What is ‘bootstrapping’?

The process of loading larger programs for execution than can fit within a single disk-sector

Serious Pentium explorations

- Experimenting with most Pentium features will require us to write larger-size demos than can fit in one disk-sector (512 bytes)
- So we need a way to load such programs into memory when no OS is yet running
- And we need a convenient way to place such programs onto a persistent storage medium so they can easily be accessed

Our classroom setup

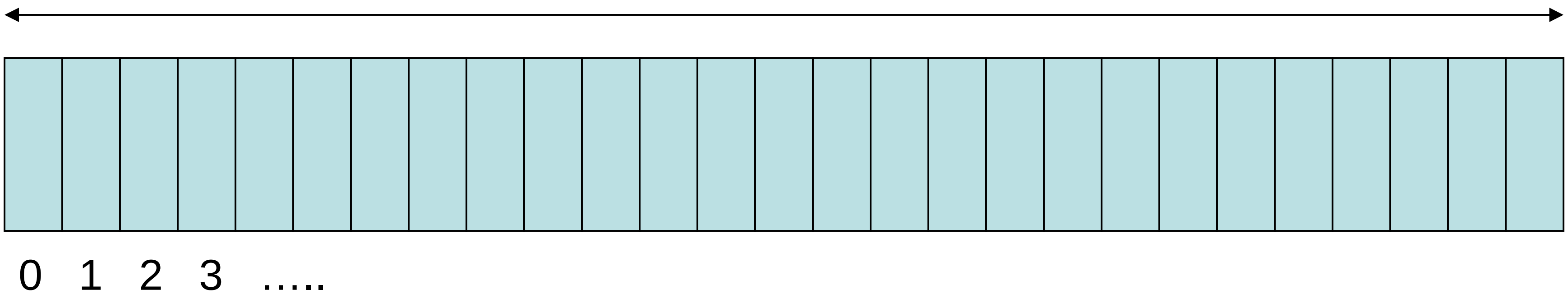
- Our workstations’ hard disks have been ‘partitioned’ in way that provides a large unused storage-area for us to use freely
- But other portions of these hard disks are dedicated to supporting vital courseware for students who are taking other classes
- We have to understand how to access our ‘free’ area without disrupting anyone else

Fixed-Size ‘blocks’

- All data-transfers to and from the hard disk are comprised of fixed-size blocks called ‘sectors’ (whose size equals 512 bytes)
- On modern hard disks, these sectors are identified by sector-numbers starting at 0
- This scheme for addressing disk sectors is known as Logical Block Addressing (LBA)
- So the hard disk is just an array of sectors

Visualizing the hard disk

A large array of 512-byte disk sectors



Disk storage-capacity (in bytes) = (total number of sectors) x (512 bytes/sector)

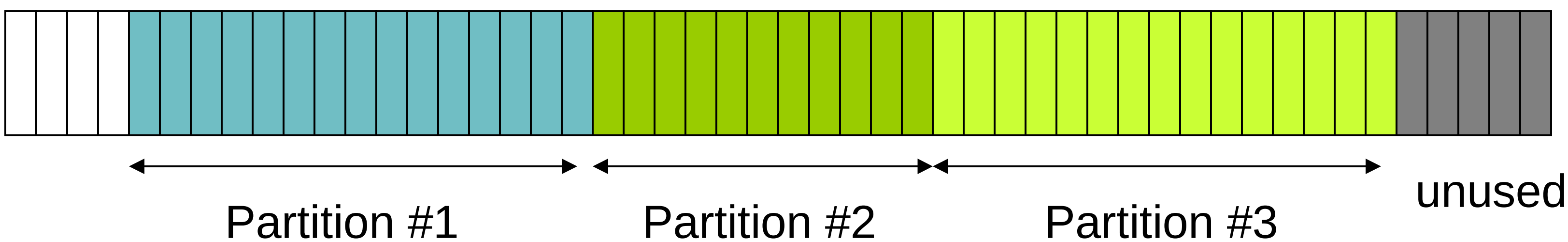
Example: If disk-capacity is 160 GigaBytes, then the total number of disk-sectors can be found by division:

(160000000000 bytes) / (512 bytes-per-sector)

assuming that you have a pocket-calculator capable of displaying enough digits!

