Computer Labs: Program Generation and Libraries

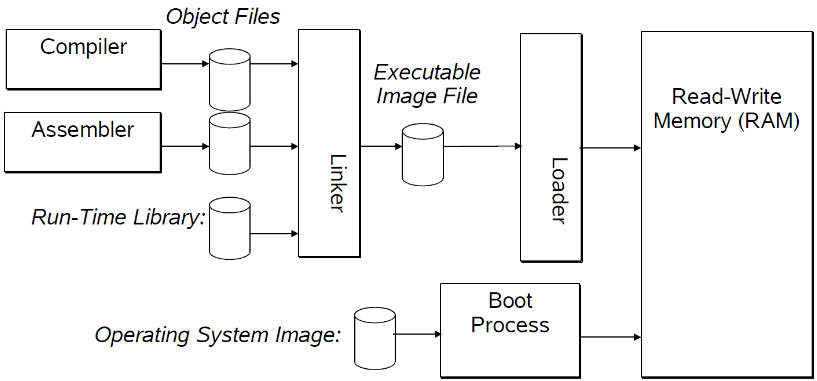
2º MIEIC

Pedro F. Souto (`pfs@fe.up.pt`)

December 5, 2012

Contents

Program Generation and Loading



Linking and Loading

Linking Essentially, this is the last step of the executable generation process.

- An executable is often generated from several object files
- Linking is the process of combining several object files in one single executable file

Loading This is one of the first steps upon execution. It consists of:

- Putting the executable to main memory. In systems with virtual memory:
  - There is no need to read the entire executable to main memory
  - It is enough to map the executable in the process's address space

It is executed by the **loader**, which is part of the operating system.

Linking Tasks

Address space allocation

- Usually, each object file is generated as if the code was placed at address 0
- The linker must rearrange the code location, and, if necessary, change absolute addresses used in jumps, loads and stores

Symbol Resolution

- The linked object files reference each other by means of symbols (names of variables/functions)
- The linker must **resolve** these cross-referencing symbols

Notes

- The GNU linker is known as `ld`
- It is able to combine object files in a single non-executable object file that can later be linked with other object files
- When invoked with the `-o` option, `gcc` invokes automatically `ld`

Program Generation: Example

foo.c

```
#include <math.h>
#include <stdio.h>

double foo(int i) {
    extern int k;
    return pow(i,k);
}
```

test.c

```
#include <stdio.h>
#include <math.h>

int k = 5;
int main(int argc, char *argv[]) {
    int n;
    n = foo(argc); // error: foo() returns a double
    printf("foo(%d) = %d \n", argc, n);
    return n;
}
```

Essentially, this program computes $\texttt{argc}^{\texttt{k}} = \texttt{argc}^5$

Compilation

Compilation

```
$ gcc -c *.c
$
```

Linking

- The executable cannot be generated from `test.c` only:

```
$ gcc test.o -o test
test.o:test.o:(.text+0x23): undefined reference to `_foo
collect2: ld returned 1 exit status
```

- Actually, both `test.o` and `foo.o` have some symbols that have to be resolved by the linker:

nm test.o

```
         U ___main
         U _foo
00000050 D _k
0000000e T _main
         U _printf
```

nm foo.o

```
         U ___floatsidf
00000000 T _foo
         U _k
         U _pow
```

Linking Continued

- One more try:

```
$ gcc test.o foo.o -o test
$
```

- What about `printf(), pow()`? We didn't define them and we didn't tell the linker where to find them
  - `printf()` belongs to the C standard library and it is automatically linked when generating an executable
  - `pow()` belongs to the C math library, which is automatically linked by `gcc` (in Linux, `gcc` is configured differently, so we must explicitly specify that the C math library needs also to be linked)
- Let's run it:

```
$ ./test 2
foo(2) = 0
```

This is not right!!! We expected to get $2^5 = 32$

What’s the Problem?

- `foo()` returns a double, but in the compilation of `test.c` the compiler assumes that it returns an integer:
  - In C, if a function is not declared/defined in a compilation unit, it is assumed to take as arguments integers and to return an integer
- The IA32 architecture uses different representations and different instructions for integral types and floating point types
  - For `foo.c` the compiler generates code that returns a `double`, but for `test.c` the compiler assumes that the value returned is an `int`, misreading the returned value
- Note that the compiler didn’t even give a warning, but it was our fault:

```
$ gcc -Wall -c test.c
test.c: In function ‘main’:
test.c: In function ’main’:
test.c:9: warning: implicit declaration of function ’foo’
```

How to Fix it?

- Must declare the type of every function before that function is invoked. E.g. in `test.c`:

  ```
  #include <stdio.h>
  #include <math.h>

  double foo(int i);
  ...
  ```
- Usually, this is done using header/include files, which are included in the source files by the C pre-processor
- In principle, for each C source file, you should define a header file with its interface
  - The declaration of external symbols defined in the source file
    - Public functions, i.e. functions that may be called by other modules
    - Public, i.e. non-static, global variables
  - The definition of symbolic constants, by means of `#define`, that may be used by the interface
- **IMP.** A header file is **NOT** a library

Naming Conflicts (gcc on Linux)

- Object files linked together cannot define the same symbol

```
foo.c:
    int f1(int i) {
         return 2*i;
    }
test.c:
    #include <stdio.h>

    int f1(int i) { //
        return 4*i;
    }
    int main() {
        printf("%d\n", f1(4));
}

$ gcc foo.c test.c -o test
/tmp/ccmvyHdi.o:/tmp/ccmvyHdi.o:(.text+0x0): multiple defi
/tmp/ccOJ5BnD.o:/tmp/ccOJ5BnD.o:(.text+0x0): first defined
collect2: ld returned 1 exit status
```

Contents

What is a Library?

