

目录

2022/2/15.....	7
2022/2/17(NO).....	13
2022/2/22.....	14
异常向量表.....	21
系统调用.....	21
内核的传参.....	23
SMAP 机制（用户态内存不能被内核态 CPU 访问）.....	25
VDSO（Virtual Dynamic Shared Object）.....	26
2021/2/24.....	27
FLEX-SC.....	28
计算机启动.....	29
页表初始化.....	35
el1_mmu_activate.....	37
异常向量表初始化.....	39
2022/3/1.....	40
内核启动前： BIOS 的作用.....	40
MBR（master boot record）.....	41
虚拟内存.....	44
内存隔离： Protection Key.....	45
虚拟地址翻译过程：分段机制.....	47
虚拟地址翻译过程：分页机制.....	48
ARM64 的页表格式.....	49
TLB：页表的 CACHE.....	52
ASID 来降低 TLB 刷新导致的开销.....	53
虚拟内存：段和 VMA.....	54
VMA 是如何添加的.....	55
2022/3/3.....	56
延迟映射/按需映射.....	58
换页机制.....	61
OS 内存管理中的更多机制.....	62
内存压缩.....	63
大页（Huge Page）.....	64
物理内存管理.....	67
直接映射机制.....	69
伙伴系统：页粒度内存管理.....	71
SLAB/SLUB/SLOB：细粒度内存管理.....	73
2022/3/8.....	74
换页策略（Page Swap）.....	77
Thrashing Problem.....	80
工作集模型.....	81
进程.....	83
进程控制块（PCB）.....	83

线程.....	86
TCB.....	89
上下文切换.....	91
2022/3/11.....	96
处理器调度.....	96
经典调度.....	101
优先级调度.....	103
多级队列.....	104
优先级反转.....	105
多级反馈队列.....	107
公平共享调度.....	112
实时调度.....	115
多核调度策略.....	117
2022/3/15.....	120
进程间通信.....	120
进程间通信（IPC）.....	123
管道：文件接口的 IPC.....	123
共享内存（内存接口的 IPC）.....	127
消息传递（Message Passing）.....	128
消息队列.....	134
轻量级远程方法调用（LRPC）.....	136
ChCore 进程通信.....	140
2022/3/17.....	142
同步原语.....	142
生产者消费者问题、竞争条件、临界区问题.....	144
竞争条件.....	146
基本同步原语：软件实现与硬件实现.....	147
Compare And Swap（CAS）操作.....	149
LL/SC 机制（Load-linked & Store-conditional）.....	150
写硬件原子操作：使用硬件辅助.....	152
锁的实现.....	152
自旋锁（Spinlock）.....	152
排号锁（Ticket Lock）.....	153
条件变量.....	155
条件变量.....	156
信号量.....	157
2022/3/22.....	162
同步原语对比.....	164
RCU：更高效的读写互斥.....	168
死锁.....	172
2022/3/24.....	175
银行家算法.....	175
多核和同步.....	177
Amdahl's Law.....	179

多核环境下的缓存.....	181
缓存一致性：全局目录项.....	183
可扩展性断崖.....	186
解决可扩展性问题：回退锁.....	188
解决可扩展性问题：MCS 锁.....	188
非一致内存访问（NUMA）.....	194
2022/3/29.....	197
同步原语：进阶与案例.....	197
cohort 锁.....	198
代理锁.....	200
读写锁的可扩展性.....	203
大读者锁.....	204
PRWLock.....	205
Linux 的同步原语：QSpinLock.....	206
Linux 的同步原语：FUTEX.....	208
内存模型和同步.....	209
2022/3/31.....	214
不同的文件系统.....	214
EXT2 文件系统.....	218
ChCore 中的文件系统.....	221
基于 Table 的文件系统：FAT.....	222
exFAT.....	226
基于数据库的文件系统：NTFS.....	227
2022/4/7.....	230
文件系统结构.....	230
虚拟文件系统（VFS）.....	232
文件系统高级功能.....	239
克隆.....	239
快照（Snapshot）.....	240
稀疏文件.....	241
文件系统的多种形式.....	242
GIT.....	242
SQLite.....	244
FUSE：用户态文件系统框架.....	245
文件系统崩溃一致性.....	247
2022/4/12.....	251
日志（Journaling）.....	252
利用内存中的页缓存提高日志性能.....	253
JBD2.....	255
写时复制（Copy-on-Write）.....	258
B-tree FS（写时复制文件系统）.....	260
Soft Updates.....	260
依赖追踪.....	263
撤销和重做.....	264

2022/4/14.....	266
日志文件系统（LFS）	268
段（Segment）	272
前滚（roll-forward）	275
F2FS（Flash 友好的文件系统）	277
Flash（闪存盘）	277
FTL(Flash Translation Layer).....	279
NAT.....	281
2022/4/19.....	284
瓦式磁盘.....	286
非易失性内存（Non-volatile Memory, NVM）	294
非易失性双列直插式内存模块(Non-volatile Dual In-line Memory Module, NVDIMM).....	295
Intel Optane DC Persistent Memory.....	295
NVM 上的写时复制.....	298
非易失性内存文件系统（Non-volatile Memory File System, NVMF）	299
PMFS.....	300
防止 NVM 上的 wild write.....	301
2022/4/21.....	303
设备管理.....	303
设备和 CPU 是怎么连接的.....	307
CPU 和设备是怎么交互的.....	309
GIC（ARM 中断控制器）	315
设备驱动.....	318
I/O 子系统.....	321
设备的缓存管理.....	324
I/O 库.....	328
2022/4/24.....	329
设备管理和网络.....	329
OS 设备管理第一类方式：内核直管.....	331
OS 设备管理第二类方式：设备驱动.....	333
OS 设备管理第三类方式：用户态库.....	334
Linux 的上下半部.....	335
内核线程（Kernel Threads）	336
操作系统的网络.....	337
Linux 的收包过程.....	338
用户态协议栈：DPDK.....	343
不同架构对比.....	346
2022/4/26.....	347
什么是系统虚拟化？	350
Type-1 虚拟机监控器.....	353
Type-2 虚拟机监控器.....	353
如何实现系统虚拟化？	354
Trap & Emulate.....	354
CPU 虚拟化.....	355

处理敏感指令的方法 1: 解释执行.....	357
处理敏感指令的方法 2: 二进制翻译.....	359
处理敏感指令的方法 3: 半虚拟化.....	362
处理敏感指令的方法 4: 硬件虚拟化.....	362
2022/4/28.....	363
Intel VT-x.....	363
Intel VT- VMCS.....	365
VT-x 的执行过程.....	366
ARM 的虚拟化技术.....	368
2022/5/5.....	371
第一版: 虚拟机只在内核态运行简单代码.....	371
第二版: 虚拟机内部支持时钟中断.....	372
第三版: 虚拟机内支持运行单一用户态线程.....	373
第四版: 虚拟机内部支持多个用户态线程.....	374
第五版: 支持多个虚拟机间的分时复用.....	375
第六版: VMM 支持多个物理 CPU.....	376
案例: QEMU/KVM.....	379
内存虚拟化.....	383
客户物理地址 (GPA)	384
内存虚拟化方案 1: 影子页表.....	385
内存虚拟化方案 2: 直接页表 (Direct Paging)	389
2022/5/10.....	389
内存虚拟化方案 3: 硬件虚拟化.....	392
第二阶段页表.....	393
内存气球机制.....	398
I/O 虚拟化.....	398
SR-IOV (Single Root I/O Virtualization)	407
2022/5/12.....	410
轻量级隔离.....	410
CHROOT.....	412
Linux Container.....	414
Mount Namespace.....	415
IPC Namespace.....	416
Network Namespace.....	417
PID Namespace.....	418
User Namespace.....	419
其他 Namespace.....	421
SFI: 软件错误隔离.....	421
SFI 的 Domain.....	422
CFI: 控制流完整性保护 (Control Flow Integrity)	423
XFI: SFI+CFI.....	426
Native Client.....	427
2022/5/17.....	428
操作系统安全机制.....	428

基于硬件的进程内隔离: ENCLAVE.....	428
Enclave 和进程之间的关系.....	430
性能隔离.....	430
操作系统的安全服务.....	434
OS 安全的三个层次.....	435
访问控制.....	436
访问控制列表 (ACL, Access Control List)	438
POSIX 的文件权限.....	438
基于角色的访问控制 (RBAC)	439
setuid 机制.....	440
Capability 权限控制.....	441
Linux 的 Capability.....	442
2022/5/19.....	442
操作系统攻防.....	442
DAC 和 MAC.....	443
Bell-LaPadula 模型.....	444
SELinux.....	444
操作系统的漏洞.....	447
操作系统内核攻防.....	448
Return-to-user 攻击 (ret2usr)	449
Rootkit.....	450
KASLR: 内核地址布局随机化.....	450
侧信道与隐秘信道.....	451
缓存信道.....	452
常量时间算法.....	454
2022/5/24.....	455
硬件辅助系统安全.....	455
不经意随机访问内存 (ORAM)	455
案例: MELTDOWN.....	455
KPTI: Linux 的 Meltdown 漏洞防御机制.....	459
当操作系统不再可信.....	461
一种新的威胁模型: 安全处理器.....	462
ENCLAVE 案例分析: Intel SGX.....	464
硬件内存加密.....	465
2022/5/26.....	467
其他平台的 enclave 技术.....	467
AMD SEV.....	469
小结.....	470
调试和测试.....	471
调试器的基本原理.....	471
案例: QEMU 的 GDB 支持.....	475
性能调试.....	476
控制流追踪.....	480
测试的基本原则和方法.....	484

2022/6/2 期末复习.....	492
进程和调度.....	500
IPC.....	506
迁移线程.....	508
同步.....	508
MCS 锁（重要）	512
cohort 锁.....	514
文件系统和崩溃一致性.....	515
崩溃一致性.....	518
设备与 I/O.....	521
系统虚拟化.....	523
CPU 虚拟化.....	525
内存虚拟化.....	528
操作系统安全.....	532

2022/2/15

我们这学期开始给大家介绍操作系统。我们希望在这节课的过程中，不仅仅是介绍经典操作系统，也希望大家了解产业界中的趋势和未来的变化。我们今天主要的内容分为这几部分。

我们先看一些 OS 的简单历史，在讲 OS 的时候，一定是计算机的 OS。当时没有操作系统，所有操作都要靠人去操作。后来就有了批处理操作系统，不用人一次次去操控。把这个过程自动化之后，计算机可以把一个计算任务做完以后自动导入下一个计算任务。计算任务的输入形式通常是以打孔纸带的形式。

批处理操作系统 : GM-NAA I/O



- Robert L. Patrick 和 Owen Mock 于 1956 年建设
- 运行在 IBM 704 上
- 主要功能：批处理运行任务

操作系统主要控制的是纸带。

通用操作系统 : OS/360



- IBM System/360 OS , 1964
- 首个通用操作系统，首次将操作系统与计算机分离
- 架构师：Gene Amdahl (Amdahl's Law)
- 项目经理：Fred Brooks (《人月神话》，1999年图灵奖得主)

在 OS360 之前，OS 这个概念不是一个通用的东西，是每台计算机都要去配置的一个软件，每台计算机都是不一样的。IBM 发现计算机型号越来越多的时候，如果要为每一个型号都设计一个 OS，工作是大同小异的，但是细微差别使得又要重新开发一套软件。这导致生产率不高。我们希望设计一套软件：不管底下的硬件怎么改动，我们都可以跑起来。但是当时的硬件不允许做到这个程度，因为当时没有 ISA 的概念，还没有形成软件和硬件分离的形式。

所以为了实现软硬件分离，还得有 ISA 的概念。OS/360 就提出了 ISA，让软件的实现不用 care 硬件的细节。ISA 提出以后，OS 自然就和硬件解耦了。

分时与多任务：Multics/Unix/Linux



当时大家觉得 IBM 的机制非常好，大家希望在 ISA 上做一些工作。当时很多大学和研究机构就希望设计一套现代 OS。Multics 当时的目标非常宏大：有动态链接、文件系统、分时功能等，但是这个太复杂了。Ken 和 Dennis 就在这个开发过程中做了一个简化版的 Multics，也就是 Unix。到 1991 年的时候，因为贝尔实验室开始收 licence fee，所以 Linus 就写出了 Linux 放到网上。

图形界面：Xerox Alto/MacOS/Windows



还有一条发展线就是图形界面。后来苹果公司参观以后，把 GUI 买下。

现代操作系统大同小异前台是图形界面，背后就是各种各样的内核。华为的鸿蒙和欧拉配合，也提出了专门挖矿的“矿鸿”。这也是 OS 的一个定制化的发展方向，OS 在不同的设备上都会有 OS，彼此之间还会互联互通，每个行业可能都会有自己的 OS。上周据说苹果正在开发 RealityOS，除此之外，C919 上面毫无疑问有大量的 OS，还有大疆、小鹏、歼 20、华为海光等都包含了国产的 OS。

Q: CPU 里有 OS 吗？

A: Intel CPU 内有一个 Management Engine，上面跑了一个 Unix (?)，是微服务架构的。

同样，一块 SSD 买来以后，上面也有 OS。注意这是泛化 OS 的概念，是固件。早期的固件就是把传入的地址转化为一个物理地址，做了一次地址的转换，这样防止了一直往同一个

物理地址写，导致被写穿，如 inode bitmap 这一块。我们不能让每个 SSD 的 inode bitmap 是最先坏的地方，所以硬件会有一次地址翻译的过程。

做翻译的这一层固件慢慢演化出了新的功能，甚至也可以跑一些应用程序在上面。一些加载内存的工作可以移交给 SSD 固件去做，这就是 **near-data computing**。

所以，OS 将来会在任何一种形态跑在任何一个地方。所以我们需要知道 OS 的定义。

什么是操作系统？

- 你觉得以下哪些属于操作系统？

- A. Windows 10 所包含的所有软件
- B. Linux 内核以及所有设备的驱动
- C. 在 Macbook 上下载安装的第三方 NTFS 文件系统
- D. 华为 Mate 30 出厂时所有的软件
- E. 大疆无人机出厂时所有的软件
- F. 火星车上运行的软件

- 你觉得应当如何定义操作系统？

- G. 运行在用户态的 I/O 框架。
- Win10 包含一些不是 OS 的东西
- 内核+驱动应该是。

Q: 如果驱动跑在用户态，算不算 OS？

- 文件系统肯定算。
- 和 A 类似，总有一些应用在。
- 没有多余的应用，都是操作无人机必备的软件和机制。虽然运行在用户态，但是有一层关键的软件层。
- 火星车肯定只跑必要的软件。

当我们去想什么是 OS 的时候，它也就没有一个明确的定义。

Q: libc 算不算 OS 的一部分？

A: 任何一个 Linux 的发行版都需要有的组件，但是它不允许在内核态。它到底算不算 OS 就变得模糊。

Q: libssl.so 算不算 OS 的一部分？

A: 没有它 OS 也能跑，但是很多应用运行就很不方便。

所以我们发现 OS 的定义没有那么清晰，它的本质就是在硬件和应用之间的软件层。OS 和应

用的边界并没有那么明确。

OS自然是在两者之间，那么自然有两个关系：服务应用（提供各种各样的 API，让应用程序能够操作硬件）和管理应用（加载应用、调度应用去执行）。

对硬件来说也一样，管理硬件（启动、关掉、重启硬件）和抽象硬件（让应用不用关心硬件的差异）。

所以，**OS**是管理硬件资源、控制程序运行、改善人机界面为应用软件提供支持的一种系统软件。注意到“改善人机界面”有时候并不需要。

我们在书里的定义：**OS**是把有限的、离散的资源，高效地抽象为无限的、连续的资源（为上层提供资源）。

OS也可以包含一层运行在用户态的框架。内核态和用户态是为了让软件分层才出现的机制。像 SSL 库，**Dalvik** 是属于 **Android**，而不属于 **Linux**。我们认为 **Android** 是 **Linux** 的发行版，它的内核是 **Linux**，但是还包含了很多框架。通讯的 **Android service** 都是用户态的通讯框架。

而一个 **hello** 在运行的时候，**OS**要做哪些事情，这就是我们这门课的内容。

【配图：操作系统考虑的一些问题】

2. ICS 中学过的内存的布局。
3. 屏幕作为一个设备是怎么被 **OS** 管理的，提供了什么的抽象。
4. 时分复用做上下文切换
5. 同时运行那么多 **hello**，发生资源不够了该怎么办

OS就要做服务应用（输出到屏幕、允许 2 个 **hello** 相互通信）和管理应用（同时运行 1000 个、加载到内存中）。

【提供的服务】

同步源语和锁是 **OS** 中重要的一个点。

【对应用的管理】

【：管理】

比如说我们怎么避免一个流氓应用（**while 1**）独占 CPU 资源。

【：管理 2】

对于虚拟机来说，所有运行在虚拟机里的应用程序，不管有多少个，从虚拟机外面来看，都是一个进程，我们给虚拟机配置几个虚拟 CPU，那么这个虚拟机进程就有多少线程。所以哪怕有 `forkbomb`，在外面也就是 4 个线程在做 `while 1`，还是可以通过上下文切换或者 `kill` 打断的。

启动进程数达到上限以后，再 `fork` 就会失败。

Q：为什么要有操作系统？我们是不是可以不要操作系统？我们是不是可以把很多可能不要放在 OS 里去做？

【应用和 OS 的解耦】

11.06

预测 OS 的未来是什么样子的最好方法就是看它背后的驱动力是什么。当社会分工发生变化，而 OS 还没有变化的时候，就有一个窗口期，而这个窗口期就是我们做贡献的风口和机会。

对 OS 来说，它是被硬件需求的。所以它对硬件说，要一个 ISA。有了 ISA 之后，再往 CPU 的人说，要有特权 ISA 和非特权 ISA。

非特权部分和特权部分的交互就是 System Call，调用过程就是 strace。

OS 所提供的能力就是抽象和权限。

Q：如果一个机器有且只有一个应用程序，开机自动运行不会退出，是否还需要 OS？

A：在这种情况下，OS 就不具备管理应用的功能。

Q：如果一个应用希望自己完全控制空间而不需要 OS 的硬件抽象，是否还需要 OS？

A：比如 DataBase 希望控制写在磁盘的内圈还是外圈。在这个情况下，除非是唯一的应用，比如有两个应用都想控制同一个硬盘，OS 在这个时候就可以起一个管理的作用。这就是 XO kernel。

所以 OS 提供的向上和向下的功能，不需要同时都起作用。

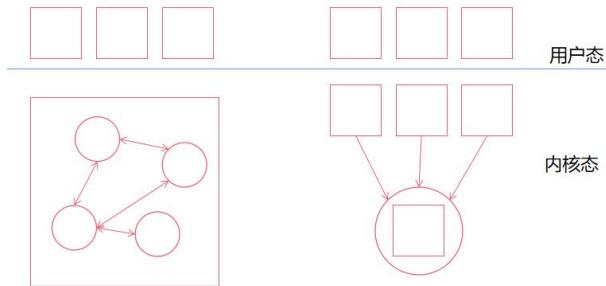
【两种演化】

外部演化就是对外所看到的接口的演化来应对一些新的场景，比如 POSIX 接口就是对新场景所提出的。同样有一些 Syscall，也会被标注为 `deprecated`。鸿蒙也是，提供了一些 high-level 的接口，比如分布式软总线（既可以跑在手表上，也可以跑在大屏幕上）

内部演化不一定是对外可见的，同样的微内核也可以实现 POSIX API，是为了提供更好的扩展、安全、灵活、兼容等功能。单一内核很有可能不能适应。将来的一台计算机就变成了 CPU, GPU, TPU, DPU 等。就有研究者说，我们应该用 multi-kernel，一个应用程序就可以在多个 PU 上来回地迁移。

【用户态还是内核态】

举个例子，早期图形界面是在内核态的，但是经常让整个系统崩掉。所以 Windows 是把图形界面放到一个用户态。Windows NT 就是微内核，但是后面就还是放弃了。所以 Windows 最后采用的方案是：把一些应该放在用户态的东西放回到内核态，架构上还是微内核，但是实际上图形界面还是跑在内核态来获得低时延。



早期内核里跑了很多东西，一个崩了全部崩了。而后期的混合架构就是微内核+一些放在内核态的东西。

AIoT: AI + IoT + 5G + Cloud + ...

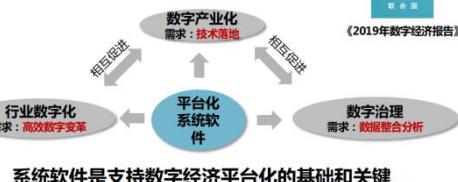


“数字化转型，以操作系统为核心的平台化系统软件是关键”

数字经济转型已成共识，产业数字化转型正加速。
操作系统作为创新平台为代码和内容制作者开发应用程序和软件创造环境。



《2019年数字经济报告》



系统软件是支持数字经济平台化的基础和关键

29

趋势-1：从封闭到开放，再到封闭

例子1：2018年10月起 Google正式对欧盟区域的Android进行收费，初步高达40美元每设备

Google to charge Android OEMs as much as \$40 per phone in EU
After the fee rising, OEMs can no longer copy Android apps, but it will cost them.

例子2：2018年10月 IBM 340亿美元收购 RedHat，构筑其云计算竞争力

IT'S OFFICIAL: IBM is acquiring software company Red Hat for \$34 billion

例子3：2016年起谷歌投入600+人力，数十亿美元，研发面向智能终端设备的自研OS Fuchsia

操作系统并不是免费的午餐，
而是构筑与控制生态的黑土地

早期 unix 封闭导致 Linux 产生，然后大家觉得 OS 就应该开源。Windows 后面开始才免费。Android 全开源。谷歌开发 Fuchsia 的目的就是从 Linux 生态解脱出来，当它开发完 Fuchsia 之后，上面的很多应用都是基于 Fuchsia 的谷歌应用，就很难迁移了。所以生态是用来控制的，底层如果不控制就会一直被人卡脖子。

趋势-2：从专用到通用，再到专用

- 专用 → 通用 → 专用（领域化）

– "I think there is a world market for maybe five computers."
- Thomas Watson, president of IBM, 1943

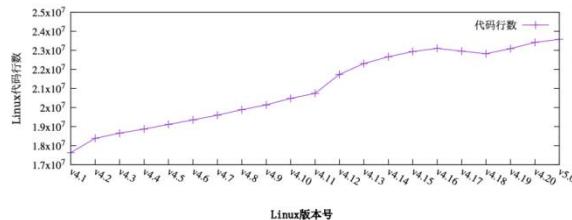
– 计算机系统正在从传统的分层解耦走向云+端的垂直整合

操作系统的类型和数量均会大幅增加

早期的 OS 是非常专用的，到 Linux 就是一个通用的设备，但是在不同的领域，通用设备就不是一个很好的 OS，比如在 64K 内存上。或者在需要一个实时性的场景下，Linux 也不能适应高响应的应用。还有就是在云端处理一切的趋势，这导致在端侧的 OS 针对的硬件和特定场景下会特别简化，而云端再去做不同的处理。

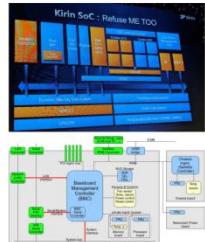
趋势-3：从简单到复杂，到更复杂

- Linux 代码规模已超过 3000 万行
- 仍然以每年 200 万行的数量在增加/更新



一个芯片上的OS不是单一OS，而是一组OS

- 复杂硬件: 芯片上的数据中心
 - 异构计算
 - ARM: 大小核(big.medium.little)设计, 加速器
 - 新型可编程设备 (智能网卡, 智能存储 (SSD), AI加速器)
 - 趋势: 分布式的, 可编程的异构设备
- 含义: 一组OS运行在一个计算机/芯片上



计算机硬件在新应用需求下迅猛发展

- 计算: 从通用计算走向领域计算, 各种 xPU 不断繁荣
 - GPU、TPU、NPU、IPU 等支撑人工智能算力需求
- 存储: 智能存储, 存算一体, 非易失内存 (SCM), 内存与持久存储走向融合
- 数据中心网络: Infiniband 等网络走向纳秒级时延
- 广域网络: 5G 大连接、低时延、高可靠使能新型高吞吐、低时延广域计算



新型硬件发展需要新的
操作系统抽象与设计来充分释放算力

34

哪怕是单机也要考虑分布式可编程的设备。2018 图灵奖就是体系结构领域，演讲：“现在，摩尔定律真的结束了，计算机体系结构将迎来下一个黄金时代”。Python 和专用结构性能也差 6 万倍，我们把通用算法拆解成专用的算法就可以提升性能。所以下一个未来就是 DSA (domain-specific architecture)。有了 DSA 产生之后，上层的 OS 又要发生变化，因为传统的软件不能充分地发挥硬件的能力，OS 怎么设计才能把性能损失给去掉，现在的 OS 关注的是通用性，但是不考虑底层硬件的特长。但是一旦 OS 变成专用了，它就失去了通用带来的好处，现在都没有很好地被解决。

我们现在教 OS 已经不用 Linux 了，因为它超过 3000 万行，历史包袱太重，根本学不懂。100 行里有 90 行是 ifdef。Linux 0.11 代码少不代表适合教学，所以我们还是要从教学角度设计一个 OS。所以一系列变化导致我们需要新的 OS。

2022/2/17(NO)

Q: 这节课内容覆盖是怎么样的？

A: 我们多了一个学分，不仅仅会覆盖上册书的内容。下册书在网上。

上节课我们讲了：OS 是一个不断演化的过程，在这个过程中，往往不是一条直线，是随着上层应用需求和底层硬件的发展往两边做适应的过程。

Review: 操作系统的演化

- 外部演化

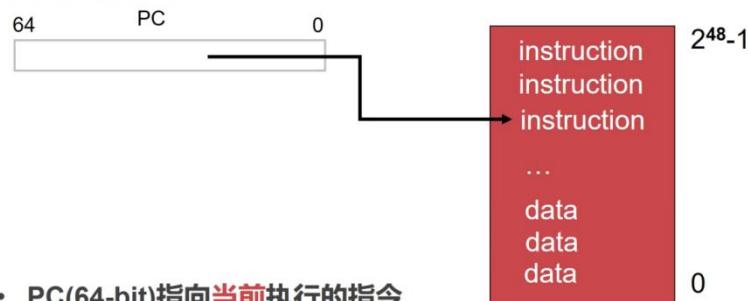
- 接口的演化: 更好的应对新场景
 - POSIX接口: 定义了一组系统调用的接口, 以为应用提供兼容性
 - Linux: 系统调用不断有新的加入、旧的退出
 - 鸿蒙: 分布式软总线等

- 内部演化

- 架构的演化: 更好地应对复杂性
 - 如: 宏内核架构、微内核架构、外核架构、多内核架构等
 - 更好的扩展性、容错性、安全性、兼容性、灵活性、性能等

2022/2/22

AArch64



- PC(64-bit)指向**当前执行的指令**
- 指令长度相同 (RISC, 32-bit)
- PC 会被跳转指令修改: B, BL, BX, BLX

14

地址是 64 位, 代码是 32 位的。

CPU寄存器 ARM

- 31个64位通用寄存器
 - X0-X30
- 1个PC寄存器
- 4个栈寄存器 (切换时保存SP)
 - SP_EL0, SP_EL1, SP_EL2, SP_EL3
- 3个异常链接寄存器 (保存异常的返回地址)
 - ELR_EL1, ELR_EL2, ELR_EL3
- 3个程序状态寄存器 (切换时保存PSTATE)
 - SPSR_EL1, SPSR_EL2, SPSR_EL3

4 个栈寄存器和我们的 x86 是不一样的。注意 x86-64 只有一个栈寄存器 rsp, 而 ARM 有

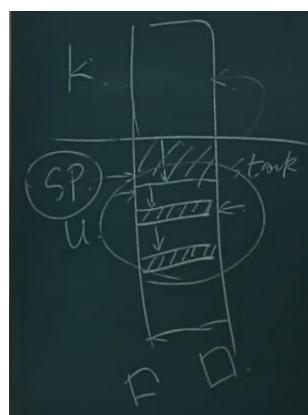
4个：`SP_EL0`, `SP_EL1`, `SP_EL2`, `SP_EL3`。EL指的是execution level, 0表示用户态，而3表示权限最高的，kernel在EL1。

这4个栈寄存器有一个好处，当我们在做上下文切换的时候，不需要换栈寄存器。比如我们从EL0用户态切换到EL1内核态的时候，CPU可以直接用`SP_EL1`所保存的栈指针所在的位置。这样的好处就是速度可以快一点。

如果是`rsp`，我们就需要保存一下`rsp`的原始值，然后再保存成内核态的栈指针。

Q：如果CPU在切换到内核态的时候使用了用户态的栈，会发生什么问题？

A：比如此时我们有两个CPU，一个在内核态，一个在用户态，那么用户态的栈信息可能就会被改掉。所以我们不能认为在内核态的时候使用用户态的栈是安全的。



所以内核必须使用自己的栈，它会为每一个线程创建一个内核栈，和线程的TCB绑定在一起。

当我们在做上下文切换的时候，切换的是`SP_EL0`（用户态栈）和`SP_EL1`（内核中对应线程的栈）。而我们的x86中，我们每次系统调用的时候就必须修改`rsp`到内核态地址，切换回来的时候又要恢复`rsp`的用户态地址。

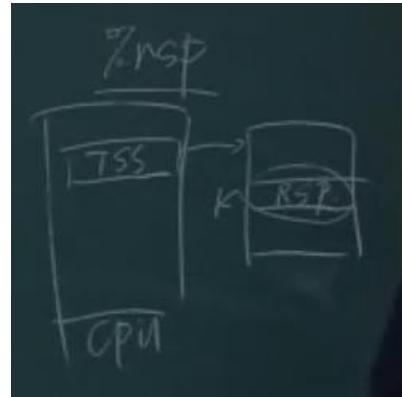
寄存器 ARM vs X86-64

X86-64

- | | | |
|--|--|-----------|
| • 31个64位通用寄存器
– X0-X30 | 思考：X86架构中，切换特权级时 <code>rsp</code> 是如何保存，以及如何恢复的？ | 16个通用寄存器 |
| • 1个PC寄存器 | | 1个%rip寄存器 |
| • 4个栈寄存器（切换时保存SP）
– <code>SP_EL0</code> , <code>SP_EL1</code> , <code>SP_EL2</code> , <code>SP_EL3</code> | | 1个%rsp寄存器 |
| • 3个异常链接寄存器（保存异常的返回地址）
– <code>ELR_EL1</code> , <code>ELR_EL2</code> , <code>ELR_EL3</code> | | 返回地址压栈 |
| • 3个程序状态寄存器（切换时保存PSTATE）
– <code>SPSR_EL1</code> , <code>SPSR_EL2</code> , <code>SPSR_EL3</code> | | EFLAGS |

Q: 在 X86 架构中，切换特权级时 `rsp` 是如何保存以及恢复的？

A: X86 要先把 `rsp` 换成内核栈。CPU 有一个 TSS，TSS 会指向一个数据结构，里面有 `rsp`。这个是内核栈的 `rsp`。我们只需要记住，在 X86 上，我们从内核态返回到用户态之前，我们先要把下次再次进入内核态时的 `rsp` 地址在 CPU 上设置好。CPU 在做切换的时候，先切换 `rsp`，然后再把一些状态压到内核栈上。所以栈切换完，OS 在接手的时候，内核栈上并不是空的，硬件帮助我们压了一些状态，比如因为什么而进入的这个内核态等信息。



但是 X86 这么做，带来了一个很大的问题就是它做的事情太多了，会导致切换的速度比较慢。

我们可以思考一下什么时候会从用户态进入内核态。

`syscall` 的话还是比较抽象。比如发生中断、发生异常（`div 0`，`page fault`）的时候都会进入内核态，此时要切栈、换栈，速度比较慢。这也是为什么后来英特尔出了一条指令 `syscall`。这条指令和以前的切换不太一样，它没有那么多的寄存器的压栈，把很多操作都丢给了软件。

在 ARM 就没有这么复杂，速度就比较快。

RISC vs CISC

ISA

RISC

- 固定长度指令格式
- 更多的通用寄存器
- Load/store 结构
- 简化寻址方式

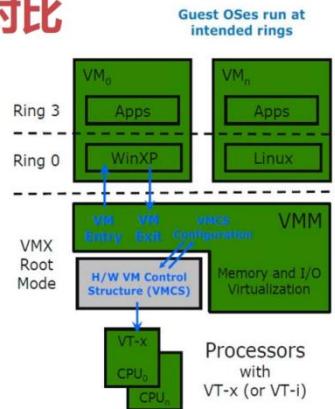
```
0000000000000000 <_start>;
0000000000000000 mrs    x8, apicr_el1
0000000000000000 cbz    x9, #xc0000000
0000000000000000 b2900000 bic    x8, x9, #00000000
0000000000000000 b4000000 cbz    x8, x9, #0001 <primary>
0000000000000000 <secondary_hang>;
0000000000000000 94000000 bl    80010 <secondary_hang>
0000000000000000 94000000 bl    80014 <proc_hang>
0000000000000000 <primary>;
0000000000000000 94001f7f bl    87000 <end_el1_to_el1>
0000000000000000 14000000 add    x9, x9, #3333<code_stack>
0000000000000000 91400000 add    x9, x9, #0x1, lsl #12
0000000000000000 91000011 rev    sp, x9
0000000000000000 91000005 lsl    sp, #1 <int_c>
0000000000000000 17fffffa b     80014 <proc_hang>
0000000000000000 14000007 b     8001c <_start_kernel_vener>
```

	RISC (AArch64)	CISC (x86-64)
指令长度	定长	变长
寻址模式	寻址方式单一	多种寻址方式
内存操作	load/store	mov
实现	增加通用寄存器数量	微码
指令复杂度	简单	复杂
汇编复杂度	复杂	简单
中断响应	快	慢
功耗	低	高
处理器结构	简单	复杂

RISC 和 CISC 对比有些特点。CISC 有很多寻址方式，比如有一个起始地址和间隔长度再乘上一个 count。这种方式很容易计算数组的取值。

特权级：与X86-64对比

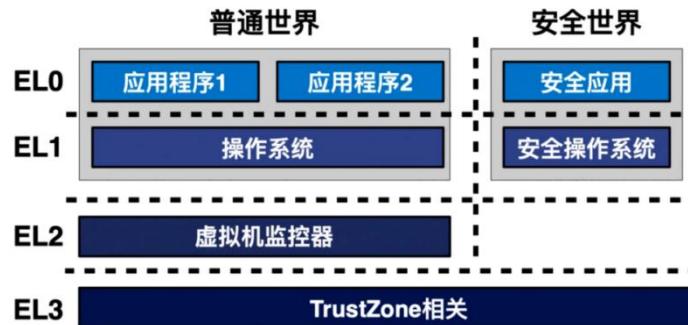
- **Non-root :**
 - Ring 3: Guest app
 - Ring 0: Guest OS
- **Root:**
 - Ring 3: App
 - Ring 0: Hypervisor



X86 为了虚拟化，引入了 Hypervisor 等特权级。

而 ARM 的特权级如下：

特权级/ARM (Exception Level)



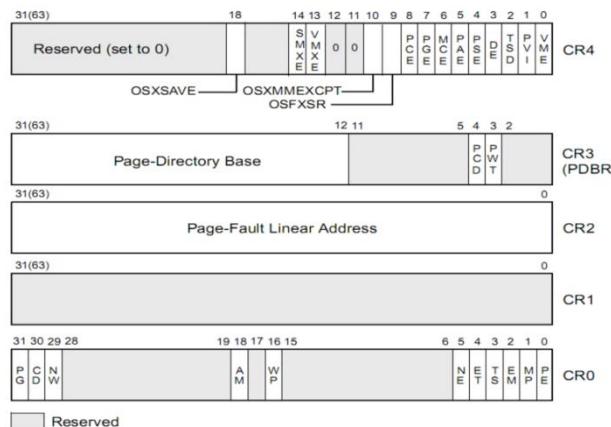
为了引入虚拟机，加上了 EL2。为了支持安全世界，又加入了安全应用和安全 OS。所以我们看到，特权级 Intel 有的，这里也有，只是形式上有点区别。

系统状态寄存器：ARM

- **抽象进程状态信息 (PSTATE)**
 - 条件标记 (Condition flags)
 - 执行状态 (Execution state controls)
 - 异常掩码 (Exception mask bits)
 - 访问, 时钟控制 (Access control bits)

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
CurrentEL	EL
SPSel	SP
PAN	PAN
UAO	UAO
DIT	DIT
SSBS	SSBS

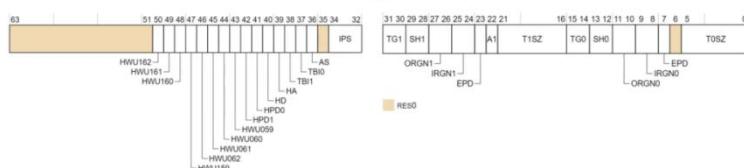
系统控制寄存器：X86-64



ARM 也有类似的用来打开很多 control register 的。

内存系统相关寄存器：ARM

• Translation Control Register (TCR)



• Translation Table Base Register (TTBR)

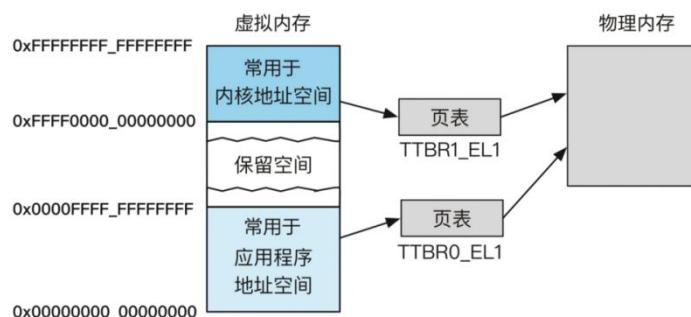
对比X86-64:
CR3寄存器



25

TTBR 和 X86-64 的 CR3 对比起来有一个特别大的区别，就是 ARM 有两个 TTBR，分别为 TTBR1_EL1 和 TTBR0_EL1。

内存空间



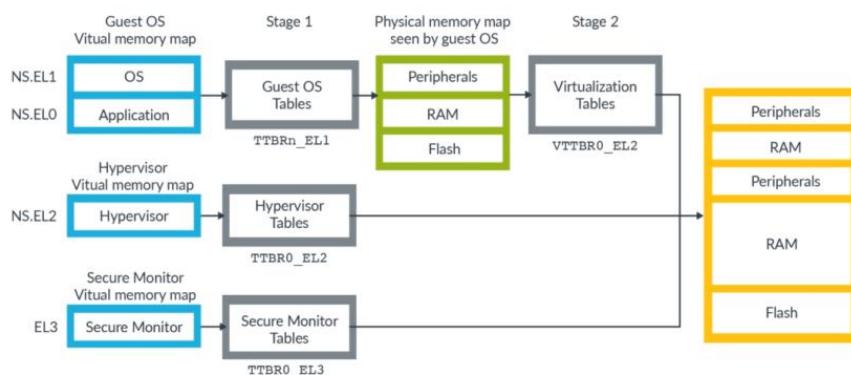
XXXX_EL1 后缀指的是这个寄存器只能在这个 EL (特权级) 使用，TTBR 显然只能被内核访问，所以这两个后缀都是 EL1。为什么要两个 TTBR 呢？ARM 的内存地址并不是把 2^{64} 全部囊括了。大概是从 2^{48} 开始分为高地址和低地址。高地址的特点就是前 16 位为 1，低地址的特点就是前 16 位为 0。

所以我们从低地址往上索引，最多只能访问到 2^{48} 。高地址也只能访问 2^{48} ，所以它总体能访问的内存量是 $2 * 2^{48} = 2^{49}$ 。高地址使用 TTBR1 去翻译，低地址使用 TTBR0 去

翻译。

由于在 OS 里基本上会用高地址，所以 TTBR1 翻译内核空间而 TTBR0 去翻译用户态。所以在进程切换的时候很方便，我们只需要考虑 TTBR0 即可了。但是在 x86 的情况下，我们只有一个 CR3，既要翻译内核地址又要翻译用户态地址。意味着页表有大量的指针指向内核态的三级页表。

地址翻译



地址翻译范围 Stage1 和 Stage2, Stage2 主要是给虚拟机用的。

那么所有的输入输出和 Intel 也有区别，它的所有输入输出都是用 memory map 实现的，复用了 store 和 load 指令。

ARM 输入/输出



MMIO与PIO

- MMIO (Memory-mapped IO)**
 - 将设备映射到连续的物理内存中，使用相同的指令
 - 如，Raspi3映射到0x3F200000
 - 行为与内存不一样，读写会有副作用（回忆volatile）
- PIO (Port IO)**
 - IO设备具有独立的地址空间
 - 使用特殊的指令（如x86中的in/out指令）

注意 ARM 没有 PIO、只有 MMIO。

CPU的执行逻辑

- CPU的执行逻辑很简单
 1. 以PC的值为地址从内存中获取一条指令并执行
 2. PC+=4, goto 1
- 执行过程中可能发生两种情况
 1. 指令执行出现错误, 比如除零或缺页 (同步异常)
 2. 外部设备触发中断 (异步异常)
- 这两种情况在ARM平台均称为「异常」
 - 均会导致CPU陷入内核态, 并根据异常向量表找到对应的处理函数执行
 - 处理函数执行完后, 执行流需要恢复到之前被打断的地方继续运行
 - 注意与x86相关概念的区别

21

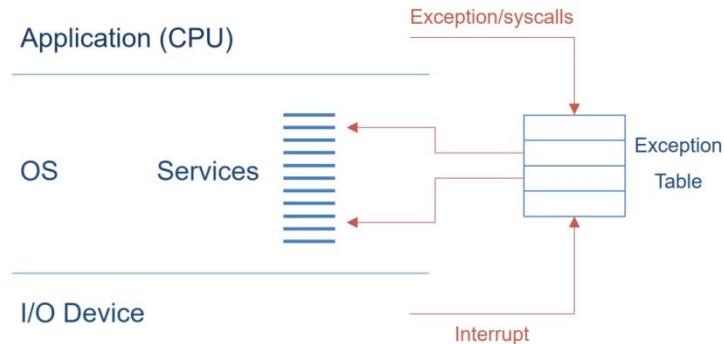
操作系统的执行流可对应地分为两部分

- 一、实现对异常向量表的设置
 - CPU上电后立即执行, 是系统初始化的主要工作之一
 - 在开启中断和启动第一个应用之前
- 二、实现对不同异常(中断)的处理函数
 - 处理应用程序出错的情况: 如除零、缺页
 - Q: 内核如果自己运行出错怎么办?
 - 一类特殊的同步异常: 系统调用, 由应用主动触发
 - 处理来自外部设备的中断: 如收取网络包、获取键盘输入等

Q: kernel 中自己运行出错怎么办?

A: triple fault (三振出局), 内核中也会缺页。

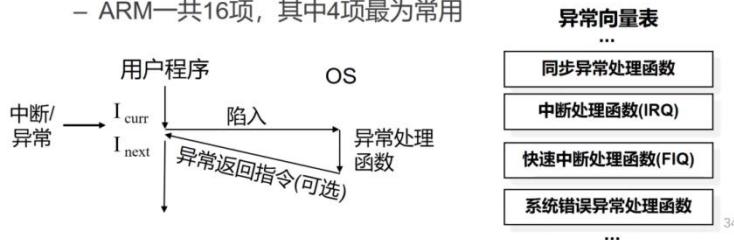
操作系统的执行流 (简化版)



异常向量表

异常向量表

- 操作系统预先在一张表中准备好不同类型异常的处理函数
 - 地址存储在**VBAR_EL1寄存器**中
 - 处理器在异常发生时自动跳转到对应位置
 - ARM一共16项，其中4项最为常用



34

同步异常是最多的，也就是系统调用。

异常处理函数

- 异常处理函数运行在内核态
 - 可以不受限制地访问所有资源
- 处理器将异常类型存储在指定寄存器中 (**ESR_EL1**)
 - 表明发生的是哪一种异常
 - 异常处理函数根据异常类型执行不同逻辑

异常处理函数

- 当异常处理函数完成异常处理后，将通过下述操作之一转移控制权：
 - 回到发生异常时正在执行的指令
 - 回到发生异常时的下一条指令
 - 结束当前进程

异常类型放在 **ESR_EL1** 中。

系统调用

接下来我们讲系统调用。

常见的Linux的系统调用

编号	名称	描述	编号	名称	描述
17	getcwd	Get current working directory	129	kill	Send signal to a process
23	dup	Duplicate a file descriptor	172	getpid	Get process ID
56	openat	Open a file	214	brk	Set the top of heap
57	close	Close a file	215	munmap	Unmap a file from memory
63	read	Read a file	220	clone	Create a process
64	write	Write a file	221	execve	Execute a program
80	fstat	Get file status	222	mmap	Map a file into memory
93	_exit	Terminate the process	260	wait4	Wait for process to stop

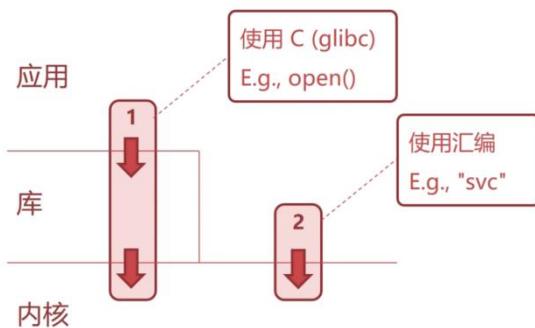
如果我们要跟踪系统调用，可以用 **strace**。

```
int main() {
    write(1, "Hello world!\n", 13);
}
```

```
$ strace -o hello.out ./hello

execve("./hello2", ["../hello2"], /* 59 vars */) = 0
uname({sys="Linux", node="kiwi", ...}) = 0
brk(0) = 0xca9000
brk(0xcaa1c0) = 0xcaa1c0
arch_prctl(ARCH_SET_FS, 0xca9880) = 0
brk(0xccb1c0) = 0ccb1c0
brk(0xccc000) = 0ccc000
write(1, "Hello world!\n", 13) = 13
exit_group(13) = ?
```

程序员角度看系统调用



从程序员角度来看，可以使用 glibc 封装的系统调用，也可以使用汇编的系统调用（汇编指令 svc）。

svc指令：从用户态进入内核态

• CPU的行为

- 将处理器状态PSTATE保存到寄存器SPSR_EL1中
- 将用户程序中svc后的第一条指令所在地址保存到寄存器ELR_EL1
- 将处理器正在使用的栈指针寄存器由SP_EL0切换为SP_EL1
- 将 PSTATE 中的特权级别切换到内核态 (EL1)
- 根据异常向量表中的配置，执行对应异常向量条目所配置的代码

eret指令：从内核态返回用户态

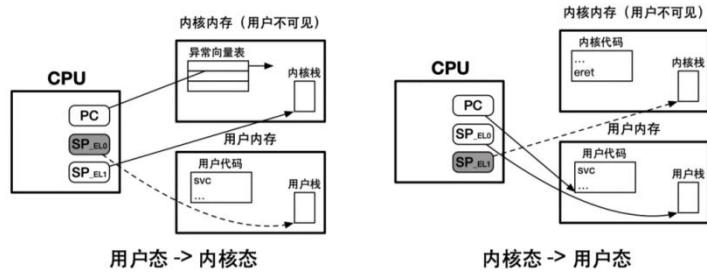
• CPU的行为

- 将SPSR_EL1中所存储的处理器状态重设到处理器状态PSTATE中
- 将处理器正在使用的栈指针寄存器由SP_EL1切换为SP_EL0
- 将ELR_EL1寄存器中所保存的返回地址重设到程序计数器PC中，执行应用程序中的代码

eret 就是从 EL1 切换回 EL0。

下图中，左图是从用户态切换到内核态的情况，右图是从内核态切换到用户态的情况。

两种模式（态）的切换：用户态 <-> 内核态



上下文保存与恢复

- 硬件只保存了部分寄存器
 - 只包含PSTATE和PC
 - 不包含通用寄存器、SP
- Q: 为什么硬件不全部保存呢?
- 由软件来（按需）完成其他上下文寄存器的保存
 - 主要是：通用寄存器（X0-X30）
 - SP不需要保存：因为硬件有SP_ELO和SP_EL1两个寄存器

Q: 内核是否可以自己调用 `syscall` 呢？

A: 可以，此时并不会发生之前提到的硬件换栈等操作。

内核的传参

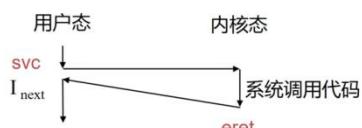
传参最多允许 8 个参数，x0~x7，返回值放在 x0 寄存器中。

系统调用的参数与返回值

参数传递

- 最多允许8个参数：x0-x7寄存器
- x8用于存放系统调用编号
- 调用者保存的寄存器必须在用户态保存

返回值存放于x0寄存器中



系统调用返回值与errno

一般库函数

- 出错时返回-1，并设置全局变量errno为具体的错误值

系统调用通过寄存器向应用传递返回值

- 一般设置为 -errno
- 库对系统调用的 wrapper code 会将系统调用的返回值转换为库函数形式的返回值

如果返回值是错误的怎么办？如果库函数出错，返回一般是-1，并且把全局变量设置为具体的错误值。而 perror 就是去读这个全局变量并且把对应的错误输出出来。

Q: 如果寄存器放不下参数怎么办?

- **寄存器放不下，只能通过内存传参**
 - 将参数放在内存中，将指针放在寄存器中传给内核
 - 内核通过指针访问相关参数
- **想一想，可能有什么问题？**
 - 情况-1：指针指向了内核区域（攻击！）
 - 情况-2：指针指向的区域被swap-out了
 - 导致内核访问时出现了page fault，怎么处理？
 - 情况-3：指针指向了未映射区域
 - 导致segmentation fault，会怎么处理？

如果用户态传了一个指针给内核，并且这个指针是指向内核区域的。这就会导致问题。比如我们 `print(内核地址)`。所以内核是不能信任用户态的指针的。包括说内核访问的时候可能出现 `page fault`。

未映射区域就是 VMA 的外面，如果指向了未映射区域，就会出现 `segmentation fault`。内核里出现这个问题显然不能报出 `segmentation fault`。

如何验证用户态的指针合法性？

- **如果仔细检查的话太费时**
 - 需要检查指针是否来自所有的合法内存区域（VMA）
- **Linux的方案**
 - 仅仅做一个简单的检查，判断是否在最大的VMA
 - 即使检查过了，指针依然有可能不合法
 - 在内核态发生非法内存访问，一般会被认为是内核bug而触发
`Oops`，并kill掉相关进程

所以碰到传指针的情况就非常复杂，我们需要一点点检查指针指向的区域是否在 VMA 里、是否在页表里、是否在用户栈上，但是这样非常麻烦。

所以工程上的实现，Linux 只做一个简单的检查看看是不是在最大的 VMA 里。当内核发现非法内存访问的时候，我们就认为这是一个 Bug 并且 kill 掉。

处理由用户指针而带来的fault

- 内核代码通过一组特定的routine来访问用户内存
 - 例如: copy_from_user
- 当发生page fault的时候, 内核会检查PC的值
 - 如果PC的值在这组routine中, 则不会Oops, 而是会尝试运行fixup代码, 去解决page fault
- 在内核中, 这是一个常常会出现漏洞的地方
 - 回顾下x86的 SMAP 特性: 防止内核访问用户态内存

SMAP 机制 (用户态内存不能被内核态 CPU 访问)

此时我们需要用到 SMAP (supervisor mode access prevention) 机制, 它可以做到用户态的内存地址不能被内核态的 CPU 访问。CPU 在内核态尝试访问用户态内存时会被直接拒绝掉。

访问用户态内存的一组routine

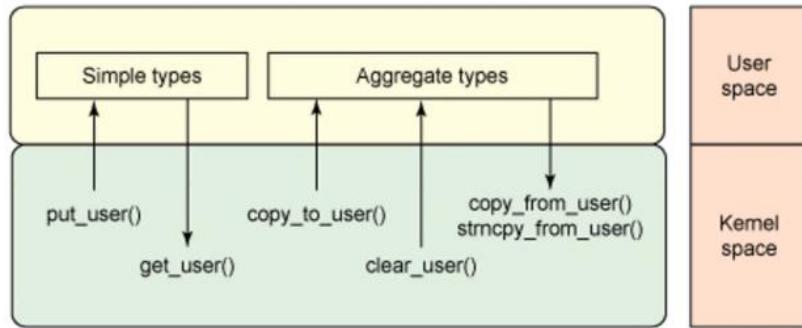
Function	Action
get_user(), __get_user()	reads integer (1,2,4 bytes)
put_user(), __put_user()	writes integer (1,2,4 bytes)
copy_from_user(), __copy_from_user()	copy a block from user space
copy_to_user(), __copy_to_user()	copy a block to user space
strncpy_from_user(), __strncpy_from_user()	copies null-terminated string from user space
strnlen_user(), __strnlen_user()	returns length of null-terminated string in user space
clear_user(), __clear_user()	fills memory area with zeros

这些函数都是在内核里访问用户态内存的函数。

Q: 一旦用了 SMAP, 这些函数不就出错了吗?

A: 在 copy_from_user 的函数第一行, 把 SMAP 关掉, 再在最后一行把 SMAP 打开。

访问用户态内存的一组routine



VDSO (Virtual Dynamic Shared Object)

Motivation

- 系统调用的时延不可忽略
 - 尤其是调用非常频繁的那些
 - 如 `gettimeofday()`
- 如何降低系统调用的时延?
 - 大部分时延都是由于U->K的模式切换带来的
 - 如果没有模式切换，那么就不需要保存回复状态

`gettimeofday` 是可以设置精度的。我们要减掉切换所带来的时延。现在 `gettimeofday` 是内核记录的时间，整个系统的时间只有一个。在最新的内核里允许时间有一个 namespace，也就是在系统上启动了两个容器，每个容器都可以设置自己的时间，把时间和容器做了绑定。

既然内核态和用户态之间的切换会带来时延，那么我们能不能在不切换的情况下就得到 `gettimeofday` 的结果呢？

The Code of `gettimeofday()`

- **内核定义**

- 在编译时作为内核的一部分

- **用户态运行**

- 将`gettimeofday`的代码加载到一块与应用共享的内存页
 - 这个页称为：vDSO
 - Virtual Dynamic Shared Object
 - Time 的值同样映射到用户态空间（只读）
 - 只有在内核态才能更新这个值

- **Q：和以前的`gettimeofday`相比有什么区别？**

在编译的时候有一段代码叫做 `gettimeofday`，当用户态的应用程序开始运行加载二进制的时候，我们偷偷在内存中加上一页 `gettimeofday` 的代码。也就是内核在用户态的空间中注入了 `gettimeofday` 的代码。然后当我们调用 `gettimeofday` 的时候，会直接调用这块代码，可以读取用户态空间只读的 Time 值。这样就不会发生内核和用户态的切换。

Q： 和以前的 `gettimeofday` 相比有什么区别？

A： 以前调用的时候，需要进入内核，去读晶振的值再返回。而在当前情况下，因为我们是读内核态更新的时间的值，如果内核没有更新这个值的话，我们读到的数字是一样的。在多核的场景下，每一个核都有可能去更新这个值。

Q： 为什么在这里不需要加锁呢？

A： 因为用户态不会写这个值，只有一个人去写这个值。比如我们有 10 个 CPU core，任何一个 core 都会更新这个 time 的值。

综上所述，这个页就叫做 vDSO。OS 把代码映射到了用户态，把值也映射到了用户态。来减轻切换到内核所带来的开销。

vDSO的共享页在哪儿？

```
$ ldd `which bash`  
linux-vdso.so.1 => (0x00007fff667ff000)  
libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f623df7d000)  
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f623dd79000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f623d9ba000)  
/lib64/ld-linux-x86-64.so.2 (0x00007f623e1ae000)
```

2021/2/24

简单来讲，vDSO 省下了我们 `gettimeofday` 控制流转换的时间。这个适合内核提供一个

值，而用户态去读这个值的场景。

FLEX-SC

接下来我们要讲另一种实现方法，也就是 FLEX-SC。

Motivation

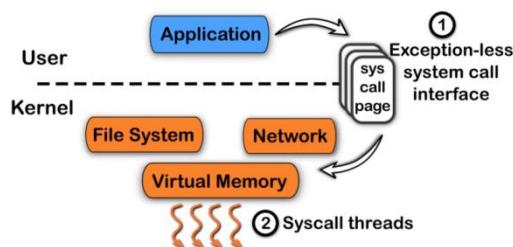
- 如何进一步降低系统调用的时延?
 - 不仅仅是 gettimeofday()
- “时间都去哪儿了？”
 - 大部分是用来做状态的切换
 - 保存和恢复状态 + 权限的切换
 - Cache pollution
- 是否有可能在不切换状态的情况下实现系统调用？

Flexible System Call

- 一种新的syscall机制
 - 引入 **system call page**，由 user & kernel 共享
 - User threads 可以将系统调用的请求 **push** 到 system call page
 - kernel threads 会从system call page **poll** system call 请求
- **Exception-less syscall**
 - 将系统调用的调用和执行解耦，可分布到不同的CPU核

和 vDSO 类似的，FLEX-SC 也会引入一个 page，叫做 **syscall page**，由用户态和内核态共享。但这个 page，用户态和内核态都可以读和写。有点像一个环状的 ring buffer，一头读、一头写。用户态的线程可以把请求 **push** 到 page 里。

System Call的另一种方法



Exception-less System Call

```
write(fd, buf, 4096);

entry = free_syscall_entry();
/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

while (entry->status != DONE)
    do_something_else();

return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
			:	

我们可以把原先的 write 变成如上这种方式。从 syscall page 中拿出一个 page，设置完对应的参数以后提交到 syscall page 中。

Q: FLEX-SC 适合什么场景？

A: 适合重要而不紧急的事情，可以减少上下文切换的开销。对于重要紧急的事情，还是可以通过 syscall 主动掉入内核态中。

计算机启动

启动流程：从上电到等待用户输入



16

内核启动的2个主要任务

- 配置页表并开启虚拟内存机制
 - 页表究竟该如何具体配置？
 - 难点：开启地址翻译的前一行指令使用物理地址，开启后立即使用虚拟地址，前后如何衔接？
- 配置异常向量表并打开中断
 - 异常向量表如何配置？
 - 打开后，异常处理的指令流如何流动？

真·第一行代码

注意: ChCore代码中也出现过bootloader, 是指boot目录下的代码, 与这里不同

- **树莓派: 上电后真正运行的第一行代码**

- 板子上电后固定从0x0地址运行firmware (也称 bootloader)
 - firmware可以放在NORFLASH (一种存储介质) 中
- 然后再由这段代码去初始化CPU、SDRAM等
- 最后再加载内核、根文件系统到内存, 实现系统启动

- **不同主板厂商的实现可以不同**

- 这部分代码由主板厂商提供, 使用人员通常不用关心
- 树莓派的启动比较特殊: 第一行代码由GPU运行
- 不同版本的树莓派也可能不一样

20

树莓派特殊的启动3阶段 (by GPU)

- **Step-1**

- GPU: 运行0x0地址的firmware, 将SD卡根目录下的bootcode.bin加载到内存中

- **Step-2**

- GPU: 运行bootcode.bin, 将SD卡根目录下的start.elf加载到内存中

- **Step-3**

- GPU: 运行start.elf, 将SD卡根目录下的kernel.img加载到内存中

- **然后, CPU 跳转到kernel.img, 开始真正运行内核代码**

入口函数位置

Q: 这是什么地址?

- **CPU从预定义的RAM地址读取第一行代码, 由硬件厂商决定**

- 树莓派: 32位为0x8000, 64位为0x80000

- x86: 0x7C00

kernel_address

`kernel_address` is the memory address to which the kernel image should be loaded. 32-bit kernels are loaded to address `0x8000` by default, and 64-bit kernels to address `0x80000`. If `kernel_old` is set, kernels are loaded to the address `0x0`.

<https://www.raspberrypi.org/documentation/configuration/config-txt/boot.md>

21

ChCore启动代码

```
→ chcore-lab $ cd boot
→ boot $ ls
boot.h config.cmake image.h init_c.c mmu.c start.S tools.S
uart.c uart.h

→ boot $ cd ../kernel
→ kernel $ ls
CMakeLists.txt exception ipc mm process syscall
common head.S main.c monitor.c sched tests
```

两个涉及到的目录：boot 和 kernel

- boot目录：编译后放在 .init 段，低地址范围（有时也叫bootloader，再次注意不要混淆）
- kernel目录：编译后放在 .text 段，高地址范围

ChCore内核的起始地址（编译脚本）

<p>boot/image.h</p> <pre>1 #pragma once 2 3 #define SZ_16K 0x4000 4 #define SZ_64K 0x10000 5 6 #define KERNEL_VADDR 0xffffffff0000000000 7 #define TEXT_OFFSET 0x80000 8 63 64 set(BINARY_KERNEL_IMG_PATH "CMakeFiles/kernel.img.dir") 65 set(init_object 66 \${BINARY_KERNEL_IMG_PATH}/\${BOOTLOADER_PATH}/start.S.o 67 \${BINARY_KERNEL_IMG_PATH}/\${BOOTLOADER_PATH}/mmu.c.o 68 \${BINARY_KERNEL_IMG_PATH}/\${BOOTLOADER_PATH}/tools.S.o 69 \${BINARY_KERNEL_IMG_PATH}/\${BOOTLOADER_PATH}/init_c.c.o 70 \${BINARY_KERNEL_IMG_PATH}/\${BOOTLOADER_PATH}/uart.c.o" 71) 72 </pre>	<p>scripts/linker-aarch64.lds.in</p> <pre>1 #include "../boot/image.h" 2 3 SECTIONS 4 { 5 . = TEXT_OFFSET; 6 img_start = .; 7 init : { 8 \${init_object} 9 } 10 . = ALIGN(SZ_16K); 11 12 init_end = ABSOLUTE(.); 13 14 </pre>
---	---

通过编译脚本控制内核二进制布局

25

TEXT_OFFSET 其实就是我们刚才讲到的树莓派 64 位的第一行代码的地址。

内核运行的第一行代码：准备进入EL1

<pre>9 10 BEGIN_FUNC(_start) 11 mrs x8, mpidr_el1 /* move core ID to x8 */ 12 and x8, x8, #0xFF /* mask */ 13 cbz x8, primary /* compare branch zero */ 14 45 46 primary: 47 48 /* Turn to el1 from other exception levels. */ 49 bl arm64_elX_to_el1 50 51 /* Prepare stack pointer and jump to C. */ 52 adr x0, boot_cpu_stack // only used for boot 53 add x0, x0, #0x1000 54 mov sp, x0 55 56 bl init_c 57 58 /* Should never be here */ 59 b . 60 END_FUNC(_start)</pre>	<p>初始时CPU运行在EL3 (由硬件厂商决定)</p> <p>设置当前EL为EL1（内核的运行级）</p> <p>设置启动时用的栈（用于C的函数调用）</p> <p>跳转到C代码（返回地址保存在X30寄存器中）</p>
--	---

boot/start.S

26

这里面为什么要有一个 core ID 呢？因为我们现在的核都是多核的，我们需要知道这个核是第几个。刚起来的时候只有一个核。

为什么要设置栈呢？因为我们要进入 C 代码，需要准备好函数调用、传参的栈。然后就可以进入 C 代码了，因为我们尽量希望避免用汇编来写代码，能用 C 就用 C。所以我们先准备好了栈。准备好了以后我们就可以到 C 的代码里去。

```

64
65 BEGIN_FUNC(arm64\_elX\_to\_el1)
66 mrs x9, CurrentEL // read from a reg, decided by the board
67
68 // Check the current exception level.
69 cmp x9, CURRENTEL_EL1
70 beq .Ltarget // if EL1, no need to eret, just ret
71 cmp x9, CURRENTEL_EL2
72 beq .Lin_el2 // if EL1, need to eret from EL2
73 // Otherwise, we are in EL3.
74
75 mrs x9, scr_el3 // scr: secure configure reg
76 mov x9, SCR_EL3_NS | SCR_EL3_HCE | SCR_EL3_RW
77orr x9, x9, x10
78 msr scr_el3, x9
79
80 // Set the return address and exception level.
81 adr x9, .Ltarget
82 msr elr_el3, x9 // elr: exception link reg
83 mov x9, SPSR_ELX_DAIF | SPSR_ELX_ELRH
84 msr spsr_el3, x9
85

128
129 isb
130 eret
131
132 .Ltarget:
133     ret // retaddr in x30
134 END_FUNC(arm64\_elX\_to\_el1)
135

```

树莓派启动后，CPU运行在EL3
(后面代码起作用的主要是_el3相关部分)

设置scr_el3寄存器：NS、HCE、RW（后一页）

为eret做准备：

1. 设置EL3的exception link register (返回地址)
 2. 设置EL3的状态寄存器SPSR
(D: debug; A: error; I: interrupt; F: fast interrupt)

isb: memory barrier, 保证顺序执行
eret: 跳到 .Ltarget, 同时进入EL1

ret: 返回到start.S的50行

4

<https://github.com/s-matyukevich/raspberry-pi-os/blob/master/docs/lesson02/rpi-os.md>

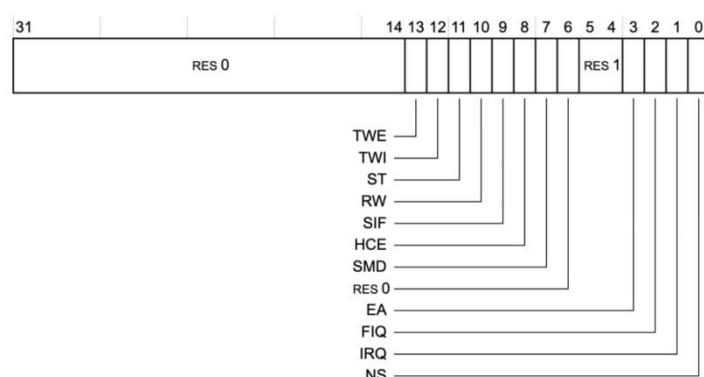
27

这段代码首先要设置一系列的 control register。

Q: 为什么要用 `eret`, 而不是使用 `ret` 呢?

A: 我们通过 eret 进入到 EL1。

Figure 4.38. SCR EL3 bit assignments



Bit	V	Semantic
[10] RW		Register width control for lower exception levels. The possible values are: 0: Lower levels are all AArch32. This is the reset value. 1: The next lower level is AArch64.
[8] HCE		Hyp Call enable. This bit enables the use of HVC instructions. The possible values are: 0: The HVC instruction is undefined at all exception levels. This is the reset value. 1: The HVC instruction is enabled at EL1, EL2 or EL3.
[0] NS		Non-secure bit. The possible values are. The possible values are: 0: EL0 and EL1 are in Secure state, memory accesses from those exception levels can access Secure memory. This is the reset value. 1: EL0 and EL1 are in Non-secure state, memory accesses from those exception levels cannot access Secure memory.

boot/start.S

```

45
46 primary:
47
48     /* Turn to el1 from other exception levels. */
49     bl arm64_elx_to_el1
50
51     /* Prepare stack pointer and jump to C. */
52     adr x0, boot_cpu_stack // only used for boot
53     add x0, x0, #0x1000
54     mov sp, x0
55
56     bl init_c
57
58     /* Should never be here */
59     b .
60 END_FUNC(_start)

```

boot/init_c.c

```

1 #include "boot.h"
2 #include "image.h"
3
4 typedef unsigned long u64;
5
6 #define INIT_STACK_SIZE 0x1000
7 char boot_cpu_stack[PLAT_CPU_NUMBER][INIT_STACK_SIZE] ALIGN(16);
8
9
10 void init_c(void)
11 {
12     /* Clear the bss area for the kernel image */
13     clear_bss();
14
15     /* Initialize UART before enabling MMU. */
16     early_uart_init();
17     uart_send_string("boot: init_c\r\n");
18
19     wakeup_other_cores(); // no need for qemu
20
21     /* Initialize Boot Page Table. */
22     uart_send_string("[BOOT] Install boot page table\r\n");
23     init_boot_pt();
24
25     /* Enable MMU. */
26     el1_mmuv1_activate();
27     uart_send_string("[BOOT] Enable el1 MMU\r\n");
28
29     /* Call Kernel Main. */
30     uart_send_string("[BOOT] Jump to kernel main\r\n");
31     start_kernel(secondary_boot_flag);
32
33     /* Never reach here */
34 }

```

设置栈为boot_cpu_stack，之后就可以调C函数

- 问：为什么调C函数之前要设置栈？
- 问：栈的大小是多少？为什么够？

Q: 这个栈应该设置多大呢？为什么够呢？

A: 每个CPU设置4K。够是因为代码是我们写的。

init_c先做了clear_bss。bss段默认应当是为零的。UART其实就是一个输出串口，这是我们调试的第一个根据，初始化完我们可以看到打印。

从 /boot 到 /kernel

The diagram illustrates the flow from assembly code to linker script to final memory layout.

kernel/head.S:

```
12 #include <common/asm.h>
13 #include <common/vars.h>
15
16 BEGIN_FUNC(start_kernel) // high memory addr
17 /*
18  * Code in bootloader specified only the primary
19  * cpu with MPIDR = 0 can be boot here. So we directly
20  * set the TPIDR_EL1 to 0, which represent the logical
21  * cpuid in the kernel
22 */
23 mov x3, #0
24 msr TPIDR_EL1, x3 // set CPU ID, only the primary will run this code
25
26 ldr x2, =kernel_stack // high memory addr
27 add x2, x2, KERNEL_STACK_SIZE
28 mov sp, x2 // switch stack, important
29 bl main
30 END_FUNC(start_kernel)
31
```

scripts/linker-aarch64.lds.in:

```
1 #include "../boot/image.h"
2
3 SECTIONS
4 {
5     . = TEXT_OFFSET;
6     img_start = .;
7     init : {
8         ${init_object}
9     }
10
11    . = ALIGN(SZ_16K);
12
13    init_end = ABSOLUTE(.);
14
15    .text.KERNEL_VADDR: init_end : AT(init_end) {
16        *(.text*)
17    }
18}
```

A yellow arrow points from the `KERNEL_VADDR` definition in the linker script to the corresponding section in the assembly code. A red arrow points from the `init_end` label in the linker script to the `init_end` label in the assembly code.

start_kernel位于高地址段: `0xffffffff000000000000 + init_end`

问：运行start_kernel之前是低地址段，为什么可以直接跳到高地址段？1

boot 的代码是 init 段里的，一旦启动完就会释放掉内存。而 kernel 里的代码是在 text 段里的，也就是我们是要一直使用的。

start_kernel 又进入到了汇编指令。为什么要再进入到汇编呢？主要还是有一些设置是只能通过汇编来完成的，设置完了我们再进入到 main。这里涉及到了一系列的寄存器的设置。把 high memory 的地址加载到我们的 x2 里，再切换我们的 stack。为什么这里又有一个切换栈的过程呢？

因为我们原来的栈只是用来启动的临时的栈，它是在低地址区域的一个临时的栈，并不是 OS 正常运行的时候所使用的栈。

Q: 为什么可以直接跳到高地址段？

A: 原因是因为在执行到这里的时候，我们已经初始化了页表并且启动了 MMU。

页表初始化

回到 init_c：页表初始化

```
65 void init_c(void)
66 {
67     /* Clear the bss area for the kernel image */
68     clear_bss();
69
70     /* Initialize UART before enabling MMU. */
71     early_uart_init();
72     uart_send_string("boot: init_c\r\n");
73
74     wakeup_other_cores(); // no need for qemu
75
76     /* Initialize Boot Page Table. */
77     uart_send_string("[BOOT] Install boot page table\r\n");
78     init_boot_pt();
79
80     /* Enable MMU. */
81     el1_mm_activate();
82     uart_send_string("[BOOT] Enable el1 MMU\r\n");
83
84     /* Call Kernel Main. */
85     uart_send_string("[BOOT] Jump to kernel main\r\n");
86     start_kernel(secondary_boot_flag);
87
88     /* Never reach here */
89 }
90 }
```

```
39 void init_boot_pt(void)
40 {
41     u32 start_entry_idx;
42     u32 end_entry_idx;
43     u32 idx;
44     u64 kva;
45
46     /* TTBR0_EL1 0-1G */
47     boot_ttbr0_l0[0] = ((u64) boot_ttbr0_l1) | IS_TABLE | IS_VALID;
48     boot_ttbr0_l1[0] = ((u64) boot_ttbr0_l2) | IS_TABLE | IS_VALID;
49
50     /* Usable memory: PHYSMEM_START ~ PERIPHERAL_BASE */
51     start_entry_idx = PHYSMEM_START / SIZE_2M;
52     end_entry_idx = PERIPHERAL_BASE / SIZE_2M; 为什么是2M?
53
54     /* Map each 2M page */
55     for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
56         boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
57             | UXN /* Unprivileged execute never */
58             | ACCESSED /* Set access flag */
59             | INNER_SHARABLE /* Shareability */
60             | NORMAL_MEMORY /* Normal memory */
61             | IS_VALID;  设置TTBR0页表 (低地址使用)
62     }
63 }
```

我们可以看一下 `init_boot_pt` 做了什么事情。我们初始化了 `boot_ttbr0_l0` 和 `l1` 的地址。它能用的物理内存索引就是 `start` 到 `end`。然后再把每个 2M 的页映射成地址。在 ICS 中我们学习的是 4K 大小的页，在 ARM 中不仅仅有 4K 的页，还有 2M 的页。原来我们 ICS 中有一个 TLB（页表缓冲），TLB 中每一行都保存着一个页表项（PTE，Page Table Entry）。

复习：[TLB 的工作原理](#)

CPU 执行机构收到应用程序发来的虚拟地址后；

首先到 TLB 中查找相应的页表数据（CPU 根据虚拟地址前 20 位到 TLB 中查找），如果 TLB 中正好存放着所需的页表，则称为 TLB 命中（TLB Hit）；

接下来 CPU 再依次看 TLB 中页表所对应的物理内存地址中的数据是不是已经在一级、二级缓存（Cache）里了，若没有则到内存中取相应地址所存放的数据。

既然说 TLB 是内存里存放的页表的缓存，那么它里边存放的数据实际上和内存页表区的数据是一致的，在内存的页表区里，每一条记录虚拟页面和物理页框对应关系的记录称之为一个页表条目（Entry），同样地，在 TLB 里边也缓存了同样大小的页表条目（Entry）。

Q：为什么在内核里我们是使用 2M 页来映射呢？2M 页相对于 4K 页有什么好处？

A：不容易发送 page miss，且占用的空间比较少。一个 2M 页等于 512 个 4K 页，如果用 4K 的 page table entry 来放在 TLB 里的话空间一下子就没有了。比如说上述 `init_boot_pt` 中的循环，用 4K 页需要 512 次，用 2M 页只需要 1 次。

`init_boot_pt` 是我们配置页表项的函数。注意此时都是在物理地址上执行的，那么物理地址怎么无缝地切换到虚拟地址呢？也就是在这个函数中的 TTBR0 页表做的事情，它维护了虚拟地址到物理地址的一一映射，使得切换的时候是无缝的。

回到 init_c: 页表初始化

```

39 void init_boot_pt(void)
40 {
41     u32 start_entry_idx;
42     u32 end_entry_idx;
43     u32 idx;
44     u64 kva;
45
46     /* TTBR0_EL1 0-1G */
47     boot_ttbr0_l0[0] = ((u64)boot_ttbr0_l1) | IS_TABLE | IS_VALID;
48     boot_ttbr0_l0[0] = ((u64)boot_ttbr0_l2) | IS_TABLE | IS_VALID;
49
50     /* Usable memory: PHYSMEM_START ~ PERIPHERAL_BASE */
51     start_entry_idx = PHYSMEM_START / SIZE_2M;
52     end_entry_idx = PERIPHERAL_BASE / SIZE_2M;
53
54     /* Map each 2M page */
55     for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
56         boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
57             | UXN /* Unprivileged execute never */
58             | ACCESSED /* Set access flag */
59             | INNER_SHAREABLE /* Shareability */
60             | NORMAL_MEMORY /* Normal memory */
61             | IS_VALID;
62     }
63 }

```

```

17 #define INNER_SHAREABLE (0x3)
18 /* Please search mair_elt for these memory types. */
19 #define NORMAL_MEMORY (0x4)
20 #define DEVICE_MEMORY (0x0)

63 }
64
65 /* Peripheral memory: PERIPHERAL_BASE ~ PHYSMEM_END */
66
67 /* Rasp3b/3b+ Peripherals: 0x3f 00 00 00 - 0x3f ff ff ff */
68 start_entry_idx = end_entry_idx;
69 end_entry_idx = PHYSMEM_END / SIZE_2M;
70
71 /* Map each 2M page */
72 for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
73     boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
74         | UXN /* Unprivileged execute never */
75         | ACCESSED /* Set access flag */
76         | DEVICE_MEMORY /* Device memory */ // non-cachable
77         | IS_VALID;
78 }

```

映射完内存地址（左）后，映射设备地址（右）

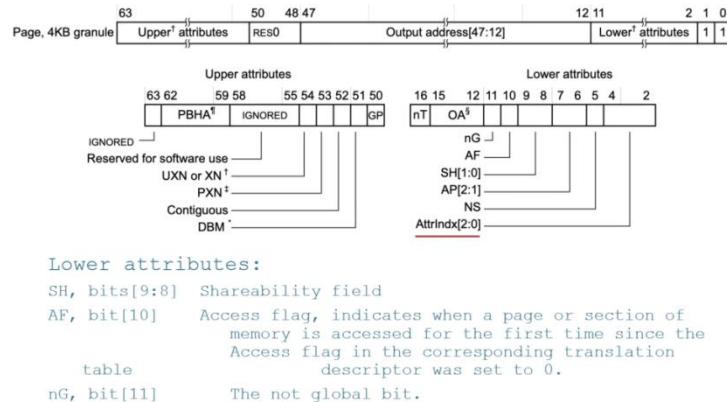
- 问：设备内存与物理内存有什么区别？
- 问：为什么要设置为 non-cachable？

Q: 设备内存和物理内存有什么区别？比如我们的网卡。

A: 它们是不能 cache 的。比如说我们有一个内存映射设备（比如一个 FPGA 返回一些数据包的情况），如果此时内存被 cache 了，那么 CPU 可能读到老的值，但是我们希望一直读到最新的值。

[1]<https://stackoverflow.com/questions/90204/why-would-a-region-of-memory-be-marked-non-cached>

Level 3 页表项



Figures from ARMv8 manual

回到 init_c: 页表初始化

```
80  /*
81   * TTBR1_EL1 0-1G
82   * KERNEL_VADDR: L0 pte index: 510; L1 pte index: 0; L2 pte index: 0.
83   */
84   kva = KERNEL_VADDR;
85   boot_ttbr1_l0[GET_L0_INDEX(kva)] = ((u64) boot_ttbr1_l1)
86   | IS_TABLE | IS_VALID;
87   boot_ttbr1_l1[GET_L1_INDEX(kva)] = ((u64) boot_ttbr1_l2)
88   | IS_TABLE | IS_VALID;
89
90   start_entry_idx = GET_L2_INDEX(kva);
91   /* Note: assert(start_entry_idx == 0) */
92   end_entry_idx = start_entry_idx + PHYSMEM_BOOT_END / SIZE_2M;
93   /* Note: assert(end_entry_idx < PTP_ENTRIES) */
94
95   /*
96    * Map each 2M page
97    * Usable memory: PHYSMEM_START ~ PERIPHERAL_BASE
98   */
99   for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
100     boot_ttbr1_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
101     | UXN /* Unprivileged execute never */
102     | ACCESSED /* Set access flag */
103     | INNER_SHARABLE /* Shareability */
104     | NORMAL_MEMORY /* Normal memory */
105     | IS_VALID;
106   }
107 }
```

```
1 #pragma once
2
3 #define SZ_16K      0x4000
4 #define SZ_64K     0x10000
5
6 #define KERNEL_VADDR 0xffffffff0000000000
7 #define TEXT_OFFSET 0x800000
8
```

对比设置 TTBR0: 方式相同，虚拟地址不同

```
55  /* Map each 2M page */
56  for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
57    boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
58    | UXN /* Unprivileged execute never */
59    | ACCESSED /* Set access flag */
60    | INNER_SHARABLE /* Shareability */
61    | NORMAL_MEMORY /* Normal memory */
62    | IS_VALID;           设置TTBR0页表 (低地址使用)
63 }
```

我们继续设置 TTBR1，我们可以发现一个物理空间被映射到了两个虚拟地址的区段。一个是一一映射的，另一个是映射到了高位地址，也就是 0xffffffff0000000000 这个位置。一个物理地址映射到了两个虚拟地址，这样子切换的时候就可以很自然地切换过去，这样我们的页表初始化就结束了。

el1_mmu_activate

页表设置完，开启翻译

```
66 void init_c(void)
67 {
68   /* Clear the bss area for the kernel image */
69   clear_bss();
70
71   /* Initialize UART before enabling MMU. */
72   early_uart_init();
73   uart_send_string("boot: init_c\r\n");
74
75   wakeup_other_cores(); // no need for qemu
76
77   /* Initialize Boot Page Table. */
78   uart_send_string("[BOOT] Install boot page table\r\n");
79   init_boot_pt();
80
81   /* Enable MMU. */
82   el1_mmu_activate(); // 用汇编写的函数，为什么？
83   uart_send_string("[BOOT] Enable el1 MMU\r\n");
84
85   /* Call Kernel Main. */
86   uart_send_string("[BOOT] Jump to kernel main\r\n");
87   start_kernel(secondary_boot_flag);
88
89   /* Never reach here */
90 }
```

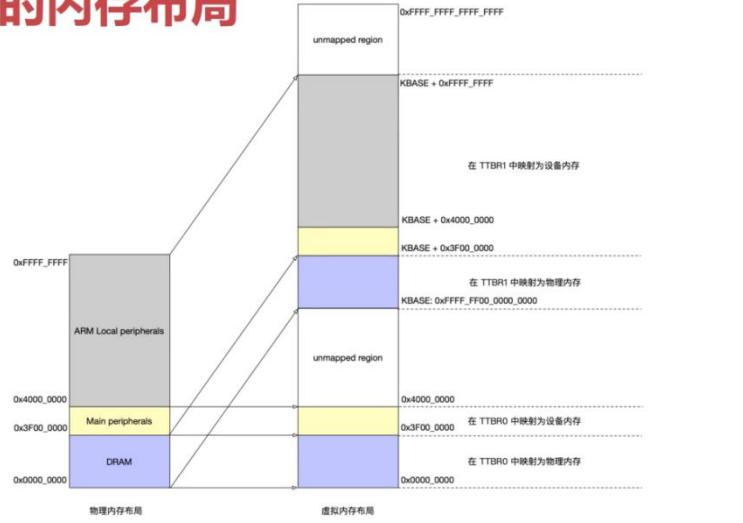
```
224 BEGIN_FUNC(el1_mmu_activate)
225   stp    x29, x30, [sp, #-16]!
226   mov    x29, sp
227
228   bl    invalidate_cache_all
229
230   /* Invalidate TLB */
231   tlb1i  vmallesis
232   isb
233   dsb   sy
234
235   /* Initialize Memory Attribute Indirection Register */
236   ldr    x8, =MMU_MAIR_ATTR1 | MMU_MAIR_ATTR2 | MMU_MAIR_ATTR3
237   msr    mair_el1, x8
238
239   /* Initialize TCR_EL1 */
240   /* set cacheable attributes on translation walk */
241   /* (SMP extensions) non-shareable, inner write-back write-allocate */
242   ldr    x8, =MMU_TCR_FLAGS1 | MMU_TCR_FLAGS0 | MMU_TCR_IPS | MMU_TCR_AS
243   msr    tcr_el1, x8 // translation control reg
244   isb
245
246   /* Write ttbr with phys addr of the translation table */
247   adrp  x8, boot_ttbr0_l0
248   msr    ttbr0_el1, x8          将页表的物理地址,
249   adrp  x8, boot_ttbr1_l0        写入 TTBR0 和 TTBR1
250   msr    ttbr1_el1, x8
251   isb
252
253   mrs    x8, sctlr_el1
254   /* Enable MMU */
255  orr    x8, x8, #SCTLR_EL1_M // set bit of MMU
256   /* Disable alignment checking */
257   bic    x8, x8, #SCTLR_EL1_A
258   bic    x8, x8, #SCTLR_EL1_SA 1. 将sctlr_el1寄存器写入x8
259   bic    x8, x8, #SCTLR_EL1_SA 2. 设置x8某些bit为1 (开关)
260  orr    x8, x8, #SCTLR_EL1_AA 3. 将x8写回sctlr_el1寄存器
261   /* Data accesses Cacheable */
262   /* Instruction access Cacheable */
263   /* Instruction access Cacheable */
264   orr    x8, x8, #SCTLR_EL1_I
265   msr    sctlr_el1, x8 // commit point
266
267   ldp    x29, x30, [sp], #16 // opposite to 226, push/pop
268   ret
269 END_FUNC(el1_mmu_activate)
```

Q: 为什么我们又开始使用汇编来写代码了呢？

A: 因为 C 不太好实现这个。

这个函数里主要就是把页表的物理地址写入到了 TTBR0 和 TTBR1 中，并且设置了 sctlr_el1 的一些位。把 x8 中的值写入了 sctlr_el1 以后，这就是一个 commit point。现在我们就开始使用虚拟地址，进入保护模式了。

此时的内存布局



开启页表前后

```
224 BEGIN_FUNC(e11_mmu_activate)
225     stp    x29, x30, [sp, #-16]!
226     mov    x29, sp
227
228     mrs    x8, sctlr_el1
229     /* Enable MMU */
230    orr    x8, x8, #SCTRL_ELL1_M // set bit of MMU
231     /* Disable alignment checking */
232     bic    x8, x8, #SCTRL_ELL1_A
233     bic    x8, x8, #SCTRL_ELL1_SA
234     bic    x8, x8, #SCTRL_ELL1_nA
235     orr    x8, x8, #SCTRL_ELL1_nA
236     /* Data accesses Cacheable */
237     orr    x8, x8, #SCTRL_ELL1_C
238     /* Instruction access Cacheable */
239     orr    x8, x8, #SCTRL_ELL1_I
240
241     msr    sctlr_el1, x8 // commit point 265时: 尚未使用页表
242
243     ldp    x29, x30, [sp] #16 // opposite to 226, push/pop
244     ret
245 END_FUNC(e11_mmu_activate) 267时: PC等地址已经过MMU翻译
246
247
```

执行267行，为何能顺利执行？

- 页表不是在低地址区域（0-1G）么？
- 虚拟地址和物理地址完全相同
- 回想下页表中的映射（TTBR1）

```
65
66 void init_c(void)
67 {
68     /* Clear the bss area for the kernel image */
69     clear_bss();
70
71     /* Initialize UART before enabling MMU. */
72     early_uart_init();
73     uart_send_string("boot: init_c\r\n");
74
75     wakeup_other_cores(); // no need for qemu
76
77     /* Initialize Boot Page Table. */
78     uart_send_string("[BOOT] Install boot page table\r\n");
79     init_boot_pt();
80
81     /* Enable MMU. */
82     e11_mmu_activate();
83     uart_send_string("[BOOT] Enable el1 MMU\r\n");
84
85     /* Call Kernel Main. */
86     uart_send_string("[BOOT] Jump to kernel main\r\n");
87     start_kernel(secondary_boot_flag);
88
89     /* Never reach here */
90 }
```

start_kernel位于高地址段：

- 0xffffffff0000000000 + init_end
- 从 init_c（低地址范围）跳过去后
- 高地址范围的地址已经被映射
- 栈在 start_kernel 已经换成了高地址

异常向量表初始化

异常向量表初始化

```
44
45 void main(void *addr)
46 {
47     /* Init uart */
48     uart_init();
49     kinfo("[ChCore] uart init finished\n");
50
51     kinfo("Address of main() is 0x%lx\n", main);
52     kinfo("123456 decimal is %o octal\n", 123456);
53
54     test_backtrace(5);
55
56     mm_init();
57     kinfo("mm init finished\n");
58
59     /* Init exception vector */
60     exception_init();
61     kinfo("[ChCore] interrupt init finished\n");
62
```

```
16 BEGIN_FUNC(set_exception_vector)
17     adr x0, e11_vector
18     msr vbar_e11, x0
19     ret
20 END_FUNC(set_exception_vector)
21
22
```

```
44 void exception_init(void)
45 {
46     exception_init_per_cpu();
47     memset(irq_handle_type, HANDLE_KERNEL, MAX_IRQ_NUM);
48 }
```

```
132 EXPORT(e11_vector)
133
134 sync_el1t:
135     handle_entry 1, SYNC_E11t
136
137 irq_el1t:
138     handle_entry 1, IRQ_E11t
139
140 fiq_el1t:
141     handle_entry 1, FIQ_E11t
142
143 error_el1t:
144     handle_entry 1, ERROR_E11t
145
146 sync_el1h:
147     handle_entry 1, SYNC_E11h
148
149 fiq_el1h:
150     handle_entry 1, FIQ_E11h
151
152 error_el1h:
153     handle_entry 1, ERROR_E11h
154
155 sync_el0_64:
156     /* Since we cannot touch x0-x7, we need some extra work here */
157     exception_enter
158     mrs x25, esr_el1
159     lsr x24, x25, #ESR_E11_EC_SHIFT
160     cmp x24, #ESR_E11_EC_SVC_64
161     b.eq _e10_syscall
162     /* Not supported exception */
163     mov x0, SYNC_E10_64
164     mrs x1, esr_el1
165     mrs x2, elr_el1
166     bl handle_entry_c
167     exception_return
168
169 e10_syscall:
```

```
29 .macro handle_entry el, type
30     exception_enter
31     mov x0, #type
32     mrs x1, esr_el1
33     mrs x2, elr_el1
34     bl handle_entry_c
35     exception_return
36 .endm
```

```
168 el0_syscall:
169 sub sp, sp, #16 * 8
170 stp x0, x1, [sp, #16 * 0]
171 stp x2, x3, [sp, #16 * 1]
172 stp x4, x5, [sp, #16 * 2]
173 stp x6, x7, [sp, #16 * 3]
174 stp x8, x9, [sp, #16 * 4]
175 stp x10, x11, [sp, #16 * 5]
176 stp x12, x13, [sp, #16 * 6]
177 stp x14, x15, [sp, #16 * 7]
178
179 bl lock_kernel
180
181 ldp x0, x1, [sp, #16 * 0]
182 ldp x2, x3, [sp, #16 * 1]
183 ldp x4, x5, [sp, #16 * 2]
184 ldp x6, x7, [sp, #16 * 3]
185 ldp x8, x9, [sp, #16 * 4]
186 ldp x10, x11, [sp, #16 * 5]
187 ldp x12, x13, [sp, #16 * 6]
188 ldp x14, x15, [sp, #16 * 7]
189 add sp, sp, #16 * 8
190
191 adr x27, syscall_table    // syscall table in x27
192 uxtw x16, w8             // syscall number in x16
193 ldr x16, [x27, x16, lsl #3] // find the syscall entry
194 btr x16
195
196
197 /* Ret from syscall */
198 // bl disable_irq
199 str x0, [sp] // set the return value of the syscall */
200 exception_return
```

```
191
192 adr x27, syscall_table    // syscall table in x27
193 uxtw x16, w8             // syscall number in x16
194 ldr x16, [x27, x16, lsl #3] // find the syscall entry
195 btr x16
196
197 /* Ret from syscall */
198 // bl disable_irq
199 str x0, [sp] // set the return value of the syscall */
200 exception_return
```

```
63 .macro exception_exit
64     ldp x11, x12, [sp, #16 * 16]
65     ldp x30, x10, [sp, #16 * 15]
66     msr sp_el0, x10
67     msr elr_el1, x11
68     msr spsr_el1, x12
69     ldp x0, x1, [sp, #16 * 0]
70     ldp x2, x3, [sp, #16 * 1]
71     ldp x4, x5, [sp, #16 * 2]
72     ldp x6, x7, [sp, #16 * 3]
73     ldp x8, x9, [sp, #16 * 4]
74     ldp x10, x11, [sp, #16 * 5]
75     ldp x12, x13, [sp, #16 * 6]
76     ldp x14, x15, [sp, #16 * 7]
77     ldp x16, x17, [sp, #16 * 8]
78     ldp x18, x19, [sp, #16 * 9]
79     ldp x20, x21, [sp, #16 * 10]
80     ldp x22, x23, [sp, #16 * 11]
81     ldp x24, x25, [sp, #16 * 12]
82     ldp x26, x27, [sp, #16 * 13]
83     ldp x28, x29, [sp, #16 * 14]
84     add sp, sp, #ARCH_EXEC_CONT_SIZE
85     eret
86 .endm
```

我们之前介绍过异常有很多种，如异常、中断等。右边这里在做的事情就是，先拿到 syscall 的 table 的基地址，再得到 syscall number，从而得到对应的 entry 跳转过去即可。

```
38 /* See more details about the bias in registers.h */
39 .macro exception_enter
40     sub sp, sp, #ARCH_EXEC_CONT_SIZE
41     stp x0, x1, [sp, #16 * 0]
42     stp x2, x3, [sp, #16 * 1]
43     stp x4, x5, [sp, #16 * 2]
44     stp x6, x7, [sp, #16 * 3]
45     stp x8, x9, [sp, #16 * 4]
46     stp x10, x11, [sp, #16 * 5]
47     stp x12, x13, [sp, #16 * 6]
48     stp x14, x15, [sp, #16 * 7]
49     stp x16, x17, [sp, #16 * 8]
50     stp x18, x19, [sp, #16 * 9]
51     stp x20, x21, [sp, #16 * 10]
52     stp x22, x23, [sp, #16 * 11]
53     stp x24, x25, [sp, #16 * 12]
54     stp x26, x27, [sp, #16 * 13]
55     stp x28, x29, [sp, #16 * 14]
56     mrs x10, sp_el0
57     mrs x11, elr_el1
58     mrs x12, spsr_el1
59     stp x30, x10, [sp, #16 * 15]
60     stp x11, x12, [sp, #16 * 16]
61 .endm
```

```
63 .macro exception_exit
64     ldp x11, x12, [sp, #16 * 16]
65     ldp x30, x10, [sp, #16 * 15]
66     msr sp_el0, x10
67     msr elr_el1, x11
68     msr spsr_el1, x12
69     ldp x0, x1, [sp, #16 * 0]
70     ldp x2, x3, [sp, #16 * 1]
71     ldp x4, x5, [sp, #16 * 2]
72     ldp x6, x7, [sp, #16 * 3]
73     ldp x8, x9, [sp, #16 * 4]
74     ldp x10, x11, [sp, #16 * 5]
75     ldp x12, x13, [sp, #16 * 6]
76     ldp x14, x15, [sp, #16 * 7]
77     ldp x16, x17, [sp, #16 * 8]
78     ldp x18, x19, [sp, #16 * 9]
79     ldp x20, x21, [sp, #16 * 10]
80     ldp x22, x23, [sp, #16 * 11]
81     ldp x24, x25, [sp, #16 * 12]
82     ldp x26, x27, [sp, #16 * 13]
83     ldp x28, x29, [sp, #16 * 14]
84     add sp, sp, #ARCH_EXEC_CONT_SIZE
85     eret
86 .endm
```

这里实际上就是在保存寄存器和恢复操作。

```
55 const void *syscall_table[NR_SYSCALL] = {  
56     [0 ... NR_SYSCALL - 1] = sys_debug,  
57     /* lab3 syscalls finished */  
58  
59     [SYS_getc] = sys_getc,  
60     [SYS_yield] = sys_yield,  
61     [SYS_create_device_pmo] = sys_create_device_pmo,  
62     [SYS_unmap_pmo] = sys_unmap_pmo,  
63     [SYS_create_thread] = sys_create_thread,  
64     [SYS_create_process] = sys_create_process,  
65     [SYS_register_server] = sys_register_server,  
66     [SYS_register_client] = sys_register_client,  
67     [SYS_ipc_call] = sys_ipc_call,  
68     [SYS_ipc_return] = sys_ipc_return,  
69     [SYS_cap_copy_to] = sys_cap_copy_to,  
70     [SYS_cap_copy_from] = sys_cap_copy_from,  
71     [SYS_set_affinity] = sys_set_affinity,  
72     [SYS_get_affinity] = sys_get_affinity,  
73     /* ... */  
74     [SYS_get_cpu_id] = sys_get_cpu_id,  
75  
76     [SYS_create_pmos] = sys_create_pmos,  
77     [SYS_map_pmos] = sys_map_pmos,  
78     [SYS_write_pmo] = sys_write_pmo,  
79     [SYS_read_pmo] = sys_read_pmo,  
80     [SYS_transfer_caps] = sys_transfer_caps,  
81  
82     /* TMP FS */  
83     [SYS_fs_load_cpio] = sys_fs_load_cpio,  
84  
85     [SYS_top] = sys_top,  
86     [SYS_debug] = sys_debug  
87 };  
  
267 /* syscalls */  
268 int sys_create_process(void)  
269 {  
270     struct process *new_process;  
271     struct vmspace *vmspace;  
272     int cap, r;  
273  
274     /* cap current process */  
275     new_process = obj_alloc(TYPE_PROCESS, sizeof(*new_process));  
276     if (!new_process) {  
277         r = -ENOMEM;  
278         goto out_fail;  
279     }  
};  
};
```

44

2022/3/1

内核启动前： BIOS 的作用

BIOS 不是一个 OS，它是 basic I/O system。

从计算机上电到内核开始运行



1. 上电后，开始执行BIOS ROM中的代码

- 自检 (POST: Power-On Self Test)
- 找到第一个可启动设备 (如第一块磁盘)
- 将可启动设备的第一个块 (512字节, 即MBR) 加载到内存0x7c00中
- 跳转到bootloader的内存地址 (物理地址0x7c00) 并继续执行

2. bootloader开始执行

- 将内核的二进制文件从启动设备加载到内存中
- 若内核文件是压缩包，则对其进行解压
- 跳转到 (解压后的) 内核加载地址 (物理地址) 并继续执行

3. 内核代码开始执行

ROM 就是写死的代码。功能比较固定，就是做自检，检查里面有没有出问题，再找到第一块可以启动的磁盘，并且加载进第一个块。一个 sector 是 512 Byte。MBR 实际上是一段代码，加载到 0x7c00 中开始执行。这样的话就是我们的启动的过程，然后我们的 BootLoader 再开始执行，进入到我们 OS 的代码中。启动设备可能就是一个 flash，会从我们的存储设备中加载到我们的内存里面。

这个过程中为什么会出现 BIOS？它的出现有一些历史原因。

BIOS (Basic Input/Output System)



- **什么是BIOS?**
 - 1981年8月，IBM生产了第一台个人电脑PC 5150，引入BIOS
 - BIOS通常保存在主板的只读内存 (ROM) 中
 - ROM仅仅是存储，没有执行能力
- **CPU负责执行BIOS**
 - x86 CPU在reset后，PC固定指向0xFFFF0
 - 0xFFFF0这个地址，就是BIOS的物理地址
- **在许多嵌入式设备中并没有BIOS**
 - 如：许多嵌入式ARM设备（但ARM服务器有BIOS）
 - 原因：设备简单 & 减少成本



x86的16位实模式

9

上电自检 (POST)

- **BIOS程序首先检查，计算机硬件能否满足运行的基本条件，这叫做“硬件自检” (Power-On Self-Test)，缩写为POST。**
- **如果硬件出现问题，主板会发出不同含义的蜂鸣，启动中止。如果没有问题，屏幕就会显示出CPU、内存、硬盘等信息。**

PCI Devices Listing ...									
Bus	Dev	Fan	Vendor	Device	SMID	SSID	Class	Device Class	IRQ
0	27	0	0006	2660	1450	0003	0003	Multimedia Device	5
0	29	0	0006	265B	1450	265B	0003	USB 1.1 Host Controller	9
0	29	1	0006	265C	1450	5006	0003	USB 1.1 Host Controller	11
0	29	2	0006	265A	1450	265A	0003	USB 1.1 Host Controller	11
0	29	3	0006	265B	1450	265B	0003	USB 1.1 Host Controller	5
0	29	7	0006	265C	1450	5006	0003	USB 1.1 Host Controller	9
0	29	8	0006	265D	1450	5006	0003	USB 1.1 Host Controller	14
0	31	3	0006	266A	1450	266A	0005	SCSI Bus Controller	11
1	0	0	100E	0421	100E	0479	0300	Display Controller	5
2	0	0	1203	0212	0000	0000	0100	Mass Storage Controller	10
2	5	0	11AB	0300	1450	0000	0000	Network Controller	12
								ACPI Controller	9

↓

10

如果自检的时候发现没有硬盘，那么哪怕机器启动起来了也没用。我们把各个厂商的接入设备通过 BIOS 进行检查，如果我们买了一个不符合标准的硬件，那么 BIOS 就不会通过自检，通过了以后才会显示 CPU 等信息。

MBR (master boot record)

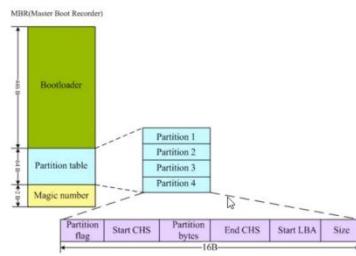
在 `winhex` 打开硬盘后，第一个起始的扇区页面就是硬盘的分区引导页面，也就是主引导记录，主引导记录 (MBR，全称为 Master Boot Record) 是采用 MBR 分区表的硬盘的第一个扇区，即 C/H/S 地址的 0 柱面 0 磁头 1 扇区，也叫做 MBR 扇区。

由于硬盘的主引导记录中仅仅为分区表保留了 64 个字节的存储空间，而每个分区的参数占据 16 个字节，故主引导扇区中总计只能存储 4 个分区的数据。也就是说，一块物理硬盘只能划分为 4 个主分区（主分区是一个比较单纯的分区，通常位于硬盘的最前面一块区域中，构成逻辑 C 磁盘。其中的主引导程序是它的一部分，此段程序主要用于检测硬盘分区的正确性，并确定活动分区，负责把引导权移交给活动分区的 DOS 或其他操作系统）磁盘。

[1]<https://zhuanlan.zhihu.com/p/423982362>

MBR (Master Boot Record)

- **MBR：主引导记录**
 - 磁盘的0柱面0磁头0扇区
称为主引导扇区
- **三部分组成**
 1. 主引导程序 (boot loader)
 2. 硬盘分区表DPT (disk partition table)
 3. 硬盘有效标志 (0x55AA)



我们约定 MBR 放在主引导扇区，首先启动。如果主引导扇区坏了怎么办？OS 就等于废了。

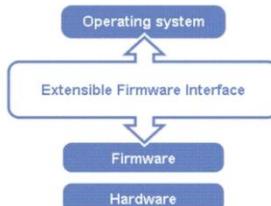
BIOS与生态

- **IBM为了PC兼容生态而设计了BIOS**
 - 主板必须兼容不同配件 (CPU、内存、网卡、硬盘等)
- **大部分嵌入式设备并没有BIOS**
 - 设备生产商负责所有的适配工作，不用过多考虑兼容性
 - 如：手机屏幕坏了，用户很难自己买一个屏幕更换
 - 可以节约更多成本 (省去了额外的ROM)
- **有没有BIOS，与是x86或ARM没有直接关系**
 - ARM服务器也有BIOS
 - 想一想：iPhone有没有BIOS？Macbook呢？Macbook M1呢？

苹果笔记本、Mac 电脑都是用 EFI 引导的，没有 BIOS。

EFI / UEFI

- **Intel提出来EFI取代BIOS interface**
 - EFI (Extensible Firmware Interface)
- **2005年，Intel再次提出UEFI取代EFI**
 - UEFI (Unified Extensible Firmware Interface)



它本质上还是在做 BIOS 在做的事情。EFI 可以改变启动扇区的配置，这样我们的零号扇

区一旦出问题，那么我们也可以修改配置启动一号扇区。

两种启动的对比

- **定制化的主板（常见的ARM开发板）**
 - 需要初始化具体主板相关硬件如GPIO和内存等
 - 初始化的时候就预先知道有哪些设备
 - 一般由厂商提供的BSP (Board Support Package)完成

- **通用的主板（常见的PC）**
 - 系统配置情况在开机时候是不知道的
 - 需要探测（Probe）、Training(内存和PCIe)和枚举（PCIe等等即插即用设备）
 - BIOS/EFI提供了整个主板、包括主板上外插的设备的软件抽象
 - 通过定义的接口把这些信息传递给OS，使OS不改而能够适配到所有机型和硬件

这个问题说明什么？



这个错误是找不到主引导分区，硬盘没插/挂了。

问题

- **Q：是不是所有的笔记本都需要BIOS？**
 - Chrome Book就没有用，用了core boot
 - 但core boot原名为LinuxBIOS，所以也算BIOS

问题

- **Q：如果一台计算机安装了2个操作系统，如何选择？**
 - Windows会在MBR安装NTLDR
 - Linux会在MBR安装grub，可以选择Windows或Linux
 - 如果先安装Linux，再安装Windows，会出什么问题？
 - 如何解决这个问题？用USB启动Linux重新安装grub
 - 如果安装了Mac OS，会怎么样？

Q: 先安装 Linux，再安装 Windows

A: MBR 被安装成了 NTLDR，我们发现 Linux 不见了。

解决方案：U 盘启动 Linux 安装 grub，grub 再加载 NTLDR，这样启动的时候就可以选择到底是启动 Windows 还是 Linux。

Q: 安装 Mac OS 会怎么样？

A: 我们需要使用 Mac 的引导程序把 Windows/Linux 装进去。

ARM设备启动的特点

- **通常与设备强相关**

- 厂商提供私有固件（firmware）用于初始化
 - 一般称为BSP：Board Support Package
 - 不同厂商的方案往往相差很大

- **缺点：对可插拔外设的兼容性**

- ARM嵌入式设备一般不支持PCIe等外设

ARM 最早的设计是嵌入式设备，兼容性较差，每个厂商提供了私有的固件用于初始化。

虚拟内存

我们今天先回顾一下虚拟内存，以及从 OS 角度怎么看待虚拟内存。最早期的计算机物理内存很小，软件也很简单。

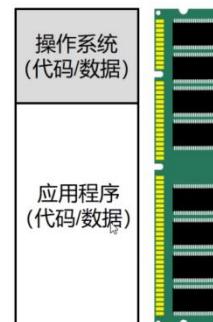
最早期的计算机系统

- **硬件**

- 物理内存容量小

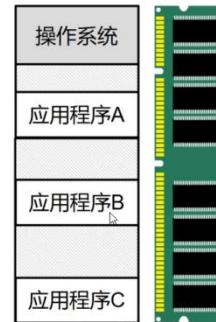
- **软件**

- 单个应用程序 + (简单) 操作系统
 - 直接面对物理内存编程
 - 各自使用物理内存的一部分



多道程序 (Multiprogramming) 时代

- 多用户多程序
 - 计算机很昂贵，多人同时使用（远程连接）
- 分时复用CPU资源
 - 保存恢复寄存器速度很快
- 分时复用物理内存资源
 - 将全部内存写入磁盘开销太高
- 同时使用、各占一部分物理内存
 - 没有安全性（隔离性）



如何让OS与不同的应用程序都高效又安全地使用物理内存资源？

21

ARM 的 M 系列都是没有虚拟内存的，不提供 MMU 支持。这是因为成本的原因，本来就非常小、用途非常单一。其实没有虚拟内存的设备的数量是远远大于有虚拟内存的设备的。

内存隔离：Protection Key

IBM OS/360的内存隔离：Protection Key

- Protection key机制
 - 内存被划分为一个个大小为2KB的内存块（Block）
 - 每个内存块有一个4-bit的key，保存在寄存器中
 - 1MB内存需要256个保存key的寄存器，占256-Byte
 - 内存变大怎么办？需要改CPU以增加key寄存器...
 - 每个进程对应一个key
 - CPU用另一个专门的寄存器，保存当前运行进程的key
 - 不同进程的key不同
 - 一个进程访问一块内存时
 - CPU检查进程的key与内存的key是否匹配



Protection Key机制的挑战

- 应用加载与隔离
 - 不同应用被加载到不同的物理地址段
 - 不同应用的key不同，以保证隔离
- 问题
 - 同一个二进制文件，程序-1加载到0000-1000地址段，程序-2加载到5000-6000地址段
 - "JMP 42"，程序-1能执行，程序-2会出错
- 解决方法
 - 代码中所有地址在加载过程中都需要增加一个偏移量，如改为："JMP 5042"
 - 新的问题：
 - 加载过程变得更慢
 - 如何在代码中定位所有的地址？如 "MOV REG1, 42"，其中的42是地址还是数据？



使用物理地址的缺点

- 物理地址对应用是可知的，导致：
 - 一个应用会因其他应用的加载而受到影响
 - 一个应用可通过自身的内存地址，猜测出其他应用的加载位置
- 是否可以让应用看不见物理地址？
 - 不用关心其他进程，不受其他进程的影响
 - 看不见其他进程的信息，更强的隔离能力

虚拟地址 VS. 物理地址

- 虚拟内存抽象下，程序使用虚拟地址访问主存
 - 虚拟地址会被硬件“自动地”翻译成物理地址
- 每个应用程序拥有独立的虚拟地址空间
 - 应用程序认为自己独占整个内存
 - 应用程序不再看到物理地址
 - 应用加载时不用再为地址增加一个偏移量

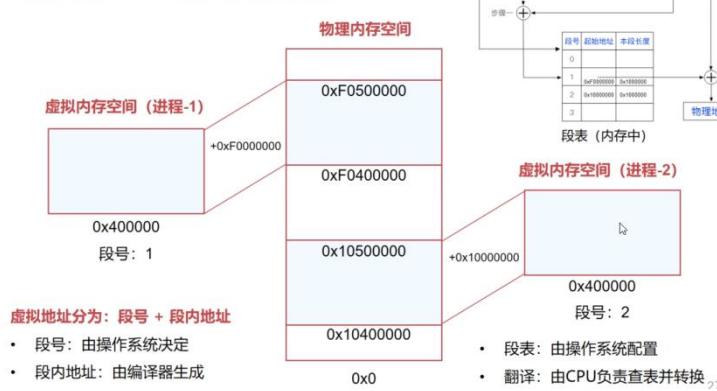
地址翻译过程



翻译规则取决于虚拟内存采用的组织机制，包括：分段机制和分页机制

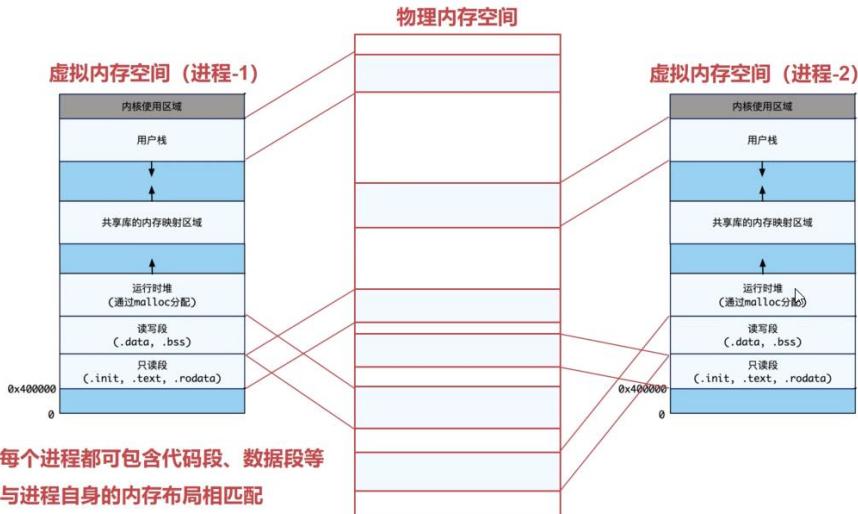
虚拟地址翻译过程：分段机制

方法-1：分段机制



`protection_key` 带来了程序中加载的位置的偏移，一种自然而然的解决方法就是使用分段的方式。

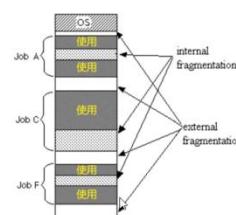
方法-1：分段机制（更细粒度）



这样我们进程的虚拟内存布局也是一系列的段所构成的。

分段机制的问题

- 对物理内存连续性的要求**
 - 物理内存也必须以段为单位进行分配
- 存在问题：内存利用率**
 - 外部碎片：段与段之间留下碎片空间
 - 内部碎片：段内部预留未使用的碎片空间
- 分段机制常见于x86平台**
 - 现代操作系统通常不再依赖分段
 - x86-64在64位模式放弃了段机制



但是仅仅使用段的话，我们必须以段为单位进行分配内存，就会带来“段”到底应该设

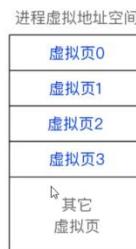
置多大，太大就会浪费空间，没有被充分用起来就是内部碎片。

虚拟地址翻译过程：分页机制

方法-2：分页机制

- 分页机制
 - 虚拟地址空间划分成连续的、等长的虚拟页
 - 物理内存也被划分成连续的、等长的物理页
 - 虚拟页和物理页的页长相等
 - 虚拟地址分为：虚拟页号 + 页内偏移

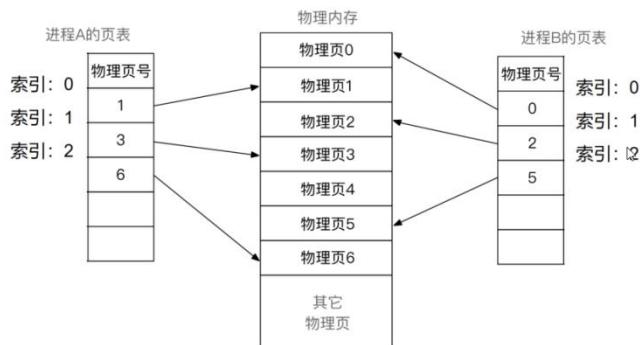
- 使用**页表**记录虚拟页号到物理页号的映射
 - 页表：Page Table



第二种方式就是分页的机制，划分成等长的虚拟页和物理页，再使用页表记录虚拟页到物理页的映射。

页表：分页机制的核心数据结构

- 页表包含多个页表项，存储物理页的页号（虚拟页号为索引）



特点：

分页机制的特点

- 物理内存离散分配
 - 任意虚拟页可以映射到任意物理页
 - 大大降低对物理内存连续性的要求
- 主存资源易于管理，利用率更高
 - 按照固定页大小分配物理内存
 - 能大大降低外部碎片和内部碎片
- 被现代处理器和操作系统广泛采用

其实现在还有大量的处理器和操作系统是没有虚拟内存的机制的。

ARM64 的页表格式

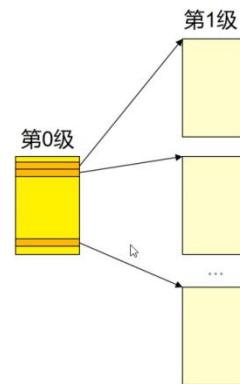
接下来我们来关注 ARM64 怎么来实现页表的格式。

多级页表

- 多级页表能有效压缩页表的大小

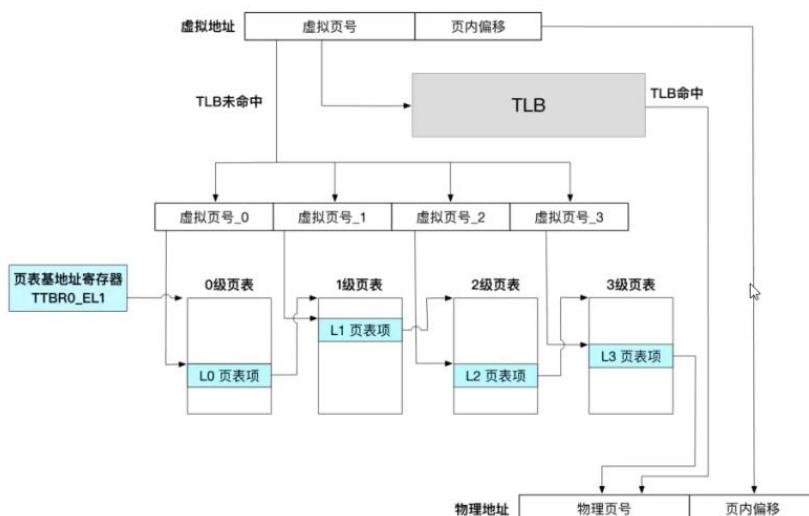
- 原因：允许页表中出现空洞

- 若某级页表中的某条目为空，那么该条目对应的下一级页表便无需存在
- 应用程序的虚拟地址空间大部分都未分配



一个应用程序的大部分虚拟内存都是没有被分配的，所以多级页表可以有效地减少虚拟内存所对应的页表的开销。

AARCH64体系结构下4级页表



这和 X86 类似，页表基址寄存器是 TTBR0_EL1。0 是代表对应用户的，EL1 是代表由内核来管理的。

Q: 有没有 TTBR0_ELO 呢？

A: 没有，页表是由内核来管理的。

64位虚拟地址解析

- 「63: 48」 16-bit
 - 必须全是0或者全是1（一般应用程序地址选0，内核地址选1）
 - 也意味着虚拟地址空间大小最大是 2^{48} Byte，即256TB
- 「47: 39」 9-bit: 0级页表索引
- 「38: 30」 9-bit: 1级页表索引
- 「29: 21」 9-bit: 2级页表索引
- 「20: 12」 9-bit: 3级页表索引
- 「11: 0」 12-bit: 页内偏移

前 16 位的 0 和 1 分开了用户态地址和内核态地址，每一个页表有 $2^9 = 512$ 个页表项。

页表基址寄存器

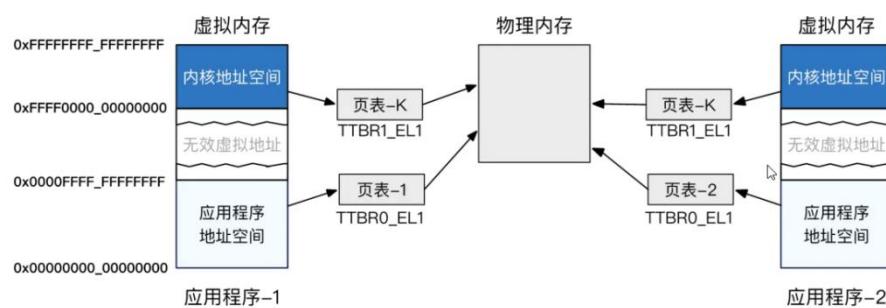
- AARCH64
 - 两个寄存器: TTBR0_EL1 & TTBR1_EL1
 - 根据虚拟地址第63位选择
 - 以Linux为例
 - 应用程序（地址首位为0）：使用TTBR0_EL1
 - 操作系统（地址首位为1）：使用TTBR1_EL1
- X86_64
 - 一个寄存器: CR3 (Control Register 3)

Q: 两个页表寄存器有什么好处？

A: 18 年之前，用户态和内核态都用的 CR3，这样切换的时候不需要修改 CR3。但是 18 年出了一个很大的漏洞，Meltdown BUG。ARM 用两个页表也没有上下文切换的开销，安全性也好。

AArch64中的两个页表基址寄存器

- TTBR0_EL1和TTBR1_EL1分别翻译部分虚拟地址



Level 3 页表项

63	51 50	48 47	Output address[47:12]	12 11	2 1 0
	Upper [†] attributes	RES0		Lower [†] attributes	1 1

• 第3级页表页中的页表项

- 第0位 (valid位) 表示该项是否有效
- 第1位必须是1
- Upper attributes包括：
 - 第54位 (XN位) 为1表示EL0不能执行 (execution never)
 - 第53位 (PXM位) 为1表示EL1不能执行
 - 第51位 (DBM位) 为1表示该页被修改了 (dirty bit)

– Lower attributes包括：

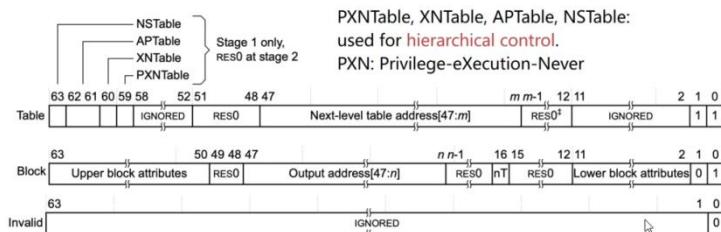
- 第7位-第6位表示读写权限位AP[2:1]

AP[2:1]	Access from higher Exception level	Access from EL0
00	Read/write	None
01	Read/write	Read/write
10	Read-only	None
11	Read-only	Read-only

- 第10位 (AF位) 是Access Flag, 若为0访问时发生异常 (可供软件追踪内存访问情况)
- 第9位-第8位是Shareability field
- 第4位-第2位是AttrIndx[2:0], 表示内存类型
 - Normal (其cacheable属性由TCR_EL1指定)
 - Device (再细分四种)

首先对应了它的一些属性，为什么把这些位用来做属性呢？因为这些都不用。对页表来说也没有页内偏移，所以也可以用来作为属性位。如 XN, PXM, Dirty Bit。权限位可以设置 copy-on-write 等。

Level 0, Level 1, Level 2 页表项



bit[0] 是valid位 (1代表有效页表项)

bit[1] 是表示页表项类型：

0表示指向1GB、2MB大页 (Super Page)

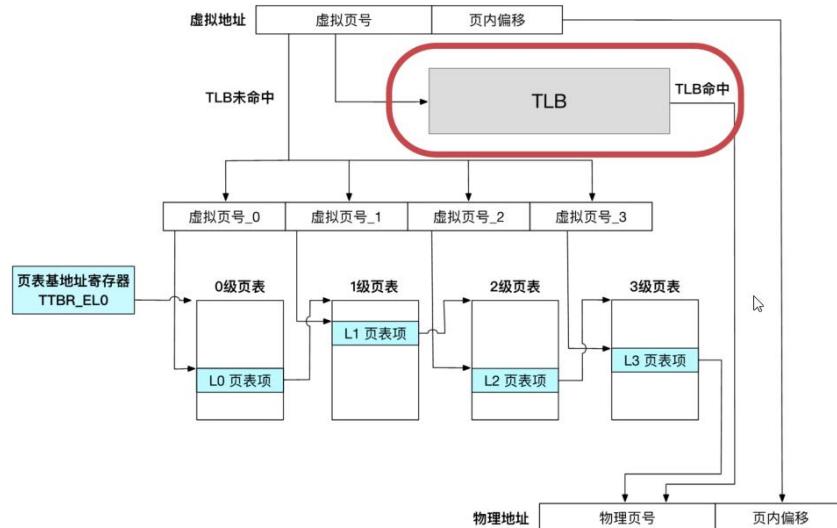
1表示指向下一级页表

这些格式大家在做 lab 的时候要认真地对照一下。

TLB：页表的 CACHE

TLB 的位置大家也非常熟悉，ICS 也考过大家 TLB 怎么翻译的，原先我们看到的 TLB 是一级的，在 ARM 也有多级的。

TLB：地址翻译的加速器



TLB：地址翻译的加速器

- **TLB 位于CPU内部，是页表的缓存**
 - Translation Lookaside Buffer
 - 缓存了虚拟页号到物理页号的映射关系
 - **有限数目的**TLB缓存项
- **在地址翻译过程中，MMU首先查询TLB**
 - TLB命中，则不再查询页表（**fast path**）
 - TLB未命中，再查询页表

翻译过程就是查询 TLB，如果 TLB 命中了就直接返回。

TLB刷新 (TLB Flush)

- **TLB 使用虚拟地址索引**
 - 当OS切换页表时需要全部刷新
- **AARCH64上内核和应用程序使用不同的页表**
 - 分别存在TTBR0_EL1和TTBR1_EL1
 - 系统调用过程不用切换
- **刷新TLB的相关指令**
 - 清空全部
 - TLBI VMALLEL1IS
 - 清空指定ASID相关
 - TLBI ASIDE1IS
 - 清空指定虚拟地址
 - TLBI VAE1IS

一般切换页表的时候，我们需要 flush TLB。上下文切换的时候，我们换了新的虚拟地址，相同虚拟地址对应的物理地址应该是不一样的，所以我们需要清空 TLB 中已有的值。

发生进程切换，每次都要刷 TLB，然后加载进来，这样效率比较低，所以现代处理器加了一个 TLBI ASIDE1IS。ASID 就是虚拟地址空间，这就意味着我们切换进程的时候不一定要刷 TLB。

还有可以指定一个虚拟地址的 TLB 刷掉。

ASID 来降低 TLB 刷新导致的开销

这些指令的话，大家在设计 OS 的时候来应用在不同的机制里。

如何降低TLB刷新导致的开销

- 新的硬件特性 ASID (AARCH64) : Address Space ID
 - OS 为不同进程分配 8 位或 16 位 ASID
 - ASID 的位数由 TCR_EL1 的第 36 位 (AS 位) 决定
 - OS 会将 ASID 填写在 TTBR0_EL1 的高 8 位或高 16 位
 - TLB 的每一项也会缓存 ASID
 - 地址翻译时，硬件会将 TLB 项的 ASID 与 TTBR0_EL1 的 ASID 对比
 - 若不匹配，则 TLB miss
- 使用了 ASID 之后
 - 切换页表（即切换进程）后，不再需要刷新 TLB，提高性能
 - 修改页表映射后，仍需刷新 TLB（为什么？）

ASID 的位数是有 AS 位决定的，使用了 ASID 以后，切换进程的时候就不需要再刷新 TLB。

Q: 我们修改页表映射的时候，为什么还是需要刷新 TLB？

A: 因为页表变了，TLB 里的映射还是原来的映射，所以需要刷新 TLB。

Q: 如果硬件里支持 8 位的 ASID，它能够支持多少进程？

A: $2^8 = 256$ 个。

Q: 但是我们 OS 中的进程数量远远不止 256 个（高峰有 8000~10000 个进程），怎么办？

A: 我们可以把最常用的进程维护一个 ASID，不重要的直接刷掉。这样就需要进程和 ASID 的映射，我们可以把 ASID 当做进程的 CACHE，前 255 个对应重要的进程，最后 1 个 ASID 被所有不重要的进程共享，切换的时候刷 TLB。但是这也有很多问题，里面有一个动态变化的过程，我们要跟踪中间的过程。

TLB 与多核

TLB 与多核

• 在多核场景下

- OS 修改页表后，需要刷新其它核的 TLB 吗？
- OS 如何知道需要刷新哪些核的 TLB？
- OS 如何刷新其它核的 TLB？

• OS 修改页表后，需要刷新其它核的 TLB 吗？

- 需要，因为一个进程可能在多个核上运行

• OS 如何知道需要刷新哪些核的 TLB？

- 操作系统知道进程调度信息

• OS 如何刷新其它核的 TLB？

- x86_64: 发送 IPI 中断某个核，通知它主动刷新
- AARCH64: 可在 local CPU 上刷新其它核 TLB
 - 调用的 ARM 指令：TLBI ASIDE1IS

虚拟内存：段和 VMA

我们继续回来讲段。

Segmentation fault

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char *p = NULL;
6     printf("%s\n", p);
7     return 0;
8 }
9
10 The output after the execution is like :
11 Segmentation fault (core dumped)
```

如果改变第5行会怎么样？比如：
char *p = 0x40000;
char *p = 0x400000000;



这个大家很熟悉了，这个就是 segmentation fault。往空指针里去写。

应用是否有权访问整个虚拟地址空间？

- OS采用段来管理虚拟地址
 - 段内连续，段与段之间非连续
 - 合法虚拟地址段：代码、数据、堆、栈
 - 非法虚拟地址段：未映射
 - 一旦访问，则触发segfault
- 思考：为什么要用段来管理？

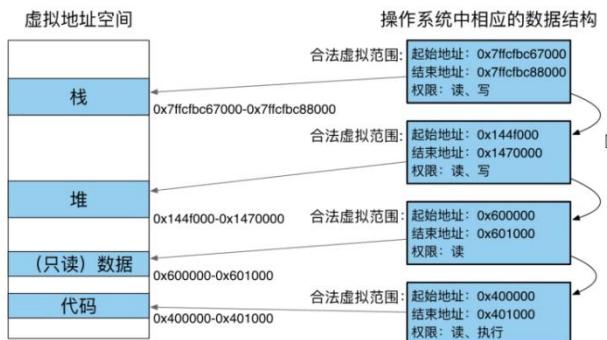


Q: 现在硬件已经没有段机制了，为什么还是 segmentation fault？

A: OS 还是使用段来管理虚拟地址。

合法虚拟地址信息的记录方式

- 记录应用程序已分配的虚拟内存区域
 - 在Linux中对应 vm_area_struct (VMA) 结构体



Q: 为什么要用段来管理呢？用页不是也可以吗？

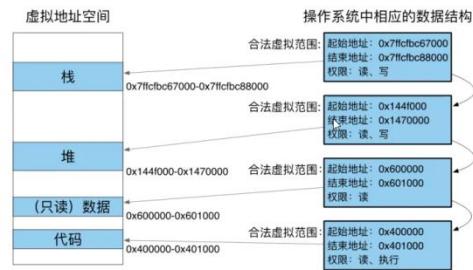
A: 会快一点。

怎么记录虚拟内存分配的空间的话，就需要使用 VMA 来记录合法的虚拟范围。一个进程远远不止我们看到的这 4 个段，至少有几十个 VMA。

VMA 是如何添加的

- 途径-1: OS在创建应用程序时分配

- 数据 (对应ELF段)
- 代码 (对应ELF段)
- 栈 (初始无内容)



- 途径2: 应用程序主动向OS发出请求

- brk() (扩大、缩小堆区域)
 - 可选策略: OS也可以在创建应用时分配初始的堆VMA
- mmap()
 - 申请空的虚拟内存区域
 - 申请映射文件数据的虚拟内存区域

- 用户态的malloc会改变VMA

- 通常是调用brk, 在堆中分配新的内存
- 部分实现也可以调用mmap, 由应用管理多个VMA

brk 就是我们的 malloc, 如果堆的扩大和缩小也会修改 VMA。

mmap: 分配一段虚拟内存区域

- 通常用于把一个文件 (或一部分) 映射到内存

```
void *mmap(void *addr,           // 起始地址
           size_t length,        // 长度
           int prot,              // 权限, 例如PROT_READ
           int flags,             // 映射的标志, 例如MAP_PRIVATE
           int fd,                // -1 或者是有效fd
           off_t offset)          // 偏移, 例如从文件的哪里开始映射
```

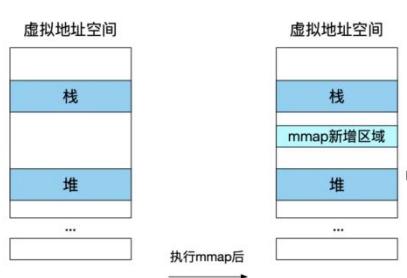
- 也可以不映射任何文件, 仅仅新建虚拟内存区域 (匿名映射)

