Kmap(Pkmap)

2016-03-01 (http://jake.dothome.co.kr/kmap/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

feature

- Kernel Mapping method for mapping HIGHMEM area to kmap address space among lowmem areas
  - Different architectures have different kmap address space locations and sizes
  - ARM: PAGE_OFFSET – 2M ~ 2M size up to PAGE_OFFSET
  - When accessing the HIGHMEM area from a user, of course, it is always mapped.
  - HIGHMEM regions do not operate with 32:1 direct mapping from the kernel, unlike NORMAL or DMA (DMA1) regions when accessed from the kernel. In other words, if you want to access the HIGHMEM area from the kernel, you can only access it through a separate mapping.
    - If you want to access the HIGHMEM area from the kernel, you can map it using kmap, vmap, or fixmap.
- Named Pkmap (Persistence Kernel Map)
- On x86 32-bit systems, direct access is not possible for memory above 896M, so indirect mapping can be used to access memory above that.
  - 2 Indirect Mappings
    - The first mapping is indirect mapping to the 2~4G area
    - The second mapping is the Indirect Mapping to the 64G Area (PAE)
      - In fact, Linux has a large overhead for the relevant mapping data, and the mapping data must be in a ZONE_NORMAL area that can be accessed directly, so if the mapping to 64G is required, the ZONE_NORMAL area will be too small, so Linux itself has to limit the maximum mapping to 16G.
      - Mapping data can be stored in mem_map, Bitmap, PTE, ... And so on.
      - For reference, the size of the mem_map in 16G mapping is 16G / 4K * 44byte = 176M, and in the case of small pte, it takes 16M in the worst case, although it depends on the case.
- Even on ARM 32-bit systems, access to physical memory above a certain size requires indirect mapping.
  - For areas exceeding 4G, LPAE is supported and mapped due to hardware constraints.
- If HIGHMEM is mapped for the required amount of time, the performance will be reduced due to the limited mapping area and its control. Therefore, using as few HIGHMEM areas as possible avoids overhead due to memory mapping, and it is much more advantageous to use a 64-bit architecture with a large virtual address space if you need a large number of gigabytes of memory.

- kmap() and kunmap() work in pairs, mapping a highmem page to a lowmem area for a very short time, using it again, and then releasing it again.
- When using the kmap() function, check the page and if it is already mapped to a lowmem area, just use it.
- The kmap() function can be sleeped, so you should use the kmap_atomic() function in the interrupt context.
    - The kmap_atomic() function uses the fixmap mapping area.

