Ministerul Educației, Culturii și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

Raport
pentru lucrare de laborator Nr. 4
la cursul Sisteme de Operare
"Boot manager"

A efectuat: Arteom KALAMAGHIN, FAF-211
A verificat: Rostislav CĂLIN

Chișinău - 2023

Subject: Floppy disk I/O operations

Tasks:

Creați în limbajul assembler o aplicație care va avea rolul de Bootдoader și va realiza următoarele:

1. Va afișa un mesaj de salut ce va include numele autorului și va aștepta introducerea de la tastatură a adresei ”sursă” de pe floppy, de unde să citească nucleul (sau alt cod compilat ce se dorește de a fi încărcat și executat). Adresa va fi introdusă în formatul SIDE, TRACK, SECTOR și adresa trebuie să fie strict în diapazonul rezervat studentului autor precum a fost în Lab3.

2. Va aștepta citirea de la tastatură a adresei ”destinație” a memoriei RAM unde să fie încărcat blocul de date citit de pe floppy. Adresa de RAM trebuie să fie în formatul XXXXh:XXXXh, identic cum a fost pentru Lab3.

3. Va transfera datele FLOPPY ==> RAM și va afișa codul de eroare cu care s-a finalizat operația dată.

4. Va afișa un mesaj pentru a tasta o tastă și a lansa nucleul (sau pentru a executa codul ce se dorește a fi executat).

5. După finalizarea execuției nucleului sau a codului executat, va afișa un mesaj pentru a tasta o tastă și a executa repetat Bootloader-ul!

Implementation:

Since the bootloader ended up larger than 512 bytes I needed a first stage bootloader to run the main boot manager.

```
org 7c00h

mov     ah, 00
int     13h

mov     ax, 0000h
mov     es, ax
mov     bx, 7e00h

mov     ah, 02h
mov     al, 2
mov     ch, 0
mov     cl, 2
mov     dh, 0
mov     dl, 0
int     13h

jmp     0000h:7e00h

times 510-($-$$) db 0
dw 0AA55h
```

The first part of the second stage bootloader did not cause any troubles since we've already extensively practiced disk I/O operations: read the addresses and call the known BIOS interruption.

```
start:
   call    reset_memory
   xor     sp, sp

   call    read_hts_address

   cmp     byte [operation_flag], 0
   je      error

   ... Read, check and cast N val ...

   call    read_ram_address

   cmp     byte [operation_flag], 0
   je      error

   mov     es, [address + 0]
   mov     bx, [address + 2]

   mov     al, [nhts + 0]
   mov     dl, 0
   mov     dh, [nhts + 2]
   mov     ch, [nhts + 4]
   mov     cl, [nhts + 6]

   mov     ah, 02h
   int     13h

   ...
```

Then I was tasked with implementing a check to ensure that the kernel was read properly. Essentially we need to check the first and the last few bytes of the piece of data read from the floppy and if we shall wind there some specific values, we may conclude that the kernel was read properly. These values are C7 06 for the first two bytes of our binary and 55 53 for the last two bytes of the “AMOGUS” signature I left at the end of the last sector with the code of the kernel. If we do not find these values, we conclude that user introduced incorrect N / HTS values, display the error and jump back at the start of the second stage boot manager.

```
...

mov     ax, [es:bx]
cmp     ax, 0x06C7
jne     wrong_in_error

mov     ax, [nhts]
imul    ax, 512

push    bx
add     bx, ax
sub     bx, 2
mov     ax, [es:bx]
pop     bx

cmp     ax, 0x5355
jne     wrong_in_error

call    wait_for_keypress
call    clear_screen

mov     ds, [address + 0]
mov     si, address + 2

mov     ax, [address + 0]
mov     bx, [address + 2]

jmp     [si]
```

Then I’ve started to work on the kernel, the plans were to implement the ability to enter, recognize and interpret several basic commands as well as a specific more advanced command I was tasked in class with :

- *about* - display a short message describing the software;
- *clear* - erase everything previously entered from the screen and return the cursor to the top of the page;
- *datetime* - display the current date and time using CMOS RTC data;
- *exit* - exit the program and return to the boot manager