- 注意: 匿名映射并非POSIX标准, 但主流OS都会支持

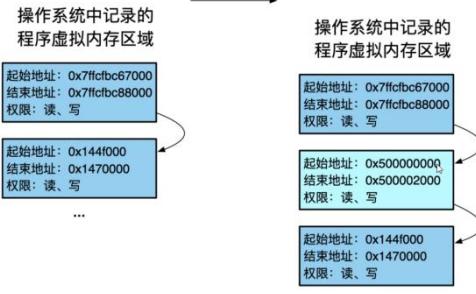
mmap匿名映射

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <sys/mman.h>
4
5 // void *mmap(void *addr, size_t length, int prot, int
6   → flags, int fd, off_t offset);
7
8 int main()
9 {
10    char *buf;
11
12    buf = mmap((void *)0x5000000000, 0x2000, PROT_READ |
13      → PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
14    printf("mmap returns %p\n", buf);
15    strcpy(buf, "Hello mmap");
16    printf("%s\n", buf);
17
18    return 0;
19
20 The output after the execution is like :
21 mmap returns 0x5000000000
22 Hello mmap
```

执行mmap后，VMA的变化



执行mmap后，VMA的变化



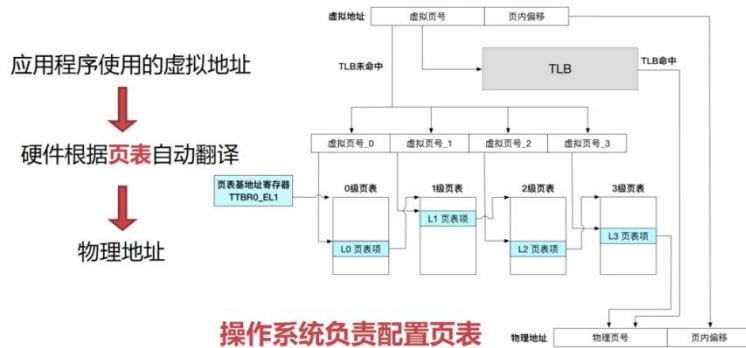
实验的提交与评分

- 具体细节见各个实验要求
- 只修改文档中提示修改的文件内容
 - 当然如果你想整点花活，也可以自由发挥
 - 但实验考查的练习题必须完成
- 评分：代码 80% + 文档 20%
 - 代码的满分都是 100 分，最后加权计算
 - make grade 不一定能完全体现真实代码得分
 - 文档不要为了字数而灌水

2022/3/3

我们上节课讲了我们需要虚拟内存，以及对虚拟内存的管理。使用 protection key 的方式实现对物理内存的保护。并且在现代 OS 中还存在段的概念，段映射下来实际上就是我们的 VMA。unmap 的过程会把现有的 VMA 进行分裂。

回顾：应用程序仅使用虚拟地址



应用程序的执行只有虚拟地址，硬件会根据页表自动翻译成物理地址。页表维护了虚拟地址到物理地址的映射？

问：VMA和页表是否冗余？

- OS通过VMA记录应用程序能够访问的虚拟地址
 - 未映射的区域没有对应的VMA结构
- OS通过配置页表控制应用程序能够访问的虚拟地址
 - 未分配的虚拟地址没有对应的页表
- 那么，VMA是否冗余？

Q: Segmentation Fault 就是没有对应的 VMA。OS 可以通过配置页表来控制应用程序可以访问的虚拟地址，如果访问未分配的虚拟地址，也会报错。那么我们为什么要引入 VMA 呢？

A: 不冗余，只是从判断一个应用程序能不能访问虚拟地址来说，是冗余的。比如说应用程序把所有的页一开始都映射好。

操作系统何时为应用程序填写页表

- 两种方式
 - 立即映射：每个虚拟页都对应了一个物理内存页
 - 延迟映射：有些虚拟页不对应任何物理内存页
 - 对应的数据在磁盘上
 - 没有对应的数据（初始化为0）

可以不用 VMA，因为页表记录了完整的映射，没有页表则表示没有映射。但 VMA 还是可以提高速度，比遍历页表更快。

延迟映射/按需映射

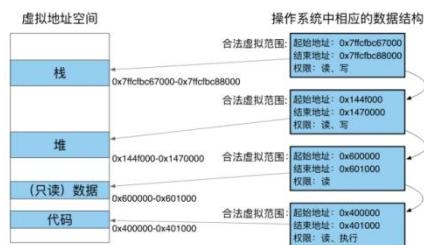
缺页异常 (Page Fault)

- CPU的控制流转移
 - CPU陷入内核，找到并运行相应的异常处理函数 (handler)
 - OS提前注册缺页异常处理函数
- x86_64
 - 异常号 #PF (13)，错误地址存放在CR2寄存器
- AARCH64
 - 触发 (通用的) 同步异常 (8)
 - 根据ESR信息判断是否缺页，错误地址存放在FAR_EL1

CPU 会根据 exception table 中的 handler 得到缺页异常的处理函数。

如何判断缺页异常的合法性？

- OS记录应用程序已分配的虚拟内存端 (VMA)
 - 不落在VMA区域，则为非法



Q: 怎么让访问空指针是合法的呢？

A: 给虚拟地址创建一个 VMA 的映射。但是程序运行的时候会很困扰，访问空地址也会运行下去会导致程序出现不可预测的行为。

大家学过数据结构，如果只有几项 VMA，用链表和数组维护就可以了，但是如果 VMA 项比较多，就要用到复杂的数据结构，如红黑树等。

按需分配考虑的权衡

- 优势：节约内存资源
- 劣势：缺页异常导致访问延迟增加
- 如何取得平衡?
 - 应用程序访存具有时空局部性 (Locality)
 - 在缺页异常处理函数中采用预取 (Prefetching) 机制
 - 即节约内存又能减少缺页异常次数

对于程序的访存，也是存在时空局部性的，所以我们利用这个局部性来做 prefetching 机制。来处理 Hard Page Fault 的时候，要从磁盘里读一个页的数据的时候，如果它访问的是第一个页，那么我们认为它大概率也会访问后续页，所以我们一下子把 8 个页读入内存中。

对于磁盘来说，一次读 4K 和一次读 32K，时间差别不会特别大。到底 prefetch 进来几个页，需要根据场景来调优。核心点就是我们的策略怎么设计。

思考

- 导致缺页异常的三种可能

1. 访问非法虚拟地址
2. 按需分配（尚未分配真正的物理页）
3. 内存页数据被换出到磁盘上

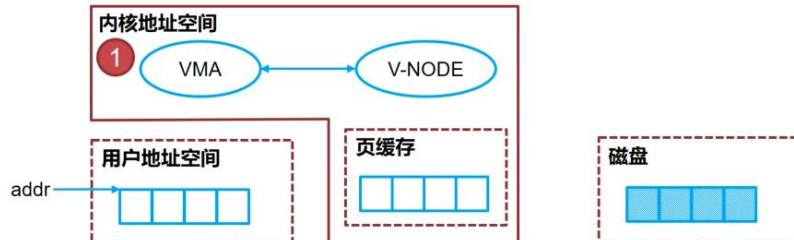
– 问：后两种都是合法的缺页异常，如何区分？

A：我们可以用 VMA 来区分，第一种情况，不在 VMA 区域；第二种和第三种，在 VMA 区域。

区分后两种的思路是：

1. 记住有哪些页被换出，比如 hash 表可以解决。
2. 更简单的方法：通过页表项某些位是否为 NULL 进行区分。（OS 自己定义规则：NULL 代表非法，非 NULL 代表 SWAP。）

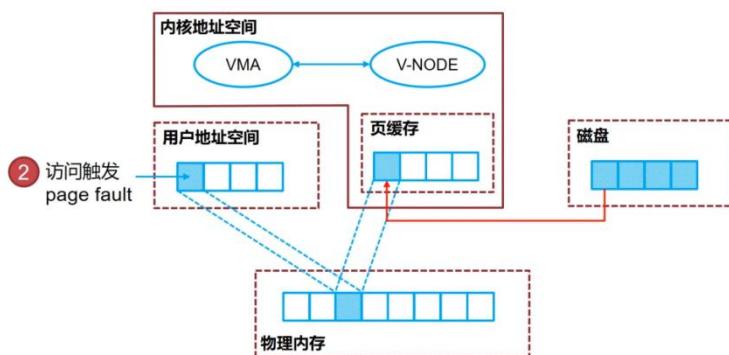
MMAP



调用 mmap 映射文件时，内核中会为 mmap 的区域创建一个 vma，并且将对应标识文件的内核对象 V-NODE 和这个 VMA 绑定。此时这块区域的页表中没有任何物理页。

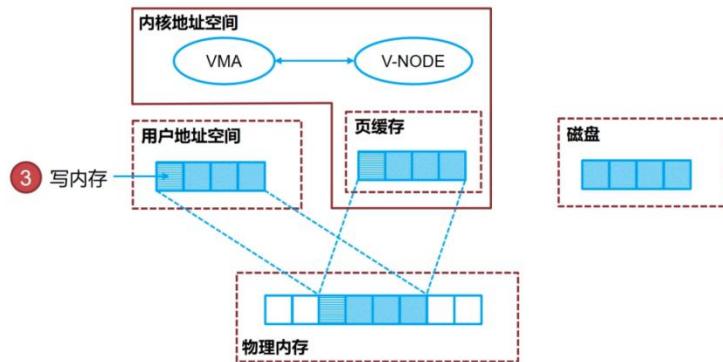
VMA 可以记录它到底是记录第二种情况还是第三种情况，如果是第三种情况，对应的磁盘的位置在哪。

MMAP



当访问这块内存时，会触发一个 page fault，内核会检查 fault 地址，发现其对应的 vma 绑定了一个 v-node，就会从磁盘读取对应的页内容到页缓存，并将页缓存与用户地址空间的内容映射到同一个物理页。

MMAP



对 mmap 内存区域的写会直接修改页缓存中的内容，但是此时并没有刷新到磁盘里去。当文件被关闭，或者调用 msync 时，页缓存中修改的数据会被刷回磁盘。我们编辑文档的时候的 **ctrl+s**，其实就是调用了 msync。

OS 需要来猜/学习用户的行为，但是并不知道用户想干什么，所以 OS 提供了一些接口上的标记来让用户告诉 OS 怎么管理内存。

MMAP 的优化

- **Prefault**
 - 每次 page fault 载入连续的多个页，减少 fault 次数
- **MAP_POPULATE**
 - 通过一个包含 MAP_POPULATE 的 flags 参数，可以在调用 mmap 时就预取所有的页，此后访问不会 fault

MAP_POPULATE 就是在调用 mmap 的时候直接映射所有的区域，直接预取所有的页，比如这个应用在 mmap 完直接会把映射的所有页访问一遍。

思考

- **问：mmap匿名映射与文件映射的区别是什么？**
 - 没有 backup file，内存的初始值从哪里来？
- **问：如果OS仅采用立即映射，还需要VMA么？**
 - VMA记录的信息和页表记录的信息有何不同？
- **问：demand-paging是否可通过网络来实现？**
 - 如果都通过网络，本地是否还需要磁盘？

A1: mmap 的 fd 如果标记为 -1，那么就没有对应的 backup file，就是一个匿名映射。初始值就是全是 0。

Q2: 如果 OS 只使用立即映射，还需要 VMA 吗？

A2: 可以不需要，但是页表记录了映射的权限信息，而 VMA 会记录匿名映射、文件映射的

类型，以及对应的文件在哪里，VMA 查找更快。

Q3: Demand-paging 是否可以通过网络来实现？

A3: 可以通过网络，不需要磁盘，实现无盘工作站。

小知识：OS可向应用提供灵活的内存管理API

- **madvise**

- `int madvise(void *addr, size_t length, int advice)`
- 将用户态的一些语义信息发给内核以便于优化
 - 例如：将madvise和mmap搭配，在使用数据前告诉内核这一段数据需要使用，**建议**OS提前分配物理页，减少缺页异常开销

- **mprotect**

- `int mprotect(void *addr, size_t len, int prot);`
- 改变一段内存的权限
 - 例如：JIT动态生成的二进制代码，需将内存由“可写”改“为可执行”

我们可以看到还有很多其他的接口，比如 **madvise** 用户告诉 OS 你应该怎么做。我们可以告诉 OS 这一块的内存是连续访问、随机访问还是其他模式。**mprotect** 就是可以改变一段内存的权限。

换页机制

可将物理内存看做是虚拟地址空间的Cache

- **情景1：**

- 两个应用程序各自需要使用 3GB 的物理内存
- 机器实际上总共只有 4GB 的物理内存

- **情景2：**

- 一个应用程序申请预先分配足够大的（虚拟）内存
- 实际上其中大部分的虚拟页最终都不会用到

换页机制 (Swapping)

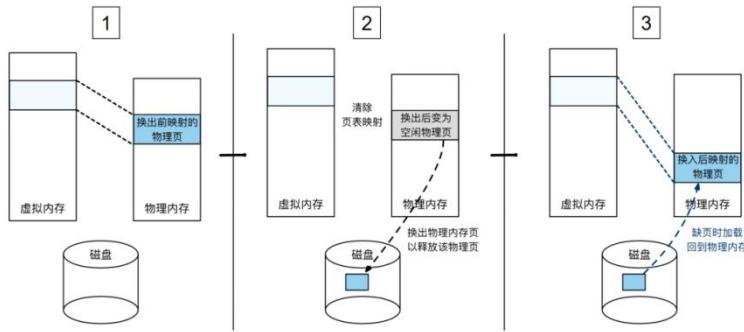
- **换页的基本思想**

- 用磁盘作为物理内存的补充，且对上层应用透明
- 应用对虚拟内存的使用，不受物理内存大小限制

- **如何实现**

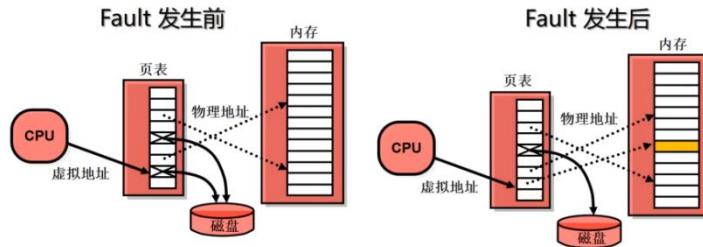
- 磁盘上划分专门的Swap分区，或专门的Swap文件
- 在处理缺页异常时，触发物理内存页的换入换出

简单的换页示例



Page Faults

- 换页 (Swapping)
- 页面分配 (Paging)
- 页面按需分配 (Demand paging)



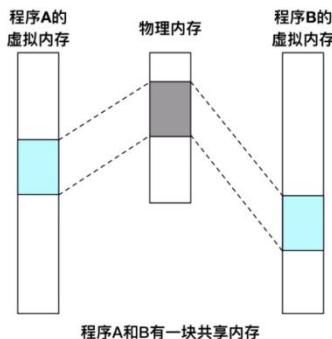
Q: 手机里内存不够了怎么办?

A: 通过换出的方式。所以华为前年发布的手机，可以让 8G 内存等价于 12G。

OS 内存管理中的更多机制

共享内存

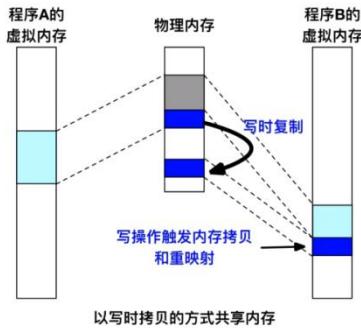
- 基本功能
 - 节约内存，如共享库
 - 进程通信，传递数据



我们会通过两个虚拟内存映射成同一个物理页的方式节约内存，典型的就是 fork。

写时拷贝 (copy-on-write)

- 实现
 - 修改页表项权限
 - 在缺页时拷贝、恢复
- 典型场景 fork
 - 节约物理内存
 - 性能加速



内存去重

- memory deduplication
 - 基于写时拷贝机制
 - 在内存中扫描发现具有相同内容的物理页面
 - 执行去重
 - 操作系统发起，对用户态透明
- 典型案例：Linux KSM
 - kernel same-page merging

还有就是内存的去重，有时候不同的应用我们可能打开几次。我们在内存中扫描页，如果有相同的物理页，我们就做去重。

比如在云里面，很多虚拟机的内容都是重复的，我们扫描一下，如果是相同的页，我们就去重变成 copy-on-write 机制。KSM 会有副作用，可能会引起侧信道攻击。

内存压缩

内存压缩

- 基本思想
 - 当内存资源不充足的时候，选择将一些“最近不太会使用”的内存页进行数据压缩，从而释放出空闲内存

内存压缩案例

- Windows 10

- 压缩后的数据仍然存放在内存中
- 当访问被压缩的数据时，操作系统将其解压即可
- 思考：对比交换内存页到磁盘，压缩的优点和缺点有哪些？

- Linux

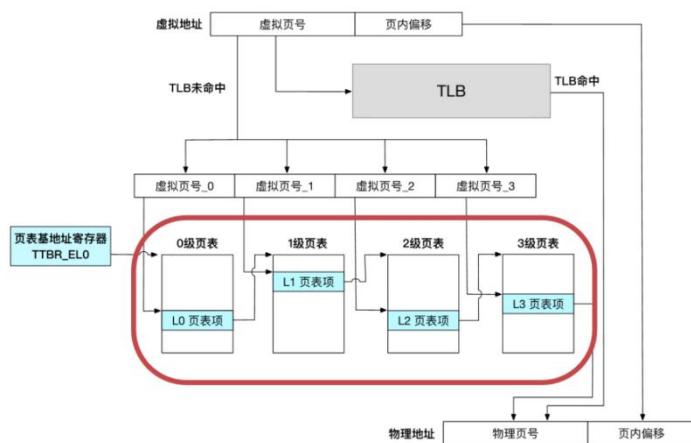
- zswap：换页过程中磁盘的缓存
- 将准备换出的数据压缩并先写入 zswap 区域（内存）
- 好处：减少甚至避免磁盘I/O；增加设备寿命

Q: 压缩的优点和缺点有什么呢？

A: 缺点就是占用 CPU，使用的时候要解压；优点就是把 I/O 给省掉了。如果 swap 压缩的内存到硬盘上的话，I/O 的总量也变少了。

大页（Huge Page）

大页：再次回顾4级页表



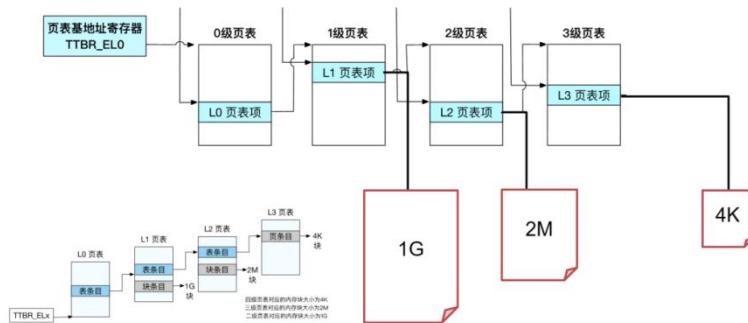
接下来我们再来看一下大页。我们先回顾一下 4 级页表。如果我们采用 4K 页，这样 4 级页表还会耗费很多 page。很多服务器几 T 的内存，page table 的耗费也是很大的，另外物理页是不连续的。所以就有了大页的机制。

大页

- 在4级页表中，某些页表项只保留两级或三级页表
- L2页表项的第1位
 - 标识着该页表项中存储的物理地址（页号）是指向 L3 页表页（该位是 1）还是指向一个 **2M 的物理页**（该位是 0）
- L1页表项的第1位
 - 类似地，可以指向一个 **1G 的物理页**

这样的话在 4 级页表里，有一些页表项只保留 2 级或 3 级页表，这样我们就可以直接使用大页。在 ARM64 中第一位如果是 0，那就是大页的标志。

大页



Q: 为什么只有 L1 和 L2 呢？L0 行不行？

A: 也是可以的。

大页的利弊

- **好处**
 - 减少TLB缓存项的使用，提高 TLB 命中率
 - 减少页表的级数，提升遍历页表的效率
- **案例**
 - 提供API允许应用程序进行显示的大页分配
 - 透明大页 (Transparent Huge Pages) 机制
- **弊端**
 - 未使用整个大页而造成物理内存资源浪费
 - 增加管理内存的复杂度

让应用可以进行显式地大页分配，OS 还会提供透明大页 (THP) 机制，实际上并没有真

正地用起来，因为会增加管理内存的复杂性。我们刚才讲的大页支持多种最小页面的大小。

AARCH64支持多种最小页面大小

- **x86_64: 4K**
- **AARCH64**
 - TCR_EL1可以配置3种：4K、16K、64K
 - 4K + 大页：2M/1G
 - 16K + 大页：32M (**思考为什么是32M?**)
 - 只有L2页表项支持大页
 - 64K + 大页：512M
 - 只有L2页表项支持大页 (ARMv8.2之前)

Q: 大家可以算一算为什么 L2 是支持大页的？

现在大量的 OS 用的还是 4K 的页。大家可以想一下用 16K 页的好处，内存分配比较快，整个内存比较容易连续。但是一个页 16K 没有用满就会造成浪费。

思考

- **什么情况适合使用大页？**
- **在物理内存足够大的今天，虚拟内存是否还有存在的必要？**
 - 如果不使用虚拟内存抽象，恢复到只用物理内存寻址，会带来哪些改变？哪些场景适合？
- **如果不依靠 MMU，是否有可以替换虚拟内存的方法？**
 - 基于高级语言实现多个同一个地址空间内运行实例的隔离
 - 基于编译器插桩实现多个运行实例的隔离
 - 更多可参考 Software Fault Isolation

Q1: 什么情况下适合使用大页？

A1: 如果我们对内存的使用比较充分的情况下，且访问是连续的，就可以使用大页。

Q2: 虚拟内存是否还需要存在呢？恢复到只用物理地址的寻址会有哪些问题？

A2: 刹车里的微控制器本身就是没有虚拟内存的。虚拟内存的目标是提供抽象来解耦，使得我们写的程序可以跨很多平台来执行。所以目前来看虚拟内存主流的机器上还是有必要的。

现在还有一些研究在研究页表是否能被替换掉。对于一些大数据来说，干脆还是使用 segment 的机制。这都是目前的一些探索的方向。

之前微软用语言来隔离，来实现在同一个地址空间内运行实例。这些都是可以供大家来做深入的思考。

虚拟内存机制的优势

- **高效使用物理内存**
 - 使用 DRAM 作为虚拟地址空间的缓存
- **简化内存管理**
 - 每个进程看到的是统一的线性地址空间
- **更强的隔离与更细的权限控制**
 - 一个进程不能访问属于其他进程的内存
 - 用户程序不能够访问特权更高的内核信息
 - 不同内存页的读、写、执行权限可以不同

企业里一个很重要的问题就是一个大的系统出了问题以后定位是谁的错误。虚拟内存就通过不同进程的虚拟空间来提供了定位和隔离的功能。

物理内存管理

我们虚拟内存介绍到这里，我们继续介绍物理内存的管理。在讲物理内存管理的时候，要继续回顾一下 OS 启动的时候做了哪些事情。在切换到虚拟内存的时候，我们做了一个低地址映射。OS 实际上管理的是虚拟地址，所以我们在内核里分配的一个指针是指向虚拟地址的，只是我们可以通过一一映射解算出物理地址。

如果我们出现了一个缺页异常，我们要分配一个页填进去。实际上首先分配的是一个虚拟地址，然后转换成物理地址，再填到页表里面。大家一定要记住，页表里面放的是物理地址。

OS什么时候需要考虑物理内存？

- 引入虚拟内存后，物理内存主要在以下四个场景出现：
 1. 用户态应用程序触发on-demand paging时
 - 此时内核需要分配物理内存页，映射到对应的虚拟页
 2. 内核自己申请内存并使用时
 - 用于内核自身的数据结构，通常通过kmalloc()完成
 3. 内核申请用于设备的DMA缓存时
 - DMA缓存通常需要连续的物理页
 4. 发生换页（swapping）时
 - 通过磁盘来扩展物理内存的容量

物理地址会在以下四个场景出现。

1. on-demand paging，此时映射的是物理地址。
2. 内核自己申请内存的时候，通常使用 kmalloc 来完成分配。
3. 内核申请设备的 DMA 访存的时候。
4. 发生换页的时候。

有一个 scatter-gather list 的机制，就是把数据分散开来，要用的时候再汇聚起来。但是我们 DMA 操作的时候还是希望连续的物理页，这样局部性较好。对于相机应用来说，连续的物理页会让它的速度非常快。超过 1~2 秒，用户会明显地觉得有点卡。还有就是我们发生换页的时候，通过磁盘扩展内存容量的时候，也需要使用到物理内存。

场景-1：应用触发on-demand paging

- **问：当应用调用malloc时，与物理内存是否有关？**
 - 应用调用malloc时，内核仅仅为其分配虚拟地址
 - 此时虚拟地址对应的VMA为valid，但对应页表的valid bit为0
 - 当第一次访问新分配的虚拟地址时，CPU会触发page fault
- **操作系统需要做（即page-fault handler）：**
 - 找到一块空闲的物理内存页 ← 物理内存管理（页粒度）
 - 修改页表，将该物理页映射到触发page-fault的虚地址所在虚拟页
 - 回到应用，重复执行触发page-fault的那行代码

此时虚拟地址的 VMA 的值为 valid，对应页表的 valid bit 为 0，也就是还没有页表。第一次访问新分配的虚拟地址的时候，就会触发 page fault，找到一块空闲的物理页，映射到虚拟页。

场景-2：内核自身用到的数据结构

- **内核运行过程中也需要用到动态内存**
 - 例如：挂载文件系统后需要大量dentry数据结构
 - 动态性：用时分配，用完释放，类似用户态的malloc
 - 申请到的内存，不一定是页粒度，可能更细（特点-1）
- **问：为什么不使用与malloc类似的机制？**
 - 即先分配虚拟地址，然后通过on-demand paging分配物理页
 - 对内核来说，开销太大：这是内核中发生的page fault！
 - 若要不触发page fault，则必须已经映射完成（特点-2）

它加载到内存的时候，可能是一个 dentry，大概也就是几十到几百个 byte，这样是用不到 4K 的。

Q：为什么我们这时候不使用 malloc 的机制，先分配虚拟内存呢？

A：可以这么做，但是 OS 就会非常复杂。并且因为 on-demand paging 机制，性能也会变差。

场景-3：设备需要分配DMA内存

- **DMA：设备绕过CPU直接访存**
 - 由于绕过CPU的MMU，因此直接访问物理地址
 - 通常需要大段连续的物理内存
- **操作系统必须有能力分配连续的物理页**
 - 需要用一种高效的方式来组织和管理物理页

因为绕过了 CPU 和 MMU，所以需要直接访问物理地址。不连续其实也可以，但是性能会受影响。

场景-4：换入换出 (swapping)

- **换出操作：物理内存不够时**
 - OS选择不常用的物理内存（不同的选择策略）
 - OS将内存中的数据写入磁盘块，并记录磁盘块与内存的关联
 - OS更新页表，将对应页表项的valid bit设置为0
- **换入操作：当换出的页被访问时，触发page fault**
 - OS判断该地址所在页被换出，找到对应的磁盘块
 - OS分配空闲的物理内存页；若没有空闲页，则再次进行换出操作
 - OS将磁盘块中的数据读入前一步找到的内存页
 - OS更新页表，将对应页表项的valid bit设置为1

这四个场景都会涉及到物理内存的分配和管理，这也是内核要做的事情。

内核物理内存管理的需求和评价指标

- **物理内存管理对操作系统的需求**
 - 管理所有空闲的物理页，有能力分配一个或多个连续物理页
 - 管理所有已分配的内存页，并能随时将其回收，变成空闲页
- **物理内存管理的评价指标**
 - 资源利用率：尽可能避免碎片（外部碎片 & 内部碎片）
 - 分配速度：即内存分配与释放操作的时延
 - 分配公平性：避免某些应用独占大量内存

比如拍照的延迟给人的感觉好不好，内存分配的效率也很关键。

直接映射机制

这里我们就可以看到一些机制。比如直接映射机制，在启动的时候，OS 把物理地址和

虚拟地址一一映射。

- OS一次性将所有物理内存映射到一段虚拟地址空间

- 任一物理地址和虚拟地址仅相差一个偏移量
 - OS可迅速根据物理地址或虚拟地址互相计算
- 看上去类似最传统的段机制，实际通过配置页表来实现

- 直接映射机制的特点

- 内核使用直接映射的虚拟地址，不会触发page fault
- 更方便找到连续的物理页：只要寻找连续的虚拟页

这样 OS 就可以快速地通过加减法进行计算物理地址和虚拟地址。我们只需要找到连续的虚拟页，就可以找到连续的物理页。

但是我们要注意几点：

关于直接映射机制，需要注意的几点

- 对内核来说：

- 已映射的地址不一定正在使用（需要通过kmalloc才能用）
- 正在使用的地址通常已映射（例外：vmalloc）

- 对应用来说：

- 正在使用的地址不一定已映射（on-demand paging）
- 已映射的地址一定正在使用（否则不会被映射）

- 同一个物理地址可以有多个虚拟地址

- 如应用任一已映射的虚拟地址，在内核的直接映射下也有一个虚拟地址

kmalloc 对应于 kfree，可以分配连续的物理内存；

vmalloc 对应于 vfree，分配连续的虚拟内存，但是物理上不一定连续。

vmalloc 分配内存的时候逻辑地址是连续的，但物理地址一般是不连续的，适用于那种一下需要分配大量内存的情况，如 insert 模块的时候。这种分配方式性能不如 kmalloc。

kmalloc 分配内存是基于 slab，因此 slab 的一些特性包括着色，对齐等都具备，性能较好。
物理地址和逻辑地址都是连续的

[1]<https://developer.aliyun.com/article/296975>

所以在内核中也有可能发生 page fault，发生缺页的内核态地址只可能位于 vmalloc 区。

<http://blog.chinaunix.net/uid-29813277-id-4425532.html>

Q：那么 Vmalloc 区为什么会产生 page fault 呢？内核使用 Vmalloc 分配内存时，不是已经分配了相应的物理内存，并创建了相应的页表项进行了映射吗？

A：的确，在使用 Vmalloc 分配物理内存时，确实进行了映射，创建了页表项。但是，其修改的仅为内核的主内核页表，并没有更新相关进程的页表。在 Vmalloc 分配内存后，第一次访问相关的线性地址时，由于相关进程的页表中并没有 Vmalloc 分配内存相应的页表项，所

以会触发 page fault，而这个 page fault 的处理过程其实就是：从内核主页表中同步相应的页表项到进程的页表中，完成这次同步后，进程中相关的页表项建立，后续再次访问相关线性地址时，就不会再产生缺页异常了。

伙伴系统：页粒度内存管理

大家都写过 malloc lab，已经比较熟悉了。这是一个固定大小的 Buddy System。

空闲物理内存管理的简单方法：bitmap

- 最简单直接的方法：

- 操作系统通过一个bitmap记录物理页是否空闲
- 分配时，通过bitmap查找空闲物理页，并在bitmap中标记非空闲
- 回收时，在bitmap中，把对应的物理页标记成空闲

- 该方法有什么问题？

如果我们的 OS 非常理想，仅仅以一个页为粒度分配内存是没有问题的，但是一些视频等需要大块内存的硬件设备。就会有问题了。可能会导致外部碎片的问题。

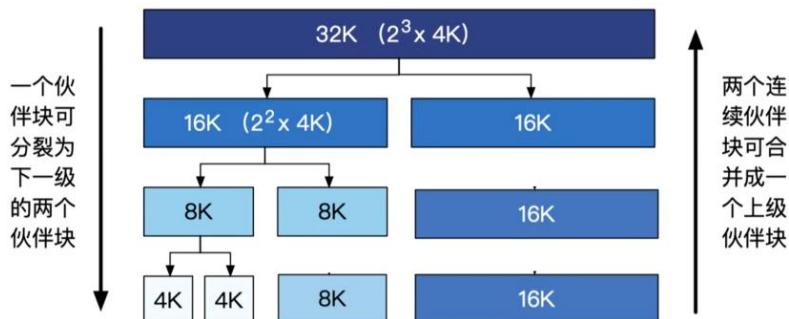
简单管理方法导致外部碎片问题



从上往下是时序的一个序列，在 Time4 的时候有 2 个页是空闲的，但是不能分配新的 2 个页了。

伙伴系统 (buddy system)

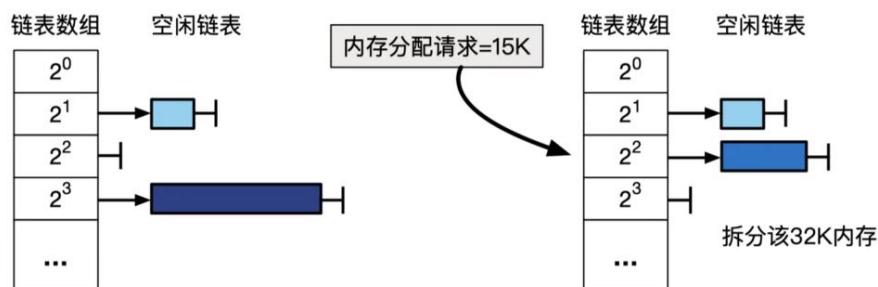
- 伙伴系统：分裂与合并



对内核来说，buddy system 的分配非常高效。

伙伴系统例子：分配15K内存

当一个请求需要分配 m 个物理页时，伙伴系统将寻找一个大小合适的块，该块包含 2^n 个物理页，且满足 $2^{n-1} < m \leq 2^n$

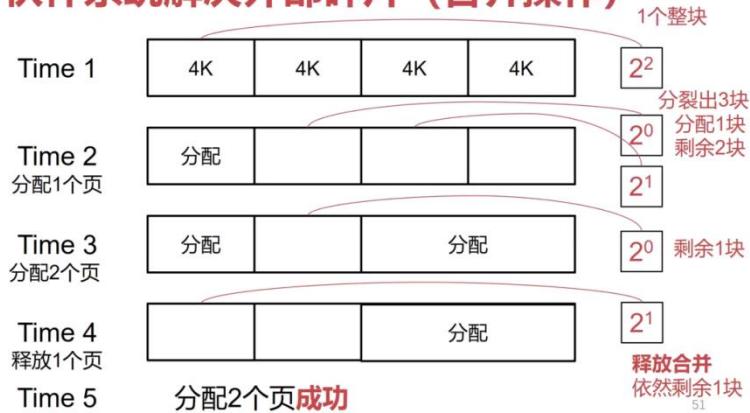


伙伴系统的巧妙之处

- 高效地找到伙伴块
 - 互为伙伴的两个块的物理地址仅有**一位**不同
 - 而且块的**大小决定**是哪一位
- 例如：
 - 块A (0-8K) 和块B (8-16K) 互为伙伴块
 - 块A和B的物理地址分别是 0x0 和 0x2000
 - 仅有第**13**位不同，块大小是8K (2^{13})

伙伴系统通过互为伙伴的块的不同的位，我们就可以知道伙伴块的大小。

伙伴系统解决外部碎片（合并操作）



伙伴系统：以页为粒度的物理内存管理

- **基于直接映射机制的物理内存管理**
 - 一旦分配，则可以直接使用，而不会触发page fault
 - 若虚拟地址连续，则物理地址也连续
- **优点**
 - 外部碎片程度降低（但并没有完全解决）
 - 释放物理页时，可快速查找同层的伙伴页，使合并机制的性能提高
- **缺点**
 - 一次内存页释放有可能导致大量合并操作，导致高时延
 - 内部碎片依然存在：如请求9KB，分配16KB（4个页）

我们可以通过直接映射的方式管理物理内存。虚拟地址连续的话，物理地址也一定是连续的。

SLAB/SLUB/SLOB：细粒度内存管理

如何实现内核的malloc (kmalloc) ?

- **简单方法：直接使用伙伴系统**
 - 每次分配，将大小对齐到页粒度（如4K）
- **优点：**
 - 分配得到的地址已映射物理内存，不会触发page fault
- **缺点：**
 - 大量数据结构都在几十Byte粒度，导致内存利用率极低
- **需在内存页的基础上，再加一层更细粒度的内存分配系统**

简单方法就可以使用伙伴系统，但是缺点就是内存利用率非常低。我们需要在以页为粒度的基础上再加一层细粒度的内存分配系统。

SLAB：建立在伙伴系统之上的分配器

- **目标：快速分配小内存对象**
- **SLAB分配器家族：SLAB、SLUB、SLOB**
 - 上世纪 90 年代，Jeff Bonwick 在 Solaris 2.4 中首创 SLAB
 - 2007 年左右，Christoph Lameter 在 Linux 中提出 SLUB
 - Linux-2.6.23 之后 SLUB 成为默认分配器
 - 发展过程中，提出针对内存稀缺场景的 SLOB
- **后续以主流的 SLUB 为例讲解**

SLUB分配器的思路

- **观察**
 - 操作系统频繁分配的对象大小相对比较固定
- **基本思想**
 - 从伙伴系统获得大块内存（名为 slab）
 - 对每份大块内存进一步细分成固定大小的小块内存进行管理
 - 块的大小通常是 2^n 个字节（一般来说， $3 \leq n < 12$ ）
 - 也可为特定数据结构增加特殊大小的块，从而减小内部碎片

2022/3/8

上节课我们介绍了 Buddy System 和 SLAB 等机制。用户态内存分配千奇百怪，可能分配各种各样的数据。但是对内核而言，它是一个固定的程序，所以它分配的对象是有限的，每个对象的大小是已知的。

为什么要讲内存分配，不一定是写内核，比如写游戏引擎或者写服务器，也可以使用类似的机制来高效地做 malloc。

SLUB分配器的思路

- 观察

- 操作系统频繁分配的对象大小相对比较固定

- 基本思想

- 从伙伴系统获得大块内存（名为slab）
- 对每份大块内存进一步细分成固定大小的小块内存进行管理
- 块的大小通常是 2^n 个字节（一般来说， $3 \leq n < 12$ ）
- 也可为特定数据结构增加特殊大小的块，从而减小内部碎片

这个观察会用到我们很多的设计里面。我们对大块内存继续分成固定大小的小块内存。

SLUB设计

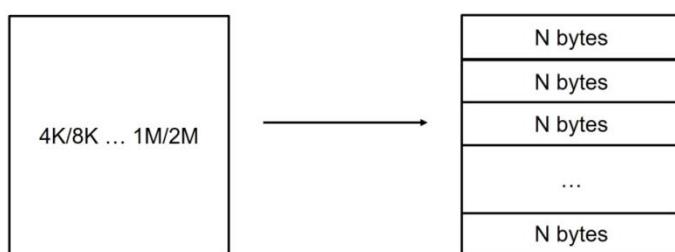
- 只分配固定大小块

- 对于每个固定块大小，SLUB分配器都会使用独立的内存资源池进行分配
- 采用**best fit**定位资源池



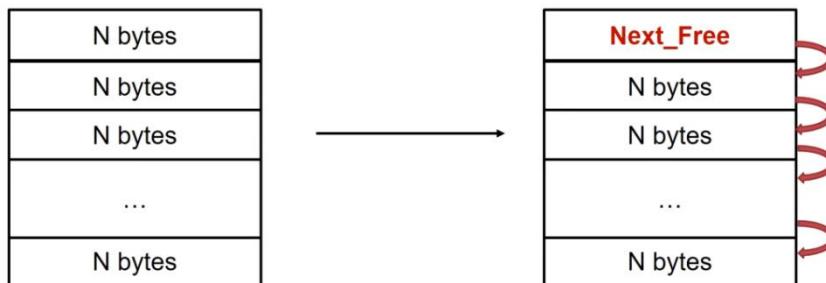
具体示例

把从伙伴系统得到的连续物理页划分成若干等份 (slot)



具体示例

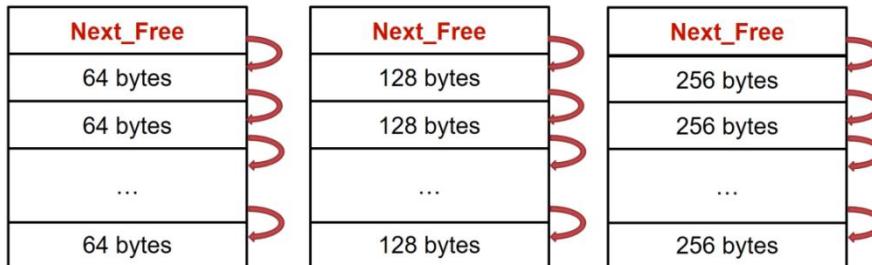
当分配时直接分配一个空闲slot:
如何区分是否空闲? 采用空闲链表



具体的方式我们采用的是空闲链表。

具体示例

分配N字节时，首先找到大小最合适的SLAB，
取走Next_Free指向的第一个；释放时直接放回Next_Free后



具体示例

释放时如何找到Next_Free?

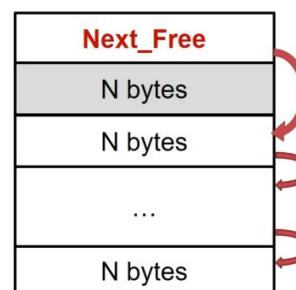
kfree (object)

提示：SLAB的大小是固定的

思路：根据object地址找到当前SLAB
的起始地址即可 (Next_Free)

ADDR & ~(SLAB_SIZE-1)

SLAB



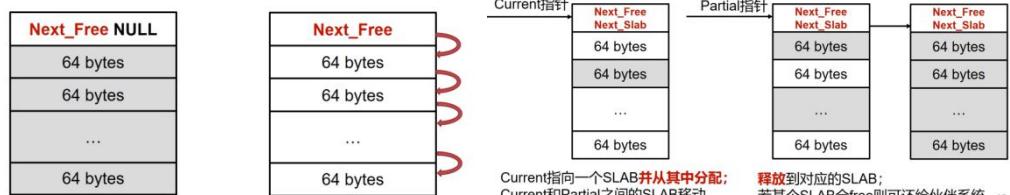
首先我们要定位到 Next_Free 指针在哪。因为 SLAB 的大小是固定的，所以我们可以通
过 Object 的地址来算出 SLAB。

如果 64 个字节的 SLAB 已经分配完了怎么办呢？只需要 Next_Free 置为空，且重新分配

一个 SLAB 就行了。

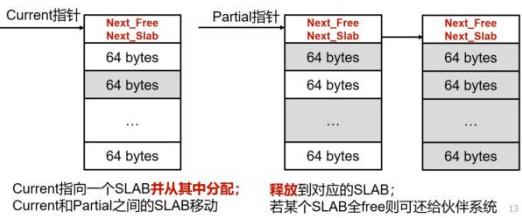
具体示例

当64字节slot的SLAB已经分配完怎么办?
再从伙伴系统分配一个SLAB



具体示例

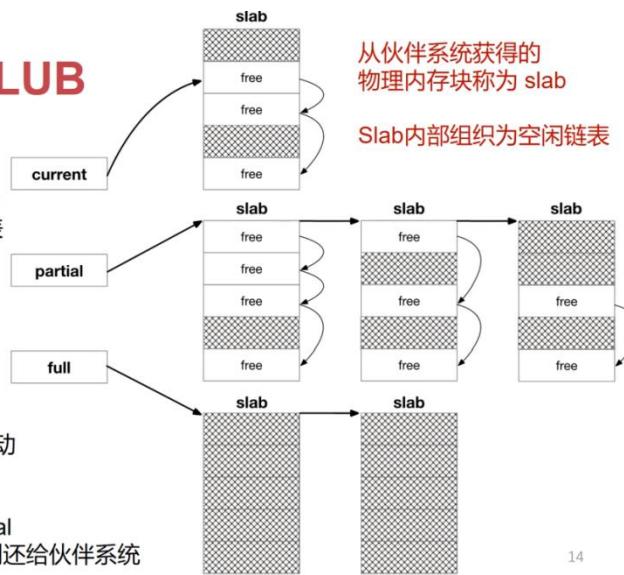
如何组织多个64字节slot的SLAB? 引入两个指针



总体来看SLUB

三个指针

- current仅指向一个 slab
- partial指向未满slab链表
- full指向全满slab链表



总的来说，我们可以有 3 个指针，网格表示已分配。Full 指针是可选的。

SLUB内存分配器小结

• 伙伴系统

- 分配单个或连续的物理页
- 有效解决外部碎片问题
- 思考：分配效率？

• SLUB

- 分配比页小的内存请求
- 有效解决内部碎片问题
- 思考：分配效率？

换页策略 (Page Swap)

用着用着，OS 中的页就不够了。如果我们物理内存用满了，就需要一个策略换进换出。我们也不能直接把内存擦除，因为我们的内存中可能还有重要信息没有保存。如果是文件页，就 swap 到对应的磁盘空间中。我们应该换出哪些，就是换页的策略问题。

理想的换页策略 (OPT策略)

假设物理内存中可以存放三个物理页，初始为空，
某应用程序一共需要访问物理页面 1~5

物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
物理内存中 存放的物理页面	3 2 2 2 2 1 4 4 4 4 4 2 2 2 2 2	3 2 2 2 2 3 4 4 4 4 4 3 3 3 3 3	3 3 3 3 3 3 5 5 5 5 3 3 3 3 3									
缺页异常 (共 6 次)	是 是 否 是 是 否 是 否 否 是 否 否 否 否											

在选择被换出的页面时，优先选择未来不会再访问的页面，
或者在最长时间内 不会再访问的页面。

FIFO策略

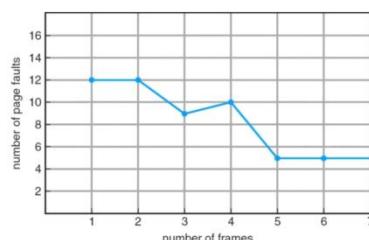
操作系统维护一个队列用于记录换入内存的物理页号，
每换入一个物理页就把其页号加到队尾，
因此最先换进的物理页号总是处于队头位置

物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
(该行为队列头)	3 2 2 2 1 4 4 4 3 3 3 5 5 2	3 2 2 2 1 4 3 3 3 5 5 2	3 2 2 2 1 4 3 3 3 5 5 2									
存储物理页号 的 FIFO 队列												
缺页异常 (共 9 次)	是 是 否 是 是 是 是 否 是 否 是 是											

Belady's Anomaly

- 访问顺序: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- 3个物理页: 9次page faults
- 4个物理页: 10次page faults



FIFO 策略，加了物理页之后，page fault 的次数反而增加了。

Second Chance策略

FIFO 策略的一种改进版本：为每一个物理页号维护一个访问标志位。

如果访问的页面号已经处在队列中，则置上其访问标志位。

换页时查看队头：1) 无标志则换出；

2) 有标志则去除标志放入队尾，继续寻找

物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
该行是队列头部	3	3	3*	3*	1	1	3*	3*	3	3*	3*	3*
FIFO 队列		2	2	2	3	3*	4	4*	4	4	4*	4*
存储物理页号			1	4	4	5	5	2	2	2	2	2
缺页异常 (共 6 次)	是	是	否	是	是	否	是	否	是	否	否	否

体现了局部性原理。它本质上还是 FIFO 策略。

LRU策略

OS维护一个链表，在每次内存访问后，OS把刚刚访问的内存页调整到链表尾端；每次都选择换出位于链表头部的页面

缺点-1：对于特定的序列，效果可能非常差，如循环访问内存

缺点-2：需要排序的内存页可能非常多，导致很高的额外负载

物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
该行为链表头部	3	3	2	2	3	1	4	3	5	4	2	2
越不常访问的页号		2	3	3	1	4	3	5	4	2	3	4
离头部更近			1	4	3	5	4	2	3	4	3	
缺页异常 (共 7 次)	是	是	否	是	是	否	是	否	是	是	否	否

这个也是利用了局部性原理，认为最近访问过的内存页更容易被访问到。

LRU如何实现呢？

• 精确排序

- CPU访问某个页时，在页表项上打时间戳（cycle数）
- OS遍历所有内存页的页表项，根据时间戳排序

• 缺点：

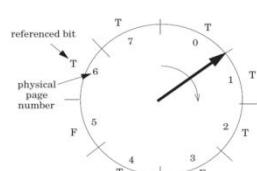
- 遍历所有页表项太费时
- 不同CPU的时间戳不一定全局排序，无法保证完全精确

• 思路：是否可用近似的方式？

时钟算法策略

• 物理页环形排列（类似时钟）

- 当物理页被访问时，CPU把对应页表项的“访问位”设成T
- OS依次（如顺时针）查看每个页的“访问位”
 - 如果是T，则置成F
 - 如果是F，则驱逐该页



思考具体实现案例：时钟算法

• 每个物理页有一个“访问位”

- 实际上，硬件是在页表项里面为虚拟页打上“访问位”
- Recap: Access Flag

• 如何实现

- OS为每个物理页维护元数据
 - 当物理页被填写到某张页表中时，把页表项的位置记录在元数据中（在Linux中称为“反向映射”：reverse mapping）
 - 根据物理页对应的页表项中的“访问位”判断是否驱逐
 - 驱逐某页时应该清空所有对应的页表项（例如共享内存）

所以我们使用 LRU 来近似时钟算法。

页替换策略小结

- **常见的替换策略**
 - FIFO、LRU/MRU、时钟算法、随机替换 ...
- **替换策略评价标准**
 - 缺页发生的概率 (参照理想但不能实现的OPT策略)
 - 策略本身的性能开销
 - 如何高效地记录物理页的使用情况?
 - 页表项中Access/Dirty Bits
- **Thrashing Problem**

换出的时候如果已知没有 Dirty 的时候，就不需要写回，降低了换出时候的代价。

Thrashing Problem

我们再介绍一个 thrashing problem，这是一个经典的问题。

Thrashing Problem

- **直接原因**
 - 过于频繁的缺页异常 (物理内存总需求过大)
- **大部分 CPU 时间都被用来处理缺页异常**
 - 等待缓慢的磁盘 I/O 操作
 - 仅剩小部分的时间用于执行真正有意义的工作
- **调度器造成问题加剧**
 - 等待磁盘 I/O 导致 CPU 利用率下降
 - 调度器载入更多的进程以期提高 CPU 利用率
 - 触发更多的缺页异常、进一步降低 CPU 利用率、导致连锁反应

颠簸的直接原因就是过于频繁的缺页异常，这样大部分 CPU 时间都来处理缺页异常了。假设大家妈妈爸爸爷爷奶奶都叫我们去做事，每件事情还没做又把你喊到其他地方去了。在这个过程中其实我们做事的时候很少，还没开始干活又把 CPU 放出去了。所以 CPU 大部分时间是在把数据从磁盘搬到内存里面。调度器会造成这个问题加剧。

工作集模型

工作集模型 (有效避免Thrashing)

- 一个进程在时间 t 的工作集 $W(t, x)$:
 - 其在时间段 $(t - x, t)$ 内使用的内存页集合
 - 也被视为其在未来(下一个 x 时间内)会访问的页集合
 - 如果希望进程能够顺利进展, 则需要将该集合保持在内存中
- 工作集模型: All-or-nothing
 - 进程工作集要么都在内存中, 要么全都换出
 - 避免thrashing, 提高系统整体性能表现

在这个过程中, 我们发现 Page Fault 主要是发生在工作集切换的时候, 在工作集中运行的时候不会 Page Fault。

跟踪工作集 $w(t, x)$

- 工作集时钟中断固定间隔发生, 处理函数扫描内存页
 - 若访问位为1, 则表示此次tick中被访问
 - 记录上次使用时间为当前时间
 - 若访问位为0, 则表示此次tick中未访问
 - Age = 当前时间 - 上次使用时间
 - 若Age大于设置的 x , 则不在工作集
 - 最后, OS将所有访问位清0
 - 注意: 访问位由CPU在访问时设为1

当前时间: 2020	
2010	1
2000	1
1970	0
1990	0

上次使用时间 访问位

我们在工作集的时间中断中收集一下, 扫描一下内存页, 并且设置 access bit。也就是内存页会有一个年龄, 如果年龄大于某个阈值, 我们就认为它不在工作集里。

工作集模型假设

假设认为, 一个应用程序在一段时间 x 内使用的内存页集合, 也会在下一段时间 x 内使用。因此, 在每个时间段 x 内, 工作集应当保存在物理内存中。

早期实现

早期工作集的实现是 all-or-nothing, 即要么将工作集都加载入物理内存, 要么全部换出。现在很少用这种原则, 但工作集的概念依然指导着操作系统的换页策略, 即优先将非工作集中的页换出。

如何追踪工作集

常见做法是“工作集时钟算法”。操作系统利用定时器周期运行一个时间追踪函数，假设时间间隔为 T 。该函数为每个内存页维护两个状态上次使用时间和访问位，均初始化为 0。

本文选择与原书《现代操作系统 原理与实现》不同的叙事顺序，根据功能进行叙述。

关于访问位的行为

每次访问某个页时，对应访问位会置为 1，每当函数检查完一个页状态后，将访问位置为 0（即每隔一个周期，访问位置为 0）。这样做的作用是，每当周期函数执行时，如果检查到某个页的访问位为 1，说明这个页在时间 T 内被访问过（然后将该位还原为 0）。

关于上次使用时间的行为

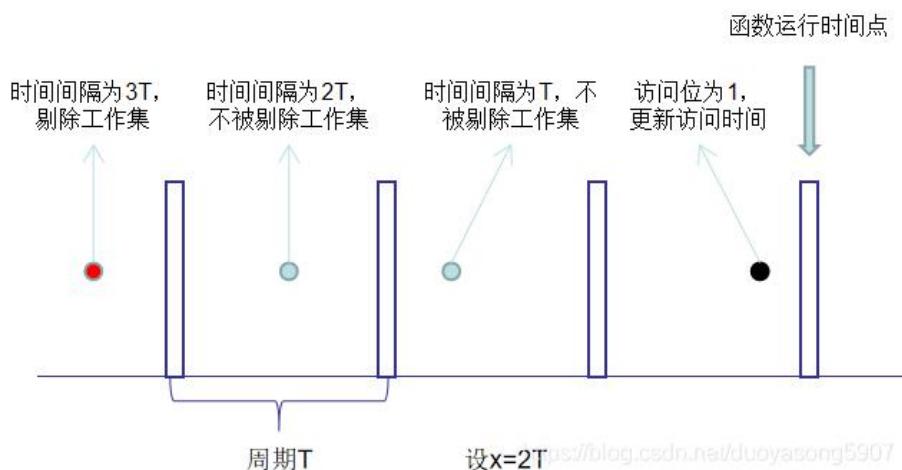
每当周期函数执行时，如果检查到某个页的访问位为 1，把当前系统时间赋值给该页的上次使用时间。检查发现访问位是 0 呢那就计算该页的年龄（当前系统时间-该页的上次使用时间），如果超过了预设的时间间隔 x ，则该页不在属于工作集。

效果解释

为什么要用周期为 T 运行的函数，并检查未访问页的年龄呢？这有什么用意？

我们考虑理想状况下，该函数是随时运行的，而不是每隔时间 T 运行的。也就是说，这个函数会时刻检查每个页距离上次访问的时间是否大于 x ，只要大于，就将其剔除工作集。这样做的话，该函数确实能时刻保持时间 x 内被访问过的页面集合。但问题是，函数是不能每个瞬间都在执行的，它只能以某个周期 T 运行。

所以，当退化为周期运行的函数时，行为就会调整，把一段时间内访问过的页，其访问时间都归约在 T 的检查点上。比如下图所示，每个小圆圈是那个页被访问的时刻。只有红圈的上次使用时间距离函数检查点为 $3T$ ，大于 $x=2T$ 。因此只有红圈代表的页面被剔除工作集。



个人认为，在实践中 x 都较大于 T ，使得周期运行的函数能模拟“时刻在检查”的效果。

总结

周期运行的函数实现了类似这样的效果：检查函数每隔时间 T 的周期执行一次，如果有页面上次访问距今超过 x ，则剔除工作集。所有 T 之内不同时刻访问的页面，其最近访问时间都归约到 T 的检查点上。

能够追踪工作集后，当发生页面置换时，可以优先从非工作集的页面置换。

版权声明：本文为 CSDN 博主「Melody2050」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/duoyasong5907/article/details/119717433>

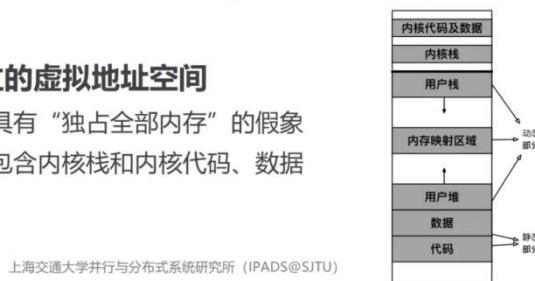
进程

进程：运行中的程序

- **进程是计算机程序运行时的抽象**
 - 静态部分：程序运行需要的代码和数据
 - 动态部分：程序运行期间的**状态**（程序计数器、堆、栈……）

- **进程具有独立的虚拟地址空间**

- 每个进程都具有“独占全部内存”的假象
- 内核中同样包含内核栈和内核代码、数据



进程有独立的虚拟地址空间，同样内核包括内核栈和内核代码、数据。

进程控制块 (PCB)

如何表示进程：进程控制块 (PCB)

- **每个进程都对应一个元数据，称为“进程控制块” PCB**
 - 进程控制块存储在内核态（**为什么？**）

- **想一想：进程控制块里至少应该保存哪些信息？**

- 独立的虚拟地址空间
- 独立的执行上下文

```
1 // ChCore 中的 PCB——process
2 struct process {
3     // 上下文
4     struct process_ctx *process_ctx;
5     // 虚拟地址空间
6     struct vmspace *vmspace;
7 };
```

进程控制块对应的就是进程对应的元数据，它是存储在内核态的。因为这是 OS 内核用来管理进程的，不能被用户态代码修改。包含虚拟地址空间和上下文。

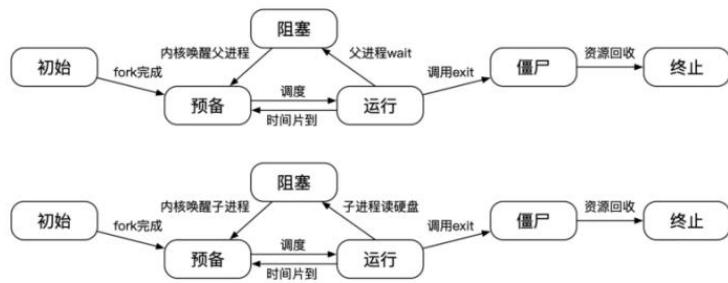
简化模型：单一线程的进程

- 假设每个进程都只有一个线程
 - 历史上确实如此！（如早期的UNIX操作系统）
- 好处：便于理解操作系统中的各种概念
 - 多线程使操作系统的管理更加复杂
 - 在单一线程的进程中：线程管理 / 调度 \approx 进程管理 / 调度
 - 线程的内容将在后面介绍

早期每个进程只有一个线程。

进程管理：即管理进程的生命周期

- 进程自创建到终止可经历多个过程
 - 称为进程状态
- 不同的系统调用和事件会影响进程的状态



进程管理有它的生命周期，包含进程从创建到终止最终的状态。执行过程中有各种各样的设备事件来影响进程的状态。

进程创建：fork()

- 语义：为调用进程创建一个一模一样的新进程
 - 调用进程为父进程，新进程为子进程
 - 接口简单，无需任何参数
- fork后的两个进程均为独立进程
 - 拥有不同的进程id
 - 可以并行执行，互不干扰（除非使用特定的接口）
 - 父进程和子进程会共享部分数据结构（内存、文件等）

fork 的话大家已经很熟悉了。

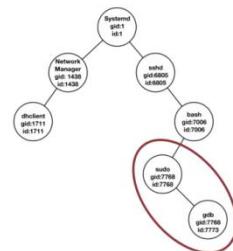
进程树与进程组

- **fork为进程之间建立了父进程和子进程的关系**

- 进程之间建立了树型结构
 - Linux可使用ps命令查看

- **多个进程可以属于同一个进程组**

- 子进程默认与父进程属于同一个进程组
 - 可以向同一进程组中的所有进程发送信号
 - 主要用于shell程序中



Linux 的 0 号进程是 `idle` 进程，`init` 进程的 PID 是 1。

进程的执行：exec

- **为进程指定可执行文件和参数**



- **在fork之后调用**

- exec在载入可执行文件后会重置地址空间

一般是 `fork + exec` 构成了创建新进程加载新文件的过程。

写时拷贝 (Copy-On-Write)

- **早期的fork实现：将父进程直接拷贝一份**

- 性能差：时间随占用内存增加而增加
 - 无用功：fork之后如果调用exec，拷贝的内存就作废了

- **基本思路：只拷贝内存映射，不拷贝实际内存**

- 性能较好：一条映射至少对应一个4K的页面
 - 调用exec的情况下，减少了无用的拷贝

现在有一些 OS 进程不只是共享页，还可以共享一部分的页表。

fork的优缺点分析

- **fork的优点**

- 接口非常简洁
- 将进程“创建”和“执行”(exec)解耦，提高了灵活度
- 刻画了进程之间的内在关系(进程树、进程组)

- **fork的缺点**

- 完全拷贝过于粗暴(不如clone)
- 性能差、可扩展性差(不如vfork和spawn)
- 不可组合性(例如：fork() + pthread())

但 fork 的缺点就在于完全 copy 过于粗暴、性能差、可扩展性差，也不能组合。fork() + pthread() 和 pthread() 创建函数再 fork 一下里面有几个线程？

fork的替代接口

- **vfork：类似于fork，但让父子进程共享同一地址空间**

- 优点：连映射都不需要拷贝，性能更好
- 缺点：
 - 只能用在“fork + exec”的场景中
 - 共享地址空间存在安全问题

- **轶事：vfork的提出最初就是为了解决fork的性能问题**

- 但写时拷贝拯救了fork

Since this function is hard to use correctly from application software, it is recommended to use posix_spawn(3) or fork(2) instead.

fork的替代接口

- **posix_spawn：相当于fork + exec**

- 优点：可扩展性、性能较好
- 缺点：不如fork灵活

- **clone：fork的“进阶版”，可以选择性地不拷贝内存**

- 优点：高度可控，可依照需求调整
- 缺点：接口比fork复杂，选择性拷贝容易出错

这是用的特别高效的一个方法。我们可以使用 posix_spawn 替代 fork。如果我们要 fork 一百次，用 posix_spawn 的效率可以高一个数量级。缺点就是它不如 fork 灵活。

线程

为什么需要线程呢？因为创建进程的开销很大。

为什么需要线程？

- **创建进程的开销较大**
 - 包括了数据、代码、堆、栈等
- **进程的隔离性过强**
 - 进程间交互：可以通过进程间通信（IPC），但开销较大
- **进程内部无法支持并行**

线程：更加轻量级的运行时抽象

- **线程只包含运行时的状态**
 - 静态部分由**进程**提供
 - 包括了执行所需的**最小状态**（主要是寄存器和栈）
- **一个进程可以包含多个线程**
 - 每个线程共享同一地址空间（方便数据共享和交互）
 - 允许进程内并行

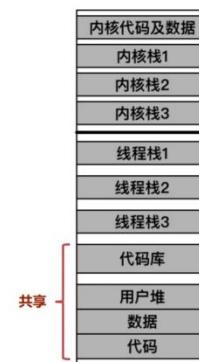
一个进程可以包含多个线程，每个线程可以共享同一个地址空间。这样就有了一个进程可以包含多个线程。

进阶模型：多线程的进程

- **一个进程可以包含多个线程**
- **一个进程的多线程可以在不同处理器上同时执行**
 - 调度的基本单元由进程变为了线程
 - 每个线程都有**状态**
 - 上下文切换的单位变为了线程

多线程进程的地址空间

- 每个线程拥有自己的栈
- 内核中也有为线程准备的内核栈
- 其它区域共享
 - 数据、代码、堆.....

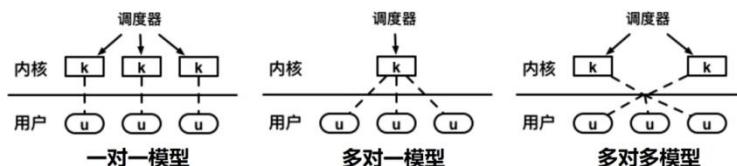


用户态线程与内核态线程

- 根据线程是否受内核管理，可以将线程分为两类
 - 内核态线程：内核可见，受内核管理
 - 用户态线程：内核不可见，不受内核直接管理
- 内核态线程
 - 由内核创建，线程相关信息存放在内核中
- 用户态线程（纤程）
 - 在应用态创建，线程相关信息主要存放在应用数据中

线程模型

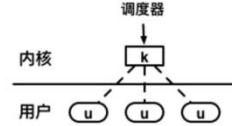
- 线程模型表示了用户态线程与内核态线程之间的联系
 - 多对一模型：多个用户态线程对应一个内核态线程
 - 一对一模型：一个用户态线程对应一个内核态线程
 - 多对多模型：多个用户态线程对应多个内核态线程



我们看到的线程都有一个内核线程（一对一模型）。还有的就是多对多的模型，一般来说用户态线程会多一点，内核态线程多了没用。

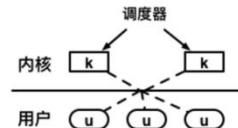
多对一模型

- 将多个用户态线程映射给单一的内核线程
 - 优点：内核管理简单
 - 缺点：可扩展性差，无法适应多核机器的发展
- 在主流操作系统中被弃用
- 用于各种用户态线程库中



多对多模型（又叫Scheduler Activation）

- N个用户态线程映射到M个内核态线程 ($N > M$)
 - 优点：解决了可扩展性问题（多对一）和线程过多问题（一对一）
 - 缺点：管理更为复杂
- Solaris在9之前使用该模型
 - 9之后改为一对一
- 在虚拟化中得到了广泛应用



N 个用户态线程映射到 M 个内核态线程，因为线程会 sleep、等待，就需要动态地改变这个映射，所以管理的复杂性比较复杂。

TCB

线程的相关数据结构：TCB

- 一对一模型的TCB可以分为两部分
- 内核态：与PCB结构类似
 - Linux中进程与线程使用的是同一种数据结构 (task_struct)
 - 上下文切换中会使用
- 应用态：可以由线程库定义
 - Linux: pthread结构体
 - Windows: TIB (Thread Information Block)
 - 可以认为是内核TCB的扩展

线程本地存储 (TLS)

- 不同线程可能会执行相同的代码
 - 线程不具有独立的地址空间，多线程共享代码段
- 问题：对于全局变量，不同线程可能需要不同的拷贝
 - 举例：用于标明系统调用错误的errno
- 解决方案：线程本地存储 (Thread Local Storage)

每个线程都需要调用系统调用，因此它们希望errno是每个线程独有的，这样它们可以在访问同名变量的情况下获得不一样的结果

线程本地存储 (TLS)

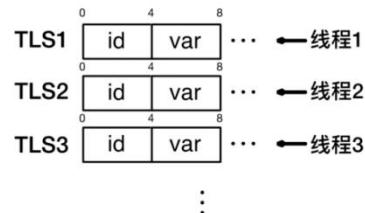
- 线程库允许定义每个线程独有的数据
 - `__thread int id;` 会为每个线程定义一个独有的id变量

- 每个线程的TLS结构相似

- 可通过TCB索引

- TLS寻址模式：基地址 + 偏移量

- X86: 段页式 (fs寄存器)
 - AArch64: 特殊寄存器tpidr_el0



X86用fs保存当前线程的TLS基地址，AArch64则使用特殊寄存器tpidr_el0

用下划线定义的就是每个线程私有的变量。它存储的方式就是一个基地址+偏移量的方式。

线程的基本操作：以

- 创建：`pthread_create`
 - 内核态：创建相应的内核态线程及内核栈
 - 应用态：创建TCB、应用栈和TLS
- 合并：`pthread_join`
 - 等待另一线程执行完成，并获取其执行结果
 - 可以认为是fork的“逆向操作”



线程的基本操作：以

- 退出：`pthread_exit`
 - 可设置返回值（会被pthread_join获取）
- 暂停：`pthread_yield`
 - 立即暂停执行，出让CPU资源给其它线程
 - 好处：可以帮助调度器做出更优的决策

然后我们的线程有一些基本的操作，就不详细讲了。

上下文切换

上下文切换的话，什么叫上下文？就是从一个线程切换到另一个线程。上下文切换是在哪完成的？我们将来看到的是一对一的，内核态线程是应该在内核态切换的。

进入内核态可以通过 `syscall` 或者 `interrupt`、外部中断，都可以使得线程进入内核态。

如何实现上下文切换？

如何实现上下文切换？

- 进程怎样切换到内核中执行？如何切换回用户态？
- 如何对上下文进行保存和恢复？
- 进程怎样切换到内核中执行？如何切换回用户态？
- 如何对上下文进行保存和恢复？
- 如何实现关键的切换步骤？



进程上下文的组成 (AArch64)

- **进程上下文需要包含哪些内容？**
 - 常规寄存器：X0-X30
 - 程序计数器（PC）：保存在ELR_EL1
 - 栈指针：SP_EL0
 - CPU状态（如条件码）：保存在SPSR_EL1
- **思考：为什么进程上下文只需要保存寄存器信息，而不用保存内存？**

首先要包括常规的寄存器，程序计数器，栈指针，CPU状态。

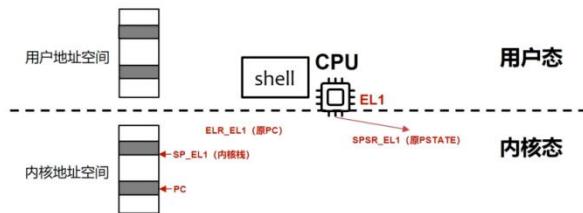
Q: 为什么进程上下文只需要保存寄存器信息，而不用保存内存呢？

A: 因为内存的数据不会因为切换而消失，但寄存器只有一组。

首先我们要切换到内核态来执行

进程的内核态执行：切换到内核态

- **AArch64提供了硬件支持，使进程切换到内核态执行**
 - 状态 (PSTATE) 写入SPSR_EL1 - 原PC值写入ELR_EL1
 - 栈指针寄存器切换到SP_EL1 - 运行状态切换到内核态EL1
 - PC移动到内核异常向量表中

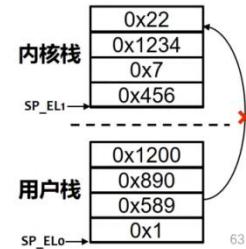


进程的内核态执行：内核栈

- 为什么需要“又一个栈”（内核栈）？
 - 进程在内核中依然执行代码，有读写临时数据的需求
 - 进程在用户态和内核态的数据应该相互隔离，增强安全性

- AArch64实现：两个栈指针寄存器

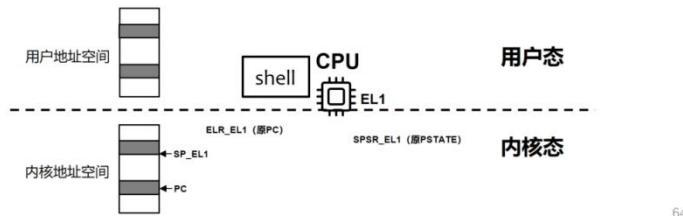
- SP_EL1, SP_ELO
 - x86只有一个栈指针寄存器，需要保存恢复



我们还有一个内核的栈，为什么需要内核栈呢？因为进程在内核中也需要执行代码，也要执行读写数据，而且进程在用户态和内核态的数据应该是相互隔离的。因为有隔离机制，我们在内核中也不能使用用户态的栈。

进程的内核态执行：返回用户态

- 进入内核态的“逆过程”，AArch64同样提供了硬件支持

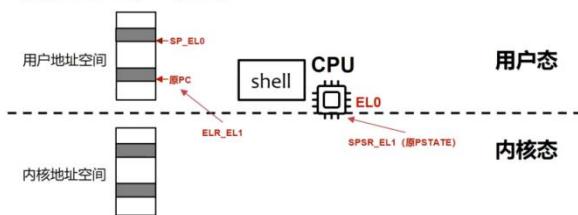


64

进程的内核态执行：返回用户态

- 进入内核态的“逆过程”，AArch64同样提供了硬件支持

- SPSR_EL1重设到CPU PSTATE
 - ELR_EL1重设到PC寄存器中
 - 栈指针寄存器切换到SP_ELO
 - 运行状态切换到用户态EL0
 - （为什么少了一步？）

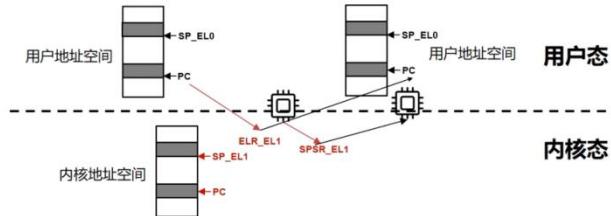


65

少了一步的原因是因为内核态的PC不需要保存，所以PC只需要从ELR_EL1中恢复即可。

内核/用户态切换与上下文切换

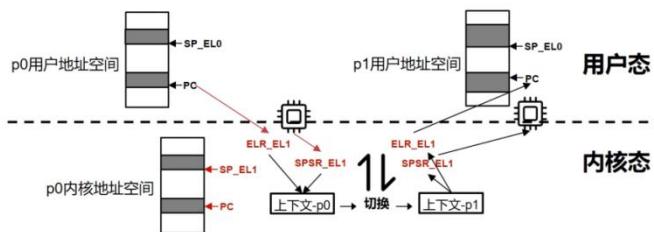
- 通过支持内核/用户态切换，可以实现进程进入内核态并返回（比如system call）



这样就可以实现进程进入到内核态，再进行返回的情况。内核到用户态的切换有上下文切换、模式切换（mode switch）。

内核/用户态切换与上下文切换

- 但是该机制并不足以实现上下文切换
 - 不同进程地址空间不同，使用的寄存器值也不同（如PC）
 - 但是：寄存器只有一个！直接恢复会导致错误
 - 解决方法：保存上下文（寄存器）到内存，用于之后恢复

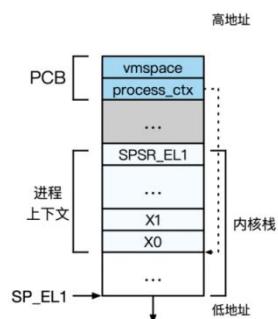


我们把所有的通用寄存器和状态寄存器都保存到内存里。

上下文与其他内核数据结构

- 与进程相关的三种内核数据结构：PCB、上下文、内核栈

- PCB保存指向上下文的引用
- 上下文的位置固定在内核栈底部



在这里的话，上下文的位置固定在内核栈底部。剩下的部分，再用来作为进程在内核中执行的栈的地址。

上下文保存 (ChCore为例)

- 进入内核后调用exception_enter完成

- 将各寄存器逐一放入内核栈中

```
1 .macro exception_enter
2 sub sp, sp, #ARCH_EXEC_CONT_SIZE           内核栈变化
3 // 保存通用寄存器 (x0-x29)
4 stp x0, x1, [sp, #16 * 0]      SP_EL1→
5 stp x2, x3, [sp, #16 * 1]
6 stp x4, x5, [sp, #16 * 2]
7 ...
8 stp x28, x29, [sp, #16 * 14]
9 // 保存 x30 和上文提到的三个特殊寄存器: sp_el0, elr_el1,
   → spsr_el1
10 mrs x21, sp_el0
11 mrs x22, elr_el1
12 mrs x23, spsr_el1
13 stp x30, x21, [sp, #16 * 15]
14 stp x22, x23, [sp, #16 * 16]
15 .endm
```



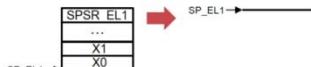
首先保存通用寄存器，然后再保存三个特殊寄存器 sp_el0,elr_el1,spsr_el1。

上下文保存 (ChCore为例)

- 上下文保存的“逆过程”：调用exception_exit完成

- 从内核栈读取出各寄存器，并清空内核栈，最后调用eret

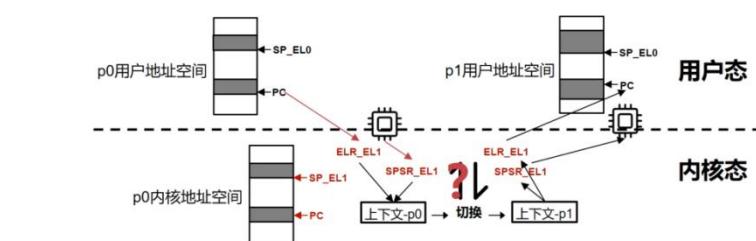
```
1 .macro exception_exit
2 ldp x22, x23, [sp, #16 * 16]           内核栈变化
3 ldp x30, x21, [sp, #16 * 15]
4 // 恢复三个特殊寄存器
5 msr sp_el0, x21
6 msr elr_el1, x22
7 msr spsr_el1, x23
8 // 恢复通用寄存器 X0-X29
9 ldp x0, x1, [sp, #16 * 0]
10 // ...
11 ldp x28, x29, [sp, #16 * 14]
12 add sp, sp, #ARCH_EXEC_CONT_SIZE
13 eret
14 .endm
```



恢复的过程就是从内核栈里把寄存器的信息读出来，再把内核栈清空。再调用 eret。如果我们把上下文保存/恢复好了以后，此时只需要切换了。

万事俱备，只欠切换！

- 上下文保存/恢复机制为进程间切换奠定了基础
- 思考：最关键的切换包含哪些步骤呢？



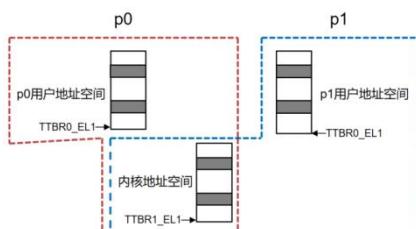
• 思考：最关键的切换包含哪些步骤呢？

- 关键1：如何切换到 p1 的地址空间？
- 关键2：如何切换到已经存储的 p1 上下文并进行恢复？

步骤1：地址空间的切换

• 回顾：AArch64 的地址空间管理

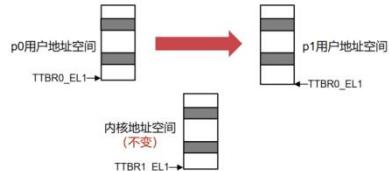
- 内核与用户态地址空间分开管理（使用两个寄存器）
- 用户地址空间独有，内核地址空间共享



步骤1：地址空间的切换

• 回顾：AArch64 的地址空间管理

- 内核与用户态地址空间分开管理（使用两个寄存器）
- 用户地址空间独有，内核地址空间共享
- 因此，只需要实现用户地址空间切换即可

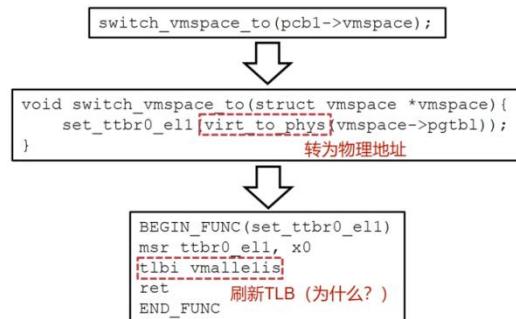


分别使用两个寄存器，一个是 TTBR0_EL1 和 TTBR1_EL1。这个切换非常简单，只需要实现用户态地址空间的切换即可。

步骤1：地址空间的切换

• ChCore 的实现：switch_to_vmspace

- 获取 p1 的 PCB，并获取其 vmspace，最后设置 TTBR0_EL1



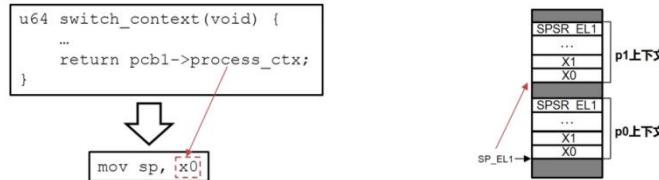
它的实现就是 switch_vmspace_to 就行了。这里会刷新 TLB。刷新 TLB 的原因是虚拟地址空间的切换会导致 TLB 中的 entry 失效。

步骤2：如何切换到p1的上下文？

- 回顾：当p1保存上下文时，其内核栈应该是怎样的？



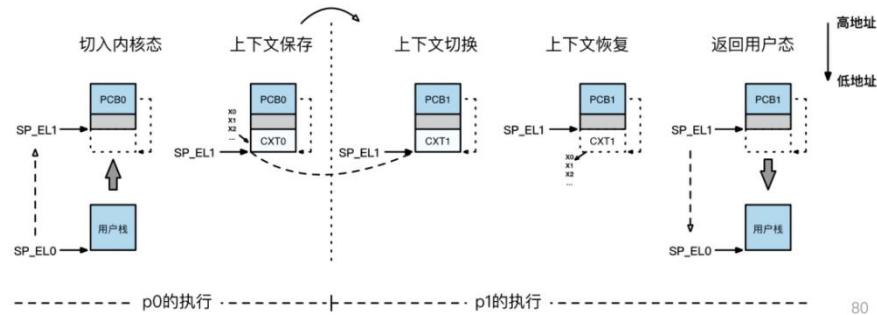
- p0/p1共享内核地址空间，因此直接切换内核栈指针即可



怎么切换到 p1 的上下文？保存上下文的时候，栈应该如上图右下角所示。我们共享内核地址空间，只需要切换栈指针就可以了。SP_EL1 从 p0 切换为 p1，就完成了切换。

总结：上下文切换栈变化全过程

- 共涉及两次权限等级切换、三次栈切换
- 内核栈的切换是线程切换执行的“分界点”



上下文切换设计 2 次权限等级的切换（从用户态到内核态，从内核态返回用户态）。

2022/3/11

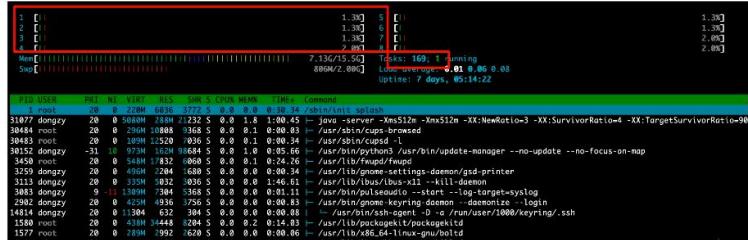
处理器调度

今天我们讨论的话题是处理器调度。它是 OS 里很重要的一个功能。

Q：为什么需要调度？

A：因为 OS 里运行任务的数量是远远大于处理器数量的。

系统中的任务数远多于处理器数

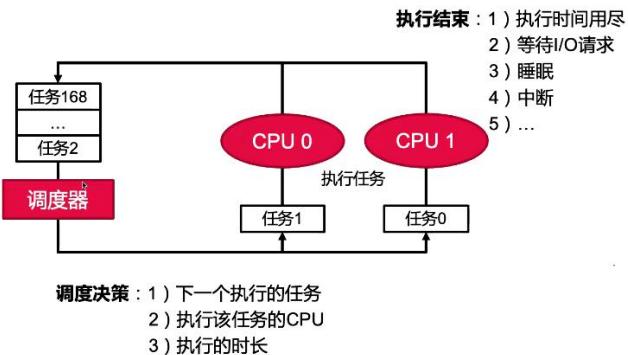


任务 (Task) : 线程、单线程进程

仅有8个处理器，如何运行169个任务？

如果我们有 100 个核，但是我们只有 10 个应用程序（线程），那么此时是不需要做任何调度的。但是现实中，运行任务的数量是远超过处理器核的。上图中，我们看到 TASK 的数量有 169 个，在服务器上的任务的数量会更多，比如 nginx 对每个用户请求都会新建一个 thread。所以任务数多了以后就势必需要把有限的 CPU 调度到不同的任务上去执行。

进程/线程调度



这是调度整体的 Big Picture。调度器有一个任务队列，把所有任务排序。调度器挑选出两个任务分别给到两个 CPU 来执行，把 thread 的上下文恢复到处理器上以后就可以执行了。有一个调度决策的问题，决定下一个调度的任务是什么？

在真实系统里，有比 FIFO 更多的方式。一旦选出了任务以后，我们也要决定放在哪个 CPU core 上面，我们要避免一个 CPU 非常繁忙但其他 CPU 没事做的情况（负载不均衡）。其次就是任务的时长，其实就是时间片的概念。如果一个任务在 CPU 上一直在运行，那是不行的，我们要让别的任务也有机会来运行。如果任务不调用 yield，我们需要有强制的方法切换到内核来设置下一个运行的任务。

这个时间片怎么设置也很有意思，比如桌面机器的时间片会短一些，因为轮转会越快，响应时间会越快，任何一个任务过 1 个时间片都有机会运行到；而服务器上的时间片会长一些。在 Linux 编译的时候，就有一个选项问你默认时间片设置多长。

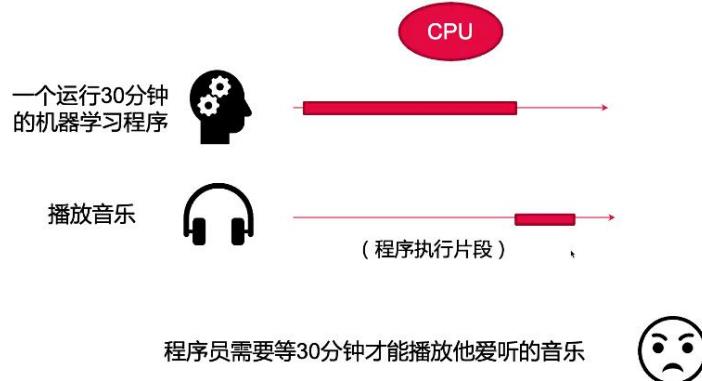
那么什么时候会执行结束呢？比如我们设置了 10ms 时间片，并不是每次到了 10ms 才执行结束，还有等待 I/O 的执行结束（DMA 结束以后会给 CPU 发一个中断），这也会把当前的任务中断掉，也会结束执行。睡眠调用 yield 也会结束执行。“执行时间用尽”这件事情也是通过中断来实现的。

这就是我们整个调度的过程，我们并没有说该怎么去调度，那是策略问题。怎么分配到 CPU 也不管，是选择策略来负责的事情。执行结束后重新恢复到内核态去运行切换到下一个任务。整个 OS 不断地在这里运行一个死循环。

Q: 如果队列空了怎么办?

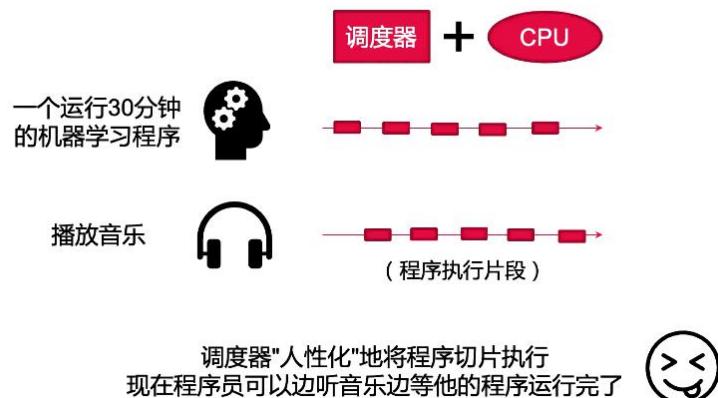
A: 不会空, 因为有 IDLE 任务, 只要一执行, CPU 就会进入低功耗的状态。IDLE 是 per-CPU 的, 10 个核就会有 10 个 idle 任务。

如果没有调度器



如果没有调度器? 没有分时复用的机制的话, 我们运行一个程序 30 分钟, 此时电脑就是假死状态, 不能响应任何鼠标键盘的输入。

调度器让生活更美好



调度的本质是把请求和资源做对接, 我们的请求可以认为是多个 thread 任务, 资源就是 CPUcall。

什么是调度?

协调请求对于资源的使用



思考：还有哪些调度适用的场景？

- I/O (磁盘)
- 打印机
- 内存
- 网络包
- ...

一旦从这个角度去看调度，还有很多别的场景适合调度，比如我们的磁盘的 IO 请求，我们要读某个磁盘上的 block 的位置，很多应用程序在同一时间段会发出很多 IO 请求，在这个情况下，资源就是磁盘，请求就是读 block 的请求，如果我们没有很好的调度，那么磁头就会以很混乱的方式读。

打印机面对多人的打印请求。比如先到先得，把打印数量多的请求往后放。读内存的请求会从多个 CPU 发到内存上，此时内存控制器就要决策先服务哪个 CPU 的请求，从而可以做排序。

网络包也是一样的，我们要考虑路由器会有多条线连进来。路由器/交换机到底应该如何去选择网络包发送，也会涉及到调度的问题。

如果我们有一个算法在 CPU 场景下工作得很好的，如果我们应用到其他场景也会存在很多的相似性，算法是可以通用的。

调度在不同场景下的目标



CPU 调度在不同场景下的目标是不一样的。批处理系统接收来自不同用户的任务，每个任务做完再做下一个，它的目标就是吞吐量，让 CPU 满负荷运转，但是高吞吐量不一定是低时延，所以在交互式系统中，低响应时间就更重要，比如我们同时在打字和编译，我们宁愿编译慢一点。网络服务器要考虑性能的增长，要考虑可扩展的问题。移动设备主要是低响应和低功耗，如果 CPU 一直在运行导致没电了也不行。还有一个就是实时调度，它是低响应的极端情况，要在规定范围内的低响应，不能超过一个阈值。

它们的共同目标就是不能发生 starvation 的情况，还有低调度开销。

调度器的目标

- **降低周转时间**：任务第一次进入系统到执行结束的时间
- **降低响应时间**：任务第一次进入系统到第一次给用户输出的时间
- **实时性**：在任务的截止时间内完成任务
- **公平性**：每个任务都应该有机会执行，不能饿死
- **开销低**：调度器是为了优化系统，而非制造性能BUG
- **可扩展**：随着任务数量增加，仍能正常工作
- ...

一旦没有办法满足所有目标，就会为了不同场景设置不同的策略满足不同的目标，并且策略之间会有 tradeoff。用户体验要好，有截止时间（DDL），公平性（每个任务有机会执行，不一定是绝对公平，但是不能饿死）。

Q: 如果我们的系统里有一个任务的优先级就是比另一个优先级高？

A: 会有可能发生 starvation，这是公平性和 starvation 的权衡。还有可能是出问题了。

有些调度器工作非常复杂，可能是 $O(N^2)$ 的复杂度，一旦任务量到达 10000，调度速度就很慢。Linux 的调度算法复杂度是 $O(1)$ 。它的排序的依据就是根据优先级。

调度的挑战

- **缺少信息（没有先知）**
 - 工作场景动态变化
- **线程/任务间的复杂交互**
- **调度目标多样性**
 - 不同的系统可能关注不一样的调度指标
- **许多方面存在取舍**
 - 调度开销 V.S. 调度效果
 - 优先级 V.S. 公平
 - 能耗 V.S. 性能
 - ...

如果我们知道每个任务的特征和运行的时间，那么调度起来很简单。但是实际上这个信息是不存在的，而且还有可能下一个任务依赖于上一个任务。或者任务 a 会给任务 b 发一个 IPC，交互复杂。并且目标多样，必须做权衡。

策略 V.S. 机制

- 策略
 - 做什么
 - 从上层去分析、解决问题
- 机制
 - 怎么做
 - 实现某一策略、功能

主题	策略	机制
上课	《OS章节：调度》	课堂、网课
上课	签到	二维码、点名
科研	写C++代码	VSCode、Sublime
科研	写论文 (Latex)	VSCode、Sublime

机制的调整是比较少的，但是具体上哪节课是无所谓的。

OS的调度策略

- 为了满足不同需求提供多种调度策略
- 以Linux两种调度器为例，每种对应多个调度策略
 - Complete Fair Scheduler (CFS)
 - SCHED_OTHER
 - SCHED_BATCH
 - SCHED_IDLE
 - Real-Time Scheduler (RT)
 - SCHED_FIFO
 - SCHED_RR

Linux 的默认调度算法是 CFS，它是为了公平性而提出的，内部也有一些不同的参数。OS 内部也有一整套支持配置。从 1 核到 128 核，已经跨了两个数量级，用同一个调度算法的话是不太行的。这意味着将来我们的算法会越来越多样化。

经典调度

CPU调度与提问调度



当前假设每个同学只提一个问题

我们通过一个例子来做一个类比。学霸脑子转的快就是 CPU，而任务就是同学的问题。

先到先服务 (First Come First Served)

同学	到达时间	解答时间 (工作量)
A	0	4
B	1	7
C	2	2

好处：简单、直观
问题：平均周转、响应时间过长



短任务优先 (Shortest Job First)

同学	到达时间	解答时间 (工作量)
A	0	4
B	1	7
C	2	2

好处：平均周转时间短
问题：1) 不公平，任务饿死
2) 平均响应时间过长



这就会发生一个问题叫做 starvation。

抢占式调度 (Preemptive Scheduling)

- 每次任务执行
 - 一定时间后会被切换到下一任务
 - 而非执行至终止
- 通过定时触发的时钟中断实现

我们通过定时触发中断的方式，来轮流运行。

时间片轮转 (Round Robin)

同学	到达时间	解答时间 (工作量)
A	0	4
B	1	7
C	2	2

好处：公平、平均响应时间短
问题：牺牲周转时间



这个问题在于 A 要等待比较长的时间，而且轮询因为要来回切换，上下文切换本身的

开销也是不能忽略的。

思考：

- 什么情况下RR的周转时间问题最为明显？

- 时间片长短应该如何确定？

- 过长的时间片会导致什么问题？
- 过短的时间片会导致什么问题？

A: 如果 ABC 是同时到的，问问题的时间也差不多，意味着最后每个人都用了 3 倍的时间完成任务，每个人的等待时间都变长了。

A: 如果我们效率（吞吐量）要更高，时间更长；如果我们响应时间要更低，那么我们设置时间片短一些。

优先级调度

它是为了解决没有优先级所带来的问题。

调度优先级

- 操作系统中的任务是不同的，例如：

- 系统 V.S. 用户、前台 V.S. 后台、...

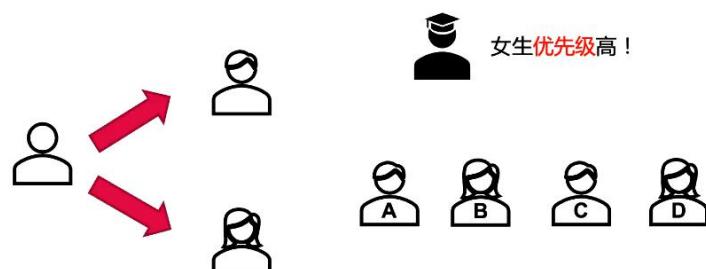
- 如果不加以区分

- 系统关键任务无法及时处理
- "后台运算"导致"视频播放"卡顿

- 优先级用于确保重要的任务被优先调度

正因为任务是不一样的，所以我们才要对任务有优先级。毫无疑问前台的任务优先级更高，因为用户能看到前台的任务，否则后台编译任务可能导致前台视频播放卡顿等。

添加条件：优先级



思考：优先级

- 以下这些调度策略算不算优先级调度？

- First Come First Served
- Shortest Job First
- Round Robin

之前的调度策略有没有优先级的概念？它可以认为是谁先到，谁优先级高、谁最短，谁优先级高，而轮询并没有优先级的概念。那么我们应该怎么设置优先级呢？系统需要有动态的判断机制。

多级队列

多级队列 (Multi-level Queue)

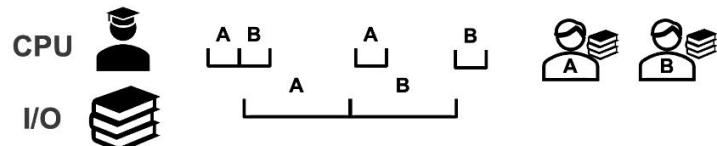
- 1) 维护多个队列，每个对应静态设置好的优先级
- 2) 高优先级的任务优先执行
- 3) 同优先级内使用Round Robin调度（也可使用其他调度策略）



这是一个极度不公平的策略，低优先级需要等待高优先级全部完成。

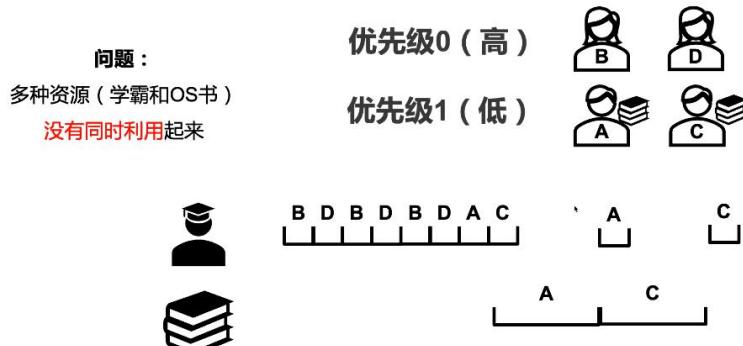
添加条件：阅读OS书（类比I/O操作）

- 学霸告诉同学需要看OS书
 - （学霸只有一本OS书，同一时间只有一个同学能够阅读）
- 阅读完OS书后，同学再和学霸确认知识点



OS书就有点像DMA的感觉，看完书了以后，同学可能再去问一下学霸是不是这样理解的。

低资源利用率问题



如果 B 和 D 一直在问学霸，那么 A 和 C 不能接触到学霸。更理想的情况下，是 CPU 让任务 A 去 I/O，然后 CPU 再处理 B 和 D。这样 CPU 和磁盘的资源可以同时复用。

思考：优先级的选取

- 什么样的任务应该有高优先级？

- I/O密集型任务
 - 如：用户交互、磁盘访问
 - 为了更高的I/O资源利用率
- 时延要求极高（必须在短时间内完成）的任务
 - 如：音频播放、游戏
- 用户主动设置的重要任务

I/O 密集型任务主要和用户交互比较多，而且磁盘访问多，大量时间在等磁盘，这样对 CPU 来说，它需要做完一段小运算后让任务尽可能快地等磁盘，这类任务等待 CPU 的时间越长，磁盘的利用率越低。所以 I/O 密集型的任务，优先级越高越好。

时延要求极高（放视频、游戏）和用户主动设置的任务。

优先级的动态调整

- 操作系统中的工作场景是**动态变化的**
- 静态设置的优先级可能导致
 - 资源利用率低
 - 一个CPU密集型可能动态转变为I/O密集型任务
 - 优先级反转
 - ...
- 所以，优先级需要**动态调整**

优先级反转

我们给任务设置静态的优先级，就会导致资源利用率低。静态设置优先级还会导致优先

级反转的问题。

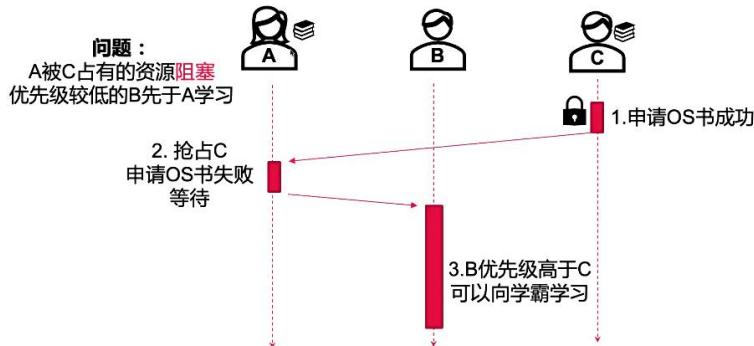
静态优先级的问题：优先级反转

- 任务都需要独占共享资源
 - 共享资源
 - 存储
 - 硬件
 - OS书
 - ...
 - 通常使用信号量、互斥锁实现独占（后续课程讲解）
- 低优先任务占用资源 -> 高优先级任务被阻塞

通常为了避免大家竞争，我们使用信号量和互斥锁的方式实现独占。我们看如下的一个例子：

静态优先级的问题：优先级反转

优先级：A>B>C

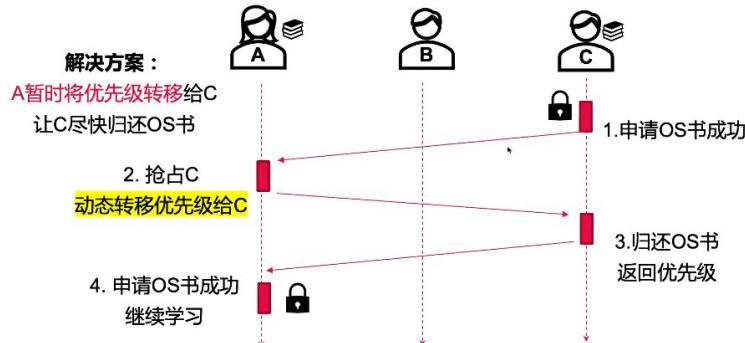


C先到申请读磁盘成功，A也想读磁盘，但是读磁盘失败，必须等着C把锁放掉（busy waiting）。这时候B来了，B就可以运行了。问题在于B在运行的时候，C就不能运行了。所以变成了C在等B，而A在等C。这就等于A在等B，但是A的优先级较高，A不应该等B。

锁的机制和调度优先级的机制结合在一起的时候就违反了优先级高先运行的要求。

解决方法：动态优先级继承

优先级：A>B>C



这就需要优先级继承的问题。一旦当A发现在等一个优先级很低的人之后，就将自己的优先级动态转移给C。等到C运行完了之后，再把锁放掉，把优先级还给A。如果此时B在中间运行，B也没有办法抢占C。

目前介绍的调度策略的限制

- **周转时间、响应时间过长**
 - FCFS
- **依赖对于任务的先验知识**
 - 需要预知任务执行时间
 - SJF
 - 需要预知任务是否为I/O密集型任务
 - MLQ (用于设置任务优先级)
- **假设调度没有开销**
 - RR (将时间片设置过短会导致调度开销过大)

这些策略都不是完美的机制。有没有一种调度算法可以解决所有问题呢？

多级反馈队列

多级反馈队列

- **Multi-Level Feedback Queue (MLFQ)**
- **Corbató**
 - 于1962年，发表了Compatible Time-Sharing System CTSS)的相关论文
 - 在该论文中提出了MLFQ以及其它概念
 - 于1990年，获得图灵奖
 - 因CTSS和Multics方面的贡献



他提出的 MLFQ 就是很重要的一个算法。

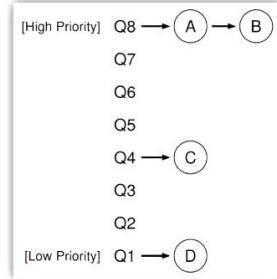
MLFQ的主要目标与思路

- **一个无需先验知识的通用调度策略**
 - 周转时间低、响应时间低
 - 调度开销低
- **通过动态分析任务运行历史，总结任务特征**
 - 类似思想的体现：分支预测、缓存
 - 需要注意：如果工作场景变化频繁，效果可能会很差

它不需要知道进程的先验知识，比如负责的是 I/O 密集的还是机器学习的。如果我们需要花 100ms 知道这个是什么类型，如果接下来 10s 都是这个类型，那就比较合适，如果进程每毫秒类型都不一样，那么效果就很差。

基本算法 (基于Multi-Level Queue)

- 规则 1:
 - 优先级高的任务会抢占优先级低的任务
- 规则 2:
 - 每个任务会被分配时间片，优先级相同的两个任务使用时间片轮转



如何设置任务优先级？

- 针对混合工作场景
 - 执行时间短的任务
 - 交互式任务
 - I/O密集型任务
 - 执行时间长的任务
 - CPU密集型计算任务
- 规则 3:
 - 任务被创建时，假设该任务是短任务，为它分配最高优先级
- 规则 4a:
 - 一个任务时间片耗尽后，它的优先级会被降低一级
- 规则 4b:
 - 如果一个任务在时间片耗尽前放弃CPU，那么它的优先级不变
 - 任务重新执行时，会被分配新的时间片

我们认为时间片用完了，就是一个长时间的任务。主动放弃 CPU 很有可能是调用 I/O 读磁盘，调用 `syscall read/write` 进入 kernel 之后就去读磁盘，此时就可以 `yield` 等待 I/O。

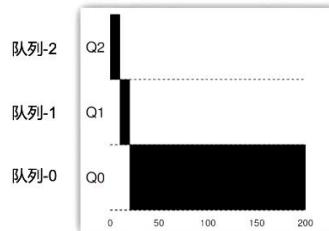
样例执行1、2

对于长任务：

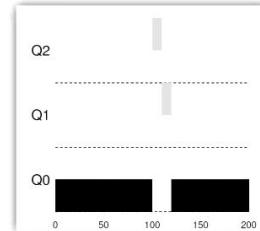
- MLFQ会逐渐降低它的优先级
- 并将它视为长任务

对于短任务：

- 它会很快执行完，证明自己是个短任务



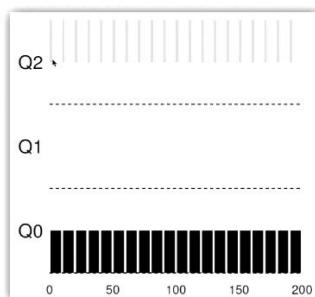
1. 一个长任务的执行



2. 长任务执行时，一个短任务被创建

39

样例执行3



对于I/O密集型任务：

- 它会在时间片执行完以前放弃CPU
- MLFQ保持它的优先级不变即可

3. 混合CPU密集型与I/O密集型任务的执行

基本算法的问题（一）

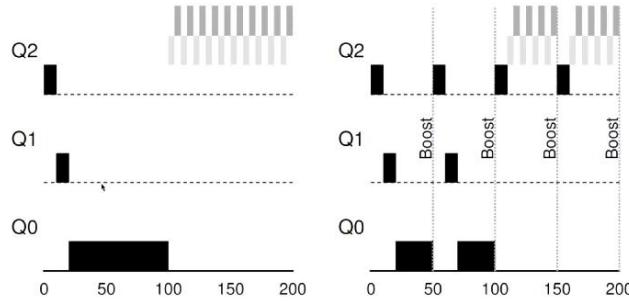
- **长任务饿死 (Starvation)**
 - 过多的短任务、I/O密集型任务可能占用所有CPU时间
- **任务特征可能动态变化**
 - CPU密集型任务→交互式任务，…

定时优先级提升 (Priority Boost)

- **规则 5:**
 - 在某个时间段S后，将系统中所有任务优先级升为最高
- **避免长任务饿死**
 - 所有任务的优先级会定时地提升最高
 - 最高级队列采用RR，长任务一定会被调度到
- **针对任务特征动态变化的场景**
 - MLFQ会定时地重新审视每个任务

这样有一个好处，就算最开始是低优先级的任务，一段时间之后有机会证明自己转变成了短时间任务。避免积累了一些不正确的状态。

样例执行4



4. 采用定时优先级提升的前后对比 (左为采用前 , 右为采用后)

基本算法的问题 (二)

- 无法应对抢占CPU时间的攻击 (Gaming Scheduler Attack)

- 恶意任务在时间片用完前发起I/O请求
 - 避免MLFQ将该任务的优先级降低，并且每次重新执行时间片会被重置
 - 几近独占CPU！

如果我们在时间片用完之前一点点时间放掉，我们又保证了自己的优先级又独占了CPU。

更准确地记录执行时间 (Better Accounting)

- 规则 6:

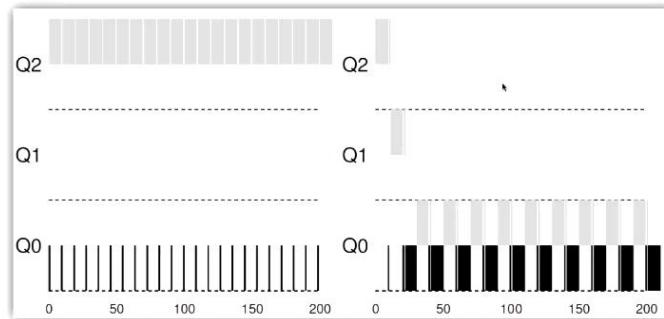
- 一个任务时间片耗尽后（无论它期间放弃了多次CPU，它的时间片不会被重置），它的优先级会被降低一级

- MLFQ会：

- 记录每个任务在当前优先级使用的时间片
 - 当累计一个完整时间片被用完后，降低其优先级

这样就是有人降得快，有人降得慢。

样例执行5



5. 使用准确记录执行时间的前后对比（左为采用前，右为采用后）

MLFQ的参数调试

- **如何确定MLFQ的各种参数？**

- 优先级队列的数量
- 不同队列的时间片长短
- 定时优先级提升的时间间隔

- **每个参数都体现了MLFQ的权衡**

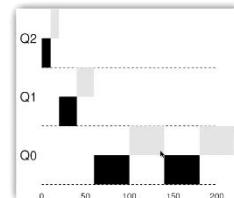
- 对于不同的工作场景，不同的参数会导致不一样的表现

可以看到 MLFP 有很多参数都可以调试，比如 Boost 的时间。

MLFQ各个队列时间片长短的选择

- **为不同队列选择不同的时间片**

- 高优先级队列时间片较短，针对短任务
 - 提升响应时间
- 低优先级队列时间片较长，针对长任务
 - 降低调度开销



这样就可以针对长任务的场景没有必要这么快的切换。

MLFQ总结

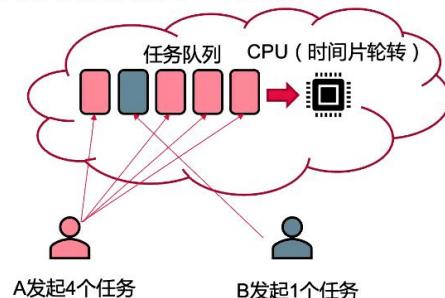
- **Multi-Level Feedback Queue**
 - 通过观察任务的历史执行，动态确定任务优先级
 - 无需任务的先验知识
 - 同时达到了周转时间和响应时间两方面的要求
 - 对于短任务，周转时间指标近似于SJF
 - 对于交互式任务，响应时间指标近似于RR
 - 可以避免长任务的饿死
- **许多著名系统的调度器是基于MLFQ实现的**
 - BSD, Solaris, Windows NT 和后续Windows操作系统

这个动态行为就算每次使用时间片的模式。

公平共享调度

场景：共享服务器

- A和B两位同学合资买了一台服务器，他们每人负担了一半的费用
- 两人应均分CPU时间
 - 而非被发起的任务数量决定
- 如果CPU使用时间片轮转调度
 - A占用80%CPU时间
 - B占用20%CPU时间



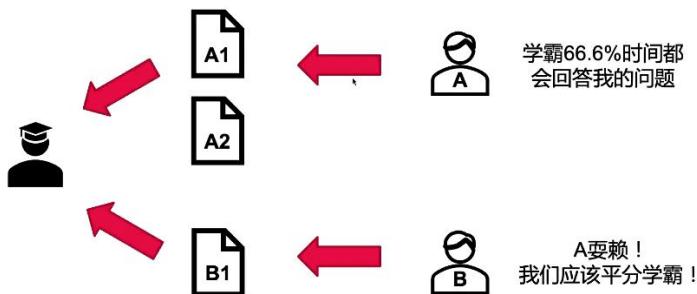
在服务器上共享，应该考虑费用问题，谁用 CPU 多，就应该多付钱。

公平共享（用户级，非任务级）

- 每个用户占用的资源是成比例的
 - 而非被任务的数量决定
- 每个用户占用的资源是可以被计算的
 - 设定“权重值”以确定相对比例（绝对值不重要）
 - 例：权重为4的用户使用资源，是权重为2的用户的2倍

每个用户所占用的资源必须成比例。

添加条件：一个同学会问多个问题



方法：使用"ticket"表示任务的权重

- ticket : 每个问题对应的权重

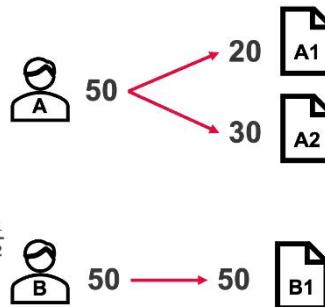
- T : ticket的总量

- 问题A1可占用学霸时间的比例

$$-\frac{\text{ticket}_{A1}}{T} = \frac{20}{100} = \frac{1}{5}$$

- 同学A可占用学霸时间的比例

$$-\frac{\text{ticket}_A}{T} = \frac{\text{ticket}_{A1} + \text{ticket}_{A2}}{T} = \frac{50}{100} = \frac{1}{2}$$



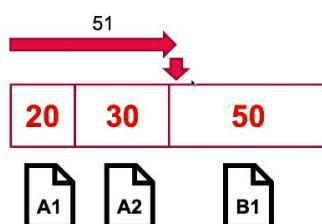
Ticket 就是每个任务对应的权重。问题 A1 就是 20 张票。问题 B1 就算 50 张票。CPU 会把 ticket 平分给用户。等到大家用完了再发 50。

公平共享：彩票调度 (Lottery Scheduling)

- 每次调度时，生成随机数 $R \in [0, T]$

- 根据R，找到对应的任务

- $R=51 \rightarrow$ 调度B1

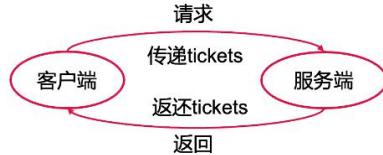


```
R = random(0, T)
sum = 0
foreach(task in task_list) {
    sum += task.ticket
    if (R < sum) {
        break
    }
}
schedule()
```

每次调度的时候会生成一个随机数 R，根据 R 就可以调度对应的任务。执行一个时间片再摇下一个彩票，长期来看 A1+A2 和 B1 是平分时间的，这样就可以保证整个过程的权重。在有一些场景里还可以让客户端的票送回服务端，比如客户端调用 RPC。

Ticket Transfer

- 场景：
 - 在通信过程中，客户端需要等到服务端返回才能继续执行
- 客户端将自己所有的ticket移交给服务端
 - 确保服务端可以尽可能使用更多资源，迅速处理
- 同样适用于其他同步场景



Q: 权重和优先级到底有什么区别呢？

思考：权重与优先级的异同？

- 权重影响任务对CPU的占用比例
 - 不会有任务饿死
- 优先级影响任务对CPU的使用顺序
 - 可能产生饿死

只要占用了 1% 的 CPU，也不会被饿死。但一直存在高优先级的任务的话，可能会饿死。

思考：随机的利弊

- 随机的好处是？
 - 简单
- 随机带来的问题是？
 - 不精确——伪随机非真随机
 - 各个任务对CPU时间的占比会有误差

随机的一个好处当然是简单，只需要生成一个随机数即可。但是在短时间内可能产生不均衡的情况。

Stride Scheduling

- 可以看做**确定性版本的Lottery Scheduling**
 - 可以沿用tickets的概念
- **Stride——步幅，任务一次执行增加的虚拟时间**

$$- \text{stride} = \frac{\text{MaxStride}}{\text{ticket}}$$

- MaxStride是一个足够大的整数
- 本例中设为所有tickets的最小公倍数

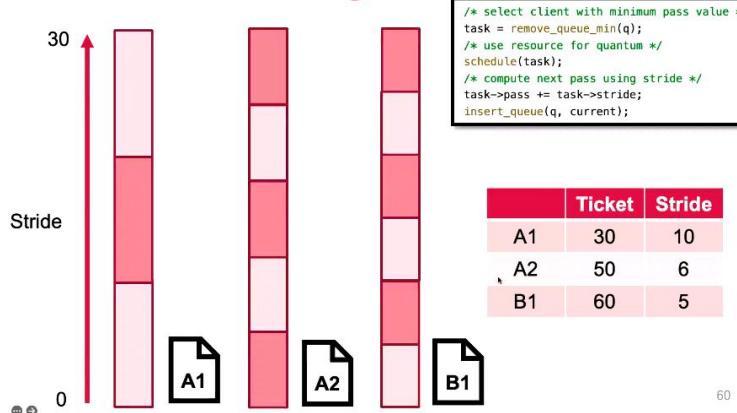
- **Pass——累计执行的虚拟时间**

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5

$$\text{MaxStride} = 300$$

它是可以认为沿用的 ticket 的概念。我们取所有 ticket 的最小公倍数 300，所以 A1 每做一个单位的运算相当于迈出了 10 步。

Stride Scheduling



每次调度的时候，我们选择走的最少的那个。走的最少代表占的 CPU 最少，也意味着没有被公平对待，所以我们就让占的最少的多走一步。相同的情况下，我们随便选就行了。所以我们就有 A1->A2->B1->A2->A1->B1->A2->....

公平共享调度

	Lottery Scheduling	Stride Scheduling
调度决策生成	随机	确定性计算
任务实际执行时间与预期的差距	大	小

预期——根据任务权重计算的执行时间期望

在实际中我们采用的是 Stride Scheduling，在 Linux 中叫做 virtual time。

实时调度

现在在智能驾驶里越来越火。

实时调度

- 每个任务都有截止时间 (Deadline)
- 软实时 (Soft Real Time)
 - 视频播放，每一帧的渲染
 - 超过截止时间 → 画质差
- 硬实时 (Hard Real Time)
 - 自动驾驶汽车的刹车任务
 - 超过截止时间 → 严重后果

速度 (千米/小时)	速度 (米 / 秒)	停车距离(米) 干地	停车距离(米) 湖地	停车距离(米) 雪地
60	16.67	17.15	25.72	51.44
90	25.00	38.58	57.87	115.74
120	33.33	68.59	102.88	205.76
150	41.67	107.17	160.75	321.50

不同条件下的刹车距离

早一秒距离就非常的多。在调度里最著名的就是最早 DDL First。

最早截止时间优先 (Earliest Deadline First)

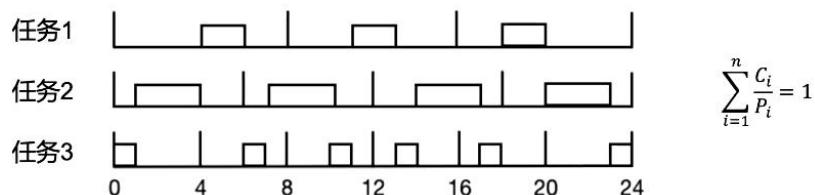
- 任务
 - C : 所需执行时间
 - P : 任务触发的时间周期
 - 假设P同时是任务截止时间
 - 反映了CPU利用率
- ($\sum_{i=1}^n \frac{C_i}{P_i} \leq 1$)
- 当时任务满足条件：
 - 这些任务对EDF是可调度的
 - 所有任务在截止时间前完成

我们要去算任务所需要的时间和任务触发周期的时间，我们加在一起除一下必须小于等于 1。

最早截止时间优先 (Earliest Deadline First)

- 每次调度截止时间最近的任务
- EDF是动态算法
 - 无需预知执行时间、任务周期
- 在任务可调度的情况下能够实现最优调度

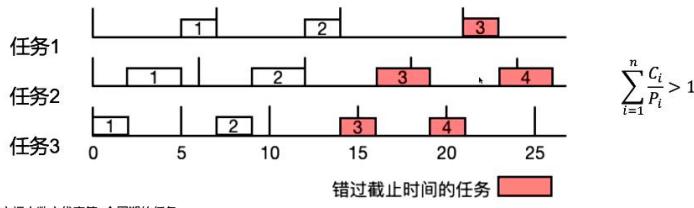
	执行时间	任务周期/截止时间
任务1	2	8
任务2	3	6
任务3	1	4



多米诺效应

- 在任务不可调度时，会造成多数任务都错过截止时间

	执行时间	任务周期/截止时间
任务1	2	7
任务2	3	6
任务3	2	5



一旦接了新的任务，会导致其他的任务全部失效。

最早截止时间优先的实际应用

- 视频实时渲染系统
 - 视频帧需要在被播放时间点前被渲染
 - 若超过截止时间
→降低后续视频帧的清晰度
- 车载防撞系统，定时处理车载雷达信号
 - 雷达信号必须在给定截止时间内被处理
 - 若超过截止时间
→需要立即减速或刹车

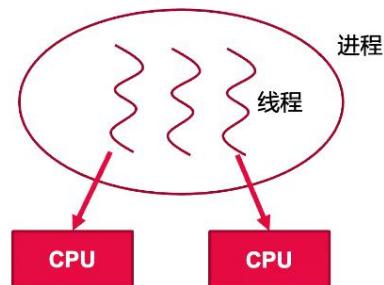
当视频帧在播放前没有渲染好，我们就需要降低清晰度来赶上 DDL。如果我们来不及处理，我们就把速度降低下来，有更多的充足的运算来处理信号。

多核调度策略

之前并没有特地区分多核和单核。

多核调度需要考虑的额外因素

- 一个进程的不同线程可以在不同CPU上同时运行

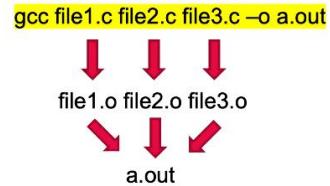


多核调度需要考虑的额外因素

- 同一个进程的线程很可能有依赖关系

- 例：GCC 编译a.out文件
- 每个线程编译一个文件 (.c到.o)
- 线程间依赖：

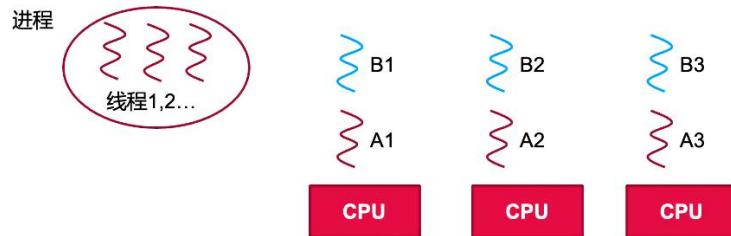
- 所有.o生成完才能进入下一步操作



比如有很多程序在编译，最后合在一起的时候就需要等所有线程运行完。最好一下子分到3个核上再合并。

群组调度：Gang Scheduling

- 在多个CPU上同时执行一个进程的多个线程

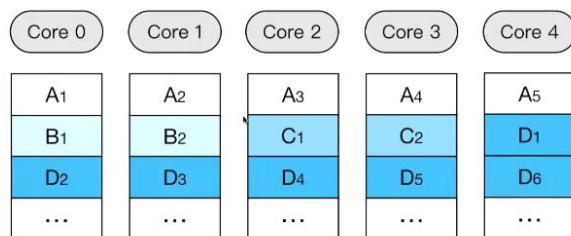


我们一下子全部放上去，再一下子全部撤走。

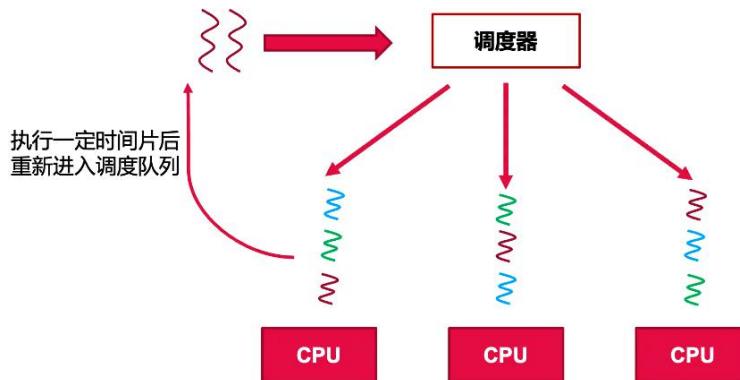
群组调度：Gang Scheduling

- 4组任务A、B、C、D

- 组内任务都是关联任务，需要尽可能同时执行



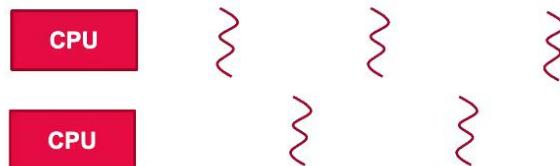
全局使用一个调度器的问题



所以 CPU 竞争全局调度器的时候，就会导致可扩展性变差。

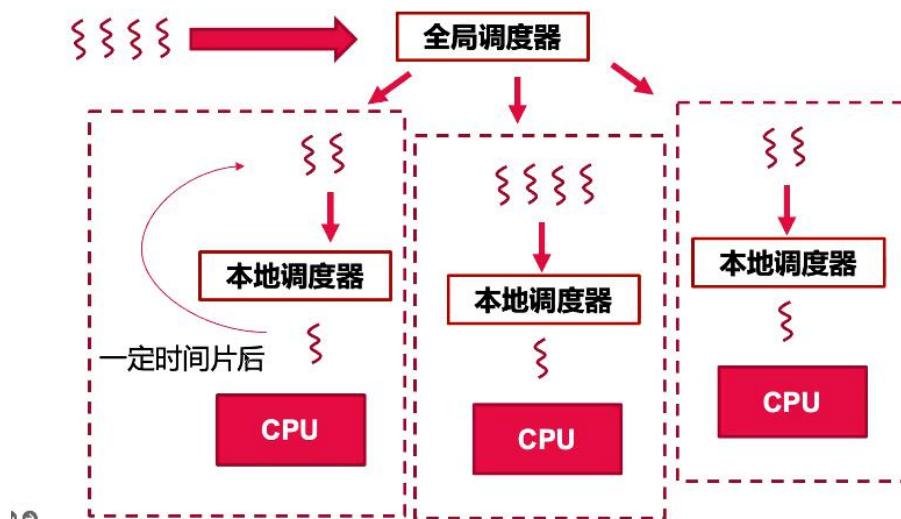
全局使用一个调度器的问题

- 所有CPU竞争全局调度器
- 同一个线程可能在不同CPU上切换
 - 切换开销大：Cache、TLB、...
 - 缓存局部性差



于是我们就提出了 2-level scheduling。

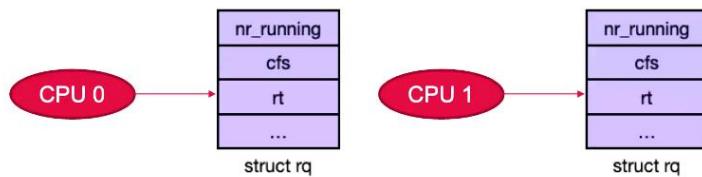
Two-level Scheduling



全局调度器负责把任务分配到本地调度器，然后再有本地调度器去分配运算。这样就可

以避免一个 thread 来来回回地做切换，可以利用好局部性的特性。

ChCore & Linux的多核调度



ChCore & Linux同样使用Two-level Scheduling的架构

每个CPU有各自的本地调度器和runq

这样就可以更适合多核架构。

调度小结

- **策略 V.S. 机制**
- **经典调度**：先到先得、短任务优先、时间片轮转
- **优先级调度**：优先级的选取、优先级反转、优先级继承
- **多级反馈队列**：MLFQ
- **公平共享调度**：彩票调度、步幅调度
- **实时调度**：最早截止时间优先
- **多核调度**：群组调度、两级调度

今天主要我们讲了调度再讲了实时调度和多核调度。

2022/3/15

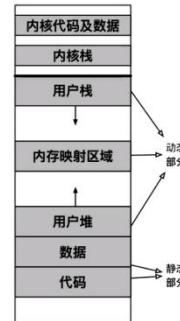
进程间通信

哪怕之前没有接触过 OS，我们也知道了整个系统要做一层虚拟化，让每个应用程序好像独占了这个系统一样。为了减少程序员的不相关的信息的干扰，我们希望产生这样的假象。

回顾: 进程

- **进程是计算机程序运行时的抽象**

- 静态部分: 程序运行需要的代码和数据
- 动态部分: 程序运行期间的**状态**
(程序计数器、堆、栈.....)



- **进程具有独立的虚拟地址空间**

- 每个进程都具有“独占全部内存”的假象
- 内核中同样包含内核栈和内核代码、数据

隔离也不是越多越好的，我们是需要进程间只够通信的。我们可以通过调度的方法和页表做到彼此之间不能访问到对方的地址空间。但是这样问题就出现了。

应用程序的功能非常复杂

- **独立进程: 一个进程就是一个应用**

- 不会去影响其他进程的执行，也不会被其他进程影响

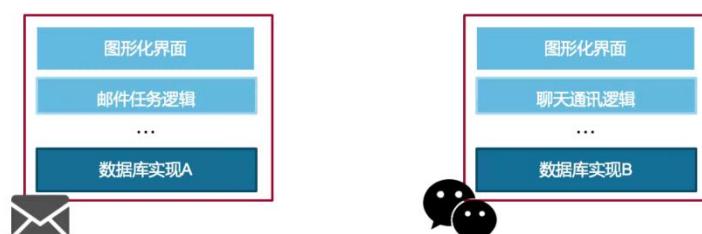


一旦应用程序的功能变多以后，一个进程就是一个应用者会带来复杂化的问题。比如我们要做一个邮件应用，它内部可能有不同的模块（文件模块、目录模块、附件模块、邮件的任务逻辑等），上面还有图形化的界面。一般邮件还会使用数据库的方式去管理邮件，这样后续查找的时候因为有索引，就会查找起来更快一些。在这个情况下，假设还有一个聊天软件。

独立进程的问题-1: 大量重复实现

- **例如：聊天软件和邮件软件都依赖数据库**

- 于是在自己的进程中各自实现一份数据库



这个聊天软件要查找聊天记录和不同的联系人和附件。这个查找的功能也是需要的，所以也要有一个类似数据库的功能。既然邮件和聊天软件都依赖于数据库，如果它们之间是完

全隔离的话，它们需要在自己的进程中实现一份数据库的代码。

独立进程的问题-2: 低效实现

- 例如，邮件软件团队的开发重心在其他组件上

– 所以只能借用低效的某数据库开源实现



但是精力是有限的，所以常用办法是 copy 一个开源数据库。这导致没有办法使用真正高效的数据库。

独立进程的问题-3: 没有信息共享

- 邮件和聊天软件都需要监控系统资源信息

- 没有信息共享

– 即使邮件软件已经完成了计算，聊天软件也要重新计算一遍

第三点就是没有信息共享，因为是完全隔离的，就算邮件软件完成了计算，聊天软件还是要继续计算一遍。

这毕竟是两个应用，不会使用一个进程。

如果不同进程之间可以协作

- 协作进程

– 和独立进程相反，可以影响其他进程执行或者被影响

- 好处

– 模块化：数据库单独在一个进程中，可以被复用
– 加速计算：不同进程专注于特定的计算任务，以获得更好性能
– 信息共享：直接共享已经计算好的数据，避免重复计算



这样我们就可以使用高效的数据库实现对外提供数据库的接口。这样就需要通过一些方法在进程之间形成联系。好处就是模块化，更好复用代码、加速计算，让专业的人把数据库做得更好、不用再重复计算。

OS 在安卓中，就提供了一系列的服务，比如存储密钥的服务和存储文件的服务，每个

人不需要实现自己的数据库。使得整个应用程序的开发逻辑进一步简化。

进程间通信 (IPC)

进程间通信 (IPC)

- **进程协作依赖于进程间通信**
- **进程间通信: 两个(或多个)不同的进程，通过内核或其他共享资源进行通信，来传递控制信息或数据**
 - 交互的双方: 发送者/接收者、客户端/服务端、调用者/被调用者
 - 通信的内容一般叫做“消息”



在隔离的进程之间传递数据。需要通过共享资源来进行通讯。从跨进程的函数调用，可以成为调用者和被调用者。一般通信的内容成为消息，它可以很大，也可以只有 1 个 bit。最常见的机制就是发送者和接受者通过内核来进行 IPC call，如果发送者和接受者共享了内存，那么可以不需要进内核。kernel bypass 在绝大部分场景里都是做得到的。

常见IPC的类型

IPC机制	数据抽象	参与者	方向
管道	字节流	两个进程	单向
共享内存	内存区间	多进程	单向/双向
消息队列	消息	多进程	单向/双向
信号量	计数器	多进程	单向/双向
信号	事件编号	多进程	单向
套接字	数据报文	两个进程	单向/双向

pipe 在 ICS 中就学过了。共享内存可以理解。消息队列是新的抽象。信号量大家已经学过了，当时学的时候是用来做进程间的同步用的，其实协同本身也是信息的传递。在当时我们提信号是因为我们有 shell，shell 作为 fork 出来的进程的父进程，它需要通过信号机制去控制其他的应用，其实 signal 可以用来做别的事情，有自定义的 signal，可以用 signal 去表征任何事情。signal 有一个特点，相对来说传递的量比较少。所以这些 IPC 机制都可以实现进程间通信，但是传递数据的量是不一样的。

两个进程之间互相之间是隔离的，两个机器之间都可以通过 socket 传递数据，我们也可以使用本地 socket 来传递消息。

今天我们更多的是来看一下 IPC 之间不同的机制的对比，以及 OS 的演变的过程。

管道：文件接口的 IPC

IPC 的第一反应就是文件，大家在写 shell 的时候，前后两个 bash 脚本交互很多时候都

是通过文件，`bash` 脚本可以把很多信息输出到文件，下一个 `bash` 就可以从文件中读取做下一步操作。之所以我们可以用文件做进程间通讯，是因为文件是全局的，整个系统共享的。所以文件就是打破进程间隔离的非常好的方法。甚至我们可以通过有没有一个文件来传递信息。

有了这个之后，大家就觉得很方便，但是如果我们要通过文件系统传递一个信息和数据，OS 本身并不知道我们的目的，OS 就会老老实实地把文件更新到文件里去。一旦 `touch` 了这个文件，有了 `inode`，得更新 `inode bitmap`，会写磁盘，但是实际上写磁盘是没有必要的操作，我们只是希望在内存中的两个进程中传递概念。所以如果没有专门用来做 IPC 的文件的话，会很无效。

Unix 管道

- 管道是 Unix 等系统中常见的进程间通信机制
- 管道(Pipe): 两个进程间的一根通信通道
 - 一端向里投递，另一端接收
 - 管道是间接消息传递方式，通过共享一个管道来建立连接
- 例子: 我们常见的命令 `ls | grep`

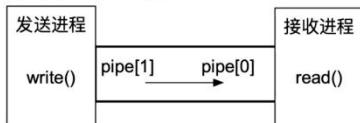
```
→ os-textbook git:(master) ls | grep ipc  
ipc.tex
```

它是间接的消息传递方式。

Unix 管道

- 管道的特点:
 - 单向通信，当缓冲区满时阻塞
 - 一个管道有且只能有两个端口：一个负责输入（发送数据），一个负责输出（接收数据）
 - 数据不带类型，即字节流
 - 基于 Unix 的文件描述符使用

```
int fd[2];
pipe(fd);
fd[0]; // read side
fd[1]; // write side
```



管道是单向的，如果要实现两个进程间互相通信，那么就需要两个管道。一个管道只有两个端口，一个输入一个输出。Unix Pipe 不支持多个输入或多个输出。

数据不带类型，什么叫数据类型呢？在文件这个抽象里，文件就是有名字的一段数据。这个数据是非常底层的一段抽象了，数据本身是没有类型的，就是一堆字节流。什么叫有定义呢？如果我们抽象出一种消息，并且设置了对应的消息类型，有类型之后，我们在另一侧可以做 filter，可以做更复杂的操作。但在这里并没有这么复杂。第四个特点就是基于 unix 的文件描述符来使用。

这里有个核心数据结构。

► Unix 管道

• 管道数据结构

```
连接缓冲区域，定长  
一个连接对应  
最多两个进程  
struct pipe {  
    struct spinlock lock;  
    char data[PIPESIZE];  
    uint nread; // number of bytes read  
    uint nwrite; // number of bytes written  
    int readopen; // read fd is still open  
    int writeopen; // write fd is still open  
};
```

data[PIPESIZE]缓冲区是在内核里的，如果没有缓冲区，一旦我们写了数据对方必须马上收掉。所以这里的缓冲区可以让发送者发完数据以后可以尽快结束，不需要接受者需要同时运行。但这里也有一个问题，PIPESIZE 有多大呢？这对发送者很有好处，但是内核没有那么多内存给我们当缓冲区的 buffer。有了这个 data 作为 buffer 之后，我们就要 nread, nwrite, 读/写了多少字节，用的时候就要% PIPESIZE，这就是一个环形的缓冲。readopen 就是读的 fd 是否开着。如果没有有人 read 的时候写，可能就会导致一些问题。

► Unix 管道

• 管道写操作

```
int  
pipewrite(struct pipe *p, char *addr, int n)  
{  
    int i;  
  
    acquire(&p->lock);  
    for(i = 0; i < n; i++) {  
        while(p->nwrite == p->nread + PIPESIZE) { //  
            if(p->readopen == 0 || proc->killed){  
                release(&p->lock);  
                return -1;  
            }  
            wakeup(&p->nread);  
            sleep(&p->nwrite, &p->lock); // pipewrite-sleep  
            p->data[p->nwrite++ % PIPESIZE] = addr[i];  
        }  
        wakeup(&p->nread); // pipewrite-wakeup  
        release(&p->lock);  
    }  
    return n;  
}
```

上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

当我们写的时候，就根据 while 循环不断做写操作。写进去一点东西以后就要去 wakeup reader。这和 bounded buffer 很类似。当 sleep 返回的时候，锁必须还拿着。然后一旦我们发现有足够的空间，我们就把消息放上去并且更新 nwrite。

Unix 管道

- 管道读操作

```
int
piperead(struct pipe *p, char *addr, int n)
{
    int i;
    acquire(&p->lock);
    检查是否有消息 ----- while(p->nread == p->nwrite && p->writeopen){ // 
        if(proc->empty){
            if(proc->killed){
                release(&p->lock);
                return -1;
            }
        }
        sleep(p->nread, &p->lock); // piperead-sleep
    }
    阻塞等待消息到来 ----- for(i = 0; i < n; i++){ // piperead-copy
        if(p->nread == p->nwrite)
            break;
        addr[i] = p->data[p->nread++ % PIPESIZE];
    }
    wakeup(&p->nwrite); // piperead-wakeup
    release(&p->lock);
    return i;
}
```

一个 reader 在 sleep 的过程中如果有 writer 往里写数据，那么它就会被唤醒。

管道的优点与问题

- 优点: 设计和实现简单

- 针对简单通信场景十分有效

- 问题:

- 缺少消息的类型，接收者需要对消息内容进行解析
 - 缓冲区大小预先分配且固定
 - 只能支持单向通信（为什么？）
 - 只能支持最多两个进程间通信

整个管道的创建就是 shell 帮我们完成的，但是需要接受者主动地对消息做解析。如果我们不关掉对应 fd 的读和写，这意味着一个 writer 写完之后自己读，把数据读回来了，真正 reader 读的时候数据就没了。

命名管道

- 管道缺乏名字，只能在有亲缘关系的进程间使用

- 也称为“匿名管道”
 - 通常通过 fork，在父子进程间传递fd

- “命名管道”则具有文件名

- 在Linux中也称为fifo，可通过mkfifo()来创建
 - 可以在没有亲缘关系的进程之间实现IPC
 - 允许一个写端，多个读端；或多个写端，一个读端

我们在默认状态下，认为管道就是匿名管道。由于它缺乏名字，如果我们加一条竖杠的话，那我们的竖杠只能在 shell 创建的两个进程之间创建管道。必须要有亲缘关系才能方便地传递 fd。在 Linux 中，可以通过 mkfifo 创建命名管道，这样我们就可以在没有亲缘关系的

进程之间实现 IPC。命名管道的多个端会让事情更复杂，如果两个读者同时去读，就会有问题。在 POSIX 规范中，定义了每次读至少有 buffer size 大的数据，所以我们不会看到两个读者一个个 byte 交叉去读。

共享内存（内存接口的 IPC）

既然我们可以通过文件接口实现进程间的共享，下一个问题就是为什么我们不用一个更加底层的接口：内存呢？OS 的功能在实现的时候，当我们想去实现一个新功能的时候，我们的反应是能不能把这个新功能包装成文件的样子，程序员会觉得更加简单，如果可以提供文件的话，那么我们为什么不通过内存这个接口。

`mmap` 是把文件接口变成内存接口，就是实现了一个接口的转换，而 OS 所提供的无非就是抽象。在 IPC 的时候，我们是否能让两个进程通过内存的方式来实现 IPC 呢？

共享内存

- 小区门口的桌子上允许临时放置快递
- 快递员在桌上放置快递，小明从桌上取快递



共享的快递桌

快递桌就是一个很好的 sender 和 receiver 的例子。有了这个内存区域之后，一个进程在里面写什么，另一个进程就可以立马读到。并且这是一个双向的过程。但是一个挑战就是我们如何做到数据的同步，不能覆盖已有的数据。

共享内存

- 基础实现: 共享区域

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

共享数据区域，容量为10
共享状态

我们再次把这个抽象成了 buffer。当没有空间的时候就去做 busy wait。

共享内存

共享内存

• 基础实现: 发送者

```
while (new_package) {
    /* Produce an item */
    while (((in + 1) % BUFFER_SIZE)
           == out)
        ; /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

当没有空间时，发送者盲等
发送者放置消息

• 基础实现: 接收者

```
while (wait_package) {
    while (in == out)
        ; // do nothing -- nothing to consume
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

当没有新消息时，接收者盲目等待
接收者获取消息

接受者就是不断去读数据，一旦有了就去++。这和 pipe 的机制是很像的，但是在 pipe 中我们使用了 sleep 机制，来避免盲等。为什么共享内存要这样呢？有没有办法共享内存也使用 sleep 机制呢？

共享内存的问题

• 缺少通知机制

- 若轮询检查，则导致CPU资源浪费
- 若周期性检查，则可能导致较长的等待时延
- **根本原因**：共享内存的抽象过于底层；缺少OS更多支持

• TOCTTOU (Time-of-check to Time-of-use) 问题

- 当接收者直接用共享内存上的数据时，可能存在被发送者恶意篡改的情况（发生在接收者检查完数据之后，使用数据之前）
- 这可能导致buffer overflow等问题

在这种机制下是缺乏一种通知机制的，一个进程写内存，另一个进程是没办法得知的。写内存就是一条 store 指令，OS 不知道，那么没有人可以通知到另一个进程，除非硬件去通知。这就是共享内存的抽象过于底层，且不需要 OS 的支持，OS 唯一要做的事情就是把同一个 memory 映射给两个进程。

而在前面的 pipe 的过程中，每次读写都要进入 syscall，这样 OS 就知道什么时候在写，就可以通过 sleep 和 wake up 机制。

我们得通过轮询来解决这个问题，显然不是足够的好。

然后还有一个问题是 TOCTTOU 问题，当接受者收到内存上的数据去用的时候，它很有可能去检查一下，当我们检查 size 的时候，size 是对的，但是当我们读的时候，可能 size 就被改掉了，很有可能出现 buffer overflow。所以我们最好把内存 copy 到自己空间再去读。

消息传递 (Message Passing)

内存既然没有通知机制，我们能不能加上通知数据，比如在共享内存写完数据后，通过发一个信号来通知另一个进程，另一个进程运行 signal handler 去做这件事情，实际上这个事情就很像 DMA 做的事情。这当然是可以的，依赖的机制比较多，且机制的组合往往不是最简单的。

那么在看过文件接口和内存接口之后，我们就会产生一个问题。毕竟文件和内存都不是用来专门做 IPC 的，那么 IPC 是否足够重要到让 OS 提供一套新的功能。

消息传递

- **消息系统**

- 通过中间层(如内核)保证通信时延，仍可以利用共享内存传递数据



好处: 1) 低时延 (消息立即转发)

2) 不浪费计算资源

我们让内核提供一个新的接口来实现消息的传递。基本的接口就是 send 和 receive。

消息传递

- **基本操作:**

- 发送消息Send(message)
- 接收消息Recv(message)

- **如果两个进程P和Q希望通过消息传递进行通信，需要:**

- 建立一个通信连接
- 通过Send/Recv接口进行消息传递

两个进程要想通信的话，先要建立起信道，再通过 send 和 receive 去发送 message。

直接通信

- **直接通信**

- 进程拥有一个唯一标识
- Send(P, message): 给P进程发送一个消息
- Recv(Q, message): 从Q进程接收一个消息

- **直接通信下的连接**

- 连接的建立是自动的 (通过标识)
- 一个连接唯一地对应一对进程
- 一对进程之间也只会存在一个连接
- 连接可以是单向的，但是在大部分情况下是双向的

如果我们知道 PID，就可以通过直接通信去发送 message。这样建立起通信以后就可以通过 Send(P)和 Receive(Q)来实现。

直接通信

- 发送者

```
while (new_package) {  
    /* Produce an item */  
    while ((in + 1) % BUFFER_SIZE)  
        == out)  
        ; /* nothing, no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    Send(XiaoMing, "Package");  
}
```

- 接收者

```
while (wait_package) {  
    Recv(Expressman, Msg);  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    return item;  
}
```

Recv会阻塞，直到Send发送消息过来

调用 Recv 的时候，就会阻塞在那里，可以使用 sleep 机制实现。Recv 一旦没有就 block 住，有了以后就直接通信。

快递员有好多苦恼 (1)



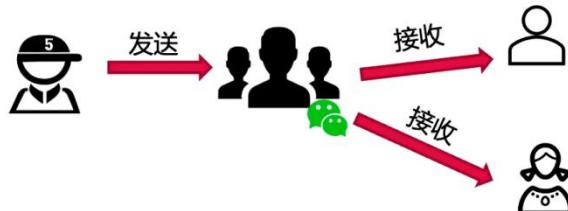
- 快递员执行Send的时候，小明还没有Recv
- 快递员知道小明妈妈经常在家，希望建立一个聊天群，在群里发布快递信息
- 小明不接电话时可以拜托妈妈下来拿快递

执行 Send 的时候还没有 Recv，如果没有 buffer 的话，快递员必须等到家里有人的时候上门给你。所以我们需要一个聊天群，不一定是让收货的人拿到。聊天群就是一个间接的发送。

间接通信：用聊天群发布快递信息

- 消息的发送和接收需要经过一个“信箱”

- 聊天群 (所有在群内的人都可以接收消息)
- 每个“信箱”有自己唯一的标识符 (这里的群号)
- 发送者往“信箱”发送消息，接收者从“信箱”读取消息



我们建立一个新抽象“信箱”，每一个信箱有自己的标识符。每个进程往信箱去发，而接收者从信箱里去读。

间接通信

- **间接通信下的连接**
 - 进程间连接的建立发生在共享一个信箱时
 - 每对进程可以有多个连接 (共享多个信箱)
 - 连接同样可以是单向或双向的
- **间接进程间通信的操作**
 - 创建一个新的信箱
 - 通过信箱发送和接收消息
 - 销毁一个信箱
- **原语**
 - Send(M, message): 给信箱M发送一个消息
 - Recv(M, message): 从信箱M接收一个消息

这样 P 也不需要知道 Q 是谁，只需要往信箱里发就可以了。

间接通信：用聊天群发布快递信息

• 发送者(快递员)

```
while (new_package) {  
    /* Produce an item */  
    while ((in + 1) % BUFFER_SIZE)  
        == out)  
        ; /* nothing, no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    Send(Mouse, "Package");  
}
```

• 接收者(小明)

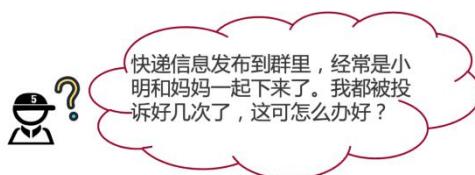
Recv(Mouse, Msg);

• 接收者(小明妈妈)

Recv(Mouse, Msg);

小明或者妈妈任何一个人只要上聊天群，就能看到快递信息

快递员有好多苦恼 (2)



• 怎么解决“信箱”共享带来的多接收者的问题呢？

但是每次一发送，两个人都收到了下楼取快递了。于是就产生了当信箱被多个读者共享的时候，有可能被多个读者同时读到，我们需要保证让一个人读到最新的信息。

信箱共享的挑战

• 信箱的共享问题

- 进程P₁、P₂和P₃共享一个信箱M；P₁负责发送，P₂、P₃负责接收
- 当一个消息发出的时候，谁会接收到最新的消息呢？

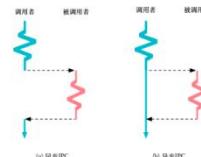
• 可能的解决方案

1. 让一个连接(信箱)只能被最多两个进程共享，避免该问题
2. 同一时间，只允许最多一个进程在执行接收信息的操作（如加锁）
3. 让消息系统（如信箱）任意选择一个接收者
 - 可以设置不同的策略：随机、Round-Robin等
 - 选择后，需要通知发送者谁是最终接收者（如signal）

1. 限制只有1个读者和写者，回到了匿名pipe的情况。
2. 在同一个时间内只允许一个进程接收。只有一个人拿到，拿到了以后另一个人就拿不到这个信息。
3. 依赖于一些策略，随机把信息给到一个人。

消息传递的同步与异步

• 消息的传递可以是阻塞的或非阻塞的



• 阻塞通常被认为是同步通信

- 阻塞的发送/接收：发送者/接收者处于阻塞状态，直到消息发出/到来
- 同步通信通常有着更低时延和易用的编程模型（通常来说）

• 非阻塞通常被认为是异步通信

- 发送者/接收者不等待操作结果，直接返回
- 异步通信的带宽一般更高

消息的传递既可以是阻塞的（同步的）也可以是非阻塞的（异步的）。我们之前讲到中断的时候也有类似的表述。如果说同步的话，第二个人就只能等着了，异步的话通常带宽和效率都会高一些。

超时机制

• 为了好评，快递员选择：



- 尽可能等待（同步的通信）
- 但是一旦超过一个值（如15分钟），就先带走快递，等下再配送

• 超时机制的引入

- Send(A, message, Time-out)
 - 超过Time-out限定的时间就返回错误信息
- 两个特殊的超时选项：①一直等待（阻塞）；②不等待（非阻塞）
- 避免由通信造成的拒接服务攻击等

一旦消息等待时间超过一个阈值没有接收者取走，就会有永远等待和不等待的情况。有了超时机制之后，send和recv会增加一个timeout。

同步通信和超时机制

- **发送者(快递员)**

```
while (new_package) {  
    /* Produce an item */  
    while (((in = (in + 1) % BUFFER_SIZE)  
           == out)  
           ; /* nothing, no free buffers */  
           buffer[in] = item;  
           in = (in + 1) % BUFFER_SIZE;  
           if (Send(Mailbox, "Package",  
                  "15min") == error)  
               goto retry;  
       }  
   }
```

快递员决定等待最多15分钟，一旦超时就放弃这一次派送

- **接收者(小明)**

```
Recv(Mailbox, Msg, Time-out);
```

- **接收者(小明妈妈)**

```
Recv(Mailbox, Msg, Time-out);
```

快递员有好多苦恼 (4)



新冠疫情缓解后，为了避免快递丢失，小区保安不允许快递员将快递放在快递桌上。快递员不能放下快递就走了。

- **有手机后，快递桌的作用有什么变化吗？**

其实没有快递桌也是可以做到的，缓冲并不是一个必须的。

通信连接的缓冲：快递桌的作用

- **缓冲：通信连接可以选择保留住还没有处理的消息**

- **常见的三种设计**

- 零容量：通信连接本身不缓冲消息，发送者只能阻塞等待接收者接收消息（保安不提供快递桌）——微内核
- 有限容量：连接可以缓冲最多N个消息，当缓冲区满之后发送者只能阻塞等待（快递只能放在桌上，但是空间有限）
- 无限容量：连接可以缓冲系统资源允许下的任意数量的消息，发送者几乎不需要等待

发送者可以直接送到接收者的手里，还有有限的 buffer 的情况，还有无限容量的快递桌的情况。如果 OS 有足够的把握接收地足够快，它就可以提供无限容量的假象。有缓存和无缓存比有什么区别呢？

有缓存和没缓存的区别

- **有缓存（相比没缓存）的优点：**

- 可以支持多个并发消息
- 可以支持非阻塞（异步的通信），发送者不需要等待消息传递到接收者就可以返回
- 接收者可以批量处理多个消息，提高吞吐率

- **有缓存（相比没缓存）的缺点：**

- 占用更多的系统资源（通常是内存资源）
- 系统软件（如内核）很难确定应该预留多少缓存

有缓存占用更多的系统资源，而系统软件是不知道应该使用多少缓存的。

共享内存和消息传递的对比

- **共享内存的优势：理论上可以实现零内存拷贝的数据传输**
 - 消息传递的方式通常：发送者用户态内存拷贝到内核内存，再拷贝到接收者用户态内存（两次拷贝）
- **消息传递的优势：**
 - 抽象更简单，并且能够较好支持变长的消息（共享内存往往需要用户态软件封装来实现这一点）
 - 不需要轮询，通常包含内核支持的通知机制
 - 安全性保证通常更强（有内核等保证），并且不会破坏发送者和接收者进程的内存隔离性

消息队列

在内核中用消息队列的机制来实现消息传递。

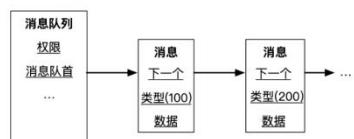
消息队列

- **一种消息传递机制**
- **设计选择：**
 - 间接通信方式，信箱为内核中维护的消息队列结构体
 - 有（有限的）缓存
 - 没有超时机制
 - 支持多个（大于2）的参与者进行通信
 - 通常是非阻塞的（不考虑如内核缓存区满等异常情况）

消息队列：带类型的消息传递

Ftok();
Msgget();
Msgsnd();
Msgrcv();
Msgctl();

- **消息队列：以链表的方式组织消息**
 - 任何有权限的进程都可以访问队列，写入或者读取
 - 支持异步通信（非阻塞）
- **消息的格式：类型 + 数据**
 - 类型：由一个整型表示，具体的意义由用户决定
- **消息队列是间接消息传递方式**
 - 通过共享一个队列来建立连接



消息就是类型 + 数据，类型就是用户定义的一个整数。有了类型之后，内核里就可以对类型的新的接口的支撑，比如对类型的过滤。

消息队列的例子

发送者	接收者
<pre>key = ftok("./msgque", 11); msgid = msgget(key, 0666 IPC_CREAT); message.mesg_type = 1; msgsnd(msgid, &message, sizeof(message), 0);</pre>	<pre>key = ftok("./msgque", 11); msgid = msgget(key, 0666 IPC_CREAT); msgrcv(msgid, &message, sizeof(message), 1, 0); msgctl(msgid, IPC_RMID, NULL);</pre>

只要发送者和接受者都通过./msgque 就可以得到这个 key 去通信。需要通过文件系统的权限保护机制去做隔离，得到 key 之后就可以设置 mesg_type，并且发送数据。

消息队列：带类型的消息传递

- **消息队列的组织**

- 基本遵循FIFO (First-In-First-Out)先进先出原则
- 消息队列的写入：增加在队列尾部
- 消息队列的读取：默认从队首获取消息

- **允许按照类型查询: Recv(A, type, message)**

- 类型为0时返回第一个消息 (FIFO)
- 类型有值时按照类型查询消息
 - 如type为正数，则返回第一个类型为type的消息

由于消息是带类型的，所以我们可以按照类型查询。又可以把 message queue 有新的做法。

消息队列 VS. 管道

- **缓存区设计:**

- 消息队列：链表的组织方式，动态分配资源，可以设置很大的上限
- 管道：固定的缓冲区间，分配过大资源容易造成浪费

- **消息格式:**

- 消息队列：带类型的数据
- 管道：数据（字节流）

- **连接上的通信进程:**

- 消息队列：可以有多个发送者和接收者
- 管道：两个端口，最多对应两个进程

消息队列更加灵活易用，但是实现也更加复杂

- **消息的管理:**

- 消息队列：FIFO + 基于类型的查询
- 管道：FIFO

首先它们在缓冲区设置方面不太一样，消息队列的缓冲区的分配是动态的，可以设置比较大的上限。还有消息队列支持基于类型的查询。所以消息队列实现的能力更强。

轻量级远程方法调用（LRPC）

前面提到的所有的 IPC 机制，无论是共享内存还是 message send/recv。都会带来性能的损失，性能损失最大的问题是因为我们要在两个进程之间做切换，开销是很大的，并且我们还需要有通知机制，还要去传输数据，都是性能的额外开销。

IPC通常会带来较大的性能损失

- **传统的进程间通信机制通常会结合以下机制：**

- 通知：告诉目标进程事件的发生
- 调度：修改进程的运行状态以及系统的调度队列
- 传输：传输一个消息的数据过去

- **缺少一个轻量的远程调用机制**

- 客户端进程切换到服务端进程，执行特定的函数 (Handler)
- 参数的传递和结果的返回

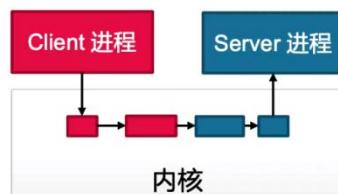
我们缺乏一个轻量级的远程函数调用的机制，客户端切换到服务端进程做一个函数再返回。我们并没有放弃掉进程的抽象，正在这个限制之下去做这个事情。

轻量级远程调用 (LRPC)

- **Lightweight Remote Procedure Call (LRPC)**

- **解决两个主要问题**

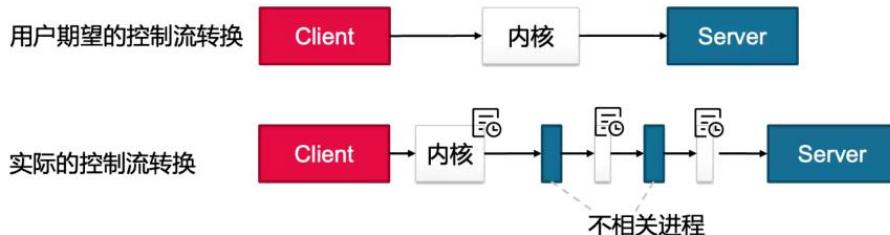
- 控制流转换: Client 进程快速通知 Server 进程
- 数据传输: 将栈和寄存器参数传递给 Server 进程



控制流转换就是如何让一个客户端进程快速通知服务端进程。最好上一条指令是 Client 的最后一条指令而下一条指令就是 Server 的第一条指令。并且我们还要传递相应的参数。

控制流转换: 调度导致不确定时延

- 控制流转换需要下陷到内核
- 内核系统为了保证公平等，会在内核中根据情况进行调度
 - Client和Server之间可能会执行多个不相关进程



Client 掉入到内核以后，没有办法和内核直接说调度到 Server 上。问题在于内核有自己的调度算法，并且发现 Client Block 了，它很有可能在 Client 调用了 message send 之后，内核很有可能就根据自己的调度算法调度别的不相关的进程，运行了很久再运行到了 Server，此时 Client 已经等了很久了。

我们要去分析一下内核在什么情况下可以听 Client 的话。如果 client 要表现出自己的诚意，如果有一种机制可以把自己的时间片分给 server，那么内核也没有理由把时间片给别人。现有的机制不支持时间片的共享，每个进程有自己的 PCB，PCB 里会记录下当前有自己时间片等信息。

此时我们就需要把时间片信息和 Client 作为进程的信息解耦开，最终我们希望做到时间片信息不切换，但是上面的信息要切换。也就是此时 client 运行的代码和数据在 server 端，所以我们要切换 ttbr 和页表，但是时间片我们不希望去切。

迁移线程: 将Client运行在Server的上下文

- 为什么需要做控制流转换?
 - 使用Server的代码和数据
 - 使用Server的权限 (如访问某些系统资源)
- 只切换地址空间、权限表等状态，不做调度和线程切换



时间片等调度方面的信息我们不切换，但是地址空间和权限之类的都需要切换。这样从内核角度，就像是一个 `thread` 一直在运行，本来用了 `client` 的地址空间，运行到一半突然换成了 `server` 的地址信息。换句话说，`thread` 不变，但是地址空间变了。

现在我们有了一个 `thread` 和多个地址空间。

数据传输：数据拷贝的性能损失

- 大部分 Unix 类系统，经过内核的传输有(至少)两次拷贝
 - Client → 内核 → Server
- 数据拷贝：
 - 慢：拷贝本身的性能就不快 (内存指令)
 - 不可扩展：数据量增大10x，时延增大10x



大部分 IPC，我们需要传递两边内存，如果使用共享内存就可以减少一些。

共享参数栈和寄存器

- 参数栈 (Argument stack，简称A-stack)
 - 系统内核为每一对 LRPC 连接预先分配好一个 A-stack
 - A-stack 被同时映射在 Client 进程和 Server 进程地址空间
 - Client 进程只需要将参数准备到 A-stack 即可
 - 不需要内核额外拷贝
- 执行栈 (Execution stack，简称E-stack)
- 共享寄存器
 - 普通的上下文切换：保存当前寄存器状态 → 恢复切换到的进程寄存器状态
 - LRPC 迁移进程：直接使用当前的通用寄存器
 - 类似函数调用中用寄存器传递参数

我们提出了参数栈和执行栈的概念。一个 thread 多个地址空间的时候，我们切换地址空间的一瞬间，我们的栈是谁呢？我们不能是 client 的栈，肯定得用 server 地址空间里的栈。但是 server 哪来的栈给我们用呢？所以 client 在进入到 server 的时候，还需要贡献一个内存页来作为栈，这个还是在 server 内的地址空间。所以通常 client 会提前和 server 说，等一会儿会切换到你，提前分配一个内存页保留。同样地，还会分配一个参数栈，同时映射在 client 和 server 的进程地址空间中，这样就可以实现参数传递了。

所以我们就有两个栈，LRPC 直接使用当前的寄存器，可以通过寄存器的方式传递，放不下就通过 A-stack 的方式去传递。

轻量远程调用：通信连接建立

- Server 进程通过内核注册一个服务描述符
 - 对应 Server 进程内部的一个处理函数(Handler)
- 内核为服务描述符预先分配好参数栈
- 内核为服务描述符分配好调用记录 (Linkage record)
 - 用于从 Server 进程处返回 (类似栈)
- 内核将参数栈交给 Client 进程，作为一个绑定成功的标志
 - 在通信过程中，通过检查 A-stack 来判断 Client 是否正确发起通信

整个过程，之前还是要做一些基础性的事情。内核要为服务描述符分配好参数栈和

Linkage Record。它是用来服务 server 返回的。Return 的时候，要发生一次进程切换的 client，内核总得有个地方记录是从哪个 client 调用过来的。Linkage Record 相当于是一个跨线程的 stack 记录谁调用了谁，使用了软件的方式实现了跨进程的 return。然后内核将参数栈交给 client，A-stack 可以判断 Client 是否正确发起通信。

轻量远程调用：基本用户接口



server 先注册，client 再连接、准备参数、IPC call。然后注册好了 handler 以后就可以调用到函数，从 A-stack 来获取参数，E-stack 的数量就觉得了有多少人可以同时调用 LRPC。如果 E-stack 不够了，那么就会阻塞在找 E-stack 这一步等待资源释放。

轻量远程调用：一次调用过程

1. 内核验证绑定对象的正确性，并找到正确的服务描述符
2. 内核验证参数栈和连接记录
3. 检查是否有并发调用 (可能导致A-stack等异常)
4. 将Client的返回地址和栈指针放到连接记录中
5. 将连接记录放到线程控制结构体中的栈上 (支持嵌套LRPC调用)
6. 找到Server进程的E-stack (执行代码所使用的栈)
7. 将当前线程的栈指针设置为Server进程的运行栈地址
8. 将地址空间切换到Server进程中
9. 执行Server地址空间中的处理函数

注意线程还是 client 的指针，只是 client 的栈指针指向 server 的 E-stack，切换到 server 的地址空间，并且把 PC 指向 server。

轻量远程调用：通信调用实现

- 内核中的代码

```
ipc_call(A-stack, arg0, .. arg7):
    verify_binding(A-stack); //验证A-stack正确性
    service_descriptor = get_desc_from_A(A-stack);
    /*其他安全检查: 是否存在并发调用 ? */

    ...
    save_ctx_to_linkage_record(); //保存调用信息到连接记录上
    save_linkage_record();
    ...

    /* 切换运行状态 */
    switch_PT(); //修改页表
    switch_cap_table(); //修改权限表
    switch_sp(); //修改栈地址
    ...