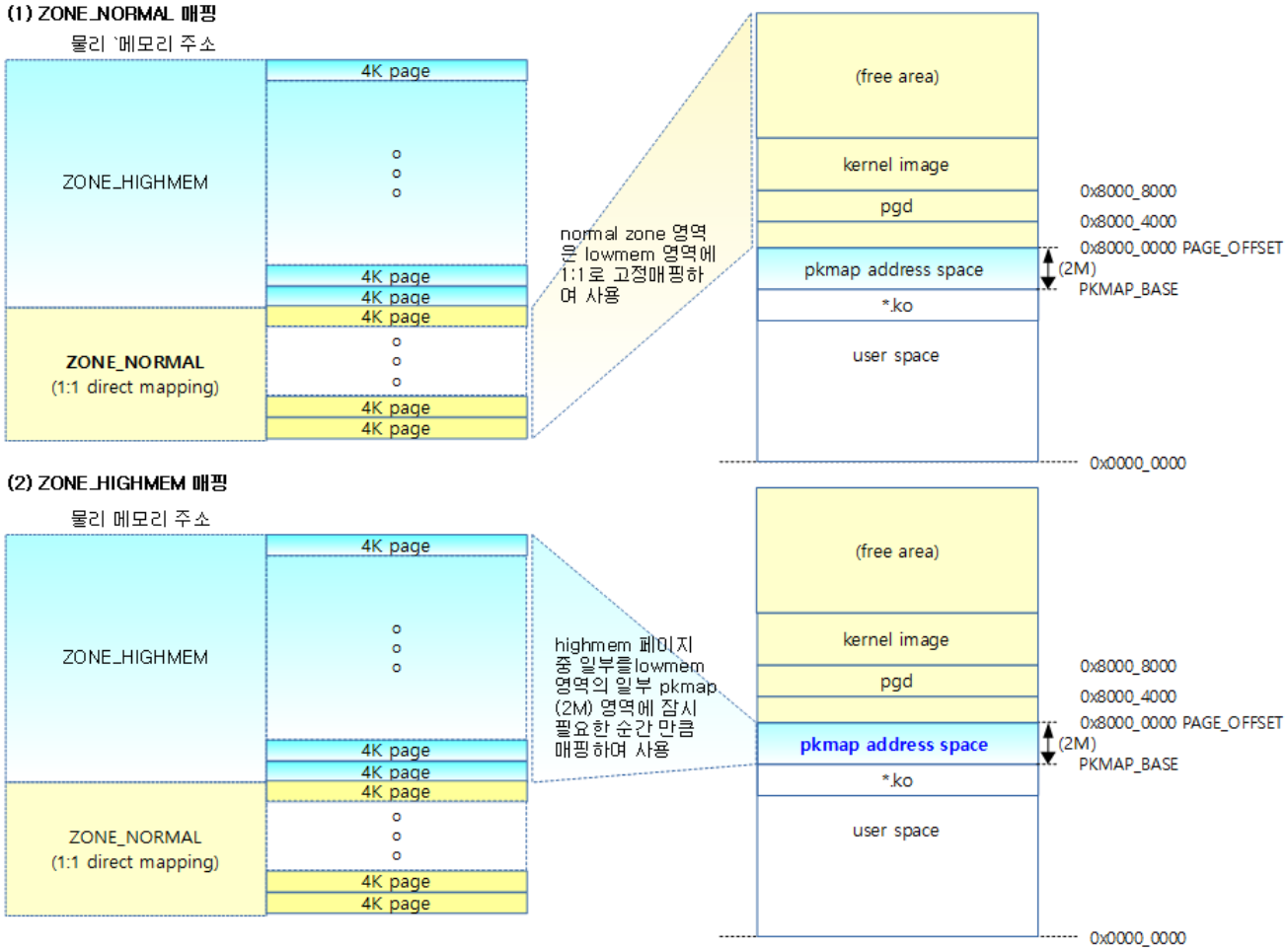
ZONE_HIGHMEM Mapping

Physical memory is mapped differently by ZONE, which is as follows.

- ZONE_DMA
    - Early devices, such as x86, were limited to a physical address area of less than 1M that devices could access while using the ISA bus. SINCE THEN, IT HAS BEEN EXTENDED TO 16M, WHICH IS THE AREA USED TO MAKE IT COMPATIBLE.
    - ARM supports different sizes for different architectures.
        - rpi2: Don't use ZONE_DMA.
- ZONE_DMA32
    - This is the area that 86-bit devices in the x64_32 can access.
- ZONE_NORMAL
    - As shown in the figure below, it is used as a 1:1 permanent mapping to the lowmem area of the kernel (starting from PAGE_OFFSET).
- ZONE_HIGHMEM
    - The area of memory that cannot be mapped 1:1 is called ZONE_HIGHMEM, and this area uses the pkmap area or fixmap area of the lowmem area to map the pages in 4K units of the HIGHMEM area for the time required for this area.
    - Since the area is limited, you need to unmap it after using it as much as you need.
        - Each architecture has a different location for the pkmap area and fixmap area, and the size of the area is also different.
            - pkmap area on ARM:
                - IT IS LOCATED JUST BELOW THE PAGE_OFFSET AND HAS A SIZE OF 2 METERS.
                - Since it uses a 2M area, it is assigned one PTE (L2) page and maps it as replacing 512 entries (pte_t).
    - RPI2 does not use ZONE_HIGHMEM by default.
        - In rpi, the PAGE_OFFSET was 0xC000_0000 (user space = 3G, kernel space = 1G), but in rpi2, the PAGE_OFFSET was set to 0x8000_0000 (user space = 2G, kernel space = 2G) to avoid using HIGHMEM.
        - Embedded applications such as rpi do not often require large virtual addresses, so PAGE_OFFSET may be used as a 1x0_8000 on systems with more than 0000G of memory. If the maximum memory was 2M, as in the case of rpi512, the PAGE_OFFSET would have been set to 0xC000_0000.
- ZONE_MOVEABLE

- This area is specifically designed for the efficiency of page fragmentation in NUMA systems and can be used as hotplug memory.

