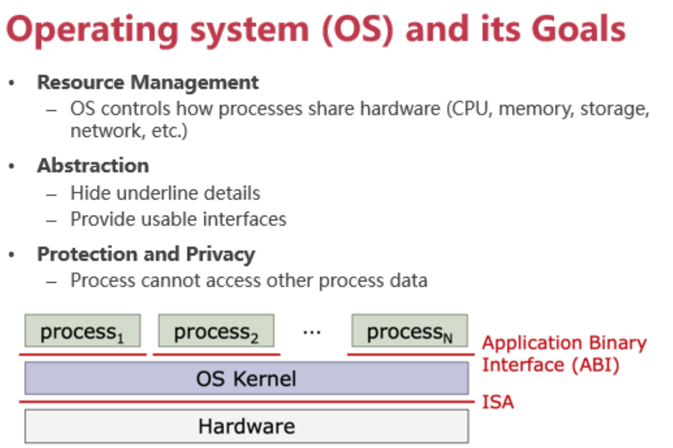
System virtualization



资源管理，向下管理各种硬件资源

向上抽象，让上层以为自己独占资源，可以正常使用资源。

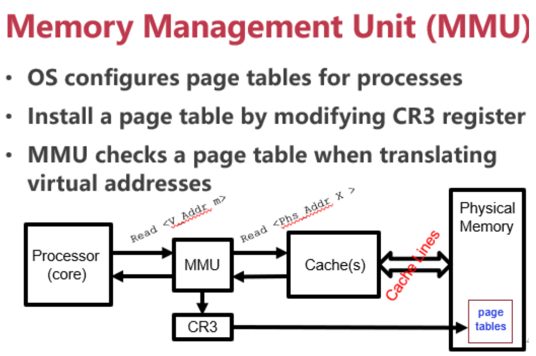
保护、隐私。不能接触到其他进程的数据。

操作系统与硬件之间的接口是 ISA。

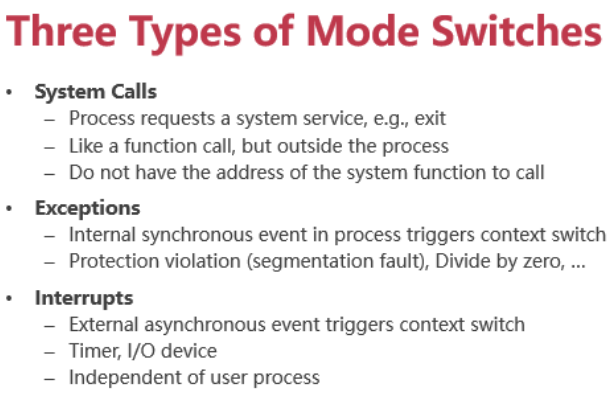
操作系统与进程之间的接口是 ABI（系统调用）。

进程抽象

1. 每个进程有自己的私有地址空间
2. 操作系统调度进程占用 CPU（分时）
3. 进程通过系统调用来使用 OS 的服务



模式转换的方式：



OS 提供的系统隔离

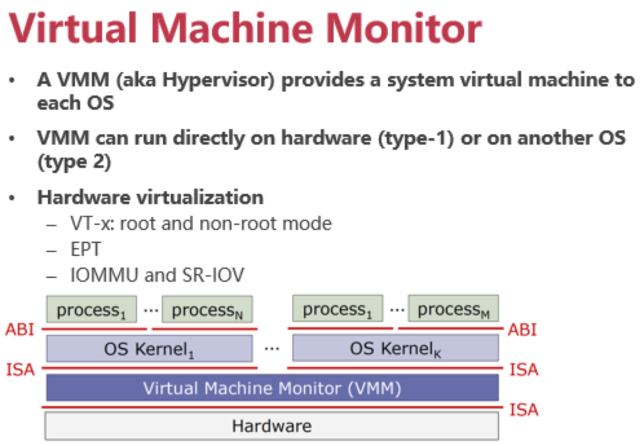
进程之间的隔离

技术：page tables，context switch，file abstraction

进程和内核之间的隔离

CPU 特权级

VMM



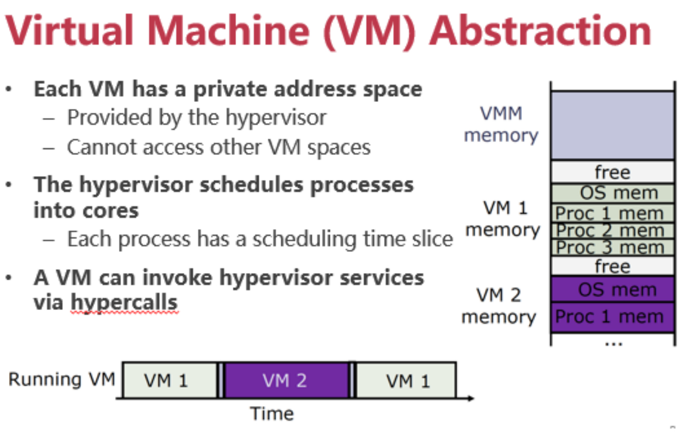
两种 VMM

Type1：直接运行在硬件上

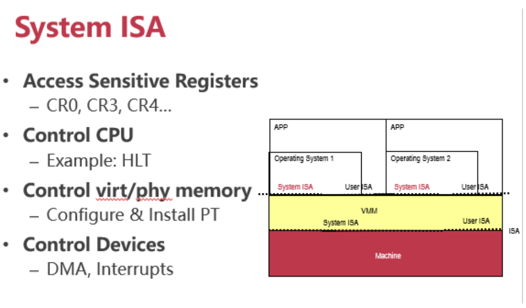
Type2：运行在已有的操作系统上

Type2 可以只关注虚拟化的部分，其他例如设备驱动的功能可以复用 OS。

和 OS 对进程的抽象类似，VM 抽象：



System ISA：可以访问敏感资源的 ISA

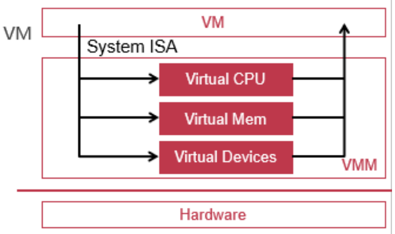


CR3：进程对应的页表地址。

系统虚拟化的步骤：

Procedure of System Virtualization

- **Step 1**
  - Catch all system ISA of a VM
- **Step 2**
  - Provide three functions
    - Virtualize CPU
    - Virtualize Memory
    - Virtualize I/O
- **Step 3**
  - Resume VM



