Systems Programming: A Tour Of Computer Systems

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

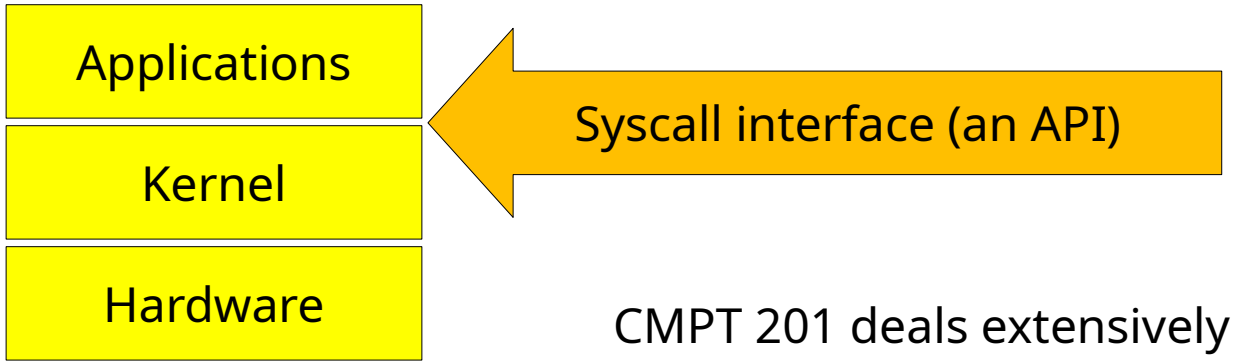
Topics

- For a program to *run*, what is needed?
- How does a computer's *hardware* work?
- What does the *OS Kernel* do?
- How does a *program interact with the OS*?

From Logic Gates To Software

OS Stack

- Let's discuss the terminology necessary for the course and generally for computer systems.
- **OS Stack**
  - ***Layers of services, each building on lower layer***



OS Stack

CMPT 201 deals extensively with system calls

Systems Programming

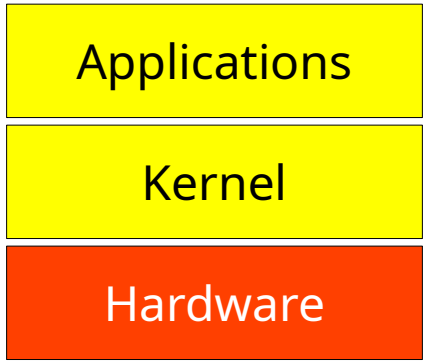
- **Systems programming:**

  Low-level programming that directly interacts with hardware or the OS, often using system calls

  - Done using systems languages, also called low-level languages, such as C, C++, Rust, which offer e.g. raw memory access

- Higher-level or application programming
  - Often don’t need a systems language, although sometimes used for performance reasons
  - Choose a language that fits your goals!

Hardware Layer



Components in Computing

- Two fundamental components in computing:
  - **Computation:**

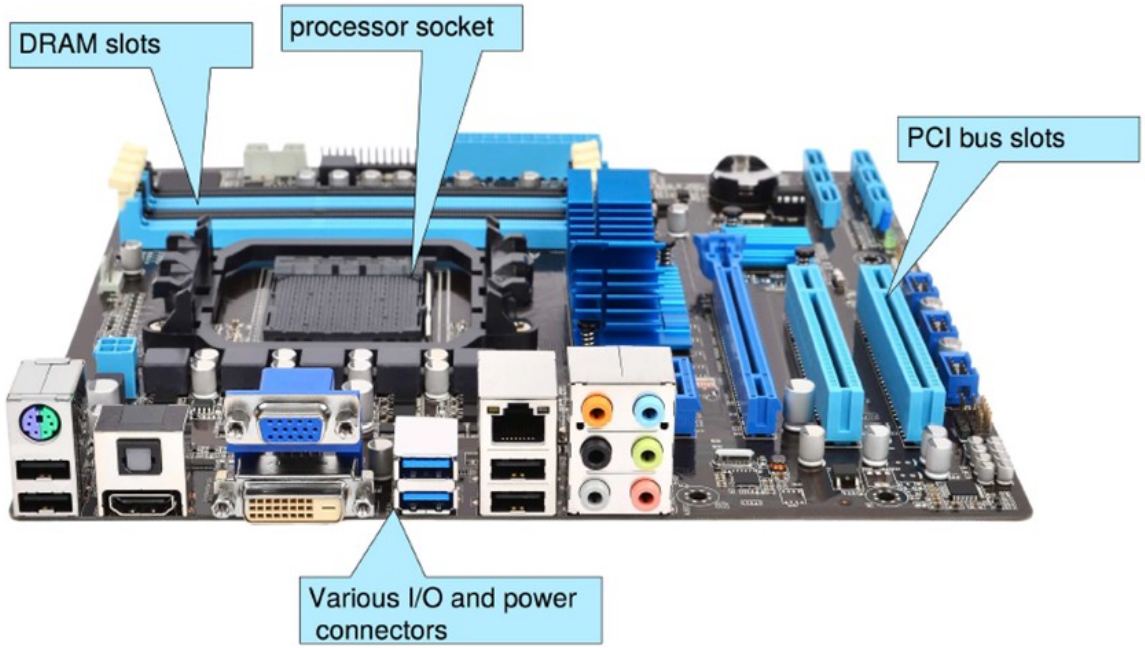
    Handled by the CPU
  - **Data:**

    Handled by memory (RAM, storage, …)
- E.g., *a* + *b* => *c*
  - *What is the computation?*
  - *What is the data?*

