第四章.内存管理（lab2）(v0.1)

4.1. 实验目标

MIT 这次实验的目标，是在他们的 JOS 操作系统中实现内存分页管理的功能。

程序的几乎所有代码都集中在 pmap.c 文件中。实际上，该实验可以分为 2 部分：物理页面管理和页表管理。前者强调对机器拥有的物理内存的管理，包括建立对应的数据结构、处理分配和回收动作等；而后者主要是强调利用 Intel x86 系列处理器的页式地址管理功能，完成（虚拟）线性地址到物理地址的转换，包括建立页目录、页表等。

在物理页面管理上，需要完成的函数包括：

boot_alloc()

page_init()

page_alloc()

page_free()

在页表管理上，需要完成的函数包括：

pgdir_walk()

boot_map_segment()

page_lookup()

page_remove()

page_insert()

而在实验过程中，我们发现，为了保证大家自己写的程序段的正确性，该实验安排了一些检查函数，以进行阶段性地检查，如果这些检查函数发现大家写的程序不符合实验原来的设想的话（主要是一堆的 assert），就会提前 panic 掉。这是一个非常好的方法，即使只写了一小段程序，都可以让这些检查程序进行一下检查，以确保走的是正确的路。

主要的检查函数有：

check_page_alloc(); 该函数检查用于管理物理内存的数据结构（在该实验中是一个双向链表）是否分配和安排妥当；

page_check(); 该函数检查页表管理是否正确；

check_boot_pgdir(); 该函数用于检查（虚拟）线性内存地址是否都按照实验设想的那样被页式地址转换机制映射到了正确的物理内存上。

4.2. 背景知识

通过实验一，我们知道，JOS 的启动过程实际上是先把 bootsector 的内容读到 0x7c00 处（注意，bootsector 的代码在编译的时候已经故意地把逻辑地址的首地址定在了 0x7c00 上），bootsector 中的代码开始执行后，会从磁盘上紧接着自己的第 2 个扇区开始，一直读 8 个扇区的内容（一共是 8×512=4KB，ELF 头的大小）到 0x10000（64KB）的地方，然后，通过对 ELF 头的解析，得到 kernel 模块编译出来后所占的大小，并将 kernel 读到物理内存 0x100000（1MB）开始的地方。然后设置好 GDT，并调用 i386_init()函数，而 i386_init()函

数在将自己的 BSS 区域清零后，调用 cons_init()函数设置好屏幕显示设备为 cprintf 的运行做好准备后就调用 i386_detect_memory();函数和 i386_vm_init();，这里前者的功能主要是读 CMOS 取得物理内存的实际大小，而后者就是实验 2 的主要函数了。当然 i386_init()函数最后会调用 monitor(NULL);并进入循环，处理用户通过终端输入的命令。

在调用 i386_init()函数以前，内存的 layout 可以用图 4-1 来表示：

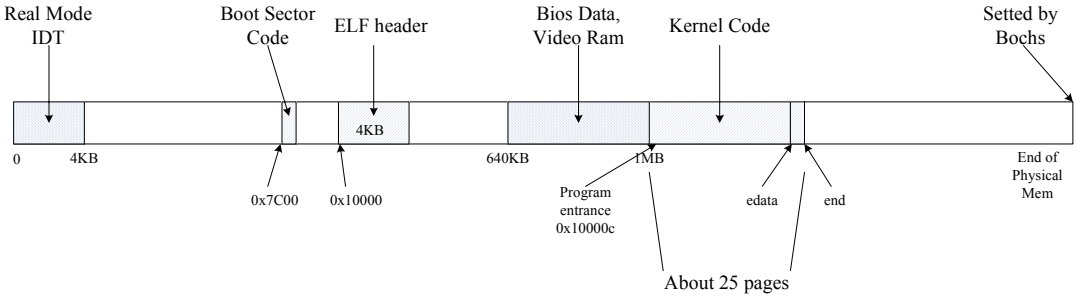

图 4-1. 调用 i386_init()函数以前内存的 layout

在调用 i386_init()后，系统将重载 GDT，新的 GDT：

```
SEG_NULL                    # null seg
SEG(STA_X|STA_R, -KERNBASE, 0xffffffff)    # code seg
SEG(STA_W, -KERNBASE, 0xffffffff)# data seg
```

注意新 GDT 后两项的 base，它们是-KERNBASE！如果 KERNBASE=0xF0000000，则 GDT 的 base=0x10000000，通过保护模式的段式地址转换机制（即 Physical Address=Logical Address + base），将程序内 KERNBASE 开始的逻辑地址转换成物理地址。

4.3. 内存分页机制的实现

在这个实验中，内存的分页机制实际上包含页面管理和页表管理两个部分。对于前一个部分，主要是讨论内核如何在如图 4-1 所示的物理内存中分配空间和建立合理的数据结构来对物理页面进行管理，而后一个部分主要是讨论如何建立两级页目录（Page directory）和页表（Page table），从而在开启 x86 的分页管理后仍然能够进行合理的逻辑地址→物理地址的转换。

