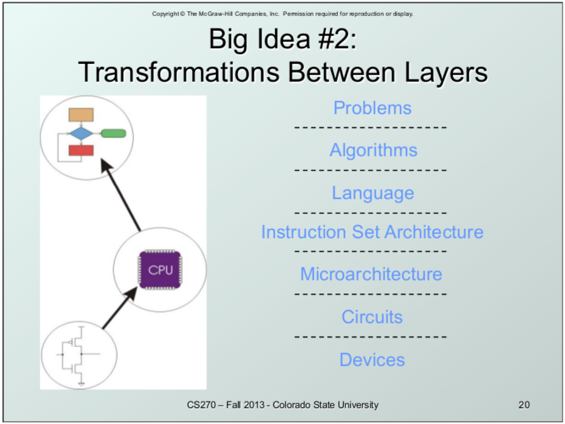
Writing an Operating System From Scratch

3 June 2015

Overview

1. Overview
2. Motivation
3. Memory
4. Stack
5. Load Disk
6. Screen Output
7. 16-bit Mode to 32-bit Mode
8. The Kernel
9. if anyone still not into whatsapp
10. In case anyone is still awake

The Big Picture



Big Idea #2: Transformations Between Layers

Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

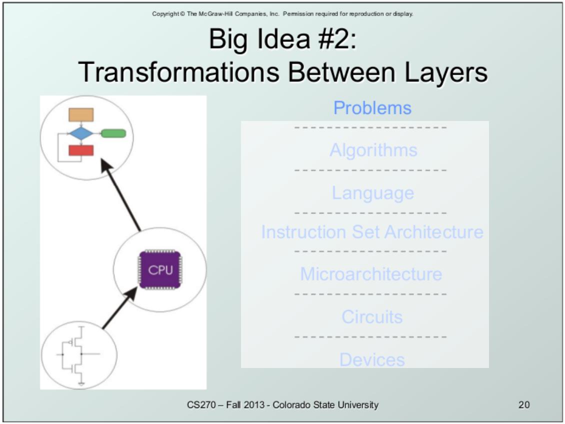
Circuits

Devices

CS270 – Fall 2013 - Colorado State University 20

Figur : From The Intro to Computing Systems

The Big Picture



Big Idea #2:
Transformations Between Layers

Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

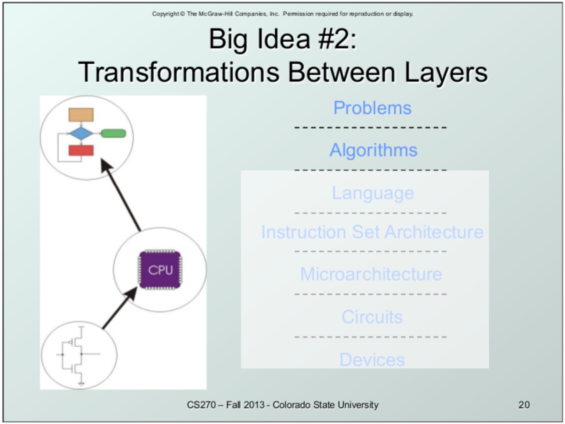
Circuits

Devices

CS270 – Fall 2013 - Colorado State University 20

Figur : From The Intro to Computing Systems

The Big Picture



Big Idea #2: Transformations Between Layers

Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

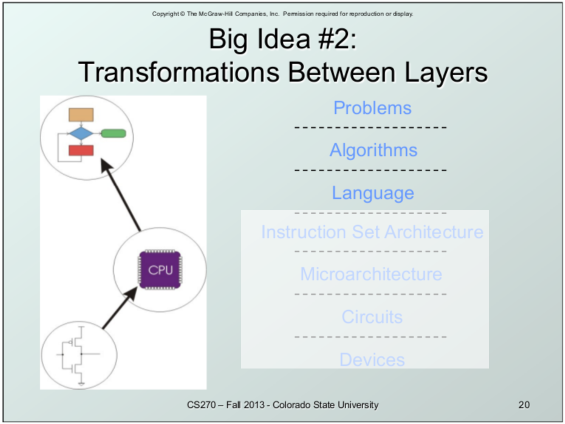
Circuits

Devices

CS270 – Fall 2013 - Colorado State University 20

Figur : From The Intro to Computing Systems

The Big Picture



Big Idea #2: Transformations Between Layers

Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

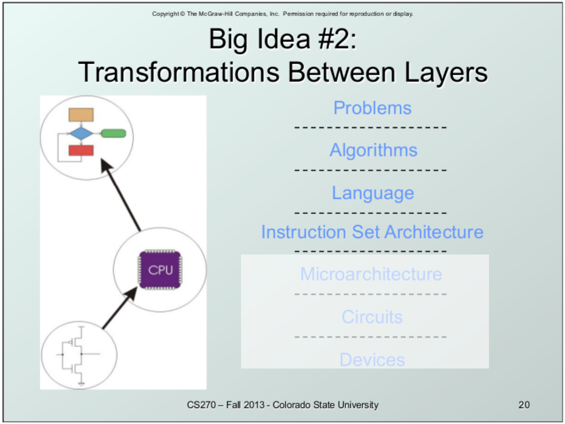
Circuits

Devices

CS270 – Fall 2013 - Colorado State University 20

Figur : From The Intro to Computing Systems

The Big Picture



Big Idea #2: Transformations Between Layers

Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

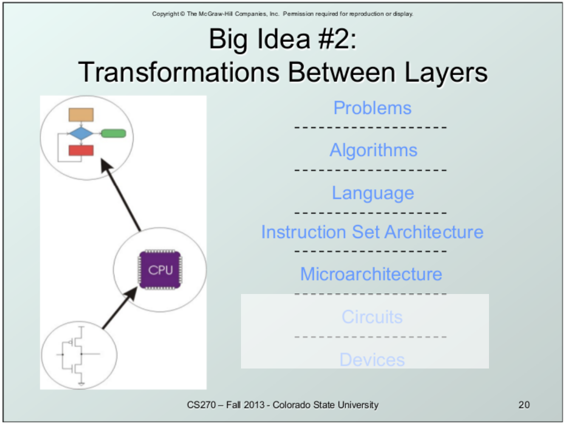
Circuits

Devices

CS270 – Fall 2013 - Colorado State University 20

Figur : From The Intro to Computing Systems

The Big Picture



Big Idea #2:
Transformations Between Layers

Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

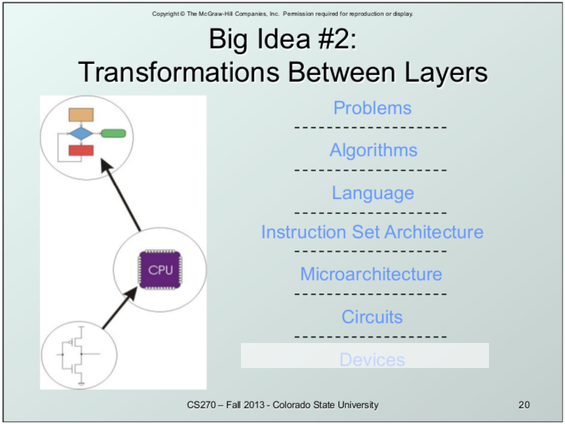
Circuits

Devices

CS270 – Fall 2013 - Colorado State University 20

Figur : From The Intro to Computing Systems

The Big Picture



Big Idea #2: Transformations Between Layers

Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

Circuits

Devices

CS270 – Fall 2013 - Colorado State University

20

Figur : From The Intro to Computing Systems

What happens when we switch on our PC?

BRIAN'S JUST BOOTING UP HIS NEW LAPTOP

-

The Boot Process

- BIOS
- POST
- Hardware and Memory Checks
- Low-level Tests

What happens when we switch on our PC?