    //回到用户态(服务端进程), 不修改参数寄存器
    ctx_restore_with_args (ret);
```

server 因为可能要访问一些文件，所以要切换到自己的 capability table（权限表）。这些文件可能它自己才有权限去访问。

轻量远程调用：讨论

- 为什么需要将栈分成参数栈和运行栈？
- LRPC中控制流转换的主要开销是什么？
- 在不考虑多线程的情况下，共享参数栈是否安全？

参数栈就是数据，运行栈就是有 return address 的。如果恶意的 client 可以控制运行栈的话，它就可以控制控制流做攻击。LRPC 还是要进内存，要换页表，势必带来一些开销。如果没有多线程的话，共享参数栈还是比较安全的，这件事情本身一旦当 client 调用 server 的时候，server 运行的时候 client 是不会运行的，那么没有人会去改 A-stack。

ChCore 进程通信

ChCore进程间通信

- 通信进程直接切换
 - 启发自LRPC和L4直接切换技术
- 同步的通信
- 通过共享内存传输大数据
- 基于Capability的权限控制
 - 类似Unix文件描述符的权限机制，Capability表示一个线程/进程对于系统资源的具体权限

ChCore 是一个微内核，它结合了 LRPC 机制，使用内存共享方式来做数据传输。

建立通信连接

1. 服务端进程在内核中注册服务
2. 客户端进程向内核申请连接目标服务端进程的服务
 - 可选: 设置共享内存
3. 内核将客户端请求请求转发给服务端
4. 服务端告诉内核同意连接 (或拒绝)
 - 可选: 设置共享内存
5. 内核建立连接，并把连接的Capability返回给客户端
 - 或返回拒绝

建立通讯的过程就是服务端申请，client 发请求给 server。服务端告诉内核同意，并且把连接的 capability 返回给客户端。

通信过程 (发起通信)

1. 客户端进程通过连接的Capability发起进程间通信请求
2. 内核检查权限，若通过则继续步骤3，否则返回错误
3. 内核直接切换到服务端进程执行 (不经过调度器)
 - 将通信请求的参数设置给服务端进程的寄存器中
4. 服务端处理完毕后，通过与步骤3相反的过程将返回值传回客户端

内核不经过调度器，直接切换到服务端进行执行。这是一个简化的步骤。return 的时候就反过来切换回来。

实现: 用户态通信调用

```
void ipc_dispatcher(ipc_msg_t *ipc_msg) {
    u64 ret;
    char* data = ipc_get_msg_data(ipc_msg);

    // handling ipc accordingly...

    ipc_return(ret);
}

void server() {
    // ...
    ipc_register_server(ipc_dispatcher);
    // ...
}

void client() {
    int server_thread_gap;
    ipc_struct_t client_ipc_struct;
    ipc_msg_t *ipc_msg;

    // get server_thread_gap ①
    ipc_register_client(server_thread_gap, &client_ipc_struct);

    u64 ret = ipc_call(&client_ipc_struct, ipc_msg);
    // ...
}
```

The diagram illustrates the communication flow between a client and a server. It shows two code snippets: one for the client and one for the server. The client code contains a call to `ipc_call` with a pointer to the `client_ipc_struct` and an `ipc_msg`. The server code contains a call to `ipc_dispatcher` with a pointer to the `ipc_msg`. A red arrow points from the `client_ipc_struct` in the client code to the `client_ipc_struct` in the server code. Another red arrow points from the `ipc_dispatcher` function back to the `client_ipc_struct` in the client code. A red circle labeled ① is placed over the `server_thread_gap` variable in the client code. A red circle labeled ② is placed over the `ret` variable in the client code.

实现: 通信系统调用

• IPC_call

```
u64 sys_ipc_call(u32 conn_cap, ipc_msg_t *ipc_msg)
{
    struct ipc_connection *conn = NULL;
    u64 arg;
    int r;

    conn = obj_get(current_thread->process, conn_cap, TYPE_CONNECTION);

    /* Message Processing*/
    // do some thing...

    arg = get_srv_vmaaddr(ipc_msg);
    thread_migrate_to_server(conn, arg);

    BUG("This function should never\n");
    return 0;
}
```

获取内核的连接对象，检查cap权限

在传递之前需要将
ipc_msg在客户进程的
虚拟地址转换到服务
进程中的虚拟地址

将控制流移交给服务进程

实现: 通信系统调用

• IPC_return

```
void sys_ipc_return(u64 ret) {
    struct ipc_connection *conn = current_thread->active_conn;

    thread_migrate_to_client(conn, ret);

    BUG("This function should never\n");
}
```

将控制流移交给客户进程

今天我们讲了不同接口的 IPC 的实现，以及一个线程是如何在不同地址空间里穿梭的。

2022/3/17

同步原语

那我们今天课程内容是同步原语，这是相对有比较挑战的一个话题。

回顾：进程间通讯

- 进程间通信: 两个(或多个)不同的进程，通过内核或其他共享资源进行通信，来传递控制信息或数据
 - 直接通讯/间接通讯
- 进程间协作：基于消息传递的抽象



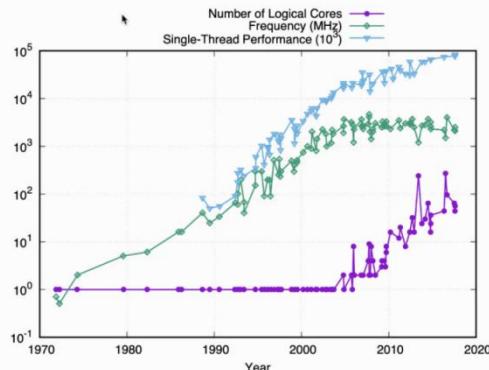
今天的主题：直接基于**共享内存**操作（如发送者直接修改全局变量）

我们上节课主要介绍了一些进程间的通信，在不同进程之间通过内核或者共享资源进行

通信来传递信息和数据，包括直接通信和简介通信。也可以在宏内核和微内核中可见。今天我们基于共享内存的操作，比如发送者可以直接修改全局变量。

多处理器与多核

- 单核性能提升遇到瓶颈
- 不能通过一味提升频率来获得更好的性能
- 通过增加CPU核数来提升软件的性能
- 桌面/移动平台均向多核迈进



为什么我们现在使用多线程的方式，主要是现在多核成为了主流。大家一直在提进入了后摩尔时代，也就是原来晶体管数量的增加体现在 CPU 主频的提升，这样我们的软件可以不通过修改得到这个红利，但是现在工艺到达了瓶颈，包括功耗的约束，比如 32nm 到 16nm，哪怕晶体管在相同的面积里，晶体管的数量会变为 4 倍，会有一些物理上的约束从而使得单位面积晶体管功耗在增加。

现在我们手机中还有动态调频的大中小核的多核处理器的情况。

多核不是免费的午餐



网图：多核的真相

假设现在需要建房子：

- 工作量 = 1000人年
- 工头找了10万人，需要多久？

面临的两个问题：

1. 工人多手杂，不听指挥，导致施工事故（**正确性问题**）
2. 工具有限，大部分工人无事可干（**性能可扩展性问题**）

多核带来了一个很大的问题就是软件的性能提升可以指望芯片工艺的进步，但是现在就没有这样一个免费的午餐，我们需要把软件的代码进行更好的并行从而才能获得更好的性能。这是往上的一个 SOC，虽然有 10 个核，但是它的性能并不好。“一人干活，多人围观”，大量的核并没有进入工作。一方面是 SOC 的设计，还有就是 SOC 上的 OS 和软件是否能更好地支持并行。

Q: 是否 10 万人做 1000 人年的事情只需要 0.001 年呢？

A: 显然不是，人多不一定听指挥，可能会导致正确性问题，第二就是我们的工具是有限的，大部分工人可以没有事情可以做。

人月神话这本书，业界习惯用人月评估进度，但是这件事情并不是可以直接累加的。

操作系统在多处理器多核环境下面临的问题

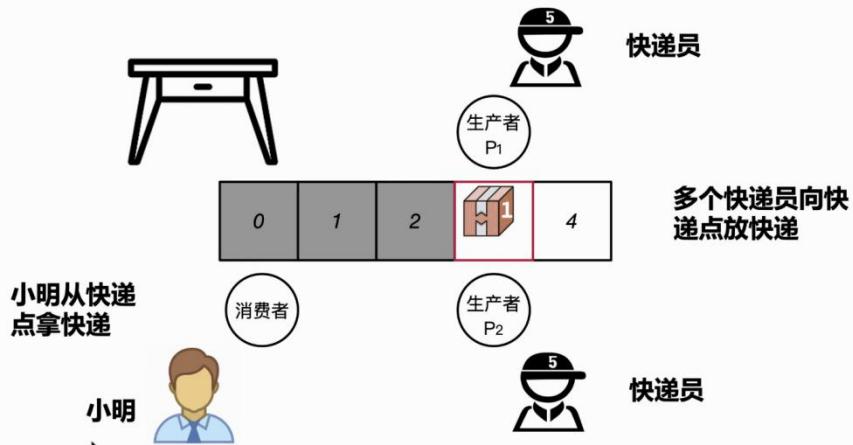
正确性保证	性能保证
<ul style="list-style-type: none">对共享资源的竞争导致错误操作系统提供同步原语供开发者使用使用同步原语带来新的问题	<ul style="list-style-type: none">多核多处理器硬件与特性可扩展性问题导致性能断崖系统软件设计如何利用硬件特性

今天我们主要介绍正确性的保证，我们如何构建出能够正确性同步的原语。这样的正确性主要是说，对共享资源的竞争需要避免这种错误。使用同步原语的话也会带来一些新的问题。有些实现正确性是可以保证的，但是性能可能有问题。

生产者消费者问题、竞争条件、临界区问题

我们还是用一个经典的问题来引入。两年前新冠疫情刚刚爆发的时候，那时候我们只能上网课，这导致了一个快递的问题。这就是一个典型的生产者-消费者问题，就会带来一些临界区和竞争条件的问题。

生产者消费者问题



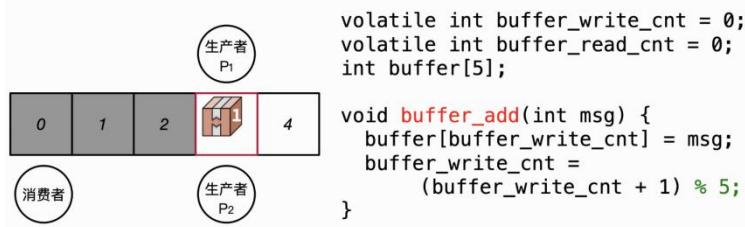
如果有快递，一般情况下会放一系列的快递桌/快递柜。假设桌子上只能放 5 个快递，那么这样的话就是有多个生产者和多个消费者，那么怎么样来达到一个同步呢？

比如我们的小明要来拿快递，它是消费者。有两个快递员，比如京东和天猫可以放快递。那么我们怎么样来使得取快递：

1. 正确
2. 不能让快递员一直等在这里，比如我们桌子满了以后，我们不能让快递员等空出位子了再把快递放上去。

首先我们来看一个最简单的版本的实现：

生产者放入有限缓冲区

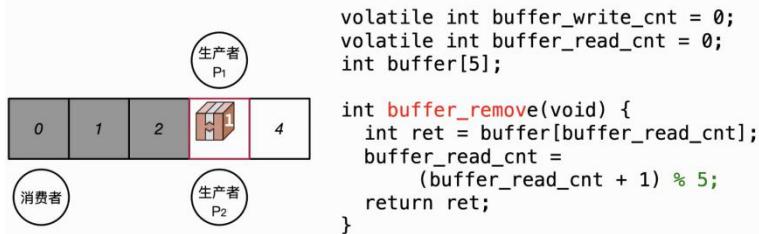


此时 `buffer_write_cnt = 3`

我们可以定义几个变量：

`write_cnt` 就是给生产者来使用的，我们可以往里面放快递，我们把放快递抽象成一个函数 `buffer_add`，是在当前的 `buffer` 里放入 `msg`，并且我们把 `buffer_write_cnt + 1` 再 `mod 5`。

消费者从有限缓冲区拿取



此时 `buffer_read_cnt = 1`

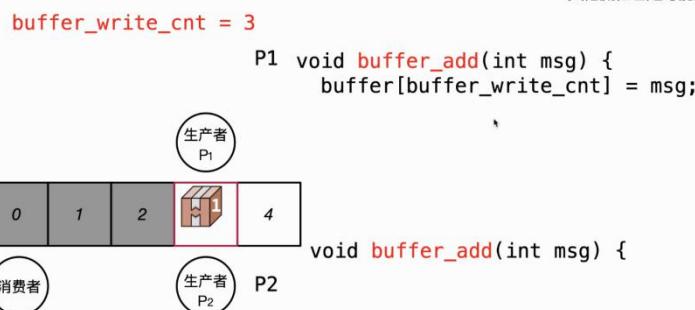
这个也是从 `buffer` 中取 `read_cnt` 处的快递。

这个代码有什么问题？我们来看如下的一个问题：

生产者消费者问题

考虑这样一个执行流程

*其他流程也是可能的

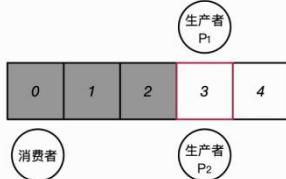


两个快递员同时在放快递的时候会出现什么问题？如果第一个快递员 `P1` 往上放了一个快递，且 `P2` 也往 3 上放了一个快递。这样一个 slot 上放了 2 个快递，这样一定有一个快递会被撞到地上。这样就产生了一个问题，也就是竞争条件。我们需要确保它们不能会新数据放到同一块缓冲区里。

竞争条件

竞争条件 Race Condition

如何确保他们不会将新产生的数据放入到同一个缓冲区中，造成数据覆盖？



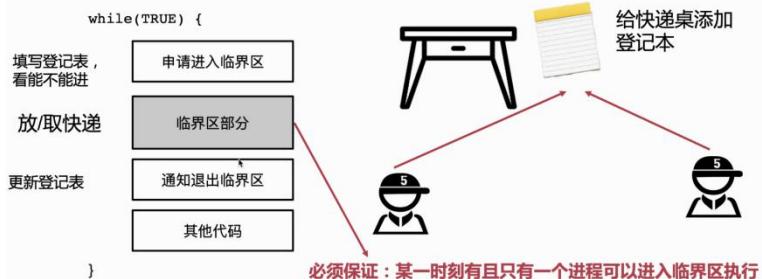
此时产生了竞争条件（竞争冒险、竞态条件）：

- 当多个进程同时对共享的数据进行操作
- 该共享数据最后的结果依赖于这些进程特定的执行顺序

主要的问题就是产生了一个竞争条件（Race Condition），当有多个进程同时对共享数据进行操作的时候，共享数据最终的状态依赖于进程的执行顺序，结果是不可预测的，显然不是我们想要的情况。

临界区问题

如何确保他们不会将新产生的数据放入到同一个缓冲区中，造成数据覆盖？



临界区问题就是我们需要确保多个快递员不会把产生的数据放到同一块缓冲区里。小区物业给快递桌添加一个登记本，它是用来登记快递的，必须登记了以后再放到快递桌里去。我们先填写登记表，看看能不能进去，如果 slot 3 中已经有物体了，那么我们就不能放到 slot 3 中。取完快递以后我们再更新登记表。这里面的关键点在于某一个时刻有且只有一个进程可以进入临界区中进行执行。

解决临界区问题的三个要求

1. 互斥访问：在同一时刻，有且仅有一个进程

可以进入临界区

2. 有限等待：当一个进程申请进入临界区之后

，必须在**有限的时间内**获得许可进入临界区
而不能无限等待

3. 空闲让进：当没有进程在临界区中时，必须

在申请进入临界区的进程中选择一个进入临
界区，保证执行临界区的**进展**

while(TRUE) {

 申请进入临界区

 临界区部分

 通知退出临界区

 其他代码

}

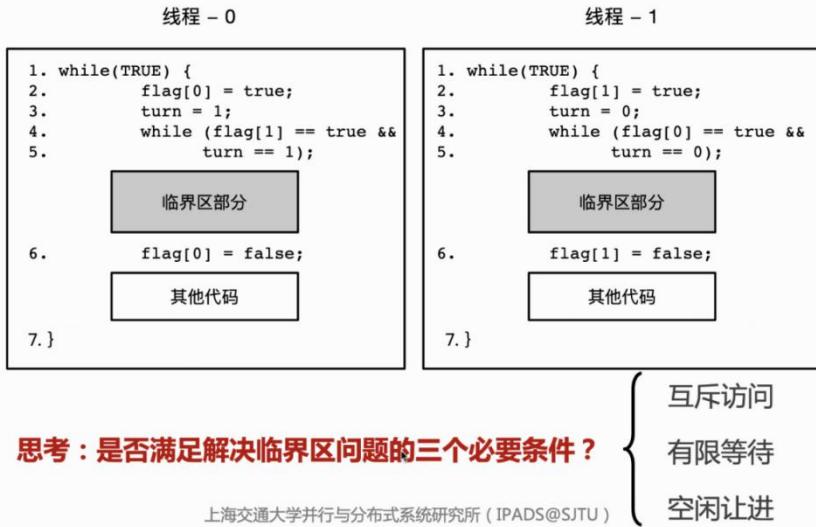
这也是我们同步里的三个关键要求，同一个时间有且只有一个进程可以进入临界区。进

程在申请以后必须在有限时间进入到临界区。第三个就是空闲让进，如果没有进程在临界区中，那么必须让申请的进程进入临界区。

基本同步原语：软件实现与硬件实现

那么我们怎么才能实现临界区呢？那么就需要我们提供一些基本的同步原语。

回顾软件解决方案：皮特森算法



首先我们回顾一些软件方法：皮特森方法，它比较经典，但是也没什么用。它来解决两个进程的互斥的问题。首先它有一个 `flag`（表示谁想进入临界区），`turn`（当前是谁的顺序可以进入到临界区）。

如果 `flag[1] == true` 且 `turn = 1`，它就等待。否则就进入临界区。同样进程 1 同理。如果 `turn=0` 就说明进程 0 在临界区里面。

临界区是有界的，所以皮森特算法是满足互斥访问、有限等待、空闲让进的。

对于线程 0：

`flag[0] = True` 将自身的访问位置为 1，`turn = 1` 向对方（线程 1）发出访问请求；
`while` 中的循环条件即阻塞条件为，若对方需要访问且对方未应答访问请求。

对于阻塞条件我们可以这样理解，访问位置为 1 即自己想要访问临界区，然后向对方发出访问请求。

- 若此时对方根本不想访问，那么不会被阻塞直接访问临界区，此时不会发生异常；
- 若此时对方将访问位已经置为 1，那么对方可能准备访问临界区或正在访问临界区：
 - 若对方正在访问临界区，那么此时对方不会应答自己的请求，则被阻塞；
 - 若对方准备访问临界区，访问前需要收到我方应答，此时先发送请求的一方必定会收到应答，即可以访问临界区，那么另一方因为请求后对方已经进入临界区，得不到应答，必须等对方退出临界区，将访问位置 0 才能访问。

在这样的机制下，双方实现了对临界区的互斥访问。

Q: 如果交换 turn 赋值语句和 flag 赋值语句的位置，能否实现互斥访问临界区？

A: 不能实现。如果交换位置，意味着在没有访问意愿的情况下先发送访问请求。此时若我方得到应答却突然阻塞，那么对方虽然不会再收到应答，但检测到我方根本不想访问，直接进入临界区。而当对方处于临界区时我方被唤醒，即使对方的访问位为 1，我方已经获得过应答，也可以访问临界区，则访问出现冲突。

[1]<https://zhuanlan.zhihu.com/p/262559638>

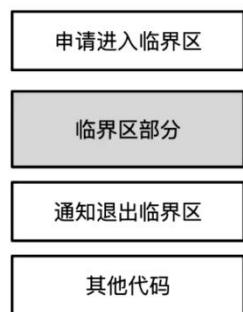
Q: 皮特森算法有什么问题？

A: 这个算法只能支持两个线程，flag[0], flag[1] 和 turn == 0, turn==1 是写死的，所以它不是一个通用的算法。

这个问题有没有更简单的方法？

有没有更简单的方法？比如关闭中断？

```
while(TRUE) {
```



这样能解决临界区问题吗？

关闭所有核心的中断

可以解决单个CPU

核上的临界区问题

如果在多个核心中，
关闭中断不能阻塞

开启所有核心的中断

其他进程执行

并不能阻止多个CPU核同时进入临界区

```
}
```

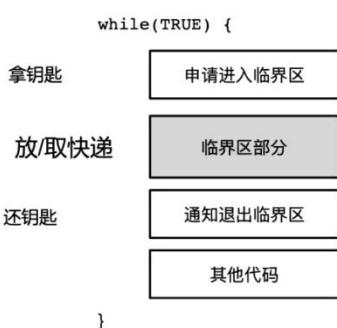
在 OS 中，我是经常会关闭中断的，比如 disable_irq、enable_irq 之类的代码。

Q: 关闭中断是不是就能解决临界区的问题了呢？

A: 可以解决单个 CPU 核上的问题，但是不能解决多个核上的其他进程的执行。

▶ 有没有更简单的方法？

如何确保他们不会将新产生的数据放入到同一个缓冲区中，造成数据覆盖？



给快递桌添加互斥锁

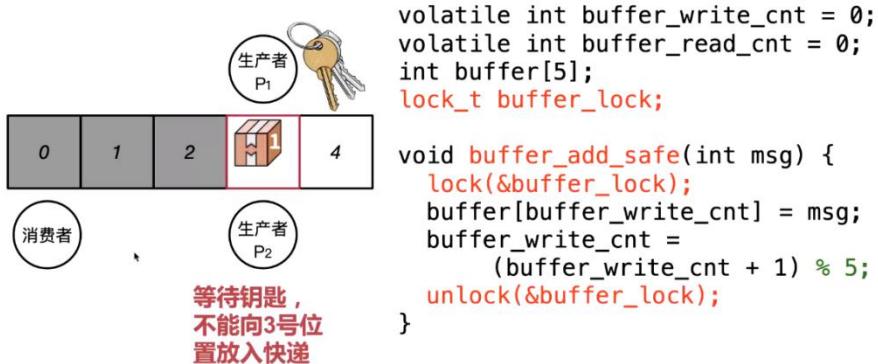


我们刚才讲皮特森算法，它有的问题就是没有硬件支持的话，虽然我们把 flag[0]=true; turn=1，在处理器执行的时候，现代处理器是乱序执行的，可能 flag[0]=true 在 turn=1 之后

执行，这就会出现问题。所以，我们在多核里面，哪怕我们认为我们的代码看起来是正确的，由于 CPU 要做很多优化，会把相关代码做一些移动，这会涉及到 memory-consistency model 的问题。

为了保证两个快递员不会把快递放到同一个格子里去，我们需要给快递桌添加互斥锁，这类似于我们丰巢的方式。我们拿取件码（钥匙）去取/放快递。

生产者放入有限缓冲区-使用互斥锁



我们来看一下怎么通过锁的方式来实现生产者和消费者。我们假如一把 `buffer_lock`，然后实现 `buffer_add_safe` 这个函数。其实就是进入临界区前加锁、退出临界区后还锁。

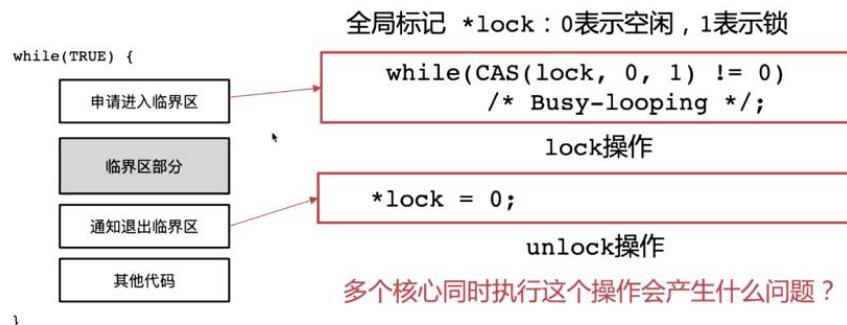
Compare And Swap (CAS) 操作

```

1. int CAS(int *addr, int expected, int new_value) {
2.     int tmp = *addr;
3.     if (*addr == expected)
4.         *addr = new_value;
5.     return tmp;
6. }

```

那么我们怎么实现这把锁呢？我们要介绍 Compare And Swap 的操作。从字面上理解，就是对比一下两个值，如果是一致的话，我们就写成一个新的值。



如果 `lock = 0`，我们就赋值成 1（拿锁），否则我们就继续等待。

Q：那么多个核执行这个操作会不会发送什么问题呢？

A：这就会涉及到 time-to-check 和 time-to-use 的问题。如果两个人同时检查，那么 `lock` 此时都等于 0，符合条件。两个人会同时去赋这个 `lock` 的新的值，所以可能两个线程同时进入

到核心区里面。

所以本质的问题还是我们希望这几个操作是原子的操作，要么没执行，要么都执行。所以需要硬件来提供这个支持，它就像一条指令一样，其他 CPU 核看不到中间的状态，也就是 all-or-nothing。

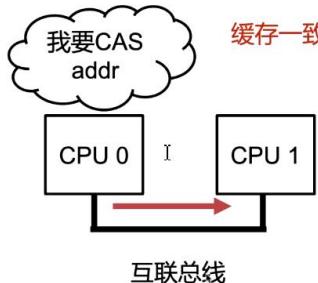
硬件原子操作

原子操作：

- 不可被打断的操作集合
- 如同执行一条指令
- 其他核心不会看到中间状态 all-or-nothing

怎么硬件怎么把这几条指令当做原子操作来执行呢？

硬件原子操作：Intel 锁总线实现



缓存一致性后面会介绍

```
1. int CAS(int *addr, int expected,
CPU 0           int new_value) {
2.     int tmp = *addr;
3.     if (*addr == expected)
4.         *addr = new_value;
5.     return tmp;
6. }
```

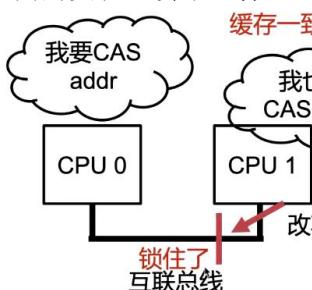
对任意地址的修改都要经过总线的

通过锁总线来实现原子操作*

在第二行锁总线，在第五行放总线

*Intel手册描述，实际实现可能与此不同（更高效）

如果大家去看 CAS 的汇编实现的话，它会用到一个 Lock 的前缀。Intel 的实现就是会把总线锁住，然后在执行过程中间保证不会被打断、状态也不会被看到。总线就是一个通用的描述，不同的实现也会不一样。



缓存一致性后面会介绍

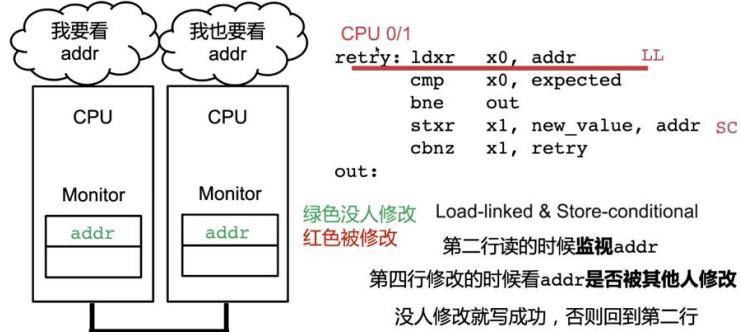
```
1. int CAS(int *addr, int expected,
CPU 1 没法向下 int new_value) {
2.     int tmp = *addr;
3. CPU 0 if (*addr == expected)
4.         *addr = new_value;
5.     return tmp;
6. }
```

这样的话，如果另一个核也需要 CAS(addr)，它就会因为总线被锁而不能往下继续执行。

LL/SC 机制 (Load-linked & Store-conditional)

因为我们目前主要使用 ARM 的方式来提供 OS 的实验，我们来看一下 ARM 里实现 LL/SC 的实现。

硬件原子操作 : ARM 使用 LL/SC 实现

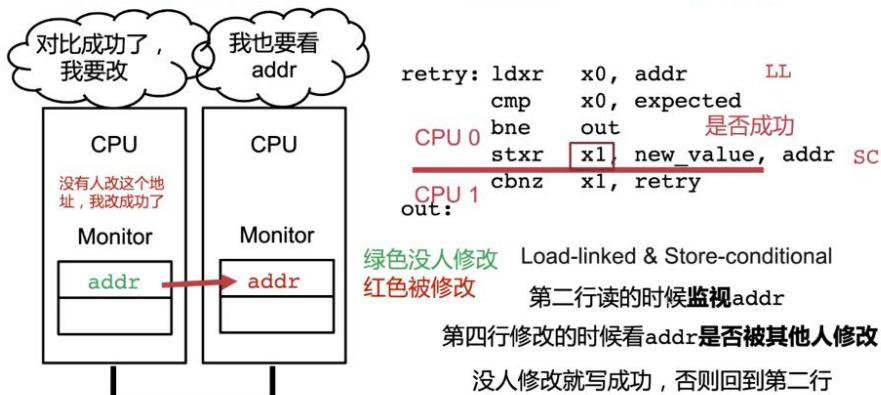


它的方式也是类似的。我们去判断一下，只有它的值和我们预期的值是相同的话，才能进去。LL/SC 方式就先进去看看，如果我们真的去写它的时候，发现原来读的地址发生变化了，说明过程中间还有其他人修改了这个地址，那么就说明我们这次 store 就是失败的。这个更像是乐观的一种方式。

Q: 那么为什么 ARM 会使用 LL/SC (Load-linked & Store-conditional) 呢？

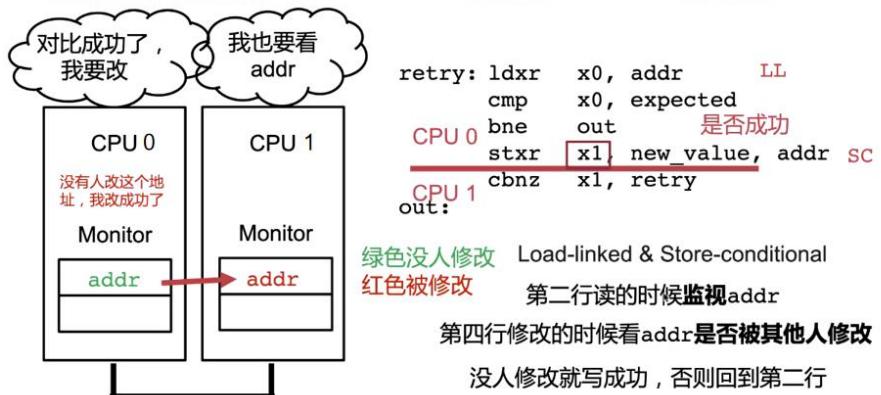
A: 因为在 CAS 的实现我们要锁总线，这样一次只能有一个在执行。而 LL/SC 是监控了一个地址，这样在过程中间一次可以支持无数的 LL/SC 在执行，这样可扩展性会更好。

硬件原子操作 : ARM 使用 LL/SC 实现



我们还看一下这个过程。在当前场景下，CPU0 和 CPU1 同时想对我们的 addr 做 CAS 操作。CPU0 率先在 SC 阶段发现没有修改过 addr，所以 CPU0 把 addr 修改成了新的 new_value。

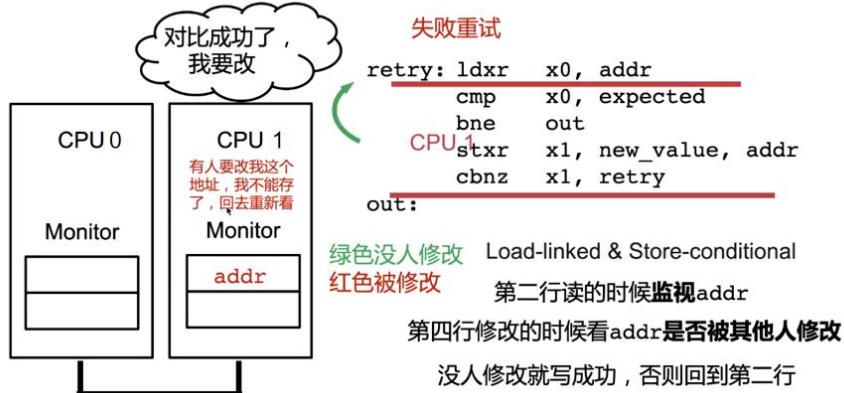
硬件原子操作 : ARM 使用 LL/SC 实现



此时 CPU1 来到了 SC 阶段，发现自己想修改的 addr 已经被人修改成了 new_value，那

么我们必须放弃这次修改，跳转回 retry 位置等待。

硬件原子操作：ARM使用LL/SC实现



我们刚才也介绍了快递员的问题，我们需要提供一个有效的机制保证快递不会被覆盖。具体地，我们可以看一下硬件怎么来支持这一点。

写硬件原子操作：使用硬件辅助

比较并替换（Compare And Swap）

```
1. int CAS(int *addr, int expected, int new_value) {  
2.     int tmp = *addr;  
3.     if (*addr == expected)  
4.         *addr = new_value;  
5.     return tmp;  
6. }
```

获取并增加（Fetch And Add）

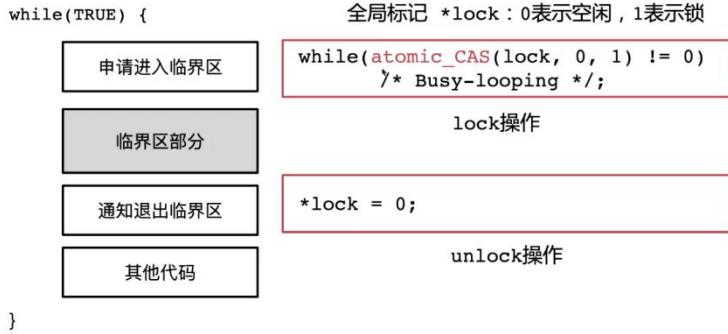
```
1. int FAA(int *addr, int add_value) {  
2.     int tmp = *addr;  
3.     *addr = *addr + add_value;  
4.     return tmp;  
5. }
```

CAS 和 FAA 都被做成了原子的指令。

锁的实现

自旋锁（Spinlock）

第一类就是我们的自旋锁：



atomic_CAS 就是支持原子操作的 CAS 指令。这样我们在拿锁的时候只需要通过循环来一直尝试拿锁，而 unlock 就只需要把 0 赋值给 lock。

自旋锁 (Spinlock)

思考：是否满足解决临界区问题的三个必要条件？

- 互斥访问 ✓
- 有限等待 ?
 - 有的“运气差”的进程可能永远也不能成功CAS => 出现饥饿
- 空闲让进 ?
 - 依赖于硬件 => 当多个核同时对一个地址执行原子操作时，能否保证至少有一个能够成功*

```

void lock(int *lock) {
    while(atomic_CAS(lock, 0, 1)
        != 0)
        /* Busy-looping */ ;
}

void unlock(int *lock) {
    *lock = 0;
}

```

自旋锁实现

*这里我们认为硬件能够确保原子操作make progress

如果有很多人在尝试进入这个锁，线程是否会无限等下去呢？比如大家排队等饭，前面老有人插队，那么就可能进入了饥饿的状态。因为大家都在 CAS 的时候，到底谁能 CAS 成功，并没有一个保证。比如 10 个人 CAS，每次有一个人进去了，可能第 10 个人一直倒霉进不去。所以不满足一个有限等待的问题，这是一个极低概率的情况下会出现饥饿。

空闲让进：这是依赖于硬件的，多个核同时对一个地址执行原子操作的时候，是否能够保证至少有一个线程可以成功呢？我们认为硬件是可以保证这件事情的。

为了避免饥饿的情况，我们需要想一种更好的方式。

排号锁 (Ticket Lock)

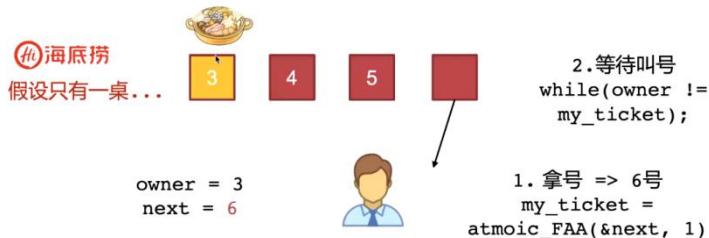
我们要保证竞争者的公平性，比如谁先下去排在前面谁先做核酸。

排号锁 (Ticket Lock)

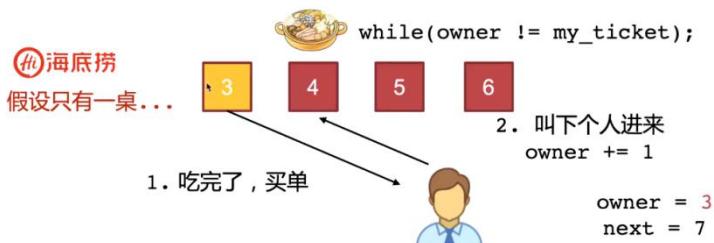
思考：我们如何保证竞争者的公平性？

通过遵循竞争者到达的顺序来传递锁。

owner : 表示当前在吃的食客 next : 表示目前放号的最新值



排号锁简单地遵循竞争者到达的顺序来传递锁。最理想的方式就是大家取个号，号快到了大家再过去。



整个操作如下图所示：



思考：是否满足解决临界区问题的三个必要条件？

- 互斥访问 ✓
- 有限等待 ?
 - 按照顺序，在前序竞争者保证有限时间释放时，可以达到有限等待
- 空闲让进* ✓

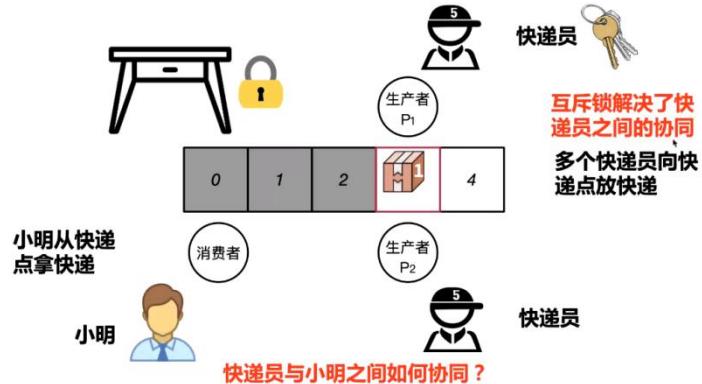
```
void lock(int *lock) {  
    volatile unsigned my_ticket =  
        atomic_FAA(&lock->next, 1);  
    while(lock->owner != my_ticket)  
        /* busy waiting */;  
}  
  
void unlock(int *lock) {  
    lock->owner ++;  
}
```

排号锁实现

这三个条件都是满足的。

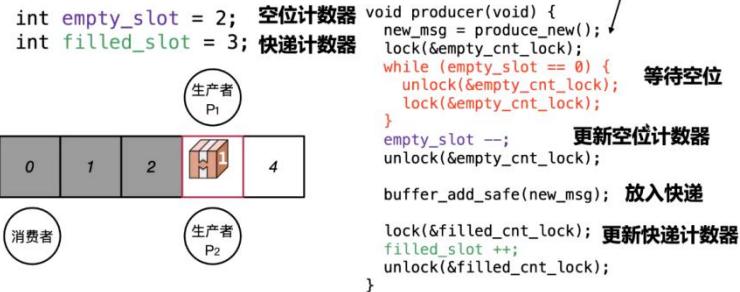
条件变量

生产者消费者问题



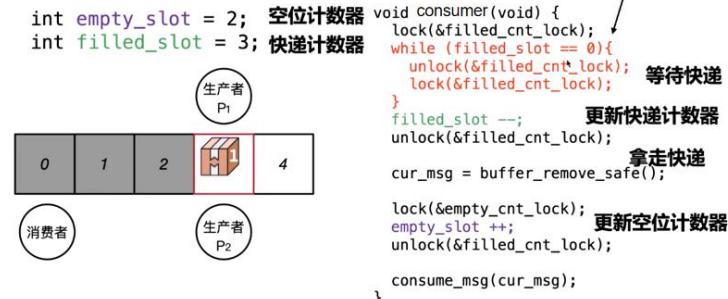
在快递问题中，快递员和小明该如何协同？在真实世界中，快递放到快递柜中，会给小明发一个短信。快递被拿走了，有新的快递柜空间空闲出来了也会给快递员发一个短信。问题就是说，我们的“短信”怎么来实现。

生产者消费者问题-生产者



我们要计算一下空位有多少、填满的格子有多少，并且我们使用互斥锁来保护我们的计算器。当没有空位的时候，我们就使用互斥锁来等待空位。我们不能一直持有锁等待，这样别的人就没有机会了，所以我们在等待的时候要 `unlock`，再尝试 `lock`。如果有空位就放入快递，再更新我们的快递计数器。

生产者消费者问题-消费者



`consumer` 这边也是去等待快递，如果快递柜中有快递，那么就拿走快递。

但是这个过程还是比较低效的，有大量的锁的操作和盲等。如果快递没有到的话，小明需要一直盲等在快递柜边上，其他事情都做不了了。所以我们需要有一种高效的方式，也就

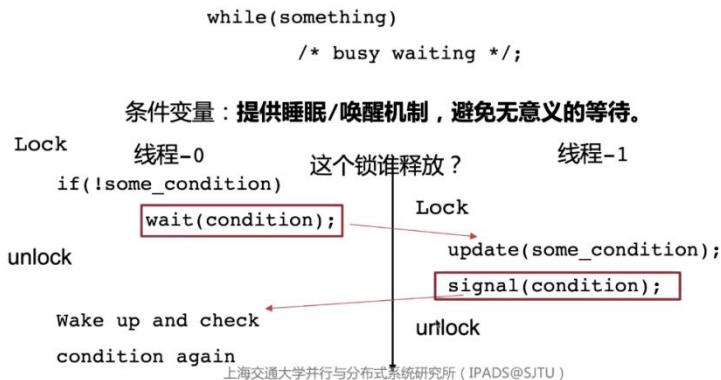
是发短信通知这种方式。

条件变量

条件变量

之前互斥锁的实现中：

需要使用互斥锁保护共享条件的更新



条件变量就是如果某个条件得到满足的时候，我们来了以后就打个电话。提供了睡眠和唤醒的机制。所以线程 0 就在说不满足某个条件的时候，就去等待这个条件。如果快递来了，那么快递员就去通知一下它。小明把自己放到通知的队列里面，这样快递员来了就可以通知到他。signal 就是告诉等待的人已经到了。我们同样需要互斥锁来保护我们共享变量的更新。

条件变量的接口

提供的两个接口：

等待的接口：

等待需要在临界区中

```
void cond_wait(struct cond *cond, struct lock *mutex);
```

1. 放入条件变量的等待队列
2. 阻塞自己同时释放锁
3. 被唤醒后重新获取锁

唤醒的接口：

```
void cond_signal(struct cond *cond);
```

1. 检查等待队列
2. 如果有等待者则移出等待队列并唤醒

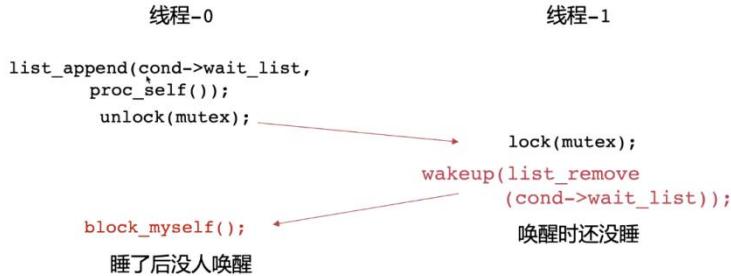
`cond_wait` 就是等待某个条件的成立，我们还需要传入一把 `mutex`。在等待（睡眠）的时候，我们不能拿着锁去等。唤醒以后要重新获取这把锁，对所有条件的操作必须持有锁，所以在 `cond_wait` 中就有释放锁和获取锁的过程。

而唤醒的接口就是根据等待队列来进行唤醒。

条件变量的实现

思考：为什么这里要原子的 atomic_block_unlock ?

wakeup必须在临界区内同理



我们 wakeup 就是从等待队列中找出一个人来唤醒。这里会带来两个问题，睡了以后没有人来唤醒它、唤醒时还没睡。

条件变量的使用示例

等待空位代码

```
1. ...
2. /* Wait empty slot */
3. lock(empty_cnt_lock);
4. while (empty_slot == 0)
5.     cond_wait(empty_cond,
6.                empty_cnt_lock);
7. empty_slot--;
8. unlock(empty_cnt_lock);
9. ...
```

生产空位代码

```
1. ...
2. /* Add empty slot */
3. lock(empty_cnt_lock);
4. empty_slot++;
5. cond_signal(empty_cond);
6. unlock(empty_cnt_lock);
7. ...
```

思考：为什么这里要用while？

Q: 这里为什么用的是 while 而不是 if 呢？

A: if 的情况下，如果有多人在执行这一段代码，当我们有一个 slot 空位的时候，如果有多人在等的话，第一个线程正常执行完 empty_slot -- 时，此时已经不满足唤醒条件了，但是 if 可能会导致其余线程仍然跳出条件段，导致出错。

信号量

信号量 (PV原语)

信号量：协调（阻塞/放行）多个线程共享有限数量的资源。



Edsger W. Dijkstra

语义上：信号量记录了当前可用资源的数量

提供了两个原语用于等待/消耗资源

P操作：消耗资源

cnt代表剩余资源数量

```
void sem_wait(sem_t *sem) {
    while(sem->cnt <= 0)
        /* Waiting */;
    s--;
}
```

信号量是比较经典的概念。Dijkstra 也是 OS 大牛，信号量又叫 PV 原语。P 表示去消耗

资源，`v` 表示增加资源。`sem_wait` 其实就是等待信号量中的值大于 0，然后减少这个资源。

v操作：增加资源

```
void sem_signal(sem_t *sem) {  
    sem->cnt ++;  
}
```

只展示语义，并非真实实现

那如果我们使用信号量来实现我们的生产者消费者的话，我们就可以把刚才的 `lock`、`wait` 等操作变成一个 `sem_wait`，这是一个更好的 `wait`。

信号量的使用

```
while(true) {  
    new_msg = produce_new();  
    + lock(&empty_slot_lock);  
    while (empty_slot == 0)  
        + cond_wait(&empty_cond,  
                    &empty_slot_lock);  
    empty_slot --;  
    + unlock(&empty_slot_lock);  
  
    buffer_add(new_msg);  
    // ...  
}  
  
void producer(void) {  
    new_msg = produce_new();  
    sem_wait(&empty_slot_sem);  
    buffer_add(new_msg);  
    // ...  
}
```

将空位看成一种资源：
生产者需要消耗空位
消耗empty_slot

使用信号量可以将其压缩到一行代码

我们把空位也看成一种资源，就可以使用信号量来进行抽象。这样代码要简单很多。

```
void producer(void) {  
    new_msg = produce_new();  
    sem_wait(&empty_slot_sem);  
    buffer_add(new_msg);  
    sem_signal(&filled_slot_sem);  
    // ...  
}  
  
void consumer(void) {  
    sem_wait(&filled_slot_sem);  
    cur_msg = buffer_remove();  
    sem_signal(&empty_slot_sem);  
    handle_msg(cur_msg);  
}
```

所以我们看到整个过程的话要简单很多。

二元信号量与计数信号量

```
void sem_init(sem_t *sem, int init_cnt) {  
    sem->cnt = init_cnt;  
}
```

当初始化的资源数量为1时，为二元信号量

其计数器只可能为0、1两个值，故被称为二元信号量

同一时刻只有一个线程能够拿到资源

当初始化的资源数量大于1时，为计数信号量

同一时刻可能有多个线程能够拿到资源

这样一个时刻只有一个线程能拿到资源。

信号量实现：基本语义

通过计数器来协调（阻塞/放行）多个线程的执行。

语义上：

```
void wait(int S) {          P操作
    while(S <= 0)
        /* Waiting */;
    S--;
}

void signal(int S) {         V操作
    S++;
}
```

这样实现是否会有问题？为什么？

Q：这个实现是否有问题？

A：首先我们要把对信号量的操作变成原子的。

```
void wait(int S) {          P操作
    while(S <= 0)
        /* Waiting */;
    atomic_add(&S, -1);
}

void signal(int S) {         V操作
    atomic_add(&S, 1);
}
```

使用原子操作后呢？是否还有问题？

信号量的错误实现

```
sem 初始值为 0
线程0           线程1           线程2
                wait(sem);      wait(sem);
                signal(sem)       离开while循环      离开while循环
                atomic_add(&sem, -1); atomic_add(&sem, -1);

sem = -1      线程1,线程2同时以为自己拿到了资源
```

这就是一个典型的并发错误。我们还是需要用到 Lock 的方式进行实现。

信号量的实现-1：忙等

```
void wait(sem_t *S) {
    lock(S->lock);
    while(S-> value <= 0) {
        unlock(S->lock);
        lock(S->lock);           Busy looping, 无意义等待
    }
    S-> value --;
    unlock(S->lock);
}

void signal(sem_t *S) {
    lock(S->lock);
    S-> value++;
    unlock(S->lock);
}
```

第一种实现就是盲等。

信号量的实现-2：条件变量

```
void wait(sem_t *S) {
    lock(S->sem_lock );
    while(S-> value == 0) {           使用条件变量避免无意义等待
        cond_wait(S->sem_cond,
                   S->sem_lock);
    }
    S-> value--;
    unlock(S->sem_lock);           能否保证有限等待？
}

void signal(sem_t *S) {
    lock(S->sem_lock);
    S-> value++;                  但每次都要signal，比较耗时
    cond_signal(s->sem_cond);
    unlock(S->sem_lock);
}
```

上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

61

第二种就是使用条件变量来避免盲等。我们每次就需要 `cond_wait` 和 `cond_signal`。但是这样的话每次都需要 `signal`，比较耗时。

信号量的实现-3：减少signal次数

```
void wait(sem_t *S) {
    lock(S->sem_lock );
    S->value--;
    while(S->value < 0) {           value减到负数代表有人等待
        cond_wait(S->sem_cond,
                   S->sem_lock);      会有什么问题？
    }
    unlock(S->sem_lock);          比如S->value = -3
}

void signal(sem_t *S) {
    lock(S->sem_lock);
    S->value++;                  signal 后S->value = -2
    if (S->value < 0)            需要额外的计数器用于单独记录
        cond_signal(s->sem_cond);有多少可以唤醒的
    unlock(S->sem_lock);         加入条件判断是否需要wake
}
```

上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

62

额外的计数器是记录有多少等待唤醒的人，如果我们想发信号的时候发现没有人等着，那么我们就可以省去这次发信号。

信号量的实现

介绍一种使用条件变量的实现：条件变量 + 互斥锁 + 计数器 = 信号量

value：正数为信号量，负数为有人等待 wakeup：等待时可以唤醒的数量

某一时刻真实的信号量： value > 0 ? value + wakeup : wakeup

```
void signal(struct sem *S) {
    lock(S->sem_lock);
    S->value++;
    if (S->value <= 0) {
        S->wakeup++;
        cond_signal(S->sem_cond);
    }
    unlock(S->sem_lock);
}

void wait(sem_t *S) {
    lock(S->sem_lock);
    S->value--;
    if (S->value < 0) {
        do {
            cond_wait(S->sem_cond, S->sem_lock);
        } while (S->wakeup == 0);
        S->wakeup--;
    }
    unlock(S->sem_lock);
}
```

上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

64

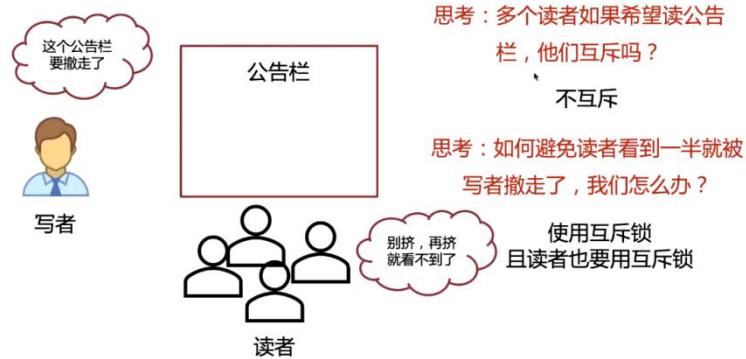
```
void wait(sem_t *S) {
    lock(S->sem_lock);
    S->value--;
    if (S->value < 0) {
        while (S->wakeup == 0){
            cond_wait(S->sem_cond, S->sem_lock);
        }
        S->wakeup--;
    }
    unlock(S->sem_lock);
}
```

思考：为何要do while？有限等待

时刻	线程 0	线程 1
T0	wait(&S) 挂起等待	
T1		signal(&S)
T2		wait(&S) 拿到资源
T3	被唤醒，发现没有可用资源	

最后我们简单过一下读写锁：

公告栏问题



通知的人是一个写者，大家都是读者。那么公告栏就是一个写者多个读者的情况，显然读者之间不是互斥的。那么大家读到一半通知被读者撤走了该怎么办呢？所以我们需要使用到互斥锁。

读写锁的使用示例

```
struct rwlock *lock;
char data[SIZE];

void reader(void) {
    lock_reader(lock);
    read_data(data);
    unlock_reader(lock);
}

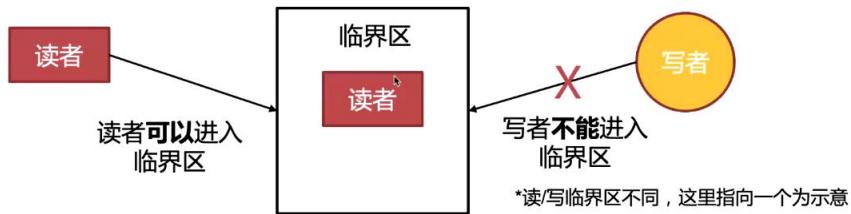
void writer(void) {
    lock_writer(lock);
    update_data(data);
    unlock_writer(lock);
}
```

读写锁

互斥锁：所有的进程均互斥，同一时刻只能有一个进程进入临界区

对于部分只读取共享数据的进程过于严厉

读写锁：区分读者与写者，允许读者之间并行，读者与写者之间互斥



2022/3/22

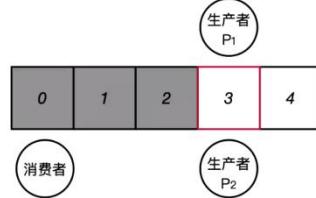
今天我们继续讲同步原语和死锁的问题。上节课介绍了四种。OS 只做三件事情：虚拟化、持久化、同步。

场景一：共享资源互斥访问

多个线程需要同时访问同一共享数据

应用程序需要保证互斥访问避免数据竞争

```
int shared_var = 0;
void thread_1(void) {
    shared_var = shared_var + 1;
}
void thread_2(void) {
    shared_var = shared_var - 1;
}
```

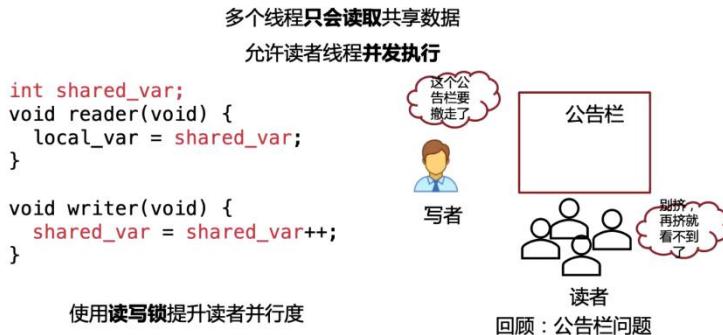


OS 管理应用程序，应用程序之间要做协调的话就需要 OS 来协助。那么共享资源互斥访

问的话，就是 OS 最常见的一个场景。最早的例子就是 count++这个例子，就会有 race condition 的问题。

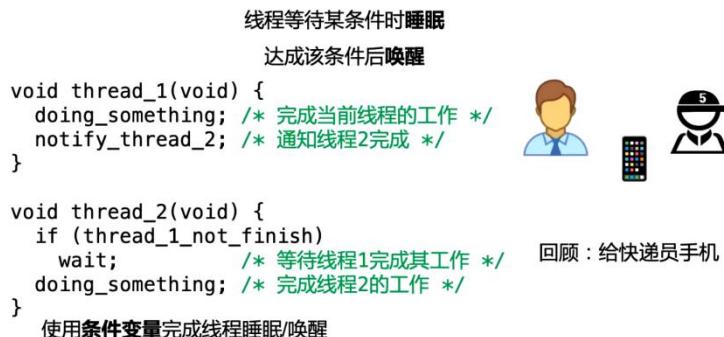
在单个生产者和消费者的情况下不会出现太大的问题，而这里说的是多个生产者和消费者的问题。解决方案就是互斥锁，我们不 care 顺序是什么，只要不同时加就可以了。

衍生场景一：读写场景并发读取



大量都是读操作很少的写操作，我们就可以通过读写锁来做优化。网页也是读的人很多、写的人很少，否则读者之间互斥是没有多大意义的。

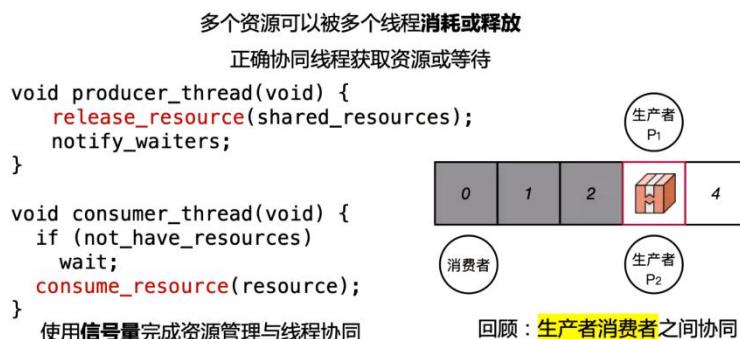
场景二：条件等待与唤醒



第二个场景就是有条件等待的问题。这和互斥是不一样的，我们最终希望给你一个场景我们能够划归到这三类经典场景。然后我们就知道可以用什么同步原语去解决了。所以我们在提场景的时候，我们会提它的核心的特征。条件等待就是需要排队，也就是先后是有关的。一定是先有生产者，再有消费者的情况。满了之后一定是先有消费者再有生产者。这样就可以抽象为条件。有人在等待条件满足。

条件变量的实现用到了互斥锁，但是它本身和互斥锁用到的场景是不同的。

场景三：多资源协调管理



从资源的角度，我们就会有释放资源和消耗资源这个抽象。假如 100 个人只有 10 个资源，那么我们怎么去协调这个资源。它不是简单的互斥，而是 10 个资源等待使用。所以我们划归到互斥问题显然不合适，我们使用值为 n 的信号量是比较合理的。信号量大于 0 就说明还有资源，可以让人进入到临界区去使用资源。

当 n=1 的时候，我们就可以等价为一个互斥锁。

场景与同步原语总结

同步原语	描述	使用场景
互斥锁	保证对共享资源的互斥访问	场景一 共享资源互斥访问
读写锁	允许读者线程并发读取共享资源	衍生场景一 读写场景并发读取
条件变量	提供线程睡眠与唤醒机制	场景二 条件等待与唤醒
信号量	协调有限数量资源的消耗与释放	场景三 多资源协调管理

同步原语对比

互斥锁 vs 读写锁

- 读写锁为特定场景（衍生场景一）下提升读者并行度
- 衍生场景一直接使用互斥锁
 - 也可以保证正确性
 - 读者之间不能并发执行
- 特定场景下的性能优化

互斥锁 vs 条件变量

- 解决不同场景（正交）
 - 互斥锁：互斥访问
 - 条件变量：条件等待与唤醒
- 条件变量需要搭配互斥锁使用
- 互斥锁中也可以使用条件变量避免循环等待
 - 场景一中也可能包含场景二的需求

它们是用来解决不同的场景的。互斥不考虑顺序，而条件等待是考虑顺序的。

互斥锁 vs 信号量

• 互斥锁与二元信号量

- 功能基本一致，二元信号量可以实现互斥
- 语义差别：二元信号量可以由不同的线程获取/释放。互斥锁语义上只能由同一个线程获取与释放
- 保护资源互斥场景，推荐使用互斥锁（方便优化）

• 互斥锁与计数信号量

- 应对不同场景
 - 互斥锁控制对唯一资源的互斥访问（即临界区）
 - 计数信号量控制多个线程对多个资源的获取与释放

条件变量 vs 信号量

• 提供了类似的接口

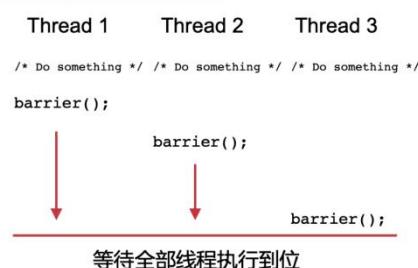
• 抽象层级的区别

- 条件变量
 - 更底层，提供睡眠唤醒机制
 - 适用范围更广
- 信号量
 - 针对具体的场景：提供对**有限资源**的管理
 - 可以使用**条件变量+互斥锁+计数器**实现信号量

同步案例-1：多线程执行屏障

• 多线程执行屏障

• 等待全部执行到屏障 后再继续执行



我们有多个 thread，我们需要执行到同一个点以后，再重新执行下一轮。也就是三者要等一个东西，等到了才能继续往下走。一旦涉及到等，那么其实就是我们场景二：等待/唤醒的场景。

符合场景2：线程等待/唤醒

```
lock(&thread_cnt_lock);
thread_cnt--;
if (thread_cnt == 0)
    cond_broadcast(cond);      唤醒所有等待的线程
while(thread_cnt != 0)
    cond_wait(&cond, &thread_cnt_lock);
unlock(&thread_cnt_lock);
```

等待全部线程执行到位

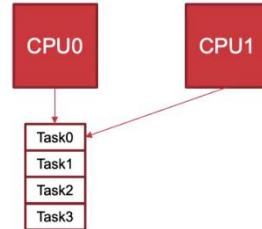
如果 `thread_cnt == 0` 就代表所有线程都到了，那么就唤醒所有等待的线程，否则我们就去等待。因为场景本身没有条件，所以我们要加入条件 `thread_cnt == 0`。

同步案例-2：等待队列工作窃取

- 每核心等待队列
- 在空时允许窃取其他核心的任务

符合场景1: 共享资源互斥访问

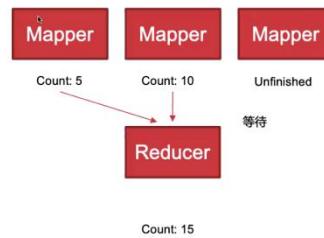
```
lock(ready_queue_lock[0]);
```



每个核上都会有一个 run queue，一些还没有被执行的任务在排队。如果 CPU1 发现自己的任务做完了，那么我们就可以偷一个任务过来。因为我们要对数据结构做更改，所以这是一个互斥的场景。当我们去拿任务的时候，必须只有一个 CPU 拿到任务，我们不关心是谁拿到，所以我们通过互斥锁来解决问题。

同步案例-3：map-reduce

- Word-count : 大文本拆分子数统计
- Mapper : 统计一部分文本自述
- Reducer : 一旦其中任意数量的 Mapper 结束，就累加其结果



出现了 Reducer 需要等 Mapper 的关系，所以符合我们的场景二。

符合场景2: 线程等待/唤醒

Mapper

```
lock(&finished_cnt_lock);
finished_cnt++;
cond_signal(&cond);
unlock(&thread_cnt_lock);
```

符合场景2: 线程等待/唤醒
Reducer

```
lock(&finished_cnt_lock);
while(finished_cnt == 0)
    cond_wait(&cond, &finished_cnt_lock);
/* collect result */           一次性拿走所有的finished
finished_cnt = 0;             的Mapper的结果
unlock(&thread_cnt_lock);
```

如果 Reducer 从 while 里跳出来了，那么说明有 Mapper 的结果到了，那么就去计算一下结果。我们也可以把 mapper 的结果看成一种资源，这样我们就可以到场景三了。

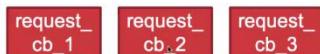
Mapper	Reducer
	<pre> while(finished_cnt != mapper_cnt) { signal(&finish_sem); wait(&finish_sem); /* collect result */ finished_cnt++; } </pre>

思考：有什么不同？

第一个非常大的区别就是，用信号量的代码更少。`finish_sem` 只能每次唤醒一个，不能做到一次把每个有结果的 `mapper` 都收走。而前面的 `CV` 一旦醒过来可能看到多个 `mapper` 都 `ready` 了。

同步案例-4：网页渲染

- 网页等待所有的请求均完成后
再进行渲染



场景2: 等待/唤醒

Request_cb	渲染线程	渲染线程
<pre> lock(&glock); finished_cnt++; if (finished_cnt == req_cnt) cond_signal(&gcond); unlock(&glock); </pre>	<pre> lock(&glock); while (finished_cnt != req_cnt) cond_wait(&gcond, &glock); unlock(&glock); </pre>	

这个和第一个案例有点像，这里是一个线程等待三个线程。我们也可以使用信号量来实现：

场景3: 视为所有请求结果为资源

Request_cb	渲染线程	
<pre> signal(&gsem); </pre>	<pre> while(remain_req != 0) { wait(&gsem); remain_req--; } </pre>	

我们可以把所有请求结果本身看成资源，并且这个代码显然比之前的代码要简单，但是缺点就是我们要 `wait` 多次。

同步案例-5：线程池并发控制

- 控制同一时刻可以执行的线程数量
- 原因：有的线程阻塞时可以允许新的线程替上
- 例子：允许同时三个线程执行



有 4 个线程，但同时有三个线程可以执行。

场景3: 视剩余可并行执行线程数量为有限资源

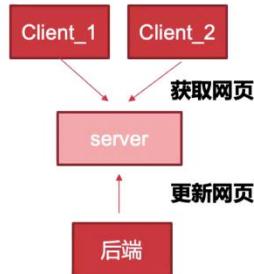
```

thread_routine () {
    wait(&thread_cnt_sem);
    /* doing something */
    signal(&thread_cnt_sem);
}

```

同步案例-6：网页服务器

- 处理响应客户端获取静态网页需求
- 处理后端更新静态网页需求
- 不允许读取更新到一半的页面

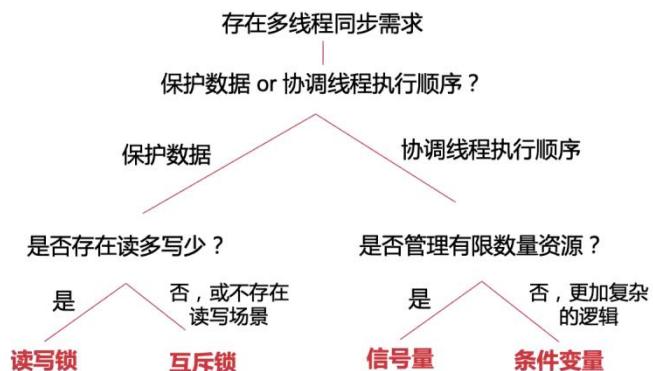


我们要保证客户端不会读到渲染到一半的网页。

衍生场景1：读写场景，可以使用读写锁

- client用读锁
- 后端用写锁

同步原语选择guideline



RCU：更高效的读写互斥

接下来我们来讲性能更高的同步原语 RCU。内核对性能的要求非常的高，要尽可能减少等待的时间。互斥锁一用要么盲等要么 block 住，性能就会变差。

Read Copy Update, RCU

读写锁读者进入读临界区之前，还是需要繁杂的操作

思考：如果我们想去除这些操作，让读者即使在有写者写的时候随意读，我们需要做什么？

如果每次读操作都要拿一个写锁，其实是大部分浪费掉的时间。

思考：如何让读者即使在有写者写的时候也能随意读？

需求1：需要一种能够**类似之前硬件原子操作**的方式，让读者要么看到旧的值，要么看到新的值，不会读到任何中间结果。

硬件原子操作：

1. 硬件原子操作有大小限制（最大128 bit）
2. 性能瓶颈

我们想做到就算有写者在，我们也可以随意读。回忆一下我们在 CSE 中学过 MVCC 的概念，它就是一种并发控制。MVCC 的思路就是：我可以让你读，但是读的是旧值，而不是新值。我们保证 before-or-after atomicity。在 OS 里怎么实现类似的 MVCC 的能力呢？我们需要有原子操作的方式让读者要么看到旧的要么看到新的值。

如果所有数据都小于 128b，那么完全 OK，但是现实中的数据结构是很大的，比如链表。

思考：如果我们想去除这些操作，让读者即使在有写者写的时候随意读，我们需要做什么？

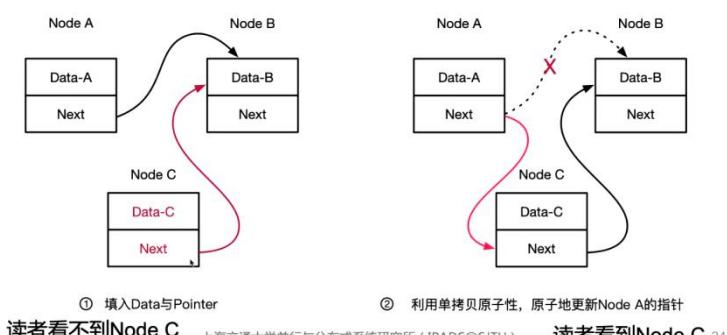
需求1：需要一种能够**类似之前硬件原子操作**的方式，让读者要么看到旧的值，要么看到新的值，不会读到任何中间结果。

单拷贝原子性 (Single-copy atomicity)：

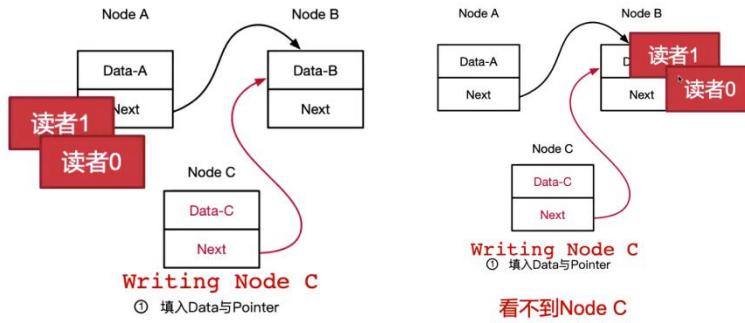
处理器任意一个操作的是否能够原子的可见，如更新一个指针

RCU 订阅/发布机制

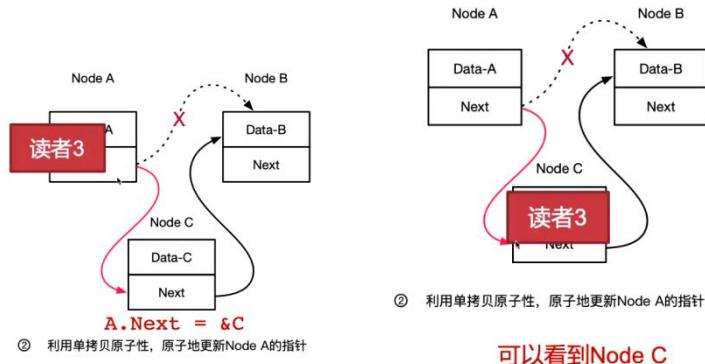
以链表为例：插入结点Node C



我们插入节点 C 以后，没有读者可以看到它。然后，当 Node A 把 next 指针从 B 指向 C 的时候，这就是一个原子操作。这样所有新进来遍历链表的人，都可以看到节点 C。

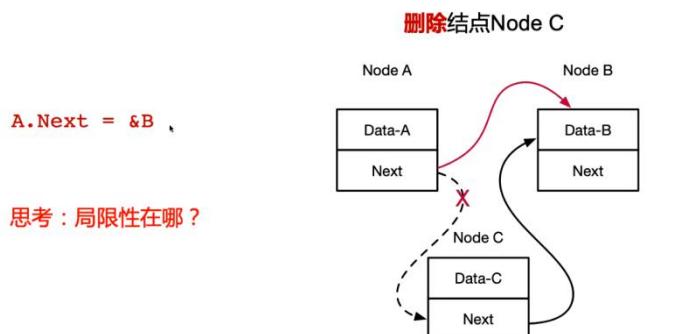


在完成原子操作前，读者 0 和读者 1 遍历链表的时候是看不到节点 C 的。



在完成原子操作后，读者 3 再尝试遍历链表的时候就可以看到节点 C 了。

RCU 订阅/发布机制

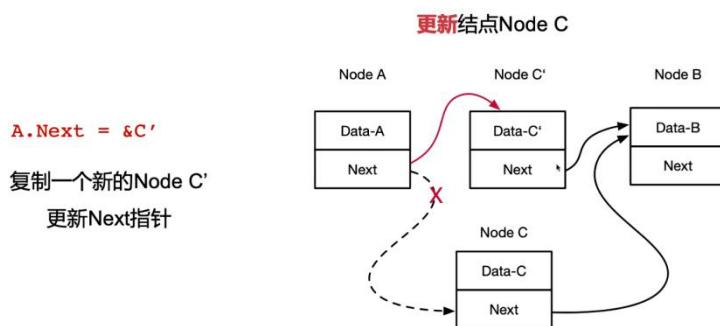


我们想删除的时候该怎么办呢？我们也是使用类似的方法修改 Next 指针。

Q: 但是这个操作在场景下可能出现问题呢？

A: 双链表有两个指针的情况下，那么我们就不能使用一行指令去解决删除元素的事情。

RCU 订阅/发布机制



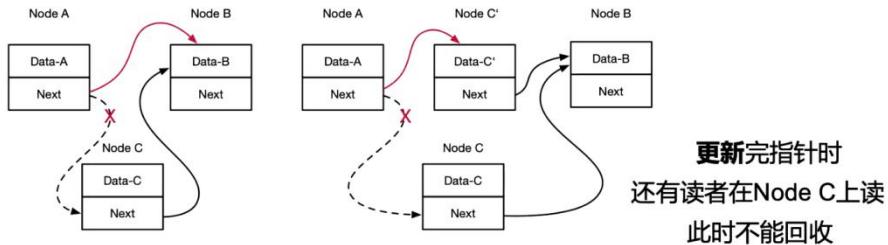
为什么我们不在原来的节点 C 中更新 Data-C' 呢？因为我们没办法保证更新的数据是小于 128bit 的，可能就不原子了。

Read Copy Update, RCU

读写锁读者进入读临界区之前，还是需要繁杂的操作

思考：局限性在哪？ 我们需要回收无用的旧拷贝

需求2：在合适的时间，回收无用的旧拷贝

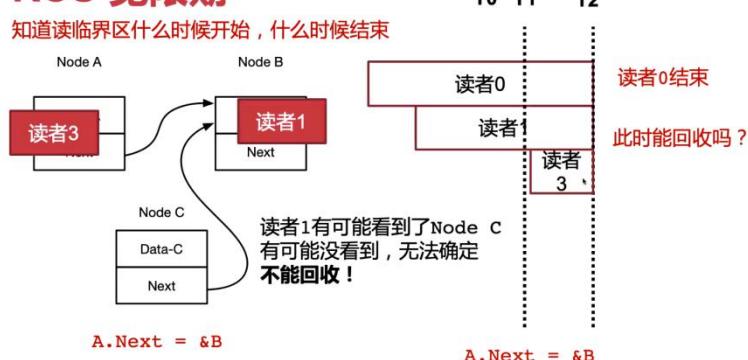


RCU 其实还有一个问题。节点 C 是旧的数据，最后我们要做节点 C 的回收，那么我们怎么知道节点 C 什么时候会被回收呢？比如此时正好有一个 reader 在读旧的节点 C。

加锁可以实现我们的目标，但是这就违背了我们不希望在读的时候加锁的初衷了。

RCU 宽限期

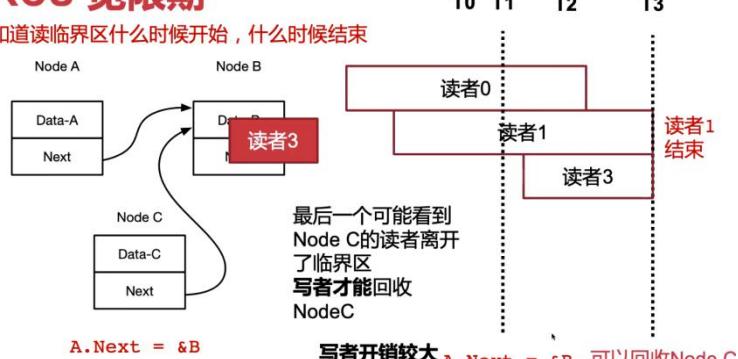
知道读临界区什么时候开始，什么时候结束



解决方案就是宽限期。我们必须等到读者 1 结束，我们才能把它回收。

RCU 宽限期

知道读临界区什么时候开始，什么时候结束



换句话说，我们删除掉节点 C 的时刻，读者 0 和读者 1 都在临界区，如果我们贸然回收节点 C 可能导致非法访问，所以我们要等读者 0 和读者 1 都结束了，再去回收节点 C。

RCU 宽限期

如何知道读临界区什么时候开始，什么时候结束？

```
void rCU_reader() {           通知RCU，读者进临界区了  
    RCU_READ_START();  
  
    /* Reader Critical Section */  
  
    RCU_READ_STOP();          通知RCU，读者出临界区了  
}
```

可以使用不同的方式实现：如计数器

每个 reader 都会运行这两个东西，是一定会执行成功的。

同步原语对比：读写锁 vs RCU

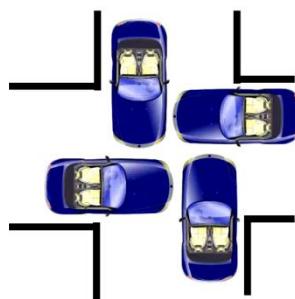
读写锁	RCU
相同点：	允许读者并行

不同点：	读写锁	RCU
	<ul style="list-style-type: none">读者也需要上读者锁关键路径上有额外开销方便使用可以选择对写者开销不大的读写锁	<ul style="list-style-type: none">读者无需上锁使用较繁琐写者开销大

对读者来说不需要上锁，但是使用和 Update 的时候，可能会比较复杂。

死锁

死锁



十字路口的“困境”

```
void proc_A(void) {  
    lock(A);  
    /* Time T1 */  
    lock(B);  
    /* Critical Section */  
    unlock(B);  
    unlock(A);  
}  
  
void proc_B(void) {  
    lock(B);  
    /* Time T1 */  
    lock(A);  
    /* Critical Section */  
    unlock(A);  
    unlock(B);  
}
```

T1时刻的死锁

死锁的原因：

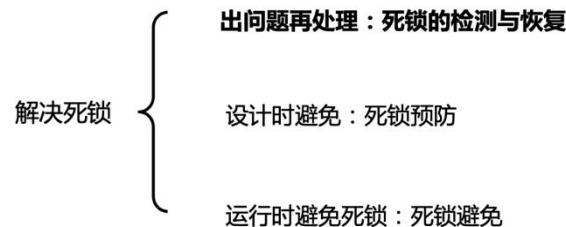
- 互斥访问：同一时刻只有一个线程能够访问。
- 持有并等待：一直持有一部分资源并等待另一部分，不会中途释放（如 proc_A 不会放锁）

A)。

- 资源非抢占：即 proc_B 不会抢 proc_A 已经持有的锁。
- 循环等待：A 等 B、B 等 A。

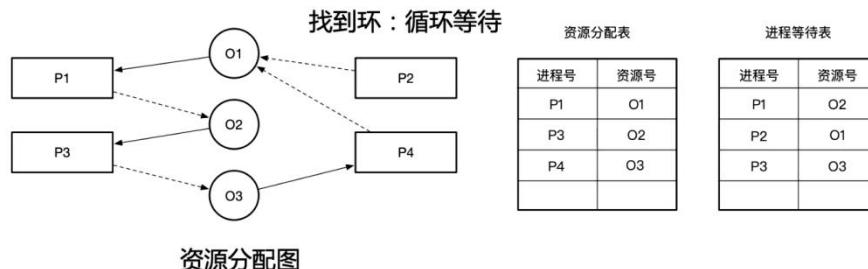
这四个条件是死锁的充要条件，我们要解决死锁的话，只需要打破其中的一个条件即可。

如何解决死锁？



分成运行时死锁检测与恢复、设计时死锁避免、运行时死锁避免。

检测死锁与恢复



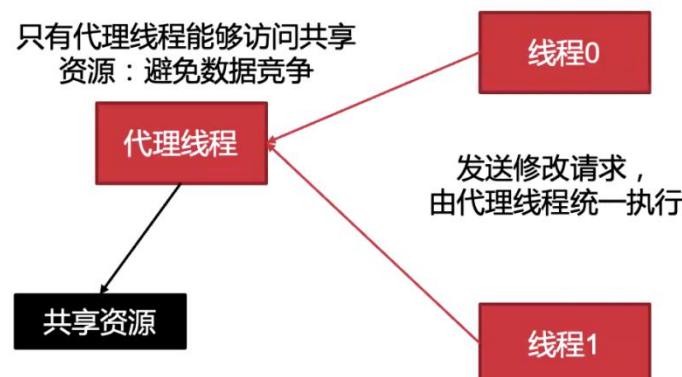
- 直接 kill 所有循环中的线程

- 如何恢复？打破循环等待！
- Kill 一个，看有没有环，有的话继续 kill
 - 全部回滚到之前的某一状态

问题：太浪费；公平性问题；有没有可能回滚（比如发了网络包）。

死锁预防：四个方向

- 1、避免互斥访问：通过其他手段（如代理执行）



*代理锁 (Delegation Lock) 实现了该功能

不要让每个线程去拿锁，只让代理线程去拿锁，这样只有一个人拿锁。

- 2、不允许持有并等待：一次性申请所有资源

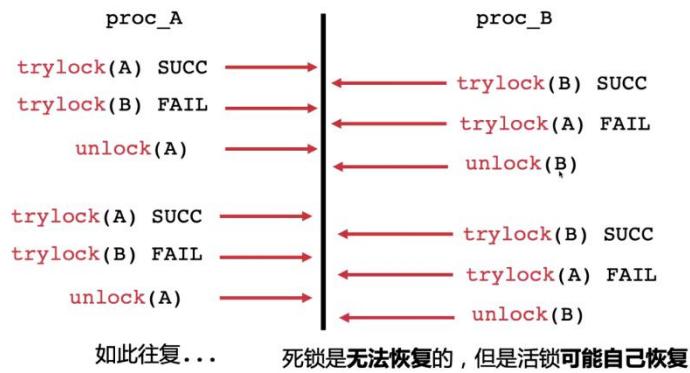
```
while (true) {
    if(trylock(A) == SUCC)
        if(trylock(B) == SUCC) {
            /* Critical Section */
            unlock(B);
            unlock(A);
            break;
        } else
            unlock(A); // 无法获取B，那么释放A
}
```

trylock非阻塞

立即返回成功或失败

一旦后者都失败了，把前者也释放掉。有了 trylock 以后就可以解决同时拿锁的问题。

避免死锁带来的活锁 Live Lock



死锁预防：四个方向

- 1、避免互斥访问：通过其他手段（如代理执行）
- 2、不允许持有并等待：一次性申请所有资源
- 3、资源允许抢占：需要考虑如何恢复
- 4、打破循环等待：按照特定顺序获取资源
 - 对所有资源进行编号
 - 让所有线程递增获取

```
void proc_A(void) {
    lock(A);
    /* Time T1 */
    lock(B);
    /* Critical Section */
    unlock(B);
    unlock(A);
}
```

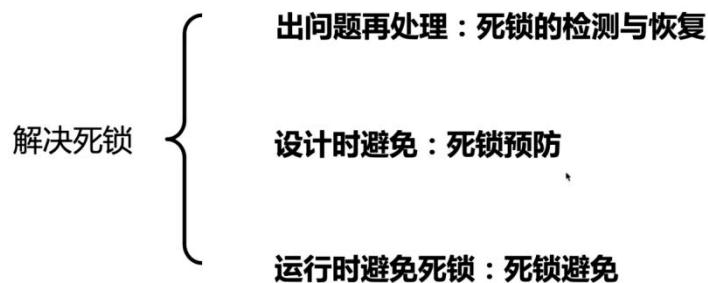
```
void proc_B(void) {
    lock(B);
    /* Time T1 */
    lock(A);
    /* Critical Section */
    unlock(A);
    unlock(B);
}
```

A : 1号 B : 2号：必须先拿锁A，再拿锁B

任意时刻：获取最大资源号的线程可以继续执行，然后释放资源

2022/3/24

回顾：如何解决死锁？



银行家算法

死锁避免：银行家算法

死锁避免：运行时检查是否会出现死锁

银行家算法的核心：

- 所有线程获取资源需要通过**管理者**同意
- 管理者**预演**会不会造成死锁
 - 如果会造成：阻塞线程，下次再给
 - 如果不会造成：给线程该资源

死锁避免的话，关键点在于要在运行时检查是否会出现死锁，如果会出现死锁那么我们不允许一些关键操作的执行。所以银行家算法的核心就是所有线程获取资源必须通过管理者的同意，管理者会预演一些看看操作会不会导致死锁。

死锁避免：银行家算法

如何预演判断？将系统划分为两个状态

对于一组线程 $\{P_1, P_2, \dots, P_n\}$ ：

- 安全状态

能找出至少一个执行序列，如 $P_2 \rightarrow P_1 \rightarrow P_5 \dots$ 让所有线程需求得到满足

- 非安全状态

不能找出这个序列，必定会导致死锁

安全性检查算法

银行家算法：保证系统一直处于安全状态，且按照这个序列执行

为了预演判断，我们需要分成两个状态。

银行家算法：安全性检查

四个数据结构：

M个资源 N个线程

- 全局可利用资源：Available[M]
- 每线程最大需求量：Max[N][M]
- 已分配资源：Allocation[N][M]
- 还需要的资源：Need[N][M]

我们定义了上述的四个数据结构。

例子：银行家算法安全性检查

安全序列：

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1	5	10	2	8	3	2		
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

某时刻系统状态

分配给能满足其全部需求的线程

max 就是线程为了往下执行要获得的资源，allocation 就是已分配的，减一下我们就得到了 need 的资源。此时我们发现，只有 P2 可以获得它全部需要的资源往下执行，所以我们先执行 P2，然后 P2 就可以还出它使用的所有资源。

安全序列：P2 ->

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1	5	10	2	8	3	2		
P2							3	2
P3	10	11	5	1	5	10		

模拟P2执行完成

分配给能满足其全部需求的线程

相同的，此时可以继续执行 P1，P1 执行完后释放出它使用的所有资源。

安全序列：P2 -> P1 ->

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1								
P2							5	10
P3	10	11	5	1	5	10		

模拟P1执行完成

分配给能满足其全部需求的线程

接下来，我们继续执行 P3 即可。

安全序列：P2 -> P1 -> P3

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1								
P2							10	11
P3								

模拟P3执行完成

分配给能满足其全部需求的线程

这种方式可以满足安全性状态要求，P2->P1->P3 这就是一个安全序列。并且对于这个例子来说，这个安全序列是唯一的。在最开始 P1 请求资源的时候，系统资源是不能满足它的，所以我们阻塞 P1 线程，让它处于一个安全的状态。

表 1 银行家算法安全性检查的例子

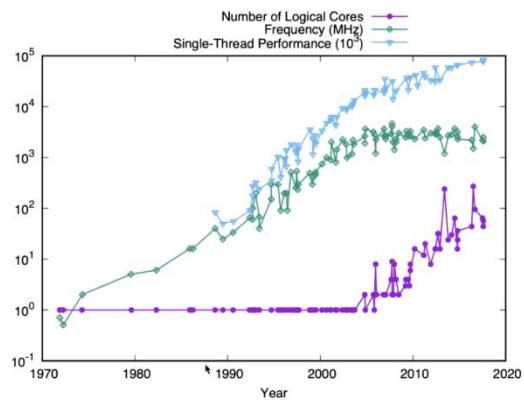
银行家算法的弊端：每个线程执行的时候需要多少资源，实际上是很难在静态的时候去判断的，所以银行家算法更多时候应用在一个封闭的场景下是可以使用的，比如资源是确定的嵌入式场景中。

多核和同步

接下来我们再介绍一下多核和同步的问题。在我们最早介绍的时候，我们介绍了多处理器和多核。

回顾：多处理器与多核

- 单核性能提升遇到瓶颈
- 不能通过一味提升频率来获得更好的性能
- 通过增加核心数来提升软件的性能
- 桌面/移动平台均向多核迈进



一开始我们只需要改进芯片工艺， $28\text{nm} \rightarrow 14\text{nm} \rightarrow 12\text{nm}$ 就可以得到性能和主频的提升。但是随着工艺红利的耗尽，因为有功耗和其他的因素，我们只能通过增加核心数来提升软件的性能。

回顾：多核不是免费的午餐



网图：多核的真相

假设现在需要建房子

工作量 = 1000人/年

工头找了10万人，需要多久？

面临的两个问题：

1. 工人多手杂，不听指挥，导致施工事故。（正确性问题）
2. 工具有限，大部分工人无事可干。（性能可扩展性问题）

但是多核并不一定带来性能的提升。比如上图中多核只有一个核在干活。

回顾：操作系统在多处理器多核环境下面临的问题

正确性保证

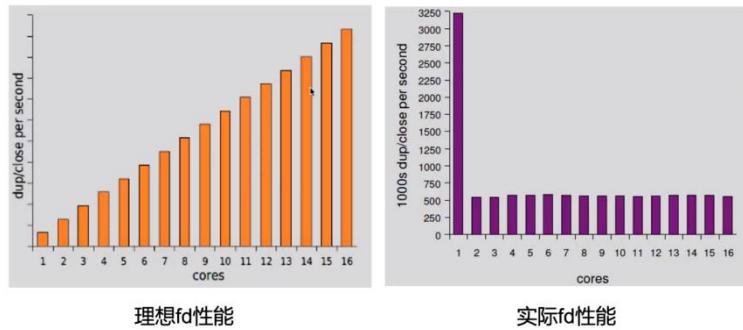
- 对共享资源的竞争导致错误
- 操作系统提供同步原语供开发者使用
- 使用同步原语带来的问题

性能保证

- 多核多处理器硬件与特性
- 可扩展性问题导致性能断崖
- 系统软件设计如何利用特性

我们之前保证的是正确性问题，主要是同步原语和同步原语带来的问题。我们必须理解多核才可以更好地使用多核和多处理器的处理能力。

多核下应用的性能表现：理想 vs 现实



这是我们 18 年的论文上的一个评测的数据，比如我们有 16 个线程分别去打开一个 file descriptor，这样理想情况下我们的性能应该是随着核的线性的增长，但是我们实际上评测出来的时候，我们发现一个核的性能数据是最好的，到了多个核之后的结果都不好。

Amdahl's Law

这样，我们就来看一下并行计算的理论加速到底是什么样子的。

并行计算理论加速比（理想上限）

Amdahl's Law

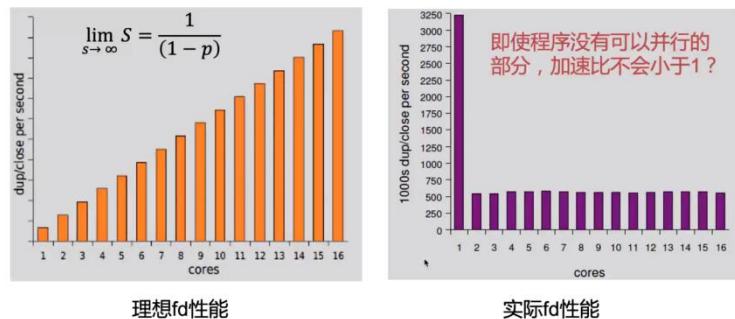
$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

加速比 同时执行的核心数
可以并行部分代码占比
当核心数增加时...
 $\lim_{s \rightarrow \infty} S = \frac{1}{1-p}$

可以并行部分占比越多，这个程序理论上最大加速比越大

Amdahl's Law 就是我们并行加速的公式，这个公式也非常简单。串行的代码肯定是不能做并行的，必须串行来执行，其他并行的部分可以平摊到 s 个核上。这样当核的数量增加的

时候，加速比贴近 $\frac{1}{(1-p)}$ 。



我们继续回到这个例子，理想情况下按照我们的公式来算的话，增加核以后，性能总会

有一些提升，这和我们的实验结论相违背。哪怕我们程序没有可并行的部分，那加速比也不应该小于 1 才对。那我们就必须来理解硬件是什么样子的。

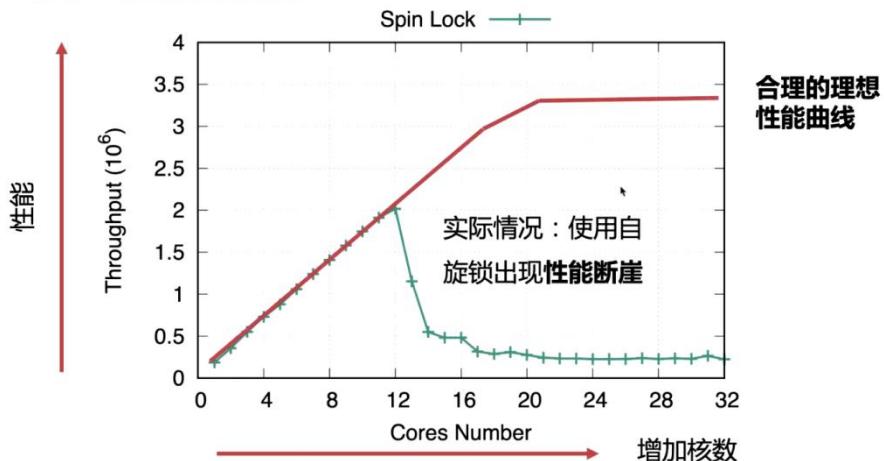
互斥锁微基准测试

使用微基准测试来复现这个现象

```
struct lock *glock;
unsigned long gcnt = 0;
char shared_data[CACHE_LINE_SIZE];
void *thread_routine(void *arg) {
    while(1) {
        lock(glock);
        /* Critical Section */
        gcnt = gcnt + 1;
        /* Read Modify Write 1 * shared cacheline */
        visit_shared_data(shared_data, 1);
        unlock(glock);
        interval();
    }
}
```

上述的例子可以用这个代码进行模拟。我们有一个锁叫做 `glock`, `gcnt` 是一个共享的变量。还有一个 `shared_data` 就是一个共享的缓存，一般是 64 Byte。`thread_routine` 这个函数主要是在做 `while (1)` 的无限循环，里面就是获取到了 `glock` 以后对 `gcnt` 加一。这和我们的 `file descriptor` 是比较类似的，我们可以认为是一个共享的数据。`interval` 就是等一段时间。如果每个线程都执行这个代码的话，我们可以通过这个代码来复现一下我们刚才碰到的情况。

可扩展性断崖



这个程序执行的结果比刚才的例子要好一点，在这个实验中从 12 个核开始，性能才开始有断崖式的下降。增加核理论上应该提升性能，但是这里出现了断崖下降。这就需要我们对锁的实现机制和多核结构有更深入的理解。十年前我们的合作者写了一篇文章：

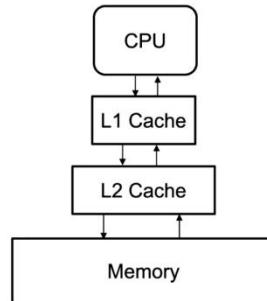
Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." Proceedings of the Linux Symposium. 2012.

也就是不可扩展的锁是很危险的，危险就是体现在让我们的性能在一些情况下会出现断崖式的下降。出现这种情况的原因就需要我们对多核架构有一些理解。

多核环境下的缓存

高速缓存 (cache) 回顾

- 多级缓存：
 - 靠近CPU贵，速度快，容量小
 - 远离CPU便宜，速度慢，容量大
- 读操作：
 - 逐层向下找
 - 没找到从内存中读取，放到缓存中
- 写操作：
 - 直写/写回策略
 - 写入高速缓存，替换时写回



我们首先来回顾一下我们的高速缓存，现代处理器都是一个多级缓存的。靠近 CPU 的比较贵、速度比较快、容量比较小，远离 CPU 的比较便宜、速度比较慢、容量比较大。读操作是从 CPU 逐层往下找的，写操作的话分为直写（write through）和写回（write back）操作。

Q：在多处理器的场景下，缓存到底是什么样子的呢？

A：一种简单的方案就是多个核共享同一个 L1 Cache、L2 Cache 和 Memory：

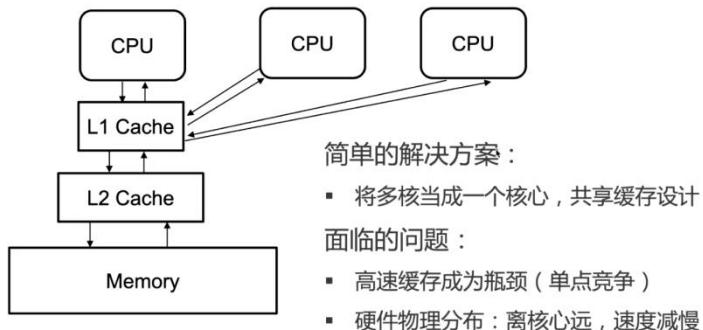


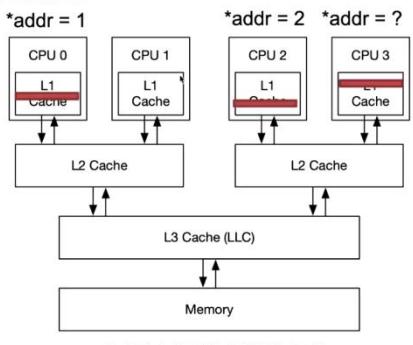
图 1 多处理器多核环境中的缓存结构

这样高速缓存就会成为瓶颈，引起单点竞争，不能发挥出多核的效果。另外一方面，我们硬件物理分布也会带来一定的影响。虽然可能是以光速的互联，但是还是离核越远速度就会越慢，比如上图中的第三个 CPU 的访问速度就会相对比较慢。

所以一个比较合理的多核环境中的缓存结构是多级缓存+私有缓存的方法。

多核环境中的缓存结构

- 多级缓存：
 - 每个核心有自己的**私有**高速缓存 (L1 Cache)
 - 多个核心共享一个**二级**高速缓存 (L2 Cache)
 - 所有核心共享一个**最末级**高速缓存 (LLC)
- 非一致缓存访问 (NUCA)
- 数据一致性问题



一个典型多核系统高速缓存架构*

*可以有其他选择，大部分多核系统采用该架构

这是一个多级的层次化的结构，每个核会有一个自己私有的 L1 Cache，多个核共享一个 L2 Cache，所有核共享一个最末级的高速缓存（LLC, Last-level Cache）。我们的缓存结构是一个层级化的结构，这就带来了非一致缓存访问（NUMA）的问题，也就是不同核访问不同 Cache 的速度是不一致的；还会带来数据一致性的问题，对于同一个地址可能多个核都会访问。

缓存一致性

- 保证不同核心对同一地址的值达成共识
- 多种缓存一致性协议：窥探式/目录式缓存一致性协议

具体怎么做？

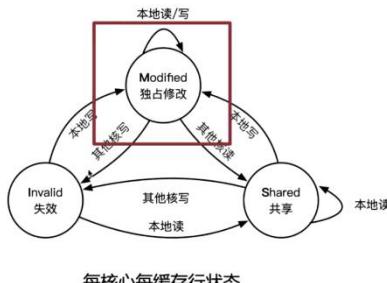
- 缓存行处于不同状态（MSI状态）
- 不同状态之间迁移
- 所有读/写缓存行操作遵循协议流程

缓存一致性主要是保存不同的 CPU 核对同一个地址的值要达成一个共识，这就需要缓存一致性的协议，分为窥探式（snooping）的和目录式的（directory）。

缓存会在不同状态间迁移，读写缓存行都遵循协议的流程。

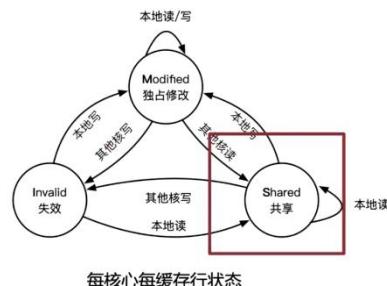
缓存一致性：MSI状态迁移

- 独占修改（Modified）
 - 该核心独占拥有缓存行
 - 本地可读可写
 - 其他核读需要迁移到共享
 - 其他核写需要迁移到失效



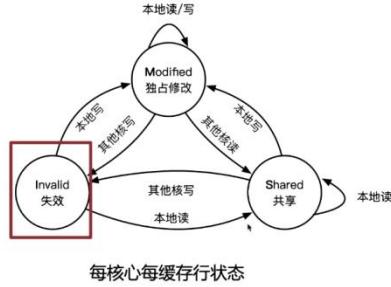
我们通过 MSI 这个状态机来进行操作。Modified 代表是独占的修改，Shared 就是共享读，Invalid 就是失效。在 Modified 中，该核独占此缓存行，可读可写。但是此时如果其他核也想读，那么状态就转移到共享状态

- 共享（Shared）
 - 可能多个核同时有缓存行的拷贝
 - 本地可读
 - 本地写需要迁移到独占修改，并使其他核该缓存行失效
 - 其他核写需要迁移到失效



对于 Shared 状态来说，本地是可以读的，如果本地要写的话，我们必须迁移到 Modified 状态，并且修改其他核的状态为 Invalid。

- 失效 (Invalid)
 - 本地缓存行失效
 - 本地不能读/写缓存行
 - 本地读需要迁移到共享，并使其他核心该缓存行迁移到共享
 - 本地写需要迁移到独占修改，并使其他核心该缓存行失效



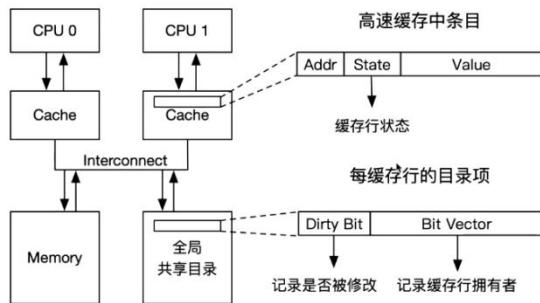
在 Invalid 状态中，本地 CPU 即不能读也不能写这个 Cache Line。如果要本地读的话，需要迁移到 Shared 状态，这也会使得其他核的此 Cache Line 也迁移到 Shared。

这就是我们的 MSI 三种状态的变化。

缓存一致性：全局目录项

Q: 我们怎么通知其他核，需要迁移缓存行状态呢？

A: 我们有全局目录项的实现。全局目录项负责记录缓存行在不同核上的状态，通过总线进行通讯。

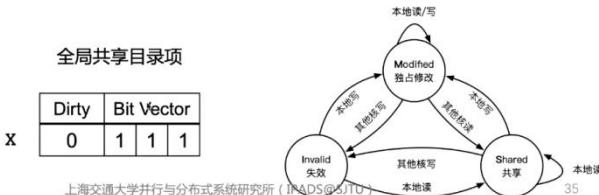


我们可以看到每个 Cache Line 里有一个 State，标志了缓存行的状态是 M、S 还是 I。我们可以通过缓存行的目录项来记录每个缓存行对应的状态是什么样的。比如缓存行是否被修改、拥有者到底是谁。对于 Bit Vector，我们有几个核，就会有几个 bit。目录项有时候会放在内存里，或者放在 LLC 里。

例子：MSI 的状态迁移

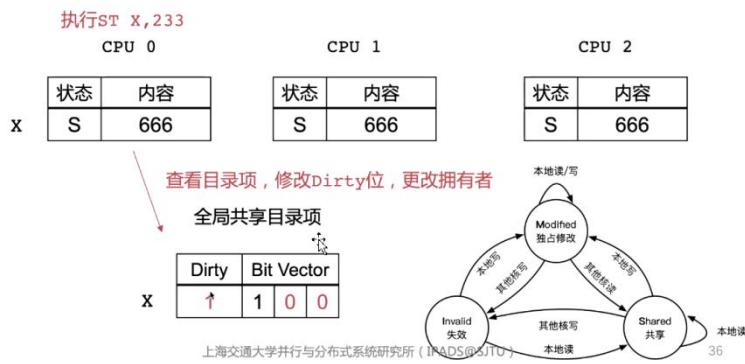
关注变量x所在缓存行，3个CPU，一个全局共享目录

	CPU 0	CPU 1	CPU 2
x	状态	内容	状态
	S	666	S
			S

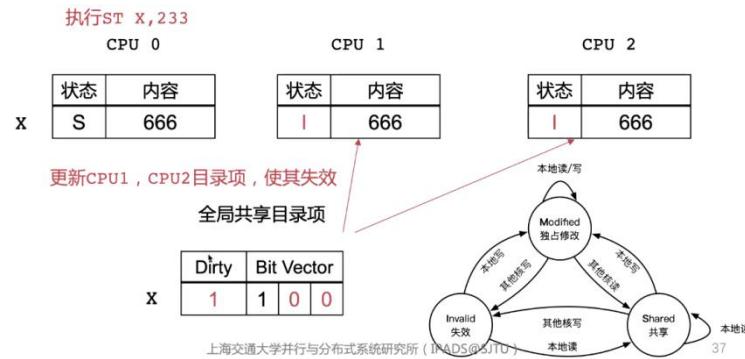


全局共享目录， x 的 dirty 是 0 的。然后三个 CPU 里 x 的 Cache Line 都是处于 Shared 的状态。

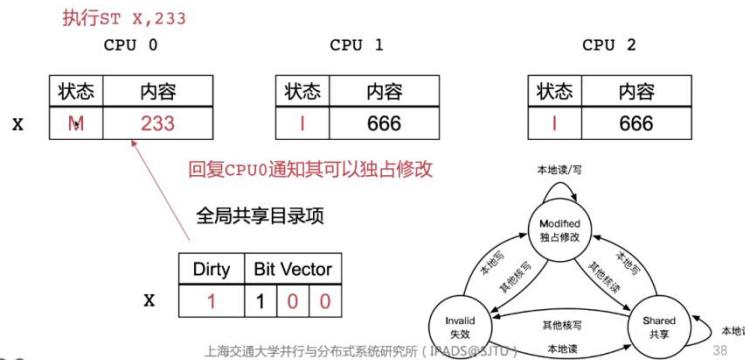
假设此时 CPU0 想把 x 的值修改为 233。它首先查看目录项，修改 dirty 位，再更改拥有者。



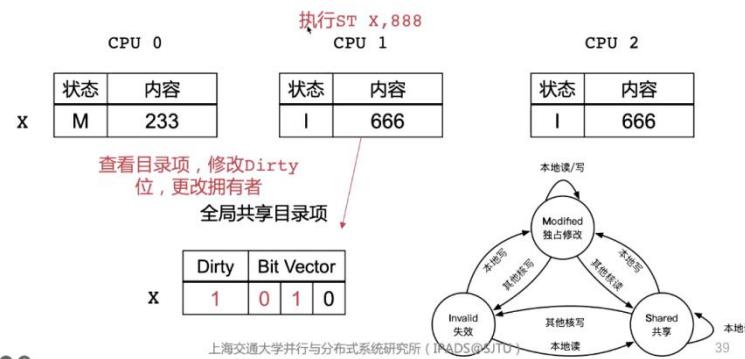
同时，我们要把 CPU1 和 CPU2 的对应的 Cache Line 标志成 Invalid 状态。



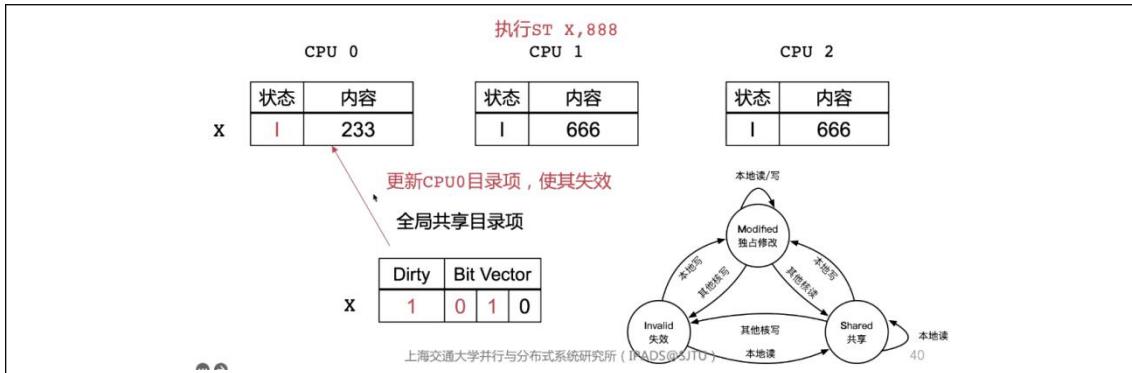
接下来，我们就可以通知 CPU0 可以独占修改了。



我们看到这就是在 Shared 状态中进行本地写的状态。

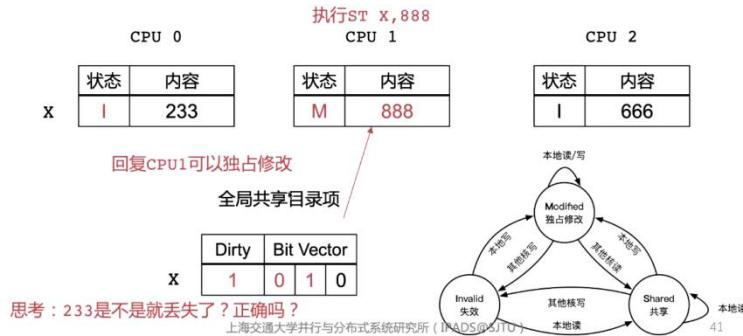


此时，CPU1 尝试对 X 进行本地写。对照状态机我们可以发现，要从 Invalid 状态变为 Modified 状态。



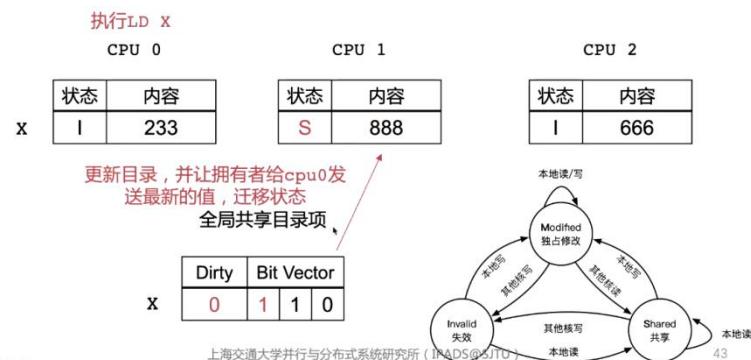
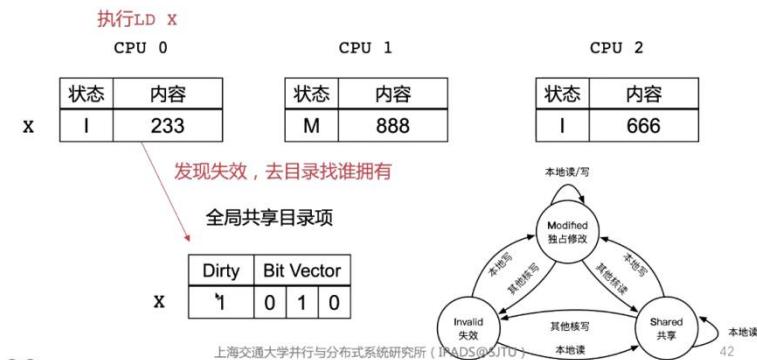
Q: 在之前 CPU0 写 666 的时候，我们需要把 CPU1 和 CPU2 全部通知一遍，但这里我们为什么只通知了 CPU0 呢？

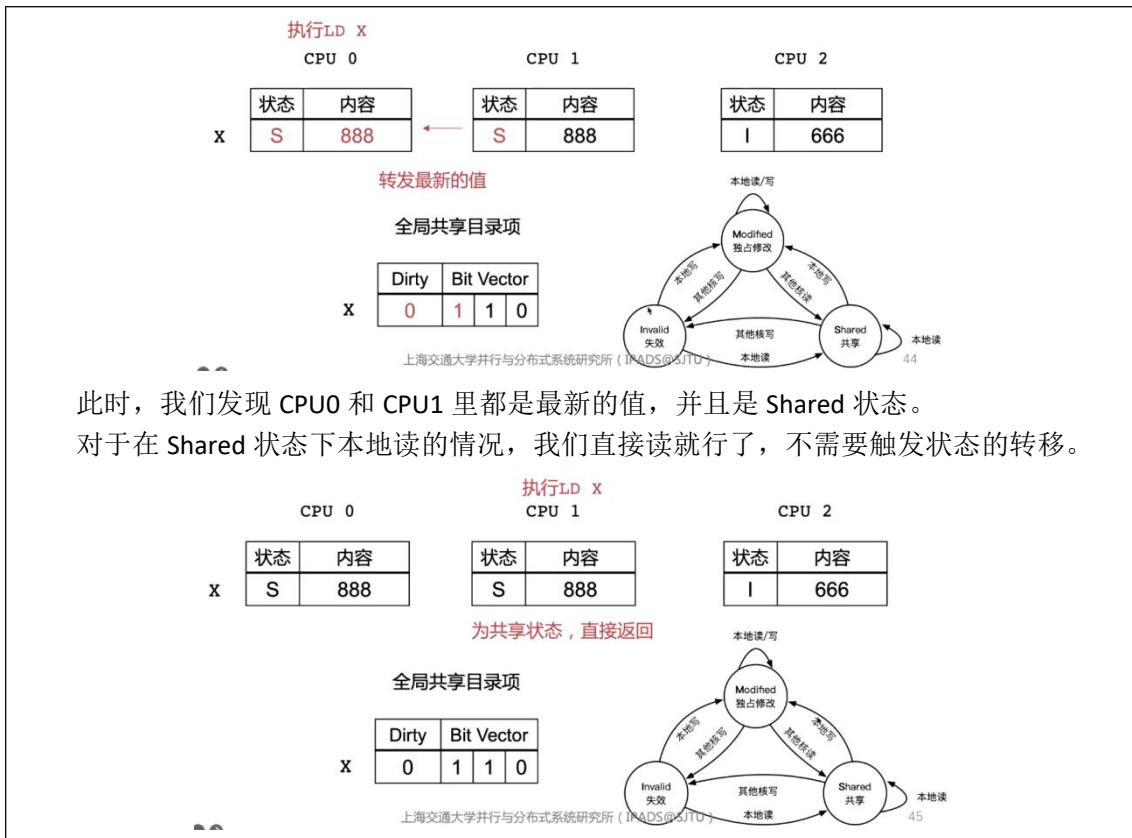
A: 因为我们有 Bit Vector 在，在写之前 Bit Vector 是 1 0 0，所以我们只需要通知 CPU0。这就是目录项的好处。比如我们有 100 个 CPU，但是只有 2、3 个 CPU 去操作数据，那么我们就不需要通知所有的 CPU 这个数据被修改了。



Q: 我们修改为 888 后，233 是不是就丢失了？

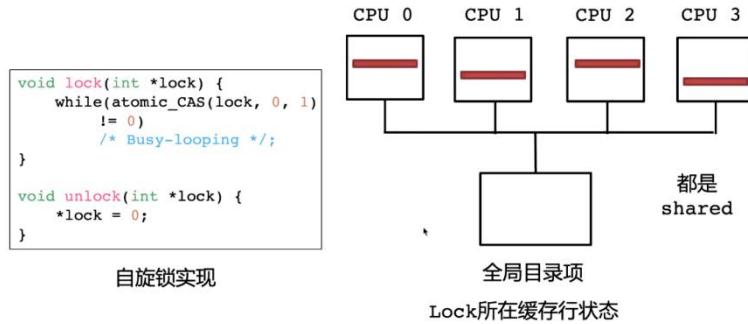
A: 是的。因为 CPU1 在 CPU0 后进行了修改，我们必须让所有 CPU 核看到的是最新的值。





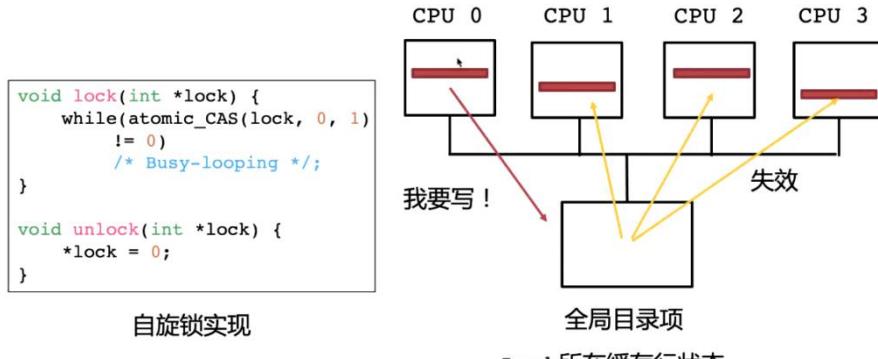
可扩展性断崖

回到可扩展性断崖

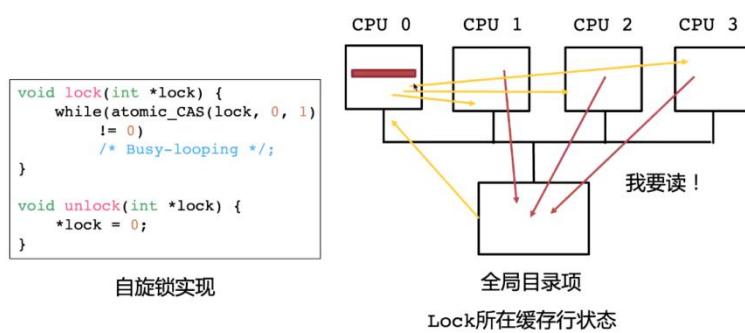


那么我们回过头来看最早的 spin lock 的实现, 我们介绍过 Compare_and_Swap 这个实现。它的实现是比较简洁的。在等锁的时候, 缓存行都是 Shared 的状态。

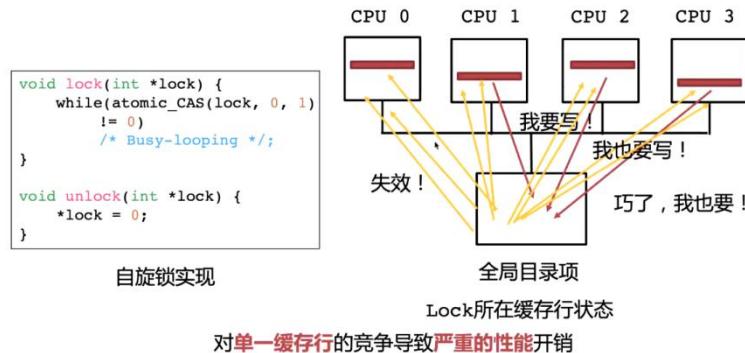
可扩展性断崖背后的原因



如果 CPU0 获得了这个锁，要去写它，实际上就是从 Shared 状态要进入 Modified 状态，这样我们就要让所有的核都 Invalid。



一旦写完了，其他核又要 Spin 去读这个 Lock 的值，所以所有核又要从 Invalid 状态到 Shared 状态。



这个过程会来回往复。

Amdahl's Law

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

加速比 可以并行部分代码占比
当核心数增加时... 同时执行的核心数

$$\lim_{s \rightarrow \infty} S = \frac{1}{(1-p)}$$

对单一缓存行的竞争导致严重的性能开销：公式中的 p 急剧下降，加速比下降

当核心数量增加的时候，公式中的 P 是在急剧下降的，因为所有核都在同一个核要读数据，所以这部分都是串行的，所以加速比下降了。

解决可扩展性问题：回退锁

回退锁（backoff lock）

如何解决可扩展性问题

Simple fix：避免对单一缓存行的高度竞争 – Back-off（回退）策略

```
void back_off(int time) {
    for (volatile int i=0; i<time; i++)
        cpu_relax();
}

void lock(int *lock) {
    while(atomic_CAS(lock, 0, 1) != 0)
        back_off(DEFAULT_TIME);
}
```

使用Back-off策略

思考：这样写能解决问题吗？
会有什么样的问题？

等待相同时间，同时停止等待，
同时开始下一轮竞争！

- 随机时间
- 指数后退

类似于错峰出行，避免大家同时去抢这个锁。

Q：这样子写能不能解决问题？

A：大家还是每过一段时间在同一个时间里去抢，还是会导至拥挤。

Back-off 是否完全解决可扩展性问题

Linus' Response

So I claim:

真实硬件/场景很难触发

- it's *really* hard to trigger in real loads on common hardware.
- if it does trigger in any half-way reasonably common setup
(hardware/software), we most likely should work really hard at fixing
the underlying problem, not the symptoms. 治标不治本
- we absolutely should *not* pessimize the common case for this

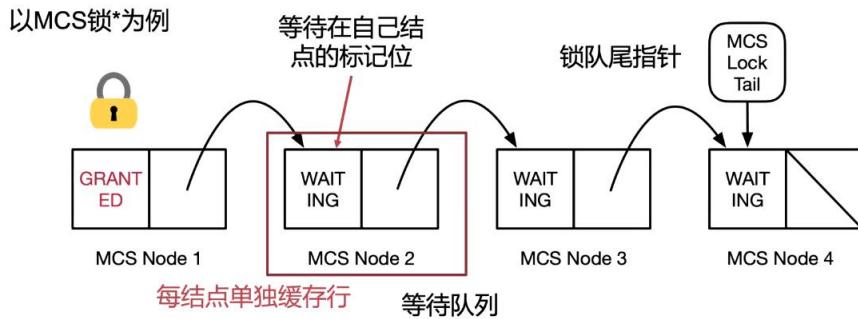
*<http://linux-kernel.2935.n7.nabble.com/PATCH-v5-0-5-x86-smp-make-ticket-spinlock-proportional-backoff-with-auto-tuning-td596698i20.html>

解决可扩展性问题：MCS 锁

MCS 锁是两个作者的名字来命名的。这个论文是 1991 年的。这个锁的实现核心在于，我们发现 Spin Lock 的实现竞争同一个 Cache Line，所以我们的目标就是避免在关键路径上出现对同一个 Cache Line 的竞争。

如何解决可扩展性问题：MCS锁

核心思路：在关键路径上避免对单一缓存行的高度竞争



在 Spin Lock 中，大家都在外面等着进食堂吃饭，每次只放一个人进去，人都堵在门口。Cache Line 就是每次决定的放进去的人，一旦有个人出来，所有人都冲上去问食堂的阿姨“现在能不能进去”。这个时候，出现的问题就是食堂阿姨就崩溃了，并且人越多她越崩溃。只有几个人的时候她还能处理。

我们刚才讲的 back-off 的机制就是，大家可以先去打打球做做别的，错峰出行。

可扩展的方式就是排队，这样我们不需要问食堂阿姨能不能进去，我们只需要记住我们的前面一个人是谁就可以了。前面一个人做完了就会来通知你可以进去了，这样我们就不需要问阿姨了。这个等待就不会有竞争，有且只有一个人通知你能不能进食堂。而且有队列的好处就是，我们可以看看队列的长度，如果很长的情况下，我们可以先去做别的事情。这样可以避免对单一 Cache Line 的高度竞争。

MCS 锁中，每个节点会指向下一个节点，每个 Spin 在自己的 waiting bit 里，如果变成 granted 了就可以进去了。这样就不用共享一个 Cache Line。

```
1. struct MCS_node {  
2.     volatile int flag;  
3.     volatile struct MCS_node *next;  
4. } __attribute__((aligned(CACHELINE_SZ)));  
5.  
6. struct MCS_lock {  
7.     struct MCS_node *tail;  
8. };
```

整体来看，我们可以先看它的数据结构。一个 MCS 节点首先有 flag 标志位，它要么是 WAITING，要么是 GRANTED，每个节点有 next 指针。每个线程都会有这样的一个结构。

MCS_lock 其实就是一个单向等待队列。

```
1. void *XCHG(void **addr, void *new_value) {  
2.     void *tmp = *addr;  
3.     *addr = new_value;  
4.     return tmp;  
5. }  
6.  
7. void lock(struct MCS_lock *lock, struct MCS_node *me) {  
8.     struct MCS_node *tail = 0;  
9.     me->next = NULL;
```

```

10.     me->flag = WAITING;
11.     tail = atomic_XCHG(&lock->tail, me);
12.     if (tail) {
13.         tail->next = me;
14.         while (me->flag != GRANTED);
15.     }
16. }
```

在 lock 的时候，我们先传入锁是谁以及当前的线程。它做的就是一个 atomic_exchange(atomic_XCHG) 交换 lock->tail 和 me，返回的是 lock->tail 的值。

如果 tail 为空，那么就说明原先的锁是一个空的锁，没有人在里面，那么我们就可以直接拥有这个锁了；如果 tail 存在，那么说明已经有人持有锁了，那么我们设置 tail->next 为自己，等待上一个人唤醒我们（置 me->flag 为 GRANTED）。因为我们是等自己的 flag，所以不会出现 cache 的高度竞争。

```

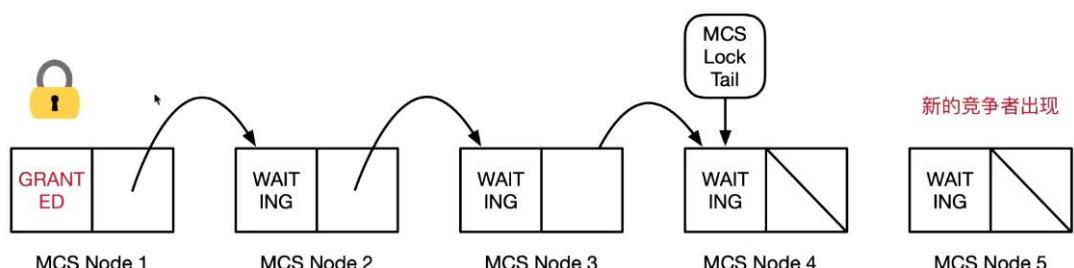
1. void unlock(struct MCS_lock *lock, struct MCS_node *me) {
2.     if (!me->next) {
3.         if (atomic_CAS(&lock->tail, me, 0) == me) return;
4.         while (!me->next);
5.     }
6.     me->next->flag = GRANTED;
7. }
```

unlock 的时候，首先我们看看后面是否还有人在等待。如果后面没有人等待的话，并且我们成功放锁了，那就直接返回。lock->tail 和 me 不相等的情况意味着此时又有人尝试拿锁了，那么我们要等待它修改了 tail->next（lock 代码中的第 13 行），并且再把锁给到它。

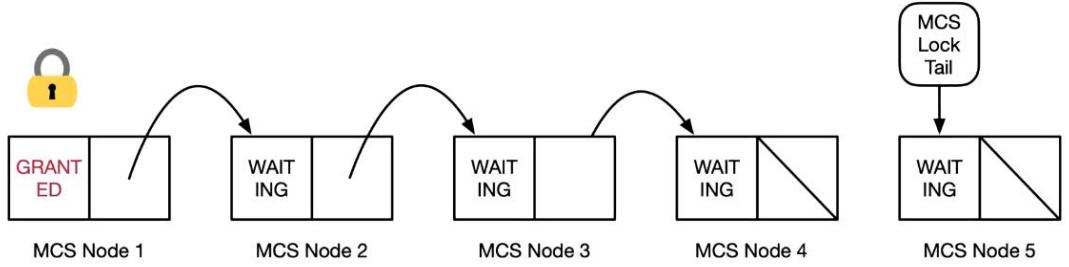
我们再来看一个具体的例子：

例子：新的竞争者加入等待队列

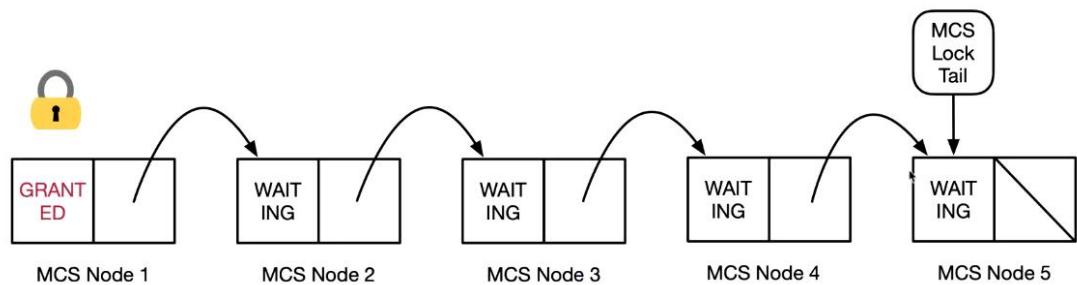
此时队列中有 4 个元素，出现了一个节点 5，它会先把自己置成 WAITING。



先填写自己结点的内容

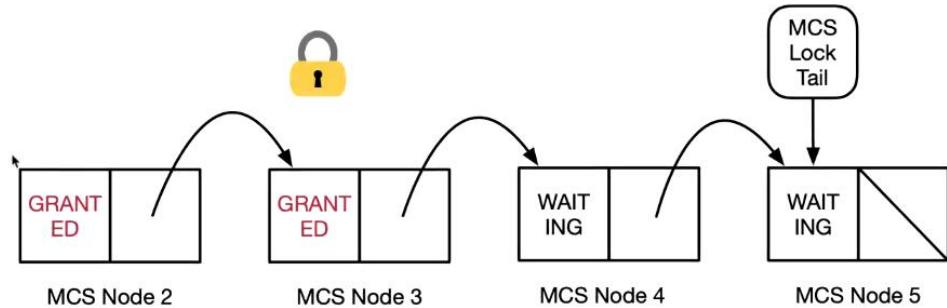
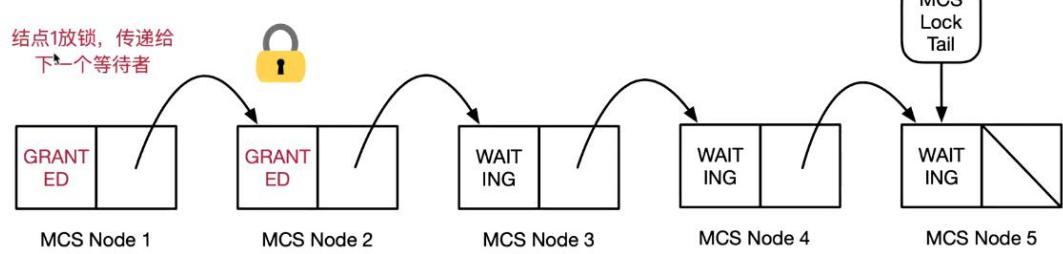


通过原子操作更新MCS锁的尾指针



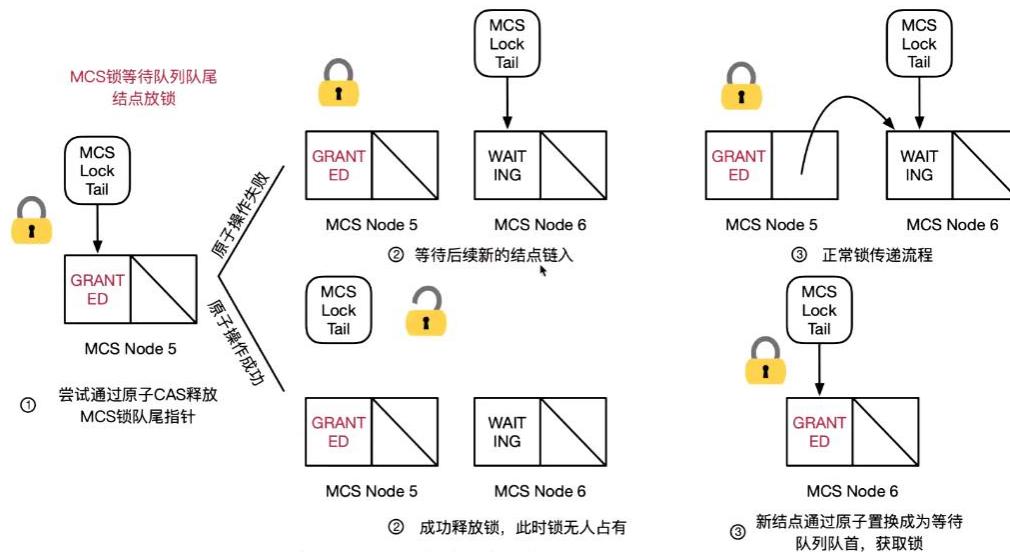
最后链接入等待队列

在 UNLOCK 的时候，我们就去找一下是否存在下一个节点，只需要置为 GRANTED 即可。



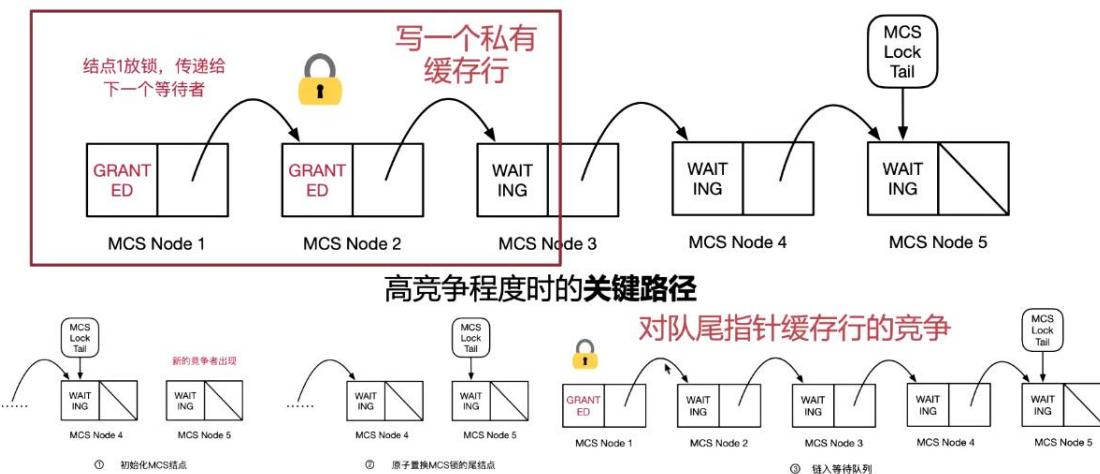
下面就是刚才讲的比较特殊的情况，如果尝试原子操作时，我们已经修改了 MCS LOCK Tail，那么我们就需要等待正常链入以后再把锁传递给新加入的节点。

MCS锁：放锁流程



MCS锁：性能分析

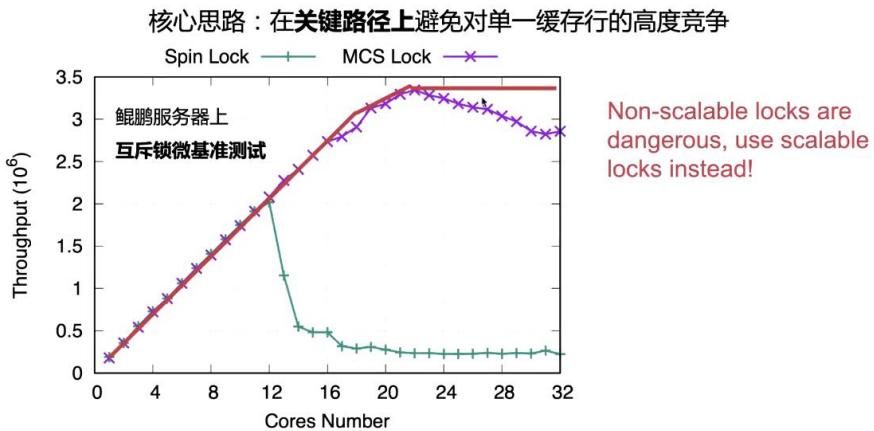
不再会高频竞争全局缓存行



高竞争程度时关键路径之外

它首先是写一个私有的 Cache Line，它不会高频竞争全局的 Cache Line，所有操作都是在队尾缓存行进行操作，所以它的可扩展性是很好的。

MCS锁：性能分析

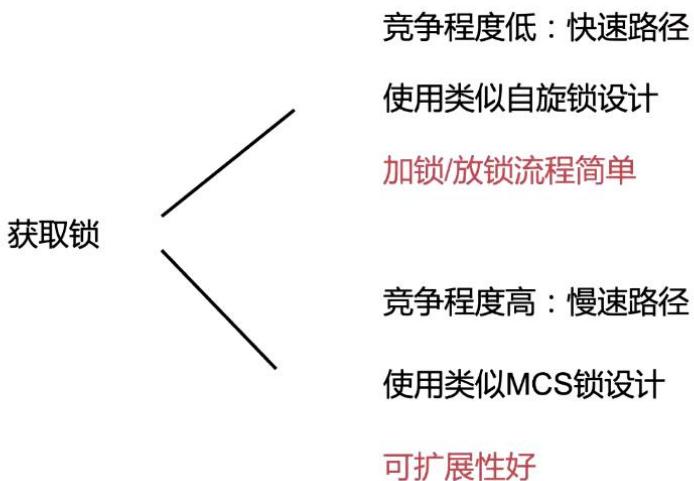


在我们的鲲鹏服务器上，性能还是比较接近我们理想的值的。对于扩展的场景的话，我们要使用可扩展的锁。

Q: 那么 MCS 锁有什么问题呢？

A: 对空间的占用会变大。我们每加一个节点都需要 cache line，可能还会出现 malloc 操作。在没有发生严重竞争的情况下，性能的 overhead 是很大的。

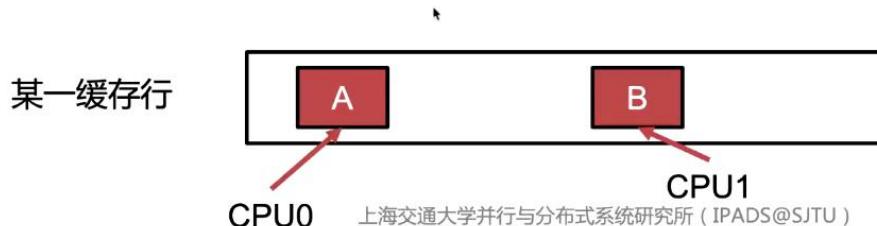
Linux Kernel中的可扩展锁：QSpinlock*



所以在 Linux 内核中，一开始是 Spin Lock，后来的 QSpinLock 就实现了一个类似于 MCS Lock 的排队锁的方法，分为快速路径和慢速路径。

系统软件开发者视角下的缓存一致性

- 多核处理器中面对私有高速缓存硬件提供的正确性设计
- 对软件开发者透明
- 系统软件开发者视角：
 1. 多个核心对于同一缓存行的高频竞争将会面临严重的性能开销
 2. 虚假共享 (False Sharing) 在多核情况下是致命的



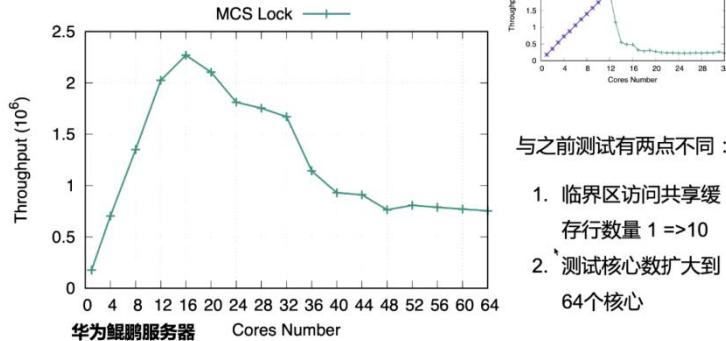
非一致内存访问 (NUMA)

互斥锁微基准测试

```
struct lock *glock;
unsigned long gcnt = 0;
char shared_data[CACHE_LINE_SIZE * 10];
void *thread_routine(void *arg) {
    while(1) {
        lock(glock);
        /* Critical Section */
        gcnt = gcnt + 1;
        /* Read Modify Write 10 * shared cacheline */
        visit_shared_data(shared_data, 10);
        unlock(glock);
        interval();
    }
}
```

我们还是回到互斥锁的微基准的测试程序上，在临界区访问更多缓存行，比如我们一次访问 10 个 Cache Line，会不会影响 mcs 的扩展性呢？

MCS锁在NUMA上的表现

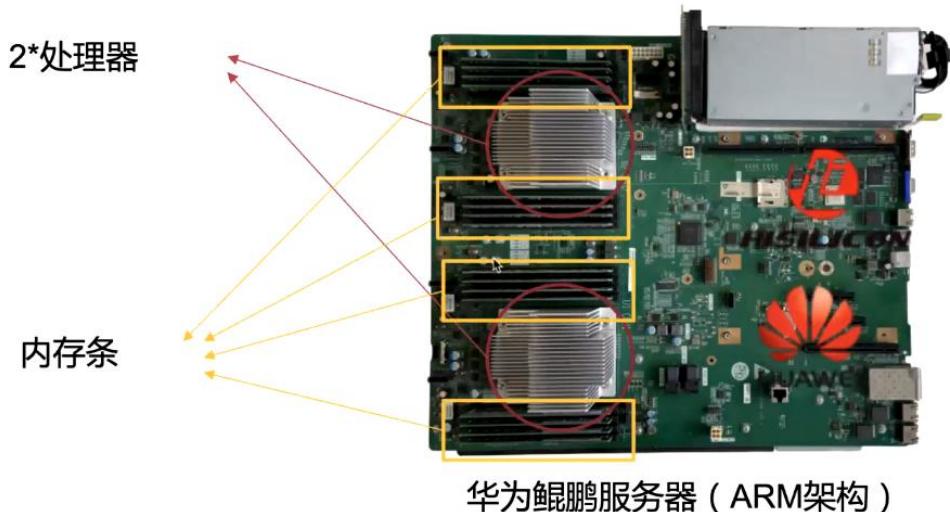


与之前测试有两点不同：

1. 临界区访问共享缓存行数量 $1 \Rightarrow 10$
2. 测试核心数扩大到 64个核心

每个 socket 上都有几个 CPU 核，在服务器的板子上面，距离越远，访问时间越长。我们可以看到在 16 个核的时候性能还是不错的，但是超过 16 个核之后性能就会下降。因为是 4 路的，每路 16 个核，跨了一个 socket 后性能就开始下降了。

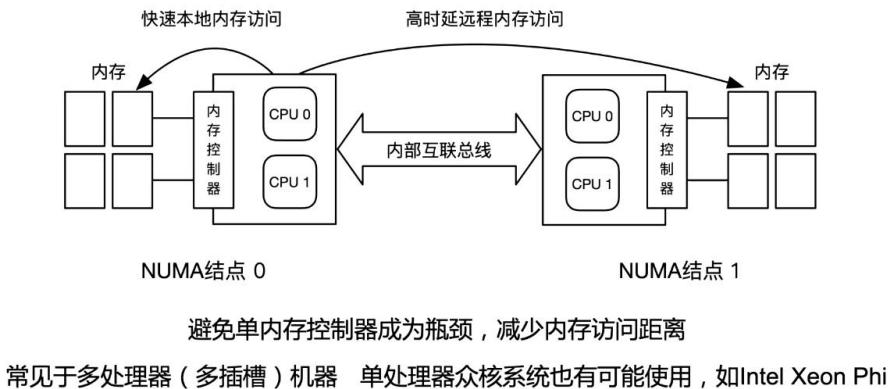
鲲鹏服务器



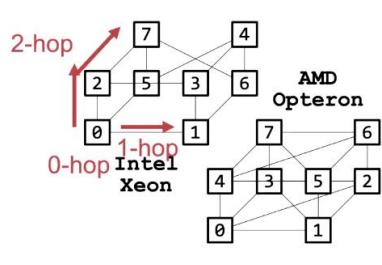
常见于多处理器 (多插槽) 机器

每个处理器上插了几块内存，所以如果处理器 2 上的核要访问处理器 1 上的内存的时候，走线就会更长，时间就会变得更慢，所以这就是非一致内存访问 (NUMA)。

非一致内存访问



▶ Intel与AMD的NUMA系统架构与特性



Inst.	0-hop	1-hop	2-hop
80-core Intel Xeon machine			
Load	117	271	372
Store	108	304	409
64-core AMD Opteron machine			
Load	228	419	498
Store	256	463	544

Intel与AMD多插槽NUMA架构
结构复杂

Intel与AMD NUMA访存时延特性
跳数(hop)越多，延迟越高

这是我们当时的一些评测，还有一跳、两条等情况。形成了一个空间的拓扑结构。

Intel与AMD的NUMA系统架构与特性

Access	0-hop	1-hop	2-hop	Interleaved
80-core Intel Xeon machine				
Sequential	3207	2455	2101	2333
Random	720	348	307	344
64-core AMD Opteron machine				
Sequential	3241	2806/2406	1997	2509
Random	533	509/487	415	466

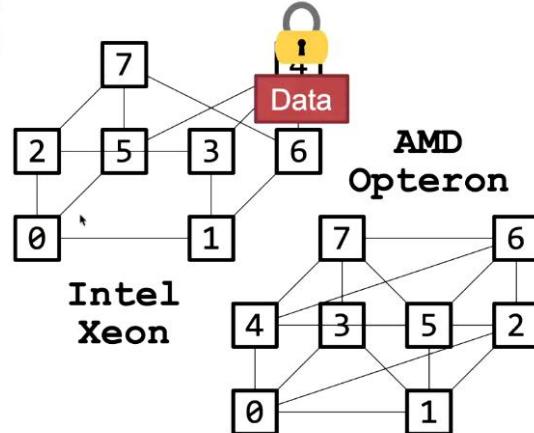
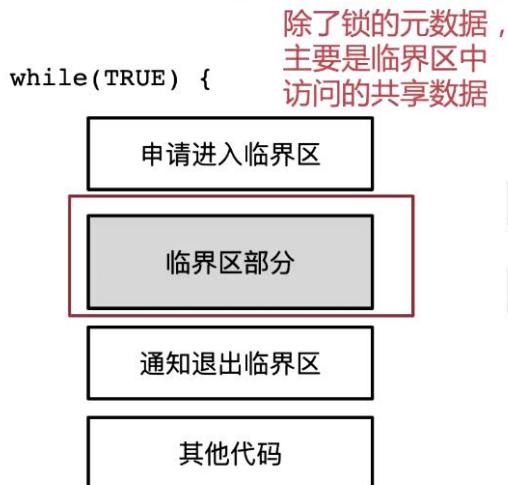
Intel与AMD NUMA访存带宽特性 (MB/s)

跳数越多，带宽受限

顺序访问有预取操作，带宽会好一点。而且跳数越多，带宽越受限。

NUMA环境中新的挑战

Challenge: 锁不知道临界区中需要访问的内容！



} 即使在cc-NUMA中没有出现缓存失效 跨结点的缓存一致性协议开销巨大

在 NUMA 环境里实现锁就会有新的挑战。挑战在于锁并不知道临界区里访问的内容在哪，有可能会跨我们的节点。因为我们有缓存一致性的保证，所以缓存不会失效，但是跨节点的缓存一致性的协议的开销会更大。

鲲鹏服务器的NUMA结构

```
$ numactl --hardware  
available: 4 nodes (0-3)  
node 0 cpus: 0 1 2 3 4 5 6 7 8 9  
10 11 12 13 14 15  
node 1 cpus: 16 17 18 19 20 21 22  
23 24 25 26 27 28 29 30 31  
node 2 cpus: 32 33 34 35 36 37 38  
39 40 41 42 43 44 45 46 47  
node 3 cpus: 48 49 50 51 52 53 54  
55 56 57 58 59 60 61 62 63  
node distances:  
node 0 1 2 3  
0: 10 15 20 20  
1: 15 10 20 20  
2: 20 20 10 15  
3: 20 20 15 10
```

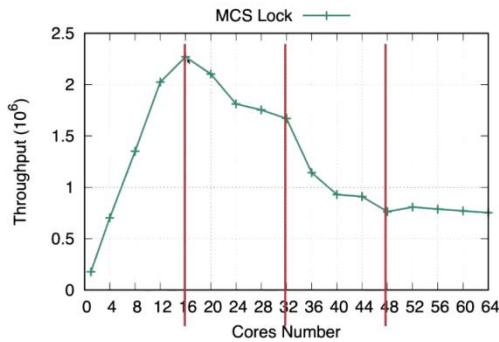


2022/3/29

同步原语：进阶与案例

我们上节课给大家简单看了一下鲲鹏处理器的结构是什么样的。我们可以看到它每个节点各有 16 个核，对 distance 也做了一个评估。所以理解了 NUMA 处理器的结构之后，相比 0 到 0,0 到其他节点的距离是有增加的。

MCS锁可扩展性



```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
6 7 8 9 10 11 12 13 14
15
node 1 cpus: 16 17 18 19
20 21 22 23 24 25 26 27
28 29 30 31
node 2 cpus: 32 33 34 35
36 37 38 39 40 41 42 43
44 45 46 47
node 3 cpus: 48 49 50 51
52 53 54 55 56 57 58 59
60 61 62 63
node distances:
node   0   1   2   3
0:    10   15   20   20
1:    15   10   20   20
2:    20   20   10   15
3:    20   20   15   10
```

所以我们每隔 16 个核（1 个节点）就会出现一个下降。所以如果我们理解了 NUMA 的拓扑结构，我们就可以更好地理解可扩展的 MCS Lock 的 NUMA 的环境下性能表现分别是怎样的。

cohort 锁

NUMA-aware设计：以cohort锁*为例

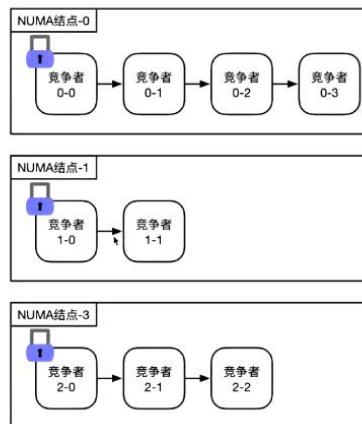
核心思路：在一段时间内将访存限制在本地

先获取每节点本地锁

再获取全局锁

成功获取全局锁
释放时将其传递给
本地等待队列的下一位

全局锁在一段时间内
只在一个结点内部传递

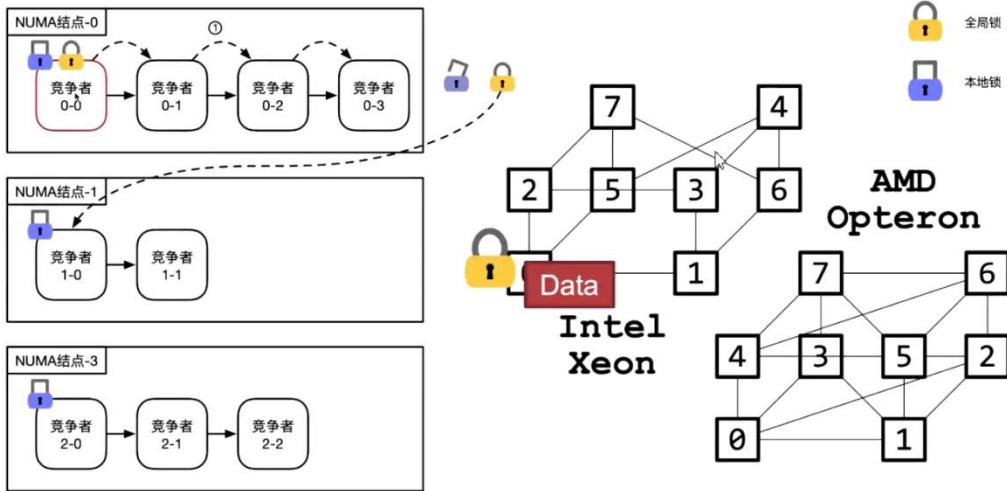


我们知道 NUMA 长这个样子以后，设计锁的时候就需要考虑 NUMA 的架构。我们这里介绍 cohort 锁。

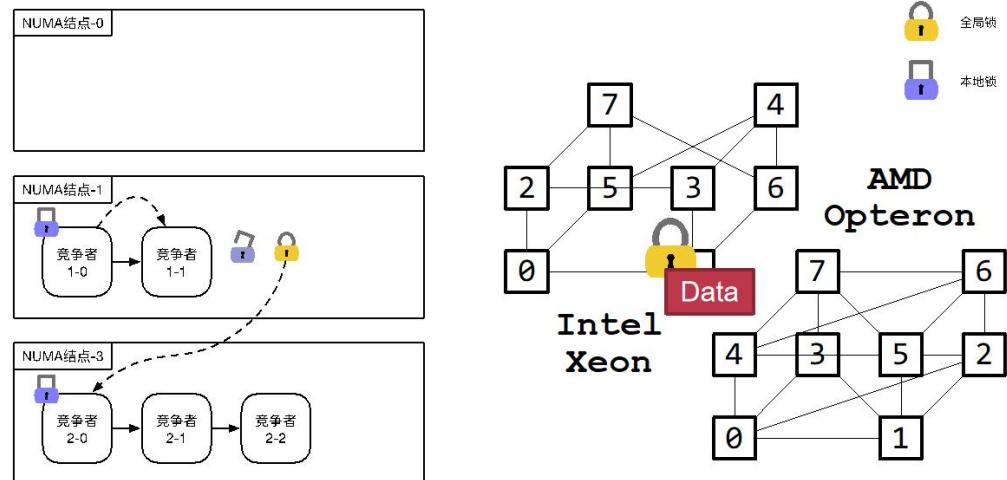
既然我们跨节点的内存访问从时延更高、带宽更小。那么我们到了一个节点访问以后，是不是能把事情做完，再到另一个 NUMA 节点去访问。就和老师给每个宿舍送盒饭，跑到这栋楼应该全部送完再跑到其他核里，而不是在相同的楼之间折返跑。我们可以先获取每个节点的本地的锁，再去获取全局锁。全局锁只会在一段时间内的一个节点内部做传递。这样的话，时延和吞吐都会特别好。

我们来看一个具体的例子：

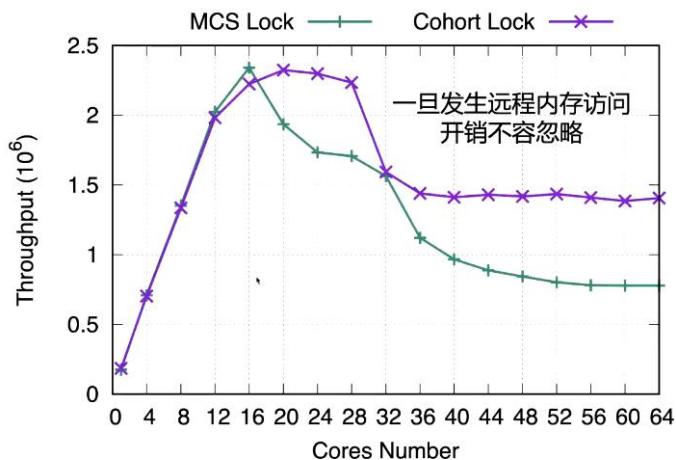
NUMA-aware 设计：以 cohort 锁为例



竞争者 0-0 释放全局锁的时候，可以释放给下一个竞争者 0-1，以此类推再释放给竞争者 0-2 和竞争者 0-3，所以我们可以看到它的访存在一段时间内是一个局部的。如果说我们数据要到下面一个 NUMA 结点了以后，同样地也是在这个 node 里面进行一个传递。



我们看看它的性能：



数量非常简单，其实就是要尽可能把本地处理器的请求完成以后再进入下一个核。性能会比 MCS Lock 好。但是它的性能还是不能特别好，增加处理器核的时候并没有增加性能，因为

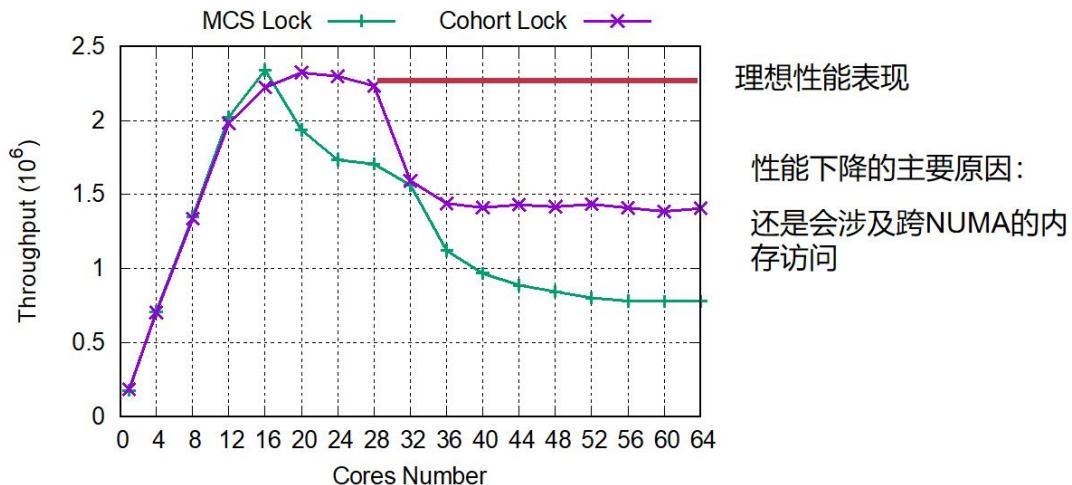
还是存在 NUMA 的情况的。

Q: cohort 锁在获取更好的扩展性的同时牺牲了什么？

A: 它牺牲了一定的公平性换来更好的吞吐性。

举一个定外卖的例子，大家可能先下单。但是外卖小哥可能把别的楼栋的人全部送完了再送过来。在你特别饿的情况下体验就非常不好。

我们能更进一步吗



在这个过程中还是出现了跨 NUMA 的内存访问，虽然缓解了性能下降，但是还是存在下降的。

代理锁

代理锁：通过代理执行避免跨节点访问。

我们能不能通过代理执行的方式缓解跨节点访问？比如我们之前都是要获得全局锁来进行访问。与其让缓存行在不同的 NUMA 节点之间移动，为什么我们不把缓存行限定在一个核心上呢？以快递小哥为例，要么放在快递柜，要么打电话让你来取，也不存在一个个楼送的情况。原来同步临界区的执行，就是先加锁，再对全局变量修改。

例子：临界区访问可以转化为远程执行的闭包

原先的逻辑：

1. lock(&gllock);
2. global_variable += 1;
3. local_variable = global_variable;
4. unlock(&gllock);

临界区可以转换为可以远程执行的闭包

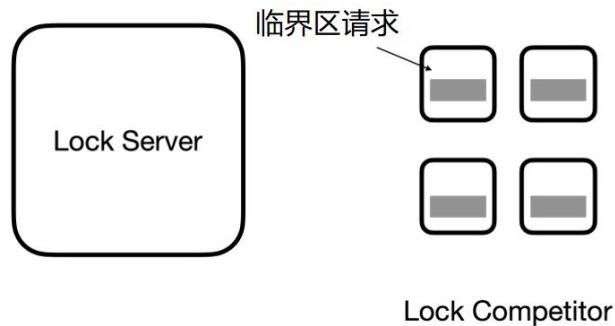
1. void *cs(void *arg) {
2. global_variable += 1;

