Aside Shared libraries and the Java Native Interface

Java defines a standard calling convention called *Java Native Interface (JNI)* that allows "native" C and C++ functions to be called from Java programs. The basic idea of JNI is to compile the native C function, say, `foo`, into a shared library, say, `foo.so`. When a running Java program attempts to invoke function `foo`, the Java interpreter uses the `dlopen` interface (or something like it) to dynamically link and load `foo.so` and then call `foo`.

7.12 Position-Independent Code (PIC)

A key purpose of shared libraries is to allow multiple running processes to share the same library code in memory and thus save precious memory resources. So how can multiple processes share a single copy of a program? One approach would be to assign a priori a dedicated chunk of the address space to each shared library, and then require the loader to always load the shared library at that address. While straightforward, this approach creates some serious problems. It would be an inefficient use of the address space because portions of the space would be allocated even if a process didn't use the library. It would also be difficult to manage. We would have to ensure that none of the chunks overlapped. Each time a library was modified, we would have to make sure that it still fit in its assigned chunk. If not, then we would have to find a new chunk. And if we created a new library, we would have to find room for it. Over time, given the hundreds of libraries and versions of libraries in a system, it would be difficult to keep the address space from fragmenting into lots of small unused but unusable holes. Even worse, the assignment of libraries to memory would be different for each system, thus creating even more management headaches.

To avoid these problems, modern systems compile the code segments of shared modules so that they can be loaded anywhere in memory without having to be modified by the linker. With this approach, a single copy of a shared module's code segment can be shared by an unlimited number of processes. (Of course, each process will still get its own copy of the read/write data segment.)

Code that can be loaded without needing any relocations is known as *position-independent code (PIC)*. Users direct GNU compilation systems to generate PIC code with the `-fpic` option to GCC. Shared libraries must always be compiled with this option.

On x86-64 systems, references to symbols in the same executable object module require no special treatment to be PIC. These references can be compiled using PC-relative addressing and relocated by the static linker when it builds the object file. However, references to external procedures and global variables that are defined by shared modules require some special techniques, which we describe next.

PIC Data References

Compilers generate PIC references to global variables by exploiting the following interesting fact: no matter where we load an object module (including shared

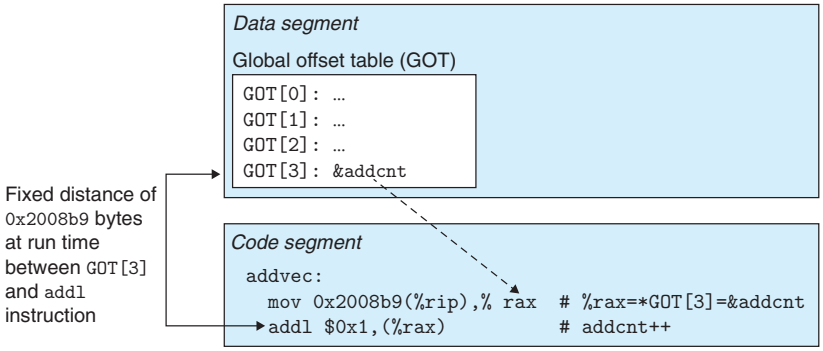


Figure 7.18 Using the GOT to reference a global variable. The `addvec` routine in `libvector.so` references `addcnt` indirectly through the GOT for `libvector.so`.

object modules) in memory, the data segment is always the same distance from the code segment. Thus, the *distance* between any instruction in the code segment and any variable in the data segment is a run-time constant, independent of the absolute memory locations of the code and data segments.

Compilers that want to generate PIC references to global variables exploit this fact by creating a table called the *global offset table (GOT)* at the beginning of the data segment. The GOT contains an 8-byte entry for each global data object (procedure or global variable) that is referenced by the object module. The compiler also generates a relocation record for each entry in the GOT. At load time, the dynamic linker relocates each GOT entry so that it contains the absolute address of the object. Each object module that references global objects has its own GOT.

Figure 7.18 shows the GOT from our example `libvector.so` shared module. The `addvec` routine loads the address of the global variable `addcnt` indirectly via `GOT[3]` and then increments `addcnt` in memory. The key idea here is that the offset in the PC-relative reference to `GOT[3]` is a run-time constant.

Since `addcnt` is defined by the `libvector.so` module, the compiler could have exploited the constant distance between the code and data segments by generating a direct PC-relative reference to `addcnt` and adding a relocation for the linker to resolve when it builds the shared module. However, if `addcnt` were defined by another shared module, then the indirect access through the GOT would be necessary. In this case, the compiler has chosen to use the most general solution, the GOT, for all references.

