Homework 7
Load Module Analyzer

Due Date

Due: Thursday, April 27, 2017 at 11:59pm

Revision History

- Fixed another typo: Thu Apr 20 18:13:54 EDT 2017 bks@cs.rit.edu
- Fixed revisions typo: Thu Apr 20 11:51:40 EDT 2017 bks@cs.rit.edu
- Initial version: Tue Apr 18 11:59:14 EDT 2017 bks@cs.rit.edu

Overview

In our discussion of program translation, we discussed the typical contents found in an object or load module file. One of these is the *Executable and Linking Format* (ELF)[1], which is used in many different operating systems (including Linux®). It is an extremely flexible format, and even allows the creation of object and load module files which contain executable code for multiple architectures.

For this assignment, you will work with a much simpler object module format. This will require that you read and process non-text, binary data, and that you understand how information can be "packed" into words so as to save space.

The object module format you will work with was designed for use with the assembler, linker, and simulator used in the CSCI-250 (Concepts of Computer Systems) course, which uses the MIPS R2000 processor as its example of CPU design and assembly language programming. It is significantly simpler than the ELF format, and was designed to be easy to work with while still providing enough information to allow linking of separately-translated source programs. Because that course uses the MIPS R2000 architecture, this suite of programs is called the R2K suite, and thus the object module format is the R2K module format.

The R2000 is a 32-bit CPU with a 4 byte addresses and instructions. You will find a description of the R2K object module format in a separate document: The R2K Object Module Format.

Supplied Code

Create a subdirectory to contain your work for this assignment, and `cd` into that directory. Retrieve the materials for this homework using the command:

```
get  csci243  hw7
```

This will copy a number of files from the course account into your working directory, as follows:

`exec.h`	Declaration of relevant structures and some CPP macros to simplify some tasks
`header.mak`	For use with `gmakemake`
`sample.asm`	A simple R2K assembly language program
`sample.obj`	The object file version of `sample.asm`
`sample.out`	The load module (i.e., linked executable) version of `sample.asm`
`stdout.1`	Output from running the command "`alm sample.out`"
`stdout.2`	Output from running the command "`alm sample.obj`"

You should not submit any of these files. A copy of `exec.h` will be provided by `try` when you submit your solution. The sample object and load module files are to be used when testing your program.

Activities

Based on the R2K format described in the document linked above, you are to write a program which analyzes R2K object modules and load modules. Your program will be invoked with one or more R2K modules on the command line:

```
alm  file-1 [ file-2 ... ]
```

For each *`file-i`* on the command line, your program will produce a summary of the information found in that object or load module, as described below.

Unless otherwise indicated, all output from your program should be printed to stdout. The only output printed to stderr are error messages associated with opening or processing an R2K module.

Command Line Argument Verification

If your program is invoked with no command-line arguments, print this usage message to stderr and exit:

```
usage: alm file1 [ file2 ... ]
```

File Processing

For each command-line argument, you must open the file for input, read its contents, and produce your report. If the open fails, use `perror()` to print an error message with the name of the file which could not be opened as the prefix, and move to the next command-line argument.

Assuming the file can be opened, you must first verify the magic number. If the magic number is not `0xface`, print the message

```
error: file is not an R2K object module (magic number 0xXXXX)
```

to the stderr, where *`file`* is the name of the file being checked, and *`XXXX`* is the magic number found in the file (printed as a four-digit hex value). After this, move to the next command-line argument.

Once you have verified that the file is an R2K module, produce your report. Print a separator line consisting of a twenty hyphen ('-') characters, followed by one line that indicates whether this is an object module or a load module. For object modules, print:

```
--------------------
File name is an R2K object module
```

where *`name`* is the name of the R2K module file. For load modules, print:

```
--------------------
File name is an R2K load module (entry point 0xaaaaaaaa)
```

where *`aaaaaaaa`* is the 32-bit entry point, printed as an eight-character hex value. After this, decode the version number and report it:

```
Module version:  2yyy/mm/dd
```

where the values you print are the decoded fields from the version number field in the module header, with the month and day printed as two-digit values. (You will not be doing any processing of the `flags` field in the header.)

Next, you must print a summary of the sections which are present in the module and their sizes. If a section has a non-zero size, print the following line for it:

```
Section name is n unit long
```

where *`name`* is the section name ("text", "rdata", "sbss", etc.), *`n`* is the number of things in it, and *`unit`* is either `bytes` or `entries`, depending on whether this section is just binary data (text and data sections, string table) or an array of elements (relocation, reference, and symbol tables).

You will not do any processing of text and data section contents in this program, although you will need to somehow skip over those sections in order to get to the following ones. See below for some suggestions on how to do this.