首先要捕获 VM 的**所有的**系统 ISA
识别并提供相关的服务。
回到虚拟机中继续执行。

CPU 虚拟化

如果简单把 OS 当作普通的 Application，在 OS 上去跑，问题是客户机 OS 要执行特权指令，但是其所在特权级不能执行特权指令。

Solution: Trap & Emulate

- **Trap: running privilege instructions will trap to the VMM**
- **Emulate: those instructions are implemented as functions in the VMM**

问题是，架构并非是"严格可虚拟化"的。

Problems of Trap & Emulate

- **Not all architectures are "strictly virtualizable"**
- **An ISA is strictly virtualizable if, when executed in a lesser privileged mode:**
  - All instructions that access privileged state trap
  - All instructions either trap or execute identically

X86 结构中，一些特权级指令，在用户特权级下运行时，不会正常生效，只是静默地跳过去，什么都不发生，不会 trap。

解决方法：

How to Deal with the 17 Instructions?

1. **Instruction Interpretation: emulate them by software**
2. **Binary translation: translate them to other instructions**
3. **Para-virtualization: replace them in the source code**
4. **New hardware: change the CPU to fix the behavior**

Sol-1：指令翻译

Sol-1: Instruction Interpretation

- **Emulate Fetch/Decode/Execute pipeline in software**
  - Emulate all the system status using memory
    - E.g., using an array `GPR[8]` for general purpose registers
  - None guest instruction executes directly on hardware
- **E.g., Bochs**

把每一条指令都用软件模拟执行。
最大的问题就是慢。

Sol-2 二进制翻译

Sol-2: Binary Translator

- **Translate before execution**
  - Translation unit is basic block (why?)
  - Each basic block -> code cache
  - Translate the 17 instructions to function calls
    - Implemented by the VMM
- **E.g., VMware, Qemu**

在执行之前提前翻译好。
翻译的单元是基础块。
把 17 个指令翻译成 call

Sol-3 半虚拟化

Sol-3: Para-virtualization

- **Modify OS and let it cooperate with the VMM**
  - Change sensitive instructions to calls to the VMM
    - Also known as **hypercall**
  - Hypercall can be seen as trap
- **E.g., Xen**
  - Was widely used by industry like Amazon's EC2

改操作系统，把敏感指令换成 hypercall。
（具有可行性是因为要改的敏感指令少，常用的操作系统少）

Sol-4 硬件虚拟化

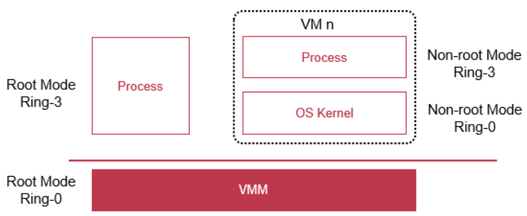
把 CPU 特权级进一步细化。
在 4 个特权级之外的维度，加了 root 和 non-root 的模式。

Sol-4: Hardware Supported CPU Virtualization

- **VMX root operation:**
  - Full privileged, intended for Virtual Machine Monitor
- **VMX non-root operation:**
  - Not fully privileged, intended for guest software

Both forms of operation support all four privilege levels from 0 to 3

Sol-4: Hardware Supported CPU Virtualization



VMCS：VM control structure

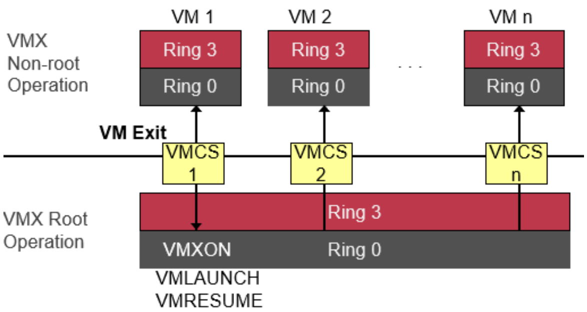
VT-x VMCS

The VMCS consists of six logical groups:

- **Guest-state area:** Processor state saved into the guest-state area on VM exits and loaded on VM entries.
- **Host-state area:** Processor state loaded from the host-state area on VM exits.
- **VM-execution control fields:** Fields controlling processor operation in VMX non-root operation.
- **VM-exit control fields:** Fields that control VM exits.
- **VM-entry control fields:** Fields that control VM entries.
- **VM-exit information fields:** Read-only fields to receive information on VM exits describing the cause and the nature of the VM exit.

VT-x 工作流程

VT-x Workflow



首先执行 VMXON 指令，告诉 CPU 要使用虚拟化了。
执行 VMLAUNCH，就会进入虚拟机中执行。
遇到特权指令后会 VM Exit，回到 VMM，执行完之后执行 VMRESUME，再回到 VM。
每个虚拟机有一个 VMCS，告诉有哪些指令需要下陷。

内存虚拟化

扩展了内存地址种类：

Terminology: 3 types of address now
- **GVA->GPA->HPA** (Guest virtual. Guest physical. Host physical)
- Guest VM's page table contains GPA

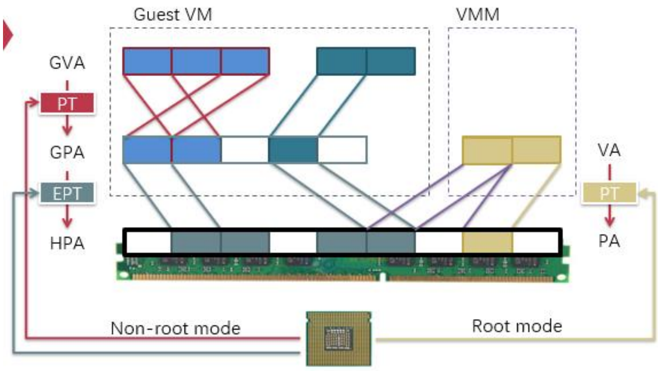
解决方法：

- Traditional solutions: **shadow paging & direct pa**
- Today's solution: **new hardware**

