2. Bare-Metal Application

A paranoid professor wants to develop a program that displays the word count of typed input.
SPECS:

- desktop (non-network)
- word = [A-Za-z]+
- output = [0-9]+
- program runs on boot

How does booting work?

x86 Boot Procedure

CPU RAM is cleared on reset...how can we load the program?

- EEPROM is expensive and static so we can’t hold our program or kernel there
    - in desktops, it is read-only & hold firmware & bootstrap location

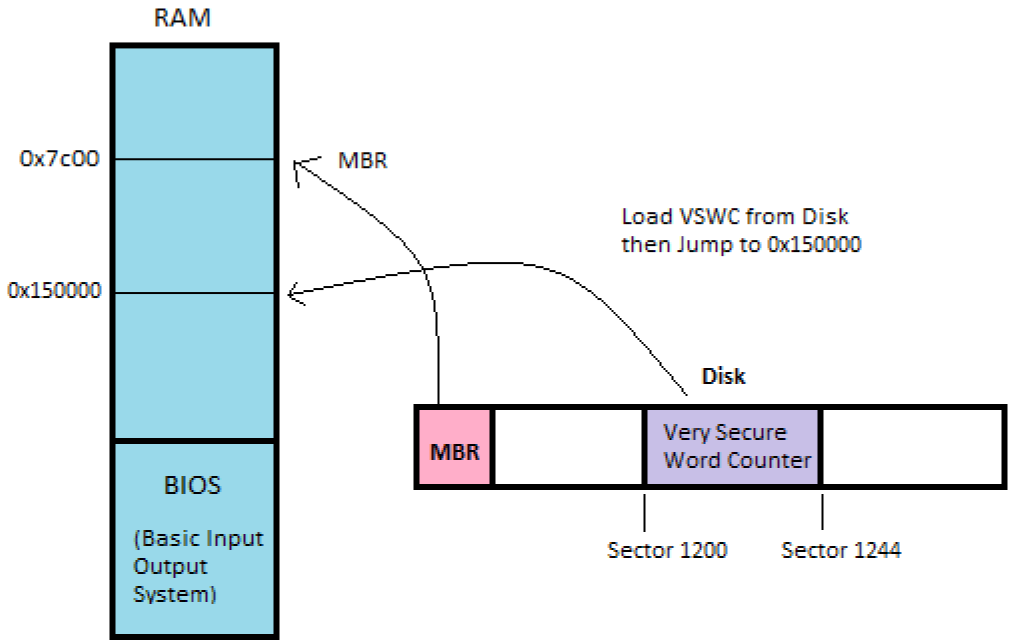
We use GUID (Globally Unique Identifier) to identify disk partitions

- 128-bit quantity
- without these IDs, firmware won't know if it changed or not
- held in GUID partition table (GPT)

Basic <u>Bootstrapping CPU Process</u>

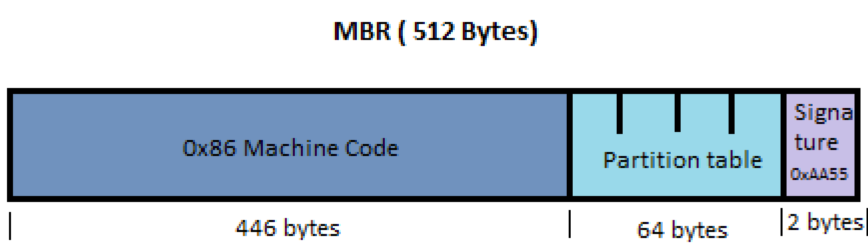
a. all registers start at 0 (which means kernel mode)
b. BIOS (firmware) sets up from EEPROM & jumps to kernel
c. kernel sets up its own stack and preloads some text to certain domains

firmware------> MBR (OS-agnostic)------> VBR (OS-specific)------> kernel------> app



BIOS Procedure:

