Recap: Read-Copy-Update (RCU)

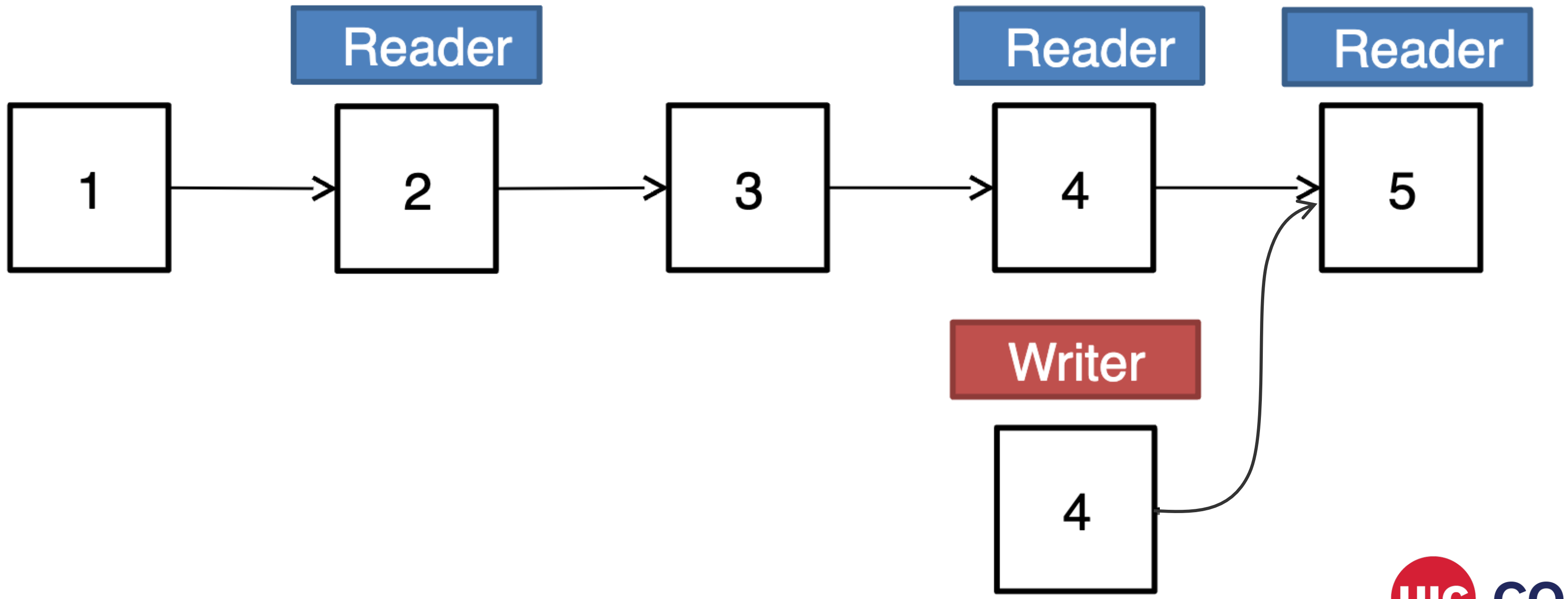
Only require locks for writes

- Lock-free reads + Single pointer update + Delayed free
- RCU-version of the linked list, hash table, …

```
static inline void INIT_LIST_HEAD_RCU(struct list_head *list)
void list_add_rcu(struct list_head *new, struct list_head *head);
void list_del_rcu(struct list_head *entry);
void list_replace_rcu(struct list_head *old, struct list_head *new);
#define list_for_each_entry_rcu(pos, head, member) ..
```

