Memory Model -3- (Sparse Memory)

2016-03-30 (http://jake.dothome.co.kr/sparsemem/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.15>

The Sparse memory model manages the following data through a structure mem_section in sectional memory units:

- mem_map
- usemap (pageblock)

section

In the Sparse memory model, a section is the smallest memory size unit that manages the online/offline (hotplug memory) of memory. The entire memory is divided into sections, which vary from architecture to architecture.

- In general, the section size is tens of MB~several GB.
  - In the case of arm64, 1G was used as the default value, but it was recently changed to 128M.
    - Note: arm64/sparsemem: reduce SECTION_SIZE_BITS (https://github.com/torvalds/linux/commit/f0b13ee23241846f6f6cd0d119a8ac8059416ec4#diff-9590f22a80ad3a29d9c97bab548e481c48ac04e5ca5441c9360c69458829634c) (2021, v5.12-rc1)
  - Note: The text diagram shows an example of using a section size of 1G.
  - Caution: Differs from the section (2M) terminology used in the mapping table.

Section arrangement 2 ways to manage

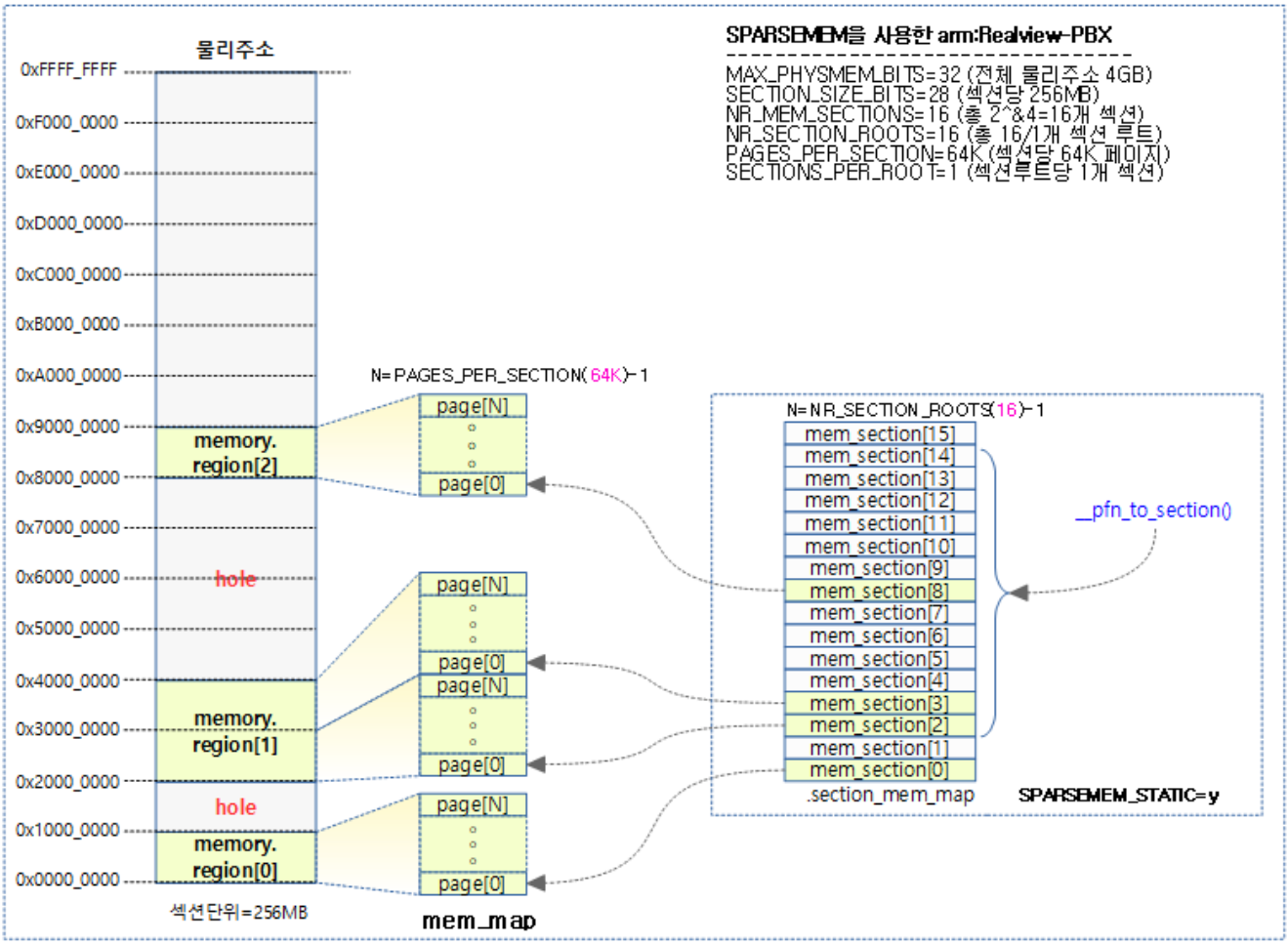
In this section, you'll learn about two ways to manage your mem_map:

- static
  - This is how to determine and use a first-stage mem_section[] array at compile time.
  - It is usually used on a 32-bit system with a small number of sections.
- Extream
  - This avoids wasting memory by allocating a mem_section mem_section array at runtime, and second-stage allocating an array of [] whenever needed.
  - It is mainly used on a 64-bit system with a large number of sections.

CONFIG_SPARSEMEM_STATIC

We use the example of a Realview-PBX board with Sparse Memory on a 32-bit ARM, with a size of 256 MB per section.

- Realview-PBX
  - 3 Memories
    - 256MB @ 0x00000000 -> PAGE_OFFSET
    - 512MB @ 0x20000000 -> PAGE_OFFSET + 0x10000000
    - 256MB @ 0x80000000 -> PAGE_OFFSET + 0x30000000
  - MAX_PHYSMEM_BITS=32 (4G memory size)
  - SECTION_SIZE_BITS=28 (256MB section size)
  - PFN_SECTION_SHIFT=(SECTION_SIZE_BITS – PAGE_SHIFT)=16
  - SECTIONS_PER_ROOT=1
  - SECTIONS_SHIFT=(MAX_PHYSMEM_BITS – SECTION_SIZE_BITS)=4
  - NR_MEM_SECTIONS=2^SECTIONS_SHIFT=16
  - PAGES_PER_SECTION=2^PFN_SECTION_SHIFT=64K
  - PAGE_SECTION_MASK=(~(PAGES_PER_SECTION-1))=0xffff_0000

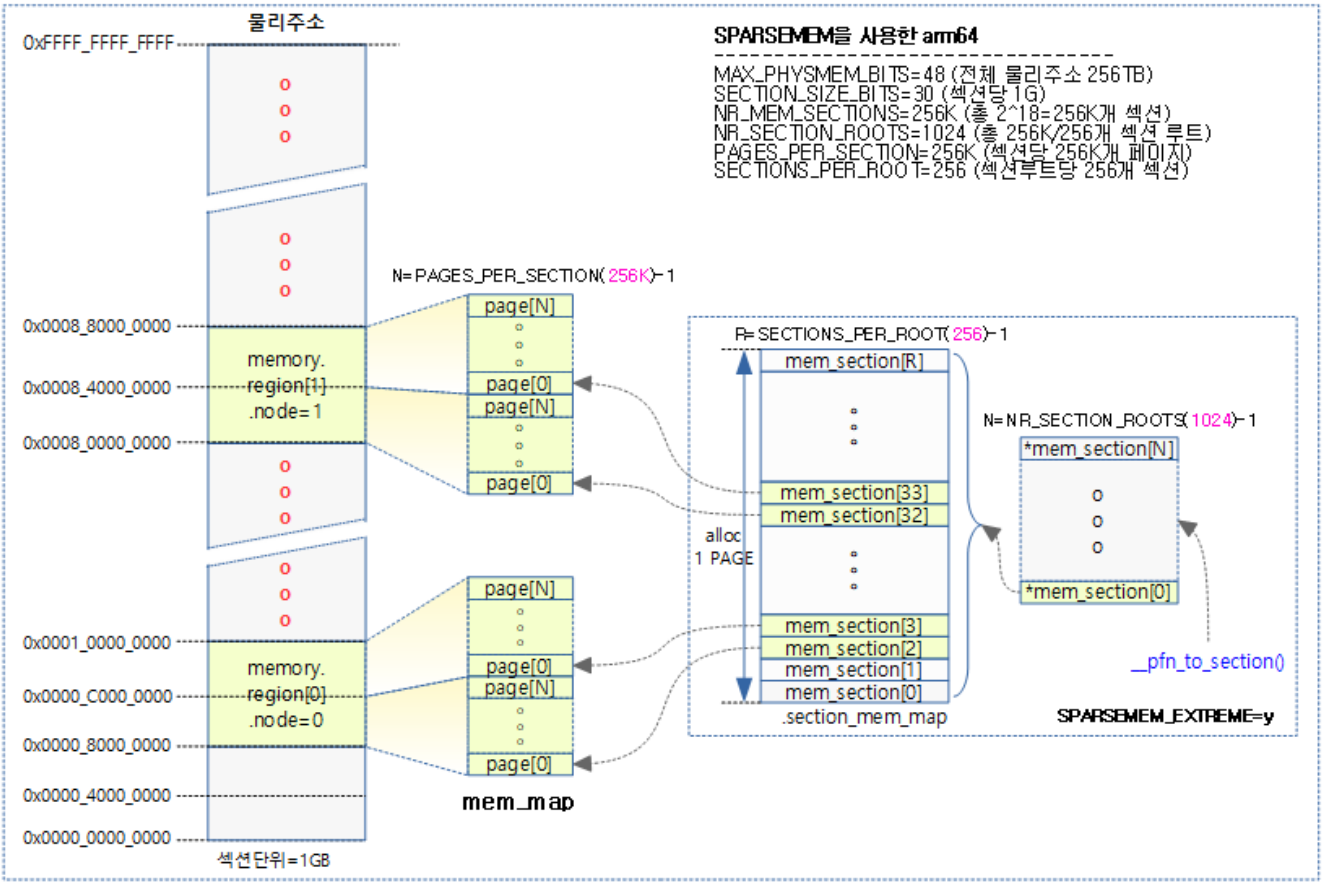


(http://jake.dothome.co.kr/wp-content/uploads/2016/03/mm-7c.png)

CONFIG_SPARSEMEM_EXTREME

As an example of using Sparse Memory on a 64-bit ARM, we use a case that consists of a 1GB size.