1. self test for zeroes (hardware sanity check)
2. scan for MBR
3. load MBR into 0x7C00
   a. if VBR is present, load them
4. set IP to 0x7C00



MBR Procedure:

i. BIOS looks for a sector with the signature 0x55AA (this is the Master Boot Record)
ii. MBR searches for a Volume Boot Records for each device
iii. MBR then bootstraps to the kernel (MBR + VBRs)

With this knowledge, we can write our boot software:

```
// this is not called; it runs on boot
void main(void) {
  // we read 80 sectors of 512 bytes -- 40 KiB
  for (int i = 0; i < 80, i++)
    read_ide_sector(i+100, 0x2000 + 512*i);
  // jump to the first instruction
  goto *0x2000;
}
```

We use the following input/output primitives:

```
#include <sys/io.h>
// CPU send signal to disk controller via bus
// disk controller sends back data from disk
// retrieve address 'a' (bus address) from disk
// this instruction is slow because signals travel on bus
unsigned char inb(unsigned short int port) { asm("...") };
// get data from port with address "port"
void outb(unsigned char value, unsigned short int port);
// write a byte of data "value" to port "port" (hardware specific
void insl(unsigned short int port, void *addr, unsigned long int count);
// read "count" bytes from "port" to "addr"
```

status/cmd	sector #	sector count	data
1f7	1f6 ~ 1f3	1f2	1f1 ~ 1f0

Drive layout:

read_ide_sector protocol:

1. inb from 0x1F7 (status register) to check if controller is ready
2. outb to 0x1F2 (parameter 1 register) to give number of sectors to be read
3. outb to 0x1F3-0x1F6 (parameter 2 register) to give 32 bit sector offset...4 writes
4. outb to 0x1F7 (status register) a bit pattern telling controller we want to READ
5. inb from 0x1F7 (status register) to check if data is ready for copying
6. insl from 0x1F0 (device cache) 128 bytes of data

```
void read_ide_sector(int sector, int address) {
  // poll for readiness (1)
  while ((inb(0x1F7) & 0xC0) != 40) continue;
  // tell the controller we want 1 sector (2)
  outb(0x1F2, 1);
  // tell the controller where sector is (3)
  for (int i = 0; i < 4; i++) outb(0x1F3+i, sector>>(8*i) & 0x
FF);
  // tell the controller we want to read (0x20=READ) (4)
  outb(0x1F7, 0x20);
  // wait for data to be ready for copying (5)
  while ((inb(0x1F7) & 0xC0) != 40) continue;
  // copy 128 bytes of data to addr from cache
  insl(0x1F0, address, 128);
}
```

Now we need a function to display the results:

- the screen pointer is represented by (base) + (row) + (column): [80]x[25]
- this utilizes memory mapped IO, no programmed IO, so it is not a bottleneck
- 16 bit quantity with low order as ASCII character and high order as appearance

```
void display(long long nwords) {
  short *screen = (short*) 0xb8000 + 80*25/2 - 80/2);
  do {
    screen[0] = (nwords % 10) + '0';
    screen[1] = 7; // gray on black
```

```
    screen -= 2;
  } while ((nwords/=10) != 10);
}
```

Now we have our utilities… let's implement!

```
// at addr 0x2000 jumped to by boot protocol
void main(void) {
  // 1TB > 2^31, so we use a 64 bit digit
  long long int nwords = 0;
  // bool for starting in the middle of a word on line
  int len, s = 50000;
  do {
    char buf[513];
    buf[512] = 0;
    len = strlen(buf);
    read_ide_sector(s++, (int) buf);
    nwords += cws(buf, len, &inword);
  } while (len == 512);
  display(nwords);
}
```

