操作系统实验

lab2 内存管理
课下任务讲解

内容提要

- 实验概述
- 实验内容
  - 实验任务1-8
- 测试结果

实验概述

- 内存访问原理
- MIPS内存映射
- 物理内存管理
  - 使用链表管理物理内存
- 虚拟内存管理
  - 建立2级页表管理内存

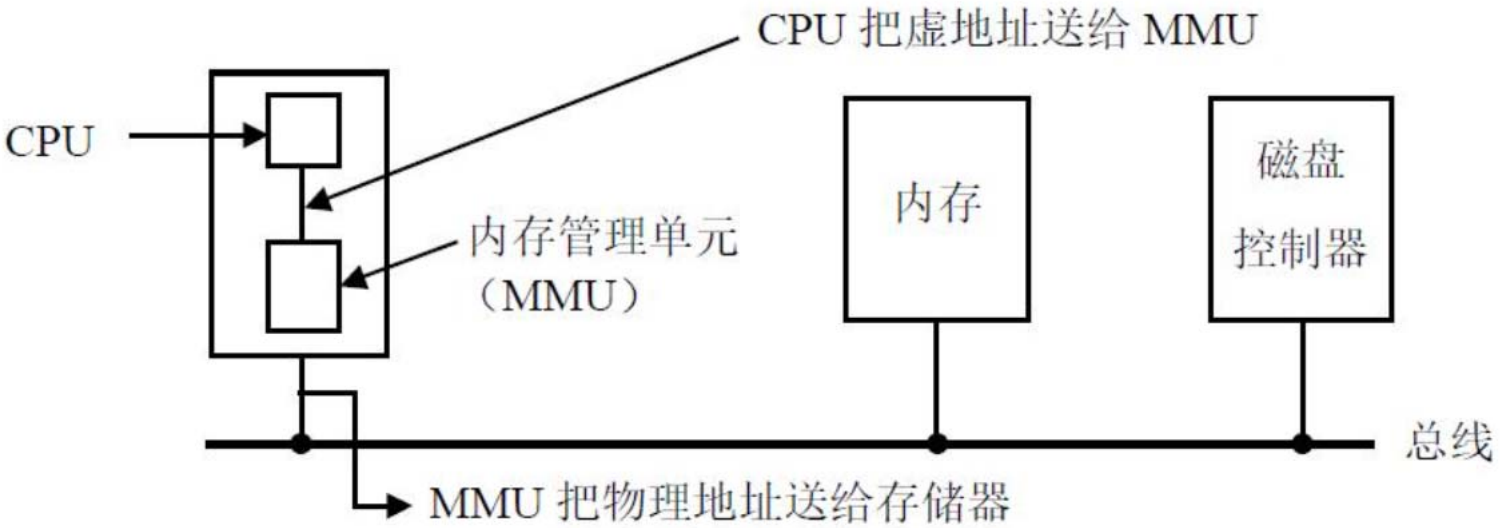
硬件和软件的功能

- MMU: 访问虚拟地址自动转换物理地址
- 操作系统: 建立与维护页目录、页表

- 代码里所有地址均为逻辑地址
- 只有通过算与查才能得到我们需要的物理地址

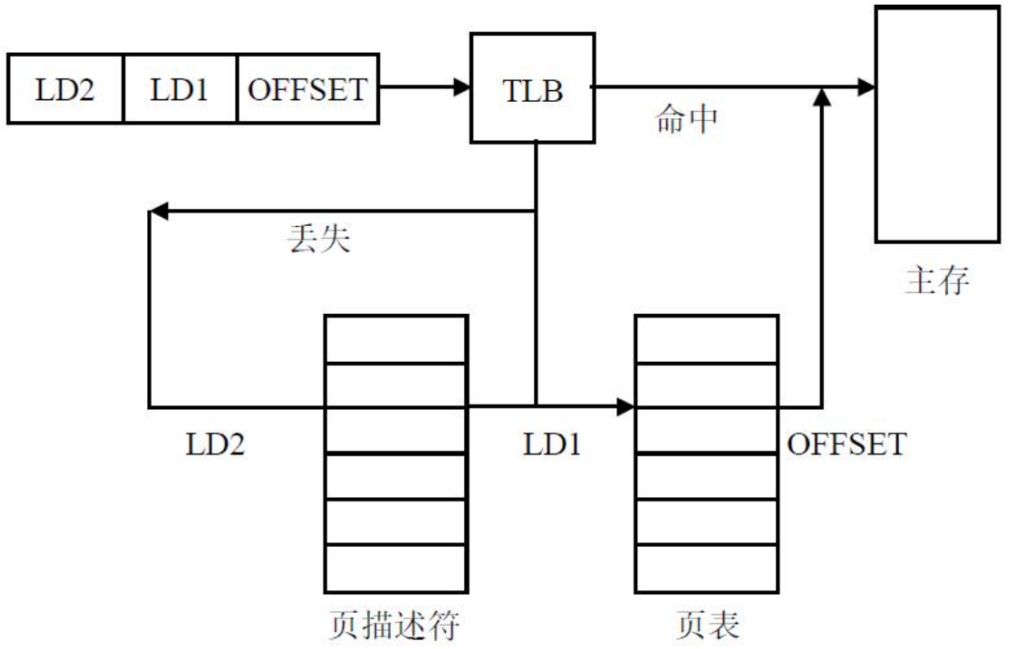
内存访问原理(1)

- MMU:全称是Memory Management Unit，中文为内存管理单元，MMU是硬件设备，它的功能是把逻辑地址映射为物理地址，并提供了一套硬件机制来实现内存访问的权限检查



内存访问原理(2)

- TLB:全称是Translation Lookside Buffer，中文为后援存储器，或者快表，它通常被安装在MMU的内部，TLB可被看作页表的高速缓存



内存访问原理(3)

- 完整的访存步骤（同时考虑cache）：
  - 1.CPU给出虚拟地址来访问数据，TLB接收到这个地址之后查找是否有对应的页表项。
  - 2.假设页表项存在，则根据物理地址在cache中查询；如果不存在，则MMU执行正常的页表查询工作之后再根据物理地址在cache中查询，同时更新TLB中的内容。
