MinsungOS

Simple CLI-based OS development project

Development timeframe

- 12.26.2023 ~ 02.29.2024

Setup

- `C`
- `Linux (Ubuntu 22.04.3 LTS)`

- **Install Assembler and Emulator**

- `sudo apt-get install qemu`

- **Install Kernel Compiler**

- `sudo apt-get install i386-elf-binutils`
- `sudo apt-get install i386-elf-gcc`
- `sudo apt-get install i386-elf-gdb`

- `export PATH="/usr/local/Cellar/x86_64-elf-binutils/<version>/bin/:/usr/local/Cellar/x86_64-elf-gcc/<version>/bin/:$PATH"`

Key features and functions

- Bootloader
- VGA Drivers
- Keyboard Drivers
- Shells
- Dynamic Memory Management

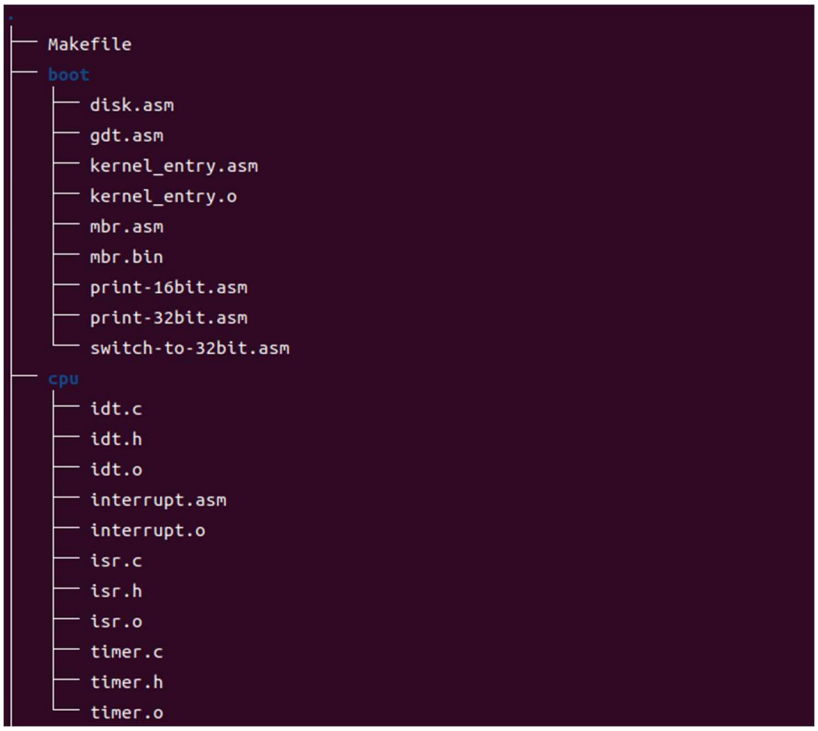
Usage

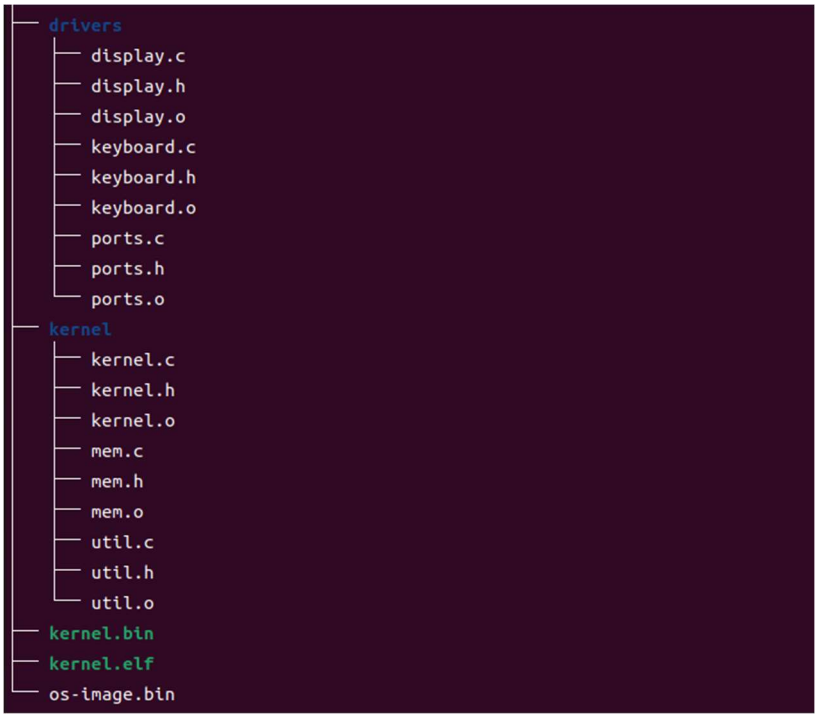
- `make run`

부트로더, VGA 드라이버, 키보드 드라이버, 쉘, 동적 메모리 관리를 합친 간단한 운영체제를 개발하였습니다.

아쉽게도 시간 관계상 부가적인 여러 기능들은 구현하지 못하였지만 어셈블리어도 처음 접해보았고 컴퓨터 구조, 운영체제에 대한 기본적인 개념들을 쌓은 것 같습니다.

어려웠지만 정말 재미있었으며 컴퓨터에 대해 지식을 조금 더 쌓고 시간이 될 때 꼭 처음부터 다시 공부하며 부가적인 기능을 추가해서 개발해보고 싶습니다.





최종 결과물 (OS)를 위해 작성한 코드인

boot의 disk.asm, gdt.asm, kernel_entry.asm, mbr.asm, print-16bit.asm, print-32bit.asm, switch-to-32bit.asm,

cpu의 idt.c idt.h, interrupt.asm, isr.c, isr.h, timer.c, timer.h,

drivers의 display.c, display.h, keyboard.c, keyboard.h, ports.c, ports.h,

kernel의 kernel.c, kernel.h, mem.c, mem.h, util.c, util.h,

Makefile에 대해 살펴보겠습니다.

