CHAPTER 2

内存管理

2.1 实验目的

1. 了解内存访问原理
2. 了解 MIPS 内存映射布局
3. 掌握使用空闲链表的管理物理内存的方法
4. 建立页表，实现分页式虚存管理
5. 实现内存分配和释放的函数

本次实验中，我们需要掌握 MIPS 页式内存管理机制，需要使用一些数据结构来记录内存的使用情况，并实现内存分配和释放的相关函数，完成物理内存管理和虚拟内存管理。

2.2 MMU、TLB 和内存访问

本章教程的题目是内存管理，而在进行内存管理之前，我们首先需要知道内存访问的整体原理。我们首先介绍内存翻译中最重要的两个部件：MMU 和 TLB，之后介绍内存访问的整体流程。

2.2.1 MMU

根据我们在“计算机组成”和“操作系统”这两门课中学到的知识，我们知道当 CPU 发出一个访存的指令，需要把虚拟地址翻译成物理地址才能对内存进行访问，而通过对页表相关知识的学习你肯定也对虚拟地址到物理地址的转换过程有了大致的了解，通常把这个过程称为内存翻译，在计算机中负责完成这一任务的部件是 MMU。

MMU 的全称是 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虚拟地址，并提供了一套硬件机制来实现内存访问的权限检查，它的位置和功能如下图2.1所示

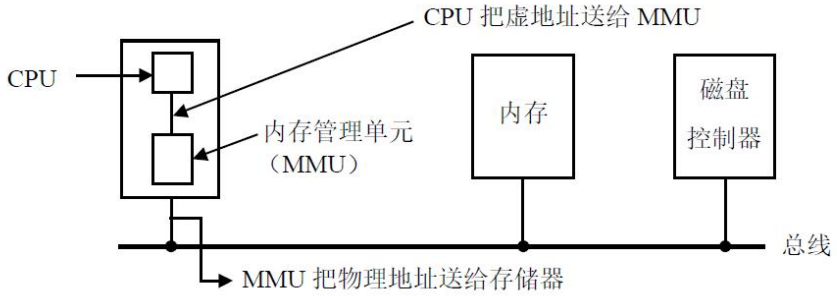

图 2.1: MMU 的位置和功能

我们所熟悉的查询二级页表访存机制，即先查询页目录，再查询相应的二级页表的工作，都是由 MMU 来完成的。

2.2.2 TLB

我们知道，访存效率在体系结构的设计中是一个很重要的问题，但我们应用的页表机制却会降低系统的性能，举例来说，如果没有页表，那么只需要一次访存，但如果有了二级页表，就需要三次访存。虽然页表机制能带来许多好处，但是这样降低效率还是无法让人接受的，为了解决这一问题，我们需要一个能让计算机能够不经过页表就把虚拟地址映射成物理地址的硬件设备，这就是 TLB。

TLB 的全称是 Translation Lookside Buffer，中文为翻译后援存储器，有的时候也叫相联存储器 (Associative Memory)，或者快表，它通常被安装在 MMU 的内部，如下图2.2所示：