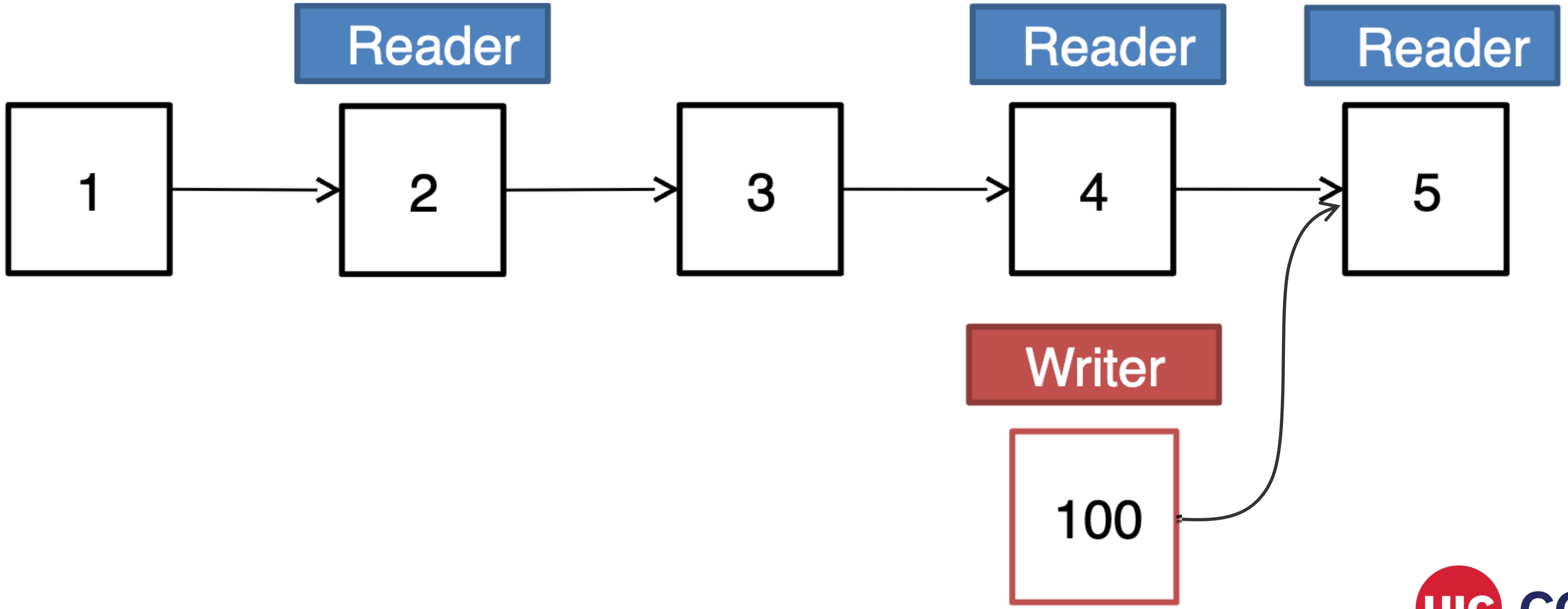
Recap: RCU-based linked list

A writer copies an element first



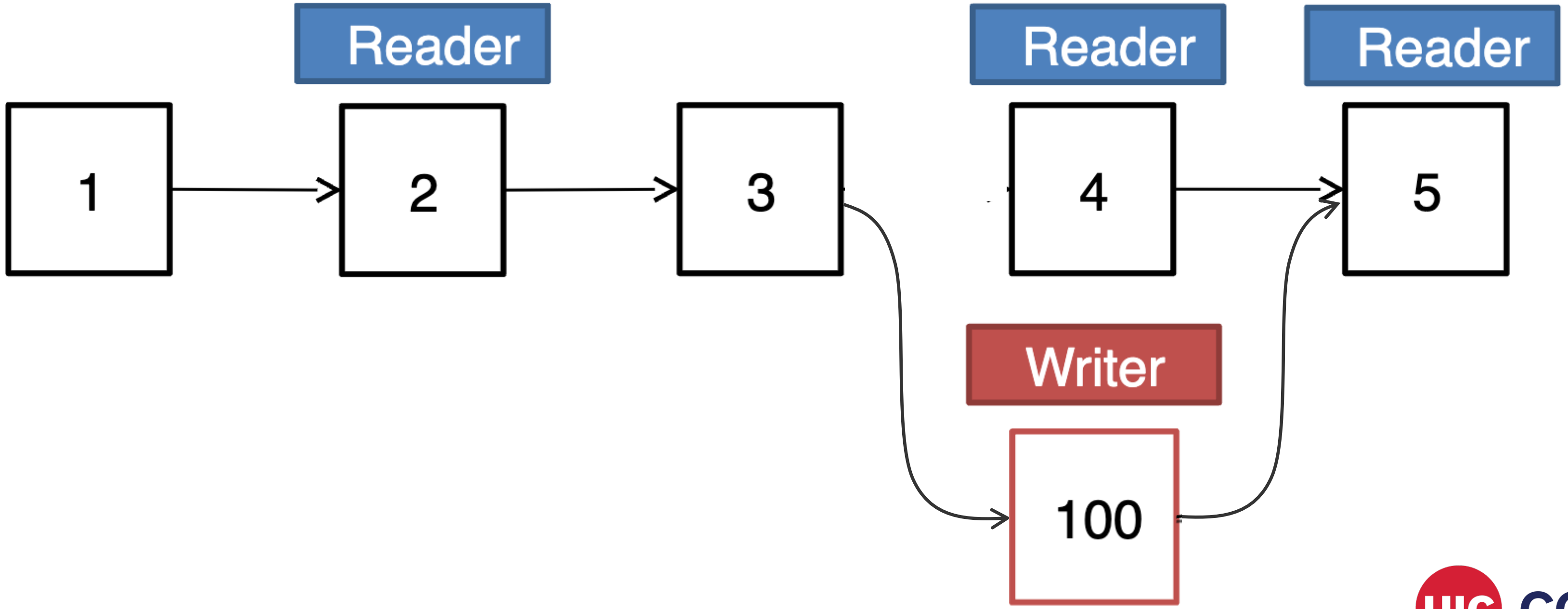
Recap: RCU-based linked list

Then it updates the element



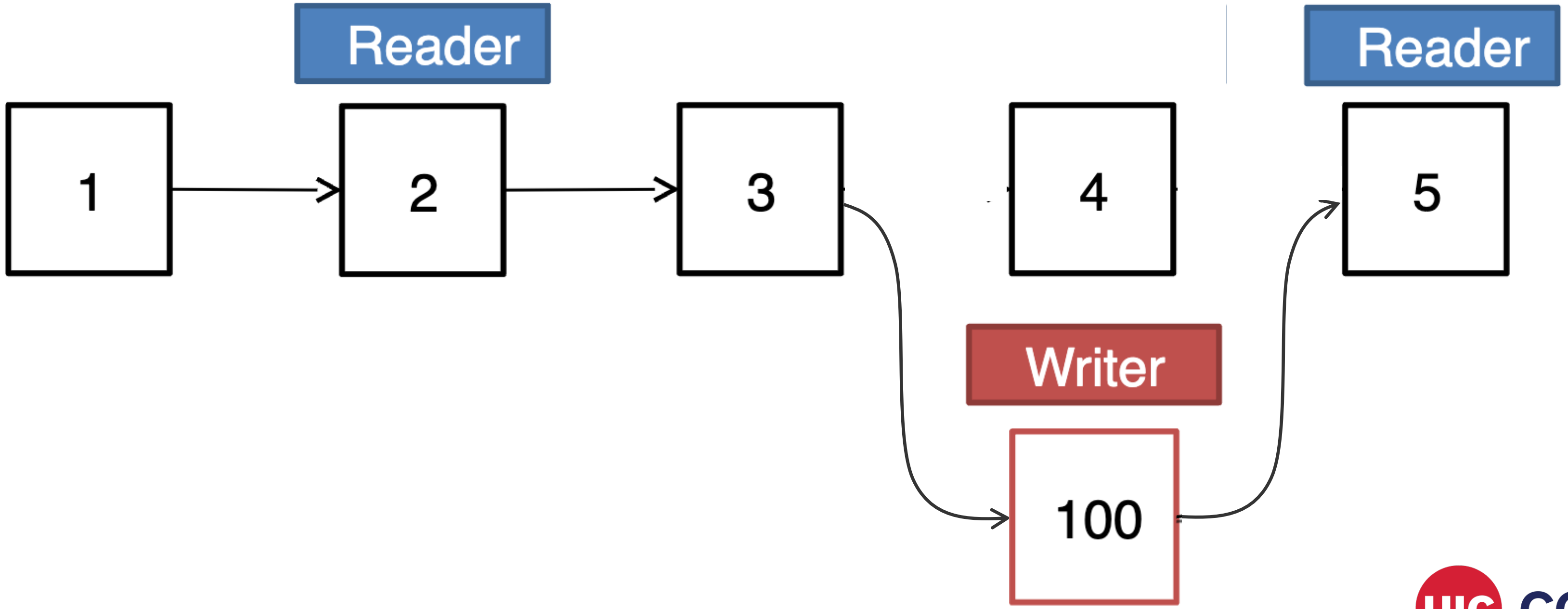
Recap: RCU-based linked list

And then it makes its change public by updating the next pointer of its previous. → New readers will traverse 100 instead of 4



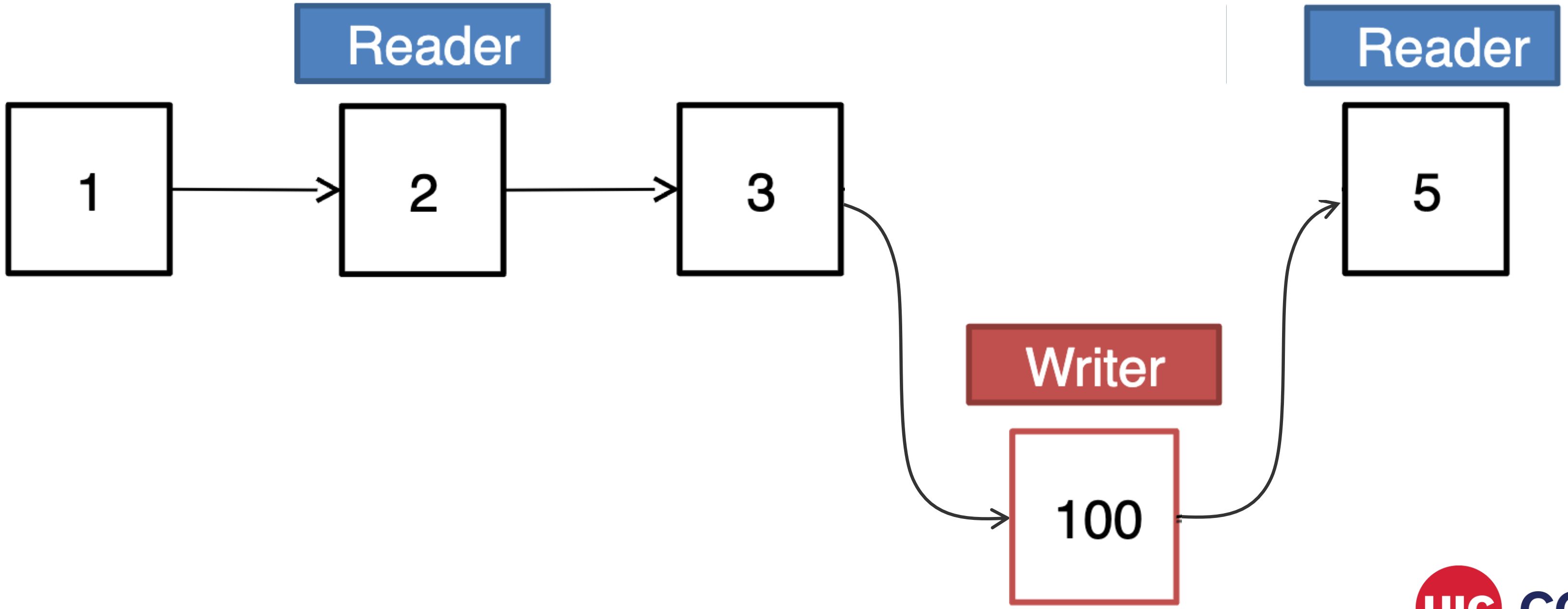
Recap: RCU-based linked list

Do not free the old node, 4, until no reader accesses it.



Recap: RCU-based linked list

When it is guaranteed that there is no reader accessing the old node, free the old node.



Recap: `jiffies`

A global variable holds the number of **timer ticks** since the system booted (`unsigned long`)

Conversion between `jiffies` and seconds

- `jiffies` = seconds * `HZ`
- seconds = `jiffies` / `HZ`

```
unsigned long time_stamp = jiffies;         /* Now */
unsigned long next_tick = jiffies + 1;      /* One tick from now */
unsigned long later = jiffies + 5*HZ;       /* 5 seconds from now */
unsigned long fraction = jiffies + HZ/10;   /* 100 ms from now */
```

Recap: Hardware clocks and timers

Real-Time Clock (RTC)

- Stores the wall-clock time (still incremented when the computer is powered off)
- Backed-up by a small battery on the motherboard

System timer

- Provide a mechanism for driving an interrupt at a periodic rate regardless of architecture
- System timers in x86
    - Local APIC timer: primary timer today
    - Programmable interrupt timer (PIT): was a primary timer until 2.6.17

Recap: Hardware clocks and timers

Processor's time stamp counter (TSC)

- `rdtsc`, `rdtscp`
- most accurate (CPU clock resolution)
- invariant to clock frequency (x86 architecture)
- seconds = clocks / maximum CPU clock Hz

Timer

Timers == dynamic timers == kernel timers

- Used to delay the execution of some piece of code for a given amount of time

```
/* include/linux/timer.h */
struct timer_list {
    struct hlist_node entry;   /* linked list of timers */
    unsigned long     expires; /* expiration time in jiffies */
    void              (*function)(struct timer_list *); /* handler */
    u32               flags;   /*
        TIMER_IRQSAFE: executed with interrupts disabled
        TIMER_DEFERRABLE: does not wake up an idle CPU */
    /* ... */
```

Use timers

```
timer_setup(timer, callback, flags);

/* mod_timer(timer, expires) is equivalent to:
 * del_timer(timer); timer->expires = expires; add_timer(timer); */
int mod_timer(struct timer_list *timer, unsigned long expires);

/* Get the parent struct, similar to container_of() */
from_timer(var, callback_timer, timer_fieldname);
```

Use timers

```
del_timer(struct timer_list *)
```

- Deactivate a timer prior
- Returns 0 if the timer is already inactive, and 1 if the timer was active
- Potential race condition on SMP when the handler is currently running on another core

Use timers

`del_timer_sync(struct timer_list *)`

- Waits for a potential currently running handler to finishes before removing the timer
- Can be called from interrupt context only if the timer is irqsafe (declared with TIMER_IRQSAFE)
- Interrupt handler interrupting the timer handler and calling `del_timer_sync()` → deadlock

