OS —— Lab2实验报告

22373407 王飞阳

一、思考题

Thinking 2.1

1. 虚拟地址
2. 虚拟地址

Thinking 2.2

1. 宏允许我们定义一次链表操作，从而使用同一套宏可以实现多个链表的管理。这消除了重复编写相同功能的需求，并有助于减少错误和提高代码的一致性。同时，可以根据需要轻松修改和定制。如果需要添加新的链表操作或修改现有操作的行为，只需更改宏的定义即可，而无需修改整个代码库。
2. 性能比较：

- 单向链表：

  插入操作较为简单，不用维护 `le_prev` 指针，但是性能差异不大

  删除操作单向链表较为复杂，时间复杂度是 $O(n)$,而双向链表的时间复杂度为 $O(1)$

- 循环链表：

  插入操作简单，可在 $O(1)$ 的时间复杂度插入链表尾部，而双向链表是时间复杂度是 $O(n)$

  删除操作循环链表与单向链表相同，时间复杂度都是 $O(n)$

Thinking 2.3

选C：

```
struct Page_list{
    struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;
        u_short pp_ref;
    }* lh_first;
}
```

Thinking 2.4

1.ASID

在多进程操作系统中，由于存在共享虚拟内存的情况，同一虚拟地址可能会映射到不同的物理地址，具体取决于进程的页表设置。因此，为了确保在TLB查找过程中正确地将虚拟地址映射到正确的物理地址，必须引入ASID。

ASID 作为虚拟地址的附加标识符，确保了在TLB查找过程中，不同进程的同一虚拟地址可以映射到各自正确的物理地址。通过将虚拟地址和对应的ASID一起传递给TLB，系统可以准确地找到正确的物理地址。因此，ASID在这种场景下是不可或缺的，它是多进程操作系统中实现地址空间隔离和共享的重要机制之一。

2.

根据文档中的信息，Section 3.3.1 提到了 ASID 字段的位数。假设 ASID 字段的位数为 n。在这种情况下，ASID 字段可以表示 2^n 个不同的地址空间。因此，4Kc 中可以容纳的最大地址空间的数量为 2^n。

由于EntryHi寄存器中ASID值共8位，因此最多可以容纳256个不同的地址空间。

Thinking 2.5

1.

当`tlb_invalidate`需要删除特定虚拟地址在 TLB 中的旧表项时，`tlb_invalidate`调用`tlb_out`实现该功能。

2.

根据当前进程的状态，以虚拟地址和进程的`ASID`值为参数调用`tlb_out`完成对`TLB`对应旧表项的清除。

3.

```
LEAF(tlb_out)
.set noreorder
    mfc0    t0, CP0_ENTRYHI     // $t0 = $EntryHi，暂时保存原EntryHi寄存器值
    mtc0    a0, CP0_ENTRYHI     // $EntryHi = $a0，此时a0为调用tlb_out时传入的参数,将
Key写入EntryHi
    nop                         // 空操作,以解决数据冒险
    tlbp                        // tlbp，根据EntryHi寄存器中的值(key)查找对应的表项，并
存入Index寄存器
    nop                         // 空操作,以解决数据冒险
    mfc0    t1, CP0_INDEX       // $t1 = $Index
.set reorder
    bltz    t1, NO_SUCH_ENTRY   //if $t1 < 0，即Index寄存器最高位为1，代表未找到相应
页，则跳转到'NO_SUCH_ENTRY'
.set noreorder
    mtc0    zero, CP0_ENTRYHI   // $EntryHi = 0
    mtc0    zero, CP0_ENTRYLO0  // $ENTRYLO0 = 0
    mtc0    zero, CP0_ENTRYLO1  // $ENTRYLO1 = 0
    nop                         // 空操作,以解决数据冒险
    tlbwi                       //将EntryHi和EntryLo0、EntryLo1中的值写入索引指定的表
项。此时旧表项被清零
.set reorder

NO_SUCH_ENTRY:
    mtc0    t0, CP0_ENTRYHI     //$EntryHi = $t0，将原EntryHi寄存器值写回
    j       ra                  //结束函数，返回
END(tlb_out)
```

Thinking 2.6

X86体系结构中的内存管理机制采用了分段和分页的组合方式。在分段机制下，内存被分为不同的段，每个段有自己的基地址和长度，并且可以拥有不同的访问权限。而在分页机制下，内存被划分为固定大小的页，通常为4KB，每个页都有自己的物理地址和虚拟地址。X86中使用了基于页表的分页机制，通过页表将虚拟地址映射到物理地址，实现了虚拟内存的概念。X86还支持多级页表，以便管理大量的内存。

