data segment, the relocation bit map, and the external symbol table.

8.1.4 Alignment requirements and endianness

Although almost every processor nowadays uses addresses that represent (8-bit) bytes, there are often **alignment requirements** for some or all memory accesses. For example, a 16-bit (2-byte) aligned address points to data whose address is a

multiple of 2. Modern processors require 16, 32, or even 64-bit aligned addresses. Requirements may differ for different types. For example, a processor might require 32-bit alignment for 32-bit words and instructions, 16-bit alignment for 16-bit words, and no particular alignment for bytes. If such restrictions are violated, the penalty is slower memory access or a processor fault, depending on the processor. So the compiler or assembler may need to do **padding** to honor these requirements, by inserting unused memory segments for data and no-op instructions for code.

Another important issue is the exact order in which data is stored in memory. For the bits in a byte there is nowadays a nearly universal convention, but there are two popular choices for storing multi-byte values. First, values can be stored with the least significant byte first, so that for hexadecimal number 1234 the byte 34 has the lowest address, and the value 12 has the address after that. This storage convention is called **little-endian**. It is also possible to place the most significant byte first, so that the byte 12 has the lowest address. This storage convention is called **big-endian**. There are no important reasons to choose one **endianness** over the other[1], but since conversion from one form to another takes some time and forgetting to convert can introduce subtle bugs, most architectures pick one of the two and stick to it.

We are now in a position to discuss issues in the construction of assemblers and linkers. We will not go into the construction of loaders, since they hardly require any special techniques and are almost universally supplied with the operating system.

8.2 Assembler design issues

An assembler converts from symbolic machine code to binary machine code, and from symbolic data to binary data. In principle the conversion is one to one; for example the 80x86 assembler instruction

```
addl %edx,%ecx
```

which does a 32-bit addition of the contents of the %edx register to the %ecx register, is converted to the binary data

```
0000 0001 11 010 001 (binary) = 01 D1 (hexadecimal)
```

The byte 0000 0001 is the operation code of the operation addl, the next two bits 11 mark the instruction as register-to-register, and the trailing two groups of three bits 010 and 001 are the translations of %edx and %ecx. It is more usual to write the binary translation in hexadecimal; as shown above, the instruction is 01D1 in this notation. The binary translations can be looked up in tables built into the assembler. In some assembly languages, there are some minor complications due to the overloading of instruction names, which have to be resolved by considering the types of the operands. The bytes of the translated instructions are packed closely, with no-op

[1] The insignificance of the choice is implied in the naming: it refers to *Gulliver's Travels* by Jonathan Swift, which describes a war between people who break eggs from the small or the big end to eat them.

instructions inserted if alignment requirements would leave gaps. A **no-op instruction** is a one-byte machine instruction that does nothing (except perhaps waste a machine cycle).

The conversion of symbolic data to binary data involves converting, for example, the two-byte integer 666 to hexadecimal 9A02 (again on an 80x86, which is a little-endian machine), the double-length (8-byte) floating point number 3.1415927 to hex 97D17E5AFB210940, and the two-byte string "PC" to hex 5043. Note that the string in assembly code is not extended with a null byte; the null-byte terminated string is a C convention, and language-specific conventions have no place in an assembler. So the C string "PC" must be translated by the code generator to "PC\0" in symbolic assembly code; the assembler will then translate this to hex 504300.

The main problem in constructing an assembler lies in the handling of addresses. Two kinds of addresses are distinguished: internal addresses, referring to locations in the same segment; and external addresses, referring to locations in segments in other object files.

8.2.1 Handling internal addresses

References to locations in the same code or data segment take the form of identifiers in the assembly code; an example is shown in Figure 8.2. The fragment starts with material for the data segment (.data), which contains a location of 4 bytes (.long) aligned on a 8-byte boundary, filled with the value 666 and labeled with the identifier var1. Next comes material for the code segment (.code) which contains, among other instructions, a 4-byte addition from the location labeled var1 to register %eax, a jump to label label1, and the definition of the label label1.

```
.data
        ...
        .align 8
var1:
        .long 666
        ...
.code
        ...
        addl var1,%eax
        ...
        jmp label1
        ...
label1:
        ...
        ...
```

Fig. 8.2: Assembly code fragment with internal symbols

The assembler reads the assembly code and assembles the bytes for the data and the code segments into two different arrays. When the assembler reads the fragment from Figure 8.2, it first meets the .data directive, which directs it to start assembling into the data array. It translates the source material for the data segment to binary, stores the result in the data array, and records the addresses of the locations at which the labels fall. For example, if the label var1 turns out to label location 400 in the data segment, the assembler records the value of the label var1 as the pair (data, 400). Note that in the assembler the value of var1 is 400; to obtain the value of the program variable var1, the identifier var1 must be used in a memory-reading instruction, for example addl var1,%eax.

Next, the assembler meets the .code directive, after which it switches to assembling into the code array. While translating the code segment, the assembler finds the instruction addl var1,%eax, for which it assembles the proper binary pattern and register indication, plus the value of the data segment label var1, 400. It stores the result in the array in which the code segment is being assembled. In addition, it marks the location of this instruction as "relocatable to the data segment" in the relocation bit map. When the assembler encounters the instruction jmp label1, however, it cannot do something similar, since the value of label1 is not yet known.