- arm64
  - 2 Memories
    - 2GB @ 0x0_8000_0000
    - 2GB @ 0x8_0000_0000
- MAX_PHYSMEM_BITS=48 (256 TB memory size)
- SECTION_SIZE_BITS=30 (1 GB section size)
- PFN_SECTION_SHIFT=(SECTION_SIZE_BITS – PAGE_SHIFT)=18
- SECTIONS_PER_ROOT=(PAGE_SIZE / sizeof (struct mem_section))=256
- SECTIONS_SHIFT=(MAX_PHYSMEM_BITS – SECTION_SIZE_BITS)=18
- NR_MEM_SECTIONS=2^SECTIONS_SHIFT=256K
- PAGES_PER_SECTION=2^PFN_SECTION_SHIFT=256K
- PAGE_SECTION_MASK=(~(PAGES_PER_SECTION-1))=0xffff_ffff_fffc_0000

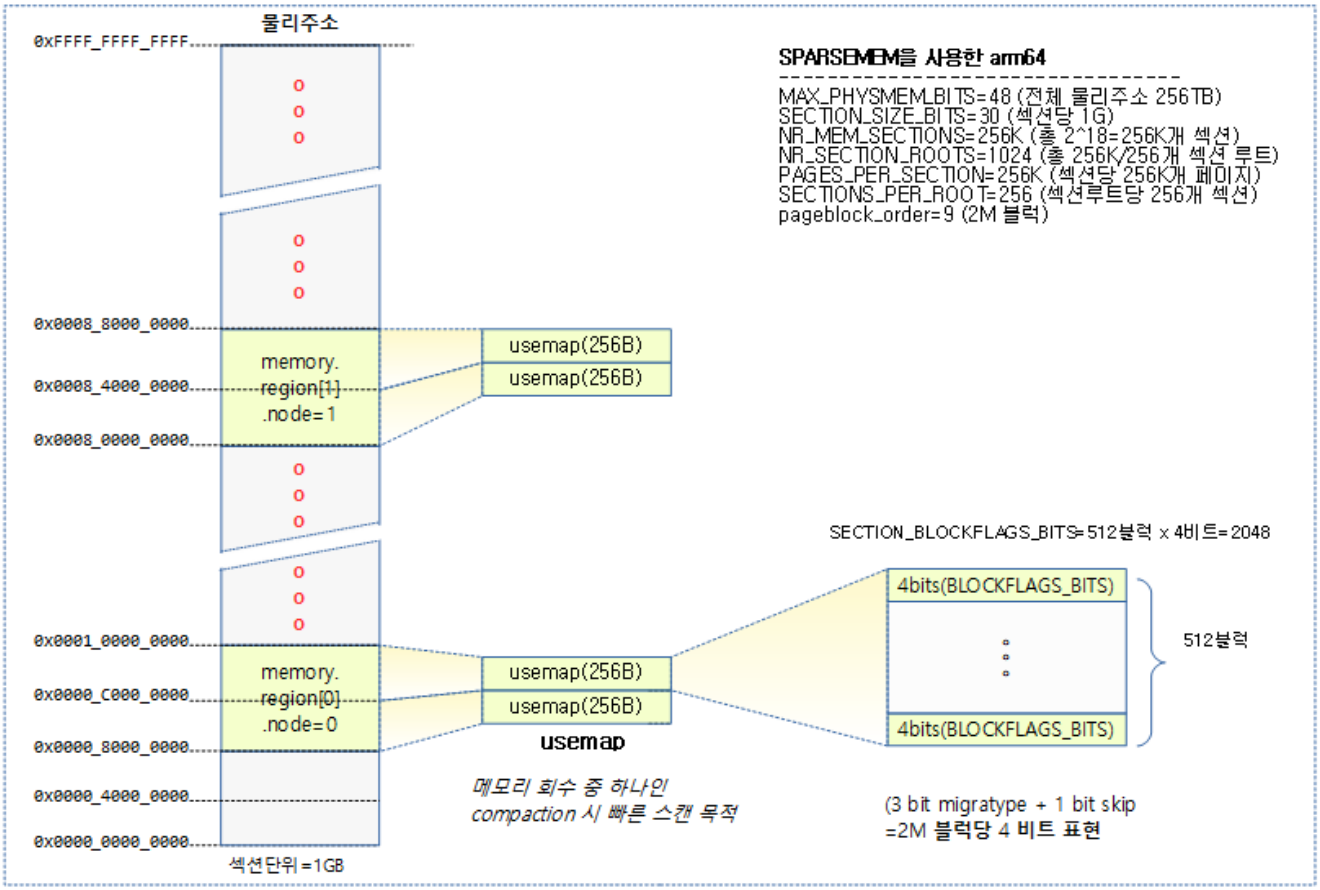


(http://jake.dothome.co.kr/wp-content/uploads/2016/03/mm-9b.png)

usemap

The usemap consists of 4 bits per page block, which is used to scan the entire memory in Compaction, one of the memory reclamation mechanisms.

- In compaction, it is used in block units (arm64 default=2M) for quick scanning.
- The 4 bits used per page block consist of 3 bits of mobility (migratype) properties and 1 bit of skip bits.



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

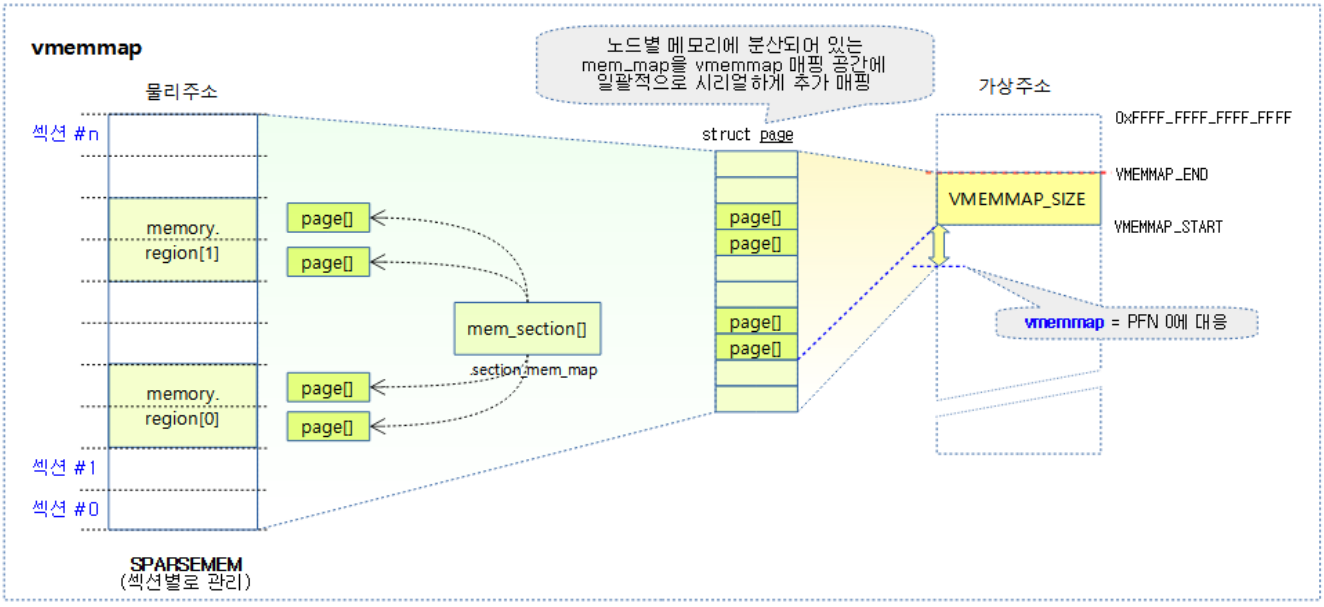
VMEMMAP

It is not used on 32bit arm. It can be operated as fast as the Flat Memory model on some 64-bit architectures, including arm64, which has CONFIG_SPARSEMEM_VMEMMAP kernel options available.

- On 64-bit systems, the vmemmap mapping space is set up separately to map mem_map consisting of section_mem_map to that area.
- If you use vmemmap, you can access the pages serially with a descriptor that is distributed in memory by node, which allows you to quickly get the performance of the page_to_pfn() or pfn_to_page() functions.

The following figure shows an example of vmemmap pointing to the location of the page structure corresponding to the PFN #0 position below the VMEMMAP_START.

- VMEMMAP_START points to the page structure for the PFN where the actual DRAM is located,
- vmemmap points to the page structure for PFN#0.
    - Note: If the DRAM start address is not 0, vmemmap will be below VMEMMAP_START. The address of the vmemmap below is not the actual mem_map mapped space, so if the kernel accesses this space, it will fail and the system will freeze.)

