Lab 8 - Write your own printf function

In this lab, you are going to implement your own printf function, in order to get a better understanding to the stack, function calling convention, and some other topics in computer architecture.

## Part 0: From nothing to something: Outputting a single character

“Print on the terminal” is among the most basic functionality that an operating system provides. Before the human kind developed fancy graphical user interfaces on a computer, people interacted with the computer via the terminal, which could display only ASCII characters, probably.

The OS only have to provide the functionality of “outputting a single character on the terminal screen” to start this task - once it can print a single character, it can print as many as it can.

However this simple functionality is an I/O operation, and is platform dependent. There are too many details to cover here. So we will use the function putchar(). This is the only function you may use in this lab. Call putchar('a') and the character “a” will be printed on the terminal screen.

## Part 1: Deal with indefinite number of arguments

Most of us are familiar with the function printf(), and we always use it like this:

printf("Elapsed time clocks: %d\n", clock() - time\_start);

Sometimes we may pass many arguments to it:

// It's really hard to debug OS code  
// so debug printf's are everywhere.  
printf("The process selected for execution is #%d: \'%s\' with a priority of %d\n", currpid, prptr->prname, prptr->prprio);

The thing that you might have noticed is that, printf() can actually accept as many arguments as you want. **It takes an indefinite number of arguments.**

The function prototype for printf() is specified as follows: (you can see it in the man page by typing man 3 printf in the terminal)

int printf(const char \*format, ...);

The first argument format, is always a string literal, and the following arguments, specified as ..., are the values to be printed. Functions taking parameters in ellipsis ... are called variadic functions.

### Get the arguments from the stack

The header file in the C library, <stdarg.h>, handles this problem by providing the type va\_list and other functions, but you shouldn’t use that.

How can we access the argument in the ellipsis? They don’t even have names that we can refer to. The idea is to use pointers to point to them.

Here is the key fact that we will use, which is the standard convention: **the arguments passed to a function are pushed onto the stack, in the *reverse* order of how they appear in the parameter list.** Therefore, the first argument, format, is pushed last onto the stack and is at the stack top, while other arguments lie deeper in the stack. You also need to keep in mind that the stack segment grows from high-addresses towards low-addresses. That means, if a push instruction is called, the value of the address held in the stack pointer sp register will decrease. **Once you know the location of the first argument, you'll know the location of other arguments, and it won't be hard to get their value.**

Remember that fixed string arguments, such as the format string for the printf function, are stored in the Data section of your program, which lies far from the stack in much lower memory addresses next to the Text section. So, what is pushed into the stack for these items is not their value, but is a pointer to the format string and all other like data objects contained in the parameter list.

You may have noticed that this argument passing convention somehow contradicts with the way how ARM machines do it. In the previous labs, you know that the arguments will be copied to registers r0-r3, and onto the stack if there are more arguments. However, this behavior makes it strange to get the address of an argument that lies in the register. So when the C compiler finds that a function want to get the address of its argument(s), the compiler will generate assembly code that puts those argument(s) onto the stack. Particularly, if the arguments are passed in the ellipsis way (...), they are all pushed onto the stack.

## Part 2: Placeholder in the format string

The format string may contain place holders introduced by the character ‘%’:

* %c is the place holder for a character in the arguments.
* %s holds a character string ending in \0.
* %x holds an integer printed in hexadecimal.
* %d holds an integer printed in decimal.
* %f holds the place for an floating number.

The standard printf() function supports more directives, and more complicated ones like %5.2f that prints a floating number, with at most 2 digits for the fraction part and pads with spaces at the left if the integral part has less than 5 digits.

For this lab you will implement these output formats and escape character:

* c, single character
* s, character string
* x, hexadecimal
* d, decimal
* %, if the character following the % directive is again a %, print one % only.

You can assume that no other illegal directives will occur. (This is actually the role of the compiler if you use standard printf())

Name your function myprintf() so that it doesn’t conflict with the standard printf(), and start your implementation like this:

int myprintf(const char \* format, ...) {

locate where the first argument are, and go through the format string, print every character by putchar(). Every time you encounter a %, find the next argument on the stack, and output that argument in the corresponding format. (Or, just output a %.)

