Per-cpu -1- (Basic)

2016-02-11 (http://jake.dothome.co.kr/per-cpu/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Per-cpu -1- (Basic)

Providing independent memory space for each CPU for the same variable is called a per-cpu variable. In SMP (Symmetry Multi Processing), synchronization mechanisms such as cache consistency and spinlock are required for resources that the cpu accesses at the same time, but per-cpu variables allow access to individual spaces, so there is no need for a synchronization mechanism, which is faster.

Per-CPU is a memory management technique that allocates variables separately by the number of CPUs. This separation of space for each CPU eliminates the need for a locking mechanism between CPUs, which provides fast performance and increases the cache hit rate of the processor.

Key Features of Per-CPU

The per-CPU area was added in kernel 2.6. By creating a per-cpu zone of data so that each cpu has a copy, there is less to consider synchronization if you only have access to data that is relevant to your own cpu in an SMP environment. It is used in designs that do not use locks and want to repeat fast writes. Protection against task preemption and interrupts is necessary.

It is often used for counting network packets, disk, and kernel objects, and it is fast because it is performed without the use of a locking mechanism when thousands of updates per second are required. It caches effectively on each CPU and is fast because it doesn't have cache line bouncing issues with shared memory.

In the initial design, each per-CPU data was used as byte aligned as the number of cache lines, but now the kernel declares per-CPU data separately according to its type, so it is used in conjunction with other per-CPU data without sorting (read-most, etc.).

Per-CPU data is defined in a separate per-CPU address space rather than the regular code or data address space used by the kernel, and can be used to filter out zone invasion errors when using a sparse static code analysis tool. The per-CPU data staticly defined in the kernel image is organized in the first chunk. Dynamic per-CPU allocation was added as a patch in 2009, introducing chunk. The asynchronous chunk population was added to kernel 2014.9-rc3 in September 18.

- Consultation:

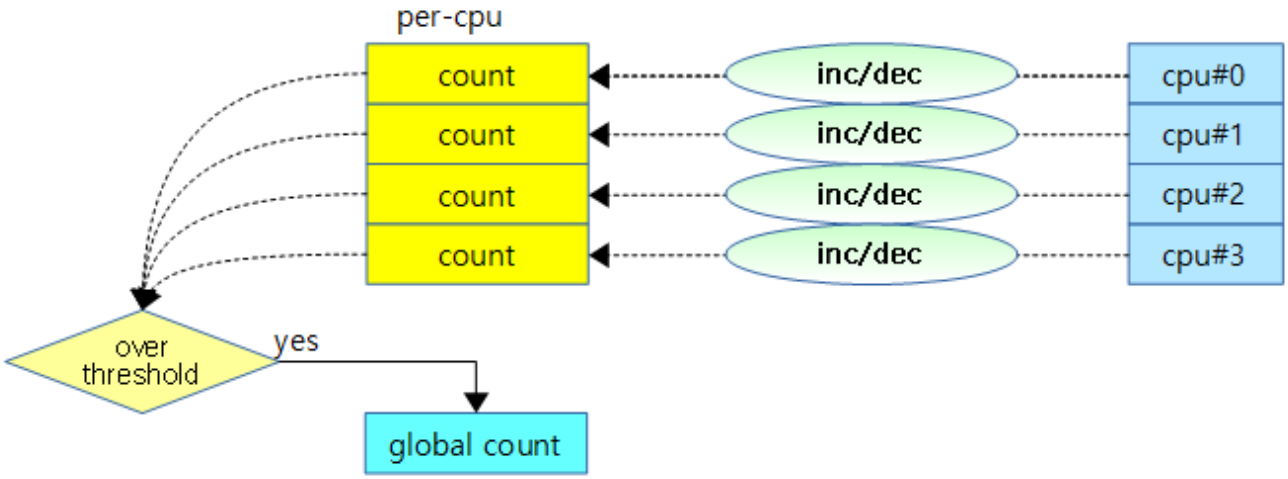
    - percpu: implement asynchronous chunk population (https://github.com/torvalds/linux/commit/1a4d76076cda69b0abf15463a8cebc172406da25)
    - percpu: implmeent pcpu_nr_empty_pop_pages and chunk->nr_populated (https://github.com/torvalds/linux/commit/b539b87fed37ffc16c89a6bc3beca2d7aed82e1c)

Per-CPU Uses

Let's take a look at some of the most common uses for per-CPUs.

- Cache Objects
    - Lists and trees used throughout the world are allocated to each CPU and used as caches. While the performance is fast, the disadvantage is that it uses as much memory as the number of CPUs, which increases the memory requirements. To compensate for this, it acts as a cache that does not manage the entire list or tree, but only provides a limited part of it (PCP cache on the buddy system, Per-CPU cache applied to slab objects, LRU cache).
- Stats Counter
    - Various statistical counters are very frequent with simple increases and decreases. If you use the stat counter variable as a per-CPU, you can use it without a lock mechanism to remember the value of each CPU increase or decrease, and if the threshold is exceeded, it will be added to the global stat counter variable. If it is okay to use an approximate value of the counter value, only the value of the global statistic counter variable should be read, and if a precise value is required, the value of the per-cpu statistic counter that has been increased or decreased on each CPU should be applied to the value of the global stat counter. In order to guarantee the accuracy of the global counter to a certain extent, if the size of the increase or decrease on each CPU exceeds a certain threshold, the global counter is updated (the update of the global counter uses atomic operations).
- Others
    - RCU infrastructure
    - Profiling, Ftrace
    - VFS components

The following figure shows a case where it is used as a per-cpu counter.



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

Unit: How much space is used to store per-CPU data per CPU?

A unit is a space where per-CPU data corresponding to a single CPU is stored. The unit is divided into three types of storage areas: The size of the unit is the value of the following three areas added and sorted upwards on a page-by-page basis.

- Static Area
  - Static per-CPU data is stored in the kernel using the DEFINE_PER_CPU( ) macro.
  - RPI2: Yes) 0x3ec0
- reserved area
  - Static per-CPU data using the DEFINE_PER_CPU( ) macro in the module is stored.
  - rpi2: e.g. default 0x2000 (8K)
- Dynamic Zones
  - **Dynamic** per-CPU data is stored via the alloc_percpu( ) function in the kernel or module.
  - rpi2 e.g. default 0x5000 (20K)
    - On 64-bit systems, the default 28K

There are two main variables involved

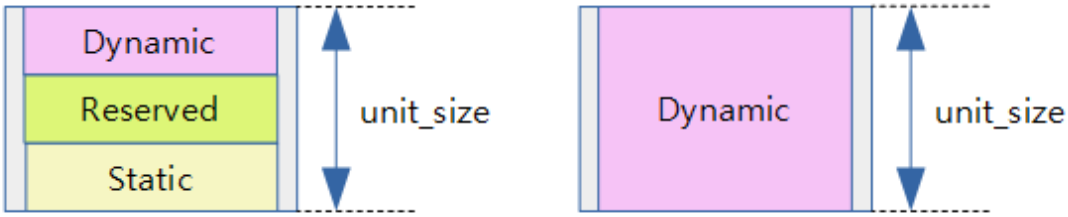
- pcpu_unit_pages
  - It contains the number of pages used by one unit.
  - RPI2: Yes) 11
- pcpu_unit_size
  - It contains the byte size used by one unit.
  - rpi2: Yes) 0xb000 (44 KB = 11 * 4 KB (PAGE_SIZE))

For systems with 4K small pages, such as ARM, multiple pages are used in a single unit when allocating per-cpu area. And use that area in a page-by-page arrangement. Add all the static_size + reserved_size + dyn_size and align the 1K and add the remaining space to the dynamic area.