扩展页表

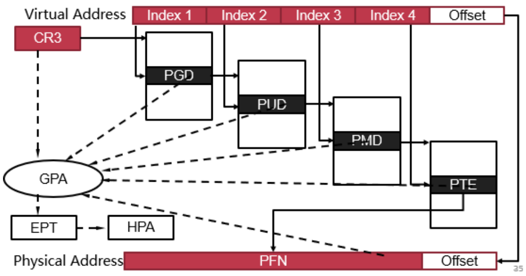
- **Hardware implementation**
  - Intel's EPT (Extended Page Table)
  - AMD's NPT (Nested Page Table)
- **Another table**
  - EPT for translation from **GPA to HPA**
  - EPT is controlled by the hypervisor
  - EPT is per-VM

EPT 用来把 GPA 翻译成 HPA



原本的地址翻译过程：

Any GPA is translated to HPA



5 次内存访问。
EPT 翻译与此类似。
意味着需要 20 次（4*5）内存访问。

IO 虚拟化

I/O Virtualization

- **Goal**
  - Multiplexing device to guest VMs
- **Challenges**
  - Each guest OS has its own driver
  - How can one device be controlled by multiple drivers?
  - What if one guest OS tries to format its disk?

四个方案：

Solutions for I/O Virtualization

1. **Direct access**: VM owns a device exclusively
2. **Device emulation**: VMM emulates device in software
3. **Para-virtualized**: split the drivers to guest and host
4. **Hardware assisted**: self-virtualization device

Sol-1 直通独占

问题：虚拟机可以通过 DMA 让设备访问到其他虚拟机的内存

解决方法：设备页表，IOMMU

IOMMU: Page Tables for Devices

- **Allow guest OS direct access to underlying device**
  - Guest just reuses its own device driver
- **Q: What if a VM asks device to access memory of other VMs?**
  - It is possible because device accesses HPA in DMA
  - Thus a device can access any memory
- **Solution: Page tables for devices**
  - Another MMU: IOMMU for devices
  - Q: what addresses will IOMMU translate?

优点：快，简化 VMM

缺点：guest 直接操作硬件接口（难以做 VM 迁移）（不走 VMM）。需要更多设备。

Direct Access Device Virtualization

- **Positives**
  - Fast, since the VM uses device just as native machine
  - Simplify monitor: limited device drivers needed
- **Negatives**
  - Hardware interface visible to guest (bad for migration)
  - Interposition is hard by definition (no way to trap & emulate)
  - Now you need much more devices! (image 100 VMs)

Sol-2 模拟设备

用软件模拟设备逻辑

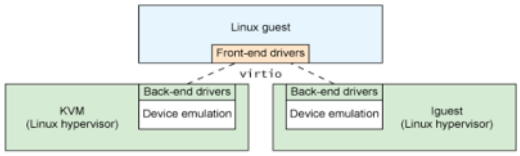
Emulated Devices

- **Positives**
  - Platform stability (good for migration)
  - Allows interposition
  - No special hardware support is needed
- **Negatives**
  - Can be slow (it' s software emulated)

Sol-3 半虚拟化

Sol-3: Para-Virtualized Devices

- **VMM offers new types of device**
  - The guest OS will run a new driver (front-end driver)
  - The VMM will run a back-end driver for each front-end
  - The VMM will finally run device driver to drive the device



VirtIO Architecture by KVM

两边互相知道，可以更好的配合，比如数据可以在前端 batch 之后再给后端。

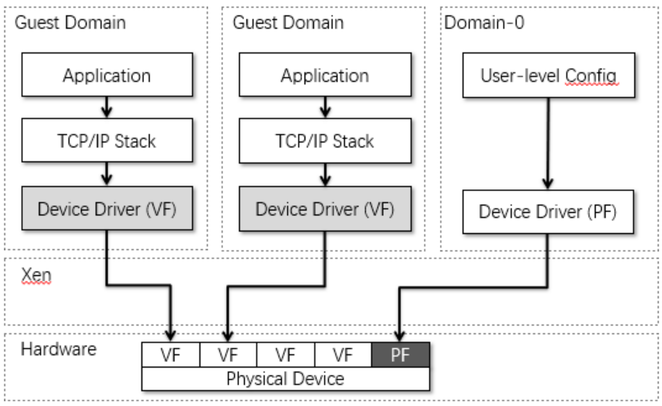
因此快。

Sol-4 硬件虚拟化

Sol-4: Hardware Support for I/O Virtualization

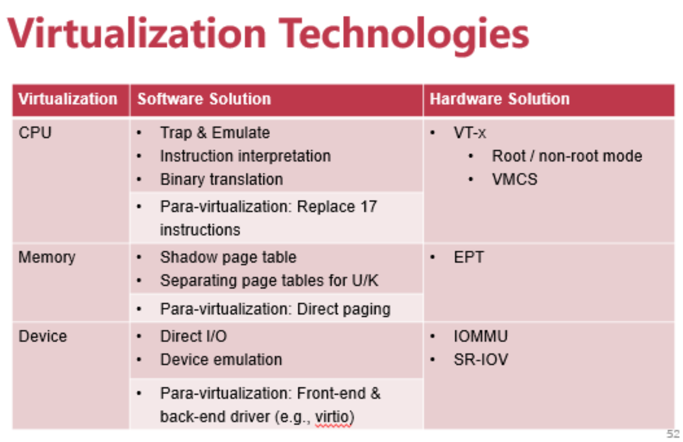
- **VMM**
  - An SR-IOV-capable device can be configured to appear in the PCI configuration space as multiple functions
- **VM**
  - The VMM assigns one or more VFs to a VM by mapping the actual configuration space of the VFs to the configuration space presented to the virtual machine by the VMM

