**<<ARM64硬件结构与系统接口>> ARMv8特性**• 扩大物理寻址– 4GB以外的物理地址• 64位虚拟地址• 自动事件信号– 低功耗、高性能的自旋锁• 硬件加速加密• 新的异常模型 **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\_EL0123• 3个异常链接寄存器(保存异常的返回地址)– ELR\_EL123• 3个程序状态寄存器(切换时保存PSTATE)– SPSR\_EL123 **RISC(Aarch64 )CISC(x86-64)比较** 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:微码,增加通用寄存器数量 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 **特权级/X86-64**• Non-root :– Ring 0：Guest app– Ring 3：Guest os• Root–Ring 0：App–Ring 3Hypervisor**特权级/ARM(Exception Level )**• El0: 用户态程序 • El1: 内核 • El2: hypervisor • El3: monitor **系统状态寄存器**X86-64 • System Flags in the EFLAGS **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 ) **系统控制寄存器(System Control Register )**• 对于系统的顶层控制 – 大小端、使用MMU、检查Tag、内存系统 **内存系统相关寄存器** • Translation Control Register(TCR )• Translation Table Base Register(TTBR )**讨论二• Aarch64的TTBR0支持(0~2^48-1)的地址映射，TTBR1支持(2^48~2^64)的地址映射，这样的硬件设计与x86-64中的CR3相比较，能够如何协助到操作系统的设计？** 内核和用户的页表能共存，不用频繁切换页表，不怕meltdown攻击 **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– 行为与内存不完全一样，读写会有副作用 • 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 )**操作系统启动过程** Bootloader和kernel放在同一个ELF文件中– Bootloader位于.init段，并通过链接器设置入口– Kernel位于.text段• 主CPU启动，其他次CPU等待 >层级切换-其他EL(>=1)到EL1 >准备函数栈和异常向量• 准备C函数栈– 设置SP寄存器• 设置异常向量– 为了debug >初始化UART• 根据UART协议进行内存空间初始化– 映射到IO的内存空间 >初始化页表并开启MMU• >初始化页表并开启MMU– 将kernel代码映射到低地址段(和物理地址相同)和高地址段两份[这里是为了启动方便，之后会把低地址的删掉] >进入kernel• 跳转到kernel的main函数 >Kernel• 真正进入ChCore• 开启OS的各种服务 **X86-64的启动过程**• 为了向前兼容，内核启动中存在兼容性工作• 通过段寄存器，进行模式转换– 实模式(16-bit)– 保护模式(32-bit)– IA32E(64-bit)• 由于实模式的内存限制(64KB)，再将内核代码按段拷贝到内存中• 进入内核 **使用模拟器进行开发**• ARM模拟器– 仅用软件完全实现硬件行为– 如同跑在真正的ARM硬件上 模拟CPU 寄存器 内存 外部设备 **<<中断、异常与系统调用>>** OS获取该输入的可能方法……– 轮询：OS不断去读该缓冲区中的值– 中断：当控制器接收到输入后，打断CPU正常执行，OS进行处理 [频率低用中断，频率高用轮询] **中断异常概念** •中断(Interrupt)– 外部硬件设备所产生的信号– 异步：产生原因和当前执行指令无关，如程序被磁盘读打断• 异常(Exception)– 软件的程序执行而产生的事件– 包括系统调用(System Call)\_用户程序请求操作系统提供服务 – 同步：产生和当前执行或试图执行的指令相关 **AArch64的中断(异步异常)**• 重置(Reset)– 最高级别的异常，用以执行代码初始化CPU核心– 由系统首次上电或控制软件、Watchdog等触发• 中断(Interrupt)– CPU外部的信号触发，打断当前执行– 如计时器中断、键盘中断等 **AArch64的(同步)异常**• 中止(Abort)– 失败的指令获取或数据访问– 如访问不可读的内存地址等• 异常产生指令(Exception generating instructions)– SVC：用户程序-> 操作系统– HVC：客户系统-> 虚拟机管理器– SMC：Normal World -> Secure World **x86-64术语**• 中断(设备产生、异步)– 可屏蔽：设备产生的信号，通过中断控制器与处理器相连，可被暂时屏蔽(如，键盘、网络事件)– 不可屏蔽：一些关键硬件的崩溃(如，内存校验错误)• 异常(软件产生、同步)– 错误(Fault): 如缺页异常(可恢复)、段错误(不可恢复)等– 陷阱(Trap): 无需恢复，如断点(int 3)、系统调用(int 80)– 中止(Abort): 严重的错误，不可恢复(机器检查)**AArch64中断的分类** • 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 **GIC中断来源**•SPI：共享外围中断- 可以被路由到一个或多个核，找到可用的核进行处理- Distributor可配置路由•PPI：私有外围中断- 指定核处理- Distributor可配置路由•SGI：软件产生中断- 核间通信**Linux的中断处理理念**• 在中断处理中做尽量少的事• 推迟非关键行为• 结构：Top half & Bottom half– Top half : 做最少的工作后返回– Bottom half ：推迟处理(softirq, tasklets, 工作队列，内核线程 )**Top Half：马上做**• 最小的、公共行为– 保存寄存器、屏蔽其他中断– 恢复寄存器，返回原来场景• 最重要：调用合适的由硬件驱动提供的中断处理handler• 因为中断被屏蔽，所以不要做太多事情(时间、空间)• 使用将请求放入队列，或者设置标志位将其他处理推迟到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)可以结合在一个向量上• 调用每个设备对应该IRQ的ISR **Bottom Half：延迟完成**• 提供一些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• 这些工作可以被中断 **中断处理没有进程上下文**• 中断(和异常相比)和具体的某条指令无关• 也和中断时正在跑的进程、用户程序无关• 中断处理handler不能睡眠！ **中断处理中的一些约束**• 不能睡眠– 或者调用可能会睡眠的任务• 不能调用schedule()调度• 不能释放信号或调用可能睡眠的操作• 不能和用户地址空间交换数据(可能引发新的异常 )**中断和异常处理必做事项**• 进入中断或异常时– 需保存处理器状态，方便之后恢复执行– 需准备好在高特权级下进行执行的环境– 需选择合适的异常处理器代码进行执行– 需保证用户态和内核态之间的隔离• 处理时– 需获得关于异常的信息，如系统调用参数、错误原因等• 返回时– 需恢复处理器状态，返回低特权级，继续正常执行流 **AArch64的中断和异常处理** EL0下的正常执行流->发生中断或异常->保存处理器状态、错误信息至寄存器(进入EL1)->查询异常向量表(Exception Vector Table)选择handler->处理->eret->返回->继续EL0下的正常执行 **发生– 信息保存**• 异常或中断发生后，硬件会将错误码和部分上下文信息存储在寄存器中– 处理器状态(PSTATE)– 当前指令地址(PC)– 异常发生原因1 )Serror与异常：Exception Syndrome Register(ESR\_EL1)2 )中断：GIC中的寄存器(使用MMIO读取)• 安全性问题– 上述寄存器均不可在用户态(EL0)中访问 **发生– 进入EL1**• 硬件会适当修改处理器状态(PSTATE)，进入EL1执行• 问题：栈内存的安全性– 进入EL1级别后，栈指针(SP)会自动换用SP\_EL1– 从而实现用户栈->内核栈– 如需在EL1下使用SP\_EL0作为栈指针，可配置SPSel寄存器•之前的常规寄存器存进内核栈 **寻找handler的代码**• 使用异常向量表(Exception Vector Table)– 每个异常级别存在独立的异常向量表– 表项为异常向量(Exception Vector)，是处理异常或跳转到异常handler的小段汇编代码– 地址位于VBAR\_EL1寄存器中– 选择表项取决于• 异常类型(同步、IRQ、FIQ、Serror)• 异常发生的特权级• 异常发生时的处理器状态(使用的栈指针/运行状态 )**返回(Exception Return)**• eret 指令– ELR\_EL1 -> PC，恢复PC状态– SPSR\_EL1 -> PSTATE，恢复处理器状态– 降至EL0，硬件自动使用SP\_EL0作为栈指针– 恢复执行 **x86-64的中断和异常处理**• 进入异常– 硬件会将上下文信息和错误码存储在内核栈上• 用异常向量表寻找handler– 不分级– 异常向量表中存handler的地址• iret返回– 恢复程序上下文– 从内核态返回用户态– 继续执行用户程序 **系统调用(Syscall)**• 指运行在用户空间的程序向操作系统内核请求需要更高权限运行的服务• 系统调用提供用户程序与操作系统之间的接口 **Linux追踪系统调用**• 每当有系统调用产生时，Linux可打印发生的系统调用、系统调用的参数和系统调用的返回值• ptrace()可追踪Linux中的系统调用情况– 广泛应用在各种debugger中• 命令行中– strace追踪系统调用– ltrace追踪库函数的调用 **系统调用与安全**• AArch64使用寄存器传参，个数有限– 如ChCore的系统调用支持使用寄存器X0-X7最多8个参数• 若系统调用需要更多参数如何处理？– 使用结构体打包参数，并将结构体的指针作为参数• 问题：内存安全性– 作为参数的指针必须经过检测！