Lab2 实验报告

游子诺

2019 年 4 月 6 日

1 实验思考题

Thinking 1

请思考 cache 用虚拟地址来查询的可能性，并且给出这种方式对访存带来的好处和坏处。另外，你能否能根据前一个问题的解答来得出用物理地址来查询的优势?

使用虚拟地址查询 cache

优点 不需要经过 TLB 和 MMU 的虚拟地址转换，可直接在 cache 中映射查找，提升一定的性能。

缺点 不同虚拟地址可指向同一个物理内存，cache 读取虚拟地址可能进一步浪费存储空间；在操作系统中，每个进程都拥有一套页目录和页表进行虚拟地址的映射，因此不同进程相同的虚拟地址可能指向同一块物理地址，cache 在接受虚拟地址时还要进程的信息，在查找索引时更复杂（等同于在内部执行了 TLB 和页管理的功能）；cache 和内存之间通信用“块”，而虚拟地址管理单位为“页”，两者单位可能不兼容。

使用物理地址查询 cache 的优势

- 充分使用 cache，使得其中不存在重复内存地址的内容。
- 和虚拟存储系统低耦合度，cache 的输入接口是物理地址，与实际物理内存之间是一对一的关系，逻辑清晰简单，使得虚拟地址管理和物理地址管理模块化。
- 允许对虚拟存储管理和物理存储管理使用两种不同单位，以更适应不同种类硬件设备。

Thinking 2

请查阅相关资料，针对我们提出的疑问，给出一个上述流程的优化版本，新的版本需要有更快的访存效率。（提示：考虑并行执行某些步骤）

- cache 缺失前的查询操作：指导书流程中，主存的访问都是基于已确定 cache 缺失的情况下执行的，但可以并发向 cache 和主存提起物理地址查询的请求，若 cache 命中则阻断还在进行中的主存查询，若不命中，则继续主存查询（节省下 cache 查询不命中的时间）。
- cache 缺失后的存取操作：若 cache 发生缺失，则直接访问主存读取数据，指导书流程中从主存得到的数据会先装填至 cache 中再返回，但其实装填和返回可并行执行，节约时间。

Thinking 3

在我们的实验中，有许多对虚拟地址或者物理地址操作的宏函数 (详见 include/mmu.h), 那么我们在调用这些宏的时候需要弄清楚需要操作的地址是物理地址还是虚拟地址，阅读下面的代码，指出 x 是一个物理地址还是虚拟地址。

```
int x;
char* value = return_a_pointer();
*value = 10;
x = (int) value;
```

x 是一个虚拟地址，在利用 C 语言和汇编语言编写操作系统时，所有实质访问了内存获取数据（不包括计算时形式化表示的物理地址）的指针其所存储的地址都是虚拟地址。在实际访问数据时，根据虚拟地址所处的区间（内核、用户、外设、操作系统等），会进行 MMU、去高位等操作化为物理地址（这些对虚拟地址的操作都是硬件层面的），而后真正地进行访问。自己先转换成物理地址再访问就是画蛇添足，会出现问题。

Thinking 4

我们注意到我们把宏函数的函数体写成了 do /* ... */ while(0) 的形式，而不是仅仅写成形如 /* ... */ 的语句块，这样的写法好处是什么？

根据循环的逻辑，显然 while 语句是不会再被执行第二次的，等同于一次顺序执行。但使用 do while(0) 包括后，宏函数内部定义的变量等将不会与宏函数引用处的变量出现同名的冲突，程序执行时默认会使用循环体内部的变量，提升安全性和适用性。

Thinking 5

注意,我们定义的 Page 结构体只是一个信息的载体,它只代表了相应物理内存页的信息,它本身并不是物理内存页。那我们的物理内存页究竟在哪呢?Page 结构体又是通过怎样的方式找到它代表的物理内存页的地址呢?请你阅读 include/pmap.h 与 mm/pmap.c 中相关代码,给出你的想法

物理内存在哪里? 从 pmap.c 函数中的 alloc 我们可以了解到,操作系统作为最底层的软件部分,当其知晓了物理内存大小和自身和物理内存的映射关系后(去除高 3 位),其在分配物理内存时就是 freemem 指针的形式化移动,其对物理内存的操作本身没有任何限制(但是为了正确性需要满足正确的内存操作逻辑),而用户程序在访问和申请非法内存空间时被拒绝是因为操作系统的回绝。

