

Tutorial 5

Linking and Loading

1 Objectives

This exercise aims to give some practical experience of what compilers, assemblers, linkers and loaders do on the Unix operating system. You are encouraged to work through the exercise, discussing it with friends.

2 What To Do

Copy the following files into your working directory from the course homepage.

assembler.s a very simple Intel assembly language program.

cversion.c a C version of the same program.

writext.s another assembly language file that provides some support for the C version.

example.c a slightly more complex C program, with procedures and arrays that is useful for experimenting with optimization.

Work through the remainder of this document, trying out the suggested commands.

2.1 Looking at assembler.s

The principles of Linux assembler programs are similar to other assembler programs you may have seen, though the Linux assembler has a different syntax to `msdos` assembler. Appendix A contains a summary of the differences you need to know about.

`assembler.s`

```
1      .text
2      .globl _start
3  _start:
4      movl    $0x4,%eax      # eax = code for 'write' system call
5      movl    $1,%ebx       # ebx = file descriptor to standard output
6      movl    $message,%ecx # ecx = pointer to the message
7      movl    $13,%edx      # edx = length of the message
8      int     $0x80         # make the system call
9
10     movl    $0x1,%eax     # eax = code for 'exit' system call
11     int     $0x80         # make the system call
12
13     .data
14     .globl message
15 message:
16     .string "Hello world\n" # The message as data
```

In `assembler.s`, the `int` instruction is a software interrupt (trap), used here to make requests to the Unix operating system. The value in register `eax` indicates which system call is to be made, and depending on the call, other information is passed to Linux in other registers.

3 Assembling assembler.s

The program `as` is the Unix assembler, the flag `-o` specifies the name of the output file, here `assembler.o`.

```
as assembler.s -o assembler.o
```

The resulting file contains binary machine instructions, initialized values and other data. The program called `file` identifies the *type* of a file by analyzing its contents. Enter the command `file assembler.o` to get:

```
assembler.o: ELF 32-bit LSB relocatable, Intel 80386, version 1, not stripped
```

Executable code on Linux is usually stored in a file in 'ELF' format (**Executable and Linkable Format**), Linux also recognises 'a.out' format (try using `as` without an output name). `Strip` removes information from an executable file, making it smaller.

Each executable file starts with an *elf header*. The *header* contains information about the program and what kind of machine it is meant to run on. The program entry point, the address where execution should start is stored there. The header is described by the C structure declaration taken from the include file `/usr/include/elf.h` below:

```
64 /* The ELF file header. This appears at the start of every ELF file. */
65
66 #define EI_NIDENT (16)
67
68 typedef struct
69 {
70     unsigned char e_ident[EI_NIDENT];    /* Magic number and other info */
71     Elf32_Half    e_type;                /* Object file type */
72     Elf32_Half    e_machine;            /* Architecture */
73     Elf32_Word    e_version;            /* Object file version */
74     Elf32_Addr    e_entry;              /* Entry point virtual address */
75     Elf32_Off     e_phoff;              /* Program header table file offset */
76     Elf32_Off     e_shoff;              /* Section header table file offset */
77     Elf32_Word    e_flags;              /* Processor-specific flags */
78     Elf32_Half    e_ehsize;             /* ELF header size in bytes */
79     Elf32_Half    e_phentsize;          /* Program header table entry size */
80     Elf32_Half    e_phnum;              /* Program header table entry count */
81     Elf32_Half    e_shentsize;          /* Section header table entry size */
82     Elf32_Half    e_shnum;              /* Section header table entry count */
83     Elf32_Half    e_shstrndx;          /* Section header string table index */
84 } Elf32_Ehdr;
```

The most important fields in this header are pointers to the rest of the information about the file, organized in two tables like the content pages of a book. Each table entry is a fixed sized header, which contains information about sections (parts of the file): the text (executable part of the program), data, symbol table and relocation information.

