Fixmap

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

<kernel v5.10>

Fixmap Main Uses

The fixed mapping area (hereinafter referred to as the fixmap area) is the space where the virtual address is fixed at compile time. Therefore, fixmap is usually used when mapping is required before the dynamic mapping subsystem (vmap, etc.) is activated. For example, a console device is temporarily mapped to this space for use before it is officially initialized. It is also used at runtime when the kernel code needs to be changed or the page table needs to be updated.
The fixmap is divided into several blocks for each purpose. Virtual addresses are determined at compile time, and physical addresses are mapped and used at runtime (including boot time) using the API for fixmap. Before virtual address mapping (VMAP) is enabled, a small number of specific resources use these fixed mapping address spaces.

Its uses vary slightly depending on the architecture and kernel version, but the main uses are as follows. Other uses are further described in the Fixmap slot classification.

1) IO Device Early Mapping

- There are two ways to map devices at early bootup time when the ioremap() function is not available.
  - Static devices are used for fixed mapping. (Early Console, etc.)
  - At the beginning of the bootup running time, when regular device mapping (ioremap) is not available, the device can be temporarily mapped to the fixmap virtual address area via the early_ioremap() function.
    - A total of 7 times, each capable of mapping 256K

2) Kernel mapping of highmem physical memory

- In the case of the ARM32 kernel, the highmem page in the physical area is mapped to the fixmap virtual address area assigned by the cpu id.
  - stack based kmap_atomic
    - Initially, the mapped index slot was fixed according to the CPU ID and usage type, but now it is operated using push/pop in the same way as the stack by the number of KM_TYPE_NR (ARM=20) on the CPU ID.
      - This means that each CPU is assigned 20 mapping slots.
    - Note: mm: stack based kmap_atomic() (https://github.com/torvalds/linux/commit/3e4d3af501cccdc8a8cca41bdbe57d54ad7e7e73)

    - Use the kmap_atomic() function to map the highmem page using the fixmap area that corresponds to the current CPU. If it has already been mapped to a kmap area, it returns its virtual address.
    - ZONE_HIGHMEM uses several mapping methods in the kernel and always iterates between mapping and unmapping, so the access speed is overhead for mapping, which always results in slower performance compared to pre-mapped ZONE_NORMAL. Of course, unlike kernels, at the user level, each task is mapped to a very large user address space (1G, 2G, or 3G, depending on the configuration).
- 64-bit systems have a very large virtual address space, so all the physical memory in the system can be mapped. Therefore, there is no need to run highmem in this case.

3) Kernel code changes

- When changing kernel code that is set to read only, the fixmap virtual address area is used temporarily.

Simple comparison with other mapping methods

- vmap
    - It can be used to map multiple pages over a long period of time, and it maps to a fairly large vmalloc address space.
        - ARM32
            - 240M SPACE
        - ARM64
            - CONFIG_VM_BITS varies by size, and you can assume that half of the VM space is almost half of the VM space, excluding some vmemmap, pci io, fixmap, kimage, module image area, etc.
            - For example, if you use CONFIG_VM_BITS=39, the VM size is 512G. Of these, the vmalloc space is about 246G.
- kmap
    - kmap address space for a certain amount of time. Once the mapping is complete, it is rescheduled and retains the mapping even if it is changed to another task.
        - ARM32
            - 2M SPACE
- fixmap
    - It can be used for a very, very short mapping of highmem pages and maps to the fixmap address space. It doesn't sleep, so it can be used in an interrupt context.
        - It must be unmapped before it can be scheduled and replaced with another task.
    - Other IO areas, etc., are fixed and mapped at boot-up time.
    - ARM32
        - It has grown from 2M to > 3M space.
    - ARM64
        - It varies between kernel versions and kernel options, but currently around 6M space

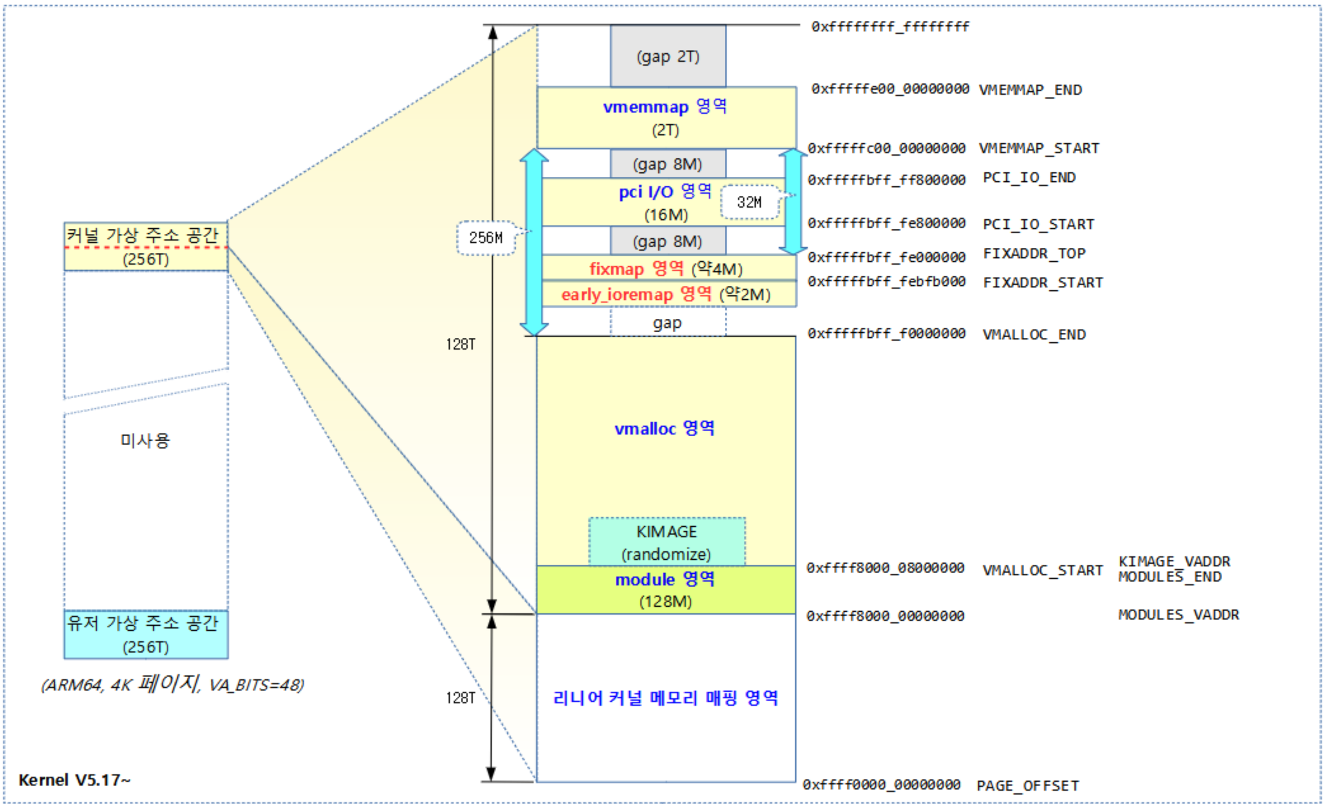
Fixmap Virtual Address Area

- The fixmap virtual address area is located in different architectures and has a different size.
  - ARM32
    - Currently, 3M's virtual address space is used to add index slots (768 ~ 0) to 767 pages from the highest address downward.
    - Use a 0M area from FIXADDR_START (0xffc0000_0) ~ FIXADDR_END (0xfff0000_3).
      - Previously, we used a 0M area from 0xffc0000_0 ~ 0xffe0000_2.
      - Note: ARM: expand fixmap region to 3MB (https://github.com/torvalds/linux/commit/836a24183273e9db1c092246bd8e306b297d9917)
    - FIXADDR_TOP range is FIXADDR_END – PAGE_SIZE (4K).
      - Index assignment is when FIXADDR_TOP (0xffef_f000) is index 0 and the index number is increased in the downward direction.
      - Index numbers can range from 0 up to 0x2ff (767).
  - ARM64
    - It uses about 6M of virtual address space and uses index slots from the highest address downward.

Kernel Address Space by Kernel Version Layout

The following figure shows the kernel v5.17~ kernel address space layout. (VA_BITS=48)

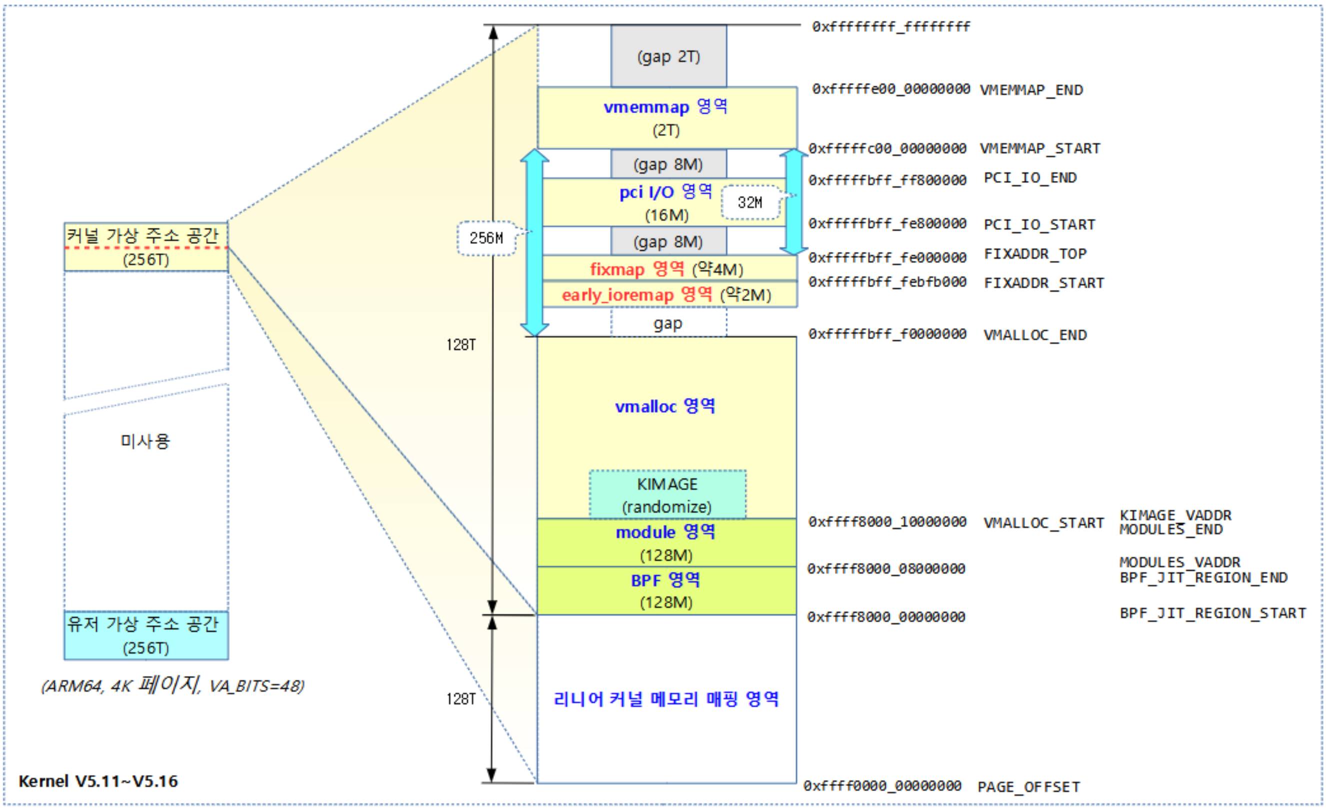
- The bpf area restriction of 128M has been lifted, allowing the use of vmalloc space.
  - arm64/bpf: Remove 128MB limit for BPF JIT programs (https://github.com/torvalds/linux/commit/b89ddf4cca43f1269093942cf5c4e457fd45c335) (2021, v5.17-rc1)