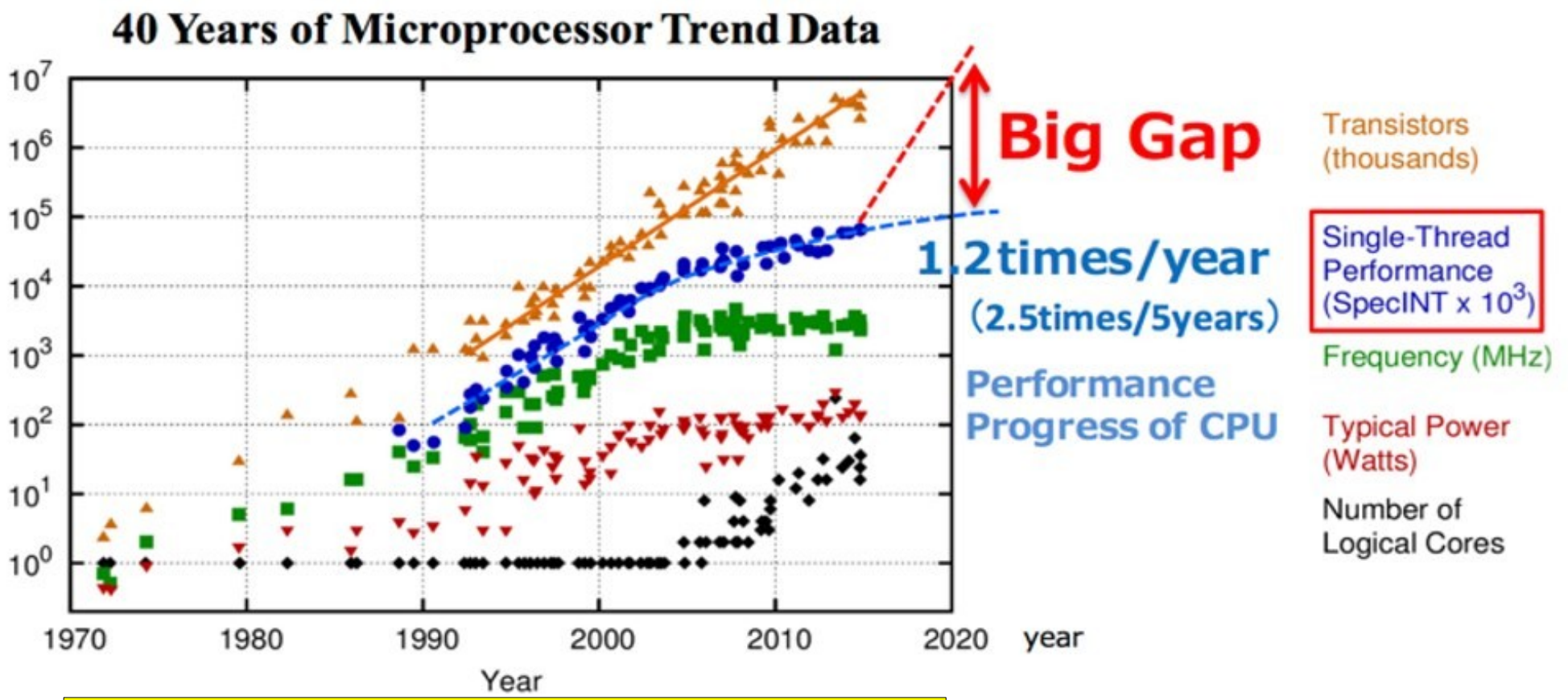
PC Motherboard

- von Neumann architecture
  - Current fundamental model of computer design.
  - Fetch *data* from *memory* to provide to the *CPU* for *computation*.
- **Hardware components:**

  CPU, ***memory***, and ***I/O devices***.



Evolution of CPU: Moore's Law



Pre early 2000: *frequency* x 2 every 18 months
Post 2005: *core count* x 2 every 18 months

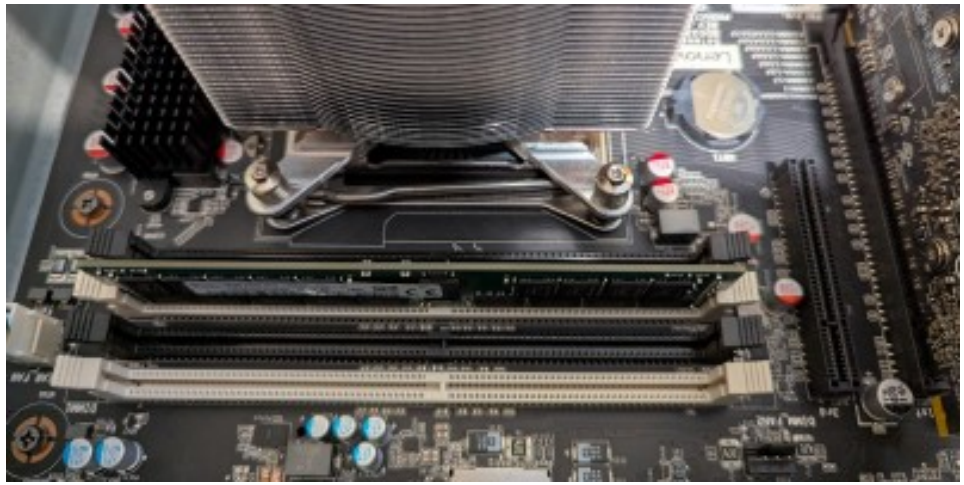
Reference: Ahmet Ceyhan, Interconnects for Future Technology Generations: CMOS with Copper/Low-κ and Beyond, PhD Thesis, 2018

Evolution of Memory

- CPU needs data from memory
  - CPU was getting faster, so memory access had to get faster too
  - Speed of memory access limited by *memory chip speed*, and *speed of light!*
  - Memory is far away from CPU, and much too slow

