Lab2 实验报告

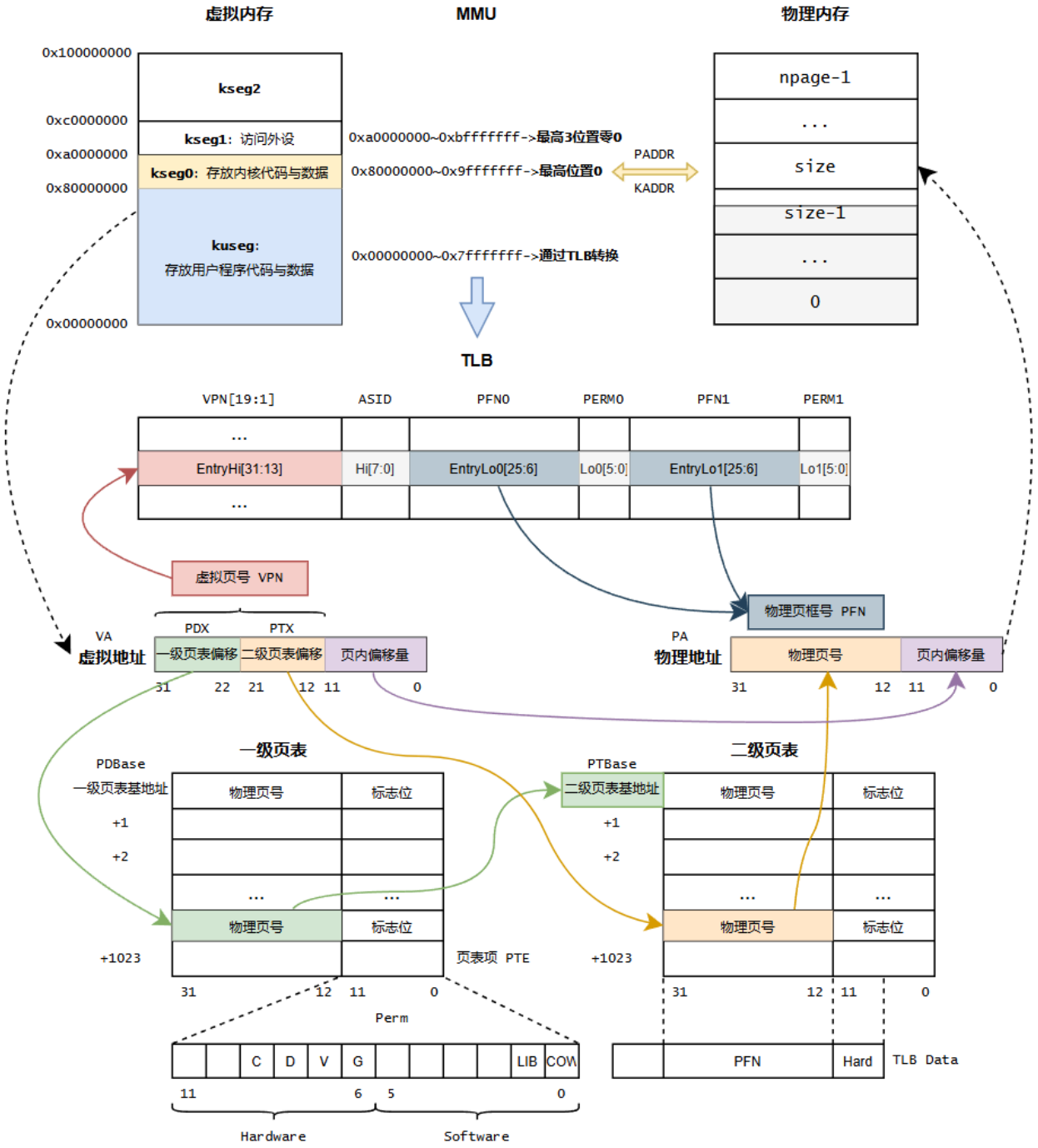
零、实验目的

- 了解 MIPS 4Kc 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

在 MOS 的设计中，两级页表作为一种内核数据结构放置于内存中，在用户程序需要访问虚拟地址时（通过中断机制）内核负责将对应的页表项填入 TLB。在 Lab2 中，我们的核心任务是管理两级页表与填写 TLB。

MOS 通过两级页表保存和描述虚拟地址与物理地址的映射关系，通过 TLB 加速虚拟地址到物理地址的转换。在本次实验中，我们将实现页表的初始化、页表项的填写、TLB 的清除与重填等功能。

整个 Lab2 在干什么？ 笔者呕心沥血画出来一个全局图：



宏观来讲，是**虚拟地址->物理地址**的过程；微观来讲，访存的具体实现是由 **TLB、两级页表结构**完成。

一、思考题

Thinking 2.1

> 请根据上述说明，回答问题：在编写的 C 程序中，指针变量中存储的地址被视为虚拟地址，还是物理地址？MIPS 汇编程序中 `lw` 和 `sw` 指令使用的地址被视为虚拟地址，还是物理地址？

都是虚拟地址。C 程序是软件层面的，处于用户态，自然使用虚拟地址更便捷。MIPS 中的地址存储为虚拟地址，通过 MMU 内存管理单元实现虚拟内存与物理内存的转换，在我们的计组实验中省去了这个硬件模块，因此直接用物理地址，但实际的 CPU 如 4Kc 的汇编指令是使用虚拟地址的。

Thinking 2.2

> 请思考下述两个问题：
>
> - 从可重用性的角度，阐述用宏来实现链表的好处。

用宏来实现链表，可以**适配不同类型的数据结构。**
例如，创建一个链表，使用 `LIST_HEAD(name, type)` ，只要传入对应的名称和类型就可以，因为宏的实现是替换字符串，类型也可以自定义了。与之对应的，如果用结构体和函数实现，类型就规定死了。

