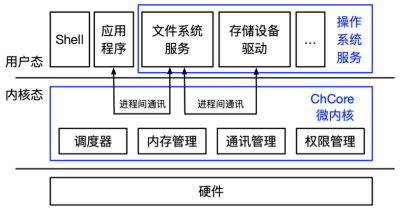
01 ARM

架构 ChCore 架构图

冯·诺依曼架构：CPU <=> Memory



CISC/RISC：定长/变长指令；单一/多种寻址方式；内存操作 Load+Store/mov；指令、处理器简单/复杂；汇编复杂/简单；功耗低/高。

ARMv8 寄存器 31 个 64 位通用寄存器 X0 至 X30、1 个 PC 寄存器、4 个栈寄存器（切换异常等级时时保存栈指针）SP_EL0 至 EL3、3 个异常链接寄存器（保存异常的返回地址）ELR_EL1 至 EL3（无 ELR_EL0因为不存在等级 0 下的异常处理器）、3 个程序状态寄存器（切换异常等级时保存 PSTATE）SPSR_EL1 至 EL3。

x86 寄存器 16 个通用寄存器、1 个PC 寄存器（%rip）、1 个栈寄存器（%rsp）不分异常等级、返回地址直接压栈，没有专用寄存器。

ARMv8 IO MMIO（Memory-mapped I/O），把 I/O 映射到特殊的内存区段，复用内存操作指令来读写。

x86 IO PIO（Port I/O），给 I/O 设备独立的地址空间，使用特殊的指令（in、out）来进行读写。

操作系统启动过程 Bootloader 和 Kernel 分别位于同一个 ELF 文件的 .init 段和 .text 段。

Bootloader 通过特殊指令启动主核，挂起次要核，落回 EL1 级，准备函数栈和异常向量，初始化 UART，初始化页表（将 内核代码映射到低地址段（和物理地址一致位置）和高地址段两份）、开启 MMU、跳转到 kernel main 函数，进入 ChCore。

硬件模拟 CPU：软 Parse 指令，转写成高级函数调用。寄存器：直接开一段内存放寄存器内容。内存：做 GPA 到 HVA 的映射（hmap）；如果 MMU 开启，要先翻译 GVA 到 GPA。硬盘：映射到文件上。VGA：映射到窗口中。时钟信号：使用 Host OS 高级 API。

中断和异常 按下键盘上的一个键，CPU 执行会被打断（是谓中断）。广义定义：中断（Interrupt）外部硬件设备所产生的信号，异步，产生原因和当前执行指令无关。异常（Exception），产生和当前执行或试图执行的指令相关，同步。

ARM 异常 异步异常=广义中断，同步异常=广义异常。

ARM 异步异常 两种。重置，最高级别，由 CPU 上电软件、看门狗触发。中断，由外部信号触发，打断指令的执行，由 GIC 管理发往 CPU。

ARM 同步异常 两种。中止（Abort），如指令失败、访存失败。由指令产生（EGI），如 svc（用户程序➡操作系统）、hvc（Guest 系统➡虚拟机管理器）、smc（Normal World ⬌ Secure World）。

ARM 中断处理 经过了 GIC 发往 CPU，由 c_handle_entry 处理，通过中断向量查阅中断向量表，并跳转到对应的中断处理器处。

中断处理函数没有上下文。可能在任何位置触发，不依赖于任何上下文。

中断处理函数不能睡眠、调用可能阻塞的任务、不能主动交出控制权。

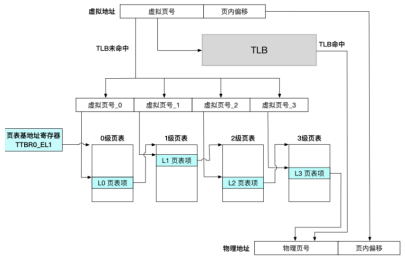
（确认异常类型的方法）

系统调用 同步异常的一种。用户态程序向内核请求做高权限任务的口信。在上表中属于 EL0_SYNC。完成处理後使用 svc #0 返回用户态。

02 内存

Protection Key 原始内存隔离策略，2K 内存块携带 4B Key。

虚拟内存机制 所有软件代码均用虚拟内存寻址，离开 CPU 後由 MMU 翻译。ARM 应用和内核用不同页表，首地址分别在寄存器 TTBR0_EL1 和 TTBR1_EL1 中。SCTLR_EL1 第 0 位置 1 即可使能这两级页表。



