CSCI 3753 Operating Systems
Spring 2019

Christopher Godley

PhD Student

Department of Computer Science

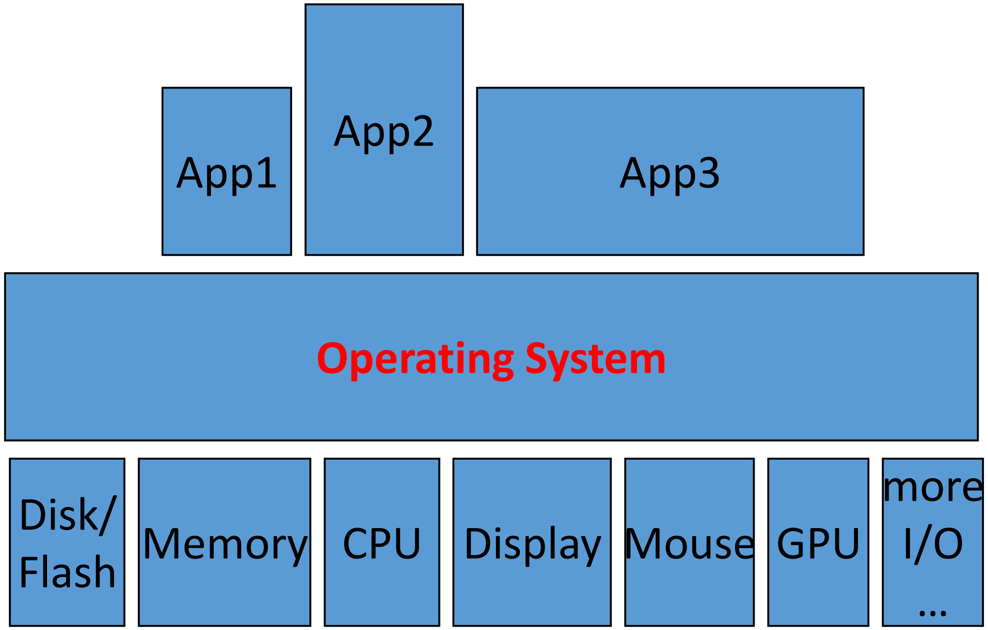
University of Colorado Boulder

What does Operating System provide?

Definition: An operating system is a layer of software between *many* applications and *diverse* hardware that

1. Provides a ***hardware abstraction*** so an application doesn't have to know the details about the hardware.

E.g. An application saving a file to disk doesn't have to know how the disk operates

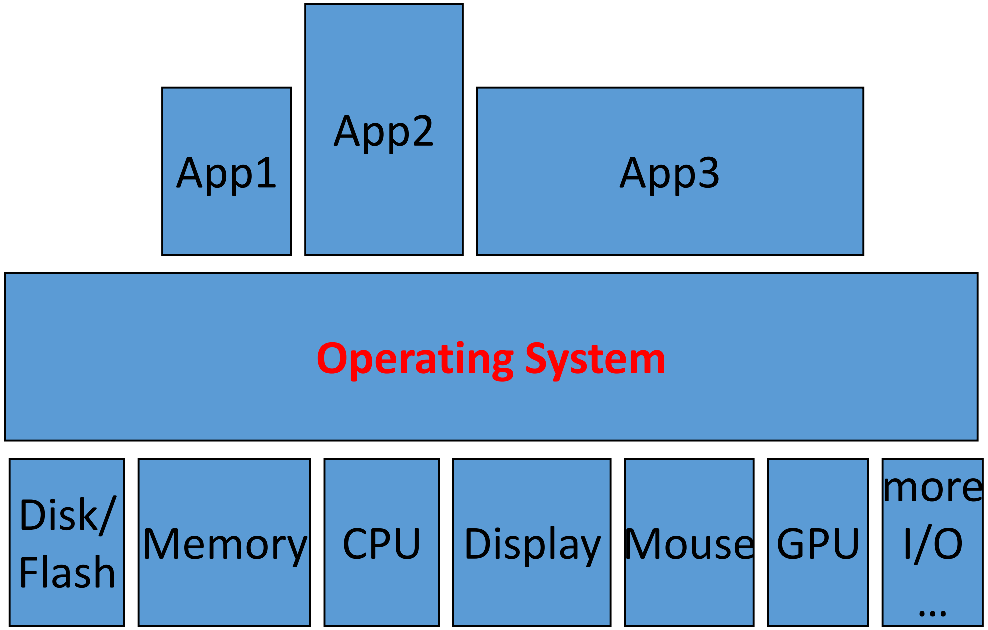


What does Operating System provide?

Definition: An operating system is a layer of software between *many* applications and *diverse* hardware that

2. **Arbitrates access** to resources among multiple applications:
   + Sharing of resources.

E.g. Sharing a Printer among applications

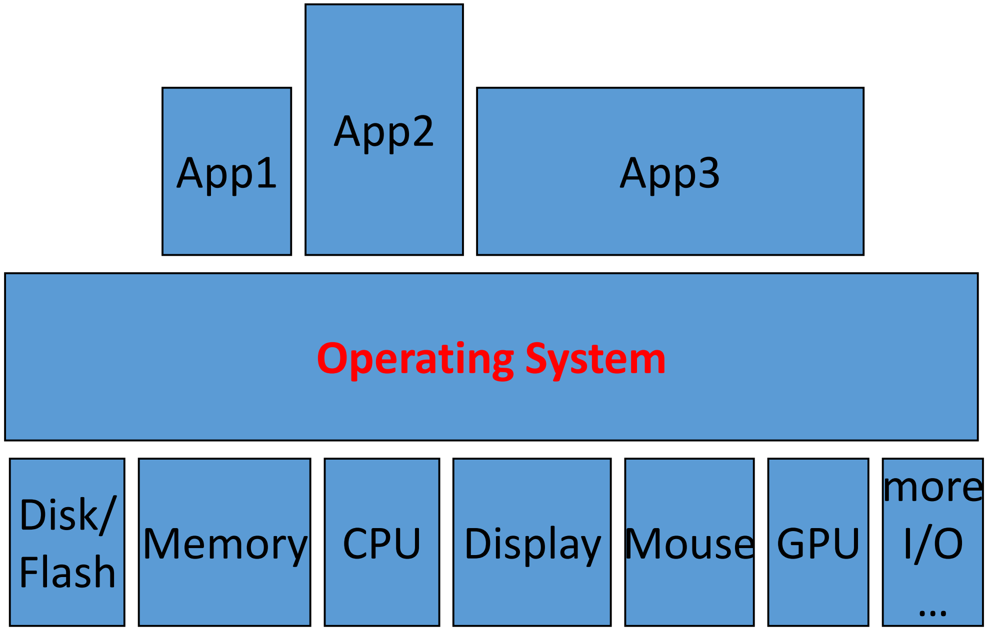


What does Operating System provide?