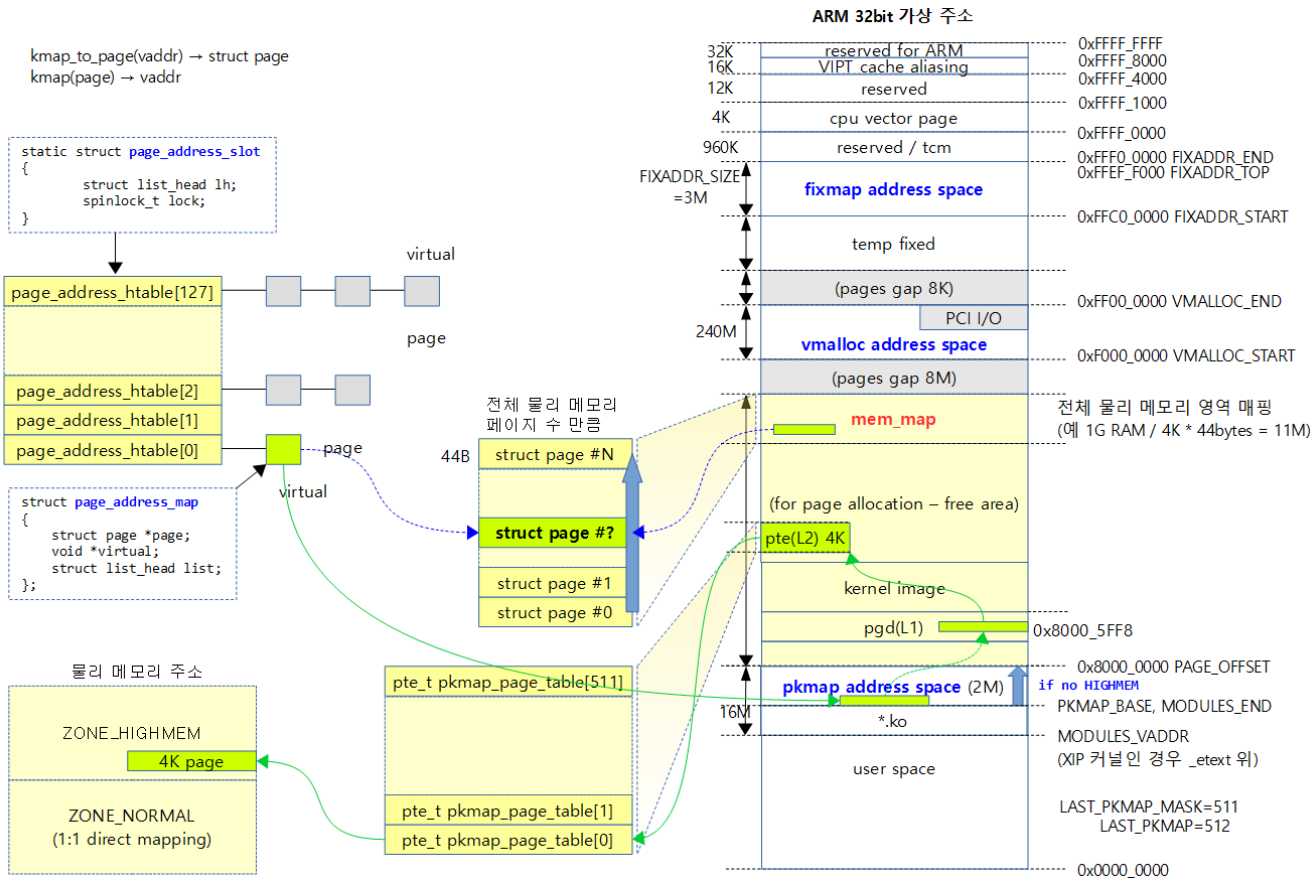
As shown in the figure below, the case ZONE_NORMAL and ZONE_HIGHMEM where the left physical memory address is mapped to the right virtual memory is shown.



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/pkmap-2.png)

The figure below shows the associated flow when mapping any one page in the HIGHMEM area.

- pkmap_page_table[] is the actual L2 page table for mapping.
- page_address_htable[] to use 128 hash methods.
  - Hash slots are managed by adding and deleting pages in a list structure.



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/kmap-1a.png)

Mapping and unmapping

kmap()

arch/arm/mm/highmem.c

```
1  void *kmap(struct page *page)
2  {
3          might_sleep();
4          if (!PageHighMem(page))
5                  return page_address(page);
6          return kmap_high(page);
7  }
8  EXPORT_SYMBOL(kmap);
```

Map the page address to a virtual address and return that virtual address. If it has already been mapped, it returns the virtual address.

- might_sleep();
  - In a kernel that requires a preemption point, it gives way to a task when necessary.
- if (! PageHighMem(page))
  - If the page address is not a HIGHMEM area, the
- return page_address(page);
  - page.
- return kmap_high(page);

- Using the page information, the physical address of the HIGHMEM area is mapped to a page in one of the kmap-mapped address areas, and the virtual address is returned.

PageHighMem()

```
#define PageHighMem(__p) is_highmem(page_zone(__p))
```

Find out whether the page is in the highmem area or not.

