I/O & Interrupts
COMP 310 Operating Systems

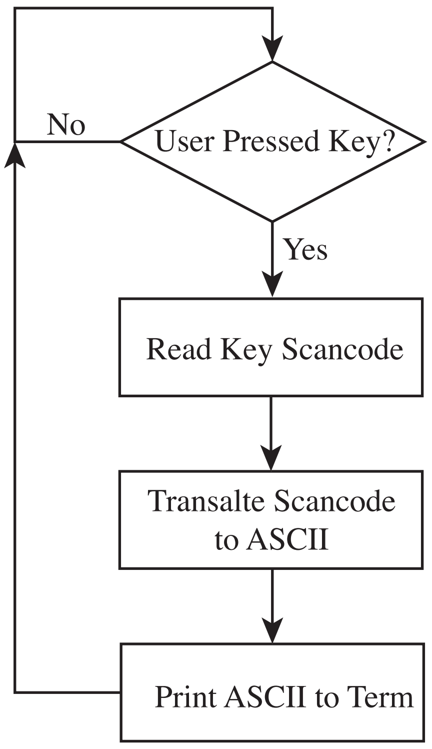
September 16, 2022

1 Introduction

Problem: How to communicate with external devices?
From the computer's perspective, we only have a CPU and memory.

- Special addresses in memory reserved to interact with hardware: sending commands, reading/writing data, etc.
- Interrupts from the external devices tell the computer that it needs attention.

1.1 Basic Example: Polling



I/O Port	Access Type	Function
`0x60`	R/W	Data Port
`0x64`	R	Status Register
`0x64`	W	Command Register

1.1.1 PS/2 Status Register

7	6	5	4	3	2	1	0
`PE`	`TOE`	`UNK`	`UNK`	`CD`	`SF`	`IS`	`OS`

`PE`	**Parity Error**	0 indicates no error, 1 indicates error
`TOE`	**Timeout Error**	0 indicates no error, 1 indicates error
`UNK`	**Chipset Specific**	
`UNK`	**Chipset Specific**	
`CD`	**Command/Data**	0 indicates data written to input buffer is data for PS/2 device, 1 indicates data written to input buffer is data fro PS/2 Controller
`SF`	**System Flag**	Set by BIOS if system passes self test (POST)
`IS`	**Input Buffer Status**	0 means empty, 1 means full
`OS`	**Output Buffer Status**	0 means empty, 1 means full

1.1.2 Reading and Writing to IO Ports on x86

On x86, we have two special instructions to communicate with IO ports: `in` and `out`. We have to use inline assembly to actually use the `in` and `out` instructions.

```
/*
 * inb
 *
 * Reads from I/O port _port, and returns a one byte value
 */
uint8_t inb (uint16_t _port) {
    uint8_t rv;
    __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
    return rv;
}

/*
 * outb
 *
 * Writes val to I/O port _port
 */
void outb (uint16_t _port, uint8_t val) {
    __asm__ __volatile__ ("outb %0, %1" : : "a" (val),  "dN" (_port) );
}
```

2 Deliverables

Write C code to do the following in your OS:

1. Read the PS/2 status register and check the LSB.
2. If the LSB is 1, then that means that means that the PS/2 controller's output buffer contains a scancode.
3. To get the scancode, read from IO port `0x60`.
4. Print the scancode to the terminal

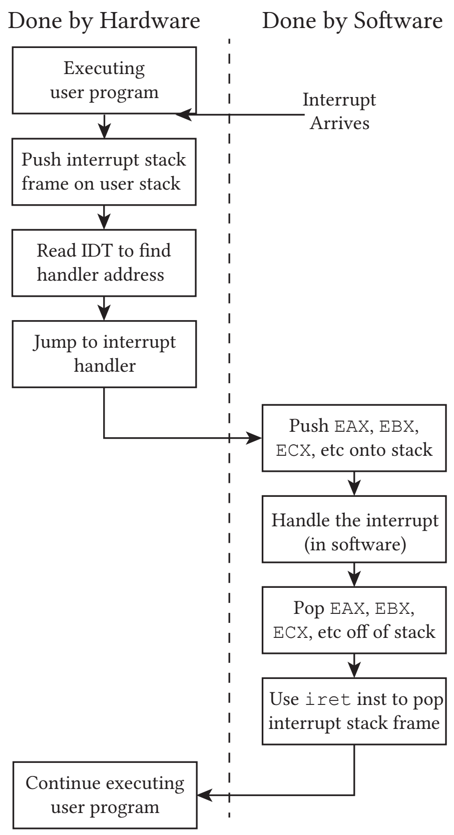
Question: is this a good approach?

