Ioremap

2016-12-13 (http://jake.dothome.co.kr/ioremap/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

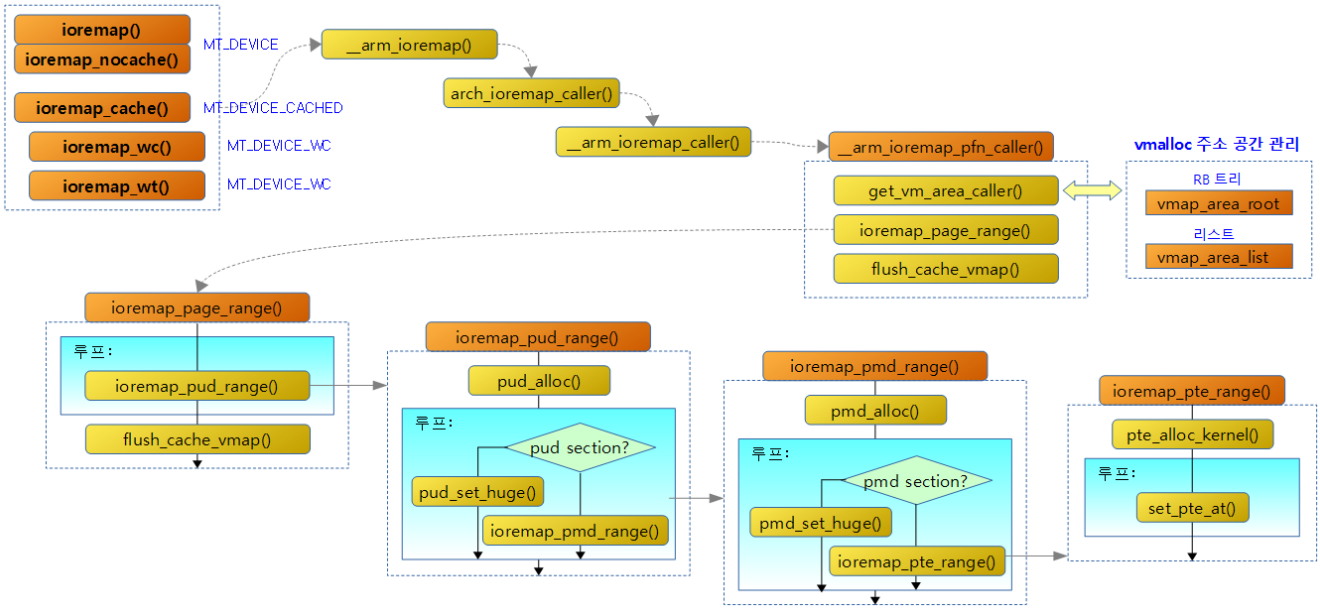
IO Mapping

The regular ioremap function uses five functions to find and map an empty vmalloc area, depending on the type of mapping. Each architecture has different types of caches that are supported, so if a particular mode is not supported, it should be downcompatible. The ioremap_cache() function uses a cache, but depending on the architecture, it can use a write cache in addition to a read cache. In order to make the options for the write cache as selectable as possible, certain architectures may use separate ioremap_wc() and ioremap_wt() functions. Note that as shown in the following table, arm and arm5 have slightly different mapping types than the original function intended.

함수명	읽기캐시	쓰기캐시	**arm** 매핑 타입	**arm64** 매핑 타입
ioremap()	X	X	MT_DEVICE	PROT_DEVICE_nGnRE
ioremap_nocache()	X	X	MT_DEVICE	PROT_DEVICE_nGnRE
ioremap_wc()	O	O	MT_DEVICE_WC	PROT_NORMAL_NC
ioremap_wt()	O	X	MT_DEVICE_WC	PROT_DEVICE_nGnRE
ioremap_cache()	O	Any (O or X)	MT_DEVICE_CACHED	PROT_NORMAL