disk.asm

```
; load 'dh' sectors from drive 'dl' into ES:BX
disk_load:
    pusha
    ; reading from disk requires setting specific values in all registers
    ; so we will overwrite our input parameters from 'dx'. Let's save it
    ; to the stack for later use.
    push dx

    mov ah, 0x02 ; ah <- int 0x13 function. 0x02 = 'read'
    mov al, dh   ; al <- number of sectors to read (0x01 .. 0x80)
    mov cl, 0x02 ; cl <- sector (0x01 .. 0x11)
                 ; 0x01 is our boot sector, 0x02 is the first 'available' sector
    mov ch, 0x00 ; ch <- cylinder (0x0 .. 0x3FF, upper 2 bits in 'cl')
    ; dl <- drive number. Our caller sets it as a parameter and gets it from BIOS
    ; (0 = floppy, 1 = floppy2, 0x80 = hdd, 0x81 = hdd2)
    mov dh, 0x00 ; dh <- head number (0x0 .. 0xF)
```

```
    ; [es:bx] <- pointer to buffer where the data will be stored
    ; caller sets it up for us, and it is actually the standard
location for int 13h
    int 0x13      ; BIOS interrupt
    jc disk_error ; if error (stored in the carry bit)

    pop dx
    cmp al, dh    ; BIOS also sets 'al' to the # of sectors read.
Compare it.
    jne sectors_error
    popa
    ret

disk_error:
    mov bx, DISK_ERROR
    call print16
    call print16_nl
    mov dh, ah ; ah = error code, dl = disk drive that dropped the
error
    call print16_hex ; check out the code at
http://stanislavs.org/helppc/int_13-1.html
    jmp disk_loop

sectors_error:
    mov bx, SECTORS_ERROR
    call print16

disk_loop:
    jmp $

DISK_ERROR: db "Disk read error", 0
```

```
SECTORS_ERROR: db "Incorrect number of sectors read", 0
```

gdt.asm

```
gdt_start: ; don't remove the labels, they're needed to compute sizes
and jumps
    ; the GDT starts with a null 8-byte
    dd 0x0 ; 4 byte
    dd 0x0 ; 4 byte

; GDT for code segment. base = 0x00000000, length = 0xfffff
; for flags, refer to os-dev.pdf document, page 36
gdt_code:
    dw 0xffff    ; segment length, bits 0-15
    dw 0x0       ; segment base, bits 0-15
    db 0x0       ; segment base, bits 16-23
    db 10011010b ; flags (8 bits)
    db 11001111b ; flags (4 bits) + segment length, bits 16-19
    db 0x0       ; segment base, bits 24-31

; GDT for data segment. base and length identical to code segment
; some flags changed, again, refer to os-dev.pdf
gdt_data:
    dw 0xffff
    dw 0x0
    db 0x0
    db 10010010b
    db 11001111b
    db 0x0
```

```
gdt_end:


; GDT descriptor
gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; size (16 bit), always one less of its true size
    dd gdt_start ; address (32 bit)


; define some constants for later use
CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start
```

kernel_entry.asm

```
global _start;
[bits 32]


_start:
    [extern start_kernel] ; Define calling point. Must have same name as kernel.c 'main' function
    call start_kernel ; Calls the C function. The linker will know where it is placed in memory
    jmp $
```

mbr.asm

```
[org 0x7c00]
KERNEL_OFFSET equ 0x1000 ; The same one we used when linking the kernel


mov [BOOT_DRIVE], dl ; Remember that the BIOS sets us the boot drive in 'dl' on boot
```

```
mov bp, 0x9000
mov sp, bp

mov bx, MSG_16BIT_MODE
call print16
call print16_nl

call load_kernel ; read the kernel from disk
call switch_to_32bit ; disable interrupts, load GDT,  etc. Finally jumps to 'BEGIN_PM'
jmp $ ; Never executed

%include "boot/print-16bit.asm"
%include "boot/print-32bit.asm"
%include "boot/disk.asm"
%include "boot/gdt.asm"
%include "boot/switch-to-32bit.asm"

[bits 16]
load_kernel:
    mov bx, MSG_LOAD_KERNEL
    call print16
    call print16_nl

    mov bx, KERNEL_OFFSET ; Read from disk and store in 0x1000
    mov dh, 31
    mov dl, [BOOT_DRIVE]
    call disk_load
    ret
```

```
[bits 32]
BEGIN_32BIT:
    mov ebx, MSG_32BIT_MODE
    call print32
    call KERNEL_OFFSET ; Give control to the kernel
    jmp $ ; Stay here when the kernel returns control to us (if ever)

BOOT_DRIVE db 0 ; It is a good idea to store it in memory because 'dl' may get overwritten
MSG_16BIT_MODE db "Started in 16-bit Real Mode", 0
MSG_32BIT_MODE db "Landed in 32-bit Protected Mode", 0
MSG_LOAD_KERNEL db "Loading kernel into memory", 0

; padding
times 510 - ($-$$) db 0
dw 0xaa55
```

print-16bit.asm

```
print16:
    pusha

; strings will be terminated by 0 byte in memory
print16_loop:
    mov al, [bx] ; 'bx' is the base address for the string
    cmp al, 0
    je print16_done

    mov ah, 0x0e ; tty
```

```
    int 0x10 ; 'al' already contains the char

    ; increment pointer and do next loop
    add bx, 1
    jmp print16_loop

print16_done:
    popa
    ret

print16_nl:
    pusha

    mov ah, 0x0e
    mov al, 0x0a ; newline char
    int 0x10
    mov al, 0x0d ; carriage return
    int 0x10

    popa
    ret

print16_cls:
    pusha

    mov ah, 0x00
    mov al, 0x03  ; text mode 80x25 16 colours
    int 0x10

    popa
```

```
    ret


; receiving the data in 'dx'
; For the examples we'll assume that we're called with dx=0x1234
print16_hex:
    pusha

    mov cx, 0 ; our index variable

; Strategy: get the last char of 'dx', then convert to ASCII
; Numeric ASCII values: '0' (ASCII 0x30) to '9' (0x39), so just add 0x30 to byte N.
; For alphabetic characters A-F: 'A' (ASCII 0x41) to 'F' (0x46) we'll add 0x40
; Then, move the ASCII byte to the correct position on the resulting string
print16_hex_loop:
    cmp cx, 4 ; loop 4 times
    je print16_hex_end

    ; 1. convert last char of 'dx' to ascii
    mov ax, dx ; we will use 'ax' as our working register
    and ax, 0x000f ; 0x1234 -> 0x0004 by masking first three to zeros
    add al, 0x30 ; add 0x30 to N to convert it to ASCII "N"
    cmp al, 0x39 ; if > 9, add extra 8 to represent 'A' to 'F'
    jle print16_hex_step2
    add al, 7 ; 'A' is ASCII 65 instead of 58, so 65-58=7

print16_hex_step2:
    ; 2. get the correct position of the string to place our ASCII char
    ; bx <- base address + string length - index of char
```

```
    mov bx, PRINT16_HEX_OUT + 5 ; base + length
    sub bx, cx  ; our index variable
    mov [bx], al ; copy the ASCII char on 'al' to the position pointed
by 'bx'
    ror dx, 4 ; 0x1234 -> 0x4123 -> 0x3412 -> 0x2341 -> 0x1234

    ; increment index and loop
    add cx, 1
    jmp print16_hex_loop

print16_hex_end:
    ; prepare the parameter and call the function
    ; remember that print receives parameters in 'bx'
    mov bx, PRINT16_HEX_OUT
    call print16

    popa
    ret

PRINT16_HEX_OUT:
    db '0x0000',0 ; reserve memory for our new string
```

