Binary Dodo

Things should be made as simple as possible, but not any simpler – Albert Einstein

LINUX, PROGRAMMING

Symbol resolution during link-editing

Date: July 1, 2016 Author: Arun 0 Comments

Below we discuss symbol resolution during the **link-editing phase**. Basically there are 3 symbol types:

- defined
- undefined
- tentative

Globally, we can view the symbol resolution process during link-editing as follows. Basically the task of the link-editor is to resolve undefined and tentative references in relocatable object files. It will need to analyze the symbol tables of the input relocatable and shared object files to find appropriate definitions for those references. If for a particular symbol it finds a single definition amongst the input files, there is no ambiguity, and that definition can be taken to resolve all references to the symbol. However, if more than one definition exist, it will need to resolve between them and choose one. The factors that determine this choice include the type of file that a symbol is found in, the binding of the symbol and the type of output file being produced (see discussion below). Probably the way all this is implemented in the linker is as follows. The linker maintains an internal symbol table of all symbols that it encounters when parsing through the symbol tables of each input file in turn. For each symbol that it reads from an input file's symbol table, it searches its internal table for that symbol. If it has not been added to the internal table yet, it is added. If the symbol is already present in the internal table, the resolution process is called to determine which of the two to keep: the one that is already in the internal table, or the one that has just been read from the input file. Since when deciding between two symbols, the resolution process needs to know the type of file that the symbols came from, the internal symbol table needs to

maintain this information for each symbol that is stored in it.To resolve two symbols with the same name (i.e., to choose a value for the symbol), the type of file that provides each symbol is important:

- between two relocatable object files
- if both are not symbol definitions, use a simple precedence relationship to choose one symbol.
- If both are definitions, the binding of the symbols will determine what happens.
- between a relocatable object and a shared object dependency
- if the symbol from the shared object is a definition and the one from the relocatable file is a tentative or undefined symbol, precedence can be applied naturally, and the symbol is resolved to the definition in the shared object file.
- if the symbol from the shared object is an undefined reference and the one from the relocatable file is a definition, here also precedence relationship takes its normal course, and the relocatable file's symbol is taken. For consistency, both the link-editor and the runtime linker agree to this fact, because indeed, the undefined reference will actually be resolved at runtime by the runtime linker (since it is found in a shared object), and they need to resolve the symbol in the same manner. The link-editor's role with undefined references in shared objects is to simply verify that the symbol is resolvable (see undefined symbols below).
- if both symbols are definitions, the symbol resolves to the one from the relocatable object file: binding is irrelevant here. This is a form of interposition.
- between two shared object dependencies
- if one is a reference and the other a definition, the definition is taken as per precedence rules.
- if link-editor finds more than one definition of a symbol in two shared object files, the linker chooses the definition of the shared file that is found first. Note that binding is irrelevant here. This is a form of interposition.

Simple resolutions

- Simple resolution type 1 – symbol references from one object resolved to symbol definitions within another object
- involves the use of a precedence relationship. This relationship has **defined** symbols dominate **tentative** symbols, which in turn dominate **undefined** symbols. That is to say, an undefined symbol reference can be resolved by either a defined or a tentative symbol, and a tentative symbol can be resolved by a defined symbol.
- can occur between 2 relocatable objects

- can occur between a relocatable object and a shared object dependency
- between relocatable objects, multiple definitions of the same symbol, but with different bindings, a global symbol will silently override a weak symbol. If they have same binding, however, and the binding is global, it is a fatal error – multiple definitions of same symbol.
- Simple resolution type 2 – interposition: multiple definitions between a relocatable object and a shared object, or between two shared objects
- binding is irrelevant here
- in multiple definitions between a relocatable file and a shared file, the relocatable file symbol wins
- in multiple definitions between shared object files, the first definition found wins. By taking the first definition, this ensures that both the link-editor and the runtime linker behave consistently. Consider a relocatable file that contains an undefined reference to a symbol A. Since we are dealing with shared objects, the output file will likely either be a shared object or a dynamic executable (it is not possible to specify shared object files as input during static linking). The link-editor will list the shared object dependencies in the output file in the order they were processed during link-editing (this order itself will be dictated by the order the files were listed on the command line during invocation of the linking command). Suppose, there are 3 shared object dependencies, in the order, libshared1.so, libshared2.so, libshared3.so. The link-editor finds, during link-editing, that symbol A is defined by both libshared1 and libshared2. It will use the first symbol value in the output file. Now, during execution, the runtime linker notices that libshared3 also has an undefined reference to symbol A that needs to be resolved. Since the runtime linker will also choose the first definition that it finds, that is from libshared1.so, we have a consistent memory location for symbol A throughout the program.