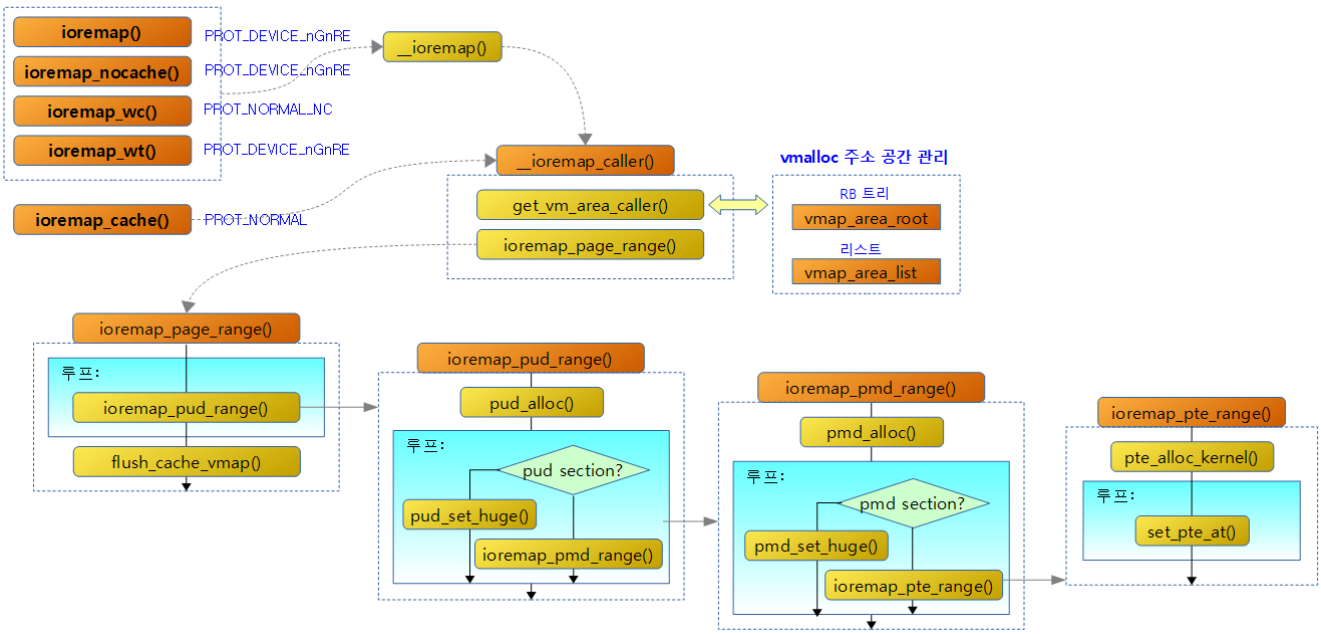
(http://jake.dothome.co.kr/wp-content/uploads/2016/12/ioremap-3.png)

32bit ARM IOREMAP Function Flowchart



(http://jake.dothome.co.kr/wp-content/uploads/2016/12/ioremap-1.png)

arm64 ioremap function flowchart



(http://jake.dothome.co.kr/wp-content/uploads/2016/12/ioremap-2.png)

ioremap(), etc.

arch/arm/include/asm/io.h

```
01 /*
02  * ioremap and friends.
03  *
04  * ioremap takes a PCI memory address, as specified in
05  * Documentation/io-mapping.txt.
06  *
07  */
08 #define ioremap(cookie,size)            __arm_ioremap((cookie), (size),
   MT_DEVICE)
09 #define ioremap_nocache(cookie,size)    __arm_ioremap((cookie), (size),
   MT_DEVICE)
```

```
10 #define ioremap_cache(cookie,size)      __arm_ioremap((cookie), (size),
   MT_DEVICE_CACHED)
11 #define ioremap_wc(cookie,size)         __arm_ioremap((cookie), (size),
   MT_DEVICE_WC)
```

On arm64, the ioremap(), ioremap_nocache(), and ioremap_wt() functions use PROT_DEVICE_nGnRE mappings to map to device types that don't use cache, and the ioremap_wc() function doesn't use caches, but the difference is that they map to memory types rather than device types.