CPU

RAM

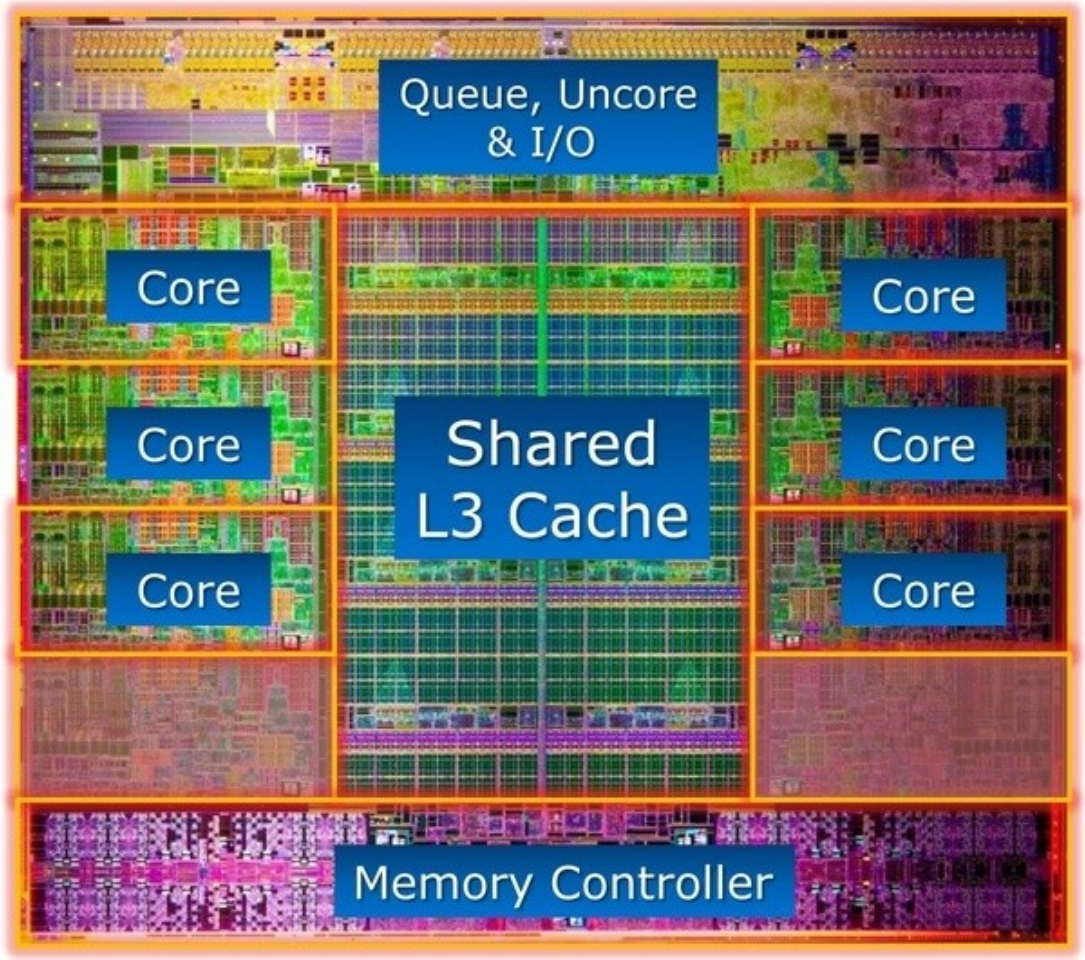


CPU vs Memory Speed

- “Solve” speed gap between CPU and memory access
- **Registers:**
  - Very small memory inside a CPU; hold data items from memory
  - Very close to CPU, so very fast access to data
- **Cache:**
  - Much larger in size than registers, but much smaller than memory
  - Quite close (physical distance) to CPU, so faster access times
  - Nowadays processors have many caches:
    L1 cache ~512KB ***(smallest, closest, fastest)***
    L2 cache ~8MB
    L3 cache ~32MB ***(largest, slowest)***

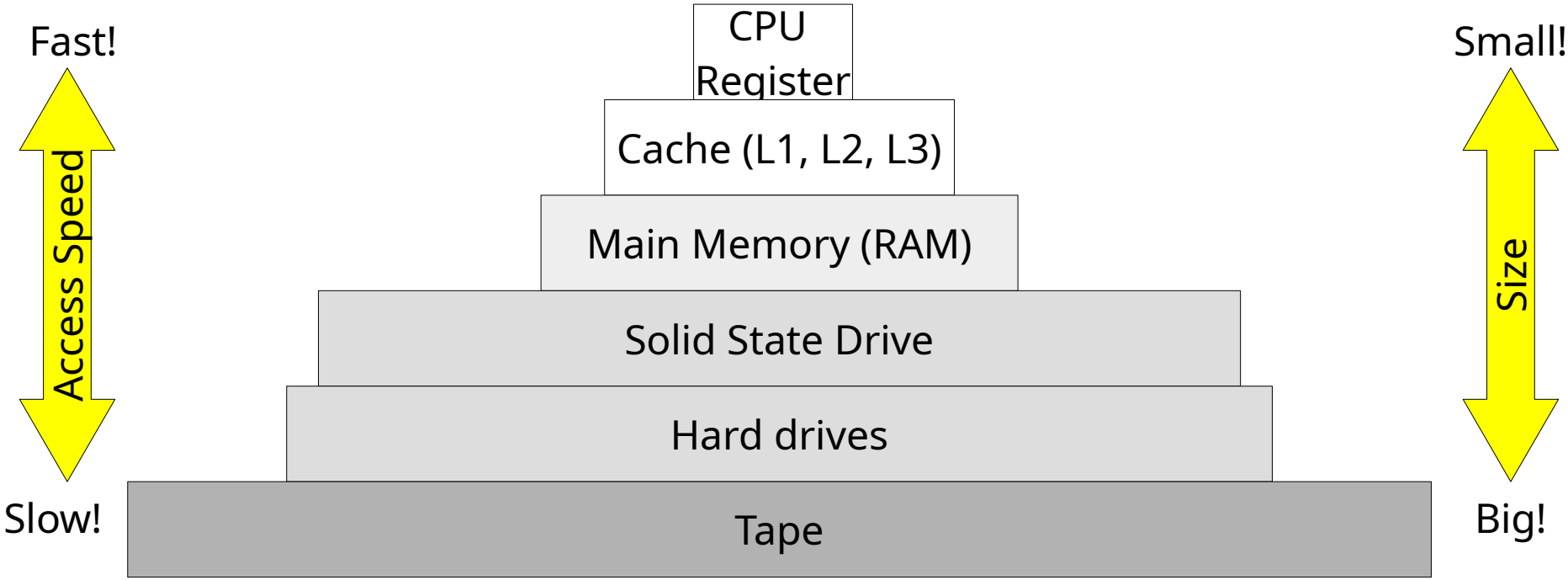
Multi-core Processor

- Desktop CPU today
  - One processor chip
  - **Multiple cores**
    - Many caches, some shared, some private
    - Some shared execution units



Memory Hierarchy

- Want the best of all worlds: **fast access, large capacity, low price**
- Intelligently bring data in from large-slow devices (hard drives) into small-fast devices (memory, cache)



Memory Hierarchy

- Trade-offs
  - **Cost**
    Bigger capacity means more expensive
  - **Distance** and **Access Speed**
    Faster means closer to CPU
  - **Persistence**
    Ability to retain data through power loss
    - **Commit** generally means making a temporary change permanent (by analogy, “git commit”): here, copying to *persistent* storage
  - **Reliability**
    How likely is it to fail – not the same as persistence!
    - ECC server memory is *reliable* but not *persistent;* a cheap SSD may be *persistent* but not *reliable*

CPU Architectures

- **Instruction Set Architectures (ISA)**
  - ***Defines a set of instructions the CPU can perform***
  - Compiler translates C programs into machine instructions
  - Example ISAs: **x86, ARM, RISC-V** (“risk-five”)
- **32-bit** vs. **64-bit** architectures
  - For CMPT 201, we care most about 32-bit vs 64-bit because it ***determines (natural) register size***

Audience Participation - Pointers

- What is a *pointer* in your C program?

a) A memory address.
b) A variable storing a memory address.
c) The data stored in an array.
d) The address of the current instruction.

