MINISTERUL EDUCAŢIEI REPUBLICII MOLDOVA

UNIVERSITATEA TEHNICĂ A MOLDOVEI

Facultatea „Calculatoare, Informatică şi Microelectronică”

FILIERA ANGLOFONĂ

RAPORT

Operational Systems

Creating an operational system and functions to it

A efectuat:

st. gr. FAF-141 (l. engleză) Cristea Victor

A verificat:

lector. sup. Balan Mihaela

Chișinău 2016

Task: Create an operational system with a bootloader which loads a kernel with several functions bellow:

1. Calculator
2. Draw Moldova’s flag
3. String inversor
4. Clr screen
5. ASCII printer
6. String verification for palindrom
7. String inversor
8. HELP
9. Exit

Work flow:

What is a Bootloader?

A bootloader is a special program that is executed each time a bootable device is initialized by the computer during its power on or reset that will load the kernel image into the memory. This application is very close to hardware and to the architecture of the CPU. All x86 PCs boot in Real Mode. In this mode you have only 16-bit instructions. Our bootloader runs in Real Mode and our bootloader is a 16-bit program.

How this works?

When you switch on the PC the BIOS want to boot up an OS which must be found somewhere in hard disks, floppy disk, CDs, etc. The order in which BIOS searches an OS is user configurable. Next the BIOS reads the first 512 byte sector of the bootable disk. Usually a sector is 512 bytes in size. This is known as the Master Boot Record (MBR). BIOS simply loads the contents of the MBR into memory location “0x7c00” and jumps to that location to start executing whatever code is in the MBR. Our bootloader should be 512 bytes in size as well.

In computing, the kernel is a computer program that manages input/output requests from software, and translates them into data processing instructions for the central processing unit and other electronic components of a computer. The kernel is a fundamental part of a modern computer's operating system.

The boot process is:

- Turn on your PC and BIOS executes
- The BIOS seeks the MBR in boot order which is user configurable.
- The BIOS loads a 512 byte boot sector into memory location “0x7c00” from the specified media and begins executing it.
- Those 512 bytes then go on to load the OS itself or a more complex bootloader.

BIOS Interrupts

These interrupts help OS and application invoke the facilities of the BIOS. This is loaded before the bootloader and it is very helpful in communicating with the I/O. Since we don't have OS level interrupts this is the only option that would be helpful.

For example to print a character to the screen using BIOS interrupt calls.

```
mov ah, 0x0e    ; function number = 0Eh : Display Character
mov al, 'O'     ; AL = code of character to display
int 0x10        ; call INT 10h, BIOS video service
```

This is a simple boot loader written in AT&T syntax.

```
        .code16
        .section .text
        .global main
main:

/*
        Disk description table, to make it a valid floppy
        FAT12 file system format
*/
        jmp _start
        .byte   144             #NOP
        .ascii  "OsandaOS"      #OEMLabel
        .word   512             #BytesPerSector
        .byte   1               #SectorsPerCluster
        .word   1               #ReservedForBoot
        .byte   2               #NumberOfFats
        .word   224             #RootDirEntries (224 * 32 = 7168 = 14 sectors to read)
        .word   2880            #LogicalSectors
        .byte   0xf0            #MediumByte
        .word   9               #SectorsPerFat
        .word   18              #SectorsPerTrack
        .word   2               #Sides
        .long   0               #HiddenSectors
        .byte   0               #LargeSectors
        .byte   0               #DriveNo
        .byte   0x29            #Signature (41 for Floppy)
        .long   0x12345678      #VolumeID
        .ascii  "My First OS"   #VolumeLabel
        .ascii  "FAT12   "      #FileSystem

_start:
        movw $0, %ax
        movw %ax, %ss
        movw %ax, %ds
        movw %ax, %es
        movw $string, %si
loop:
        movb    $0xe, %ah
        movb    (%si), %al
```

```
        cmpb    $0, %al
        je      done
        int     $0x10
        addw    $1, %si
        jmp     loop
done:
        jmp     done            #infinite loop

string:
        .ascii  "Welcome to my First OS :)"
        .byte 0
        .fill 0x1fe - (. - main) ,1,0   #Pad remainder of boot sector with 0s
        .word   0xaa55                  #The standard PC boot signature
```

Assemble the file using binary as the format.

nasm -f bin -o myfirst.bin   myfirst.nasm

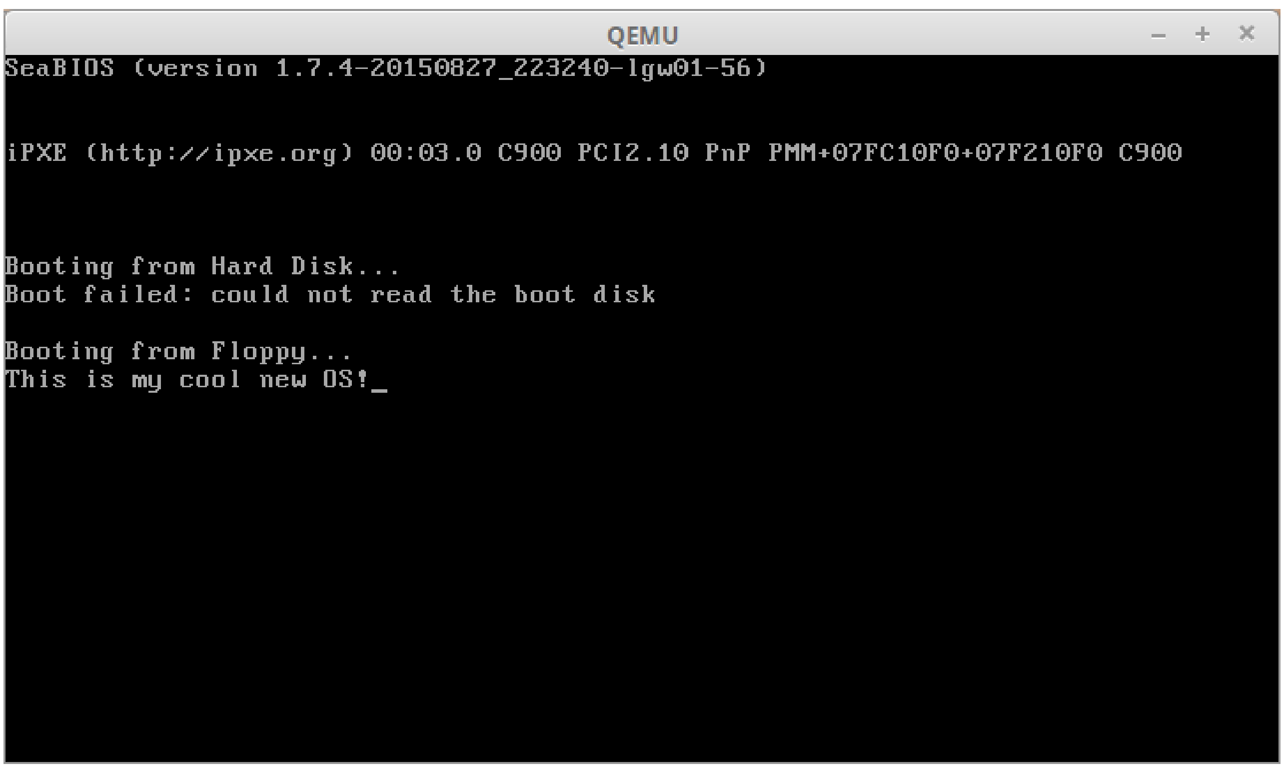
If you use the file utility you will see that it's a legit 1.4MB floppy Disk and a 32-bit boot sector.