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/fixmap-10.png)

The following figure shows the kernel v5.11~ kernel address space layout. (VA_BITS=48)

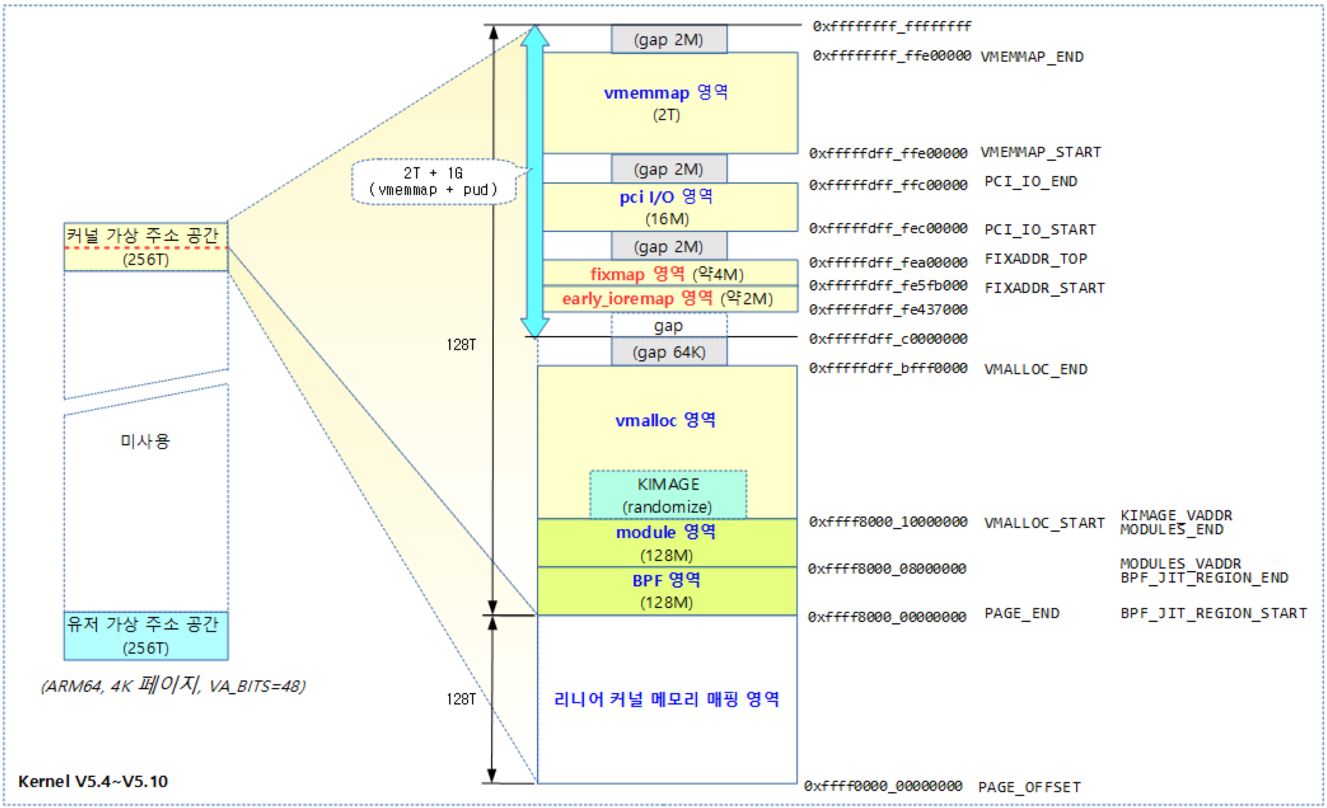
- The vmemmap, pci, and fixmap areas have been made some adjustments.
  - arm64: mm: make vmemmap region a projection of the linear region (https://github.com/torvalds/linux/commit/8c96400d6a39be763130a5c493647c57726f7013) (2020, v5.11-rc1)
  - arm64: mm: tidy up top of kernel VA space (https://github.com/torvalds/linux/commit/9ad7c6d5e75b160c9ce5775db610d964af45b83f) (2020, v5.11-rc1)



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/fixmap-9c.png)

The following figure shows the kernel address space layout for kernels v5.4 through v5.10. (VA_BITS=48)

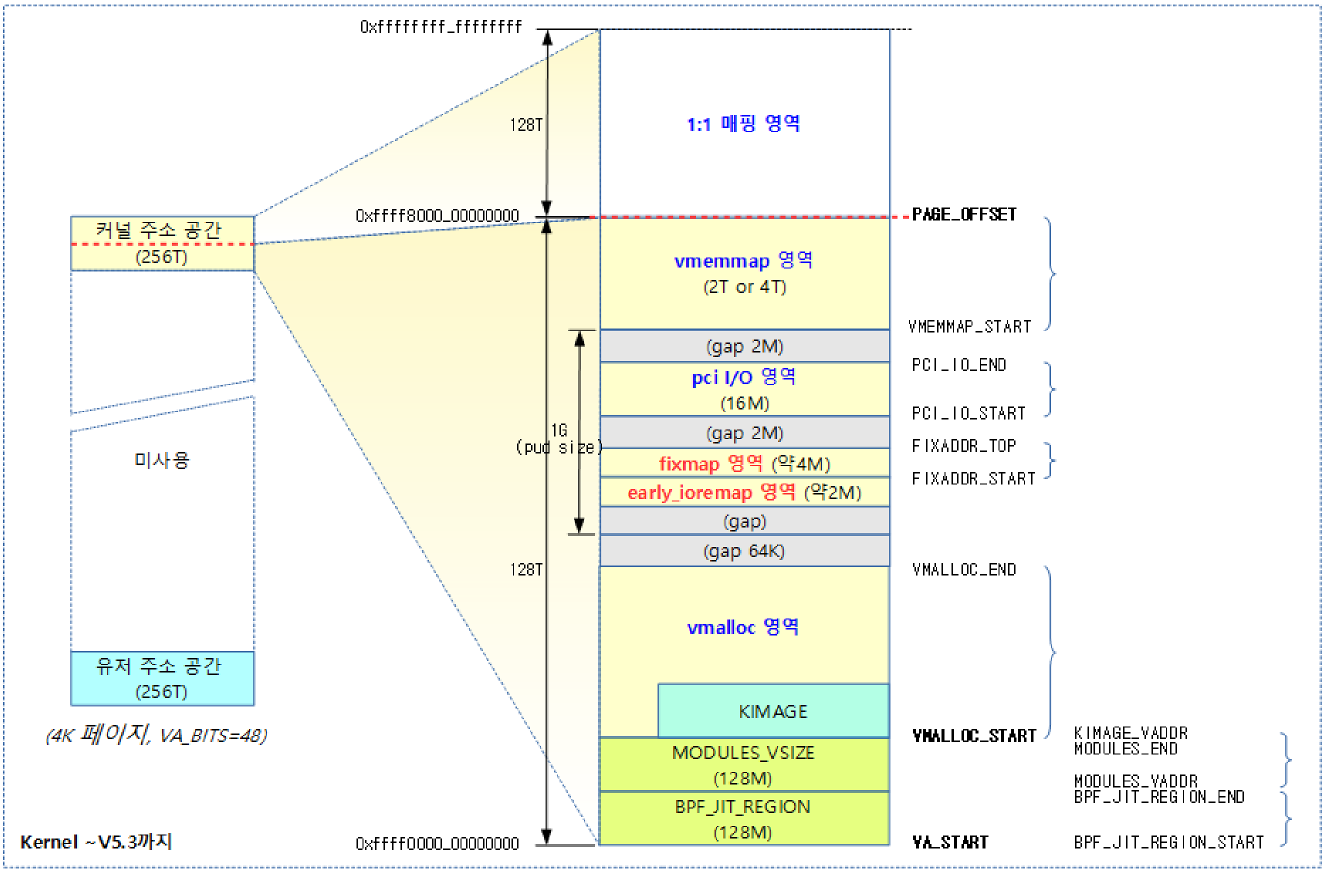
- As of kernel v5.4, you can see that half of the kernel area has been flipped. VA_START was renamed to PAGE_END instead.
  - Note: arm64: memory: rename VA_START to PAGE_END (https://github.com/torvalds/linux/commit/77ad4ce69321abbe26ec92b2a2691a66531eb688#diff-b58f478335b857b9aa2599a7e129552f) (2019, v5.4-rc1)
- As of kernel v5.4, the 1:1 mapping area has been lowered to allow KASAN's shadow size to be changed at boot time.
  - Note: arm64: mm: Flip kernel VA space (https://github.com/torvalds/linux/commit/14c127c957c1c6070647c171e72f06e0db275ebf#diff-b58f478335b857b9aa2599a7e129552f) (2019, v5.4-rc1)
- The 32T KASAN area is omitted, but if you use this option, the bpf, module, and vmalloc starts will go up that much. (The end of vmalloc is the same)
- Depending on the kernel options, the location of the fixmap can vary from page to page.



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/fixmap-8c.png)

The following figure shows the kernel address space layout up to kernel ~v5.3. (VA_BITS=48)

- The size of the vmemmap space depends on the size of the page descriptor (struct page), which depends on the compilation options
  - If it's 64 bytes or less, use the 2T region.
  - If it exceeds 64 bytes, use the 4T region.



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/fixmap-7b.png)

Fixmap Index Slot

- The virtual address is fixed according to the fixmap index slot number.
  - e.g. ARM32
    - index=0 -> vaddr=0xffef_f000 (FIXADDR_TOP)
    - index=1 -> vaddr=0xffef_e000
    - ...
    - index=767 -> vaddr=0xffc0_0000 (FIXADDR_START)

Specific page mapping examples)

Page 0, which begins with the physical address 4000x0000_1, maps the fixmap to the index of slot 1.

- set_fixmap(1, 0x4000_0000)

Fixmap Slot Classification

Fixmaps are provided in slots for different purposes depending on the architecture and kernel version.

- HOLE
  - In the case of ARM32, it is not used.
  - In the case of ARM64, 1 slot is provided for debugging purposes, and is currently not used as a spare entry page.

- Added in kernel v3.19-rc1.
  - Note: arm64: Add FIX_HOLE to permanent fixed addresses (https://github.com/torvalds/linux/commit/dab78b6dcb2bfc90038f35ada826844273dde4d6#diff-54386009a8506bfeb240099dfd205550)
- FDT
  - In the case of ARM32, it is not used.
  - In the case of the ARM64, it provides a slot that covers the device tree (FDT) of 4M.
    - The FDT is up to 2M, but because it uses alignment in 2M increments, it requires an area of up to 4M.
  - Added in kernel v4.2-rc1.
    - Note: arm64: use fixmap region for permanent FDT mapping (https://github.com/torvalds/linux/commit/61bd93ce801bb6df36eda257a9d2d16c02863cdd#diff-54386009a8506bfeb240099dfd205550)
- EARLYCON
  - One index slot is used for input and output before regular mapping for the purpose of using the serial device as a console. (early console)
  - This area was added in kernel v2015.8-rc4 in August 3.
    - Note: ARM: 8415/1: early fixmap support for earlycon (https://github.com/torvalds/linux/commit/a5f4c561b3b19a9bc43a81da6382b0098ebbc1fb)
- KMAP
  - Highmem, which is only used on 32-bit systems, is the space used when mapping the physical memory area, and the index slot used varies depending on the number of CPUs (NR_CPUS).
    - It was used when Fixmap was first introduced.
  - In the case of ARM32, there are 20 index slots depending on the number of CPUs.
    - The number has been increased from 16 to 20.
  - In the case of x86_32, there are 41 index slots depending on the number of CPUs.
- TEXT_POKE
  - When kernel code, kprobes, static keys, etc., are used, the read-only kernel code is changed, and these codes are mapped for a while using one or two slots before being changed.
  - In the case of the ARM32, there are two slots.
  - In the case of the ARM64, there are two slots.
- APEI_GHES
  - If you are currently using the GHES driver on ARM64 and X86_64, you will be given 2 slots.
  - The GHES driver uses fixmap to prevent the ioremap_page_range() function from being used for use in the irq context.
  - Added in kernel v4.15-rc.
    - Note: ACPI / APEI: Replace ioremap_page_range() with fixmap (https://github.com/torvalds/linux/commit/4f89fa286f6729312e227e7c2d764e8e7b9d340e)
  - This was added in kernel v5.1.