  - 3.如果cache命中，则直接返回给CPU数据；如果没有命中则按照相应的算法进行cache的替换或者装填，之后返回给CPU数据。

内存访问原理(4)

- 软件层面的内存管理要做什么？
  - 在内存中填充页表地址映射——MMU可以翻译地址，但不能翻译空的页表。
  - 掌握内存的使用情况——内存的使用情况是操作系统运行时必须考虑的要素。
  - 管理内存——(1)高效、节约地管理内存是构建操作系统必须考虑的要素。(2)通过虚拟内存来保护物理内存，同时扩展了内存空间。
  - 提供内存分配和释放的接口——任何应用程序都需要物理内存来运行。

MIPS内存映射(1)

- 1.kuseg (TLB-mapped cacheable user space, 0x00000000 - 0x7fffffff)： 这一段是用户模式下可用的地址，大小为2G，也就是MIPS约定的用户内存空间。需要通过MMU进行虚拟地址到物理 地址的转换。
- 2.kseg0 (direct-mapped cached kernel space, 0x80000000 - 0x9fffffff)： 这一段是内核地址，其内存虚存地址到物理内存地址的映射转换不通过MMU，使用时只需要将地址的最高位清零 (& 0x7fffffff)， 这些地址就被转换为物理地址。也就是说，这段逻辑地址被连续地映射到物理内存的低端512M空间。对这段地址 的存取都会通过高速缓存(cached)。通常在没有MMU的系统中，这段空间用于存放大多数程序和数据。对于有 MMU 的系统，操作系统的内核会存放在这个区域。

MIPS内存映射(2)

