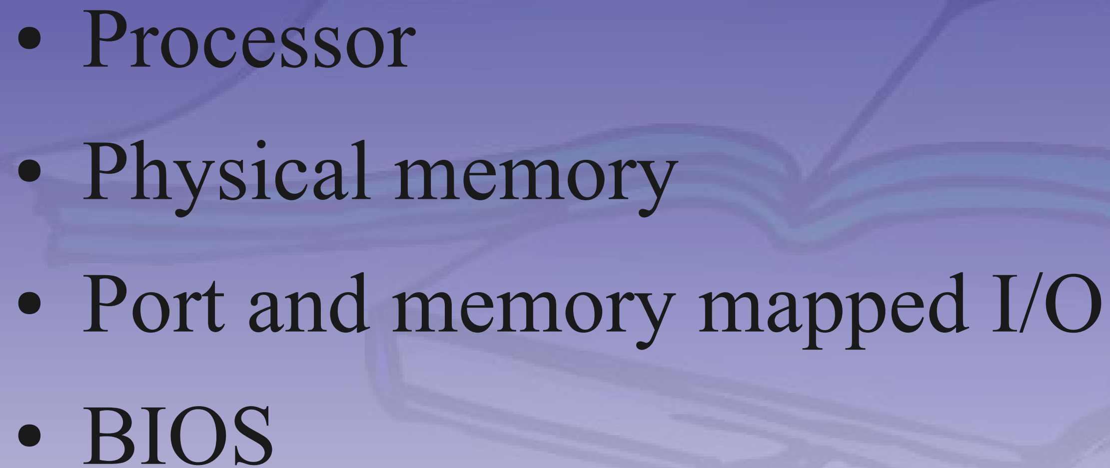
CS422: Assignment 1

Alex Vaynberg
Dept. of Computer Science
Yale University

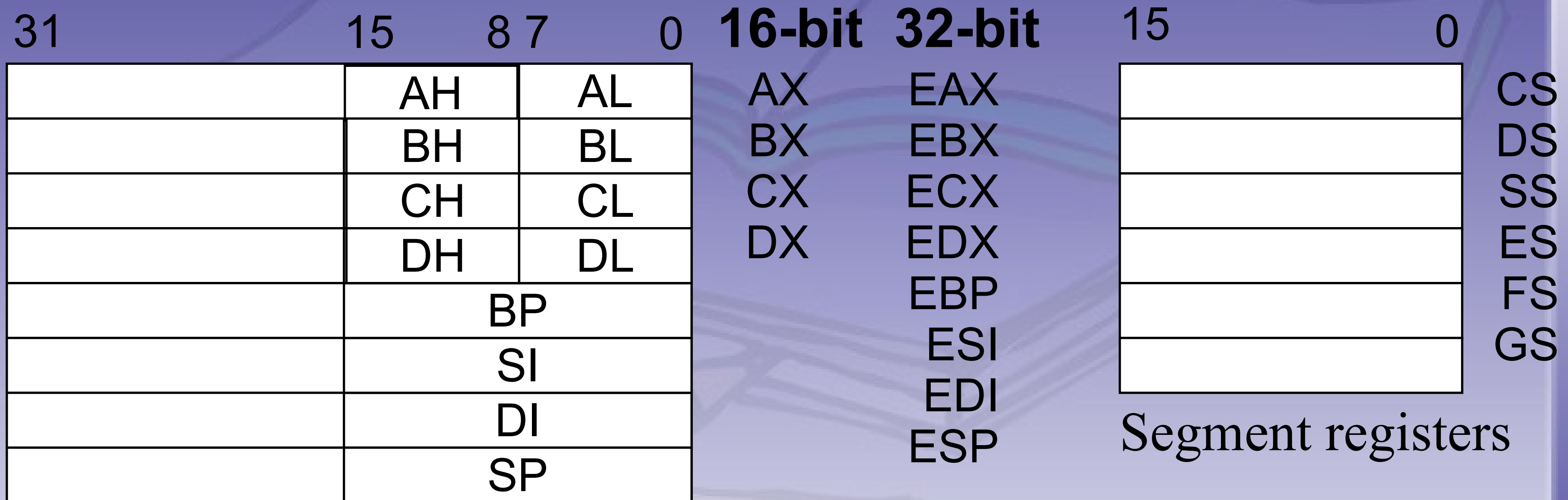
- The impossibly short intro to x86 assembly.
- BIOS. No, not the ones you write for match.com
- The x86 booting process.
- Bootloader and Kernel: a tale of two heroes.
- ELFes: What they are, and how to get rid of them.
- What is your quest? And where to start it?
- Submitting your glory!