相比之下，MIPS体系结构中的内存管理机制更加简单直接，主要采用分页方式。MIPS使用固定大小的页来管理内存，通常也是4KB。MIPS中的页表是一种二级结构，包含了页表和页表项。通过页表项将虚拟地址映射到物理地址，实现了虚拟内存的功能。MIPS中的TLB（Translation Lookaside Buffer）用于加速地址转换过程，类似于X86中的页表缓存。

在内存管理上，X86和MIPS的主要区别在于采用的机制和细节实现。X86采用了复杂的分段和分页结合的机制，支持多级页表和特权级别的访问控制，使得其在处理复杂内存场景时更为灵活。而MIPS则相对简洁，主要采用分页方式，虽然实现起来更加直接，但在某些高级功能上可能相对不足。两者在性能和适用场景上也有所差异，具体取决于应用的需求和实现的细节。

Thinking A.1

1.

一个页面大小4KB，则可将512GB的空间划分为128M个4KB页。则题目中的第三级页表的基地址 PT_{base} 应该位于第 $PT_{base} >> 12$ 个页表项。而每个页表项占用8字节，则该页表项对于地址的偏移为 $PT_{base} >> 12 << 3$ ，故三级页表页目录的基地址为 $PT_{base} + (PT_{base} >> 12 << 3)$

2.

自映射项通常位于页表的最后一个项，以方便计算和访问。

故其地址为：$PT_{base} + (PT_{base} >> 12 << 3) + 512 \times 4KB$

二、难点分析

物理内存管理

页式管理系统

MOS结构中采取页式管理结构，将物理内存总 `memsize` 按4KB一页分成 `npage` 页。

在物理内存管理中，通过 `pages` 结构体数组对这 `npage` 页进行管理。

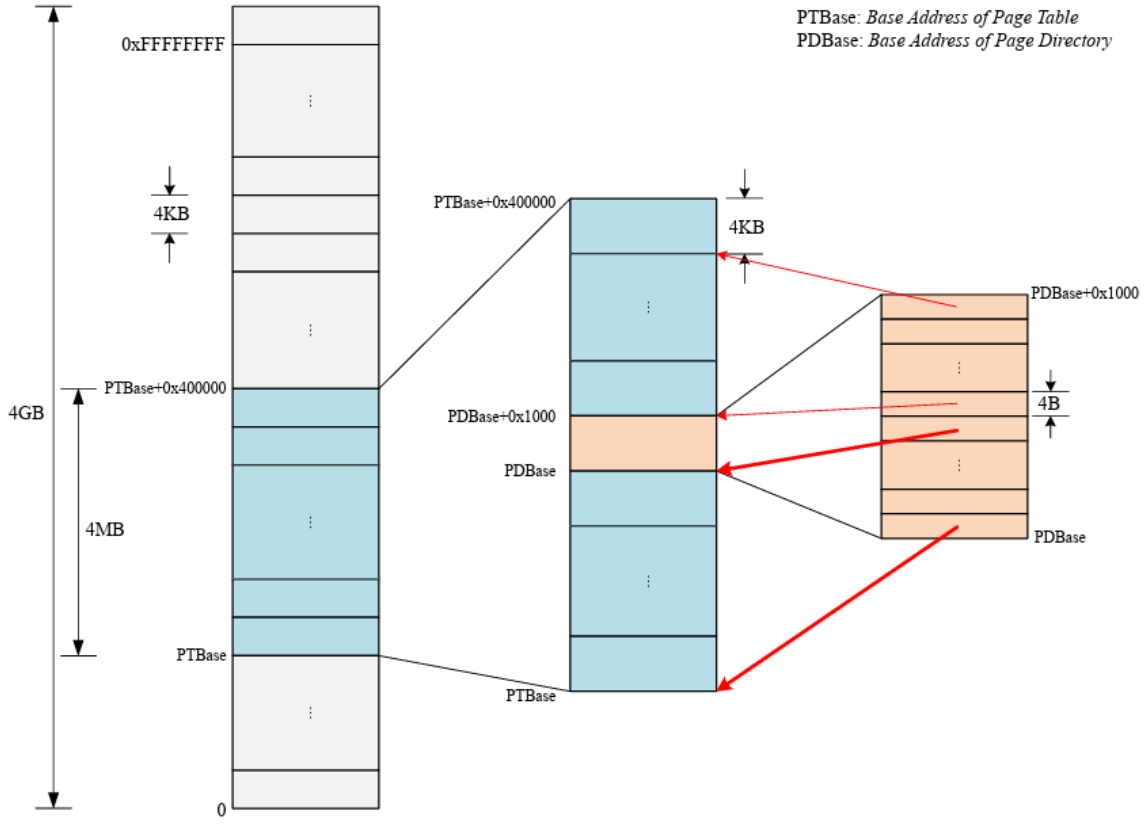
在 `Page` 结构体中， `pp_ref` 记录了该物理内被映射的次数， `pp_link` 是为了后续采取链表结构管理空闲链表所定义。每个 `Page` 结构体对应的物理页并不需要记录，因为结构体与物理页一一对应，通过 `(p - pages)<<12` 即可计算得到。详细过程主要使用到了以下函数：