Audience Participation - Pointers

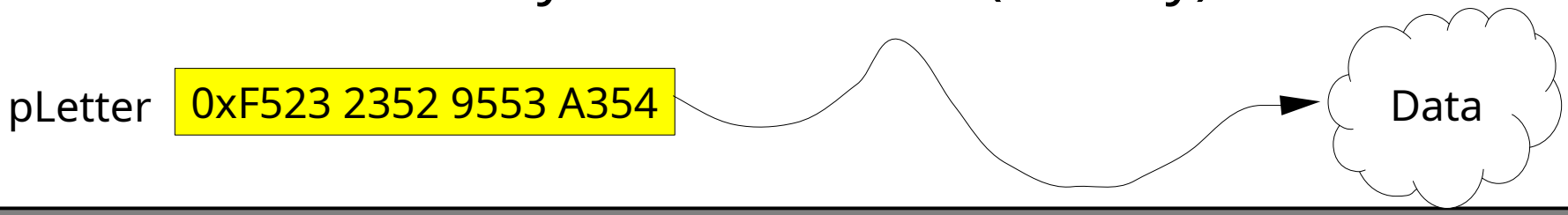
- Which of the following is true for this code?

```
char* pLetter;
long long* pCounter;
```

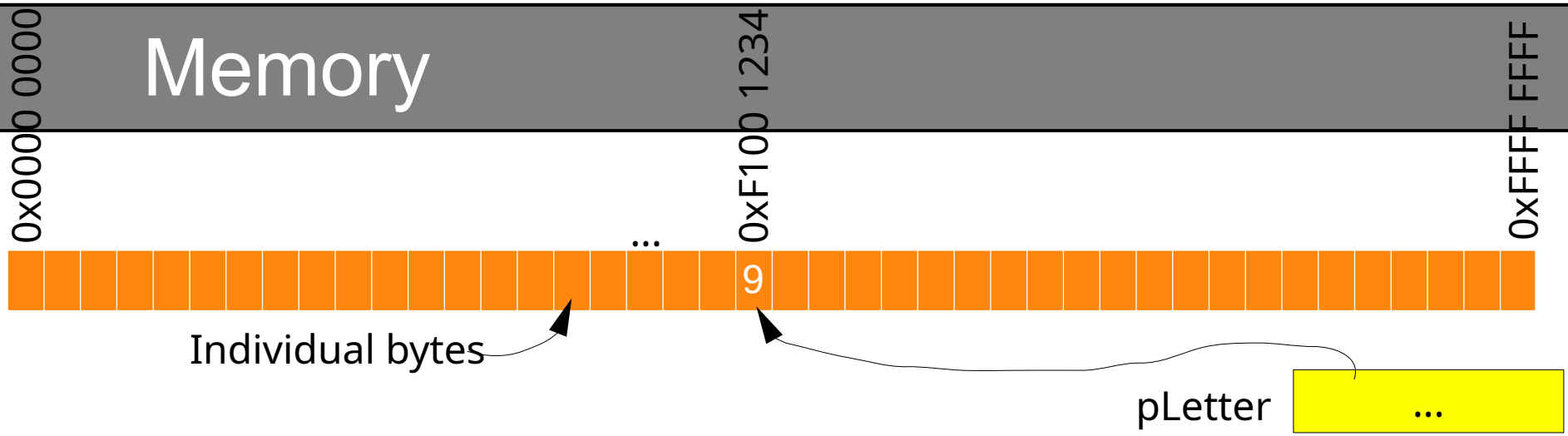
a) `sizeof(pLetter) < sizeof(pCounter)`
b) `sizeof(pLetter) > sizeof(pCounter)`
c) `sizeof(pLetter) == sizeof(pCounter)`
d) Depends on if the system is 32-bit or 64-bit

32 vs 64 bit Register Size Implications

- 32-bit CPU can do 64-bit computations
  - But it's not as efficient: need multiple operations
- ***(General) register size = pointer variable size:***
  32-bit uses 32-bit pointers, 64-bit uses 64-bit pointers
  - Computers spend a lot of time doing pointer arithmetic!
- ***Pointer size determines size of memory address space***
  - Pointers *are* memory addresses
- Affects bus/memory channel width (loosely)

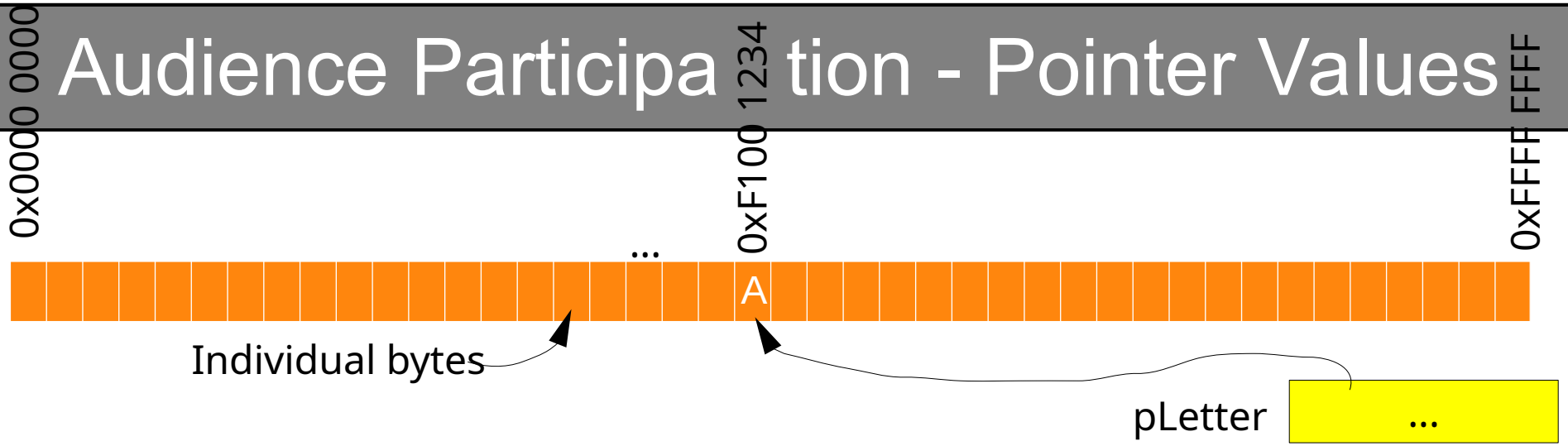


Memory



- Memory made up of bytes (1 byte = 8 bits)
  - Each byte has an address
- 32-bit vs 64-bit word size
  - The number of bits stored in a CPU's register.
- In a 32-bit system (32-bit word):
  - Addresses are 32-bits:
    0x0000 0000 to 0xFFFF FFFF
  - (Data is retrieved from memory 32-bits at a time (4 bytes)
    but memory addresses are still byte addresses)