x86 Arch



- Processor
- Physical memory
- Port and memory mapped I/O
- BIOS

Processor and Registers



EFLAGS register

EIP (Instruction Pointer register)

Interlude: Assem with Registers

INTEL: MOV y,x GAS: mov[b,w,~~l~~] x, y

- Moves data from x to y
- Operands: constants, registers, memory locations
- There are restrictions on x and y
  - can not move a constant into a segment register
  - can not move from 8-bit to 16-bit register
  - x and y can not both be memory locations
- GAS requires explicit size (byte, word, ~~long~~)

Interlude: MOV examples

movw $5, %ax

- moves 5 into general purpose register AX

movw %ax, %ss

- moves the contents of AX into SS

movw %bx, (%si)

- moves the contents of BX into memory location given by SI in the data segment
  - what is this mysterious data segment?

Interlude: x86 Arithmetic

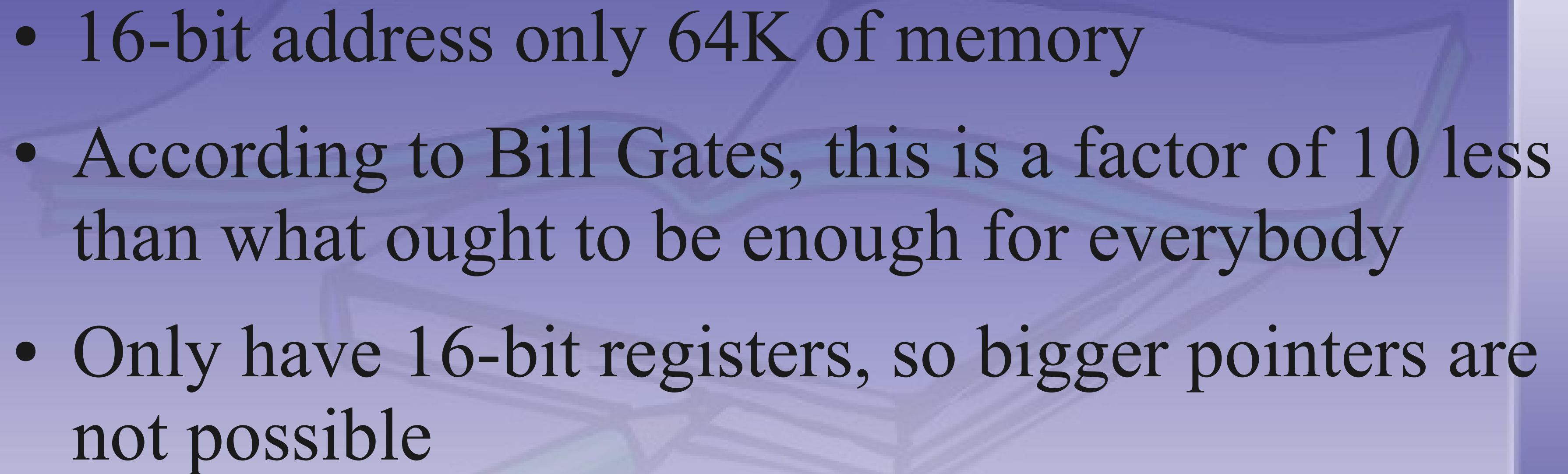
- How does one use only two registers to add?
  - replace first operand with results
- add[b,w,l], sub[b,w,l], mul[b,w,l]
  - mul places result in (e)ax, overflow in (e)dx
  - division is harder – avoid
- addw %ax, %bx
  - adds contents of bx to contents of ax, updating ax
- mulw $0x10
  - multiplies the contents of ax by 16, trashes dx

