虚拟存储选讲

蒋炎岩

南京大学　　计算机科学与技术系　　计算机软件研究所

本讲概述

为什么同一个程序、同一个地址可以启动很多份?

本讲内容

- 虚拟存储：机制
- 存储器体系结构
- Meltdown

虚拟存储：机制

虚拟地址空间

操作系统中的每个进程都有独立的地址空间

- **vm-demo.c** (同一个指针、不同的物理内存)
- MMU: 硬件维护数据结构 M
  - 在运行时把地址 x 映射到 $M(x)$
  - 所有内存访问都将执行这个转换 (取指令、访存……)

虚拟地址空间设计

- 低特权程序无权修改 $M(x)$
- 访问未映射的内存产生异常 (Segmentation Fault)
- 能够控制 read/write/execute 权限
  - 思考题：代码、数据、堆栈分别设置为什么权限？

理解虚拟地址空间：VME API

内存是分为“页面”的

- 支持在内存中创建“地址空间”对象 (M)
  - 支持建立/取消页面到页面的映射
  - 可以将 M “加载”到系统上

```
bool     vme_init  (void *(*alloc)(int), void (*free)(void *));
void     protect   (AddrSpace *a);
void     unprotect (AddrSpace *a);
void     map       (AddrSpace *a, void *va, void *pa, int prot);
Context *ucontext  (AddrSpace *a, Area kstack, void *entry);
```

虚拟存储：硬件实现

地址翻译的实现

实现数据结构好办，就是个 `map<uintptr_t,uintptr_t>`

- 查询/修改速度要快，最好 $O(1)$
- 空间消耗要少 (`map` 消耗 $c \cdot n$ 的内存, $c > 1$)

我们的老朋友：局部性

- 内存是连续排布的：代码、数据、堆栈……
- 而且访问也是局部的
- 因此可以按页面 (4KB, 4MB, ...) 来分配和管理内存

地址翻译：普通实现

4GB/4KB = 1024^2，那就实现成一个 2 层的 1024 叉树。

数据结构实现的技巧

- 未映射的地址空间不需要分配
- 根节点可以控制子节点的属性

地址翻译：文艺实现

维护一张查找表。

- $(x_i, y_i, b_i) \Rightarrow \forall m \in [x_i, y_i), M(x_i + m) = b_i + m$
- 在每次内存访问时由硬件负责查表
  - 查找失败 ➡ 异常 (操作系统重填)

这是可编程 MMU (MIPS)

- 非常灵活，可以实现任意页表
- 但 TLB miss 重填相对较慢
- MIPS 是学生最容易实现完整计算机系统的指令集

地址翻译：二逼实现

用一个 hash table 维护 $(x, asid) \rightarrow M(x)$ 的映射。

为什么这是可以的？

- 只要系统里所有进程都 ASLR，hash 冲突就会很少
- J Huck, J Hays. Architectural support for translation table management in large address space machines. In *Proc. of ISCA*, 1993.

存储器体系结构

第 1 层：寄存器

寄存器其实是地址空间非常小的内存 (**mov-regs.S**)

