

CS 241: Systems Programming

Lecture 30. Dynamic Libraries

Spring 2020

Prof. Stephen Checkoway

Last time

Static libraries (or archives) are a way of bundling a collection of object files together

- Use the compiler to create `.o` files
- Use `ar` to create `.a` file
- For each program, we want to create, use the `.a` at the end of the link line

```
$ clang -o prog main.o libfoo.a
```

Using a library: -l (lower case L)

We specify a library using a command line option: -l

- `$ clang -o prog main.o -lfoo`

Using -lfoo tells the linker to look for the file

- `libfoo.a` — a static library
- `libfoo.so` — a dynamic library on ELF-based systems
- `libfoo.dylib` — a dynamic library on macOS

Dynamic libraries

Like static libraries, dynamic libraries start as a collection of object files (.o)

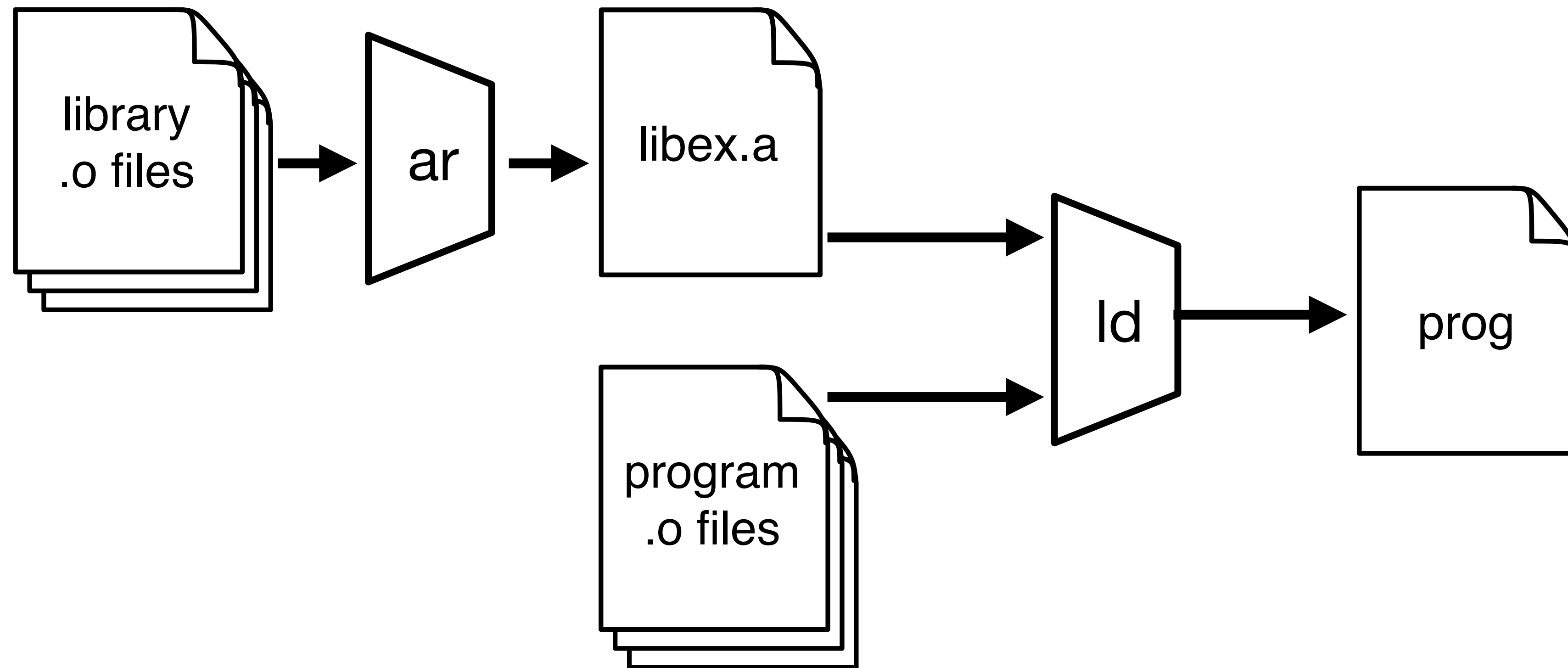
- When linking an executable against a static library, the **program linker** copies the relevant library code/data into the output