After that convert the binary file to floppy image.

dd status=noxfer conv=notrunc if= lmyfirst.bin of=fmyfirst.flp

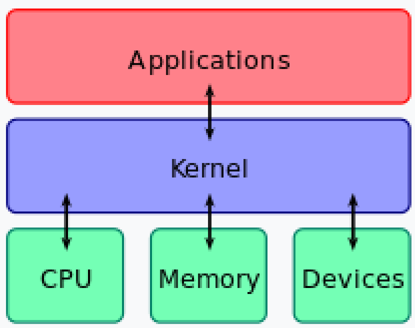
You can also convert the binary file to an ISO using tools such as UltraISO, etc. You may burn and try in your PC instead of emulating.
Use Qemu to test our newly created bootloader

qemu-system-i386 -fda myfirst.flp



Kernel

The **kernel** (also called **nucleus**) is a computer program that constitutes the central core of a computer's operating system. It has complete control over everything that occurs in the system.[1] As such, it is the first program loaded on startup, and then manages the remainder of the startup, as well as input/output requests from software, translating them into data processing instructions for the central processing unit. It is also responsible for managing memory, and for managing and communicating with computing peripherals, like printers, speakers, etc. The kernel is a fundamental part of a modern computer's operating system.



A kernel connects the application software to the hardware of a computer.

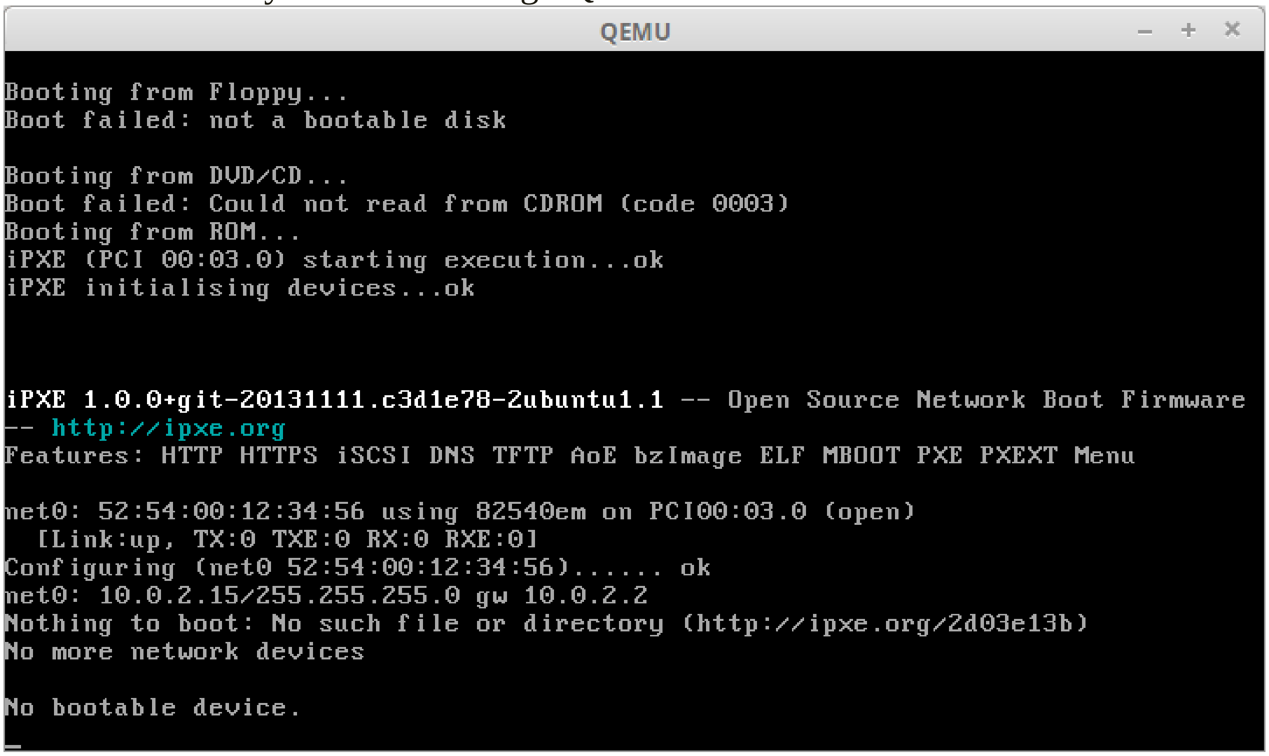
The critical code of the kernel is usually loaded into a *protected area* of memory, which prevents it from being overwritten by other, less frequently used parts of the operating system or by applications. The kernel performs its tasks, such as executing processes and handling interrupts, in *kernel space*, whereas everything a user normally does, such as writing text in a text editor or running programs in a GUI (graphical user interface), is done in *user space*. This separation prevents user data and kernel data from interfering with each other and thereby diminishing performance or causing the system to become unstable (and possibly crashing).[1]

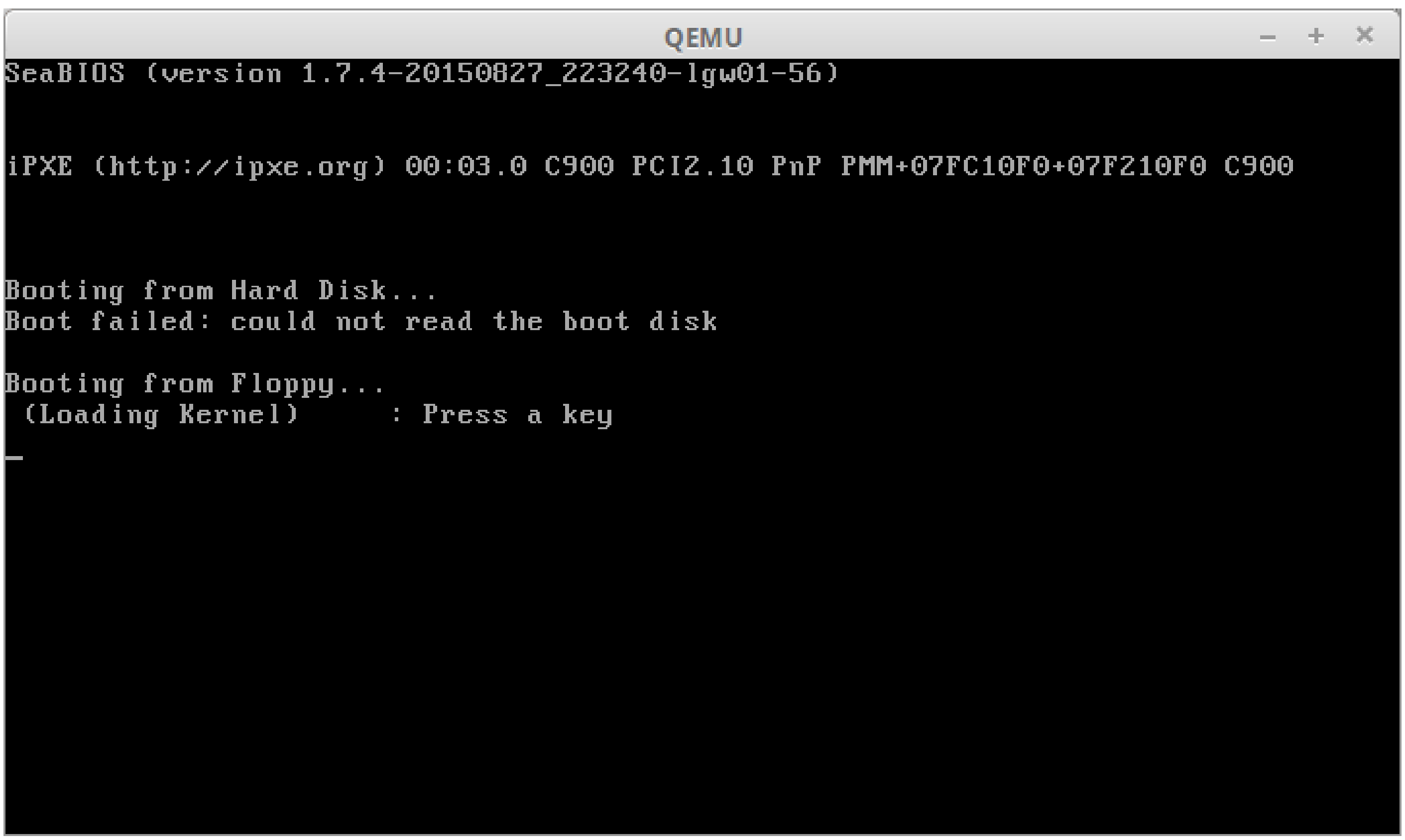
When a *process* makes requests of the kernel, the request is called a system call. Various kernel designs differ in how they manage system calls and resources. For example, a monolithic kernel executes all the operating system instructions in the same address space in order to improve the performance[of the system. A microkernel runs most of the operating system's background processes in user space,[3] allowing a more modular design of the kernel which makes it easier to maintain.[4]