1 页面管理

在 JOS 系统中，对于物理内存的页面实际上是通过双向链表来进行管理的。

在讨论页面管理之前，有几个常识性的问题需要先澄清一下。首先，是 i386 的页面大小问题。虽然在很多操作系统书讲到页式地址管理的时候总会说页面大小是可以根据系统的安排而调整的，一般是 1KB 的整数倍，如 1KB，2KB，4KB 或者 8KB 等。但是，在 32 位的 i386 上，页面的大小是固定的 4KB！当然，在 Extended Paging 模式下（在 Pentium 以后的处理器中进行了支持），为了寻址更大的空间，页面的大小甚至可以为 4MB，虽然这种模式在本实验以及后续实验中应该不会碰到。

其次，既然确定了页面大小，那么，给定一个大小的物理内存（例如 2MB、4MB 或者 16MB 甚至更多，在我们的实验中可以在 Bochs 的配置文件中给定），一共有多少个页面呢？是算图 4-1 中的物理内存空间还是算所有的空间？试想，如果一个页面管理机制只管理图 4-1 中的空闲物理内存空间的话，已经存在在内存中的内核代码就无法通过页式地址转换来

进行逻辑地址→物理地址的转换了，这种情况下，内核代码的运行就成了问题！所以，答案是：在页面管理上，系统必须管理所有的物理内存空间。同时，需要对图 4-1 中不同的物理内存区域进行区别对待：对于已经存在在内核中的一些实模式数据（如 Real Mode IDT）和系统区域（如 BIOS 数据，显存区域等），系统不能把它们当作空闲内存分配出去给其他进程使用；而对于内核本身所在的区域，当然也不能把它当做空闲区分配出去给其他进程使用，而且同时，还需要建立起合适的映射关系，达到将逻辑地址转换到物理地址的目标（始终记住：内核程序的逻辑地址是从 KERNBASE=0xF0000000 开始的！）。在确定了页面大小和分页的范围后，物理内存里应该分多少个页面就很清楚了：应该拿物理内存的大小去整除 4KB！简单的说，页面的数目可以用以下等式表达：

页面的数目（npage） = 物理内存的大小>>12

下面，我们分别对链表的结构和链表在内存中的存储和放置进行讨论。

1) 页面管理链表的结构

现在我们来讨论用于页面管理的双向链表结构。首先，我们来看一下构成这个链表的结点的情况。该结点的结构是在 memlayout.h 中规定的：

```
typedef LIST_ENTRY(Page) Page_LIST_entry_t;
struct Page {
    Page_LIST_entry_t pp_link;  /* free list link */
    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.
    uint16_t pp_ref;
};
```

而 LIST_ENTRY 的定义则在 queue.h 中：

```
#define    LIST_ENTRY(type)                                \
struct {                                                   \
    struct type *le_next;      /* next element */          \
    struct type **le_prev;     /* ptr to ptr to this element */ \
}
```

通过分析，我们可以写出 Page 结构：

```
struct Page {
    struct {
        struct Page *le_next;
        struct Page **le_prev;
    } pp_link;
    uint16_t pp_ref;
};
```

所以该结点的结构可以用图 4-2 来表示：

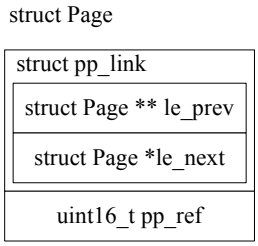

图 4-2. 页面管理双向链表结点结构

该结构存在 3 个成员：指向链表下一个结点的指针、一个指向指针的指针以及一个 short int 类型的 pp_ref，注意，short int 类型在 Bochs 模拟出来的 x86 平台上仍然是占 32 位，所以整个结构一共会占用 12 字节。

另外，系统还定义了一个链表头，该链表头的定义比较隐晦，是通过两个宏来完成的，它们分别是 memlayout.h 中的：

```
LIST_HEAD(Page_list, Page);
```

和 queue.h 中的：

```
#define    LIST_HEAD(name, type)                                \
struct name {                                                   \
        struct type *lh_first;       /* first element */        \
}
```

以及在 pmap.c 中定义的全局变量：

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

通过分析，我们发现 Page_list 结构实际上可以写成：

```
struct Page_list{
        struct Page *lh_first;
};
```

它只包含了一个指向 Page 结构的指针 lh_first。同时，page_free_list 这个全局变量实际上就是传说中的指向页面管理双向链表的头结构了，注意，它不是一个 struct Page 类型的指针或空结构！

同时，系统还定义了一些宏来对这个链表头进行操作：

在 queue.h 中：

```
#define    LIST_FIRST(head)      ((head)->lh_first)    //取得头指针

#define    LIST_INIT(head) do {                                 \
        LIST_FIRST((head)) = NULL;                              \
} while (0)             //将链表重置
```

#define **LIST_EMPTY**(head) ((head)->lh_first == NULL) //与 LIST_INIT 一样，只不过系统好象没有用过。

```
#define    LIST_HEAD_INITIALIZER(head)                          \
        { NULL }       //将链表头自身赋值为空，不过好象也没有用过
```

在 queue.h 中，还定义了很多用于操纵结点和对该双向链表进行操作的宏：

```
#define    LIST_NEXT(elm, field)       ((elm)->field.le_next)
```

//该宏返回 elm 所的下一个页面管理结点的地址。使用的时候，elm 应该为一个指向页面管理结点的指针，field=pp_link （下同）。

```
#define    LIST_INSERT_HEAD(head, elm, field) do {                    \
        if ((LIST_NEXT((elm), field) = LIST_FIRST((head))) != NULL)      \
            LIST_FIRST((head))->field.le_prev = &LIST_NEXT((elm), field);\
        LIST_FIRST((head)) = (elm);                                  \
        (elm)->field.le_prev = &LIST_FIRST((head));                  \
    } while (0)
```

