Symbol Interposition and Library Load Order

Michael Kerrisk, man7.org © 2026

mtk@man7.org

January 2026

Outline

Rev: # d49d0b54be95

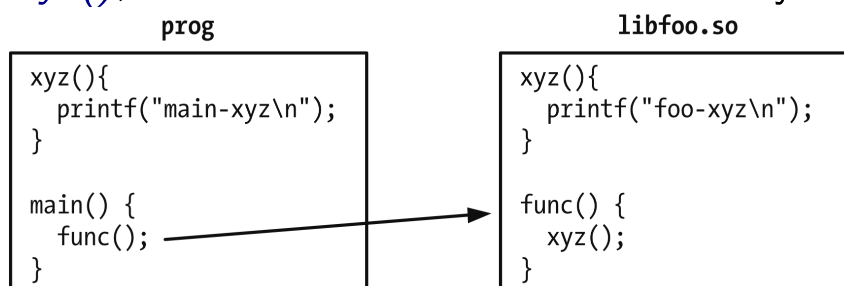
7 Symbol Interposition and Library Load Order 7-1
7.1 Symbol resolution and symbol interposition 7-3
7.2 Symbol resolution and library load order 7-9
7.3 Link-map lists (namespaces) 7-15
7.4 The global look-up scope 7-17
7.5 `LD_DEBUG`: tracing the dynamic linker 7-19
7.6 Exercises 7-25

Outline

7 Symbol Interposition and Library Load Order 7-1
7.1 Symbol resolution and symbol interposition 7-3
7.2 Symbol resolution and library load order 7-9
7.3 Link-map lists (namespaces) 7-15
7.4 The global look-up scope 7-17
7.5 LD_DEBUG: tracing the dynamic linker 7-19
7.6 Exercises 7-25

Run-time symbol resolution

- Suppose main program and shared library both define a function *xyz()*, and another function inside library calls *xyz()*



- To which symbol does reference to *xyz()* resolve?
- The results may be a little surprising:

```
$ cd shlibs/sym_res_demo
$ cc -g -c -fPIC -Wall foo.c
$ cc -g -shared -o libfoo.so foo.o
$ cc -g -o prog prog.c libfoo.so
$ LD_LIBRARY_PATH=. ./prog
main-xyz
```

- Definition in main program overrides version in library!

Symbol interposition

- When a symbol definition inside an object is overridden by an outside definition, we say symbol has been **interposed**
  - **Interposition** can occur for both functions and variables
- Behavior shown on slide 7-4 has a good historical reason...
- Shared libraries are designed to mirror traditional static library semantics:
  - Definition of global symbol in main program overrides version in library
  - Global symbol appears in multiple libraries?
    - $\Rightarrow$ reference is resolved to first definition when **scanning libraries in left-to-right order as specified in static link command line**
- Interposition behavior made transition from static to shared libraries easier

Interposition vs libraries as self-contained subsystems

- Symbol interposition semantics **conflict with model of shared library as a self-contained subsystem**
  - Shared library can't guarantee that reference to its own global symbols will bind to those symbols at run time
  - Properties of shared library may change when it is aggregated into larger system
- Can sometimes be desirable to force symbol references within a shared library to resolve to library's own symbols
  - I.e., prevent interposition by outside symbol definition

Forcing global symbol references to resolve inside library

- *–Bsymbolic* linker option causes references to global symbols within shared library to resolve to library's own symbols

```
$ cd shlibs/sym_res_demo
$ cc -g -c -fPIC -Wall foo.c
$ cc -g -shared -Wl,-Bsymbolic -o libfoo.so foo.o
$ cc -g -o prog prog.c libfoo.so
$ LD_LIBRARY_PATH=. ./prog
foo-xyz
```

  - Adds ELF `DF_SYMBOLIC` flag in `.dynamic` section of object
  - To see if object was built with this option, use either of:

```
objdump -p libfoo.so | grep SYMBOLIC
readelf -d libfoo.so | grep SYMBOLIC
```

- `DF_SYMBOLIC` flag in a library affects only the library itself (not dependencies of the library)
- More extensive example: `shlibs/demo_Bsymbolic`