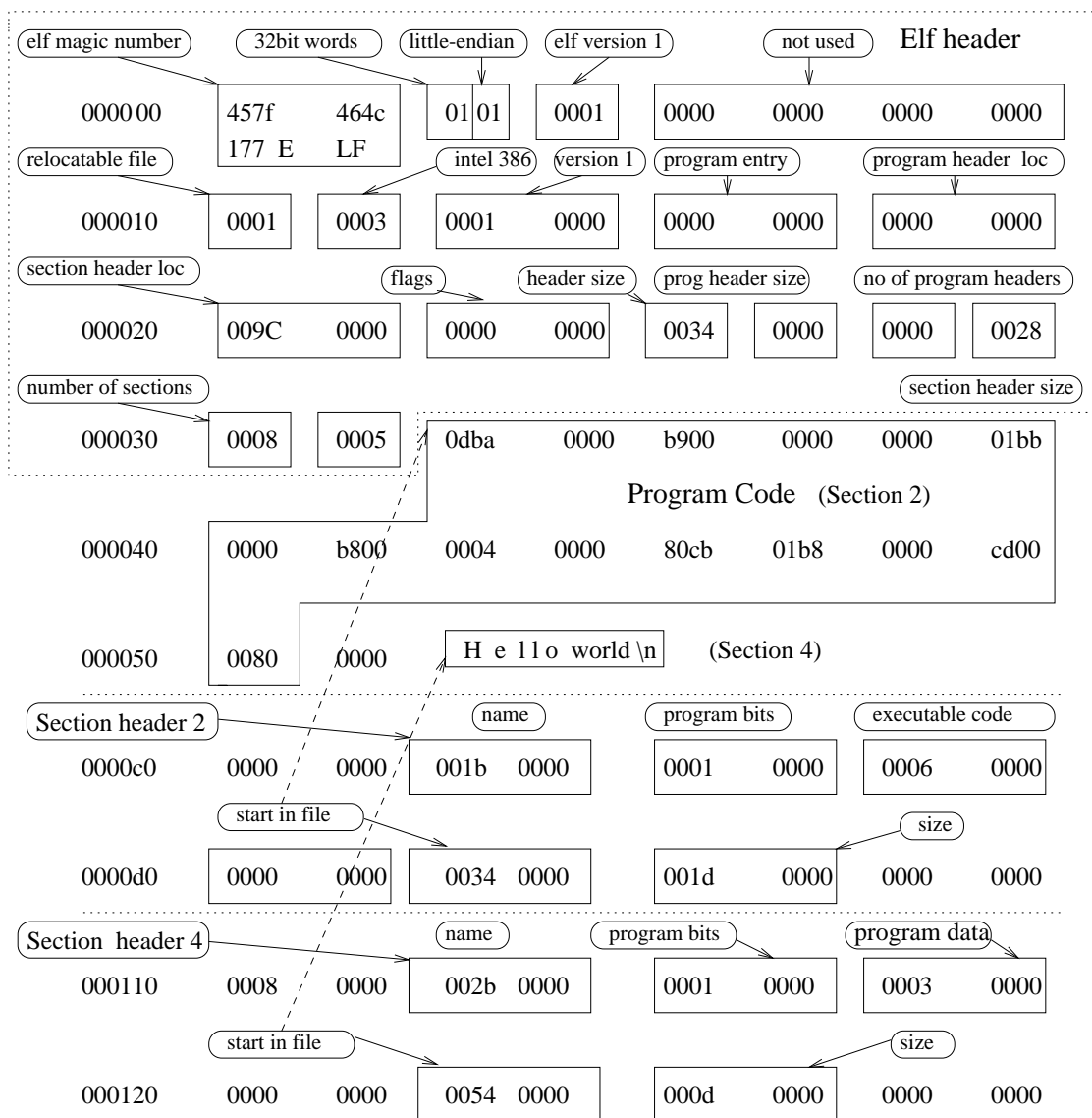
Examine the contents of a non-text file like `assembler.o` using the Unix octal-dumping utility `od`. Type:

```
od -xc -Ax assembler.o
```

The flags `-xc` and `-Ax` tell `od` to show the file in hexadecimal and ASCII, giving addresses in hexadecimal, starting from zero. The `od` output is a sequence of pairs of lines. Each pair shows the starting address, the next 16 bytes as 8 hexadecimal words, and the same 16 bytes as ASCII characters if they represent printable characters.

The result should start something like this (only the relevant output is shown to make the picture smaller).

The magic number identifies the file as ELF



the next 4 bytes say how to read it: that the file contains 32 bit words, and the order of bytes in a word, on Intel machines the lowest significant byte is in the lowest address i.e. **little-endian** (look at the order of the bytes in the first and second rows). Machines with other architectures can identify and read the file even if they can't run it.

the file is relocatable code for an Intel386 (or upwards compatible) cpu

the entry point of the program (the address of first instruction to run) is 0. The value is filled in later.

A relocatable file cannot be run yet, so there are no 'program headers'. The rest of information is held in sections headers: there are 8 sections, the header for the first section starts at 0x9C and the headers are 0x28 bytes long. The definition is given in Adding 0x64 to the name word points in the file to a string, which names the section.

The first section header is not used and not printed,

the second header starts at 0c4

and shows where the program code is in the file.

relocation The next header (not shown) says that relocation information starts at address 0x24c (the last thing in the file): details how the program should be modified to load it to a different address.

data Then comes the header that points at the data for the program.

symbol table The other important information is the the symbol table, found near the end of the file. The symbol table is a list of labels (procedures names, variable names) in the program

4 Examining the symbol table

The Unix symbol names utility `nm` lists all the symbols defined or used in an `elf` format file. Try running:

```
nm assembler.o
```

The only labels defined in this program are `start` and `message`. The Unix manual page `man nm` explains what the output means.

