O.S. Lab 2: Interrupt-Driven I/O

In this project you will continue working with BIOS calls, tie them to interrupt 33 and write a new kernel to demonstrate them.

Step 1: Reading from the keyboard – Interrupt 22 (0x16)

The BIOS interrupt for reading a character from the keyboard is 22 (0x16). When called, AH must equal 0 (actually, it is okay if AX equals 0 since AL does not matter). The interrupt returns the ASCII code for the key pressed; i.e. **char* c = interrupt(22,0,0,0,0)** stores the key stroke in the variable *c*.

You will now write a kernel function *readString*. *readString* should take a character array with at least 80 elements but nothing in them. *readString* should call interrupt 22 repeatedly and save the results in successive elements of the character array until the ENTER key is pressed (ASCII 13 or 0xD). It should then add 0x0 (end of string) as the last character in the array and return.

Two other points:

- All characters typed should be printed to the screen (otherwise the user will not see what the user is typing). After reading a character, the character should be printed to the screen using interrupt 16.
- Your function should be able to handle the BACKSPACE key. When a backspace (ASCII 0x8) is pressed, it should print the backspace to the screen but not store it in the array. Instead it should decrease the array index. (Make sure the array index does not go below zero). Note that the character being overwritten is not erased on screen. All that happens is that the cursor is backed up so you can type over the character being replaced.

Write a test function in *main()* to prompt the user for a string, accept a string as input and print the string back out when you press ENTER.

Step 2. Integer I/O

So that we can ultimately create some more interesting test programs, let's add I/O for 16-bit unsigned integers to our system. We will need functions **void readInt(int*)** and **void writeInt(int)**. They are pretty much inverses of each other:

```
readInt(int* n)
{
  *n = 0; use readString() to get input;
  convert the input string to its numeric value n;
}
```

```
writeInt(int x)
{
  convert x to a string;
  use printString() to output;
}
```

(Use interrupt 33, options 1 and 0 instead of **readString()** and **printString()**, respectively, in the final version of your code.) The problem is now converting integers to and from ASCII strings. These are common questions that turn up in job interviews; see *http://www.programminginterview.com/content/strings*. The trick now is to rewrite those functions to work under the restrictions of 16-bit ANSI C. For one thing bcc does not support integer division and modulus as primitive functions, so we have to provide them ourselves:

```
int mod(int a, int b) {
  int x = a;
  while (x >= b) x = x - b;
  return x;
}
```

```
int div(int a, int b) {
  int q = 0;
  while (q * b <= a) q++;
  return (q - 1);
}
```

One thing that helps us is the fact that, since we are dealing exclusively with 16-bit integers, there are at most five digits (MAX_INT = 32767) to print with no sign. (We really should worry about overflow and underflow but won't for now.) We also have to treat the integer zero as a special case.

Adding Interrupt 33 Function Calls

As you did in the last lab, extend the interrupt handler to incorporate our new functionality:

Function	AX=	BX=	CX=	Action
Read string	1	Address of character array to store entered keys in		Call **readString(bx)**
Write integer	13	Integer to print	Destination: 0 = screen 1 = printer	Call **writeInt(bx,cx)**
Read integer	14	Address of integer variable to store entered number in		Call **readInt(bx)**

Revised Kernel: The MadLib Game

Mad libs is a word game where a player is prompted for a list of words which are then substituted for blanks in a story. The often comical or nonsensical story is then read aloud for the amusement of the participants. A *main()* kernel function that prompts you for inputs then places them into the context of a story is available for download from the web site. Cut-and-paste this new code into the proper place in your kernel, compile and run to test your new functions.

When you run your new kernel, ask yourself if it is behaving in a manner you would expect had you run it under Windows or Linux. If it doesn't pass this ease-of-use test you may have to redo your o.s. functions.

Conclusion

When finished, submit a .tar or .zip file (no .rar files) containing all needed files (**bootload**, **bdos.txt**, **kasm.o**, **compileOS.sh**, **kernel.c** and **README**) to the drop box. **README** should explain what you did and how the t.a. can verify it. Your .zip/.tar file name should be your name. Be sure to include your name under "Author" in the opening banner of the kernel.

Last updated 1.17.2018 by T. O'Neil, based on material by M. Black and G. Braught. Previous revisions 12.21.2016, 2.11.2016, 1.16.2016, 1.14.2015, 2.21.2014.