BRIAN'S JUST BOOTING UP HIS NEW LAPTOP

-

The Boot Process

- BIOS
- POST
- Hardware and Memory Checks
- Low-level Tests

What happens when we switch on our PC?

BRIAN'S JUST BOOTING UP HIS NEW LAPTOP

-

The Boot Process

- BIOS
- POST
- Hardware and Memory Checks
- Low-level Tests

What happens when we switch on our PC?

BRIAN'S JUST BOOTING UP HIS NEW LAPTOP

-

The Boot Process

- BIOS
- POST
- Hardware and Memory Checks
- Low-level Tests

What happens when we switch on our PC?

-

BRIAN'S JUST BOOTING UP HIS NEW LAPTOP

The Boot Process

- BIOS
- POST
- Hardware and Memory Checks
- Low-level Tests

Basic Input Output Software

- Responsible for Booting the OS
- Needs an easy location to find our OS
- first sector of the hard disks (i.e. Cylinder 0, Head 0, Sector 0)

Question?

What if the Boot Sector is not present in the hard disk?

Basic Input Output Software

- Responsible for Booting the OS
- Needs an easy location to find our OS
- first sector of the hard disks (i.e. Cylinder 0, Head 0, Sector 0)

Question?

What if the Boot Sector is not present in the hard disk?

Basic Input Output Software

- Responsible for Booting the OS
- Needs an easy location to find our OS
- first sector of the hard disks (i.e. Cylinder 0, Head 0, Sector 0)

Question?

What if the Boot Sector is not present in the hard disk?

Basic Input Output Software

- Responsible for Booting the OS
- Needs an easy location to find our OS
- first sector of the hard disks (i.e. Cylinder 0, Head 0, Sector 0)

Question?

What if the Boot Sector is not present in the hard disk?

The Unique Identity for the Boot Sector

The last two bytes of an intended boot sector must be set to the magic number **0xaa55**

Two conditions to seize the system's reins:

- Recognize the boot sector
- Stay in it.

With this criteria, let us begin writing our **own** boot sector.

The Unique Identity for the Boot Sector

The last two bytes of an intended boot sector must be set to the magic number **0xaa55**

Two conditions to seize the system's reins:

- Recognize the boot sector
- Stay in it.

With this criteria, let us begin writing our **own** boot sector.

The Unique Identity for the Boot Sector

The last two bytes of an intended boot sector must be set to the magic number **0xaa55**
Two conditions to seize the system's reins:

- Recognize the boot sector
- Stay in it.

With this criteria, let us begin writing our **own** boot sector.

The Unique Identity for the Boot Sector

The last two bytes of an intended boot sector must be set to the magic number **0xaa55**
Two conditions to seize the system's reins:

- Recognize the boot sector
- Stay in it.

With this criteria, let us begin writing our **own** boot sector.

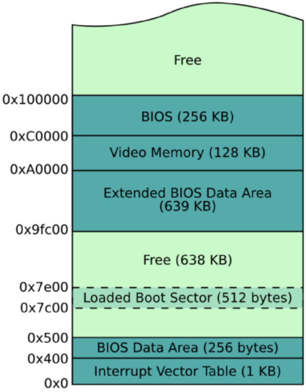
The Unique Identity for the Boot Sector

The last two bytes of an intended boot sector must be set to the magic number **0xaa55**
Two conditions to seize the system's reins:

- Recognize the boot sector
- Stay in it.

With this criteria, let us begin writing our **own** boot sector.

