Worksheet 2 - Inputs & Interrupts

Worksheet 2 is split into two parts, this is the second part. Each part is worth 50% of the total for worksheet 2, which itself is worth 50% of the module assessment, with worksheet 1 making up the other 50%.

Marking scheme

- Task 1 - 20%
- Task 2 - 20%
- Task 3 - 40%
- Readme - 20%

All work should be documented in the worksheets README.md.

Introduction

In the previous worksheet, we implemented a framebuffer to write characters to the terminal. This worksheet builds on that foundation by implementing keyboard input handling through interrupts. We'll be following Chapter 6 of the OS handbook, which covers interrupt handling and basic I/O operations.

Core Concepts

Interrupts Overview

Interrupts are signals to the processor indicating that it should pause its current execution to handle some immediate task (like keyboard input). In this worksheet, we'll focus on handling keyboard interrupts through the Programmable Interrupt Controller (PIC).

Input/Output (I/O)

Our OS needs to communicate with hardware devices. We'll use I/O ports to read keyboard input and configure the interrupt controller. This requires both assembly-level port access and higher-level C code to process the data.

Implementation Steps

1. Extending the I/O Library

First, we'll add the ability to read input from I/O ports. Add this function to your **io.s** file:

```
    global inb
    ; inb - returns a byte from the given I/O port
    ; stack: [esp + 4] The address of the I/O port
    ;        [esp    ] The return address
    inb:
        mov dx, [esp + 4]       ; move the address of the I/O port to the dx
         ↪ register
        in  al, dx              ; read a byte from the I/O port and store it
         ↪ in the al register
        ret                     ; return the read byte
```

Update your **io.h** header with:

```
unsigned char inb(unsigned short port);
```

2. Setting Up Type Definitions

Create a new file **types.h** for consistent data types:

```
#ifndef INCLUDE_TYPES_H
#define INCLUDE_TYPES_H

typedef unsigned int   u32int;
typedef int            s32int;
typedef unsigned short u16int;
typedef short          s16int;
typedef unsigned char u8int;
typedef char           s8int;

/* Frame buffer colors */
#define BLACK      0
#define BLUE       1
#define LIGHT_GREY 7

#endif
```

3. Programmable Interrupt Controller (PIC) Setup

The PIC handles external interrupts from devices. Create **pic.h** and **pic.c** with the provided code. Key functions include: - `pic_remap()`: Reconfigures interrupt numbers to avoid conflicts - `pic_acknowledge()`: Signals completion of interrupt handling

```
#ifndef INCLUDE_PIC_H
#define INCLUDE_PIC_H

#include "drivers/type.h"

/*                      I/O port */
#define PIC_1       0x20        /* IO base address for master PIC */
#define PIC_2       0xA0        /* IO base address for slave PIC */
#define PIC_1_COMMAND   PIC_1
#define PIC_1_DATA  (PIC_1+1)
#define PIC_2_COMMAND   PIC_2
#define PIC_2_DATA  (PIC_2+1)

#define PIC_1_OFFSET 0x20
#define PIC_2_OFFSET 0x28

#define PIC_2_END PIC_2_OFFSET + 7

#define PIC_1_COMMAND_PORT 0x20
#define PIC_2_COMMAND_PORT 0xA0

#define PIC_ACKNOWLEDGE 0x20

#define PIC_ICW1_ICW4           0x01   /* ICW4 (not) needed */
#define PIC_ICW1_SINGLE         0x02   /* Single (cascade) mode */
#define PIC_ICW1_INTERVAL4      0x04   /* Call address interval 4 (8) */
#define PIC_ICW1_LEVEL          0x08   /* Level triggered (edge) mode */
#define PIC_ICW1_INIT           0x10   /* Initialization - required! */

#define PIC_ICW4_8086           0x01   /* 8086/88 (MCS-80/85) mode */
#define PIC_ICW4_AUTO           0x02   /* Auto (normal) EOI */
#define PIC_ICW4_BUF_SLAVE      0x08   /* Buffered mode/slave */
#define PIC_ICW4_BUF_MASTER     0x0C   /* Buffered mode/master */
#define PIC_ICW4_SFNM           0x10   /* Special fully nested (not) */

void pic_remap(s32int offset1, s32int offset2);
```

```
void pic_acknowledge(u32int interrupt);

#endif /* INCLUDE_PIC_H */
```