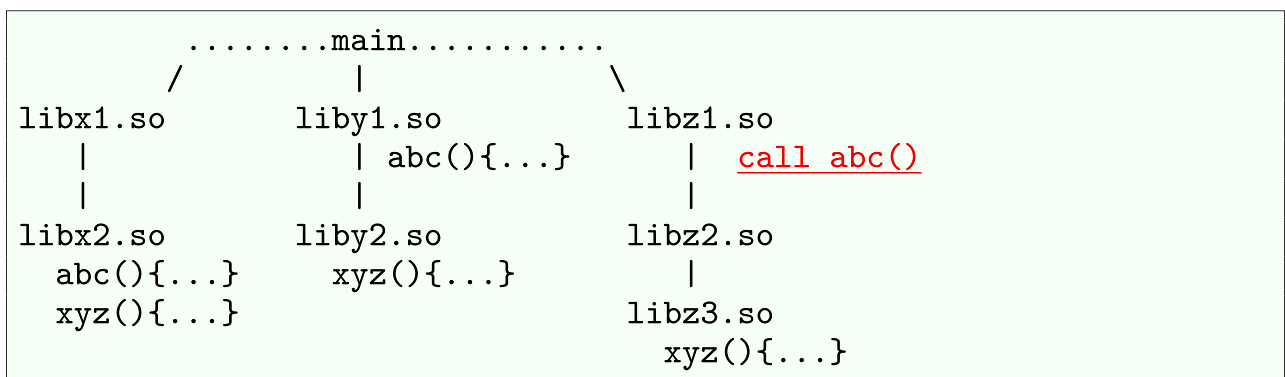
Forcing global symbol references to resolve inside library

- **Problem**: *–Bsymbolic* affects **all** symbols in shared library ☹
  - And there are other problems...
- Preferable to control "local reference binds to local definition" behavior on a per-symbol basis
  - Other techniques (described later) allow this ☺

Outline

7 Symbol Interposition and Library Load Order 7-1
7.1 Symbol resolution and symbol interposition 7-3
7.2 Symbol resolution and library load order 7-9
7.3 Link-map lists (namespaces) 7-15
7.4 The global look-up scope 7-17
7.5 LD_DEBUG: tracing the dynamic linker 7-19
7.6 Exercises 7-25

Symbol resolution and library load order

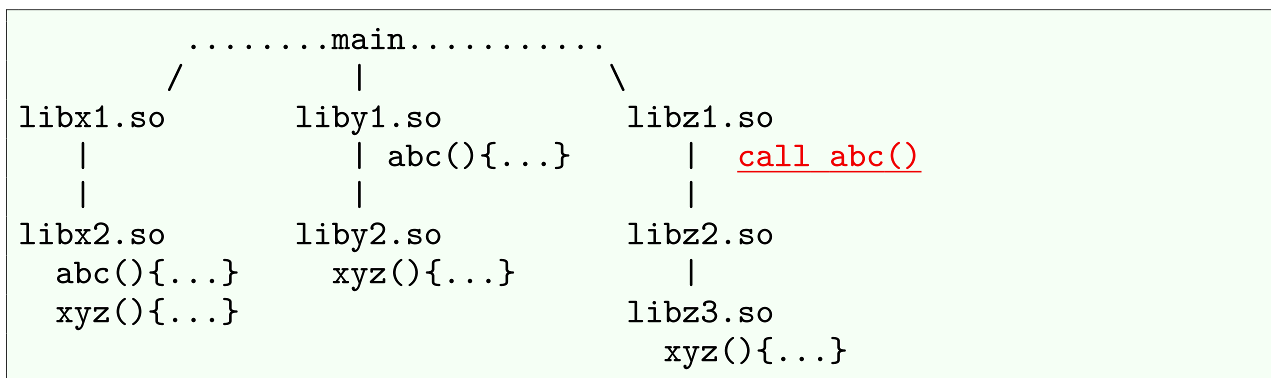


- `main` has three dynamic dependencies
    - Order of the dependencies was determined by order in link command line

```
cc main.o libx1.so liby1.so libz1.so -o main
```