– 指向NULL -> kernel crash– 指向内核内存-> 安全漏洞 **用户指针检测**• 完备的指针检测十分耗时– 需要遍历用户进程的所有合法内存区域进行检测• Linux解决方法：非全面检查– Linux仅初步检测用户指针是否属于对应进程的用户内存区域的最大可能边界– 即使通过初步检测，用户指针仍然可能非法(如指向尚未分配的栈空间等)– 直接将非法的指针交给内核使用会导致内核出现页错误，内核态的页错误通常以为着bug，内核会打印异常信息并中止用户进程– Linux采用了一些复杂机制来防止这一情况发生 **处理用户指针问题**• 内核代码仅使用特定代码片段访问用户指针(如copy\_from\_user)– 由访问用户指针而导致内核内存错误的代码段是确定的• 当内核发生页异常(Page Fault)时，内核会检查异常发生的PC– 若异常发生的PC属于访问用户指针的代码段，Linux尝试对其进行修复– 若不属于，则报告问题并终止用户程序• 然而– Linux中很多地方违反了这一规定，导致了许多安全漏洞 **系统调用与性能**• 系统调用会造成大量性能开销• 硬件优化：新的系统调用指令– x86提出了syscall/sysenter/sysexit来代替int进行系统调用• 软件优化– 开放性问题 **<<操作系统结构>>** 降低操作系统复杂性•重要设计原则：策略与机制的分离–策略(Policy)：要做什么——相对静态–机制(Mechanism)：怎么做——相对动态 –操作系统可仅通过调整策略来适应不同应用的需求 **宏内核(MonolithicKernel)**•整个系统分为内核与应用两层–内核：运行在特权级，集中控制所有计算资源–应用：运行在非特权级，受内核管理，使用内核服务 **宏内核的优缺点分析**•宏内核拥有丰富的沉淀和积累–拥有巨大的统一的社区和生态–针对不同场景优化了30年•宏内核的结构性缺陷–安全性与可靠性问题：模块之间没有很强的隔离机制–实时性支持：系统太复杂导致无法做最坏情况时延分析–系统过于庞大而阻碍了创新：Linux代码行数已经过2千万 **宏内核难以满足的场景**•向上向下的扩展–很难去剪裁/扩展一个宏内核系统支持从KB级别到TB级别的场景•硬件异构性–很难长期支持一些定制化的方式去解决一些特定问题•功能安全–一个广泛共识：Linux无法通过汽车安全完整性认证(ASIL-D)•信息安全–单点错误会导致整个系统出错，而现在有数百个安全问题(CVE)•确定性时延–Linux花费10+年合并实时补丁，目前依然不确定是否能支持确定性时延 **微内核的系统架构**•设计原则：最小化内核功能–将操作系统功能移到用户态，称为"服务"(Server)–在用户模块之间，使用消息传递机制通信 **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•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 **混合内核架构**•宏内核与微内核的结合–将需要性能的模块重新放回内核态 **外核架构(Exokernel)**•Exokernel不提供硬件抽象–"只要内核提供抽象，就不能实现性能最大化"–只有应用才知道最适合的抽象(end-to-end原则)•Exokernel不管理资源，只管理应用–负责将计算资源与应用的绑定，以及资源的回收–保证多个应用之间的隔离 **Exokernel+LibOS**•库OS(LibOS)–策略与机制分离：将对硬件的抽象以库的形式提供–高度定制化：不同应用可使用不同的LibOS，或完全自定义–更高性能：LibOS与应用其他代码之间通过函数调用直接交互 **Exokernel架构的设计**•外核的功能–追踪计算资源的拥有权–保证资源的保护–回收对资源的访问权•对应的三个技术–安全绑定(Securebinding)–显式回收(Visiblerevocation)–中止协议(Abortprotocol)•设计原则："将管理与保护分离" **安全绑定**•将LibOS与计算资源绑定–可用性：允许某个LibOS访问某些计算资源(如物理内存)–隔离性：防止这些计算资源被其他LibOS访问 **显式资源回收与中止协议**•Exokernel与应用之间的协议–Exokernel显式告知应用资源的分配情况–应用在租期结束之前主动归还资源•若应用不归还资源，则强制中止–Exokernel拥有对资源的控制权–主动解除资源与应用间的绑定关系 **Unikernel(单内核)**•虚拟化环境下的LibOS–每个虚拟机只使用内核态–内核态中只运行一个应用+LibOS–通过虚拟化层实现不同实例间的隔离•适合容器等新的应用场景–每个容器就是一个虚拟机–每个容器运行定制的LibOS以提高性能 **Exokernel架构的优缺点分析**•优点–OS无抽象，能在理论上提供最优性能–应用对计算有更精确的实时等控制–LibOS在用户态更易调试，调试周期更短•缺点–对计算资源的利用效率主要由应用决定–定制化过多，导致维护难度增加 **Multikernel的设计**•Multikernel的思路–默认的状态是划分而不是共享–维持多份状态的copy而不是共享一份状态–显式的核间通信机制•Multikernel的设计–在每个core上运行一个小内核•包括CPU、GPU等–OS整体是一个分布式系统–应用程序依然运行在OS之上 **<<内存管理>>** **IBM360的内存隔离：ProtectionKey** –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 **虚拟地址**•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 **分段机制**•虚拟地址空间分成若干个不同大小的段–段表存储着分段信息，可供MMU查询–虚拟地址分为：段号+段内地址(偏移)•物理内存也是以段为单位进行分配–虚拟地址空间中相邻的段，对应的物理内存可以不相邻•存在问题–分配的粒度太粗，外部碎片–段与段之间留下碎片空间，降低主存利用率 **分页机制**•更细粒度的内存管理–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–任意虚拟页可以映射到任意物理页–大大缓解分段机制中常见的外部碎片•虚拟地址分为：–虚拟页号+页内偏移•主流CPU均支持分页机制，可替换分段机制 **页表** •页表包含多个页表项，存储虚拟页到物理页的映射 每个进程都有自己独立的页表 **单级页表的问题**•若使用单级页表结构，一个页表有多大？–32位地址空间，页4K，页表项4B，页表大小：2^32 / 4K \* 4 = 4MB–64位地址空间，页4K，页表项8B，页表大小：2^64 / 4K \* 8 = 33,554,432 GB•使用多级页表减少空间占用–若某级页表中的某条目为空，那么对应的下一级页表无需存在–实际应用的虚拟地址空间大部分都未被使用，因此无需分配页表–减少空间的原因：允许页表中出现"空洞" **64位虚拟地址翻译** 16+9+9+9+9+12 前16位都为1或0，用于判断是高地址还是低地址选不同页表 中间4个9是4级索引 最后12位是页内偏移 **页表使能**•CPU启动流程–上电后默认进入物理寻址模式–系统软件配置控制寄存器，使能页表，进入虚拟寻址模式•AARCH64 –SCTLR\_EL1 –第0位(M位)置1，即在EL0和EL1权限级使能页表•x86\_64–CR4，第31位(PG位)置1，使能页表 **AARCH64页表项**•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–Upperattributes包括：\_第54位(XN位)为1表示EL0不能执行(eXecutionNever)\_第53位(PXN位)为1表示EL1不能执行\_第51位(DBM位)，类似于x86\_64中的dirtybit –Lowerattributes包括：\_第7位-第6位表示读写权限位AP[2:1] \_第10位(AF位)是AccessFlag，若设为0则访问时发生异常[可供软件追踪内存访问情况]\_第9位-第8位是Shareabilityfield(用于核间、核与设备间的共享)\_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(设为non-cacheable，设备内存，又再细分四种)**ARM的CacheLockdown特性**•Cache lockdown寄存器–可以配置部分Cache不被evict，使数据一直驻留在CPU内部–非统一标准，取决于具体的实现(也可不实现)•Cache lockdown用途–提高性能：可保证访问部分重要数据永远cachehit•硬件的Cache替换策略不够完美–提高安全性：限制部分数据永远不离开CPU•若数据量大于Cache容量，可加密后离开CPU **TLB刷新(TLBFlush)**•TLB使用虚拟地址索引–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•x86\_64上只有唯一的基地址寄存器–内核映射到应用页表的高地址–避免系统调用时TLB刷新的开销•刷TLB相关指令–清空全部–清空指定ASID相关–清空指定虚拟地址 **如何降低TLB刷新的开销**•为不同的页表打上标签–TLB缓存项都具有页表标签，切换页表不再需要刷新TLB•x86\_64：PCID(ProcessContextID)–PCID存储在CR3的低位中–在KPTI使用后变得尤为重要•KernelPageTableIsolation•即内核与应用不共享页表，防御Meltdown攻击•AARCH64：ASID(AddressSpaceID)–OS为不同进程分配8/16ASID，将ASID填写在TTBR0\_EL1的高8/16位–ASID位数由TCR\_EL1的第36位(AS位)决定 **TLB与多核**•需要刷新其它核的TLB吗？–一个进程可能在多个核上运行•如何知道需要刷新哪些核？–操作系统知道进程调度信息•怎么刷新其它核？