To have closure on the bootloader, I will describe how the problem with label addressing was solved. Since we don’t know preliminarily at which address user will request the kernel to be loaded we cannot explicitly define the origin as we did before using the *org* operation. This can be solved by passing the OFFSET value from the second stage bootloader to the kernel via some register and then “advancing” the base pointer to the data segment value. It seems like not all the labels require this - declared strings definitely do. You may see how it is done on the next page…

```
...

mov     word [kernel_origin + 0], 0000h
mov     word [kernel_origin + 2], 0000h

mov     word [kernel_origin + 0], ax
mov     word [kernel_origin + 2], bx

...

print_str:
   push    cx

   call    get_cursor_pos

   xor     ax, ax
   mov     es, ax
   mov     bp, si

   mov     bl, 07h
   pop     cx

   mov     ax, 1301h
   int     10h

   ret
```

Now I can finally tell you how the CLI works. Nothing hard, first of all we read a token and go through all the known commands it may correspond to (if we went through the entire list we notify the user that this is an unknown command and restart the cycle). We check if the length is the same and then char by char check if the token inserted is exactly the same word as the command's name. If it is the case we return the identifier of the operation to interpret that is recognized by the interpreter and a corresponding piece of code is run.

```
start:
   call    break_line_for_input
   call    read_input

   cmp     byte [input_buffer], 00h
   je      cli_cycle_end

   call    break_line

   mov     si, input_buffer
   mov     di, about_command_name
   add     di, word [kernel_origin + 2]
   mov     dx, about_name_len
   mov     byte [command], 1
   call    check_command

   cmp     byte [command], 0
   jne     command_identified

   ... Similar checks for about, clear and time ...

   jmp     unknown_err_display

   command_identified:
       call    interpret_command

   cli_cycle_end:
       jmp     terminate
```

```
check_command:
  push    si
  dec     si
  mov     cx, -1

  find_len_loop:
      inc     si
      inc     cx

      cmp     byte [si], 00h
      je      check_command_len

      jmp     find_len_loop

  check_command_len:
      pop     si

      cmp     cx, dx
      jne     not_identified

  check_command_letters:
      dec     cx
      jz      identified

      mov     ax, [si]
      mov     bx, [di]

      cmp     ax, bx
      jne     not_identified

      inc     si
      inc     di

      jmp
check_command_letters

  identified:
      ret

  not_identified:
      mov     byte [command], 0
      ret
```

```
interpret_command:
  cmp     byte [command], 1
  je      interpret_about

  cmp     byte [command], 2
  je      interpret time

  cmp     byte [command], 3
  je      interpret_clear

  cmp     byte [command], 4
  je      interpret_exit

  jmp     interpretation_end

  interpret_about:
      mov     si, about_string
      add     si, word [kernel_origin + 2]
      mov     cx, about_string_len
      call    print_str

      jmp     interpretation_end

  ... Code for handling all the
      other commands ...

  interpretation_end:
      ret
```

The last thing I would like to mention is the implementation of *datetime*. For this purpose I've found INT 1ah ah=02h - read time from CMOS RTC and ah=04h - read date from CMOS RTC to be the best options. In short, after this interruptions all the data required ends up in the registers described in the documentation in Binary-Coded Decimal format (01h = 1d, 20h = 20d, 99h = 99d) which is very handy I just need to translate this data in the characters that will be displayed on the screen, stored in a buffer.

```
interpret_time:
   call    get_date

   mov     al, dl
   mov     si, dt_ascii_buffer + 0
   call    bcd_to_ascii
   mov     byte [dt_ascii_buffer + 2], 2fh

   ... Extract month and year ...

   call    get_time

   mov     al, ch
   mov     si, dt_ascii_buffer + 11
   call    bcd_to_ascii
   mov     byte [dt_ascii_buffer + 13], 3ah

   ... Extract minute ...

   mov     si, dt_ascii_buffer
   mov     cx, 16
   call    print_str

   jmp     interpretation_end

get_time:

   ; int 1ah ah=02h - read time from CMOS RTC

   mov     ah, 02h
   int     1ah

   ; ch - hours, cl - minutes, dh - seconds (all in BCD)

   ret

get_date:

   ; int 1ah ah=04h - read date from from CMOS RTC

   mov     ah, 04h
   int     1ah

   ; ch - century, cl - year, dh - month, dl - day (all in BCD)

   ret

bcd_to_ascii:

   ; ax - value to translate
   ; si - pointer to a place in the buffer to store the characters

   xor     ah, ah
   mov     bl, 10h
   div     bl

   add     al, 30h
   add     ah, 30h

   mov     [si], al
   mov     [si + 1], ah

   ret
```

