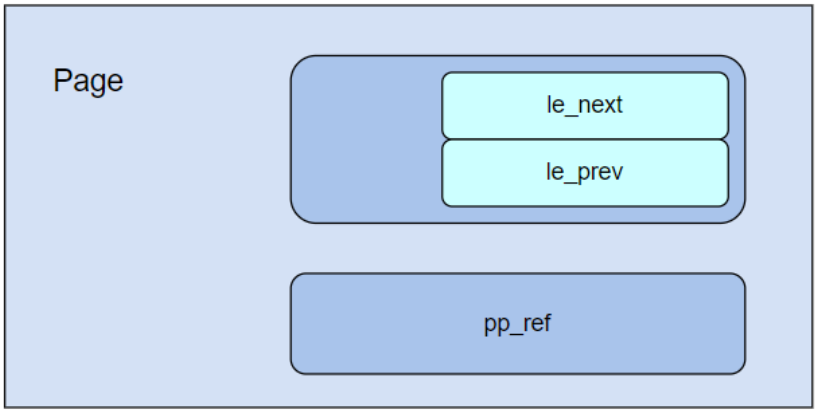
实验目的

- 了解 MIPS-R3000 的访存流程与内存映射布局
- 掌握与实现物理内存的管理方法（链表法）
- 掌握与实现虚拟内存的管理方法（两级页表）
- 掌握 TLB 清除与重填的流程

实验难点

1. **物理内存管理部分**

   本实验采用链表法实现物理内存的管理，补充链表操作的代码本身问题不大，但理解链表宏与表头结构体的设计存在一些难度，也就是Page结构体的设计部分。



   它只是一个信息载体，记录相应物理内存页的信息；作为结构体，它存储了头指针，并内置一个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的结构体，从而构成链表。需要寻找真正的内存页时，应当借助 `pmap.h` 中的 `page2ppn` 和 `page2pa` 函数，前者得到这个Page结构体相对于页表的偏移项数，后者则利用此偏移，通过移位得到对应的物理内存页地址。

2. **虚拟内存管理部分（多级页表的理解）**

   理论课中提供了多级页表管理内存的大致框架，但实验指导书让我看到了更多其中的实现细节，其中页表结构、TLB和自映射机制是需要理解的地方。通过学习，我大致总结了以下的基本知识：

   具体的地址转换过程、TLB重填和与自映射相关的计算部分在指导书中已经描述得极为详尽，但想要熟练运用与计算，仍需要多加理解练习。比如，扩展到更多级页表时的计算情况如何。

3. **页面与地址之间的转化**

   实验代码中有许多零碎的函数、常量包揽了页面与地址转换相关的内容，分布比较散乱，其应用也让人不容易记住，上机实验时也因为对这块的理解记忆不够而丢了分。

   通过`pmap.h`中这些用于数据结构转化的函数，可以实现（虚拟地址 -- （物理地址） -- 内核虚拟地址）之间的转换。

4. **C语言中指针、结构体等运用的复习**

   这块记忆与使用不清楚的话写代码容易出锅。

思考题

Thinking 2.1

C指针变量存储的地址是虚拟地址，MIPS汇编程序中使用的也是逻辑地址。如果理解无错误，采用了虚拟内存技术的操作系统中，所有程序访问的都应该是逻辑地址。

Thinking 2.2

1. 最浅显的好处是实现工整，使得对链表的插入、删除等诸多基本操作在代码中变得更简洁可读，便于重复调用。当然，它的可重用性也体现在对不同数字类型的适应上，通过对链表结构的设计（表头为一结构体，其中内置头指针与包含前后指针的结构体）使得链表节点的数据类型相对独立，能够使用在各种数据类型中而不必修改。
2. 单向链表增删节点的操作最简单，但是只能从头到尾遍历，因为它只能对后继节点操作，无法操作前驱节点，导致时间复杂度会比较高；双向链表可进可退，能够对前驱和后继节点都进行增删，在需要对前驱节点操作时比单向链表更快捷，但增删节点的操作比较复杂，因为需要多分配一个指针存储空间；循环链表对首尾节点的操作比单向链表更方便，其操作性能也优于各自对应的非循环链表类型。