- is_highmeme()
  - Find out whether the zone is a highmem zone.

page_zone()

```
static inline struct zone *page_zone(const struct page *page)
{
        return &NODE_DATA(page_to_nid(page))->node_zones[page_zonenum(page)];
}
```

Returns a pointer to the zone struct that corresponds to page.

- NODE_DATA()
  - #define NODE_DATA(nid) (&contig_page_data)
    - In a UMA system, we use 1 node, which doesn't need to be managed as an array, so &contig_page_data returns a global struct variable.
- page_to_nid()
  - Determine the node id by the page number.
- page_zonenum()
  - Get the zone value from the page member variable flags.

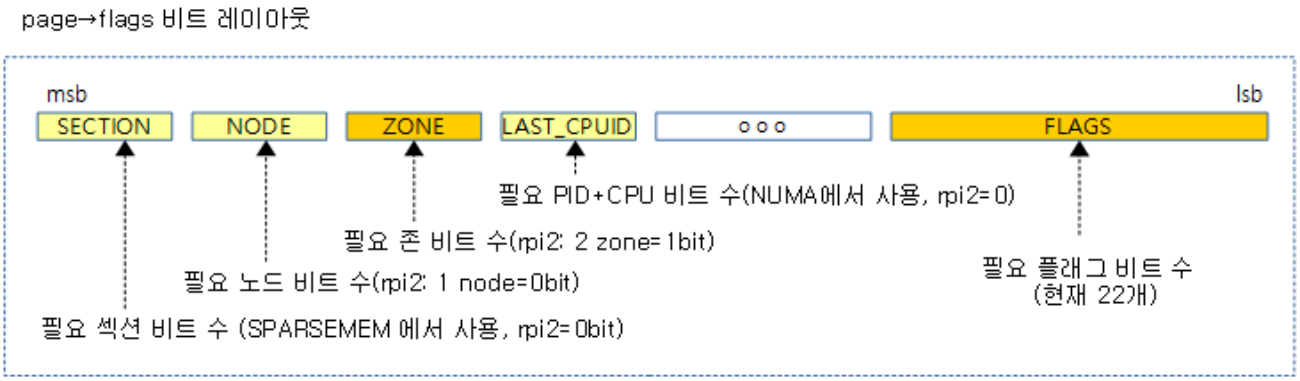
page_zonenum()

```
static inline enum zone_type page_zonenum(const struct page *page)
{
        return (page->flags >> ZONES_PGSHIFT) & ZONES_MASK;
}
```

Get the zone value from the page member variable flags.

- #define ZONES_PGSHIFT (ZONES_PGOFF * (ZONES_WIDTH != 0))
  - #define ZONES_PGOFF (NODES_PGOFF – ZONES_WIDTH)
    - #define NODES_PGOFF (SECTIONS_PGOFF – NODES_WIDTH)
      - #define SECTIONS_PGOFF ((sizeof(unsigned long)*8) – SECTIONS_WIDTH)
        - SECTIONS_WIDTH is always 0 if it is not a sparse memory
        - rpi2 doesn't use SPARSEMEM, so it doesn't need the section bits, so SECTIONS_PGOFF=32
      - NODES_WIDTH

        - If you can represent 32 pages with a 22-bit integer + sectionbits, nodebits, and zonbits, it returns CONFIG_NODES_SHIFT
        - rpi2: 0 because it is a single node
      - rpi2 uses a single node, so NODES_PGOFF=32
    - ZONES_WIDTH
      - It depends on the number of zones, 1 zone is 0, 2 zones is 1, 3~4 zones are 2, and other zones are errors.
      - rpi2: 2 because it uses 1 zones
    - ZONES_PGOFF=2 of rpi31
  - ZONES_PGSHIFT=2 of RPI31
- Extract only the zone bit from the page->flags value by shifting it to the right and masking it.



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/page-1.png)

is_highmem()

```
01 /**
02  * is_highmem - helper function to quickly check if a struct zone is a
03  *              highmem zone or not.  This is an attempt to keep referen
   ces
04  *              to ZONE_{DMA/NORMAL/HIGHMEM/etc} in general code to a mi
   nimum.
05  * @zone - pointer to struct zone variable
06  */
07 static inline int is_highmem(struct zone *zone)
08 {
09 #ifdef CONFIG_HIGHMEM
10         int zone_off = (char *)zone - (char *)zone->zone_pgdat->node_zon
   es;
11         return zone_off == ZONE_HIGHMEM * sizeof(*zone) ||
12                (zone_off == ZONE_MOVABLE * sizeof(*zone) &&
13                 zone_movable_is_highmem());
14 #else
15         return 0;
16 #endif
17 }
```

Find out whether the zone is a highmem zone.