```

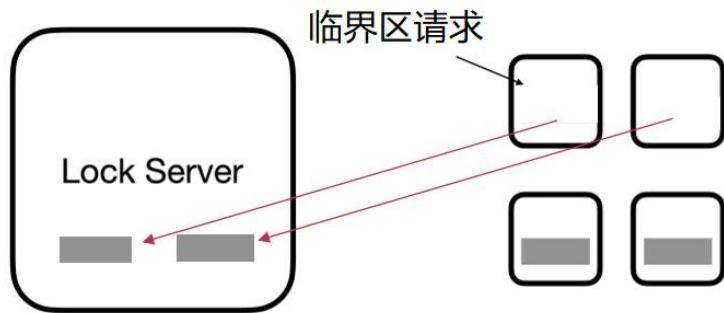
3.     *arg = global_variable;
4.     return NULL;
5. }
6.
7. send_request(cs, &local_variable);

```

这个闭包就是可以一个单独执行的代码。在里面传入一个参数，做完以后再把 `global_variable` 赋值为 `arg` 再返回。我们把对临界区的访问变成了发送一个请求过去。这样我们发现 `global_variable` 是不动的。就像快递小哥在一个地方摆了摊等待我们去拿。所以基本上数据移动是最少的。



临界区的请求我们全部发给 Lock Server。



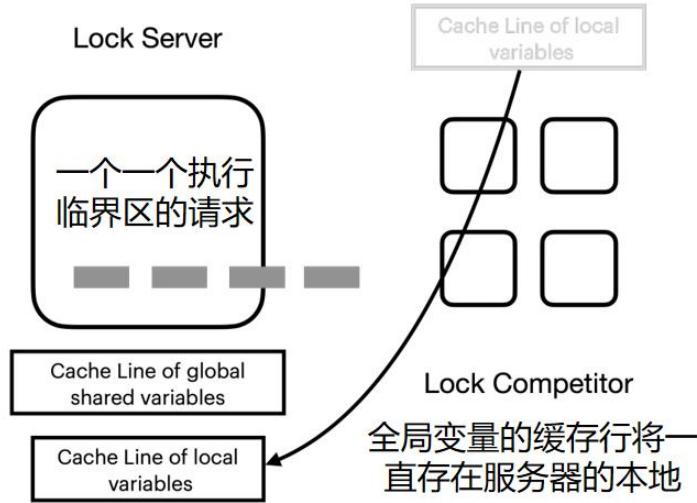
Lock Server 上的处理逻辑如下：

```

1. foreach req in req_list:
2.     ret = req.cs(req.arg)
3.     send_resp(req.core, ret)

```

这样全局变量的缓存行一直是在一个节点的本地。缓存一直在本地的话就不会出现跨 NUMA 节点的访问。



那么谁应该成为锁的服务器呢？它有几种不同的方式。

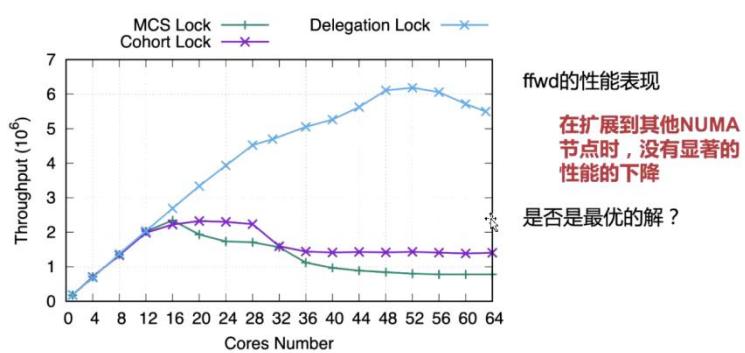
固定锁服务器： ffwd¹, rcl

- 提前准备好一个锁服务器，所有临界区的请求都在这个服务器上执行

非固定锁服务器： sanl², flat-combining

- 当没有锁服务器时，第一个尝试获取锁的竞争者升级成锁服务器

代理锁性能表现



可以看到代理锁在 52 个核的时候性能还是在增加的，但是还是会存在一定的开销。代理锁的设计比 cohort 和 MCS 的性能好很多。

Q：代理锁有什么问题呢？

A：代理锁需要很多代码修改，并且它的额外逻辑很复杂，锁竞争少的时候性能比较差。竞

争多的时候需要动态扩展服务器，就要求设计更精巧。

系统软件开发者视角下的NUMA架构

- NUMA会暴露给操作系统，操作系统可以选择暴露给软件
 - 软件可以用接口来分配本地的内存，也可不用直接分配，如（ libnuma ）
 - 访问**远程内存**会带来**严重时延/带宽**问题造成性能瓶颈
 - 对于所有进程：调度时避免跨NUMA结点迁移
 - 对于没有NUMA-aware的应用：尽可能保证其分配的内存的本地性
- 新的分配里面，我们也要 NUMA_alloc，尽可能分配本地的内存。

读写锁的可扩展性

回顾： 读写锁

性能问题：
读者需要抢一把全局互斥锁
并对一个全局计数器自增

```
struct rwlock {
    int reader;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader += 1;
    if (lock->reader == 1) /* No reader there */
        lock(&lock->writer_lock);
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader -= 1;
    if (lock->reader == 0) /* Is the last reader */
        unlock(&lock->writer_lock);
    unlock(&lock->reader_lock);
}

void lock_writer(struct rwlock *lock) {
    lock(&lock->writer_lock);
}

void unlock_writer(struct rwlock *lock) {
    unlock(&lock->writer_lock);
}
```

它的设计是有一个 lock_reader 和 lock_writer。如果 reader 等于 1 的情况下，就把 writer 锁了，保证在 reader 在里面的时候，writer 进不来。这个结构在核特别多的情况下，核的竞争是非常严重的，我们希望读者特别多的情况下，多个读者都可以进去。但是现在需要持锁，对全局计数器自增以后才能进入临界区。

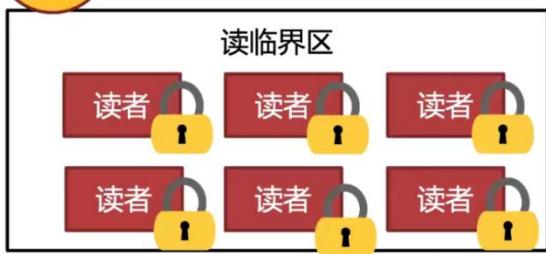
大读者锁

大读者锁

```
struct rwlock {  
    unsigned int total_reader = 0;  
    struct lock reader_lock[MAX_READER];  
    struct lock writer_lock;  
};
```

写者

每个读者将私有一把专属的读者锁：
进读临界区：直接获取私有的互斥锁



```
void lock_reader(struct rwlock *lock)  
{  
    lock(&lock->reader_lock[rid]);  
}  
  
void unlock_reader(struct rwlock *lock)  
{  
    unlock(&lock->reader_lock[rid]);  
}
```

```
--thread int rid; /* 每线程唯一读者编号 */  
int rwlock_init_per_thread(struct rwlock *lock)  
{  
    lock(&lock->writer_lock);  
    rid = lock->total_reader++;  
    unlock(&lock->writer_lock);  
    return rid >= MAX_READER? -1: 0;  
}
```

每一个 reader 都有一个 reader_lock，而只有一个 writer_lock。既然读者之间是相互不干扰的，那么我们给每个读者私有的互斥锁，这样它就不用拿这个全局的锁了。所以我们看到 reader 的 lock 和 unlock 非常简单。

大读者锁



```
void lock_writer(struct rwlock *lock)  
{  
    int i, j;  
    lock(&lock->writer_lock);  
again:  
    for(i = 0; i < lock->total_reader; i++) {  
        if (trylock(&lock->reader_lock[i]) == FAIL) {  
            for (j = 0; j < i - 1; j++)  
                unlock(&lock->reader_lock[j]);  
            goto again;  
        }  
    }  
}  
  
void unlock_writer(struct rwlock *lock)  
{  
    for(int i = 0; i < lock->total_reader; i++)  
        unlock(&lock->reader_lock[i]);  
    unlock(&lock->writer_lock);  
}
```



写者进入临界区：
需要获取所有的私有
读者锁
一旦失败，立刻重试

上海交通大学并行与分布式系统实验室 (IPADS@SJTU)

32

因为我们写的话，需要保证所有读者都不在读，它就尝试把所有的 reader lock 都拿到。写的过程是比较长的，但是我们认为写者会比较少。

大读者锁问题

读者关键路径上仍然有上锁操作：涉及原子操作，性能影响仍然较大

```
void lock_reader(struct rwlock *lock)
{
    lock(&lock->reader_lock[rid[lock->id]]);
}

void unlock_reader(struct rwlock *lock)
{
    unlock(&lock->reader_lock[rid[lock->id]]);
}
```

写者开销巨大

它虽然比原有的读写锁已经好很多了，但是还是涉及到原子操作。我们希望避免在 lock_reader 里避免原子性的操作。

PRWLock

PRWLock 进一步减少读者关键路径性能开销

核心思想：通过版本号协同读者与写者，读者关键路径上只有三个本地访存操作



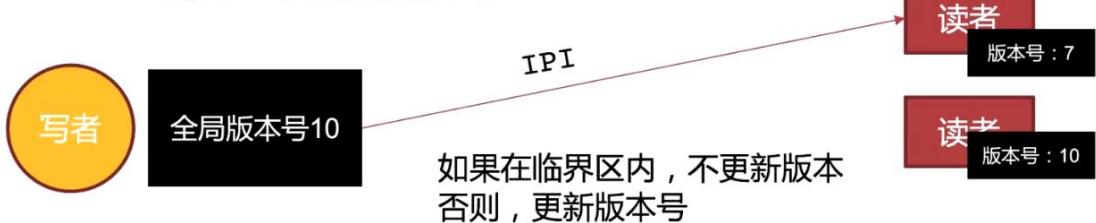
Reader 不主动拿锁，而是隐式地通过版本号进行同步。核心思想就是通过版本号来协同读者和写者。当写者有需要的时候，它会去更新一个全局版本号。而读者进入和退出临界区的时候会更新到最新的全局版本号。



如果所有的读者都知道写者要进去（自己的版本号等于全局版本号），那么读者不会进去，此时写者就可以进去。

有的读者不知道写者要进去怎么办？

少数很久没获取读者锁的读者，通过发送IPI让其更新版本号



```
void lock_reader(struct prwlock *lock)
{
    int core_id = sched_getcpu();
    lock->per_core_reader_state[core_id] = HOLD;
    while (is_locked(&lock->writer_lock)) {
        lock->per_core_reader_state[core_id] = FREE;
        lock->per_core_version[core_id] = lock->version;
        while (is_locked(&lock->writer_lock))
            /* Busy 本号并等待写者 */
        lock->per_core_reader_state[core_id] = HOLD;
    }
}

void unlock_reader(struct prwlock *lock)
{
    int core_id = sched_getcpu();
    lock->per_core_reader_state[core_id] = FREE;
    lock->per_core_version[core_id] = lock->version;
}
```

更新本地状态，供IPI时判断是否要更新

Slow Path: 有写者在了，更新版本号并等待写者

更新本地的version

读者关键路径上
只有三个本地访存操作，没有原
子操作

37

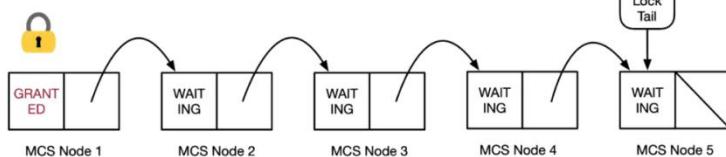
这样读者在关键路径上只有 3 个访存操作，这样就避免了全局锁的操作。

Linux 的同步原语：QSpinLock

我们之前讲了很多设计，比如 Spin Lock（大家同时竞争一个 Cache Line），MCS Lock（需要每一个锁的线程分配一个节点，在自己的节点上 Spin，比较浪费内存）。我们来看一下 Linux 的同步原语。

Linux中的同步原语: QSpinlock

回顾：MCS锁



维护锁等待队列

优势：在锁竞争程度高时，关键路径上只有传递给下一个竞争者的开销

在竞争程度低时，锁性能如何？

在竞争程度低的时候，锁的性能就不太好。

Linux中的同步原语: QSpinlock

回顾 : MCS锁 在竞争程度低时 , 锁性能如何 ?

回顾上锁流程 :

```
void lock(struct MCS_lock *lock, struct MCS_node *me)
{
    struct MCS_node *tail = 0;
    me->next = NULL;           填自己的node
    me->flag = WAITING;
    tail = atomic_XCHG(&lock->tail, me); 更新队尾
    if (tail) {
        tail->next = me;
        while (!SlowPath = GRANTED)
    }
}
```

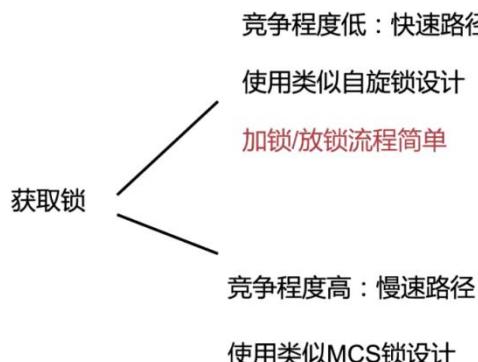
多次访存操作

单次访存操作

对比 spinlock : `while(atomic_CAS(lock, 0, 1) != 0)`
 /* Busy-looping */;

MCS Lock 中有 **多次访存操作** , 而 Spin Lock 在竞争比较低的时候 , 单次访存操作就会好很多。

Linux中的同步原语: QSpinlock



这个实现也非常简单。

Linux中的同步原语: QSpinlock

一个简单的示例 (Linux将其压缩至一个word且更加高效 , 但核心思路类似)

```
void lock(lock_t lock) {                                快速路径 , 只需要set一个1bit
    /* fast path */
    if (spin_trylock(&lock->spin) == SUCC)
        return;
    /* slow path */      失败则加入mcs队列
    mcs_lock(&lock->mcs);   队首会忙等spinlock
    spin_lock(&lock->spin);
    mcs_unlock(&lock->mcs); 成功获取之后会让下一位队首忙等
}                                                        包含两把锁 :
                                                      
void unlock(lock_t lock) {                            typedef struct lock {
    spin_unlock(&lock->spin);                      spinlock spin;
}                                         mcslock mcs;
}                                         }lock_t;
```

我们只需要简单把两个锁结合起来。

Linux 的同步原语： FUTEX

我们之前已经学习过 `wait`, `signal`, `condition`。那么在真实的系统里，到底是怎么设计的呢？

MUTEX 自然是 mutual-exclusive 的锁，而 FUTEX 是 fast userspace MUTEX 的意思。我们先回顾一下：

Linux中的同步原语: futex

如何在互斥锁中避免循环等待 ? **场景二：使用条件变量**

```
void lock(lock_t lock) {  
    spin_lock(&lock->spin);           需要搭配一个忙等的自旋锁  
    while(lock->val == 1)  
        cond_wait(&lock->cond, &lock->spin);  
    lock->val = 1;  
    spin_unlock(&lock->spin);  
}  
                                         引入额外的加锁放锁操作，性能影响  
  
void unlock(lock_t lock) {  
    spin_lock(&lock->spin);  
    lock->val = 0;  
    cond_signal(&lock->cond);  
    spin_unlock(&lock->spin);  
}
```

上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

45

Linux中的同步原语: futex

Futex: Fast User-space muTEX

提供了两个新的接口：

```
void futex_wait(int *uaddr, int val);
```

等待在某个地址上：判断uaddr上的值与val是否相等

- 如果相等则挂起线程
- 否则返回

两个操作互斥：同时只有一个操作发生

不会出现当wait时判断完后、挂起之前，
其他线程调用wake唤醒造成唤醒丢失

唤醒等待在该地址上的线程

使用Futex实现阻塞互斥锁

```
void lock(struct lock *lock) {
    while(atomic_CAS(&lock->val, 0, 1) != 0) {
        atomic_FAA(&lock->waiters, 1);           waiters : 表示是否有等待者
        futex_wait(&lock->val, 1);               atomic避免多个竞争者同时修改
        atomic_FAA(&lock->waiters, -1);
    }
}

void unlock(struct lock *lock)          struct lock {
{                                int val;
    lock->val = 0;                int waiters;
    if (lock->waiters != 0)          };
        futex_wake(&lock->val);
}
```

使用Futex实现阻塞互斥锁：正确性

Thread 1 一种可能的执行序列

```
void lock(struct lock *lock) {
    while(atomic_CAS(&lock->val, 0, 1) != 0) {
        发现val不为0
    }

    Thread 2
    void unlock(struct lock *lock)
    {
        lock->val = 0;
        if (lock->waiters != 0)  解锁，但此时waiter=0，不wake
    }
}
```

使用Futex实现阻塞互斥锁：正确性

一种可能的执行序列

```
void lock(struct lock *lock) {
    while(atomic_CAS(&lock->val, 0, 1) != 0) {
        atomic_FAA(&lock->waiters, 1);
        futex_wait(&lock->val, 1);

        调用wait，发现val不是1了，直接返回，不会挂起
    }

    Thread 2
    void unlock(struct lock *lock)
    {
        lock->val = 0;
        if (lock->waiters != 0)  解锁，但此时waiter=0，不wake
    }
}
```

这就是为什么我们要判断 lock 的 val 是否等于我们期望的值的。

使用Futex实现阻塞互斥锁：正确性

Thread 1

另一种情况

```
void lock(struct lock *lock) {
    while(atomic_CAS(&lock->val, 0, 1) != 0) {
        atomic_FAA(&lock->waiters, 1);
        futex_wait(&lock->val, 1);

        * 调用wait，发现val还是1，挂起
    }

    Thread 2
    void unlock(struct lock *lock)
    {
        lock->val = 0;
        if (lock->waiters != 0)
    }
}
```

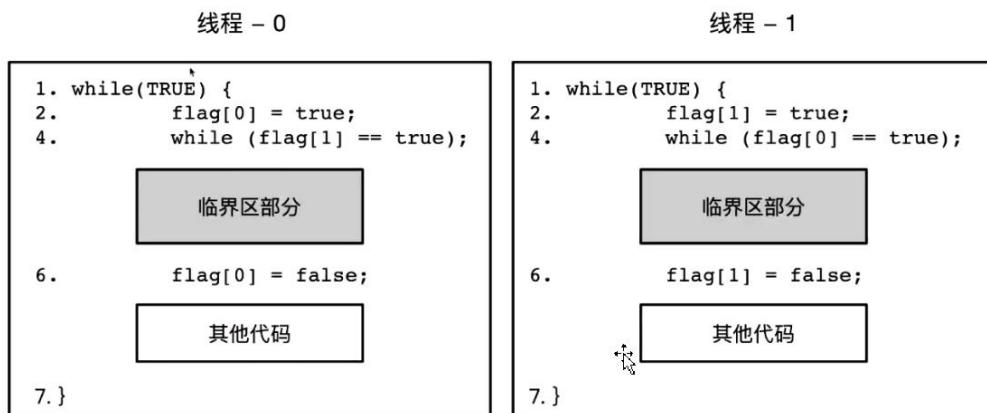
```
    Thread 3
    void unlock(struct lock *lock)
    {
        lock->val = 0;
        if (lock->waiters != 0)
            futex_wake(&lock->val);
    }

    在放锁时可以发现waiter > 0, wake
}
```

内存模型和同步

我们之前介绍了原子操作和很多 Lock。但是我们没有提同步原语都是依赖于内存模型的。

LockOne：皮特森算法的前身



多核环境下能够互斥访问吗？

皮特森算法是没有办法保证互斥的，其次它只能支持两个线程。那么多核环境能不能实现互斥呢？这实际上是有一些问题的。

LockOne在现实硬件中能够保证互斥访问吗？

缓存一致性耗时：阻塞处理器流水线，造成巨大性能开销

处理器允许部分访存操作乱序执行，从而提供更好的并行性

```
void proc_A(void) { flag[0] = 1; Read from void proc_B(void) { flag[1] = 1; Read from while(flag[1]); while(flag[0]); }
```

Read From，必须读到对方设置的最新值

它会阻塞流水线，并且处理器真正执行过程允许访存乱序执行。因为 flag[0] 和 flag[1] 是乱序的，这里面就会出问题。

```
void proc_A(void) { flag[0] = 1; while(flag[1]); } 乱序 void proc_B(void) { flag[1] = 1; while(flag[0]); } 乱序
```

LockOne在现实硬件中能够保证互斥访问吗？



只有最早的处理器才不会发生这种情况。

用原子指令也会有这个问题么？会！

同步原语中也存在内存乱序问题

```
void proc_A(void) {           void proc_B(void) {  
    while(atomic_CAS(&lock,      while(atomic_CAS(&lock,  
        0, 1));                0, 1));  
  
    global_cnt++;             local_cnt = global_cnt;  
  
    lock = 0;                 lock = 0;  
}
```

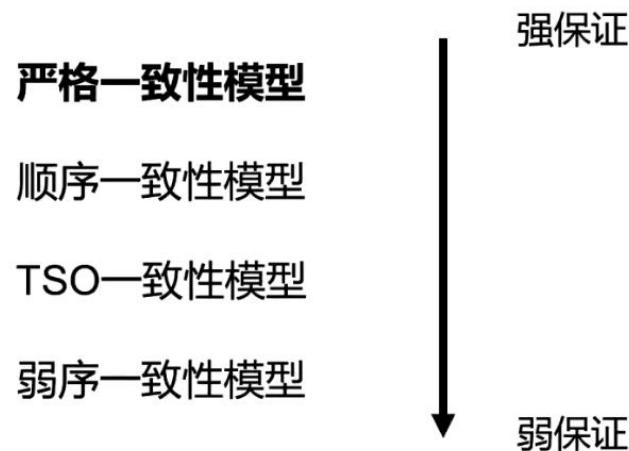
同步原语中也会有乱序的问题。我们期待的一点是拿到锁以后更新这个部分，再释放掉锁。

```
void proc_A(void) {           void proc_B(void) {  
    while(atomic_CAS(&lock,      while(atomic_CAS(&lock,  
        0, 1));                0, 1));  
  
    global_cnt++;             local_cnt = global_cnt;  
    lock = 0;                 lock = 0;  
    }                         }  
}                           }  
→ global_cnt++;
```

乱序到放锁之后才全局可见
读到错误的值

因为这是两个乱序的变量，我们的++可能在释放锁之后执行。

几种内存模型



这就涉及到我们的内存模型了。内存一致性模型就是我们的不同的变量在多个处理器上看到的数据是怎么样的。

内存模型：弱序一致性模型

- 弱序一致性模型 (Weak-ordering Consistency)
不保证任何对**不同的地址**的读写操作顺序

```
int data = 0;
int flag = NOT_READY;

void proc_A(void) {
    data = 666;
    flag = READY;
}

void proc_B(void) {
    while (flag != READY) ;
    handle(data);
}
```



```
int data = 0;
int flag = NOT_READY;

void proc_A(void) {
    flag = READY;
}
-----  

    data = 666;           思考：TSO中有这个问题吗？为什么？
}
```

TSO（所有的写都是同一个顺序）会出现这个问题吗？写的顺序一定是和执行顺序一致的。

内存模型：弱序一致性模型

- 弱序一致性模型 (Weak-ordering Consistency)

不保证任何对**不同的地址的读写**操作顺序

与TSO相比：

- 硬件逻辑更加简单
- 处理器复杂度下降
- 工艺/成本/功耗下降
- 并行程序性能受到影响 (需要手动保证顺序)
- ARM硬件复杂度、性能、成本、功耗、软件使用场景权衡的结果

不同架构使用不同的内存模型



*2012年之前

不同架构使用不同的内存模型



*2020年

▶ 如何在弱的内存模型中保证顺序

LockOne算法

通常的做法：添加硬件内存屏障（barrier/fence）

```
void proc_A(void) {           void proc_B(void) {  
    flag[0] = 1;               flag[1] = 1;  
    保证写-读顺序          ↓ 保证写-读顺序  
    while(flag[1]);           while(flag[0]);
```

传输数据例子

任何访存操作不会逾越内存屏障

```
void proc_A(void) {           void proc_B(void) {  
    data = 666;                while (flag != READY);  
    保证写-写顺序          ↓ 保证读-读顺序  
    flag = READY;             handle(data);  
}
```

那么我们在弱的内存中保证顺序的话，我们必须加一个硬件屏障 barrier。

2022/3/31

不同的文件系统

文件的索引节点：inode

- 常用的元数据
 - 文件类型
 - 文件大小
 - 链接数
 - 文件权限
 - 拥有用户/组
 - 时间（创建、修改、访问时间）
- 具体文件数据的位置

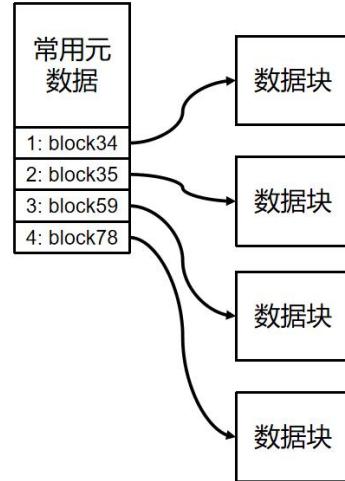


图 2 Unix V6 文件系统

理解一个文件系统的11个问题

1. 文件系统使用磁盘块的基本单位是什么?
2. 一个文件的组织方式?
3. 空闲空间的组织方式?
4. 目录的结构是什么? 1. 如何根据文件名查找到一个文件?
5. 是否支持硬链接? 2. 如何读取一个文件?
6. 是否支持软链接? 3. 如何为一个文件分配新的磁盘空间?
7. 磁盘存储的整体布局是什么? 4. 如何挂载一个文件系统?

文件查找的例子：以查找“/OS/其他/算法笔记”为例

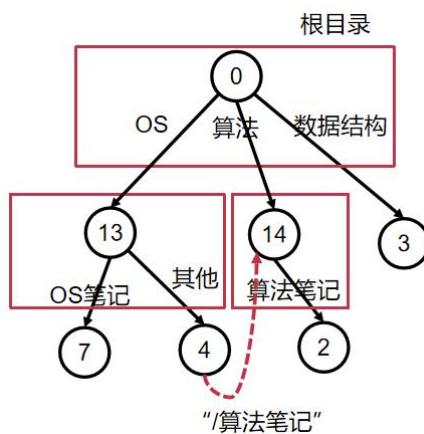
逻辑如下：

- 从根目录开始，在各级目录中逐层找下一级
 - 在“/”中找“OS”文件（目录文件）
 - 在“/OS”中找“其他”文件（目录文件）
 - 在“/OS/其他”中找“算法笔记”文件

代码如下：

```
char *pathname = "/OS/其他/算法笔记"
...
inode = get_root_inode();
...
while (has_next_component(pathname)) {
    ...
    name = get_next_component(pathname);
    inode = lookup(inode, name);
    ...
    if (is_symlink(inode)) {
        inode = resolve_symlink(inode);
    }
    ...
}
...
return inode;
```

最终流程如下：



注意到其中的 /OS/其他/算法笔记 是符号链接到了 /算法/算法笔记 处。

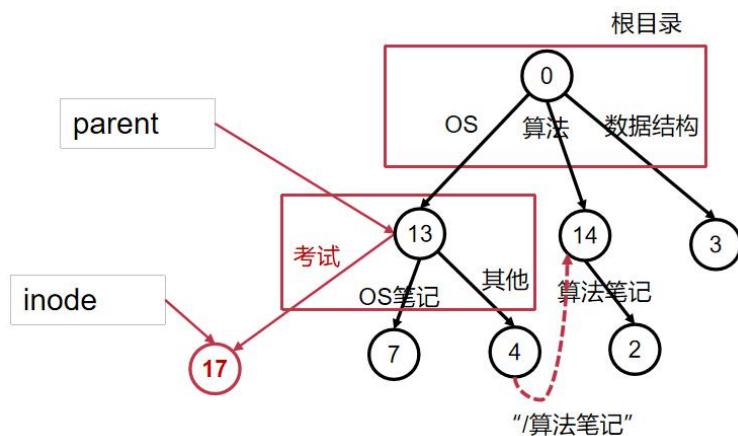
文件创建的例子：以创建 “/OS/考试” 为例

代码如下：

```
char *pathname = "/OS/考试"
...
inode = search_parent_dir(pathname, &leaf);
...
create_file(inode, leaf);
```

```
create_file(parent, leaf):
...
inode = allocate_inode();
...
init_inode(inode);
...
add_dentry_to_dir(parent, left, inode);
...
```

最终结果如下：



打开和读写文件

- 打开文件
 - 查找目标inode，分配新的fd结构保存inode，返回fd号
- 读写文件
 - 默认从fd结构中记录的位置开始读写，读写位置向文件尾移动
 - 可通过lseek调整fd结构中记录的读写位置

```
int fd = open("/OS/考试", O_CREAT | O_RDWR);
write(fd, buf, 5000);
write(fd, buf, 5000);
lseek(fd, 0, SEEK_SET);
read(fd, buf, 5000);
write(fd, buf, 5000);
...
```

```
typedef struct fd {
    off_t pos;
    struct inode *inode;
    ...
} fd_t;

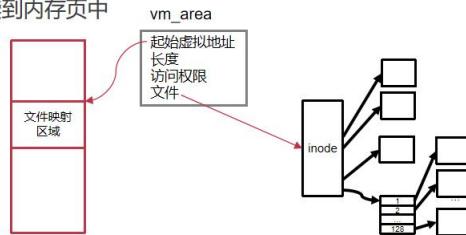
struct task {
    ...
    fd_t fd_table[MAX_FD];
    ...
};
```

mmap 是 OS 中非常重要的组成部分，mmap 实际上是打通了内存和文件这两种接口，允许应用程序用访问内存的方式（load & store）来访问文件。因为“Everything is a file.” 我们访问键盘、显示器、网络设备都可以抽象成文件，所以 mmap 就允许我们用访问内存的方式去访问所有的设备。

mmap(): 用内存接口来访问文件

- mmap可将文件映射到虚拟内存空间中
 1. mmap时分配虚拟地址，并标记此段虚拟地址与该文件的inode绑定
 2. 访问mmap返回的虚拟地址时，触发缺页中断 (page fault)
 3. 缺页中断处理函数，通过虚拟地址，找到该文件的inode
 4. 从磁盘中将inode中对应的数据读到内存页中
 5. 将内存页映射添加到页表中

```
fd = open("/OS/考试", O_RDWR);
addr = mmap(NULL, length, PROT_WRITE,
            MAP_SHARED, fd, 0);
memset(addr, 0, length);
```



mmap() : 文件内存映射的优势

- 对于随机访问，不用频繁lseek
- 减少系统调用次数
- 可以减少数据copy
 - 如拷贝文件，数据无需经过中间buffer
- 访问的局部性更好
- 可以用madvise为内核提供访问提示，提高性能

Q: 当我们在用 mmap 映射一块文件之后，它到底是变快了还是变慢了？

A: 用 mmap 反而能加速访问，因为如果我们调用 read / write 来访问文件的话，因为这是一

个系统调用，所以一定会发生一次 context switch / trap。而使用了 mmap 之后，因为我们映射到内存了，load & store 不会发生到内存的切换（除非是第一次的 page fault），load 完了之后就不会发生 trap 操作。尤其是我们要在多个地方读少量数据的时候，如果我们没有使用 mmap，就会有大量的 syscall 降低性能。第三就是可以减少数据的 COPY，因为它已经在内存里了。第四就是，mmap 之后，memory 就是磁盘的一个 cache，局部性会更好。

我们还可以通过 madvice 告诉内核，接下来我们可能会访问哪些数据。让内核提前做好映射，而不是访问的时候再触发 page fault。比如我们要访问 4M(1000 个页)，如果 OS 提前处理好，那么我们就省去了 1000 次 page fault。注意 madvice 和磁盘关系不是很大，它就是和内存相关的一个函数，但是因为 mmap 打通了内存和文件的接口，所以 madvice 也可以对文件的访问提议。

EXT2 文件系统

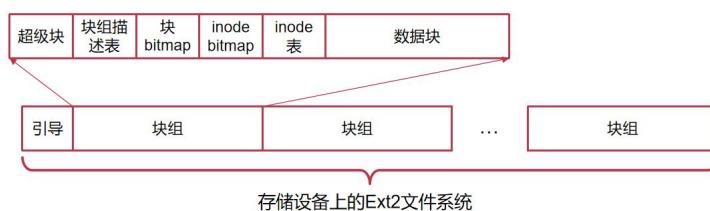
EXT2 文件系统和我们之前讲过的 inode 文件系统非常像，但是它有一个特点就是会把磁盘分成几个块组。每个块组内都有自己的 superblock，换句话说它是把磁盘分成了几个小磁盘。superblock 是记录了整个文件系统的元数据，非常重要，所以互为备份。

Ext2文件系统的存储布局

将磁盘分为多个块组，每个块组中都有超级块，互为备份

超级块 (Super Block) 记录了整个文件系统的元数据

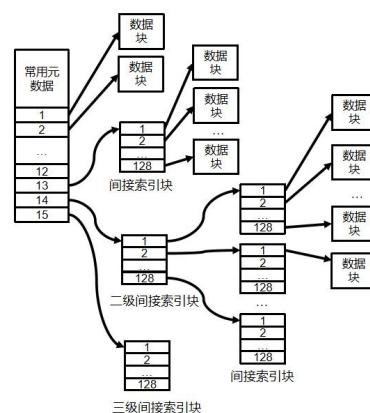
块组描述表记录了块组中各个区域的位置和大小



在 inode 方面，它有如下构成，和我们之前讲的一样。

Ext2的常规文件

- Ext2的inode中
 - 12个直接指针
 - 1个间接指针
 - 1个二级间接指针
 - 1个三级间接指针



一个比较大的区别就是，它使用了 extent (区段) 来进行优化。在一个 inode 文件系统中，我们要保存一个 1G 的视频，会被拆成多少个磁盘块呢？ $1G / 4K = 25$ 万个 block。我们需要记录这么多的 block 的 number，每个 block number 是 8 byte，那么总量是 $8 \text{ byte} * 250000 = 4\text{M}$ 的 inode。这显然是不合理的，如果这些数据块在物理上是连续的话，我们只需要保存起始地址和长度即可了。

所以区段的概念就是对于连续存储的物理块，我们只需要保存起始地址和长度即可。16 Byte 和 4M 的 inode 相差了 5 个数量级。

使用区段 (Extent) 来优化

问题：在Ext2的设计中，保存一个1GB的视频文件，文件被拆成多少数据块？需要多少元数据来维护这些数据块？

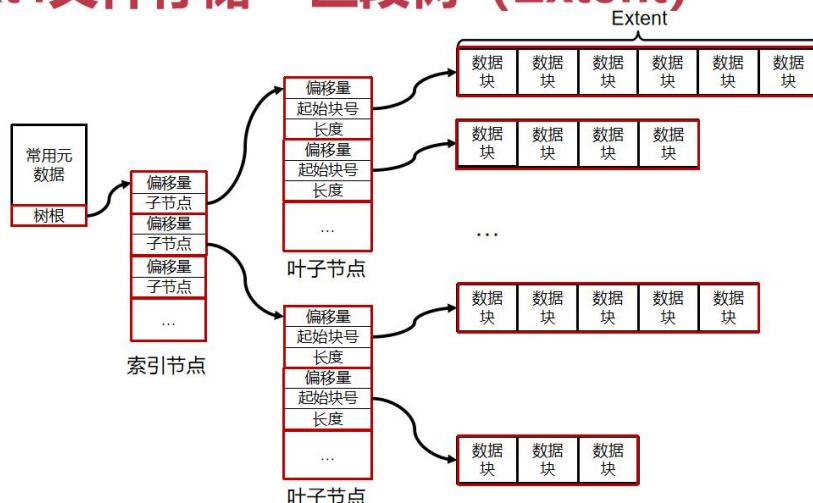
如果这些数据块物理上连续，只需要保存起始块地址和长度即可！

区段 (Extent) 是由物理上连续的多个数据块组成

- 一个区段内的数据可以连续访问，无需按4KB数据块访问
- 可以减少元数据的数量

所以最终 ext4 文件系统就是使用区段树 (Extent Tree) 的方式来进行优化。里面的每一项都是以偏移量+子节点的方式存在。起始块号就指向了连续的 Extent 的第一个。所以这个偏移量中间差了一个长度。

Ext4文件存储 – 区段树 (Extent)



有了 extent 结构之后，比起之前基于块的文件系统是要灵活许多的。因为 Ext 也可以支持很细碎的文件。现在为了支持区段，我们需要在叶子节点/索引节点里记录偏移量和长度，这样就可以很方便地利用起来磁盘里的区域。

我们经常会讲到内部碎片和外部碎片。内部碎片就是我们尽可能压缩每个块的大小到 4K，因为 ext 很灵活，所以无法利用外部碎片的情况就会少很多。

目录项有文件名和目录项长度。一旦目录改名，我们就需要有一个新的 entry 指向这个文件。在 Ext2 如果要做文件名的查找，我们就需要顺序遍历。

Ext2的目录文件

- 内容为一组**目录项**的特殊文件
- 目录项**
 - 文件名+对应文件的inode号
 - “.” 表示当前目录
 - “..” 表示父目录

Ext2的目录项结构

```
struct ext2_dix_entry {  
    __le32 inode; /* inode 号 */  
    __le16 rec_len; /* 目录项长度 */  
    __le16 name_len; /* 文件名长度 */  
    char name[]; /* 文件名 */  
};
```

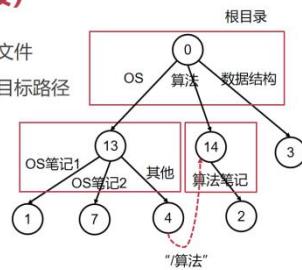
Ext2的目录复用常规文件格式，目录项连续存放



软硬链接也是支持的，这个基本上我们都学过了。

符号链接（软连接）

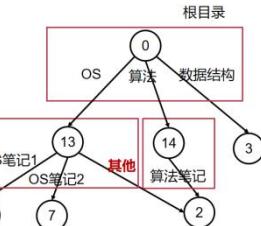
- 内容为一个**路径**的特殊文件
- 查找文件时会“跳”到目标路径
 - “/OS/其他/算法笔记”
 - 2号inode



硬链接

- 不是一种文件
 - 创建硬链接增加了一个指向现有inode的目录项
 - “/OS/其他”与“/算法/算法笔记”为同一份文件

问题：为何通常硬链接不支持链接到目录？



其他还有一些常见的文件类型，比如字符和块设备文件。

其他常见的文件类型

- 字符/块设备文件**
 - 主要记录设备的Major和Minor号
 - Major表示设备类型
 - Minor表示设备编号
- FIFO文件**
 - 即命名管道，用于进程间通讯
- SOCK文件**
 - UNIX域套接字，用于进程间通讯

它们其实并不是平时我们认为的文件，它只记录了 Major 号和 Minor 号，这个文件的大小是固定的。这种文件有特定的属于自己的类型。设备文件也有属于自己的类型。Major 表示设备的类型，Minor 是设备的编号。这样我们就可以识别不同的设备，比如我们用户想打开打印机这个设备，我们就会打开打印机设备文件，拿着这两个编号就可以找到对应的驱动去做处理。

FIFO 就是命名管道。比如 `ps aux | grep`，这个就没有建立一个新的文件。我们也可以在磁盘上建立一个 FIFO 的文件，这样两个进程就可以做相应的操作。FIFO 文件也是有自己的

编号，还有一类就是 SOCK 文件。

自此，我们就把 EXT2 讲完了，它和 inode FS 最大的区别就是有一个 extent 的概念，可以利用区段去保存，对大文件的支持是非常好的。

ChCore 中的文件系统

在 ChCore 中，事情就比较简单了。这是一个内存文件系统，数据是存在内存中的，不考虑在磁盘上的持久化问题。它有一个 nlink，类似于 ref_cnt，type 也是必要的。然后就是一个 union 记录了一个哈希表和 radix tree。

ChCore中的文件系统

- ChCore中实现了一个内存文件系统
 - 数据存在内存中，无持久化和存储格式

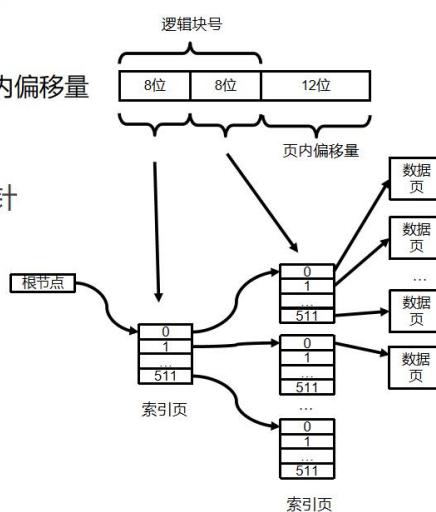
ChCore中的inode

```
struct inode {  
    int nlinks;           /* 链接数 */  
    u64 type;            /* 文件类型 */  
    size_t size;          /* 文件大小 */  
    union {  
        struct htable dentries; /* 目录文件的哈希表 */  
        struct radix data;     /* 常规文件的基数树 */  
    };  
};
```

它的常规文件就是通过 Radix Tree（基数树）来进行表示的，并且提供了 key-value 的索引，和我们前面的 inode 多级间接指针是非常类似的。我们根据文件内的偏移，找到实际内存地址在哪。所以保存了 Radix Tree 的话，就能保存下来最终的数据在内存中的地址。

ChCore的常规文件

- 基数树 (Radix Tree)
 - 提供<键,值>索引
 - 如前述inode中的多级间接指针
 - 类似内存页表的结构
- ChCore中的基数树
 - 保存<逻辑块号, 块指针>



在 ChCore 中，目录使用 hash table 来保存。当我们要去保存文件的时候，其实是把整

个文件名用哈希表保存起来，如果哈希冲突了，我们就加一个链表。通过哈希表的话，找目录中的文件是比较快的。整个 ChCore 中的目录是保存在一个哈希表里面的。在 Linux 里的根目录下通过 `find` 找一个文件名是非常慢的，因为找文件名对 ext4 文件系统来说是很难的，因为文件名分散在磁盘上的很多地方。而 ChCore 的文件名集中在哈希表里，所以查找会比较快。

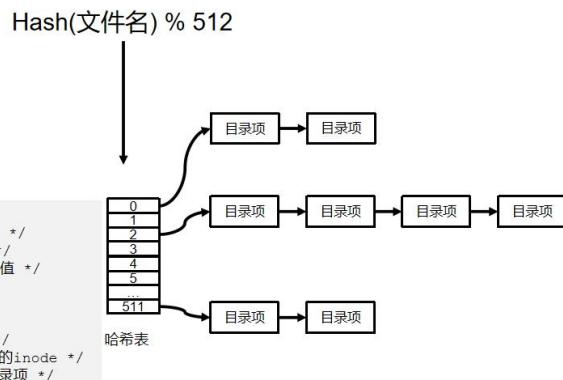
ChCore的目录文件

- 哈希表
 - 保存目录项
 - 使用链表解决冲突

ChCore中的目录项结构

```
struct string {
    char *str;          /* 文件名字符串 */
    size_t len;         /* 文件名长度 */
    u64 hash;          /* 文件名的哈希值 */
};

struct dentry {
    struct string name; /* 文件名 */
    struct inode *inode; /* 文件对应的inode */
    struct dentry *next; /* 下一个目录项 */
};
```



基于 Table 的文件系统: FAT

FAT 又是一种不同的实现。FAT 一开始就会对整个磁盘做格式化，格式化的时候会有对应的参数。一般情况下，32~64 MB 对的情况下，Cluster (Block) 的大小是 512 byte。最大的 Volume 下支持的 Cluster 大小是 16K。在整个 FAT 文件系统里，最头上的叫做“引导”，是用来启动的，像 DOS 和 Windows，引导区都是放 OS 启动的代码。FAT1 和 FAT2 是相互备份的。

FAT32存储布局

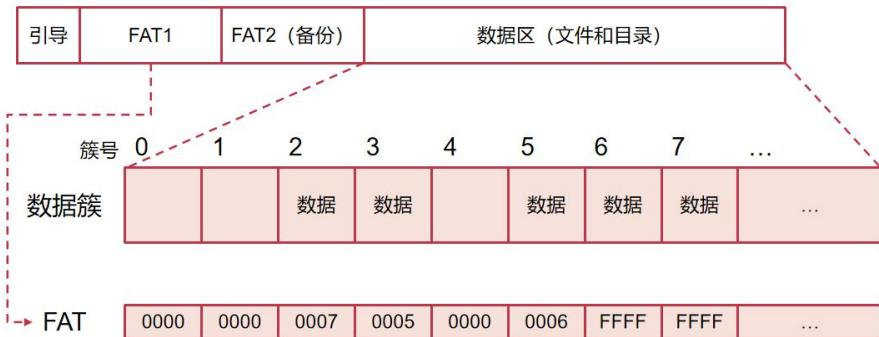


Volume Size	FAT32 Cluster Size
<32MB	Not Supported
32MB ~ 64MB	512 bytes
65MB ~ 128MB	1KB
129MB ~ 256MB	2KB
257MB ~ 8GB	4KB
8GB ~ 16GB	8KB
16GB ~ 32GB	16KB

Default FAT Cluster Sizes (簇)

逻辑上很简单，我们只需要看数据区即可。数据区中的每一块（每一簇）都对应了 FAT 中的一小项。所以这里我们也可以看出来，FAT 表的大小和数据区的大小是成比例的。

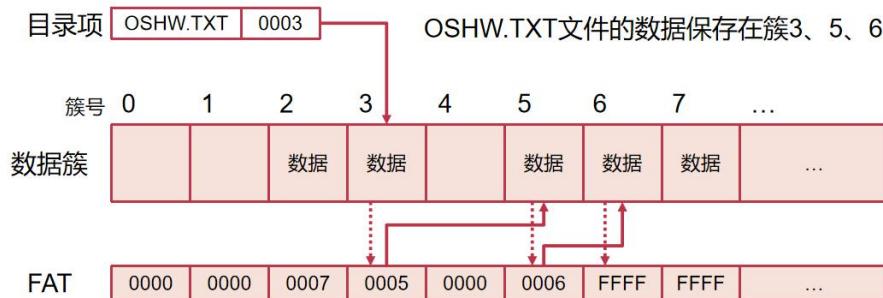
FAT：文件分配表



FAT为每个数据簇增加了一个next指针，让簇可以串联在一起

上图中 FAT 的一小项是 2 个 Byte，而一个簇可能是 16K。FAT 基本上占总数据的万分之一。

FAT 表的每一项其实就是一个指针，让不同的簇可以连接在一起。如下图所示，假设我们现在有一个文件，它的 number 指向了 3 号数据，下一个 block 就在 data[3].next 的位置。全 F 就表示是文件的结尾。



FAT为每个数据簇增加了一个next指针，让簇可以串联在一起

在 FAT 里每个文件也可以认为有一个编号，就是第一个簇所在的地址。只要我们知道第一个，我们就可以把文件用链表的方式给组织起来。

接下来我们探究 FAT 中文件的目录项怎么组织。目录也是一种特殊的文件。目录文件包含若干个目录项，每个目录项记录 32 个字节，它的大小是固定的。一旦大小固定了，我们的文件名万一特别长超过了 32 个字节，那就很麻烦了。所以这里分成了短文件名目录项、长文件名目录项、卷标目录项和“.” “..” 目录项。

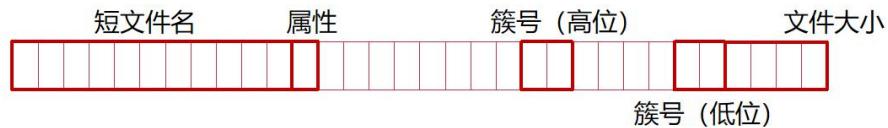
FAT32中的目录项

目录同样是一种（特殊的）文件——与基于inode的文件系统一样

目录文件包含若干个目录项，每个目录项记录32个字节

四种目录项：

短文件名目录项、长文件名目录项、卷标目录项、".."和".."目录项

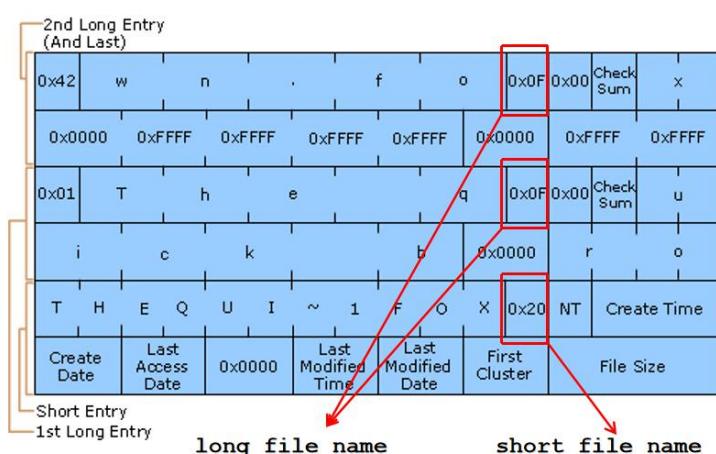


我们来看一个例子，这是 FAT 的一个目录项。每两行是一个目录项。文件名字叫做 “The quick brown.fox” 。这个文件名显然太长了，在 32 byte 里放不下。于是，我们必须把文件名变成 8.3，也就是 8 个字符.3 个字符。8.3 是微软里经典的命名规范，于是它这里就存了 Thequi~1.fox，是根据原文件名生成的短文件名。每个长文件名都有一个短文件名。因为长文件名一个目录项存不下，所以它需要多占据两个目录项。

0x0F 表示这两项是附属于短文件名的。0x42 表示是第二个，0x01 表示是第一个。这样一个三合一的目录项，就组成了一个长文件名的目录项。

FAT32中的目录项

e.g. file with "The quick brown.fox" and "Thequi~1.fox"



Q: 长文件名生成短文件名，如果生成的短文件名冲突了怎么办？

A: 规则如下:

如何生成短文件名？

- "The quick brown.fox" -> "Thequi~1.fox"
- **如果后一个文件已经存在，怎么办？**
 - 尝试: THEQUI~2FOX
 - 若还冲突，则尝试: THEQUI~3FOX
 - 若还冲突，则尝试: ...
 - 若还冲突，则尝试: T~999999FOX
 - 若还冲突，则报错

当我们根据目录去找的时候，肯定也是从根目录开始找到对应的文件名，找到对应的结构。一层层往下找，这个就跟 inode 是很像的。

Q: 如果我们使用短文件名直接去访问文件可以吗？

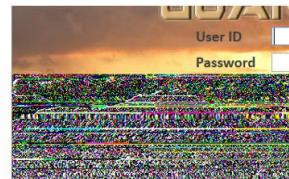
A: 当然可以，反正每个长文件都拥有两个文件名，两个文件名都可以访问到。这个和我们之前的硬链接的不同的，需要区分两个功能。

Q: FAT32 最大能支持多大的大小呢？

A: 它只能支持 4G 的文件。原因很简单，因为 file_size 只有 32 位。我们要扩展的话起码要扩展 file_size 的域。

思考时间 🧐

- FAT32最大支持多大的单个文件？为什么？
- 应该如何扩展FAT，使其能支持更大的文件？
- 为什么U盘一般用FAT？
- 为什么FAT不支持link（硬链接）？
- 为什么有时候会出现这样的错误？
- 为什么FAT会有大量的随机读写？



一般 U 盘放的文件都小于 4G，大的文件可能放在移动硬盘里。

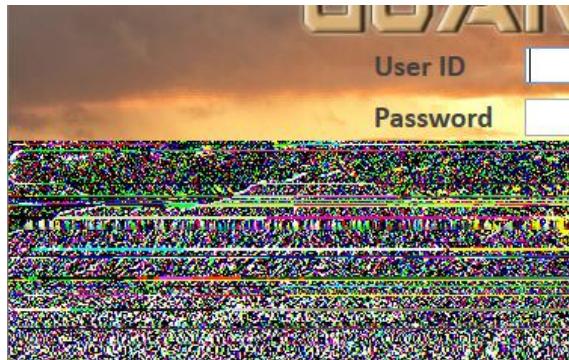
Q: 为什么 FAT 不支持硬链接呢？如果在 FAT 里支持硬链接会怎么样呢？

A: 因为在一个目录项里面，我们绑定了一个文件的 mtime、atime、size、name 等。假设我们要支持硬链接，那就说明另外一个地方有一个文件名同样也占了一个目录项，并且指向同一块数据（First Cluster），这个目录项同样要记录 mtime、atime、size、name。这导致这些原数据记录在两个地方，有两份。一旦文件被改了，必须找到所有指向文件的目录项，去修改它里面的 file_size。所以，不是不能实现硬链接，只是说从效率来说很明显是不合理的。

在 inode 里，文件名和元数据是分开的，整个元数据有一个 id，而不是说数据有一个 id。

所以我们可以建立起字符串到元数据的映射。而在 FAT 里文件名是元数据的一部分，所以很难支持硬链接。

Q：有时候照片可能出现如下图所示的错误，大家能不能想到为什么？



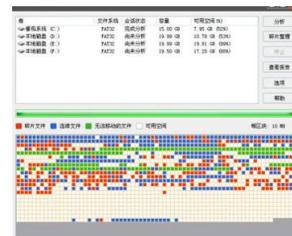
A：是因为一旦我们的某一项的 next 指针坏了，我们后面的数据都乱了。所以 FAT 文件系统有时候会出现这种一半好一半坏的情况。

Q：FAT 会不会有大量的随机读写？为什么？

A：我们从 FAT 连接结构说起，空闲块也是通过链表的方式连接，所以并没有局部性的概念。所以使用过程中会依赖于分散，我们就得去做磁盘整理。

思考：磁盘碎片

- 磁盘碎片是如何产生的?
 - 思考场景：增大一个文件
 - 表现形式：磁盘使用块不连续
 - 磁盘碎片会导致什么问题？
- 如何避免磁盘碎片?
 - 做好磁盘的预留
 - 利用内存缓存延迟写入磁盘
 - 写入时尽可能整合在一起



早期Windows的磁盘整理

当我们去增大一个文件的时候，我们就要在后面加一块。如果后面不够了，我们就要加到很远很远的地方。这就导致磁盘块的使用不连续，这会导致读文件的时候，磁头要跳来跳去。

一种方法就是创建文件的时候做一点预留的位置，这只能在磁盘比较充裕的情况下才能做。第二种方法就是利用内存缓冲写的操作，内存把很多写的操作吸收掉了，而且我们知道要写多长，这样我们就可以在磁盘上找到一块合适的大小写进去。

今天的 Windows 的碎片问题远远没有那么严重，主要还是 FAT32 时代的问题。

exFAT

exFAT 建立在 FAT 的基础之上，其次使用了 bitmap 去记录了空闲的 block，而不是用一个链表的方式，这样就可以加快空间的分配。使用 Unicode 保存长文件名，而不是简单的字符，可以支持的字符种类更多。并且它支持 4G 以上的文件，扩充了目录项的格式。

exFAT Highlights

- 与FAT32并不兼容
- 使用Bitmap加快空间分配
- Unicode保存长文件名
- 允许4GB以上文件 (新的目录项格式、文件大小用8个字节)
- 目录中查找文件时使用哈希对比
 - 对文件名的大写形式做哈希
 - 先匹配哈希值，再检查文件名防止冲突
- 使用校验码保证元数据完整性
- 为闪存做优化
 - 可调参数，与存储单元边界对齐
 - OEM域
 - 没有日志

基于数据库的文件系统：NTFS

讲完 FAT 之后，我们立刻会去将 NTFS。它和 FAT 有很多类似的地方，也有自己的特色。首先它针对不同的磁盘，Cluster Size 也是不同的，这样可以比较好地避免内部碎片。可以看到 MFT 和数据区是相间隔的。

NTFS存储布局



Volume Size	NTFS Cluster Size
7MB ~ 512MB	512 bytes
513MB ~ 1GB	1KB
1GB ~ 2GB	2KB
2GB ~ 2TB	4KB

Default NTFS Cluster Sizes

MFT 就是 NTFS 的主文件表。它本质上是一个关系型数据库，逻辑上 MFT 的每一行就对应了一个文件，而每一列就对应了文件的某个元数据。NTFS 中所有文件都会在 MFT 中有一行，它记录了所有文件的元数据。1/8 的空间专门用来保存 MFT。这个预留是有必要的。在 NTFS 中，真正做到了一切皆文件：

1. 被分配的空间一定属于某一个文件。
2. 用于存放文件系统元数据的空间也属于一个文件，如 MFT。

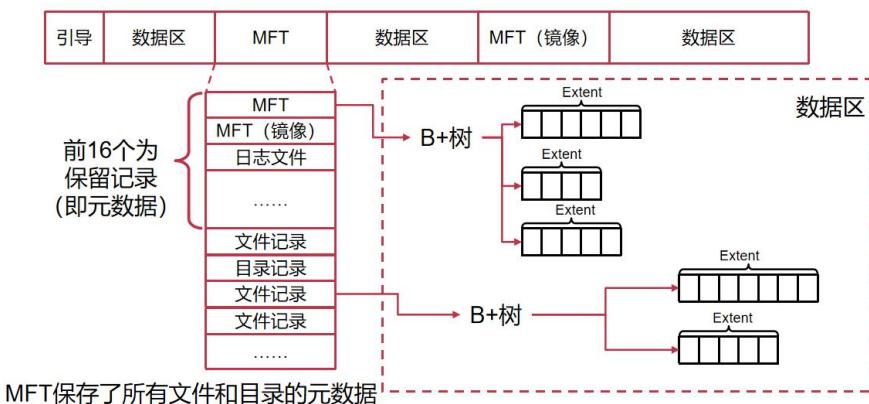
NTFS主文件表MFT

- MFT是一个关系型数据库 (from 微软文档)
 - MFT中的每一行对应着一个文件
 - 每一列为这个文件的某个元数据
 - NTFS 中所有的文件均在 MFT 中有记录
 - 一般会预留整个文件系统存储空间的12.5%，专门保存MFT
- 一切皆文件
 - NTFS 中的所有被分配使用的空间均被某个文件所使用
 - 用于存放文件系统元数据的空间，也会属于某个保留的元数据文件
 - 如：MFT本身，也是一个文件，其元数据保存在MFT中（递归了？）

我们来看如下的一个图：

MTF 第一项就是 MFT，它记录了 MFT 本身的元数据。还会有一些日志文件，前十六项都是记录 NTFS 的元数据的文件。在 EXT4 中，日志文件在整个磁盘块的最后，它不是一个文件，是一块磁盘空间。而在 NTFS 中把日志也认为是一个文件。

NTFS主文件表MFT



保存的一些属性有哪些数量和描述就保存在 4 中。哪些已经使用和空闲的簇 (bitmap)，就保存在 6 中。它把所有东西都作为一个文件。

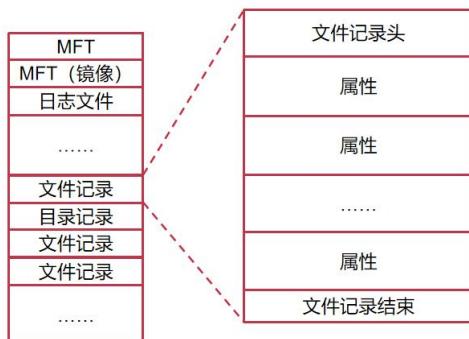
主文件表包含的文件（保留文件）

元数据文件	文件名	序号	文件说明
MFT	\$Mft	0	保存了 MFT
MFT 镜像	\$MftMirr	1	MFT 的备份
日志文件	\$LogFile	2	用于保证文件系统元数据一致性
卷	\$Volume	3	保存卷信息，如卷标和版本
属性定义	\$AttrDef	4	保存支持的属性名、数量和描述
根文件夹	.	5	根文件夹
簇的位图	\$Bitmap	6	标记已经使用的和空闲的簇
启动扇区	\$Boot	7	保存用于加载和启动卷的信息和代码
坏簇文件	\$BadClus	8	保存卷中损坏的簇
安全文件	\$Secure	9	保存卷中每个文件的唯一安全描述符
大写表	\$Upcase	10	用于进行 Unicode 相关转换
拓展文件	\$Extend	11	用于文件系统拓展，如磁盘限额
		12–15	未使用的记录

所以整个 MFT 就在头上记录了元数据信息。后面就是文件和目录信息，有点像是 inode，它通过 B+树记录了所有的块，块是可以以 extent 机制存在的。

有一个特点就是，所有的文件都在同一个地方，而且主文件表里每一项记录了属性。包含了文件的标准元数据、文件名、数据、索引等。

主文件表记录



常用属性包括：

- 文件标准元数据（大小、时间等）
- 文件名
- 数据
- 索引根

NTFS 保存的位置包含了常驻文件和非常驻文件。常驻文件就是小文件，比如大小小于 1K。既然文件很小，那么这种情况下，我们可以直接把文件内嵌到 MFT 文件中，这样我们就不需要后面还有一个 B+树了。这种记录对小文件就特别友好。硬链接本身会拥有一个单独的目录项指向文件的 ID（MFT 中的序号）。

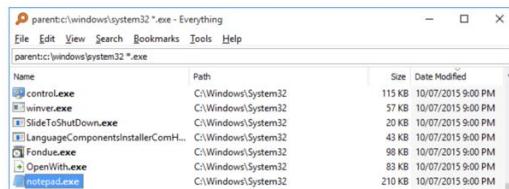
NTFS数据保存位置和目录项

- 非常驻文件（大文件/目录）
 - 数据区的B+树和区段
- 常驻文件（小文件/目录）
 - 大小不超过MFT记录的最大值（1KB）
 - 内嵌在MFT中保存（在“数据”属性中）
- 目录项与硬链接
 - 包含文件名、文件ID（在MFT中的序号）
 - 支持硬链接：每个硬链接拥有一个单独的目录项

在 Windows 里有一个 Everything 的工具，它搜任何东西都可以秒搜。它第一件事情就是读取 MFT，并且把里面的每个文件生成一个索引。这是文件名集中带来的好处，我们可以用最快的方法找到所有的文件名。

思考时间

- 为什么Everything查找文件这么快？



Name	Path	Size	Date Modified
control.exe	C:\Windows\System32	115 KB	10/07/2015 9:00 PM
winvver.exe	C:\Windows\System32	57 KB	10/07/2015 9:00 PM
SlideTo ShutDown.exe	C:\Windows\System32	20 KB	10/07/2015 9:00 PM
LanguageComponentsInstallerComH...	C:\Windows\System32	43 KB	10/07/2015 9:00 PM
Fondue.exe	C:\Windows\System32	98 KB	10/07/2015 9:00 PM
OpenWith.exe	C:\Windows\System32	83 KB	10/07/2015 9:00 PM
notepad.exe	C:\Windows\System32	210 KB	10/07/2015 9:00 PM

- 为什么NTFS存取小文件很高效？

NTFS 存储小文件高效是因为小于 1K 的小文件可以存在 MFT 里。

今天我们讲了 Unix 第六版、Ext2~Ext4、ChCore FS、FAT 和 NTFS。无论是哪种文件系统，对外提供的接口都是一样的，只是内部组织形式不同。我们在使用一个 OS 的时候，很多时候会挂载多个不同文件系统的硬盘，我们希望设计一个可以容纳这些不同 FS 的系统。

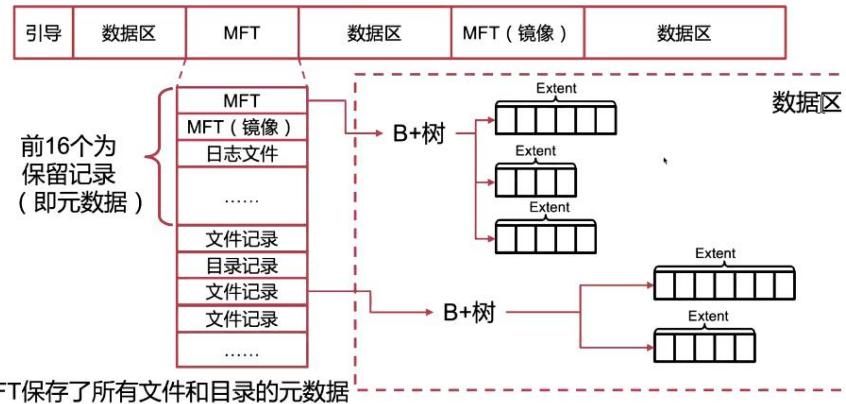
2022/4/7

文件系统结构

我们今天的内容是文件系统的结构。我们上节课介绍了 NTFS 文件系统的结构，也介绍了它的主文件表 MFT 这个结构。MFT 一方面有一个主的一部分，还有 mirror 做容错。MFT 里保存了所有文件和目录的元数据，元数据里包括文件的信息和文件的指针。MFT 存有的文

件和目录的信息可以认为是全局的信息，方便 Windows 对文件做查找和索引。对每个目录里，也保存了文件名和对应的信息，这样方便文件打开一个目录的时候获取下面对应的文件和目录。上节课我们也讨论过里面有冗余，比如 MFT 里和文件夹里都存在文件的名字。但是 MFT 和文件目录树分别用于不同的目的，通过 MFT，一个 OS 可以快速遍历文件建立索引，同样，我们通过文件目录树中的信息，可以快速地打开文件。

Review: NTFS主文件表MFT



我们可以回顾一下我们在讲内存管理的时候，`page table` 和 `struct page` 中都有对应的信息，也是有冗余的。

我们上节课还介绍了 NTFS 中文件和硬链接，每个文件的文件名是保存在目录和 MFT 中的属性中。所以我们的硬链接需要同时更新两个文件名。

Review: NTFS的目录和硬链接

- **每个文件的文件名保存了两份**
 - 一份在目录中
 - 一份在MFT的文件记录中的属性 (ATTRIBUTE) 中
- **硬链接需要同时更新两份文件名**
 - 在目录中：新建一个目录项，指向文件MFT记录的ID
 - 在文件的MFT记录中：新增一个属性，其中有文件名，以及parent目录
- **为了兼容DOS支持8.3短文件名**
 - 通过一个特殊的硬链接实现：会随长文件名更新而更新

这是一个 MFT 的 entry，里面有对应的 attribute。attribute2 的类型是一个 FILE_NAME，它会保存 parent 文件的引用，还保存了对应的文件名。

```

MFT entry: 42 information:
    Is allocated           : true
    File reference         : 42-1
    Base record file reference : Not set (0)
    Journal sequence number : 0
    Number of attributes   : 4

Attribute: 1
    Type                  : $STANDARD_INFORMATION (0x00000010)
    Creation time          : Dec 01, 2019 08:37:58.333261200 UTC
    Modification time       : Dec 01, 2019 08:37:58.333261200 UTC
    Access time             : Dec 01, 2019 08:37:58.333261200 UTC
    Entry modification time : Dec 01, 2019 08:37:58.337681600 UTC
    Owner identifier        : 0
    Security descriptor identifier : 263
    Update sequence number  : 2080
    File attribute flags    : 0x00000020
                                Should be archived (FILE_ATTRIBUTE_ARCHIVE)

Attribute: 2
    Type                  : $FILE_NAME (0x00000030)
    Parent file reference  : 41-1
    Creation time          : Dec 01, 2019 08:37:58.333261200 UTC
    Modification time       : Dec 01, 2019 08:37:58.333261200 UTC
    Access time             : Dec 01, 2019 08:37:58.333261200 UTC
    Entry modification time : Dec 01, 2019 08:37:58.333261200 UTC
    File attribute flags    : 0x00000020
                                Should be archived (FILE_ATTRIBUTE_ARCHIVE)
    Name                  : testfile3

```

虚拟文件系统（VFS）

今天我们首先会介绍 VFS。它里面会包括我们如何提供缓存、支持多文件系统，以及用户态文件系统 FUSE。我们还会介绍系统里一个很重要的概念：crash consistency。在任何时候出现断电等崩溃的时候，如何使得我们数据不会丢失、文件系统不会损坏。

首先，我们会面临的问题是，在现实生活中有各种各样的文件系统。苹果在 17 年提出了 Mac 的 APFS。在安卓里用的比较多的是 F2FS (Flash-Friendly File System)。我们可以看到一个 OS 需要提供各种各样的文件系统，我们需要对用户屏蔽文件系统的异构性。这个异构性其实就是我们的 VFS，它提供了一个中间层。对上提供 POSIX API，对下提供不同的文件系统抽象。

如何在一个系统中同时支持多个文件系统？

- **计算机中的异构文件系统**
 - Linux 和 Windows 双启动，两个分区有各自的文件系统
 - Mac 用 APFS，U 盘一般用 FAT/exFAT，移动硬盘用 NTFS

- **如何对用户屏蔽文件系统的异构性？**
 - VFS : Virtual File System
 - 中间层，对上提供 POSIX API，对下对接不同的文件系统驱动

它的理念很像我们的面向对象。VFS 可以认为是所有的 FS 的 interface，所有 FS 实际上

就是去实现了这样的一个 interface。Linux VFS 就是定义了一些接口。我们在读取 inode 文件的时候，首先通过 VFS 找到对应的文件系统，再去调用对应的接口。

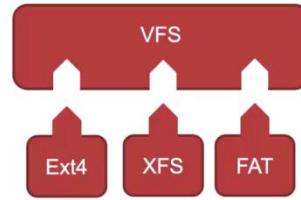
Linux中的虚拟文件系统VFS

Linux的VFS定义了一些系列接口

具体的文件系统实现这些接口

如在读取一个inode的文件时

- VFS先找到该inode所属文件系统
- 再调用该文件系统的读取接口



Windows的类似机制

- Installable File System

VFS 提供了统一的管理。如果每个文件系统都有自己的一个根节点，那么它们应该怎样配合在一起呢？文件系统 1 和文件系统 2 里有不同的根节点和不同的文件。

虚拟文件系统 VFS

操作系统同时使用多个文件系统

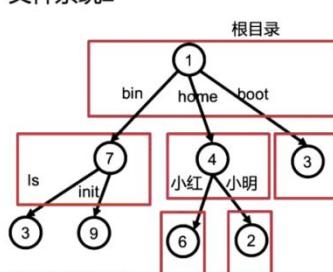
虚拟文件系统提供统一的管理，对应用程序提供统一的视图和抽象

问题：每个文件系统都有自己的根节点，它们如何配合在一起？

文件系统1



文件系统2



上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

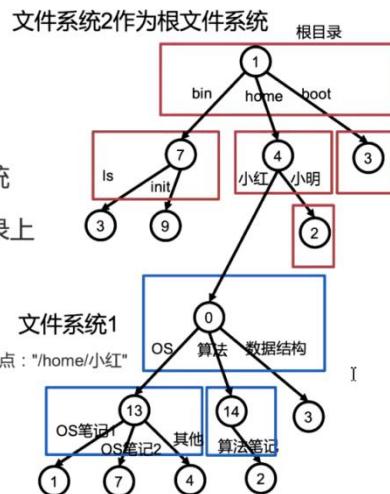
在一个 OS 中，所有人才认为自己的根目录，那么怎么处理呢？VFS 会维护统一的一个文件系统树，通过挂载的概念使得根目录是唯一的。我们可以把文件系统 2 作为根文件系统，那么我们可以从中任意选择一个文件系统 1 的挂载点（/home/小红）。这样我们就把多个文件系统通过挂载的形式形成统一的文件系统的树（只有一个根节点）。

虚拟文件系统 VFS

VFS维护一个统一的文件系统树

操作系统内核启动时会挂载一个根文件系统

其他文件系统可以挂载在文件系统树的目录上



比如我们在 Mac/Linux 中插入一个 U 盘，U 盘通常是 FAT 文件系统，它可能是挂载在主机的文件系统里的，如/mount/... 目录下。这就是我们挂载的作用。如果我们没有 unmount，直接把 U 盘里可能会导致数据的丢失，这是因为数据可能有一些在 cache 中，并没有写回到 U 盘中。直接拔掉就会导致数据的丢失。

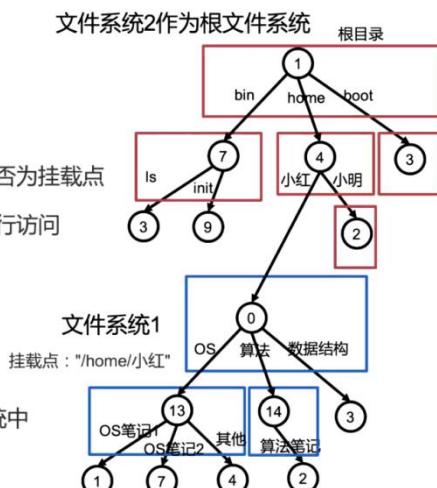
比如在这个例子中文件系统 1 是 FAT 文件系统，那么在访问到挂载点下的文件时，VFS 就会使用 FAT 来进行访问。这样我们就可以使用对应文件系统的操作来访问文件系统。

虚拟文件系统 VFS

VFS维护所有的挂载信息

- 查找文件时的每一步，检查当前目录是否为挂载点
- 若是，则使用被挂载的文件系统继续进行访问

文件系统1中的"/OS/OS笔记1"通过操作系统中的"/home/小红/OS/OS笔记1"访问



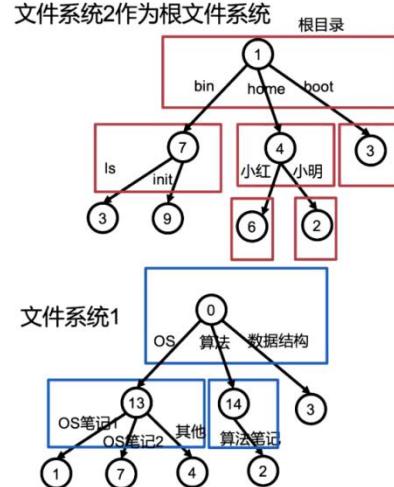
在挂载的时候，会把原有的文件暂时覆盖掉。

虚拟文件系统 VFS

挂载在逻辑上覆盖挂载点原有的结构

- 无法访问文件系统2中的"/home/小红"

挂载点下的数据在卸载后依然可以访问



接下来我们来看一下 Linux 中的文件系统对应的结构。它会对应一些 inode 的操作接口。`dentry` 结构就是查找一个文件（`lookup`）操作的返回值。一个具体的文件系统就会对这些接口实现。

Linux中的虚拟文件系统VFS

Linux的VFS定义的一些inode上的操作接口

```
struct inode_operations {  
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);  
    ...  
    int (*create) (struct inode *,struct dentry *, umode_t, bool);  
    int (*link) (struct dentry *,struct inode *,struct dentry *);  
    int (*unlink) (struct inode *,struct dentry *);  
    int (*symlink) (struct inode *,struct dentry *,const char *);  
    int (*mkdir) (struct inode *,struct dentry *,umode_t);  
    int (*rmdir) (struct inode *,struct dentry *);  
    ...  
};
```

Ext2对这些接口的实现

```
const struct inode_operations ext2_dir_inode_operations = {  
    .create    = ext2_create,  
    .lookup    = ext2_lookup,  
    .link      = ext2_link,  
    .unlink    = ext2_unlink,  
    .symlink   = ext2_symlink,  
    .mkdir     = ext2_mkdir,  
    .rmdir     = ext2_rmdir,  
    ...  
};
```

问题

- FAT没有inode，如何挂载到VFS？

- VFS层对上提供的接口，每个文件都有一个inode
- FAT的inode从哪里来？

- FAT的驱动需要提供inode

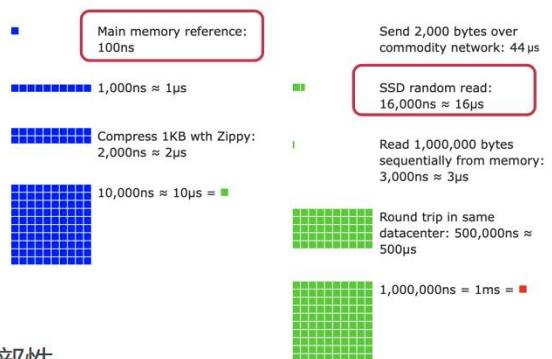
- 磁盘上的FAT并没有inode：硬盘上的数据结构
- 内存中的VFS需要inode：只在内存中的数据结构

我们在挂载的时候，VFS 会去创建对应的 inode，这样我们就可以通过挂载操作对应到 FAT 里的操作，从而能够对文件系统进行访问。FAT 驱动需要提供 inode，但是实际上在磁盘上是没有 inode 结构的。

首先，我们明确硬盘和磁盘的访问速度都是比内存慢很多的。我们的主存的访问时间是 100ns，而我们对 SSD 的访问是 16 微秒。

页缓存 (Page Cache)

- 存储访问非常耗时



- 文件访问具有时间局部性

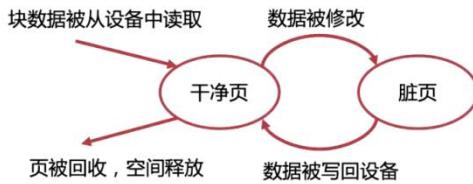
- 一些目录/文件的数据块会被频繁的读取或写入

那么我们怎么样来弥补文件和网络的时延所带来的差距呢，我们自然而然会引入 page cache 的概念。文件系统的访问是有局部性的，一些数据会被频繁读取和写入，如果我们放到内存里就可以加快速度。这样就可以提升文件系统的性能。读入以后，我们把数据暂时缓存在内存中。对缓存，我们在用户需要/定期/内存不够用的时候才会把数据写回到设备里。

页缓存

- 通过缓存提升文件系统性能

- 在一个块被读入内存并被访问完成后，并不立即回收内存
 - 将块数据暂时缓存在内存中，下一次被访问时可以避免磁盘读取
 - 在一个块被修改后，并不立即将其写回设备
 - 将块数据暂时留在内存中，此后对于该数据块的写可直接修改在此内存中
 - 定期或在用户要求时才将数据写回设备



为什么我们打开几个文件操作一段时间后，可用内存只剩一点了？因为 OS 把很多内存用来做缓存了，如果我们打开了更加消耗内存的应用，OS 就会把缓存刷回去。CTRL_S 就会触发 fsync，把文件里的数据写回到我们的 flash 里去。比如说这里有对应的 fsync、f_datasync，来更新数据和元数据。

Q: 数据不及时写回会发生什么问题？

A: 一个典型问题就是我们写了很长时间的一个文档，此时应用闪崩/电脑故障了以后，我们数据就找不回来了。

页缓存之外

存储中的每个数据结构，在内存中均有对应的结构

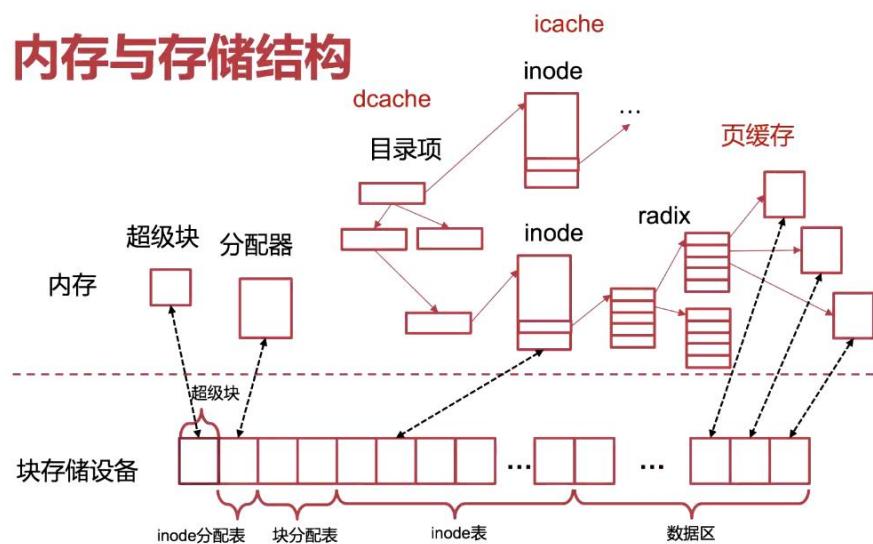
- 存储的数据页：页缓存中的内存页
- 存储中的inode：icache中的inode
- 存储中的目录项：dcache中的目录项
- 存储中的超级块：内存中的超级块结构
- 存储中的分配表：内存中的分配器

我们还有很多对应的 cache 结构。内存中也有 superblock 结构，分配表对应到内存中的分配器。

Q：为什么我们要为每个结构设计单独的缓存？我们能不能只使用页缓存呢？

A：我们之前物理内存分配的时候，我们提到过 slab（混凝土块）。比如我们造桥的时候，很多混凝土块我们已经预制好了大小，直接拉过去组装就行了。在这里就是说同一个数据结构的大小是类似的，我们去分配它就会特别高效。icache 中的 inode 和 dcache 中的目录项，大小都是固定的。

这个缓存是可以附着在页缓存之上，但是是单独管理的。这样我们就来看一下内存和存储的结构分别是怎样的，我们学完这部分之后要把这张图给记住：



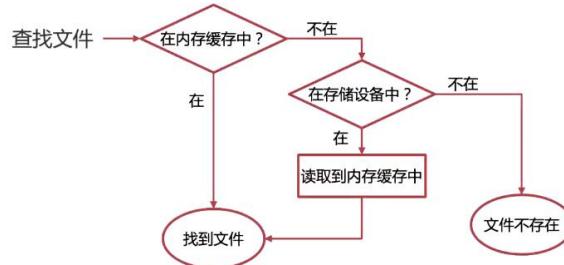
我们在块存储里有 superblock、inode-table、block table 和对应的 block。这些块对应到内存里都有对应的结构。内存里也会有对应的 inode、对应的 radix tree、对应的页缓存等。通过这种方式，经常访问的文件都可以直接在内存中操作，很少有 flush 的操作。

如果我们要访问的文件不在内存中，那么我们就去存储结构中去找。内存中的数据结构会添加一下额外的信息来帮助我们访问，在写回的时候并不会写回到 flash 里去。

缓存情况下的文件查找

由于内存大小限制，内存中缓存的数据是存储中数据的子集

当要访问的数据不在内存中时，会从存储中读取并构造内存中相应的对象



我们来看一下整个存储栈在 OS 里到底长什么样子。一般而言，我们通过 `libc` 来访问系统调用，首先是被 VFS 接管，然后去查找对应的挂载点，调用具体的文件系统的实现。这些文件系统之下，可能有公共的对于块的管理和抽象。这些缓存写回就会涉及到 I/O 的调度，把缓存刷到 SSD 中。通过 I/O 的调度器进行调度，这个过程中，如果我们操作了同一个 block，我们可能会把一些 I/O 的请求合并，从而避免了对 flash 一直的写，提升了寿命。然后再是我们的设备驱动和设备进行通讯，写到固态硬盘和机械硬盘里去。

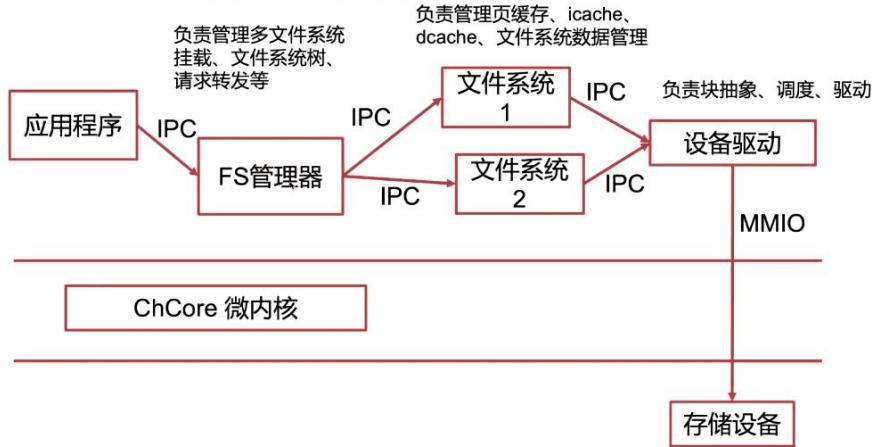
宏内核(Linux)中的存储栈



在 ChCore 也有一个对应的文件和存储的栈。因为我们支持了应用间通讯，所以应用程序会通过一个 IPC 访问文件系统管理器，它同样会对应到具体的文件系统，再去访问到对应的设备驱动和存储设备。

在这个过程中大体是和 Linux 类似的，但是简单一些。

ChCore中的文件与存储结构



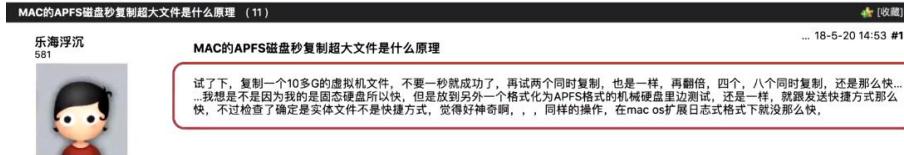
到目前为止，我们介绍了页的缓存和存储栈的结构。

文件系统高级功能

接下来我们简单介绍一下高级的功能。

我们先介绍一个网络上的问题，简而言之就是 MAC 的复制巨大文件的速度非常快。

▶ 文件复制



这背后到底是什么原因呢？文件复制主要是如下两种方法：

read/write

```
$ cp A B
```

1. 打开文件A
2. 创建并打开文件B
3. 从A中读出数据到buffer
4. 将buffer中的数据写入B
5. 重复3、4直到文件A被读完

mmap

```
$ cp A B
```

1. 打开文件A
2. 获取A的大小为x
3. 创建并打开文件B
4. 改变B的大小为x(fallocate/ftruncate)
5. 将A和B分别mmap到内存空间
6. memcpy

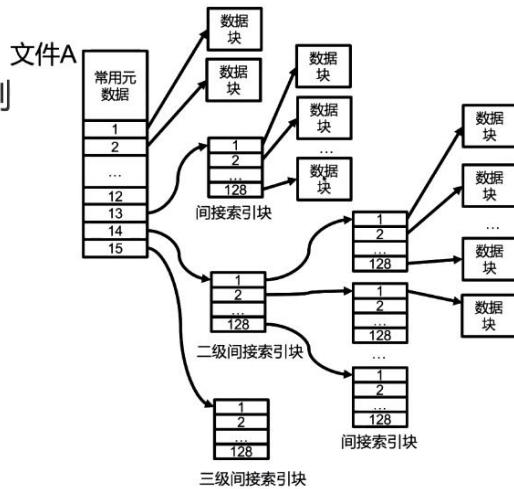
但是这两种方法都做不到复制 10G 的文件不到 1 秒完成。我们一般的理解是文件越大，复制耗时越长。

克隆

背后的原因和我们 fork 一个进程是差不多的，fork 的时候，我们是使用写时复制。而 APFS 也是采样 clone 的方式，只复制了关键的元数据，其他数据也是使用 copy-on-write 的方式进行共享的。

克隆

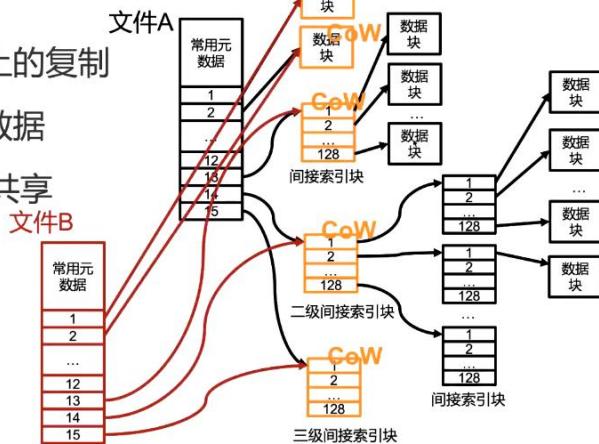
- 文件系统层面上的复制
- 只复制关键元数据
- 其他部分CoW共享



我们 copy 的时候，只是复制了元数据和一些间接索引块。

克隆 (Clone)

- 文件系统层面上的复制
- 只复制关键元数据
- 其他部分CoW共享



Q: 这种方法有什么缺点呢？

A: 比如我们为了备份一个文件，我们 copy 了一份。结果我们存储文件 A 的磁盘坏了，结果我们两个文件都坏了。这就是 clone 功能的缺点。

快照 (Snapshot)

它同样也是使用了 CoW 的方式，我们可以把树状文件系统的根 copy 一份保存，树根一下的节点我们就设置为 CoW。比如我们做了一些修改之后，我们想回退回去，这也是一个很好的方式。

快照 (Snapshot)

- 同样使用CoW
- 对于基于inode表的文件系统
 - 将inode表拷贝一份作为快照保存
 - 标记已用数据区为CoW
- 对于树状结构的文件系统
 - 将树根拷贝一份作为快照保存
 - 树根以下的节点标记为CoW

稀疏文件

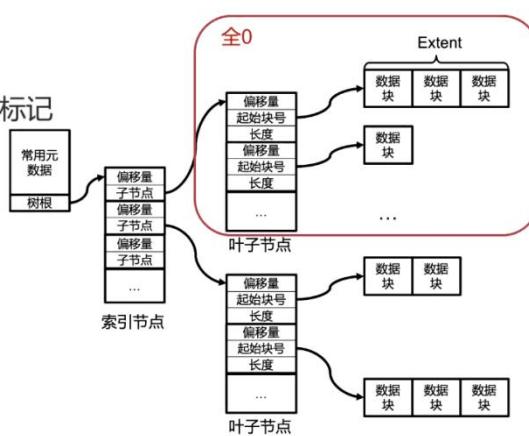
还有一个是稀疏文件，我们在做课程实验的时候，会使用到虚拟机，虚拟机会有一个 virtual disk，比如我们开始创建的时候创建了 16G。这是否代表我们直接在硬盘中划分出了 16G 的空间给虚拟机使用呢？实际上并不是。虚拟磁盘块在我们主机中是一个文件，在还没有使用的时候，大部分数据为 0。

稀疏文件

- 一个文件大部分数据为0，则为稀疏文件
 - 如虚拟机镜像文件
- 稀疏文件中大量的0数据，白白消耗空间

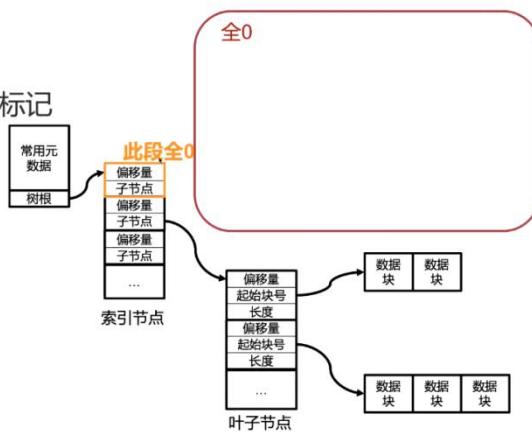
稀疏文件

- 在索引中增加标记
- 删除此全0块



稀疏文件

- 在索引中增加标记
- 删除全0块



在我们树状的索引结构中，可以通过特殊的标记来节省大量全零的空间。

文件系统的一些其他高级功能

- 加密
- 压缩
- 去重
- 数据和元数据校验
- 配额管理 (QoS)
- 软件 RAID
- 多设备管理
- 子卷
- 事务 (Transaction)

我们默认是 OS 里是不加密的，但是公司里的电脑就会使用到加密功能，比如 Windows 有 bitlock 的功能。这样我们写到磁盘里的数据默认是加密的。

软件 RAID 就是做冗余，如果有磁盘块坏了是可以恢复回来的。

压缩也是常见的方法。比如华为手机在 19 年的时候使用了 EROFS 文件系统，这样就把空间省出来了。原理就是把只读文件压缩存储到磁盘里去，这样可以减少空间占用和写带宽。

多设备的管理就是让文件系统管理多种设备。

去重就是在企业存储里，也是文件系统里重要的方式。Data Domain 公司做的事情就是去重，企业经常会做备份存储，也就是没过一段时间把数据做备份。但是大量的备份之间是有重复数据的，可以通过去重的方式减少对磁盘的占用。

文件系统的多种形式

GIT

我们很多日常生活中的东西，看起来不是文件系统，但是采取了文件系统的理念。GIT 本质上就是一个内存寻址的文件系统。这个实际上就是文件系统之上的文件系统。

GIT : 内容寻址文件系统

- 表面上GIT是一个版本控制软件
- 但实际上GIT是一个**内容寻址**的文件系统！
- 其核心是一个键值存储
 - 值：加入GIT的数据
 - 键：通过数据内容算出的40个字符SHA-1校验和
 - 前2个字符作为子目录名，后38个字符作为文件名
 - 所有对象均保存在.git/objects目录中（文件内容会被压缩）
- 是一个“**文件系统之上的文件系统**”

首先它有一个 BLOB 对象，对应到文件系统中的文件。树对象对应的是目录。我们提交了一个对象以后，我们就可以通过 git log 看到对应的名字。

GIT对象与文件系统

- BLOB对象**：对应文件系统中的文件

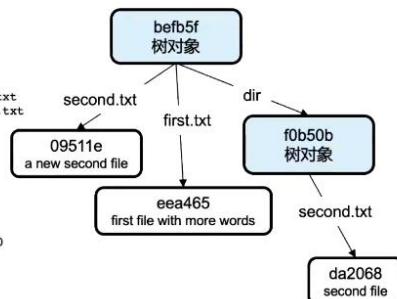
first file with more words

- 树对象**：对应文件系统中的目录

```
040000 tree f0b50bef52478d42d68ff21914c430d8148f23cd dir  
100644 blob eea465ab73df5cd24379ef0d61da949b3b5138ea first.txt  
100644 blob 09511ef0b04fb20dd32f3cb090f9c6db49cd2627 second.txt
```

- 提交对象**

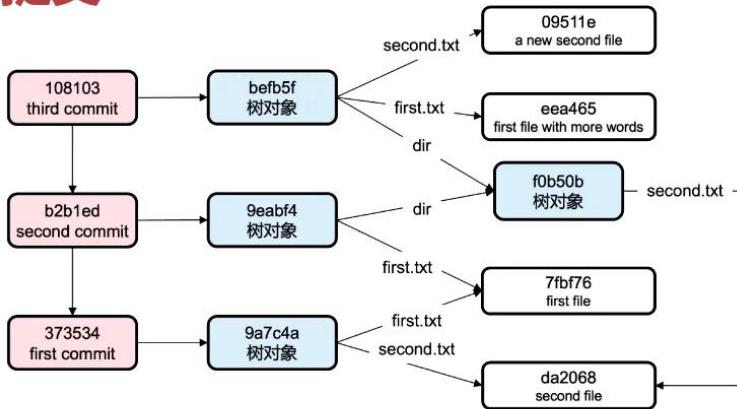
```
tree befb5f375306393e542026391d95104579da48ce  
parent b2b1eda1c44d0a2cb10ad7f522ae78f9ded7e95e  
author IPADS <ipads@ipads.se.sjtu.edu.cn> 1587414317 +0800  
committer IPADS <ipads@ipads.se.sjtu.edu.cn> 1587414317 +0800  
third commit
```



接下来我们来看 git 提交的时候发生的具体的事情。第一次 commit 后有一个 first.txt 和 second.txt。第二次 commit 的时候，first.txt 没有变，但是 second.txt 被移动到了 dir 文件夹下。第三次 commit 的时候，在根目录下又创建了一个新的 second.txt 文件，并且也更改了 first.txt 的内容。

checkout 其实就是把 first.txt(eea465)替换为了 first.txt(7fbf76)这个文件。这是在文件树里发生的操作。

GIT的提交



小问题：在执行 `git checkout b2b1ed -- first.txt` 时都发生了什么？

SQLite

我们手机里的通讯录就是通过 SQLite 来实现的。它对于小文件的修改，读写可以比文件系统快 35%。

► SQLite : 文件系统的竞争者

The screenshot shows the SQLite homepage with the following content:

- SQLite** logo
- Small. Fast. Reliable.**
Choose any three.
- 35% Faster Than The Filesystem**
- Home About Documentation Download License Support Purchase Search**
- ▶ Table Of Contents**
- 1. Summary**
- A callout box highlights performance statistics:
 - SQLite reads and writes small blobs (for example, thumbnail images) **35% faster** than the same blobs can be read from or written to individual files on disk using `fread()` or `fwrite()`.
 - Furthermore, a single SQLite database holding 10-kilobyte blobs uses about 20% less disk space than storing the blobs in individual files.
 - The performance difference arises (we believe) because when working from an SQLite database, the `open()` and `close()` system calls are invoked only once, whereas `open()` and `close()` are invoked once for each blob when using blobs stored in individual files. It appears that the overhead of

SQLite 用了数据库的理念，还是很多操作还是有点像文件系统。

SQLite : 文件系统的竞争者

- 表面上SQLite是一个数据库
- 但实际上SQLite也可以是一个文件系统！
- 其核心还是一个数据库
 - 在关系型数据库的表中，记录文件名和BLOB类型文件数据
 - 通过查找文件名，获取对应文件数据
 - 存储大量小文件
- 文件系统里的文件里的文件系统里的文件

SQLite引发的思考🤔

- 对于小文件，为何一般文件系统不如SQLite效率高？
- 文件系统如何针对小文件进行改进？
- 还有哪些针对小文件特殊处理的场景？

A1：打开文件慢是因为我们要遍历文件树查找，使得查找效率不高。而 SQLite 可以用文件名做 key，我们直接查找就可以查到。

A2：优化存储目录结构，比如用小型数据库做文件的索引。比如 Google 的文件系统就使用了 LevelDB 做文件的索引。

A3：html 里有很多小图片，其实都是一些拼在一起的。也是有一些新的优化的机会。

FUSE：用户态文件系统框架

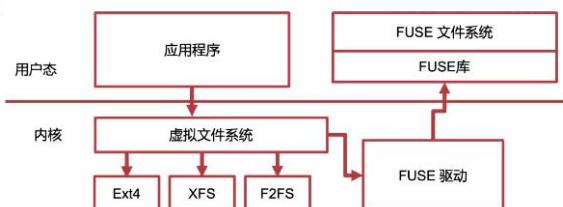
我们刚才介绍的文件系统都是在内核态的，因为在内核态实现起来比较高效，`syscall`进入内核态以后，Linux 在同一个地址空间中做文件系统、VFS、I/O 的操作。FUSE(Filesystem in Userspace)是一个典型的用户态文件系统框架，可以方便我们实现各种各样的文件系统的功能。

具体而言，VFS 创建 FUSE 的驱动（认为是一个文件系统的实例）。

FUSE用户态文件系统框架

为什么要用户态文件系统？

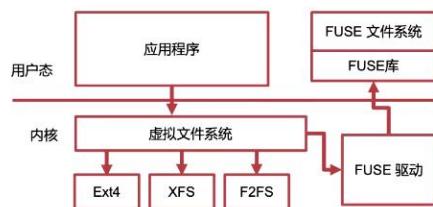
- 快速试验文件系统新设计
- 大量第三方库可以使用
- 方便调试
- 无需担心把内核搞崩溃
- 实现新功能



以下是 FUSE 的基本流程：

FUSE基本流程

1. FUSE文件系统向FUSE驱动注册（挂载）
2. 应用程序发起文件请求
3. 根据挂载点，VFS将请求转发给FUSE驱动
4. FUSE驱动通过中断、共享内存等方式将请求发给FUSE文件系统
5. FUSE文件系统处理请求
6. FUSE文件系统通知FUSE驱动请求结果
7. FUSE驱动通过VFS返回结果给应用程序



Q: 从这个过程中，我们可以看出 FUSE 有什么问题？

A: 我们需要到内核态再到用户态绕一圈，并且这个过程中还会涉及到大量的数据的 copy，所以唯一的问题就是慢。

FUSE API

• 底层API

- 直接与内核交互
- 需要负责处理inode和查找等操作
- 需要处理内核版本等差异

• 高层API

- 构建于底层API之上
- 以路径名为参数
- 无需关注inode、路径和查找

比如说我们有一个 hello 文件系统，这是它的 `lookup`、`read` 的操作。建立连接的时候，会把操作表传过去进行挂载。

```
1. static void hello_ll_lookup(fuse_req_t req, fuse_ino_t parent, const char *name) {
2.     struct fuse_entry_param e;
3.     if (parent != 1 || strcmp(name, hello_name) != 0)
4.         fuse_reply_err(req, ENOENT);
5.     else {
6.         memset(&e, 0, sizeof(e));
7.         e.ino = 2;
8.         e.attr_timeout = 1.0;
9.         e.entry_timeout = 1.0;
10.        hello_stat(e.ino, &e.attr);
11.        fuse_reply_entry(req, &e);
12.    }
13. } // lookup : 根据父目录 ino 和文件名, 返回查找结果
14. static void hello_ll_read(fuse_req_t req, fuse_ino_t ino, size_t size,
15.                           off_t off, struct fuse_file_info *fi) {
16.     (void) fi;
17.     assert(ino == 2);
18.     reply_buf_limited(req, hello_str, strlen(hello_str), off, size);
19. } // read: 根据 ino, 返回请求
20. static const struct fuse_lowlevel_ops hello_ll_oper = {
21.     .lookup      = hello_ll_lookup,
22.     .getattr     = hello_ll_getattr,
23.     .readdir     = hello_ll_readdir,
24.     .open        = hello_ll_open,
25.     .read        = hello_ll_read,
```

```

26. }; // 提供函数指针作为回调函数
27.
28. int main() {
29.     ...
30.     se = fuse_session_new(&args, &hello_ll_oper,
31.                           sizeof(hello_ll_oper), NULL); // 建立连接
32.     fuse_session_mount(se, opts.mountpoint); // 挂载
33.     ret = fuse_session_loop(se); // 等待请求和回调
34. }
35. // https://libfuse.github.io/doxygen/example\_2hello\_11\_8c.html

```

表 2 FUSE 底层 API

FUSE高层API

```

static int hello_read(const char *path, char *buf, size_t size, off_t offset,
                      struct fuse_file_info *fi)
{
    const char *content = "Hello World!\n";
    memcpy(buf, content + offset, MIN(size, strlen(content) - offset));
    return size;
} // read : 根据path，在buf中填入数据，返回写入字节数

static const struct fuse_operations hello_oper = {
    .init = hello_init,
    .getattr = hello_getattr,
    . readdir = hello_readdir,
    .open = hello_open,
    .read = hello_read,
}; // 提供函数指针作为回调函数

int main()
{
    ...
    ret = fuse_main(args(argc, args.argv, &hello_oper, NULL)); // 挂载、注册并等待请求和回调
    ...
}

```

用户态出 Lab 是比较合适的，还有就是用 ssh 挂载远端的目录到本地。用 FUSE 可以做文件系统沙箱，限制应用读取公共目录下的文件。

现实中，FUSE能用来做什么？

- 出Lab！
- SSHFS (用ssh挂载远端目录到本地)
- Android Sandbox
- GMailFs (以文件接口收发邮件)
- WikipediaFS (用文件查看和编辑Wikipedia)
- 网盘同步
- 分布式文件系统 (Lustre、GlusterFS等)
- *Everything is a file; can everything be done with a filesystem?*

文件系统崩溃一致性

文件系统崩溃一致性要解决的问题如下，文件系统不一致可能会导致文件损坏、OS 无法启动等严重问题。

文件系统的崩溃一致性

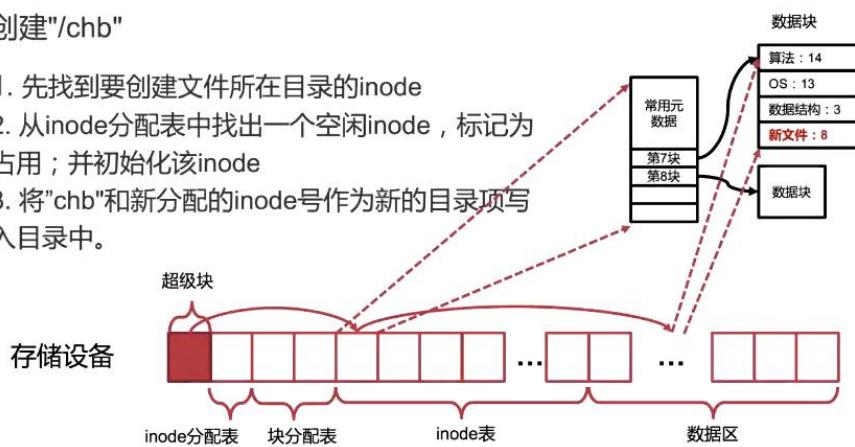
- 文件系统中保存了多种数据结构
- 各种数据结构之间存在依赖关系与一致性要求
 - inode中保存的文件大小，应该与其索引中保存的数据块个数相匹配
 - inode中保存的链接数，应与指向其的目录项个数相同
 - 超级块中保存的文件系统大小，应该与文件系统所管理的空间大小相同
 - 所有inode分配表中标记为空闲的inode均未被使用；标记为已用的inode均可以通过文件系统操作访问
 -
- 突发状况（崩溃）可能会造成这些一致性被打破！

我们回顾一下在根目录下创建/chb 这个文件的流程。

回顾：文件的创建

创建"/chb"

1. 先找到要创建文件所在目录的inode
2. 从inode分配表中找出一个空闲inode，标记为占用；并初始化该inode
3. 将"chb"和新分配的inode号作为新的目录项写入目录中。



我们假设就有这三步。第一种情况下就是指针指向了一个空的地方，会出现覆盖。（本来以为写了自己的文件，实际上写了别人的文件）；第二种情况下，没有把 inode 标记为占用，也会出现上述的问题。值得注意的是这三个操作可能在内存里都完成了，但是没有刷新到 flash 里去。

考虑内存缓存下的崩溃情况

创建"/chb"的修改包括：

1. 标记inode为占用
2. 初始化inode
3. 将目录项写入目录中

两种常见情况：

3. 将目录项写入目录中

2. 初始化inode
1. 标记inode为占用

3. 将目录项写入目录中

2. 初始化inode

1. 标记inode为占用

目录项指向了未分配/未初始化的inode

常见的有八种情况，

考虑内存缓存下的崩溃情况

创建"/chb"的修改包括：

1. 标记inode为占用
2. 初始化inode
3. 将目录项写入目录中

崩溃随时可能发生！

共有8种情况： {}, // 没有操作被持久化

{1}
{2}
{3}
{1, 2} (与{2, 1}相同)
{1, 3}
{2, 3}
{1, 2, 3}

注意：此处的创建文件还未考虑修改时间戳、写入新目录项需要分配新的数据块、修改超级块中的统计信息等情况。考虑后情况会更复杂！

思考时间

- 手机和笔记本电脑等设备有电池，是否还需要保证文件系统崩溃一致性？
- 数据中心一般会配有UPS（不间断电源），是否还需要保证文件系统崩溃一致性？

A: 都是需要的，有电池的情况下也会有其他的故障。

首先我们系统得正常启动，其次我们打开一个 a.txt 得是正确的內容，不是乱码也不是别人的文件。凡是我们按了 ctrl+s 的文件，修改应该都在。

崩溃一致性：用户期望

重启并恢复后...

1. 维护文件系统数据结构的内部的不变量
例如，没有磁盘块既在free list中也在一个文件中
2. 仅有最近的一些操作没有被保存到磁盘中
例如：我昨天写的OS Lab的文件还存在
用户只需要关心最近的几次修改还在不在
3. 没有顺序的异常

```
$ echo 99 > result ; echo done > status
```

我们的磁盘在出错以后就会停下，不会做别的操作。

一些（简化的）假设

- 磁盘是fail-stop, 磁盘会忠实执行文件系统下发的命令，不会多做也不会少做
- 磁盘可能不会执行最近的几次操作
- 保障：磁盘不会写飞(wild writes)

我们要保证一致性，也要保证顺序的问题。

为什么保证崩溃一致性这么困难呢？

崩溃 = halt/restart CPU

let disk finish current sector write, assume no h/w damage, no wild write to disk

目标：自动恢复

Can fs always make sense of on-disk **metadata** after restart?

Given that the crash could have occurred at **any point**?

例子：

Crash during mkdir, leave directory without . and ..

Crash during free blocks

那么怎么保持崩溃一致性呢？一般是两种方法离线恢复（启动的时候检查磁盘有没有问题）和在线恢复（运行过程中检查不一致性）。

方法：在线与离线恢复

离线恢复

文件系统检查工具, 例如 : windows中的chkdsk , Linux中的fsck
例如, ext3

在线恢复

运行过程中, 检查一些重要的不一致性
例子, ext4 (同时也使用fsck, 但是非常简单)

原子性就是要么都可见、要么都不可见。但是一般的文件系统, 三点都不保证。

文件系统操作所要求的三个属性

`creat("a"); fd = creat("b"); write(fd,...); crash`

持久化/Durable: 哪些操作可见

a和b都可以

原子性/Atomic: 要不所有操作都可见, 要不都不可见

要么a和b都可见, 要么都不可见

有序性/Ordered: 按照前缀序(Prefix)的方式可见

如果b可见, 那么a也应该可见

我们下节课会介绍崩溃一致性的保障方法。

崩溃一致性保障方法

- Soft updates
 - 日志
 - 写时复制
- } 原子更新技术

2022/4/12

这六种情况里, 有一类是没有问题 (benign, 良性) 的。但是依然其他五种情况是非常危险的。

Review : 创建文件时崩溃的6种情况

	文件系统结构			是否影响使用	典型问题
	inode	位图	inode 结构	目录项	
情况 1	持久	未持久	未持久	否	空间泄漏
情况 2	未持久	持久	未持久	否	无
情况 3	未持久	未持久	持久	是	信息错乱
情况 4	未持久	持久	持久	是	信息错乱
情况 5	持久	未持久	持久	是	信息错乱
情况 6	持久	持久	未持久	否	空间泄漏

应用程序对文件系统有这三个属性的要求，否则对上层语义产生影响。

Review : 文件系统操作要求的三个属性

`creat("a"); fd = creat("b"); write(fd,...); crash`

持久化/Durable: 哪些操作可见

a和b都可以

原子性/Atomic: 要不所有操作都可见，要不都不可见

要么a和b都可见，要么都不可见

有序性/Ordered: 按照前缀序(Prefix)的方式可见

如果b可见，那么a也应该可见

要保证崩溃一致性有一系列的方法

日志 (Journaling)

我们之前在 CSE 中讲过，当我们要保证 all-or-nothing 的时候，日志和 shadow-copy 是两种方法。既然发生崩溃之后，数据要 all-or-nothing，那么我们就把要做的操作记到额外的空间中，这就会有 commit-point 的概念，就是日志写完前后的状态。这种方法也叫 redo log。在我们的文件系统里，基本上我们认为大家了解 redo log 即可。

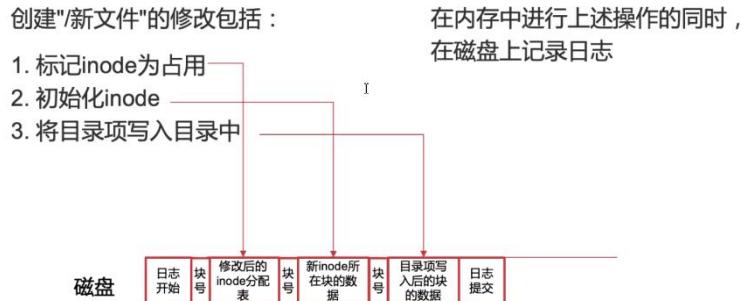
日志

- 在进行修改之前，先将修改记录到日志中
- 所有要进行的修改都记录完毕后，提交日志
- 此后再进行修改
- 修改之后，删除日志

磁盘上有一块空间是日志，它可以放在磁盘的末尾（整个磁盘只有 1 个日志），也可以以文件的形式存在（可以在一个磁盘里有多个日志，好处就是增加了一些灵活性，我们甚至可以对不同的目录做不同的日志）。NTFS 中的 MFT 里前 16 个保留文件中有一个就是日志。

日志里像一个数组一样记录块号+修改的数据，最后再是一个 commit。我们可以写 4K，也可以把我们修改的数据写进去，这样就可以减少日志的大小。

日志



Q: 既然我们后面有 commit，为什么前面还要记录多少个日志呢？日志开始这里头上的元数据到底有什么用呢？

A: 当我们把日志写回磁盘上之后，它其实就没用了。这个时候，我们就可以修改日志的开头，把里面的“当前有多少个日志还没有写”，设置为 0。这样后面的数据都不用看了。



在"日志提交"写入存储设备之前崩溃

- 恢复时发现日志不完整，忽略日志，"/新文件"未被创建

在"日志提交"写入存储设备之后崩溃

- 将日志中的内容，拷贝到对应位置，"/新文件"被创建成功

在写 flush 的时候，最大的问题就是性能会很差。如果我们要创建很多文件，每次文件都要 flush 两次磁盘（日志提交前 flush 和日志提交后 flush），可能带来 10~20 毫秒的时延。如果我们创建一个文件就要 10ms，那么 1 秒只能创建 100 个文件。

所以写日志本身一旦到工程实现上，依然没有解决磁盘的 flush 太多的问题。还有一个问题是每个修改都要 copy 新数据到日志，再写回到该写的地方，这样所有操作都需要写两遍。而且同一个 inode 的操作我们要记录多次日志。

问题

此种方法有什么问题？

- 问题1. 每个操作都写磁盘，内存缓存优势被抵消
- 问题2. 每个修改需要拷贝新数据到日志
- 问题3. 相同块的多个修改被记录多次

.....



利用内存中的页缓存提高日志性能

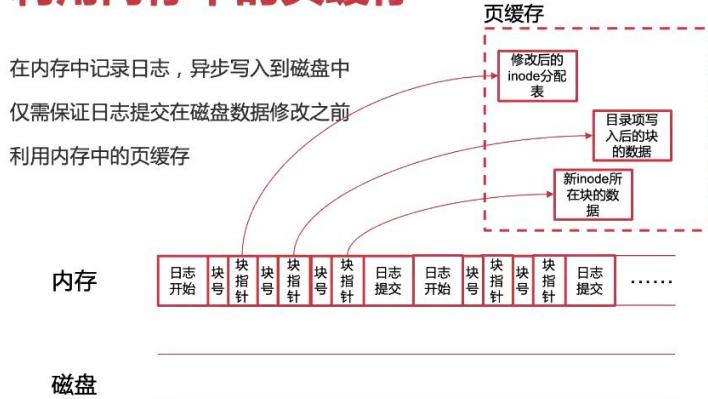
那么 OS 到底是怎么去做的呢？我们就需要利用到我们的内存了。我们希望把日志记录

里，异步地写到磁盘里去。我们是需要知道数据是不能写回的，如果要把数据写回，我们先要把日志写回。所以此时的文件系统变得更复杂，需要追踪日志有没有写回。

但是好处也是显而易见的，可以利用内存中的 page cache，整个性能的读写性能都会快很多。

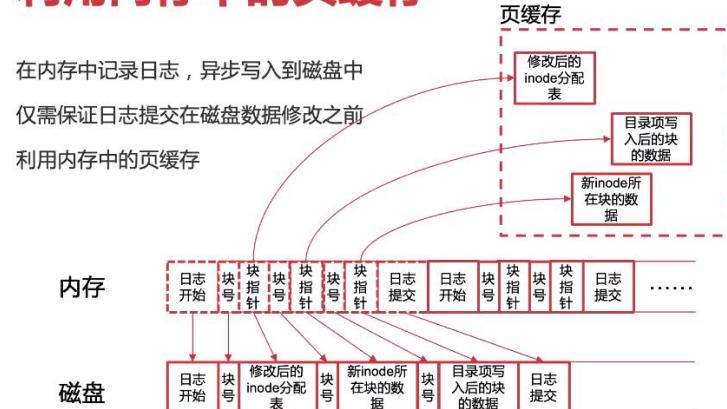
下图中，块指针指向了修改后的 inode 的分配表。为什么 page cache 有它呢？因为如果我们要修改 free inode bitmap，当我们要去读它的时候，我们要去读磁盘。所有读磁盘的操作都会产生一次页缓存的填充，所以就会有 hit。

利用内存中的页缓存



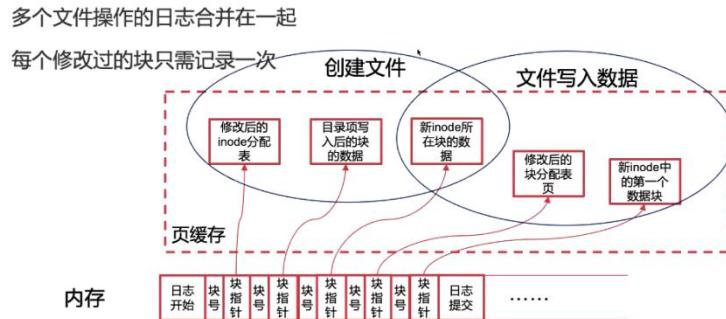
一旦当我们的 page cache 要写磁盘的时候，我们先写到磁盘上日志的区域，再写到对应的数据区域。我们充分利用内存中的页缓存机制，就可以做到优化。

利用内存中的页缓存



批操作就是可以把磁盘写合并在一起（写吸收）。缺点也很明显，我们的时延更长，但是时延在这里不是特别关键，因为一旦写了日志我们不需要等待日志写完。一旦发生了崩溃，我们可能会丢掉大量的 transaction。于是我们不能让 batch 变得无限大，不能缓存一个小时的数据再写到磁盘上。

批量处理日志以减少磁盘写



日志隔一段时间就会触发一次，ext4 默认的是每 5 秒触发一次。

日志提交的触发条件

- 定期触发
 - 每一段时间（如5s）触发一次
 - 日志达到一定量（如500MB）时触发一次
- 用户触发
 - 例如：应用调用fsync()时触发

为了去讲日志怎么用，我们来看 Linux kernel 里的日志系统是怎么实现的，它叫做 **Journal Block Device 2**。它可以以文件的形式保存日志。**transaction** 就是多个合在一起的原子操作。在 CSE 中，我们讲 **transaction** 之间是可以保证 **before-or-after** 的，而在里相同的概念的 **handle**。

比如说，我们在 5 秒中创建了 10 个文件，我们就有 10 个 **handle**，合在一起变成了一个 **transaction**，再写到磁盘里去。

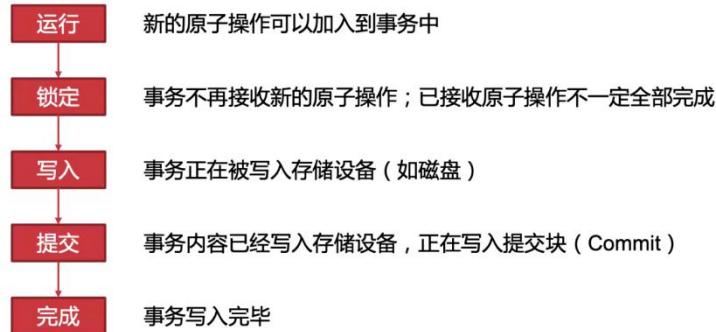
JBD2

Case : Linux中的日志系统JBD2

- Journal Block Device 2
- 通用的日志记录模块
 - 日志可以以文件形式保存
 - 日志也可以直接写入存储设备块
- 概念
 - Journal : 日志，由文件或设备中某区域组成
 - Handle : 原子操作，由需要原子完成的多个修改组成
 - Transaction : 事务，多个批量在一起的原子操作

`transaction` 可以认为是一个麻袋，麻袋的口打开之后，上面的 `syscall` 变成一个个 `handle` 落在麻袋里。等到 5 秒钟之后，麻袋口一扎，准备异步写入磁盘，并且打开一个新的麻袋去接下 5 秒钟的 `handle`。所有数据都写完之后，提交一个 `commit`，在这个过程中，`handle` 本身被放在一个更大的 `transaction` 里 `commit`。`transaction` 和具体的操作是解耦的，而是每 5 秒中的对磁盘的写操作打包在一起。

JBD2事务的状态



我们来看一些接口，首先在挂载文件系统的时候，JBD2 是要初始化的。如果已经有了日志（之前发生了 `crash`），就可以去加载做一些额外的恢复。

后台有一个守护进程，每个 5 秒得到一个 `transaction` 去处理。

JBD2部分接口和使用方法

系统调用处理：

文件系统挂载时：	<code>// 不使用日志时的创建文件</code>
<code>journal_t journal;</code>	
<code>// 初始化日志系统 (日志存在文件中)</code>	<code>// 1. 标记inode为占用</code>
<code>journal = jbd2_journal_init_inode(inode)</code>	<code>// bh: buffer_head 对应存储设备中的最小访问单元</code>
<code>// 读取并恢复已有日志 (如果存在)</code>	<code>bitmap_bh = read_inode_bitmap(sb, group)</code>
<code>jbd2_journal_load(journal)</code>	<code>set_bit(ino, bitmap_bh->b_data)</code>
后台进程：	<code>// 2. 初始化inode</code>
<code>while (sleep_5s()) {</code>	<code>inode_bh = get_inode_bh(sb, ino)</code>
<code> // 提交事务和回收日志空间 (并开始新的事务)</code>	<code>init_inode(inode_bh)</code>
<code> jbd2_journal_commit_transaction(journal)</code>	<code>}</code>
文件系统卸载时：	<code>// 3. 将目录项写入目录中</code>
<code>// 释放日志系统</code>	<code>data_bh = get_data_page(dir_inode)</code>
<code>jbd2_journal_destroy(journal)</code>	<code>add_dentry_to_data(page, filename, ino)</code>

使用 JBD2，我们代码的逻辑是什么呢？一开始我们要得到一个 `handle`，在每次写之前，就要调用 `jbd2_journal_get_write_access`(打开写权限)。也就是通知马上要修改一个东西了。在写完之后，它会告诉你写完了。这样整个过程中写了什么东西会被记录下来，同样写 `inode` 和目录是一样的。

```

系统调用处理：
handle_t handle;
// 原子操作：创建新文件
handle = jbd2_journal_start(journal, nblocks=8)

// 1. 标记inode为占用
// bh: buffer_head 对应存储设备中的最小访问单元
bitmap_bh = read_inode_bitmap(sb, group)
jbd2_journal_get_write_access(handle, bitmap_bh)
set_bit(ino, bitmap_bh->b_data)
jbd2_journal_dirty_metadata(handle, bitmap_bh)

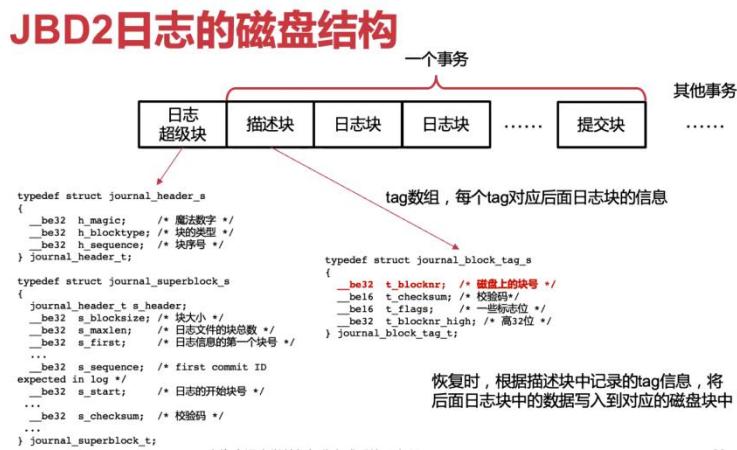
// 2. 初始化inode
inode_bh = get_inode_bh(sb, ino)
jbd2_journal_get_write_access(handle, inode_bh)
init_inode(inode_bh)
jbd2_journal_dirty_metadata(handle, inode_bh)