    - Note: firmware: arm_sdei: Add ACPI GHES registration helper (https://github.com/torvalds/linux/commit/f96935d3bc38a5f4b5188b6470a10e3fb8c3f0cc#diff-54386009a8506bfeb240099dfd205550)
- ENTRY_TRAMP
  - For security, KASLR (Kernel Address Sanitizer Location Randomization) technology is used to hide the kernel location in the user space. In addition, a separate top-level page table (pgd) for KPTI (Kernel Page Table Isolation, aka KAISER), which is operated by preparing a kernel page table separately to prevent access to the kernel area from the user space, is used on this ENTRY_TRAMP page. Kernels that use these options have to flush TLB every time they switch between kernel and user, resulting in a performance loss of about 5%. Future CPUs are said to be designed to use technology that completely separates kernel space access from user space, so that performance is not degraded without using this option.
  - Added in kernel v4.16-rc1.
    - KAISER: hiding the kernel from user space (https://lwn.net/Articles/738975/) | LWN.net
    - The current state of kernel page-table isolation (https://lwn.net/Articles/741878/) | LWN.net
    - arm64: mm: Map entry trampoline into trampoline and kernel page tables (https://github.com/torvalds/linux/commit/51a0048beb449682d632d0af52a515adb9f9882e)
    - arm64: kaslr: Put kernel vectors address in separate data page (https://github.com/torvalds/linux/commit/6c27c4082f4f70b9f41df4d0adf51128b40351df)
- BTMAPS
  - This is where devices are temporarily mapped via early_ioremap() at early bootup time when regular ioremap() is not available.
  - Up to seven mappings can be made, each of which can use 7K.
  - Added in kernel v3.15-rc1.
    - Note: arm64: add early_ioremap support (https://github.com/torvalds/linux/commit/bf4b558eba920a38f91beb5ee62a8ce2628c92f7#diff-54386009a8506bfeb240099dfd205550)
- FIX_PTE, FIX_PMD, FIX_PUD, FIX_PGD
  - It is used at runtime to generate kernel page tables for atomic processing for the purpose of applying them to TLB without problems, and uses a total of 1 index slots, 4 for each.
  - The kernel page table has been changed to read-only, and whenever you modify a page table entry, you use this area to modify the entry.
  - Added in kernel v.4.6-rc1.
    - Note: arm64: mm: add functions to walk tables in fixmap (https://github.com/torvalds/linux/commit/961faac114819a01e627fe9c9c82b830bb3849d4#diff-54386009a8506bfeb240099dfd205550)

fixed_address – ARM64

arch/arm64/include/asm/fixmap.h

```
/*
 * Here we define all the compile-time 'special' virtual
 * addresses. The point is to have a constant address at
 * compile time, but to set the physical address only
 * in the boot process.
 *
 * Each enum increment in these 'compile-time allocated'
 * memory buffers is page-sized. Use set_fixmap(idx,phys)
 * to associate physical memory with a fixmap index.
 */
```

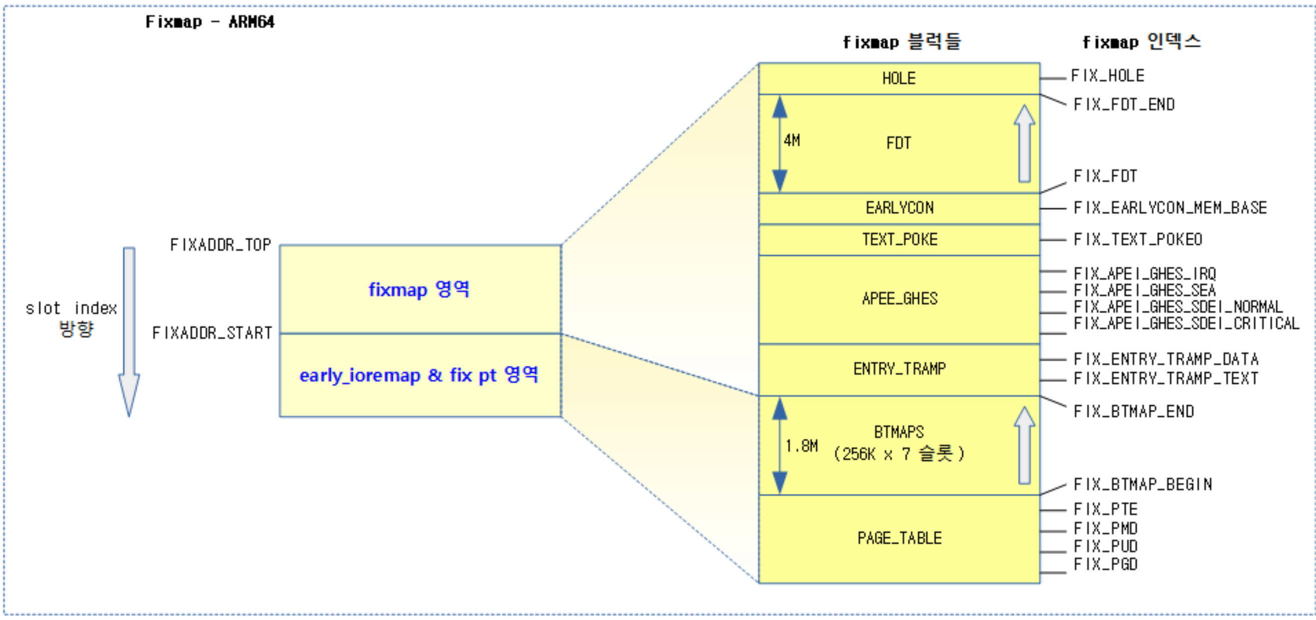
```
enum fixed_addresses {
        FIX_HOLE,

        /*
         * Reserve a virtual window for the FDT that is 2 MB larger than the
         * maximum supported size, and put it at the top of the fixmap region.
         * The additional space ensures that any FDT that does not exceed
         * MAX_FDT_SIZE can be mapped regardless of whether it crosses any
         * 2 MB alignment boundaries.
         *
         * Keep this at the top so it remains 2 MB aligned.
         */
#define FIX_FDT_SIZE            (MAX_FDT_SIZE + SZ_2M)
        FIX_FDT_END,
        FIX_FDT = FIX_FDT_END + FIX_FDT_SIZE / PAGE_SIZE - 1,

        FIX_EARLYCON_MEM_BASE,
        FIX_TEXT_POKE0,

#ifdef CONFIG_ACPI_APEI_GHES
        /* Used for GHES mapping from assorted contexts */
        FIX_APEI_GHES_IRQ,
        FIX_APEI_GHES_SEA,
#ifdef CONFIG_ARM_SDE_INTERFACE
        FIX_APEI_GHES_SDEI_NORMAL,
        FIX_APEI_GHES_SDEI_CRITICAL,
#endif
#endif /* CONFIG_ACPI_APEI_GHES */

#ifdef CONFIG_UNMAP_KERNEL_AT_EL0
        FIX_ENTRY_TRAMP_DATA,
        FIX_ENTRY_TRAMP_TEXT,
#define TRAMP_VALIAS            (__fix_to_virt(FIX_ENTRY_TRAMP_TEXT))
#endif /* CONFIG_UNMAP_KERNEL_AT_EL0 */
        __end_of_permanent_fixed_addresses,

        /*
         * Temporary boot-time mappings, used by early_ioremap(),
         * before ioremap() is functional.
         */
#define NR_FIX_BTMAPS           (SZ_256K / PAGE_SIZE)
#define FIX_BTMAPS_SLOTS        7
#define TOTAL_FIX_BTMAPS        (NR_FIX_BTMAPS * FIX_BTMAPS_SLOTS)

        FIX_BTMAP_END = __end_of_permanent_fixed_addresses,
        FIX_BTMAP_BEGIN = FIX_BTMAP_END + TOTAL_FIX_BTMAPS - 1,

        /*
         * Used for kernel page table creation, so unmapped memory may be used
         * for tables.
```

```
         */
        FIX_PTE,
        FIX_PMD,
        FIX_PUD,
        FIX_PGD,

        __end_of_fixed_addresses
};
```

The fixmap is divided into two main areas and has the following characteristics.

- __end_of_permanent_fixed_addresses
    - It is a persistent mapping space that is not unmapped after boot.
- __end_of_fixed_addresses
    - The last area of the fixmap, and the area after __end_of_permanent_fixed_addresses, is a space that can be mapped and unmapped.

The following image shows the ARM64 fixmap blocks.



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/fixmap-5a.png)

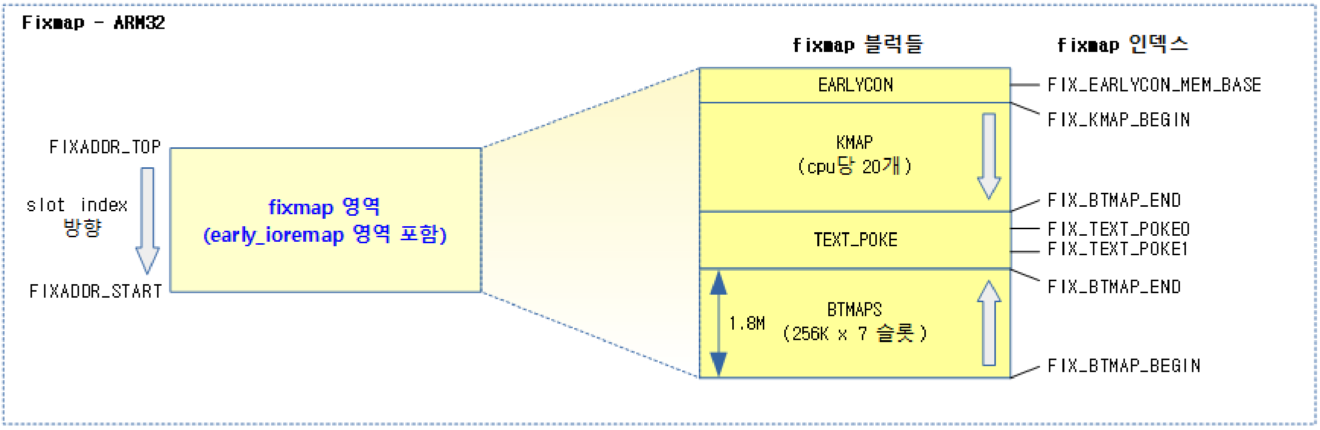
fixed_address – ARM32

arch/arm/include/asm/fixmap.h

```
enum fixed_addresses {
        FIX_EARLYCON_MEM_BASE,
        __end_of_permanent_fixed_addresses,

        FIX_KMAP_BEGIN = __end_of_permanent_fixed_addresses,
        FIX_KMAP_END = FIX_KMAP_BEGIN + (KM_TYPE_NR * NR_CPUS) - 1,

        /* Support writing RO kernel text via kprobes, jump labels, etc. */
        FIX_TEXT_POKE0,
        FIX_TEXT_POKE1,

        __end_of_fixmap_region,

        /*
```

```
         * Share the kmap() region with early_ioremap(): this is guaranteed
         * not to clash since early_ioremap() is only available before
         * paging_init(), and kmap() only after.
         */
#define NR_FIX_BTMAPS           32
#define FIX_BTMAPS_SLOTS        7
#define TOTAL_FIX_BTMAPS        (NR_FIX_BTMAPS * FIX_BTMAPS_SLOTS)

        FIX_BTMAP_END = __end_of_permanent_fixed_addresses,
        FIX_BTMAP_BEGIN = FIX_BTMAP_END + TOTAL_FIX_BTMAPS - 1,
        __end_of_early_ioremap_region
};
```

The following image shows the ARM32 fixmap blocks.