Timer race conditions

Timers run in the `softirq` context → Several potential race conditions exist

- Protect data shared by the handler and other entities
- Use `del_timer_sync()` rather than `del_timer()`
- Do not directly modify the expire field; use `mod_timer()`

```
/* THIS CODE IS BUGGY! DO NOT USE! */
del_timer(&my_timer);
my_timer->expires = jiffies + new_delay;
add_timer(&my_timer);
```

Timer example

```
struct my_timer_data {
    struct timer_list timer;
    // Add other data members as needed
    // ...
};
static struct my_timer_data my_data;

static int __init my_module_init(void)
{
    pr_info(PRINT_PREF "Initializing my_module\n");

    // Initialize the timer
    timer_setup(&my_data.timer, my_timer_callback, 0);

    // Set the timer to expire after 1000 milliseconds (1 second)
    mod_timer(&my_data.timer, jiffies + msecs_to_jiffies(1000));
```

Timer example

```c
void my_timer_callback(struct timer_list *t)
{
    struct my_timer_data *my_data = from_timer(my_data, t, timer);

    // Perform actions when the timer expires
    pr_info(PRINT_PREF "Timer expired!\n");

    // Restart the timer if needed
    mod_timer(&my_data->timer, jiffies + msecs_to_jiffies(1000));

static void __exit my_module_exit(void)
{
    pr_info(PRINT_PREF "Exiting my_module\n");

    // Delete the timer when unloading the module
    del_timer(&my_data.timer);
}
```

Delaying execution

Sometimes the kernel needs to wait for some time without using timers (bottom-halves)

- For example, drivers communicating with the hardware
- Needed delay can be quite short, sometimes shorter than the timer tick period

Several solutions

- Busy looping
- Small delays and BogoMIPS
- `schedule_timeout()`

Busy looping

Spin on a loop until a given amount of ticks has elapsed

- Can use `jiffies`, `HZ`, or `rdtsc`
- Busy looping is good for delaying *very short period time* but in general it is sub-optimal as wasting CPU cycles.

Busy looping

```c
/* Example 1: wait for 10 time ticks */
unsigned long timeout = jiffies + 10;    /* timeout in 10 ticks */
while(time_before(jiffies, timeout));    /* spin until now > timeout */

/* Example 2: wait for 2 seconds */
unsigned long timeout = jiffies + 2*HZ; /* 2 seconds */
while(time_before(jiffies, timeout));

/* Example 3: wait for 1000 CPU clock cycles */
unsinged long long timeout = rdtsc() + 1000;
while(rdtsc() > timeout);
```

Small delays and BogoMIPS

What if we want to delay for time shorter than one clock tick?

- If `HZ` is 100, one tick is 10 ms
- If `HZ` is 1000, one tick is 1 ms

Use `mdelay()`, `udelay()`, or `ndelay()`

- Implemented as a busy loop
- `udelay/ndelay` should only be called for delays < `1ms` due to risk of overflow

```
/* include/linux/delay.h */

void mdelay(unsigned long msecs);
void udelay(unsigned long usecs); /* only for delay <1ms due to overflow */
void ndelay(unsigned long nsecs); /* only for delay <1ms due to overflow */
```

Small delays and BogoMIPS

Kernel knows how many loop iterations the kernel can be done in a given amount of time: **BogoMIPS**

- Unit: iterations/jiffy
- Calibrated at boot time
- Can be seen in `/proc/cpuinfo`

```
└─$ cat /proc/cpuinfo | grep "bogomips"
bogomips        : 6000.00
bogomips        : 6000.00
```

https://en.wikipedia.org/wiki/BogoMips

`schedule_timeout()`

`schedule_timeout()` put the calling task to sleep for at least n ticks

- Must change task status to `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`
- Should be called from process context without holding any lock

```
set_current_state(TASK_INTERRUPTIBLE); /* can also use TASK_UNINTERRUPTIBLE */
schedule_timeout(2 * HZ); /* go to sleep for at least 2 seconds */
```

Sleeping on a waitqueue w/ timeout

Tasks can be placed on wait queues to wait for a specific event

To wait for such an event with a timeout:

- Call `schedule_timeout()` instead of `schedule()`

Further readings

LKD3: Chapter 11: Timers and Time Management

Operating System Virtualization

Xiaoguang Wang

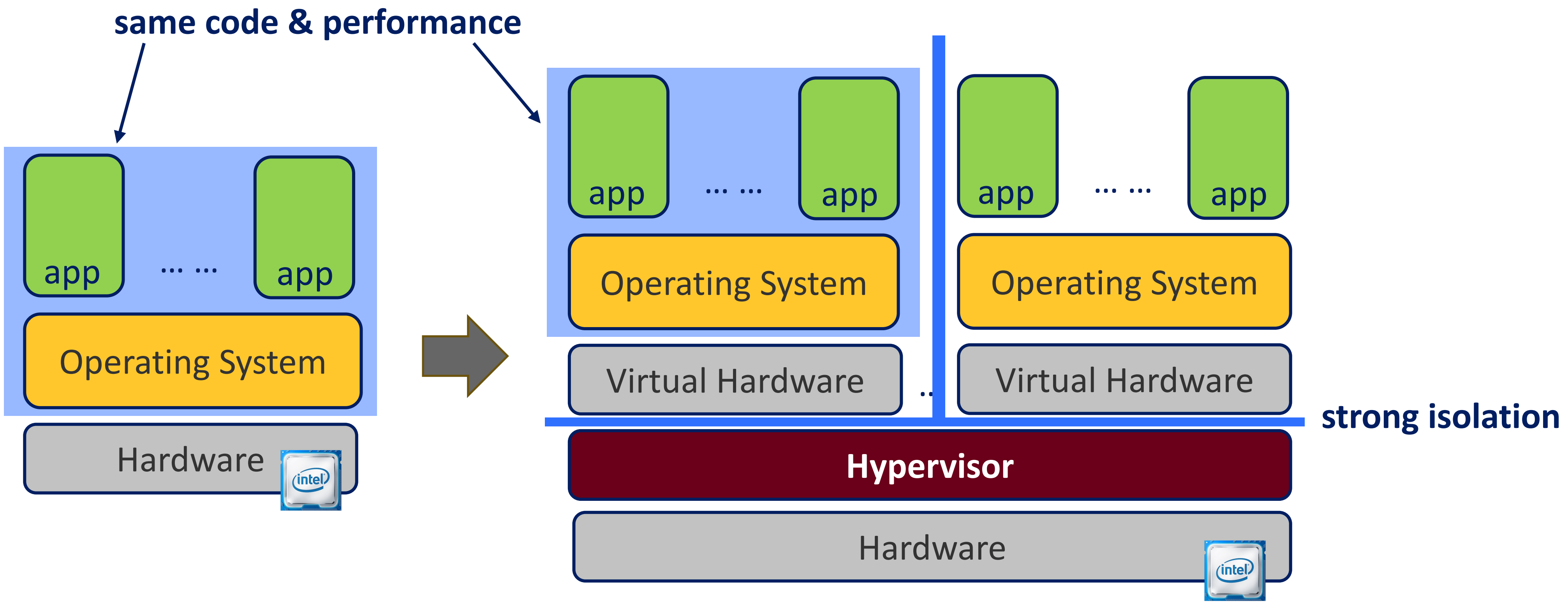
UIC COMPUTER SCIENCE

Introduction

What is virtualization?

- A set of software and hardware components to run multiple operating systems at the same time on the same physical machine
- Virtual Machine Monitor (Hypervisor) is a software layer that allows several virtual machines to run on a physical machine

Introduction



A bit of history

Virtual machines were popular in 60s-70s

- Share resources of mainframe computers [Goldberg 1974]
- Run multiple single-user operating systems

Interest is lost by 80s-90s

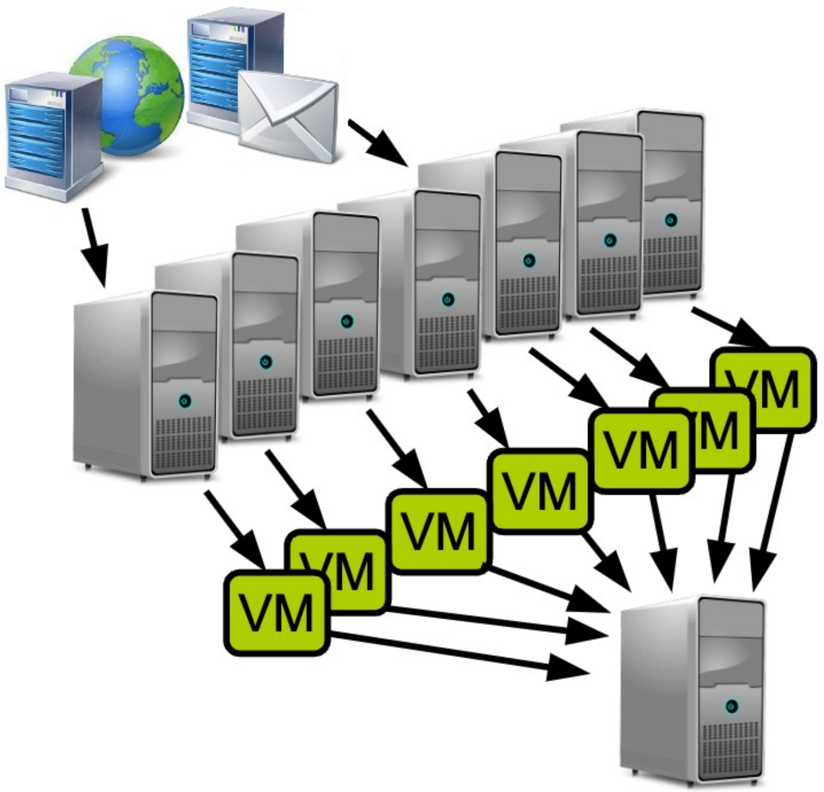
- Development of multi-user OS
- Rapid drop in hardware cost

