Demo

Keyboard Interrupts:

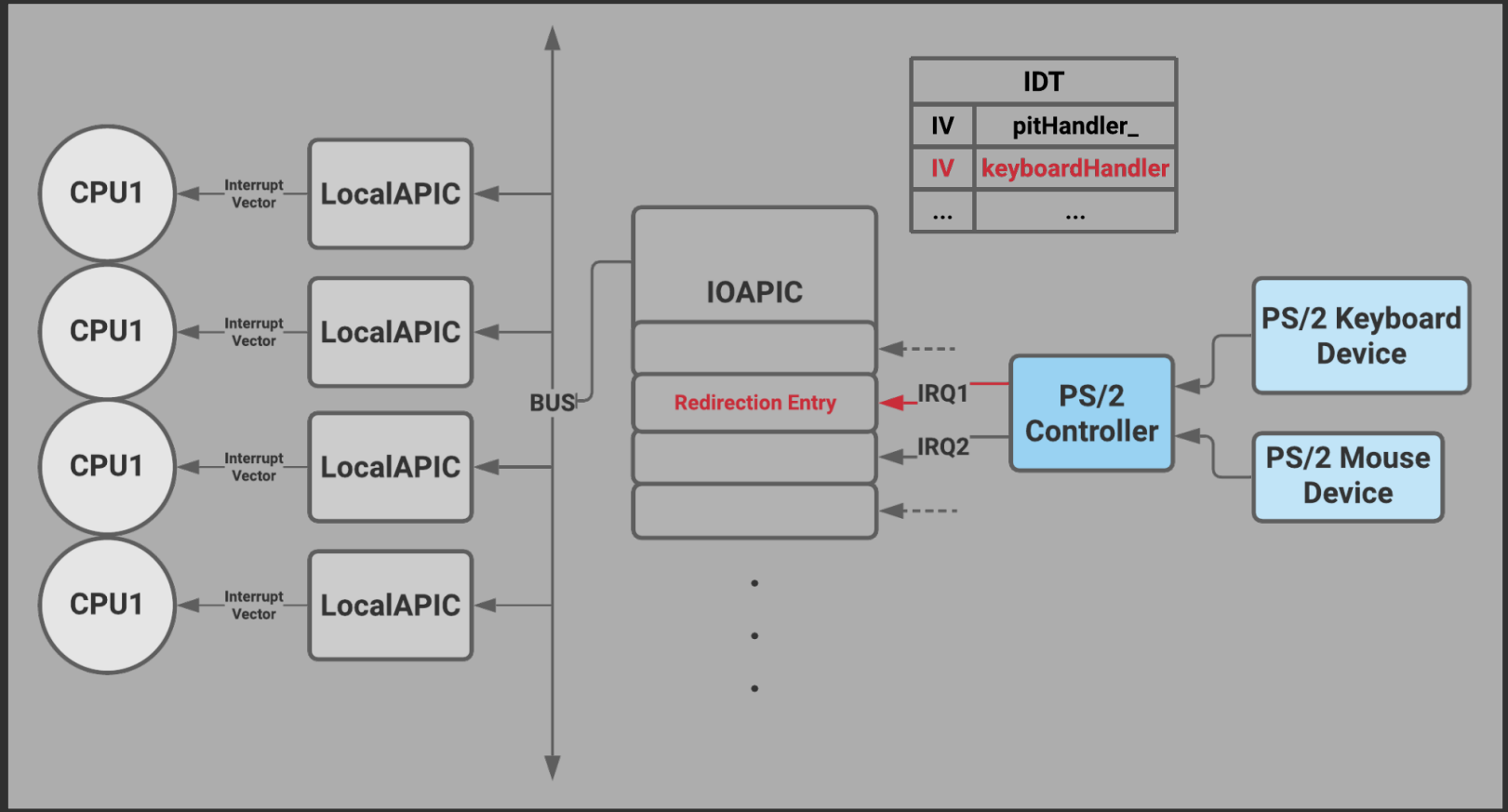
- Programming the IOAPIC
- IDT Entry and the CPU
- Keyboard Handler w scancode to ASCII translation

VGA Graphics

- Mode 13 and Bios Interrupts
- Frame Buffer and Drawing to VGA Memory
- .psf Bitmaps to Draw Characters