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/fixmap-4a.png)

Initialize

early_fixmap_init() – ARM64

We want to use the FixMap virtual address area early, which can be used by mapping a specific physical address to some fixed virtual address area before dynamic mapping is enabled. To support these fixmap virtual address areas, we prepare a complete mapping table that can use the fixmap virtual address regions by concatenating the three static page tables bm_pud[], bm_pmd[], and bm_pte[] to the entries corresponding to the fixmap virtual address in the PGD table. Since it is not possible to use a regular mapping function and a regular memory allocator, the three-page table is staticly created and utilized at compile time. Note that the early_fixmap_init() function maps the bottom 3M of the entire fixmap area to cover first, excluding the FDT.

arch/arm64/mm/mmu.c

```
/*
 * The p*d_populate functions call virt_to_phys implicitly so they can't be used
 * directly on kernel symbols (bm_p*d). This function is called too early to use
 * lm_alias so __p*d_populate functions must be used to populate with the
```

```
 * physical address from __pa_symbol.
 */
```

```
01 void __init early_fixmap_init(void)
02 {
03         pgd_t *pgdp;
04         p4d_t *p4dp, p4d;
05         pud_t *pudp;
06         pmd_t *pmdp;
07         unsigned long addr = FIXADDR_START;
08 
09         pgdp = pgd_offset_k(addr);
10         p4dp = p4d_offset(pgdp, addr);
11         p4d = READ_ONCE(*p4dp);
12         if (CONFIG_PGTABLE_LEVELS > 3 &&
13             !(p4d_none(p4d) || p4d_page_paddr(p4d) == __pa_symbol(bm_pud))) {
14                 /*
15                  * We only end up here if the kernel mapping and the fixmap
16                  * share the top level pgd entry, which should only happen on
17                  * 16k/4 levels configurations.
18                  */
19                 BUG_ON(!IS_ENABLED(CONFIG_ARM64_16K_PAGES));
20                 pudp = pud_offset_kimg(p4dp, addr);
21         } else {
22                 if (p4d_none(p4d))
23                         __p4d_populate(p4dp, __pa_symbol(bm_pud), PUD_TYPE_TABLE);
24                 pudp = fixmap_pud(addr);
25         }
26         if (pud_none(READ_ONCE(*pudp)))
27                 __pud_populate(pudp, __pa_symbol(bm_pmd), PMD_TYPE_TABLE);
28         pmdp = fixmap_pmd(addr);
29         __pmd_populate(pmdp, __pa_symbol(bm_pte), PMD_TYPE_TABLE);
30 
31         /*
32          * The boot-ioremap range spans multiple pmds, for which
33          * we are not prepared:
34          */
35         BUILD_BUG_ON((__fix_to_virt(FIX_BTMAP_BEGIN) >> PMD_SHIFT)
36                      != (__fix_to_virt(FIX_BTMAP_END) >> PMD_SHIFT));
37 
38         if ((pmdp != fixmap_pmd(fix_to_virt(FIX_BTMAP_BEGIN)))
39              || pmdp != fixmap_pmd(fix_to_virt(FIX_BTMAP_END))) {
40                 WARN_ON(1);
41                 pr_warn("pmdp %p != %p, %p\n",
42                         pmdp, fixmap_pmd(fix_to_virt(FIX_BTMAP_BEGIN)),
43                         fixmap_pmd(fix_to_virt(FIX_BTMAP_END)));
44                 pr_warn("fix_to_virt(FIX_BTMAP_BEGIN): %08lx\n",
45                         fix_to_virt(FIX_BTMAP_BEGIN));
46                 pr_warn("fix_to_virt(FIX_BTMAP_END):   %08lx\n",
47                         fix_to_virt(FIX_BTMAP_END));
48 
49                 pr_warn("FIX_BTMAP_END:       %d\n", FIX_BTMAP_END);
50                 pr_warn("FIX_BTMAP_BEGIN:     %d\n", FIX_BTMAP_BEGIN);
51         }
52 }
```

Enable the fixmap virtual address zone.

- In line 7 of code, assign the lowest virtual address of the fixmap area to addr.
  - Unlike arm32, arm64 has a DTB equivalent to 4M at the top address of the fixmap, and most of the items you want to use at the beginning are below the fixmap area, so we plan

to activate the lower address first.

- In line 9~11 of the code, read the p4d entry value corresponding to the virtual address addr.
  - From the pgd table, we get the pgd entry pointer, pgdp, then the next level, p4dp, and then we read the p4d entry value from this pointer and assign it to p4d.
- In line 12~20 of code, if the page table conversion level is more than 4 levels, and the kernel uses 16K as the page size, there will only be a maximum of 4 p2d entries. One of them is a virtual address space for kernel memory, and the other is used to contain all of the several spatial addresses used by the kernel for various purposes, including the kernel image area or the fixmap area. In other words, the kernel image and fixmap area exist in a single bm_pud[ ] table. In this case, the bm_pud[ ] page table is already enabled and in use for the kernel image, so there is no need to re-enable it for fixmap. So we get the pud entry pointer that corresponds to the fixmap start address right away.
  - This condition only allows the use of 4 levels + 16K pages, otherwise it will output a bug message.
- In other cases, from lines 21~25 of the code, the bm_pud[ ] table is not shared like the fixmap area and kernel image area, but is mainly used for the fixmap area. So, in order to use the fixmap area, we activate the bm_pud[ ] table by associating it with the pgd entry, and then get the pud entry pointer that corresponds to the fixmap start address.
  - Includes 4 levels + 4K pages and all pages in 3 (4K, 16K, 64K).
- In lines 26~27 of code, if there is no PMD table connected to the PUD, use the bm_pmd[] table to connect it.
- On line 28 of the code, we get the pmd entry pointer that corresponds to the addr address.
- If there is no PTE table linked to PMD on line 29, use the bm_pte[] table to connect it.
  - Map the bottom 2M of the entire fixmap area, excluding FDT, to cover it.
- In line 38~51 of the code, if the PMD entry address values in the PUD table corresponding to the beginning and end of the btmap area used by the early_ioremap() function are different from the PMD entry address values read above, a warning message is displayed.

bm_pte, bm_pmd, and bm_pud use a static array generated at kernel build time without any page allocation as a page table for fixmap

arch/arm64/mm/mmu.c

```
static pte_t bm_pte[PTRS_PER_PTE] __page_aligned_bss;
static pmd_t bm_pmd[PTRS_PER_PMD] __page_aligned_bss __maybe_unused;
static pud_t bm_pud[PTRS_PER_PUD] __page_aligned_bss __maybe_unused;
```

Mapping two virtual spaces in a kernel image

- The kernel image (kimage) is mapped to two virtual spaces at bootup time.
  - 1) Linear Mapping Virtual Space
    - This is the space where the entire DRAM that the kernel is loaded is mapped to.
    - e.g. 0xfff_0000_0000_0000~ (4K, 4 level pages)
  - 2) kimage virtual space
    - This is a space where only the kernel image is mapped.

        - e.g. 0xffff_8000_1000_0000~ (4K, 4 level pages, KASLR=n)

Translation APIs for Virtual and Physical Addresses

- The following API is used to convert 1) linear mapping (lm) virtual addresses and physical addresses.
    - Restrictions on Use
        - It can be used after linear mapping is completed via the paging_init() function.
        - CONFIG_DEBUG_VIRTUAL Displays a warning message if the kernel option does not use the lm virtual address.
    - virt_to_phys(), __virt_to_phys(), __va()
    - phys_to_virt(), __phys_to_virt(), __pa()
    - lm_alias()
        - Convert the lm virtual address for the kernel symbol.
- The following API is used to convert the virtual address of the kernel symbol 2) to the physical address.
    - Restrictions on Use
        - none
    - __kimg_to_phys()
    - __phys_to_kimg()
    - __pa_symbol()
        - Converts kernel symbol virtual addresses to physical addresses.
- The following API is used to convert 1) lm (linear mapping) virtual addresses and pages.
    - Restrictions on Use
        - It is available after vmemmap, which consists of an array of page descriptors, is enabled.
    - virt_to_page()
    - page_to_virt()

Using the __p*d_populate() function

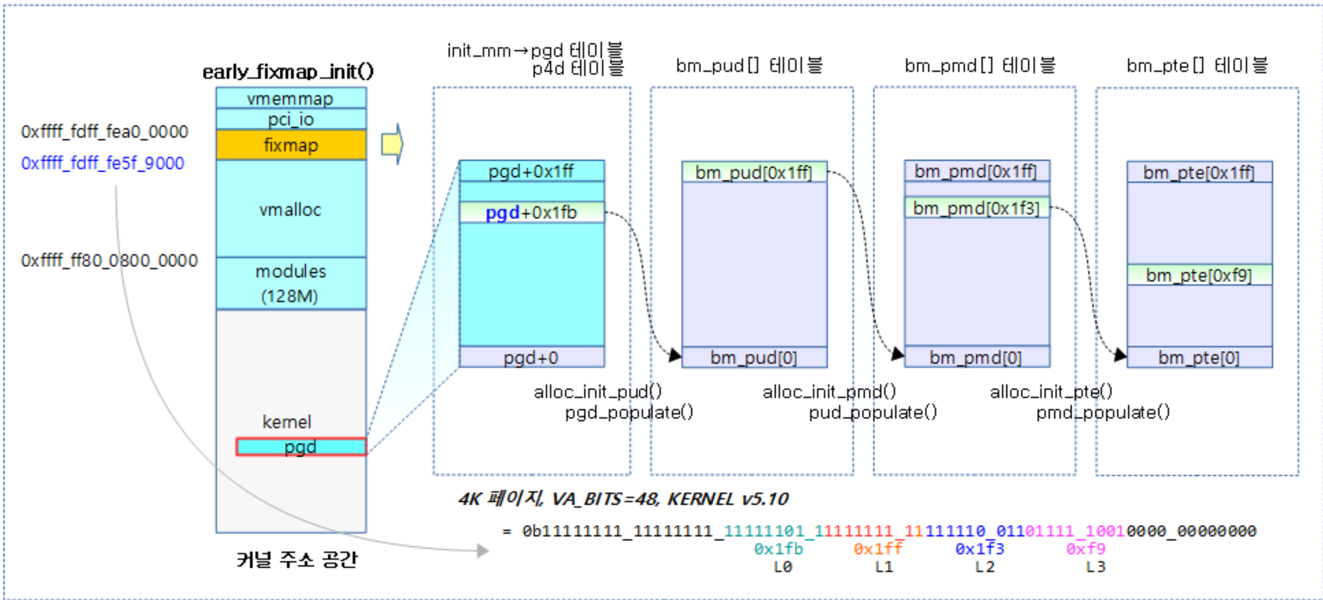
Learn why the early_fixmap_init() function uses the __p*d_populate() function instead of the p*d_populate() function.