__arm_ioremap()

arch/arm/mm/ioremap.c

```
1 void __iomem *
2 __arm_ioremap(phys_addr_t phys_addr, size_t size, unsigned int mtype)
3 {
4         return arch_ioremap_caller(phys_addr, size, mtype,
5                 __builtin_return_address(0));
6 }
7 EXPORT_SYMBOL(__arm_ioremap);
```

Find an empty vmalloc space equal to the request physical address and size, map it to the mtype attribute, and return a virtual address.

Global __arm_ioremap_caller Function Pointer Variable

arch/arm/mm/ioremap.c

```
1 void __iomem * (*arch_ioremap_caller)(phys_addr_t, size_t,
2                                       unsigned int, void *) =
3         __arm_ioremap_caller;
```

At compile time, assign the address of the __arm_ioremap_caller() function to the global arch_ioremap_caller function pointer variable.

__arm_ioremap_caller()

arch/arm/mm/ioremap.c

```
01 void __iomem *__arm_ioremap_caller(phys_addr_t phys_addr, size_t size,
02         unsigned int mtype, void *caller)
03 {
04         phys_addr_t last_addr;
05         unsigned long offset = phys_addr & ~PAGE_MASK;
06         unsigned long pfn = __phys_to_pfn(phys_addr);
07
08         /*
09          * Don't allow wraparound or zero size
10          */
11         last_addr = phys_addr + size - 1;
12         if (!size || last_addr < phys_addr)
13                 return NULL;
14
15         return __arm_ioremap_pfn_caller(pfn, offset, size, mtype,
16                         caller);
17 }
```

Find an empty vmalloc space equal to the request physical address and size, map it to the mtype attribute, and return a virtual address.

- In line 5~6 of the code, cut the physical address page by page to get the remaining offset value and pfn value.
- On line 11 of the code, find the end address of the page that you want to map page by.
  - e.g. phys_addr=0x1234_5000, size=0x2000
    - last_addr=0x1234_6fff
- Returns null on lines 12~13 if size is 0 or if the end address exceeds the system range.
- In line 15 of the code, find the empty vmalloc space by size for the pfn, map it to the mtype attribute, and return the virtual address.

__arm_ioremap_pfn_caller()

arch/arm/mm/ioremap.c

```
01 void __iomem * __arm_ioremap_pfn_caller(unsigned long pfn,
02         unsigned long offset, size_t size, unsigned int mtype, void *cal
   ler)
03 {
04         const struct mem_type *type;
05         int err;
06         unsigned long addr;
07         struct vm_struct *area;
08         phys_addr_t paddr = __pfn_to_phys(pfn);
09
10 #ifndef CONFIG_ARM_LPAE
11         /*
12          * High mappings must be supersection aligned
13          */
14         if (pfn >= 0x100000 && (paddr & ~SUPERSECTION_MASK))
15                 return NULL;
16 #endif
17
18         type = get_mem_type(mtype);
19         if (!type)
20                 return NULL;
21
22         /*
23          * Page align the mapping size, taking account of any offset.
24          */
25         size = PAGE_ALIGN(offset + size);
26
27         /*
28          * Try to reuse one of the static mapping whenever possible.
29          */
30         if (size && !(sizeof(phys_addr_t) == 4 && pfn >= 0x100000)) {
31                 struct static_vm *svm;
32
33                 svm = find_static_vm_paddr(paddr, size, mtype);
34                 if (svm) {
35                         addr = (unsigned long)svm->vm.addr;
36                         addr += paddr - svm->vm.phys_addr;
37                         return (void __iomem *) (offset + addr);
38                 }
39         }
40
41         /*
42          * Don't allow RAM to be mapped - this causes problems with ARMv
   6+
43          */
```

```
44          if (WARN_ON(pfn_valid(pfn)))
45                  return NULL;
46
47          area = get_vm_area_caller(size, VM_IOREMAP, caller);
48          if (!area)
49                  return NULL;
50          addr = (unsigned long)area->addr;
51          area->phys_addr = paddr;
52
53  #if !defined(CONFIG_SMP) && !defined(CONFIG_ARM_LPAE)
54          if (DOMAIN_IO == 0 &&
55              (((cpu_architecture() >= CPU_ARCH_ARMv6) && (get_cr() & CR_X
    P)) ||
56                 cpu_is_xsc3()) && pfn >= 0x100000 &&
57                 !((paddr | size | addr) & ~SUPERSECTION_MASK)) {
58                  area->flags |= VM_ARM_SECTION_MAPPING;
59                  err = remap_area_supersections(addr, pfn, size, type);
60          } else if (!((paddr | size | addr) & ~PMD_MASK)) {
61                  area->flags |= VM_ARM_SECTION_MAPPING;
62                  err = remap_area_sections(addr, pfn, size, type);
63          } else
64  #endif
65                  err = ioremap_page_range(addr, addr + size, paddr,
66                                           __pgprot(type->prot_pte));
67
68          if (err) {
69                  vunmap((void *)addr);
70                  return NULL;
71          }
72
73          flush_cache_vmap(addr, addr + size);
74          return (void __iomem *) (offset + addr);
75  }
```