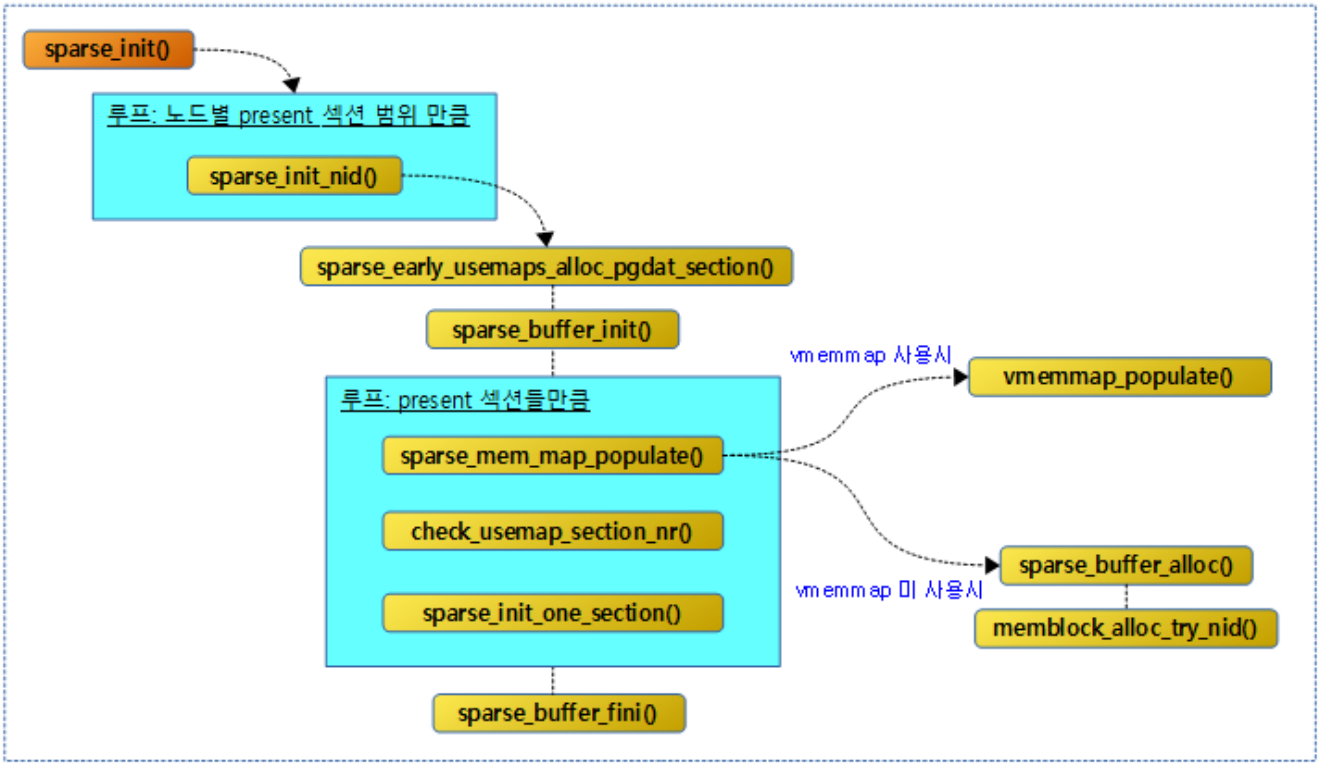


(http://jake.dothome.co.kr/wp-content/uploads/2016/03/vmemmap-1c.png)

Initialize the Sparse Memory Model

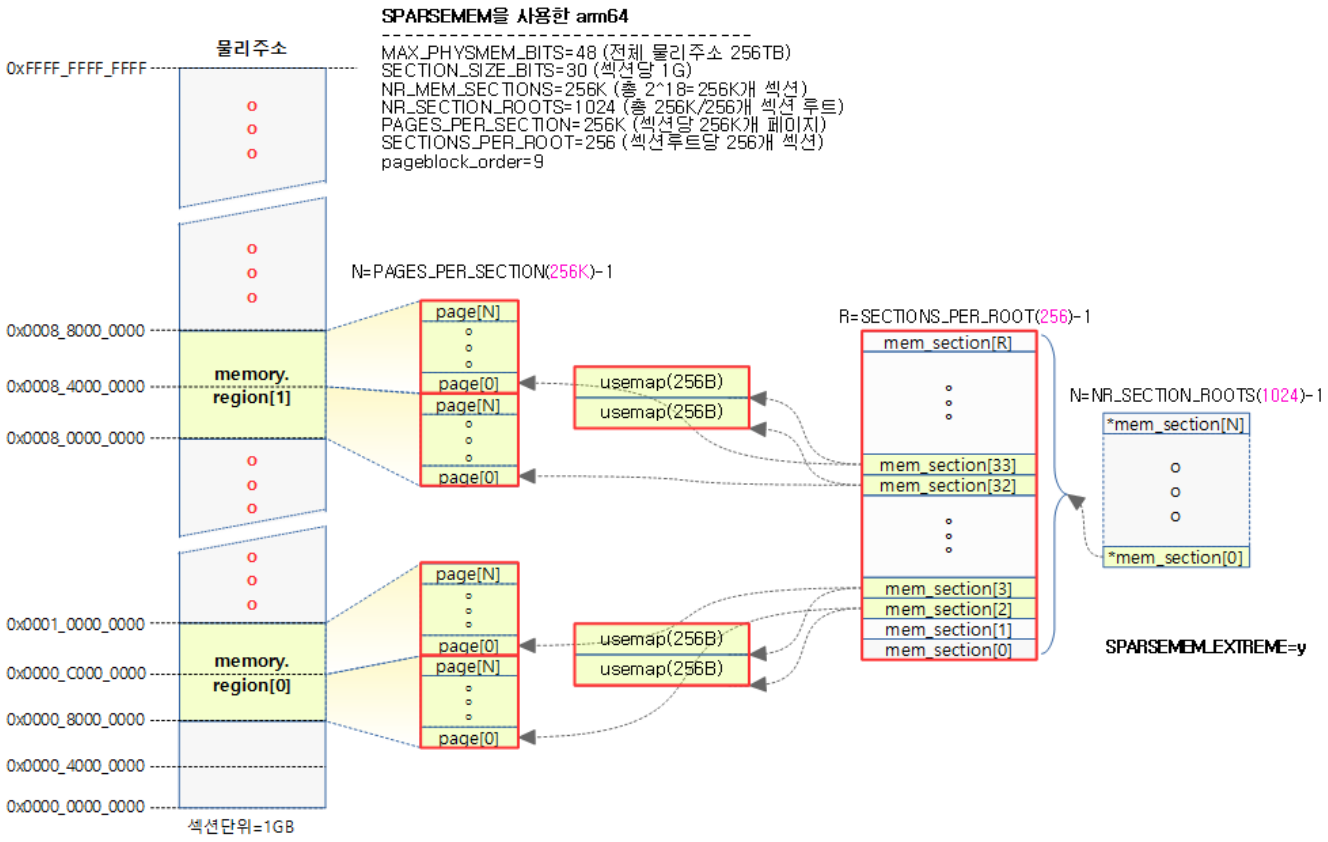
The following flowchart is the logic for initializing Sparse memory, and it does the following:

- Assign a usemap for each node and connect it to a temporary usemap_map.
- With the CONFIG_SPARSEMEM_VMEMMAP kernel option, the mem_map is mapped to vmemmap.
- If you don't use the CONFIG_SPARSEMEM_VMEMMAP kernel option, assign each node a mem_map of the active sections at the same time, if possible. (Similar to the CONFIG_SPARSEMEM_ALLOC_MEM_MAP_TOGETHER kernel options for previous versions of the x86 kernel.)



(http://jake.dothome.co.kr/wp-content/uploads/2016/03/sparse_init-6.png)

The following image shows the mem_map, usemap, and mem_section created by sparse_init().



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

sparse_init()

mm/sparse.c

```
1  /*
2   * Allocate the accumulated non-linear sections, allocate a mem_map
3   * for each and record the physical to section mapping.
4   */
```

```
01 void __init sparse_init(void)
02 {
03         unsigned long pnum_end, pnum_begin, map_count = 1;
04         int nid_begin;
05
06         memblocks_present();
07
08         pnum_begin = first_present_section_nr();
09         nid_begin = sparse_early_nid(__nr_to_section(pnum_begin));
10
11         /* Setup pageblock_order for HUGETLB_PAGE_SIZE_VARIABLE */
12         set_pageblock_order();
13
14         for_each_present_section_nr(pnum_begin + 1, pnum_end) {
15                 int nid = sparse_early_nid(__nr_to_section(pnum_end));
16
17                 if (nid == nid_begin) {
18                         map_count++;
19                         continue;
20                 }
21                 /* Init node with sections in range [pnum_begin, pnum_en
   d) */
22                 sparse_init_nid(nid_begin, pnum_begin, pnum_end, map_cou
   nt);
23                 nid_begin = nid;
24                 pnum_begin = pnum_end;
25                 map_count = 1;
26         }
27         /* cover the last node */
28         sparse_init_nid(nid_begin, pnum_begin, pnum_end, map_count);
29         vmemmap_populate_print_last();
30 }
```

Learn how to use the Sparse physical memory model, initialize a mem_map that is set up with page descriptors to allocate and manage them in sections.
In order to manage distributed memory on a section-by-section basis, each section is assigned a mem_map and a usemap so that it can be managed. The usemap determines the unit of page blocks, and then manages the mobility characteristics with a bitmap of 4 bits per page block.

- In line 6 of the code, we mem_section connect all the memories registered in the memblock and activate them section by section.
- On lines 8~9 of the code, get the starting section number and the NID for it.
- In line 12 of code, we first set up a global variable pageblock_order to record and manage the mobility characteristic of 4 bits per page block size for total memory.
    - The default setting for ARM64 systems is 9 for pageblock_order.
- In line 14~20 of the code, it goes through the present section, and if it is the same node, it increases the map_count so that the mem_map and so on are allocated from the same node at the same time.
    - When you are traversing a present section, the pnum_end contains the number of the present section you are traversing.
- If the node is changed from code lines 22~25, it initializes the sections in the corresponding present scope.

- In line 28 of the code, we also initialize the sections in that present scope for the last remaining node that was not processed in the loop above.
- In line 29 of the code, it is not implemented in arm and arm64, so there is no message to output.