Definition: An operating system is a layer of software between *many* applications and *diverse* hardware that

3 . **Provide protections**:

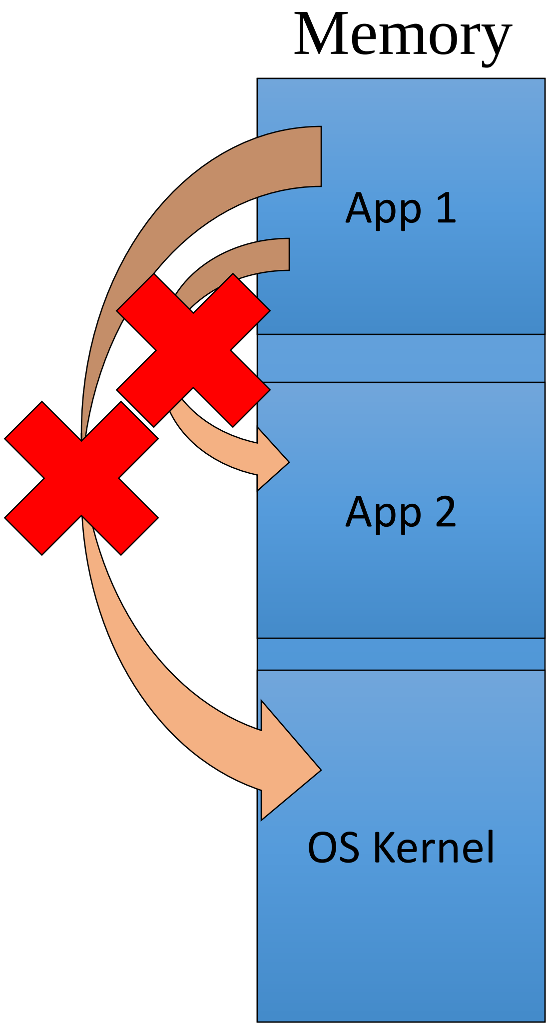
- *Isolation* protects **app's from each other**
- *Isolation* also to protect the **OS from applications**
- *Isolation* to **limit resource consumption by any one app**



Protection in Operating Systems

1. Prevent applications from writing into privileged memory
   – e.g. of another app or OS kernel

2. Prevent applications from invoking privileged functions
   – e.g. OS kernel functions



Privileged Instruction Examples

- Memory address mapping
- Flush or invalidate data cache
- Invalidate TLB (Translation Lookaside Buffer) entries
- Load and read system registers
- Change processor modes from K to U
- Change the voltage and frequency of processors
- Halt/reset processor
- Perform I/O operations

What is an unit of work for an OS?

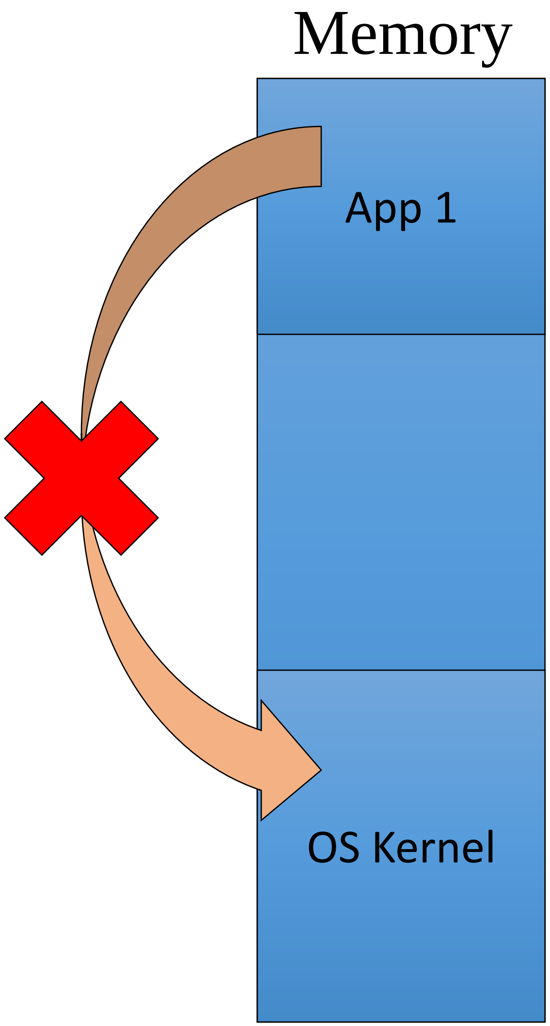
- Application
- Task
- Job
- Process

What does a TASK consist of?

- Code – placed into memory
- Data – stored in memory
- OS data for task – task descriptors

How can we access the OS functionality?

- **Problem:** If a task is protected from getting into the OS code and data, OS functionality are restricted from these tasks
- How does CPU know if a certain instruction should be allowed?
- How does OS grant a task access to certain OS data structures but not the other?
- How to switch from running the task's code to running OS's code
- Need to use a hardware assistant called **mode bit**

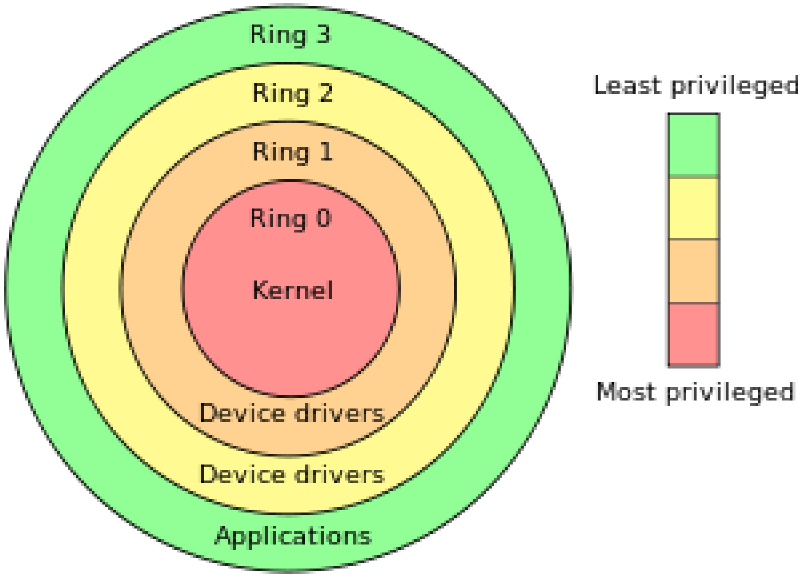


Kernel Mode vs User Mode

- Processors include a hardware *mode* bit that identifies whether the system is in *user* mode or *supervisor/kernel* mode
  - Requires extra support from the CPU hardware for this OS feature
- Supervisor or kernel mode (mode bit = 0)
  - Can execute all machine instructions, including privileged instructions
  - Can reference all memory locations
  - Kernel executes in this mode
- User mode (mode bit = 1)
  - Can only execute a subset of non-privileged instructions
  - Can only reference a subset of memory locations
  - All applications run in user mode

