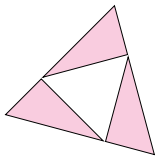

Chapter 9

Building a Runnable Program

9.1 Back-End Compiler Structure

9.2 Intermediate Forms

9.3 Code Generation

9.4 Address Space Organization

Assemblers, linkers, and loaders typically operate on a pair of related file formats: *relocatable* object code and *executable* object code. Relocatable object code is acceptable as input to a linker; multiple files in this format can be combined to create an executable program. Executable object code is acceptable as input to a loader: it can be brought into memory and run. A relocatable object file includes the following descriptive information:

import table – Identifies instructions that refer to named locations whose addresses are unknown, but are presumed to lie in other files yet to be linked to this one;

relocation table – Identifies instructions that refer to locations within the current file, but that must be modified at link time to reflect the offset of the current file within the final, executable program.

export table – Lists the names and addresses of locations in the current file that may be referred to in other files.

Imported and exported names are known as *external symbols*.

An executable object file is distinguished by the fact that it contains no references to external symbols. It also defines a starting address for execution. An executable file may or may not be relocatable, depending on whether it contains the tables above.

Internally, an object file is typically divided into several sections, each of which is handled differently by the linker, loader, or operating system. The first section includes the import, export, and relocation tables, together with an indication of how much space will be required by the program for non-initialized static data. Other sections commonly include: code (instructions), read-only data (constants, jump tables for `case` statements, etc.), initialized but writable static data, and high-level symbol table information saved by the compiler. The initial descriptive section is used by the linker and loader. The high-level symbol table section is used by debuggers and performance profilers. Neither of these tables is usually brought into memory at run time; neither is needed by the running program.

In its runnable (loaded) form, a program is typically organized into several *segments*. On some machines (e.g. the x86 or PA-RISC), segments are visible to the assembly language programmer, and must be named explicitly in instructions. More commonly on modern machines, segments are simply subsets of the address space that the operating system manages in different ways. Two or three of them—code, constants, and initialized data—correspond to sections of the object file. Code and constants are usually read-only, and are often combined in a single segment; the operating system arranges to receive an interrupt if the program attempts to modify them. (In response to such an interrupt it will most likely print an error message and terminate the program.) Initialized data is writable. At load time, the operating system either reads code, constants, and initialized data from disk, or arranges to read them in at run time, in response to "invalid access" (*page fault*) interrupts or dynamic linking requests.

In addition to code, constants, and initialized data, the typical running program has two or more additional segments. These include:

uninitialized data – May be allocated at load time or on demand in response to page faults. Usually zero-filled, both to provide repeatable symptoms for programs that erroneously read data they have not yet written, and to enhance security on multi-user systems, by preventing a program from reading the contents of pages written by previous users.

stack – May be allocated in some fixed amount at load time. More commonly, is given a small initial size, and is then extended automatically by the operating system in response to (faulting) accesses beyond the current segment end.

heap – Like stack, may be allocated in some fixed amount at load time. More commonly, is given a small initial size, and is then extended in response to explicit requests (via system call) from heap-management library routines.

files – In many systems, library routines allow a program to *map* a file into memory. The `map` routine interacts with the operating system to create a new segment for the file, and returns the address of the beginning of the segment. The contents of the segment are usually fetched from disk on demand, in response to page faults.

9.5 Assembly

9.6 Linking

Most language implementations—certainly all that are intended for the construction of large programs—support separate compilation: fragments of the program can be compiled and assembled more-or-less independently. After compilation, these fragments (known as *compilation units*) are "glued together" by a *linker*. In many languages and environments, the programmer explicitly divides the program into modules or files, each of which is separately compiled. More integrated environments may abandon the notion of a file in favor of a database of subroutines, each of which is separately compiled.

The task of a *linker* is to join together compilation units. A *static linker* does its work prior to program execution, producing an executable object file. A *dynamic linker* (to be described in section 9.7) does its work after the program has been brought into memory for execution.

Each of the compilation units of a program to be linked must be a relocatable object file. Typically, some of these files will have been produced by compiling fragments of the application being constructed, while others will be general-purpose library packages needed by the application. Since most programs make use of libraries, even a "one-file" application typically needs to be linked.

Linking involves two subtasks: relocation and the resolution of external references. Some authors refer to relocation as *loading*, and call the entire "joining together" process "link-loading". Other authors (including the current one) use "loading" to refer to the process of bringing an executable object file into memory for execution. On very simple machines, or on machines with very simple operating systems, loading entails relocation. More commonly, the operating system uses virtual memory to give every program the impression that it starts at some standard address (e.g. zero). In section 9.7 we shall see that on many systems loading entails a certain amount of linking.