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

static inline u_long page2pa(struct Page *pp) {
    return page2ppn(pp) << PGSHIFT;
}

static inline struct Page *pa2page(u_long pa) {
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa: %x", pa);
```

```
    }
    return &pages[PPN(pa)];
}

static inline u_long page2kva(struct Page *pp) {
    return KADDR(page2pa(pp));
}
```

页目录自映射

这部分作为理论课内容，简单用示意图来展示二级页表的自映射：



链表法管理空闲物理页框

在此部分中，我们通过使用以下宏定义，实现了一个双向链表

- LIST_EMPTY(head)，判断 head 指针指向的头部结构体对应的链表是否为空。
- LIST_FIRST(head)，将返回 head 对应的链表的首个元素。
- LIST_INIT(head)，将 head 对应的链表初始化。
- LIST_NEXT(elm, field)，返回指针 elm 指向的元素在对应链表中的下一个元素的指针。 elm 指针指向的结构体需要包含一个名为field的字段，类型是一个链表项LIST_ENTRY(type)， 下面出现的field含义均和此相同。
- LIST_INSERT_AFTER(listelm, elm, field)，将 elm 插到已有元素 listelm 之后。
- LIST_INSERT_BEFORE(listelm, elm, field)，将 elm 插到已有元素 listelm 之前。
- LIST_INSERT_HEAD(head, elm, field)，将 elm 插到 head 对应链表的头部。
- LIST_REMOVE(elm, field)，将 elm 从对应链表中删除

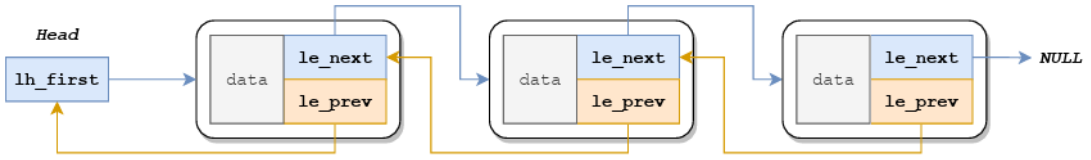


图 2.2: macro list

通过转换以后可以很容易得到用于管理空闲链表的结构体定义如下：

