3.5 Booting

Boot

- Came from the idiomatic expression, "to pull yourself up by your bootstraps"
  - without the help of others
  - it’s a difficult situation
- In OS
  - load its OS into memory
    - which kind of means that you need an OS in memory to do it
- Solution
  - load a tiny OS into memory
    - known as the ***bootstrap loader***
    - then again, who loads this tiny OS into memory?
      - how about first loading a tiny bootstrap loader?

PDP-8

toggle switches

- How about manually put into memory a simple bootstrap loader?
  - approach taken by PDP-8
    - "toggles in" the program
  - read OS from paper tape

PDP-8 Boot Code

```
07756 6032 KCC
07757 6031 KSF
07760 5357 JMP .-1
07761 6036 KRB
07762 7106 CLL RTL
07763 7006 RTL
07764 7510 SPA
07765 5357 JMP 7757
07766 7006 RTL
07767 6031 KSF
07770 5367 JMP .-1
07771 6034 KRS
07772 7420 SNL
07773 3776 DCA I 7776
07774 3376 DCA 7776
07775 5356 JMP 7756
07776 0000 AND 0
07777 5301 JMP 7701
```

VAX-11/780

VAX-11/780 Boot

- Separate "console computer"
  - LSI-11
  - hard-wired to always run the code contained in its on-board read-only memory
  - then read boot code (i.e., the bootstrap loader) from floppy disk
  - then load OS from root directory of first file system on primary disk

- Code on floppy disk (the bootstrap loader) would handle:
  - *disk device*
  - *on-disk file system*
  - it needs the right *device driver*
  - it needs to know how the disk is setup
    - what sort of *file system* is on the disk
    - how the disk is *partitioned*
      - a disk may hold multiple and different file systems, each in a separate partition

Configuring the OS

- Early Unix
  - OS statically linked to contain all needed device drivers
    - device drivers were statically linked to the OS
  - all device-specific info included with drivers
  - *disk drivers* contained *partitioning description*
  - therefore, the following actions may all require compiling a new version of the OS:
    - adding a new device
    - replacing a device
    - modifying disk-partitioning information

Configuring the OS

- **Later Unix**
  - **OS statically linked to contain all needed device drivers**
  - **at boot time, OS would *probe* to see which devices were present and discover device-specific info**
  - **partition table in first sector of each disk**

- **Even later Unix**
  - **allowed device drivers to be *dynamically loaded* into a *running system***

IBM PC

Issues

- Open architecture
  - although MS-DOS was distributed in binary form only
  - large market for peripherals, most requiring special drivers
  - how to access boot device?
  - how does OS get drivers for new devices?