//这个宏的功能，是将 elm 指向的页面管理结点成为整个页面管理双向链表的第一个元素，在实现上，是要求链表头结构（page_free_list）的 lh_first 指针指向该结构。这里需要考虑两种情况，一种是链表以前就是空的情况，在这种情况下，链表头结构的 lh_first 指针为空；另一种情况，是链表以前不为空的情况，这种情况下，链表头结构的 lh_first 指针不为空。我们通过仔细的阅读，可以发现，LIST_INSERT_HEAD 宏能够很好地处理这两种情况。对于空队列的情况，新插入的页面管理结点将会把自己的 pp_link.le_next 域赋值成链表头结构的 lh_first 的值（NULL），然后，链表头结构的 lh_first 指向新增加的结点，同时，将新增加的结点的 pp_link.le_prev 域赋值成链表头结构的 lh_first 域的地址；而对于以前的链表非空的情况，需要额外做的一步是把以前的链表首结点的 pp_link.le_prev 指针赋值为新增加的结点的 le_next 域的地址。

```
#define    LIST_INSERT_BEFORE(listelm, elm, field) do {               \
        (elm)->field.le_prev = (listelm)->field.le_prev;             \
        LIST_NEXT((elm), field) = (listelm);                         \
        *(listelm)->field.le_prev = (elm);                           \
        (listelm)->field.le_prev = &LIST_NEXT((elm), field);         \
    } while (0)

#define    LIST_INSERT_AFTER(listelm, elm, field) do {                \
        if ((LIST_NEXT((elm), field) = LIST_NEXT((listelm), field)) != NULL)\
            LIST_NEXT((listelm), field)->field.le_prev =             \
                &LIST_NEXT((elm), field);                            \
        LIST_NEXT((listelm), field) = (elm);                         \
        (elm)->field.le_prev = &LIST_NEXT((listelm), field);         \
    } while (0)
```

//这两个宏所要做的事情，是把新的结点（elm 参数），插入到结点 listelm 之前或者之后。对于 Page 结构的 pp_link.le_next 指针的使用，应该比较容易理解，需要解释的是 pp_link.le_prev 的使用。在链表的结点中，这个域所指向的前一个结点的（(elm)->pp_link.le_next）的地址！

页面管理链表主要就通过以上 3 个宏搭建起来，如图 4-3 所示：

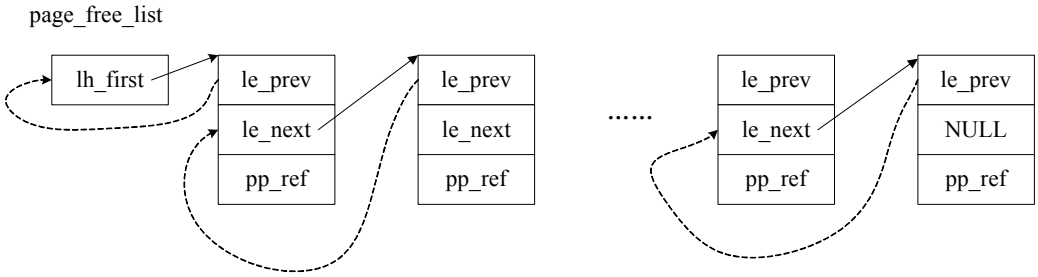

图 4-3. 页面管理链表结构

注意，图中实线和虚线表示的意义是不完全一样的。

```
#define    LIST_REMOVE(elm, field) do {                    \
        if (LIST_NEXT((elm), field) != NULL)                \
            LIST_NEXT((elm), field)->field.le_prev =        \
                (elm)->field.le_prev;                       \
        *(elm)->field.le_prev = LIST_NEXT((elm), field);    \
    } while (0)
```

//该宏所做的工作是将 elm 所指向的结点从页面管理链表中消除。这里要注意 pp_link.prev 的用法，它只是用来 index 前一个结点的 le_next 域，从而直接修改里面的内容，而不能索引整个前面的结点（比方说访问前一个结点的 pp_ref 域的内容），虽然，从链表构造的角度来说，这样设计就足够了。

还有一个宏将来在 pmap.c 中会用到：

```
#define    LIST_FOREACH(var, head, field)                   \
        for ((var) = LIST_FIRST((head));                    \
            (var);                                          \
            (var) = LIST_NEXT((var), field))
```

这个宏实际上写的是一个 for 循环头，它的作用是在该循环中遍历页面管理链表。需要注意的是，这个循环是不能用 pp_link.le_prev 域来替代的！

2) 页面管理链表在内存中的存储和放置

由于启动页式地址管理一上来就是对于物理页面的管理问题，所以，页面管理链表要在这个启动页式地址管理以前完成。我们发现在 i386_vm_init(void)函数中，是通过 boot_alloc 函数为页目录（Page directory）分配 4KB 的空间的（如以下语句），所以，我们也可以用这个函数为页面管理链表中的众多管理结点分配空间。

```
pgdir = boot_alloc(PGSIZE, PGSIZE);
```

