**German University in Cairo**
**Faculty of Media Engineering and Technology**
**Dr. Aysha Alsafty**
**Eng. Nourhan Ehab**
**Eng. Eslam Osama**

# **CSEN 602-Operating Systems**, Spring 2017
# **Course Project - Milestone 1: System Calls**
Due on Saturday 25/2/2017 by 11:59 pm

## Milestone Objective
One of the major services an operating system provides are system calls. In this milestone 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 screen, read in a line from the keyboard, and read a sector from the disk. This will create the foundation needed for the next milestone.

## Before you start
You will need the same utilities you used in the last milestone, and you will also need to have completed the previous milestone successfully.

## Step 1: Printing to the Screen by using Interrupt 0x10
Recall the BIOS interrupt `0x10` you used in Milestone 0 used to print out a single character to the screen. The objective of this step is to extend the code you wrote in Milestone 0 to print any arbitrary input string to the screen utilizing interrupt `0x10`.

## Task 1
To complete step 1, you need to write a void `printString(char*)` function. Your `printString` takes a character array (the equivalent of the Java String) as a parameter. The last character in the array should be the terminating character `'\0'`. Your function should print out each character of the array until it reaches `'\0'`, at which point it should stop. You should test your function by calling: `printString("Hello World\0");`

**Important Note:**
When adding functions to your `C` program, make sure they always follow `main()`. `main()` must always be your first function. When adding a function, you will need to declare it at the top. A declaration is the function definition, minus parameter names, followed by a semicolon. For example, if your function is `void printString(char* chars)`, you will need to write at the top of your program: `void printString(char*);`

## Step 2: Reading from the keyboard by using Interrupt 0x16

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

### Task 2

You should write a function `readString` that takes a character array with at least 80 elements. `readString` should call interrupt `0x16` repeatedly and save the results in successive elements of the character array until the `ENTER` key is pressed (ASCII `0xd`). Once `ENTER` is pressed, it should then add a `0xa` (line feed) and `0x0` (end of string) as the last two characters 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 in a character, the character should **immediately** be printed to the screen using interrupt `0x10`. 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 (and update the screen so that the last character does not appear anymore) but not store it in the array. It should, however, decrease the array index. (Make sure the array index does not go below zero).
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: Reading a sector from the disk by using 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)

Note that `AX=AH*256+AL`, `CX=CH*256+CL`, and `DX=DH*256+DL`.

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 a floppy disk:

```
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 )
```

## Task 3

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. Unfortunately, `bcc` does not support `MOD` and `DIV`. You will need to write your own `MOD` and `DIV` functions.

To test your work, you should read in a sector containing ASCII text and print it out using `printString`. In order to do this:

1. Add the following to `main()`:

   ```
   char buffer[512];
   readSector(buffer, 30);
   printString(buffer);
   ```

2. Inside `M1.zip`, you will find a text file `message.txt`. After you compile your `C` file and copy the resulting machine code into `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
   ```

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

## Step 4: Creating your own Interrupt

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. It calls a function in your `C` code `handleInterrupt21(int ax, int bx, int cx, int dx)` that you will need to write. The `AX`, `BX`, `CX`, `DX` parameters passed in the interrupt call will show up in your `handleInterrupt21`function as parameters `ax`, `bx`, `cx`, `dx` respectively.

### Task 4
Your task in this step is fairly simple. First go to `kernel.asm` and uncomment by removing the semicolons before the line `.extern _handleInterrupt21`, and all the lines under `_interrupt21ServiceRoutine:`. In your `C` program, create the function `void handleInterrupt21(int ax, int bx, int cx, int dx)`. In it, use `printString` to print out a message (like "Hello world"). In `main()`, add the call `makeInterrupt21()`. Then, underneath, call interrupt `0x21`:
`interrupt(0x21,0,0,0,0)`. Compile and run your program. If it works, Bochs will print out your message.

## Step 5: Make printString, readString, and readSector Interrupt Calls
In this final step, you should have your interrupt `0x21` handler provide `printString`, `readString` and `readSector` services. Your interrupt `0x21` will be defined as follows:

- The parameter `AX` will determine which function to run.
- If `AX=0`, then `printString` will be called. `BX` should contain the address of the string to print.
- If `AX=1`, then `readString` will be called. `BX` should contain the address of the character array where the pressed keys ASCII codes are stored.
- If `AX=2`, then `readSector` will be called. `BX` should contain the address of the character array where the contents read from the sector will be stored. `CX` should contain the sector number.
- If `AX>=3`, print an error message.

### Task 5
Your task is to write the function `handleInterrupt21` that reads the value in `AX` and calls one of the three functions you just wrote. You should test your work by

making interrupt `0x21` calls and seeing that they work correctly. In `main()`, try the following:

```
char line[80];
makeInterrupt21();
interrupt(0x21,1,line,0,0);
interrupt(0x21,0,line,0,0);
```

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. Note that, unlike the `printString` and `readString` functions which can only be called from within your `kernel.c` program, these interrupt `0x21` routines can be called from other programs that do not have these functions.

**Project Deliverables and Submission**
For this milestone you are required to submit ONE zip containing all of your files. You should use this webform `https://podio.com/webforms/17714218/1190687` to submit your project. Late submissions will not be accepted.
Have fun :)