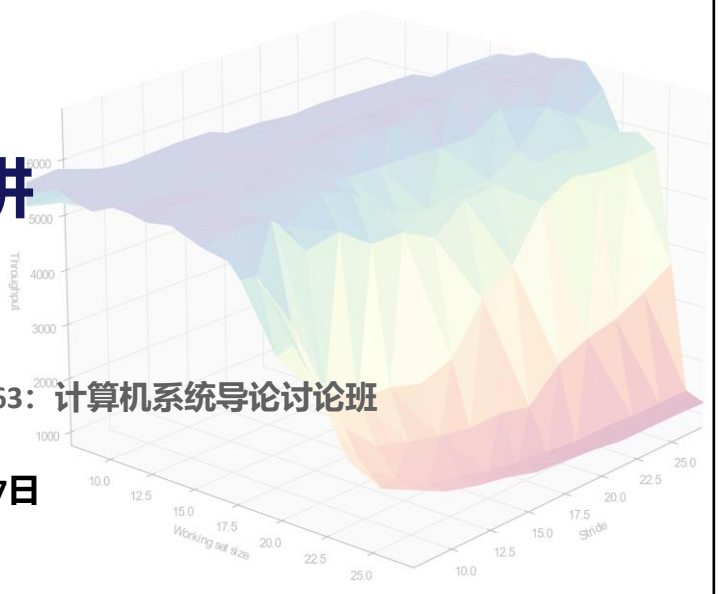
第九讲



PKU 04832363：计算机系统导论讨论班

王畅

2021年12月7日

Activity 1

轮流回答问题

书本内容梳理

- **虚拟内存的三大设计目的和效果（总体理解）**
  - 主存成为磁盘的高速缓存（解决内存不足问题，提高利用率）
  - 进程地址空间独立，简化了链接、加载和共享的过程（P566）
  - 提供了保护内存的抽象机制
- **虚拟内存部分将和ECF、存储器层次结构等综合考察，不论是考题和lab都是最具思考难度的**

考点1：物理/虚拟地址空间（硬件部分）

- **寻址的大致过程**
  - 虚拟地址通过MMU帮助翻译（硬件实现！）
  - 通常嵌入式系统和早期超级计算机会采用物理寻址
  - 高速缓存常常是物理寻址的
- **地址空间的概念**
  - 采用二进制
  - 地址空间是主存、交换空间、内存映射I/O端口、ROM的抽象（而不是书中所说的单单只是主存和磁盘I/O设备的抽象）
- **虚拟地址的结构**
  - 虚拟地址通常包括虚拟页号VPN、虚拟页偏移量VPO

4

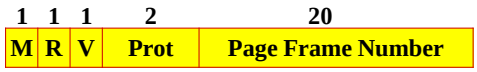
虚拟内存是硬件和软件共同配合实现的机制。翻译主要由MMU完成，其中需要用到的页表由软件处理。

考点2：虚拟内存的实现（重要！）

- **虚拟内存抽象成磁盘上的数组**
  - 从而内存是磁盘的**缓存**（swap分区）
  - 由此分为三种：未分配（没人用）、未缓存（有人用，不在主存）、已缓存（有人用，且在主存）
  - **分页**（实现不唯一，还有分段或段页式等）：虚拟和物理内存都被分为同样大小的虚拟页和物理页，物理页又叫页框或页帧
  - DRAM缓存就是指主存，显然它是全相联的。大的不命中处罚导致页框一般都比较大（Linux一般4KB，Windows可达2MB）且是写回的（dirty bit）
- **虚拟页的各种信息存储在页表中（软件实现！）**
  - 每个页面都对应一条页表条目（PTE）
  - 权限位（页故障，不是保护故障）、有效位、访问位、dirty位、实际物理地址位置
  - 缺页、调入页面（交换swap）、分配页面的原理和缓存类同（局部性、工作集、抖动的概念）