虚拟内存地址 一般：从高到低，16 位 OS/User 标识位、4 级 9 位共 36 位页表索引，12 位页内偏移。

TLB 缓存 的是 VPN（虚拟页号）和 PPN（物理页号）之间的映射关系。切换进程时需要清除遗留 TLB 项。

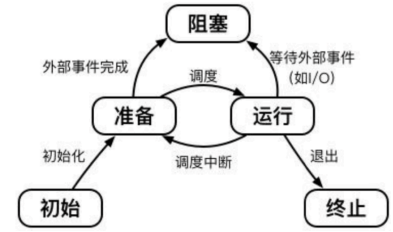
TLB 优化 PCID：每个进程的唯一 ID 用于区分 TLB 项，无需刷新 TLB。ASID 类似。

全局 TLB 设定 notGlobal 位，可以选择此条目的 TLB 缓存是否对所有进程可见。

大页 省略掉页表靠後数级的查表，直接得到 Huge Page No. ARM 支持的页尺寸有 4K（标准 9 位四级页表，低 12 位对齐）、16K（11 位三级页表，低 14 位对齐）、64K（13 位二级页表，低 16 位对齐）。除开高 16 位，虚拟地址有效位数只能是 36、39、42、47、48。

03 进程和线程和纤程

状态机 状态切换时要保留其上下文。

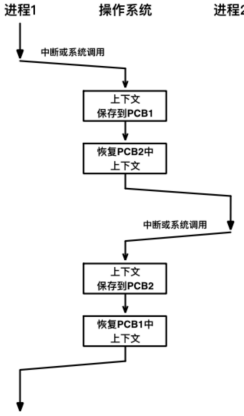


切换过程 中断或系统调用触发，内核保存恢复上下文并交予控制权。

PCB 即进程控制块，包含进程的标识符、内存、句柄、状态等。

Linux fork 拷贝调用进程作为其子进程，父进程返回子进程 ID，子进程返回 0。不共享 ID、内存，但共享 fork 之前的文件句柄、I/O 设备，且他们的引用计数增加。

Linux exec 读取传入程序，并重置调用者内存空间，从头开始执行传入的程序。



写时复制 要进行大规模复制时，只拷贝内存映射。读，直接访问共享的内存；写，触发拷贝。

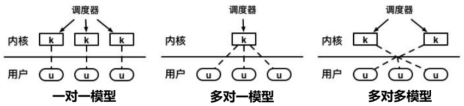
fork + exec 也存在无意义的内存复制，用写时复制就可以减少无意义的拷贝。

