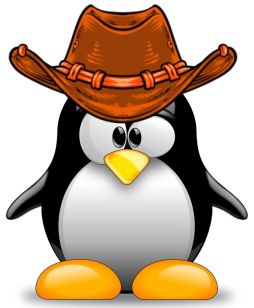
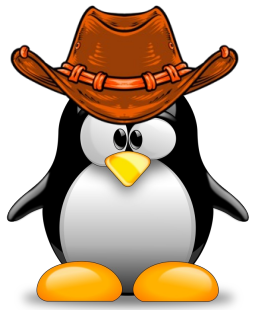


CSE 330:
Operating Systems

Adil Ahmad

Lecture #3: System boot process



Brief recap and continued from Lecture #2

Computer privilege modes or rings

Denotes the level of **configurable system state** by a software on a machine

User mode
(U-mode)

Only **unprivileged** instructions allowed at U-mode

Supervisor mode
(S-mode)

Both **unprivileged** and **privileged** instructions allowed at S-mode

What makes the OS so special that it can do its tasks?

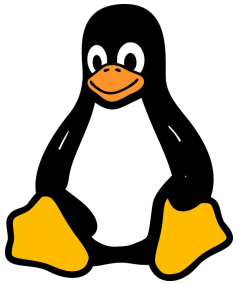
OS executes at the S-mode and controls system state

OS design (and life) is all about **trade-offs!**

- Monolithic has better performance and compatibility
- Micro-kernel has better modularity and security

Modern OSs adopt a monolithic approach, but with elements of micro-kernels

- The kernel is kept *as small as possible*
- But most servers (e.g., device drivers) are in the privileged kernel space



Windows 11

Fuchsia by Google

seL4 Security. Performance. Proof.

Each process has an (isolated) *address space*

- A process requires hardware resources to execute, including CPU and main memory (also called DRAM)
  - DRAM → Dynamic Random Access Memory
- OS does not show all system memory to a process, rather abstracts a portion of memory to each process
- View of memory shown to each process is called its “address space”
  - Typically, each process’ memory view is different from others

Processes can spawn other instances of itself

- Ask OS to create a completely different clone of the same program.
  - Clone will be a new “process” with isolated memory
  - E.g., running two instances of a web browser

- Ask OS to execute its context *simultaneously* on multiple CPUs
  - Each context is what we call a “thread”
  - Threads share the same address space but can execute different functions independently of each other
    - Useful for information sharing
  - E.g., two tabs within a web browser (*not always true* but for illustrative purpose!)

Recall that processes (and their threads) are *unprivileged*

Will the process ever require access to “privileged” functionality?

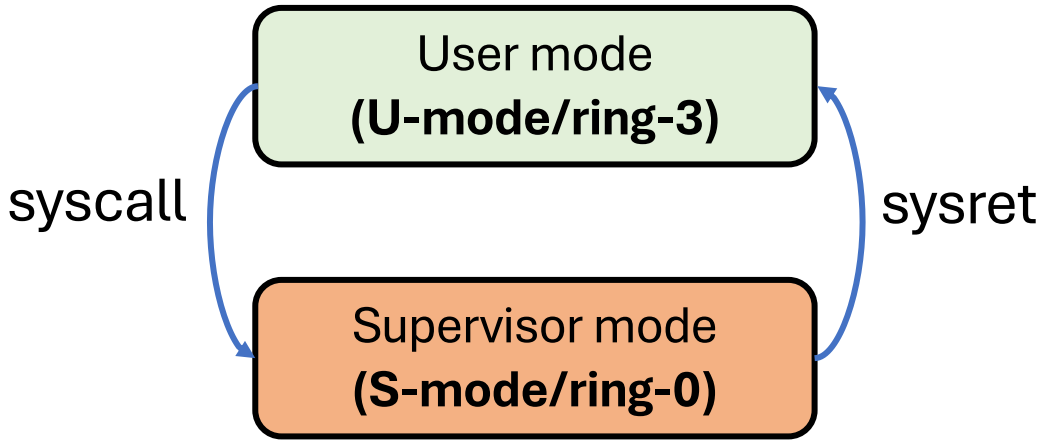
- **Yes!**
  - Reading a file from the SSD
  - Writing to a file in USB storage
  - Sending data through the network
  - Reading a hardware provided timer, etc.

- User processes need a mechanism to ask the operating system kernel to help with privileged tasks like reading to disk
  - OS kernels can check if the process is “authorized” or not

Process request privileged functions using **system calls**

- Think of them as “special functions” that send a request to the kernel, instead of another part of the current process

- Made using special instructions in modern CPUs to ensure security



System calls require:

(a) Switching privilege/address spaces

(b) Copying arguments/results

Some examples of famous system calls in Linux

fd = open(file, permissions, …)

- *Open a file for reading, writing, or both*

s = close(file)

- *Close an open file*

n = read(fd, buf, nbytes)

- *Read data from a file into a buffer*

n = write(fd, buf, nbytes)

- *Write data from a buffer into a file*

pos = lseek(fd, offset, whence)

- *Move the file pointer*

s = stat(name, &buf)