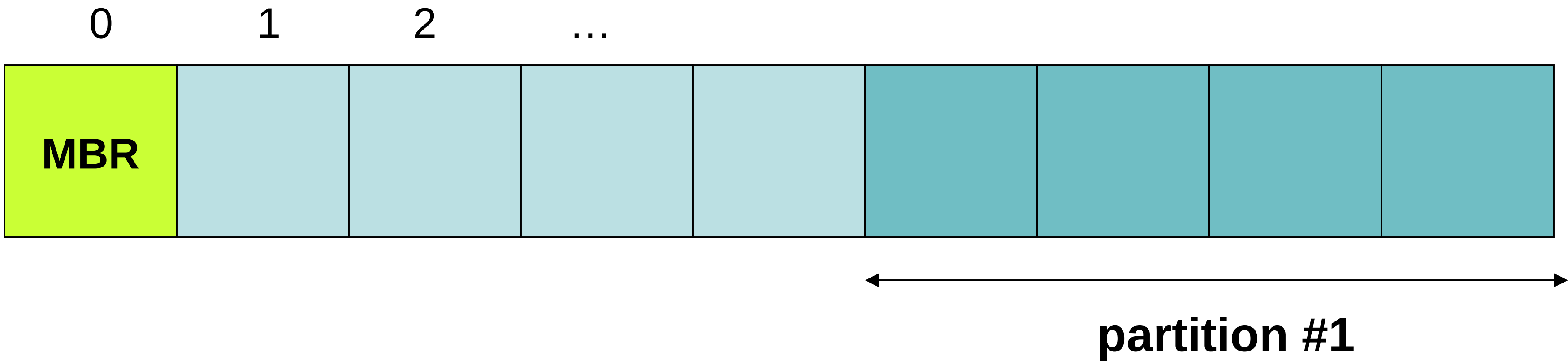
Disk Partitions

- The total storage-area of the hard disk is usually subdivided into non-overlapping regions called ‘disk partitions’



Master Boot Record

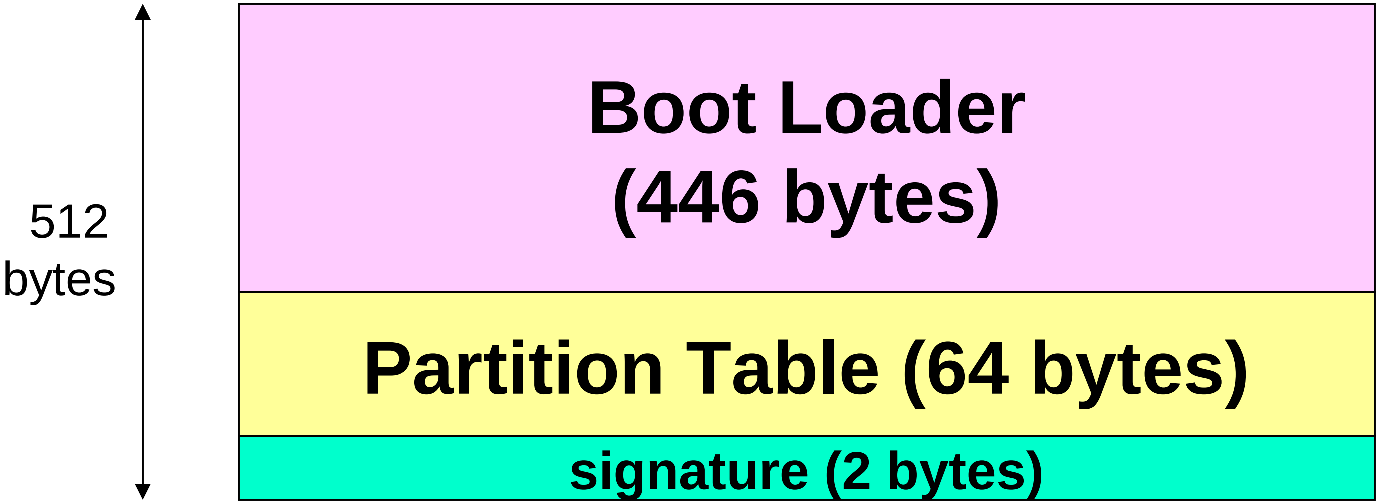
- A small area at the beginning of the disk is dedicated to ‘managing’ the disk partitions



- In particular, sector number 0 is known as the Master Boot Record (very important!)