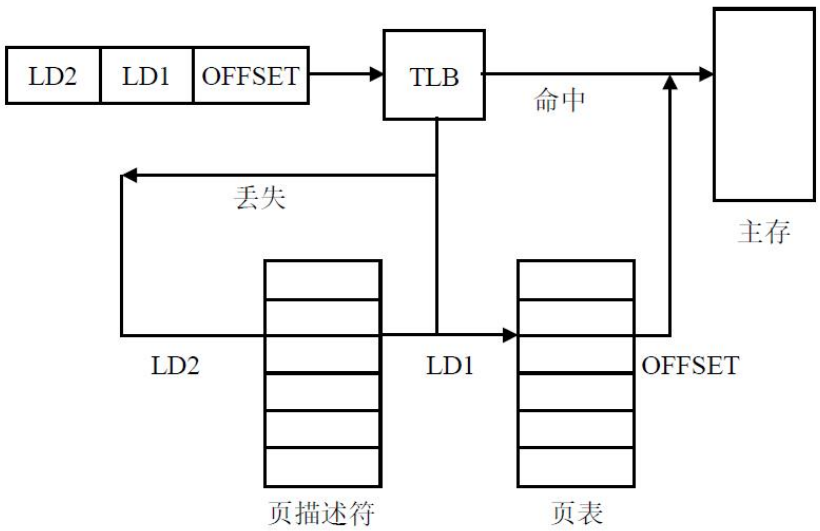

图 2.2: TLB 示意图

简单来说，TLB 就是页表的高速缓存，每个 TLB 的条目中包含有一个页面的所有信息 (有效位、虚页号、物理页号、修改位、保护位等等)，这些条目中的内容和页表中相同页面的条目中的内容是完全一致的。

当一个虚拟地址被送到 MMU 中进行翻译的时候，硬件首先在 TLB 中寻找包含这个地址的页面，如果它的虚页号在 TLB 中，并且没有违反保护位，那么就可以直接从 TLB 中得到相应的物理页号，而不去访问页表；如果发现虚页号在 TLB 中不存在，那么 MMU 将进行常规的页表查找，同时通过一定的策略来将这一页的页表项替换到 TLB 中，之后再次访问这一页的时候就可以直接在 TLB 中找到。

容易发现，如果 TLB 命中，那么我们只需要访问两次内存就可以得到需要的数据，从而从整体上降低了平均访存时间。

到此为止，我们已经基本了解了内存访问的机制。

2.2.3 内存访问

在这之前，我们一直忽略了访存的一个重要部件————cache，相信你已经多次在课堂上接触过这个概念，计算机中的访存过程中一定存在着和 cache 相关的过程，而之前对 cache 的忽略就意味着我们对内存访问机制的了解是不全面的，这就产生了下面的三个问题:

1. TLB、cache、MMU、页表这些东西之间有什么关系?
2. 一个完整的访存流程是怎样的?
3. 我们在这次实验中要完成的内存管理在这个流程中起了什么作用?

这一节中我们将分别回答这些问题，在你往下看之前，也许你可以先思考一下这些问题，看看下面给出的解答是否符合你的预期。

接下来的讲解将会涉及到 cache 中的一些概念，我们假设你对这些概念都有一定的了解，如果你感觉有些不确定或者忘了这些内容，你可以回头翻翻计组课本中的相关内容。

一般来说，cache 中用来查询相应地址是否存在时所用的地址是物理地址的一部分，这就是 cache 中的 tag，如果是这样，那么很自然的，访存时候所用的虚拟地址应该首先经过 TLB 和 MMU 转换成物理地址，然后在 cache 中查找是否命中，基于这样的考虑，那么完整的访存流程应该是这样的:

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。

看上去上面的访存过程没什么问题，但如果上述过程是按顺序执行的，那么自然就产生了问题，每次访存都需要先经过 TLB，如果 TLB 命中还能接受，可如果 TLB 没有命中，同时这个地址又恰好在 cache 中存在，那么这种情况下先将地址经过 MMU 翻译再在 cache 中查询岂不是会造成性能上的损失？

我们把这个问题留给你来思考。

Thinking 2.1 请思考 cache 用虚拟地址来查询的可能性，并且给出这种方式对访存带来的好处和坏处。另外，你能否能根据前一个问题的解答来得出用物理地址来查询的优势? ■

虽然提出了相应的疑问，但是我们可以认为上述过程是大致正确的，当然，这里还有一个问题留给你思考。

Thinking 2.2 请查阅相关资料，针对我们提出的疑问，给出一个上述流程的优化版本，新的版本需要有更快的访存效率。(提示：考虑并行执行某些步骤) ■

到此，我们已经解决了前面提出的前两个问题。

回想前面的访存过程我们可以发现有一个过程是我们没有提到的，即页表内容的填充。虽然 MMU 可以自动访问页表得到虚拟地址相应的物理地址，但如果只有一个空荡荡的页表，那么 MMU 是无法工作的；同时，操作系统需要在软件层面上对内存进行管理和控制，以便为上层的应用提供相应的接口。我们这次的实验就是围绕着这些内容来展开的。