- 5bit → 32 个寄存器
- 6bit → 64 个寄存器
  - 因此寻址通常是非常快的

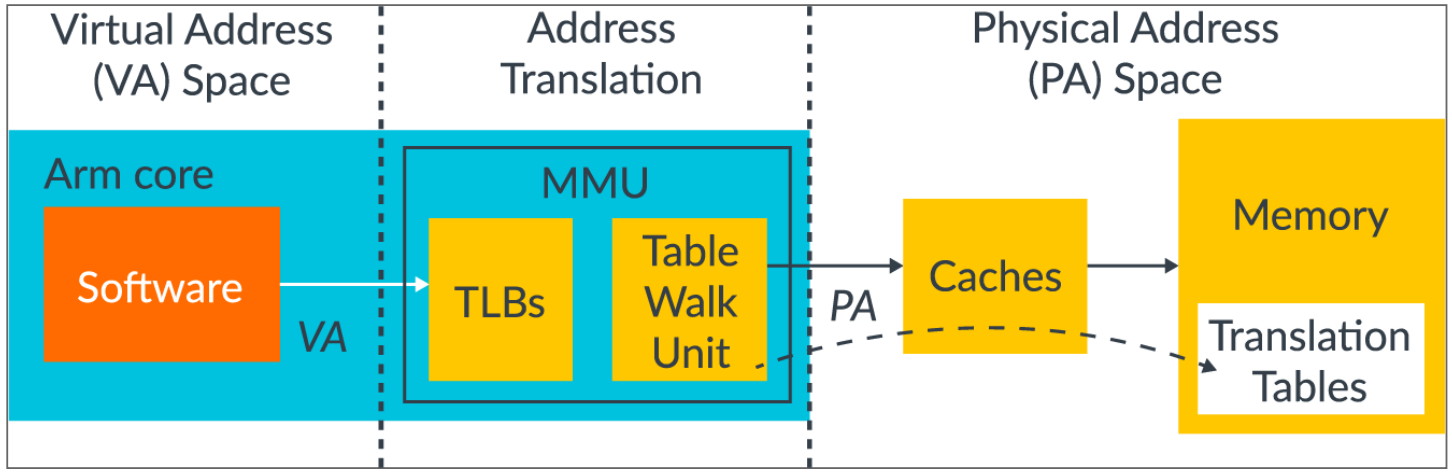
实际上，会有更大的 Physical Register File (PRF)

- 我们执行的指令 (如 `mov $1, %rax`) 会被“重命名”
  - 即分配 PRF 的地址
  - 乱序执行、投机执行、异常处理……

第 1.5 层：MMU

Armv8-A 手册

- Radix tree
  - 每个节点可以是 4KB, 16KB, 64KB



第 2 层：缓存

内存访问符合 Locality of References，“局部性原理”

> A phenomenon in which the same values, or related **storage** locations, are frequently accessed, depending on the **memory access pattern**.

从另一个角度，根据内存访问的历史，通常能较为准确地预测未来可能访问的内存，并且访问临近内存居多

- 缓存对寄存器/CPU 是完全透明的 (吗？)

第 3 层：内存 (DRAM)

内存 DRAM 即一个大数组 (三维数据结构)