SR-IOV：硬件的一种虚拟化标准



PF 具有管理功能，被 VMM 控制。

总结表格：



Virtualization Technologies

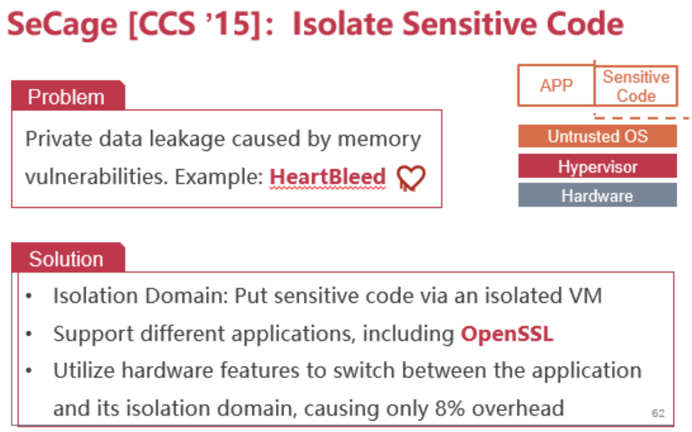
Virtualization	Software Solution	Hardware Solution

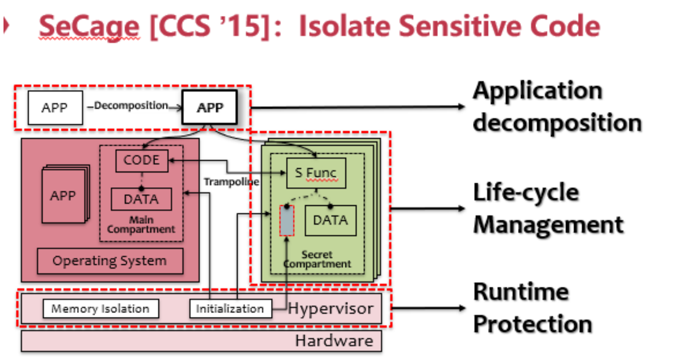
CPU	• Trap & Emulate
	• Instruction interpretation
	• Binary translation
	• Para-virtualization: Replace 17 instructions
		• VT-x
		  • Root / non-root mode
		  • VMCS

Memory	• Shadow page table • Separating page tables for U/K • Para-virtualization: Direct paging	• EPT
Device	• Direct I/O • Device emulation • Para-virtualization: Front-end & back-end driver (e.g., virtio)	• IOMMU • SR-IOV