Thinking 2.3 在我们的实验中，有许多对虚拟地址或者物理地址操作的宏函数 (详见 include/mmu.h), 那么我们在调用这些宏的时候需要弄清楚需要操作的地址是物理地址还是虚拟地址，阅读下面的代码，指出 x 是一个物理地址还是虚拟地址。

```
1 ^^Iint x;
2 ^^Ichar* value = return_a_pointer();
3 ^^I*value = 10;
4 ^^Ix = (int) value;
```

■

2.3 MIPS 虚存映射布局

32 位的 MIPS CPU 最大寻址空间为 4GB(2^32 字节)，这 4GB 虚存空间被划分为四个部分：

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。

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。

2.4 内存管理与内存翻译

内存管理的实质是为每个程序提供自己的内存空间。最为朴素的内存管理就是直接将物理内存分配给程序，一个萝卜一个坑。但从之前的叙述中，我们可以发现，MIPS 系统中，使用了虚拟内存的技术。因此在我们的系统中，同时存在着两套地址，一套是真实的物理地址，另一套则是虚拟地址，这为我们的内存管理带来了许多内存翻译的工作。这些工作大多就是由前文所述的 MMU 来完成。为什么我们要这么做呢？原因有很多：

1. 隐藏与保护：因为加入了虚拟内存这一中间层，真实的物理地址对用户级程序是不可见的，它只能访问操作系统允许其访问的内存区域。

2. 为程序分配连续的内存空间：利用虚拟内存，操作系统可以从物理分散的内存页中，构建连续的程序空间，这使得我们拥有更高的内存利用率。

3. 扩展地址空间范围：如前文所述，通过虚拟内存，MIPS32 位机拥有了 4GB 的寻址能力，这真的很 cool。:)

4. 使内存映射适合你的程序：在大型操作系统中，可能存在相同程序的多个副本同时运行，这时候通过内存翻译这一中间层，你能使他们都使用相同的程序地址，这让很多工作都简单了很多。

5. 重定位：程序入口地址和预先声明的数据在程序编译的过程中就确定了。但通过 MMU 的内存翻译，我们能够让程序运行在内存中的任何位置。

为了这些好处，我们需要付出地址翻译工作的代价。接下来我们在 lab 的工作中，也将一直遇到这个问题。建议各位在 lab 的过程中，不妨思考当前我们使用的这个地址究竟是物理地址还是虚拟地址，搞清楚这一点，对我们的 lab 和操作系统的理解都大有帮助。

2.5 物理内存管理

2.5.1 初始化流程说明

在第一个实验中，我们将内核加载到内存中的 kseg0 区域 (0x80010000)，成功启动并跳转到 init/main.c 中的 main 函数开始运行，现在我们需要在 main 函数中调用定义在 init/init.c 中的 mips_init() 函数，并进一步通过

1. `mips_detect_memory();`
2. `mips_vm_init();`
3. `page_init();`

这三个函数来实现物理内存管理的相关数据结构的初始化。

2.5.2 内存控制块

在 MIPS CPU 中，地址转换以 4KB 大小为单位，称为页。整个物理内存按 4KB 大小分成了许多页，我们大多数时候的内存分配，也是以页为单位来进行。为了记录分配情况，我们需要使用 Page 结构体来记录一页内存的相关信息：

```
1  typedef LIST_ENTRY(Page) Page_LIST_entry_t;
2
3  struct Page {
4      Page_LIST_entry_t pp_link;  /* free list link */
5      u_short pp_ref;
6  };
```

其中，`pp_ref` 用来记录这一物理页面的引用次数，`pp_link` 是当前节点指向链表中下一个节点的指针，其类型为 `LIST_ENTRY(Page)` 。我们在 include/queue.h 中定义了一系列的宏函数来简化对链表的操作。请阅读这些宏函数的代码，理解它们的原理和巧妙之处。

Thinking 2.4 我们注意到我们把宏函数的函数体写成了 `do { /* ... */ } while(0)` 的形式，而不是仅仅写成形如 `{ /* ... */ }` 的语句块，这样的写法好处是什么？ ■

Exercise 2.1 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。请完成 queue.h 中的`LIST_INSERT_AFTER`函数和`LIST_INSERT_TAIL`函数。 ■