–x86\_64: 发送IPI中断某个核，通知它主动刷新–AARCH64: 可进行全局TLB刷新 **换页机制(Swapping)**•换页的基本思想–将物理内存里面存不下的内容放到磁盘上–虚拟内存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–在处理缺页异常时，触发物理内存页的换入换出 **缺页异常(PageFault)**•缺页异常–CPU控制流传递–提前注册缺页异常处理函数•x86\_64–异常号#PF(13)，错误地址在CR2•AARCH64–触发(通用的)同步异常(8)，–根据ESR信息判断是否缺页，错误地址在FAR\_EL1 **TLB设计通常也采用分级结构** 分成L1和L2，L1中分数据和指令 **全局TLB**•再看(第3级)页表项–Lowerattributes的第11位是nG(notGlobal)位•nG==0:相应TLB缓存项对所有进程有效•nG==1:仅对特定进程(ASID)有效[作用：如OSLib等，很多内存是跨进程共享的] **按需分配中的权衡**•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–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 **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 **工作集模型(WorkingSetModel)**•一个进程在时间t的工作集W(t, x )(Peter Denning)：–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，则需要讲该集合保持在内存中•工作集模型：–all-or-nothing模型–进程工作集要不都在内存中，否则全都换出–避免thrashing，提高系统整体性能表现 **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–访问位为1则说明在此次tick中被访问，记录上次使用时间为当前时间–访问位为0(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–将所有访问位清0•注意访问位(accessbit)需要硬件支持 **物理内存管理中的碎片问题**•外部碎片(空闲的但不连续，无法被使用)•内部碎片(分配大小大于实际需要 )**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–一个是0，另一个是1–块的大小决定是哪一位[省去了meta data] **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bestfit定位资源池 **SLUB数据结构** 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 分配使用currentslab-若满发生两个移动释放到对应的slab-移动full到partial-若partial全free则还给伙伴系统 **物理内存管理的其它问题**•安全问题–Rowhammer–CacheSideChannel•性能问题–性能隔离、QoS等 **性能考虑**•cachemiss的开销很高–物理内存分配同样需要考虑到降低cache冲突几率–典型机制：cachecoloring•保证性能隔离是QoS的必要前提–Intel CAT–AARCH64 MPAM **共享内存**•基本功能–节约内存，如共享库–进程通信，传递数据 **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•**memorydeduplication–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：LinuxKSM–kernel same-page merging •导致新的sidechannel–访问被合并的页会导致访问延迟明显•潜在攻击–攻击者可以确认目标进程中含有构造数据 [解决：关键数据不合并] **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•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 **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 **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 **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 )**Linux上有趣的内存管理API**•madvise–intmadvise(void \*addr, size\_tlength, intadvice)•使用–用户态的语义信息告诉内核，便于优化–例如：将madvise和mmap搭配起来使用，在使用数据前告诉内核这一段数据需要使用，从而减少缺页异常 **<<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 **进程至少应当拥有以下五种状态：**–新生状态(new )：进程刚被创建–运行状态(running )：进程正在处理器上运行–准备状态(ready )：进程可以运行，但没有被调度–阻塞状态(blocked )：进程进入等待状态，短时间不再运行–终结状态(terminated )：进程完成了执行 **进程的相关数据结构：Process Control Block** •存放进程相关的各种信息–进程的标识符、内存、打开的文件–进程在切换时的状态(上下文 context )**进程的上下文切换**•进程通过中断或系统调用进入内核•上下文保存在对应的 PCB 中–被调度时，从 PCB 中取出上下文并恢复 **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 )**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 **进程的执行：**exec •为进程指定可执行文件和参数*int execve(const char \*pathname, char \* const argv [], char \*const envp)*•在 fork 之后调用–exec 在载入可执行文件后会重置地址空间 **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 的情况里，减少了无用的拷贝上 **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() )**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 •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 **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** •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 **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、代码库 **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 **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 **一对一模型** •每个用户线程映射单独的内核线程–优点：解决了多对一模型中的可扩展性问题–缺点：内核线程数量大，开销大•主流操作系统都采用一对一模型 **多对多模型(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 **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)•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 **线程的基本操作：以pthreads 为例** •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” •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**线程的上下文切换：以ChCore 为例** •线程的上下文即重要的寄存器信息–常规寄存器： x0 x30–程序计数器(PC elr\_el1–栈指针： sp\_el0–CPU 状态(如条件码)： spsr\_el1•主要分为三步–进入内核态，保存上下文–切换页表与内核栈–恢复上下文，返回用户态 ChCore的 TCB 结构 •上半部分：线程的相关信息-线程优先级、状态等 •下半部分：线程上下文•TCB 下面为线程的内核栈–刚进入内核时的线程内核栈为空–sp\_el1 指向栈顶 **第一步：进入内核态、保存上下文** •应用线程可通过异常、中断或系统调用进入内核态–运行状态将切换到内核态(EL1–开始使用 sp\_el1 作为栈指针(用户栈切换到内核栈)–保存应用线程的 PC elr\_el1–保存应用线程的 CPU 状态(spsr\_el1–以上均由硬件自动完成 •保存上下文 **第二步：切换页表和内核栈** •操作系统确定下一个被调度的线程(调度器决定)•切换页表–将页表相关寄存器的值置为目标线程的页表基地址•切换内核栈–找到目标内核栈的栈顶指针(目标线程的 TCB–修改 sp\_el1 的值至目标内核栈–可以认为是线程执行的分界点(切换之后变为目标线程执行 )**第三步：上下文恢复，返回用户态** •上下文恢复：取出栈上的值并存回寄存器 •返回用户态：调用 eret ，由硬件执行一系列操作–将 elr\_el1 中的返回地址存回 PC–改为使用 sp\_el0 作为栈指针(内核栈切换到用户栈)–将 CPU 状态设为 spsr\_el1 中的值–运行状态切换为用户态(EL0 )**一对一线程模型的局限** •复杂应用：对调度存在更多需求–生产者消费者模型：生产者完成后，消费者最好马上被调度–内核调度器的信息不足，无法完成及时调度••“短命”线程：执行时间亚毫秒级(如处理 web 请求)–内核线程初始化时间较长，造成执行开销–线程上下文切换频繁，开销较大 **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 **Linux 对于纤程的支持： ucontext** •每个 ucontext 可以看作一个用户态线程–makecontext: 创建新的 ucontext–setcontext: 纤程上下文切换–getcontext: 保存当前的 ucontext **Windows对于纤程的支持： Fiber 库** •与 ucontext 类似的编程模型–createFiber: 创建新的纤程–SwitchToFiber: 纤程切换•支持纤程本地存储(FLS)–当一个内核线程对应单个纤程时， FLS 与 TLS 结构相同–当一个内核线程对应多个纤程时， TLS 可分裂为多个 FLS **<<进程/线程调度>> 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 **Linux中的调度策略** • 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 **经典调度** • FIFO**-**先到先得：简单、直观-问题：平均周转、响应时间过长 •短任务优先：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 •轮询：公平、平均响应时间-问题：牺牲周转时间 [如果大家时间都差不多，RR效率很差] **优先级调度** •多级队列：1)维护多个优先级队列2)高优先级的任务优先执行3)同优先级内使用Round Robin调度 **什么样的任务应该有高优先级？**– I/O绑定的任务• 资源利用率– 用户设置的重要任务– 时延要求极高(必须在短时间内完成)的任务– 等待时间过长的任务\_公平性 **优先级反转与继承** 占据了资源的低优先级任务可能会被高优先级抢占后调度不回自己，可以通过优先级继承解决[暂时把优先级让给它] **任务权重**：可能不是每个人都应该享受相同的待遇 **一种公平共享的实现：Lottery Scheduling** • 每次调度时，生成随机数𝑹 ∈ [𝟎, 𝑻] • 根据R，找到对应的任务 **Ticket Transfe**r 把优先级借给帮我忙的任务 **思考：权重与优先的异同?