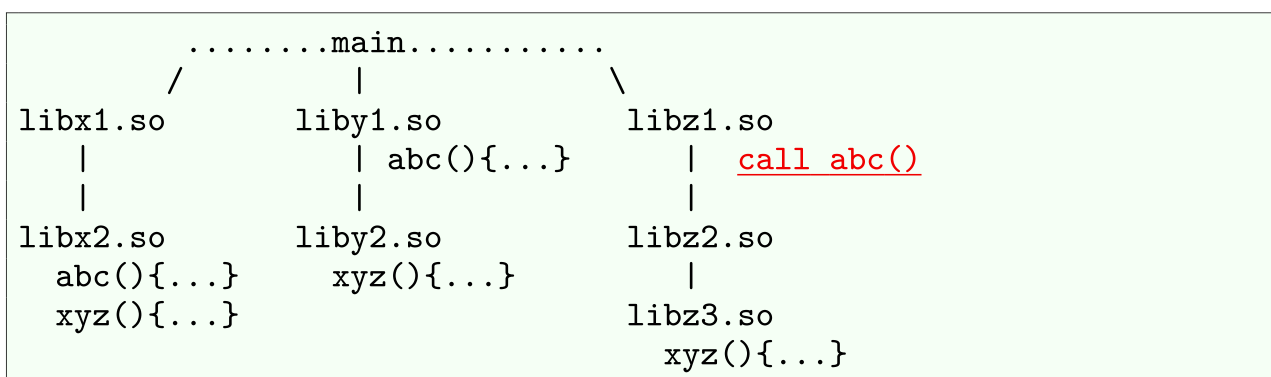
man7.org

Symbol resolution and library load order



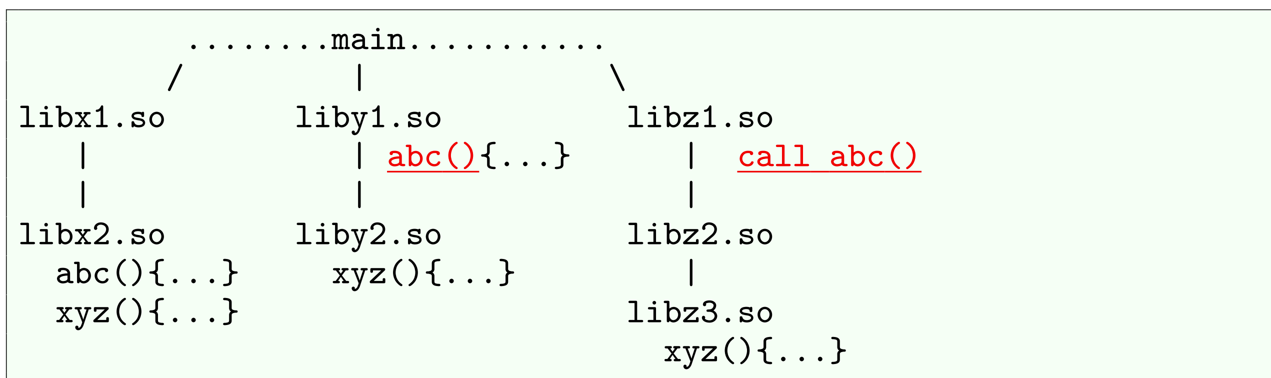
- Some of the libraries in turn have dependencies
- **Note**: `main` has no direct dependencies other than `libx1.so`, `liby1.so`, and `libz1.so`
  - Likewise, `libz1.so` has no direct dependency on `libz3.so`