Audience Participa tion - Pointer Values



- Which of the following is true?

```
char ch = 'A';
char* pLetter = &ch;
```

a) pLetter == 'A'
b) pLetter == 0x0000 000A
c) pLetter == 0xF100 1230
d) pLetter == 0xF100 1234

Audience Participation - Memory

- Which of the following is true?

a) 1,000 = MB, 1,000,000 = KB, 1,000,000,000 = GB
b) 1,000 = GB, 1,000,000 = MB, 1,000,000,000 = KB
c) 1,000 = KB, 1,000,000 = MB, 1,000,000,000 = GB
d) 1,000 = GB, 1,000,000 = KB, 1,000,000,000 = MB

- If memory (RAM) stored just *16 bytes* (16 locations), how many bits do we *need* in our address?

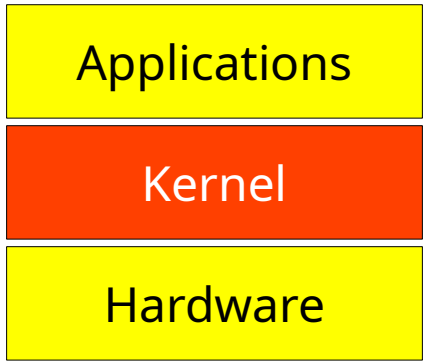
a) 2-bits
b) 4-bits
c) 8-bits
d) 16-bits

Why 64-bits?

- Why are most computers 64-bit architectures now?
  - Has a 64-bit register
  - Has a 64-bit pointer
  - Allows us to **address 2^{64} different bytes** in memory
    2^{64}= 16,000,000,000 GB = 16 Exabytes (VERY large)
- In a 32-bit architecture, how much memory can the CPU access?

  a) 65,526 bytes
  b) 2,147,483,648 bytes
  c) 4,294,967,296 bytes
  d) 18,446,744,073,709,551,616 bytes

Kernel Layer



What is the OS?

- Operating System (OS)
  - Central software managing the computer's resources
- OS Includes
  - **Kernel:**
    Main part that actively manages resources.
  - **Supporting tools:**
    such as GUI, command line;
    These are what differentiate Linux distributions (“distros”)

What does a Kernel do?

- *Resource management*
  - many programs want to access the hardware at the same time
  - kernel manages (mediates) access
- *Program control*
  - the kernel controls programs (running, stopping, etc.)
- *Protection*
  - the kernel provides protection (isolation) for users and programs
    - E.g., users can’t access each other’s data
    - E.g., programs can’t interfere with each other’s execution

Event-Driven

- When does a kernel do some work?
  - Generally, the OS lets other programs run and waits for something it needs to do
  - The kernels is *event driven:* it responds to events
- Events can be:
  - *Hardware interrupts:* a hardware event like a mouse click, or network packet received
  - *Syscalls:* a user-space-application generated call to the kernel e.g., application asking kernel to printf() to the screen.
  - *Signals:* a software interrupt that announces an event to a process
    e.g., SIGINT = ctrl+c, SIGSEGV = segmentation (page) fault

User Mode vs. Kernel Mode

- Privilege mode of CPU execution
  - Kernel Mode runs the OS kernel; allows full privilege and full access to the hardware.
  - Often called "Ring 0"
  - User Mode runs applications; *cannot execute "privileged instructions",* e.g.
    - instructions that allow direct access to hardware
    - access to certain regions of memory (kernel memory)
- Which best explains why we need a user mode?

