Most people take functions like printf() for granted, sometimes even thinking of them as "primitives" of the C language. But in an OS kernel, we have to implement all I/O ourselves.

Read through kern/printf.c, lib/printfmt.c, and kern/console.c, and make sure you understand their relationship. It will become clear in later labs why printfmt.c is located in the separate lib directory.

**Exercise 8.** We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

In lib/printfmc.c, vprintfmt function, bunch of cases for %\_\_. Case u (%u) will print the unsigned decimal number. So, unsigned octal code will be similar except we change the value of the base to 8 (octal).

case ‘u’:

num = getuint(&ap, lflag);

base = 10;

goto number;

case ‘o’:

num = getuint(&ap, lflag);

base = 8;

goto number;

Be able to answer the following questions:

1. Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

Console.c interacts with the console (keyboard/mouse/whatever input device) to take in input.

The sections for console.c are:

* Stupid I/O delay routine necessitated by historical PC design flaws
* Serial I/O code section
* Text-mode CGA/VGA display output
* Keyboard input code
  + Mappings, toggle/shift codes according to ascii values. (out of 256).
* General device-independent console code
  + Manage console input buffer
  + Stash characters received from the keyboard or serial port whenever the corresponding interrupt occurs
  + Initializes the console device
  + Outputs a character to the console
  + cputchar(int c) – ‘high’ – level console I/O. used by readline and cprintf. Calls static function cons\_putc, which calls: serial\_putc(c), lpt\_putc(c), and cga\_putc(c)

Console.c exports the cputchar function, which printf.c uses to output a charcter to console, iscons function, and getchar function.

1. Explain the following from console.c:
   * 1. if (crt\_pos >= CRT\_SIZE) {
     2. int i;
     3. memmove(crt\_buf, crt\_buf + CRT\_COLS, (CRT\_SIZE - CRT\_COLS) \* sizeof(uint16\_t));
     4. for (i = CRT\_SIZE - CRT\_COLS; i < CRT\_SIZE; i++)
     5. crt\_buf[i] = 0x0700 | ' ';
     6. crt\_pos -= CRT\_COLS;
     7. }

1. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

Trace the execution of the following code step-by-step:

int x = 1, y = 3, z = 4; cprintf("x %d, y %x, z %d\n", x, y, z);

* + In the call to cprintf(), to what does fmt point? To what does ap point?
  + List (in order of execution) each call to cons\_putc, va\_arg, and vcprintf. For cons\_putc, list its argument as well. For va\_arg, list what ap points to before and after the call. For vcprintf list the values of its two arguments.

1. Run the following code.

unsigned int i = 0x00646c72; cprintf("H%x Wo%s", 57616, &i);

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. [Here's an ASCII table](http://web.cs.mun.ca/~michael/c/ascii-table.html) that maps bytes to characters.

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

[Here's a description of little- and big-endian](http://www.webopedia.com/TERM/b/big_endian.html) and [a more whimsical description](http://www.networksorcery.com/enp/ien/ien137.txt).

1. In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

cprintf("x=%d y=%d", 3);

1. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret[ANSI escape sequences](http://rrbrandt.dee.ufcg.edu.br/en/docs/ansi/) embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on [the 6.828 reference page](https://pdos.csail.mit.edu/6.828/2017/reference.html) and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Notes/vocab:

The C library function **void \*memmove(void \*str1, const void \*str2, size\_t n)** copies **n** characters from **str2** to **str1**, but for overlapping memory blocks, memmove() is a safer approach than memcpy().