- The p*d_populate() function takes the virtual address of the table to be linked as the third argument.
    - The p*d_populate() function uses the __pa() function internally to convert a virtual address for linear to a physical address. However, the __pa() function is still in the early stages of bootup, so the linear mapping has not been completed, so it cannot be used. Therefore, you cannot use kernel symbols such as bm_pud[] as virtual address arguments, so you cannot use this function.
- The __p*d_populate() function takes the physical address of the table to be linked as the second argument.
    - There is a separate function that converts kernel symbols into physical addresses. You can use the __pa_symbol() function to convert a kernel symbol virtual address to a physical

address. Use the __p*d_populate() function using this converted physical address.
  - consultation
    - arm64: mm: restrict virt_to_page() to the linear mapping (https://github.com/torvalds/linux/commit/9f2875912dac35d9272a82ea9eec9e5884b42cd2#diff-9236b527713c6c3c1d1ac8bb3ea6b737c7a566770a6580ae6b57e0708bc25e26) (2016, v4.7-rc1)
    - arm64: Use __pa_symbol for kernel symbols (https://github.com/torvalds/linux/commit/2077be6783b5936c3daa838d8addbb635667927f#diff-9236b527713c6c3c1d1ac8bb3ea6b737c7a566770a6580ae6b57e0708bc25e26) (2017, v4.11-rc1)
    - arm64: Add support for CONFIG_DEBUG_VIRTUAL (https://github.com/torvalds/linux/commit/ec6d06efb0bac6cd92846e42d1afe5b98b57e7c2#diff-9236b527713c6c3c1d1ac8bb3ea6b737c7a566770a6580ae6b57e0708bc25e26) (2017, v4.11-rc1)
    - mm: Introduce lm_alias (https://github.com/torvalds/linux/commit/568c5fe5a54f2654f5a4c599c45b8a62ed9a2013) (2017, v4.11-rc1)

The following illustration shows how the page table at each stage is activated for fixmap.

- Using 4K pages, VA_BITS=48, the page table from step 4 was used, and the p4d table after pgd uses pgd as it is on ARM64.



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

early_fixmap_init() – ARM32

arch/arm/mm/mmu.c

```
01 void __init early_fixmap_init(void)
02 {
03         pmd_t *pmd;
04
05         /*
06          * The early fixmap range spans multiple pmds, for which
07          * we are not prepared:
08          */
09         BUILD_BUG_ON((__fix_to_virt(__end_of_early_ioremap_region) >> PMD_SHIFT)
10                      != FIXADDR_TOP >> PMD_SHIFT);
11
12         pmd = fixmap_pmd(FIXADDR_TOP);
13         pmd_populate_kernel(&init_mm, pmd, bm_pte);
14
15         pte_offset_fixmap = pte_offset_early_fixmap;
16 }
```

Activate the fixmap area.

- In line 12 of the code, we get the pmd entry that corresponds to the top area of the fixmap virtual address.
- In line 13 of code, use the static bm_pte page prepared at compile time to connect and activate the PTE table to run the fixmap.
    - Each pte table covers 2M. The pte area is 4M, so we need two pte tables, but we only activate the top-level zone at first.

Key Fixmap APIs

set_fixmap()

include/asm-generic/fixmap.h

```
#define set_fixmap(idx, phys)                           \
        __set_fixmap(idx, phys, FIXMAP_PAGE_NORMAL)
```

Fixmap's request index @idx maps a page that corresponds to a physical address @phys, which is set to the normal kernel page mapping property.

- ARM32
    - L_PTE_YOUNG | L_PTE_PRESENT | L_PTE_XN | L_PTE_DIRTY | L_PTE_MT_WRITEBACK
- ARM64
    - FIXMAP_PAGE_NORMAL Use the -> PAGE_KERNEL -> __pgprot(PROT_NORMAL) attribute
        - PTE_TYPE_PAGE | PTE_AF | PTE_SHARED | PTE_MAYBE_NG | PTE_PXN | PTE_UXN | PTE_WRITE | PTE_ATTRINDX(MT_NORMAL)

clear_fixmap()

include/asm-generic/fixmap.h

```
#define clear_fixmap(idx)                       \
        __set_fixmap(idx, 0, FIXMAP_PAGE_CLEAR)
```

Unmap one page of physical addresses mapped to the @idx area of the request index of the fixmap, and set the attribute to CLEAR(1).

- PTE_DIRTY -> __pgprot(0)

set_fixmap_nocache()

include/asm-generic/fixmap.h

```
#define set_fixmap_nocache(idx, phys) \
        __set_fixmap(idx, phys, FIXMAP_PAGE_NOCACHE)
```

In the @idx area of the request index of the fixmap, it maps one page corresponding to the physical address @phys without a cache, and the attribute is set to FIXMAP_PAGE_NOCACHE.

- ARM32
  - L_PTE_YOUNG | L_PTE_PRESENT | L_PTE_XN | L_PTE_DIRTY | L_PTE_MT_DEV_SHARED | L_PTE_SHARE
- ARM64
  - FIXMAP_PAGE_NOCACHE Use the -> PAGE_KERNEL_NOCACHE -> __pgprot(PROT_NORMAL_NC) attribute
    - PTE_TYPE_PAGE | PTE_AF | PTE_SHARED | PTE_MAYBE_NG | PTE_PXN | PTE_UXN | PTE_WRITE | PTE_ATTRINDX(MT_NORMAL_NC)

set_fixmap_io()

include/asm-generic/fixmap.h

```
#define set_fixmap_io(idx, phys) \
        __set_fixmap(idx, phys, FIXMAP_PAGE_IO)
```

Map one page corresponding to the physical address @phys to the @idx area of the request index of the fixmap, and set the attribute to FIXMAP_PAGE_IO.

- ARM32
  - Use the same properties as nocache
- ARM64
  - FIXMAP_PAGE_IO -> PAGE_KERNEL_IO -> __pgprot (PROT_DEVICE_nGnRE) attribute
    - PTE_TYPE_PAGE | PTE_AF | PTE_SHARED | PTE_MAYBE_NG | PTE_PXN | PTE_UXN | PTE_WRITE | PTE_ATTRINDX(MT_DEVICE_nGnRE)

__set_fixmap() – ARM32

arch/arm/mm/mmu.c

```
/*
 * To avoid TLB flush broadcasts, this uses local_flush_tlb_kernel_range().
 * As a result, this can only be called with preemption disabled, as under
 * stop_machine().
```

```
 */
```

```
01 void __set_fixmap(enum fixed_addresses idx, phys_addr_t phys, pgprot_t prot)
02 {
03         unsigned long vaddr = __fix_to_virt(idx);
04         pte_t *pte = pte_offset_kernel(pmd_off_k(vaddr), vaddr);
05 
06         /* Make sure fixmap region does not exceed available allocation. */
07         BUILD_BUG_ON(FIXADDR_START + (__end_of_fixed_addresses * PAGE_SIZE) >
08                      FIXADDR_END);
09         BUG_ON(idx >= __end_of_fixed_addresses);
10 
11         if (pgprot_val(prot))
12                 set_pte_at(NULL, vaddr, pte,
13                         pfn_pte(phys >> PAGE_SHIFT, prot));
14         else
15                 pte_clear(NULL, vaddr, pte);
16         local_flush_tlb_kernel_range(vaddr, vaddr + PAGE_SIZE);
17 }
```

Map one page that corresponds to a physical address @phys to the fixmap index @idx number area with @prot attributes. You can also use this function for unmapping purposes, in which case you use the phys physical address and flags property as 1.

- In line 3 of code, we get the virtual address of the fixmap area that matches the index number @idx.
- In line 4 of the code, we find the entry pointer to the page table that corresponds to the virtual address.
- If the mapping is requested in line 11~13 of the code, use the flag to map the fixmap entry to the physical address @phys.
- If the unmapping is requested in line 14~15 of the code, clear the PTE entry corresponding to the fixmap slot index.
- On line 16 of the code, perform a TLB flush for that virtual address page area.

__set_fixmap() – ARM64

arch/arm64/mm/mmu.c

```
/*
 * Unusually, this is also called in IRQ context (ghes_iounmap_irq) so if we
 * ever need to use IPIs for TLB broadcasting, then we're in trouble here.
 */
```

```
void __set_fixmap(enum fixed_addresses idx,
                               phys_addr_t phys, pgprot_t flags)
{
        unsigned long addr = __fix_to_virt(idx);
        pte_t *ptep;

        BUG_ON(idx <= FIX_HOLE || idx >= __end_of_fixed_addresses);

        ptep = fixmap_pte(addr);

        if (pgprot_val(flags)) {
                set_pte(ptep, pfn_pte(phys >> PAGE_SHIFT, flags));
```

```
13         } else {
14                 pte_clear(&init_mm, addr, ptep);
15                 flush_tlb_kernel_range(addr, addr+PAGE_SIZE);
16         }
17 }
```

Map the physical address @phys to the @flags attribute to the fixmap virtual address space of the slot location corresponding to the @idx. You can also use this function for unmapping purposes, in which case you use the phys physical address and flags property as 0.

- In line 4 of code, we get the virtual address of the fixmap area that matches the index number @idx.
- In line 9 of the code, we find the entry pointer to the page table that corresponds to the virtual address.
- If the mapping is requested in line 11~12 of the code, use the flag to map the fixmap entry to the physical address @phys.
- If the unmapping is requested in code lines 13~16, TLB flush the corresponding 1 page area of the virtual address after clearing the PTE entry corresponding to the fixmap slot index.

fix_to_virt()

include/asm-generic/fixmap.h

```
/*
 * 'index to address' translation. If anyone tries to use the idx
 * directly without translation, we catch the bug with a NULL-deference
 * kernel oops. Illegal ranges of incoming indices are caught too.
 */
```

```
static __always_inline unsigned long fix_to_virt(const unsigned int idx)
{
        BUILD_BUG_ON(idx >= __end_of_fixed_addresses);
        return __fix_to_virt(idx);
}
```

Get the virtual address as an index for the fixmap area.

```
#define __fix_to_virt(x)        (FIXADDR_TOP - ((x) << PAGE_SHIFT))
```

virt_to_fix()

include/asm-generic/fixmap.h

```
static inline unsigned long virt_to_fix(const unsigned long vaddr)
{
        BUG_ON(vaddr >= FIXADDR_TOP || vaddr < FIXADDR_START);
        return __virt_to_fix(vaddr);
}
```

Returns the fixmap index that matches the fixmap area.