Hardware support for virtualization is lost

Use cases: server consolidation

Consolidation: running X virtual machines on Y physical hosts (Y < X)

- Better utilize the physical resource
- Save money
- Historical motivation for developing virtualization technologies



Use cases: software development

Flexible OS diversity: different OS on the same machine

- E.g. Qemu with Linux for kernel development

Rapid and cost-efficient provisioning

- Way faster than a physical machine

VMs are self-contained

- Practical way to “pack” an application with all its software dependencies (OS versions, libraries, etc.)
- Useful for development, automated testing, and deployment (DevOps)

Use cases: migration, checkpoint/restart

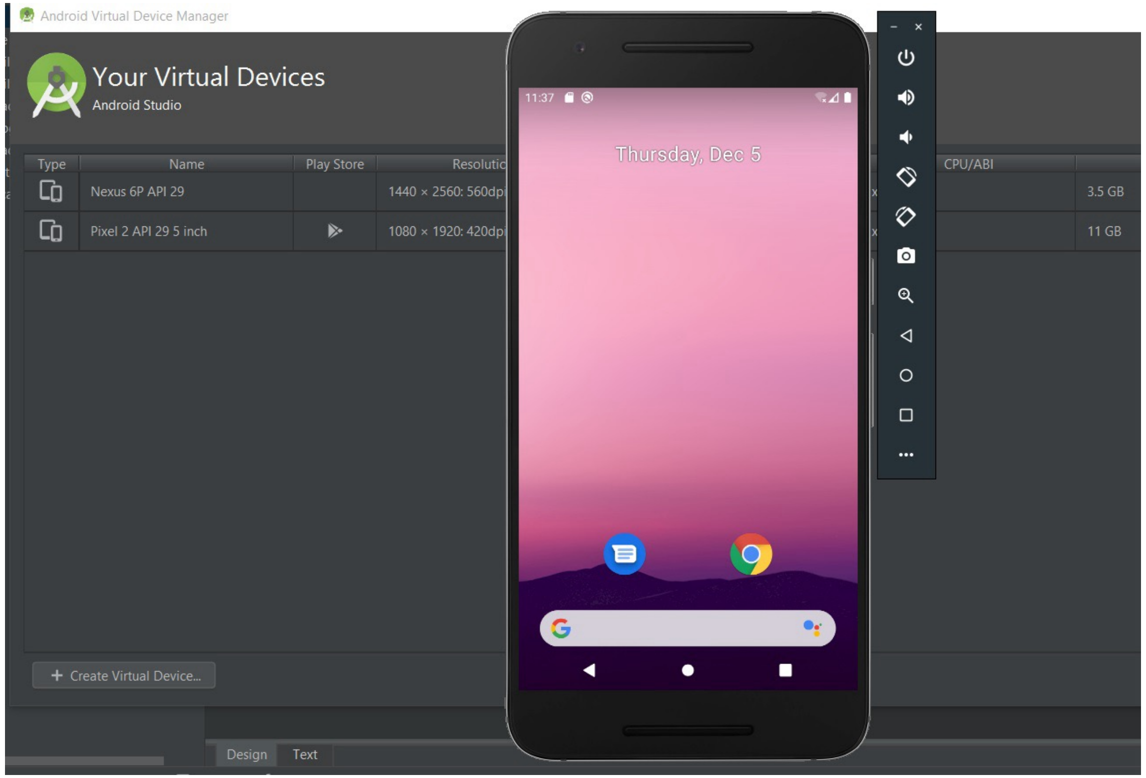
The state of a running VMs is easily identifiable hence the VM can be:

- Live-migrated: transparently move a running VM between hosts:
    - To free resources for maintenance, power saving, load balancing, or when a fault is expected
- Checkpointed and restarted: VM's state dumped on disk, can resume later

Both technique are straightforward for a VM as opposed to an application (i.e., a process)

Use cases: hardware emulation

For development, backward compatibility



Use cases: cloud computing

Virtualization enables cloud computing

- Allows cloud providers securely share their computing infrastructure between clients (tenants)

Cloud principle: offloading local tasks to remote computing resources, e.g.:

- Renting VMs to put a web server (IaaS)
- Deploy and run a web application using Google app engine (PaaS)
- Offload mail server online to Gmail/Outlook (SaaS)

Goals: save on management, infrastructure, development, maintenance costs

Use cases: security

Virtualization provides very strong isolation between guests

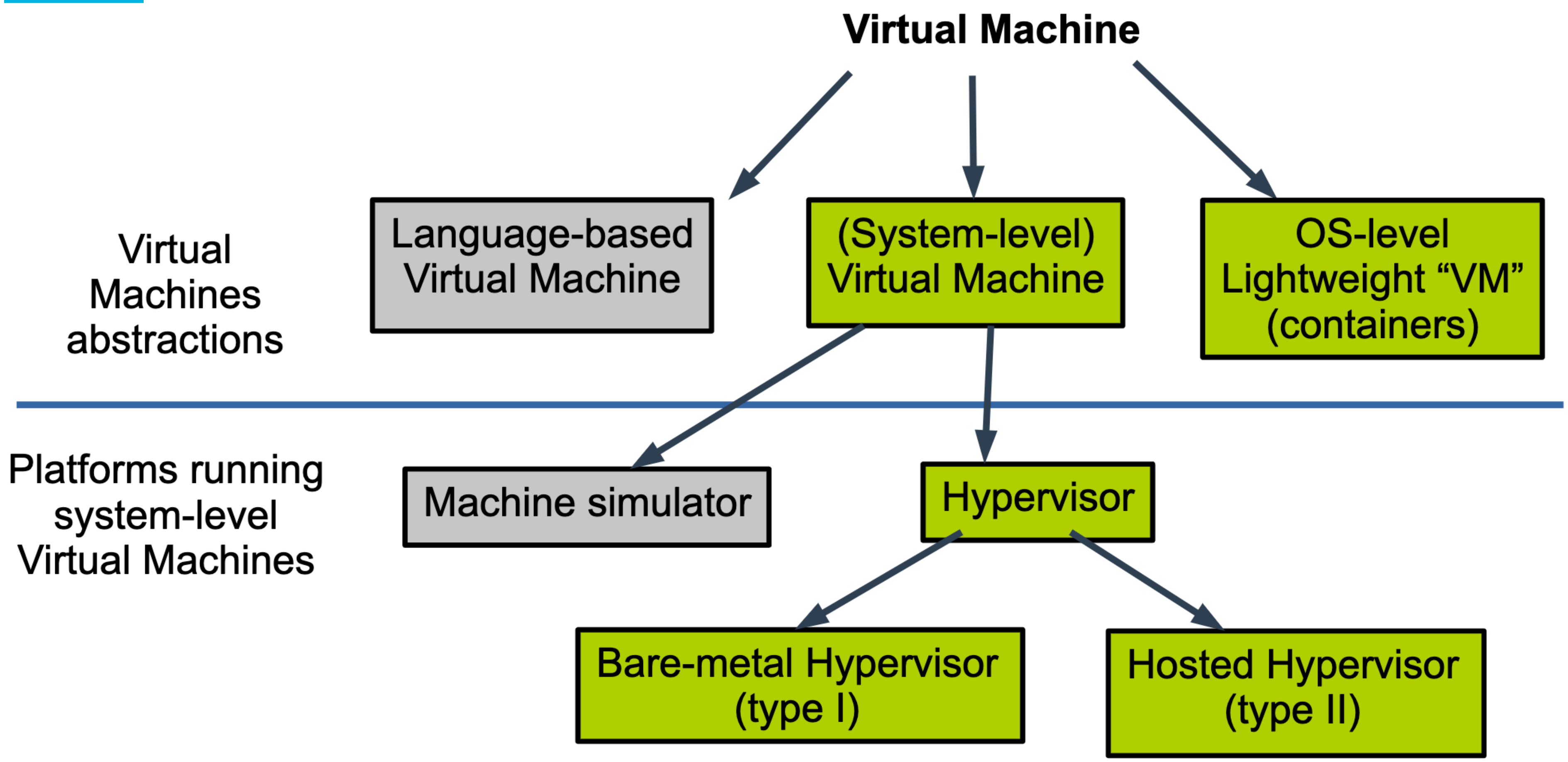
Sandboxing

- Cloud, virus/malware analysis, honeypots, process/task level isolation through virtualization

VM introspection

- Analysis of the guest behavior from a higher privileged level (i.e., VMM)

