CS 105
Lab 2: Debugger
Playing with X86 Assembly

See class calendar for lab and due dates

Introduction and Goals

The goals of this assignment are to do some basic investigation of the X86 architecture and assembly language, and to begin learning how to use GDB. The lab pages have links to a quick GDB summary and to a printable GDB reference card; you can also find other information on Google.

It will be useful to know that you can get the compiler to generate the assembler source for a program by using "`gcc -S foo.c`". You should also know that to use the debugger effectively, you will need to compile with the "`-g`" switch. In fact, you should just get in the habit of always compiling with "`-g`"; the situations where it's undesirable are extremely unusual. (Along the same lines, it's usually wise to compile *without* "`-O`" because that will make debugging more difficult, and debugging is nearly always more important than optimization.)

Finally, note that the prologue and epilogue generated by the compiler may be different from what we saw in class. In particular, the prologue often contains other instructions that manipulate the stack pointer, and the epilogue may use the `leave` instruction as a substitute for the `mov`/`pop` pair. The general rule of thumb is that any instructions that add to, subtract from, or `and` with the stack pointer are definitely part of the prologue, and (for this assignment) can safely be ignored.

Collect your answers to all of the following questions in a plain-text file named "`lab02.txt`". Identify each section by problem number, and each answer by question number.

Submission. Use `cs105submit` to submit ONLY `lab02.txt` answer file.

Problem 1—Debugging Optimized Code (16 Points)

Download the file `problem1.c` from the lab Web site. This file contains a function that has a small `while` loop, and a simple `main` that calls it. Briefly study the `loop_while` function to understand how it works (you don't need to fully decode it; just get a clue about what's going on).

It will also be useful to know what the `atoi` function does. Type "`man atoi`" in a terminal window to find out.

Finally, it will be useful to have a slight clue about `printf`. Since `printf` is quite complicated, for now we'll just say that it prints answers, and `"%d"` means "print in decimal". We encourage you to read more about `printf` in Kernighan & Ritchie or online.

Compile the program with the `-g` switch and with **no** optimization. Run `gdb problem1` and set a breakpoint in `main` ("`b main`"). Run the program by typing "`r`" or "`run`". The program will stop in `main`.

1. Type "`c`" (or "`continue`") to continue past the breakpoint. What happens?
2. Type "`bt`" (or "`backtrace`") to get a trace of the call stack and find out how you got where you are. Take note of the numbers in the left column. Type "`up` *n*", where *n* is one of those numbers, to get to `main`'s *stack frame* so that you can look at main's variables. What line are you on?

3. Usually when bad things happen in the library it's your fault, not the library's. In this case, the problem is that `main` passed a bad argument to `atoi`. There are two ways to find out what the bad argument is: look at `atoi`'s stack frame, or print the argument. Figure out how to look at `atoi`'s stack frame. What argument was passed? What is wrong with that value?

4. Rerun the program with an argument of 5 by typing "`r 5`". Continue from the the breakpoint. What does the program print?

5. Type "`r`" (without any further parameters) to run the program yet again. When you get to the breakpoint, examine the variables `argc` and `argv` by using the `print` command. For example, type "`print argv[0]`." Also try "`print argv[0]@argc`", which is gdb's notation for saying "print elements of the `argv` array starting at element 0 and continuing for `argc` elements." What is the value of `argc`? What are the elements of the `argv` array? Where did they come from, given that you didn't add anything to the `run` command?

6. The `step` or `s` command is a useful way to follow a program's execution one line at a time. Type "`s`". Where do you wind up?

7. Gdb always shows you the line that is about to be executed. Sometimes it's useful to see some context. Type "`list`" What lines do you see? Hit the return key. What do you see now?

8. Type "`s`" to step to the next line. Then hit the return key three times. What do you think the return key does?

9. What are the values of `result`, `a`, and `b`?

10. Type "`quit`" to exit gdb. (You'll have to tell it to kill the "inferior process", which is the program you are debugging. Insulting!) Recompile the program, this time optimizing it with -O1. Debug it, set a breakpoint at `loop_while`, and run it with an argument of 10. Step three times. What four lines are shown to you? Why do you think the debugger is showing you those lines in that order?

