CSE 361S Intro to Systems Software
Lab #2

Due: Thursday, September 22, 2011

Introduction

This lab will introduce you to the GNU tools in the Linux programming environment we will be using for CSE 361S this semester, which includes gcc, as, and gdb. You will learn how to compile, assemble, execute, and debug C and assembly language programs.

Assembling and Executing

1. Login and open a terminal window.

   Login using your cec account.

   If you are in the Windows lab (Lopata 401), boot the Linux virtual machine (VM). Login to the VM, again using your cec account.

   After you are logged in to Linux, click on Applications -> System Tools -> Terminal to open a new terminal shell window.

   The terminal window will default to the home directory for your cec account. If you wish, use the mkdir {name} command to create a sub-directory with the name of your class (e.g. use the command: mkdir cse361s). Then switch to this directory using the cd {name} command (e.g. cd cse361s).

2. Download files for Lab 2.

   Download the files needed for Lab 2 into the current directory (either your home directory or the directory you just created) by going to the class webpage and right-clicking on the link labeled cse361s_lab2.tar and selecting "Save Link Target As...").

   Unpack the files using the Linux command tar xf cse361s_lab2.tar. This will create a directory named lab2 in your directory. This directory contains the C and assembly files needed for this lab.

3. Compile/Assemble the Lab 2 programs.

   After untarring the Lab 2 files, use the cd lab2 command to switch to the lab2/ directory. Then compile and assemble the programs for Lab 2 using either the make or the make all command (make and make all amount to the same command since typing make by itself defaults to make all). Note: make -n will show you which
commands make will execute, without actually executing them. This can be quite helpful at times.

In the process of "making" the programs, you will see a listing of the commands used to compile and assemble the C and assembly programs using gcc and as. It is recommended that you try executing each one of these commands on your own and use ls to check what files are generated as a result of each command. Be sure to use the make clean command to delete all the executables and .o files before executing the "make" commands on your own.

You may also want to use the less Makefile command to see how the Makefile performs the compilation/assembly process.

4. Execute the C and assembly executables.

After "making" the programs, try running the C and assembly programs using the lab2_cprog and lab2_asmprog executables, respectively. To execute a program that resides in the current directory, you need to tell the shell to look in the current directory. The current directory is denoted with a dot, ".", so lab2_cprog is executed with the command:

./lab2_cprog

5. Debug the C executable, lab2_cprog.

To begin debugging the C executable using the gdb debugger, execute the command: gdb lab2_cprog.

We will start off by introducing a few simple debugging commands in gdb.

- First of all, start running the program in gdb.

To begin debugging, you must first begin executing the program within gdb. When gdb starts up, the program is merely loaded; it is not yet executing. Before running the program, use the break main command to set a break point at the first instruction in main(). Then type run (or r) to begin running the program. You will note that the program breaks immediately at the entrance to main().

- Use the stepi (or si) command to trace into instructions.

The stepi (or si) command traces into each instruction. Effectively, this means that it executes the next single machine/assembly instruction. For procedure or interrupt calls, this means that it executes only the instruction that transfers control to the procedure, causing it to jump into that procedure/interrupt (i.e. it steps 'into' the procedure/interrupt).
Use the `si` command a couple times to watch the program's progress as you step through the code. When you get to the function call for `get_number()`, notice that you actually "step into" the procedure so that you can watch the execution of each individual statement in that procedure. If you ever want to re-start from the beginning of the program, simply use the `run (r)` command again, which will re-start execution of the program from the beginning, stopping immediately at the first breakpoint (at `main()`).

- Use the `nexti` (or `ni`) command to trace over instructions.

The `nexti` (or `ni`) command traces over each instruction. Effectively, this means that it executes the next single instruction in the active procedure. For procedure or interrupt calls, this means that it executes the full procedure or interrupt (i.e. it steps 'over' the procedure/interrupt).

Try stepping through the program again, but this time use the `ni` command instead. Notice that this time, you don't jump into the `get_number()` function. Stepping "over" this function simply causes it to execute "behind the scenes", so that you don't have to watch all the gory details if you don't want/need to.

- Use the `step` and `next` commands and compare them to `stepi` and `nexti`.

Use the online `help (h)` facility to learn about `step` and `next`. Compare them with `stepi` and `nexti`. You will likely find these two commands more convenient when working with C code.

- Watching variables.

Sometimes it is desirable to watch how the value of a variable changes as you trace through the program. It is possible to watch variables in this fashion using the `display {variable name}` command.

