CS 213, Spring 2000
Homework Assignment H1
Assigned: Feb. 1 Due: Mon., Feb. 15, 11:59PM

Prof. Mowry (`tcm@cs.cmu.edu`) is the lead person for this assignment.

The purpose of this assignment is to learn the IA32/Linux assembly language. You will become familiar with how C code is translated into both assembly and machine code, and you will learn to use tools such as disassemblers and debuggers. You will do this by looking at a series of assembly language ("`.s`") and machine code ("`.o`") files and reverse engineering them to find the C source code that produced them. Reverse engineering this code will improve your understanding of the C constructs as well as the assembly and machine codes.

Introduction

In this assignment you will be given either an assembly code or a machine code file for a function. Your task is to derive C source code which compiles into "equivalent" code. None of the functions requires more than a few lines of code.

What do we mean by "equivalent"? Certainly if your C code produces *exactly* the same assembly or object code when it is compiled as the file we give you, then it is clearly equivalent. However, due to heuristics within compilers (not to mention any differences in how they are configured), seemingly insignificant (and functionally meaningless) changes to the C source code can sometimes result in different outputs from the compiler. To help you avoid the frustrations of trying to reverse engineer all of the quirky features of a compiler, we define "equivalent" to mean *functionally equivalent* for this assignment. Hence the compiled C source code must produce exactly the same output as the function we give you for all possible inputs.

We will give you a simple test program which will do limited random testing of your C source code to see if it appears to be correct. Since these tests are not exhaustive (with the exception of case **b11**, which can easily be tested precisely), there is still a chance that your C code is not functionally equivalent to the code we have given you. We will perform additional testing on the code that you hand in. Hence we encourage you to try to get as close of a match as you can, and to spend some time thinking about whether your code really is functionally equivalent or not.

When you are trying to figure out what a given functions does, try creating a small example to see what code the compiler emits. If you can create a series of small functions that produce part of the answer, you can then piece them together to create a solution.

Logistics

You must work alone on this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on Web page `assigns.html` in the class WWW directory.

All files for this assignment are in the directory:

```
/afs/cs.cmu.edu/academic/class/15213-s00/H1
```

You will want to do your work on one of the class "fish" machines to be sure that you are using the correct version of the GCC compiler. See the class WWW pages for more information on these machines.

To get a copy of the assignment, log in to a fish machine (or any Andrew UNIX machine) and execute the following commmand:

```
   /afs/cs.cmu.edu/academic/class/15213-s00/H1/H1_setup
```

This will create a protected subdirectory under ∼`/213hw` called `H1` which will contain the files `Makefile`, `TEAM_MEMBER`, `a[1-6].c`, `a[1-6]-solve.s`, `b[7-11].c`, `b11.h`, `b[7-11]-solve.o`, and various `test_*` files. In this assignment you will only modify and hand in the files `a[1-6].c`, `b[7-10].c`, `b11.h`, as well as `TEAM_MEMBER`. Please edit the `TEAM_MEMBER` file now so you do not forget.

The files `a[1-6]-solve.s` and `b[7-11]-solve.o` are the assembly code and object code files, respectively, that you are trying to duplicate. The files `a[1-6].c` and `b[7-10].c` contain the skeletons of the C functions you are to write. For problem 11, you are only to modify `b11.h`, although you will also want to look at `b11.c`.

To generate the assembly code from `a1.c`, use the command `make a1.s`. This will generate the assembly code using the same compiler flags as were used in compiling the solution code. The flag settings are very important in determining what code gets generated. The same method holds for generating assembly for your other files.

You cannot view the object code in files `b[7-11]-solve.o` directly. Instead, you need to use a disassembler. Run the command `make b7.odis` to generate a compiled and diassembled version of `b7.c`. Similarly, you can generate a disassembled version of the target code with the command `make b7-solve.odis`. The makefile command strips off the file name that would otherwise be in the disassembled files, and hence the ultimate goal is to make these two files identical. Similar processes can be used for the other problems.

Note that a "`.o`" file has not been linked yet; hence the addresses are not in their final form, and the links to library procedures haven't been resolved yet.

The problems are self-testing. To check your code for problem 1, run the command `make check1`. This will perform some limited functional testing of your C source code in `a1.c` to check whether it corresponds to the behavior of `a1-solve.s`. If so, you will see the message "`a1 appears to be OK`". (As mentioned before, since the testing is limited, a positive result does not guarantee functional equivalence.) If the testing fails, you will see the message"`a1 NOT OK`". Other problems are tested in a similar fashion. Running `make check` will check all of your files.

Some Hints

The following are some hints regarding the original C source that generated the solution files:

a1: Very simple arithmetic.

a2: Accesses a global variable and dereferences a pointer.

a3: Some arithmetic that gets optimized by the compiler, as described in Class 04.

a4: Simple conditionals.

a5: A `while` loop.

a6: A `for` loop.

b7: A procedure using a switch statement. You just need to fill in the body of the switch statement. You will find it useful to run GDB on the file `b7.o` and extract the contents of the jump table with the examine ("`x`") command.

b8: Fill in the missing code for `caller`.

b9: A simple recursive routine.

b10: A more complex recursive routine.

b11: The code in `b11.c` is already correct (do not modify it). You just need to find the correct values for the constants `ROW` and `COL` in `b11.h`.

Evaluation

Each problem is worth 5 points. If your C code passes the simple test that we give you, you are guaranteed to get at least 80% of the credit for that case (i.e. 4 out of 5 points). To receive full credit, however, your code must pass our more rigorous tests.

You are not allowed to use any `ASM` statements (i.e. direct embedded assembly code) in your C source code, since this defeats the purpose of the assignment. Similarly, using any tools which automatically convert either assembly or object code into source code is also strictly forbidden.

Hand In

Make sure you have edited the file `TEAM_MEMBER` to identify who you are (remember—you are a one-person "team"). By running the command `make handin NAME=yourname`, where `yourname` is replaced with your Andrew Id, a tarfile will be created and copied to the handin directory.

You only have write permission to this directory. You can submit updated versions with the command `make handin NAME=yourname VERSION=XX`, where `yourname` is your Andrew ID, and `XX` is the version number, i.e., 2, 3, ... Only your highest numbered version submitted before the deadline will be graded.