CS 453/698: Software and Systems Security

Module: Operating Systems Security

Lecture: Secure Boot, Virtualization, Sandboxing, Compartmentalization

Adam Caulfield

University of Waterloo

Spring 2025

Reminders & Recap

Reminders:

- A2 is due June 20

Recap – last time we covered:

- Malware
  - Types: virus, worm, trojan, logic bomb
  - Detection methods: signature vs. behavioral
- Reflections on Trusting Trust
  - We can't trust anything, but we have to trust something
  - System should have minimal & verifiable **Root of Trust**
- Adversary's steps for an attack
  - Security mechanisms target a step in the attack
  - Design depends on other assumptions and root of trust

Today

Trust in OSes:

- Secure boot
- Inter-process isolation

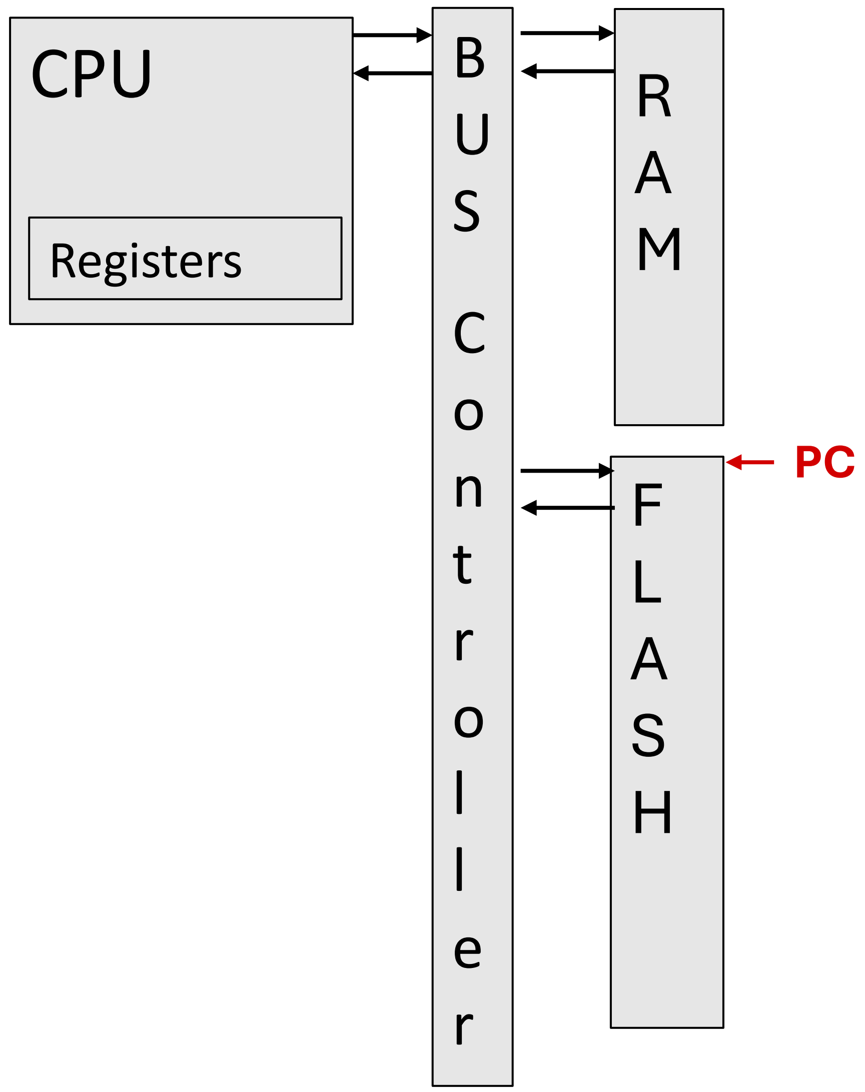
Virtualization

Compartmentalization

Sandboxing

- seccomp

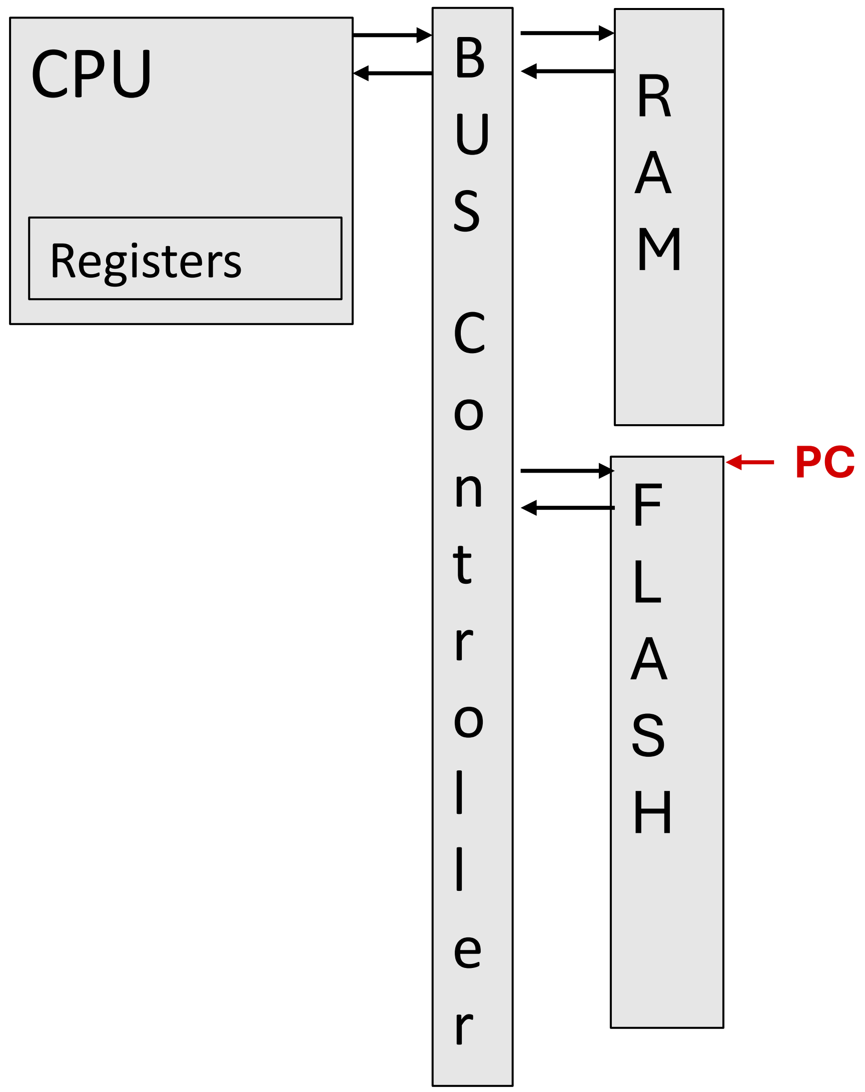
System model



Recall:

- Simple computer system model
  - CPU + Registers
  - Bus controller
  - RAM
  - FLASH
- “Boot” by assigning PC to top of FLASH

System model

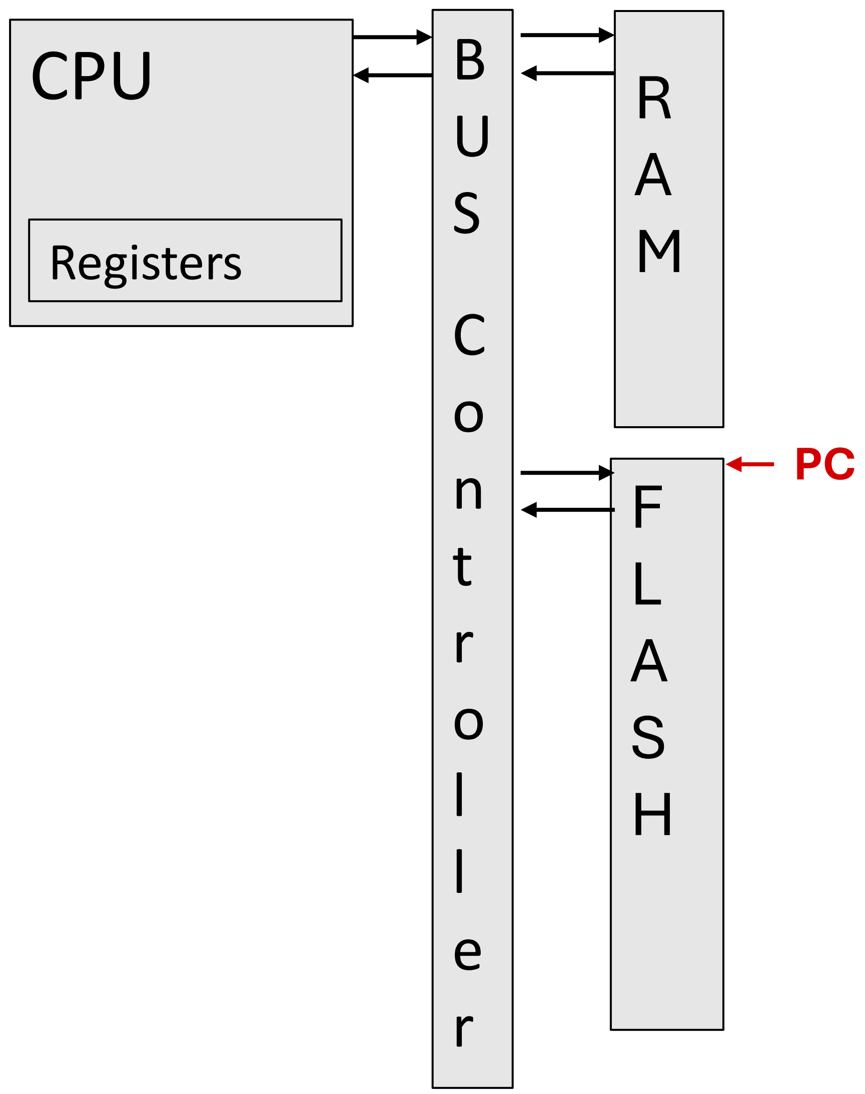


Recall:

- Simple computer system model
  - CPU + Registers
  - Bus controller
  - RAM
  - FLASH
- “Boot” by assigning PC to top of FLASH

Question: How do we ensure code will only execute if it hasn’t been modified?

System model



Recall:

- Simple computer system model
  - CPU + Registers
  - Bus controller
  - RAM
  - FLASH
- "Boot" by assigning PC to top of FLASH

Question: How do we ensure code will only execute if it hasn't been modified?

Answer: Secure boot

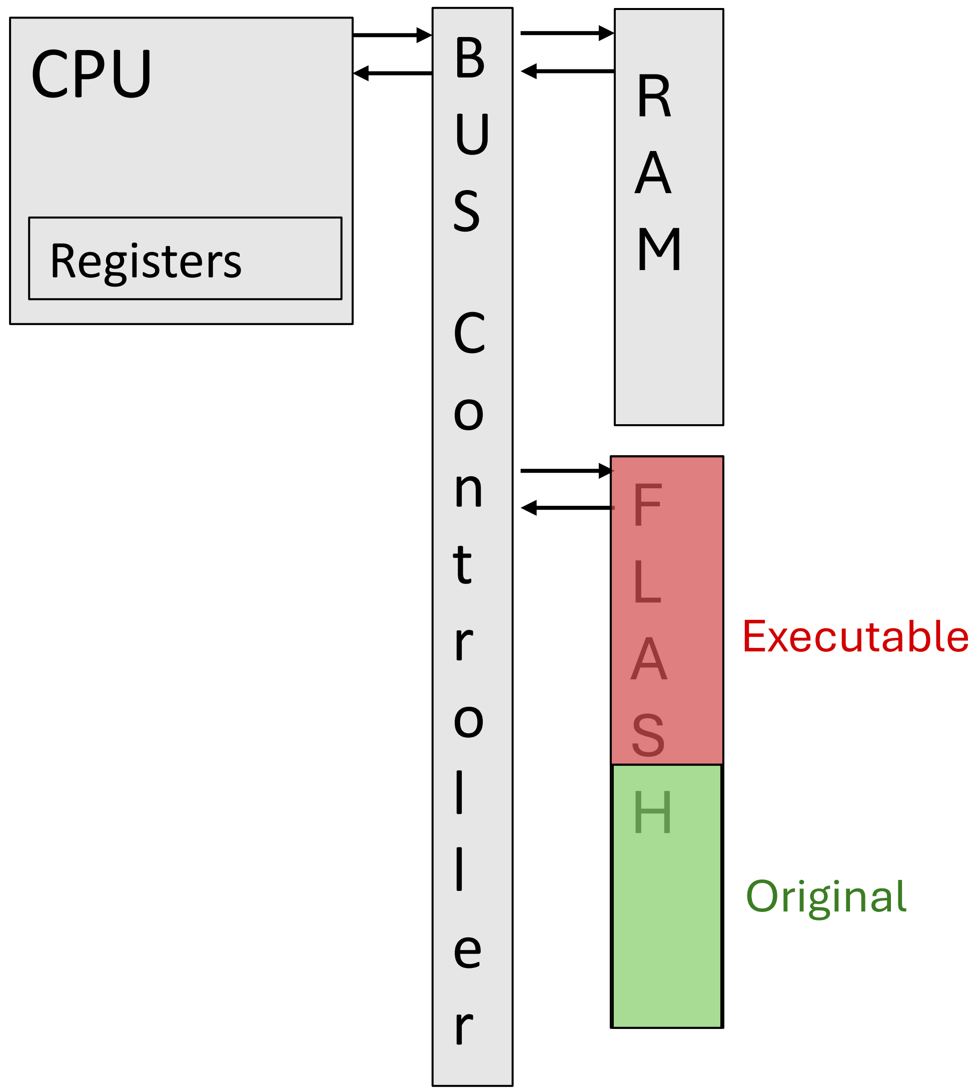
Secure Boot

Let's start from scratch…

Secure boot: only start executing if the correct software is installed

- Simplest approach:
  - Compare against the original -- simple
- Where to store the original?
  - RAM?
  - FLASH?

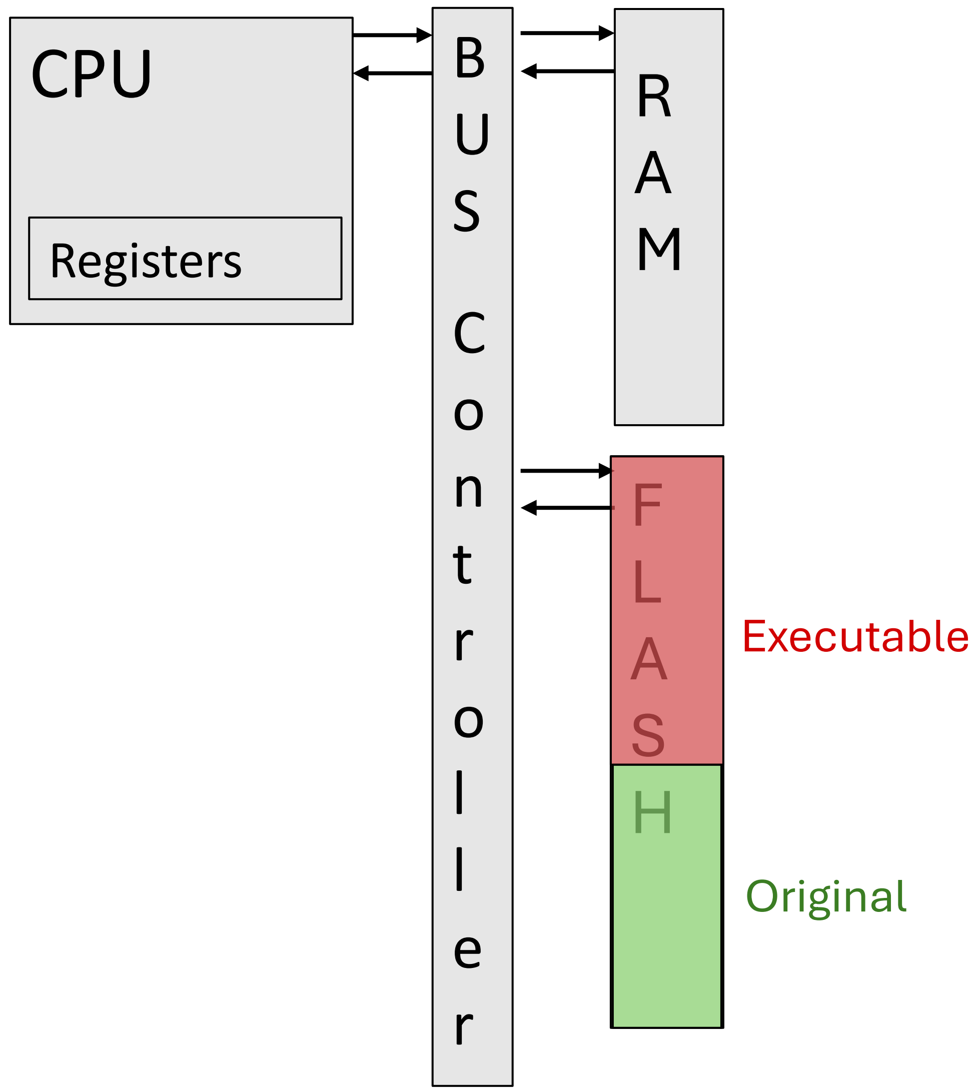
Secure Boot



Simple approach:

- Split FLASH into two parts
  - The "executable program"
  - The original for verification
- Compare them at boot
- We did it! ☺
  - Efficient?
  - Secure?

