CS 252 (Summer 19): Computer Organization

Assembly Project #4

Variables on the Stack

due at 5pm, Thu 1 Aug 2019

1 Purpose

In this Project, you'll write functions which use variables which must be stored on the stack. You **must not** use global variables; you must use extra-large stack frames.

1.1 Required Filenames to Turn in

Name your assembly language file `asm5.s`.

1.2 Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- `add, addi, sub, addu, addiu`
- `and, andi, or, ori, xor, xori, nor`
- `beq, bne, j`
- `jal, jr`
- `slt, slti`
- `sll, sra, srl`
- `lw, lh, lb, sw, sh, sb`
- `la`
- `syscall`

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), **do not use them!** We want you to learn the fundamentals of how assembly language works - you can use fancy tricks after this class is over.

1.3 Files

You can find the files for this project in the following locations:

- The GitHub Classroom repository
- The project directory on the class website:
  `http://lecturer-russ.appspot.com/classes/cs252/summer19/asm/asm5/`.
- The mirror of the class website, accessible on any department computer:
  `/home/russelll/cs252m19_website/`

2 Tasks

Your file must define the functions listed below:

2.1 Task 1: `countLetters()`

Implement the following function. You **must** use stack space to store the array. (Notice that I've carefully designed this function so that there are more `ints` to store than you have available registers!)

You **must not** use any global variables! (If you use `.data` for anything other than string constants, you probably are breaking this rule.)

```
void countLetters(char *str)
{
    int letters[26];    // this function must fill these with zeroes
    int other    = 0;

    printf("----------------\n%s\n----------------\n", str);

    char *cur = str;
    while (*cur != '\0')
    {
        if (*cur >= 'a' && *cur <= 'z')
            letters[*cur-'a']++;
        else if (*cur >= 'A' && *cur <= 'Z')
            letters[*cur-'A']++;
        else
            other++;

        cur++;
    }

    for (int i=0; i<26; i++)
        printf("%c: %d\n", 'a'+i, letters[i]);
    printf("<other>: %d\n", other);
}
```

2.2 Task 2: `subsCipher()`

Implement the following function, which is a "substitution cipher." A substitution cipher is an ancient style of encryption, where each letter or symbol is replaced with another; in order to encode or decode the message, you simply need a table which "maps" from the cleartext to the ciphertext[1].

In this function, I will pass you a string (which is null-terminated), and a "map," which is a table of characters[2]. The table will have 128 entries (since the "normal" ASCII characters only use up the first 128 possible values in a byte); your program will look up each character in the message, and encode it, using the map[3]. However, you must not modify the original string - and so you must store the encoded message in a buffer that is allocated on the stack.

This function calls `strlen()` - you will have to provide the implementation for this function.

Most importantly, this function uses an array variable which has a **length which is not known until the function runs.** This is not something which is **not** typically allowed in C - but which is relatively straightforward to implement in assembly. See the discussion below to see how it works.

```
void subsCipher(char *str, char *map)
{
    // NOTE: len is one more than the length of the string; it includes
    //       an extra character for the null terminator.
    int len = strlen(str)+1;

    int len_roundUp = (len+3) & ~0x3;
    char dup[len_roundUp];    // not legal in C, typically.  See spec.

    for (int i=0; i<len-1; i++)
        dup[i] = map[(int)str[i]];
    dup[len-1] = '\0';

    printSubstitutedString(dup);
}
```

NOTE: The bitmask constant above is the **bitwise negation** of `0x3`. In other words, it is "every bit set except for the two lowest." If you accidentally use negative three, you will get an incorrect result, because the negative three (in two's complement) is `1111...1101` .

[1]Substitution ciphers are notoriously easy to crack. They can often be cracked by hand - and so are never used for any important secrets!

[2]Since the length of the map is fixed, it is not required to have a null terminator. However, my testcase adds one, just so that it's possible to print out the map. This null terminator should never be used by your code.

[3]Since the map is only 128 characters long, there would be a serious problem if I sent you any input string which used the upper 128 values - you'd be accessing undefined memory. For this reason, I'll promise to send you only strings which use the ordinary 128 ASCII characters.

3 Flexible-Length Arrays on the Stack

When you declare an array on the stack in C, you typically need to use a constant size for the array[4]. The simple reason for this is that the compiler needs to know the size of the stack frame, so that it can allocate the proper size (and so that it can load/store local variables).