这个空间的大小应该等于物理页面的数目×页面管理链表的结点大小（12B）。同时，为了方便管理，这个空间应该占用整数个 4KB 的页面。

至于这个空间放置在什么位置，就要看 boot_alloc 的实现了。

我们先来看下 boot_alloc()函数的实现：

```
static void* boot_alloc(uint32_t n, uint32_t align)
{
    extern char end[];
    void *v;
```

```
    if (boot_freemem == 0)
        boot_freemem = end;

    boot_freemem = ROUNDUP( boot_freemem, align );
    v = boot_freemem;
    boot_freemem += n;

    return v;
}
```

在调用这个函数的时候，传入的参数为：

n = npages* sizeof(struct Page)；align = PGSIZE (4KB);

系统中定义了一个 static char* 类型的全局变量 boot_freemem，我们看到该函数的流程是先让这个全局变量（实际上就是一个地址了）等于内核装入后的最末尾的位置（见图 4-1），然后将他望上（高地址）移到 align（这里是 4KB）对齐的位置，并从这里分配空间，分配完了以后，boot_freemem 继续望上移，并将前一个 align 对齐的位置返回回去。所以，我们的页面管理链表实际上存储在紧接着内核以及页目录的内存的“相对高端”的地方，而且该空间的大小是随着系统物理内存的大小而变化的，系统物理内存越大，这个空间占用的地方也就越多。这是因为物理内存大的话，物理页面的数目也就会比较多。这个空间的位置如图 4-4 所示。

同时，需要说明的是，系统定义了一个全局变量 pages：

struct Page* pages;　　// Virtual address of physical page array

boot_alloc()函数返回的地址就赋值到这个变量里。

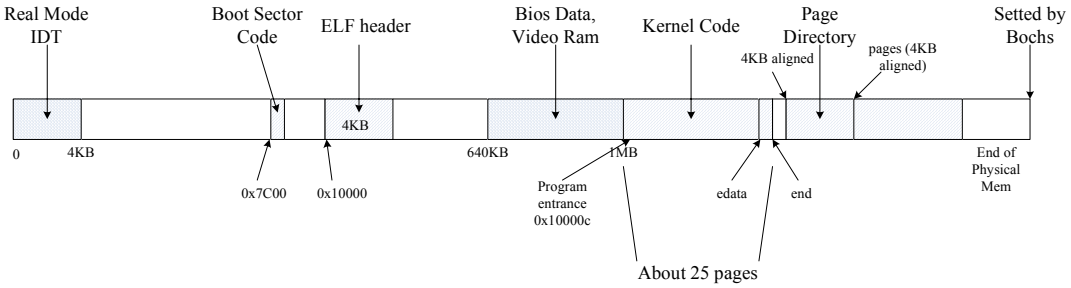

图 4-4. 页面管理空间的放置

对于 pages 的使用，就太巧妙了！因为它指向的，是有很多个 Page 结构（页面管理链表的结点）的连续内存区域。而这些结构与实际的物理页面（例如，可以根据页面的起始物理地址增加的顺序进行编号，从 1 到 n）是一一对应的，所以可以用下标的方法，唯一地表示一个物理页！例如，一个拥有 2MB 的计算机，它一共有 2MB/4KB=512 个物理页面，那么 pages[100]就引用的是物理内存 0x64000 到 0x65000 的物理页面所对应的页面管理链表结点！同时，系统还会通过 3.1.1 中提到的工具宏来将 pages 指向的这一组页面管理链表结点组织成 3.1.1 中所讨论的双向链表。这就意味着，对于页面管理和组织，至少有 2 种办法：其 1 是通过 pages[下标]，另一种是通过双向链表。对于前一种方法，好处是直观和确定性，而对于后一种方法，其好处是灵活，同时可以通过 pp_ref 域知道该页面被使用或引用的情况，虽然可能存在不确定的情况（例如分配一个空白页面，要看该链表在当时的组织，而不能确定这个空白页面到底在什么地方）。

3) 页面管理的操作

在 JOS 系统中，页面操作主要申请页面的操作 page_alloc()以及释放页面的操作 page_free()，另外，页面系统初始化的系统调用在这里也需要进行讨论。

由于系统的物理内存由 4KB 大小的很多物理页面组成，同时这些物理页面与页面管理双向链表中的管理结点有一一对应的关系，根据一个物理页的首地址可以计算出它所对应的管理结点的位置，可以用 pages[下标]来索引。在页面管理数据结构初试化的时候，系统会把所有管理结点加入页面管理双向链表，但是，由于有些物理内存已经被 x86 系统所使用或者预先放了一些系统数据（如图 4-4 所示的 BIOS、实模式 IDT 等），同时，有些内存已为内核本身所占用，这些内存当然不能把它们当自由页面分配出去！所以，在建立好页面管理双向链表后，需要将这些已经被使用的内存对应的页面管理结点从链表中剔除出去。

page_alloc()函数被调用后，将取页面管理双向链表中的第一个管理结点，并将该结点返回出去，需要注意的是，由于无法确定得到的物理内存页面是否是“干净”的，如果已经有数据，就会对我们以后的程序判断造成影响，后果很严重！所以得到页面管理结点后，一定要将它所对应的物理页面（对应的物理页面的首地址可以用 page2kva（）宏得到）清零。

2 页表管理

在本节中，主要讨论 JOS 系统在启用 x86 页式存储系统前所做的准备工作。

由于 x86 页式管理系统本身涉及很多技术上的细节问题，所以，在本节中，我们首先讨论线性地址到物理地址的转换过程，接下来讨论页目录（Page directory）、页表（Page table）和真正的数据页面的关联关系。

