Lab2实验报告

19376273 陈厚伦

思考题

Thinking2.1

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?

cache使用虚拟地址查询是可能的，但是综合来看不如用物理地址查询。

cache使用虚拟地址查询的优势：节省了TLB未命中时查询页表的这一次访问内存操作。

cache使用虚拟地址查询的劣势：cache缺失时更新cache需要访问页表，增大了时间开销。不同的进程同一个虚拟地址对应可能是不同的物理地址，或者不同虚拟地址对应的是一个物理地址，cache还需要标记进程相关信息，安全性较差，不利于数据共享。

物理地址查询的优势：物理地址和内存数据是一一对应并且是稳定的，安全且利于共享。

Thinking2.2

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。

虚拟地址。一般高级程序语言编程使用的是虚拟地址。

Thinking2.3

我们在 include/queue.h 中定义了一系列的宏函数来简化对链表的操作。实际上，我们在 include/queue.h 文件中定义的链表和 glibc 相关源码较为相似，这一链表设计也应用于 Linux 系统中(sys/queue.h 文件)。请阅读这些宏函数的代码，说说它们的原理和巧妙之处。

① 使用宏定义链表的初始化、前插、后插、头插、尾插、删除等操作，定义了带表头节点的链表，和一般的函数定义非常相似，使用过程也几乎是相似的。

② 结构体宏定义过程中预留了许多内置结构体的名称，可以通过宏替换实现非常灵活的定义方式，提高了链表内容的可变性，实现了C语言的“泛型”。例如节点结构体可以自己定义，节点中前后指针的信息已经被封装为一个新的结构体，并且在节点结构体中，可以定义节点结构体的其余属性，而不影响链表的基本结构。

③ 每一个节点存放指向下一个节点的指针和指向“前一个节点指向下一项的指针”的指针，可以实现只知道一个节点的情况下节点的删除操作。

Thinking2.4

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?

前者宏替换之后是一条语句，后者宏替换之后是一个语句块。前者替换时不容易产生歧义，而且替换时方便书写，按照语句写即可（XXX;），后者替换时没有；不符合一般写代码的习惯。

Thinking2.5

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，并思考一下。

```
/* Code in pmap.h */
extern struct Page *pages;

static inline u_long
page2ppn(struct Page *pp)
{
        return pp - pages;
}

/* Get the physical address of Page 'pp'.
 */
static inline u_long
page2pa(struct Page *pp)
{
        return page2ppn(pp) << PGSHIFT;
}
```

pages是指向Page结构体的指针，相当于定义了一个Page数组，实际的物理内存页按照地址顺序和数组项是一一对应的。找到这个pp指向的Page在数组中的位置`(pp - pages)`，然后乘一个物理页的大小即通过位运算得到实际物理地址`(page2ppn(pp) << PGSHIFT)`。

Thinking2.6

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)。

```
struct Page_list{

    struct {

         struct {

            struct Page *le_next;

            struct Page **le_prev;

        } pp_link;

        u_short pp_ref;
```

```
    }* lh_first;

}
```

Thinking2.7

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?

此处的b指针是一个虚拟地址。

Thinking2.8

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?

((0xC000_0000 / 4M) * 4K) + 0xC000_0000 = 0xC030_0000

Thinking2.9

注意到页表在进程地址空间中连续存放，并线性映射到整个地址空间，思考：是否可以由虚拟地址直接得到对应页表项的虚拟地址？上一节末尾所述转换过程中，第一步查页目录有必要吗，为什么?

