# Operating Systems

Lab Class 1 - Exercises

Low-level programming

This lab will show you how it feels to program without the abstractions provided by an operating system (and a high-level programming language).

## 1 Getting Started

The command prompt is our cockpit. You can open the command prompt of Windows by clicking on the search (magnifying glass) icon next to the start button and typing `cmd`; then select *Command prompt*.
The command prompt shows your *current directory* followed by `>`. It executes the commands that you type after pressing the *Enter* key.
The most important commands are:

- `dir` shows the files in the current directory
- `N:` changes to the current drive to `N`
- `cd some_directory` changes the current directory to `some_directory`
- `cd ..` changes to the parent directory (denoted by `..`)
- `notepad++ some_file` opens the file `some_file` in a text editor
- Pressing the → | (tabulator) key suggests a completion of the command while you are typing; you can avoid fully typing directory and filenames that way.
- Use double quotes to delimit directory or filenames that contain spaces.
- Pressing the ↑ key allows you to select commands that you have typed previously, modify them and execute them again.

If you are unfamiliar with this environment please try these commands.
We will use **QEMU**, a virtual machine that simulates various CPUs and hardware devices. This is a great tool for learning about operating systems because we can do whatever we want in software without risking damage to the hardware (if it breaks, we can just reset the virtual machine). That said, you could run the code that you are going to write in this lab on a 'real' PC, provided that it is well-tested—a stray bug in your code could damage your computer, e.g. corrupt your hard disk.

1. We can get QEMU to simulate a PC with an x86-compatible CPU. Open Software Hub on your Lab PC and find QEMU. Once found, start the application - a console window will appear. You will see that QEMU runs the BIOS program. This program in turn tries to run a *bootloader*, but cannot find one. The first task below will solve this problem and implement a boadloader program.

The BIOS offers us a number of basic functions like checking hardware facilities, reading keyboard input and displaying text. Operating system usually implement their own drivers for these functions, but for us they are useful in this exercise as they allow us to access and communicate with the hardware more easily.

2. Download the `os-lab1.zip` file from Study Direct and unzip the content into your `N:` drive. There should be a directory `os-lab1` after unzipping. In the command prompt change the directory into that folder:

```
N:
cd os-lab1
```

## 2 Bootloader

3. Open the file `boot1.asm` in a text editor program of your choice, e.g.

```
notepad boot1.asm
```

The file contains the following assembly program:

```
org 0x7c00

_start:
        ; move 0x0e into the ah register
        ; numbers starting with 0x are hexadecimal
        ; this is a parameter for the interrupt service routine that
        ; tells it we want to print a character
        mov ah, 0x0e
        ; move 'a' into the al register
        ; this is the character we want to print
        mov al, 'a'
        ; call the BIOS routine to print the character by
        ; sending interrupt number 0x10
        int 0x10
        ; jump to the current memory location
        ; i.e. loop forever
        jmp $

; fill the rest of our program with zeros,
; a boot loader needs to be 512 bytes long.
times 510-($-$$) db 0

; this identifies the program as a bootloader
dw 0xaa55
```

This program simply writes the character `'a'` on the screen. Read the comments in the program carefully to understand how it works.

4. Then compile the program by running:

```
"C:\Program Files\NASM\nasm.exe" boot1.asm -f bin -o boot.bin
```

The **NASM** compiler translates the assembler program into the machine-executable binary file `boot.bin`.
(You may have to start NASM on Software Hub as well before you can do that.)

5. We can now run the program as bootloader in the virtual machine:

```
"C:\Program Files\QEmu\qemu-system-i386.exe" boot.bin
```

6. Modify the program to print a character other than `'a'` and run it again.

7. Next we will use the stack to print a message on the screen. To this end, open the file `boot2.asm` in the editor. It looks as follows:

```
org 0x7c00

_start:

        ; move 0x0e into the ah register
        ; numbers starting with 0x are hexadecimal
        ; this is a parameter for the interrupt service routine that
        ; tells it we want to print a character
        mov ah, 0x0e

_marker:
        ;push the character 'a' into the stack
        push 'a'

        ;pop the top of the stack into the ax register
        pop ax
        ;print the character
        mov ah, 0x0e
        int 0x10

        ; jump to the current memory location
        ; i.e. loop forever
        jmp $

;fill the rest of our program with zeros
times 510-($-$$) db 0

;this identifies the program as a bootloader
dw 0xaa55
```

The program in its current form prints just the character `'a'`. Modify it so that you first push the characters of a short message onto the stack (by calling `push` for each character) and then popping the values from the stack and calling the BIOS print routine. Compile and run the program to test whether it works correctly.

8. We can program loops with the help of `jmp`. Modify the line

```
jmp $
```

to say

```
jmp _marker
```

Compile and run the program. It will now print your message in an infinite loop.

9. Next we will define a constant that holds the message we want to print and use the more high-level printing routine provided by the file `printing.asm` to display it on the screen. Open the program `boot3.asm` in your editor:

```
org 0x7c00
_start:
        ;load the start address of msg into the source index register
        mov si,msg
        ; call a function to print it.
        ; This is a bit like calling a function in Java,
        ;except that the function parameters are set up before the call
        call printstring

        jmp $


%include "printing.asm"

;this defines the message we want to print.
;13 and 10 are the codes for a new line at the end of the string
;0 marks the end of the string
msg         db "MyNewOS 0.1",10,13,0

times 510-($-$$) db 0

dw 0xaa55
```

Read carefully the comments in this program to understand what the code is doing. Compile the program and run it.
If you are interested how this printing routine works then please have a look at the file `printing.asm` to work out how it does it.