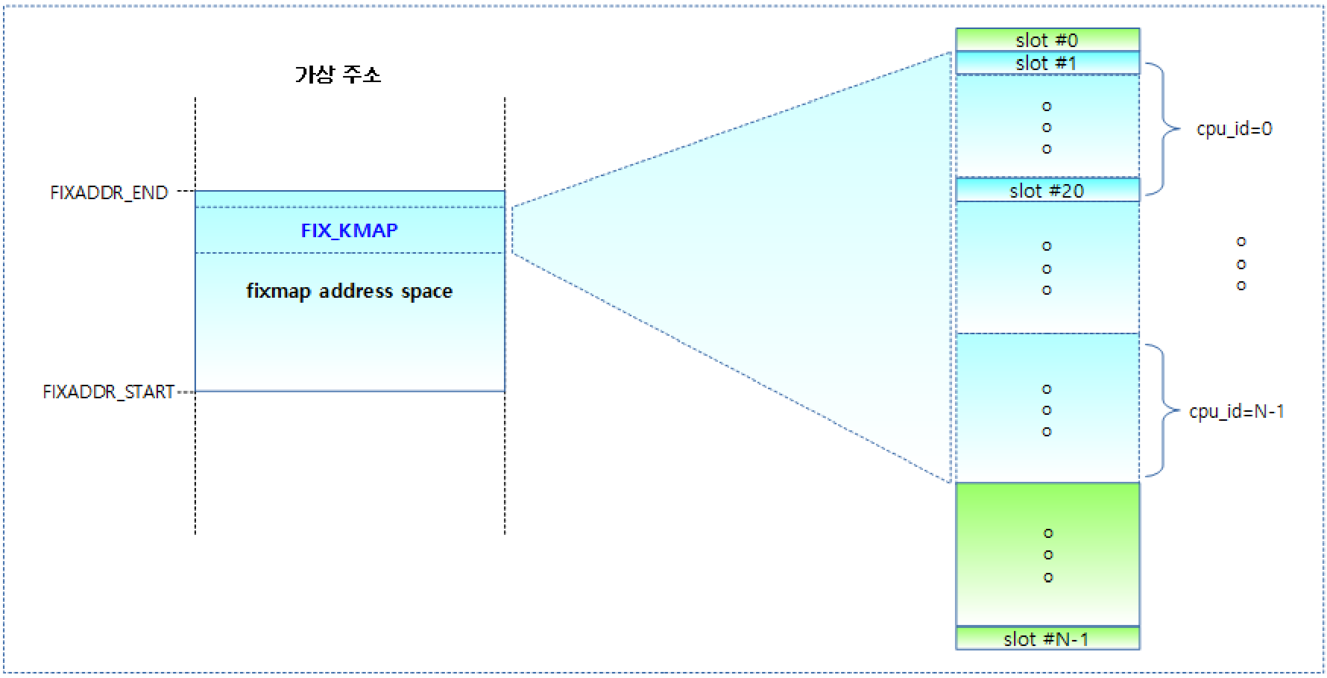
__virt_to_fix()

include/asm-generic/fixmap.h

```
#define __virt_to_fix(x)        ((FIXADDR_TOP - ((x)&PAGE_MASK)) >> PAGE
_SHIFT)
```

Highmem Page Mapping

The following figure shows the FIX_KMAP blocks used for highmem page mapping in the fixmap area.



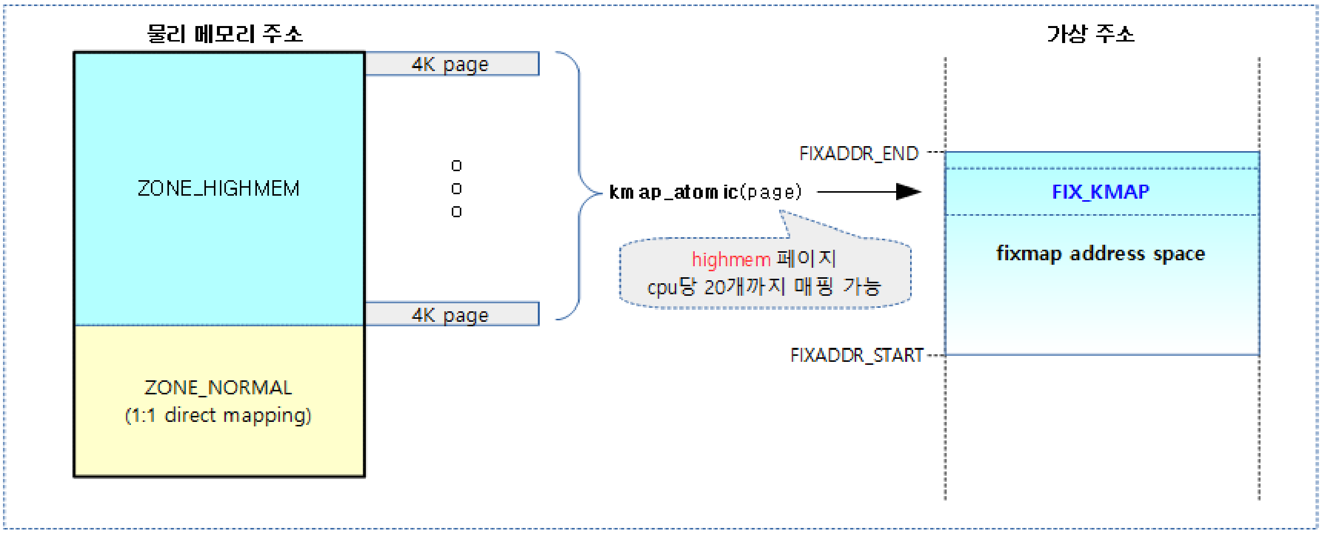
(http://jake.dothome.co.kr/wp-content/uploads/2016/03/fixmap-3c.png)

highmem mapping example)

Map the highmem page to the kmap block of the fixmap. It is used in a push/pull like a stack in the number of areas allocated for each CPU ID.

- kmap_atomic(page)

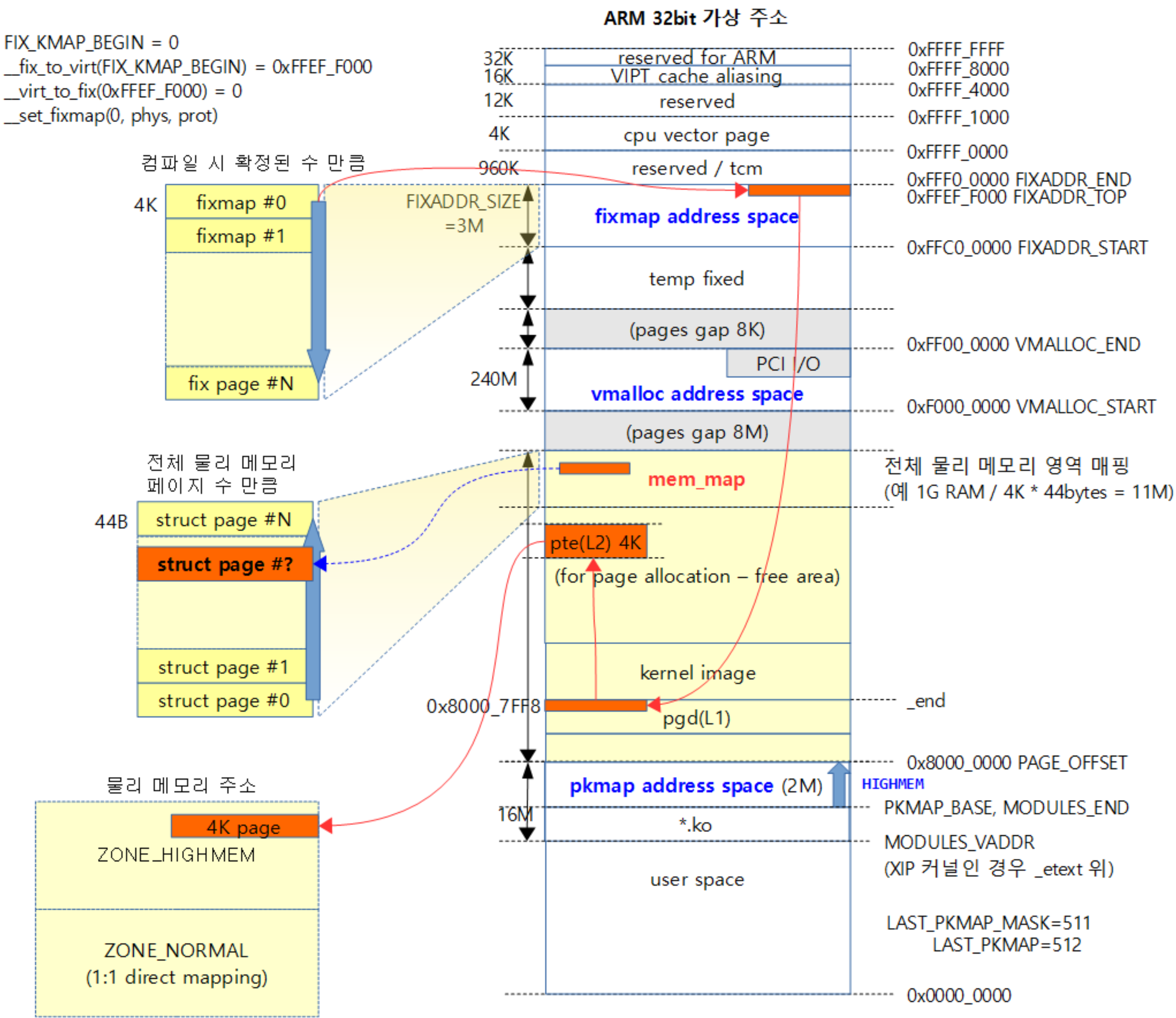
The following figure shows the mapping of a highmem physical memory page to a kmap block in the fixmap space.



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

The following figure shows the relationship between the page descriptor and the page table when mapping a highmem physical page to the fixmap area in ARM32.

- The page descriptors included in the mem_map, PGD and PTE page tables, etc., are all combined to create a complex look.



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

HighMem Page Allocation & Unallocate

kmap_atomic() – ARM32

arch/arm/mm/highmem.c

```
void *kmap_atomic(struct page *page)
{
        unsigned int idx;
        unsigned long vaddr;
        void *kmap;
        int type;

        preempt_disable();
        pagefault_disable();
        if (!PageHighMem(page))
                return page_address(page);

#ifdef CONFIG_DEBUG_HIGHMEM
        /*
         * There is no cache coherency issue when non VIVT, so force the
         * dedicated kmap usage for better debugging purposes in that case.
         */
        if (!cache_is_vivt())
                kmap = NULL;
        else
#endif
                kmap = kmap_high_get(page);
        if (kmap)
                return kmap;

        type = kmap_atomic_idx_push();

        idx = type + KM_TYPE_NR * smp_processor_id();
        vaddr = __fix_to_virt(idx);
#ifdef CONFIG_DEBUG_HIGHMEM
        /*
         * With debugging enabled, kunmap_atomic forces that entry to 0.
         * Make sure it was indeed properly unmapped.
         */
        BUG_ON(!pte_none(get_fixmap_pte(vaddr)));
#endif
        /*
         * When debugging is off, kunmap_atomic leaves the previous mapping
         * in place, so the contained TLB flush ensures the TLB is updated
         * with the new mapping.
         */
        set_fixmap_pte(idx, mk_pte(page, kmap_prot));

        return (void *)vaddr;
}
EXPORT_SYMBOL(kmap_atomic);
```

Map the highmem page to the fixmap area atomically. If it is already mapped to a kmap area, it returns the mapped virtual address.

- pagefault_disable()
  - increment the preemption counter to disable the preemption and perform barrier().
- if (! PageHighMem(page))

    - If the page address is not a highmem area, the
- return page_address(page);
    - If it is already mapped to kmap, it returns the virtual address corresponding to page.
- kmap = kmap_high_get(page);
    - Increment the pkmap reference counter and return the virtual address corresponding to the highmem page.
- if (kmap)
    - Returns if kmap has already been found.
- type = kmap_atomic_idx_push();
    - __kmap_atomic_idx increments and knows the previous value.
- idx = type + KM_TYPE_NR * smp_processor_id();
    - Type value + KM_TYPE_NR(20) * CPU ID
- vaddr = __fix_to_virt(idx);
    - FixMap retrieves the virtual address from the IDX number.
    - fixmap is idx 0 points to FIXADDR_TOP.
        - FIXADDR_TOP
            - 0xffef_f000= (FIXADDR_END(0xfff00000UL) – PAGE_SIZE)
    - Index numbers can be from 0 up to 0x2ff (767), and the mapped virtual address is 0xffc0_0000 ~ 0xffef_ffff for a total of 3M.
        - The actual allowable number of slot index numbers is limited to KM_TYPE_NR (ARM=20) per CPU.
- set_fixmap_pte(idx, mk_pte(page, kmap_prot));
    - mk_pte()
        - Combine the page address and the kmap_prot attribute values to create a PTE entry.
    - Map the PTE entry to the fixmap area that corresponds to the IDX number.