Memory Activation

memblocks_present()

mm/sparse.c

```
1  /*
2   * Mark all memblocks as present using memory_present().
3   * This is a convenience function that is useful to mark all of the syst
   ems
4   * memory as present during initialization.
5   */
```

```
1  static void __init memblocks_present(void)
2  {
3          unsigned long start, end;
4          int i, nid;
5
6          for_each_mem_pfn_range(i, MAX_NUMNODES, &start, &end, &nid)
7                  memory_present(nid, start, end);
8  }
```

It traverses the memories registered in the memblock, connects them to the mem_section, and activates them section by section.

memory_present()

mm/sparse.c

```
1  /* Record a memory area against a node. */
```

```
01 static void __init memory_present(int nid, unsigned long start, unsigned
   long end)
02 {
03         unsigned long pfn;
04
05 #ifdef CONFIG_SPARSEMEM_EXTREME
06         if (unlikely(!mem_section)) {
07                 unsigned long size, align;
08
09                 size = sizeof(struct mem_section*) * NR_SECTION_ROOTS;
10                 align = 1 << (INTERNODE_CACHE_SHIFT);
11                 mem_section = memblock_alloc(size, align);
12                 if (!mem_section)
13                         panic("%s: Failed to allocate %lu bytes align=0
   x%lx\n",
14                               __func__, size, align);
15         }
16 #endif
17
18         start &= PAGE_SECTION_MASK;
19         mminit_validate_memmodel_limits(&start, &end);
20         for (pfn = start; pfn < end; pfn += PAGES_PER_SECTION) {
21                 unsigned long section = pfn_to_section_nr(pfn);
22                 struct mem_section *ms;
```

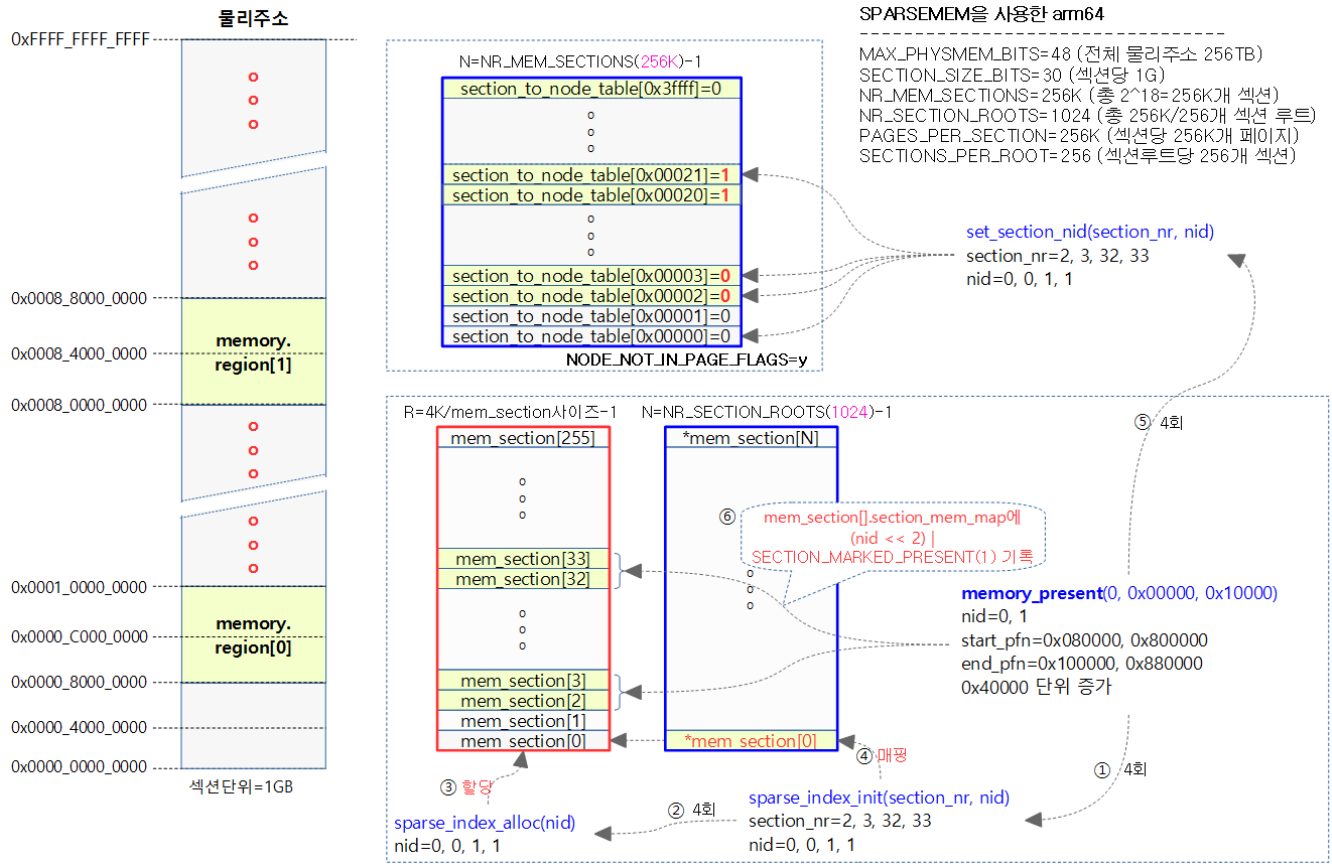
```
23
24                  sparse_index_init(section, nid);
25                  set_section_nid(section, nid);
26
27                  ms = __nr_to_section(section);
28                  if (!ms->section_mem_map) {
29                          ms->section_mem_map = sparse_encode_early_nid(ni
    d) |
30                                                          SECTION_IS_ONLIN
    E;
31                          __section_mark_present(ms, section);
32                  }
33          }
34  }
```

The memory_present() function only works if you use the CONFIG_HAVE_MEMORY_PRESENT kernel options, and records the node ID in each section, but you'll see the code for more details.

- If you use the CONFIG_SPARSEMEM_EXTREME kernel option in line 6~15 of code, it will create a mem_section[] array if the first mem_section is not initialized.
  - arm64 e.g. 4K page, sizeof(struct mem_section)=16
    - NR_SECTION_ROOTS=1024
- After converting the starting pfn of the range requested in line 18~19 to an address ordered down by sections, it verifies that the pfn in the range is actually within the specified physical memory range, and if it exceeds it, it forcibly restricts the address.
- Starting from code lines 20~21, increase by section from pfn and get the section number from this value.
- This only works if you use the CONFIG_SPARSEMEM_EXTREME kernel option in line 24 of code, which dynamically allocates the mem_section[] for that section. If you use the CONFIG_SPARSEMEM_STATIC kernel options, you don't need to do anything because you already have a static array in place.
- If a NODE_NOT_IN_PAGE_FLAGS is defined in line 25 of code, point that section to its node ID as an index in a separate global section_to_node_table[] array.
  - NODE_NOT_IN_PAGE_FLAGS kernel option is used in 32-bit architectures where there are not enough bits to store the node number in the flags field of the page structure.
- In lines 27~32 of code, set the node id and the online and present flags to the section_mem_map in the mem_section struct of that section.

The figure below shows how an array of mem_section[] is allocated in the process of the memory_present() function being called.

- The red box indicates that the memory allocation is dynamically received by the sparse_index_alloc() function.
- A blue box means that the array is staticly allocated at compile time.



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

mminit_validate_memmodel_limits()

mm/sparse.c

```
1  /* Validate the physical addressing limitations of the model */

01 void __meminit mminit_validate_memmodel_limits(unsigned long *start_pfn,
02                                                unsigned long *end_pfn)
03 {
04         unsigned long max_sparsemem_pfn = 1UL << (MAX_PHYSMEM_BITS-PAGE_
   SHIFT);
05
06         /*
07          * Sanity checks - do not allow an architecture to pass
08          * in larger pfns than the maximum scope of sparsemem:
09          */
10         if (*start_pfn > max_sparsemem_pfn) {
11                 mminit_dprintk(MMINIT_WARNING, "pfnvalidation",
12                         "Start of range %lu -> %lu exceeds SPARSEMEM max
   %lu\n",
13                         *start_pfn, *end_pfn, max_sparsemem_pfn);
14                 WARN_ON_ONCE(1);
15                 *start_pfn = max_sparsemem_pfn;
16                 *end_pfn = max_sparsemem_pfn;
17         } else if (*end_pfn > max_sparsemem_pfn) {
18                 mminit_dprintk(MMINIT_WARNING, "pfnvalidation",
19                         "End of range %lu -> %lu exceeds SPARSEMEM max %
   lu\n",
20                         *start_pfn, *end_pfn, max_sparsemem_pfn);
21                 WARN_ON_ONCE(1);
22                 *end_pfn = max_sparsemem_pfn;
23         }
24 }
```

For the start pfn used as an argument, limit the ending pfn value to not exceed the physical memory address maximum pfn value.

- Starting from code lines 10~16, it will output a warning if the pfn is greater than max_sparsemem_pfn, and set the max_sparsemem_pfn for the start_pfn and end_pfn.
- In line 17~23 of the code, if the ending pfn is greater than max_sparsemem_pfn, it will print a warning and set the max_sparsemem_pfn in the end_pfn.

Initialize Section Index

sparse_index_init()

mm/sparse.c

```
01 #ifdef CONFIG_SPARSEMEM_EXTREME
02 static int __meminit sparse_index_init(unsigned long section_nr, int ni
   d)
03 {
04         unsigned long root = SECTION_NR_TO_ROOT(section_nr);
05         struct mem_section *section;
06
07         /*
08          * An existing section is possible in the sub-section hotplug
09          * case. First hot-add instantiates, follow-on hot-add reuses
10          * the existing section.
11          *
12          * The mem_hotplug_lock resolves the apparent race below.
13          */
14         if (mem_section[root])
15                 return 0;
16
17         section = sparse_index_alloc(nid);
18         if (!section)
19                 return -ENOMEM;
20
21         mem_section[root] = section;
22
23         return 0;
24 }
25 #endif
```

If you use the CONFIG_SPARSEMEM_EXTREME kernel options, you will be assigned a dynamically mem_section table and configure it.

- Get the root number from the section number on line 4 of the code.
- In lines 14~15 of the code, if a value exists in the root section of that root index, it exits the function because the mem_section[] table has already been configured.
- In line 17~19 of the code, the node is assigned a section table for step 2 and configured. For hot-plug memory, each mem_section[] table must be located on its node.
- On line 21 of the code, set the start address of the newly assigned mem_section step 1 table in the first-step mem_section[] pointer array, which corresponds to the root number.
- Normal result on line 23 of code returns 0.

sparse_index_alloc()

mm/sparse.c

```
01 #ifdef CONFIG_SPARSEMEM_EXTREME
02 static noinline struct mem_section __ref *sparse_index_alloc(int nid)
03 {
04         struct mem_section *section = NULL;
05         unsigned long array_size = SECTIONS_PER_ROOT *
06                                    sizeof(struct mem_section);
07
08         if (slab_is_available()) {
09                         section = kzalloc_node(array_size, GFP_KERNEL, n
   id);
10         } else {
11                 section = memblock_alloc_node(array_size, SMP_CACHE_BYTE
   S,
12                                               nid);
13                 if (!section)
14                         panic("%s: Failed to allocate %lu bytes nid=%d
   \n",
15                               __func__, array_size, nid);
16         }
17
18         return section;
19 }
20 #endif
```

If you use the CONFIG_SPARSEMEM_EXTREME kernel option, you will be allocated memory for the mem_section[] table.

