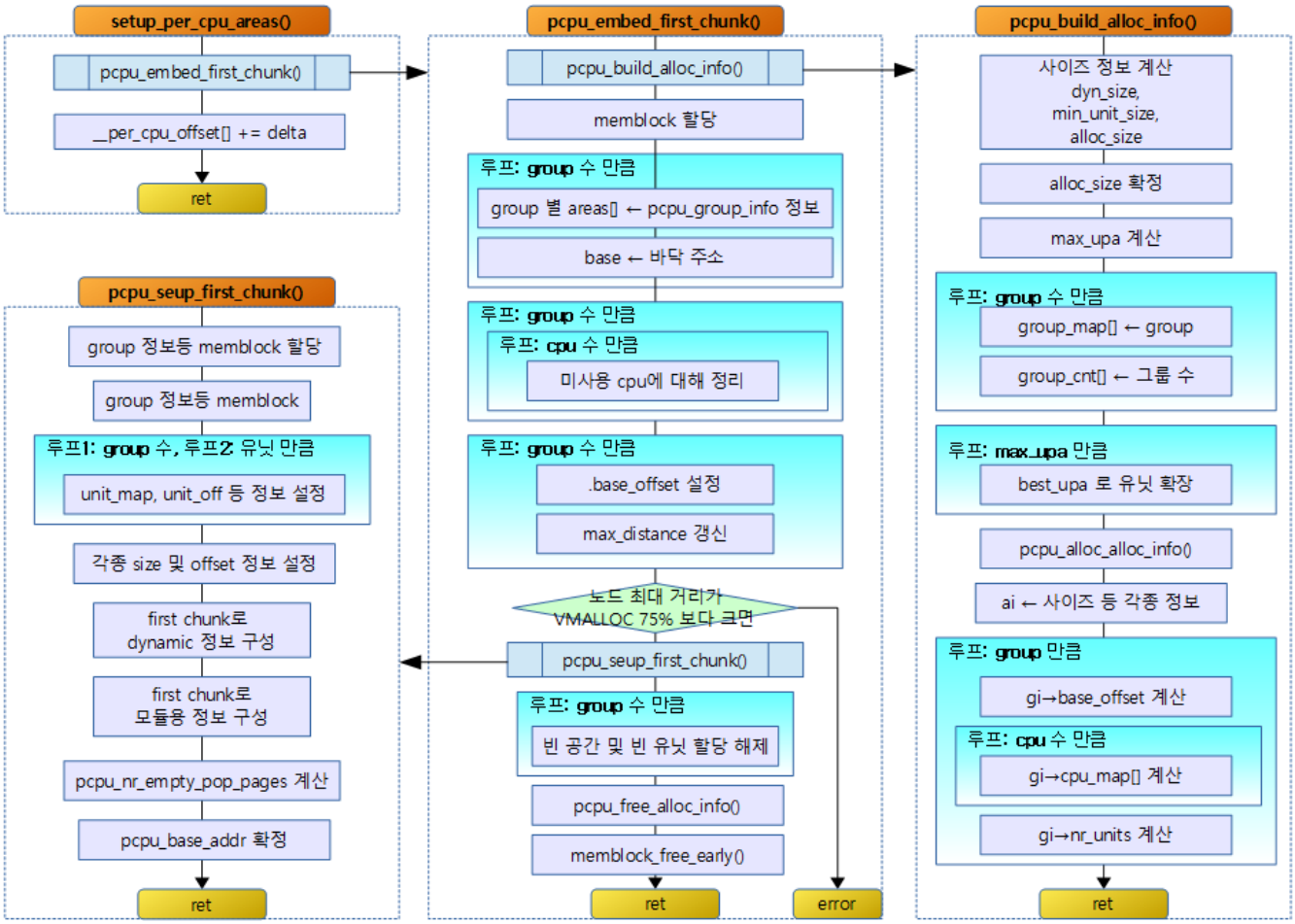
Per-cpu -2-(initialize)

2016-02-16 (http://jake.dothome.co.kr/setup_per_cpu_areas/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Per-cpu -2-(initialize)

The flowchart below is based on SMP or NUMA systems.



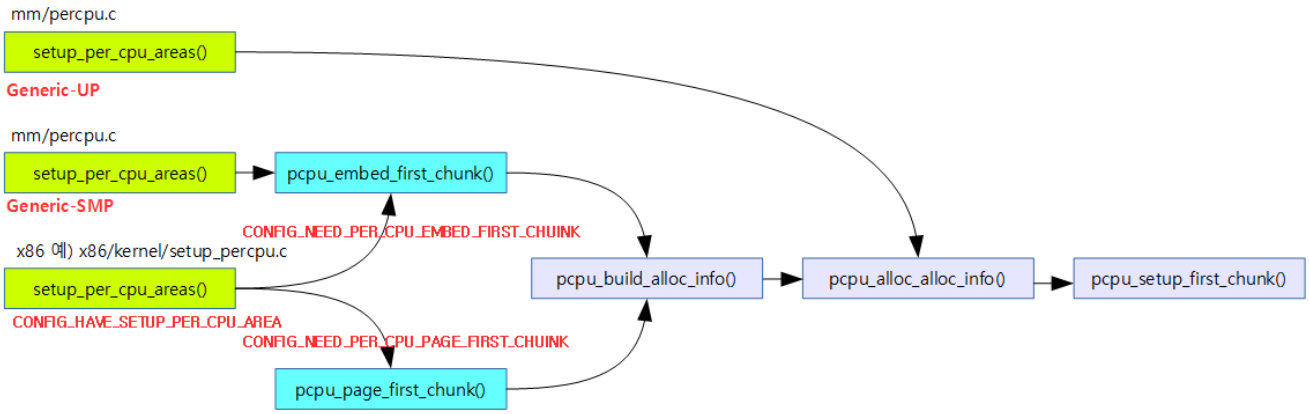
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-1a.png)

3 per-CPU implementations based on the number of cores

There are three implementations of the setup_per_cpu_areas() function to be used for per-cpu setup.

- UP Generic:
  - Common setup_per_cpu_areas() implementations that work in the UP system
- SMP Generic:
  - Common setup_per_cpu_areas() implementations that work in SMP systems
- Architecture-specific implementation:

- It uses CONFIG_HAVE_SETUP_PER_CPU_AREA kernel options and provides a separate setup_per_cpu_areas() function for each arch directory.
- ia64, sparc64, powerpc64, x86, and arm64 architectures that support NUMA use a separate per-cpu native implementation.
- The reason for using this separate and unique implementation is to support the fact that the architecture can also configure thousands of CPUs, and there are systems that also use NUMA designs and sparse memory, so different node configurations have different memory locations for allocating per-cpu data.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-24.png)

1) Per-cpu setup for UP Generic

setup_per_cpu_areas() – UP Generic

mm/percpu.c

```
/*
 * UP percpu area setup.
 *
 * UP always uses km-based percpu allocator with identity mapping.
 * Static percpu variables are indistinguishable from the usual static
 * variables and don't require any special preparation.
 */
```

```
void __init setup_per_cpu_areas(void)
{
        const size_t unit_size =
                roundup_pow_of_two(max_t(size_t, PCPU_MIN_UNIT_SIZE,
                                         PERCPU_DYNAMIC_RESERVE));
        struct pcpu_alloc_info *ai;
        void *fc;

        ai = pcpu_alloc_alloc_info(1, 1);
        fc = memblock_alloc_from_nopanic(unit_size,
                                         PAGE_SIZE,
                                         __pa(MAX_DMA_ADDRESS));
        if (!ai || !fc)
                panic("Failed to allocate memory for percpu areas.");
        /* kmemleak tracks the percpu allocations separately */
        kmemleak_free(fc);

        ai->dyn_size = unit_size;
        ai->unit_size = unit_size;
        ai->atom_size = unit_size;
        ai->alloc_size = unit_size;
        ai->groups[0].nr_units = 1;
        ai->groups[0].cpu_map[0] = 0;

```

```
25          if (pcpu_setup_first_chunk(ai, fc) < 0)
26                  panic("Failed to initialize percpu areas.");
27          pcpu_free_alloc_info(ai);
28  }
```

Prepare a per-cpu allocator for the UP system. The per-cpu data type for UP systems does not mean much in terms of performance improvement, but it prepares a first chunk consisting of 1 group (node) and 1 unit.

- In code lines 3~5, the unit size is determined by the largest size compared to the per-cpu minimum unit size of PCPU_MIN_UNIT_SIZE (32K) and the dynamic area size of PERCPU_DYNAMIC_RESERVE (32bit=20K, 64bit=28K).
- In line 9 of the code, the allocation information to be used temporarily to create the first chunk is assigned to a structure pcpu_alloc_info to 1 group (node) and 1 unit for configuration.
- In lines 10~14 of the code, use memblock to allocate 1 unit.
- After setting the allocation information in lines 18~26 of the code, call the pcpu_setup_first_chunk() function to construct the first chunk.
- In line 27 of code, unassign the allocation information used to create the first chunk.

2) Per-cpu setup for SMP Generic

setup_per_cpu_areas() – SMP Generic

The following is the setup_per_cpu_areas() function based on the SMP Generic implementation.

mm/percpu.c

```
01  /*
02   * Generic SMP percpu area setup.
03   *
04   * The embedding helper is used because its behavior closely resembles
05   * the original non-dynamic generic percpu area setup.  This is
06   * important because many archs have addressing restrictions and might
07   * fail if the percpu area is located far away from the previous
08   * location.  As an added bonus, in non-NUMA cases, embedding is
09   * generally a good idea TLB-wise because percpu area can piggy back
10   * on the physical linear memory mapping which uses large page
11   * mappings on applicable archs.
12   */
```

```
01  void __init setup_per_cpu_areas(void)
02  {
03          unsigned long delta;
04          unsigned int cpu;
05          int rc;
06
07          /*
08           * Always reserve area for module percpu variables.  That's
09           * what the legacy allocator did.
10           */
11          rc = pcpu_embed_first_chunk(PERCPU_MODULE_RESERVE,
12                                      PERCPU_DYNAMIC_RESERVE, PAGE_SIZE, N
    ULL,
13                                      pcpu_dfl_fc_alloc, pcpu_dfl_fc_fre
    e);
14          if (rc < 0)
15                  panic("Failed to initialize percpu areas.");
16
```

```
17          delta = (unsigned long)pcpu_base_addr - (unsigned long)__per_cpu
    _start;
18          for_each_possible_cpu(cpu)
19                  __per_cpu_offset[cpu] = delta + pcpu_unit_offsets[cpu];
20  }
```

Prepare a per-cpu allocator for the SMP system.

- In code lines 11~15, configure an embed first chunk to use per-CPU data.
  - ARM32 uses the embedded type, while a few other architectures use the page-type first chunk.
  - PERCPU_MODULE_RESERVE
    - It is 8K in size to cover all the static per-cpu data used by the module.
  - PERCPU_DYNAMIC_RESERVE
    - Indicates the size at which dynamic per-CPU allocation can be made to the first chunk.
    - When setting the value, make sure that there is about 1-2 pages left.
    - The default value is slightly more capacity than before, and it looks like this:
      - 32K for 20-bit systems
      - 64K for 28-bit systems
- In code lines 17~19, the pcpu_base_addr and __per_cpu_start may be different, and delta occurs at this time, which applies both delta to the __per_cpu_offset[] value.

3) Architecture-specific implementation – for ARM64 NUMA

ARM64 systems now support NUMA systems by default. Even if it is a non-NUMA system, use a NUMA configuration that uses one node.

setup_per_cpu_areas() – ARM64

arch/arm64/mm/numa.c

```
01  void __init setup_per_cpu_areas(void)
02  {
03          unsigned long delta;
04          unsigned int cpu;
05          int rc;
06
07          /*
08           * Always reserve area for module percpu variables.  That's
09           * what the legacy allocator did.
10           */
11          rc = pcpu_embed_first_chunk(PERCPU_MODULE_RESERVE,
12                                      PERCPU_DYNAMIC_RESERVE, PAGE_SIZE,
13                                      pcpu_cpu_distance,
14                                      pcpu_fc_alloc, pcpu_fc_free);
15          if (rc < 0)
16                  panic("Failed to initialize percpu areas.");
17
18          delta = (unsigned long)pcpu_base_addr - (unsigned long)__per_cpu
    _start;
19          for_each_possible_cpu(cpu)
20                  __per_cpu_offset[cpu] = delta + pcpu_unit_offsets[cpu];
21  }
```

Prepare a per-cpu allocator for ARM64 systems. The code is identical to SMP Generic, except that a pcpu_cpu_distance is added to handle NUMA.

Configuring the Embed First Chunk

pcpu_embed_first_chunk()

In this function, the first chunk is constructed in an embed way that allows you to use a huge page when you create it. This information is then used to create additional chunks. When it was first designed, it was created in bootmem, but now it is assigned to an early memory allocator called memblock.

- @reserved_size:
  - Specifies the size of the reserved area to be allocated as a static per-CPU for module use.
  - On ARM, if the CONFIG_MODULE is set, it will be PERCPU_MODULE_REWSERVE (8K), otherwise it will be 0.
- @dyn_size:
  - Specify the dynamic area size so that CPU-PER data can be dynamically allocated.
  - On ARM, it makes as many as PERCPU_DYNAMIC_RESERVER (32bit=20K, 64bit=28K).
- @atom_size:
  - For a atom_size given with the minimum allocation size, an allocation area of the nth order of 2 is created.
  - ARM uses PAGE_SIZE (4K).
- @cpu_distance_fn:
  - It is a function pointer to determine the distance between CPUs, and is used when supporting NUMA architectures.
  - In the NUMA architecture, nodes have different memory access times from each other.
  - Since the configuration between the node and the CPU is complex for each NUMA system implementation, this function is provided by the vendor that provides the NUMA system.
  - According to the implementation of the NUMA architecture, the distance values such as LOCAL_DISTANCE(10), REMOTE_DISTANCE(20), etc., are known between CPUs.
  - In this function, this distance value is used to distinguish groups (nodes).
  - In ARM32 it is null.
- @alloc_fn:
  - Specifies the function to which the per-cpu page will be assigned.
  - If null is used, it uses the vmalloc area.
- @free_fn:
  - Specifies the function to free the per-cpu page.
  - null frees it from the vmalloc region.

mm/percpu.c -1/2-

```
01 #if defined(BUILD_EMBED_FIRST_CHUNK)
02 /**
03  * pcpu_embed_first_chunk - embed the first percpu chunk into bootmem
04  * @reserved_size: the size of reserved percpu area in bytes
```

```
 * @dyn_size: minimum free size for dynamic allocation in bytes
 * @atom_size: allocation atom size
 * @cpu_distance_fn: callback to determine distance between cpus, optional
 * @alloc_fn: function to allocate percpu page
 * @free_fn: function to free percpu page
 *
 * This is a helper to ease setting up embedded first percpu chunk and
 * can be called where pcpu_setup_first_chunk() is expected.
 *
 * If this function is used to setup the first chunk, it is allocated
 * by calling @alloc_fn and used as-is without being mapped into
 * vmalloc area.  Allocations are always whole multiples of @atom_size
 * aligned to @atom_size.
 *
 * This enables the first chunk to piggy back on the linear physical
 * mapping which often uses larger page size.  Please note that this
 * can result in very sparse cpu->unit mapping on NUMA machines thus
 * requiring large vmalloc address space.  Don't use this allocator if
 * vmalloc space is not orders of magnitude larger than distances
 * between node memory addresses (ie. 32bit NUMA machines).
 *
 * @dyn_size specifies the minimum dynamic area size.
 *
 * If the needed size is smaller than the minimum or specified unit
 * size, the leftover is returned using @free_fn.
 *
 * RETURNS:
 * 0 on success, -errno on failure.
 */
```

```
int __init pcpu_embed_first_chunk(size_t reserved_size, size_t dyn_size,
                                  size_t atom_size,
                                  pcpu_fc_cpu_distance_fn_t cpu_distance_fn,
                                  pcpu_fc_alloc_fn_t alloc_fn,
                                  pcpu_fc_free_fn_t free_fn)
{
        void *base = (void *)ULONG_MAX;
        void **areas = NULL;
        struct pcpu_alloc_info *ai;
        size_t size_sum, areas_size;
        unsigned long max_distance;
        int group, i, highest_group, rc;

        ai = pcpu_build_alloc_info(reserved_size, dyn_size, atom_size,
                                   cpu_distance_fn);
        if (IS_ERR(ai))
                return PTR_ERR(ai);

        size_sum = ai->static_size + ai->reserved_size + ai->dyn_size;
        areas_size = PFN_ALIGN(ai->nr_groups * sizeof(void *));

        areas = memblock_alloc_nopanic(areas_size, SMP_CACHE_BYTES);
        if (!areas) {
                rc = -ENOMEM;
                goto out_free;
        }

        /* allocate, copy and determine base address & max_distance */
        highest_group = 0;
        for (group = 0; group < ai->nr_groups; group++) {
                struct pcpu_group_info *gi = &ai->groups[group];
                unsigned int cpu = NR_CPUS;
                void *ptr;

                for (i = 0; i < gi->nr_units && cpu == NR_CPUS; i++)
                        cpu = gi->cpu_map[i];
```

```
37                  BUG_ON(cpu == NR_CPUS);
38
39                  /* allocate space for the whole group */
40                  ptr = alloc_fn(cpu, gi->nr_units * ai->unit_size, atom_s
    ize);
41                  if (!ptr) {
42                          rc = -ENOMEM;
43                          goto out_free_areas;
44                  }
45                  /* kmemleak tracks the percpu allocations separately */
46                  kmemleak_free(ptr);
47                  areas[group] = ptr;
48
49                  base = min(ptr, base);
50                  if (ptr > areas[highest_group])
51                          highest_group = group;
52          }
53          max_distance = areas[highest_group] - base;
54          max_distance += ai->unit_size * ai->groups[highest_group].nr_uni
    ts;
55
56          /* warn if maximum distance is further than 75% of vmalloc space
    */
57          if (max_distance > VMALLOC_TOTAL * 3 / 4) {
58                  pr_warn("max_distance=0x%lx too large for vmalloc space
    0x%lx\n",
59                                  max_distance, VMALLOC_TOTAL);
60  #ifdef CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK
61                  /* and fail if we have fallback */
62                  rc = -EINVAL;
63                  goto out_free_areas;
64  #endif
65          }
```

- With the argument values given in line 14~17 of the code, the AI (pcpu_alloc_info Struct) value is constructed and created. pcpu_alloc_info The entire struct field is set and returned. All pcpu_group_info structure information in the pcpu_alloc_info structure is also configured, and the cpu_map[ ] is also configured.
- Obtain the size_sum at line 19 of code. size_sum may be different from unit_size. When a atom_size uses 4K pages, the unit_size and size_sum are the same. If the atom_size uses 1M, 2M, 16M, etc., the unit_size is usually larger than the size_sum.
  - rpi2: size_sum = 0xb000 = 0x3ec0 + 0x2000 + 0x5140
  - In the 0x5140 of dyn_size, 0x140 is added by size_sum's page alignment performance.
- In lines 20~26 of code, the addresses of the areas where the per-cpu data will be located for each group (node) are managed in an array called areas[ ], which is allocated by memblock. The area_size is the size required by the address (void *) size of the number of groups (nodes) and is sorted down to page size.
  - It loops around the number of groups and allocates a per-CPU data area from the memblock by the number of units in each group (AI->nr_units) * the unit size (unit_size). The first per-CPU area is created through a memblock before a formal memory manager such as a buddy system is running, i.e., it is created in the low-mem area without allocating it to each group (node) memory.
- Traversing the number of groups in code lines 30~37 to find out which units in the group are unit->cpu unmapped CPUs.
  - Units that are not mapped to cpu_map[] for the first time are initialized to NR_CPUS values.