Thinking 2.3

C项正确。

```
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

在`./mm/pmap.c`中，`mips_vm_init()`调用了`boot_map_segment()`，`boot_map_segment()`的实现本身调用了`boot_pgdir_walk()`。

Thinking 2.5

1. 由于同一虚拟地址在不同的地址空间中通常映射到不同的物理地址，因此，发生进程切换时，必须要通过ASID来识别出进程的TLB entry，保证硬件可以区分不同的进程地址空间。
2. 64

Thinking 2.6

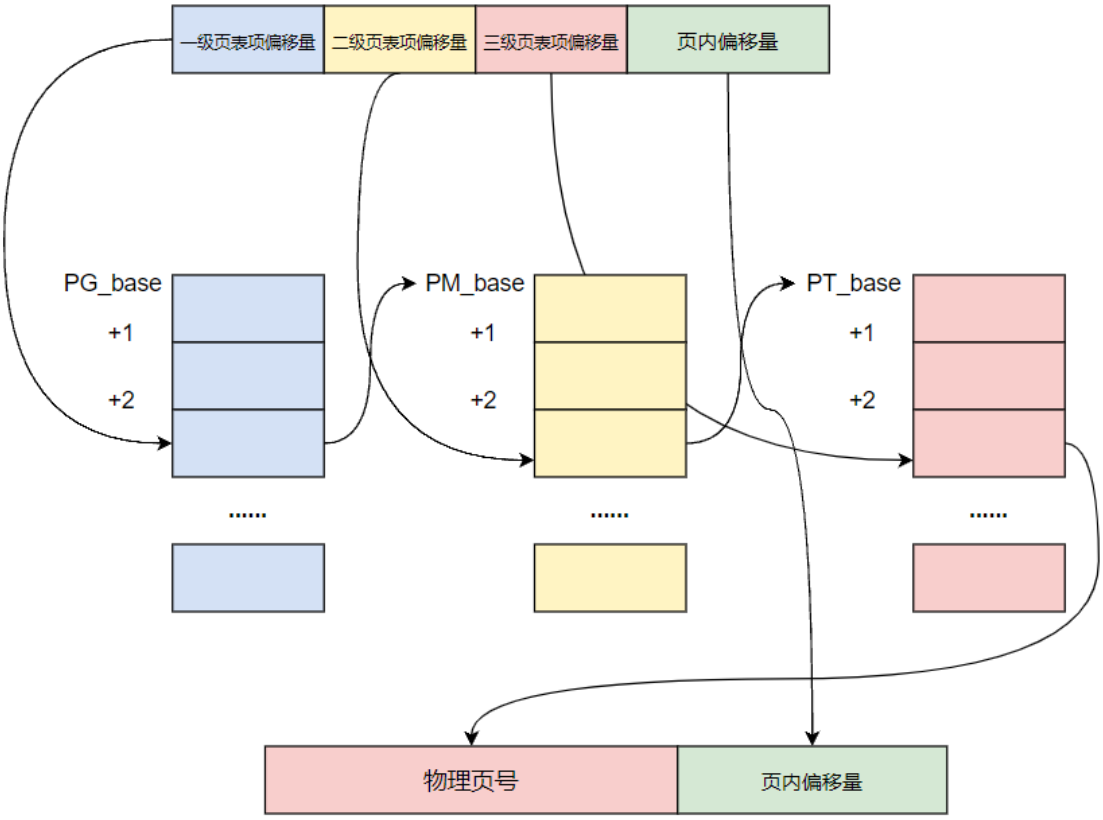
1. `tlb_invalidate`调用`tlb_out`
2. 使得进程对应页表`pgdir`中虚拟地址`va`对应的tlb项失效
3.
```
LEAF(tlb_out)
//1: j 1b
nop
    mfc0    k1,CP0_ENTRYHI      ##将寄存器EntryHi中的数据存到k1中，防止其数据被存入的键值覆盖
    mtc0    a0,CP0_ENTRYHI      ##将a0中存储的键值存入EntryHi
    nop
    tlbp                        ##根据EntryHi中的键值Key查找TLB中与之对应的表项，如果命中，则将表项的索引存入Index寄存器，否则Index最高位置1
    nop
    nop
    nop
    nop
    mfc0    k0,CP0_INDEX        ##将寄存器Index中的值存到k0中，用来比较
    bltz    k0,NOFOUND          ##将k0与0比较，如果小于0说明没有命中
    nop
    mtc0    zero,CP0_ENTRYHI    ##将EntryHi清零
    mtc0    zero,CP0_ENTRYLO0   ##将EntryLo清零
    nop
    tlbwi                       ##将EntryHi和EntryLo存储的值写入Index寄存器中的值指定的TLB表项中
NOFOUND:

    mtc0    k1,CP0_ENTRYHI      ##恢复EntryHi的原始内容

    j   ra
    nop
END(tlb_out)
```

Thinking 2.7

(模仿二级页表计算)



1. 易知三级页表的地址转换关系应为：

> pgd 基址 + pgd 偏移 = pgd 表项（存 pmd 基址）
> pmd 基址 + pmd 偏移 = pmd 表项（存 pte 基址）
> pte 基址 + pte 偏移 = pte 表项（存物理页地址）

故三级页表页目录的基地址
$PG_{base} = PT_{base} - PM_{offset} - PG_{offset} = PT_{base} - PT_{base} >> 9 - PT_{base} >> 18$

2. 映射到页目录自身的页目录项：$PT_{base} >> 30$

Thinking 2.8

x86和MIPS在内存管理上的区别

1. TLB不命中的处理

   MIPS 会触发TLB Refill 异常，内核的 tlb_refill_handler 会以 pgd_current 为当前进程的 PGD 基址，索引获得转换失败的虚址对应的 PTE，并将其填入 TLB；

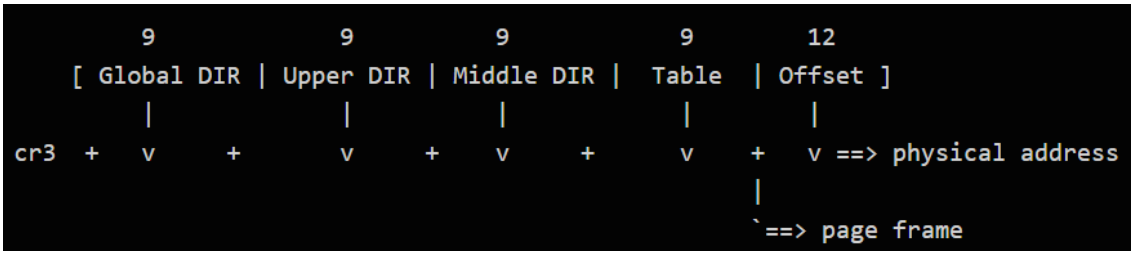
   而 X86 在 TLB 不命中时，是由硬件 MMU 以 CR3 为当前进程的 PGD 基址，索引获得 PFN 后，直接输出 PA。同时 MMU 会填充 TLB 以加快下次转换的速度。

   另外，对于转换失败的虚址，MIPS 使用 BadVAddr 寄存器存放，X86 使用 CR2 存放。

2. x86_64架构的分页与MIPS略有不同，当然这是基于不同场景的策略选择的问题，并不代表MIPS就不能搞这种更多级的多级页表管理内存。

Paging for 64-bit Architecture (x86_64)

Page size	4KB
Number of address bits used	48
Number of paging leveles	4
Linear address splitting	9 + 9 + 9 + 9 + 12



3. 其他很多部分个人感觉其实两种架构的处理方式都比较类似，当然也有本人学艺不精的因素在，没能体会到更多显著的区别。

参考：https://jasoncc.github.io/kernel/jasonc-mm-x86.html

https://lwn.net/Articles/250967/

实验体会

进入内存管理部分后，我深刻感到实验难度的骤然变大。撰写代码的过程或许并不复杂，但对MOS源码的阅读、对各种存储管理方式的理解以及对存储细节的具体实现都给我带来了不小的挑战。本单元实验的指导书写得事无巨细，我也在进行实验前对理论课学到的内存管理知识做了整理与扩展阅读，即使如此，面对实验代码时也依旧感到十分耗费精力，尤其是**TLB和页目录自映射机制**那部分的内容。

lab2中，补全代码的任务基本上都在`pmap.c`中进行，但书写代码需要用到的函数、宏定义几乎遍布整个实验操作系统，其功能、位置等十分零散，不利于记忆，在上机实验时我也因为没发现一些函数功能而极大地增加了自己写地址转换相关代码的难度。因此我学到的最重要的一课就是"**读代码**"还有整理代码——认真地理解性阅读实验操作系统中的每一个代码，并及时对其功能、封装的函数等进行梳理与整合记忆，这样才能在使用时得心应手。当然，有关内存管理的具体理论知识也需要进一步夯实，毕竟它对后续实验的影响也是深远的——在动手进行实验、阅读指导书之前，请务必复习好**理论课**的知识。

（来自正在做lab3的笔者：读代码 + 整理代码 + 结合理论课的模式确实能够极大地推进实验完成的速度以及对指导书的理解！）