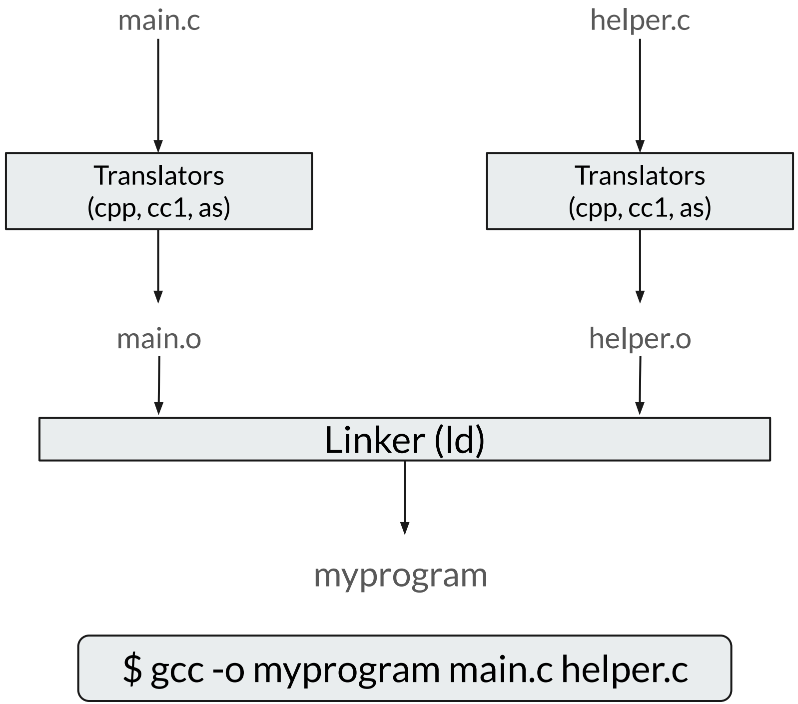
Linking, Compiling, and Make filing...

CS 410 Software Systems (Spring 2024)

Slides: Anton Njavro

Compilation Pipeline

- Our program takes a lengthy path from being an ASCII source file to getting executed in memory.
- Understanding the nuances of this process enables us to build large programs and avoid very subtle, yet difficult bugs.
- **Linker** helps us combine various chunks of data and code into an executable file that can be loaded in memory and executed.
- Often hidden in simple programs due to **compiler drivers**.

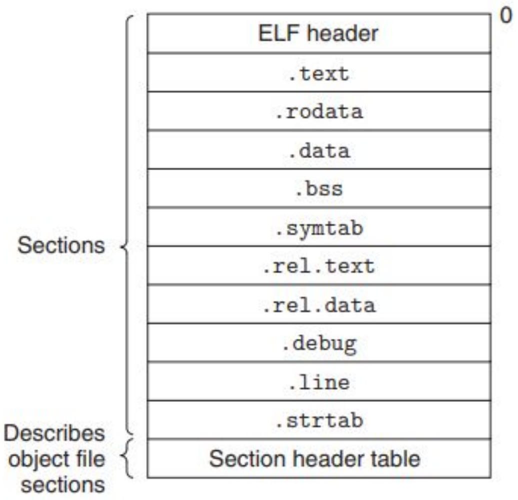


Static Linking

- GNU Linker (ld) takes as an input a set of **relocatable object files** and command-line arguments and generates a **executable object file** which can be loaded by **loader.**
- Relocatable object files are made of different code and data sections, each section itself is a contiguous sequence of bytes.
- Linker has two main tasks:
  - Symbol Resolution
  - Relocation
- Linkers have minimal understanding of target machine.

Object Files

- Three main forms of object files are:
  - **Relocatable object file**
  - **Executable object file**
  - **Shared object file**
- Compilers and assemblers generate **relocatable object files.**
- Linker creates **executable object file.**
- There are formats guiding how object files are organized: Windows (PE), MacOS (Mach-O), Linux (ELF).
- First UNIX systems from Bell Labs used (**a.out**)