- In lines 5~6 of the code, the root entry consists of 1 page, and the size of the array of structs mem_section be full. SECTIONS_PER_ROOT refers to the number of mem_section per root entry. This constant is defined as follows:
  - #define SECTIONS_PER_ROOT (PAGE_SIZE / sizeof(struct mem_section))
    - e.g. arm64: 4K / 16 bytes = 256
- If the slab memory allocator is in action in line 8~9 of the code, it allocates memory via the kzalloc( ) function.
- If the slab memory allocator does not work in lines 10~16 of the code, it is allocated to the memblock of that node
- Returns the array of mem_section allocated in line 18 of code.

set_section_nid()

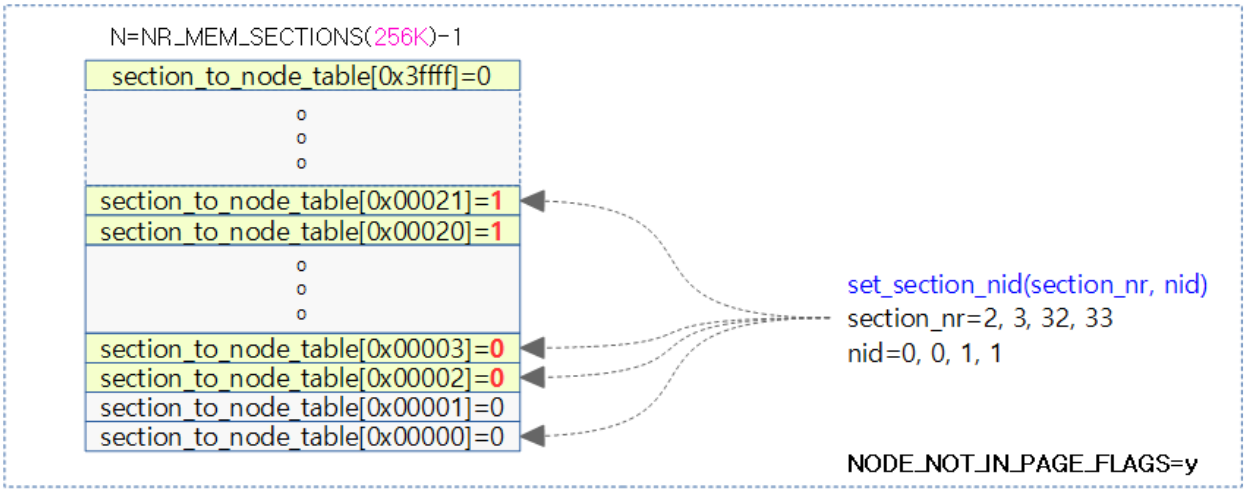
mm/sparse.c

```
01 #ifdef NODE_NOT_IN_PAGE_FLAGS
02 static void set_section_nid(unsigned long section_nr, int nid)
03 {
04         section_to_node_table[section_nr] = nid;
05 }
06 #else /* !NODE_NOT_IN_PAGE_FLAGS */
07 static inline void set_section_nid(unsigned long section_nr, int nid)
08 {
09 }
10 #endif
```

If you use the NODE_NOT_IN_PAGE_FLAGS option, store the node number in the section_to_node_table[] that corresponds 1:1 to the section number.

- This option is used in 32-bit architectures where there are not enough bits to store the node number in the page struct member variable flags.

The figure below uses the set_section_nid() function to store the node number in the given section number.



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

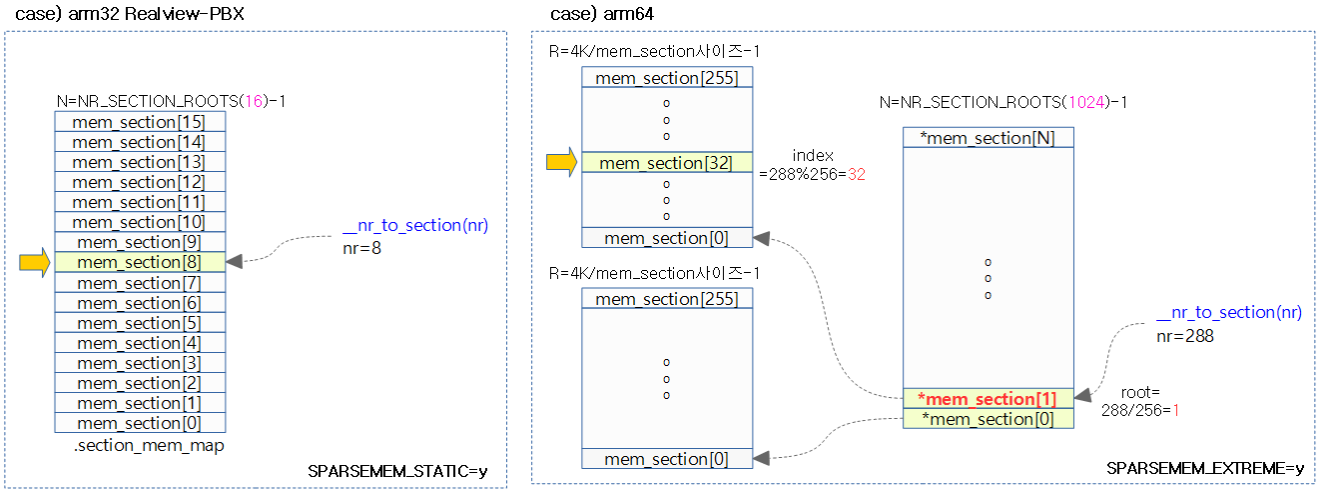
__nr_to_section()

mm/sparse.c

```
01 static inline struct mem_section *__nr_to_section(unsigned long nr)
02 {
03 #ifdef CONFIG_SPARSEMEM_EXTREME
04         if (!mem_section)
05                 return NULL;
06 #endif
07         if (!mem_section[SECTION_NR_TO_ROOT(nr)])
08                 return NULL;
09         return &mem_section[SECTION_NR_TO_ROOT(nr)][nr & SECTION_ROOT_MA
   SK];
10 }
```

Get the information about the mem_section structure that corresponds to the section number.

The figure below shows two examples of steps to use the __nr_to_section() function to get the mem_section struct information by section number.



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

__section_mark_present()

mm/sparse.c

```
1  /*
2   * There are a number of times that we loop over NR_MEM_SECTIONS,
3   * looking for section_present() on each.  But, when we have very
4   * large physical address spaces, NR_MEM_SECTIONS can also be
5   * very large which makes the loops quite long.
6   *
7   * Keeping track of this gives us an easy way to break out of
8   * those loops early.
9   */
```

```
1  unsigned long __highest_present_section_nr;
2  static void __section_mark_present(struct mem_section *ms,
3                  unsigned long section_nr)
4  {
5          if (section_nr > __highest_present_section_nr)
6                  __highest_present_section_nr = section_nr;
7
8          ms->section_mem_map |= SECTION_MARKED_PRESENT;
9  }
```

Indicates the existence of section memory.

- In line 5~6 of the code, update the global variable __highest_present_section_nr with the highest section number.
- In line 8 of code, add a SECTION_MARKED_PRESENT flag bit to ms->section_mem_map that points to mem_section.

for_each_present_section_nr()

mm/sparse.c

```
1  #define for_each_present_section_nr(start, section_nr)          \
2          for (section_nr = next_present_section_nr(start-1);     \
3               ((section_nr >= 0) &&                              \
4                (section_nr <= __highest_present_section_nr));    \
5               section_nr = next_present_section_nr(section_nr))
```

@start rotates through each present section, including the last present section. @section_nr is the output argument and is the present section number.

next_present_section_nr()

mm/sparse.c

```
01 static inline int next_present_section_nr(int section_nr)
02 {
03         do {
04                 section_nr++;
05                 if (present_section_nr(section_nr))
06                         return section_nr;
07         } while ((section_nr <= __highest_present_section_nr));
08
09         return -1;
10 }
```

@section_nr requested iterations from the next section number to the last section, and if it finds a present section, it returns the section number.

Setting a Pageblock Order

set_pageblock_order()

mm/page_alloc.c

```
01 #ifdef CONFIG_HUGETLB_PAGE_SIZE_VARIABLE
02
03 /* Initialise the number of pages represented by NR_PAGEBLOCK_BITS */
04 void __init set_pageblock_order(void)
05 {
06         unsigned int order;
07
08         /* Check that pageblock_nr_pages has not already been setup */
09         if (pageblock_order)
10                 return;
11
12         if (HPAGE_SHIFT > PAGE_SHIFT)
13                 order = HUGETLB_PAGE_ORDER;
14         else
15                 order = MAX_ORDER - 1;
16
17         /*
18          * Assume the largest contiguous order of interest is a huge pag
    e.
19          * This value may be variable depending on boot parameters on IA
    64 and
20          * powerpc.
21          */
22         pageblock_order = order;
23 }
24 #else /* CONFIG_HUGETLB_PAGE_SIZE_VARIABLE */
25
26 /*
27  * When CONFIG_HUGETLB_PAGE_SIZE_VARIABLE is not set, set_pageblock_orde
    r()
28  * is unused as pageblock_order is set at compile-time. See
29  * include/linux/pageblock-flags.h for the values of pageblock_order bas
    ed on
```

```
 * the kernel config
 */
void __init set_pageblock_order(void)
{
}

#endif /* CONFIG_HUGETLB_PAGE_SIZE_VARIABLE */
```

CONFIG_HUGETLB_PAGE_SIZE_VARIABLE configure the pageblock_order at runtime only if you use kernel options.

pageblock_order

include/linux/pageblock-flags.h

```
#ifdef CONFIG_HUGETLB_PAGE

#ifdef CONFIG_HUGETLB_PAGE_SIZE_VARIABLE

/* Huge page sizes are variable */
extern unsigned int pageblock_order;

#else /* CONFIG_HUGETLB_PAGE_SIZE_VARIABLE */

/* Huge pages are a constant size */
#define pageblock_order         HUGETLB_PAGE_ORDER

#endif /* CONFIG_HUGETLB_PAGE_SIZE_VARIABLE */

#else /* CONFIG_HUGETLB_PAGE */

/* If huge pages are not used, group by MAX_ORDER_NR_PAGES */
#define pageblock_order         (MAX_ORDER-1)

#endif /* CONFIG_HUGETLB_PAGE */
```

The size of the page block order is determined by the availability of CONFIG_HUGETLB_PAGE and CONFIG_HUGETLB_PAGE_SIZE_VARIABLE kernel options.