If the relocation table is not empty, print an introductory line followed by one line (indented by three spaces) for each entry in the table; e.g., for *N* entries:

```
Relocation information:
   0xaaaaaaa1 (sect1) type 0xttt1
   0xaaaaaaa2 (sect2) type 0xttt2
    . . .
   0xaaaaaaaN (sectN) type 0xtttN
```

where each *aaaaaaaa* is the 32-bit address field printed as an eight-digit hex value, each *sect* is the name of the section to which this entry belongs, and each *tttt* is the relocation type printed as a four-digit hex value. (Note that you are *not* interpreting the type codes, just printing them out.)

Similarly, if the reference table is not empty, print the reference table using the following format:

```
Reference information:
   0xaaaaaaa1 type 0xttt1 symbol name1
   0xaaaaaaa2 type 0xttt2 symbol name2
    . . .
   0xaaaaaaaN type 0xtttN symbol nameN
```

where each *aaaaaaaa* and *tttt* are as in the relocation output, and each *name* is the symbol name (printed from the string table). (Again, note that you are not interpreting the type codes, just printing them out.)

Next, do the same thing for the symbol table if it is not empty; print a symbol table report using the following format:

```
Symbol table:
   value 0xvvvvvvv1 flags 0xfffffff1 symbol name1
   value 0xvvvvvvv2 flags 0xfffffff2 symbol name2
    . . .
   value 0xvvvvvvvN flags 0xfffffffN symbol nameN
--------------------
```

where each *vvvvvvvv* is the 32-bit value associated with the symbol (as an eight-digit hex value), each *ffffffff* is the 32-bit flag word for the symbol (also printed as an eight-digit hex value), and each *name* is the symbol name (printed from the string table). (Once more note that you are not interpreting the type codes or the flags, just printing them out.)

Finally, print a separator line and then move to the next command-line argument:

```
--------------------
```

An Example

Consider the contents of the R2K load module supplied to you with the name `sample.out`, printed as a series of 32-bit hex values:

```
220fcefa 00000000 64004000 84000000 20000000 08000000 00000000 00000000
00000000 00000000 00000000 08000000 38000000 fcffbd23 0000bfaf 00000234
21408000 2148a000 01002a29 04004015 00008a8c 04008420 ffff2921 05001008
0000bf8f 0400bd23 0800e003 fcffbd23 0000bfaf 0010043c 00008434 0010013c
2000258c 0000100c 0000bf8f 0400bd23 0800e003 0dad0b00 0000a48f 0400a58f
0800a68f 0e00100c 00000000 20204000 11000234 0c000000 0a000000 14000000
0c000000 eb000000 4f2f0000 b6680100 efffffff 00000000 07000000 00000000
b1000000 2c004000 00000000 a3000000 20000010 05000000 b1400000 38004000
0b000000 a1000000 14004000 10000000 b1400000 00004000 15000000 67000000
11000000 1f000000 a2000000 00000010 19000000 b1400000 64004000 29000000
656e6f64 756f6300 6d00746e 006e6961 706f6f6c 6d757300 72726100 53007961
455f5359 32544958 725f5f00 5f5f6b32 72746e65 005f5f79
```

Note the byte ordering in the first 32-bit value, which contains both the magic number and version number fields. For comparison, here is the same module printed as individual bytes in hex:

```
fa ce 0f 22 00 00 00 00 00 40 00 64 00 00 00 84
00 00 00 20 00 00 00 08 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08
00 00 00 38 23 bd ff fc af bf 00 00 34 02 00 00
00 80 40 21 00 a0 48 21 29 2a 00 01 15 40 00 04
8c 8a 00 00 20 84 00 04 21 29 ff ff 08 10 00 05
8f bf 00 00 23 bd 00 04 03 e0 00 08 23 bd ff fc
af bf 00 00 3c 04 10 00 34 84 00 00 3c 01 10 00
8c 25 00 20 0c 10 00 00 8f bf 00 00 23 bd 00 04
03 e0 00 08 00 0b ad 0d 8f a4 00 00 8f a5 00 04
8f a6 00 08 0c 10 00 0e 00 00 00 00 00 40 20 20
34 02 00 11 00 00 00 0c 00 00 00 0a 00 00 00 14
00 00 00 0c 00 00 00 eb 00 00 2f 4f 00 01 68 b6
ff ff ff ef 00 00 00 00 00 00 00 07 00 00 00 00
00 00 00 b1 00 40 00 2c 00 00 00 00 00 00 00 a3
10 00 00 20 00 00 00 05 00 00 40 b1 00 40 00 38
00 00 00 0b 00 00 00 a1 00 40 00 14 00 00 00 10
00 00 40 b1 00 40 00 00 00 00 00 15 00 00 00 67
00 00 00 11 00 00 00 1f 00 00 00 a2 10 00 00 00
00 00 00 19 00 00 40 b1 00 40 00 64 00 00 00 29
64 6f 6e 65 00 63 6f 75 6e 74 00 6d 61 69 6e 00
6c 6f 6f 70 00 73 75 6d 00 61 72 72 61 79 00 53
59 53 5f 45 58 49 54 32 00 5f 5f 72 32 6b 5f 5f
65 6e 74 72 79 5f 5f 00
```