- rpi2: e.g. 0x3ec0 + 0x2000 + 0x5000 + 0x140 (4K align to add extra space to the dynamic area)

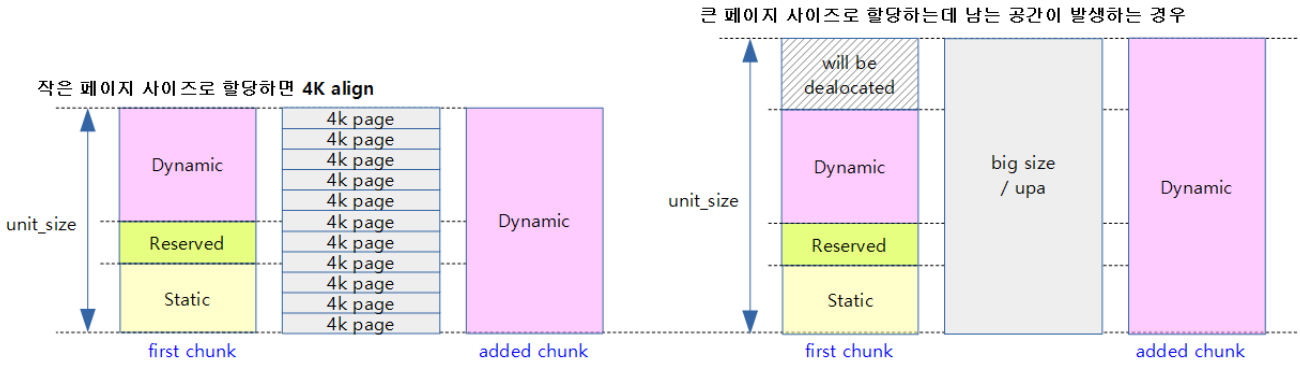
In some architectures, such as x86_64 and ARM64, large pages, such as 2M pages, can be allocated using them. Due to the large size of this space, multiple units can be divided into one allotment size. In the process of placing a unit, there is a surplus space at the end of the unit, which is released from memory allocation.

The following figure shows that when the per-CPU area was first created, it was divided into three zones. Also, when creating an additional per-cpu region, it shows a unit containing only the dynamic region with the same unit size.



The following illustration shows that the assignment page is divided into two types: small and large.

- Compare the dynamic areas between the first chunk and the added chunk.
- You can see that only the first chunk has static and reserved areas.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-17a.png)

CPU -> Unit Mapping

Per-CPU data uses NR_CPUS arrays, so using a larger number than the actual number of CPUs in existence can result in wasted memory. This is because a system that is configured to support thousands of CPUs (NR_CPUS) sometimes has only a few CPUs that are capable of running at actual boot.

- Note: cpu alloc v1: Optimize by removing arrays of pointers to per cpu objects (https://lwn.net/Articles/258248/) | LWN.net

In order to reduce wasted per-CPU arrays, additional patches have been applied in some architectures to make efficient use of space. In a NUMA system, each node has a different number of CPUs, and if you use a large page, the number of units is the same for each allocated memory. Instead of always mapping the cpu and units the same, we mapped them to a non-linear/sparse >unit for NUMA.

This can be easily implemented by adding a CPU->Unit mapping array and can be used to find matched units. When configured with an asymmetric NUMA system, the number of units and the number of CPUs are not the same because some units are not mapped.

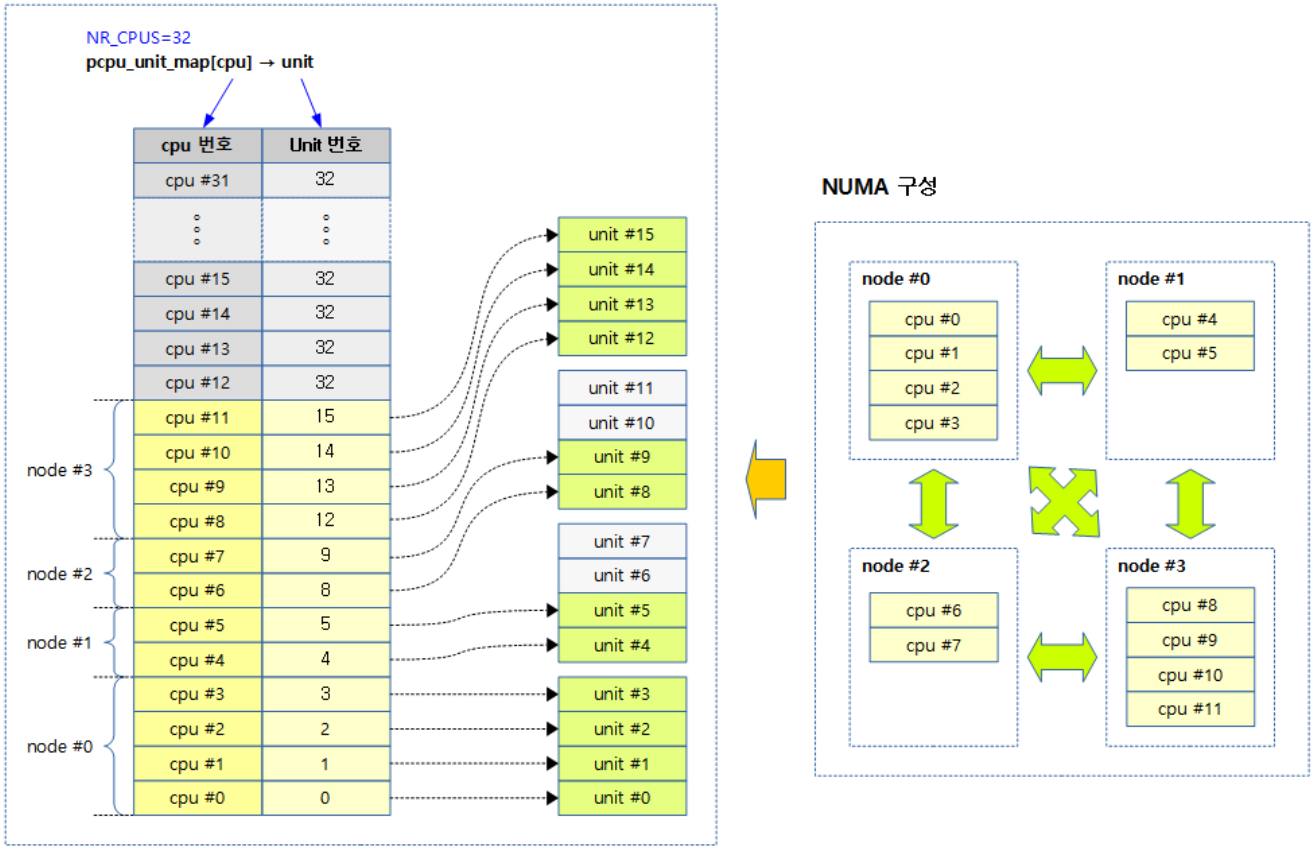
The arrangement used here is as follows:

- pcpu_unit_map[]
  - The CPU ID is mapped to the Unit Index.
  - The mapping is valid for all chunks.
  - e.g. 0,1,2,3
- pcpu_unit_offsets[]
  - The baseaddr of that chunk contains the offset to each unit.
  - The unit offset is the same for all chunks.
  - e.g. 0, 0x8000, 0x10000, 0x18000

NUMA support for per-CPUs

To support NUMA, per-cpu adds non-linear/sparse cpu->unit mapping to the same mapping of the cpu and units. This can be easily implemented by adding a CPU->Unit mapping array and can be used to find matched units. The base address of unit0 is not fixed. If the mapping changes, the base address of unit0 changes.
In ARM64, the group mapping structure information below is not used after kernel initialization, but rather global variables are used for the actual cpu->unit mapping. In the x86 and SPARC64 architectures, the CPU->unit mapping is done using the following two structure information even after kernel initialization.

