Lab E: Linking ELF Object Files

This lab may be done either solo or in pairs.

In the previous lab, you learned to investigate and change ELF files using hexedit, and other command-line tools. In this lab, you will continue to manipulate ELF files, this time using your own code (written in C), and perform a limited pass I operation of the linkage editor ("linker").

We will parse the ELF file and extract useful information from it. In particular, we will access the data in the section header table, and in the symbol table. We will also learn to use the mmap system call.

Important

This lab is written for 32-bit machines. Some of the computers in the labs already run on a 64-bit OS (use *uname -a* to see if the linux OS is 64-bit or not). 32-bit and 64-bit machines have different instruction sets and different memory layout. Make sure to include the *-m32* flag when you compile files, and to use the Elf32 data structures (and not the Elf64 ones).

In order to know if an executable file is compiled for 64-bit or 32-bit platform, you can use readelf, or the *file* command-line tool (for example: `file /bin/ls`).

Useful Tips

You will no longer be using *hexedit* to process the file and strings to find the information; nevertheless, in some cases you may still want to use these tools for debugging purposes. In order to take advantage of these tools and make your tasks easier, you should:

- Print debugging messages: in particular the offsets of the various items, as you discover them from the headers.
- Use *hexedit* and *readelf* to compare the information you are looking for, especially if you run into unknown problems. *hexedit* is great if you know the exact location of the item you are looking for.
- Note that while the object files you will be processing will be linked using *ld*, and will, in most cases, use direct system calls in order to make the ELF file simpler, there is no reason why the programs you write need use this interface. You are allowed to use the standard library when building your own C programs.
- In order to preserve your sanity, even if the code you MANIPULATE may be without stdlib, we advise that for your OWN CODE you DO use the C standard library!
- In order to keep sane in the following lab as well, **understand** what you are doing and **keep track** of that and of your code, as you will be using them in a future lab.

Lab E Assignment

The goal of this lab is to implement a limited pass I (merging) of a linkage editor. You begin by allowing access to ELF object files, examining, and printing out their structures (section headers and symbol table). After this part is correctly implemented and debugged, you will be implementing the part that does the merging. This assignment will be limited to merging 2 ELF files, and with additional simplifying assumptions and restrictions specified later on.

You must use only the mmap system call to read data from your ELF files from this point onwards. However, you should use write( ) to generate the merged output file.

Part 0

This part is about learning to use the mmap system call. Read about the mmap system call (`man mmap`).

Write a program that uses the mmap to examine the header of a 32-bit ELF file (include and use the structures in elf.h). The program is first activated as:

myELF

The program then uses a menu similar to lab 4, with available operations, as follows:

Choose action:

0-Toggle Debug Mode

1-Examine ELF File

2-Print Section Names

3-Print Symbols

4-Check Files for Merge

5-Merge ELF Files

6-Quit

Note that the menu should use the same technique as in lab 1, i.e. an array of structures of available options. Toggle Debug Mode is as in Lab 4. Quit should unmap and close any mapped or open files, and "exit normally". Examine ELF Files queries the user for an ELF file name(s) to be used and examined henceforth. For now, options 2, 3, 4, 5, should call stub functions that print "not implemented yet". All file input should be read using the mmap system call. You are NOT ALLOWED to use read, or fread.

To make your life easier throughout the lab, for each ELF file, map the entire file with one mmap call.

The *examine* ELF file option should print the following information from the header.

1. Bytes 1,2,3 of the magic number (in ASCII). Henceforth, you should check that the number is consistent with an ELF file, and refuse to continue if it is not.
2. The data encoding scheme of the object file.
3. Entry point (hexadecimal address).
4. The file offset in which the section header table resides.
5. The number of section header entries.
6. The size of each section header entry.
7. The file offset in which the program header table resides.
8. The number of program header entries.
9. The size of each program header entry.

The above information should be printed in the above exact order (print it out as nicely as *readelf* does, and verify using readelf that your output is correct). If invoked on an ELF file, examine should initialize a global file descriptor variable for this file, and leave the file open. When invoked on a non-ELF file, or the file cannot be opened or mapped at all, you should print an error message, unmap the file (if already mapped) close the file (if already open), and set the respective file descriptor variable to -1 to indicate no valid file. You probably also should use a global variable to indicate the memory location of the mapped file.