Multiple Rings/Modes of Privilege

- Intel x86 CPUs support four modes or rings of privilege
- Common configuration:
  - OS like Linux or Windows runs in ring 0 (highest privilege), Apps run in ring 3, and rings 1-2 are unused

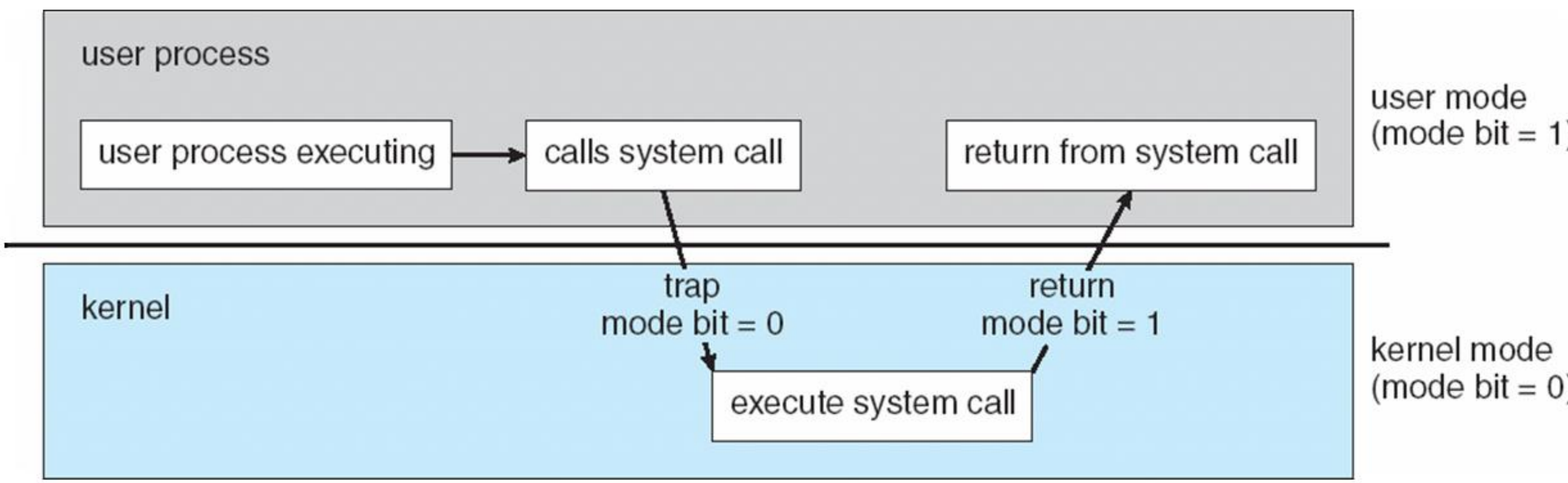


- **Virtual machines (one possible configuration)**
  - **VM's hypervisor runs in ring 0, guest OS runs in ring 1 or 2, Apps run in ring 3**

System Call

System Calls: How Apps and the OS Communicate

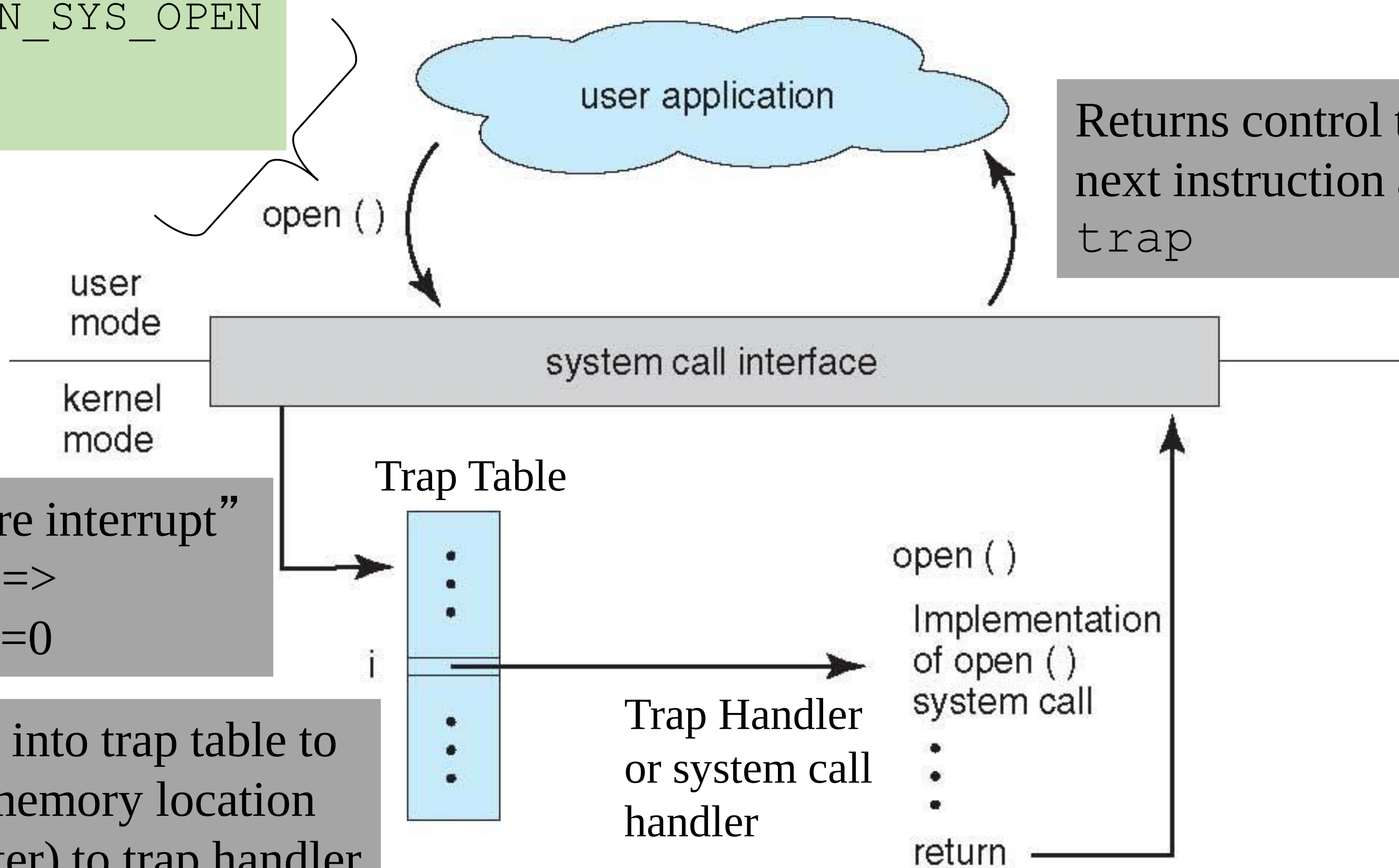
- The `trap` instruction is used to switch from user to supervisor mode, thereby entering the OS
  - `trap` sets the mode bit to 0
  - On x86, use INT assembly instruction (more recently SYSCALL/SYSENTER)
  - mode bit set back to 1 on return
- Any instruction that invokes `trap` is called *a system call*
  - There are many different classes of system calls



