The True Story of Hello World

(or at least a good part of it)

Most of our computer science students have been through the famous "Hello World" program at least once. When compared to a typical application program ---almost always featuring a web-aware graphical user interface, "Hello World" turns into an very uninteresting fragment of code. Nevertheless, many computer science students still didn't get the real story behind it. The goal of this exercise is to cast some light in the subject by snooping in the "Hello World" life-cycle.

The source code

Let's begin with Hello World's source code:

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     printf("Hello World!\n");
6.     return 0;
7. }
```

Line 1 instructs the compiler to include the declarations needed to invoke the *printf* C library (*libc*) function.

Line 3 declares function *main*, which is believed to be our program entry point (it is not, as we will see later). It is declared as a function that takes no parameter (we disregard command line arguments in this program) and returns an integer to the parent process --- the *shell*, in our case. By the way, the shell dictates a convention by which a child process must return an 8-bit number representing it status: *0* for normal termination, *0 > n < 128* for process detected abnormal termination, and *n > 128* for signal induced termination.

Line 4 through 8 comprise the definition of function *main*, which invokes the *printf* C library function to output the "Hello World!\n" string and returns 0 to the parent process.

Simple, very simple!

Compilation

Now let's take a look at the compilation process for "Hello World". For the upcoming discussion, we'll take the widely-used GNU compiler (*gcc*) and its associated tools (*binutils*). We can compile the program as follows:

```
# gcc -Os -c hello.c
```

This produces the object file *hello.o*. More specifically,

```
# file hello.o
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

tells us hello.o is a relocatable object file, compiled for the IA-32 architecture (I used a standard PC for this study), stored in the Executable and Linking Format (ELF), that contains a symbol table (not stripped).

By the way,

```
# objdump -hrt hello.o
hello.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000011  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000048  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000048  2**2
                  ALLOC
  3 .rodata.str1.1 0000000d  00000000  00000000  00000048 2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000033  00000000  00000000  00000055  2**0
                  CONTENTS, READONLY

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 hello.c
00000000 l    d  .text  00000000
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000000 l    d  .rodata.str1.1 00000000
00000000 l    d  .comment       00000000
00000000 g    F  .text  00000011 main
00000000         *UND*  00000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_386_32          .rodata.str1.1
00000009 R_386_PC32        puts
```

tells us *hello.o* has 5 sections:

1. **.text**: that's "Hello World" compiled program, i.e. IA-32 opcodes corresponding to the program. This will be used by the program *loader* to initialize the *process' code segment*.
2. **.data**: "Hello World" has neither initialized global variables nor initialized static local variables, so this section is empty. Otherwise, it would contain the variable initial values to be loaded into the *data segment*.
3. **.bss**: "Hello World" also doesn't have any non-initialized variable, either global or local, so this section is also empty. Otherwise, it would indicate how many bytes must be allocated and zeroed in the *data segment* in addition to section *.data*.
4. **.rodata**: this segment contains the "Hello World!\n" string, which is tagged read-only. Most operating systems do not support a *read-only data segment* for processes (running programs), so the contents of *.rodata* go either to the process' *code segment* (because it's read-only), or to the *data segment* (because it's data). Since the compiler doesn't know the policy adopted by your OS, it creates this extra ELF section.
5. **.comment**: this segment contains 33 bytes of comments which cannot be tracked back to our program, since we didn't write any comment. We'll soon see where it comes from.

It also shows us a *symbol table* with symbol *main* bound to address 00000000 and symbol *puts* undefined. Moreover, the *relocation table* tells us how to relocate the references to external sections made in section *.text*. The first relocatable symbol corresponds to the "Hello World!\n" string contained in section *.rodata*. The second relocatable symbol, *puts*, designates a *libc* function which was generated as a result of invoking *printf*. To better understand the contents of *hello.o*, let's take a look at the assembly code:

```
1. # gcc -Os -S hello.c -o -
2.         .file   "hello.c"
3.         .section        .rodata.str1.1,"aMS",@progbits,1
4. .LC0:
5.         .string "Hello World!"
6.         .text
7.         .align 2
```

```
 8. .globl main
 9.         .type   main,@function
10. main:
11.         pushl   %ebp
12.         movl    %esp, %ebp
13.         pushl   $.LC0
14.         call    puts
15.         xorl    %eax, %eax
16.         leave
17.         ret
18. .Lfe1:
19.         .size   n,.Lfe1-n
20.         .ident  "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"
```

From the assembly code, it becomes clear where the ELF section flags come from. For instance, section *.text* is to be 32-bit aligned (line 7). It also reveals where the *.comment* section comes from (line 20). Since *printf* was called to print a single string, and we requested our nice compiler to optimize the generated code (*-Os*), *puts* was generated instead. Unfortunately, we'll see later that our *libc* implementation will render the compiler effort useless.