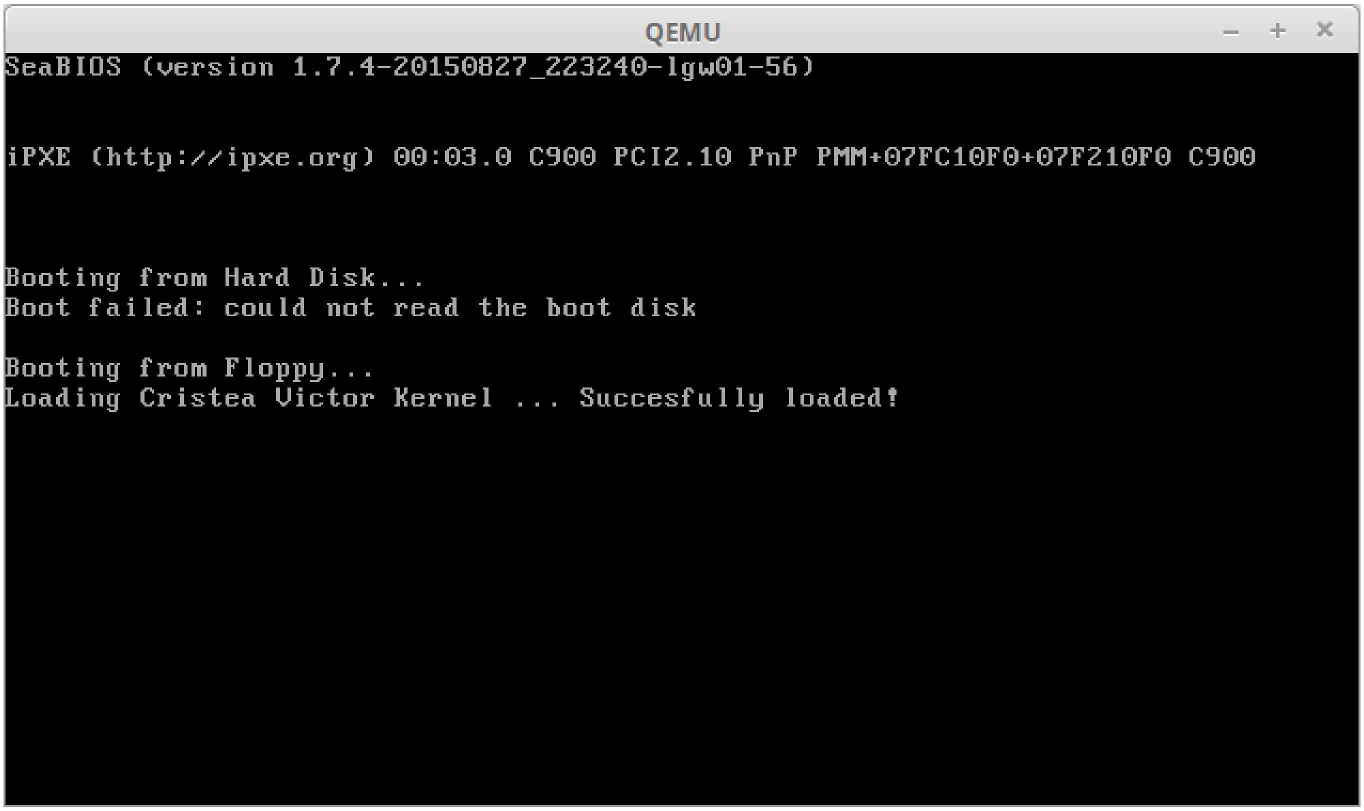
The kernel's interface is a low-level abstraction layer.

Bootloader is loaded, and I can go to kernel.

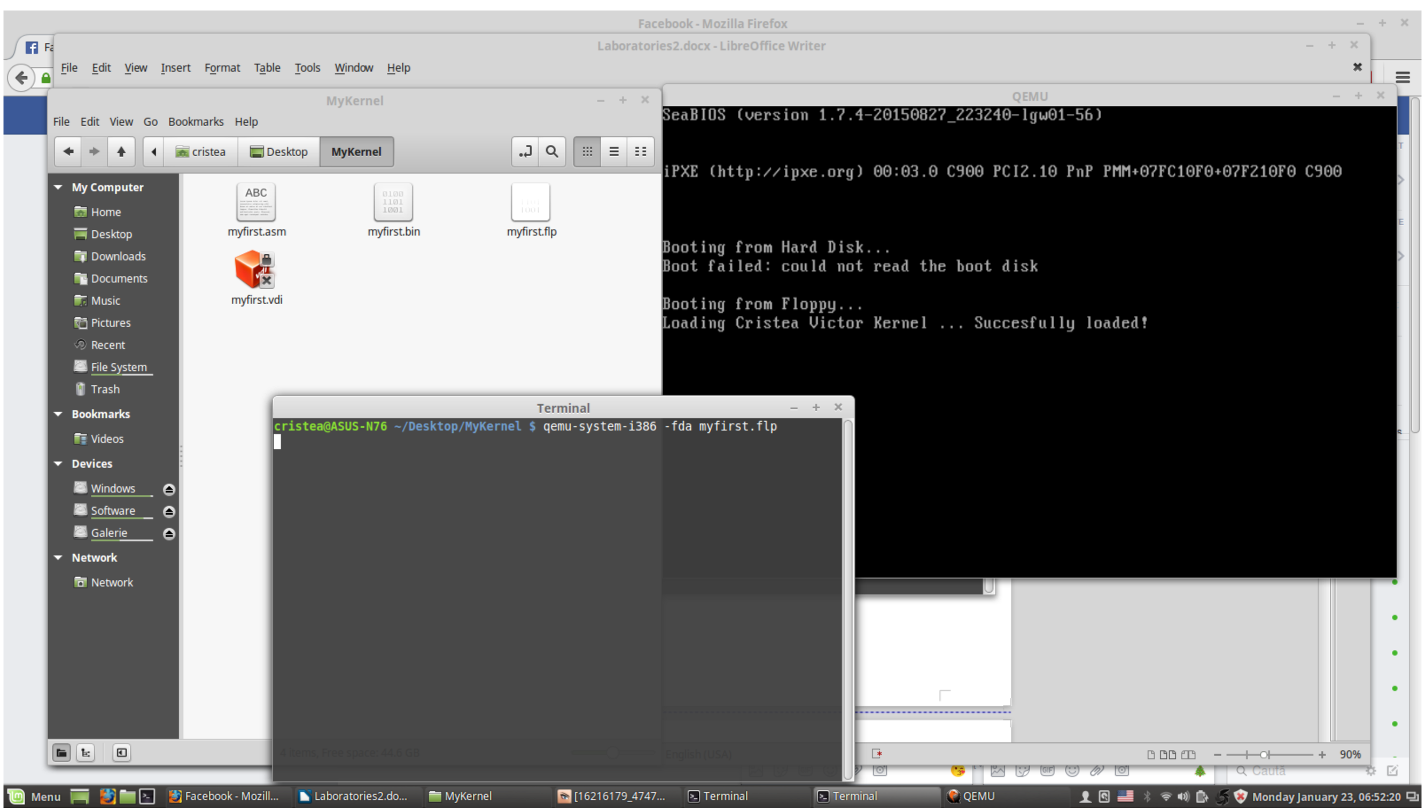
Now I can use my functions through Qemu Terminal:

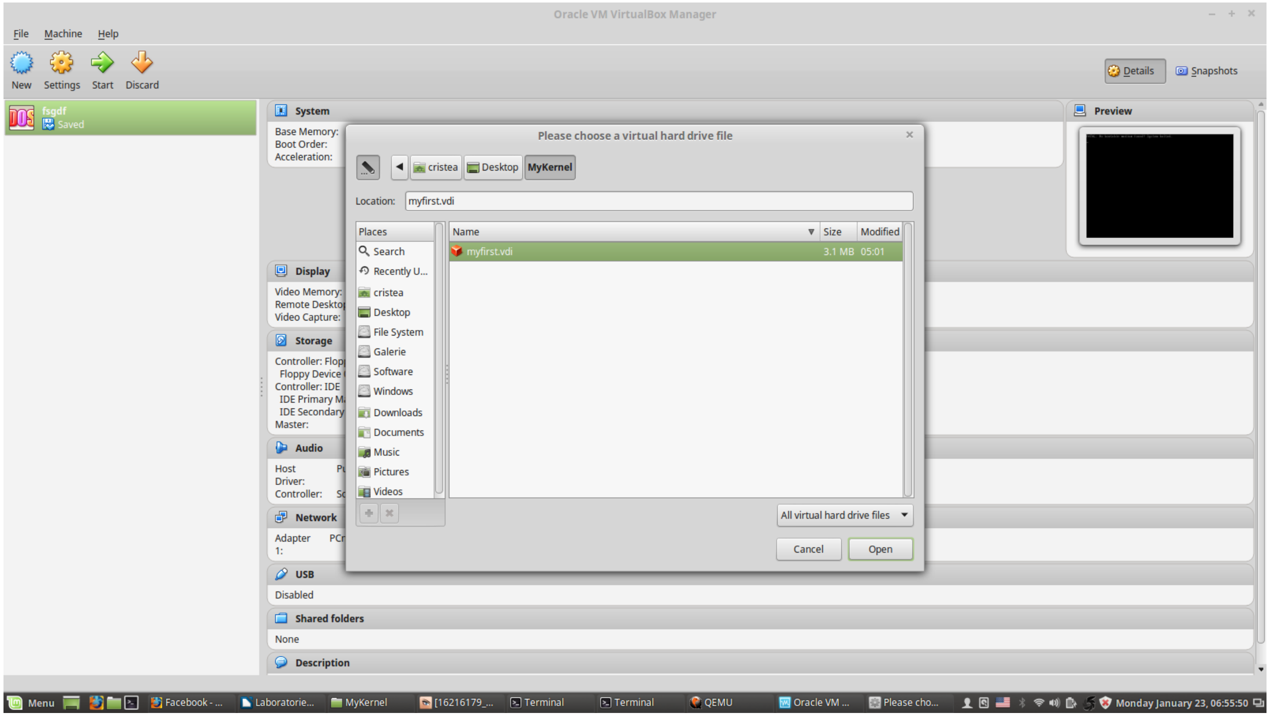






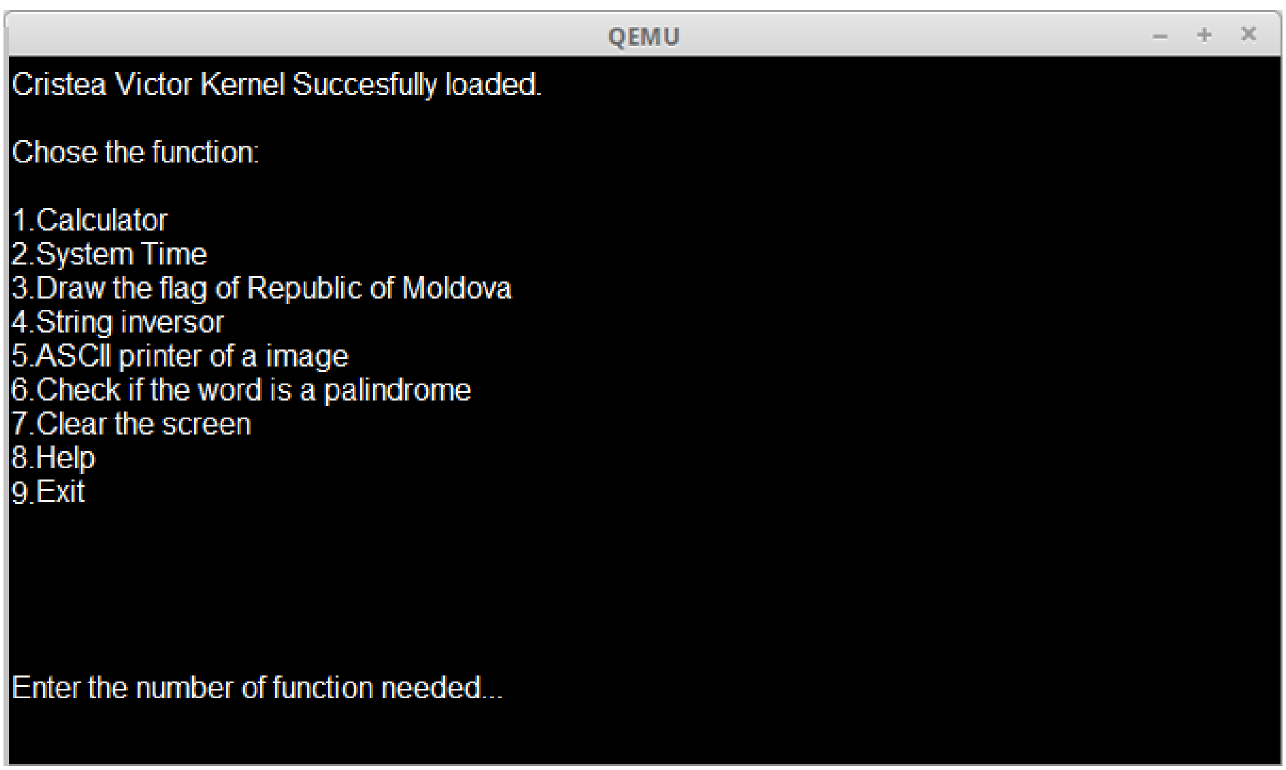
So here I have a Virtual disk image for launching through Virtualbox or Vmware and FLP for launching through QEMU.