- *Get a file's status info (e.g., size, etc.)*

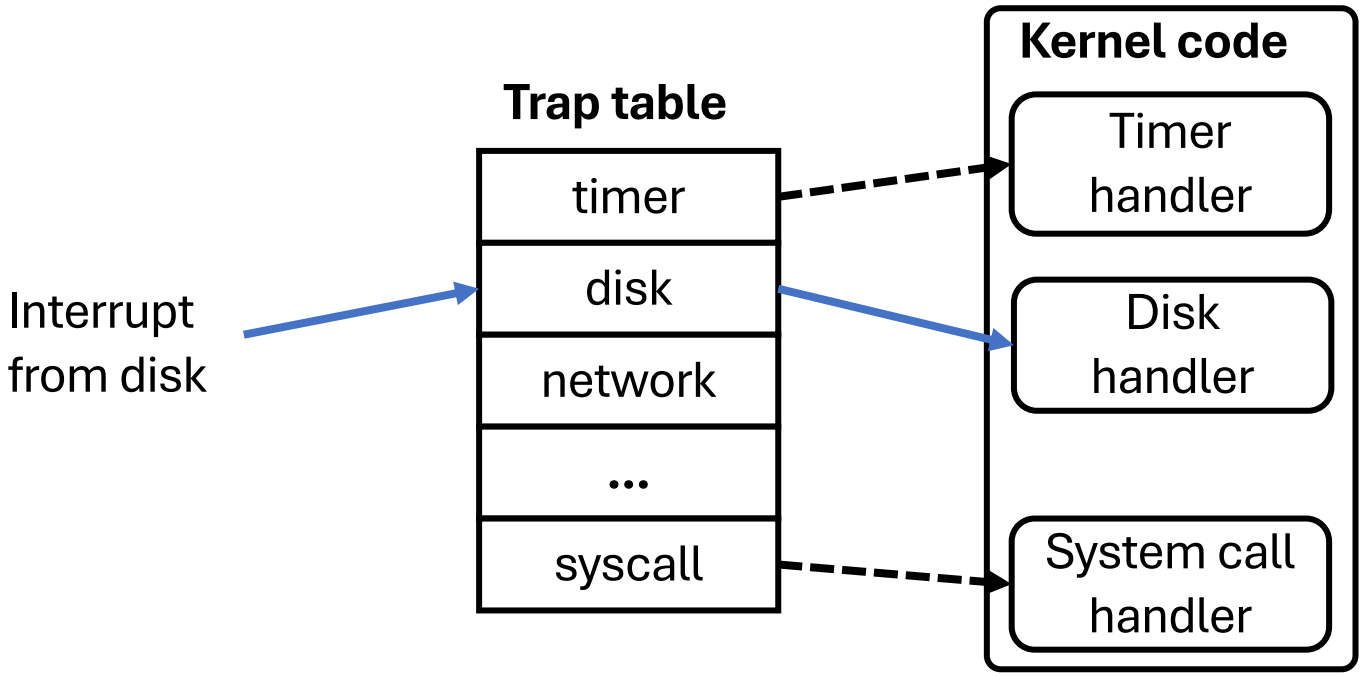
Processes also *trap* to OS for interrupts and exceptions

- **Interrupts** → CPU-enabled event notifiers, e.g.,
  - Bluetooth device sends an event/data packet
  - Sleep/wake between different CPU threads

- **Exceptions** → Something abnormal done by the process, e.g.,
  - Divide-by-0 and now needs to be stopped
  - Access memory region outside of the “address space”

Exception and system call handling

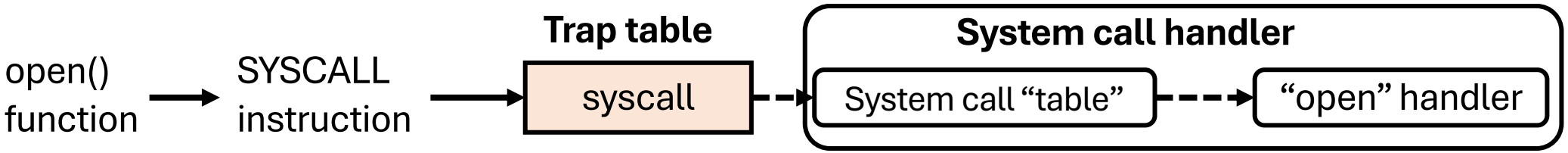
OS sets up a **trap table** inside its memory and tells the CPU its location



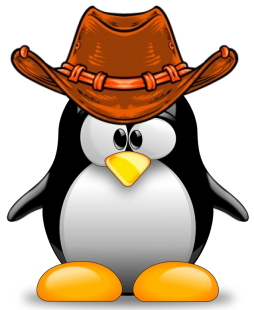
The CPU knows which entry to access based on interrupt/exception type

More detailed illustration of system call handling

- Process executes SYSCALL which tells the CPU a system call is made
  - CPU jumps to the system call handler in S-mode
  - Arguments are passed between process and kernel using CPU registers



- After system call is handled, OS executes SYSRET instruction
  - CPU jumps back to U-mode and resumes process
  - System call return value is also provided in a register



