Memblock – (1)

2016-01-26 (http://jake.dothome.co.kr/memblock-1/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.10>

Memblock

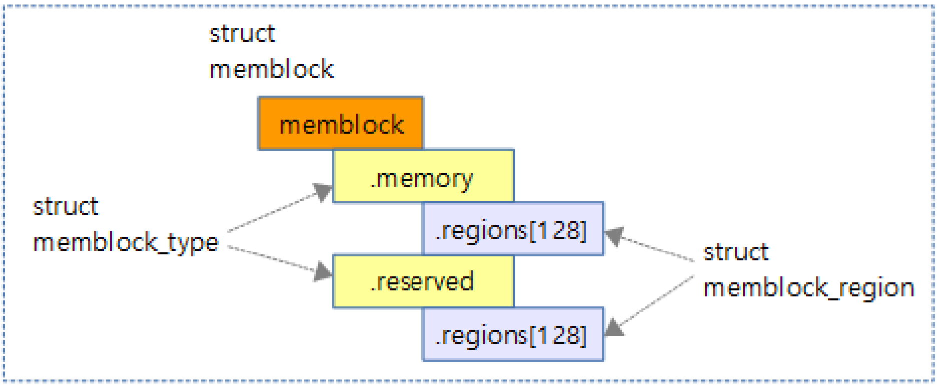
The memblock memory allocator is the first memory allocator that is activated at kernel bootup time, registering and using a range of memory before other kernel memory allocators are ready. It is mainly used at boot time, but it is also used at runtime if the memory hotplug feature is enabled. Prior to the introduction of memblock in kernel v2010.2.6 in 35, it used a memory allocator called bootmem. While bootmem only manages a fraction of the memory required for bootup (lowmem), memblock manages the entire memory. Memblock is the only memory manager that can allocate memory early until the kernel's buddy system is ready to use as a page allocator. Therefore, we use the term early memory allocation. In addition, memblocks are also known as LMBs (Logical Memory Blocks). It also supports hot-plug memory and is used to add memory added at runtime using memblock and then switch to a buddy allocator.

Structure of Memblock

Memblock is divided into two types as follows

- Memory Type
  - The memory type registers and uses the area of physical memory to be used. It can be registered because it is restricted by kernel parameters to use only a small area of the actual physical memory. Up to 128 regions can be used in the regions[ ] array initially, and can then continue to expand in double-fold increments.
- reserved type
  - The reserved type registers and uses the area of physical memory that is in use or will be used. Up to 128 regions can be used in the regions[ ] array initially, and can then continue to expand in double-fold increments.

Initially, the memblock area is registered with the following type and area arrangement:



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock1a.png)

Physmem type

The physmem type was added in 2014 and is used by registering a physically detected memory region, and this value is not modified after it has been registered. Unlike the memory type, the code was added with the plan to register and use the actual physical memory size rather than the memory limited by the kernel parameters, and it is currently used in the S390 architecture. regions[ ] array, initially using up to 4 entries.

CONFIG_HAVE_MEMBLOCK_PHYS_MAP

- Options added in kernel 2014.1-rc3 in January 16
- You can add up to 4 (INIT_PHYSMEM_REGIONS) areas of available physical memory, and the input area will not be modified.
- Unlike other memblock structures, it contains the entire range of memory.

CONFIG_ARCH_KEEP_MEMBLOCK

- This is an option that can be controlled in vmlinux.lds.h so that data registered in memblock can be preserved without removing it from memory after the initial boot process.

include/linux/memblock.h

```
#ifndef CONFIG_ARCH_KEEP_MEMBLOCK
#define __init_memblock __meminit
#define __initdata_memblock __meminitdata
void memblock_discard(void);
#else
#define __init_memblock
#define __initdata_memblock
static inline void memblock_discard(void) {}
#endif
```

include/linux/init.h

```
#define __meminit        __section(.meminit.text) __cold notrace \
                                                  __latent_entropy
#define __meminitdata    __section(.meminit.data)
```

CONFIG_MEMORY_HOTPLUG

- CONFIG_ARCH_KEEP_MEMBLOCK Use it in conjunction with the options.
  - If you don't use the CONFIG_MEMORY_HOTPLUG option, you can delete the memblock area after the boot process ends. In such cases, memblcok should not be used after boot.
  - If you are using CONFIG_MEMORY_HOTPLUG, it will be preserved so that it can continue to be used even when the boot process ends.

include/asm-generic/vmlinux.lds.h

```
#if defined(CONFIG_MEMORY_HOTPLUG)
#define MEM_KEEP(sec)    *(.mem##sec)
#define MEM_DISCARD(sec)
#else
#define MEM_KEEP(sec)
#define MEM_DISCARD(sec) *(.mem##sec)
#endif
```

Initialize

The global structure name memblock is initialized at compile time as shown below.

- The cnt variable is set to an initial 1 with the number of each zone.
  - Even if there is no zone data, the default is set to 1, and if you add one for the first time, the CNT will not change. After that, each additional increments by 1.
- bottom_up variable is false by default in arm and arm64, so when requesting an allocation, it searches from the top to the bottom to find the free space.
- The current_limit is initially set to MEMBLOCK_ALLOC_ANYWHERE (~(phys_addr_t) 0) and the values that can be set are:
  - MEMBLOCK_ALLOC_ANYWHERE (~(phys_addr_t) 0):
    - Maximum of physical addresses
  - MEMBLOCK_ALLOC_ACCESSBLE (0):
    - Maximum number of physical memory addresses
  - Limit the maximum limit to the address value you enter

memblock_region Arrays

mm/memblock.c

```
static struct memblock_region memblock_memory_init_regions[INIT_MEMBLOCK_REGIONS] __initdata_memblocck;
static struct memblock_region memblock_reserved_init_regions[INIT_MEMBLOCK_RESERVED_REGIONS] __initddata_memblock;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
static struct memblock_region memblock_physmem_init_regions[INIT_PHYSMEM_REGIONS]
```

```
k;
#endif
```

Specifies the number of entry arrays to use in the memblock area at compile time. The arrays are 128, 128, and 4 in order from top to bottom. Physical memory registrations are limited to a maximum of four, as they are not registered a few times.