## Part 3: Print the actual arguments

Among the directives to print, printing a character may be the simplest: find the corresponding character in the arguments and call putchar().

Printing a character string is also easy: find the corresponding argument, and call the putchar() function for every character. Keep in mind that the argument is a **pointer** to the string, and character strings would end with \0.

If the directive tells you to output the argument as a decimal or hexadecimal, it’s better to write a function to do it.

### Hexadecimal: again, operate the stack

Write a function printx in ARM assembly that takes an integer as the argument and outputs the integer in hexadecimal. Call printx from your myprintf() function to print numbers in hexadecimal. You may use putchar() function from C standard library. You **may not** use any global variables in printx.

Notice that the hexadecimal-represented number may have an indefinite number of digits, so we suggest that you start from the least significant bits of the integer. If you start from the MSBs, you may output unnecessary 0s, which is incorrect for this part of this lab. (Yes, standard printf() supports padding zeros at the left, but you’ll eventually encounter this problem again in the following part).

By some logical operation you can get the rightmost 4 bits of the integer. However, since you get these 4 bits first, it’s not the right time to output now, otherwise the least significant hexadecimal digit would be printed first. Therefore, it’s good idea to save this digit somewhere. Registers are not a good choice because you don’t have indefinitely many of them to match the indefinite number of digits to output, so the expandable stack in main memory is a good place. push and pop instructions will be extremely helpful, since the last digit pushed onto the stack will be the first one that gets printed.

Once you start to print the digits, you need to convert the digits to some characters before calling putchar(). If the digit is below 10, you should print it as a character from the set 0-9. If the digit is greater than 10, you should print it as a character from the set a-f. (It’s your choice to printing uppercase or lowercase letters, but you be consistent. You can also let printx take a second argument to specify which case to use, and in that case you can implement the %X formatting directive in addition to %x).

Keep in mind that putchar will interpret the passed argument in ASCII encoding. If you call putchar(9), the terminal won’t print a “9” character. Instead a tab character will be printed, according to the ASCII encoding.

### Decimal: divide by 10

Write a function printd in ARM assembly that takes an **signed** 32-bit integer as the argument and output the integer in decimal. You may use putchar() function from C standard library. You **may not** use any global variables in printd.

Printing an integer in decimal form is similar to that of hexadecimal, but there is another problem to handle. A signed 32-bit integer is stored in a register by 2’s complement, and there is no straightforward way to get the decimal representation by just looking at groups of the bits. Printing in hexadecimal is a lot easier since you can simply group the bits in nibbles (4-bits).

Inevitably you need to perform division operation on the integer. To get the digits 0-9, divide the integer by 10 and get the remainder, and repeat the division if the quotient is still larger than zero. Unfortunately there are no native instructions for integer division in ARMv7 that we use - the CPU just don’t have the ALU functionality for that. There are some pseudo-instructions given in the ARM manual, but they are actually a set of instructions that unpack the division operation as some loop.

Fortunately we have a trick to solve the problem faster. The divisor in this context is **always** 10 so there are some smart programmers who figured out instructions for the special case of dividing by a constant integer. There are many articles discussing about this topic if you Google, but a tested version is here: read the “Division by a constant integer” section of [this link](http://thinkingeek.com/2013/08/11/arm-assembler-raspberry-pi-chapter-15/) and the associated [tool](https://github.com/rofirrim/raspberry-pi-assembler/blob/master/chapter15/magic.py) to generate instructions to perform division. Please make sure that you understand how to use the code before plugging it in your work.

Now you know how to write the code. Here are some cases that you need to consider:

* What if the number to print is 0?
* What if the number to print is negative? Note that you need to see the number as a signed integer: if the MSB is 1, the number is negative.

## Turn in

1. Your submission should contain three files:
   * main.c, which calls myprintf() several times.
   * myprintf.c, which implements myprintf() function.
   * printinteger.s, which implements printx() and printd().
2. Put all three of these files in a folder named lab8-src.
3. Submit this folder electronically by following these instructions. (You have to be ssh’d into data.cs.purdue.edu first, the turnin command is only on those machines)

$ turnin -c cs250 -p lab8 lab8-src  
$ turnin -c cs250 -p lab8 -v

The second command verifies that you have successfully submitted your files.

GBA 2017.04.04