Thinking2.1

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？

A:在我们编写的C程序中，指针变量中存储的时虚拟地址；MIPS汇编程序中，`lw`、`sw`使用的是物理地址；

Thinking2.2

请你思考下述两个问题：

请从可重用性的角度，阐述用宏来实现链表的好处。

A：这些宏函数实现的都是对链表的一些基本操作，包括对链表的头插，尾插，中间插入，删除某个节点，遍历等等功能。本操作系统中的链表结构为：链表头是一个结构体，其中存储了头指针，其内置一个结构体，其中包含了指向下一节点的指针和指向上一节点内指向下一节点的指针的指针，从而形成链表。这种结构的巧妙之处在于让C语言这个本没有"泛型"的语言实现了类似"泛型"的功能，因为指向前方与后方的指针被结构体内置起来，其与链表每个节点中的数据类型相对独立，可以在任何数据类型中得到很好的使用。使用宏实现链表可以方便我们程序的移植以及代码的可读性，通过宏我们可以简化相对应的C代码，进而方便程序的阅读，同时，使用宏可以在维护的时候更改更少的部分完成我们自身的维护。

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。

A：see `/usr/include/sys/queue.h`，对于单向链表如果我们想要对某个位置的链表进行操作，必须一个一个遍历所有的节点，这样的复杂度很高，使用双向链表就可以即对前驱节点进行操作，也可以对后继节点进行操作，而循环链表就可以从表中任意一个元素出发查找元素，但是对未知位置的查找删除还是双向链表更快一点。

Thinking2.3

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。

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

```

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

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

```

正确的展开结构为C。

Thinking2.4

请你寻找上述两个 `boot_*` 函数在何处被调用。

`see in ./pmap.c`

Thinking2.5

请你思考下述两个问题：

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

A:在没有ASID时，进程如果完全占有虚拟地址空间，就会导致页表的重复调用并产生相应的冲突，ASID使得在OS中所有的进程都可以独自占用整个虚拟地址，从程序的角度来说就是，所有进程都可以当作自己独自占用整个虚拟地址空间，并且不会有与其他进程的冲突。

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

see in IDT R30xx Family Software Reference Manual，由于ASID只有6位长，操作系统软件必须提供

如果并发使用的地址空间超过64个，请手动；

但这种情况可能不会经常发生。在这样的系统中，新任务分配新的ASID，直到分配完所有64个ASID；当时，所有任务刷新其ASID“取消分配”，TLB刷新；在重新输入每个任务时，会给出一个新的ASID。因此，ASID会刷新这种情况相对少见。

Thinking2.6

请你思考下述两个问题：

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

`tlb_invalidate` 调用 `tlb_out`

请用一句话概括 tlb_invalidate 的作用

使用 tlb_invalidate 函数可以实现删除特定虚拟地址的映射，每当页表被修改， 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

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

LEAF(tlb_out)
//1: j 1b
    nop
//把CP0_ENTRYHI原有值存储到$k1中
    mfc0    k1,CP0_ENTRYHI
//把a0中值存放到CP0_ENTRYHI；
//CP0_ENTRYHI存放了虚拟地址空间及其标志位
    mtc0    a0,CP0_ENTRYHI
    nop
//查询CP0_ENTRYHI中虚拟地址是否存在TLB中；
//如果有则把匹配项的index保存到Index寄存器中；
//没有匹配则置Index的最高位为1.
    tlbp
//nop用于等待tlbp执行完毕（流水线暂停）
    nop
    nop
    nop
    nop
//读取改写后的CP0_INDEX到$k0
    mfc0    k0,CP0_INDEX
//如果$k0中值小于0，即CP0_INDEX最高位置1，即TLB缺失
//则跳转到NOFOUND
    bltz    k0,NOFOUND
    nop
//清空CP0_ENTRYHI和CP0_ENTRYLOW
    mtc0    zero,CP0_ENTRYHI
    mtc0    zero,CP0_ENTRYLO0
    nop
//更新TLB
    tlbwi
NOFOUND:
//读取CP0_INDEX到$k1
    mtc0    k1,CP0_ENTRYHI
    j       ra
    nop
END(tlb_out)
```

Thinking 2.7

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 × 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

PTbase + PTbase >> 9

• 映射到页目录自身的页目录项（自映射）

PDE = PTbase|(PTbase)>>9|(PTbase)>>18|(PTbase)>>27

Thinking 2.8

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。

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，当开放PSE时，页面扩大，原有的线性地址左移，这样的好处是表可以减少一级，但缺点是页面过大时会造成浪费。