```
struct Page_list{
    struct Page {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;
        u_short pp_ref;
    }* lh_first;
}
static Page_list free_page_list;
```

页表管理函数

主要是`page_init(), page_alloc(), page_free()`三个函数，在理解和编写完宏函数之后，我们的页表管理也正式建立起来。

虚拟内存管理

CPU 发出地址

在实际程序中，访存、跳转等指令以及用于取指的PC寄存器中的访存目标地 址都是虚拟地址。我们编写的C程序中也经常通过对指针解引用来进行访存，其中指针的值也 会被视为虚拟地址，经过编译后生成相应的访存指令。

虚拟地址映射到物理地址的方式

在4Kc 上，软件访存的虚拟地址会先被MMU硬件映射到物理地址，随后使用物理地址来访问内存或其他外设。与实验相关的映射与寻址规则（内存布局）如下：

- 若虚拟地址处于0x80000000~0x9fffffff (kseg0)，则将虚拟地址的最高位置0得到物理 地址，通过 cache 访存。这一部分用于存放内核代码与数据。
- 若虚拟地址处于0xa0000000~0xbfffffff (kseg1)，则将虚拟地址的最高 3 位置 0 得到 物理地址，不通过cache 访存。这一部分可以用于访问外设。
- 若虚拟地址处于0x00000000~0x7fffffff (kuseg)，则需要通过 TLB 转换成物理地址， 再通过cache 访存。这一部分用于存放用户程序代码与数据

MMU采用硬件TLB来完成地址映射。TLB 需要由软件进行填写，即操作系统内核负责维护TLB中的数据。所有对低2GB空间（kuseg） 的内存访问操作都需要经过TLB。

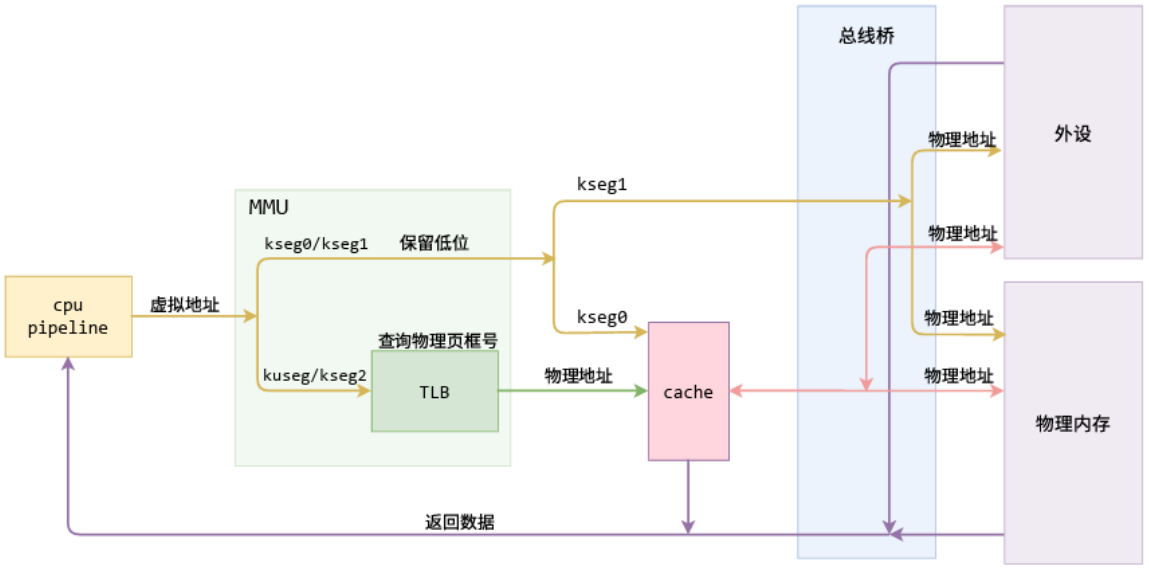


图 2.1: cpu-tlb-memory 关系

kseg0

对于处于 `kseg0` 内的虚拟地址，我们可以直接用如下两个宏来访问对应物理地址：

```
#define PADDR(kva)
                  \
    ({
              \
        u_long _a = (u_long)(kva);
          \
        if (_a < ULIM)
          \
            panic("PADDR called with invalid kva %08lx", _a);
     \
        _a - ULIM;
          \
    })