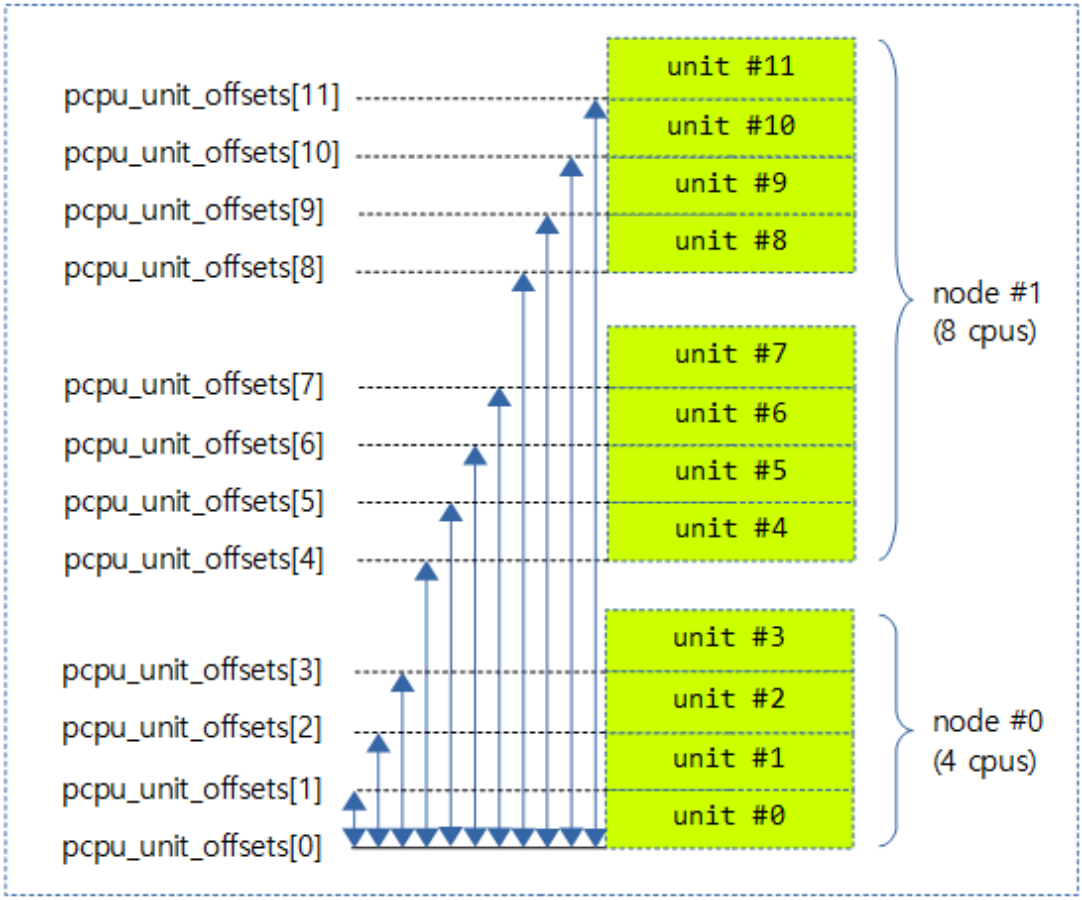
The following figure shows the unit number mapped to the CPU number available in the NUMA system. (Of the 32 NR_CPUS, there are 12 units that correspond to the actual 16 possible CPUs.)



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

The following diagram shows that the starting offset of each unit is specified from the unit at the lowest address.

- The initialization of the pcpu_unit_offset[] array is done by reference to the pcpu_setup_first_chunk() function.



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

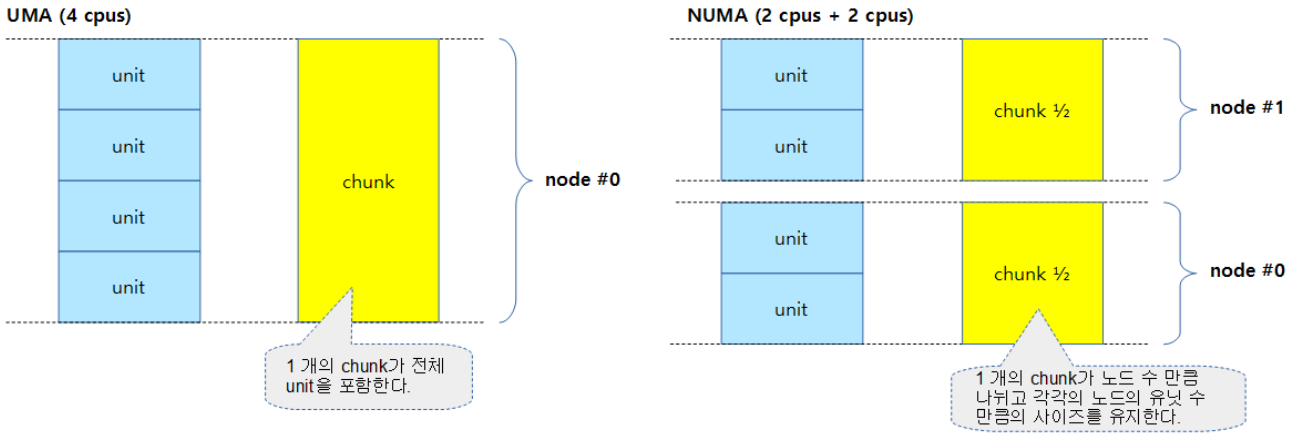
Chunk

A chunk is a unit that collects all units corresponding to all CPUs, and the characteristics are as follows:

- One chunk is divided by nr_units number
- If you run out of chunks, create and use additional chunks
- Each unit in every added chunk that is not the first chunk only has a dynamic area
- All chunks are managed as pcpu_chunk structs
- In a NUMA system, one chunk is allocated using its own node (group) memory

The first chunk is the first chunk created during the kernel initialization process. Each unit in the First Chunk is divided into three areas: Static, Reserved, and Dynamic.

The following figure shows that a chunk is divided and allocated by the number of nodes.