9.6.1 Relocation and Name Resolution

Each relocatable object file contains the information required for linking: the import, export, and relocation tables. A static linker uses this information in a two-phase process analogous to that described for assemblers in section 9.5. In the first phase, the linker gathers all of the compilation units together, chooses an order for them in memory, and notes the address at which each will consequently lie. In the second phase, the linker processes each unit, replacing unresolved external references with appropriate addresses, and modifying instructions that need to be relocated to reflect the addresses of their units. These phases are illustrated pictorially in figure 9.1. Addresses and offsets are assumed to be written in hexadecimal notation, with a page size of 4K (1000_{16}) bytes.

Libraries present a bit of a challenge. Many consist of hundreds of separately compiled program fragments, most of which will not be needed by any particular application. Rather than link the entire library into every application, the linker needs to search the library to identify the fragments that are referenced from the main program. If these refer to additional fragments, then those must be included also, recursively. Many systems support a special library format for relocatable object files. A library in this format may contain

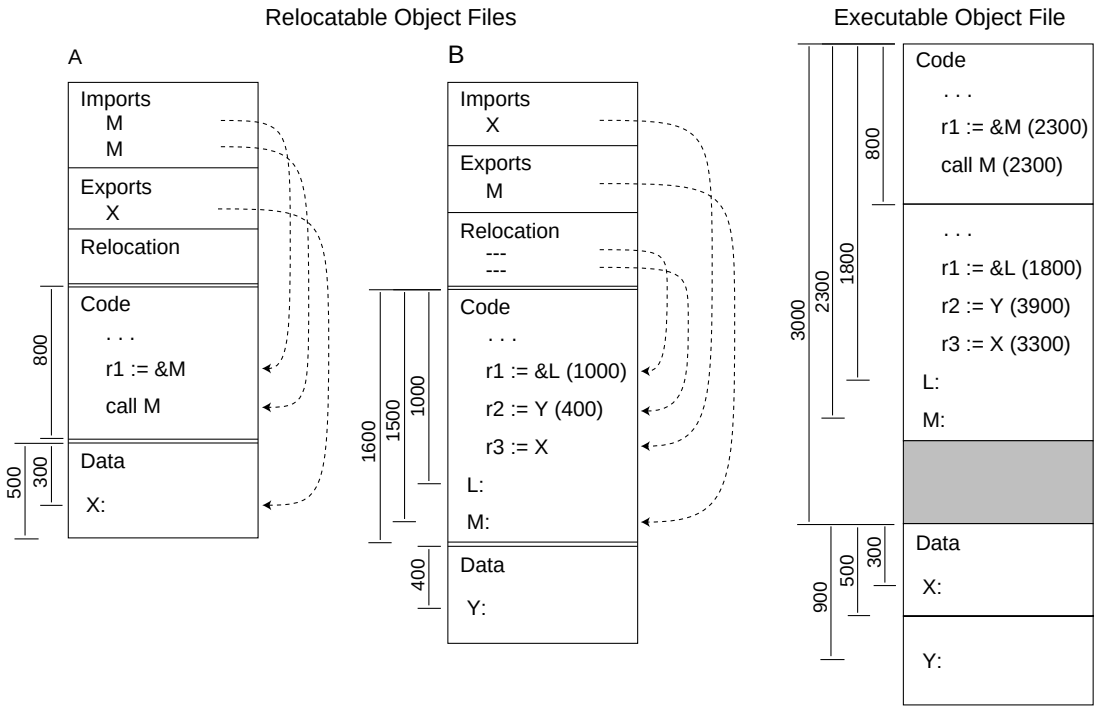

Figure 9.1: Linking relocatable object files A and B to make an executable object file. A's code section has been placed at offset 0, with B's code section immediately after, at offset 800. To allow the operating system to establish different protections for the code and data segments, A's data section has been placed at the next page boundary (offset 3000), with B's data section immediately after (offset 3500). External references to M and X have been set to use the appropriate addresses. Internal references to L and Y have been updated by adding in the starting addresses of B's code and data sections, respectively.

an arbitrary number of code and data sections, together with an index that maps symbol names to the sections in which they appear.

9.6.2 Type Checking

9.7 Dynamic Linking

On a multi-user system, it is common for several instances of a program (an editor or web browser, for example) to be executing simultaneously. It would be highly wasteful to allocate space in memory for a separate, identical copy of the code of such a program for every running instance. Many operating systems therefore keep track of the programs that are running, and set up memory mapping tables so that all instances of the same program share the same read-only copy of the program's code segment. Each instance receives its own writable copy of the data segment. Code segment sharing can save enormous amounts of space. It does not work, however, for instances of programs that are similar but not identical.

Many sets of programs, while not identical, have large amounts of library code in common, e.g. to manage a graphical user interface. If every application has its own copy of the library, then large amounts of memory may be wasted. Moreover, if programs are statically linked, then much larger amounts of disk space may be wasted on nearly identical copies of the library in separate executable object files.