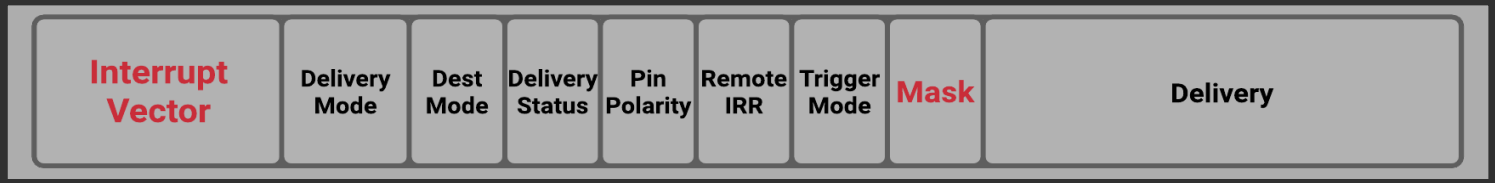
Tetris

Keyboard Interrupts



Programming the IOAPIC

- Create Redirection Entry
  - IV = 0x9
  - Mask = 0



- Write to the IOAPIC
  - 2x 32-bit writes to corresponding register indices
  - Register 0x12, 0x13

```
RedirectionEntry f = getRedirectionEntry(0x12);
f.setVector(0x9);
f.setMask(0);
writeRedirectionEntry(0x12, f);
```

```
void writeRedirectionEntry(uint32_t reg, RedirectionEntry newRE) {
    Config::writeIOApic(reg, newRE.lower);
    Config::writeIOApic(reg + 1, newRE.upper);
}
```

IDT Entry and the CPU

- Set the IDT entry with a pointer to our handler

```
IDT::interrupt(0x9, (uint32_t)keyboardHandler_);
```

- ~Keyboard Handler

```
extern "C" void keyboardHandler() {
    Keyboard::handle_interrupt();
    SMP::eoi_reg.set(0);
}
```

Send an EOI (End of Interrupt) signal to indicate we have handled the interrupt

Keyboard Handler w scancode to ASCII translation

```
// Scancode -> ASCII
const uint8_t lower_ascii_codes[256] = {
    0x00,  ESC,  '1',  '2',     /* 0x00 */
     '3',  '4',  '5',  '6',     /* 0x04 */
     '7',  '8',  '9',  '0',     /* 0x08 */
     '-',  '=',   BS, '\t',     /* 0x0C */
     'q',  'w',  'e',  'r',     /* 0x10 */
     't',  'y',  'u',  'i',     /* 0x14 */
     'o',  'p',  '[',  ']',     /* 0x18 */
    '\n', 0x00,  'a',  's',     /* 0x1C */
     'd',  'f',  'g',  'h',     /* 0x20 */
     'j',  'k',  'l',  ';',     /* 0x24 */
    '\'',  '`', 0x00, '\\',     /* 0x28 */
     'z',  'x',  'c',  'v',     /* 0x2C */
     'b',  'n',  'm',  ',',     /* 0x30 */
     '.',  '/', 0x00,  '*',     /* 0x34 */
    0x00,  ' ', 0x00, 0x00,     /* 0x38 */
    0x00, 0x00, 0x00, 0x00,     /* 0x3C */
    0x00, 0x00, 0x00, 0x00,     /* 0x40 */
    0x00, 0x00, 0x00,  '7',     /* 0x44 */
     '8',  '9',  '-',  '4',     /* 0x48 */
     '5',  '6',  '+',  '1',     /* 0x4C */
     '2',  '3',  '0',  '.',     /* 0x50 */
    0x00, 0x00, 0x00, 0x00,     /* 0x54 */
    0x00, 0x00, 0x00, 0x00      /* 0x58 */
};
```