Interlude: Conditional Jumps

- cmpw and j[e,g,z]
- cmpw compares two words-size integers
  - result is stored in FLAGS
- je – takes the jump if equality at last comparison
- jg – takes the jump if oper1 is greater than oper2
  - GNU syntax is unintuitive

```
cmp %ax, %bx
jg  label    # jump to label if bx > ax
```

Segmented Memory Addressing



- 16-bit address only 64K of memory
- According to Bill Gates, this is a factor of 10 less than what ought to be enough for everybody
- Only have 16-bit registers, so bigger pointers are not possible

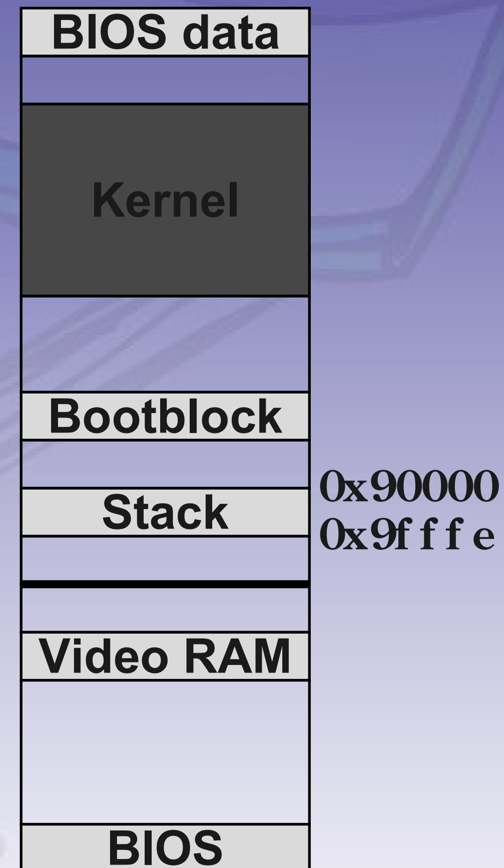
Segmented Memory Addressing

+	SI		F	F	F	B
16 *	ES	F	0	0	0	
	ES:SI	F	F	F	F	B

- Solution: use two registers
- The segment register is shifted 4 bits
- Gives a total of twenty bits
- (and a super secret 21st bit. SHHHH)

Memory Architecture

- It is not just a collection of bits
- Memory controller maps certain ranges of memory to other tasks
- VGA framebuffer at 0xb8000
- Interrupt table at 0x00000
- BIOS, MMIO devices live in HIGH memory area
- OS programmers need to follow conventions defined by the arch



Memory-Mapped I/O

- Talking to a device by reading and writing from specific locations in memory
- When, where, and what are in device spec
- Standards exist (PnP, PCI, x86 BIOS, VESA)

Example: *(%fs = 0xb800)* movw $0x71, %fs:($0x100)

- 80x25 screen (2 bytes per character) at 0xb8000
- above command places 0x71 'q' at offset 0x100
- video card receives data, and places q at pos (1,0)

Port-Mapped IO

- Communicate with devices by sending data to an integer numbered port assigned to a device
- Same idea as memory-mapped IO
- No bulk writes: ports take bytes or words only
- IN and OUT instructions
- We will not use this