API – System Call – OS Relationship

```
open() {
…
trap N_SYS_OPEN
…
}
```

Returns control to the next instruction after `trap`



"software interrupt" = a trap => modebit=0

Index into trap table to find memory location (pointer) to trap handler

Trap Table

- Trap handling: The process of indexing into the trap table to jump to the trap handler routine is also called *dispatching*
- The trap table is also called a *jump table* or a *branch table*
- "A trap is a *software interrupt*"
- Trap handler (or system call handler) performs the specific processing desired by the system call/trap

Classes of System Calls Invoked by `trap`

system call interface

Process control	File Management	Device Management	Information Management	Comm-unications

Process control

- end, abort
- load, execute
- fork, create, terminate
- get attributes, set
- wait for time
- wait event, signal event
- allocate memory, free

File Management

- create, delete
- open, close
- read, write, reposition
- get attributes, set

Device Management

- request device, release
- read, write, reposition
- get attributes, set
- logically attach or detach devices

Note Similarity

Information Management

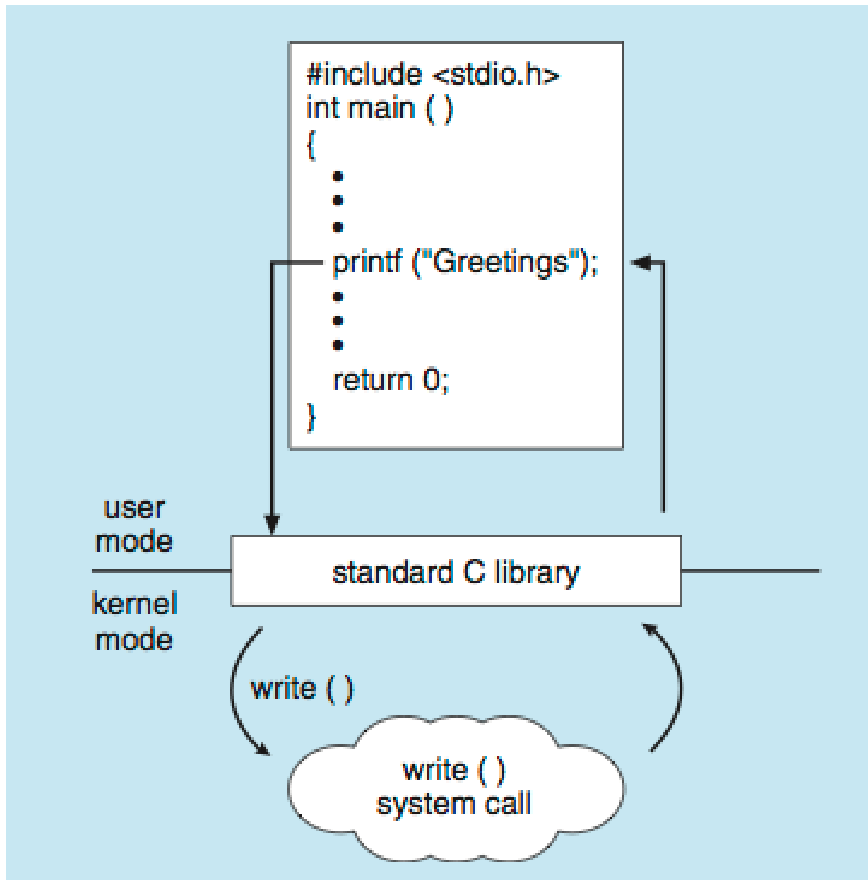
- get time/date, set
- get system data, set
- get process, file, or device attributes, set

Comm-unications

- create connection, delete
- send messages, receive
- transfer status info
- attach remote devices, detach