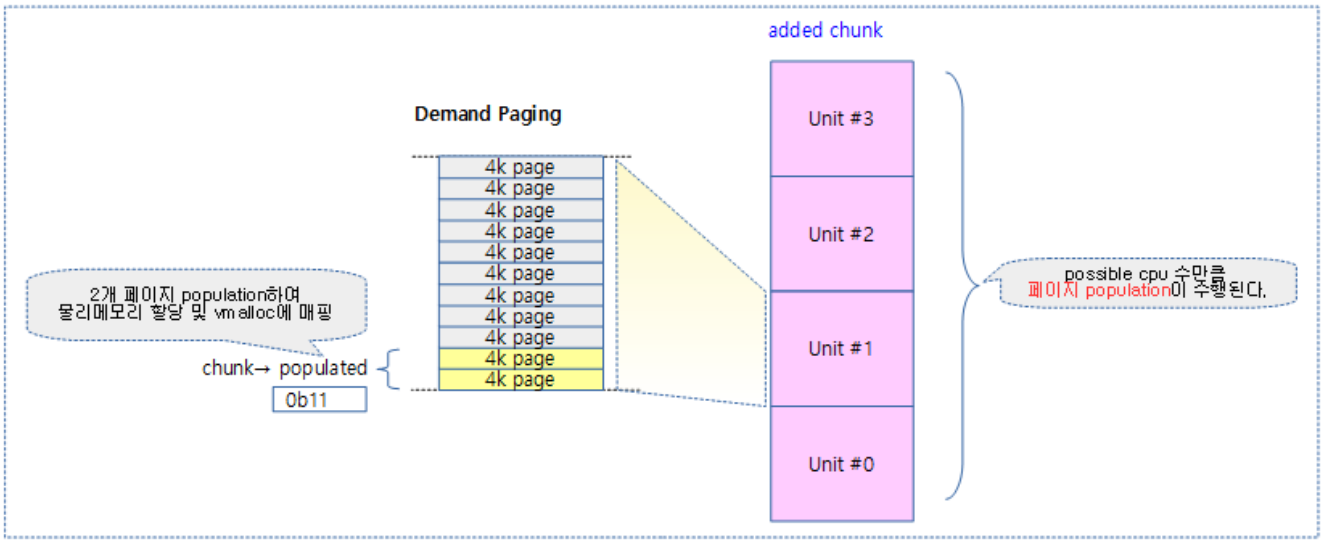


(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-13.png)

Demand Paging (Population by Page)

- All additional chunks use the Demand Paging method to save memory.
  - Exclude the first chunk because it is already mapped
- When a new chunk is allocated, it does not allocate and map pages contained in every unit, but only when there is an actual per-cpu allocation request, which allocates physical memory for those pages and maps them to vmalloc space. This is called the page population.
- Whether or not it is populated for the entire page configured within a chunk is managed by a bitmap, a populated member variable.
  - When per-CPU data is allocated and used, if the pages corresponding to the specified area of the chunk are not set to a populated bitmap, the allocation of physical memory pages and mapping by vmap is performed.

The following illustration shows the populated pages.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-30.png)

Configuring First Chunk

The first chunk is a chunk that plays a key role in determining the layout of each group and unit, and is specially managed. At compile time, .data.. percpu section and copied to each static area of the first chunk in the boot init routine setup_per_cpu_areas ( ).

reserved areas are initialized to PERCPU_MODULE_RESERVE size. If you use a module, it will be 8K, and if you don't use a module, it will be 0K. The dynamic area is initialized to the PERCPU_DYNAMIC_RESERVE size. On a 32-bit system, it will be 20K, and on a 64-bit system, it will be 28K. Until the v3.17 kernel, they were smaller than they are now, at 12K and 20K, respectively.

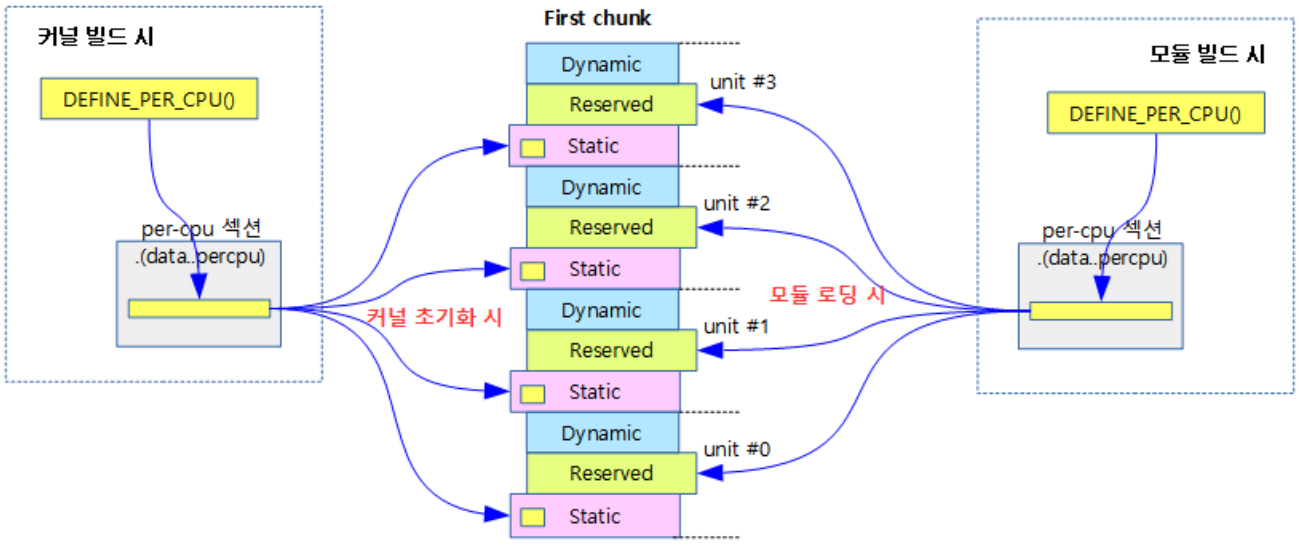
There are two ways to do this, and you can choose one of them as a percpu_alloc option at boot time.

- Embed method
  - If your architecture supports it, use huge page sizes for unit allocation to increase the efficiency of the TLB cache.
    - ARM64 uses the embed method, but does not yet support the use of huge pages on per-cpu.
  - When constructing a first chunk, space is left behind the dynamic of each unit. This space can be deallocated and used for other purposes
  - In the case of UMA systems, the first chunk is allocated and used at a time.
  - In the case of a NUMA system, the first chunk is divided and allocated to each node's memory.
  - If you add chunks, you should allocate them from top to bottom in the vmalloc space so that they are spaced at the same spacing as the node-by-node configuration when you created the first chunk.
    - In the case of 32-bit NUMA systems, the vmalloc space is very small (arm=240M, x86=120M), so if the embed method cannot be used, it is switched to the page method.
      - In order to account for the placement of additional chunks of vmalloc space, the embed method is restricted if the farthest max_distance between the base addresses of the chunks allocated to each node exceeds 75% of the vmalloc space.
        - Note: percpu: make embedding first chunk allocator check vmalloc space size (https://github.com/torvalds/linux/commit/6ea529a2037ce662fc6bfa572b46d47407d08805)
      - On x86, if it can't be generated in an embed way, it will automatically retry using the page method.
  - Search for the empty vmalloc space from top to bottom.
    - When allocating vmalloc space by the vmalloc() and vmap() functions, the vmalloc space is searched from bottom to top, and the chunk of the per-cpu does the opposite, preventing chunks from overlapping each other with each additional chunk as much as possible.
    - When allocating the vmalloc space in the NUMA system, the per-cpu should be generated in the opposite direction to prevent it from mixing with the normal

vmalloc area, so that it can be generated quickly and the probability of failure is reduced.

