Lab2实验报告

Thinking

Thinking1

请你根据上述说明，回答问题：在我们编写的 *C* 程序中，指针变量中 存储的地址是虚拟地址还是物理地址？*MIPS* 汇编程序中 *lw, sw* 使用的是虚拟地 址还是物理地址

C程序指针变量是虚拟地址，MIPS程序里也是虚拟地址

T2

- 请从可重用性的角度，阐述用宏来实现链表的好处。

答：用宏实现链表，可以用较少的代码实现许多链表的功能，既可以减少重复的代码，也可以减轻维护、找bug的难度，只需要修改一次宏，就可以免去修改大量重复代码。

- 请你查看实验环境中的 */usr/include/sys/queue.h*，了解其中单向链表与循环链表的实现，比较它们与本实验中使用的双向链表，分析三者在插入与删除操作上的性能差异。

答：

链表类型	**插入**	**删除**
双向链表	实现向后、向前插入容易，不需循环。性能好	删除 元素只需要一条指令，`*(elm)->field.le_prev = (elm)->field.le_next`
单向链表	实现向后插入只需要两条指令即可；无法实现向前插入，除非给定链表头，需要循环遍历，性能很差	删除链表项时，需要从链表头开始遍历找到该项的前一个元素，性能差。
循环链表	在链表尾部插入一个元素只需3条指令；实现向后插入也只需2条指令；实现向前插入不需要给定链表头，但是 需要循环遍历，性能差	需要从链表头开始遍历 找到该项的前一个元素，性能差

T3

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。

```
//include/pmap.h
//LIST_HEAD(Page_list, Page);
struct Page_list{
    struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;
        u_short pp_ref;
    }* lh_first;
    //struct Page *lh_first;
}
//struct Page_list page_free_list;
```

T4

请你寻找上述两个 boot_* 函数在何处被调用。

两个函数都是在mm/pmap.c文件中被调用的。

boot_pgdir_walk被boot_map_segment函数调用，用来获取虚拟地址为va+i所对应的二级页表项的地址。

```
pgtable_entry = boot_pgdir_walk(pgdir,va + i,1);
```

boot_map_segment函数被mips_vm_init()函数调用，将虚拟地址UPAGES映射到为Page结构体所分配的物理内存；将虚拟地址UENVS映射到为进程管理所用到的Env结构体按页分配的物理地址。

```
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);
```

T5

- 请阅读上面有关 *R3000-TLB* 的叙述，从虚拟内存的实现角度，阐述 *ASID* 的必要性

对于一个可以由多个进程的操作系统，每个进程都自认为自己可以使用全部的地址空间。这就导致可能有一个虚拟地址对应多个不同的物理地址。ASID就用来 唯一标识进程，并为每个进程提供地址空间保护。TLB试图解析虚拟页号时，它确保当前运行进程的 ASID与虚拟页相关的ASID相匹配。除了提供地址空间保护外，ASID允许TLB同时包含多个进程的条目。如果TLB不支持ASID，每次当操作系统从一个进程转变到另一个进程时，TLB 就必须被冲刷（flushed），这会大大降低访存的效率。

- 请阅读《*IDT R30xx Family Software Reference Manual*》的 *Chapter 6*，结合 *ASID* 段的位数，说明 *R3000* 中可容纳不同的地址空间的最大数量

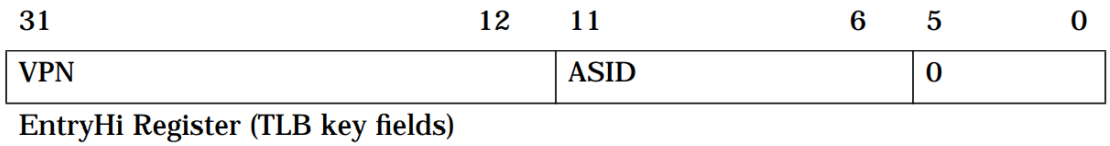


Figure 6.1. EntryHi and EntryLo register fields

ASID共6位，故可容纳不同的地址空间的最大数量为2^6 = 64

T6

- *tlb_invalidate* 和 *tlb_out* 的调用关系是怎样的?