工作集：指在某段时间间隔Δ里，进程实际要访问的物理页面的集合

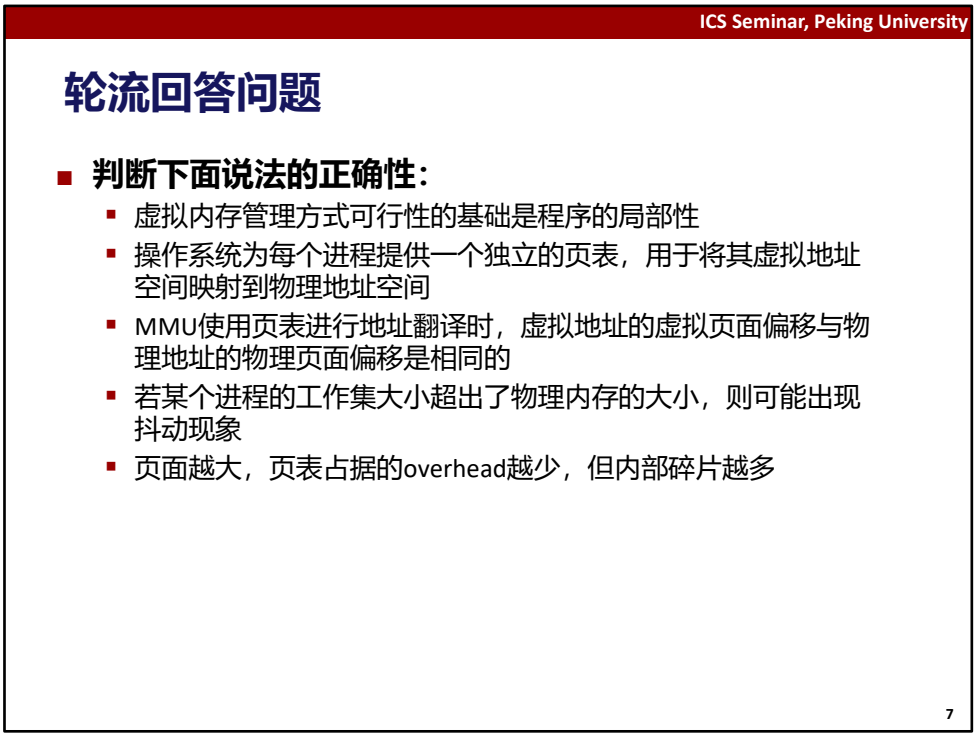
页表项的基本结构



- The page frame number (PFN) determines physical page
- The Valid bit says whether or not the PTE can be used
  - It is checked each time the virtual address is used
- The Protection bits say what operations are allowed on page
  - Read, write, execute
- The Referenced bit says whether the page has been accessed
  - It is set when a read or write to the page occurs
- The Modified (dirty) bit says whether or not the page has been written
  - It is set when a write to the page occurs

6

注意区分虚拟地址和页表项。页表项中不含VPN。



都是对的。尤其注意页表每个进程都有一个，是软件实现的，属于上下文的一部分；TLB原则上也应该各有一个，但是TLB是硬件，所以上下文切换时要刷新TLB。内部碎片内是由于系统分配给进程的空间大于其所申请的大小而产生的空间浪费。

轮流回答问题

- **假定整型变量A的虚拟地址为0x12345cf0，另一整型变量B的虚拟地址0x12345d98，假定一个page 的长度为0x1000字节，A的物理地址数值和B的物理地址数值关系应该为：**
  - A. A的物理地址数值始终大于B的物理地址数值
  - B. A的物理地址数值始终小于B的物理地址数值
  - C. A的物理地址数值和B的物理地址数值大小取决于动态内存分配策略
  - D. 无论如何都无法直接判定两个物理地址值的大小

B。注意到A和B的VPN一样，说明其虚拟页和物理页是同一个，因此前后顺序由offset决定。当然如果不在同一个虚拟页中，则实际的物理地址大小就不确定。

考点3：地址翻译（重要！）

- **最完整的过程**
  - 生成虚拟地址，传送给MMU
  - MMU计算VPN和VPO，通过`PTBR`和VPN试图访问页表项
    - 页表项一定常驻内存，但会先查TLB缓存，没有再查高速缓存，最后才去主存，以提高翻译效率
    - TLB缓存的结构：高度相联，一个块一个PTE，组索引是VPN最低几位；**上下文切换需要清空/刷新TLB！**
  - 根据页表项，看是否有访问权限等，如果没有，返回错误。否则，计算物理页的地址，看是否在高速缓存中，在则取数据
  - 若不在，查对应物理页是否在主存中，在则取数据，并对高速缓存作一定的操作
  - 若还是不在（看页表有效位），则触发缺页异常，进入内核缺页异常处理程序，调入页面；如果需要换出页面，根据替换算法选择牺牲页，并检查牺牲页是否dirty，若dirty需要写回磁盘。最后更新PTE并返回重新执行产生缺页异常的指令
- **做题方法：仔细模拟，不要漏过任何一步**

轮流回答问题

- **对于虚拟存储系统，一次访存过程中，下列命中组合不可能发生的是：**
  - A. TLB未命中，Cache未命中，Page未命中
  - B. TLB未命中，Cache命中，Page命中
  - C. TLB命中，Cache未命中，Page命中
  - D. TLB命中，Cache命中，Page未命中