Standard C Library Example

- C program invoking printf() library call, which calls write() system call



e.g. INT or SYSCALL

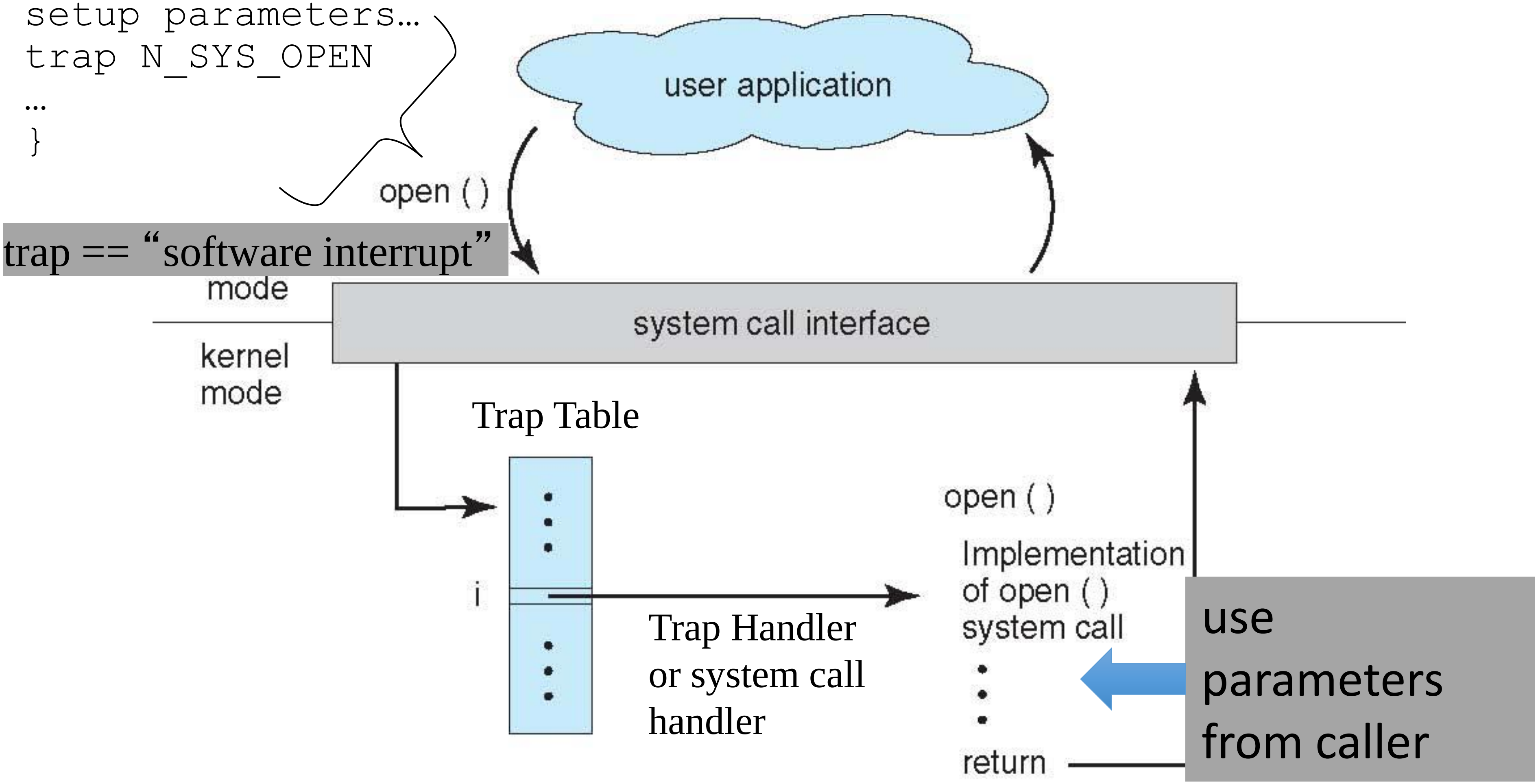
Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Manipulation	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

```
int open(const char *pathname, int flags);
```

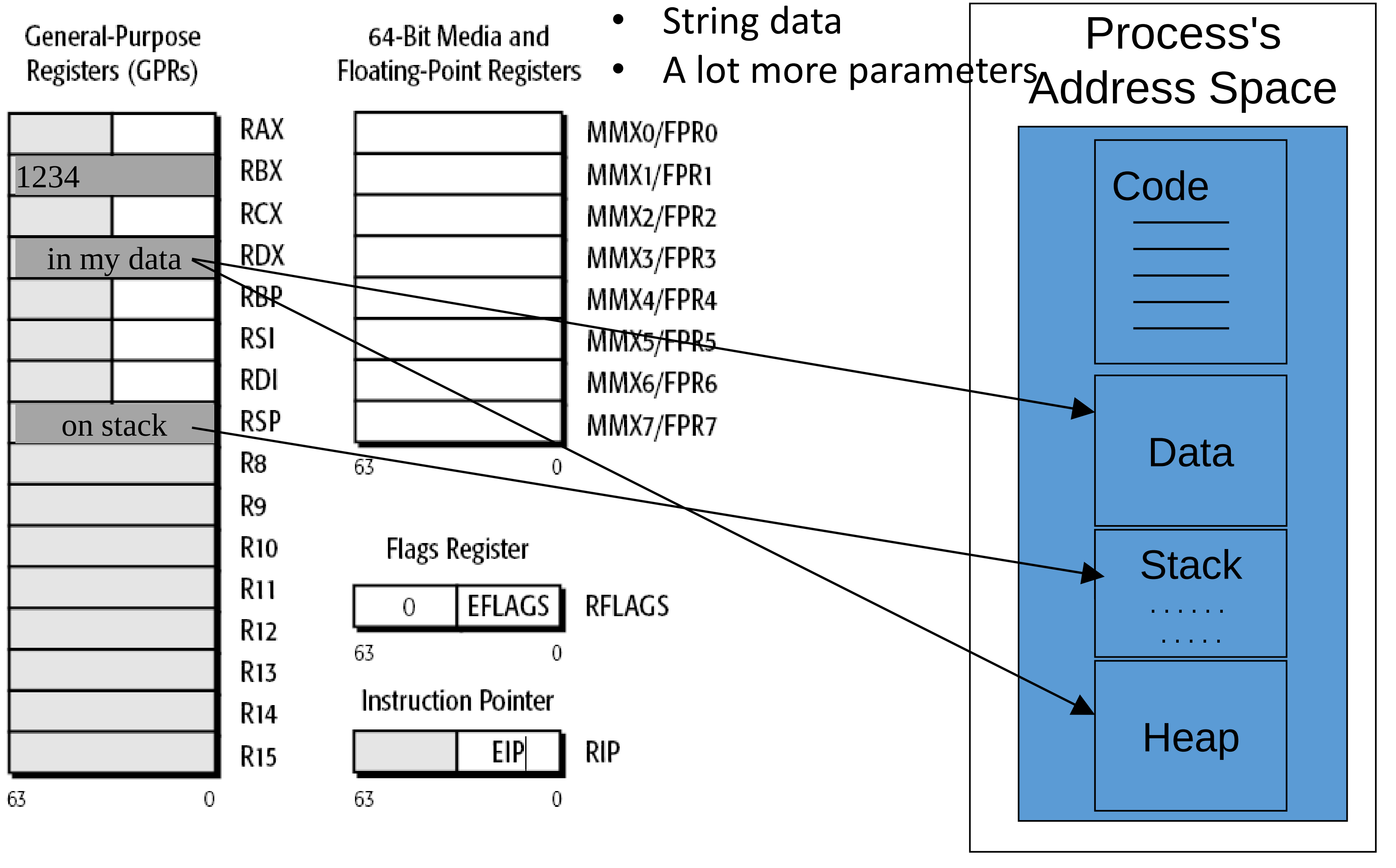
Passing Parameters to System Call

```
open() {
…
setup parameters…
trap N_SYS_OPEN
…
}
```



