Lab2

> 姓名：文柳懿
> 学号：21351002

Thinking

Thinking 2.1

C程序中，指针变量存储的地址是虚拟地址；`lw`,`sw`指令使用的地址也是虚拟地址。
硬件(MMU)将这些虚拟地址转为物理地址，以便访问物理内存。

Thinking 2.2

1.用宏实现链表的好处：

- **代码复用性增强**：宏是预处理器指令，在编译前插入代码。因此可在多个地方复用。
- **提高性能**：宏在编译时展开，在运行时没有额外的函数调用开销。
- **跨平台兼容性**：宏是C/C++等编程语言的一部分，在Window、Linux等操作系统上，只要编译器支持宏，就可以使用宏操作链表。

2.不同链表的比较：

- 头部结构体定义比较：

```
//单向链表：
#define SLIST_HEAD(name, type)                                          \
struct name {                                                           \
        struct type *slh_first; /* first element */                     \
}

//双向链表：
#define LIST_HEAD(name, type)                                           \
struct name {                                                           \
        struct type *lh_first;  /* first element */                     \
}

//循环链表：
#define CIRCLEQ_HEAD(name, type)                                        \
```

```
struct name {                                                   \
        struct type *cqh_first;         /* first element */     \
        struct type *cqh_last;          /* last element */      \
}
```

- 链表项结构体定义比较：

```
//单向链表：链表项包含一个指向下一个元素的指针sle_next.
#define SLIST_ENTRY(type)                                       \
struct {                                                        \
        struct type *sle_next;  /* next element */              \
}

//双向链表：链表项包含指向下一个元素的指针le_next,和指向前一个链表项le_next的指针`le_prev`.
#define LIST_ENTRY(type)                                        \
struct {                                                        \
        struct type *le_next;   /* next element */              \
        struct type **le_prev;  /* address of previous next element */  \
}

//循环链表：链表项包含指向前一个和下一个元素的指针cqe_prev,cqe_next.
#define CIRCLEQ_ENTRY(type)                                     \
struct {                                                        \
        struct type *cqe_next;          /* next element */      \
        struct type *cqe_prev;          /* previous element */  \
}
```

- 单向链表：

```
//只能在指定节点后插入，不能在之前插入
#define SLIST_INSERT_AFTER(slistelm, elm, field) do {           \
        (elm)->field.sle_next = (slistelm)->field.sle_next;     \
        (slistelm)->field.sle_next = (elm);                     \
} while (/*CONSTCOND*/0)

//移除：
#define SLIST_REMOVE(head, elm, type, field) do {               \
        //判断是否为头节点
        if ((head)->slh_first == (elm)) {                       \
                SLIST_REMOVE_HEAD((head), field);               \
        }                                                       \
        else {                                                  \
                struct type *curelm = (head)->slh_first;        \
        //要删除指定节点时，需从头节点开始遍历，找到该节点的前序节点
                while(curelm->field.sle_next != (elm))          \
                        curelm = curelm->field.sle_next;        \
```

```
                curelm->field.sle_next =                                \
                    curelm->field.sle_next->field.sle_next;             \
        }                                                               \
} while (/*CONSTCOND*/0)
```

- 循环链表：链表项包含指向前一个和下一个元素的指针`cqe_prev`,`cqe_next`.

```
//节点后插入：
#define CIRCLEQ_INSERT_AFTER(head,listelm,elm,field) do {
    (elm)->field.cqe_next=(listelm)->field.cqe_next;
    (elm)->field.cqe_prev = (listelm);
    //注意listelm是否为尾节点：若是，更新尾节点为elm.
    if ((listelm)->field.cqe_next == (void *)(head))                \
            (head)->cqh_last = (elm);                               \
    else (listelm)->field.cqe_next->field.cqe_prev = (elm);         \
    (listelm)->field.cqe_next = (elm);
}while (/*CONSTCOND*/0)

//节点前插入：
#define CIRCLEQ_INSERT_BEFORE(head, listelm, elm, field) do {           \
        (elm)->field.cqe_next = (listelm);                              \
        (elm)->field.cqe_prev = (listelm)->field.cqe_prev;              \
        //若listelm为头节点：若是，更新头节点为elm.
        if ((listelm)->field.cqe_prev == (void *)(head))                \
                (head)->cqh_first = (elm);                              \
        else                                                            \
                (listelm)->field.cqe_prev->field.cqe_next = (elm);      \
        (listelm)->field.cqe_prev = (elm);                              \
} while (/*CONSTCOND*/0)

//移除：
#define CIRCLEQ_REMOVE(head, elm, field) do {                           \
        //若被移除元素elm是尾节点：
        if ((elm)->field.cqe_next == (void *)(head))                    \
                (head)->cqh_last = (elm)->field.cqe_prev;               \
        else                                                            \
                (elm)->field.cqe_next->field.cqe_prev =                 \
                    (elm)->field.cqe_prev;                              \
        //若被移除元素elm是头节点：
        if ((elm)->field.cqe_prev == (void *)(head))                    \
                (head)->cqh_first = (elm)->field.cqe_next;              \
        else                                                            \
                (elm)->field.cqe_prev->field.cqe_next =                 \
                    (elm)->field.cqe_next;                              \
} while (/*CONSTCOND*/0)
```

- 双向链表：链表项包含指向下一个元素的指针`struct type *le_next`,和指向前一个链表项

`struct type **le_next` 的指针 `le_prev`.

