Memblock – (2)

2016-01-26 (http://jake.dothome.co.kr/memblock-2/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.10>

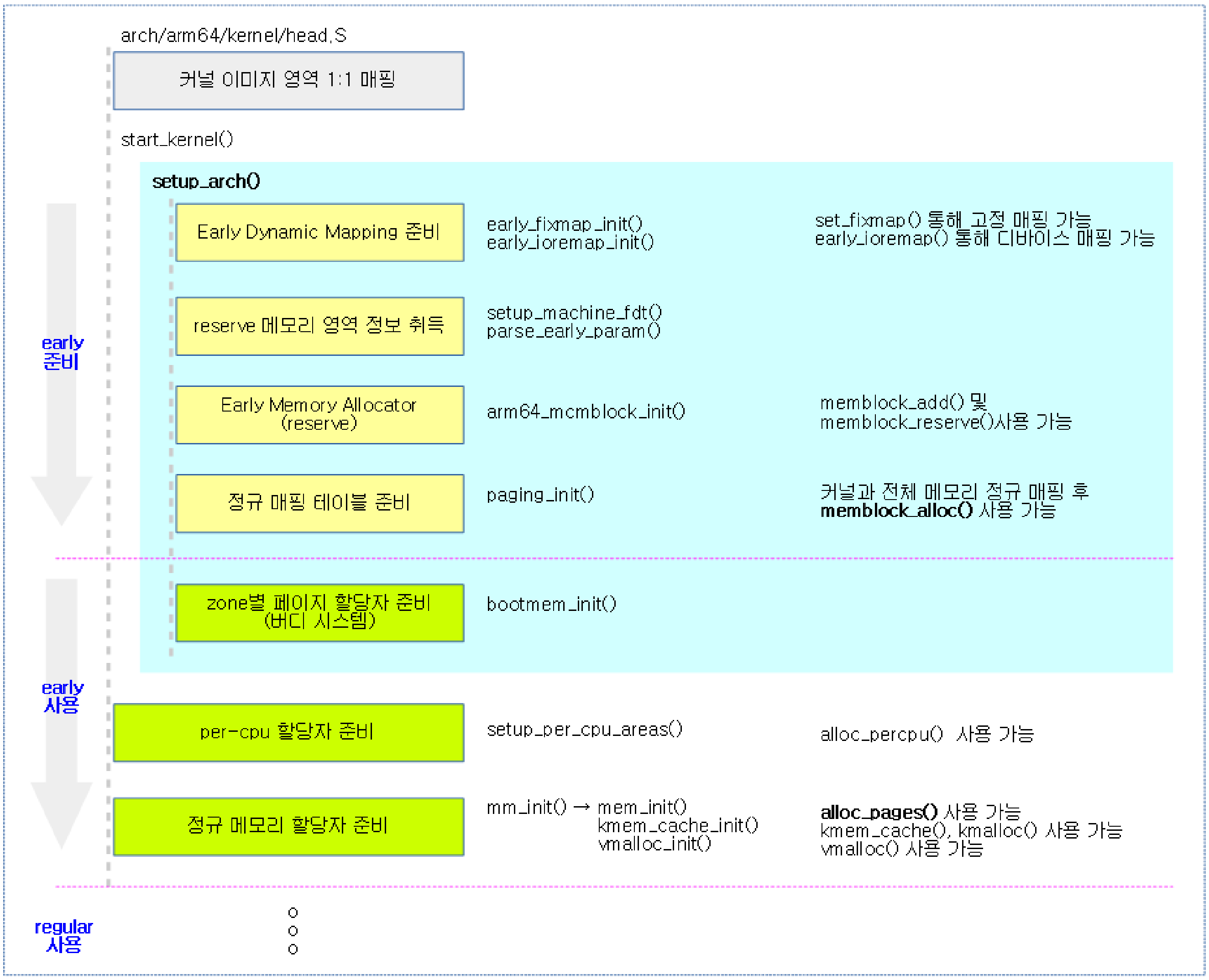
Early Memory Allocation

As an early memory allocator, you will use the memblock API to request the allocation and release of memory during the kernel bootup process and memory hot-swap.

Using a memblock before paging_init() completes

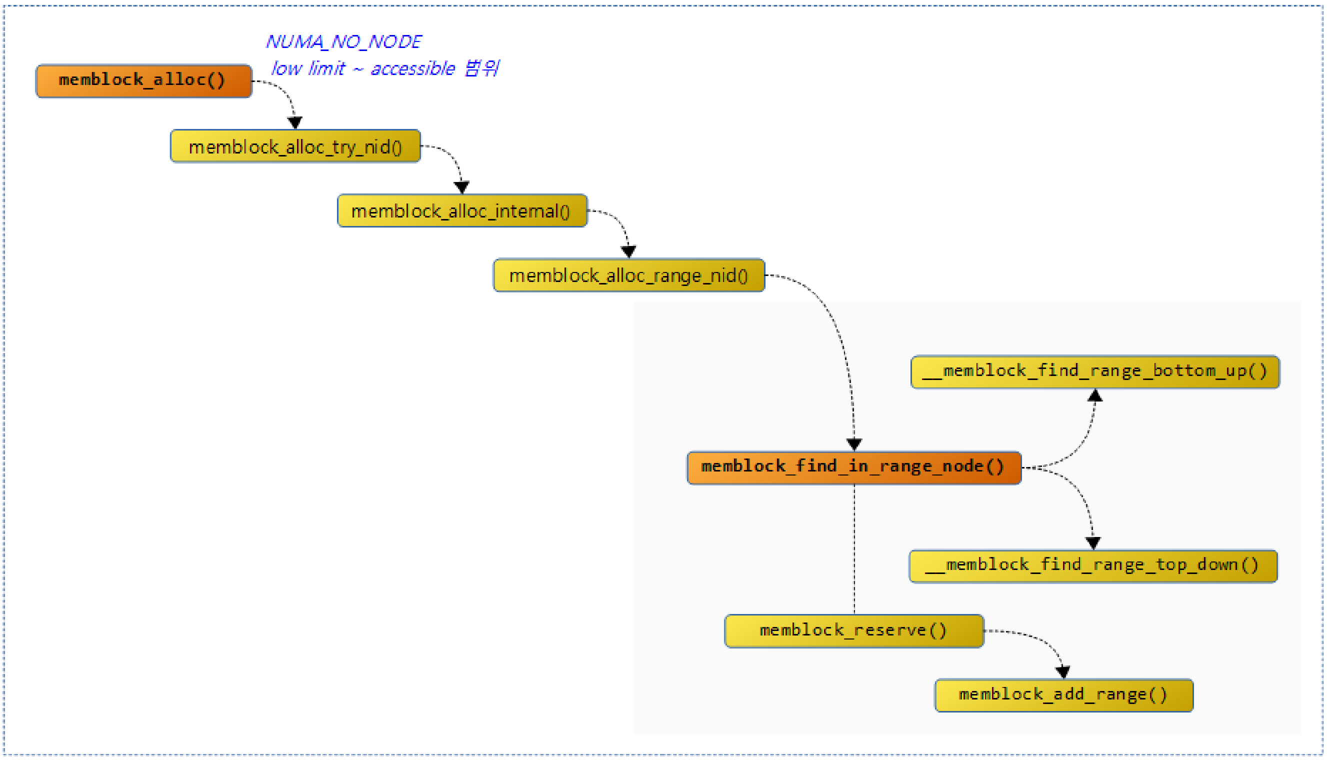
Even before paging_init() is complete, memory can be added with the memblock_add() function, and reserve space can be registered with the memblock_reserve() function. However, since it has not yet been linearly mapped to kernel memory, memory allocation cannot be done with memblock_alloc(). Therefore, memblock_alloc() should be used after paging_init() is complete.

The following figure shows the key memory mappings and the phased state of the allocators before the regular memory allocator is ready.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memory-1a.png)

The following figure shows the flow of APIs related to memblock allocation.


NUMA_NO_NODE
low limit ~ accessible 범위
memblock_alloc()
memblock_alloc_try_nid()
memblock_alloc_internal()
memblock_alloc_range_nid()
__memblock_find_range_bottom_up()
memblock_find_in_range_node()
__memblock_find_range_top_down()
memblock_reserve()
memblock_add_range()

(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock-9d.png)

memblock_alloc()

mm/memblock.c

```
static inline void * __init memblock_alloc(phys_addr_t size,  phys_addr_t align)
{
        return memblock_alloc_try_nid(size, align, MEMBLOCK_LOW_LIMIT,
                                      MEMBLOCK_ALLOC_ACCESSIBLE, NUMA_NO_NODE);
}
```

Request to allocate @size memblocks, sorted by @align units. Returns the virtual address of the allocated memory on success, and null on failure.

- MEMBLOCK_LOW_LIMIT(0)
  - When allocating, limit the bottom end of memory to be allocated by top down.
- MEMBLOCK_ALLOC_ACCESSIBLE(0)
  - Allocation is limited to the current_limit value of the memblock.
- NUMA_NO_NODE(-1)
  - No matter which node you allocate from, you can request it without restriction.

memblock_alloc_try_nid()

mm/memblock.c

```
/**
 * memblock_alloc_try_nid - allocate boot memory block with panicking
 * @size: size of memory block to be allocated in bytes
 * @align: alignment of the region and block's size
 * @min_addr: the lower bound of the memory region from where the allocation
 *        is preferred (phys address)
 * @max_addr: the upper bound of the memory region from where the allocation
 *            is preferred (phys address), or %MEMBLOCK_ALLOC_ACCESSIBLE to
 *            allocate only from memory limited by memblock.current_limit value
 * @nid: nid of the free area to find, %NUMA_NO_NODE for any node
 *
 * Public function, provides additional debug information (including caller
 * info), if enabled. This function zeroes the allocated memory.
 *
 * Return:
 * Virtual address of allocated memory block on success, NULL on failure.
 */
```

```
void * __init memblock_alloc_try_nid(
                        phys_addr_t size, phys_addr_t align,
                        phys_addr_t min_addr, phys_addr_t max_addr,
                        int nid)
{
        void *ptr;

        memblock_dbg("%s: %llu bytes align=0x%llx nid=%d from=%pa max_addr=%pa %pS\n",
                     __func__, (u64)size, (u64)align, nid, &min_addr,
                        &max_addr, (void *)_RET_IP_);
        ptr = memblock_alloc_internal(size, align,
                                      min_addr, max_addr, nid, false);
        if (ptr)
                memset(ptr, 0, size);

        return ptr;
}
```

Request the allocation of @size memblock areas in the node @nid, sorted by @align units. Returns the virtual address of the allocated memory on success, and null on failure.