High-level categories of virtual machines

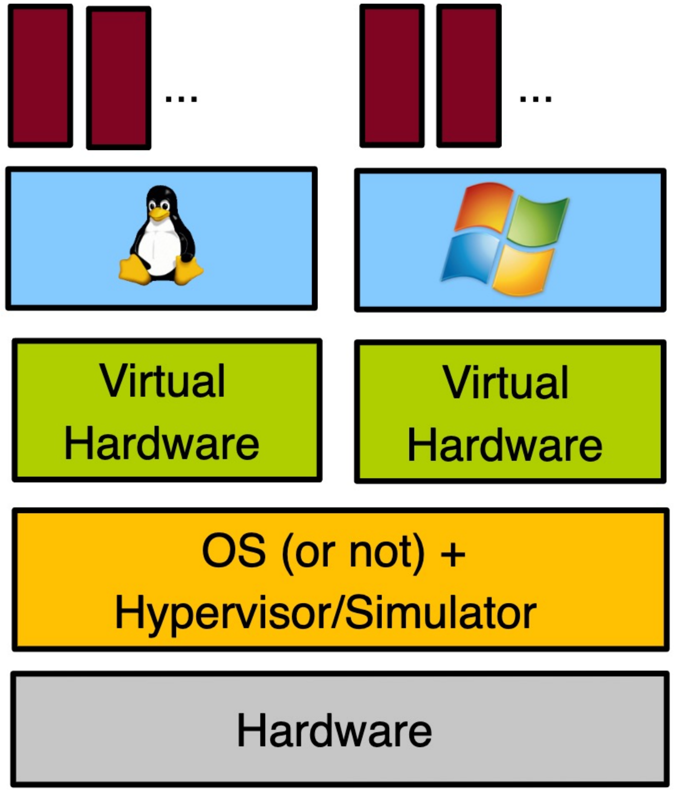


Ref: Pierre Olivier's slides

System-level Virtual Machines

Creates a model of the hardware for (mostly) unmodified OSes to run on top of it

- Each VM running on the computer has its own copy of the virtualized hardware
- Can run different OSes on one machine
- A hypervisor or a simulator



Machine simulators

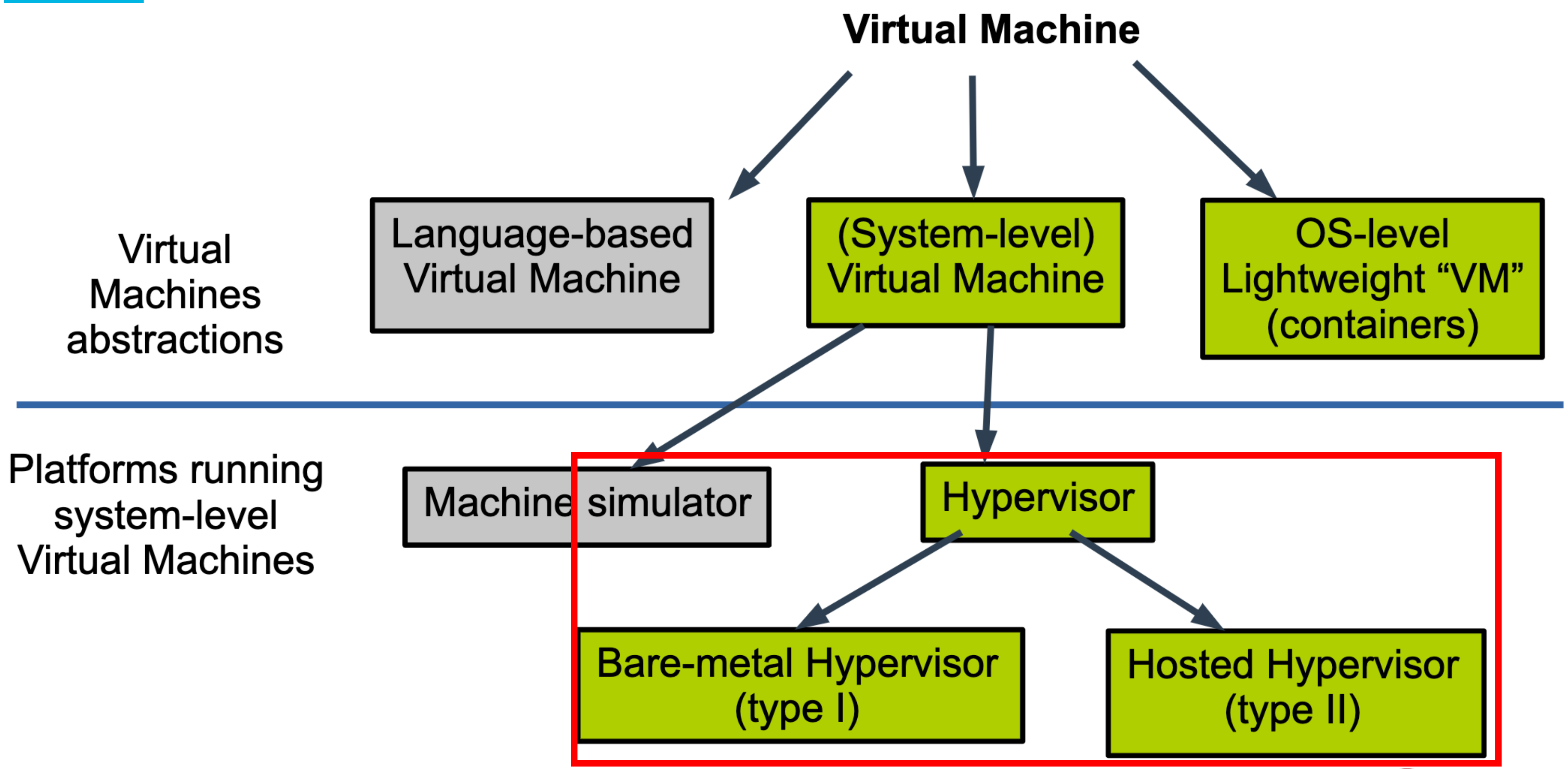
Create on a physical host machine a virtual machine of a different architecture

- Cross-ISA emulation: for usage as substitute
- Compatibility with legacy applications, software prototyping
    - E.g., QEMU in its full emulation mode

Architecture simulators (e.g., Gem5)

- Simulation for analysis and study: computer architecture prototyping, performance/power consumption analysis, research, etc.
- Each guest instruction is interpreted in software (5x to 1000x slowdown)

High-level categories of virtual machines



Ref: Pierre Olivier’s slides

Hypervisor-based VM

A hypervisor or Virtual Machine Monitor (VMM) creates a VM of the same architecture as the host

- Relies on hardware-based virtualization technologies (Intel VT-x, AMD SVM) to direct execute code on CPUs for close to native performance
- VM code executes directly on the **guest-mode** physical CPU, at a lower privilege level than the hypervisor (**host-mode**)

```
└─$ egrep -c '(vmx|svm)' /proc/cpuinfo
16
```

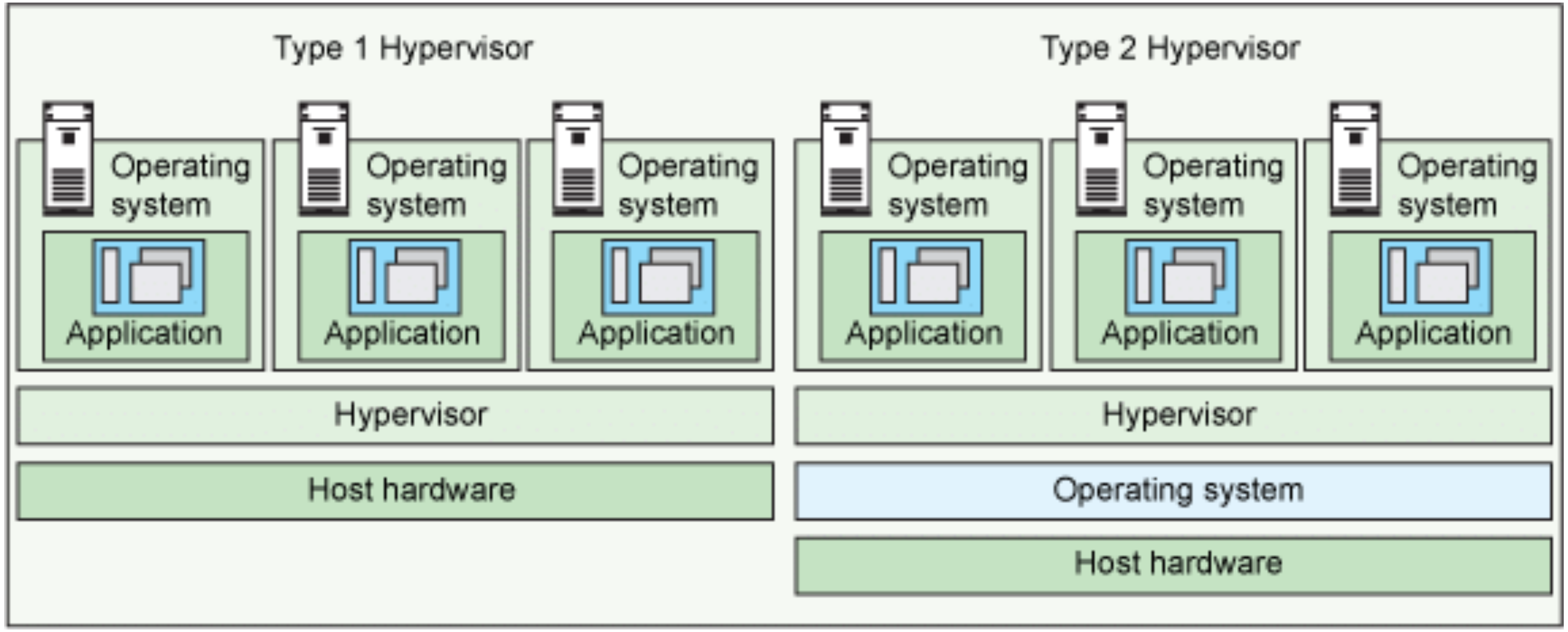
```
└─$ sudo lscpu | grep "Virt"
Virtualization:                       VT-x
```