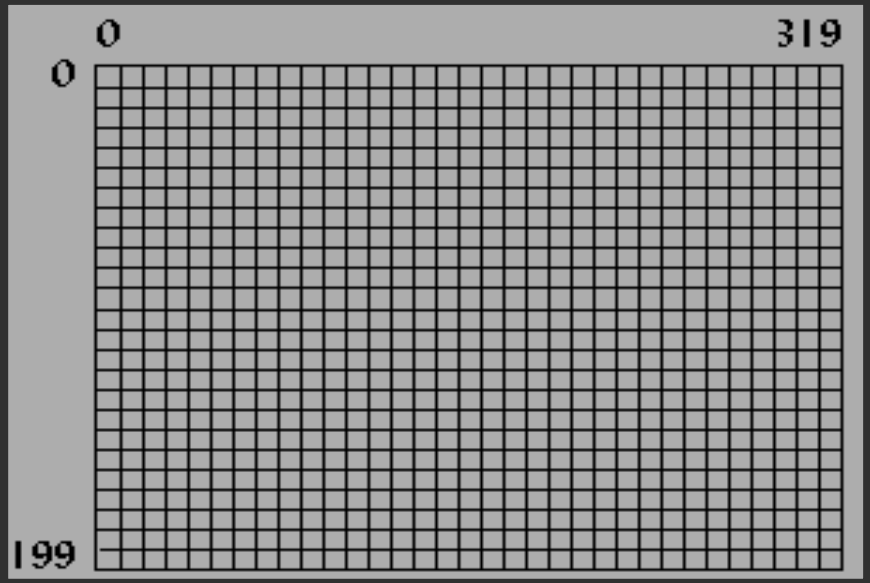
- Map Set 1 scan codes to ASCII values
  - PS/2 Controller translates PS/2 Device from Set 2 to Set 1 for us
- Identical array but for upper_ascii_codes
  - Keyboard storing state of shift/caps buttons

```
void Keyboard::handle_interrupt() {
    // the current key being pressed
    unsigned char byte = inb(0x60);

    // key release
    if(byte & 0x80) {
        // check if one of the toggle keys and adjust state
        ...
    } else {
        // key being pressed, check if toggle keys and adjust state
        ...
        // map to ascii value, store in queue
        ...
    }
}
```

VGA Graphics

Mode 13: 320 x 200



- VGA Memory: 0xA0000
- 256 colors // 1 byte pixels

Mode Number	Mode	WxHxD
00	text	40×25×16×2 (B/W text)
01	text	40×25×16
02	text	80×25×16 (Gray text)
03	text	80×25×16
04	CGA	320×200×4
05	CGA	320×200×4
06	CGA	640×200×2
07	MDA monochrome text	80×25 Monochrome text
08	PCjr	160×200×16
09	PCjr	320×200×16
0A	PCjr	640×200×16
0B	reserved	
0C	reserved	
0D	EGA	320 ×200× 16
0E	EGA	640×200×16
0F	EGA	640×350 Monochrome graphics
10	EGA	640×350×16
11	VGA	640×480×2
12	VGA	640×480×16
13	VGA	320×200×256

Mode 13 and Bios Interrupts

- BIOS Interrupt (0x10) with %ah = 0x00 (set video mode) and %al = 0x13 (desired video mode)
- Have to be in 16-bit real mode
- Switch in the bootloader

```
switchVideoModes:
    movb $0x00, %ah
    movb $0x13, %al
    int $0x10
    ret
```

Called in mbr.S:27

Frame Buffer and Drawing to VGA Memory

- Map (x, y) screen coords to the linear screen buffer

```
// maps (x,y) coordinates to linear memory
uint32_t offset(uint32_t x, uint32_t y) {
    return x + (y << 8) + (y << 6);
}
```

- Then plotting a pixel simply becomes

```
void plot(uint32_t x, uint32_t y, Color color, uint8_t *double_buffer ) {
    if (double_buffer != nullptr) {
        double_buffer[offset(x, y)] = color;
    } else {
        VGA[offset(x, y)] = color;
    }
}
```