a) Isolation

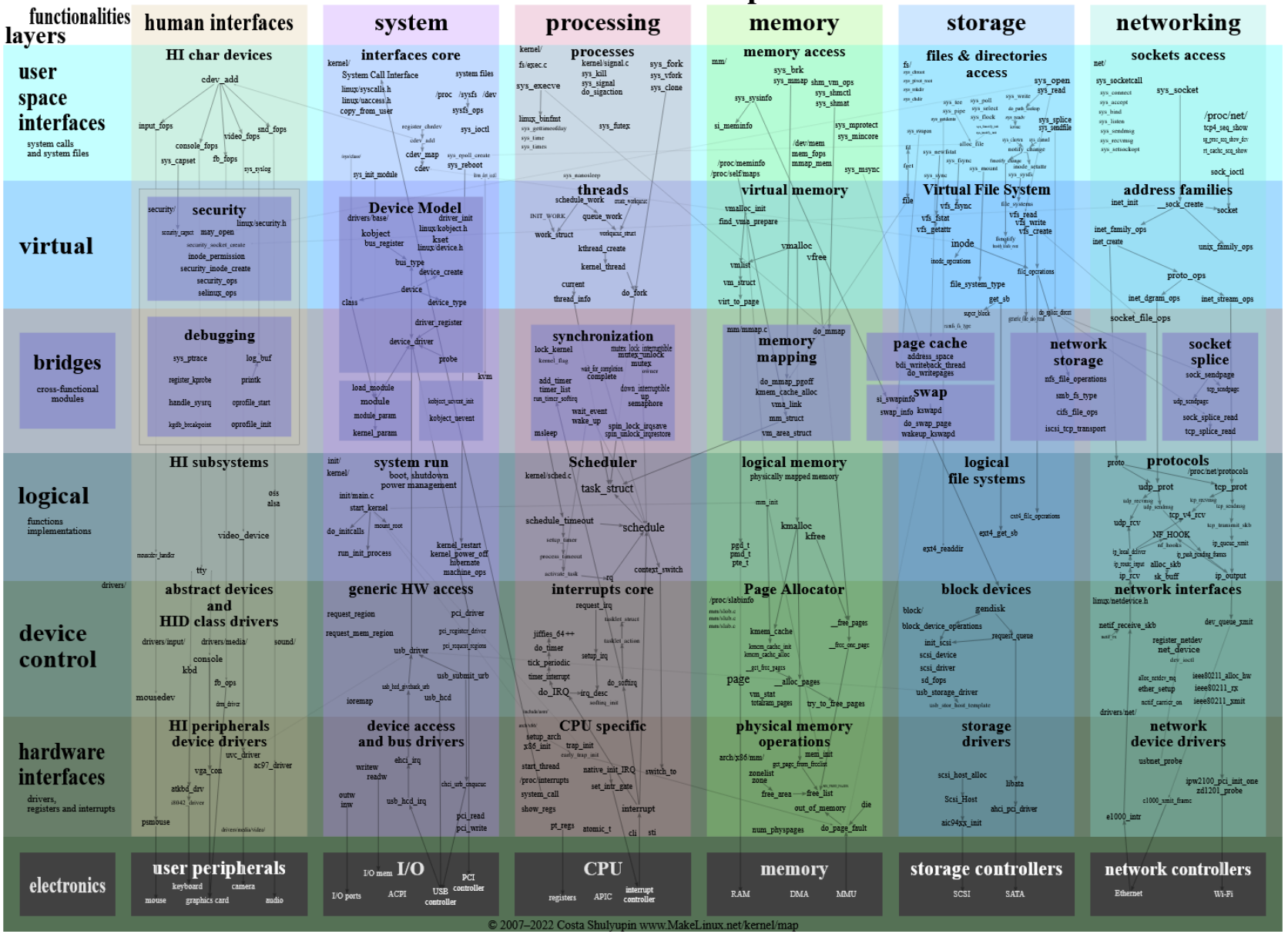
b) Efficiency

c) Null pointers

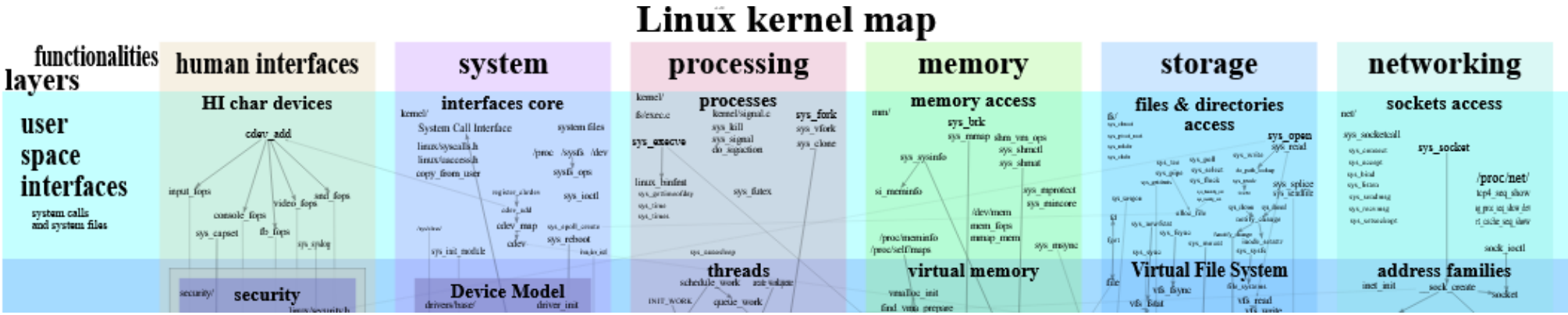
d) Abstraction

Root user (aside)

- User/Kernel *Mode* vs Root *User:*
  - The “mode” (privilege level of code) is different than the user-level
  - The **root** user is still a user, but with full admin privileges
    - Root can run programs and access files that normal users cannot
    - Root user often called a super user
  - Root user cannot (directly) access kernel memory or protected instructions


Linux kernel map
functionalities
layers
human interfaces
system
processing
memory
storage
networking
user space interfaces
system calls and system files
virtual
bridges
cross-functional modules
logical
functions implementations
device control
hardware interfaces
drivers, registers and interrupts
electronics
HI char devices
interfaces core
processes
memory access
files & directories access
sockets access
security
Device Model
threads
virtual memory
Virtual File System
address families
debugging
synchronization
memory mapping
page cache
network storage
socket splice
swap
HI subsystems
system run
Scheduler
logical memory
logical file systems
protocols
abstract devices and HID class drivers
generic HW access
interrupts core
Page Allocator
block devices
network interfaces
HI peripherals device drivers
device access and bus drivers
CPU specific
physical memory operations
storage drivers
network device drivers
user peripherals
I/O
CPU
memory
storage controllers
network controllers
keyboard
camera
mouse
graphics card
audio
I/O mem
I/O ports
ACPI
USB controller
PCI controller
registers
APIC
interrupt controller
RAM
DMA
MMU
SCSI
SATA
Ethernet
Wi-Fi
© 2007–2022 Costa Shulyupin www.MakeLinux.net/kernel/map

Important Terms in the Kernel



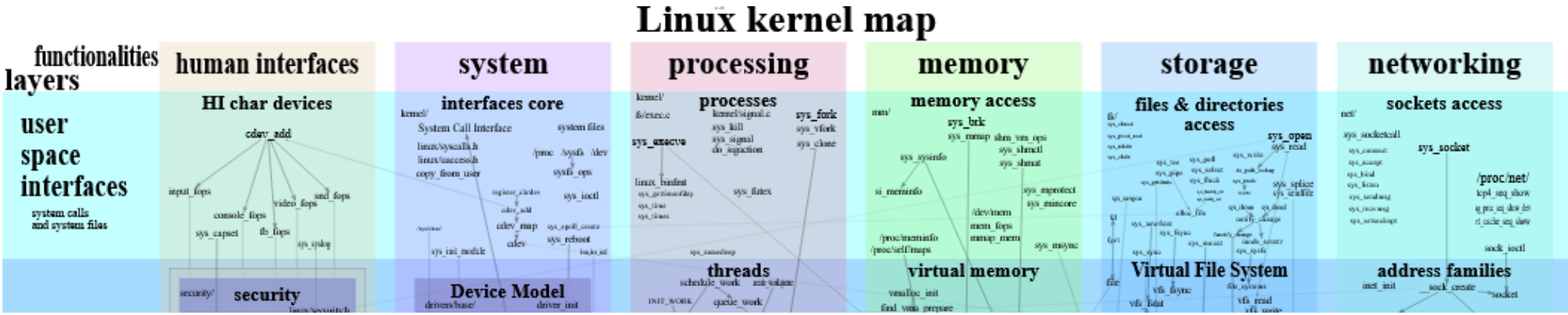
- System
  - **Device drivers:** every device needs a device driver to control it
    E.g., network card device driver talks to hardware to send/receive data to/from the physical network.
- Processing
  - **Processes, threads, synchronization,** and **scheduling**

Covered later

- Memory
  - **Virtual memory, physical memory,** and **paging**

Covered later

Important Terms in the Kernel (cont)



- Storage
  - **File systems** and **VFS** (Virtual File System)
- VFS is an interface – *data structures and operations that a file system should support,* e.g. read and write
  - Different filesystems, but also services that *pretend* to be normal files, so general tools can work seamlessly with them
    e.g., "`cat /proc/cpuinfo`"
- Networking
  - **Sockets, TCP, UDP,** and **IP**

Covered later

Audience Participation - Kernel

- Which of the following is true?

