14 Building a Runnable Program

14.7 Dynamic Linking

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 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.

14.7.1 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 observe the following rules.

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.

EXAMPLE 14.19
PIC under MIPS/IRIX

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

As described in Section © 8.2.2, any nonleaf 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 (to be executed after the return) 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. ■

14.7.2 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

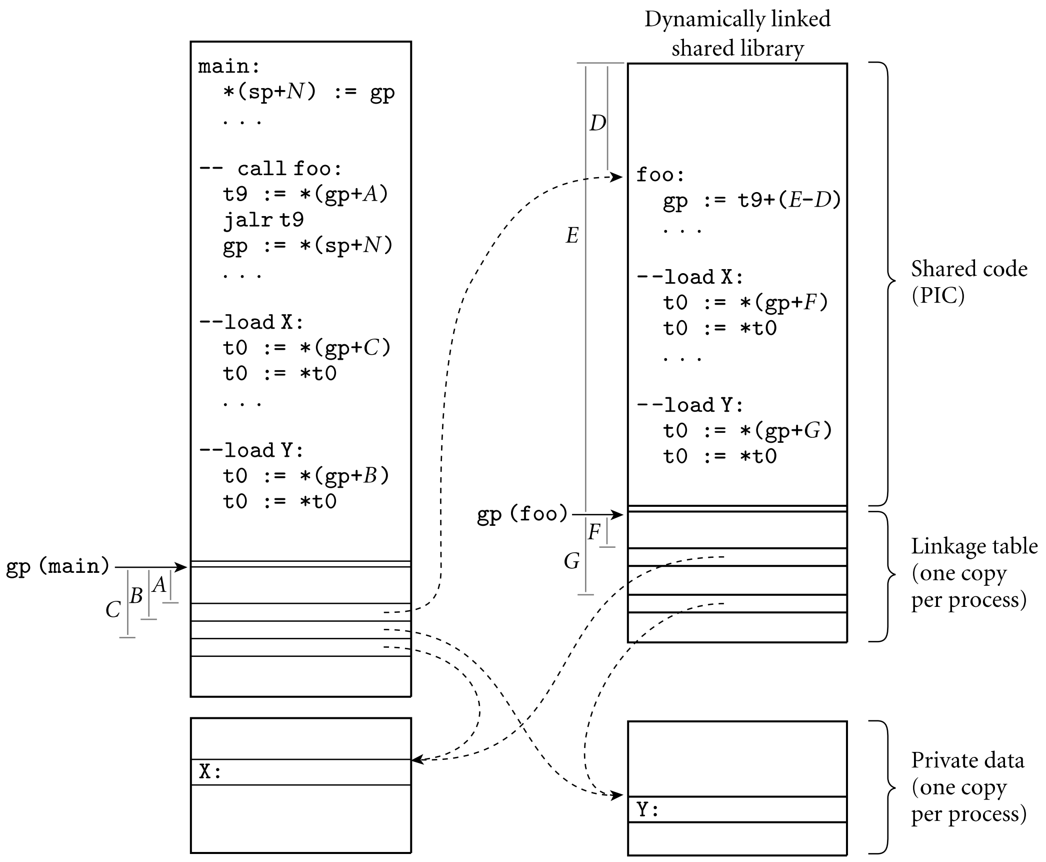

Figure 14.12 **A dynamically linked shared library.** Because `main` calls `foo`, which lies in the library, its prologue and epilogue must save and restore both `ra` (not shown) and `gp`. Calls to `foo` are made indirectly, using an address stored in `main`'s linkage table. Similarly, references to 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.

in Common Lisp or Java) must use lazy linking to delay the resolution of external references in compiled components.

EXAMPLE 14.20
Dynamic linking under MIPS/IRIX

The run-time data structures for lazy linking are almost the same as those in Figure 14.12, 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 (nonlazy) 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.

✓ CHECK YOUR UNDERSTANDING

28. Explain the addressing challenge faced by dynamic linking systems.

29. What is *position-independent code*? What is it good for? What special precautions must a compiler follow in order to produce it?

30. Explain the significance of the `gp` (global pointer) register in a program with dynamic linking.

31. What is the purpose of a *linkage segment*?

32. What is *lazy* dynamic linking? What is its purpose? How does it work?