为了将概念加以区分，我们将地址分为三类：逻辑地址（Virtual Address）、线性地址(Linear Address)和物理地址(Physical Address)。逻辑地址是指程序在编译连接后，变量名字等的符号地址，在 JOS 系统中的内核部分，该地址是以 KERNBASE（默认等于 0xF0000000，实际上可以根据具体的情况加以修改）；线性地址是指经过 x86 保护模式的段地址变换后的地址，该变换的过程是 逻辑地址+段首地址；物理地址是指内存存储单元的编址，如 1GB 的内存，它的物理编址是从 0x00000000 到 0x40000000。需要注意的是，在本文中，这三类地址的长度都是 32 位。

在 Lab1 中，我们已经了解了 x86 的段式地址变换，也就是逻辑地址到线性地址的转换过程。如果 x86 系统未开启页式内存管理，得到的线性地址实际上也就是物理地址，但如果页式地址被开启后，得到的线性地址需要再次经过页式地址转换才能得到物理地址。

需要注意的是，在 JOS 的简单文档中提到了内核地址（Kernel Address）的概念，这个概念实际上就是指的逻辑地址。

1) 线性地址的转换

当启用了 x86 页式内存管理后，当处理器碰到一个线性地址后，它会把这个地址分成 3 部分（见图 4-5）。它们分别是页目录索引（Directory）、页表索引（Table）和页内偏移（Offset），这 3 个部分把原本 32 位的线性地址分成了 10 位、10 位和 12 位的 3 个片段。既然页内偏移地址占 12 位，页的大小就自然为 4KB 了，这个我们在前面交代过。

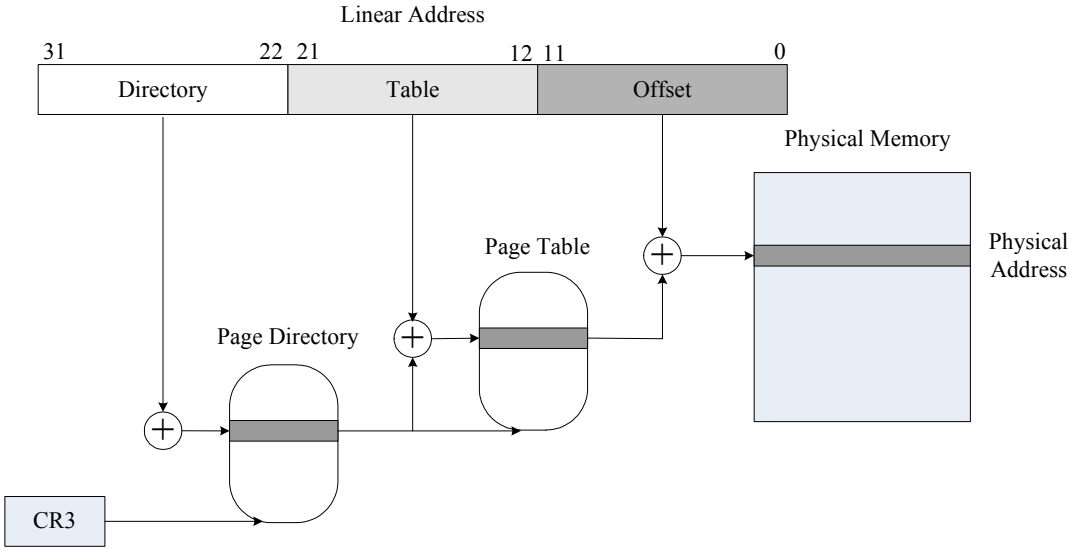

图 4-5. x86 分页模式下线性地址到物理地址的转换

页目录项在 3.1.2 中讨论页面管理时就接触过，在图 4-4 的内存结构图中就能够很直观地看到。我们知道它占用了一个 4KB 的物理页面，实际上它在内部分成了 1024 个单元（10 位有 1024 个可能的值），每个单元占 4 字节（32 位，也就是保护模式下一个 uint32_t 类型所占的内存空间大小），它们称为页目录项（Page Directory Entry），这些单元与页表中包含的单元在格式上是一致的，不同的是页表中的单元称为页表项（Page Table Entry）。图 4-6 描述了这些页目录（表）项的格式。

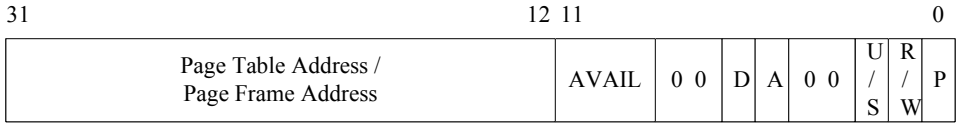

图 4-6. 页目录（表）项的格式

其中的高20位存储的是一个地址，但因为只有高20位（使用的时候低位会被全部清零），所以只能寻址 4KB 对齐的地址空间（这就是为什么我们在为页目录分配空间时要寻找内核代码后第一个 4KB 对齐的地址的原因），同时，由于 x86 把所有物理内存分成了 4KB 大小的页，每个页的首地址必然是 4KB 对齐的！所以这个表项中的高 20 位地址能够定位到内存中任何一个物理页面的首地址。如果这个表项在页目录表中，那么这个地址寻址的就是页目录的下级，也就是页表所在物理页面的首地址，而如果这个表项在页表中，这个地址寻址的就是真正的数据页（Page Frame）的首地址。