- When using CONFIG_HUGETLB_PAGE
  - When using CONFIG_HUGETLB_PAGE_SIZE_VARIABLE kernel options
    - The pageblock_order is determined at runtime in the set_pageblock_order() function.
  - Without CONFIG_HUGETLB_PAGE_SIZE_VARIABLE kernel options
    - It is determined by the value of the HUGETLB_PAGE_ORDER at compile time.
      - =HPAGE_SHIFT(21) – PAGE_SHIFT(12)=9 (with arm64 default settings)
- CONFIG_HUGETLB_PAGE If you don't use kernel options
  - It is determined at compile time as a MAX_ORDER-1 value.

Per-node sparse initialization

sparse_init_nid()

mm/sparse.c

```
/*
```

```
 * Initialize sparse on a specific node. The node spans [pnum_begin, pnum_end)
 * And number of present sections in this node is map_count.
 */
```

```
01 static void __init sparse_init_nid(int nid, unsigned long pnum_begin,
02                                    unsigned long pnum_end,
03                                    unsigned long map_count)
04 {
05         struct mem_section_usage *usage;
06         unsigned long pnum;
07         struct page *map;
08
09         usage = sparse_early_usemaps_alloc_pgdat_section(NODE_DATA(nid),
10                         mem_section_usage_size() * map_count);
11         if (!usage) {
12                 pr_err("%s: node[%d] usemap allocation failed", __func__, nid);
13                 goto failed;
14         }
15         sparse_buffer_init(map_count * section_map_size(), nid);
16         for_each_present_section_nr(pnum_begin, pnum) {
17                 unsigned long pfn = section_nr_to_pfn(pnum);
18
19                 if (pnum >= pnum_end)
20                         break;
21
22                 map = __populate_section_memmap(pfn, PAGES_PER_SECTION,
23                                 nid, NULL);
24                 if (!map) {
25                         pr_err("%s: node[%d] memory map backing failed. Some memory will not be available.",
26                                __func__, nid);
27                         pnum_begin = pnum;
28                         sparse_buffer_fini();
29                         goto failed;
30                 }
31                 check_usemap_section_nr(nid, usage);
32                 sparse_init_one_section(__nr_to_section(pnum), pnum, map, usage,
33                                 SECTION_IS_EARLY);
34                 usage = (void *) usage + mem_section_usage_size();
35         }
36         sparse_buffer_fini();
37         return;
38 failed:
39         /* We failed to allocate, mark all the following pnums as not present */
40         for_each_present_section_nr(pnum_begin, pnum) {
41                 struct mem_section *ms;
42
43                 if (pnum >= pnum_end)
44                         break;
45                 ms = __nr_to_section(pnum);
46                 ms->section_mem_map = 0;
47         }
48 }
```

For the range of sections below @pnum_begin ~ @pnum_end corresponding to the @nid nodes, @map_count assigns and initializes the mem_map and usemaps associated with the present sections.

- In lines 9~14 of code, allocate a usemap[] array equal to the number of @map_count sections at once on the node @nid.
- In line 15 of the code, we prepare a sparse buffer equal to the number of @map_count sections by allocating it at the node @nid at a time.

- In line 16~30 of the code, it goes from the @pnum_begin section to the end present section and is assigned the mem_map of that section.
  - If possible, it is allocated in a sparse buffer.
- If the usemap and node information (pgdat) assigned in line 31 of the code are not the same as the recorded section, it prints a warning message. If the usemap information is not configured in the same node space, a circular dependancy problem can occur when memory hot remov must be removed on a per-node basis.
- Initialize one of the sections in line 32~33 of the code.
- Allocate in line 36 and deallocate the remaining sparse buffer.

How to Bracket

When begin is 1 and end is 5

- [begin, end]
  - Begin and above ~ Below end
  - 1~5
- (begin, end)
  - Greater than bigin ~ less than end
  - 2~4
- [begin, end)
  - More than begin ~ less than end
  - 1~4
- (begin, end]
  - Above begin, below end
  - 2~5

Assign a usemap

sparse_early_usemaps_alloc_pgdat_section()

mm/sparse.c

```
01 #ifdef CONFIG_MEMORY_HOTREMOVE
02 static unsigned long * __init
03 sparse_early_usemaps_alloc_pgdat_section(struct pglist_data *pgdat,
04                                          unsigned long size)
05 {
06         struct mem_section_usage *usage;
07         unsigned long goal, limit;
08         int nid;
09         /*
10          * A page may contain usemaps for other sections preventing the
11          * page being freed and making a section unremovable while
12          * other sections referencing the usemap remain active. Similarl
   y,
13          * a pgdat can prevent a section being removed. If section A
14          * contains a pgdat and section B contains the usemap, both
15          * sections become inter-dependent. This allocates usemaps
16          * from the same section as the pgdat where possible to avoid
17          * this problem.
18          */
19         goal = pgdat_to_phys(pgdat) & (PAGE_SECTION_MASK << PAGE_SHIFT);
20         limit = goal + (1UL << PA_SECTION_SHIFT);
```

```
21          nid = early_pfn_to_nid(goal >> PAGE_SHIFT);
22  again:
23          usage = memblock_alloc_try_nid(size, SMP_CACHE_BYTES, goal, limi
    t, nid);
24          if (!usage && limit) {
25                  limit = 0;
26                  goto again;
27          }
28          return usage;
29  }
30  #endif
```

If you use CONFIG_MEMORY_HOTREMOVE kernel options, assign a usemap. Whenever possible, try to allocate memory so that the usemap can fit in the section containing the node information (pgdat). If it fails, try the assignment again, regardless of the location.

- In lines 19~20 of the code, the goal has the physical address of the start of the section at the node, and the limit is limited to 1 section size.
- In line 21 of the code, get the node information for each section that you have already stored in the mem_map.
- In line 22~23 of the code, again: is the label. Within the goal~limit range of the specified node, i.e., the section area containing the node information, the SMP_CACHE_BYTES align requests the allocation of memblock space equal to size.
- If you try once in line 24~27 of the code and it can't be allocated, try again by setting the limit to 0.
- Returns the usemap assigned in line 28 of code.

Allocate/release Sparse buffers for mem_map

Attempt to allocate the mem_map[] array for the entire present section memory corresponding to the node at once. If this attempt is made and allocated, this sparse buffer memory is used as a mem_map[] array. If the allocation fails, it will be fallback, and it will simply be assigned by section.

- Consultation:
  - mm/sparse: abstract sparse buffer allocations (https://github.com/torvalds/linux/commit/35fd1eb1e8212c02f6eae24335a9e5b80f9519b4#diff-a2fb2d11bf168fb9d82bdc8284653333)
  - mm/sparse: optimize memmap allocation during sparse_init() (https://github.com/torvalds/linux/commit/c98aff649349d9147915a19d378c9c3c1bd85de0#diff-a2fb2d11bf168fb9d82bdc8284653333)

sparse_buffer_init()

mm/sparse.c

```
1  static void __init sparse_buffer_init(unsigned long size, int nid)
2  {
3          WARN_ON(sparsemap_buf); /* forgot to call sparse_buffer_fini()?
   */
4          sparsemap_buf =
5                  memblock_alloc_try_nid_raw(size, PAGE_SIZE,
6                                             __pa(MAX_DMA_ADDRESS),
```

```
7                                                       MEMBLOCK_ALLOC_ACCESSIBL
  E, nid);
8               sparsemap_buf_end = sparsemap_buf + size;
9       }
```

Allocate a sparse buffer on a per-page page basis in the node @nid for the @size requested.

sparse_buffer_alloc()

mm/sparse.c

```
01  void * __meminit sparse_buffer_alloc(unsigned long size)
02  {
03          void *ptr = NULL;
04
05          if (sparsemap_buf) {
06                  ptr = PTR_ALIGN(sparsemap_buf, size);
07                  if (ptr + size > sparsemap_buf_end)
08                          ptr = NULL;
09                  else
10                          sparsemap_buf = ptr + size;
11          }
12          return ptr;
13  }
```

Allocate @size amount of memory from the sparse buffer. (for mem_map)

sparse_buffer_fini()

mm/sparse.c

```
1   static void __init sparse_buffer_fini(void)
2   {
3           unsigned long size = sparsemap_buf_end - sparsemap_buf;
4
5           if (sparsemap_buf && size > 0)
6                   memblock_free_early(__pa(sparsemap_buf), size);
7           sparsemap_buf = NULL;
8   }
```

deallocate the remaining sparse buffer.

Usemap assignment section or check nodes

check_usemap_section_nr()

mm/sparse.c

```
01  #ifdef CONFIG_MEMORY_HOTREMOVE
02  static void __init check_usemap_section_nr(int nid,
03                  struct mem_section_usage *usage)
04  {
05          unsigned long usemap_snr, pgdat_snr;
06          static unsigned long old_usemap_snr;
07          static unsigned long old_pgdat_snr;
08          struct pglist_data *pgdat = NODE_DATA(nid);
09          int usemap_nid;
10
11          /* First call */
```

```
12          if (!old_usemap_snr) {
13                  old_usemap_snr = NR_MEM_SECTIONS;
14                  old_pgdat_snr = NR_MEM_SECTIONS;
15          }
16
17          usemap_snr = pfn_to_section_nr(__pa(usage) >> PAGE_SHIFT);
18          pgdat_snr = pfn_to_section_nr(pgdat_to_phys(pgdat) >> PAGE_SHIF
    T);
19          if (usemap_snr == pgdat_snr)
20                  return;
21
22          if (old_usemap_snr == usemap_snr && old_pgdat_snr == pgdat_snr)
23                  /* skip redundant message */
24                  return;
25
26          old_usemap_snr = usemap_snr;
27          old_pgdat_snr = pgdat_snr;
28
29          usemap_nid = sparse_early_nid(__nr_to_section(usemap_snr));
30          if (usemap_nid != nid) {
31                  pr_info("node %d must be removed before remove section %
    ld\n",
32                          nid, usemap_snr);
33                  return;
34          }
35          /*
36           * There is a circular dependency.
37           * Some platforms allow un-removable section because they will j
    ust
38           * gather other removable sections for dynamic partitioning.
39           * Just notify un-removable section's number here.
40           */
41          pr_info("Section %ld and %ld (node %d) have a circular dependenc
    y on usemap and pgdat allocations\n",
42                  usemap_snr, pgdat_snr, nid);
43  }
44  #endif
```

If you use CONFIG_MEMORY_HOTREMOVE kernel options, the usemap section should not be deleted until the pgdat is located, or else all other sections have been deleted. Therefore, it is a routine to find out the relationship between them in a message. If the assigned usemap and node information (pgdat) are not the same as the recorded section, the information is printed. If the usemap information is not configured in the same node space, a circular dependancy problem can occur when memory hot remov must be removed on a per-node basis.