What happens when you have more than one I/O device? What about if you have some other tasks running that don't involve I/O? Will this approach affect responsiveness?

3 Interrupts

Interrupts are functions that are **called by the hardware** to handle IO. The interrupt handling process is basically the same for all processors, but the details of the data structures are different. Here's how the process works in general:

- There is an array of pointers called a vector table in memory somewhere. Location is defined by the CPU manufacturer.
- Each entry in the array points to a function that handles a particular IO event.
- When an IO event occurs:
    1. We stop executing the user code
    2. Push an interrupt stack frame onto the stack
    3. Jump to the address in the vector table and run the interrupt handler.
    4. Pop the interrupt stack frame and return to user code.


Done by Hardware
Done by Software
Executing user program
Interrupt Arrives
Push interrupt stack frame on user stack
Read IDT to find handler address
Jump to interrupt handler
Push EAX, EBX, ECX, etc onto stack
Handle the interrupt (in software)
Pop EAX, EBX, ECX, etc off of stack
Use iret inst to pop interrupt stack frame
Continue executing user program

Interrupt Stack Frame When the i386 takes an interrupt, it needs to save the context of the program that was running immediately before the interrupt arrived so it can pick up where it left off after it finishes processing the interrupt. The CPU creates an *exception stack frame* to store important registers such as the `ESP`, `EFLAGS`, and `EIP`. The user registers (`EAX`, `EBX`, `ECX`, etc) should be saved by software in the interrupt service routine. The diagram below shows the structure of the i386 exception stack frame.

ESP+16	unused	SS
ESP+12	ESP	
ESP+8	EFLAGS	
ESP+4	unused	CS
ESP+0	EIP	

3.0.1 Installing an Interrupt Vector on i386

The i386 interrupt descriptor table (IDT) is an array of structs that point to handlers for various I/O events. The IDT can live anywhere in memory. Usually we will just create an array to store it. We have to tell the processor where it is using the `lidt` instruction, which loads the base address of the IDT. Descriptors in the array (for some reason descriptors are called "gates" by Intel) have the following format:

63 48
offset

47	46 45	44	43 40	39 32
P	DPL	0	Type	Reserved

31 16
Segment

15 0
offset

Field	Description
Offset	A 32-bit value, split in two parts. It represents the address of the entry point of the Interrupt Service Routine.
Selector	A Segment Selector with multiple fields which must point to a valid code segment in your GDT.
Type	A 4-bit value which defines the type of gate this Interrupt Descriptor represents. There are five valid type values: 0101: Task Gate, note that in this case, the Offset value is unused and should be set to zero. 0110 16-bit Interrupt Gate 0111: 16-bit Trap Gate 1110: 32-bit Interrupt Gate 1111: 32-bit Trap Gate
DPL	A 2-bit value which defines the CPU Privilege Levels which are allowed to access this interrupt via the INT instruction. Hardware interrupts ignore this mechanism.
P	Present bit. Must be set (1) for the descriptor to be valid.

A C data struct that implements an interrupt gate is given below.

```
// A struct describing an interrupt gate.
struct idt_entry
{
   uint16_t base_lo;             // The lower 16 bits of the address to jump to when this interrupt fires.
   uint16_t sel;                 // Kernel segment selector.
   uint8_t  always0;             // This must always be zero.
   uint8_t  flags;               // More flags. See documentation.
   uint16_t base_hi;             // The upper 16 bits of the address to jump to.
} __attribute__((packed));
```