Find an empty vmalloc space equal to the request physical address and size, map it to the mtype attribute, and return a virtual address.

- If it knows the mem_type struct corresponding to the mtype requested in line 18~20 of the code, it returns null.
- In line 25 of the code, to get the page size required for the mapping, we sort the size plus offset by page.
    - 예) offset=0x678, size=0x1000, PAGE_SIZE=4K
        - size=0x2000 (2 pages)
- Unless the physical address is 30 on line 4 and the pfn is greater than 0x10000 (4G)
- In line 33~38 of the code, if the vmalloc space is already contained within the static mapping area with the same type, it will find and return only the virtual address without mapping.
- In lines 47~49 of code, we use the VM_IOREMAP flag to find an empty area in the vmalloc space, get the zone information in the form of a vm_struct struct, and if it is null, exit the function.
- Assign the starting virtual address found in lines 50~51 to addr and paddr to the start physical address of the zone.
- If you don't use the CONFIG_SMP kernel option in lines 53~64 of the code and it's not LPAE, it supports supersection or section mapping.
- In lines 65~66 of code, map the physical address paddr to the virtual address range.
- If mapping lines 68~71 fails, try to unmap it.
- However, if the data cache uses PIPT or VIPT non-aliasing, such as in the armv73 and armv7 architectures, it does not
  need to be flushed

find_static_vm_paddr()

arch/arm/mm/ioremap.c

```
01 static struct static_vm *find_static_vm_paddr(phys_addr_t paddr,
02                         size_t size, unsigned int mtype)
03 {
04         struct static_vm *svm;
05         struct vm_struct *vm;
06
07         list_for_each_entry(svm, &static_vmlist, list) {
08                 vm = &svm->vm;
09                 if (!(vm->flags & VM_ARM_STATIC_MAPPING))
10                         continue;
11                 if ((vm->flags & VM_ARM_MTYPE_MASK) != VM_ARM_MTYPE(mtyp
   e))
12                         continue;
13
14                 if (vm->phys_addr > paddr ||
15                         paddr + size - 1 > vm->phys_addr + vm->size - 1)
16                         continue;
17
18                 return svm;
19         }
20
21         return NULL;
22 }
```

If the requested area is included in one of the area entries mapped to the vmalloc space and is already static-mapped to the same mapping type, it returns the vm_struct information. Otherwise, it returns null.

