LINUX, PROGRAMMING

Symbol binding types in ELF and their effect on linking of relocatable files

Date: May 12, 2016 Author: Arun 0 Comments
You might want to complement the reading of this article with another related article: Investigating linking with COMMON symbols in ELF

I use GCC 4.8.5 in the examples of this article.

General classification of symbols

Symbols can be classified into different categories, which determines in which section they are found in an ELF file:

- Defined symbols

These have been initialized to a value, and thus space will be reserved in the .data section of the elf file if the initial value is non-zero, or in .bss if the initial value is zero.

- Undefined symbols

These are symbols that have been referenced in the file, but which have not been defined by this file. Thus they are placed in the pseudo elf section, UND (remember this section does not exist physically).

- Common symbols

These are symbols that have not been initialized. It is a hint to the linker that during linking it can find other symbols of the same name in other object files, and all those symbols can be merged into a single symbol. Common symbols are placed in the pseudo elf section, COM, and thus do not occupy any space in the object file. Additionally, they make sense only in relocatable object files, and thus can only be found in these types of object files. As i see it, a common symbol is just a special case of an undefined symbol, because the linker will try to look for a definition. The difference with undefined symbols is that if the linker does not find a definition, then it creates the symbol in .bss in the output, so that it will initialize to zero. But if the linker finds that an UND symbol has no definition anywhere, this will be a linking error.

Symbol Bindings

ELF symbols will also have different bindings.

- LOCAL

A symbol of local binding will only be used to resolve and relocate references in its own file. Let's say a file, file1.c, contains a symbol named 'mylocal' which is static. When generating the object file, the assembler will encounter references to 'mylocal' in the program text. Since it generates all sections relative to address zero, it will need to include relocation entries for those references in the output object file. Those relocation entries will point to the 'mylocal' symbol, which will have local binding, since it was static in C code. When relocating the references to 'mylocal' in the text of file1.o, the linker will use the value of symbol 'mylocal'. But this symbol value will not be used to resolve relocations in other object files, for example if another file also has a reference to a symbol named 'mylocal'.

- GLOBAL

These symbols participate in symbol resolution between different object files. The reference from one file can be satisfied by a definition from another file.

- WEAK

Same as global, but has lower precedence than global itself. For example, if the linker notices two symbols of the same name, but one global and the other weak, it will resolve references to that symbol to the value of the global one: the weak definition is ignored.

By default C compiler will generate global variables and functions as GLOBAL symbol binding, but if the C identifier is static, then its corresponding symbol in the elf object file when compiled will have binding LOCAL. To explicitly generate a symbol with weak binding, we can use a #pragma weak directive on the identifier.

As per my tests, if we use a pragma weak on a static variable in C code, the symbol is anyways generated with a LOCAL binding, and not a WEAK binding. This makes sense, because for a symbol to be declared WEAK, first of all it has to have all characteristics of

GLOBAL; that is, the variable was intended for use in cross-file symbol resolution. A static variable does not exhibit this characteristic (since we made it static we know it will be used only in this file), so the compiler rightly sets its corresponding symbol to LOCAL binding. We know that local symbols do not participate in the symbol resolution process. So that leaves us interesting cases to investigate: what happens during linking of relocatable files, when two or more symbols of the same binding and same name are encountered, and the sections in which they are defined are same/different? Or what happens when their bindings are different (global and weak)?

Let us assume we have two files: file1.o and file2.o, and both have a symbol named 'mysym'. The table below lists some possible combinations that the linker may encounter, and how it will respond, that is what the symbol's value will be in the output file (which may be an executable, a shared object or another relocatable).

#	**Binding of symbol in file1**	**Binding of symbol in file2**	**Section of symbol in file1**	**Section of symbol in file2**	**Symbol chosen for output file**	**Binding of symbol in output**	**Section of symbol in output**
1	GLOBAL	GLOBAL	.data	.bss	Linker error: multiple definition of symbol	Linker error: multiple definition of symbol	Linker error: multiple definition of symbol
2	GLOBAL	GLOBAL	.data	COM	file1	GLOBAL	.data
3	GLOBAL	GLOBAL	.bss	UND	file1	GLOBAL	.bss
4	GLOBAL	GLOBAL	COM	UND	file1 (Space for the symbol hadn't been reserved in the relocatables, so it will be reserved now in the final output file)	GLOBAL	.bss