Format of the MBR

- The MBR is subdivided into three areas:
  - The boot loader program (e.g., GRUB)
  - The 'partition table' data-structure
  - The MBR signature (i.e., 0x55, 0xAA)

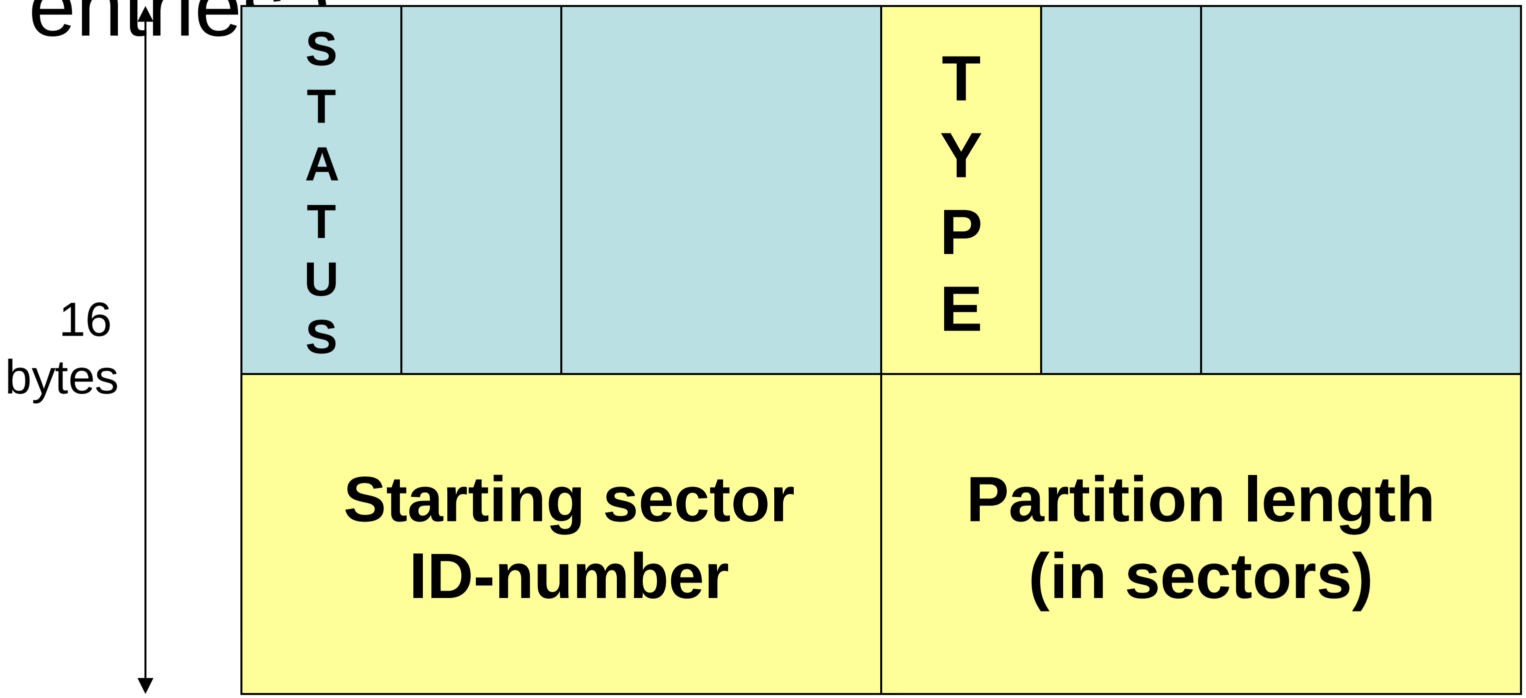


‘Reading’ the MBR

- To see the hard disk’s Partition Table, we must ‘read’ the entire Master Boot Record
- (We ignore the boot-loader and signature)
- But we will need to understand the format of the data stored in that Partition Table
- We first need to know how to devise code that can transfer the MBR (sector 0) from the hard-disk into a suitable memory-area

Partition Table Entries

- The MBR is an array containing four data-structures (called ‘partition-table entries’):