memblock array

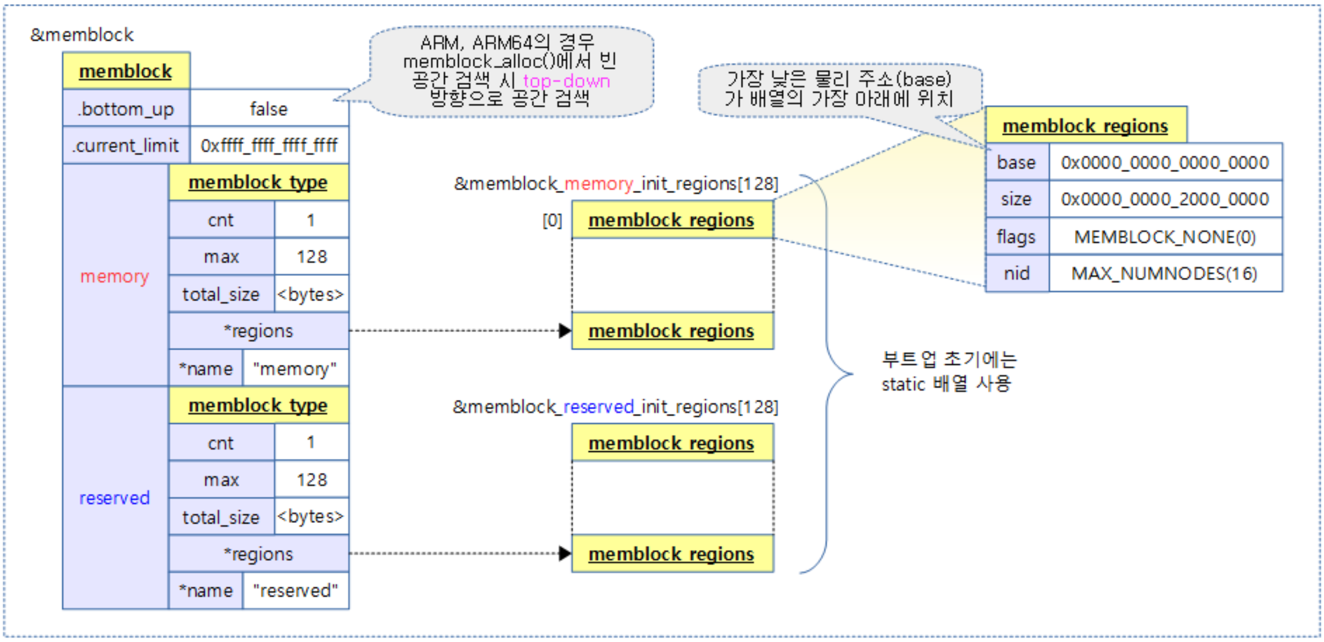
mm/memblock.c

```
01 struct memblock memblock __initdata_memblock = {
02         .memory.regions         = memblock_memory_init_regions,
03         .memory.cnt             = 1,    /* empty dummy entry */
04         .memory.max             = INIT_MEMBLOCK_REGIONS,
05         .memory.name            = "memory",
06
07         .reserved.regions       = memblock_reserved_init_regions,
08         .reserved.cnt           = 1,    /* empty dummy entry */
09         .reserved.max           = INIT_MEMBLOCK_REGIONS,
10         .reserved.name          = "reserved",
11
12         .bottom_up              = false,
13         .current_limit          = MEMBLOCK_ALLOC_ANYWHERE,
14 };
```

At compile time, prepare a memblock that manages 3 types of memblock areas. In the initial bootup, each memblock area points to a static array, which can then be added dynamically as it needs to be expanded.

- In lines 2~5 of the code, initialize the memory memblock with an array of 128 entries.
- In lines 7~10 of code, initialize reserved memblock to an array of 128 entries. The size of this array can be expanded in double-fold increments in the future when the reserved area is registered and filled.
- Set the initial value of line 12 to perform a search of the empty area in the direction of the address from top to bottom.
- In line 13 of the code, the maximum memory allocation limit value is initialized to be the largest address of the address used by the system

The following figure shows the association between memblock, memblock_type, and memblock_regions structures.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock16c.png)

Main Structure

struct memblock

include/linux/memblock.h

```
struct memblock {
        bool bottom_up;      /* is bottom up direction? */
        phys_addr_t current_limit;
        struct memblock_type memory;
        struct memblock_type reserved;
};
```

- bottom_up
  - This option allows memory allocation to be searched from bottom to top to allocate free space, and is currently applied to x86_64 NUMA systems first. This works if the memory hot-plug feature is used on a NUMA system. This feature is designed to induce the kernel to allocate memory as close to the kernel memory address as possible, so that when a node's memory is turned off from the system, the migration rate can be suppressed as much as possible if there are only a few pages already allocated and used by that node (1 = allocation from bottom to top).
- current_limit
  - This is used when you want to limit the allocation of kernel memory in the kernel bootup process routine. On a 64-bit system, all physical memory is mapped 1:1 directly to virtual memory, so it is not managed separately as a lowmem area, so the current_limit is specified by the MEMBLOCK_ALLOC_ANYWHERE (0xffffffff_ffffffff) value used at the time of the initial build of the kernel.
- memory
  - Physical Memory Area Registration
- reserved
  - reserved zone registration

struct memblock_type

include/linux/memblock.h

```
struct memblock_type {
        unsigned long cnt;
        unsigned long max;
        phys_addr_t total_size;
        struct memblock_region *regions;
        char *name;
};
```

- cnt
  - Use area entry chatter. It is designed so that even if no entry area is registered, it will start from 1. This value does not change when the first entry area is added, but increases with each second entry.
- max
  - Maximum number of available zone entries
- total_size
  - Plus the size of all registered zones of that memblock type, in bytes
- *regions
  - Pointer to an area
- *name
  - Area Name

struct memblock_region

include/linux/memblock.h

```
struct memblock_region {
        phys_addr_t base;
        phys_addr_t size;
        enum memblock_flags flags;
#ifdef CONFIG_NEED_MULTIPLE_NODES
        int nid;
#endif
};
```

- base
  - Starting Physical Address
- size
  - Area size
- flags
  - flags can use four macro constants and a combination of three bit requests:
    - MEMBLOCK_NONE (0x0): No outlier requests
    - MEMBLOCK_HOTPLUG (0x1): Memory hotplug zone
    - MEMBLOCK_MIRROR (0x2): Mirrored area
    - MEMBLOCK_NOMAP (0x4): Areas that the kernel does not directly map
- nid
  - Node ID

Memblock's Native Management API

When adding memory, if you are not a multi-node (NUMA, etc.) system, you can use the memblock_add( ) function, and internally the node id will be treated as 0. And if you are using a multi-node system, use the memblock_add_node( ) function to specify the node ID. When the machine is designed, the registered memory regions are not supposed to overlap with each other, but in order to handle various exceptions, it is necessary to adjust the requested areas so that they do not overlap in various situations.

Adding a Memory Area

memblock_add()

mm/memblock.c

```
/**
 * memblock_add - add new memblock region
 * @base: base address of the new region
 * @size: size of the new region
 *
 * Add new memblock region [@base, @base + @size) to the "memory"
 * type. See memblock_add_range() description for mode details
 *
 * Return:
 * 0 on success, -errno on failure.
 */
```

```
int __init_memblock memblock_add(phys_addr_t base, phys_addr_t size)
{
        phys_addr_t end = base + size - 1;

        memblock_dbg("memblock_add: [%pa-%pa] %pF\n",
                     &base, &end, (void *)_RET_IP_);

        return memblock_add_range(&memblock.memory, base, size, MAX_NUMNODES, 0);
}
```

Add a memory area to the memory memblock.

- Add @size from the physical memory start address @base to the memory memblock.

Add a reserve area

memblock_reserve()

mm/memblock.c

```
int __init_memblock memblock_reserve(phys_addr_t base, phys_addr_t size)
{
```

```
        phys_addr_t end = base + size - 1;

        memblock_dbg("%s: [%pa-%pa] %pS\n", __func__,
                     &base, &end, (void *)_RET_IP_);

        return memblock_add_range(&memblock.reserved, base, size, MAX_NUMNODES, 0);
}
```

reserved area to reserved memblock.

- Add @size from the physical memory start address @base to the reserved memblock.

