Project Topic Two

The System Binary Tools

CSC3002

January 20, 2020

Contents

Chapter 1

From Lines of Code to A Running Program

As you have written several programs, you may come to the following questions: Why can a piece of human readable code be compiled into binary files and why can binary files be loaded and finally generate outputs?

Let us take a look at how C++ files are compiled and finally loaded to run. The files are first passed to a **compiler** (g++/icpc/clang++/cl.exe, etc.) and then the compiler will generate object files (.o/.obj). You can regard these files as containers full of machine code. If the program is single-filed (more precisely, free-standing, although actually, your program cannot be free-standing in most cases as the system runtime should be linked dynamically), it is now ready to be transformed into an executable. However, in most cases, your code may split into several different files. For example, `a.cpp` may contains a function that calls another function defined in `b.cpp`. Then, the object file generated from `a.cpp` cannot run on its own. Therefore, a linker will be needed. When the symbol of a function (or a label in the assembly language) cannot be resolved within one file, the compiler will make some marks within the generated object files (usually the file follows the format of ELF). Then the **linker** will recognize these marks, combine the object files and relocate the addresses of the symbols to their real definitions. After that, we will get the final executable files.

There are actually two types of linking in the real world: static linking and dynamic linking. The static one works just as what we have already described before. The dynamic one, however, is quite different. It does not really combine the object file together. Instead, the dependency library is stored as a dynamic library (`.so/.dll/.dylib`) and the special marks are made within in dependent executable by the linker. When the executable starts running, these marks will lead the **loader** to load the dependency library together and relocate the symbols. This kind of linking is useful if the dynamic library has a lot of dependents. (For instance, consider every GUI executable has a statically linked libc, then there will be a file size boom in your disk.)

In practice, there are actually more works to be done. For example, the starting addresses of the machine code in an executable is fixed in generating. However, as multiple programs are running in the same time, there will be some conflicts in the memory. Hence, the loader will actually relocate the whole binary code and protect the programs from unexpected interfering from each other.

Chapter 2

Project Goals

In this project, you can implement some of the following system tools:

1. Editor
2. Assembler (MIPS)
3. Simulator (MIPS)
4. Compiler (of a simple language that compiles to your assembly language, such that you can use your assembler to generate the machine code)
5. Linker
6. Loader
7. Debugger (debugger on only the assembly language is okay)

It is suggested to integrate these parts into one GUI program if the time permits.

You can design your own file format or just follow the ELF tradition. You have the freedom to combine several modules on your own. For example, if you do not feel like writing a loader but would like to choose the linker, then you can just choose not to support dynamic linking. If you do not want to have multiple programs running together to reduce the complexity of your loader, it is also acceptable. You can even limit the simulated program to a single file so that there is no need for a static linker. The features are decided by yourselves.

The design, the amount of effective code and the language skills are the most important criteria in grading. If your group happens to have members taking related courses at the same time, you can import the work from the other courses, but directly re-submitting of the same outcome is **not** acceptable. For instance, you should try to add new features to your existed assembler (error handling/ support label/parallel code generation, etc.) and demonstrate them in your final report.

You need to decide the following things in your proposal:

1. What modules would you like to choose?
2. How will you display your final outcome?
3. Time table and work distribution in your group.

You are expected to hand in the following things when the project is finished:

1. Source code (archived).
2. A manual on how to prepare the environment and compile your source code (please make sure it is reproducible).
3. A demo of the assembly language program or your simple language program.
4. A report of your implementation with a manual of how to test the features.

Appendix A

References and Tips

A.1 Editor

A good editor actually requires quite a bit work to be done. Several data structures can be used to improve the performance of the editor including piece-table and gap vector(a vector that keeps the gap after the cursor position by moving other data). There is a paper discusses these detail.

The way to store strings is another issue. Using correct way to generate strings can largely improve the space efficiency or locality.

Of course, if you choose the editor, you are not required to take all of these into consideration. You should first implement a very-trivial editor by using existed GUI components and then do the optimization if you have extra time.

A.2 Assembler & Simulator

Apart from the minimal MIPS instructions, you can also add several special instructions or define a list of system calls to help you display text or do something else to illustrate your work.

A.3 Compiler

As for compiler, there is actually a complete theory for this field. There are a few steps of compiling: lexical analysis, syntax analysis, semantic analysis and code generation. Compilers need to handle some hard problems like scoped symbol resolution, error recover, register assignment and so on.

However, a simple language consisting of arithmetic expressions (or even simple function declaration and definition) is relatively easy to compile. To implement the compilation of such a simple language, you may even simply ignore some of the profound theories of compilers.