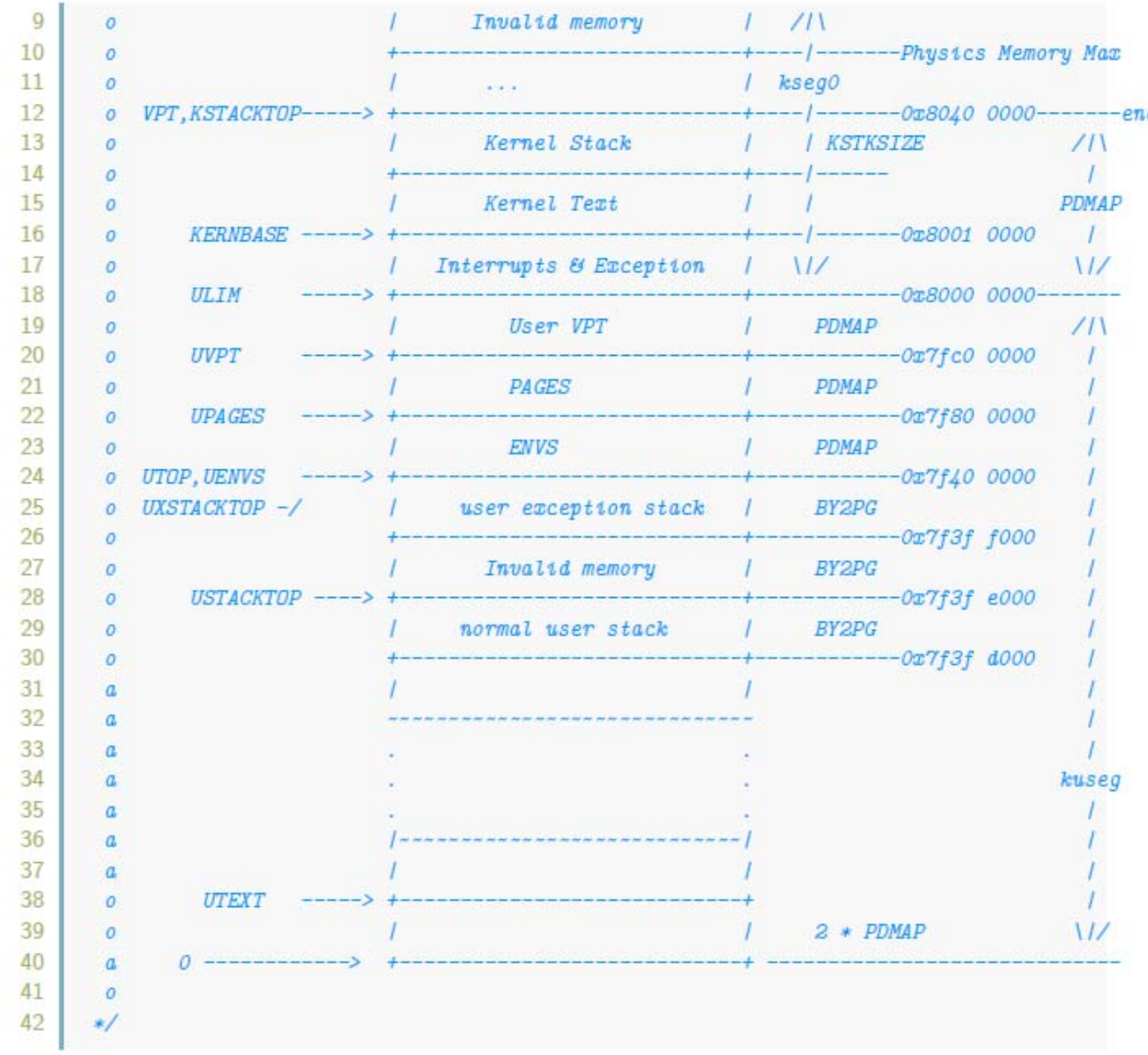
3.kseg1 (direct-mapped uncached kernel space, 0xa0000000 - 0xbfffffff)： 与kseg0类似，这段地址也是内核地址，将虚拟地址的高 3 位清零(& 0x1fffffff)，就可以转换到物理地址，这段逻辑地址也是被连续地映射到物理内存的低端512M空间。但是 kseg1 不使用缓存(uncached)，访问速度比较慢， 但对硬件I/O寄存器来说，也就不存在Cache一致性的问题了，这段内存通常被映射到I/O寄存器，用来实现对外设的访问。

4.kseg2 (TLB-mapped cacheable kernel space, 0xc0000000 - 0xffffffff): 这段地址只能在内核态下使用，并且需要 MMU 的转换。

- 具体查看include/mmu.h

两种算法

- 2～3G: 直接映射(减去ULIM)
- 0～2G: 查表(页目录和页表)