Your Examine ELF File should be able to handle up to two ELF files, keep both open and mapped at the same time. So you should keep 2 separate file descriptors, map_start variables, and other relavant informatopn for each file. That is, each time this function is called it should get a new file name, open the file and map it, and print out the above stated information, while keeping any previous file information as well. Calling the function for the 3rd time or more you may print out an error message and do nothing, or (bonus) you may decide to support more than 2 ELF files.

You can test your code on the following file: a.out, and also any of the files in the "ELF object file examples" folder.

Part 1 - Sections

Extend your myELF program from Part 0 to allow printing of all the Section names in an 32-bit ELF file (like `readelf -S`). That is, implement the "Print Section Names" option.

For each ELF file already opened by Examine ELF File, Print Section Names should visit all section headers in the section header table, and for each one print its index, name, address, offset, size in bytes, and type number. Note that this is done for all files currently mapped, so if there is no file just print an error message and return.

The format for each ELF file should be:

File ELF-file-name

[index] section_name section_address section_offset section_size section_type

[index] section_name section_address section_offset section_size section_type

[index] section_name section_address section_offset section_size section_type

....

Verify your output is correct by comparing it to the output of *readelf*. For full credit the "section type" should be symbolic as per readelf output, and should be implemented using a lookup into an array of (number, type-name) pairs, or an array of type names, and **not** a long "if-else" statements. In debug mode you should also print the value of the important indices and offsets, such as shstrndx and the section name offsets.

You can test your code on the following file: a.out, and also any of the files in the "ELF object file examples" folder.

Hints

Global information about the ELF file is in the ELF header, including location and size of important tables. The size and name of the sections appear in the section header table. Recall that the actual name **strings** are stored in an appropriate **section** (.shstrtab for section names), and not in the section header!

Part 2 - Symbols

Extend your myELF program from part 1 to support an option that displays information on all the symbol names in a 32-bit ELF file.

The new Print Symbols option, for each open ELF file, should visit all the symbols in that ELF file (if none, print an error message and return). For each symbol, print its index number, its name and the name of the section in which it is defined. (similar to `readelf -s`). Format should be:

File ELF-file0name

[index] value section_index section_name symbol_name

[index] value section_index section_name symbol_name

[index] value section_index section_name symbol_name

...

Verify your output is correct by comparing it to the output of *readelf*. In debug mode you should first print the size of each symbol table, the number of symbols therein, and any other useful information.

Hints:

Symbols are listed in the designated sections. The section in which a symbol is defined (if it is defined) is the index of the symbol, which is an index into the section header table, referring to the section header of the appropriate section, and from there the section name can be retrieved as above. Symbol name is an attribute of the symbol structure, but recall again that the actual name string is stored in a string table, a separate section(.strtab).

Part 3 - Linker pass I

In this part you shall write a limited version of a linker pass 1 (as mentioned in the last lecture). It will merge 2 ELF files (the ones opened by Examine ELF File) into one relocateable object file. In this part, you should be using as input the files from the the "ELF object file examples" folder, where "F1a.o", F1b.o", and "F1c.o" are intended for use as the first file, and "F2a.o", "F2b.o" are intended for use as the second file. Merging files should result in a single relocatable object file, mimicking the one generated by pass I of the linker. For example, invoking pass I of the linker: "ld -r -m elf_i386 F1a.o F2a.o -o F12a.ro" generates the relocatable object file "F12a.ro". Your program should generate the same type of output ELF file given these same files. The linker can be used on (only) the resulting file to create a working executable file: "ld -m elf_i386 F12a.ro -o F12a" generates the "F12a" executable file that can actually be run.

Part 3.1: Check Files for Merge

Implement Check Files for Merge: CheckMerge. The CheckMerge function first checks that 2 ELF files have been opened and mapped, (print an error message and return otherwise). Here we assume that there are exactly 2 such ELF files, and each file contains exactly one symbol table (otherwise, print "feature not supported" and return). CheckMerge then, for each ELF file, loops over all symbols (except for symbol number 0, which is a dummy symbol) in its symbol table SYMTAB1 (except for the first symbol, which is a dummy null symbol) and for each symbol sym:

- If sym is UNDEFINED, search for sym in SYMTAB2, the symbol table of the other ELF file. If not found in SYMTAB2, or found in SYMTAB2 but UNDEFINED there as well, print an error message: "Symbol sym undefined".
- If sym is defined, i.e. has a valid section number, again search for sym in SYMTAB2. In this case, if sym is found in SYMTAB2 amd is defined, print an error message: "Symbol sym multiply defined".