- In line 40~44 of the code, the allocation size (number of units in the group * unit size) is assigned by aligning it to the atom_size through the allocator function received as an argument.
  - During bootup, the memblock is allocated to the allocator function passed as an argument.
    - generic: pcpu_dfl_fc_alloc()
    - arm64: pcpu_fc_alloc()
- If the CONFIG_DEBUG_KMEMLEAK kernel option is set in line 46 of code, it is called to analyze for memory leaks.
- In line 47 of code, in the areas[ ] array area, write down the base address assigned to the per-CPU data area of that group (node).
- On line 49 of the code, remember the smallest address assigned to each group. The initial value of the base is ULONG_MAX, so it must be updated the first time.
- Calculate the max_distance from code lines 53~65 to check if it exceeds 75% of the VMALLOC_TOTAL, and output a warning if it is exceeded. It then goes to the out_free_areas: label to free up the memory that has already been allocated and return an error. The additional chunks are spaced apart by the same as the group (node) offset when the first chunk was created, which creates a constraint that requires memory allocation and mapping to the vmalloc area, which prevents the use of embed when creating the first chunk if it exceeds a maximum of 75% of the vmalloc area. Note that when using the embed method, it is designed to reduce the probability of failure for additional chunk allocations by requiring the allocation of vmalloc areas to be used from the opposite side, i.e., from top to bottom, in order to maintain spacing between each group.
  - In other architectures that use CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK kernel options, if this function fails and escapes, it is prepared again in a page format.
  - The max_distance contains the largest address of all the group's base_offset and adds to it by the size of the unit.

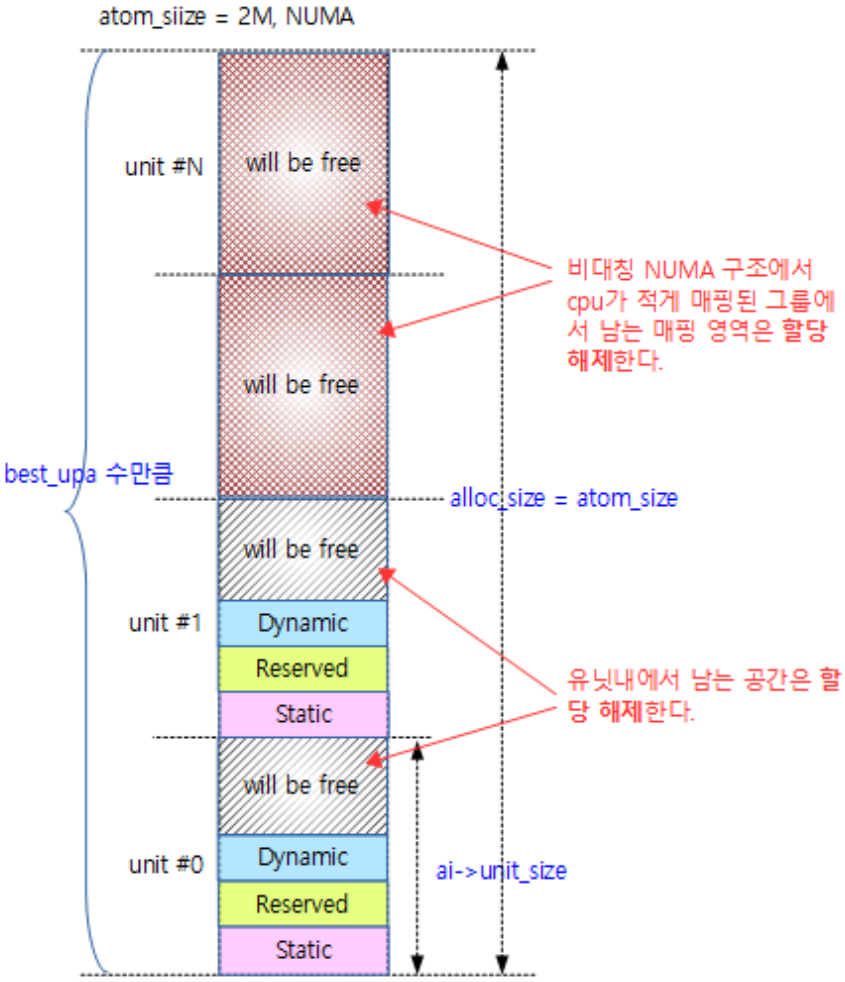
mm/percpu.c -2/2-

```
01          /*
02           * Copy data and free unused parts.  This should happen after al
    l
03           * allocations are complete; otherwise, we may end up with
04           * overlapping groups.
05           */
06          for (group = 0; group < ai->nr_groups; group++) {
07                  struct pcpu_group_info *gi = &ai->groups[group];
08                  void *ptr = areas[group];
09
10                  for (i = 0; i < gi->nr_units; i++, ptr += ai->unit_size)
    {
11                          if (gi->cpu_map[i] == NR_CPUS) {
12                                  /* unused unit, free whole */
13                                  free_fn(ptr, ai->unit_size);
14                                  continue;
15                          }
16                          /* copy and return the unused part */
17                          memcpy(ptr, __per_cpu_load, ai->static_size);
18                          free_fn(ptr + size_sum, ai->unit_size - size_su
    m);
19                  }
20          }
```

```
21
22          /* base address is now known, determine group base offsets */
23          for (group = 0; group < ai->nr_groups; group++) {
24                  ai->groups[group].base_offset = areas[group] - base;
25          }
26
27          pr_info("Embedded %zu pages/cpu @%p s%zu r%zu d%zu u%zu\n",
28                  PFN_DOWN(size_sum), base, ai->static_size, ai->reserved_
    size,
29                  ai->dyn_size, ai->unit_size);
30
31          rc = pcpu_setup_first_chunk(ai, base);
32          goto out_free;
33
34  out_free_areas:
35          for (group = 0; group < ai->nr_groups; group++)
36                  if (areas[group])
37                          free_fn(areas[group],
38                                  ai->groups[group].nr_units * ai->unit_si
    ze);
39  out_free:
40          pcpu_free_alloc_info(ai);
41          if (areas)
42                  memblock_free_early(__pa(areas), areas_size);
43          return rc;
44  }
45  #endif /* BUILD_EMBED_FIRST_CHUNK */
```

- In line 6~15 of the code, loop around the number of groups and put the area allocation address of each group in the ptr, and if it is a unit that is not unit->cpu mapped, release it by the unit size in the memblock area corresponding to that unit.
- In lines 17~18 of code, copy the static per-CPU data located in _ _per_cpu_load to the address corresponding to the unit and free the size from the memblock area, excluding the size copied within the unit, i.e., the amount of unused space
  - If you're using a small page for a unit configuration, there's no unnecessary space, so there's no area to remove.
- In lines 23~25 of the code, set the base offset value for each group.
- Output the first chunk configuration information from lines 27~29 of the code.
- In line 31~32 of the code, set the management information for the first chunk to various global variables, and go to out_free: label to free the information (AI) that was created temporarily.
- In code lines 34~38, out_free_areas:label. Uses the number of groups to get the addresses of the places where the per-cpu data regions were allocated via areas[] and deallocates those areas from the memblock.
- In code lines 39~43, out_free:label. Delete all management information used to create per-cpu data, and allocate areas information from the memblock area.
  - During bootup, the allocator function passed as an argument deallocates the memblock.
    - generic: pcpu_dfl_fc_free()
    - arm64: pcpu_fc_free()

The following figure shows the deallocating of empty and unused unit space within all units of the per-cpu first chunk.



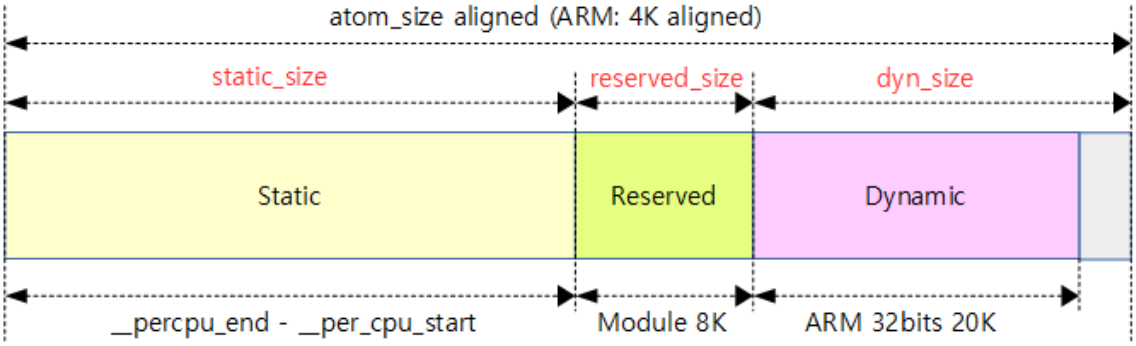
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-10.png)

Build with per-cpu allocation information

Calculate the size of the unit that will be used for one assignment and learn how to divide the area inside the unit.

Dividing the area within the unit

The following illustration shows the division of three areas inside a unit of First Chunk.



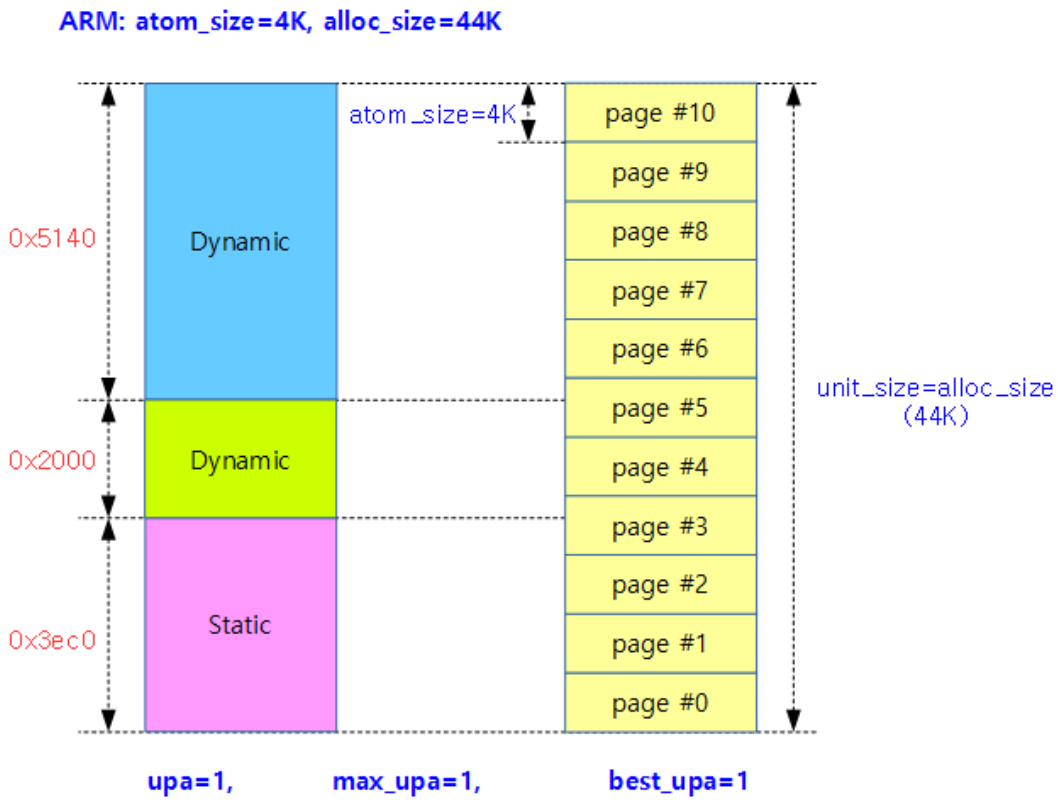
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-3a.png)

UPA(Unit Per Alloc) 산출

- The maximum number of units to be used in a single assignment. Temporarily calculate max_upa and best_upa values to obtain optimal UPA.
  - If the allocation size is 1 page, 1 unit usually requires tens of KB, so if you are assigned 1 page, you cannot include <> unit.
  - The ARM32 and ARM64 defaults use a one-page allocation, so upa always yields 1.
- The reason for getting the upa is that every time a huge page is allocated to a huge page in a system that uses huge pages, it gets a large size (1, 2M, ...) is assigned, so the number of effective units in this space is calculated.
  - On a system with 8 CPUs, if you allocate 1 huge page and divide it into 256MB increments, the entire unit will be placed on one huge page, so the UPA value is just 8.
- In addition, in a NUMA system, memory nodes are placed on each node, and per-cpu data must be accessed from each node for faster performance, so the upa must be obtained appropriately in order for the unit layout per one allocated memory to be effectively deployed.

As shown in the figure below, if one page is used as the minimum allocation unit, the units cannot fit on a single page. Rather, it is a situation where multiple allocations are needed for a single unit. In this case, the UPA value is 1 as the minimum.

- ARM32 e.g. atom_size=1 page (4K), alloc_size=0xb000
  - upa=1



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-20a.png)

max_upa

- The minimum allocation unit is the maximum number of units that can be included per allocation unit if it supports a large size, such as a huge page.
- For a 4K page, a single allocation size starting with the nth order of 2 contains the maximum number of units that can be assigned.
- The ARM32 and ARM64 mainstream kernels have not yet enforced the use of huge pages as the minimum allocation unit. Always use page-only.
- e.g. alloc_size=2M, min_unit_size=0xb000 and upa=46
    - Final max_upa=32

best_upa

- When using multi-node memory, such as a NUMA system, memory allocation using max_upa to each node is wasteful. Therefore, the max_upa is divided into nodes and the best number of units is calculated for use.
- In the case of an asymmetric NUMA system, the number of unused units should not exceed 1/3 of the actual number of units to be used (equal to the number of possible CPUs) in order to avoid wasting more units than necessary.
- Symmetrical NUMA system e.g. alloc_size=2M, min_unit_size=0xb000, max_upa=32, nr_cpu=8, nr_group=2
    - Since there are 4 nodes per node, best_upa=4.
- Asymmetric NUMA system e.g. alloc_size=2M, min_unit_size=0xb000, max_upa=32, nr_cpu=8+1, nr_group=2
    - If you try UPA in the order of 32, 16, 8, 4, 2, 1, and so on, the number of assignments will be 2, 2, 2, 3, 5, and 9, respectively. And the number of unused units becomes 55, 23, 7, 3, 1, 0. It can be seen that the best_upa that does not exceed 9, which is 3/1 of the units to be used (number of CPUs = 3), will be 4.
    - Number of allocations = accumulate DIV_ROUND_UP by the number of groups (number of CPU in that group / UPA).
        - DIV_ROUND_UP(8 / 32) + DIV_ROUND_UP(1 / 32) = 1 + 1 = 2
        - DIV_ROUND_UP(8 / 16) + DIV_ROUND_UP(1 / 16) = 1 + 1 = 2
        - DIV_ROUND_UP(8 / 8) + DIV_ROUND_UP(1 / 8) = 1 + 1 = 2
        - DIV_ROUND_UP(8 / 4) + DIV_ROUND_UP(1 / 4) = 2 + 1 = 3
        - DIV_ROUND_UP(8 / 2) + DIV_ROUND_UP(1 / 2) = 4 + 1 = 5
        - DIV_ROUND_UP(8 / 1) + DIV_ROUND_UP(1 / 1) = 8 + 1 = 9
    - Wasted = Allocation * UPA – Number of units to use (possible CPU = 9).
        - 2 * 32 – 9 = 55 (55 unused units are judged wasted by more than 9/1 of the 3 CPUs)
        - 2 * 16 – 9 = 23 (23 unused units are judged wasted by more than 9/1 of the 3 CPUs)
        - 2 * 8 – 9 = 7 (57 unused units are judged wasted by more than 9/1 of the 3 CPUs)
        - 3 * 4 – 9 = 3 (3 unused units do not exceed 9/1 of the 3 CPUs, so OK)
        - 5 * 2 – 9 = 1 (1 unused units do not exceed 9/1 of the 3 CPUs, so OK)
        - 9 * 1 – 9 = 0 (0 unused units do not exceed 9/1 of the 3 CPUs, so OK)

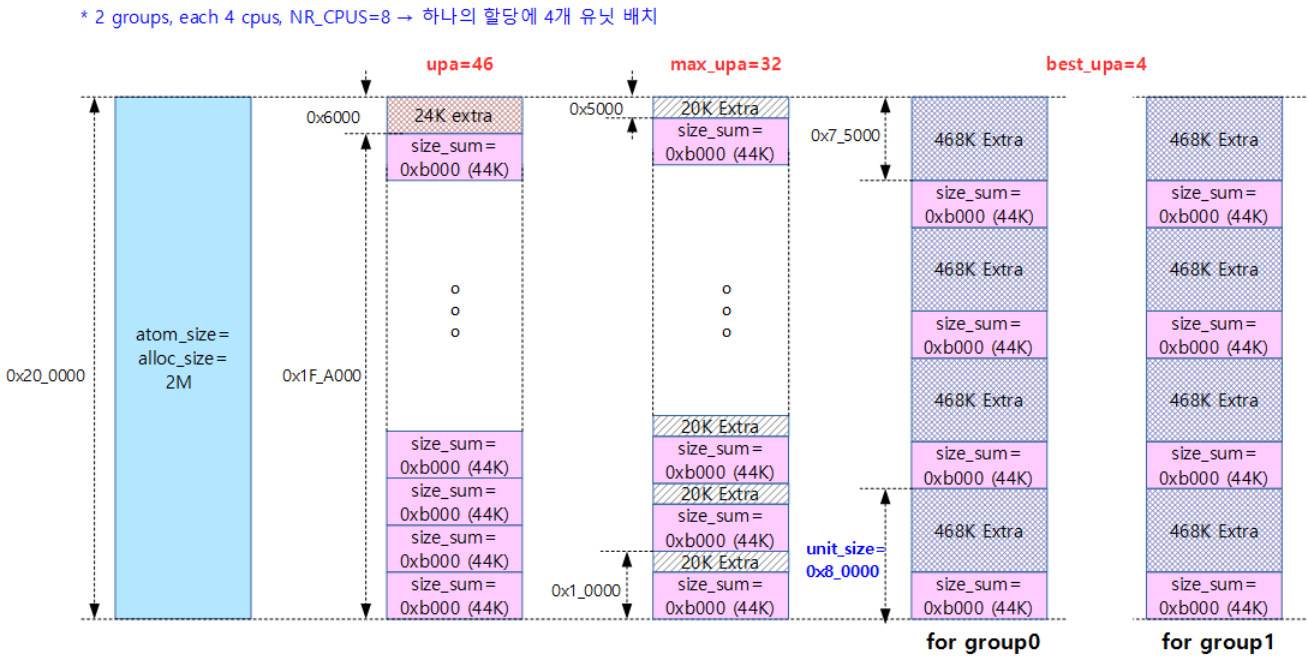
max_upa and best_upa output

The calculation steps consist of three steps:

- Step 1) First UPA = Allocation Size / Minimum Unit Size to Assign (min_unit_size)
- Step 2) max_upa = huge Adjust the UPA so that there is no remainder when dividing the page allocation unit by UPA
- Step 3) best_upa = Find and adjust the minimum number of allocations to the extent that the unused units do not exceed 1/3 of the actual units for upa adjustment for the unbalanced NUMA system.

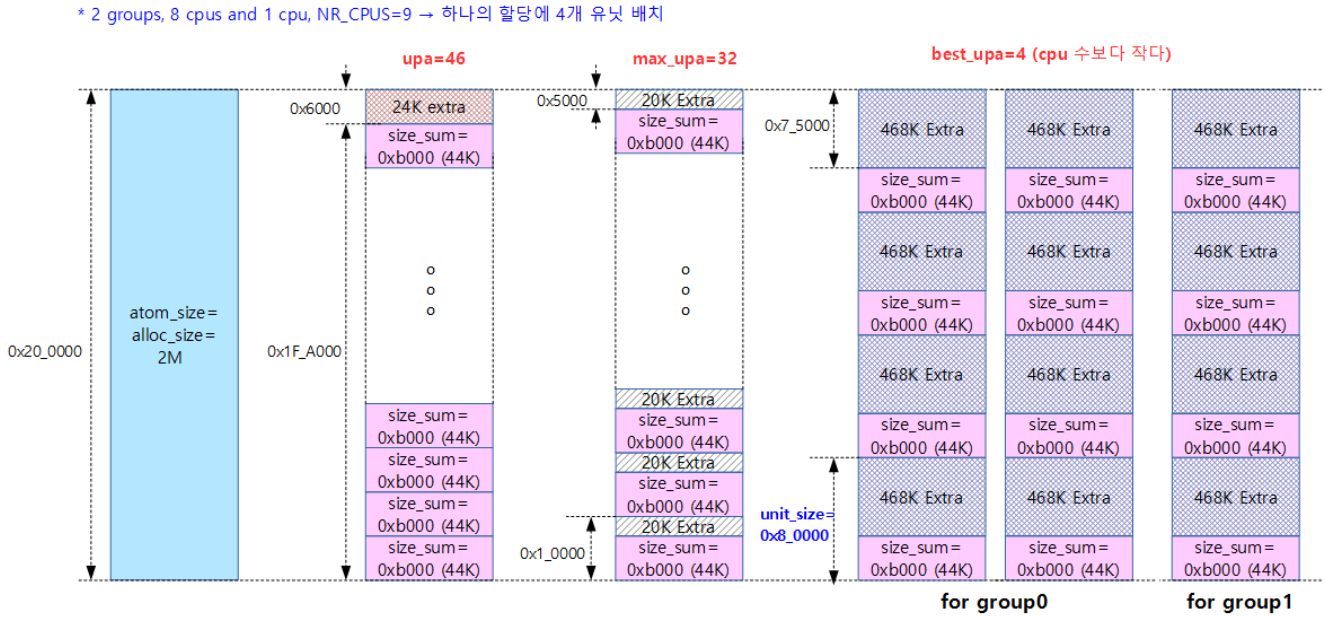
The following figure shows the upa, max_upa, and best_upa calculated from a symmetrical NUMA system using a 2M allocation in operation after they have been calculated.