In this version, the individual bytes appear in the "correct" sequence. The first row of hex values are the first 16 bytes of the load module; here they are regrouped into sizes that match the first fields of the `exec_t` structure:

```
face 0f22 00000000 00400064 00000084
```

The first two bytes contain the 16-bit magic number, which is correct (0xface). The next two bytes contain the version number; the value is 0x0f22, which decodes to 2007/09/02. Following that are the flag word (0x00000000), the entry point (0x00400064 - because this is non-zero, this is a load module rather than an object module), and the first element of the data array (0x00000084, which is the size of the text segment, in bytes). Your program should produce this output to this point:

```
--------------------
File sample.out is an R2K load module (entry point 0x00400064)
Module version:  2007/09/02
```

Collectively, here are the 13 32-bit words comprising the header block:

```
face0f22 00000000 00400064 00000084
00000020 00000008 00000000 00000000
00000000 00000000 00000000 00000008
00000038
```

From this, we see that sections 0 (text), 1 (rdata), 2 (data), 8 (symbol table), and 9 (string table) have non-zero sizes, and therefore only those sections will be reported as having contents:

```
Section text is 132 bytes long
Section rdata is 32 bytes long
Section data is 8 bytes long
Section symtab is 8 entries long
Section strings is 56 bytes long
```

The next 132 bytes comprise the text section:

```
         23bdfffc afbf0000 34020000
00804021 00a04821 292a0001 15400004
8c8a0000 20840004 2129ffff 08100005
8fbf0000 23bd0004 03e00008 23bdfffc
afbf0000 3c041000 34840000 3c011000
8c250020 0c100000 8fbf0000 23bd0004
03e00008 000bad0d 8fa40000 8fa50004
8fa60008 0c10000e 00000000 00402020
34020011 0000000c
```

Following this are the 32 bytes of the read-only data section:

```
                  0000000a 00000014
0000000c 000000eb 00002f4f 000168b6
ffffffef 00000000
```

Next are the 8 bytes of the data section:

```
                  00000007 00000000
```

Remember that your program is not looking at the contents of any text or data sections in the module file; however, you must move past them in order to get to the following sections. See below for some suggestions as to how to do this.

The relocation and reference sections are empty in this module, which means that immediately after the data section will be the symbol table. Each entry is 12 bytes long, and there are 8 entries:

```
000000b1 0040002c 00000000 000000a3
10000020 00000005 000040b1 00400038
0000000b 000000a1 00400014 00000010
000040b1 00400000 00000015 00000067
00000011 0000001f 000000a2 10000000
00000019 000040b1 00400064 00000029
```

Each group of three 32-bit words is a symbol table entry (shown here with the decimal equivalent of the string table index in parentheses):

Flags	Address	Index
000000b1	0040002c	00000000 (0)
000000a3	10000020	00000005 (5)
000040b1	00400038	0000000b (11)
000000a1	00400014	00000010 (16)

000040b1	00400000	00000015 (21)
00000067	00000011	0000001f (31)
000000a2	10000000	00000019 (25)
000040b1	00400064	00000029 (41)

The string table is the remaining 56 bytes of the module:

```
64 6f 6e 65 00 63 6f 75 6e 74 00 6d 61 69 6e 00
6c 6f 6f 70 00 73 75 6d 00 61 72 72 61 79 00 53
59 53 5f 45 58 49 54 32 00 5f 5f 72 32 6b 5f 5f
65 6e 74 72 79 5f 5f 00
```

Broken into NUL-terminated strings, here are the offsets into the table, the bytes of the string, and their ASCII equivalents:

Index	Bytes	ASCII
0	64 6f 6e 65 00	done
5	63 6f 75 6e 74 00	count
11	6d 61 69 6e 00	main
16	6c 6f 6f 70 00	loop
21	73 75 6d 00	sum
25	61 72 72 61 79 00	array
31	53 59 53 5f 45 58 49 54 32 00	SYS_EXIT2
41	5f 5f 72 32 6b 5f 5f 65 6e 74 72 79 5f 5f 00	__r2k__entry__