因此,对于页管理操作系统也是类似的,物理内存本是一个整体,操作系统自定义将其划分为 4kb 的内存页(可以理解为“更大的字节”),结构体数组 pages 中每一个元素对应一页,之间通过链表的方式连接。

Page 结构体映射到物理内存页 对于 pages 中的一页 Page,找到其对应的物理页顺序:用 PPN 宏函数 (&Page-pages 指针减法),再根据物理页顺序,使用 page2pa(乘以每页的大小),就得到了页的起始地址。

Thinking 6

请阅读 include/queue.h 以及 include/pmap.h, 将 Page_list 的结构梳理清楚, 选择正确的展开结构 (请注意指针)。

答案 C

Thinking 7

在 mmu.h 中定义了 bzero(void *b, size_t) 这样一个函数, 请你思考,此处的 b 指针是一个物理地址,还是一个虚拟地址呢?

答案 由之前物理地址和虚拟地址的辨析可知,此处 b 指针是一个虚拟地址。

Thinking 8

了解了二级页表页目录自映射的原理之后,我们知道,Win2k 内核的虚存管理也是采用了二级页表的形式,其页表所占的 4M 空间对应的虚存起始地址为 0xC0000000,那么,它的页目录的起始地址是多少呢?

计算页表存储在第几页:`0xC0000000>>12 = 0xC0000`

根据顺序确定页表中项相对页表的偏移量（32=8*4）：0x000C0000 *4 = 0x00200000
计算页表项的地址：addr = 0x0020000+0xC0000000 = 0xC0020000

Thinking 9

思考一下 tlb_out 汇编函数，结合代码阐述一下跳转到 NOFOUND 的流程？从 MIPS 手册中查找 tlbp 和 tlbwi 指令，明确其用途，并解释为何第 10 行处指令后有 4 条 nop 指令

跳转到 NOFOUND 流程 TLBP 会在 TLB 中寻找匹配的项，如果没有找到则会将 CP0-INDEX 高位置 1，因而 CP0-INDEX 变为一个负数，结合 bltz 指令，当 CP0-INDEX 为负数就会跳转到 NOFOUND 标签后的指令。

加入 nop 的原因 TLB 查询需要时间，加入 nop 以等待 TLB 查询完成，防止由于过早读取而造成忽略了 NOFOUND

Thinking 10

显然，运行后结果与我们预期的不符，va 值为 0x88888，相应的 pa 中的值为 0。这说明我们的代码中存在问题，请你仔细思考我们的访存模型，指出问题所在。

定义的 va 已经不在内核部分了，从 va->pa 的过程中其实经过了“软件 MMU”，而最后一句中 pa->va 的转换还是假设 va 是内核地址，+ULIM 方式得到虚拟地址是错误的。

Thinking 11

在 X86 体系结构下的操作系统，有一个特殊的寄存器 CR4，在其中有一个 PSE 位，当该位设为 1 时将开启 4MB 大物理页面模式，请查阅相关资料，说明当 PSE 开启时的页表组织形式与我们当前的页表组织形式的区别。

PSE 技术允许 4MB 的页面于传统的 4KB 物理页面共存，以传统的两级页表虚拟内存管理系统来说，具体有以下几个方面的不同：

- 页目录项标志位：同一个页目录中，根据第 7 位标志位的不同，可以并行存在小页和大页的情况。若第 7 位为 0，则为小页，指向的是一个二级页表；若第 7 位为 1，则为大页，指向的直接是一个 4MB 的物理页地址。
- 页目录项地址位：当为大页状态时，页目录中 20 位地址仅有高 10 位有效，低 10 位都是 0。