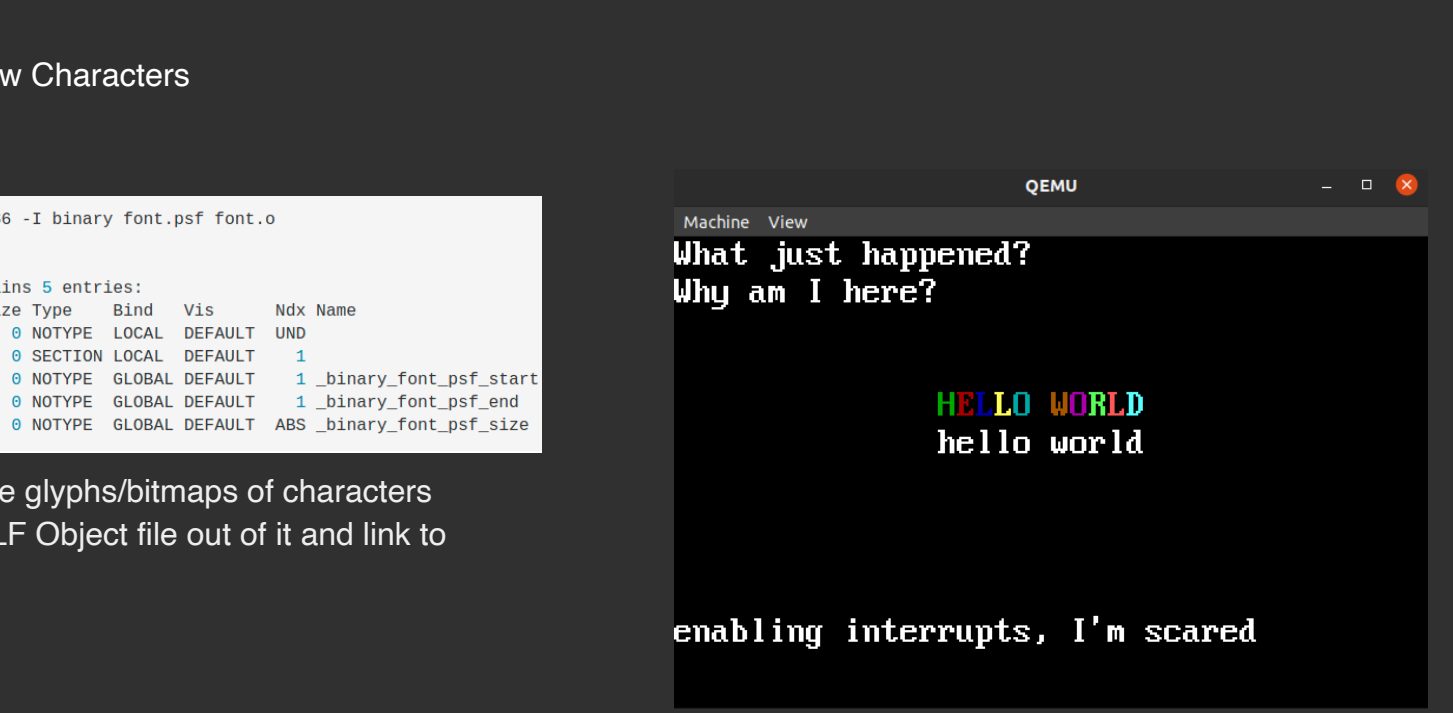
VGA Mode 13 - 256 Color Palette

.psf Bitmaps to Draw Characters

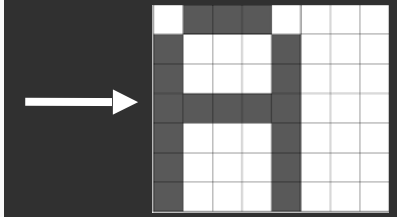
```
objcopy -O elf32-i386 -B i386 -I binary font.psf font.o
readelf -s font.o

Symbol table '.symtab' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    1 _binary_font_psf_start
     3: 0000000000008020     0 NOTYPE  GLOBAL DEFAULT    1 _binary_font_psf_end
     4: 0000000000008020     0 NOTYPE  GLOBAL DEFAULT  ABS _binary_font_psf_size
```

- .psf files store glyphs/bitmaps of characters
- Create an ELF Object file out of it and link to kernel




```
00000000b  byte  0
00000000b  byte  1
00000000b  byte  2
00010000b  byte  3
00111000b  byte  4
01101100b  byte  5
11000110b  byte  6
11000110b  byte  7
11111110b  byte  8
11000110b  byte  9
11000110b  byte 10
11000110b  byte 11
11000110b  byte 12
00000000b  byte 13
00000000b  byte 14
00000000b  byte 15
```



Bitmap to Pixels

0 being indicating to draw background color

1 being indicating to draw foreground color

Tetris

- Shape class and Tetris class
- Tetris uses a buffer to store the game state and write to screen
  - We found that most of the time, it’s actually faster to write directly to the screen
- Move method
- Uses Pit::jiffies to force the block to move down periodically
- Difficulties:
  - Representing the shapes
  - Deciding how to update the shapes
  - Checking for collisions

Future Extensions

- Be able to “pop” a row once it is completed
- Use more threads to update different parts of the screen, since updating the screen to pop a row takes a long time
- Make it run in user mode