memblock_alloc_internal()

mm/memblock.c

```
/**
 * memblock_alloc_internal - allocate boot memory block
 * @size: size of memory block to be allocated in bytes
 * @align: alignment of the region and block's size
 * @min_addr: the lower bound of the memory region to allocate (phys address)
 * @max_addr: the upper bound of the memory region to allocate (phys address)
 * @nid: nid of the free area to find, %NUMA_NO_NODE for any node
 * @exact_nid: control the allocation fall back to other nodes
 *
 * Allocates memory block using memblock_alloc_range_nid() and
 * converts the returned physical address to virtual.
 *
 * The @min_addr limit is dropped if it can not be satisfied and the allocation
 * will fall back to memory below @min_addr. Other constraints, such
 * as node and mirrored memory will be handled again in
 * memblock_alloc_range_nid().
 *
 * Return:
 * Virtual address of allocated memory block on success, NULL on failure.
 */
```

```
static void * __init memblock_alloc_internal(
                                phys_addr_t size, phys_addr_t align,
                                phys_addr_t min_addr, phys_addr_t max_addr,
                                int nid, bool exact_nid)
{
        phys_addr_t alloc;

        /*
         * Detect any accidental use of these APIs after slab is ready, as at
         * this moment memblock may be deinitialized already and its
         * internal data may be destroyed (after execution of memblock_free_all)
         */
        if (WARN_ON_ONCE(slab_is_available()))
                return kzalloc_node(size, GFP_NOWAIT, nid);

        if (max_addr > memblock.current_limit)
                max_addr = memblock.current_limit;

        alloc = memblock_alloc_range_nid(size, align, min_addr, max_addr, nid,
20                      alloc = memblock_alloc_range_nid(size, align, 0, max_addr, nid,
21                                                      exact_nid);
22
23          /* retry allocation without lower limit */
24           if (!alloc && min_addr)
25                  alloc = memblock_alloc_range_nid(size, align, 0, max_addr, nid,
26                                                  exact_nid);
27
28          if (!alloc)
29              return NULL;
30
31          return phys_to_virt(alloc);
32  }
```

Requests the allocation of @size memblock areas in the requested node @nid, sorted by @align. Converts the assigned physical address into a virtual address and returns it.

- If this function is called after the regular memory slab allocator has been executed in lines 13~14, the slab allocator is used to allocate memory.
- In lines 16~17 of code, limit the @max_addr to not exceed the current_limit of the memblock.
- In line 19~21 of the code, try to allocate memory within a limited limit range.
- If the allocation fails in line 24~26 of the code, release the limit range and then try to allocate the memory.
- If the allocation still fails on lines 28~29 of the code, it returns null.
- If the assignment is successful on line 31 of the code, it is converted to a virtual address (lm) and returned.

memblock_alloc_range_nid()

mm/memblock.c

```
01 /**
02  * memblock_alloc_range_nid - allocate boot memory block
03  * @size: size of memory block to be allocated in bytes
04  * @align: alignment of the region and block's size
05  * @start: the lower bound of the memory region to allocate (phys address)
06  * @end: the upper bound of the memory region to allocate (phys address)
07  * @nid: nid of the free area to find, %NUMA_NO_NODE for any node
08  * @exact_nid: control the allocation fall back to other nodes
09  *
10  * The allocation is performed from memory region limited by
11  * memblock.current_limit if @end == %MEMBLOCK_ALLOC_ACCESSIBLE.
12  *
13  * If the specified node can not hold the requested memory and @exact_nid
14  * is false, the allocation falls back to any node in the system.
15  *
16  * For systems with memory mirroring, the allocation is attempted first
17  * from the regions with mirroring enabled and then retried from any
18  * memory region.
19  *
20  * In addition, function sets the min_count to 0 using kmemleak_alloc_phys for
21  * allocated boot memory block, so that it is never reported as leaks.
22  *
23  * Return:
24  * Physical address of allocated memory block on success, %0 on failure.
```

```
25  */
```

```
01 phys_addr_t __init memblock_alloc_range_nid(phys_addr_t size,
02                                         phys_addr_t align, phys_addr_t start,
03                                         phys_addr_t end, int nid,
04                                         bool exact_nid)
05 {
06         enum memblock_flags flags = choose_memblock_flags();
07         phys_addr_t found;
08 
09         if (WARN_ONCE(nid == MAX_NUMNODES, "Usage of MAX_NUMNODES is deprecated. Use NUMA_NO_NODE instead\n"))
10                 nid = NUMA_NO_NODE;
11 
12         if (!align) {
13                 /* Can't use WARNs this early in boot on powerpc */
14                 dump_stack();
15                 align = SMP_CACHE_BYTES;
16         }
17 
18 again:
19         found = memblock_find_in_range_node(size, align, start, end, nid,
20                                             flags);
21         if (found && !memblock_reserve(found, size))
22                 goto done;
23 
24         if (nid != NUMA_NO_NODE && !exact_nid) {
25                 found = memblock_find_in_range_node(size, align, start,
26                                                     end, NUMA_NO_NODE,
27                                                     flags);
28                 if (found && !memblock_reserve(found, size))
29                         goto done;
30         }
31 
32         if (flags & MEMBLOCK_MIRROR) {
33                 flags &= ~MEMBLOCK_MIRROR;
34                 pr_warn("Could not allocate %pap bytes of mirrored memory\n",
35                         &size);
36                 goto again;
37         }
38 
39         return 0;
40 
41 done:
42         /* Skip kmemleak for kasan_init() due to high volume. */
43         if (end != MEMBLOCK_ALLOC_KASAN)
44                 /*
45                  * The min_count is set to 0 so that memblock allocated
46                  * blocks are never reported as leaks. This is because many
47                  * of these blocks are only referred via the physical
48                  * address which is not looked up by kmemleak.
49                  */
50                 kmemleak_alloc_phys(found, size, 0, 0);
51 
52         return found;
53 }
```

Request the allocation of @size memblock area arranged by @align in the @nid of the requested node in the range of @start ~ @end. If successful, the assigned physical address is returned.

- In line 6 of the code, get the flag to use for the assignment. (Mirror Flag)

- If you used MAX_NUMNODES (default=9) as the nid value in line 10~16 of the code, you should use the value NUMA_NO_NODE(-1) instead. This is warned in a message.
- In lines 12~16 of code, if there is no @align request value, perform a stack dump to find out which function was called, and then use the SMP_CACHE_BYTES value (ARM1:64) to sort the @aline value by L64 cache.
- In code lines 18~22, again: is the label. Look for empty areas and reserve them to allocate them.
- If the @nid is specified in line 24~30 of the code, but the @exact_nid is false, we unlimit the node, look for empty areas again, and reserve them to allocate them.
- In line 32~37 of code, if there is a mirror setting in the system memory, remove the mirror flag to try to allocate it outside of the mirror area, and then move it to the again: label to try again.
- If the area to be finally allocated at line 39 is not found, it returns 0.
- In code lines 41~52, the out: label. If you assigned it normally, it returns the physical address it finds.

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

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

Add @size area from the physical address @base to the reserved area of the memblock.

Find Memblock Empty Areas

memblock_find_in_range()

mm/memblock.c

```
/**
 * memblock_find_in_range - find free area in given range
 * @start: start of candidate range
 * @end: end of candidate range, can be %MEMBLOCK_ALLOC_ANYWHERE or
 *       %MEMBLOCK_ALLOC_ACCESSIBLE
 * @size: size of free area to find
 * @align: alignment of free area to find
 *
 * Find @size free area aligned to @align in the specified range.
 *
 * Return:
 * Found address on success, 0 on failure.
 */
```

```
phys_addr_t __init_memblock memblock_find_in_range(phys_addr_t start,
                                        phys_addr_t end, phys_addr_t size,
                                        phys_addr_t align)
{
        phys_addr_t ret;
        enum memblock_flags flags = choose_memblock_flags();

again:
        ret = memblock_find_in_range_node(size, align, start, end,
                                            NUMA_NO_NODE, flags);

        if (!ret && (flags & MEMBLOCK_MIRROR)) {
                pr_warn("Could not allocate %pap bytes of mirrored memory\n",
                        &size);
                flags &= ~MEMBLOCK_MIRROR;
                goto again;
        }

        return ret;
}
```

In the range @start ~ @end, @align find the @size of space requested in the reserved memblock type.

- If you can't find a space when a mirror flag is requested, remove the mirror flag and try to find the space again.

memblock_find_in_range_node()

mm/memblock.c

```
/**
 * memblock_find_in_range_node - find free area in given range and node
 * @size: size of free area to find
 * @align: alignment of free area to find
 * @start: start of candidate range
 * @end: end of candidate range, can be %MEMBLOCK_ALLOC_ANYWHERE or
 *       %MEMBLOCK_ALLOC_ACCESSIBLE
 * @nid: nid of the free area to find, %NUMA_NO_NODE for any node
 * @flags: pick from blocks based on memory attributes
 *
 * Find @size free area aligned to @align in the specified range and node.
 *
 * When allocation direction is bottom-up, the @start should be greater
 * than the end of the kernel image. Otherwise, it will be trimmed. The
 * reason is that we want the bottom-up allocation just near the kernel
 * image so it is highly likely that the allocated memory and the kernel
 * will reside in the same node.
 *
 * If bottom-up allocation failed, will try to allocate memory top-down.
 *
 * Return:
 * Found address on success, 0 on failure.
 */
```

```
phys_addr_t __init_memblock memblock_find_in_range_node(phys_addr_t size,
                                        phys_addr_t align, phys_addr_t start,
                                        phys_addr_t end, int nid,
                                        enum memblock_flags flags)
{
        phys_addr_t kernel_end, ret;

        /* pump up @end */
        if (end == MEMBLOCK_ALLOC_ACCESSIBLE ||
10               end == MEMBLOCK_ALLOC_KASAN)
11                   end = memblock.current_limit;
12 
13           /* avoid allocating the first page */
14           start = max_t(phys_addr_t, start, PAGE_SIZE);
15           end = max(start, end);
16           kernel_end = __pa_symbol(_end);
17 
18           /*
19            * try bottom-up allocation only when bottom-up mode
20            * is set and @end is above the kernel image.
21            */
22           if (memblock_bottom_up() && end > kernel_end) {
23                   phys_addr_t bottom_up_start;
24 
25                   /* make sure we will allocate above the kernel */
26                   bottom_up_start = max(start, kernel_end);
27 
28                   /* ok, try bottom-up allocation first */
29                   ret = __memblock_find_range_bottom_up(bottom_up_start, e
   nd,
30                                                         size, align, nid,
   flags);
31                   if (ret)
32                           return ret;
33 
34                   /*
35                    * we always limit bottom-up allocation above the kerne
   l,
36                    * but top-down allocation doesn't have the limit, so
37                    * retrying top-down allocation may succeed when bottom-
   up
38                    * allocation failed.
39                    *
40                    * bottom-up allocation is expected to be fail very rare
   ly,
41                    * so we use WARN_ONCE() here to see the stack trace if
42                    * fail happens.
43                    */
44                   WARN_ONCE(IS_ENABLED(CONFIG_MEMORY_HOTREMOVE),
45                             "memblock: bottom-up allocation failed, memory
   hotremove may be affected\nn
46 ");
47           }
48 
49           return __memblock_find_range_top_down(start, end, size, align, n
   id,
50                                                 flags);
51 }
```

In the range of @start ~ @end @nid of the requested node, @align find the space of the requested @size in the reserved memblock type.

