Lab 4: Function Calls

CS/CoE 0447(B) Spring 2015 — Dr Kosiyatrakul (Tan)

Released: Friday 6 February 2015, 06:00 EST
Due: Thursday 12 February 2015, 23:59 EST

Submission timestamps will be checked and enforced strictly by CourseWeb; **late submissions will not be accepted.** Remember that, per the course syllabus, if you are not marked by your recitation instructor as having attended a recitation, your score will be cut in half. **Follow all instructions.**

For this lab, you are going to create functions in assembly and use those functions in a simple program. All functions **must** follow the calling conventions discussed in class, which are as follows:

- Use the `jal` instruction to call a function.
- Use `jr $ra` to return back to the caller.
- Use the four **$a registers** to pass arguments to a function.
- Use the two **$v registers** to return values back to the caller.
- For clarity and consistency, prefer lower-numbered registers like $a0 and $v0 before using higher-numbered ones.
- The eight **$s registers** are "saved registers." A function should save the values of these registers to the stack if it will overwrite their values, and restore them before returning to the caller.
- The ten **$t registers** are "temporary registers." A caller should save the values of these registers to the stack before calling a function if they will still be needed after the function returns, at which point it should restore them.

Because the functions you will need for this program depend on one another, you should write each of the functions in the order detailed below before attempting to write the main program.

Function 1: `_strLength`

The `_strLength` function should return the length of a null-terminated string.

- **Argument:** $a0 = address of a null-terminated string
- **Return value:** $v0 = length of the specified string

Note that the null character has a value equal to 0. You can load the value of a byte at a given address in memory using the `lbu` ("load byte unsigned") instruction. Combined with the loop, comparison, and branching constructs discussed in previous labs, you can detect when a null character is reached, then simply return the number of characters in the string to the caller.

Function 2: `_readString`

The `_readString` function should read a null-terminated string into memory from user input.

- **Arguments:**
  - $a0 = address of an input buffer
  - $a1 = maximum number of characters to read into the buffer
- **Return values:** None.

In MIPS, **syscall 8** can be used to read a string from keyboard input. However, this function has a strange behavior. In order to signal the end of the input string, the user must press the "Enter" key; however, MARS includes the newline character (`\n`) as part of the string that has been input before terminating the string with a null character (`\0`). For example, if we use syscall 8 to read a string and the user types "`Hello`" then presses "Enter", the buffer will contain "`Hello\n\0`".[1]

Your implementation of `_readString` should make use of syscall 8, but should behave more like Java's `Scanner` object, where the buffer does **NOT** contain the newline character. That is, if the user enters "`Hello`" then presses "Enter", the buffer should simply contain "`Hello\0`".

Don't forget to create an input buffer to store the input string. This can be done by creating a space in data memory as follows:

```
.data
    buffer:    .space    64
```

The above example reserves a space of 64 bytes, and the address of this chunk of memory can be accessed by the label `buffer` using the `la` ("load address") instruction (*e.g.*, `la $t0, buffer`).

Note that syscall 8 requires two arguments: the address of an input buffer, and the maximum number of characters to read. So your `_readString` function needs to take the same arguments. **Don't worry about the user entering more than the maximum number of characters;** let syscall 8 take care of it.

REQUIREMENT: This function **MUST** use the function `_strLength` you wrote above to figure out where to put the null character. For example, the buffer you receive from syscall 8 may contain "`Hello\n\0`". When you give this string to `_strLength`, it should return `6`, since the string contains five letters and a newline. This gives us a hint that we should place a null character at index 5, so as to overwrite the newline character and obtain the straightforwardly null-terminated string we want. **You may make the simplifying assumption that syscall 8 will always return strings with a newline character right before the null character.**[2]

[1] This behavior actually follows the traditional semantics of UNIX's `fgets` command. (In case you were curious.)

[2] The only situation in which this assumption wouldn't be valid is when the user attempts to enter more characters than the number specified. So in other words, we're assuming that the user will not enter a string that is longer than our buffer allows. Read up on the MARS syscall reference for more details.

Function 3: `_subString`

The `_subString` function should work a lot like the `substring()` method of Java's `String` class.

- **Arguments:**
  - $a0 = address of an input string
  - $a1 = address of an output buffer
  - $a2 = start index for the input string (inclusive)
  - $a3 = end index for the input string (exclusive)
- **Return value:** $v0 = address of the output buffer (same as $a1)[3]

Your `_subString` function must be able to gracefully handle several edge cases with respect to invalid start and end indices. In particular, if the end index provided by the user is greater than the length of the string, set it to be equal to the length of the string. If either index provided is less than 0, or if the end index is less than the start index, the output buffer should be set to the empty string (that is, its first character should be `\0`). **You must use your `_strLength` function to help detect these cases.**

The main program

Now that you have your functions in place, use them to write a main program which asks the user to enter a string. Once the user presses "Enter", your program should report the length of the input string on the console screen, then ask user to enter a start index and an end index to their desired substring. Lastly, compute and display the substring associated with specified indices.

A few sample executions of the program are below (user input is bold):

Enter a string: **Hello** This string has 5 characters. Specify start index: **1** Specify end index: **4** Your substring is: Ell	Enter a string: **baseball** This string has 8 characters. Specify start index: **6** Specify end index: **2** Your substring is:
Enter a string: **I love Pittsburgh!** This string has 18 characters. Specify start index: **0** Specify end index: **11** Your substring is: I love Pitt	Enter a string: **CS 0447 rules!** This string has 14 characters. Specify start index: **4** Specify end index: **23** Your substring is: 447 rules!

▶ **Save your program as `lab04.asm`. Please match the provided function names, calling conventions, and output format exactly. Submit your program via CourseWeb.**

[3] We'll return this address solely for convenience. This must be the same value as we passed in via $a1, which means you'll probably want to store it somewhere for safekeeping so you don't accidentally overwrite it.