Some fields contain ‘obsolete’ information

TYPE-ID

- Each partition-table entry has a TYPE-ID
  - TYPE-ID is 0x07 for a ‘Windows’ partition
  - TYPE-ID is 0x83 for our ‘Linux’ partition
  - TYPE-ID is 0x00 when the entry is ‘unused’
- You can find a list of TYPE-ID numbers posted on the internet (see our website)
- Our disks have an extra ‘Linux’ partition that nobody else is using this semester

BIOS Disk Drive Services

- An assortment of disk-access functions is available under software Interrupt 0x13
- Originally there were just six functions (to support IBM-PC floppy diskette systems)
- More functions were added when PC/XTs introduced the use of small Hard Disks
- Now, with huge hard disk capacities, there is a set of “Enhanced Disk Drive” services

Phoenix Technologies Ltd

- You can find online documentation for the BIOS EDD specification 3.0 (see website)
- We’ll use function 0x42 to read the MBR
- It requires initializing some fields in a small data-structure (the “Disk-Address Packet”)
- Then we load parameters in four registers (DS:SI = address of the DAP, DL = disk-ID and AH = 0x42) and execute ‘int $0x13’

EDD Disk-Address Packet

<table>
<tr><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="2">segment-address of transfer-area</td><td colspan="2">offset-address of transfer area</td><td>reserved (=0x00)</td><td>sector count</td><td>reserved (=0x00)</td><td>packet length</td></tr>
<tr><td colspan="8">Logical Block Address of disk-sector (64-bits)</td></tr>
<tr><td colspan="8">Physical-address of memory transfer-area (64-bits)
(in case segment:offset above is 0xFFFF:FFFF)</td></tr>
</table>

The MBR parameters

Here are assembly language statements that you could use to create a Disk Address Packet for reading the hard-disk's Master Boot Record into the memory-area immediately following the 512-byte BOOT_LOCN area

```
#----------------------------------------------------------------------------------
packet:   .byte    16, 0                # packet-size = 16 bytes
          .byte    1, 0                 # sector-count = 1 sector
          .word    0x0200, 0x07C0       # transfer-area's address
          .quad    0                    # MBR's Logical Block Address
#----------------------------------------------------------------------------------
```

Our demo-program (named 'finalpte.s') uses statements similar to these.

How we search the Partition Table

- Our demo-program ‘finalpte.s’ locates the final valid entry in the MBR Partition Table
- It displays the contents of that entry (i.e., four longwords) in hexadecimal format
- To do its search, it simply scans the table entries in backward order looking for the first entry that has a nonzero ‘type’ code