- In line 7 of code, loop around all entries in the global static_vmlist.
- In line 9~10 of the code, if the area does not use the VM_ARM_STATIC_MAPPING flag, skip it.
- If the type mapped to the area in line 11~12 is different from the request mapping type, skip it.
- In line 14~16 of the code, if the scope of the existing area does not include the scope of the request, skip it.

remap_area_sections()

arch/arm/mm/ioremap.c

```
01 static int
02 remap_area_sections(unsigned long virt, unsigned long pfn,
03                     size_t size, const struct mem_type *type)
04 {
05         unsigned long addr = virt, end = virt + size;
06         pgd_t *pgd;
07         pud_t *pud;
08         pmd_t *pmd;
09
10         /*
11          * Remove and free any PTE-based mapping, and
12          * sync the current kernel mapping.
13          */
14         unmap_area_sections(virt, size);
15
16         pgd = pgd_offset_k(addr);
17         pud = pud_offset(pgd, addr);
18         pmd = pmd_offset(pud, addr);
```

```
        do {
                pmd[0] = __pmd(__pfn_to_phys(pfn) | type->prot_sect);
                pfn += SZ_1M >> PAGE_SHIFT;
                pmd[1] = __pmd(__pfn_to_phys(pfn) | type->prot_sect);
                pfn += SZ_1M >> PAGE_SHIFT;
                flush_pmd_entry(pmd);

                addr += PMD_SIZE;
                pmd += 2;
        } while (addr < end);

        return 0;
}
```

If a 1M section mapping is possible, map the corresponding pmd entry to the section page.

- In ARM, PMD entries consist of two entries: PMD[0] and PMD[1].

unmap_area_sections()

lib/ioremap.c

```
/*
 * Section support is unsafe on SMP - If you iounmap and ioremap a region,
 * the other CPUs will not see this change until their next context switch.
 * Meanwhile, (eg) if an interrupt comes in on one of those other CPUs
 * which requires the new ioremap'd region to be referenced, the CPU will
 * reference the _old_ region.
 *
 * Note that get_vm_area_caller() allocates a guard 4K page, so we need to
 * mask the size back to 1MB aligned or we will overflow in the loop below.
 */
static void unmap_area_sections(unsigned long virt, unsigned long size)
{
        unsigned long addr = virt, end = virt + (size & ~(SZ_1M - 1));
        pgd_t *pgd;
        pud_t *pud;
        pmd_t *pmdp;

        flush_cache_vunmap(addr, end);
        pgd = pgd_offset_k(addr);
        pud = pud_offset(pgd, addr);
        pmdp = pmd_offset(pud, addr);
        do {
                pmd_t pmd = *pmdp;

                if (!pmd_none(pmd)) {
                        /*
                         * Clear the PMD from the page table, and
                         * increment the vmalloc sequence so others
                         * notice this change.
                         *
                         * Note: this is still racy on SMP machines.
                         */
                        pmd_clear(pmdp);
                        init_mm.context.vmalloc_seq++;

                        /*
                         * Free the page table, if there was one.
                         */
```

```
39                          if ((pmd_val(pmd) & PMD_TYPE_MASK) == PMD_TYPE_T
ABLE)
40                                  pte_free_kernel(&init_mm, pmd_page_vaddr
(pmd));
41                  }
42
43                  addr += PMD_SIZE;
44                  pmdp += 2;
45          } while (addr < end);
46
47          /*
48           * Ensure that the active_mm is up to date - we want to
49           * catch any use-after-iounmap cases.
50           */
51          if (current->active_mm->context.vmalloc_seq != init_mm.context.v
malloc_seq)
52                  __check_vmalloc_seq(current->active_mm);
53
54          flush_tlb_kernel_range(virt, end);
55  }
```

Clear the pmd-mapped entry in the vmalloc space.

- In line 18 of the code, the request area should flush the data cache by the pmd section units. However, if the data cache uses PIPT or VIPT non-aliasing, such as in the armv7 and armv8 architectures, it does not need to be flushed
- In line 19~21 of the code, we find the pgd, pud, and pmd entry addresses corresponding to the request virtual address through the kernel page table.
- If PMD has already been mapped in code lines 25~33, clear the mapping.
- In line 34 of the code, the kernel increments the value of the vmalloc sequence in the memory context to indicate that the kernel's vmalloc properties have been updated.
- In line 39~40 of the code, if an existing pmd entry points to a table, it deallocates that table.
- Loop around by incrementing the PMD unit from code lines 43~45 to the end address.
- If lines 51~52 of the code are updated by comparing the kernel's vmalloc information with the current task's vmalloc information, copy the kernel's pgd-mapped vmalloc entries to the pgd table of the current task's memory descriptor.
- In line 54 of code, flush the TLB cache for that kernel zone