Symbol resolution and library load order



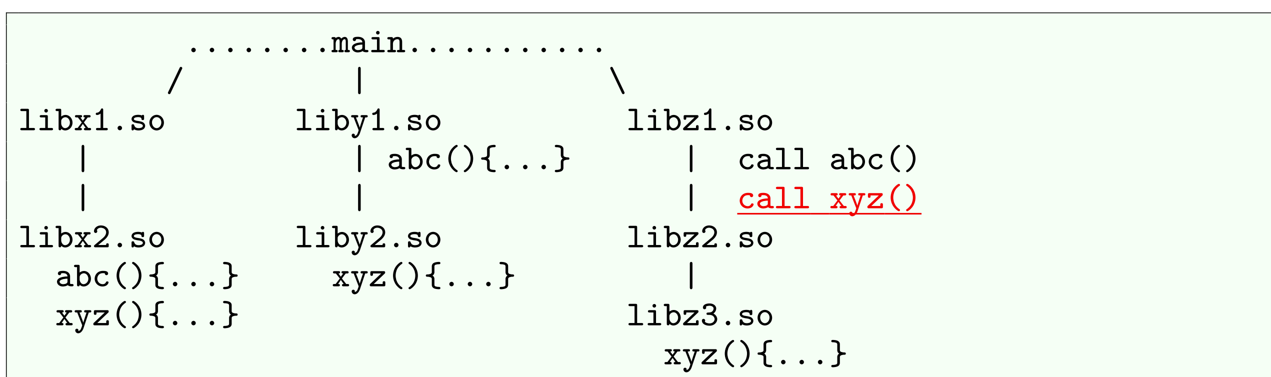
- `libx2.so` and `liby1.so` both define public function *abc()*
- When *abc()* is called from inside `libz1.so`, which instance of *abc()* is invoked?
  - Call to *abc()* resolves to definition in `liby1.so`

Symbol resolution and library load order



- Dependent libraries are added in **breadth-first order**
  - Immediate dependencies of `main` are loaded first
    - (In order given in link command line)
  - Then dependencies of those dependencies, and so on
    - Libraries that are already loaded are skipped (but are reference counted)
- Symbols are resolved by searching libraries in load order

Symbol resolution and library load order



- A quiz...
- `libx2.so`, `liby2.so`, and `libz3.so` all define public function *xyz()*
- When *xyz()* is called from inside `libz1.so`, which instance of *xyz()* is invoked?
  - Call to *xyz()* resolves to definition in `libx2.so`

Outline

7 Symbol Interposition and Library Load Order 7-1
7.1 Symbol resolution and symbol interposition 7-3
7.2 Symbol resolution and library load order 7-9
7.3 Link-map lists (namespaces) 7-15
7.4 The global look-up scope 7-17
7.5 LD_DEBUG: tracing the dynamic linker 7-19
7.6 Exercises 7-25

Link-map lists ("namespaces")

- The set of all objects that have been loaded by application is recorded in a **link-map list** (AKA "namespace")
  - Doubly linked **list that is arranged in library load order**
    - Main program is at front of link-map list
  - See definition of `struct link_map` in `<link.h>`
  - *dl_iterate_phdr(3)* can be used to iterate through list
    - Example program: `shlibs/dl_iterate_phdr`

(See also *dlinfo(3)*, which obtains info about a dynamically loaded object)

Outline

7 Symbol Interposition and Library Load Order 7-1
7.1 Symbol resolution and symbol interposition 7-3
7.2 Symbol resolution and library load order 7-9
7.3 Link-map lists (namespaces) 7-15
7.4 The global look-up scope 7-17
7.5 LD_DEBUG: tracing the dynamic linker 7-19
7.6 Exercises 7-25

The global look-up scope

- In most cases, symbol resolution is performed via an ordered search of objects listed in the **global look-up scope** (GLS)
- GLS is a list of following objects (in this order)
  - The main program
  - All dependencies of main, loaded in breadth-first order
  - Libraries opened with *dlopen(RTLD_GLOBAL)*
- Order of objects in GLS is similar to link-map list order
  - (There can be some differences when *dlopen()* is used)
- In some cases, symbol look-ups may search additional scopes
  - E.g., "local" scope and "self" scope
  - See discussion of *Look-up scopes* (later)

Outline

7 Symbol Interposition and Library Load Order 7-1
7.1 Symbol resolution and symbol interposition 7-3
7.2 Symbol resolution and library load order 7-9
7.3 Link-map lists (namespaces) 7-15
7.4 The global look-up scope 7-17
7.5 LD_DEBUG: tracing the dynamic linker 7-19
7.6 Exercises 7-25

The LD_DEBUG environment variable

- LD_DEBUG can be used to trace operation of dynamic linker
  - `LD_DEBUG="value" prog`
    - *value* is one or more words separate by space/comma/colon
  - Ignored (for security reasons) in privileged programs
  - To send trace output to file (instead of *stderr*), use `LD_DEBUG_OUTPUT=path`
  - To list LD_DEBUG options, without executing program:

```
$ LD_DEBUG=help ./prog
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit
```