Where we have booted the system

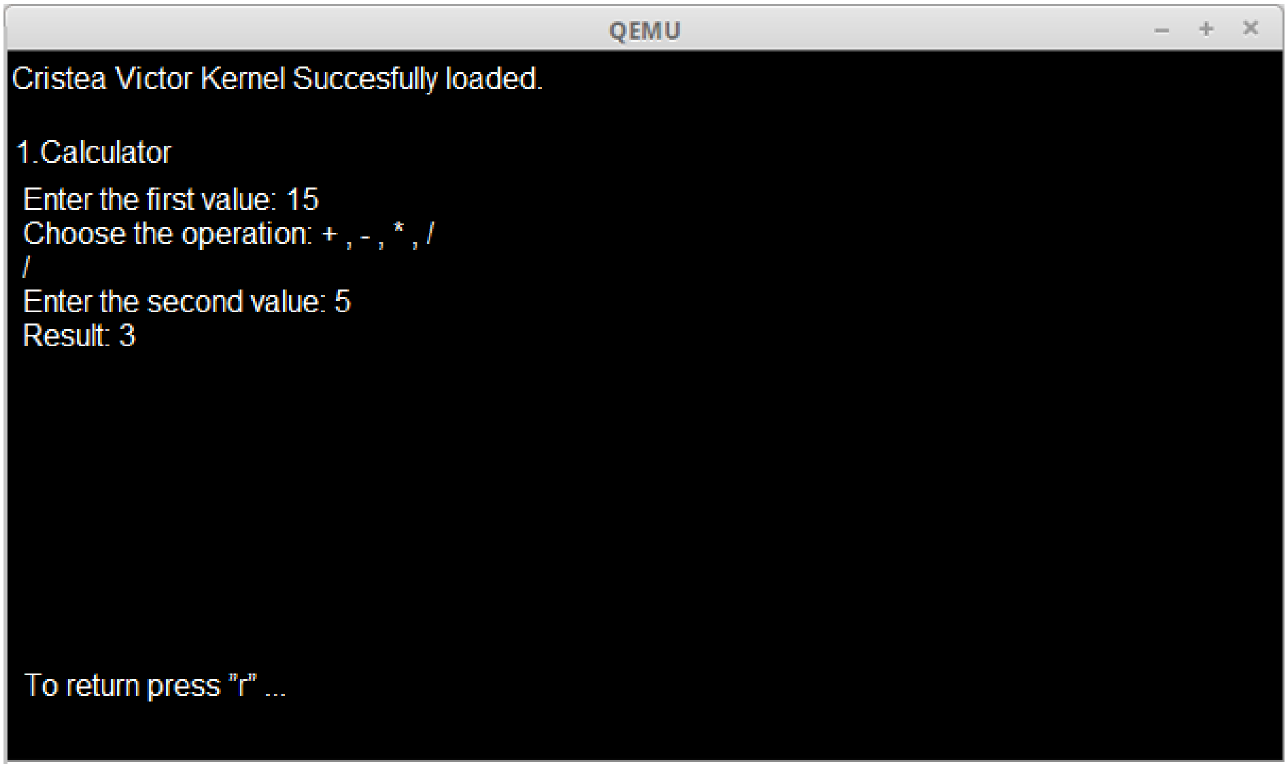
At the references you'll get github location of the project



Sequence of code for Main function which loads the screen above:

```
Main:

mov si, WaitInputMessage
call Writeline

xor ax, ax
int 16h ; wait for keypress

; set destination segment
mov   ax, 7E0h
mov   es, ax
xor   bx, bx

ReadFromFloppy:
  mov ah, 2 ; read sectors
  mov al, 3 ; number of sectors to read
  mov ch, 0 ; track number
  mov cl, 2 ; sector number (kernel is in the second sector)
  mov dh, 0 ; head number
  mov dl, 0 ; drive number
  int 13h   ; call BIOS
  jc ReadFromFloppy       ; Error, so try again


; check data integrity
mov al, byte [es:0h]
cmp al, 0B8h
je LoadKernel


; print error message and halt system
mov si, ErrorMessage
call Writeline

cli
hl
```

Appendix for Calculator




```
        operation1: .word 0
        operand1:   .word 0
        operand2:   .word 0
        operation2: .word 0
        operand3:   .word 0
        priority:   .word 0

        msg1: .asciiz     "\nEnter
the first valuer: "

        msg3: .asciiz "Choose the
operation:+,-,*,/: "
        msg4: .asciiz "Enter the
second value: "

        msg7: .asciiz "Result: "


        .text
        .globl main

main:
                #get 1st operation
                li $v0, 4
```

```
#system call for print str
      la $a0,msg1
#addr of string to print
      syscall
      li $v0, 5
#read 1st operand
      syscall
      sw $v0, operand1



      #get 1st operand
      li $v0, 4
      la $a0, msg2
      syscall

      li $v0, 5
#read 1st operation
      syscall
      sw $v0, operation1
      #get 2nd operand
      li $v0, 4
      la $a0, msg3
      syscall
      li $v0, 5
      syscall
      sw $v0, operand2
      #get 2nd operation
      li $v0, 4
      la $a0, msg4
      syscall
      li $v0, 5
      syscall
      sw $v0, operation2
      #get 3rd operand
      li $v0, 4
      la $a0, msg5
      syscall

      li $v0, 5
      syscall
      sw $v0, operand3
      #get operation priority
      li $v0, 4
      la $a0, msg6
```