print-32bit.asm

```
[bits 32] ; using 32-bit protected mode

; this is how constants are defined
VIDEO_MEMORY equ 0xb8000
WHITE_ON_BLACK equ 0x0f ; the color byte for each character
```

```
print32:
    pusha
    mov edx, VIDEO_MEMORY

print32_loop:
    mov al, [ebx] ; [ebx] is the address of our character
    mov ah, WHITE_ON_BLACK

    cmp al, 0 ; check if end of string
    je print32_done

    mov [edx], ax ; store character + attribute in video memory
    add ebx, 1 ; next char
    add edx, 2 ; next video memory position

    jmp print32_loop

print32_done:
    popa
    ret
```

switch-to-32bit.asm

```
[bits 16]
switch_to_32bit:
    cli ; 1. disable interrupts
    lgdt [gdt_descriptor] ; 2. load the GDT descriptor
    mov eax, cr0
    or eax, 0x1 ; 3. set 32-bit mode bit in cr0
```

```
    mov cr0, eax
    jmp CODE_SEG:init_32bit ; 4. far jump by using a different segment

[bits 32]
init_32bit: ; we are now using 32-bit instructions
    mov ax, DATA_SEG ; 5. update the segment registers
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    mov ebp, 0x90000 ; 6. update the stack right at the top of the free space
    mov esp, ebp

    call BEGIN_32BIT ; 7. Call a well-known label with useful code
```

idt.c

```
#include "idt.h"
#include "../kernel/util.h"

idt_gate_t idt[IDT_ENTRIES];
idt_register_t idt_reg;

void set_idt_gate(int n, uint32_t handler) {
    idt[n].low_offset = low_16(handler);
    idt[n].sel = KERNEL_CS;
    idt[n].always0 = 0;
```

```
    idt[n].flags = 0x8E;
    idt[n].high_offset = high_16(handler);
}

void load_idt() {
    idt_reg.base = (uint32_t) &idt;
    idt_reg.limit = IDT_ENTRIES * sizeof(idt_gate_t) - 1;
    /* Don't make the mistake of loading &idt -- always load &idt_reg
*/
    asm volatile("lidt (%0)" : : "r" (&idt_reg));
}
```

idt.h

```
#pragma once

#include <stdint.h>

/* Segment selectors */
#define KERNEL_CS 0x08

/* How every interrupt gate (handler) is defined */
typedef struct {
    uint16_t low_offset; /* Lower 16 bits of handler function address
*/
    uint16_t sel; /* Kernel segment selector */
    uint8_t always0;
    /* First byte
     * Bit 7: "Interrupt is present"
     * Bits 6-5: Privilege level of caller (0=kernel..3=user)
```

```
     * Bit 4: Set to 0 for interrupt gates
     * Bits 3-0: bits 1110 = decimal 14 = "32 bit interrupt gate" */
    uint8_t flags;
    uint16_t high_offset; /* Higher 16 bits of handler function address
*/
} __attribute__((packed)) idt_gate_t;

/* A pointer to the array of interrupt handlers.
     * Assembly instruction 'lidt' will read it */
typedef struct {
    uint16_t limit;
    uint32_t base;
} __attribute__((packed)) idt_register_t;

#define IDT_ENTRIES 256

void set_idt_gate(int n, uint32_t handler);

void load_idt();
```