Linux vfork fork，但是共享内存空间。不安全，且相比 CoW fork 提升不明显。别用。

posix_spawn 经过优化的 fork + exec。

clone 类似 fork，选择性拷贝内存给子进程。

线程 相比进程，没有独立内存空间，上下文只包括寄存器（和栈。本质上栈也是大家共享的，只是通过不同的 SP 访问不同部分。）线程调度仍然需要借助操作系统。



ChCore 中每创建一个用户态线程，都会新起一个内核线程来管理，属「一对一模型」。

线程 PCB 中只有寄存器，无需实际栈内容。

纤程 用户态线程，无需进入内核调度的线程，也无需创建内核管理线程。上下文只有「寄存器」，保存恢复在用户态就可以做。

04 调度机制

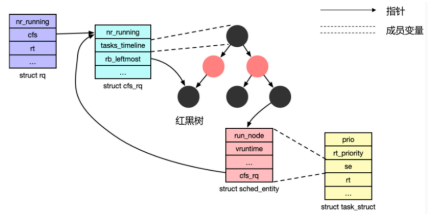
不区分进程线程地说，主流的调度策略包括

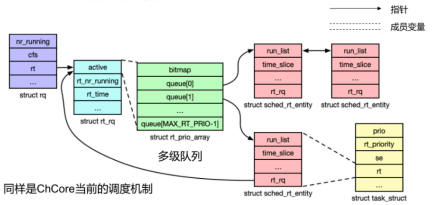
CFS Run Queue 调度示意图

RT Run Queue 调度示意图

先到先得 先来的先服务，简单直观公平，不会饿死。但平均周转时间、响应时间太长。

最短任务先 每完成一个任务後，取待执行的耗时最短任务执行。不公平，可能饿死；平均响应时间过长。





Round Robin 调度 每隔一段时间片轮转地调度一次未执行完成的任务。公平，响应时间短。但周转时间长，频繁调度浪费时间。

优先级调度 类似于 RR 调度，只是给不同优先级进程不同的时间片长度和调度频次。

彩票调度 任务按权重分不同宽度区间，生成随机数，观察落点来决定被调度者。公平。

步长调度 消除彩票调度的随机性，用非随机均匀数列代替随机数，更加公平。

最早 DDL 调度 对于有 DDL 的任务，总是先做 DDL 靠前的任务，依次完成。

多米诺效应 假如存在一个 DDL 紧的耗时任务，尝试完成它不仅无法成功，还会导致其他任务接连失败。因此在调度时还需判断可行性，以避免此情况。

调度术语

周转时间 = 任务完成时刻减去任务提交时刻

平均周转时间 = 所有周转时间和除以任务数

等待时间 = 周转时间减去实际运行时间

平均等待时间 = 所有等待时间和除以任务数

响应时间 = 首次调度时刻减去任务提交时刻

注意「时间」和「时刻」的区分。

05 IPC

意义 用共享库，减少重复实现；提供公有信息，减少计算浪费。

定义 两个或多个不同的进程通过通过内核或其他共享资源进行通信，借此传递控制信息或数据。

同步异步 同步指消息必须被接收後，发送者才会停止阻塞。异步指消息一经发出，发送者就停止阻塞。

超时同步 尝试同步通信，但在等待超过一定时间之后就放弃发送，等待下次再发。

缓冲区 零容量（无缓冲区）：只能支持同步消息通信；发送者肯定需要等待接收者接收消息。有限容量（N 缓冲区）：可以支持 N 条异步消息，超过此数，则回落为同步式。发送者大部分时候无需等待。无限容量：可以缓冲任意条消息；发送者永远无需等待。

Unix 管道 例如 ls | grep，即创建 ls 进程及 grep 进程，并在他们之间建立通信管道。只支持单向通信，通信内容无类型。本质上是一个有限容量的 N 缓冲区实现，大小限定为 PIPESIZE。发送者会在缓冲区满时阻塞。接收者会在缓冲区空时阻塞。

sleep 和 wakeup 调用 sleep(chan) 会导致当前进程阻塞，并立即放弃被调度权；调用 wakeup(chan) 会使得所有之前调用过 sleep(chan) 且当前仍然阻塞着的进程变得「可调度」。

管道就利用了这两个函数实现。

消息队列 链表实现，无限容量缓冲区；因为只要能创建消息，就一定能将其加入链表。完全异步通信，发送者无需等待。

LRPC 不经过内核调度的通信。不做任何内核调度，仅把地址空间、权限表等内容转移。上下文仅有「返回地址」和「栈指针」。

在 LRPC 被创建的时候，提前分配一个「参数栈」，同时被映射到双方的内存空间。

这样，要发送一条消息，直接把参数放到这个栈上就可以了，免去内核的数据拷贝（共享参数栈）。同时，也可以约定某些寄存器用来传参，在切换时不清除这部分寄存器的内容（共享寄存器）。

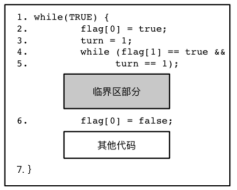
06 同步问题

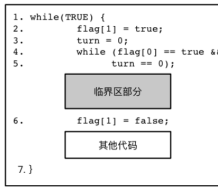
生产者和消费者 先放/取数据再更新索引，只有一生产者和一消费者时不会有竞争问题。

有多个生产者或多个消费者时，可能出现「一地多写」、「一值多读」的问题。

需求 互斥访问：要避免「一空多放」、「一物多拿」这种情况，不能让两个进程同时进入临界区执行。有限等待：一个进程申请进入临界区後，必须在有限时间内获得许可进入，而不能无限等待下去。空闲让进：当没有进程在临界区中时，就应该允许一个申请者进入临界区。