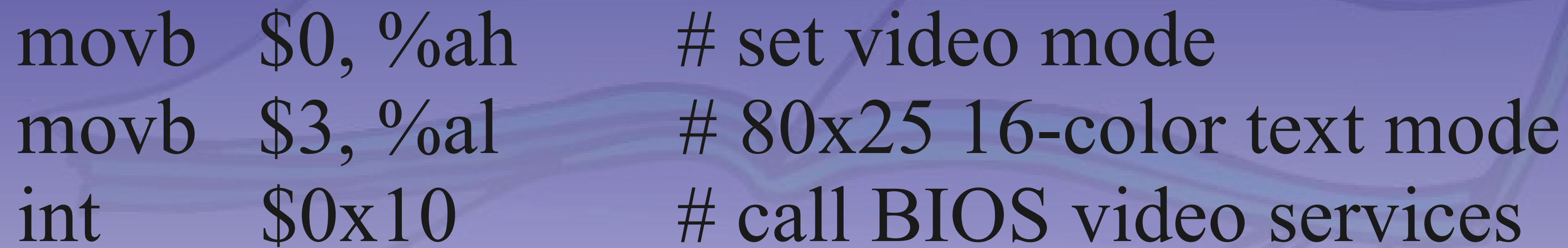
BIOS

- The operating system for the operating system
- Performs the bootup process
- Provides code to do many common operations
  - reduces the need to talk to hardware directly
- Chose one of the following:
  - Talk to the floppy controller directly: do a stepping of heads, timing, etc.
  - Issue a BIOS read sector command

BIOS and the OS

- Most OS's will try to ignore the BIOS
  - better performance
  - fewer restrictions
- This is not so simple
  - Different controllers, different protocols
  - May have to have different code for different manufacturer's floppy drive
- Boot needs to be more unified
  - Bootloader knowing IDE, SCSI, SATA, ATAPI???

Using the BIOS: Clear Screen



```
movb   $0, %ah        # set video mode
movb   $3, %al        # 80x25 16-color text mode
int    $0x10          # call BIOS video services
```

x86 Boot Up

- Wait for stable voltage
- Power-on self-test
- Execute video card BIOS
- Execute BIOS of other devices
  - this includes setting up port and memory mappings
- Find bootable media, load its bootsector
- Jump to boot
- This is where you come in