And what about the assembly code produced? No surprises here: a simple call to function *puts* with the string addressed by *.LC0* as argument.

Linking

Now let's take a look at the process of transforming *hello.o* into an executable. One might think the following command would do:

```
# ld -o hello hello.o -lc
ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

But what's that warning? Try running it!

Yes, it doesn't work. So let's go back to that warning: it tells the linker couldn't find our program's entry point *_start*. But wasn't it *main* our entry point? To be short here, *main* is the start point of a C program from the programmer's perspective. In fact, before calling main, a process has already executed a bulk of code to "clean up the room for execution". We usually get this surrounding code transparently from the compiler/OS provider.

So let's try this:

```
# ld -static -o hello -L`gcc -print-file-name=` /usr/lib/crt1.o /usr/lib/crti.o hello.o
/usr/lib/crtn.o -lc -lgcc
```

(if this command doesn't work, see Troubleshooting [1] and [2].)

Now we should have a real executable. Static linking was used for two reasons: first, I don't want to go into the discussion of how dynamic libraries work here; second, I'd like to show you how much unnecessary code comes into "Hello World" due to the way libraries (*libc* and *libgcc*) are implemented. Try the following:

```
# find hello.c hello.o hello -printf "%f\t%s\n"
hello.c 84
hello.o 788
hello   445506
```

You can also try "`nm hello`" or "`objdump -d hello`" to get an idea of what got linked into the executable.

For information about dynamic linking, please refer to Program Library HOWTO.

Loading and running

In a POSIX OS, loading a program for execution is accomplished by having the *father* process to invoke the *fork* system call to replicates itself and having the just-created *child* process to invoke the *execve* system call to load and start the desired program. This procedure is carried out, for instance, by the *shell* whenever you type an external command. You can confirm this with *truss* or *strace*:

```
# strace -i hello > /dev/null
[????????] execve("./hello", ["hello"], [/* 46 vars */]) = 0
...
[08053d44] write(1, "Hello World!\n", 13) = 13
...
[0804e7ad] _exit(0) = ?
```

Besides the *execve* system call, the output shows the call to *write* that results from *puts*, and the call to *exit* with the argument returned by function *main* (*0*).

To understand the details behind the loading procedure carried out by *execve*, let's take a look at our ELF executable:

```
# readelf -l hello
Elf file type is EXEC (Executable file)
Entry point 0x80480e0
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x55dac 0x55dac R E 0x1000
  LOAD           0x055dc0 0x0809edc0 0x0809edc0 0x01df4 0x03240 RW  0x1000
  NOTE           0x000094 0x08048094 0x08048094 0x00020 0x00020 R   0x4

Section to Segment mapping:
  Segment Sections...
   00     .init .text .fini .rodata __libc_atexit __libc_subfreeres .note.ABI-tag
   01     .data .eh_frame .got .bss
   02     .note.ABI-tag
```

The output shows the overall structure of *hello*. The first *program header* corresponds to the process' *code segment*, which will be loaded from file at offset 0x000000 into a memory region that will be mapped into the process' *address space* at address 0x08048000. The code segment will be 0x55dac bytes large and must be page-aligned (0x1000). This segment will comprise the *.text* and *.rodata* ELF segments discussed earlier, plus additional segments generated during the linking procedure. As expected, it's flagged read-only (R) and executable (X), but not writable (W).

The second *program header* corresponds to the process' *data segment*. Loading this segment follows the same steps mentioned above. However, note that the segment size is 0x01df4 on file and 0x03240 in memory. This is due to the *.bss* section, which is to be zeroed and therefore doesn't need to be present in the file. The data segment will also be page-aligned (0x1000) and will contain the *.data* and *.bss* ELF segments. It will be flagged readable and writable (RW). The third program header results from the linking procedure and is irrelevant for this discussion.

If you have a *proc file system*, you can check this, as long as you get "Hello World" to run long enough (hint: *gdb*), with the following command:

