kaneton K0
Bootstrap

Matthieu Bucchianeri and Renaud Voltz

January 7, 2008

Delivery date:	Sunday, 28th 11:42 pm
Assistants :	Matthieu Bucchianeri - chichelover@epita.fr Renaud Voltz - voltz_r@epita.fr
Dedicated googlegroup:	kaneton-students
Programming languages:	Assembly, C
Architecture:	Intel 32-bit
Students per group:	1
Tarball:	k0-login_l.tar.bz2
Permissions:	we don't care

Before starting

- You will use NASM to assemble your code. NASM is present on the PIE. To test, you will use QEMU, which can be run using the script ~bucchi_m/qemu/run.sh and adding the correct parameters (`-fda` for a floppy image).
- A bootsector is not an ELF binary, but a flat object (without any headers). Obtaining flat binaries with NASM is done using the `-f` flag (refer to the manual).
- The bootsector is loaded at address 0x7c00, you must find a way to tell NASM that the code will be loaded there.
- Remember that the microprocessor starts in 16-bit mode, so you must find a directive to tell NASM to assemble 16-bit code. Then, when you switch to 32-bit mode, find another directive to tell NASM the new assembly mode.
- Your bootsector must end with a signature (0xAA55). This means that blank characters must be inserted until byte 510, and then these two bytes must be present.

```
times 510-($-$$) db 0   ; fill the rest of the sector with zeros
dw    0xAA55            ; add the bootloader signature to the end
```

Implementation

Exercise 1: string display

- **Source tree**
  directory: /k0-login_l/ex1/
  filename: ex1.s

- **Subject**
  Print a string at the (20, 10) coordinates. You must use the BIOS calls.

- **Steps**

  1. `print_char`
     Print a character at the current cursor position, and update the cursor position.
  2. `print_string`
     Print the string pointed by *%si* register at the cursor position and update the cursor position.
  3. `cursor_set`
     Set the cursor position.

Exercise 2: libc

- **Source tree**
  directory: /k0-login_l/ex2/
  filename: ex2.s

- **Subject**
  Write a program wich dumps the registers values.

- **Steps**

  1. `malloc`
     Very stupid malloc:
     – Declare the heap.
     – Declare a break value at the begining of the heap.
     – malloc returns the break value in *%ax* and then increments it.
  2. `itoa`
     Basic itoa (hey, why not using it to test your malloc ?!).
  3. `itoa_hex`
     Hexadecimal itoa.
     16-bit hexadecimal outputs must match the following format: `0x00a2`.

- **Output**

```
ax = 0x1234 = 4660
bx = 0x0000 = 0
cx = 0xabcd = 43981
dx = 0x00ff = 255

bp = 0x1000 = 4096
sp = 0x0ff8 = 4088
ip = 0x7c00 = 31744
```

Exercise 3: keyboard inputs

- **Source tree**
  directory: /k0-login_l/ex3/
  filename: ex3.s

- **Subject**
  Write a prompt which gets a string from the keyboard and wich displays it when you press `ENTER`.

  You must display alpha-numeric characters and punctuation.
  Do not implement key combinations (using modifiers like `SHIFT`, `ALT`, `CTRL`, ...).
  Pressing `ENTER` must result in a newline.

- **Steps**
  1. `kbd_get_scancode`
     Get the next scancode from the keyboard buffer.
  2. `scancode_to_ascii`
     Convert a scancode to an ASCII character.

- **Output**

```
Enter your name: Renaud
Hello Renaud !
```

Exercise 4: floppy drive

- **Source tree**
  directory: /k0-login_l/ex4/
  filename: ex4.s

- **Subject**
  Write a program which loads the bootsector of a floppy disk and which checks wether it does contain a bootloader.

  (The bootsector contains a bootloader if it is ended by the 0xAA55 magic.)

  Your program must print the magic value as shown in the given output.

- **Steps**
  1. `floppy_read_sector`
     Read n sectors from the floppy drive (A:).

- **Output**

```
Loading floppy bootsector ... OK
magic found: 0xaa55

Loading floppy bootsector ... OK
ERROR: bad magic: 0xt824
```

Exercise 5: operating modes switching

- **Source tree**
  directory: /k0-login_l/ex5/
  filename: ex5.s

- **Subject**
  Write a program which turns the microprocessor into protected mode.
  Once in protected mode, your program must clear the whole screen and print a message indicating that protected mode is enabled.

- **Steps**

  1. `pmode_enable`
     Switch from real mode to protected mode.
  2. `print_string_fb`
     Print a string (in protected mode). See appendix *VGA text framebuffer*.
  3. `memset`
     Basic memset function.
  4. `memcpy`
     Basic memcpy function.

Exercise 6: ELF loader

- **Source tree**

directory:	/k0-login_l/ex6/
filenames:	ex6.lds
	ex6.s

- **Subject**

  You will now write a complete bootloader. This one will load an ELF file from the disk (located at the sector just after the bootsector) and then relocate it in memory at the right place, before jumping to it.

  The ELF file **must** contain two segments, one with the code (which must be loaded at 1 Mb) and the other with the data (loaded at 2 Mb). Example:

```
42sh> readelf -l bootloader

Elf file type is EXEC (Executable file)
Entry point 0x1000cc
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00100000 0x00100000 0x000df 0x000df R E 0x1000
  LOAD           0x002000 0x00200000 0x00200000 0x00022 0x00028 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data .rodata .bss
```

  Your bootloader **must** reset the BSS memory. To keep the thing simple, put the *.bss* section at the end of the second segment. Its size can be determined by computing the difference between MemSize and FileSiz.

  Your tarball **must** include the ld-script used to create such ELF binaries.

  Before starting, you should watch the ELF documentation, especially about the ELF header and the Program Header. The task is not as harder as it looks like (our code dealing with ELF files is about 30 instructions long).

- **Steps**

  1. Get the `floppy_read_sector` function and call it correctly to store the binary in a temporary location.
  2. Reuse your `pmode_enable` function.
  3. Read the loaded binary to find its segments and extract their load addresses, size and source location into the file.
  4. Use your `memcpy` to relocate the code and the initialized data, use your `memset` to reset the BSS section.
  5. Find the binary entry point (2 instructions !) and jump on it after having initialized a correct stack).

  You must write your own test ELF binary in C, which must be able to write text on the screen. See appendix *VGA text framebuffer*.