Passing Parameters to System Calls

- Integer data
- String data
- A lot more parameters



System Call Parameter Passing

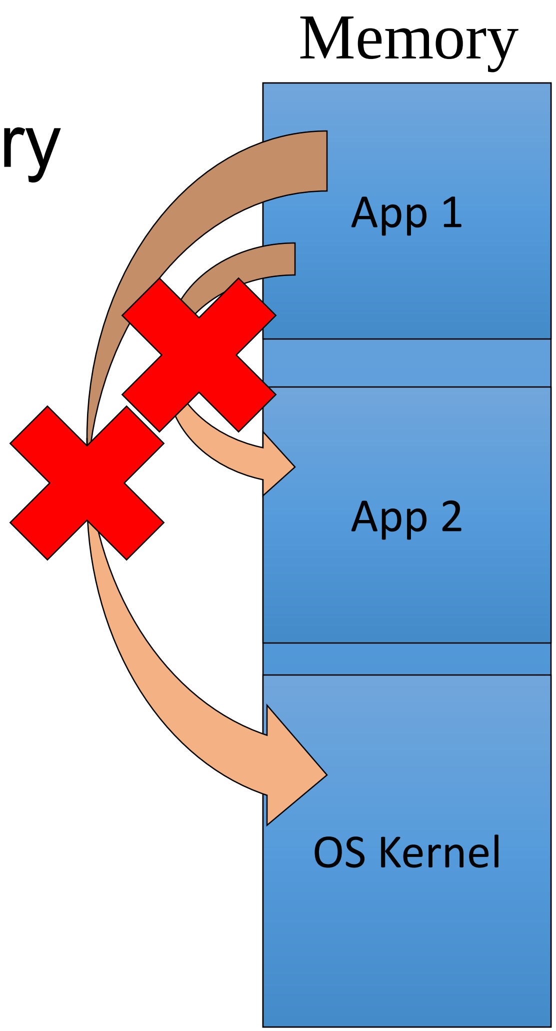
- Three general methods used to pass parameters to the OS
  1. Register: Simplest, pass the parameters in *registers*
     - In some cases, may be more parameters than registers
  2. *Pointer*: Parameters stored in a block*,* or table, in memory, and address of block passed as a parameter in a register
     - This approach taken by Linux and Solaris
  3. *Stack:* Parameters placed, or **pushed***,* onto the **stack** by the program and **popped** off the *stack* by the operating system
- Block and stack methods do not limit the number or length of parameters being passed

Protection of Applications

1. The OS can't just access any memory

   E.g. App1 tell the OS to access App2's data

2. Need to explicitly ask itself for permission
3. When in the kernel, extra caution is needed to access to data



API for User Space memory access from Kernel Space

Function	Description
access_ok	Checks the validity of the user space memory pointer
get_user	Gets a simple variable from user space
put_user	Puts a simple variable to user space
clear_user	Clears, or zeros, a block in user space
copy_to_user	Copies a block of data from the kernel to user space
copy_from_user	Copies a block of data from user space to the kernel
strnlen_user	Gets the size of a string buffer in user space
strncpy_from_user	Copies a string from user space into the kernel

Loading an OS

How do we get a computer started?

- We have hardware and we give it power, what happens?
  - CPU only knows how to perform its basic operations, load, move, add, …
  - Computer only does what it is told
  - How do we get it to run the operating system?
  - How does is know where or how to load the first program?

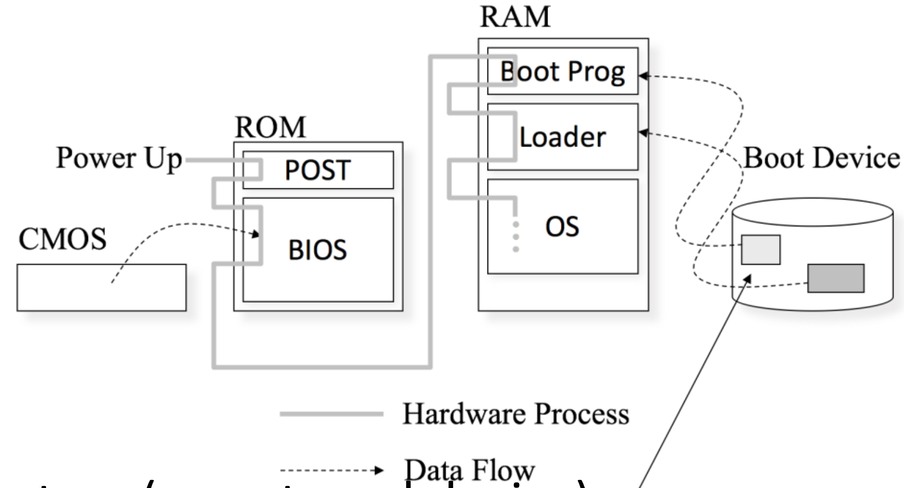
System Boot

- *Booting* or *"Bootstrapping"* a system is the procedure for loading and starting the operating system

- *Bootstrap* program or *Bootstrap* loader locates the OS kernel, loads it into memory, and starts its execution