- If lines 9~11 tell you to search within the limit range, limit the @end to the current_limit of the memblock.
- In line 14 of the code, @start excludes page 0.
- In line 22~48 of the code, try to search bottom-up according to the architecture. In this case, the @end should be higher than the kernel space.
  - Starting with kernel version 3.13-rc1, the ability to allocate bottom up memory has been added.
- Try to search top-down on line 49~50 of the code.
  - Uses top-down from ARM, ARM64, x86 NUMA, etc.

MEMBLOCK_HOTPLUG

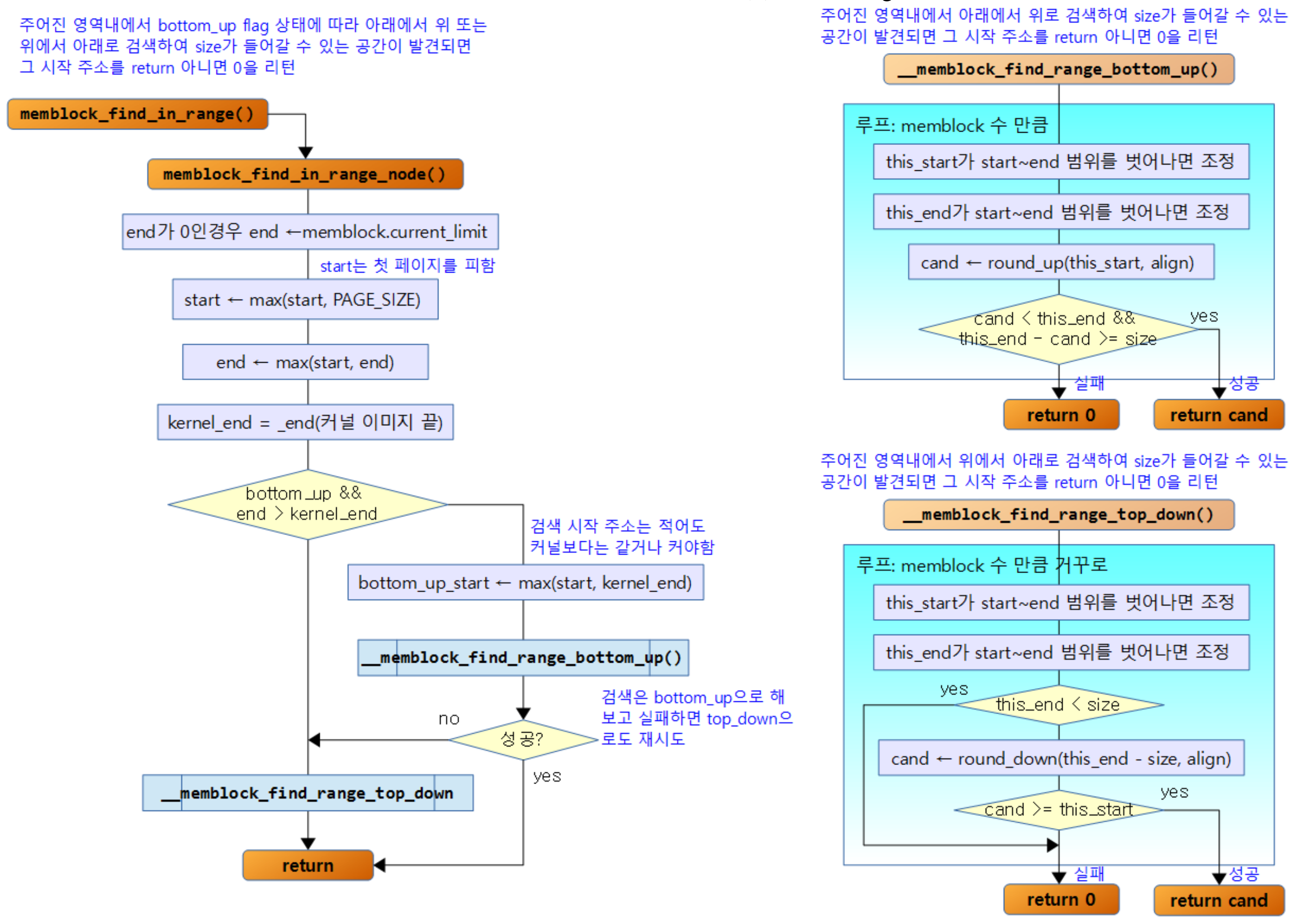
- With the addition of the memory hotplug feature to Linux, a new movable attribute field has been created in memory nodes.
- For movable nodes, a code has been added that allows the entire node to migrate the pages in use to another node while performing the detachable function due to the hotplug function.
- The kernel and the memory it uses are installed on nodes that do not support hotplugs, and these nodes always have non-movable properties.
- Currently, x86 64-bit high-capacity servers support hotplugs, which have at least 16G of capacity and are usually located on top of non-moveable nodes with kernel images. Under these conditions, when the kernel allocates memory, it does so starting with the movable node at the top of the memory. In this case, when a moveable node is removed, a large amount of memory that has been allocated and used has to be migrated from the movable node to another node.
- In order to reduce the frequency of migration of data pages due to the hotplug feature, it became necessary to allocate memory from the bottom up (from the lower address to the higher address), and code was added to allow memory allocation to be allocated from where the kernel is, i.e. from the end of the kernel (_end) to the top. The memory thus allocated is very likely to be co-located on a non-movable node with the kernel.
- Until now (kernel v4.4), the ARM architecture has not had memory hotplugs, unlike CPU hotplugs. Therefore, memory allocation uses a top-down approach.
  - CONFIG_MEMORY_HOTPLUG, CONFIG_HOTPLUG_CPU
- 참고: mm/memblock.c: introduce bottom-up allocation mode (https://github.com/torvalds/linux/commit/79442ed189acb8b949662676e750eda173c06f9b)

MEMBLOCK_MIRROR

- Among the functions of the x86 XEON server system, the memory can be set to dual mirrors to ensure high reliability. If you mirror the entire memory, you don't need to intervene kernel software. However, if you need to mirror only a part of the memory instead of the entire memory, set this flag in the memory memblock area registered to prioritize kernel memory and use it. This information is configured by the server system's UEFI firmware by passing the mirror settings to the kernel.

See: Address Range Mirror (2016) | Taku Izumi – Download PDF (https://events.static.linuxfound.org/sites/events/files/slides/Address%20Range%20Memory%20Mirroring-RC.pdf)

The following figure shows the processing of the memblock_find_in_range() function.


주어진 영역내에서 bottom_up flag 상태에 따라 아래에서 위 또는 위에서 아래로 검색하여 size가 들어갈 수 있는 공간이 발견되면 그 시작 주소를 return 아니면 0을 리턴
memblock_find_in_range()
memblock_find_in_range_node()
end가 0인경우 end ←memblock.current_limit
start는 첫 페이지를 피함
start ← max(start, PAGE_SIZE)
end ← max(start, end)
kernel_end = _end(커널 이미지 끝)
bottom_up && end > kernel_end
검색 시작 주소는 적어도 커널보다는 같거나 커야함
bottom_up_start ← max(start, kernel_end)
__memblock_find_range_bottom_up()
검색은 bottom_up으로 해보고 실패하면 top_down으로도 재시도
no
성공?
yes
__memblock_find_range_top_down
return
주어진 영역내에서 아래에서 위로 검색하여 size가 들어갈 수 있는 공간이 발견되면 그 시작 주소를 return 아니면 0을 리턴
__memblock_find_range_bottom_up()
루프: memblock 수 만큼
this_start가 start~end 범위를 벗어나면 조정
this_end가 start~end 범위를 벗어나면 조정
cand ← round_up(this_start, align)
cand < this_end && this_end - cand >= size
yes
실패
성공
return 0
return cand
주어진 영역내에서 위에서 아래로 검색하여 size가 들어갈 수 있는 공간이 발견되면 그 시작 주소를 return 아니면 0을 리턴
__memblock_find_range_top_down()
루프: memblock 수 만큼 거꾸로
this_start가 start~end 범위를 벗어나면 조정
this_end가 start~end 범위를 벗어나면 조정
yes
this_end < size
cand ← round_down(this_end - size, align)
yes
cand >= this_start
실패
성공
return 0
return cand

(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock4.png)

__memblock_find_range_top_down()

mm/memblock.c

```
/**
 * __memblock_find_range_top_down - find free area utility, in top-down
 * @start: start of candidate range
 * @end: end of candidate range, can be %MEMBLOCK_ALLOC_ANYWHERE or
 *       %MEMBLOCK_ALLOC_ACCESSIBLE
 * @size: size of free area to find
 * @align: alignment of free area to find
 * @nid: nid of the free area to find, %NUMA_NO_NODE for any node
 * @flags: pick from blocks based on memory attributes
 *
 * Utility called from memblock_find_in_range_node(), find free area top-down.
 *
 * Return:
 * Found address on success, 0 on failure.
 */
```