PIC Function Calls

Suppose that a program calls a function that is defined by a shared library. The compiler has no way of predicting the run-time address of the function, since the shared module that defines it could be loaded anywhere at run time. The normal approach would be to generate a relocation record for the reference, which

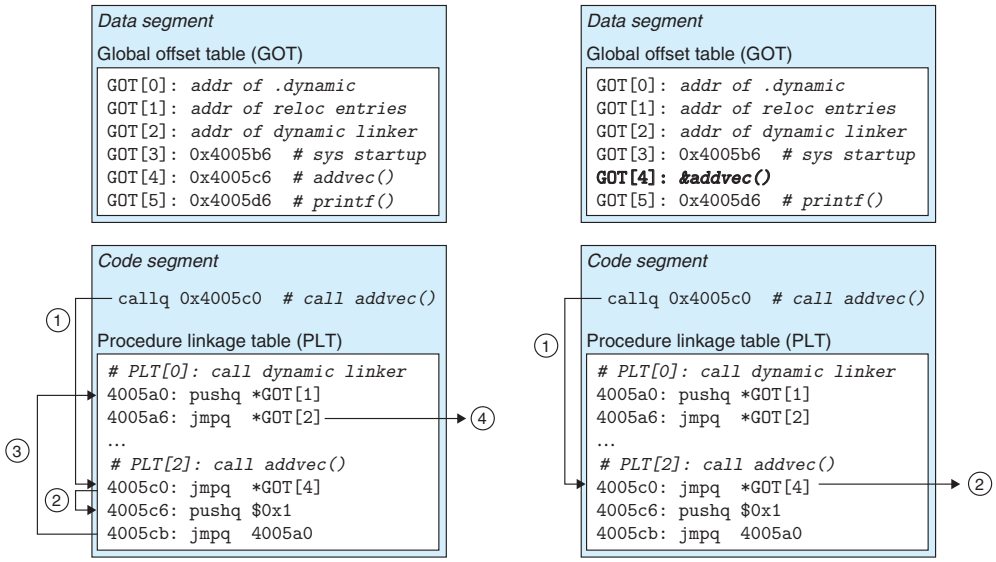
the dynamic linker could then resolve when the program was loaded. However, this approach would not be PIC, since it would require the linker to modify the code segment of the calling module. GNU compilation systems solve this problem using an interesting technique, called *lazy binding*, that defers the binding of each procedure address until the *first time* the procedure is called.

The motivation for lazy binding is that a typical application program will call only a handful of the hundreds or thousands of functions exported by a shared library such as `libc.so`. By deferring the resolution of a function's address until it is actually called, the dynamic linker can avoid hundreds or thousands of unnecessary relocations at load time. There is a nontrivial run-time overhead the first time the function is called, but each call thereafter costs only a single instruction and a memory reference for the indirection.

Lazy binding is implemented with a compact yet somewhat complex interaction between two data structures: the GOT and the *procedure linkage table (PLT)*. If an object module calls any functions that are defined in shared libraries, then it has its own GOT and PLT. The GOT is part of the data segment. The PLT is part of the code segment.

Figure 7.19 shows how the PLT and GOT work together to resolve the address of a function at run time. First, let's examine the contents of each of these tables.

Procedure linkage table (PLT). The PLT is an array of 16-byte code entries. `PLT[0]` is a special entry that jumps into the dynamic linker. Each shared library function called by the executable has its own PLT entry. Each of



(a) First invocation of `addvec` (b) Subsequent invocations of `addvec`

Figure 7.19 Using the PLT and GOT to call external functions. The dynamic linker resolves the address of `addvec` the first time it is called.

these entries is responsible for invoking a specific function. `PLT[1]` (not shown here) invokes the system startup function (`__libc_start_main`), which initializes the execution environment, calls the `main` function, and handles its return value. Entries starting at `PLT[2]` invoke functions called by the user code. In our example, `PLT[2]` invokes `addvec` and `PLT[3]` (not shown) invokes `printf`.

Global offset table (GOT). As we have seen, the GOT is an array of 8-byte address entries. When used in conjunction with the PLT, `GOT[0]` and `GOT[1]` contain information that the dynamic linker uses when it resolves function addresses. `GOT[2]` is the entry point for the dynamic linker in the `ld-linux.so` module. Each of the remaining entries corresponds to a called function whose address needs to be resolved at run time. Each has a matching PLT entry. For example, `GOT[4]` and `PLT[2]` correspond to `addvec`. Initially, each GOT entry points to the second instruction in the corresponding PLT entry.

Figure 7.19(a) shows how the GOT and PLT work together to lazily resolve the run-time address of function `addvec` the first time it is called:

Step 1. Instead of directly calling `addvec`, the program calls into `PLT[2]`, which is the PLT entry for `addvec`.

Step 2. The first PLT instruction does an indirect jump through `GOT[4]`. Since each GOT entry initially points to the second instruction in its corresponding PLT entry, the indirect jump simply transfers control back to the next instruction in `PLT[2]`.

Step 3. After pushing an ID for `addvec` (`0x1`) onto the stack, `PLT[2]` jumps to `PLT[0]`.

Step 4. `PLT[0]` pushes an argument for the dynamic linker indirectly through `GOT[1]` and then jumps into the dynamic linker indirectly through `GOT[2]`. The dynamic linker uses the two stack entries to determine the run-time location of `addvec`, overwrites `GOT[4]` with this address, and passes control to `addvec`.

Figure 7.19(b) shows the control flow for any subsequent invocations of `addvec`:

Step 1. Control passes to `PLT[2]` as before.

Step 2. However, this time the indirect jump through `GOT[4]` transfers control directly to `addvec`.

7.13 Library Interpositioning

Linux linkers support a powerful technique, called *library interpositioning*, that allows you to intercept calls to shared library functions and execute your own code instead. Using interpositioning, you could trace the number of times a particular