  `tlb_invalidate`调用`tlb_out`，完成对特定`tlb`表项的清空

- 请用一句话概括 *tlb_invalidate* 的作用

  使用 `tlb_invalidate` 函数可以根据特定虚拟地址，删除其在`tlb`中的表项。每当kuseg中的页表被修改， 就需要调用该函数，以保证下次访问该虚拟地址时诱发 TLB 重填以保证访存的正确性。

- 逐行解释 *tlb_out* 中的汇编代码

```
LEAF(tlb_out)
//定义叶子函数
//1: j 1b
nop
    mfc0    k1,CP0_ENTRYHI
    //将目前EntryHi寄存器的值保存到k1寄存器，以便函数结束时恢复原先EntryHi中的内容
    mtc0    a0,CP0_ENTRYHI
    // a0 is the key, moved to EntryHi
    nop
    tlbp
    // 根据EntryHi的key值，将对应的tlb表项的索引保存到Index寄存器中
    nop
    nop
    nop
    nop
    mfc0    k0,CP0_INDEX
    //将要清空的tlb表项的索引保存到k0
    bltz    k0,NOFOUND
    //如果k0 < 0, 说明tlb中没有key对应的表项
    nop
    mtc0    zero,CP0_ENTRYHI
    //将EntryHi置0
    mtc0    zero,CP0_ENTRYLO0
    //将EntryLo置0
    nop
    tlbwi
    // 根据Index的索引，将该tlb表项的key和data置0
NOFOUND:

    mtc0    k1,CP0_ENTRYHI
    //恢复调用函数前的EntryHi的值
    j   ra
    //返回函数
    nop
END(tlb_out)
//结束函数定义
```

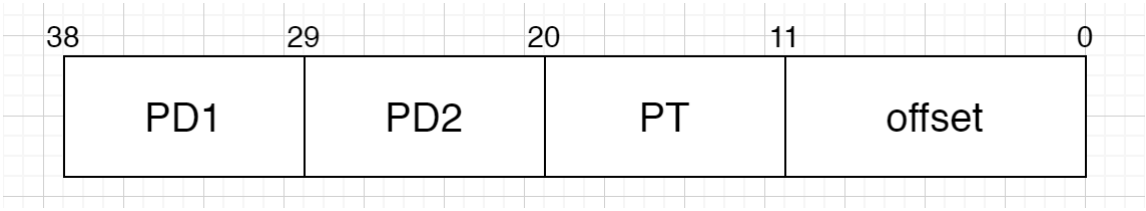
T7

在现代的 *64* 位系统中，提供了 *64* 位的字长，但实际上不是 *64* 位页式存储系统。假设在 *64* 位系统中采用三级页表机制，页面大小 *4KB*。由于 *64* 位系统中字长为 *8B*，且页目录也占用一页，因此页目录中有 *512* 个页目录项，因此每级页表都需要 *9* 位。因此在 *64* 位系统下，总共需要 $3 \times 9 + 12 = 39$ 位就可以实现三级页表机制，并不需要 *64* 位。现考虑上述 *39* 位的三级页式存储系统，虚拟地址空间为 *512*

GB，若记三级页表的基地址为 *PTbase* ，请你计算：

- 三级页表页目录的基地址

首先，虚拟地址可以分为四段：



可以看出，有1个一级页表，2^9 个二级页表，2^{18} 个三级页表，共2^{27} 个页。由自映射可知，一级页表是2^9 个二级页表中的一个，所有的二级页表又都包含在2^{18} 个三级页表中，同时，在地址空间里，所有三级页表的地址是连续分布的，二级页表也是。知道这些，就可以计算了。

二级页表的基地址： $PD2_{base} = PT_{base} + (PT_{base} >> 12) * 8 = PT_{base} | PT_{base} >> 9$

一级页表基地址：$PD1_{base} = PD2_{base} + (PD2_{base} >> 12) * 8 = PT_{base} | PT_{base} >> 9 | PT_{base} >> 18$

三级页表页目录的基地址就是上式

- 映射到页目录自身的页目录项(自映射)

该页目录项的基地址就是：

$PD1_{base} = PD1_{base} + (PD1_{base} >> 12) * 8 = PT_{base} | PT_{base} >> 9 | PT_{base} >> 18 | PT_{base} >> 27$

T8

简单了解并叙述 *X86* 体系结构中的内存管理机制，比较 *X86* 和 *MIPS* 在内存管理上的区别

X86在内存管理方面采用段页式存储管理方式，地址映射机制分为两个部分：

1. 段映射机制，将逻辑地址映射到线性地址。逻辑地址是指令发出的地址，需要从地址段描述结构（GDT或LDT表）获取基地址，将逻辑地址与基地址相加 得到 线性 地址。
2. 页映射机制，将线性地址映射到物理地址。线性地址就相当于虚拟地址，此处访存与页式访存类似。