> - 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。

打开 `/usr/include/sys/queue.h` ，找到其中单向链表和循环链表的宏：
单向链表：

- `SLIST_HEAD(name, type)`：创建一个头部，与实验中的双向链表一样。
- `SLIST_ENTRY(type)`：一个链表项，只包含指向下一个的指针。

循环链表：

- `CIRCLEQ_HEAD(name, type)`：与实验中的双向链表不同，包含两个指针，一个指向队列第一个，一个指向队列最后一个，这是循环的实现方式。
- `CIRCLEQ_ENTRY(type)`：一个链表项。与实验的双向链表有很大不同，这个包含了两个指针，一个是指向下一个的指针，一个是指向上一个的指针。（实验中是指向下一个的指针和指向上一个的指向下一个的指针的指针（））

知道了结构是什么样的，性能分析就很容易，看看代码中的循环结构就可以：

时间复杂度	插入	删除
实验中的双向链表	$O(1)$	$O(1)$
单向链表	在之前插入 $O(n)$，在之后插入 $O(1)$	$O(n)$
循环链表	$O(1)$	$O(1)$

Thinking 2.3

> 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。（如下）

```
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;
}
```

在 `include/pmap.h` 中，可以找到 `extern struct Page_list page_free_list;`
由我们实验实现的双向链表，可以知道一个是一个指针和一个指针的指针，所以排除 A 项；同时头部也包含一个指针，因此应该是 C 项。

Thinking 2.4

> 请思考下面两个问题：
>
> - 请阅读上面有关 TLB 的描述，从虚拟内存和多进程操作系统的实现角度，阐述 ASID 的必要性。

同一虚拟地址在不同地址空间中通常映射到不同的物理地址，ASID 用于区分不同的地址空间。

> - 请阅读 MIPS 4Kc 文档《MIPS32® 4K™ Processor Core Family Software User's Manual》的 Section 3.3.1 与 Section 3.4，结合 ASID 段的位数，说明 4Kc 中可容纳不同的地址空间的最大数量。

"The purpose of the TLB is to translate virtual addresses and their corresponding Address Space Identifier (ASID) into a physical memory address. The translation is performed by comparing the upper bits of the virtual address **(along with the ASID bits)** against each of the entries in the tag portion of the JTLB structure.
"In this figure the virtual address is extended with an **8-bit address-space identifier (ASID)**, which reduces the frequency of TLB flushing during a context switch. This 8-bit ASID contains the number assigned to that process and is stored in the CP0 EntryHi register."

ASID 段共有 8 位，可以被设置为 256 个不同的值。因此最多可容纳 256 个不同的地址空间。

Thinking 2.5

> 请回答下述三个问题：
>
> - `tlb_invalidate` 和 `tlb_out` 的调用关系？

在 `kern/tlbex.c` 中：

```
13 void tlb_invalidate(u_int asid, u_long va) {
14         tlb_out((va & ~GENMASK(PGSHIFT, 0)) | (asid & (NASID - 1)));
15 }
```

说明是 `tlb_invalidate` 调用 `tlb_out`

> - 请用一句话概括 `tlb_invalidate` 的作用。

删除特定虚拟地址（`asid` 和 `va` 确定）在 TLB 中的旧表项。

> - 逐行解释 `tlb_out` 中的汇编代码。

```
LEAF(tlb_out)
.set noreorder
    mfc0    t0, CP0_ENTRYHI # 把 CP0 寄存器 EntryHi 中的值读取到 t0
    mtc0    a0, CP0_ENTRYHI # 把 a0 的值保存到 CP0 寄存器 EntryHi 中
    nop
    tlbp # 查询虚拟地址的物理地址映射
    nop
    mfc0    t1, CP0_INDEX # 将 CP0 寄存器 Index 中的值读取到 t1
.set reorder
    bltz    t1, NO_SUCH_ENTRY # 如果 t1 < 0，跳转到 NO_SUCH_ENTRY
.set noreorder
    mtc0    zero, CP0_ENTRYHI # 清空 CP0 寄存器 EntryHi
    mtc0    zero, CP0_ENTRYLO0 # 清空 CP0 寄存器 EntryLo0
    mtc0    zero, CP0_ENTRYLO1 # 清空 CP0 寄存器 EntryLo1
    nop
    tlbwi # 将 CP0 的三个寄存器写入 TLB 的 Index
.set reorder

NO_SUCH_ENTRY:
    mtc0    t0, CP0_ENTRYHI # 将 t0 重新赋值到 CP0 寄存器 EntryHi
    j       ra # 函数返回
END(tlb_out)
```

Thinking 2.6

> 请结合 Lab2 开始的 CPU 访存流程与下图中的 Lab2 用户函数部分，尝试将函数调用与 CPU 访存流程对应起来，思考函数调用与 CPU 访存流程的关系。

对于内核，调用 `mips_detect_memory` 函数探测内存，接着调用 `mips_vm_init` 进行虚拟内存初始化，需要用到 `alloc` ，接着进行物理页面初始化 `page_init` ，这一步将可用的物理内存分页，由链表组织。
对于用户，提供虚拟地址寻找物理地址时，查询目标页面是否在 TLB 中，如果在，直接读取到

PPN，结合 offset 完成访存；如果不在，触发 TLB Miss，首先根据页目录和虚拟地址寻找页表，如果页表无效，分配新的页表并填写页目录项，反之从页表查找页表项，如果页表项无效，分配新的页面填写到页表项，反之取出相邻奇偶页填写到 CP0 的 EntryLo，写回 TLB，再次访存。

Thinking 2.7

> 从下述三个问题中任选其一回答：
>
> - 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。
> - 简单了解并叙述 RISC-V 中的内存管理机制，比较 RISC-V 与 MIPS 在内存管理上的区别。
> - 简单了解并叙述 LoongArch 中的内存管理机制，比较 LoongArch 与 MIPS 在内存管理上的区别。