在 include/pmap.h 中，我们使用 LIST_HEAD 宏来定义了一个结构体类型 Page_list，在 mm/pmap.c 中，创建了一个该类型的变量 page_free_list 来以链表的形式表示所有的空闲物理内存：

```
LIST_HEAD(Page_list, Page);

static struct Page_list page_free_list; /* Free list of physical pages */
```

Thinking 2.5 注意，我们定义的 Page 结构体只是一个信息的载体，它只代表了相应物理内存页的信息，它本身并不是物理内存页。那我们的物理内存页究竟在哪呢？Page 结构体又是通过怎样的方式找到它代表的物理内存页的地址呢？请你阅读 include/pmap.h 与 mm/pmap.c 中相关代码，给出你的想法。 ■

Thinking 2.6 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 (请注意指针)。

```
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;
}
```

■

2.5.3 内存分配和释放

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：

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

Exercise 2.2 我们需要在 mips_detect_memory() 函数中初始化这几个全局变量，以确定内核可用的物理内存的大小和范围。根据代码注释中的提示，完成 mips_detect_memory() 函数。 ■

在操作系统刚刚启动时，我们还没有建立起有效的数据结构来管理所有的物理内存，因此，出于最基本的内存管理的需求，我们需要实现一个函数来分配指定字节的物理内存。这一功能由 mm/pmap.c 中的 alloc 函数来实现。

```
static void *alloc(u_int n, u_int align, int clear);
```

alloc 函数能够按照参数 align 进行对齐，然后分配 n 字节大小的物理内存，并根据参数 clear 的设定决定是否将新分配的内存全部清零，并最终返回新分配的内存的首地址。

Thinking 2.7 在 mmu.h 中定义了`bzero(void *b, size_t)`这样一个函数，请你思考，此处的 b 指针是一个物理地址，还是一个虚拟地址呢？ ■

有了分配物理内存的功能后，接下来我们需要给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）和进程控制块数组（envs）分配所需的物理内存。mips_vm_init() 函数实现了这一功能，并且完成了相关的虚拟内存与物理内存之间的映射。

完成上述工作后，我们便可以通过在 mips_init() 函数中调用 page_init() 函数将余下的物理内存块加入到空闲链表中。

Exercise 2.3 完成 page_init 函数，使用 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。思考如何区分已分配的内存块和未分配的内存块，并注意内核可用的物理内存上限。 ■

有了记录物理内存使用情况的链表之后，我们就可以不再像之前的 alloc 函数那样按字节为单位进行内存的分配，而是可以以页为单位进行物理内存的分配与释放。page_alloc 函数用来从空闲链表中分配一页物理内存，而 page_free 函数则用于将一页之前分配的内存重新加入到空闲链表中。