Secure Boot



Simple approach:

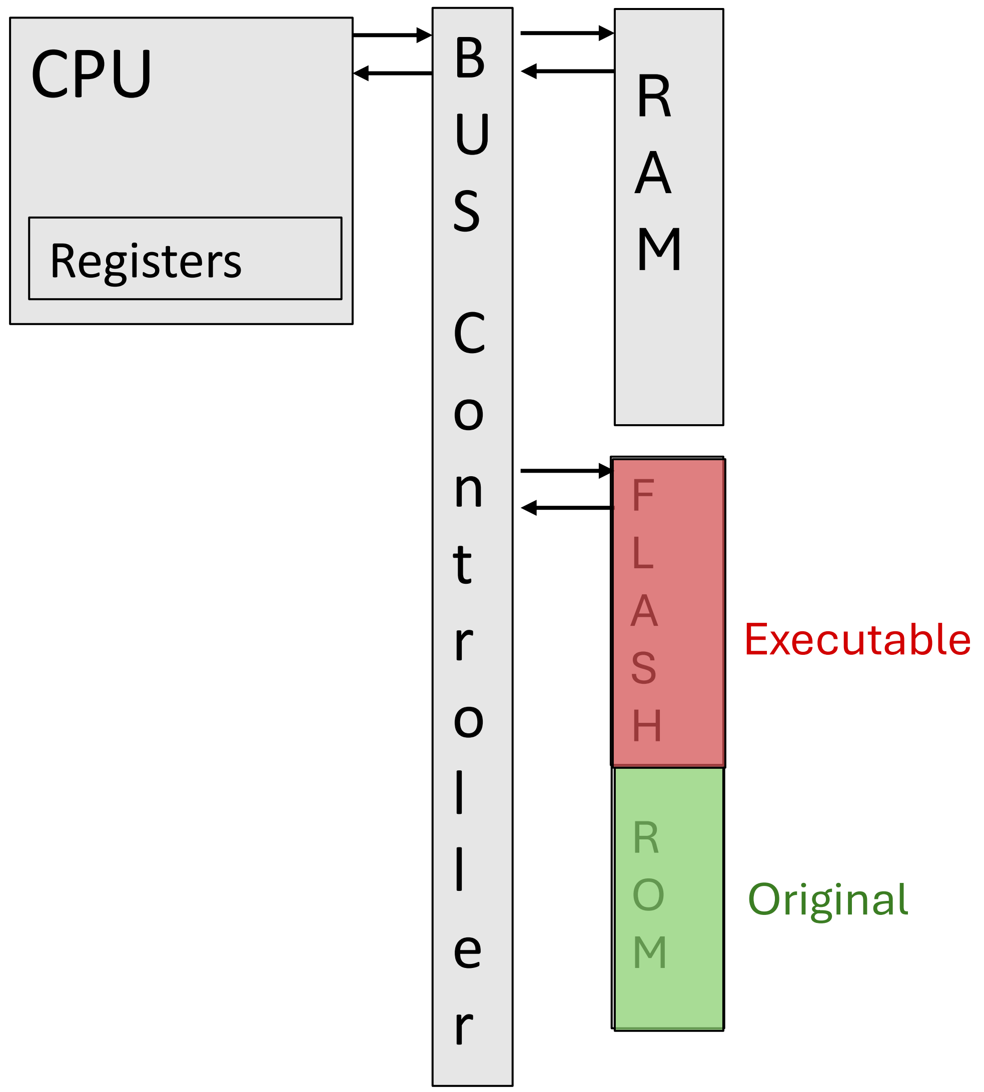
- Split FLASH into two parts
  - The “executable program”
  - The original for verification
- Compare them at boot
- Just kidding ☹
  - Trivially insecure
  - FLASH is writeable

Secure Boot

- Read-only memory (ROM)
  - A form of non-volatile memory
  - Data stored in ROM cannot be electronically modified after the manufacture of the device
  - Useful for storing information that should never change

Let’s incorporate ROM into the simple secure boot…

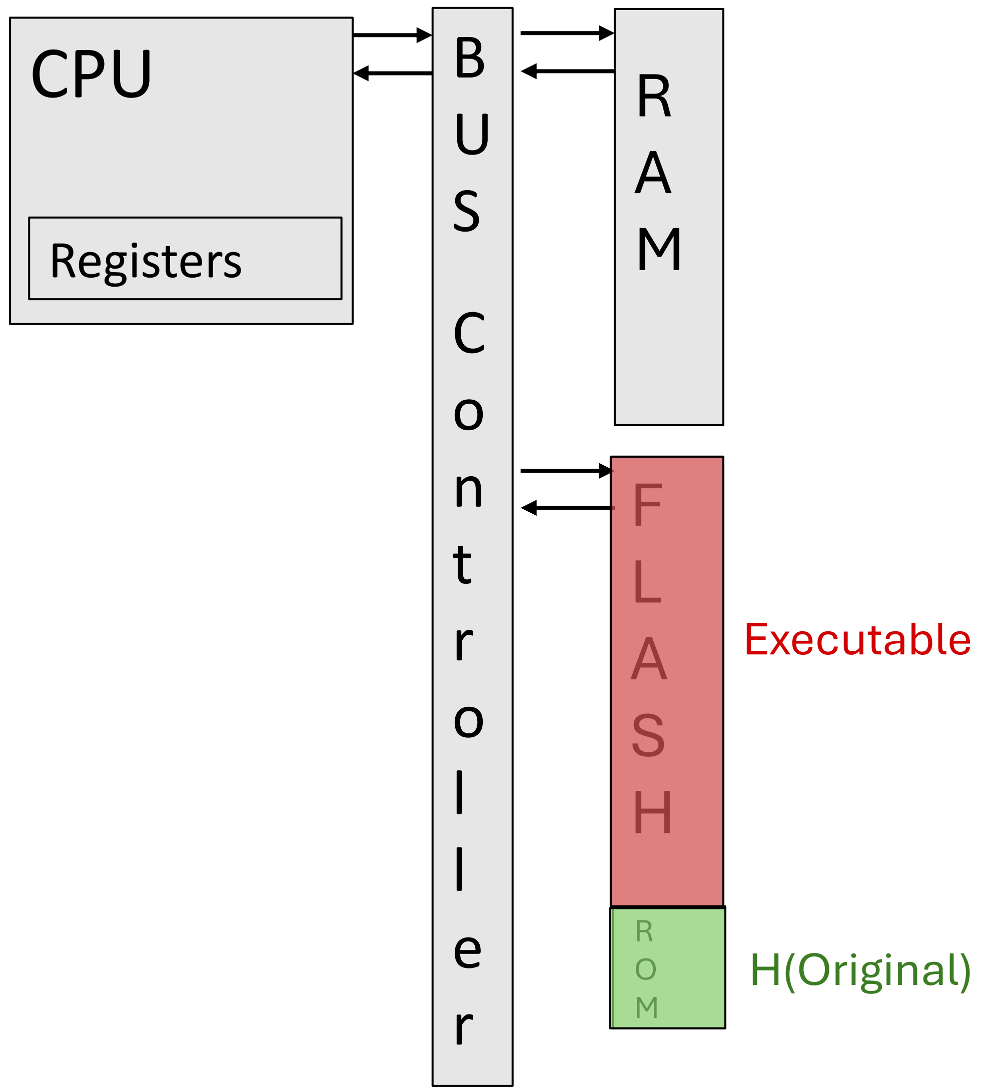
Secure Boot



New approach:

- Store the original in ROM
- Compare the executable to the (now immutable) original
- We did it! ☺
  - Secure?
  - Efficient?
  - Convenient?

Secure Boot



A better idea:

- Store a hash of the original in ROM instead
  - Using a cryptographic hash function
- **At boot:**
  - Compute a hash of the executable
  - Compare to the hash stored in ROM
- Does it work?

Secure Boot

Advantages of storing the hash?

Secure Boot

Advantages of storing the hash?

Compression

- Executable = arbitrary size in flash
- H(Original) = fixed size in ROM
- Reduced and fixed ROM space

Collision Resistant:

- Due to use of cryptographically secure hash function
- Every software will always hash into a different byte string

Secure Boot

Problems with our current version?

Secure Boot

Problems with our current version?

ROM is installed by the manufacturer

- Therefore, the manufacturer must determine which software can run at manufacturing time...
- What if the software has a bug?

Who computes the hash of the executable before comparing?

- Must happen before running any software... dedicated hardware?

User has no control over the executable → not programmable!

Secure Boot

Problems with our current version?

ROM is installed by the manufacturer

- Therefore, the manufacturer must determine which software can run at manufacturing time…
- What if the software has a bug?

Who computes the hash of the executable before comparing?

- Must happen before running any software… dedicated hardware?

User has no control over the executable → not programmable!

Secure, but not practical or useful

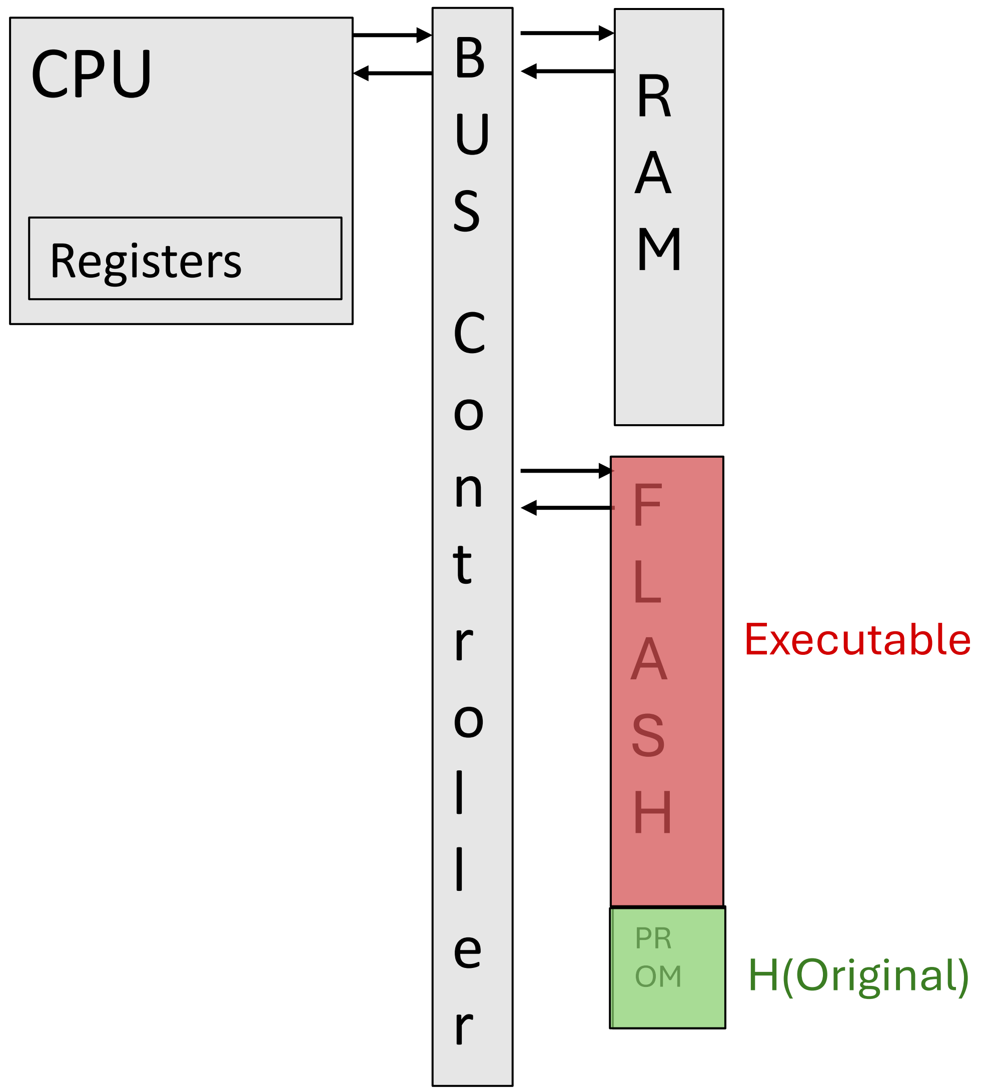
Secure Boot

We need a new type of memory…

Programmable ROM

- Aka “write-once read-many”
- Can be written to once, cannot be modified thereafter
- Assures that data cannot be tampered with after the first write
- Build using fuse hardware circuits

Secure Boot



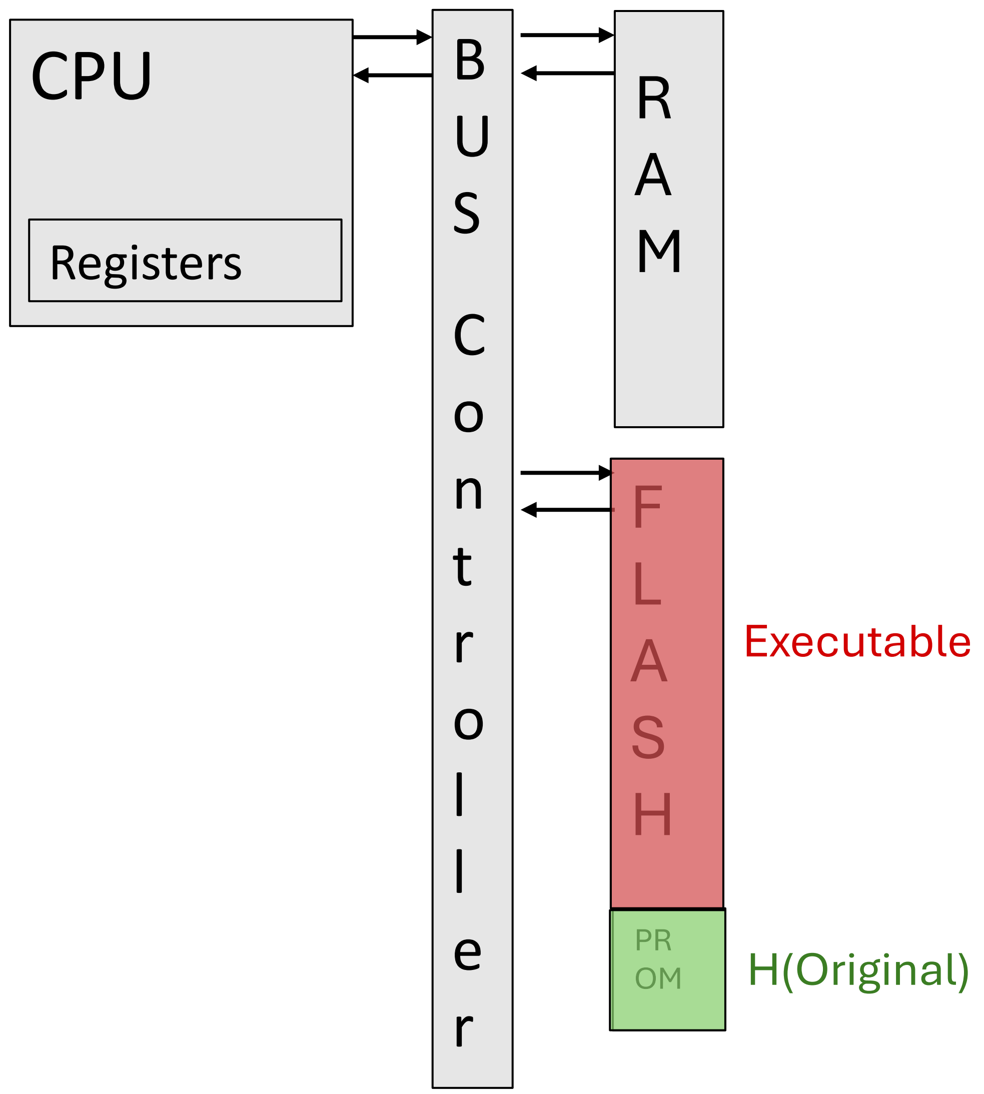
How about this?

- Device is manufactured with PROM
- User buys device
- User decides which code to burn in
- User computes H(code) and writes to PPROM

Now:

- User/owner regains control

Secure Boot



How about this?

- Device is manufactured with PROM
- User buys device
- User decides which code to burn in
- User computes H(code) and writes to PPROM

Now:

- User/owner regains control

But:

- Can program device once, and who checks?

Secure Boot

How can we ensure:

- Only authorized code boots?
- While also allowing it to change?

Secure Boot

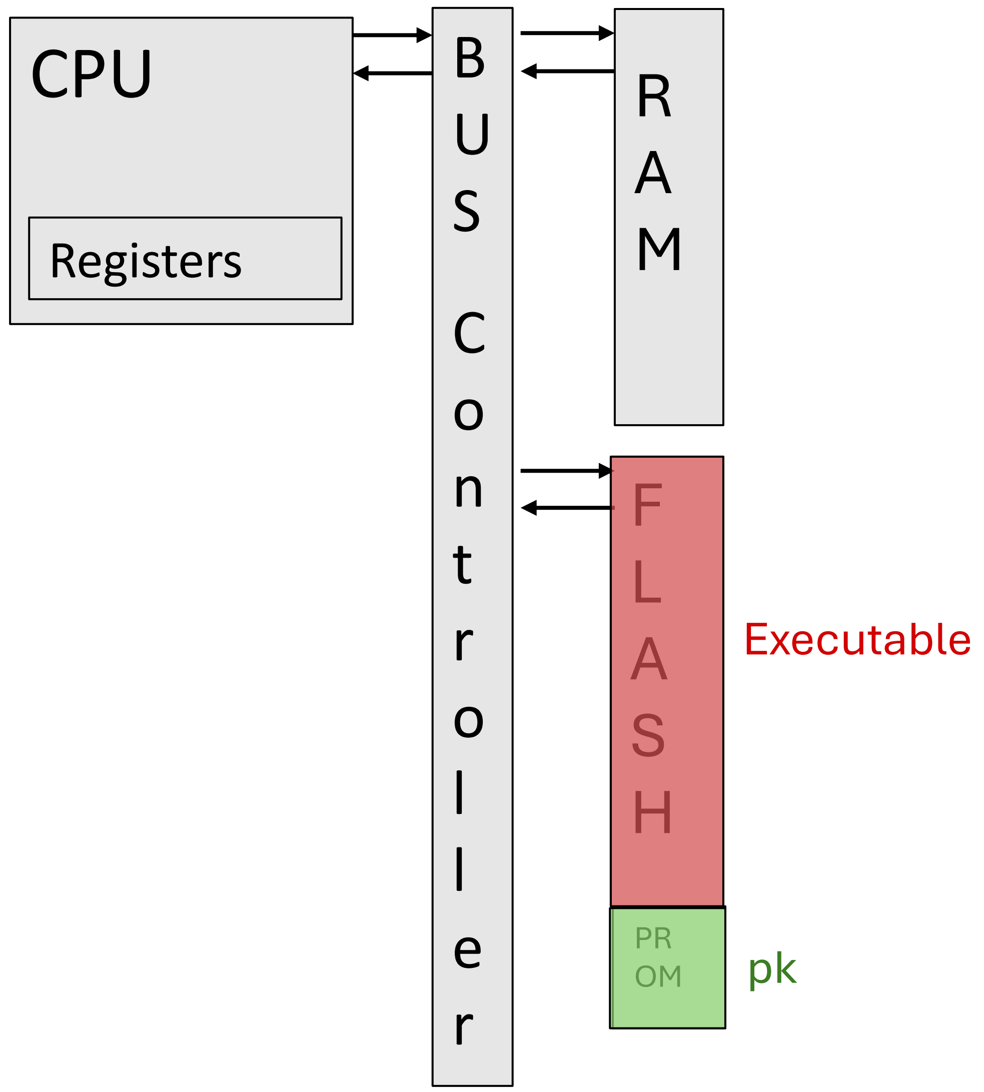
How can we ensure:

- Only authorized code boots?
- While also allowing it to change?