LoongArch：

- **页表结构**：LoongArch 使用了三级页表结构，而 MIPS 在某些情况下可能使用两级页表结构。这使得 LoongArch 能够更灵活地管理大内存空间。
- **TLB 缓存**：LoongArch 的 TLB 结构可能与 MIPS 略有不同，具体取决于 LoongArch 处理器的架构。不过，两者都利用 TLB 来加速地址转换。
- **特权模式**：LoongArch 可能支持更多的特权模式，例如用户模式、管理模式、超级模式等。这使得 LoongArch 能够更好地支持不同级别的特权操作。
- **异常处理**：LoongArch 和 MIPS 在异常处理机制上可能有一些差异，例如异常号的定义和异常处理程序的调用方式。

二、难点分析

项目结构

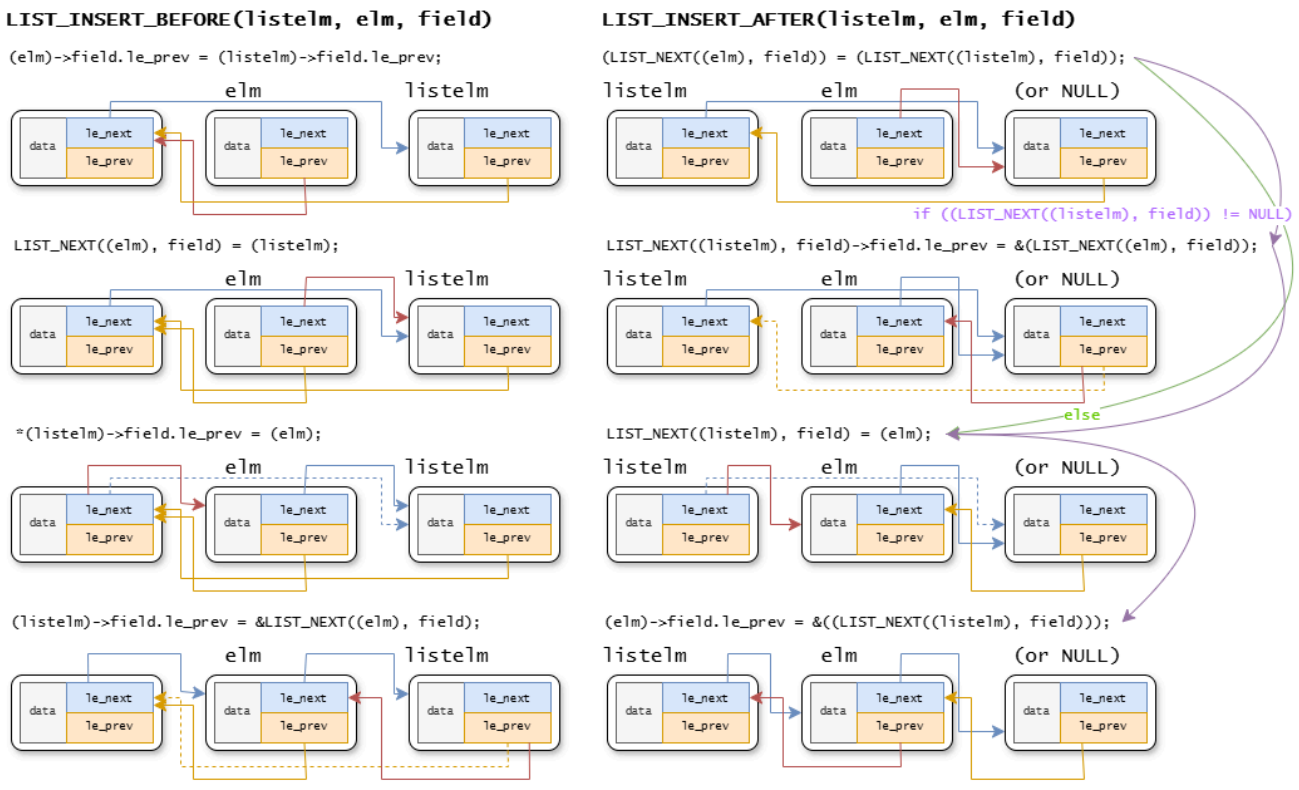
```
.
├── include
│   ├── elf.h // lab1：段头表节头表的结构体定义
│   ├── error.h // lab2：try 函数和报错处理
│   ├── machine.h // lab1：printcharc 函数
│   ├── mmu.h // lab1：内存布局；lab2：内存管理相关宏定义
│   ├── pmap.h // lab2：物理页结构体定义，页地址转换函数
│   ├── print.h // lab1：vprintfmt 函数声明
│   ├── printk.h // lab1：printk 函数声明
│   ├── queue.h // lab2：双向链表定义
│   ├── string.h // 常用字符串操作函数
│   └── types.h // 类型定义宏
├── init
│   ├── init.c // lab1：mips_init 内核程序启动；lab2：内存管理加载
│   └── start.S // lab1：启动，跳转到 mips_init 函数
├── kern
│   ├── machine.c // lab1：printcharc 实现
│   ├── pmap.c // lab2：内存管理
```

```
│   ├── printk.c // lab1: printk 实现
│   ├── tlb_asm.S // lab2: TLB
│   └── tlbex.c // lab2: tlb_invalidate
├── kernel.lds // lab1: linker script
├── lib
│   ├── print.c // lab1: vprintfmt 实现
│   └── string.c // Pre: 字符串处理函数实现
├── Makefile // 顶层 Makefile
├── target
│   └── mos // 构建出的内核 elf 可执行文件
├── tests // 存放测试信息
└── tools
    └── readelf
        ├── elf.h // lab1: 段头表节头表结构体
        └── readelf.c // lab1: 解析 elf 文件
```