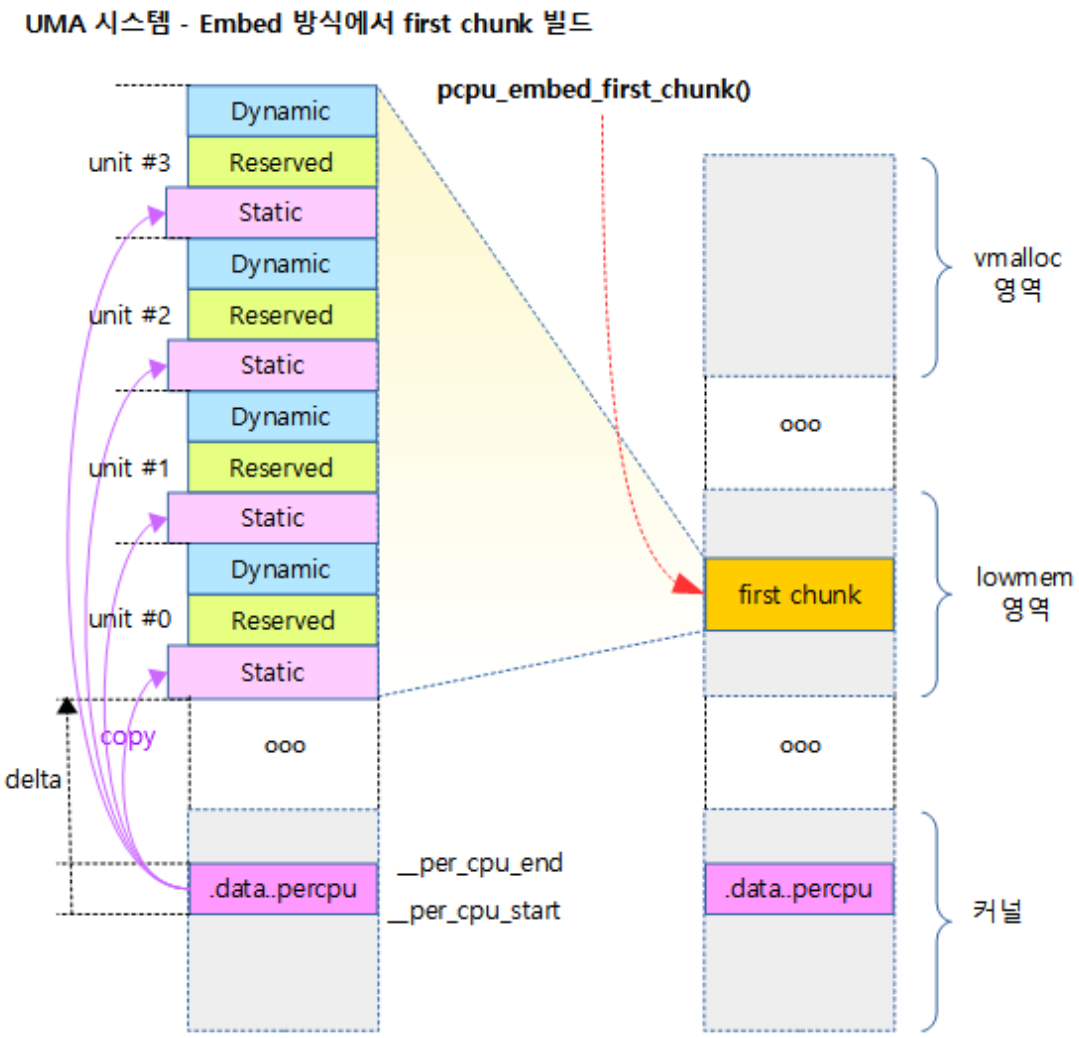
- Paged Method
  - When constructing the first chunk, it is allocated and mapped to the VMALLOC space for a minimum number of pages in the vmalloc space.
    - Since the slub memory allocator is not yet working at kernel initialization, mapping to the vmalloc region is not possible, so the early registration method is used so that the slub memory allocator can be registered after it has been activated.
    - After the slub memory allocator is running, call the percpu_init_late() function to read, allocate, and map all the chunk maps that you have already created.
  - Instead of dividing chunks into nodes in vmalloc space, the number of units is configured in succession to the number of CPUs possible.
  - Since it doesn't use large pages, it may be slower than the embed method, but it can still be used on 32-bit NUMA systems with less vmalloc space.
  - Search for the empty vmalloc space from the bottom to the top.
    - Since you don't need to divide chunks by the number of nodes in the vmalloc space, you can use the same way you allocate vmalloc space by the vmalloc() and vmap() functions.

The following figure shows that static per-cpu data is declared and copied to the static area of each unit of the first chunk when the kernel is initialized after the build, and also copied to the reserved area of each unit of the first chunk when the module is loaded.



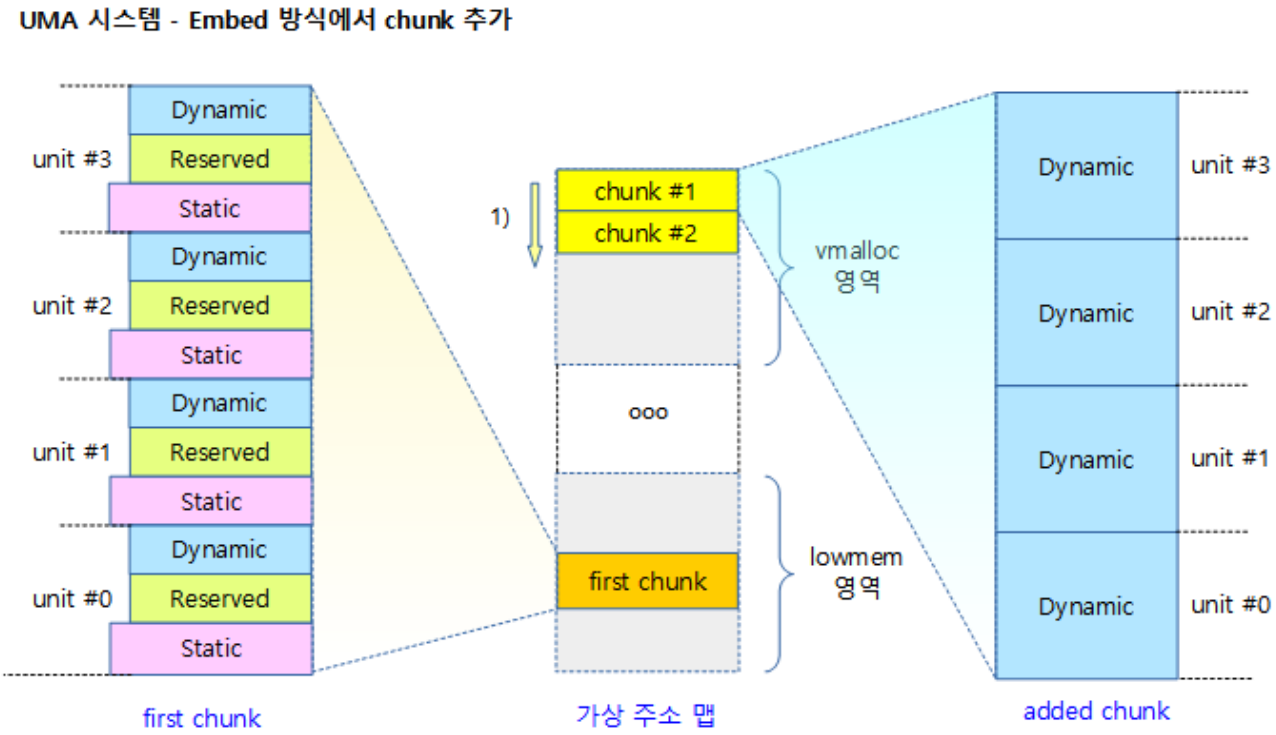
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-14.png)

The following figure shows the location of the space where the first chunk is created using the embed method in the UMA system.