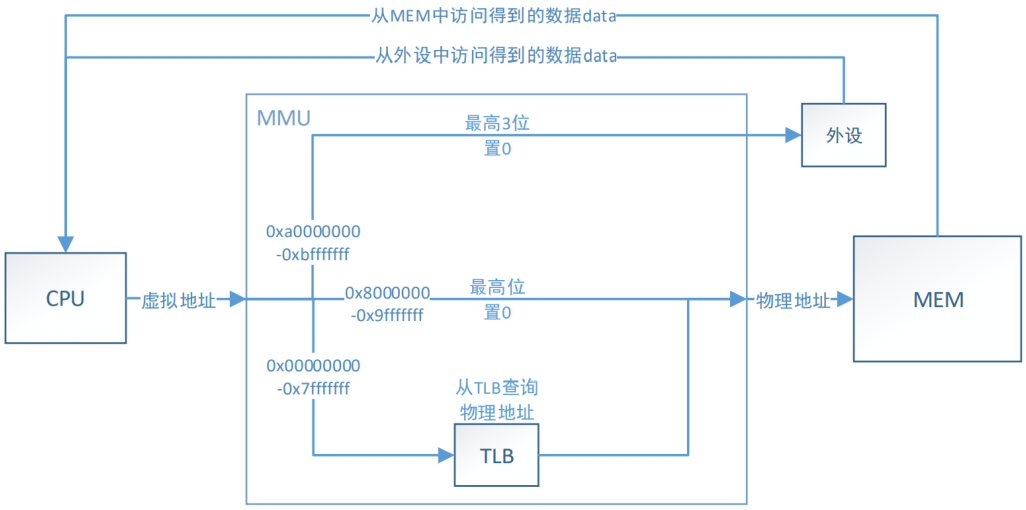
区别：

1. X86 采用的是段页式的管理，读或写数据所需地址映射需要经过2步，访存3次；MIPS采用纯分页管理，如果只采用1级页表，映射只需要1步，访存两次。
2. x86下tlb由硬件实现，软件对于 tlb控制只有 一种方式：tlb刷新。当页表被修改时，或发生进程切换时，由于原有TLB中缓存的内容已经失效，此时必须通过软件触发TLB刷新操作。但是查询或填充tlb不需要软件操作； MIPStlb功能需要软件实现，包括刷新、重填 、查询。

实验难点

Lab2难点有很多

访存流程



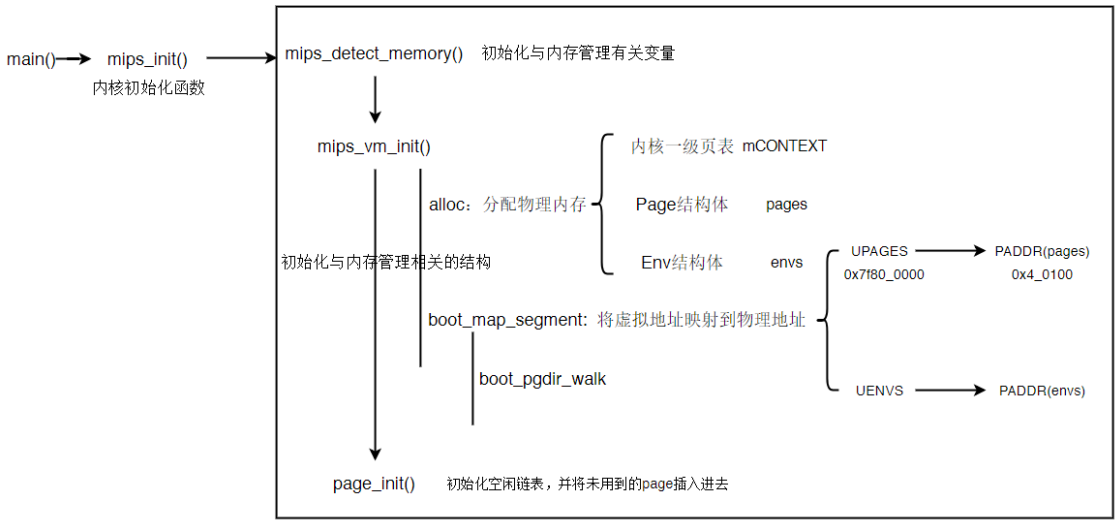
这张图很好，说明CPU发出虚拟地址进行访存的时候，是通过MMU进行的。这就说明，类似于`lw k0, 0x8000_0000(zero)`指令，访存的时候会由硬件MMU自动进行变换。但是，低2G空间进行访存的时候，需要先在Lab2用软件实现TLB才行。

