Linking and Loading - Part III

Godmar Back

Virginia Tech

October 17, 2024

Software Engineering Aspects

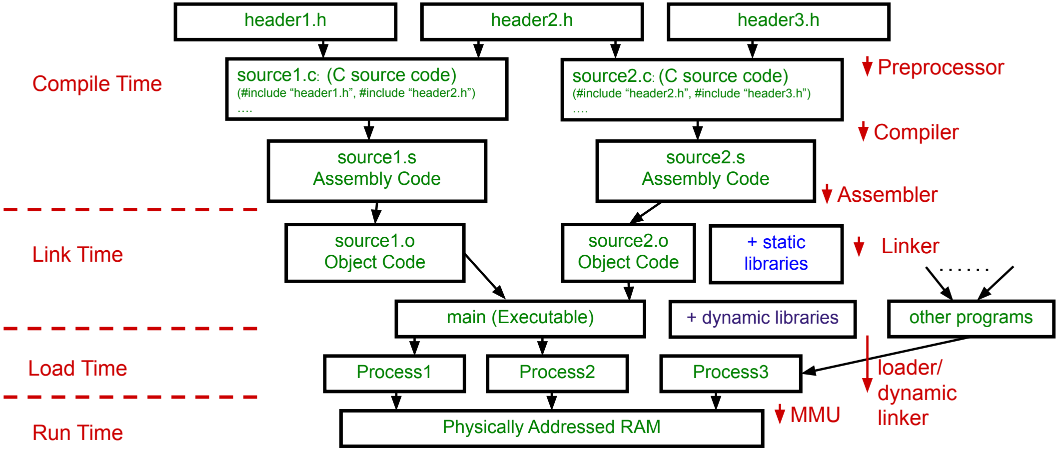

Figure 1: Compilation, Linking, and Loading in a typical System

Static Libraries

- Idea: precompile commonly used functions into object .o modules, then package those .o modules into .a archives called *static libraries*
  - .a archives are maintained by the `ar(1)` command, which creates simple sequential archives
  - NB: since .o modules are either included in their entirety, or not at all, typical libraries such as the C library (libc) or the Math library (libm) thus contain thousands of .o files
- Questions
  - how does the linker select which .o modules to include in the linking process?
  - how are dependencies handled between libraries, i.e., if there is an external reference R in module1.o in library A that is defined in module2.o in library B?
  - how expressive/powerful are static libraries as a package system?

VIRGINIA TECH

A closer look at the linking process (sans libraries)

- The linker processes .o modules in the order given on the command line
- Maintains set D of global symbols that have been defined by some already processed module
- Maintains set U of global symbols that have been referenced by some already processed module but for which no definition was seen yet
- For each .o module processed, add new external references encountered to U unless they are already in D
- Add to D, and remove (if applicable) from U the global symbols defined by this .o module (if already in D, report "multiply defined" error)
- If at the end there are any symbols left in U, report "undefined symbol" failure
- **Side note:** this discussion applies to global symbols only. Local symbol references are always resolved from the corresponding local symbol definition (which exists if the code compiled correctly)

VIRGINIA TECH

Extending the linking process to static libraries

- Rule: when processing a library, the linker will include a .o module from this library if and only if it defines a symbol that is currently in set U
  - "currently" refers to the position in the processing order given on the command line
  - .o files in the same library that define symbols referenced by other .o modules in the library are included
- Advantages:
  - Include only those .o files that are needed
  - Can override a library symbol by specifying a definition in a library that will be listed first
- Disadvantages:
  - Linking behavior depends on the exact order in which .o files and libraries are listed on the command line
  - May be necessary to list libraries in a certain order (classic `-lXm -lXt -lX11`), or even multiple times if they have mutual dependencies, or use special linker grouping option (`--start-group`/`--end-group`)
  - Error prone and confusing
    - Linker maps help to track down how the linker resolved symbols

VIRGINIA TECH

Linking Static Libraries Visualized

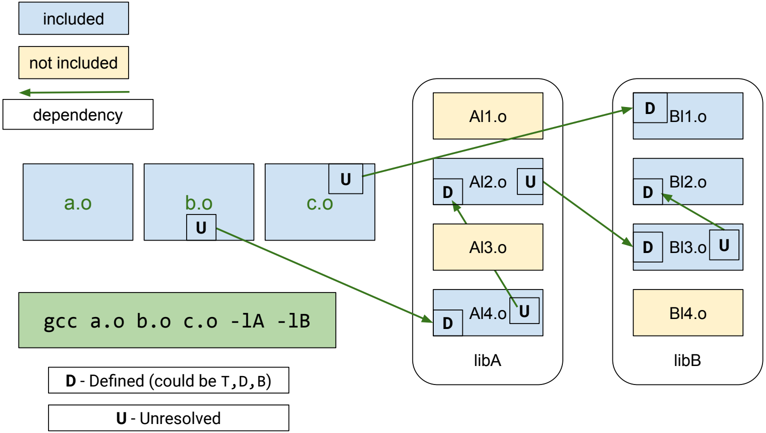

Figure 2: Selection of modules when linking with static libraries

Drawbacks of Static Libraries

- Duplicate code if functionality is used by many programs
  - e.g., the C library
  - Cost in terms of storing larger executables in the file system
  - Cost in terms of needing more memory for each process that loads these executables; inability to share this memory between processes even if they make use of the same library
- Any updates requires recompilation (and redistribution) of each executable that uses the code in question
  - Costly to push updates to system libraries
- Side Note: the inverse is that statically linked binaries come with all dependencies included, and will work as long as the underlying OS supports the system call API/ABI (Linux still runs binaries built in the 1990's)

Shared Libraries

- aka shared objects (.so), or on Windows as dynamic-link libraries (DLL)
- are loaded into a process's virtual address space at run time
- this is implemented by cooperation of the build tools with the dynamic linker/loader (ld-linux.so/ld-linux-x86-64.so in Linux)
  - the executable still contains external references (U) that will be resolved at load time
  - recursive: a dynamically linked library may in turn have dependencies
- also directly accessible via `dlopen()` for programs wishing to load shared objects at run time, as done in plugin-based systems or applications
  - flexible API, see [1] for details
- such shared objects' memory can shared by multiple processes, even if located at different virtual addresses (memory must be read-only and content not be dependent on the position at which it is mapped)
- retains (mostly) the same semantics as if the program and libraries had been linked statically

Implementation of Shared Libraries

- Position-Independent Code (handles intra-library references)
  - 64-bit x86: PC-relative addressing mode (`$rip`)
  - 32-bit x86: requires "PC materialization" trick to obtain value of `$eip`
- Indirection (needed for inter-library references, or references from executable to library)
- If a library defines global function f or variable x, the addresses f and &x are not known until the library is loaded
  - Solution: indirect function calls (via entries in PLT (Procedure Linkage Table))
  - On-demand loading via trampolines: first access triggers jump into dynamic linker
  - subsequent jumps go straight to loaded function
- In general, shared libraries introduce a marginal cost at runtime

VIRGINIA TECH.

References

[1] David M. Beazley, Brian D. Ward, and Ian R. Cooke.
The inside story on shared libraries and dynamic loading.
Scientific Programming, pages 90–97, Sep/Oct 2001.