10. Modify `boot3.asm` to print a different message. Compile it and run it.

## 3 Hardware check

When the OS starts up it will need to gather information about the hardware it is running on. One of the first things it might do is to check how much memory is available. At the moment, we are running in 16-bit mode, which means that we can only access a small amount of the machine's total memory. To see how much we have available at this point, we can call another BIOS routine, which will store the answer (in kb) in a register.

11. Open the file `boot4.asm` in your editor:

```
org 0x7c00
_start:
        ;load the start address of msg into the source index register
        mov si,msg
        ;call a function to print it.
        ;This is a bit like calling a function in Java,
```

```
        ;except that that function parameters are set up before the call
        call printstring

        ;clear ax
        xor ax,ax
        ;check the memory size
        int 0x12
        ;show a message if there was an error
        jc error
        ;did the interrupt return a zero?
        test ax,ax
        jz error
        mov word [hex16], ax ;look at register
        call printhex16
        call newline
        jmp no_error
error:
        mov si, errormsg
        call printstring
no_error:
        jmp $


%include "printing.asm"

;this defines the message we want to print.
;13 and 10 are the codes for a new line at the end of the string
;0 marks the end of the string
msg             db "MyNewOS 0.1",10,13,0
errormsg        db "Error",10,13,0

times 510-($-$$) db 0

dw 0xaa55
```

Read the code carefully to understand how it is doing it.

12. Compile and run the code. Look at the value that the program outputs and convert it to decimal. It should be $< 640$kb.

## 4 User input

13. You can read a character from the keyboard by loading the value `0` into the `ah` register and calling routine `0x16`. It returns the character in the `al` register. Write a program that reads a character from the keyboard and print it on the screen.

14. Now extend your code to read and print out characters in an infinite loop.

15. Modify the code to print out the hex value of each character.

## 5 Reflections

Well done! You have now written some software in one of the most difficult programming environments and with one of the most difficult languages. If you were going to extend your small program into a basic operating system, consider the following:
16. How would you accept commands from the user?
17. How would you run a user program?
18. How would you manage access to resources for a user program?

## 6 Graphics

BIOS routine `0x10` allows us to control the video mode.
19. Set `al` to `0x13` and `ah` to `0x00`. Then call `int 0x10`. This switches to 300x200 graphics mode. To set a pixel, call `int 0x10` with `ah` set to `0x0C`, `al` to a colour value (see `http://en.wikipedia.org/wiki/BIOS_color_attributes`). `cx` must contain the column and `dx` the row of the pixel.
20. Now draw an image on the screen:

- Entire screen in blue
- French flag
- Chessboard
- Gradient from lightgreen to darkgreen
- ...

You will find the following instructions useful to generate these (and other) images: `jcxz`, `dec`, `jcc`, `inc`, `sub`, `add`, `div`, `mul`. See `http://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086.2F8088_instructions` to understand how you can use them.

For more video functions see `http://en.wikipedia.org/wiki/INT_10H`.

# Appendix

## General Registers (AX, BX, CX, and DX)

- `AX` (Accumulator): This is the accumulator register. It gets used in arithmetic, logic and data transfer instructions. In manipulation and division, one of the numbers involved must be in `AX` or `AL`. `AX` means we are referring to the extended 16bit `A` register. Its upper and lower byte are separately accessible as `AH` (A high byte) and `AL` (A low byte). All general-purpose registers can be accessed as one 16bit register or as two 8bit registers. The two registers `AH` and `AL` are part of the big whole `AX`. Any change in `AH` or `AL` is reflected in `AX` as well. `AX` is a composite or extended register formed by gluing together the two parts `AH` and `AL`.

  `AX`: [ `AH` , `AL` ]

- BX (Base Register), CX (Count register), DX (Data Register), SP (Stack Pointer), BP (Base Pointer), SI (Source Index), DI (Destination Index) , `ALU` (Arithmetic and Logic Unit).

## BIOS Functions

BIOS (Basic Input/Output System) was created to offer generalised low-level services to early PC system programmers. The BIOS functions are organised by interrupt number:

- INT 0x10 = Video display functions
- INT 0x16 = keyboard functions
- INT 0x13 = mass storage (disk, floppy) access
- INT 0x15 = memory size functions

## Interrupt Jump Table

See `http://www.ctyme.com/intr/int.htm`
Examples:

- *VIDEO - TELETYPE OUTPUT* **(Int 10 / `AH`=0Eh)**
  Inputs:
  `AH` = 0Eh (i.e. 0x0e)
  `AL` = character to write
  `BH` = page number
  `BL` = foreground colour (graphics modes only)
  Returns: Nothing
  Description: Display a character on the screen

- *KEYBOARD - GET KEYSTROKE* **(Int 16 / `AH`=00h)**
  `AH` = 00h
  Returns:
  `AH` = BIOS scan code
  `AL` = ASCII character

## Getting Started + Bootloader Record

`http://goo.gl/5D6JMm`