Add a zone

memblock_add_range()

mm/memblock.c

```
/**
 * memblock_add_range - add new memblock region
 * @type: memblock type to add new region into
 * @base: base address of the new region
 * @size: size of the new region
 * @nid: nid of the new region
 * @flags: flags of the new region
 *
 * Add new memblock region [@base,@base+@size) into @type.  The new region
 * is allowed to overlap with existing ones - overlaps don't affect already
 * existing regions.  @type is guaranteed to be minimal (all neighbouring
 * compatible regions are merged) after the addition.
 *
 * RETURNS:
 * 0 on success, -errno on failure.
 */
```

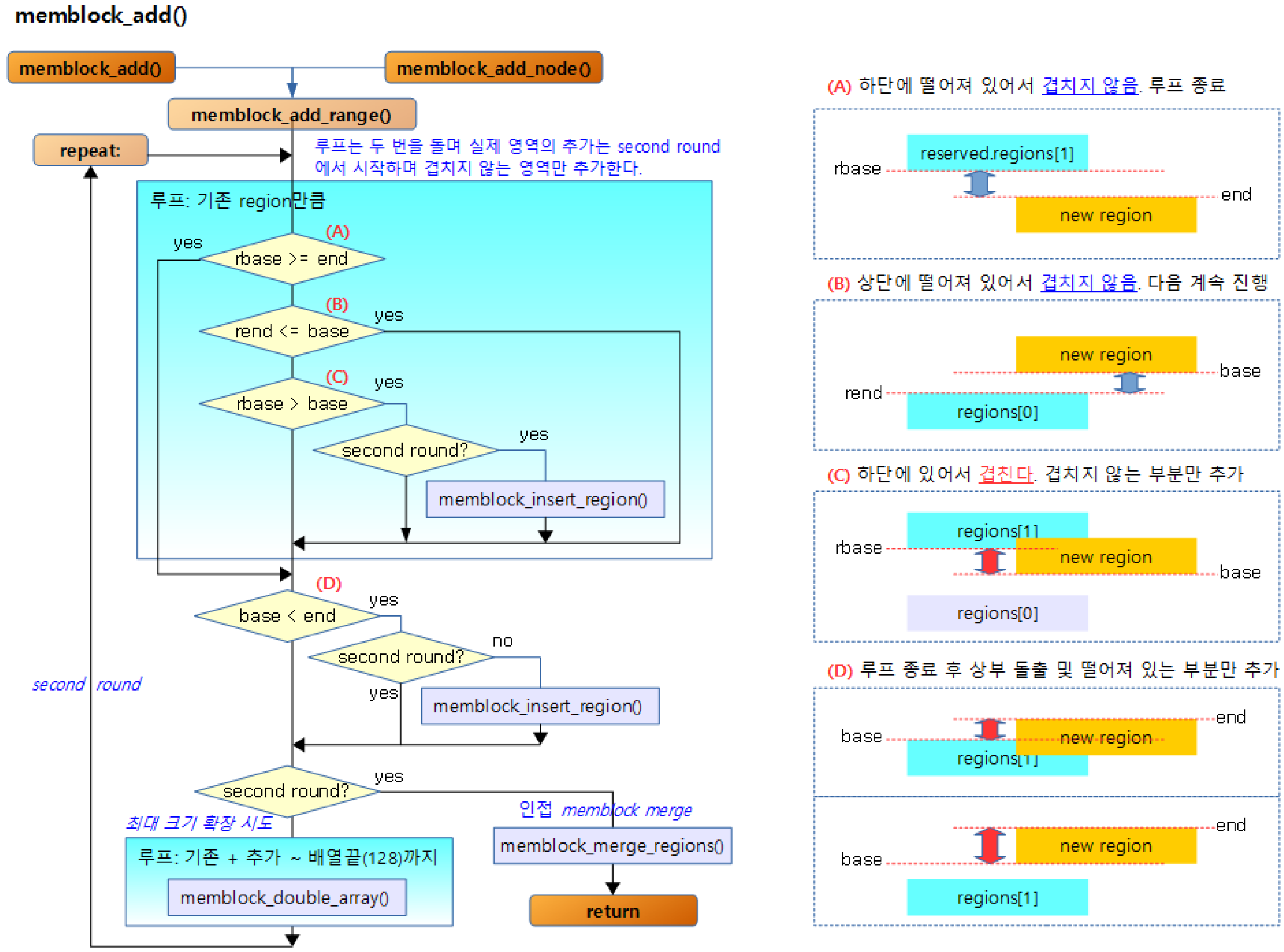
```
int __init_memblock memblock_add_range(struct memblock_type *type,
                                phys_addr_t base, phys_addr_t size,
                                int nid, enum memblock_flags flags)
{
        bool insert = false;
        phys_addr_t obase = base;
        phys_addr_t end = base + memblock_cap_size(base, &size);
        int idx, nr_new;
        struct memblock_region *rgn;

        if (!size)
                return 0;

        /* special case for empty array */
        if (type->regions[0].size == 0) {
                WARN_ON(type->cnt != 1 || type->total_size);
                type->regions[0].base = base;
                type->regions[0].size = size;
                type->regions[0].flags = flags;
                memblock_set_region_node(&type->regions[0], nid);
                type->total_size = size;
                return 0;
        }
repeat:
        /*
```

```
         * The following is executed twice.  Once with %false @insert and
         * then with %true.  The first counts the number of regions needed
         * to accommodate the new area.  The second actually inserts them.
         */
        base = obase;
        nr_new = 0;

        for_each_memblock_type(idx, type, rgn) {
                phys_addr_t rbase = rgn->base;
                phys_addr_t rend = rbase + rgn->size;

                if (rbase >= end)
                        break;
                if (rend <= base)
                        continue;
                /*
                 * @rgn overlaps.  If it separates the lower part of new
                 * area, insert that portion.
                 */
                if (rbase > base) {
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
                        WARN_ON(nid != memblock_get_region_node(rgn));
#endif
                        WARN_ON(flags != rgn->flags);
                        nr_new++;
                        if (insert)
                                memblock_insert_region(type, idx++, base,
                                                       rbase - base, nid,
                                                       flags);
                }
                /* area below @rend is dealt with, forget about it */
                base = min(rend, end);
        }

        /* insert the remaining portion */
        if (base < end) {
                nr_new++;
                if (insert)
                        memblock_insert_region(type, idx, base, end - base,
                                               nid, flags);
        }

        if (!nr_new)
                return 0;

        /*
         * If this was the first round, resize array and repeat for actual
         * insertions; otherwise, merge and return.
         */
        if (!insert) {
                while (type->cnt + nr_new > type->max)
                        if (memblock_double_array(type, obase, size) < 0)
                                return -ENOMEM;
                insert = true;
                goto repeat;
        } else {
                memblock_merge_regions(type);
                return 0;
        }
}
```

This function uses the requested arguments to add new memblock regions, inserting duplicates of existing memblock regions so that they don't overlap, and finally merge memblocks that have the same flag type and are bordered by neighboring memblocks. The return value is always 0.

- If the memblock area is empty in line 15~23 of the code, set the first memblock area and exit the function without checking for duplicates.
- In line 24 of code, it's repeated once via the repeat label, and when the first run is called the first round and the next execution is called the second round, the first round only increments the counter (nr_new) to check the number of memblocks that need to be inserted in the loop. In the second round, we spin a loop and actually insert the memblock into the place so that it doesn't overlap with the adjacent memblock.
- Loop around the memblock types requested in lines 33~35 of code.
- In code lines 37~38, the new memblock area is at the bottom of the memblock area it is comparing and does not overlap, so it exits the loop because it no longer needs to go through an iterative loop.
    - This corresponds to (A) in the figure below. Note that the regions[] array is sorted by base address. regions[0] is the lowest base address.
- In lines 39~40 of code, the new memblock area is on top of the memblock area you are comparing and do not overlap, so skip it to compare it to the next memblock.
    - This corresponds to (B) in the figure below.
- Lines 45~55 of the code Since the new memblock area overlaps with the memblock area being compared, the requested new memblock is inserted in this position, and the area is adjusted by the size of the area that does not overlap with the area of the memblock being compared.
    - This corresponds to (C) in the figure below.
- In case you need to insert base in line 57, you can preset the start address of the memblock area you want to insert.
- If the end of the new memblock area protrudes to the top after the loop ends in line 61~66, only the protruding part is added as a memblock.
    - This corresponds to (D) in the figure below.
- In line 75~80 of code, in the case of the first round, call the memblock_double_array( ) function to increase the number of entries in the memblock area by twice as much, only if the sum of the existing memblocks and the number of memblocks to be inserted exceeds the maximum number of controls. Repeat until you have enough entries.
- In line 81~84 of code, merge the memblocks inserted in the second round using the memblock_merge_regions( ) function to merge the memblocks that are adjacent to the surrounding memblocks and have the same flag type.