In the early 1990's, most operating system vendors adopted *dynamic linking*, in order to avoid this waste. Each dynamically linked library resides in its own code and data segments. Every program instance that uses a given library has a private copy of the library's data segment, but shares a single system-wide read-only copy of the library's code segment. These segments may be linked to the remainder of the code when the program is loaded into memory, or they may be linked incrementally on demand, during execution. In addition to saving space, dynamic linking allows a programmer or system administrator to install backward-compatible updates to a library without rebuilding all existing executable object files: the next time it runs, each program will obtain the new version of the library automatically.

To be amenable to dynamic linking, a library must either (1) be located at the same address in every program that uses it, or (2) have no relocatable words in its code segment, so that the content of the segment does not depend on its address. The first approach is straightforward but restrictive: it generally requires that we assign a unique address to every sharable library; otherwise we run the risk that some newly created program will want to use two libraries that have been given overlapping address ranges. In Unix System V R3, which took the unique-address approach, shared libraries could only be installed by the system administrator. This requirement tended to limit the use of dynamic linking to a relatively small number of popular libraries. The second approach, in which a shared library can be linked at any address, allows users to employ dynamic linking whenever they want.

9.7.1 Position-Independent Code

A code segment that contains no relocatable words is said to constitute *position-independent code* (PIC). To generate PIC, the compiler must

1. use PC-relative addressing, rather than jumps to absolute addresses, for all internal branches.

2. similarly, avoid absolute references to statically allocated data, by using displacement addressing with respect to some standard base register. If the code and data segments are guaranteed to lie at a known offset from one another, then an entry point to a shared library can compute an appropriate base register value using the PC. Otherwise the caller must set the base register as part of the calling sequence.

3. use an extra level of indirection for every control transfer out of the PIC segment, and for every load or store of static memory outside the corresponding data segment. The indirection allows the (non-PIC) target address to be kept in the data segment, which is private to each program instance.

Exact details vary among processors, vendors, and operating systems. Conventions for SGI's compilers for the MIPS architecture, under the IRIX 6.2 version of Unix, are illustrated in

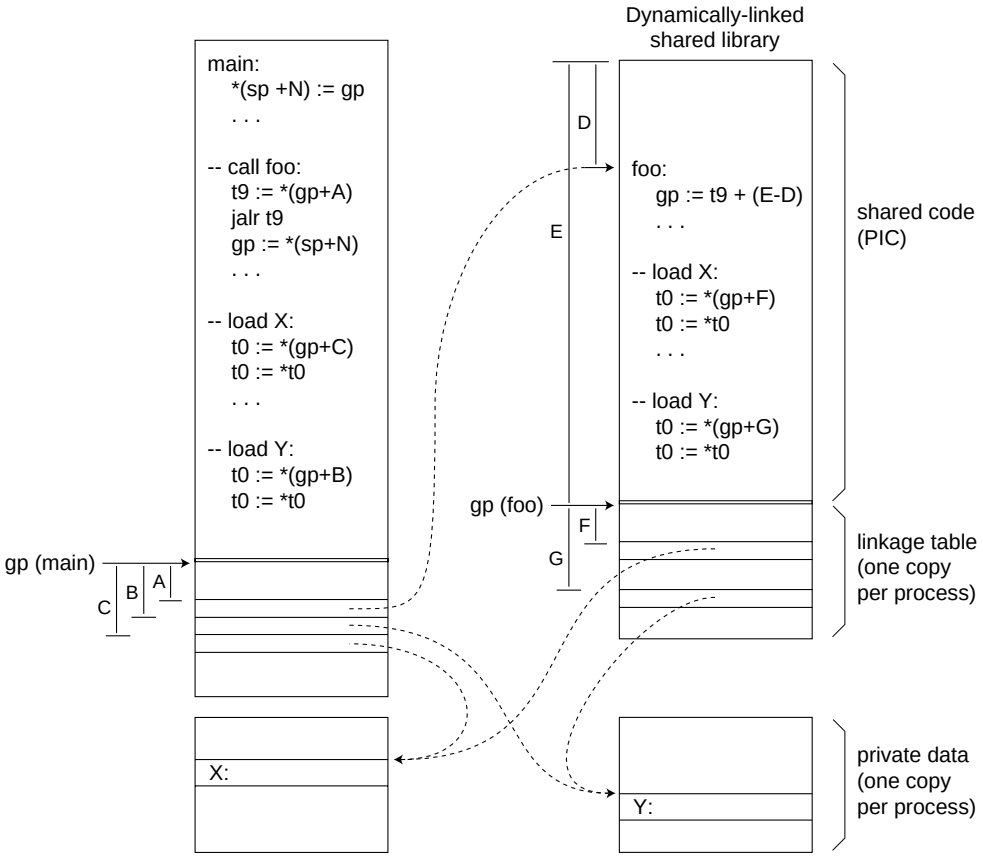

Figure 9.2: A dynamically linked shared library. Because main calls foo, which lies in the library, its prologue and epilogue must save and restore both ra (not shown) and gp. Calls to foo are made indirectly, using an address stored in main's linkage table. Similarly, references to variables X and Y, both of which are globally visible, must employ a level of indirection. In the prologue of foo, gp is set to point to foo's linkage table, using the value in t9. The calling sequence in main restores the old gp when foo returns.

figure 9.2. Each shared code segment is accompanied, at a static offset, by a non-shared *linkage table* and, at an arbitrary offset, by a non-shared data segment. The linkage table lists the addresses of all external symbols referenced in the code segment.