性能比较：

	单向链表	循环链表	双向链表
链表头部插入	O(1)	O(1)	O(1)
链表尾部插入	O(n)	O(1)	O(n)
指定位置插入	O(n)	O(n)	O(n)
删除头节点	O(1)	O(1)	O(1)
删除尾节点	O(n)	O(1)	O(n)
删除指定节点	O(n)	O(n)	O(n)

Thinking 2.3

`npage` 个 `Page` 和 `npage` 个物理页面一一对应。

```
struct Page_list{
    struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;      //双向链表项
        u_short pp_ref;     //当前物理页的引用次数：多少虚拟页映射至该物理页
    }* lh_first;
}
```

Thinking 2.4

1.ASID的必要性：
ASID(Address Space Identifier)是地址空间标识符，用于在多进程操作系统中标识不同进程的虚拟地址空间。

- 隔离不同进程的地址空间：在多进程环境中，每个进程有相互独立的虚拟地址空间，通过ASID进行隔离；
- 避免TLB污染：当多个进程共享同一个TLB时，若无ASID区分，可能出现一个进程的地址翻译结果覆盖另一个进程的翻译结果的情况，即所谓的“TLB污染”，导致错误的内存访问。通过

ASID，操作系统可以将每个进程的地址翻译结果分别存储在TLB中，避免了这种污染。

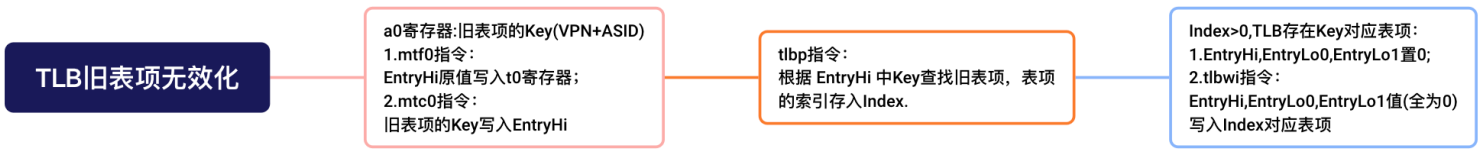
- 优化性能：当进程切换时，操作系统可以通过ASID，清除或标记与当前进程无关的TLB条目，确保当前进程快速获取其地址翻译结果。

2. `4Kc` 可容纳不同的地址空间的最大数量：
假设 `ASID` 的位数为 `n` ,其可表示的不同地址空间的最大数量为 `2^n` .
`4Kc` 中， `ASID` 位数为8，因此最多可以容纳2^8=256个不同地址空间。

Thinking 2.5

1. `tlb_invalidate` 调用 `tlb_out` .
2. `tlb_invalidate` 的作用：

```
//生成一个从 l（低位）到 h（高位）的位掩码：
#define GENMASK(h, l) (((~0UL) << (l)) & (~0UL >> (BITS_PER_LONG - 1 - (h))))
#define NASID 256
 //TLB旧表项无效化:删除进程asid对应的虚拟地址va,在TLB中的旧表项
void tlb_invalidate(u_int asid, u_long va) {
    //(va & ~GENMASK(PGSHIFT, 0)) :将va的低PGSHIFT位清零
    //(NASID - 1)生成一个位掩码，用于确保asid的值在有效范围内(0~255)
    //新的Entryhi值：VPN+ASID
        tlb_out((va & ~GENMASK(PGSHIFT, 0)) | (asid & (NASID - 1)));
}
```



`tlb_out` 汇编代码：

```
LEAF(tlb_out)       //声明叶子函数
.set noreorder      //关闭MIPS汇编器的指令重排优化，确保指令按特定顺序执行
    mfc0    t0, CP0_ENTRYHI     //从CP0的ENTRYHI寄存器，读取值到t0寄存器
    mtc0    a0, CP0_ENTRYHI     //将a0（旧表项的Key：即VPN+ASID）写入ENTRYHI寄存器
    nop
    /* Step 1: Use 'tlbp' to probe TLB entry */
    /* Exercise 2.8: Your code here. (1/2) */
    tlbp        //根据 EntryHi 中的 Key 查找对应的旧表项，将表项的索引存入 Index

    nop
    /* Step 2: Fetch the probe result from CP0.Index */
    mfc0    t1, CP0_INDEX       //从CP0_INDEX寄存器读取索引至t1
```

```
.set reorder
    bltz    t1, NO_SUCH_ENTRY  //t1<0,跳转至NO_SUCH_ENTRY标签处
.set noreorder
    //t1>0(TLB中存在Key对应的表项),清除TLB条目:
    //将CP0_ENTRYHI,CP0_ENTRYLO0,CP0_ENTRYLO1置0.
    mtc0    zero, CP0_ENTRYHI
    mtc0    zero, CP0_ENTRYLO0
    mtc0    zero, CP0_ENTRYLO1
    nop
    /* Step 3: Use 'tlbwi' to write CP0.EntryHi/Lo into TLB at CP0.Index  */
    /* Exercise 2.8: Your code here. (2/2) */
    tlbwi      //将ENTRYHI,ENTRYLO,ENTRYL1寄存器的内容(全为0)，写入到由CP0_INDEX指定的
TLB条目中

.set reorder

NO_SUCH_ENTRY:
    mtc0    t0, CP0_ENTRYHI    //t0（原始ENTRYHI值）写回ENTRYHI寄存器
    j       ra              //跳转回ra（返回地址）寄存器指定的地址
END(tlb_out)
```

Exercise

Exercise 2.3: `page_init` 函数

```
void page_init(void) {
        /* Step 1: Initialize page_free_list. */
        /* Hint: Use macro `LIST_INIT` defined in include/queue.h. */
        /* Exercise 2.3: Your code here. (1/4) */
        LIST_INIT(&page_free_list);

        /* Step 2: Align `freemem` up to multiple of PAGE_SIZE. */
        /* Exercise 2.3: Your code here. (2/4) */
        freemem=ROUND(freemem, PAGE_SIZE);

        /* Step 3: Mark all memory below `freemem` as used (set `pp_ref` to 1) */
        /* Exercise 2.3: Your code here. (3/4) */
        int index=(freemem-KSEG0)/PAGE_SIZE;
        while(index--) pages[index].pp_ref=1;

        /* Step 4: Mark the other memory as free. */
        /* Exercise 2.3: Your code here. (4/4) */
        for(index=(freemem-KSEG0)/PAGE_SIZE;index<npage;index++){
            pages[index].pp_ref=0;
```

```
            LIST_INSERT_HEAD(&page_free_list,&pages[index],pp_link);
        }
}
```