__fix_to_virt()

include/asm-generic/fixmap.h

```
#define __fix_to_virt(x)        (FIXADDR_TOP - ((x) << PAGE_SHIFT))
```

mk_pte() – ARM & ARM64

arch/arm64/include/asm/pgtable.h

```
#define mk_pte(page,prot)       pfn_pte(page_to_pfn(page), prot)
```

set_fixmap_pte() – ARM32

arch/arm/mm/highmem.c

```
static inline void set_fixmap_pte(int idx, pte_t pte)
{
        unsigned long vaddr = __fix_to_virt(idx);
        pte_t *ptep = pte_offset_kernel(pmd_off_k(vaddr), vaddr);

        set_pte_ext(ptep, pte, 0);
```

```
        local_flush_tlb_kernel_page(vaddr);
}
```

Map the PTE entry to the fixmap area that corresponds to the IDX number.

- unsigned long vaddr = __fix_to_virt(idx);
    - Find the virtual address of the fixmap area that corresponds to the idx number.
- pte_t *ptep = pte_offset_kernel(pmd_off_k(vaddr), vaddr);
    - pmd_off_k()
        - Get the PMD entry address value by the virtual address value.
    - pte_offset_kernel()
        - Use the pmd entry address value and the vaddr value to determine the pte entry address.
- set_pte_ext(ptep, pte, 0);
    - Store the PTE value in the PTE entry address.
        - rpi2: cpu_v7_set_pte_ext() call
- local_flush_tlb_kernel_page(vaddr);
    - Flush the TLB cache that corresponds to vaddr.

local_flush_tlb_kernel_page() – ARM32

arch/arm/include/asm/tlbflush.h

```
static inline void local_flush_tlb_kernel_page(unsigned long kaddr)
{
        const unsigned int __tlb_flag = __cpu_tlb_flags;

        kaddr &= PAGE_MASK;

        if (tlb_flag(TLB_WB))
                dsb(nshst);

        __local_flush_tlb_kernel_page(kaddr);
        tlb_op(TLB_V7_UIS_PAGE, "c8, c7, 1", kaddr);

        if (tlb_flag(TLB_BARRIER)) {
                dsb(nsh);
                isb();
        }
}
```

__local_flush_tlb_kernel_page() – ARM32

arch/arm/include/asm/tlbflush.h

```
static inline void __local_flush_tlb_kernel_page(unsigned long kaddr)
{
        const int zero = 0;
        const unsigned int __tlb_flag = __cpu_tlb_flags;

        tlb_op(TLB_V4_U_PAGE, "c8, c7, 1", kaddr);
        tlb_op(TLB_V4_D_PAGE, "c8, c6, 1", kaddr);
        tlb_op(TLB_V4_I_PAGE, "c8, c5, 1", kaddr);
        if (!tlb_flag(TLB_V4_I_PAGE) && tlb_flag(TLB_V4_I_FULL))
                asm("mcr p15, 0, %0, c8, c5, 0" : : "r" (zero) : "cc");

```

```
        tlb_op(TLB_V6_U_PAGE, "c8, c7, 1", kaddr);
        tlb_op(TLB_V6_D_PAGE, "c8, c6, 1", kaddr);
        tlb_op(TLB_V6_I_PAGE, "c8, c5, 1", kaddr);
}
```

tlb_op() – ARM32

arch/arm/include/asm/tlbflush.h

```
#define tlb_op(f, regs, arg)    __tlb_op(f, "p15, 0, %0, " regs, arg)
```

__tlb_op() – ARM32

arch/arm/include/asm/tlbflush.h

```
#define __tlb_op(f, insnarg, arg) \
        do { \
                if (always_tlb_flags & (f)) \
                        asm("mcr " insnarg \
                            : : "r" (arg) : "cc"); \
                else if (possible_tlb_flags & (f)) \
                        asm("tst %1, %2\n\t" \
                            "mcrne " insnarg \
                            : : "r" (arg), "r" (__tlb_flag), "Ir" (f) \
                            : "cc"); \
        } while (0)
```

kmap_atomic_idx_push() – 32bit

include/linux/highmem.h

```
static inline int kmap_atomic_idx_push(void)
{
        int idx = __this_cpu_inc_return(__kmap_atomic_idx) - 1;

#ifdef CONFIG_DEBUG_HIGHMEM
        WARN_ON_ONCE(in_irq() && !irqs_disabled());
        BUG_ON(idx >= KM_TYPE_NR);
#endif
        return idx;
}
```

Returns the slot index for the fixmap to be used by that CPU, and increments its value.

- If you use the CONFIG_DEBUG_HIGHMEM option, if the fixmap index slot exceeds KM_TYPE_NR (ARM=20), it will output a message about the bug and stop.
- int idx = __this_cpu_inc_return(__kmap_atomic_idx);

- __kmap_atomic_idx Increase per-CPU data by 1
- IDX returns the value before the increase.

__this_cpu_inc_return()

```
#define __this_cpu_inc_return(pcp)      __this_cpu_add_return(pcp, 1)
```

- Returns pcp plus 1

__this_cpu_add_return()

```
#define __this_cpu_add_return(pcp, val)
\
({
\
        __this_cpu_preempt_check("add_return");
\
        raw_cpu_add_return(pcp, val);
\
})
```

- If the preemption is not disable, or the irq is not disable, the stack dump
- __kmap_atomic_idx Increase per-CPU data by 1

raw_cpu_add_return()

```
#define raw_cpu_add_return(pcp, val)    __pcpu_size_call_return2(raw_cpu
_add_return_, pcp, val)
```

- Calling a macro function to classify an optimized addition function based on the size of the scalar data type PCP

__pcpu_size_call_return2()

```
#define __pcpu_size_call_return2(stem, variable, ...)
\
({
\
        typeof(variable) pscr2_ret__;
\
        __verify_pcpu_ptr(&(variable));
\
        switch(sizeof(variable)) {
\
        case 1: pscr2_ret__ = stem##1(variable, __VA_ARGS__); break;
\
        case 2: pscr2_ret__ = stem##2(variable, __VA_ARGS__); break;
\
        case 4: pscr2_ret__ = stem##4(variable, __VA_ARGS__); break;
\
        case 8: pscr2_ret__ = stem##8(variable, __VA_ARGS__); break;
\
        default:
\
                __bad_size_call_parameter(); break;
\
```

```
        }                                                       \
        pscr2_ret__;                                            \
})
```

- Depending on the size of the variable, it is called with the number stem (function name argument) + 1/2/4/8

this_cpu_add_return_4()

```
#define this_cpu_add_return_4(pcp, val) this_cpu_generic_add_return(pcp, val)
```

- The ARM architecture calls generic code regardless of size.

this_cpu_generic_add_return()

```
#define this_cpu_generic_add_return(pcp, val)                   \
({                                                              \
        typeof(pcp) __ret;                                      \
        unsigned long __flags;                                  \
        raw_local_irq_save(__flags);                            \
        raw_cpu_add(pcp, val);                                  \
        __ret = raw_cpu_read(pcp);                              \
        raw_local_irq_restore(__flags);                         \
        __ret;                                                  \
})
```

- Add the val value to the per-cpu variable and read it back and return it.

raw_cpu_add()

```
#define raw_cpu_add(pcp, val)           __pcpu_size_call(raw_cpu_add_, pcp, val)
```

- Calling a macro function to classify an optimized addition function based on the size of the scalar data type PCP

__pcpu_size_call()

```
#define __pcpu_size_call(stem, variable, ...)                   \
do {                                                            \
        __verify_pcpu_ptr(&(variable));                         \
        switch(sizeof(variable)) {                              \
```

```
                case 1: stem##1(variable, __VA_ARGS__);break;
\
                case 2: stem##2(variable, __VA_ARGS__);break;
\
                case 4: stem##4(variable, __VA_ARGS__);break;
\
                case 8: stem##8(variable, __VA_ARGS__);break;
\
                default:
\
                        __bad_size_call_parameter();break;
\
        }
\
} while (0)
```

- Depending on the size of the variable, it is called with the number stem (function name argument) + 1/2/4/8

raw_cpu_add_4()

```
#define raw_cpu_add_4(pcp, val)         raw_cpu_generic_to_op(pcp, val, +=)
```

- The ARM architecture calls generic code regardless of size.

raw_cpu_generic_to_op()

```
#define raw_cpu_generic_to_op(pcp, val, op)
\
do {
\
        *raw_cpu_ptr(&(pcp)) op val;
\
} while (0)
```

- e.g. pcp=__kmap_atomic_idx, val=1, op= +=
  - __kmap_atomic_idx += 1, which is per-cpu int data

kunmap_atomic() – 32bit

include/linux/highmem.h

```
/*
 * Prevent people trying to call kunmap_atomic() as if it were kunmap()
 * kunmap_atomic() should get the return value of kmap_atomic, not the page.
 */
#define kunmap_atomic(addr)                                     \
do {                                                            \
        BUILD_BUG_ON(__same_type((addr), struct page *));       \
        __kunmap_atomic(addr);                                  \
} while (0)
```

Use kmap_atomic() to release the highmem area that is mapped to the fixmap area.

__kunmap_atomic() – 32bit

arch/arm/mm/highmem.c

```
void __kunmap_atomic(void *kvaddr)
{
        unsigned long vaddr = (unsigned long) kvaddr & PAGE_MASK;
        int idx, type;

        if (kvaddr >= (void *)FIXADDR_START) {
                type = kmap_atomic_idx();
                idx = type + KM_TYPE_NR * smp_processor_id();

                if (cache_is_vivt())
                        __cpuc_flush_dcache_area((void *)vaddr, PAGE_SIZE);
#ifdef CONFIG_DEBUG_HIGHMEM
                BUG_ON(vaddr != __fix_to_virt(idx));
                set_fixmap_pte(idx, __pte(0));
#else
                (void) idx;  /* to kill a warning */
#endif
                kmap_atomic_idx_pop();
        } else if (vaddr >= PKMAP_ADDR(0) && vaddr < PKMAP_ADDR(LAST_PKMAP)) {
                /* this address was obtained through kmap_high_get() */
                kunmap_high(pte_page(pkmap_page_table[PKMAP_NR(vaddr)]));
        }
        pagefault_enable();
}
EXPORT_SYMBOL(__kunmap_atomic);
```

Use kmap_atomic() to release the highmem areas that are mapped to the fixmap or kmap areas.