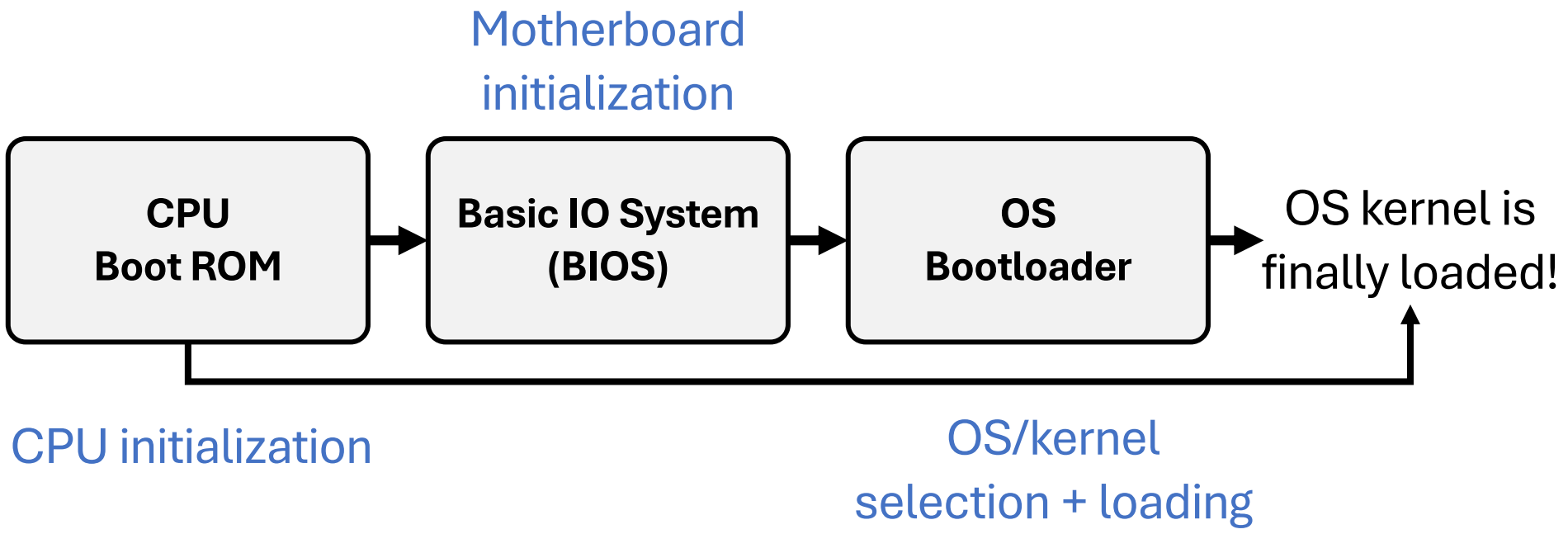
Today's topic: System boot

An overview of the **system boot process**

- There are many steps before an OS boots on your computer
- They are required to ensure that your OS operates correctly, e.g.,
  - Check that the CPU/motherboard is functional *(vendor-specific)*
  - Ensure the “correct” and “expected” OS kernel is booted
  - Provide the OS with initial system configuration (e.g., RAM, devices)

Let’s take a closer look at the system boot process

Three important components of a typical boot, before the OS kernel is loaded



CPU Boot ROM

First piece of code that executes when a CPU starts

How does the CPU know where to execute from?

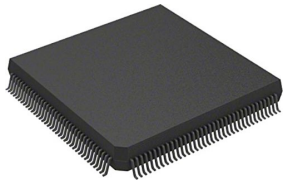
Hardcoded memory location and the Boot ROM is copied there

Important tasks:

Test and initialize critical CPU state (e.g., certain registers)

Typically, it is **implemented within**:

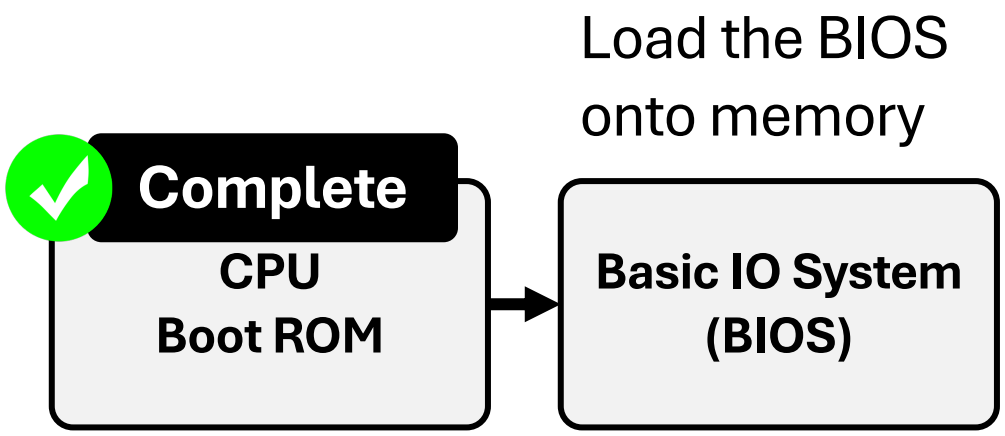
(a) the CPU die as Mask ROM (MROM)
(b) *read-only* flash storage part of the SoC



Mask ROM

Let’s move to the BIOS

Three important components of a typical boot, before the OS kernel is loaded



- CPU init.
- BIOS loading

The Basic Input Output System (aka BIOS)

Firmware code that lives on the motherboard and supplied by your *motherboard manufacturer*