4. Interrupt Descriptor Table (IDT)

Create the interrupt handling infrastructure with these files: - **interrupts.h**: Defines the structures for interrupt descriptors - **interrupt_handlers.s**: Contains assembly-level interrupt handling - **interrupt_asm.s**: Defines individual interrupt handlers - **interrupts.c**: Implements the C-level interrupt handling

Interrupts are handled by the IDT (Interrupts Descriptor Table), as suggested in the OS handbook we are going to use structs to define each of these entries in the IDT. This should be added to a new file called **interrupts.h**.

```
#ifndef INCLUDE_INTERRUPTS
#define INCLUDE_INTERRUPTS

#include "type.h"

struct IDT
{
    u16int size;
    u32int address;
} __attribute__((packed));

struct IDTDescriptor {
    /* The lowest 32 bits */
    u16int offset_low; // offset bits 0..15
    u16int segment_selector; // a code segment selector in GDT or LDT

    /* The highest 32 bits */
    u8int reserved; // Just 0.
    u8int type_and_attr; // type and attributes
    u16int offset_high; // offset bits 16..31
} __attribute__((packed));

struct cpu_state {
    u32int eax;
    u32int ebx;
```

```
    u32int ecx;
    u32int edx;
    u32int ebp;
    u32int esi;
    u32int edi;
} __attribute__((packed));

struct stack_state {
    u32int error_code;
    u32int eip;
    u32int cs;
    u32int eflags;
} __attribute__((packed));

void interrupt_handler(struct cpu_state cpu, u32int interrupt, struct
 ↪  stack_state stack);
void interrupts_install_idt();

// Wrappers around ASM.
void load_idt(u32int idt_address);
void interrupt_handler_33();
void interrupt_handler_14();

#endif /* INCLUDE_INTERRUPTS */
```

Now we have created ptototypes for three functions in the **interrupts.h** file. These are functions written in assembler that we also need to define.

```
void load_idt(u32int idt_address);
void interrupt_handler_33();
void interrupt_handler_14();
```

For the function **load_idt** lets create **interrupt_handlers.s**

```
global  load_idt

; load_idt - Loads the interrupt descriptor table (IDT).
; stack: [esp + 4] the address of the first entry in the IDT
;        [esp    ] the return address
```

```
load_idt:
        mov eax, [esp + 4]
        lidt [eax]
        ret
```

Now for our interrupt handler routines it's slightly more complicated as we need to define a handler for every interrupt even though we are only really interested in the keyboard input right now. To do this we can create the assembler file **interrupt_asm.s**. This will define a number of interrupts with the name **interrupt_handler_x**, however we only really need the prototype **void interrupt_handler_33();** as this is for the keyboard.

```
;Generic Interrupt Handler
;
extern interrupt_handler

%macro no_error_code_interrupt_handler 1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword 0                     ; push 0 as error code
    push    dword %1                    ; push the interrupt number
    jmp common_interrupt_handler    ; jump to the common handler
%endmacro

%macro error_code_interrupt_handler 1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword %1                    ; push the interrupt number
    jmp     common_interrupt_handler    ; jump to the common handler
%endmacro

common_interrupt_handler:               ; the common parts of the generic
 ↪  interrupt handler
    ; save the registers
    push    eax
    push    ebx
    push    ecx
    push    edx
    push    ebp
    push    esi
    push    edi
```

```
        ; call the C function
        call    interrupt_handler

        ; restore the registers
    pop edi
    pop esi
    pop ebp
    pop edx
    pop ecx
    pop ebx
        pop     eax

    ; restore the esp
    add     esp, 8

    ; return to the code that got interrupted
    iret

no_error_code_interrupt_handler 33  ; create handler for interrupt 1
 ↪  (keyboard)
```

Next we are going to need some code to handle enabling and disabling of interrupts in our OS. Lets ass the following files to our repo. You could equally call **sti** and **cli** directly but this makes the code more readable.

hardware_interrupt_enabler.h

```
#ifndef INCLUDE_HARDWARE_INTERRUPT_ENABLER_H
#define INCLUDE_HARDWARE_INTERRUPT_ENABLER_H

void enable_hardware_interrupts();

void disable_hardware_interrupts();

#endif /* INCLUDE_USER_MODE_H */
```

hardware_interrupt_enabler.s

```
global enable_hardware_interrupts

enable_hardware_interrupts:
```

```
  sti
  ret

global disable_hardware_interrupts

disable_hardware_interrupts:
  cli
  ret
```