- zone_off
  - The address of the currently requested zone structure minus the address of the first zone structure.
- zone_off size determines if it is in the HIGHMEM area and returns.

page_address()

mm/highmem.c

```
01 /**
02  * page_address - get the mapped virtual address of a page
03  * @page: &struct page to get the virtual address of
04  *
05  * Returns the page's virtual address.
06  */
07 void *page_address(const struct page *page)
08 {
09         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);
```

Retrieves an existing hash table with the address of the page structure specified as an argument and finds and returns a virtual address value that has already been mapped.

- return lowmem_page_address(page);
  - page, as an argument, returns the virtual address of the lowmem area.
- step = page_slot(page);
  - Retrieves the struct pointer page_address_slot the hash(slot) in the hash table.
- spin_lock_irqsave(&pas->lock, flags);
  - Perform a spin lock to retrieve the list entry.
- if (!list_empty(&pas->lh)) {
  - If the list is not empty, the
- list_for_each_entry(pump, &pas->lh, list) {
  - It looks up all the list entries and assigns them one by one to a PAM (page_address_map structure pointer).
  - if (pam->page == page) {
    - If the page in the entry and the page requested by the argument are the same,
  - ret = pam->virtual;
    - Returns the virtual value of the entry.

- spin_unlock_irqrestore(&pas->lock, flags);
  - Since all the list entries have been searched, a spin unlock is performed.

lowmem_page_address()

include/linux/mm.h

```
1 static __always_inline void *lowmem_page_address(const struct page *pag
  e)
2 {
3         return __va(PFN_PHYS(page_to_pfn(page)));
4 }
```

Find the virtual address value with the page address in the lowmem area.

- page_to_pfn()
  - Page address to get the pfn value.
  - Implementation routines use one of the following four types, depending on the memory model:
    - CONFIG_FLATMEM, CONFIG_DISCONTIGMEM, CONFIG_SPARSEMEM_VMEMMAP, CONFIG_SPARSEMEM
    - This is a macro action that uses CONFIG_FLATMEM.

```
1 #define __page_to_pfn(page)     ((unsigned long)((page) - mem_map) + ARC
  H_PFN_OFFSET
```

- PFN_PHYS()
  - Get the physical address by the pfn value.

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

- __va()
  - Converts the physical address of the lowmem area to a virtual address value.

```
1 #define __va(x)                 ((void *)__phys_to_virt((phys_addr_t)
  (x)))
```

kmap_high()

mm/highmem.c

```
01 /**
02  * kmap_high - map a highmem page into memory
03  * @page: &struct page to map
04  *
05  * Returns the page's virtual memory address.
06  *
07  * We cannot call this from interrupts, as it may block.
08  */
09 void *kmap_high(struct page *page)
10 {
11         unsigned long vaddr;
12
13         /*
```

```
         * 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;
}

EXPORT_SYMBOL(kmap_high);
```

Using the page structure information, it maps the page frame (4K) to a virtual address and returns that address.

- lock_kmap()
  - If you use the kmap() and kunmap() functions at the same time using spin locks for global kmap_lock, sirialization is done in order.
- vaddr = (unsigned long)page_address(page);
  - Retrieves the virtual address that has already been mapped to the page address.
- if (!vaddr)
  - If it's a new page with no mapped address
- vaddr = map_new_virtual(page);
  - Assign a new virtual address.
- pkmap_count[PKMAP_NR(vaddr)]++
  - Increments the pkmap counter for the pkmap entry for that address.
- unlock_kmap();
  - Perform a spin unlock on a global kmap_lock.

map_new_virtual()