Some tasks completed by BIOS

(i) Identify, test, initialize system (POST)

(ii) Configure system settings (BIOS menu)

(iii) System power management (ACPI)

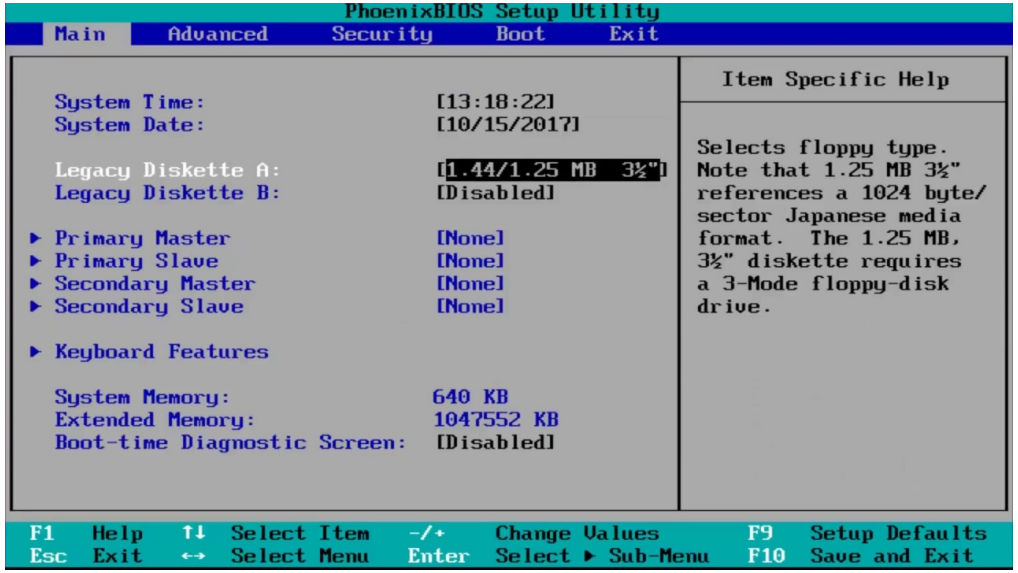
Important BIOS functions during boot

Power-On Self Test (POST)

- Test the computer hardware
  - Contains diagnostic routines for initializing CPU, DRAM, and peripherals
- Ever heard a loud beep when you start your PC? That's the POST!
- Simply used to diagnose hardware problems and prevent boot
  - Catastrophic if your hardware boots up

Motherboard Configuration

- Assign boot devices,
- set power limits, etc.



Does the BIOS remain functional after system boot?

Yes! Several motherboard-related functionality can only be provided by BIOS

Some important *post-boot* functionality provided by the BIOS:

(a) CPU power state change → idle, performance, etc.

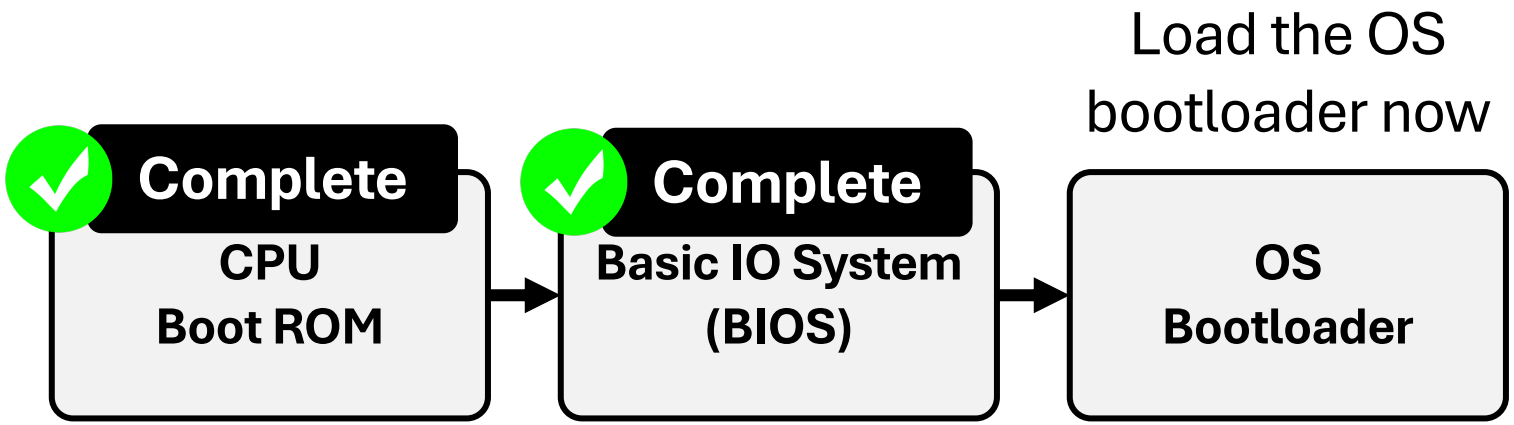
(b) Plug-and-play device enumeration (e.g., when you attach a USB device, it *informs* the kernel)

What x86 privilege level does it reside at?

x86 System Management Mode (SMM), even more privileged than ring-0 (OS)

Done with the BIOS, let's move to the bootloader

Three important components of a typical boot, before the OS kernel is loaded



- CPU init.
- BIOS loading

- POST
- MB init/config
- Secure boot

The OS Bootloader

User-supplied code on a bootable disk (e.g., GRUB)

Some functionality provided is as follows:

(i) Setup architectural features in some computers

(ii) Verify the OS kernel for **secure boot operations**

(iii) Load the OS kernel and set up its initial layout

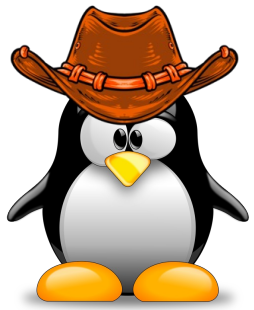
How does the BIOS know where to load bootloader from?