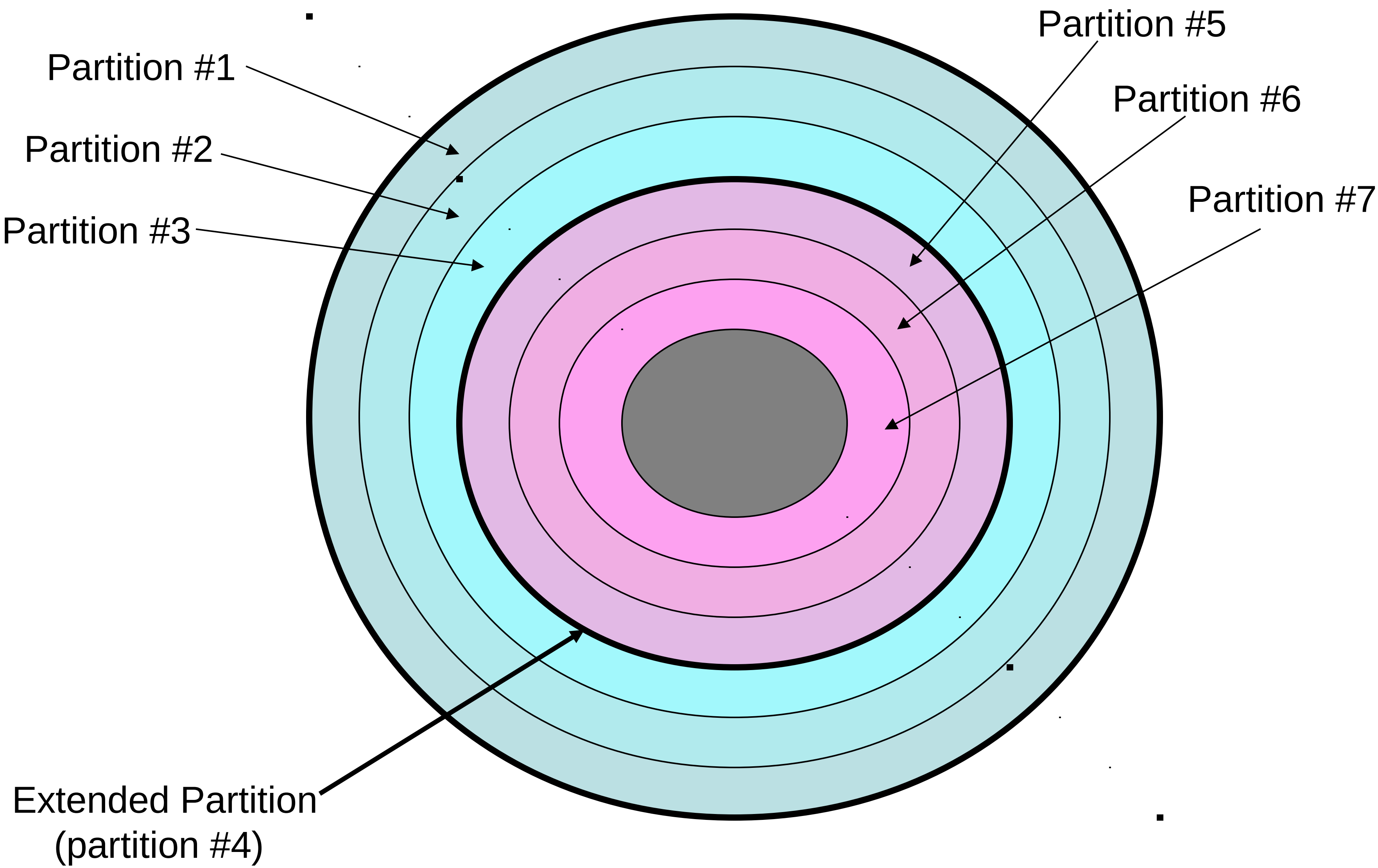
Instructions we could use

```
        mov     $0x03FE, %si        # point DS:SI to signature-word
        mov     $4, %cx             # setup count of table’s entries
nxpte:
        sub     $16, %si            # back up to the previous entry
        cmpb    $0x00, 4(%si)       # entry’s type-code is defined?
        loope   nxpte               # no, examine the next entry

        jcxz    nopte               # search fails if CX reached 0

        #  If we get here, then DS:SI is pointing to the final valid PT-entry
        jmp     found

nopte:  #  We should never arrive here -- unless no valid partitions exist
```

‘Extended’ partitions

- The hard-disk’s Master Boot Record only has room for four Partition-Table Entries
- But some systems need to use more than four disk-partitions, so a way was devised to allow one of the MBR’s partition-table entries to describe an ‘extended’ partition
- This scheme does not seem to have been ‘standardized’ yet -- hence, confusion!

The Linux scheme



Our ‘cs686ipl.s’ boot-loader

- We created a ‘boot-loader’ that will work with Linux systems that have ‘extended’ disk-partitions (“Initial Program Loader”)
- It uses the EDD Read_Sectors function, and it will read 127 consecutive sectors from the disk’s final Linux-type partition
- It transfers these disk-sectors into the memory-arena at address 0x00010000

Our ‘controls.s’ demo-program

- To demonstrate our boot-loader, we wrote a short program that can be ‘loaded’ into memory at 0x10000 and then executed
- It will display some useful information (and thereby verify that our boot-loader worked)
- Our boot-loader requires that a special ‘program signature’ (i.e., 0xABCD) must occupy the first word of any program it attempts to execute (as a ‘sanity’ check)

Depiction of ‘boot-strapping’

We install our ‘cs686ipl.b’ loader into the boot-sector of disk-partition number 4:

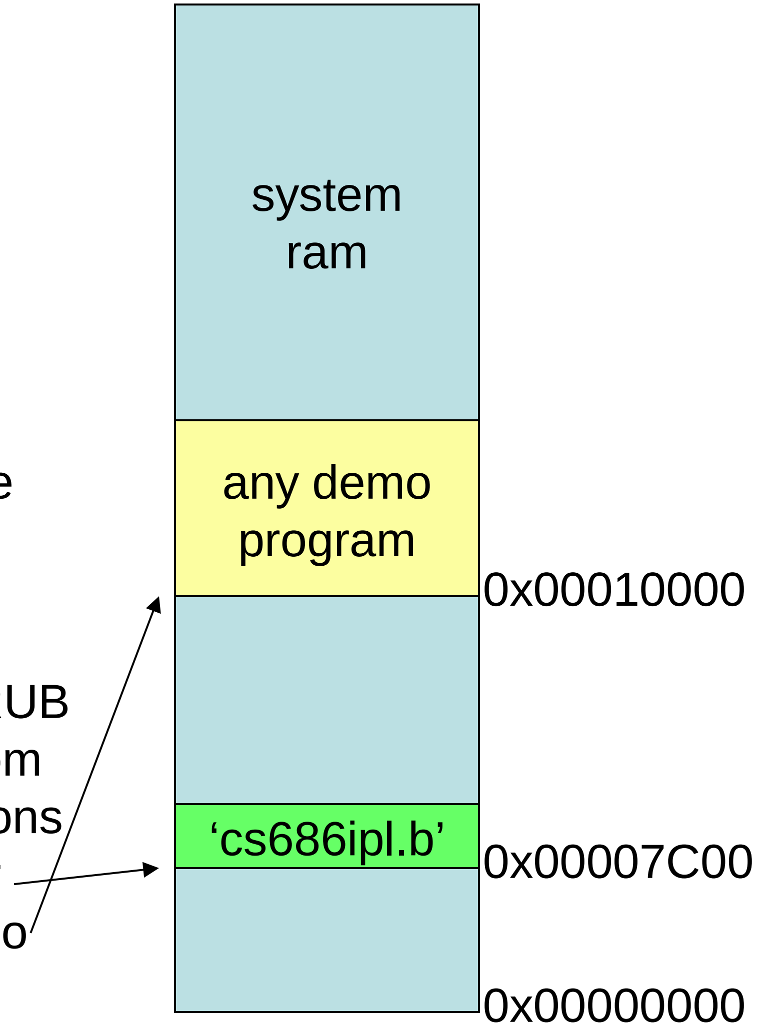
```
$ dd  if=cs686ipl.b  of=/dev/sda4
```

We install our ‘controls.b’ demo-program into the subsequent disk-sectors:

```
$ dd  if=controls.b  of=/dev/sda4  seek=1
```

Then we ‘reboot’ our machine to begin the bootstrapping process…

Step 1: The ROM-BIOS firmware loads GRUB
Step 2: The user selects a disk-partition from the GRUB program’s menu’s options
Step 3: GRUB loads our ‘cs686ipl.b’ loader
Step 4: ‘cs686ipl.b’ loads our program demo



In-class exercise #1

- Install the ‘cs686ipl.b’ boot-loader on your assigned ‘anchor’ machine:

  $ dd if=cs686ipl.b of=/dev/sda4

- Also install the ‘controls.b’ demo-program on your assigned ‘anchor’ machine:

  $ dd if=controls.b of=/dec/sda4 seek=1

- Then use the ‘fileview’ utility-program (from our class website, under ‘Resources’) to view the first few disk-sectors in partition number 4:

  $ fileview /dev/sda4

In-class exercise #2

- Try ‘rebooting’ your ‘anchor’ machine, to see the information shown by ‘controls.b’
  - Use ‘ssh’ to log on to the ‘colby’ gateway
  - Use ‘telnet’ to log onto your ‘anchor’ machine
  - Use the ‘sudo reboot’ command to reboot and then watch for the GRUB menu-selection that will launch the ‘boot-strapping’ process which will execute the demo-program you installed

In-class exercise #3

- Can you apply what you’ve learned in our prior exercises to enhance our ‘controls.s’ demo-program so that it will display some additional information about the CPU’s register-values at boot-time?
- For example, could you display the values held in the GDTR and IDTR registers? (Remember those are 48-bit registers)