对32-bit ARM而言，高于760M（0x2f80,0000）的physical memory都属于highmem，kernel不能直接access，需要做kmap处理。

由struct page *page获得virtual address

如果是non-highmem

```
#define page_address(page) lowmem_page_address(page)

static __always_inline void *lowmem_page_address(const struct page *page)
{
        return __va(PFN_PHYS(page_to_pfn(page)));
}
```

#define PFN_PHYS(x)   ((phys_addr_t)(x) << PAGE_SHIFT)

如果是highmem

```
void *kmap(struct page *page)
{
        might_sleep();
        if (!PageHighMem(page))
                return page_address(page);
        return kmap_high(page);
}
EXPORT_SYMBOL(kmap);
```

核心函数kmap_high()

```c
/**
 * kmap_high - map a highmem page into memory
 * @page: &struct page to map
 *
 * Returns the page's virtual memory address.
 *
 * We cannot call this from interrupts, as it may block.
 */
void *kmap_high(struct page *page)
{
        unsigned long vaddr;

        /*
         * For highmem pages, we can't trust "virtual" until
         * after we have the lock.
         */
        lock_kmap();
        vaddr = (unsigned long)page_address(page);
        if (!vaddr)
                vaddr = map_new_virtual(page);
        pkmap_count[PKMAP_NR(vaddr)]++;
        BUG_ON(pkmap_count[PKMAP_NR(vaddr)] < 2);
        unlock_kmap();
        return (void*) vaddr;
}
```

linux/mm/highmem.c

```
1.  /**
2.   * page_address - get the mapped virtual address of a page
3.   * @page: &struct page to get the virtual address of
4.   *
5.   * Returns the page's virtual address.
6.   */
7.  void *page_address(const struct page *page)
8.  {
9.          unsigned long flags;
10.         void *ret;
11.         struct page_address_slot *pas;
12.
13.         if (!PageHighMem(page))
14.                 return lowmem_page_address(page);
15.
16.         pas = page_slot(page);
17.         ret = NULL;
18.         spin_lock_irqsave(&pas->lock, flags);
19.         if (!list_empty(&pas->lh)) {
20.                 struct page_address_map *pam;
21.
22.                 list_for_each_entry(pam, &pas->lh, list) {
23.                         if (pam->page == page) {
24.                                 ret = pam->virtual;
25.                                 goto done;
26.                         }
27.                 }
28.         }
29. done:
30.         spin_unlock_irqrestore(&pas->lock, flags);
31.         return ret;
32. }
33.
34. EXPORT_SYMBOL(page_address);
```

line 16

pas = page_slot(page);

```
/*
 * Hash table bucket
 */
static struct page_address_slot {
        struct list_head lh;                    /* List of page_address_maps */
        spinlock_t lock;                        /* Protect this bucket's list */
} ____cacheline_aligned_in_smp page_address_htable[1<<PA_HASH_ORDER];

static struct page_address_slot *page_slot(const struct page *page)
{
        return &page_address_htable[hash_ptr(page, PA_HASH_ORDER)];
}
```

#define PA_HASH_ORDER   7

page_slot()由struct page *page作为hash key在page_address_htable hash table中得到one entry。如果为NULL,即该page还未在highmem中建立过mapping,则map_new_virtual(page)（后面分析）。

如果已经建立过mapping,

```
        if (!list_empty(&pas->lh)) {
                struct page_address_map *pam;

                list_for_each_entry(pam, &pas->lh, list) {
                        if (pam->page == page) {
                                ret = pam->virtual;
                                goto done;
                        }
                }
        }
```

则在已有的mapping中查找到正确的struct page_address_map,返回该page对应的virtual.

```c
/*
 * Describes one page->virtual association
 */
struct page_address_map {
	struct page *page;
	void *virtual;
	struct list_head list;
};
```

page是位于highmem的struct page，而virtual则是该page所mapping的virtual address。凡是不同struct page，但hash key冲突的都被链接在一个list上。

```c
	pkmap_count[PKMAP_NR(vaddr)]++;
	BUG_ON(pkmap_count[PKMAP_NR(vaddr)] < 2);
```

```c
static int pkmap_count[LAST_PKMAP];
```

```c
#define LAST_PKMAP		PTRS_PER_PTE
```

```c
#define PTRS_PER_PTE		512
```

pkmap_count[512]中记录了the specific page (in highmem) 's reference count。

目前（3.18.7）kernel好像最多允许512 pages in highmem被mapping到virtual address。

如果在512 entries中找到一个empty entry,则

```c
	vaddr = PKMAP_ADDR(last_pkmap_nr);
	set_pte_at(&init_mm, vaddr,
		   &(pkmap_page_table[last_pkmap_nr]), mk_pte(page, kmap_prot));

	pkmap_count[last_pkmap_nr] = 1;
	set_page_address(page, (void *)vaddr);
```

last_pkmap_nr是0 to 512的pkmap_count[]的index。

#define PKMAP_ADDR(nr)		(PKMAP_BASE + ((nr) << PAGE_SHIFT))