** • 权重影响任务对CPU的占用比例– 永远不会有任务饿死• 优先级影响任务对CPU的使用顺序– 可能产生饿死 **Stride Scheduling** • 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– 𝑠𝑡𝑟𝑖𝑑𝑒 = MaxStride/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 **<<进程间通信>>** **共享内存** •系统内核为两个进程映射共同的内存区域–快递员和小明的快递桌•挑战 做好同步–发送者不能覆盖掉未读取的数据 –接收者不能读取别的数据 **共享内存的问题** •轮询导致资源浪费•固定一个检查时间，时延长 **直接通信** •通过id直接链接–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 **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 **信箱共享的挑战** •信箱的共享–进程P1、P2和P3共享一个信箱 M–P 1 负责发送消息， P 2 、 P 3 负责接收消息–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–让一个连接 信箱 只能被最多两个进程共享，避免该问题–同一时间，只允许最多一个进程在执行接收信息的操作–让消息系统任意选择一个接收者 需要通知发送者谁是最终接收者 **消息传递的同步与异步** • 消息的传递可以是阻塞的，也可以是非阻塞的• 阻塞通常被认为是同步通信– 阻塞的发送/接收: 发送者/接收者一直处于阻塞状态，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• 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 •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 **通信连接的缓冲** •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–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–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 快递可以放在门口任何位置 **Unix管道** •管道是 Unix 等宏内核系统中非常重要的进程间通信机制•管道(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 • 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 **扩展Sleep/Wakeup 通信机制** •信道(是等待和通知的媒介)•一个进程可以通过 sleep 接口将自己等待在一个信道上•另外一个进程可以通过 wakeup 将等待在某个信道上的进程唤醒 **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–只能支持最多两个进程间通信 **消息队列: 带类型的消息传递**• 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 (type message)–类型为 0 时返回第一个消息(–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 **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 **轻量级远程调用(LRPC )**•解决两个主要问题–控制流转换 调用者进程快速通知被调用者进程–数据传输 将栈和寄存器参数传递给被调用者进程 **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–调用者和被调用者之间可能会执行多个不相关进程 **迁移线程:将调用者运行在被调用上下文** •为什么需要做控制流转换–使用被调用者的代码和数据–使用被调用者的权限 如访问某些系统资源•只切换地址空间、权限表等状态，不做调度和真正的线程切换 **数据传输：万恶之源的数据拷贝**•大部分 Unix 类系统，经过内核的传输有 至少 两次拷贝–调用者 内核 被调用者•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 **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 被同时映射在调用者进程和被调用者进程地址空间–调用者进程只需要将参数准备到 A stack 即可•不需要内核额外拷贝•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 **轻量远程调用:通信连接建立** •被调用者进程通过内核注册一个服务描述符–对应被调用者进程内部的一个处理函数(•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)–调用记录被用作从被调用者进程处返回•内核将参数栈交给客户端进程，作为一个绑定成功的标志–在通信过程中，通过检查 A stack 来判断调用者是否正确发起通信 **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.将调用者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.找到被调用者进程的 运行栈 执行代码所使用的栈7.将当前线程的栈指针设置为被调用者进程的运行栈地址8.将地址空间切换到被调用者进程中9.执行被调用者地址空间中的处理函数 **CHCORE的IPC 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 **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 **<<同步原语>>** **竞争条件Race Condition** • 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 **解决临界区问题的三个要求** • 互斥访问：在同一时刻，有且仅有一个进程可以进入临界区。(一个快递员放快递)• 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。• 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。 **软件解决方案：皮特森算法** while轮询检查flag，执行完后设置flag -更近一步，执行临界区前后关闭中断[能解决单核问题，但可能多核同时进入临界区] **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 **Intel锁总线实现** 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\* **ARM使用LL/SC实现** Load-linked & Store-conditional 第二行读的时候监视addr第四行修改的时候看addr是否被其他人修改没人修改就写成功，否则回到第二行 **硬件辅助** 比较并替换(Compare And Swap)获取并增加(Fetch And Add)**自旋锁** while(atomic\_CAS(lock, 0, 1 )!= 0 )• 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 **排号锁(Ticket Lock )**1. 拿号my\_ticket =atmoic\_FAA(&next, 1 )2.等待叫号while(owner !=my\_ticket); 3.叫下一个 owner+=1 [owner：表示拿锁的人next：表示目前放号的最新值] • 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\* ✓ **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 •偏向性：写者等待时，读者能入则偏向读，否则偏向写 **Read Copy Update, RCU** •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**。** 方案：用链表插入删除修改节点 问题：什么时候删除旧拷贝？ •需求2：在合适的时间，回收无用的旧拷贝 方案：宽限，直到最后一个相关的读者离开 **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 **检测死锁与恢复** • 直接kill所有循环中的进程• 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** 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 **死锁避免：运行时检查是否会出现死锁** 银行家算法• 所有进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：阻塞进程，下次再给§ 如果不会造成：给进程该资源 **优先级反转思考**：为什么会出现优先级反转？操作系统：基于优先级调度 锁：按照锁使用的策略进行调度 双重调度导致 如何解决？打通两重调度，给另一个调度实例方法 • 不可打断临界区协议(Non-preemptive Critical Sections, NCP )-进入临界区后不允许其他进程打断：禁止操作系统调度• 优先级继承协议(Priority Inheritance Protocol,PIP )-高优先级进程被阻塞时，继承给锁持有者自己的优先级：锁给操作系统调度hint • 即时优先级置顶协议(Immediate Priority Ceiling Protocols, IPCP )-获取锁时，给持有者该锁竞争者中最高优先级：锁给操作系统调度hint • 原生优先级置顶协议(Original Priority Ceiling Protocols, OPCP)-高优先级进程被阻塞时，给锁持有者该锁竞争者中最高优先级：锁给操作系统调度hint **<<多核多处理器与性能可扩展性>>** **理想并行计算加速比S** = 1 / [(1-p)+p/s] [p是可并行代码比，s是核心数] **可扩展性断崖** 核心数堆到一定程度性能跳楼 **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 **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 [怎么做？