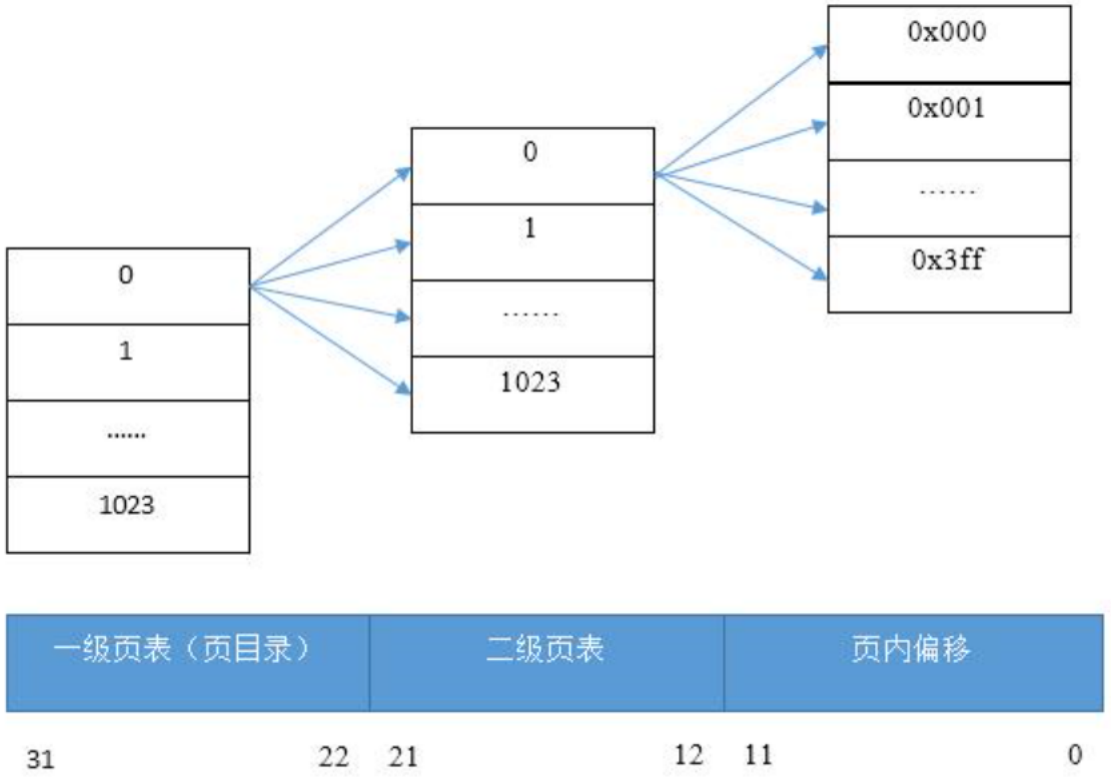
虚拟内存管理(1)

- kseg0 区域的地址转换：
  - 从虚拟地址到物理地址的转换只需要清掉 最高位的零即可，反过来，将对应范围内的物理地址转换到内核虚拟地址，也只需要将最高位设置为1即可。我们在 include/mmu.h 中定义了 PADDR 和 KADDR 两个宏来实验这一功能。