Unlike static libraries, dynamic libraries are produced by the linker

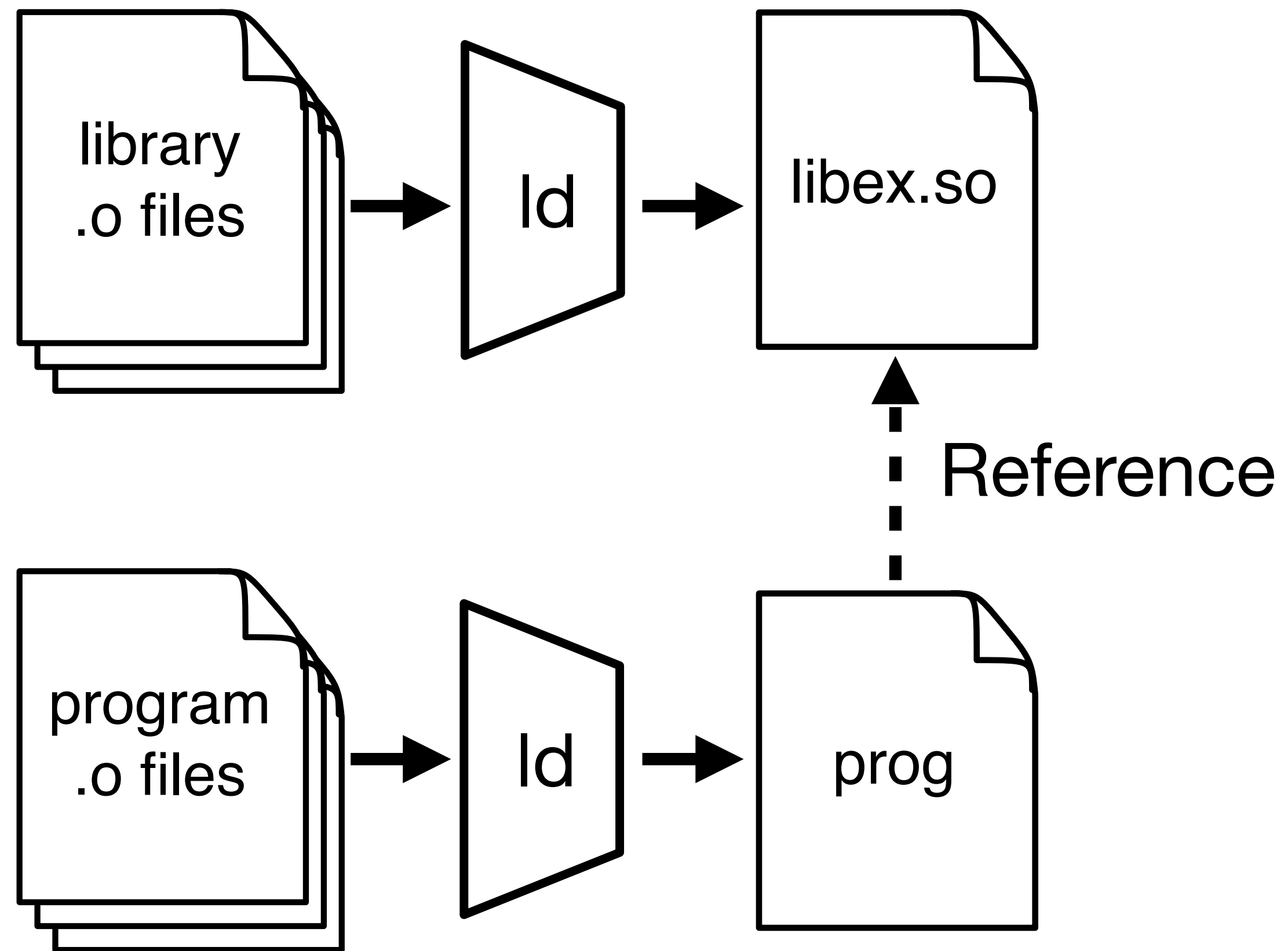
- When linking an executable against a dynamic library, the **program linker** inserts references to the library into the output, but does not copy the library code/data into the output

At run time the **dynamic linker** (the loader) loads the executable and all of its required libraries into memory

Static library



Dynamic library



Differences at runtime

Programs linked to static libraries

- Library code/data is part of the program
- Only the object files needed are included
- Code/data is placed at a known fixed address (or offset)
- Each such program has its own copy of the code/data

Programs linked to dynamic libraries

- Library code/data is loaded into memory separately
- The whole library is included, not just the needed bits
- Library code/data is loaded at a semi-arbitrary address
- Multiple programs can share a single copy of library code and read-only data; they need their own copy of the writable data
- The program loader needs to perform more work at program start up

When a library is used by many applications (e.g., libc), which of the following is **not** a benefit of using a **dynamic** library as compared to using a static library?

- A. Smaller memory usage for an individual application
- B. Smaller total memory usage across multiple applications
- C. Smaller total disk usage across multiple applications
- D. Faster program linking

When a library is used by only one application, which of the following is **not** a benefit of using a **static** library as compared to a dynamic library?