// 3. 将目录项写入目录中
data_bh = get_data_page(dir_inode)
jbd2_journal_get_write_access(handle, data_bh)
add_dentry_to_data(page, filename, ino)
jbd2_journal_dirty_metadata(handle, data_bh)

jbd2_journal_stop(handle) // 结束原子操作

```

JBD2 日志的磁盘结构如下：

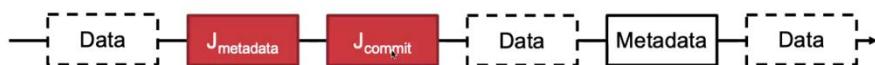


我们之前也提过 JBD2 有三种日志模式：

Ext4用JBD2实现的三种日志模式

Writeback Mode : 日志只记录元数据

最快，但是一致性最差！



Ordered Mode : 日志只记录元数据+数据块在元数据日志前写入磁盘 **默认模式**



Journal Mode : 元数据和数据均使用日志记录 **一致性最好，但数据写入两次！**



思考一下：三种模式各自有何问题和优势？

第一种方法虽然最快，但是一致性很差，因为很容易出现元数据和数据之间的不一致。Journal Mode 是最好的，但是所有数据都要写两次。

Ordered Mode 有时候可能出现无法回退的问题。比如我们把 64K 的文件新增到 128K，这样新增的 64K 数据写到一半的时候发生了 crash，那么原来的 64K 数据还在。但是如果我们是覆盖 8K 的内容，改了前 4K 的数据之后发生了 crash，这种情况下是不能回退的。用户就会看到一个一半新一半旧的数据。

Ordered Mode

- 权衡一致性和性能

- 数据的数量大，只需要写入一次
- 元数据的数量少，写入两次相对可接受

- 可能出现的问题

- 数据只有一份，若出现问题无法回退（all-or-nothing）
- 部分情况下，一致性还是可以保证的（如新增数据时）
- 部分情况下，数据会丢失，但元数据依然可以保证一致性

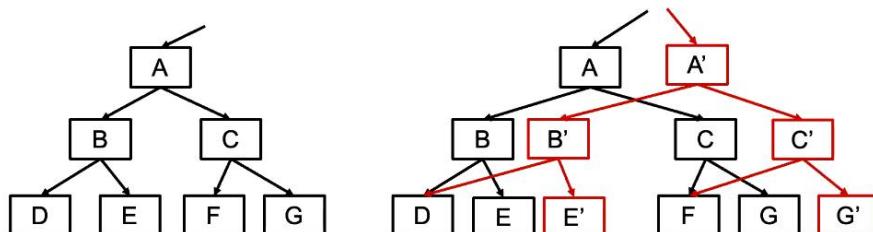
Q: 我们为什么要覆盖掉数据呢？我们是不是能针对这种情况，做一些改善？

A: 我们之前提过 multi-version concurrency control (MVCC)，既然我的数据不希望写两份，我们就把新数据写到一个额外的位置，这样新数据和旧数据都是在的，数据有两个版本。我们把元数据里指向旧数据的指针切到新数据即可。并且我们还可以保证原子性，这就是我们接下来要讲的写时复制。

写时复制 (Copy-on-Write)

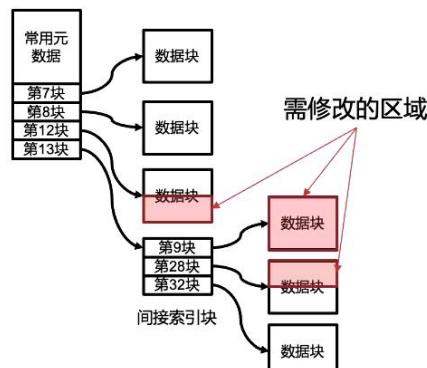
我们最早提到 CoW 的时候，是进程在 `fork` 的时候，可以把页表都 CoW 一份。在磁盘上我们也可以做到类似的操作。

- 在修改多个数据时，不直接修改数据，而是将数据复制一份，在复制上进行修改，并通过递归的方法将修改变成原子操作
- 常用于树状结构

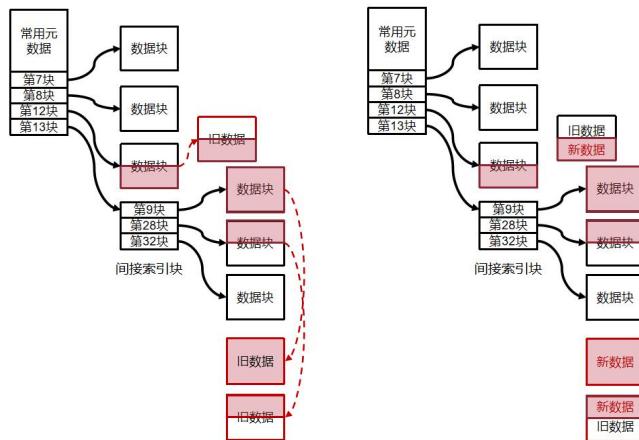


文件中的写时复制

- 文件数据散落在多个数据块内
- 使用日志：数据需要写两遍
- 写时复制保证多个数据块原子更新



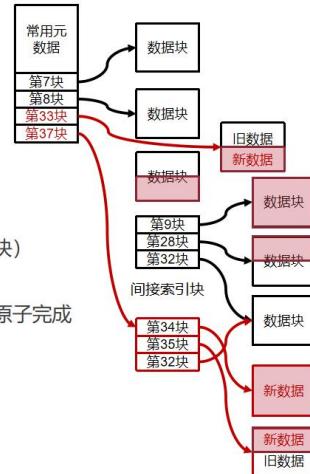
我们把需要修改的块单独 copy 一份出来，写上新的数据。



然后我们再向上递归，比如我们原先的指针是指向旧数据的，现在我们让它指向新数据。我们整个间接索引块也需要更新。因为文件系统可以以 4K 的粒度做原子修改，我们最终修改 33、37 对应的 inode 这个操作是原子的。修改前我们读到的是旧数据，而修改后读到的就是新的数据。

文件中的写时复制

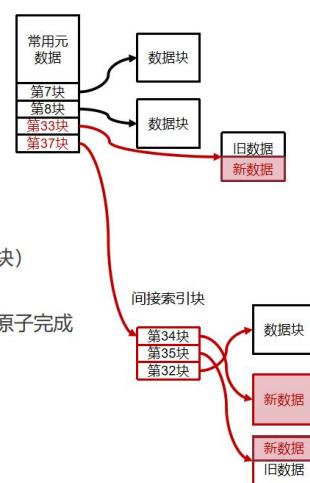
- 文件数据散落在多个数据块内
- 使用日志：数据需要写两遍
- 写时复制保证多个数据块原子更新
 - 将要修改的数据块进行复制（分配新的块）
 - 在新的数据块上修改数据
 - 向上递归复制和修改，直到所有修改能原子完成
 - 进行原子修改



最后一步，因为 inode 的修改已经落盘了。我们把旧数据回收掉。

文件中的写时复制

- 文件数据散落在多个数据块内
- 使用日志：数据需要写两遍
- 写时复制保证多个数据块原子更新
 - 将要修改的数据块进行复制（分配新的块）
 - 在新的数据块上修改数据
 - 向上递归复制和修改，直到所有修改能原子完成
 - 进行原子修改
 - 回收资源



有了写时复制之后，既没有多写数据，又保证了一致性。

思考时间 😊

- 对于文件的修改，写时复制一定比日志更高效吗？
- 写时复制和日志各自的优缺点有哪些？
- 能否只用写时复制来实现一个文件系统？

A1: 写时复制的一个非常明显的缺点就是它的粒度。对于一个磁盘来说，它必须把整个 `block copy` 过来。如果我们写数据只需要写 4 个 `byte`，我们也需要 `copy 4K`。而日志可以只记录这 4 个 `byte`，写的内容会少一些。

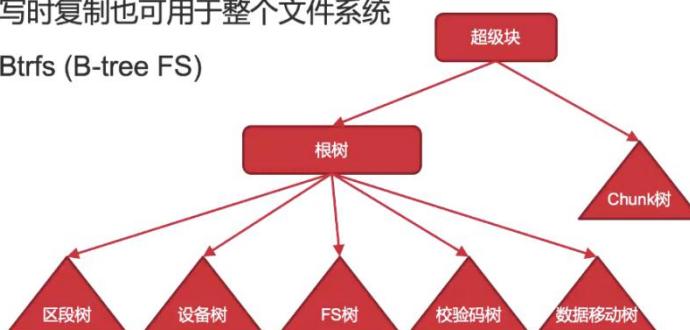
A2: 写时复制需要对文件系统做修改，日志也是要做修改的。相对来说，写时复制在树状结构的前提之下，去修改比较多的数据，写时复制是有优势的，因为我们数据只需要写一次。而在日志中，如果我们数据要保证完全的 `all-or-nothing`，我们必须使用 `journal mode`，数据必须要写两次。如果我们写的数量很少的情况下，写时复制的粒度就是 4K，而日志只需要修改几个 `byte`，更合适。

A3: 可以的，就是 `Btrfs`（B-tree FS）。每次更新都可以从下往上修改指针。`root` 里有很多的扩展，整个这些元数据都是以树的方式组织在一起。任何一个地方有更新，都可以使用 `CoW` 的方式层层往上实现原子性的操作。要实现原子性的更新，必须要有小于 4K 的东西，比如指针。

B-tree FS（写时复制文件系统）

"写时复制"文件系统

- 写时复制也可用于整个文件系统
- `Btrfs`（B-tree FS）



在 B-tree FS 里，可以用很简单的方式实现普通文件系统很难实现的功能，那就是 `snapshot`。我们只需要保存根树的指针，并且我们不把旧的数据删掉，那么快照就在那了。创建 `snapshot` 的作用就是，我们可以很方便地 `rollback`。它和 `journal` 有很多类似的地方，总之就是不要覆盖。

如果我们把文件的新数据看成是一个日志，那么相当于把日志打散了，每个文件都有一个日志。当日志起作用的时候，就是我们更新 `inode` 的时候。

Soft Updates

`soft update` 和前面的思路就不太一样。

为什么前面六种情况中，有一种情况是没问题的呢？如果我们目录项没持久，也就是没有把文件挂载进目录里。在这个情况下，我们对文件系统没有任何影响，也不会占用位置。

回顾：创建文件时崩溃的六种情况

文件系统结构				是否影响使用	典型问题
inode 位图	inode 结构	目录项			
情况 1 持久	未持久	未持久	未持久	否	空间泄漏
情况 2 未持久	持久	未持久	未持久	否	无
情况 3 未持久	未持久	持久	持久	是	信息错乱
情况 4 未持久	持久	持久	持久	是	信息错乱
情况 5 持久	未持久	持久	持久	是	信息错乱
情况 6 持久	持久	持久	未持久	否	空间泄漏

这就给我们带来了启示：

Soft Updates

- 一些不一致情况是**良性的**
 - 某inode被标记为占用，却从文件系统中无法遍历到该inode
 - 如创建文件：
 - 标记inode为占用
 - 初始化inode
 - 将目录项写入目录中
 - 合理安排修改写入磁盘的**次序 (order)**，可**避免恶性不一致**情况的发生
- 相对其它方法的优势
 - 无需恢复便可挂载使用
 - 无需在磁盘上记录额外信息

所以 soft updates 不需要在磁盘上记录额外的信息，因为它保证了顺序，就算出问题也没事。文件系统里的操作是不是都可以保证 order 吗？这就是 soft updates 要回答的问题。

我们要跟踪数据更新的 dependency，其实就是 order。比如我们更新 inode 后，要更新 dentry 等。在保证 dependency 的情况下，写入到磁盘中去。

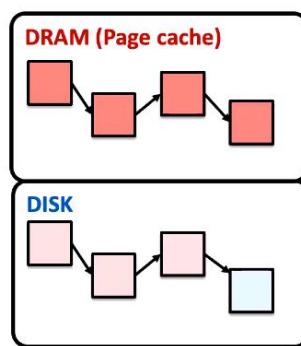
Soft Updates的总体思想

最新的元数据在内存中

- 在DRAM中更新，跟踪dependency
- ✓ DRAM 性能更好
- ✓ 无需同步的磁盘写

磁盘中的元数据总是一致的

- 在遵循dependency的前提下写入磁盘
- ✓ 一直能保证一致性
- ✓ **发生崩溃后，重启立即可用**



传统的 Soft Updates

我们之前提到一个 block 被两个文件使用是最大的隐患。

Soft Updates的三个次序规则

1. 不要指向一个未初始化的结构

- 如：目录项指向一个inode之前，该inode结构应该先被初始化

2. 一个结构被指针指向时，不要重用该结构

- 如：当一个inode指向了一个数据块时，这个数据块不应该被重新分配给其他结构

3. 不要修改最后一个指向有用结构的指针

- 如：Rename文件时，在写入新的目录项前，不应删除旧的目录项
这个 dependency 都是写在内存里的，不是日志。

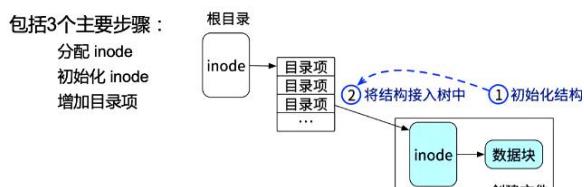
Soft Updates

• 对于每个文件系统请求，将其拆解成对多个结构的操作

- 记录对每个结构的修改内容（旧值、新值）
- 记录这个修改依赖于那些修改（应在哪些修改之后持久化）
- 如创建文件：
 1. 标记inode为占用（对bitmap的修改）
 2. 初始化inode（对inode的修改，依赖于1）
 3. 将目录项写入目录中（对目录文件的内容修改，依赖于1和2）

我们创建的文件是一个目录，所以要分配一个新的数据块这个操作。在满足这个 order 的时候，如果发生了 crash，还是会发生空间泄露，导致分配的 inode 没人用了。但是这个是可以通过磁盘的 check 来解决的。

创建文件



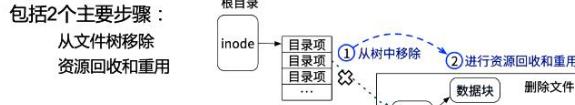
• 操作步骤

- 根据规则1，inode 初始化的持久化应早于增加目录项操作的持久化
- 根据规则2，inode 分配同样应该在增加目录项之前持久化
- 需同时创建“.” 和 “..” 两个目录项，需要分配并初始化一个新的数据块
- 根据规则1，数据块的初始化操作应该先于 inode 中索引的修改持久化
- 根据规则2，数据块分配信息（如 bitmap）的持久化同样应该在 inode 初始化之前

• 异常情况

- 依然可能会发生空间泄漏，即 inode 分配信息被持久化但却未被文件系统使用
- 对文件系统结构没有影响；可通过定期扫描找到未使用的 inode 节点并修复

删除文件



• 操作步骤

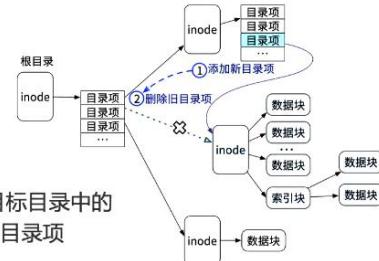
- 根据规则2，文件系统需要将目标文件先从整个文件系统树中去掉，再进行资源回收等操作前

• 异常情况

- 删除目录项为删除操作的原子更新点，系统崩溃只可能发生在操作之前或之后
- 若崩溃发生在此之前，则没有删除操作被执行，因此不会产生不一致的情况
- 若崩溃发生在此之后，并不会造成文件系统中其他文件和数据的不一致性
 - 此过程中造成的空间泄漏，也可以通过定期检查的方法进行修复

因为我们要排序，所以我们要想明白每一个操作的顺序。

文件重命名



• 操作步骤

- 根据规则3，在进行文件移动时，需要先保证目标目录中的目录项被写入完毕，之后才能删除源目录中的目录项

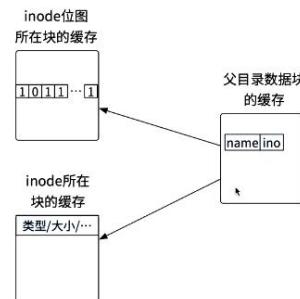
• 异常情况

- 目标目录中的目录项写入完毕后发生崩溃
- 重启后会发现两个目录中的目录项均指向该 inode 结构
- 并未造成被移动文件的数据丢失，也不影响文件系统中其他文件的一致性

依赖追踪

依赖追踪就是应用之前的三条规则知道“谁在谁之前”，形成了一个有向无环图。我们可以把一组操作打包成多个，尽可能降低 flush 的操作。

依赖追踪



• Soft Update原理

- 使用内存结构表示还未写回到存储设备的修改
- 并异步地将这些修改持久化到存储设备中
- 问题：如何保证这些修改的持久化顺序呢？

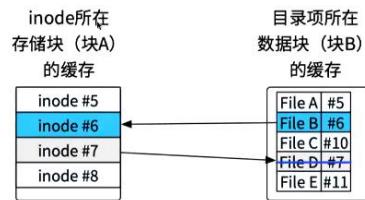
• 依赖追踪

- 根据3条规则，对修改之间需要遵守的顺序进行记录
 - 如果修改 A 需要在修改 B 之前写入到存储，则称 B 依赖于 A
- Soft update会将这些修改之间的依赖关系记录下来

有环就很麻烦了，会打破我们的 order。这是完全有可能的，核心是因为文件系统看到的是 inode、dentry 粒度，而磁盘看到的是磁盘块的粒度，所以存在这种依赖是完全有可能的。

第二个问题是写回迟滞，一个结构一直产生新的依赖，导致它一直没有办法写到磁盘里。

依赖追踪的两个问题



- **问题1：环形依赖**

- 一个块通常包含多个文件系统结构
- 环形依赖：块 A 需要在块 B 前写回，同时块 B 需要在块 A 前写回

- **问题2：写回迟滞**

- 当一个结构中的数据被频繁修改时，该结构很可能由于一直产生新的依赖导致长时间无法被写回到存储设备之中

要解决这两个问题，我们需要撤销和重做机制。这两个是 *soft updates* 里最复杂的机制。为了把环形打破，我们必须把块粒度细化成文件系统的结构粒度。我们需要记录下每个结构上的修改记录。一旦出现环形依赖，就要把部分操作撤销。

比如说我们删除一个文件，就得撤销，以为我们后面有别的操作依赖于你，就会形成环。撤销的方法就是把内存结构还原成执行前的状态。如果环没有打破，那就继续撤销，直到最后没有环。

撤销和重做

撤销和重做

- **解决环形依赖**

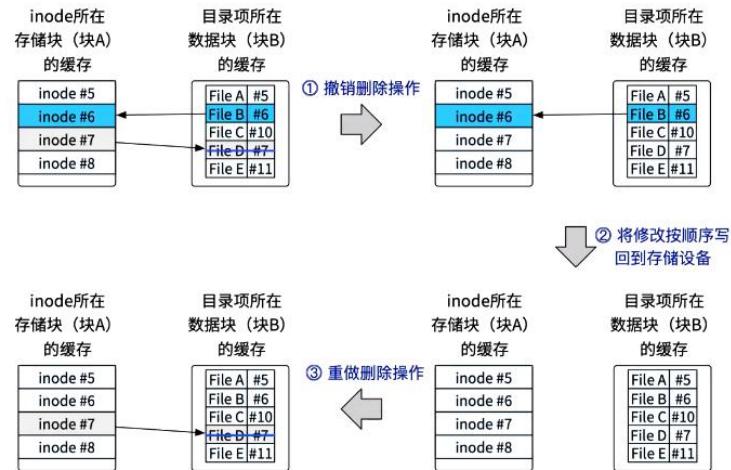
- 将依赖追踪从块粒度细化为结构粒度，使用撤销和重做打破循环依赖

- **记录每个结构上的修改记录**

- 当需要将某个结构写回到存储设备时，检测是否有环形依赖
- 当出现环形依赖时，其先将部分操作撤销
 - 即将内存中的结构还原到此操作执行前的状态
- 撤销之后环形依赖被打破，根据打破后的依赖将修改按照顺序持久化
- 持久化完毕之后，将此前被撤销的操作恢复，即重做。
 - 在重做完成后，将最新的内存中的结构按照新的依赖关系再次持久化

我们通过如下的例子来阐释。假设我们此前创建了一个文件 B 和删除了一个文件 D 形成了一个环形依赖。那么在撤销删除之后，块 A 和块 B 只有一条依赖关系了。我们就可以把块 A 先写、块 B 再写。写完之后，A 和 B 又只有一条依赖关系了。那就可以继续重做删除操作了。

撤销和重做



撤销和重做

- **不仅能够打破环形依赖，还能解决写回迟滞的问题**
 - 若某个结构被频繁修改，导致不断有新的依赖产生时，可将部分新的修改撤销，在快速完成持久化后将修改重做
 - 避免新依赖不断推迟该结构上修改的持久化

总的来说，崩溃一致性的核心就是元数据彼此之间是存在不变量关系的 (dependency)，比如所有 link 加起来等于 ref_cnt 的和。但是正因为这些数据存在磁盘上的很多地方，一旦发生断电/崩溃会导致不变量被破坏。解决这个问题就有两个思路，1.原子更新。日志的 commit point 在于日志的提交记录写入与否，正因为整个文件系统是 4K 的粒度，我们可以在这个保证前提之下可以去做这个事情。但是如果有一天磁盘可以保证 64K 原子写，那么我们的性能可以得到更大的提升。

对 CoW 来说，它是去中心化版的日志。commit point 在我们更新一个文件的 inode 的时候。如果树足够大，一旦更新叶子节点上的 4 个 byte，导致这一串间接索引都要更新。我们要保证一次更新的数据小于 4K。

而 soft updates 是根据文件系统的语义进行排序。这就需要对文件系统非常了解。

Review : 崩溃一致性保障方法

- 日志
 - 写时复制
 - Soft updates
- } 原子更新技术

2022/4/14

上节课主要讲了崩溃一致性以及解决的三种问题。一个经典的例子就是创建文件的时候发生崩溃会有六种情况。

Review: 创建文件时崩溃的6种情况

	文件系统结构			是否影响使用	典型问题
	inode 位图	inode 结构	目录项		
情况 1	持久	未持久	未持久	否	空间泄漏
情况 2	未持久	持久	未持久	否	无
情况 3	未持久	未持久	持久	是	信息错乱
情况 4	未持久	持久	持久	是	信息错乱
情况 5	持久	未持久	持久	是	信息错乱
情况 6	持久	持久	未持久	否	空间泄漏

其实日志的思路非常简单，就是把新做的操作变成日志的形式先记下来。一旦发生崩溃了，我们就从 to-do-list 里找到我们要做的事情 redo。日志的额外的好处就是，我们可以先不用真的去做操作，可以在内存里做写吸收。这样可以提高一部分的性能。

Review: 日志 (Journaling)

创建"/新文件"的修改包括：

1. 标记inode为占用
2. 初始化inode
3. 将目录项写入目录中

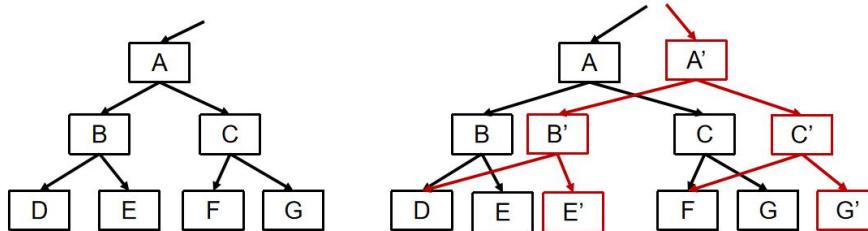
在内存中进行上述操作的同时，在磁盘上记录日志



写时复制和日志有很多相似的操作，它是把旧数据复制一份做修改，然后我们把这个修改一次性地对外修改出来。所以我们要做成指针一样的结构。在这个过程中，因为我们的旧数据没有被覆盖掉，就算发生了 crash，旧数据和新数据都在，实在不行我们可以还原到旧数据的情况。

Review: 写时复制 (Copy-on-Write)

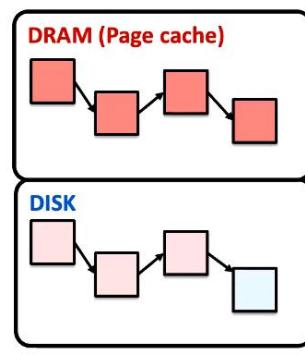
- 在修改多个数据时，不直接修改数据，而是将数据复制一份，在复制上进行修改，并通过递归的方法将修改变成原子操作
- 常用于树状结构



第三个思路就是 Soft Updates，它核心思路是如果磁盘上的数据在上一个时刻是一致的，我们只需要保证下一个写入 4K 后的状态依旧是一致的即可。我们利用的时候，就是磁盘崩溃的时候有一些情况是良性的。重点就变成了我们怎么对操作排队，这就需要我们记录一下写操作 dependency，满足写到磁盘上去的和后面的不会形成不一致的问题。如果可能导致不一致，那么我们就换一个顺序。

Review : Soft Updates

- 最新的元数据在内存中**
 - 在DRAM中更新，跟踪dependency
 - ✓ DRAM 性能更好
 - ✓ 无需同步的磁盘写
- 磁盘中的元数据总是一致的**
 - 在遵循dependency的前提下写入磁盘
 - ✓ 一直能保证一致性
 - ✓ **发生崩溃后，重启立即可用**



大家回想一下银行家算法，它是用来解决死锁问题的。进程之间形成了循环的等锁，银行家算法就是只要有一种办法能够完成执行，我们就认为它处于正确的状态。所以当进程 A,B,C 都向 OS 要资源的时候，到底先把资源给谁，就需要我们做状态的预演。看看假如给了某一个进程，执行后的状态是不是一致的状态。

当时 soft updates 就是针对 FFS (Fast File System) 做的。而追踪 dependency 的关系是和文件系统的操作和实现紧密相关的，所以在一个文件系统上能够运行的 soft updates 机制要应用到 ext4 和 NTFS 的难度是很大的，因为每个文件系统的操作我们需要重新考察依赖情况。所以使得 soft updates 在跨文件系统的场景下的实现是很难的。它的好处就是所有数据只需要写一遍，而不需要写两遍。

总结一下，日志和写时复制是靠原子更新，而 soft updates 是合理安排更新顺序。

Review : 崩溃一致性保障方法

- 日志
 - 写时复制
 - Soft updates
- } 原子更新技术；不覆盖唯一的一份数据
合理安排更新的顺序，尽可能一直保持一致

在磁盘上，开启日志后性能反而有所提升，为什么？

A1：它特别适合磁盘的原因是因为磁盘很喜欢顺序写。用日志文件系统可以很好地利用到磁盘的这个特点。

我们为什么不能实现这样一个顺序写的文件系统呢？比如我们所有文件都使用日志的方式写，且操作缓存到 cache 中。这样就可以同时利用了日志的顺序写和 cache 的特点。这就是我们要讲的日志文件系统（LFS）。

日志文件系统（LFS）

日志文件系统（Log-structured FS）

The Design and Implementation of a Log-Structured File System

Mendel Rosenblum and John K. Ousterhout
Electrical Engineering and Computer Sciences, Computer Science Division
University of California
Berkeley, CA 94720
mendel@sprite.berkeley.edu, ouster@sprite.berkeley.edu

Abstract

This paper presents a new technique for disk storage management called a *log-structured file system*. A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from disk efficiently. In order to maintain large areas on disk for fast writing, we split the log into *segments* and use a *segment cleaner* to compress the live information from heavily fragmented segments. We present a series of simulations that demonstrate the efficiency of a simple cleaning policy based on cost and benefit. We have implemented a prototype log-

magnitude more efficiently than current file systems. Log-structured file systems are based on the assumption that file access is mainly sequential and that increasing memory sizes will make the cache more and more effective at satisfying read requests[1]. As a result, disk traffic will become dominated by writes. A log-structured file system writes all new information to disk in a sequential structure called a log. This approach yields write performance quadratically by avoiding almost all seeks. The sequential nature of the log also permits much faster crash recovery: current Unix file systems typically must scan the entire disk to restore consistency after a crash, but a log-structured file system need only examine the most recent portion of the log.

1992 ACM Transactions on Computer Systems



John K. Ousterhout

Sprite 负责人
TCL/TK 作者
Magic VLSI CAD 作者
Co-scheduling 提出者
Raft 一致性协议
目前是Stanford大学教授



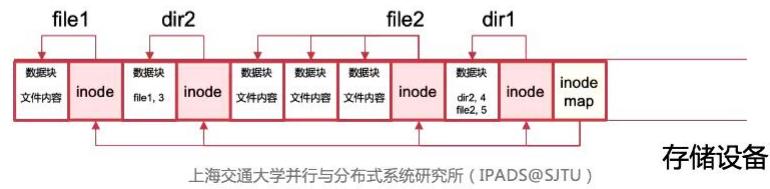
Mendel Rosenblum

VMware 联合创始人
目前是Stanford大学教授

这是一篇经典的论文。它的思路和我们之前说的一样的，我们利用磁盘顺序写快的特点：

日志文件系统 (Log-structured FS)

- 假设：文件被缓存在内存中，文件读请求可以被很好的处理
 - 但是文件写操作成为瓶颈
- 块存储设备的顺序写比随机写速度快很多
 - 磁盘寻道时间
- 将文件系统的修改以日志的方式顺序写入存储设备



这个文件系统和以前的文件系统长得不太一样。这种结构里，`inode` 和数据块是交织在一起的，因为谁改了，就会 `append` 到磁盘日志的后面。并不像我们之前设计的 `inode` 文件系统中，`inode` 的信息全部放在磁盘的头部等。为了顺序写，LFS 必须这样设计数据存储方式。

但是 LFS 也有点固定的结构放在固定的位置，比如 `superblock` 和 `checkpoint`。如果 `superblock` 也是 `append`，那么我们挂载的时候就很麻烦。其他大部分数据都是顺序 `append` 的。

我们记录程序运行的时候，要记录一些 `log`，这些 `log` 就是持久的。这里用 `log` 更合适，因为所有数据和元数据都在日志里，它是持久的。

Sprite LFS的数据结构

- 固定位置的结构
 - 超级块、**检查点 (checkpoint) 区域**
- 以日志 (Log) 形式保存的结构
 - `inode`、`间接块` (索引块)、`数据块`
 - `inode map` : 记录每个 `inode` 的当前位置
 - 段概要** : 记录段中有效块
 - 段使用表** : 段中有效字节数、段的最后修改时间
 - 目录修改日志**

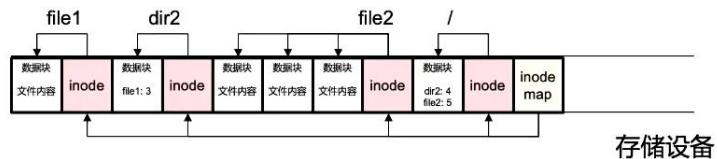
Log和Journal的区别？

- 中文都是“日志”
- Journal是暂时的
- Log是持久的

比如我们现在有 4 个 `inode`。

创建文件举例

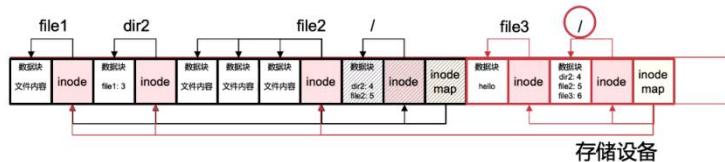
- 一个日志文件系统
 - 有4个inode，位置记录在inode map中
 - 对应4个文件分别为：/，/dir2，/file2，/dir2/file1



当我们创建一个新的文件的时候，我们要先找到根目录，并且要修改根目录对应的数据块，因为我们不能直接去改这个磁盘块（不能往前修改数据，我们只能往后）。我们应该先写这个文件，文件写完之后再去更新目录。我们最后一步是更新 inode map，因为我们很难通过 inode table 去找了，所以我们需要通过 inode map 去找，inode map 就会指向刚刚最新的两个 inode，分别是根目录的 inode 和 file3 的 inode。

创建文件举例

- echo hello > /file3
 - 创建文件
 - 修改文件数据



Q: 在这个过程中，如果我们要根据一个文件名找到一个文件，我们应该怎么做呢？

A: 根目录的 inode 在不断变化，我们先得找到根目录。我们可以通过最后的 inode map (就是记录 inode number 到 block 的映射关系) 找到根目录的 inode。我们从根目录的数据中找到 dir2 的 inode number 是 4，我们需要知道它具体的 block number，所以继续要从 inode map 中去找。

Q: 这种设计如果我们在写最后 inode map 之前发生了断电，这个时候根目录在哪？

A: 我们重启之后，会读到前一个 inode map 的位置，最后一个 inode map 往后的数据都没了。这是一个非常简单的实现。

如果我们每次都往后写，前面的数据是不是有浪费呢？随着文件系统一直使用，最后会写到末尾，并且我们发现前面大量的数据块都没用了，我们就需要回收它。一个直观的想法，我们通过 inode map 找到有用的数据和没用的数据，把两块数据分布聚集在一起。

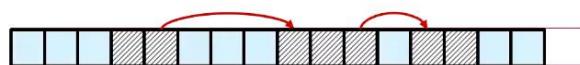
无效空间的回收

- 存储设备最开始是一个连续的空闲空间
- 随文件系统的使用，日志写入位置会接近存储设备末端
- 需重新利用前面的设备空间
 - 文件系统数据被修改后，此前的块被无效化
 - 如何组织前面无效空间，以重新利用它们？

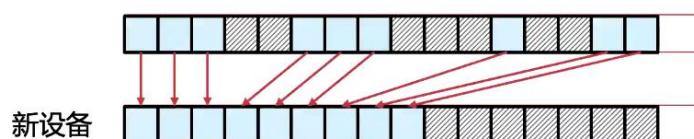
一种方法就是变成空闲的区间，每次写的时候可以顺着指针去写，但是这种方式随着磁盘空间依赖于碎，指针最后到处指，最后就会从顺序变成随机写。第二种方法就是停下来扫描一遍，我们把所有能用的归在磁盘头部，这样我们依然可以做顺序的操作。我们需要 stop-the-world，每次停下来 copy 几分钟是不能接受的。

空间回收管理方法：串联与拷贝

- 【方法-1：串联】：将所有空闲空间用链表串起来
 - 磁盘空间会越来越碎，影响到LFS的大块顺序写的性能

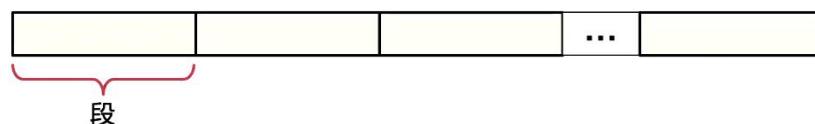


- 【方法-2：拷贝】：将所有的有效空间整理拷贝到新的存储设备
 - 数据需要拷贝



这个方法就是把两个方法做一个结合。我们可以不要对整个磁盘做一个 stop-the-world 做 GC 操作。我们就可以把它分成一段，在段内我们做 GC。若干个段做完 GC 之后，剩下的段可以串在一起。

- 方法-3：串联和拷贝两种方法的结合：段

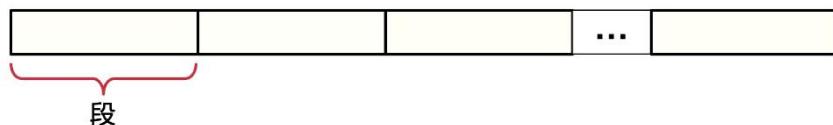


注意到我们段的大小不能太小，如果是 4K 那就和 block 没什么区别了。所以段不能太短也不能太长（整个系统停下来就太久了）。一旦某一个段里都是无效数据，我们就可以把 free 的段用链表继续串在一起。

段 (Segment)

空间回收管理方法：段 (Segment)

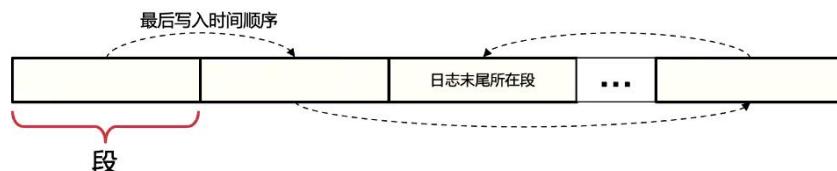
- 一个存储设备被拆分为定长的区域，称为“段”
 - 段的大小需要足以发挥出顺序写的优势，512KB、1MB等
- 每段内只能顺序写入
 - 只有当段内全都是无效数据之后，才能被重新使用
- 干净段用链表维护（对应串联方法）



我们要记录哪些段是空闲的，哪些段是可以用的。它要记录段内的有效字节数。段和段之间物理上是断开的，逻辑上使用链表连接在一起。段方法肯定比原来的性能差一些，磁头跳转还是有的。

段使用表

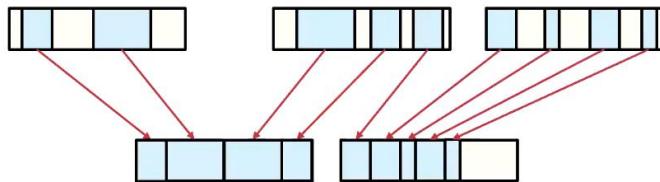
- 段使用表
 - 记录每个段中有效字节数
 - 归零时变为干净段
 - 记录了每个段最近写入时间
 - 将非干净段按时间顺序连在一起，形成逻辑上的连续空间



有了段以后，我们怎么样才能做到 GC 呢？我们要把段里的数据读入内存，识别出有效数据，然后把有效数据整理后写到干净段里。下图中蓝色部分是有效数据。

段清理

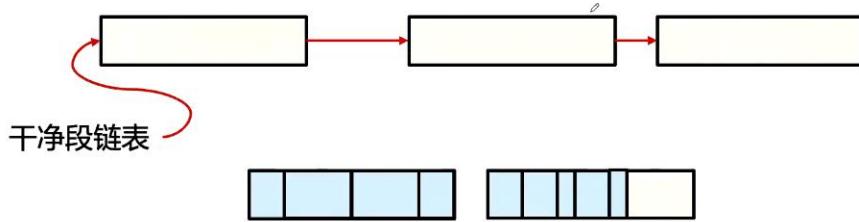
1. 将一些段读入内存中准备清理
2. 识别出有效数据
3. 将有效数据整理后写入到干净段中（对应拷贝方法）



Q: “整理”是什么意思呢？

A: 这里面如果有指针（block number）的话，都要相应修改。因为这里我们的数据发生了变化。

4. 标记被清理的段为干净



Q: 怎么识别出有效数据？如果我们只看一个段里有一个 block 和 inode，我们怎么知道这个 inode 是不是别人指过来的呢？

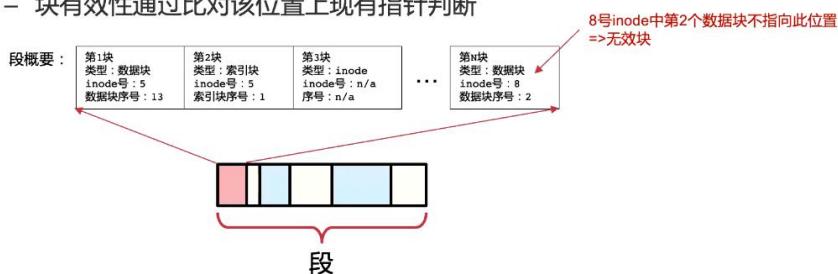
A: 我们只能找到 inode map 是不是指向你这里，并且我们还需要遍历一遍数据，但是这就非常慢了。

所以，我们需要在每个段的段头加了一个“段概要”，去描述这个段里哪些数据到底是有用的。它记录下每一块是做什么的。但是段概要是段里的某一块数据在第一次写的时候被更新的，所以它是有可能过时的。我们可以真的找到 inode 为 5 的 13 个数据块，看看它的 block number 是不是和第一块序列是一致的（不一致，说明数据已过时）。

如果不一致，那就说明我们的数据过时了。我们不需要扫描整个文件系统，只需要扫描 inode = 5 的第 13 个数据块就行了。段概要其实就是维护了一个反向映射，从而让我们不需要全盘扫描。

识别有效数据

- 每个段中保存有段概要
 - 每个块被哪个文件的哪个位置所使用
 - 如：数据块可使用inode号和第几个数据块来表示位置
 - 块有效性通过比对该位置上现有指针判断



就算我们知道了怎么清理，还是有很多策略的选择问题。每个策略在不同的场景 /workload 可能效果都不一样。

清理的策略有多种维度

- 清理策略
 - 什么时候执行清理？
 - 后台持续清理？晚上清理？磁盘要满的时候清理？
 - 一次清理多少段？清理哪些段？
 - 有效数据应该以什么顺序排序写入新的段？
 - 维持原顺序？相同目录放一起？相近修改时间放一起？
- 段使用表为策略提供辅助信息
 - 记录每个段中有效字节数
 - 记录每个段最近写入的时间

一种优化就是我们定期地做 checkpoint，用 checkpoint 方式可以减少挂载时候检查的时间。我们只需要检查 checkpoint 后写入的日志即可。

挂载和恢复

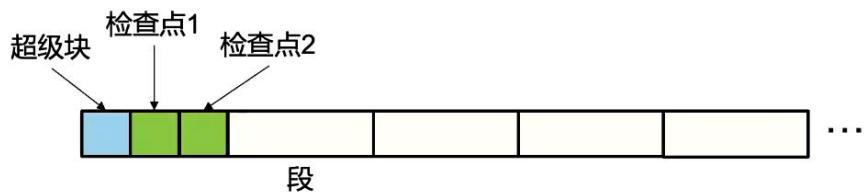
- 简单方法：扫描所有日志，重建出整个文件系统的内存结构
 - 缺点：大量无效数据也被扫描
- 优化：定期写入检查点（checkpoint）
 - 写入前的有效数据，可以通过检查点找到
 - 只需扫描检查点之后写入的日志
 - 减少挂载/恢复时间

它的位置是固定的，一定是在 superblock 之后。好处就是位置固定，不需要乱找。里面包含段使用表、时间戳、inode map 的位置等信息。

检查点

- 检查点内容

- inode map 的位置（可找到所有文件的内容）
- 段使用表
- 当前时间
- 最后写入的段的指针



Q: 为什么要维护两个检查点区域？

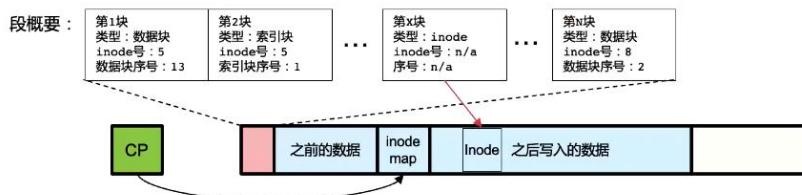
A: checkpoint 是为了防止崩溃，如果我们只有 1 个 checkpoint，那么写到一半的时候发生崩溃就会出问题。所以 checkpoint 有一个新的一个旧的，保证了不会覆盖唯一的数据。

前滚 (roll-forward)

检查点之后，我们还是要做恢复。也就是找到 checkpoint 后，通过段概要中的新的 inode 去恢复 inode。

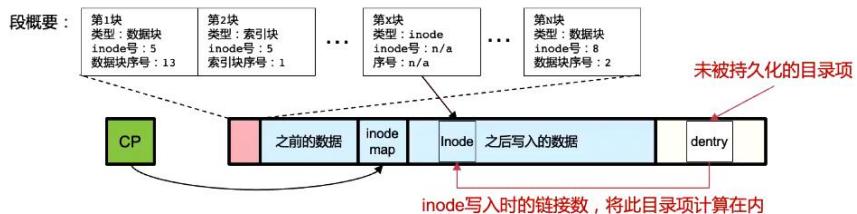
恢复：前滚 (roll-forward)

- 尽量恢复检查点后写入的数据
- 通过段概要里面的新inode，恢复新的inode
 - 其inode中的数据块会被自动恢复
- 未被inode"认领"的数据块会被删除



它一个大的缺点就是不能保证 inode 的 link 和 ref_cnt 是一致的。下面这个 case 中，inode 被持久化了，但是目录项没有被持久化。解决方案就是我们要加一个“目录修改日志”。

- 段概要无法保证inode的链接数一致性
 - 如：inode被持久化，但是指向其的目录项未被持久化



目录修改日志要在 inode 被写进去之前写入。

恢复：目录修改日志

- 目录修改日志
 - 记录了每个目录操作的信息
 - create、link、rename、unlink
 - 以及操作的具体信息
 - 目录项位置、内容、inode的链接数
- 目录修改日志的持久化在目录修改之前
 - 恢复时，根据目录修改日志保证inode的链接数是一致的

其他LFS的实现

- | | |
|--|---|
| <ul style="list-style-type: none"> • 光盘 <ul style="list-style-type: none"> – UDF • Flash/SSD <ul style="list-style-type: none"> – JFFS, JFFS2 – UBIFS – LogFS – YAFFS, YAFFS2 – F2FS • 非易失性内存 <ul style="list-style-type: none"> – NOVA | <ul style="list-style-type: none"> • RAID <ul style="list-style-type: none"> – WAFL (by NetApp) • NVM <ul style="list-style-type: none"> – NOVA • Cloud <ul style="list-style-type: none"> – ObjectiveFS via FUSE uses cloud object stores (e.g. Amazon S3, Google Cloud Storage and private cloud object store) |
|--|---|

包括今天的 Linux 也提供了 LFS 的实现，我们可以把一个磁盘直接格式化成 LFS。

关于LFS的讨论

- 之前的假设：大多数读请求可以通过内存缓存处理
 - 实际情况：缓存未命中时去磁盘上读，会非常慢
 - 因为文件会非常分散

通过局部性，确实可以把大量的读请求 cache 进来，一旦没有命中，我们就不得不去磁盘上读。

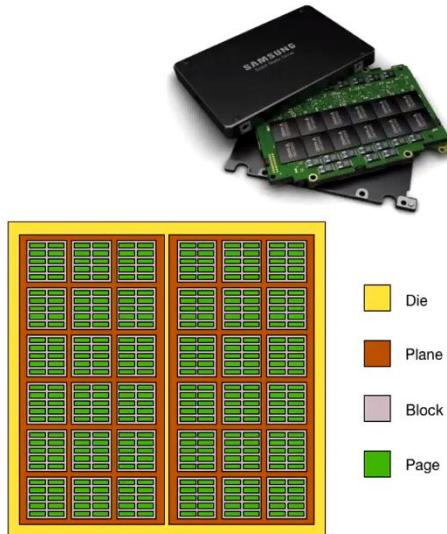
F2FS (Flash 友好的文件系统)

Flash 和磁盘非常不一样。闪存有很多层，它是靠电子器件堆起来的，绿色的就是一个 page。每个 Page 里有很多 cell（最小单位），每个 cell 有不同的级别。若干个 page 组成了 1 个 block（灰色区域），若干个 block 组成了 1 个 plane（红色区域），1~2 个 plane 组成了一个 die。一个 flash 上有好几个 die。

Flash (闪存盘)

闪存盘的组织

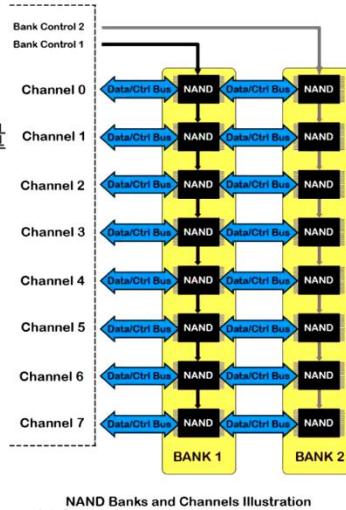
- (NAND) 闪存盘组织结构
 - A chip/package
 - => 1/2/4 dies
 - => 1/2 planes
 - => n blocks (块)
 - => n pages (页)
 - => n cells
 - => 1/2/3/4 levels



通道就是直接操作 SSD 的方法，多个通道可以同时写入和同时读。可以起到很好的并发性能。

闪存盘的组织

- **通道 (Channel)**
 - 控制器可以同时访问的闪存芯片数量
- **多通道 (Multi-channel)**
 - 低端盘有2或4个通道
 - 高端盘有8或10个通道



每次读可以读 8~16K，每次写，我们必须把整个 block (4~8MB) 全部擦掉，擦完只后再去写，而不是 *inplace* 地修改。每个块被擦除的次数是有限的，写多了就坏了。

闪存盘的性质

- **非对称的读写与擦除操作**
 - 页 (page) 是读写单元 (8-16KB)
 - 块 (block) 是擦除单元 (4-8MB)
- **Program/Erase cycles**
 - 写入前需要先擦除
 - 每个块被擦除的次数是有限的
- **随机访问性能**
 - 没有寻道时间
 - 随机访问的速度提升，但仍与顺序访问有一定差距

异质 cell 就是每个 cell 可能可以存不同的 bit 数。一个 cell 如果能存储更多的 bit，它的密度越高、性能越差、磨损也越快。

闪存盘的性质

- 磨损均衡

- 频繁写入同一个块会造成写穿问题
- 因此需要将写入操作尽可能均匀地分摊在整个设备

- 多通道

- 高并行性

- 异质Cell

- 存储1到4个比特：SLC、MLC、TLC、QLC

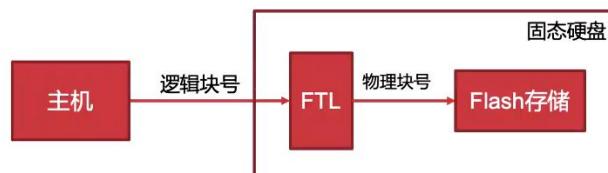
考虑到磨损均衡，我们就需要 FTL（Flash Translation Layer）。我们对 Flash 这一层加一层逻辑地址到物理地址的转换。其实和页表很像。逻辑地址是对外可用的，物理地址是内部用的，通过这种方式就可以做到磨损均衡。写同一个逻辑地址可以映射到不同的物理地址，从而做到磨损均衡。这通常是固态硬盘的固件去完成的。

FTL(Flash Translation Layer)

Flash Translation Layer (FTL)

- 逻辑地址到物理地址的转换

- 对外使用逻辑地址，内部使用物理地址（思考：和什么很像？）
- 可由软件实现，也可以由固件实现
- 可用于垃圾回收、数据迁移、磨损均衡（wear-levelling）等



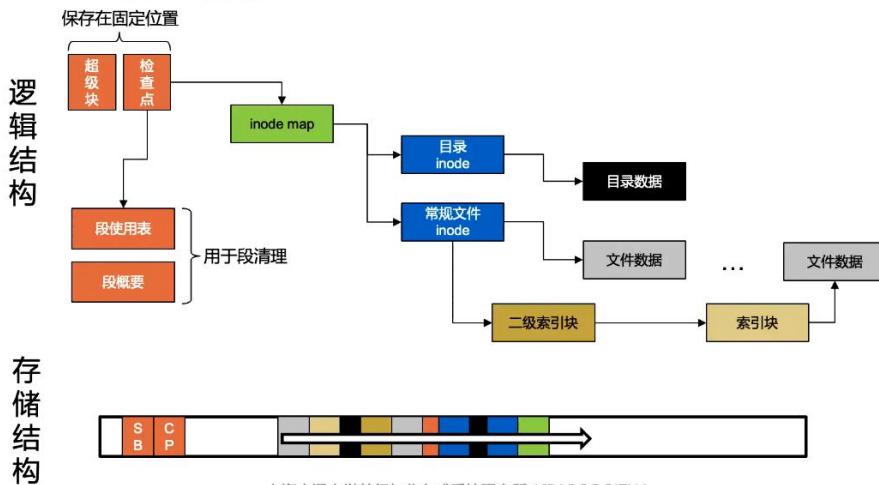
每次我们写一个 byte，我们不得不把 4K 擦掉再更新一下。所以两者其实是类似的。所以我们也希望顺序地 append 上去。

观察：LFS与Flash的相似性

- LFS
 - Segment清理后才能使用
 - 顺序写入
 - 需要清理（垃圾回收）
- Flash
 - Block擦除后才能使用
 - 需要考虑磨损均衡
 - 需要垃圾回收

一个自然的方法就是我们希望把 LFS 用在 Flash 上。于是我们就有了如下的设计，逻辑结构和我们之前的 inode、LFS 是一样的。在物理上是连续的，一直往后写，这样擦除操作就可以很少。

LFS的结构



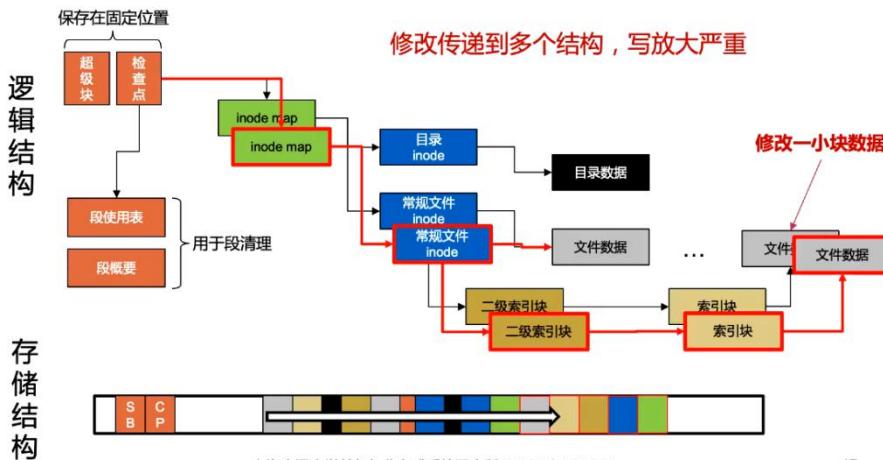
上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

44

1. 递归更新问题

这样的结构一旦我们修改了一部分数据，我们需要递归更新上去，最终我们修改了文件、数据、各级索引块。都做了一次 CoW。二级索引块记录的是索引块对应的 block number，如果索引块有一个别名，而二级索引块只记录了别名，而别名到真实 block number 有一层映射，那么我们就不需要改二级索引块了。

LFS的问题-1：递归更新问题



上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

47

NAT

我们只需要修改一些索引块的别名到 block number 的映射，就可以避免往上修改了。所以我们就有了 NAT (Node Address Table)。Node 就是索引块，我们给它一个地址就是 Node Address。

间接 Node 保存的是直接 node 的 node 号。

F2FS的改进-1：NAT

- 引入一层 indirection : NAT (node地址转换表)

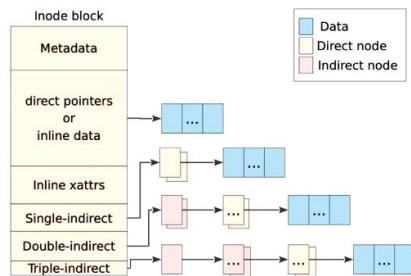
想起了什么原则？

- NAT : Node Address Table

- 维护node号到逻辑块号的映射
- Node号需转换成逻辑块号才能使用

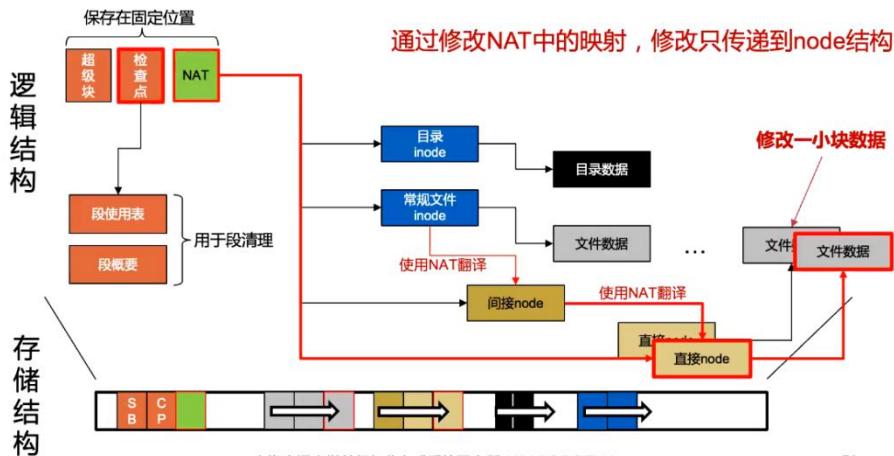
- F2FS中的文件结构

- 直接node : 保存数据块的逻辑块号
- 间接node : 保存node号
(相当于索引块)
- 数据块 : 保存数据



有了 NAT 之后，我们只需要修改文件的数据、直接 node 以及 NAT 表中维护的映射内容。好处是减少了磁盘的改动，坏处就是每次 inode 想访问间接 node 的时候得先去看一下 NAT 表。

F2FS的改进-1：NAT

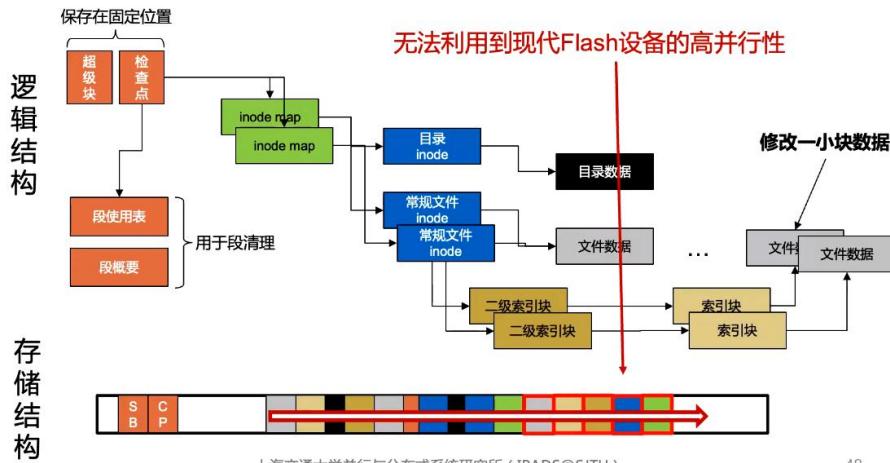


上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

51

第二个问题就是单日志顺序写入问题。顺序写适合磁盘，但是并没有利用到现代 Flash 设备的高并行性。（明明可以多个人写，我们却只让一个人写）

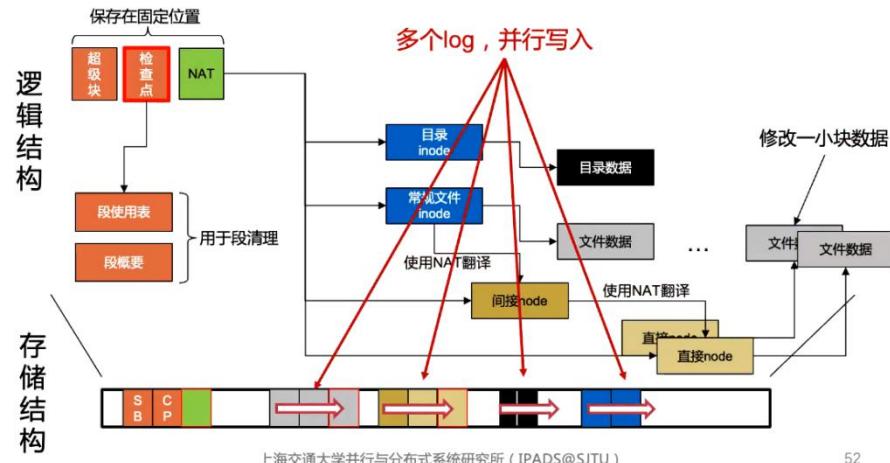
LFS的问题-2：单一log顺序写入



48

最后，它就是允许不同类型多个日志同时写。

F2FS的改进-2：多log并行写入



52

它还做了一些 fancy 的优化，划分成冷、热、温三种。

多Log写入

- 按热度将结构分类
 - 每个类型和热度对应一个log
 - 默认打开6个log
 - 用户可进一步配置
- 根据硬件信息可以进一步调整
 - 调整zone、section大小
 - 与硬件GC单元对齐等

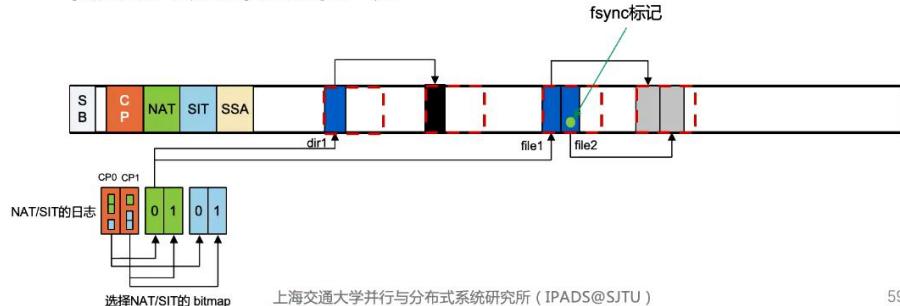
类型	热度	对象
Node	热	目录的直接node块 (包括inode块)
	温	常规文件的间接node块
	冷	间接node块
	热	存放目录项的数据块
Data	温	常规文件的数据块
	冷	被清理过程移动的数据块
	冷	用户指定的冷数据块 多媒体文件的数据块

一旦崩溃了，我们就回滚到检查点，再把日志 redo 一下。一旦用户发了 fsync，传统的文件系统是要做 checkpoint，这里做的优化就是不真的做 checkpoint，而是把相关的表加上一个 fsync 标记。恢复的时候看到 fsync 的 block，也会恢复。

检查点

回想一下：为何要需要检查点？

- 检查点、NAT、SIT（段信息表）各有两份
- 检查点中保存有NAT和SIT的日志，避免NAT和SIT的频繁更新
- 恢复时回滚到最近的检查点

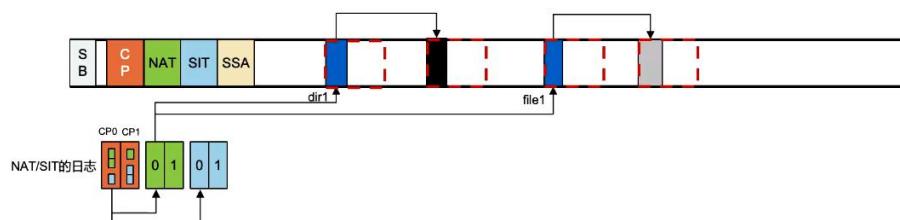


59

我们来看如下的一个例子，5秒到了一个我们创建 checkpoint 把 `dir1` 和 `file1` 记下来。

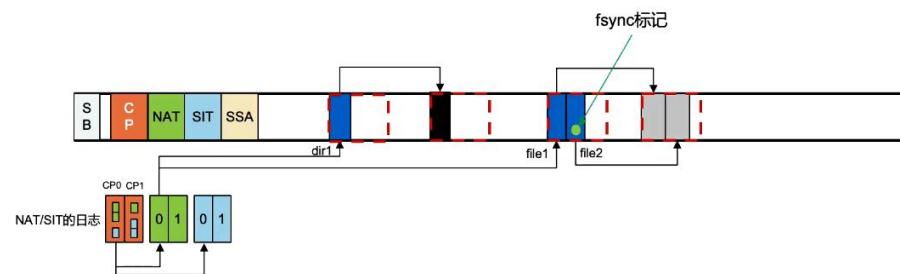
崩溃与恢复

1. 创建`dir1`和`file1`
2. 创建检查点0



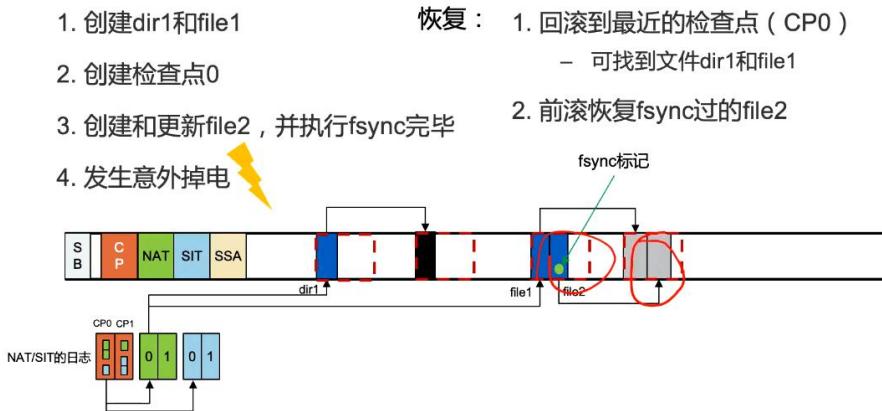
绿色的区域代表这个文件是被 `fsync` 过的。

3. 创建和更新`file2`，并执行`fsync`完毕



此时发生了掉电。我们先回滚到 `checkpoint0`，此时有 `dir1` 和 `file1`。然后我们 `roll-forward`，发现 `file2` 有 `fsync` 标记，所以继续恢复 `file2` 和对应的数据。

崩溃与恢复：前滚 (roll-forward)



前滚 : fsync()的处理

- 原有LFS
 - 创建新的检查点
- F2FS
 - 无需创建新的检查点
 - 持久化文件数据块和直接node，并在直接node上附带fsync标记
 - 前滚：恢复检查点之后fsync过的数据

前滚 : 恢复检查点之后fsync过的数据

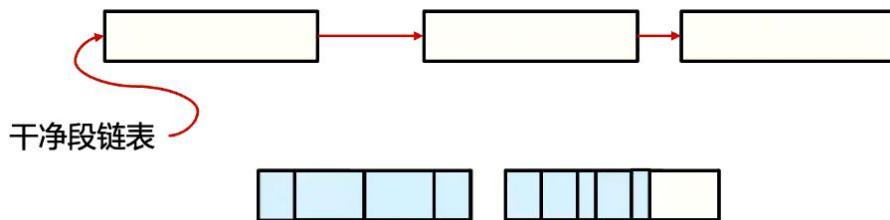
- 前滚恢复
 - 1. 查找带有fsync标记的直接node
 - 2. 对于每个直接node，对比其中的数据块指针，识别新旧数据块
 - 3. 更新SIT（段信息表），标记旧的数据块为无效
 - 4. 根据直接node中新数据块的记录，更新NAT和SIT
 - 5. 创建新的检查点

2022/4/19

有了这个过程之后，我们可以让磁盘大量时间都在顺序写，磁盘空闲的时候再进行段清理。我们通过指针把空闲段连在一起，就是一个 `freelist`。如果我们有多个磁盘，我们是不是可以让写操作全部写到一个磁盘上，写满了之后换下一个磁盘写，然后第一个磁盘去做清理。这样就可以形成一个和 RAID 不一样的模式。

Review : 段清理

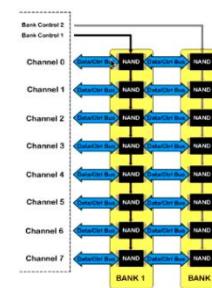
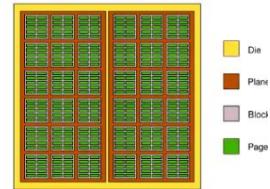
1. 将一些段读入内存中准备清理
2. 识别出有效数据
3. 将有效数据整理后写入到干净段中 (对应拷贝方法)
4. 标记被清理的段为干净



接下来，我们提到了闪存盘，它的特点和磁盘不一样。

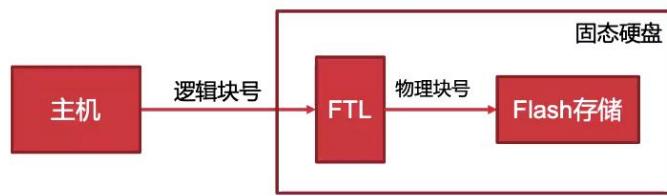
闪存盘的性能特性

- **非对称读写**
 - Block需要擦除后才能使用，需要考虑垃圾回收
 - 页 (page) 是读写单元 (8-16KB)，块 (block) 是擦除单元 (4-8MB)
- **磨损均衡**
 - 频繁写入同一个块会造成**写穿**问题
 - 因此需要将写入操作尽可能均匀地分摊在整个设备
- **随机访问性能**
 - 随机访问的速度提升，但仍与顺序访问有一定差距
- **多通道**
 - 高并行性



Flash Translation Layer (FTL)

- 逻辑地址到物理地址的转换
 - 对外使用逻辑地址
 - 内部使用物理地址
 - 可软件实现，也可以固件实现
 - 用于垃圾回收、数据迁移、磨损均衡（wear-leveelling）等

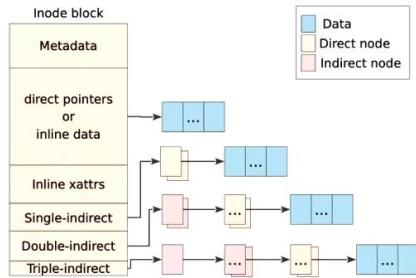


F2FS 在很多地方和 Flash 的访问模式非常像，但是直接用过来会导致写放大的问题。

F2FS的改进-1：NAT

- 引入一层 indirection : NAT (node地址转换表)
 - NAT : Node Address Table
 - 维护node号到逻辑块号的映射
 - Node号需转换成逻辑块号才能使用
- F2FS中的文件结构
 - 直接node : 保存数据块的逻辑块号
 - 间接node : 保存node号
(相当于索引块)
 - 数据块 : 保存数据

想起了什么原则？

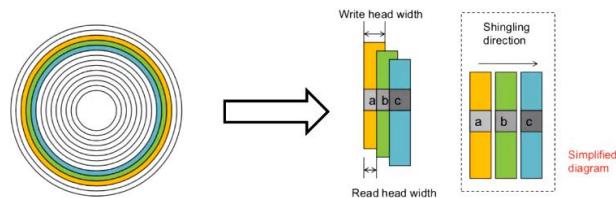


瓦式磁盘

前面我们讲了普通磁盘和 SSD，现在我们来看瓦式磁盘。它是一种新型磁盘。它发现读写需要的磁头的宽度是不一样的，读的时候可以读很细，而写的时候比较宽。所以写的时候我们覆盖掉一点之前的区域，从而提升磁盘的利用率。

瓦式磁盘

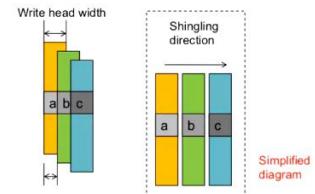
- 传统磁盘密度难以提升
 - 写磁头的宽度难以减小
- 瓦式磁盘将磁道重叠，提升存储密度
 - 减小读磁头的宽度



这样做顺序写是没问题的，随机写就会有问题。我们分成很多块，在一块之内只能顺序写。

瓦式磁盘的问题：随机写

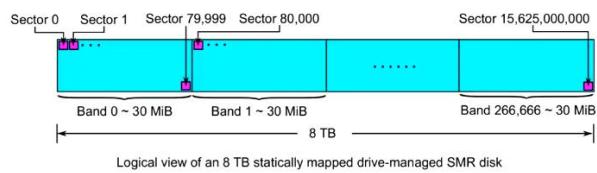
- 随机写会覆盖后面磁道的数据
 - 只能顺序写入
- 避免整个磁盘只能顺序写入
 - 磁盘划分成多个Band，Band间增大距离
 - 每个Band内必须顺序写入



这样想想就很慢，为了改一个 byte，要先读 30M，再写 30M。

方法一：多次拷贝

- 修改Band X中的4KB数据
 1. 找到空闲Band Y
 2. 从Band X的数据拷贝到Band Y，拷贝时将4KB修改写入
 3. 将Band Y中的数据拷贝回Band X
- 4KB随机写 → 120MB访问

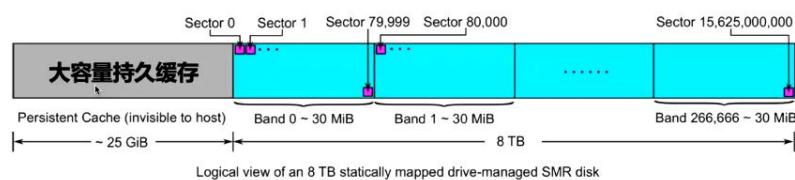


第二种方法就是加一个缓存。我们每次写的时候先写到传统磁盘的 cache 区域。读的时

候先从 cache 中读。一旦磁盘空闲的时候，再通过多次拷贝的方法把缓存中的数据清理到瓦式磁盘的位置上。

方法二：缓存+动态映射

- 大容量持久缓存
 - 在磁盘头部预留的区域，磁道不重叠，可随机写入
 - 给固件（STL）单独使用，外部不可见
- 动态映射：Shingle Translation Layer (STL)
 - 从外部（逻辑）地址到内部（物理）地址的映射



如下的几种瓦式磁盘有一些区别，第一种，OS 不需要知道磁盘内部逻辑。无需修改软件，可以直接替换传统磁盘，这是消费者最喜欢的方法。

后面两种就叫做 Host-aware 和 Host-managed，也就是我们需要使用驱动来管理磁盘。

瓦式磁盘种类

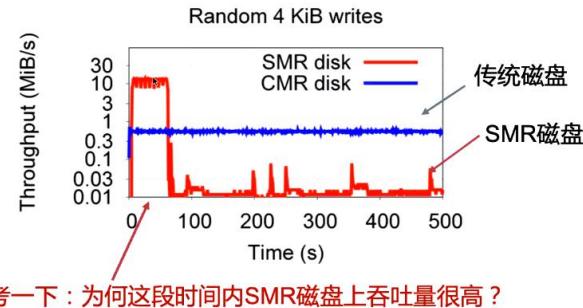
SMR磁盘种类	接口	随机写处理方法
Drive-managed SMR (DM-SMR)	普通块设备接口	固件进行缓存和清理
Host-aware SMR (HA-SMR)	特殊指令接口	固件进行缓存和清理
Host-managed SMR (HM-SMR)	特殊指令接口	必须顺序写，随机写请求被拒绝

无需修改软件，可直接替换传统磁盘！

下图的实验中，一开始瓦式磁盘比传统磁盘的性能高一个数量级，然后性能慢了 1000 倍。

DM-SMR上使用Ext4

- 当随机写入时，Ext4吞吐量非常低！



A: 一开始在写大容量缓存，只需要记录下到真实地址的映射即可。相当于把随机写变成了顺序写。写完之后就变成了 Band 区域，我们要以 30M 的粒度去写，性能就变差了。

如何改进Ext4来适应瓦式磁盘呢？

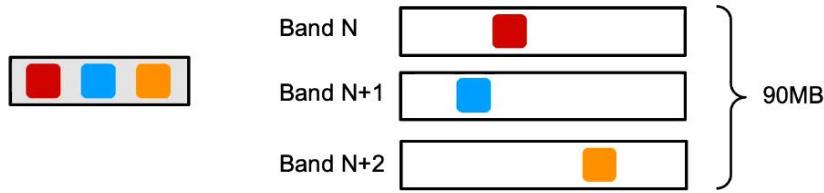
- 以DM-SMR磁盘为目标**
 - HM-SMR和HA-SMR需要文件系统的设计新的文件系统
 - 在成熟的Ext4上优化：消除元数据写回造成的随机写入
- 修改量小**
 - Ext4 + journaling 代码 ~50,000 行
 - 仅仅修改 ~40 行代码，新文件中添加 ~600 LOC 代码
- 效果显著**
 - 元数据修改较少时(<1%)，在SMR磁盘上有1.7-5.4倍性能提升
 - 有大量元数据修改时，在SMR和普通磁盘上，有2-13倍性能提升
 - 特定场景下达到40倍性能提升

两种思路，要么就是为我们的瓦式磁盘设计一个新的文件系统，因为我们硬件变化了，设计一个新的文件系统是比较自然的思路，比如 F2FS 就是为了 Flash 硬件而设计的。

另一个思路就是在成熟的 ext4 上进行修改，便于软件工程的迭代以及用户社群的维护。所以这个思路就是复用 ext4 的文件系统，这是 OS 里非常常见的思路。

我们来看一下这么大的性能提升是怎么做到的。

观察：持久缓存对吞吐量的影响



若持久化缓存中的写比较分散，清理时需要清理大量Band
假设清理1个Band需要1秒，吞吐量为1个随机写/秒



若三个随机写在一个Band中，则只需清理一个Band；吞吐量为3个随机写/秒
随机写的跨度→脏band数量→清理时的工作量→吞吐量

比如当我们的写入操作很分散的情况下，就有可能落到3个Band中，如果我们清理1个Band就写1个，那么我们速度就是1个随机写/秒，如果我们是写4K的话，那么我们1秒有效写入量只有4K。

但如果我们三个随机写在一个Band中，那么我们只需要清理1个Band，这样吞吐量就是之前的三倍。我们需要做的事情就是把前面的三次随机写放在一个Band中。

Q: 数据本来就是分散的，我们怎么去控制呢？

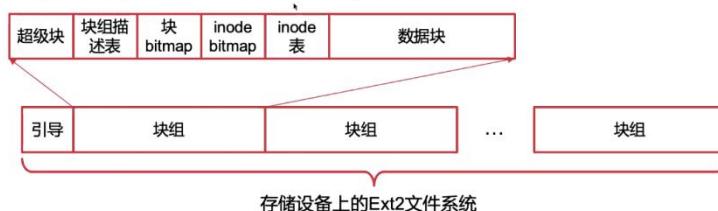
A: 这时候我们就要看文件系统的布局了。在ext2文件系统会把整个文件系统分成多个块组。每个块组元数据都在头部，然后后面存放数据。整个磁盘像被分割成多个小磁盘一样，这样元数据和数据的偏移量不至于太大，可以增加部分的局部性。

回顾：Ext2文件系统的存储布局

将磁盘分为多个块组，每个块组中都有超级块，互为备份

超级块（Super Block）记录了整个文件系统的元数据

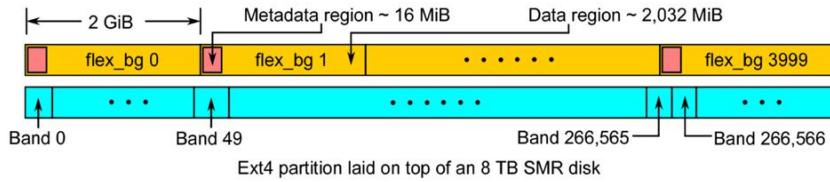
块组描述表记录了块组中各个区域的位置和大小



ext4也是使用块组去区分的，导致我们的元数据非常分散。每次数据修改要产生多次元数据的修改。分散的元数据的随机写会降低我们的吞吐量。

然而：Ext4的元数据非常分散

- 类似Ext2，Ext4同样使用块组（flex_bg）将文件系统分成多个区域
 - 每个块组前16MB用来保存元数据，其余保存数据
- 每次数据修改产生多处元数据的修改
- 8TB分区上有4,000个块组，元数据分散在4,000个Band！
- 分散的元数据随机写 → 脏band数量↑ → 清理工作的负担↑ → 吞吐量↓



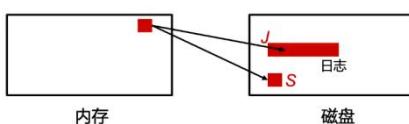
ext4 使用 JBD 来记录元数据日志，所有的元数据要先写到日志里，再写到对应的区域。在 Ext4 中有 128M 作为日志区域，因为这个区域是一个临时区域，比较快地就会写到该去的区域了。

由于日志本身存的很分散，就会导致大量的随机写。

回顾：Ext4上的元数据写回

- Ext4使用JBD2记录元数据日志

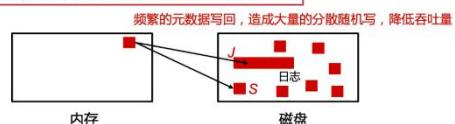
- 128MB的日志区域
- 1. JBD2首先将元数据写入日志区域J，标记元数据为脏
- 2. 脏元数据在日志提交后被写回到其应有位置S



回顾：Ext4上的元数据写回

- Ext4使用JBD2记录元数据日志

- 128MB的日志区域
- 1. JBD2首先将元数据写入日志区域J，标记元数据为脏
- 2. 脏元数据在日志提交后被写回到其应有位置S

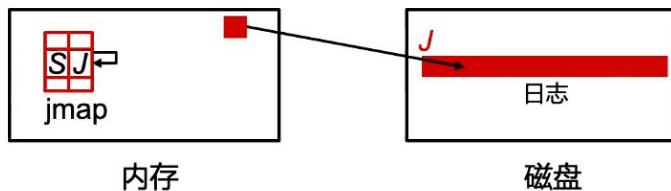


到此为止，答案已经呼之欲出了。因为日志频繁写回会造成大量的分散的随机写，所以我们可以让日志容量变大。我们让日志写回到元数据应在的区域 S 的间隔变大，但这样元数据很可能就保存在日志中的 J 处，而不是真正应该在的 S 处。

当我们去访问元数据的时候，我们需要知道是去 S 访问还是去 J 访问，所以我们就在内存中加一个表。如果这个表中记录了 S 在 J，那么我们就去访问 J；反之，如果没有记录 S，那就说明是在原来的地方，我们直接访问 S 即可。

回到问题：Ext4上的元数据分散

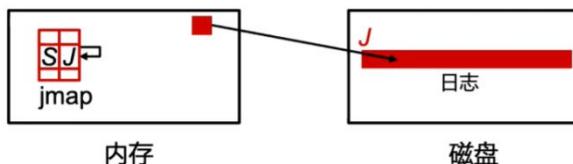
- 修改磁盘布局需要大规模修改Ext4
 - 人力成本、新增Bug、破坏原有功能……
- 怎么办？
- 引入Indirection：以LFS形式增加一个元数据缓存



所以这个优化相当于把原先临时的 J 的生命周期从几秒钟延长到了几分钟甚至几个小时，只要我们映射好 jmap 映射好，哪怕它一直在日志里都没问题。

解决方法：以LFS形式增加一个元数据缓存

- 以LFS形式维护10GB日志空间作为元数据缓存
 1. JBD2首先将元数据写入日志区域J，将元数据标记为clean（无需写回）
 2. JBD2在内存中的jmap中将S映射到J
- Indirection: 元数据访问需要通过 jmap 进行一次地址转换



Q1: jmap 应该保存在磁盘里还是内存里？

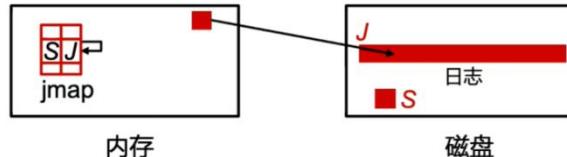
A1: 如果 jmap 放在磁盘中，这就意味着每次我们写回的时候就要更新一个随机的磁盘上的地方，性能就会下降。

Q2: 这样的结构设计，如果 crash 了应该怎么办？

A2: 其实很简单，我们可以把 journal 项 J 本身要写到哪个位置 S 这个信息放在 Journal 里面。这样的话，当每次挂载 FS 的时候，我们就可以去读取日志，从中恢复出内存中的 jmap。

日志满了怎么办？

- 日志空间清理
 - 无效的元数据（被新修改覆盖过的元数据）可以直接被回收
 - 对于冷的元数据，可将其写回到Ext4中其原本的位置S
 - 热的元数据继续保留在日志中
 - 挂载FS时，读取日志，恢复出jmap



Q: 怎么才能记录什么数据是“冷”的，什么数据是“热”的呢？选择 1：把冷热信息记录在磁盘上；选择 2：把冷热信息记录在内存里。

A: 显然选择 1, 性能会更高, 但是重启之后冷热信息可能就丢了。但注意到 crash 的频率也不会特别高, 比如一个月一次, 那我们甚至可以在重新挂载 FS 的时候清空日志里的数据, 全部写回, 因为一个月做一次花销也不大。所以, 这种统计类的信息, 还是适合放在 jmap 里。

比如，每访问一次，`jmap` 计数加 1；每个一段时间，清理日志空间的时候，把 `jmap` 里的统计信息清除。这样我们就可以区分出哪些是冷的数据，哪些是热的数据。

所以，具体到这个优化对 ext4 源代码的修改，其实也非常简单，把原来的 submit_bh 改成了 jbd2_submit_bh，实现在 jmap.c 中。这个函数拦截了原先 Ext4 对 block 的读写请求，每次做读写操作之前，我们先去查找 jmap，如果不在 jmap 中，我们直接调用原先的 submit_bh，如果在 jmap 中，我们就从日志中读取 block。

实现：对Ext4的修改

修改的~40行代码：将请求转给ibd2

```
+ jbd2_submit_bh(journal, READ | REQ_META | REQ_PRIO, bh, __func__);
```

新增文件:jmap.c(维护jmap)

```
+void jbd2_submit_bh(journal_t *journal, int rw, struct buffer_head *bh,  
+                      const char *func)  
+{  
+    sector_t fsblk = bh->b_blocknr;  
+    ...  
+    je = jbd2_jmap_lookup(journal, fsblk, func);  
+    if (!je) {  
+        submit_bh(rw, bh);  
+        return;  
+    }  
+    ...  
+    logblk = je->mapping.logblk;  
+    read_block_from_log(journal, rw, bh, logblk);  
+}
```

拦截Ext4对block的请求

查找jmap

} 不在jmap中，直接访问磁盘对应block

} 根据jmap中记录的地址，在日志中读取block

同样的，我们还要实现 cleaner 函数，把日志中的数据写回 Ext4 中应该在的位置。

实现：对Ext4的修改

新增文件：cleaner.c（维护10GB日志空间）

```
+static void do_clean_batch(struct work_struct *work) 将10GB日志空间的元数据写回到其在Ext4中的原本位置
+{
+    struct cleaner_ctx *ctx;
+    handle_t *handle = NULL;
+    int nr_live, err;
+
+    ...
+    nr_live = find_live_blocks(ctx);
+    if (nr_live == 0)
+        goto done;
+
+    ...
+    read_live_blocks(ctx, nr_live); } 扫描日志中有效块，并加入ctx中临时保存
+
+    handle = jbd2_journal_start(ctx->journal, nr_live); ← 新的JBD2原子更新
+
+    ...
+    attach_live_blocks(ctx, handle, nr_live); } 将ctx中的有效块加入到JBD2原子更新中
+
+    err = jbd2_journal_stop(handle); ← 结束JBD2原子更新
+
+done:
+    ...
+}
```

文件系统的稳定性、兼容性、容错机制，都是可以继承 Ext4 的，但是在瓦式磁盘上又可以提高 40 倍的性能。

非易失性内存（Non-volatile Memory, NVM）

接下来，我们讲非易失性内存，它是非常非常重要的，它有一个非常大的变化。传统的文件接口现在变成了内存的接口。它到底用文件接口还是内存接口就带来了新的思路，以前我们不能用访问内存的思路去访问一个磁盘，但是现在既可以用内存接口，又可以用文件接口。

非易失性内存的思路非常简单，核心问题就是为什么内存断电之后就没了？如果断电之后，数据还在内存里，那么就很好。

非易失性内存

如果内存里面的数据重启后还在，岂不是很棒？

思考一下：会有哪些好处？

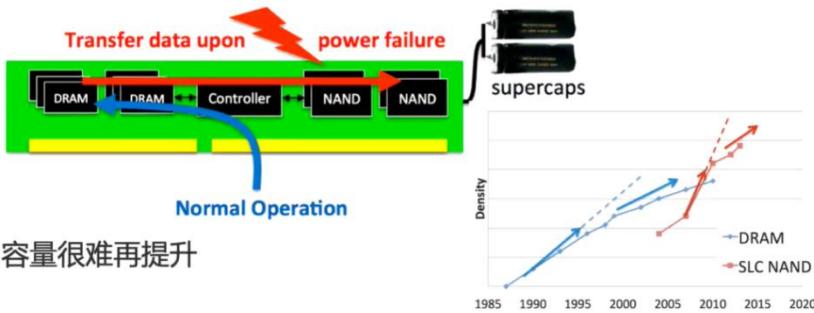
事实上很早就有人做过这样的尝试，它的思路是在内存上放一个大电容，平时不用的时候插在电路上，一旦断电了，就可以认为需要启用备用电源。大电容只要插的时候足够长，就可以把电容充满，断电以后，电容就把内存里的所有数据写到 flash 里。

如下图所示，内存上有一个 controller，平时走的是蓝色路径，写到内存中。一旦断电了，就走红色路径，把内存中的数据写到 NAND 这个 flash 里去。电容的量正好足够把 DRAM 里的所有数据写到 NAND 里去。

非易失性双列直插式内存模块(Non-volatile Dual In-line Memory Module, NVDIMM)

NVDIMM

- 在内存条上加上Flash和超级电容
 - 平时数据在DRAM中；断电后转移到Flash中持久保存



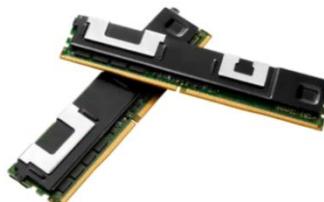
但是这种设计有一个问题就是它依然是受到 DRAM 的工艺限制，所以容量很难提升。

Intel Optane DC Persistent Memory

Intel 就提出了一种新的方式。这个使用内存接口，但是可以做到单条 512G，但是它依然有磨损均衡的问题，性能比 DRAM 慢十倍。

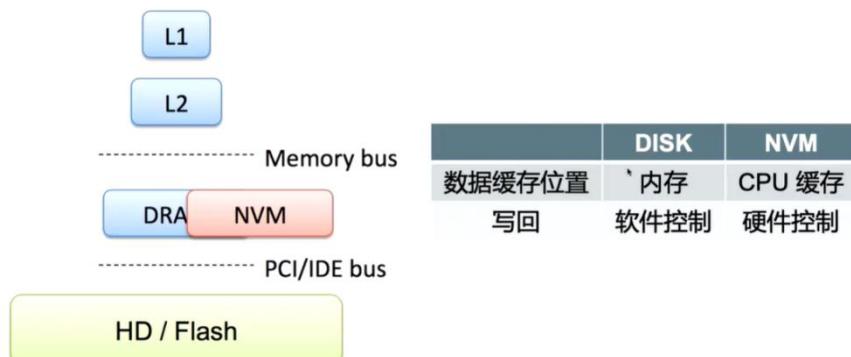
Intel Optane DC Persistent Memory

- ✓ 内存接口
- ✓ 字节寻址
- ✓ 持久保存数据
- ✓ 高密度 (512GB/DIMM)
- ✓ 需要磨损均衡，但耐磨度比NAND好10倍
- ✓ 比DRAM慢十倍以内，比NAND快1000倍



这会带来一些新问题。在原来的情况下，磁盘的话，数据会缓存在内存里，不会落盘，我们需要通过软件把数据写回到磁盘里。在非易失性内存的情况下，问题就上移了，也就是 CPU 的 cache 如果没有写回到 memory，信息就会丢失，因为虽然内存已经是 NVM 了，但是 CPU 的 L1, L2 cache 断电之后还是会丢失数据的。

非易失性内存带来的新问题



一种解决方案就是给 CPU 的 L1, L2 cache 加电容，一旦发送断电电容就把数据挪到 NVM 里去，现在也有这样的主板。这种方法对主板要求比较高，并不是最终的解决方案。

比如我们写两个数据 A 和 B，但是写到 DRAM 中的顺序可能就变成了 B, A。

内存写入顺序

- Writeback模式的CPU缓存
 - 虽然能提升性能，但会打乱数据写入内存的顺序

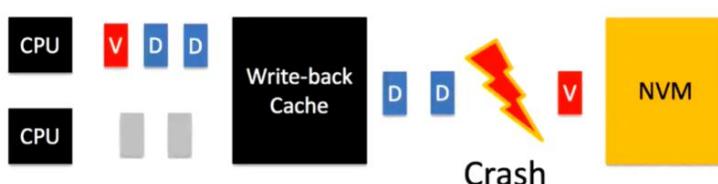


这会导致什么问题呢？我们来看下面的例子，我们希望 valid = 1 是最后写进去的，但是如果我们将 Write-back Cache 先把 valid = 1 写入 NVM 中，然后发生了 crash。那么我们就得到了 valid = 1，但是 data 还是旧的数据，打破了原子性。

非易失性内存写入顺序

- 考虑持久性和一致性，写入顺序很重要

```
STORE data[0] = 0xF00D  
STORE data[1] = 0xBEEF  
STORE valid = 1
```





错误的写入顺序在恢复时将垃圾数据视为有效数据！

Q: 有什么解决方案呢？

Option 1: 关闭 CPU cache，那性能就会下降很多。

Option 2: 使用 write-through，也就是读的时候可以 cache，但是写的时候就写穿到 NVM 上。但是这种情况，如果写很多，性能也会变差。

Option 3: 每次写入之后，flush 整个 cache，可以把 cache 的数据 flush 到 memory 里去。

我们可以在 STORE data[0] 和 STORE data[1] 之后调用 CLFLUSH，所以 valid 一定会在 data[0] 和 data[1] 都写到内存中以后再写到内存里。但是 CLFLUSH 必须要顺序执行，在 CLFLUSH 执行的时候，没有人可以执行 STORE 指令。一旦 CLFLUSH 之后，cache 里的数据就没了。

使用CLFLUSH保证顺序

- 使用CLFLUSH指令将数据逐出 (Evict) 缓存，以保证顺序

```

STORE data[0] = 0xF00D
STORE data[1] = 0xBEEF
CLFLUSH data[0]
CLFLUSH data[1]
STORE valid = 1

```

- CLFLUSH的缺点
 - 顺序执行，阻塞CPU流水线
 - 会将cacheline无效化 (Cache-Line Flush 的语义)

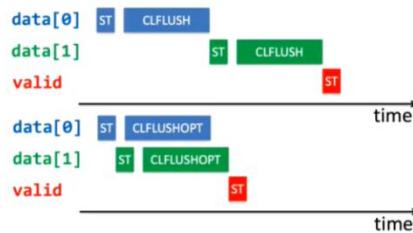


软件提出了很多需求，最终还是需要硬件来解决，在 Intel x86 中提出了新指令 CLFLUSHOPT，它可以和 STORE 同时运行，并且在流水线中同时运行。我们需要 SFENCE 手动顺序执行。

Intel x86 拓展指令集

- 新指令 : CLFLUSHOPT
 - 可以看做可并行执行的CLFLUSH
 - 需要用 sfence 来保证顺序

```
STORE data[0] = 0xFOOD
STORE data[1] = 0xBEEF
CLFLUSHOPT data[0]
CLFLUSHOPT data[1]
SFENCE // explicit ordering point
STORE valid = 1
```

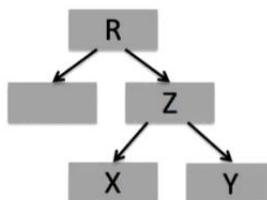


- 新指令 : CLWB
 - Cache Line Write Back
 - 与CLFLUSHOPT类似，区别在于不会将cacheline无效化

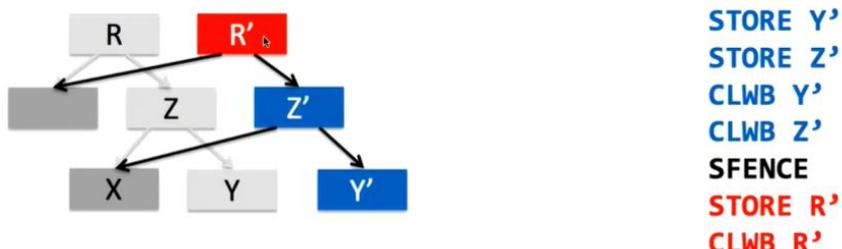
我们还引入了 CLWB, 它不会讲 cacheline 无效化, 也就是下次访问的时候, 还是 cache hit。

NVM 上的写时复制

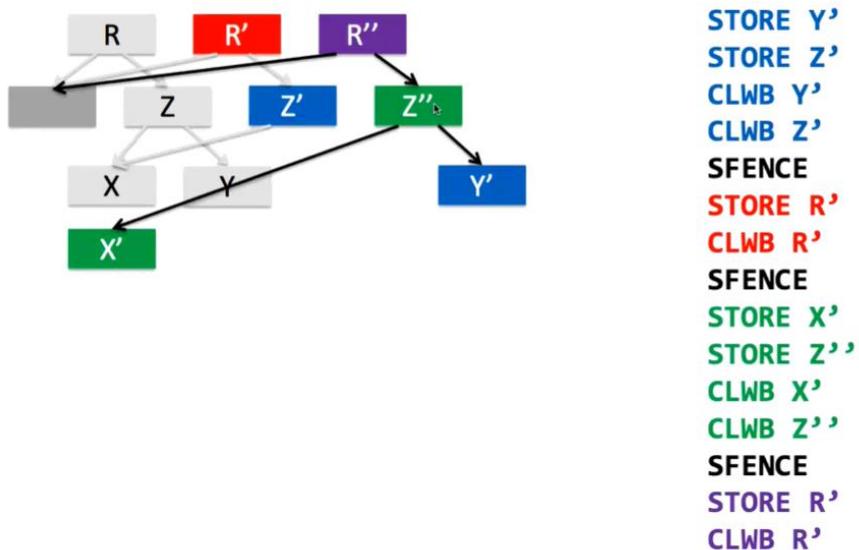
有了这些指令之后, 我们再来看 NVM 的 copy-on-write 怎么实现。这是我们原本的数据结构, 我们想去修改 Y。



我们修改了 Y 以后, 由于 Z 指向 Y 且 R 指向 Z, 所以我们要复制出来一份 Y', Z', R'。最终我们要保证原子性, 所以必须把 R' 的写入放在 Y' 和 Z' 写入完成之后。SFENCE 相当于一个 all-or-nothing 的 commit point。



如果我们此时又把 X 修改成了 X', 同样地, 我们需要复制出一个 Z'' 和 R''。



现在我们不需要再 4K 地写入了，粒度可以很细，在 byte 粒度做到这一点，Y, Z' 可以很大也可以很小。

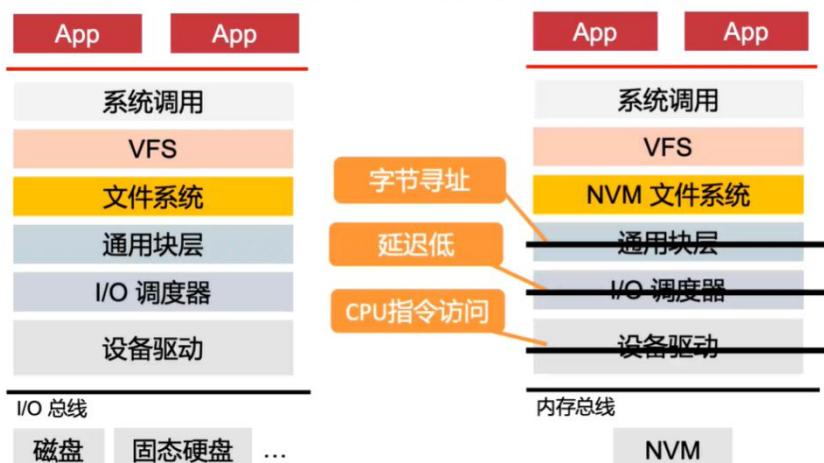
非易失性内存文件系统（Non-volatile Memory File System, NV NFS）

我们明是个内存，但是为什么要提供文件系统的 read/write 呢？因为文件系统接口实在是太受欢迎了，很难撼动它的地位。

通用块层就是把磁盘抽象成 4K 的块的集合，文件系统可以认为下层就是一个 block 的大的数组，至于它是在 page cache 里 hit 了还是 miss 了，文件系统可以不 care。文件系统就是实现 Ext4 和 FAT 等。VFS 都是内存里的数据结构，提供了 vnode 等抽象，无论底层是 FAT 还是 inode，都会转换成 vnode。这样，对上提供的文件系统接口就比较简单了。等我们讲了虚拟化以后，虚拟机里有 6 层，外面还有 6 层。

那么如果我们底层是 NVM，IO 总线变成了内存总线，也不需要驱动和 IO 调度器了，并且它的粒度是 byte 为单位的。

非易失性内存改变存储栈



所以，NVMFS 是在没有下面这几层的情况下构建的。我们就要考虑一致性机制和非易失性内存文件系统之间的关系。

对于原子指令来说，基本上都可以保证内存访问的原子性。写时复制这个方法有很多不同的 FS 里都使用了。有些使用了日志、log-structured、soft-update 等来保证原子性。

一致性技术与非易失性内存文件系统

- 原子指令：ALL
- 写时复制：BPFS^[SOSP '09], PMFS^[EuroSys '14], NOVA^[FAST '16]
- 日志 (Journaling): PMFS, NOVA
- Log-structured: NOVA
- Soft updates: SoupFS^[USENIX ATC '17]

PMFS

我们拿 PMFS 举个例子，它是一个比较老的文件系统，它综合了多种之前提到的技术来优化 NVM 和体系结构。它首先有一个 FS Root, 然后是一些 b-tree 的指针，然后有 inode page、directory inode 和 file inode。通过这样的一个方式结合在一起，应用可以直接通过 mmap 的方式去访问 NVM，不一定要通过 read/write 的方式去访问。

PMFS

- 为NVM和体系结构优化
 - 多种原子更新技术
- 允许应用直接访问NVM
 - DAX mmap
 - DAX: Direct Access
- Wild writes保护

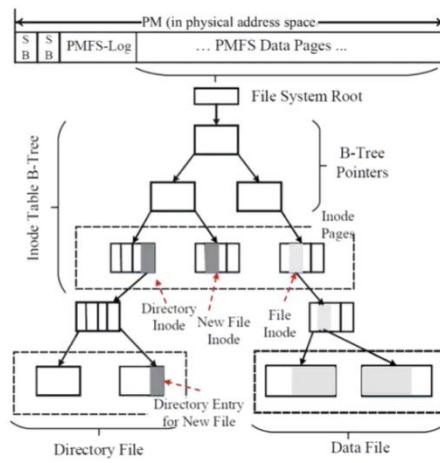


Figure 3: PMFS data layout

PMFS中的一致性保证

- 现有方法
 - 写时复制 (Shadow Paging) 用于文件数据更新
 - 日志 用于元数据更新，如inode
 - Log-structured updates
- NVM专有的方法
 - 原子指令更新 用于小修改

如果我们只要更新 inode 的访问时间，我们就没有必要写到 log 里去，可以直接原子更新。而在之前，我们就必须按照 4K 的粒度进行更新数据。

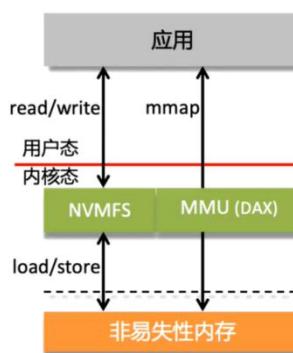
拓展的原子指令更新

- 8字节更新
 - CPU原本就支持8字节的原子更新
 - 更新inode的访问时间
- 16字节更新
 - 使用 `cmpxchg16b` 指令
 - 同时更新inode中的文件大小和修改时间
- 64字节更新
 - 使用硬件事务内存 (HTM)
 - 更新inode中的多个数据

有了这些优化之后，真正应用访问的时候可以通过 `mmap` 的方式，这个性能就非常快。

让应用直接访问NVM

- DAX (direct access)
 - 文件mmap时
 - 通过建立页表映射
 - 将数据页映射给应用



防止 NVM 上的 wild write

wild write 就是俗称的跑飞了。一旦把数据映射给应用程序之后，应用程序可以任意去写，一旦应用程序不小心运行了一条 store 指令，那可能就把内存的数据写坏了。

有两种访问，SMAP 就是防止内核错误地修改用户内存，这个 bit 一旦打开，内核就没有权限修改用户态的内存；还有就是 write protection 机制，一旦要写，我们临时把 write protection bit 关掉，关了以后，内核就可以去写被映射为 read only 的 memory。

如何防止NVM上的wild writes ?

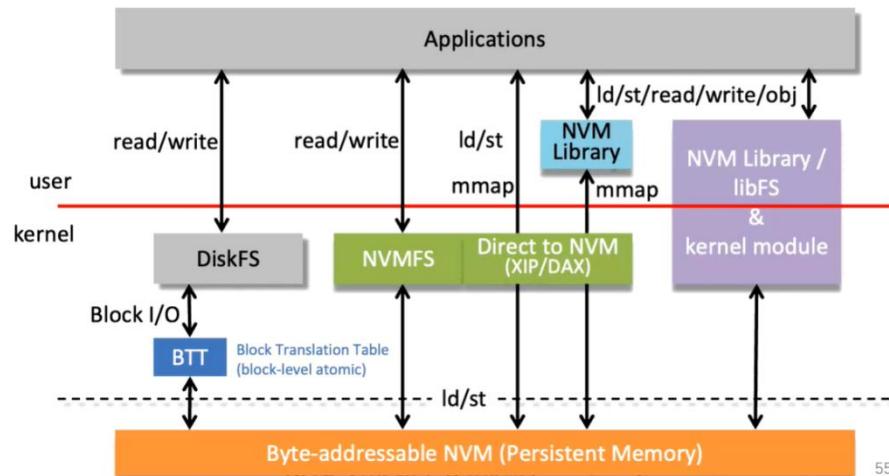
- 程序Bug产生的wild writes会破坏NVM上的数据
- Supervisor Mode Access Protection (SMAP)
 - 防止内核错误地修改用户内存
- Write windows (PMFS提出)
 - 挂载时，NVM映射为只读
 - 写入时，x86的CR0.WP临时设置为0，内核可以修改只读映射

	User	Kernel
User	Process Isolation	SMAP
Kernel	Privilege Levels	Write windows

Table 2: Overview of PM Write Protection

我们最后总结一下，NVM 还在不断发展的过程中。最早的时候，我们就把 NVM 当成 block 的设备来用，每次写还是 4K 操作，和 block 一模一样，这个虽然大材小用，但是保证了兼容性。后来就产生了 NVMFs，对下 load store，堆上提供了文件系统的语义。后来，我们干脆允许应用 mmap，并且我们把 read/write 到 load/store 的转换变成一个用户态的 lib，这样就不会调用到 read/write 的 syscall，提高了性能。

不同NVM文件系统的层次



2022/4/21

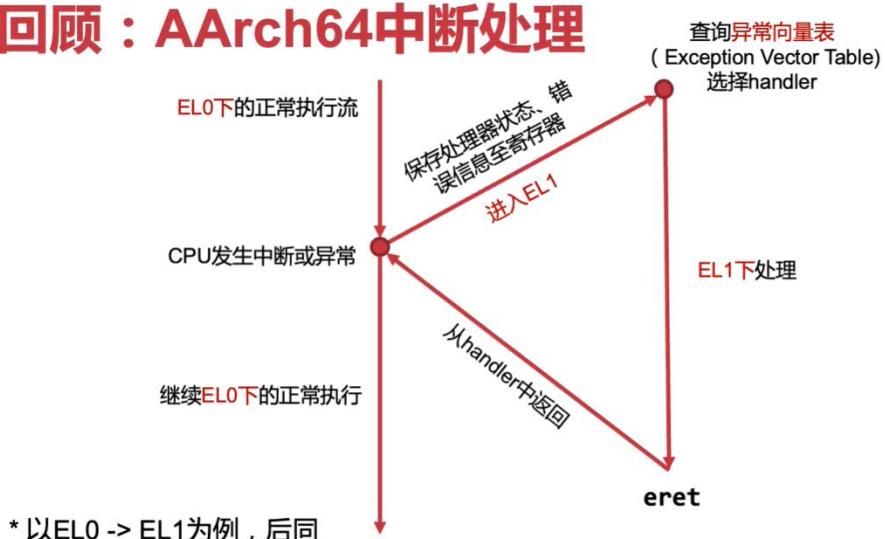
设备管理

我们前几节课在讲文件系统，底层硬件发生了很多变化：磁盘->瓦式磁盘->SSD->NVM，这些设备一开始对上抽象为一个大数组，后来我们又进一步添加了读和写性能的差异体现在文件系统的设计里。但尽管如此，我们还是以 OS 作为绝对的主力，我们没有怎么讲硬件的实现和硬件的交互协议。

有很多种不同的硬件，OS 为了应对设备的异构性做出了什么样的设计，是这节课要讨论的逻辑。

我们先来回顾一下，在异常系统调用这部分，我们介绍了中断的处理。当一个应用程序在执行的时候，突然来了一个中断，就会切到内核态查异常向量表处理中断，处理完再返回用户态继续执行。

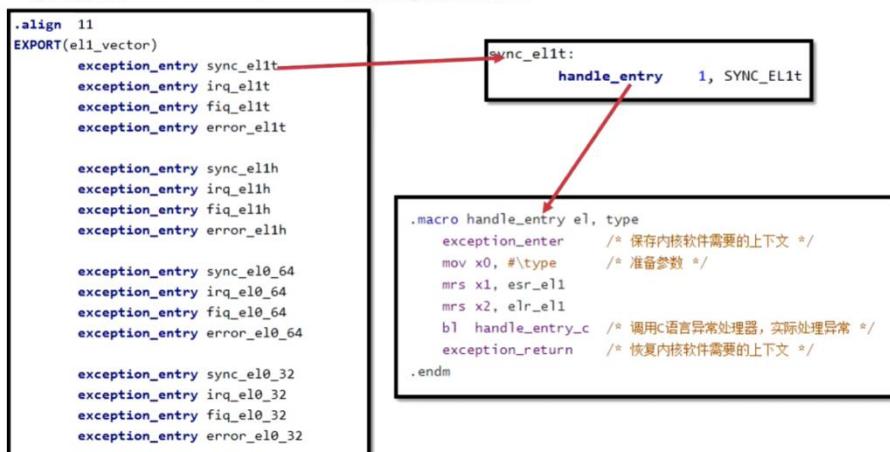
回顾：AArch64中断处理



* 以EL0 -> EL1为例，后同

在 ChCore 中，异常向量表的配置是比较清晰的。我们配置了一个 `vector` 作为入口函数，它会对应到某个具体的 `exception`。每个异常向量都是一个简单的跳转指令。跳转到 `handler`，然后返回到用户态。

回顾：ChCore 异常处理



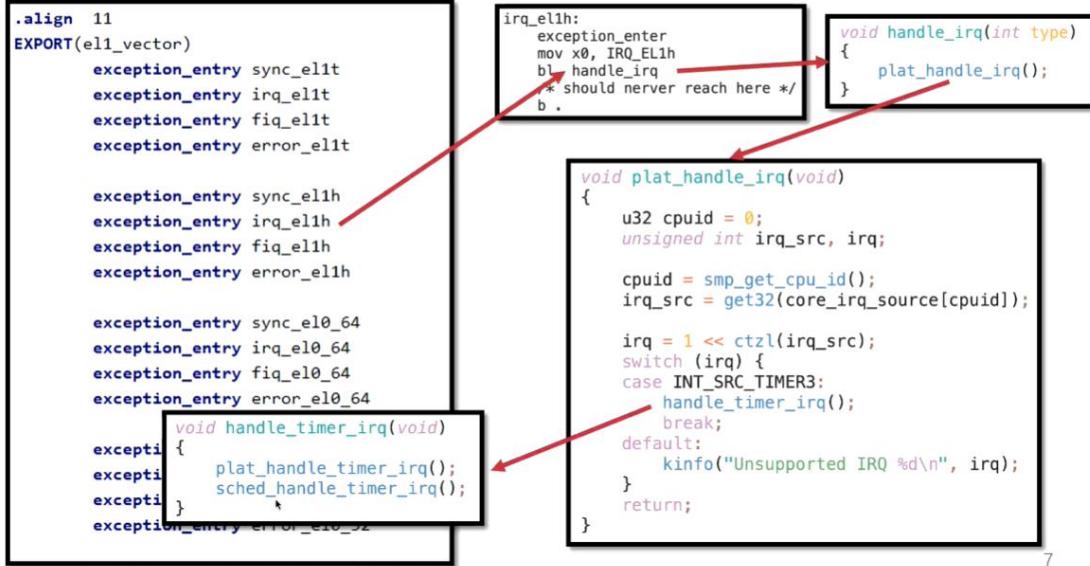
我们发现 `exception_enter` 和 `exception_return` 所做的事情就是保存常规的、特殊的寄存器到栈上，在退出的时候，恢复这些上下文。因为硬件只有一份，所以在使用之前就要保存一下。

回顾：ChCore异常处理



我们继续来看处理时钟中断的例子：

ChCore的中断处理与时钟响应



如果发现是 `irq` 的类型是 `INT_SRC_TIMER3`，就会触发一次调度。

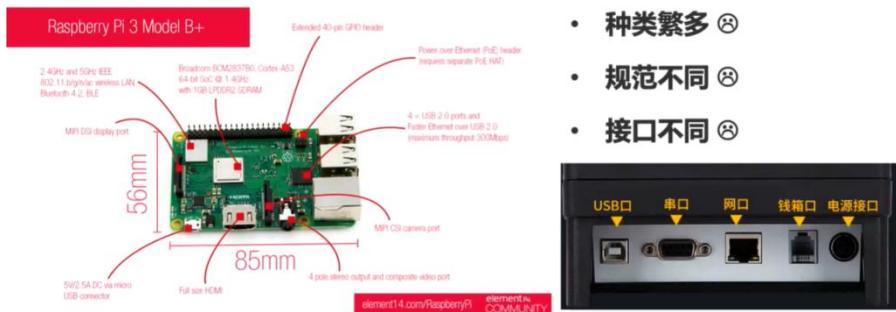
整个 OS 在中断机制之下，一个具体的硬件设备怎么和 OS 交互呢？

操作系统的I/O层次



树莓派上封装了很多设备，比如有一个 GPIO 的 40pin 的 header，在之上我们可以插入各种各样的设备。还有一个 USB 2.0 设备、Power over Ethenet、音响输出、电源输入、蓝牙等。

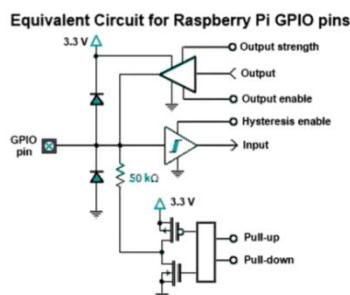
计算机系统上的硬件设备很多



有些设备是非常简单的，比如如果我们要在树莓派上装一个灯，我们可以用最简单的 GPIO 的 LED，我们可以用针脚来控制 LED 的状态，其实我们就是在使用 ARM CPU core 去控制针脚上的 01 变化。这个变化去驱动 LED 等做闪烁。

GPIO LED

- 有专门的“输入/输出”引脚
- 通过引脚控制LED状态
- 每种01组合只呈现一种发光状态



还有一个就是 PS/2 的键盘控制器，是古早的圆口键盘，每次按下一个键的时候，就会

产生一个电信号，最终变成 Scan Code 传递给 CPU。

8042 (PS/2 键盘控制器)

- 电信号→数字信号→编码 (Scan Code)
- 每次只能键入一个字符

思考：如果长按呢？

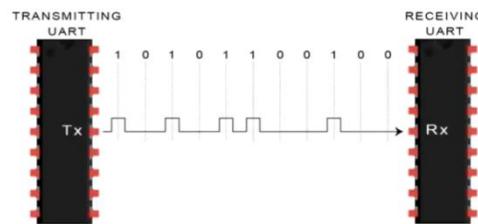


长按就是不断地输入同一个字符的 Scan Code。

还有一个就是 UART，在调试 OS 的时候，或者没有别的设备的时候，串口可以取代显示器做输出。所以它相对显示器来说，会更加简单。串口在一些设备上还是非常有必要的。它是一种异步的收发传输器，每次我们发的时候就不能收，收的时候就不能发。如果我们需要同时收发，我们就需要建立两个信道才能做到这一点，这个速度也比较慢，常见的是 9600 的波特率。

UART (串口)

- 通用异步收发传输器
 - Universal Asynchronous Receiver/Transmitter
- 半双工
- 每次只能传输一个字符



Flash 闪存

- 按照页/块的粒度进行读写/擦除

- 支持页/块随机访问



Ethernet 网卡

- 每次传输一帧数据（以太网帧）

- Wifi、蓝牙类似



思考题

- 树莓派上的这些设备可以和前面提到的哪些设备归为一类？

- SD 存储卡
- RTC 实时时钟
- DS18B20 温度传感器
- CSI 摄像头
- 板载无线蓝牙



存储卡可以和 Flash 分一类，都是存储设备。温度传感器和键盘很像，每次传一个信息过来。摄像头和网卡、串口很像，不断有数据过来。这些设备可以分成很多类，难点是怎么把分的类搞得比较简单。一种常见的分类方式就是空间和时间。

Flash 和存储卡在空间上可以来回地访问，不存在数据的新旧之分，我们就叫做块设备。

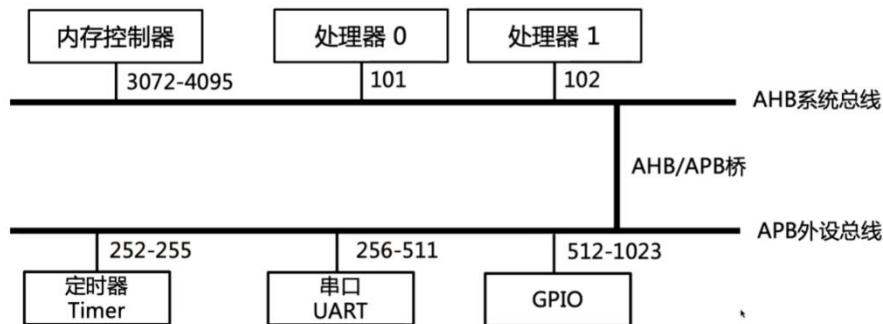
还有一类是时间上有新旧的，同样的地方产生不同的数据，这就叫做字符设备/字符流设备。字符设备不一定比块设备小，比如高清摄像头每秒钟产生的数据量是很大的。

有一类比较特殊的就是网络设备，虽然它可以归到字符设备，但是我们把它单独拎出来。

设备和 CPU 是怎么连接的

我们写程序本质上就是和 CPU 打交道，我们必须通过代码让 CPU 作为我们的代理去和设备打交道。整个系统里面，设备和 CPU 是通过总线连接在一起的。这个总线有很多种，在 ARM 上主要分为 AHB 系统总线和 APB 外设总线。AHB 连接的是 CPU 和内存，速度会快一些。APB 的性能相对会低一些。APB 和 AHB 之间就通过 bridge 可以连接到一起，bridge 可以有很多个。

硬件总线：以AMBA为例



总线也不是简单的一根线，是有层次结构的，如快速、慢速、bridge。控制总线，就是当我们发送数据之后，需要发送 ACK 的时候，ACK 走的就是控制总线。

仲裁协议和以太网当时讲的载波监听冲突检测是一样的。如果两个人同时发，那么两个人发送的数据都失效。总线仲裁器就是来选择哪些模块可以使用总线。

硬件总线的特点

- **一组电线**
 - 将各个I/O模块连接到一起，包含了地址总线、数据总线和控制总线
- **使用广播**
 - 每个模块都能收到消息
 - 总线地址：标识了预期的接收方
- **仲裁协议**
 - 决定哪个模块可以在什么时间收发消息
 - 总线仲裁器：用于选择哪些模块可以使用该总线

如果有共享时钟，那么我们只需要在上升沿之后读一下数据就可以得到数据了。这样一个周期我们就可以传递一次数据。但是很多时候在发送方和接收方之间是没有时钟的，所以我们要使用 ACK 这个显式的信号。

同步 VS. 异步

- **同步数据传输**
 - 源 (Source) 和目标 (destination) 借助**共享时钟**进行协作
 - 例子：DDR内存访问
- **异步数据传输**
 - 源 (Source) 和目标 (destination) 借助**显式信号**进行协作
 - 例子：对信号的确认 (ack)

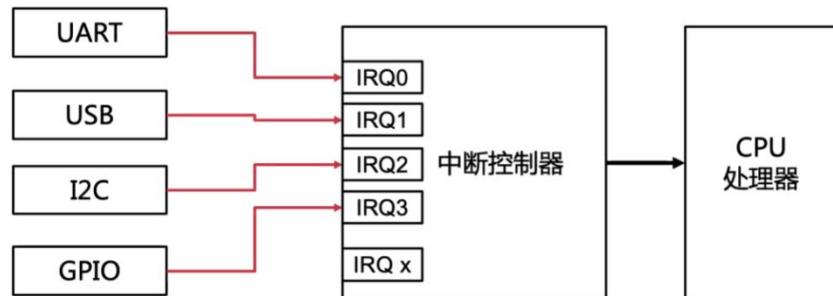
发送方发一个 ready 信号，源就可以释放总线了。数据是通过总线来传播的。

总线事务

- ① 源（发送方）获取总线的使用权（具有排他性）
- ② 源（发送方）将目标（接收方）的地址写到总线上
- ③ 源（发送方）发出 READY 信号，提醒其他模块（广播）
- ④ 目标（接收方）在拷贝完数据后，发出 ACKNOWLEDGE 信号
 - 同步模式下，无需 READY 和 ACKNOWLEDGE，只要在每个时钟周期进行检查即可
- ⑤ 源（发送方）释放总线

设备和 CPU 的互联的线还有一条很细的中断线。CPU 上有一个针脚，专门是用来设置中断的。这些设备都有一条线连到中断控制器，再由中断控制器连一条线到 CPU。CPU 进到中断处理函数中以后，再来看是谁引发的中断。

中断线

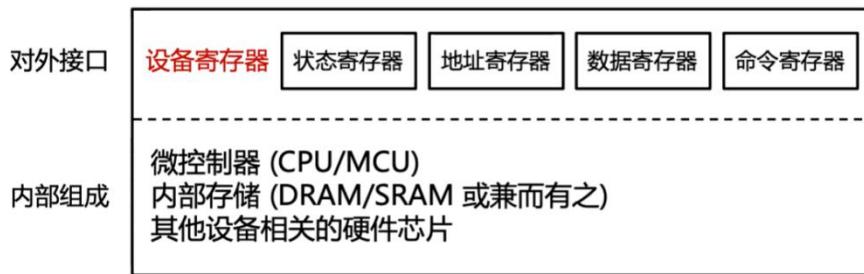


设备和 CPU 连接主要分为总线（数据线、地址线、控制线）和中断线。

CPU 和设备是怎么交互的

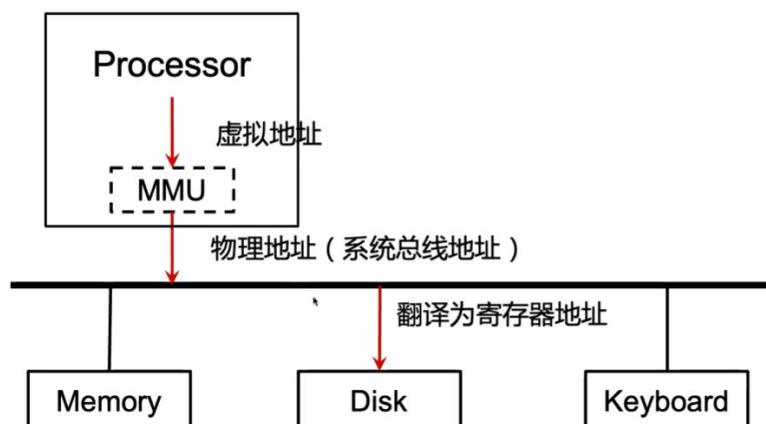
一个硬件设备对于 CPU 来说，有一个很简单接口：设备寄存器。在设备的内部，可以很复杂，比如有自己的微处理器、DRAM、Flash 颗粒等。但是它对 CPU 提供的接口就是一串寄存器。

硬件设备的接口：设备寄存器



一个非常简单的方法就是 MMIO，就是把设备上的寄存器映射成一个虚拟内存地址。MMU 是把虚拟地址翻译成物理地址。keyboard 上的寄存器要能变成物理地址的形态，物理地址不仅仅是内存地址，也可以被翻译成寄存器的地址。所以，CPU 就可以通过 mmio 这样的 load/store 的指令去控制外部设备。

内存映射 I/O (MMIO)



这是 ChCore 上的 UART MMIO。我们可以看到其中的 get32 和 put32，实际上就是一个 load 和 store 指令。uart_recv 就是从串口上收数据，uart_send 就是从串口上发数据。映射到虚拟空间之后，OS 就可以直接使用 load & store 指令对寄存器进行操作。

ChCore的UART MMIO

```
u32 pl011_nb_uart_recv(void)
{
    if (!(get32((u64)UART_PPTR + UART_FR)
        & UART_FR_RXFE))
        return (get32((u64)UART_PPTR + UART_DR) & 0xFF);
    else
        return NB_UART_NRET;
}

void pl011_uart_send(u32 ch)
{
    /* Wait until there is space in the FIFO or device is disabled */
    while (get32((u64)UART_PPTR + UART_FR)
        & UART_FR_TXFF) {
    }
    /* Send the character */
    put32((u64)UART_PPTR + UART_DR, (unsigned int)ch);
}
```

```
BEGIN_FUNC(get32)
    ldr w0,[x0]
    ret
END_FUNC(get32)
```

- **MMIO: 复用ldr和str指令**
 - 映射到物理内存的特殊地址段

```
BEGIN_FUNC(put32)
    str w1,[x0]
    ret
END_FUNC(put32)
```

作为 load 指令，我们必须考虑 volatile。当我们读内存数据的时候，如果我们没有写的情况下，频繁读同一个内存数据，作为编译器来说，它认为你在写无用的代码。但是对于设备来说，并不是这样，在设备上有一个寄存器，这个寄存器很有可能是告诉你有没有 ready。这个值多次读是可能发生变化的，如果我们没有加 volatile 关键字，编译器会认为下面两次读内存是多余的。加了 volatile，CPU 就知道这块内存可能是映射到设备寄存器上了，不会做奇怪的优化。

MMIO地址应使用Volatile关键字

```
void main(void)
{
    void          *pdev = (void *) 0x40400000;
    size_t        size = (1024*1024);
    int          *base;
    volatile int  *pcid, cid;

    base = mmap(pdev, size, PROT_READ|PROT_WRITE,
                MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
    if (base == MAP_FAILED) errx(1, "mmap failure");

    pcid = (int *) (((void *) base) + 0xf0704);
    cid = *pcid;
    printf("cid = %d\n", cid);
    cid = *pcid;
    printf("cid = %d\n", cid);
    munmap(base, size);
}
```

若不加 volatile，编译器会认为这两个printf()
多余，并消除第二个内存加载操作

可编程 IO 指的就是由 CPU 发起的访问设备的行为。除了 MMIO 以外，还有一类叫做 Port IO。它有专门的指令，我们主要集中在 MMIO 里。

可编程I/O (Programmable I/O)

- 形式1：MMIO (Memory-mapped I/O)

- 将设备映射到连续物理内存中
- 使用内存访问指令 (load/store)
- 行为与内存不完全一样，读写有副作用
- 在Arm、RISC-V等架构中使用

- 形式2：PIO (Port I/O)

- IO设备具有独立的地址空间
- 使用专门的PIO指令 (in/out)
- 在x86架构中使用

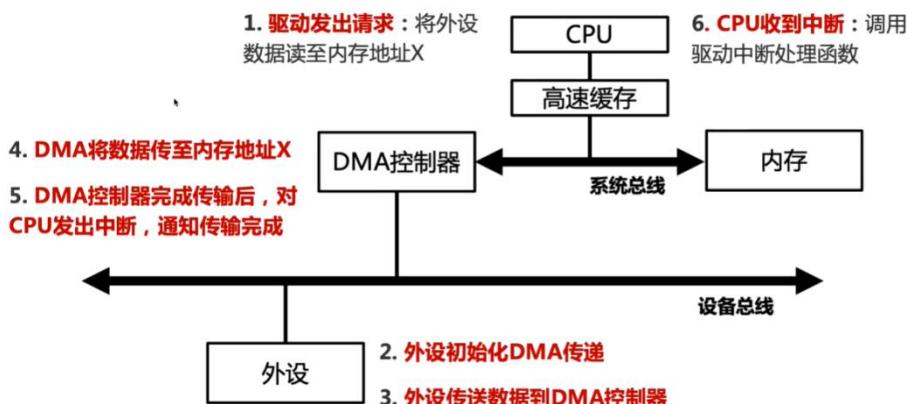
如果我们发两个控制指令也就算了，如果我们想通过 PIO 的方式从设备中获取数据，这显然很慢，这就需要我们的 DMA 了。

是否存在更高效的数据交互方式？

- 可编程 I/O (Programmable I/O)
 - 通过CPU in/out 或 load/store 指令
 - 消耗CPU时钟周期和数据量成正比
 - 适合于简单小型的设备
- 直接内存访问 (Direct Memory Access, DMA)
 - 设备可直接访问总线
 - DMA与内存互相传输数据，传输不需要CPU参与
 - 适合于高吞吐量I/O

有了 DMA 之后，我们就可以让驱动发出一个请求。整个过程中，除了第一步 CPU 依旧参与了，剩下的 2345 过程都是绕开 CPU 的。

DMA

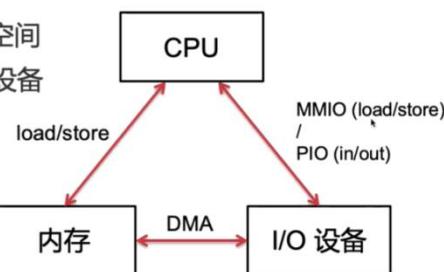


假如一个应用程序想直接发起一个 DMA 操作，显然它必须通过内核发起。如果它一定要发起，说明应用程序可能通过 DMA 绕开页表的权限管理。

CPU访问设备方式小结

• MMIO

- 将设备寄存器映射到物理地址空间
- CPU通过读写设备寄存器操作设备



• DMA

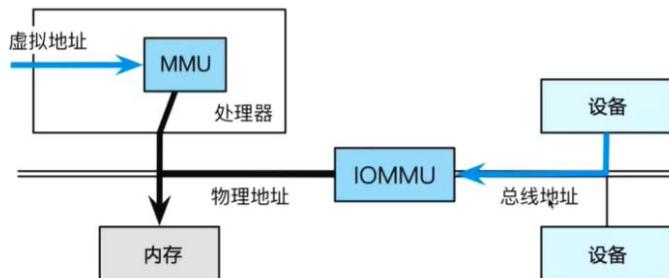
- 设备使用物理地址访问内存
- 思考：如何保证设备访存的安全性？

所以，我们需要加一层 IOMMU 的抽象。

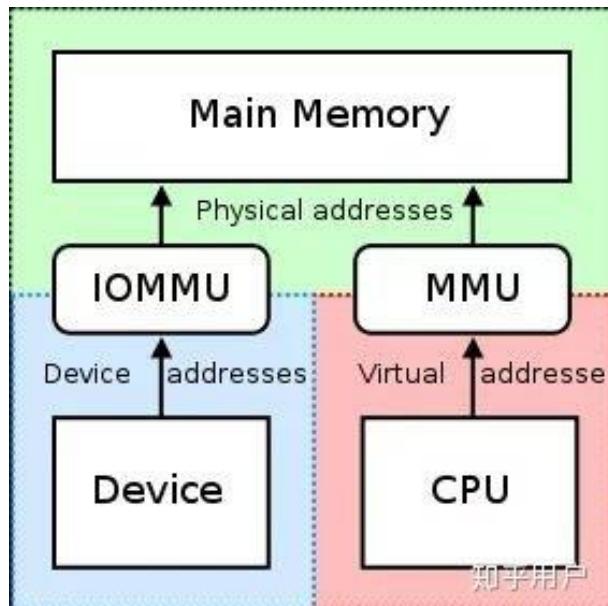
IOMMU

- 避免设备直接使用物理地址访问内存

- 设备所使用的地址，由IOMMU翻译为实际的物理地址
- 广泛应用于虚拟机场景中（允许虚拟机独占某个设备）



在计算机领域，IOMMU（Input/Output Memory Management Unit）是一个内存管理单元（Memory Management Unit），它的作用是连接 DMA-capable I/O 总线（Direct Memory Access-capable I/O Bus）和主存（main memory）。传统的内存管理单元会把 CPU 访问的虚拟地址转化成实际的物理地址。而 IOMMU 则是把设备（device）访问的虚拟地址转化成物理地址。为了防止设备错误地访问内存，有些 IOMMU 还提供了访问内存保护机制。参考下图：



IOMMU 的一个重要用途是在虚拟化技术（virtualization）：虚拟机上运行的操作系统（guest OS）通常不知道它所访问的 host-physical 内存地址。如果要进行 DMA 操作，就有可能破坏内存，因为实际的硬件（hardware）不知道 guest-physical 和 host-physical 内存地址之间的映射关系。IOMMU 根据 guest-physical 和 host-physical 内存地址之间的转换表（translation table），re-mapping 硬件访问的地址，就可以解决这个问题。

[1]<https://www.zhihu.com/question/325947168/answer/694085423>

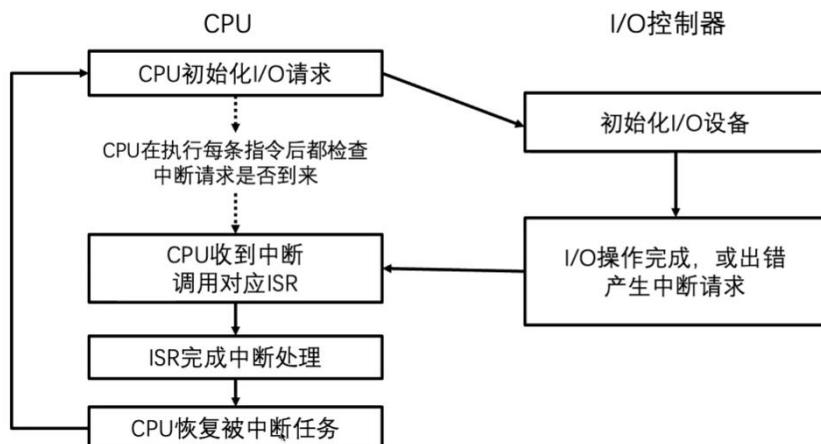
处理器内部是有高速缓存，当 DMA 变化的时候，此时 CPU 的 cache 里如果还有数据怎

么办呢？这意味着 cache 的数据是旧的，而 DMA 中的内存的数据是最新的。

思考题：DMA的内存一致性

- 现代处理器通常带有高速缓存 (CPU Cache)
- 当DMA发生时，DMA缓冲区的数据仍在cache中怎么办？
- 解决方法：
 - 方案1：将DMA区域映射为non-cacheable
 - 方案2：由软件负责维护一致性，软件主动刷缓存
 - 部分架构在硬件上保证了DMA一致性，如总线监视技术

CPU中断处理流程



AArch64的中断分类

- IRQ (Interrupt Request)
 - 普通中断，优先级低，处理慢
- FIQ (Fast Interrupt Request)
 - 一次只能有一个FIQ
 - 快速中断，优先级高，处理快
 - 常为可信任的中断源预留
- SError (System Error)
 - 原因难以定位、较难处理的异常，多由异步中止 (Abort) 导致
 - 如从缓存行 (Cacheline) 写回至内存时发生的异常

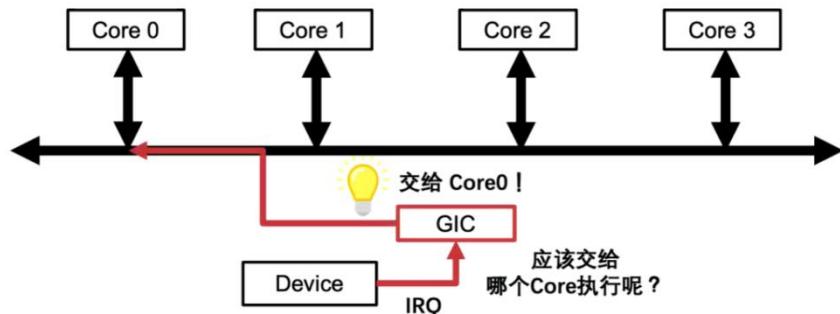
连接CPU的不同针脚

可在中断控制器 (Interrupt Controller) 中配置

在多核场景下，我们不希望一个设备的中断打断所有核，所以我们需要通过 GIC (generic interrupt controller) 来判断应该交给哪个核去执行。

问题：多核CPU如何处理中断？

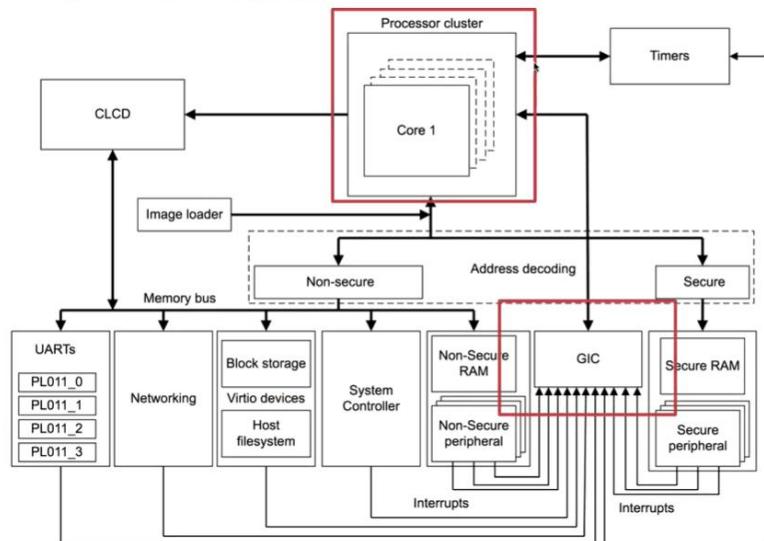
- 如何避免中断一次性打断所有核呢？



GIC 的内部结构如下图所示：

GIC (ARM 中断控制器)

ARM 中断控制器——GIC



GIC

- 全写 : Generic Interrupt Controller
- 组件1 : Distributor
 - 负责全局中断的分发和管理
- 组件2 : CPU Interface
 - 类似 “门卫” , 判断中断是否要发给CPU处理

Distributor

- **中断分发器 :**
 - 将当前最高优先级中断转发给对应CPU Interface
- **寄存器 : GICD**
- **作用 :**
 - 中断使能
 - 中断优先级
 - 中断分组
 - 中断触发方式
 - 中断的目的core

CPU Interface

- **CPU接口 :**
 - 将GICD发送的中断，通过IRQ中断线发给连接到 interface 的核心
- **寄存器 : GICC**
- **作用 :**
 - 将中断请求发给CPU
 - 配置中断屏蔽
 - 中断确认 (acknowledging an interrupt)
 - 中断完成 (indicating completion of an interrupt)
 - 核间中断 (Inter-Processor Interrupt , IPI)，用于核间通信

GIC

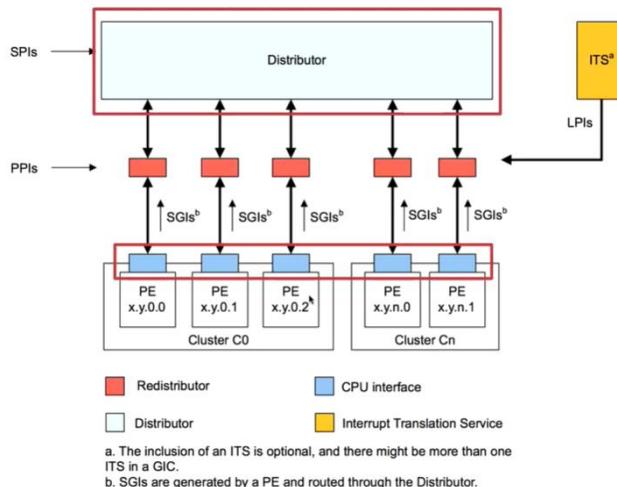


Figure 3-2 GIC logical partitioning with an ITS

ARM中断的生命周期

- ① **Generate** : 外设发起一个中断
- ② **Distribute** : Distributor对收到的中断源进行仲裁，然后发送给对应的CPU Interface
- ③ **Deliver** : CPU Interface将中断传给core
- ④ **Activate** : core读 GICC_IAR 寄存器，对中断进行确认
- ⑤ **Priority drop**: core写 GICC_EOIR 寄存器，实现优先级重置
- ⑥ **Deactivate** : core写 GICC_DIR 寄存器，来无效该中断

最高优先级的就是复位键，不能说按下 reset 之后，CPU 说要先做别的事情。

问题：多个中断同时发生怎么办？

- **中断优先级：**
 - 当多个中断同时发生时（NMI、软中断、异常），CPU首先响应高优先级的中断
- **ARM Cortex-M 处理器的中断优先级如下所示：**

类型	优先级（值越低，优先级越高）
复位 (reset)	-3
不可屏蔽中断 (NMI)	-2
硬件故障 (Hard Fault)	-1
系统服务调用 (SVcall)	可配置
调试监控 (debug monitor)	可配置
系统定时器 (SysTick)	可配置
外部中断 (External Interrupt)	可配置

低优先级的中断在 handler 里，允许高优先级的中断抢占。

中断嵌套

- **中断也能被“中断”**
- **在处理当前中断 (ISR) 时：**
 - 更高优先级的中断产生；或者
 - 相同优先级的中断产生
- **那么该如何响应？**
 - 允许高优先级抢占
 - 同级中断无法抢占
- **ARM的FIQ能抢占任意IRQ，FIQ本身不可抢占**

如何禁止中断被抢占？

- **中断屏蔽：**

- 屏蔽全局中断：不再响应任何外设请求
 - 屏蔽对应中断：只停止对应IRQ的响应

- **屏蔽策略：**

- 屏蔽全局中断：
 - 1. 系统关键步骤（原子性）
 - 2. 保证任务响应的实时性
 - 屏蔽对应中断：通常都是这种情况，对系统的整体影响最小

在高频网络包的情况下，可能包和包之间的间隔只有 150 个 cycle，但是每次我们保存寄存器就需要 100 个 cycle，就很容易造成活锁。

高频中断的问题：活锁

- **网络场景下的中断使用（网卡设备）**

- 当每个网络包到来时都发送中断请求时，OS可能进入活锁
 - **活锁**：CPU只顾着响应中断，无法调度用户进程和处理中断发来的数据

- **解决方案：合二为一（中断+轮询），兼顾各方优势**

- 默认使用中断
 - 网络中断发生后，使用轮询处理后续达到的网络包
 - 如果没有更多中断，或轮询中断超过时间限制，则回到中断模式
 - 该方案在Linux网络驱动中称为 **NAPI (New API)**

中断合并 (Interrupt Coalescing)

- **中断合并：**

- 设备在发送中断前，需要等待一小段时间
 - 在等待期间，其他中断可能也会马上到来，因此将多个中断合并为同一个中断，进而降低频繁中断带来的开销

- **注意：**

- 等待过长时间会导致中断响应时延增加
 - 这是系统中常见的“折衷”（trade-off）

设备驱动

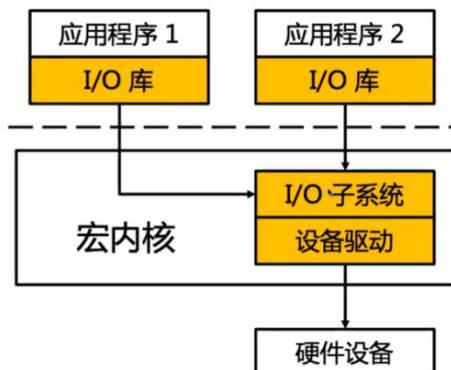
讲完硬件和 CPU 的连接和交互，接下来我们就来讲设备驱动。

设备驱动

- **设备驱动**
 - 专门用于操作硬件设备的代码集合
 - 通常由硬件制造商负责提供
 - 驱动程序包含中断处理程序
- **驱动特点**
 - 和设备功能高度相关
 - 不同设备间的驱动复杂度差异巨大
 - 是操作系统 bugs 的主要来源

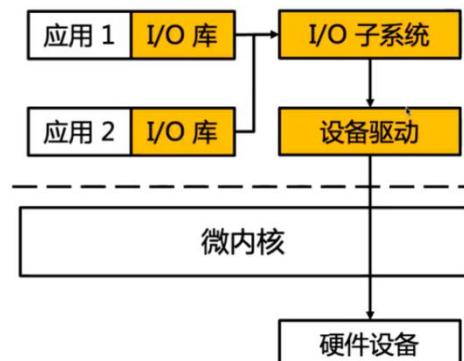
宏内核I/O架构

- **宏内核I/O架构**
 - 设备驱动在内核态
 - 优势：通常性能更好
 - 劣势：容错性差
 - 中断形式为内核ISR
- **案例：**
 - Linux、BSD
 - Windows



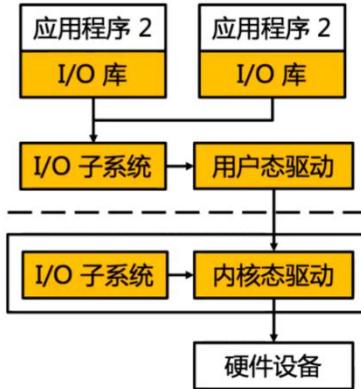
微内核I/O架构

- **微内核I/O架构**
 - 设备驱动主体在用户态
 - 优势：可靠性和容错性更好
 - 劣势：IPC性能开销
 - 中断为用户态驱动线程
- **案例：**
 - 谷歌Fuchsia手机系统
 - ChCore微内核系统



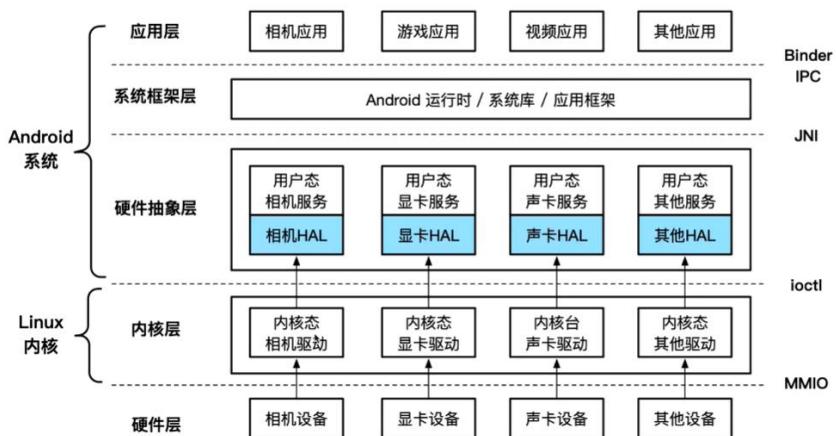
混合I/O架构

- **混合I/O架构**
 - 设备驱动分解为用户态和内核态
 - 优势1：驱动开发和Linux内核解耦
 - 优势2：允许驱动以闭源形式存在，保护硬件厂商的知识产权
- **案例：**
 - 谷歌安卓系统：硬件抽象层（HAL）
 - 华为鸿蒙系统：硬件驱动框架（HDF）



安卓在驱动之上做了一层 HAL（硬件抽象层），底下的内核的驱动是必须开源的，但是 HAL 就可以不开源，而且用户态的 HAL 可以保证一定的稳定性。

案例：安卓的硬件抽象层



驱动模型的好处

- **电源管理：**
 - 描述设备在系统中的拓扑结构（树形结构）
 - 保证能正确控制设备的电源，先关闭设备和再关闭总线
- **驱动开发者：**
 - 允许同系列设备的驱动代码之间的复用
 - 将设备和驱动联系起来，方便相互索引
- **系统管理员：**
 - 帮助用户枚举系统设备，观察设备间拓扑和设备的工作状态

案例：Linux驱动模型



Linux 做的事情就是提供一个驱动框架，让驱动开发工作尽可能少。

Linux Device Driver Model (LDDM)

- 支持电源管理与设备的热拔插
- 利用sysfs向用户空间提供系统信息
- 维护内核对象的依赖关系与生命周期，简化开发工作
 - 驱动人员只需告诉内核对象间的依赖关系
 - 启动设备时会自动初始化依赖的对象，直到启动条件满足为止

I/O 子系统

为什么需要I/O子系统？

- 根据不同需求和场景，出现了大量设备：
 - 通信、存储、智能加速器、人机交互等
 - 数以千计的设备类型，个性千差万别
- 每种设备有自己的协议、规范：
 - 如何标准化设备接口？
- 设备的异步不可预测性和慢速性：
 - 如何提高应用程序的设备读写效率？
- 设备的不可靠性（介质失效或传输错误）：
 - 如何得知设备的状态并修复错误？



I/O 子系统的目标

- 提供统一接口，涵盖不同设备：

- 如下代码对各种设备通用：

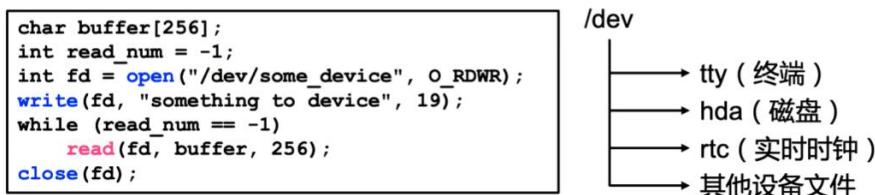
```
FILE fd = fopen("/dev/something", "rw");
for (int i = 0; i < 10; i++) {
    fprintf(fd, "Count %d\n", i);
}
close(fd);
```

- 满足I/O硬件管理的共同需求，提供统一抽象

- 管理硬件资源；隐藏硬件细节

统一抽象——设备文件

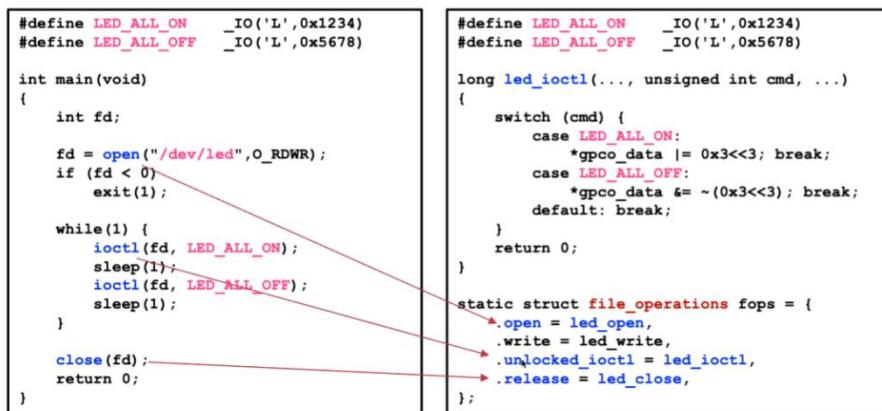
- 为应用程序提供的相同的设备抽象：设备文件
- 操作系统将外设细节和协议封装在文件接口的内部
- 复用文件系统接口：open(), read(), write(), close, etc.



最终就可以通过 open/read/write 去控制。但是如果我们想把网卡的速度从 1000M 设置成 100M，这样的问题很多很多，核心原因就是设备千奇百怪，所以我们提供了 ioctl。

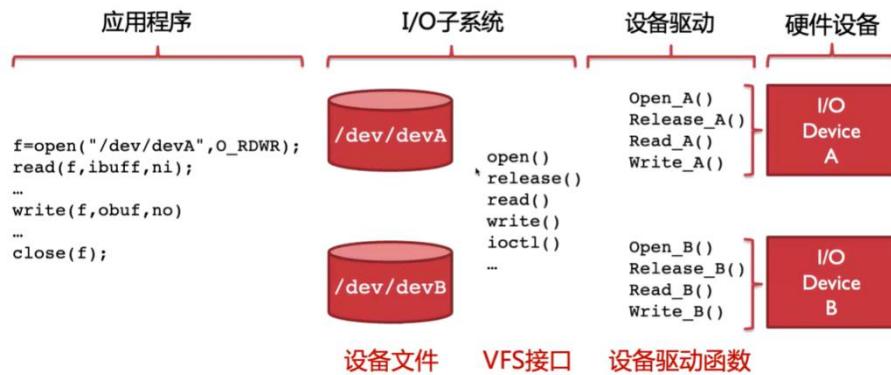
设备操作的专用接口：ioctl

- 应用程序和驱动程序事先协商好“操作码”和对应语义



I/O 子系统中有些是通用的。

设备文件操作与设备驱动函数的对接

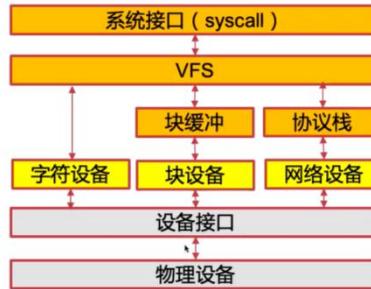


VFS 就是一个典型的 I/O 子系统，它不关心底下的实现是什么，它是一个设备无关的逻辑。

设备的逻辑分类

- **Linux设备分类**

- 字符设备
- 块设备
- 网络设备



字符设备 (cdev)

- **例子 :**

- 键盘、鼠标、串口、打印机等
- 大多数伪设备 : /dev/null, /dev/zero, /dev/random

- **访问模式 :**

- **顺序访问** , 每次读取一个字符
- 调用驱动程序和设备直接交互

- **文件抽象 :**

- open(), read(), write(), close()

块设备 (blkdev)

- **例子：**

- 磁盘、U盘、闪存等（以存储设备为主）

- **访问模式：**

- 随机访问，以块为单位进行寻址（如512B、4KB不等）
- 通常为块设备增加一层缓冲，避免频繁读写I/O导致的慢速

- **通常使用内存抽象：**

- 内存映射文件(Memory-Mapped File)：mmap()访问块设备
- 提供文件形式接口和原始I/O接口（绕过缓冲）

内存映射文件：mmap访问磁盘

- 可减少系统调用次数
- 可减少数据的拷贝次数

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
    const char *message = "memory mapped file";

    int fd = open("mmap.c", O_RDWR);

    // 标准I/O：用lseek()寻址，再用write()写入数据
    lseek(fd, 0x10, SEEK_SET);
    write(fd, message, strlen(message));

    close(fd);
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>

int main()
{
    struct stat sbuf;
    const char *message = "memory mapped file";

    int fd = open("mmap.c", O_RDWR);
    fstat(fd, &sbuf); // 获取文件长度

    // 内存映射文件：用指针直接寻址，将数据写入内存
    char *data = mmap((caddr_t)0, sbuf.st_size,
                      PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    strcpy(data+0x10, message);

    close(fd);
    return 0;
}
```

网络设备 (netdev)

- **例子：**

- 以太网、WiFi、蓝牙等（以通信设备为主）

- **访问模式：**

- 面向**格式化报文**的收发
- 在驱动层之上维护多种协议，支持不同策略

- **套接字抽象：**

- socket(), send(), recv(), close(), etc.

设备的缓存管理

这不仅仅是块设备需要缓冲。Page Cache 就是在 I/O 这一层，尽可能用内存减少对设备的访问。它是属于单缓冲，整个系统只有一块，它要去匹配读写不对称的问题。

设备的缓冲管理：单缓冲区

- **问题：**
 - 读写性能不匹配：慢速的存储设备 vs. 高速的CPU
 - 读写粒度不匹配：小数据的访问存在读写放大的问题
- **解决方法：**
 - 开辟内存缓冲区，避免频繁读写I/O
- **单缓冲区例子：Linux的page cache**



现在就不再是 cache 了，叫做 buffer。比如说我们很关心玩游戏的 FPS，我们在渲染一帧的时候，就往后置缓冲区里去写。写完之后，我们交换前置缓冲区和后置缓冲区，这样我们又多了一个后置缓冲区可以去写。这样 GPU 就可以读我们写好的东西，在读的时候，我们 CPU 写另一块区域。这样允许我们 GPU 和 CPU 同时工作。

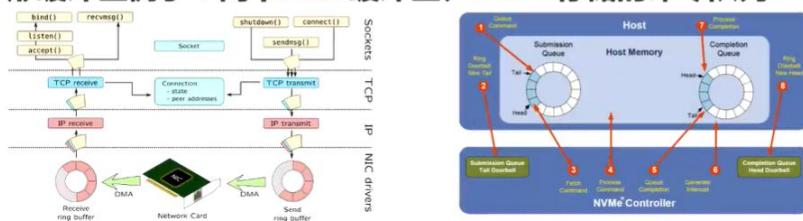
设备的缓冲管理：双缓冲区

- 维护两个缓冲区，轮流使用
- 第一个缓冲区被填满但没被读取前，使用第二个缓冲区填充数据
- 双缓冲区例子：显存刷新，防止屏幕内容出现闪烁或撕裂
 - 前置缓冲区被读取后，通过“交换”（swap）将前置和后置身份互换
- 游戏中甚至启用三重缓冲



设备的缓冲管理：环形缓冲区

- 容许更多缓冲区存在，提高I/O带宽
- 组成：一段连续内存区域+两个指针，读写时自动推进指针
 - 读指针：指向有效数据区域的开始地址
 - 写指针：指向下一个空闲区域的开始地址
- 环形缓冲区例子：网卡DMA缓冲区、NVMe存储的命令队列



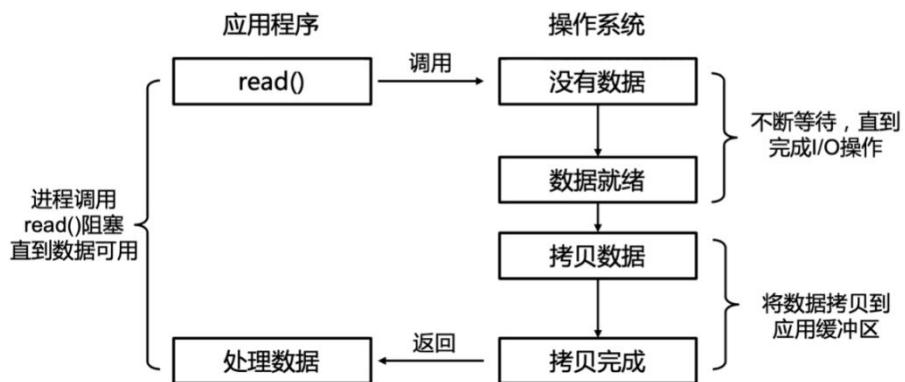
设备的缓冲区管理小结

- 多种缓冲区实现方式
 - 单缓冲区：块设备的缓冲区
 - 双缓冲区：常用于游戏或流媒体的显存同步
 - 环形缓冲区：网卡的数据交互和NVMe存储的命令交互
- 思考题：缓冲区是否总能提高性能？
 - 缓冲区意味着数据的多次拷贝，使用过多反而损伤性能

问题：如何同时监听多个设备？

- 如果要支持多个fd怎么办？
 - 阻塞I/O：一旦一个阻塞，则其余无法响应
- 改成非阻塞可以么？
 - 非阻塞I/O：需要程序不断轮询设备情况：浪费CPU
- 能否采用异步通知机制？让内核主动来通知？
 - 异步I/O：允许程序先做其他事，等设备数据就绪再接收通知
 - 多路复用I/O：仍为阻塞，但一旦设备数据就绪就收到通知

I/O模型：阻塞I/O模型

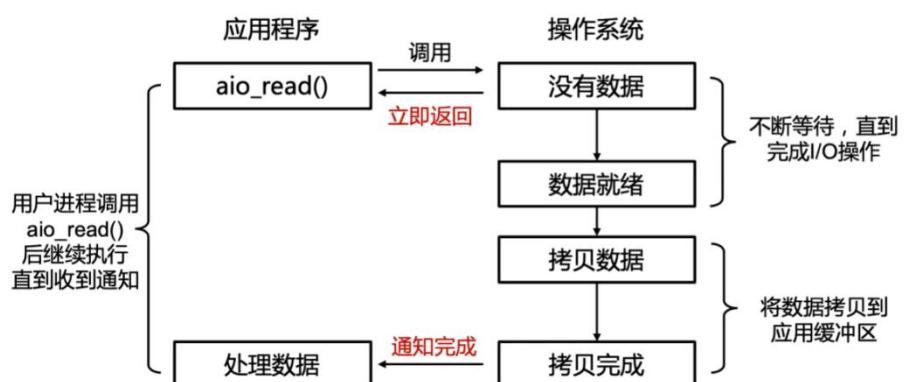


I/O模型：非阻塞I/O模型

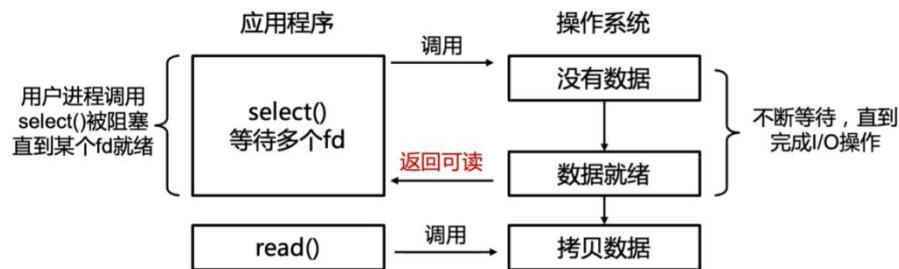


异步 I/O 模型适合事件驱动的应用程序。

I/O模型：异步I/O模型



I/O模型：I/O多路复用模型



I/O模型小节

- 阻塞I/O：一直等待**
 - 进程请求读数据不得，将其挂起，直到数据来了再将其唤醒
 - 进程请求写数据不得，将其挂起，直到设备准备好了再将其唤醒
- 非阻塞I/O：不等待**
 - 读写请求后直接返回（可能读不到数据或者写失败）
- 异步I/O：稍后再来**
 - 等读写请求成功后再通知用户
 - 用户执行并不停滞（类似DMA之于CPU）
- I/O多路复用：同时监听多个请求，只要有一个就绪就不再等待**

I/O 库



I/O库

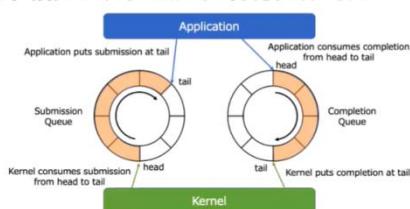
- 以共享库的形式，和应用程序直接链接
- 简化应用程序I/O开发的复杂度
- 提供更好的性能和更灵活的I/O管理能力
- 例子：
 - glibc：提供用户态I/O缓冲区管理
 - Linux AIO和io_uring：支持异步I/O

案例：glibc的buffer I/O

- **fread()/fwrite()和read()/write() 有何区别？**
 - 前者是I/O库接口（函数调用），后者是VFS接口（系统调用）
 - 使用用户态缓冲区，减少模式切换次数
- **缓冲模式配置：**
 - 全缓冲（_IOFBF）：缓冲区满了才flush缓冲区
 - 行缓冲（_IOLBF）：遇到换行符就flush缓冲区
 - 无缓冲（_IONBF）：和直接调用read()/write()效果一样

案例：io_uring (Linux 5.1)

- **问题：频繁的读写I/O请求导致高频模式切换开销**
- **提供“提交队列”和“完成队列”两个环形缓冲区**
 - 用户程序和内核共享队列，减少拷贝；允许批量处理多个I/O请求，减少切换
- **支持轮询模式，相比于异步通知机制有更低时延**



2022/4/24

设备管理和网络

今天我们继续来讲设备，以网络作为一个 case study，网络比较重要，并且和存储有比较大的区别。

我们回顾一下三大类设备：字符型设备（流形式，在时间上有顺序）、块设备（磁盘存储）和网络设备。网络设备其实也是属于时间上有顺序。

OS 管理设备有三种方式，我们分成这三类的目的就是要弄清除 OS 在管理设备方面的不同选择的原因。

设备管理主要的三类方式

- **第一类（内核直管）：静态I/O资源分配+向上提供文件接口**
 - 代表设备：Console (CGA+键盘)
 - 特征：中断号等固定，内核直接提供文件接口（无需外部驱动）
- **第二类（设备驱动）：动态I/O资源分配+向上提供文件接口**
 - 代表设备：PCI、USB，如硬盘、网卡等
 - 特征：中断号等动态分配，内核提供驱动框架，设备制造商提供驱动模块
- **第三类（用户态库）：动态I/O资源分配+向上提供内存接口**
 - 代表设备：智能网卡、智能SSD
 - 特征：内核直接将设备暴露给用户态，设备制造商提供用户态库

最早使用的是内核直管，1980s 的时候，设备不是特别多，最重要的就是键盘鼠标显示器，每台电脑在启动的时候都需要把这些设备启动起来，所以 OS 就定义了一些专门的中断向量号，这都是静态写死的，这种模式比较简单，OS 把硬件物理地址直接映射成虚拟地址对上提供文件接口给应用程序访问。

Q：这种模式有什么问题？

A：硬件设备的中断向量号是固定的，一共才十五个，没有办法处理几百个设备的情况。所以静态划分的方式不容易扩展。如果换一个不同的设备，如果我们要在 IBM PC 规范里制定每个设备要使用的物理内存，那很明显赶不上变化。所以一定需要动态资源分配的方式。

为了动态绑定设备，就出现了可插拔的设备，OS 不需要因为硬件的改动而重新编译。硬件厂商制定了 PCI 总线这样的标准，在设计这个规范的时候，就提出了：允许软件遍历 PCI 上有什么设备。有了这样的一个 PCI 和 USB 之后，OS 都是动态通过硬件总线提出的枚举的能力，动态获得硬件的信息，拿到以后再对硬件做相应的驱动。这就是 Linux 里动态插入的 ko 文件，不需要重启。为了不让设备提供商写一大堆硬件驱动，所以出现了驱动的框架。

第三类就是，OS 认为自己的事情做的太多了。一旦把硬件分成三大类，总有硬件不满意，希望进一步定制化功能。OS 依然是动态 I/O 资源分配，但是它不再对上提供文件接口，它直接把最底层的内存接口暴露出来，也就是用户态可以像 OS 一样去操作设备了。这样设备制造商就可以提供一个用户态的程序，而不受 OS 框架的约束。

OS 管理的设备向上提供的接口就是文件接口和内存接口。

OS的设备管理：对上与对下

- **向下对接设备：分配必要的I/O资源**
 - 中断号 + 虚拟地址映射空间
 - 静态绑定：由硬件规范确定
 - 动态分配：动态扫描后再分配
- **对上提供接口：应用使用设备的方式**
 - **文件接口**：通过系统调用
 - open、read、write、close、ioctl等
 - **内存接口**：寄存器直接映射到用户态虚拟内存空间
 - 无需系统调用，应用可直接访问设备

OS 设备管理第一类方式：内核直管

Console 就是我们看到的终端，无非就是显示器和键盘。这两个合在一起对上就是一个设备，提供了从键盘读和写到显示器的操作。

第一类：内核直管

- **以Console为例**
 - 包含设备：显示器 (CGA) + 键盘
- **I/O资源静态绑定**
 - 中断号（即异常向量表）
 - 寄存器所映射的虚拟内存地址
- **文件接口**
 - read：读取键盘输入，通常以行为单位
 - write：向显示器输出字符并显示

我们直接来看代码。当我们写的时候，就会调用 consolewrite，它是从 syscall_write 一路下来的。consolewrite 有一个 inode 参数，其实这就是我们的 console 设备。其中我们调用了 consputc，它最终会调用到 cgaputc。cga 就是显示器，当时的显示器是 80 x 25（1 行 80 字符，显示 25 行），我们是向它的寄存器区域直接写一个字符，这就是对设备的最终控制。

从 syscall 到 consolewrite，再到 cgaputc，没多少代码，不需要很复杂的框架和驱动。

```

273 int
274 consolewrite(struct inode *ip, char *buf, int n)
275 {
276     int i;
277
278     iunlock(ip);
279     acquire(&cons.lock);
280     for(i = 0; i < n; i++)
281         consputc(buf[i] & 0xff);
282     release(&cons.lock);
283     ilock(ip);
284
285     return n;
286 }

```



```

165 void
166 consputc(int c)
167 {
168     if(panicked){
169         cli();
170         for(;;)
171             ;
172     }
173
174     if(c == BACKSPACE){
175         uartputc('\b'); uartputc(' '); uartputc('\b');
176     } else
177         uartputc(c);
178     cgaputc(c);
179 }

```



```

131 static void
132 cgaputc(int c)
133 {
134     int pos;
135
136     // Cursor position: col + 80*row.
137     outb(CRTPORT, 14);
138     pos = inb(CRTPORT+1) <> 8;
139     outb(CRTPORT, 15);
140     pos |= inb(CRTPORT+1);
141
142     if(c == '\n')
143         pos += 80 - pos%80;
144     else if(c == BACKSPACE){
145         if(pos > 0) --pos;
146     } else
147         crt[pos++] = (c&0xff) | 0x0700; // black on white
148
149     if(pos < 0 || pos > 25*80)
150         panic("pos under/overflow");
151
152     if((pos/80) >= 24){ // Scroll up.
153         memmove(crt, crt+80, sizeof(crt[0])*23*80);
154         pos -= 80;
155         memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
156     }
157
158     outb(CRTPORT, 14);
159     outb(CRTPORT+1, pos>>8);
160     outb(CRTPORT, 15);
161     outb(CRTPORT+1, pos);
162     crt[pos] = ' ' | 0x0700;
163 }

```

我们来看看 consoleread 在做什么。从 244 行开始是一个循环，等待键盘输入。每次读一个值就把 r++，拿到 c。

```

235 int
236 consoleread(struct inode *ip, char *dst, int n)
237 {
238     uint target;
239     int c;
240
241     iunlock(ip);
242     target = n;
243     acquire(&cons.lock);
244     while(n > 0){
245         while(input.r == input.w){
246             if(myproc()->killed){
247                 release(&cons.lock);
248                 ilock(ip);
249                 return -1;
250             }
251             sleep(&input.r, &cons.lock);
252         }
253         c = input.buf[input.r++ % INPUT_BUF];
254         if(c == C_D){ // EOF
255             if(n < target){
256                 // Save ^D for next time, to make sure
257                 // caller gets a 0-byte result.
258                 input.r--;
259             }
260             break;
261         }
262         *dst++ = c;
263         --n;
264         if(c == '\n')
265             break;
266     }
267     release(&cons.lock);
268     ilock(ip);
269
270     return target - n;
271 }

```

input.buf 何时更新？

Q: input.buf 什么时候更新？

A: 按键盘的时候更新。

按了键盘以后，产生一个中断，陷入到内核的 exception handler，然后陷入到 trap。一旦发现是键盘的中断，就会调用键盘注册的 kbdintr（中断处理函数）。

注意到 kbdgetc 中有 inb 函数，它就是映射到某一个地址开始的数据，得到 data。data 最终会变成 c 返回，然后放到 input.buf 中去。

```

191 void
192 consoleintr(int (*getc)(void))
193 {
194     int c, doprocnum = 0;
195
196     acquire(cons_conslock);
197     while(c = getc()) == 0{
198         switch(c){
199             case CC_P: // Process listing,
200                 // procname() locks cons.lock indirectly; invoke later
201                 doprocnum++;
202                 break;
203             case CC_U: // kill line.
204                 while(input.e != input.w &&
205                     input.buf[(input.e-1) % INPUT_BUF] != '\n'){
206                     input.e++;
207                     compute(BACKSPACE);
208                 }
209                 break;
210             case CC_H: case '\x07': // Backspace
211                 if(c == '\x07') input.w--;
212                 input.e--;
213                 compute(BACKSPACE);
214             }
215             break;
216         default:
217             if(c == '\r' || c == '\n' || input.e == input.r + INPUT_BUF){
218                 c = (c == '\r') ? '\n' : c;
219                 input.buf[(input.e++ % INPUT_BUF)] = c;
220                 compute(c);
221                 if(c == '\r' || c == '\n' || input.e == input.r + INPUT_BUF){
222                     input.w = input.e;
223                     wakeup(&input.r);
224                 }
225             }
226             break;
227         }
228     }
229     release(cons.lock);
230     if(doprocnum) {
231         procdump(); // now call procdump() w/o. cons.lock held
232     }
233 }

```

```

6 int
7 kbdget(void)
8 {
9     static uint shift;
10    static uchar *charcode[4] = {
11        normalmap, shiftmap, ctmmap, ctlmap
12    };
13    uint st, data, c;
14
15    st = inb(KBDSTAT);
16    if((st & KBD_DIB) == 0)
17        return -1;
18    data = inb(KBDATAP);
19
20    if(data == 0xE0){
21        shift |= E0ESC;
22        return -1;
23    } else if((data & 0x80) == 0x80){
24        /* key released
25        data = (shift & E0ESC ? data : data & 0x7F);
26        shift ^= -(shiftcode[data] | E0ESC);
27        return 0;
28    } else if(shift & E0ESC){
29        // Last character was an E0 escape; or with 0x80
30        data |= 0x80;
31        shift ^= -E0ESC;
32    }
33
34    shift |= shiftcode[data];
35    shift ^= togglecode[data];
36    c = charcode[shift & (CTL | SHIFT)][data];
37    if(shift & CAPSLOCK){
38        if('a' <= c && c <= 'z')
39            c += 'A' - 'a';
40        else if('A' <= c && c <= 'Z')
41            c += 'a' - 'A';
42    }
43    return c;
44 }

```

怎么到读写的呢？读是通过 `syscall` 下来的，在 `fileread` 里，我们调用 `readi`。在 `readi` 里有一个 `devsw.read()`，也就是 `consoleinit` 的时候会初始化它的主设备类型。下图的代码其实就是我们之前讲到的 VFS。

```

69 int
70 sys_read(void)
71 {
72     struct file *f;
73     int n;
74     char *p;
75
76     if(argfd(0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
77         return -1;
78     return fileread(f, p, n);
79 }

```

```

96 int
97 fileread(struct file *f, char *addr, int n)
98 {
99     int r;
100
101    if(f->readable == 0)
102        return -1;
103    if(f->type == FD_PIPE)
104        return piperead(f->pipe, addr, n);
105    if(f->type == FD_INODE){
106        unlock(f->ip);
107        if(r = read(f->ip, addr, f->off, n)) > 0)
108            f->off += r;
109        iunlock(f->ip);
110    }
111    return r;
112    panic("fileread");
113 }

```

```

288 void
289 consoleinit(void)
290 {
291     initlock(&cons.lock, "console");
292
293     devsw[CONSOLE].write = consolewrite;
294     devsw[CONSOLE].read = consulread;
295     conslocking = 1;
296
297     ioapicenable(IRQ_KBD, 0);
298 }
299

```

```

452 int
453 readi(struct inode *ip, char *dst, uint off, uint n)
454 {
455     uint tot, m;
456     struct buf *bp;
457
458     if(ip->type == T_DEV){
459         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
460             return -1;
461         return devsw[ip->major].read(ip, dst, n);
462     }
463
464     if(off > ip->size || off + n < off)
465         return -1;
466     if(off + n > ip->size)
467         n = ip->size - off;
468
469     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
470         bp = bread(ip->dev, bmap(ip, off/BSIZE));
471         m = min(n - tot, BSIZE - off%BSIZE);
472         memmove(dst, bp->data + off%BSIZE, m);
473         brelse(bp);
474     }
475     return n;
476 }

```

9

OS 设备管理第二类方式：设备驱动

OS 会提供一系列的接口规范给驱动实现。OS 提供了 VFS（提供权限检查逻辑）、设备驱动框架（不同的设备要有不同的框架）、模块动态插入框架（Linux 要支持 `ko` 文件的动态加载，主要是和加载器相关）和中断处理框架（早期一个中断过来，处理完就返回了，这可能导致中断的处理时间不确定，整个系统就 `block` 住了，所以提出了两阶段的处理方法）。

所以内核为驱动提供的辅助功能是非常多的，这个复杂度有没有必要是存在争议的。有人认为 OS 的责任太大，每次有人要修改就必须帮它改，OS 应该尽可能精简把选择权交给用户。

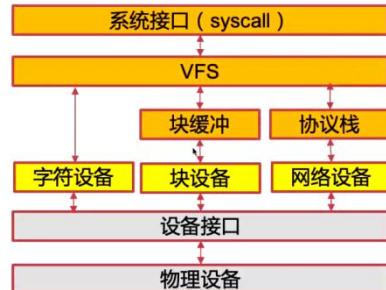
第二类：设备驱动

- 操作系统的设备驱动框架
 - 提供标准化的数据结构和接口
 - 将驱动开发简化为对数据结构的填充和实现
 - 方便操作系统统一组织和管理设备
- 操作系统为驱动提供的辅助功能
 - VFS (对上提供文件接口，可复用权限检查等逻辑)
 - 与设备类型相关的框架 (如：块设备层、网络协议栈等)
 - 模块动态插入框架 (如：Linux的module插入)
 - 中断处理框架 (如：Linux的 top half + bottom half)

回顾：Linux的设备分类

• Linux设备分类

- 字符设备
- 块设备
- 网络设备



Linux 也有用户态的驱动框架，更加灵活可靠，驱动 crash 了也不算严重，但是缺点就是性能开销会严重。

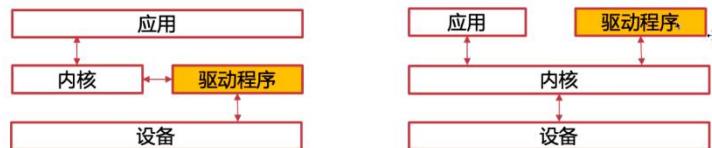
宏内核vs微内核的驱动

• 宏内核

- 驱动在内核态
- 优势：性能更好
- 劣势：容错性差

• 微内核

- 驱动在用户态
- 优势：可靠性好
- 劣势：性能开销 (IPC)



OS 设备管理第三类方式：用户态库

在某些场合下，大家更重视可靠性。所以就出现了第三类，这种情况下，OS 只需要做完映射。应用程序独占设备之后可以对别的应用提供服务，相当于做一次 IPC。通常是使用轮询访问设备，因为中断要进内核，速度会变慢。典型的例子就是 DPDK 和 SPDK，都是为

了性能所提出的方案。

第三类：用户态库

- **操作系统内核负责：**

- 将设备寄存器映射到用户可访问的虚拟地址空间
- 通常只允许某一个应用访问该设备，防止发生冲突

- **用户态库负责：**

- 100%控制硬件设备，并向上或对外提供服务
- 通过轮询访问设备（问：为何不用中断？）
 - 通常需要独占一个CPU
- 典型例子：Intel DPDK（网卡）、SPDK（存储）
- 通常用于高性能场景（问：为什么性能高？）

A: 性能高的原因：轮询，没有中断。

这三大类其实就是从简单（设备很简单）到复杂（设备变复杂），再到简单的过程（OS自己想变得精简、设备还是很复杂）。第二类还是我们现在的主流，第三类还没有特别普及。

Linux 的上下半部

我们以 Linux 的中断处理框架来介绍 OS 的中断处理流程。OS 在进入 Kernel 的时候，会屏蔽掉中断。关掉的过程中，如果我们做复杂的逻辑，就会导致系统失去响应。比如 page fault 的流程就很长，我们又不能限制说不能发生 page fault。

上半部就是把当前必须要做的事情做完，比如把硬件中输入的字符读走，否则就可能被覆盖。软件处理的部分是可以推迟的，就放到下半部中。

Linux上下半部

- **面临问题**

- 中断处理过程中若运行复杂逻辑，会导致系统失去响应更久
- 中断处理时不能调用会导致系统block的函数

- **将中断处理分为两部分**

- 上半部：尽量快，提高对外设的响应能力
- 下半部：将中断的处理推迟完成

Top Half : 马上做

- **最小化公共例程 :**
 - 保存寄存器、屏蔽中断
 - 恢复寄存器，返回现场
- **Top half要做的事情 :**
 - 将请求放入队列（或设置flag），将其他处理推迟到bottom half
 - 现代处理器中，多个I/O设备共享一个IRQ和中断向量
 - 多个ISR (interrupt service routines)可以绑定同一向量上
 - 调用每个设备对应的IRQ的ISR

Q: 为什么要共享一个 IRQ?