Results:

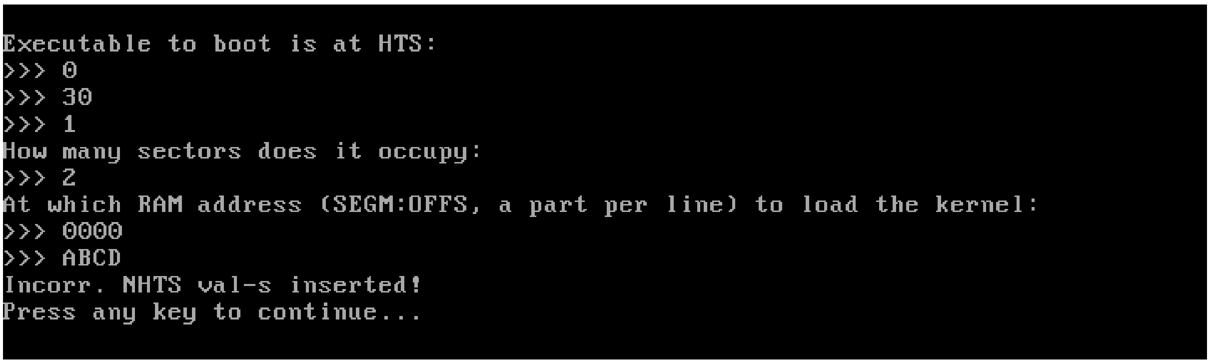

Figure 1. Incorrect N / HTS val-s inserted

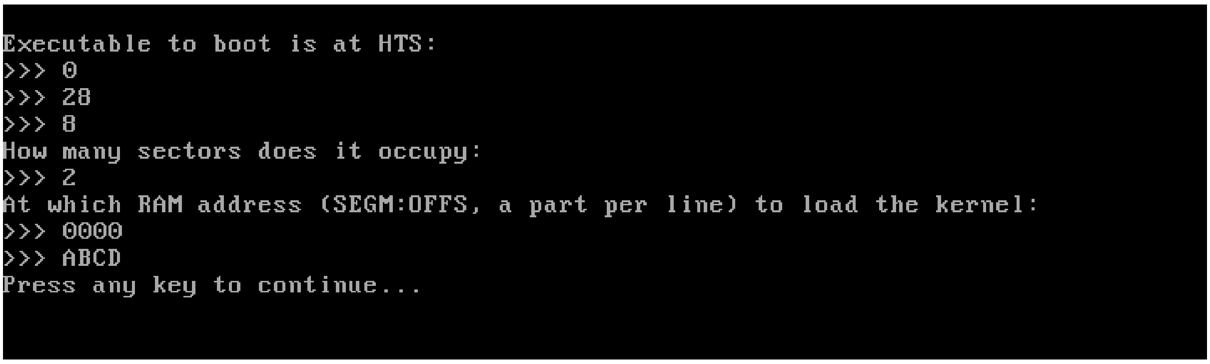

Figure 2. Kernel successfully booted

Figure 3. CLI

Conclusion:

In conclusion, this project has been a fulfilling and enlightening experience, marked by the successful implementation of a quite complex boot manager capable of reading from an arbitrary disk place and validating the binary read. The development of a basic Command Line Interface (CLI) capable of parsing and interpreting simple one-token commands was a challenge in which I fully developed my assembly coding skills. The biggest challenge for me was dynamically adjusting the origin of the kernel, which gave me some valuable insights in the labeling and addressing processes. Also I found out about BCD which is an interesting decimal representation format. Moving forward, the knowledge gained from this laboratory work lays a solid foundation for more advanced OS development endeavors.