- 1.链表宏 `LIST_INIT` 初始化 `page_free_list`：
  由定义，`LIST_INIT(head)` 即 `(head)->lh_first = NULL`.
  有：

```
struct Page_list{
     struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;      //双向链表项
        u_short pp_ref;     //当前物理页的引用次数：多少虚拟页映射至该物理页
     }* lh_first;
}
struct Page_list page_free_list;
```

因此填入：`LIST_INIT(&page_free_list);`,对 `page_free_list` 取址，满足 `head` 为指针的要求。

- 2. `freemem` 按页对齐
- 3.将 `freemem` 以下页面对应的页控制块中的 `pp_ref` 标为 1:
  有定义：

```
struct Page *pages;
```

`kseg0` 参与物理内存分配，`include/mmu.h` 中定义起始地址：

```
#define KSEG0 0x80000000U
```

`npage` 个 `Page` 和 `npage` 个物理页面一一顺序对应。
数组 `struct Page *pages` 存放 `Page` 结构体；首个 `Page` 的地址为 `P`，则 `P[i]` 对应从 `0` 开始计数的第 `i` 个物理页面。
将 `freemem` 以下的内存空间按 `PAGE_SIZE` 划分；逐个页控制块置1.

```
int index=(freemem-KSEG0)/PAGE_SIZE;
while(index--) pages[index].pp_ref=1;
```

- 4.将其它页面对应的页控制块中的 `pp_ref` 标为 0，插入空闲链表
  可分配的页面：`index~(npage-1)`.

宏`LIST_INSERT_HEAD`定义：

```
/*
* Remove the element "elm" from the list.
* The "field" name is the link element as above.
*/
#define LIST_INSERT_HEAD(head, elm, field)
   do{
       //elm插入head的头节点之前
       if ((LIST_NEXT((elm), field) = LIST_FIRST((head))) != NULL)
           LIST_FIRST((head))->field.le_prev = &LIST_NEXT((elm), field);
       LIST_FIRST((head)) = (elm);
       (elm)->field.le_prev = &LIST_FIRST((head));
   } while (0)
```

```
for(index=(freemem-KSEG0)/PAGE_SIZE;index<npage;index++){
       pages[index].pp_ref=0;
       LIST_INSERT_HEAD(&page_free_list,&pages[index],pp_link);
}
```

Exercise 2.4: `page_alloc`函数

```
int page_alloc(struct Page **new) {
        /* Step 1: Get a page from free memory. If fails, return the error code.*/
        struct Page *pp;
        /* Exercise 2.4: Your code here. (1/2) */
        //取出空闲块链表page_free_list头部的页控制块
        if(LIST_EMPTY(&page_free_list)) return -E_NO_MEM;//没有可用页，返回异常值
        pp=LIST_FIRST(&page_free_list);

        LIST_REMOVE(pp, pp_link);       //从空闲链表中移除

        /* Step 2: Initialize this page with zero.
         * Hint: use `memset`. */
        /* Exercise 2.4: Your code here. (2/2) */
        memset((void *)page2kva(pp),0,PAGE_SIZE);
        *new = pp;
        return 0;
}
```

- 1.取出空闲块链表`page_free_list`头部的页控制块，分配出去：
  `LIST_EMPTY`定义：

- 2.清空此页中的数据：`memset`函数
  找到该页(`pp`)物理地址：`page2kva`函数定义：

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

其中，`KADDR(x)`将物理地址`x`翻译为`kseg0`中的虚拟地址：

```
// translates from physical address to kernel virtual address
#define KADDR(pa)(...)
```

使用`memset`清空数据：

```
memset((void *)page2kva(pp),0,PAGE_SIZE);   //(void*)转为字节类型
*new = pp;
```

Exercise 2.5: `page_free`函数

将`pp`指向的页控制块重新插入到`page_free_list`中.

```
void page_free(struct Page *pp) {
        assert(pp->pp_ref == 0);
        /* Just insert it into 'page_free_list'. */
        /* Exercise 2.5: Your code here. */
        LIST_INSERT_HEAD(&page_free_list,pp,pp_link);
}
```

`LIST_INSERT_HEAD`定义：

```
/*
 * Insert the element "elm" at the head of the list named "head".
 * The "field" name is the link element as above.
 */
#define LIST_INSERT_HEAD(head, elm, field)                              \   do{
\
    if ((LIST_NEXT((elm), field) = LIST_FIRST((head))) != NULL)         \
        LIST_FIRST((head))->field.le_prev = &LIST_NEXT((elm), field);   \
    LIST_FIRST((head)) = (elm);                                         \
    (elm)->field.le_prev = &LIST_FIRST((head));                         \
} while (0)
```