2 实验难点图示

在 lab2 中，我认为较为困难的部分还是物理内存管理相关概念的理解，在此我用 ppt 做了一张丑图，涵盖了 lab2-1 中我认为困难和重点的部分。

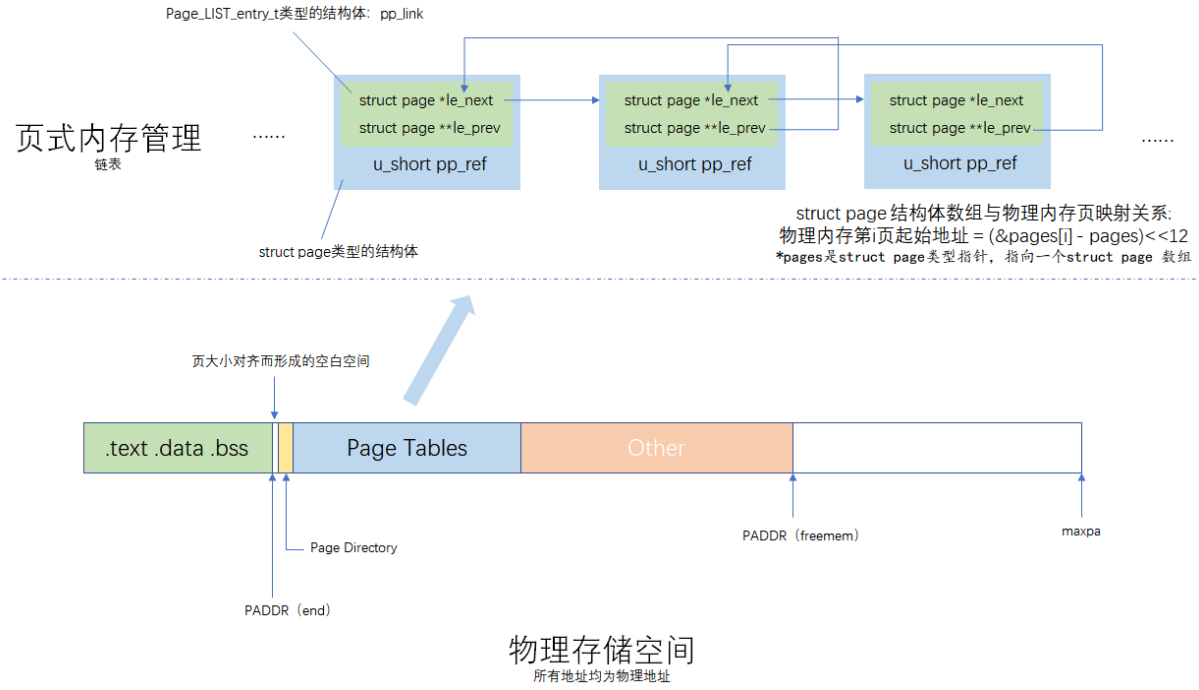

图 1: 物理内存管理重点图示

- freemem 的初始值 end[ ] 和 linker_script 的关系。
- 物理内存各个区段在使用 mem_init 函数后的分布情况。
- PADDR, KADDR, page2pa 等宏函数的理解和应用，重点关注三个对象：虚拟地址、物理地址和页（页基于虚拟地址）。
- 基于链表的页式内存管理，尤其是嵌套结构体分析和二重指针的功能。
- struct page 类型结构体和物理内存页映射的方法和公式，主要涉及到指针的减法。

3 体会与感想

本次作业是关于内存管理控制的，从中我体会到操作系统对物理内存管理的重要性，体会到了“形式化的内存管理”实现的具体方式，清晰了虚拟内存管理中页目录页表项的组织形式和各种不同的条件情况。

3.1 物理内存管理

首先，lab2 让我清晰地了解到操作系统在物理内存管理上的重要性。作为运行在硬件之上的最底层软件，操作系统对内存管理的重要性不言而喻，在物理内存管理章节中，alloc 函数分配内存空间、boot_segment 等让我体会到操作系统对物理内存操作的高度自由，没有所谓的非空内存和空内存，硬件就像机器一样：“操作系统指哪里，硬件就返回哪里”……自由是双刃剑，在灵活的同时，也埋下很多潜在的风险，因此操作系统的编写必须严谨细致，也要通过完备的规则（页式管理，虚拟内存等技术）“避免未来位置情况下的错误行为”。

其次，在物理内存管理中，“形式化管理”的思想也给我很深的影响。在理解 lab2 的过程中，处处给我困惑的一个很大原因就是**自映射**的问题，一般的管理者并不处在被管理者中，但是内存管理不一样：freemem 指针分配内存然而自己却也是内存的一部分、page 索引物理页内存然而自己也是物理内存之一、某个页表中存在指向自己的页表项等。其实，自映射并不矛盾，操作系统中有许多**以小管大**的情况，采用的就是形式化管理 + 保护的措施，例如：freemem 指针通过移动分配内存，但是其自身存在与 0x80000000 end[ ] 之间的 data 段，因此保护了自己免受影响；pages 管理依照结构体指针减法求得页的顺序与物理页一一映射，管理范围涉及到了自己，但是 pages 定义在内核中，后续 OS 操作不会更改此内容、外部程序无法访问内核程序两个条件也将 pages 保护起来。