- 참고: memory hotplug: allocate usemap on the section with pgdat (https://github.com/torvalds/linux/commit/48c906823f3927b981db9f0b03c2e2499977ee93)

- On line 12~15 of the code, the initial value of the call contains the section number in progress.
- In line 17~20 of the code, if the section where the usemap is assigned and the section where the node information is recorded are the same, it is normal and the function is exited.
- If the section has already been played once in line 22~24 of the code, the function is exited to skip.
- If you proceed again with the same section number in line 26~27, remember the section numbers in order to skip them.
- If the @nid requested in lines 29~34 does not have a usemap node, it prints the information that the section with the use_map should be hot removed first.

- If there is a usemap in the @nid requested in lines 41~42 of the code, it outputs the information that a circular dependancy problem may occur when memory must be removed on a node-by-node basis during a memory hot remove operation.

usemap_size()

mm/sparse.c

```
1 unsigned long usemap_size(void)
2 {
3         return BITS_TO_LONGS(SECTION_BLOCKFLAGS_BITS) * sizeof(unsigned
  long);
4 }
```

Returns usemap size.

- SECTION_BLOCKFLAGS_BITS
  - Number of pageblock bits per section (when pageblock_order=9)
    - arm64=2048
- e.g. arm64
  - 2048 / 8=256(byte)

sparse_mem_map_populate()

This is a function when vmemmap is not used. The arm64 default configuration uses vmemmap, so it doesn't use this function.

mm/sparse.c

```
01 #ifndef CONFIG_SPARSEMEM_VMEMMAP
02 struct page __init *sparse_mem_map_populate(unsigned long pnum, int nid,
03                 struct vmem_altmap *altmap)
04 {
05         unsigned long size = section_map_size();
06         struct page *map = sparse_buffer_alloc(size);
07
08         if (map)
09                 return map;
10
11         map = memblock_alloc_try_nid(size,
12                                         PAGE_SIZE, __pa(MAX_DMA_ADDRES
  S),
13                                         MEMBLOCK_ALLOC_ACCESSIBLE, ni
  d);
14         return map;
15 }
16 #endif /* !CONFIG_SPARSEMEM_VMEMMAP */
```

On systems that don't use the CONFIG_SPARSEMEM_VMEMMAP kernel option, it uses a pre-allocated sparse buffer to allocate mem_map[]. If that fails, request a memblock allocation of a mem_map[] array for each separate section.

sparse_init_one_section()

mm/sparse.c

```
static void __meminit sparse_init_one_section(struct mem_section *ms,
                unsigned long pnum, struct page *mem_map,
                struct mem_section_usage *usage, unsigned long flags)
{
        ms->section_mem_map &= ~SECTION_MAP_MASK;
        ms->section_mem_map |= sparse_encode_mem_map(mem_map, pnum)
                | SECTION_HAS_MEM_MAP | flags;
        ms->usage = usage;
}
```

Encode the mem_map region and link it to the mem_section, and also connect the usemap area.

Encoding/decoding of mem_map addresses

The section_mem_map members of the mem_section structure that manages each section contain two pieces of information: However, they are not used at the same time.

- At boot time, it briefly contains the early node number for the section.
  - Use the sparse_encode_early_nid() and sparse_early_nid() functions.
  - After mem_map is enabled, the node number is stored and used in page->flags.
- mem_map each section manages – encodes a few flags in the pfn value for the section.
  - Use the sparse_encode_mem_map() and sparse_decode_mem_map() functions.
  - Subtracting and storing the pfn value for a section saves one arithmetic operation when using the pfn_to_page() function on systems that do not use vmemmap.
  - e.g. 0x8000_0000 ~ 0xC000_0000 (1G) is encoded as the mem_map address – 0x80000(section pfn) + flag value.

Early Node Number Encoding/Decoding

sparse_encode_early_nid()

mm/sparse.c

```
/*
 * During early boot, before section_mem_map is used for an actual
 * mem_map, we use section_mem_map to store the section's NUMA
 * node.  This keeps us from having to use another data structure.  The
 * node information is cleared just before we store the real mem_map.
 */
```

```
static inline unsigned long sparse_encode_early_nid(int nid)
{
        return (nid << SECTION_NID_SHIFT);
}
```

Node numbers are used from SECTION_NID_SHIFT (6) bits, so shift them to the left.

sparse_early_nid()

mm/sparse.c

```
static inline int sparse_early_nid(struct mem_section *section)
```

```
2 {
3         return (section->section_mem_map >> SECTION_NID_SHIFT);
4 }
```

mem_section extracts node information from the section_mem_map, which is a member variable of the structure, and returns it.

Regular mem_map encoding/decoding

sparse_encode_mem_map()

mm/sparse.c

```
1 /*
2  * Subtle, we encode the real pfn into the mem_map such that
3  * the identity pfn - section_mem_map will return the actual
4  * physical page frame number.
5  */
```

```
1 static unsigned long sparse_encode_mem_map(struct page *mem_map, unsigne
  d long pnum)
2 {
3         unsigned long coded_mem_map =
4                 (unsigned long)(mem_map - (section_nr_to_pfn(pnum)));
5         BUILD_BUG_ON(SECTION_MAP_LAST_BIT > (1UL<<PFN_SECTION_SHIFT));
6         BUG_ON(coded_mem_map & ~SECTION_MAP_MASK);
7         return coded_mem_map;
8 }
```

Returns the base pfn value corresponding to the address – section of the assigned mem_map as the encoding value.

- If you don't use vmemmap, it has the effect of removing one arithmetic operation from the pfn_to_page() function.

sparse_decode_mem_map()

mm/sparse.c

```
1 /*
2  * Decode mem_map from the coded memmap
3  */
```

```
1 struct page *sparse_decode_mem_map(unsigned long coded_mem_map, unsigned
  long pnum)
2 {
3         /* mask off the extra low bits of information */
4         coded_mem_map &= SECTION_MAP_MASK;
5         return ((struct page *)coded_mem_map) + section_nr_to_pfn(pnum);
6 }
```

Remove the flag information from the encoded mem_map address and decode it to return only the mem_map address.

VMEMMAP RELATED

- arm64 uses CONFIG_SPARSEMEM_VMEMMAP by default.
- If you have enough system resources to use vmemmap, the pfn_to_page() and page_to_pfn() functions will work faster and most effectively.

__populate_section_memmap()

mm/sparse-vmemmap.c

```
01 struct page * __meminit __populate_section_memmap(unsigned long pfn,
02                 unsigned long nr_pages, int nid, struct vmem_altmap *alt
   map)
03 {
04         unsigned long start = (unsigned long) pfn_to_page(pfn);
05         unsigned long end = start + nr_pages * sizeof(struct page);
06
07         if (WARN_ON_ONCE(!IS_ALIGNED(pfn, PAGES_PER_SUBSECTION) ||
08                 !IS_ALIGNED(nr_pages, PAGES_PER_SUBSECTION)))
09                 return NULL;
10
11         if (vmemmap_populate(start, end, nid, altmap))
12                 return NULL;
13
14         return pfn_to_page(pfn);
15 }
```

If you use the CONFIG_SPARSEMEM_VMEMMAP kernel option, the @pfn corresponding to the mem_map region is mapped to the vmemmap area @nr_pages. After obtaining the address range with the requested section, if there are any tables that are not configured in the pgd, pud, and pmd tables associated with that address range, the node allocates pages to organize and map them.

- The spaces for mem_map allocated in the sparse buffer are mapped by connecting them to the PTE entries.

vmemmap_populate()

arch/arm64/mm/mmu.c

```
01 #if !ARM64_KERNEL_USES_PMD_MAPS
02 int __meminit vmemmap_populate(unsigned long start, unsigned long end, i
   nt node,
03                 struct vmem_altmap *altmap)
04 {
05         WARN_ON((start < VMEMMAP_START) || (end > VMEMMAP_END));
06         return vmemmap_populate_basepages(start, end, node, altmap);
07 }
08 #else   /* !ARM64_KERNEL_USES_PMD_MAPS */
09 int __meminit vmemmap_populate(unsigned long start, unsigned long end, i
   nt node,
10                 struct vmem_altmap *altmap)
11 {
12         unsigned long addr = start;
13         unsigned long next;
14         pgd_t *pgdp;
15         p4d_t *p4dp;
16         pud_t *pudp;
17         pmd_t *pmdp;
18
```

```
19          WARN_ON((start < VMEMMAP_START) || (end > VMEMMAP_END));
20          do {
21                  next = pmd_addr_end(addr, end);
22
23                  pgdp = vmemmap_pgd_populate(addr, node);
24                  if (!pgdp)
25                          return -ENOMEM;
26
27                  p4dp = vmemmap_p4d_populate(pgdp, addr, node);
28                  if (!p4dp)
29                          return -ENOMEM;
30
31                  pudp = vmemmap_pud_populate(p4dp, addr, node);
32                  if (!pudp)
33                          return -ENOMEM;
34
35                  pmdp = pmd_offset(pudp, addr);
36                  if (pmd_none(READ_ONCE(*pmdp))) {
37                          void *p = NULL;
38
39                          p = vmemmap_alloc_block_buf(PMD_SIZE, node, altm
    ap);
40                          if (!p) {
41                                  if (vmemmap_populate_basepages(addr, nex
    t, node, altmap))
42                                          return -ENOMEM;
43                                  continue;
44                          }
45
46                          pmd_set_huge(pmdp, __pa(p), __pgprot(PROT_SECT_N
    ORMAL));
47                  } else
48                          vmemmap_verify((pte_t *)pmdp, node, addr, next);
49          } while (addr = next, addr != end);
50
51          return 0;
52  }
53  #endif  /* !ARM64_KERNEL_USES_PMD_MAPS */
```

If there are unconfigured tables in the pgd, p4d, pud, and pmd tables associated with the request address range, node allocates pages to organize and map them.

