Linkers and Loaders

Copyright © 2001 Thomas W. Doeppner. All rights reserved.
In this lecture we cover the topic of linking and loading. We first introduce the basic concepts, looking at the historical development of linkers and loaders. We conclude by examining how things are done in Solaris.
Linkers and Loaders

- Compilers and assemblers produce output with:
  - relocatable addresses
  - external references

- Linkers
  - resolve references to separately compiled or assembled subroutines

- Loaders
  - bind relocatable addresses to absolute addresses

The purpose of a linker is to put together a program from separately compiled (or assembled) pieces. Thus if your program calls a routine, e.g. `printf`, the linker is responsible for finding `printf` and arranging that all your references to `printf` get connected to the code for `printf`.

The purpose of a loader is to arrange matters so that a program may execute starting at some given address. The code produced by the assembler or loader may contain relocatable addresses, i.e. references to items whose exact location is not known until the program has been prepared by the loader. The loader must ensure that these references refer to the exact address at which the referenced item is finally located.

In many systems the functionalities of the linker and loader are combined. In Unix systems, for instance, the linker and loader functionality is provided by the `ld` program (called simply the "loader").
Programs often contain locations that are to refer to other locations in the program. For example, the location identified by the name \( aX \) in the slide above should refer to the location containing \( X \). Since both \( aX \) and \( X \) are global variables, \( aX \) should be initialized before the program begins execution.

This might seem to be something that the compiler should be able to do, i.e., produce code that initializes \( aX \) with the address of \( X \). However, if the code in the slide is compiled separately from \textit{main}, the location of \( X \) won't be known to the compiler (since where this code will reside in memory won't be decided until after the code is compiled). To deal with this problem, the compiler will mark \( aX \) as referring to a \textit{relocatable address}, meaning that its initial value will depend upon where a certain part of the program is placed in memory.

Similarly, the address assigned to \textit{routineB} won't be known at compile time; thus it also must also be marked as relocatable.
Relocation

• How to do it:
  – transfer vectors
  – direct editing

Two approaches to relocation are in common use. The first is to refer to things indirectly via the use of transfer vectors; the second is to directly edit executable code, setting the values of relocatable addresses once they are known.
Transfer Vectors

<table>
<thead>
<tr>
<th>Offset</th>
<th>Assembler Code</th>
<th>Value</th>
<th>Reloc Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.extrn subr1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>.extrn subr2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>main:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>movl aparm, r1</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>call subr1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>call subr2</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>ret</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>aparm:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>.long parm</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>parm:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>.skip 40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To make it possible for the loader to do relocation, the compiler and assembler must provide additional information. In the simple scheme illustrated here, associated with each assembled instruction is a flag indicating whether relocation is required. (This scheme was actually used on a computer in the '60s. Here we assume that each instruction is four bytes long and that any reference to an address is at a fixed location within the instruction (and occupies only two bytes).) For a relocatable address, the assembler provides its value relative to the beginning of the module. Then, if the relocation flag is set, the loader adds the address of the beginning of the module to this value.

The remaining issue is linkages to separately assembled procedures. The scheme used here is a bit indirect, but straightforward. The assembler produces a header that contains the length of the module, the relative location of the first instruction, and a transfer vector. This transfer vector initially consists of the names of all the external procedures called by this module. Each such name is no more than eight bytes in length and is padded on the right with blanks. Thus, for the example given in the picture, the transfer vector would consist of “subr1   ” and “subr2   ”. For each call to an external procedure in the module, the assembler produces code that directs the call to the relative location of the name of the called routine within the transfer vector. The loader, once it determines the locations of subr1 and subr2, replaces each element of the transfer vector with a branch instruction that branches to the address of the indicated procedure. Thus, the call to subr1 at offset 20 causes the return address to be pushed onto the stack and control to transfer to relative location 0 (the location of subr1 in the transfer vector). But, since the loader has replaced this with a branch to the final location of this procedure, control immediately transfers to this location. When the procedure returns, control returns to offset 24 within the module.
Here is what the code of the previous slide looks like after relocation, assuming that it was loaded into memory starting at location 100, `subr1` was loaded starting at location 200, and `subr3` at location 300.
A Direct-Editing Loader

- For each separately compiled or assembled module:
  - a header describing the sizes of the module’s components
  - an external symbol directory (ESD)
  - a relocation and linkage directory (RLD)
  - object code

The main problem with the just-described simple loader is that, though the scheme works fine for relocation and linkages to external procedures, it is not easily extended to let a module reference data that has been defined elsewhere.