10

D，如果cache命中，则page一定在内存中（SRAM中），不可能不命中。A是冷不命中的情形；B可以是PTE不在TLB中，但页面在缓存中；C可以是页面不在缓存中，但在主存中。

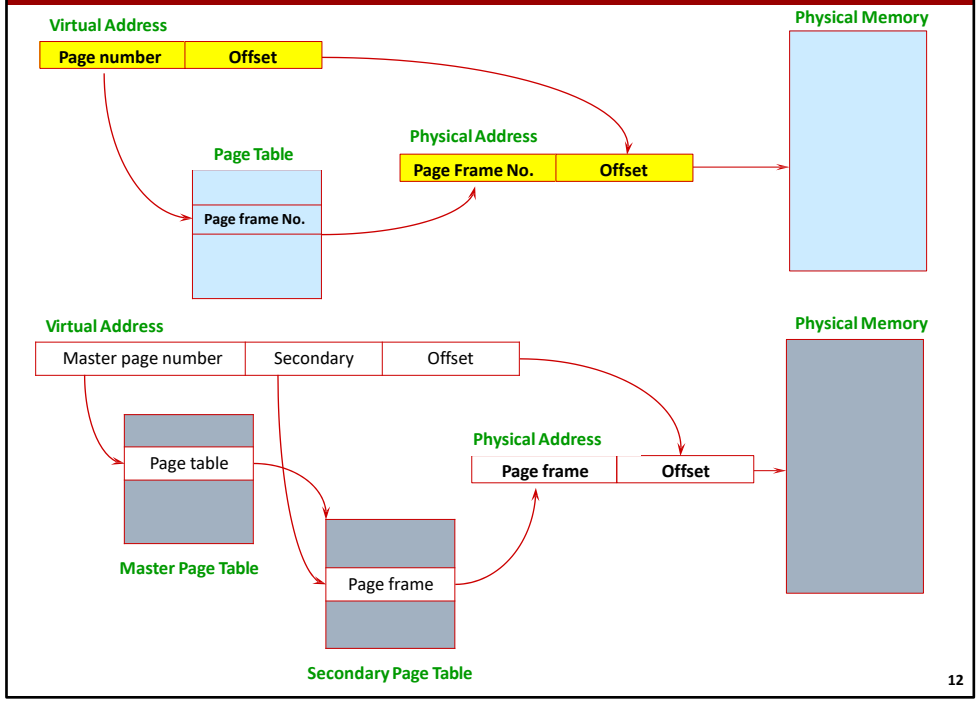
考点3：地址翻译+多级页表

- **主要动机：节约存储页表用的内存空间**
  - 内存通常是一片一片分配的，有大片空闲
  - 只有一级页表需要常驻内存
  - 页表项从线性表重新组织为树型结构，大量未分配的子树不需要分配下一级页表项
- **变化**
  - *k*级页表的VPN划分为*k*个PTE，分别表示各级页表中的“索引”
  - 每次地址翻译时要访问*k*次PTE（重复TLB、高速缓存、主存的搜索）
- **页表空间和结构的计算方法**
  - 除非额外说明，一级页表项恰好占据一页。每级页表项数目尽量均匀。除非额外说明，64位系统页表项8字节，32位4字节

11

没有分配的大块内存对应的页表项不存在。如果页表项存在，那么在Linux中一定是在主存中的。

页表结构的计算方法尽量均匀的原因：希望尽量让每个表都刚好占据内存中的一页，这样便可以令页表在物理内存中按页对齐，在设计PTE条目的时候只需要指明页开头即可。一般来说，系统可能支持多种不同的页大小，但是VPN各级的长度不会变化。


Virtual Address
Page number
Offset
Physical Memory
Page Table
Physical Address
Page Frame No.
Offset
Page frame No.
Virtual Address
Master page number
Secondary
Offset
Physical Memory
Page table
Physical Address
Page frame
Offset
Master Page Table
Page frame
Secondary Page Table

轮流回答问题

- **虚拟内存中两层页表和单层页表相比，最主要的优势在于：**
  - A. 更快的地址翻译速度
  - B. 能够提供页面更加精细的保护措施
  - C. 能够充分利用代码的空间局部性
  - D. 能够充分利用稀疏的内存使用模式