可以。前20位分成前10位与后10位本来就是人为划分的，并没有改变这个顺序存储的结构。二级页表机制需要放入页目录，而页表可以根据需要创建二级页表，节省了内存。第一步查页目录有必要，可以分局页目录获得所需要的二级页表，并且可以得到二级页表是否有效的信息。

Thinking2.10

观察给出的代码可以发现，page_insert 会默认为页面设置 PTE_V的权限。请问，你认为是否应该将 PTE_R 也作为默认权限？并说明理由。

page_insert的作用便是将va这个虚拟地址和我们指定的页的物理地址建立映射关系，此时页表项一定是有效的，所以默认设置了PTE_V的权限。PTE_R是脏位，在这里将PTE_R设置为默认权限是不必要的。根据R3000手册，这里的PTE_R相当于写权限位，当PTE_R为1时其实是开启了写权限，认为它是脏的，所以它被丢弃时都要被写回。实际上，这一页并不一定会被写，所以默认PTE_R会造成不必要的写回，浪费了时间。

Thinking2.11

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。

tlbp功能：找到一个TLB中匹配的条目

tlbwi功能：写一个由索引寄存器 [INDEX] 索引的TLB条目

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>
```

```
LEAF(tlb_out)
//1: j 1b
nop
        // 先把HI的值保存到k1
        mfc0    k1,CP0_ENTRYHI
        // a0是传递进函数的参数 HI储存虚拟地址空间和标志位
        mtc0    a0,CP0_ENTRYHI
        nop
        // tlbp指令 查询HI中虚拟地址是否在TLB中
        // 有匹配则把匹配的index写入INDEX寄存器
        // 无匹配INDEX寄存器最高位置1 变成负数
        tlbp
        nop
        nop
        nop
        nop
        mfc0    k0,CP0_INDEX
        // 读出INDEX 如果<0 则虚拟地址不在TLB中 进入NOFOUND
        bltz    k0,NOFOUND
        nop
        // 如果>=0 则虚拟地址在TLB中 进入下面部分 把HI LO0寄存器置0
        mtc0    zero,CP0_ENTRYHI
        mtc0    zero,CP0_ENTRYLO0
        nop
        // 前面已经把index写入了INDEX寄存器
        // 写INDEX寄存器中索引的TLB条目
        tlbwi
NOFOUND:
        // 恢复HI
        mtc0    k1,CP0_ENTRYHI
        // 直接返回
        j       ra
        nop
END(tlb_out)
```

第10行处指令后有4条nop指令原因：TLB是流水线式的，确保tlbp指令执行完毕。

Thinking2.12

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。这说明代码中存在问题，请仔细思考我们的访存模型，指出问题所在。

```
u_long* va = 0x12450;
u_long* pa;
// 将va虚拟地址和其要对应的物理页pp的映射关系以perm的权限设置加入页目录
page_insert(boot_pgdir, pp, va, PTE_R);
// 取va对应的物理页框的物理地址
pa = va2pa(boot_pgdir, va);
printf("va: %x -> pa: %x\n", va, pa);
*va = 0x88888;
printf("va value: %x\n", *va);
printf("pa value: %x\n", *((u_long *)((u_long)pa + (u_long)ULIM)));
```

原因在与函数`va2pa`的实现细节：

```
static inline u_long
va2pa(Pde *pgdir, u_long va)
{
        // 查页表的过程
        Pte *p;
        // 查一级页表(页目录)
        // pgdir_entryp = pgdir + PDX(va) 得到页目录项
        pgdir = &pgdir[PDX(va)];

        if (!(*pgdir & PTE_V)) {
                return ~0;
        }
        // 查二级页表 页表在内核中 二级页表的物理地址转内核虚拟地址使用KADDR()
        p = (Pte *)KADDR(PTE_ADDR(*pgdir));