The Answer: BIOS

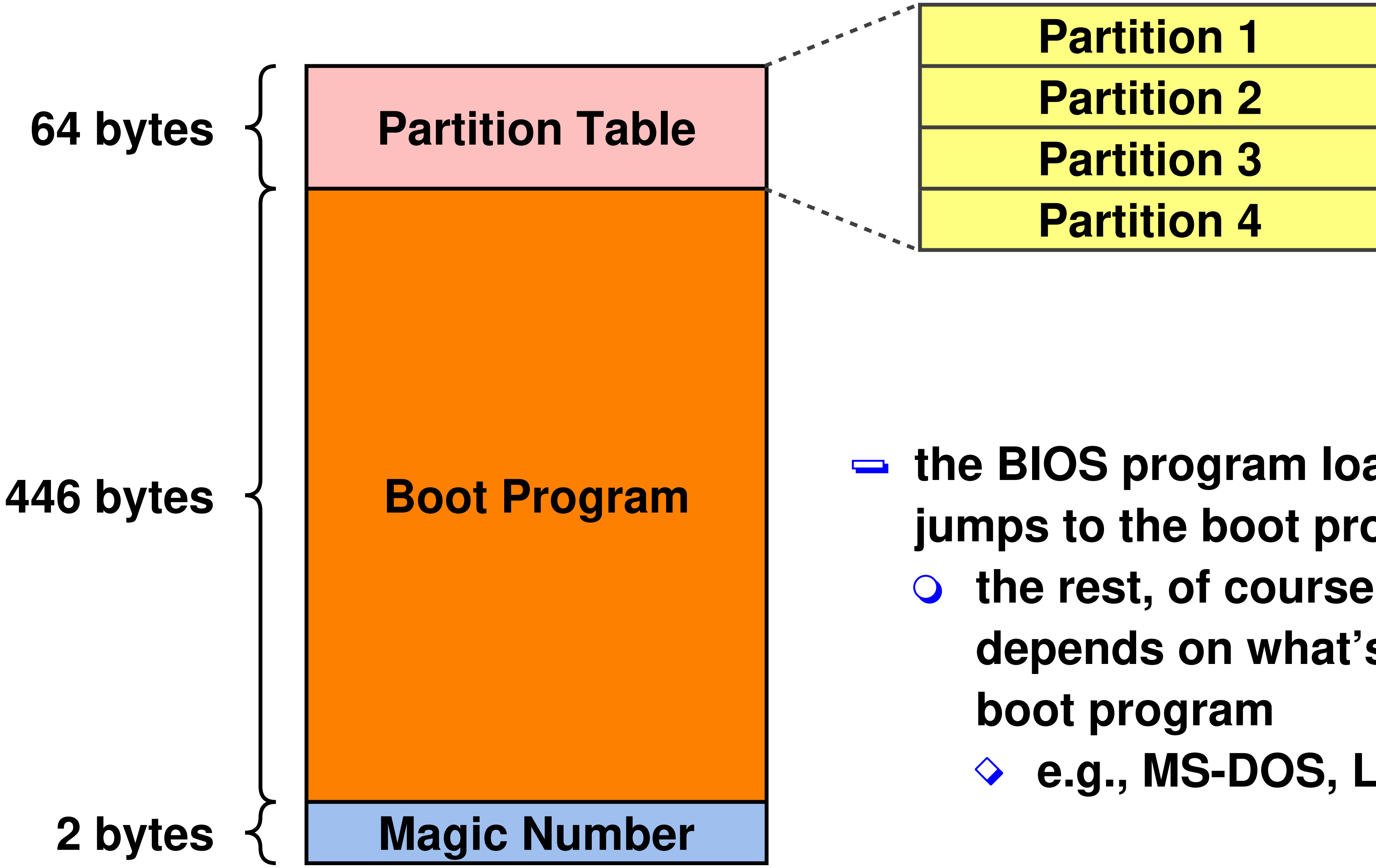
- Basic Input-Output System *(BIOS)*
  - code stored in read-only memory (ROM)
  - configuration data in non-volatile RAM (NVRAM)
    - such as *CMOS*
    - including set of boot-device names
  - the BIOS provides three primary functions
    - power-on self test *(POST)*
      - so it knows *where* to load the boot program
    - load and transfer control to boot program
    - provide *drivers* for all devices

- Main BIOS on motherboard
  - supplied as a chip on the "motherboard"
  - contains everything necessary to perfrom the above 3 functions
  - additional BIOSes on other boards
    - provide access to additional devices

POST

- On power-on, CPU executes BIOS code
  - located in last 64KB of first megabyte of address space
    - starting at location 0xf0000
    - CPU is hard-wired to start executing at 0xffff0 on startup
      - the last 16 bytes of this region
      - jump to POST
- POST
  - initializes hardware
  - *counts* memory locations
    - by testing for working memory
- Next step is to find a boot device
  - the CMOS is configured with a boot order
- Next step is to load the Master Boot Record *(MBR)* from the first sector of the boot device, if it's a floppy/diskette
  - or cylinder 0, head 0, sector 1 of a hard disk (Ch 6)

Getting the Boot Program



Master Boot Record (MBR)