#define PKMAP_BASE		(PAGE_OFFSET - PMD_SIZE)

==> 0xC000,0000 - 1UL << 21 = 3G - 2M = 0xBFE0,0000

在Documentation/arm/memory.txt中有如下说明

PAGE_OFFSET high_memory-1 Kernel direct-mapped RAM region.
This maps the platforms RAM, and typically
maps all platform RAM in a 1:1 relationship.

PKMAP_BASE PAGE_OFFSET-1 Permanent kernel mappings
One way of mapping HIGHMEM pages into kernel
space.

即from 0xBFE0,0000 to 0xC000,0000，被kernel用于mapping来自highmem的内容。

There is the following log in kernel boot log

Virtual kernel memory layout:

vector : 0xffff0000 - 0xffff1000 (4 kB)

fixmap : 0xffc00000 - 0xffe00000 (2048 kB)

vmalloc : 0xf0000000 - 0xff000000 (240 MB)

lowmem : 0xc0000000 - 0xef800000 (760 MB)

pkmap : 0xbfe00000 - 0xc0000000 (2 MB)

modules : 0xbf000000 - 0xbfe00000 (14 MB)

.text : 0xc0008000 - 0xc05cde80   (5912 kB)

.init : 0xc05ce000 - 0xc0602000   (208 kB)

.data : 0xc0602000 - 0xc0638728   (218 kB)

.bss : 0xc0638728 - 0xc06a8af4   (449 kB)

那么one page in highmem怎么确定它被mapping的virtual address呢？

```
#define PKMAP_ADDR(nr)           (PKMAP_BASE + ((nr) << PAGE_SHIFT))
```

nr为该page在pkmap_count[512]数组中的index

(nr) << PAGE_SHIFT
把该index转换到相对[page的offset

把该offset + PKMAP_BASE，即得到virtual address。

从上面的code上看到一个限制，当前active的可被利用的highmem最多为2M。

4096（1 page size）* 512 = 2M。

```
set_pte_at(&init_mm, vaddr,
        &(pkmap_page_table[last_pkmap_nr]), mk_pte(page, kmap_prot));
```

```
pte_t *pkmap_page_table;
```

```c
static void __init kmap_init(void)
{
#ifdef CONFIG_HIGHMEM
        pkmap_page_table = early_pte_alloc(pmd_off_k(PKMAP_BASE),
                PKMAP_BASE, _PAGE_KERNEL_TABLE);

        fixmap_page_table = early_pte_alloc(pmd_off_k(FIXADDR_START),
                FIXADDR_START, _PAGE_KERNEL_TABLE);
#endif
}
```

pkmap_page_table是用来mapping kmap的pte array。

mk_pte(page, kmap_prot)生成pte.由page可以获得physical address，而由physical address + 该page的access properities = pte value.

而set_pte_at()则是把pte与virtual address关联。同时kmap属于init_mm地址空间。

如果2M的kmap空间use up,那么在map_new_virtual()中

```
                   /*
                    * Sleep for somebody else to unmap their entries
                    */
                   {
                           DECLARE_WAITQUEUE(wait, current);
                           wait_queue_head_t *pkmap_map_wait =
                                   get_pkmap_wait_queue_head(color);

                           __set_current_state(TASK_UNINTERRUPTIBLE);
                           add_wait_queue(pkmap_map_wait, &wait);
                           unlock_kmap();
                           schedule();
                           remove_wait_queue(pkmap_map_wait, &wait);
                           lock_kmap();

                           /* Somebody else might have mapped it while we slept */
                           if (page_address(page))
                                   return (unsigned long)page_address(page);

                           /* Re-start */
                           goto start;
                   }
```

调用kmap()的process只能睡眠，等着其他process调用kunmap()，从而再次查询在pkmap_count[512]中是否有empty entry。

总结：

在atomic context中不能调用kmap() / kunmap(),因为该调用会sleep。

判断某个address的空间是否来自kmap(highmem)，只要判断其是否在0xBFE0,0000和

0xC000,0000之间。

```
struct page *kmap_to_page(void *vaddr)
{
        unsigned long addr = (unsigned long)vaddr;

        if (addr >= PKMAP_ADDR(0) && addr < PKMAP_ADDR(LAST_PKMAP)) {
                int i = PKMAP_NR(addr);
                return pte_page(pkmap_page_table[i]);
        }

        return virt_to_page(addr);
}
EXPORT_SYMBOL(kmap_to_page);
```