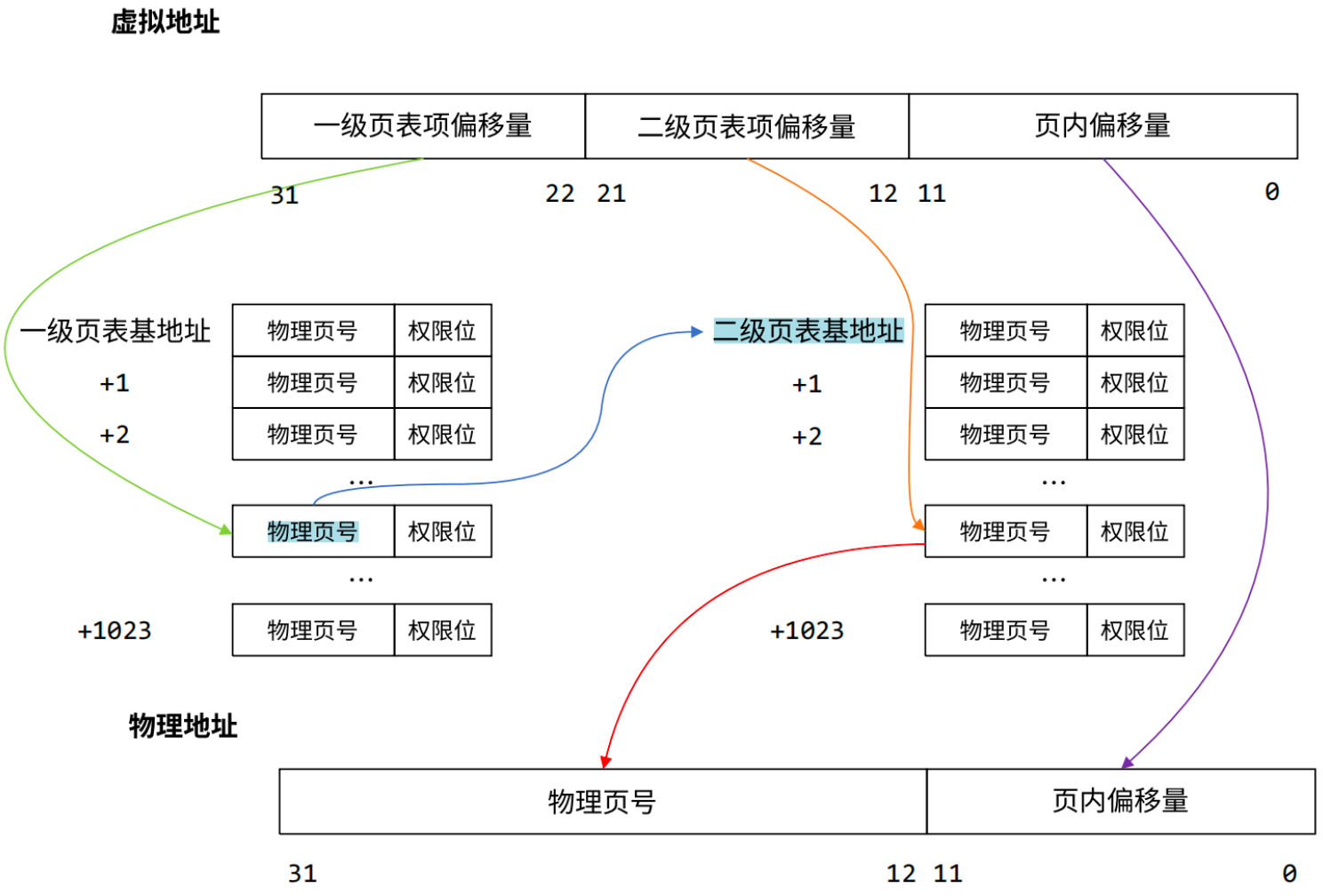
Exercise 2.6: `pgdir_walk` 函数

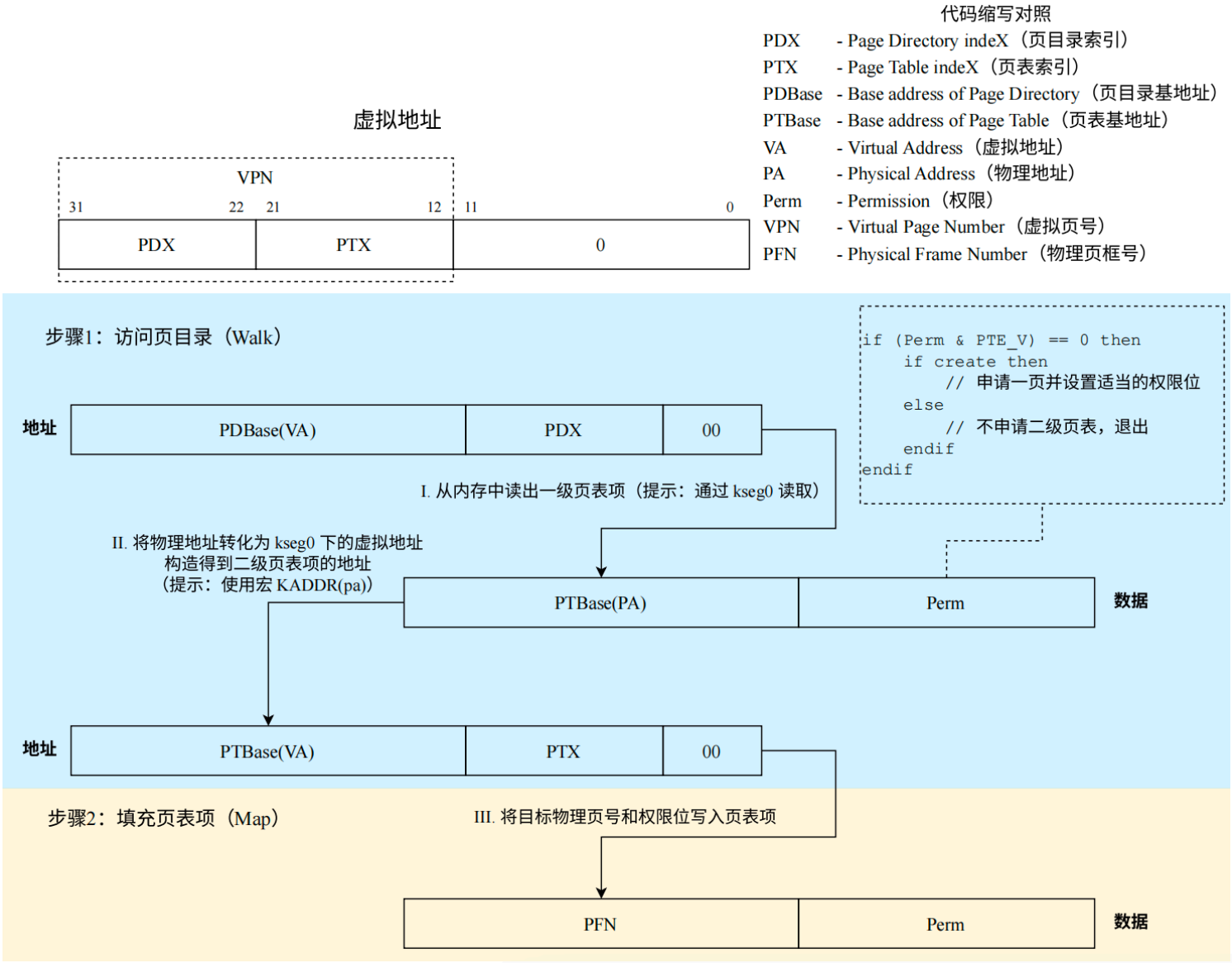
```
static int pgdir_walk(Pde *pgdir,u_long va,int create,Pte **ppte);
```