```
# cat /proc/`ps -C hello -o pid=`/maps
08048000-0809e000 r-xp 00000000 03:06 479202     .../hello
0809e000-080a1000 rw-p 00055000 03:06 479202     .../hello
080a1000-080a3000 rwxp 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
```

The first mapped region is the process' *code segment*, the second and third build up the *data segment* (*data* + *bss* + *heap*), and the fourth, which has no correspondent in the ELF file, is the

stack. Additional information about the running *hello* process can be obtained with GNU *time*, *ps*, and */proc/pid/stat*.

Terminating

When "Hello World" executes the *return* statement in *main* function, it passes a parameter to the surrounding functions discussed in section linking. One of these functions invokes the *exit* system call passing by the *return* argument. The *exit* system call hands over that value to the *parent* process, which is currently blocked on the *wait* system call. Moreover, it conducts a clean process termination, with resources being returned to the system. This procedure can be partially traced with the following:

```
# strace -e trace=process -f sh -c "hello; echo $?" > /dev/null
execve("/bin/sh", ["sh", "-c", "hello; echo 0"], [/* 46 vars */]) = 0
fork()                                  = 8321
[pid  8320] wait4(-1,  <unfinished ...>
[pid  8321] execve("./hello", ["hello"], [/* 46 vars */]) = 0
[pid  8321] _exit(0)                    = ?
<... wait4 resumed> [WIFEXITED(s) && WEXITSTATUS(s) == 0], 0, NULL) = 8321
--- SIGCHLD (Child exited) ---
wait4(-1, 0xbffff06c, WNOHANG, NULL)    = -1 ECHILD (No child processes)
_exit(0)
```

Closing

The intention of this exercise is to call attention of new computer science students to the fact that a Java applet doesn't get run by magic: there's a lot of system software behind even the simplest program. If consider it useful and have any suggestion to improve it, please e-mail me.

Troubleshooting

[1] cannot find /usr/lib/crt1.o

When running the command:

```
$ ld -static -o hello -L`gcc -print-file-name=` /usr/lib/crt1.o /usr/lib/crti.o hello.o
/usr/lib/crtn.o -lc -lgcc
```

You may get the following errors:

```
ld: cannot find /usr/lib/crt1.o: No such file or directory
ld: cannot find /usr/lib/crti.o: No such file or directory
ld: cannot find /usr/lib/crtn.o: No such file or directory
```

The files mentioned may be in another directory in your system. You can find them by running:

```
$ find /usr/lib -name 'crt1.o'
```

which returns, for example:

```
/usr/lib/x86_64-linux-gnu/crt1.o
```

If you don't find the files with the command above, you may need to install the libc6-dev package and then re-run the "find" command. On a 64-bit Ubuntu 14.04 machine:

```
$ sudo apt-get install libc6-dev
```

Then you need to adapt the ld command to the correct directory, for example:

```
$ ld -static -o hello -L`gcc -print-file-name=` /usr/lib/x86_64-linux-gnu/crt1.o hello.o
/usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crtn.o -lc -lgcc
```

[2] undefined reference to `_Unwind_Resume'

In newer versions of GCC, you may run into undefined references such as:

```
//usr/lib/x86_64-linux-gnu/libc.a(iofclose.o): In function `_IO_new_fclose':
(.text+0x20c): undefined reference to `_Unwind_Resume'
//usr/lib/x86_64-linux-gnu/libc.a(iofclose.o):(.eh_frame+0x1f3): undefined reference to
`__gcc_personality_v0'
//usr/lib/x86_64-linux-gnu/libc.a(iofflush.o): In function `_IO_fflush':
(...)
```

You need to include gcc's exception support library (-lgcc_eh), but that will cause circular dependencies. Use the following command:

```
$ ld -static -o hello -L`gcc -print-file-name=` /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-
linux-gnu/crti.o hello.o /usr/lib/x86_64-linux-gnu/crtn.o --start-group -lc -lgcc -lgcc_eh --end-
group
```

The --start-group --end-group flags are used to solve the mentioned circular dependencies. You can read about them with

```
$ man ld
```

FAQ

This section is dedicated to student's frequently asked questions.

- **What is "libgcc"? Why is included in linkage?**

  Internal compiler libs, such as libgcc, are used to implement language constructs not directly implemented by the target architecture. For instance, the module operator in C ("%") might not be mappable to a single assembly instruction on the target architecture. Instead of having the compiler to generate in-line code, a function call might be preferable (specially for memory limited machines such as microcontrollers). Many other primitives, including division, multiplication, string manipulation (e.g. memory copy) are typically implemented on such libraries.

- **Is there a diagram summarizing this all?**

  Yes. Check this.