The following figure shows the flow for adding a requested memory area



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock2a.png)

In the figure below, six existing regions are registered in gray, and when a new memblock is added in light blue for the six cases, it is compared in the direction of the arrow starting from region[6].

- If the area to be added overlaps with the existing memblock area, it is first divided into non-overlapping areas and then the adjacent blocks are finally merged.
- Number of insertions
  - The number of actual insertions that occur when a memblock is added
- CNT count after merge
  - The number of memblocks that will eventually remain after being merged.

(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock3.png)

memblock_insert_region()

mm/memblock.c

```
/**
 * memblock_insert_region - insert new memblock region
 * @type:       memblock type to insert into
 * @idx:        index for the insertion point
 * @base:       base address of the new region
 * @size:       size of the new region
 * @nid:        node id of the new region
 * @flags:      flags of the new region
 *
 * Insert new memblock region [@base,@base+@size) into @type at @idx.
 * @type must already have extra room to accomodate the new region.
 */
```

```
static void __init_memblock memblock_insert_region(struct memblock_type *type,
                                                   int idx, phys_addr_t base,
                                                   phys_addr_t size,
                                                   int nid,
                                                   enum memblock_flags flags)
{
        struct memblock_region *rgn = &type->regions[idx];

        BUG_ON(type->cnt >= type->max);
        memmove(rgn + 1, rgn, (type->cnt - idx) * sizeof(*rgn));
        rgn->base = base;
        rgn->size = size;
        rgn->flags = flags;
        memblock_set_region_node(rgn, nid);
        type->cnt++;
```

```
16         type->total_size += size;
17 }
```

insert the new field into the specified index memblock position of the specified type. memblock_type struct's field cnt and total_size increment the counter and add the increased size.

- Copy back one space from the memblock to the last memblock of the specified index in line 10.
- In lines 11~14 of code, specify the node ID to support multi-nodes.
- In lines 15~16 of code, the memblock type information is incremented by the counter and added with the increased size.

Extending memblock arrays

memblock_double_array()

The array in which the memblock is located is small, so it can be doubled if it is called when it needs to be expanded. Let's analyze the memblock_double_array( ) function with the following code:

mm/memblock.c -1/2-

```
/**
 * memblock_double_array - double the size of the memblock regions array
 * @type: memblock type of the regions array being doubled
 * @new_area_start: starting address of memory range to avoid overlap with
 * @new_area_size: size of memory range to avoid overlap with
 *
 * Double the size of the @type regions array. If memblock is being used to
 * allocate memory for a new reserved regions array and there is a previously
 * allocated memory range [@new_area_start, @new_area_start + @new_area_size]
 * waiting to be reserved, ensure the memory used by the new array does
 * not overlap.
 *
 * Return:
 * 0 on success, -1 on failure.
 */
```

```
static int __init_memblock memblock_double_array(struct memblock_type *type,
                                                phys_addr_t new_area_start,
                                                phys_addr_t new_area_size)
{
        struct memblock_region *new_array, *old_array;
        phys_addr_t old_alloc_size, new_alloc_size;
        phys_addr_t old_size, new_size, addr, new_end;
        int use_slab = slab_is_available();
        int *in_slab;

        /* We don't allow resizing until we know about the reserved regions
         * of memory that aren't suitable for allocation
         */
        if (!memblock_can_resize)
                return -1;

        /* Calculate new doubled size */
        old_size = type->max * sizeof(struct memblock_region);
```

```
19          new_size = old_size << 1;
20          /*
21           * We need to allocated new one align to PAGE_SIZE,
22           *   so we can free them completely later.
23           */
24          old_alloc_size = PAGE_ALIGN(old_size);
25          new_alloc_size = PAGE_ALIGN(new_size);
26
27          /* Retrieve the slab flag */
28          if (type == &memblock.memory)
29                  in_slab = &memblock_memory_in_slab;
30          else
31                  in_slab = &memblock_reserved_in_slab;
32
33          /* Try to find some space for it. */
34          if (use_slab) {
35                  new_array = kmalloc(new_size, GFP_KERNEL);
36                  addr = new_array ? __pa(new_array) : 0;
37          } else {
38                  /* only exclude range when trying to double reserved.regions */
39                  if (type != &memblock.reserved)
40                          new_area_start = new_area_size = 0;
41
42                  addr = memblock_find_in_range(new_area_start + new_area_size,
43                                                  memblock.current_limit,
44                                                  new_alloc_size, PAGE_SIZE);
45                  if (!addr && new_area_size)
46                          addr = memblock_find_in_range(0,
47                                  min(new_area_start, memblock.current_limit),
48                                  new_alloc_size, PAGE_SIZE);
49
50                  new_array = addr ? __va(addr) : NULL;
51          }
```

- In lines 14~15 of the code, memblock_allow_resize( ) is called, restricting this function to work only after the memblock_can_resize global variable is set to 1.
  - Once the kernel image and memory region have been mapped, and the arm64_memblock_init() routine is performed and the memblock is ready to be used, it will be used in the next function call path.
    - setup_arch() -> memblock_allow_resize() is called at the end of -> paging_init().
- In line 18~25 of the code, prepare to allocate the new managed area to be twice the size of the existing managed area.
- In line 28~31 of the code, check whether the existing memblock management map is already in operation by switching to a slab, rather than being assigned by static or memblock.
- In lines 34~36 of the code, in the step where you can use slab, which is a regular memory allocator, memory is allocated with kmalloc( ).
- If the slab fails in line 37~40 of the code, it avoids the area requested by the memblock_add( ) function and finds a space where the new admin area can be used as memblock_find_in_range( ). If the request type is not of type reserved, i.e. type memory, the search for the allocation area starts from 0.
  - If the management area is expanded while registering a memory type, the newly allocated area can be searched for a vacant space within all previously registered memory areas. In any case, the newly added memory space will be an area that does not interfere with the

existing memory space. First of all, the space to be newly allocated to the management area should avoid the additional request area, so search the upper part of the request area first.

- In line 42~48 of the code, if it is not assigned in the first search, or if the request type is reserved, it will search the bottom of the request avoiding the additional request area.

mm/memblock.c -2/2-

```
01         if (!addr) {
02                 pr_err("memblock: Failed to double %s array from %ld to %ld entries !\n",
03                        type->name, type->max, type->max * 2);
04                 return -1;
05         }
06 
07         new_end = addr + new_size - 1;
08         memblock_dbg("memblock: %s is doubled to %ld at [%pa-%pa]",
09                         type->name, type->max * 2, &addr, &new_end);
10 
11         /*
12          * Found space, we now need to move the array over before we add the
13          * reserved region since it may be our reserved array itself that is
14          * full.
15          */
16         memcpy(new_array, type->regions, old_size);
17         memset(new_array + type->max, 0, old_size);
18         old_array = type->regions;
19         type->regions = new_array;
20         type->max <<= 1;
21 
22         /* Free old array. We needn't free it if the array is the static one */
23         if (*in_slab)
24                 kfree(old_array);
25         else if (old_array != memblock_memory_init_regions &&
26                  old_array != memblock_reserved_init_regions)
27                 memblock_free(__pa(old_array), old_alloc_size);
28 
29         /*
30          * Reserve the new array if that comes from the memblock.  Otherwise, we
31          * needn't do it
32          */
33         if (!use_slab)
34                 BUG_ON(memblock_reserve(addr, new_alloc_size));
35 
36         /* Update slab flag */
37         *in_slab = use_slab;
38 
39         return 0;
40 }
```