ioremap_page_range()

lib/ioremap.c

```
01  int ioremap_page_range(unsigned long addr,
02                         unsigned long end, phys_addr_t phys_addr, pgprot_
t prot)
03  {
04          pgd_t *pgd;
05          unsigned long start;
06          unsigned long next;
07          int err;
08
09          BUG_ON(addr >= end);
10
11          start = addr;
12          phys_addr -= addr;
13          pgd = pgd_offset_k(addr);
14          do {
15                  next = pgd_addr_end(addr, end);
```

```
16                 err = ioremap_pud_range(pgd, addr, next, phys_addr+addr,
   prot);
17                 if (err)
18                         break;
19         } while (pgd++, addr = next, addr != end);
20
21         flush_cache_vmap(start, end);
22
23         return err;
24 }
25 EXPORT_SYMBOL_GPL(ioremap_page_range);
```

Map a physical address to a given virtual address area with a prot attribute.

- In lines 11~12 of the code, keep the request virtual address in start for a while, and subtract the request virtual address from the physical address.
  - e.g. addr=0x1000_0000, phys_addr=0x1234_5000
    - phys_addr=0x0234_5000
- In line 13 of the code, we get the kernel pgd entry address that corresponds to the virtual address addr.
- In line 15 of the code, get the virtual address managed by the next pgd entry, but if it is the last, get the end value.
- In line 16~18 of code, map the pud entries in the addr ~ next virtual address range within one pgd entry range, and escape the loop in case of an error.
- At line 19 of the code, select the next pgd entry and the next virtual address, and repeat the loop until the end.
- Flush the mapped virtual address area in line 21 of code. However, if the data cache uses PIPT or VIPT non-aliasing, such as in the armv7 and armv8 architectures, it does not need to be flushed.

ioremap_pud_range()

lib/ioremap.c

```
01 static inline int ioremap_pud_range(pgd_t *pgd, unsigned long addr,
02                 unsigned long end, phys_addr_t phys_addr, pgprot_t prot)
03 {
04         pud_t *pud;
05         unsigned long next;
06
07         phys_addr -= addr;
08         pud = pud_alloc(&init_mm, pgd, addr);
09         if (!pud)
10                 return -ENOMEM;
11         do {
12                 next = pud_addr_end(addr, end);
13                 if (ioremap_pmd_range(pud, addr, next, phys_addr + addr,
   prot))
14                         return -ENOMEM;
15         } while (pud++, addr = next, addr != end);
16         return 0;
17 }
```

On a pud basis, a physical address is mapped to a given virtual address area with a prot attribute.

- In line 7 of the code, subtract the request virtual address from the physical address.
  - e.g. addr=0x1000_0000, phys_addr=0x1_1234_5000

    - phys_addr=0x1_0234_5000
- In line 8~10 of the code, we are assigned a table for PUD. If it fails, the function exits with a -ENOMEM error.
- In line 12 of the code, get the virtual address managed by the next PUD entry, but if it's the last, get the end value.
- In lines 13~14 of code, map one PUD entry to the PMD entries in the addr ~ next virtual address range. If it fails, exit the function with the -ENOMEM result.
- In section 15 of the code, select the next PUD entry and the next virtual address, and repeat the loop until the end.

ioremap_pmd_range()

lib/ioremap.c

```
01 static inline int ioremap_pmd_range(pud_t *pud, unsigned long addr,
02                 unsigned long end, phys_addr_t phys_addr, pgprot_t prot)
03 {
04         pmd_t *pmd;
05         unsigned long next;
06
07         phys_addr -= addr;
08         pmd = pmd_alloc(&init_mm, pud, addr);
09         if (!pmd)
10                 return -ENOMEM;
11         do {
12                 next = pmd_addr_end(addr, end);
13                 if (ioremap_pte_range(pmd, addr, next, phys_addr + addr,
   prot))
14                         return -ENOMEM;
15         } while (pmd++, addr = next, addr != end);
16         return 0;
17 }
```

In pmd, the physical address is mapped to a given virtual address area with a prot attribute.

