CS61 Section Notes Solutions

Week 8 (Fall 2011)

Linking

`gcc` is a *compiler driver* that invokes the actual compiler, linker, etc. as needed on behalf of the user. Consequently, there is a lot of mysterious stuff going on in the background that many programmers aren't aware of. Fortunately, you're taking CS61!

1. The Unix *Executable and Linkable Format* (ELF) is a common and representative object file format. Take a few minutes to brainstorm; what are the components of a relocatable ELF object?

It's not important that you recall or memorize all of these; just keep general concepts in mind.

header	- contains encoding information to help parse and read the file
.text	- contains the code
.rodata	- contains read-only data (e.g. static strings)
.data	- contains *initialized* global variables (has to store data in the ELF file)
.bss	- contains *uninitialized* global variables (doesn't store data in-file; saves space)
.symtab	- *symbol table:* contains information about functions and global variable refs
.rel.text	- a list of locations in .text that will need to be relocated when linked
.rel.data	- as above, but for .data
.debug	- a more complete symbol table, only present in debug compiles
.line	- a mapping between lines of C and assembly; only present in debug compiles
.strtab	- a table of strings referenced in .symtab and .debug, and for section names

2. Now, consider the following C code...

main.c	swap.c

```
void swap();

int buf[2] = {1, 2};

int main() {
    swap();
    return 0;
}
```
	```
	extern int buf[];

	int *bufp0 = &buf[0];
	static int *bufp1;

	void swap() {
	    int temp;

	    bufp1 = &buf[1];
	    temp = *bufp0;
	    *bufp0 = *bufp1;
	    *bufp1 = temp;
	}
	```

For each of the following symbols in `swap.o.symtab`, fill out the symbol type (local, global, or extern) and the module (`main.o` or `swap.o`) and ELF section where it is defined.

Symbol	Type	Module	Section
`buf`	**extern**	**main.o**	**.data**
`bufp0`	**global**	**swap.o**	**.data**
`bufp1`	**local**	**swap.o**	**.bss**
`swap`	**global**	**swap.o**	**.text**
`temp`	**Trick question!**	**Not in .symtab!**	**On the stack!**

3. Consider the following C code:

`main.c`	`swap.c`

```
int *x = 0x8000000;
int y;
char z = 'a';
float f = 3.14;
float w;
static int q;
...
```

```
int x;
static int y;
static double z;
double f = 3.1415
int w;
static int q;
...
```

In trying to link the object files of main and swap, to which version of each variable (x, y, z, f, w, and q), will the linker resolve (the swap version, or the main version)? Are there any that cannot be resolved?

x will be resolved to main's x, since strong trumps weak
y in main is not the same variable as y in swap, since swap's is local.
z is the same as y
f won't compile - you can't have two strong symbols linked together
Both w's resolve to the same 4 bytes in the .bss segment
Both q's are local, and so independent of one another

Libraries

When two or more object files are linked together, the linker must...

- *Resolve symbols:* The linker must associate each symbol *reference* with a single *definition*, just as you did in the previous problem.
- *Relocate:* Once symbols have been resolved, the linker must update each symbol reference to point to the actual location of the symbol definition in memory. These placeholders are listed in the .rel section of each ELF object file.

The linker then combines the .text, .data, etc. of each object file to make a new object file. In a sense, the new object file contains all the contents of the old object files.

Sometimes, you want to be able to *package* many objects together (such as a bunch of related printing functions), but only *copy* the objects that you actually *reference*. In this case, what you want is a *library:* a collection of conditionally-linked objects.

4. Because libraries depend on references to link, the objects that reference them must be processed by the compiler driver before the library is processed. Let a -> b denote that a depends on b; that is, b defines a symbol that a references. For each question (a) through (c), in what order must the following be processed?

   a. p.o -> libx.a

b. p.o -> libx.a -> liby.a
c. p.o -> libx.a -> liby.a **and** liby.a -> libx.a -> p.o

a. p.o, then libx.a
b. p.o, then libx.a, then liby.a
c. p.o, then libx.a, then liby.a, then libx.a (but not p.o again; why? Because p.o is not a library, so all of its contents are linked unconditionally).

Dynamic Linking

Up to this point we have discussed only *static linking*, which takes place at compile time and involves copying over the linked objects into a new executable. In contrast, *dynamic linking* against a *shared library* of code can take place at load or even run time.

5. Take a few minutes to brainstorm: what are the pros and cons of static versus dynamic linking?

Pros:

- You can update the library without having to recompile your executables.
- Different software teams can work collaboratively but independently; think about how useful drivers are!
- You can write programs that depending on the environment, user input, etc. switch between different code implementations.

Cons

- Security! Are you sure your library hasn't been altered? Also, a recently (August 2010) discovered attack can cause potentially malicious DLLs to be loaded (see http://en.wikipedia.org/wiki/DLL_Hijacking).
- Different components of your project now have differently-maintained versions. This can lead to silent bugs and serious headaches if two versions behave slightly differently.
- The library designer has to be very forward-thinking about the library interface, errors, etc., because it can't easily change.