11. Quit gdb again and recompile with -O2. Debug the program and set a breakpoint in `loop_while`. Run it with an argument of 20. Where does the program stop?

12. Hmmm... that's kind of odd. Disassemble the `main` function by typing "`disassem main`". What is the address of the instruction that calls `atoi`? What is the address of the instruction that calls `printf`?

13. That wasn't too hard. Where's the call to `loop_while`?

14. A handy feature of `print` is that you can use it to convert between bases. For example, what happens when you type "`print/x 42`"? How about "`p 0x2f`"?

15. Remember that functions return results in `%eax`. So the result of `atoi` will be in `%eax`. After the call to `atoi` there are four `lea`s (which are gdb's version of `leal`). What algebraic function do they compute on `%eax`? (You can ignore the `add` and the `push`es; they have to do with calling `printf`.)

16. Now you (kind of) understand the optimized `main`. What happened to the call to `loop_while`?

Problem 2—Stepping and Looking at Data (19 Points)

Download the file `problem2.c` from the lab Web site. This file contains three `static` constants and three functions. Read the functions and figure out what they do. (If you're new to C, you may need to consult your C book or some online references.) Here are some hints: `argv` is an array containing the strings that were passed to the program on the command line (or from gdb's `run` command); `argc` is the number of arguments that were passed. By convention, `argv[0]` is the name of the program, so `argc` is always at least 1. The `malloc` line allocates a variable-sized array big enough to hold `argc` integers (which is slightly wasteful, since we only store `argc-1` integers there, but what the heck).

By now we hope you've learned that optimization is bad for debugging. So compile the program without optimization (but with `-g`) and bring up the debugger on it.

1. Gdb provides you lots of ways to look at memory. For example, type "`print puzzle1`" (something you should already be familiar with). What is printed?

2. Gee, that wasn't very useful. Sometimes it's worth trying different ways of exploring things. How about "`p/x` puzzle1"? What does that print? Is it more edifying?

3. You've just looked at puzzle1 in decimal and hex. There's also a way to treat it as a string, although the notation is a bit inconvenient. The "`x`" (examine) command lets you look at arbitrary memory in a variety of formats and notations. For example, "`x/bx`" examines bytes in hexadecimal. Let's give that a try. Type "`x/4bx &puzzle1`" (the "`&`" symbol means "address of"; it's necessary because the `x` command requires addresses rather than variable names). How does the output you see relate to the result of "`p/x puzzle1`"? (Incidentally, you can look at any arbitrary memory location with `x`, as in "`x/wx 0x8048500`".)

4. OK, that was interesting and a bit weird (and we'll be covering it in class soon). But we still don't know what's in `puzzle1`. We need help! And fortunately gdb has help built in. So type "`help x`". Then experiment on `puzzle1` with various forms of the `x` command. For example, you might try "`x/16i &puzzle1`". (`x/16i` is one of our favorite gdb commands—but since here we suspect that `puzzle1` is data, not instructions, the results might be interesting but probably not correct.) Keep experimenting until you find a sensible value for `puzzle1`. (**Hint:** Although `puzzle1` is declared as an `int`, it's not. But on a 32-bit machine an int is 4 bytes, 2 halfwords, or one (in gdb terms) word.) What is the human-friendly value of `puzzle1`?

5. Having solved `puzzle1`, look at the value carefully. Is it correct? (You might wish to check it online.) If it's wrong, why is it wrong?

6. Now we can move on to `puzzle2`. It pretends to be an *array* of `int`s, but you might suspect that it isn't. Using your newfound skills, figure out what it is. (**Hint:** since there are two `int`s, the entire value occupies 8 bytes.) What is the human-friendly value?

7. Are you surprised?

8. Is it correct?

9. We have one puzzle left. By this point you may have already stumbled across its value. If not, figure it out; it's often the case that in a debugger you need to make sense of apparently random data. What is stored in `puzzle3`?