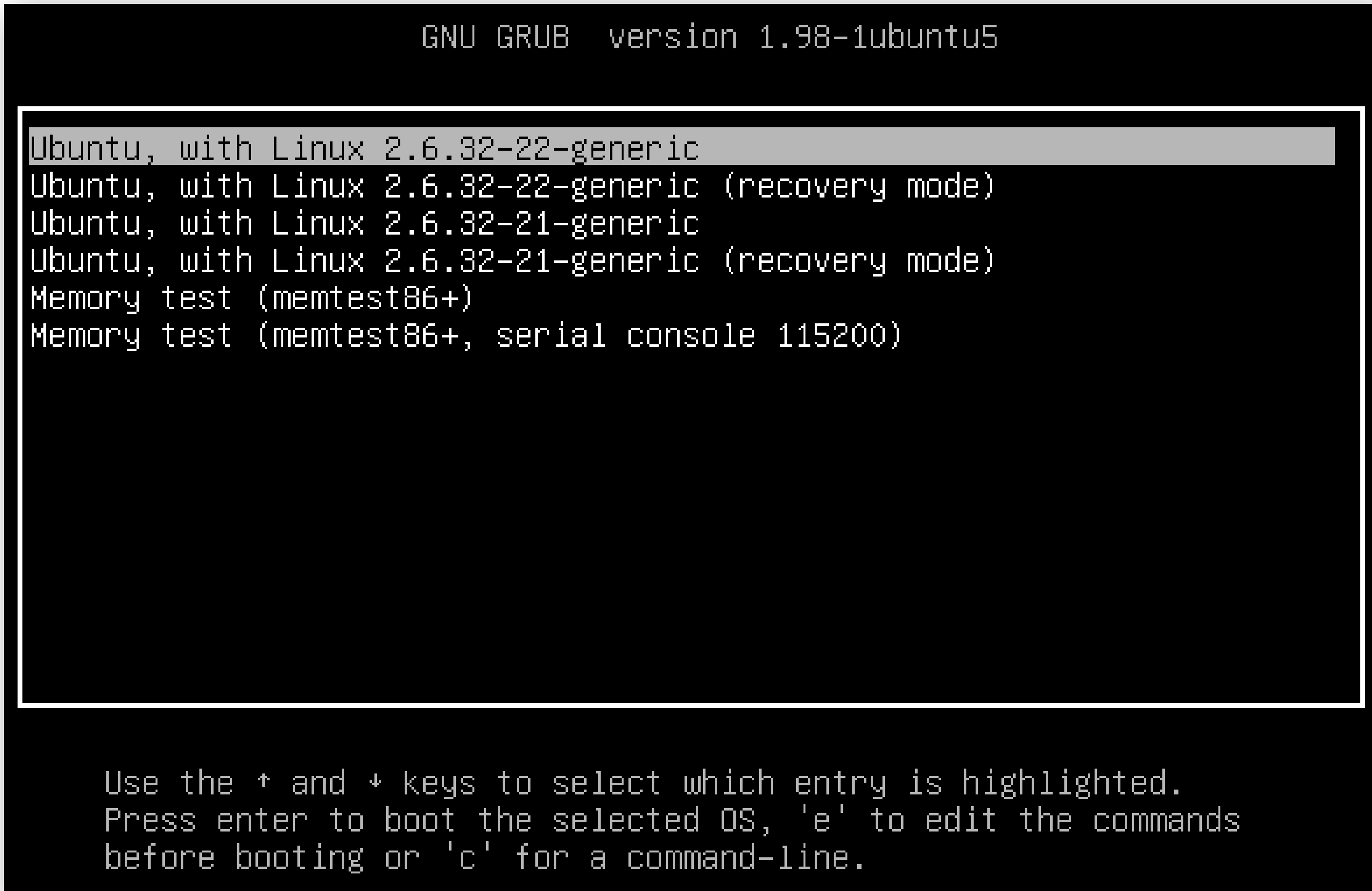
- Booting is usually a two step process
  - Load and execute a simple bootstrap loader
  - which fetches a more complex kernel from disk and switches execution to the kernel

System Boot



- **Initial boot loader** must be within the system (no external device)
  - ROM or EPROM
- Steps to Booting
  - run diagnostics to make sure the hardware is working (CPU, Memory)
  - find a device to boot from
    - (disk, flash, dvd, network, …)
  - load the primitive boot loader and transfer control
  - Primitive boot loader then loads the secondary stage bootloader
    - Examples of this secondary bootloader include LILO (Linux Loader), and GRUB (Grand Unified Bootloader)
    - Can select among multiple OS’s (on different partitions) – i.e. dual booting
    - Once OS is selected, the bootloader goes to that OS’s partition, finds the boot sector, and starts loading the OS’s kernel

GRUB Boot Loader



What is a Virtual Machine?

- A simulated computer running within a real computer
- The virtual computer runs an operating system that can be different than the host operating
- All the requests to access real hardware are routed to the appropriate host hardware, then virtual operating system or applications don’t know they are virtual
- Similar to a person embedded in the Matrix (virtual people)

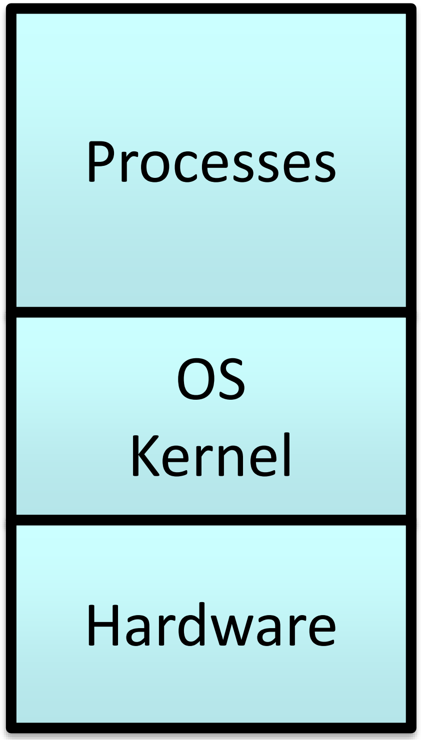
Virtual X concept.

- A process already is given the illusion that it has its
  - Own memory, via virtual memory
  - Own CPU, via time slicing
- Virtual machine extends this idea to give a process the illusion that it also has its own hardware
  - Moreover, extend the concept from a process to an entire OS being given the illusion that it has its own memory, CPU, and I/O devices