Exercise 2.4 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。并在 init/init.c 的函数 mips_init 中注释掉 page_check();。此时运行结果如下。 ■

```
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: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

至此，我们的内核已经能够按照分页的方式对物理内存进行管理，

2.6 虚拟内存管理

我们通过建立两级页表来进行虚拟内存的管理，在此基础上，我们将实现根据虚拟地址在页表中查找对应的物理地址，以及将一段虚存地址映射到一段的物理地址的功能，然后实现虚存的管理与释放，最后为内核建立起一套虚存管理系统。

2.6.1 两级页表机制

我们的操作系统内核采取二级页表结构，如图2.3所示：

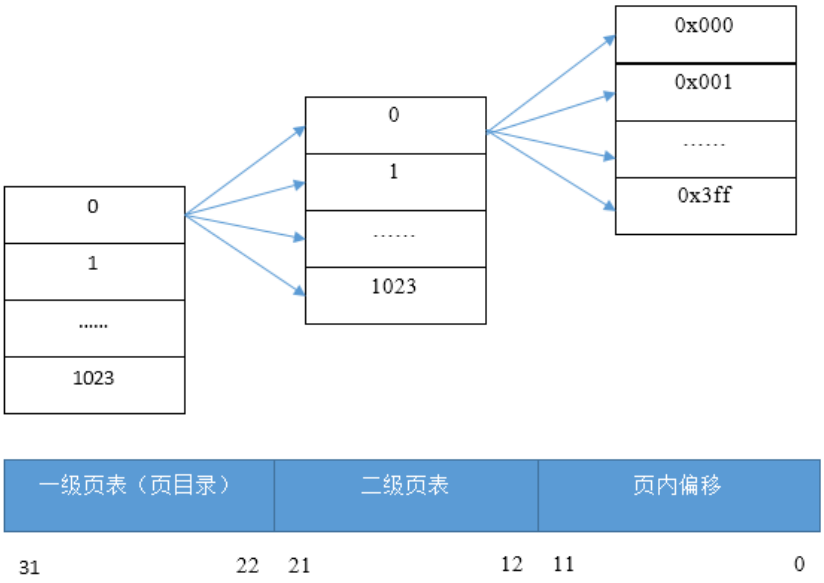

图 2.3: 二级页表结构示意图

第一级表称为页目录 (page directory)，一共 1024 个页目录项，每个页目录项 32 位 (4 Byte)，页目录项存储的值为其对应的二级页表入口的物理地址。整个页目录存放在一个页面 (4KB) 中，也就是我们在 mips_vm_init 函数中为其分配了相应的物理内存。第二级表称为页表 (page table)，每一张页表有 1024 个页表项，每个页表项 32 位 (4 Byte)，页表项存储的是对应页面的页框号 (20 位) 以及标志位 (12 位)。每张页表占用一个页面大小 (4KB) 的内存空间。

对于一个 32 位的虚存地址，其 31-22 位表示的是页目录项的索引，21-12 位表示的是页表项的索引，11-0 位表示的是该地址在该页面内的偏移。

2.6.2 地址转换

对于操作系统来说，虚拟地址与物理地址之间的转换是内存管理中非常重要的内容。在这一部分，我们将详细探讨咱们的内核是如何进行地址转换的。

首先从较为简单的形式开始。在前面的实验中，我们通过设置 lds 文件让操作系统内核加载到内存的 0x80010000 位置，在上文我们对 MIPS 存储器映射布局的介绍中我们知道，这一地址对应的是 kseg0 区域，这一部分的地址转换不通过 MMU 进行。我们也称这一部分虚拟地址为内核虚拟地址。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：

```
// translates from kernel virtual address to physical address.
#define PADDR(kva)            \
  ({                  \
    u_long a = (u_long) (kva);        \
    if (a < ULIM)           \
      panic("PADDR called with invalid kva %08lx", a);\
    a - ULIM;             \
  })

// translates from physical address to kernel virtual address.
#define KADDR(pa)             \
  ({                  \
    u_long ppn = PPN(pa);           \
    if (ppn >= npage)           \
      panic("KADDR called with invalid pa %08lx", (u_long)pa);\
    (pa) + ULIM;            \
  })
```

在 PADDR 中，我们使用了一个宏 ULIM ，这个宏定义在 include/mmu.h 中，其值为 0x80000000。对于小于 0x80000000 的虚拟地址值，显然不可能是内核区域的虚拟地址。在 KADDR 中，一个合理的物理地址的物理页框号显然不能大于我们在 mm/pmap.c 中所定义的物理内存总页数 npage 的值。

接下来，我们讨论如何通过二级页表进行虚拟地址到物理地址的转换。

首先，我们可以通过 PDX(va) 来获得一个虚拟地址对应的页目录索引，然后直接凭借索引在页目录中得到对应的二级页表的基址 (物理地址)，然后把这个物理地址转化为内核虚拟地址 (KADDR)，之后，通过 PTX(va) 获得这个虚存地址对应的页表索引，然后就可以从页表中得到对应的页面的物理地址。整个转换的过程如图2.4 所示：

2.6.3 页目录自映射

上文中我们讲二级页表结构的时侯提到，要映射整个 4G 地址空间，一共需要 1024 个页表和 1 个页目录的。一个页表占用 4KB 空间，页目录也占用 4KB 空间，也就是说，整个二级页表结构将占用 4MB+4KB 的存储空间，1024*4KB+1*4KB=4MB+4KB。

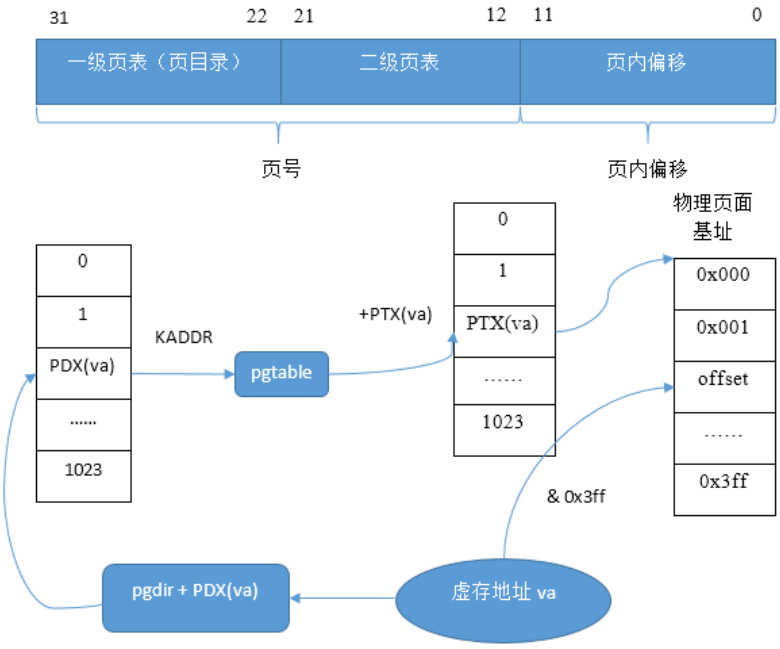

图 2.4: 地址转换过程

在 include/mmu.h 中的内存布局图里有这样的内容：