Public key cryptography

- Use a signature to authorize the executables
- Burn the public key into secure memory

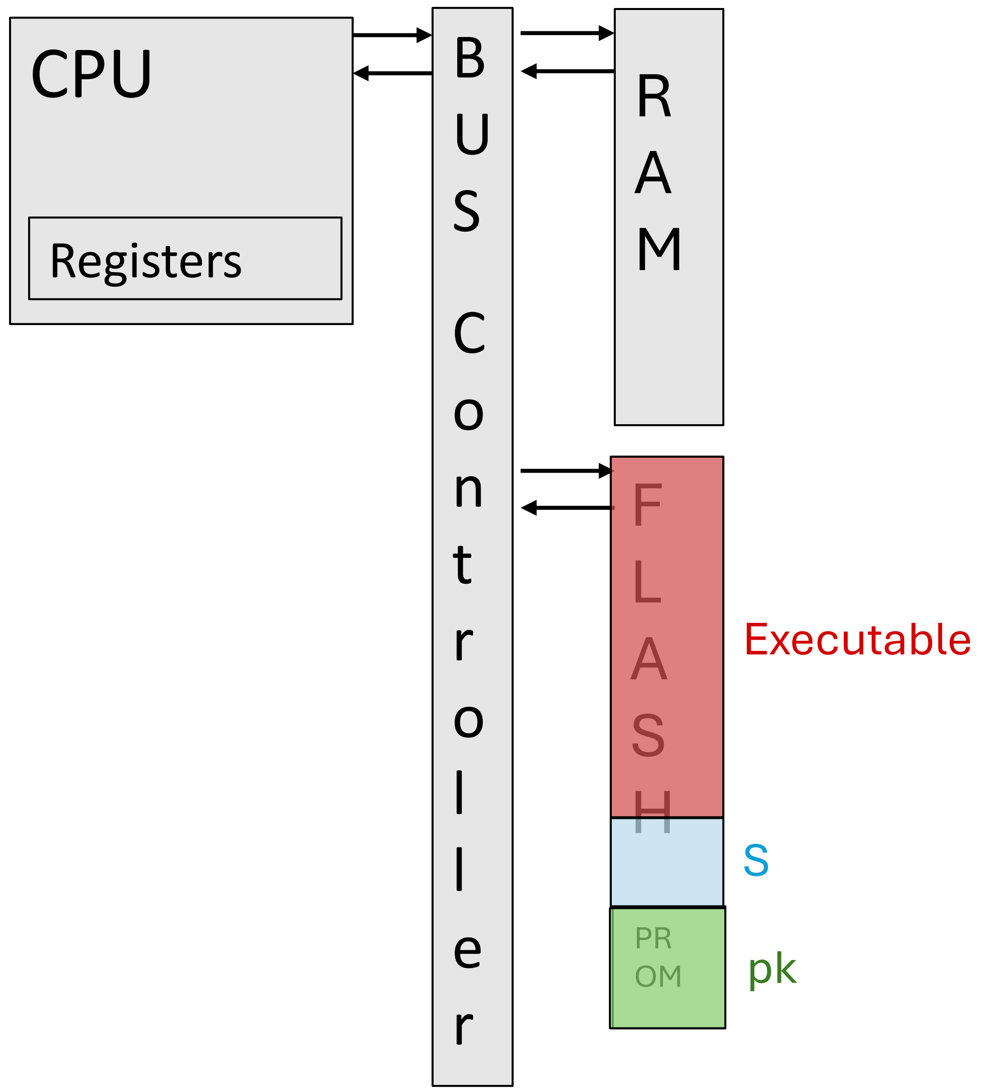
Secure Boot



Using public-key crypto:

- User picks a secret/public key pair
- Burns the public key (pk) into PROM

Secure Boot



Using public-key crypto:

- User picks a secret/public key pair
- Burns the public key (pk) into PROM

- User signs the executable
  - S = sign(sk, executable)
  - Places a S into ***reserved*** *and* ***fixed*** region of FLASH memory

- **<u>At boot:</u>**
  - Check if contents of this region contain a valid signature

Secure Boot

Last remaining question: where is the check implemented?

Secure Boot

Last remaining question: where is the check implemented?

Option 1: do everything in hardware

- Unnecessarily expensive

Secure Boot

Last remaining question: where is the check implemented?

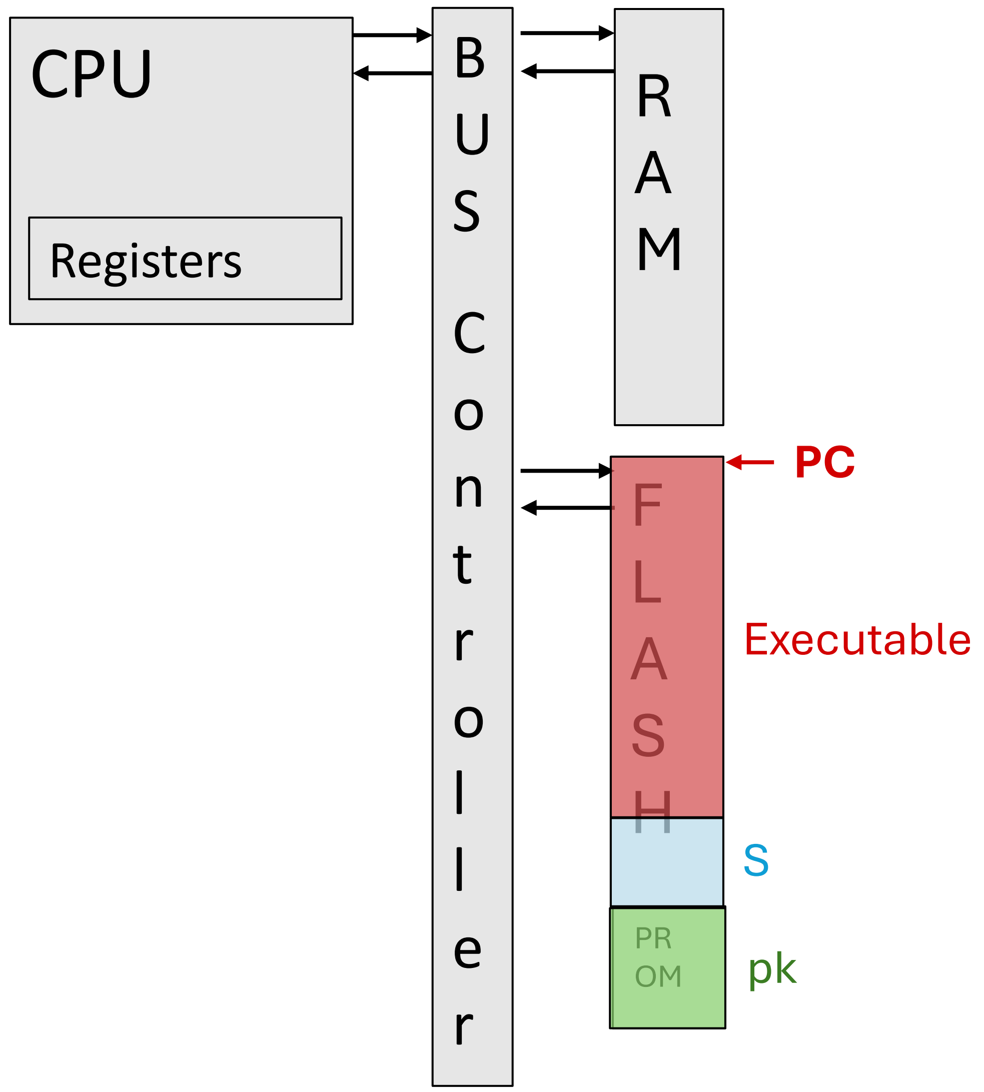
Option 1: do everything in hardware

- Unnecessarily expensive

Option 2: implement the checks in software

- <u>**Chicken-egg problem:**</u>
  - Now we need to check that the “checking software” hasn’t been modified?
- How do we ensure it runs first?

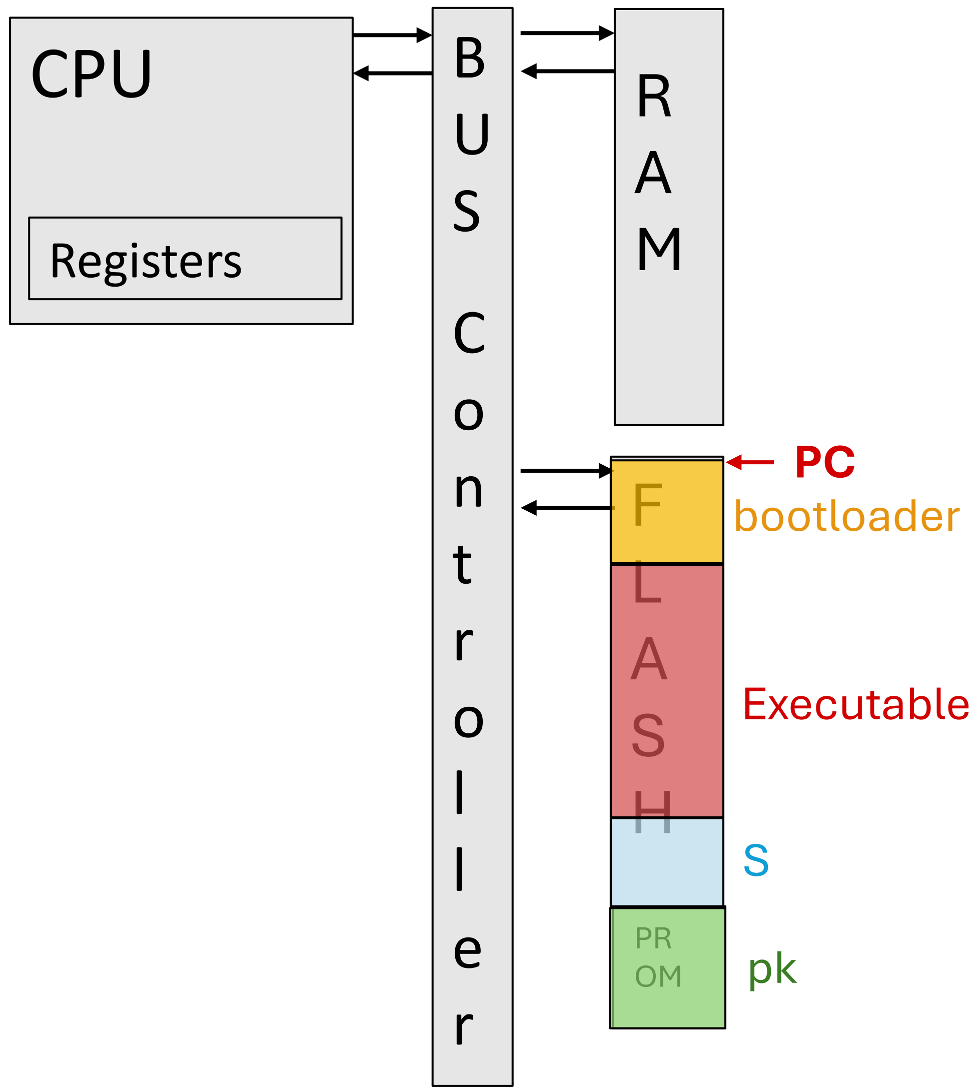
Secure Boot



<u>When the CPU boots:</u>

- Moves “PC” to the top of flash

Secure Boot

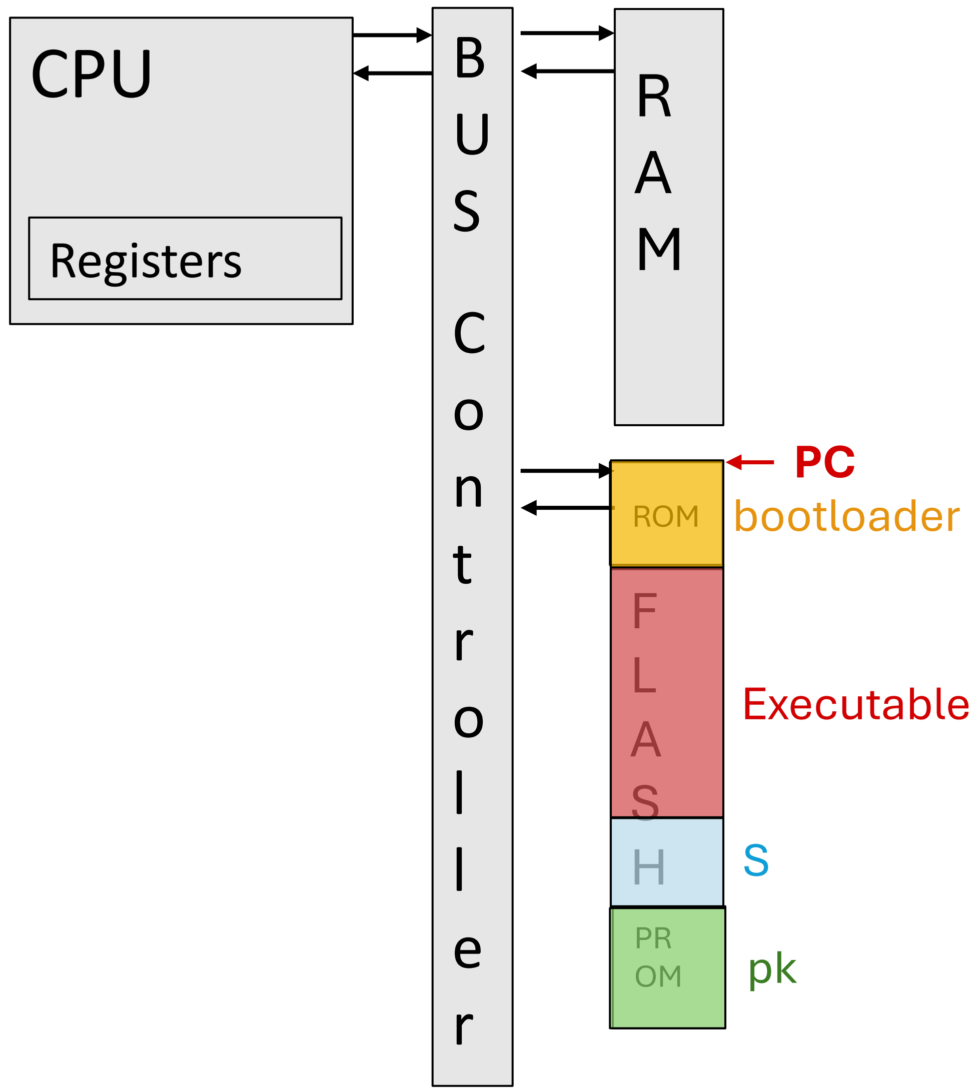


When the CPU boots:

- Moves “PC” to the top of flash
- So we can put our “checker” there
  - The bootloader

How to we make sure it is immutable?

Secure Boot



<u>When the CPU boots:</u>

- Moves "PC" to the top of flash
- So we can put our "checker" there
    - The bootloader

How to we make sure it is immutable?

- Use ROM

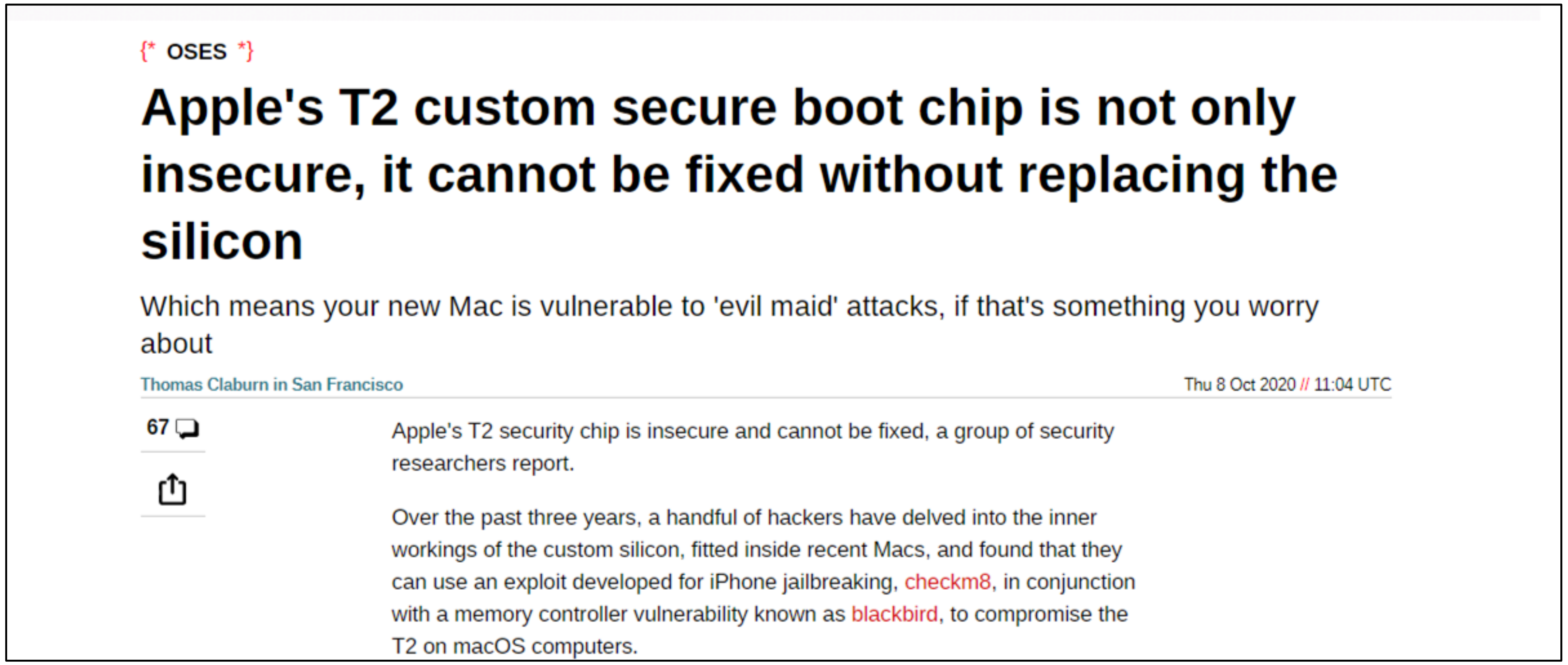
Done!