二级页表检索函数：给定一个虚拟地址`va`,在一级页表基地址`pgdir`指向的页目录中,查找该虚拟地址`va`对应的页表项,将其地址写入`*ppte`.
过程：找到一级页表项->读出物理页号，找到二级页表基地址->找到二级页表项

```
/* Overview:
 *   Given 'pgdir', a pointer to a page directory, 'pgdir_walk' returns a pointer
to the page table entry for virtual address 'va'.
 *
 * Pre-Condition:
 *   'pgdir' is a two-level page table structure.
 *   'ppte' is a valid pointer, i.e., it should NOT be NULL.
 *
 * Post-Condition:
 *   If we're out of memory, return -E_NO_MEM.
 *   Otherwise, we get the page table entry, store
 *   the value of page table entry to *ppte, and return 0, indicating success.
 *
 * Hint:
 *   We use a two-level pointer to store page table entry and return a state code
to indicate whether this function succeeds or not.
 */
//pgdir为指向页目录基地址的指针.
static int pgdir_walk(Pde *pgdir,u_long va,int create,Pte **ppte){
    Pde *pgdir_entryp;
    struct Page *pp;

    //1.pgdir_entryp指向va对应的一级页表项:
    //pgdir为一级页表基地址，PDX(va)为一级页表偏移量
    //*pgdir_entryp为页表项内容(32位：20位物理页号+12位权限位)
    pgdir_entryp=pgdir+PDX(va);

    //2.处理对应一级页表项无效的情况:
    //int page_alloc(struct Page **new)     //申请空闲的物理页块，其地址存入new.
    if(!(*pgdir_entryp&PTE_V)){     //对应二级页表不存在/有效位为0.
        if(create){
            if(page_alloc(&pp)==-E_NO_MEM){ //pp指向分配的空闲页控制块
                //分配空闲物理页失败，超出内存
                *ppte=NULL;
                return -E_NO_MEM;
            }
            pp->pp_ref++;   //申请到的物理页的引用次数+1
```

```
            //更新一级页表项pgdir_entryp:
            //page2pa(pp)获得:页控制块pp对应物理基地址
            *pgdir_entryp=page2pa(pp)|PTE_C_CACHEABLE | PTE_V;
        }else{
            *ppte=NULL;return 0;
        }
    }

    //3.将二级页表项的虚拟地址，写入*ppte
    //二级页表基地址：PTE_ADDR(*pgdir_entryp);PTX(va)为二级页表偏移量
    //(页表在内核kseg0中,用KADDR转为虚拟地址)
    *ppte=(Pte *)KADDR(PTE_ADDR(*pgdir_entryp)+PTX(va)*4);
    return 0;
}
```





Exercise 2.7: `pgdir_insert` 函数

```
int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int
perm);
```

增加地址映射函数：给定一个虚拟地址 `va` ,在一级页表基地址 `pgdir` 指向的页目录中,将虚拟地址 `va` 映射至页控制块 `pp` 对应的物理页面.