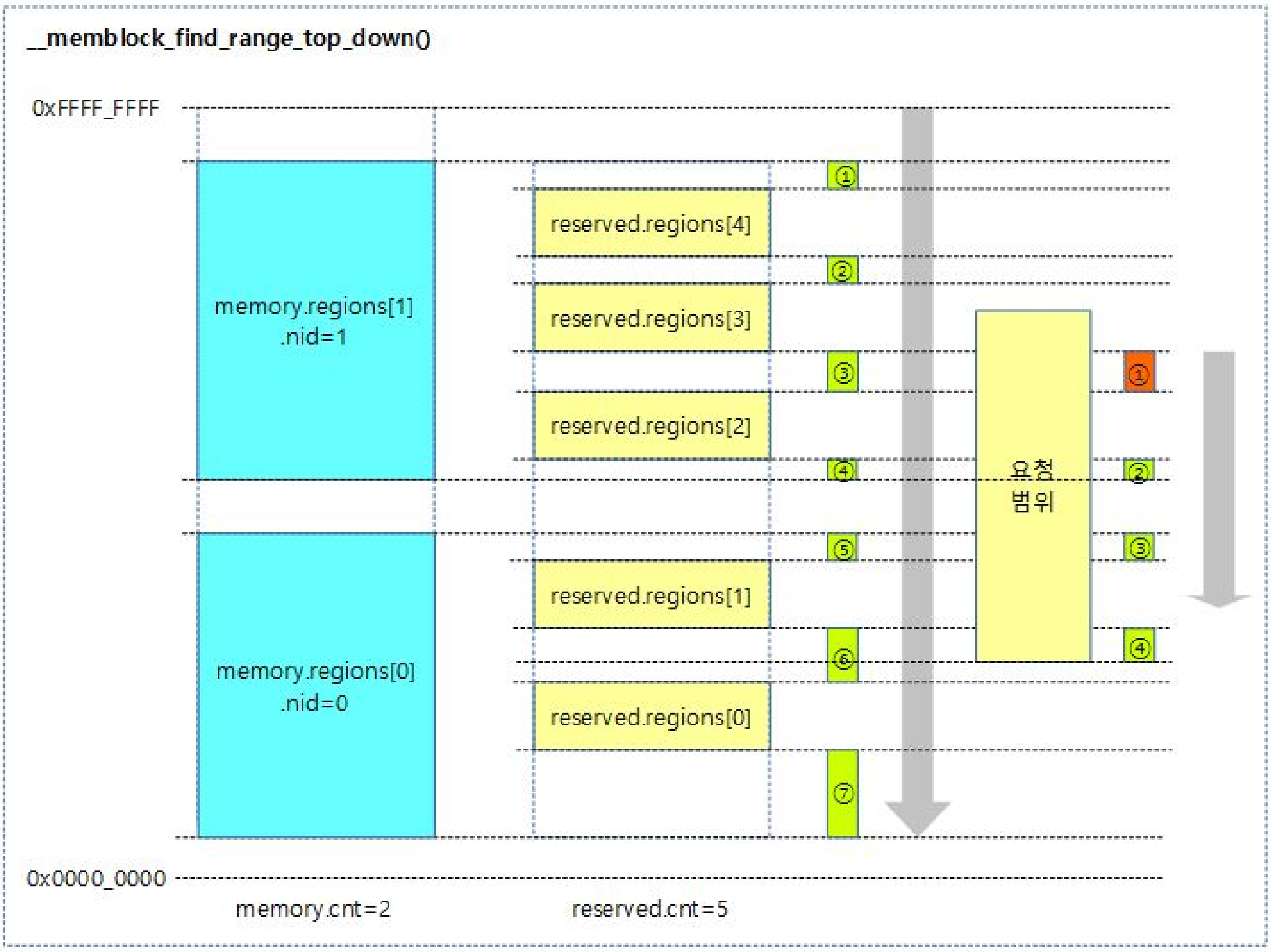
```
static phys_addr_t __init_memblock
__memblock_find_range_top_down(phys_addr_t start, phys_addr_t end,
                               phys_addr_t size, phys_addr_t align, int nid,
                               enum memblock_flags flags)
{
        phys_addr_t this_start, this_end, cand;
        u64 i;

        for_each_free_mem_range_reverse(i, nid, flags, &this_start, &this_end,
                                        NULL) {
```

```
11                  this_start = clamp(this_start, start, end);
12                  this_end = clamp(this_end, start, end);
13
14                  if (this_end < size)
15                          continue;
16
17                  cand = round_down(this_end - size, align);
18                  if (cand >= this_start)
19                          return cand;
20          }
21
22          return 0;
23  }
```

It searches the unreserved space of the segment where the memory exists from top to bottom, and if it finds @size of space @align within the range of @start ~ @end of the requested node @nid, it returns the starting address of the physical address.

- In code lines 9~12, loop around the empty memblock space and learn it one by one.
- In line 14~15 of the code, when calculating the following cand, we need to perform this_end – size, but the result is less than 0, so we need to skip it in advance in the under-flow.
- Compare the requested size in code lines 17~19 with the aligned size. If the range of the known free domain can contain size, it will successfully return the cand address.

In the figure below, when the __memblock_find_range_top_down() function is executed, 7 free spaces are obtained through for_each_free_mem_range_reverse(), and the cases included in the range from start to end are compressed into 4 and the address of this space is returned if it meets the match condition that includes the requested size in order from 1 to 4.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock_find_range_top_down-1c.jpg)

Iteration Macros

for_each_mem_region()

mm/memblock.c

```
/**
 * for_each_mem_region - itereate over memory regions
 * @region: loop variable
 */
```

```
#define for_each_mem_region(region)
\
        for (region = memblock.memory.regions;
\
             region < (memblock.memory.regions + memblock.memory.cnt);
\
             region++)
```

It designates the memory area of the memblock to the @region from beginning to end.

for_each_reserved_mem_region()

mm/memblock.c

```
/**
 * for_each_reserved_mem_region - itereate over reserved memory regions
 * @region: loop variable
 */
```

```
#define for_each_reserved_mem_region(region)
\
        for (region = memblock.reserved.regions;
\
             region < (memblock.reserved.regions + memblock.reserved.cn
t); \
             region++)
```

It iterates the reserved area of the memblock from beginning to end, assigning it to the @region.

for_each_memblock_type()

mm/memblock.c

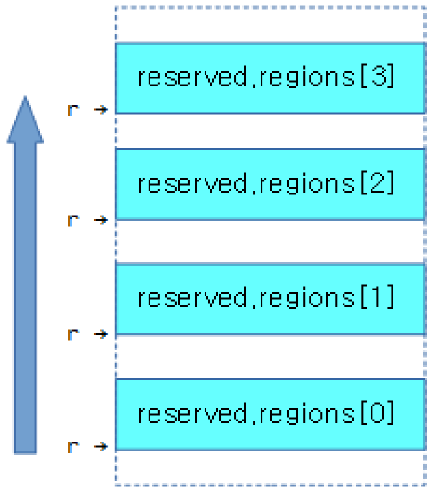
```
#define for_each_memblock_type(i, memblock_type, rgn)
\
        for (i = 0, rgn = &memblock_type->regions[0];
\
             i < memblock_type->cnt;
\
             i++, rgn = &memblock_type->regions[i])
```

The area of the requested @memblock_type is assigned to the @rgn from beginning to end, and the order number starting from 0 is assigned to the @i.

Example of the operation process)

```
.   struct memblock_region *r;
    for_each_reserved_mem_region(r) {
        print_info(“base=0x%x, size=0x%x\n”, r.base, r.size);
    }
```

When there are four reserved regions, the r value iterates from the bottom memblock region to the topmost memblock region to specify the memblock_region struct pointer.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock10.png)

for_each_mem_pfn_range()

include/linux/memblock.h

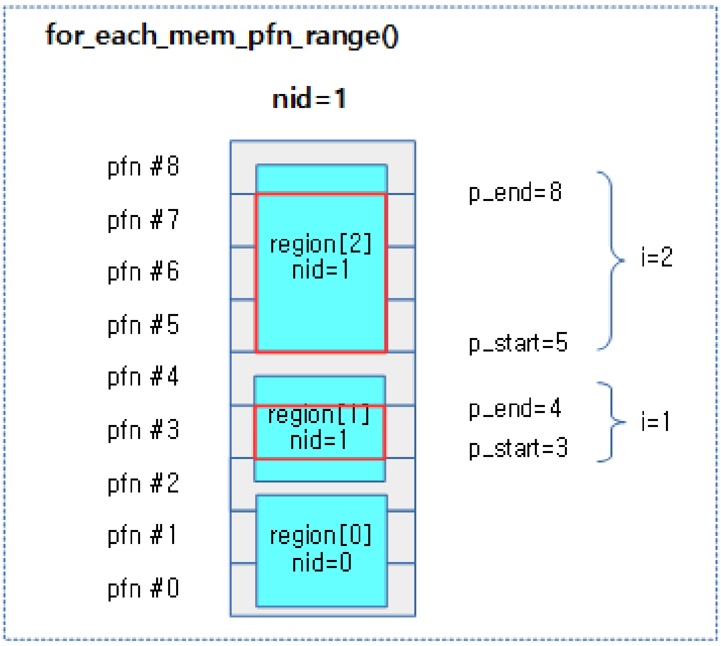
```
/**
 * for_each_mem_pfn_range - early memory pfn range iterator
 * @i: an integer used as loop variable
 * @nid: node selector, %MAX_NUMNODES for all nodes
 * @p_start: ptr to ulong for start pfn of the range, can be %NULL
 * @p_end: ptr to ulong for end pfn of the range, can be %NULL
 * @p_nid: ptr to int for nid of the range, can be %NULL
 *
 * Walks over configured memory ranges.
 */
```

```
#define for_each_mem_pfn_range(i, nid, p_start, p_end, p_nid) \
        for (i = -1, __next_mem_pfn_range(&i, nid, p_start, p_end, p_nid); \
             i >= 0; __next_mem_pfn_range(&i, nid, p_start, p_end, p_nid))
```

The loop is based on the @nid of the requested node, excluding fragmented pages, i.e., a memblock area with at least one intact page. The information of the matched memblock region is @i with the index number of the memblock area, @p_start contains the pfn of the beginning of the intact page (not the fragmented page), the @p_end contains the end pfn + 1 of the intact page (not fragmented page), and finally the node ID is obtained from the @p_nid.

In the figure below, when running the for_each_mem_pfn_range() macro loop with the value of nid=1, the case of matching the conditions is represented by a red box, and each value is as follows.

- Except for the memblock area where nid=0 at the bottom, as shown below, two matches occur.
  - i value is 1, which gives us p_start=3, p_end=4, p_nid=1.
  - i value is 2, so we get p_start=5, p_end=8, p_nid=1.
- The actual memory memblocks are very large pages and are almost always aligned, so they are very different from the figure below, but they should be used to help you understand how they are calculated.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/for_each_mem_pfn_range-1b.png)

__next_mem_pfn_range()

mm/memblock.c

```
/*
 * Common iterator interface used to define for_each_mem_range().
 */
```