- In line 16~20 of the code, copy all the existing memblock areas to the newly assigned admin area start address, and reset the empty areas that are not copied to 0. It also changes the max to twice the original value.
- In lines 23~27 of code, turn off the existing management area before expansion. The initial management area is in the area of array variables declared at compile time, so I can't delete it,

so I just throw it away. If the management area to be deleted is not the initial area, it is handled according to the allocator type as follows.
- If you are already using Slab, use kfree( ) to release the existing area.
- If you didn't use slabs, use memblock_free ( ) to release the existing area.

memblock_allow_resize()

mm/memblock.c

```
void __init memblock_allow_resize(void)
{
        memblock_can_resize = 1;
}
```

Substitute 1 so that the memblock area can be expanded if needed.

merge memblocks

memblock_merge_regions()

mm/memblock.c

```
/**
 * memblock_merge_regions - merge neighboring compatible regions
 * @type: memblock type to scan
 *
 * Scan @type and merge neighboring compatible regions.
 */
```

```
01 static void __init_memblock memblock_merge_regions(struct memblock_type *type)
02 {
03         int i = 0;
04
05         /* cnt never goes below 1 */
06         while (i < type->cnt - 1) {
07                 struct memblock_region *this = &type->regions[i];
08                 struct memblock_region *next = &type->regions[i + 1];
09
10                 if (this->base + this->size != next->base ||
11                     memblock_get_region_node(this) !=
12                     memblock_get_region_node(next) ||
13                     this->flags != next->flags) {
14                         BUG_ON(this->base + this->size > next->base);
15                         i++;
16                         continue;
17                 }
18
19                 this->size += next->size;
20                 /* move forward from next + 1, index of which is i + 2 */
21                 memmove(next, next + 1, (type->cnt - (i + 2)) * sizeof(*next));
22                 type->cnt--;
23         }
24 }
```

If adjacent memory blocks use the same flag type, they are merged

- In line 6~8 of the code, the loop is reversed by the number of request memblock types.

- If there is no boundary in line 10~17 of the code, or if the flag state between the two memblocks is different, the memblocks are skipped without merging.
- If there is a boundary in line 19~22 of the code, merge the memblock. Note that in the case of merging, the index i value that specifies the regions[] array is not incremented for comparison with the next block once again.

memblock_cap_size()

mm/memblock.c

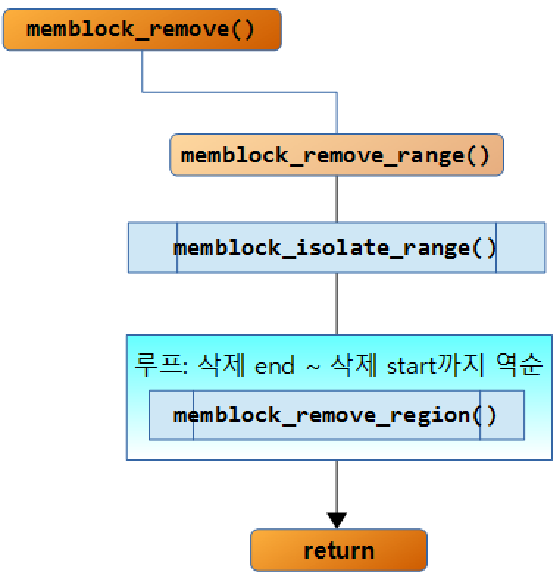
```
/* adjust *@size so that (@base + *@size) doesn't overflow, return new size */
static inline phys_addr_t memblock_cap_size(phys_addr_t base, phys_addr_t *size)
{
        return *size = min(*size, PHYS_ADDR_MAX - base);
}
```

If the area overflows beyond the unsigned long value, the overflow is truncated.

  - 64bit: min(0xffff_ffff_ffff_ffff – base, size)
  - 32bit: min(0xffff_ffff – base, size)
- If the area overflows, the size is recalculated, and the size is reduced by 1, making the last address byte of the system unusable.
  - e.g. 32bit: base=0xffff_0000, size=0xffff -> size=0xffff (this is normal)
  - e.g. 32bit: base=0xffff_0000, size=0x10000 -> size=0xffff (1 is smaller, so 0xffff_ffff address cannot be used)

Delete memblock

The following figure shows the calling relationship of the memblock_remove() function.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock5.png)

memblock_remove()

mm/memblock.c

```
int __init_memblock memblock_remove(phys_addr_t base, phys_addr_t size)
{
        phys_addr_t end = base + size - 1;

        memblock_dbg("%s: [%pa-%pa] %pS\n", __func__,
                     &base, &end, (void *)_RET_IP_);

        return memblock_remove_range(&memblock.memory, base, size);
}
```

Remove @size from the @base of the physical memory address from the memory type memblock area.

memblock_free()

mm/memblock.c

```
/**
 * memblock_free - free boot memory block
 * @base: phys starting address of the  boot memory block
 * @size: size of the boot memory block in bytes
 *
 * Free boot memory block previously allocated by memblock_alloc_xx() AP
I.
 * The freeing memory will not be released to the buddy allocator.
 */
```

```
int __init_memblock memblock_free(phys_addr_t base, phys_addr_t size)
{
        phys_addr_t end = base + size - 1;

        memblock_dbg("%s: [%pa-%pa] %pS\n", __func__,
                     &base, &end, (void *)_RET_IP_);

        kmemleak_free_part_phys(base, size);
        return memblock_remove_range(&memblock.reserved, base, size);
}
```

Remove @size from the @base of physical memory addresses from the reserved type memblock area.

memblock_remove_range()

mm/memblock.c

```
static int __init_memblock memblock_remove_range(struct memblock_type *t
ype,
                                          phys_addr_t base, phys_addr_t
size)
{
        int start_rgn, end_rgn;
        int i, ret;

        ret = memblock_isolate_range(type, base, size, &start_rgn, &end_
rgn);
        if (ret)
                return ret;

        for (i = end_rgn - 1; i >= start_rgn; i--)
                memblock_remove_region(type, i);
        return 0;
```

```
14 }
```

Remove the area from the physical address @base to @size from the memblock of the requested @type.

- In line 7~9 of the code, separate the memblock based on the start and end addresses of the area to be removed.
- Delete the memblock area corresponding to the area to be removed in line 11~12 of the code.

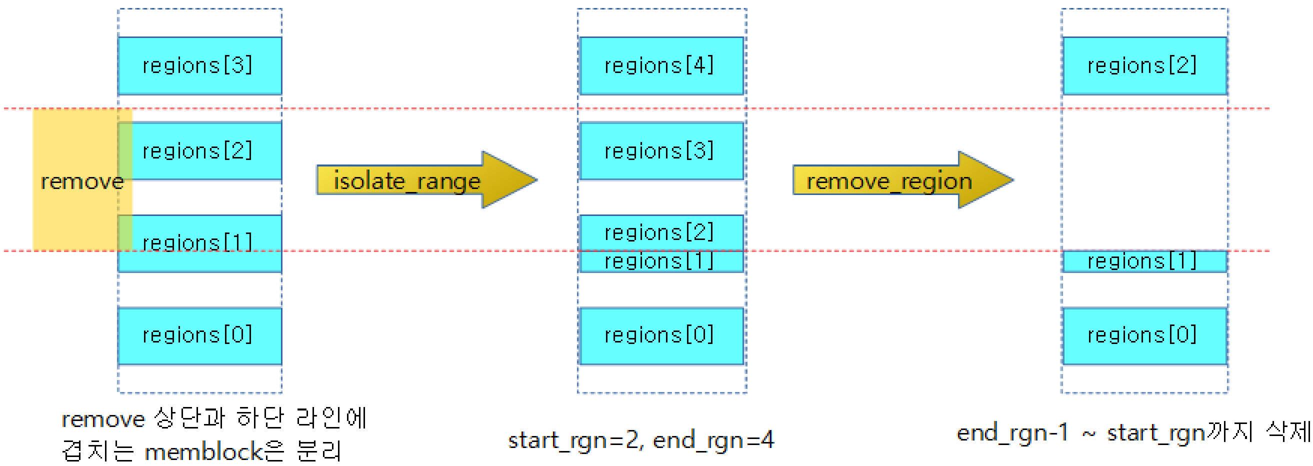
memblock_remove_region()