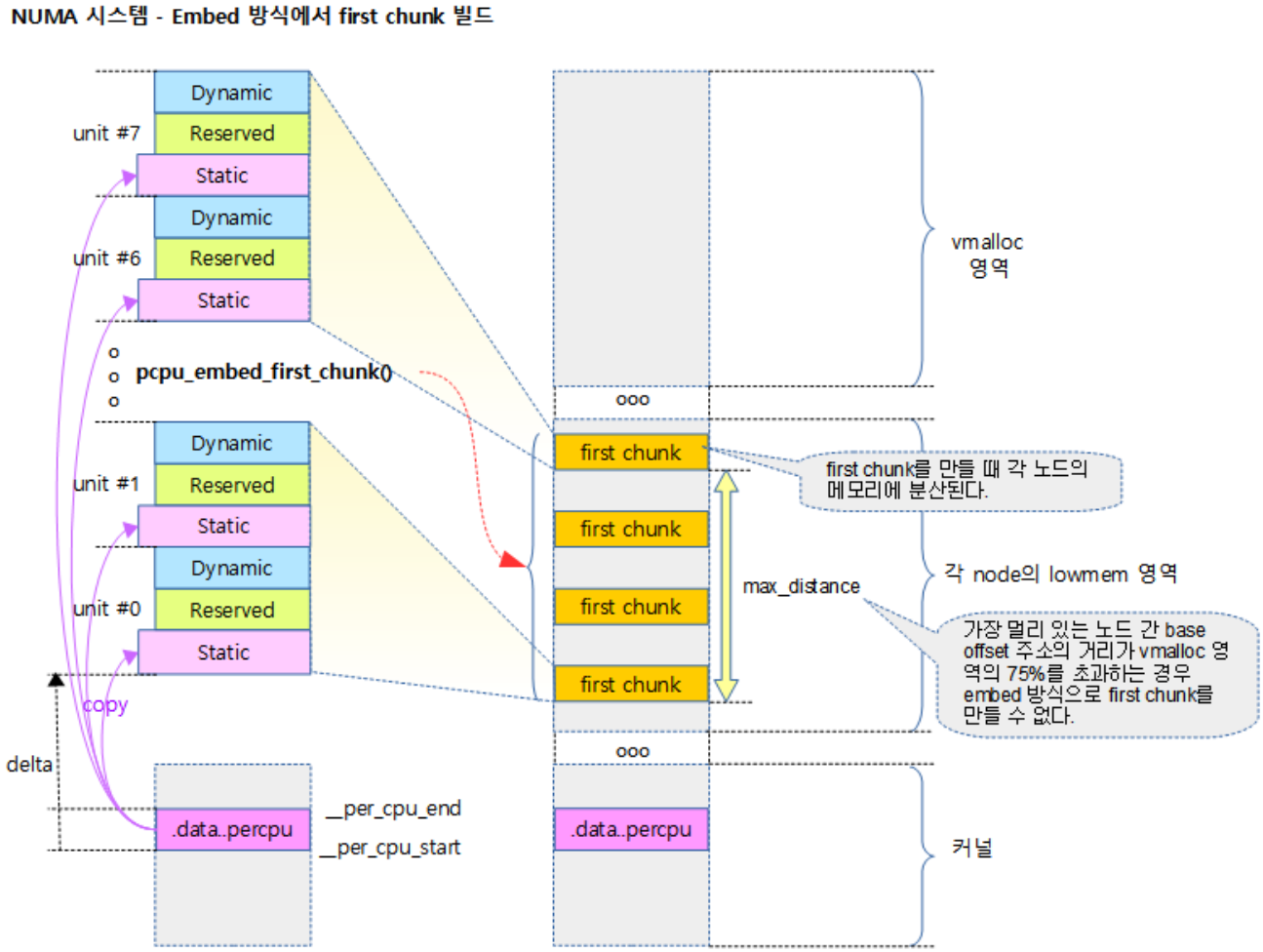
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-15.png)

The following figure shows the location of the space where additional chunks are created using the embed method in the UMA system.



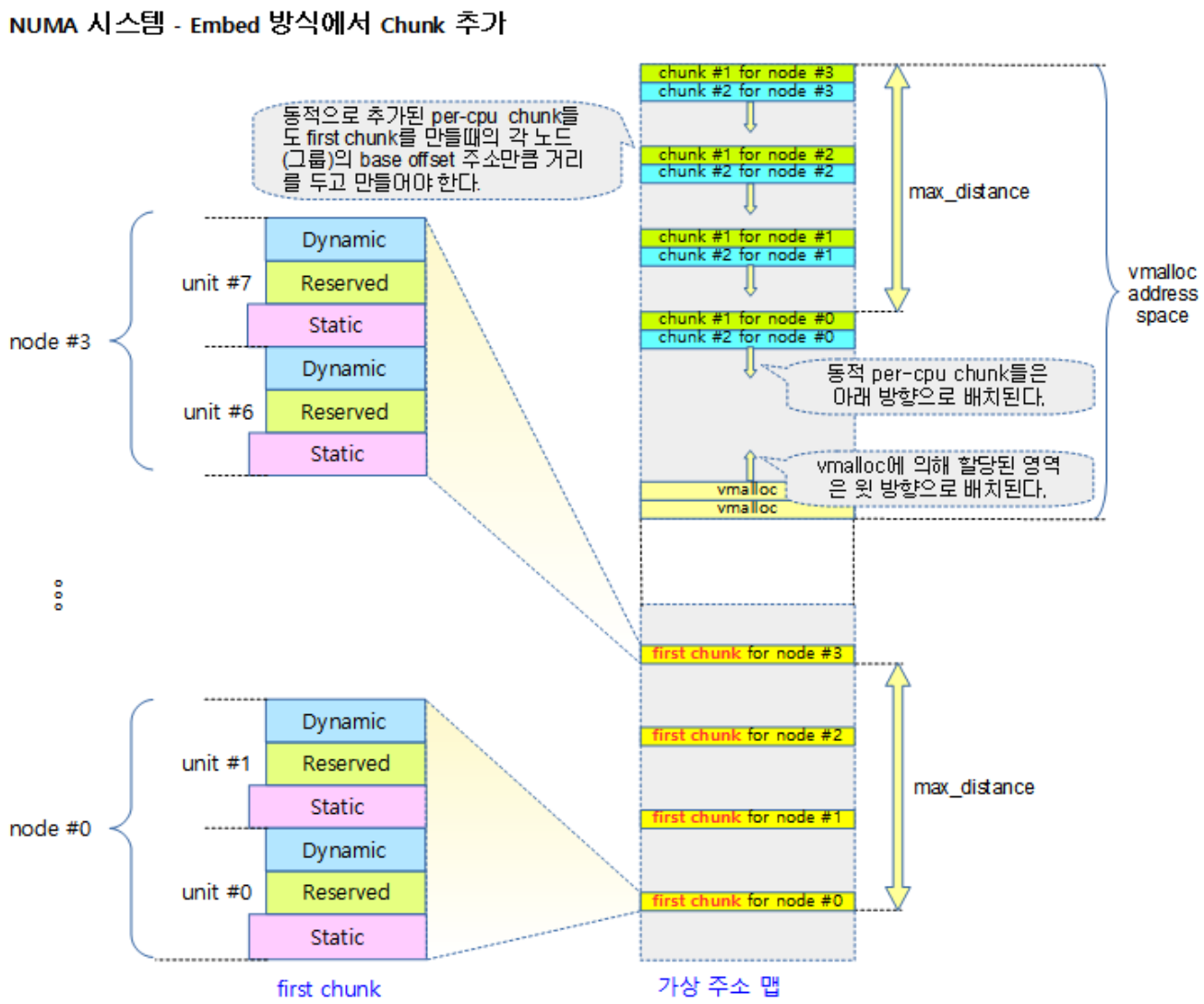
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-16.png)