13

D。多级页表降低翻译速度（但由于TLB的存在，影响不大）；页式存储管理一般都能达成B、C两项。

轮流回答问题

- **已知某系统页面长8KB，页表项4字节，采用多层分页策略映射64位虚拟地址空间。若限定最高层页表占1页，则就惯例而言，它可以采用多少层的分页策略?**

5层。VPO需要13位，页面大小8KB，每级页表尽量有2K个条目，即11位，所以可以分5层填充64位虚拟地址空间。

轮流回答问题

- **假设有一台64位的计算机的物理页块大小是8KB，采用三级页表进行虚拟地址寻址，问它的虚拟地址的 VPN（Virtual Page Number，虚拟页号码）有多少位?**

15

注意64位的机器的虚拟和物理地址空间都不一定是64位的（但32位的机器可能确实都是32位的），这里VPO为13位，所以一页能存储1K条目，根据多级页表的结构原则，VPN为3*10=30位。

轮流回答问题

- **Intel的IA32体系结构采用二级页表，称第一级页表为页目录（Page Directory），第二级页表为页表（Page Table）。页面的大小为4KB，页表项4字节。以下给出了页目录与若干页表中的部分内容，例如，页目录中的第1个项索引到的是页表3，页表1中的第3个项索引到的是物理地址中的第5个页。则十六进制虚拟地址8052CB经过地址转换后形成的物理地址应为十进制的多少?**

页目录		页表1		页表2		页表3	
VPN	页表号	VPN	页号	VPN	页号	VPN	页号
1	3	3	5	2	1	2	9
2	1	4	2	4	4	3	8
3	2	5	7	8	6	5	3

VPO为12位，第一级页表应该有1K条目，即VPN1为10位，所以VPN2也是10位。由此8052CB的VPN为1000 0000 0101，所以VPN1为2，VPN2为5，从而对应页表1，页号7，所以物理地址是72CB=29687。做题的时候注意进制换算，不要抄错了。

考点4：虚拟内存系统实例（硬件）

- **记住Core i7页表和页表项的基本结构**
  - CR3寄存器是PTBR寄存器，上下文切换要刷新！
  - 四级页表，页表项8字节一个，推测虚拟地址有几位？
  - 图9-22全部地址的分位意义（VPN、VPO、TLBT、TLBI、PPN、PPO、CT、CI、CO的相互联系）
- **几个重要字段（注意对比前三级表和第四级的不同！）：**
  - P位子页表是否在物理内存中（有效位），Linux中前三级P恒1（不是说页表都存在，意思是页表只要存在就会在内存中）
  - dirty位只有第四级有，但引用位每级都有（MMU负责）
  - PS位定义页大小，只有第一级表有
  - 权限位举例：R/W读写、U/S用户模式或内核模式、XD是否可执行
  - 先检查P位，再检查R/W、XD、U/S

17

主要对比不同。

轮流回答问题

- **在Core i7中，以下哪个页表项属于4级页表项，不属于1级页表项?**
  - A. G位 (Global Bit)
  - B. D位 (Dirty Bit)
  - C. XD位 (Disable or enable instruction fetch)
  - D. U/S位 (User or supervisor mode access permission)

B，只有最后一级对应真正的物理页面，所以dirty位有用。

轮流回答问题

- **在Core i7中，关于虚拟地址和物理地址的说法，不正确的是：**

A. VPO = CI + CO
B. PPN = TLBT + TLBI
C. VPN1 = VPN2 = VPN3 = VPN4
D. TLBT + TLBI = VPN

D，VPN用来查找TLB中的条目，其中低位是组索引。

Windows页表自映射（了解）

- **原理**
  - 32位系统下，所有的PTE所占的空间刚好是4MB。如果将这些PTE连续地放在内存中，那么这4MB内存空间对应的PTE（“PTE的PTE”）刚好在一个4KB页中，而这个4KB在页目录表中刚好也占一项。
  - 如果能再合理地设置二级页表项的地址，那么就可使PTE的PTE所占的4KB的内容与PDE所占的4KB的内容完全相同（为什么能这样做？因为PDE的目标地址和PTE的PTE的目标地址是一样的）。这样一来，可以就可以将页目录表也作为一个页表，这也意味着，页目录表中有一项会指向自己。
  - 人话：页目录除了作为页目录之外，还作为整个大页表在内存中的二级页表。
- **以IA32体系结构为例：**
  - 二级页表，12位VPO，10位VPN1，10位VPN2
  - 页表安排在0xc0000000~0xc0400000，页目录安排在0xc0300000~0xc0400000

20

IA32系统应该采用二级页表，分为页目录和页表，各10位。

为什么PDE的目标地址和PTE的PTE的目标地址是一样的？意思是二级页表每个表都是1个页面，这个页面当然也需要页表项，我们不难发现它的页表项其实就是PDE，所以PDE可以“一人分饰两角”。