mm/memblock.c

```
static void __init_memblock memblock_remove_region(struct memblock_type *type, unsigned long r)
{
        type->total_size -= type->regions[r].size;
        memmove(&type->regions[r], &type->regions[r + 1],
                (type->cnt - (r + 1)) * sizeof(type->regions[r]));
        type->cnt--;

        /* Special case for empty arrays */
        if (type->cnt == 0) {
                WARN_ON(type->total_size != 0);
                type->cnt = 1;
                type->regions[0].base = 0;
                type->regions[0].size = 0;
                type->regions[0].flags = 0;
                memblock_set_region_node(&type->regions[0], MAX_NUMNODES);
        }
}
```

Delete the memblock corresponding to index r from the memblock of the requested @type. Jump to the location where the parent memblocks were deleted.

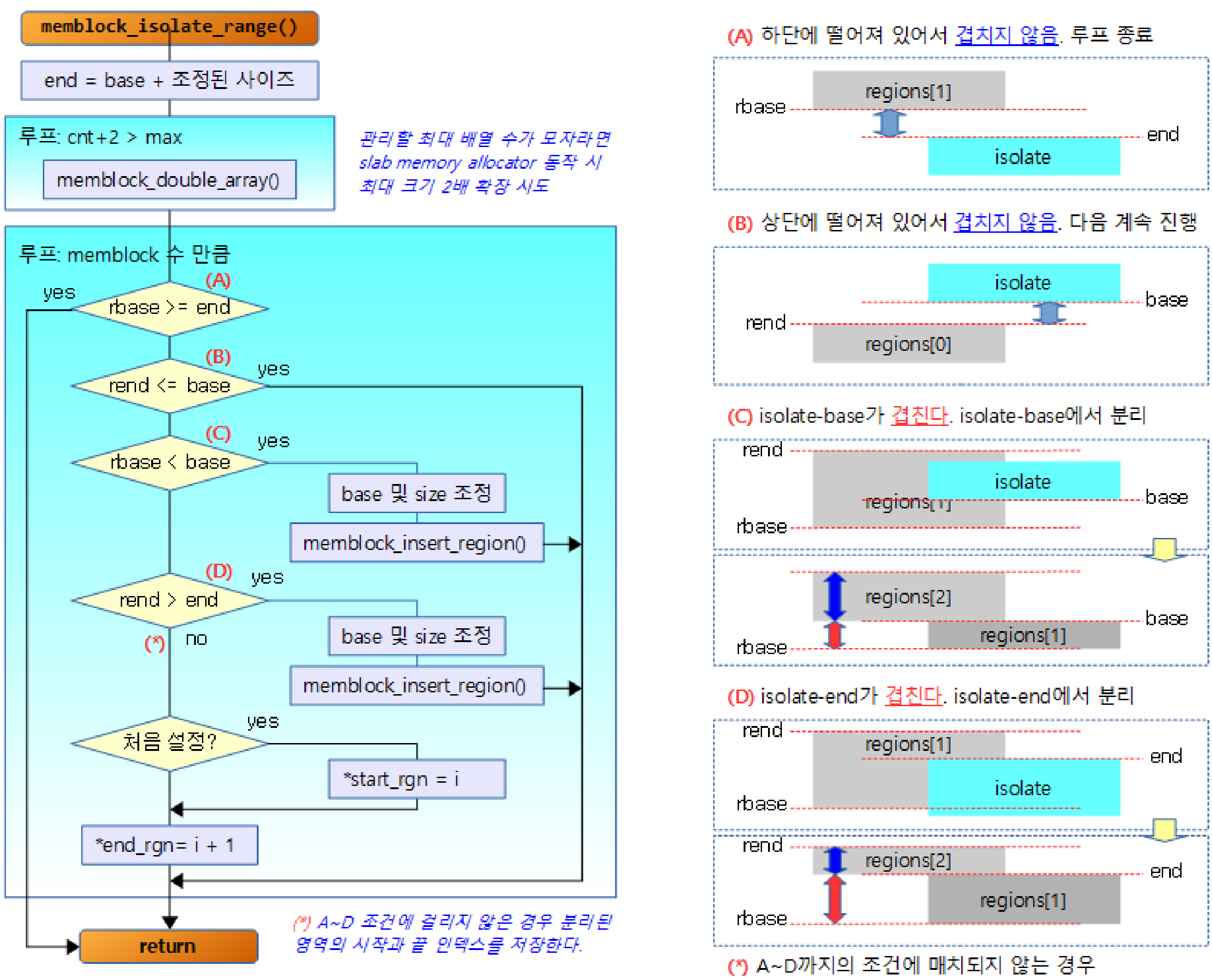
The following figure shows four memblock regions, of which the yellow remove range changes when you delete it.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock6.png)

Separation of memblocks

The following figure shows how the memblock_ioslate_range() function is processed.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock-7b.png)

memblock_isolate_range()

mm/memblock.c

```
/**
 * memblock_isolate_range - isolate given range into disjoint memblocks
 * @type: memblock type to isolate range for
 * @base: base of range to isolate
 * @size: size of range to isolate
 * @start_rgn: out parameter for the start of isolated region
 * @end_rgn: out parameter for the end of isolated region
 *
 * Walk @type and ensure that regions don't cross the boundaries defined by
 * [@base, @base + @size).  Crossing regions are split at the boundaries,
 * which may create at most two more regions.  The index of the first
 * region inside the range is returned in *@start_rgn and end in *@end_rgn.
 *
 * Return:
 * 0 on success, -errno on failure.
 */
```

```
static int __init_memblock memblock_isolate_range(struct memblock_type *type,
                                        phys_addr_t base, phys_addr_t size,
                                        int *start_rgn, int *end_rgn)
{
        phys_addr_t end = base + memblock_cap_size(base, &size);
        int idx;
        struct memblock_region *rgn;

        *start_rgn = *end_rgn = 0;

        if (!size)
                return 0;

        /* we'll create at most two more regions */
        while (type->cnt + 2 > type->max)
                if (memblock_double_array(type, base, size) < 0)
                        return -ENOMEM;

        for_each_memblock_type(idx, type, rgn) {
                phys_addr_t rbase = rgn->base;
                phys_addr_t rend = rbase + rgn->size;

                if (rbase >= end)
                        break;
                if (rend <= base)
                        continue;

                if (rbase < base) {
                        /*
                         * @rgn intersects from below.  Split and continue
                         * to process the next region - the new top half.
                         */
                        rgn->base = base;
                        rgn->size -= base - rbase;
                        type->total_size -= base - rbase;
                        memblock_insert_region(type, idx, rbase, base - rbase,
                                               memblock_get_region_node(rgn),
                                               rgn->flags);
                } else if (rend > end) {
                        /*
                         * @rgn intersects from above.  Split and redo the
                         * current region - the new bottom half.
                         */
                        rgn->base = end;
                        rgn->size -= end - rbase;
                        type->total_size -= end - rbase;
                        memblock_insert_region(type, idx--, rbase, end - rbase,
                                               memblock_get_region_node(rgn),
                                               rgn->flags);
                } else {
                        /* @rgn is fully contained, record it */
                        if (!*end_rgn)
                                *start_rgn = idx;
                        *end_rgn = idx + 1;
                }
        }

        return 0;
}
```