Peterson 锁 软件实现。flag[i]：进程 i 是否有意进入临界区。turn：允许进入临界区进程。





CAS 和 FAA

```
CAS(ptr, expected, new) -> v:
if (*ptr = expected) then (*ptr := new, return expected) or return *ptr
FAA(ptr, add) -> v:
tmp = *ptr, *ptr += add, return tmp
```

只要能把「取值」和「放值」两个操作组合成一个原子操作，就能实现锁。

排号锁

```
lock:
    my_ticket = FAA(lock.next, 1)
    # 拿号
    while (lock.owner != my_ticket): # 等号
        pass
    # lock obtained!

unlock:
    lock.owner += 1     # 叫号
    # lock released!
```

基于 fetch_and_add 实现。

读写锁 模型：同时可以有「一个写者」或「多个读者」进入临界区。

问题：有读者正在临界区中读，有写者在等待读者读完。此时，新读者能否进入？

不可以（偏向写者的读写锁。有写者等待时，不允许新的读者进入。防止写者饿死）、可以（偏向读者的读写锁。有写者等待时，也允许新读者进入。并行性更好。）

RCU 机制 硬件上支持 128B 的原子操作；操作链表来更新数据流，在写的同时允许读。
代价就是，允许一部分读者读到旧数据，即旧数据不可以被立即销毁，需要保留一定的宽限期。
在「链表更新操作」之前到来的所有读者都退出之后，就可以回收旧数据了。

死锁 四个必要条件：有互斥访问、有「持有并等待」、资源不可被抢占、有循环等待。

乐观解决方案 假设不会死锁，在运行时检测并恢复：生成等待图，查找其中的环。在找到环后，1）把环中所有进程杀死，或者 2）逐个杀死其中的进程，直到环消失。

悲观解决方案 在运行时避免死锁的产生。
1）避免互斥访问、2）不允许持有并等待；一开始就把所有的资源申请完毕，如果其中一个失败就全部放弃。

老大哥解决方案 进程获取资源都先过一遍 OS，沙盒判断一下是否会演变成死锁。

活锁 多方反复同时尝试申请资源，导致任何人都申请不了全部资源。可能自行恢复。

07 文件系统

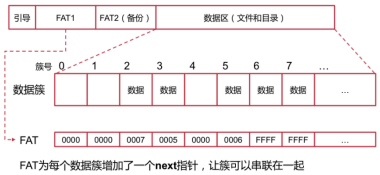
inode 文件系统 超级块管理 inode 和数据块的分配，inode 中直接间接指向数据块。目录的内容是一组目录项（记录 inode、子文件（或目录）名）的特殊文件。文件/目录的名字存放在其上层目录中。所以根目录无名。

软链接 即符号连接。特殊的文件，其内容是一个路径。访问时会跳到目标去。类似于快捷方式（Shortcuts）。

硬链接 不是单独的「文件」，而是同一个 inode 出现在多个目录项中，引用计数多次。

区段 大文件采用 inode 组织太散，不方便读取。找到一块连续区段保存文件，只记录起始和结束地址，减少元数据的数量。

FAT 文件系统 以簇为基本单位。数据连续存储，每个块对应一个 FAT 表条目。



FAT为每个数据簇增加了一个**next**指针，让簇可以串联在一起

VFS 文件系统的抽象，不同的文件系统可以协同工作。不同的文件系统都有相同的根结点（如 Ext4 的 /）。而 VFS 树的根结点只能有一个。因此系统在构造 VFS 的时候，会选择一个主文件系统构造根结点，将其他文件系统挂在特殊的子节点处（如 /dev/）。遍历文件树时，假如遇到了这个特殊的挂载点，就切换文件系统。

Flash FS 专用于闪存盘的文件系统。闪存每个块都有写入/擦除寿命。不可以把高频读写的数据集中存储。另外，尽可能少写数据。

FTL 层 逻辑地址到物理地址的转换层。任意逻辑地址都被 FTL 平摊到物理地址上。
为避免高强度读写同一块，FTL 会定期搬移、重映射数据，可以解决超级块磨损的问题。

