COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

Today

- Symbols
- Libraries
- Attributes
- Common libraries
- Good library design
- More System V functions

Libraries

There are two types of libraries:

- static libraries
- shared libraries

We will talk about the differences later!

Symbols

- A Symbol is a `name` for a particular piece of data
- The name is used to refer to the data instead of using its address in memory – which may not be known
- Symbol resolution is the process of mapping from names to data (addresses)

Symbol Resolution: Linker

- The **linker** attempts to resolve symbols within a given set of object files including static libraries
- The linker will leave symbols refering to shared libraries symbolic
- The linker can store information about which libraries should be searched by the loader to resolve unresolved symbols

Symbol Resolution: Loader

- The loader resolves the remaining symbols and initiates execution
- Symbols that remain unresolved may cause runtime errors
- Usually, the compiler notices, but libraries could change after the main binary was compiled!

$\Rightarrow$ Library versioning is used to specify compatibility

Questions

?

Questions

Why do we have static and shared libraries?

What are the advantages and disadvantages of the two types of libraries?

Naming Conflicts

- The same name could be used by multiple symbols!
- This is usually a bug, resulting in linker errors
- Some symbols are defined as **weak**, specifically allowing them to be re-defined

$\Rightarrow$ Do not do this at home (wait until graduate school)

Avoiding Naming Conflicts

- Use a common unique prefix for all symbols exported by a library
- Do not use names of functions in GNU libc
- Use the `static` keyword on functions and non-local variables to ensure that they do **not** get exported
- Check that your library only exports (non-debug) symbols that you want to have exported!

Inspecting Binaries

- nm
- ldd
- file

Debugging Symbols

- Debugging symbols are used for `gdb` to determine the names of (non-exported) function names and local variables
- You can use the `strip` command to remove (debugging) symbols from a binary
- `strip` can also be used to remove other (exported) symbols, potentially rendering a library useless
- Read the man-page for details

Questions

?

GCC attributes

- Non-standard extensions of the C language
- Some are supported by other C compilers
- Few people use other C compilers
- We will discuss some of the most important ones

GCC attributes: alias

The "alias" attribute creates a second name for a symbol:

```
void the_real_fun () { /* Do something. */; }
void fun ()
  __attribute__ ((weak, alias ("the_real_fun")));
```

GCC attributes: constructor

The "constructor" attribute ensures that the function (which must not take any arguments) is run before `main` or immediately after the library is loaded. The function must not be `static`.

```
void init ()
  __attribute__((constructor)) { /* ... */ };
```

GCC attributes: destructor

The "destructor" attribute ensures that the function (which must not take any arguments) is run after `main` or immediately before the library is unloaded. The function must not be `static`.

```
void fini ()
  __attribute__((destructor)) { /* ... */ };
```

GCC attributes: deprecated

The "deprecated" attribute ensures that using the symbol will generate a compiler warning:

```
void old_function ()
  __attribute__((deprecated));
```

GCC attributes: nonnull

The "nonnull" attribute ensures that passing "NULL" for certain arguments will generate a compiler warning:

```
void * fun (int * p1, void * p2)
  __attribute__((nonnull (1, 2)));
```

GCC attributes: noreturn

The "noreturn" attribute tells the compiler that the function will never return, allowing it to generate better code:

```
void spin ()
  __attribute__((noreturn))
{
  while (1);
}
```

Common libraries

- libm: mathematical functions
- libz: compression
- libsqlite3: database
- libgmp: unbounded precision arithmetic
- libgcrypt: cryptography
- libcurl: file downloads (http, ftp, etc.)

Find thousands of libraries on `http://freshmeat.net/`.

Homework

Compile and run the following C code (which uses libm) using GCC:

```
#include <math.h>
int main(int argc, char ** argv) {
  double d = asin(0.14);
  return (int) d;
}
```

Good library design

- First, learn what is already out there!
- Often it is easier to use or improve an existing library than to roll your own
- Have at least two different clients for the library
- Export as few symbols as possible; do not export variables

$\Rightarrow$ Adding new symbols is backwards-compatible, deleting symbols is not!

Back to GNU libc

"First, learn what is already out there!"

$\Rightarrow$ Knowing GNU libc inside-out is fundamental.

Fundamental Character API

- int toupper(int c)
- int tolower(int c)
- int isspace(int c)
- int isupper(int c)
- int isdigit(int c)
- int isXXXXX(int c)

getopt

- function for parsing command line arguments
- a few variants exist (getopt_long, argp_parse)
- we will just cover the basics

getopt

```
int main (int argc, char **argv) {
  int index, c;
  while (-1 != (c = getopt (argc, argv, "ab:c:")))
    switch (c) {
    // on next slide
    }
  }
  for (index = optind; index < argc; index++)
    printf ("Non-option argument %s\n", argv[index]);
  // application code
  return 0;
}
```

getopt – switch body

```
int aflag = 0; int bnum  = 0; char * cvalue = NULL;

case 'a': aflag = 1; break;
case 'b':
  if (1 != sscanf(optarg, "%d", &bnum)) {
    fprintf (stderr,
             "Option -%c requires an argument.\n", 'b');
    abort();  }
  break;
case 'c': cvalue = optarg; break;
default:  fprintf (stderr, "Unknown option -%c.\n", c);
           abort ();
```

mmap

```
void * mmap(void *start,
            size_t length,
            int prot,
            int flags,
            int fd,
            off_t offset)}
int munmap(void *start, size_t length);
```

Example: display file content

```
#define FILENAME "/etc/services"
struct stat buf;
stat(FILENAME, &buf);
int fd = open(FILENAME, O_RDONLY);
const char * data = mmap(NULL, stat.st_size, PROT_READ,
                         MAP_SHARED, fd, 0);
printf("File ‘%s’ contains:\n%.*s",
       FILENAME,
       stat.st_size, data);
munmap(data, stat.st_size);
close(fd);
```

Questions

?

Question

Why should you consider using `mmap` instead of `read` and `write`?

Question

When would it be better to use `read` and `write` instead of `mmap`?

Question

Can you `mmap` standard-input (stdin, 0)?

Question

What about standard-output (stdout, 1)?