Now finally we can create our **interrupts.c** which contains the code to setup the interrutps themself and provide a handler to get data from the keyboard itself.

interrupts.c

```
#include "interrupts.h"
#include "pic.h"
#include "io.h"
#include "frame_buffer.h"
#include "keyboard.h"

#define INTERRUPTS_DESCRIPTOR_COUNT 256
#define INTERRUPTS_KEYBOARD 33
#define INPUT_BUFFER_SIZE 256

u8int input_buffer[INPUT_BUFFER_SIZE];
u8int buffer_index = 0;

struct IDTDescriptor idt_descriptors[INTERRUPTS_DESCRIPTOR_COUNT];
struct IDT idt;

u32int BUFFER_COUNT;

void interrupts_init_descriptor(s32int index, u32int address)
{
    idt_descriptors[index].offset_high = (address >> 16) & 0xFFFF; // offset
 ↪  bits 0..15
    idt_descriptors[index].offset_low = (address & 0xFFFF); // offset bits
 ↪  16..31

    idt_descriptors[index].segment_selector = 0x08; // The second (code)
 ↪  segment selector in GDT: one segment is 64b.
    idt_descriptors[index].reserved = 0x00; // Reserved.
```

```
    /*
       Bit:     | 31              16 | 15 | 14 13 | 12 | 11     10 9 8    | 7 6
 ↪  5 | 4 3 2 1 0 |
       Content: | offset high        | P  | DPL   | S  | D and  GateType | 0
 ↪  0 0 | reserved
        P   If the handler is present in memory or not (1 = present, 0 = not
 ↪  present). Set to 0 for unused interrupts or for Paging.
        DPL Descriptor Privilige Level, the privilege level the handler can
 ↪  be called from (0, 1, 2, 3).
        S   Storage Segment. Set to 0 for interrupt gates.
        D   Size of gate, (1 = 32 bits, 0 = 16 bits).
    */
    idt_descriptors[index].type_and_attr =  (0x01 << 7) |           // P
                        (0x00 << 6) | (0x00 << 5) | // DPL
                       0xe;              // 0b1110=0xE 32-bit interrupt gate
}

void interrupts_install_idt()
{

    interrupts_init_descriptor(INTERRUPTS_KEYBOARD, (u32int)
 ↪  interrupt_handler_33);


    idt.address = (s32int) &idt_descriptors;
    idt.size = sizeof(struct IDTDescriptor) * INTERRUPTS_DESCRIPTOR_COUNT;
    load_idt((s32int) &idt);

    /*pic_remap(PIC_PIC1_OFFSET, PIC_PIC2_OFFSET);*/
    pic_remap(PIC_1_OFFSET, PIC_2_OFFSET);

    // Unmask keyboard interrupt (IRQ1)
    outb(0x21, inb(0x21) & ~(1 << 1));
}

/* Interrupt handlers
 ↪  ************************************************************/

void interrupt_handler(__attribute__((unused)) struct cpu_state cpu, u32int
 ↪  interrupt, __attribute__((unused)) struct stack_state stack) {
    u8int input;
```

```
    u8int ascii;

    switch (interrupt) {
        case INTERRUPTS_KEYBOARD:

            while ((inb(0x64) & 1)) {
                input = keyboard_read_scan_code();

                // Only process if it's not a break code
                if (!(input & 0x80)) {
                    if (input <= KEYBOARD_MAX_ASCII) {
                        ascii = keyboard_scan_code_to_ascii(input);
                        if (ascii != 0) {

                            // We have detected a backspace
                            if (ascii == '\b') {
                                // Remove the last character
                            }
                            // We have detected a newline
                            else if (ascii == '\n') {
                                // Move our position to a newline
                            }
                            // We have detected a regular character
                            else {
                                // Add the new character to the display
                            }
                        }
                    }
                }

                buffer_index = (buffer_index + 1) % INPUT_BUFFER_SIZE;
            }

            pic_acknowledge(interrupt);
            break;

        default:
            break;
    }
}
```

