Lab 7.1

Out: Thursday, March 17, 2005

What you'll learn.

You'll be experimenting with compiling in new ways, looking at assembly code, examining library code, and measuring performance. This lab consists of very little coding – instead, it's just about learning where things are and determining what code is actually generated by the compiler.

How you'll do it.

You'll be writing one or two very simple .c files that contain fewer than ten lines each. You'll then be using `gprof` and `objdump` and new flags to `gcc` to generate assembly code and gauge program execution time.

Task 1: Creating two test files.

We want a simple test of performance. You'll be creating two test files. In both files, you'll be copying one large array of integers to another a whole lot of times. In one file you will do this with pointer arithmetic; in the other, you'll perform the task using `memcpy`.

Task: Create two .c files. Name one memcpy_test.c and one intcpy_test.c. In each, you should only have one function: `main`. Inside of main, in each file, create two integer arrays on the stack that hold 1024 integers apiece. Then make a `for` loop in each file that executes (10 * 1024 * 1024) times. Inside memcpy_test.c, use the function `memcpy` to copy all 1024 integers from one array to the other. Inside intcpy_test.c, create another `for` loop that loops over all 1024 integers and copies from one array to the other. Note that, if you allocate separate `int` pointers named `first` and `second`, you can easily copy from one array to another with the line: `(*first++) = (*second++);` – make sure that both files compile using `gcc`.

Task 2: Look at the generated assembly code.

The -S flag.

Open the manpage for `gcc` by typing `man gcc` in a shell. Search for the string "-S" by typing a '/' and then typing "-S." Hit 'n' to skip from search result to search result. Read about what happens when you pass the "-S" flag to `gcc`.

> **Task:** Compile both test files to assembly code. Look at the similarities and differences between the two. Where does the copy actually occur in each?

Where's `memcpy`?

In the assembly code you generated before from `memcpy_test.c`, there was a "call memcpy" line, and that was the only place the function "memcpy" was mentioned in the generate assembly code. This is because "memcpy" is part of a standard library (pre-compiled functions) that must linked into your program's executable in the linker stage. Unfortunately, the linker stage occurs after `gcc` produces the assembly file you were examining.

The solution? `objdump`.

`objdump` is a program that lets you examine an object file – i.e. an executable file. Just type `objdump` in a shell by itself to get an idea of what it can do. The flag that we'll be using is the "-S" flag. Read in the manpage for more information on what this does.

> **Task:** Compile both test files to executables. Look at the similarities and differences between the two using `objdump`. What are the differences between this and the previously generated assembly code? Does this help us identify what is occurring during `memcpy`?

The -static flag.

Read in the `gcc` manpage about the "-static" flag.

So that gives you a great explanation of "-shared." That is, if you know what "dynamic linking" and "shared libraries" are...

As you saw in the last task, the object code for `memcpy` was still not a part of the executable. This is because, with default options to `gcc`, the library containing `memcpy` is only linked with the executable at runtime. This is call dynamic linking and involves the executable determining the location of such shared library functions only at runtime.

The important part is that by specifying the "-static" flag, we can make `gcc` include the library code that contains `memcpy` into the executable.

> **Task:** Specify the "-static" flag when you compile memcpy_test.c into an executable. Use `objdump` again and see what it outputs, and try to see what the function `memcpy` actually does.

Things to note.

Whenever a command-line program generates a ton of output, you can use output redirection in the shell to save the output to a file. This makes it easier to search through the output and perform operations on it.

So, for example, you can do "objdump -S a.out > output.txt" and save the output of `objdump`. Do this for the previous task and search for the string "memcpy." Just try to see where the function actually is.

Also, it might be worth your time to look at the size of the program when it's compiled with "-static" versus when it's not. Use the command "ls -lh" to easily see the sizes of files in your current directory. Also note the size of the redirected `objdump` output – you should make sure to delete this file before logging out.

Task 3: Compare execution times.

A little program named `gprof`.

`gprof` is a tool that lets you easily analyze the execution time of your program.

To use it, you need to compile your executable with the flags "-pg" and "-fprofile-arcs."

You then run your executable like normal. When it's finished running, then you can run the command `gprof` with your executable name as an argument. This will then print a lot of output to the screen. The most important information is at the beginning, so it may be preferable to run the command and pipe it through `less`, i.e. "`gprof` a.out | `less`." You can hit the letter 'h' while within `less` to read about how to use `less`.

> **Task:** Compile your two test files with ONLY the "-pg" and "-fprofile-arcs" flags. Run the programs. Compare the `gprof` output. Now compile memcpy_test.c with the "-static" flag and view its `gprof` output. What has changed?

Compiling with optimization.

You can specify that `gcc` should compile with optimizations by giving it flags like "-O3." That's Oh (as in the letter) Three. There are different optimization levels; only O, O1, O2, and O3 are valid options.

> **Task:** Compile your two test files into assembly with optimization turned on. Has anything changed from before (without optimization)?. Compare the execution time using `gprof` with and without optimization for both files.

Task 4: Use `objdump` with C++ executable.

Understanding C++ assembly code can be a lot harder than understand C assembly code. To understand the difference, try looking at the assembly code for one of your previous labs written in C++.

> **Task:** Using code from a prior lab written in C++, look at the `objdump` output for an executable that is partly comprised of source code that contained classes and other C++ features. Do you notice any major differences in readability between C and C++ compiled assembly code?