这个地址以后的都是对应物理页面的标志位，下面对它们进行解释：

P —— Present，该位用来判断对应物理页面是否存在，如果存在，该位置 1，否则为 0；

R/W —— Read/Write，该位用来判断对所指向的物理页面的访问权限，如果为 1，表示页面可写，否则，是只读页面；

U/S —— User/Supervisor，该位用来定义页面的访问者应该具备的权限。如果为 1，表示该页面是 User 权限的，大家都可以访问，如果为 0，表示只能是 Ring0 中的程序能访问；

D —— Dirty，是否被修改；

A —— Accessed，最近是否被访问；

AVAIL —— Available，可以被系统程序所使用；

0 —— 保留位，不能使用。

这些位系统在访问一个页面时就会自动地去判断，如果访问不符合规矩（如页面根本就不存在，或者权限不对的情况），系统就会产生异常，让系统去处理。

页目录所在的物理页面的首地址在启动 x86 的页式地址管理前，需要放到 CR3 中，这样 x86 在进行页式地址转换时会自动地从 CR3 中取得页目录地址，从而找到当前的页目录，如果无法找到该页目录，地址转换就无从谈起了。

通过线性地址的高 10 位，可以得到该线性地址在页目录中对应的表项，通过该页目录项中的地址（页目录表项的高 20 位）可以得到页表所在物理页的首地址，从而找到页表；然后，在通过线性地址的中间 10 位取得该地址在页表中的位置，从而得到页表项。最后，从找到的页表项所存储的地址（高 20 位）定位数据页（Page Frame），并将该数据页首地址加上线性地址中的页内偏移，从而得到物理地址。

现举例说明上述线性地址到物理地址的转换过程。

例如现在要将线性地址 0xF0119796 转换成物理地址。

首先取得该线性地址的高 10 位（页目录项偏移），结果为 0x3C0，中间 10 位（页表项偏移）为 0x46，偏移地址为 0x1796。

首先，处理器会通过 CR3 取得页目录，并取得其中的第 0x3C0（十进制的 960）项页目录项，取得该页目录项的高 20 位地址，从而得到对应的页表，再次取得页表中的第 0x46（十进制的第 70）项页表项，并进而取得该页表项的高 20 位，并将这高 20 位加上线性地址的低 12 位（为 0x1796），从而得到物理地址。

通过图 4-5 所示的地址转换机制，我们发现，它对 0~4GB 空间内的任何线性地址进行转换，这时，页目录的每个页目录项都将指向一个包含 1024 个页表项的页表（占用一个 4KB 的物理页面），整个页式转换机制将占用额外的 1024+1=1025 个 4KB 的物理页面，也就是大约 4MB。然而，由于系统运行过程中可能不会用到这么大的地址空间，所以不会有那么多页表被创建。

在现代操作系统中，我们经常可以听到“用户进程的虚地址空间为 4GB”的说法，怎么来实现呢？原理其实很简单：为每个用户进程创建一个页目录，并将这个页目录保存到用户进程的上下文中，当该用户进程被切换过来执行的时候，就将该用户进程的页目录地址写入 CR3，并重新启动一次页式内存管理。当然如果采用这种方式，势必占用更多的内存用于页式地址变换。

对于老版本的 Linux 系统以及我们现在接触的 JOS 系统，为了避免太大的内存开销（另一个原因是因为还没有虚拟内存的支持），在整个系统中只使用一个页目录。这就意味着，整个系统的线性地址空间只有 4GB，而且是所有（操作系统内核、各个用户程序）代码所公用的！这种情况下，就必须对线性地址进行合理的规划了。在 JOS 系统中，就对线性地址有所规划，具体的见后面的 JOS 的内存组织部分。

对于页式地址管理，由于页目录以及页表都存放在物理内存的页面中，要进行地址变换就势必先要到内存中访问页目录和页表。由于 CPU 和内存速度的不匹配，这样将势必极大地降低系统的效率。为了提高地址翻译的速度，提高系统的效率，x86 系统中设计了用于地址翻译的缓存来解决这一问题，这一缓存称为 TLB（Translation Look-aside buffer，即旁路转换缓冲，或称为页表缓冲），在该缓存中存放了用于最近几次地址翻译的页表项，由于程序执行的局部性原理，下一次的地址转换往往跟上一次的地址转换采用的是同一个页目录表项和页表项），同时，由于 TLB 跟处理器的距离更近，这样就极大地提高了地址翻译的效率和速度。但是，这样做可能存在一个潜在的问题：页目录（表）数据项的不一致性。以前系

统里对应一个线性地址只有唯一的存放在内存中的页目录和页表，用于完成翻译的工作，但是，现在由于 TLB 的存在，系统可能在高速缓存中也存放了一份页表项数据，用于更快地对地址进行翻译，大多数时候，它们是一致的，但是也有例外的情形。因为 TLB 中的数据对于程序员来说，是不可见的，程序对于页表项或者页目录项的修改并不能马上反映到 TLB 中，这样就可能导致错误的地址翻译，因为为了提高翻译的速度，处理器总是尽量地采用 TLB 中的页表数据进行地址的翻译。所以，为了避免这种数据的不一致性所导致的地址翻译的错误情形的出现，系统程序员就必须在对页表进行修改后，使 TLB 中旧的页表数据失效。使其失效的办法有两个，一个是重载 CR3，使整个 TLB 中的数据都失效，也可以采用 invlpg 指令。