- if (kvaddr >= (void *)FIXADDR_START) {
  - If the address is a fixmap zone
- type = kmap_atomic_idx();
  - __kmap_atomic_idx Per-CPU data value minus 1 index value
- idx = type + KM_TYPE_NR * smp_processor_id();
  - KM_TYPE_NR=20
- if (cache_is_vivt())
  - If the L1 d-cache type is VIVT
  - rpi2:
    - L1 d-cache is CACHEID_VIPT_NONALIASING
    - L1 i-cache is CACHEID_VIPT_I_ALIASING
- __cpuc_flush_dcache_area((void *)vaddr, PAGE_SIZE);
  - flush the d-cache of the page 1 area of that address.
- kmap_atomic_idx_pop();
  - __kmap_atomic_idx Reduces per-CPU data by 1.
- } else if (vaddr >= PKMAP_ADDR(0) && vaddr < PKMAP_ADDR(LAST_PKMAP)) {
  - If the virtual address is a pkmap mapping area
- kunmap_high(pte_page(pkmap_page_table[PKMAP_NR(vaddr)]));
  - Unmap the highmem page address mapped to the kmap area.
- pagefault_enable();
  - Enable preemption.

consultation

- page_address_init() (http://jake.dothome.co.kr/page_address_init) | Qc
- Kmap(Pkmap) (http://jake.dothome.co.kr/kmap) | Qc
- Early ioremap (http://jake.dothome.co.kr/early-ioremap) | Qc
- Fixing kmap_atomic() (https://lwn.net/Articles/356378/) | LWN.net

---

13 thoughts to “Fixmap”

JAE-KOOK CHOI
2019-07-21 17:17 (http://jake.dothome.co.kr/fixmap/#comment-215352)

I have a question.
The following illustration shows how the page table at each stage is activated for fixmap.
If you use 4K pages, VA_BITS=39, you use the page table from step 3, so the intermediate pud table is omitted.

If the L3 area (10 bits, maximum number of 1024) is 111111_1111, it should be bm_pte_[0x3ff], but the bm_pte[] table size is up to 512.
If so, it means that there are areas where the fixmap area is not mapped, or that the picture is wrong. I'm not sure what's right.

RESPONSE (/FIXMAP/?REPLYTOCOM=215352#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2019-07-22 08:58 (http://jake.dothome.co.kr/fixmap/#comment-215358)

Hello?

You have a good ability to concentrate. Incorrect illustrations have been corrected.
L1, L2, and L3 are all 9 bits each.

Note that the size of the index is determined by the size of the page.
– There are 4 indexes available for 512K pages, using 9 bits.
– There are 16 indexes available on 2048K pages, using 11 bits.
– There are 64 indexes available on 8192K pages, using 13 bits.

I appreciate it.

RESPONSE (/FIXMAP/?REPLYTOCOM=215358#RESPOND)

JAE-KOOK CHOI
2019-08-12 17:42 (http://jake.dothome.co.kr/fixmap/#comment-215877)

I have a question.
The fixmap area is supposed to be the space where the virtual address is determined at compile time, so
why don't other mapping subsystems set the mapping area at compile time and use fixmap to map it?

And when it says "Temporarily map to this space for use before officially initializing the console device", does that mean that the console device is the device driver??
If that's true, as I understand it, it's developed as a module before the device driver is officially added, so does that mapping area become the fixmap area?

RESPONSE (/FIXMAP/?REPLYTOCOM=215877#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2019-08-12 21:33 (http://jake.dothome.co.kr/fixmap/#comment-215879)

vmap mapping to vmalloc area, or normal mapping to userspace, use the dynamic mapping method.
In other words, when allocating memory, there is a way to allocate it statically at compile time, similar to allocating it at runtime using the malloc() API.

Even when mapping to the vmalloc space normally, modern kernels use some areas of the fixmap, which are used to atomically reflect pgd->pud->pmd->pte->page at once.

When we say console device, we literally mean a console device. The code that works in response to the device is called a device driver. Device drivers programmed in such a way that they are embedded in a module or kernel are mapped using the ioremap() API, which is one of the regular mappings. When it is activated, it does not work until the device drivers are up and running long after the kernel boots up and the kernel core is running. For this reason, there are ways to start up the console device specifically for debug output before the device driver is officially running. These are called early consoles. In order to use the Early Console used for this purpose, we will use the early_ioremap() API, which uses the fixmap area instead of ioremap(), which is one of the regular mappings, to get the device up and running.

즉 다시 말씀드리면 콘솔 디바이스를 포함한 일반 디바이스 드라이버들은 vmalloc 영역을 사용하는 ioremap() API를 사용하고, early 콘솔 출력을 위해 fixmap을 사용하는 early_ioremap() API를 사용합니다.

감사합니다.

응답 (/FIXMAP/?REPLYTOCOM=215879#RESPOND)

최재국
2019-08-13 23:43 (http://jake.dothome.co.kr/fixmap/#comment-215909)

질문있습니다 ..ㅠㅠ
early_fixmap_init() – ARM64에서
pgd가 init_mm.pgd이여야 되는것 아닌가요??
함수초기에 pgd_t *pgdp, pgd;로 선언 되어져 있는데 init_mm.pgd가 아니고 지역변수 pgd로 매핑했을때 해당 함수가 끝나면 결국 의미없는것 아닌가요??

항상 답변해주셔서 감사합니다.

응답 (/FIXMAP/?REPLYTOCOM=215909#RESPOND)

최재국
2019-08-14 17:44 (http://jake.dothome.co.kr/fixmap/#comment-215927)

아 제가 잘못생각하고 있었습니다. 감사합니다.

응답 (/FIXMAP/?REPLYTOCOM=215927#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2019-08-15 15:37 (http://jake.dothome.co.kr/fixmap/#comment-215942)

금방 확인하셨군요.
오늘도 즐거운 하루되세요.

응답 (/FIXMAP/?REPLYTOCOM=215942#RESPOND)

이대로
2019-12-27 21:02 (http://jake.dothome.co.kr/fixmap/#comment-227370)

질문 있습니다!!

fixed_address – ARM64
에서 그려주신 그림에 보면
FIXADDR_START가 FIX_PGD를 포함하고 있는데...

FIXADDR_START가 정의되어 있는 부분을 보면

```
__end_of_permanent_fixed_addresses, <————————————–
/*
* Temporary boot-time mappings, used by early_ioremap(),
* before ioremap() is functional.
*/
#define NR_FIX_BTMAPS (SZ_256K / PAGE_SIZE)
#define FIX_BTMAPS_SLOTS 7
#define TOTAL_FIX_BTMAPS (NR_FIX_BTMAPS * FIX_BTMAPS_SLOTS)
FIX_BTMAP_END = __end_of_permanent_fixed_addresses,
FIX_BTMAP_BEGIN = FIX_BTMAP_END + TOTAL_FIX_BTMAPS – 1,
/*
* Used for kernel page table creation, so unmapped memory may be used
* for tables.
*/
FIX_PTE,
FIX_PMD,
FIX_PUD,
FIX_PGD,
__end_of_fixed_addresses
};
#define FIXADDR_SIZE (__end_of_permanent_fixed_addresses << PAGE_SHIFT)
#define FIXADDR_START (FIXADDR_TOP – FIXADDR_SIZE)
```

FIXADDR_START는 fixed_addresses enum 구조체에서 FIX_BTMAP, FIX_PTE, FIX_PMD, FIX_PUD, FIX_PGD를 제외한
부분인 __end_of_permanent_fixed_addresses를 가리키고 있는 것이 아닌가요?

응답 (/FIXMAP/?REPLYTOCOM=227370#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2019-12-30 18:54 (http://jake.dothome.co.kr/fixmap/#comment-227416)

안녕하세요?

결론을 말씀드리면, 이대로님이 생각하시는대로 FIXADDR_START ~ FIXADDR_TOP 영역에 BTMAP과 4개의 페이지테이블이 포함되지 않습니다. (그림을 수정해야겠습니다. 감사 ^^)

조금 더 설명을 드리자면,

고정 주소 엔트리들 중 (A) 영역만 fixmap 영역으로 인정하고,
(B)영역에 위치한 early_ioremap 영역과, 4개의 페이지 테이블 영역을
fixmap 영역에서 제외시켰습니다.

arch/arm64/include/asm/fixmap.h

enum fixed_addresses {
(A)
__end_of_permanent_fixed_addresses,
(B)
__end_of_fixed_addresses,
}

단 fixmap 영역이 아니라고 해도 논리적으로만 범위를 벗어난 것이고,
실제 내부 동작은 모두 동일하게 다음과 같은 fixmap API들을 사용합니다.
– __fix_to_virt()
– __virt_to_fix()
– fixmap_pte()
– set_fixmap_offset()
– __set_fixmap()
– …

새해엔 더 행복하시길 바랍니다.

감사합니다.

응답 (/FIXMAP/?REPLYTOCOM=227416#RESPOND)

이대로
2020-01-07 00:47 (http://jake.dothome.co.kr/fixmap/#comment-227887)

안녕하세요!

답변 주신 것을 바탕으로 다시 코드를 보다가 궁금한 점이 생겨 질문 올립니다.
(5.1버전)
'early_fixmap_init() – ARM64'에서
'참고로 early_fixmap_init() 함수에서는 전체 fixmap 영역 중 FDT를 제외한 아랫 부분 2M를 먼저 커버하기 위해 매핑합니다.'
부분에서 말씀하신 2M에 속하는 것은 __end_of_permanent_fixed_addresses부터
FIX_EARLYCON_MEM_BASE 부분까지 인 것인지요?

early_fixmap_init()에서는 FIX_PTE, FIX_PMD, FIX_PUD, FIX_PGD 부분을 사용하는 것이 없다고 생각되는데..
혹시 제 생각이 틀린 것인지 궁금합니다.

항상 친절하게 답변을 달아주셔서 감사합니다
새해 복 많이 받으세요~

응답 (/FIXMAP/?REPLYTOCOM=227887#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2020-01-07 08:40 (http://jake.dothome.co.kr/fixmap/#comment-227939)

안녕하세요?

정확히 __end_of_permanent_fixed_addresses 위치의 2M를 먼저 커버하는데,
다음 조건과 같이 그 2M에 FIX_BTMAP이 포함됩니다.

BUILD_BUG_ON((__fix_to_virt(FIX_BTMAP_BEGIN) >> PMD_SHIFT)
!= (__fix_to_virt(FIX_BTMAP_END) >> PMD_SHIFT))

early_fixmap_init()에서 FIX_PTE, FIX_PMD, FIX_PUD, FIX_PGD 부분은
명확히 포함시키진 않았습니다만 보통 2M 범위에 포함됩니다.

감사합니다. 새해 복 많이 받으세요.

응답 (/FIXMAP/?REPLYTOCOM=227939#RESPOND)

PARAN LEE (HTTP://WWW.BHRAL.COM)
2023-04-23 23:36 (http://jake.dothome.co.kr/fixmap/#comment-307918)

문영일님, 안녕하세요! 이파란입니다.
레이아웃이 또 변경되었네요 ^^;

Arm 64 가 가상 메모리 레이아웃이 6.0 부터 변경된 패치입니다.
arm64/bpf: Remove 128MB limit for BPF JIT programs
–
https://github.com/torvalds/linux/commit/b89ddf4cca43f1269093942cf5c4e457fd45c335
(https://github.com/torvalds/linux/commit/b89ddf4cca43f1269093942cf5c4e457fd45c335)

Documentation/arm64: update memory layout table.
–
https://github.com/torvalds/linux/commit/5bed6a93920dea85b1828cbe3aa3e333ed663ea3
(https://github.com/torvalds/linux/commit/5bed6a93920dea85b1828cbe3aa3e333ed663ea3)

위 패치로 아래 독스에 반영되어있네요.
– https://www.kernel.org/doc/html/latest/arm64/memory.html
(https://www.kernel.org/doc/html/latest/arm64/memory.html)

감사합니다~
이파란 드림.

응답 (/FIXMAP/?REPLYTOCOM=307918#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2023-04-24 14:46 (http://jake.dothome.co.kr/fixmap/#comment-307919)

다행히 소소한 변경이군요. ^^
언제나 감사를 드립니다.

응답 (/FIXMAP/?REPLYTOCOM=307919#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

< Kmap(Pkmap) (http://jake.dothome.co.kr/kmap/)

Memory Model -2- (mem_map) > (http://jake.dothome.co.kr/mem_map/)

문c 블로그 (2015 ~ 2023)