- The extra space is not used as per-CPU data space, but is deallocated and returned to the buddy system.



The following figure shows the upa, max_upa, and best_upa calculated from a symmetrical NUMA system using a 2M allocation in operation after they have been calculated.

- If one group used 8 CPUs and the other used 1 CPU, 3 allocations would be used, and the UPA (best_upa) value would be 4.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-22.png)

pcpu_build_alloc_info()

mm/percpu.c -1/4-

```
01 /* pcpu_build_alloc_info() is used by both embed and page first chunk */
02 #if defined(BUILD_EMBED_FIRST_CHUNK) || defined(BUILD_PAGE_FIRST_CHUNK)
03 /**
04  * pcpu_build_alloc_info - build alloc_info considering distances betwee
   n CPUs
05  * @reserved_size: the size of reserved percpu area in bytes
06  * @dyn_size: minimum free size for dynamic allocation in bytes
07  * @atom_size: allocation atom size
08  * @cpu_distance_fn: callback to determine distance between cpus, option
   al
09  *
10  * This function determines grouping of units, their mappings to cpus
11  * and other parameters considering needed percpu size, allocation
12  * atom size and distances between CPUs.
13  *
14  * Groups are always multiples of atom size and CPUs which are of
15  * LOCAL_DISTANCE both ways are grouped together and share space for
16  * units in the same group.  The returned configuration is guaranteed
17  * to have CPUs on different nodes on different groups and >=75% usage
18  * of allocated virtual address space.
19  *
20  * RETURNS:
21  * On success, pointer to the new allocation_info is returned.  On
22  * failure, ERR_PTR value is returned.
23  */
```

```
01 static struct pcpu_alloc_info * __init pcpu_build_alloc_info(
02                                 size_t reserved_size, size_t dyn_size,
03                                 size_t atom_size,
04                                 pcpu_fc_cpu_distance_fn_t cpu_distance_f
   n)
05 {
06         static int group_map[NR_CPUS] __initdata;
07         static int group_cnt[NR_CPUS] __initdata;
08         const size_t static_size = __per_cpu_end - __per_cpu_start;
09         int nr_groups = 1, nr_units = 0;
10         size_t size_sum, min_unit_size, alloc_size;
11         int upa, max_upa, uninitialized_var(best_upa);  /* units_per_all
   oc */
```

```
12          int last_allocs, group, unit;
13          unsigned int cpu, tcpu;
14          struct pcpu_alloc_info *ai;
15          unsigned int *cpu_map;
16
17          /* this function may be called multiple times */
18          memset(group_map, 0, sizeof(group_map));
19          memset(group_cnt, 0, sizeof(group_cnt));
20
21          /* calculate size_sum and ensure dyn_size is enough for early al
    loc */
22          size_sum = PFN_ALIGN(static_size + reserved_size +
23                               max_t(size_t, dyn_size, PERCPU_DYNAMIC_EARLY
    _SIZE));
24          dyn_size = size_sum - static_size - reserved_size;
25
26          /*
27           * Determine min_unit_size, alloc_size and max_upa such that
28           * alloc_size is multiple of atom_size and is the smallest
29           * which can accommodate 4k aligned segments which are equal to
30           * or larger than min_unit_size.
31           */
32          min_unit_size = max_t(size_t, size_sum, PCPU_MIN_UNIT_SIZE);
```

Reserve size, dynamic size, atom size, and interCPU distance factors to prepare per-CPU chunk allocation information. Most atom_size use 4K pages to induce alignment of the allocation area, and some architectures that use huge pages support 1M, 2M, 16M, and so on.

- When it was first called to create an ARM 32-bit architecture first chunk, the arguments specified are:
  - reserved_size=8K, dyn_size=20K, atom_size=4K
  - Most atom_size use 4K pages to align the allocation area, and some architectures that use huge pages support 1M, 2M, 16M, and so on.
- In line 8 of code, the static size is calculated by subtracting the __per_cpu_start from the __per_cpu_end, which is determined at compile time.
  - rpi2 e.g. slightly different for each setting -> static_size = 0x3ec0 bytes
  - rock960 e.g. static_size = 0xdad8
- This macro is intended to get rid of the compile warning when using an uninitialized local variable in line 11 of code.
  - uninitialized_var(best_upa);
    - This macro is intended to eliminate the compile warning when using uninitialized local variables.
    - #define uninitialized_var(x) x = x
- In lines 22~23 of the code, the size_sum is the size of the static area + reserved area + dynamic area plus the size sorted by page. The dyn_size is at least PERCPU_DYNAMIC_EARLY_SIZE (0x3000=12K). The dynamic size is 32K on 20-bit and 64K on 28-bit systems.
  - rpi2 e.g. size_sum = 0xb000 = PFN_ALIGN(0xaec0)
  - rock960 example) size_sum = 0x1_7000 = PFN_ALIGN(0x17ad8)
- Rebalance the dyn_size at line 24 of code.
  - rpi2 e.g. dyn_size = 0x5140
  - rock960 e.g. dyn_size=0x7528
- In line 32 of code, limit the min_unit_size to at least PCPU_MIN_UNIT_SIZE (0x8000=32K).

- rpi2 예) min_unit_size = 0xb000 (0x3ec0(static) + 0x2000 + 0x5000(dynamic) + 0x140(align to dynamic))
- rock960 예) min_unit_size = 0x1_7000 (0xdad8(static) + 0x2000(reserved) + 0x7000(dynamic) + 0x528(align to dynamic))

mm/percpu.c -2/4-

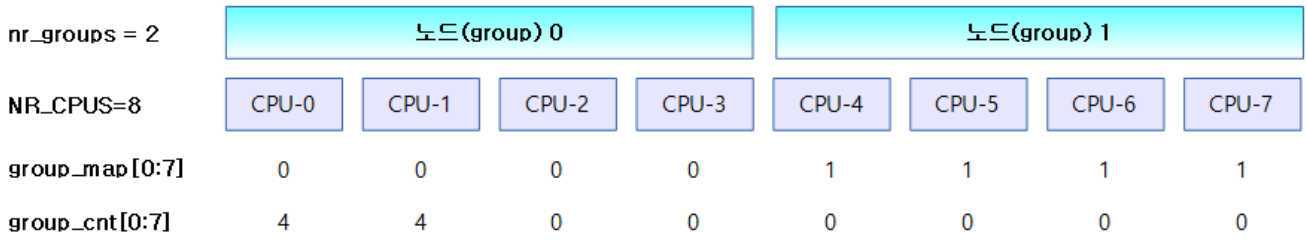
```
01          /* determine the maximum # of units that can fit in an allocation */
02          alloc_size = roundup(min_unit_size, atom_size);
03          upa = alloc_size / min_unit_size;
04          while (alloc_size % upa || (offset_in_page(alloc_size / upa)))
05                  upa--;
06          max_upa = upa;
07
08          /* group cpus according to their proximity */
09          for_each_possible_cpu(cpu) {
10                  group = 0;
11          next_group:
12                  for_each_possible_cpu(tcpu) {
13                          if (cpu == tcpu)
14                                  break;
15                          if (group_map[tcpu] == group && cpu_distance_fn &&
16                              (cpu_distance_fn(cpu, tcpu) > LOCAL_DISTANCE ||
17                               cpu_distance_fn(tcpu, cpu) > LOCAL_DISTANCE)) {
18                                  group++;
19                                  nr_groups = max(nr_groups, group + 1);
20                                  goto next_group;
21                          }
22                  }
23                  group_map[cpu] = group;
24                  group_cnt[group]++;
25          }
```

- In line 2 of code, round up the minimum allocation size in atom_size (ARM: 4K). The second argument of roundup( ) must be of the scalar type. Once the allocation size (alloc_size) has been calculated, the number of units that will be placed in this assignment must be calculated.
  - If a few other architectures support atom_size 1M, 2M, and 16M, the alloc_size is created in atom_size units.
  - rpi2: alloc_size = 0xb000
  - rock960: alloc_size=0x1_7000
- In line 3 of code, first calculate UPA, which is the number of units that can fit in an allocation.
  - If the atom_size is a small 4K, such as ARM, the upa value is always 4 because the alloc_size is always equal to or greater than 1K.
  - IN OTHER SPECIAL ARCHITECTURES, atom_size CAN BE SPECIFIED AS 1M, 2M, AND 16M, AND SO ON, IN WHICH CASE MULTIPLE UNITS CAN FIT IN A SINGLE ASSIGNMENT.
  - e.g. architecture with a atom_size of 2M: upa=46
    - upa(46)=alloc_size(2M) / min_unit_size(44K)
- Calculate max_upa in code lines 4~6 (equal to or less than UPA). The alloc_size has a chance to break out of the loop only to the nth power of 2 for the page size. The max_upa contains the

maximum number of units that can be assigned to a single allocation size that starts with the nth power of 4 for a 2K page. In other words, the max_upa is a value of 2^N.

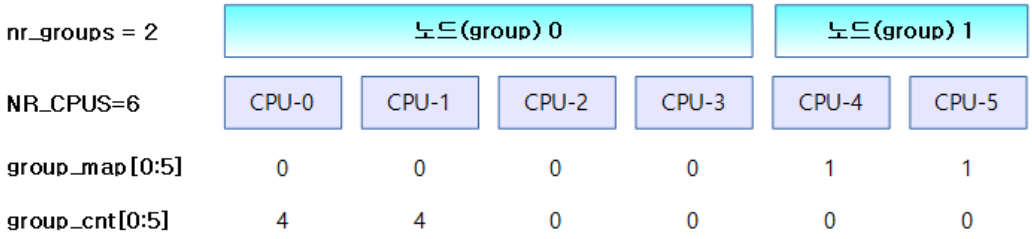
- In code lines 9~25, calculate the group information group_map[ ], group_cnt[ ], nr_groups values.
  - group_map[]
    - UMA systems, such as ARM, use a single group (node), so the value is 0.
    - In the NUMA system, groups (nodes) are grouped together as shown in the figure below.
      - If the distance value between CPUs is greater than LOCAL_DISTANCE(10), they are recognized as different groups (nodes).
  - group_cnt[]
    - In a UMA system, such as ARM, there is only one group (node), so only group_cnt[0] contains the number of CPUs.
    - In a NUMA system, the number of CPUs owned by each group (node) is included, as shown in the figure below. As shown in the figure below, the number of CPUs of 4 CPUs was entered in each of the two groups.
  - nr_groups
    - Number of groups (nodes)

The figure below shows a symmetrical NUMA system with four CPUs placed in each group (node).



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-2a.png)

The figure below shows an asymmetric NUMA system with four CPUs in the first group (nodes) and two CPUs in the second group (nodes).



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-18.png)

mm/percpu.c -3/4-

```
01 .        /*
02           * Wasted space is caused by a ratio imbalance of upa to group_c
   nt.
03           * Expand the unit_size until we use >= 75% of the units allocat
   ed.
```

```
04          * Related to atom_size, which could be much larger than the uni
    t_size.
05          */
06         last_allocs = INT_MAX;
07         for (upa = max_upa; upa; upa--) {
08                 int allocs = 0, wasted = 0;
09
10                 if (alloc_size % upa || (offset_in_page(alloc_size / up
    a)))
11                         continue;
12
13                 for (group = 0; group < nr_groups; group++) {
14                         int this_allocs = DIV_ROUND_UP(group_cnt[group],
    upa);
15                         allocs += this_allocs;
16                         wasted += this_allocs * upa - group_cnt[group];
17                 }
18
19                 /*
20                  * Don't accept if wastage is over 1/3.  The
21                  * greater-than comparison ensures upa==1 always
22                  * passes the following check.
23                  */
24                 if (wasted > num_possible_cpus() / 3)
25                         continue;
26
27                 /* and then don't consume more memory */
28                 if (allocs > last_allocs)
29                         break;
30                 last_allocs = allocs;
31                 best_upa = upa;
32         }
33         upa = best_upa;
```

- Loop from lines 7~11 to max_upa~1 and skip unless you have the nth power of 2 for the page.
  - e.g. UPA values go down to 32 (max_upa) -> 16 > 8 > 4 > 2 -> 1.
- In line 13~17 of the code, go around the loop (group = 0, 1) by the number of groups and calculate the values of this_allocs, allocs, and wasted.
  - this_allocs
    - Number of assignments required for each group
  - allocs
    - Number of assignments required for the entire group
    - Accumulate DIV_ROUND_UP by the number of groups (the number of CPU in that group / UPA).
  - wasted
    - Calculate the number of wasted units
    - Number of Assignments * Number of units to deploy per assignment (UPA) – Number of units to use (possible number of CPUs = 9).
- If the remaining units in line 24~25 exceed 1/3 of the units actually used, it will be considered wasteful and skipped.
- If the calculated allocation size in code lines 28~33 is larger than the existing allocation size, it escapes the loop and determines the best_upa.

The table below shows the application of min_unit_size=44K in a symmetrical NUMA system and the tracking of best_upa values using various conditions such as:

- X-axis condition: Assigned size
- Y-axis conditions: number of CPUs by group, number of groups, total number of CPUs

min_unit_size=0xb0000 (44K)

alloc_size	cpu	nr_group	nr_cpu	2M	1M	512K	256K	128K	64K	32K
upa				46	23	11	5	2	1	1
max_upa				32	16	8	4	2	1	1
best_upa	2	1	**2**	2	2	2	2	2	1	1
	2	2	**4**	2	2	2	2	2	1	1
	2	4	**8**	2	2	2	2	2	1	1
	4	1	**4**	4	4	4	4	2	1	1
	4	2	**8**	4	4	4	4	2	1	1
	4	4	**16**	4	4	4	4	2	1	1

(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-4b.png)

The table below also uses min_unit_size=44K in a symmetrical NUMA system and uses various conditions to track the allocs and wasted values.

- X-axis condition: UPA
- Y-Axis Condition: group

cpu=2 group=1 nr_cpu=2

upa		32	16	8	4	2	1
this_allocs		1	1	1	1	1	2
allocs	group=0	1	1	1	1	1	2
wasted	group=0	30	14	6	2	0	0

cpu=4 group=1 nr_cpu=4

upa		32	16	8	4	2	1
this_allocs		1	1	1	1	2	4
allocs	group=0	1	1	1	1	2	4
wasted	group=0	28	12	4	0	0	0

cpu=2 group=2 nr_cpu=4

upa		32	16	8	4	2	1
this_allocs		1	1	1	1	1	2
allocs	group=0	1	1	1	1	1	2
wasted	group=0	30	14	6	2	0	0
allocs	group=1	2	2	2	2	2	4
wasted	group=1	60	28	12	4	0	0

cpu=4 group=2 nr_cpu=8

upa		32	16	8	4	2	1
this_allocs		1	1	1	1	2	4
allocs	group=0	1	1	1	1	2	4
wasted	group=0	28	12	4	0	0	0
allocs	group=1	2	2	2	2	4	8
wasted	group=1	56	24	8	0	0	0

cpu=2 group=4 nr_cpu=8

upa		32	16	8	4	2	1
this_allocs		1	1	1	1	1	2
allocs	group=0	1	1	1	1	1	2
wasted	group=0	30	14	6	2	0	0
allocs	group=1	2	2	2	2	2	4
wasted	group=1	60	28	12	4	0	0
allocs	group=2	4	4	4	4	4	8
wasted	group=2	90	42	18	6	0	0
allocs	group=3	4	4	4	4	4	8
wasted	group=3	120	56	24	8	0	0

cpu=4 group=4 nr_cpu=16

upa		32	16	8	4	2	1
this_allocs		1	1	1	1	2	4
allocs	group=0	1	1	1	1	2	4
wasted	group=0	28	12	4	0	0	0
allocs	group=1	2	2	2	2	4	8
wasted	group=1	56	24	8	0	0	0
allocs	group=2	3	3	3	3	6	12
wasted	group=2	84	36	12	0	0	0
allocs	group=3	4	4	4	4	8	16
wasted	group=3	112	48	16	0	0	0

(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-5c.png)

As shown in the table below, I checked the following two cases in an asymmetric NUMA system.

- 1) group-0 has 4 CPUs, group-1 has 2 CPUs, and when max_upa=32
  - allocs=2, best_upa=4
    - There are 2 units for assignment and 4 units in one allocation space.
  - This is a case where there are 4 best_upa for the maximum number of CPUs per group.
- 2) group-0 has 8 CPUs, group-1 has 1 CPUs, and when max_upa=32
  - allocs=3, best_upa=4
    - Assign 3 units, 4 units in one allotted space.
  - Normally, there are 8 best_upa for the maximum number of CPUs per group, which leaves too much space for 1 CPU to be allocated, so the number of allocations is increased and the best_upa halved until there is no waste.

cpu={ 4, 2 } group=2 nr_cpu=6

upa		32	16	8	4	2	1
this_allocs	group=0	1	1	1	1	2	4
allocs	group=0	1	1	1	1	2	4
wasted	group=0	28	12	4	0	0	0
this_allocs	group=1	1	1	1	1	1	2
allocs	group=1	2	2	2	2	3	6
wasted	group=1	58	26	10	2	0	0

cpu={ 8, 1 } group=2 nr_cpu=9

upa		32	16	8	4	2	1
this_allocs	group=0	1	1	1	2	4	8
allocs	group=0	1	1	1	2	4	8
wasted	group=0	24	8	0	0	0	0
this_allocs	group=1	1	1	1	1	1	1
allocs	group=1	2	2	2	3	5	9
wasted	group=1	55	23	7	3	1	0

(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-23.png)

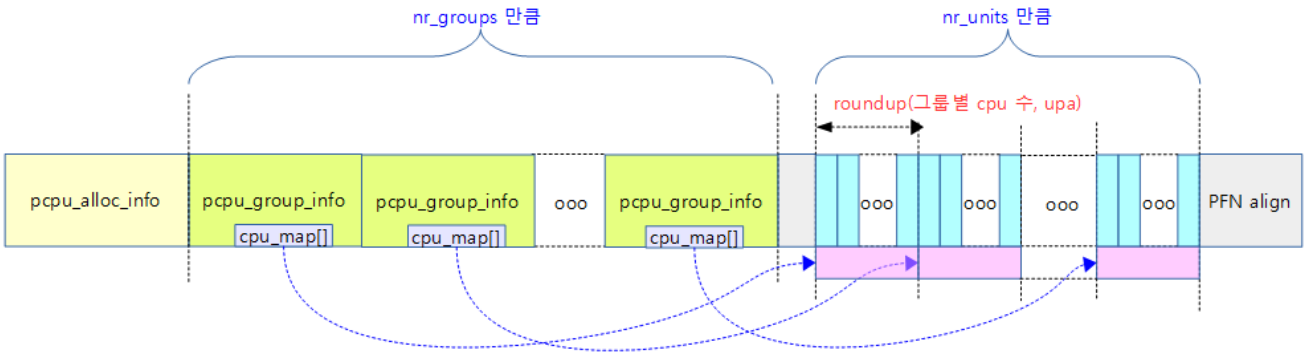
mm/percpu.c -4/4-

```
01          /* allocate and fill alloc_info */
02          for (group = 0; group < nr_groups; group++)
03                  nr_units += roundup(group_cnt[group], upa);
04
05          ai = pcpu_alloc_alloc_info(nr_groups, nr_units);
06          if (!ai)
07                  return ERR_PTR(-ENOMEM);
08          cpu_map = ai->groups[0].cpu_map;
09
10          for (group = 0; group < nr_groups; group++) {
11                  ai->groups[group].cpu_map = cpu_map;
12                  cpu_map += roundup(group_cnt[group], upa);
13          }
14
15          ai->static_size = static_size;
16          ai->reserved_size = reserved_size;
17          ai->dyn_size = dyn_size;
18          ai->unit_size = alloc_size / upa;
19          ai->atom_size = atom_size;
20          ai->alloc_size = alloc_size;
21
22          for (group = 0, unit = 0; group_cnt[group]; group++) {
23                  struct pcpu_group_info *gi = &ai->groups[group];
24
25                  /*
26                   * Initialize base_offset as if all groups are located
27                   * back-to-back.  The caller should update this to
28                   * reflect actual allocation.
29                   */
30                  gi->base_offset = unit * ai->unit_size;
31
32                  for_each_possible_cpu(cpu)
33                          if (group_map[cpu] == group)
34                                  gi->cpu_map[gi->nr_units++] = cpu;
35                  gi->nr_units = roundup(gi->nr_units, upa);
36                  unit += gi->nr_units;
37          }
38          BUG_ON(unit != nr_units);
39
40          return ai;
41  }
42  #endif /* BUILD_EMBED_FIRST_CHUNK || BUILD_PAGE_FIRST_CHUNK */
```

- In line 2~3 of the code, loop around the number of groups and calculate the nr_units. The nr_units contains the total number of units you'll create. Loop around the number of groups, rounding up the number of CPUs per group to UPA sizes and then adding up.
  - If there are two NUMA nodes and the design is asymmetrical, one group (node) uses 2 CPUs and the other group (nodes) uses 8 CPU, then the total number of nr_units is 1 + 4 = 8, assuming a UPA(best_upa) value of 4.

