CSC 565 - Operating Systems
Spring, 2008

Project B - System Calls

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 are provided with a function *interrupt* that makes an interrupt happen. The interrupt function takes five parameters: the interrupt number, and the interrupt parameters passed in the AX, BX, CX, and DX registers, respectively. It returns the value returned from the interrupt in the AL register.

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. Calculate the value of AX. AX is always AH*256 + AL.
3. Call the interrupt routine. Since registers BX, CX, and DX are not used, pass 0 for those parameters.

Example:

```
char al = 'Q';
char ah = 0xe;
int ax = ah * 256 + al;
interrupt(0x10, ax, 0, 0, 0);
```

Or you could simply write:

```
interrupt(0x10, 0xe*256+'Q', 0, 0, 0);
```

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 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 unprintable character 0x0. Your function should print out each character of the array until it reaches 0x0, at which point it should stop.

You should test your function by calling:

printString("Hello World\0");

Notes:

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 - Interrupt 0x16

The BIOS interrupt for reading a character from the keyboard is 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.

You should write a function *readString*. readString should take a character array with at least 80 elements but nothing in them. *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). 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 a character, the character should 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 but not store it in the array. Instead it should 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 - 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)

Note that CH and CL, and DH and DL, can be combined to CX and DX by applying the formulas: 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 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.

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

For mod (a, b), use the pseudocode:

```
while a > = b
        a = a - b
--> return a
```

For div (a, b), use the pseudocode:

```
let quotient = 0
while quotient * b  <  a
        q = q + 1
--> return q
```

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. Download from Blackboard the file *message.txt*
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

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.

Your task

Your task in this step is fairly simple.

First go to *kernel.asm* and uncomment (remove the semicolons) the line ".extern _handleInterrupt21" and all the lines under _handleInterrupt21:.

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:

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 *handleInterrupt21* that reads the value in AX and calls one of the three functions you just wrote.

Testing

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.

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.

Submission

You should submit a .zip or .tar file (no .rar files please) containing all your files and a shell script for compiling on Blackboard Digital Dropbox. Be sure that all files have your name in comments at the top. Your .tar/.zip file name should be your name. You must include a README file that explains 1) what you did, and 2) how to verify it.