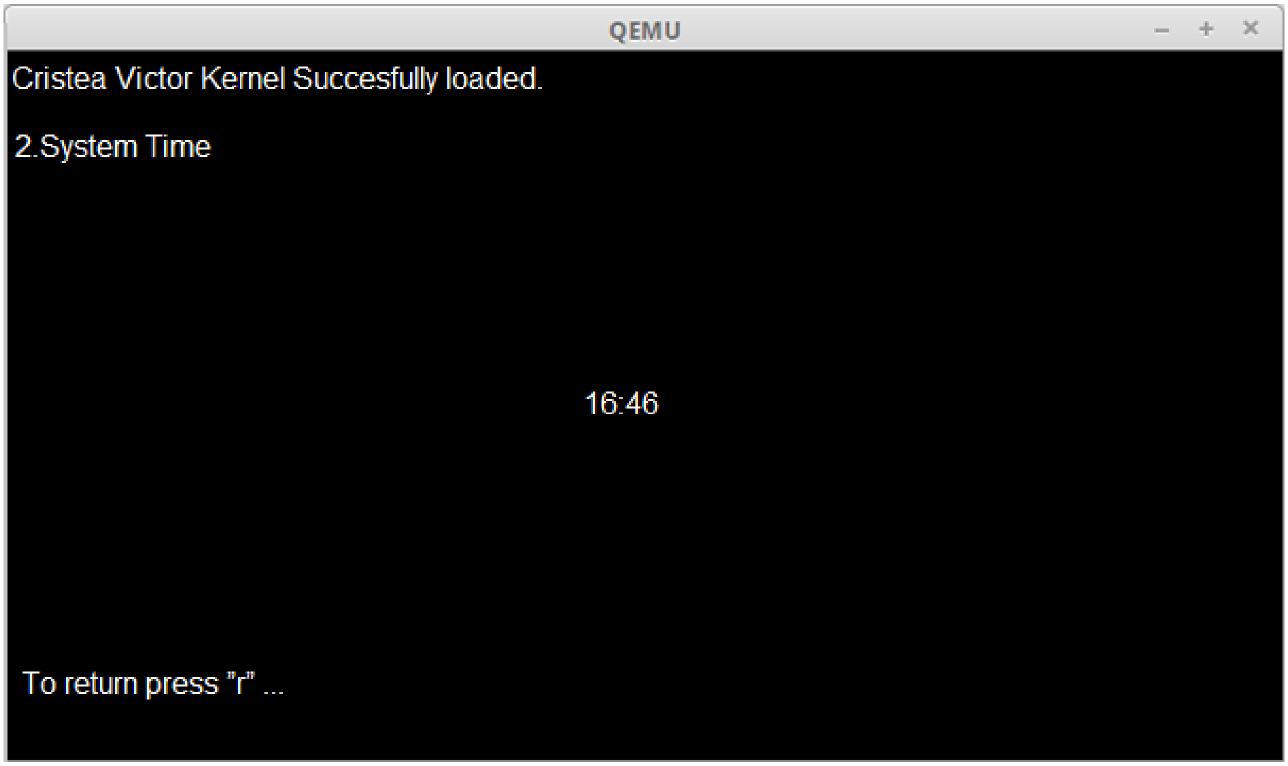
```
            syscall
            li $v0, 5
            syscall
            sw $v0, priority

            #put operations and
operands into $t0-$t5
            lw $t0, operation1
            lw $t1, operand1
            lw $t2, operand2
            lw $t3, operation2
            lw $t4, operand3
            lw $t5, priority

            #determine which
priority to use

            #default operation
priority
            beq $t5, 0, default
            #1st operation priority
            beq $t5, 1, first
            #2nd operation priority
            beq $t5, 2, second


            li    $v0, 10
      # system call code for exit =
10
            syscall
default:
            beq $t0, 3, first
            beq $t0, 4, first
            beq $t3, 3, second
            beq $t3, 4, second

first:
            #perform operation1 on
operand1 and operand2
            #set priority to 1 for
default operations
            li $t5, 1
            move $a0, $t1
            move $a1, $t2
            beq $t0, 1, addit1
```

```
            beq $t0, 2, subtr1
            beq $t0, 3, multi1
            beq $t0, 4, divis1
first2:
            #perform operation2 on
result and operand3
            #place operand in $a1
            move $a1, $t4

            beq $t3, 1, addit2
            beq $t3, 2, subtr2
            beq $t3, 3, multi2
            beq $t3, 4, divis2
            j Exit


second:           #perform
operation2 on operan2 and operand3
            #set priority to 2 for
default operations
            li $t5, 2
            move $a0, $t2
            move $a1, $t4
            beq $t3, 1, addit2
            beq $t3, 2, subtr2
            beq $t3, 3, multi2
            beq $t3, 4, divis2

second2:
            #perform operation 1 on
operand1 and result
            move $a1, $a0     #puts
result into $a1
            move $a3, $a0
      #saves result in case it's
lost on 2nd pass
            move $a0, $t1
            beq $t0, 1, addit1
            beq $t0, 2, subtr1
            beq $t0, 3, multi1
            beq $t0, 4, divis1

            move $a0, $a3
      #retains result
            j Exit

addit1:
```

```
            #place the operands in
$a0 and $a1
            #reset operation
            li $t0, 0
            jal addition
addit2:
            #place the operands in
$a0 and $a1
            #reset operation
            li $t3, 0

            jal addition
subtr1:           #place the
operands in $a0 and $a1
            #reset operation
            li $t0, 0
            jal subtraction

subtr2:           #place the
operands in $a0 and $a1
            #reset operation
            li $t3, 0

            jal subtraction

multi1:           #place the
operands in $a0 and $a1
            #reset operation
            li $t0, 0

            jal multiplication

multi2:     #place the operands in
$a0 and $a1
            #reset operation
            li $t3, 0

            jal multiplication

divis1:           #place the
operands in $a0 and $a1
            #reset operation
            li $t0, 0

            jal division
```

```
divis2:     #place the operands in
$a0 and $a1
            #reset operation
            li $t3, 0

            jal division

addition:
            #take the operands
perform addition
            add $a0, $a0, $a1

            beq $t5, 1, Jumpfirst
            beq $t5, 2, Jumpsecond
subtraction:      #take the
operands perform subtraction
            sub $a0, $a0, $a1

            beq $t5, 1, Jumpfirst
            beq $t5, 2, Jumpsecond
multiplication:  #take the operands
perform multiplication
            mult $a0, $a1
            mflo $a0
            beq $t5, 1, Jumpfirst
            beq $t5, 2, Jumpsecond
division:   #take the operands
perform division
            div $a0, $a0, $a1

            beq $t5, 1, Jumpfirst
            beq $t5, 2, Jumpsecond
Jumpfirst:
            j first2
Jumpsecond:
            j second2
Exit:             ##prints result
            move $v1, $a0
            li $v0, 4
            la $a0, msg7
            syscall
            li $v0, 1
            move $a0, $v1
            syscall

            #jumps back to main
            j main
```

Appendix for System Time




```
.MODEL SMALL
 .STACK 100H

 .DATA
   PROMPT   DB  'Current System Time is : $'
   TIME     DB  '00:00:00$'         ; time format hr:min:sec

 .CODE
   MAIN PROC
     MOV AX, @DATA                  ; initialize DS
     MOV DS, AX

     LEA BX, TIME                   ; BX=offset address of string TIME

     CALL GET_TIME                  ; call the procedure GET_TIME

     LEA DX, PROMPT                 ; DX=offset address of string PROMPT
     MOV AH, 09H                    ; print the string PROMPT
     INT 21H

     LEA DX, TIME                   ; DX=offset address of string TIME
     MOV AH, 09H                    ; print the string TIME
     INT 21H

     MOV AH, 4CH                    ; return control to DOS
     INT 21H
   MAIN ENDP

;**************************************************************************;
```

```
 ;------------------------------  GET_TIME  --------------------------------
;

   GET_TIME PROC
    ; this procedure will get the current system time
    ; input : BX=offset address of the string TIME
    ; output : BX=current time

    PUSH AX                         ; PUSH AX onto the STACK
    PUSH CX                         ; PUSH CX onto the STACK

    MOV AH, 2CH                     ; get the current system time
    INT 21H

    MOV AL, CH                      ; set AL=CH , CH=hours
    CALL CONVERT                    ; call the procedure CONVERT
    MOV [BX], AX                    ; set [BX]=hr  , [BX] is pointing to hr
                                    ; in the string TIME

    MOV AL, CL                      ; set AL=CL , CL=minutes
    CALL CONVERT                    ; call the procedure CONVERT
    MOV [BX+3], AX                  ; set [BX+3]=min  , [BX] is pointing to
min
                                    ; in the string TIME

    MOV AL, DH                      ; set AL=DH , DH=seconds
    CALL CONVERT                    ; call the procedure CONVERT
    MOV [BX+6], AX                  ; set [BX+6]=min  , [BX] is pointing to
sec
                                    ; in the string TIME

    POP CX                          ; POP a value from STACK into CX
    POP AX                          ; POP a value from STACK into AX

    RET                             ; return control to the calling procedure
   GET_TIME ENDP                    ; end of procedure GET_TIME

  ;-------------------------------  CONVERT  ------------------------------;

   CONVERT PROC
    ; this procedure will convert the given binary code into ASCII code
    ; input : AL=binary code
    ; output : AX=ASCII code

    PUSH DX                         ; PUSH DX onto the STACK

    MOV AH, 0                       ; set AH=0
    MOV DL, 10                      ; set DL=10
    DIV DL                          ; set AX=AX/DL
    OR AX, 3030H                    ; convert the binary code in AX into ASCII

    POP DX                          ; POP a value from STACK into DX

    RET                             ; return control to the calling procedure
   CONVERT ENDP                     ; end of procedure CONVERT

 END MAIN
```