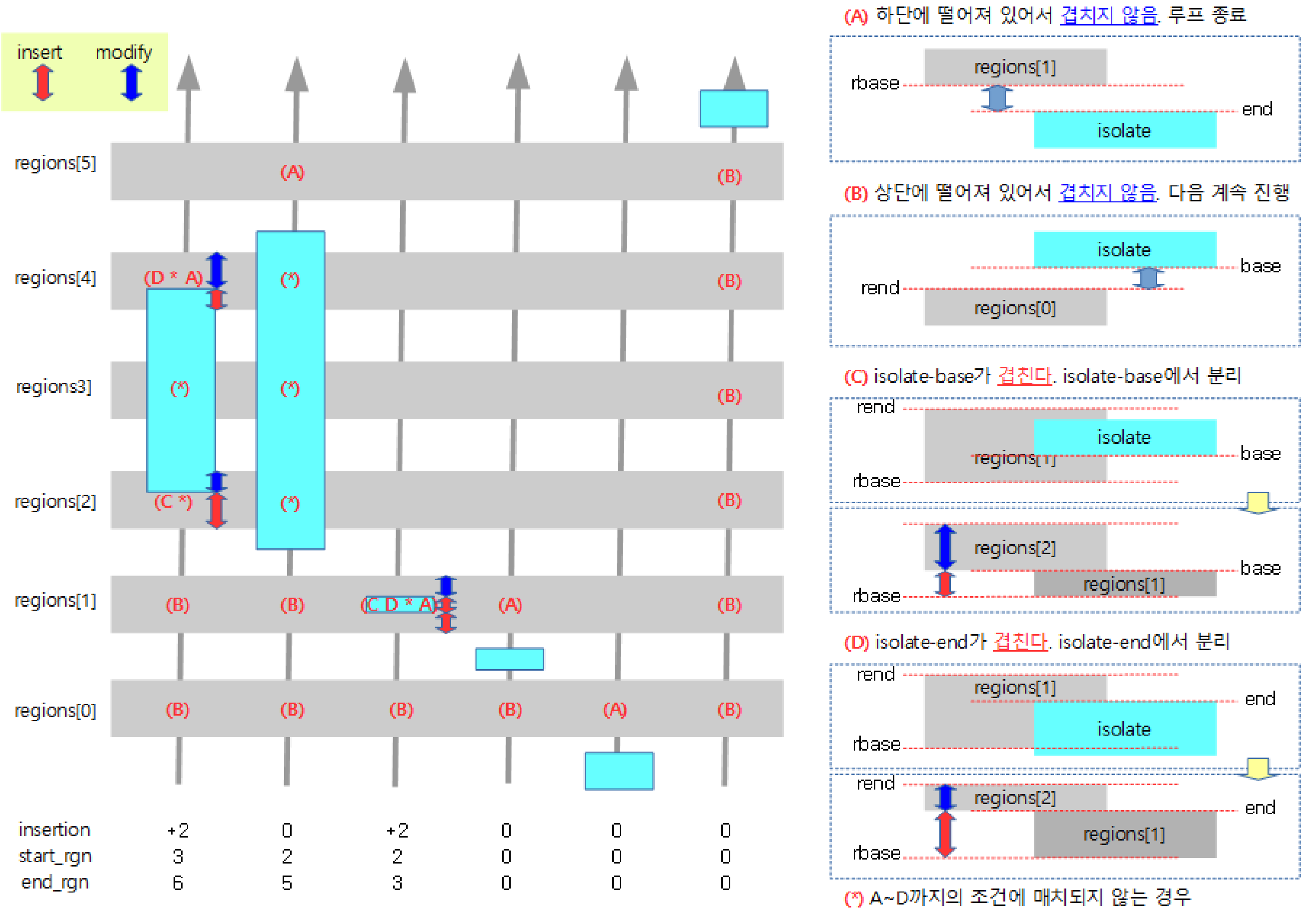
Separate the zones from the memblocks that overlap the top and bottom lines of the request area. The separated memblock start index is stored in the @start_rgn of the output argument, and the end index + 1 value is stored in the output argument @end_rgn. If there are no isolated memblock entries, zeros are stored in both output arguments.

- In line 5 of code, the end of the zone physical address is restricted to not exceed the end of the physical address that the architecture supports.
- In lines 15~17 of code, when we try to use the maximum number of those types, we double the management array. If it can't be scaled, it returns -ENOMEM indicating that there is not enough memory.
- In lines 19~21 of the code, the loop goes through the first memblock area to the last zone, and rbase and rend contain the start and end addresses of the memblock in the index of the loop.
- In lines 23~24 of code, if the start address of the current index area is greater than or equal to the top of the given area, it no longer needs to be processed by the parent index memblock, so it exits the loop. (Condition A in the figure below)
- In lines 25~26 of code, if the end address of the current index area is equal to or greater than the top of the given area, it is not yet an overlapping area, so it skips to handle the next index memblock. (Condition B in the figure below)
- In line 28~38 of the code, if the start address of the current index area is less than the start address of the given area, i.e., the start address of the given area overlaps with the current index memblock, it will be handled as follows. (Condition C in the figure below)
  - Move the current index memblock to the top, starting from the overlapping line at the bottom. In addition, starting from the overlapping line at the bottom of the index memblock, insert a new memblock at the bottom.
- In line 39~49 of the code, if the end address of the current index area is greater than the end address of the given area, i.e., the end address of the given area overlaps with the current index memblock, it will be handled as follows: (Condition D in the figure below)
  - Move the current index memblock to the top, starting from the overlapping line at the top. In addition, starting from the overlapping line at the top of the index memblock, insert the memblock at the bottom.
- If you don't encounter any conditions in code lines 50~55, the current index area is included in the given area. In this case, the current index value is specified for the start_rgn, and the current index value + 1 is specified for the end_rgn.

The following figure shows the comparison of six memblock regions in gray, each of which is given a light blue isolation region, in the direction of the arrow starting from region[6].

- If the area to be separated overlaps with an existing memblock area, the overlapping part divides the memblock. And add the parts that don't have a memblock.
- insertion
  - Shows the actual number of insertions that occur when a memblock is added.
- start_rgn
  - The index number of the existing memblock area where the beginning of the isolation range overlaps