One way of fixing this is to use the direct-editing approach. This scheme permits the direct linkage to external procedures (as opposed to the indirect linkage of the previous scheme) and supports references to external data. To make all of this possible, the compiler and assembler must provide additional information to the loader: 1) a header supplying the sizes of each component of the module, 2) an external symbol directory listing all symbols defined in this module and used elsewhere, as well as all symbols defined elsewhere but used in this module, and 3) a relocation and linkage directory providing instructions to the loader on how to effect relocation and linking.
In this example, `main` and `result` are declared as global, meaning that they are defined in the module but possibly referenced from elsewhere. `Sum` is declared as `external`, meaning that it is defined elsewhere but used in this module. This information is recorded in the `external symbol directory` (ESD); the relative location, relevant only for `main` and `result`, is the offset within the module of the value of these symbols. Thus, if the module is located, for example, at location 1000, the “value” of `result` is 1024, which is obtained by adding the load point of the module (1000) to the relative location of `result`. At offset 12 is a word that should contain the address of `parm`. This is initialized by the assembler to contain the offset of `parm`; the loader should adjust this to contain the address where `parm` is finally loaded. Since `parm` is located within the module `main`, what the loader should do is to add the address where `main` is loaded to the initial contents of offset 12, resulting in the final address of `parm`. The directions for doing this are given in the `relocation and linkage directory` (RLD). Its first line should be interpreted as containing the directive to add the value of the symbol `main` to the contents of offset 12 within this module. Similarly, the RLD says to add the value of the symbol `sum` (an external reference) to the contents of offset 16. The word at offset 16 is to contain the address of `sum` so the assembler initializes it to 0.

In summary, the ESD contains two types of information: it identifies those symbols defined in this module and used elsewhere (entries) and it identifies those symbols defined elsewhere and used in this module (externals). The RLD supplies instructions to the loader on how to relocate those locations that require relocation in this module.
### Example: After Relocation

<table>
<thead>
<tr>
<th>Offset</th>
<th>Assembler Code</th>
<th>Value</th>
<th>ESD</th>
<th>Symbol</th>
<th>Type</th>
<th>Relative Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>main:</td>
<td></td>
<td></td>
<td>main</td>
<td>entry</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>.globl main,result</td>
<td></td>
<td></td>
<td>result</td>
<td>entry</td>
<td>24</td>
</tr>
<tr>
<td>100</td>
<td>.extern sum</td>
<td></td>
<td></td>
<td>sum</td>
<td>external</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>push aparm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>movl asum,r0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>call r0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>.long parm</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>sum:</td>
<td></td>
<td></td>
<td>sum</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>.long sum</td>
<td>400</td>
<td></td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>parm:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>.long 10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>result:</td>
<td></td>
<td></td>
<td>result</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>.long</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here is what happens after the example has been relocated, assuming that the main routine was loaded into memory starting at location 100 and that _sum_ was loaded starting at location 400.
Consider the situation shown in the slide: we have two processes, each containing a program that calls `printf`. Up to this point in our discussion, the two processes have no means for sharing a single copy of `printf`—each must have its own. If you consider that pretty much every C program calls `printf`, a huge amount of disk space in the world could be wasted because of all the copies of `printf`. Furthermore, when each program is loaded into primary memory, large amount of such memory is wasted because of multiple copies of `printf`. What is needed is a means for programs to share a single copy of `printf` (as well as other routines).

However, sharing of code is not trivial to implement. A big problem is relocation. The code for `printf` might well contain relocatable addresses, such as references to global data and other procedures.
Relocation and Shared Libraries

1) Prerelocation: relocate libraries ahead of time
2) Limited sharing: relocate separately for each process
3) Position-Independent Code: no need for relocation

If all users of `printf` agree to load it and everything it references into the same locations in their address spaces, we would have no relocation problem. But such agreement is, in general, hard to achieve. It is, however, the approach used in Windows.

A possibility might be for the users of `printf` to share a single on-disk copy, but for this copy to be relocated separately in each process when loaded. This would allow sharing of disk space, but not of primary storage.