10. We've done all this without actually running the program. But now it's time to execute! Set a breakpoint in `fix_array`. Run the program with the arguments `1 1 2 3 5 8 13 21 44 65`. When it stops, print `a_size` and verify that it is 10. Did you really need to use a `print` command to find the value of `a_size`? (**Hint:** look carefully at the output produced by gdb.)

11. What is the value of `a`?

12. Step six times. You'll note that one of the lines executed is a right curly brace; this is common in gdb and often indicates the return from a function. After returning, what is the value of `a`?

13. Step again (a seventh time). What is the value of `a` now? What is `i`?

14. At this point you should (again) be at the call to `hmc_pomona_fix`. You already know what that function does, and stepping through it is a bit of a pain. The authors of debuggers are aware of that fact, and they always provide two ways to step line-by-line through a program. The one we've been using (`step`) is traditionally referred to as "step into"—if you are at the point of a function call, you move stepwise *into* the function being called. The alternative is "step over"—if you are at a normal

line it operates just like `step`, but if you are at a function call it does the whole function just as if it were a single line. Let's try that now. In gdb, it's called `next` or just `n`. What line do we wind up at? (Incidentally, in gdb as in most debuggers, the line shown is the *next* line to be executed.)

15. Use `n` to step past that line, verifying that it works just like `s` when you're not at a function call. What's `a` now?

16. It's often useful to be able to follow pointers. Gdb is unusually smart in this respect; you can type complicated expressions like `p *a.b->c[i].d->e`. Here, we have kind of lost track of `a`, and we just want to know what it's pointing at. Type "`p *a`". What do you get?

17. Often when debugging, you know that you don't care about what happens in the next three or twelve lines. You could type "`s`" or "`n`" that many times, but we're computer scientists, and CS types sneer at work that computers could do for them—especially mentally taxing tasks like counting to twelve. So on a guess, type "`next 12`". What line are you at?

18. What is the value of `a` now?

19. What is the value of `*a`?

Problem 3—Assembly-Level Debugging (19 Points)

So far, we've been taking advantage of the fact that gdb understands your program at the source level: it knows about strings, source lines, call chains, and even complicated C++ data structures. But sometimes it's necessary to get down and dirty with the assembly code.

To be sure we're all on the same page, let's quit gdb and bring it up on `problem2` again. Set a breakpoint in `fix_array` and run it with arguments of `1 42 2 47 3`.

1. What is the output? Whoop-tee-doo.

2. Set a breakpoint in `main`. Run the program again. Where does it stop?

3. Booooooooooring. Type "list" to see what's nearby, then type "`b 35`" and "`c`". Where does it stop now?

4. Shocking. But since that's the start of the loop, typing "`c`" will take you to the next iteration, right?

5. Oops. Good thing we can start over by just typing "`r`". Continue past that first breakpoint to the second one, which is what we care about. But why, if we're in the `for` statement, didn't it stop the second time? Type "`info b`" (or "`info breakpoints`" for the terminally verbose). Lots of good stuff there. The important thing is in the "address" column. Take note of the address given for breakpoint 2, and then type "`disassem main`". You'll note that there's a helpful little arrow right at breakpoint 2's address, since that's the instruction we're about to execute. Looking back at the corresponding source code, what part of the `for` statement does this assembly code correspond to?

6. The code at `+44` jumps to `main+95`, which has four instructions that jump back to `main+53`. This is all part of the `whilefor` loop pattern we covered in class. We've successfully breaked ("broken?" "Set a breakpoint?") at the initialization of the loop. But we'd like to have a breakpoint *inside* the for loop, so we could stop on every iteration. The jump to `main+53` tells us that we want to stop there. But that's not a source line; it's in the middle clause of the `for` statement. No worries, though, because gdb will let us set a breakpoint on *any* instruction even if it's in the middle of a statement. Just type "`b (*main+53)`" or "`b *0x8048509`" (assuming that's the address of `main+53`, as it was when we wrote these instructions). The asterisk tells gdb to interpret the rest of the command as an address in memory, as opposed to a line number in the source code. What does "`info b`" tell you about the line number you chose? (Fine, we could have just set a breakpoint at that line. But there are more complicated situations where there isn't a simple line number, so it's still useful to know about the asterisk.)