Generally Is a collection of modules (files/functions/classes) that can be used to structure code, for reasons

- Modularity
- Code reuse

In C, we can think of libraries as a structuring mechanism above that of files (modules).

More narrowly

> *"A "program library" is simply a file containing compiled code (and data) that is to be incorporated later into a program; program libraries allow programs to be more modular, faster to recompile, and easier to update."*
>
> *David Wheeler*

I'd add that they also make it easier to distribute code

IMPORTANT In portuguese we should say **Biblioteca** rather than "livraria" (bookstore)

Naming Resolution and Libraries

1. If a name is defined in an object file, use that definition
   - If the same name is defined in a library, that definition will be ignored
2. If a name is not defined in an object file, use the definition found in the first library
   - The same name may be defined in different libraries
   - If the same name is defined in different libraries, its resolution will depend on the order in which the libraries are specified
3. If a name is not defined in an object file, and it is defined in different modules of a given library use the first definition found
   - The same name may be defined in different modules that comprise a library
   - That name's resolution depends on the order in which they are added to the library
   - All definitions but the first will be hidden

Static vs. Shared Libaries

Static the library code/data is included in the executable when the code is linked (i.e. at compile time)

- Program is self-contained

Shared the library code/data is not included in the executable when the code is linked.

- Some of the actions typically performed by the linker must be carried out by the loader, when the program is “brought” to memory
- Leads to smaller executables
- Makes it easier to update libraries

Static libraries

```
$ gcc -static -o stat_fact *.o
$ ls -l stat_fact
... 579173 ... st_fact
$ nm stat_fact.o
...
08048d40 T printf
...
```

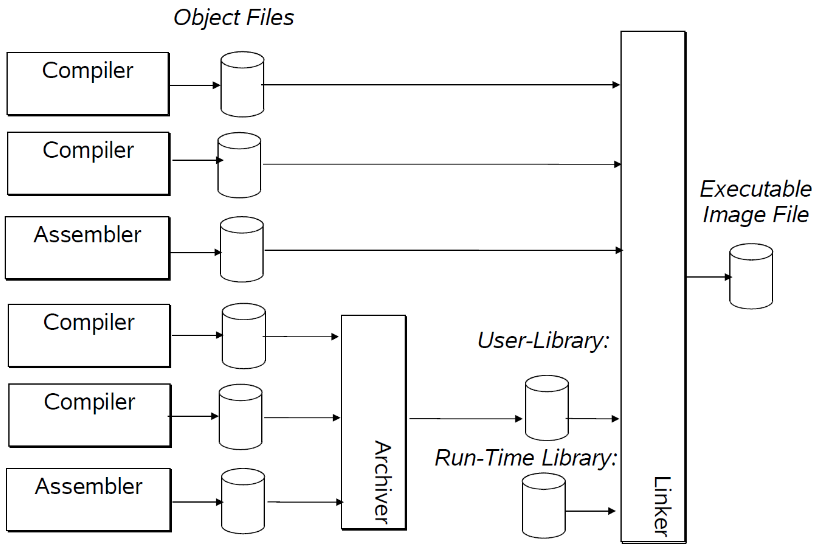
Shared libraries

```
$ gcc -o dyn_fact *.o
$ ls -l dyn_fact
...  8322 ... dyn_fact
$ nm dyn_fact
...
080483e4 T main
         U printf@@GLIBC_2.0
```

System vs. User Libaries

System libraries Those that are part of the “system”

User libraries Those that are developed by a programmer



Static Library Generation with `ar` (GNU)

- The `ar` utility can be used to generate a static library and to execute different operations on such a library

Create a library with several object files

```
$ ar cr libmy.a foo.o bar.o
```

List the object files in a library

```
$ ar tv libpong.a
rw-r--r-- 42/42   676 Mar 7 15:15 2006 unmap_video.o
...
rw-r--r-- 42/42   695 Mar 7 15:15 2006 move_block.o
```

Extract a file from an archive, does not delete it

```
$ ar x libpong.a read_xpm.o
x - read_xpm.o
```

Delete a file from an archive

```
ar dv libpong.a read_xpm.o
d - read_xpm.o
```

Add/replace a file to an archive

```
ar rv libpong.a read_xpm.o
a - read_xpm.o
ar rv libpong.a read_xpm.o
r - read_xpm.o
```

Shared (Dynamic) Library Generation

- Minix 3 does not support shared libraries currently
- In the Linux world it is easier to use `gcc` to generate a shared library
  - Although you can also invoke directly `ld` with the `-shared` option
- For details check Chapter 3 – Shared Libraries – of David Wheeler's Program Library HOWTO

Linking with a User Library

```
gcc -o pong pong.o -L. -lpong -lm
```

-l<name> means that the compiler should use a library with name `lib<name>.a` or `lib<name>.so` (shared library) to resolve symbols, if necessary

-L. specifies that the linker should search for libraries in the current directory, in addition to the default directories, which depend on the system

Alternatively, you can specify the full name of the library:

```
gcc -o pong pong.o libpong.a -lm
```

IMP. Note that the executable includes only the object files in the libraries that are needed for symbol resolution

Contents

Further Reading and Acknowledgments

Further Reading

- Brian Gough, An Introduction to GCC, Network Theory Ltd.
- David A. Wheeler Program Library HOWTO

Thanks to

- João Cardoso on whose transparencies these ones rely heavily