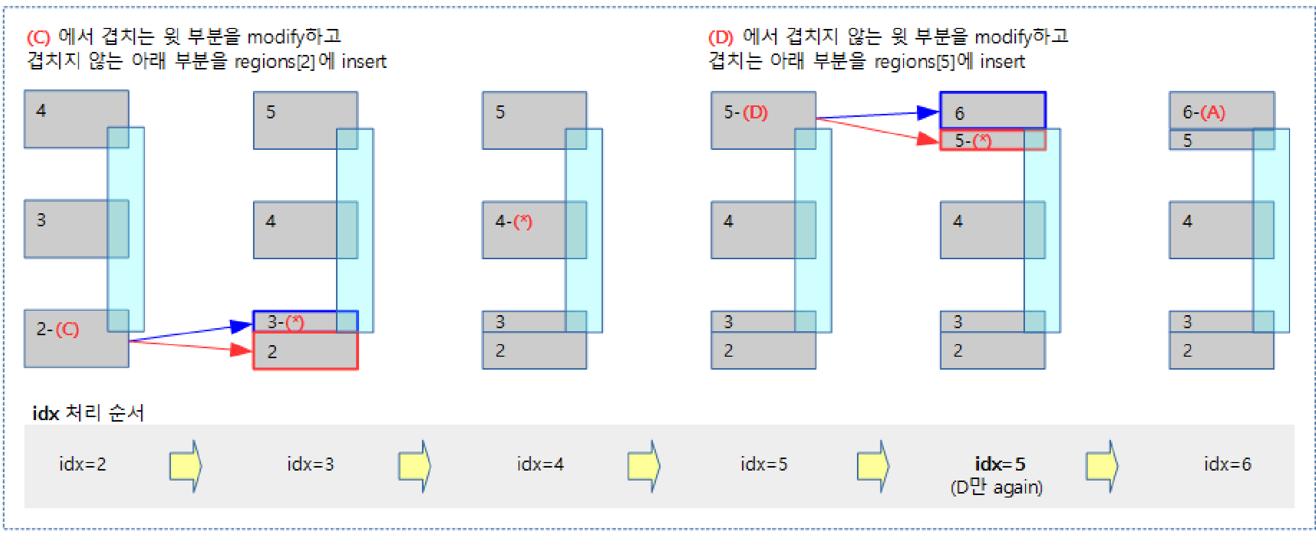
- end_rgn
  - Index number of the existing memblock area where the ends of the isolation range overlap + 1



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock-8c.png)

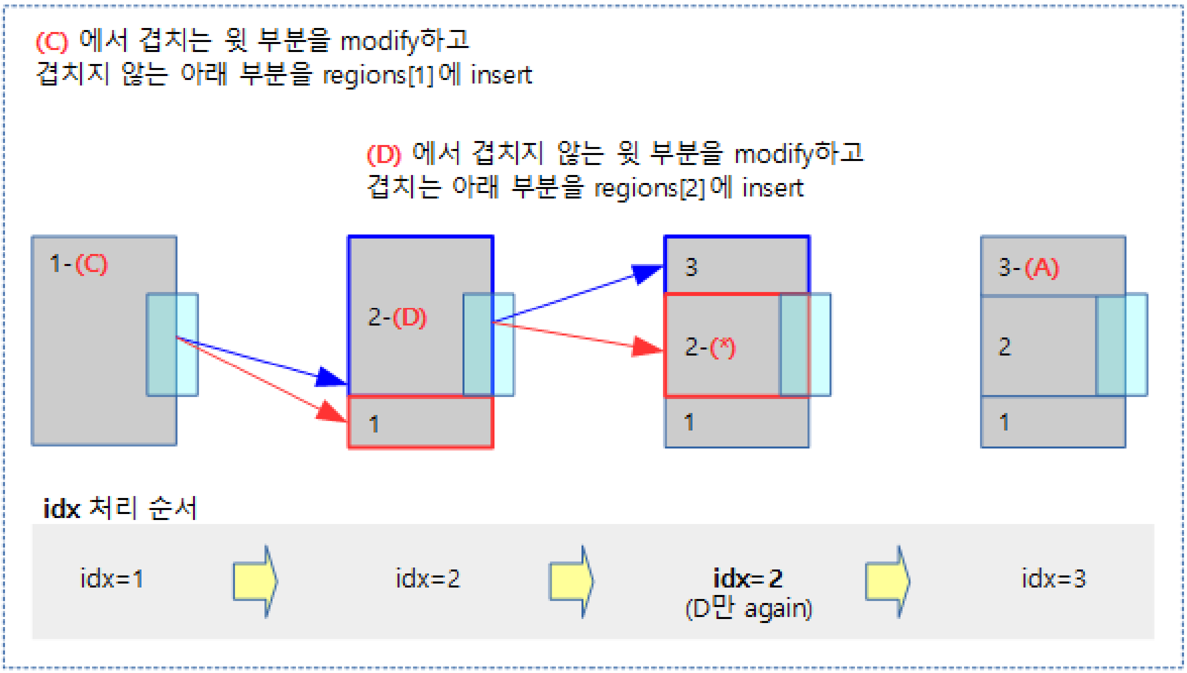
The following illustration shows the first part of the six examples above in more detail.

- If you make an insert in case (C) and (D), slide the idx area being processed up and add it.
- (D) If you proceed with a case, be careful to proceed with the area corresponding to IDX once again.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock17.png)

The following illustration shows the third part of the six examples above in more detail.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock18.png)

consultation

- Memblock – (1) | Sentence C – Current post
- Memblock – (2) (http://jake.dothome.co.kr/memblock-2) | Qc
- arm_memblock_init() (http://jake.dothome.co.kr/arm_memblock_init) | Qc
- arm64_memblock_init() (http://jake.dothome.co.kr/arm64_memblock_init) | Qc

- mm: Use memblock interface instead of bootmem (https://lwn.net/Articles/575443/) | LWN.net

---

7 thoughts to "Memblock – (1)"

SUNGJU LEE
2019-05-22 15:05 (http://jake.dothome.co.kr/memblock-1/#comment-213925)

There are two types of memblocks, memory and reserve, but what is the difference between these two?
Analyzing the actual code, I know that...
In the early days of the kernel, most of the time when the memory is allocated, it is used in the reserve type space, and

only once in the early days of the memory type. And later on, when I for_each_loop around a memblock and do certain actions, I mostly refer to the memory type...
I don't know what the difference is... I'm not exactly sure what the concept is all about.

RESPONSE (/MEMBLOCK-1/?REPLYTOCOM=213925#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2019-05-22 17:25 (http://jake.dothome.co.kr/memblock-1/#comment-213927)

Hello?

The memory type registers the range of the actual physical DRAM.
In the reserved type, as you know, it registers the area to be avoided when assigning.
When the memblock_alloc() function finds a space equal to size, it determines between free spaces within the memory type area, excluding reserved type areas.

I appreciate it.

RESPONSE (/MEMBLOCK-1/?REPLYTOCOM=213927#RESPOND)

SUNGJU LEE
2020-01-18 20:14 (http://jake.dothome.co.kr/memblock-1/#comment-229198)

There is a question in Figure D of memblock_isolate_range().

In order to enter the D condition, the C condition (rbase < base) must not be satisfied, and in the figure, region[1] and
isolate are drawn as the rbase end.
Am I misunderstanding?

RESPONSE (/MEMBLOCK-1/?REPLYTOCOM=229198#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-01-19 17:04 (http://jake.dothome.co.kr/memblock-1/#comment-229327)

Hello?

You understand very well. As you mentioned, the isolate box illustration sample of D condition is incorrect and has been corrected.
I appreciate it. And Happy New Year.

RESPONSE (/MEMBLOCK-1/?REPLYTOCOM=229327#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-01-20 23:31 (http://jake.dothome.co.kr/memblock-1/#comment-229365)

In addition, the sequence related to insertion under C and D conditions has been modified.

RESPONSE (/MEMBLOCK-1/?REPLYTOCOM=229365#RESPOND)

KWON YONGBEOM
2020-01-22 00:09 (http://jake.dothome.co.kr/memblock-1/#comment-229439)

You've made it easier and clearer. I think it will be very helpful for those who are analyzing for the first time like us.
I appreciate it. Happy New Year !

RESPONSE (/MEMBLOCK-1/?REPLYTOCOM=229439#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-01-29 21:22 (http://jake.dothome.co.kr/memblock-1/#comment-230386)

I wish you a very happy and fulfilling year.
I appreciate it.

RESPONSE (/MEMBLOCK-1/?REPLYTOCOM=230386#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

setup_dma_zone (http://jake.dothome.co.kr/setup_dma_zone/)

Memblock – (2) (http://jake.dothome.co.kr/memblock-2/)