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

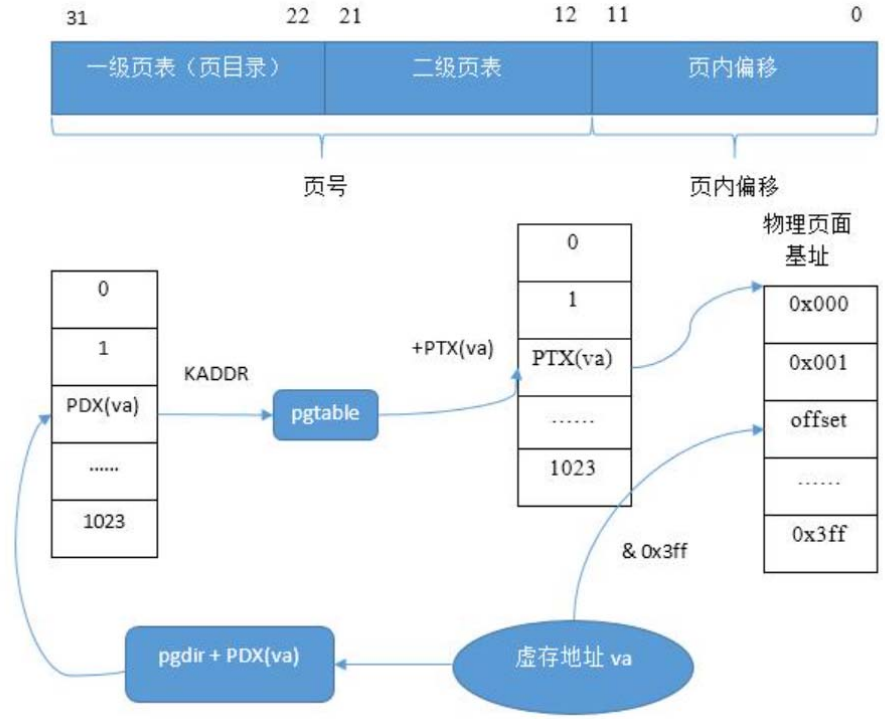
虚拟内存管理(2)

- 二级页表机制：



虚拟内存管理(3)

- 二级页表中虚拟地址到物理地址的转换：
- 虚拟地址->页目录项->页表项->物理页+页偏移->物理地址



虚拟内存管理(4)

```
1  static inline u_long
2  va2pa(Pde *pgdir, u_long va)
3  {
4      Pte *p;
5
6      pgdir = &pgdir[PDX(va)];
7
8      if (!(*pgdir & PTE_V)) {
9          return ~0;
10     }
11
12     p = (Pte *)KADDR(PTE_ADDR(*pgdir));
13
14     if (!(p[PTX(va)]&PTE_V)) {
15         return ~0;
16     }
17
18     return PTE_ADDR(p[PTX(va)]);
19 }
```

完成页表查询!

虚拟内存管理(5)

- 页目录的自映射:
  - 在二级页表结构中，页目录对应着二级页表，1024个页目录项存储的也是全部1024个页表的物理地址。也就是说，一个页表的内容和页目录的 内容是完全一样的，正是这种完全相同，使得将1024个页表加1个页目录映射到地址空间只需要4M的地址空间，其中的一个页表和页目录完全重合了。这就是页目录的自映射
  - 有了自映射机制，存储页表和页目录只需要4MB

虚拟内存管理(6)

- 页目录的自映射举例：
  - 页目录究竟在哪儿呢？
  - 我们知道，这 4M 空间的起始位置也就是第一个二级页表对应着页目录的第一个页目录项，同时，由于 1M 个页表项和 4G 地址空间是线性映射，不难算出 0x7fc00000 这一地址对应的应该是第 (0x7fc00000 >> 12) 个页表项，这个页表项 也就是第一个页目录项。一个页表项 32 位，占用 4 个字节的内存，因此，其相对于页表起始地址 0x7fc000000 的 偏移为 (0x7fc00000 >> 12) <<2 = 0x1ff000 ，于是得到地址为 0x7fc00000 + 0x1ff000 = 0x7fdff000 。也就是说，页目录的虚存地址为 0x7fdff000。

顶层初始化函数

```
void mips_init()
{
        printf("init.c:\tmips_init() is called\n");

        // Lab 2 memory management initialization functions
        mips_detect_memory();
        mips_vm_init();
        page_init();

        physical_memory_manage_check();// check if your physical memory manage code is correct
        page_check();// check if your memory manage code is correct

        panic("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");

        while (1);

        panic("init.c:\tend of mips_init() reached!");
}
```

物理内存管理

- 内存控制块——管理物理内存:

```
typedef LIST_ENTRY(Page) Page_LIST_entry_t;
struct Page {
        Page_LIST_entry_t pp_link;  /* free list
link */
        u_short pp_ref;
};
```

- 空闲内存链表——管理空闲物理内存:

```
LIST_HEAD(Page_list, Page);
static struct Page_list page_free_list; /*
Free list of physical pages */
```