物理内存管理

链表宏

MOS 中使用宏对链表的操作进行封装，链表宏定义位于 `include/queue.h` ，实现了双向链表的功能，其中对于链表的结构是比较难理解的，以至于理解 `LIST_INSERT_AFTER` 函数有一定难度，这里笔者做了两个图来对比 `LIST_INSERT_BEFORE` 和 `LIST_INSERT_AFTER` 两个宏：



这个写法也太抽象了，记住以下几点：

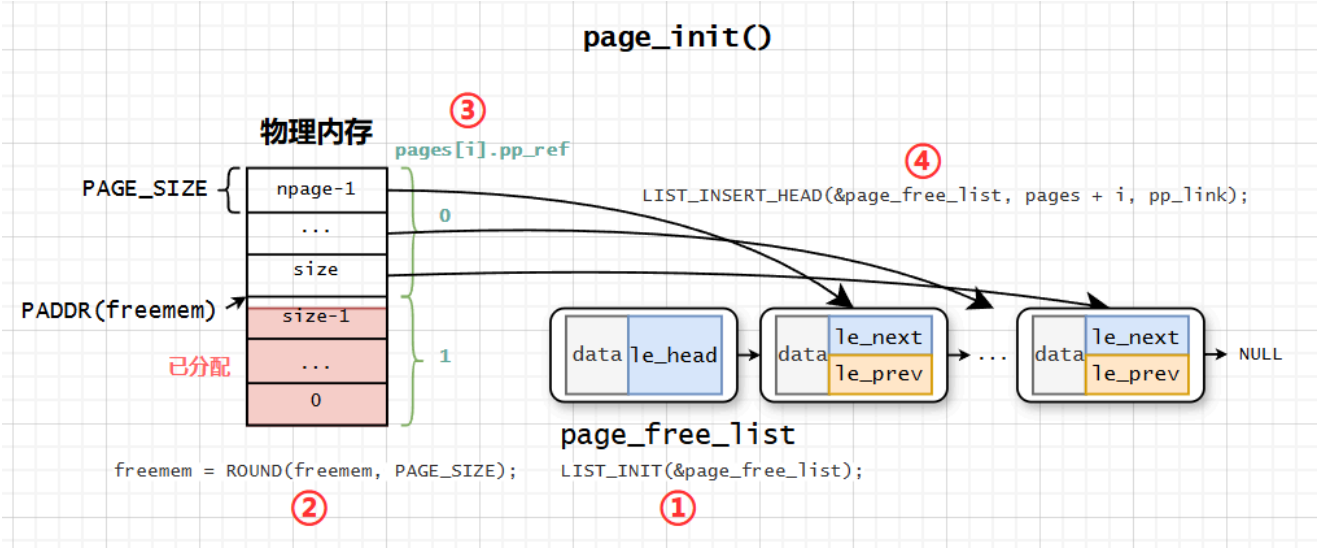
- `(elm)->field.le_prev` 指的是 `elm` 链表项的 `le_prev` **指针**，在这里我们不直接写 `le_next` ，而是用 `LIST_NEXT` 函数，如下面所说。

- `LIST_NEXT((elm), filed)` 指的是 `elm` 链表项的 `le_next` 所指向的**链表项**，你可以直接用链表项去连接，如 `LIST_NEXT((elm), field) = (listelm);` 是将 `elm` 的 `le_next` 指向了 `listelm`，而 `LIST_NEXT((listelm), field)->field.le_prev` 是 `listelm` 的 `le_next` 所指向的链表项的 `le_prev` 。
- `&LIST_NEXT((elm), filed)` 指的是 `elm` 链表项的 `le_next` 这个**指针本身**，如 `(elm)->field.le_prev = &((LIST_NEXT((listelm), field)));` 是将 `elm` 的 `le_prev` 指向了 `listelm` 的 `le_next` 本身。