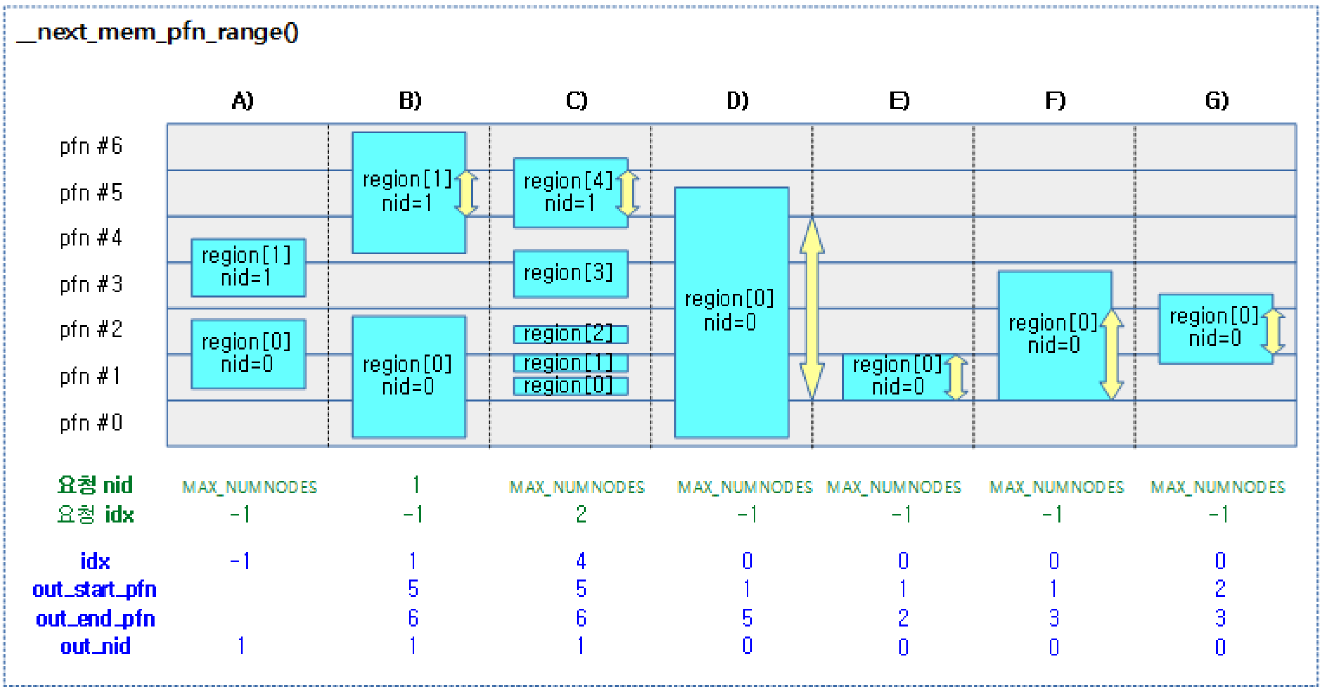
```
void __init_memblock __next_mem_pfn_range(int *idx, int nid,
                                unsigned long *out_start_pfn,
                                unsigned long *out_end_pfn, int *out_nid)
{
        struct memblock_type *type = &memblock.memory;
        struct memblock_region *r;

        while (++*idx < type->cnt) {
                r = &type->regions[*idx];

                if (PFN_UP(r->base) >= PFN_DOWN(r->base + r->size))
                        continue;
                if (nid == MAX_NUMNODES || nid == r->nid)
                        break;
        }
        if (*idx >= type->cnt) {
                *idx = -1;
                return;
        }

        if (out_start_pfn)
22                      *out_start_pfn = PFN_UP(r->base);
23              if (out_end_pfn)
24                      *out_end_pfn = PFN_DOWN(r->base + r->size);
25              if (out_nid)
26                      *out_nid = r->nid;
27      }
```

Starting with the next memblock area, the requested @nid and unfragmented pages are given the starting pfn and ending pfn+1 values in the @out_start_pfn and @out_end_pfn. In addition, the ID value of the matched node is also obtained from the @out_nid.

- In line 8~12 of the code, exclude the requested @idx area of the area registered in the memory memblock type and loop from that area.
- In lines 14~15 of code, if the memblock area does not contain more than 1 intact unfragmented page, skip it.
- In line 16~17 of code, if the node @nid of the request node matches the node of the zone, or if it is a request for any node, it will exit the loop.
- If it is not matched by the end of the loop on code lines 19~22, it returns -1 to idx.
- In lines 24~25 of the code, the out_start_pfn contains the starting pfn, which contains the page in its entirety, not fragmentation.
- In code lines 26~27, the out_end_pfn contains the end pfn, which contains the page intact and not fragmented.
- In lines 28~29 of code, put the node ID of the memblock area in the out_nid.

The figure below shows how the __next_mem_pfn_range() function is executed for each of the seven cases, looking for blocks that match the conditions from request idx+7. The figure below is also much smaller than the actual memory memblocks, so it should be used to help you understand how to calculate them

- In the case of A), there is no one complete unfragmented page.
- In the case of B), the node ID is specified.
- @idx
  - Index of the matched memblock area
- @out_start_pfn
  - Start of an unfragmented page pfn
- @out_end_pfn
  - End of unfragmented page pfn + 1
- @out_nid
  - Node IDs of matched memblock zones



	A)	B)	C)	D)	E)	F)	G)
요청 nid	MAX_NUMNODES	1	MAX_NUMNODES	MAX_NUMNODES	MAX_NUMNODES	MAX_NUMNODES	MAX_NUMNODES
요청 idx	-1	-1	2	-1	-1	-1	-1
idx	-1	1	4	0	0	0	0
out_start_pfn		5	5	1	1	1	2
out_end_pfn		6	6	5	2	3	3
out_nid	1	1	1	0	0	0	0

(http://jake.dothome.co.kr/wp-content/uploads/2016/01/next_mem_pfn_range-1c.png)

for_each_free_mem_range()

include/linux/memblock.h

```
/**
 * for_each_free_mem_range - iterate through free memblock areas
 * @i: u64 used as loop variable
 * @nid: node selector, %NUMA_NO_NODE for all nodes
 * @flags: pick from blocks based on memory attributes
 * @p_start: ptr to phys_addr_t for start address of the range, can be %NULL
 * @p_end: ptr to phys_addr_t for end address of the range, can be %NULL
 * @p_nid: ptr to int for nid of the range, can be %NULL
 *
 * Walks over free (memory && !reserved) areas of memblock.  Available as
 * soon as memblock is initialized.
 */
```

```
#define for_each_free_mem_range(i, nid, flags, p_start, p_end, p_nid)          \
        for_each_mem_range(i, &memblock.memory, &memblock.reserved,   \
                           nid, flags, p_start, p_end, p_nid)
```

In the memory area of the specified node @nid, the area except the reserved area is free memory, and the @p_start@flags knows the starting physical address, @p_end the end physical address, @p_nid the node ID, and the flag in the memory area of the specified node.

for_each_mem_range()

include/linux/memblock.h

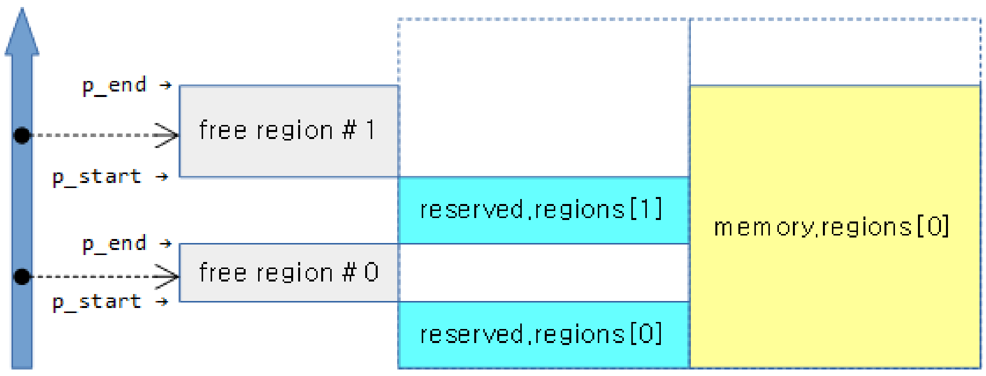
```
/**
 * for_each_mem_range - iterate through memblock areas from type_a and not
 * included in type_b. Or just type_a if type_b is NULL.
 * @i: u64 used as loop variable
 * @type_a: ptr to memblock_type to iterate
 * @type_b: ptr to memblock_type which excludes from the iteration
 * @nid: node selector, %NUMA_NO_NODE for all nodes
 * @flags: pick from blocks based on memory attributes
 * @p_start: ptr to phys_addr_t for start address of the range, can be %
NULL
 * @p_end: ptr to phys_addr_t for end address of the range, can be %NULL
 * @p_nid: ptr to int for nid of the range, can be %NULL
 */
```

```
#define for_each_mem_range(i, type_a, type_b, nid, flags,
\
                           p_start, p_end, p_nid)
\
        for (i = 0, __next_mem_range(&i, nid, flags, type_a, type_b,
\
                                     p_start, p_end, p_nid);
\
             i != (u64)ULLONG_MAX;
\
             __next_mem_range(&i, nid, flags, type_a, type_b,
\
                              p_start, p_end, p_nid))
```

When given a region of type A and a region of type B of the specified node id, it assigns this area to the p_start, p_end, and p_nid arguments in that order.

- The argument @type_b can also be given as null.

As shown below, the area excluding the reserved memblock type area in the memory memblock type area on the right is the area where the free area in the left gray box is matched. Loops are provided for those matched areas.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock11.png)

__next_mem_range()

mm/memblock.c

```
/**
 * __next__mem_range - next function for for_each_free_mem_range() etc.
 * @idx: pointer to u64 loop variable
 * @nid: node selector, %NUMA_NO_NODE for all nodes
 * @flags: pick from blocks based on memory attributes
 * @type_a: pointer to memblock_type from where the range is taken
 * @type_b: pointer to memblock_type which excludes memory from being ta
ken
```

```
 * @out_start: ptr to phys_addr_t for start address of the range, can be %NULL
 * @out_end: ptr to phys_addr_t for end address of the range, can be %NULL
 * @out_nid: ptr to int for nid of the range, can be %NULL
 *
 * Find the first area from *@idx which matches @nid, fill the out
 * parameters, and update *@idx for the next iteration.  The lower 32bit of
 * *@idx contains index into type_a and the upper 32bit indexes the
 * areas before each region in type_b.  For example, if type_b regions
 * look like the following,
 *
 *      0:[0-16), 1:[32-48), 2:[128-130)
 *
 * The upper 32bit indexes the following regions.
 *
 *      0:[0-0), 1:[16-32), 2:[48-128), 3:[130-MAX)
 *
 * As both region arrays are sorted, the function advances the two indices
 * in lockstep and returns each intersection.
 */
```

```
void __init_memblock __next_mem_range(u64 *idx, int nid,
                                      enum memblock_flags flags,
                                      struct memblock_type *type_a,
                                      struct memblock_type *type_b,
                                      phys_addr_t *out_start,
                                      phys_addr_t *out_end, int *out_nid)
{
        int idx_a = *idx & 0xffffffff;
        int idx_b = *idx >> 32;

        if (WARN_ONCE(nid == MAX_NUMNODES,
        "Usage of MAX_NUMNODES is deprecated. Use NUMA_NO_NODE instead\n"))
                nid = NUMA_NO_NODE;

        for (; idx_a < type_a->cnt; idx_a++) {
                struct memblock_region *m = &type_a->regions[idx_a];

                phys_addr_t m_start = m->base;
                phys_addr_t m_end = m->base + m->size;
                int         m_nid = memblock_get_region_node(m);

                /* only memory regions are associated with nodes, check it */
                if (nid != NUMA_NO_NODE && nid != m_nid)
                        continue;

                /* skip hotpluggable memory regions if needed */
                if (movable_node_is_enabled() && memblock_is_hotpluggable(m))
                        continue;

                /* if we want mirror memory skip non-mirror memory regions */
                if ((flags & MEMBLOCK_MIRROR) && !memblock_is_mirror(m))
                        continue;

                /* skip nomap memory unless we were asked for it explicitly */
                if (!(flags & MEMBLOCK_NOMAP) && memblock_is_nomap(m))
                        continue;

                if (!type_b) {
```

```
39                              if (out_start)
40                                      *out_start = m_start;
41                              if (out_end)
42                                      *out_end = m_end;
43                              if (out_nid)
44                                      *out_nid = m_nid;
45                              idx_a++;
46                              *idx = (u32)idx_a | (u64)idx_b << 32;
47                              return;
48                      }
```