5	GLOBAL	GLOBAL	UND	UND	Linker error: undefined reference in both files	Linker error: undefined reference in both files	Linker error: undefined reference in both files
6	WEAK	WEAK	.data	.bss	file1 (since there are multiple definitions here and they are all weak, the linker has the freedom to choose any of them: gcc seems to take the one from the first file specified on the command line. See comments below)	WEAK	.data

7	WEAK	WEAK	.data	COM (Note: it was not possible to generate a common symbol for weak binding: the symbol was put in .bss by the compiler)	file1	WEAK	.data
8	WEAK	WEAK	.bss	UND	file1	WEAK	.bss
9	WEAK	WEAK	COM (Note: it was not possible to generate a common symbol for weak binding: the symbol was put in .bss by the compiler)	UND	file1	WEAK	.bss
10	WEAK	WEAK	UND	UND	N/A – no symbol has a valid value that can be taken	WEAK	UND

11	GLOBAL	WEAK	.data	.bss	file1	GLOBAL	.data
12	GLOBAL	WEAK	.bss	.data	file1	GLOBAL	.bss
13	GLOBAL	WEAK	.data	COM (Note: it was not possible to generate a common symbol for weak binding: the symbol was put in .bss by the compiler)	file1	GLOBAL	.data
14	GLOBAL	WEAK	COM	.data	file1 (Space for the symbol hadn't been reserved in file1, so it will be reserved now in the final output file)	GLOBAL	.bss
15	GLOBAL	WEAK	.bss	UND	file1	GLOBAL	.bss
16	GLOBAL	WEAK	UND	.bss	file2	WEAK	.bss

17	GLOBAL	WEAK	COM	UND	file1 (Space for the symbol hadn't been reserved in the relocatables, so it will be reserved now in the final output file)	GLOBAL	.bss
18	GLOBAL	WEAK	UND	COM (Note: it was not possible to generate a common symbol for weak binding: the symbol was put in .bss by the compiler)	file2	WEAK	.bss
19	GLOBAL	WEAK	UND	UND	Linker error: undefined reference in both files	Linker error: undefined reference in both files	Linker error: undefined reference in both files

Notes:

For the tests performed to obtain the results in the table above, note the following:

- In the examples above we are talking about what happens when 2 similarly named symbols are encountered in two different relocatable object files.
- A weak symbol can be generated by specifying a #pragma directive or __attribute__((weak)) in gcc.
- All examples above use only 2 symbols for illustration, but the concept extends equally to any number of symbols – see comments below for a clearer understanding.

Comments for each case in the table above

1. The linker has a confusion here: it sees two symbols of the same name that have each been explicitly initialized: one with some non-zero value, and the other with zero. It gives an error as it is not sure which value to take.
2. The COM variable merges with the initialized variable; this is what common symbols are there for after all!
3. The .bss variable resolves the undefined reference.
4. Since there is only one COM variable and no other explicit definition, the COM "merges with itself", and goes into .bss (it will be initialized to zero at runtime). The undefined reference then resolves to the newly generated .bss symbol.
5. Fairly simple. Since the linker sees no definition anywhere for the symbol, it signals an error.
6. The same test when both symbols were GLOBAL resulted in a link error. However, when both are WEAK, the linker does not give an error, but picks the symbol from the first file specified.
7. If we define a weak uninitialized symbol in C, the compiler generates it in the .bss section. Thus this case becomes similar to case 6 above.
8. The undefined symbol resolves to the .bss symbol.
9. Same as case 8 since the compiler generates a weak uninitialized variable as a .bss symbol, and not as a COM symbol.
10. Interesting to note that there is no linking error here, although we might have expected that as it was in case 5. If both symbols are weak, the linker generates an UND symbol in the executable. But note that here we are generating dynamic executables (linked by default to the dynamic version of libc by gcc), and if we were generating a static executable instead, even then we would not have any error, but the output symbol would be in .bss. In our example, since we are generating a dynamic executable, the value of the weak symbol that is generated in the output is zero. At runtime, when the runtime linker cannot find a resolution for this undefined symbol, it binds references to the symbol to the value zero, and does not generate a runtime relocation error. This 'feature' is used sometimes to test for functionality: e.g, before calling a function, we can test to see if it is equal to zero; if it zero it is undefined, so we skip the call to avoid a call to memory address zero, which will surely cause a crash. If it is non-zero, we can make the call. However, this technique is discouraged, as it may not work predictably, because of the way optimizing compilers and runtime symbol binding mechanism works. This is an interesting topic to research.
11. The GLOBAL .data is taken, the WEAK .bss is ignored. The GLOBAL symbol overrides the WEAK one.

