Making a Simple C kernel with Basic `printf` and `clearscreen` Functions

This tutorial has the purpose of showing a simple kernel. Let's start with the entry of the kernel which is in our example in a file called *kernel_start.asm*.

```
[BITS 32]
[global start]
[extern _k_main] ; this is in the c file

start:
  call _k_main

  cli  ; stop interrupts
  hlt ; halt the CPU
```

Okay, this code is 32-bit code(the `[BITS 32]` does that) and calls a function called k_main, which is defined in a C file called *kernel.c*. Now you're probably wondering why it's called `k_main` in the C file, but `_k_main` in the assembly file. This is because C/C++ compilers add an underscore(_) in front of all C/C++ functions... *Unless* you link this to an ELF file. ELF does not need the underscore. Once `k_main` is called, the instruction `cli` is executed. `cli` turns off interrupts(though they were never on in this example). Then, `hlt` is executed which tells the CPU to stop executing. We could do `jmp $` instead of `hlt`, but that eats up tons of CPU time and could cause the CPU to overheat. Note that interrupts can awake the CPU from a `hlt` instruction. That is why we disable interrupts before doing `hlt` as we want this to be a complete stop.

Now, let's move on to the definitions and function prototypes defined at the start of *kernel.c*.

```
#define WHITE_TXT 0x07 // white on black text

void k_clear_screen();
unsigned int k_printf(char *message, unsigned int line);
void update_cursor(int row, int col);
```

Nothing real special here except for the `#define WHITE_TXT 0x07`. We'll come back to that define in a little bit, for now, just remember that it's there.

Now, let's move on to the `k_main` function.

```
k_main() // like main in a normal C program
{
        k_clear_screen();
        k_printf("Hi!\nHow's this for a starter OS?", 0);
};
```

`k_main` is the entry point to our kernel. We call this function in the *kernel_start.asm* file.

`k_clear_screen` does what you would expect... clear the screen. `k_printf("Hi!\nHow's this for a starter OS?", 0);` prints the text:
Hi!
How's this for a starter OS?
Starting on the first line of video memory(0 is the first line, 1 is the second, 2 is the third, etc). The `\n` specifies a newline just like it would in the C/C++ `printf` function.

In protected mode, you can't call BIOS Interrupts to clear the screen, we have to do this ourselves by writing directly to the Video Memory.

```
void k_clear_screen() // clear the entire text screen
{
        char *vidmem = (char *) 0xb8000;
        unsigned int i=0;
        while(i < (80*25*2))
        {
                vidmem[i]=' ';
                i++;
                vidmem[i]=WHITE_TXT;
                i++;
        };
};
```

In the above function(`k_clear_screen`), the pointer *vidmem* points to 0xb8000 which is the start of video memory in protected mode. We declare the pointer as a `char` so we can write a byte at a time to the video memory. The text mode on an x86 is 80 x 25 charactors. Each charactor needs 2 bytes. The first byte is the charactor, the second byte is the attribute byte which controlls color and blinking. So, we take 80*25(the amount of charactors that can be displayed on the screen) and multiply it by 2 since we access video memory a byte at a time. The loop is pretty much self explanitory. The `vidmem[i]=' ';` writes a space to the video memory(*i* points to the exact spot). We add 1 to *i*, `i++;` to get to the next byte of video memory(the attribute byte) and put 0x07 there. 0x07 specifies a black background with white, non-blinking, text.

Now on to the `k_printf` function!

```
unsigned int k_printf(char *message, unsigned int line) // the message and then the line #
{
        char *vidmem = (char *) 0xb8000;
        unsigned int i=0;

        i=(line*80*2);

        while(*message!=0)
        {
                if(*message=='\n') // check for a new line
                {
                        line++;
                        i=(line*80*2);
                        *message++;
                } else {
                        vidmem[i]=*message;
                        *message++;
                        i++;
                        vidmem[i]=WHITE_TXT;
                        i++;
                };
        };

        return(1);
};
```

The `k_printf` function works much like the `k_clear_screen` function. `while(*message!=0)` loops until we reach the end of the string of text that is passed to the function. `if(*message=='\n')` checks to see if the next charactor of the string is for a newline.. If it is, we we add 1 to *line* so that the charactors that come after a *\n* will be down one more line. If a charactor is not a newline(*\n*), we just put the charactor into video memory and set the attribute byte to 0x07(black background with white non-blinking text).

Compiling the Kernel

First, download the kernel source. You will also need an assembler(NASM), C compiler(DJGPP or gcc), and a linker(LD).

Now, up near the top of the linker file, you will see this line:
```
.text 0x100000
```
The hex number needs to be set to where the kernel will be loaded into memory. In this case, that is at the 1MB mark(0x100000 in hex).

Let's compile our "boiler plate" assembly code file first:

```
nasm -f aout kernel_start.asm -o ks.o
```

This compiles *kernel_start.asm* to *ks.o* in aout format. Now for our C file:

```
gcc -c kernel.c -o kernel.o
```

The next and last step is to link *ks.o* and *kernel.o* into one file. In this case, we are going to link them together into a flat binary file with the linker script *link.ld*. We link the two files together with this command:

```
ld -T link.ld -o kernel.bin ks.o kernel.o
```

It is important that ks.o is linked first or the kernel will not work. The kernel is called *kernel.bin* and is ready to be run by a bootsector/loader that sets up Protected Mode and enables the A20(John Fine's bootf02 bootsector does this). If you would like to have GRUB be able to load this kernel, you can download the GRUB version here(you compile it and link it the same way).

Conclusion

There! A basic kernel. You probably will want to write a better `k_printf` function, as the one used in this example is rather simple and doesn't handle things like %s, %d, %c, etc. Still, this should be enough to get you on the track towards making a better one.

This tutorial was written by Joachim Nock and K.J.

Updated September 13, 2002 by K.J.