interrupt.asm

```
; Defined in isr.c
[extern isr_handler]
[extern irq_handler]

; Common ISR code
isr_common_stub:
    ; 1. Save CPU state
    pusha ; Pushes edi,esi,ebp,esp,ebx,edx,ecx,eax
```

```
    mov ax, ds ; Lower 16-bits of eax = ds.
    push eax ; save the data segment descriptor
    mov ax, 0x10  ; kernel data segment descriptor
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    ; 2. Call C handler
    push esp ; push registers_t *r pointer
    call isr_handler
    pop eax ; clear pointer afterwards

    ; 3. Restore state
    pop eax
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    popa
    add esp, 8 ; Cleans up the pushed error code and pushed ISR number
    iret ; pops 5 things at once: CS, EIP, EFLAGS, SS, and ESP

; Common IRQ code. Identical to ISR code except for the 'call'
; and the 'pop ebx'
irq_common_stub:
    ; 1. Save CPU state
    pusha
    mov ax, ds
    push eax
```

```
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    ; 2. Call C handler
    push esp
    call irq_handler ; Different than the ISR code
    pop ebx  ; Different than the ISR code

    ; 3. Restore state
    pop ebx
    mov ds, bx
    mov es, bx
    mov fs, bx
    mov gs, bx
    popa
    add esp, 8
    iret

; We don't get information about which interrupt was caller
; when the handler is run, so we will need to have a different handler
; for every interrupt.
; Furthermore, some interrupts push an error code onto the stack but others
; don't, so we will push a dummy error code for those which don't, so that
; we have a consistent stack for all of them.
```

```
; First make the ISRs global
global isr0
global isr1
global isr2
global isr3
global isr4
global isr5
global isr6
global isr7
global isr8
global isr9
global isr10
global isr11
global isr12
global isr13
global isr14
global isr15
global isr16
global isr17
global isr18
global isr19
global isr20
global isr21
global isr22
global isr23
global isr24
global isr25
global isr26
global isr27
global isr28
```

```
global isr29
global isr30
global isr31

; 0: Divide By Zero Exception
isr0:
    push byte 0
    push byte 0
    jmp isr_common_stub

; 1: Debug Exception
isr1:
    push byte 0
    push byte 1
    jmp isr_common_stub

; 2: Non Maskable Interrupt Exception
isr2:
    push byte 0
    push byte 2
    jmp isr_common_stub

; 3: Int 3 Exception
isr3:
    push byte 0
    push byte 3
    jmp isr_common_stub

; 4: INTO Exception
isr4:
```

```
    push byte 0
    push byte 4
    jmp isr_common_stub

; 5: Out of Bounds Exception
isr5:
    push byte 0
    push byte 5
    jmp isr_common_stub

; 6: Invalid Opcode Exception
isr6:
    push byte 0
    push byte 6
    jmp isr_common_stub

; 7: Coprocessor Not Available Exception
isr7:
    push byte 0
    push byte 7
    jmp isr_common_stub

; 8: Double Fault Exception (With Error Code!)
isr8:
    push byte 8
    jmp isr_common_stub

; 9: Coprocessor Segment Overrun Exception
isr9:
    push byte 0
```

```
    push byte 9
    jmp isr_common_stub

; 10: Bad TSS Exception (With Error Code!)
isr10:
    push byte 10
    jmp isr_common_stub

; 11: Segment Not Present Exception (With Error Code!)
isr11:
    push byte 11
    jmp isr_common_stub

; 12: Stack Fault Exception (With Error Code!)
isr12:
    push byte 12
    jmp isr_common_stub

; 13: General Protection Fault Exception (With Error Code!)
isr13:
    push byte 13
    jmp isr_common_stub

; 14: Page Fault Exception (With Error Code!)
isr14:
    push byte 14
    jmp isr_common_stub

; 15: Reserved Exception
isr15:
```

```
    push byte 0
    push byte 15
    jmp isr_common_stub

; 16: Floating Point Exception
isr16:
    push byte 0
    push byte 16
    jmp isr_common_stub

; 17: Alignment Check Exception
isr17:
    push byte 0
    push byte 17
    jmp isr_common_stub

; 18: Machine Check Exception
isr18:
    push byte 0
    push byte 18
    jmp isr_common_stub

; 19: Reserved
isr19:
    push byte 0
    push byte 19
    jmp isr_common_stub

; 20: Reserved
isr20:
```

```
    push byte 0
    push byte 20
    jmp isr_common_stub

; 21: Reserved
isr21:
    push byte 0
    push byte 21
    jmp isr_common_stub

; 22: Reserved
isr22:
    push byte 0
    push byte 22
    jmp isr_common_stub

; 23: Reserved
isr23:
    push byte 0
    push byte 23
    jmp isr_common_stub

; 24: Reserved
isr24:
    push byte 0
    push byte 24
    jmp isr_common_stub

; 25: Reserved
isr25:
```

```
    push byte 0
    push byte 25
    jmp isr_common_stub

; 26: Reserved
isr26:
    push byte 0
    push byte 26
    jmp isr_common_stub

; 27: Reserved
isr27:
    push byte 0
    push byte 27
    jmp isr_common_stub

; 28: Reserved
isr28:
    push byte 0
    push byte 28
    jmp isr_common_stub

; 29: Reserved
isr29:
    push byte 0
    push byte 29
    jmp isr_common_stub

; 30: Reserved
isr30:
```

```
    push byte 0
    push byte 30
    jmp isr_common_stub

; 31: Reserved
isr31:
    push byte 0
    push byte 31
    jmp isr_common_stub

; IRQs
global irq0
global irq1
global irq2
global irq3
global irq4
global irq5
global irq6
global irq7
global irq8
global irq9
global irq10
global irq11
global irq12
global irq13
global irq14
global irq15

; IRQ handlers
irq0:
```

```
    push byte 0
    push byte 32
    jmp irq_common_stub

irq1:
    push byte 1
    push byte 33
    jmp irq_common_stub

irq2:
    push byte 2
    push byte 34
    jmp irq_common_stub

irq3:
    push byte 3
    push byte 35
    jmp irq_common_stub

irq4:
    push byte 4
    push byte 36
    jmp irq_common_stub

irq5:
    push byte 5
    push byte 37
    jmp irq_common_stub

irq6:
```

```
    push byte 6
    push byte 38
    jmp irq_common_stub

irq7:
    push byte 7
    push byte 39
    jmp irq_common_stub

irq8:
    push byte 8
    push byte 40
    jmp irq_common_stub

irq9:
    push byte 9
    push byte 41
    jmp irq_common_stub

irq10:
    push byte 10
    push byte 42
    jmp irq_common_stub

irq11:
    push byte 11
    push byte 43
    jmp irq_common_stub

irq12:
```

```
    push byte 12
    push byte 44
    jmp irq_common_stub

irq13:
    push byte 13
    push byte 45
    jmp irq_common_stub

irq14:
    push byte 14
    push byte 46
    jmp irq_common_stub

irq15:
    push byte 15
    push byte 47
    jmp irq_common_stub
```

isr.c

```
#include "isr.h"
#include "idt.h"
#include "../drivers/display.h"
#include "../drivers/ports.h"
#include "../kernel/util.h"

isr_t interrupt_handlers[256];
```

```
/* Can't do this with a loop because we need the address
 * of the function names */
void isr_install() {
    set_idt_gate(0, (uint32_t) isr0);
    set_idt_gate(1, (uint32_t) isr1);
    set_idt_gate(2, (uint32_t) isr2);
    set_idt_gate(3, (uint32_t) isr3);
    set_idt_gate(4, (uint32_t) isr4);
    set_idt_gate(5, (uint32_t) isr5);
    set_idt_gate(6, (uint32_t) isr6);
    set_idt_gate(7, (uint32_t) isr7);
    set_idt_gate(8, (uint32_t) isr8);
    set_idt_gate(9, (uint32_t) isr9);
    set_idt_gate(10, (uint32_t) isr10);
    set_idt_gate(11, (uint32_t) isr11);
    set_idt_gate(12, (uint32_t) isr12);
    set_idt_gate(13, (uint32_t) isr13);
    set_idt_gate(14, (uint32_t) isr14);
    set_idt_gate(15, (uint32_t) isr15);
    set_idt_gate(16, (uint32_t) isr16);
    set_idt_gate(17, (uint32_t) isr17);
    set_idt_gate(18, (uint32_t) isr18);
    set_idt_gate(19, (uint32_t) isr19);
    set_idt_gate(20, (uint32_t) isr20);
    set_idt_gate(21, (uint32_t) isr21);
    set_idt_gate(22, (uint32_t) isr22);
    set_idt_gate(23, (uint32_t) isr23);
    set_idt_gate(24, (uint32_t) isr24);
    set_idt_gate(25, (uint32_t) isr25);
    set_idt_gate(26, (uint32_t) isr26);
```

```
    set_idt_gate(27, (uint32_t) isr27);
    set_idt_gate(28, (uint32_t) isr28);
    set_idt_gate(29, (uint32_t) isr29);
    set_idt_gate(30, (uint32_t) isr30);
    set_idt_gate(31, (uint32_t) isr31);

    // Remap the PIC
    port_byte_out(0x20, 0x11);
    port_byte_out(0xA0, 0x11);
    port_byte_out(0x21, 0x20);
    port_byte_out(0xA1, 0x28);
    port_byte_out(0x21, 0x04);
    port_byte_out(0xA1, 0x02);
    port_byte_out(0x21, 0x01);
    port_byte_out(0xA1, 0x01);
    port_byte_out(0x21, 0x0);
    port_byte_out(0xA1, 0x0);

    // Install the IRQs
    set_idt_gate(32, (uint32_t)irq0);
    set_idt_gate(33, (uint32_t)irq1);
    set_idt_gate(34, (uint32_t)irq2);
    set_idt_gate(35, (uint32_t)irq3);
    set_idt_gate(36, (uint32_t)irq4);
    set_idt_gate(37, (uint32_t)irq5);
    set_idt_gate(38, (uint32_t)irq6);
    set_idt_gate(39, (uint32_t)irq7);
    set_idt_gate(40, (uint32_t)irq8);
    set_idt_gate(41, (uint32_t)irq9);
    set_idt_gate(42, (uint32_t)irq10);
```

```
    set_idt_gate(43, (uint32_t)irq11);
    set_idt_gate(44, (uint32_t)irq12);
    set_idt_gate(45, (uint32_t)irq13);
    set_idt_gate(46, (uint32_t)irq14);
    set_idt_gate(47, (uint32_t)irq15);

    load_idt(); // Load with ASM
}

/* To print the message which defines every exception */
char *exception_messages[] = {
        "Division By Zero",
        "Debug",
        "Non Maskable Interrupt",
        "Breakpoint",
        "Into Detected Overflow",
        "Out of Bounds",
        "Invalid Opcode",
        "No Coprocessor",

        "Double Fault",
        "Coprocessor Segment Overrun",
        "Bad TSS",
        "Segment Not Present",
        "Stack Fault",
        "General Protection Fault",
        "Page Fault",
        "Unknown Interrupt",

        "Coprocessor Fault",
```

```
        "Alignment Check",
        "Machine Check",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",

        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved"
};

void isr_handler(registers_t *r) {
    print_string("received interrupt: ");
    char s[3];
    int_to_string(r->int_no, s);
    print_string(s);
    print_nl();
    print_string(exception_messages[r->int_no]);
    print_nl();
}

void register_interrupt_handler(uint8_t n, isr_t handler) {
    interrupt_handlers[n] = handler;
```

```
}

void irq_handler(registers_t *r) {
    /* Handle the interrupt in a more modular way */
    if (interrupt_handlers[r->int_no] != 0) {
        isr_t handler = interrupt_handlers[r->int_no];
        handler(r);
    }

    // EOI
    if (r->int_no >= 40) {
        port_byte_out(0xA0, 0x20); /* follower */
    }
    port_byte_out(0x20, 0x20); /* leader */
}
```