```
/**
 *      ULIM     -----> +----------------------------+------------0x8000 0000-------
 *                      |         User VPT           |     PDMAP                  /|\
 *      UVPT     -----> +----------------------------+------------0x7fc0 0000    |
 */
```

不难计算出 UVPT(0x7fc00000) 到 ULIM(0x80000000) 之间的空间只有 4MB ，这一区域就是进程的页表的位置，于是我们不禁想问：页目录所占用的 4KB 内存空间在哪儿？

答案就在于页目录的自映射机制！

如果页表和页目录没有被映射到进程的地址空间中，而一个进程的 4GB 地址空间又都映射了物理内存的话，那么就确实需要 1024 个物理页 (4MB) 来存放页表，和另外 1 个物理页 (4KB) 来存放页目录，也就是需要 (4M+4K) 的物理内存。但是页表也被映射到了进程的地址空间中，也就意味着 1024 个页表中，有一个页表所对应的 4M 空间正是这 1024 个页表所占用的 4M 内存，这个页表的 1024 的页表项存储了 1024 个物理地址，分别是 1024 个页表的物理地址。而在二级页表结构中，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**。

接下来我们会想到这样一个问题：那么，这个与页目录重合的页表，也就是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 0x7fc00000 的偏移为 (0x7fc00000 >> 12) * 4 = 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。

Thinking 2.8 了解了二级页表页目录自映射的原理之后，我们知道，Win2k 内核的虚存管理也是采用了二级页表的形式，其页表所占的 4M 空间对应的虚存起始地址为 0xC0000000，那么，它的页目录的起始地址是多少呢？ ■

2.6.4 创建页表

将虚拟地址转换为物理地址的过程中，如果这个虚拟地址所对应的二级页表不存在，有时，我们可能需要为这个虚拟地址创建一个新的页表。我们需要申请一页物理内存来存放这个页表，然后将他的物理地址赋值给对应的页目录项，最后设置好页目录项的权限位即可。

我们的内核在 mm/pmap.c 中分定义了 boot_pgdir_walk 和 pgdir_walk 两个函数来实现地址转换和页表的创建，这两个函数的区别仅仅在于当虚存地址所对应的页表的页面不存在的时候，分配策略的不同和使用的内存分配函数的不同。前者用于内核刚刚启动的时候，这一部分内存通常用于存放内存控制块和进程控制块等数据结构，相关的页表和内存映射必须建立，否则操作系统内核无法正常执行后续的工作。然而用户程序的内存申请却不是必须满足的，当可用的物理内存不足时，内存分配允许出现错误。boot_pgdir_walk 被调用时，还没有建立起空闲链表来管理物理内存，因此，直接使用 alloc 函数以字节为单位进行物理内存的分配，而 pgdir_walk 则在空闲链表初始化之后发挥功能，因此，直接使用 page_alloc 函数从空闲链表中以页为单位进行内存的申请。

上文中介绍了页目录自映射的相关知识，我们了解到页目录其实也是一个页表。我们知道，在咱们实验使用的内核中，一个页表指向的物理页面存在的标志是页表项存储的值的 PTE_V 标志位被置为 1 。因此，在将页表的物理地址赋值给页目录项时，我们还为页目录项设置权限位。

Exercise 2.5 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。 ■

2.6.5 地址映射

一个空荡荡的页表自然不会对我们的内存翻译有帮助，我们需要在具体的物理内存与虚拟地址间建立映射，即将相应的物理页面地址填入对应虚拟地址的页表项中。

Exercise 2.6 实现 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。 ■

2.6.6 page insert and page remove