3.2 虚拟内存管理

虚拟内存管理相较于物理内存管理，从理解上要容易一些，主要内容是二级页表的管理：页目录项和页表项的 32 位是如何组织的？页目录和页表存储在哪里？二级虚存管理的自映射现象？建立虚存和物存的映射本质是什么？（是把页表虚拟地址对应项的值更新为物理页）弄清这几个问题对 pmap.c 中各函数的作用就比较清晰了。

而在实践编码中，我还是遇到了一些问题：首先是各种条件分支的完备判断，集中体现在 dir_work 函数上，dir_walk 函数中 create、PTV-E、alloc 失败等都会产生条件分支，题目中的提示便省略了当没有找到且 create==0 和 create==1 但没有内存的两种情况下对某些变量的操作，这时候一定要靠使用完整的 if-else 条件语法构建所有的分支情况，对于没有提示的分支，从其被调用的地方找依据（例如本题中没有找到且 create==0 时 *ppte 应该赋予 0）。其次，有关 pp_ref 的变量理解也要注意，pp_ref 指

代该物理页被引用的次数，只有变为 0 才能被释放，除了某个物理页被虚拟地址索引会让 pp_ref 增加以外，当物理页被用作页表时，其也被页目录所索引，因此 pp_ref 也该增加，这点请尤其注意。

最后，学完内存管理后我也还有疑惑，需要后面的知识才能解答，比如在我看来目前搭载的页目录和页表系统仅是适用于“内核程序”，每个进程都有一套完整的虚拟内存，那么管理每个进程的虚拟地址又是被声明、存放和使用的呢？

4 指导书残留难点与反馈

lab2 实践中我遇到了两个难点，分别隶属物理内存管理和虚拟内存管理两个部分.

物理内存管理

Page 结构体与实际物理内存之间的映射 指导书有该内容的思考题，但我认为存在的位置有些不对，因为实验一上来就让我们填写 queue.h 中的链表宏函数，而一看链表的结构显然就发现其中并没有存储任何和地址相关的内容，进而编写链表函数对如何进行物理页管理完全是懵圈的，导致编码时很迟疑，浪费的时间很多。建议两种方式：

- 不更改顺序，在进行链表宏函数编写前先强调“只需明确了解链表结构和各种操作，对于结构体和物理页映射关系后续会讲解”
- 更改顺序，先显示地说明（不用思考题）结构体是如何与物理页建立映射的（指针减法），再实现链表操作的宏函数。

Page 链表构造的实现 lab2 中 page 链表是基于传统的“单向链表”进行了改进，新增了 struct page** p_pre 二重指针，这个新增的二重指针相对于传统的双向指针修改前继的 next 更巧妙，也给了我更大的困惑，建议在指导书中增加该部分的解读，最好的方式是画一个链表的图：其中 1 号链表块的 next 指向 2 号链表块，而 2 号链表块中的 p_pre 又引出来单独指向 1 号链表块中的 next 指针。

虚拟内存管理

页目录、页表和页框的大小 惊奇地发现 directory、table 和 page 的大小都是 4kb，这其中是制定者人为规定的巧合？还是有科学依据呢？期待解答！

下面，我给出我的一种思路：假设页目录和页表每项的大小都是 32 位，页大小设计的目的是为了“在二级页表的方法下，一张页大小的页目录和若干一张页大小的页表能够将所有的内存映射完”，设页内偏移量的二进制表示有 x 位，因此一页有 2^x 字节，由于每项大小为 32，所以一个页目录或页表中有 $2^(x-2)$ 项。由二级页表的查找方法，高 $x-2$ 位索引页目录中的项，中 $x-2$ 位索引页表的项，其余低 x 位为页内偏移量，若地址总长度为 32 位，则可得方程并解得 $x=12$

$$x+(x-2)+(x-2)=32$$

页目录项的构成 指导书 2.6.1 节——两级页表机制，有讲到页目录和页表中每一项（32 位）的组成情况，其中说明了页表中的 32 位每项是由 20 位地址 +12 位标志位组成的，而对于页目录则没有这样说明（其实也是这样的），这样一个说明而另一个没

有说明的表述容易使读者错误地认为 directory 和 table 中项的构成是不同的，希望进一步优化该部分表述。