- the BIOS program loads and jumps to the boot program
  - the rest, of course, depends on what's in the boot program
    - e.g., MS-DOS, Linux

MS-DOS Boot Program

- One of the hard drive partitions is labeled as the *active* partition
- The MS-DOS boot program finds the active partition
  - loads the first sector from it
    - which contains the "volume boot program"
  - pass control to that program
    - which then load the OS from that partition

Linux Booting (1)

- Two stages of booting provided by one of:
  - *lilo* (Linux Loader)
    - uses sector numbers of kernel image
    - therefore, must be modified if a kernel image moves
  - *grub* (Grand Unified Boot Manager)
    - understands various file systems
    - can find a kernel image given a file system *path name*
  - both allow dual (or greater) booting
  - select which system to boot from menu
    - perhaps choice of Linux or Windows
- The next step is for the kernel to *configure* itself

Linux Booting (2)

assembler code (startup_32)

- Kernel image is compressed
  - step 1: set up stack, clear BSS, uncompress kernel, then transfer control to it

assembler code (different startup_32)

- *Process 0* is created
  - step 2: set up initial page tables, turn on *address translation* (Ch 7)
  - process 0 knows how to handle some aspects of paging

C code (start_kernel)

- Do further initialization
  - step 3: initialize rest of kernel, create the "init" process (i.e., *process 1*, which is the ancestor of all other user processes)
  - invoke the *scheduler*

- Your kernel 1 assignment starts at step 3 above
  - NOTE: `weenix` is not exactly Linux

BIOS Device Drivers

- Originally, the BIO provided drivers for all devices
  - OS would call BIOS-provided code whenever it required services of a device driver
- These drivers sat in low memory and provided minimal functionality
  - later systems would copy them into primary memory
  - even later systems would provide their own drivers
  - nevertheless, BIO drivers are still used for booting
    - how else can you do it?

Beyond BIOS

- BIOS
  - designed for 16-bit x86 of mid 1980s
  - not readily extensible to other architectures
- Open Firmware
  - designed by Sun
  - portable
  - drivers, boot code in Forth
    - compiled into *bytecode*
- Intel developed a replacement for BIOS called *EFI (Extensible Firmware Interface)*
  - also uses *bytecode*

Ch 4: Operating-System Design

Bill Cheng

http://merlot.usc.edu/william/usc/

OS Design

- We will now look at how OSes are constructed
  - what goes into an OS
  - how they interact with each other
  - how is the software structured
  - how performance concerns are factoered in
- We will introduce new components in this chapter
  - scheduling (Ch 5)
  - file systems (Ch 6)
  - virtual memory (Ch 7)
- We will start with a simple hardware configuration
  - what OS is needed to support this
- Applications views the OS as the "computer"
  - the OS needs to provide a *consistent* and *usable interface*
    - while being *secure* and *efficient*
  - that's a pretty tall order!