Hypervisor-based VM

VMM emulates only sensitive instructions

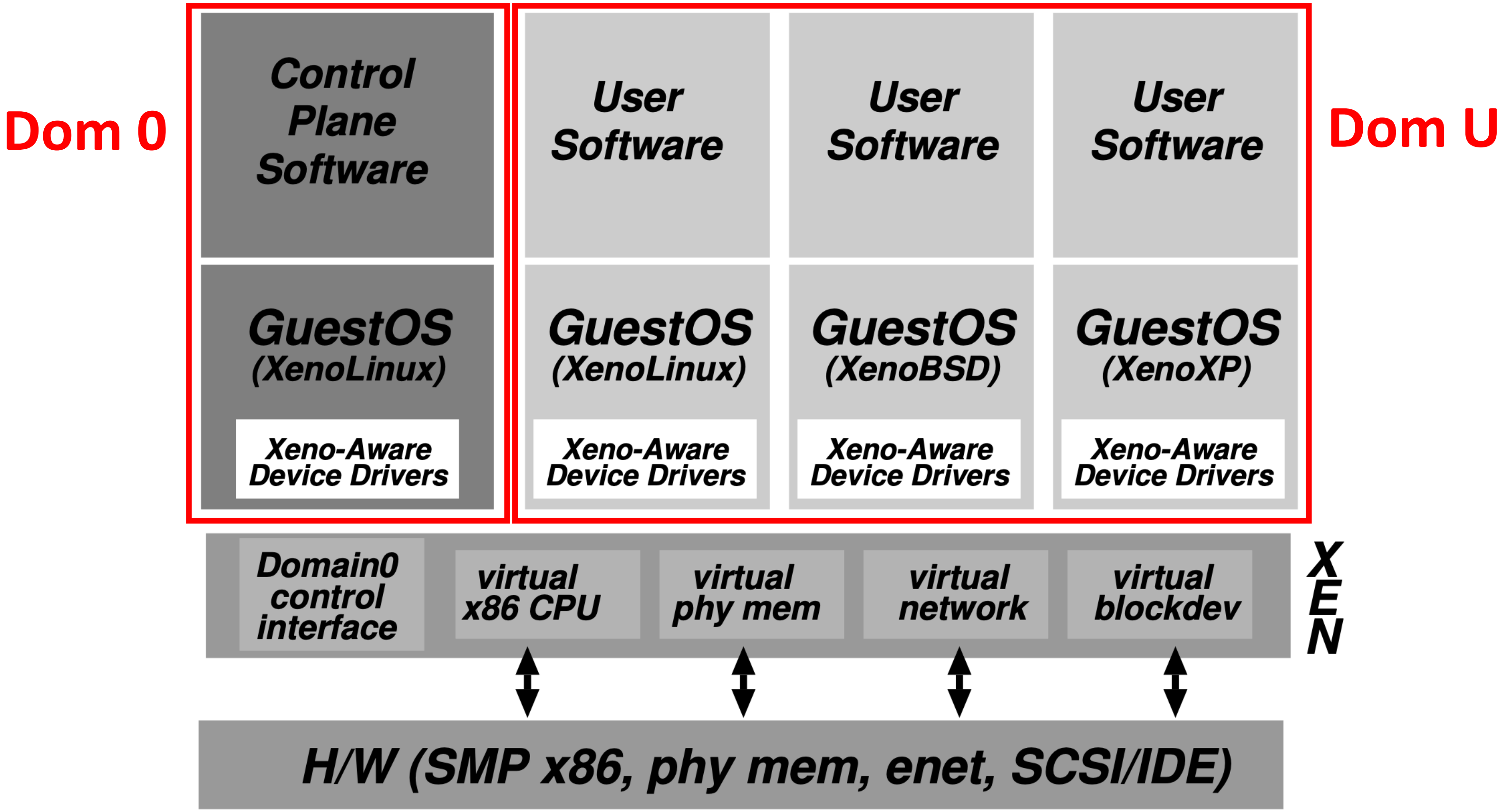
- Upon encountering a sensitive instruction, trap to the hypervisor and emulate the instruction execution
- E.g, Xen-HVM, Linux KVM, VMware ESXi, MS Hyper-V, Oracle VirtualBox.

Type I vs. II Hypervisors



Ref: Performance Analysis of Live and Offline VM Migration Using KVM

An example of the Type I hypervisor



Xen and the Art of Virtualization, SOSP'03

What needs to be emulated?

CPU and memory

- Register state
- Memory state

Memory management unit (MMU)

- Page tables, segments

I/O devices

- Disk, network interface, serial line

Platform

- Interrupt controller, timer, buses
- BIOS

x86 is not virtualizable

Some instructions (sensitive) read or update the state of virtual machine and don't trap (nonprivileged)

- 17 sensitive, non-privileged instructions [Robin et al 2000]

Examples

- `push %cs` can read code segment selector (`%cs`) and learn its CPL

Solution space

1. Parse the instruction stream and detect all sensitive instructions dynamically

- Trap and emulation (Binary translation)

2. Change the operating system

- Paravirtualization (Xen, L4, Denali, Hyper-V)

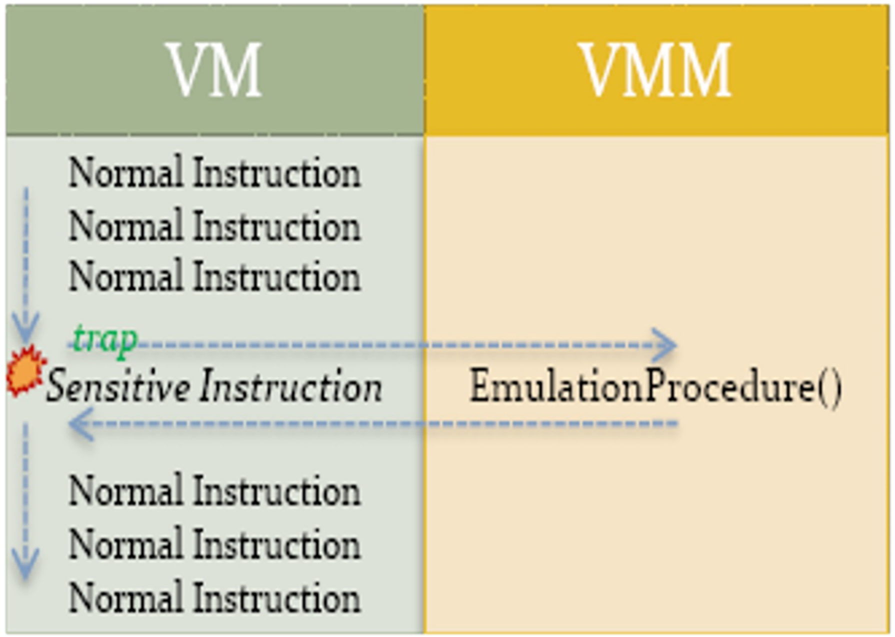
3. Make all sensitive instructions privileged!

- Hardware supported virtualization
  - Intel VT-x, AMD SVM

CPU Virtualization

Trap & Emulate

- Dynamic binary translate (DBT)
  - Code cache
- Sensitive instructions include
  - Instruction that changes processor mode
  - Instruction that accesses hardware directly
  - Instruction whose behavior is different in user/kernel mode



CPU Virtualization

Para-virtualization

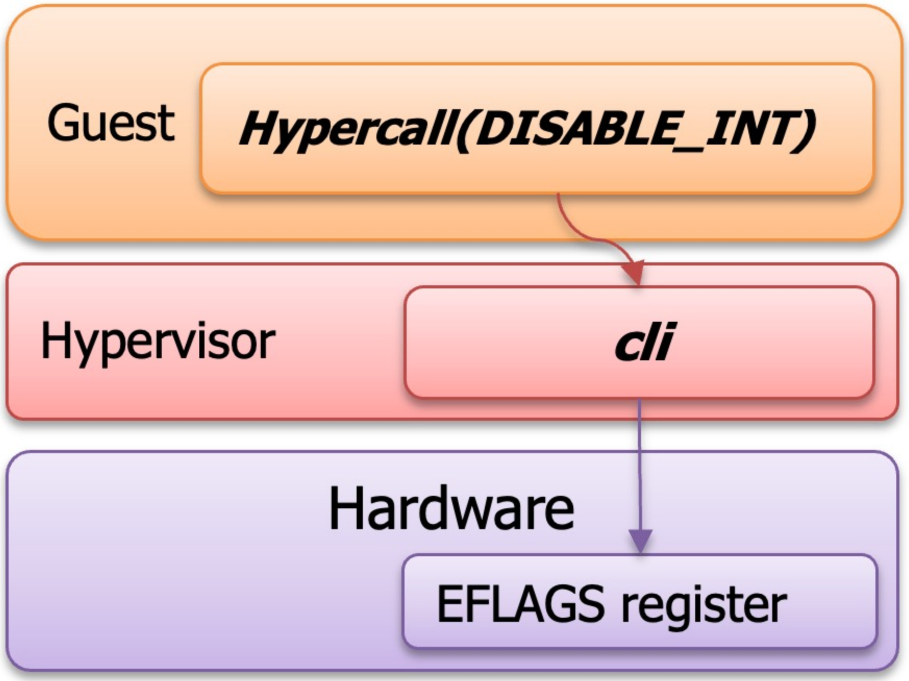
- Requires modifications to the guest OS
  - Guest is aware that it is running on a VM
  - Example: “cli” → hypercall(DISABLE_INT)