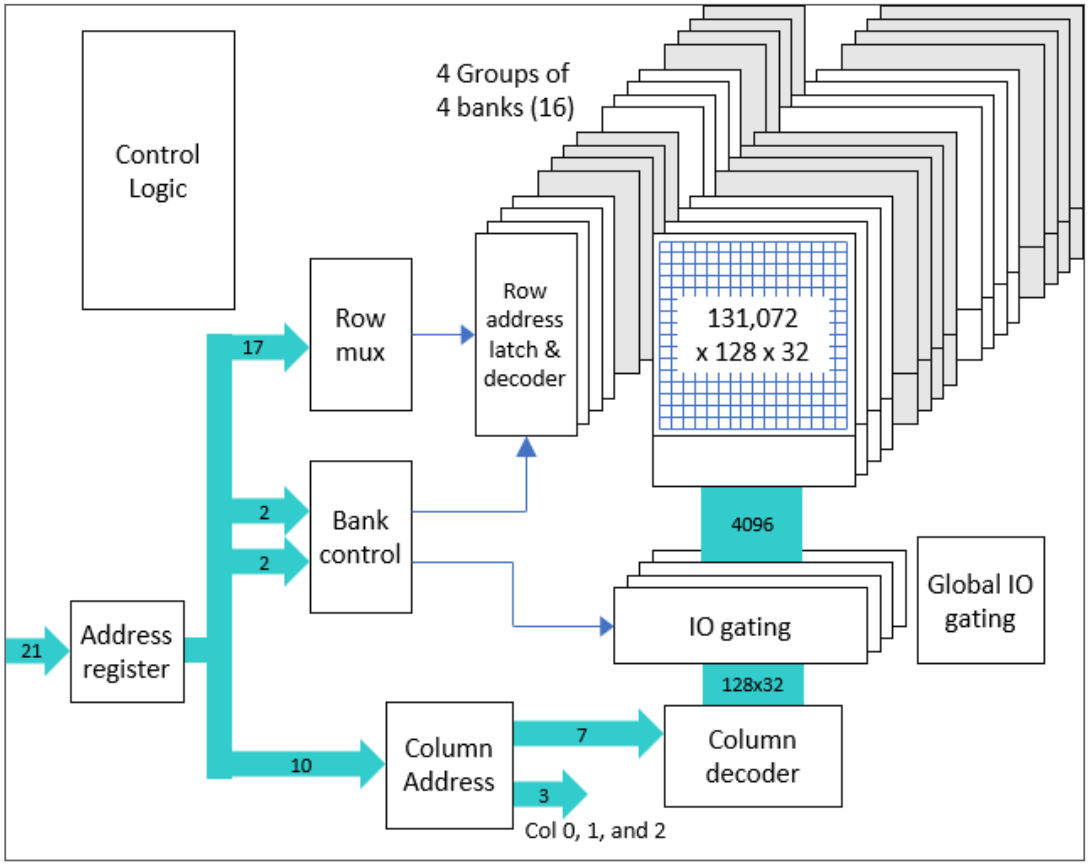


Image source: qdpma.com

第 3.5 层：NVRAM

当内存/cache 越来越大、prefetch 越来越好……

- DRAM 的快慢已经不是那么敏感了
  - 巨大的功耗 (定时刷新)
  - 内存错误率无法忽视
- 新的存储技术
  - PCM (phase change memory)
  - 产品: Intel Optane DC

第 4 层：外部存储器

SSD Flash; 磁盘

- 更大的延迟
- 更大的容量
- 更低的成本

中断 + 虚存 + 缓存 + 投机执行 = ?

问题 (1): 实现上的一个大麻烦

缓存虚拟地址还是物理地址?

缓存虚拟地址

- 每次切换 M (进程) 都要清空缓存
- M 的别名 (aliasing) 问题

缓存物理地址

- 实现简单
- 每次 cache 访问都要等地址翻译

问题 (2): 把操作系统代码/数据的映射

```
char buf[SIZE];
...
ssize_t nwrite = write(fd, buf, SIZE); // in printf
...
```

操作系统

- 希望直接向 buf 里写入
- 内核/用户进程映射同一地址空间 (但访问权限不同)
  - *(KERNEL_ADDRESS) 将引发 page fault (SIGSEGV)

操作系统：实现

在地址空间中同时维护操作系统内核与用户进程的内存映射。

例子

- `0xc0000000-0xffffffff` 内存用户进程不可访问 (U-bit = 1)
  - 操作系统代码/数据映射到此
- 中断处理程序入口是 `0xc0001234`
  - 中断发生后，处理器自动切换运行级别
  - 合法跳转到操作系统代码执行 (`%eip = 0xc0001234`)
    - 但 Ring 3 访问将导致 segmentation fault

Meltdown (2017)

复杂系统里有惊喜！

- 中断 + 虚存 + 缓存 + 投机执行 = Meltdown
- **Meltdown: Reading kernel memory from user space**

```
// %rcx: 无权限访问的地址; %rbx: 未载入缓存的数组
        xorq   %rax, %rax
retry:  movzbq (%rcx), %rax   // 非法访问; Page Fault
        shlq   $0xc, %rax
        jz     retry
        movq   (%rbx, %rax), %rbx // (%rbx + (%rcx) * 4096)
```

- 思考题：如何修复这一硬件漏洞？

End.