Secure Boot

What is in our Trusted Computing Base (TCB)?

- CPU hardware & BUS Controller
- ROM and PROM
- Bootloader code
  - If bootloader code has a vulnerability, it cannot be patched
    - Why
  - Must use secure signature/verification algorithm
  - Must be memory safe (e.g., no buffer overflows, etc)
  - Buggy bootloaders have happened…

Secure Boot


{* OSES *}

Apple's T2 custom secure boot chip is not only insecure, it cannot be fixed without replacing the silicon

Which means your new Mac is vulnerable to 'evil maid' attacks, if that's something you worry about

Thomas Claburn in San Francisco Thu 8 Oct 2020 // 11:04 UTC

67

Apple's T2 security chip is insecure and cannot be fixed, a group of security researchers report.

Over the past three years, a handful of hackers have delved into the inner workings of the custom silicon, fitted inside recent Macs, and found that they can use an exploit developed for iPhone jailbreaking, checkm8, in conjunction with a memory controller vulnerability known as blackbird, to compromise the T2 on macOS computers.

Read more about an example case

Secure Boot

What is in our Trusted Computing Base (TCB)?

- CPU hardware & BUS Controller
- ROM and PROM
- Bootloader code
    - If bootloader code has a vulnerability, it cannot be patched
        - Why
    - Must use secure signature/verification algorithm
    - Must be memory safe (e.g., no buffer overflows, etc)
    - Buggy bootloaders have happened…

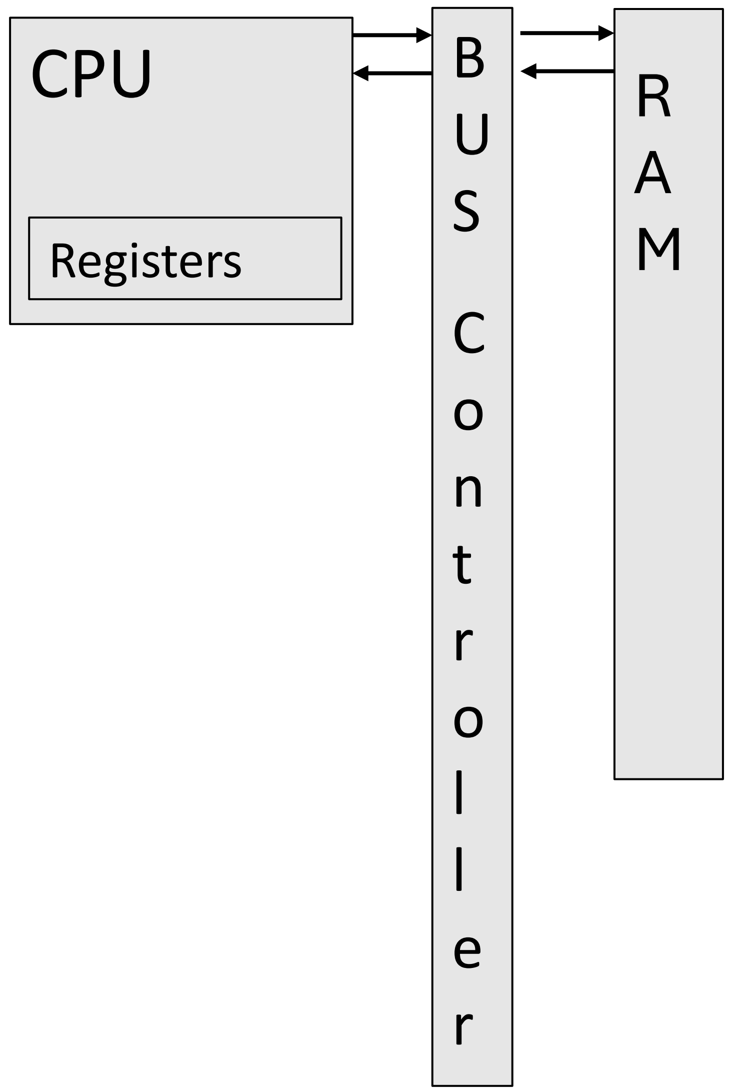
What is secure boot's Root of Trust?

What is secure boot's software Root of Trust?

Secure Boot

This was all for simple system… what about complex system?

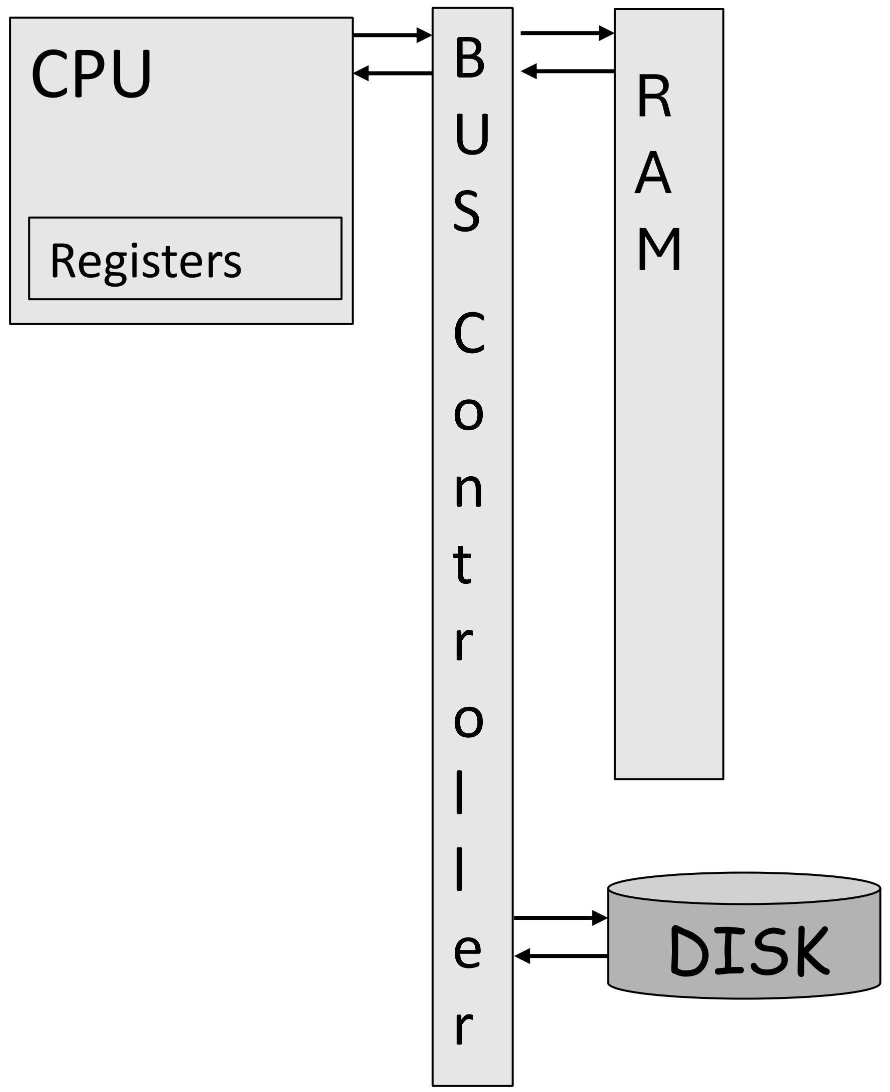
Updated system model



Main differences:

- Main memory
  - Instructions are also executed from RAM
  - RAM contains both data and programs
  - Still volatile

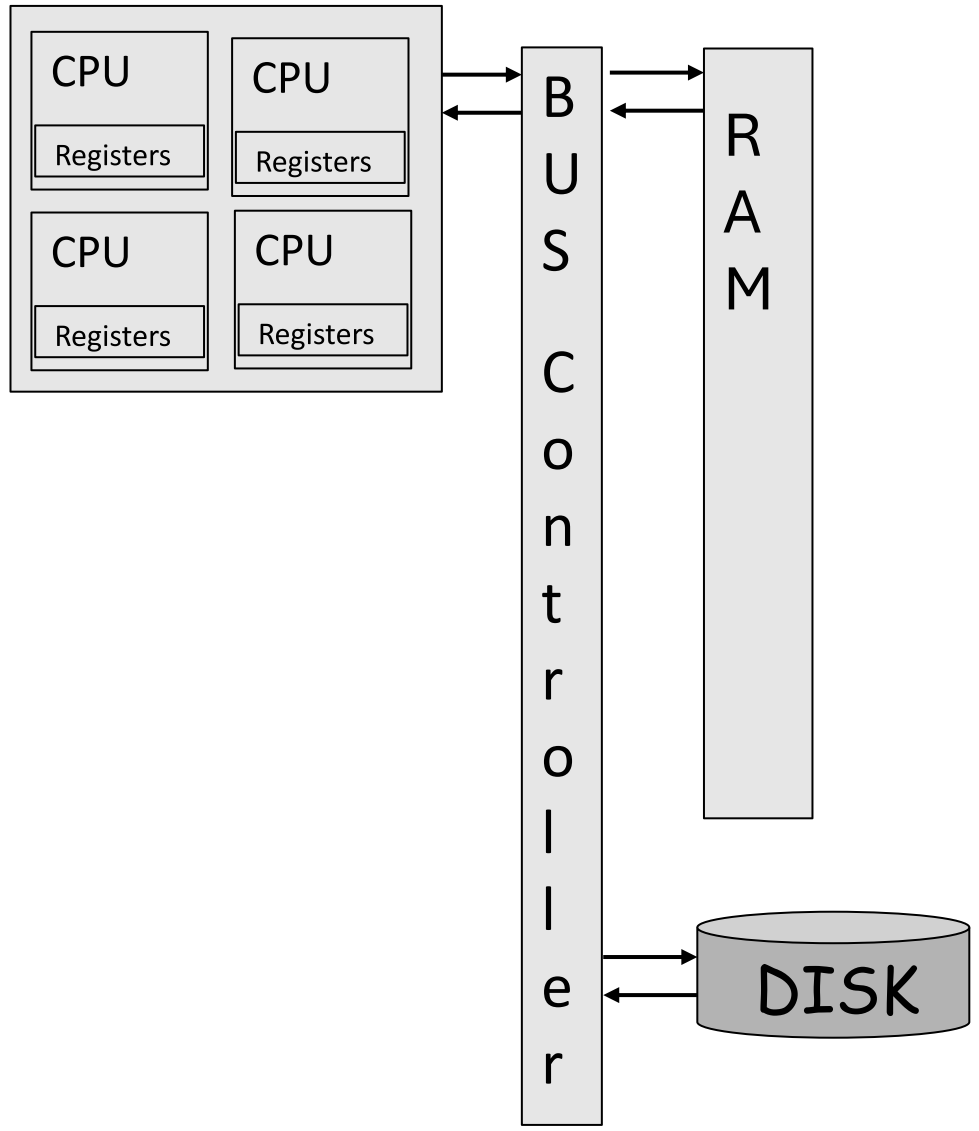
Updated system model



Main differences:

- Main memory
  - Instructions are also executed from RAM
  - RAM contains both data and programs
  - Still volatile
- Disk:
  - Persistent information
  - Lager and slower than RAM
  - Not executable
  - Programs are stored in disk and loaded into RAM before execution

Updated system model

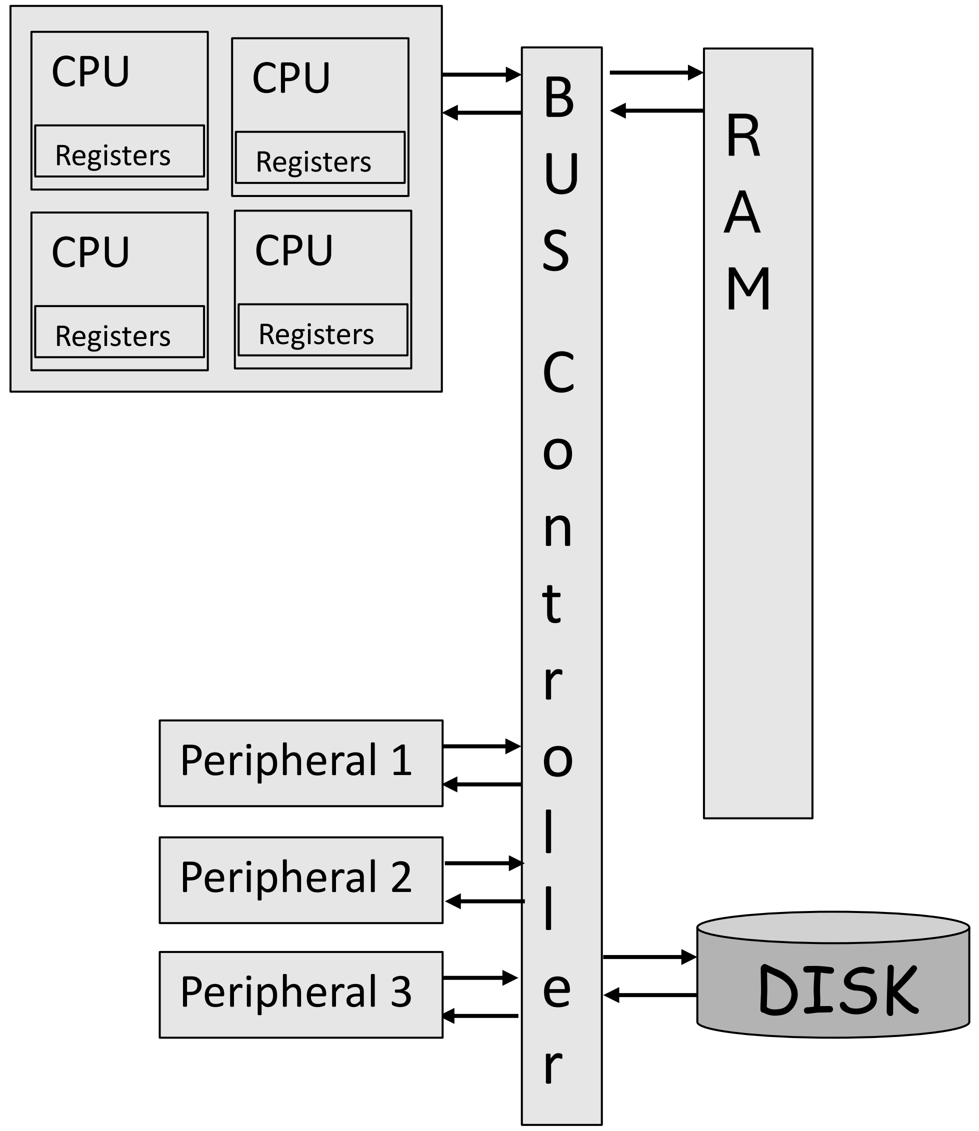


Main differences:

- CPU
  - Multi-core
  - Local shared cache, speculative, pipelined
  - At boot:
    - Only one core is active

- BUS Controller
  - Motherboard

Updated system model

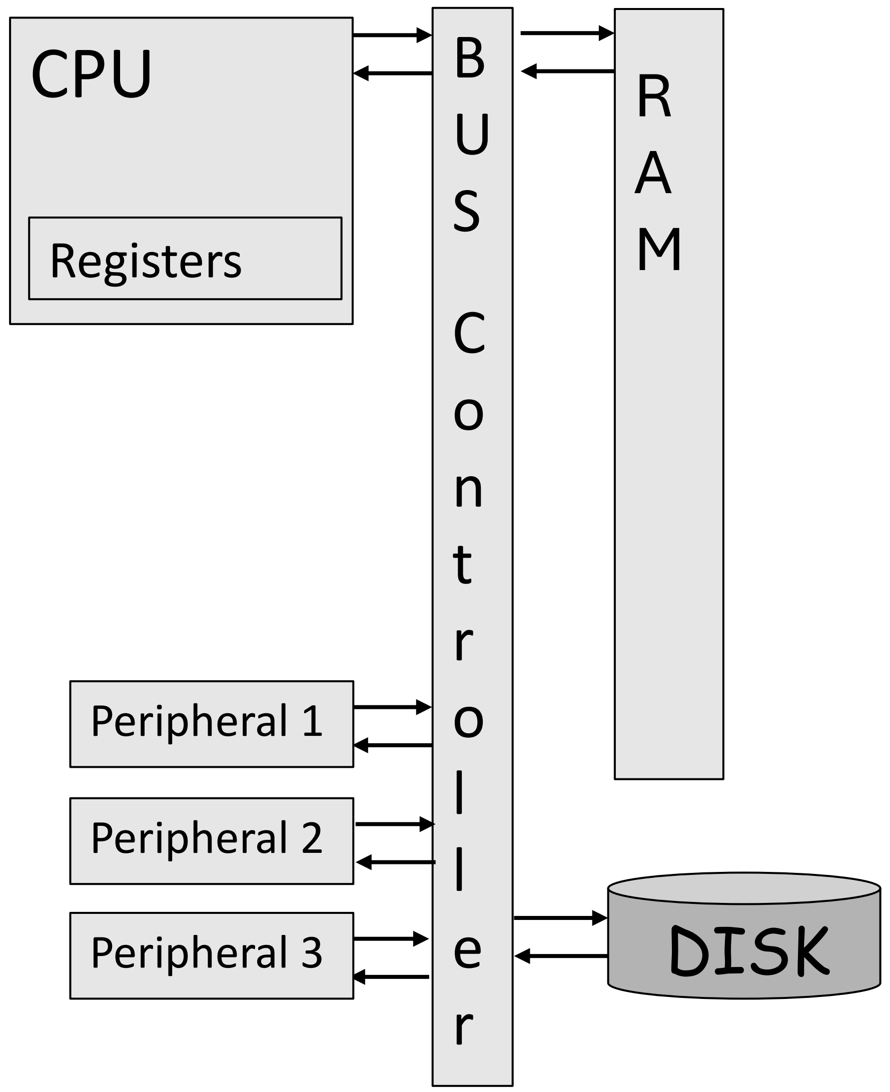


Main differences:

- CPU
  - Multi-core
  - Local shared cache, speculative, pipelined
  - At boot:
    - Only one core is active

- BUS Controller
  - Motherboard
  - Connects peripherals
    - Graphics card
    - Network card
    - Mouse, keyboard
    - Often are computers themselves (simple model)

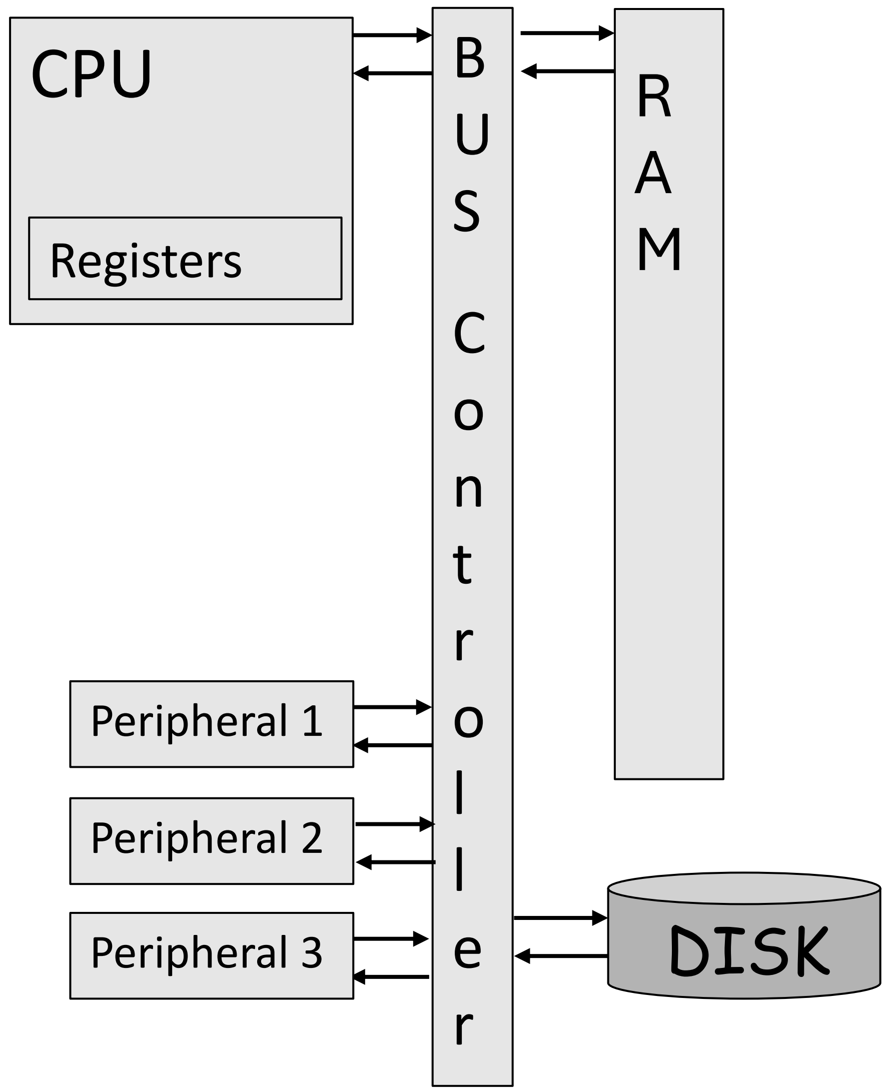
Updated system model



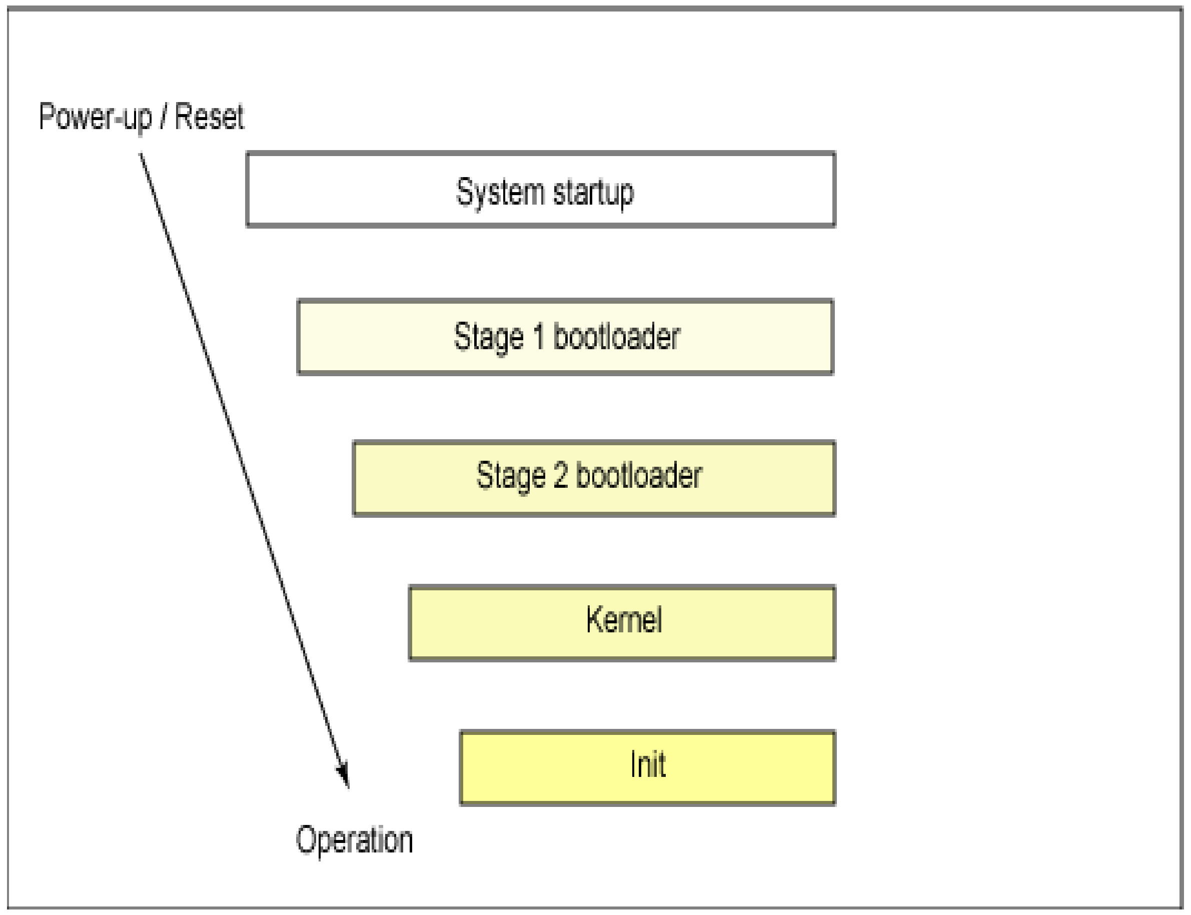
New challenges:

- The HW doesn’t know where the OS resides or how to load it
- Bootloader is responsible for locating it, loading into RAM, and starting its exec
- Multiple bootable disks, with multiple partitions

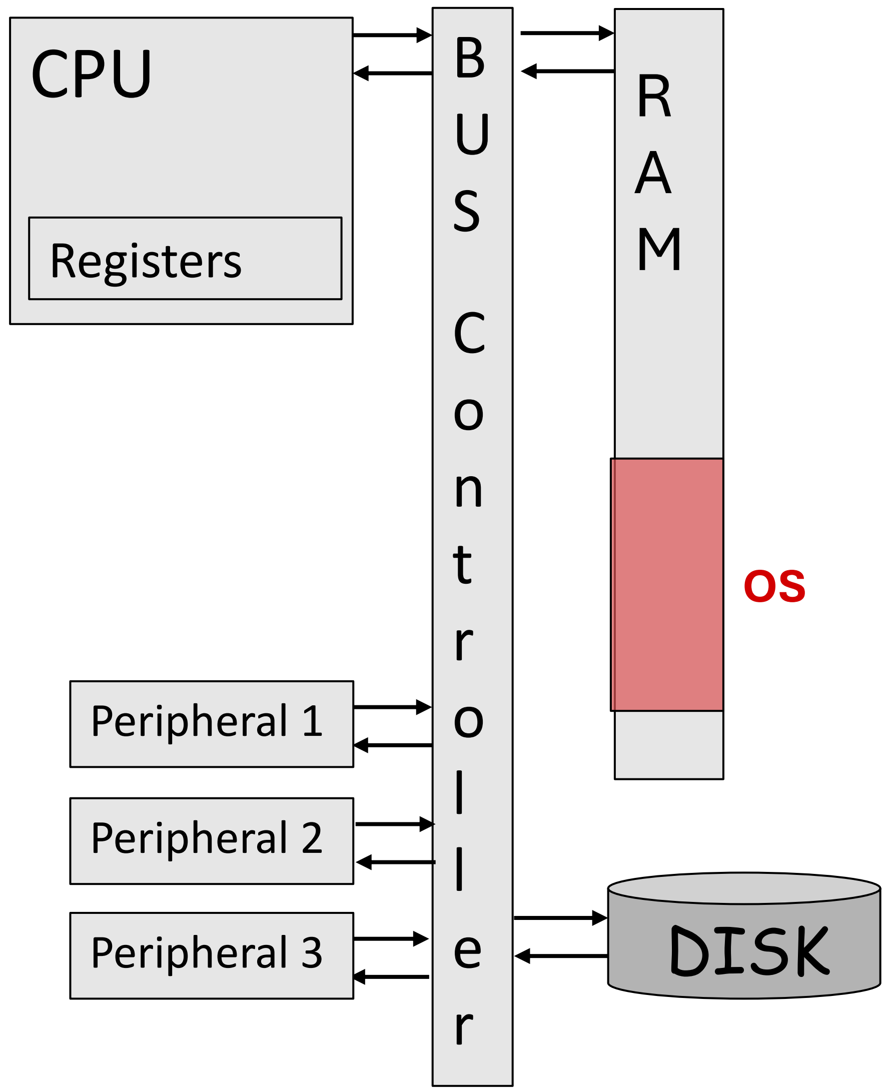
Updated system model



Secure boot chain in PCs:



Updated system model



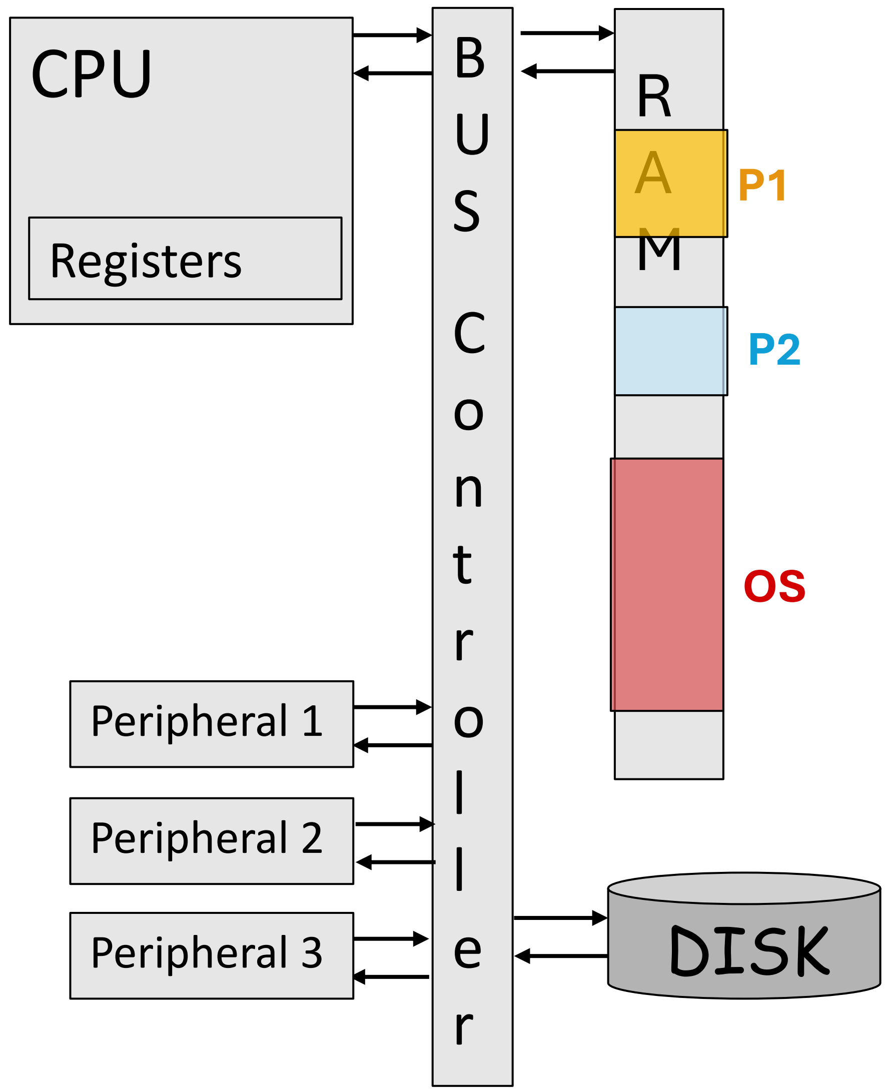
Afterwards we have

- Running operating system

At this point, we trust that the correct OS is running.

What does the OS do?

Updated system model



Afterwards we have

- Running operating system

At this point, we trust that the correct OS is running.

What does the OS do?

- Load other processes
  - ***Provide isolation***
- Provide access to system resources
  - Disk, peripherals, etc
  - ***Secure & indirect access***

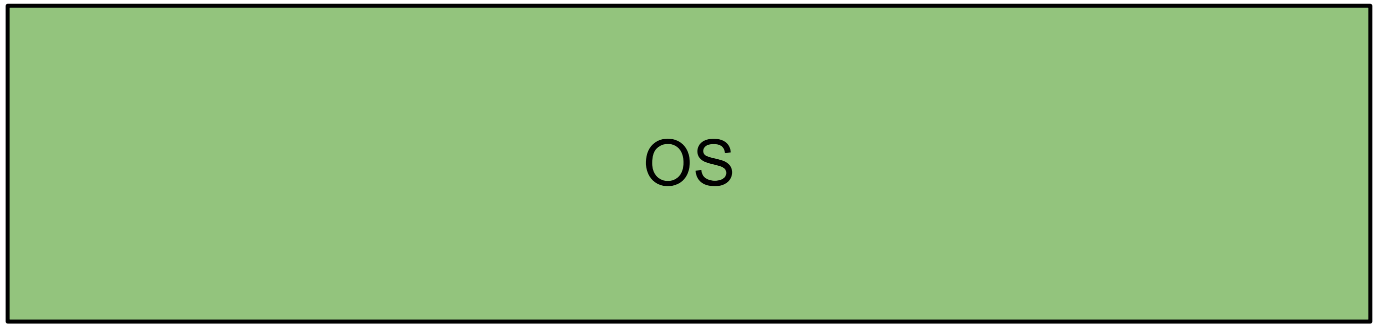
Process isolation

Definition:

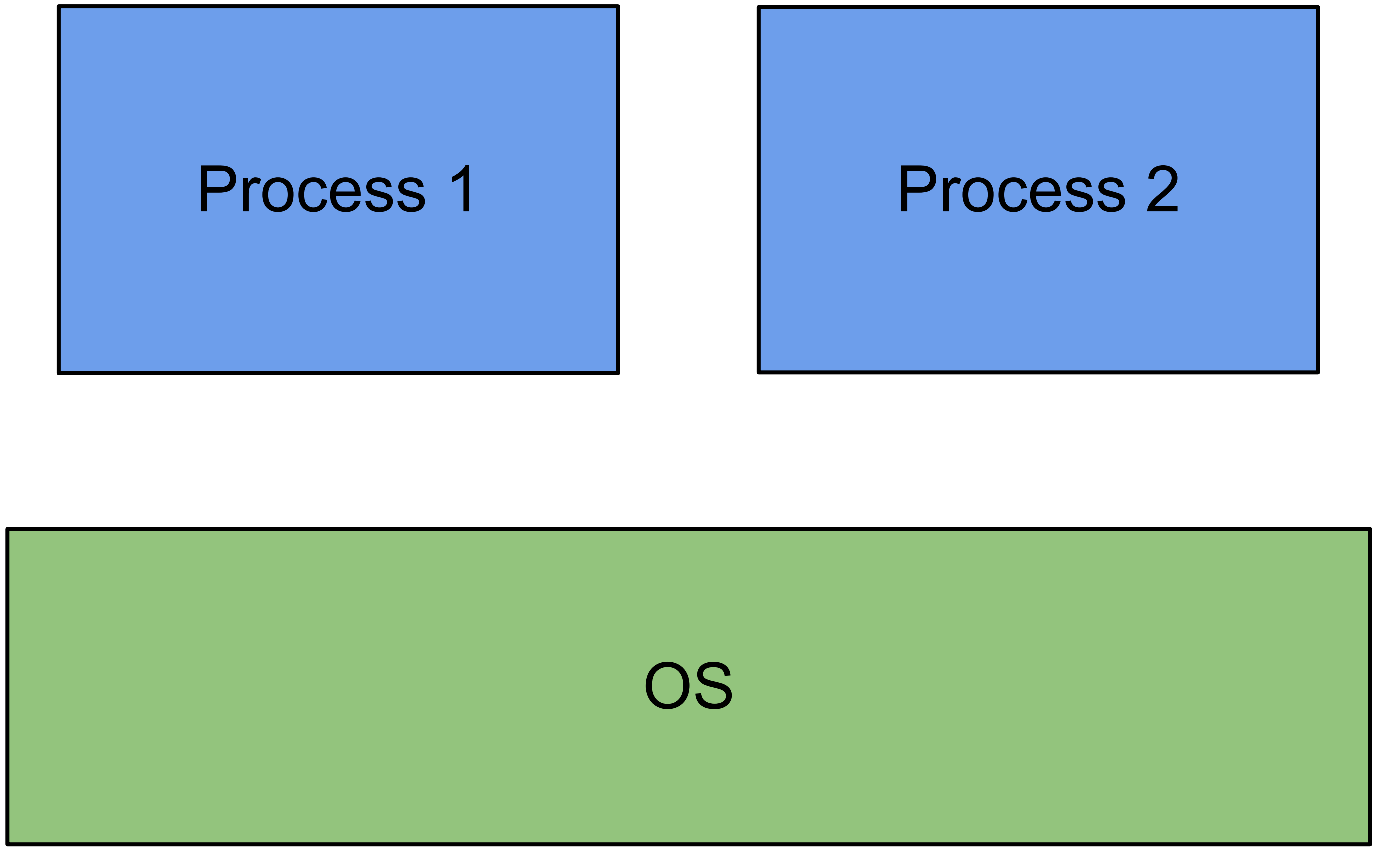
The OS (leveraging some hardware support) is responsible for assuring that a given process can not interfere (read from, write to , tamper with) another process in the system

One of the most fundamental concepts and important goals in systems security!