The following figure shows the location of the space where the first chunk is created using the embed method in the NUMA system.



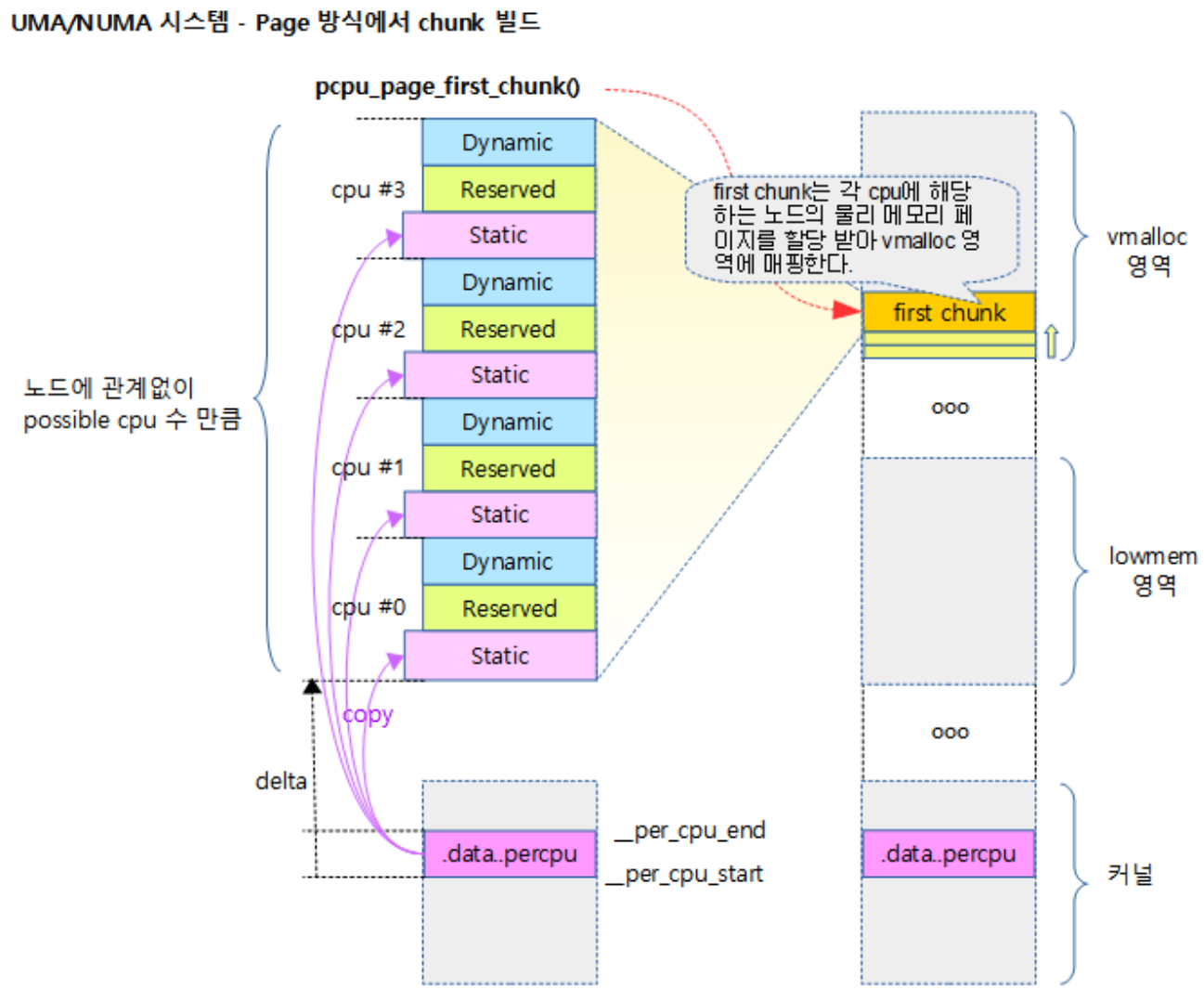
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-17.png)

The following figure shows the location of the space in which additional chunks are created using the embed method in the NUMA system.



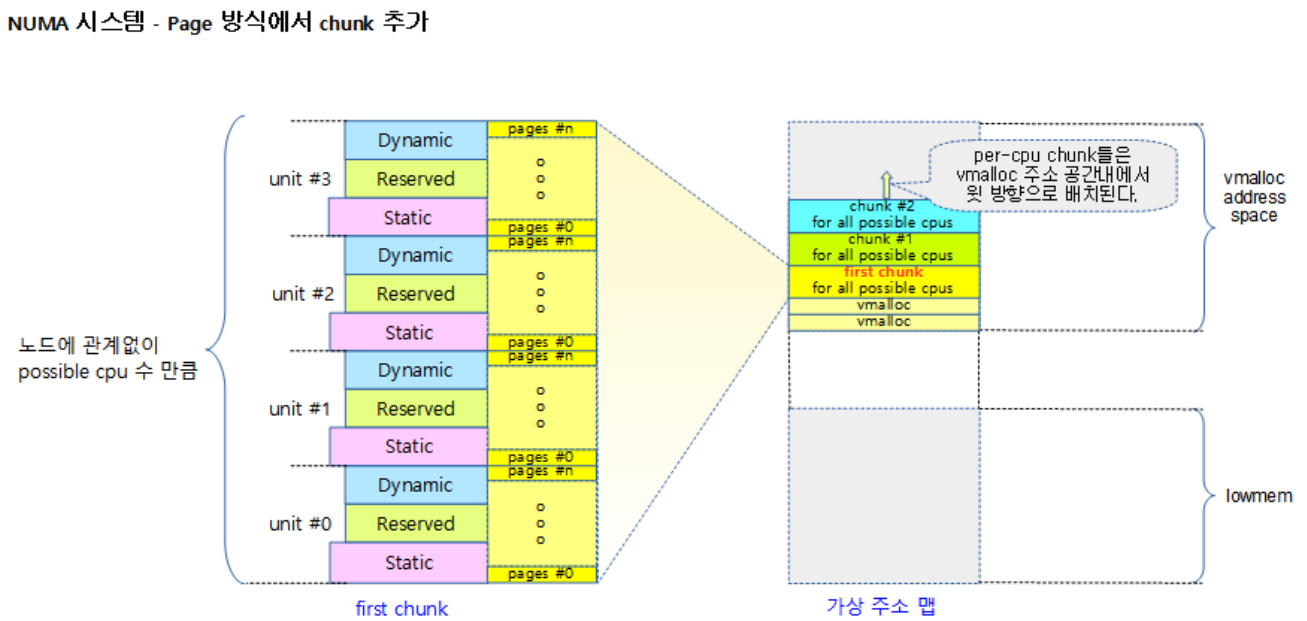
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-18.png)

The following figure shows the location of the space in which the first chunk is made using the Paged method.