// translates from physical address to kernel virtual address
#define KADDR(pa)
                 \
    ({
              \
        u_long _ppn = PPN(pa);
          \
        if (_ppn >= npage) {
          \
            panic("KADDR called with invalid pa %08lx", (u_long)pa);
      \
        }
         \
        (pa) + ULIM;
          \
    })
```

从而实现从虚拟地址到物理地址，从物理地址到虚拟地址的双向映射

kuseg

而对于`kuseg`段，需要填写二级页表，并通过`TLB`来访问物理地址。

二级页表的结构如下：

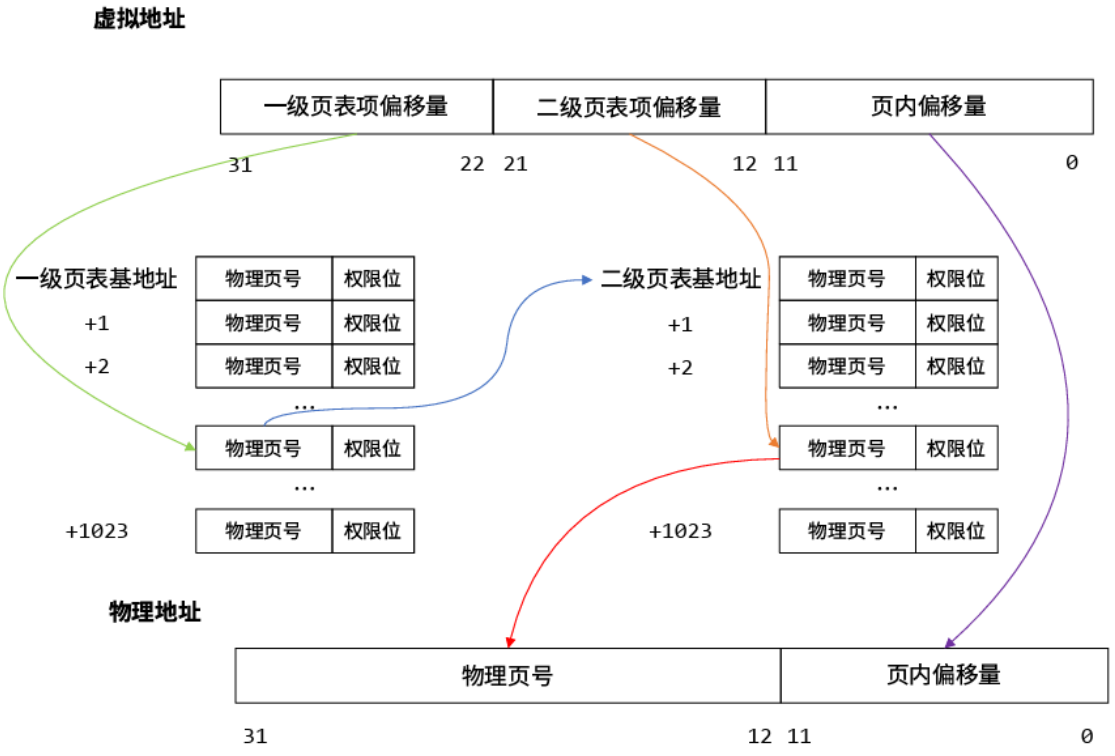


图 2.3: 两级页表结构的地址变换机制

而在实验中，二级页表的填写过程如下：

- 计算一级页目录入口`pgdir_entryp = pgdir + PDX(va)`
- 由一级页目录入口获得二级页表基地址`pgtable = (Pte *)KADDR(PTE_ADDR(*pgdir_entryp))`