The first sector of the disk (e.g., USB) is the Master Boot Record (MBR), or in newer systems (called the GPT)

What does the MBR typically contain?

(a) Disc sector partition information

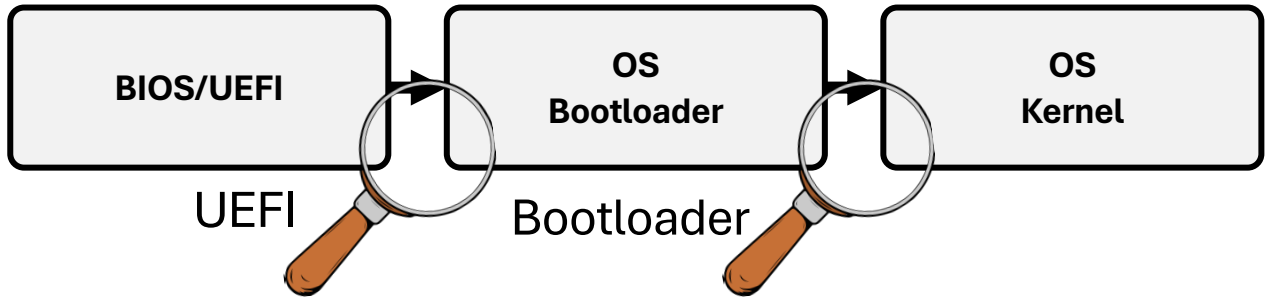
(b) Location of the bootloader



Examining the secure boot process

The motivation behind “Secure Boot”

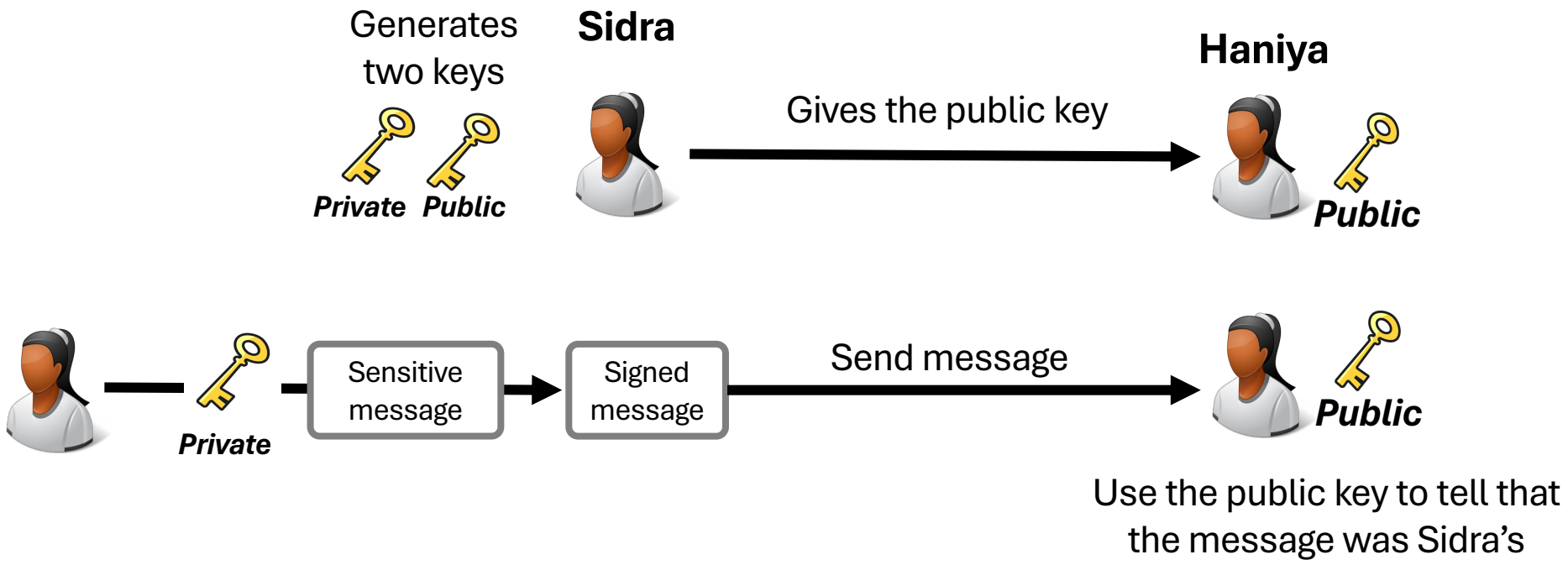
Important not to load an incorrect or **malicious** kernel image