- A. Smaller memory usage for the application
- B. Smaller disk usage for the application
- C. Faster program startup
- D. Better program performance (it runs faster) separate from its start up speed
- E. Bugs in the library can be fixed independently of the application

Creating a **foo** shared object

Steps

- ▶ Object files need to be compiled as position-independent code (PIC)
`$ clang -fPIC -o a.o a.c`
- ▶ The compiler/linker needs to be informed that it's producing a shared object with a given soname (see next slide)

```
$ clang -fPIC -shared -Wl,-soname=libfoo.so.1 \  
    -o libfoo.so.1.0.0 *.o
```

Option details

- ▶ `-fPIC` — produce position-independent code
- ▶ `-shared` — produce a shared object
- ▶ `-Wl,-soname=blah` — pass `-soname=blah` to the linker

soname (ELF-based systems)

Each dynamic library has a **soname**

- ▶ `lib<name>.so.<ABI version>`
- ▶ ABI is application binary interface
- ▶ The soname specifies the name of the library and its ABI version
- ▶ Multiple versions of a library with a compatible ABI have the same soname
- ▶ Versions of a library with incompatible ABIs (different functions or parameters) have a different soname
 - `libc.so.5`
 - `libc.so.6`

soname vs. file name (Linux)

Example sonames

- zlib (a compression library) has the soname `libz.so.1`
- libc's soname is `libc.so.6`
- PCRE's library's soname is `libpcre.so.3`

On the file system the soname is a symbolic link to the actual library

- The file name is *usually* `lib<name>.so.<major>.<minor>.<patch>`
- The major version number is often the ABI version
 - `libz.so.1` -> `libz.so.1.2.11`
 - `libpcre.so.3` -> `libpcre.so.3.13.3`
 - `libc.so.6` -> `libc-2.27.so` <- Nonstandard name!

One additional symbolic link

For a given library **foo**, there are typically two symbolic links

- ▶ `libfoo.so` -> `libfoo.so.1.0.0`
- ▶ `libfoo.so.1` -> `libfoo.so.1.0.0`

The first symbol link is used at link time, the second at run time

The two need not be in the same directory

- ▶ `/usr/lib/x86_64-linux-gnu/libz.so -> /lib/x86_64-linux-gnu/libz.so.1.2.11`
- ▶ `/lib/x86_64-linux-gnu/libz.so.1 -> libz.so.1.2.11`
- ▶ `/lib/x86_64-linux-gnu/libz.so.1.2.11`

Linking to a .so

We specify a library using a command line option: `-l`

- `$ clang -o prog main.o -lblah`

`libblah.so` is a symlink to `libblah.so.1.0.0` which has a soname of `libblah.so.1`

- The compiler records `libblah.so.1` in the output prog

Example: bash

We can see the library sonames recorded in a binary using the `--dynamic (-d)` option to `readelf`

```
[clyde:~] steve$ readelf -d /bin/bash | grep NEEDED
0x00000000000000000001 (NEEDED)           Shared library: [libtinfo.so.5]
0x00000000000000000001 (NEEDED)           Shared library: [libdl.so.2]
0x00000000000000000001 (NEEDED)           Shared library: [libc.so.6]
```

Compiler search paths

When the compiler searches for files, it looks in a variety of paths

- Header files come from the header search path
- Library files come from the library search path

We can add a directory to a specific search path via command line arguments to `clang`

- Headers: `-Ipath` (e.g., `-Iinclude`)
- Libraries: `-Lpath` (e.g., `-Llib`)

Example

We have a library, `foo`, we want to link against with

- headers in `foo/include`
- libraries in `foo/lib`

In our `Makefile`, we add

- `-Ifoo/include` to `CFLAGS`
- `-Lfoo/lib -lfoo` to `LDFLAGS`

Runtime search paths

When the program starts, the dynamic linker looks at the sonames recorded in the binary and looks for a file with a matching name (which is usually a symlink) and loads that library

An additional runtime path can be added to the program at link time by using `-Wl,-rpath=<path>` to add path to the list of directories searched

By using the special symbol `$ORIGIN` we can add a path relative to the directory of the program

Actual library paths for bash

We can print the paths of the libraries that will be loaded

```
[clyde:~] steve$ ldd /bin/bash
linux-vdso.so.1 (0x00007ffe065b4000)
libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f50701b5000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f506ffb1000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f506fbc0000)
/lib64/ld-linux-x86-64.so.2 (0x00007f50706f9000)
```

`linux-vdso.so.1` is a virtual dynamic library (see `$ man 7 vdso` for details)
`ld-linux-x86-64.so.2` is the actual dynamic linker (loads everything else into memory)

In-class exercise

<https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-30.html>

Grab a laptop and a partner and try to get as much of that done as you can!