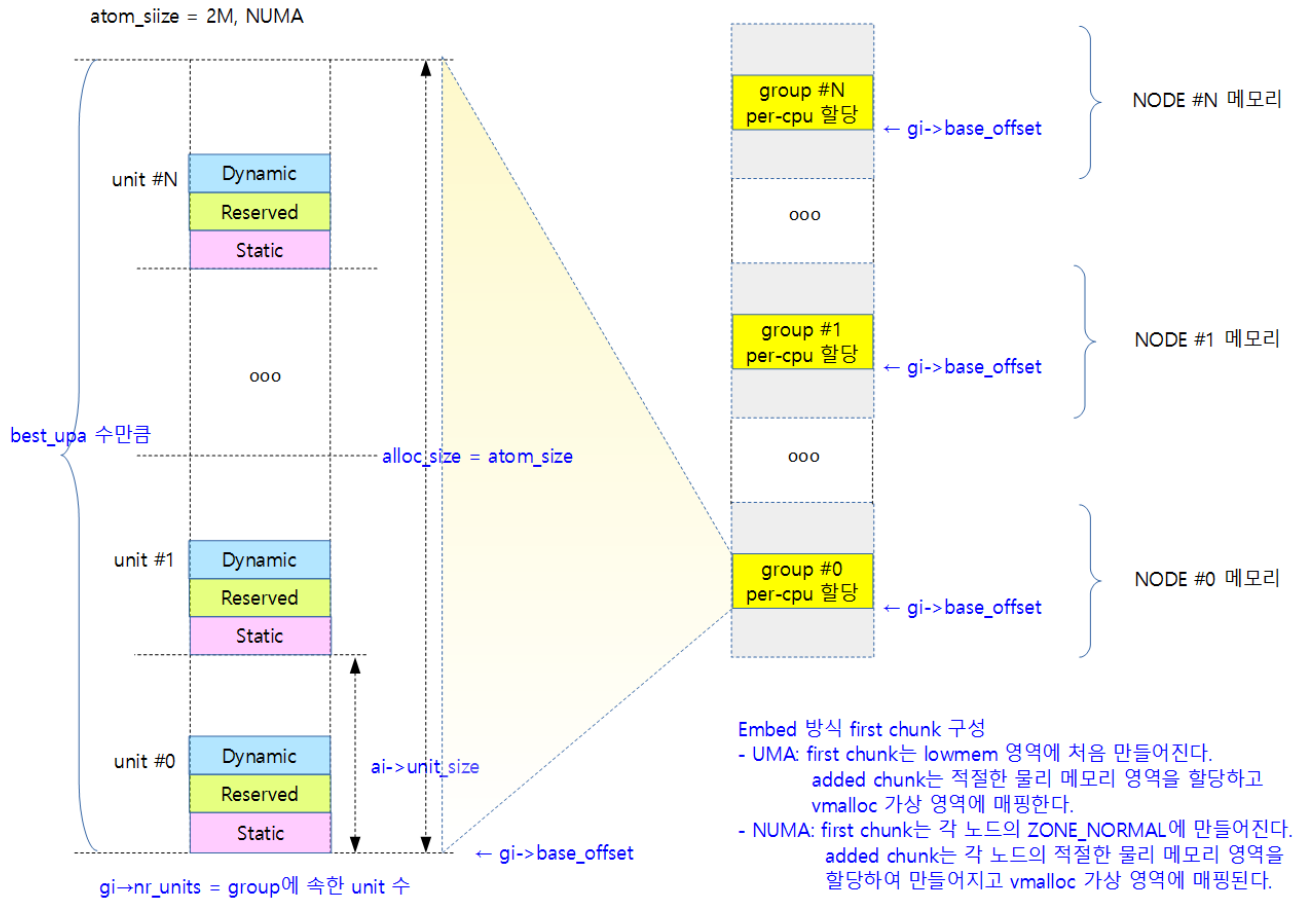
- In line 5~7 of code, create a pcpu_alloc_info struct in the memblock area and return it. Within pcpu_alloc_info structure, there is also an array of pcpu_group_info[ ] structures and an array of cpu_map[ ] within them.
- In lines 8~13 of the code, specify the cpu_map area belonging to each group in the cpu_map[ ] array for each group. Then, the number of CPUs in the group is sorted up to the nearest UPA.
    - e.g. nr_groups = 4, NR_CPUS = 16, nr_units = 4, the allocation is as follows: The mapping information is initialized to a NR_CPUS value.
    - ai->groups[0].cpu_map[0~3] -> unit->cpu mapping array of group 0
    - ai->groups[1].cpu_map[0~3] -> unit->cpu mapping array of group 1
    - ai->groups[2].cpu_map[0~3] -> unit->cpu mapping array of group 2
    - ai->groups[3].cpu_map[0~3] -> unit->cpu mapping array of group 3
- In lines 15~20 of the code, specify the size and so on in the fields inside the AI (pcpu_alloc_info structure).
- In lines 22~37 of the code, loop around the number of groups and set the base_offset, cpu_map, and nr_units fields.
    - base_offset: Base addresses by group
    - cpu_map: unit->cpu mapping value, unmapped value contains NR_CPUS value
    - nr_units: Number of units per group (including empty units that cannot be mapped in asymmetric NUMA systems)

The following figure shows the configuration of the pcpu_alloc_info structure that represents the PCPU allocation information.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-7a.png)

The figure below shows a case where a NUMA system supports huge page memory, and the per-cpu data allocation is divided among the memory of each node.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-8b.png)

The table below shows the placement of cpu_map[] in a symmetrical NUMA system with two nodes.

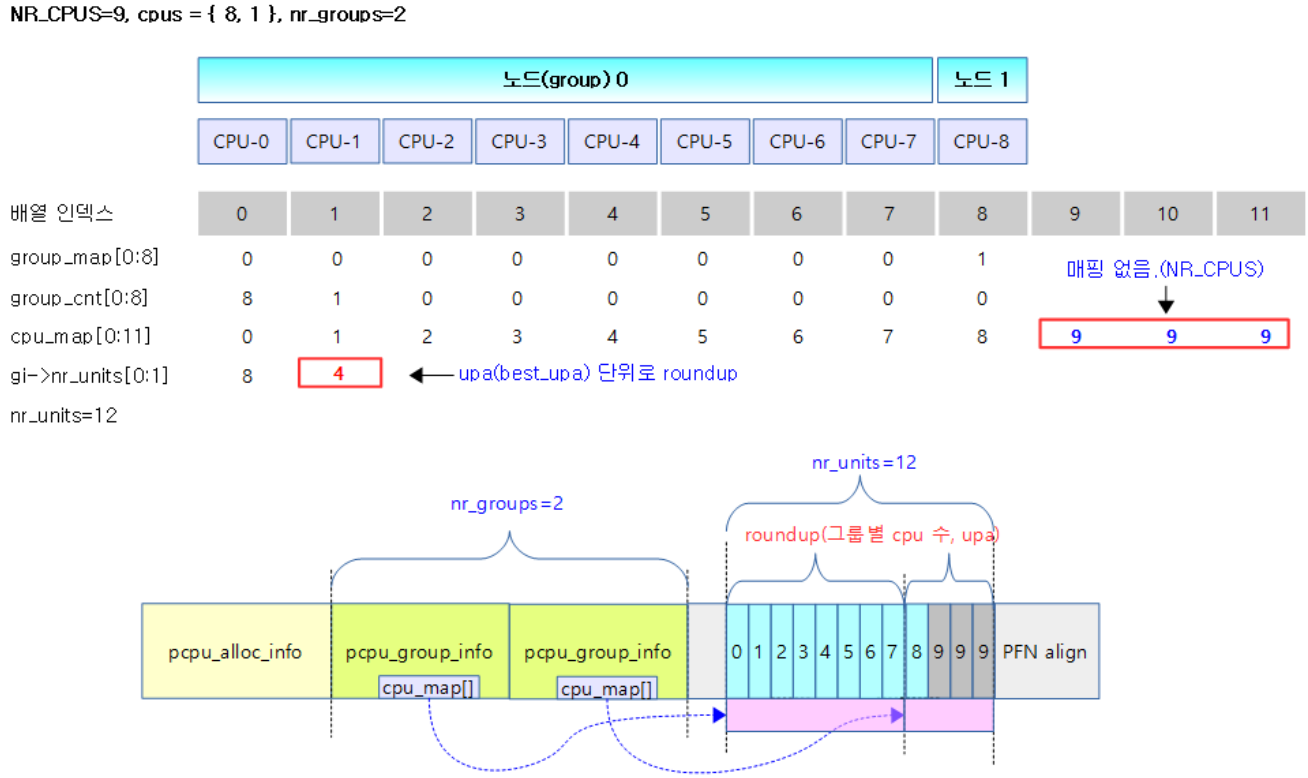
- e.g. NR_CPUS=8, cpus={ 4, 4 }, max_upa=32
  - allocs(number of allocations)=2, nr_units=8, best_upa=4

nr_groups = 2	노드(group) 0				노드(group) 1			
	CPU-0	CPU-1	CPU-2	CPU-3	CPU-4	CPU-5	CPU-6	CPU-7
배열 인덱스	0	1	2	3	4	5	6	7
group_map[0:7]	0	0	0	0	1	1	1	1
group_cnt[0:7]	4	4	0	0	0	0	0	0
cpu_map[0:7]	0	1	2	3	4	5	6	7
gi->nr_units[0:1]	4	4						

(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-9.png)

The table and figure below show the placement of cpu_map[] in an asymmetrical NUMA system with two nodes.

- e.g. NR_CPUS=9, cpus = { 8, 1 }, max_upa=32, best_upa=4
  - allocs(number of allocations)=3, nr_units=12, best_upa=4



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-22a.png)

Configuring per-CPU allocation information

pcpu_alloc_alloc_info()

mm/percpu.c

```
01 /**
02  * pcpu_alloc_alloc_info - allocate percpu allocation info
03  * @nr_groups: the number of groups
04  * @nr_units: the number of units
05  *
06  * Allocate ai which is large enough for @nr_groups groups containing
07  * @nr_units units.  The returned ai's groups[0].cpu_map points to the
08  * cpu_map array which is long enough for @nr_units and filled with
09  * NR_CPUS.  It's the caller's responsibility to initialize cpu_map
10  * pointer of other groups.
11  *
12  * RETURNS:
13  * Pointer to the allocated pcpu_alloc_info on success, NULL on
14  * failure.
15  */
```

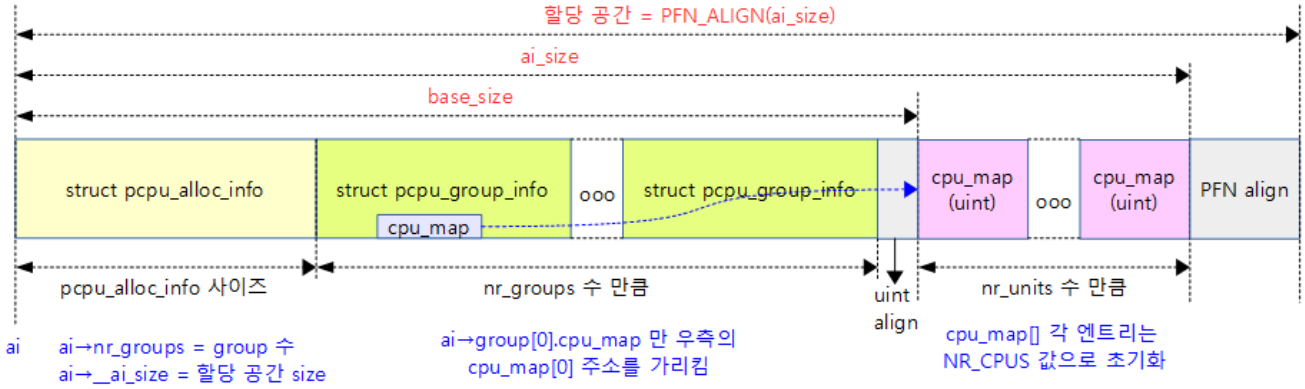
```
01 struct pcpu_alloc_info * __init pcpu_alloc_alloc_info(int nr_groups,
02                                                       int nr_units)
03 {
04         struct pcpu_alloc_info *ai;
05         size_t base_size, ai_size;
06         void *ptr;
07         int unit;
08
09         base_size = ALIGN(sizeof(*ai) + nr_groups * sizeof(ai->groups
   [0]),
10                           __alignof__(ai->groups[0].cpu_map[0]));
11         ai_size = base_size + nr_units * sizeof(ai->groups[0].cpu_map
   [0]);
```

```
12
13          ptr = memblock_alloc_nopanic(PFN_ALIGN(ai_size), 0);
14          if (!ptr)
15                  return NULL;
16          ai = ptr;
17          ptr += base_size;
18
19          ai->groups[0].cpu_map = ptr;
20
21          for (unit = 0; unit < nr_units; unit++)
22                  ai->groups[0].cpu_map[unit] = NR_CPUS;
23
24          ai->nr_groups = nr_groups;
25          ai->__ai_size = PFN_ALIGN(ai_size);
26
27          return ai;
28  }
```

Configuration information such as per-CPU groups and units are created in a pcpu_alloc_info structure and returned.

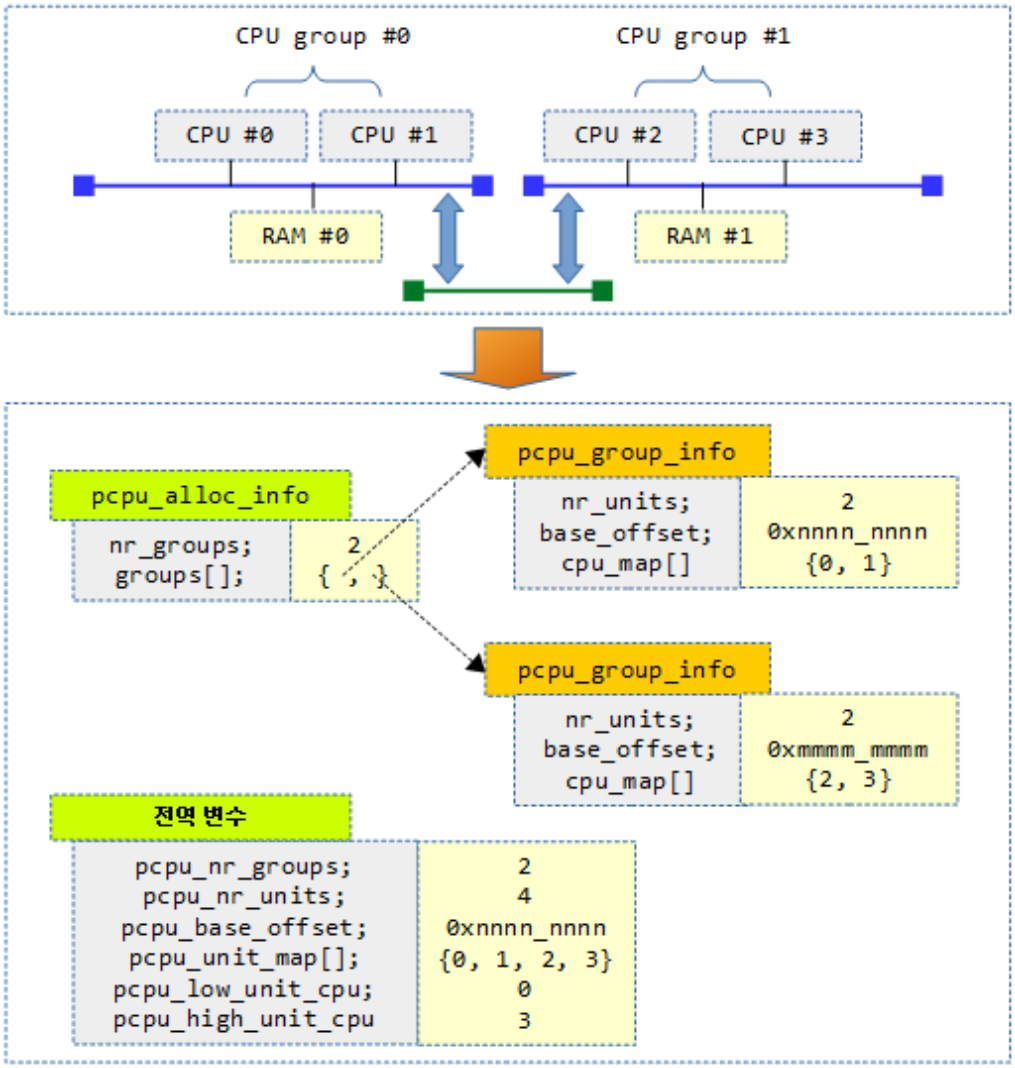
- In line 9~10 of code, sort the size of one struct pcpu_alloc_info base_size value + nr_groups size of the pcpu_group_info struct by the size of the unsigned int, which is of type ai->groups[0].cpu_map[0].
- In line 11 of code, add the size cpu_map[nr_units] to the base_size as the ai_size value.
- In lines 13~15 of the code, sort the ai_size by page size and set the address of the area assigned from the memblock to the ptr.
- In code lines 16~22, connect the cpu_map[ ] space only to the first group. Then, set all the values of the unit->cpu mapping in the first group to the default value (NR_CPUS) that does not map anything.
- In lines 24~25 of the code, substitute the number of groups and the size of the AI assignment.