However, assembly can break this rule. Since we have direct control of the stack pointer, we can decrement the stack pointer as much as we want, in order to expand the stack frame. But there are a couple of problems with this:

- We need some way to save the length of the buffer for later - so that we can shrink the stack frame again. There are a number of ways to do this - but a simple one is to simply store the length in a register.

  For this reason, I've saved the length in a C variable. You don't have to use this variable, but if you do, it will be easy to clean up.

- The stack pointer must always be word aligned. (Can you figure out why this is true?)

So, our code at the beginning of the function does three things:

- Calculates the size of the array we'll need (remembering to account for the null terminator)
- Rounds that length up to a multiple of four
- Allocates an array of that size on the stack

4 Good Exam Questions

This project has a couple of interesting things that you should study and understand - I think that they would be excellent Short Answer questions for a future exam.

First, how did we round the length up to a multiple of 4? Could you generalize this to rounding up to a different power of 2? How would you modify this so that you could round up to a value which is **not** a power of 2? And how could you round down?

Similarly, could you adapt the "round up" or "round down" code to handle rounding a fraction after division? For instance, imagine that you had `count` items, and you needed to spread them across bins which could only carry 6 items. How many bins would you need?

[4]There are exceptions. Some compilers will allow you to declare an array which uses a variable as the length if this array is the **last local variable for that function.** In effect, the compiler is implementing exactly what we're doing in this project! However, I don't believe that all compilers allow this.

Additionally, there is a C library call, `alloca()`, which allows you to perform these sort of allocations. However, its use is discouraged, and it is not necessarily available in all environments.

Second, can you explain why the stack pointer always needs to be word-aligned? What would happen if it wasn't?

4.1 Requirement: Don't Assume Memory Layout!

It may be tempting to assume that the variables are all laid out in a particular order. Do not assume that! Your code should check the variables in the order that we state in this spec - but you **must not** assume that they will actually be in that order in the testcase. Instead, you must use the `la` instruction for **every variable** that you load from memory.

To make sure that you don't make this mistake, we will include testcases that have the variables in many different orders.

5 Running Your Code

You should always run your code using the grading script before you turn it in. However, while you are writing (or debugging) your code, it often handy to run the code yourself.

5.1 Running With Mars (GUI)

To launch the Mars application (as a GUI), open the JAR file that you downloaded from the Mars website. You may be able to just double-click it in your operating system; if not, then you can run it by typing the following command:

```
java -jar <marsJarFileName>
```

This will open a GUI, where you can edit and then run your code. Put your code, plus **one**[5] testcase, in some directory. Open your code in the Mars editor; you can edit it there. When it's ready to run, assemble it (F3), run it (F5), or step through it one instruction at a time (F7). You can even step **backwards** in time (F8)!

5.1.1 Running the Mars GUI the First Time

The first time that you run the Mars GUI, you will need to go into the `Settings` menu, and set two options:

- `Assemble all files in directory` - so your code will find, and link with, the testcase
- `Initialize Program Counter to 'main' if defined` - so that the program will begin with `main()` (in the testcase) instead of the first line of code in your file.

[5]Why can't you put multiple testcases in the directory at the same time? As far as I can tell (though I'm just learning Mars myself), the Mars GUI only runs in two modes: either (a) it runs only one file, or (b) it runs **all** of the files in the same directory. If you put multiple testcases in the directory, it will get duplicate-symbol errors.

5.2 Running Mars at the Command Line

You can also run Mars without a GUI. This will only print out the things that you explicitly print inside your program (and errors, of course).[6] However, it's an easy way to test simple fixes. (And of course, it's how the grading script works.) Perhaps the nicest part of it is that (unlike the GUI, as far as I can tell), you can tell Mars exactly what files you want to run - so multiple testcases in the directory is OK.

To run Mars at the command line, type the following command:

```
java -jar <marsJarFileName> sm <testcaseName>.s <yourSolution>.s
```

6 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

6.1 Testcases

You can find a set of testcases for this project in the GitHub Classroom; they are also present on the class website.

For assembly language programs, the testcases will be named `test_*.s` . For C programs, the testcases will be named `test_*.c` . For Java programs, the testcases will be named `Test_*.java` . (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted.

[6]Mars has lots of additional options that allow you to dump more information, but I haven't investigated them. If you find something useful, be sure to share it with the class!

These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcaes of your own, in order to better test your code.**

6.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_asm5`, along with a helper script, `mips_checker.pl`. Place both scripts, all of the testcase files (including their `.out` files), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac or Linux box, but no promises!)

6.3 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

7 Turning in Your Solution

You must turn in your code using GitHub Classroom. Turn in only your program; do not turn in any testcases or other files.