需要指出的是，在 JOS 系统对页式内存管理进行支持进行实现的过程中，一定要把图 4-6 放在脑海中，不然很容易就迷失了方向！同时，要区分启动分页前和启动分页后的区别。在启动分页前，由于系统仍然是采用段式地址变换的，线性地址就等于物理地址，这时访问内存应该延续以前的做法，采用逻辑地址（也就是 JOS 文档中提到的内核地址，从 0xF0000000 开始）来进行对内存的读写访问（段式地址变换，也就是物理地址=逻辑地址+base，会自动地将 0xF0000000 开始的地址转换成 0x00000000 开始的实际物理地址），但是，千万不要把逻辑地址写到页目录项和页表项中，因为页式地址变换是地址变换的最后一步（x86 的做法是先将逻辑地址通过段式地址变换转换成线性地址，然后再通过页式地址变换转换成物理地址），写入页目录项和页表项的应该是实际的物理地址！也就是从 0x00000000 开始的地址。

2）页目录、页表和数据页的关联

虽然我们对 x86 的页式地址翻译机制进行了了解，回到 JOS 系统，我们发现，完成了页面管理后，系统仅仅创建了页目录（为其分配了空间），页表（Page Table）和数据页面（Page Frame）现在还没有踪影！这就要求我们继续完成 JOS 系统中对页表进行操作的所有函数，其中包括：

pgdir_walk()
page_lookup()
page_remove()
page_insert()
boot_map_segment()

这里，我们首先对 JOS 系统中定义的对页目录或页表项进行操作的宏进行解释，在具体的系统实现过程中，我们会发现，这些宏非常地有用。

```
#define PTXSHIFT 12          // offset of PTX in a linear address
#define PPN(la)              (((uintptr_t) (la)) >> PTXSHIFT)
#define VPN(la)              PPN(la)          // used to index into vpt[]
//线性地址 la 所在的物理页面在页面管理结构中的下标（物理页面号）

#define PDXSHIFT       22          // offset of PDX in a linear address
#define PDX(la)              ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)
#define VPD(la)              PDX(la)          // used to index into vpd[]
//这两个宏都是取得线性地址的页目录项地址部分

#define PTXSHIFT 12          // offset of PTX in a linear address
#define PTX(la)              ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)
```

//这个宏取得的是线性地址的页表项地址部分

#define PGOFF(la) (((uintptr_t) (la)) & 0xFFF)
//这个宏取得的是线性地址的页内偏移部分

#define PTE_ADDR(pte) ((physaddr_t) (pte) & ~0xFFF)
//取得页表项中的物理地址（指向的物理页面的首地址）

有用的 inline 函数
static inline ppn_t **page2ppn**(struct Page *pp)
返回 Page 结构 pp 所对应的页面下标；
static inline physaddr_t **page2pa**(struct Page *pp)
返回 Page 结构 pp 所对应的物理页面的物理首地址；
static inline struct Page* **pa2page**(physaddr_t pa)
返回物理地址 pa 所在的物理页面所对应的页面结构
static inline void* **page2kva**(struct Page *pp)
与 page2pa 类似，只不过返回的是 Page 结构 pp 所对应的物理页面的内核首地址（逻辑地址）

下面对这些函数进行说明。
pte_t * **pgdir_walk**(pde_t *pgdir, const void *va, int create)
检查虚拟地址（应该是线性地址）va 已经能够用页表（页目录+页表的体系）翻译，如果能够，则返回该地址对应的页表项的地址；如果不能，同时 create=0 的话，则返回空（NULL）；但是，如果 create=1 的话，为该地址创建对应的页表（因为没有实际物理页面相对应，即使创建，返回的页表项中的地址部分也为空！），并返回 va 所对应的页表项的地址。
注意：该函数返回的页表项地址为内核地址！

struct Page * **page_lookup**(pde_t *pgdir, void *va, pte_t **pte_store)
在页式地址翻译机制中查找线性地址 va 所对应的物理页面，如果找到，则返回该物理页面，并将对应的页表项的地址放到 pte_store 中；如果找不到，或其他原因，则返回空（NULL）。

void **page_remove**(pde_t *pgdir, void *va)
删除线性地址 va 所对应的物理页面。
注意：在删除页面的时候，调用的是 page_decref()，仅减低该页面的引用度，而不一定要将页面删除。同时，由于页表项发生了修改，删除操作完成后，应该调用 tlb_invalidate() 更新 TLB。

int **page_insert**(pde_t *pgdir, struct Page *pp, void *va, int perm)
这是 JOS 在实现页面支持中最重要的一个函数，该函数的功能是将页面管理结构 pp 所对应的物理页面分配给线性地址 va。同时，将对应的页表项的 permission 设置成 PTE_P&perm。
注意：一定要考虑到线性地址 va 已经指向了另外一个物理页面或者干脆就是这个函数