5 Examining the instructions

The utility `objdump` is handy for looking at executable files, it displays information about the format and the sections of the file, try:

```
objdump --all-headers assembler.o
```

Use the option `--disassemble` to make `objdump` display the executable part of the file as assembler code.

6 Using the linker

The purpose of the linker program `ld` is to manage labels in a program which may consist of modules which have been compiled separately. In the simple example given above all that needs to be done is to relocate the text and data so that it resides at the right start address. To do this type

```
ld -N assembler.o -o assembler
```

This command takes the code from `assembler.o` and relocates it for loading at address `0x08048074` where the Unix loader expects it to be, and saves the resulting executable program in the file `assembler`. This is in the same format as `assembler.o`, but examining it using `od` shows that it now has the entry point `0x08048074` and a few other bits and pieces have also changed. You may wish to examine it again, either using `od` or `objdump`.

7 Running the program

Just type `assembler` to run the program. At this point the data structure contained in the file `assembler` is interpreted by the operating system loader. It checks the machine type and examines the header to determine what kind of environment the process will need. Having set up a new process of the appropriate kind, the instructions and initialized data are loaded into free memory, and the operating system branches to the entry point. This behavior is described in detail by the manual page for the `execve` system call.

8 Looking at `cversion.c`

The C version of the same simple program is given in `cversion.c` : This program imports two externally-defined functions `write()` and `exit()` which are defined in `writeln.s`: The rest of the exercise concentrates on constructing a running program from these two parts, examining the intermediate files at every stage. The steps involved in producing a runnable program are :

1. Use the C compiler `gcc` to translate the C program into an assembler program. Type:

```
gcc -S cversion.c
```

The `-S` flag means stop after producing the assembler version. Examine the results in an assembler file called `cversion.s` .

2. Assemble this file using `as`, type:

```
as cversion.s -o cversion.o
```

This results in the file `cversion.o`, which is in `elf` format. What happens if you try to link it using `ld`, as you did with `assembler.o`? Try `nm` on it. What does its output mean?

3. Assemble the file `writeln.s`:

```
as writeln.s -o writeln.o
```

Try `nm` on this. What does its output mean this time?

4. Link the two using `ld`. Type:

```
ld -N cversion.o writeln.o -o cversion
```

This results in a file called `cversion`, which is also in the `elf` format. Run it and then examine its contents using `od`, `nm` and `objdump`.

9 Experimenting with optimization

Now you know how to run the C compiler and examine its output, you can experiment to see how clever the C compiler can be. For example, does it replace constant expressions by their values? Does it avoid recomputing an expression's value inside a loop if that value cannot change during the lifetime of the loop?

To perform experiments like this, use the slightly more complex C program provided to start with, `example.c` Use the C compiler to compile `example.c` into an assembly language version:

```
gcc -S example.c
```

Normally the C compiler tries to compile quickly rather than to produce very good code. But if you use the `-O` flag, it will try to optimise the code.

```
gcc -O -S example.c
```

or

```
gcc -O2 -S example.c
```

or even

```
gcc -O2 -finline-functions -S example.c
```

Examine the assembly code produced by each level of optimisation – you may find that the code gets rather complicated at high optimization levels.

10 Notes

The exercise has been carefully constructed to avoid using some of the more advanced features which would normally come into play when compiling under Linux. In particular:

- Normally, `write()` and `exit()` would be system calls, found in the *C standard library* `/usr/lib/libc.a`. The standard library and the *C run-time system* `/usr/lib/crt1.o` are normally linked with C programs automatically – linking is normally done using `gcc` not `ld`. This was prevented here by performing the link phase explicitly, forcing the linker to omit all such libraries.
- The `-N` flag of the `ld` program prevents the use of Linux's shared libraries mechanism. The shared libraries scheme does not link in all the library routines with each executable thus making the stored executables smaller. Instead, a single copy of the library routines is loaded into memory *at run-time* and shared between all programs running on the same machine at the same time. This speeds program loading, reduces disk swapping, and therefore speeds program execution.
- Adding the option `-v` to `gcc` will show what it is doing and which files are used. To convert a C program into a binary file `gcc` runs four programs, `cpp` the C pre-processor, `cc1` the compiler, `as` and `ld`.

A Linux PC Assembler Syntax

The assembler used on Linux has a syntax similar to other Unix assemblers and differs from the MS-DOS assembler:

- Unix tools like the assembler and C compiler, when they read or write numbers, prefix `0x` to indicate the number is written in hexadecimal and `0` that it is octal.
- Register names start with `%"`, `'e'` at the start of a register name e.g. `%eax` shows the full 32 bits are used.
- The operands written are in the order: source, destination, the opposite order to MS-DOS assembler.
- Constants in the assembler source are written with a leading `$`.

*Olav Beckmann, Imperial College London, November 14, 2005
With lots of acknowledgements to Paul Kelly*