Memory Organization after boot


Free
0x100000
BIOS (256 KB)
0xC0000
Video Memory (128 KB)
0xA0000
Extended BIOS Data Area (639 KB)
0x9fc00
Free (638 KB)
0x7e00
Loaded Boot Sector (512 bytes)
0x7c00
0x500
BIOS Data Area (256 bytes)
0x400
Interrupt Vector Table (1 KB)
0x0

The BIOS places the boot sector at 0x7C00; which now becomes the global offset.

```
    [org 0x7c00]
```

Stack Usage

- The Base Pointer(BP) register stores the base address (i.e. bottom) of the stack.
- The Stack Pointer(SP) stores the top of the stack.
- The stack grows downwards from BP

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

Stack Usage

- The Base Pointer(BP) register stores the base address (i.e. bottom) of the stack.
- The Stack Pointer(SP) stores the top of the stack.
- The stack grows downwards from BP

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

Stack Usage

- The Base Pointer(BP) register stores the base address (i.e. bottom) of the stack.
- The Stack Pointer(SP) stores the top of the stack.
- The stack grows downwards from BP

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

Stack Usage

- The Base Pointer(BP) register stores the base address (i.e. bottom) of the stack.
- The Stack Pointer(SP) stores the top of the stack.
- The stack grows downwards from BP

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

Stack Usage

- The Base Pointer(BP) register stores the base address (i.e. bottom) of the stack.
- The Stack Pointer(SP) stores the top of the stack.
- The stack grows downwards from BP

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

When was 512 Bytes enough?

- Operating systems usually don’t fit into a single (512 byte) sector
- Instead they must *bootstrap* the rest of their code from the disk into memory
- BIOS provides us routine to load and read disks

When was 512 Bytes enough?

- Operating systems usually don't fit into a single (512 byte) sector
- Instead they must *bootstrap* the rest of their code from the disk into memory
- BIOS provides us routine to load and read disks

When was 512 Bytes enough?

- Operating systems usually don't fit into a single (512 byte) sector
- Instead they must *bootstrap* the rest of their code from the disk into memory
- BIOS provides us routine to load and read disks

When was 512 Bytes enough?

- Operating systems usually don't fit into a single (512 byte) sector
- Instead they must *bootstrap* the rest of their code from the disk into memory
- BIOS provides us routine to load and read disks

Disk Loading : BIOS Interrupt 0x13

- BIOS read sector function 0x02 = READ
- Read number of sectors specified by dh
- Select Cylinder 0
- Select Head 0
- Start reading from the sector after boot sector
- Error Checking Criteria

Disk Loading : BIOS Interrupt 0x13

- BIOS read sector function 0x02 = READ
- Read number of sectors specified by dh
- Select Cylinder 0
- Select Head 0
- Start reading from the sector after boot sector
- Error Checking Criteria

Disk Loading : BIOS Interrupt 0x13

- BIOS read sector function 0x02 = READ
- Read number of sectors specified by dh
- Select Cylinder 0
- Select Head 0
- Start reading from the sector after boot sector
- Error Checking Criteria

Disk Loading : BIOS Interrupt 0x13

- BIOS read sector function 0x02 = READ
- Read number of sectors specified by dh
- Select Cylinder 0
- Select Head 0
- Start reading from the sector after boot sector
- Error Checking Criteria

Disk Loading : BIOS Interrupt 0x13

- BIOS read sector function 0x02 = READ
- Read number of sectors specified by dh
- Select Cylinder 0
- Select Head 0
- Start reading from the sector after boot sector
- Error Checking Criteria

Disk Loading : BIOS Interrupt 0x13

- BIOS read sector function 0x02 = READ
- Read number of sectors specified by dh
- Select Cylinder 0
- Select Head 0
- Start reading from the sector after boot sector
- Error Checking Criteria

Disk Loading : BIOS Interrupt 0x13

- BIOS read sector function 0x02 = READ
- Read number of sectors specified by dh
- Select Cylinder 0
- Select Head 0
- Start reading from the sector after boot sector
- Error Checking Criteria

Printing : BIOS Interrupt 0x10

```
loop:
    mov al, [bx]
    cmp al, 0
    je out
    int 0x10
    add bx, 0x01
    jmp loop
```

The 32-bit Protected Mode

Why life gets complicated?