5. Keyboard Input Processing

Implement keyboard handling in **keyboard.h** and **keyboard.c**. These files manage: - Reading scan codes from the keyboard - Converting scan codes to ASCII characters - Handling special keys (backspace, enter)

keyboard.h

```
#ifndef INCLUDE_KEYBOARD_H
#define INCLUDE_KEYBOARD_H

#define KEYBOARD_MAX_ASCII 83

#include "drivers/type.h"

u8int keyboard_read_scan_code(void);

u8int keyboard_scan_code_to_ascii(u8int);

#endif /* INCLUDE_KEYBOARD_H */
```

keyboard.c

```
#include "io.h"
#include "frame_buffer.h"

#define KEYBOARD_DATA_PORT 0x60

/** read_scan_code:
 *  Reads a scan code from the keyboard
 *
 *  @return The scan code (NOT an ASCII character!)
 */
u8int keyboard_read_scan_code(void)
{
    return inb(KEYBOARD_DATA_PORT);
}

u8int keyboard_scan_code_to_ascii(u8int scan_code)
{
    // Ignore key releases (scan codes with bit 7 set)
    if (scan_code & 0x80) {
```

```
        return 0;
    }

    // Scan code to ASCII mapping for standard US QWERTY keyboard
    switch(scan_code) {

        // Numbers row
        case 0x02: return '1';
        case 0x03: return '2';
        case 0x04: return '3';
        case 0x05: return '4';
        case 0x06: return '5';
        case 0x07: return '6';
        case 0x08: return '7';
        case 0x09: return '8';
        case 0x0A: return '9';
        case 0x0B: return '0';
        case 0x0C: return '-';
        case 0x0D: return '=';
        case 0x0E: return '\b';  // Backspace

        // Top letter row
        case 0x10: return 'q';
        case 0x11: return 'w';
        case 0x12: return 'e';
        case 0x13: return 'r';
        case 0x14: return 't';
        case 0x15: return 'y';
        case 0x16: return 'u';
        case 0x17: return 'i';
        case 0x18: return 'o';
        case 0x19: return 'p';
        case 0x1A: return '[';
        case 0x1B: return ']';
        case 0x1C: return '\n';  // Enter

        // Middle letter row
        case 0x1E: return 'a';
        case 0x1F: return 's';
        case 0x20: return 'd';
        case 0x21: return 'f';
        case 0x22: return 'g';
```

```
        case 0x23: return 'h';
        case 0x24: return 'j';
        case 0x25: return 'k';
        case 0x26: return 'l';
        case 0x27: return ';';
        case 0x28: return '\'';
        case 0x29: return '`';

        // Bottom letter row
        case 0x2B: return '\\';
        case 0x2C: return 'z';
        case 0x2D: return 'x';
        case 0x2E: return 'c';
        case 0x2F: return 'v';
        case 0x30: return 'b';
        case 0x31: return 'n';
        case 0x32: return 'm';
        case 0x33: return ',';
        case 0x34: return '.';
        case 0x35: return '/';
        case 0x39: return ' ';  // Space bar

        // Numpad
        case 0x37: return '*';  // Numpad *
        case 0x47: return '7';
        case 0x48: return '8';
        case 0x49: return '9';
        case 0x4A: return '-';
        case 0x4B: return '4';
        case 0x4C: return '5';
        case 0x4D: return '6';
        case 0x4E: return '+';
        case 0x4F: return '1';
        case 0x50: return '2';
        case 0x51: return '3';
        case 0x52: return '0';
        case 0x53: return '.';

        default: return 0;  // Unknown scan code
    }
}
```

Don't forget to setup a makefile for your project. It will look something like this but will need to be changed based on your project.

```
OBJECTS = source/loader.o \
          source/kmain.o \
          source/io.o \
          drivers/frame_buffer.o \
          drivers/hardware_interrupt_enabler.o \
          drivers/interrupt_asm.o \
          drivers/interrupt_handlers.o \
          drivers/interrupts.o \
          drivers/keyboard.o \
          drivers/pic.o

CC = gcc

CFLAGS = -I. -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector \
        -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c

LDFLAGS = -T ./source/link.ld -melf_i386

AS = nasm

ASFLAGS = -f elf

all: kernel.elf

kernel.elf: $(OBJECTS)
    ld $(LDFLAGS) $(OBJECTS) -o kernel.elf

os.iso: kernel.elf
    cp kernel.elf iso/boot/kernel.elf
    genisoimage -R                              \
                -b boot/grub/stage2_eltorito    \
                -no-emul-boot                   \
                -boot-load-size 4               \
                -A os                           \
                -input-charset utf8             \
                -quiet                          \
                -boot-info-table                \
                -o os.iso                       \
                iso
```

```
run: os.iso
    qemu-system-i386 -nographic -boot d -cdrom os.iso -m 32 -d cpu -D
 ↪  logQ.txt

run-curses: os.iso
    qemu-system-i386 -curses \
        -monitor telnet::75454,server,nowait \
        -chardev stdio,id=char0 \
        -serial chardev:char0 \
        -boot d \
        -cdrom os.iso \
        -m 32 \
        -d cpu \
        -no-reboot \
        -no-shutdown \
        -D logQ.txt

%.o: %.c
    $(CC) $(CFLAGS)  $< -o $@

%.o: %.s
    $(AS) $(ASFLAGS) $< -o $@

clean:
    rm -rf *.o source/*.o drivers/*.o kernel.elf os.iso ios/boot/kernel.elf
```