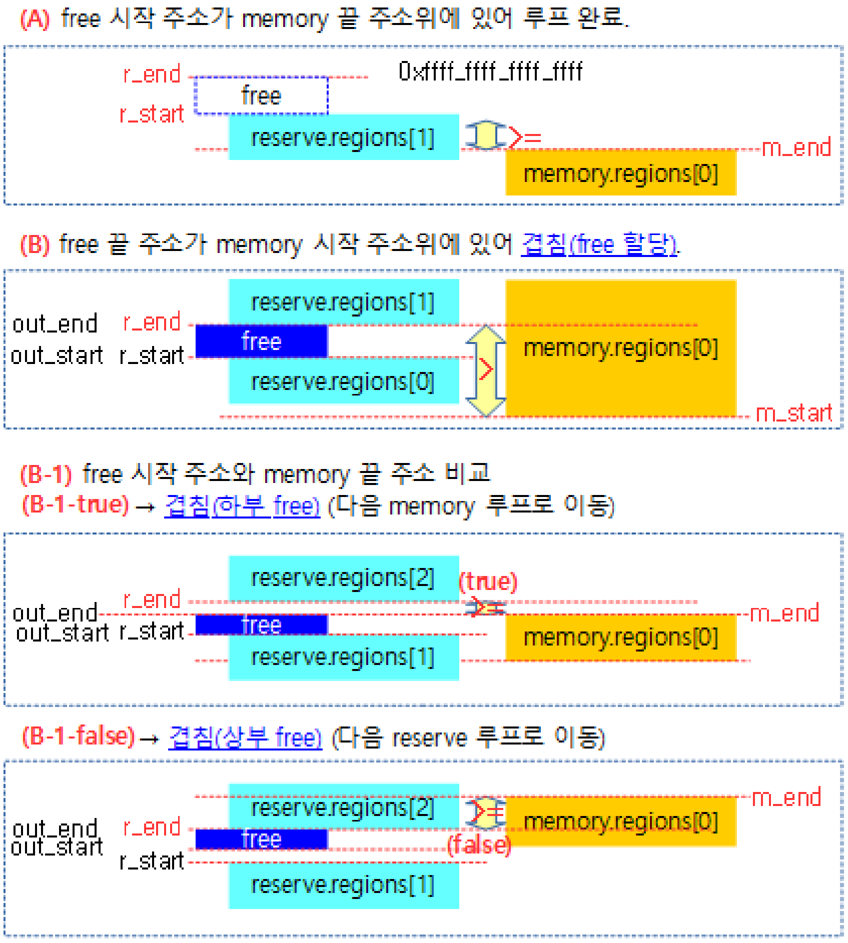
Reserved Memblock finds and returns the next free space of the requested index @idx for the zone corresponding to the node @nid value and flag @flags among the zones. In a 64-bit @idx, the index of the memory-type memblock area and the index of the reserved type memblock area are combined and the index is specified. Determine the start and end physical addresses of the free area found in the output arguments @out_start and @out_end. It also knows the node ID in the @out_nid.

- In line 8~9 of the code, divide the IDX value in half, using LSB as the counter for idx_a and MSB as the counter for idx_b.
- In lines 11~13 of code, if you use a deprecated MAX_NUMNODES as the node id argument, it will print a warning.
- In line 15~24 of the code, loop through the type_a's memblock area and skip it unless it is not the requested @nid and any node is requested.
    - When this function is called, the type_a is of type memory and the type_b is of type reserved.
    - The m_start and m_end are the start and end addresses of the memblock area of the current first-order loop index.
- In line 27~28 of code, if it is only for hotplug and movable nodes, skip it.
- In line 31~32 of the code, the mirror flag is requested, but if it is not in the mirror area, skip it.
- If there is no nomap flag in lines 35~36 and it is a nomap area, skip it.
- If the area for the type_b is not specified in code lines 38~48 (null), it determines the out_start and out_end with the area for the memblock of the current first-order loop index, increments the idx_a by 1, and successfully exits the function.

```
01                      /* scan areas before each reservation */
02                      for (; idx_b < type_b->cnt + 1; idx_b++) {
03                              struct memblock_region *r;
04                              phys_addr_t r_start;
05                              phys_addr_t r_end;
06
07                              r = &type_b->regions[idx_b];
08                              r_start = idx_b ? r[-1].base + r[-1].size : 0;
09                              r_end = idx_b < type_b->cnt ?
10                                      r->base : PHYS_ADDR_MAX;
11
12                              /*
13                               * if idx_b advanced past idx_a,
14                               * break out to advance idx_a
15                               */
16                              if (r_start >= m_end)
17                                      break;
18                              /* if the two regions intersect, we're done */
19                              if (m_start < r_end) {
20                                      if (out_start)
21                                              *out_start =
22                                                      max(m_start, r_start);
23                                      if (out_end)
24                                              *out_end = min(m_end, r_end);
25                                      if (out_nid)
26                                              *out_nid = m_nid;
27                                      /*
28                                       * The region which ends first is
29                                       * advanced for the next iteration.
30                                       */
31                                      if (m_end <= r_end)
32                                              idx_a++;
33                                      else
34                                              idx_b++;
35                                      *idx = (u32)idx_a | (u64)idx_b << 32;
36                                      return;
37                              }
38                      }
39              }
40
41              /* signal end of iteration */
42              *idx = ULLONG_MAX;
43      }
```

- In lines 2~10 of code, go around the 1nd loop by the memblock regions of type_b + 2.
  - The r region refers to the area where the second-order loop index is currently pointing.
  - r_start points to the end address of the current previous memblock if the idx_b is greater than 0, and 0 if the idx_b is 0.
  - The r_end specifies the starting address of the r area if the idx_b is less than the number registered, or the maximum address of the system.
- In lines 16~17 of code, if the reserve memblock area is outside the memory memblock area, exit the secondary loop to prepare the next memory memblock.
  - Condition A) r_start >= m_end
- This is when two areas intersect in code lines 19~26. In the out_start, the largest address is the end of the reserve area value or the beginning of the memory zone value. Then, in the out_end, the smallest address of the starting address of the upper reserve area value or the end address of the memory area value is entered.
  - Condition B) m_start < r_end
- In code lines 31~36, if the end address of the reserve area is greater than the end address of the memory area, it increments the idx_a and exits to prepare for the next memory block, and if it is not large, it increments the idx_b and continues to exit to prepare the next reserve area.
  - Condition B-1) m_end < r_end
- At line 42 of code, when it completes the first loop to the end, it can no longer process it, so it gives IDX a value of ULLONG_MAX (System Maximum Address) and exits.

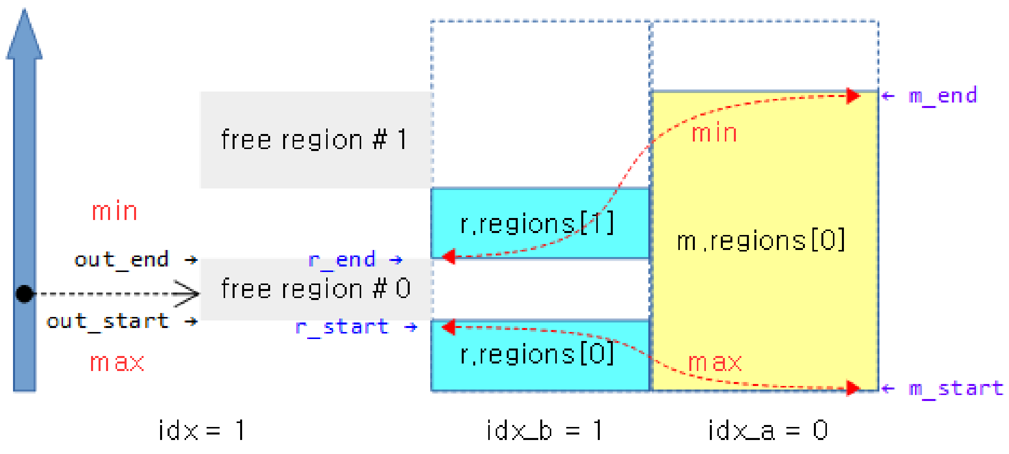
The following figure shows the cases used to compute the free region when the __next_mem_range() function loops around the free area between the reserved regions.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock12a.png)

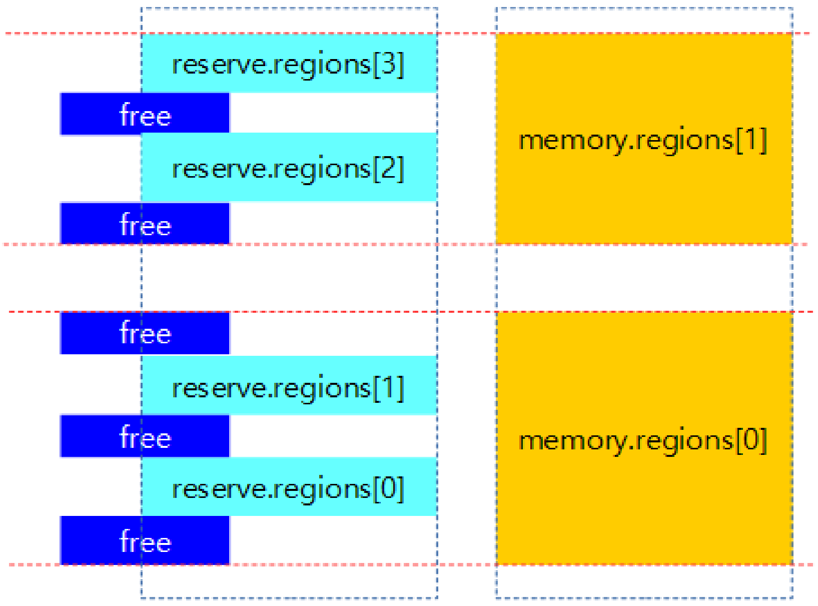
This is an example of using the __next__mem_range (1x2UUL) macro with 0 meory memblock and 1 reserve memblocks registered.