- In line 7 of the code, subtract the request virtual address from the physical address.
- In line 8~10 of the code, we are assigned a table for pmd. If it fails, the function exits with a -ENOMEM error.
- In line 12 of the code, get the virtual address managed by the next pmd entry, but if it is the last, get the end value.
- In code lines 13~14, map the pte entries in the addr ~ next virtual address range within one PMD entry range. If it fails, exit the function with the -ENOMEM result.
- In section 15 of the code, select the next pmd entry and the next virtual address, and repeat the loop until the end.

ioremap_pte_range()

lib/ioremap.c

```
01 static int ioremap_pte_range(pmd_t *pmd, unsigned long addr,
02                 unsigned long end, phys_addr_t phys_addr, pgprot_t prot)
03 {
04         pte_t *pte;
05         u64 pfn;
```

```
06
07          pfn = phys_addr >> PAGE_SHIFT;
08          pte = pte_alloc_kernel(pmd, addr);
09          if (!pte)
10                  return -ENOMEM;
11          do {
12                  BUG_ON(!pte_none(*pte));
13                  set_pte_at(&init_mm, addr, pte, pfn_pte(pfn, prot));
14                  pfn++;
15          } while (pte++, addr += PAGE_SIZE, addr != end);
16          return 0;
17  }
```

In a PTE unit, a physical address page is mapped to a given virtual address area with a prot attribute.

- pfn is found in line 7 of code.
- In line 8~10 of the code, we are assigned a table for the PTE. If it fails, the function exits with a -ENOMEM error.
- In lines 13~14 of the code, map the pte entry corresponding to the virtual address addr to the pfn physics page to the prot type and increment the pfn.
- On line 15 of the code, increment the next pte entry, increment the virtual address by the next page, and loop through the loop until the end.

IO Unmapping

iounmap()

arch/arm/include/asm/io.h

```
1  #define iounmap                         __arm_iounmap
```

Unmap the io of the request physical address.

__arm_iounmap()

arch/arm/mm/ioremap.c

```
1  void __arm_iounmap(volatile void __iomem *io_addr)
2  {
3          arch_iounmap(io_addr);
4  }
5  EXPORT_SYMBOL(__arm_iounmap);
```

Unmap the io of the request physical address.

arch_iounmap()

arch/arm/mm/ioremap.c

```
1  void (*arch_iounmap)(volatile void __iomem *) = __iounmap;
```

At compile time, the global arch_iounmap function pointer variable contains the address of the __iounmap() function.

__iounmap()

arch/arm/mm/ioremap.c

```
01 void __iounmap(volatile void __iomem *io_addr)
02 {
03         void *addr = (void *)(PAGE_MASK & (unsigned long)io_addr);
04         struct static_vm *svm;
05
06         /* If this is a static mapping, we must leave it alone */
07         svm = find_static_vm_vaddr(addr);
08         if (svm)
09                 return;
10
11 #if !defined(CONFIG_SMP) && !defined(CONFIG_ARM_LPAE)
12         {
13                 struct vm_struct *vm;
14
15                 vm = find_vm_area(addr);
16
17                 /*
18                  * If this is a section based mapping we need to handle
   it
19                  * specially as the VM subsystem does not know how to ha
   ndle
20                  * such a beast.
21                  */
22                 if (vm && (vm->flags & VM_ARM_SECTION_MAPPING))
23                         unmap_area_sections((unsigned long)vm->addr, vm-
   >size);
24         }
25 #endif
26
27         vunmap(addr);
28 }
```

Unmap the io of the request physical address.