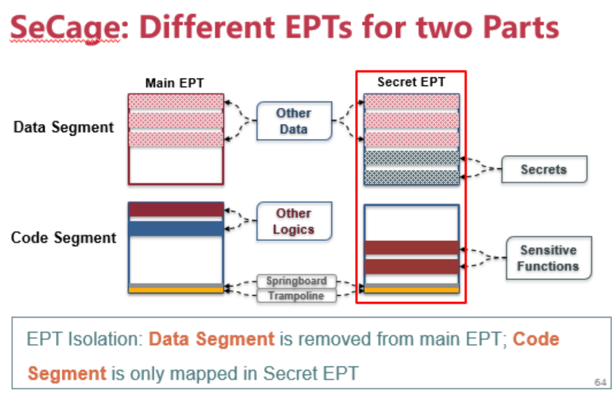
52

Topic：Isolation Via VMM





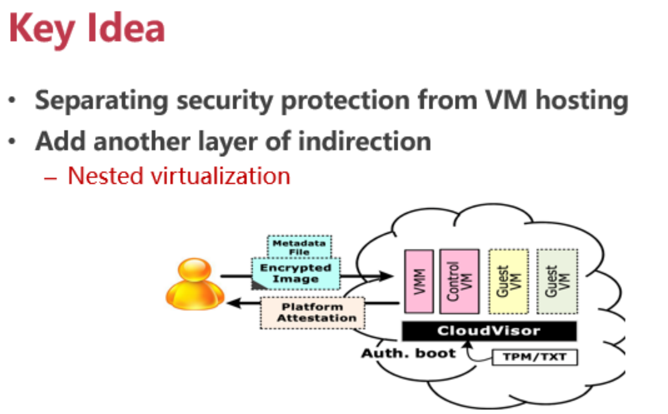
秘密数据部分摘出来，为其单独分配一个 EPT，使得 OS 不能直接访问。



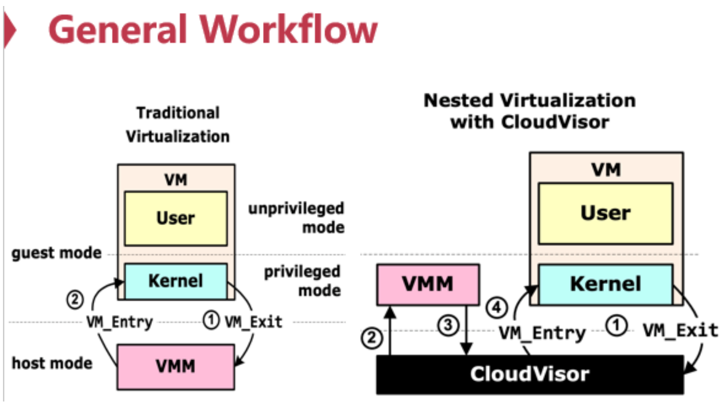
缺点：如果秘密数据很多，被频繁用，那么每次访问需要到 VMM 里换页表，很慢。

解决方案：intel 加了一个指令 VMFUNC，可以在 VM 里换页表，不需要 VM Exit

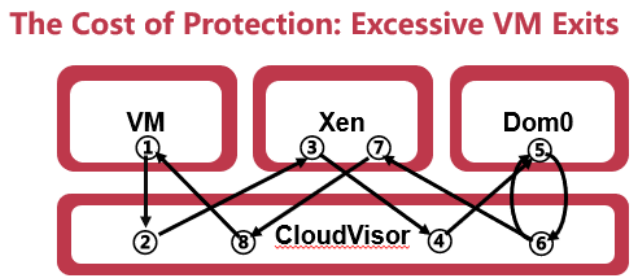
CloudVisor：在恶意的 hypervisor 情况下保护 VM



管理和安全分离。VMM 被降权，CloudVisor 在最底层，起保护作用。

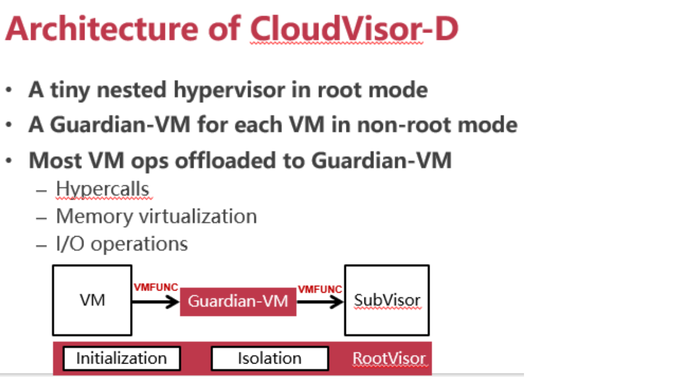


开销过大，慢



Operation	Times
Hypercall	>= 2X
EPT Violation	2 – 6 X
DMA Operation	>= 2X

改进 CloudVisor-D



Guardian-VM 在 non-root mode，承担保护安全的作用。

用 VMFUNC 切换页表。

数据流控制流分离

Language Virtual Machines

Native 语言，比如 C++，编译、部署。

便利性问题：在多个平台下，多个 ISA，需要每个平台都有对应的编译器，编译出对应的 binary。部署环境越来越复杂，不同的操作系统，不同的硬件平台。
安全性问题：

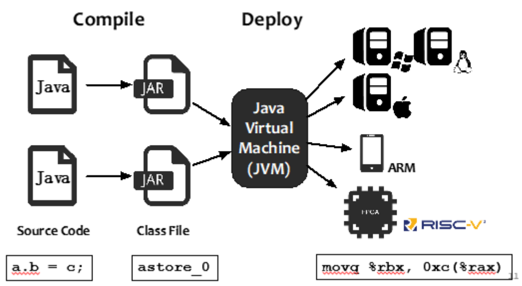
- **Out-of-bound access**

```
char * chs = malloc(10);
Char ch = chs[10]; // oops!
```

- **Use-after-free**

```
free(chs);
Char ch = chs[5]; // oops!
```

解决思路：加一层抽象



一次编译，处处执行。
安全问题的解决：

Security Enhancement

- **Out-of-bound access ← runtime check**

```
char * chs = malloc(10);
Char ch = chs[10]; // ArrayOutOfBoundException
```

- **Use-after-free ← no free at all!**

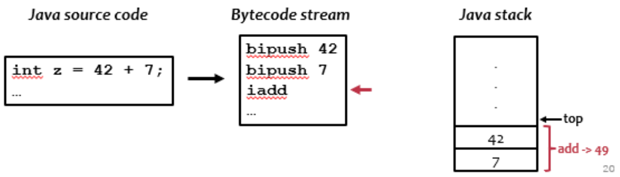
```
~~free(chs);~~
Char ch = chs[5]; // No problem!
```

托管给 JVM 运行，JVM 负责运行时检查、内存管理的工作。

执行模式：解释执行
在 Java 栈上执行。

Stack-based Interpretation

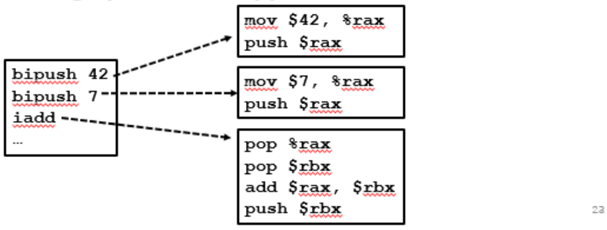
- **All instructions (bytecode) are operated on the stack**



问题：efficiency

Problem: Efficiency

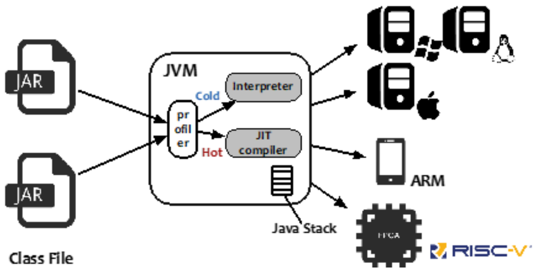
- **Too many memory operations (>=1 per bytecode)**
  - Low utilization of registers
- **Missing optimization opportunities**



内存操作太多，对寄存器的利用率低，难以优化。
解决方法：JIT compiler

Solution: JIT (Just-In-Time) Compiler

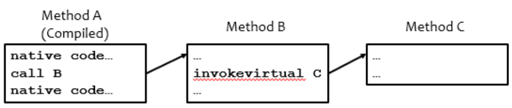
- **Hot code will be compiled and executed**
  - "Hot" means that it has been executed for times



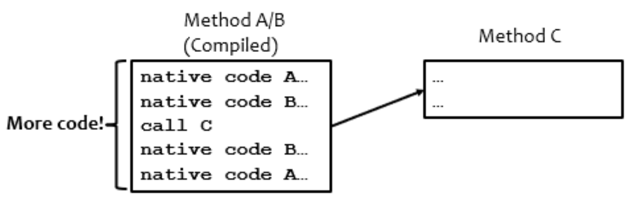
并不是所有的代码都编译，有一个 profiler 进行统计，偶尔执行的就直接解释执行，经常执行的就进行编译。