Continue scanning symbols even after errors were found. Note that using "F1a.o" and "F2a.o" should result in no errors, but using "F1b.o" and "F2b.o" should result in errors.

Part 3.2: Merge ELF Files

Note: in this semester, this entire part 3.2 is a bonus assignment, not required for a full grade.

In a real linkage editor, this is done only if checking for merge passes with no errors. However, your implementation will allow this even if the previous step found errors. Here you should write a Merge function, which does the following:

- Create a new ELF file called "out.ro".
- Create an ELF header for this file and write it into the file "out.ro".
- Create a new section header table for this file.
- Create merged sections.
- Write the merged sections and the new section header table into the file "out.ro".
- Close the file "out.ro". (You may or may not need to update the ELF header before you close the file, depending on your implementation.)

To implement the above, you should follow the procedure stated for pass I in the lecture. However, below are some simplifying assumptions and restrictions that should make implementation easier. So you may assume all the following restrictions:

- There are exactly 2 ELF files being merged.
- The second ELF file does not contain any sections that do not exist in the first ELF file.
- The second ELF file does not contain any symbola that do not exist in the first ELF file.
- The second ELF file does mot have any relocations.

With all the above symplifying assumptions, the following method can be used for merging.

- Use a copy of the ELF header of the first file as the ELF header for the merged file. You need to modify only the "e_shoff" field, as specified below.
- Use a copy of the section header table of the first ELF file as an initial version of the section header table for the merged file. You will need to modify the "sh_off" and "sh_size" fields in each section header, as specified below.
- Mergable sections: ".text" ".data", ".rodata" sre merged as follows. Concatenate the section contents from the first ELF file and the section contents from the second ELF file, to create the merged appropriate section. E.g. to create the merged ".text" section, copy the contents of ".text" from the first ELF file, and to that append the contents of ".text" from the second ELF file. The merged section obviously has a size that is the sum of the sizes (should accordingly change the appropriate section header).
- Section ".shstrtab" of the first ELF file can be used without changes as the ".shstrtab" of the merged file, and the same for ".symtab" and any relocation sections (because of the restricting assumptions above).

- The symbola table of the first ELF file can be used as that of the merged file, after copying over symbol values and definition of symbols from the second ELF file, for every symbol that is UNDEFINED in the first ELF file.

A practical implementation of the above then looks as follows:

- Create "out.ro" and copy an initial version of the ELF header as its header.
- Create (in memory) an initial version of the section header table for the merged file by copying that of the first ELF file.
- Loop over the entries of the new section header table, and process each section according to the above scheme (concatenate ".text", copy ".shstrtab" as-is, etc.), and immediately write (append) the appropriate merged section to "out.ro". Note that to concatenate e.g. ".text", simply write the contents of ".text" from the first ELF file, at the end of "out.ro", then find the contents of ".text" in the second ELF file (by finding ".text" in its section header table) and write it again at the end of "out.ro" (no need to merge them in memory!) So now you know the file offset of the merged section and its length, so update the appropriate section header table entry fields (offset and size).
- Write the merged (and modified) section header table entry, appended to the end of "out.ro".
- Fix the "e_shoff" field in ELF header of "out.ro" to point to the location where you actually wrote the section header table, and close "out.ro".

The basic requirement here is that "readelf" on "out.ro" will show that all sections have been merged as stated above, and that the symbols have been resolved, when operating on pairs of ELF files that obey the restrictions stated above. For a (double) bonus, running ld (the linker pass II) on "out.ro" should work correctly in these cases, and generate an executable file that runs correctly!

Deliverables:

You need to submit "myELF.c" and a makefile which compiles and links your code. Your code should incoporate all features from parts 0, 1, 2, 3 (except possibly for part 3.2 which is a bonus part) as a single program supporting all the features.