The following figure shows how a pcpu_alloc_info structure consists of an array of pcpu_group_info structures equal to the number of groups, followed by an array of cpu_map integer arrays equal to the total number of units.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-6.png)

The following figure shows the configuration of the Group information if there are two CPUs on each of the two nodes.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-5c.png)

Memblock allocators used to create the First Chunk for per-cpu by architecture

pcpu_dfl_fc_alloc() – Generic (ARM32, …)

mm/percpu.c

```
1  static void * __init pcpu_dfl_fc_alloc(unsigned int cpu, size_t size,
2                                         size_t align)
3  {
4          return  memblock_alloc_from_nopanic(
5                          size, align, __pa(MAX_DMA_ADDRESS));
6  }
```

Memblock makes a memory allocation request that aligns @size @align. The virtual address of the allocated memory is returned.

pcpu_dfl_fc_free() – Generic (ARM32, …)

mm/percpu.c

```
1  static void __init pcpu_dfl_fc_free(void *ptr, size_t size)
2  {
```

```
3         memblock_free_early(__pa(ptr), size);
4 }
```

Requests @size amount of memory deallocating from the virtual address @ptr allocated via memblock.

pcpu_fc_alloc() – for ARM64

arch/arm64/mm/numa.c

```
1 static void * __init pcpu_fc_alloc(unsigned int cpu, size_t size,
2                                    size_t align)
3 {
4         int nid = early_cpu_to_node(cpu);
5
6         return  memblock_alloc_try_nid(size, align,
7                         __pa(MAX_DMA_ADDRESS), MEMBLOCK_ALLOC_ACCESSIBL
  E, nid);
8 }
```

Memblock makes a memory allocation request that aligns @size @align. The virtual address of the allocated memory is returned.

pcpu_fc_alloc() – for ARM64

arch/arm64/mm/numa.c

```
1 static void __init pcpu_fc_free(void *ptr, size_t size)
2 {
3         memblock_free_early(__pa(ptr), size);
4 }
```

Requests @size amount of memory deallocating from the virtual address @ptr allocated via memblock.

Configuring the First Chunk

First chunk's two management maps

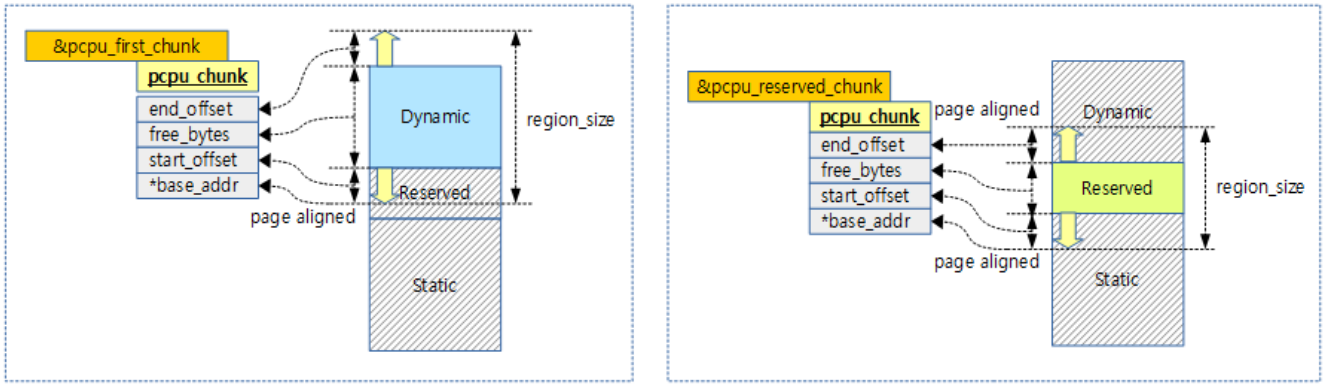
The new general chunks only manage the dynamic area, but in the case of the first chunk, there are three zones (static, reserved, and dynamic), so the management of these areas is different.

- Static Area
  - It is determined at compile time and does not increase or decrease, so there is no need for separate allocation management after the bootup is completed.
- reserved area
  - Allocation management is required when the module is loaded, and the reserved area is used to perform allocation management of the reserved area using the pcpu_chunk structure.
  - If the module is not in use, there is no need to maintain the reserved area.
- Dynamic Zones

- Dynamic regions always require allocation management at runtime, and use pcpu_chunk structs to perform allocation management of dynamic regions.

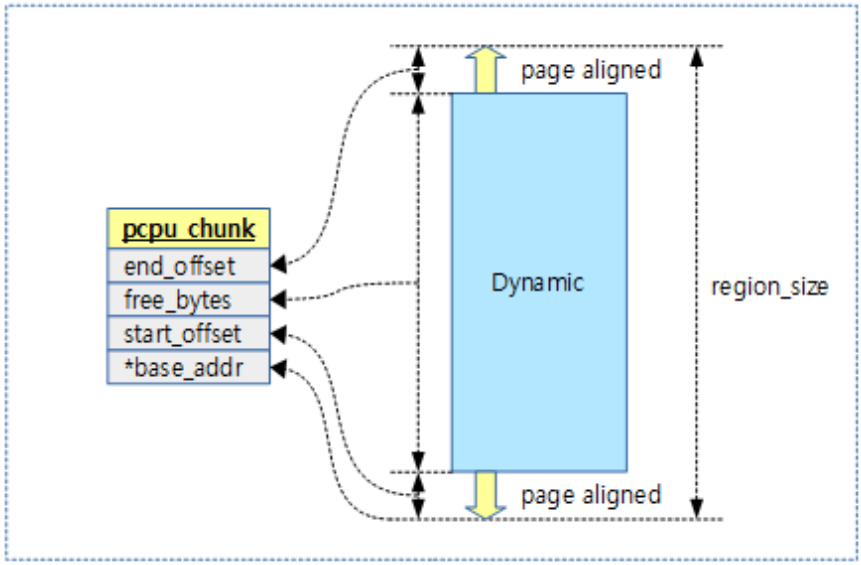
The following figure shows how the two regions of a first chunk are managed by two pcpu_chunk structures.

- The base address of the area to be managed is used by adding the sorted offset to the page.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-11d.png)

You can see that a regular chunk only manages the dynamic area as follows, so it manages it as a single pcpu_chunk structure.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-25b.png)

pcpu_setup_first_chunk()

mm/percpu.c -1/3-

```
01 /**
02  * pcpu_setup_first_chunk - initialize the first percpu chunk
03  * @ai: pcpu_alloc_info describing how to percpu area is shaped
04  * @base_addr: mapped address
05  *
06  * Initialize the first percpu chunk which contains the kernel static
07  * perpcu area.  This function is to be called from arch percpu area
08  * setup path.
```

```
 *
 * @ai contains all information necessary to initialize the first
 * chunk and prime the dynamic percpu allocator.
 *
 * @ai->static_size is the size of static percpu area.
 *
 * @ai->reserved_size, if non-zero, specifies the amount of bytes to
 * reserve after the static area in the first chunk.  This reserves
 * the first chunk such that it's available only through reserved
 * percpu allocation.  This is primarily used to serve module percpu
 * static areas on architectures where the addressing model has
 * limited offset range for symbol relocations to guarantee module
 * percpu symbols fall inside the relocatable range.
 *
 * @ai->dyn_size determines the number of bytes available for dynamic
 * allocation in the first chunk.  The area between @ai->static_size +
 * @ai->reserved_size + @ai->dyn_size and @ai->unit_size is unused.
 *
 * @ai->unit_size specifies unit size and must be aligned to PAGE_SIZE
 * and equal to or larger than @ai->static_size + @ai->reserved_size +
 * @ai->dyn_size.
 *
 * @ai->atom_size is the allocation atom size and used as alignment
 * for vm areas.
 *
 * @ai->alloc_size is the allocation size and always multiple of
 * @ai->atom_size.  This is larger than @ai->atom_size if
 * @ai->unit_size is larger than @ai->atom_size.
 *
 * @ai->nr_groups and @ai->groups describe virtual memory layout of
 * percpu areas.  Units which should be colocated are put into the
 * same group.  Dynamic VM areas will be allocated according to these
 * groupings.  If @ai->nr_groups is zero, a single group containing
 * all units is assumed.
 *
 * The caller should have mapped the first chunk at @base_addr and
 * copied static data to each unit.
 *
 * The first chunk will always contain a static and a dynamic region.
 * However, the static region is not managed by any chunk.  If the first
 * chunk also contains a reserved region, it is served by two chunks -
 * one for the reserved region and one for the dynamic region.  They
 * share the same vm, but use offset regions in the area allocation map.
 * The chunk serving the dynamic region is circulated in the chunk slots
 * and available for dynamic allocation like any other chunk.
 *
 * RETURNS:
 * 0 on success, -errno on failure.
 */
int __init pcpu_setup_first_chunk(const struct pcpu_alloc_info *ai,
                                  void *base_addr)
{
        size_t size_sum = ai->static_size + ai->reserved_size + ai->dyn_
size;
        size_t static_size, dyn_size;
        struct pcpu_chunk *chunk;
        unsigned long *group_offsets;
        size_t *group_sizes;
        unsigned long *unit_off;
        unsigned int cpu;
        int *unit_map;
        int group, unit, i;
        int map_size;
        unsigned long tmp_addr;

#define PCPU_SETUP_BUG_ON(cond) do {
\
```

```
17          if (unlikely(cond)) {                                   \
18                  pr_emerg("failed to initialize, %s\n", #cond);  \
19                  pr_emerg("cpu_possible_mask=%*pb\n",            \
20                           cpumask_pr_args(cpu_possible_mask));   \
21                  pcpu_dump_alloc_info(KERN_EMERG, ai);           \
22                  BUG();                                          \
23          }                                                       \
24  } while (0)
25
26          /* sanity checks */
27          PCPU_SETUP_BUG_ON(ai->nr_groups <= 0);
28  #ifdef CONFIG_SMP
29          PCPU_SETUP_BUG_ON(!ai->static_size);
30          PCPU_SETUP_BUG_ON(offset_in_page(__per_cpu_start));
31  #endif
32          PCPU_SETUP_BUG_ON(!base_addr);
33          PCPU_SETUP_BUG_ON(offset_in_page(base_addr));
34          PCPU_SETUP_BUG_ON(ai->unit_size < size_sum);
35          PCPU_SETUP_BUG_ON(offset_in_page(ai->unit_size));
36          PCPU_SETUP_BUG_ON(ai->unit_size < PCPU_MIN_UNIT_SIZE);
37          PCPU_SETUP_BUG_ON(!IS_ALIGNED(ai->unit_size, PCPU_BITMAP_BLOCK_S
    IZE));
38          PCPU_SETUP_BUG_ON(ai->dyn_size < PERCPU_DYNAMIC_EARLY_SIZE);
39          PCPU_SETUP_BUG_ON(!ai->dyn_size);
40          PCPU_SETUP_BUG_ON(!IS_ALIGNED(ai->reserved_size, PCPU_MIN_ALLOC_
    SIZE));
41          PCPU_SETUP_BUG_ON(!(IS_ALIGNED(PCPU_BITMAP_BLOCK_SIZE, PAGE_SIZ
    E) ||
42                              IS_ALIGNED(PAGE_SIZE, PCPU_BITMAP_BLOCK_SIZ
    E)));
43          PCPU_SETUP_BUG_ON(pcpu_verify_alloc_info(ai) < 0);
44
45          /* process group information and build config tables accordingly
    */
46          group_offsets = memblock_alloc(ai->nr_groups * sizeof(group_offs
    ets[0]),
47                                         SMP_CACHE_BYTES);
48          group_sizes = memblock_alloc(ai->nr_groups * sizeof(group_sizes
    [0]),
49                                       SMP_CACHE_BYTES);
50          unit_map = memblock_alloc(nr_cpu_ids * sizeof(unit_map[0]),
51                                    SMP_CACHE_BYTES);
52          unit_off = memblock_alloc(nr_cpu_ids * sizeof(unit_off[0]),
53                                    SMP_CACHE_BYTES);
54
55          for (cpu = 0; cpu < nr_cpu_ids; cpu++)
56                  unit_map[cpu] = UINT_MAX;
```

Initializes the first chunk of per-cpu data.

- @ai: This is where you have all the information you need to initialize.
- @base_addr: Addresses to be mapped
- Before entering this function, the memory of the per-cpu area is already allocated and the struct is prepared to manage the mapping.

- In lines 46~53 of the code, the group information and unit->cpu mapping table required for mapping are allocated and initialized as follows.
    - group_offsets[]

        - Allocate an array with group_offsets variables equal to the number of groups (nr_groups).
    - group_sizes[]
        - Allocate an array with group_sizes variables equal to the number of groups (nr_groups).
    - unit_map[]
        - Allocate an array with unit_map to be used for unit->cpu mapping equal to the number of CPUs (nr_cpu_ids).
    - unit_off[]
        - Allocate an array with unit_off equal to the number of CPUs (nr_cpu_ids)
- In code lines 55~56, initialize the contents of the unit->cpu mapping array to the unmapped (UINT_MAX) value.

mm/percpu.c -2/3-

```
01          pcpu_low_unit_cpu = NR_CPUS;
02          pcpu_high_unit_cpu = NR_CPUS;
03
04          for (group = 0, unit = 0; group < ai->nr_groups; group++, unit +
    = i) {
05                  const struct pcpu_group_info *gi = &ai->groups[group];
06
07                  group_offsets[group] = gi->base_offset;
08                  group_sizes[group] = gi->nr_units * ai->unit_size;
09
10                  for (i = 0; i < gi->nr_units; i++) {
11                          cpu = gi->cpu_map[i];
12                          if (cpu == NR_CPUS)
13                                  continue;
14
15                          PCPU_SETUP_BUG_ON(cpu >= nr_cpu_ids);
16                          PCPU_SETUP_BUG_ON(!cpu_possible(cpu));
17                          PCPU_SETUP_BUG_ON(unit_map[cpu] != UINT_MAX);
18
19                          unit_map[cpu] = unit + i;
20                          unit_off[cpu] = gi->base_offset + i * ai->unit_s
    ize;
21
22                          /* determine low/high unit_cpu */
23                          if (pcpu_low_unit_cpu == NR_CPUS ||
24                              unit_off[cpu] < unit_off[pcpu_low_unit_cpu])
25                                  pcpu_low_unit_cpu = cpu;
26                          if (pcpu_high_unit_cpu == NR_CPUS ||
27                              unit_off[cpu] > unit_off[pcpu_high_unit_cp
    u])
28                                  pcpu_high_unit_cpu = cpu;
29                  }
30          }
31          pcpu_nr_units = unit;
32
33          for_each_possible_cpu(cpu)
34                  PCPU_SETUP_BUG_ON(unit_map[cpu] == UINT_MAX);
35
36          /* we're done parsing the input, undefine BUG macro and dump con
    fig */
37  #undef PCPU_SETUP_BUG_ON
38          pcpu_dump_alloc_info(KERN_DEBUG, ai);
39
40          pcpu_nr_groups = ai->nr_groups;
```

```
41          pcpu_group_offsets = group_offsets;
42          pcpu_group_sizes = group_sizes;
43          pcpu_unit_map = unit_map;
44          pcpu_unit_offsets = unit_off;
45
46          /* determine basic parameters */
47          pcpu_unit_pages = ai->unit_size >> PAGE_SHIFT;
48          pcpu_unit_size = pcpu_unit_pages << PAGE_SHIFT;
49          pcpu_atom_size = ai->atom_size;
50          pcpu_chunk_struct_size = sizeof(struct pcpu_chunk) +
51                  BITS_TO_LONGS(pcpu_unit_pages) * sizeof(unsigned long);
52
53          pcpu_stats_save_ai(ai);
```

- In lines 1-2 of code, set the NR_CPUS to the pcpu_low_unit_cpu and pcpu_high_unit_cpu variables as initial values.
- In line 4~8 of the code, loop around the number of groups and set the information about the groups.
  - The group_offsets stores base_offset values from existing group information.
  - The group_sizes stores the number of nr_units in the existing group information, multiplied by the unit size.
- In line 10~13 of code, loop around the number of units in the group and skip the units that are not mapped to unit->cpu.
- Perform the CPU->unit mapping on lines 19~28 of the code. Here's what you'll see:
  - Configure a table in unit_map that represents the unit number with the CPU index
  - Set the offset value required for each CPU access to the unit_off (base_offset of the group + unit size)
  - pcpu_low_unit_cpu
    - Stores the lowest base address of all CPUs
  - pcpu_high_unit_cpu
    - Stores the highest base address of all CPUs
- On line 31 of the code, set the global pcpu_nr_units plus the number of units in each group.
  - In the case of asymmetric NUMA systems, unmapped units are also included.
  - For example, if the number of CPUs for the two groups is cpus = {2, 8}, and the best_upa = 1, the nr_units for each group is 4 and 8.
- Line 38 of the code prints debugging information related to per-CPU allocation to the console.
- In line 40~51 of the code, initialize the PCPU-related global variables.

mm/percpu.c -3/3-

```
01          /*
02           * Allocate chunk slots.  The additional last slot is for
03           * empty chunks.
04           */
05          pcpu_nr_slots = __pcpu_size_to_slot(pcpu_unit_size) + 2;
06          pcpu_slot = memblock_alloc(pcpu_nr_slots * sizeof(pcpu_slot[0]),
07                                     SMP_CACHE_BYTES);
08          for (i = 0; i < pcpu_nr_slots; i++)
09                  INIT_LIST_HEAD(&pcpu_slot[i]);
10
11           /*
12           * The end of the static region needs to be aligned with the
13           * minimum allocation size as this offsets the reserved and
```

```
14          * dynamic region.  The first chunk ends page aligned by
15          * expanding the dynamic region, therefore the dynamic region
16          * can be shrunk to compensate while still staying above the
17          * configured sizes.
18          */
19         static_size = ALIGN(ai->static_size, PCPU_MIN_ALLOC_SIZE);
20         dyn_size = ai->dyn_size - (static_size - ai->static_size);
21
22         /*
23          * Initialize first chunk.
24          * If the reserved_size is non-zero, this initializes the reserv
   ed
25          * chunk.  If the reserved_size is zero, the reserved chunk is N
   ULL
26          * and the dynamic region is initialized here.  The first chunk,
27          * pcpu_first_chunk, will always point to the chunk that serves
28          * the dynamic region.
29          */
30         tmp_addr = (unsigned long)base_addr + static_size;
31         map_size = ai->reserved_size ?: dyn_size;
32         chunk = pcpu_alloc_first_chunk(tmp_addr, map_size);
33
34         /* init dynamic chunk if necessary */
35         if (ai->reserved_size) {
36                 pcpu_reserved_chunk = chunk;
37
38                 tmp_addr = (unsigned long)base_addr + static_size +
39                            ai->reserved_size;
40                 map_size = dyn_size;
41                 chunk = pcpu_alloc_first_chunk(tmp_addr, map_size);
42         }
43
44         /* link the first chunk in */
45         pcpu_first_chunk = chunk;
46         pcpu_nr_empty_pop_pages = pcpu_first_chunk->nr_empty_pop_pages;
47         pcpu_chunk_relocate(pcpu_first_chunk, -1);
48
49         /* include all regions of the first chunk */
50         pcpu_nr_populated += PFN_DOWN(size_sum);
51
52         pcpu_stats_chunk_alloc();
53         trace_percpu_create_chunk(base_addr);
54
55         /* we're done */
56         pcpu_base_addr = base_addr;
57         return 0;
58 }
```

- In line 5~7 of the code, calculate the maximum number of slots + 2 to manage the list of chunks in unit size, and allocate pcpu_slot[ ] from memblock for that number.
  - __pcpu_size_to_slot()
    - Recognize the slot by unit size.
    - PCPU_SLOT_BASE_SHIFT=5
    - e.g. slot value according to size
      - 32K -> 13번 slot
      - 64K -> 12번 slot
      - 1M -> 8번 slot
      - 2M -> 7번 slot
  - The first 0 slot is for chunks that don't have space to allocate.
  - The last slot is the one that always has 1 empty chunk waiting for it.

    - There are times when you have to wait for more than two, but you keep one because it is organized by the schedule work.
- Initialize the list structure of pcpu_slot[ ] by the number of slots allocated in line 8~9.
- In lines 19~20 of code, the size of the static area should be aligned by at least 4 bytes. The size of the dynamic area is also recalculated accordingly.
- In line 30~32 of code, allocate and initialize the first chunk management pcpu_chunk.
- In line 35~42 of the code, if the module is supported and a reserved area is required, the previously allocated chunk is designated for reserved, and a new first chunk management pcpu_chunk is allocated and initialized.
- Specify the first chunk assigned in lines 45~47 for dynamic, and then add it to the slot that manages the chunk list.
- In line 50 of the code, all the pages in the first chunk are multiplied, so pcpu_nr_populated the counter increments the counter by that amount.
- In line 52 of code, increment the stat counters for adding chunks.