A: 因为硬件中断向量号资源比较少。

Top Half 主要是和硬件相关的；Bottom Half 主要和软件相关，有四套机制。这些机制在运行的时候，中断都是打开的，也就是在运行的过程中还是可以做 Top Half。

Bottom Half : 延迟去做

- **提供可以推迟完成任务的机制**
 - softirqs
 - tasklets (建立在softirqs之上)
 - 工作队列
 - 内核线程
- **这些机制都可以被中断**

内核线程 (Kernel Threads)

我们来举个例子，就是内核线程。如果我们能把 Bottom Half 的逻辑跑在内核线程里，说明这些逻辑是可以被调度的，是可以调用 sleep 的。所以内核线程是最灵活的一种方式，它复用了内核的调度逻辑。内核线程就是跑在内核里的线程，它的页表没有用户态的映射，所以它对内核都是一样的。

总结

特点	ISR	SoftIRQ	Tasklet	WorkQueue	KThread
禁用所有中断 ?	Briefly	No	No	No	No
禁用相同优先级的中断 ?	Yes	Yes	No	No	No
比常规任务优先级更高 ?	Yes	Yes*	Yes*	No	No
在相同处理器上运行 ?	N/A	Yes	Yes	Yes	Maybe
允许在同一CPU上有多个实例同时运行 ?	No	No	No	Yes	Yes
允许在多个CPU上运行同时多个相同实例 ?	Yes	Yes	No	Yes	Yes
完整的模式切换 ?	No	No	No	Yes	Yes
能否睡眠 (拥有自己的内核栈) ?	No	No	No	Yes	Yes
能否访问用户空间 ?	No	No	No	No	No

* 可以由 ksoftirqd 调度

操作系统的网络

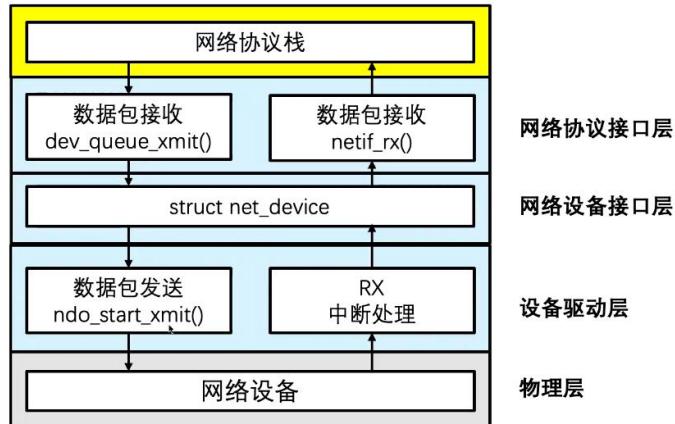
网络协议栈传统是 OS 提供的。

网络协议栈的分层模型



整个 Linux 的网络驱动模型如下，协议栈之下就是发包和收包。网络设备提供商只需要提供设备驱动层的两个函数。

Linux网络驱动模型

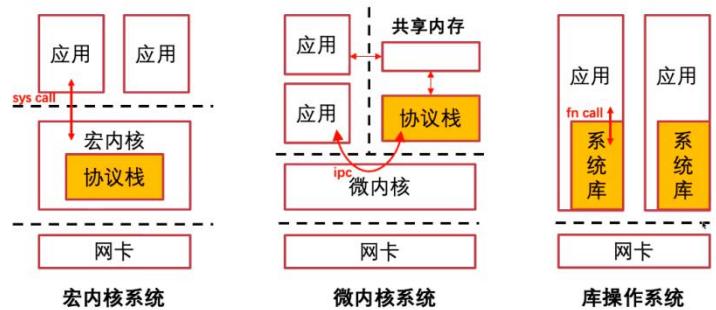


中断合并

- **Interrupt coalescing**
 - 当外设中断次数累计到一定阈值时，再向CPU发送中断
 - 或者到某个timeout，向CPU发送中断
- 可避免“中断风暴”
- 更少的中断次数意味着：
 - 更高的吞吐量
 - 但也增加了中断响应的延迟

前两个属于第二类，第三个属于第三类。网卡如果不能被应用独占，可以虚拟化成多块网卡，再给应用独占。

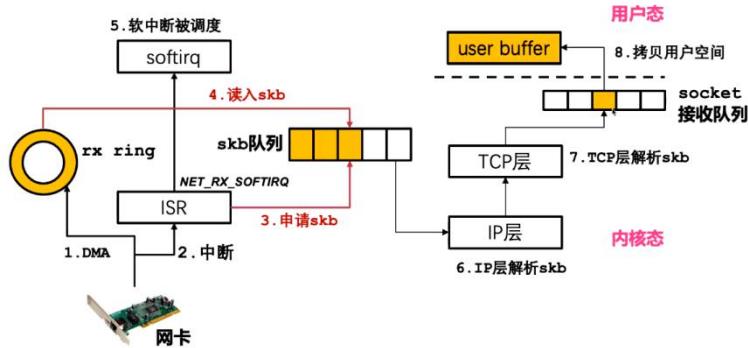
架构对比



Linux 的收包过程

受到包以后，网卡先用 DMA 复制到 rx ring 中，然后再发中断，OS 就会把 ring 读到 skb 队列中。读完之后，就可以做 bottom half 了。内核做层层解析，最后把数据解析完扔给 socket，这属于用户态接口。这里面包含了三次 copy。

Linux收包过程



收包：数据链路层（1）

- 网卡收到数据包（以太网帧）：
 - DMA 将数据帧传送至内核内存中的 rx_ring
 - 网卡中断被触发
- CPU 收到网卡中断，调用网卡 ISR（上半部）：
 - 分配 sk_buff (skb) 数据结构，负责管理 rx_ring 中的数据包
 - 将 skb 入队 (input_pkt_queue)
- 上半部发出一个软中断 (NET_RX_SOFTIRQ)：
 - 通知内核处理 skb 包

收包：网络层

- IP 层的入口函数 ip_rcv()：
 - 检查是否为 IP 包
 - 检查 IP 版本号
 - 对完整性 (checksum) 和长度进行检查
- ip_rcv() 结束调用 ip_router_input()，进行路由处理
 - 查找路由
 - 决定该数据包（报文）是发到本机，还是被转发，或是被丢弃

收包：数据链路层（2）

- 进入软中断处理流程（下半部）：
 - 把 input_pkt_queue 的 skb 移动到 process_queue 处理队列中
 - 根据报文类型（ARP 或是 IP），把报文递交给对应协议进行处理
 - 调用网络层协议的 handler 处理 skb 包
- 移动 skb：操作指针
- 如果接收队列已满 (input_pkt_queue)：
 - 丢弃后续数据：可能发生活锁！

收包：传输层

- 传输层的处理入口 tcp_v4_rcv()
 - 对 TCP header 进行检查
- 调用 _tcp_v4_lookup，查找该数据包对应的 open socket
 - 如果找不到，该数据包被丢弃
 - 否则检查 socket 和 connection 的状态
- socket 和 connection 正常
 - 调用 tcp_pqueue() 使 TCP 载荷从内核进入用户空间，放进 socket 接收队列

收包：应用层

- socket 被唤醒，调用 system call
 - 最终调用 tcp_recvmsg()，从 socket 接收队列中获取数据
- 用户态调用 read 或者 recvfrom
 - 转化为 sys_recvfrom 调用
 - 对 TCP 来说，调用 tcp_recvmsg()：该函数从 socket buffer 中拷贝数据到 user buffer

网络包的处理和一般的不太一样，我们要经常要加头部，所以我们希望避免多次的数据拷贝。

网络包的管理

- 要求高效地处理分层

- 发包时需要不断添加新的头部，收包则相反

- 避免连续存放网络包

- 移动过程中数据拷贝会有很大开销

- Linux数据结构：`sk_buff`（简称skb）

- 让分层的处理变得高效：零拷贝

- 快速申请和释放内存：防止内存碎片

sk_buff

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff     *next;
    struct sk_buff     *prev;

    // 真正指向的数据buffer
    struct sock *sk;

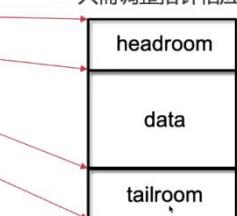
    // 缓冲区的头部
    unsigned char *head;
    // 实际数据的头部
    unsigned char *data;
    // 实际数据的尾部
    unsigned char *tail;
    // 缓冲区的尾部
    unsigned char *end;
};
```

- **sk_buff本身不存储报文**

- 通过指针指向真正的报文内存空间

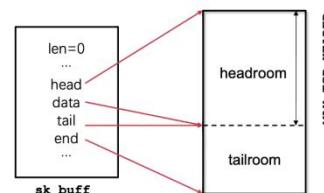
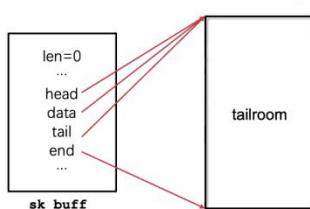
- **在各层传递时**

- 只需调整指针相应位置即可



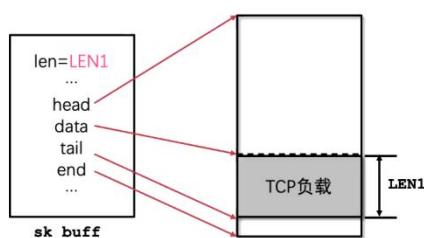
例子：发送数据包

- Step1: TCP层发数据时，首先用`alloc_skb`申请缓冲区。 Step2: TCP用`skb_reserve`来保留足量空间存储所有头部



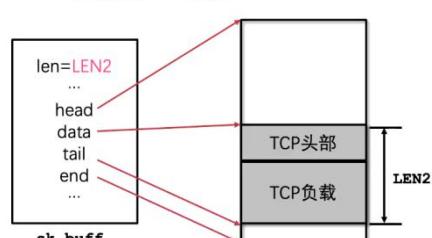
例子：发送数据包

- Step3: TCP层填入TCP负载（应用层数据）



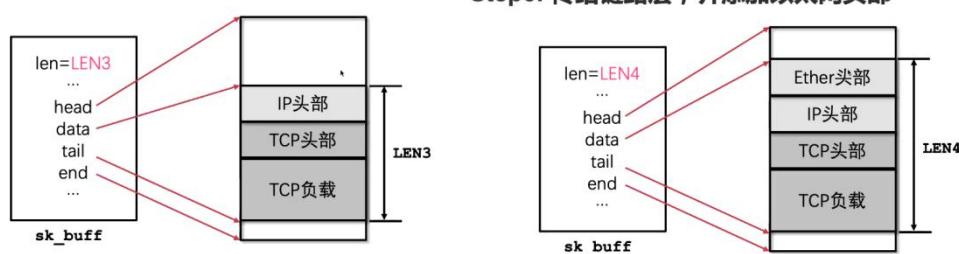
例子：发送数据包

- Step4: TCP层填入TCP头部



例子：发送数据包

- Step5: 传给IP层，并添加IP头部



例子：发送数据包

- Step6: 传给链路层，并添加以太网头部

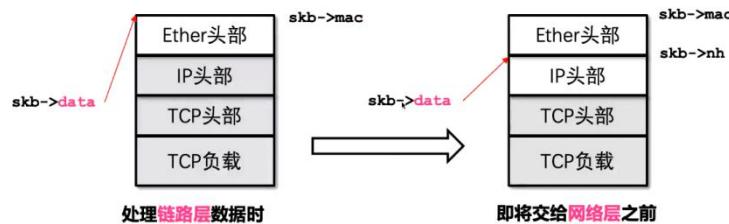
当前层在处理 skb 的时候，会将下一层的 skb 初始化好。

skb的协议栈头部

- 当前层处理skb时：

- 同时负责将下一层头部指针初始化好，并移动data指针

```
struct sk_buff {
    ...
    union { ... } h; // <--- 传输层头部
    union { ... } nh; // <--- 网络层头部
    union { ... } mac; // <--- 数据链路层头部
    ...
};
```



skb control buffer

- **char cb[40];**
- **用于每层维护私有的信息**

```
struct tcp_skb_cb {
    __u32    seq;           /* Starting sequence number */
    __u32    end_seq;        /* SEQ + FIN + SYN + datalen */
    ...
    __u8     tcp_flags;      /* TCP header flags. (tcp[13]) */
    __u8     sacked;         /* State flags for SACK. */
    __u8     txstamp_ack:1,   /* Record TX timestamp for ack? */
    __u32    ack_seq;        /* Sequence number ACK'd */
    ...
};
```

skb的组织

- 用双向链表对sk_buff进行管理：

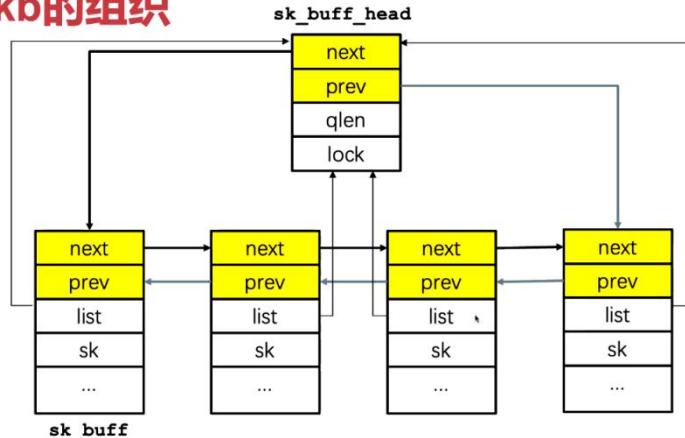
```
struct sk_buff_head {  
    /* These two members must be first. */  
    struct sk_buff *next;  
    struct sk_buff *prev;  
    __u32 qlen; // 链表的元素长度  
    spinlock_t lock; // 添加元素时必须持有该锁  
};
```

- Linux NAPI的批处理 *

- 让网卡中断处理的下半部softirq积累足量的skb
- NAPI周期性轮询，并一次性处理完所有skb (batching)
- netif_receive_skb_list()*

* Batch processing of network packets, LWN, <https://lwn.net/Articles/763056/>

skb的组织



skb小结

- sk_buff**

- 用于 Linux 网络子系统中各层之间的数据传递
- 不同协议层的处理函数通过控制sk_buff结构来共享网络报文

- 收包：**

- 网卡收到数据包后，将以太网帧数据转换为sk_buff数据结构
- 各层剥去相应的协议头部，直至交给用户

- 发包：**

- 网络模块必须建立一个包含待传输的数据包的sk_buff
- 各层在sk_buff 中添加对应的协议头部直至交给网卡发送

用户态协议栈：DPDK

Linux网络协议栈的问题

- 网络设备的速度越来越高
- CPU朝着多核方向发展
- 内核协议栈逐渐成为高性能网络的**性能瓶颈**

Linux网络协议栈的问题

- 中断处理
 - 大量网络包到来→频繁的硬件中断请求
 - 中断频繁打断较低优先级的软中断或者系统调用的执行过程→网络处理缓慢
- 上下文切换
 - 线程间的调度产生频繁上下文切换开销
 - 锁竞争的开销严重：cacheline sharing
- 内存拷贝
 - 数据从网卡DMA传到内核缓冲区，再从内核空间拷贝到用户空间
 - 占据Linux 内核协议栈数据包整个处理流程的 57.1%

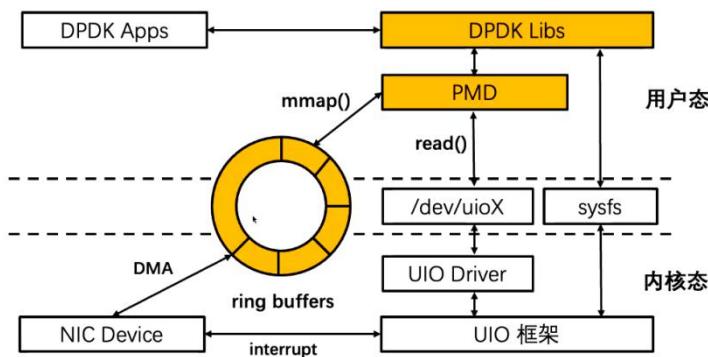
Linux网络协议栈的问题

- 内存管理
 - 内存页大小为4K，对TLB要求更高
- 局部性失效
 - 数据包处理可能跨多核：导致CPU 缓存失效，空间局部性差
 - NUMA 架构：存在跨 NUMA内存访问，性能影响很大

Intel DPDK

- Data Plane Development Kit
- 绕过Linux内核协议栈 (bypass kernel)
 - 直接在用户空间实现数据包的收发和处理
- Linux User I/O
 - 在用户态访问 MMIO (和设备交互) 与DMA ring buffers
- 轮询模式驱动 : Poll Mode Driver (PMD)

DPDK架构图



DPDK特性

- 抛弃中断，使用轮询模式
- 使用大页 (2MB)，减少TLB miss
- 控制平面与数据平面相分离：
 - 内核态负责“控制平面”：内核仅负责控制指令的处理
 - 用户态负责“数据平面”：将数据包处理、内存管理、处理器调度等任务转移到用户空间去完成，有效避免了繁重的模式切换

DPDK特性 (2)

- **用多核编程代替多线程技术**
 - 绑核：设置 CPU 亲和性，减少彼此间调度切换
 - 无锁环形队列：解决资源竞争问题
- **CPU 核尽量使用所在 NUMA 节点的内存**
 - 避免跨NUMA内存访问

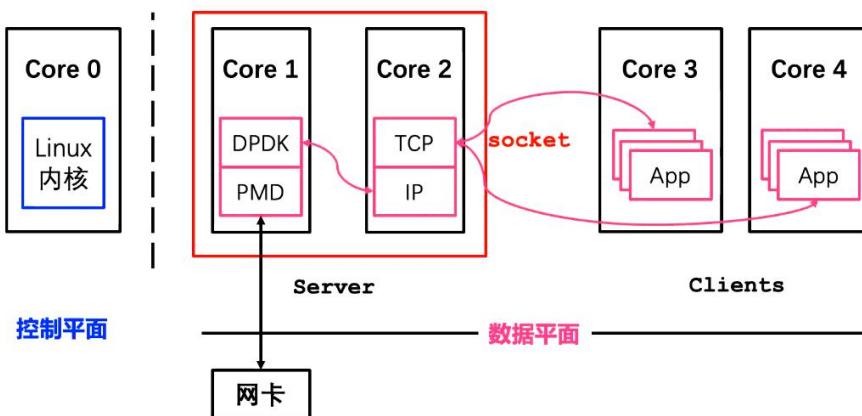
我们可以把原来宏内核做的事情拆散到用户态去做这件事情。

DPDK的使用

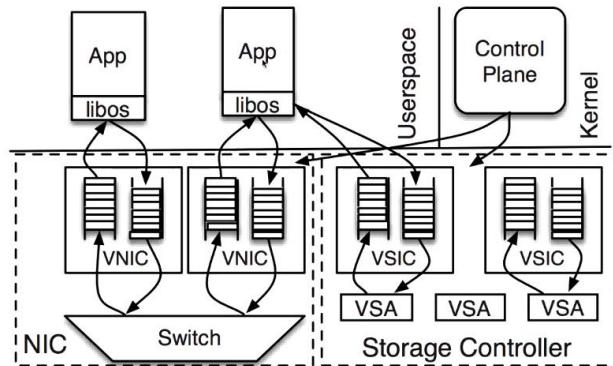
- **DPDK本身只是2层协议，不提供socket接口**
 - 适合于软路由场景
- **如果要用于应用程序，提供socket抽象**
 - **类微内核方案**：将 DPDK+TCP/IP协议栈 作为一个Server，为每个应用单独维护 socket fd 和进程间的对应关系
 - **LibOS方案**：需要逐个应用添加协议栈支持

如果大家都要用，还不如放在一个 Core 上。

类微内核方案

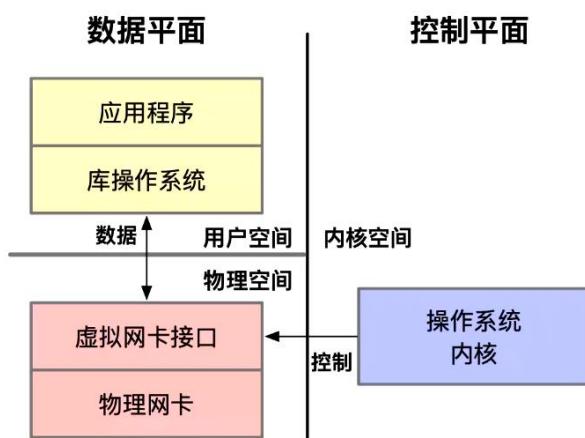


LibOS方案



Arrakis: The Operating System is the Control Plane, OSDI 2014

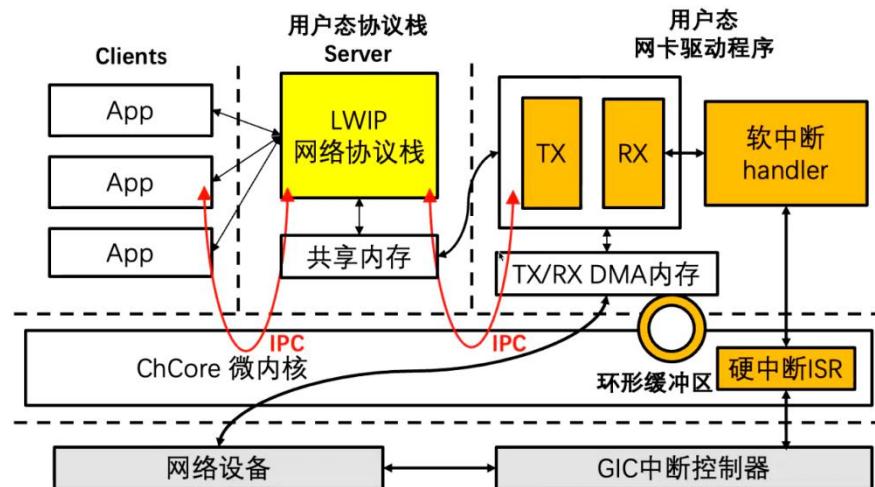
控制平面与数据平面分离



不同架构对比

LWIP 是开源的 DPDK 实现。

微内核网络架构图



不同系统架构下网络设计对比

- **宏内核系统网络模块：**

- 控制平面和数据平面都经过内核
- 驱动和协议栈处在同一地址空间，没有模式切换
- 驱动程序的安全问题会波及协议栈

- **微内核系统网络模块：**

- 用户态驱动+协议栈，安全性好
- 只有一个协议栈，运维成本低
- 控制平面通信需要借助IPC完成，有一定开销

- **库操作系统 (LibOS) 网络模块：**

- 用户态或non-root模式下协议栈，鲁棒性好
- 每个实例都有自己的协议栈
- 一旦需要更新，可维护性成本高
- 多实例polling的情况下会导致浪费CPU*

2022/4/26

今天我们开始讲虚拟化，它是系统领域一个非常重要的 topic，它不仅可以让我们了解虚拟机是怎么运行的。还可以通过虚拟化的视角审视 OS 的本质是什么。从应用程序自己来看，用户态的 ISA 只能在自己的访问范围内去控制硬件。一旦牵涉到和底层相关的打印数据、网络等，那就需要 OS 提供的系统调用了。

既然之前是应用程序在跑，OS 需要为应用程序提供运行环境。现在我们把应用程序换成完整的一台机器（上面跑了很多应用程序），它所看到的硬件的运行环境是什么。我们给它包一层使它不知道别人也在使用这个机器。

虚拟化技术在半个世纪以前就有了，当时大型机非常集中，大家要通过很慢的网络连到大型机里。我们开发 OS 为了再搞一台机器放着就很麻烦，我们希望让开发 OS 的人能够复用这一台计算机。让计算机能够既可以开发 OS，也运行别的应用。PC 时代，计算机资源变得分布，虚拟化技术没有太大的需求了。云产生了以后，大家觉得不需要那么多的资源，租云服务器便宜得多。

计算设备集中与分散的变化

- **大型机时代**
 - 集中式计算资源，所有用户通过网络连接大型机，共享计算资源
 - 20世纪70年代，虚拟化技术已经兴起 (!)
- **PC时代**
 - 分布式计算资源，每个PC用户独占计算资源
 - 20世纪90年代，虚拟化技术沉寂
- **云时代**
 - 集中式计算资源，所有人通过网络连接，共享计算资源
 - 21世纪，虚拟化技术再次兴起

现代公司的IT部署方式：云

- **云服务器代替物理服务器**
 - 云服务器配置与物理服务器一致
 - 所有云服务器维护由服务商提供



云计算为云租户带来的优势

- 按需租赁、无需机房租赁费
- 无需雇佣物理服务器管理人员
- 可以快速低成本地升级服务器
- ...

对用户来说，看到的是一个独立的服务器，但是对云厂商来说，它希望在物理服务器上形成一种流动性，也就是用户负载很轻的时候，希望把多人的虚拟服务器整合在一台服务器上，从而节省电费。引入虚拟化层之后，就可以更顺畅的流动。

系统虚拟化是云计算的核心支撑技术

- **新引入的一个软件层**
 - 上层是操作系统（虚拟机）
 - 底层硬件与上层软件解耦
 - 上层软件可在不同硬件之间切换



虚拟化带来的优势

• "Any problem in computer science can be solved by another level of indirection" --- David Wheeler

- 1、服务器整合：提高资源利用率
- 2、方便程序开发
- 3、简化服务器管理
- ...

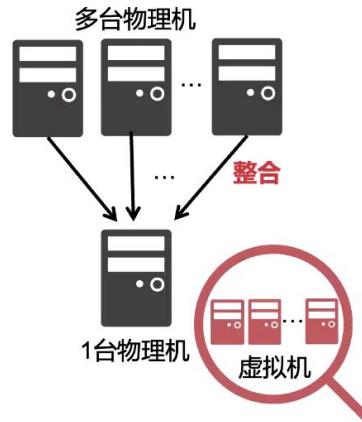
对 AMS 和阿里云，利用率还是可以高一点。其他数据中心的利用率都很低，存储需求是很高的，比如监控视频的数据量很大，摄像头每时每刻产生着大量的数据。所以数据中心主要在做存储，但是 CPU 利用率上不去。

Q: Webserver 和数据库为什么不整合在一个物理机上呢？为什么要用到虚拟机呢？

A: 底层的库的版本依赖会很复杂，所以需要虚拟机。不需要关心上面的库/软件栈/OS。

虚拟化优势-1：服务器整合

- 单个物理机资源利用率低
 - CPU 利用率通常仅 <20%
- 利用系统虚拟化进行资源整合
 - 一台物理机同时运行多台虚拟机
- 显著提升物理机资源利用率
- 显著降低云服务提供商的成本



华为有手机工厂，一个应用可以在几千个手机上运行，有专门的触摸设备去测试行为是不是一致。

虚拟化优势-2：方便程序开发

- 调试操作系统
 - 单步调试操作系统
 - 查看当前虚拟硬件的状态
 - 寄存器中的值是否正确
 - 内存映射是否正确
 - 随时修改虚拟硬件的状态
- 测试应用程序的兼容性
 - 可以在一台物理机上同时运行在不同的操作系统
 - 测试应用程序在不同操作系统上的兼容性

比如公司现在只需要给大家配置一个虚拟机而不是笔记本了，这显然比配笔记本更方便。

虚拟化优势-3：简化服务器管理

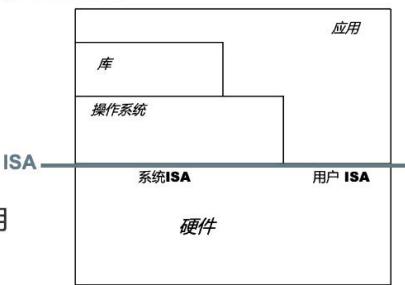
- 通过软件接口管理虚拟机
 - 创建、开机、关机、销毁
 - 方便高效
- 虚拟机热迁移
 - 方便物理机器的维护和升级

什么是系统虚拟化？

首先就是 ISA（指令集架构），OS 既用系统 ISA，又用用户 ISA。而应用程序只用用户 ISA。ISA 是非常底层的接口。

操作系统中的接口层次: ISA

- ISA层
 - Instruction Set Architecture
 - 区分硬件和软件
 - 用户ISA
 - 用户态和内核态程序都可以使用
 - `mov x0, sp`
 - `add x0, x0, #1`
 - 系统ISA
 - 只有内核态程序可以使用
 - `msr vbar_el1, x0`

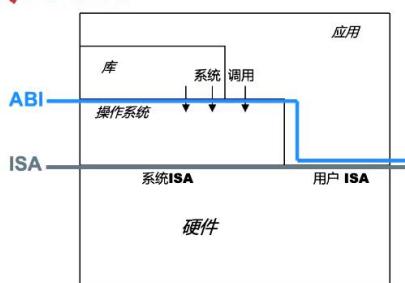


ISA – instruction set architecture

再往上一层就是 ABI，它会包含 ISA 之上的东西。比如 `syscall` 就是 ABI 的一部分，用户的 ISA 也是 ABI 的一部分。包含了 calling convention 的情况下，一个应用可以在二进制文件不修改的情况下在两个相同的 ABI 机器（比如 Win10 和 Win11）上运行。

操作系统中的接口层次: ABI

- ABI
 - Application Binary Interface
 - 提供操作系统服务或硬件功能
 - 包含用户ISA和系统调用

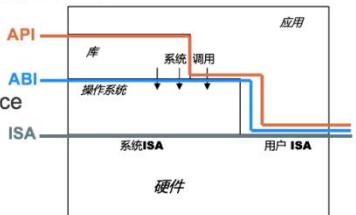


ABI – application binary interface

ISA – instruction set architecture

操作系统中的接口层次: API

- **API**
 - Application Programming Interface
 - 不同用户态库提供的接口
 - 包含库的接口和用户ISA
 - UNIX环境中的**clib**:
 - 支持UNIX/C编程语言



API – application programming interface

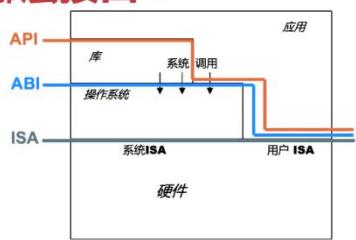
ABI – application binary interface

ISA – instruction set architecture

如果我们要实现虚拟化，到底是在哪一层去做呢？

思考：这些程序用了哪层接口？

- **Hello world**
- **Web game**
- **Dota**
- **Office 2016**
- **Windows 10**
- **Java applications**
- **ChCore**



API – application programming interface

ABI – application binary interface

ISA – instruction set architecture

1. 如果是 C 的 Hello world, Linux 上编译的 hello world 是不能直接在 Windows 上跑的, 所以是 ABI。如果是 Python 的 Hello world, 不同 OS 都可以跑, 此时就是 API 层次。
2. 对 Web game, 只需要考虑有没有使用浏览器, 也是 API 层次。
3. Dota 这种 native 游戏, 只能到 ABI 层。但是 Linux 直接可以模仿 Windows 的 ABI 接口实现。
4. office 也是 ABI 层。
5. Win10 和 ChCore 是 ISA 层次。
6. Java 本身也用到了虚拟化技术, 它底层就是虚拟机。

其实, 定义虚拟机取决于我们实现了哪个层次的接口。如果我们实现的是 ISA 接口, 那就叫做系统虚拟化。

如何定义虚拟机？

- **从操作系统角度看"Machine"**

- ISA 提供了操作系统和Machine之间的界限

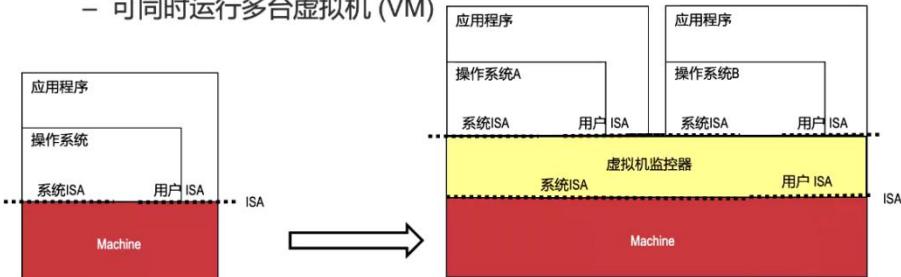


系统虚拟化对上提供 ISA 的多路复用。让本来只能一个人用的系统 ISA 变成多个实体使用的系统 ISA，所以 OS A 和 OS B 就可以在黄色的虚拟机监控器之上运行起来。

虚拟机和虚拟机监控器

- **虚拟机监控器 (VMM/Hypervisor)**

- 向上层虚拟机暴露其所需要的ISA
- 可同时运行多台虚拟机 (VM)



虚拟化系统的定义如下：

系统虚拟化的标准

- **Popek & Goldberg, 1974 “Formal Requirements for Virtualizable Third Generation Architectures”**

- **高效系统虚拟化的三个特性**

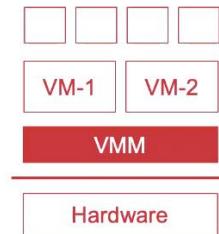
- 为虚拟机内程序提供与该程序原先执行的硬件完全一样的接口
- 虚拟机只比在无虚拟化的情况下性能略差一点
- 虚拟机监控器控制所有物理资源

如果有一个虚拟机控制了部分硬件资源，可能导致其他虚拟机受到影响，这是从安全考虑。

Type-1 虚拟机监控器

Type-1 是根据理论直接导出的，它直接在硬件上运行一层 VMM，在 VMM 之上运行 VM，它看起来充当了 OS 的角色，要实现管理所有的物理资源。VMM 要管理所有的硬件，所以必须要有硬件的驱动。

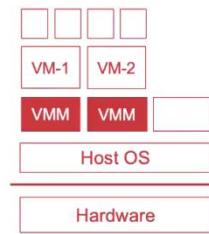
- **VMM直接运行在硬件之上**
 - 充当操作系统的角色
 - 直接管理所有物理资源
 - 实现调度、内存管理、驱动等功能
- **性能损失较少**
- **例如Xen, VMware ESX Server**



Type-2 虚拟机监控器

它是基于 Host OS 的，也就是我们机器上已经安装了 Linux 或者 Windows 了。我们基于 Host OS 再去安装一个 VMM。每个 VMM 之上只运行一个 VM。大家最常见的就是 QEMU，每个 QEMU 可以运行一个虚拟机。

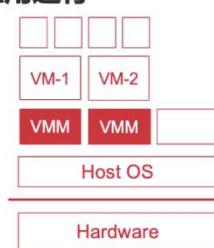
- **VMM依托于主机操作系统**
 - 主机操作系统管理物理资源
 - 虚拟机监控器以进程/内核模块的形态运行
 - 易于实现和安装
 - 例如：QEMU/KVM
- **思考：**
 - Type-2类型有什么优势？



好处就是易于实现、易于安装、可以复用 OS 的功能，OS 来做调度和内存分配，VMM 只需要 malloc 内存就行了，不需要做什么物理内存管理。

Type-2的优势

- **在已有的操作系统之上将虚拟机当做应用运行**
- **复用主机操作系统的大部分功能**
 - 文件系统
 - 驱动程序
 - 处理器调度
 - 物理内存管理



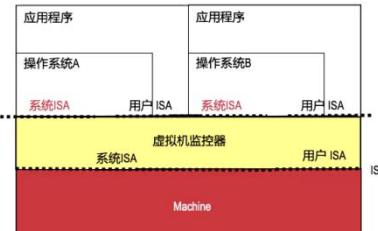
缺点就是性能可能没有那么好，因为 Host OS 是通用的，并不是为虚拟化专门设计的。

如何实现系统虚拟化？

系统 ISA 里修改的状态是整台机器都可以看到的，比如我们打开 CPU 的中断，那么这个全局的中断必然就打开了。ttbr 是页表的寄存器，一旦改了就是改了，如果两个 OS 同时修改，可能就会冲突。如果这个冲突解决了，那么我们就可以让虚拟机同时运行。内存可以通过映射的方式让不同的内存划分到不同的虚拟机，硬盘可以抽象成文件，可以通过文件系统让多个虚拟机去用。但是 CPU 存在系统状态的唯一性，所以导致多个 OS 不能直接去操作。用户 ISA 也是不会冲突的，因为应用程序也是多个一起使用用户 ISA 的，所以多个虚拟机也可以使用同样的办法解决。

系统ISA：操作系统运行环境

- **读写敏感寄存器**
 - sctrl_el1、ttbr0_el1/ttbr1_el1...
- **控制处理器行为**
 - 例如: WFI (陷入低功耗状态)
- **控制虚拟/物理内存**
 - 打开、配置、安装页表
- **控制外设**
 - DMA、中断

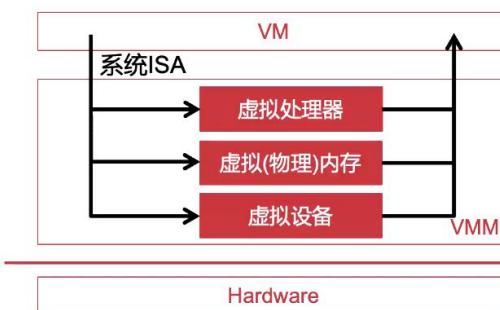


Trap & Emulate

以前系统 ISA 默认一个人用，现在要多人用。我们的解决方案就是 Trap & Emulate。第一步是捕捉到这个调用系统 ISA 的行为，并且产生 Trap 去 VMM，在 VMM 里用一串指令用软件模拟实现系统 ISA 的虚拟化的效果。效果可以分为控制虚拟 CPU（开关中断）/内存（改 TTBR）/设备（访问硬件设备寄存器、让硬盘格式化等）的行为。这三类行为都需要通过软件的方式解决掉。

系统虚拟化的流程：Trap & Emulate

- **第一步 (Trap)**
 - 捕捉所有系统ISA并陷入
- **第二步 (Emulate)**
 - 由具体指令实现相应虚拟化
 - 控制虚拟处理器行为
 - 控制虚拟内存行为
 - 控制虚拟设备行为
- **第三步**
 - 回到虚拟机继续执行



比如我们要关中断，我们并不是把 CPU 上的真正的中断关掉，而是把虚拟机上的虚拟中断关掉，可能只是修改了虚拟机里的一个 bit。比如有一个虚拟设备要给我们虚拟机发中断，那么可能就会跳到虚拟机中断的处理器，如果此时虚拟机的虚拟中断 bit 是 0 的话，那

么我们就忽略掉虚拟中断。这个其实都是由 VMM 来检查的。Trap 得知道 VM 的 OS 确实执行了一条系统 ISA，emulate 就得知道 VMM 具体应该怎么去模拟实现相关的 ISA。

这个过程是 OS 很像，application 也是通过 trap 的方式到内核做系统调用、异常处理等。VM 和 VMM 的关系也是这样的。

系统虚拟化技术主要分为三类：

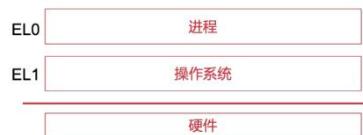
系统虚拟化技术

- **处理器虚拟化**
 - 捕捉系统ISA
 - 控制虚拟处理器的行为
- **内存虚拟化**
 - 提供“假”物理内存的抽象
- **设备虚拟化**
 - 提供虚拟的I/O设备

CPU 虚拟化

回顾：ARM的特权级

- **EL0: 用户态进程**
- **EL1: 操作系统内核**



OS 启动的时候先运行虚拟机监控器。如果要启动一个虚拟机，我们先启动一个进程，再加载虚拟机的镜像。虚拟机监控器加载镜像到内存之后，就跳转到镜像的第一条代码去执行，同时把状态从 EL1 切换到 EL0。

处理器虚拟化：一种直接的实现方法

- **把虚拟机当做应用程序**
 - 将虚拟机监控器运行在EL1
 - 将客户操作系统和其上的进程都运行在EL0
 - 当操作系统执行系统ISA指令时下陷

- 写入TTBR0_EL1
- 执行WFI指令
- ...



如果运行的某一行代码是系统 ISA，就会触发 trap。CPU 就发现 EL0 的用户程序居然想

调用系统 ISA，就会触发一个权限异常的 trap，到 EL1 之后，虚拟机监控器知道了 VM 的 OS 想修改 ttbr 来切换进程。它把 0x20001000 放到 ttbr0_el1 里，其实就是虚拟机监控器把 VM 的地址设置了一套新的内存对应关系里去。这里修改的就是 VMM 控制的页表。

Trap & Emulate

- Trap: 在用户态EL0执行特权指令将陷入EL1的VMM中
- Emulate : 这些指令的功能都由VMM内的函数实现



如果所有系统 ISA 都可以通过 trap & emulate 解决，那么虚拟化问题也解决了。但是 ARM 不是严格的可虚拟化架构。我们希望敏感指令都触发 trap，但是在 ARM 里，不是所有的敏感指令都是特权指令。

ARM不是严格的可虚拟化架构

- 敏感指令
 - 读写特殊寄存器或更改处理器状态
 - 读写敏感内存：例如访问未映射内存、写入只读内存
 - I/O指令
- 特权指令
 - 在用户态执行会触发异常，并陷入内核态

我们在用户态执行这个指令就会变成 NOP，导致 trap 不了。

ARM不是严格的可虚拟化架构

- 在ARM中：不是所有敏感指令都属于特权指令
- 例子: CPSID/CPSIE指令
 - CPSID和CPSIE分别可以关闭和打开中断
 - 内核态执行：PSTATE.{A, I, F} 可以被CPS指令修改
 - 在用户态执行：CPS 被当做NOP指令，不产生任何效果
 - 不是特权指令

修改硬件是很麻烦的事情。软件开发者需要说服硬件制造者这样的修改可以卖更多的产品。最终方法就是改硬件，但是程序员不能在硬件被修改之前等着，所以开发了前面的这些

方法。

如何处理这些不会下陷的敏感指令？

处理这些不会下陷的敏感指令，使得虚拟机中的操作系统能够运行在用户态（EL-0）

- 方法1：解释执行
- 方法2：二进制翻译
- 方法3：半虚拟化
- 方法4：硬件虚拟化（改硬件）

处理敏感指令的方法 1：解释执行

纯软件模拟执行，没有 trap & emulate 的概念，纯软件去模拟执行。南大的著名 LAB 就是写 x86 模拟器，去模拟执行它的指令。

方法1：解释执行

- 使用软件方法一条条对虚拟机代码进行模拟

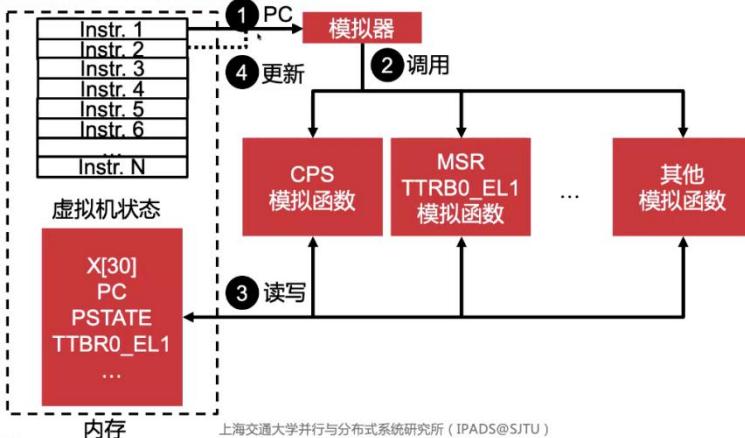
- 不区分敏感指令还是其他指令
- 没有虚拟机指令直接在硬件上执行

- 使用内存维护虚拟机状态

- 例如：使用`uint64_t x[30]`数组保存所有通用寄存器的值

我们可以用数组模拟所有的通用寄存器。虚拟机镜像如下图左边所示，这个指令序列有一个顺序，我们的模拟器就会去执行，去调用各种各样不同指令的模拟函数。每个指令对应一个函数，这个函数会对处理器的状态（通用寄存器/特殊寄存器）进行更新，然后更新 PC，挪到下一行指令继续更新。

方法1：解释执行



上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

这就是我们刚才说的例子，每次 fetch 一条指令。然后判断指令的不同类型去执行对应的操作。

例子：使用解释执行来模拟中断的设置

```
void CPU_Run(void)
{
    while (1) {
        inst = Fetch(CPUState.PC);

        CPUState.PC += 4;
        switch (inst) {
        case ADD:
            CPUState.GPR[rd] = GPR[rn] + GPR[rm];
            break;
        ...
        case CLI:
            CPU_CLI(); break;
        case STI:
            CPU_STI(); break;
        }

        if (CPUState.IRQ && CPUState.IE) {
            CPUState.IE = 0;
            CPU_Vector(EXC_INT);
        }
    }
}
```

```
void CPU_CLI(void)
{
    CPUState.IE = 0;
}

void CPU_STI(void)
{
    CPUState.IE = 1;
}

void CPU_Vector(int exc)
{
    CPUState.LR = CPUState.PC;
    CPUState.PC = disTab[exc];
}
```

Q: 这种方式有什么优缺点？

现在我们不考虑 trap & emulate 的方式，不用考虑任何硬件。它就是一个软件做出来的 CPU。缺点就是太慢了，性能会慢 40~50 倍。

解释执行的优缺点

- **优点：**

- 解决了敏感函数不下陷的问题
- 可以模拟不同ISA的虚拟机
- 易于实现、复杂度低

- **缺点：**

- 非常慢：任何一条虚拟机指令都会转换成多条模拟指令

处理敏感指令的方法 2：二进制翻译

二进制翻译的思路就是我们不再是一行行去读，是一个个 basic block 去读。翻译的时候一旦发现了代码里有一条不会下陷的敏感指令，我们就把它动态地翻译成对一个函数的调用，这也很直观，如果说其他代码没有敏感指令，那么我们就不翻译。这样我们就把 trap 变成了扫描的时候手动插一个 trap 进去。

方法2：二进制翻译

- 提出两个加速技术

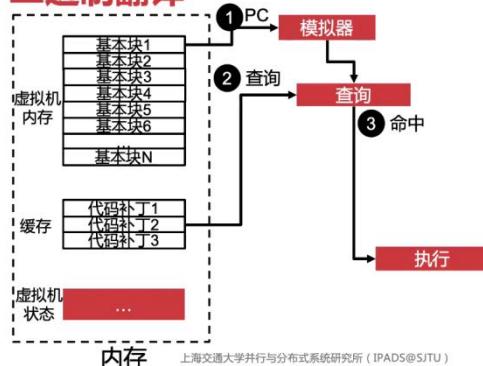
- 在执行前**批量翻译**虚拟机指令
- **缓存**已翻译完成的指令

- 使用基本块(Basic Block)的翻译粒度（为什么？）

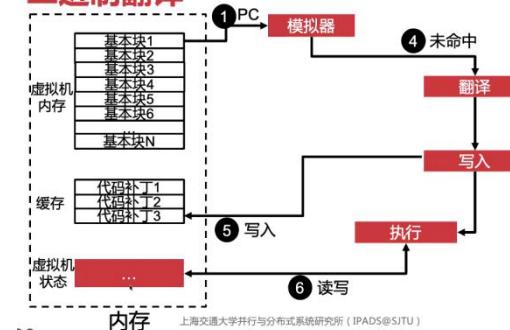
- 每一个基本块被翻译完后叫代码补丁

对于每个 basic block，我们先查询一下这个代码有没有翻译过，如果已经翻译过了，那么在内存中会有代码补丁存在，这样就不需要来来回回翻译了。

二进制翻译

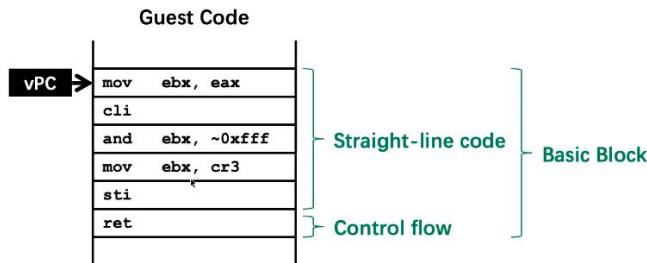


二进制翻译

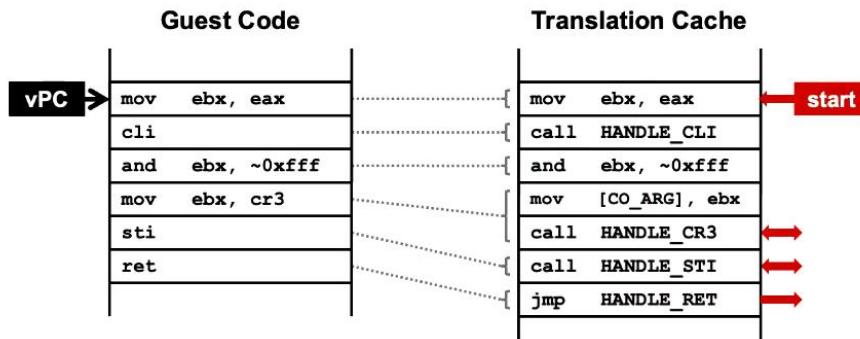


那么我们怎么翻译 basic block 呢？我们现在有一个 basic block，有 cli (关中断)。

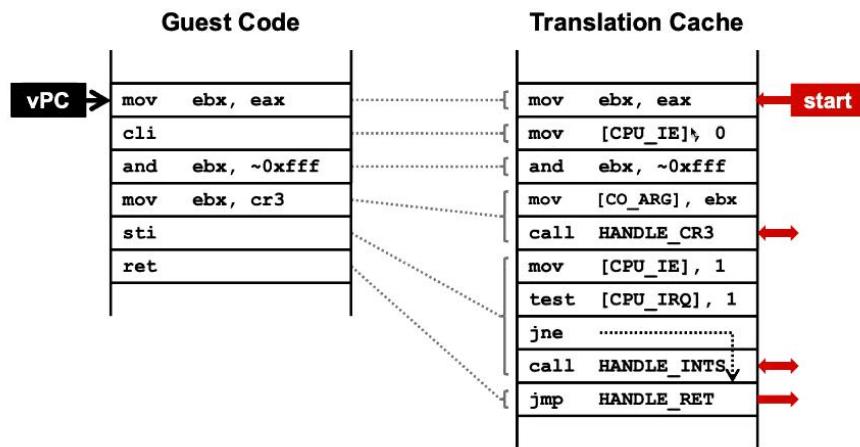
二进制翻译的 Basic Blocks



我们翻译后的结果如下，把敏感指令全部翻译成 call。对 cr3 的重要寄存器的执行，我们也是要做 trap & emulate 的。



我们继续展开，把 call 调用变成 inline 的优化，test [CPU_IRQ], 1 就是判断刚才这段时间内有没有 CPU 的中断到来，如果有就去处理中断，否则就返回。



整个翻译过程如下，如果没有找到 bb，就去翻译，如果找到了，那么我们就继续往下走。translate 就是每次 fetch，如果是 privilege，我们就去调用函数。对于不会影响状态的指令，我们就直接 emit 即可。

二进制翻译的翻译器 (Translator) -1

```
void BT_Run(void)
{
    CPUState.PC = _start;
    BT_Continue();
}

void BT_Continue(void)
{
    void *tcpc;

    tcpc = BTFindBB(CPUState.PC);

    if (!tcpc) {
        tcpc = BTTranslate(CPUState.PC);
    }

    RestoreRegsAndJump(tcpc);
}

void *BTTranslate(uint32 pc)
{
    void *start = TCTop;
    uint32 TCPC = pc;

    while (1) {
        inst = Fetch(TCPC);
        TCPC += 4;

        if (IsPrivileged(inst)) {
            EmitCallout();
        } else if (IsControlFlow(inst)) {
            EmitEndBB();
            break;
        } else {
            /* ident translation */
            EmitInst(inst);
        }
    }

    return start;
}
```

二进制翻译的翻译器 (Translator) -2

```
void BT_CalloutSTI (BTSavedRegs regs)
{
    CPUState.PC = BTFindPC(regs.tcp);
    CPUState.GPR[] = regs.GPR[];

    CPU_STI();

    CPUState.PC += 4;

    if (CPUState.IRQ
        && CPUState.IE) {
        CPUVector();
        BT_Continue();
        /* NOT_REACHED */
    }

    return;
}
```

二进制翻译对 self-modify code 支持很差，比如 java 在运行时会把 code 编译成真正的二进制，而二进制会在执行的过程中动态地修改自己的代码。这就很麻烦，我们好不容易翻译完了，把二进制修改了，那么我们有可能运行旧的代码，会发生很麻烦的事情。

中断的插入粒度变大就是说，只有在 basic block 之后才会让整个 translator 运行，在正常的二进制运行的过程中，每个代码运行之后都会产生中断。但是使用二进制翻译的过程是一个 basic block，在这里面并没有任何代码去检查中断。所以，我们必须在运行完一段代码之后，主动检查一下中断有没有开。解释执行中这个过程是比较简单的，因为它每次执行完一行代码就会检查一下中断有没有开。二进制翻译的执行过程，basic block 中是没有中断的，所以这会导致一些行为和真机是不一样的。

二进制翻译的缺点

- **不能处理自修改的代码(Self-modifying Code)**
- **中断插入粒度变大**
 - 模拟执行可以在任意指令位置插入虚拟中断
 - 二进制翻译时只能在基本块边界插入虚拟中断 (为什么?)

我们改改 OS，把敏感指令全部删掉，变成一个 Hypercall。软件既然不会触发硬件的 trap，我们直接把它变成一个会 trap 的指令，trap 到 hypervisor。但是它不太满足“虚拟环境和真实环境完全一样，它是知道自己跑在虚拟机里的。”

处理敏感指令的方法 3：半虚拟化

方法3：半虚拟化 (Para-virtualization)

- 协同设计
 - 让VMM提供接口给虚拟机，称为Hypercall
 - 修改操作系统源码，让其主动调用VMM接口
- Hypercall可以理解为VMM提供的系统调用
 - 在ARM中是HVC指令
- 将所有不引起下陷的敏感指令替换成超级调用
- 思考：这种方式有什么优缺点？

对于闭源的 OS 很难，并且也很难支持不同的版本。

半虚拟化方法的优缺点

- 优点：
 - 解决了敏感函数不下陷的问题
 - 协同设计的思想可以提升某些场景下的系统性能
 - I/O等场景
- 缺点：
 - 需要修改操作系统代码，难以用于闭源系统
 - 即使是开源系统，也难以同时在不同版本中实现

处理敏感指令的方法 4：硬件虚拟化

最后一种就是硬件虚拟化，我们引入新的模式。在 x86 下，引入 root 和 non-root。而 ARM 是引入了 EL2 特权级，它是最高权限控制物理资源。hypervisor 和 VMM 跑在 EL2 中。

方法4：硬件虚拟化

- **x86和ARM都引入了全新的虚拟化特权级**
- **x86引入了root模式和non-root模式**
 - Intel推出了VT-x硬件虚拟化扩展
 - Root模式是最高特权级别，控制物理资源
 - VMM运行在root模式，虚拟机运行在non-root模式
 - 两个模式内都有4个特权级别：Ring0~Ring3
- **ARM引入了EL2**
 - VMM运行在EL2
 - EL2是最高特权级别，控制物理资源
 - VMM的操作系统和应用程序分别运行在EL1和EL0

2022/4/28

Intel VT-x

我们以它为代表介绍 x86 的系统虚拟化的设计。大家学过 ICS，应该知道我们在 kernel 里是有 4 个 ring 的：ring0, ring1, ring2, ring3。Intel 的 CPU 将特权级别分为 4 个级别：ring0、ring1、ring2、ring3。Windows 只使用其中的两个级别 ring0 和 ring3，ring0 只给操作系统用，RING3 谁都能用。如果普通应用程序企图执行 ring0 指令，则 Windows 会显示“非法指令”错误信息。

ring3 就是用户态，对标 ARM 的 EL0。ring0 就是 kernel 运行的模式，对标 ARM 的 EL1。有了虚拟化之后，多了一个正交的模式，也就是 root 模式和 non-root 模式，也就是在不同模式下都有 4 个 ring。所以此时系统里就有 8 个 ring。

- **x86引入了root模式和non-root模式**
 - Intel推出了VT-x硬件虚拟化扩展
 - Root模式是最高特权级别，控制物理资源
 - VMM运行在root模式，虚拟机运行在non-root模式
 - 两个模式内都有4个特权级别：Ring0~Ring3

我们的 hypervisor 跑在 root 里的 ring0 和 ring3。如果我们想运行一个虚拟机，要解决让不可下陷的指令可以下陷，由 hypervisor 捕捉到；其次，并不是所有的指令都应该下陷，比如 syscall 进程想调用 svc 或者遇到了 page fault 或者除零错误，但是这个 syscall 不需要让 hypervisor 知道，hypervisor 知道了也不能做什么事情，我们希望它直接下陷到系统内核里。

所以 non-root ring-3 触发的所有 exception 是不需要下陷的，可以进一步提高系统虚拟化的性能。

VT-x的处理器虚拟化



上图中，我们发现 root ring0 里跑的 hypervisor 既可以看到进程，也可以看到虚拟机，在它看来，虚拟机和进程没什么区别。

Q：虚拟机在跑的时候，虚拟机中是维护了很多进程和线程的抽象的，这些抽象应该被我们的虚拟机监控器看到吗？虚拟机监控器应该参与到虚拟机内部的进程和线程的调度吗？

A：理论上是不应该的。

比如我们 OS 先要创建一个新的进程，它要调用 fork 这种 syscall，如果我们的虚拟机监控器能够看到这件事情，那么就等于说它要把创建出来的新的进程加到 running_queue 里调度。这个方法实现的问题就是：如果我们的虚拟机是有问题的，比如它有一个 Fork Bomb，也就是它疯狂通过 while (1) fork();去创建了大量的进程，这个时候所有创建出来的进程都会被虚拟机监控器看到，然后不停地加到 running queue 中，这个队列就会爆掉。

Q：如果不这么做的话，我们应该怎么样实现这个安全的调度呢？

A：虚拟机中创建的进程，应该加到虚拟化操作系统的队列里去维护，但是我们真正的 CPU 资源是由虚拟机监控器去管理的。所以现在的情况就是，我们的虚拟机监控器确实参与到你虚拟机的调度了，但是虚拟机监控器不应该调度你里面的线程，我们应该提供一个新的抽象：virtual-CPU。

OS 看到的都是一个个 CPU，在调度的时候把进程放到一个个队列里去。所以虚拟机监控器其实就是提供了 virtual-CPU 的抽象，虚拟机里的 OS 看到的就是一个个 virtual CPU。virtual CPU 在虚拟机监控器内部实现其实就是一个个的线程。虚拟机 OS 其实也是维护每个核上的队列，在运行的时候，就是虚拟机监控器调度时间片来做的。

在启动的时候，虚拟机告诉虚拟机监控器需要 4 个 v-CPU，虚拟机监控器创建 4 个线程对应。把时间片交给虚拟机之后，它自己会维护 run-queue 中的虚拟机线程去跑，这就是两层的模型。所以，因为这 4 个 v-CPU 对应的 4 个线程，所以它们可能同时在跑，也有可能只有几个在跑，要对应到物理的 CPU 上有没有同时在跑这几个线程以及当前的虚拟机监控器的调度是什么样的。

这会造成一个很有意思的问题，虚拟化中叫做 double-scheduling。就是上层调度所带来的影响。

在传统内核中，我们现在有两个核，有一个核上拿了 spin lock 进入 critical section，另

外一个核上的也想进入 critical section，尝试去拿 spin lock，但是发现锁被人持有着。然后第一个核出 critical section 后放 spin lock，第二个核拿到以后就可以进 critical section 了。

但是在虚拟化中，并不是所有的 v-CPU 在同一时刻都在运行，所以持有 spin lock 的 v-CPU 可能在 critical section 执行到一半的时候被调度出去了，此时别的 v-CPU 再想通过等待的方式获取 spin lock，等待时间就会特别长，导致性能就会比较差。

Q: 我们该怎么恢复虚拟机中的页表基地址寄存器呢？也和其他寄存器一样从栈上 pop 出来吗？

A: 普通的寄存器是没有问题的。因为我们 x86 一个核上只能看到一个页表，一部分区域是映射内核空间的，一部分区域是映射进程空间的。在没有虚拟化的时候，Context Switch 都不用换页表。但是，引入虚拟化之后，虚拟机 OS 内核中也要有一个页表，但是虚拟机监控器也是一个内核，导致也要把自己的虚拟地址映射到同一个 CR3 的位置。我们要做到进入虚拟机就使用虚拟机 OS 页表，在外面就使用虚拟机进程的页表。

这导致我们必须要换页表，但是在切换的时候，我们要格外小心。我们要保证代码在切换页表前后都要能够执行。我们必须保证这个页必须在切换前后都能够映射。客户机在页表里能够看到一页是不属于自己的，这就破坏了虚拟机不知道自己是虚拟机的假设。

所以用软件恢复比较难做，为了解决这样一个难题，INTEL VT-X 提出了一个机制，VMCS。它其实就是一个内核里的页，记录了 VM 运行的所有寄存器状态。在虚拟机监控器想跑虚拟机的时候（VM Entry），自动地从内存加载到我们的 CPU 里；在从虚拟机回到虚拟机的时候（VM Exit），CPU 又自动地把虚拟机状态保存到 VMCS 里。这些都不需要我们去做，避免了我们需要精巧地设计一个跳转的页。

Intel VT- VMCS

Virtual Machine Control Structure (VMCS)

- **VMM提供给硬件的内存页 (4KB)**
 - 记录与当前VM运行相关的所有状态
- **VM Entry**
 - 硬件自动将当前CPU中的VMM状态保存至VMCS
 - 硬件自动从VMCS中加载VM状态至CPU中
- **VM Exit**
 - 硬件自动将当前CPU中的VM状态保存至VMCS
 - 硬件自动从VMCS加载VMM状态至CPU中

Q: VMCS 里为什么要保存虚拟机监控器的状态呢？

A: 因为我们机器里，系统寄存器只有一份。在 VM exit 的时候，我们回到系统虚拟机。在执行第一条指令之前，机器上的所有寄存器都是 Host 了。这一串硬件做的事情有点像是调度里的上下文切换。

VT-x VMCS的内容

- 包含6个部分

- Guest-state area: 发生VM exit时，CPU的状态会被硬件自动保存至该区域；发生VM Entry时，硬件自动从该区域加载状态至CPU中
- Host-state area : 发生VM exit时，硬件自动从该区域加载状态至CPU中；发生VM Entry时，CPU的状态会被自动保存至该区域
- VM-execution control fields : 控制Non-root模式中虚拟机的行为
- VM-exit control fields : 控制VM exit的行为
- VM-entry control fields : 控制VM entry的行为
- VM-exit information fields : VM Exit的原因和相关信息（只读区域）

VM entry 和 VM exit 的开销是比较大的。VMCS 是保存虚拟机上下文状态的特殊的内存页，如果一个 hypervisor 跑 4 个 v-CPU 的虚拟机，那么对应的是 4 个线程，那么在这个情况下应该需要几个 VMCS 呢？很简单，需要 4 个。因为 VMCS 只保存了一个 v-CPU 对应的状态。hypervisor 管理了 4 个 v-CPU，那么我们应该维护 4 个 VMCS 的状态。

VM-execution control field 还涉及虚拟机行为的控制，比如一些虚拟机的 `syscall` 因为性能考量，应该直接被 Guest 处理掉。但是下陷与否是可以通过这个字段控制的。我们可以选择抓捕 `page fault`，然后在 hypervisor 中做分析。

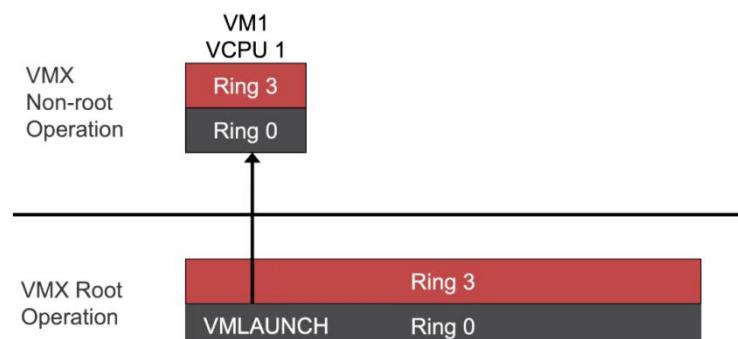
我们是不让虚拟机直接管理中断的，在 hypervisor 看来是 `virtual interrupt`，它其实是 hypervisor 插进去的。它其实就是在 `vm-entry` 的 fields 中设置 flag 来插入软件的中断。

第一步：hypervisor 在 root 的 ring0 调用 `VMXON`，也就是告诉 CPU 准备硬件虚拟化了。

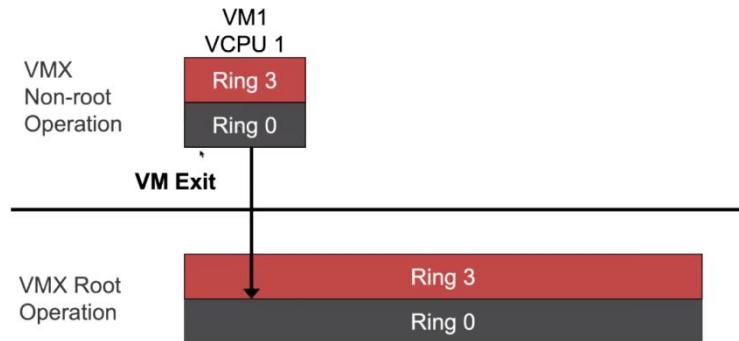
VT-x 的执行过程



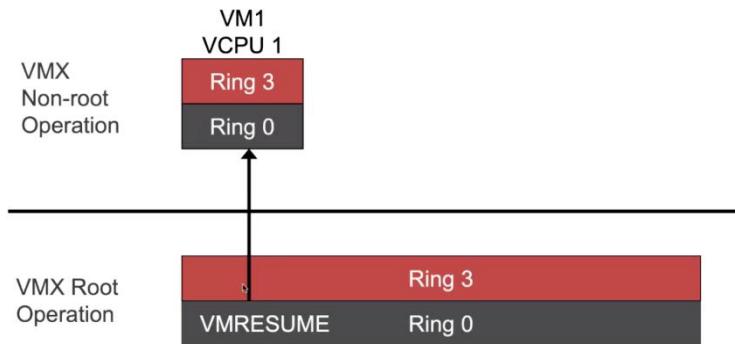
第二步就是在第一次打开虚拟机的时候调用 `vmlaunch`，告诉 CPU 我们要启动一个虚拟机。在跑这个指令之前，我们应该把 VMCS 准备好。



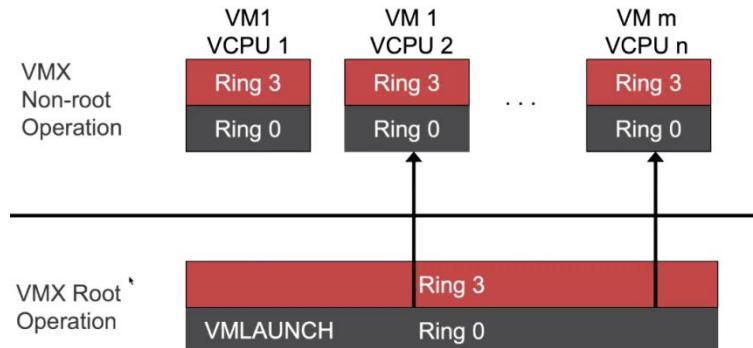
在虚拟机运行过程中，会有 VM Exit，把相关的页保存到 VMCS 里。



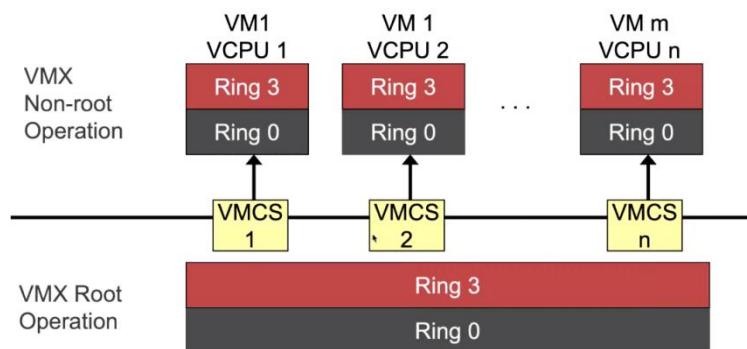
保存完之后，我们再调用 `vmresume`。它会帮助我们恢复虚拟执行。



其他的所有 v-CPU 也是相同的操作。



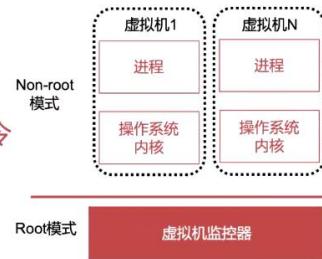
根据下图我们可以看到，一共有 m 个虚拟机，一共有 n 个 v-CPU。每个 v-CPU 都会对应一个 VMCS，所以一共有 n 个 VMCS。我们的 hypervisor 在保存和恢复的时候都是通过控制 VMCS 来的。



VM-Exit 就是通过各种方式下陷到 hypervisor 中进行处理。

x86中的VM Entry和VM Exit

- **VM Entry**
 - 从VMM进入VM
 - 从Root模式切换到Non-root模式
 - 第一次启动虚拟机时使用**VMLAUNCH**指令
 - 后续的VM Entry使用**VMRESUME**指令



- **VM Exit**
 - 从VM回到VMM
 - 从Non-root模式切换到Root模式
 - 虚拟机执行敏感指令或发生事件(如外部中断)

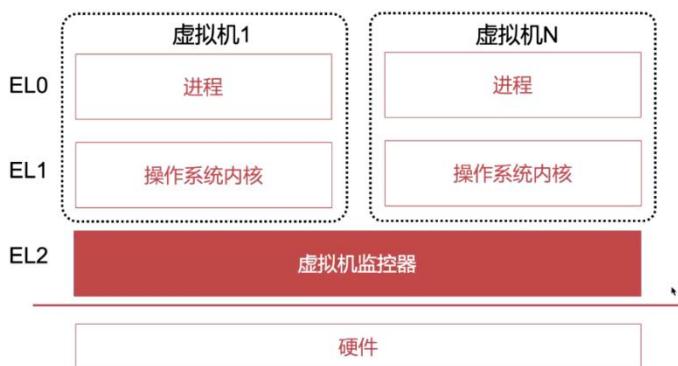
以上就是 x86 的一个典型虚拟化机制。

ARM 的虚拟化技术

ARM 并没有采取和 x86 一样的设计，而是另辟蹊径。

在没有硬件虚拟化的时候，有 EL0 和 EL1。有了硬件虚拟化之后，多了一个 EL2 专门跑虚拟机监控器。此时系统里 EL2 是最高特权级。在 x86 中，某一个瞬间只能有 1 个 CR3。但是在 ARM 中与众不同，比如 ttbr0_el1 就是专门为 el1 使用的。所有寄存器后面都有下划线_el2，这些不同导致 hypervisor 对 ARM 做硬件虚拟化支持的时候会有很多障碍。第二个不同是，有了 el2 之后，我们的页表格式也会发生变化，这个页表的格式和过去 el1 的格式不一样。

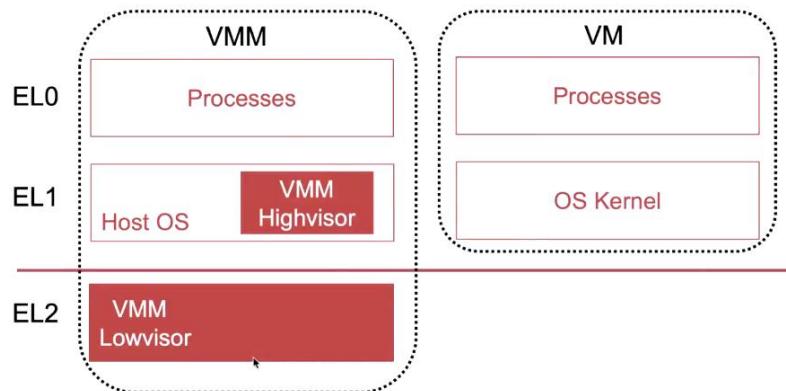
ARM的处理器虚拟化



type-1 就是不依托 Host-OS。如果是 **type-1**，可以很自然地使用这样的一种虚拟化方式。但在 **type-2** 里（Linux），我们要把它变成 hypervisor 就很难做。Linux 里是有 KVM 的，在 x86 里，直接加载虚拟化就行。但是因为 ARM 的特殊要求，这就带来了很多复杂性。

比如我们要把 Linux 变成 Type-2 的 hypervisor，因为我们 ARM 设计的不同，就造成了我们没办法简单的把 Linux 放在 EL2 里跑。为了解决这个问题，Linux 社区不得不为 ARM 做了精巧的设计，也就是把 Linux 里硬件虚拟化的部分分为 VMM Highvisor（跑在 EL1 里）和 VMM Lowvisor（跑在 EL2 里）。Linux 里其他的功能依然跑在 EL1，这样我们可以继续复用之前的代码。

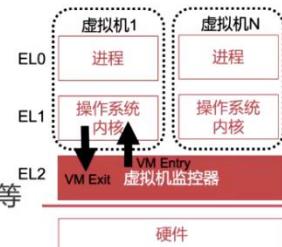
ARMv8.0中的Type-2 VMM架构



在 ARM 里，使用 `eret` 进入 VM。但 ARM 里没有 VMCS。在进出 hypervisor，机器里有很多基址寄存器，不会互相影响。硬件自动地在 `eret` 的瞬间，做到了切换 el1 页表，不需要我们主动设计跳转页。在下陷的一瞬间，EL2 可以看到 Guest Kernel 在 EL1 里使用的所有寄存器。命名不同的好处就是不需要 VMCS 了，因为它的开销还是很大的。

ARM的VM Entry和VM Exit

- **VM Entry**
 - 使用 `ERET` 指令从 VMM 进入 VM
 - 在进入 VM 之前，VMM 需要 **主动** 加载 VM 状态
 - VM 内状态：通用寄存器、系统寄存器、
 - VM 的控制状态：`HCR_EL2`、`VTTBR_EL2` 等
- **VM Exit**
 - 虚拟机执行敏感指令或收到中断等
 - 以 `Exception`、`IRQ`、`FIQ` 的形式回到 VMM
 - 调用 VMM 记录在 `vbar_el2` 中的相关处理函数
 - 下陷第一步：VMM **主动** 保存所有 VM 的状态



ARM硬件虚拟化的新功能

- ARM中**没有**VMCS
- VM能直接控制**EL1**和**EL0**的状态
 - 自由地修改 `PSTATE` (VMM 不需要捕捉 `CPS` 指令)
 - 可以读写 `TTBRO_EL1`/`SCTRL_EL1`/`TCR_EL1` 等寄存器
- VM Exit时VMM仍然可以直接访问VM的**EL0**和**EL1**寄存器
- 思考题1：为什么ARM中可以不需要VMCS？
- 思考题2：ARM中没有VMCS，对于VMM的设计和实现来说有什么优缺点？

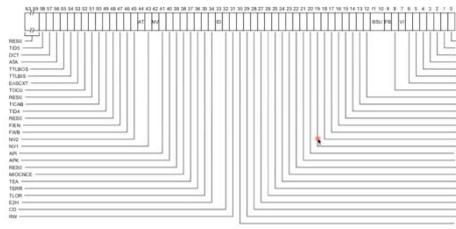
ARM 没有 VMCS 对性能有好处。

它有一大排 bit，决定了虚拟机里的下陷和行为。

HCR_EL2寄存器简介

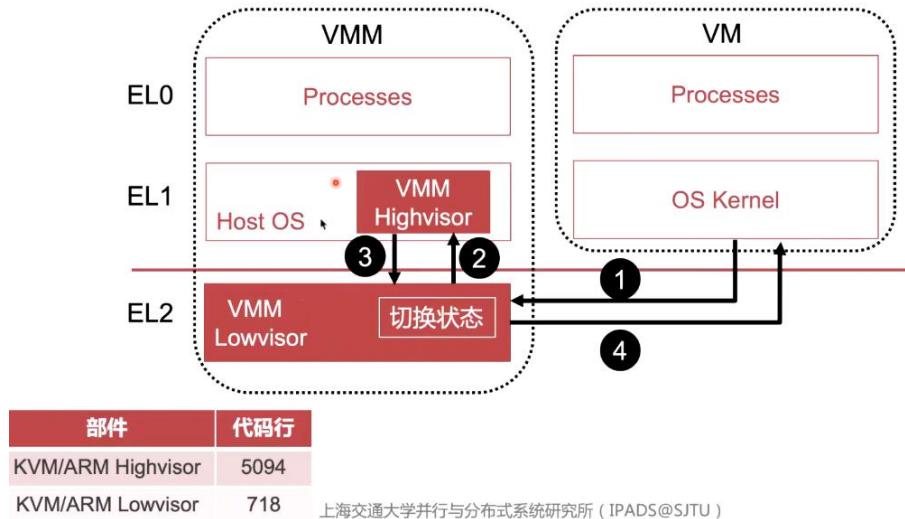
- HCR_EL2 : VMM控制VM行为的系统寄存器
 - VMM有选择地决定VM在某些情况下下陷
 - 和VT-x VMCS中VM-execution control area类似
- 在VM Entry之前设置相关位，控制虚拟机行为
 - TRVM(32位)和TVM(26位): VM读写内存控制寄存器是否下陷，例如SCTRL_EL1、TTBR0_EL1
 - TWE(14位)和TWI(13位): 执行WFE和WFI指令是否下陷
 - AMO(6位)/IMO(5位)/FMO(4位): Exception/IRQ/FIQ是否下陷
 - VM(0位): 是否打开第二阶段地址翻译

HCR_EL2寄存器简介



对于 type-2 的 hypervisor，不得不做 split 的操作。虚拟机下来一瞬间，进入到我们的 Lowvisor，保存和恢复寄存器（EL1）状态到内存里，然后继续交给 Highvisor 处理。我们下到 EL2 中，又需要把 Highvisor 的寄存器存到内存里面，然后恢复虚拟机 EL1 的寄存器和状态。本来这个设计是为了增加性能，但是 type-2 没有办法享受到这个好处。

ARMv8.0中的Type-2 VMM架构



Q: 这种架构有什么缺点？

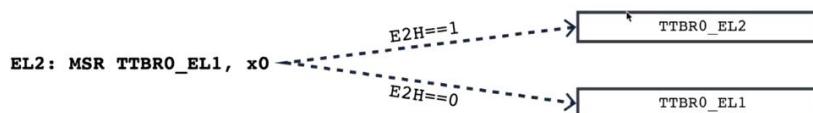
A: 我们需要转到 EL1 里，VM-exit 的开销也很大。

ARM 发现硬件虚拟化给 Linux 社区带来了很大的困难。

ARMv8.1中的Type-2 VMM架构

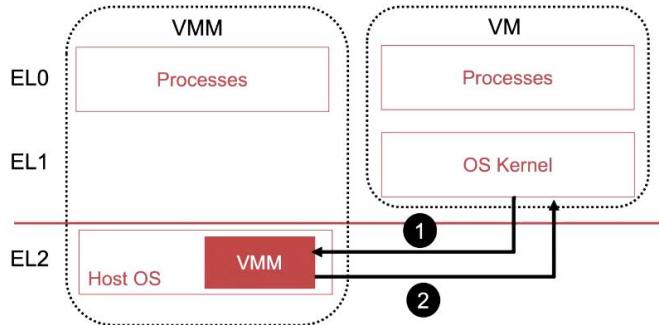
• ARMv8.1

- 推出Virtualization Host Extensions(VHE)，在HCR_EL2.E2H打开
 - 寄存器映射
 - 允许与EL0共享内存
- 使EL2中可直接运行未修改的操作系统内核 (Host OS)



有了这个之后，我们就不再需要设计精巧的 Highvisor 和 Lowvisor 架构了，我们 OS 可以直接跑在 EL2 里。我们直接 `eret` 就可以回到虚拟机中，这样下陷的开销降低了。

ARMv8.1中的Type-2 VMM架构



2022/5/5

我们的虚拟化分成 CPU 虚拟化、内存虚拟化、IO 虚拟化。CPU 虚拟化到底是什么呢？我们过去的 OS 看到的是真实机器上的物理核，现在用了虚拟机之后，我们经常听到给虚拟机配 4 个 v-CPU。那 v-CPU 和真实 CPU 的关系是什么呢？我们今天会专门去讲讲这个。它是一个个版本演化的，因为我们还没讲到内存虚拟化和 IO 虚拟化，所以这个虚拟机的功能还是不足的，比如它可能暂时还不能分配页表。

CPU虚拟化功能：迭代演进、分步理解

- 第一版：支持只有内核态的虚拟机
- 第二版：支持虚拟机内的时钟中断
- 第三版：支持虚拟机内单一用户态线程
- 第四版：支持虚拟机内多个用户态线程
- 第五版：支持多个虚拟机间的分时复用
- 第六版：支持多个物理CPU
- 第七版：支持多个虚拟CPU

第一版：虚拟机只在内核态运行简单代码

第一个版本就是虚拟机是非常简单的，整个物理机上只有一个虚拟机，它没有内核态和用户态的切换，它只有一个 VM 内核态，里面只能运行用户 ISA，也就是不会修改系统状态的指令、也没有敏感指令（只有加减乘除、`jump`、`return` 等）。这个虚拟机在用户态运行的时候不会造成任何的下陷。

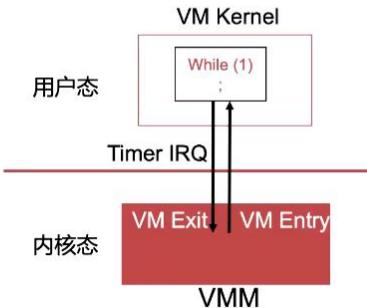
第一版：虚拟机只在内核态运行简单代码

- **VM的能力**

- 只支持一个VM
- 没有内核态与用户态的切换
- 只有内核态，且仅运行用户ISA的指令（与用户态没有区别）

- **VMM的实现**

- 处理时钟中断造成的VM Exit



我们可以从上右图看到，虚拟机内部其实就是一个 `while(1)` 的死循环，我们也可以把它看成一个进程。

Q: 支持这样一个虚拟机，我们的 hypervisor 需要做什么事情？

A: 只需要能把 `while(1);` 运行下去即可，所以我们的 hypervisor 需要支持时钟中断。我们需要让系统永远能够检查内核态有没有异常、或者收回整个 CPU 控制权。所以我们要支持一个 timer，这和我们 ChCore 中实现的一样。CPU 自己会触发这个 physical interrupt 来唤醒我们的内核态，在这个例子里，我们执行 `while(1);` 的时候会定时触发物理中断把控制权拿回 VMM 中，然后再去检查用户态的执行情况。检查完以后，我们继续往后设置一个时间中断（offset）。进出虚拟机可以认为是 ChCore 进出一个用户态进程。

第二版：虚拟机内部支持时钟中断

在第二版中，我们允许虚拟机内部自己也可以处理时钟中断。这个也很常见，因为一般用户态程序没有办法配置时钟中断，而内核中我们需要支持不断触发时钟中断做一些操作，比如中断。所以虚拟机内部是有处理时钟中断的需求的。

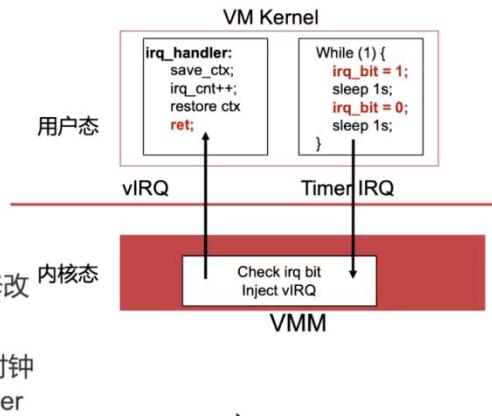
Q: 我们已经在机器上用到了时钟中断（hypervisor），但是现在我们虚拟机也要用，但是只有一个中断的硬件，应该怎么办呢？

A: 其实很简单，因为我们有了软件层的 hypervisor，我们不应该直接把物理时钟中断直接暴露给虚拟机。我们给它一个 virtual 时钟中断。VM kernel 只需要借助这个中断触发所带来的控制流的变化，所以 VMM 可以通过软件的方式模拟时钟中断。

那么怎么样才能让用户态感知不到到底是软件还是硬件呢？我们要借助没有虚拟化的时候的时钟中断的逻辑，我们的 hypervisor 不应该打破这种抽象。所以，虚拟机应该能够开关时钟中断来配置；并且当中断触发了以后，应该有一个 handler；我们还需要中断的屏蔽接口。

第二版：虚拟机内部支持时钟中断

- VM的能力
 - 设置irq_handler
 - 开关时钟中断 (irq_bit)
 - 运行时钟中断处理函数
- VMM的实现
 - 捕捉VM对irq_handler的修改
 - 捕捉VM对irq_bit的修改
 - 根据irq_bit决定插入虚拟时钟中断vIRQ并调用irq_handler



如上右图所示，这是用户态使用 `interrupt` 的。有了中断以后，我们代码分成没有中断触发的时候的主逻辑（`while(1)`循环）。当 `irq_bit=1` 的时候，我们认为中断打开（愿意接收中断），`sleep` 可以是任何的内核的功能。它的意图就是显示两个不同的区间。打开中断以后，hypervisor 可以知道 VM 有没有打开中断。如果 hypervisor 拿到了一个物理的中断（Timer IRQ），并且 VM 打开了中断，那么 hypervisor 负责把虚拟中断插入 VM 中（hypervisor 主动调用 VM 中的 `irq_handler`），执行完之后再 `ret` 回 hypervisor 中（此处也是一个 trap），由 hypervisor 继续恢复 VM 原先的执行流。

Q: 怎么支持这样的操作呢？

A: 在启动的时候，虚拟机需要设置 `irq_handler`，这个设置是要 trap 下来的，hypervisor 需要记住 `irq_handler` 的地址。在运行过程的时候，hypervisor 需要知道 VM 的中断有没有被屏蔽，所以在设置 `irq` 的权限位的时候，也需要下陷（trap）到 hypervisor 中。我们可以认为权限位是一个特殊的寄存器，访问的时候就需要下陷。当 `irq` 触发的时候，hypervisor 就可以知道是否需要插入。

第三版：虚拟机内支持运行单一用户态线程

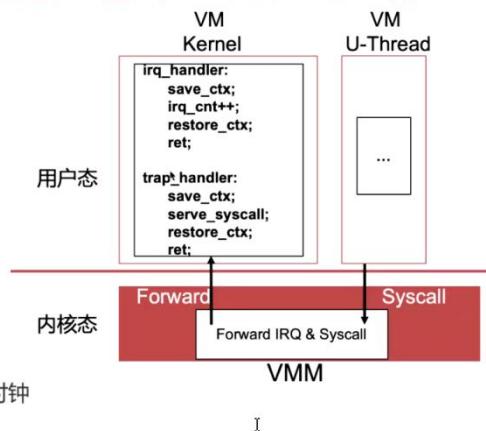
Q: 为什么我们需要时钟中断呢？

A: 时钟中断实际上是为了支持多线程的。线程具体做什么不重要。

所以，第三版的 VM 中，VM 中包含了用户态和内核态，并且用户态中运行了一个用户态线程（U-Thread）。我们用户态线程可以调用 `syscall`，可以使用内核提供的服务。并且用户态线程依旧是可以在 VM 的虚拟时钟中断被打断的。

第三版：虚拟机内支持运行单一用户态线程

- VM的能力
 - 虚拟机包含内核态与用户态
 - 用户态运行一个用户态线程
 - U-Thread
 - 用户态线程可调用内核syscall
 - 用户态线程可被时钟中断打断
- VMM的实现
 - 捕捉并转发U-Thread系统调用 syscall
 - 转发syscall至VM内核
 - 捕捉并转发U-Thread执行时的时钟中断



由上右图可知，我们内核态中不仅仅可以包括 `irq_handler`，还可以包括 `trap_handler`（它只用来处理 `syscall`）。当用户态线程在执行的时候，比如调用了一个 `syscall` 的特殊指令（如 ARM 里的 `svc`），这是因为是一个系统 ISA，所以它会下陷到 hypervisor，hypervisor 发现 `svc` 不是它自己能处理的，所以它就把这个 `syscall` 转发给 VM 的内核中的特定的 `trap_handler`。VM 在启动的时候会注册很多的 handler，hypervisor 就知道了每个 handler 各自的 offset。所以 hypervisor 也会设置到对应的 handler 的 offset 的位置，这个逻辑和第二版是一致的。

这就是我们为了支持单一的用户态线程，hypervisor 需要添加的 `forward` 功能。当用户态线程调用 `syscall` 的时候，我们需要转发到相应的虚拟机内核。而如果是 VM kernel 调用了 `syscall`，说明虚拟机内核出问题了，因为我们的 hypervisor 并没有提供给 VM kernel 调用 `syscall` 的服务。

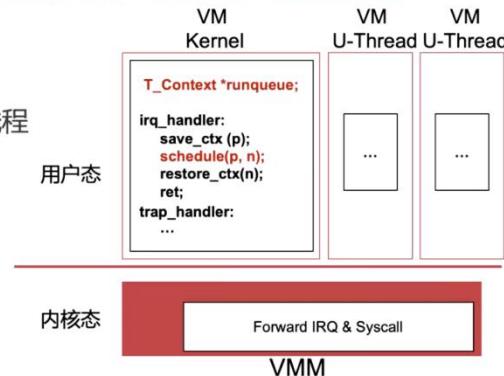
通过这个区别，我们的内核态应该标记处哪些是用户态、哪些是内核态的。比如启动的时候，我们可以知道内核态是哪个，或者我们可以标记出 U-thread 的位置，只有 U-thread 的 `syscall` 再做转发。

第四版：虚拟机内部支持多个用户态线程

第四版中，我们对虚拟机的拓展就是让它能够支持多个用户态线程。这个虚拟机里就有多个线程，这些线程可以被虚拟机内核做调度、切换到相应的线程去执行。它在里面维护一个 `run_queue`，里面的元素就是维护了 `thread_context`（比如用户态线程的寄存器状态）。

第四版：虚拟机内部支持多个用户态线程

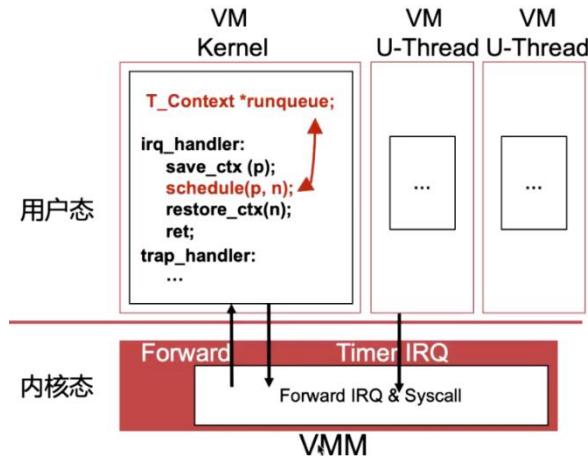
- VM的能力
 - 用户态运行多个用户态线程
 - 内核可调度用户态线程
- VMM的实现
 - 与第四版相同



思考：Fork bomb是否会影响VMM？

比如线程 1 跑着跑着触发了一个物理的时钟中断进入到 hypervisor，hypervisor 发现这

一个线程的时间片已用尽，那么应该给它插入一个虚拟的时钟中断，进入到 VM kernel 处理时钟中断，它会先保存当前 thread 的 context，第二步就是它自己实现调度逻辑。这个调度器内部可以使用各种各样的算法实现，外部接口很简单，就是 `schedule(p, n)`，`p` 就是前一个线程、`n` 就是后一个线程。所以我们下一步恢复即将跑的线程的信息加载到所有的寄存器里，最后就是调用 `return` 去跑最新的用户态线程。



所以，我们为了支持用户态线程，内核态的 hypervisor 其实没有做什么事情，只是多了 VM kernel 里的一个 `runqueue` entry 去保存多个线程的信息。

Q: 如果一个 VM U-thread 会调用 `fork bomb`（不断以几何倍数地创建其他的线程），是否会影响 VMM（hypervisor）？

A: 不会，因为 hypervisor 不知道上面在做什么，甚至不知道 VM 中有几个线程，因为线程的所有信息都被存在 VM kernel 的 `runqueue` 中。hypervisor 不关心我们虚拟机内核的实现，它只需要做一些 `trap & emulate` 的事情。

Q: 如果 `runqueue` 的内存占用在不断增长，会有什么影响？

A: 最终导致 VM kernel 的内存爆掉，但是还是不会对 hypervisor 产生影响（之后的内存虚拟化会提这件事情）。hypervisor 可以限制虚拟机的内存的使用量。

第五版：支持多个虚拟机间的分时复用

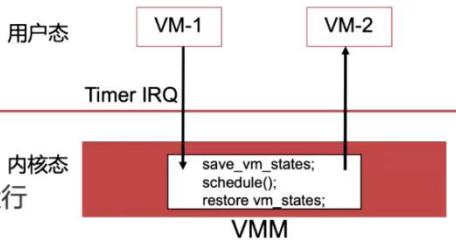
对于单个虚拟机，功能基本上完整了。我们希望物理机上可以跑更多的虚拟机。下图就是 2 个虚拟机的例子，虚拟机之间的切换是借助于物理时钟中断去触发的，无论跑的是哪个虚拟机，都会下陷到我们的 hypervisor，这时候 hypervisor 就可以有自己的调度，hypervisor 调度 VM 的时候也应该有自己的 `run_queue`，里面保存了虚拟机每次在切换的时候需要恢复的数据。

我们可以认为一个虚拟机就是一个内核态的线程。最简单来说，线程就是一组寄存器和一个 `stack`。我们调度是调度内核态的线程，但是它是为每一个虚拟机单独服务的。

当时钟中断触发以后，进到内核态的 VMM，把所有 VM 保存当 `context` 里，scheduler 决定下一个运行的 VM，然后再恢复对应的 VM 的状态，然后再跳转到新的虚拟机去执行。

第五版：支持多个虚拟机间的分时复用

- VM的能力
 - 支持多个VM
- VMM的实现
 - 每个VM对应一个内核线程
 - 维护VM_runqueue队列
 - 每个元素对应一个VM的运行状态、
 - 由VMM实现VM间切换
 - 保存和恢复VM寄存器

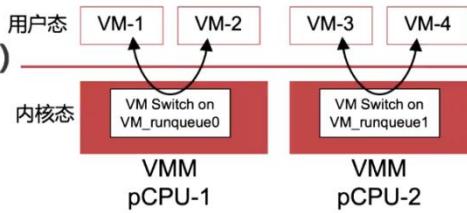


第六版：VMM 支持多个物理 CPU

我们刚才讲到所有的 VM 的调度都是在一个 CPU 上的，但是我们现在的机器都有多个 CPU。我们希望 hypervisor 利用多个物理 CPU，让 VM 跑在多个 CPU 上。我们对 hypervisor 做一些拓展，我们对每个物理核上维护一个 runqueue，只为自己对应的物理核做服务。

第六版：VMM支持多个物理CPU

- VM的能力
 - 与第五版相同
- VMM的实现（基于第五版）
 - 为每个pCPU维护不同的 VM_runqueue



VM-1 触发了时钟中断下陷到内核，hypervisor 在 pCPU-1 上跑调度器，继续调度 VM-2。在这个调度过程中，pCPU-2 完全不受影响，可以做自己的事情，VM-3 可能此时正在运行。通过这种方法，我们可以在多个 CPU 上运行我们的虚拟机，但是每一个时刻，一个 VM 只能在一个 CPU 上去跑。

既然我们的 hypervisor 可以跑多个物理 CPU，那么虚拟机是不是也有多个 CPU 的概念呢？这就是我们虚拟机中的 v-CPU (virtual CPU) 的概念。

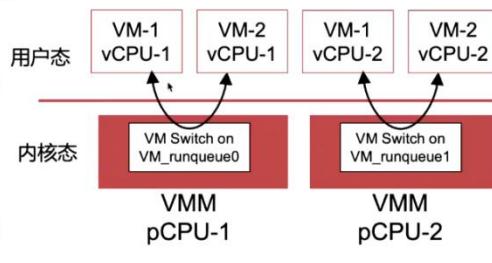
第七版：虚拟机支持多个虚拟CPU

- VM的能力（与第六版的区别）

- 虚拟机有多个Virtual CPU (vCPU)

- VMM的实现

- 在VM_runqueue中标记出VM和vCPU的类型



hypervisor 要做的事情就是在虚拟机启动的时候根据它配置的核心数（如：2个 v-CPU），hypervisor 在内核态里加两个内核态的 context 线程。在调度的时候，就可以从一个 v-CPU 切换到另一个 v-CPU。调度也和上一版非常相似，只是上一版的调度单元是虚拟机，而这一版的调度单元是 v-CPU，VM 有多 CPU 之后，就可以做传统 OS 做的事情。

VM 虽然看到的是两个 v-CPU，它不能决定自己真的跑在物理 CPU 上，比如上图中，同一个 VM 的两个 v-CPU 都跑在了同一个物理 CPU 上，这导致 VM 同一时刻只会有一个 v-CPU 运行。

上节课我们提到了，虚拟化会带来传统 OS 的一个概念的打破。也就是我们传统 OS 使用 spin lock 进入 critical section，它会认为这个 critical section 的持续时间很短。但是在虚拟化的情况下，比如 vCPU-1 拿锁进入了 critical section，vCPU-2 在外面以 spin lock 的形式等待。但是因为是 vCPU，所以此时 vCPU-1 此时不一定在运行，所以这会造成等锁的人等待时间进一步加长了，需要等待多个时间片。

这 7 个版本都是一个理想化的模型，我们假设系统 ISA 都可以被系统捕捉到（可以下陷）。但是在 ARM 中开关中断是不会下陷的，所以我们之前讲的很多例子都会解决这个问题。

Review: ARM不是严格的可虚拟化架构

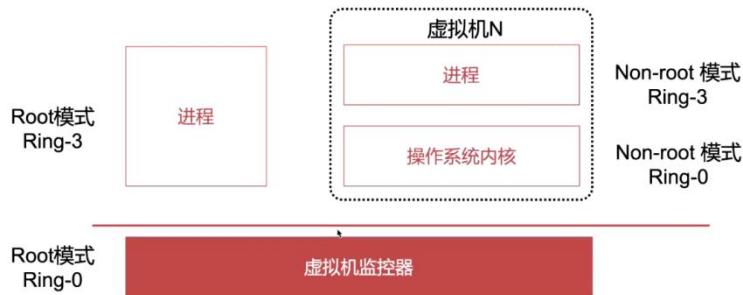
- 在ARM中：不是所有敏感指令都属于特权指令

- 例子：CPSID/CPSIE指令

- CPSID 和 CPSIE 分别可以关闭和打开中断
 - 内核态执行：PSTATE.{A, I, F} 可以被 CPS 指令修改
 - 在用户态执行：CPS 被当做 NOP 指令，不产生任何效果
 - 不是特权指令

上节课我们讲到，可以通过改硬件的方式提供新的硬件虚拟化的机制让所有的系统 ISA 执行的时候会下陷，我们提到了 Intel 的 VT-x 技术和 ARM 的技术。

Review: Intel VT-x的处理器虚拟化

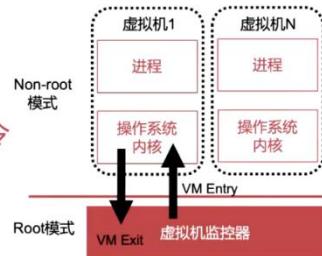


Intel VT-x 就是加了一个全新的 Non-root 模式和 Root 模式，让虚拟机跑在 Non-root 模式里，hypervisor 跑在 root 模式里，虚拟机里的所有系统 ISA 都会下陷到 hypervisor 里去操作，它还可以做到不该让 hypervisor 知道的下陷可以不下陷。

x86 在进出虚拟机的时候会有特殊的操作：VMCS（提前在 hypervisor 中把虚拟机需要使用的状态加载到内存页中），然后我们调用 VM launch 和 VM resume 进入到虚拟机中。虚拟机在下陷的时候，硬件会把所有的寄存器状态加载到 VMCS 中，hypervisor 可以读这个 VMCS 做相应的处理。

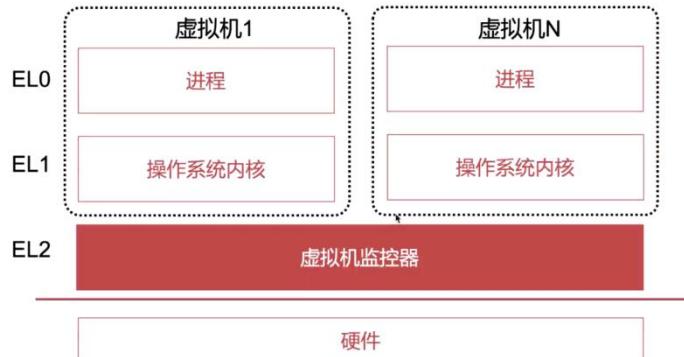
Review: x86中的VM Entry和VM Exit

- **VM Entry**
 - 从VMM进入VM
 - 从Root模式切换到Non-root模式
 - 第一次启动虚拟机时使用**VMLAUNCH**指令
 - 后续的VM Entry使用**VMRESUME**指令
- **VM Exit**
 - 从VM回到VMM
 - 从Non-root模式切换到Root模式
 - 虚拟机执行敏感指令或发生事件(如外部中断)



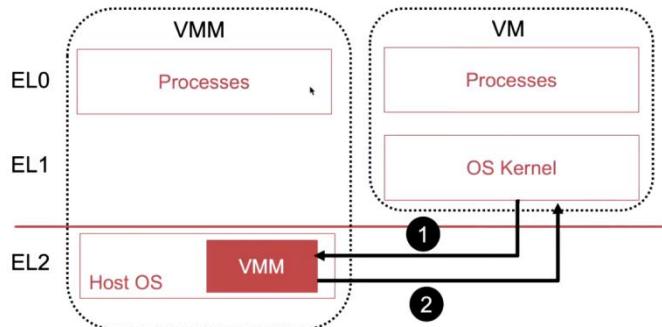
ARM 的解决方案不一样。它凭空多加了一层 EL2，一开始是为 Type-1 hypervisor 设计的，所以在遇到 Type-2 hypervisor 的时候需要对 Linux 做一些定制化的修改

Review: ARM的处理器虚拟化



在 ARM 8.1 中对硬件做了一些修改，使得 Linux 可以运行在 EL2 中。下陷到 hypervisor 做相应的处理，处理完直接回到虚拟机中的刚才的位置（可能是 VM 内核态，也可能是 VM 用户态）。

Review: ARMv8.1中的Type-2 VMM架构



案例：QEMU/KVM

我们举一个真实的，工业界中使用最多的 hypervisor 是怎么支持硬件虚拟化和提供 v-CPU 抽象的。QEMU 和 KVM 其实是两个完全不同的东西，QEMU 发展的更早。它当时的目的是在非 x86 机器上通过软件模拟 x86 机器。

QEMU发展历史



- **2003年，法国程序员Fabrice Bellard发布了QEMU 0.1版本**
 - 目标是在非x86机器上使用动态二进制翻译技术模拟x86机器
- **2003-2006年**
 - 能模拟出多种不同架构的虚拟机，包括S390、ARM、MIPS、SPARC等
 - 在这阶段，QEMU一直使用[软件方法](#)进行模拟
 - 如二进制翻译技术

KVM 叫做 Kernel-based Virtual Machine，它的虚拟化功能都是基于 Linux 功能去实现的。它使用了硬件虚拟化的机制。QEMU 不再决定使用软件方法，会主动使用 KVM 去替代软件模拟，QEMU 需要使用硬件虚拟化的能力去实现虚拟机，但是怎么使用需要 QEMU 去主动配置。

KVM发展历史

- 2005年11月，Intel发布带有VT-x的两款Pentium 4处理器
- 2006年中期，Qumranet公司在内部开发KVM(Kernel-based Virtual Machine)，并于11月发布
- 2007年，KVM被整合进Linux 2.6.20
- 2008年9月，Redhat出资1亿多美元收购Qumranet
- 2009年，QEMU 0.10.1开始使用KVM，以替代其软件模拟的方案

QEMU/KVM架构

- **QEMU运行在用户态，负责实现策略**
 - 也提供虚拟设备的支持
- **KVM以Linux内核模块运行，负责实现机制**
 - 可以直接使用Linux的功能
 - 例如内存管理、进程调度
 - 使用硬件虚拟化功能
- **两部分合作**
 - KVM捕捉所有敏感指令和事件，传递给QEMU
 - KVM不提供设备的虚拟化，需要使用QEMU的虚拟设备

KVM 这部分可以直接使用 Linux 的功能，它要分配虚拟机内存的时候，可以直接调用物理内存的管理模块。这两部分相互配合完成了 hypervisor 的整个的功能。

由于 KVM 是运行在内核态的，QEMU 需要调用 KVM 的用户态接口才能使用它的功能，这个接口就是文件接口。这个文件是/dev/kvm，我们可以通过文件接口 open/close/read/write 进行交互，因为这个文件是特殊的，我们只需要使用特殊的接口 ioctl 即可。它是一个和驱动相关的 syscall，只要我们实现了相应功能，它就可以暴露给用户态。当用户态 QEMU 在这个文件上调用 open 之后，拿到了一个 fd，后面调用的就是 ioctl，第一个参数就是 fd，第二个参数应该指定的是相应的 command，比如 CREATE_VM、CREATE_VCPU（创建 VM 之后，就可以创建 v-CPU，其实就是在 KVM 中创建 v-CPU 的 context）。

QEMU使用KVM的用户态接口

- QEMU使用/dev/kvm与内核态的KVM通信

- 使用ioctl向KVM传递不同的命令：CREATE_VM, CREATE_VCPU, KVM_RUN等

```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
while (true) {
    ioctl(KVM_RUN)                                     Invoke VMENTRY
    exit_reason = get_exit_reason();
    switch (exit_reason) {
        case KVM_EXIT_IO: /* ... */
        break;
        case KVM_EXIT_MMIO: /* ... */
        break;
    }
}
```

这个例子就是我们创建 VM, v-CPU, 设置好它的新的寄存器。最后调用 KVM_VCPU_RUN, 这个调用会被 KVM 截获住，然后它会做相应的准备操作、准备好环境，然后在内核中调用真正的 VM_entry。

ioctl(KVM_RUN)时发生了什么

- x86中

- KVM加载vCPU对应的VMCS, VMLAUNCH/VMRESUME进入Non-root模式

- ARM中

- KVM主动加载VCPUs对应的所有状态，使用eret指令进入虚拟机，开始执行

当虚拟机下陷的时候，它第一步依旧是下陷到 KVM 中，KVM 里把状态保存好放到 struct 里，然后交给用户态的 QEMU 进行相应的处理。

所以在上例中，ioctl(KVM_RUN)其实是一个阻塞的 syscall。一旦 ioctl 恢复下一行执行了，说明 KVM 把虚拟机跑完了，遇到了一个 VM_EXIT，这个需要交给我们去处理。所以下一行它就根据 exit_reason (下陷的原因)，QEMU 再做不同的处理。

所以 QEMU 就是这样的逻辑去使用 KVM 去跑相应的虚拟机。

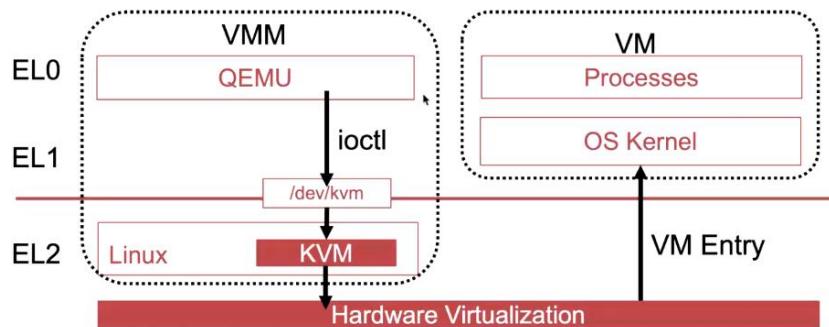


图 3 QEMU 和 KVM 相互配合

在上图中，QEMU 跑在 EL0，调用了 ioctl 进入 KVM 做相应的处理（如：加载寄存器），然后进入到虚拟机。虚拟机在运行过程中，下陷又回到 KVM，再回到 QEMU 中，根据 exit_reason 去做不同的处理。

QEMU/KVM的流程

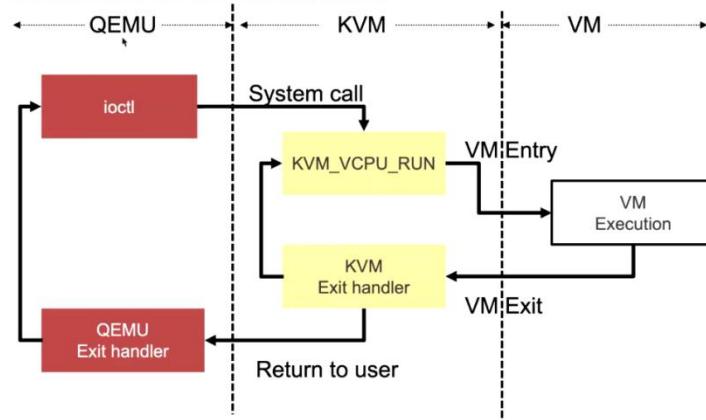


图 4 QEMU 和 KVM 的细粒度执行细节

中间是内核态 KVM，右边是 VM。进入到内核态的 KVM，使用虚拟化机制进入虚拟机。虚拟机下陷回到 KVM，KVM 里有一部分的 exit_handler，有一些 VM exit 是 KVM 可以自己处理的，比如调度 VM 的时候，这个过程可以自己处理不给 QEMU 看到。如果 KVM 遇到了一个自己不能处理的 exit_reason，它就交给用户态的 QEMU 去处理。

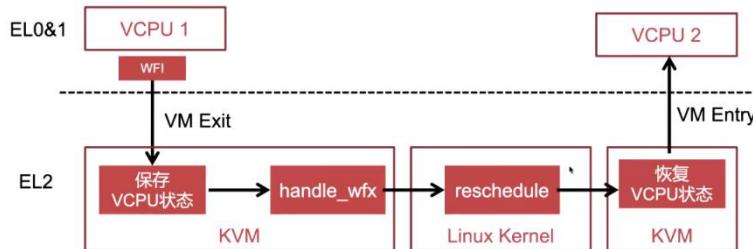
比如 VM 要发送网络包，KVM 不会自己处理，而是交给 QEMU 去处理。

Q: QEMU 作为一个用户态进程，应该怎么发网络包？

A: QEMU 可以利用丰富的 Linux syscall，比如创建一个 socket，在此之上建立 TCP/IP 连接，在 socket 接口里把包发出去。

这里有一个处理 VM 的 WFI 指令的例子。当虚拟机执行过程中，无事可做了，希望主动释放掉自己的时间片，它会下陷到 KVM 里，KVM 先保存状态，然后进到它自己内部的 wfx_handler 中，处理过程中，KVM 知道这个 VM 的时间片到了，那它就主动调用 Linux 的 reschedule 调度函数，它可以去调度另一个用户态线程或者另一个虚拟机。在这个例子中，KVM 会恢复另一个虚拟机的状态，进到另一个 v-CPU 中去跑。

例：WFI 指令 VM Exit 的处理流程



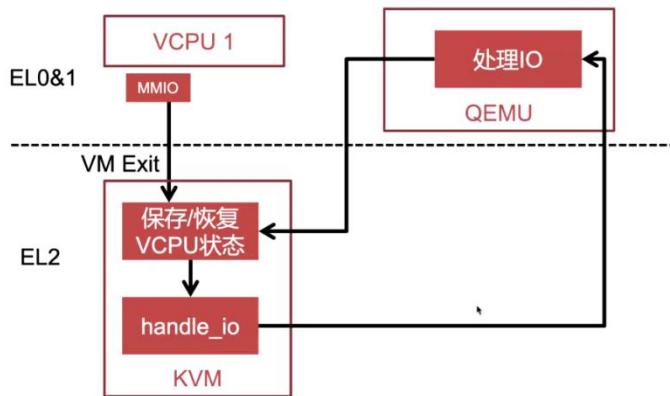
Q: Linux Kernel 在调度的时候需要知道自己在调度的是一个 VM、一个 v-CPU 或者是一个用户态进程吗？

A: 它不需要知道。它看到的都是各种各样的内核线程，只不过 v-CPU 的状态会保存在 task struct 里。当它要运行这个内核态线程的时候，它才会发现这是一个 KVM 相关的线程，去执行相应的功能。所以 Linux Kernel 的调度模块我们可以完全复用它。

当虚拟机执行到 MMIO 的时候，因为这是一个访问内存的指令，所以它也会下陷。KVM

保存状态后，自己的 `handle_io` 发现处理不了，就会把 `exit_reason` 转发给用户态的 QEMU。QEMU 调用 `ioctl` 回到 KVM 中，JVM 恢复 VM 的执行。

例：I/O 指令 VM Exit 的处理流程



以上就是所有 CPU 虚拟化的内容，接下来一个是内存虚拟化和 IO 虚拟化。

内存虚拟化

讲 CPU 虚拟化的时候，我们有意忽略了一个部分，就是 `memory` 怎么去处理。比如上例中，为什么访问 MMIO 区域的时候会下陷呢？是因为 KVM 有意地给它配置了一个页表，这个页是缺页，所以它在访问特定地址的时候会下陷，可以理解为是一个 `page fault`。所以特殊的 `page fault` 怎么处理，就是我们的内存虚拟化关心的内容了。

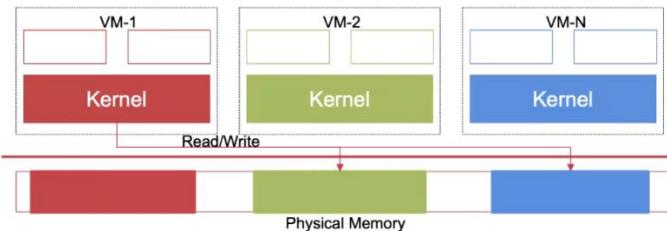
我们之前提到，OS 提供了一个虚拟内存的抽象，每个进程都以为自己看到了是 4G 连续内存。而在虚拟机里，有没有相应的变化呢？每个 OS 总觉得自己管理的是整个机器，看到的是机器上的所有内存，这个内存应该是从 0 开始增长的，它使用这个内存向上为各个进程提供虚拟内存的抽象。

Q：如果我们真的让 VM 使用了物理内存会发生什么事情呢？

A：下图就是一个例子，比如我们现在有 3 个虚拟机，每个虚拟机都觉得自己管理的是所有的内存，物理内存中的相同颜色分别对应了上面 VM 的真实的数据。如果 VM Kernel 能够看到真实的物理地址，它可以读别人的物理内存了。这显然是有问题的，一个虚拟机不应该看到别人的物理内存。包括说还会有正确性的问题，别人虚拟机修改了我的内存，会导致我的运行出现正确性和安全性（隐私数据被别的虚拟机读到）的问题。

为什么需要内存虚拟化？

- 操作系统内核直接管理物理内存
 - 物理地址从0开始连续增长
 - 向上层进程提供虚拟内存的抽象
- 如果VM使用的是真实物理地址



所以，以下是我们内存虚拟化的目标：

地址空间在虚拟机看来应该是完整的，它应该能访问任何地址。但是虽然它能访问所有范围的内存地址，但是它不应该读到别的虚拟机的内存信息。

内存虚拟化的目标

- 为虚拟机提供虚拟的物理地址空间
 - 物理地址从0开始连续增长
- 隔离不同虚拟机的物理地址空间
 - VM-1无法访问其他的内存

客户物理地址（GPA）

三种地址

- 客户虚拟地址(Guest Virtual Address, GVA)
 - 虚拟机内进程使用的虚拟地址
 - 客户物理地址(Guest Physical Address, GPA)
 - 虚拟机内使用的“假”物理地址
 - 主机物理地址(Host Physical Address, HPA)
 - 真实寻址的物理地址
 - GPA需要翻译成HPA才能访存
- } VMM管理

我们先介绍客户物理地址（Guest Physical Address, GPA）的概念。在没有虚拟化的时候只有两种地址：物理地址和虚拟地址。虚拟地址就是进程在运行的时候看到的地址，真实寻址的时候就是物理地址。引入了虚拟化之后，我们又新加了一个客户物理地址的概念。客户

物理地址是虚拟机看到的物理地址，是假的物理地址。

过去的虚拟地址叫做客户虚拟地址（GVA），真正用来寻址的我们叫做主机物理地址（HPA）。GVA 和 GPA 是虚拟机内部看得到的。GPA 到 HPA 的映射关系是由 hypervisor 管理的。其实排列组合以后还有一种地址叫做主机虚拟地址（Host Virtual Address, HVA），这个其实也是 hypervisor 用到的虚拟地址，这个地址也很重要，但是不会被客户机所看到。

怎么实现内存虚拟化？

- 1、影子页表(Shadow Page Table)
- 2、直接页表(Direct Page Table)
- 3、硬件虚拟化

内存虚拟化方案 1：影子页表

影子页表是什么意思呢？我们现在有三个地址，GPA, GVA 和 HPA。之前两个地址之间的映射需要页表，现在有三个地址了。这样我们就应该有两个页表去做映射。

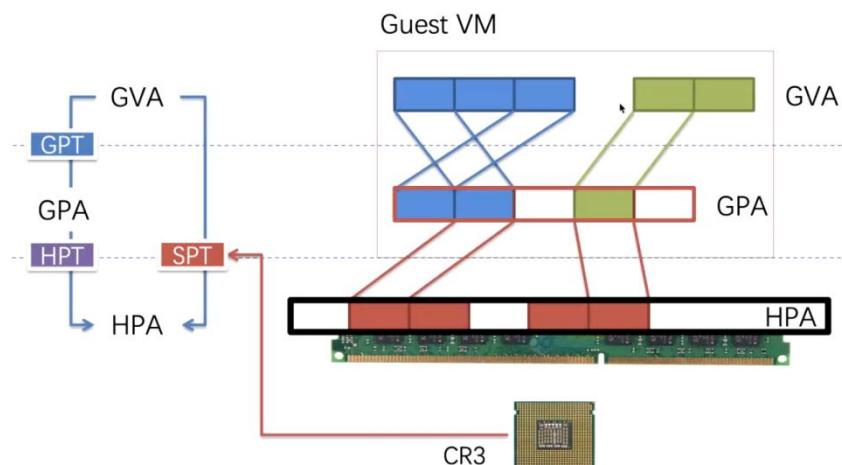
GVA 到 GPA 的映射应该是 Guest 内部的页表映射，这叫做 Guest page table。GPA 到 HPA 的映射是 hypervisor 看到的，所以 hypervisor 应该提供 host page table。

问题在于影子页表是一个很老的技术，主要用在 x86 上。而在 x86 架构中，只有一个基址寄存器 CR3，同时只有一个页表生效。此时就需要软件来解决硬件的缺陷。因为我们现在只能使用一个页表，从一个地址翻译到另一个地址。我们希望这个页表提供 GVA 到 HPA 的直接翻译。

Q: 为什么我们不用管 GPA 呢？

A: 因为在虚拟机运行之后，它大部分使用到的都是虚拟地址。所以我们只需要把 GVA 翻译成 HPA 即可。

1、影子页表



但是这件事情不太容易，我们需要把两个页表转成一个页表。

2个页表 -> 1个页表

1. VMM intercepts guest OS setting the virtual CR3
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest physical address is translated into host physical address
4. Finally, VMM loads the host physical address of the shadow page table

当我们配置 Guest 的 GPT 的时候，它自己就是配置从虚拟地址到物理地址的映射。它要装页表的时候，需要写 CR3，这其实是一个特权操作，这时候 hypervisor 就可以做 trap 和 emulate。然后我们再做相应的检查，下面就是检查的操作。

我们去遍历整个 GVA，我们去看看在 GPT 中 GVA 的映射是否是有效的。如果 PTE_P 这一位被设置成了 1，那么这个 entry 就是有效的，如果是 0 的话，就代表这个虚拟地址没有映射。

如果这个 entry 是有效的，我们就把 GPA 和对应的 HPA 得到。此处右移 12 位是因为，后面 12 位都是这个 PTE 的属性，并不属于真正的地址段。对于有效的映射，我们根据 GVA 得到了 HPA，我们就构造出了一个新的页表 shadow_page_table。它的索引就是 GVA，指向的物理地址就是 HPA。

Shadow Page Table 设置

```
set_cr3 (guest_page_table):
    for GVA in 0 to 220
        if guest_page_table[GVA] & PTE_P:
            GPA = guest_page_table[GVA] >> 12
            HPA = host_page_table[GPA] >> 12
            shadow_page_table[GVA] = (HPA<<12)|PTE_P
        else
            shadow_page_table[GVA] = 0
    CR3 = PHYSICAL_ADDR(shadow_page_table)
```

通过这样的一个遍历操作，我们构建好这个新的页表，并且把这个页表设置到 CR3 上。在这个操作之后，这个 CPU 上就只剩下了我们的影子页表了，然后我们再回到虚拟机去执行。它以为自己装的是 Guest Page Table，其实我们中间做了很多事情。

Q: 影子页表是保存在 Guest 内存里还是保存在 hypervisor 内存里？

A: 应该保存在 hypervisor 内存里，因为我们是 trap & emulate 实现的。Guest 只需要知道自己有一个 guest page table，它是不需要读到 shadow page table 里面的映射的。

Q: 如果 guest 读不到 shadow page 会不会有问题呢？

A: guest 修改页表的时候只能修改自己的 guest page table，但是只修改 guest page table 是没有用的。因为生效的 CR3 指向的 shadow page table。guest 在修改 guest page table 的时候应该通知 hypervisor，并且 hypervisor 需要把这个修改同步进影子页表里面，这样对页表的修改才会生效。

Q: 怎么让 guest 在修改 GPT 的时候，让 hypervisor 知道呢？

A: 一个很简单的思路就是 hypervisor 把 guest 对 GPT 的修改 trap 到，同步到影子页表里。但是，我们不能让 hypervisor 把 guest 的所有对内存的 store 指令都 trap 住，因为这个数量实在是太多了，如果每个 store 只变成很多下陷，那么这个虚拟机会慢到跑不起来。

实际上，我们只需要 trap guest 对 GPT 的修改，所有页表都是内存中的数据页，我们只需要 trap 所有对 GPT 所在的内存页的修改。这样我们就可以达到我们的修改效果了，我们可以把 GPT 所在的内存页变成 read-only，我们在 shadow_page_table 中把 GPT 所在的内存页的映射设置成 read-only。这样，guest 之后想修改我们 GPT 的时候就会触发一个 page fault 进到 hypervisor 中，hypervisor 根据 guest 想修改的内容，同步进 shadow page table 里去，然后再恢复虚拟机的执行。

Guest OS修改页表，如何生效？

- **Real hardware would start using the new page table's mappings**
 - Virtual machine monitor has a separate shadow page table
- **Goal:**
 - VMM needs to intercept when guest OS modifies page table, update shadow page table accordingly
- **Technique:**
 - Use the read/write bit in the PTE to mark those pages read-only
 - If guest OS tries to modify them, hardware triggers page fault
 - Page fault handled by VMM: update shadow page table & restart guest

Q: 如果 Guest 想读 CR3 需不需要下陷呢？不下陷会不会有问题？

A: 如果不下陷，那么 Guest 可以直接读到 shadow page table 的基址，它会发现设置的是 GPT，但是读到的是另一个地址。Guest 可能会有点混乱，所以当它读 CR3 的时候也应该下陷，hypervisor 返回一个 Guest 的 GPT 的基地址。

Q: 既然 Guest 配置的 GPT 最终生效的是一个 shadow page table，虚拟机内核是跑在用户态的，hypervisor 配置影子页表的时候，应该把所有虚拟机内核的映射变成用户态访问的映射。并且虚拟机中的用户进程所访问的页也应该是用户态可访问的。如果它们都在一个页表里面，就意味着内核和用户进程的隔离不复存在。

Guest内核如何与Guest应用隔离？

- **How do we selectively allow / deny access to kernel-only pages in guest PT?**
 - Hardware doesn't know about the virtual U/K bit
- **Idea:**
 - Generate two shadow page tables, one for U, one for K
 - When guest OS switches to U mode, VMM must invoke set_ptp(current, 0)

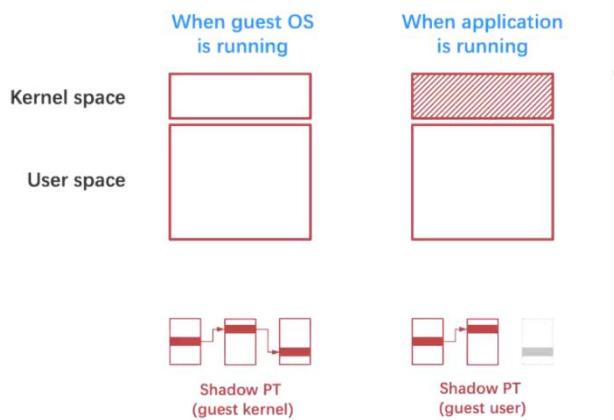
1个页表 -> 2个页表

在页表里面，每个 PTE 都会有一个属性，是 U 还是 K。如果是 K，那么 PTE 对应的页只能被内核态访问。如果是 U，内核和用户态都能访问。如果我们现在只有一个页表，它里面既有 VM 内核态又有 VM 用户态进程，shadow page table 给这些页配置的所有映射都应该是 U bit（用户态进程可以访问到虚拟机 Kernel，这是绝对不行的）。虚拟机应该保证 kernel 和用户态进程的隔离。

所以，我们应该产生 2 个 shadow page table，一个为虚拟机的内核服务，一个为虚拟机的用户态进程服务。

我们配置两个页表，左边这个页表是 guest kernel 去跑的时候，可以看到整个虚拟机中所有的内存。当它去跑用户态进程的时候，我们在新的用户态进程配置的页表里把 Kernel 里的映射全部挖掉，只留下用户态可以访问的范围。

Two Memory Views of Guest VM



以下为代码实现，如果 kmode 是 true，我们就把所有内存都加进去。如果 kmode 是 false，我们只加入 userspace 中的内存映射。

构建2个不同的Guest Pages

31	12 11 9 8 7 6 5 4 3 2 1 0
	Physical-Page Base Address AVL G P A D A P C W / U T S R / P