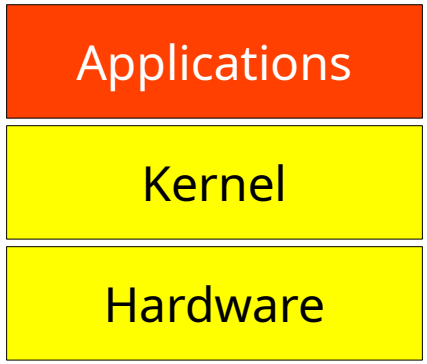
a) The root user runs programs in kernel mode.

b) Syscalls allow the kernel to execute user-level applications.

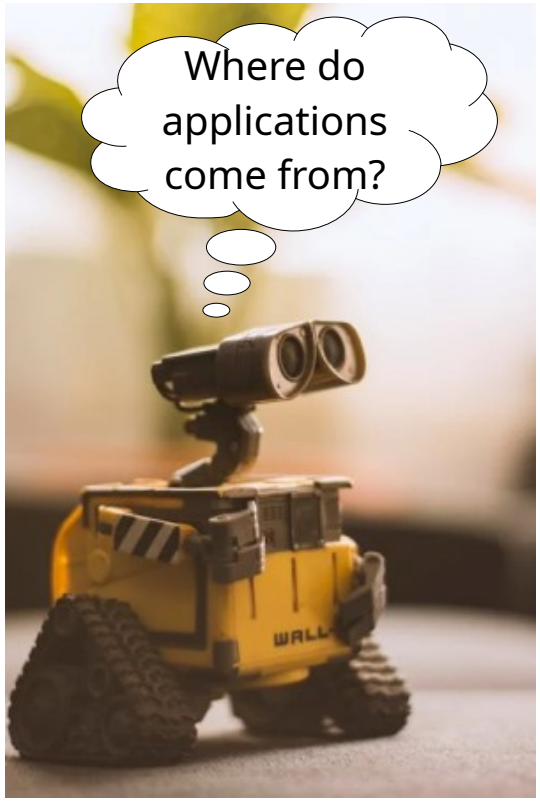
c) A hardware interrupt is generated when dereferencing a null pointer.

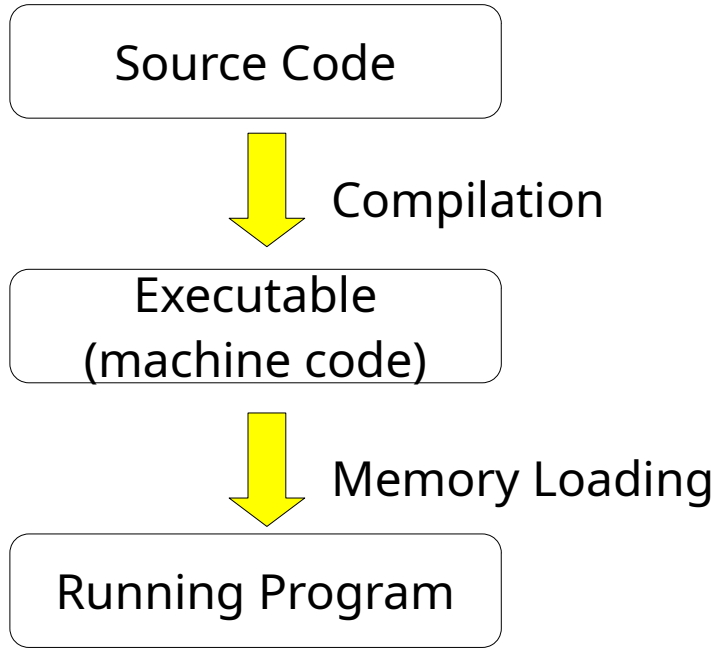
d) User mode prevents applications from executing privileged instructions.

Applications Layer



Lifetime of a Program (briefly)





Compilation vs. Interpretation (briefly)

- Two major ways to run a program:
  - **Compilation** (e.g., C, C++)
  - **Interpretation** (e.g., Python, Bash)
- Performance vs Portability Trade-off
  - Compiled code:
    - Source is translated into (usually optimized) machine code
    - Performs better: code is directly executed
    - Not portable: machine code targets specific ISA
      - E.g., can't run x68 executable on ARM machine
  - Interpretation may be slower, but same script can run anywhere there is an interpreter

Compilation vs. Interpretation (briefly)

- Beware: the devil is in the details
  - Someone has to port and compile the interpreter first, and if you have the source code, can't you just recompile a compiled program?
  - If your interpreted code just makes a few calls into the compiled runtime, it may perform just as well as any compiled program…
    - E.g. TensorFlow, SciPy
  - And what about JIT compilation!
- Don't think too hard, or you will end up in PL research

POSIX (briefly)

- **POSIX** (Portable Operating System Interface):
  - A standard for (user-level) software portability across different OSes
  - Includes programming interface (file I/O, C standard library, etc.) and shell utilities
  - We see it in C too:
    `#define _POSIX_C_SOURCE 200809L`

Image: https://www.linkedin.com/pulse/understanding-posix-standard-bridges-operating-systems-logzeta-1bl4f/

ABI

- An **A*P*I** (Application *Programming* Interface) is a collection of related functions, often in a library
  - Your code calls or accesses the functions of the API
- An **A*B*I** (Application *Binary* Interface) is a ***standard*** for *how code interoperates* in general
  - Tedious details of passing parameters, how function calls and system calls are made, how data is organized in memory
  - Informally, people also lump **syscalls** into the **OS ABI**
    - Syscalls are APIs provided directly by the kernel
- Compilers generate executables that follow the ABI for the OS
  - E.g., Windows ABI is different from Linux ABI
  - Cannot copy a Windows binary (“.exe”) to a Linux machine and run it (and vice versa)
    - …unless you have an ABI translator, like WINE or WSL1

