Linkers & Loaders

Las Vegas C/C++ User Meetup

Josh Lynn

The Build Process

- Source code
  - *.c, *.cpp, *.s files
- Compilation step
  - Run the compiler on the source code and produce an object file
  - Output from this step contains:
    - Variable symbols
    - Function symbols
    - Global data symbols
- Linking process
  - Analyzes compilation output to produce another file by combining multiple object files
  - Produces an executable file, library file, or another object file

Linking

- The process of resolving “symbols” contained in a compilation unit object file (typically a ‘.o’ file)
- Symbols are defined as:
  - Defined “external” symbols: “public” or “entry” symbols which allow it to be called from other modules
  - Undefined “external” symbols: reference outside modules where these symbols are defined
  - Local symbols: used internally to the object file to facilitate relocation
- Organize the layout of the memory in the executable
  - Organize layout of data memory
  - Organize layout of code memory
  - Von Neumann architecture

Linking (cont.)

- Static Linking
  - Performed at build time
  - Leads to larger executable
  - No external runtime dependencies
  - The linker copies all the required code from the library into the final binary

Linking (cont.)

- Dynamic Linking
  - Performed at runtime
  - Can be easier to update runtime dependencies (just send out a new DLL/SO)
    - Applying the changes only requires re-running the application
  - Can be slightly slower
  - Build time linker will produce information in the executable on where and how to load the dynamic libraries. And where to find the symbols within the dynamic libraries when they're loaded
  - Commonly used for system wide libraries, like the C standard library, that are used across multiple applications.
  - Compiled with the `-shared -fPIC` flags on *nix to produce a shared, position independent library (.so) file

Linking (cont.)

- Linkers typically produce a lib or exe files with a well-defined format
- Sections of the linker output (ELF file):
  - .text
    - Code
  - .data
    - Initialized global variables
  - .bss
    - Block Storage Start; uninitialized global variables
  - .symtab
    - Symbol table with information about functions and global variables defined and referenced in the program

Linking (cont)

- ELF sections continued
  - .rel.text, .rel.data
    - relocation information for global variables, and data referenced but not defined in the current module
  - .debug
    - debugging symbol table with entries for local and global variables. Present only if the compiler is invoked with a -g option
  - .line
    - mapping between line numbers in the original C source program and machine code instructions in the .text section
  - .strtab
    - string table for the symbol tables in the .symtab and .debug sections

Loading

- The program responsible for loading programs and dependent libraries
- OS dependent
  - Embedded systems usually do not have a loader, but rely on a hardware bootloader
- Basic process
  - Validate the given executable
  - Load the dependencies of the executable (i.e. shared libraries
  - Allocate and initialize hardware, memory, and libraries
  - Start running the program from a known starting-point symbol location.
- During the loading process, the loader can try to relocate libraries, code, and data in memory.
  - “Relocating loaders” are more common in older systems like OS/360 than modern systems with virtual memory

Loading (cont.)

- Unix Loader Requirements
  - Validation
  - Loading the program into main memory
  - Copy the command line arguments to the program onto the stack
  - Initialize registers (especially the stack pointer)
  - Jump to the program starting point (i.e. _start symbol in the executable file)

Loading (cont.)

- Windows 7 (and newer) Loading Requirements
  - Initialize structures in the loader DLL (ntdll.dll)
  - Validation of executable to load
  - Create the memory heap
  - Allocate environment variables and PATH block
  - Addition of executable and DLL to module list
  - Load KERNEL32.DLL into main memory
  - Load executable’s imports (DLLs) into main memory recursively
  - Initialize all the DLLs that have been loaded
  - Start garbage collection for loader environment (and .NET runtime if necessary)
  - Start running the executable
- Most of the modern Windows loader is geared towards .NET

Questions?

References

John R Levine. 2000. *Linkers and Loaders.* Academic Press/Elsevier, San Diego, CA.

Sandeep Grover. 2002. *Linkers and Loaders* (November 2002). Retrieved 11 September 2018 from https://www.linuxjournal.com/article/6463