递归更新问题 更新一个文件需要修改多重数据结构，放大了数据修改量。

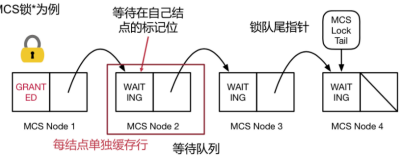
NAT 结构 硬件直接提供 Node 到块的映射，无需手写元数据。

08 多核心

Amdahl 法则 核心数为 s，可并行的代码占比为 p，则加速倍数是 $S = \frac{1}{(1-p)+\frac{p}{s}}$。无限增加核心数的理论上限是 $\frac{1}{1-p}$。

可扩展性断崖 核心数增加到一定程度时，性能大幅跳水，甚至差于少核心数的情况。最常见、最主要的原因是「实现不佳的锁」。

MCS 锁 尽量地避免在关键路径上，对单一缓存行的高度竞争（这就是扩展性断崖的根源：保证正确性所带来的开销超过了多核心并行带来的优势）。



MCS 锁类似于排号锁，只不过用链表实现，结点放在不同缓存行中，减少竞争。

一致性
严格一致性（对一个地址的任意的读操作都能读到这个地址最近一次写的数据；访存操作顺序与全局时钟的顺序完全一致）
顺序一致性（不要求操作按照真实发生的时间顺序全局可见，但执行结果必须与一个全局的顺序执行一致，且这个全局顺序中每一个核心的读写操作与其程序顺序一致）
TSO 一致性（针对不同地址的「读 + 读」、「读 + 写」、「写 + 写」顺序都能得到保证；「写 + 读」的顺序不保证）
弱序一致性（不保证任何对不同的地址的读写操作顺序）

09 崩溃

日志 在进行修改之前，先把修改记录到日志中；所有要进行的修改记录完毕后，提交日志；此后，再进行修改；修改完成后，删除日志。

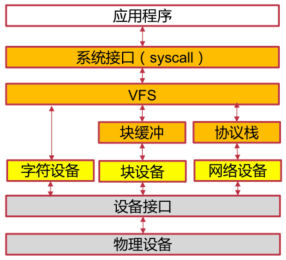
崩溃 在「日志提交」之前崩溃，修改不会发生，文件不会被创建。在「日志提交」后崩溃，恢复时会按日志重做一次，文件创建成功。提交是区分文件创建是否成功的标志。

日志模式 Writeback Mode，日志只记录元数据。能够恢复元数据的情况下，不一定恢复数据。不能保证数据比元数据先落盘。
Ordered Mode，日志只记录元数据，但是保证在此之前将数据块写入磁盘。数据先落盘，元数据随后到。一致性不能保证，但能恢复得到数据。
Journal Mode，日志中同时记录元数据和数据。重放日志可以把所有的数据和元数据带到一致性的状态。很慢很慢。

10 设备

Linux 抽象 字符设备（Char），如 LED、键盘、串口。抽象成文件。
块设备（Block），如 Flash 闪存盘。抽象成文件，或者 mmap 的一块内存。

网络设备（Network），如网卡。抽象成套接字（Socket）。



中断 ARM 中断分为 IRQ（普通中断）、FIQ（高级别中断，一次仅有一个）、System Error（Abort 带来的）。其中 IRQ、FIQ 是由 GIC 管理的中断，连接到 CPU 的不同针脚。

GIC 中断控制器，会将当前最高优先级的中断转发给对应的 CPU。同时只能有一个 FIQ 和若干个 IRQ 能进入 GIC。IRQ 内分不同的优先级，而 FIQ 的优先级高于任何 IRQ。GIC 会在合适的时候把最高优先级的异常推给 CPU，有时会打断正在处理的中断，如果新中断的级别高于当前处理的等级。

中断处理过程 外设发起一个中断、Distributor 仲裁并转发给 CPU。CPU 将中断发给核心，核心读取 GICC_IAR 确认中断并处理。处理完成后写 GICC_EOIR 重置优先级，最后写 GICC_DIR 来消除该中断（分两步）。

不能睡 中断处理函数不可以拿锁，不可以做阻塞性的事情，不可以 Sleep（Sleep 也是靠中断唤醒的）。因为中断处理函数不会被 sched()，这样的行为就会造成内核挂起。