The LD_DEBUG environment variable

- `libs`: show locations where each library is searched for
- `files`: emit message as each library is opened
- `reloc`: emit message at start of relocation processing for each object
- `symbols`: for each symbol relocation, show which library symbol tables are searched
- `bindings`: for each symbol relocation, show object containing definition to which symbol binds
  - Corresponds to final entry shown by `symbols` (unless symbol is undefined)
- `versions`: display version dependency checks that are performed for each object
  - Relates to symbol-versioned libraries

The LD_DEBUG environment variable

- All of the preceding `LD_DEBUG` values also cause DL to display messages when:
  - Each object's constructors and destructors are executed
  - On transfer of control to *main()*
- In addition, there are the following `LD_DEBUG` values:
  - `scopes`: display search scopes for symbol relocation (objects that will be searched during relocation for this object)
    - See the discussion of *Look-up scopes* (later)
  - `unused`: used to implement "`ldd -u`" (in conjunction with setting `LD_TRACE_LOADED_OBJECTS=1`)

LD_DEBUG example

(Abridged) example of output from LD_DEBUG:

```
$ LD_DEBUG="reloc symbols bindings" ./prog
...
32150:  relocation processing: ./prog
...
32150:  symbol=x;  lookup in file=./prog [0]
32150:  symbol=x;  lookup in file=./libdemo.so.1 [0]
32150:  binding file ./prog [0] to ./libdemo.so.1 [0]: normal symbol `x'
```

- "relocation processing" message from reloc
  - One message per library
- "symbol...lookup in file" messages from symbols
  - One group of messages for each symbol relocation
- "binding file...symbol" message from bindings
  - One message for each relocated symbol, showing origin of reference, object containing defn, and symbol name
- Number at start of each line is PID of process

LD_DEBUG example

Another LD_DEBUG example:

```
$ LD_DEBUG=scopes date
21945:
21945:  Initial object scopes
21945:  object=date [0]
21945:   scope 0: date /lib64/libc.so.6 /lib64/ld-linux-x86-64.so.2
...
```

- LD_DEBUG=scopes shows look-up scopes of each loaded object
- Here, we see the global look-up scope that is visible to the executable object, "date"

Outline

7 Symbol Interposition and Library Load Order 7-1

7.1 Symbol resolution and symbol interposition 7-3

7.2 Symbol resolution and library load order 7-9

7.3 Link-map lists (namespaces) 7-15

7.4 The global look-up scope 7-17

7.5 LD_DEBUG: tracing the dynamic linker 7-19

7.6 Exercises 7-25

Exercises

The files in the directory `shlibs/sym_res_load_order` set up the scenario shown earlier under the heading *Symbol resolution and library load order* (slide 7-14). (You can inspect the source code used to build the various shared libraries to verify this.) The `main` program uses *dl_iterate_phdr()* to display the link-map order of the loaded shared objects.

1. Use *make(1)* to build the shared libraries and the main program, and use the following command to run the program in order to verify the link-map order and also to see which versions of *abc()* and *xyz()* are called from inside `libz1.so`:

```
LD_LIBRARY_PATH=. ./main
```

2. Run the program using `LD_DEBUG=libs` and use the dynamic linker's debug output to verify the order in which the shared libraries are loaded, and which locations are searched for each library.

```
$ LD_DEBUG=libs LD_LIBRARY_PATH=. ./main 2>&1 | less
```

Exercises

3. Run the program and use the dynamic linker's debug output to show which libraries are searched and what definitions are finally bound for the calls to *abc()* and *xyz()*.

```
$ LD_DEBUG="symbols bindings" LD_LIBRARY_PATH=. ./main 2>&1 | less
```

4. The order in which the dependencies of `main` appear in the global look-up scope is determined by the order that the libraries are specified in the link command used to build `main`. Verify this as follows:
   - **Modify the last target** in the `Makefile` to rearrange the order in which the libraries are specified in the command that builds `main` to be: `libz1.so liby1.so libx1.so`
   - Remove the executable using `make clean`.
   - Rebuild the executable using `make`.
   - Run the executable again, and note the difference in symbol binding for the call to *xyz()* and the differences in the link-map order that is displayed by *dl_iterate_phdr()*.

man7.org

This page intentionally blank