Tasks

Part 1 - Display Keyboard Input

Extend the `interrupt_handler` function in **interrupts.c** to process keyboard input. Your implementation should:

1. Read the scan code from the keyboard
2. Convert it to ASCII
3. Handle special cases:
    - Backspace: Remove last character
    - Enter: Move to new line
    - Regular characters: Display on screen

4. Clear screen when appropriate

Example addition to interrupt_handler:

```
case INTERRUPTS_KEYBOARD:
    while ((inb(0x64) & 1)) {
        input = keyboard_read_scan_code();
        if (!(input & 0x80)) {  // Key press, not release
            ascii = keyboard_scan_code_to_ascii(input);
            if (ascii == '\b') {
                // Handle backspace
                fb_backspace();
            } else if (ascii == '\n') {
                // Handle newline
                fb_newline();
            } else if (ascii != 0) {
                // Display character
                fb_write_char(ascii);
            }
        }
    }
    pic_acknowledge(interrupt);
    break;
```

Part 2 - Input Buffer API

Create a system to efficiently store and retrieve keyboard input. You need to implement two key functions:

1. `getc()` Function Requirements:
    - Remove and return a single character from the buffer
    - Handle empty buffer conditions appropriately
    - Consider using a circular buffer to manage input efficiently
    - Must be callable from other parts of your OS
2. `readline()` Function Requirements:
    - Read characters until a newline is encountered
    - Use your `getc()` function to retrieve characters
    - Store the result in a provided buffer
    - Handle buffer size limits safely

- Return when enter/newline is detected

Think carefully about: - Buffer size limitations - How to handle buffer overflow - Synchronization between interrupt handler and buffer access - Efficient memory usage - Error conditions

Part 3 - Terminal Implementation

Create a basic terminal interface that processes user commands. Your terminal should feel similar to a basic Unix shell.

Requirements:

1. Command Processing
    - Display a prompt (e.g., “myos>”)
    - Accept user input until enter is pressed
    - Parse input into command and arguments
    - Execute appropriate function based on command
    - Handle unknown commands gracefully
2. Suggested Commands (Pick at least 2)
    - echo [text] - Display the provided text
    - clear - Clear the screen
    - help - Show available commands
    - version - Display OS version
    - shutdown - Prepare system for shutdown
3. Implementation Hints:
    - Split your input string at space characters to separate command and arguments
    - Consider using a command structure like:

```
struct command {
    const char* name;
    void (*function)(char* args);
};
```

    - Store commands in an array or table for easy lookup
    - Remember to null-terminate your strings
    - Handle empty input gracefully
4. Command Parsing Steps:

- Identify the command (first word)
- Separate arguments (remaining text)
- Look up command in your command table
- Pass arguments to appropriate handler
- Display results

Think about: - How to handle command line editing - Error messages for unknown commands - Command output formatting - Memory management - Input string validation ## Extension Tasks

If you complete the main tasks, try these extensions: 1. Add command history (accessible with up/down arrows) 2. Implement tab completion for commands 3. Add color support to the terminal 4. Create a basic file system command (ls) ## Common Issues

- Keyboard not responding: Check PIC initialization
- Characters not displaying: Verify framebuffer code
- System hanging: Check interrupt acknowledgment
- Buffer overflow: Verify array bounds
- Missing characters: Check scan code conversion

Resources

- OS Dev Wiki: https://wiki.osdev.org/
- OS Handbook
- Previous worksheet materials