Built around a “chain” of security checks

Concept 1: Asymmetric key cryptography

Establish identity of an individual over insecure channel (e.g., internet)



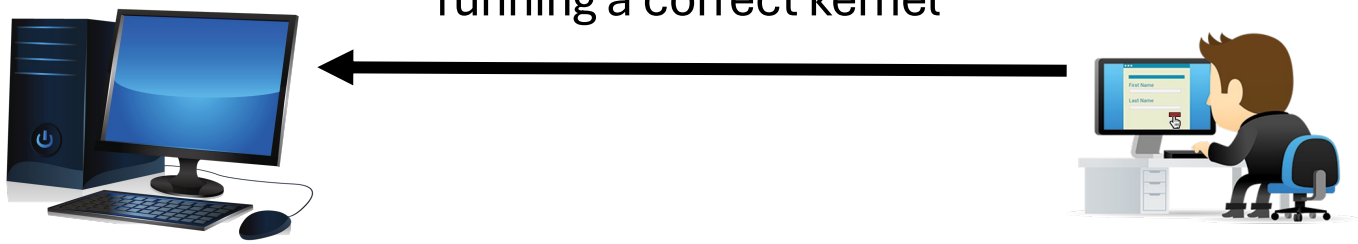
Overview of the secure boot process

- **Step #1:** Install a “public” key of a trusted party into the UEFI
- **Step #2:** Build a kernel bootloader, and sign it using the “private” key
- **Step #3:** During UEFI boot, check that the bootloader is correctly signed by the trusted key
  - **If not correctly signed, refuse to boot and terminate**
- **Same steps are repeated for the bootloader → kernel chain**

Secure boot can extend to **Remote Attestation**

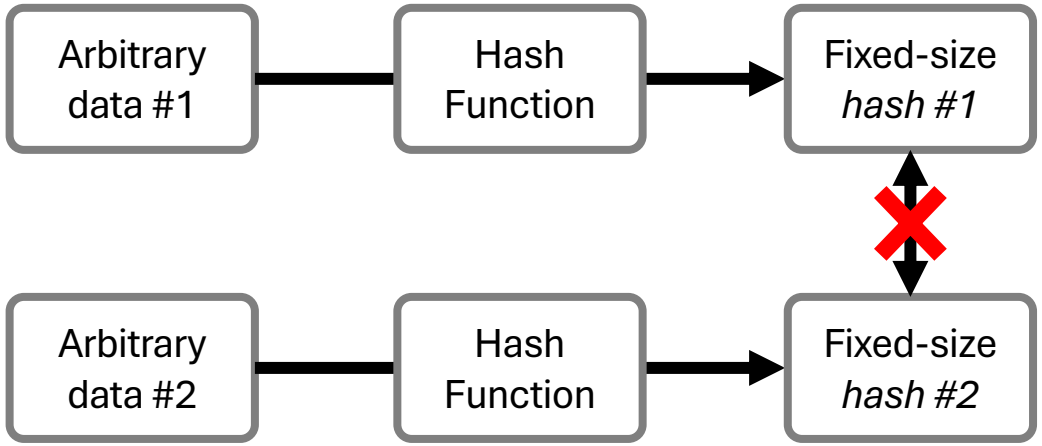
A system administrator monitoring remote computers

Prove to me that you're running a correct kernel



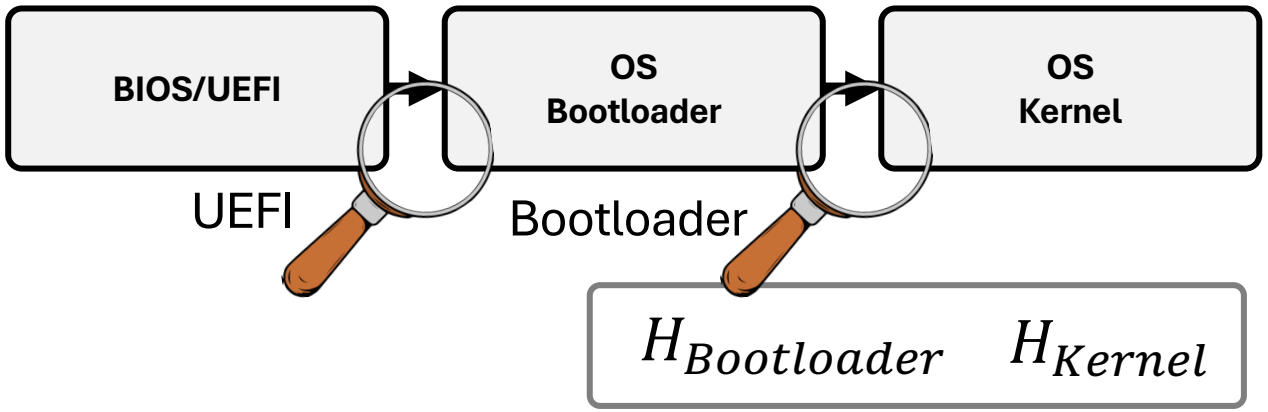
Concept 2: Cryptographic hashing functions

Mathematical algorithm that maps data of an arbitrary size to a bit array of a fixed size