For example, re-start execution from the beginning of the program again, but this time use the command `display n` to watch the value of variable "n" as you step through the beginning of the program. To see a full listing of all the variables that are currently being watched, use the `info display` command. Likewise, if you want to delete a variable from being watched, you can use the command `delete display {variable number}`.

- Display memory.

Often, you will want to watch a whole block of memory as opposed to a small set of variables. There's no way you can automatically have a block of memory displayed after each step like you can with variables, but you can still easily display the contents of memory with a single command after each step. The command `x/{number of bytes} {start address}` will display the specified number of bytes starting at the specified...
address (e.g. \texttt{x/32 0x7f3e9a10} will show the contents of memory between addresses 0x7f3e9a10 and 0x7f3e9a2f, inclusive. Furthermore, to prevent you from having to re-type this command after each step, you can simply use the up and down arrows to scroll through the list of commands recently executed in \texttt{gdb}.

The primary problem in displaying memory is that you must know the address(es) of the memory locations you which to view. Fortunately, the memory addresses of variables can be easily determined in \texttt{gdb} using the \texttt{print \&\{variable name\}} command (e.g. \texttt{print \&n} will give the address for variable "n").

- Display registers.

It is also often useful (moreso with assembly debugging than C debugging) to view the values in the register file. Use the \texttt{info registers} command to do this.

- View (local) program code.

While the \texttt{si} and \texttt{ni} commands display the next instruction to execute, you may often want to see more complete sections of code. You can display the 10 lines of code around the next statement to execute in the debugger using the \texttt{list} (or \texttt{l}) command. Or if you want to display the 10 lines of code around a function declaration, you can use the \texttt{list \{function name\}} command (e.g. \texttt{l main}).

- View assembly code.

An alternate way to view the assembly code (or when debugging in C and needing to view the equivalent assembly), use the \texttt{disassemble \{label/function name\}} (or \texttt{disas \{label/function name\}}) command (e.g. \texttt{disas main}).

- Other debugging functions:

Exit the debugger using the \texttt{quit} (or \texttt{q}) command.

Continue executing a program using the \texttt{continue} (or \texttt{c}) command. Execution will only stop after reaching a breakpoint, the end of the program, or a similar end-of-program procedure/interrupt call. This function is particularly useful in conjunction with breakpoints.

Also feel free to experiment with other debugging functions. You can use the \texttt{help} (or \texttt{h}) command to find directions for using other debugging utilities.

Continue to trace through \texttt{lab2_cprog} in \texttt{gdb} and examine what each instruction does. Be sure to use \texttt{si} and \texttt{ni} as needed.

6. Debug the assembly executable, \texttt{lab2_asmprog}.
To begin debugging the assembly executable using the *gdb* debugger, execute the command: 

```
gdb lab2_asmprog
```

Proceed to trace through the `lab2_asmprog` executable just like you did with the C program -- first start by setting a break point at the main function, run the program, and then proceed to step through following the break point. Note that in assembly, any label in the source code (e.g. `main:`, `i_loop:`, `exit_i_loop:`, `print_loop`, etc.) is treated as a function name, so it works best to use `si` when stepping through the code, but making sure to use `ni` on those assembly function calls to external functions (e.g. `get_number` and `printf`).

7. Questions:

Try to answer the following questions. Put your answers in a file using the same naming convention as before: `<lastname><firstinitial>_lab2.txt`, and upload your file to telesis for submission.

1. What number series is produced by the C program? By the assembly program? Are the two programs equivalent?
2. What happens when the user enters a value not in the allowable range?
3. What do the `CMP` and `JAE` instructions do? What does `JMP` do? How are they similar? How are they different?
4. What are the `#define` preprocessing directives for in the C program? Is there an equivalent in the assembly program (and if so, what)?
5. You likely found the experiences of debugging in C versus debugging in assembly both somewhat similar and somewhat different. What debugging commands did you find more useful for debugging C? For debugging assembly?
6. How could you change the C code so that it printed out every 2nd odd number, starting from 1 (i.e. 1, 5, 9, 13, etc.)? How could you similarly change the assembly code (*Note: this is a bit tougher to figure out...*)?
7. **Challenge Question:** How might you go about changing the C code so that it requested two numbers, `n` and `x`, so that the program would print out `n` numbers including every `x`th number starting from 1? How about in the assembly code (*Note: this is the really tricky part...*)?