There are two solutions to this problem: backpatching and two-scans assembly. When using *backpatching*, the assembler keeps a backpatch list for each label whose value is not yet known. The backpatch list for a label L contains the addresses $A_1 \ldots A_n$ of the locations in the code and data segments being assembled, into which the value of L must eventually be stored. When an applied occurrence of the label L is encountered and the assembler decides that the value of L must be assembled into a location A_i, the address A_i is inserted in the backpatch list for L and the location at A_i is zeroed. The resulting arrangement is shown in Figure 8.3, which depicts the assembly code, the assembled binary code, and one backpatch list, for the label label1. When finally the defining occurrence of L is found, the address of the position it labels is determined and assigned to L as its value. Next the backpatch list is processed, and for each entry A_k, the value of L is stored in the location addressed by A_k.

In *two-scans assembly*, the assembler processes its input file twice. The purpose of the first scan is to determine the values of all labels. To this end, the assembler goes through the conversion process described above, but without actually assembling any code: the assembler just keeps track of where everything would go. During this process it meets the defining occurrences of all labels. For each label L, the assembler can record in its symbol table the value of L, since that value derives from the position that L is found to label. During the second scan, the values of all labels are known and the actual translation can take place without problems.

Some additional complications may occur if the assembly language supports features like macro processing, multiple segments, labels in expressions, etc., but these are mostly of an administrative nature.

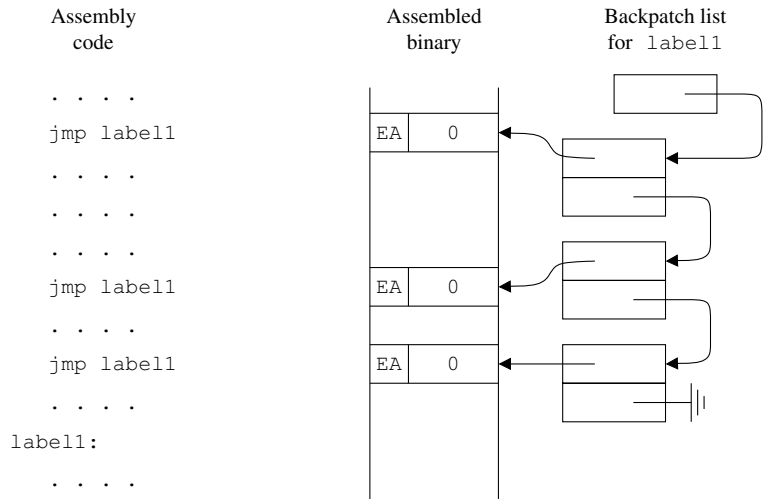

Fig. 8.3: A backpatch list for labels

8.2.2 Handling external addresses

The external symbol and address information of an object file is summarized in its external symbol table, an example of which is shown in Figure 8.4. The table specifies, among other things, that the data segment has an entry point named options at location 50, the code segment has an entry point named main at location 100, the code segment refers to an external entry point printf at location 500, etc. Also there is a reference to an external entry point named file_list at location 4 in the data segment. Note that the meaning of the numbers in the address column is completely different for entry points and references. For entry points, the number is the value of the entry point symbol; for references, the number is the address where the value of the referred entry point must be stored.

The external symbol table can be constructed easily while the rest of the translation is being done. The assembler then produces a binary version of it and places it in the proper position in the object file, together with the code and data segments, the relocation bit maps, and possibly further header and trailer material.

Additionally the linker can create tables for the debugging of the translated program, using information supplied by the compiler. In fact, many compilers can generate enough information to allow a debugger to find the exact variables and statements that originated from a particular code fragment.

External symbol	Type	Address
options	entry point	50 data
main	entry point	100 code
printf	reference	500 code
atoi	reference	600 code
printf	reference	650 code
exit	reference	700 code
msg_list	entry point	300 data
Out_Of_Memory	entry point	800 code
fprintf	reference	900 code
exit	reference	950 code
file_list	reference	4 data

Fig. 8.4: Example of an external symbol table

8.3 Linker design issues

The basic operation of a linker is simple: it reads each object file and appends each of the four components to the proper one of four lists. This yields one code segment, one data segment, one relocation bit map, and one external symbol table, each consisting of the concatenation of the corresponding components of the object files. In addition the linker retains information about the lengths and positions of the various components. It is now straightforward to do the relocation of the internal addresses and the linking of the external addresses; this resolves all addresses. The linker then writes the code and data segments to a file, the executable code file; optionally it can append the external symbol table and debugging information. This finishes the translation process that we started in the first line of Chapter 2!

Real-world linkers are often more complicated than described above, and constructing one is not a particularly simple task. There are several reasons for this. One is that the actual situation around object modules is much hairier than shown here: many object file formats have features for repeated initialized data, special arithmetic operations on relocatable addresses, conditional external symbol resolution, etc. Another is that linkers often have to wade through large libraries to find the required external entry points, and advanced symbol table techniques are used to speed up the process. A third is that users tend to think that linking, like garbage collection, should not take time, so there is pressure on the linker writer to produce a blindingly fast linker.

One obvious source of inefficiency is the processing of the external symbol table. For each entry point in it, the entire table must be scanned to find entries with the same symbol, which can then be processed. This leads to a process that requires a time $O(n^2)$ where n is the number of entries in the combined external symbol table. Scanning the symbol table for each symbol can be avoided by sorting it first; this brings all entries concerning the same symbol together, so they can be processed efficiently.