上下半部 中断处理函数要尽量短小精悍，避免长时间占用。解决方案是把一个中断处理的任务分成两部分。紧要的事情放在 Top Half 做，也就是仅仅把请求放入队列，然后将其他的事情推迟到 Bottom Half 处理。
因此中断处理的过程可以大概是：

```
def handler():
    save_regs() # 保存寄存器
    send_request(unique_id) # 请求入队列
    restore_regs() # 恢复寄存器

    STOP_BLOCKING()
    # 不再阻塞低等级的中断，允许打断
    postponed_stuff()
    # 反正现在高等中断已经进入队列了
    # 现在就可以做点无关紧要的事情
    # 这时候就算是低等级的中断来打断也 OK

    REVOKE_INTERRUPT()
    # 中断彻底处理完毕，归还中断号
```

11 网络

驱动模型 应用层 / 传输层 / 网络层 / 数据链路层 / 物理层。分别对应「网络请求进程 / TCP、UDP 协议、端口 / IP 协议、主机地址 / 网络设备驱动 / 网络设备」。

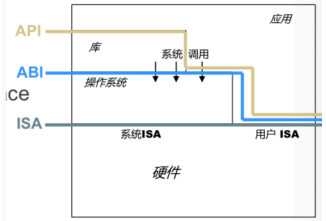
中断合并 网卡收到数据包的频率极高，因此先放入缓冲区；累积到一定数量再打包发给 CPU。吞吐量更高了，延迟也高了。

Linux 吃包过程 1、网卡收到数据包（DMA 将数据帧传送到内核内存中的 rx_ring 中）；2、网卡触发一个硬件中断；3、CPU 收到中断（先进行 Top Half 操作，分配好 sk_buff 数据结构，管理 rx_ring 中的数据包。（skb 本身不包含数据，只是指向 rx_ring 内的数据而已）再将处理请求入队）；4、之后是推迟过后的 Bottom Half（根据报文类型递交给对应协议进行处理，调用 handler 处理 skb 包）；5、进入 IP 层（调用 ip_rcv() 函数，通过 ip_router_input 进行路由处理）；6、进入传输层（tcp_v4_rcv()，检查 TCP Header，调用 __tcp_v4_lookup() 查找数据包对应的 Open Socket，调用 tcp_prequeue() 把 TCP Payload 丢到用户空间，放入 Socket 接收队列）；7、进入应用层（socket 被唤醒之后通过系统调用，通过 tcp_recvmsg() 从 Socket 接受队列中获取数据，用户态调用 read 或 recvfrom，拿到数据）。

12 虚拟化

OS 架构 ISA，用来控制 CPU 的指令的集合。软件和硬件的交汇处。
ISA 中的一部分只能由系统专用（如 msr 指令），用户无法使用。这就把 ISA 分成了两部分。
ABI，将用户态和内核态区分开来，通过系统调用连接。
API，最接近应用的一个层级，不涉及操作系统底层调用的库，例如 clib。
虚拟机管理器需要直接替代 Guest OS 的硬件，且自己也需要运行在硬件之上。

虚拟化的追求 为虚拟机内程序提供和物理机上一致的接口，性能仅仅比无虚拟化时略差一点，虚拟机监控器控制所有的物理资源。



Type I 虚拟机 直接运行在硬件之上，充当操作系统的角色。仅仅实现调度、内存管理、驱动等功能。其余功能都由 Guest OS 直接跟硬件交流。性能损失少。

Type II 虚拟机 把虚拟机当作应用运行，尽量复用 Host OS 的功能。性能损失多。

执行指令的方式 解释执行（软件一条条模拟虚拟机指令，不区分指令类别。虚拟机指令并非真的在执行，类似模拟器。）、二进制翻译（先把要执行的指令批量翻译成真实指令并进行缓存。效率比较高。但本质上执行的仍然不是虚拟机指令。）、半虚拟化（VMM 提供接口给虚拟机（Hyper Call），把 Guest OS 的源码修改掉，把系统调用的部分改成 Hyper Call，普通指令就直接执行。）、硬件虚拟化。