Now we only need the actual word count…

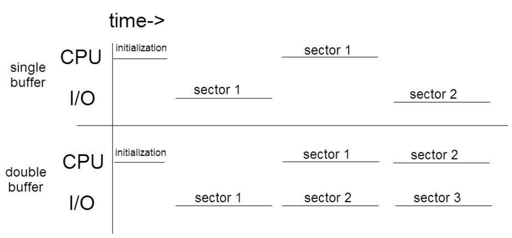
```
int cws(char *buf, int bufsize, bool *inword) {
  int w = 0;
  for (int i = 0; i < bufsize, i++) {
    bool alpha = isalpha((unsigned char)buf[i])'
    w += alpha & !*inword;
    *inword = isalpha(buf[i]);
  }
  return w;
}
```

& we are done! our problem has a few issues, however…

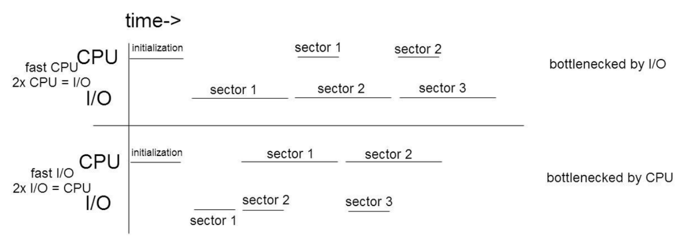
a. Duplication of Code
    i. BIOS must already do RAM reading, but we re-wrote it by hand
    ii. → Code is not easily reusable
b. VERY special purpose—not generalizable
    i. what if we wanted to boot with UEFI instead?
c. Inefficient
    i. We spend a long time waiting
        1. We could use yield() instead
    ii. insl() chews up the bus
    iii. copy disk to CPU to RAM
d. Faults crash the entire CPU

We can fairly easily increase efficiency using double buffering

- we load the next output data while the first is being printed



CPU and I/O speed roughly equivelant



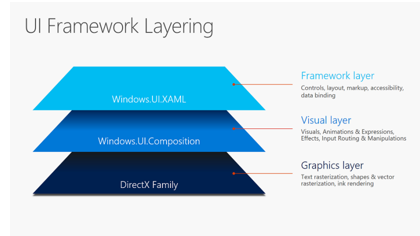
I/O and CPU speed vary

This can nearly double the speed of the program!

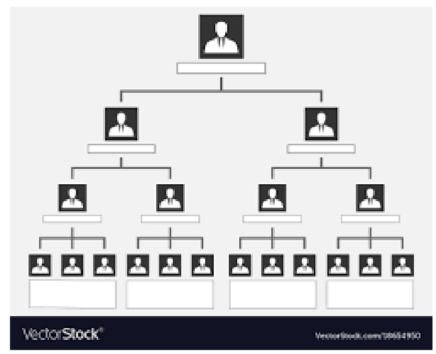
The one-piece solution clearly has many faults; we don't utilize some of our best tools:

1. **Modularity**
    a. a divide-and-conquer approach
    b. A system is divided into interacting subsystems called **modules**
        i. Effects to subvert modular borders can cause effects to propagate
    c. these modules communicate through interfaces
2. **Abstraction**
    a. the ability to treat others entirely based on external specs
    b. is based on of the quality of the interface
3. **Layering**

   a. a system which has layers of modules which can only interact with modules of a distance <= 1 layer from itself



4. **<u>Hierarchy</u>**
   a. a system which has subsystems assembled upon self-contained subsystems
   b. Contains the number of interactions between N elements to N*(N-1)
5. **<u>Iteration</u>**
   a. starting with a simple system which fulfills some of the spec, then adding more.
   b. Makes it easier to catch bugs and make adjustments



We can defeat the repeating of code using Modularity!

But what are the benefits of this? Well..

~ Let's say the number of bugs in a program **α** N &
that the cost to fix a bug **α** N , where N is the number of lines in a program

→ the time to debug is O(N^2), but breaking it into K modules makes the time O(N^2/K)

NOTE: In reality, this is a simplification, since it assumes all bugs are 100% local, but the generalization still applies as we approach perfect modularity

We need some metrics with which to judge the quality of modularity