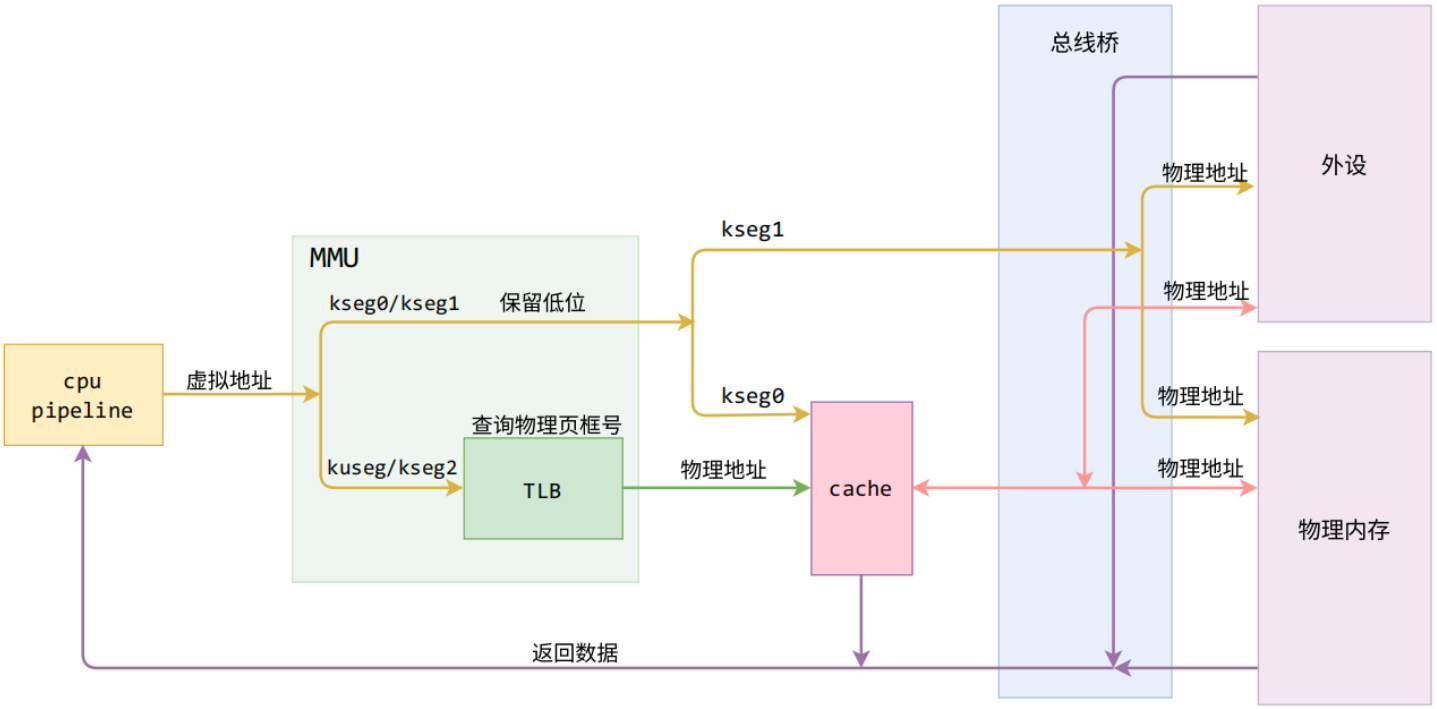
```
/* Lab 2 Key Code "tlb_invalidate" */
/* Overview:
 *   Invalidate the TLB entry with specified 'asid' and virtual address 'va'.
 *
 * Hint:
 *   Construct a new Entry HI and call 'tlb_out' to flush TLB.
 *   'tlb_out' is defined in mm/tlb_asm.S
```

```
 */
void tlb_invalidate(u_int asid, u_long va) {
        tlb_out((va & ~GENMASK(PGSHIFT, 0)) | (asid & (NASID - 1)));
}
```

```
int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm){
    Pte *pte;

    //1.获取va对应的二级页表项
    //[1]检查是否va已经存在映射?
    pgdir_walk(pgdir,va,0,&pte);//不存在,返回NULL;存在,为该虚拟地址对应的二级页表项
    if(pte!=NULL && (*pte & PTE_V)){    //存在且有效
    //[2]现有映射的页，和待插入的页，是否一致?
        //*(&pte)=pte里存储二级页表项的地址，pte为(Pte *)类型，*pte为二级页表项内容
        //pa2page()由二级页表项内容,找到pages中对应的页控制块
        if(pa2page(*pte)!=pp){
            page_remove(pgdir,asid,va); //不一样，移除现有映射
        }else{          //一样：更新映射的权限位
            tlb_invalidate(asid, va);       //(1)删掉TLB中缓存的页表项
            *pte = page2pa(pp) | perm | PTE_C_CACHEABLE | PTE_V;
            //(2)更新内存中的页表项：下次加载va所在页时，TLB从页表中加载该项
        }
    }

        /* Step 2: Flush TLB with 'tlb_invalidate'. */
        /* Exercise 2.7: Your code here. (1/3) */
        tlb_invalidate(asid,va);

        /* Step 3: Re-get or create the page table entry. */
        /* If failed to create, return the error. */
        /* Exercise 2.7: Your code here. (2/3) */
        //重新获取/申请va对应的二级页表项pte
        if(pgdir_walk(pgdir,va,1,&pte)==-E_NO_MEM){
                //create=1,允许申请新的页控制块,将va对应的二级页表项地址存入pte
                return -E_NO_MEM;
        }

        /* Step 4: Insert the page to the page table entry with 'perm |
PTE_C_CACHEABLE | PTE_V'
         * and increase its 'pp_ref'. */
        /* Exercise 2.7: Your code here. (3/3) */
        //填写二级页表项:物理页号(由新映射的pp页控制块决定)+权限位
        *pte=page2pa(pp)| perm | PTE_C_CACHEABLE | PTE_V;
        pp->pp_ref++;

        return 0;
}
```

Notes

虚拟地址->物理地址：



宏/函数定义：

`include/mmu.h`：

```
//一级页表偏移量:va的22~31位
#define PDSHIFT 22
#define PDX(va) ((((u_long)(va)) >> PDSHIFT) & 0x03FF)
//二级页表偏移量:va的12~21位
#define PGSHIFT 12
#define PTX(va) ((((u_long)(va)) >> PGSHIFT) & 0x03FF)

//页表项的物理页号部分:31~12位,也即:物理页基地址
#define PTE_ADDR(pte) (((u_long)(pte)) & ~0xFFF)
//页表项的权限位部分:11~0位
#define PTE_FLAGS(pte) (((u_long)(pte)) & 0xFFF)

//物理地址的物理页号:(右移12位)
#define PPN(pa) (((u_long)(pa)) >> PGSHIFT)
//逻辑地址的虚拟页号:(右移12位)
#define VPN(va) (((u_long)(va)) >> PGSHIFT)
```

```
//内核起始地址:
#define ULIM 0x80000000
//kseg0中虚拟地址x->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;                                              \
    })

//物理地址->kseg0处虚拟地址
#define KADDR(pa)                                                   \
    ({                                                              \
        u_long _ppn = PPN(pa);                                      \
        if (_ppn >= npage) {                                        \
            panic("KADDR called with invalid pa %08lx", (u_long)pa);    \
        }                                                           \
        (pa) + ULIM;                                                \
    })
```

include/pmap.h:

```
//页表
typedef LIST_ENTRY(Page) Page_LIST_entry_t;
struct Page_list{
    //页控制块:pp_link+pp_ref.
     struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;      //双向链表项
        /*即:Page_LIST_entry_t pp_link;*/
        u_short pp_ref;     //当前物理页的引用次数: 多少虚拟页映射至该物理页
     }* lh_first;
}
extern struct Page *pages;
extern struct Page_list page_free_list;

//页控制块pp的物理页号:
static inline u_long page2ppn(struct Page *pp) {
        return pp - pages;
}
//页控制块pp,对应物理页的基地址: (填充pte常用)
//获取物理页号page2ppn(pp),左移12位
```