Consider this example:

```
$> cat main.c
int un_a=108;
extern void swap();
int main() {
  swap();
  return 0;
}
$> cat swap.c
#include
#pragma weak un_a
int un_a=0;
int swap() {
  printf("Value of un_a in swap.c = %d\n", un_a);
}
$> gcc -o prog main.c swap.c
$> ./prog
Value of un_a in swap.c = 108
```

We can think of it like this: swap.o will form part of a library. The programmer who is coding swap.c says "ok, i am initializing my global variable to a certain value, but i am making it weak so that if the user of the library wishes to override the value, he can".

If both symbols were initialized to non-zero values, that is both in .data, the result would be exactly the same: the strong symbol overrides the weak one.

12. Same as case 11, except here the strong symbol has initial value 0.
13. Same as case 11 and 12, since weak common symbol is not possible to generate; the compiler generates a bss symbol.

14. Since the COM symbol is GLOBAL, and the .data one WEAK, the COM overrides the .data, and it is treated as a .bss so that its initial value will be zero.

15. Normal and simple case: the .bss resolves the undefined symbol.

16. This is a different case from the rule that GLOBAL overrides WEAK. Since the only GLOBAL here is an undefined symbol, the linker resorts to taking the WEAK symbol. Note also that in the output file, the symbol will also then have WEAK binding.

17. The undefined symbol resolves to the COM one, which itself is treated as a .bss in the final file.

18. Same as case 16. Here the COM is 'materialized' into a .bss in the output file.

19. Simple case.

Conclusions

- The binding of the symbol in the output file is the same as that of the symbol that is chosen.
- The section of the symbol in the output file is the same as that of the symbol that is chosen (if the section of the chosen symbol is COM, the output symbol's section becomes .bss).
- With GCC we cannot generate a WEAK symbol with section header index COM: it is automatically generated in .bss section.

Rough algorithm to summarize the findings above

When the linker is combining several relocatable object files, and it finds several symbols with the same name among those files, it goes through the following routine:

- Are there any GLOBAL symbols?
- If there are GLOBAL symbols, is there at least one of them that is not in section UND (i.e, they may be in .bss, .data or COM)?
  - If yes, make a list consisting of all GLOBAL symbols
  - If no, make a list consisting of all WEAK symbols (all the GLOBALs are undefined)
- If there are no GLOBAL symbols, make a list consisting of all WEAK symbols
- At this point we have a list of symbols that all have the same binding: either weak or global
- From the list that we have obtained, are there more than one symbol that is in any one of the sections .bss or .data? (only one symbol can be in .data or .bss; the others must all be UND or COM. That symbol will resolve the others)
  - If yes (there is more than one definition), are the symbols in our list of binding WEAK or GLOBAL?
    - If WEAK, take the definition of the symbol that came first from the command line input files
    - If GLOBAL, linking error
  - If no (there is only one definition), take the symbol with that definition
  - If there are no definitions at all:
    - Are the symbols in our list of binding WEAK or GLOBAL?
      - If WEAK, generate an UND symbol in the output file (see comments for case 10 above)
      - If GLOBAL, signal a linking error

The primary concern of the linker is, after all, to bind all references in the program to correct memory locations. The relocation entries are there to act as a map for the linker, as to where patches are needed in the program, which symbol provides the value for the patch, and how that value should be manipulated (relocation type) before storing it in the location specified. To obtain symbol values, the linker has to perform a first pass on all input files to create a symbol table, filling it as it reads each input file, and using the algorithm above to determine which symbol to take in case of conflicts between files (remember that the

algorithm is for conflicts between relocatable files: the mechanism used for conflicts between relocatable and shared files, or between different shared object files is much simpler: between relocatables and shared objects, the relocatable object definition is taken, and between shared objects, the first definition is taken; binding is irrelevant in both cases). For the first linking error in the algo above, if this type of error is encountered, the symbol table construction process will stop. For the second linking error, the linker may store the UND value in the internal symbol table entry for that symbol.

After having constructed the symbol table, the linker may perform the relocations one by one. When going through the relocation entries, if it encounters a reference to symbol that is still UND in the symbol table, an error is signalled and the linking stopped if the symbol is of binding GLOBAL. If the binding is WEAK, use a value of zero for the patch if we are building a static executable, or a value of UND if we are building a dynamic executable or shared object (see comments for case 10 above).

References:

I highly recommend reading this documentation from oracle's website if you really want to understand the linking and loading stuff:

Linker and Libraries Guide – http://docs.oracle.com/cd/E19253-01/817-1984/

C ELF LINKING & LOADING LINUX

Published by Arun

View all posts by Arun