- In line 7~9 of code, if there is a static mapping to the request address in the vmalloc space, it just exits the function.
- In line 11~25 of the code, if the system (built kernel) does not support SMP and LPAE, it will be turned off if the section is mapped to that address.
- In line 27 of code, unmap the io physical address mapped to the vmalloc space by calling the vunmap() function.

Checking the mapping status of a vmalloc space

You can check which API the physical address is mapped to the vmalloc virtual address space, as follows:

```
01 $ sudo cat /proc/vmallocinfo
02 0xba800000-0xbb000000 8388608 iotable_init+0x0/0xb8 phys=3a800000 iorema
   p
03 0xbb804000-0xbb806000    8192 raw_init+0x50/0x148 pages=1 vmalloc
04 0xbb806000-0xbb809000   12288 pcpu_mem_zalloc+0x44/0x80 pages=2 vmalloc
05 0xbb899000-0xbb89c000   12288 pcpu_mem_zalloc+0x44/0x80 pages=2 vmalloc
06 0xbb89c000-0xbb89e000    8192 dwc_otg_driver_probe+0x650/0x7a8 phys=3f00
   6000 ioremap
```

```
0xbb89e000-0xbb8a0000    8192 devm_ioremap_nocache+0x40/0x7c phys=3f300000 ioremap
0xbbc00000-0xbbc83000  536576 bcm2708_fb_set_par+0x108/0x13c phys=3db79000 ioremap
0xbc9f7000-0xbc9ff000   32768 SyS_swapon+0x618/0xf6c pages=7 vmalloc
0xbc9ff000-0xbca01000    8192 SyS_swapon+0x850/0xf6c pages=1 vmalloc
0xbcc39000-0xbcc3b000    8192 SyS_swapon+0xaa0/0xf6c pages=1 vmalloc
0xbce7b000-0xbce7f000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xbce83000-0xbce87000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xbce8b000-0xbce8f000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xbce93000-0xbce97000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xbce9b000-0xbce9f000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xbcea7000-0xbceab000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xbceab000-0xbceaf000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xbceb7000-0xbceb9000    8192 bpf_prog_alloc+0x44/0xb0 pages=1 vmalloc
0xbf509000-0xbf50d000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xbf50d000-0xbf511000   16384 n_tty_open+0x20/0xe4 pages=3 vmalloc
0xf3000000-0xf3001000    4096 iotable_init+0x0/0xb8 phys=3f000000 ioremap
0xf3003000-0xf3004000    4096 iotable_init+0x0/0xb8 phys=3f003000 ioremap
0xf3007000-0xf3008000    4096 iotable_init+0x0/0xb8 phys=3f007000 ioremap
0xf300b000-0xf300c000    4096 iotable_init+0x0/0xb8 phys=3f00b000 ioremap
0xf3100000-0xf3101000    4096 iotable_init+0x0/0xb8 phys=3f100000 ioremap
0xf3200000-0xf3201000    4096 iotable_init+0x0/0xb8 phys=3f200000 ioremap
0xf3201000-0xf3202000    4096 iotable_init+0x0/0xb8 phys=3f201000 ioremap
0xf3215000-0xf3216000    4096 iotable_init+0x0/0xb8 phys=3f215000 ioremap
0xf3980000-0xf39a0000  131072 iotable_init+0x0/0xb8 phys=3f980000 ioremap
0xf4000000-0xf4001000    4096 iotable_init+0x0/0xb8 phys=40000000 ioremap
0xfea50000-0xff000000 5963776 pcpu_get_vm_areas+0x0/0x5e0 vmalloc
```

consultation

- Early ioremam (http://jake.dothome.co.kr/early-ioremap) | 문c
- Vmap (http://jake.dothome.co.kr/vmap) | Qc

LEAVE A COMMENT

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

Comments

name *

email *

Website

WRITE A COMMENT

‹ delayacct_init() (http://jake.dothome.co.kr/delayacct_init/)

Early ioremap › (http://jake.dothome.co.kr/early-ioremap/)

Munc Blog (2015 ~ 2024)