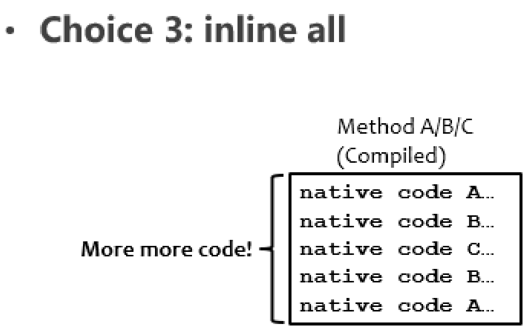
JIT：激进 or 保守

- **Choice 1: only compile A**

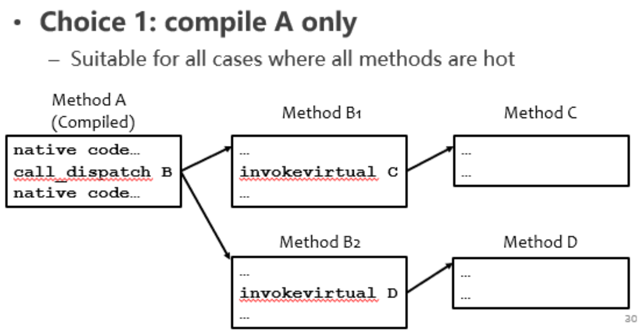


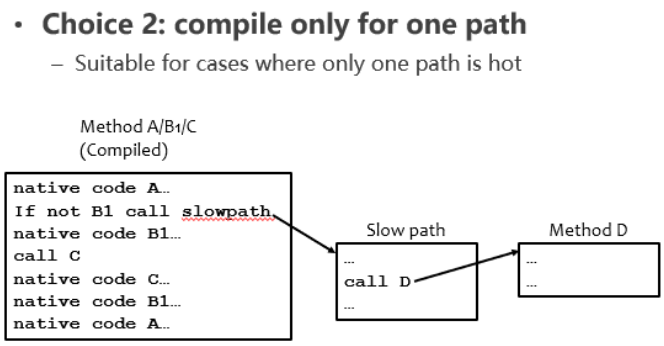
- **Choice 2: inline B**



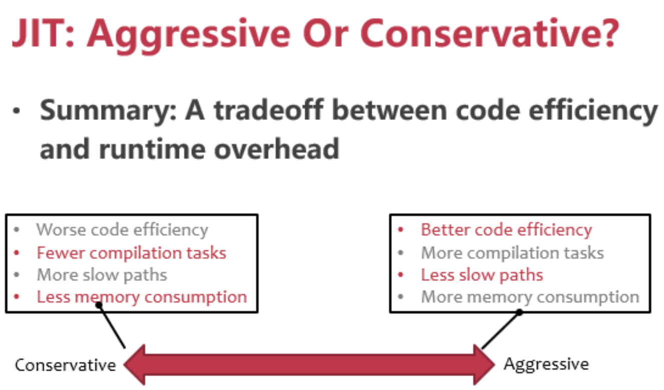


在多态的情况下





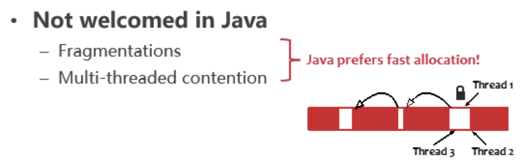
激进和保守的权衡：
代码执行效率和运行时编译开销的权衡



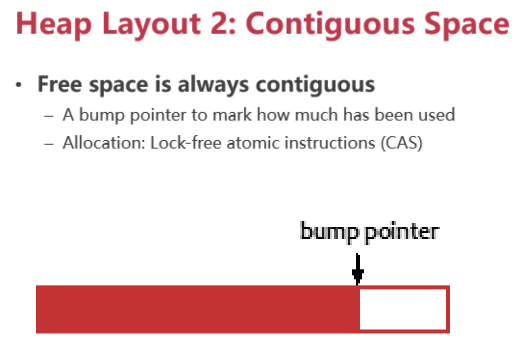
内存管理
数据放在 Java Heap 里

Free Memory Management
GC：识别死对象；管理堆空间。
堆的组织-1：free list
Free 的块组织成链表

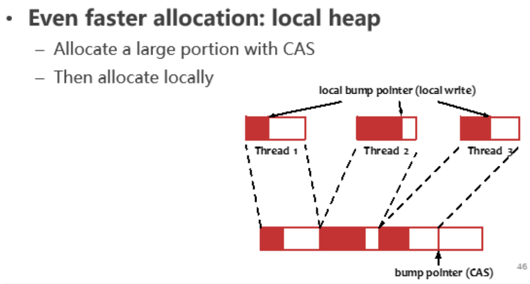
在 Java 中不受欢迎



堆的组织-2：连续空间



为了维持连续空间，需要在垃圾回收之后把 live object 压在一起。
小优化：local heap
每个线程有一个本地堆，省掉 CAS 开销。



GC 算法-1：引用计数
记录有多少指向该对象的引用。引用计数变为 0 时就进行回收。
在 JAVA 中不受欢迎的原因：性能差。和 free list 有较强耦合。不能处理**循环引用**。

GC 算法-2：tracing 追踪
通过图遍历的方法把所有的活对象都找到。
从“root”开始遍历。
Root 包含：线程栈上的对象。全局对象，包含每个类的静态变量等。
可以解决循环引用问题。

Tracing GC 应用比较广泛。

Tracing GC is Popular

- **Can be integrated with different layout**
  - Free-list: Mark-Sweep (Boehm GC)
  - Contiguous: Mark-Copy (PS, G1, Shenandoah...)
- **Can also be used for different purposes**
  - Throughput-oriented: Stop-the-world tracing
  - Latency-oriented: concurrent tracing

吞吐量优先 or 延迟优先 ？

Throughput-oriented GC

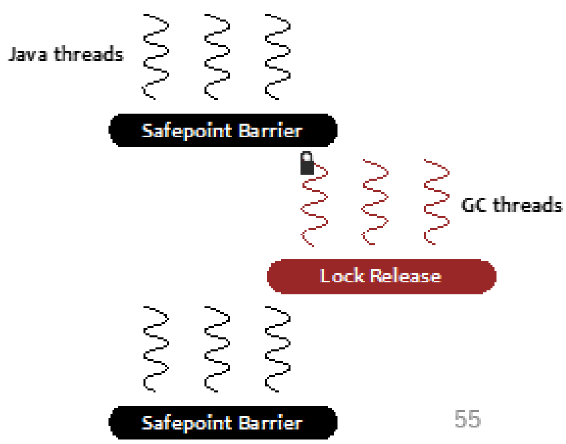
- **Design goal: high GC throughput**
  - Stop-the-world: Java threads must be paused during GC
  - Task-based parallelism: dividing collections into tasks
- **Consider latency together with throughput**
  - Generational