- Farewell BIOS and all its useful interrupts
- Need to manage a very complicated data structure called Global Descriptor Table

The 32-bit Protected Mode

Why life gets complicated?

- Farewell BIOS and all its useful interrupts
- Need to manage a very complicated data structure called Global Descriptor Table

The 32-bit Protected Mode

Why life gets complicated?

- Farewell BIOS and all its useful interrupts
- Need to manage a very complicated data structure called Global Descriptor Table

The 32-bit Protected Mode

The GD Table is essential to define segment and protected mode attributes

- Disable Interrupts(cli)
- Load GDT
- Update Segment Registers and Stack
- Go to a place in memory where you know legit-code in 32-bit mode is written

The 32-bit Protected Mode

The GD Table is essential to define segment and protected mode attributes

- Disable Interrupts(cli)
- Load GDT
- Update Segment Registers and Stack
- Go to a place in memory where you know legit-code in 32-bit mode is written

The 32-bit Protected Mode

The GD Table is essential to define segment and protected mode attributes

- Disable Interrupts(cli)
- Load GDT
- Update Segment Registers and Stack
- Go to a place in memory where you know legit-code in 32-bit mode is written

The 32-bit Protected Mode

The GD Table is essential to define segment and protected mode attributes

- Disable Interrupts(cli)
- Load GDT
- Update Segment Registers and Stack
- Go to a place in memory where you know legit-code in 32-bit mode is written

The 32-bit Protected Mode

The GD Table is essential to define segment and protected mode attributes

- Disable Interrupts(cli)
- Load GDT
- Update Segment Registers and Stack
- Go to a place in memory where you know legit-code in 32-bit mode is written

The Kernel

- Connects the Application to CPU/Memory/Devices
- Need for 32-bit Mode Code : protecting some kernels
- Performs Tasks such as executing processes and handling interrupts

Before we write kernel-code in C, we must understand how C-compilation works?

The Kernel

- Connects the Application to CPU/Memory/Devices
- Need for 32-bit Mode Code : protecting some kernels
- Performs Tasks such as executing processes and handling interrupts

Before we write kernel-code in C, we must understand how C-compilation works?

The Kernel

- Connects the Application to CPU/Memory/Devices
- Need for 32-bit Mode Code : protecting some kernels
- Performs Tasks such as executing processes and handling interrupts

Before we write kernel-code in C, we must understand how C-compilation works?

The Kernel

- Connects the Application to CPU/Memory/Devices
- Need for 32-bit Mode Code : protecting some kernels
- Performs Tasks such as executing processes and handling interrupts

Before we write kernel-code in C, we must understand how C-compilation works?

Life of a C Program : geek.c

```
int function()
{
  return 0xf00;
}
```

The Compilation Process

- gcc -ffreestanding -c geek.c -o geek.o
- objdump -d geek.o
- ld -o geek.bin -Ttext 0x0 –oformat binary geek.o

Life of a C Program

gcc -ffreestanding -c geek.c -o geek.o

The flag *-ffreestanding* is used to compile system-independent code.

objdump -d geek.o

The *objdump* command is used to see the machine code. It has debugging information, labels etc.

ld -o geek.bin -Ttext 0x0 –oformat binary geek.o

The *i386-elf-ld* links together all of the routines described in the input object files into one executable binary file

Entering the Kernel

A very simple assembly routine that is always attached to the start of the kernel machine code(or rather C code).
The sole purpose of the routine is to call the entry function of the kernel.

```
[bits 32]
[extern main]
```

Inline Assembly

Syntax of Inline Assembly

- Source and destination registers are switched from NASM
- Inputs and outputs are separated by colons

```
__asm__("in %%dx, %%al" : "=a" (result) : "d" (port)
```

Thriving in the Kernel(contd).

Over to Terminal