Allows us to verify contents without transmitting all contents

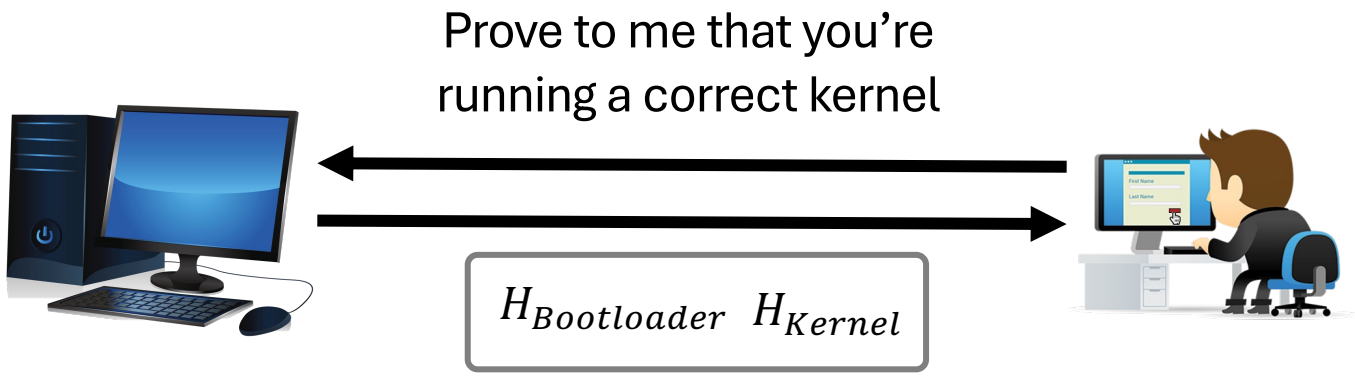
Measuring (hashing) components during the boot process



Not only check the signature, but also create a hash of the binary/software you are loading

Extending to **Remote Attestation**

A system administrator monitoring remote computers

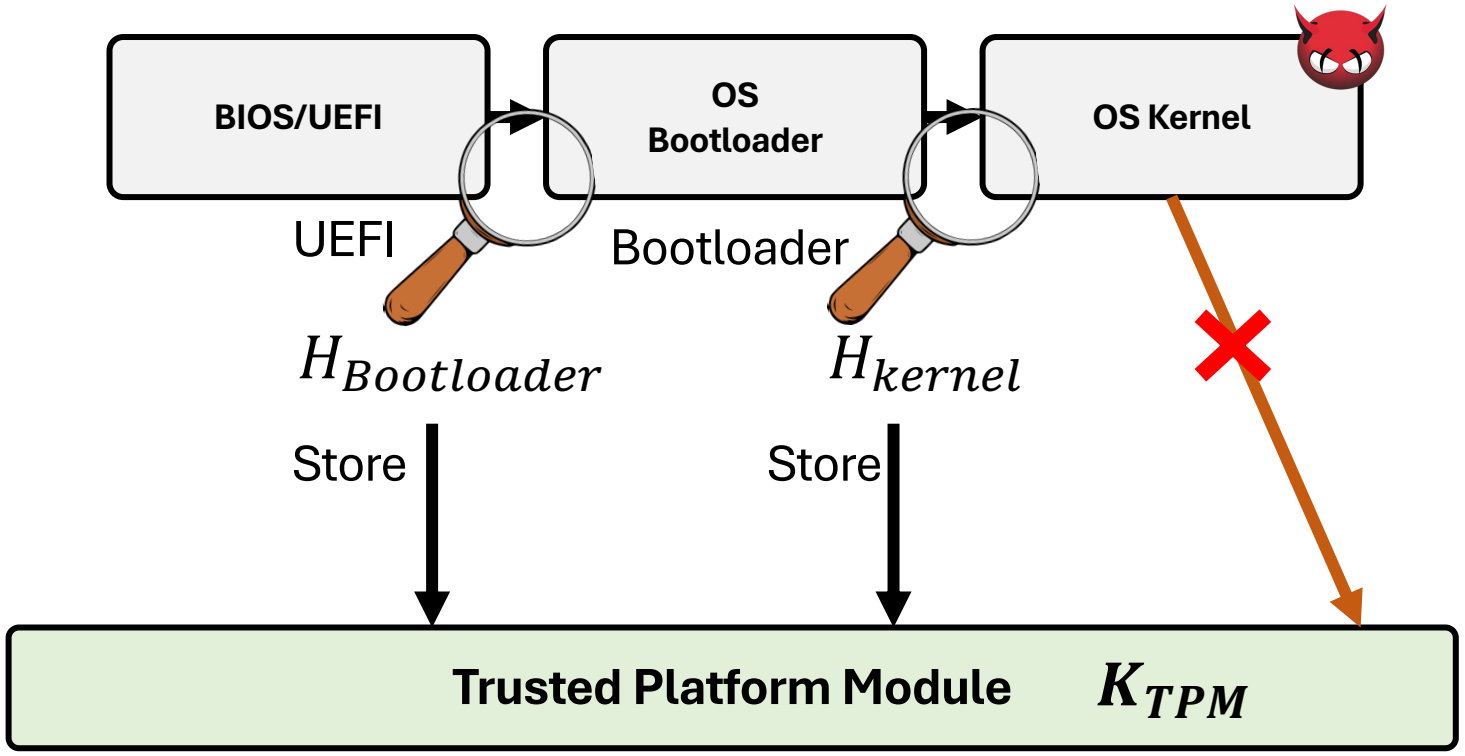


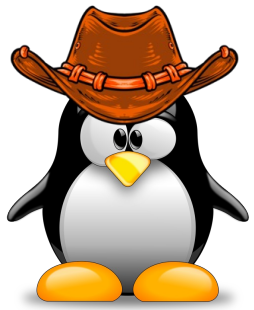
See any problem with this approach?