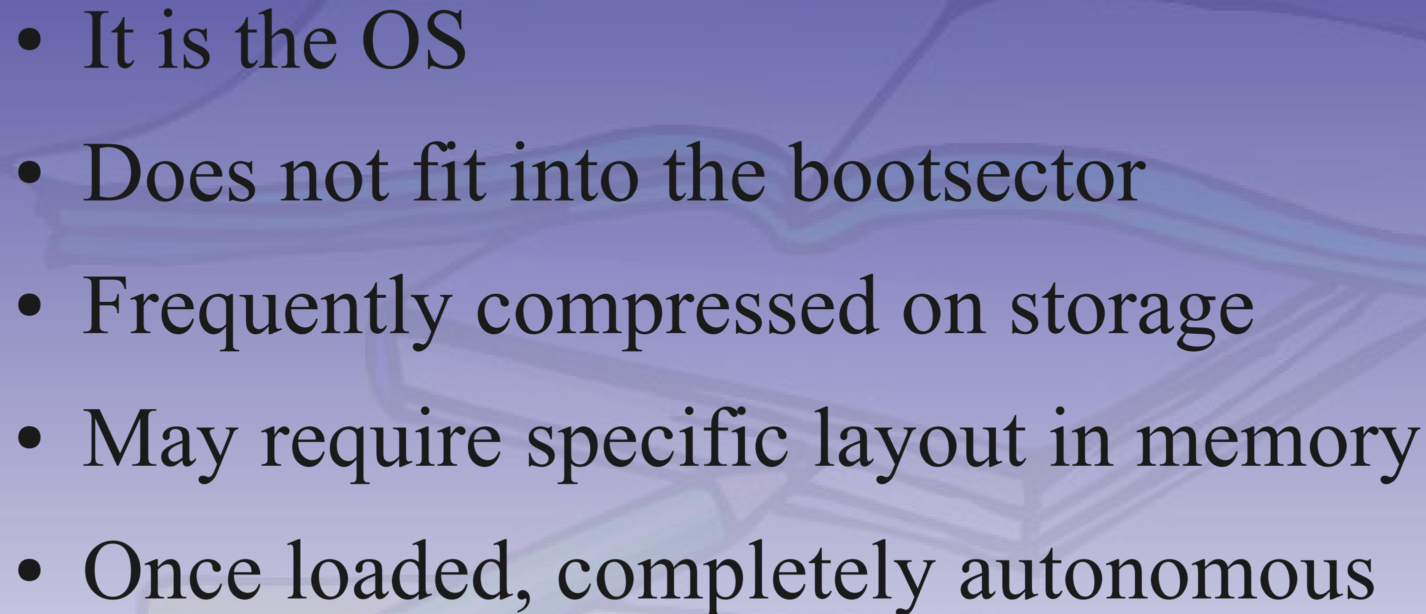
The Kernel and the Bootloader

- Why can the kernel not load itself? Or can it?
- The tasks of the bootloader:
  - Find and select proper kernels
  - Give user a choice of different booting mechanisms
  - Load the kernel into memory
  - Run it
- Restrictions
  - BIOS is stupid, only loads one sector
  - Bootloader is restricted to 510 bytes (or even less)

The bootsector

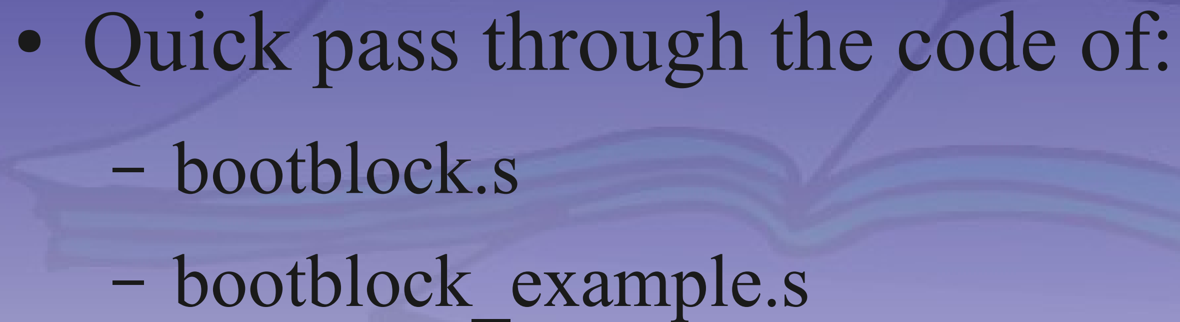
- 512 bytes
- Contains (in order):
  - bootloader code
  - (optional) partition table (64 bytes)
  - magic number: 0x55aa
- Placed in 0x07c00 before being executed
- Can be unloaded once the kernel is loaded

The Kernel



- It is the OS
- Does not fit into the bootsector
- Frequently compressed on storage
- May require specific layout in memory
- Once loaded, completely autonomous

Project1: bootsect.s



- Quick pass through the code of:
  - bootblock.s
  - bootblock_example.s

Embedding and Linking Format

- Unix standard for organizing binaries in a file
- Contains segment data, initial memory parameters, symbol table for linking and debugging, etc.
- Very nice for advanced binary file loaders
- `readelf` – utility for dumping ELF information
- `elf.h` – C library for working with ELF

Unix standard != BIOS usable

1980's redux

- Remember DOS's .com binaries?
  - Code, Data, and Stack in the same 64K
  - pushing onto the stack overwrites some of the code
- BIOS has not really evolved since then
  - execution begins at the first byte of bootsector
  - data is embedded into code
- GNU as only generates ELF binaries
- It is your job to take those ELF binaries and write out usable bootsector code

Making Bootable Binaries

- compiled bootloader in ELF in bootblock.o
- compiled kernel in kernel.o
- To form a complete bootable image:
  - strip ELF from bootblock, place it at byte 0
  - write magic number at byte 510
  - write partition table at byte 446 (optional)
  - at sector 1 (byte 512) place the kernel's code (strip ELF)
  - pad the kernel to a multiple of 512 bytes
  - make sure to tell the bootloader how big the kernel is

Project 1

- Begins at /c/cs421/as/1
- Update bootblock.s so that:
  - it writes something on the screen (useful for debug)
  - loads a kernel
  - sets up CS, DS, and SS
  - jumps into the kernel
- write createimage.c
  - needs to do exactly what createimage.given does
  - REMEMBER TO UPDATE THE MAKEFILE