As described in section 8.2.1, any non-leaf subroutine must allocate space in its stack frame to hold the value of the `ra` (return address) register, and must save and restore this register in its prologue and epilogue. Similarly, any subroutine that may call into a dynamically linked shared library must save the `gp` (global pointer) register in the prologue, and restore it after every call into a dynamically linked shared library. At code-generation time, the compiler must know which external symbols lie in such libraries. For a call to one of them, the usual `jal` (jump-and-link) instruction is replaced by a sequence of three instructions. The first of these loads register `t9` from the linkage table, using `gp`-relative addressing. The second is a `jalr` (jump-and-link-register) instruction, which takes its target address from `t9`. The third restores the `gp`. In a similar vein, any load or store of a datum located in a dynamically linked shared library must employ a two-instruction sequence.

The first instruction loads the address of the datum from the linkage table using gp-relative addressing. The second loads or stores the datum itself.

The prologue of any subroutine foo that serves as an entry to a dynamically linked shared library must establish a new gp. To do so it takes the value in t9 (i.e. the address of foo) and adds the (statically known) signed difference between foo's offset within the code segment and the distance between the code and the linkage table.

9.7.2 Fully Dynamic (Lazy) Linking

If all or most of the symbols exported by a shared library are referenced by the parent program, then it makes sense to link the library in its entirety at load time. In any given execution of a program, however, there may be references to libraries that are not actually used, because the input data never causes execution to follow the code path(s) on which the references appear. If these "potentially unnecessary" references are numerous, we may avoid a significant amount of work by linking the library *lazily* on demand. Moreover even in a program that uses all its symbols, incremental lazy linking may improve the system's interactive responsiveness by allowing programs to begin execution faster. Finally, a language system that allows the dynamic creation of program components (e.g. as in Common Lisp or Java) must use lazy linking to delay the resolution of external references in compiled components.

The run-time data structures for lazy linking are almost the same as those in figure 9.2, but they are incrementally created. At load time, the program begins with the main code segment and linkage table, and with all data segments for which addresses need to appear in that linkage table. In our specific example, we would load the data segments of both main and foo, because the addresses of both X (which belongs to main) and Y (which belongs to foo) need to appear in the main linkage table. We would not, however, load the code segment or linkage table of foo, despite the fact that the address of foo needs to appear in the linkage table. Instead, we would initialize that linkage table entry to refer to a *stub* routine, created by the compiler and included in the main code segment. The code of the stub looks like this:

```
t9 := *(gp+k)      -- lazy linker entry point
t7 := ra
t8 := n            -- index of stub
call *t9           -- overwrites ra
```

The lazy linker itself resides in a (non-lazy) shared library, linked to the program at load time. (Here we have assumed that its address lies at offset k in the linkage table.)

After branching to the lazy linker, control never returns to the stub. Instead, the linker uses the constant n to index into the import table of the program's object file, where it finds the information it needs to identify both the name and the library of the unresolved reference. The linker then loads the library's code segment into memory if it is not already there. At this point it can change ("patch") the linkage table entry through which the stub was called, so that it now points to the library routine. If it needed to load the library's code segment, the linker also creates a copy of the library's linkage table. It initializes all data entries in that table, loading (copies of) the segments to which those entries refer if they (the segments) have not already been loaded as part of an earlier linking operation.

For each subroutine entry in the library's linkage table, the linker checks to see whether the relevant code segment has already been loaded. If so, it initializes the entry with the subroutine's address. If not, it initializes it with the address of its stub. Finally, the linker copies `t7` into `ra` and jumps to the newly linked library routine. At this point, everything appears as though the call had happened in the normal fashion.

As execution proceeds, further references to not-yet-loaded symbols extend the "frontier" of the program. Because invocations of the linker occur on subroutine calls and not on data references, the current frontier always includes a set of code segments and the data segments to which those code segments refer. Each linking operation brings in one new code segment, together with all of the additional data segments to which that code refers. If we were willing to intercept page faults, we could arrange to enter the linker on references to not-yet-loaded data. This approach would avoid loading data segments that are never really used, but the overhead of the faults might greatly increase execution time.