        if (!(p[PTX(va)]&PTE_V)) {
                return ~0;
        }
        // va对应的不带偏移的物理地址
        return PTE_ADDR(p[PTX(va)]);
}
```

这个函数并没有将va的低12位页内偏移拼接到物理地址上，所以得到是va对应的物理页框的首地址，所以程序运行后两个位置打印出来的值不一致。

Thinking2.13

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。

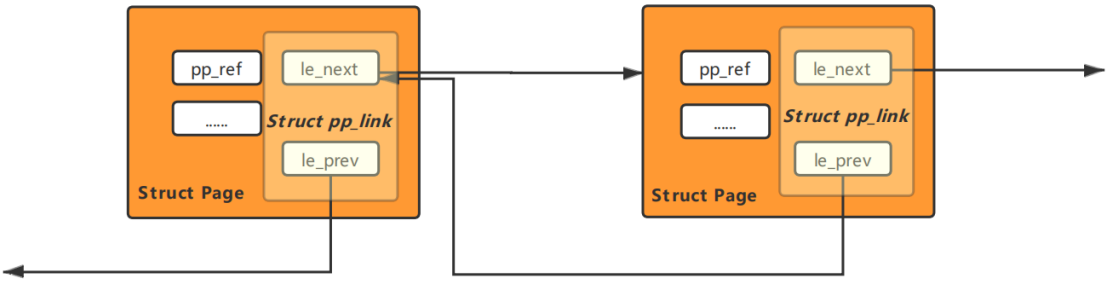
在我们的二级页表存储机制下，页目录的页目录项对应页表页，页表页的页表项对应页表，如果我们只看页目录（一级页表），忽略二级页表，可以看做页目录（一级页表）的每一项对应一个4MB的大页。所以对于x86体系结构下，页表组织形式不需要有太大改变，需要在一级页表中加一个标记记录是否开启大物理页面模式，当PSE位有效时，可以看成页目录为页表的一级页表存储机制。

实验难点图示

> LINKED LIST

本实验链表通过宏定义，使用起来有非常好的灵活度。链表指针域与一般单向链表与双向链表不同，虽然有两个指针，但其实是一个伪双向链表。

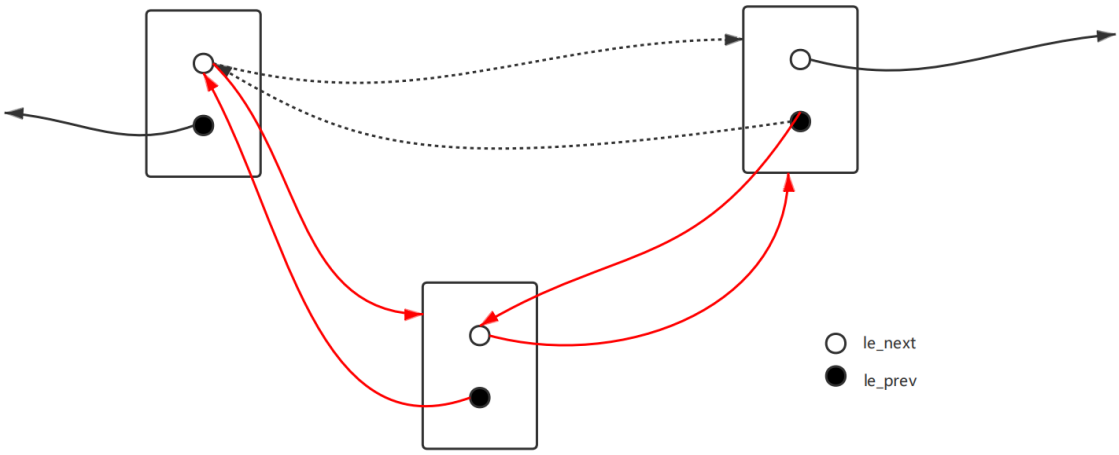
下面通过示意图说明链表结构：



每一个page结构体中包含Page_LIST_entry_t类型的pp_link结构体和一个pp_ref，其中pp_link中包含两个指针，（Page*）le_next指向链表中下一个结构体，（Page **）le_prev指向链表中上一个结构体的le_next。这是一个伪双链表，le_prev指向上一个结构体的le_next，但是不能够由此访问到上一个结构体，所以只能单向遍历。

这种定义最大的好处是简化了节点删除操作。在一般单链表中删除一个节点必须要知道它的上一个节点，所以要从头节点遍历，增大了节点删除的时间复杂度。在上述的结构下，通过le_prev访问并修改上一个节点的le_next，就可以实现删除，不需要遍历。

课下部分要求完成后插与尾插，基本思路与数据结构课程中学习到的链表相似，不做过多赘述，这里仅以后插为例展示链表的插入过程：



链表结构与链表操作都是用宏定义的方式实现的，虽然较之一般的函数实现可读性略差，但是给链表结构带来了许多灵活性，让C语言也可以实现某种意义下的“泛型”。宏定义方式规定了某些结构体的结构特征，但是并没有写死，预留了可供我们替换的空间，允许我们根据这个框架进行灵活的定义。

在LIST_HEAD的宏定义中，规定了这个头结点结构体中有一个结构体指针，但是其中的结构体名name和结构体指针所指类型type则是我们可以灵活替换的。