isr.h

```
#pragma once

#include <stdint.h>

/* ISRs reserved for CPU exceptions */
extern void isr0();

extern void isr1();

extern void isr2();
```

```
extern void isr3();

extern void isr4();

extern void isr5();

extern void isr6();

extern void isr7();

extern void isr8();

extern void isr9();

extern void isr10();

extern void isr11();

extern void isr12();

extern void isr13();

extern void isr14();

extern void isr15();

extern void isr16();

extern void isr17();
```

```
extern void isr18();

extern void isr19();

extern void isr20();

extern void isr21();

extern void isr22();

extern void isr23();

extern void isr24();

extern void isr25();

extern void isr26();

extern void isr27();

extern void isr28();

extern void isr29();

extern void isr30();

extern void isr31();

/* IRQ definitions */
extern void irq0();
```

```
extern void irq1();

extern void irq2();

extern void irq3();

extern void irq4();

extern void irq5();

extern void irq6();

extern void irq7();

extern void irq8();

extern void irq9();

extern void irq10();

extern void irq11();

extern void irq12();

extern void irq13();

extern void irq14();

extern void irq15();
```

```
#define IRQ0 32
#define IRQ1 33
#define IRQ2 34
#define IRQ3 35
#define IRQ4 36
#define IRQ5 37
#define IRQ6 38
#define IRQ7 39
#define IRQ8 40
#define IRQ9 41
#define IRQ10 42
#define IRQ11 43
#define IRQ12 44
#define IRQ13 45
#define IRQ14 46
#define IRQ15 47

/* Struct which aggregates many registers.
 * It matches exactly the pushes on interrupt.asm. From the bottom:
 * - Pushed by the processor automatically
 * - `push byte`s on the isr-specific code: error code, then int number
 * - All the registers by pusha
 * - `push eax` whose lower 16-bits contain DS
 */
typedef struct {
    uint32_t ds; /* Data segment selector */
    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax; /* Pushed by
pusha. */
```

```
    uint32_t int_no, err_code; /* Interrupt number and error code (if
applicable) */
    uint32_t eip, cs, eflags, useresp, ss; /* Pushed by the processor
automatically */
} registers_t;


void isr_install();


void isr_handler(registers_t *r);


typedef void (*isr_t)(registers_t *);


void register_interrupt_handler(uint8_t n, isr_t handler);
```

timer.c

```
#include "timer.h"
#include "../drivers/display.h"
#include "../drivers/ports.h"
#include "../kernel/util.h"
#include "isr.h"


uint32_t tick = 0;


static void timer_callback(registers_t *regs) {
    tick++;
    print_string("Tick: ");


    char tick_ascii[256];
    int_to_string(tick, tick_ascii);
```

```
    print_string(tick_ascii);
    print_nl();
}

void init_timer(uint32_t freq) {
    /* Install the function we just wrote */
    register_interrupt_handler(IRQ0, timer_callback);

    /* Get the PIT value: hardware clock at 1193180 Hz */
    uint32_t divisor = 1193180 / freq;
    uint8_t low  = (uint8_t)(divisor & 0xFF);
    uint8_t high = (uint8_t)( (divisor >> 8) & 0xFF);
    /* Send the command */
    port_byte_out(0x43, 0x36); /* Command port */
    port_byte_out(0x40, low);
    port_byte_out(0x40, high);
}
```

timer.h,

```
#pragma once

#include "../kernel/util.h"

void init_timer(uint32_t freq);
```