要指向的物理页面的情况。如果线性地址 va 已经指向了另外一个物理页面，则先要调用 page_remove 将该物理页从线性地址 va 处删除，再将 va 对应的页表项的地址赋值为 pp 对应的物理页面。如果 va 指向的本来就是参数 pp 所对应的物理页面，则将 va 对应的页表项中的物理地址赋值重新赋值为 pp 所对应的物理页面的首地址即可。

static void **boot_map_segment**(pde_t *pgdir, uintptr_t la, size_t size, physaddr_t pa, int perm)

在页表中，将线性地址[la, la+size]映射到物理地址[pa, pa+size]。

注意：size 一定是 PGSIZE(4KB)的整数倍。这个函数带来的疑问是：是否有可能有一个物理页面，它已经被分配作为存储页表的页面了，但由于在[pa, pa+size]范围内，所以又被用来成为某线性地址对应的物理页面了？！

我的感觉是：的确有这种可能！但主要是看 JOS 中的内存是如何组织的，我们在 3.2.3 中分析这个问题。

3） JOS 的内存组织

由于系统的物理内存一般是固定的（在我们的实验中，由 Bochs 配置文件的参数指定），小的内存可以只有 2MB（我常用来做实验的极限情况）。由于页式地址管理机制的采用，物理内存不再成为限制条件，所以这里主要讨论的是 JOS 操作系统中线性地址的规划。

同时，由于 JOS 系统中只用了一个页目录页面（不是像很多现代操作系统那样，每个进程，无论是内核进程还是用户进程，都有自己的页目录），所以整个系统的线性地址只有 4GB。如何规划和使用这 4GB 内存空间就成了问题。这里，我们分析 JOS 的线性地址规划，如图 4-7 所示。

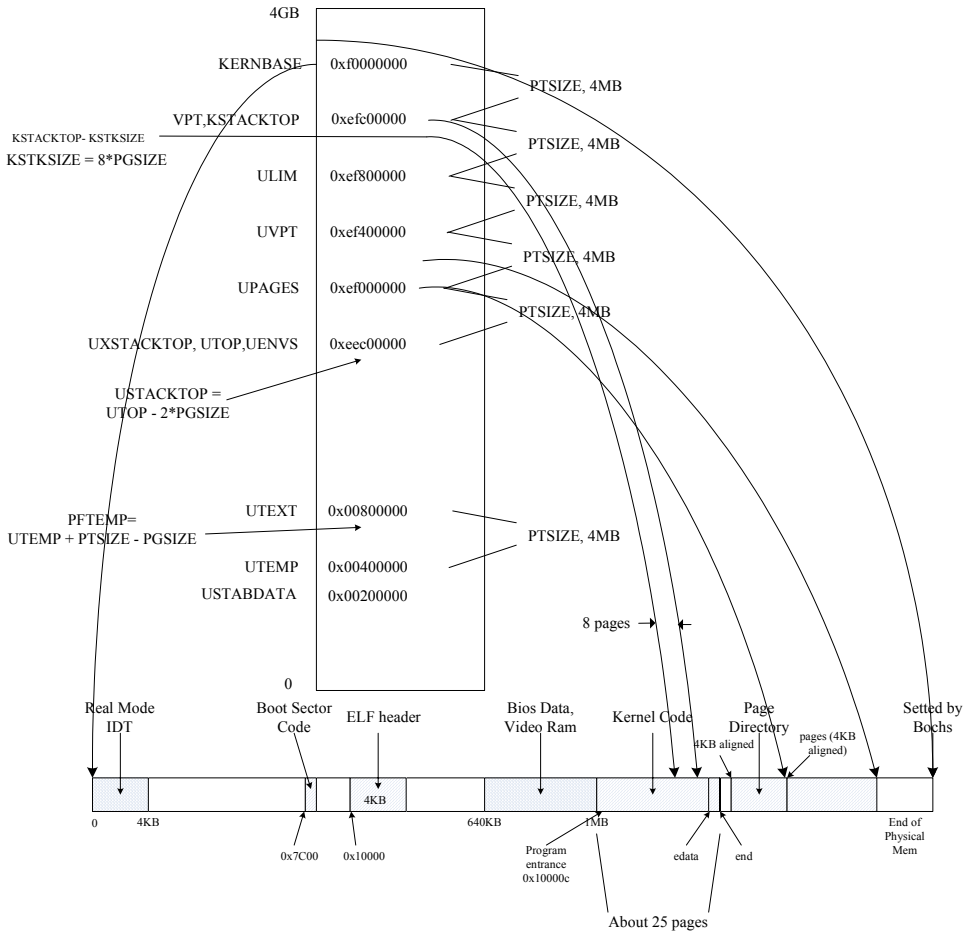

图 4-7. JOS 的线性地址规划

按照 JOS 的要求，一共有三个线性地址到物理地址的映射是需要的：

[UPAGES, sizeof(PAGES)] => [pages, sizeof(PAGES)]

这里 PAGES 代表页面管理结构所占用的空间；

[KSTACKTOP – KSTKSIZE, 8] => [bootstack, 8]

其中 bootstack 为内核编译时预先留下的 8 个页面（用做内核堆栈）；

[KERNBASE, pages in the memory] => [0, pages in the memory]

这个地址映射范围比较广，含盖了所有物理内存。

其中，最后一个地址映射最重要，因为 JOS 其后启动新的段式地址，新的段 base=0x0（见 struct Segdesc gdt[]），如果没有这个地址映射，以前的内核地址（0xf0000000 开始的地址是无法变换到实际的物理地址的）！