```
#define LIST_HEAD(name, type)
\
        struct name {
\
                struct type *lh_first;  /* first element */
\
        }
```

> 页面管理、虚实地址转换机制

本次实验我们操作的虚拟地址空间主要是kseg0，最大物理存储是64MB，所在位置已经在下面的内存地图中加以标注，在alloc内存时，**从0x8040_0000开始分配。**

这次实验中我们使用页结构体数组与空闲链表来进行页的管理。

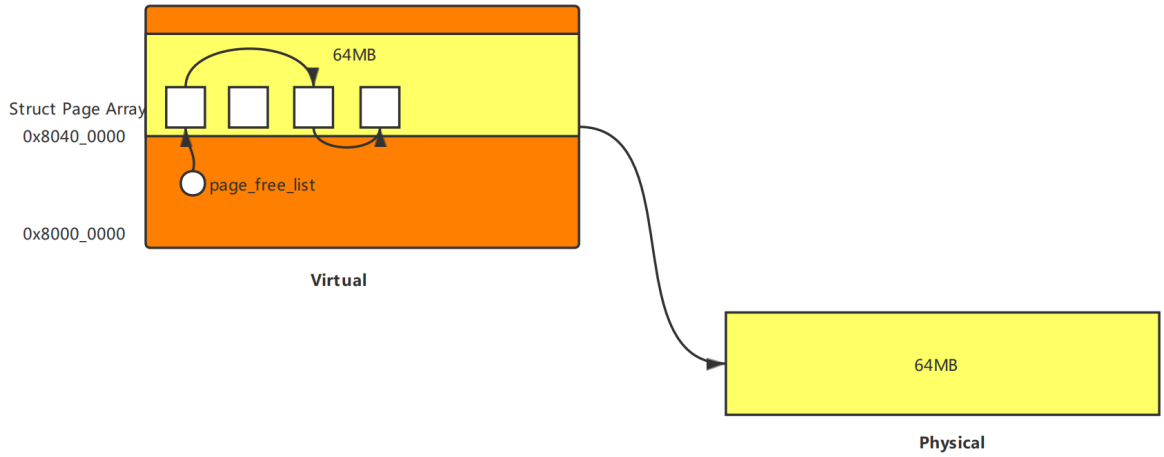
要明确**页结构体和页面不是一个概念**，它们的大小与存储位置完全不同，我们在一个单独的内存建立了页面结构体方便我们记录页面信息（pp_link、pp_ref），但是结构体本身不是页面，只不过它与虚拟页面是顺序对应的，页结构体在结构体数组中的位置即使虚拟页面在存储空间的位置，乘页面大小可以得到虚拟页面的地址。

在mips初始化函数中可以找到如下语句，其利用alloc函数为page结构体数组分配了空间，在64MB第一页中，只占较小的内存。