Another possibility is for `printf` to be written in such a way that relocation is not necessary. Code written in this fashion is known as position-independent code (PIC).
Here is an example of the use of position-independent code (PIC). Processes A and B are sharing the library containing `printf` (note that `printf` contains a call to another shared routine, `doprint`), though each has it mapped into a different location. Each process maintains a private table, pointed to by register `r1`. In the table are the addresses of shared routines, as mapped into the process. Thus, rather than call a routine directly (via an address embedded in the code), a position-independent call is made: the address of the desired routine is stored at some fixed offset within the table. The contents of the table at this offset are loaded into register `r2`, and then the call is made via `r2`. 
Sun’s Solaris operating system is a modern implementation of Unix. In this section we look at its support for linking and loading, covering the features shown on the slide. Note that in the html version of this lecture, the slide title contains a hypertext link to Sun’s documentation on linkers and libraries.
Solaris supports two kinds of libraries—static libraries, contained in *archives*, whose names end with “.a” (e.g. `libc.a`) and *shared* objects, whose names end with “.so” (e.g. `libc.so`). When *ld* is invoked to handle the linking of object code, it is normally given a list of libraries in which to find unresolved references. If it resolves a reference within a `.a` file, it copies the code from the file and statically links it into the object code. However, if it resolves the reference within a `.so` file, it records the name of the shared object (not the complete path, just the final component) and postpones actual linking until the program is executed.

If the program is fully bound and relocated, then it is ready for direct execution. However, if it is not fully bound and relocated, then *ld* arranges things so that when the program is executed, rather than starting with the program’s main routine, a runtime version of *ld*, called *ld.so*, is called first. *ld.so* maps all the required libraries into the address space, completes the linkages, and then calls the main routine.
Here’s an example of the use of shared libraries. The current directory contains only a static version of `libpriv1`, both a static and a shared version of `libpriv2`, and only a shared version of `libpriv3`. Using the `cc` command, we compile and link our program `prog`. The “-L.” means look in the current directory (“.”) for libraries, and the various “-l” parameters name libraries to use (with the initial “lib” and the “.a” and “.so” suffixes stripped off). The `cc` command, after compiling the program, invokes `ld` to do the initial linking. Since there is only a static version of `libpriv1`, it uses that. For `libpriv2` it has both the static and shared versions, but unless told otherwise it uses the shared version. For `libpriv3` it has no choice but to use the shared version. `Ld` marks the resulting output file, `prog`, with the names of the desired shared libraries (`libpriv2.so` and `libpriv3.so`), but when the program is run, it immediately fails—the runtime loader, `ld.so.1`, does not know where to find the shared libraries. So, we recompile and relink `prog`, this time supplying the “-R.” parameter, which tells `ld` to put into `prog` an indication that `ld.so.1` should look inside of the current directory to find the shared libraries. To verify that things are OK, we use the `ldd` program, which checks to make sure it can find all the desired shared libraries.
Suppose we are tired of the usual version of `printf` and would like to supply our own. This is straightforward to do—we simply create our own library containing our new `printf` and supply that to `ld`, which always looks in the system library (`libc`) after searching all user-supplied libraries. In this case, it would normally find `printf` in `libc` (which comes in two versions: `libc.a` and `libc.so`), but, because of our inclusion of `libmystdio`, it takes `printf` from there.

Thus we have a means for substituting one version of a library for another. The Sun documentation refers to this as interposition, but we use this term for something else.
Substitution (2)

- I want to provide a new version of `read` for my use
- However, I don’t want this to affect standard programs that call `read` (e.g., `fread`)
- Solution:
  - standard programs call `_read`
  - the symbol `read` is defined to be identical to `_read` unless superseded by another definition:

  ```
  #pragma weak read = _read
  ```

Substitution can be a bit more complicated that it would seem at first glance. Suppose I want to substitute my version of a routine for the standard version wherever I use it, but I’d like the standard version to be used when it is called from within the system library (`libc`). For example, unless I substitute for it explicitly, I am not supposed to be able to change the semantics of `fread`. However, `fread` calls `read`, and I can change `read`. How do I make certain that my changes to `read` do not affect `fread`?

Solaris provides the notion of a weak symbol: I give such a symbol an initial value (via the pragma as shown in the slide). If I do nothing else to change the value of the symbol, it remains as defined. But if I give it a different value, for example, by supplying a new version of `read`, then the symbol takes on the new value. (If I hadn’t defined it to be a weak symbol, the linker would complain about multiple definitions.) So, in our example, what I normally refer to as `read` is actually defined as `_read`, along with the weak symbol `read` defined to be identical to `_read`. Thus, unless I redefine `read`, whenever I refer to it I get `_read`. If I redefine it, it takes on its new value, but system-supplied programs such as `fread` call `_read` directly and thus are not affected by my redefinition.
Another issue is dealing with multiple versions of the same library. I may have written one program to use the old version of a shared library. I then create a new version of the library that is different enough from the old version to be incompatible (e.g., I may have added new arguments to some of its members). I don’t want to have to be bothered with referring explicitly to a library’s version number when I link with it (by default, I want the current version), but I also want to make sure that when I run my old programs, they are dynamically linked (at run time) with the appropriate old version of the library.