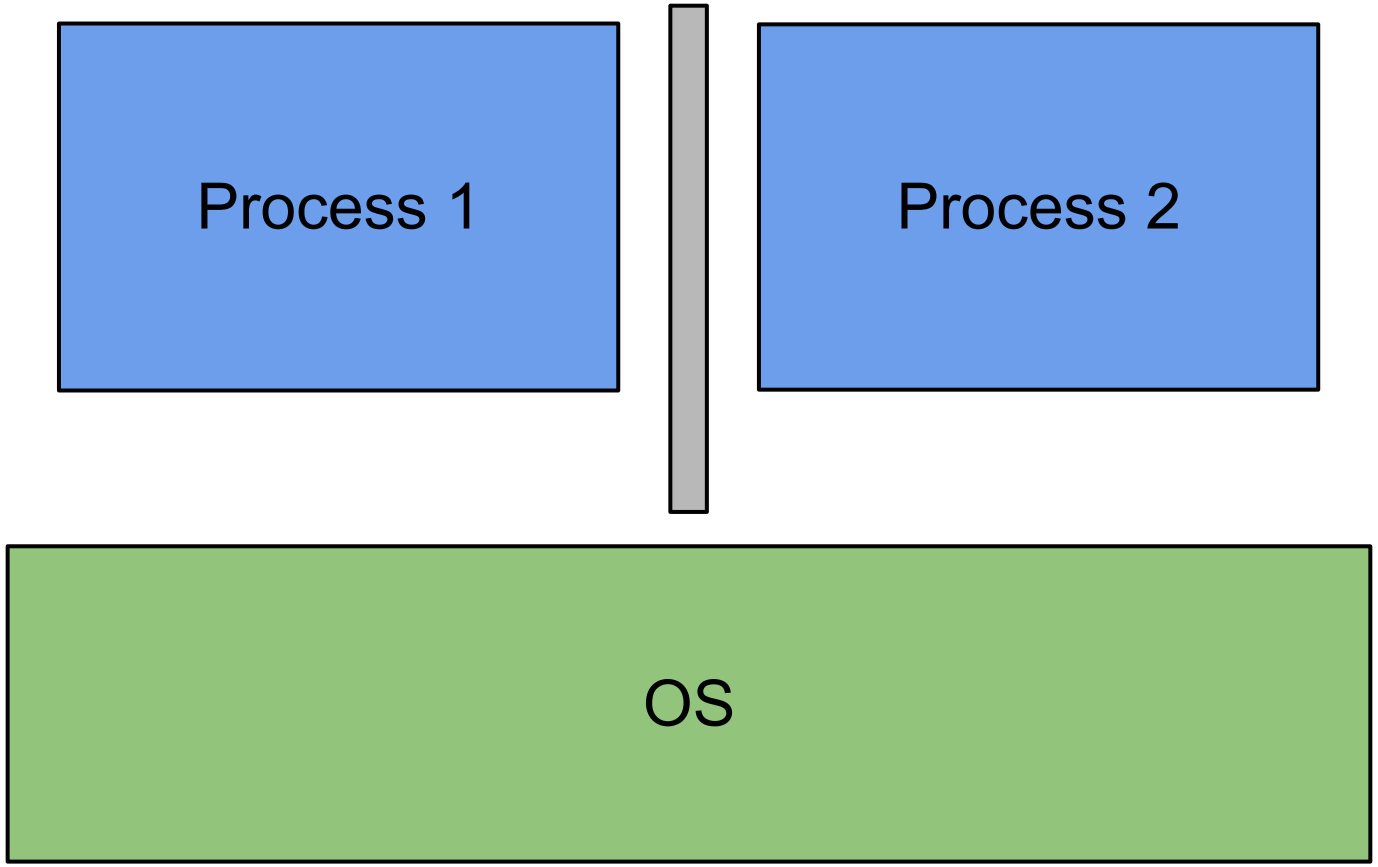
Process isolation



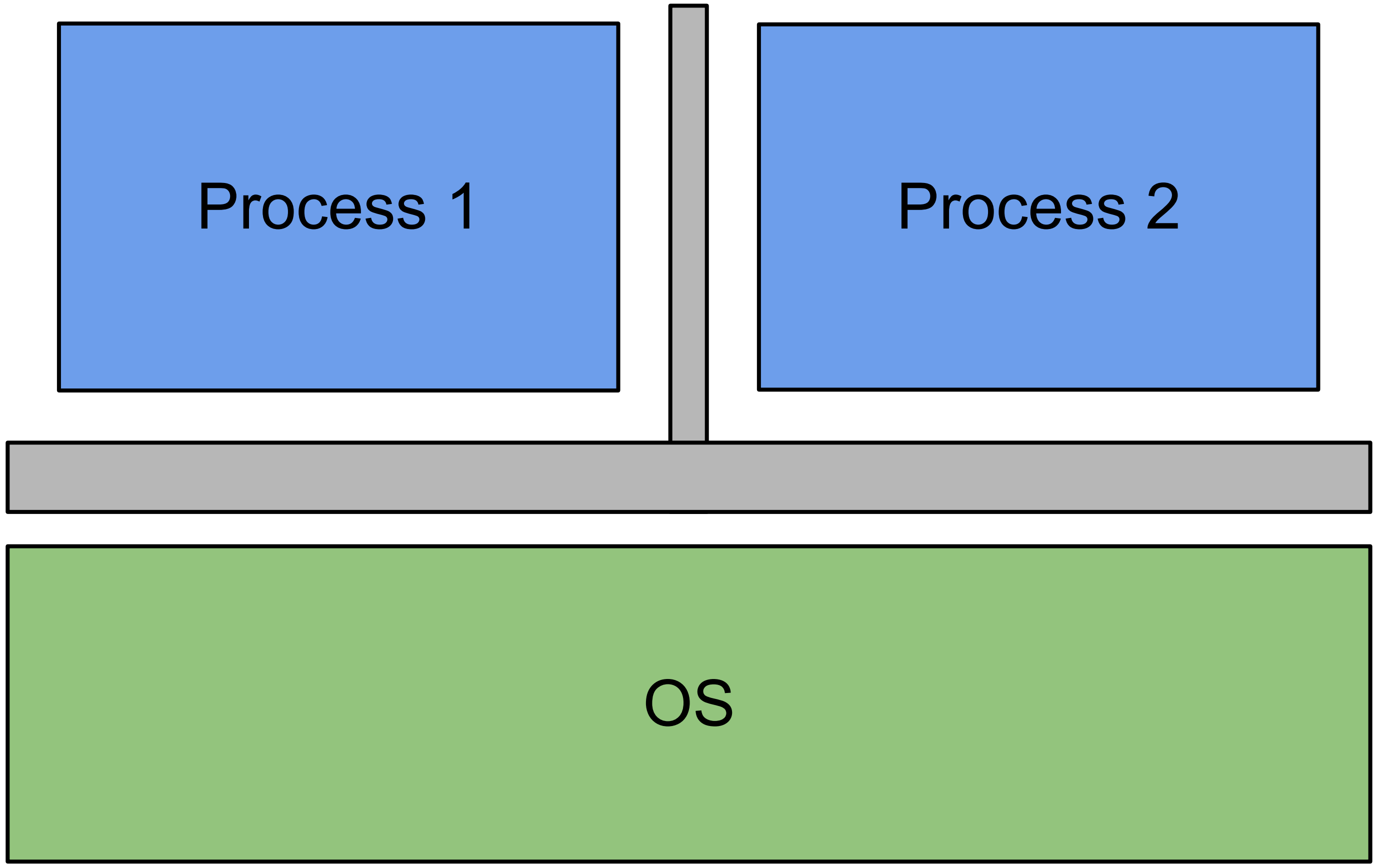
Process isolation



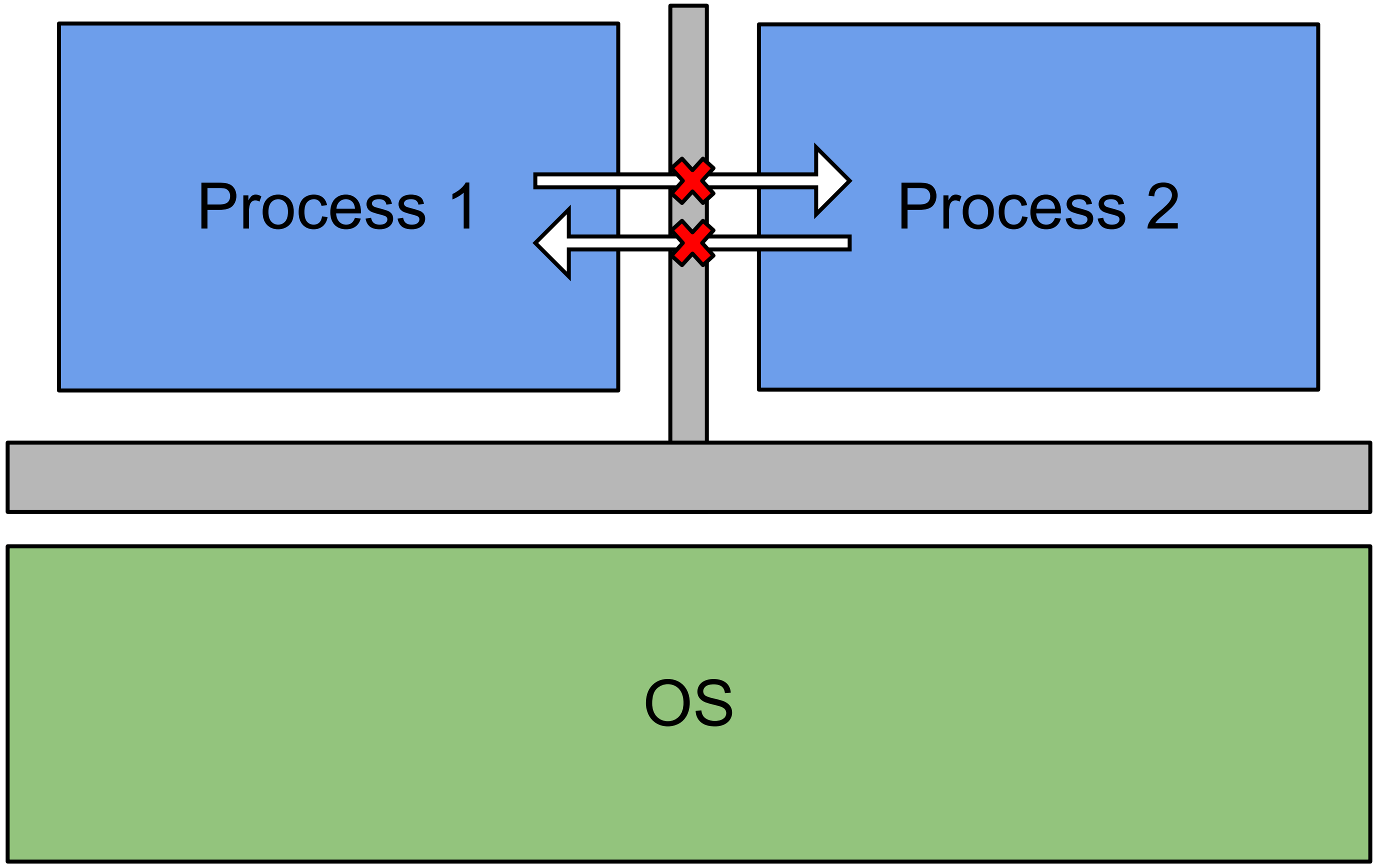
Process isolation



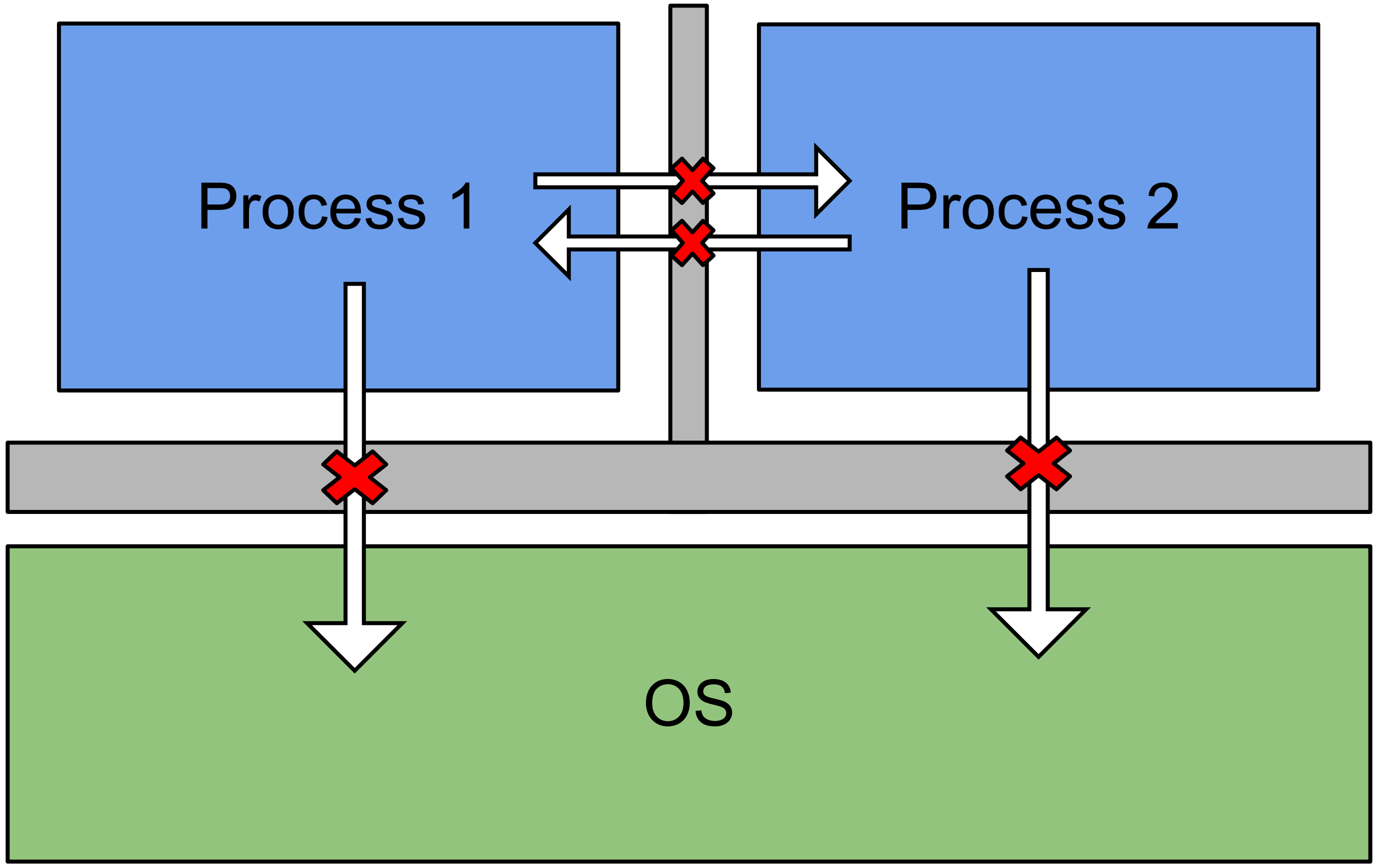
Process isolation



Process isolation



Process isolation



Privilege Levels

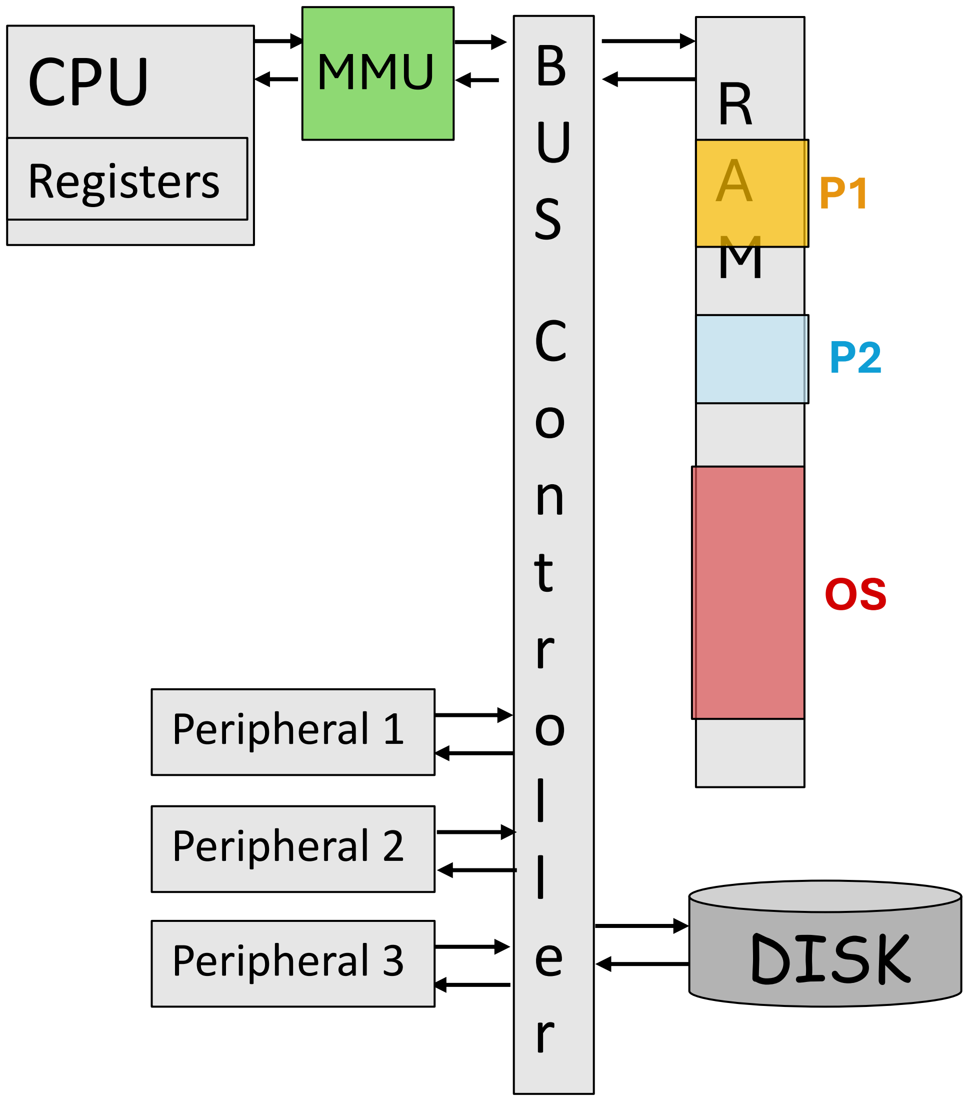
Definition:

Modern systems split software into different modules, each module having a certain level of access to system and processor resources.

Example:

- OS manages/controls user applications

Updated system model



How does the OS gain higher privilege?

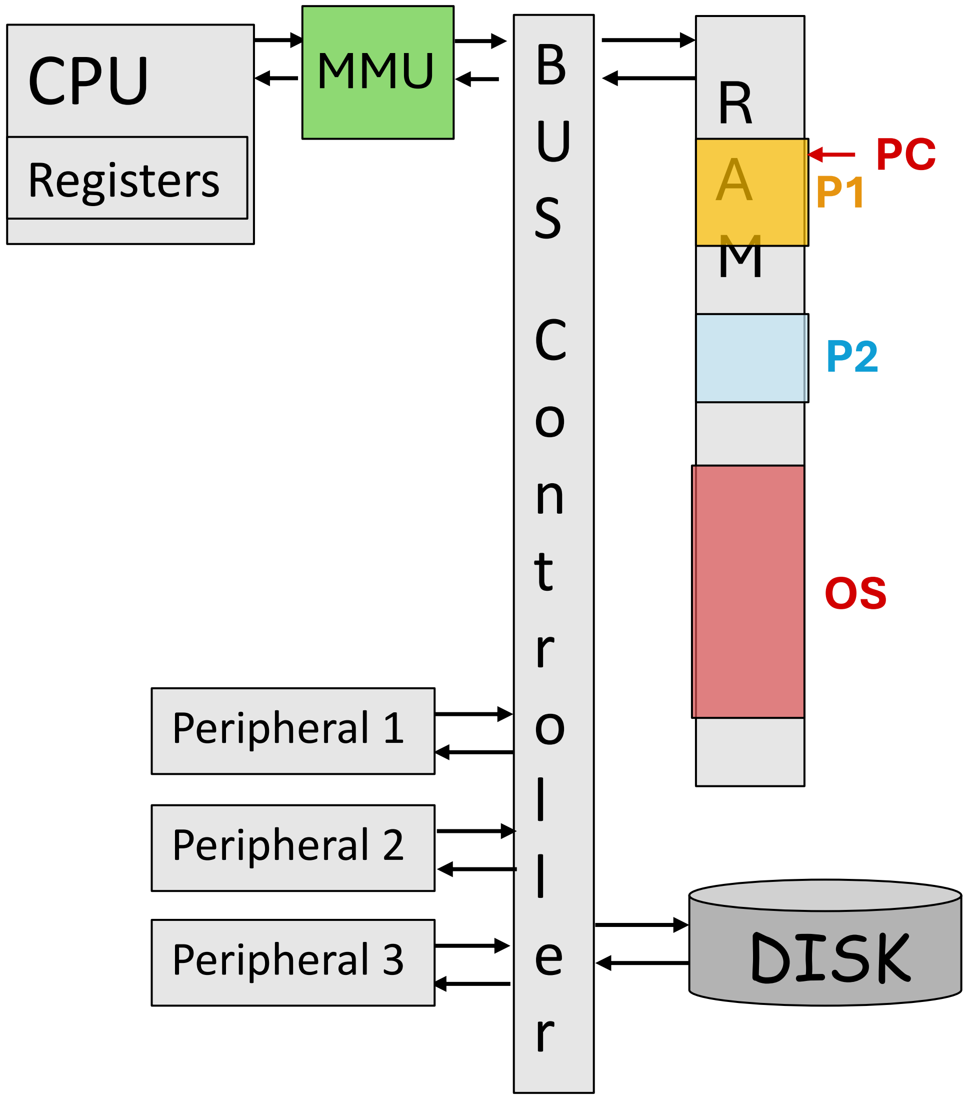
- It is the first software to run

How does it provide process isolation?

Memory Management Unit (MMU)

- Translates address referenced by instructions based on the current process
- Virtual Address vs. Physical Addresses

Updated system model

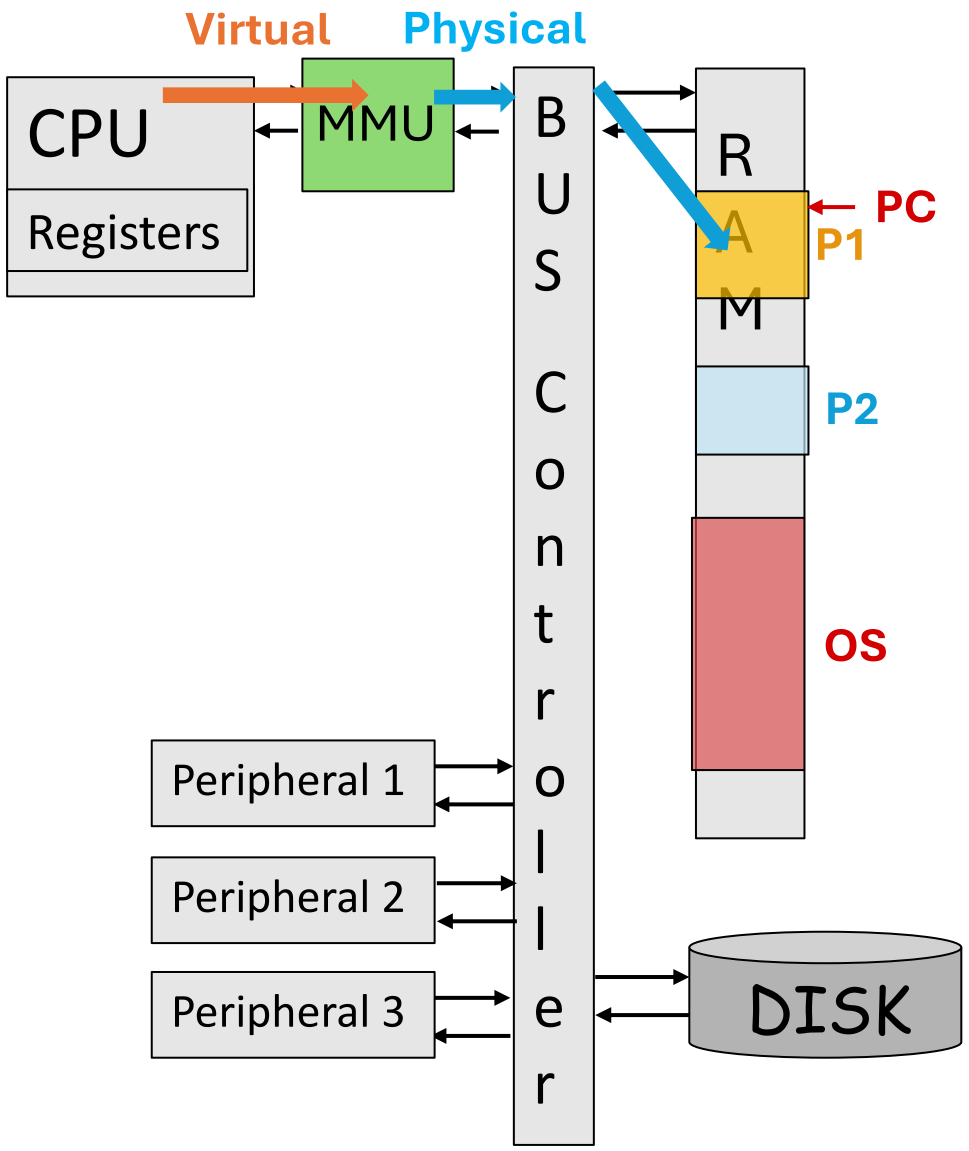


Quick review...

MMU Translation:

- MMU interposes every memory access by the CPU