```
pages = (struct Page *)alloc(npage * sizeof(struct Page), BY2PG, 1);
```

空闲链表由页结构体构成，管理空闲页，页的分配与回收通过链表的插入与删除操作就可以完成，在内核的数据段存储了头结点page_free_list，通过链表将page结构体数组中空闲页结构体串联起来，就完成了空闲链表的构建。

页面管理可以通过如下示意图加以说明：



在代码中我们使用到到的地址都是虚拟地址，需要转化到物理地址，内核在kseg0段，这一段的地址映射是连续的并且不经过MMU，在mmu.h中定义了地址转换中最关键的两个函数是KADDR与PADDR，通过$\pm ULIM(0x80000000)$实现映射。以这两个宏函数为核心，建立了一系列（内核）虚拟地址、物理地址、页结构体之间相互转换的（宏）函数：PPN、VPN、page2ppn、page2pa、pa2page、page2kva。

这里需要梳理exercise中地址的转换过程。

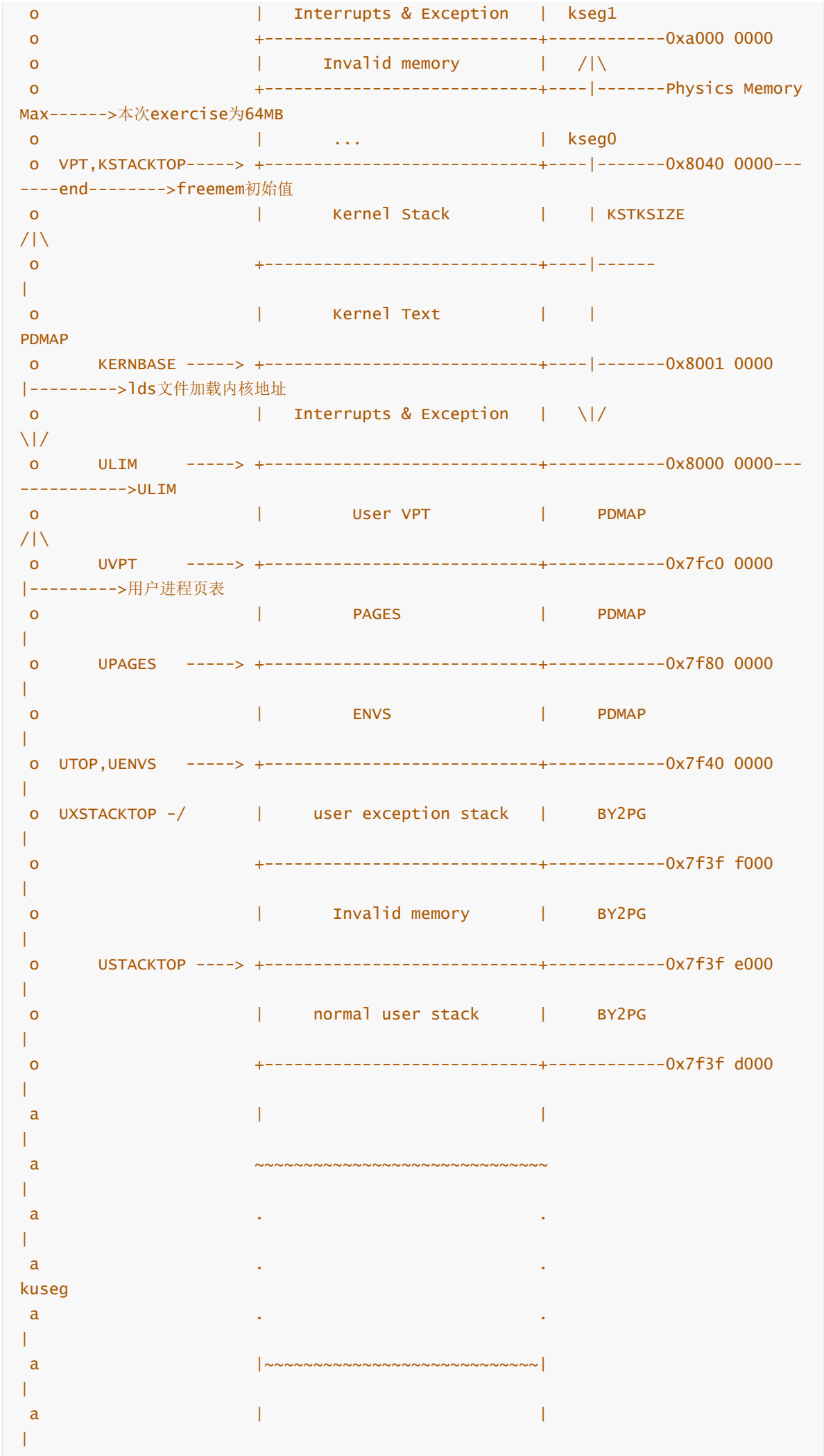
Page --> pa --> kva:

找到page在结构体数组的索引，左移12位得到页的物理地址，加0x8000_0000得到内核虚拟地址。

kva --> pa --> Page:

内核虚拟地址减0x8000_0000得到物理地址，根据物理地址计算PPN，pages数组的第PPN个就是kva对应的page结构体。