- 在 GC 期间暂停线程的运行，以提高吞吐量。
- 把垃圾回收拆分成小的任务，并行处理。
- 分代。

Stop the world：

JVM leverages *safepoint* to pause all Java threads

– Java threads queue up for a lock held by GC threads



Aiming at high GC throughput

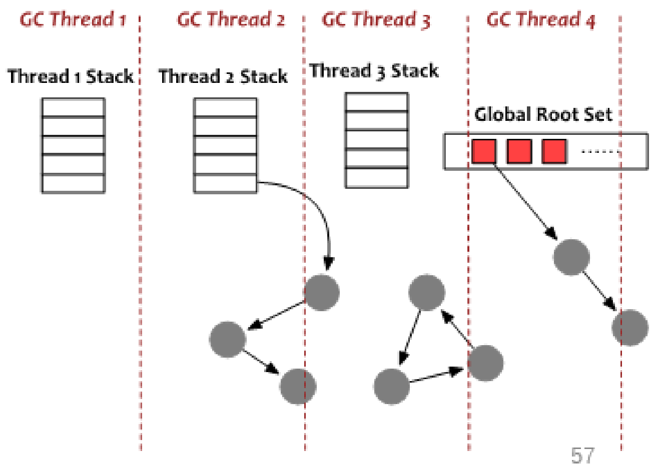
– CPUs are monopolized by GC threads

– No coordination between GC and Java threads

GC 线程可以充分利用 CPU；
无需考虑 GC 对 Java 线程的影响。

基于任务的并行

遍历过程：可以通过划分 roots 来分成多个任务，多个 GC 线程并行执行。



问题：根据起点分配，可能会造成负载不均衡。
可以通过 work-stealing 实现动态平衡。

考虑时延：

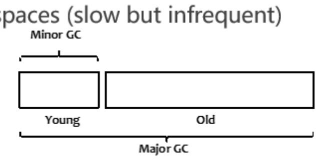
GC 暂停的时间和 live object 的大小正相关。当 live object 特别大时暂停的时间会很长。
因此不能等到完全没有空间了才回收。

Solution：分代

分为新生代、老生代。
刚分配的对象在新生代，存活时间长的对象在老生代。

Solution: Generational GC

- **Dividing Java heap into multiple spaces**
  - In PS it has two spaces: *young* and *old*
  - The size of young gen can be small and fixed
- **GC is also two-fold**
  - Minor GC: only collecting young-space (fast and frequent)
  - Major GC: collecting both spaces (slow but infrequent)



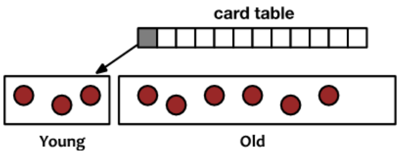
GC 也分为两个过程，对新生代区域的 GC 和对全局的 GC。
基于的假设：大多数对象死的很快。因此对新生代区域的 GC 效率很高。
但是，有些场景下“分代假设”不成立，比如大数据场景 spark Hadoop。对象可能都移到老生代里去了。
分代的问题：跨区的引用
我们只想对新生代区进行遍历、回收。但是有些引用是从老生代指向新生代的。
方法是把新生代中被老生代引用的对象标为 root。
那么如何识别有哪些是跨区的引用呢？

- 扫描整个老生代？（那还分个锤子代）
- 记住所有的引用（开销太大）

方法：card table

Solution: Card Table

- **Dividing old-space into many *regions***
  - Using 1 bit (card) in a table for each region
- **When a cross-space write happens, dirty the card**
  - Scanning dirty cards only during minor GC



是一个折中的办法。

面向时延的 GC

目标是更短的停顿，更小的延迟

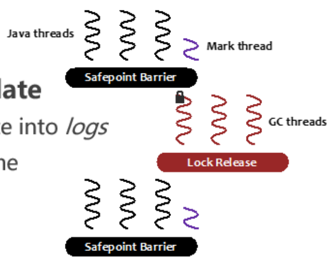
Design goal: shorter pauses/app latency

- Concurrent marking
  - Choose which parts are valuable for collection
- Concurrent collection
  - Collecting when application threads are active

并行标记

Java 线程和标记过程并行，处理并发更新的方法是 Java 线程会将更新内容写入 logs，标记线程会定期消费日志。

- **Handling concurrent update**
  - Java threads will write update into *logs*
  - Marking threads will consume the logs periodically

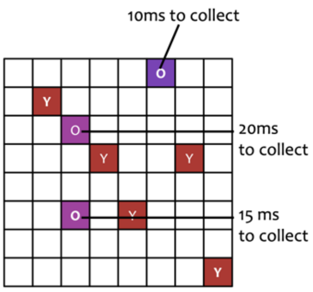


并发标记的好处：

- 标记不需要暂停 Java 线程
- 标记的结果可以指导 GC 线程进行更高效的垃圾回收。

An Example: Region-based Profiling

- **The whole heap is split into many *regions***
- **GC threads estimate per-region collection cost according to concurrent marking**
  - Meeting the “soft limits” from users



“GC pause should <50ms”

可以根据标记的结果预估每个区域垃圾回收的时间。以便满足用户对时延的要求。

并行回收

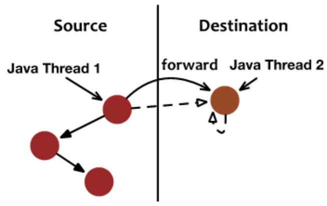
会使得暂停时间非常短。

要解决的主要问题：双拷贝，引用更新。

双拷贝问题：GC 时要进行拷贝，存在一个时间窗口使得两个拷贝同时存在，可能会使得不同的线程指向的是不同的拷贝，此时如果发生更新，会不一致。

解决：间接指针

- **Solution: indirect pointer**
  - Always point to the newest version
  - All read/writes will be forwarded to the newest one