硬件虚拟化 直接由硬件提供 VM 的支持。
VMM 加载 VM 的状态（如寄存器）
VMM 使用 eret 指令进入 VM 执行
VM 内执行到敏感指令或收到中断时，控制权回到 VMM，调用 VMM 记录在 VBAR_EL2 中的处理函数；VMM 运行在高于 EL0 和 EL1 的特权等级 EL2 上。

内存虚拟化 为虚拟机提供虚拟的物理地址空间（同样从 0 开始增长的），隔离虚拟机之间的地址空间，不让 VM 之间互相访存。

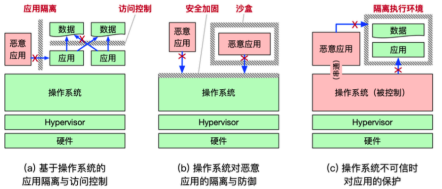
三种地址 客户虚拟地址（GVA），虚拟机内进程使用的虚拟地址；客户物理地址（GPA），虚拟机内使用的假物理地址；主机虚拟地址（HVA）、主机物理地址（HPA）。
VMM 需要考虑如何取得 HPA。创建虚拟机的时，会在 pmap 中记录 GPA 到 HVA 的映射关系。因此总是可以采用 GVA => GPA => HVA => HPA 的翻译过程，就是慢了点。

影子页表策略 影子页表：**GVA => HPA**。各个虚拟机中的每个进程都有自己的影子页表。首先，拿着 GVA 去查阅影子页表。如果没查到，那么就回到 GVA => GPA => HVA => HPA 的路径，并且把结果填入影子页表中。这样影子页表慢慢就能够作为 GVA => HPA 的 Cache 存在了。注意，假如 Guest 更改了 GVA => GPA 的映射，毫无疑问 VMM 应该随之更新影子页表。因此 VMM 会将所有的 Guest 页表标记为写保护，在捕获到修改之后更新影子页表以保证不会从影子页表中读出错误的值。简而言之，影子页表可以找不到条目，但是不可以找到个错的。

第二阶段页表策略 所谓「第一阶段页表」，指的是 GVA => GPA 的 Guest 页表。而「第二阶段页表」，指的是 **GPA => HPA** 的翻译过程。为了避免和 HVA => HPA 页表混淆，这个页表头被放在 VTTBR_EL2 寄存器里。
「第二阶段页表」只需要为每个 Guest OS 创建一个就行了。虽然翻译步骤 GVA => GPA => HPA 相比影子页表多一步，但是因为 GPA => HPA 也可以被 TLB 缓存，因此有缓存时性能不差很多。

IO 虚拟化 设备模拟法：VMM 模拟出一个 Proxy 外设。半虚拟化法，用特殊的驱动程序，不调用 System Call 而是 Hyper Call，共享内存传递指令和命令。设备直通法：设备相关的事情不经过 VMM，直接跟硬件交流。

13 安全性



(a) 基于操作系统的应用隔离与访问控制　(b) 操作系统对恶意应用的隔离与防御　(c) 操作系统不可信时对应用的保护

概念 可信计算基：为实现计算机系统安全保护的所有安全保护机制的集合。包括软件、硬件和固件。攻击面：一个组件被其他组件攻击的所有方法的集合；可能来自上层、同层和底层。防御纵深：为系统设置多道防线，为防御增加的冗余。

认证和授权 认证：确定发起请求实体的身份；授权：确认实体所拥有的资源访问权限。

隐秘信道与侧信道 隐秘信道：无法直接通信的两方，通过不被用于通信的机制传输数据。侧信道：通过被攻击者无意泄露的信息窃取数据。

硬件飞地 在 OS 都不可信的情况下，必须有一块净土（飞地）作为可信基来保证安全。Enclave 仅仅信任 CPU 的计算逻辑及其 Cache。

Intel SGX 以 CPU 外壳为界；以外的信息都是秘文，以内的信息（包括 Cache 和寄存器）都是明文。数据进出 CPU 时进行加密解密操作。

Merkle 哈希树 对内存中的数据计算一级 Hash，一级 Hash 计算出二级 Hash，构成树状结构。终极 Hash 存在 CPU 中。在读取内存时，一级一级地求 Hash 验证完整性。把最高级的 Hash 信息存在 CPU 里，以避免回放攻击。