display.c

```
#include "display.h"
#include "ports.h"
```

```c
#include <stdint.h>
#include "../kernel/mem.h"
#include "../kernel/util.h"

void set_cursor(int offset) {
    offset /= 2;
    port_byte_out(REG_SCREEN_CTRL, 14);
    port_byte_out(REG_SCREEN_DATA, (unsigned char) (offset >> 8));
    port_byte_out(REG_SCREEN_CTRL, 15);
    port_byte_out(REG_SCREEN_DATA, (unsigned char) (offset & 0xff));
}

int get_cursor() {
    port_byte_out(REG_SCREEN_CTRL, 14);
    int offset = port_byte_in(REG_SCREEN_DATA) << 8; /* High byte: << 8
*/
    port_byte_out(REG_SCREEN_CTRL, 15);
    offset += port_byte_in(REG_SCREEN_DATA);
    return offset * 2;
}

int get_offset(int col, int row) {
    return 2 * (row * MAX_COLS + col);
}

int get_row_from_offset(int offset) {
    return offset / (2 * MAX_COLS);
}

int move_offset_to_new_line(int offset) {
```

```
    return get_offset(0, get_row_from_offset(offset) + 1);
}

void set_char_at_video_memory(char character, int offset) {
    uint8_t *vidmem = (uint8_t *) VIDEO_ADDRESS;
    vidmem[offset] = character;
    vidmem[offset + 1] = WHITE_ON_BLACK;
}

int scroll_ln(int offset) {
    memory_copy(
            (uint8_t * )(get_offset(0, 1) + VIDEO_ADDRESS),
            (uint8_t * )(get_offset(0, 0) + VIDEO_ADDRESS),
            MAX_COLS * (MAX_ROWS - 1) * 2
    );

    for (int col = 0; col < MAX_COLS; col++) {
        set_char_at_video_memory(' ', get_offset(col, MAX_ROWS - 1));
    }

    return offset - 2 * MAX_COLS;
}

/*
 * TODO:
 * - handle illegal offset (print error message somewhere)
 */
void print_string(char *string) {
    int offset = get_cursor();
    int i = 0;
```

```
    while (string[i] != 0) {
        if (offset >= MAX_ROWS * MAX_COLS * 2) {
            offset = scroll_ln(offset);
        }
        if (string[i] == '\n') {
            offset = move_offset_to_new_line(offset);
        } else {
            set_char_at_video_memory(string[i], offset);
            offset += 2;
        }
        i++;
    }
    set_cursor(offset);
}

void print_nl() {
    int newOffset = move_offset_to_new_line(get_cursor());
    if (newOffset >= MAX_ROWS * MAX_COLS * 2) {
        newOffset = scroll_ln(newOffset);
    }
    set_cursor(newOffset);
}

void clear_screen() {
    int screen_size = MAX_COLS * MAX_ROWS;
    for (int i = 0; i < screen_size; ++i) {
        set_char_at_video_memory(' ', i * 2);
    }
    set_cursor(get_offset(0, 0));
}
```

```
void print_backspace() {
    int newCursor = get_cursor() - 2;
    set_char_at_video_memory(' ', newCursor);
    set_cursor(newCursor);
}
```

display.h

```
#pragma once

#define VIDEO_ADDRESS 0xb8000
#define MAX_ROWS 25
#define MAX_COLS 80
#define WHITE_ON_BLACK 0x0f

/* Screen i/o ports */
#define REG_SCREEN_CTRL 0x3d4
#define REG_SCREEN_DATA 0x3d5

/* Public kernel API */
void print_string(char *string);

void print_nl();

void clear_screen();

int scroll_ln(int offset);
```

```
void print_backspace();
```

keyboard.c

```
#include <stdbool.h>
#include "keyboard.h"
#include "ports.h"
#include "../cpu/isr.h"
#include "display.h"
#include "../kernel/util.h"
#include "../kernel/kernel.h"

#define BACKSPACE 0x0E
#define ENTER 0x1C

static char key_buffer[256];

#define SC_MAX 57

const char *sc_name[] = {"ERROR", "Esc", "1", "2", "3", "4", "5", "6",
                         "7", "8", "9", "0", "-", "=", "Backspace",
"Tab", "Q", "W", "E",
                         "R", "T", "Y", "U", "I", "O", "P", "[", "]",
"Enter", "Lctrl",
                         "A", "S", "D", "F", "G", "H", "J", "K", "L",
";", "'", "`",
                         "LShift", "\\", "Z", "X", "C", "V", "B", "N",
"M", ",", ".",
                         "/", "RShift", "Keypad *", "LAlt", "Spacebar"};
const char sc_ascii[] = {'?', '?', '1', '2', '3', '4', '5', '6',
                         '7', '8', '9', '0', '-', '=', '?', '?', 'Q',
'W', 'E', 'R', 'T', 'Y',
```

```
                        'U', 'I', 'O', 'P', '[', ']', '?', '?', 'A',
'S', 'D', 'F', 'G',
                        'H', 'J', 'K', 'L', ';', '\'', '`', '?', '\\',
'Z', 'X', 'C', 'V',
                        'B', 'N', 'M', ',', '.', '/', '?', '?', '?', '
'};

static void keyboard_callback(registers_t *regs) {
    uint8_t scancode = port_byte_in(0x60);
    if (scancode > SC_MAX) return;
    if (scancode == BACKSPACE) {
        if (backspace(key_buffer)) {
            print_backspace();
        }
    } else if (scancode == ENTER) {
        print_nl();
        execute_command(key_buffer);
        key_buffer[0] = '\0';
    } else {
        char letter = sc_ascii[(int) scancode];
        append(key_buffer, letter);
        char str[2] = {letter, '\0'};
        print_string(str);
    }
}

void init_keyboard() {
    register_interrupt_handler(IRQ1, keyboard_callback);
}
```

keyboard.h

```
#pragma once


void init_keyboard();
```

ports.c

```
#include <stdint.h>


/**
 * Read a byte from the specified port
 */
unsigned char port_byte_in(uint16_t port) {
    unsigned char result;
    /* Inline assembler syntax
     * !! Notice how the source and destination registers are switched from NASM !!
     *
     * '"=a" (result)'; set '=' the C variable '(result)' to the value of register e'a'x
     * '"d" (port)': map the C variable '(port)' into e'd'x register
     *
     * Inputs and outputs are separated by colons
     */
    asm("in %%dx, %%al" : "=a" (result) : "d" (port));
    return result;
}


void port_byte_out(uint16_t port, uint8_t data) {
```

```
    /* Notice how here both registers are mapped to C variables and
     * nothing is returned, thus, no equals '=' in the asm syntax
     * However we see a comma since there are two variables in the
input area
     * and none in the 'return' area
     */
    asm("out %%al, %%dx" : : "a" (data), "d" (port));
}

unsigned short port_word_in(uint16_t port) {
    unsigned short result;
    asm("in %%dx, %%ax" : "=a" (result) : "d" (port));
    return result;
}

void port_word_out(uint16_t port, uint16_t data) {
    asm("out %%ax, %%dx" : : "a" (data), "d" (port));
}
```

ports.h

```
#pragma once

#include <stdint.h>

unsigned char port_byte_in(uint16_t port);

void port_byte_out(uint16_t port, uint8_t data);

unsigned short port_word_in(uint16_t port);
```

```
void port_word_out(uint16_t port, uint16_t data);
```