• 缓存行处于不同状态(MSI状态)• 不同状态之间迁移• 所有地读/写缓存行操作遵循协议流程 **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 • 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 • 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 **缓存一致性：全局目录项** 全局目录项：记录缓存行在不同核上的状态，通过总线通讯 **传统spin锁的问题** 多核访问同一个元素。对单一缓存行的竞争导致严重的性能开销 **Simple fix：避免对单一缓存行的高度竞争– Back-off 策略** 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 [治标不治本] **MCS锁** 等锁的人用链表串起来，还锁时通过指针修改下一个等待者的元素 **Linux Kernel中的可扩展锁：QSpinlock** 竞争程度低：•快速路径-使用类似自旋锁设计-加锁/放锁流程简单 •竞争程度高：慢速路径-使用类似MCS锁设计-可扩展性好 **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 -先获取每结点本地锁 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 **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 **多核的执行顺序** 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 **内存模型：严格一致性模型** • 严格一致性模型(Strict Consistency)对一个地址的任意的读操作都能读到这个地址最近一次写的数据 访存操作顺序与全局时钟的顺序完全一致 **内存模型：顺序一致性模型** • 顺序一致性模型(Sequential Consistency)不要求操作按照真实发生的时间顺序(全局时钟)全局可见 执行结果必须与一个全局的顺序执行一致 且这个全局顺序中一个核心的读写操作与其程序顺序保持一致。 **内存模型：TSO一致性模型** • TSO 一致性模型(Total Store Ordering)针对不同地址的读-读、读-写、写-写顺序都能得到保证 只有写-读的顺序不能够得到保证 **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 **在弱的内存模型中保证顺序** 通常的做法：添加硬件内存屏障(barrier/fence )**系统软件开发者视角下的内存模型** • 内存模型不是透明的• 软件需要手动根据运行架构保证访存操作顺序• 同步原语(互斥锁、信号量等)拥有保证访存顺序的语义• 系统需要使用保序手段(如barrier)提供正确的同步原语，保证软件正确性• 硬件内存屏障(如barrier)开销很大，需要合适的地方用合适的方法保证顺序• 正常情况下，软件需要使用同步原语来同步 **<<文件系统结构>>** **Ext2存储布局** 超级块记录了整个文件系统的元数据-每个块组中都有超级块，互为备份 一个块组:[超级块|快组描述表|块bitmap|inode bitmap|inode表|数据块] **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 • 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 **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 **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 **硬连接** • 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 **常见的其他文件类型** • 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 **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 **ChCore的常规文件** • 基数树结构– 提供<键,值>索引– 如前述inode中的多级间接指针– 如页表结构• ChCore中的基数树– 保存<逻辑块号，块指针> **ChCore的目录文件** • 哈希表– 保存目录项– 使用链表解决冲突 **文件的查找** • 从根目录开始，在各级目录中逐层找下一级(别忘检查软连接 )**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 **文件内存映射：用mmap()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3. 缺页中断处理函数，通过虚拟地址，找到该文件的inode4. 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 **FAT：文件分配表** [引导 | FAT1 | FAT2(备份 )|数据区] FAT为每个数据簇增加了一个next指针，让簇可以串联在一起 **FAT32的目录项** 每个目录记录32个字节，作为目录文件的数据保存在数据簇中 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 -内容:文件名 属性 簇号(高/低位 )文件大小 **NTFS存储布局** [引导|数据区 |MFT |数据区 |MFT(镜像)| 数据区 ] **NTFS主文件表MFT** [MFT|MFT(镜像)|日志文件|. . . .|文件记录|目录记录|文件记录|文件记录] **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– 包含文件名、文件ID(在MFT中的序号 )**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 •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 •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 **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 **高级功能: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 **快照** • 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** • 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 • 在索引中增加标记• 删除全0块 **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)• 文件系统之上的文件系统 **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 **FUSE用户态文件系统框架** • 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 **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 [fuse驱动向上支持vfs和fuse库，起到连接作用] **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 **FUSE能用来做什么？**• 出Lab！• SSHFS(用ssh挂载远端目录到本地)• NTFS-3G• 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? **<<崩溃一致性>>** **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2. 仅有最近的一些操作没有被保存到磁盘中3. 没有顺序的异常 **要求的属性** 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 **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 **利用内存中的页缓存** 在内存中记录日志，异步写入到磁盘中 仅需保证日志提交在磁盘数据修改之前 利用内存中的页缓存 **批量处理日志减少磁盘写** 多个文件操作的日志合并在一起 每个修改过的块只需记录一次 **日志提交** • 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 JBD2事务的状态 JBD2事务的状态 1运行 新的原子操作可以加入到事务中 2锁定 事务不接受新的原子操作，但现有原子操作还不一定全部完成 3写入 事务正在被写入存储设备 4提交 事务内容已经写入存储设备，正在写入提交块 5完成 事务写入完毕 **JBD2日志的磁盘结构** [日志超级块|描述块|日志块|日志块|. . . . |提交块 ] **Ext4用JBD2实现的三种日志模式** 1Writeback Mode：日志只记录元数据-最快，但是一致性最差！2Ordered 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2Journal 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！ **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 **Soft Updates的总体思想** • 最新的元数据在内存中• 磁盘中的元数据总是一致的 **Soft Updates的三个次序规则**1. 不要指向一个未初始化的结构2. 一个结构被指针指向时，不要重用该结构3. 