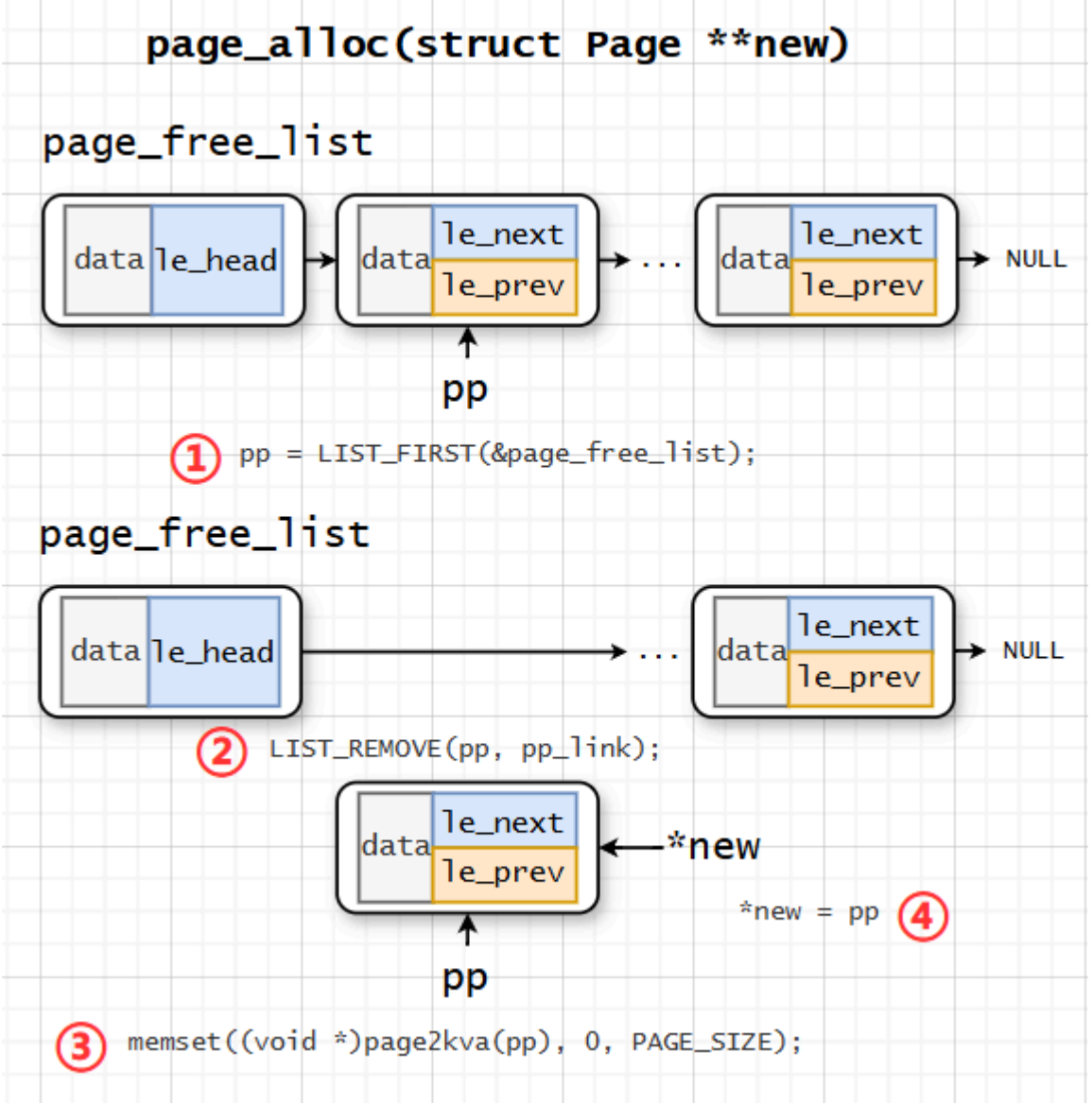
`page` 相关函数

在 `kern/pmap.c` 中有很多这样的函数，它们到底在干什么？

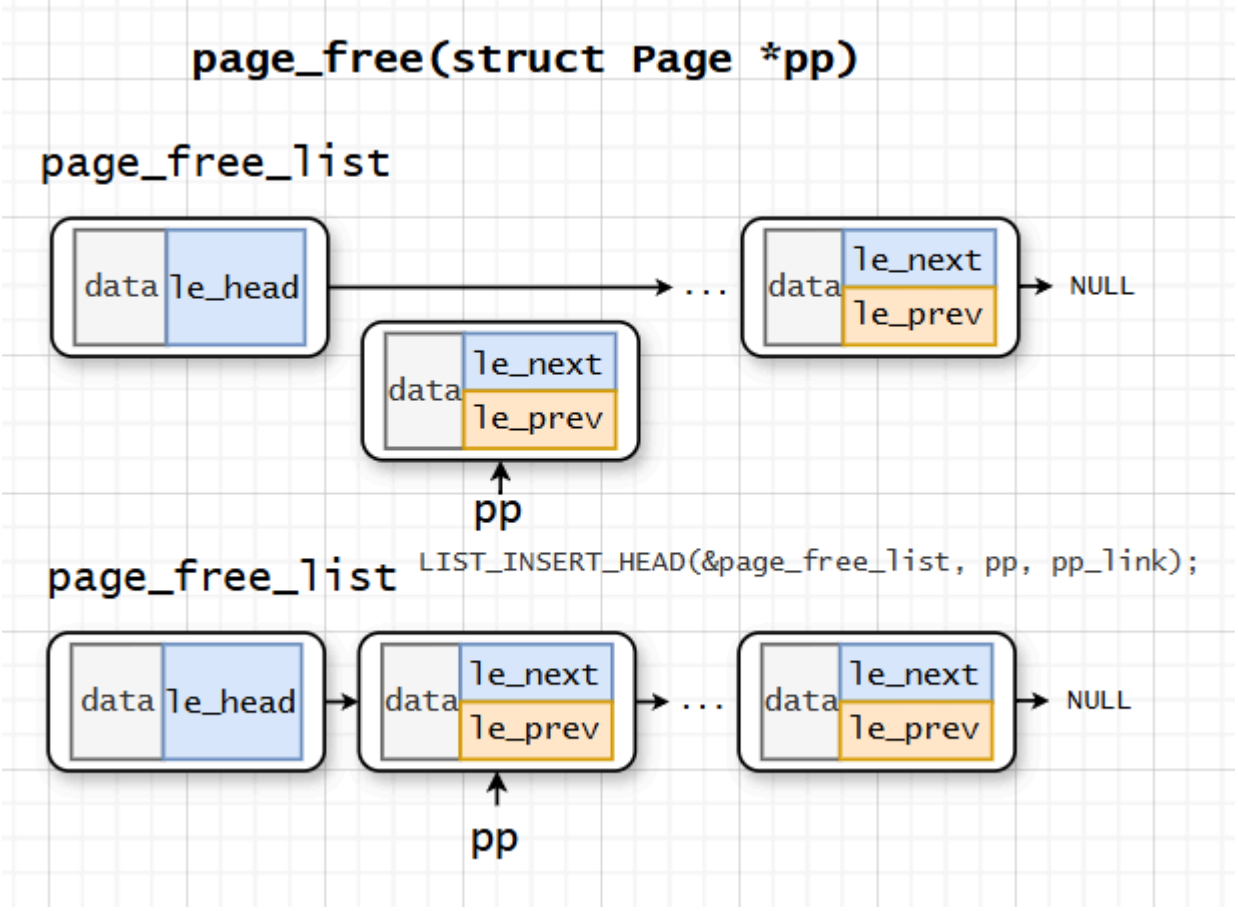
- `page_init()`：将未分配的物理页面加入链表 `page_free_list`，维护每个页面的 `pp_ref`。