对于之前的kseg0的虚拟地址，可以使用`PADDR`和`KADDR`宏进行虚拟地址和物理地址的转换。当然，这里有一个小疑问：C语言程序不都使用虚拟地址吗，为什么要有PADDR呢，不是MMU可以自动转换吗？答案是，填入页表项的物理页号，必须是物理地址的高20位，因此需要进行从虚拟地址到物理地址的转换。

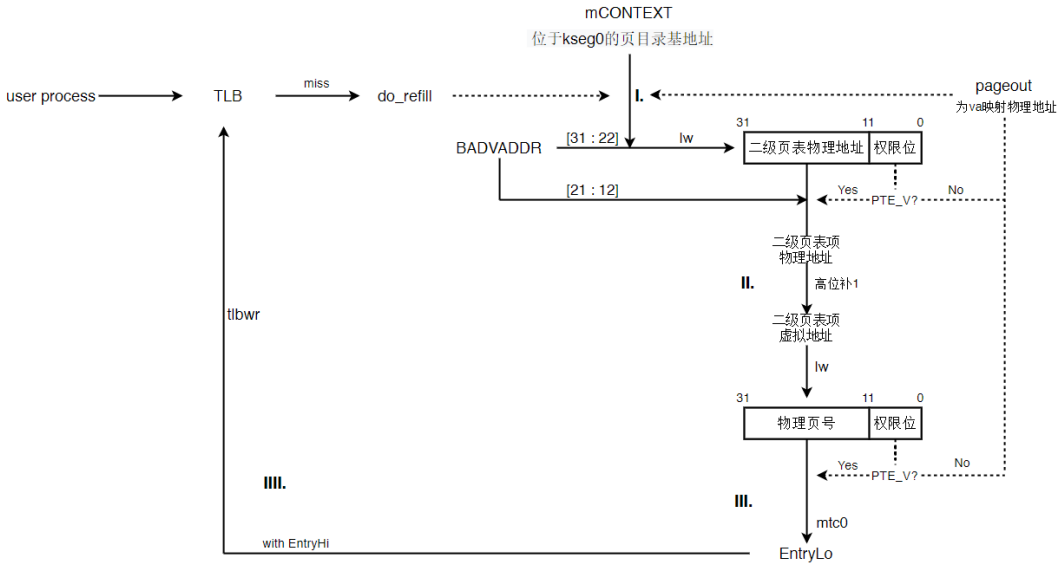
对于kuseg的虚拟地址，采用两级页表结构进行管理。

内核初始化

首先，main函数进行内核有关内存管理的初始化。

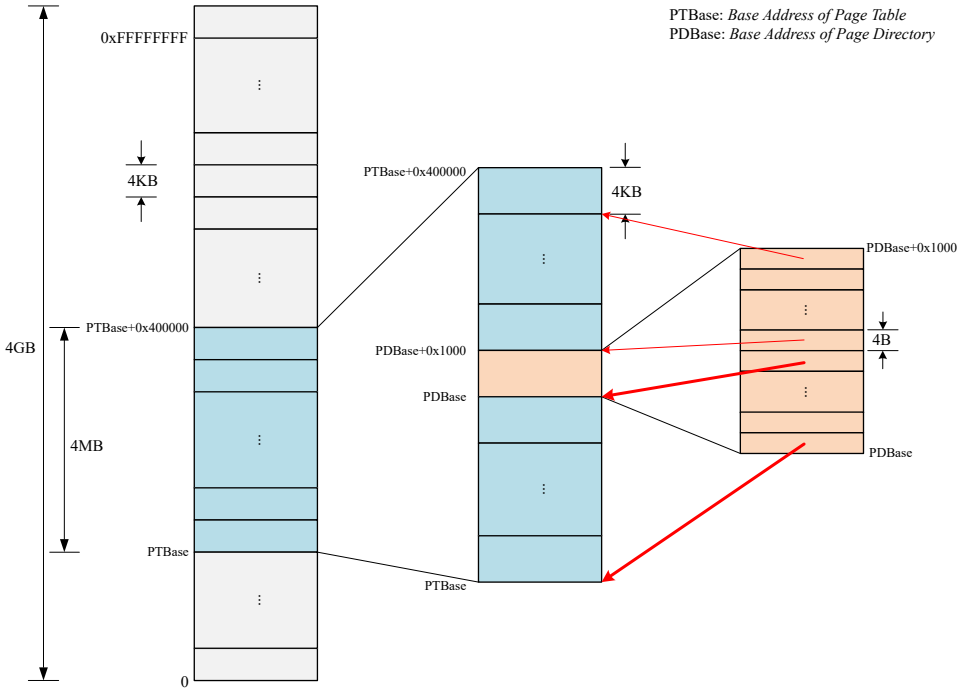


TLB重填过程



多级页表与页目录自映射

MOS 中，将页表和页目录映射到了用户空间中的 0x7fc00000-0x80000000（共4MB）区域，这意味着 MOS 中允许在用户态下访问当前进程的页表和页目录