kernel.c

```
#include "../cpu/idt.h"
#include "../cpu/isr.h"
#include "../cpu/timer.h"
#include "../drivers/display.h"
#include "../drivers/keyboard.h"

#include "util.h"
#include "mem.h"

void* alloc(int n) {
    int *ptr = (int *) mem_alloc(n * sizeof(int));
    if (ptr == NULL_POINTER) {
        print_string("Memory not allocated.\n");
    } else {
        // Get the elements of the array
        for (int i = 0; i < n; ++i) {
//            ptr[i] = i + 1; // shorthand for *(ptr + i)
        }

        for (int i = 0; i < n; ++i) {
//            char str[256];
//            int_to_string(ptr[i], str);
//            print_string(str);
        }
```

```
//        print_nl();
    }
    return ptr;
}

void start_kernel() {
    clear_screen();
    print_string("Installing interrupt service routines (ISRs).\n");
    isr_install();

    print_string("Enabling external interrupts.\n");
    asm volatile("sti");

    print_string("Initializing keyboard (IRQ 1).\n");
    init_keyboard();

    print_string("Initializing dynamic memory.\n");
    init_dynamic_mem();

    clear_screen();

    print_string("init_dynamic_mem()\n");
    print_dynamic_node_size();
    print_dynamic_mem();
    print_nl();

    int *ptr1 = alloc(5);
    print_string("int *ptr1 = alloc(5)\n");
    print_dynamic_mem();
    print_nl();
```

```
    int *ptr2 = alloc(10);
    print_string("int *ptr2 = alloc(10)\n");
    print_dynamic_mem();
    print_nl();

    mem_free(ptr1);
    print_string("mem_free(ptr1)\n");
    print_dynamic_mem();
    print_nl();

    int *ptr3 = alloc(2);
    print_string("int *ptr3 = alloc(2)\n");
    print_dynamic_mem();
    print_nl();

    mem_free(ptr2);
    print_string("mem_free(ptr2)\n");
    print_dynamic_mem();
    print_nl();

    mem_free(ptr3);
    print_string("mem_free(ptr3)\n");
    print_dynamic_mem();
    print_nl();

    print_string("> ");
}

void execute_command(char *input) {
```

```
    if (compare_string(input, "EXIT") == 0) {
        print_string("Stopping the CPU. Bye!\n");
        asm volatile("hlt");
    }
    else if (compare_string(input, "") == 0) {
        print_string("\n> ");
    }
    else {
        print_string("Unknown command: ");
        print_string(input);
        print_string("\n> ");
    }
}
```

kernel.h

```
#pragma once

void execute_command(char *input);
```

mem.c

```
#include <stdbool.h>
#include <stdint.h>
#include "mem.h"
#include "../drivers/display.h"
#include "util.h"

// http://www.sunshine2k.de/articles/coding/cmemalloc/cmemory.html#ch33
```

```
void memory_copy(uint8_t *source, uint8_t *dest, uint32_t nbytes) {
    int i;
    for (i = 0; i < nbytes; i++) {
        *(dest + i) = *(source + i);
    }
}

/*
 * The following code is based on code licensed under MIT licence
 * and thus also licensed under MIT license I guess?
 * For further details, see http://www.sunshine2k.de/license.html.
 */
#define DYNAMIC_MEM_TOTAL_SIZE 4*1024
#define DYNAMIC_MEM_NODE_SIZE sizeof(dynamic_mem_node_t)

typedef struct dynamic_mem_node {
    uint32_t size;
    bool used;
    struct dynamic_mem_node *next;
    struct dynamic_mem_node *prev;
} dynamic_mem_node_t;

static uint8_t dynamic_mem_area[DYNAMIC_MEM_TOTAL_SIZE];
static dynamic_mem_node_t *dynamic_mem_start;

void init_dynamic_mem() {
    dynamic_mem_start = (dynamic_mem_node_t *) dynamic_mem_area;
    dynamic_mem_start->size = DYNAMIC_MEM_TOTAL_SIZE -
DYNAMIC_MEM_NODE_SIZE;
    dynamic_mem_start->next = NULL_POINTER;
```

```
    dynamic_mem_start->prev = NULL_POINTER;
}

void print_dynamic_node_size() {
    char node_size_string[256];
    int_to_string(DYNAMIC_MEM_NODE_SIZE, node_size_string);
    print_string("DYNAMIC_MEM_NODE_SIZE = ");
    print_string(node_size_string);
    print_nl();
}

void print_dynamic_mem_node(dynamic_mem_node_t *node) {
    char size_string[256];
    int_to_string(node->size, size_string);
    print_string("{size = ");
    print_string(size_string);
    char used_string[256];
    int_to_string(node->used, used_string);
    print_string("; used = ");
    print_string(used_string);
    print_string("}; ");
}

void print_dynamic_mem() {
    dynamic_mem_node_t *current = dynamic_mem_start;
    print_string("[");
    while (current != NULL_POINTER) {
        print_dynamic_mem_node(current);
        current = current->next;
    }
```

```
    print_string("]\n");
}


void *find_best_mem_block(dynamic_mem_node_t *dynamic_mem, size_t size)
{
    // initialize the result pointer with NULL and an invalid block
size
    dynamic_mem_node_t *best_mem_block = (dynamic_mem_node_t *)
NULL_POINTER;
    uint32_t best_mem_block_size = DYNAMIC_MEM_TOTAL_SIZE + 1;


    // start looking for the best (smallest unused) block at the
beginning
    dynamic_mem_node_t *current_mem_block = dynamic_mem;
    while (current_mem_block) {
        // check if block can be used and is smaller than current best
        if ((!current_mem_block->used) &&
            (current_mem_block->size >= (size + DYNAMIC_MEM_NODE_SIZE))
&&
            (current_mem_block->size <= best_mem_block_size)) {
            // update best block
            best_mem_block = current_mem_block;
            best_mem_block_size = current_mem_block->size;
        }


        // move to next block
        current_mem_block = current_mem_block->next;
    }
    return best_mem_block;
}
```

```
void *mem_alloc(size_t size) {
    dynamic_mem_node_t *best_mem_block =
            (dynamic_mem_node_t *)
find_best_mem_block(dynamic_mem_start, size);

    // check if we actually found a matching (free, large enough) block
    if (best_mem_block != NULL_POINTER) {
        // subtract newly allocated memory (incl. size of the mem node)
from selected block
        best_mem_block->size = best_mem_block->size - size -
DYNAMIC_MEM_NODE_SIZE;

        // create new mem node after selected node, effectively
splitting the memory region
        dynamic_mem_node_t *mem_node_allocate = (dynamic_mem_node_t *)
(((uint8_t *) best_mem_block) +
                                                                DY
NAMIC_MEM_NODE_SIZE +
                                                                be
st_mem_block->size);
        mem_node_allocate->size = size;
        mem_node_allocate->used = true;
        mem_node_allocate->next = best_mem_block->next;
        mem_node_allocate->prev = best_mem_block;

        // reconnect the doubly linked list
        if (best_mem_block->next != NULL_POINTER) {
            best_mem_block->next->prev = mem_node_allocate;
        }
        best_mem_block->next = mem_node_allocate;
```

```
        // return pointer to newly allocated memory (right after the
new list node)
        return (void *) ((uint8_t *) mem_node_allocate +
DYNAMIC_MEM_NODE_SIZE);
    }


    return NULL_POINTER;
}


void *merge_next_node_into_current(dynamic_mem_node_t
*current_mem_node) {
    dynamic_mem_node_t *next_mem_node = current_mem_node->next;
    if (next_mem_node != NULL_POINTER && !next_mem_node->used) {
        // add size of next block to current block
        current_mem_node->size += current_mem_node->next->size;
        current_mem_node->size += DYNAMIC_MEM_NODE_SIZE;


        // remove next block from list
        current_mem_node->next = current_mem_node->next->next;
        if (current_mem_node->next != NULL_POINTER) {
            current_mem_node->next->prev = current_mem_node;
        }
    }
    return current_mem_node;
}


void *merge_current_node_into_previous(dynamic_mem_node_t
*current_mem_node) {
    dynamic_mem_node_t *prev_mem_node = current_mem_node->prev;
    if (prev_mem_node != NULL_POINTER && !prev_mem_node->used) {
        // add size of previous block to current block
```

```
        prev_mem_node->size += current_mem_node->size;
        prev_mem_node->size += DYNAMIC_MEM_NODE_SIZE;

        // remove current node from list
        prev_mem_node->next = current_mem_node->next;
        if (current_mem_node->next != NULL_POINTER) {
            current_mem_node->next->prev = prev_mem_node;
        }
    }
}

void mem_free(void *p) {
    // move along, nothing to free here
    if (p == NULL_POINTER) {
        return;
    }

    // get mem node associated with pointer
    dynamic_mem_node_t *current_mem_node = (dynamic_mem_node_t *)
((uint8_t *) p - DYNAMIC_MEM_NODE_SIZE);

    // pointer we're trying to free was not dynamically allocated it
seems
    if (current_mem_node == NULL_POINTER) {
        return;
    }

    // mark block as unused
    current_mem_node->used = false;
```

```
    // merge unused blocks
    current_mem_node = merge_next_node_into_current(current_mem_node);
    merge_current_node_into_previous(current_mem_node);
}
```