```

set_ptp(guest_pt, kmode):
    for gva in 0 .. 220:
        if guest_pt[gva] & PTE_P and
            (kmode or guest_pt[gva] & PTE_U):
            gpa = guest_pt[gva] >> 12
            hpa = host_pt[gpa] >> 12
            shadow_pt[gva] = (hpa << 12) | PTE_P | PTE_U
        else:
            shadow_pt[gva] = 0
    PTP = shadow_pt

```

Q：现在我们有两个页表了，我们什么时候决定用哪个页表呢？

A：我们应该在做虚拟机内核态切换到用户态的过程中，在 hypervisor 里帮它做页表的切换。

shadow page table 是非常精彩的技术，通过软件的方法解决了硬件的缺陷。但是现在已经几乎不用了。

内存虚拟化方案 2：直接页表（**Direct Paging**）

最后我们说一下 direct paging。影子页表保证了 guest 不知道它跑在虚拟化环境下。如果我们使用 PV 的思想，我们可以让 VM 知道自己跑在虚拟化下，我们直接提供接口给它配置页表。这就是 direct paging 的思想，由 guest 直接配置 GVA 到 HPA 的映射。我们要检查一下配置的页表的映射有没有问题，不能让 GVA 映射到另一个虚拟地址空间。

2、**Direct Paging (Para-virtualization)**

- **Modify the guest OS**
 - No GPA is needed, just GVA and HPA
 - Guest OS directly manages its HPA space
 - Use *hypercall* to let the VMM update the page table
 - The hardware CR3 will point to guest page table
- **VMM will check all the page table operations**
 - The guest page tables are read-only to the guest

好处就是容易实现、性能很好（修改 GPT 减少 trap，可以做 write batch cache）。

坏处就是透明性差，需要不同的内核做修改，并且 guest 知道太多的信息了。HPA 可以帮助 guest 做 rowhammer 攻击。

2、**Direct Paging (Para-virtualization)**

- **Positive**
 - Easy to implement and more clear architecture
 - Better performance: guest can batch to reduce trap
- **Negatives**
 - Not transparent to the guest OS
 - The guest now knows much info, e.g., HPA
 - May use such info to trigger *rowhammer* attacks

2022/5/10

我们今天继续讲系统虚拟化剩下的内容，今天主要内容是剩下的内存虚拟化和 IO 虚拟化的知识。

上节课讲到 v-CPU 演进的 7 个版本，我们通过不断增加虚拟机的功能。比如我们增加时钟中断处理、用户态线程、多个用户态线程，到最后怎么在一个多物理 CPU 上运行虚拟机，

下图展示的是最终版本。

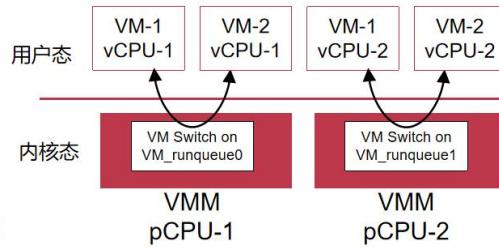
Review: 虚拟机支持多个虚拟CPU

- VM的能力（与第六版的区别）

- 虚拟机有多个Virtual CPU (vCPU)

- VMM的实现

- 在VM_runqueue中标记出VM和vCPU的类型



在 hypervisor 看来，每个 v-CPU 其实就是在 hypervisor 空间中创建一个对应 v-CPU 的一个 context，这样它能够保证每次下陷的时候，可以把当前下陷的 v-CPU 的所有寄存器状态都保存在这个 context，保存完之后，hypervisor 就可以根据自己的调度策略切换到别的 v-CPU。这和之前的内核线程的调度是很像的。v-CPU 机制其实并不是从零实现的，它在内核中复用了内核切换的机制。

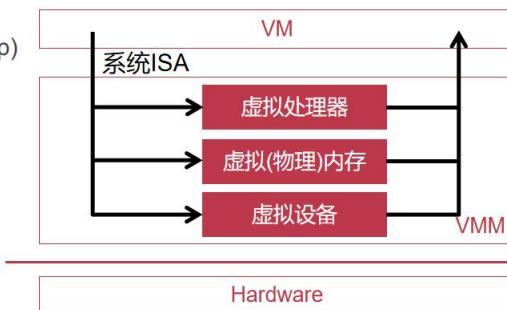
Review: 系统虚拟化的流程

- 第一步

- 捕捉所有系统ISA并陷入(Trap)

- 第二步

- 由具体指令实现相应虚拟化
 - 控制虚拟处理器行为
 - 控制虚拟内存行为
 - 控制虚拟设备行为



- 第三步

- 回到虚拟机继续执行

但是我们假设所有系统 ISA 操作都会造成下陷，ARM 和 x86 都没有这种保证，某些系统 ISA 不能下陷。为了解决这个问题，有四种方法，它们都可以解决有些系统调用不可下陷的缺陷，目前主流的是硬件虚拟化。

Review: 如何处理这些不会下陷的敏感指令？

处理这些不会下陷的敏感指令，使得虚拟机中的操作系统能够运行在用户态 (EL-0)

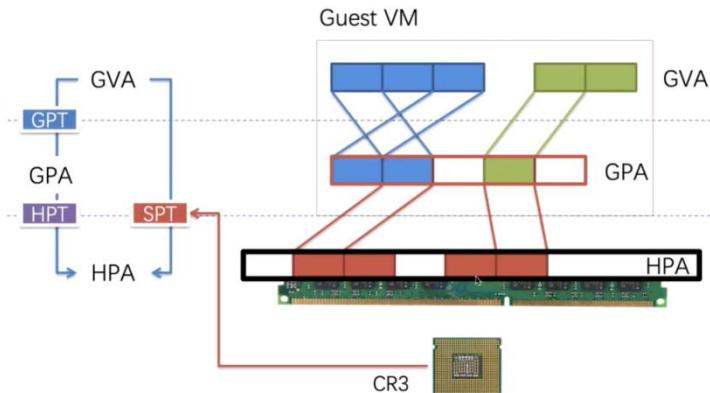
- 方法1：解释执行
- 方法2：二进制翻译
- 方法3：半虚拟化
- 方法4：硬件虚拟化（改硬件）

CPU 虚拟化主要是解决不可下陷的指令所带来的缺陷。

Q: 为什么需要内存虚拟化？

A: 因为我们希望虚拟机看到一个从零开始看似完整的物理地址空间，并且我们要保证每个虚拟机之间的物理地址是相互隔离的，不能让一个虚拟机通过直接修改物理内存而修改了其他虚拟机的数据。

Review : 影子页表



但是在过去的硬件环境中，这是很难做的。因为虚拟机内部维护的是 GVA 到 GPA 的映射，而 GPA 不是一个真实的物理寻址地址，真实可寻址的物理地址是 HPA，所以我们 hypervisor 做第二层的 GPA 到 HPA 的翻译。问题就是我们只有一个页标记地址寄存器 CR3。最后我们使用的是 shadow page table，虽然 VM 依然维护的是 GPT (GVA 到 GPA) 的映射，我们在设置 CR3 的时候陷入到 hypervisor，它会把 GVA 直接翻译到 HPA。这个页表是对 guest 透明的，但是 guest 每次对页表修改的时候，hypervisor 都可以把修改同步到影子页表上。

为了捕捉到 guest 对页表的修改，我们必须把 guest 页表所在的页表页设置为 read-only，这样每次 guest 尝试修改 GPT 的时候，就可以下陷到 hypervisor 中做同步。

所以，我们发现影子页表在实现的过程中，因为我们要同步对 GPT 的修改，所以是比较复杂的。为了实现用户态和内核态的隔离，它必须分别实现一个页表。

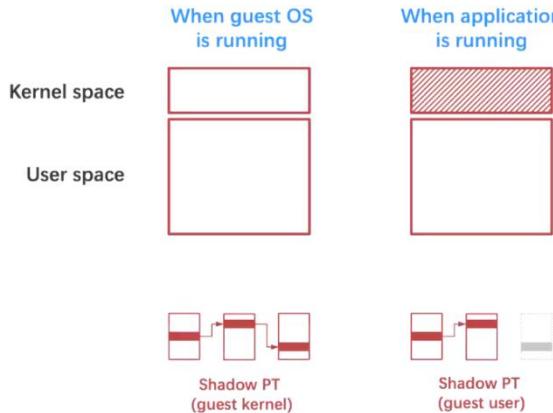
Q: 我们上周发布了一个作业，现在我们有 1 个虚拟机，它有 4 个 v-CPU，里面有 10 个进程，这时候应该为这个虚拟机创建多少个影子页表呢？

A: 根据影子页表的操作，guest 每次想装一个页表，就会下陷到 hypervisor，hypervisor 看到这是一个新的页表之后就会为它生成一个新的影子页表，从这个操作来看，如果有 10 个进程，理论上应该有 10 个不同的影子页表。但是我们要维持虚拟机内核和虚拟机用户态进程之间的隔离，所以我们必须为内核和用户态分别创建一个页表，所以 10 个影子页表不能维持这样的隔离，我们需要为虚拟机内核单独维护 1 个影子页表，所以在这个场景下应该是 11 个页表。

Q: 11 个页表足够吗？当 Kernel 态想访问用户态内存的时候应该怎么做呢？

A: 在 VMM 访问的时候，我们动态地把用户影子页表中的内存加载过来即可。

Two Memory Views of Guest VM



接下来是半虚拟化，这个我们之前在 CPU 处理下陷指令的时候也提到过。我们把 OS 和 hypervisor 协同设计，直接告诉 Guest OS 是跑在虚拟化环境中的。Guest OS 可以通过调用特殊的 hypercall 去处理对应的问题。Direct Paging 就是半虚拟化在内存虚拟化中的体现。我们提供新的 hypercall 接口，可以通过调用这个接口直接对页表做修改，好处就是我们不再是以 PTE 这个粒度去做修改（因为影子页表每次 Guest 对 page table entry 做修改都会下陷），在这里我们可以修改了十几个 PTE 之后再统一调用 PTE 告诉 hypervisor。这样就可以把修改的粒度变粗，大大减少下陷的次数、提高性能。

但是 hypervisor 还是要保证虚拟机之间的隔离，所以 Guest 调用 hypercall 想对页表做修改的时候，hypervisor 依旧会对页表修改请求做检查，检查 VM 是不是想把虚拟地址映射到别的 VM 的物理地址上，检查无误后 hypervisor 才会把页表做装上。

这个方法的坏处就是，我们需要对 Guest OS 做修改，在过去要对 Linux 的每个版本做适配，现在 Linux 已经把对半虚拟化的支持 merge 到 upstream 里去了。

Review : Direct Paging (Para-virtualization)

- **Modify the guest OS**
 - No GPA is needed, just GVA and HPA
 - Guest OS directly manages its HPA space
 - Use *hypercall* to let the VMM update the page table
 - The hardware CR3 will point to guest page table
- **VMM will check all the page table operations**
 - The guest page tables are read-only to the guest*

内存虚拟化方案 3：硬件虚拟化

最后一个内存虚拟化技术是上节课没来得及介绍的：硬件虚拟化。在我们之前的两种方法中，无非原因就是硬件不行，硬件只能支持一个页表的情况下我们需要用到影子页表或者

直接对 Guest OS 做修改。

实际上我们改硬件就可以把这些问题都解决了。虚拟机内部，VM OS 为了为它的进程维持虚拟地址空间的抽象，本来就为每个进程配置页表，而底下 hypervisor 为了完成 GPA 翻译成 HPA，也需要一个页表。那么既然需要两个页表，硬件加两个页表就解决问题了。

所以，从 Intel VT-x 到 ARM，都是在硬件中加上两阶段的页表。第一阶段页表（EPT）就是完成从 GVA 到 GPA 的映射和翻译，翻译完之后，hypervisor 的第二阶段页表会进入工作阶段，把 GPA 直接由 MMU 翻译成 HPA。

新的 hypervisor 控制的第二阶段页表在虚拟机配置的时候就会生成，把 GPA 翻译成 HPA 之后，它会装到新的页表基址寄存器中，在运行虚拟机的时候，MMU 会自动地完成 GPA 到 HPA 的映射。

硬件虚拟化对内存翻译的支持

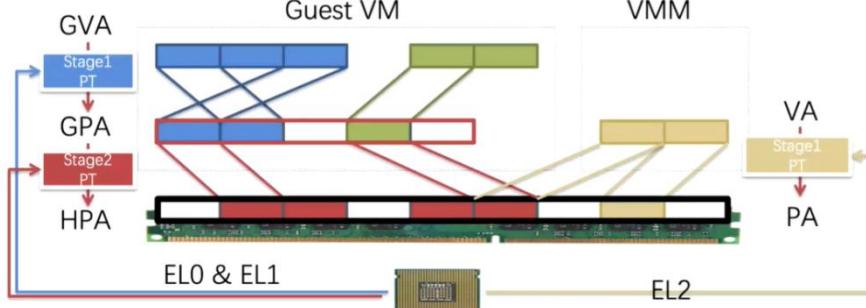
- Intel VT-x 和 ARM 硬件虚拟化都有对应的内存虚拟化
 - Intel Extended Page Table (EPT)
 - ARM Stage-2 Page Table (第二阶段页表)
- 新的页表
 - 将 GPA 翻译成 HPA
 - 此表被 VMM 直接控制
 - 每一个 VM 有一个对应的页表

第二阶段页表

我们可以看一下下图，本来之后一个页表现在我们有两个页表了。Stage1 PT 是完全由 Guest 维护的，不需要 trap 到 hypervisor 中。hypervisor 需要配置 Stage2 PT，在所有系统运行之前，hypervisor 应该把这个页表装上。在访问虚拟地址的时候，硬件会自动地判断待翻译的地址是不是一个 GVA，如果是 GVA 就走两阶段翻译 GVA->GPA->HPA。

第二阶段页表

- 第一阶段页表：虚拟机内虚拟地址翻译 (GVA->GPA)
- 第二阶段页表：虚拟机客户物理地址翻译 (GPA->HPA)

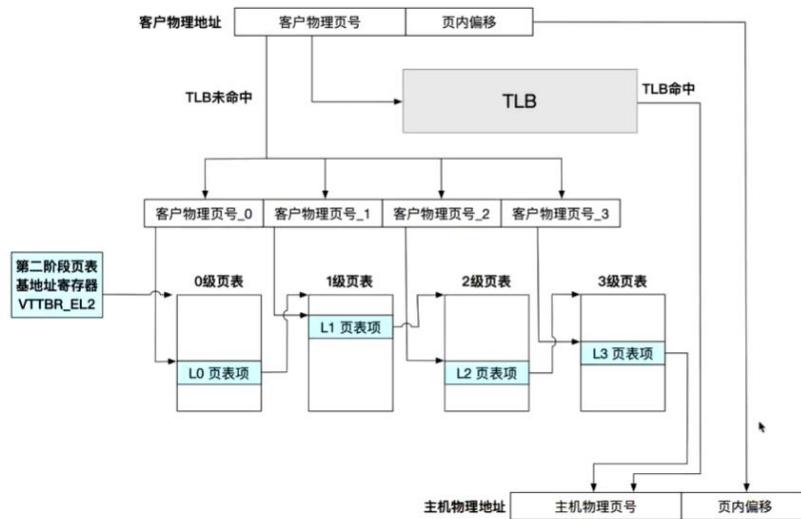


Q: 如果我们不起虚拟机的话，hypervisor 该怎么办呢？

A: hypervisor 也是可以自己独立地配置 State1 PT 和 Stage2 PT 的。我们关闭虚拟化的时候，hypervisor 也是可以像过去一样创建自己的进程。这就是我们期待的模型。

以下就是第二阶段页表的结构，它和第一阶段是一样的，也是 4 级页表。翻译的过程也是一样的，把 GPA 不同位置的偏移量拿出来分别用做 4 级页表不同的 index，逐级下找。最终得到一个我们想要的 HPA。

第二阶段4级页表



所以，在 TLB 完全 miss 的时候，我们有了第二阶段以后，又加入了四级页表。

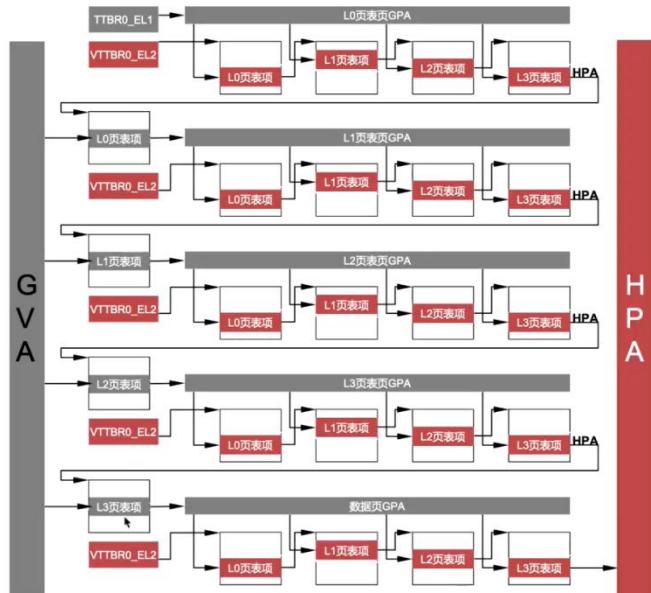
Q: 那么在 TLB 全部 miss 的时候，一个 GVA 翻译到 HPA 的时候，需要经过多少次的内存访问呢？是 4+4 还是 4*4？

A: 实际上是 24 次内存访问。首先我们需要知道 Guest 在给它虚拟机内的进程配置第一阶段页表的时候，第一阶段页表的基地址是 GPA。GPA 是不能用来真正访存的，所以 GPA 要经过第二阶段的翻译。所以第一阶段页表翻译的时候，中间遇到的所有 GPA，都需要经过一次底下的第二阶段页表的翻译。

我们现在虚拟机内部进程想要访问一个指针，这个指针是一个 GVA。MMU 发现我们先要访问一个 GVA，我们必须先翻译成 GPA，所以它去找 ttbr0_el1 (第一阶段页表基地址) 得到第一阶段 L0 页表的 GPA，但是 GPA 不能直接访存，需要翻译成 HPA，所以我们要拿第一阶段 L0 页表的 GPA 依次走完第二阶段页表的 4 次访存，最终得到 ttbr0_el1 存储的 HPA。然后，我们继续根据 L0 页表项得到第一阶段 L1 页表的 GPA 基地址，以此类推。

翻译过程

- 总共24次内存访问
 - 为什么?
 - 25-1



我们根据上图，数一下一共有多少次内存访问。首先有 20 次第二阶段页表的逐级访存操作。然后每次我们拿到了一个第一阶段 PTE 的 HPA 之后，我们需要对这个位置访存才能拿出第一阶段的下一级页表。

Stage2 的页表其实是存在一个新的页表基地址寄存器里的。在虚拟机启动之前 hypervisor 需要把这个装上去。

VTTBR_EL2

- 存储虚拟机第二阶段页表基地址
 - 只有1个寄存器：VTTBR_EL2
- 对比第一阶段页表
 - 有2个页表基地址寄存器：TTBR0_EL1、TTBR1_EL1
- VMM在调度VM之前需要在VTTBR_EL2中写入此VM的第二阶段页表基地址
- 第二阶段页表使能
 - HCR_EL2第0位

Q：我们之前提到，如果我们要使用影子页表，需要对每个 Guest 使用的影子页表建立一个对应的影子页表。有了硬件虚拟化之后，1 个虚拟机包含 4 个 v-CPU，里面有 10 个进程，我们需要配置多少个第二阶段页表呢？

A：1 个，因为我们是在整个虚拟机范围内控制的，我们不关心里面有几个页表。硬件只需要保证每个 Guest 里的 GPA 能够被翻译即可。这和 v-CPU 无关，v-CPU 看到的物理地址是一致的。但这是理论上的模型，我们在使用的时候也可以为每个 v-CPU 配置一个第二阶段页表。

在没有 TLB 的存在下，我们需要 24 次内存访问。页表可以把第一阶段的翻译结果和第二阶段的翻译结果存储在 TLB 里，大部分时间内我们是不需要承受 24 次内存访问的。只有当我们虚拟机中的 workload 的随机访存范围特别大的时候，TLB 不能装载 GPA 到 HPA 的映射了，就会有 TLB miss。

TLB：缓存地址翻译结果

- 回顾：TLB不仅可以缓存第一阶段地址翻译结果
- TLB也可以第二阶段地址翻译后的结果
 - 包括第一阶段的翻译结果(GVA->GPA)
 - 包括第二阶段的翻译结果(GPA->HPA)
 - 大大提升GVA->HPA的翻译性能：不需要24次内存访问
- 切换VTTBR_EL2时
 - 理论上应将前一个VM的TLB项全部刷掉

理论上在 VM 切换的时候，我们应该把 TLB 里全部刷掉。但实际上硬件里会配置和虚拟机相关的 VM tag，这样在切换页表的时候就不需要把主动地把 TLB 中的页表项刷掉。

TLB刷新

- 刷TLB相关指令
 - 清空全部
 - TLBI VMALLS12E1IS
 - 清空指定GVA
 - TLBI VAE1IS
 - 清空指定GPA
 - TLBI IPAS2E1IS
- VMID (Virtual Machine IDentifier)
 - VMM为不同进程分配8/16 VMID，将VMID填写在VTTBR_EL2的高8/16位
 - VMID位数由VTCR_EL2的第19位 (VS位) 决定
 - 避免刷新上个VM的TLB

我们之前提到，在使用影子页表的时候，如果它想要改第一阶段页表的时候，会触发下陷，由 hypervisor 来进行同步。而我们现在有了硬件虚拟化之后，缺页异常应该怎么处理呢？

因为 VM 有自己的页表，所以 VM 内部的所有 page fault 应该由 VM 自己处理，而第二阶段页表的缺页才应该由 hypervisor 介入，这个过程应该是对 Guest 透明的。

如何处理缺页异常

- 两阶段翻译的缺页异常分开处理
- 第一阶段缺页异常
 - 直接调用VM的Page fault handler
 - 修改第一阶段页表不会引起任何虚拟机下陷
- 第二阶段缺页异常
 - 虚拟机下陷，直接调用VMM的Page fault handler

我们不再需要维护影子页表，也不需要捕捉每次 GPT 的更新，也不需要像直接页表一样打破 Guest 的透明性。并且我们不需要为每个 VM 进程维护一个影子页表，降低了内存开

销。

第二阶段页表的优缺点

- **优点**

- VMM实现简单
- 不需要捕捉Guest Page Table的更新
- 减少内存开销：每个VM对应一个页表

- **缺点**

- TLB miss时性能开销较大

有了第二阶段页表之后，hypervisor 可以在 GPA 到 HPA 这里做很多之前在 Guest 内部所做的事情。比如现在我们现在物理内存 4G，里面跑了两个虚拟机各自用到了 3G 内存。这样为虚拟机提供的内存就有 6G，那么怎么样才能做到内存的超售呢？

内存超售的很重要的机制就是内存的换页（swapping），也就是把虚拟机暂时用不到的内存的页存到磁盘上去。

我们看看如下的例子：

我们每个虚拟机中有 2 个进程和 1 个内核。在没有虚拟化的时候，内核会使用 LRU（最近最少使用）内存保存到磁盘上去，然后把页表项的保留位设置为 0。虚拟机换页就是 hypervisor 把页表中的 128M 拿出来保存到磁盘上，然后把对应的 PTE 的保留位设置为 0。

Q：是 hypervisor 做 LRU 好还是 Guest Kernel 做 LRU 好？

A：虚拟机内核看到的信息是更多的，它知道哪些页是给哪个进程用的，哪些页是没有用的，它清楚内存页更多的语义。而 hypervisor 不知道这些，它只维护页表，不知道 VM 里有多少进程，也不知道页是属于进程 1、进程 2 的。

如何实现虚拟机级别的内存换页？

- **具体场景：将虚拟机A的128MB内存转移到虚拟机B中**

- 虚拟机A对内存的使用较少
- 虚拟机B对内存需求较大

- **问题**

- VMM无法识别虚拟机内存的语义
- 两层内存换页机制
- VM与VMM的换页机制可能彼此冲突，造成开销。

这两个换页机制可能会造成冲突，比如 VM-A 已经根据 hypervisor 的换页机制把一部分内存给了 VM-B 了，此时数据在 Guest 中已经是空的了。但是 VM-A 不知道自己页被换掉了，可能会访问这个页，触发了 Stage2 Page Fault，hypervisor 需要把刚刚 VM-A 换到磁盘上的页重新拿进内存加上映射还给 VM-A。假设 VM-A 拿到这个页是为了做自己的换页机制，拿这个页是为了读到它的内容存到 VM-A 的虚拟磁盘上去。我们就会发现这个过程中有很多冗余

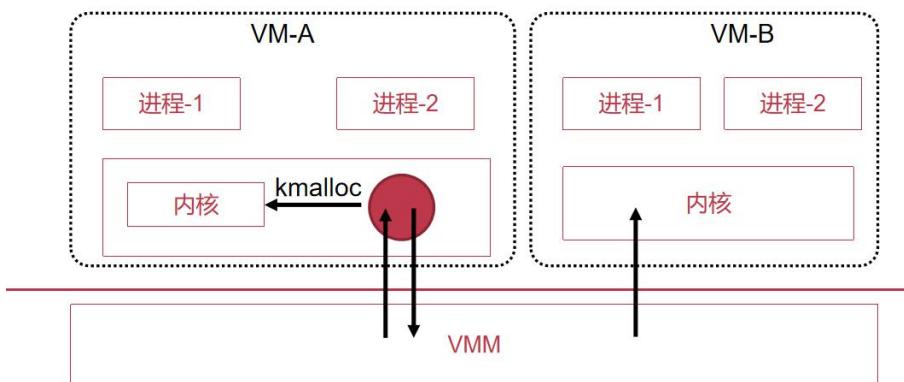
的操作，这是两个换页机制冲突所带来的问题。

内存气球机制

解决这个问题就需要内存的气球机制。我们给 Guest 内核装一个驱动。等到 hypervisor 决定把 VM-A 的一部分内存交给 VM-B 的时候，hypervisor 就可以通过一个 upcall 调用这个驱动。这个驱动可以调用虚拟机内核的 kmalloc 接口，拿到 128M 的空间。拿到之后，这个驱动把这 128M 的地址直接交还给 hypervisor，然后这些页被置空，然后映射给 VM-B。

这个过程是不需要存磁盘的，因为这 128M 是这个驱动调用 kmalloc 分配出来的，里面的数据是没有用的。hypervisor 就可以省掉写磁盘的操作。但是 VM-A 为了分配出这 128M，可能变相地去做 swap 机制，去把它内存中的 LRU 页放到虚拟磁盘上。当 VM-A 又需要使用 128M 的时候，我们又可以把这个还给 VM-A 使用，也就是让内存中的气球变小。

内存气球机制



自此我们就介绍完了内存虚拟化。

I/O 虚拟化

IO 虚拟化和之前的方法大同小异。MMIO 访问一段特定的地址范围，可以被翻译到对设备的访问控制。数据交换的时候我们需要借助的 DMA 异步通讯机制。OS 通过这三种手段和外部设备交互。

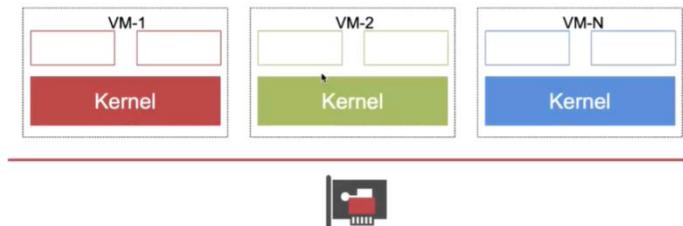
为什么需要IO虚拟化

- 回顾：操作系统内核直接管理外部设备
 - PIO/MMIO
 - DMA
 - Interrupt
- 如果VM能直接管理物理设备
 - 会发生什么？

我们假设一个物理机上只有一个网卡，这会导致所有虚拟机都有相同的 MAC 地址。并且虚拟机可以直接读到别人的网络包，这也是一个问题。

如果VM直接管理物理网卡

- 正确性问题：所有VM都直接访问网卡
 - 所有VM都有相同的MAC地址、IP地址，无法正常收发网络包
- 安全性问题：恶意VM可以直接读取其他VM的数据
 - 除了直接读取所有网络包，还可能通过DMA访问其他内存



如果我们所有虚拟机共享的同一个物理磁盘，并且可以控制物理磁盘。那么如果一个虚拟机告诉磁盘要格式化整个磁盘，就会导致非常严重的问题。所以我们不能让虚拟机直接管理物理设备。这样就可以实现虚拟机和物理设备的隔离。

比如每个 VM 可以平分带宽，从而提高资源利用率。

I/O虚拟化的目标

- 为虚拟机提供虚拟的外部设备
 - 虚拟机正常使用设备
- 隔离不同虚拟机对外部设备的直接访问
 - 实现I/O数据流和控制流的隔离
- 提高物理设备的利用资源
 - 多个VM同时使用，可以提高物理设备的资源利用率

类似地，也有三种方法。

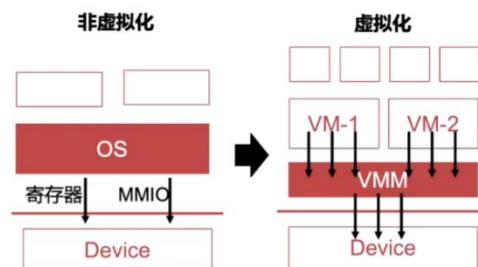
怎么实现I/O虚拟化？

- 1、设备模拟 (Emulation)
- 2、半虚拟化方式 (Para-virtualization)
- 3、设备直通 (Pass-through)

设备模拟的意思就是全部通过软件的行为去模拟出来假的硬件设备。我们需要用软件的方法虚拟化 PIO/MMIO、DMA、interrupt。这个其实也就是 trap & emulate，转化为对物理设备的操作，我们就可以在 VMM 中对多个 VM 做服务。

方法1：设备模拟

- OS与设备交互的硬件接口
 - 模拟寄存器(中断等)
 - 捕捉MMIO操作



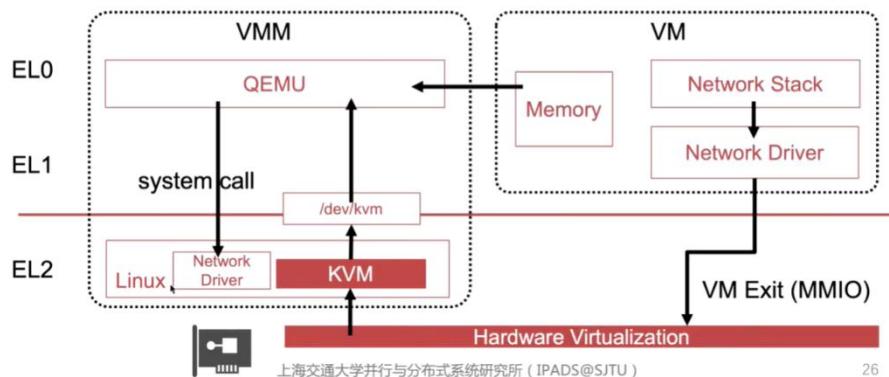
- 硬件虚拟化的方式
 - 硬件虚拟化捕捉PIO指令
 - MMIO对应内存在第二阶段页表中设置为invalid

我们继续以 QEMU/KVM 举例，最初的 QEMU 是通过软件来模拟，提供了大量的设备模拟。虚拟机 OS 在启动的过程中会有 prob 过程发现其中的设备，QEMU 可以在启动的时候告诉 Guest OS 支持哪些设备。网卡里一个很主流的网卡叫 E1000，这边假设的就是 QEMU 配合 KVM 捕捉到 VM 对虚拟网卡的访问。

Guest 内部有网络协议栈，决定通过驱动发出去，交给 VM 网卡驱动。网卡驱动先把 DMA 配置好，把包放在一个内存地址中，然后触发 MMIO，因为是 page fault 就会触发下陷进入到 KVM 中，KVM 发现这是一次 MMIO 的下陷，它不知道应该怎么做，因为这是 QEMU 主动配置的，所以 KVM 把转发过程进一步转发给 QEMU。QEMU 根据 exit_reason 发现是 MMIO，然后再查自己的表，发现是 VM 要做 DMA 了，所以 QEMU 直接读虚拟机内存，把网卡驱动配置的内存拿出来，QEMU 拿到这个网络包之后，再调用 syscall 把网络包发出去。

例：QEMU/KVM设备模拟1

- 以虚拟网卡举例——发包过程



26

Q: Guest 内部是有一套网络协议栈的，现在传输到 QEMU 之后又要调用 `syscall` 经过层层打包再发出去，是不是会对性能有些影响？

A: socket 支持 raw socket，可以不经过网络协议栈，直接把 QEMU 的网络协议包打包发出去。但 QEMU 需要一个特殊的权限。

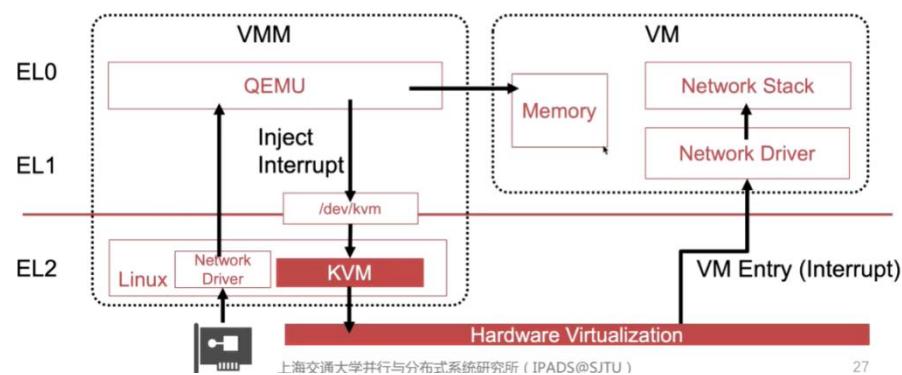
Q: 为什么 QEMU 可以直接读到虚拟机的内存呢？

A: 实际上虚拟机就是 QEMU 创建出来的，虚拟机的所有内存都是 QEMU 调用 `mmap` 从内核中拿到的。对于 QEMU 而已，它知道 GPA 和 HVA 的对应关系，因为地址都是 QEMU 告诉 KVM 的。

收包的时候，QEMU 先把网络包扔掉虚拟机的内存里，但是此时虚拟机不知道。所以 QEMU 需要继续给 VM 插入一个中断。虚拟机看到有一个 DMA 的中断就知道是网络包来了。

例：QEMU/KVM设备模拟2

- 以虚拟网卡举例——收包过程



27

所有的包是先到物理机的网络驱动，然后到用户态，然后 QEMU 把网络包复制到内存里。我们发现这个路径很长，希望做一些优化。我们可以在 QEMU 处做防火墙。

设备模拟的优缺点

- **优点**

- 可以模拟任意设备
 - 选择流行设备，支持较“久远”的OS（如e1000网卡）
- 允许在中间拦截（Interposition）：
 - 例如在QEMU层面检查网络内容
- 不需要硬件修改

- **缺点**

- 性能不佳

我们可以继续打破抽象，让虚拟机知道自己运行在虚拟化的环境中。我们可以把IO设备拆解为前端驱动和后端驱动。这个方案和一般设备驱动不一样，它是知道自己运行在虚拟化环境中了，所以我们可以不使用MMIO和DMA这种抽象。

方法2：半虚拟化方式

- **协同设计**

- 虚拟机“知道”自己运行在虚拟化环境
- 虚拟机内运行前端(front-end)驱动
- VMM内运行后端(back-end)驱动

- **VMM主动提供Hypercall给VM**

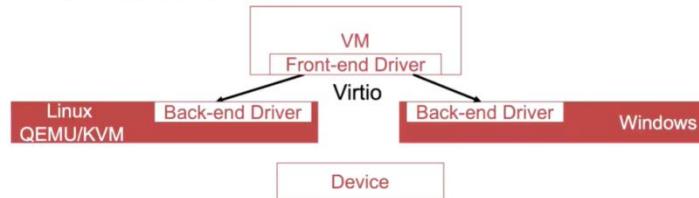
- **通过共享内存传递指令和命令**

基于这样的想法，现在最主流的是VirtIO的框架。它已经形成了一套标准，对不同的设备怎么定义前端和后端接口。一个虚拟机植入了前端以后，只要遵循这样的标准，可以在Linux QEMU/KVM上跑，也可以在Windows上跑。我们也希望把Back-end搬到Device上，比如现在的智能网卡等，不需要在使用hypervisor去模拟了。

VirtIO: Unified Para-virtualized I/O

- **标准化的半虚拟化I/O框架**

- 通用的前端抽象
- 标准化接口
- 增加代码的跨平台重用



这个标准里最重要的就是 Virtqueue，是 VM 和 hypervisor 之间传递 IO 请求的队列。这个队列里的每个 entry、field 里都是有定义的。

Virtqueue

- **VM和VMM之间传递I/O请求的队列**

- **3个部分**

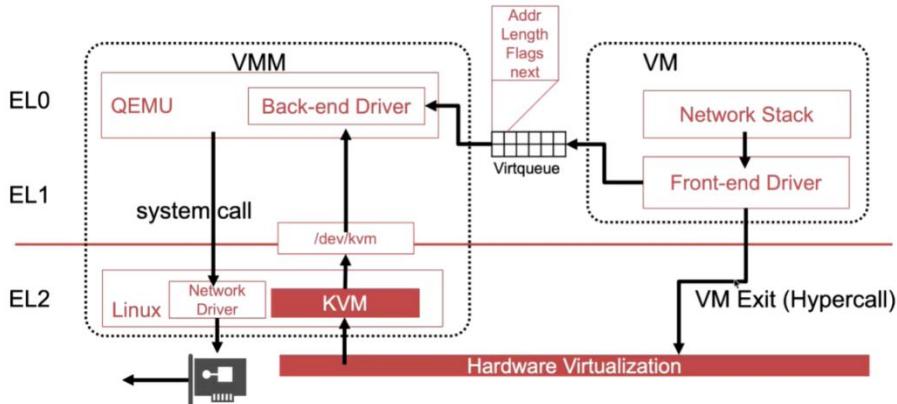
- Descriptor Table
 - 其中每一个descriptor描述了前后端共享的内存
 - 链表组织
- Available Ring
 - 可用descriptor的索引，Ring Entry指向一个descriptor链表
- Used Ring
 - 已用descriptor的索引

网络包现在只需要放到 Guest 内存里，把对应的地址、length、flag 放到 Virtqueue 中，然后我们主动调用 hypercall，触发下陷并且 KVM 转发到 QEMU 中。QEMU 中的 back-end driver 直接查询 Virtqueue，把 next 连接的网络包拿出来。然后调用 syscall 发出去。

Q: 这个过程好像和之前没什么区别。为什么说半虚拟化好呢？

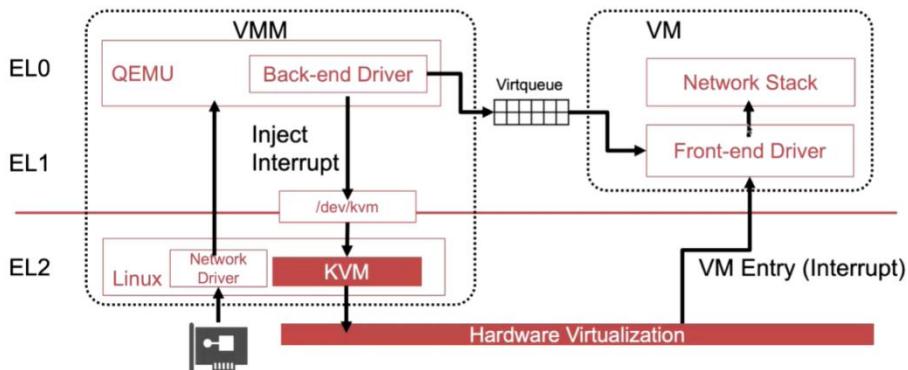
A: 好在 virtqueue 的设计。软件模拟的方法非常细粒度，DMA 可能是 128 Byte。但我们现在可以主动地把包做一个串联。可以大大把过去十几次的软件模拟操作合成一次。

例：QEMU/KVM半虚拟化：发网络包



收包也是类似的。

例：QEMU/KVM半虚拟化：收网络包



一开始网络包在 Linux Kernel 里，为什么 KVM 不能直接把包拿过来给 VM 呢？所以就提出了一个 V-Host 的概念，也就是我们不需要把后端驱动实现在 QEMU 里，我们可以实现在 KVM 里，每个网络包直接进到 back-end driver 中。

半虚拟化方式的优缺点

• 优点

- 性能优越
 - 多个MMIO/PIO指令可以整合成一次Hypercall
- VMM实现简单，不再需要理解物理设备接口

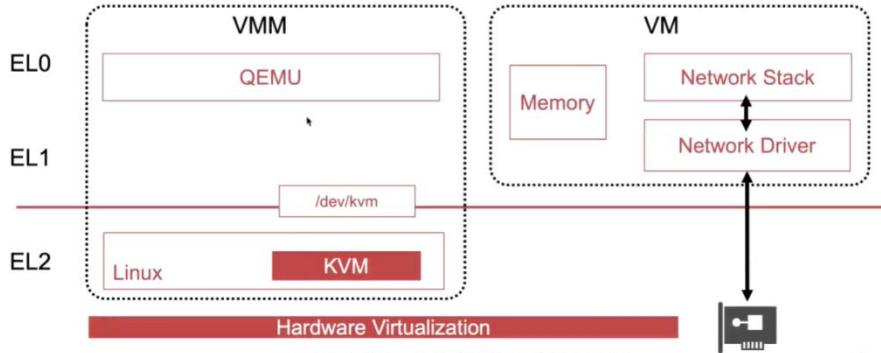
• 缺点

- 需要修改虚拟机操作系统内核

我们可以让虚拟机直接看到网卡。hypervisor 把网卡的 MMIO 地址范围直接映射给虚拟机，这样在访问的时候就不会下陷了。性能接近没有虚拟化的性能。

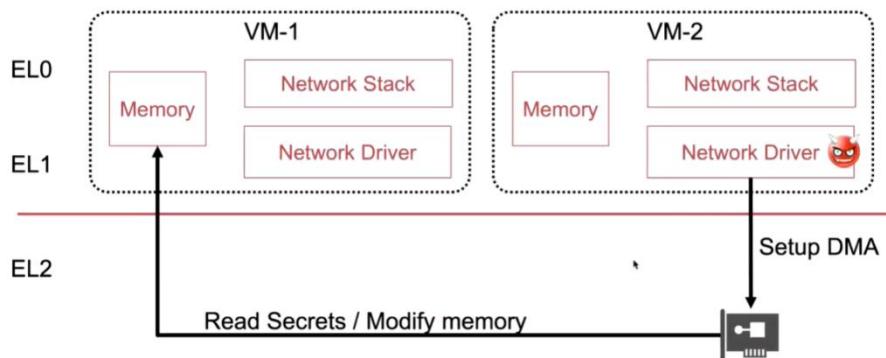
方法3：设备直通

- 虚拟机直接管理物理设备



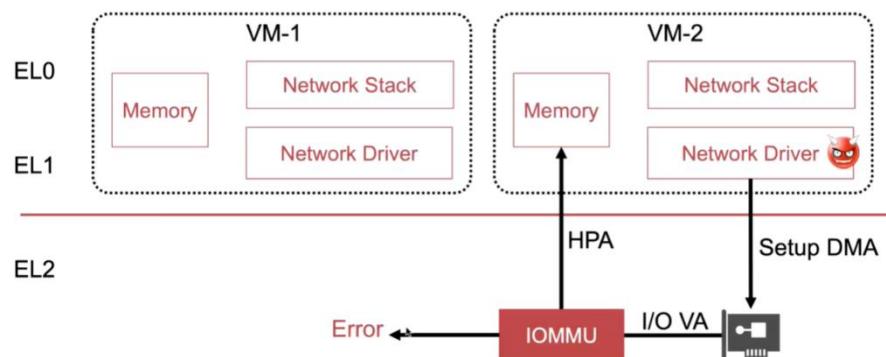
但是会有如下的问题。DMA 是一个异步的数据搬运机制。是完全绕过 CPU 的，所以 CPU 在给 Guest 配置页表的时候，都是作用在 CPU 的 MMU 里。那么它就难以对 DMA 做限制。所以 Guest 用的地址就可以直接通过 DMA 绕过 CPU 内存翻译的检查，读到另一个 VM 的内存，甚至可以直接写别的虚拟机的内存。

问题1：DMA恶意读写内存



为了解决这个问题，我们在 DMA 的数据通路上加一个页表，叫做 IOMMU。IOMMU 就是把虚拟机内部的地址翻译成 HPA。如果是允许的，我们就翻译成该访问的内存，如果不允许，那么我们的 error。这个 IOMMU 是在主板上的，和 CPU 无关，页表可以被 hypervisor 配置的。

使用IOMMU

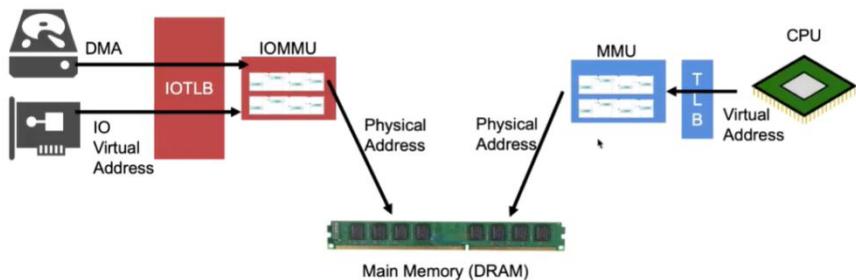


Q: 这个页表应该配置什么内容呢？

A: 我们直接用 CPU 的 MMU 中 stage2 的页表。

最终都是翻译成真正用来寻址的物理地址。

IOMMU与MMU



ARM SMMU

- **SMMU是ARM中IOMMU的实现**
 - System MMU
- **SMMU的设计与AARCH64 MMU一致**
 - 也存在两阶段地址翻译
 - 第一阶段：OS为进程配置：IOVA->GPA
 - 第二阶段：第一阶段翻译完之后进行第二阶段
 - VMM为VM配置：GPA->HPA

第二个问题就是可能造成浪费，或者物理设备不可能被虚拟机独占。

问题2：设备独占

- **Scalability不够**
 - 设备被VM-1独占后，就无法被VM-2使用
- **如果一台物理机上运行16个虚拟机**
 - 必须为这些虚拟机安装16个物理网卡

为了解决这个问题，硬件设备厂商遵循了 SR-IOV 规范。这是物理设备的标准，遵循这个标准的设备会在设备层面做一层虚拟化。也就是虽然我们是一个网卡，但它在插到主板上之后就可以看到 16 张网卡（Virtual Function）。每个虚拟机可以配置一个独立的 VF，其实就是一套独立的 MMIO 范围。

VF 可以保证数据通路是隔离的。hypervisor 拿到的是 PF，它控制着其他 15 个 VF，可以

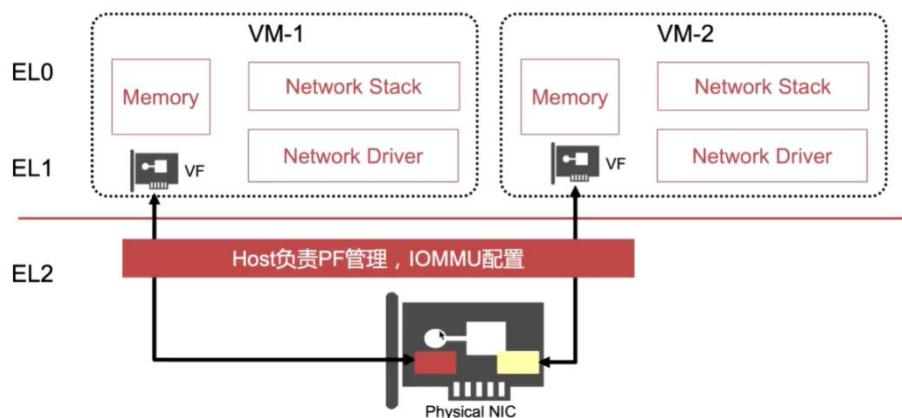
决定哪个 VF 可以使用，这其实也是控制面。

SR-IOV (Single Root I/O Virtualization)

Single Root I/O Virtualization (SR-IOV)

- SR-IOV是PCI-SIG组织确定的标准
- 满足SRIOV标准的设备，在设备层实现设备复用
 - 能够创建多个Virtual Function(VF)，每一个VF分配给一个VM
 - 负责进行数据传输，属于数据面 (Data-plane)
 - 物理设备被称为Physical Function(PF)，由Host管理
 - 负责进行配置和管理，属于控制面 (Control-plane)
- 设备的功能
 - 确保VF之间的数据流和控制流彼此不影响

SR-IOV的使用



因为数据通路上不需要软件模拟或者转发，所以性能非常好。都不需要 back-end driver 了。VM 只需要装上对应驱动即可了，启动 VM 的时候配置 PF 和 MMU 即可。缺点就是依赖于设备，设备必须满足 SRIov 标准，并且依赖于 IOMMU，它的功能还在持续演进中。第二个缺点就是它不能实现 interposition，因为所有数据通路上的数据包都不经过 hypervisor，hypervisor 看不到里面有什么东西，不能做防火墙的检查，所以 hypervisor 对虚拟机的迁移做支持。但是虚拟机的热迁移是非常重要的概念。数据中心里可能有几万台服务器，要对有些服务器更新，那就必须依赖于虚拟机热迁移迁移到别的物理机上，而 SRIov 就很难做热迁移。

设备直通的优缺点

- **优点**

- 性能优越
- 简化VMM的设计与实现

- **缺点**

- 需要特定硬件功能的支持 (IOMMU、SRIOV等)
- 不能实现Interposition : 难以支持虚拟机热迁移

现在流行的还是半虚拟化，性能影响少，但是有 interposition。

I/O虚拟化技术对比

	设备模拟	半虚拟化	设备直通
性能	差	中	好
修改虚拟机内核	否	驱动+修改	安装VF驱动
VMM复杂度	高	中	低
Interposition	有	有	无
是否依赖硬件功能	否	否	是
支持老版本OS	是	否	否

前面我们讲的都是 hypervisor 直接调用 handler，但是硬件也提供了中断虚拟化的支持。

中断虚拟化

- **VMM在完成I/O操作后通知VM**

- 例如在DMA操作之后

- **VMM在VM Entry时插入虚拟中断**

- VM的中断处理函数会被调用

- **虚拟中断类型**

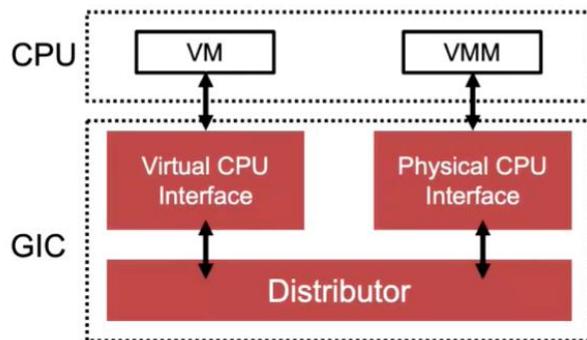
- 时钟中断
- 核间中断
- 外部中断

ARM中断虚拟化的实现方法

- 打断虚拟机执行
 - 通过List Register插入
- 不打断虚拟机执行
 - 通过GIC ITS插入

Virtual CPU Interface

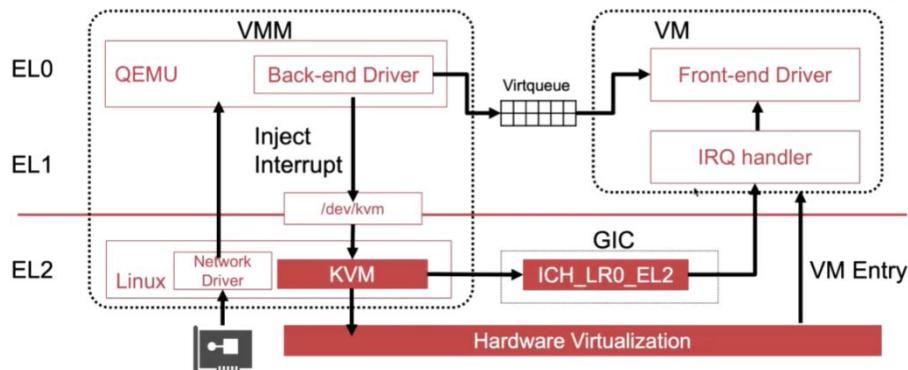
- GIC为虚拟机提供的硬件功能
 - VM通过Virtual CPU Interface与GIC交互
 - VMM通过Physical CPU Interface与GIC交互



在插入中断的时候，过去需要改 Guest IP 进行跳转。有了中断虚拟化之后，hypervisor 只需要修改寄存器的特点 bit。GIC 在启动过程中会发现 bit 置上，硬件会主动调用 handler。

插入虚拟中断：以半虚拟化举例

思考：这种方式有什么问题？

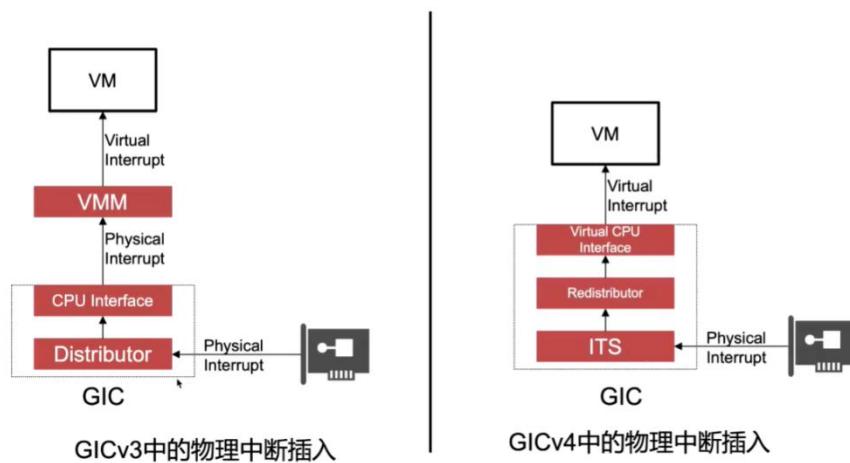


在虚拟机最开始运行的时候，把物理中断和虚拟机内部的中断号进行连接，中断来了以后直接进入到虚拟机中，不需要经过 hypervisor。

不打断虚拟机执行：GIC ITS

- GIC第4版本推出了Direct injection of virtual interrupts
 - 将物理设备的物理中断与虚拟中断绑定
 - 物理设备直接向虚拟机发送虚拟中断
- VMM在运行VM前
 - 配置GIC ITS (Interrupt Translation Service)
 - 建立物理中断与虚拟中断的映射
 - 映射内容
 - 设备与物理中断的映射
 - 分配虚拟中断号
 - 发送给哪些物理核上的虚拟处理器

虚拟中断的直接插入



2022/5/12

轻量级隔离

今天我们主要来讲轻量级的隔离，上完虚拟化之后，大家知道虚拟机之所以实用，就是可以把整个环境隔离开。虚拟化主要分为CPU虚拟化、内存虚拟化、I/O虚拟化。CPU虚拟化最重要的就是Trap & Emulate，重点就是我们在哪里拦截。实际上我们就是在特权指令这里拦了一道关卡。Emulate可能真的去做特权指令的操作，也有可能做模拟的操作。比如IO原来想控制显示器，我们也可以使用Canvas（如：QT库）把像素点画到显示器上。Trap & Emulate和内存虚拟化、I/O虚拟化都是密切相关的。在影子页表的实现中，Trap & Emulate也起了关键的作用。

Review : 虚拟化的技术

虚拟化	软件方案	硬件方案
CPU	<ul style="list-style-type: none">Trap & Emulate指令解释执行二进制翻译Para-virtualization	<ul style="list-style-type: none">EL-2 (ARM)
内存	<ul style="list-style-type: none">Shadow page tableSeparating page tables for U/KPara-virtualization: Direct paging	<ul style="list-style-type: none">Stage-2 PT (ARM)
I/O	<ul style="list-style-type: none">Direct I/O设备模拟Para-virtualization: Front-end & back-end driver (e.g., virtio)	<ul style="list-style-type: none">SMMU (ARM) / IOMMU (x86)SR-IOV

虚拟化分为软件方法和硬件方法，现在都是硬件方法。之所以讲软件方法是因为影子页表容易帮助大家理解页表的机制。还有一种叫做直接页表，它是破坏了封装性的，VM 是知道自己跑在 VM 里的。这种暴露本身会导致它能看到最终的 HPA，一旦一个 VM 可以看到 HPA，就代表它可以知道内存上的布局，从而利用攻击改变相邻位的值。我们在做虚拟化的时候，看起来我们完全可以让一个 VM 知道自己在 VM 里，但是暴露的信息本身可能导致侧信道、RawHammer 攻击变得简单。

对于今天的虚拟化来说，硬件方案很成熟，它不仅提供了虚拟化的功能。还提供了哪些特权指令可以发生 Trap、哪些特权指令可以自己完成。比如 svc 指令到底要不要触发 VM_EXIT 是可以自己配置的，对于性能要求较高的 VM，可以由硬件自动识别异常向量表找到入口。另一种做法，就是我们可以把 svc 全部设置成 VM_EXIT，然后再回去，跳转到对应的位置，这种方式可以记录下 VM 调用了哪些 syscall，这样 hypervisor 可以重新实现一下 VM 内部的 syscall，也可以只记录一个日志，或者做 syscall 的权限管理，hypervisor 可以 ban 掉危险的 syscall。

在某些情况下用影子页表可能比两级页表要快。随着时间的发展，硬件也在不断加速，比如更大的两级页表的 TLB。

云厂商普遍用虚拟化来隔离

• 虚拟化的优势

- 可以运行完整的软件栈，包括不同的操作系统
- 灵活的整体资源分配（支持动态迁移）
- 方便的添加、删除、备份（只需文件操作）
- 虚拟机之间的强隔离（能抵御 fork bomb）

• 虚拟化的问题：太重

- 云：性能损失，尤其是I/O虚拟化
- 用户：两层操作系统导致资源浪费

现在主要还是两层 Linux，也就是 Linux 虚拟机跑在 Linux 里，这样我们的 Cache 和调度都有两层，还会导致网络栈有两层，这种操作就导致了系统资源被浪费掉。这里面是有一些权衡在里面的。

Windows Server 用过的人不多，它是运行在服务器上。包括阿里之前提出了云电脑，就

是很小一块卡片，插上显示器和鼠标键盘就能用，它背后就是一个远程桌面。我们要一个新的用户就建立一个用户，但是所有用户都看得见一个根目录。我们希望做到把用户看到的文件系统做的不一样。

是否有更轻量级的隔离？

- **多用户机制**
 - 如：Windows Server允许多个用户，同时远程使用桌面
 - 多个用户可以共享一个操作系统，同时进行不同的工作
- **缺点：多个用户之间缺少隔离**
 - 例如：所有用户共同操作一个文件系统根目录
- **如何想让每个用户看到的文件系统视图不同？**
 - 对每个用户可访问的文件系统做隔离

CHROOT

我们不能运行用户共享一个根目录，如果共享根目录了，看到的文件系统一定是一样的。所以 Linux 提出为每个执行环境提出单独的文件系统的视图。如果我们能改变用户的根目录，那么我们就可以把用户限制在很小的范围内。

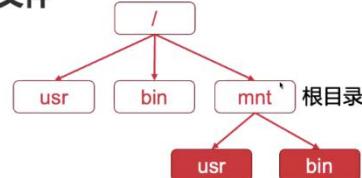
文件系统视图的隔离

- **为每个执行环境提供单独的文件系统视图**
- **原理**
 - Unix系统中的“一切皆文件”设计理念
 - 对于用户态来说，文件系统相当重要
- **方法**
 - 改变文件系统的根目录，即chroot

所以 Chroot 的效果就是可以控制进程的根目录是什么。比如进程在 mnt 根目录下的时候，只能看到 usr 和 bin。这在交叉编译的时候很有用，比如我们要在 x86 上编译 ARM，我们先把 ARM 的工具链准备好，然后再把硬盘拆下来放到 x86 里，Chroot 到这个目录下就可以开始跑了。

Chroot效果

- 控制进程能够访问哪些目录子树
- 改变进程所属的根目录
- 进程只能看到根目录下属的文件



它的原理很简单，当我们找文件的时候有文件名，只能从根目录去找。这个简单设计，一旦遇到“..”这种硬链接就很危险，用户进程可以一路“..”到达真正的根目录去

Chroot原理

- **进程只能从根目录向下开始查找文件**
 - 操作系统内部修改了根目录的位置
- **一个简单的设计**
 - 内核为每个用户记录一个根目录路径
 - 进程打开文件时内核从该用户的根目录开始查找
- **上述设计有什么问题？**
 - 遇到类似“..”的路径会发生什么？
 - 一个用户想要使不同进程有不同的根目录怎么办？

所以我们要做特殊的检查，使得“..”不能打破隔离。

Chroot在Linux中的实现

- **特殊检查根目录下的“..”**
 - 使得“..”与“/”等价
 - 无法通过“..”打破隔离
- **每个TCB都指向一个root目录**
 - 一个用户可以对多个进程chroot

```
struct fs_struct{  
    .....  
    struct path root, pwd;  
};  
  
struct task_struct{  
    .....  
    struct fs_struct *fs;  
    .....  
};
```

Chroot 不能限制 root 本身去调用 chroot 跳出。为了保证 chroot 有效，必须通过 setuid 消除目标进程的 root 权限。

正确使用Chroot

- 需要root权限才能变更根目录
 - 也意味着chroot无法限制root用户
- 确保chroot有效
 - 使用setuid消除目标进程的root权限

```
chdir("jail");
chroot(".");
setuid(uid); // UID > 0
```

Chroot 本质上就是基于文件系统的 Name Space 来实现。如果用户不通过直接文件名访问，而是有能力通过 inode 访问，就可以绕过这一点，所以我们不允许用户通过 inode 来访问。当然在 inode 层，我们也可以实现对用户的限制。

基于Name Space的限制

- 通过文件系统的name space来限制用户
 - 如果用户直接通过inode访问，则可绕过
 - 不允许用户直接用inode访问文件
- 其它层也可以限制用户
 - 例如：inode层可以限制用户



Chroot能否实现彻底的隔离？

- 不同的执行环境想要共享一些文件怎么办？
- 涉及到网络服务时会发生什么？
 - 所有执行环境共用一个IP地址，所以无法区分许多服务
- 执行环境需要root权限该怎么办？
 - 全局只有一个root用户，所以不同执行环境间可能相互影响
- 不能，因为还有许多资源被共享...

Q: 如果我们想要多个执行环境共享一些文件怎么办了？隔离和共享是双刃剑。

Q: 一旦涉及到网络服务怎么办？我们 chroot 完了，网络 IP 还是一致的，我们希望 http 服务能够隔离开。

Q: 如果执行环境就是要 root，一旦给了某个执行环境 root 权限，就会影响到其他环境。

Linux Container

所以 Chroot 解决了一些问题的同时，引来了更多的问题。所以出现了 Linux Container，它是基于前面这么多问题的情况下出现的解决方案。

它是由 Linux 内核提供的隔离机制。

LinuX Container (LXC)

- **基于容器的轻量级虚拟化方案**

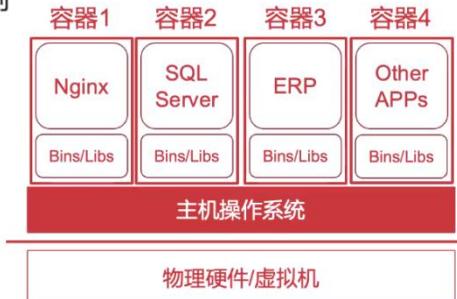
- 由Linux内核提供资源隔离机制

- **安全隔离**

- 基于**namespace**机制

- **性能隔离**

- Linux cgroup



Chroot 改变的文件系统的 namespace，而在这里提出了 7 种 namespace。每个 namespace 能够保证进程只能访问自己封装后的局部资源。

Linux Namespace (NS)

- **每种NS封装一类全局资源**

- 进程只能访问封装后的局部资源
 - 目前一共有七种NS

- **进程通过系统调用控制NS**



Mount Namespace

1、Mount Namespace

- **容器内外可部分共享文件系统**

- 思考：如果容器内修改了一个挂载点会发生什么？

- **假设主机操作系统上运行了一个容器**

- Step-1：主机OS准备从/mnt目录下的ext4文件系统中读取数据
 - Step-2：容器中进程在/mnt目录下挂载了一个xfs文件系统
 - Step-3：主机操作系统可能读到错误数据

每个 namespace 有自己独立的文件系统树。每个容器都会有自己的 mnt namespace。这样对于容器来说，自己的挂载对别人是不可见的。

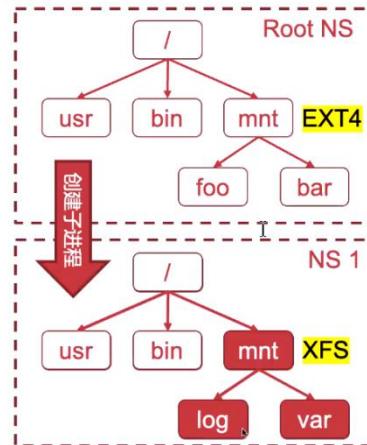
Mount Namespace的实现

- **设计思路**

- 在内核中分别记录每个NS中对于挂载点的修改
- 访问挂载点时，内核根据当前NS的记录查找文件

- **每个NS有独立的文件系统树**

- 新NS会拷贝一份父NS的文件系统树
- 修改挂载点只会反映到自己NS的文件系统树



不修改 kernel 的情况下，这个是不能实现的。因为要 copy 一份文件系统树。、

IPC Namespace

IPC 对象就是可以用来做 IPC 的内核的对象。如果我们要在两个进程之间共享内存，我们可以给共享内存维护一个名字。my_mem 显然可以传给进程 A 和进程 B 上，这里就有一个命名空间的问题。my_mem 在原来的例子中是全局唯一的，但是现在有两个容器，容器 A 和容器 B 都想使用 my_mem，对于它们自己来说，是不想和别人共享的，因为它们根本看不到对方。简单的解决方案就是它们不能使用重名的对象名字，这听上去是没问题的，但是暴露了别人在使用这个名字。

这个就和直接页表的时候暴露 HPA 给虚拟机是一样的，如果我们知道了别人在使用 my_mem 这件事情，可能泄露了一些别的信息。

2、IPC Namespace

- 不同容器内的进程若共享IPC对象会发生什么？

- 假设有两个容器A和B

- A中进程使用名为“my_mem”共享内存进行数据共享
- B中进程也使用名为“my_mem”共享内存进行通信
- B中进程可能收到A中进程的数据，导致出错以及数据泄露

更好的操作就是把 my_mem 让不同的人有不同的 namespace。

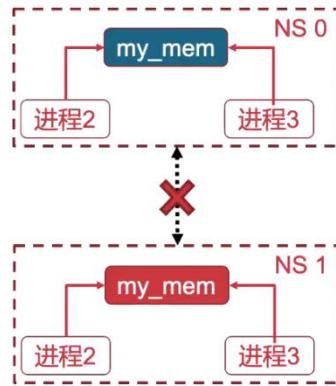
IPC Namespace的设计

- 直接的想法
 - 在内核中创建IPC对象时，贴上对应NS的标签
 - 进程访问IPC对象时内核来判断是否允许访问该对象
- 可能的问题
 - 可能有timing side channel隐患
 - 对于同名的IPC对象不好处理
- 更进一步
 - 将每个NS创建的IPC对象放在一起管理

所以每个 IPC 对象只能属于一个 IPC Namespace，从而避免了之前的问题。

IPC Namespace的实现

- 使每个IPC对象只能属于一个NS
 - 每个NS单独记录属于自己的IPC对象
 - 进程只能在当前NS中寻找IPC对象
- 图例
 - 即使不同NS的共享内存ID均为 my_mem → 不同的共享内存



在内核里谈隔离的时候，要讨论到软件所创建的隔离。

Network Namespace

3、Network Namespace

- 不同的容器共用一个IP会发生什么？
- 假设有两个容器均提供网络服务
 - 两个容器的外部用户向同一IP发送网络服务请求
 - 主机操作系统不知道该将网络包转发给哪个容器

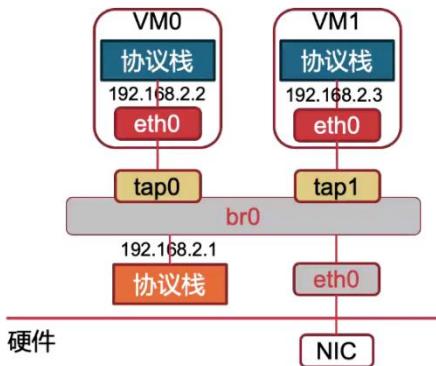
在 Linux 中，是把多 IP 支持到内核里面的，支持虚拟的网桥、虚拟的 IP、虚拟的网络设备等。

Linux对于多IP的支持

- 在虚拟机场景下很常见

- 每个虚拟机分配一个IP
 - IP绑定到各自的网络设备上
- 内部的二级虚拟网络设备
 - br0: 虚拟网桥
 - tap: 虚拟网络设备

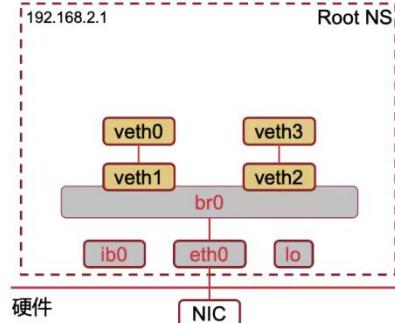
- 如何应用到容器场景 ?



在容器里，我们可以在虚拟的网桥上创建虚拟机的网卡。这样不同的容器之间虚拟的网络设备也可以是没有关系的。

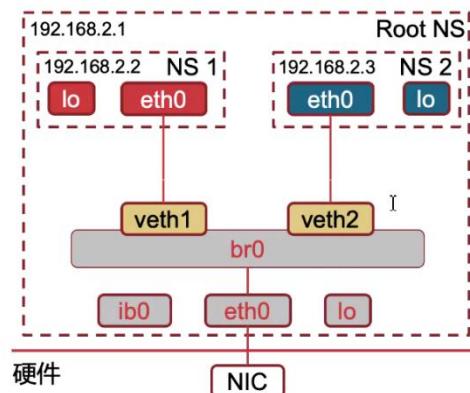
Network Namespace的实现

- 每个NS拥有一套独立的网络资源
 - 包括IP地址、网络设备等
- 新NS默认只有一个loopback设备
 - 其余设备需后续分配或从外部加入
- 图例
 - 创建相连的veth虚拟设备对
 - 一端加入NS即可连通网络
 - 分配IP后可分别与外界通信



Network Namespace的实现

- 每个NS拥有一套独立的网络资源
 - 包括IP地址、网络设备等
- 新NS默认只有一个loopback设备
 - 其余设备需后续分配或从外部加入
- 图例
 - 创建相连的veth虚拟设备对
 - 一端加入NS即可连通网络
 - 分配IP后可分别与外界通信



PID Namespace

PID 也是一种资源。我们必须让不同容器之间的 PID Namespace 不同，最简单的方法就是前面加一个前缀数字。

4、PID Namespace

- 容器内进程可以看到容器外进程的PID会发生什么？

- 假设容器内存在一个恶意进程

- 恶意进程向容器外进程发送SIGKILL信号
- 主机操作系统或其他容器中的正常进程会被杀死

Q: 容器能不能防御 fork bomb?

A: 不能防御，容器复用了进程和线程的机制。容器中 fork 出了一个 thread 后，整个机器上也多了一个 thread。TCB (thread control block) 还是由 host 内核来控制的，如果我们要自己实现一套这个机制，那就变成虚拟机了。

我们需要在父 NS 中看到子 NS。

PID Namespace的设计

- 直接的想法**

- 将每个NS中的进程放在一起管理，不同NS中的进程相互隔离

- 存在的问题**

- 进程间关系如何处理（比如父子进程）？

- 更进一步**

- 允许父NS看到子NS中的进程，保留父子关系

最终实现如下：

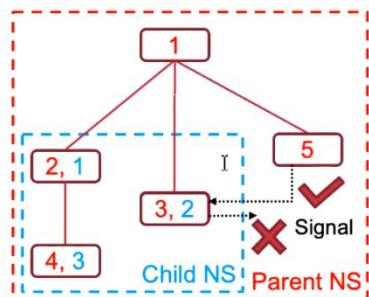
PID Namespace的实现

- 对NS内外的PID进行单向隔离**

- 外部能看到内部的进程，反之则不能

- 图例**

- 子NS中的进程在父NS中也有PID
- 进程只能看到当前NS的PID
- 子NS中的进程无法向外发送信号



User Namespace

最简单的场景就是 root。比如我们要修改容器里的/etc。

5、User Namespace

- 容器内外共享一个root用户会发生什么？
- 假设一个恶意用户在容器内获取了root权限
 - 恶意用户相当于拥有了整个系统的最高权限
 - 可以窃取其他容器甚至主机操作系统的隐私信息
 - 可以控制或破坏系统内的各种服务

于是我们就需要允许容器里拥有一些更高的权限，而不是容器外的更高的权限。这个可以基于 Linux Capability 机制，也就是一条条列好哪些能做、哪些不能做。这是比较符合直观的设置，缺点就是比较麻烦。

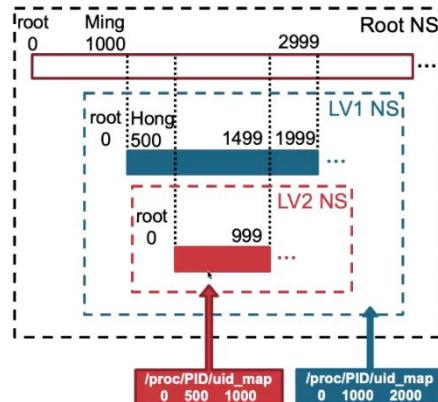
对应一个容器，root 用户在容器外是没有特权的，但是在容器内部就可以操作 userid=0 的文件，因为它在容器内的 userid 也是 0。

我们可以来看一下这个例子，一个普通用户 Ming (userid=1000) 在 child namespace (LV1 NS) 中，就是 root。还可以再嵌套一层，这个叫 Hong 的人它的 userid 是 500，它在 LV2 NS 中的 userid 就是 0。

通过这种方式，依然存在嵌套的关系，和进程的概念是比较类似的。对于红色的 Namespace，我们给它 0~1000。而 LV1 NS 的范围是 0~2000。LV1 NS 的 userid=500 对应到 Root NS 中就是 userid 1500，存在着这样的一个映射关系。任何一个用户在里面有一个 userid，在最外面也有一个 userid。因为它最终的所有操作都要从容器调到内核里，所以一个用户同时会有好几个 userid。在内核里就可以判断这个用户想在哪个容器中做事情，从而判断权限。

User Namespace的实现

- 对NS内外的UID和GID进行映射
 - 允许普通用户在容器内有更高权限
 - 基于Linux Capability机制
 - 容器内root用户在容器外无特权
 - 只是普通用户
- 图例
 - 普通用户在子NS中是root用户



进一步限制容器内Root

- 如果容器内root要执行特权操作怎么办?
 - insmod? 一旦允许在内核中插入驱动，则拥有最高权限
 - 关机/重启? 整个云服务器会受影响
- 1、从内核角度来看，仅仅是普通用户
- 2、限制系统调用
 - Seccomp机制

其他 Namespace

每个 Cgroup 要去限制资源的使用，而不是去做安全的隔离。Cgroup 是可以来防御 forkbomb 的，它可以限制容器最多能够创建多少个 pid，那么 fork bomb 到 500 就退出了。

其他Namespace

- 6、UTS Namespace
 - 每个NS拥有独立的hostname等名称
 - 便于分辨主机操作系统及其上的多个容器
- 7、Cgroup Namespace
 - cgroupfs的实现向容器内暴露cgroup根目录
 - 增强隔离性：避免向容器内泄露主机操作系统信息
 - 增强可移植性：取消cgroup路径名依赖

SFI: 软件错误隔离

前面我们说的这些都是利用了系统已有的能力去做隔离，根本思路还是由内核对线程和 NS 做隔离。要突破进程的边界显然是做不到的，所以其实质就是 OS 指定了一层层的 NS，使得我们不能突破出来。

SFI 希望不过分依赖内核，希望在用户态以软件的方式做隔离。它提出了一些概念，比如在一个 address space 中只有一个进程。我们想在进程内做隔离的话，我们就可以设置一个 domain，也就是把内存区域划分给不同的 domain，这样一旦 1000~2000 中的代码出了问题，它也不能修改 3000 位置的数据。

所以它必须在 domain 运行的时候检查所有的访存操作（Load & Store）和 jump 操作。

内部沙盒：三个限制

- 同一个地址空间内的隔离
 - 每个“进程”就是一个独立的“domain”
 - 每个domain都有自己的内存区域（如上下边界）
 - 每个domain在运行时，会检查所有的访存操作，保证不越界
- Tradeoff：检查和通信
 - 增加的开销：检查访存，domain之间的切换
 - 减少的开销：不用再陷入内核

如果我们能实现这样一个 domain 的概念，我们就可以把我们不太相信的代码隔离在 domain 中，跑完之后给一个结果，跑的过程中是不会跳转出来的。所以我们增加了检查访存和 domain 之间的切换的开销。

SFI 的 Domain

Domain 就是允许在一个内存区域的代码+数据，它本身形成一个闭包，有一个唯一的 id。每个 domain 肯定比进程小，但是也有自己的堆、栈、数据区。对于要加载到 domain 中的代码，我们对里面的所有地址就加上一个 mask，使得都在 domain 之中。比如我们想让进程运行在 0~1000 地址中，那么哪怕地址前面有再多的 1，我们也 mask 掉。

Domain

- 运行在一个内存区域的代码和数据
 - 通过一个唯一的ID来标识
 - 代码和数据是分离的（如堆、栈）
- 二进制重写，保证无法从domain逃逸
 - 所有的地址在访问前都会被mask
 - 属于粗粒度的内存安全（相比Java的类型安全，如数组边界检查）
 - 在每条jmp语句之前同样插入代码，保证无法跳出特定区域
- 问：是否有可能抵御buffer overflow？

Q: 有没有办法抵御 buffer overflow？栈出了问题覆盖掉了指针。

A: 就算我们覆盖了指针，我们在执行的时候还是要添加 mask，可能不知道跳转到哪里去了，但是只能在自己的 domain 中。所以可以抵御一部分。

检查所有的jmp和store指令

- 如果目标地址可以静态确定
 - 用内存段的upper bits来mask
 - 可能导致domain的crash，但至少不会篡改其他domain的内存
- 如果地址只能动态确定
 - 在访问地址的代码前插入检查代码（二进制重写）
- 保证代码不会跳过检查
 - 目标地址放在一个特定的寄存器中
 - 保证这个值只能够被插入的代码原子地修改，且这个值只能来自内存段

我们来举如下的一个例子。我们的 `write x`, `x` 是一个指针。所以我们先把 `x` 前面的位去掉，然后 `or` 一下 `1200`，就把 `x` 转换成了 domain 中的地址，可以去写数据了。

简单的 SFI 示例

- domain = 0x1200 ~ 0x12FF

- 原始代码:

`write x`

- Naïve SFI:

`x := x & 00FF`

`x := x | 1200`

`write x`

Q: 但是如果攻击者直接跳转到了 `write x` 中会发生什么情况呢？这就可能把 `x` 写坏掉，绕过了前面的 mask 检查，`write x` 就可能覆盖别的 domain 的值。

A: 解决方案就是我们可以把计算的值都放到寄存器 `tmp` 中。我们一开始可以初始化 `tmp` 为 `1200`, `tmp` 的值是永远不可能超过 domain 的范围的。这时候如果攻击者直接跳转到了 `write tmp`, 充其量就是修改了 domain 中的值。

- domain = 0x1200 ~ 0x12FF

- 原始代码:

`write x`

- Naïve SFI:

`x := x & 00FF`

`x := x | 1200`

`write x`

} 将 `x` 转换到一个 domain 内的地址

如果直接跳转到这里怎么办？

- Better SFI:

`tmp := x & 00FF`

`tmp := tmp | 1200`

`write tmp`

但依然不够，需要保护控制流

CFI: 控制流完整性保护 (Control Flow Integrity)

如果我们想继续保护控制流，就必须使用 CFI。我们先得到 CFG，在运行的过程中，保

证和 CFG 一致。

回顾：控制流完整性

- 程序运行前：预先确定应用程序的控制流图CFG
 - 方法-1：通过对源代码的静态分析
 - 方法-2：通过对二进制的静态分析
 - 方法-3：通过动态执行的profiling
 - 方法-4：显示规定安全策略的规范
- 程序运行时：实际的控制流必须符合CFG

通过二进制插桩的方式实现CFI

- 扫描程序的二进制并重写部分代码
 - 类似SFI的实现
- 插入检查代码保证运行时的执行符合CFG
 - 当运行分支跳转指令后，立即检查是否合法
- 目标：防止跳转到任意代码
 - 即使攻击者已经完整控制了可写部分的内存，也依然可以保证
 - 问：如果攻击者修改了函数指针，是否能防御？

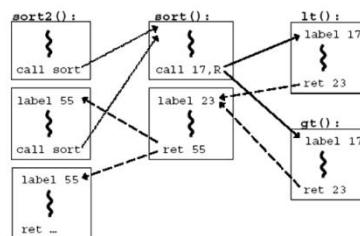
我们在跳转的地方插入一个 id，在每次跳转的时候检查 label 是否和跳转的地方是一致的。

CFI的例子

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



控制流的正确性保证

- 对于每一次跳转，首先静态确定其合法目标地址
- 在每个合法地址插入一个唯一的标识（bit-pattern）
 - 如果从一个源地址可能跳转到两个目标地址，则这两个地址等价
 - 这导致CFG的准确度下降
 - 对于等价的地址，使用相同的标识
- 插入的指令会在运行时检查标识是否正确

每个 ID 是唯一的，并且不会出现在检查外的其他地方。并且我们要求代码是不可写的，否则我们的 buffer overflow 虽然没有直接破坏 CFG，但是破坏了 return 的逻辑。只要加载的时候有 CFI 的保证，那么运行的时候就有保证。

CFI机制：防止被绕过

- 唯一的ID
 - 在二进制文件中，ID不能出现在除了检查之外的其他地方
 - 否则可能跳转过去（可能是指令中间），从而打破CFG的约束
- 代码不可写
 - 程序无法在运行时修改代码（自修改代码的支持会更困难）
- 数据不可执行
 - 因为假设是攻击者有能力修改数据
- 几点保证：硬件支持+禁止部分系统调用+加载时检查
 - 禁止那些有可能影响保护状态的系统调用，比如mprotect

改进CFI的准确度

- 假设静态分析后，允许： $A \rightarrow C, B \rightarrow [C|D]$
 - 于是C、D的标识是一样的
 - 导致A也可以跳转到D，但实际是不允许的
 - 解决方法：复制一份B的代码，使C和D的标识不同
- 假设静态分析后，允许： $A \rightarrow F, B \rightarrow F$
 - 但实际运行时， $A \rightarrow F$ return-to B，无法检查出来
 - 解决方法：Shadow Stack，专门且仅仅保存返回地址，用于检查

CFI的安全性保证

- 能够有效防御非法劫持控制流的攻击
 - Stack overflow、return-to-libc、指针篡改等
- 但并不能防御那些不修改原CFG的攻击
 - 使用错误的参数进行系统调用（如：把机密数据发出网络）
 - 替换文件名（如：通过文件名窃取机密数据）
 - 修改某个表示权限的数据字段（从而获取更高权限）
 - 其他data-only的攻击

XFI: SFI+CFI

光有 CFI 是不够的，我们把 SFI 和 CFI 结合，就成为了 XFI。

XFI : 通过SFI和CFI保护操作系统

- 通过二进制重写实现安全检查
 - 支持遗留代码
- 支持CFI防止被绕过
- 支持对内存的细粒度访问控制策略
 - 相比而言，CFI仅能提供简单的内存安全性
- 依赖加载时的验证
- 可用来保护操作系统中的驱动，以及多媒体编解码器模块

两个栈

- **Scoped Stack : 用于保存返回值和部分本地变量**
 - 记录下所有的函数调用、返回、异常处理
 - 无法被overflow，只能通过一个计算后的指针来访问
 - 通过软件检查来保护栈的完整性
 - 在程序加载时，会验证这些检查确实存在
- **Allocation Stack : 记录一些数组或本地变量**
 - 这些变量的地址是可以传递的

整个区域运行在一个 domain 中，每次访存都会检查，避免修改其他模块的内存。

内存的访问控制

- 模块（例如驱动）只能访问分配给它的内存
 - 即使是自己的内存也有限制，如无法随意修改scoped stack
- 模块外的host则可以允许模块访问其他的连续内存段
 - 粒度非常细，可以到字节粒度
 - 通过静态验证的方法来确定对常量地址和scoped stack的访问
 - 通过动态的指令来实现对其他内存的访问检查

防止被绕过

- XFI保护机制自身的完整性
 - 基本的CFI
 - scoped stack能够避免函数乱序返回（即A->F return-to B攻击）
 - 一些危险的指令不会执行，或受限执行
 - 比如在加载时扫描二进制，禁止模块中出现一些特权指令
- 所以XFI可以在内核中运行

Native Client

有了 XFI 之后，Google 提出了一个很像的 Native Client，它的目标和 Kernel 很像。早期浏览器只能运行 java 和 js，微软允许浏览器跑二进制代码，但是这个很危险。

Native Client

- 目标：允许浏览器安全地运行一个下载的x86二进制
 - 毕竟二进制比js、java等快得多
 - 传统的ActiveX：仅仅依赖签名，没有运行时限制
 - 传统的.Net：中间层的字节码 + 验证
- NativeClient：运行不可信x86代码的用户态隔离沙箱
 - 一个受限的x86汇编指令的子集
 - 类似SFI的沙箱机制，保证内存的安全性
 - 受限的系统接口（系统调用）
 - 接近本地代码的运行性能

NaCl中的CFI

NaCL沙箱

- 受限的x86汇编代码子集
 - 允许可靠的反汇编，并且有效的验证
 - 禁止不安全的指令：syscall, int, ret, 依赖内存的jmp和call, 所有特权指令, 对段状态修改的指令, ...
- 禁止访问特定段之外的内存
 - 类似SFI
- CFI保证

- 对直接跳转，静态计算出跳转地址，并验证合法性
 - 对于后续的汇编指令来说，必须是reachable的
- 对间接跳转，必须通过如下方式编码

```
and %eax, 0xffffffe0
jmp *%eax
```

 - 保证跳转目标地址是32位对齐的
- 没有ret指令
 - 使用间接跳转指令来取代

与沙盒外的Host交互

- 可信的运行时环境
 - 用于创建线程、内存管理、其他系统服务
- 从不可信→可信部分的控制流转换
 - 固定的入口：trampolines
 - 重设寄存器，恢复线程栈，执行可信的服务handler
- 从可信部分→不可信代码
 - 称为：springboard
 - 可以跳转到任意不可信的地址，并执行线程

NaCl沙盒的其他方面

- 禁止通过OS访问网络
 - 禁止调用类似connect或accept之类的系统调用
 - 仅允许用JavaScript通过浏览器访问网络
- IMC : Inter-Module Communication服务
 - 一种特殊的IPC，类似socket的抽象
 - 允许JavaScript通过DOM对象访问，实现模块之间的通信
- 新的替代
 - WebAssembly (WASM)：语言级虚拟机

2022/5/17

操作系统安全机制

安全是一个 OS 非常重要的组成部分，OS 有一本书很有名 *three easy pieces*，分别为虚拟化、**persistence**（文件系统）和并发。又加了一个安全的章节。

轻量级隔离已经是安全的常见方式了。

Review : 轻量级隔离

- Chroot**
- 容器的Namespace**
- SFI、CFI、XFI**
- NaCl (SFI的一种实现)**

基于硬件的进程内隔离：ENCLAVE

上述这些都是基于软件的隔离，实际上都是基于硬件的 feature 的。那么硬件能不能直接提供隔离呢？硬件提供一个很小的运行环境，它抽象出来就是和 OS 隔离的运行环境。有

了这个运行环境之后，OS 不知道里面跑的是什么。所以这其实是一种对外的隔离，它只保护这个小环境，它认为内部是可信的，通过硬件加密的方式保护。

Enclave：以Intel SGX为例

- **SGX: Software Guard eXtension**
 - 2015年首次引入Intel Skylake架构
 - 保护程序和代码在运行时的安全 (data in-run)
 - 其他安全包括：存储时安全和传输时安全
- **关键技术**
 - Enclave内部与外部的隔离
 - 内存加密与完整性保护
 - 远程验证

它用 Merkle Tree 保护完整性，如果我们能把内存算一遍哈希，把哈希的 root 放到 CPU 里，这样我们就可以判断内存是否有被篡改。但是这会导致性能很差，读出来的内存要算哈希，还要算其他人的哈希算出总的哈希去和 CPU 内部存的 root 哈希去比对。

解决方法就是把一部分哈希直接保存在 CPU 里，保存一系列的哈希，当我们要去读某一块数据的时候，只需要把它周围的一块哈希算一遍，然后和 CPU 内部保存的位置比较一下。

内存的加密和哈希校验可以防御物理攻击（比如 NVM 断电、冷冻等），不能用来防御软件攻击的。我们需要加密的方式来保存内存数据。

硬件内存加密与保护机制

- **硬件加密保护隐私性**
 - CPU外皆为密文，包括内存、存储、网络等
 - CPU内部为明文，包括各级Cache与寄存器
 - 数据进出CPU时，由进行加密和解密操作
- **硬件Merkle Tree保护完整性**
 - 对内存中数据计算一级hash，对一级hash计算二级hash，形成树
 - CPU内部仅保存root hash，其它hash保存在不可信的内存中
 - 当内存中的数据被修改时，更新Merkle Tree

只有 256M 是用来做加密内存的，对硬件来说是无感的。CPU 把这块内存划分出来之后，专门给 Enclave 去使用。CPU 在创建 Enclave 的时候，必须在 CPU 的逻辑上要保存一些状态，它要记录和 Enclave 相关的数据结构，比如 128M 中拿了多少 EPC。如果运行到一半发现内存不够了，我们就需要 swap。

EPC (Enclave Page Cache)

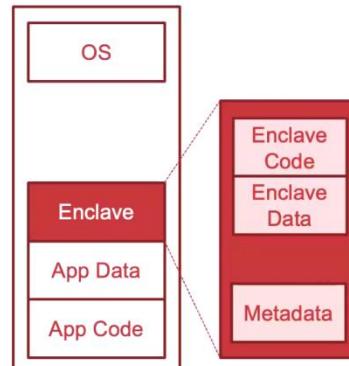
- **CPU预留一部分内存，仅允许Enclave访问**
 - 连续的128MB/256MB，这部分不会暴露给软件操作
 - 全部加密，并保证数据的完整性（即无法篡改）
 - 可能的篡改方法：通过总线直接修改
- **操作系统负责将EPC添加并映射至Enclave中**
 - 操作系统也可以触发swap，将数据从EPC交换到DRAM中
 - 由专门的硬件指令来进行swap，粒度为一个内存页（4K）
 - 注意：页表依然由不可信的操作系统控制

Enclave 和进程之间的关系

我们在虚拟地址空间中找了一块出来作为 Enclave，可能是 10M。这 10M，我们要往里填入 EPC，OS 然后要把 Enclave Code 加载到 EPC 里去，然后把 EPC 映射到 Enclave 虚拟地址中。加完之后，和 Enclave 相关的信息也要同样放入 Enclave 中。这块也是受到 EPC 保护的。

Enclave与进程的关系

- **Enclave是进程的一部分**
 - Enclave内外共享一个虚拟地址空间
 - Enclave内部可以访问外部的内存
 - 反之则不行
- **创建Enclave的过程**
 1. OS创建进程
 2. OS分配虚拟地址空间
 3. OS将Enclave的code加载到EPC中
 - 并将EPC映射到Enclave的虚拟地址
 - 循环3，完成所有code加载和映射
 4. 完成进程创建



当应用要访问 EPC 数据的时候，就必须使用专门的代码进入和退出。这种模式意味着这里面的代码可以和 OS 是完全没有关系的，可以有一套自己的定义和格式。它可以做到一个应用程序内部可以做隔离。

希望大家建立起一个光谱：虚拟化隔离、容器隔离、容器内部隔离、进程间隔离、进程内部隔离（软件：SFI+CFI、硬件：Enclave）。有了这个谱系之后，大家遇到先隔离的时候，就可以去调一个。

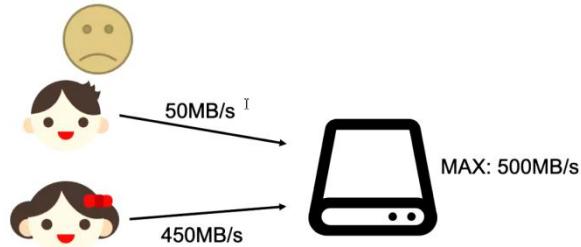
性能隔离

前面做的是安全性隔离。资源隔离其实更 tricky，比如两个人同时访问磁盘。如果是以

下情况，可能就不太平衡，或者我们希望控制不公平 or 公平。

资源竞争问题

- 小明和小红同时访问磁盘



Cgroups 就是来做这种事情的。要做性能隔离，是一个系统级别的工程，和多个资源相关，并且动态不断在变化的。

Control Cgroups (Cgroups)

- Cgroups是什么
 - Linux内核（从Linux2.6.24开始）提供的一种资源隔离的功能
- Cgroups可以做什么
 - 将线程分组
 - 对每组线程使用的多种物理资源进行限制和监控
- 怎么用Cgroups
 - 名为cgroupfs的伪文件系统提供了用户接口

Cgroup 其实是对线程分组，对每组线程使用的资源做监控和限制，它提供了一套伪文件系统。

Cgroups的常用术语

- 任务 (task)
- 控制组 (cgroup)
- 子系统 (subsystem)
- 层级 (hierarchy)

一个任务就是系统中的一个线程，线程是执行的最基本的单位。如果最小力度是进程，那么对线程很难处理。我们现在有两个任务。

控制组 (Control Group)

任务 (Task)

- 任务就是系统中的一个线程
- 例如：有两个任务 task1 和 task2



- Cgroups 进行资源监控和限制的单位

- 任务的集合

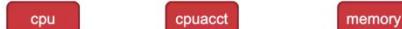
- 控制组 cgroup1 包含 task1
- 控制组 cgroup2 包含 task2
- 控制组 cgroup3 由 task1 和 task2 组成



层级 (Hierarchy)

子系统 (Sub-system)

- 可以跟踪或限制控制组使用该类型物理资源的内核组件
- 也被称为资源控制器



- 由控制组组成的树状结构
- 通过被挂载到文件系统中形成
- 一个任务在每个层级结构中只能属于一个控制组
- 一个子系统只能附加于一个层级结构
- 一个层级结构可以附加多个子系统

对于每个资源控制模型来说，有一些量。比如最大值：每个 cgroup 只能使用 1G（物理）内存。我们分配内存是使用 buddy system 的，OS 可以在分配的时候添加一个计数器。如果超过了，就 swap 自己已经分配的 1G 内存。

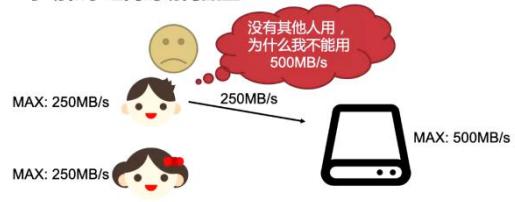
对 IO 来说，没有这么简单了。我们必须监控动态的速度，也就是向硬件发送请求的时候，记录发送速度，如果第一秒发太多了，那么我们第二秒就停一停。我们也可以以 10ms 为粒度做监控，使得平摊速度相同，但是粒度太细 overhead 会增加。

资源控制模型

资源控制模型

- 最大值

- 直接设置一个控制组所能使用的物理资源的最大值，例如：
 - 内存子系统：最多能使用 1GB 内存
 - 存储子系统：最大能使用 100MB/s 的磁盘 I/O



如何对任务使用资源进行监控和限制

- Cgroups 进行监控和限制的单位是什么？
 - 控制组
- 如何知道一个控制组使用了多少物理资源？
 - 计算该控制组所有任务使用的该物理资源的总和
- 如何限制一个控制组
 - 使该控制组的所有任务使用的物理资源不超过这个限制
 - 在每个任务使用物理资源时，需要保证不违反该控制组的限制

控制组可以是进程，也可以到达线程的粒度。

CPU子系统

- 回顾CFS (完全公平调度器)
 - 可以为每个任务设定一个 “权重值”
 - “权重值” 确定了不同任务占用资源的比例
- CPU子系统允许为不同的控制组设定CPU时间的比例
 - 直接利用CFS来实现按比例分配的资源控制模型
 - 为控制组设定权重：向cpu.shares文件中写入权重值（默认1024）

内存子系统

- 监控控制组使用的内存
 - 利用page_counter监控每个控制组使用了多少内存
- 限制控制组使用的内存
 - 通过修改memory.limit_in_bytes文件设定控制组最大内存使用量

内存子系统

- Linux分配内存首先需要charge同等大小的内存
 - 只有charge成功，才能分配内存
- Charge内存的简化代码（大小为nr_pages）：

```
new = atomic_long_add_return(nr_pages, &page_counter->usage);
if (new > page_counter->max) {
    atomic_long_sub(nr_pages, &page_counter->usage);
    goto failed;
}
```
- 释放内存时会执行相反的uncharge操作

存储子系统 (blkio)

- 限制最大IOPS/BPS
 - blkio.throttle.read_bps_device 限制对某块设备的读带宽
 - blkio.throttle.read_iops_device 限制对某块设备的每秒读次数
 - blkio.throttle.write_bps_device 限制对某块设备的写带宽
 - blkio.throttle.write_iops_device 限制对某块设备的每秒写次数
- 设定权重 (weight)
 - blkio.weight 该控制组的权重值
 - blkio.weight_device 对某块设备单独的权重值

隔离分为安全隔离和性能隔离。

小结：隔离的不同方式

- 物理机隔离
- 虚拟机隔离
- 文件系统隔离 : Chroot
- 容器隔离 : Name Space
- 进程隔离 : 传统方式
- 进程内隔离-软件方法 : SFI/CFI/XFI/NaCl
- 进程内隔离-硬件方法 : Enclave
- (性能隔离) : Cgroup

操作系统的安全服务

安全是 OS 绕不开的。系统有很多要保护的隐私数据，既要被合法的访问，又不能被非法的访问。后者是很难被证明 100% 成立的。

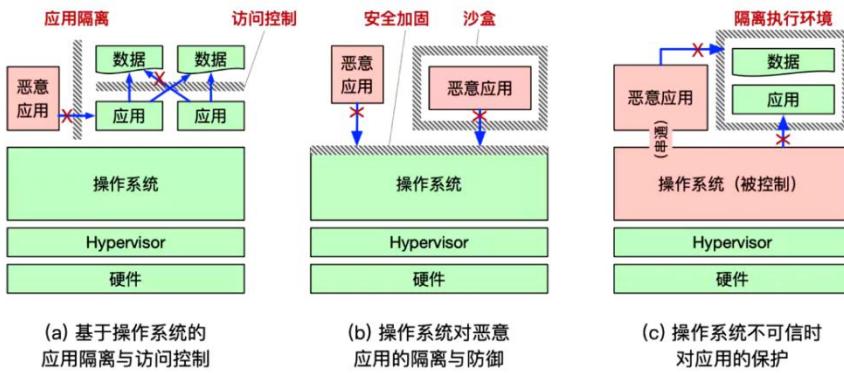
安全是操作系统的重要功能和服务

- 系统中有许多需要保护的数据
 - 如账号密码、信用卡号、地理位置、照片视频等
 - 操作系统需要允许这些数据被合法访问，但不允许被非法访问
- 系统中可能存在许多恶意应用
 - 操作系统需要与这些恶意应用作斗争，保护自己，限制对方
- 操作系统不可避免的存在漏洞
 - 操作系统需要考虑自己被完全攻破的情况下依然提供一定的保护

OS 在启动的时候，认为是没什么问题的。所以我们一开始就要做安全的初始化工作，从而万一之后被控制了还是可以保护好执行环境。

OS 安全的三个层次

操作系统安全的三个层次



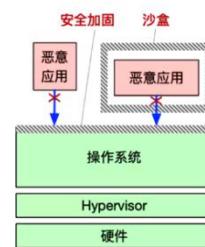
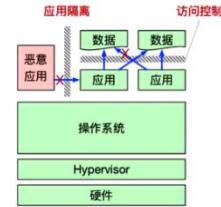
层次一：基于OS的应用隔离与访问控制

- 威胁模型**
 - 操作系统是可信的，能够正常执行且不受攻击
 - 应用程序可能是恶意的，会窃取其他应用数据
 - 应用程序可能存在bug，导致访问其他应用数据
- 应用隔离**
 - 内存数据隔离：依赖进程间不同虚拟地址空间的隔离
 - 文件系统隔离：文件系统是全局的，需限制哪些应用不能访问哪些文件
 - 操作系统提供对文件系统的**访问控制机制**

OS 提供了三百多个 syscall，可能一些不常用的 syscall 中存在 bug，我们可以给应用做 filter，让它只可以使用常用的 30 个 syscall。

层次二：OS对恶意应用的隔离与防御

- 威胁模型**
 - 操作系统存在bug和安全漏洞
 - 操作系统的运行过程依然可信
 - 恶意应用利用操作系统漏洞攻击，获取更高权限或直接窃取其他应用的数据
- 操作系统防御**
 - 防御常见的操作系统bug/漏洞
 - 沙盒机制限制应用的运行



层次三：OS不可信时对应用的保护

- **威胁模型**

- 操作系统不可信，有可能被攻击者完全控制
- 恶意应用可能与操作系统串通发起攻击



- **基于更底层的应用保护**

- 基于Hypervisor的保护：可信基更小
- 基于硬件Enclave的保护：硬件通常更可信

我们希望可行基和攻击面越小越好。这三个层次也是防御纵深的概念，这三个可以共存，并不矛盾。防御纵深越深，攻击要突破的关卡越多。

操作系统安全的三个概念

- **可信计算基 (Trusted Computing Base)**

- 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（硬件上的软件）

- **攻击面 (Attacking Surface)**

- 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 (Defense in-depth)**

- 为系统设置多道防线，为防御增加冗余，以进一步提高攻击难度

访问控制

访问控制包括认证和授权。

访问控制与引用监视器

- **访问控制 (Access Control)**

- 按照访问实体的身份来限制其访问对象的一种机制
- 为了实现对不同应用访问不同数据的权限控制
- 包含“认证”和“授权”两个重要步骤

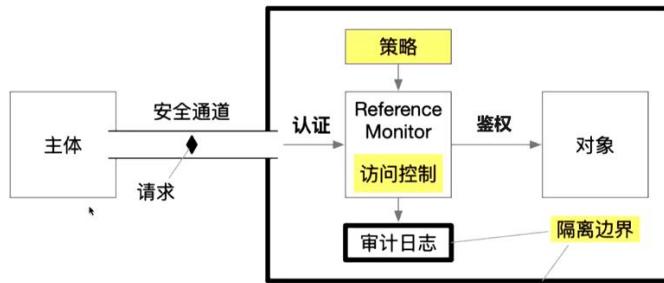
- **引用监视器 (Reference Monitor)**

- 是实现访问控制的一种方式
- 主体必须通过引用 (reference) 的方式间接访问对象
- Reference monitor 位于主体和对象之间，进行检查

整个模型长这样，主体可以是一个进程或者线程。访问对象可以是一个内存或者是一个文件，进程要访问文件不能直接访问磁盘，一定要经过引用监视器 (Reference Monitor)，

也就是我们的文件系统。文件系统会判断主体有没有权限访问对象。

引用监视器 (Reference Monitor) 机制



Reference Monitor 负责两件事：

1. Authentication : 确定发起请求实体的身份，即**认证**
2. Authorization : 确定实体确实拥有访问资源的**权限**，包含**授权**和**鉴权**

认证机制

- 知道什么 (Something you know)
 - 例如密码/口令、手势密码、某个问题的答案等
- 有什么 (Something you have)
 - 例如 USB-key、密码器等实物
- 是什么 (Something you are)
 - 如指纹、虹膜、步态、键盘输入习惯等属于人的一部分

认证：从用户到进程

- 进程与用户之间如何绑定？
 - 每个进程的PCB中均包含了user字段
 - 每个进程都来自于父进程，继承了父进程的user
 - 用户在登录后运行的第一个进程 (shell)，初始化user字段
 - 在Windows下，窗口管理器会扮演类似shell的角色

访问控制列表（ACL， Access Control List）

访问控制列表

- 权限矩阵
 - 对象与实体的关系

	对象-1	对象-1	对象-3
实体-1	读/写	读/执行	读
实体-2		读/执行	读/写
实体-3	读		读/写

- 矩阵有多大？

- 假如系统中有 100 个用户，每种权限用 1 个 bit 来表示，那么每个文件都至少需要 300 个 bit 来表示 100 个用户的 3 种权限
- 假设这些 bit 都保存在 inode 中，通常 inode 的大小为 128-Byte 或 256-Byte，300 个 bit 相当于一个 inode 的 15% 至 30%
- 每当新建一个用户的时候，都必须要更新所有 inode 中的权限 bit，不现实

POSIX 的文件权限

三类用户就是 owner、owner group 和 others，每个用户可以属于不同的 group。打开文件的时候，进行鉴权和授权。

授权机制：POSIX的文件权限

- 将用户分为三类

- 文件拥有者、文件拥有组、其他用户组
- 每个文件只需要用9个bit即可：3种权限（读-写-执行）× 3 类用户

- 何时检查用户权限？

- 每次打开文件时，进行鉴权和授权
 - open()包含可读/可写的参数，OS根据用户组进行检查（鉴权）
 - 引入fd，记录本次打开权限（授权），作为后续操作的参数
- 每次操作文件时，根据fd信息进行检查（鉴权）

在 Windows，有很多的子权限，通过 editor 可以加入更多的 user group。

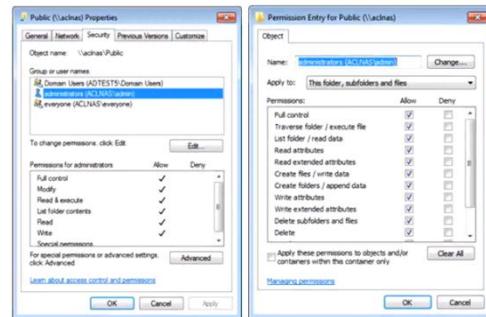
例：Windows的ACL

- Windows使用不同于POSIX的ACL机制

- 以文件和目录为粒度
- 为多个用户和用户组设置不同的权限

- 对比POSIX

- 只有3类用户/组



基于角色的访问控制（RBAC）

这两种方案似乎都有点问题，我们想提出一种更好的方案。我们把人和角色进行分离。比如一个人在公司同时兼任开发、测试和经理多个角色。

用户可以拥有一个或多个角色。RBAC 相对来说可以让角色和权限直观一些。

基于角色的访问控制（RBAC）

- RBAC：将用户（人）与角色解耦的访问控制方法

- Role-Based Access Control
- 提出了角色的概念，与权限直接相关
- 用户通过拥有一个或多个角色，间接地拥有权限
- "用户-角色"，以及"角色-权限"，一般都是多对多的关系

- RBAC的优势

- 设定角色与权限之间的关系比设定用户与权限之间的关系更直观
- 可一次性地更新所有拥有该角色用户的权限，提高了权限更新的效率
- 角色与权限之间的关系比较稳定，而用户和角色之间的关系变化相对频繁
 - 设计者负责设定权限与角色的关系（机制）
 - 管理者只需要配置用户属于哪些角色（策略）

setuid 有点像 RBAC。所有用户的密码都保存在/etc/shadow 里，但是用户又可以修改自己的密码。解决方案就是使用 passwd 的时候，临时变成 root 来运行这个二进制程序。我们加了一个 bit，使得我们可以以 root 身份运行，由 passwd 自己的代码逻辑保证这个用户只能修改自己的密码。这就很危险。

setuid 机制

最小特权级原则：setuid 机制

- 问题：passwd 命令如何工作？
 - 用户有权限使用 passwd 命令修改自己的密码
 - 用户的密码保存在 /etc/shadow 中，用户无权访问
 - 本质上是以文件为单位的权限管理粒度过粗——怎么解决？
- 解决方法：运行 passwd 时使用 root 身份 (RBAC 的思想)
 - 如何保证用户提权为root后只能运行passwd？
 - 在passwd的inode中增加一个SUID位，使得用户仅在执行该程序时才会被提权，执行完后恢复，从而将进程提权的时间降至最小
 - passwd程序本身的逻辑会保证某一个用户只能修改其自身的密码

```
-rwsr-xr-x 1 root 63736 Jul 27 2018 /usr/bin/passwd
```

setuid的安全风险

- setuid 不同于RBAC
 - setuid在Linux下通常用于以root身份运行，拥有的权限，远超过必要
 - 必要权限：用户能够读写 /etc/passwd 文件中的某一行
 - 实际权限：
 - 用户能够访问整个 /etc/passwd 文件
 - 用户（短暂地）拥有root用户的权限
- setuid的安全隐患
 - 一旦 passwd 程序存在漏洞，如 buffer-overflow 导致的返回地址修改，则攻击者很容易以root身份通过ROP运行 execv("/bin/sh")

Capability 权限控制

权限控制的另一种思路：Capability

- **Capability表示一种能力**
 - 例如：读取/foo文件，写入/foo文件，等等
 - 有点像钥匙，能打开某一把锁的话就能进行某个操作
 - 每个进程可以拥有一组能力
- **Capability怎么实现？**
 - 仅仅是一串bit，但必须保存在内核中，否则进程就可以任意伪造
 - 通常保存在进程的PCB中，在进程进行某个操作的时候内核检查
 - 可以把不同Capability的组合对应为ACL中的不同组
 - 因此使用Capability的控制粒度可以很细，而且不需要建立大量的组

Capability的典型操作

1. 服务端通过系统调用创建一个 Capability，获得相应的 ID；
2. 服务端通过系统调用，将此 Capability ID 传递给某个客户端；
3. 客户端通过 IPC 调用服务端的某个服务函数，以 Capability ID 作为参数；
4. IPC 调用过程中，操作系统根据该 Capability ID 检查该客户端是否有权限调用服务端函数，检查通过则切换至服务端继续运行；
5. 服务端执行函数，并将结果返回给客户端。

这个机制和 fd 类似，fd 也可以在不同进程之间传递，可以通过 send message 在不同进程之间传递 fd。

fd与Capability的类似之处

- **文件描述符 fd 可以看做是一类 Capability**
 - 用户不能伪造 fd，必须通过内核打开文件（回顾 file_table/fd_table）
 - fd 只是一个指向保存在内核中数据结构的“指针”
 - 拥有 fd 就拥有了访问对应文件的权限
 - 一个文件可以对应不同 fd，相应的权限可以不同
- **fd 也可以在不同进程之间传递**
 - 父进程可以传递给子进程（回顾pipe）
 - 非父子进程之间可以通过 sendmsg 传递 fd

Linux 的 Capability

这个语义是不允许自定义的，Linux 事先提供了语义。

Linux的Capability

- 提供细粒度控制进程的权限
 - 初衷：解决root用户权限过高的问题
- 与前面说的Capability的不同
 - 语义都是预先由内核定义，而不允许用户进程自定义
 - 不允许传递，而是在创建进程的时候，与该进程相绑定
 - 没有提供 Capability ID，无法通过 ID 索引内核资源进行操作

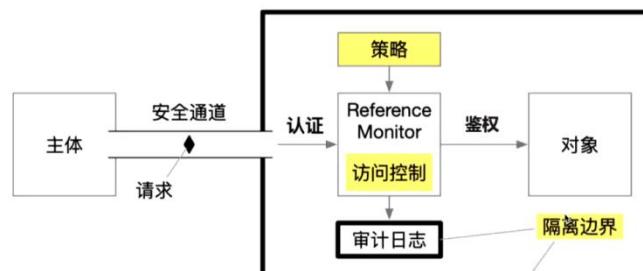
Capability 名称	具体描述
CAP_AUDIT_CONTROL	允许控制内核审计（启用和禁用审计，设置审计过滤规则，获取审计状态和过滤规则）
CAP_AUDIT_READ	允许读取审计日志（通过 multicast netlink socket）

2022/5/19

操作系统攻防

今天我们主要来讨论安全更深的问题，也就是攻防。上节课讲了 OS 安全的三个层次。引用监视器的关键就是不能让一个主体直接访问一个对象，必须中间过程中有个人管着。听上去也是很自然的事情，引用监视器做两件事情，一个是认证你是谁，第二个就是判断你有没有权力去访问资源。

Review: 引用监视器 (Reference Monitor)



Reference Monitor 负责两件事：

1. **Authentication**：确定发起请求实体的身份，即**认证**
2. **Authorization**：确定实体确实拥有访问资源的**权限**，包含**授权**和**鉴权**

对用户认证完，要把身份传递到进程里来。

Review : 认证机制

- **知道什么 (Something you know)**
 - 例如密码/口令、手势密码、某个问题的答案等
- **有什么 (Something you have)**
 - 例如 USB-key、密码器等实物
- **是什么 (Something you are)**
 - 如指纹、虹膜、步态、键盘输入习惯等属于人的一部分
- **用户身份与进程身份的绑定**
 - 每个进程的PCB中均包含user字段，继承自父进程，最早来自登录的shell

认证的时候有两个大的分类，一个是 ACL。就像是我们进考场的时候要拿学生证一样，学生证的作用就是 ACL。访问控制列表分为传统 ACL 和 POSIX 风格。

Review : 访问控制

- **访问控制列表 (ACL)**
 - 传统的ACL (如Windows)
 - POSIX风格 (如Linux)
 - RBAC (Role-Based Access Control)
- **Capability**
 - 类似文件系统的fd
 - 与Linux的capability机制不同

DAC 和 MAC

比如 Linux 下的命令 `chmod 777`，打开所有权限（9个bit全设为1）。但是这真的可以吗？DAC 意味着一个资源的主人有权限设置这个资源对别人的可见与否的权限的。

但是军队也要使用文件系统，设置成绝密的文件只有一小部分人能看，但是看的人并没有权力把它给别人。DAC 的问题就是过于灵活，一个将军如果不小心把文件放到公有目录下，那么其他人都看得到了，这就很危险。

所以在 DAC 以外，还有一个 MAC。在所有用户之上有系统，强制定下了一套规则，任何人都不能改变。只要不破坏 MAC 的规则，那么剩下的可以和 DAC 结合。

DAC与MAC

- **自主访问控制 (DAC: Discretionary Access Control)**
 - 指一个对象的拥有者有权限决定该对象是否可以被其他人访问
 - 例如，文件系统就是一类典型的 DAC
 - 但是对部分场景（如军队）来说，DAC过于灵活
 - 例如，文件拥有者是否真的有权随意设置文件权限？
- **强制访问控制 (MAC: Mandatory Access Control)**
 - 由“系统”增加一些强制的、不可改变的规则
 - 例如，在军队中，如果某个文件设置为机密，那么就算是指挥官也不能把这个文件给没有权限的人看——这个规则是由军法（系统）规定的
 - MAC与DAC可以结合，此时MAC的优先级更高

Bell-LaPadula 模型

Bell-LaPadula 模型就是一个强制访问控制（MAC）的模型，它就是给政府和军队使用的。星属性就是一个4级的士兵不能把任何信息写到3级的文件里。所以第一条规则是限制读的，第二条规则是限制写的。

Bell-LaPadula 模型

- **BLP属于强制访问控制 (MAC) 模型**
 - 一个用于访问控制的状态机模型
 - 目的是为了用于政府、军队等具有严格安全等级的场景
- **BLP 规定了两条 MAC 规则和一条 DAC 规则：**
 - 简单安全属性：某个安全级别的主体无法读取更高安全级别的对象
 - *属性（星属性）：某一安全级别的主体无法写入任何更低安全级别的对象
 - 自主安全属性：使用访问矩阵来规定自主访问控制（DAC）

SELinux

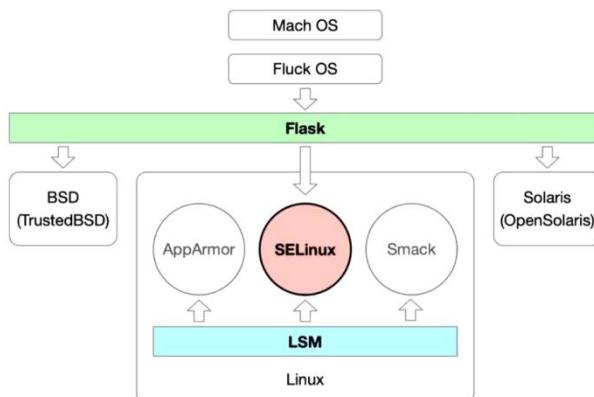
它其实是一套让 OS 提供更灵活的 access control 和更细粒度保护的框架。Linux 里已经有 SELinux 的能力，但是我们平时是不打开的，我们可以添加一些模块来打开它。

SELinux的历史

- SELinux , 由NSA发起 , 2003年并入Linux
 - 是 Flask 安全架构在 Linux 上的实现
 - Flask 是一个 OS 的安全架构 , 可灵活提供不同的安全策略
 - 是一个 Linux 内核的安全模块 (LSM)
 - 在Linux内核的关键代码区域插入了许多 hook进行安全检查
- SELinux 提供一套访问控制的框架
 - 支持不同的安全策略 , 包括强制类型访问 (MAC)

最早的时候是 Mach OS, 然后演变到了 Fluck OS, 然后开发了一套框架 Flask 来支持很多 OS, 这套框架是和 OS 解耦的。有人把 Flask 框架支持到 Linux 里, 所以提出了 LSM 这套机制。OS 的发展一直在遵循插件化的过程, 我们可以通过 insert module 指令插入 kernel module 到内核。LSM (Linux Security Module) 的目标就是 Security。这个接口是埋在内核很多的 check, 比如 if user 是三星的, 而文件是两星的, 我们就不能写。这个操作是我们在 open 这个文件的时候加入的一些 hook。这样的 hook 如果写的差, 可能就分散在几千个地方, 实现的好可能就收敛在几百个地方。

SELinux、Flask与LSM



有了 SELinux 之后，我们允许用户自己设计很多 policy。

SELinux引入的概念

- **用户 (User) : 指系统中的用户**
 - 与 Linux 系统用户并没有关系
- **策略 (Policy) : 一组规则 (Rule) 的集合**
 - 默认是"Targeted"策略，主要对服务进程进行访问控制
 - MLS (Multi-Level Security)，实现了 Bell-LaPadula 模型
 - Minimum，考虑资源消耗，仅应用了一些基础的策略规则，一般用于手机等平台
- **安全上下文 : 是主体和对象的标签 (Label)**
 - 用于访问时的权限检查
 - 可通过"ls -Z"的命令来查看文件对应的安全上下文

一条条 policy 最后就变成了一个主体，是否可以在一个操作上做什么操作。有了这个 policy 之后，当我们访问文件的时候，就可以找到这个 policy，看看 user 是否可以做这个 operation，这是非常 general 的实现。每一条规则都是一个访问向量，保存在安全服务器上。

SELinux的访问向量

- **SELinux 将访问控制抽象为一个问题 :**
 - 一个 < 主体 > 是否可以在一个 < 对象 > 上做一个 < 操作 >
 - 3W: Who, Which (obj), What (operation)
- **AVC: Access Vector Cache**
 - SELinux 会先查询AVC，若查不到，则再查询安全服务器
 - 安全服务器在策略数据库中查找相应的安全上下文进行判断

上下文就是一个标签，是用户、角色、类型和 MLS 层级。MLS 就是类似军官的 MAC 模型。用户登录之后，系统会根据角色分配给用户分配一个安全上下文。每个进程的 type 称为一个 domain。

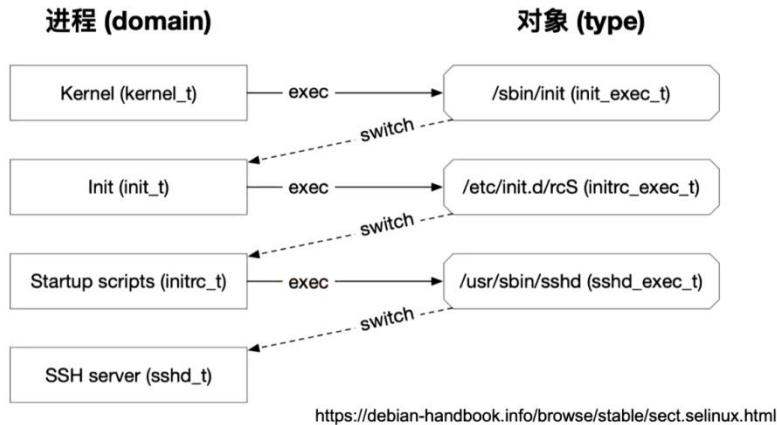
SELinux的安全上下文

- **SELinux本质上是一个标签系统**
 - 所有的主体和对象都对应了各自的标签
- **标签的格式 <用户:角色:类型:MLS层级>**
 - 用户登录后，系统根据角色分配给用户一个安全上下文
 - 类型 (Type) 用于实现访问控制
 - 每个对象都有一个 type
 - 每个进程的type称为 domain
 - 一个角色对应一个domain
 - 重要的服务进程被标记为特定的domain
 - 例如：/usr/sbin/sshd 的类型为 sshd_exec_t

有了这些标签的时候，kernel 去执行 init 的时候，身份就变成了 init_exec_t。被执行的时候，把自己的标签变成 init_t。init_t 可以执行 initrc_exec_t，执行它只会又变成了 initrc_t，它又有能力去执行 sshd_exec_t，就变成了 sshd_t。

如果让 `sshd_t` 直接去执行 `init_exec_t` 是不行的。进程已经被打上了 `sshd_t` 的标签，因为没有 rule，所以它不能执行别的对象。身份就这样一点点被传下去了。

进程的domain与对象的type



我们来看如下的一个例子，这里有两个文件，`page-1` 和 `page-2`。它的标签是 `admin_home_t`，它应该只能 `admin` 的 `home` 目录下。我们做了两个操作，分别是 `cp` 和 `mv` 到 `/var/www/html/` 下。`chown apache: page*` 的意思就是我们把 `page-1` 和 `page-2` 的 `owner` 都换成 `apache`，`page1` 已经变成了 `http` 服务器下可以访问的规则。`cp` 的规则就是会默认打上这个标签，但是 `page2` 是 `mv` 过来的，就不能打开。原因就是 `apache` 应用程序是不能访问标签为 `admin_home_t` 的文件，它只能访问 `httpd_sys_content_t` 标签的文件。

实例

```
[root@CentOS-8 ~]# ls -lZ
total 14
... unconfined_u:object_r:admin_home_t:s0 ... page-1.html
... unconfined_u:object_r:admin_home_t:s0 ... page-2.html
[root@CentOS-8 ~]# cp page-1.html /var/www/html/
[root@CentOS-8 ~]# mv page-2.html /var/www/html/
[root@CentOS-8 ~]# cd /var/www/html/
[root@CentOS-8 html]# chown apache: page*
[root@CentOS-8 html]# ls -lZ
total 12
... unconfined_u:object_r:httpd_sys_content_t:s0 ... index.html
... unconfined_u:object_r:httpd_sys_content_t:s0 ... page-1.html
... unconfined_u:object_r:admin_home_t:s0 ... page-2.html
```

`page-2.html` 被标记为 `admin_home_t`，即使被错误放入网页目录，也无法被 `apache` 访问



操作系统的漏洞

OS 既然有 `bug` 和漏洞。我们要对漏洞本身做分析，我们反过来想怎么从漏洞角度去消灭掉它。我们可以从三个角度来看漏洞。一个是攻击者利用的漏洞、攻击的模块、攻击的目的。很多漏洞我们都很熟悉，比如堆栈溢出、整形溢出等。比如我们 `kalloc` 一块内存有些地方没有初始化就读了，未初始化的读是很危险的。

漏洞分类的三个角度

- **漏洞类型（指攻击所利用的漏洞类型）**
 - 栈/堆缓冲区溢出、整形溢出、空指针/指针计算错误、内存暴露、use-after-free、double-free、未初始化读取、格式化字符串错误、竞争条件错误、参数检查错误、认证检查错误，等等等等
- **攻击模块（指攻击所利用漏洞的所在的模块）**
 - 调度模块、内存管理模块、通信模块、文件系统、设备驱动等
- **攻击效果（指攻击的目的或攻击导致的结果）**
 - 执行任意代码、内存篡改、窃取数据、拒绝服务、破坏硬件等

有了这三个角度之后，我们就可以针对性地去判断。

操作系统安全很难指标化

- **如何评价某个操作系统比另一个更安全？**
 - 缺陷密度？缺陷数量？代码行数？
- **缺陷密度**
 - 常用指标：每 1,000 行代码的平均缺陷数量
 - Linux 的缺陷密度近年来已经小于 0.5
 - GPU 驱动的缺陷密度仅为 0.19，160 个缺陷
 - SMACK (Linux 的一个安全模块) 高达 1.11，6 个缺陷

Windows XP 不开源，测试的人就少，显然不能用漏洞的绝对值去衡量 OS 谁更安全。我们只知道代码行数越少越安全。

操作系统安全很难指标化

- **缺陷数量**
 - 缺陷的编号方法
 - CVE (Common Vulnerabilities and Exposures)
 - 一种常见的漏洞编号方式，CVE-2020-10757
 - 国家计算机网络应急处理协调中心的 CNCVE 编号
 - 国家信息安全漏洞共享平台的 CNVD 编号
 - 中国信息安全测评中心的 CNNVD 编号
 - 相对值直接比较也缺乏说服力
 - Linux 内核的 CVE 数量目前排在第 3 位 (2,357 个)
 - Windows XP 则排第 28 位 (741 个)，比 Linux 更安全？

操作系统内核攻防

虽然我们对长度进行了检查，但是还远远不够，count 仍然可能移除。

整形溢出漏洞

```
unsigned long count = /* from user space */;
if (count > 1<<30)
    return -EINVAL;
table = vmalloc(sizeof(struct
    ↳ rps_dev_flow_table) +
    count * sizeof(struct
    ↳ rps_dev_flow));
...
for (i = 0; i < count; i++)
    table->flow[i] = ...;
```

防御方法：增加对溢出的检查代码；利用自动化工具查找并修复

攻击者可以在用户态部署恶意代码，然后在内核态造成溢出，返回到用户态的恶意代码。

Return-to-user 攻击（ret2usr）

Return-to-user 攻击（ret2usr）

- **内核错误地运行了用户态的代码**
 - 由于内核与应用程序共享同一个页表，内核运行时可以任意访问用户态的虚拟地址空间，内核可能执行位于用户态的代码
- **攻击者的常用方法**
 - 先在用户态中初始加载一段恶意代码，然后利用内核的某个漏洞，修改内核中的某个函数指针指向这段恶意代码的地址
 - 也可以利用内核的栈溢出漏洞，覆盖栈上的返回地址为恶意代码的地址，使内核在执行 ret 指令时跳转到位于用户态的代码

方法 2 需要 FLUSH TLB，会造成大量的 TLB miss。

ret2usr 攻击的防御方法

- **方法一：仔细检查内核中的每个函数指针**
 - 需对内核所有模块进行检查，很难做到 100% 的覆盖率
- **方法二：在陷入内核时修改页表，将用户态所有的内存都标记为不可执行**
 - 由于修改页表后必须要刷新 TLB 才能生效，因此修改页表、刷新 TLB，以及后续运行触发 TLB miss 都会导致性能下降
 - 在返回用户态之前必须将页表恢复，并再次刷掉 TLB，这样又会导致用户态执行时出现 TLB miss，因此对性能的影响非常大
- **方法三：硬件保证 CPU 处于内核态时不得运行任何用户态的代码**
 - 如 Intel 的 SMEP (Supervisor Mode Execution Prevention) 技术
 - ARM 同样有类似 SMEP 的技术，称为 PNX (Privileged eXecute-Never)

当一个用户态应用程序用到 malloc 的 page 后，就会触发 page fault。buddy system 分配出来这一块内存，但是分配给应用程序后，内存的直接映射依然存在。所以，依然可以通过直接映射的方式访问这块地址。所以，用户态和内核的直接映射都用到了这个

地址。

所以应用程序把恶意代码放在这块内存，然后想办法在内核中返回到内核的这块映射的地址。

SMEP 不能完全解决 ret2usr : ret2dir

- **操作系统管理内存的方法“直接映射”**

- 将一部分或所有的物理内存映射到一段连续的内核态虚拟地址空间
- 分配给应用程序后，直接映射依然存在
- 因此，同一块物理内存系统中有多个虚拟地址
 - 例如，某个内存页分配给了应用程序，那么内核既可以通过应用程序的虚拟地址访问（前提是内核与应用在一个地址空间），也可以通过直接映射的虚拟地址访问

- **基于直接映射的攻击，可绕过SMEP**

- 攻击者首先推算出位于用户态的恶意代码在内核直接映射区域的虚拟地址，然后在 ret2usr 攻击中让内核跳转到该地址执行（内容依然为攻击者控制）
- 攻击成功还有一个前提：直接映射区域必须是可执行的
 - 在 3.8.13 以及之前的 Linux 版本，将直接映射区域的权限设置为了“可读-可写-可执行”
- 这种利用直接映射区域的 ret2usr 攻击被称为“ret2dir”攻击

我们在直接映射的时候标记为不可执行，并且内核不能使用直接映射中的页来执行。

Rootkit

如果 hook 一个指针，是很难被发现的。

Rootkit：获取内核权限的恶意代码

- **Rootkit 是指以得到 root 权限为目的的恶意软件**

- Rootkit 可以运行在用户态，也可以运行在内核态

- **用户态的Rootkit**

- 可以将自己注入到某个具有 root 权限的进程中，并接收攻击者的命令

- **内核态的Rootkit**

- 可以 hook 某个内核中的关键函数，从而在该函数被调用时触发运行

- 可以是以内核线程的方式运行

- 可以修改内核中的系统调用表，用恶意代码来替换掉正常的系统调用

KASLR：内核地址布局随机化

我们发现恶意攻击经常需要一些指针，所以我们就需要内核地址布局随机化。但是运行时修改地址，overhead 比较高。

KASLR：内核地址布局随机化

- **ASLR 与 KASLR**
 - ASLR 通过随机化地址空间布局来提高系统攻击难度
 - KASLR 是对内核启用地址随机化
- **KASLR 可缓解 ret2dir 攻击**
 - 攻击者需要知道用户态恶意代码在内核中直接映射区域的地址
 - KASLR 通过将内核的虚拟地址布局进行随机化，使攻击者准确定位内核地址的难度大大提升

侧信道与隐秘信道

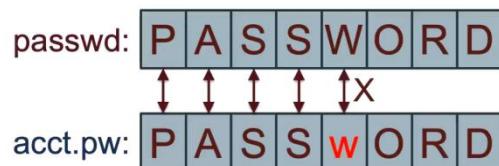
什么是隐秘信道？

- **隐秘信道 (Covert Channel)**
 - 原本无法直接通信的两方，通过原本不被用于通信的机制进行数据传输
 - 常见的隐秘信道：时间、功耗、电磁泄露、声音等
- **例：消费记录的应用 A，在没有网络的情况下如何把数据发出去？**
 - 假设有一个应用 B 运行在同一个手机
 - 若 A 可播放声音，B 可录音，则 A 把数据编码为声音发送给 B
 - 若 A 可打开闪光灯，B 可摄像，则 A 把数据编码为光的闪烁长短与频率发送给 B
 - 若 A 可震动，B 可访问运动传感器，则 A 把数据编码为震动频率发送给 B
 - 若 B 可访问 CPU 温度，则 A 可长时间运行计算密集代码，CPU 升温表示 1，反之为 0
 - ...

一个个猜，猜对了时间更久，这就是时间信道。

Review: Guessing Password (Tenex)

```
checkpw (user, passwd):  
    acct = accounts[user]  
    for i in range(0, len(acct.pw)):  
        if acct.pw[i] != passwd[i]:  
            return False  
    return True
```



侧信道可能就是把被攻击者正常操作所释放的一些信息捕获，从而窃取数据。所以侧信道攻击的难度更大。

侧信道与隐秘信道的关系

- **侧信道与隐秘信道很类似**
 - 两者都使用类似的方式（信道）进行数据的传递
- **侧信道攻击和隐秘信道攻击的不同**
 - 隐秘信道攻击：两方是互相串通的，其目的就是为了将信息从一方传给另一方
 - 侧信道攻击：一方是攻击者，另一方是被攻击者，攻击者窃取被攻击者的数据
 - 即被攻击者无意间通过侧信道泄露了自己的数据

缓存信道

我们想猜出 i 是不是等于 0，如果运行 `func_b` 的时间比较久，说明出现了 cache miss，那么说明之前运行了 `func_a`，也就是 $i=0$ 。

缓存信道（Cache Channel）

- **利用缓存的状态推测执行的信息**
 - 例如：可根据 `func_a` 还是 `func_b` 的代码在缓存中，来判断 i 的值
 - 判断方法：`func_a` 和 `func_b` 的时延
- **常见的四种攻击方式**
 - Flush+reload
 - Flush+flush
 - Prime+probe
 - Evict+time

```
if (i == 0)
    func_a();
else
    func_b();
```

Flush+Reload

- 假设：攻击进程和目标进程共享一块内存
- 攻击步骤
 - 1. 攻击进程首先将 cache 清空（如：不断访问其他内存占满cache或直接flush）
 - 2. 等待目标进程执行
 - 3. 攻击进程访问共享内存中的某个变量，并记录访问的时间
 - 若时间长，则表示 cache miss，意味着目标进程没有访问过该变量
 - 若时间短，则表示 cache hit，意味着目标进程访问过该变量
- 特点分析
 - 优点：可以跨CPU核，甚至跨多个CPU；噪音低
 - 缺点：攻击准备难度高，需构造与目标进程完全相同的内存页

Flush+Flush

- 基于缓存刷新时间（如clflush）来推测数据在缓存中的状态
 - 1. 攻击进程首先将 cache 清空（Flush）
 - 2. 等待目标进程执行
 - 3. 运行clflush再次清空不同的缓存区域
 - 若时间较短说明缓存中无数据
 - 时间较长则说明缓存中有数据，目标进程曾访问对应的内存
- 特点分析
 - 优点：只需清空缓存而不需实际访存，因此具有一定的隐蔽性
 - 缺点：clflush对于有数据和无数据的时间差异不明显，攻击精度不高

Evict+Reload

- 场景：CPU没有 clflush 指令
 - 1. 将关键数据所在的 cache set 都替换成攻击进程的数据
 - 前提：攻击者知道关键数据的内存地址，以及CPU上内存-cache的映射机制
 - 2. 等待目标进程执行
 - 3. 访问 cache set 中的某个数据
 - 若时间很短，说明目标进程没有将该数据 evict，即没有访问过某个关键数据
 - 反之，则说明目标进程访问了某个关键数据
- 特点分析
 - 优点：无需依赖 flush 指令
 - 缺点：无法支持动态分配的内存；需要了解 LLC 的 eviction 策略；Cache 必须是 inclusive；无法很好地支持多 CPU

Flush+Reload 要共享内存，而 Prime-Probe 要求共享 Cache。

Prime+Probe

- 攻击的具体步骤如下：
 - 1. 攻击进程用自己的数据将 cache set 填满 (Prime)
 - 2. 等待目标进程执行
 - 3. 再次访问自己的数据
 - 若时间很短，说明目标进程没有将该数据 evict，即没有访问过某个关键数据
 - 反之，则说明目标进程访问了某个关键数据
- 特点分析：
 - 优点：不需要共享内存；支持动态和静态分配的内存
 - 缺点：噪音更多；需要考虑 LLC 的实现细节，如组相连等；Cache 必须是 inclusive；无法很好地支持多CPU；需要首先定位目标进程使用的cache set

侧信道攻击的防御

- 侧信道攻击很难被完全防御住，根本原因在于共享
 - 当被攻击者在做了某个操作后，对系统整体产生了影响
 - 这个影响能够被使用同样系统的攻击者发现，那么就构成了一个最简单的侧信道：发现影响和没发现影响（即做了操作和没做操作）可以被编码为 0 和 1
- 防御侧信道的根本方法：不共享
 - 将攻击者和被攻击者运行在完全隔离的物理主机，使其没有任何共享，包括计算硬件、网络，甚至空间（光、温度、声音）
 - 更实际的方法是针对常见攻击进行防御

常量时间算法

解决方法就是让算法的运行时间固定。

常量时间 (Constant Time) 算法

- 算法的运行时间与输入无关
 - 无法通过运行时间得到与输入相关的任何信息
 - 代码执行没有分支跳转
- 常见的实现方法：cmov
 - Conditional MOV
- 缺点：计算变得更慢
 - 需要做两份运算

```
/* 传统实现方式 */
if (secret == 0)
    x = a + b;
else
    x = a / b;

/* 常量时间实现方式 */
v1 = a + b;
v2 = a / b;
cond = (secret == 0)
x = cmov(cond, v1, v2)
```

2022/5/24

硬件辅助系统安全

OS 在硬件和应用中间，通常 OS 不信任应用，比如把它放在沙盒里、隔离、检查参数等。

不经意随机访问内存（ORAM）

这个名字很奇怪，RAM 就是 DRAM。不经意本质上就是说把访存行为和程序执行过程解耦。假设我们现在是一个 memory controller，我们自己做一个小硬件，底下是内存的金手指（金属触点），上面是一个内存插槽。所以它先把内存插到自己身上，然后再把自己插到内存的金属触点，它就成为了一个中间人，可以看到任何读写操作和数据（加密）。

我们可以通过访问了多大的、哪些地方的数据来推测出被攻击者做了什么事情。

不经意随机访问内存（ORAM）

- **ORAM 将访存行为与程序执行过程解耦**
 - 攻击者即使能够观察到所有的访存请求，也无法反推出与程序执行相关的信息
- **最简单的实现：定时、定量、定位的访问方式**
 - 无论实际是否有访存需求，均以**固定周期**，访问**固定位置**，每次访问**固定的大小**
 - 例如，CPU 顺序循环访问所有的有效内存区域，程序按需获得真正想要访问的数据，若还没访问到则等待，若已经访问过了则等待下次循环
 - 类似上海和北京之间的高铁，无论乘客是谁，都按照时刻表运行，哪怕有时候位子没坐满也发车，因此根据高铁的班次并不能反推出谁坐了高铁
- **ORAM 会引入很大的额外负载**
 - 产生大量的无效内存访问，导致有效访存的吞吐率下降
 - 访存需要等待一定的时刻，导致时延大幅度增加

案例：MELTDOWN

我们来讲一个具体的例子，meltdown 攻击。曾经我们认为侧信道攻击是很难的，防御住物理攻击，我们认为 cache 判断是很难的。直到 2018 年出现了 meltdown 攻击，一下子把学术界和工业界的人都震动了，阿里的人半夜三更被叫起来修补丁，然后就开始各种性能可能降低 30% 的升级。当时体系结构的人立马聚在一起讨论防御这个攻击。

这个漏洞的效果就是允许应用程序读取任意内核的内存数据，这个很显然是非常严重的问题。内核有一段 memory 做直接映射的，也就是把所有物理内存都映射到了这块虚拟内存里。这就会导致应用程序可以看到所有物理内存数据，这会导致云厂商的信任基石彻底土崩瓦解。

它利用了 CPU 的投机执行机制，高端芯片都是有这个的。

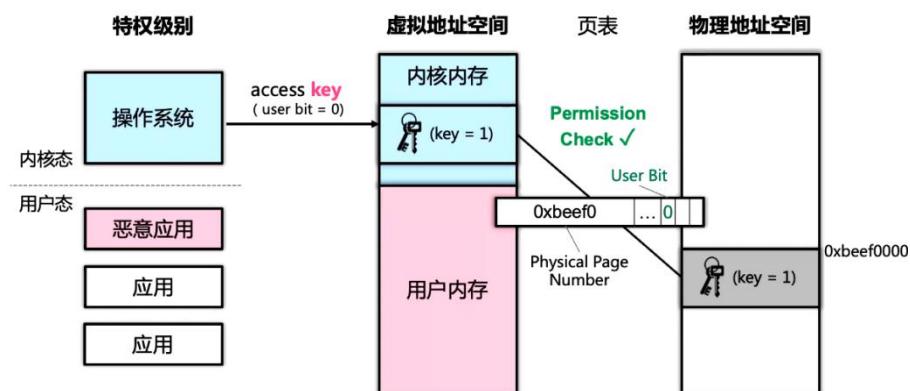
Meltdown漏洞

- **Meltdown (熔断) 漏洞**
 - 和 Spectre (幽灵) 漏洞一同被公布
 - 在 2017 年被发现，在 2018 年被公布
- **效果：允许应用程序读取任意内核内存数据**

- 利用了 CPU 的投机执行机制
 - 漏洞利用简单、攻击效果显著
- 几乎所有的主流 CPU 都受到了影响
 - 包括 Intel、AMD 和部分 ARM 处理器
- 许多软件厂商不得不紧急打补丁并做出对应的防御措施

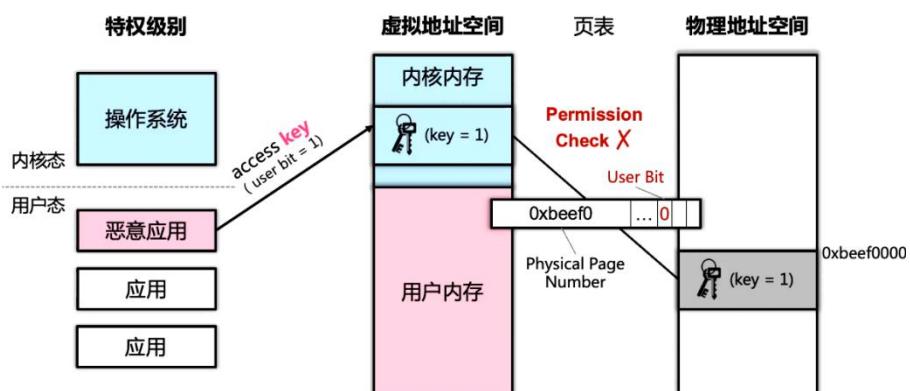
我们知道内核数据的映射方式是不一样的，在页表的 PTE 上有一个 bit。OS 访问 key 的时候，CPU 处于内核态，显然可以访问这个关键数据。

背景-1：基于页表权限的内核数据保护



恶意程序要访问的时候，MMU 检查发现没有权限，就禁止这个操作。

背景-1：基于页表权限的内核数据保护



现代 CPU 为了执行效率，内部有多个执行单元，不会等待指令结束再发。所有的指令 in-order commit，但是投机执行的操作会提前。比如 if 指令执行错了，CPU 都会假设 if 是对

的，然后去执行。

这种好处就是如果 if 是对的，那么后面的指令就做完了。但是 if 是错的话，后面的指令就白做了，不过这就稍微浪费一点电而已。有些状态是很难被 CPU 回滚的。

背景-2：CPU投机执行（预测执行）

- **CPU为了性能会进行投机执行**
 - 处理器内部会并发执行多条指令，无依赖关系的邻近指令在处理器内部的执行顺序会被打乱，部分指令会被提前
- **问：若提前执行了错误指令怎么办？**
 - 如前一条指令异常，处理器投机执行了后续本不应被执行的指令
- **答：执行结果丢弃/回滚**
 - 对于不应被执行的指令，处理器会丢弃/回滚其对寄存器、内存等执行状态的修改
- **复杂的CPU能确保所有状态均正确丢弃/回滚吗？不能**

这个执行不会被 commit，但是跨权限的内存执行也会被执行到。所以就是我们应用程序尝试在一个 if 之后跨权限访问内核段的一个地址，但是这个 if 是 False，但是此时还没有算出来。CPU 就先去执行了这个访存指令，这会导致 CPU 的 cache 发生变化。此时 if 结果为 False，CPU 就取消掉了这个指令。这就导致了 Cache 的状态发生了变化，理论上我们应该把 Cache 的状态也回滚。但是 CPU Cache 的状态回滚很复杂，比如发生了 eviction, rollback 就很麻烦很麻烦，而且访问内存就变慢了，我们要记录在临时的地方，导致很多复杂性。

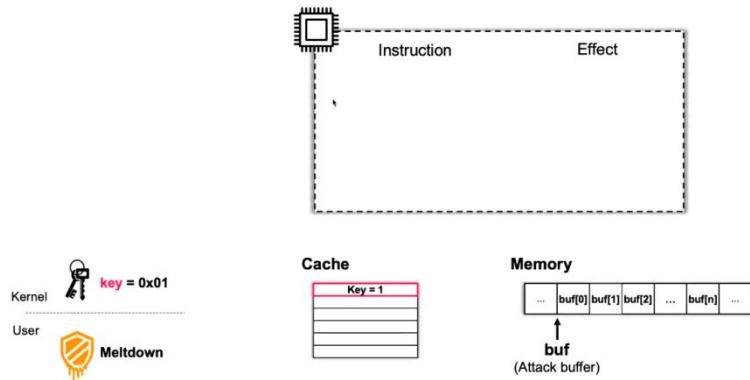
于是，它就可以根据 cache 的变化，反推出我们执行了哪个执行。

Meltdown漏洞原理

- **起因：迟到的内存访问异常**
 - 在投机执行期间，跨权限的内存访问不会立刻触发异常，而是仍会继续执行后续指令
- **故障：遗漏的缓存状态回滚**
 - 当硬件抛出内存访问异常时，CPU理应回滚被错误执行指令对所有状态的修改
 - **但是，实际上CPU未回滚被错误执行指令对CPU缓存的状态修改**
- **结果：应用可任意读取操作系统内存**
 - 利用缓存隐秘信道，窃取非法访问到的内核数据

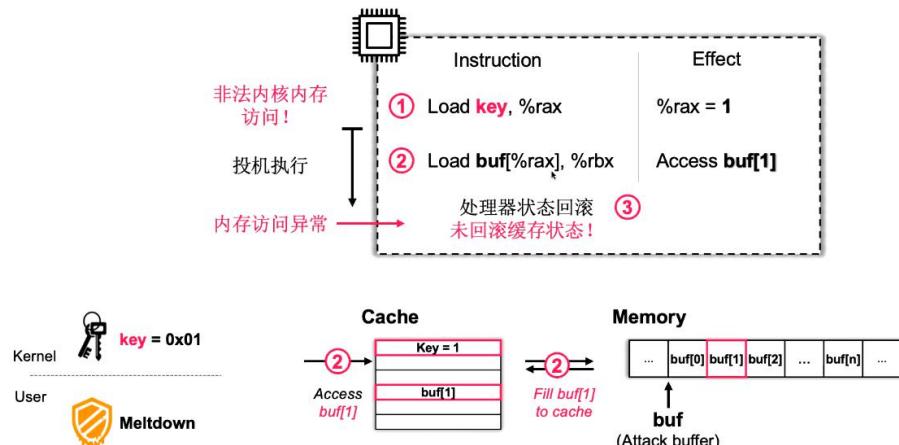
我们来看如下的一个例子：

Meltdown漏洞原理



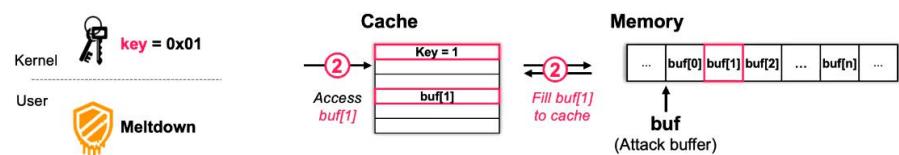
比如我们现在投机执行了，加载 key，并且把 $buf[key]$ 的位置读入，这样的话如果 $key=1$ ，我们 cache 中就有了 $buf[1]$ ；如果 $key=0$ ，我们 cache 就有了 $buf[0]$ ，并且它们的位置是不一样的。我们可以在组相连的粒度来判断到底哪个 buffer 被修改了。

Meltdown漏洞原理



Meltdown漏洞原理

- 通过未回滚缓存状态窃取目标数据
 - 使用目标数据作为索引访问攻击数组 buf (指令2)
 - 被访问元素会被加载到缓存中
 - 异常发生后，根据数组元素是否被缓存，窃取目标数据
 - 若buf中第 i 个元素在缓存中，则目标数据 key = i



我们可以一个个 bit 偷出来，1 秒钟可以偷几百 K。这种攻击叫做隐秘信道。串通体现在 CPU 投机执行的错误指令，在 100 个 cycle 的时间窗口里可以看到内核的数据，但是它们不能通过正常操作传递出来，但是它们和外面的攻击者可以通过 cache channel 来串通。

Meltdown漏洞危害

- 允许对内核内存的随意访问
 - 用户态与内核态共享同一虚拟地址空间，仅通过页表中的权限位进行内核内存的隔离保护
 - 利用Meltdown可随意读取任意内核内存
- 允许对所有内存资源的随意访问
 - 内核地址空间大多映射了所有物理内存
 - 如Linux中的direct map区域直接映射了整个物理内存区域
 - 利用Meltdown可对direct map区域进行读取，从而访问所有内存数据

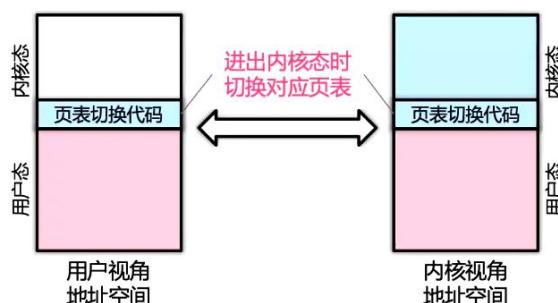
Meltdown漏洞的软件防御方法

- 观察：Meltdown仅能够绕过页表中的用户/内核权限位
 - Meltdown漏洞能够允许攻击者无视“**内存权限**”限制，访问内核内存数据
 - Meltdown漏洞无法访问“**无内存映射**”的内存数据
 - 如一个虚拟地址在页表中无映射，Meltdown漏洞则无数据可拿
- 思路：去除内核地址在用户态的映射
 - 为**用户态构建专用页表**，将内核地址在用户态的映射去除
 - 攻击者利用Meltdown访问内核地址时，页表无法定位目标物理地址

KPTI: Linux 的 Meltdown 漏洞防御机制

用户态是知道内核态的虚拟地址的，但是没有办法访问。如果内核态和用户态是两张页表，那就没有这个问题了。

- KPTI (Kernel Page Table Isolation)
 - 将用户态与内核态所用页表分离
 - 用户页表：仅映射用户地址空间与部分内核地址空间
 - 内核页表：映射包括用户与内核的全部地址空间（与原有一致）



但是这会导致显著的 (~30%) 的性能损失。

KPTI造成的性能损失

- **页表切换导致显著的性能损失**
 - 切换操作本身的时延增加
 - 切换操作：旧页表保存 -> 新页表加载 -> 修改CR3 -> 刷新TLB
 - 进出内核均需切换页表并刷新TLB，额外指令导致切换时延增长
 - 操作发生在系统关键路径
 - 切换后的运行时性能损失
 - TLB刷新导致TLB未命中率提升，直接影响程序执行性能
 - 部分应用性能降低可达30%

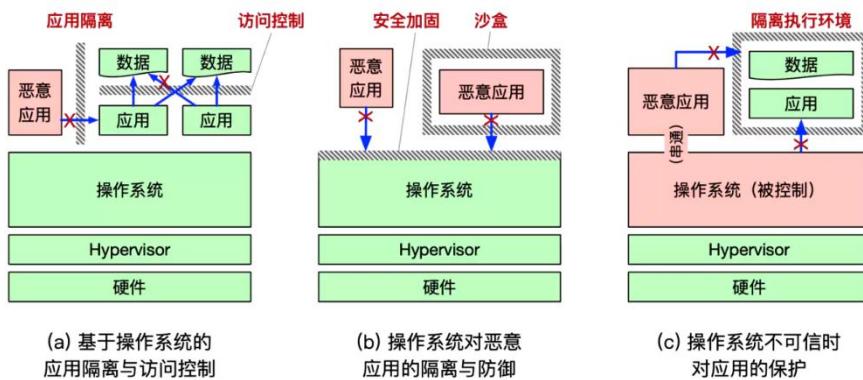
总结

- **Meltdown漏洞**
 - 综合利用硬件漏洞与侧信道的全新攻击方法
 - 影响几乎所有的主流处理器（包括Intel、AMD和部分ARM）
 - 绕过页表中的内核权限，直接窃取任意内核数据
- **KPTI (Kernel Page Table Isolation)**
 - 通过分离用户态与内核态页表，进行Meltdown漏洞防御
 - Meltdown无法从无映射的虚拟地址窃取数据
 - 利用ASID等硬件特性提升性能
 - 消除内核进出时非必要的TLB刷新操作

课后练习（选做）

- **Meltdown漏洞利用**
 - 查看自己的Linux环境是否启用了KPTI
 - 关闭KPTI，尝试在自己的电脑上复现Meltdown攻击
 - 参考PoC：<https://github.com/paboldin/meltdown-exploit>
- **了解Spectre（幽灵）漏洞**
 - Spectre漏洞与Meltdown漏洞的原理有何异同？
 - 尝试在自己的电脑上复现Spectre攻击
 - 参考文献：“Spectre Attacks: Exploiting Speculative Execution”
 - 参考PoC：<https://github.com/flxwu/spectre-attack-demo>

回顾：操作系统安全的三个层次



当操作系统不再可信……

OS 一定有 BUG 的，所以势必可能被攻击。

为什么要假设操作系统是恶意的？

- **系统的复杂性**
 - 软件：恶意软件，OS本身可能存在漏洞
 - 硬件：外设越来越智能，本身可能存在漏洞，甚至是恶意构造
 - 环境：云计算环境、IoT设备，面临这更复杂多变的
 - 人：运维外包（如云计算等）导致接触计算机的人更复杂
- **一种更简单的威胁模型**
 - “除了应用，别的都不可信”

恶意操作系统如何攻击应用？

- **应用的攻击面**
 - 同层：其他应用程序
 - 底层：操作系统、Hypervisor、硬件
- **操作系统窃取应用的数据**
 - 操作系统控制着页表，可直接映射应用的内存并读取数据
- **操作系统改变应用的执行**
 - 操作系统控制着页表，可直接在应用内部新映射一段恶意代码
 - 操作系统可任意改变程序的RIP，劫持其执行流

现实中的类似攻击

- **系统对应用的攻击**

- 攻击者获取具有root权限的bash (如利用sticky位程序的漏洞)
- 攻击者获得insmod的权限 , 在内核中插入恶意的模块
- 攻击者篡改内核的文件 , 下次启动后加载
- 攻击者获得kexec的权限 , 动态执行另一个内核
- 攻击者获得hypervisor的权限 , 从更底层发起攻击
- 攻击者能够控制某个设备 (如智能网卡) , 直接访问物理内存
-

一种新的威胁模型：安全处理器

应用程序都是运行在 user ISA 上的。我们希望构造出一个很小的环境，这个环境只使用 user-level 的 ISA，只信任 CPU 的计算逻辑（cache+运算单位）。这一小块就是 Enclave（飞地）。

一种新的威胁模型：安全处理器

- **不信任CPU外的硬件**

- 包括内存 (DRAM) 、设备、网络

- **仅信任CPU**

- 包括cache、所有计算逻辑 (Anyway , 总得信任CPU吧...)

- **Enclave (飞地)**

- 又称为可信执行环境 , TEE (Trusted Execution Environment)
 - 什么是“可信”？信什么呢？

Enclave/TEE：可信执行环境

- **Enclave/TEE的定义**

- Enclave , 又称“可信执行环境” (TEE , Trusted Execution Environment) , 是计算机系统中一块通过底层软硬件构造的安全区域 , 通过保证加载到该区域的代码和数据的完整性和隐私性 , 实现对代码执行与数据资产的保护 —— Wikipedia

- **Enclave的两个主要功能**

- 远程认证 : 验证远程节点是否为加载了合法代码的Enclave
- 隔离运行 : Enclave外无法访问Enclave内部的数据

- **Enclave带来的能力 : 限制访问数据的软件**

- 可保证数据只在提前被认证的合法节点间流动
 - 合法节点 : 部署了合法软件的节点

Enclave/TEE在工业界发展迅速



- 多家云厂商利用TEE/Enclave保护数据

- 2018年，Microsoft Azure 基于Intel SGX推出Confidential Computing
- 2019年，Amazon 推出了Nitro Enclave保护用户的关键数据
- 2020年，Google Cloud 基于AMD SEV推出加密虚拟机Secure VM

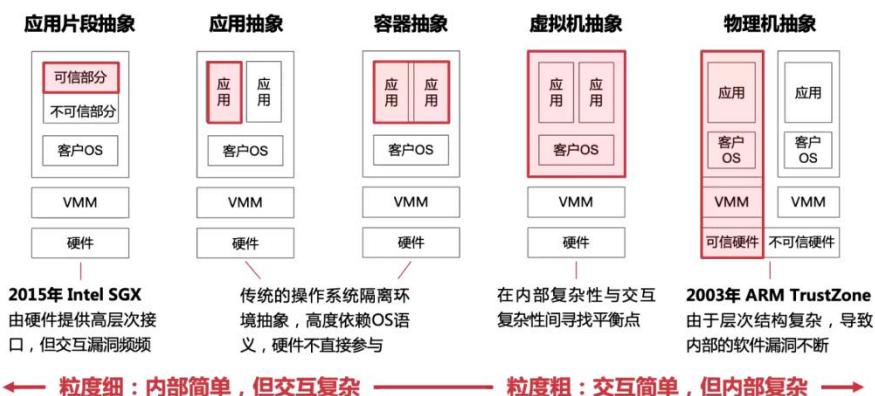
- 2019年8月，多家公司成立机密计算联盟

- 国内公司包括：华为、阿里云、腾讯、百度、字节跳动等
- 国外公司包括：Arm、AMD、Intel、Redhat、Facebook、Google等



有了这个之后，enclave 就有不同的粒度。比如 svc 不能在 Intel SGX 中运行，需要在外部运行。好处就是在 enclave 中运行的代码比较少，比较安全。但是它需要和 OS 交互，这可能暴露 access pattern。第二种就是应用或者容器。第三个就是虚拟机隔离，虚拟机内部有自己的交互，对外就是 trap & emulate，这是比较难去反推用户数据的。

硬件Enclave提供不同粒度的隔离环境



ENCLAVE 案例分析：Intel SGX

Intel SGX

- **SGX: Software Guard eXtension**
 - 2015年首次引入Intel Skylake架构
 - 保护程序和代码在运行时的安全 (data in-run)
 - 其他安全包括：存储时安全和传输时安全
- **关键技术**
 - Enclave内部与外部的隔离
 - 内存加密与完整性保护
 - 远程验证

内存是密文，进 CPU 的时候做一次解密。

硬件内存加密与保护机制

- **硬件加密保护隐私性**
 - CPU外皆为密文，包括内存、存储、网络等
 - CPU内部为明文，包括各级Cache与寄存器
 - 数据进出CPU时，由进行加密和解密操作
- **硬件Merkle Tree保护完整性**
 - 对内存中数据计算一级hash，对一级hash计算二级hash，形成树
 - CPU内部仅保存root hash，其它hash保存在不可信的内存中
 - 当内存中的数据被修改时，更新Merkle Tree

OS 看得见，但不能访问这块内存。当一个 Enclave 内存不够了，OS 可以用 swap 指令把 4k 数据从 EPC 里 copy 到外面，这个数据是加密且有哈希保护的。

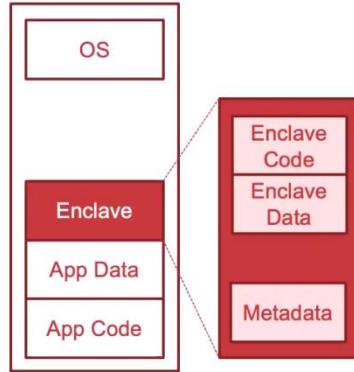
EPC (Enclave Page Cache)

- **CPU预留一部分内存，仅允许Enclave访问**
 - 连续的128MB/256MB，这部分不会暴露给软件操作
 - 全部加密，并保证数据的完整性（即无法篡改）
 - 可能的篡改方法：通过总线直接修改
- **操作系统负责将EPC映射至Enclave中**
 - 操作系统也可以触发swap，将数据从EPC交换到DRAM中
 - 由专门的硬件指令来进行swap，粒度为一个内存页（4K）
 - 注意：页表依然由不可信的操作系统控制

Enclave 能做到：应用从外面 load/store Enclave 里面的代码，CPU 会拒绝掉。但是 Enclave 如果想把数据写到外面来，CPU 不会被拒绝掉。如果 Enclave 有漏洞，不小心把数据写出去了，那就数据泄露了。所以我们需要用软件的办法来解决这个事情。在这种情况下，Enclave 里面有一条 store 写到了 200~300（Enclave 的规定范围）之外。所以，我们只需要在 store 前做一个 check 就行了。

Enclave与进程的关系

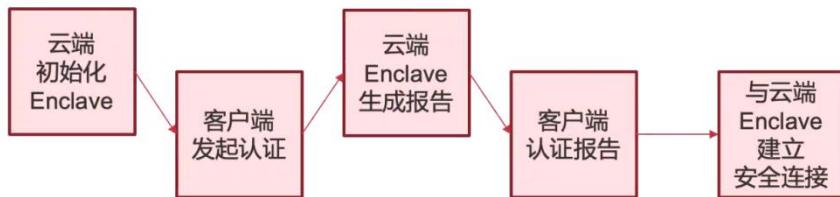
- **Enclave是进程的一部分**
 - Enclave内外共享一个虚拟地址空间
 - Enclave内部可以访问外部的内存
 - 反之则不行
- **创建Enclave的过程**
 1. OS创建进程
 2. OS分配虚拟地址空间
 3. OS将Enclave的code加载到EPC中
 - 并将EPC映射到Enclave的虚拟地址
 - 循环3，完成所有code加载和映射
 4. 完成进程创建



发起认证里发了一个随机数，阿里云拿到了这个随机数以后，用 enclave 的哈希加上随机数用硬件签名，产生一个 report 发给我。我并不知道这个 report 是真的还是假的，我们必须找另一个人去验证这个 report。也就是我们去找 intel 验证一下这个 report 是真的还是假的。

远程验证 (Remote Attestation)

- **要解决的问题：如何远程判断某个主体是Enclave？**
 - 例如，如何判断某个在云端的服务运行环境是安全的
 - 必须在认证之后，再进行下一步的操作，例如发送数据



硬件内存加密

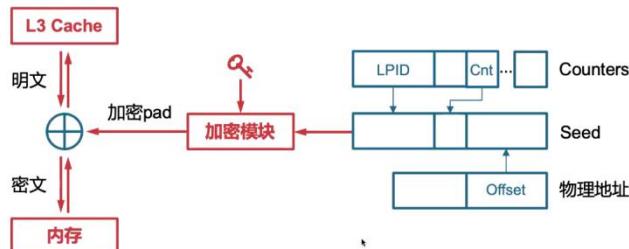
多密钥加密就是不同的地方的相同数据加密，密文也不一样，但是我们密钥就太多了。

硬件内存加密

- **加密的最小单位**
 - Cache line : 64Byte (512bit)
- **方法一：单密钥加密**
 - 缺点：同样的明文会产生同样的密文
- **方法二：多密钥加密**
 - 缺点：如何保存这些密钥 ? CPU内部放不下
- **方法三：单密钥 + 多 seed**
 - 为每个cache line单独生成一个seed，用密钥加密后，对数据进行异或

seed 的组成是 LPID 和物理地址的 offset 组成的。最终读内存的时候，只需要做个异或就可以得到明文了，前提是加密模块已经算好了。

生成seed用于加密



硬件内存加密

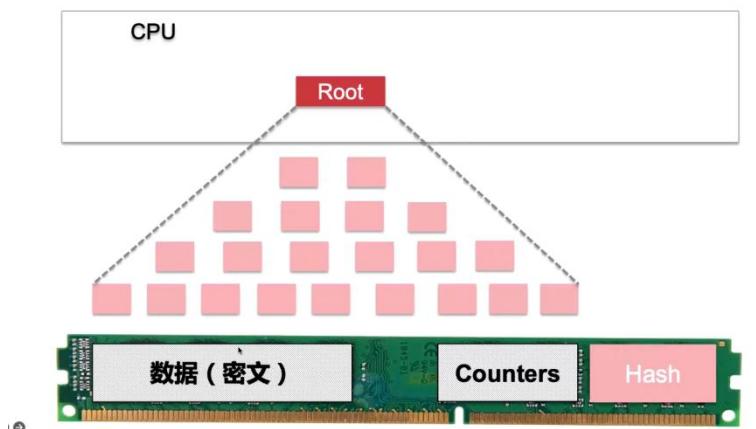
- **采用加密pad的方式而不是直接加密数据**
 - 每个内存区域对应一个pad
 - 对pad进行加密，然后数据与pad进行异或以实现加解密
- **加密pad由key和seed共同组成**
 - Seed可以由时空因子组成
 - 明文只需一次异或运算得到密文
- **任意两次加密的pad必须不同**
 - 如果两次加密的pad相同，可以轻松的反推明文

内存完整性保护

- **Merkle hash Tree**
 - 可以保证内存不会受到拼接和欺骗攻击
 - 不知道hash key无法计算对应的mac
 - 无法防御回放攻击
 - 攻击者可以将mac和data同时替换成老版本
- **将root hash (mac) 存储在CPU中**
 - 防御回放攻击
 - 攻击者无法修改root mac的值

在 128M 中，我们 4K 4K 算一个哈希，最终算到根节点保存在 CPU 里。每次读取的时候就会算一遍和 root 哈希做比对。

内存完整性保护 : Merkle Hash Tree



2022/5/26

其他平台的 enclave 技术

今天我们来讲 OS 的调试和测试，前几节课我们都在讲安全。我们上节课讲到 OS 不可信的时候，该怎么样做到防御纵深。这块东西叫做 ENCLAVE/TEE。Intel SGX 的基本思路就是 CPU 在启动的时候把一小块 memory reserve 起来。窃取数据是非常容易的，enclave 技术就可以让大家使用数据的时候看不到数据的明文。

ARM TrustZone 技术

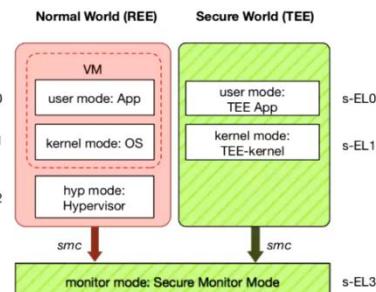
- ARMv6版本开始的安全硬件特性
 - 包括ARM11及Cortex A系列
 - 目前大部分手机芯片均有该硬件特性
- 同时运行一个安全的OS和一个普通的OS
 - 两个系统之间互相隔离运行
 - 安全的OS具有更多的权限
- TrustZone是一个全系统级别的安全架构
 - 处理器、内存和外设的安全隔离



我们可以在安全世界中运行比较重要的代码，比如指纹识别。安全世界中可以跑另一个OS。

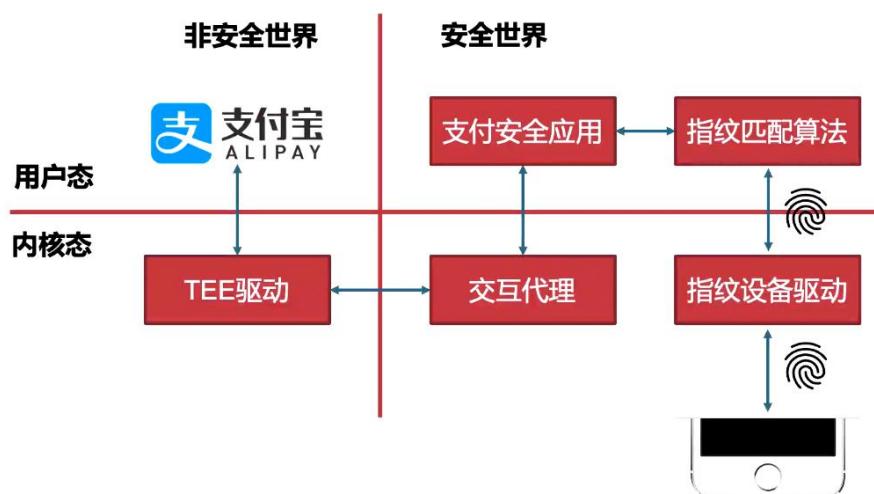
TEE硬件状态与软件架构

- TEE内部运行一个完整的操作系统
 - 与Android版本无关，也无需适配
 - TEE与Android通过共享内存进行交互
- TEE内部也分内核态与用户态
 - TEE用户态可运行多个安全应用（TA）
 - 安全应用可支持动态下载和动态更新
- TEE内部结构
 - Secure OS + 中间件 + 安全应用 + 外部交互



在非安全世界装支付宝的同时，会在安全世界中同步安装一个支付安全应用。支付宝的TA在安全世界中验证你的指纹。就算OS被root了，也没有办法偷到指纹。指纹匹配算法也是在安全世界中执行的。一旦安全世界被攻破了，不需要验证指纹的情况下，任何应用程序都可以向服务器发出一个合法的转账请求。

用TrustZone保护指纹的录入和识别



AMD SEV

这个思路是把虚拟机整个保护起来加密。AMD 是在商用 CPU 上应用了这个技术。它现在改进以后，可以认为不需要信任 hypervisor。

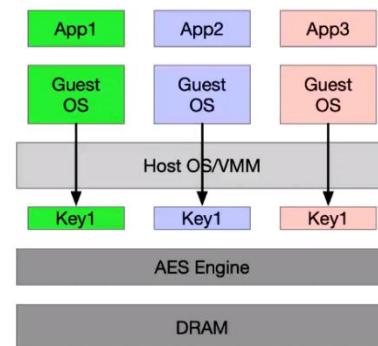
AMD SEV

• 以虚拟机为粒度的Enclave

- 对不同的虚拟机进行加密
- 每个虚拟机的密钥均不相同
- Hypervisor有自己的密钥

• 安全模型的缺陷

- 依然部分依赖Hypervisor
 - 如：为VM设置正确的密钥



之前这些都要硬件做改动。在 RISC-V 平台，有 machine-mode，它可以插入底层代码，它已经具有了 x86 中的 risc 指令的能力。有了这个能力之后，我们可以添加指令和功能来实现我们的目的。

RISC-V平台的Enclave

• RISC-V具有一个新的模式：Machine-Mode

- 位于操作系统和Hypervisor之下，直接访问物理地址
- 具有最高权限，可访问所有的计算资源，并提供新的功能
- 在M-Mode下实现的软件monitor，可实现Enclave的接口



蓬莱 Enclave：诞生自 SJTU



Name	Links	Licence	Maintainers
enCrypt	Website	Commercial, free for non-commercial evaluation	SEGGER
CoreGuard	Website	Proprietary	Dover Microsystems, Inc.
MultiZone API	GitHub	OC	Hex Free Security Inc.
Secure IoT Stack	GitHub	MIT, GPLv2, GPLv3, Evaluation license	Hex Free Security Inc.
MultiZone Security TEE & DCAE	GitHub	Evaluation license	Hex Free Security Inc.
Keystone Enclave	Website, GitHub	BSD License	Keystone Team
SecureIDP	Website	Proprietary	SecureID Corp.
HyperID	GitHub	Proprietary	Intrinsic ID
PGLAI	GitHub	Proprietary	PGLAI



<http://penglai-enclave.systems/>

小结

接下来我们小结一下。以前我们是把 OS 变得更安全，而现在我们认为 OS 不可能安全。这就可能导致单点错误，一旦 intel 被攻破，那么所有签过名的 report 就不可信。所以它现在也希望把认证机制分布开来。

控制系统复杂性

- **Enclave的抽象是一种简化**
 - 对威胁模型和信任关系的简化
 - 例如：Intel SGX将对软硬件环境的信任规约到对Intel的信任
 - 这种简化有可能带来新的问题：Single-point of Failure
- **Enclave的主要技术**
 - 保护技术：基于权限的隔离与基于加密的控制
 - 远程验证：对密钥的管理

我们把一个函数维护在一个 Enclave 中，虽然传入的参数都是加密的，但是如果这个函数的目的都是每次把 count+1，这样我们可以在外部做拦截，我们没有保护交互的过程。我们就可以通过侧信道反推出值落在什么区间里。

Enclave的不足

- **仅靠隔离是不够的，还需要考虑交互安全**
 - Enclave依然需要OS提供服务：调度、系统调用、资源分配...
 - 即使隔离，OS依然可能发起的攻击包括
 - 接口攻击：合法的系统调用返回错误的值
 - 例：malloc返回指向栈的地址，导致内部自己破坏掉栈
 - DoS攻击：拒绝分配计算资源（恶意调度）
- **依然受到侧信道等攻击的威胁**
 - Spectre、L1TF

调试和测试

大纲

- 操作系统对调试器的相关支持
- 如何对操作系统进行调试
- 性能调试及代码追踪
- 测试的基本原则和方法

调试器的基本原理

为什么需要调试器

- 定位和修复BUG，帮助程序员理解程序行为
- 基本功能
 - 中断程序运行读取内部状态
 - 获取程序异常退出原因
 - 动态修改软件状态
 - 控制流跟踪
- 以Linux和GDB为例介绍操作系统如何提供调试功能

我们来看它为什么能做到。ptrace 可以建立这样的一个调试关系。子进程可以把调试权交给父进程，这样父进程就可以来监控子进程。

调试器 – 建立调试关系

Linux的调试支持 : ptrace系统调用

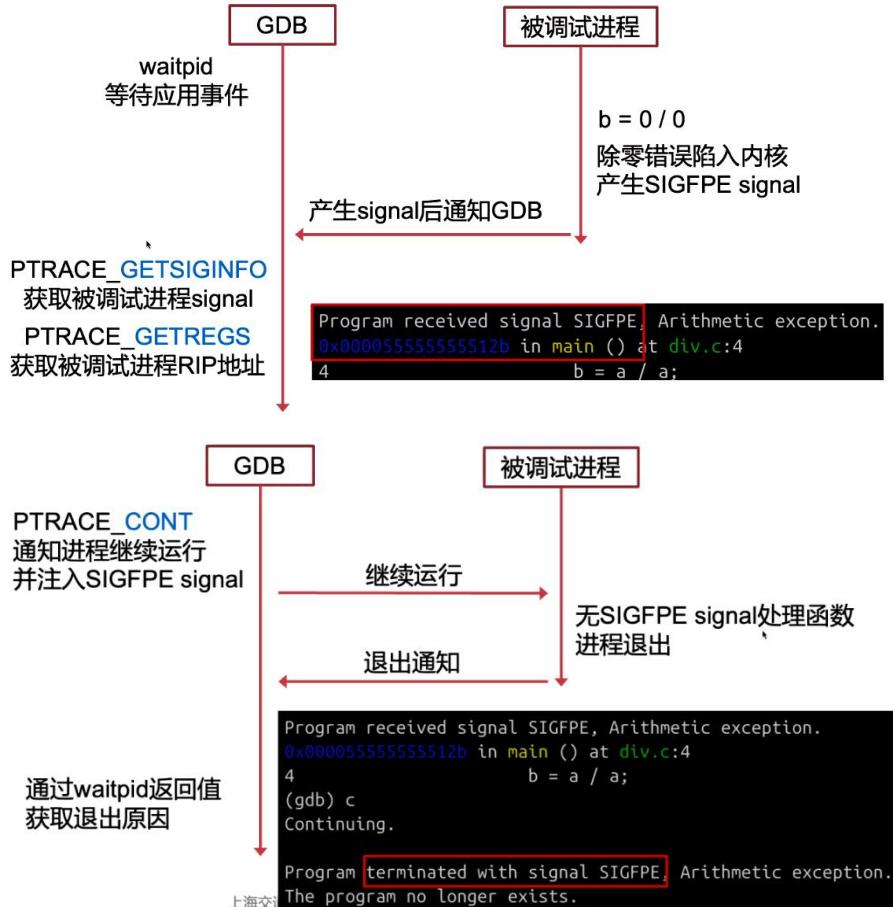
- GDB建立调试控制关系
 1. 子进程通过PTRACE_TRACE_ME将调试权交给父进程
 2. 通过PTRACE_ATTACH调试指定pid的进程
- 如何调试下述触发除零错误程序