\+ Pros

- Near-native performance
- No hardware support required

\- Cons

- Requires specifically modified guest



VT-x: Root/Non-Root Transitions

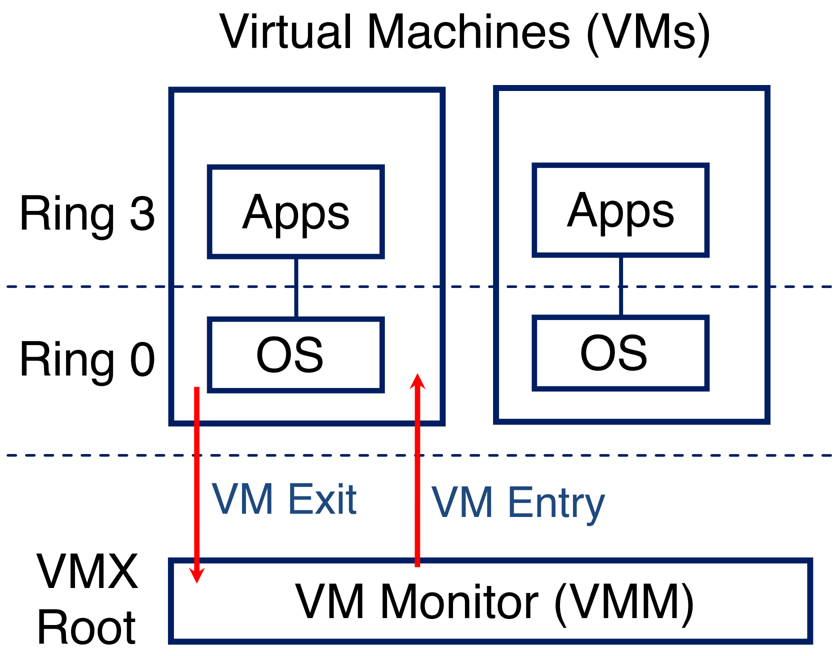
VMX (Virtual Machine Extension) supports virtualization of processor hardware.

Two new VT-x operating modes

- Less-privileged mode (VMX non-root) for guest OSes
- More-privileged mode (VMX root) for VMM

Two new transitions

- VM entry to non-root operation
- VM exit to root operation



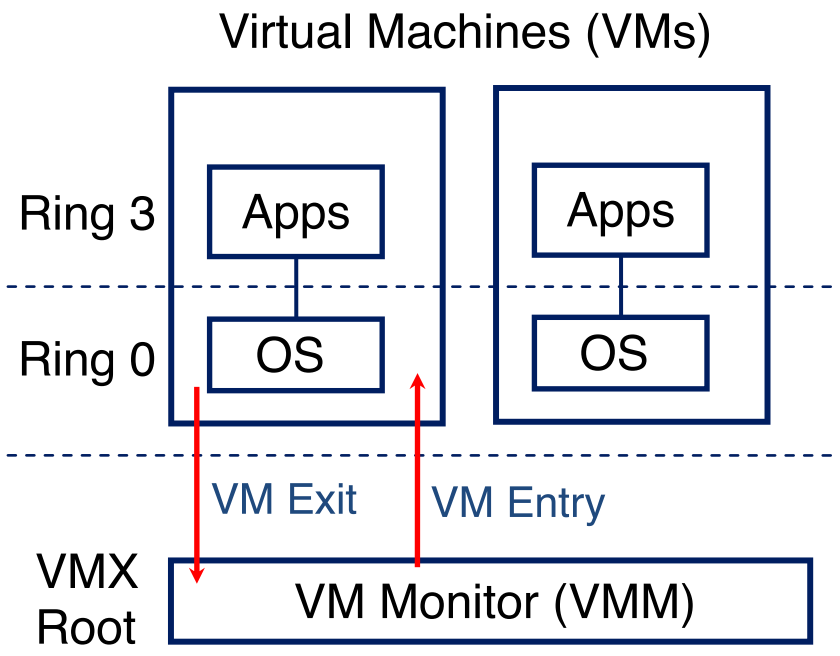
VT-x: Root/Non-Root Transitions

Execution controls determine when exits occur

- Access to privilege state, occurrence of exceptions, etc.
- Flexibility provided to minimize unwanted exits

VM Control Structure (VMCS) controls VMX operation

- Also holds guest and host state



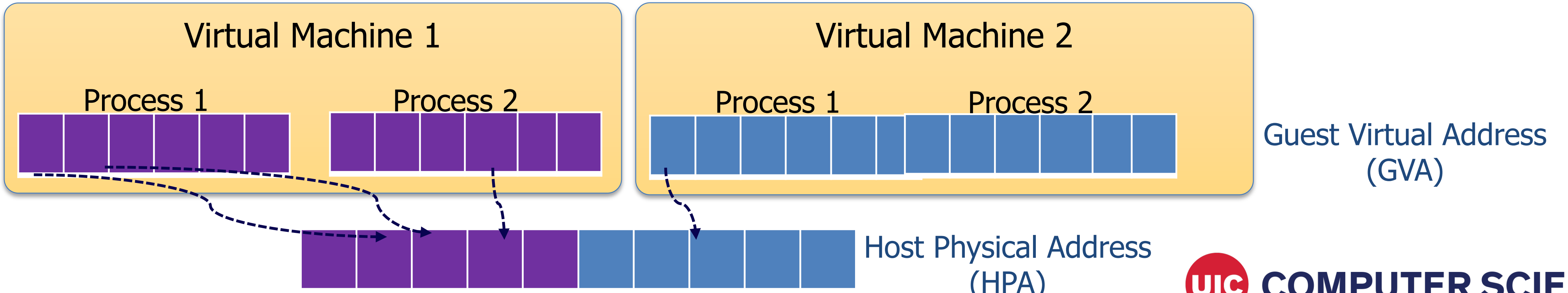
Memory Virtualization

Guest Virtual Address

Host Physical Address

Shadow Page Table

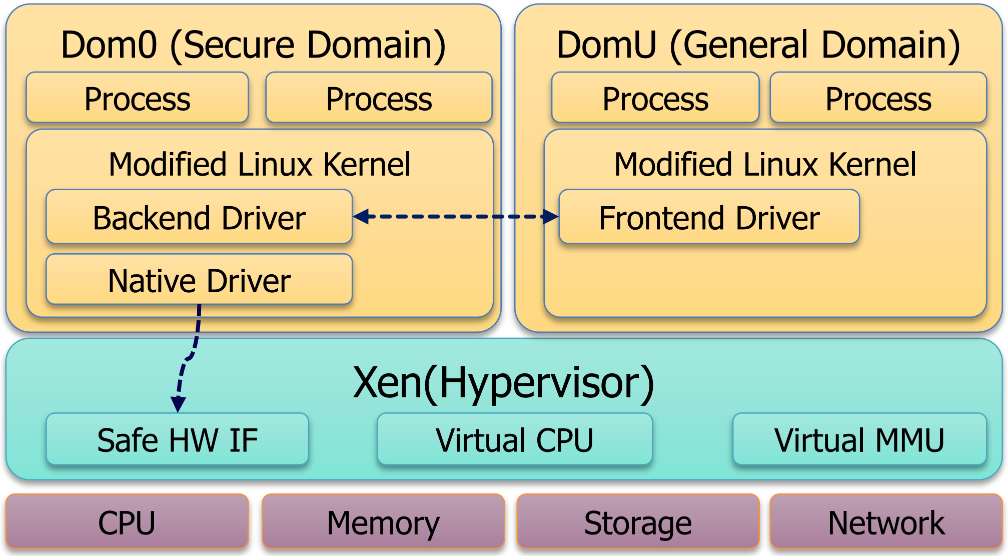
Nested (Extended) Page Table



I/O Virtualization

Front-end/Back-end Driver Model

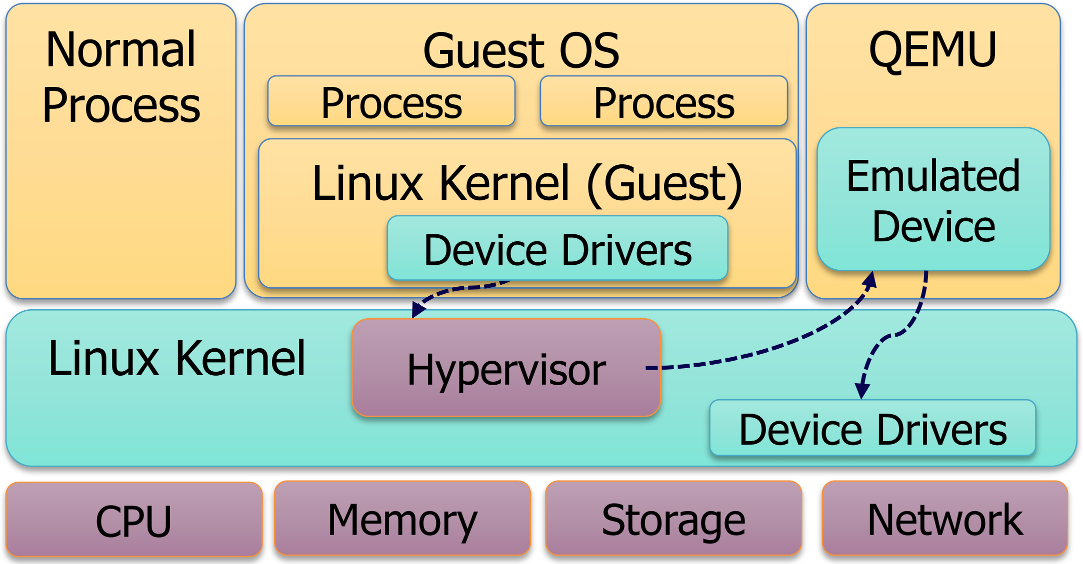
- Guest OS uses para-virtualized front-end driver to send requests to backend driver.
- Back-end driver on secure domain receives the requests, performs actual IO using the native driver.