- The idx_a index in loop 1 is 0, which uses information from memory.regions[0]
- Loop 2 has a idx_b index of 1, which is using reserve.regions[1] and the area information from the previous reserve memblock.
- As a definite free zone, the r_start is specified as the end address of reserve.regions[0] and the r_end is specified as the start address of reserve.regions[1].



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock13.png)

With the following 2 meory memblocks and 4 reserve memblocks registered, use the __next__mem_range(0x1UUL) macro to express the range in which the free area is searched.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock14a.png)

for_each_free_mem_range_reverse()

mm/memblock.c

```
/**
 * for_each_free_mem_range_reverse - rev-iterate through free memblock a
reas
 * @i: u64 used as loop variable
 * @nid: node selector, %NUMA_NO_NODE for all nodes
 * @flags: pick from blocks based on memory attributes
 * @p_start: ptr to phys_addr_t for start address of the range, can be %
NULL
 * @p_end: ptr to phys_addr_t for end address of the range, can be %NULL
 * @p_nid: ptr to int for nid of the range, can be %NULL
 *
 * Walks over free (memory && !reserved) areas of memblock in reverse
 * order.  Available as soon as memblock is initialized.
 */
```

```
#define for_each_free_mem_range_reverse(i, nid, flags, p_start, p_end,
\
                                        p_nid)
\
        for_each_mem_range_rev(i, &memblock.memory, &memblock.reserved,
\
                               nid, flags, p_start, p_end, p_nid)
```

The loop is reversed, and the area of memory of the specified node ID is free memory, except for the reserved area, and this area is specified in the p_start, p_end, and p_nid arguments in that order.

for_each_mem_range_rev()

mm/memblock.c

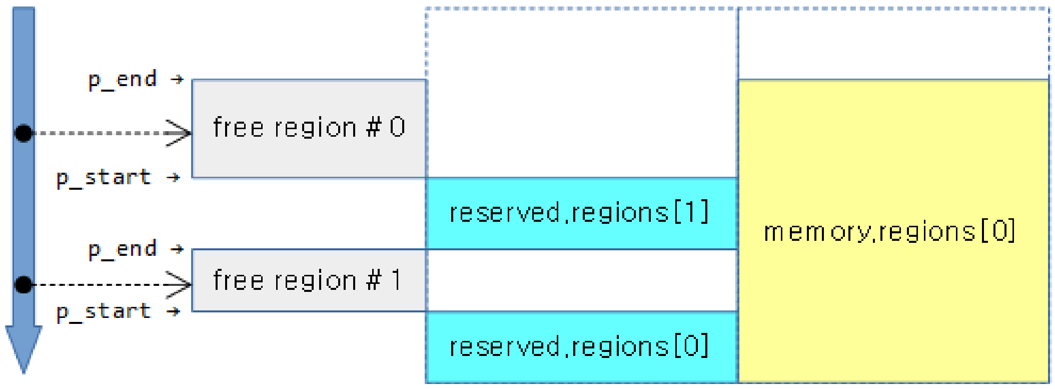
```
/**
 * for_each_mem_range_rev - reverse iterate through memblock areas from
 * type_a and not included in type_b. Or just type_a if type_b is NULL.
 * @i: u64 used as loop variable
 * @type_a: ptr to memblock_type to iterate
 * @type_b: ptr to memblock_type which excludes from the iteration
 * @nid: node selector, %NUMA_NO_NODE for all nodes
 * @flags: pick from blocks based on memory attributes
 * @p_start: ptr to phys_addr_t for start address of the range, can be %
NULL
 * @p_end: ptr to phys_addr_t for end address of the range, can be %NULL
 * @p_nid: ptr to int for nid of the range, can be %NULL
 */
```

```
#define for_each_mem_range_rev(i, type_a, type_b, nid, flags,
\
                               p_start, p_end, p_nid)
\
        for (i = (u64)ULLONG_MAX,
\
                     __next_mem_range_rev(&i, nid, flags, type_a, type_
b,\
                                          p_start, p_end, p_nid);
\
             i != (u64)ULLONG_MAX;
\
             __next_mem_range_rev(&i, nid, flags, type_a, type_b,
\
                                  p_start, p_end, p_nid))
```

Go through the loop in reverse order, and when given a region of type A to exclude a region of type B in the specified node id, specify this area in order to the p_start, p_end, and p_nid arguments. It is also possible that the B-type region is null.

- The index is a 64-bit value, divided in half, with the top 32 bits pointing to the index for the memory memblock area and the bottom 32 bits pointing to the index for the reserved memblock area.
  - At the beginning, the loop starts in reverse order, so the ~0UUL value is assigned to start the loop.
- ARM uses this top-down macro in search to allocate free memory.

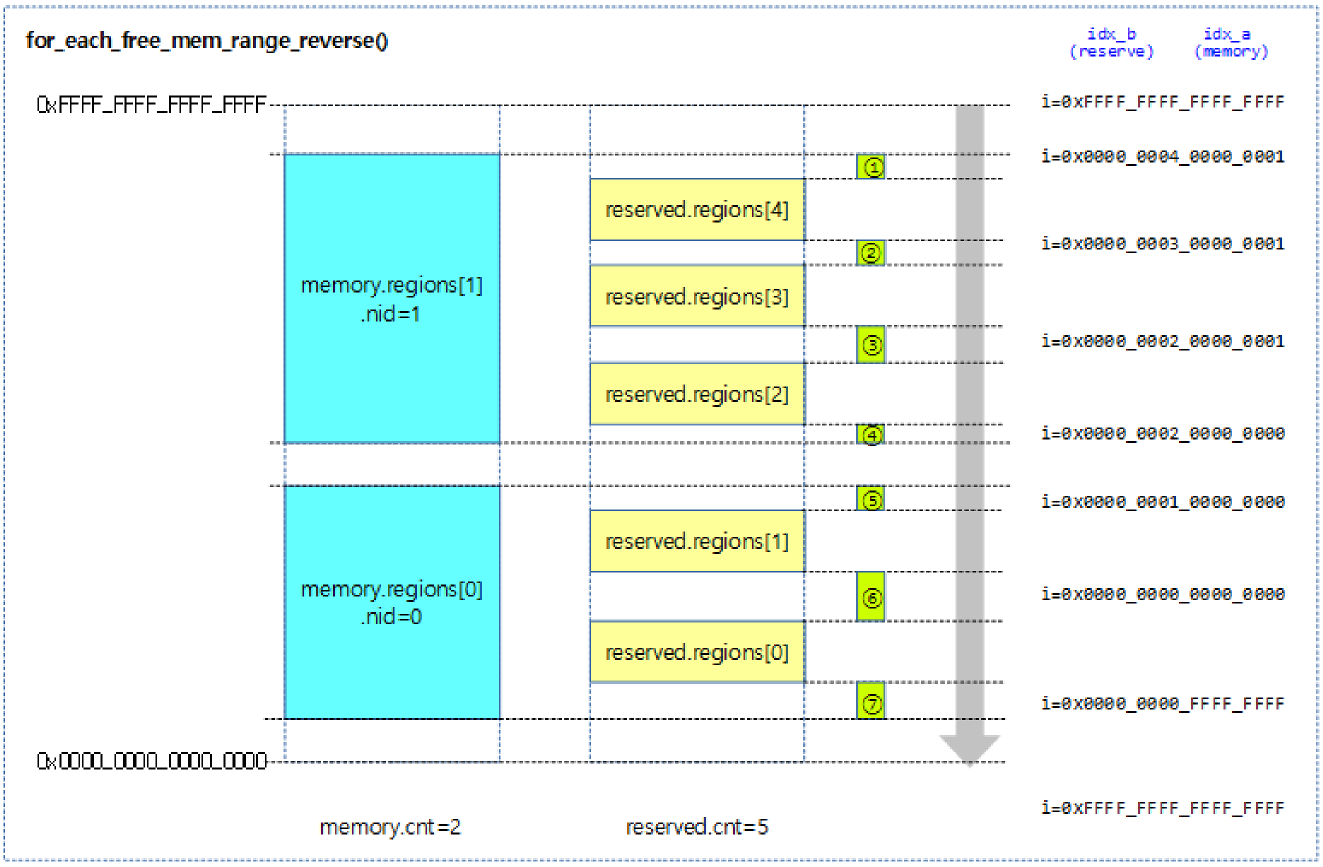
As shown below, the area except the reserved area in the memory area is the free area, and the two areas are provided in a reverse loop.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock15.png)

As shown below, when there are multiple memory regions, the tracking of the i value is expressed.

- You can see that the initial value of i starts with 0xffff_ffff_ffff_ffff and changes as shown each time it returns the free area.
- At the end of the for each loop, the value i changes back to 0xffff_ffff_ffff_ffff.



(http://jake.dothome.co.kr/wp-content/uploads/2016/01/for_each_free_mem_range_reverse-1e.png)

__next_mem_range_rev()

mm/memblock.c

If the idx value enters ~0UUL, substitute -1 from the number of memblocks of the type_a form to the idx_a, and substitute the value of the number of memblocks of the type_b form to the idx_b. If there are no more matching values and the function exits after the loop, the idx value is set back to ~0UUL. (no more data)