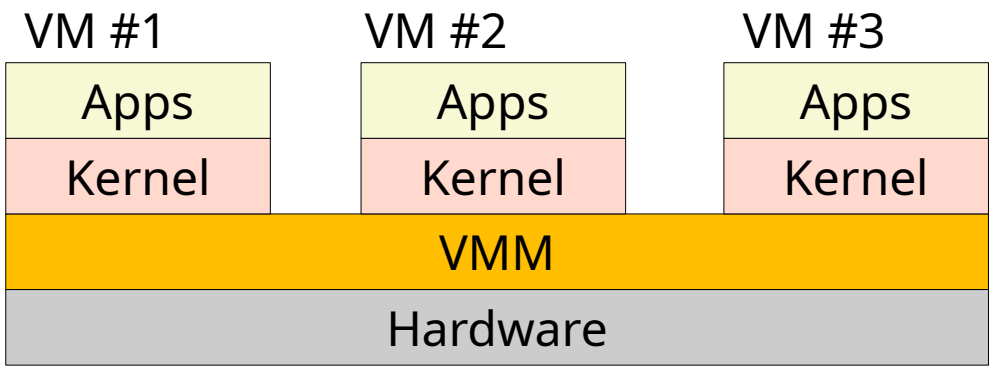
Virtualization

Virtualization of Traditional OS Stack

- Virtualization allows part(s) of our OS stack to be swapped out
  - Lets us be much more flexible!
  - Software can control the environment:
    ”Spin up 3 virtual machines to host new databases”
- **Hypervisor:** software that provides virtualization
  - Also called the Virtual Machine Monitor (VMM)
  - Hypervisor can run at different levels of our OS stack, giving different levels of flexibility

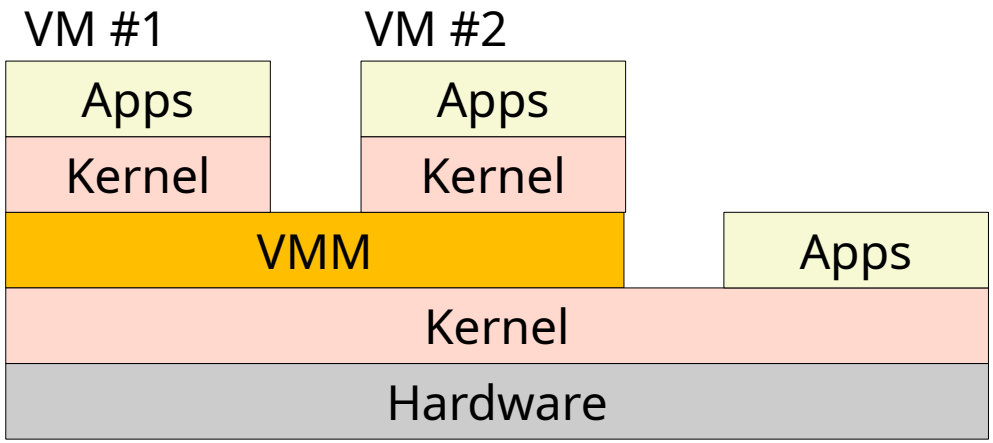
On Hardware

- VMM directly atop hardware
  - VMM emulates hardware for each VM (Virtual Machine)
  - Often used in a data center environment
  - Not new: architecture used by IBM since the 1970s



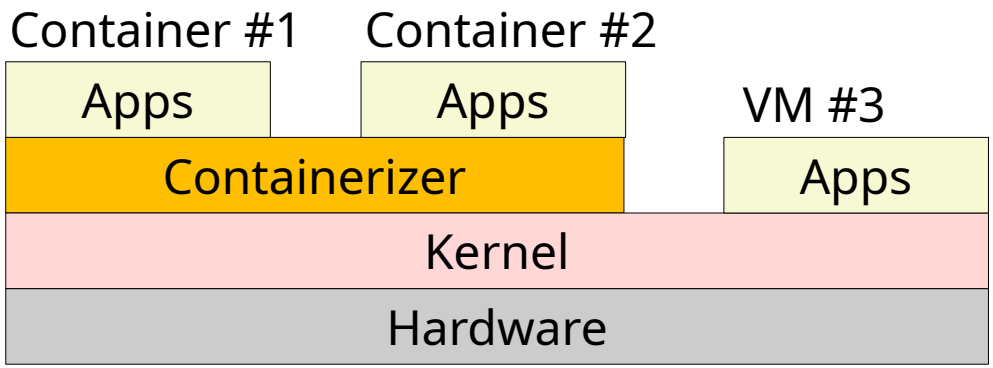
On Kernel

- VMM atop the Kernel
  - VMM is an application running atop a kernel, along with other applications (often using special kernel services e.g. KVM)
  - The VMM creates/runs/manages VMs
  - This is often used in a desktop environment, e.g., VMWare Workstation, VirtualBox, QEMU



Containerization

- Containerization
  - Containerization creates a **container,** not a virtual machine.
  - Container includes *an isolated set of applications and data.*
  - Uses the same OS kernel as rest of the system
  - Uses Linux features for isolation: *process isolation* (namespaces), *resource control/isolation* (cgroups), etc.
  - This is the most popular form of virtualization these days, e.g., **Docker,** Podman.



Audience Participation - Virtualization

- Which of the following is a major benefit of virtualization?

a) Allows user level applications to call the kernel.

b) Allows parts of the OS stack to be swapped out under software control.

c) Allows the kernel to control different pieces of hardware when they are connected at runtime.

d) Allows application to run without using an OS kernel.

Theoretical View?

- We are taking a very practical view
- “Virtualization” and “virtual machine” as abstract concepts are very broad
- E.g., the “Java Virtual Machine” is an example of “process virtualization”
- The ABI and syscall interface can be understood together as a kind of VM
- For the purposes of this course, *we are ignoring this broader theoretical view,* even though it is valid

Summary

- OS Stack is the layers of service
  - Hardware, Kernel, Application
- Memory hierarchy
  - allows programs to access large memories quickly
- Pointers hold addresses,
  - 32 vs 64 bits limit how much memory we can access
- Kernel mode gives OS kernel access to all resources
  - User mode limits what an application can do
- Applications use the OS’s ABI to use services
- Virtualization allows parts of the OS stack to be swapped out under software control