实验任务1

- 在阅读queue.h 文件之后，相信你对宏函数的巧妙之处有了更深的体会。
- 比较与同学们在数据结构课上学的队列有什么不同?
  - LIST_ENTRY中， struct type **le_prev
- 请完成queue.h 中的LIST_INSERT_AFTER函数和LIST_INSERT_TAIL函数

实验任务2

- 在mips_detect_memory() 函数中初始化全局变量
- 比较简单，注意概念的理解
- 物理内存为64MB，页的大小为4KB。

首先，我们需要注意在 mm/pmap.c 中定义的与内存相关的全局变量：

```
1  u_long maxpa;          /* Maximum physical address */
2  u_long npage;          /* Amount of memory(in pages) */
3  u_long basemem;        /* Amount of base memory(in bytes) */
4  u_long extmem;         /* Amount of extended memory(in bytes) */
```

实验任务3

- 完成page_init 函数，使include/queue.h 中定义的宏函数将未分配的物理页加入到空闲链表page_free_list 中去.
- 操作系统内核必须的数据结构
  - 页目录（pgdir）、内存控制块数组（pages）和进程控制块数（envs）分配所需的物理内存。mips_vm_init() 函数实现了这一功能，并且使用boot_map_segment()完成了相关的虚拟内存与物理内存之间的映射。
- 仔细阅读已有代码，注意区别哪些内存已经被使用，哪些还没有被使用？

实验任务4

- 完成mm/pmap.c 中的page_alloc和page_free函数，基于空闲内存链表page_free_list，以页为单位进行物理内存的管理。
- 仔细阅读queue.h中定义的数据结构，理解各种MACRO的使用方法
- 仔细阅读pmap.h中定义的各种函数和数据结构，理解各个成员的作用
- 仔细阅读pmap.c中的相关代码

测试结果1

- 物理内存部分结果
  - 在init/init.c 的函数mips_init 中注释掉page_check();
  - make后运行“gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux”进行测试

```
main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
pmap.c:  mips vm init success
*temp is 1000
phycial_memory_manage_check() succeeded
panic at init.c:17: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

实验任务5

- 完成mm/pmap.c 中的boot_pgdir_walk 和 pgdir_walk 函数，实现虚拟地址到物理地址的转换以及创建页表的功能。
- 结合内存访问原理一节的讲解，区别什么时候需要用虚拟地址，什么时候需要用物理地址。
- 仔细阅读mmu.h和相关代码
- 概念模糊是常见的问题，需要多翻翻书

实验任务6

- mm/pmap.c 中的boot_map_segment 函数，实现将制定的物理内存与虚拟内存建立起映射的功能。
- 和上题一样，注意对二级页表机制的理解
- 利用已有的函数来完成

实验任务7

- 完成 mm/pmap.c 中的 page_insert 函数。
  - Step1:先判断va 是否有效，如果va 已经有了映射的物理地址的话，则去判断这个物理地址是不是我们要插入的那个物理地址。如果不是，那么就把该物理地址移除掉；如果是，则修改权限，放到tlb 中，返回。
  - Step2:更新TLB，应该注意到只要对页表的内容有修改，都必须用tlb_invalidate来让tlb 更新，否则后面紧接着对内存的访问很有可能出错。
  - Step3:重新获取页面以检查是否插入，若成功插入则设置相关标记位。否则报错。
- 最重要的函数。
  - pgdir_walk、pa2page、page_remove、tlb_invalidate、page2pa

实验任务8

- 完成 mm/tlb_asm.S 中 tlb_out 函数。
- 这个函数的作用是使得某一虚拟地址对应的tlb表项失效。从而下次访问这个地址的时候诱发tlb重填，以保证数据更新。
- 重点是理解tlbp和tlbwi两条指令的含义以及大致的执行流程。

测试结果2

- 虚拟内存部分结果
  - 在init/init.c 的函数mips_init 中，page_check 还原，并注释掉 physical_memory_manage_check();。
  - make后运行“gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux”进行测试
  - 运行的正确结果：

```
main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
mips_vm_init:boot_pgdir is 80400000
pmap.c:  mips vm init success
start page_insert
va2pa(boot_pgdir, 0x0) is 3ffe000
page2pa(pp1) is 3ffe000
pp2->pp_ref 0
end page_insert
page_check() succeeded!
panic at init.c:55: ----------------------------------------
```