- Translates it based on the current process

Updated system model



Quick review...

MMU Translation:

- MMU interposes every memory access by the CPU
- Translates it based on the current process
- Processor has no choice but to access its own memory
- Translation is based on page tables
    - Assigned and created by OS

See CS350 lectures for more....

Virtualization

Enables running (or simulating) multiple full computer systems

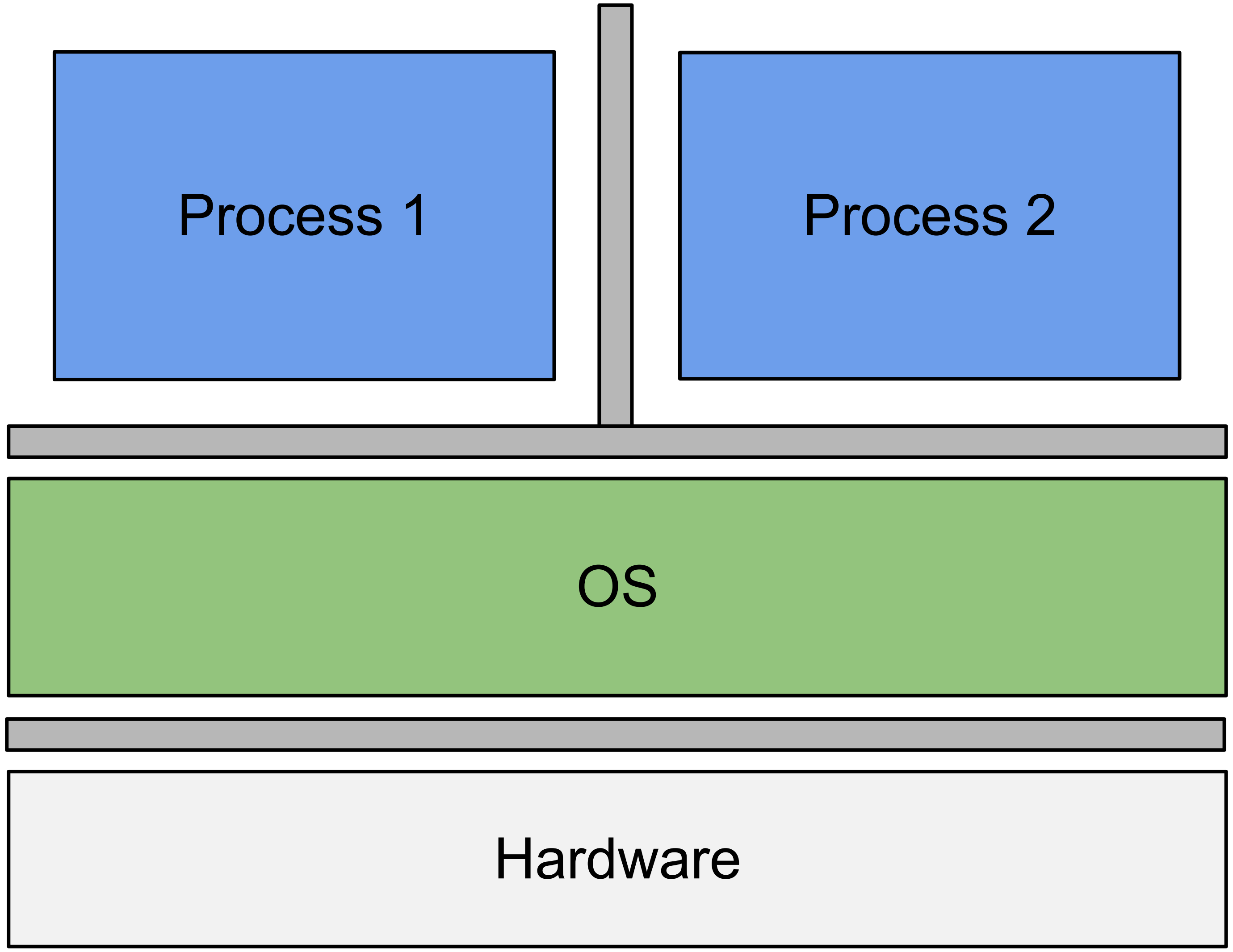
- Isolate workloads
- Run multiple operating systems atop a single host
- Strong isolation strength
  - Can create a level of separation between OS/kernel and the hardware

Various abstractions:

- Full virtualization: guest OS runs unmodified with virtualized HW
  - Interfaces with a VMM
- Paravirtualization: guest OS is aware it is virtualized
  - Instrumented to interact with VMM
- Whole-system emulation: entire system is virtualized
  - QEMU running in user-space

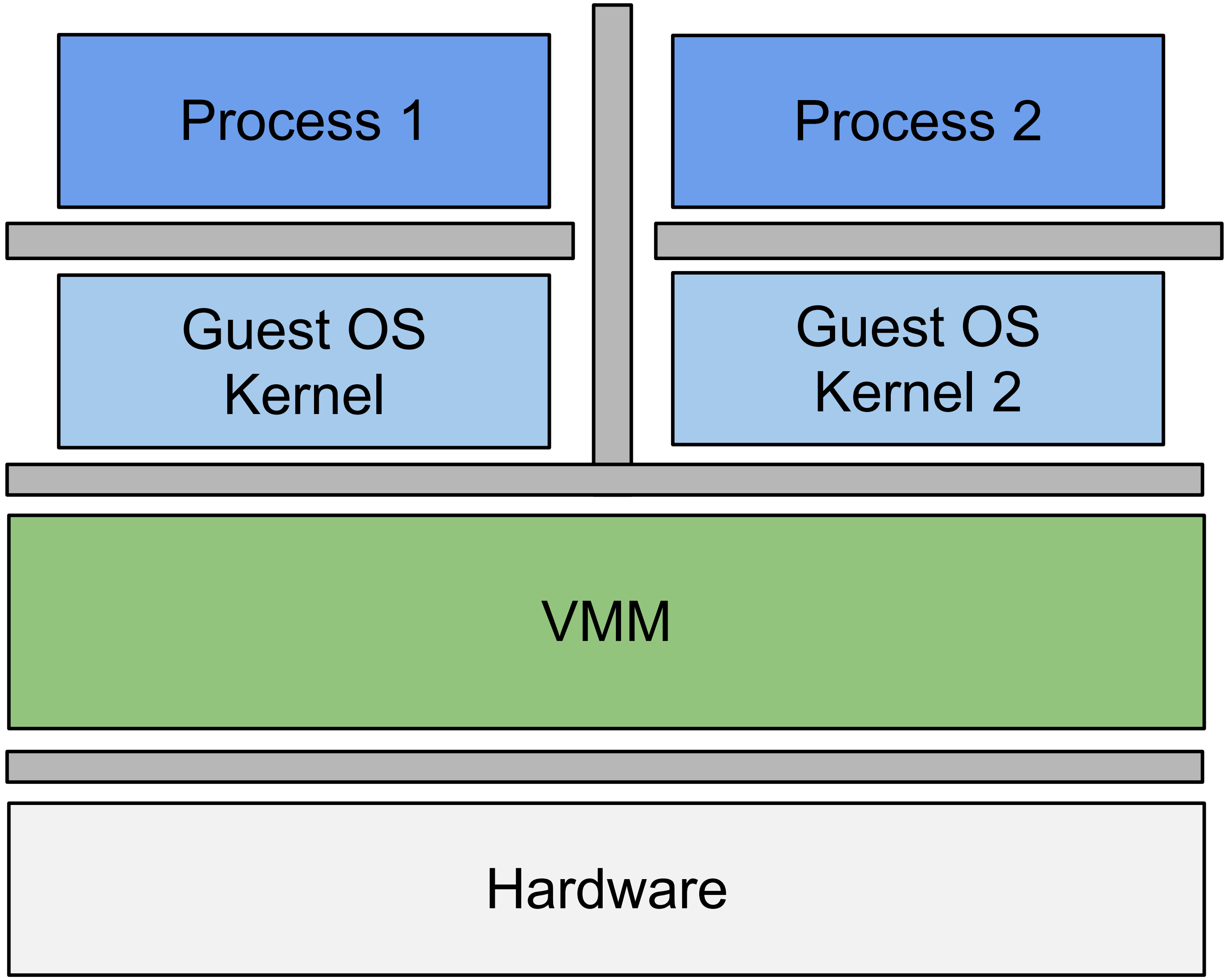
Virtualization Example:

Baseline OS-based isolation



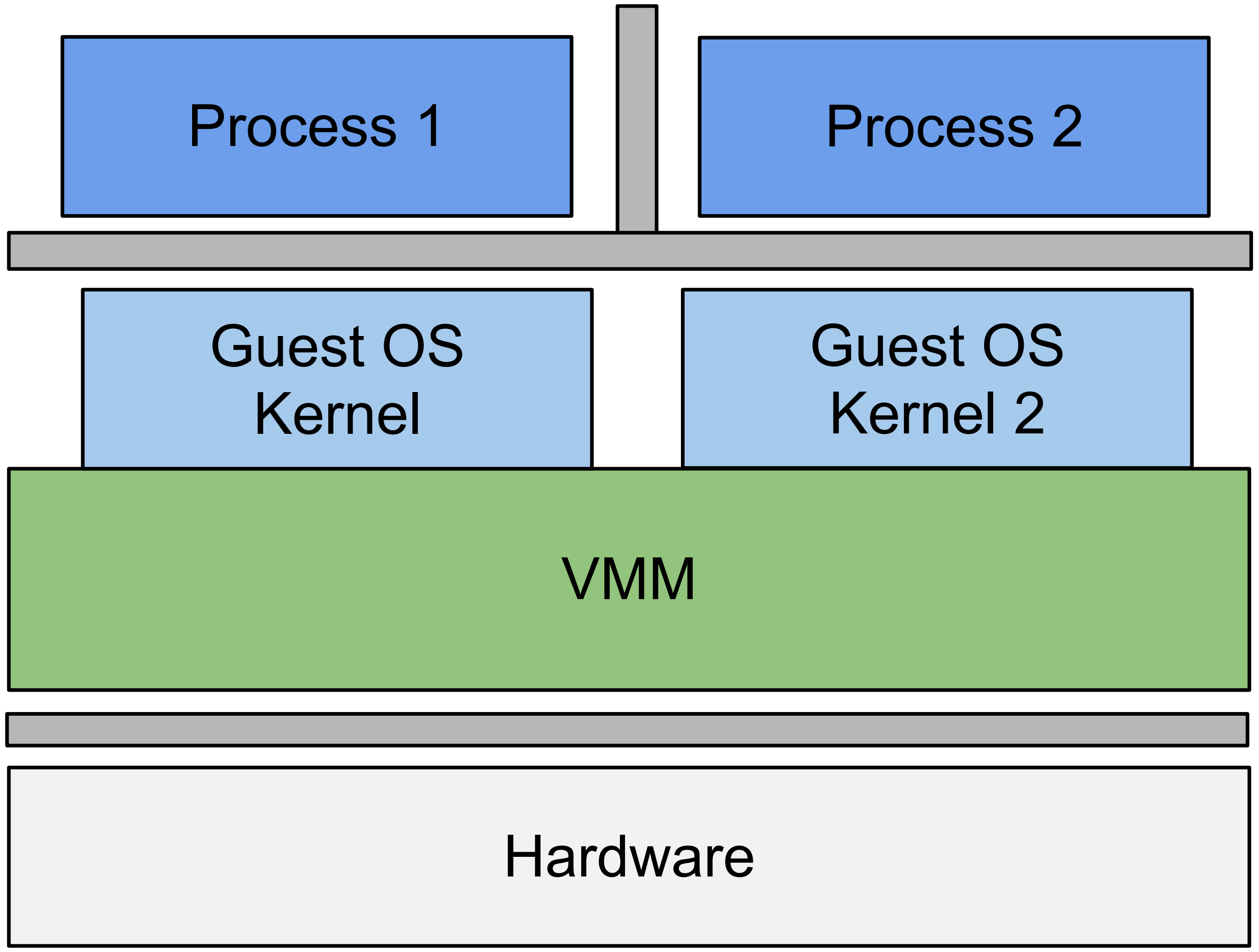
Virtualization Example:

Full virtualization: Guest OS run atop Virtual Machine Manager (VMM)



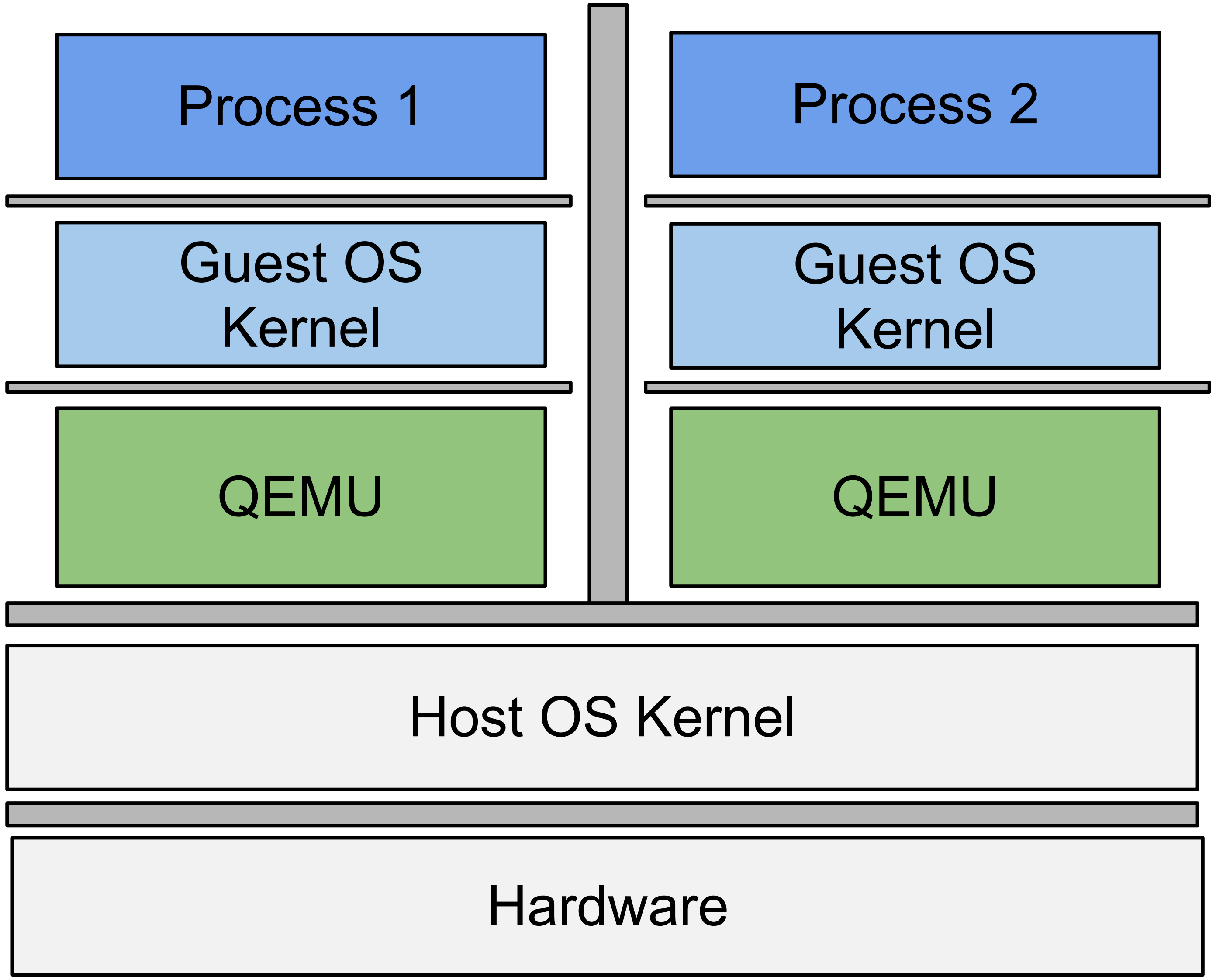
Virtualization Example:

Paravirtulization: instrument the guest kernel with hypercalls to VMM



Virtualization Example:

Whole System Emulation: run the entire stack in user-space



Compartmentalization

Definition: *compartmentalization of a program P is (1) a policy to separate P into two or more protection domains (called compartments), and (2) the enforcement of this policy at runtime*

- They can be applied to any program
  - ***Applications***
  - OS / kernels
  - Hypervisors
  - Firmware
- Example:
  - Within a single application, an HTTP parser and a crypto library
  - Compartmentalize so bugs in HTTP parser cannot affect crpyto library

Compartmentalization

Key Idea:

- Restrict control and data flow in the application so that each compartment has the permissions it requires to do its job

- An application of ***principle of lease privilege***

Takes three steps:

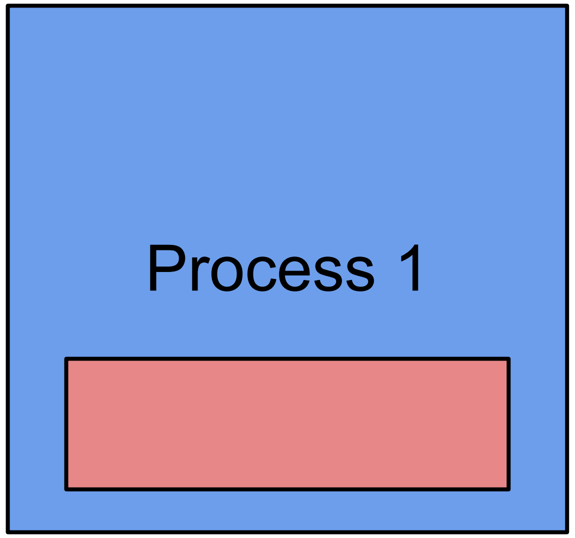
1. Policy definition: how many compartments, what goes in each?
2. Data classification: what data is shared? what is private?
3. Integrate isolation mechanism & data sharing strategy

Compartmentalization

Types of trust models:

Sandbox:

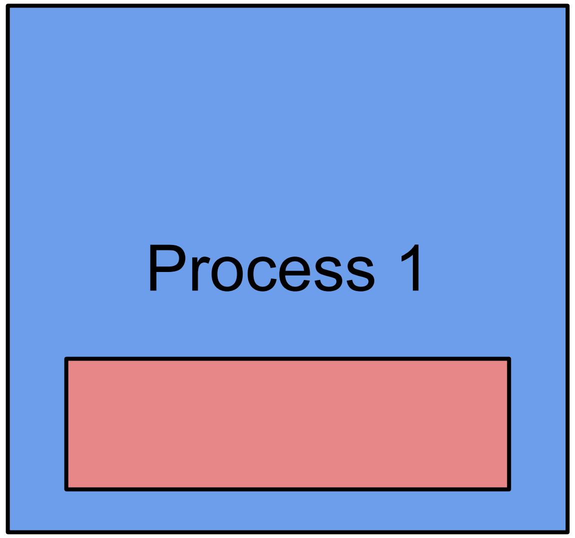
- Part of the program is untrusted
- Placed in a compartment to isolated the rest of the program from it



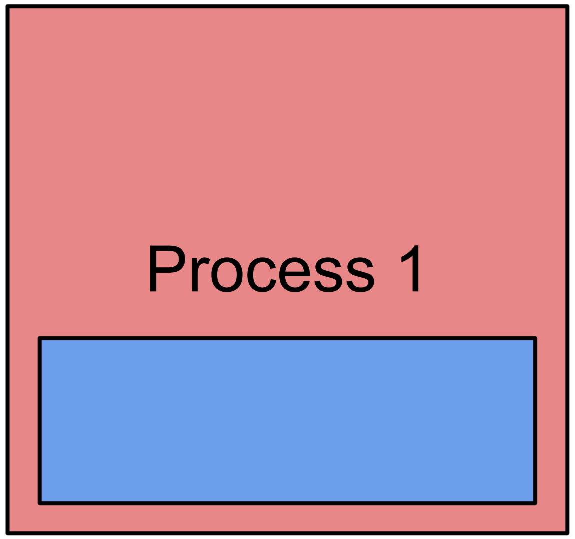
Compartmentalization

Types of trust models:

Sandbox:
- Part of the program is untrusted
- Placed in a compartment to isolated the rest of the program from it



Safebox:
- Part of the program is security critical
- Placed in a compartment to isolated it from the rest of the program

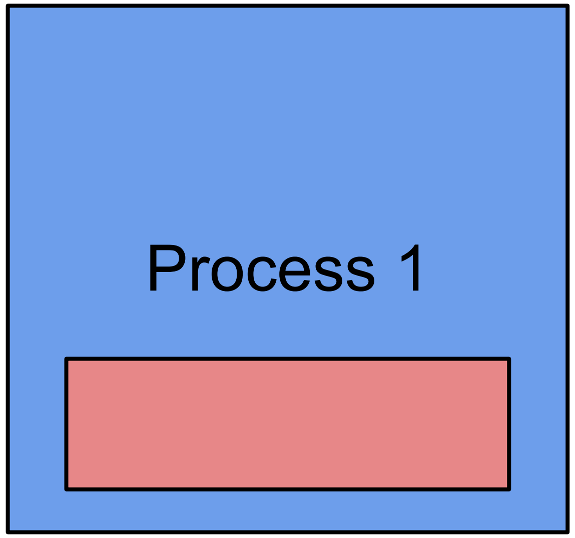


Compartmentalization

Types of trust models:

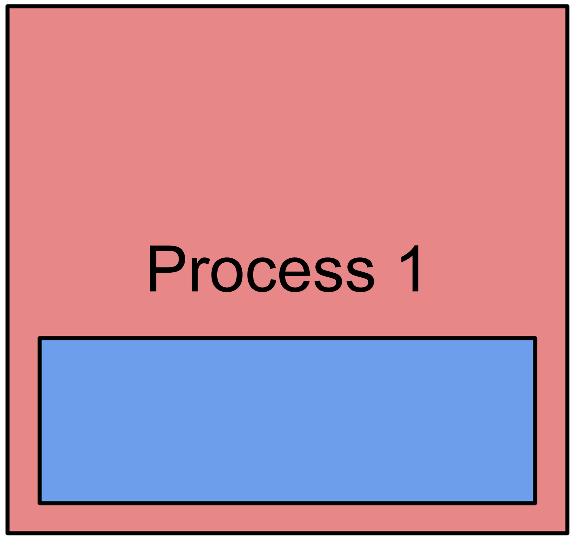
Sandbox:

- Part of the program is untrusted
- Placed in a compartment to isolated the rest of the program from it



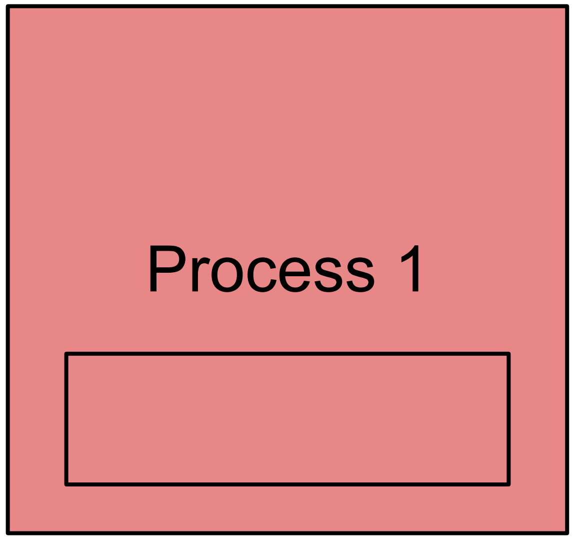
Safebox:

- Part of the program is security critical
- Placed in a compartment to isolated it from the rest of the program



Mutual distrust:

- Compartments all distrust each other
- Isolated so that they cannot access each other



Compartmentalization

Isolation mechanism

- Must be established outside of and before the compartment
- Similarly to OS ***inter-process*** isolation
- ***Intra-process*** isolation (e.g., sandboxing) works if the isolating logic initializes before the confined code

What if a compartment performs system calls?

- Application-level (intra-process) enforcement cannot make these restrictions
- Per-process flags could allow privileged operations
  - See *linux capabilities*
- Further restrictions must be enforced at OS / kernel boundary
- How can we enable this fine-grained sandboxing?

Sandboxing

Case study: seccomp

What if I know the exact system calls that should be used by the compartment?

seccomp:

- Prevents execution of certain system calls by an application
- Implements a customizable filter
- Designed to sandbox untrusted code that is compute intensive

Example: strict mode

- Only permits *read(), write(), _exit(), sigreturn()*
- Any other system call leads to SIGKILL

Sandboxing

Seccomp: uses a Berkeley Packet Filter (BPF)

Can filter based on system call number and argument values

Steps to use a BPF:

- Construct filter in the BPF rules
- Install filter using seccomp()
- exec() new program or invoke function in dynamically loaded libraries

Once installed, every system call triggers execution of filter

Sandboxing

BPF is stateless:

- Filtering decision is solely based on only the current system call

Can be expanded to an *extended BPF (eBPF)* which are stateful

- Allows more complex to be performed quickly and safely

- Read more on seccomp
    - Important highlight: *"System call filtering isn't a sandbox. It provides a clearly defined mechanism for minimizing the exposed kernel surface. It is meant to be a tool for sandbox developers to use."*

A3 → exploiting *seccomp-based sandboxes*

End of class

Reminders:

- A2 due June 20
- A3 coming up
  - Released June 24
  - Due July 11

Additional Resources:

- A Secure and Reliable Bootstrap Architecture
- Software Security: Princples, Policies, and Protection
- SoK: Software Compartmentalization
- [Linux Security] Understand and Practice Seccomp Syscall Filter