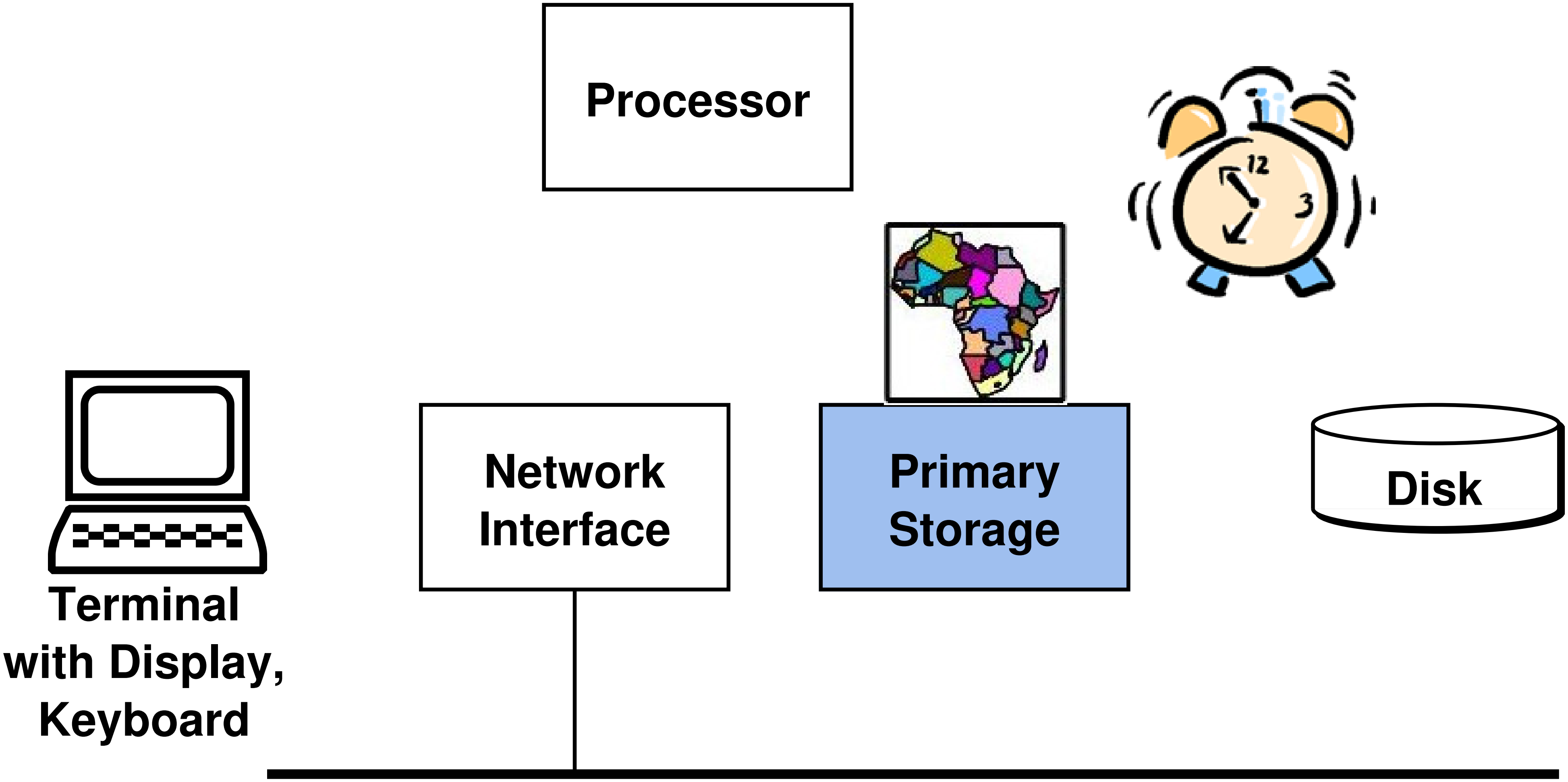
OS Design

- Our goal is to *build a general-purpose OS*
  - can run a variety of applications
    - some are interactive
    - many use network communication
    - all read/write to a file system
  - it's like most general-purpose OSes
    - Linux
    - Solaris
    - FreeBSD
    - Mac OS X
    - Chromium OS (has a Linux kernel)
    - Windows (the only one that's not directly based on Unix)
  - all these OSes are quite similar, functionally!
    they all provide:
    - processes
    - threads
    - file systems
    - network protocols with similar APIs
    - user interface with display, mouse, keyboard
    - access control based on file ownership and that file owers can control

OS Design Issues

- **Performance**
  - efficiency of application
- **Modularity**
  - tradeoffs between modularity and performance
- **Device independence**
  - for new devices, don't need to write a new OS
- **Security/Isolation**
  - isolate OS from application

Simple Configuration



- Early 1980s OS, so we can focus on the basic OS issues
  - no support for bit-mapped displays and mice
  - generally *less efficient* design

OS Components