In the slide, we first create the old version of the library. (The “-G” flag instructs ld to create a shared object; the “-h” flag supplies a name to be included in the shared object; the “-pic” directs the compiler to produce position-independent code.) The name of the library has a “.1” at the end, indicating version number 1. Next we use the ln command to create a soft link in the file system from the name libmystuff.so (without the version number to libmystuff.so.1)—this link is used to refer to the current version of the library, which at the moment is version 1).

Now we compile our program and link with version 1 of the library. Here we are compiling and linking a C++ program. As with cc (for C programs), we use the “-R” flag to tag the executable with the name of the directory where the shared object is found at runtime. However, for CC, there must be a space between the “R” and the directory name (sigh).

Next we remove the old link to version 1 of the shared object, create version 2 of the shared object, and set a new link to version 2. We then compile and link a new program with the new version of the library (merely by referring to libmystuff.so). Finally we use ldd to convince ourselves that when we run the old program and the new program both will be dynamically linked to the proper version of the library.
Position-Independent Code

• Data references
  – global offset table (got)
  – completely filled in by ld.so when program starts up

• Procedure references
  – procedure linkage table (plt)
  – partially filled in by ld.so when program starts up
  – each entry is completed only when used for the first time

To provide position-independent code, Solaris employs two data structures: the global offset table and the procedure linkage table. The former is used to effect linkages to data items—it is completely filled in by the runtime linker when the program starts up. The latter is used to effect linkages to procedures—it is partially filled in by the runtime linker when the program starts up. Each entry is completed as needed when the procedure it refers to is called for the first time.
To establish position-independent references to global variables, the compiler produces, for each module, a *global offset table*. This is stored at some fixed position from the beginning of the module; when a module is entered (i.e. called), a sequence of instructions using PC-relative addressing puts the address of the table in a register. (This address can be found with PC-relative addressing because the offset of the table from the beginning of the module is known.) When the program is exec'd, the *ld.so* program fills in the table with the actual addresses of the global variables referenced by the module. References to these variables within the module are made by loading a register with the contents of the slot in the table (whose address is a fixed offset from the beginning of the table) and then referencing the variable via this register.
When a program starts up, the runtime linker finds and maps into the address space all of the necessary shared objects. However, it defers until later the task of linking references to procedures to their actual locations. Instead, for each procedure defined in a shared object, it links all references to an entry in the object’s procedure linkage table that causes the procedure call instead to invoke the runtime linker, which in turn finds the procedure and then modifies the PLT entry to transfer to the procedure directly on subsequent calls.

The code for this is a bit complicated, as shown in the picture for the SPARC architecture: On the first call to a shared procedure (in this case, `printf`), control transfers to `.plt101`, the PLT entry associated with `printf`. Code here stores the offset of this entry in global register `g1` (to identify which entry was called), then branches to the beginning of the table where a call to the runtime linker is encountered. The linker, when entered, determines, by the identification number following the instruction that called it, from which PLT it was called. It then, using the value in `g1`, looks up in a symbol table the procedure that should be invoked (`printf`). It changes the PLT entry to call `printf` directly the next call, then calls `printf`, arranging so that `printf` returns directly to the original caller.
Dynamic Linking

```c
proc(...) {
    void *handle;
    int (*fptr)(int);
    int i;

    ...

    handle = dlopen("mylib.so", RTLD_LAZY)
fptr = dlsym(handle, "my_func");
i = (*fptr)(17);

    ...
}
```

Solaris also allows shared objects to be loaded into a program in the middle of execution. The example shows how one first loads the object, then, via the call to `dlsym`, how one determines at what location a particular procedure in the shared object has been loaded, and finally how one calls the procedure.
Finally we look at the notion of *interpositioning*. The idea is to be able to position a procedure between the caller and the ultimate callee, i.e., to put a “wrapper” around a procedure. The problem is that both the wrapper and the procedure have the same name. The solution is to give the linker the shared objects for both, but to supply the one containing the wrapper first. This causes calls to the procedure to go to the wrapper. Inside the wrapper, as shown in the slide, one calls the runtime linker to obtain the address where the next definition of the item of the same name resides.