What if the kernel becomes compromised between the time when the hash was generated and "current" time

Trusted Platform Modules (TPMs)

Tamper-proof devices that securely keep measurements

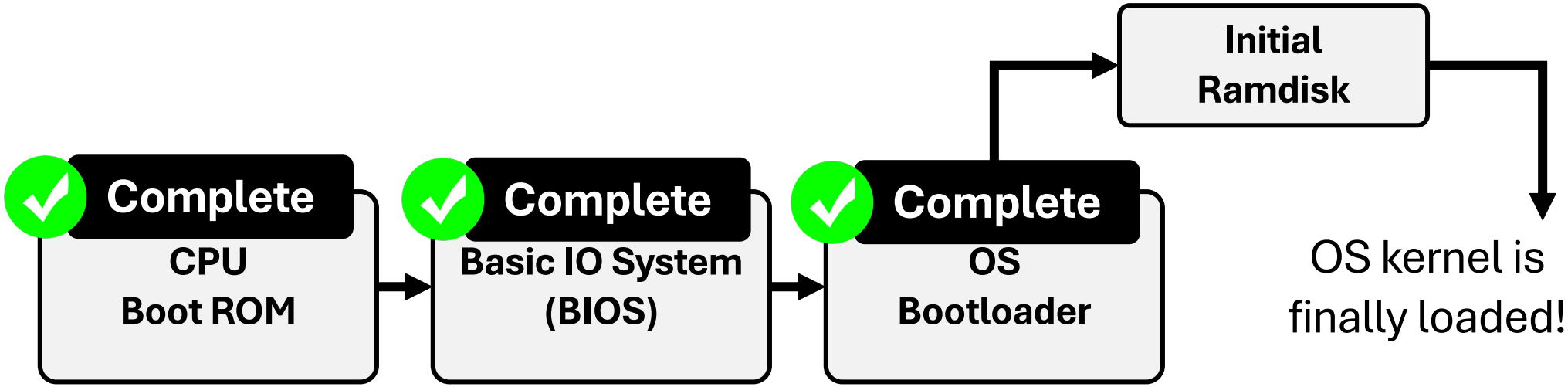




Let’s wrap up system boot

Wait, before we load the kernel, there's another stage

~~Three~~ Four **important stages** of a typical boot, before the OS kernel is loaded



- CPU init.
- BIOS loading

- POST
- MB init/config
- Secure boot

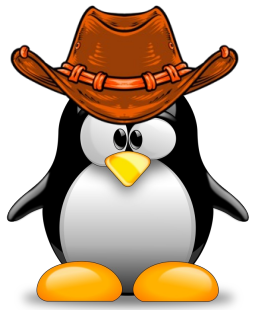
- Arch. setup
- Load/verify kernel image

The Initial Ramdisk

Before the actual filesystem is loaded, Linux loads a temporary filesystem into memory (for several functionalities) using the initial ramdisk

Why is the initial ramdisk needed?

- Some OS tasks are only needed at boot (e.g., device identification, root file system mounting, kernel memory randomization using KASLR, etc.)
- Bloating the actual kernel with these task handlers is inefficient
- Initial ramdisk handles these menial tasks for the kernel



Project #1 Overview and FAQs

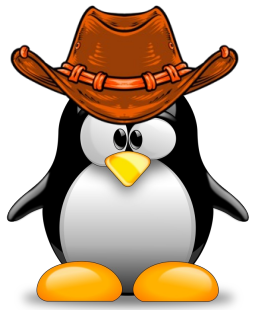
High-level overview of what is required

- Help you set-up a “Virtual Machine” running a Linux OS kernel
  - A VM is simply an emulated entire system inside your current host kernel (e.g., Windows/Mac)

- Steps to complete the project:
  1. Install the virtualization software (i.e., VirtualBox on x86 and UTM on MacOS) and set it up by following provided links (or use Google to find some resources!)
  2. Download Ubuntu (“desktop version”) from the provided link and use the steps explained in the attached links to install it into the VM
  3. Once Ubuntu starts, you need to install the custom kernel we asked by following steps provided in the document
  4. Create screenshots of all different components we mentioned

FAQs

- **How much memory should we allocate to the Virtual Machine?**
  - At least 4 GB, but up to 8GB if your system has 16GB of memory
- **What happens if I installed the "server" version?**
  - You can still install "desktop" from the command prompt using links I provided on Slack
- **How long does the kernel build process take?**
  - It can take up from 1 – 2 hours depending on your hardware; please be patient. If it seems to be going, let it go through.
- **Will we require this VM for next assignments?**
  - We will provide a VM image for you to use if you don't complete project #1.
- Please make sure to check slack regularly!



Questions? Otherwise, see you next time!