- `page_alloc(struct Page **new)`：从 `page_free_list` 头部扣下一块物理页面，对其初始化，并把地址赋给 `new` 指向的空间。（这里的 `page2kva` 是将页控制块转为了 `kseg0` 对应的虚拟地址）



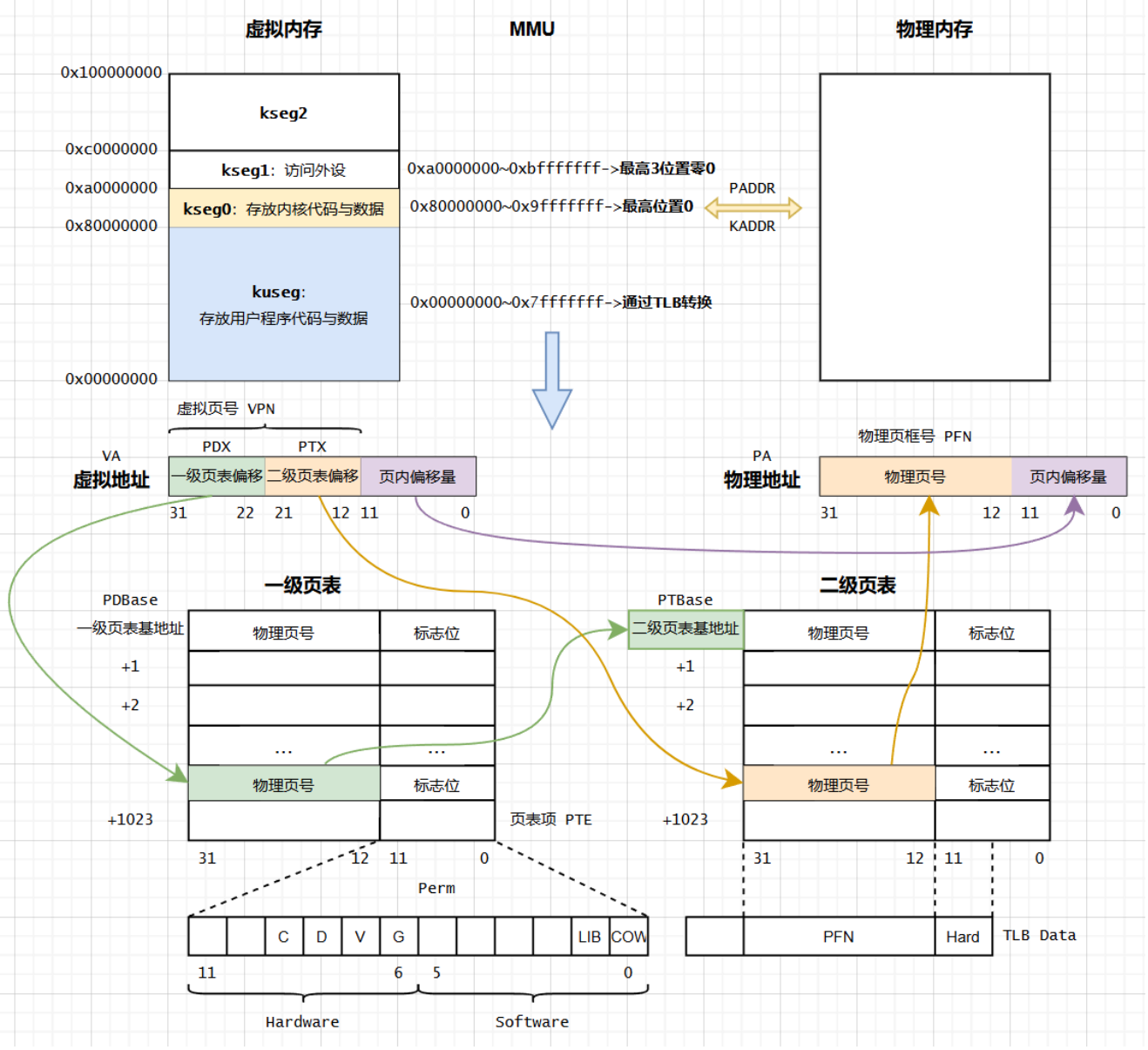
- `page_decref(struct Page *pp)`：将对应页控制块的 `pp_ref` 减 1，如果减到 0 了，就用 `page_free` 回收。
- `page_free(struct Page *pp)`：将对应页控制块重新从头部插入 `page_free_list`，保证它的 `pp_ref` 是 0。



虚拟内存管理

两级页表结构

MOS 中，对于 `kseg0` 的虚拟地址，使用 `PADDR` 和 `KADDR` 两个宏进行转换；对于 `kuseg` 的虚拟地址，用两级页表结构进行地址转换。