不要修改最后一个指向有用结构的指针 **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 )**Sprite LFS的数据结构** • 固定位置的结构– 超级块、检查点(checkpoint)区域• 以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 **空间回收管理方法：段(Segment )**• 一个设备被拆分为定长的区域，称为段– 段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 )**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 **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 **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 **挂载和恢复**• 扫描所有日志，重建出整个文件系统的内存结构– 大量无效数据也被扫描• 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 **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 **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 • 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• 解决方案：目录修改日志 **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 **<<新型文件系统>>** **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 **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 • 磨损均衡– 频繁写入同一个块会造成写穿问题– 将写入操作均匀的分摊在整个设备• 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-levelling)等 **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 **F2FS的改进2：**多log并行写入 **闪存友好的磁盘布局** • 组织层级– Block：4KB，最小的读写单位– Segment：2MB– Section：多个segment(垃圾回收/GC粒度)– Zone：多个section • 系统元数据(随机写入)– 存放在一起：局部性更好– CP：检查点– SIT：段信息表– NAT：node地址转换表– SSA：段概要区域• 数据区(多Log顺序写入)– 区分冷/温/热数据– 区分文件数据(data segment)与元数据(node segment )**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 [热node 直接数据块 温node常规文件间接数据块 冷node间接数据块 热data 存目录项的数据块 温data常规文件数据块 冷data被清理过程移动 用户指定 多媒体数据] **清理(Cleaning)**• 以section为粒度– 与硬件FTL的GC单位是对齐的• 过程(与LFS类似)1. 选择需要清理的section• Greedy：选择有效块最少的section• Cost-effective：同时考虑数据修改时间2. 识别有效数据3. 有效数据拷贝到干净section4. 标记被清理的section为pre-free• 在下一次checkpoint之后被标记为free **自适应日志** • 文件系统使用一段时间后，干净section不足，需要频繁清理• F2FS动态调整数据段的日志方法– 干净section充足时，使用常规方法• 日志写到干净section中• 没有干净section时需要进行清理操作– 干净section不足时，使用threaded logging• 使用脏段中无效的块• 避免清理操作• 但会产生一些随机写 **检查点**• 检查点、NAT、SIT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 **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node2. 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，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SIT5. 创建新的检查点 **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 **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 **Band内随机写怎么办？**方法一：多次拷贝• 修改Band X中的4KB数据1. 找到空闲Band Y2. 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 *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• 修改Band X中的4KB数据1. 将修改写入缓存，标记Band X为dirty2. 修改STL映射(让原位置指向持久化缓存)3. 空闲时，根据缓存内容，将dirty Band进行清理• 4KB随机写→ 修改4KB缓存 **观察：持久缓存对吞吐量的影响** 随机写的跨度→ 脏band数量→ 清理时的工作量→ 吞吐量 **Ex4元数据随机写回太多吞吐量低？** • 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 **日志满了怎么清理** • 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– 对于冷元数据，可将其写回到Ext4中其原本的位置S– 热元数据继续保留在日志中 **非易失性内存写入顺序** 由于cache writeback的存在，写入顺序是乱的 **Intel x86 拓展指令集** • CLFLUSHOPT– 可并行执行的CLFLUSH– 需要sfence保证顺序• CLWB– CLFLUSHOPT的基础上，不会将cacheline无效化 **非易失性内存改变存储栈** 不需要通用块层 IO调度器和驱动，很快 **PMFS中的一致性保证** • 现有方法– 写时复制(Shadow Paging)[用于文件数据]– 日志[用于元数据]– Log-structured updates• NVM专有的方法– 原子指令更新[适用于小数据更新] **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 **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，CR0.WP临时设置为0，内核可以修改只读映射 **<<设备管理>>** **设备抽象(以Linux为例)**• 对设备进行分类– 字符设备(char)： LED、键盘、串口– 块设备(block)： 闪存– 网络设备(network)： Ethernet网卡• 对设备进行管理– 字符抽象：文件系统(read/write )– 块抽象：文件系统(read/write )，mmap– 网络抽象：socket，文件系统兼容(用read/write读写socket)**字符设备** • 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• 通常使用文件抽象：– open(), read(), write(), close( )**块设备**• 访问模式：– 随机访问，以块粒度进行读写– 在系统层增加一层缓冲，避免和慢设备频繁交互• 通常使用内存抽象：– 内存映射文件(Memory-Mapped File)：直接访问数据– 同样可以使用文件抽象，但内存抽象更受欢迎(灵活性更好)**网络设备** • 访问模式：– 面向格式化报文的收发– 在驱动层以上维护多种协议，支持不同策略• 通常使用套接字抽象：– socket(), send(), recv(), close(), etc **可编程I/O**• 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)• MMIO(Memory-mapped IO)– 将设备映射到连续物理内存中– 使用内存访问指令– 行为与内存不完全一样，读写会有副作用 DMA 1. 驱动发出请求：将外设数据读至内存地址X 2. 外设初始化DMA传递3. 外设传送数据到DMA控制器4. DMA将数据传至内存地址X 5. DMA控制器完成传输后，对CPU发出中断，通知传输完成6. CPU收到中断：调用驱动中断处理函数 **GIC** • Generic Interrupt Controller• Distributor– 负责全局中断的分发和管理• 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– 中断的目的core– 中断触发方式– 中断状态管理 **CPU Interface**• CPU接口：– 将GICD发送的中断信息，通过IRQ、FIQ管脚，发送给连接到interface的core• 寄存器：GICC• 作用：– 将中断请求发送给CPU– 中断确认(acknowledging an interrupt)– 中断完成(indicating completion of an interrupt)– 设置中断优先级屏蔽– 定义中断抢占策略 **中断确认**• CPU开始响应中断：– IRQ状态：pendingàactive• 寄存器：GICC\_IAR，记录当前等待处理的中断号• 通过访问GICC\_IAR寄存器，来对中断进行确认 **中断完成**• CPU处理完中断：– IRQ状态：activeàinactive• 优先级重置(priority drop)：– 将当前中断屏蔽的最高优先级进行重置，以便能够响应低优先级中断– 寄存器：GICC\_EOIR• 中断无效(interrupt deactivation)：– 将中断的状态置为inactive状态– 寄存器： GICC\_DIR **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 Interface③ Deliver：CPU Interface将中断传给core④ 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 **如何设计中断处理函数(ISR)？**• 中断应该尽快响应– 提高系统对外部的实时响应能力• 尽量短– Linux上半部：马上处理• 可重入– 应允许在中断过程的任意时刻被抢占 **EOI(End of Interrupt)**• 告诉中断控制器，本次IRQ已经处理完毕• 如果不显示执行EOI，中断控制器会假设中断没有响应，并屏蔽后续的中断请求 **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：只对对应IRQ停止响应• 什么策略合适？– 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 **如果发端拼命发数据，收端首先收到什么数据？**– 收端来不及处理数据，导致之前的数据的被覆盖了？– 收端在处理中断上下文中，导致后来数据没有收到？ **对策**• 静态配置法– UART驱动需要在初始化时指定“波特率”• 动态协商法– TCP设计了流量控制机制，发端逐步试探出收端收包能力的上限 **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 )**为什么需要驱动模型**• 设备的整体趋势：– 数量和规模越来越大– 更新速度越来越快：驱动代码量在快速增长• 驱动开发者的要求：– 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 **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• 维护驱动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– LDDM启动设备会将依赖对象自动初始化，直到启动条件满足为止 **Linux设备驱动抽象**• Device(设备)：用于抽象系统中所有的硬件– 包括CPU和内存• Bus(总线)：CPU连接Device的通道– 所有的Device都通过bus相连• Class(分类)：具有相似功能或属性的设备集合– 类似面向对象程序设计中的Class– 抽象出一套可以在多个设备之间共享的数据结构和接口– 从属于相同Class的设备驱动程序，直接继承 **LDDM特点**• device\_driver和device都注册到bus上• bus\_type的match()– 如果设备与驱动相匹配，将调用device\_driver的probe()，交给device\_driver来完成余下工作• device\_driver的probe()– 驱动程序的入口，probe成功后内核生成设备实例，驱动注册的file\_operations可以被应用程序所访问• device为bus\_type 、device\_driver父类的多继承– 要实例化一个device，先实例化父类driver和bus **Softirq特点**• 执行时间点：– 中断之后(上半部之后)– 系统调用或是异常发生之后– 调度器显式执行ksoftirqd• 并发：– 可以在多核上同时执行– 必须是可重入的– 或根据需要加锁• 可中断：Softirq运行时可再被中断抢占 **Softirq Rescheduling** • 软中断要求能被重调度– 在处理软中断A时，能切换至软中断B(挂起A唤醒B)• 问题：中断产生中断，可能活锁 • 方案：配额(quota)+ ksoftirqd– Softirq调度器每次只运行有限数量的请求– 剩余请求有内核线程ksoftirqd代为执行，和用户进程抢CPU– ksoftirqd和用户进程都被调度器调度 **Tasklet**• 问题：Softirq是静态的• 方案：引入Tasklet– 基于Softirqs，但可以被动态创建和销毁！– 同一时期，同种类型的Tasklet一次只能运行在一个– 不同类型的Tasklet可以同时运行在不同CPU上• 缺点：– 和Softirq一样缺乏进程上下文，无法睡眠！ **Tasklet的优势**• 可动态分配，数量不限• 直接运行在调度它的CPU上(缓存亲和性)• 执行期间不能被其它下半部抢占– 不存在重入的问题– 无需加锁• 曾是最受原因的延迟处理机制 **Tasklet的问题**• 难以正确实现– 要防止休眠代码• 任务不可抢占性(仍可被中断)– 比其他任务的优先级都高，影响任务实时性– 导致不可控的延迟**工作队列(Work Queues)**• Softirq和Tasklet使用中断上下文• 工作队列使用进程上下文– 可以睡眠！• 方式：– 在内核空间维护FIFO队列， workqueue内核进程不断轮询队列– 中断负责enqueue(fn, args)， workqueue负责dequeue并执行fn(args)• 特点：– 只在内核空间，不和任何用户进程关联，没有跨模式切换和数据拷贝 **内核线程(Kernel Threads)**• 始终运行在内核态– 和工作队列一样，没有用户空间上下文• 中断线程化\*(threaded interrupt handlers)– Linux 2.6.30引入– 想要取代Tasklet和Workqueue– 每个中断线程都有自己的上下文 **I/O请求的阻塞与非阻塞** • 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– 用户提供缓存区，内核等成功操作完数据后来再通知用户，用户执行并不停滞(类似中断请求之于CPU)**<<系统虚拟化>>** **虚拟化带来的优势**•服务器整合：提高资源利用率•方便程序开发•简化服务器管理 **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–系统ISA•只有内核态程序可以使用 **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 **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 **Type-1虚拟机监控器**•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 **Type-2虚拟机监控器**•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 **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 **系统ISA**•读写敏感寄存器•控制处理器行为•控制虚拟/物理内存•控制外设 **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–DMA、中断 **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 **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 **非可虚拟化架构：Non-virtualizable** •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•在ARM中：不是所有敏感指令都属于特权指令 **如何处理这些不会下陷的敏感指令？**•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 )**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 **二进制翻译的缺点** •不能处理自修改的代码(Self-modifyingCode)•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 **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 **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 **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•ARM引入了EL2 **VirtualMachineControlStructure(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•VMEntry •VMExit •包含6个部分–Guest-statearea:发生VMexit时，CPU的状态会被硬件自动保存至该区域；发生VMEntry时，硬件自动从该区域加载状态至CPU中–Host-statearea：发生VMexit时，硬件自动从该区域加载状态至CPU中；发生VMEntry时，CPU的状态会被自动保存至该区域–VM-executioncontrolfields：控制Non-root模式中虚拟机的行为–VM-exitcontrolfields：控制VMexit的行为–VM-entrycontrolfields：控制VMentry的行为–VM-exitinformationfields：VMExit的原因和相关信息(只读区域 )**x86中的VMEntry和VMExit**•VMEntry–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VMEntry使用VMRESUME指令•VMExit–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VMEntry和VMExit**•VMEntry–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•VMExit–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 **ARMv8.1中的Type-2VMM架构**•ARMv8.1–推出VirtualizationHostExtensions(VHE)，在HCR\_EL2.E2H打开•寄存器映射：•允许与EL0共享内存–EL2中可直接运行未修改的操作系统内核(HostOS)**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 **<<内存虚拟化/IO虚拟化>>** **三种地址**•客户虚拟地址(GuestVirtualAddress,GVA)–虚拟机内进程使用的虚拟地址•客户物理地址(GuestPhysicalAddress,GPA)–虚拟机内使用的”假”物理地址•主机物理地址(HostPhysicalAddress,HPA)–真实寻址的物理地址–GPA需要翻译成HPA才能访存 **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 **第二阶段页表项**•第3级页表页中的页表项–与第一阶段页表完全一致•第0-2级页表页中的页表项–与第一阶段在高位有不(少了读写执行权限等flag )**翻译过程**• 总共24次内存访问–25-1 **TLB刷新**•刷TLB相关指令–清空全部–清空指定GVA–清空指定GPA •VMID(Virtual Machine IDentifier)–VMM为不同进程分配8/16VMID，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 **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Pagefaulthandler–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Pagefaulthandler **第二阶段页表的优缺点**•优点–VMM实现简单–不需要捕捉GuestPageTable的更新–减少内存开销：每个VM对应一个页表•缺点–TLBmiss时性能开销较大 **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 **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 **设备模拟的优缺点**•优点–可以模拟多种设备•因而可以支持较“久远”的OS–允许在中间拦截(Interposition):•例如在QEMU层面检查网络内容–不需要硬件虚拟化•缺点–性能不佳 **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 **Virtqueue** •VM和VMM之间传递I/O请求的队列•3个部分–DescriptorTable•其中每一个descriptor描述了前后端共享的内存•链表组织–AvailableRing•可用descriptor的索引，RingEntry指向一个descriptor链表–UsedRing•已用descriptor的索引 **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 **方法3：设备直通**•虚拟机直接管理物理设备 **问题1：DMA恶意读写内存** 使用IOMMU **ARMSMMU**•SMMU是ARM中IOMMU的实现–SystemMMU•SMMU的设计与AARCH64MMU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 **设备复用 Single Root I/O Virtualization(SRIOV)**•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PhysicalFunction(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 **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 **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VMEntry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ListRegister插入•不打断虚拟机执行–通过GICITS插入 **VirtualCPUInterface** •GIC为虚拟机提供的硬件功能–VM通过VirtualCPUInterface与GIC交互–VMM通过PhysicalCPUInterface与GIC交互 **不打断虚拟机执行：GICITS**•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GICITS(InterruptTranslationService)•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 **<<轻量级虚拟化>>** •轻量级虚拟化是为了更好的启动性能和运行密度•两种思路–更轻量级的虚拟化技术–在内核中增加更多的 name space•三种隔离技术–虚拟机隔离–容器隔离–虚拟化容器 **<<安全>>** 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 **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 **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 **访问控制与引用监视器** •访问控制–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 •Reference Monitor 负责两件事：1.认证 Authentication ：确定发起请求实体的 身份2.授权 Authorization：确定实体确实拥有访问资源的 权限 **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 **最小特权级：SUID** 机制•问题： passwd 如何工作？–用户有权限使用 passwd 修改自己的密码–用户的密码保存在 etc 目录下的文件，用户无权访问•解决方法：运行 passwd 时使用 root 身份–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 sticky 位，使得仅在执行该文件时才会被提权，从而将进程提权的时间和能力降至最小 **基于角色的访问控制(RBAC**•RBAC 将用户与角色解耦的访问控制方法–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)**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•强制访问控制(MAC Mandatory Access Control–什么数据能被谁访问，完全由底层的系统决定 **BellLaPadula 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 **能力机制(Capability)**•Capability 列表是权限矩阵的实现方法之一–从实体角度出发，列出该实体所拥有的能访问的对象及相应的权限–文件系统中， fd 就是一种 capability–Linux 还定义了许多其他的 capability **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 **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•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 **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 **操作系统漏洞分类的三个角度**•漏洞类型–指攻击所利用的漏洞类型–包括：栈 堆缓冲区溢出错误、整形溢出错误、空指针 指针计算错误、内存暴露错误、 use after free 错误、 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•攻击模块–指攻击所利用漏洞的所在的内核模块–包括调度模块、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–包括提升权限、执行任意代码、内存篡改、窃取数据、拒绝服务、破坏硬件等 **Returnto user 攻击(ret2usr**•内核错误的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 **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 TLBmiss 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 PXN Privileged eXecute Never **ret2di**r•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–因此，同一块物理内存在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，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 **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 防御 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 **IOS的系统安全** -文件加密保护机制•每个文件都加密，且秘钥均不相同 -进程沙盒机制•所有第三方 App 都运行在一个沙盒中 -第三方 App 都以 mobile 的非特权用户身份运行 **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 **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•即被攻击者无意通过侧信道泄露了自己的数据 缓存信道(Cache Channel)•常见的四种攻击方式–Flush+reload–Flush+flush–Prime+probe–Evict+time **Flush+Reload**•攻击步骤–1. 攻击进程首先将 cache 清空•可通过不断访问其他内存来占满 cache ，或直接通过 flush 将 cache 清空–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–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 **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 **防御侧信道** 1常量时间(Constant Time )算法 2不经意随机访问内存(ORAM )**Meltdown攻击原理**•CPU 的漏洞–若让 CPU 预测执行一条跨权限非法内存访问的指令， CPU 不会进行权限检查；若最后预测条件不满足， CPU 不会报错–攻击者可通过预测执行的方式，非法访问内核数据，然后根据内核数据改变 cache 状态，最后观测 cache 状态反推出内核数据•问：属于侧信道还是隐秘信道？•答：属于隐秘信道(Covert Channel–攻击的两侧一边是构造 cache 状态，另一边是读取 cache 状态，两边都是攻击者控制；中间的墙，就是用户态与内核态的隔离 **KPTI:内核页表隔离防meltdown <<硬件辅助系统安全>> 硬件Enclave**•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 **隔离的方法**•基于预留的隔离(硬件)–例如： PRM Processor Reserved Memory–CPU 预留一部分物理内存，不提供给操作系统•基于页表的隔离(操作系统)–例如：保证操作系统无法映射应用的物理内存页•问题：页表是由操作系统自己管理的，监守自盗？•基于插桩的隔离(编译器)–例如： SFI Software Fault Isolation•在每次访存前插入边界检查，性能损失较大 **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 **EPC(Enclave Page Cache)**•CPU 预留一部分内存，仅允许 Enclave 访问•操作系统负责将 EPC 映射至 Enclave 中 **硬件内存加密**•方法一：单密钥加密•方法二：多密钥加密•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 **<<调试与测试>>** **调试器** 1Linux的调试支持：ptrace系统调用 2配置断点-到断点陷入内核3内存断点-断点寄存器监视内存 **调试操作系统** 1虚拟机模拟2跑在用户态3自己实现GDBstub **性能调试** 1找性能瓶颈2硬件计数器-获取事件发生次数3采样-事件累计到一定数量中断看状态-硬件支持精确采样，立刻中断 4控制流追踪 5从调用栈反推 6硬件支持7理解程序行为 8静态追踪-print 9动态追踪-动态插入追踪函数 **测试** 1单元测试和集成测试 2新代码引入-回归测试3兼容性测试-向下硬件向上应用4可靠性-压力测试[模糊测试增加覆盖率] 5性能测试-找对场景 6持续集成CI