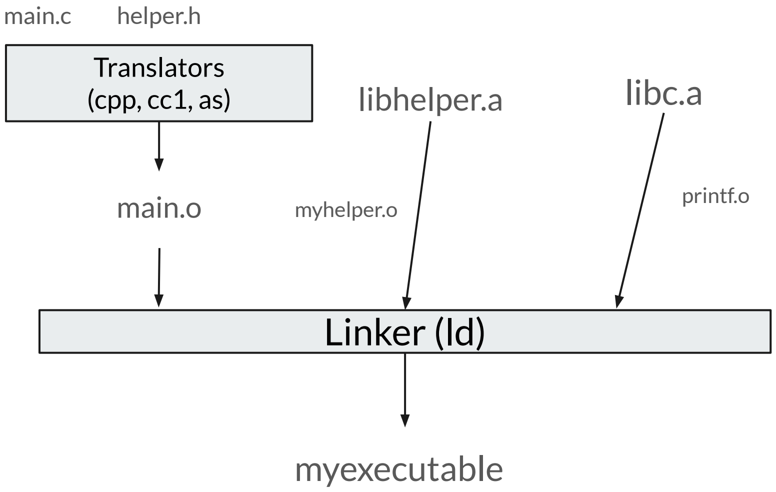
ELF Relocatable object file

Linking with Static Libraries

- How should we go about building larger-scale programs?
- Have one large source file with every function defined and referenced within it? Split it among multiple files and always pass them along to linker?
- That would be bloated and clunky!
- There exists a mechanism to package related object files into a single file called **static library**.
- We provide just the library name to the linker, and linker decides which exact object files from it are needed.
- Consider ISO C99 library (libc.a) and its very commonly used functions (printf, atoi, scanf,rand...)
- What are the alternatives to this approach and their pitfalls?

More on Static Libraries...

- As mentioned, at link time the linker copies only the object modules that are referenced by the program. (Reduces the size of executable on disk and in memory)
- Programmer only needs to include library name, no need for specific object module reference.
- C compiler driver **always passes libc.a to the linker!**
- In Linux static libraries are stored on disk in format called **archive.**
- Archive: Collection of concatenated relocatable object files, with a header describing the location and size of each member.
- First we would compile needed library modules, and then we can use **ar** command to join them in a library.



Executable Object Files

- Format is similar to that of relocatable object file.
- ELF Header still describes overall format of the file. It includes entry point as well which is the address of first instruction to be executed.
- .rodata, .data, and .text sections have been relocated to their run-time memory addresses.
- There are no more .rel sections.
- Contiguous chunks of executable file should be mapped to contiguous memory segments.
- Instructions for that are contained in **program header table.**

Dynamic Linking with Shared Libraries

- While useful, static libraries still have some issues. That's where dynamic libraries come to shine…
- If there is a change to static library, programmer must become aware of it and perform relinking.
- Another issue is unnecessary code duplication for common functions (e.g printf).
- **Shared library** is an object module that at either run time or load time can be placed in memory and linked with program memory.
- For that we need what are known as **dynamic linkers.**
- In any given file system, there is only one (**.so**) for a library, and its code and data are shared by all of the executable object files that reference it.
- Single copy of **.text** can be shared among different processes.
- Idea is to do some linking statically when executable is being created, then do rest of it dynamically at runtime.

```
$ gcc -shared -fpic -o libvector.so addvec.c multvec.c
```

Loading and Linking Shared Libraries from Program

- Applications are also able to request from dynamic linker “on the fly” to load and link arbitrary shared libraries, **while the application is running.**
- Usecases:
  - Distributing software
  - High-performance Web servers
  - Many others…
- Linux provides simple interface to dynamic linker via: **dlopen(), dlsym(), dlclose(), dlerror()**
- **dlopen()** loads and links shared library.
- **dlsym()** takes a handle to the open shared library and a symbol name (variable/function) and returns the address of the symbol (Returns NULL if it wasn’t found).

Makefiles

- Makefiles are a way in which we can organize and execute build process. It comes especially handy when working on larger more modular projects.
- Command **make** invokes instructions from **Makefile** file located in same directory.
- Makefile contains recipes on how to build specific targets and what dependencies are needed.
- Well written Makefile can optimize on knowing which dependencies got altered, therefore avoiding compilation for files that were not altered.
-

Macros

- These are the substitutions defined towards the top of the makefile usually. (e.g **CFLAGS = -g -Wall)** used to specify compile flags for GCC. Or (**CC = gcc**) used to specify GCC as C compiler.
- They are similar to **#define** statements in C, and should be used for any expression which is likely to be used repeatedly in a makefile. Once a macro has been assigned, we can reference it later using **$(MACRO_NAME)**

Targets

- The target name is generally the name of the file that will be produced when this target is built.
- The first target listed in a makefile is the default target, meaning that it is the target which is built when make is invoked with no arguments.
- Other targets can be built using **make [target-name]** at the command line.
- The Make utility will then examine the timestamps of each file on which the parent target depends.

```
target-name : dependencies
    action
```

References:

- Stanford Guide to Makefiles: (https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_make.html)
- Computer Systems A Programmer's Perspective (Third Edition) (Bryant, O'Hallaron)