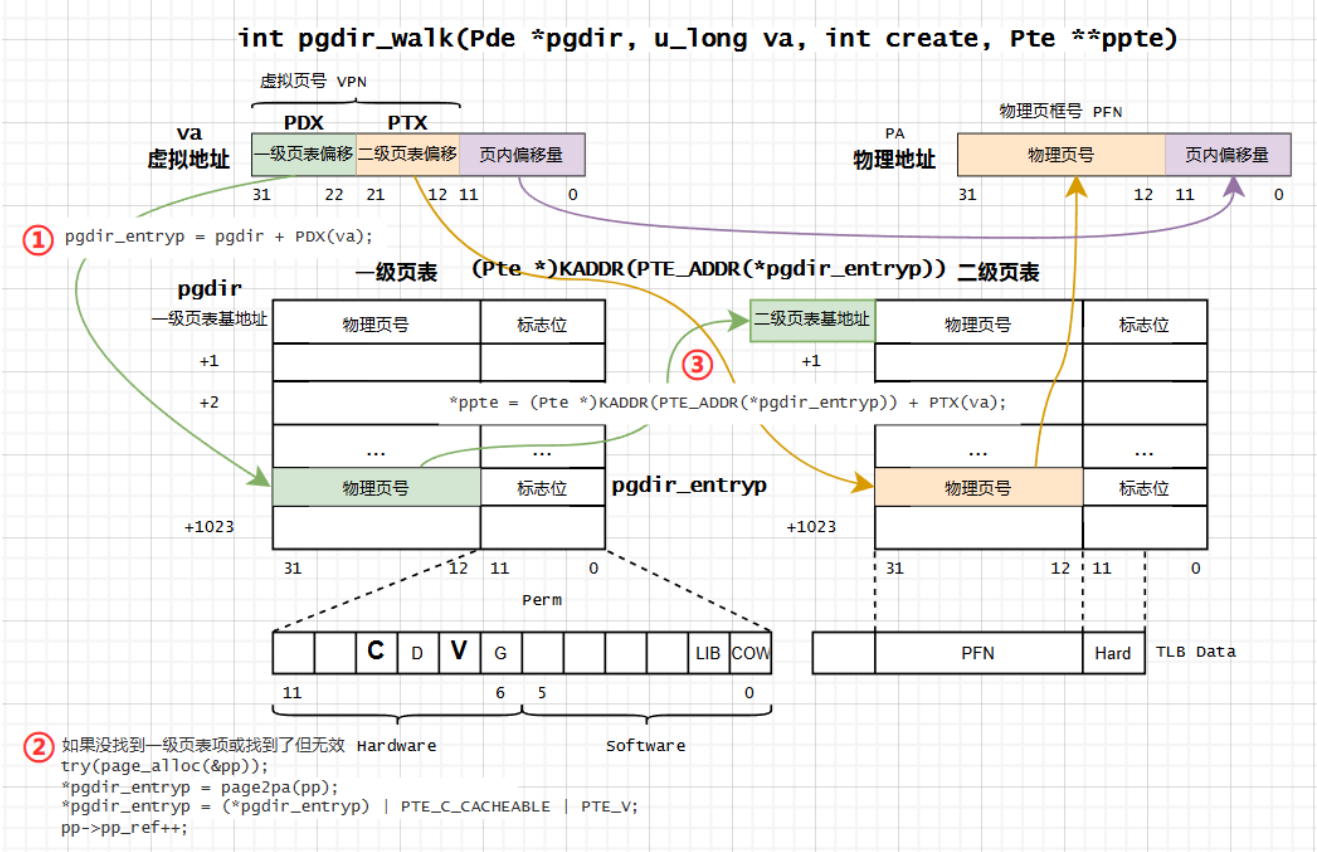
每个页表项的标志位，都分为硬件标志位和软件标志位。其中：

- `PTE_V`：有效位。表示该页表项有效。
- `PTE_D`：可写位。表示允许经由该页表项对物理页进行写操作。
- `PTE_G`：全局位。表示 TLB 仅通过虚页号匹配表项，而不匹配 ASID。（忽略）
- `PTE_C_CACHEABLE`：可缓存位。表示允许 CPU 使用 cache 加速对这些页面的访问请求，一般对于所有物理页面都是可缓存。
- `PTE_COW`：写时复制位。实现 `fork` 的写时复制机制。（忽略）
- `PTE_LIBRARY`：共享页面位。实现管道机制。（忽略）

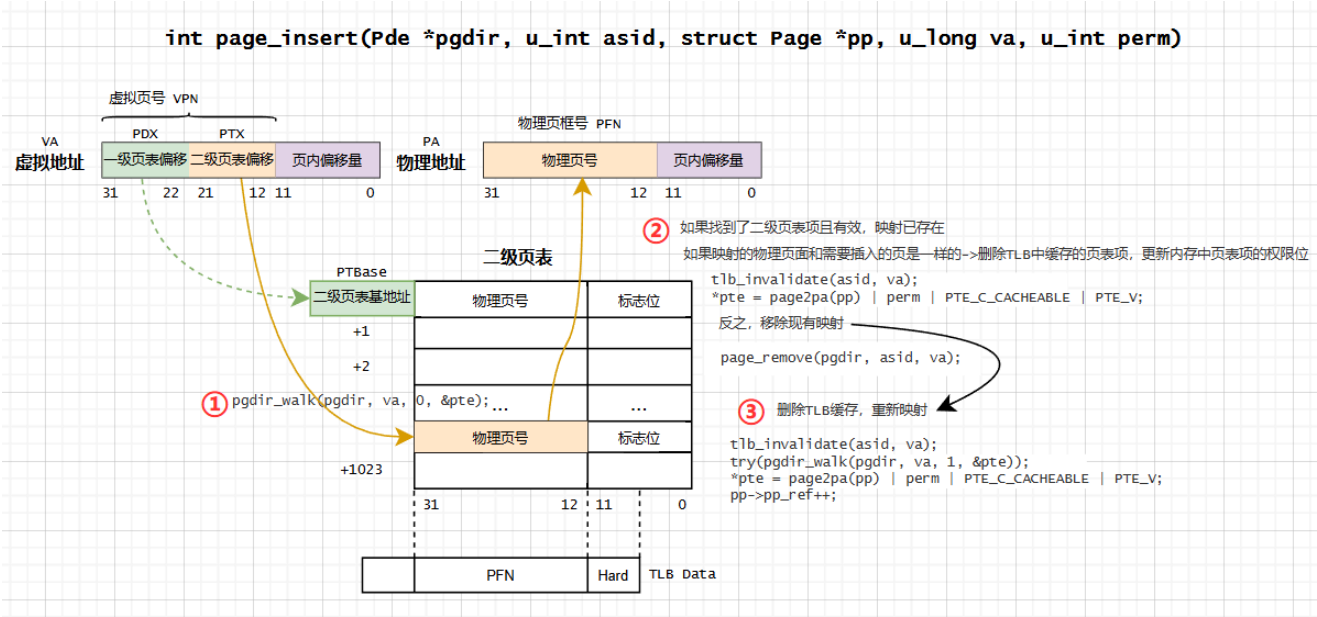
页表相关函数

`kern/pmap.c` 中有很多这样的函数，它们是在干什么？

- `int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)`：给定虚拟地址 `va`，在给定的一级页表基地址 `pgdir`，查找虚拟地址对应的二级页表项，地址写入 `*ppte`。