```
/*
 o     4G ----------->  +----------------------------+------------0x100000000
 o                      |       ...                  |  kseg3
 o                      +----------------------------+------------0xe000 0000
 o                      |       ...                  |  kseg2
 o                      +----------------------------+------------0xc000 0000
```


o | Interrupts & Exception | kseg1
o +----------0xa000 0000
o | Invalid memory | /|\
o +----|-------Physics Memory
Max------>本次exercise为64MB
o | ... | kseg0
o VPT,KSTACKTOP-----> +----|-------0x8040 0000---
----end-------->freemem初始值
o | Kernel Stack | | KSTKSIZE
/|\
o +----|------
|
o | Kernel Text | |
PDMAP
o KERNBASE -----> +----|-------0x8001 0000
|--------->lds文件加载内核地址
o | Interrupts & Exception | \|/
\|/
o ULIM -----> +----------0x8000 0000---
---------->ULIM
o | User VPT | PDMAP
/|\
o UVPT -----> +----------0x7fc0 0000
|--------->用户进程页表
o | PAGES | PDMAP
|
o UPAGES -----> +----------0x7f80 0000
|
o | ENVS | PDMAP
|
o UTOP,UENVS -----> +----------0x7f40 0000
|
o UXSTACKTOP -/ | user exception stack | BY2PG
|
o +----------0x7f3f f000
|
o | Invalid memory | BY2PG
|
o USTACKTOP ----> +----------0x7f3f e000
|
o | normal user stack | BY2PG
|
o +----------0x7f3f d000
|
a | |
|
a ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|
a . .
|
a . .
kuseg
a . .
|
a |~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
|
a | |
|

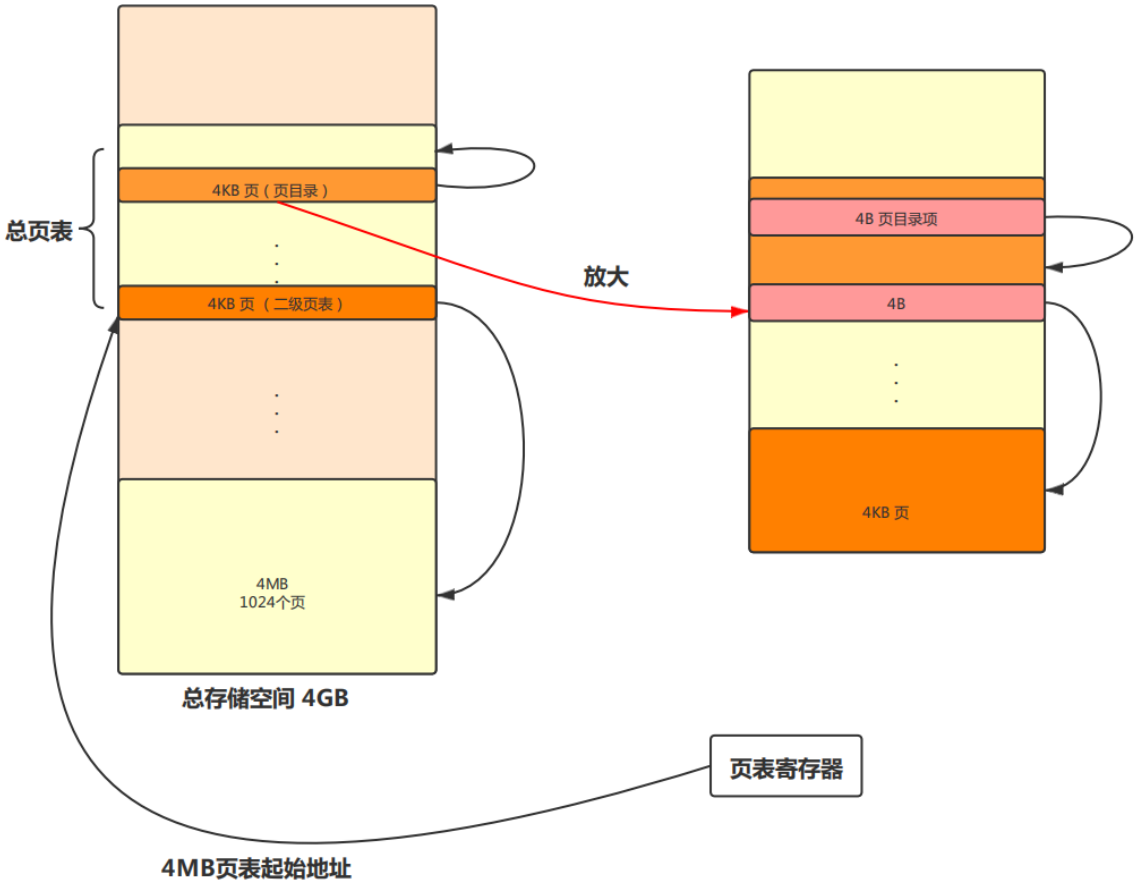
```
 o       UTEXT    -----> +----------------------------+
|
 o                       |                            |      2 * PDMAP
\|/
 a      0 ------------>  +----------------------------+ ------------------------
----
 o
*/
```

> 虚拟存储与页表自映射

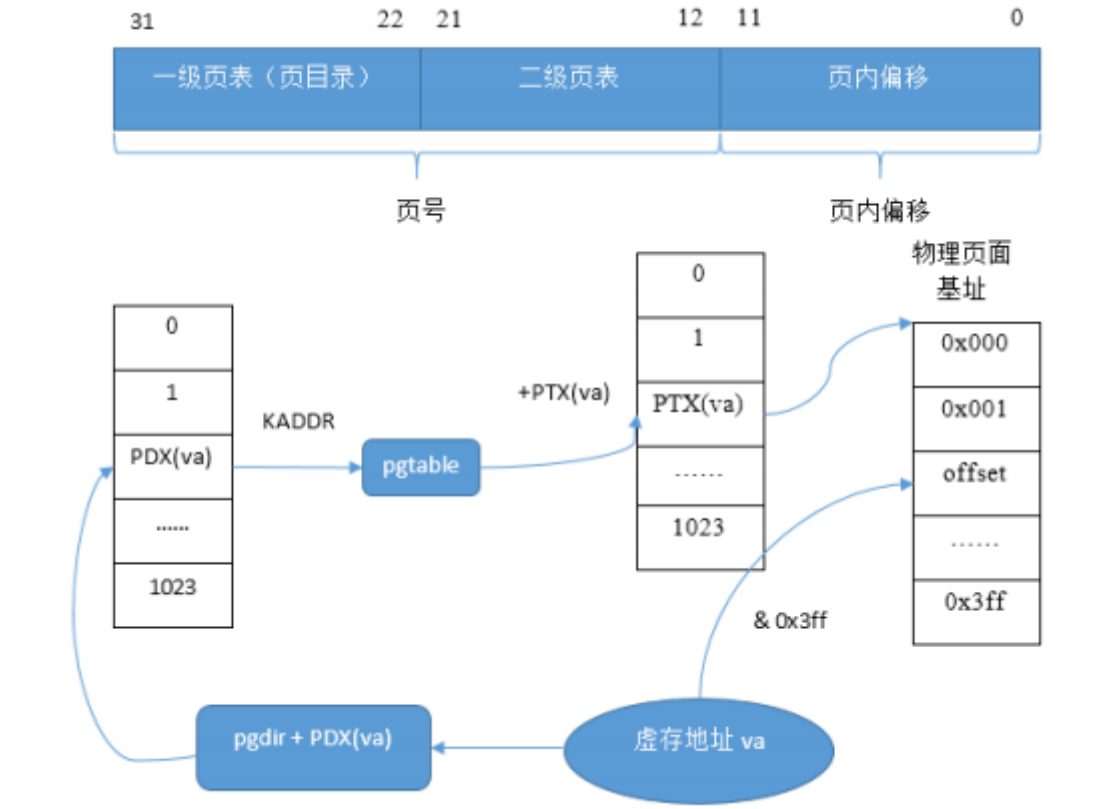
在本次实验的虚拟内存管理中，难点在于页表的映射。页表的自映射机制可以参考函数不动点来理解：

线性函数$f(x)$将$[a,b]$线性映射到$[a',b']$，则$(x-a)/(b-a)=(f(x)-a')/(b'-a')$

自映射机制可以通过如下图示加以说明：



