Lab 7.2

Out: Friday, March 18, 2005

What you'll learn.

You'll be continuing the assembly examination we began in the last lab. We'll now begin to look at some of the optimizations that the compiler can do for you and how it can impact the assembly code generated.

How you'll do it.

You'll be writing one or two very simple .c files that now contain fewer than 30 lines. The first will be a program that uses x86 asm (provided) in your C code in order to make a C program that takes an infinite number of arguments.

Task 1: From C−> Assembly.

We all know that C can be compiled into object code or to assembly code by now. But we haven't really looked into how we can insert assembly code into our C programs. In this simple program we want you to insert a specified line of assembly so that you can take an infinite number of arguments into your program. Armed with this, you can now create a math program that will sum many numbers.

Task: Create a .c file, and name it inf_sum.c. Recall from lecture that an ellipses (...) implies an infinite number of arguments. Write a function that takes an infinite number of arguments. Your function should take an integer as its first argument specifying how many integers to sum. You should also maintain a pointer to an integer representing the top of the current stack. To get the current stack base pointer in your summation function, insert the following line of assembly: `asm("movl %%ebp, %%eax":"=a"(stack_pointer));`, where `stack_pointer` is an `int*`. You should then generate a sum of ints, by using `stack_pointer` and the first argument to the function (the number of ints to sum). To get the items out of the stack, add 3 + i to stack_pointer, where i is the ith element from the top of the stack. Explain to a TA why we need to add 3. What is the relation between the function calls in assembly and the number 3? Think about what is necessary when you make a function/subroutine call in assembly. You may need to look at the assembly in order to answer this question.

Task: Modify your code from before to use the pointer to the first integer argument instead of the stack base pointer. Can you still compute the sum of a variable number of integers? Try compiling with optimization level -O3, and see how this affects your code. Does it still work appropriately? If not, why doesn't it? If so, why might optimizations features like inlining sometimes cause problems for tricky variable-argument functions?

The "Real Way"

The better way to accomplish a function with an infinite number of arguments without mucking around in assembly is to use the `va_arg` functions. Check out `man va_arg` for an explanation. This method will be portable to other architectures, is easier to understand, and won't fail under optimization. It is what is used in the implemention of functions like `printf`.

Task 2: Let's override some functions...

malloc

By now, all of you know (or SHOULD know) the friendly beast known as malloc. If you're not sure what it does, `man malloc`. For an executive summary, malloc will allocate size bytes off of the heap and give a pointer to that memory to you. We're going to put a little "fun" in this function.

Dynamic libraries

We will be building what is known as a dynamically linked library/shared object. As you may or may not know by now, getting a C/C++ program up and running is divided into several parts, preprocessing, compilation, linking and loading. Preprocessing and compilation are done in order to generate the proper assembly/object code from your C code. However, as we have done in several labs, there have been a lot of `#include` statements at the top of your programs. These signify to the compiler and linker/loader that other libraries' code is needed in order to get yours up and running. A majority of these libraries are stored in files known as shared object files. These are useful because they allow multiple programs to use their code, thus cutting down on the number of times a given function need be defined and declared. As you probably noticed in the last lab, statically compiled programs will get much much bigger. Additionally, only one copy of the shared object need be in memory at a given time, and all can access it.

You can also specify that the linker and loader look a place other than the standard shared object files to get the function definitions that you want to use. This is useful for debugging if certain library functions don't do exactly what you want. This is done by changing the environment variable `LD_PRELOAD`. This should specify the library that you want to use as your new shared object source file. Additionally, you should set the environment variable `LD_LIBRARY_PATH`, which specifies to the loader where it can find the library files, especially the one specified in `LD_PRELOAD`. Read the manpages on `gcc` to see what is going on behind the scenes. Assuming that most of you are using the `tcsh` shell, you can use the command `setenv LD_PRELOAD libraryname` to change an environment variable for your shell.

So for this task, we'll be creating our own shared object file library that will basically act as a wrapper around the memory allocation function `malloc`. This function will basically display the arguments specified to malloc, and then call the "real" malloc. It should then display the actual pointer returned to the user.

For malloc, this is easy enough: there is a standard hook that wraps around malloc that is called `__malloc_hook.` Look at the manpage for this function, and copy the code that is in the manpage to a .c file. Compiling the c file into a shared object file can be done with the following command: `gcc -g -fPIC -ldl -shared malloc_lib.c -o malloc_lib.so`.

Task: First, get the shared library compiled, and change your environment variables so that you can see all of the malloc calls being made from compiled binaries. Try Firefox, or Mozilla, or Gaim. Watch how the memory is being used, and the pointers that malloc returns.

Task: Modify the library you just wrote to display a histogram of the first 100 allocations. Print out a * for each occurrence of a malloc size within a particular range of bytes. You can specify the ranges so long as they are not unreasonable (i.e. one * for each megabyte or less). A reasonable range might be powers of 2 bytes, like 0-8 ,8-16, 16-32.

A generic wrapper

Not every function has hooks, so writing a generic wrapper is not as easy as this one. We have provided a .c file that shows how to write a generic wrapper for malloc in `/course/cs034/pub/lab7`. The code to do this is not very long. But getting all of the nuanced syntax in order for the library to do exactly what we want is difficult to configure because of the libraries and functions that this task requires. Read the manpage for `dlsym` to see what you're up against. Notable in this manpage and in our .c file is RTLD_NEXT. This is a handle argument that is sent to the dlsym function, and looks for the standard definition of the function that you are passing in quotes. You will need to #include the dlfcn.h header.

Task: OPTIONAL: Read the manpage for `dlsym` and `dlopen`, then look at the source file provided. Understand what the two functions in the program are doing, and how they should work.