构建自映射方法如下：

1. 给定页表基址PT_{base}，基址需要4MB对齐，即低22位都为0
2. 页目录表的基址$PD_{base} = PT_{base} | (PT_{base} >> 10)$
3. 自映射目录表项地址为$PDE_{self\text{-}mapping} = PT_{base} | (PT_{base} >> 10) | (PT_{base} >> 20)$

对2和3的说明：

2. 把4MB空间全看做是页表项的集合。由于每个虚拟页占空间4KB(12位)，故$PT_{base} >> 12$ = 第几个页表项管理的起始地址为PT_{base} (第一个页表项地址为PT_{base})。页表项占空间4B，因此($PT_{base} >> 12$) * 4表示PT_{base} 对应的页表项相对于PT_{base} 的offset, PD_{base} 的第一个页目录项pde就对应着PT_{base}， 因此，$PD_{base} = PT_{base} + PT_{base} >> 10$

3. PD_{base} >> 12表示其对应第几个页表项，$PDE_{self\text{-}mapping}$ 就是指向PD_{base}的页表项，

   故$PDE_{self\text{-}mapping} = PT_{base} + (PD_{base} >> 12) * 4$

补充：对于自映射，我们说PT_{base} 的低22位为0。但同时，由于整个页表连续分布在4MB的区域内，故1M个页表项地址的高10位都相同。虚拟地址的前10位仍然是页目录项的索引，中间10位仍是二级页表项的索引。虚拟地址的前10位 也是自映射页目录表项是第几个 页目录项。

体会与感想

内存管理实在是难，这个难并不是说学不会的难，而是说没有理解内存管理本质的难。课程组只是一味地说，如何实现内存管理，但是却没有讲，内存管理的一步步实现是为了什么，是为了解决什么问题。导致学习的时候感觉很乱，直到看了强生的博客才知道了内存管理的目的和原因。总共学了2个星期，还是有很多问题，迷糊不清。具体问题和解答如下

疑问

1. page结构体分配的虚拟地址空间在kseg0, UPAGES是个啥?

   page结构体本身，会占用一定的物理空间。UPAGES是pages链表的物理地址在kuseg中对应的一块虚拟地址，专门弄出来一个UPAGES是为了让用户态可以访问到page结构体。UPAGES和pages两个不同的虚拟地址都对应同一个物理地址。不过是UPAGES是通过页表对应的，pages是直接最高位抹0对应的

2. page和物理页面一一顺序对应，是在哪个函数实现的 呢?

   内存地址转换函数(`pa2page`，`page2pa`)就实现了这个功能。以致于`page_alloc`在分配内存时，就认为page和物理页面顺序对应。

3. 这里的页表的虚拟地址都是在kuseg,假设页表基地址为p, *p需要tlb映射，但是此时还未建立tlb映射，怎么办?

   一旦物理内存分配了空间，kseg中的虚拟地址其实也被分配了（因为映射关系是线性的）。因此，kseg0中也有一个页表(包括一级页表和二级页表)，而且，lab2中的页表都是指kseg0中的页表。因为此时低2G还未建立tlb的映射机制。

   而且，*p的p，必须是一个虚拟地址。因为CPU访存时，是发出虚拟地址，经过MMU转换后自动得到物理地址访存。因此，如果在内核态下需要访问物理地址paddr，需要经过KADDR(paddr)转换成虚拟地址后访问。

4. 什么时候修改页表 ，需要用`tlb_invalidate`?

   当然是当虚拟地址在kuseg时，要访问的物理地址通过tlb时，需要`tlb_invalidate`。

5. boot和非boot函数的本质区别

   区别就是，boot是在系统启动时被使用的，此时va是在kseg0；非boot函数是在用户进程运行中被使用的，va在kuseg中，故需要使用tlb进行地址的变换，而且一旦修改了页表，需要调用`tlb_invalidate`函数。