From this, we will produce the symbol table report. Putting that together with the output produced so far, here is the complete output for this module:

```
--------------------
File sample.out is an R2K load module (entry point 0x00400064)
Module version:  2007/09/02
Section text is 132 bytes long
Section rdata is 32 bytes long
Section data is 8 bytes long
Section symtab is 8 entries long
Section strings is 56 bytes long
Symbol table:
   value 0x0040002c flags 0x000000b1 symbol done
   value 0x10000020 flags 0x000000a3 symbol count
   value 0x00400038 flags 0x000040b1 symbol main
   value 0x00400014 flags 0x000000a1 symbol loop
   value 0x00400000 flags 0x000040b1 symbol sum
   value 0x00000011 flags 0x00000067 symbol SYS_EXIT2
   value 0x10000000 flags 0x000000a2 symbol array
```

```
      value 0x00400064 flags 0x000040b1 symbol __r2k__entry__
   --------------------
```

Note above that there are 2 lines with a string of hyphens ('-'); one is at the start, and the other at the end. When the program processes multiple files on the command line, there will be 2 adjacent lines of hyphens between each file's information.

Submitting Your Solution

You should submit the following file(s) as your solution:

- `alm.c` which contains minimally your main function;
- other `.c` and `.h` files you create;
- the `revisions.txt` file, and
- *optionally*, a `readme.txt` file containing any additional information you wish to provide to your instructor and/or the grader.

Do not submit any of the files given to you when you ran the `get` command described earlier.

Submit your solution with the following command:

```
try  grd-243  hw7-1  alm.c revisions.txt [other files]
```

As with earlier homework assignments, you can check the status of your submission with the following `try` command:

```
try  -q  grd-243  hw7-1
```

You will see a list of files that were submitted, and will then be asked "Would you like to see the contents of these files?"; if you answer "y", the contents of the file you submitted will be shown in your shell window.

Grading

Your submission will be grade out of 100 points, with points distributed as follows:

- Performance (50%):
  The program produces the correct output for each object or load module given to it on the command line. It also correctly handles error cases such as no command-line arguments, and object or load module files that cannot be opened (in which case an *appropriate* error message is printed).
- Design (30%):
  The program source code is reasonably separate into functions which each perform specific tasks. Common tasks should be abstracted into supporting functions rather than being replicated in several

different functions. Source code is organized into separate source files as appropriate, with functions performing related tasks grouped together; e.g., functions which manipulate a particular data structure (such as a queue or a heap) should all be in one source file. When multiple source files are used, header files containing shared variables, data types, and/or function prototypes also exist, but they do not contain private types, variables, and/or function prototypes.

- Documentation (10%):
  Every submitted `.c` and `.h` file has a file comment block at the top which contains the author's full name, the purpose of the file, and information about version control. All header files must use *include guards* to protect against multiple inclusion problems.

  The revision history must be submitted in a separate, plain text file called `revisions.txt`.

- Style (10%):
  The layout and style of the source code is consistent in indentation of lines, spacing of tokens, and organization of components (type specifications, declarations, variables, functions, constants, etc.). Lines are of a reasonable length, with the majority shorter than 80 characters and no lines longer than 100 characters.

Hints, Suggestions, etc.

You are free to use either the stdio input function `fread()` or the `read()` system call to read the contents of the object module files. If you choose to use the stdio package, you will need to use `fopen()` and `fclose()` to open and close the module files, and will be working with a `FILE *` handle to read from the files. If you choose to use the system call, you'll need to use `open()` and `close()` instead, and will be using an integer file descriptor to read from the files.

The header block is a fixed-sized block in the module file, but all the other sections vary in length depending on the code that was assembled. However, you *do* know how much space each will require, so dynamically allocating storage to hold them is easy.

You will need to skip over the bytes found in the text and data sections in the module file. One way to do this would be to actually read them in, which will advance the file i/o pointer (within the operating system's open file table) automatically. Alternatively, you can use a *seek* function to just advance the i/o pointer directly: this can be done with the `fseek()` function (if you are using `fread()` to read the module) or the `lseek()` system call (if you are using `read()` to read the module). See their manual pages ("`man fseek`", "`man lseek`") for usage details.

Remember to close each object module file after you are done processing it, and to `free()` any dynamically-allocated memory you used to hold the contents of the file's sections!

Don't forget that the MIPS R2000 is a big-endian machine; this means that the bytes comprising 16-bit and 32-bit numeric values are "swapped". To deal with them, you can either write your own byte-swap routines, or you can use C library functions to do the swapping; see the manual page for the endian functions ("`man endian`") for details, paying particular attention to the required header file to include and the functions `be16toh()` and `be32toh()` which convert from "big-endian" order to "host" (i.e., the computer you're running the program on) order. Alternatively, you can use the network byte ordering functions ("`man ntohs`"), especially `ntohs()` and `ntohl()`, to do the same conversion.

---

[1]See `https://refspecs.linuxbase.org/elf/elf.pdf` for details.