只要确定了页表所在的4MB空间的位置，就可以确定出页目录的地址。实际上，假设位置是x，则页目录应该是x + 4 * (x >> 22)。所以在windows中一般存储页表连续表的开头而不是页目录的位置。0xc0000000的位置是精心选择的，但是并不唯一（应该满足什么条件？）。具体分析可以看，例如，https://www.cnblogs.com/siviltaram/p/windowskernelmapping.html。

这样做虽然能每个进程的页表都节约4KB，但是代价则是需要从虚拟地址空间中分配出连续的4MB对齐的4MB的空间。

考点5a：Linux虚拟内存实例（软件）

- **一种段页式内存管理（但主体还是页式）**
- **图9-26（对比ECF部分进程的图）**
  - 用户栈向低地址增长
  - 代码段在`0x4000000`开始（这也是为什么一般可执行文件无需编译为位置无关代码）
  - 堆向高地址增长（动态分配内存）
- **如何实现的?**
  - 每个进程一个`task_struct`，里面有一个`pgd`表示其第一级页表基址，以及`mmap`链表表示进程每个段的虚拟内存的情况（包括起始位置、终止位置、权限等）

考点5b：内存映射（重要！）

- **结合ECF和System I/O**
- **内存区域可以对应（映射）磁盘文件，以填充这段内存的内容**
  - 按需页面调度，映射≠进入主存
  - 还可以映射匿名文件（初始化为0）
- **共享对象和写时复制**
  - 共享对象多个进程都可见，通过不同的虚拟内存映射到同一块物理内存来实现
  - 写时复制：要写私有对象时，触发保护故障，再复制（lazy）
  - `fork`和`execve`的实现（`fork`：先标记为只读，再`COW`）
- **`mmap`函数**
  - 知道其用法
  - 一定，一定要`man mmap`看一下手册！

22

`mmap`用起来比直接读写文件可能更快。一般映射时有共享对象和私有对象，共享对象的物理内存同时被映射到多个进程的虚拟内存中，所以其修改各进程都可见。对于每个映射私有对象的进程，其页表条目都是标记为只读，试图写的时候会触发保护故障，然后进行写时复制（`COW`）：故障处理程序重新创建页面副本，更新页表条目，返回后重新执行写操作。`fork`通常就是采用`COW`机制实现的。`mmap`映射文件后如果触发`COW`，则新的更改不会体现在文件中。

`mmap`的第一个参数是建议的起始地址，系统有权选择不在那里开始，因此返回值不一定等于`start`参数。

`mmap`的常见用途：对文件需要频繁读写，则可以映射到内存中，用内存读写代替`I/O`，提高速度；用特殊的匿名文件进行映射，可以为关联的进程提供共享的内存空间。

轮流回答问题

- **进程P1通过fork()函数产生一个子进程P2。假设执行fork()函数之前，进程P1占用了53个（用户态的）物理页，则fork函数之后，进程P1和进程P2共占用_____个（用户态的）物理页；假设执行fork()函数之前进程P1中有一个可读写的物理页，则执行fork()函数之后，进程P1对该物理页的页表项权限为_____。**

53；只读。fork后虚拟内存仍然映射到相同的物理页，直到试图写。因此后一个为只读（不论父子进程）。

轮流回答问题

- 当x处为MAP_PRIVATE时，标准输出上的两个整数是什么？如果是MAP_SHARED呢？

```c
int main()
{
    pid_t pid;
    int child_status;
    long *f = mmap(NULL, 8,
                   PROT_READ | PROT_WRITE,
                   ______X______ | MAP_ANONYMOUS, -1, 0);
    *f = 0;
    if ((pid = fork()) > 0)
    { // Parent
        waitpid(pid, &child_status, 0);
        *f = *f + 1;
        printf("Parent: %ld\n", *f);
    }
    else
    { // Child
        *f = *f + 1;
        printf("Child: %ld\n", *f);
    }
    return 0;
}
```


如果是私有的，则数据相互独立，为1、1；如果是共享对象，二者都能看到，所以是1、2。

Activity 2

问题求解

考点S：ECF、缓存、文件和虚存的综合考题

- **熟练掌握地址翻译的整个过程（结合cache、TLB，不要忘记任何一个环节）**
  - 主要考怎么用页表，但不能忘记其他内容
- **内存映射和共享对象的基本机制**
  - 进程地址空间独立，`fork`出来的进程，虚拟内存映射到的位置暂时是同一个位置，直到触发COW机制
- **用户态，只能访问虚拟地址**
- **大题练习题下次做**

any questions?

Thanks & 感谢观看