Lab 03 - Atoi

Out: September 24-25, 2012

1 Introduction

Many programming languages, such as Java, provide a convenient mechanism for converting between integers and strings. Java, for example, allows integers to be concatenated with an empty string for conversion in one direction, and provides the static method `Integer.parseInt()` for conversion in the other direction. C provides a function `atoi()` (ASCII-to-integer), which converts a null-terminated[1] ASCII character string into an integer, in its standard library <`stdlib.h`>, and provides `sprintf()` to convert integers and other data types to character strings. Such language features shield the programmer from the underlying representation of these data types.

x86 assembly, however, is a language without such features; all data are treated the same by the language instructions — it is the responsibility of the programmer to interpret each datum and to use those data appropriately.

In this lab, you will be implementing an `atoi()` function of your own, to convert integers to character strings.

2 Assignment

This handout contains the following files:

- *atoi.c*: a C file which calls your x86 `atoi()` function.
- *my_atoi.h*: a header file declaring your x86 `atoi()` function.
- *my_atoi.s*: the assembly file in which you will be writing `atoi()`.
- *Makefile*: a makefile.
- *lab03.pdf*: this document.

In this lab, you will be implementing the `atoi()` function, which converts strings of ASCII characters into an integer, in x86. Converting between these data types will require you to deal directly with the byte-level representation of strings. You will be writing your code in the provided file *my_atoi.s*. The code for dealing with function calls is already provided in this file; you will only be writing the function body of `atoi()`. Also provided is a short C program which calls your function; use it to test your assembly program.

There are several ways to implement `atoi()`. The easiest algorithm is provided below.

[1] Rather than store metadata about the size of a string, strings in C are represented simply as arrays of contiguous characters. Consequently, the character which denotes the end of the string is the null-terminating character '`\0`.'

2.1 Algorithm

The simplest approach to implementing `atoi()` is to iterate through the characters of the string from left to right, updating an accumulator with the growing integer value. Initially, the accumulator is set to 0. At each character `c`, you multiply the accumulator by 10, and then add to it the value of the digit `c` (keep in mind that the character's ASCII code value is not the same as its integer value). When you reach the end of the array or a non-integer character, return the number in the accumulator:

```
atoi(char[] array):
    int accumulator = 0
    for each character c in array until first non-digit character:
        accumulator = accumulator * 10
        accumulator = accumulator + (integer represented by c)
    return accumulator
```

Some advice:

- Each ASCII character is exactly one byte long. Make sure that each x86 instruction you use to retrieve and manipulate each character operates on a single byte.
- Each subsequent character is offset exactly one byte from the previous character (which follows from the above).
- To retrieve the integer value of a digit character, you can subtract 48.

3 Assembling and Testing

For this lab, there are two principal ways to write and run x86: you can use either a text editor with the provided Makefile and `gdb`, or the recently-developed `x86scope` IDE. While `x86scope` provides syntax highlighting and several other features, some of its features are currently unsupported.

3.1 A Text Editor, Makefile, and `gdb`

You are provided with a makefile, which will build the `atoi` program for you. To build `atoi`, run `make atoi`, or just `make`. To clean up, run `make clean`.

After building the testing program, you can run it by typing `./atoi` or on the command line in your lab handout directory. This program will print the output generated by your program for all input you enter. Use them to verify the correctness of your output. Typing CTRL-D on a blank line (which closes the input stream) will terminate the test program.

In the previous lab you learned how to use `gdb` to debug C programs. `gdb` can also be used to debug x86 programs; several additional instructions, as well as the executable `objdump`, will make debugging x86 code a much less challenging task than it otherwise would be. Run your x86 program in `gdb` the same way you would run a C program: `gdb <executable>`.

Several new `gdb` instructions are very useful for debugging x86:

- `break *<address>` sets a breakpoint at the given address;
- `stepi` steps through a single x86 instruction;
- `print` can print values stored in registers and at memory addresses. To print the value contained in a register, use `print` followed by the register name with the '`%`' replaced by a '`$`'. For example,

  ```
  print $ebx
  ```

  prints the value stored in `%ebx`.

  You can use the values in registers as variables as well. For example, the following are valid `gdb` commands:

  ```
  print $ebx * 7 + 3
  print *(void**)($esp + $eax)
  ```

  You can access the value of the program counter with the variable `$eip`.