7. We can look at the current value of the array by typing "`p array[0]@argc`". But the current value isn't interesting. Let's continue a few times and see what it looks like then. Typing "`c`" over and over is tedious (especially if you need to do it 10,000 times!) so let's continue to breakpoint 3 and then try "`c 4`". What are the full contents of `array`?

8. Perhaps we wish we had done "`c 3`" instead of "`c 4`". We can rerun the program, but we really don't need all the breakpoints; we're only working with breakpoint 3. Type "`info b`" to find out what's going on right now. Then use "`d 1`" or "`delete 1`" to completely get rid of breakpoint 1. But maybe breakpoint 2 will be useful in the future, so type "`disable 2`". Use "`info b`" to verify that it's no longer enabled ("Enb"). Continue past breakpoint 1, where we're stopped. Where do we stop next? (Well, that was hardly a surprise.)

9. Sometimes, instead of stepping through a program line by line, we want to see what the individual instructions do. Of course, instructions manipulate registers. Quit gdb and restart it, setting a breakpoint in `fix_array`. Run the program with arguments of `1 42 2 47 3`. Type "`info registers`" to see all the processor registers in both hex and decimal. How many registers have *not* been covered in class?

10. Well, that's because they're weird and not terribly important. (Except `eflags`, which holds the condition codes among other things. Note that instead of being given in decimal, it's given symbolically—things like `CF`, `ZF`, etc.) Which flags are set right now? What preceding instruction caused those flags to be set?

11. Often, looking at *all* the registers is excessive. Perhaps we only care about one. Type "`p $eax`". What is the value? Is "`p/x $eax`" more meaningful? (**Hint:** compare the output to the numbers shown in the disassembly of `fix_array`.)

12. We mentioned a fondness for "`x/16i`". Actually, what we really like is "`x/16i $eip`". Compare that to the result of "`disassem fix_array`". Then, immediately after typing "`x/16i $eip`", hit the return key. (There is no gradeable question for this homework step, so the official question is "Are you still having fun?" If your answer is "Yes", professional help is available. But you still need to put your answer on the grading sheet.)

13. Finally, we mentioned stepping by instructions. That's done with "`stepi`" ("step one instruction"). Type that now, and note that gdb gives a new instruction address but still says that you're in the for loop. Hit return to `stepi` again, and keep hitting return until the displayed line `doesn't` contain a hexadecimal instruction address. Why doesn't gdb print an address?

14. It's useful to use "`x/16i $eip`" here to make sure we understand what's about to happen. You should see three `mov` instructions followed by a `call`. Use `stepi 3` to get past the `movs`. What instruction address will be executed next?

15. As with source-level debugging, at the assembly level it's often useful to skip over function calls. At this point you have a choice of typing "`stepi`" or "`nexti`". If you type "`stepi`", what do you expect the next instruction to be (hexadecimal address)? What about "`nexti`"? (By now, your debugging skills should be strong enough that you can try one, restart the program, and try the other, so there's little excuse for getting this one wrong!)

16. Almost there! Stepping one instruction at a time can be tedious. You can always use "`stepi` *n*" to zip past a bunch, but when you're dealing with loops and conditionals it can be hard to decide whether it's going to be 1,042 or 47,093 instructions before you reach the next interesting point in your program. Sure, you could set a breakpoint at the next suspect line. But sometimes the definition of "interesting" in *inside* a line. Let's say, just for the sake of argument, that you are interested in how the `leave` instruction works. You can set a breakpoint there by typing "`b *0x080484d2`". Do so, and then continue. What source line is listed?

17. The `leave` instruction manipulates registers in some fashion. Start by looking at what `%ebp` points to. You can find out the address with "`p $ebp`" and then use the `x` command, or you could just try "`x/x $ebp`". Or you could get wild and use C-style typecasting: "`p/x *(int *)$ebp/`' (try it!). What is the value?

18. Use "`info reg`" to find out what all the registers. Then use "`stepi`" to step past the `leave` instruction, and look at all the registers again. Which registers have changed, and what are their old and new values?

19. Where did the new value in `%ebp` come from?

20. Extra credit (2 points): Where did the new value in `%esp` come from?