CS 241: Systems Programming
Lecture 27. Static Libraries

Spring 2024

Prof. Stephen Checkoway

Code reuse is good!

Why?

Multiple forms of code reuse

Source code reuse

- Distribute source code that can be included in many programs

Binary code reuse

- Distribute binary code that can be linked into programs
- Static libraries: code linked in at compile time (actually link time)
- Dynamic libraries: code linked in at runtime

Code compilation model

Source code goes in, object file comes out

In C:

- foo.c -> foo.o

In Rust:

- lib.rs -> library_name-hash.o
- main.rs -> bin_name-hash.o
- bin/foo.rs -> foo-hash.o

Linking step combines object files and libraries into a final executable (or library)

Static libraries ("archives")

Nothing more than a collection of object files (.o) bundled together

A "foo" library composed of object files a.o, b.o, …, z.o

- Traditionally named `lib`*`foo`*`.a`
- Compile object files as normal, e.g.,

  ```
  $ clang -c -o a.o a.c
  ```
- Put them in an archive:

  ```
  $ ar crs libfoo.a a.o b.o … z.o
  ```

We can link our programs with the archive, e.g.,

- `$ clang -o prog1 prog1.o libfoo.a`
- `$ clang -o prog2 prog2.o libfoo.a`

Rust static libraries

Rust libraries are distributed as source code

Compiling a Rust project causes each library to be built as a static library

- libc -> liblibc-73ce9a2ad47cacba.rlib

Rust’s .rlibs are just standard archive files (although this is an implementation detail)

ar(1)

ar is the archive utility

- It can create archives of arbitrary files
- It can add files to or update files in an archive
- It can delete or extract files from an archive

```
$ ar crs libfoo.a a.o b.o … z
```

- `c` — create an archive
- `r` — add (with replacement) files to the archive
- `s` — create a symbol table

Linking with static libraries

The linker (which we usually invoke via the compiler driver clang or gcc) only includes object files from an archive which are "needed"

For example,

- `a.c` defines `void fun1(void);`
- `b.c` defines `void fun2(void);`
- `c.c` defines `int blah;`
- `libfoo.a` contains `a.o`, `b.o`, and `c.o`
- If the program uses `fun1()` and `blah` but not `fun2()` in its `main.c` then

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

  is essentially

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

Symbols

Symbols have

- a name — the identifier used in the program); and
- a value — an offset into a section (.text, .data, .bss, etc.)

```
$ readelf -s maze.o

Symbol table '.symtab' contains 59 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    45: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND free
    46: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND malloc
    47: 00000000000005e0   135 FUNC    GLOBAL DEFAULT    2 maze_free
    48: 0000000000000700   143 FUNC    GLOBAL DEFAULT    2 maze_get_cols
```

Symbols

Symbols have

- a name — the identifier used in the program); and
- a value — an offset into a section (.text, .data, .bss, etc.)

UND is undefined
2 is .text (in this case)

```
$ readelf -s maze.o

Symbol table '.symtab' contains 59 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    45: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND free
    46: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND malloc
    47: 00000000000005e0   135 FUNC    GLOBAL DEFAULT    2 maze_free
    48: 0000000000000700   143 FUNC    GLOBAL DEFAULT    2 maze_get_cols
```

Defined/undefined symbols

Defined symbols have a value relative to a section in the object file (or binary)

Undefined symbols are references to symbols defined in other object files (or dynamic libraries)

Archive symbol table

Maps symbols to object files inside the archive

Created using the `s` option to `ar(1)` or the `ranlib(1)` tool

Linking with static libraries

The linker maintains a list of currently undefined symbols, initially empty

For each input files (objects and archives) from left-to-right

- If it's an object file, add the contents and symbols to the program
  - Remove defined symbols from the undefined symbol list
  - Add new undefined symbols to the undefined symbol list
- If it's an archive, perform the following until no new object files are added
  - If any object file in the archive defines a symbol in the undefined symbol list, add the object file from the archive as above

Linkers add object files from archives that define currently undefined symbols in a loop.

`libex.a` contains `a.o` and `b.o`.
`prog` is linked as

```
$ clang -o prog foo.o bar.o libex.a
```

	a.o	b.o	foo.o	bar.o
Defined symbols	fun1	fun2 bar	main foo	bar
Undefined symbols	malloc free bar		bar fun1	

Which object files are linked into `prog`?

A. `foo.o`, `bar.o`, `a.o`, and `b.o`

B. `foo.o`, `bar.o`, and `a.o`

C. `foo.o`, `bar.o`, and `b.o`

D. `foo.o`, `a.o`, and `b.o`

E. `foo.o`, and `bar.o`

Duplicate symbols are an error.

`libex.a` contains `a.o` and `b.o`.
`libbar.a` contains `bar.o`.
`prog` is linked as

```
$ clang -o prog foo.o libex.a \
  libbar.a
```

	a.o	b.o	foo.o	bar.o
Defined symbols	fun1	fun2 bar	main foo	bar
Undefined symbols	malloc free bar		bar fun1	

Which object files are linked into `prog`?

A. `foo.o`, `bar.o`, `a.o`, and `b.o`

B. `foo.o`, `bar.o`, and `a.o`

C. `foo.o`, `bar.o`, and `b.o`

D. `foo.o`, `a.o`, and `b.o`

E. Duplicate symbol error

Duplicate symbols are an error.

`libex.a` contains `a.o` and `b.o`.
`libbar.a` contains `bar.o`.
`prog` is linked as

```
$ clang -o prog foo.o libex.a bar.o
```

	a.o	b.o	foo.o	bar.o
Defined symbols	fun1	fun2 bar	main foo	bar
Undefined symbols	malloc free bar		bar fun1	

Which object files are linked into `prog`?

A. `foo.o`, `bar.o`, `a.o`, and `b.o`

B. `foo.o`, `bar.o`, and `a.o`

C. `foo.o`, `bar.o`, and `b.o`

D. `foo.o`, `a.o`, and `b.o`

E. Duplicate symbol error

Moral of the story

Specify your static libraries at the end of the link line

Dynamic libraries

Dynamic libraries are produced by the (program) linker and are combined at run time by the loader (dynamic linker)

We'll talk more about them next time!