- The code below is the same except for reverses __next_mem_range(), so it doesn't interpret the source.

```
/**
 * __next_mem_range_rev - generic next function for for_each_*_range_rev()
 *
 * @idx: pointer to u64 loop variable
 * @nid: node selector, %NUMA_NO_NODE for all nodes
 * @flags: pick from blocks based on memory attributes
 * @type_a: pointer to memblock_type from where the range is taken
 * @type_b: pointer to memblock_type which excludes memory from being taken
 * @out_start: ptr to phys_addr_t for start address of the range, can be %NULL
 * @out_end: ptr to phys_addr_t for end address of the range, can be %NULL
 * @out_nid: ptr to int for nid of the range, can be %NULL
 *
 * Finds the next range from type_a which is not marked as unsuitable
 * in type_b.
 *
 * Reverse of __next_mem_range().
 */
```

```
void __init_memblock __next_mem_range_rev(u64 *idx, int nid,
                                          enum memblock_flags flags,
                                          struct memblock_type *type_a,
                                          struct memblock_type *type_b,
                                          phys_addr_t *out_start,
                                          phys_addr_t *out_end, int *out_nid)
{
        int idx_a = *idx & 0xffffffff;
        int idx_b = *idx >> 32;

        if (WARN_ONCE(nid == MAX_NUMNODES, "Usage of MAX_NUMNODES is deprecated. Use NUMA_NO_NODE instead\n"))
                nid = NUMA_NO_NODE;

        if (*idx == (u64)ULLONG_MAX) {
                idx_a = type_a->cnt - 1;
                if (type_b != NULL)
                        idx_b = type_b->cnt;
                else
                        idx_b = 0;
        }

        for (; idx_a >= 0; idx_a--) {
                struct memblock_region *m = &type_a->regions[idx_a];

                phys_addr_t m_start = m->base;
                phys_addr_t m_end = m->base + m->size;
                int m_nid = memblock_get_region_node(m);

                /* only memory regions are associated with nodes, check it */
                if (nid != NUMA_NO_NODE && nid != m_nid)
                        continue;

                /* skip hotpluggable memory regions if needed */
                if (movable_node_is_enabled() && memblock_is_hotpluggable(m))
                        continue;

                /* if we want mirror memory skip non-mirror memory regions */
                if ((flags & MEMBLOCK_MIRROR) && !memblock_is_mirror(m))
                        continue;

                /* skip nomap memory unless we were asked for it explicitly */
                if (!(flags & MEMBLOCK_NOMAP) && memblock_is_nomap(m))
                        continue;

                if (!type_b) {
                        if (out_start)
                                *out_start = m_start;
                        if (out_end)
                                *out_end = m_end;
                        if (out_nid)
                                *out_nid = m_nid;
                        idx_a--;
                        *idx = (u32)idx_a | (u64)idx_b << 32;
                        return;
                }
```

```
                /* scan areas before each reservation */
                for (; idx_b >= 0; idx_b--) {
                        struct memblock_region *r;
                        phys_addr_t r_start;
                        phys_addr_t r_end;

                        r = &type_b->regions[idx_b];
                        r_start = idx_b ? r[-1].base + r[-1].size : 0;
                        r_end = idx_b < type_b->cnt ?
                                r->base : PHYS_ADDR_MAX;
                        /*
                         * if idx_b advanced past idx_a,
                         * break out to advance idx_a
                         */

                        if (r_end <= m_start)
                                break;
                        /* if the two regions intersect, we're done */
                        if (m_end > r_start) {
                                if (out_start)
                                        *out_start = max(m_start, r_start);
                                if (out_end)
                                        *out_end = min(m_end, r_end);
                                if (out_nid)
                                        *out_nid = m_nid;
                                if (m_start >= r_start)
                                        idx_a--;
                                else
                                        idx_b--;
                                *idx = (u32)idx_a | (u64)idx_b << 32;
                                return;
                        }
                }
        }
        /* signal end of iteration */
        *idx = ULLONG_MAX;
}
```

Other Memblock APIs

- memblock_type_name()
  - Learns the type name (string) of the memblock.
- memblock_cap_size()
  - If the zone exceeds the maximum address of the system, it will be truncated as much as it overflows.
- memblock_addrs_overlap()
  - Check if the zones overlap each other.
- memblock_overlaps_region()
  - Checks if the registered memblock areas overlap with the given area as an argument and returns the index of the overlapping area, or -1
- get_allocated_memblock_reserved_regions_info()

  - If a region of type reserved is not the address assigned at initialization, but is reassigned by a new buddy/slab memory allocator, it returns the start address of the allocated area, or 0. This function is called when the bootmem area is terminated via the free_low_memory_core_early() function and switched to the buddy allocator.
- get_allocated_memblock_memory_regions_info()
  - If a region of type memory is not the address assigned at initialization, but is reassigned by a new buddy/slab memory allocator, it returns the start address of the allocated area, or 0. This function is called when the bootmem area is terminated via the free_low_memory_core_early() function and switched to the buddy allocator.
- memblock_double_array()
  - Expands the array of memblock_region of a given type to twice the size.
- memblock_free()
  - Delete the given area in the reserved area.
- memblock_set_node()
  - Set the nodeIDs of all memblocks corresponding to the given area. When the area falls in the middle of the memblock, it separates it and sets the nodeid.
- memblock_phys_mem_size()
  - Full size of memblock registered in the memory area
- memblock_mem_size()
  - Finds the total number of registered memory memblock pages up to a given limit_pfn.
- memblock_start_of_DRAM()
  - The start address of the first memblock registered in the memory zone (DRAM start address)
- memblock_end_of_DRAM()
  - End address of the last memblock registered in the memory area (DRAM end address)
- memblock_enforce_memory_limit()
  - What if the memory area exceeds the limit?
- memblock_search()
  - From a memblock of a given type to a given address, we use a binary search algorithm to find the memblock and get its index value.
- memblock_is_reserved()
  - Determine if the reserved area contains an address by searching the reserved area.
- memblock_is_memory()
  - Determine if the memory area contains an address by searching the memory area.
- memblock_search_pfn_nid()
  - After finding a memblock with a binary search algorithm at a given address in the memory domain, it finds the start and end pages of the found area.
- memblock_is_region_reserved()
  - Check to see if the given area overlaps with the reserved memblock area.
- memblock_trim_memory()
  - All memory memblocks are aligned to a specified align byte. If the size is smaller than the align size, delete it.
- memblock_set_current_limit()
  - Set the limit of the memblock.

- memblock_dump()
  - Prints information about memblocks of a given type.
- memblock_dump_all()
  - Prints information about memory and reserved memblocks.
- memblock_allow_resize()
  - Set global variable memblock_can_resize=1 (allows resize)
- early_memblock()
  - early_param() and if it receives a "debug" string as an argument, set the global variable memblock_debug=1 to make the memblock_dbg() function work.
- memblock_debug_show()
  - If the DEBUG_FS is running, it can output debug.

Debug Output

If you used the CONFIG_DEBUG_FS kernel options, you can check the memblock registration status as follows:

e.g. rpi2 – ARM32

```
# cat /sys/kernel/debug/memblock/memory
   0: 0x00000000..0x3affffff

# cat /sys/kernel/debug/memblock/reserved
   0: 0x00004000..0x00007fff   <- 페이지 테이블
   1: 0x00008240..0x0098c1ab   <- 커널
   2: 0x2fffbf00..0x2fffff08
   3: 0x39e9e000..0x39f95fff
   4: 0x39f989c4..0x3a7fefff
   5: 0x3a7ff540..0x3a7ff583
   6: 0x3a7ff5c0..0x3a7ff603
   7: 0x3a7ff640..0x3a7ff6b7
   8: 0x3a7ff6c0..0x3a7ff6cf
   9: 0x3a7ff700..0x3a7ff70f
  10: 0x3a7ff740..0x3a7ff743
  11: 0x3a7ff780..0x3a7ff916
  12: 0x3a7ff940..0x3a7ffad6
  13: 0x3a7ffb00..0x3a7ffc96
  14: 0x3a7ffc9c..0x3a7ffd14
  15: 0x3a7ffd18..0x3a7ffd60
  16: 0x3a7ffd64..0x3a7ffd7e
  17: 0x3a7ffd80..0x3a7ffd9b
  18: 0x3a7ffda4..0x3a7ffdbe
  19: 0x3a7ffdc0..0x3a7ffe37
  20: 0x3a7ffe40..0x3a7ffe43
  21: 0x3a7ffe48..0x3affffff
```

e.g. rock960 – ARM64

```
$ /sys/kernel/debug/memblock$ cat memory
   0: 0x0000000000200000..0x00000000f7ffffff

$ /sys/kernel/debug/memblock$ cat reserved
   0: 0x0000000002080000..0x00000000033b5fff  <- 커널
   1: 0x00000000ef400000..0x00000000f5dfffff
   2: 0x00000000f5eef000..0x00000000f5f01fff
   3: 0x00000000f6000000..0x00000000f7bfffff
   4: 0x00000000f7df4000..0x00000000f7df4fff
   5: 0x00000000f7df5e00..0x00000000f7df5fff
   6: 0x00000000f7e74000..0x00000000f7f61fff
   7: 0x00000000f7f62600..0x00000000f7f6265f
   8: 0x00000000f7f62680..0x00000000f7f626df
   9: 0x00000000f7f62700..0x00000000f7f6282f
  10: 0x00000000f7f62840..0x00000000f7f62857
  11: 0x00000000f7f62880..0x00000000f7f62887
  12: 0x00000000f7f648c0..0x00000000f7f6492b
  13: 0x00000000f7f64940..0x00000000f7f649ab
  14: 0x00000000f7f649c0..0x00000000f7f64a2b
  15: 0x00000000f7f64a40..0x00000000f7f64a47
  16: 0x00000000f7f64a64..0x00000000f7f64aea
  17: 0x00000000f7f64aec..0x00000000f7f64b1a
  18: 0x00000000f7f64b1c..0x00000000f7f64b4a
  19: 0x00000000f7f64b4c..0x00000000f7f64b7a
  20: 0x00000000f7f64b7c..0x00000000f7f64baa
  21: 0x00000000f7f64bac..0x00000000f7fcdff7
  22: 0x00000000f7fce000..0x00000000f7ffffff
```

consultation

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

6 thoughts to "Memblock – (2)"

KIM SAK GAT

2016-03-26 19:59 (http://jake.dothome.co.kr/memblock-2/#comment-170)

In for_each_mem_pfn_range(), region[0], nid=1 should become region[2], nid=1.

RESPONSE (/MEMBLOCK-2/?REPLYTOCOM=170#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2016-03-27 13:17 (http://jake.dothome.co.kr/memblock-2/#comment-172)

Fixed! I appreciate it.

RESPONSE (/MEMBLOCK-2/?REPLYTOCOM=172#RESPOND)

A WANDERER PASSING BY.
2022-03-16 23:20 (http://jake.dothome.co.kr/memblock-2/#comment-306402)

Hello. I would like to know if the leftmost part of this picture should be changed to reserved.cnt=>memory.cnt.
http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock_find_range_top_down-1b.png (http://jake.dothome.co.kr/wp-content/uploads/2016/01/memblock_find_range_top_down-1b.png)

RESPONSE (/MEMBLOCK-2/?REPLYTOCOM=306402#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2022-03-17 11:07 (http://jake.dothome.co.kr/memblock-2/#comment-306404)

Yes, I made a lot of mistakes. Fixed. I appreciate it^^

RESPONSE (/MEMBLOCK-2/?REPLYTOCOM=306404#RESPOND)

PASSING WANDERER
2022-03-17 18:08 (http://jake.dothome.co.kr/memblock-2/#comment-306407)

What do you mean? I'm always looking at it well. There is a Munc blog in Korea, so it is very good to study.

RESPONSE (/MEMBLOCK-2/?REPLYTOCOM=306407#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2022-03-18 19:52 (http://jake.dothome.co.kr/memblock-2/#comment-306411)

Thank you for taking a good look. Have a nice day. ^^

RESPONSE (/MEMBLOCK-2/?REPLYTOCOM=306411#RESPOND)

LEAVE A COMMENT

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

Comments

이름 *

이메일 *

웹사이트

댓글 작성

Memblock – (1) (http://jake.dothome.co.kr/memblock-1/)

compressed/head.S – restart: (http://jake.dothome.co.kr/restart/)