- If you're using 4K pages, the ARM64_KERNEL_USES_PMD_MAPS value is 1.
- @start: The starting virtual address to which the mem_map will be mapped (mapped to the vmemmap zone)
- @end: End virtual address to which the mem_map will be mapped (mapped to the vmemmap zone)

- In line 6 of code, map the mem_map in 4K increments.
- Traversing addr in code lines 20~21, and since we are going to map it to pmd units, we will get the end address of pmd units and assign it to next.
- In lines 23~25 of the code, populate the pgd entry corresponding to the corresponding virtual address location.
- In lines 27~29 of code, populate the p4d entry corresponding to that virtual address location.
- In lines 31~33 of code, populate the PUD entry corresponding to that virtual address location.
- If it is not mapped to the pmd entry address in code lines 35~46, it allocates PMD size (2M) of memory and then maps it with the pmd_set_huge() command.

- If PMD is already mapped in code lines 47~48, the vmemmap_verify() function will output a warning log if the mem_map is assigned to another node.
- On line 49 of the code, go to the address of the phonetic PMD unit and continue the loop.
- Normal result on line 51 of code returns 0.

vmemmap_populate_basepages()

mm/sparse-vmemmap.c

```
01 int __meminit vmemmap_populate_basepages(unsigned long start, unsigned l
   ong end,
02                                         int node, struct vmem_altmap *a
   ltmap)
03 {
04         unsigned long addr = start;
05         pgd_t *pgd;
06         p4d_t *p4d;
07         pud_t *pud;
08         pmd_t *pmd;
09         pte_t *pte;
10
11         for (; addr < end; addr += PAGE_SIZE) {
12                 pgd = vmemmap_pgd_populate(addr, node);
13                 if (!pgd)
14                         return -ENOMEM;
15                 p4d = vmemmap_p4d_populate(pgd, addr, node);
16                 if (!p4d)
17                         return -ENOMEM;
18                 pud = vmemmap_pud_populate(p4d, addr, node);
19                 if (!pud)
20                         return -ENOMEM;
21                 pmd = vmemmap_pmd_populate(pud, addr, node);
22                 if (!pmd)
23                         return -ENOMEM;
24                 pte = vmemmap_pte_populate(pmd, addr, node, altmap);
25                 if (!pte)
26                         return -ENOMEM;
27                 vmemmap_verify(pte, node, addr, addr + PAGE_SIZE);
28         }
29
30         return 0;
31 }
```

Map the mem_map in 4K increments.

- @start: The starting virtual address to which the mem_map will be mapped (mapped to the vmemmap zone)
- @end: End virtual address to which the mem_map will be mapped (mapped to the vmemmap zone)

- At line 11 of the code, traverse page by page.
- In lines 12~14 of the code, populate the pgd entry corresponding to the corresponding virtual address location.
- In lines 15~17 of code, populate the p4d entry corresponding to that virtual address location.
- In lines 18~20 of code, populate the PUD entry corresponding to that virtual address location.

- In lines 21~23 of the code, populate the pmd entry corresponding to the corresponding virtual address location.
- In lines 24~26 of code, map the @altmap to the PTE entry corresponding to the corresponding virtual address location.
- In line 27 of the code, the vmemmap_verify() function prints a warning log if the mem_map is assigned to another node.
- Normal result on line 30 of code returns 0.

vmemmap_verify()

mm/sparse-vmemmap.c

```
01 void __meminit vmemmap_verify(pte_t *pte, int node,
02                                 unsigned long start, unsigned long end)
03 {
04         unsigned long pfn = pte_pfn(*pte);
05         int actual_node = early_pfn_to_nid(pfn);
06
07         if (node_distance(actual_node, node) > LOCAL_DISTANCE)
08                 pr_warn("[%lx-%lx] potential offnode page_structs\n",
09                         start, end - 1);
10 }
```

If the mem_map node and @node to which the @pte points are not on the same local node and are separated by remote distance, it outputs a warning log.

The code and description of the following functions are omitted.

- vmemmap_pgd_populate()
- vmemmap_p4d_populate()
- vmemmap_pud_populate()
- vmemmap_pmd_populate()

Structs and Key Variables

mem_section[]

mm/sparse.c

```
1 #ifdef CONFIG_SPARSEMEM_EXTREME
2 extern struct mem_section **mem_section;
3 #else
4 extern struct mem_section mem_section[NR_SECTION_ROOTS][SECTIONS_PER_ROO
  T];
5 #endif
```

mem_section are section-by-section linked to information in the mem_map[] and usemap[] arrays.

mem_section Structure

include/linux/mmzone.h

```
01 struct mem_section {
02         /*
03          * This is, logically, a pointer to an array of struct
04          * pages.  However, it is stored with some other magic.
05          * (see sparse.c::sparse_init_one_section())
06          *
07          * Additionally during early boot we encode node id of
08          * the location of the section here to guide allocation.
09          * (see sparse.c::memory_present())
10          *
11          * Making it a UL at least makes someone do a cast
12          * before using it wrong.
13          */
14         unsigned long section_mem_map;
15
16         struct mem_section_usage *usage;
17 #ifdef CONFIG_PAGE_EXTENSION
18         /*
19          * If SPARSEMEM, pgdat doesn't have page_ext pointer. We use
20          * section. (see page_ext.h about this.)
21          */
22         struct page_ext *page_ext;
23         unsigned long pad;
24 #endif
25         /*
26          * WARNING: mem_section must be a power-of-2 in size for the
27          * calculation and use of SECTION_ROOT_MASK to make sense.
28          */
29 };
```

- section_mem_map
  - It doesn't point directly to the mem_map of the actual section, but uses the encoded values as follows:
    - Address pointing to the mem_map of the section – used to encode the flags in the pfn value for the section. (mem_map are used in page-by-page order.)
  - The following is the flag information to use for encoding.
    - bit0: SECTION_MARKED_PRESENT
      - Expresses whether there is memory in the section.
    - bit1: SECTION_HAS_MEM_MAP
      - Indicates whether a mem_map is linked to a section.
      - The linked mem_map is stored without subtracting the section base pfn value (encoding)
    - bit2: SECTION_IS_ONLINE
      - online section.
    - bit3: SECTION_IS_EARLY
      - Early section.
    - bit4: SECTION_TAINT_ZONE_DEVICE
      - John expresses whether or not the device is.
    - bits[6~]: Node number
      - Node numbers and mem_map information are not stored at the same time.
      - The node number is only used briefly during the early bootup. The node number is then managed in the flags member of the page structure.
- *usage

    - mem_section_usage refers to a struct. (Manages usemap and subsection bitmaps)
- *page_ext
    - It is used when using CONFIG_PAGE_EXTENSION kernel options for debugging, and points to where the page_ext structs are located.

mem_section_usage Structure

include/linux/mmzone.h

```
struct mem_section_usage {
#ifdef CONFIG_SPARSEMEM_VMEMMAP
        DECLARE_BITMAP(subsection_map, SUBSECTIONS_PER_SECTION);
#endif
        /* See declaration of similar field in struct zone */
        unsigned long pageblock_flags[0];
};
```

- subsection_map
    - A bitmap expressing the presence of subsectional units (2M) of memory within a section memory
- pageblock_flags[]
    - The address that points to the usemap

guitar

SECTION_SIZE_BITS & MAX_PHYSMEM_BITS

- arm
    - 28, 32 (256M, 4G, Realview-PBX)
    - 26, 29 (64M, 512M, RPC)
    - 27, 32 (128M, 4G, SA1100)
- 32bit arm with LPAE
    - 34, 36 (16G, 64G, Keystone)
- arm64
    - 27, 48 (128M, 256T)
- x86
    - 26, 32 (64M, 4G)
- x86_32 with PAE
    - 29, 36 (512M, 64G)
- x86_64
    - 27, 46 (128M, 64T)

consultation

- Memory Model -1- (Basic) (http://jake.dothome.co.kr/mm-1) | 문c
- Memory Model -2- (mem_map) (http://jake.dothome.co.kr/mem_map) | 문c

- Memory Model -3- (Sparse Memory) (http://jake.dothome.co.kr/sparsemem/) | Sentence C – Current post
- Memory Model -4- (APIs) (http://jake.dothome.co.kr/mem_map_page) | 문c
- ZONE Type (http://jake.dothome.co.kr/zone-types) | Qc
- bootmem_init (http://jake.dothome.co.kr/bootmem_init-64) | Qc
- zone_sizes_init() (http://jake.dothome.co.kr/free_area_init_node/) | Qc
- NUMA -1- (ARM64 initialization) (http://jake.dothome.co.kr/numa-1) | Qc
- build_all_zonelists() (http://jake.dothome.co.kr/build_all_zonelists) | Qc

3 thoughts to “Memory Model -3- (Sparse Memory)”

AS IT IS

2020-05-28 23:53 (http://jake.dothome.co.kr/sparsemem/#comment-251996)

Hi, in the part where the usemap picture is, 1 section is 1GB(2^30), and the usemap is divided into 2M(2^21) blocks and expressed as 4bits, so
I understood that the number of blocks in
one
usemap is 512 = 1GB / 2M = 2^(30-21) = 2^9.

usemapWhat does the 'B' mean in usemap(256B) in the middle of the image?
If 'B' means block, then there are 2 blocks in one section, which is 512M, so
it would be 512B...?

RESPONSE (/SPARSEMEM/?REPLYTOCOM=251996#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)

2020-06-22 14:10 (http://jake.dothome.co.kr/sparsemem/#comment-257091)

Hello Lee,

As Kwon said, B is for Byte. (Sorry for the shortness)
It expresses 4 bits of mobility per block, so 512 blocks x 4 bits = 256 bytes.

I appreciate it.

RESPONSE (/SPARSEMEM/?REPLYTOCOM=257091#RESPOND)

KWON YONGBEOM

2020-05-30 02:07 (http://jake.dothome.co.kr/sparsemem/#comment-252222)

Hello.

Moon Young-il says it's hard to access for the time being (TT)

There are 4 512-bits, so it's 2048 bits, and if you divide by 8 to calculate bytes, you get 256B(yte).

RESPONSE (/SPARSEMEM/?REPLYTOCOM=252222#RESPOND)

LEAVE A COMMENT

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

Comments

name *

email *

Website

WRITE A COMMENT

‹ zone_sizes_init() (http://jake.dothome.co.kr/free_area_init_node/)

Memory Model -4- (APIs) › (http://jake.dothome.co.kr/mem_map_page/)

Munc Blog (2015 ~ 2023)