```
// Overview:
//   Map the physical page 'pp' at virtual address 'va'.
//   The permissions (the low 12 bits) of the page table entry
//   should be set to 'perm|PTE_V'.
//
// Post-Condition:
//   Return 0 on success
//   Return -E_NO_MEM, if page table couldn't be allocated
//
// Hint:
//   If there is already a page mapped at `va`, call page_remove()
//   to release this mapping.The `pp_ref` should be incremented
//   if the insertion succeeds.
int
page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
{
^^Iu_int PERM;
^^IPte *pgtable_entry;
^^IPERM = perm | PTE_V;

^^I/* Step 1: Get corresponding page table entry. */
^^Ipgdir_walk(pgdir, va, 0, &pgtable_entry);

^^Iif (pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0) {
^^I^^Iif (pa2page(*pgtable_entry) != pp) {
^^I^^I^^Ipage_remove(pgdir, va);
^^I^^I}
^^I^^Ielse{
^^I^^I^^Itlb_invalidate(pgdir, va);
^^I^^I^^I*pgtable_entry = (page2pa(pp) | PERM);
^^I^^I^^Ireturn 0;
^^I^^I}
^^I}

^^I/* Step 2: Update TLB. */

^^I/* hint: use tlb_invalidate function */

^^I/* Step 3: Do check, re-get page table entry to validate the insertion. */
^^I
^^I/* Step 3.1 Check if the page can be insert, if can' t return -E_NO_MEM */

^^I/* Step 3.2 Insert page and increment the pp_ref */

^^Ireturn 0;
}
```

这个函数**非常重要**！它将在 **lab3 和 lab4** 中被反复用到，这个函数将 va 虚拟地址和其要对应的物理页 pp 的映射关系以 perm 的权限设置加入页目录。我们大概讲一下函数的执行流程与执行要点。

流程大致如下：先判断 va 是否有对应的页表项: 如果页表项有效（或者叫 va 是否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；如果是的话，则修改权限，放到 tlb 中。

有一个值得指出的要点:我们能看到,只要对页表的内容修改,都必须 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。

可以说 tlb_invalidate 函数是它的一个核心子函数，这个函数实际上又是由 tlb_out 汇编函数组成的。

Exercise 2.7 完成 mm/pmap.c 中的 page_insert 函数。 ■

Listing 8: TLB 汇编函数