Chunk Management

Initialize the allocation and mapping of two chunks for the First Chunk

pcpu_alloc_first_chunk()

mm/percpu.c

```
01 /**
02  * pcpu_alloc_first_chunk - creates chunks that serve the first chunk
03  * @tmp_addr: the start of the region served
04  * @map_size: size of the region served
05  *
06  * This is responsible for creating the chunks that serve the first chun
    k.  The
07  * base_addr is page aligned down of @tmp_addr while the region end is p
    age
08  * aligned up.  Offsets are kept track of to determine the region serve
    d. All
09  * this is done to appease the bitmap allocator in avoiding partial bloc
    ks.
10  *
11  * RETURNS:
12  * Chunk serving the region at @tmp_addr of @map_size.
13  */
```

```
01 static struct pcpu_chunk * __init pcpu_alloc_first_chunk(unsigned long t
    mp_addr,
02                                                         int map_size)
03 {
04         struct pcpu_chunk *chunk;
05         unsigned long aligned_addr, lcm_align;
06         int start_offset, offset_bits, region_size, region_bits;
07
08         /* region calculations */
09         aligned_addr = tmp_addr & PAGE_MASK;
10
```

```
11          start_offset = tmp_addr - aligned_addr;
12
13          /*
14           * Align the end of the region with the LCM of PAGE_SIZE and
15           * PCPU_BITMAP_BLOCK_SIZE.  One of these constants is a multiple
    of
16           * the other.
17           */
18          lcm_align = lcm(PAGE_SIZE, PCPU_BITMAP_BLOCK_SIZE);
19          region_size = ALIGN(start_offset + map_size, lcm_align);
20
21          /* allocate chunk */
22          chunk = memblock_alloc(sizeof(struct pcpu_chunk) +
23                                 BITS_TO_LONGS(region_size >> PAGE_SHIFT),
24                                 SMP_CACHE_BYTES);
25
26          INIT_LIST_HEAD(&chunk->list);
27
28          chunk->base_addr = (void *)aligned_addr;
29          chunk->start_offset = start_offset;
30          chunk->end_offset = region_size - chunk->start_offset - map_siz
    e;
31
32          chunk->nr_pages = region_size >> PAGE_SHIFT;
33          region_bits = pcpu_chunk_map_bits(chunk);
34
35          chunk->alloc_map = memblock_alloc(BITS_TO_LONGS(region_bits) * s
    izeof(chunk->alloc_map[0]),
36                                            SMP_CACHE_BYTES);
37          chunk->bound_map = memblock_alloc(BITS_TO_LONGS(region_bits + 1)
    * sizeof(chunk->bound_map[00
38  ]),
39                                            SMP_CACHE_BYTES);
40          chunk->md_blocks = memblock_alloc(pcpu_chunk_nr_blocks(chunk) *
    sizeof(chunk->md_blocks[0]),
41                                            SMP_CACHE_BYTES);
42          pcpu_init_md_blocks(chunk);
43
44          /* manage populated page bitmap */
45          chunk->immutable = true;
46          bitmap_fill(chunk->populated, chunk->nr_pages);
47          chunk->nr_populated = chunk->nr_pages;
48          chunk->nr_empty_pop_pages =
49                  pcpu_cnt_pop_pages(chunk, start_offset / PCPU_MIN_ALLOC_
    SIZE,
50                                     map_size / PCPU_MIN_ALLOC_SIZE);
51
52          chunk->contig_bits = map_size / PCPU_MIN_ALLOC_SIZE;
53          chunk->free_bytes = map_size;
54
55          if (chunk->start_offset) {
56                  /* hide the beginning of the bitmap */
57                  offset_bits = chunk->start_offset / PCPU_MIN_ALLOC_SIZE;
58                  bitmap_set(chunk->alloc_map, 0, offset_bits);
59                  set_bit(0, chunk->bound_map);
60                  set_bit(offset_bits, chunk->bound_map);
61
62                  chunk->first_bit = offset_bits;
63
64                  pcpu_block_update_hint_alloc(chunk, 0, offset_bits);
65          }
66
67          if (chunk->end_offset) {
68                  /* hide the end of the bitmap */
69                  offset_bits = chunk->end_offset / PCPU_MIN_ALLOC_SIZE;
70                  bitmap_set(chunk->alloc_map,
71                             pcpu_chunk_map_bits(chunk) - offset_bits,
72                             offset_bits);
```

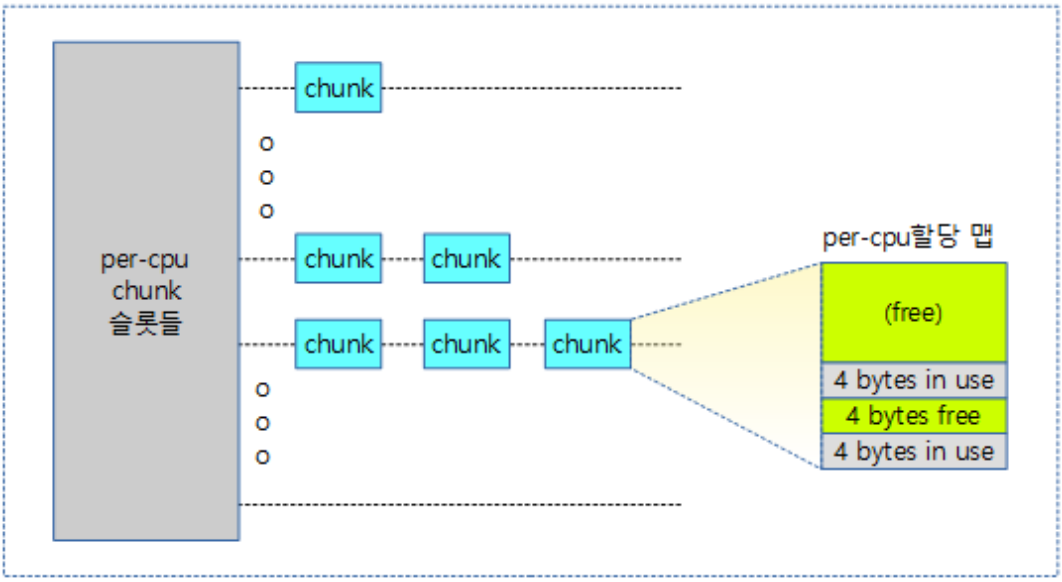
```
73                  set_bit((start_offset + map_size) / PCPU_MIN_ALLOC_SIZE,
74                          chunk->bound_map);
75                  set_bit(region_bits, chunk->bound_map);
76
77                  pcpu_block_update_hint_alloc(chunk, pcpu_chunk_map_bits
    (chunk)
78                                               - offset_bits, offset_bit
    s);
79          }
80
81          return chunk;
82  }
```

Initialize the map of the first chunk.

- In line 9~11 of the code, the base address of the area to be managed is used by page alignment, so it is used by adding the sorted offset.
- In lines 18~19 of the code, sort the map_size plus offset by the least common multiple of the two values, PCPU_BITMAP_BLOCK_SIZE the page size.
  - In the current kernel, the PCPU_BITMAP_BLOCK_SIZE value is equal to the page size. Therefore, the minimum common multiple value is always the page size.
- In lines 22~24 of code, pcpu_chunk allocates memory for information. This also includes the memory to be used for populated bitmaps.
- In lines 28~30 of the code, specify the aligned start address, start offset, and end offset.
- Specify the number of pages in the recalculated area in lines 32~33 of the code, and calculate the number of bits required for the chunk.
  - Calculate in words.
  - e.g. nr_pages=10, 4K pages
    - region_bits=10K
- In line 35~39 of code, the number of bits to be used for bound_map is allocated to the alloc_map, which is managed in words.
- In line 40~42 of code, it allocates and initializes managed metadata per 1024-byte block unit.
- In line 45~50 of the code, the first chunk is already enabled, so we handle it.
- In lines 52~53 of the code, specify the actual @map_size as free bytes, and specify the number of free words in the contig_bits.
- The start_offset section from code lines 55~65 does not need to be managed, so it is set and hidden on the map.
- The end_offset section from code lines 67~79 does not need to be managed, so it is set and hidden on the map.

Per-cpu memory allocation map in chunks

Use the slot list to manage chunks, as shown in the following figure.



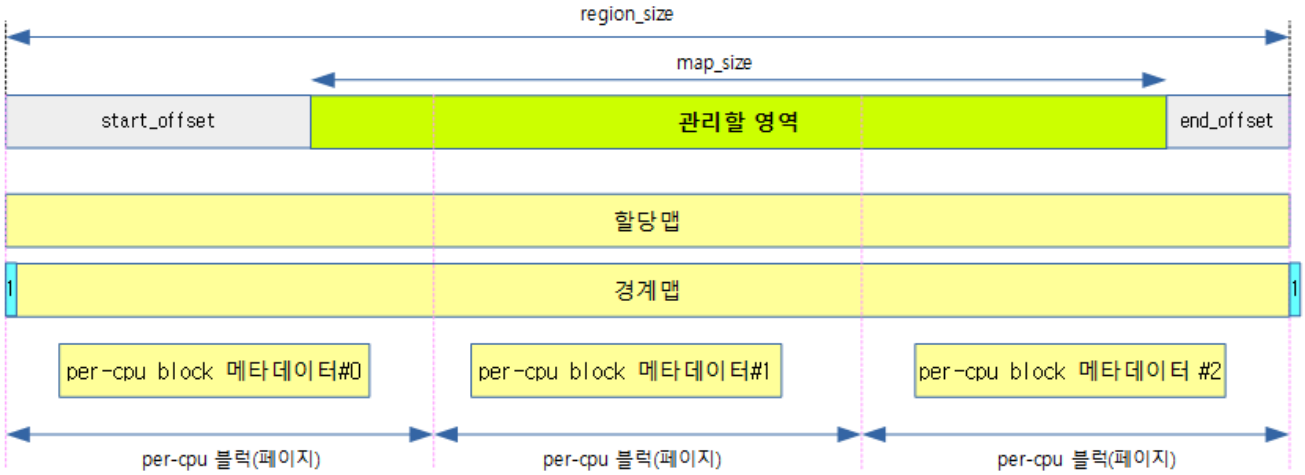
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-29.png)

We'll explain how to manage these chunks later, but let's first understand the behavior of the per-cpu allocation map inside a chunk.

1) bitmap method allocator

- In the way applied in kernel v4.14-rc1, the scalable problem occurs when there is a large allocation of per-CPU data. To solve this, it was changed to a chunk area map allocator that is managed in a bitmap manner.
- consultation
  - percpu: replace area map allocator with bitmap (https://github.com/torvalds/linux/commit/40064aeca35c5c14789e2adcf3a1d7e5d4bd65f2#diff-5050eed868076fe2656aea8c2eb7312a)
  - percpu: introduce bitmap metadata blocks (https://github.com/torvalds/linux/commit/ca460b3c96274d79f84b31a3fea23a6eed479917#diff-5050eed868076fe2656aea8c2eb7312a)

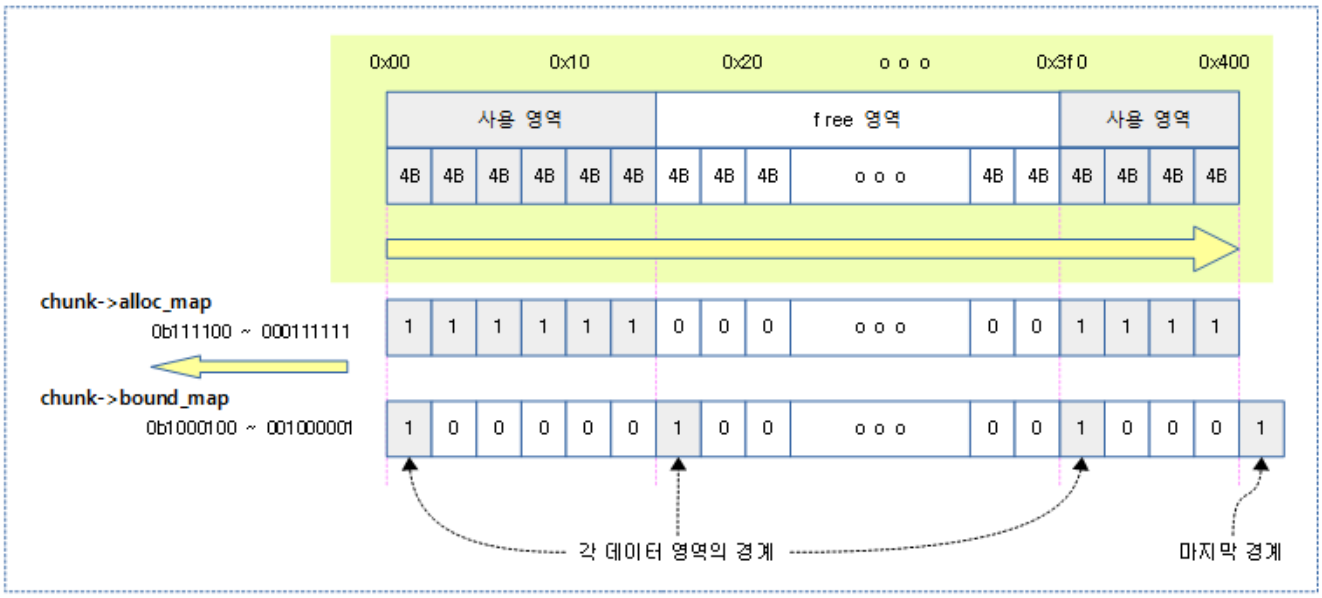
The following figure shows the per-CPU region managed by the chunk, with two bitmaps and metadata managed in per-CPU blocks.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-31c.png)

The following figure shows how to get a detailed representation of the assignment map (alloc_map) and the boundary map (bound_map) for the managed map area.

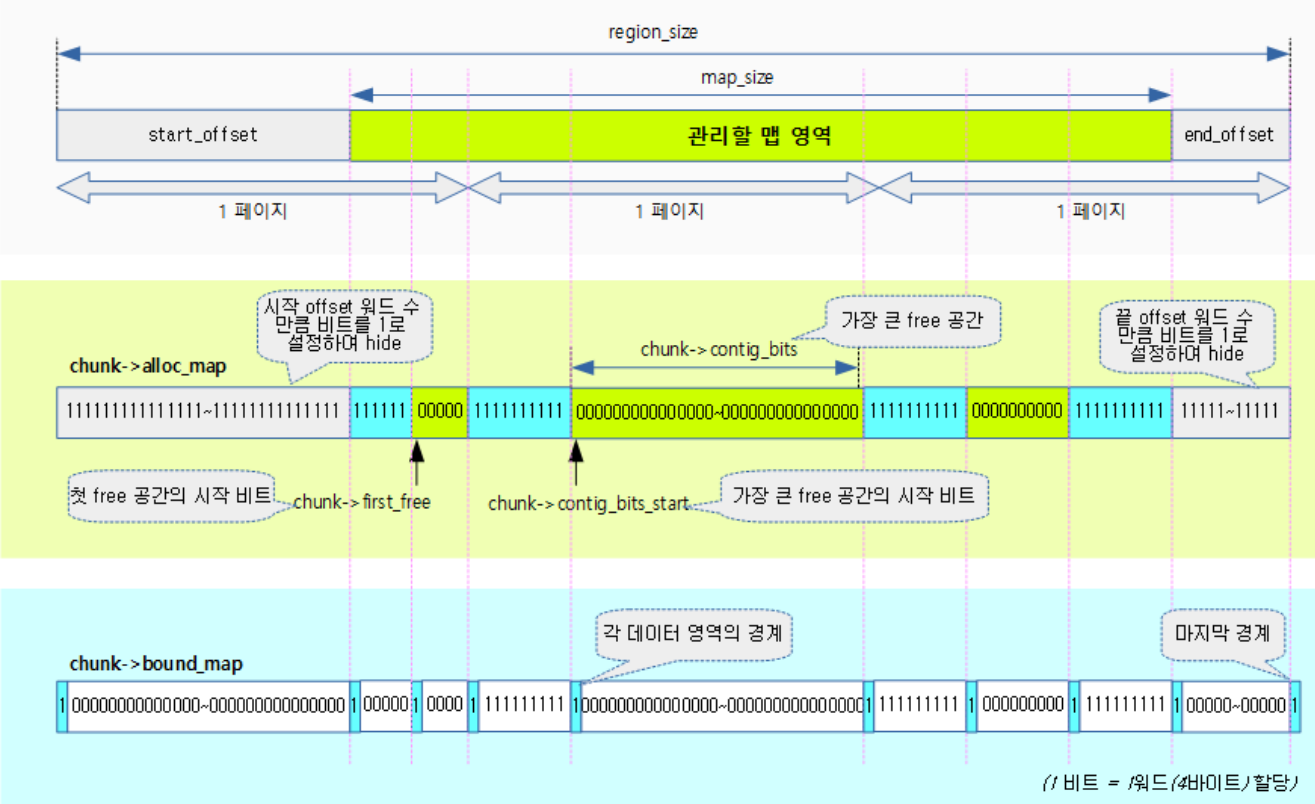
- Allocation maps and boundary maps are made up of bitmaps, and the bitmap size manages the region_size in words.
  - Based on a 4K page, the number of bits used to manage one page, i.e., the number of bits used to manage one block, is 1x1 bits.
    - e.g. 0x5520 byte area - > bitmap required to manage 0x6000 bytes by arranging the page is 0x1800 bits.
  - Allocation Map (alloc_map)
    - One bit represents the assignment of one word consisting of four bytes. (1=free, 4=used)
  - Boundary Map (bound_map)
    - One bit corresponds to one word of four bytes, but only sets 1 to the starting bit.
    - The boundary map uses 1 BT more than the allocation map. The last 1 bit is additionally configured to represent the last of the data.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-28.png)

The following illustration shows how bitmaps are represented in more detail for a managed map area.