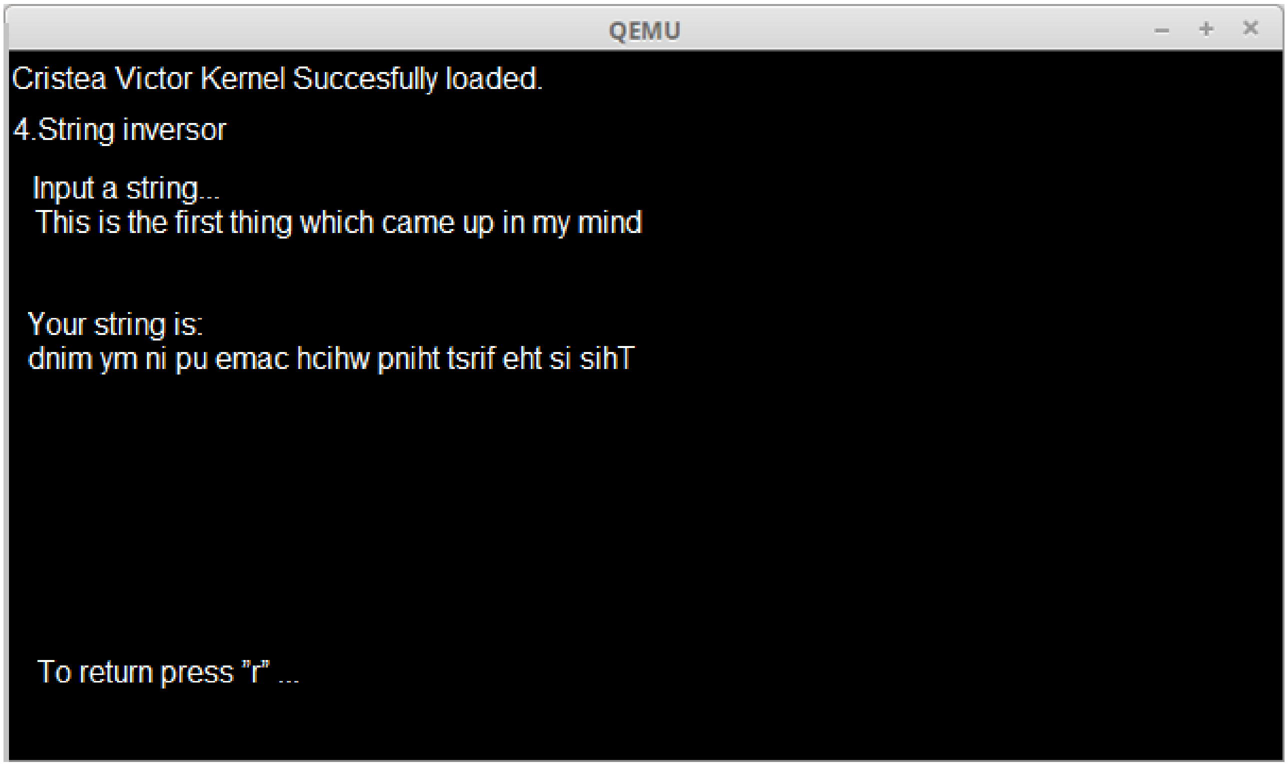
Appendix for Drawing


```
DrawImage:

    cmp word [ID], "BM"
    je DrawBMP

    ret

DrawBMP:

    cmp word [Depth], 24
    je DrawBMP24

    cmp word [Depth], 32
    je DrawBMP32

    ret

DrawBMP24:

    pushad
    mov [.X], eax
    mov [.Y], ebx
    mov [.x], eax
    mov [.y], ebx
    xor edx, edx
    mov [.align], edx
    mov ebx, 4
    mov eax, [Width]
    div ebx
    cmp edx, 0
    je .skip

    mov dword [.align], 1
    xor edx, edx
```

```
    mov eax, [Width]
    inc eax
    div ebx
    cmp edx, 0
    je .skip

    mov dword [.align], 2
    xor edx, edx

    mov eax, [Width]
    add eax, 2
    div ebx
    cmp edx, 0
    je .skip

    mov dword [.align], 3
.skip:

    mov esi, BMP
    add esi, [Offset]
    mov [.pointer], esi
    mov ebx, [Height]
    add [.y], ebx

.drawLine:

    mov esi, [.pointer]
    mov al, [esi]
    mov [.color], al
    mov al, [esi + 1]
    mov [.color + 1], al
    mov al, [esi + 2]
    mov [.color + 2], al
    add dword [.pointer], 3

    mov esi, .params
    call DrawPixel

    inc dword [.x]
    mov ebx, [Width]
    add ebx, [.X]
    cmp [.x], ebx
    jne .drawLine

    mov ebx, [.X]
    mov [.x], ebx
    dec dword [.y]

    mov ebx, [.pointer]
    add ebx, [.align]
    mov [.pointer], ebx

    mov ebx, [.Y]
    cmp [.y], ebx
    je .done

    jmp short .drawLine

.done:
    mov dword [.y], 0
    popad
```

```
    ret

.X dd 0
.Y dd 0
.align dd 0
.pointer dd 0

.params:
    .color dd 0
    .x dd 0
    .y dd 0

DrawBMP32:
    pushad
    mov [.X], eax
    mov [.Y], ebx
    mov [.x], eax
    mov [.y], ebx
    mov esi, BMP
    add esi, [Offset]
    mov [.pointer], esi
    mov ebx, [Height]
    add [.y], ebx
.drawLine:
    mov esi, [.pointer]
    mov eax, [esi]
    mov [.color], eax
    add dword [.pointer], 4

    mov esi, .params
    call DrawPixel

    inc dword [.x]
    mov ebx, [Width]
    add ebx, [.X]
    cmp [.x], ebx
    jne .drawLine

    mov ebx, [.X]
    mov [.x], ebx
    dec dword [.y]

    mov ebx, [.Y]
    cmp [.y], ebx
    je .done

    jmp short .drawLine
.done:

    mov dword [.y], 0
    popad
    ret

.X dd 0
.Y dd 0
.pointer dd 0
.params:

    .color dd 0
    .x dd 0
    .y dd 0
```