mm/highmem.c

```
static inline unsigned long map_new_virtual(struct page *page)
{
        unsigned long vaddr;
        int count;
        unsigned int last_pkmap_nr;
        unsigned int color = get_pkmap_color(page);

start:
        count = get_pkmap_entries_count(color);
        /* Find an empty entry */
        for (;;) {
                last_pkmap_nr = get_next_pkmap_nr(color);
                if (no_more_pkmaps(last_pkmap_nr, color)) {
                        flush_all_zero_pkmaps();
                        count = get_pkmap_entries_count(color);
                }
                if (!pkmap_count[last_pkmap_nr])
                        break;  /* Found a usable entry */
                if (--count)
                        continue;

                /*
```

```
23                  * Sleep for somebody else to unmap their entries
24                  */
25                 {
26                         DECLARE_WAITQUEUE(wait, current);
27                         wait_queue_head_t *pkmap_map_wait =
28                                 get_pkmap_wait_queue_head(color);
29
30                         __set_current_state(TASK_UNINTERRUPTIBLE);
31                         add_wait_queue(pkmap_map_wait, &wait);
32                         unlock_kmap();
33                         schedule();
34                         remove_wait_queue(pkmap_map_wait, &wait);
35                         lock_kmap();
36
37                         /* Somebody else might have mapped it while we s
   lept */
38                         if (page_address(page))
39                                 return (unsigned long)page_address(pag
   e);
40
41                         /* Re-start */
42                         goto start;
43                 }
44         }
45         vaddr = PKMAP_ADDR(last_pkmap_nr);
46         set_pte_at(&init_mm, vaddr,
47                    &(pkmap_page_table[last_pkmap_nr]), mk_pte(page, kmap
   _prot));
48
49         pkmap_count[last_pkmap_nr] = 1;
50         set_page_address(page, (void *)vaddr);
51
52         return vaddr;
53 }
```

- unsigned int color = get_pkmap_color(page);
  - It is a color value that is returned when data cache aliasing is required in memory in the highmem area, but it is only used by certain architectures (mips & xtensa) and is not yet used by ARM, so it returns 0.
  - 참고: mm/highmem: make kmap cache coloring aware (https://github.com/torvalds/linux/commit/15de36a4c3cf33aa4e194bfbff002048aa4a21c3)
- count = get_pkmap_entries_count(color);
  - ARM returns the maximum number of mapping entries for highmem.
    - LAST_PKMAP(512)
- last_pkmap_nr = get_next_pkmap_nr(color);
  - Returns the last entry number plus 1.
    - Repeat 0~511.
- if (no_more_pkmaps(last_pkmap_nr, color)) {
  - What if last_pkmap_nr = 0?
- flush_all_zero_pkmaps();
  - Unuse unused pkmap entries.
- count = get_pkmap_entries_count(color);
  - ARM always returns LAST_PKMAP 512.
- if (!pkmap_count[last_pkmap_nr])
  - If an empty entry is found (the entry counter is 0)
- DECLARE_WAITQUEUE(wait, current);

    - Create a wait entry with the current task.
- wait_queue_head_t *pkmap_map_wait = get_pkmap_wait_queue_head(color);
    - I know the wait_queue.
- __set_current_state(TASK_UNINTERRUPTIBLE);
    - Replace the current task with uninterruptible.
- add_wait_queue(pkmap_map_wait, &wait);
    - pkmap_map_wait Add a wait entry to the queue.
- unlock_kmap();
    - Spin-unlock to sleep().
- schedule();
    - Schedule a lease.
- remove_wait_queue(pkmap_map_wait, &wait);
    - pkmap_map_wait Removes the current wait entry from the queue.
- lock_kmap();
    - Spin lock again.
- if (page_address(page))
    - If someone (another task) has done the mapping, it will exit.
- goto start;
    - Try the beginning again.
- vaddr = PKMAP_ADDR(last_pkmap_nr);
    - Obtain the virtual address by the last entry number used.

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

- set_pte_at(&init_mm, vaddr, &(pkmap_page_table[last_pkmap_nr]), mk_pte(page, kmap_prot));
    - Set the PTE entry.
- pkmap_count[last_pkmap_nr] = 1;
    - Make use counter 1.
- set_page_address(page, (void *)vaddr);
    - Substitute the virtual address of the page.

flush_all_zero_pkmaps()

mm/highmem.c

```
01 static void flush_all_zero_pkmaps(void)
02 {
03         int i;
04         int need_flush = 0;
05
06         flush_cache_kmaps();
07
08         for (i = 0; i < LAST_PKMAP; i++) {
09                 struct page *page;
10
11                 /*
12                  * zero means we don't have anything to do,
13                  * >1 means that it is still in use. Only
14                  * a count of 1 means that it is free but
```

```
                 * needs to be unmapped
                 */
                if (pkmap_count[i] != 1)
                        continue;
                pkmap_count[i] = 0;

                /* sanity check */
                BUG_ON(pte_none(pkmap_page_table[i]));

                /*
                 * Don't need an atomic fetch-and-clear op here;
                 * no-one has the page mapped, and cannot get at
                 * its virtual address (and hence PTE) without first
                 * getting the kmap_lock (which is held here).
                 * So no dangers, even with speculative execution.
                 */
                page = pte_page(pkmap_page_table[i]);
                pte_clear(&init_mm, PKMAP_ADDR(i), &pkmap_page_table
[i]);

                set_page_address(page, NULL);
                need_flush = 1;
        }
        if (need_flush)
                flush_tlb_kernel_range(PKMAP_ADDR(0), PKMAP_ADDR(LAST_PK
MAP));
}
```