```
div.c
1 int main() {
2     volatile int a = 5, b;
3     while (1) {
4         b = a / a;
5         a = a / 2;
6     }
7     return 0;
8 }
```

需求1：捕捉到进程除零错误
GDB捕捉除零错误产生的输出：
Program received signal SIGFPE, Arithmetic exception.
0x00005555555512b in main () at div.c:4

产生除零错误以后，就会陷入到内核产生一个 SIGFPE 的 signal，然后退出到 waitpid 处。

GDB捕捉异常信号流程



调试器 – 配置断点

• 需求2: 停止进程运行，用以观察进程状态

- 发送SIGINT至进程
- 或断点：在执行到特定指令地址时停止运行

• 使用断点调试

- 在第4行插入断点
- 观察变量a的值是否为0

```
div.c  
1 int main() {  
2     volatile int a = 5, b;  
3     while (1) {  
4         b = a / a;  
5         a = a / 2;  
6     }  
7     return 0;  
8 }
```

断点是有硬件的支持。一旦设置好 flag，就进入了单步调试状态。

断点的硬件支持

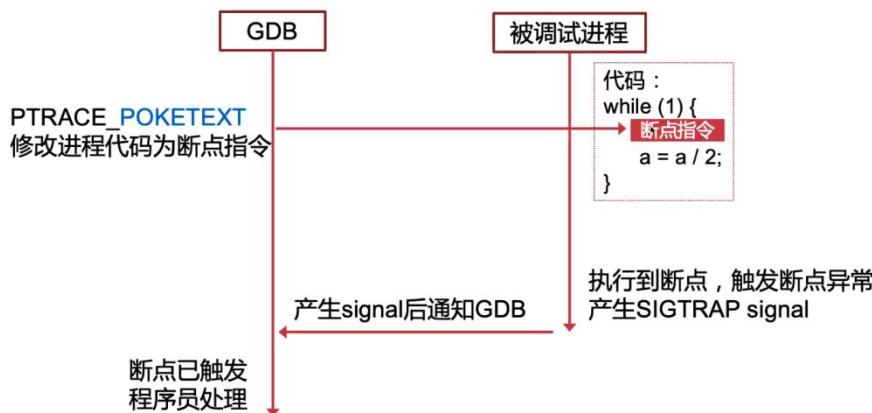
• 断点异常指令

- 在执行到特定指令时，触发断点异常陷入内核
- x86的int 3指令，AArch64的BKP指令

• 单步调试

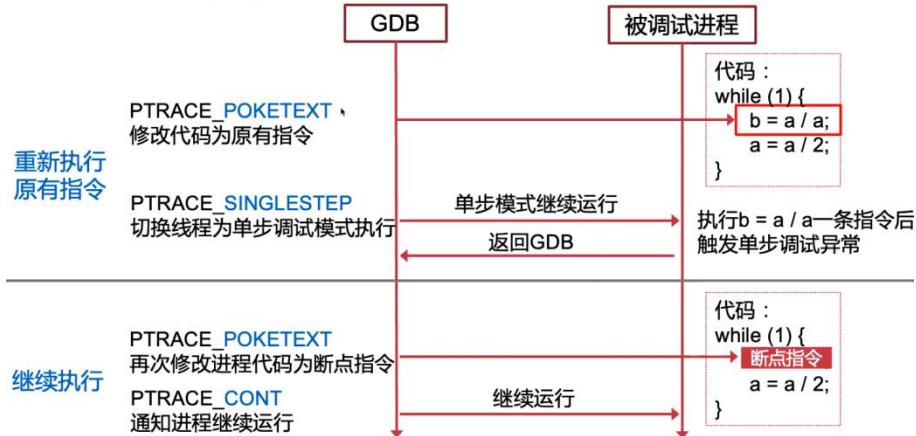
- 程序在用户态执行一条指令后立即陷入内核
- 通过特殊寄存器配置：x86的Trap Flag，AArch64的Software Step

GDB配置断点及断点触发



触发断点之后，我们再把这行代码改回来，回到单步调试模式下。

‣ GDB断点恢复运行



第三个需求是监控一个变量是否遭到修改。我们停在内存地址的修改上。

调试器 – 配置内存断点

- 需求3: 变量遭到异常修改时中断运行

- 内存断点

- 在变量a被修改时中断运行，观察是否为0

```
div.c
1 int main() {
2     volatile int a = 5, b;
3     while (1) {
4         b = a / a;
5         a = a / 2;
6     }
7     return 0;
8 }
```

在调用watch a命令后
GDB捕捉到a的值由5变为2

```
Hardware watchpoint 2: a
old value = 5
New value = 2
main () at div.c:4
```

内存断点的硬件支持

- Naïve实现

- 把内存地址所在页设为只可读
- 访问时触发page fault
- 缺点：对该页所有写操作均导致page fault，性能较差

- 断点寄存器

- 当访存地址为寄存器中的值时，触发断点异常

断点寄存器

- x86断点寄存器

- 访存地址等于断点寄存器触发中断

- 访存条件可配置

- 数据写（内存断点）
- 数据读和写
- 指令地址（断点）

断点寄存器 DR0 DR1 DR2 DR3

调试控制寄存器 DR7

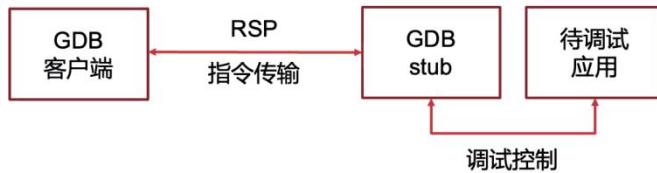
配置中断条件

- GDB配置被调试应用的断点寄存器

- 通过PTRACE_POKEUSER设置

我们也可以通过串口和网络进行远程调试。

远程调试



- **GDB客户端负责指令发送**
 - GDB远端串行协议 (GDB Remote Serial Protocol , RSP)
 - 通过串口线、网络等连接传输控制指令
- **GDB stub 负责实际调试**

操作系统调试器常见实现方法

- **调试操作系统调试支持的难点**
 - 缺乏操作系统提供给用户态的调试功能支持
 - 硬件相关问题，如外部设备、页表等
- **模拟器**
 - 虚拟机：完整模拟底层硬件，在模拟器中提供GDB stub
 - 用户态模拟：例如 User-mode Linux，忽略硬件相关的实现，使 Linux内核以普通进程的方式运行
- **内核自身实现的调试器**
 - 操作系统内部实现GDB stub，如Linux的KGDB

案例：QEMU 的 GDB 支持

案例：QEMU的GDB支持

- **与调试普通进程对比**
 - 不再有进程抽象相关的支持（如signal和系统调用跟踪）
 - ptrace相关接口替换为虚拟机管理接口
 - 如内存读写由PTTRACE_POKETEXT替换为直接读写虚拟机内存（假设hypervisor能直接访问虚拟机内存）
- **挑战**
 - 不能干扰客户机操作系统内部使用调试功能
 - 断点指令失效
 - 动态代码装载覆盖断点指令使断点失效

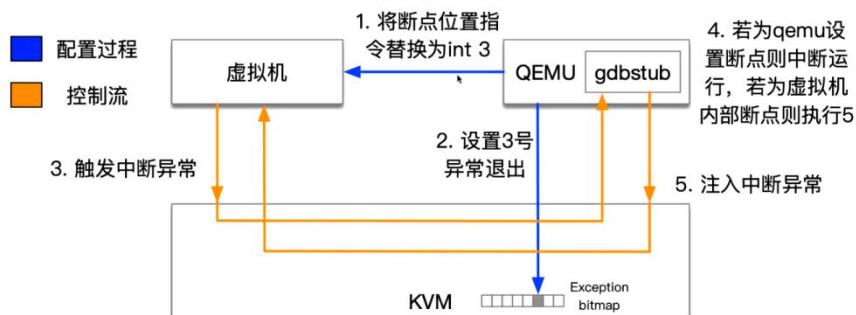
如果不设置 3 号异常退出，就不会触发 VM_EXIT 从虚拟机中退出。

案例：QEMU GDB的断点支持

断点调试

步骤2: 配置虚拟机内部产生断点异常时，退出虚拟机

步骤4和5: QEMU判断断点是否是虚拟机内部断点



断点指令相关问题

- **使用断点指令在操作系统调试中的困难**
 - 动态代码装载复写断点指令使断点失效
- **解决方法：硬件断点**
 - 指令地址等于断点寄存器即触发中断
 - 缺点：影响虚拟机内部使用硬件断点

性能调试

除了 debug 的调试，不仅仅有功能性的 bug，也有性能上的 bug。

为什么需要性能调试

- 程序功能性正确，但性能未达到理想情况
- 分析程序性能瓶颈
 - 程序运行时哪部分代码耗时较长
 - 哪部分内存发生较多缓存缺失
 - 跳转指令是否发生大量错误预测

hackbench 就是确认内核中耗时较多的函数，我们可以对代码路径上插桩，然后把运行的时间加一起，但是我们要修改大量的代码。

实际性能调试样例

- 分析 **hackbench** 测试用例执行时的性能瓶颈
 - Hackbench: Linux Test Project 中的调度测试之一
 - 多个进程相互使用socket读写进行同步
- 步骤一：确定内核执行中耗时较长的函数
 - 在可能的代码路径上插桩获取时间，统计时间占比最长的部分
 - 缺点：大量修改内核代码，统计复杂，通用性极低

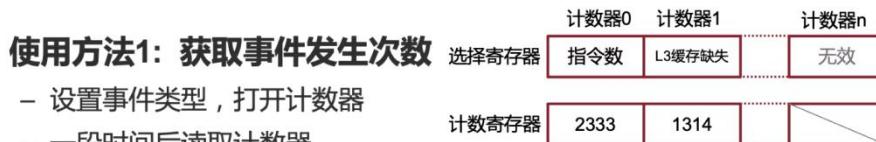
我们要从整体去考虑得到统计的信息，但是在每个函数头尾都加入统计信息就太麻烦了，于是我们要靠硬件。比如我们要获取某一个类型，我们就打开硬件计数器，每过一段时间通过 `syscall` 拿到数值。

但是这种方法依旧需要知道哪些函数运行了哪些代码，我们还是要加上起点和终点。

硬件计数器

监控程序执行过程中处理器发生某些事件的次数

- E.g., 执行指令数量，各级缓存缺失 (cache miss) 次数



- 设置事件类型，打开计数器
- 一段时间后读取计数器
 - 用户态通过特定指令或系统调用读取
- 使用该方法分析 `hackbench` 性能瓶颈仍需大量插桩，意义不大

我们就可以使用采样的方法，比如我们的硬件计数器是 32 位的，很快就可以溢出。每次溢出的时候 OS 把 rip 记录下来。这样我们就可以根据它落在哪个函数里从而知道触发中断的时候在哪里。

硬件计数器 – 采样

直接读取计数缺点

- 缺点：可能涉及对原有代码修改（插桩）

使用方法2：采样

- 设置事件类型，打开计数器
- 当计数器溢出时，产生中断
 - 在中断处理中获取地址信息
 - 清空计数器，等待下一次中断
- 分析 `hackbench` 性能瓶颈：每经过一定cycle数触发一次中断，统计中断时指令地址，观察这些地址属于哪些函数



溢出产生性能调试中断

perf 是辅助用户态的性能统计程序。

Linux性能计数器采样支持

- 性能相关事件 perf events
 - 以event的抽象暴露性能计数器 (以及一些其它性能调试方法)
 - perf_event_open通知内核需要使用哪些计数器
 - 采样过程由内核完成
 - 采样结果放入内核与用户态共享的内存中 , 减少读取大量采样信息时的开销
- 前端工具perf
 - 直接使用perf events相关系统调用仍然较复杂
 - perf工具包装常见的性能分析方法

采样分析hackbench

• 针对cycle数进行采样

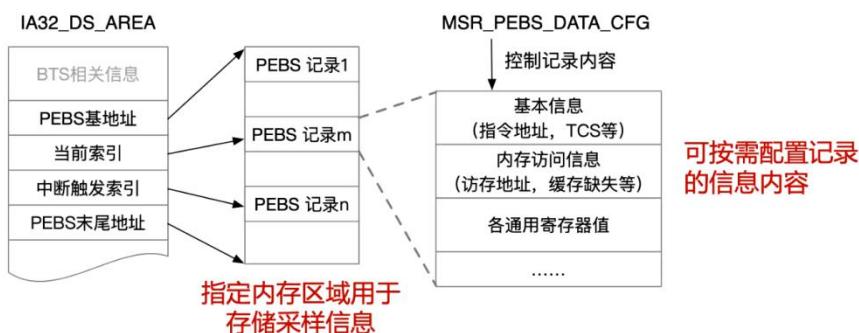
- perf record -e cycles hackbench	确定两个影响性能因素
- perf report	1. socket读写用户态数据
收集到的采样次数	预估的event次数
	2. 系统调用产生的上下文切换
<pre>Samples: 53K of event 'cycles', Event count (approx.): 39862780526 Overhead Command Shared Object Symbol 7.09% hackbench [kernel.vmlinux] [k] syscall_return_via_sysret 6.36% hackbench [kernel.vmlinux] [k] entry_SYSCALL_64 6.16% hackbench [kernel.vmlinux] [k] copy_user_enhanced_fast_string 4.68% hackbench [kernel.vmlinux] [k] unix_stream_read_generic 2.18% hackbench [kernel.vmlinux] [k] __check_object_size 2.04% hackbench [kernel.vmlinux] [k] _raw_spin_lock_irqsave</pre>	

基于中断采样的缺点

- 中断时收集信息的缺陷
 - 采样获取的指令地址不准确
 - 中断发送需要时间 , CPU收到中断时的指令地址 , 与产生采样点指令地址可能存在偏移 (skid)
 - 乱序执行
 - 中断时无法收集完整的采样信息
 - E.g., 缓存缺失时 , 对应的内存地址未知
- 更精确的采样支持需要 :
 - 计数器溢出时马上收集信息
 - 能够收集更广泛的信息

精确采样硬件支持

- 例如x86的PEBS (Processor/Precise Event Based Sampling)
 - 计数器溢出时，立即记录相关信息至内存

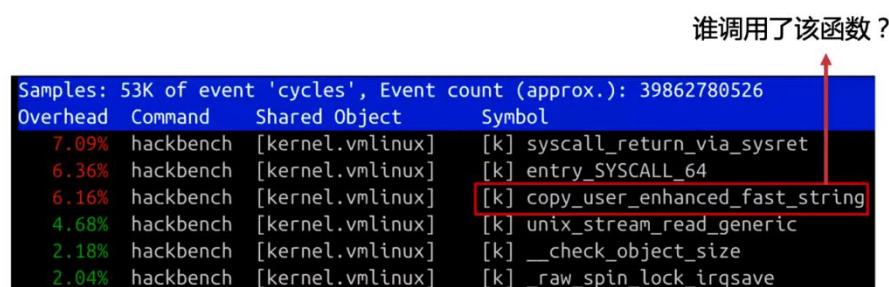


精确采样硬件支持

- 有无必要启用精确采样分析 `hackbench`
 - `perf record -e cycles:ppp hackbench`
 - 不具备必要性：即使指令地址有偏移，针对cycle数采样情况下，各函数收到中断的概率大致不变
- 何时需要启用精确采样
 - 需要确定发生特定事件（缓存缺失、跳转预测失败）时指令地址
 - 需要除了指令地址外的其他采样信息
 - E.g., 获取缓存缺失地址：`perf mem record`

控制流追踪

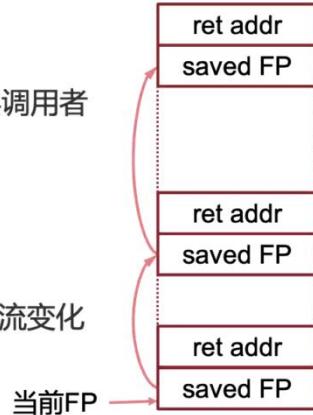
- 步骤一：确定哪些函数占用了较长执行时间 - 采样 ✓
- 步骤二：确定是如何执行到该函数的？



控制流追踪

控制流追踪

- **基于软件的控制流追踪**
 - backtrace：根据调用栈递归获取上层调用者
- **缺点**
 - 编译器优化可能去除栈指针存储
 - 只能处理函数调用
 - 无法应对jmp、中断等导致的控制流变化



但是最多记录 16 次跳转，因为硬件也不是无限多的。所以我们之前提到的 CFI 也可以通过这 16 次跳转看看跳转的 pattern，看看是不是攻击

控制流追踪

- **基于硬件的控制流追踪**
 - 记录jmp、call、中断等导致跳转的前后位置，构建完整控制流
 - e.g., Last Branch Record (Intel)
 - 两组寄存器分别构成栈，记录最近N次跳转的信息



追踪hackbench控制流

- 指示perf采样时记录控制流变化

- perf record -e cycles -g hackbench
 - perf report

```
- 32.74% vfs_read
  - 30.53% new_sync_read
    - 29.41% sock_read_iter
      - 28.14% unix_stream_recvmsg
        - 27.58% unix_stream_read_generic
          - 9.90% unix_stream_read_actor
            - skb_copy_datagram_iter
              - __skb_datagram_iter
                - 5.50% __copy_to_iter
                  - 5.15% copyout
                    copy_user_enhanced_fast_string
```

确定了一条访问
copy_user_xxx
的代码路径

由vfs_read导致

程序执行追踪

- 步骤一：确定哪些函数占用了较长执行时间 - 采样 ✓
- 步骤二：确定是如何执行到该函数的 – 控制流跟踪 ✓
- 步骤三：理解程序行为，为什么会产生这种调用关系
 - hackbench 中大量时间处理socket读写，读写数据规模有多大
 - 作为调度测试，hackbench 是否对调度器产生了足够压力
- 需要具有更多程序语义的跟踪机制

静态追踪方法

- 在代码编写时静态插桩获取信息的方法
 - 简单可靠的方法：打印
- 在常用的函数中预置静态的跟踪函数
 - 打印可能造成性能开销
 - 提供打开或关闭选项，关闭时应几乎不产生性能开销
 - e.g., Linux 的 Tracepoint

静态追踪方法

- 以 hackbench 为例，如何了解线程切换情况
 - 分析调度是一种常见需求，因此Linux内核在__schedule函数中内置了Tracepoint

```
static void __sched notrace __schedule(bool preempt)
{
    ...
    if (likely(prev != next)) {
        ...
        trace_sched_switch(preempt, prev, next);
        ...
    }
}
```

用户态应用通过接口修改
trace_sched_switch_enabled
控制Tracepoint开关

if (unlikely(trace_sched_switch_enabled)) {
 调用打印方法
}

编译生成

出现了 13 万次的调度。

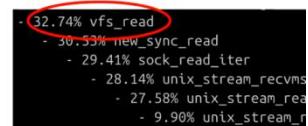
分析hackbench中线程切换

- Tracepoint作为perf event事件
 - perf record -e sched:sched_switch hackbench
 - 作为调度测试，hackbench确实触发了大量调度
 - 获取了调度前后进程名、pid、优先级等信息

```
amples: 132K of event 'sched:sched_switch', Event count (approx.): 132159
Verhead: Trace output
0.10% prev_comm=hackbench prev_pid=3157715 prev_prio=120 prev_state=S => next_comm=hackbench next_pid=3157718 next_prio=120
0.09% prev_comm=hackbench prev_pid=3157725 prev_prio=120 prev_state=S => next_comm=hackbench next_pid=3157728 next_prio=120
0.09% prev_comm=hackbench prev_pid=3157718 prev_prio=120 prev_state=S => next_comm=hackbench next_pid=3157720 next_prio=120
0.09% prev_comm=hackbench prev_pid=3157726 prev_prio=120 prev_state=S => next_comm=sh next_pid=3157764 next_prio=120
0.09% prev_comm=hackbench prev_pid=3157731 prev_prio=120 prev_state=S => next_comm=swapper/5 next_pid=0 next_prio=120
```

动态追踪方法

- 静态追踪方法缺陷
 - 修改静态定义的跟踪点需要重新编写、部署内核
 - Hackbench中，已知socket读操作耗时较长
 - 是否是数据量较大导致的
 - vfs_read没有预置静态Tracepoint
- 动态方法
 - 程序运行时，在不确定的代码位置插入一段动态指定的追踪函数
 - e.g., Linux kprobe，实现方式类似于断点调试



Linux动态追踪方法kprobe

- 使用和调试器类似的原理动态插入代码

- e.g., 配置handler函数在执行 指令2 之前执行



分析hackbench数据读写大小

- 使用 kprobe 探究 hackbench 中数据读写大小

- 目标：获取vfs_read每次调用时count的大小

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
```

- 方法：在vfs_read执行前插入打印count值的函数

- 定义probe点

• perf probe --add 'vfs_read count=%dx:u64'

↑
定义probe位置

↑
x86上第三个参数位于rdx中，大小为64bit

我们得出的结果是读的次数很多，导致 vfs_read 时间占比较大。如果要优化，我们就应该对读数据 batch 一下，减少读的次数，增大每次读入的数据。

分析hackbench数据读写大小

- 使用 kprobe 探究 hackbench 中数据读写大小

- 新定义的kprobe点作为perf event采样

- perf record -e probe:vfs_read hackbench

```
Samples: 262K of event 'probe:vfs_read'
Overhead  Trace output
 98.47%  (ffffffff902a0700) count=100
  1.53%  (ffffffff902a0700) count=1
  0.00%  (ffffffff902a0700) count=68
  0.00%  (ffffffff902a0700) count=32
  0.00%  (ffffffff902a0700) count=784
  0.00%  (ffffffff902a0700) count=832
```

单次读数据量不大为100byte，但是读次数较多，因此vfs_read时间占比较大

测试的基本原则和方法

测试的目的

- **验证程序功能正确性**
 - 程序是否会崩溃
 - 功能是否与设计一致
- **基准测试**
 - 确定程序在特定运行环境下的性能指标
- **操作系统测试的必要性**
 - 作为基础设施，操作系统的正确性和性能直接影响上层应用

操作系统测试的基本方法与原则

- **问题1：如何快速使错误暴露？**
- **测试规模由小至大：小规模测试中暴露的错误更方便定位**
 - 先对各功能模块做独立测试：单元测试
 - 以细粒度方式进行测试：函数或功能模块粒度

如测试顺序遍历
链表元素的方法

验证其中每个元素
都为预期的值

```
for_each_in_hlist(iter, hnode, &head) {
    i--;
    printf("traverse: %d %p->%p\n", iter->value,
           iter, iter->hnode.next);
    mu_check(iter->value == data[i].value);
}
```

操作系统测试的基本方法与原则

- **测试规模由小至大**
 - 单元测试：以函数或功能模块为粒度测试
 - ChCore：单独编译内存管理、调度器等模块，在无需运行内核条件下测试
 - Linux：KUNIT，仅编译部分代码，在UML模式下运行
 - 单元测试完成后进行集成测试
 - 集成测试：整合各个功能模块统一测试
 - ChCore：完整部署内核并运行用户态应用，对网络、文件系统、同步原语等测试

操作系统测试的基本方法与原则

- 代码迭代中，尽早确认新修改是否引入BUG
 - 产生BUG的原因
 - 新的修改本身异常
 - 代码修改触发原有隐藏的异常
 - 回归测试
 - 小规模代码修改后马上运行测试，即使该测试与修改部分无直接关系
 - ChCore：每次代码被push到远端，以及代码合并进主线前，都会运行完整的测试

兼容性测试

- 问题2：如何确保操作系统能够运行在不同硬件平台，支持各类不同应用？
- 测试不同硬件环境下兼容性
 - ChCore：各类测试在虚拟化的x86，虚拟化的AArch64，真实的AArch64硬件均部署运行
 - Linux：kernelci 验证Linux在各类不同硬件上能否完成基本测试

兼容性测试 <https://kernelci.org>



兼容性测试

• 测试向上能否兼容应用

- 向后兼容性，操作系统开发迭代后仍能运行较老的应用
- 不同操作系统间如何提供统一的接口
 - 如针对Linux开发的应用能够部署在众多Linux发行版中
- 遵循各类标准
 - 如Linux目录树标准 FHS (Filesystem Hierarchy Standard)
 - 如API 标准 POSIX (Portable Operating System Interface)
- 针对标准进行测试
 - 使用 POSIX Test Suite 验证 POSIX 接口符合标准

兼容性测试

• 测试向上能否兼容应用

- 案例：POSIX Test Suite 验证 fork 接口是否符合标准

fork测试目录部分文件，罗列多项测试

```
messcat_src.txt 9-1.c 7-1.c 4-1.c 22-1.c 21-1.c 18-1.c 17-1.c 14-1.c 12-1.c  
assertions.xml 8-1.c 6-1.c 3-1.c 2-1.c 19-1.c 17-2.c 16-1.c 13-1.c 1-1.c
```

fork的各项接口标准

```
<assertion id="3" tag="ref:XSH6TC2:12994:12995">  
    The new process' ID does not match any existing  
    process or group ID.  
</assertion>  
<assertion id="4" tag="ref:XSH6TC2:12996:12997">  
    The parent process ID (ppid) of the child process  
    is the process ID (pid) of the parent process  
    (caller of fork()).  
</assertion>
```

4-1.c 测试方法

```
* The steps are:  
* -> create a child  
* -> check its parent process ID is  
*      the PID of its parent.  
  
* The test fails if the IDs differ.
```

具体测试

操作系统稳定性

• 问题3: 如何验证操作系统的可靠性？

- 基本功能正常前提下，需要确保极端状况下操作系统的正常运行

• 压力测试

- 压榨处理器、内存、I/O等资源至极限
- 频繁进行系统调用
- 长时间测试 (长稳测试)

• 提高测试时的代码覆盖率

- 未测试代码出现异常概率更高

压力测试案例：syzkaller模糊测试

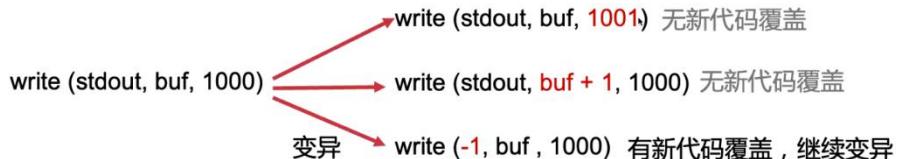
- 模糊测试 (fuzzing)

- 为操作系统构造大量随机系统调用并执行
 - 系统调用参数随机变化，期望能覆盖更多代码
 - 但是纯随机变化参数效果不佳
 - 大量输入可能属于同一等价类，代码执行路径相同
 - 基于变异 (mutation) 的参数生成
 - 在已有参数基础上随机变化
-
- ```
graph LR; A[write (stdout, buf, 1000)] --> B[write (stdout, buf, 1001)]; A --> C[write (stdout, buf + 1, 1000)];
```

## 压力测试案例：syzkaller模糊测试

- 模糊测试 ( fuzzing )

- 基于变异 ( mutation ) 的参数生成
  - 部分随机变化会引入新的代码覆盖
    - 收集产生新代码覆盖的输入，作为新的等价类
    - 在新输入基础上继续变异



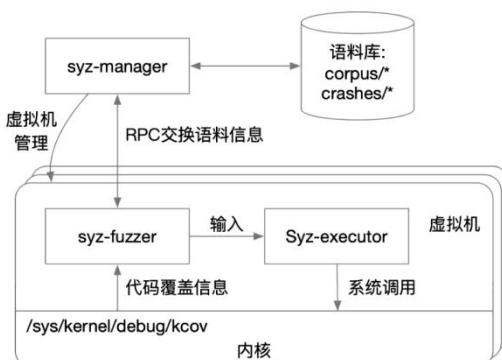
## 压力测试案例：syzkaller模糊测试

- 多个虚拟机运行操作系统

- 提升测试效率
- 虚拟机内部不断进行：
  - 系统调用
  - 系统调用参数变异
  - 收集代码覆盖信息

- 共享语料库

- 提升变异的效率



## 性能测试

- **问题4: 如何定量比较不同软硬件配置下性能表现**
  - 性能测试
- **选择合适的测试程序**
  - 明确测试的性能指标（如吞吐量，延迟等）
  - 明确测试的场景（如文件系统读写测试使用顺序还是随机）
  - 已有测试不满足需求时可以自己实现测试程序
    - 但必须确保测试符合真实场景，有代表性

## 性能测试

- **控制无关变量**
  - 以文件系统测试为例
  - 软件配置：文件系统类型，配置（日志级别，缓存）
  - 硬件配置：硬盘种类和型号
  - 其他无关因素：内核版本，时钟中断频率，无其它占用大量资源的应用
- **减少随机不稳定因素**
  - 考虑预热一段时间再测试
  - 绑核心运行
  - 避免跨NUMA节点的内存访问

## 持续集成

- **问题5: 如何有效地进行和管理测试？**
  - 测试考量因素众多，流程复杂
- **持续集成 ( CI, Continuous Integration )**
  - 开发者较频繁地将代码合并入主线，使用自动化测试保障代码正确性
  - 自动化部署和回归测试
    - ChCore：代码在push到远端分支时，自动开始进行如下流程
      - 各个平台版本内核的编译
      - 静态检查工具分析
      - 各模块单元测试和整体集成测试
      - 针对IPC和系统调用的性能测试

# 持续集成

- **门禁系统：确保主线代码的可靠性**

- 通过自动化测试才能合并进入主线
- ChCore的门禁设置：按顺序通过如下测试



# 总结

- **操作系统的调试器支持**

- 操作系统提供的调试支持
- 如何对操作系统进行调试

- **性能调试**

- 使用采样的方式分析性能
- 软件跟踪机制

- **测试环节的基本方法与原则**

## Linux安全漏洞修复流程



- **基于真实的漏洞修复 (Credit: Fan Yang)**

- CVE-2020-10757
- Linux commit 5c7fb56e5e3f

- **阶段1：发现bug**

- 调研是否已被发现、是否有解决方案
- 简化复现过程
- 严重性评估，若属于安全漏洞，可以提供exploit
- 若有解决方案，可以提供patch

知情范围：自己

# 漏洞修复



#### • 阶段2：汇报

- 非安全漏洞：kernel邮件列表，bugzilla 公开渠道
  - 安全漏洞：security@kernel.org 非公开渠道

- 申请CVE id (Common Vulnerability and Exposures List)
  - 意味着公开？此时CVE处于RESERVED状态，信息不公开

知情范围：邮件列表订阅者+自己

漏洞修复



- 阶段3：处理漏洞

- 相关子系统开发、维护人员加入讨论

Will Deacon <will@kernel.org>  
回复: \*\*\*UNCHECKED\*\*\* [vs-plain] User can control PTE value to read/write anywhere, when "mrei  
DAX file to a mmaped anonymous memory region  
收件人: 杨帆 <Fan\_Yang@sjtu.edu.cn>,  
抄送: Marcus Meissner <meissner@suse.de>, Andrew Morton <akpm@linux-foundation.org>,  
Security Officers <security@kernel.org>, +Williams, Dan J <dan.j.williams@intel.com>, DAX, nvdimmm相关开发者  
+Kirill A. Shutemov <kirill.shutemov@linux.intel.com>, +Mel Gorman <mgorman@suse.de>  
**huge page**相关开发者                   **memory management**相关开发者

知情范围：相关开发者+邮件列表订阅者+自己

漏洞修复



#### • 阶段3：处理漏洞

- ### - Patch提议与讨论

Should this be using pmd\_devmap() instead, i.e. along the lines of  
5c7fb56e5e3f ("mm, dax: dax-pmd vs thp-pmd vs hugetlbfss-pmd")?  
I too thought about "|| pmd\_devmap()". I am wondering if a new helper  
which does  

```
pmd_trans_huge() || pmd_devmap()
```

  
Agreed. Looks like a right fix.  
So I take that back. I think the fix for this case right now is to  
just add the pmd\_devmap() case, and the cleanup is to try to change  
these all to be "for large pages, do this.." "Ugh. So is that one-liner sufficient?  
Linus  
patch  
Yes, it looks sufficient for the bug at hand.

## 漏洞修复



### 阶段3：处理漏洞

#### - 测试与patch review

```
fsdax: | remap 4K size | remap 2M size
| normal page mapped[1] | observed move_ptes, OK | observed move_normal_pmd, OK
| huge pmd mapped | observed split_huge_pmd, OK | observed move_huge_pmd, OK

[1] DAX file system use huge page by default, I'd probably drop this paragraph, as it's likely just to provoke a reaction from people who like making noise about this stuff.

devdax: | remap 4K size | I would propose trimming this comment to just:
| normal page mapped[2] | move_ptes, as expected /* TODO: kernel-wide: replace "pmd_trans_huge() || pmd_devmap()" pattern with p
| huge pmd mapped | not supported [3] ..or just deleting it since it won't live long.

[2] by setting --align 4K when ndctl create-name ...Reviewed-by: Dan Williams <dan.j.williams@intel.com>
[3] dmesg shows "__dev_dax_pte_fault: fail, unalign" Fixes: 5c7fb56e5e3f ('mm, dax: dax-pmd vs thp-pmd vs hugetlbfss-pmd')
Cc: <stable@vger.kernel.org> Acked-by: Kirill A. Shutemov <kirill.shutemov@linux.intel.com>
```

76

## 漏洞修复



### 阶段3.5：Embargo Period (可选)

- 各大发行版加入讨论，准备修复
- 为什么不能直接修复？
  - 在公开的邮件列表发patch、往主线push都属于公开
  - 预防各发行版未及时修复已公开的漏洞
- 决定是否需要该阶段：评估危害

this vulnerability as compared to the original report? Is it limited to scenarios with a DAX filesystem backed by nvdimm/pmem or is there a wider attack surface?

We need to figure out when (in which commit, kernel version) the problem was introduced. Have you researched this? Also, if it's fairly recent, what stable and/or distros kernels it's possibly been backported to.

77

## 漏洞修复



### 阶段3.5：Embargo Period

#### - 协商Embargo Period长短

At SUSE we would be fine with either an immediate release or shorter term embargo.  
Ciao, Marcus  
  
We (Amazon Linux) are also fine with pushing out immediately.  
Regards,  
Anthony Liguori  
  
We (VMware Photon OS) are also okay with pushing out the fix immediately.  
Thank you!  
Regards,  
Srivatsa

知情范围：各大发行版+相关开发者+邮件列表订阅者+自己

78

## 漏洞修复



### • 阶段4：公开

- 并入主线 Just FYI, this is now in my tree as commit 5bfea2d9b17f.  
Linus
- Backport受影响的stable版本
- 公布至公开的邮件列表、安全话题社区，比如oss-security@lists.openwall.com
- 更新CVE状态至公开

知情范围：所有人！

79

## 2022/6/2 期末复习

我们还是从头回顾一下。上完这一个学期的 OS 之后，大家应该能够说出 OS 的定义。OS 的话，上面是应用，下面是硬件，OS 就是中间这一层。这也就是我们为什么一开始说，如果应用可以直接访问硬件，那就不需要 OS 了。

当然“无限的、连续的资源”是对应用程序的幻象，这也是为了简化。

## 操作系统：在硬件和应用之间的软件层

### 操作系统和应用的关系

- **服务**应用：如提供硬件操作
- **管理**应用：如加载、调度等

### 应用

### 操作系统

### 硬件

### 操作系统和硬件的关系

- **管理**硬件：操作硬件以完成功能
- **抽象**硬件：应用不关心硬件差异

“**操作系统是管理硬件资源、控制程序运行、改善人机界面和为应用软件提供支持的一种系统软件。**”  
《计算机百科全书(第2版)》

“**操作系统将有限的、离散的资源，高效地抽象为无限的、连续的资源。**”

《现代操作系统：原理与实现》

3

所以 OS 没有严格的定义，只是一个约定。取决于语境，有些可以算 OS，也可以不算 OS。

# 究竟什么是操作系统？

- “操作系统”并没有严格的唯一定义
  - 是一个相对概念：相对“应用”而言
  - 操作系统也可包含运行在用户态的框架（framework）
- 例如：**SSL库、Dalvik（Java虚拟机）是否属于操作系统？**
  - 对Android来说，属于操作系统框架层；App属于应用
  - 对Linux来说，不属于操作系统；hello属于应用
- 我们课程中的操作系统：以hello作为典型应用

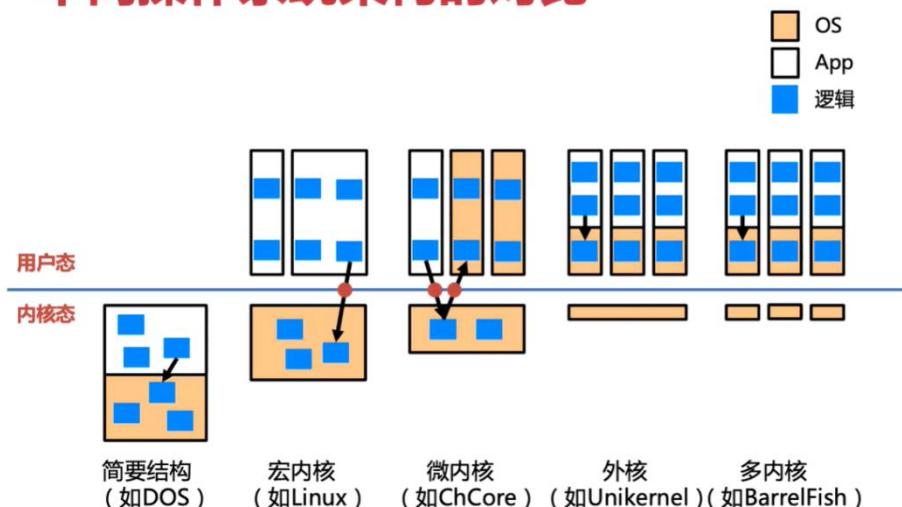
这张图就是OS的架构，我们根据用户态和内核态做了一个划分。DOS只要出了问题，可能整个内核就崩了。做了用户态的隔离之后，用户态的应用程序崩溃不会影响OS的崩溃。

微内核的话就是把内核态的东西往用户态放，但是我们不能绝对地认为微内核的隔离性比宏内核要好。在Windows里，窗口系统是放在内核态的，这样用户使用的latency比较低。如果我们把它放到用户态，它的响应速度就慢了，但是好处就是窗口系统崩了不会影响我们的内核。但是如果我们现在只能使用鼠标，窗口系统崩了我们就做不了任何事情了。所以虽然内核活着，但是我们不能去操作了，实际上也就等于崩溃了。

所以微内核和宏内核之间没有绝对的好坏，隔离性也没有严格的高低。外核就是说我们内核里不提供抽象，所有东西都给用户态去做。把自己的OS变成一个library，好处就是OS不提供抽象，所有东西都由应用程序来负责，性能可以做到极致，缺点就是如果我们自己是用户，用户要自己管理自己的libOS，就很麻烦。

多内核和前面这些的关系不大，核心是因为环境变了。多核场景下使用一个CPU控制所有设备是比较麻烦的，是一个分布式系统。我们可能有一千个核，并且我们的硬件是异构的，在这种情况下我们在每个CPU上跑一个内核，再对外提供服务会比较自然。

## 不同操作系统架构的对比



# 微内核的优缺点分析

## • 优点

- 易于扩展：直接添加一个用户进程即可为操作系统增加服务
- 易于移植：大部分模块与底层硬件无关
- 更加可靠：在内核模式运行的代码量大大减少
- 更加安全：即使存在漏洞，服务与服务之间存在进程粒度隔离
- 更加健壮：单个模块出现问题不会影响到系统整体

## • 上世纪80/90年代，“微内核”一度成为下一代操作系统的代名词

微内核的缺点就是 IPC 的性能比较差，因为 IPC 要做各种各样的切换。我们可以通过软件的方法来做共享内存，这样我们就可以减少数据的 copy。IPC 和微内核的关系很紧密。

# 微内核的优缺点分析

## • 缺点

- 性能较差：内核中的模块交互由函数调用变成了进程间通信
- 生态欠缺：尚未形成像Linux一样具有广泛开发者的社区
- 重用问题：重用宏内核操作系统提供兼容性，带来新问题



微内核是一种思想，而不是一个技术。我们经常会提到 ChCore。Linux 有没有微内核的影子？Windows 还自称自己是微内核。一部分功能拆到用户态，这样 crash 之后就可以去做 recovery。微内核和宏内核是在互相渗透的。

# 操作系统结构的演进与生态

## • 系统软件需要一条演进之路

- 尽可能集成现有的POSIX API/Linux ABI
- 避免棘手的系统调用（如fork）
- 避免不可扩展的POSIX API

## • 系统软件一直在不断演化

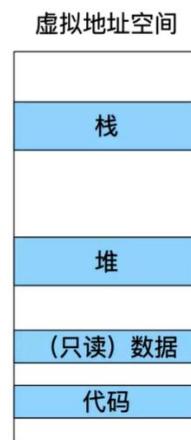
- 例：Linux User-space I/O (UIO)，向微内核近了一步
- 单节点下也存在更多的分布式、低时延的可编程设备
- 非易失性内存的出现可能推动存储层次在OS中的完全改革

接下来就是虚拟内存，我们对虚拟内存都不陌生。整个虚拟内存是需要硬件和软件协同才能做的事情。虚拟地址到物理地址的翻译一定是硬件翻译的，怎么翻译才是 OS 去配置的。OS 翻译的粒度是 page。OS 映射的时候会延迟映射，也就是它会把 page fault 认为是一种合理的情况存在。OS 实际上不想立马给出去那么多物理内存，所以它把映射推迟了。所以应用程序以为自己拿到了，实际上还没有分配物理内存。这个意味着虚拟地址到物理地址的映射可能是缺失的，这就需要 OS 记录给应用分配了哪些物理地址，这个信息就记录在 VMA 里。

VMA 完全是 OS 的概念，和硬件没有关系。硬件只负责页表，认为页表不在就是 page fault。而 OS 如果发现 page fault 的地址在 VMA 中，那么就说明是需要 OS 来处理的 page fault，如果不 VMA 中，那就是 segmentation fault。

## VMA : VM Area

- **OS采用段来管理虚拟地址**
  - 段内连续，段与段之间非连续
  - 合法虚拟地址段：代码、数据、堆、栈
  - 非法虚拟地址段：未映射
    - 一旦访问，则触发segfault
- **思考：为什么要用段来管理？**



## VMA是如何添加的

- **途径2: 应用程序主动向OS发出请求**
  - brk() (扩大、缩小堆区域)
    - 可选策略：OS也可以在创建应用时分配初始的堆VMA
  - mmap()
    - 申请空的虚拟内存区域
    - 申请映射文件数据的虚拟内存区域
- **用户态的malloc会改变VMA**
  - 通常是调用brk，在堆中分配新的内存
  - 部分实现也可以调用mmap，由应用管理多个VMA

换页机制也是纯软件实现的。硬件只支持了页表和 page fault。OS 通过软件支持了 delay mapping 和 swapping。Swapping 的前提是存在 VMA，其次这个页表一定要修改为 not present 的，这样访问的时候就会发生 page fault。相当于把这个地址映射到了磁盘上去，这块 VMA 映射到了一块磁盘上，这个完全是 OS 做的事情。

# 换页机制 ( Swapping )

## • 换页的基本思想

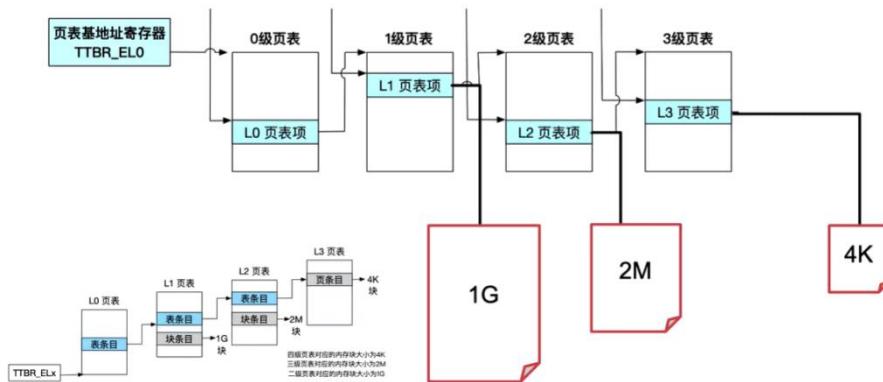
- 用磁盘作为物理内存的补充，且对上层应用透明
- 应用对虚拟内存的使用，不受物理内存大小限制

## • 如何实现

- 磁盘上划分专门的Swap分区，或专门的Swap文件
- 在处理缺页异常时，触发物理内存页的换入换出

大页也是硬件翻译的时候提供的机制。

## 大页



## 大页的利弊

### • 好处

- 减少TLB缓存项的使用，提高 TLB 命中率
- 减少页表的级数，提升遍历页表的效率

### • 案例

- 提供API允许应用程序进行显示的大页分配
- 透明大页 ( Transparent Huge Pages) 机制

### • 弊端

- 未使用整个大页而造成物理内存资源浪费
- 增加管理内存的复杂度

on-demand paging 和物理地址是无关的，应用程序不需要 care。OS 当然在这个时候需要分配一块物理地址。

## 场景-1：应用触发on-demand paging

- 问：当应用调用malloc时，与物理内存是否有关？
  - 应用调用malloc时，内核仅仅为其分配虚拟地址
  - 此时虚拟地址对应的VMA为valid，但对应页表的valid bit为0
  - 当第一次访问新分配的虚拟地址时，CPU会触发page fault
- 操作系统需要做（即page-fault handler）：
  - 找到一块空闲的物理内存页 ← 物理内存管理（页粒度）
  - 修改页表，将该物理页映射到触发page-fault的虚地址所在虚拟页
  - 回到应用，重复执行触发page-fault的那行代码

内核要用到数据结构的时候，是直接从 buddy system 分配出来的，和用户态的 malloc 还是有区别的。因为 malloc 是先分配虚拟地址，等到 page fault 的时候再分配物理地址，而 OS 这里是直接分配物理地址。

## 场景-2：内核自身用到的数据结构

- 内核运行过程中也需要用到动态内存
  - 例如：挂载文件系统后需要大量dentry数据结构
  - 动态性：用时分配，用完释放，类似用户态的malloc
  - 申请到的内存，不一定是页粒度，可能更细（特点-1）
- 问：为什么不使用与malloc类似的机制？
  - 即先分配虚拟地址，然后通过on-demand paging分配物理页
  - 对内核来说，开销太大：这是内核中发生的page fault！
  - 若要不触发page fault，则必须已经映射完成（特点-2）

当设备需要通过 DMA 分配内存时，需要连续的物理地址。

## 场景-3：设备需要分配DMA内存

- DMA：设备绕过CPU直接访存
  - 由于绕过CPU的MMU，因此直接访问物理地址
  - 通常需要大段连续的物理内存
- 操作系统必须有能力分配连续的物理页
  - 需要用一种高效的方式来组织和管理物理页

## 场景-4：换入换出 ( swapping )

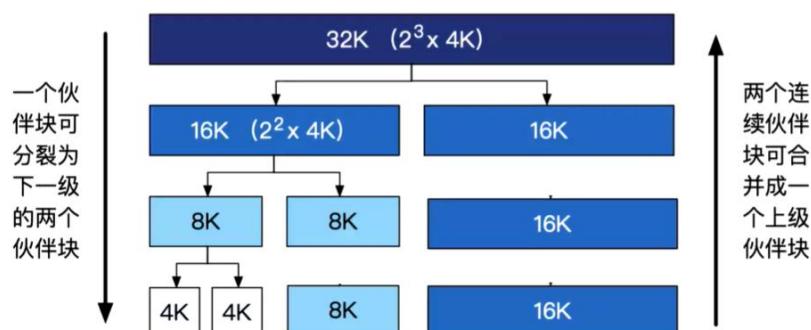
- 换出操作：物理内存不够时
  - OS选择不常用的物理内存（不同的选择策略）
  - OS将内存中的数据写入磁盘块，并记录磁盘块与内存的关联
  - OS更新页表，将对应页表项的valid bit设置为0

- 换入操作：当换出的页被访问时，触发page fault
  - OS判断该地址所在页被换出，找到对应的磁盘块
  - OS分配空闲的物理内存页；若没有空闲页，则再次进行换出操作
  - OS将磁盘块中的数据读入前一步找到的内存页
  - OS更新页表，将对应页表项的valid bit设置为1

物理地址的管理很关键，因为我们通常是拿到一大块物理内存（如：4G）。我们可以很细地管理，如 4K 4K 管理，但是这一定会出问题。比如我们需要 1G 内存，那我们以 4K 的粒度做根本做不完。所以一定是层级式的伙伴系统。一个伙伴块可以分裂和合并。一个伙伴块知道自己多大，它可以算出 flip bit，一下子找到另一个 buddy 在什么地方，就可以立马合并。

## 伙伴系统 ( buddy system )

- 伙伴系统：分裂与合并

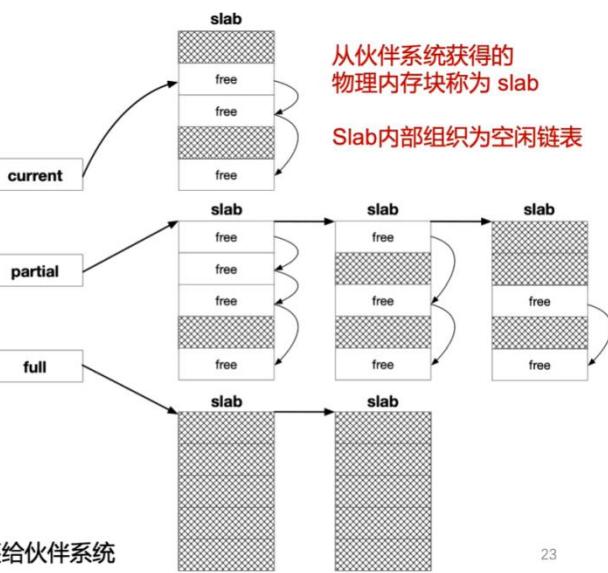


对于 SLUB，它会把常用的小数据结构做好分类，一旦不够了，就从伙伴系统要大块内存过来，一下子往里填满数据结构（如 dentry）。别人要 malloc dentry 的时候就从这里拿一个过去。所以 SLUB 和伙伴系统是互补的。

## ▶ SLUB机制

三个指针

- current仅指向一个 slab
- partial指向未满slab链表
- full指向全满slab链表



SWAP 会有自己的策略。

## Swap与页替换策略

- 常见的替换策略

- FIFO、LRU/MRU、时钟算法、随机替换 ...

- 替换策略评价标准

- 缺页发生的概率 ( 参照理想但不能实现的OPT策略 )
- 策略本身的性能开销
  - 如何高效地记录物理页的使用情况 ?
  - 页表项中Access/Dirty Bits

- Thrashing Problem

但是我们不能破坏掉工作集。如果我们破坏掉工作集，如最小工作集是 2M 内存，我们只分配 1.5M，那么就一定会发生颠簸。所以工作集可以很好的防止颠簸，如果内存不够，干脆停止加载。

## 跟踪工作集w(t, x)

- 工作集时钟中断固定间隔发生，处理函数扫描内存页

- 若访问位为1，则表示此次tick中被访问
  - 记录上次使用时间为当前时间
- 若访问位为0，则表示此次tick中未访问
  - Age = 当前时间 – 上次使用时间
  - 若Age大于设置的x，则不在工作集
- 最后，OS将所有访问位清0
  - 注意：访问位由CPU在访问时设为1

当前时间 : 2020

|      |   |
|------|---|
| 2010 | 1 |
| 2000 | 1 |
| 1970 | 0 |
| 1990 | 0 |

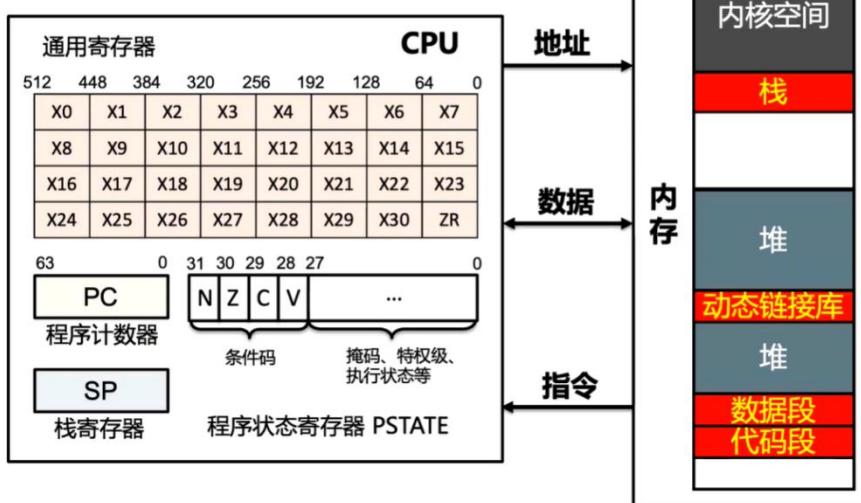
上次使用时间 访问位

虚拟内存的核心是翻译的时机，以及硬件、软件各自在做什么事情。而物理内存管理主要是伙伴系统和 SLUB。

## 进程和调度

在有了虚拟地址抽象之后，进程就比较简单了。进程里有两个上下文的概念。我们一般很多时候不是很明白上下文是什么意思。同一个名字，在不同上下文下的意思是不一样的，比如 `a.txt` 在不同目录上下文对应的文件是不同的。对于 CPU 来说也是一样的，`X30` 寄存器也要指定是用户的还是内核态的、还是 hypervisor 的 `X30`，这就需要上下文，这个上下文就是当前到底谁在执行。

## 处理器上下文



当前一个进程在运行，这个进程 `syscall` 到内核，此时内核的上下文就是这个进程，当它要访问一个文件的时候，它是以这个进程的上下文的权限来判断有没有权限访问这个文件的。这个上下文一般是 `current` 指针指向 `PCB`，意味着进程下陷到 `kernel` 中使用的内核栈是和 `process` 对应的。这个上下文会在调用 `schedule` 之后修改 `current`，修改为下一个调度的进程，栈理论上也是要修改的。但是 `ChCore` 是没有修改栈的，只是修改了 `current`，这是因为 `ChCore` 从内核态返回到用户态的时候，栈是空的。所以 `ChCore` 在做进程切换的时候，`ChCore` 把 `current` 指向的 `PCB` 修改了，但是没有修改栈，但是它在处理器上的要修改的栈是修改了。所以 `ChCore` 依旧是在使用上一个进程的栈，切换到进程再回来的时候才用了新的进程的栈。

处理器上下文就是寄存器（硬件概念），内核的上下文指的就是当前使用的进程（`current` 指针指过去的 `PCB`，软件概念）。进程在调用 `syscall` 的时候，在内核里，用的是进程上下文，用的是进程的栈，这时候来了一个中断，要去读硬盘，这时候应该是谁的上下文呢？理论上应该是硬盘的上下文，因为这个硬盘操作和我们进程无关。这就是 Linux 之后发明了中断上下文的概念，我们每个 `call` 有一个上下文。`syscall` 来了以后立马把栈换成中断上下文，回来了再修改。如果不使用中断上下文，我们也可以使用被打断的进程的内核栈临时使用一下。

# 如何表示进程：进程控制块 ( PCB )

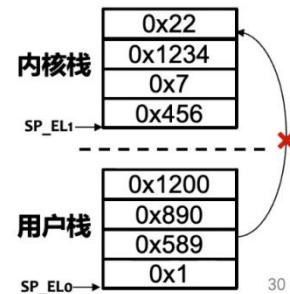
- 每个进程都对应一个元数据，称为“进程控制块” PCB
  - 进程控制块存储在内核态（为什么？）
- 想一想：进程控制块里至少应该保存哪些信息？
  - 独立的虚拟地址空间
  - 独立的执行上下文

```
1 // ChCore 中的 PCB——process
2 struct process {
3 // 上下文
4 struct process_ctx *process_ctx;
5 // 虚拟地址空间
6 struct vmspace *vmspace;
7 };
```

## 进程的内核态执行：内核栈

- 为什么需要“又一个栈”（内核栈）？
  - 进程在内核中依然执行代码，有读写临时数据的需求
  - 进程在用户态和内核态的数据应该相互隔离，增强安全性

- AArch64实现：两个栈指针寄存器
  - SP\_EL1, SP\_ELO
  - x86只有一个栈寄存器，需要保存恢复

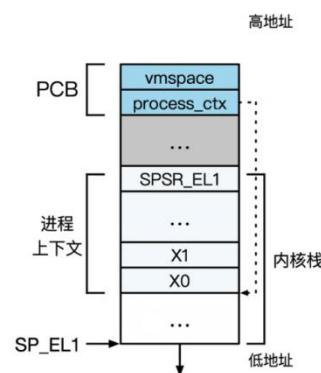


在 ARM 上，有 SPEL1 和 SPEL0，是不需要保存的。

## 上下文与其他内核数据结构

- 与进程相关的三种内核数据结构：PCB、上下文、内核栈

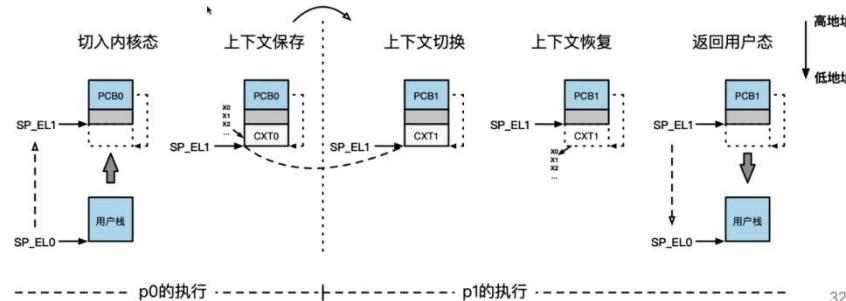
- PCB保存指向上下文的引用
- 上下文的位置固定在内核栈底部



这个栈的切换还包含了 current 的切换。

# 上下文切换栈变化全过程

- 共涉及两次权限等级切换、三次栈切换
- 内核栈的切换是线程切换执行的“分界点”

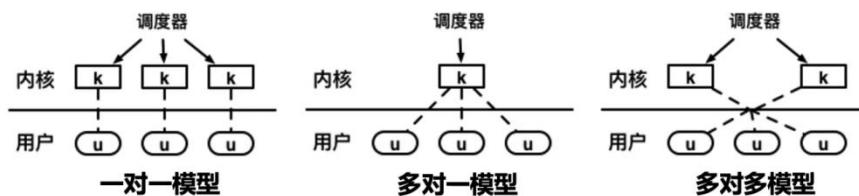


32

然后就是线程模型。因为单线程进程是没有利用多核的。线程可以认为是默认地址空间共享的两个进程。有些内核态可以做到纯内核态线程。

## 线程模型

- 线程模型表示了用户态线程与内核态线程之间的联系
  - 多对一模型：多个用户态线程对应一个内核态线程
  - 一对一模型：一个用户态线程对应一个内核态线程
  - 多对多模型：多个用户态线程对应多个内核态线程



### 多线程模型：一对一模型、多对一模型和多对多模型

迄今为止，我们只是泛泛地讨论了线程。不过，有两种不同方法来提供线程支持：用户层的用户线程或内核层的内核线程。

用户线程位于内核之上，它的管理无需内核支持；而内核线程由操作系统来直接支持与管理。几乎所有的现代操作系统，包括 Windows、Linux、Mac OS X 和 Solaris，都支持内核线程。

最终，用户线程和内核线程之间必然存在某种关系。本节研究三种常用的方法建立这种关系：多对一模型、一对一模型和多对多模型。

### 多对一模型

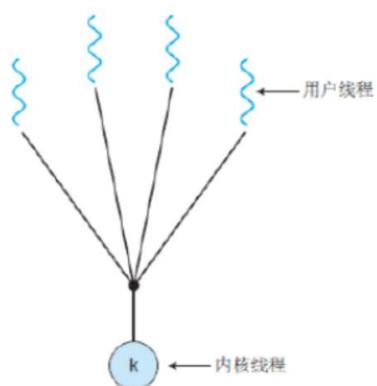


图 1 多对一模型

多对一模型（图 1）映射多个用户级线程到一个内核线程。

线程管理是由用户空间的线程库来完成的，因此效率更高。不过，如果一个线程执行阻塞系统调用，那么整个进程将会阻塞。再者，因为任一时间只有一个线程可以访问内核，所以多个线程不能并行运行在多处理核系统上。

Green threads 线程库为 Solaris 所采用，也为早期版本的 SunOS 所采纳，它就使用了多对一模型。然而，现在几乎没有系统继续使用这个模型，因为它无法利用多个处理核。

### 一对一模型

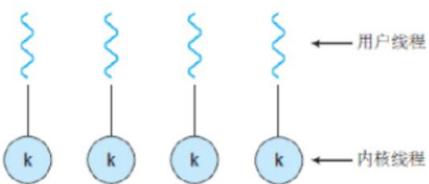


图 2 一对一模型

一对一模型（图 2）映射每个用户线程到一个内核线程。

该模型在一个线程执行阻塞系统调用时，能够允许另一个线程继续执行，所以它提供了比多对一模型更好的并发功能；它也允许多个线程并行运行在多处理器系统上。

这种模型的唯一缺点是，创建一个用户线程就要创建一个相应的内核线程。由于创建内核线程的开销会影响应用程序的性能，所以这种模型的大多数实现限制了系统支持的线程数量。Linux，还有 Windows 操作系统的家族，都实现了一对一模型。

### 多对多模型

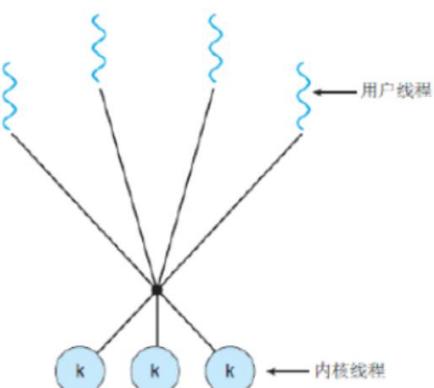


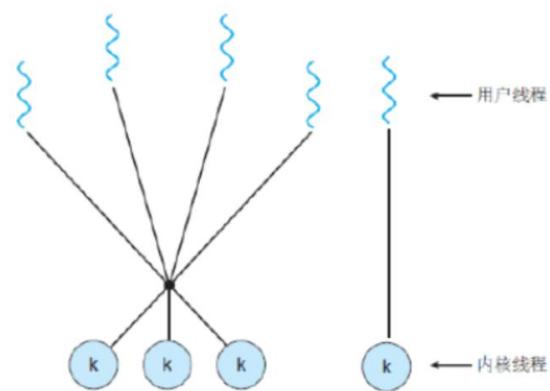
图 3 多对多模型

多对多模型（图 3）多路复用多个用户级线程到同样数量或更少数量的内核线程。内核线程的数量可能与特定应用程序或特定机器有关（应用程序在多处理器上比在单处理器上可能分配到更多数量的线程）。

现在我们考虑一下这些设计对并发性的影响。虽然多对一模型允许开发人员创建任意多的用户线程，但是由于内核只能一次调度一个线程，所以并未增加并发性。虽然一对一模型提供了更大的并发性，但是开发人员应小心，不要在应用程序内创建太多线程（有时系统可能会限制创建线程的数量）。

多对多模型没有这两个缺点：开发人员可以创建任意多的用户线程，并且相应内核线程能在多处理器系统上并发执行。而且，当一个线程执行阻塞系统调用时，内核可以调度另一个线程来执行。

多对多模型的一种变种仍然多路复用多个用户级线程到同样数量或更少数量的内核线程，但也允许绑定某个用户线程到一个内核线程。这个变种，有时称为双层模型（图 4）。



Solaris 操作系统在第 9 版以前支持这种双层模型；但从第 9 版后，就使用了一对一模型。

## 进程vs. 线程

- **线程和进程的相似之处：**
  - 都可以与其他进程/线程并发执行（可能在不同核心上）
  - 都可以进行上下文切换
    - 引入线程后，调度管理单位由进程变为线程

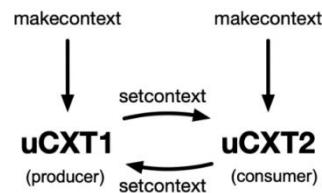


# 纤程（用户态线程）

- 比线程更加轻量级的运行时抽象
  - 不单独对应内核线程
  - 一个内核线程可以对应多个纤程（多对一）
- 纤程的优点
  - 不需要创建内核线程，开销小
  - 上下文切换快（不需要进入内核）
  - 允许用户态自主调度，有助于做出更优的调度决策

## 纤程的优势

- 纤程切换及时
  - 当生产者完成任务后，可直接用户态切换到消费者
  - 对该线程来说是最优调度（内核调度器和能做到）
- 高效上下文切换
  - 切换不进入内核态，开销小
  - 即时频繁切换也不会造成过大开销



有了多个 thread 之后，我们就要调度它。但是调度器非常难。

## 调度器的目标

- 降低周转时间：任务第一次进入系统到执行结束的时间
- 降低响应时间：任务第一次进入系统到第一次给用户输出的时间
- 实时性：在任务的截止时间内完成任务
- 公平性：每个任务都应该有机会执行，不能饿死
- 开销低：调度器是为了优化系统，而非制造性能BUG
- 可扩展：随着任务数量增加，仍能正常工作
- ...

MLFQ 是一个很好的调度器，没有很多先验假设，而是动态更新优先级，许多调度器都

是基于 MLFQ 去做的。

## MLFQ

- **Multi-Level Feedback Queue**
  - 通过观察任务的历史执行，动态确定任务优先级
    - 无需任务的先验知识
  - 同时达到了周转时间和响应时间两方面的要求
    - 对于短任务，周转时间指标近似于 SJF
    - 对于交互式任务，响应时间指标近似于 RR
  - 可以避免长任务的饿死
- **许多著名系统的调度器是基于MLFQ实现的**
  - BSD, Solaris, Windows NT 和后续Windows操作系统

## IPC

进程一旦分开了，它们之间怎么去配合就是下一个问题。管道的抽象是字节流，两端一边写一边 `read`。共享内存很底层，本质就是在用户态管理一块共享的内存区间。消息队列也是要进内核的，消息是由内核来管理的。信号量、信号、`socket` 大家都学过了。

## 常见IPC的类型

| IPC机制 | 数据抽象 | 参与者  | 方向    |
|-------|------|------|-------|
| 管道    | 字节流  | 两个进程 | 单向    |
| 共享内存  | 内存区间 | 多进程  | 单向/双向 |
| 消息队列  | 消息   | 多进程  | 单向/双向 |
| 信号量   | 计数器  | 多进程  | 单向/双向 |
| 信号    | 事件编号 | 多进程  | 单向    |
| 套接字   | 数据报文 | 两个进程 | 单向/双向 |

共享内存的缺点就是缺乏通知机制。根本原因是这个机制太底层了，没有 OS 的参与。而 OS 是可以把应用调度、唤醒、`sleep` 的。

## 共享内存

- **缺少通知机制**
  - 若轮询检查，则导致CPU资源浪费
  - 若周期性检查，则可能导致较长的等待时延
  - **根本原因**：共享内存的抽象过于底层；缺少OS更多支持
- **TOCTTOU ( Time-of-check to Time-of-use ) 问题**
  - 当接收者直接用共享内存上的数据时，可能存在被发送者恶意篡改的情况（发生在接收者检查完数据之后，使用数据之前）
  - 这可能导致buffer overflow等问题

所以在这种情况下，我们就需要提出消息机制了。如果 OS 没有支持消息机制，就会浪费掉时间。

## 共享内存和消息传递的对比

- **共享内存的优势：理论上可以实现零内存拷贝的数据传输**
  - 消息传递的方式通常：发送者用户态内存拷贝到内核内存，再拷贝到接收者用户态内存（**两次拷贝**）
- **消息传递的优势：**
  - 抽象更简单，并且能够较好支持变长的消息（共享内存往往需要用户态软件封装来实现这一点）
  - 不需要轮询，通常包含内核支持的通知机制
  - 安全性保证通常更强（有内核等保证），并且不会破坏发送者和接收者进程的内存隔离性

消息队列更灵活，可以定义 filter。进程间通信有很多接口，可以用内存接口、文件接口也可以用消息队列。

## 消息队列 VS. 管道

- **缓存区设计：**
    - 消息队列：链表的组织方式，动态分配资源，可以设置很大的上限
    - 管道：固定的缓冲区间，分配过大资源容易造成浪费
  - **消息格式：**
    - 消息队列：带类型的数据
    - 管道：数据（字节流）
  - **连接上的通信进程：**
    - 消息队列：可以有多个发送者和接收者
    - 管道：两个端口，最多对应两个进程
  - **消息的管理：**
    - 消息队列：FIFO + 基于类型的查询
    - 管道：FIFO
- 消息队列更加灵活易用，但是实现也更加复杂

## 迁移线程

有一类很有意思的是迁移线程，一个 client 线程要穿透多个地址空间，它要穿越成 server，做完该做的事情之后再回到。它只是一个线程，期间 OS 在 client 调用 server 并没有做调度。client 在进入 server 之后，client 会少一个线程、server 会多一个线程，等到 client 返回的时候再恢复。

这样的好处就是避免了无谓的上下文切换，使得调用速度加快。但是我们还是要进 kernel 换页表，有没有办法直接在用户态切换也是一个很有意思的方向。

## 迁移线程：将Client运行在Server的上下文

- **为什么需要做控制流转换？**
  - 使用Server的代码和数据
  - 使用Server的权限 (如访问某些系统资源)
- **只切换地址空间、权限表等状态，不做调度和线程切换**



## 同步

同步是比较难的地方。同步的难点在于人的脑子很难理解同时发生的事情。我们的思路就是从问题出发讲一些场景，再加一些同步原语。到了 OS，我们要融会贯通，要在各种新的场景中应用，而不局限在原语本身。

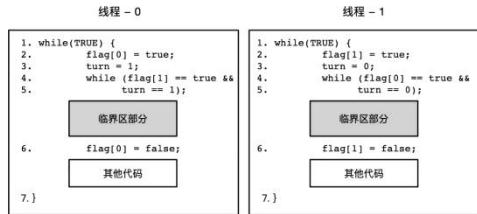
## 竞争条件与同步原语

- **多线程问题：竞争条件**
- **同步原语**
  - 互斥锁：保证**互斥访问**
  - 条件变量：提供**睡眠/唤醒**
  - 信号量：**资源管理**
  - 读写锁：**区分读者**提高并行度
- **同步原语带来的问题**
  - 死锁 **检测、预防与避免**

## 多线程计数中的数据竞争

|                   |                                   |
|-------------------|-----------------------------------|
| 线程1               | 线程2                               |
| a++;              | a++;                              |
| 初始值为3             |                                   |
| T0 reg_a = a; (3) |                                   |
|                   | reg_a = reg_a + 1; reg_a = a; (3) |
| T2 a = reg_a; (4) | reg_a = reg_a + 1;                |
|                   | a = reg_a; (4)                    |
| 丢失了线程1的更新         |                                   |

## 软件解决方案：皮特森算法



flag: 表示需要进入临界区

turn: 表示当前哪个线程可以进入临界区

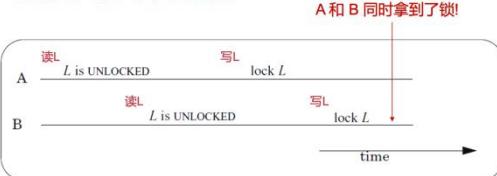
## 互斥锁的接口：拿锁和放锁

### • 互斥锁 ( Mutual Exclusive Lock ) 接口

- Lock(lock) : 尝试拿到锁 "lock"
  - 若当前没有其他线程拿着lock，则拿到lock，并继续往下执行
  - 若lock被其他线程拿着，则不断循环等待放锁 ( busy loop )
- Unlock(lock)
  - 释放锁

- 可保证同时只有一个线程能够拿到锁

## 用原子指令实现锁



操作-1: 读L，检查状态是否为Locked

操作-2: 写L，将其状态设置为Locked

} 根本原因：这两步并非原子完成！

硬件方法：用原子指令来保证两步是原子的！

## 使用 Test-and-Set 实现 Spin Lock

### 原子指令：Test-and-Set

#### • 历史

- 1960年代初期，Burroughs B5000首先引入

```
1 int TestAndSet(int *old_ptr, int new) {
2 int old = *old_ptr; // fetch old value at old_ptr
3 *old_ptr = new; // store 'new' into old_ptr
4 return old; // return the old value
5 }
```

```
1 typedef struct __lock_t {
2 int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6 // 0 indicates that lock is available, 1 that it is held
7 lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11 while (TestAndSet(&lock->flag, 1) == 1)
12 ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16 lock->flag = 0;
17 }
```

## 排号锁 ( Ticket Lock )

思考：是否满足解决临界区问题的三个必要条件？

- 互斥访问 ✓
- 有限等待 ?
  - 按照顺序，在前序竞争者保证有段时间释放时，可以达到有限等待
- 空闲让进\* ✓

```
void lock(int *lock) {
 volatile unsigned my_ticket =
 atomic_FAA(&lock->next, 1);
 while(lock->owner != my_ticket)
 /* busy waiting */;
}

void unlock(int *lock) {
 lock->owner++;
}
```

排号锁实现

\*这里我们认为硬件能够确保原子操作make progress

上面就是使用 lock 实现互斥这件事情。同步原语要做到互斥访问和协调访问（讲究先后顺序，而互斥是不保证谁先谁后的）。条件变量是保证先后的一种方法，也就是满足条件的人才可以进入临界区，sender 就应该在 receiver 之前，producer 就应该在 consumer 之前。并行最难的地方就是出现并发访问错误和顺序错误。

# 条件变量的接口

提供的两个接口：

等待的接口： 等待需要在临界区中

```
void cond_wait(struct cond *cond, struct lock *mutex);
```

1. 放入条件变量的等待队列
2. 阻塞自己同时释放锁：即调度器可以调度到其他线程
3. 被唤醒后重新获取锁

唤醒的接口：

```
void cond_signal(struct cond *cond);
```

1. 检查等待队列
2. 如果有等待者则移出等待队列并唤醒

条件变量的接口中要传入一个 lock，这个 lock 会和 yield\_wait 的逻辑整合起来。我们 wait 是要 sleep 的，我们不能带着一个锁去 sleep。要保证这件事情，我们必须在拿了一把锁之后（acquire lock）再放锁（release lock）。这是我们说比较 tricky 的地方。

```
yield_wait(): // called by wait() ←
id = cpus[CPU].thread
cpus[CPU].thread = null
threads[id].sp = SP
SP = cpus[CPU].stack

do:
 id = (id + 1) mod N
 release(t_lock)
 enable_interrupt()
 disable_interrupt()
 acquire(t_lock)
 while threads[id].state != RUNNABLE
 SP = threads[id].sp
 threads[id].state = RUNNING
 cpus[CPU].thread = id
```

## YIELD\_WAIT()

```
wait(cv, lock):
 disable_interrupt()
 acquire(t_lock)
 release(lock)
 threads[id].cv = cv
 threads[id].state = WAITING
 yield_wait()
 release(t_lock)
 enable_interrupt()
 acquire(lock)
```

接下来是信号量，信号量的核心是 PV 原语。它的最大特点是有数量的，也就是多少资源和多少人的关系。如果资源是 n 个，那么我们信号量就是 n。如果资源是 1 个，就退化成了一个一般的锁，但是一个锁不能进化成 n 个资源的 PV。所以 PV 更通用一些，它的实现是 lock + cv（条件变量）

# 信号量（PV原语）

信号量：协调（阻塞/放行）多个线程共享有限数量的资源



Edsger W. Dijkstra

语义上：信号量的值记录了当前可用资源的数量

提供了两个原语用于等待/消耗资源

P操作：消耗资源      cnt代表剩余资源数量

```
void sem_wait(sem_t *sem) {
 while(sem->cnt <= 0)
 /* Waiting */;
 S--; 注意：只展示语义，并非真实实现
}
```

P操作：荷兰语Passeren，相当于pass

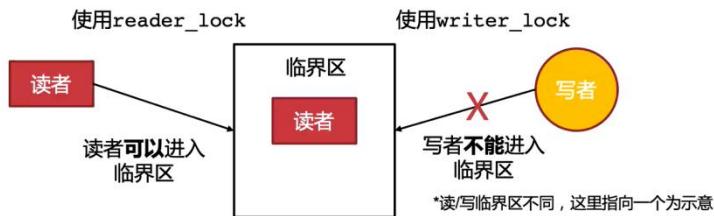
而读写锁是读者可以都进，而写者可以 exclusive。这是一个优化。

## 读写锁

互斥锁：所有的线程均互斥，同一时刻只能有一个线程进入临界区

对于部分只读取共享数据的线程过于严厉

读写锁：区分读者与写者，允许读者之间并行，读者与写者之间互斥



四种原语分别为 spin lock、conditional variable（条件变量）、信号量和读写锁。

死锁的四个条件是充分且必要，必须要满足这 4 个条件。要解除死锁必须破坏其中一个条件。

## 死锁产生的原因

• 互斥访问

• 持有并等待

• 资源非抢占

• 循环等待

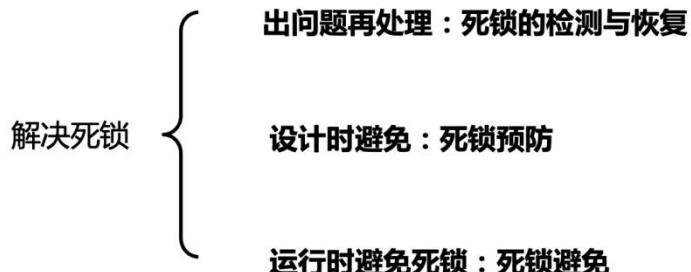
A等B，B等A

```
void proc_A(void) {
 lock(A);
 /* Time T1 */
 lock(B);
 /* Critical Section */
 unlock(B);
 unlock(A);
}

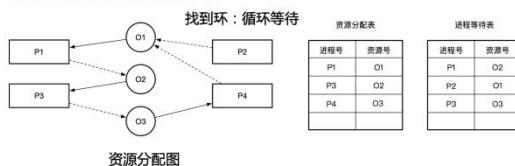
void proc_B(void) {
 lock(B);
 /* Time T1 */
 lock(A);
 /* Critical Section */
 unlock(A);
 unlock(B);
}
```

T1时刻的死锁

## 如何解决死锁？



## 检测死锁与恢复



- 如何恢复？打破循环等待！
- 直接 kill 所有循环中的线程
  - Kill 一个，看有没有环，有的话继续 kill
  - 全部回滚到之前的某一状态

## 死锁预防：四个方向

```
void proc_A(void) {
 lock(A);
 /* Time T1 */
 lock(B);
 /* Critical Section */
 unlock(B);
 unlock(A);
}
```

- 1、避免互斥访问：通过其他手段（如代理执行）
- 2、不允许持有并等待：一次性申请所有资源

3、资源允许抢占：需要考虑如何恢复

- 4、打破循环等待：按照特定顺序获取资源
- 对所有资源进行编号
  - 让所有线程递增获取

A : 1号 B : 2号：必须先拿锁A，再拿锁B

任意时刻：获取最大资源号的线程可以继续执行，然后释放资源

银行家算法现在也不常用，408 和面试经常会考。

## 银行家算法：安全性检查

四个数据结构：

M个资源 N个线程

- 全局可利用资源：Available[M]
- 每线程最大需求量：Max[N][M]
- 已分配资源：Allocation[N][M]
- 还需要的资源：Need[N][M]

银行家算法的核心：

- 所有线程获取资源需要通过管理者同意
- 管理者预演会不会造成死锁
  - 如果会造成：阻塞线程，下次再给
  - 如果不会造成：给线程该资源

## MCS 锁（重要）

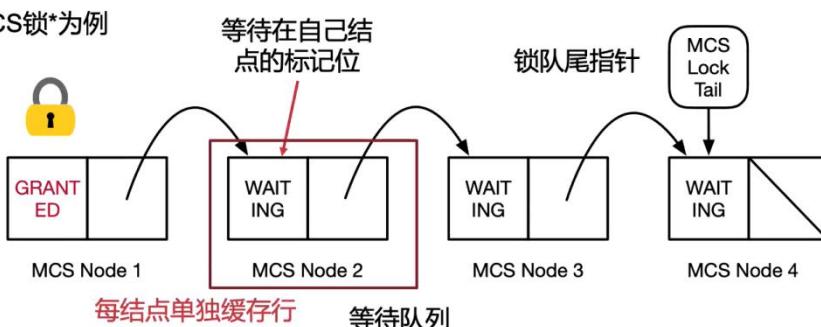
有了这些之后，我们开始讲一些高级的话题，比如 MCS 锁。一旦我们有了锁，并且使用 spin lock，多核情况下抢这个 bit 会导致大量的 cache miss。很多个核抢占同一条 cache line，就会导致大量的问题。所以我们希望让大家不要去抢同一条 cache line。

所以我们允许大家去等，每个人只要等 1 个 bit 就行，一旦有人放锁了，那个人只需要改下一个人的 lock bit 就行。所以每个人看着自己的 bit 等待别人修改，这样比大家去抢同一个地方比较好。

## 如何解决可扩展性问题：MCS锁

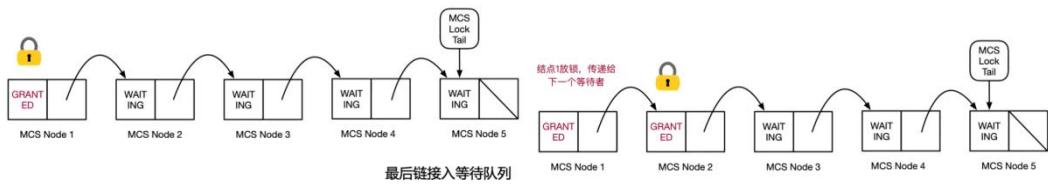
核心思路：在关键路径上避免对单一缓存行的高度竞争

以MCS锁\*为例

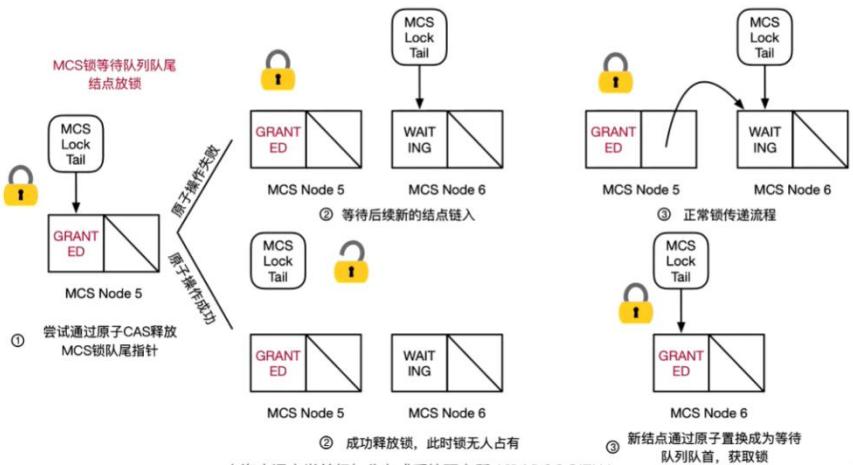


## MCS锁：新的竞争者加入等待队列

## MCS锁：锁持有者的传递



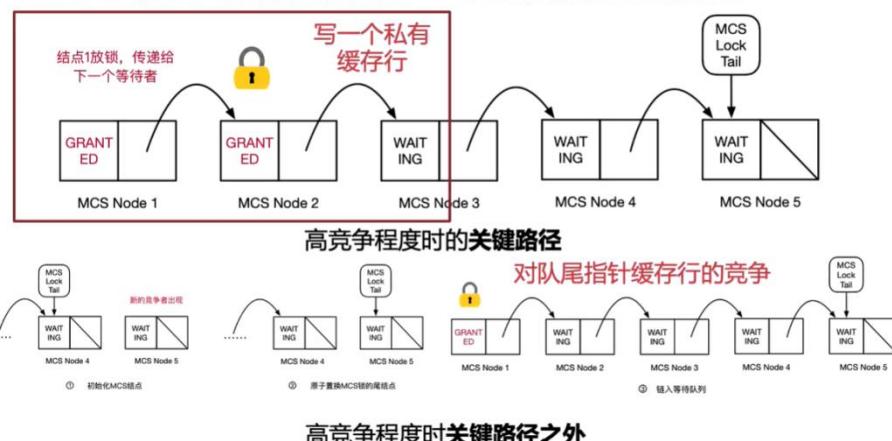
## MCS锁：放锁流程



性能上它肯定比原来的设计要好。

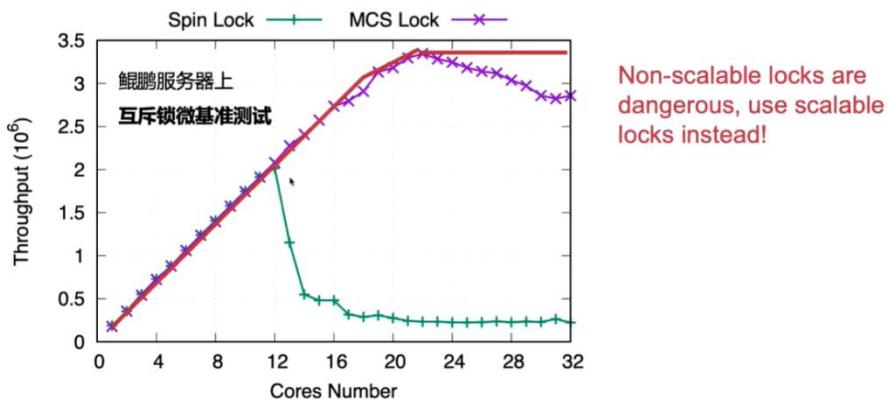
## MCS锁：性能分析

不再会高频竞争全局缓存行



# MCS锁：性能分析

核心思路：在关键路径上避免对单一缓存行的高度竞争



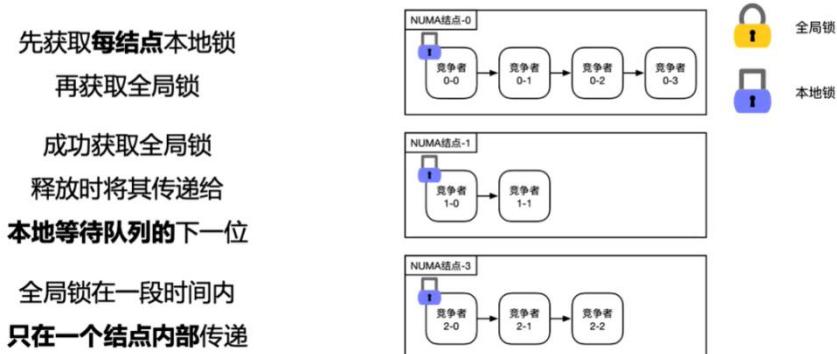
在这张图的实验结果中，如果给大家这张图，问为什么 MCS 锁好，为什么 spin lock 在多核情况下会 collapse，需要大家讲明白理由。我们需要知道 MCS 锁的原理和代码，能够读懂代码并且做出相应的修改。

## cohort 锁

MCS 锁之后就是 cohort 锁。它是针对访问 NUMA 来设计的，原因就是 CPU 访问不同的内存的速度是不一样的，离 CPU 远的访问速度就慢。

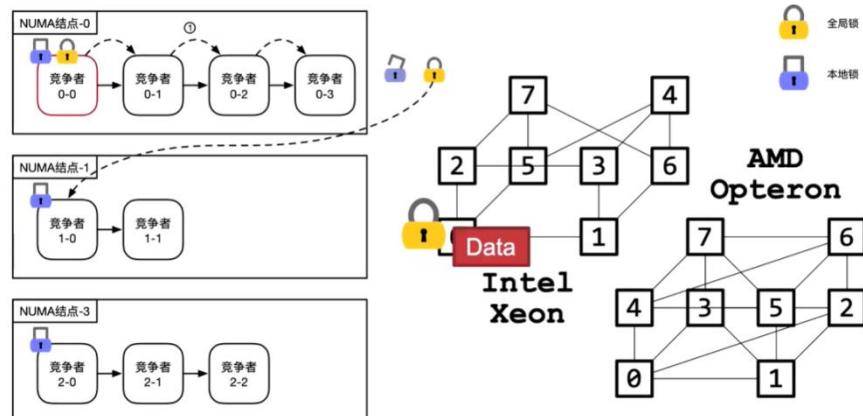
## NUMA-aware 设计：以 cohort 锁\*为例

核心思路：在一段时间内将访存限制在本地



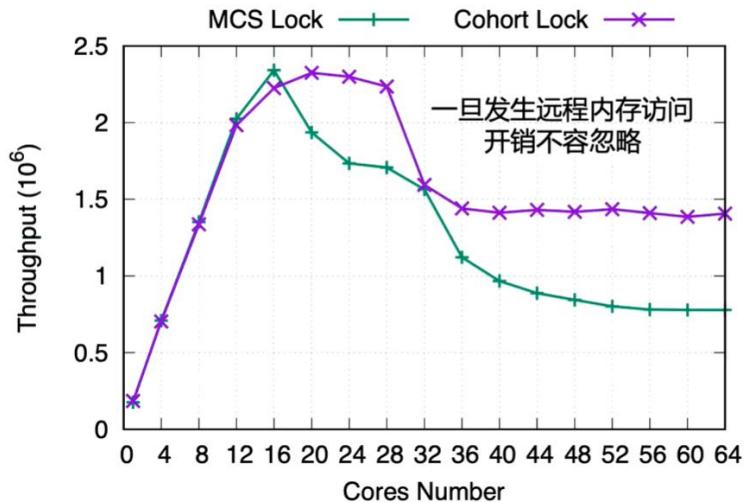
为了性能，NUMA-aware 的 lock 的设计就比较自然了。我们分为两级 local 和 global 的。现在一个 NUMA 上处理，再把 global 传给下一个 NUMA 节点处理。

## NUMA-aware设计：以cohort锁\*为例



这样，它带来的好处就是，还是可以比较好的维持住性能。

## NUMA-aware设计：以cohort锁\*为例

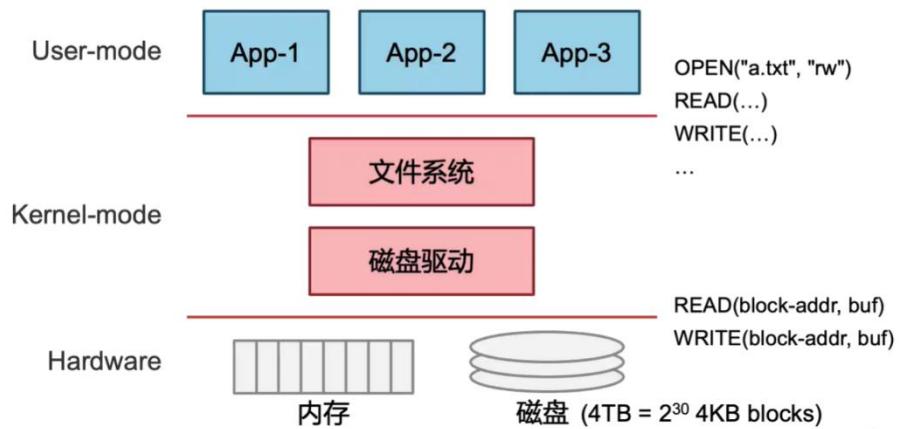


我们讲完了进程、进程的通信、进程的调度、线程和线程之间是怎么同步的。

## 文件系统和崩溃一致性

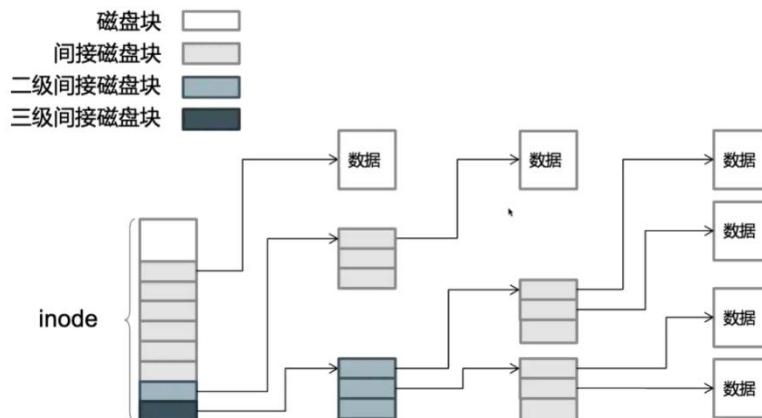
文件系统和前面的章节有很大的不一样，它是和数据相关的。一开始是和硬盘等设备是相关的，随着时代的发展，文件系统和内存的关联也越来越强。文件系统对外的接口就是 open/read/write。

# 文件系统的位置



把文件串起来就是 **inode**，这个大家都很熟悉了。

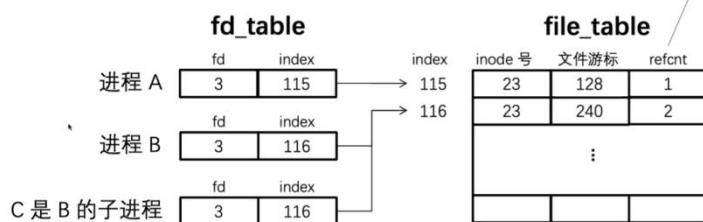
## inode：多级磁盘块索引，记录大文件



在内存中，记录了文件的动态的信息，尤其是 **file cursor**。所以内存里比起磁盘上多记录了一些和当前运行状态相关的数据结构。

## 对打开文件的管理

Note: this refcnt is not the refcnt of inode!



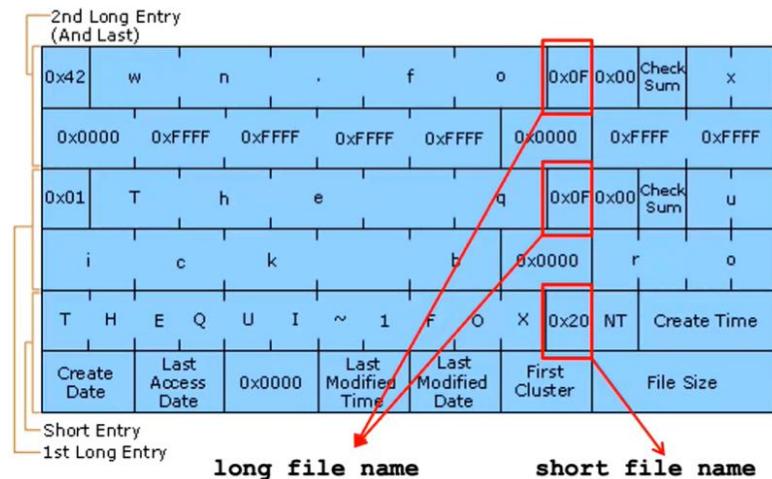
- 三个进程 A、B、C 都打开了 inode 号为 23 的文件
- 进程 A 和 B 不共享文件游标
- 进程 B 和 C 共享文件游标

然后我们讲了一些非 **inode** 的文件系统，比如 **FAT**，它其实是把文件名和 **inode** 内嵌在一起。在下图的目录项中，我们可以看到文件名，然后有相应的一个 **inode** 的值，指向某一个实际的 **first cluster**，元数据都记录在 **directory entry** 里。它这个做法就会导致文件名和 **inode** 没有严格拆开，而是把文件名和元数据打包在一起指向文件的数据。

这个带来的问题就是没有办法做 hard link, 因为 hard link 做的事情是多建立了一个文件名到 inode number 的映射。

## FAT32中的目录项

e.g. file with “The quick brown.fox” and “Thequi~1.fox”



有了 FAT 之后，我们整个系统里可能有多种文件系统，不同的磁盘可能跑了不同的文件系统。我们上层的文件系统需要使用一个系统把底层实现的异构性消除掉，这就是 VFS 做的事情。

VFS 提出了 vnode 的概念，其实就是内存中的 inode，对于 EXT4 这种 inode 文件系统来说，可以直接把 inode 填到 vnode 里去；而对于 FAT，需要主动地从磁盘中把元数据填到 vnode 的结构中，FAT 没有 owner 的概念，所以没有对应项。所以，我们把 FAT 的结构读进内存里变成了 vnode 的形式。

VFS 再往上，我们就可以假设有 vnode number，这个 vnode number 其实就是文件系统构造出来的内存里的数据，和真正的磁盘上的数据是没有关联的。

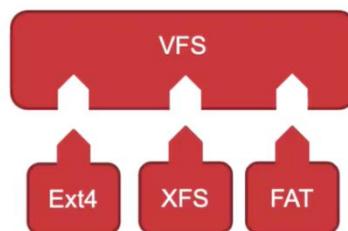
## Linux中的虚拟文件系统VFS

Linux的VFS定义了一些系列接口

具体的文件系统实现这些接口

如在读取一个inode的文件时

- VFS先找到该inode所属文件系统
- 再调用该文件系统的读取接口

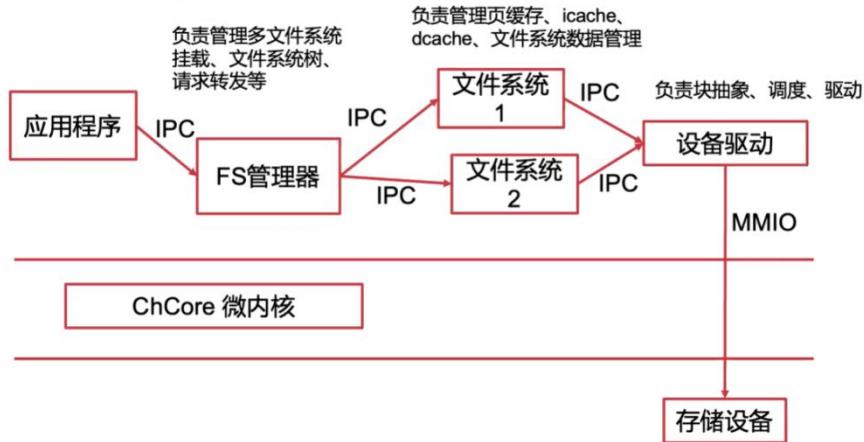


Windows的类似机制

- Installable File System

这种方案性能上可能有损失。

# ChCore中的文件与存储结构

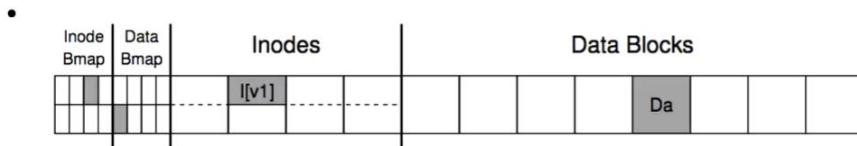


## 崩溃一致性

崩溃一致性大家不陌生，但是大家不一定能成体系。

- 追加文件数据操作会导致**三次磁盘写**
  - 假设：单次磁盘写能够保证 all-or-nothing

1. 将数据写入磁盘的数据块
2. 将inode写入磁盘的inode table
3. 更新inode和data在磁盘上的bitmap



所有的元数据写都是加了一个 **flush to disk** 的。它的好处就是当我们做 **fsck** 的时候，可以尽可能多地根据元数据的信息恢复到原来的样子。在这个场景下，我们最后把 **inode** 写入 **inode table** 中。前面都 **flush** 到磁盘了，最后再 **flush** 一下就可以保证原子性了。

## 方法-1：同步元数据写+fsck

若非正常重启，则运行fsck检查磁盘，具体步骤：

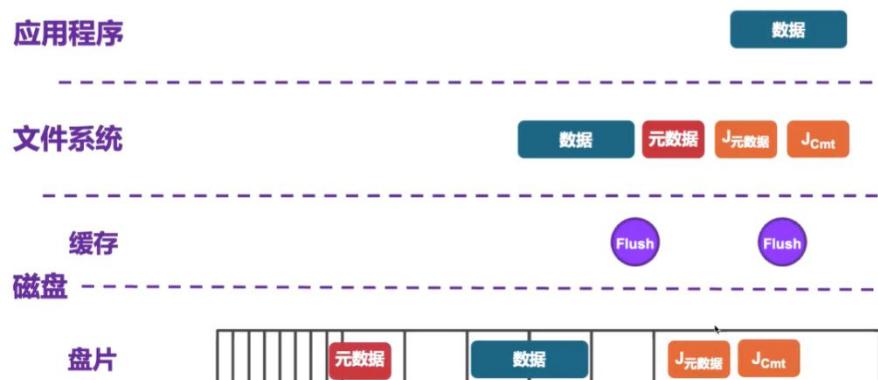
- 1. 检查superblock
  - 例：保证文件系统大小大于已分配的磁盘块总和
  - 如果出错，则尝试使用superblock的备份
- 2. 检查空闲的block
  - 扫描所有inode的所有包含的磁盘块
  - 用扫描结果来检验磁盘块的bitmap
  - 对inode bitmap也用类似方法

## 方法-2：日志 ( Journaling )

- 在进行修改之前，先将修改记录到日志中
- 所有要进行的修改都记录完毕后，提交日志
- 此后再进行修改
- 修改之后，删除日志

顺序：写数据和写元数据、flush、写 Journal commit、flush，最后 flush 元数据。如果元数据没来得及 flush，我们可以从日志中恢复。

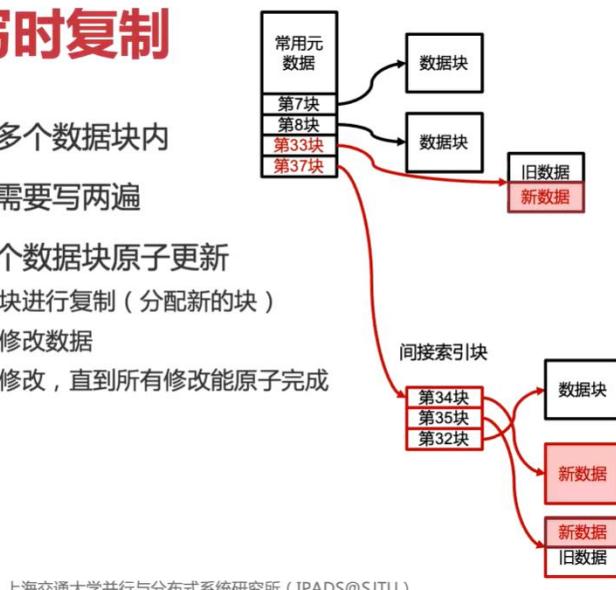
## Ordered Mode：两次Flush保证顺序



还有一个方法是 copy-on-write。也就是我们所有数据都不覆盖，我们只修改指针，这样就避免了 crash 的时候，数据一半新、一半旧的情况。因为我们最后的修改指针的方法是原子的。

# 文件中的写时复制

- 文件数据散落在多个数据块内
- 使用日志：数据需要写两遍
- 写时复制保证多个数据块原子更新
  - 将要修改的数据块进行复制（分配新的块）
  - 在新的数据块上修改数据
  - 向上递归复制和修改，直到所有修改能原子完成
  - 进行原子修改
  - 回收资源



但是写时复制有写放大的问题，上面一层层指针都要改。

还有一种方法就是 soft-update，它和同步元数据修改+fsck 的方法很像。相同点都要考虑谁最后一个做（让系统对外可见）。它不一样的地方在于，对磁盘的 4K 做了优化，它会去做 track。如果下一个操作依赖于上一个操作，它会先把下一个操作 rollback（让它先不做），如果上一个操作已经做了，我们再重新做下一个操作。这就导致比较复杂，并且和语义的耦合非常紧，带来了一些新的问题。

好处就是每次只需要写一次。

## Soft Updates的三个次序规则

### 1. 不要指向一个未初始化的结构

- 如：目录项指向一个inode之前，改inode结构应该先被初始化

### 2. 一个结构被指针指向时，不要重用该结构

- 如：当一个inode指向了一个数据块时，这个数据块不应该被重新分配给其他结构

### 3. 不要修改最后一个指向有用结构的指针

- 如：Rename文件时，在写入新的目录项前，不应删除旧的目录项

## Soft Updates

- 对于每个文件系统请求，将其拆解成对多个结构的操作
  - 记录对每个结构的修改内容（旧值、新值）
  - 记录这个修改依赖于那些修改（应在哪些修改之后持久化）
  - 如创建文件：
    1. 标记inode为占用（对bitmap的修改）
    2. 初始化inode（对inode的修改，依赖于1）
    3. 将目录项写入目录中（对目录文件的内容修改，依赖于1和2）

## 设备与 I/O

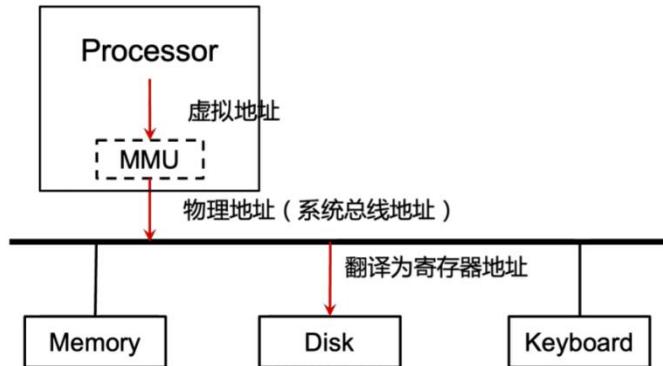
文件系统和设备是一类的，因为存储设备就是靠文件系统做的上层抽象。并且所有设备都可以认为是文件，复用文件的 API。但是设备不仅仅有数据流，还有控制流。这时候就要引入 `ioctl` 这个命令。设备驱动层的核心是让不同的厂商提供一个东西，而 I/O 子系统是把设备分成了块设备、字符设备和网络设备，可以很好地共用大部分代码。分类的作用是控制复杂度，用户态 IO 理论上可以绕过 OS 的 I/O 子系统，直接和硬件或者驱动打交道。

## 操作系统的I/O层次



一个硬件设备对于 OS 来说，抽象就是一组寄存器，我们把寄存器映射到内存地址之后，软件就可以以写物理地址的方式去操作我们的硬件。至于它是映射到用户态空间还是内核态空间，这就取决于我们的实现方式。

## 内存映射 I/O (MMIO)

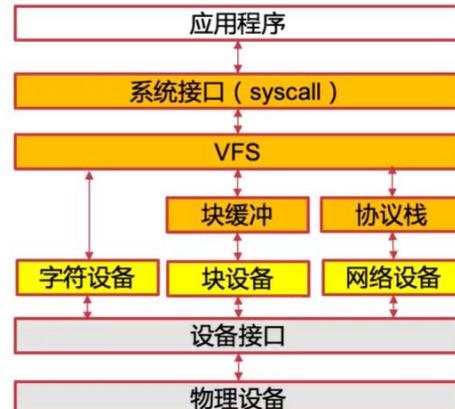


因为设备是文件，所以 `syscall` 下来之后，先进入 VFS，再往下转发。所以 VFS 甚至可以去操作非存储的设备的 `read/write`。

## 设备的逻辑分类

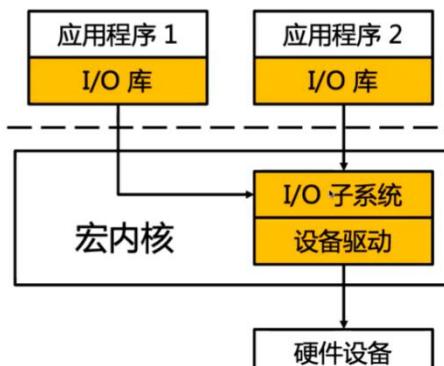
- **Linux设备分类**

- 字符设备
- 块设备
- 网络设备



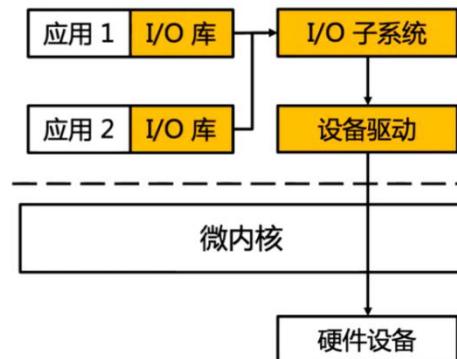
## 宏内核I/O架构

- 宏内核I/O架构
  - 设备驱动在内核态
  - 优势：通常性能更好
  - 劣势：容错性差
  - 中断形式为内核ISR
- 案例：
  - Linux、BSD
  - Windows



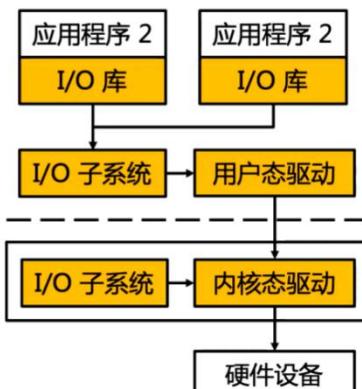
## 微内核I/O架构

- 微内核I/O架构
  - 设备驱动主体在用户态
  - 优势：可靠性和容错性更好
  - 劣势：IPC性能开销
  - 中断为用户态驱动线程
- 案例：
  - 谷歌Fuchsia手机系统
  - ChCore微内核系统



## 混合I/O架构

- 混合I/O架构
  - 设备驱动分解为用户态和内核态
  - 优势1：驱动开发和Linux内核解耦
  - 优势2：允许驱动以闭源形式存在，保护硬件厂商的知识产权
- 案例：
  - 谷歌安卓系统：硬件抽象层 ( HAL )
  - 华为鸿蒙系统：硬件驱动框架 ( HDF )

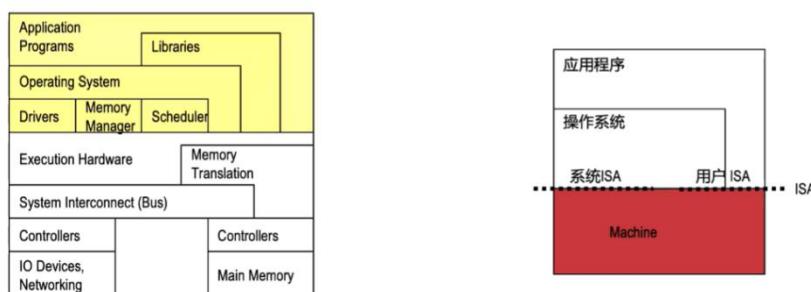


## 系统虚拟化

虚拟化就是要在非特权级运行一些特权级的软件。虚拟化就是用一层去实现另一层的接口，没有接口的创新。

### 如何定义虚拟机？

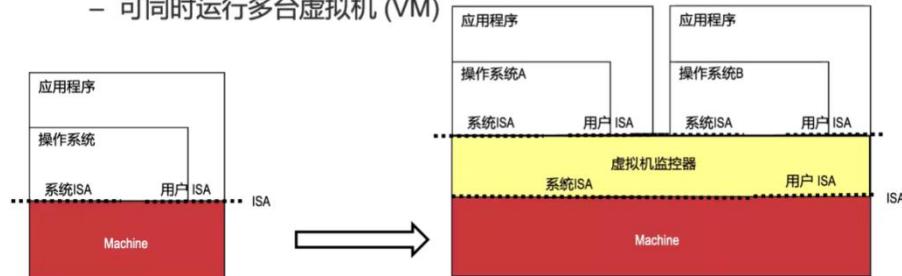
- 从操作系统角度看"Machine"
  - ISA 提供了操作系统和Machine之间的界限



# 虚拟机和虚拟机监控器

- **虚拟机监控器 (VMM/Hypervisor)**

- 向上层虚拟机暴露其所需要的ISA
- 可同时运行多台虚拟机 (VM)



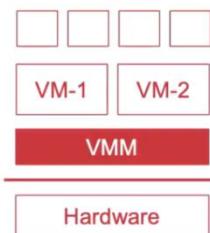
## Type-1 虚拟机监控器

- **VMM直接运行在硬件之上**

- 充当操作系统的角色
- 直接管理所有物理资源
  - 实现调度、内存管理、驱动等功能

- **性能损失较少**

- **例如Xen, VMware ESX Server**



QEMU/KVM 是连在一起讲的。KVM 只是一个 kernel 里的一个驱动，真正控制虚拟机的是 QEMU，跑 3 个虚拟机就有 3 个 QEMU。

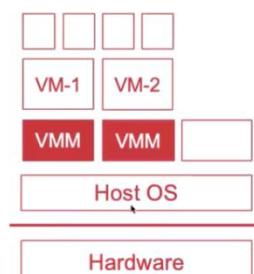
## Type-2 虚拟机监控器 ( 基于Host OS )

- **VMM依托于主机操作系统**

- 主机操作系统管理物理资源
- 虚拟机监控器以进程/内核模块的形态运行
- 易于实现和安装
- 例如：QEMU/KVM

- **思考：**

- Type-2类型有什么优势？



虚拟化分为 CPU 虚拟化、内存虚拟化和 IO 虚拟化。对 CPU 虚拟化来说，trap & emulate 是经典中的经典。如果所有的特权指令都可以被 trap，我们就可以通过 trap & emulate 实现一个 VM。

# CPU 虚拟化

## 处理器虚拟化：一种直接的实现方法

- **把虚拟机当做应用程序**
  - 将虚拟机监控器运行在EL1
  - 将客户操作系统和其上的进程都运行在EL0
  - 当操作系统执行系统ISA指令时下陷



但是 ARM 不是一个标准可虚拟化的架构，特权指令在非特权级执行的时候，不会下陷，所以就有四个方法。

## Trap & Emulate

- **Trap: 在用户态EL0执行特权指令将陷入EL1的VMM中**
- **Emulate : 这些指令的功能都由VMM内的函数实现**



## 方法2：二进制翻译

- 提出两个加速技术

- 在执行前**批量翻译**虚拟机指令
- **缓存**已翻译完成的指令

- 使用基本块(Basic Block)的翻译粒度 (**为什么?**)

- 每一个基本块被翻译完后叫代码补丁

- 不能处理自修改的代码(**Self-modifying Code**)

- 中断插入粒度变大

- 模拟执行可以在任意指令位置插入虚拟中断

- 二进制翻译时只能在基本块边界插入虚拟中断 (**为什么?**)

## 方法1：解释执行

- **使用软件方法一条条对虚拟机代码进行模拟**
  - 不区分敏感指令还是其他指令
  - 没有虚拟机指令直接在硬件上执行
- **使用内存维护虚拟机状态**
  - 例如：使用uint64\_t x[30]数组保存所有通用寄存器的值

## 方法4：硬件虚拟化

### 方法3：半虚拟化 (Para-virtualization)

- 协同设计
  - 让VMM提供接口给虚拟机，称为Hypercall
  - 修改操作系统源码，让其主动调用VMM接口
- Hypercall可以理解为VMM提供的系统调用
  - 在ARM中是HVC指令
- 将所有不引起下陷的敏感指令替换成超级调用
- 思考：这种方式有什么优缺点？

- x86和ARM都引入了全新的虚拟化特权级

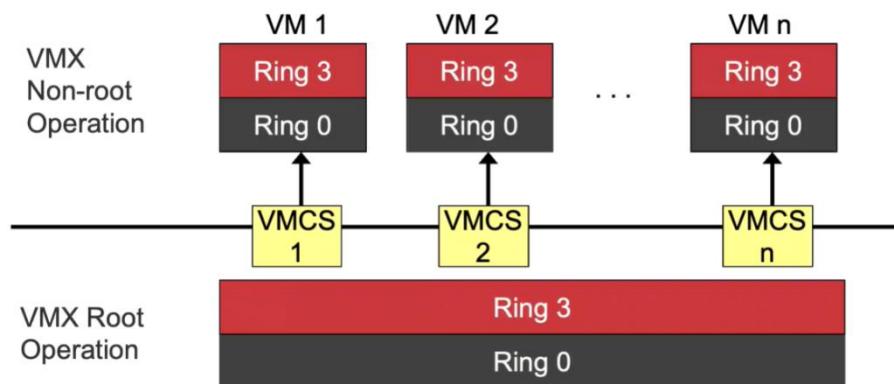
- x86引入了root模式和non-root模式
  - Intel推出了VT-x硬件虚拟化扩展
  - Root模式是最高特权级别，控制物理资源
  - VMM运行在root模式，虚拟机运行在non-root模式
  - 两个模式内都有4个特权级别：Ring0~Ring3

- ARM引入了EL2

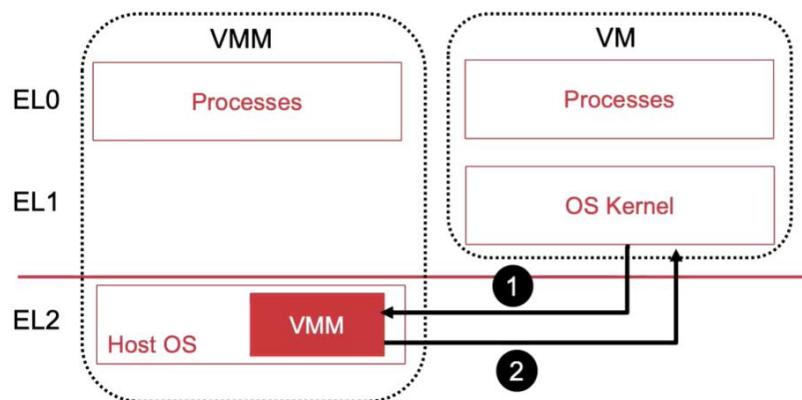
- VMM运行在EL2
  - EL2是最高特权级别，控制物理资源
  - VMM的操作系统和应用程序分别运行在EL1和EL0

硬件虚拟化的核心就是增加一个新的特权级，一旦出现特权指令就触发 VM\_EXIT。

## VT-x的执行过程



## ARMv8.1中的Type-2 VMM架构

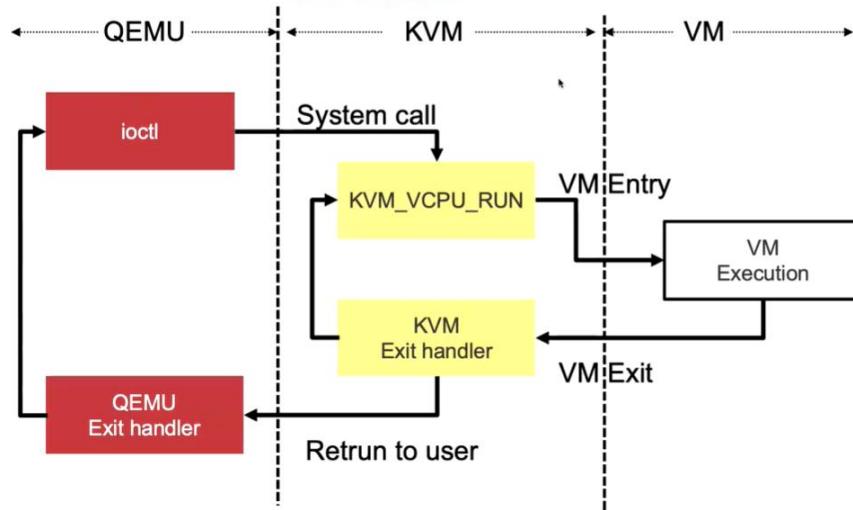


## VT-x和VHE对比

|                         | VT-x          | VHE  |
|-------------------------|---------------|------|
| 新特权级                    | Root和Non-root | EL2  |
| 是否有VMCS？                | 是             | 否    |
| VM Entry/Exit时硬件自动保存状态？ | 是             | 否    |
| 是否引入新的指令？               | 是(多)          | 是(少) |
| 是否引入新的系统寄存器？            | 否             | 是(多) |
| 是否有扩展页表(第二阶段页表)？        | 是             | 是    |

QEMU 分配了一块很大的 memory，相当于加载 VM，然后 ioctl 来运行它。运行过程中会有 VM\_EXIT 到 KVM，如果 KVM 自己可以 emulate，那它就 emulate 返回，否则就会返回到 QEMU。QEMU 处理完之后，再调用 ioctl 进入到 KVM 和 VM。

## QEMU/KVM的流程



# 内存虚拟化

## 三种地址

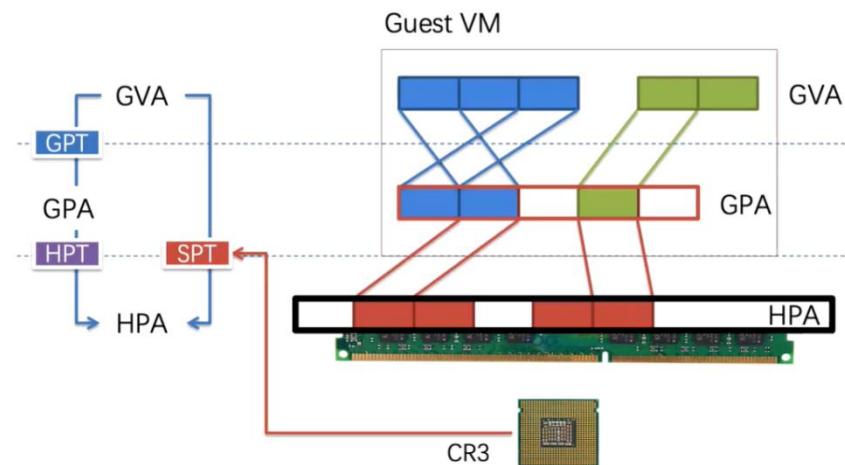
- **客户虚拟地址(Guest Virtual Address, GVA)**
    - 虚拟机内进程使用的虚拟地址
  - **客户物理地址(Guest Physical Address, GPA)**
    - 虚拟机内使用的“假”物理地址
  - **主机物理地址(Host Physical Address, HPA)**
    - 真实寻址的物理地址
    - GPA需要翻译成HPA才能访存
- VMM管理

## 怎么实现内存虚拟化?

- 1、影子页表(Shadow Page Table)
- 2、直接页表(Direct Page Table)
- 3、硬件虚拟化

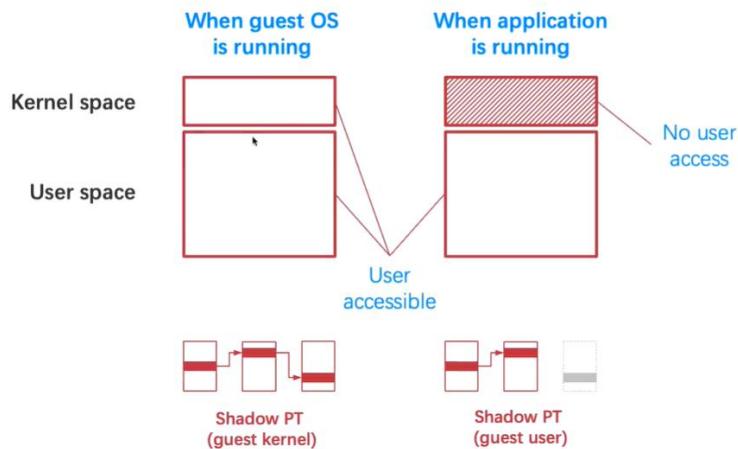
我们的 TTBR 可以直接指向影子页表，也就是把 GPT 和 HPT 二合一生成影子页表，也就是从 GVA 直接映射到 HPA，因为硬件只支持一张表的翻译。

### 1、影子页表



但是导致的问题就是 guest 的用户态和内核态之间没有任何的隔离了。我们需要把一个页表分成两个页表，当用户态运行的时候，它是不能访问内核空间的。

# 一个页表 → 两个页表



直接告诉 VM 你是一个虚拟机，通过 *hypercall* 调用 VMM 提供的内存映射，性能会好一些。

## 2、直接映射 (Para-virtualization)

- **Modify the guest OS**
  - No GPA is needed, just GVA and HPA
  - Guest OS directly manages its HPA space
  - Use *hypercall* to let the VMM update the page table
  - The hardware CR3 will point to guest page table
- **VMM will check all the page table operations**
  - The guest page tables are read-only to the guest

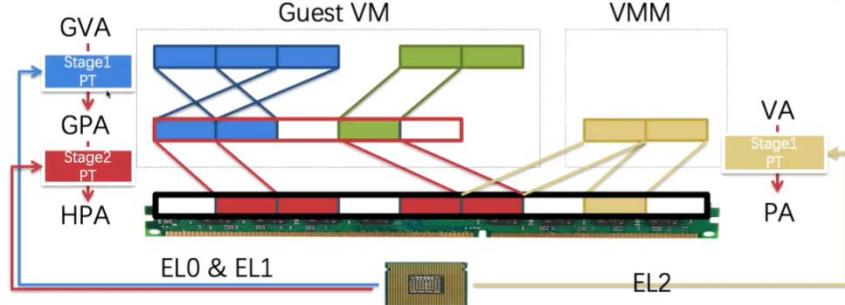
既然我们要做两次翻译，那么硬件就支持两次翻译。

## 3、硬件虚拟化对内存翻译的支持

- **Intel VT-x和ARM硬件虚拟化都有对应的内存虚拟化**
  - Intel Extended Page Table (EPT)
  - ARM Stage-2 Page Table (第二阶段页表)
- **新的页表**
  - 将GPA翻译成HPA
  - 此表被VMM直接控制
  - 每一个VM有一个对应的页表

## 第二阶段页表

- 第一阶段页表：虚拟机内虚拟地址翻译 ( GVA->GPA )
- 第二阶段页表：虚拟机客户物理地址翻译 ( GPA->HPA )



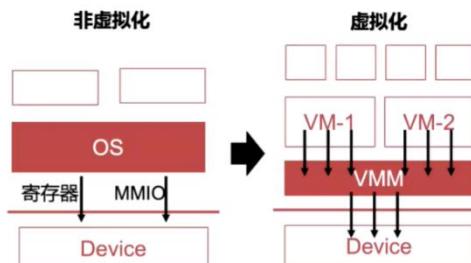
## 怎么实现I/O虚拟化？

- 1、设备模拟 ( Emulation )
- 2、半虚拟化方式 ( Para-virtualization )
- 3、设备直通 ( Pass-through )

设备模拟的方式就是用软件的方式截获每一条 io 指令，并且使用 `syscall` 的方式去实现这些 io 指令。

## 方法1：设备模拟

- OS与设备交互的硬件接口
  - 模拟寄存器(中断等)
  - 捕捉MMIO操作
- 硬件虚拟化的方式
  - 硬件虚拟化捕捉PIO指令
  - MMIO对应内存在第二阶段页表中设置为invalid



半虚拟化的方式就是在 guest os 中放一个前端驱动，两个驱动对接虚拟的设备。

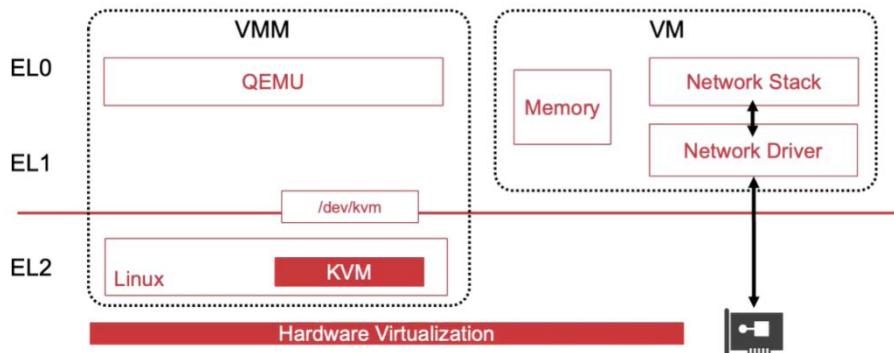
## 方法2：半虚拟化方式

- 协同设计
  - 虚拟机“知道”自己运行在虚拟化环境
  - 虚拟机内运行前端(front-end)驱动
  - VMM内运行后端(back-end)驱动
- VMM主动提供Hypercall给VM
- 通过共享内存传递指令和命令

设备直通就是把设备送给虚拟机，让它100%控制这个设备。但是设备能够访问所有的物理内存，所以我们就需要iommu去限制设备不能随意访问内存。

## 方法3：设备直通

- 虚拟机直接管理物理设备



## I/O虚拟化技术对比

|               | 设备模拟 | 半虚拟化  | 设备直通   |
|---------------|------|-------|--------|
| 性能            | 差    | 中     | 好      |
| 修改虚拟机内核       | 否    | 驱动+修改 | 安装VF驱动 |
| VMM复杂度        | 高    | 中     | 低      |
| Interposition | 有    | 有     | 无      |
| 是否依赖硬件功能      | 否    | 否     | 是      |
| 支持老版本OS       | 是    | 否     | 否      |

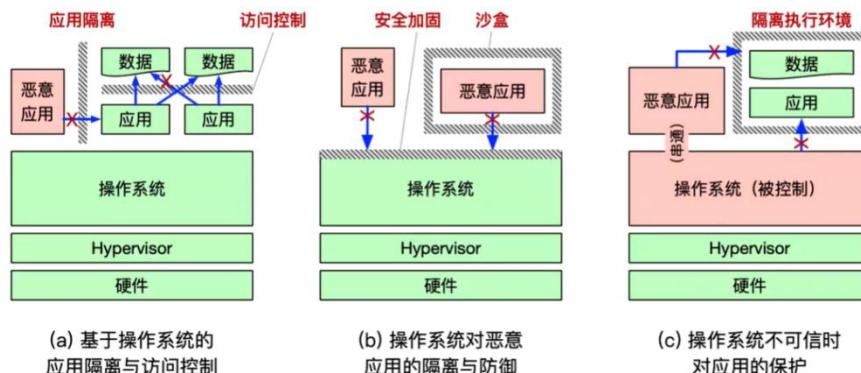
# 虚拟化的技术

| 虚拟化 | 软件方案                                                                                                                                                | 硬件方案                                                                                    |
|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| CPU | <ul style="list-style-type: none"><li>Trap &amp; Emulate</li><li>指令解释执行</li><li>二进制翻译</li><li>Para-virtualization</li></ul>                         | <ul style="list-style-type: none"><li>EL-2 (ARM)</li></ul>                              |
| 内存  | <ul style="list-style-type: none"><li>Shadow page table</li><li>Separating page tables for U/K</li><li>Para-virtualization: Direct paging</li></ul> | <ul style="list-style-type: none"><li>Stage-2 PT (ARM)</li></ul>                        |
| I/O | <ul style="list-style-type: none"><li>Direct I/O</li><li>设备模拟</li><li>Para-virtualization: Front-end &amp; back-end driver (e.g., virtio)</li></ul> | <ul style="list-style-type: none"><li>SMMU (ARM) / IOMMU (x86)</li><li>SR-IOV</li></ul> |

## 操作系统安全

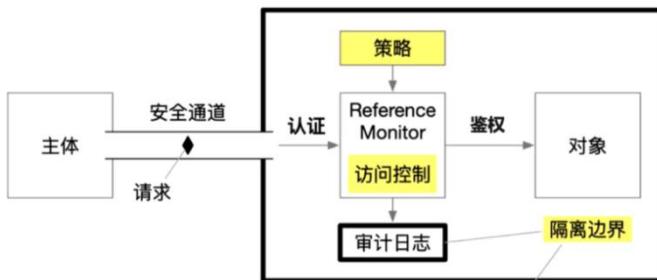
最后一部分是安全。

### 操作系统安全的三个层次



引用监视器模型无非就是去做认证和鉴权。

# 引用监视器 ( Reference Monitor ) 机制



Reference Monitor 负责两件事：

1. **Authentication**：确定发起请求实体的身份，即**认证**
2. **Authorization**：确定实体确实拥有访问资源的权限，包含**授权**和**鉴权**

访问控制分为两大类：ACL（保安拿着名单问你是谁）和 Capability（拿着钥匙开门，不关心你是谁）。Windows 的 ACL 就是每个不同的文件都可以去配置不同的用户，而 POSIX 就是 9 个 bit。RBAC 就是一种泛化。

Capability 是描述一种能力，像令牌一样，谁拿了这个令牌，谁就可以去访问这个设备。

## 访问控制

- **访问控制列表 ( ACL )**
  - 传统的ACL ( 如Windows )
  - POSIX风格 ( 如Linux )
  - RBAC ( Role-Based Access Control )
- **Capability**
  - 类似文件系统的fd
  - 与Linux的capability机制不同

## 侧信道与隐秘信道的关系

- **侧信道与隐秘信道很类似**
  - 两者都使用类似的方式（信道）进行数据的传递
- **侧信道攻击和隐秘信道攻击的不同**
  - 隐秘信道攻击：双方是互相串通的，其目的就是为了将信息从一方传给另一方
  - 侧信道攻击：一方是攻击者，另一方是被攻击者，攻击者窃取被攻击者的数据
    - 即被攻击者无意间通过侧信道泄露了自己的数据

Meltdown 迫使 OS 单独使用一个页表，而不是和应用共享页表。所以大家从 ICS 学起的地址空间布局发生了变化。将来发生下陷的时候，是要切换到 OS 自己的页表的。OS 如果要访问 guest 的数据，必须通过软件方式扫描应用程序的页表，才能找到对应的数据。

## Meltdown漏洞原理

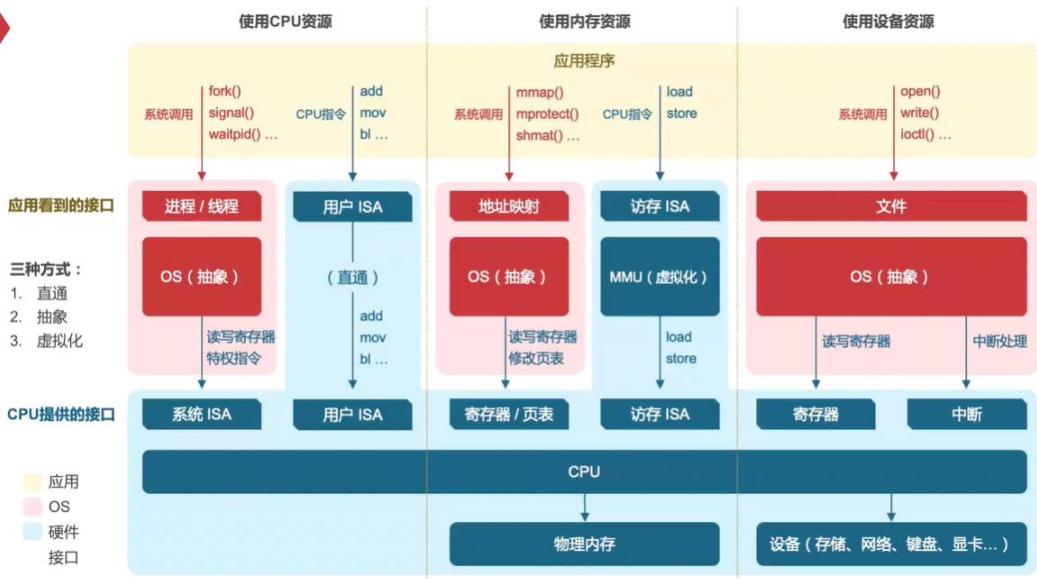
- **起因：迟到的内存访问异常**
  - 在投机执行期间，跨权限的内存访问不会立刻触发异常，而是仍会继续执行后续指令
- **故障：遗漏的缓存状态回滚**
  - 当硬件抛出内存访问异常时，CPU理应回滚被错误执行指令对所有状态的修改
  - **但是，实际上CPU未回滚被错误执行指令对CPU缓存的状态修改**
- **结果：应用可任意读取操作系统内存**
  - 利用缓存隐秘信道，窃取非法访问到的内核数据

OS 一旦沦陷了，我们还有最后一个防御的关卡，就是可信基。就像实际上的人脸识别在 trustzone 中，并不担心我们的手机 OS 有没有被越狱。

## Enclave/TEE：可信执行环境

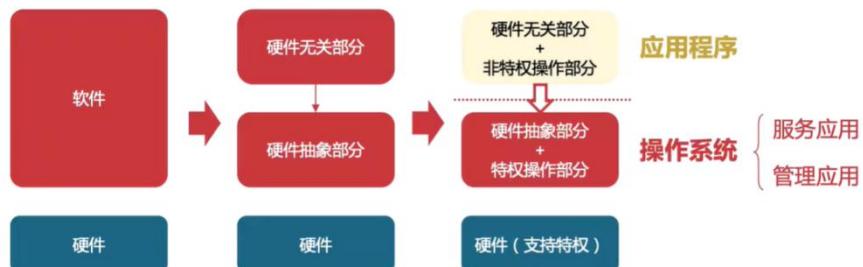
- **Enclave/TEE的定义**
  - Enclave，又称"可信执行环境"（TEE，Trusted Execution Environment），是计算机系统中一块通过底层软硬件构造的安全区域，通过保证加载到该区域的代码和数据的完整性和隐私性，实现对代码执行与数据资产的保护 —— Wikipedia
- **Enclave的两个主要功能**
  - 远程认证：验证远程节点是否为加载了合法代码的Enclave
  - 隔离运行：Enclave外无法访问Enclave内部的数据
- **Enclave带来的能力：限制访问数据的软件**
  - 可保证数据只在提前被认证的合法节点间流动
    - 合法节点：部署了合法软件的节点

总结一下的话就是这张图，



逐步出现了抽象和特权级，就产生了操作系统。

## 操作系统的分化



## 操作系统的八个前沿研究领域

- |                  |                  |
|------------------|------------------|
| <b>1. 异构操作系统</b> | <b>5. 异构硬件</b>   |
| <b>2. 新的应用接口</b> | <b>6. 系统安全</b>   |
| <b>3. 同步原语</b>   | <b>7. 操作系统测试</b> |
| <b>4. 持久性内存</b>  | <b>8. 形式化证明</b>  |

这些都激励着我们要学习更多的 OS 相关的知识点，应用到新的应用场景和硬件平台上，从而为这个时代贡献更多属于我们自己的力量。



## 祝大家期末取得好成绩！

上海交通大学并行与分布式系统研究所 ( IPADS@SJTU )

143

我们今天就到这，祝大家期末取得好成绩。

## 笔者的话

感谢你看到这里，操作系统是我本科阶段最后一门要上课的专业课，或许也是我人生中最后一门要如此用心地整理笔记的课程，我有一些感触有感而发。2022年3月，疫情肆虐上海，刚刚上了三周的所有课程被迫转到线上。三月到六月，笔者和所有同学一样，是在宿舍里度过的。大三下本身就是压力非常大的时候，操作系统、云操作系统、软件测试、无人系统设计、志愿者、校内组里的科研任务、线上暑研的任务、对疫情的担忧、对未知前途的焦虑……，有时候真的不知道自己能不能坚持下来。如今的我蓦然回首，当初的一切努力皆已成通途。感谢老师的倾情授课，感谢同学们和我讨论Lab到每个深夜，感谢所有在我情绪波动的时候愿意听我倾诉的人。

鲍辰

2022/11/13