Computer Architecture
Lab 2

The goals of this lab are to work with strings and system calls.

You should turn in the source files for the two programs you're asked to write. Both documents should be submitted via the tool available on the course's Sakai site for this lab. These are due by the start of the next class period.

Syscall

System calls are services a program can request of the operating system. The MARS assembler simulates several of these. They are listed in your text on page A-44.

To activate a system call, you put the number for the routine you are requesting into register `$v0`, any arguments the call might require in registers `$a0-$a3,` and then issue the `syscall` instruction. This lab uses system calls for console I/O.

First, an Example

We can use the `.asciiz` directive to store strings in the `.data` segment:

```
         .data
greet:   .asciiz "Welcome to Computer Architecture!"
```

We now have memory locations beginning with `greet:` that hold the ASCII representation of the characters following the `.asciiz` directive. The string is **null terminated**, as we expect of our C and C++ strings. (If we don't care about null termination we can use the `.ascii` directive.)

To print this message to the screen, we'll need to load the address of it into `$a0` and put the number `4` in `$v0` (`4` is the number for the `print_string` system call.)

We can use our trusty `la` to accomplish the first and break out `li` to do the second. Since constants are called **immediate** values at the machine level, the `li` instruction stands for **load immediate.** Here is the code:

```
            .text
            la $a0, greet
            li $v0, 4
            syscall
```

Try this and make sure you see the message displayed on the little console window in MARS (down near the bottom on the RUN I/O tab.)

Printing an integer requires the **`print_int`** system call, which is number **`1`**. It takes an integer argument in **`$a0`** that it will print to the screen. Put a **`year: .word 2019`** data item in the **`.data`** section and add the following lines to the **`.text`** section:

```
la $t0, year
lw $a0, 0($t0)
li $v0, 1
syscall
```

Run this and see the year printed after the message. Be sure you understand what's going on.

We can mix string and integer output by successively calling system routines **`4`** and **`1`**. Add the following to the .data section:

```
wow:  .asciiz “Wow, “
sure: .asciiz “ sure was a great year!”
```

Now add code before and after the printing of the year to get the program to output “Wow, 2019 sure was a great year!” You should see it in the console in MARS. Ask me for help if you can't get this to work, or ask someone next to you in the lab.

One problem is that everything runs together. We can use escape characters just like in C and C++. Add a **`\n`** to the end of the **`greet:`** string (and the **`sure:`** string while you're at it) and then the **`wow:`** string should print on the next line.

Now, 2019 isn't over yet so I suppose we aren't assured it will be great, and at the very least it's awkward to say it *was* a great year. So, let's have the user input a year of significance: add a question in an **`askyr:`** label such as the following:

```
askyr:    .asciiz "What year did you start college?"
```

To get input from the console we need to call the **`read_int`** system routine. This is number **`5`** and will deposit the input value in register **`$v0`**. Let's write MIPS code to print our question to the screen, read in user input, and store the user's year in memory in the **`year:`** label.

Try to do this on your own! The answer is on the next page.

Our program thus far:

```
.text
la $a0, greet
li $v0, 4
syscall

la $a0, askyr
syscall

li $v0, 5
syscall
la $t0, year
sw $v0, 0($t0)

la $a0, wow
li, $v0, 4
syscall

la $t0, year
lw $a0, 0($t0)
li $v0, 1
syscall

la $a0, sure
li, $v0, 4
syscall
```

This is great! Now we don't need `cin` and `cout` anymore! We can do it this way!

Let's personalize the experience a bit and ask the user for his or her name. Create a second question for that:

```
askname: .asciiz "What is your name? "
```

Reading in a string is a bit more complex because we have to allocate some amount of storage for it and the `read_string` system call needs to know how much space is available.

Let's create a 20 character string for the name in a manner similar to how we allocated space for the `z` vector in the previous lab.

Here's the code to put in the `.data` segment:

```
name:      .word 0:19
size:      .word 20
```

Then create some sort of closing message that uses the name. Something like the following:

```
close1:    .asciiz "This has been fun, "
close2:    .asciiz ", let's continue our conversation another
                   time."
```

Now add the code to read in the string and print it out inside the closing message.

To call the **`print_string`** routine, we need to load number **4** into **`$v0`**, put the address of the destination string into **`$a0`**, and the size of the string into **`$a1`**.

Try it all out and see your program run! If you need help, here is mine:

```
.text
la $a0, greet
li $v0, 4
syscall

la $a0, askyr
syscall

li $v0, 5
syscall
la $t0, year
sw $v0, 0($t0)

la $a0, wow
li, $v0, 4
syscall

la $t0, year
lw $a0, 0($t0)
li $v0, 1
syscall
```

```
la $a0, sure
li, $v0, 4
syscall

la $a0, askname
syscall

la $a0, name
la $t0, size
lw $a1, 0($t0)
li $v0, 8
syscall

la $a0, close1
li $v0, 4
syscall

la $a0, name
syscall

la $a0, close2
syscall
```

There you go! Now you can do I/O in assembly! Reference figure A.9.1 on page A-44 of the text for more info on the available system services and their required registers.

You'll notice a line break after the name is printed to the console. This is because **`read_string`** contains the **`\n`** at the end of the input so when it's output again the **`\n`** creates a newline. You can get around this with a loop and the **`print_char`** routine, but I think this is enough for one lab so we'll just live with it for now.

Lab 2 Exercise

Now try one on your own.

Create a listing of a user's favorite discipline and courses within that discipline.

First, ask for a user's name. Then ask the user to type in his or her favorite Providence College discipline, using the three-letter code employed by the registrar (CSC, BIO, THL, etc.) Then ask for the three favorite course numbers in that discipline. Finally, output the results using reasonable prompts. Something like "Thanks, Billy, I see you like PHL courses X, Y, and Z. That is so great. That is all now. Thanks."

You should employ your software engineering skills to think about generalizing your code. Think about how to extend this to store any number of disciplines and courses. You don't have to code that up for the lab, but it's worth stretching your CS muscles a bit to at least consider how to use good software design skills to make this potentially more general and usable in a reasonably efficient manner.