Lab 03 - x86-64: atoi

Due: October 1, 2017 at 4pm

1 Introduction

Many programming languages, such as Java, provide a convenient mechanism for converting between integers and strings. For example, Java 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 *abstract* the underlying representation of these data types from the programmer, making them easier to use.

x86-64 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 utilize it appropriately.

2 Assignment

This handout contains the following files:

[1] Rather than store metadata about the size of a string, strings in C are represented simply as arrays of characters. Consequently, a different method of determining the end of a string is needed. C does this by using the null-terminating character '\0' as the last character in the array.

- *atoi.c*: a C file which calls your x86-64 atoi() function.
- *my atoi.h*: a header file declaring your x86-64 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 in x86-64. 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.

To get started, run the command **`cs0330_install lab03`**.

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

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 the value of character `c` (keeping 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 x86-64 instruction you use to retrieve and manipulate each character operates on a single byte.
- Each subsequent character is therefore offset exactly one byte from the previous character.
- To retrieve the integer value of a digit character, you can subtract 48. Why? (hint: consider the ASCII values for each digit character).
- The result for **`atoi`** of any non-digit value, like “hi,” should be 0. Non-digit inputs are illegal inputs for **`atoi()`**! So atoi should stop trying to convert to an int at the first non-digit character. For example, “55g” should return 55.

3 Assembling and Testing

For this lab, you will use a text editor with the provided Makefile and gdb to write and run x86-64.

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 it 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 x86-64 programs; several additional instructions, as well as the executable objdump, will make debugging x86-64 code a much less challenging task than it otherwise would be. Run your x86-64 program in gdb the same way you would run a C program: **`gdb <executable>`**.

Several new gdb instructions are very useful for debugging x86-64:

- **`break *<address>`** sets a breakpoint at the given address;
- **`stepi`** steps through a single x86-64 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 $rbx
  ```

  prints the value stored in **`%rbx`**.

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

  ```
  print $rbx * 7 + 3
  print *(void**)($rsp + $rax)
  ```

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

  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 x86-64 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 the class website, or 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 x86-64, providing the address of each instruction next to the instruction itself in its output. Run

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

to see the disassembled x86-64. 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.

4 Useful Hints

- On the website, you can find the x86-64 cheatsheet which contains a lot of useful information about x86-64 registers, conventions, and instructions. Use this document as a quick reference as you work with x86-64. All the information it provides except for the information on procedure calls should be useful for this lab.
- 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. Similarly, there is no need to handle overflow.
- It is useful to write out your program in C before you attempt to write it in x86-64. 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 x86-64 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 x86-64. 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. Submit your work using **`33lab_checkoff lab03 [--verbose]`**. You can run this script as many times as you wish to check what your grade was - it will never lower your grade. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.