- 计算二级页表入口`pgtable_entry = pgtable + PTX(va)`
- 为二级页表入口设置对应的物理地址页号`*pgtable_entry = PPN(pa) or *pgtable_entry = PPN(page2pa(page))`

具体流程图如下：

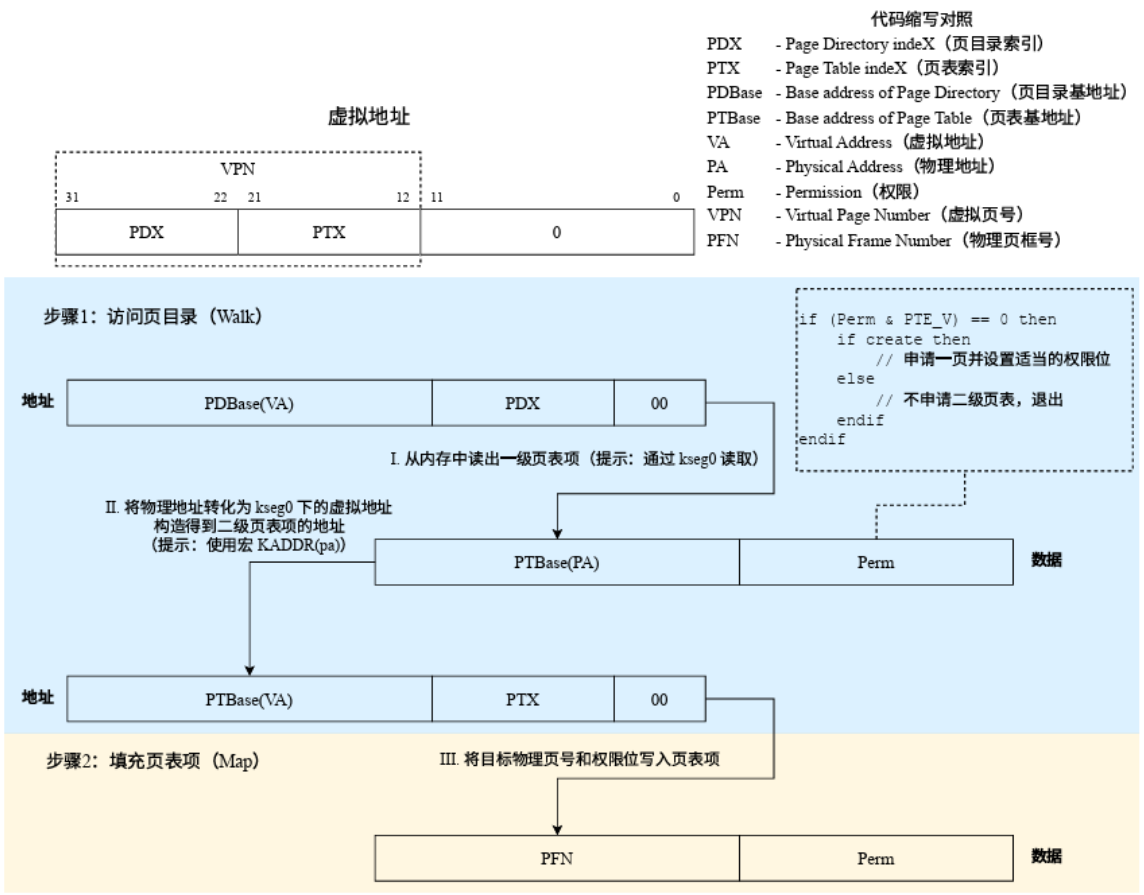
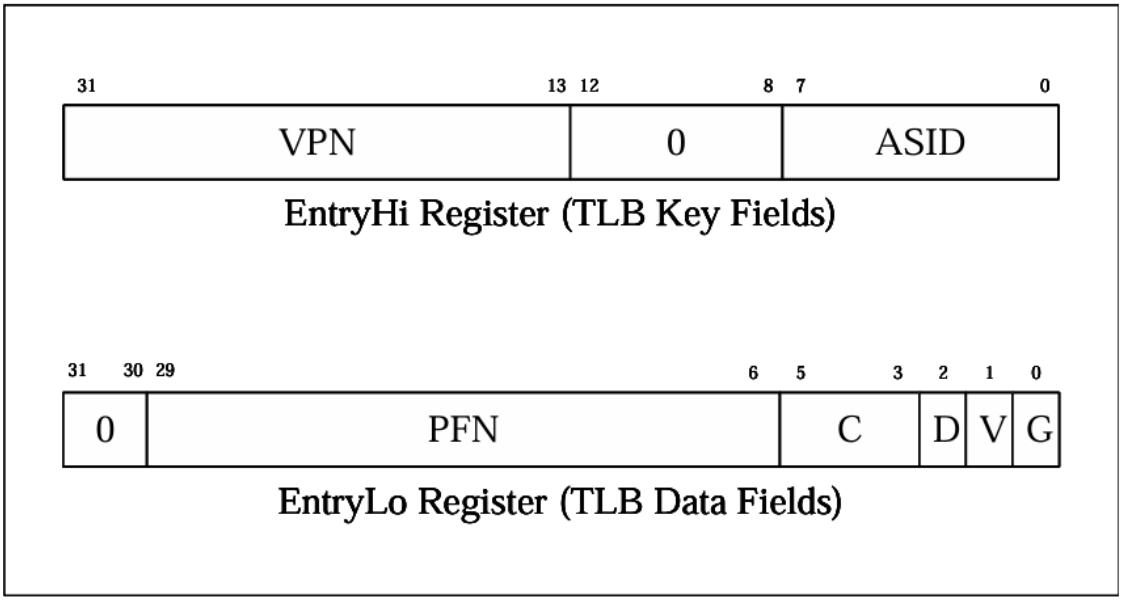


图 2.4: walk & map(insert) 系列函数图解流程

TLB

所有的在低 2GB 空间的访存操作都需要经过 TLB。

每个TLB表项都有两个组成部分，包括一组Key和两组Data。其中 EntryLo0、EntryLo1 拥有完全相同的位结构，EntryLo0 存储 Key 对应的偶页而 EntryLo1 存储 Key 对应的奇页，而EntryHi存储 Key ，包括了VPN和ASID



TLB 事实上构建了一个映射 `TLB ---> < PFN,N,D,V,G >`。

实验体会

本次实验相交于lab1，无论是实验难度还是代码量、阅读量都有所提升。

尤其是对于页表式管理以及物理内存和虚拟的内存的理解还仍不够到位，需要下来再自己多通过学习和查询资料，加深对这部分知识的理解，也有利于之后实验的开展。