Michael Black, 3/9/2009

CSC 565 - Operating Systems
Spring, 2008

Project B - System Calls

due: Tuesday, February 26

Objective

One of the major services an operating system provides are system calls. In this project you will learn how to use some of the system calls provided by the BIOS. You will then write your own system calls to print a string to the video, read in a line from the keyboard, and read a sector from the disk. This will create the foundation needed for the next project.

What you will need

You will need the same utilities you used in the last project, and you will also need to have completed the previous project successfully. Additionally, you will need to download the *kernel.asm* for this project.

Step 1: Printing to the Screen - Interrupt 0x10

In the previous project you saw how to print to the screen by directly writing characters to the video memory. The problem with this is that you have to keep track of the cursor position yourself, as well as scrolling when you reach the end of the page. Alternatively, the BIOS provides a software interrupt that will take care of printing to the screen for you. Interrupt 0x10 calls the BIOS to perform a variety of I/O functions. If you call interrupt 0x10 with 0xE in the AH register, the ASCII character in the AL register is printed to the screen at the current cursor location.

Since interrupts may only be called in assembly language, you will have to implement a function called *printChar* in assembly language, then you'll use this function to implement your *printString* function in C.
To use interrupt 10 to print out the letter 'Q', you will need to do the following:

1. Figure out the parameters. To print out 'Q', AH must equal 0xE and AL must equal 'Q' (0x51)
2. Call the interrupt 0x10.

Example:

```
mov al, 'Q'
mov ah, 0x0e
int 0x10
```

You can find the register parameters for the various BIOS interrupts online. A good resource is *http://www.ctyme.com/intr/int.htm.*

Your task

To complete step 1, you need to write two functions: *void printChar(char ch)* in assembly language and *void printString(char*)* in C.

You should test your function by calling:

printString("Hello World\0");

Notes:

Remember to add the *start.asm* source file, so you can use functions in the C code without any problem.

Step 2: Reading from the keyboard - Interrupt 0x16

The BIOS interrupt for reading a character from the keyboard is 0x16. When called, AH must equal 0. The interrupt returns the ASCII code for the key pressed in the AL register.

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

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 the function *printChar*.

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.

If your function works, you should be able to call in main():

```
char line[80];
printString("Enter a line: \0");
readString(line);
printString(line);
```

When you run this in Bochs, it should prompt you to enter a line. When you press ENTER, it should echo what you typed back to you on the next line.

Step 3 - Read a sector from the disk - Interrupt 0x13

Interrupt 0x13 can be used to read or write sectors from the floppy disk. Reading sectors takes the following parameters:

AH = 2 (this number tells the BIOS to read a sector as opposed to write)
AL = number of sectors to read (use 1)
BX = address where the data should be stored to (pass your char* array here)
CH = track number
CL = relative sector number
DH = head number
DL = device number (for the floppy disk, use 0)

This interrupt requires you to know the cylinder, head, and track number of the sector you want to read. In this project we will be dealing with absolute sector numbers. Fortunately, there is a conversion.

For floppy disks:

relative sector = (sector MOD 18) + 1
head = (sector / 18) MOD 2 (this is integer division, so the result should be rounded down)
track = (sector / 36)

Your task

Your task is to write a function *readSector(char* buffer, int sector)* which takes two parameters: a predefined character array of 512 bytes or bigger, and a sector number to read. Your function should compute the relative sector, head, and track, and call interrupt 0x13 to read the sector into buffer. This function should be implemented completely in assembly language.

Testing this

To test this, you should read in a sector containing ASCII text and print it out using printString.

1. Add the following to *main()*:
   ```
   char buffer[512];
   readSector(buffer, 30);
   printString(buffer);
   ```

2. Create a text file called *message.txt* and something to it.
3. After you compile your floppya.img, type the following to put message.txt at sector 30:
   dd if=message.txt of=floppya.img bs=512 count=1 seek=30 conv=notrunc

Run Bochs. If the message in *message.txt* prints out, your readSector function works.

Step 4 - Create your own interrupt and make printString, readString, and readSector interrupt calls

An operating system should provide services to user programs by creating its own interrupts. You will now create an interrupt 0x21 handler. When an interrupt 0x21 is called, it should run your own code.

Creating an interrupt service routine is simply a matter of creating a function, and putting the address of that function in the correct entry of the interrupt vector table. The interrupt vector table sits at the absolute bottom of memory and contains a 4 byte address for each interrupt number. To add a service routine for interrupt 0x21, write a function to be called on interrupt 0x21, and then put the address of that function at 0x00084 (21*4) in memory.

Unfortunately, this really has to be done in assembly code. You are consequently provided, in kernel.asm, with two functions. *makeInterrupt21()* simply sets up the interrupt 0x21 service routine. Function *interrupt21ServiceRoutine()* is henceforth automatically called whenever an interrupt 0x21 happens.

In this step, you should have your interrupt 0x21 handler provide *printString*, *readString* and *readSector* services. Your interrupt 0x21 will be defined as follows:

AX = a number that determines which function to run

print string:
AX = 0
BX = address of the string to print

read string:
AX = 1
BX = address of the character array where the keys entered will go

read sector:
AX = 2
BX = address of the character array where the sector will go
CX = the sector number

if AX = 3 or more, print an error message

Your task is to write the function *interrupt21ServiceRoutine* that reads the value in AX and calls one of the three functions you just wrote.

Step 5 Create a simple API

To have access to the system calls you just implemented in step 4 we have to use assembly language, apart from that we have to remember the number assigned to every system call. So in practice it's easier to have an API the make the respective system call.
So in this step you'll create an API with the following functions:

*syscall_printString(char *str)*
*syscall_printString(char *str)*
*syscall_readSector(char *buffer, int sector)*

Your Task

Your task is to create a new file called *os_api.asm***,** this file will contain the implementation of your API as described before.

Testing

You should test your work by using the implemented API and seeing that they work correctly.

In main(), try the following:

```
char line[80];
syscall_readString(line);
syscall_printString(line);
```

If your program works, it should wait for you to read in a line. Then it should echo it back to you on the next line.

Unlike the printString and readString functions, which can only be called from within your kernel.c program, these API routines can be called from other programs that do not have these functions.