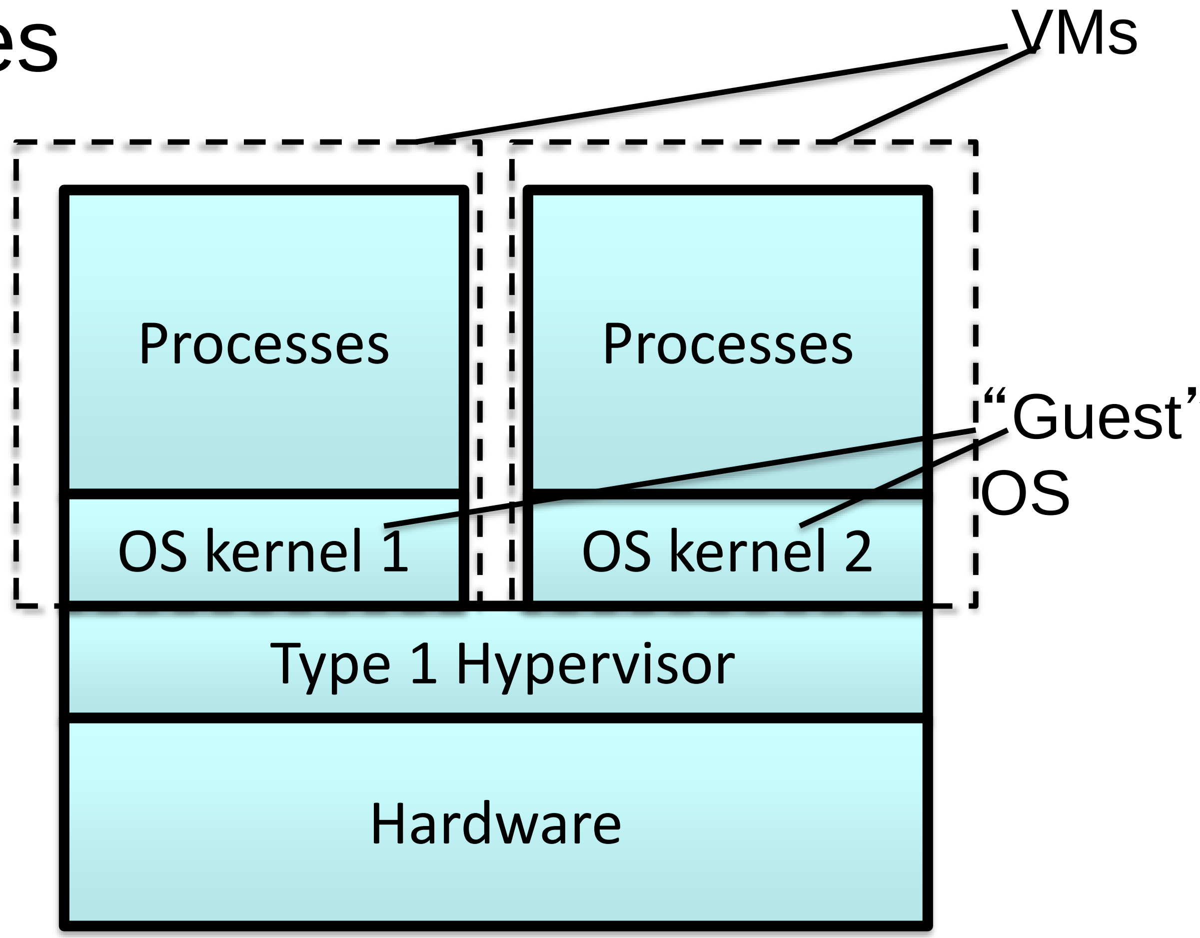
Virtual Machines

- Benefits include:
  - Can run multiple OS’s simultaneously on the same host
  - Fault isolation if an OS fails – doesn't crash another VM. This is also useful for debugging a new OS.
  - Easier to deploy applications – can deploy an app within a VM instance that is customized for the app, rather than directly deploying the app itself and worrying about compatibility with the target OS – useful for cloud server deployments

Virtual Machines



Traditional OS



A Type 1 *Hypervisor* provides a virtualization layer for guest OSs and resides just above the hardware.

Virtual Machines

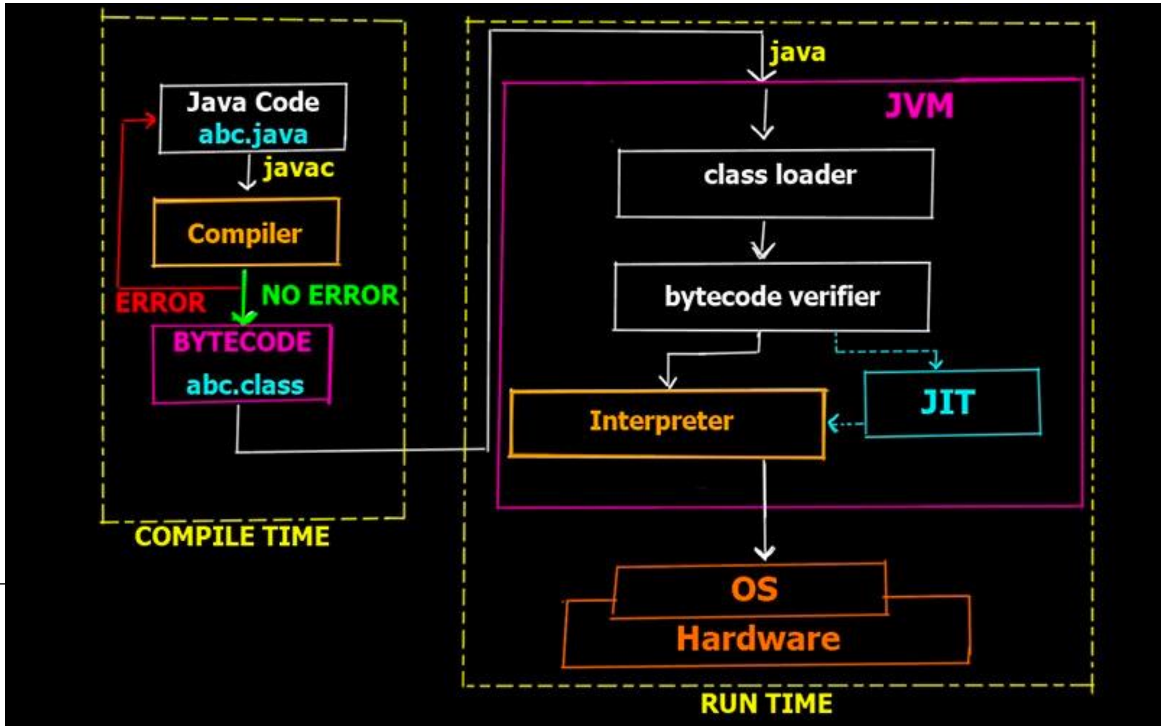
- How it basically works:
  - Goal: want to create a virtual machine that executes at close to native speeds on a CPU, so emulation and interpreting instruction by instruction are not good options – too much software overhead
  - Solution: have the guest OS execute normally, directly on the CPU, except that it is not in kernel mode.
    Therefore, any special privileged instructions invoked by the guest OS will be trapped to the hypervisor, which is in kernel mode.
  - The hypervisor then emulates only these privileged instructions and when done passes control back to the guest OS, also known as a "VM entry"
  - This way, most ordinary (non-privileged) instructions operate at full speed, and only privileged instructions incur the overhead of a trap, also known as a "VM exit", to the hypervisor/VMM.
  - This approach to VMs is called *trap-and-emulate*

Virtual Machines

- Cloud Computing
  - Very easy to provision and deploy VM instances on the cloud
  - E.g. Amazon's Elastic Compute Cloud (EC2) uses Xen virtualization
    - There are different types of VMs or instances that can be deployed:
      - Standard, High-Memory, High-CPU
    - Users can create and reboot their own VMs
    - To store data persistently, need to supplement EC2 with an additional cloud service, e.g. Amazon's Simple Storage Service (S3)

Java Virtual Machines

- Process VMs, e.g. Java VMs
  - Differ from System VMs in that the goal is NOT to try to run multiple OSs on the same host, but to **provide portable code execution** of a single application across different hosts
- Java applications are compiled into Java byte code that can be run on any Java VM
  - Java VM acts as an *interpreter* of byte code, translating each byte code instruction into a local action on the host OS



Java Virtual Machines

- Just in time compilation can be used to speed up execution of Java code
  - Java byte code is compiled at run time into native machine code that is executed directly on the hardware, rather than being interpreted instruction by instruction
- Note Java VMs virtualize an abstract machine, not actual hardware, unlike system VMs
  - i.e. the target machine that Java byte code is being compiled for is a software specification

Questions?