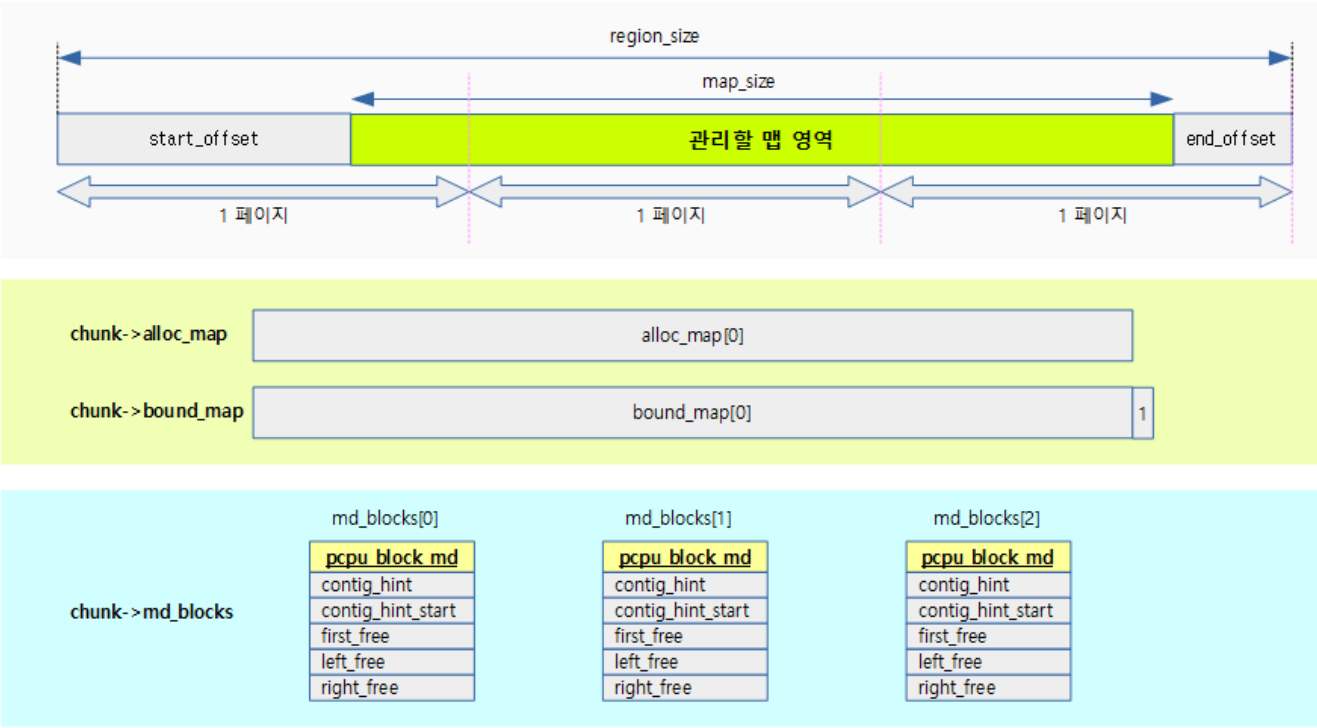
- Since the map area is arranged on a page-by-page basis, it is set to 1 so that all parts of the start_offset and end_offset are allotted for concealment and processing.
- The number of bits in the largest free region within a chunk is stored in chunk->conting_bits.
- The starting bit of the largest free area within a chunk is stored in chunk->contig_bits_start.
- The starting bit, which points to the first free area within the chunk, is stored in chunk->first_free.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-24.png)

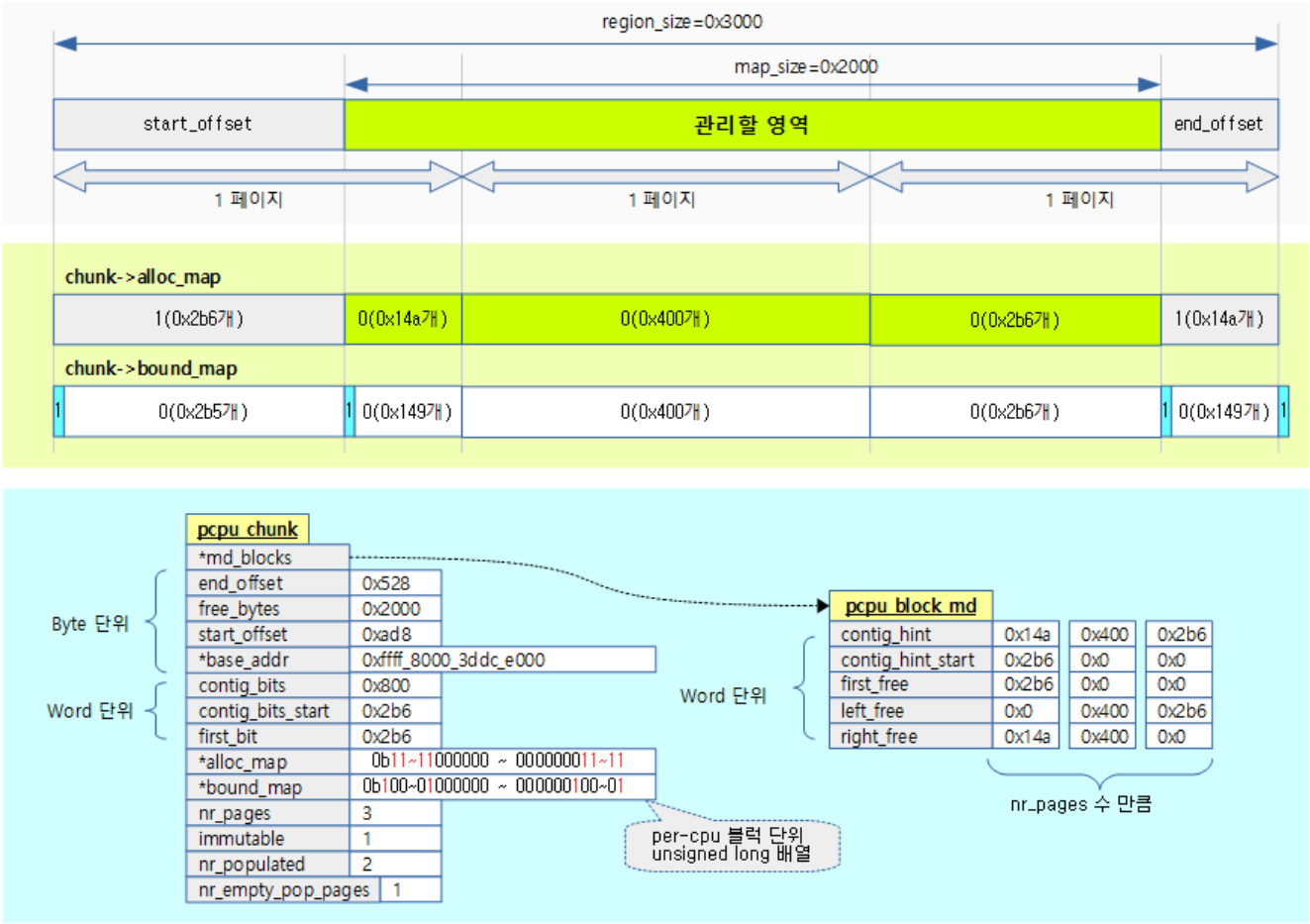
The size of the map area to be managed within a chunk is managed in per-cpu blocks (default pages), and the following figure shows an example of using three per-cpu blocks.

- The per-CPU block metadata is managed in conjunction with an array of pcpu_block_md structures.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-27a.png)

The following figure shows the initialization of the chunk for page 2 (8K) of the reserve area of the first chunk, and the values that are initialized to manage the map for each block.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-25.png)

The following illustration shows several examples of left_free and right_free on the map for each block.

- If the free area is attached to the left end or the right end, the number of bits in that area is used for left_free and right_free, respectively.

(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-26a.png)

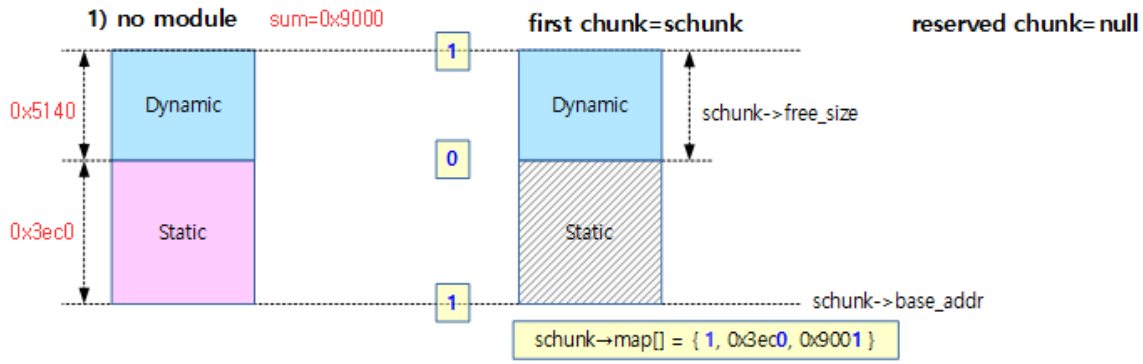
2) Area map method allocator

This is how it was managed until kernel v4.13.

The figure below shows an example of map[] that is assigned when the area map is initialized using the area map method, and the reserved(module) area is not requested.

- rpi2: dyn_size=0x5140, static_size=0x3ec0
  - 1 bit of LSB in the map value is an indication of in-use (1=in use) and 0 bits is an indication of free (0=available).
  - As shown below, it is in-use until 0x0~0x3ec0, and it is mapped to free status until 0x3ec0~0x9000.
- A schunk is chosen for the first chunk, and the reserved chunk has no area, so null is assigned to it.
- schunk->free_size is the same as dyn_size.

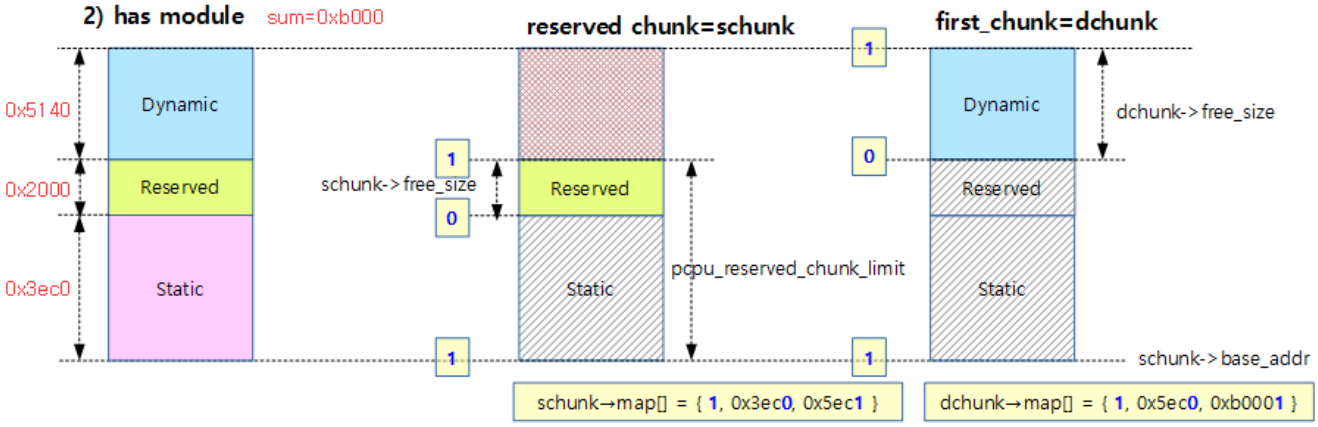
- First Chunk is used to manage the mapping of dynamic assignments.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-12.png)

The figure below is an example of a map[] that is divided into two parts, schunk and dchunk, when a reserved(module) area is requested when initializing using the Area map method.

- rpi2: dyn_size=0x5140, static_size=0x3ec0, reserved=0x2000
  - The mapping in schunk is in-use until 0x0~0x3ec0, and it is mapped to free until 0x3ec0~0x5ec0.
  - The mapping in dchunk is in-use until 0x0~0x5ec0, and it is mapped to free until 0x5ec0~0xb000.
- The dchunk was chosen to manage the mapping of the dynamic area to the first chunk, and the reserved chunk is managed by substituting the schunk.
  - schunk->free_size is the same as reserved_size.
  - dchunk->free_size is like dyn_size.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-13a.png)

Updating the Per-CPU Block Hint for an Allocation

pcpu_block_update_hint_alloc()

mm/percpu.c -1/2-

```
01 /**
02  * pcpu_block_update_hint_alloc - update hint on allocation path
03  * @chunk: chunk of interest
```

```
 * @bit_off: chunk offset
 * @bits: size of request
 *
 * Updates metadata for the allocation path.  The metadata only has to be
 * refreshed by a full scan iff the chunk's contig hint is broken.  Block level
 * scans are required if the block's contig hint is broken.
 */
```

```
static void pcpu_block_update_hint_alloc(struct pcpu_chunk *chunk, int bit_off,
                                         int bits)
{
        struct pcpu_block_md *s_block, *e_block, *block;
        int s_index, e_index;   /* block indexes of the freed allocation */
        int s_off, e_off;       /* block offsets of the freed allocation */

        /*
         * Calculate per block offsets.
         * The calculation uses an inclusive range, but the resulting offsets
         * are [start, end).  e_index always points to the last block in the
         * range.
         */
        s_index = pcpu_off_to_block_index(bit_off);
        e_index = pcpu_off_to_block_index(bit_off + bits - 1);
        s_off = pcpu_off_to_block_off(bit_off);
        e_off = pcpu_off_to_block_off(bit_off + bits - 1) + 1;

        s_block = chunk->md_blocks + s_index;
        e_block = chunk->md_blocks + e_index;

        /*
         * Update s_block.
         * block->first_free must be updated if the allocation takes its place.
         * If the allocation breaks the contig_hint, a scan is required to
         * restore this hint.
         */
        if (s_off == s_block->first_free)
                s_block->first_free = find_next_zero_bit(
                                        pcpu_index_alloc_map(chunk, s_index),
                                        PCPU_BITMAP_BLOCK_BITS,
                                        s_off + bits);

        if (s_off >= s_block->contig_hint_start &&
            s_off < s_block->contig_hint_start + s_block->contig_hint) {
                /* block contig hint is broken - scan to fix it */
                pcpu_block_refresh_hint(chunk, s_index);
        } else {
                /* update left and right contig manually */
                s_block->left_free = min(s_block->left_free, s_off);
                if (s_index == e_index)
                        s_block->right_free = min_t(int, s_block->right_free,
                                                    PCPU_BITMAP_BLOCK_BITS - e_off);
                else
                        s_block->right_free = 0;
        }
```

After per-CCPU allocation, the largest contiguous free area and its associated members are updated. Note that the function that is called after deallocating is pcpu_block_update_hint_free().

- In line 14~17 of the code, we get the value of each block index and block offset corresponding to the start offset and the end offset.
- In line 19~20 of the code, calculate the start and end blocks with the index values of the start and end blocks.
- In line 28~32 of code, if you are allocating in the free space located at the beginning, exclude the space to be allocated and then find the free space location to update the first_free.
- If the allocation is within the largest free space range from code lines 34~37, the contig_hint is updated.
- This is the case when the code line 38~46 allocates from a place that is not within the largest free space range. If you allocate in the leftmost free space, the left_free is updated. And if the area to be allocated is within one per-CPU block and is allocated in the free space on the right, the right_free is updated for each. If the area is managed by dividing it into two or more per-CPU blocks, the right_free will be zero.

mm/percpu.c -2/2-

```
01          /*
02           * Update e_block.
03           */
04          if (s_index != e_index) {
05                  /*
06                   * When the allocation is across blocks, the end is alon
    g
07                   * the left part of the e_block.
08                   */
09                  e_block->first_free = find_next_zero_bit(
10                                  pcpu_index_alloc_map(chunk, e_index),
11                                  PCPU_BITMAP_BLOCK_BITS, e_off);
12
13                  if (e_off == PCPU_BITMAP_BLOCK_BITS) {
14                          /* reset the block */
15                          e_block++;
16                  } else {
17                          if (e_off > e_block->contig_hint_start) {
18                                  /* contig hint is broken - scan to fix i
    t */
19                                  pcpu_block_refresh_hint(chunk, e_index);
20                          } else {
21                                  e_block->left_free = 0;
22                                  e_block->right_free =
23                                          min_t(int, e_block->right_free,
24                                                PCPU_BITMAP_BLOCK_BITS - e
    _off);
25                          }
26                  }
27
28                  /* update in-between md_blocks */
29                  for (block = s_block + 1; block < e_block; block++) {
30                          block->contig_hint = 0;
31                          block->left_free = 0;
32                          block->right_free = 0;
33                  }
34          }
35
36          /*
```

```
37                 * The only time a full chunk scan is required is if the chunk
38                 * contig hint is broken.  Otherwise, it means a smaller space
39                 * was used and therefore the chunk contig hint is still correc
   t.
40                 */
41                if (bit_off >= chunk->contig_bits_start  &&
42                    bit_off < chunk->contig_bits_start + chunk->contig_bits)
43                        pcpu_chunk_refresh_hint(chunk);
44        }
```

- In code lines 4~11, if the allocation area is divided into two or more per-CPU blocks, the members corresponding to the last block will be updated. First, find and update the free area that starts first after the allocated area in the last block.
- In code lines 13~15, if you allocate to the end of the last block, point to the next block.
- In code lines 16~19, if the assignment location is within the largest free space, the hints related to the block are updated.
- If it is in a different position in code lines 20~25, the left_free should be 0 and the right_free updated.
- In code lines 29~33, change all the values of the blocks between the start and end blocks to in use.
- In lines 41~43 of the code, update the hints associated with the chunk if the allocation request is within the largest free space within the chunk.

pcpu_block_refresh_hint()

mm/percpu.c

```
1  /**
2   * pcpu_block_refresh_hint
3   * @chunk: chunk of interest
4   * @index: index of the metadata block
5   *
6   * Scans over the block beginning at first_free and updates the block
7   * metadata accordingly.
8   */
```

```
01 static void pcpu_block_refresh_hint(struct pcpu_chunk *chunk, int index)
02 {
03         struct pcpu_block_md *block = chunk->md_blocks + index;
04         unsigned long *alloc_map = pcpu_index_alloc_map(chunk, index);
05         int rs, re;     /* region start, region end */
06
07         /* clear hints */
08         block->contig_hint = 0;
09         block->left_free = block->right_free = 0;
10
11         /* iterate over free areas and update the contig hints */
12         pcpu_for_each_unpop_region(alloc_map, rs, re, block->first_free,
13                                    PCPU_BITMAP_BLOCK_BITS) {
14                 pcpu_block_update(block, rs, re);
15         }
16 }
```

Update the hint information of the @index metadata block requested in @chunk.

- Determine where the alloc_map starts with the metadata requested in line 4 of the code.
- In line 8~9 of the code, clear all the hint information first.

- In code lines 12~14, it traverses the free area from first_free in the block to update the hint information.

Update block metadata according to the free zone

pcpu_block_update()

mm/percpu.c

```
01 /**
02  * pcpu_block_update - updates a block given a free area
03  * @block: block of interest
04  * @start: start offset in block
05  * @end: end offset in block
06  *
07  * Updates a block given a known free area.  The region [start, end) is
08  * expected to be the entirety of the free area within a block.  Chooses
09  * the best starting offset if the contig hints are equal.
10  */
```

```
01 static void pcpu_block_update(struct pcpu_block_md *block, int start, in
   t end)
02 {
03         int contig = end - start;
04
05         block->first_free = min(block->first_free, start);
06         if (start == 0)
07                 block->left_free = contig;
08
09         if (end == PCPU_BITMAP_BLOCK_BITS)
10                 block->right_free = contig;
11
12         if (contig > block->contig_hint) {
13                 block->contig_hint_start = start;
14                 block->contig_hint = contig;
15         } else if (block->contig_hint_start && contig == block->contig_h
   int &&
16                    (!start || __ffs(start) > __ffs(block->contig_hint_st
   art))) {
17                 /* use the start with the best alignment */
18                 block->contig_hint_start = start;
19         }
20 }
```

When the range [@start, @end) of the per-cpu block meter data @block is in the free area, the values related to the free area are updated.

- In line 3~7 of the code, specify the start of the first free area in the block, and record the size in the left_free only if it is free from the start.
- In code lines 9~10, the size is recorded in the right_free only if the end of the block is adjacent to the free area.
- On code lines 12~19, update the largest contiguous free area in the block.

Updating hints for Chunk

pcpu_chunk_refresh_hint()

mm/percpu.c

```
/**
 * pcpu_chunk_refresh_hint - updates metadata about a chunk
 * @chunk: chunk of interest
 *
 * Iterates over the metadata blocks to find the largest contig area.
 * It also counts the populated pages and uses the delta to update the
 * global count.
 *
 * Updates:
 *      chunk->contig_bits
 *      chunk->contig_bits_start
 *      nr_empty_pop_pages (chunk and global)
 */
```

```
tatic void pcpu_chunk_refresh_hint(struct pcpu_chunk *chunk)
{
        int bit_off, bits, nr_empty_pop_pages;

        /* clear metadata */
        chunk->contig_bits = 0;

        bit_off = chunk->first_bit;
        bits = nr_empty_pop_pages = 0;
        pcpu_for_each_md_free_region(chunk, bit_off, bits) {
                pcpu_chunk_update(chunk, bit_off, bits);

                nr_empty_pop_pages += pcpu_cnt_pop_pages(chunk, bit_off,
bits);
        }

        /*
         * Keep track of nr_empty_pop_pages.
         *
         * The chunk maintains the previous number of free pages it hel
d,
         * so the delta is used to update the global counter.  The reser
ved
         * chunk is not part of the free page count as they are populate
d
         * at init and are special to serving reserved allocations.
         */
        if (chunk != pcpu_reserved_chunk)
                pcpu_nr_empty_pop_pages +=
                        (nr_empty_pop_pages - chunk->nr_empty_pop_page
s);

        chunk->nr_empty_pop_pages = nr_empty_pop_pages;
}
```

Updates hints to help search for chunk's maximum sequential free size.