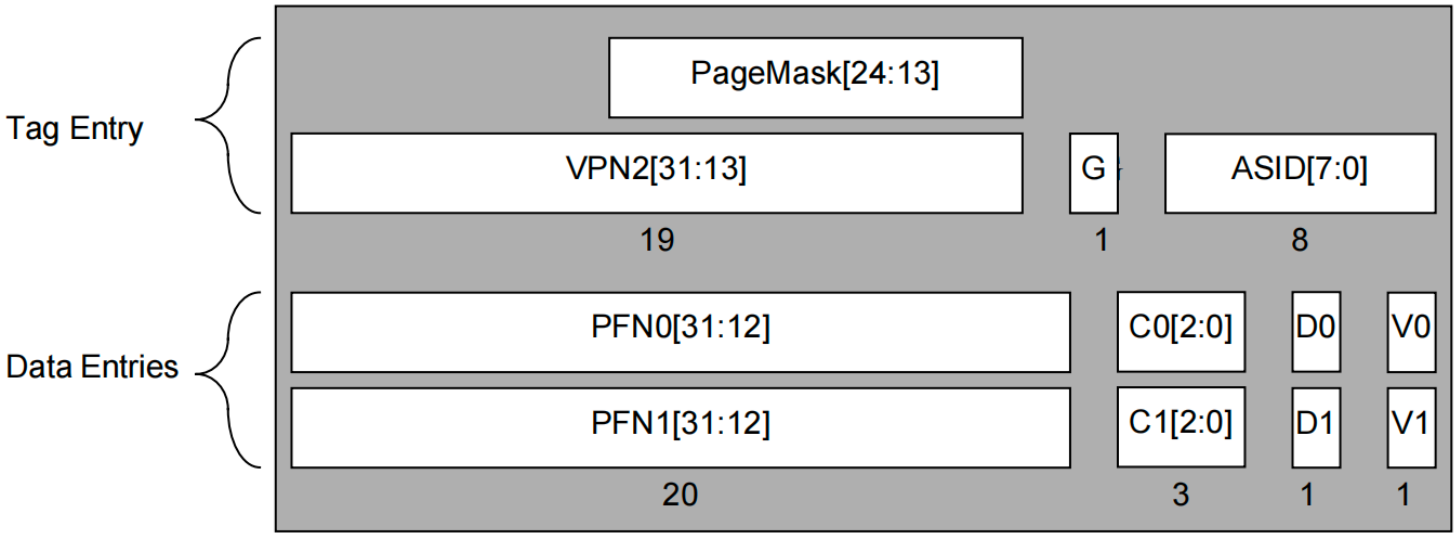
```
static inline u_long page2pa(struct Page *pp) {
        return page2ppn(pp) << PGSHIFT;
}

//获取物理地址pa,对应页控制块：(读取pte)
static inline struct Page *pa2page(u_long pa) {
        //物理页号PPN(pa)
        if (PPN(pa) >= npage) {
                panic("pa2page called with invalid pa: %x", pa);
        }
        return &pages[PPN(pa)];
}

//页控制块pp,对应物理页虚拟地址
static inline u_long page2kva(struct Page *pp) {
        //KADDR:物理地址->虚拟地址
        return KADDR(page2pa(pp));
}

//虚拟地址va,对应物理页地址pa
static inline u_long va2pa(Pde *pgdir,u_long va){   //pgdir指向页目录基地址
    Pte *p;

    //1.pgdir指向va对应的一级页表项：
    //pgdir指向页目录基地址,PDX(va)为一级页表偏移量
    pgdir=&pgdir[PDX(va)];
    if(!(*pgdir & PTE_V)){      //检查页表项是否有效(是否映射至物理页)
        return ~0;
    }

    //2.p指向二级页表基地址：
    //PTE_ADDR(*pgdir)获取物理页基地址;KADDR(pa)转为虚拟地址
    p=(Pte *)KADDR(PTE_ADDR(*pgdir));
    if(!(p[PTX(va)] & PTE_V)){
        return ~0;
    }
    //3.p[PTX(va)]指向va对应的二级页表项：
    //PTX(va)为二级页表偏移量,
    return PTE_ADDR(p[PTX(va)]);
}
```

4Kc：

`TLB`组成：一组`Key`+两组`Data`.

> 映射：< VPN, ASID > TLB---→ < PFN, N, D, V, G >

> 1. `TLB` 采用奇偶页的设计，即使用 `VPN` 中的高 19 位与 `ASID` 作为`Key`，一次查找到两个 `Data`（一对相邻页面的两个页表项）；
> 2.用 `VPN` 中的最低 1 位在两个 `Data` 中选择命中的结果。



`PageMask`:定义页大小；定义决定奇偶页选择的数位。

PageMask[11:0]	Page Size	Even/Odd Bank Select Bit
0000_0000_0000	4KB	VAddr[12]
0000_0000_0011	16KB	VAddr[14]
0000_0000_1111	64KB	VAddr[16]
0000_0011_1111	256KB	VAddr[18]
0000_1111_1111	1MB	VAddr[20]
0011_1111_1111	4MB	VAddr[22]
1111_1111_1111	16MB	VAddr[24]

`PageMask` 位于 `CP0` `PageMask` 寄存器；
`VPN2`，`ASID` 位于 `CP0` `EntryHi` 寄存器；
`PFN0`，`C0`，`D0`，`V0`，`G` 位于 `CP0` `EntryLo0` 寄存器；
`PFN1`，`C1`，`D1`，`V1`，`G` 位于 `CP0` `EntryLo1` 寄存器.

虚拟地址到物理地址的翻译：

1.Virtual address (VA) represented by the virtual page number (VPN) is compared with tag in TLB.

2. If there is a match, the page frame number (PFN0 or PFN1) representing the upper bits of the physical address (PA) is output from the TLB.