mem.h

```
#pragma once

#include <stdint.h>
#include <stddef.h>

#define NULL_POINTER ((void*)0)

void memory_copy(uint8_t *source, uint8_t *dest, uint32_t nbytes);

void init_dynamic_mem();

void print_dynamic_node_size();

void print_dynamic_mem();

void *mem_alloc(size_t size);

void mem_free(void *p);
```

util.c

```
#include <stdint.h>
#include <stdbool.h>

```

```
int string_length(char s[]) {
    int i = 0;
    while (s[i] != '\0') ++i;
    return i;
}

void reverse(char s[]) {
    int c, i, j;
    for (i = 0, j = string_length(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

void int_to_string(int n, char str[]) {
    int i, sign;
    if ((sign = n) < 0) n = -n;
    i = 0;
    do {
        str[i++] = n % 10 + '0';
    } while ((n /= 10) > 0);

    if (sign < 0) str[i++] = '-';
    str[i] = '\0';

    reverse(str);
}

void append(char s[], char n) {
```

```
    int len = string_length(s);
    s[len] = n;
    s[len+1] = '\0';
}

bool backspace(char s[]) {
    int len = string_length(s);
    if (len > 0) {
        s[len - 1] = '\0';
        return true;
    } else {
        return false;
    }
}

/* K&R
 * Returns <0 if s1<s2, 0 if s1==s2, >0 if s1>s2 */
int compare_string(char s1[], char s2[]) {
    int i;
    for (i = 0; s1[i] == s2[i]; i++) {
        if (s1[i] == '\0') return 0;
    }
    return s1[i] - s2[i];
}
```

util.h

```
#pragma once
```

```c
#include <stdint.h>
#include <stdbool.h>

#define low_16(address) (uint16_t)((address) & 0xFFFF)
#define high_16(address) (uint16_t)(((address) >> 16) & 0xFFFF)

int string_length(char s[]);

void reverse(char s[]);

void int_to_string(int n, char str[]);

bool backspace(char s[]);

void append(char s[], char n);

int compare_string(char s1[], char s2[]);
```

Makefile

```
# $@ = target file
# $< = first dependency
# $^ = all dependencies

# detect all .o files based on their .c source
C_SOURCES = $(wildcard kernel/*.c drivers/*.c cpu/*.c)
HEADERS = $(wildcard kernel/*.h  drivers/*.h cpu/*.h)
OBJ_FILES = ${C_SOURCES:.c=.o cpu/interrupt.o}

CC ?= x86_64-elf-gcc
```

```
LD ?= x86_64-elf-ld


# First rule is the one executed when no parameters are fed to the
Makefile
all: run


# Notice how dependencies are built as needed
kernel.bin: boot/kernel_entry.o ${OBJ_FILES}
    $(LD) -m elf_i386 -o $@ -Ttext 0x1000 $^ --oformat binary


os-image.bin: boot/mbr.bin kernel.bin
    cat $^ > $@


run: os-image.bin
    qemu-system-i386 -fda $<


echo: os-image.bin
    xxd $<


# only for debug
kernel.elf: boot/kernel_entry.o ${OBJ_FILES}
    $(LD) -m elf_i386 -o $@ -Ttext 0x1000 $^


debug: os-image.bin kernel.elf
    qemu-system-i386 -s -S -fda os-image.bin -d guest_errors,int &
    i386-elf-gdb -ex "target remote localhost:1234" -ex "symbol-file
kernel.elf"


%.o: %.c ${HEADERS}
    $(CC) -g -m32 -ffreestanding -fno-pie -fno-stack-protector -c $< -o
$@ # -g for debugging
```

```
%.o: %.asm
    nasm $< -f elf -o $@

%.bin: %.asm
    nasm $< -f bin -o $@

%.dis: %.bin
    ndisasm -b 32 $< > $@

clean:
    $(RM) *.bin *.o *.dis *.elf
    $(RM) kernel/*.o
    $(RM) boot/*.o boot/*.bin
    $(RM) drivers/*.o
    $(RM) cpu/*.o
```

아래는 MinsungOS 실행화면 입니다.