Complex resolutions

Complex resolutions need to be done when different symbols of the same name are encountered that have differing attributes. Attributes may be the size, type etc of the symbol.Some examples:

- We define an array in file1.c of size 4 bytes, and an array of the same name in file2.c of size 8 bytes, the linker will give a warning, and will likely take the definition of the largest array.
- Similarly, if we define a function and a data item both with the same name, we will have two symbols with differing elf types: OBJT and FUNC. In this case the rules in simple resolution above can apply: if both are global and from relocatable files, it is a fatal error; if it is between a relocatable and a shared file, the symbol from the relocatable file is taken; between multiple shared files, the first definition is taken; if between relocatable files, and one is global, the other weak, the global definition is taken.

Undefined symbols

After all input files have been parsed, the link-editor scans its internal symbol table to see if any undefined symbols remain. Undefined symbols can affect the link-edit process according to the type of symbol, together with the type of output file being generated.

- When generating an executable output file
- if an undefined symbol in a relocatable file is not matched by any definition, a fatal error occurs
- similarly, if a shared object is used when building a dynamic executable, if the link-editor finds that an undefined symbol in the shared object cannot be resolved to a definition (the runtime linker will likely also not be able to resolve it), a fatal error occurs
- suppose a relocatable references (but does not define) a symbol U, and links directly against the shared library libfoo.so only. Libfoo.so has a dependency on another shared object, libbar.so. Libbar.so provides a definition for symbol U. So implicitly, the reference from the relocatable is resolvable to the definition from the implicit shared object dependency (implicit because it was not specified explicitly on the command line). But the linker will not allow this. What happens if a new version of libfoo comes out that is no longer dependent on libbar? The symbol U is no longer resolvable for the executable that has already been built. The system administrator replaces the file libfoo.so by the new version, and when running the program, the runtime linker no longer sees a definition a symbol U. This is why it is required to explicitly mention all dependencies on the link-editing command line, and not rely on the fact that a dependency will be loaded as part of an explicit dependency's dependency.
- When generating a shared object file
  - Undefined references are allowed to remain in a shared object file. This is to allow an executable program to provide the definition for a symbol referenced by the shared object. During runtime relocation by the runtime linker, when the latter sees an undefined reference, the default search model is to search each object for a definition, starting with the dynamic executable itself, and progressing through the dependencies in the same order that they were loaded. However, it is recommended to build self-contained shared objects such that all references to external symbols are satisfied by named dependencies. This provides the maximum flexibility as it does not require users of the shared object to determine and establish dependencies in order to satisfy the shared object's requirements.
- Weak symbols
- Weak symbol references that remain unresolved, do not result in a fatal error condition, no matter what output file type is being generated.

    - If a static executable is being generated, the symbol is converted to an absolute symbol with an assigned value of zero.
    - If a dynamic executable or shared object is being produced, the symbol is left as an undefined weak reference with an assigned value of zero. During process execution, the runtime linker searches for this symbol. If the runtime linker does not find a match, the reference is bound to an address of zero instead of generating a fatal relocation error.
    - Historically, these undefined weak referenced symbols have been employed as a mechanism to test for the existence of functionality.

- Compilation systems view this address comparison technique as having undefined semantics, which can result in the test statement being removed under optimization. In addition, the runtime symbol binding mechanism places other restrictions on the use of this technique. These restrictions prevent a consistent model from being made available for all dynamic objects.
- Undefined weak references in this manner are discouraged. Instead, you should use dlsym(3C) with the `RTLD_DEFAULT`, or `RTLD_PROBE`handles as a means of testing for a symbol's existence.

References

http://docs.oracle.com/cd/E19253-01/817-1984/chapter2-88783/index.html

C ELF LINKING & LOADING LINUX

Published by Arun

View all posts by Arun