- flush_cache_maps()

```
#define flush_cache_kmaps() \
        do { \
                if (cache_is_vivt()) \
                        flush_cache_all(); \
        } while (0)
```

- for (i = 0; in < LAST_PKMAP; i++) {
  - Loops as many as the entire PKMAP entry (512)
- if (pkmap_count[i] != 1)
  - If the usage counter exceeds 1, it means that it is already in use.
  - 1 is the case when it is free.
- pkmap_count[i] = 0;
  - First, set the counter to 0.
- page = pte_page(pkmap_page_table[i])
  - Page is determined by the value of the PTE entry matched to the corresponding loop counter.
- pte_clear(&init_mm, PKMAP_ADDR(i), &pkmap_page_table[i]);
  - PTE Entry Clear
- set_page_address(page, NULL);
  - Set null to the virtual value of an item mapped to a PKMAP entry
- need_flush = 1;
  - If the PTE entry changes even once, set it to 1.
- flush_tlb_kernel_range(PKMAP_ADDR(0), PKMAP_ADDR(LAST_PKMAP));
  - Perform a tlb flush for the PKMAP 2M area.

set_page_address()

mm/highmem.c

```
01 /**
02  * set_page_address - set a page's virtual address
03  * @page: &struct page to set
04  * @virtual: virtual address to use
05  */
06 void set_page_address(struct page *page, void *virtual)
07 {
08         unsigned long flags;
09         struct page_address_slot *pas;
10         struct page_address_map *pam;
11
12         BUG_ON(!PageHighMem(page));
13
14         pas = page_slot(page);
15         if (virtual) {          /* Add */
16                 pam = &page_address_maps[PKMAP_NR((unsigned long)virtua
   l)];
17                 pam->page = page;
18                 pam->virtual = virtual;
19
20                 spin_lock_irqsave(&pas->lock, flags);
21                 list_add_tail(&pam->list, &pas->lh);
22                 spin_unlock_irqrestore(&pas->lock, flags);
23         } else {                /* Remove */
24                 spin_lock_irqsave(&pas->lock, flags);
25                 list_for_each_entry(pam, &pas->lh, list) {
26                         if (pam->page == page) {
27                                 list_del(&pam->list);
28                                 spin_unlock_irqrestore(&pas->lock, flag
   s);
29                                 goto done;
30                         }
31                 }
32                 spin_unlock_irqrestore(&pas->lock, flags);
33         }
34 done:
35         return;
36 }
```

Search for page in the PKMAP mapping hash slot and set the page->virtual value if found.

- step = page_slot(page);
  - Get the hash slot information from the page address.
- if (virtual) { /* Add */
  - When the Add Mapping command is requested
- pam = &page_address_maps[PKMAP_NR((unsigned long)virtual)];
  - PKMAP mapping pam is identified by entry number.
- pam->page = page;
  - page struct address.
- pam->virtual = virtual;
  - Record the virtual address.
- list_add_tail(&pump->list, &pas->lh);
  - Add pam->list to pas->lh.
  - In other words, it registers a page mapping struct in the list of the corresponding hash slot management array.
- } else { /* Remove */
  - When a command to delete a mapping is requested.

- list_for_each_entry(pump, &pas->lh, list) {
  - Retrieves the list of all the hash slot management arrays.
- if (pam->page == page) {
  - If the page is found, the
- list_del(&pam->list);
  - Delete the mapping.

kunmap()

Unmap the highmem page address mapped to the kmap area. If it is not a highmem page address, it will simply return.

arch/arm/mm/highmem.c

```
1 void kunmap(struct page *page)
2 {
3         BUG_ON(in_interrupt());
4         if (!PageHighMem(page))
5                 return;
6         kunmap_high(page);
7 }
8 EXPORT_SYMBOL(kunmap);
```

- if (! PageHighMem(page))
  - If the page address is not in the HIGHMEM area, it simply returns.
- kunmap_high(page);
  - Remove the page from the kmap mapping area.

kunmap_high()

Unmap the highmem page address mapped to the kmap area.