  Lastly, you can format numbers using different bases with various `print` arguments:

  - `/x` formats data as a hexadecimal number, which is extremely useful for printing pointers;
  - `/d` formats data as a decimal number; and
  - `/t` formats data as a binary number.
- `info registers` prints the value contained in each register and condition code.
- `disas` is used to "disassemble" the program, producing the x86 instructions corresponding to the raw bytes of the program. With no argument, this command disassembles the current function. The command can also take arguments such as a function name or an address range.

A handy "GDB cheatsheet" can be found on page 255 of the textbook.

In addition to using GDB, you may want to see the assembly code of your program in its entirety. The executable `objdump` provides a convenient way to do this. When run with the `-d` flag, `objdump` disassembles an executable file into x86, providing the address of each instruction next to the instruction itself in its output. Run

```
objdump -d <executable>
```

to see the disassembled x86. You may find it helpful to pipe the output of `objdump` to a file, which you can then view in the text editor of your choice.

3.2 Using `x86scope`

For this lab, we have provided you with a basic IDE that will help you debug your x86 code: `x86scope`. Once you install the lab, you will be able to run `x86scope` and get hacking!

The first thing you should do is import the lab directory into your workspace. In the File menu, there is an import option which will bring in all of the sources in the selected directory and set

everything up so it will compile. Point it at the directory where the source for the lab you just installed is. You should see it in your workspace after you are done with this.

Now, you will be able to start working on the x86 code. From the workspace panel on the left, open the *my_atoi.s* file and you will see some boilerplate code that sets up the function for you. Only edit code in the middle lest ye be zapped by the sea sorceress Ursula. Also, `x86scope` doesn't let you edit the C files, which you won't need to do for this assignment anyway.

When you are ready to run your program, you can use the options in the Build menu to compile and run (the play button will also do this for you when your program is not already running). Your program will open up in a new terminal window, in which you can interact with it. If you need a closer look at the inner workings of your program, you can set breakpoints by clicking the gutter next to the line numbers. When your program encounters a breakpoint during execution, the line will be highlighted and the register and stack views will update accordingly. You can use the buttons in the top bar to continue, step forwards one instruction, or continue to the end (the "step backwards" functionality promised by the first one is waiting on a `gdb` update and subsequent testing, so it's not functional at the moment).

If you encounter any bugs in your use of `x86scope`, send mail to `ejcaruso@cs.brown.edu` and explain what you were doing and what funny things started happening (include stack traces if possible). This program is still in very rough beta, so your help is appreciated!

4 Useful Hints

- Provided in the stencil for this assignment, and on the website, is a document *x86_cheatsheet.pdf* which contains a lot of useful information about x86 registers, conventions, and instructions. Use this document as a quick reference as you work with x86.
- If your `atoi()` function is given a string that is not a valid integer, it should not crash. However, like the `atoi()` function provided in C, it does not need to recognize this fact or display an error message. For the purposes of this assignment, your function should return the integer corresponding to all digits up to the first non-digit character in the string. If the first character is a non-digit character, your function should return 0.
- You do not need to handle negative integers as input. You should consider your integers unsigned for the purpose of this assignment — the TA-provided test programs will cast negative inputs to their unsigned equivalents. Similiarly, there is no need to handle overflow.
- It is useful to write out your program in C before you attempt to write it in x86. You should always provide a C or C-like high-level description of what your assembly code is doing in the comments to any code you write in assembly language — it makes your code much easier to follow.
- A common bug is to use the incorrect addressing mode, particularly with absolute addresses. Consult the x86 Cheat Sheet section on addressing modes if your code seems to be producing a lot of segmentation faults.
- Remember that strings in C are null-terminated — when your loop reaches the null byte, it has reached the end of the string.

The purpose of this lab is to become comfortable coding in x86 — with that in mind, the TAs will be willing to answer any conceptual questions that you may have. Assembly language can be very challenging at first — don't hesitate to ask for help if you need it.

5 Getting Checked Off

Once you've completed your `atoi()` program, you've finished the lab. Call a TA over to have them check your work. If you do not complete the lab during your lab session, you can get credit for it by completing it on your own time and bringing it to TA hours before next weeks lab. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.