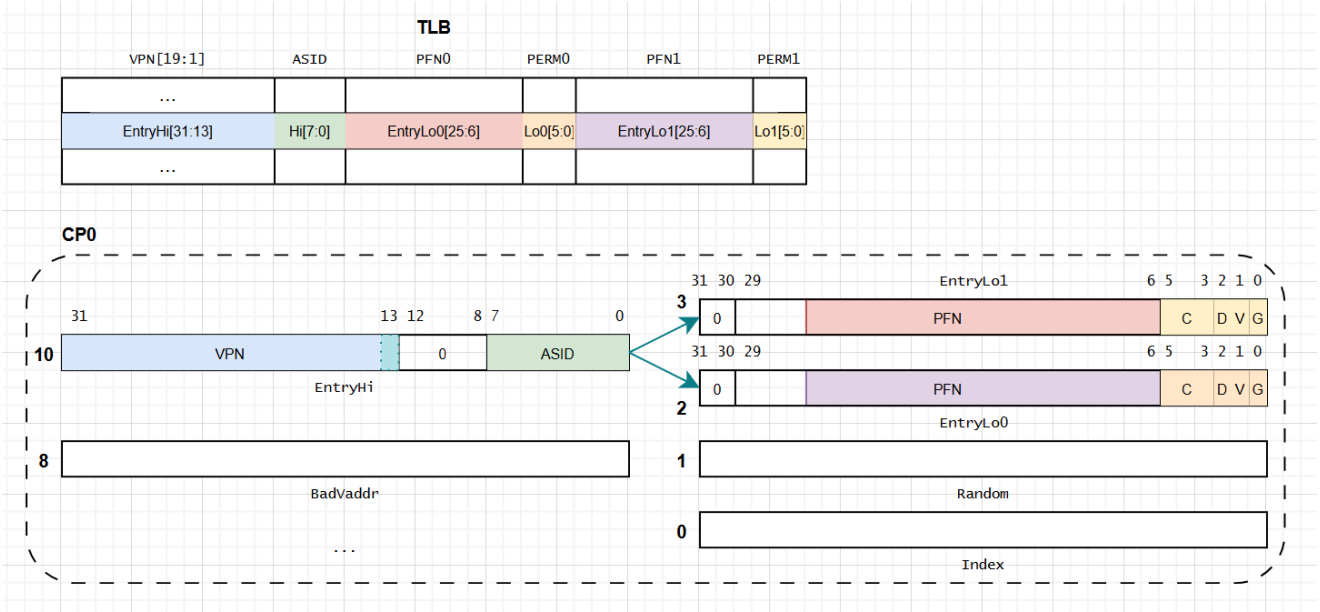
- `int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm)`：由给定的一级页表基地址 `pgdir`，将虚拟地址 `va` 映射到页控制块 `pp` 对应的物理页面，页表项权限设置为 `perm`。



- `struct Page * page_lookup(Pde *pgdir, u_long va, Pte **ppte)`：由给定的一级页表基地址 `pgdir`，找到虚拟地址 `va` 映射的物理页面的页控制块，同时将 `ppte` 指向的空间设为对于的二级页表项地址。
- `void page_remove(Pde *pgdir, u_int asid, u_long va)`：由给定的一级页表基地址 `pgdir`，找到虚拟地址 `va` 映射的物理页面的页控制块，删除该映射，对应的物理页面的 `pp_ref` 减一。

TLB

TLB 是一个缓存物理页号的单元，用来分担两级页表结构查找的压力。CP0 是一个寄存器组，内部有一些与 TLB 息息相关的寄存器。



TLB 和两级页表结构是什么关系

TLB 是两级页表结构的 “cache”。两级页表结构通过虚拟地址的虚拟页号，分别读取一级页表、二级页表，从二级页表项中得到物理页号，然后该物理页号加上虚拟地址的页内偏移得到目标物理地址。TLB 通过虚拟地址的虚拟页号，直接读到对应的物理页号，加上虚拟地址的页内偏移得到目标物理地址。

访存流程：CPU 发出访存指令，首先查询目标页面在不在 TLB 中，读取 PPN（PFN，物理页号，定位到具体的物理页面，也就是二级页表存的内容），加上虚拟地址的页内偏移信息，完成访存。如果没有查询到目标页面，触发 TLB Miss。

TLB 缺失

TLB 可以看作是两级页表结构的“捷径”，那么当 TLB 无效的时候，只能老老实实地完成两级页表索引的流程。

首先，根据虚拟地址的页目录偏移（一级页表偏移），得到一级页表项：如果一级页表项无效，分配新的二级页表，把它填写到一级页表项；如果一级页表项有效，继续。

然后，根据一级页表项的物理页号（二级页表基地址），找到对应的二级页表，由虚拟地址的页表偏移（二级页表偏移），得到二级页表项：如果二级页表项无效，分配新的物理页面，填写到页表项；如果二级页表项有效，继续。

最后，取出相邻奇偶页填写到 CP0 的 EntryLo 寄存器，写回 TLB，再次访存。

三、实验体会

Lab2 真的很难。

在笔者一遍一遍精读指导书，通读+反刍的策略下，终于稍微弄懂了访存流程！特别是一个个抽象的小函数，需要反复理解才能领会。

四、原创说明

此报告参考了以下资料或博客：

1. os-lab2实验报告 | hugo
2. BUAA-OS-lab2 | YannaのBlog