mm/highmem.c

```
01 /**
02  * kunmap_high - unmap a highmem page into memory
03  * @page: &struct page to unmap
04  *
05  * If ARCH_NEEDS_KMAP_HIGH_GET is not defined then this may be called
06  * only from user context.
07  */
08 void kunmap_high(struct page *page)
09 {
10         unsigned long vaddr;
11         unsigned long nr;
12         unsigned long flags;
13         int need_wakeup;
14         unsigned int color = get_pkmap_color(page);
15         wait_queue_head_t *pkmap_map_wait;
16
17         lock_kmap_any(flags);
18         vaddr = (unsigned long)page_address(page);
19         BUG_ON(!vaddr);
20         nr = PKMAP_NR(vaddr);
21
22         /*
23          * A count must never go down to zero
```

```
24              * without a TLB flush!
25              */
26             need_wakeup = 0;
27             switch (--pkmap_count[nr]) {
28             case 0:
29                     BUG();
30             case 1:
31                     /*
32                      * Avoid an unnecessary wake_up() function call.
33                      * The common case is pkmap_count[] == 1, but
34                      * no waiters.
35                      * The tasks queued in the wait-queue are guarded
36                      * by both the lock in the wait-queue-head and by
37                      * the kmap_lock.  As the kmap_lock is held here,
38                      * no need for the wait-queue-head's lock.  Simply
39                      * test if the queue is empty.
40                      */
41                     pkmap_map_wait = get_pkmap_wait_queue_head(color);
42                     need_wakeup = waitqueue_active(pkmap_map_wait);
43             }
44             unlock_kmap_any(flags);
45
46             /* do wake-up, if needed, race-free outside of the spin lock */
47             if (need_wakeup)
48                     wake_up(pkmap_map_wait);
49     }
50
51     EXPORT_SYMBOL(kunmap_high);
```

- unsigned int color = get_pkmap_color(page);
  - ARM doesn't yet use d-cache aliasing in the highmem area, so it returns 0.
- lock_kmap_any(flags);
  - Use the kmap global spin lock to sirialization using the kmap() and kunmap() functions at the same time.
- vaddr = (unsigned long)page_address(page);
  - Retrieves the virtual address that has already been mapped to the page address.
- nr = PKMAP_NR(vaddr);
  - Find the slot number (0~127) corresponding to the virtual address.
- switch (–pkmap_count[nr]) {
  - Reduces use counters.
- case 1:
  - If the reduced value is 1
- pkmap_map_wait = get_pkmap_wait_queue_head(color);
  - Prepare wait_queue.
- need_wakeup = waitqueue_active(pkmap_map_wait);
  - If the wait_queue has a task, it is true
- unlock_kmap_any(flags);
  - Perform a kmap global spin unlock.
- wake_up(pkmap_map_wait);
  - Wake up a task that exists in the standby queue.

get_pkmap_wait_queue_head()

mm/highmem.c

```
/*
 * Get head of a wait queue for PKMAP entries of the given color.
 * Wait queues for different mapping colors should be independent to avoid
 * unnecessary wakeups caused by freeing of slots of other colors.
 */
static inline wait_queue_head_t *get_pkmap_wait_queue_head(unsigned int color)
{
        static DECLARE_WAIT_QUEUE_HEAD(pkmap_map_wait);

        return &pkmap_map_wait;
}
```

- The ARM architecture doesn't yet use a color value for d-cache for the HIGHMEM region, so it doesn't use a color value as an argument.
- wait_queue static inline and returns.

waitqueue_active()

include/linux/wait.h

```
static inline int waitqueue_active(wait_queue_head_t *q)
{
        return !list_empty(&q->task_list);
}
```

- Returns true if the wait_queue has a task, false if it doesn't.

consultation

- page_address_init() (http://jake.dothome.co.kr/page_address_init) | Qc
- Fixmap (http://jake.dothome.co.kr/fixmap) | Qc
- kmap(): Mapping Arbitrary Pages Into Kernel VM (http://osinside.net/linuxMM/kmap_k.html)
- High Memory Management (https://www.kernel.org/doc/gorman/html/understand/understand012.html)
- What does the Virtual kernel Memory Layout in dmesg imply? (http://unix.stackexchange.com/questions/5124/what-does-the-virtual-kernel-memory-layout-in-dmesg-imply)
- Understanding The Linux Virtual Memory Manager – Download (http://www.csn.ul.ie/~mel/docs/vm/guide/pdf/understand.pdf)
- HIGH MEMORY HANDLING (https://www.kernel.org/doc/Documentation/vm/highmem.txt) | kernel.org
- High Memory (http://egloos.zum.com/hermes2/v/839611) | Merculy

LEAVE A COMMENT

Your email will not be published. Required fields are marked with *

Comments

name *

email *

Website

WRITE A COMMENT

‹ Bootmem with bitmap (http://jake.dothome.co.kr/bootmem/)

Fixmap › (http://jake.dothome.co.kr/fixmap/)

Munc Blog (2015 ~ 2024)