3. The Offset, which does not pass through the TLB, is then concatenated with the PFN.

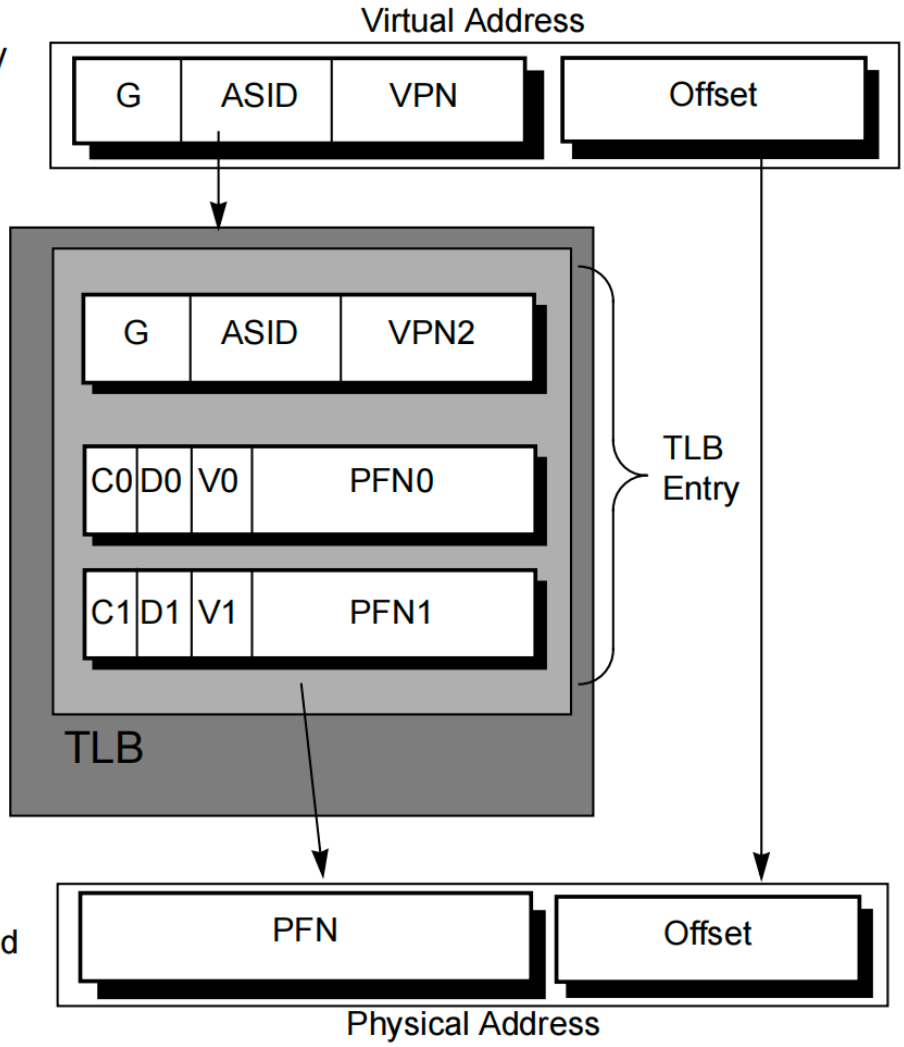


Figure 3-8 Overview of a Virtual-to-Physical Address Translation in the 4Kc Core

对比不同页大小的地址位数分配：

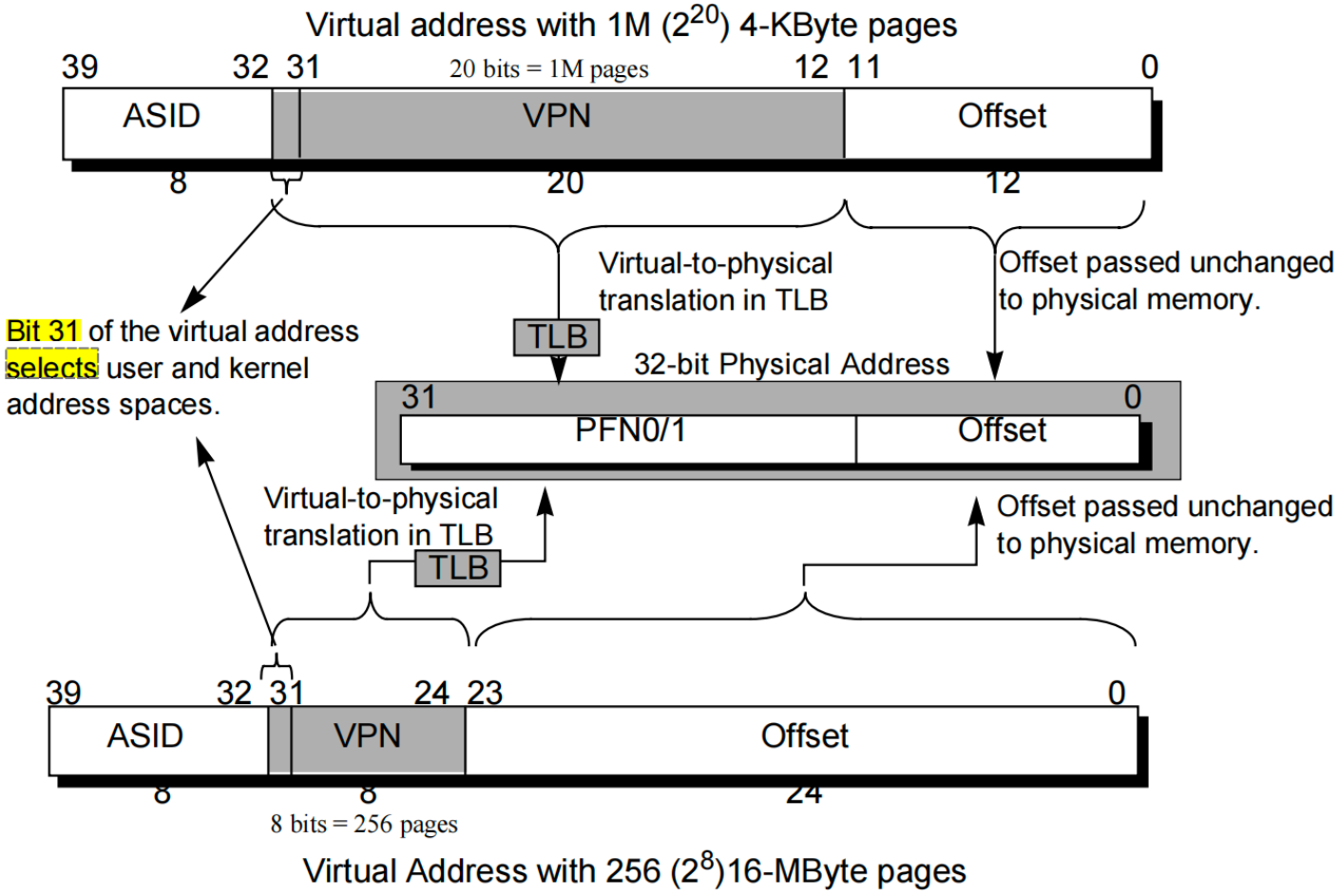


Figure 3-9 32-bit Virtual Address Translation