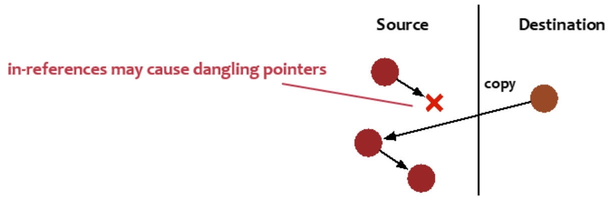


解决：双写，要解决双写的原子性问题。

引用更新问题：

In-Reference Updates

- **Only part of heap space is collected each time**
- **Only out-references are updated**



In-Reference Updates

- **Design 1: Stop-The-World update**
  - Large pause time (violating the goal of Shenandoah)
- **Design 2: memorizing all in-references**
  - Large memory overhead (all references are duplicated)
- **Design 3: lazily updated in next phases**
  - A next marking phase will scan the whole heap

较好的是 lazily update，在下一轮扫全栈的时候才真正回收，在此之前用**间接指针**占位。

总结：

Summary: Throughput vs. Latency

- **PSGC: Stop-The-World**
  - Throughput-oriented
  - Large GC pauses
- **G1GC: adjustable & partially-concurrent**
  - Concurrent marking
  - Controllable GC pauses
- **Shenandoah: mostly-concurrent**
  - Concurrent marking & collection
  - Ultra-low GC pauses
  - Hurting application throughput

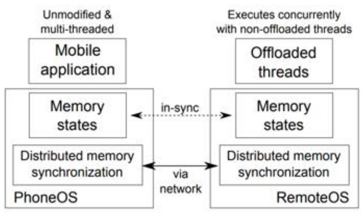
引入 JVM 这一层抽象的**好处**是可以捕捉更多的应用相关的语义。**坏处**是运行时的行为更加复杂（比如 GC）。基于这些好处和坏处，有以下一些工作。

12 年的工作：COMET

The Goal of COMET

- **Transparent offloading**
  - No programming effort, arbitrary apps
- **Fine-grained offloading**
  - Restricting transferred bytes
- **Robust offloading**
  - Resisting network failures

COMET builds a distributed shared memory (DSM) between phones and clouds

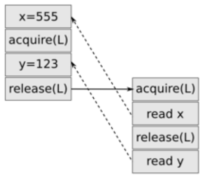


传统 DSM 是基于页的内存管理。

在 JVM 下的好处是可以做**细粒度**的 DSM。

Core Concept: Java Memory Model (JMM)

- **Dictates which writes a read can observe**
- **Specifies 'happens-before' partial order**
  - Accesses in single thread are totally ordered
  - Accesses in different threads are ordered by locks



94

单线程内部的读写操作是按顺序的。

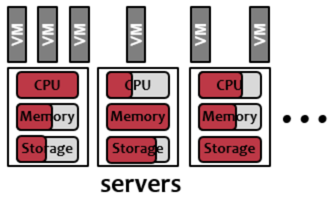
多个线程之间的顺序是锁来保证

- **Used to establish 'happens-before' relation**
- **Synchronizes**
  - Bytecode sources
  - Java thread context (all frames of all threads, including pc/registers/method)
  - Java heap (only dirty fields in tracked set)

关于 VM 带来的问题的工作：Semeru

Resource Disaggregation

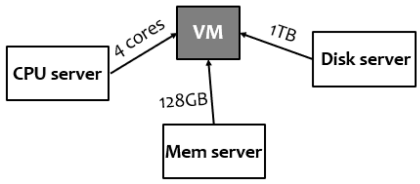
- **Traditional data center: monolithic server model**
  - Each server hosts all types of hardware resources
  - Inducing resource under-utilization



可能因为某一种资源用满了造成其他资源的浪费。

Resource Disaggregation

- **Disaggregated data center: the next generation?**
  - Each server(s) contains different types of resources
  - Better resource utilization

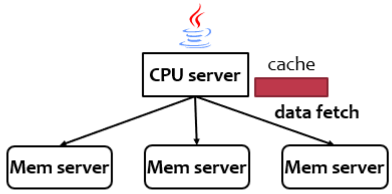


1

一种简单的模型，CPU server + memory server

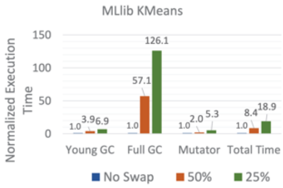
A simplified model: CPU server + memory server

- Applications are running on CPU server
- Data are stored on memory servers
- CPU servers keep a local memory cache (so locality matters)



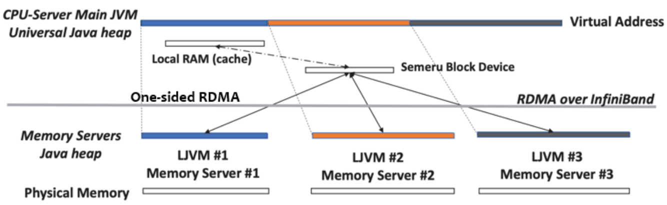
发现资源分离对 JVM 不友好。

- **Workload: *KMeans* in Spark**
  - One CPU server, two memory servers, G1GC
- **Result: 18.9X for 25% cached memory**
  - Full GC: 126.1X
  - Reason: too many data fetching



Semeru Overview

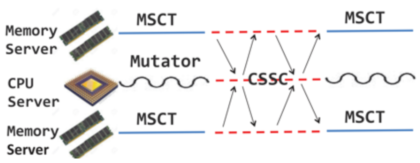
- **A unified Java heap among all servers**
  - Each memory server runs a LJVM



Key Insight: Close-Data GC

- **Let memory servers handle most GC tasks**
- **Now GC is divided into two sub-phases:**
  - Memory server concurrent tracing (MSCT)
  - CPU server STW collection (CSSC)

G1 mixed



108

思路：memory server 处理大多数 GC 工作

CPU server 把 root 发到 memory server

Conclusion

- **Language virtual machines: an "indirection" for portability and safety**
- **A typical example: JVM**
  - Code execution: interpreter & JIT compiler
  - Memory management (GC): basics & modern GC designs
  - JVM in systems: COMET & Semeru