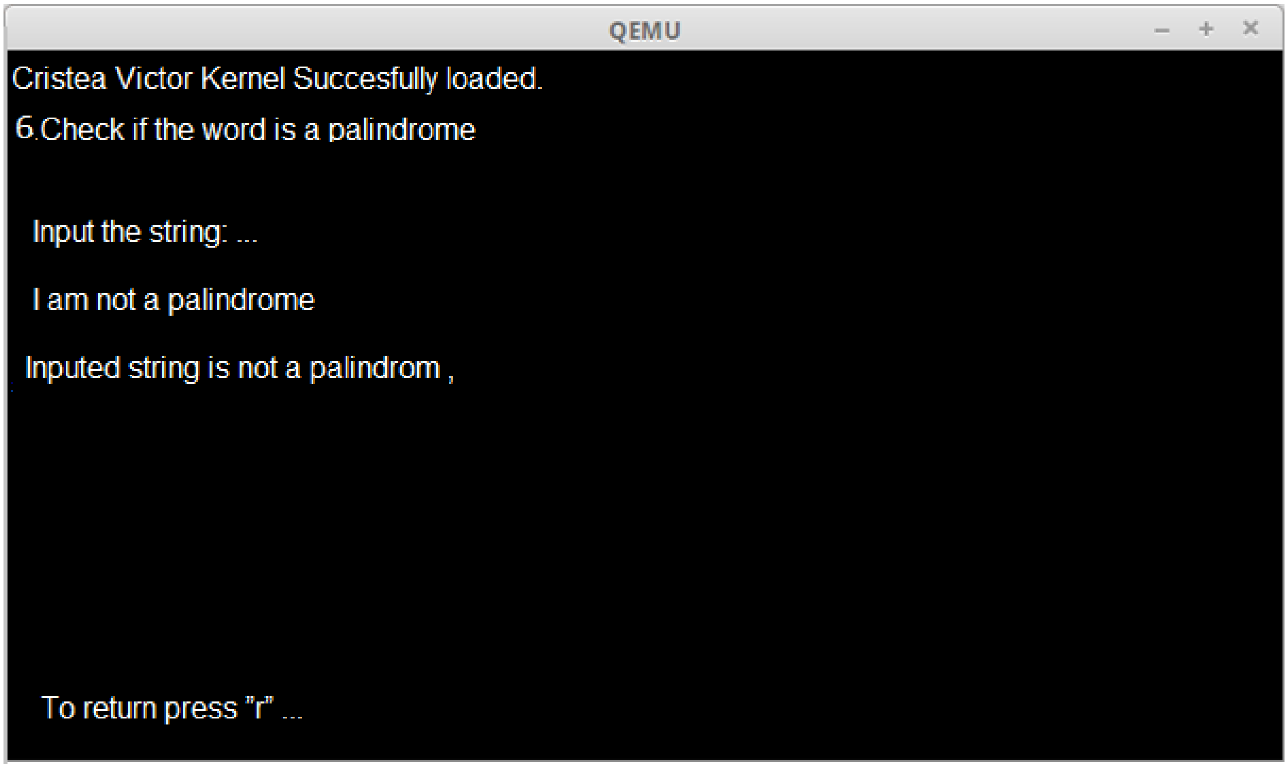
Appendix for string inversor




```
DATA SEGMENT
        STR1 DB "ENTER YOUR STRING HERE ->$"
        STR2 DB "YOUR STRING IS ->$"
        STR3 DB "REVERSE STRING IS ->$"
        INSTR1 DB 20 DUP("$")
        RSTR DB 20 DUP("$")
        NEWLINE DB 10,13,"$"
        N DB ?
        S DB ?

DATA ENDS

CODE SEGMENT
        ASSUME DS:DATA,CS:CODE
START:

        MOV AX,DATA
        MOV DS,AX

        LEA SI,INSTR1

;GET STRING
        MOV AH,09H
        LEA DX,STR1
        INT 21H

        MOV AH,0AH
        MOV DX,SI
        INT 21H


        MOV AH,09H
```

```
        LEA DX,NEWLINE
        INT 21H

;PRINT THE STRING

        MOV AH,09H
        LEA DX,STR2
        INT 21H

        MOV AH,09H
        LEA DX,INSTR1+2
        INT 21H

        MOV AH,09H
        LEA DX,NEWLINE
        INT 21H

;PRINT THE REVERSE OF THE STRING

        MOV AH,09H
        LEA DX,STR3
        INT 21H
        MOV CL,INSTR1+1
        ADD CL,1
        ADD SI,2
     L1:
        INC SI

        CMP BYTE PTR[SI],"$"
        JNE L1

        DEC SI

        LEA DI,RSTR

     L2:MOV AL,BYTE PTR[SI]

        MOV BYTE PTR[DI],AL

        DEC SI
        INC DI
        LOOP L2

        MOV AH,09H
        LEA DX,NEWLINE
        INT 21H

        MOV AH,09H
        LEA DX,RSTR
        INT 21H


        MOV AH,09H
        LEA DX,NEWLINE
        INT 21H

        MOV AH,4CH
        INT 21H


CODE ENDS
END START
```

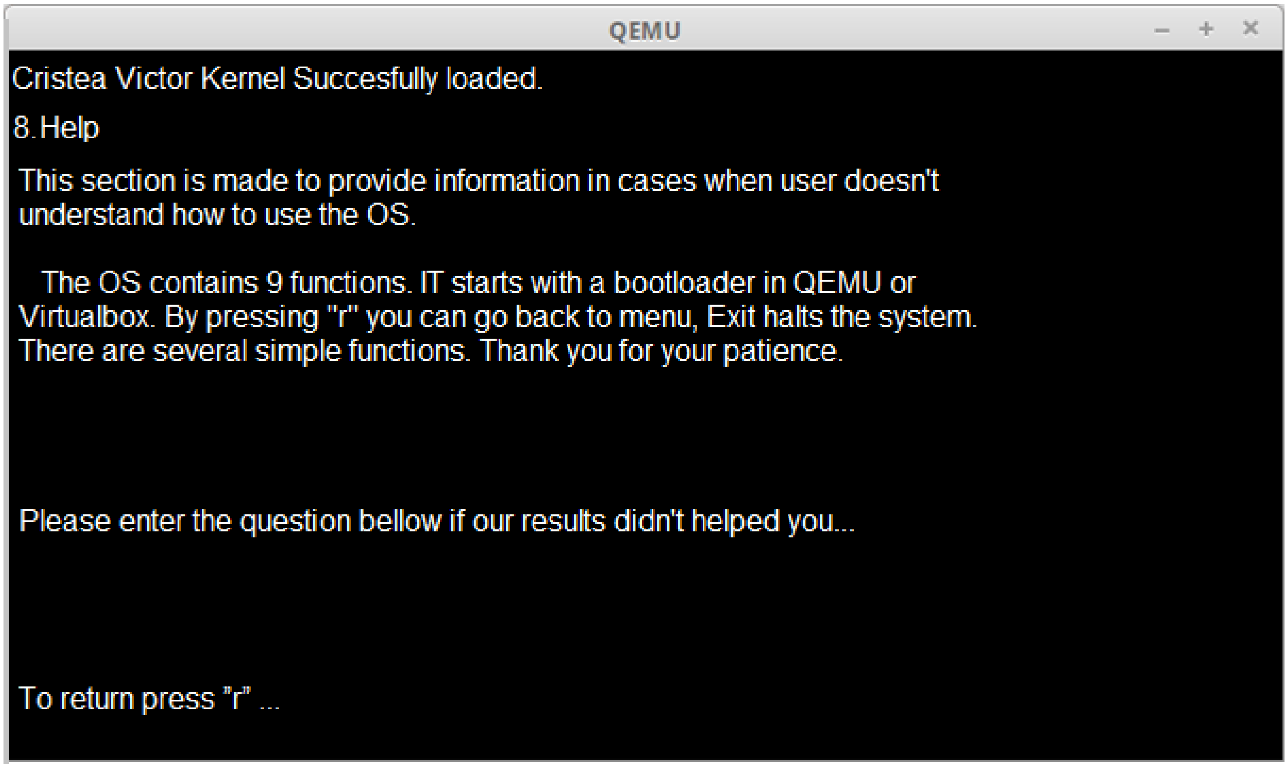
Appendix for Palindrome




```
; this sample checks if string is a palindrome or not.
; palindrome is a text that can be read backwards
; and give the same meaning as if it was read forward.
; for example: "abba" is polindrome.
; note: this program is case sensitive, "abba" is not "abba".

name "pali"

org 100h

jmp start

m1:
s db 'able was ere ere saw elba'
s_size = $ - m1
  db 0Dh,0Ah,'$'

start:

; first let's print it:
mov ah, 9
mov dx, offset s
int 21h


lea di, s
mov si, di
```

```
add si, s_size
dec si  ; point to last char!

mov cx, s_size
cmp cx, 1
je is_palindrome  ; single char is always palindrome!

shr cx, 1     ; divide by 2!

next_char:
    mov al, [di]
    mov bl, [si]
    cmp al, bl
    jne not_palindrome
    inc di
    dec si
loop next_char


is_palindrome:
   ;  the string is "palindrome!"
   mov ah, 9
   mov dx, offset msg1
   int 21h
jmp stop

not_palindrome:
   ;  the string is "not palindrome!"
   mov ah, 9
   mov dx, offset msg2
   int 21h
stop:


; wait for any key press:
mov ah, 0
int 16h


ret


msg1 db "  this is palindrome!$"
msg2 db "  this is not a palindrome!$"
```

Function HELP



Conclusion: In this laboratory works I have understood how to create an operating system, elaborating a menu for control through command Help. Writing code in assembly, then compiling with nasm, then converting and finally start on a virtual machine QEMU.It was difficult to start with but after couple of hours of reading it was real to obtain a result.

REFFERENCES

https://osandamalith.com/2015/10/26/writing-a-bootloader/
http://mikeos.sourceforge.net/write-your-own-os.html
https://github.com/cristeav49/SOMIPP