- In line 6 of the code, the maximum contiguous free size is set to 0.
- From code lines 8~11, it iterates through the block metadata corresponding to bits from bit_off and updates the hints that help to search for the maximum contiguous free size of each block.
- Calculate the number of blank pages populate on line 13 of code.
- In lines 24~26 of the code, if the chunk is not for modules, it calculates the total number of blank pages pupulated.
- In line 28 of the code, assign the number of blank pages populated to the members in the chunk.

pcpu_chunk_update()

mm/percpu.c

```
/**
 * pcpu_chunk_update - updates the chunk metadata given a free area
 * @chunk: chunk of interest
 * @bit_off: chunk offset
 * @bits: size of free area
 *
 * This updates the chunk's contig hint and starting offset given a free area.
 * Choose the best starting offset if the contig hint is equal.
 */
```

```
01 static void pcpu_chunk_update(struct pcpu_chunk *chunk, int bit_off, int bits)
02 {
03         if (bits > chunk->contig_bits) {
04                 chunk->contig_bits_start = bit_off;
05                 chunk->contig_bits = bits;
06         } else if (bits == chunk->contig_bits && chunk->contig_bits_start &&
07                    (!bit_off ||
08                     __ffs(bit_off) > __ffs(chunk->contig_bits_start))) {
09                 /* use the start with the best alignment */
10                 chunk->contig_bits_start = bit_off;
11         }
12 }
```

Updates hints that help search for up to a maximum of consecutive free sizes in chunks as long as they @bits from the free area @bit_off.

- If the @bits to free in line 3~5 of the code exceeds the largest contiguous free area in the chunk, these values are updated.
- In code lines 6~11, if the space to be free matches the length of the contiguous free area of the chunk, the starting position of the contiguous space is updated.

Activating a Page (Population)

In very large systems with thousands of NR_CPUS in the NUMA architecture, it is difficult to actually allocate all possible CPUs, so lazy allocation is used.

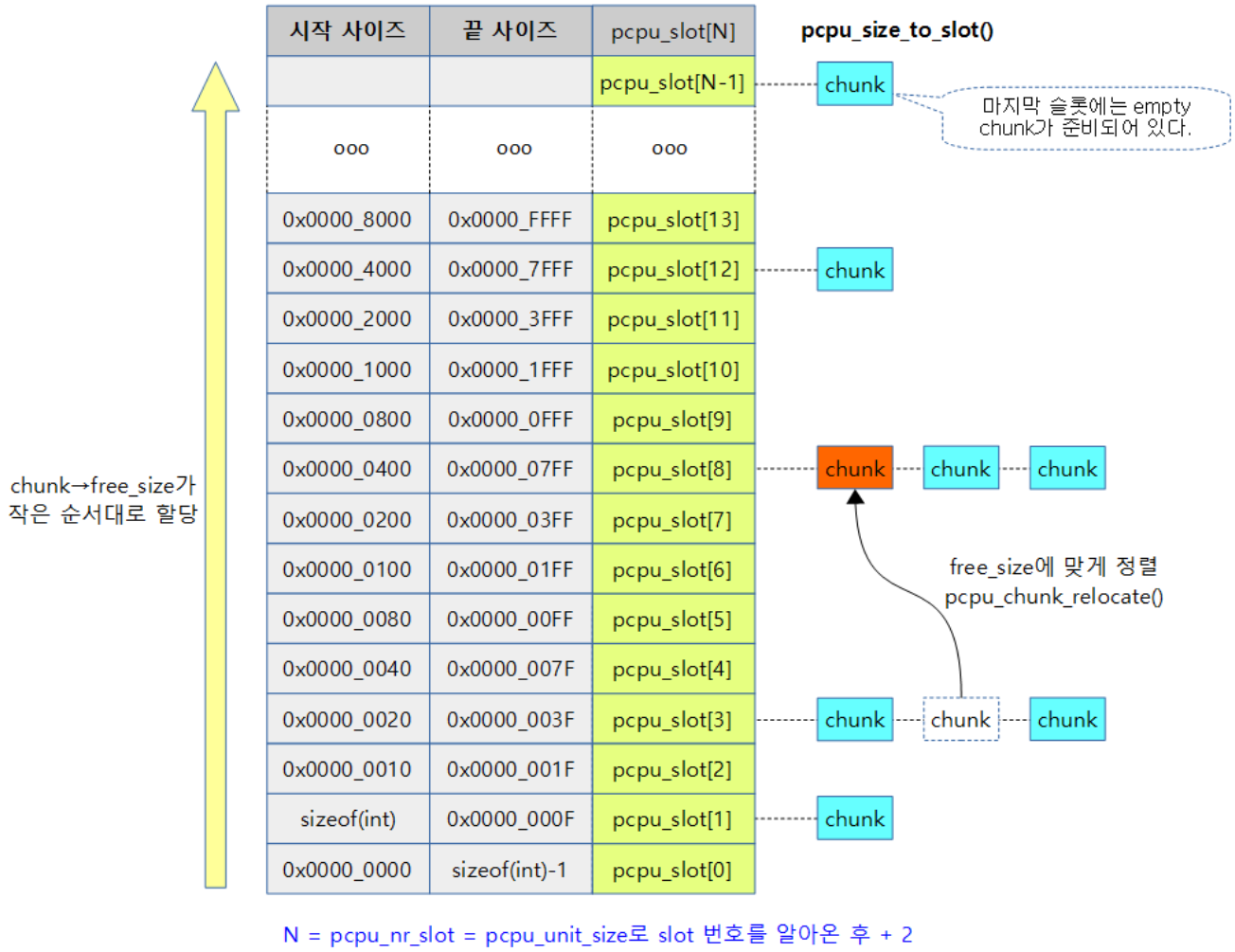
- Newly allocated per-cpu chunks are in an unactivated page state and can be used through the activation process.

Management of Chunks

Here's how to select the chunks you need to dynamically allocate per-CPU memory.

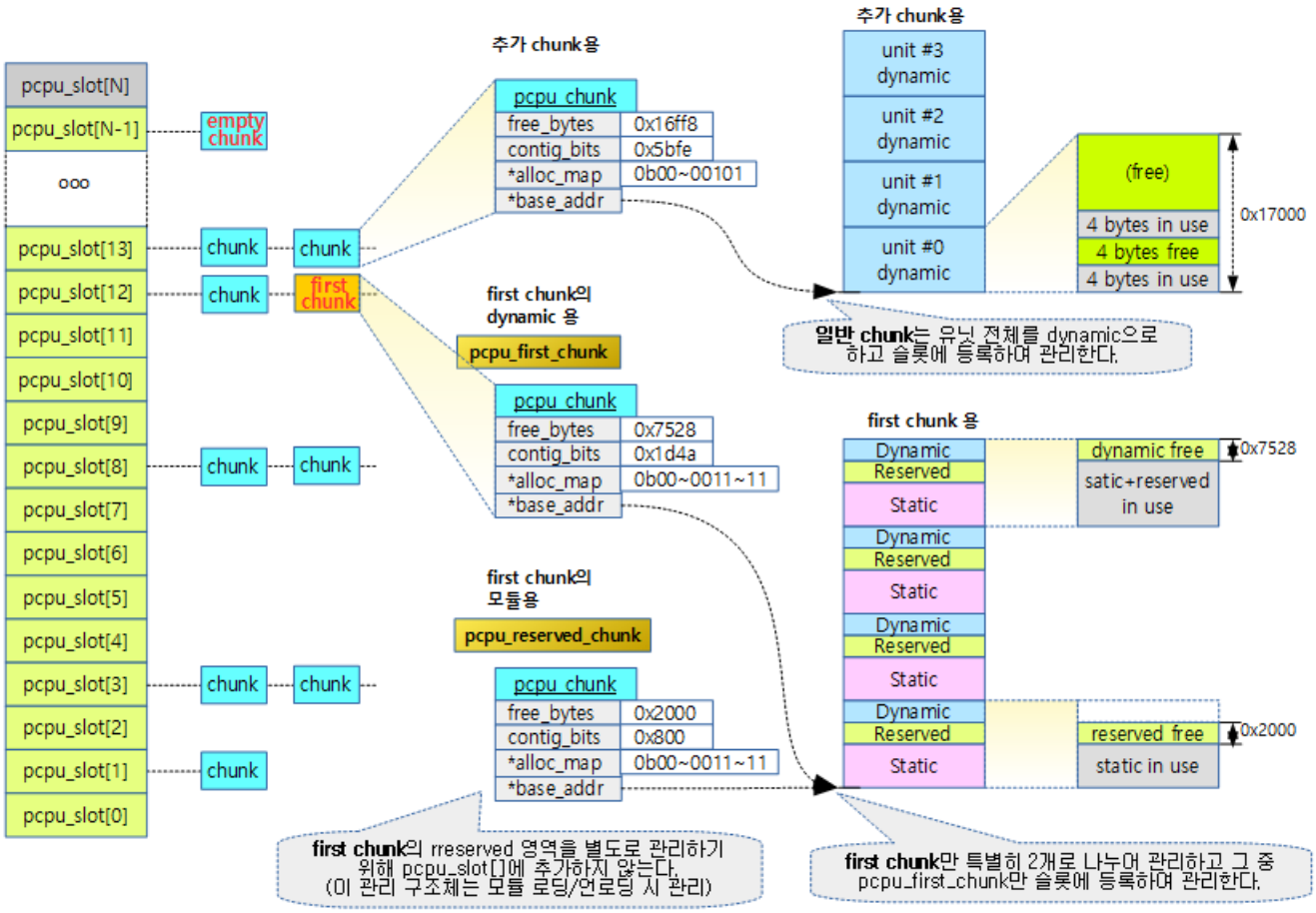
- 1) Calculate the appropriate slot number for the size to be assigned
- 2) Find the chunk with the maximum contiguous empty space (contig_hint) that can cover the size to be allocated from the chunks in the slot.
- 3) If there is no empty space, move to the upper slot and repeat step 2).
  - The top slot always has one empty chunk waiting in place.
  - The bottom 0 slot is where there is a chunk that has no space to allocate.

- The figure below shows where the pcpu_slot list by size of chunks is managed.
  - You can use the pcpu_chunk_relocate() function to move the chunk.
  - RPI2: When the free_size of the first chunk is 0x5140, the first chunk is first added to the pcpu_slot[12] list because nslot=12.



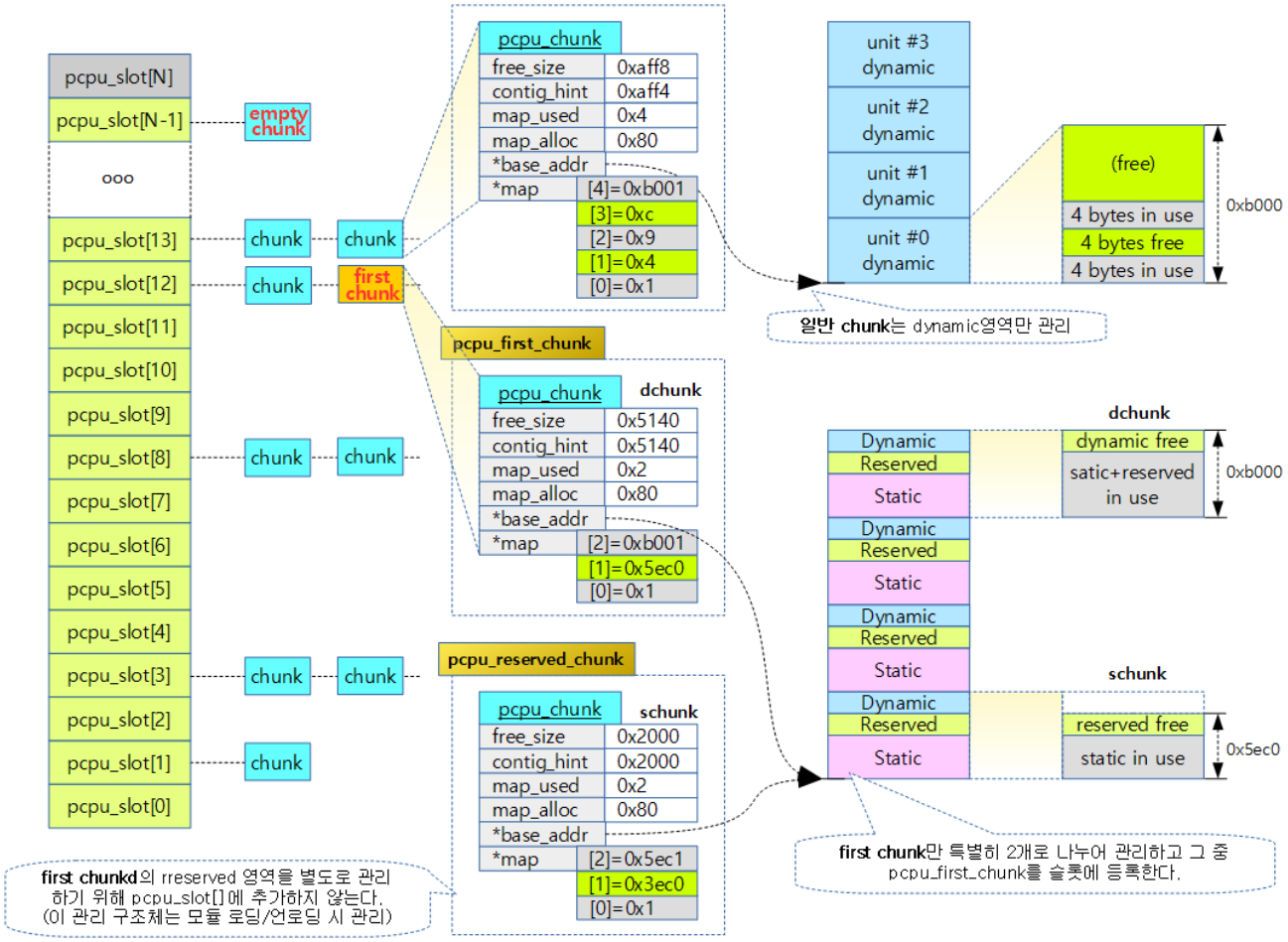
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-16a.png)

The following figure shows how per-CPU chunks are managed in slots, acting as maps managed in a new bitmap method.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/pcpu_setup_first_chunk-2.png)

The following figure shows how per-cpu chunks that work as maps managed in the traditional area map method are managed in slots.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/pcpu_setup_first_chunk-1a.png)

Chunk Relocation

pcpu_chunk_relocate()

mm/percpu.c

```
01 /**
02  * pcpu_chunk_relocate - put chunk in the appropriate chunk slot
03  * @chunk: chunk of interest
04  * @oslot: the previous slot it was on
05  *
06  * This function is called after an allocation or free changed @chunk.
07  * New slot according to the changed state is determined and @chunk is
08  * moved to the slot.  Note that the reserved chunk is never put on
09  * chunk slots.
10  *
11  * CONTEXT:
12  * pcpu_lock.
13  */
```

```
01 static void pcpu_chunk_relocate(struct pcpu_chunk *chunk, int oslot)
02 {
03         int nslot = pcpu_chunk_slot(chunk);
04
05         if (chunk != pcpu_reserved_chunk && oslot != nslot) {
06                 if (oslot < nslot)
07                         list_move(&chunk->list, &pcpu_slot[nslot]);
08                 else
09                         list_move_tail(&chunk->list, &pcpu_slot[nslot]);
10         }
11 }
```

If the chunk's slot needs to be moved, it should be relocated.

- In line 3 of the code, we find the new slot that corresponds to the size in the chunk. When the chunk is exhausted, the slot number becomes 0.
- In line 5~10 of the code, if the requested chunk is not even a chunk for the module, and the slot needs to be moved due to the change in the free size of the chunk, remove the chunk from the list of existing slots and move it to a new slot.
  - If you enter this function while creating the first chunk, the condition is always true because the chunk is the first chunk, and oslot=-1.

Calculate slot numbers with free zone size

pcpu_chunk_slot()

mm/percpu.c

```
1 static int pcpu_chunk_slot(const struct pcpu_chunk *chunk)
2 {
3         if (chunk->free_bytes < PCPU_MIN_ALLOC_SIZE || chunk->contig_bit
  s == 0)
4                 return 0;
5
6         return pcpu_size_to_slot(chunk->free_bytes);
7 }
```

Find out which slot covers the size of the empty space in the chunk.

- In line 3~4 of the code, if there is no space in the chunk to allocate a size of sizeof(int), the nslot will be 0.
- In line 6 of code, the result of pcpu_size_to_slot() is always greater than or equal to 1.
  - The slot number is 4 until the free_bytes is 0~1xF.
  - rpi2: chunk->free_size=0x5140 is nslot=12.

pcpu_size_to_slot()

mm/percpu.c

```
1 static int pcpu_size_to_slot(int size)
2 {
3         if (size == pcpu_unit_size)
4                 return pcpu_nr_slots - 1;
5         return __pcpu_size_to_slot(size);
6 }
```

Returns the slot number based on size. If size is the same as unit size, returns the top-level slot number.

__pcpu_size_to_slot()

mm/percpu.c

```
1 static int __pcpu_size_to_slot(int size)
2 {
```

```
3         int highbit = fls(size);        /* size is in bytes */
4         return max(highbit - PCPU_SLOT_BASE_SHIFT + 2, 1);
5 }
```

Returns the slot number based on size.

- 1~15 will be assigned to slot 1.
- It looks like a bug and should be fixed with +2 -> -1.

PCPU_SLOT_BASE_SHIFT

As shown in the comment in the code below, it seems that the original designer's intent was to collect chunks in slot 31, which is the same slot up to 1 bytes. However, due to the above code bug, 1~15 bytes were assigned to slot 1.

mm/percpu.c

```
1 /* the slots are sorted by free bytes left, 1-31 bytes share the same sl
  ot */
2 #define PCPU_SLOT_BASE_SHIFT            5
```

pcpu_free_alloc_info()

mm/percpu.c

```
01 /**
02  * pcpu_free_alloc_info - free percpu allocation info
03  * @ai: pcpu_alloc_info to free
04  *
05  * Free @ai which was allocated by pcpu_alloc_alloc_info().
06  */
07 void __init pcpu_free_alloc_info(struct pcpu_alloc_info *ai)
08 {
09         memblock_free_early(__pa(ai), ai->__ai_size);
10 }
```

- Call the memblock_free_early() function to delete all pcpu_alloc_info structs from the memblock.
  - Have >__ai_size:
    - pcpu_alloc_info Size including all sub-group information and mapping information
  - Inside the function, we use the memblock_remove_range() function.

pcpu_schedule_balance_work()

mm/percpu.c

```
01 /*
02  * Balance work is used to populate or destroy chunks asynchronously.  W
  e
03  * try to keep the number of populated free pages between
04  * PCPU_EMPTY_POP_PAGES_LOW and HIGH for atomic allocations and at most
  one
05  * empty chunk.
06  */
07 static void pcpu_balance_workfn(struct work_struct *work);
```

```
08 static DECLARE_WORK(pcpu_balance_work, pcpu_balance_workfn);
09 static bool pcpu_async_enabled __read_mostly;
10 static bool pcpu_atomic_alloc_failed;
11
12 static void pcpu_schedule_balance_work(void)
13 {
14         if (pcpu_async_enabled)
15                 schedule_work(&pcpu_balance_work);
16 }
```

If the pcpu_async_enable flag is set, perform schedule_work().

- Use the scheduler to call pcpu_balance_workfn() to manage the number of free chunks and page allocations.
- Reclaim an empty chunk
  - All but one of the completely empty chunks are reclaimed from this routine.
- Keep a few free populated pages
  - Always try to maintain some free populated pages for atomic allocation.
  - Number of pages: PCPU_EMPTY_POP_PAGES_LOW(2) ~ PCPU_EMPTY_POP_PAGES_HIGH(4) range
  - If atomic allocation has failed before, use the maximum value (PCPU_EMPTY_POP_PAGES_HIGH).

consultation

- Per-cpu -1- (Basic) (http://jake.dothome.co.kr/per-cpu/) | 문c
- Per-cpu -2-(initialize) | Sentence C – Current post
- per-cpu -3- (dynamic allocation (http://jake.dothome.co.kr/per-cpu-dynamic)) | Qc
- Per-cpu -4- (atomic operations) (http://jake.dothome.co.kr/per-cpu-atomic) | 문c

---

LEAVE A COMMENT

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

Comments

name *

email *

Website

WRITE A COMMENT

‹ Per-cpu -1- (Basic) (http://jake.dothome.co.kr/per-cpu/)

Bit Operations › (http://jake.dothome.co.kr/bit-operations/)

Munc Blog (2015 ~ 2024)