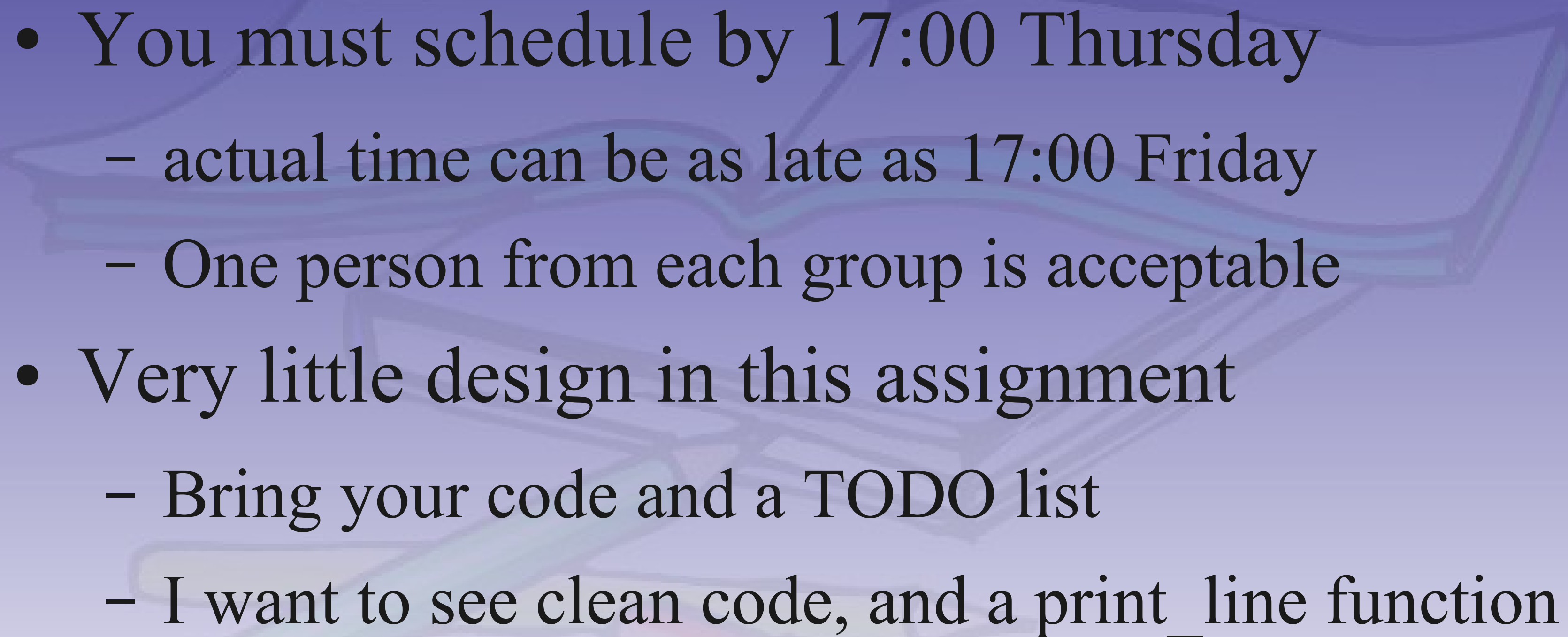
Tools Needed

- make / gcc
- bochs
- hex editor (Zoo has khexedit)
- cat /dev/random > /dev/sda
  - or dd if=your_image_goes_here of=/dev/sda
- Ctrl-alt-delete
- Read the documentation! Know the tools!

WARNING:

STUPIDITY CAUSES DATA LOSS

Design Review



- You must schedule by 17:00 Thursday
  - actual time can be as late as 17:00 Friday
  - One person from each group is acceptable
- Very little design in this assignment
  - Bring your code and a TODO list
  - I want to see clean code, and a print_line function

- Remember to apply for a zoo account:
  - http://zoo.cs.yale.edu/accounts.html
- Your OS needs to work on bochs
  - either as a floppy or a disk image is fine
- Due Thursday 2008-01-31 23:59
  - you have 72 hours of lateness available