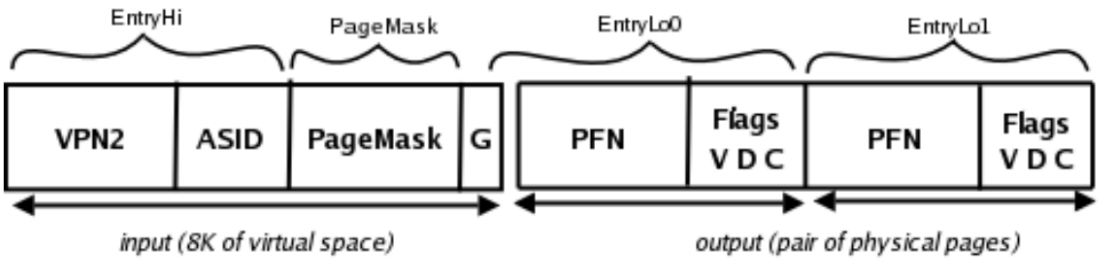
二级页表映射与查询机制在实验指导书中已经表示地非常明确了。



> TLB与TLB指令

TLB不是按照索引访问的，而是通过内容比较进行访问，所以硬件需要完成比较的功能，TLB比较小。

TLB数据项结构如下：



一般的TLB表项为一对二的形式，一个虚拟地址对应相邻的两个物理页框号，Entryhi和表项的高位对应，EntryLo0和EntryLo1分别对应表项的地位，各存储了一个物理页框号和允许位。

TLBWI，将EntryHi,EntryLo0,EntryLo1的内容写到由Index寄存器索引的TLB表项中。

TLBP按照内容查找，找到和Entryhi内容相同的TLB表项，将索引写到index寄存器。这个操作比较慢，所以TLBP指令后面要跟4个nop。

体会与感想

本节的一大难点在于神奇链表的理解，宏定义与特殊的指针是最难的部分，通过参考网络上的讲解并与同学深入讨论，逐渐理解了这个神奇的链表。链表其实并不是特别复杂，但是写得特别精妙。读一份好的代码犹如读一本好书，这一次的lab代码每读一遍都能够有新的收获，从中学习有工业化特色的代码。

页表的自映射机制比较琐碎凌乱，但是精髓并不是很复杂，类比函数自映射机制可以轻松推导出页表自映射中计算公式。TLB指令从未接触过，花费了很多时间了解。

lab2OS代码上贯下连性非常强，要多读代码对各种函数有较强的敏感度，在阅读时才能够减少障碍，快速定位到需要看的函数，通过不断阅读记住其中的细节，慢慢地才能够熟练应用。

对指导书理解的还是不是很透彻，一直以为我们的页表也是自映射的，其实代码中并没有建立自映射的语句，所以不是自映射的。在自己研读指导书的基础上，多和老师同学们沟通交流，才是学好OS的好方法。

指导书反馈

建议指导书专门开辟一部分讲一讲TLB访存原理，对CP0协处理寄存器的使用进行一下讲解。

可以将同学们容易混淆或者理解错误的地方标红或加粗一下。

残留难点

关于TLB使用的汇编代码仍然不是很熟练，对各个CP0寄存器的功能不熟，还需要进一步看MIPS手册与《See mips run linux》对MIPS体系结构进一步了解。