```
1  #include <asm/regdef.h>
2  #include <asm/cp0regdef.h>
3  #include <asm/asm.h>
4
5  LEAF(tlb_out)
6          nop
7          mfc0        k1,CP0_ENTRYHI
8          mtc0        a0,CP0_ENTRYHI
9          nop
10         // insert tlbp or tlbwi
11         nop
12         nop
13         nop
14         nop
15         mfc0        k0,CP0_INDEX
16         bltz        k0,NOFOUND
17         nop
18         mtc0        zero,CP0_ENTRYHI
19         mtc0        zero,CP0_ENTRYLO0
20         nop
21         // insert tlbp or tlbwi
22 NOFOUND:
23
24         mtc0        k1,CP0_ENTRYHI
25
26         j        ra
27         nop
28 END(tlb_out)
```

这个汇编函数相对其他汇编函数来说相对简单，在空出的两行位置，你需要填写 tlbp 或 tlbwi 指令，那么留给你思考几个问题.

Thinking 2.9 思考一下 tlb_out 汇编函数，结合代码阐述一下跳转到 **NOFOUND** 的流程？从 MIPS 手册中查找 tlbp 和 tlbwi 指令，明确其用途，并解释为何第 10 行处指令后有 4 条 nop 指令。 ■

Exercise 2.8 完成 mm/tlb_asm.S 中 tlb_out 函数。 ■

2.6.7 访存与 TLB 重填

通过之前的实验，我们可以知道，虚拟地址通过 MMU 转换成物理地址，然后通过物理地址我们能够在主存中获得相应的数据。而实际上，在 MIPS 架构中，关于这一块内存翻译的内容，很大程度上与 TLB 有关。TLB 可以看做是一块页表的高速缓存，里

面存储了一些物理页面与虚拟页面的对应关系。而当 CPU 访问相应内存地址时，会先去 TLB 中查询。当 TLB 中没有相应对应关系时会触发一个 **TLB 缺失异常**。而 MIPS 将这个异常的处理，全权交给了软件。因此若发生缺失异常，则会跳转到相应异常处理程序中，再由我们的二级页表进行相应的地址翻译，对 TLB 进行重填。换句话说，MIPS 中并没有一个执行内存地址翻译的 MMU 处理器，CPU 完成了相应工作。

如果大家仔细地理解了上面这段话，就会发现 MMU 的正常工作需要**异常处理**的支持。由于我们暂时还不了解异常的相关内容，因此在本次实验中实际上并没有真正地启用 MMU，而只是先创建了页表。在 lab3 中学习完中断和异常后，我们就能够见到一个真正启用的 MMU 了。

而现在，为了深究整个过程，让我们来一起探索一个假设情境：在 page_check 最后一句 printf 之前添加如下代码段。提醒一下，由于 MMU 暂时没有启用，这段代码目前不能在我们的系统中实际运行：

```
    u_long* va = 0x12450;
    u_long* pa;

    page_insert(boot_pgdir, pp, va, PTE_R);
    pa = va2pa(boot_pgdir, va);
    printf("va: %x -> pa: %x\n", va, pa);
    *va = 0x88888;
    printf("va value: %x\n", *va);
    printf("pa value: %x\n", *((u_long *)((u_long)pa + (u_long)ULIM)));
```

这段代码旨在计算出相应 va 与 pa 的对应关系，设置权限位为 PTE_R 是为了能够将数据写入内存。

如果 MMU 能够正常工作，实际输出将会是：

```
    va: 12450 -> pa: 3ffd000
    va value: 88888
    pa value: 0
    page_check() succeeded!
```

Thinking 2.10 显然，运行后结果与我们预期的不符，va 值为 0x88888，相应的 pa 中的值为 0。这说明我们的代码中存在问题，请你仔细思考我们的访存模型，指出问题所在。 ■

另外，还可以提醒大家的是，在 gxemul 中，有 tlbdump 这个命令，可以随时查看 tlb 中的内容。

Thinking 2.11 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，请查阅相关资料，说明当 PSE 开启时的页表组织形式与我们当前的页表组织形式的区别。 ■

2.7 正确结果展示

实验二做完之后，在 init/init.c 的函数 mips_init 中，将 page_check 还原，并注释掉 physical_memory_manage_check();。运行的正确结果应该是这样的：

```
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: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

地址 **3ffe000** 和最后一行的数字 **55** 是不固定的。

2.8 实验思考

- 思考-不同地址类型查询 cache 的比较
- 思考-优化访存过程
- 思考-地址类型判断
- 思考-使用 do-while(0) 语句的好处
- 思考-Page 结构体与物理内存页
- 思考-bzero 参数探究
- 思考-自映射机制页目录地址的计算
- 思考-NOFOUND 的奥妙及两条 MIPS 指令
- 思考-访存的问题
- 思考-大物理页面