I/O Virtualization

Emulation

- Behavior of a particular device is emulated as a software module.
- Guest OS uses the native device driver for the particular device.
- VMM intercepts all the access from guest OS to the device.
- The intercepted accesses are sent to the emulated device.
- The Emulated device do the actual IO operations.



I/O Virtualization

Single-Root IO Virtualization (SR-IOV)

- Hardware support (e.g., NIC)

A SR-IOV-enabled device can present several instances of itself

- Each assigned to a different VM
- The hardware multiplexes itself

A device has at least one Physical Function controlled by the hypervisor

- The instances of itself visible to VMs are called Virtual Functions

I/O Virtualization

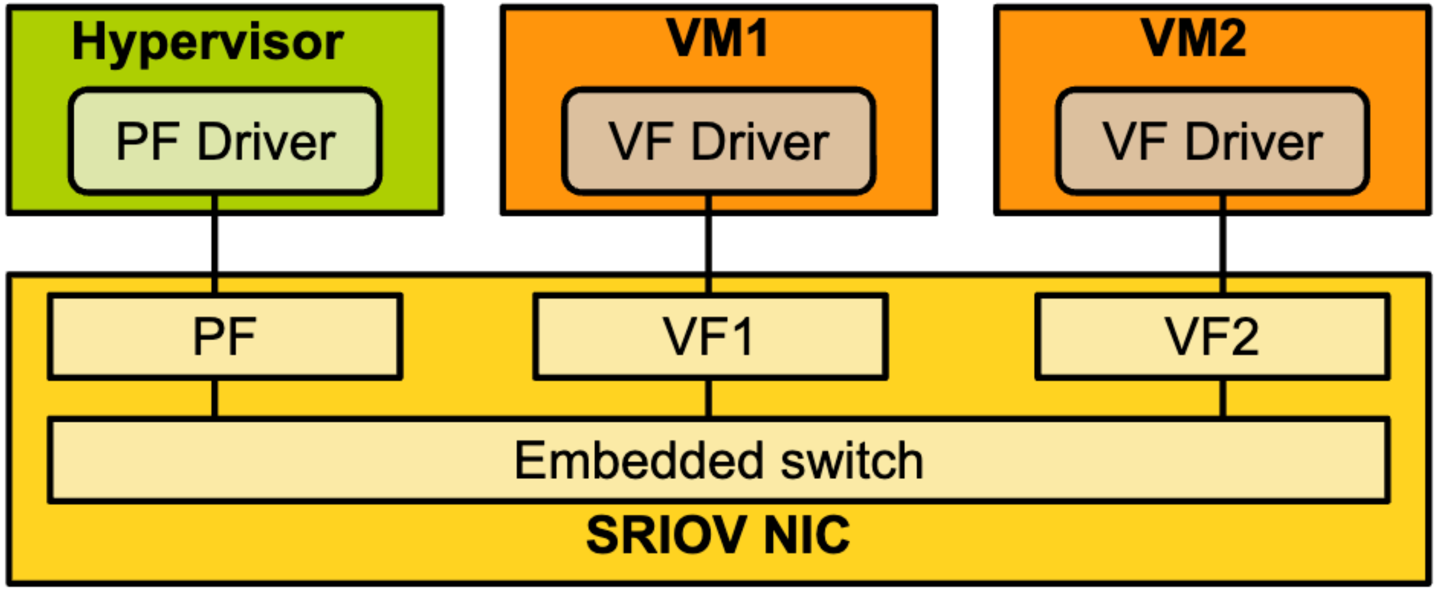
Single-Root IO Virtualization

- Hardware support (e.g., N

A SR-IOV-enabled device ca

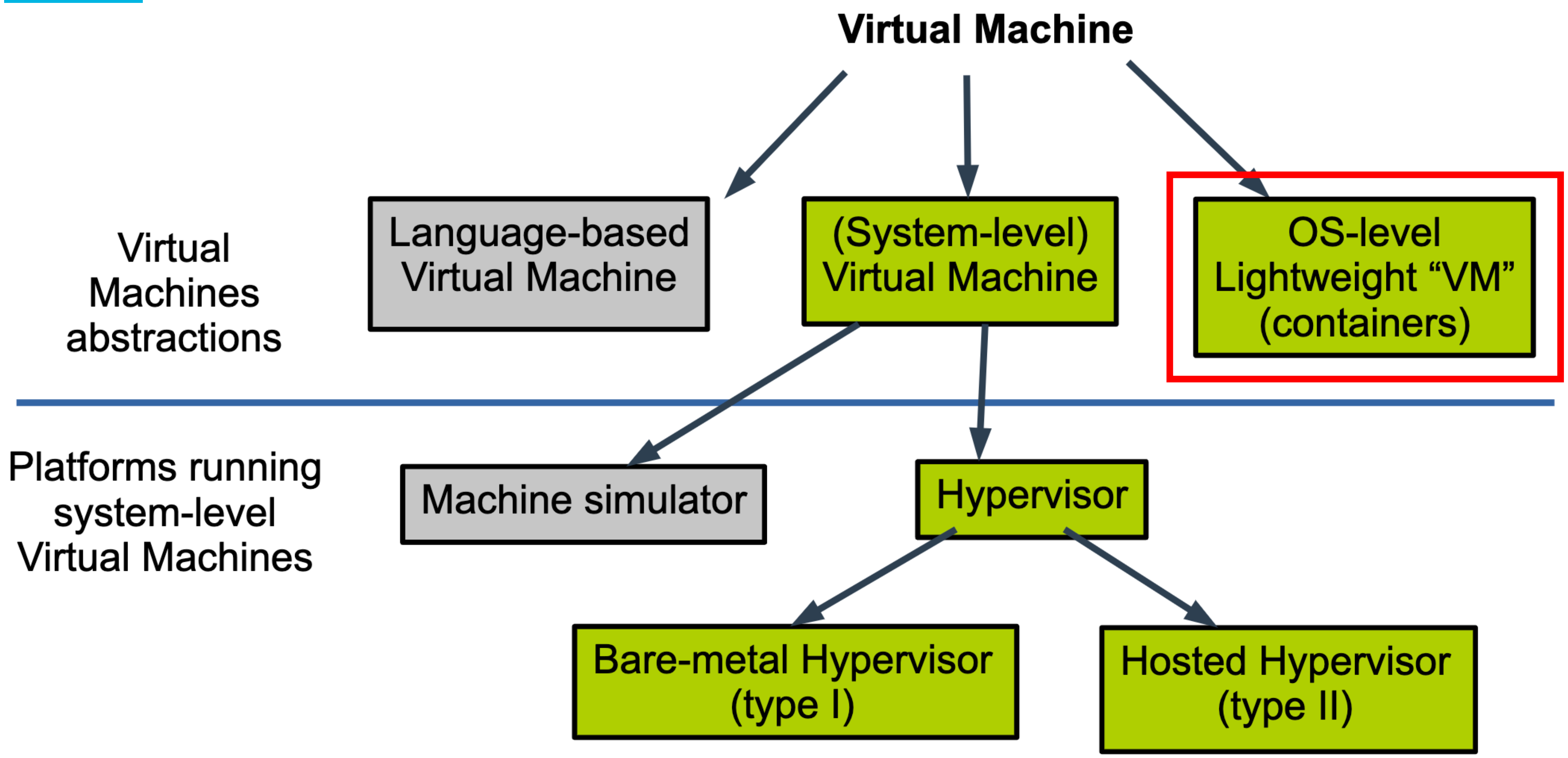
- Each assigned to a differe
- The hardware multiplexes

A device has at least one Ph



- The instances of itself visible to VMs are called Virtual Functions

High-level categories of virtual machines



Ref: Pierre Olivier's slides

OS-level Lightweight VMs

E.g., containers

- Isolation of native applications through OS mechanisms (chroot, cgroups)
  - Also packs the dependencies (and libraries) into the container
  - An illusion of running applications in an OS
  - Lightweight: fast boot than a hypervisor-based VM
- No attempt is made to virtualize the hardware
- Cannot run a different OS type (e.g., run windows on a Linux host)

Writing a container in a few lines of Go code:
https://github.com/lizrice/containers-from-scratch