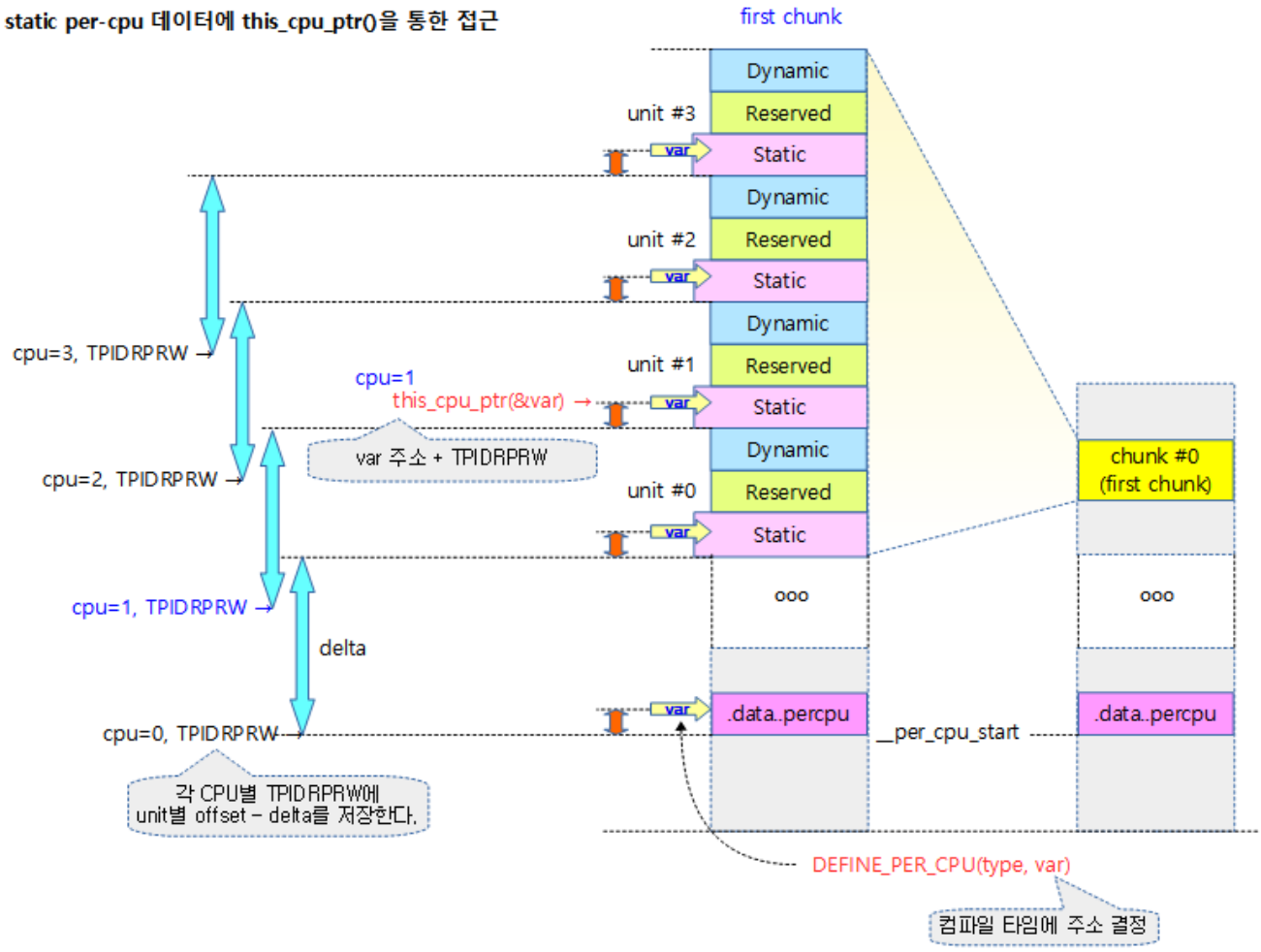
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-19.png)

The following figure shows the location of the space where additional chunks are made using the Paged method.



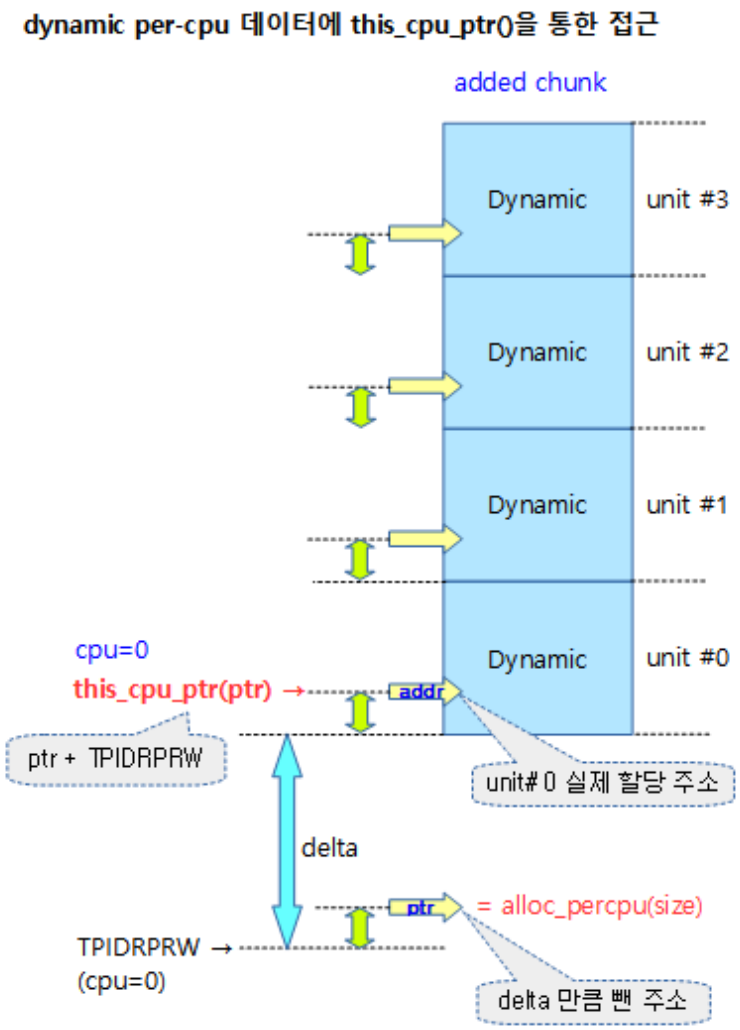
(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-20.png)

The following figure shows the static per-cpu data accessed via the this_cpu_ptr() function.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-21.png)

The following figure shows accessing the dynamic per-cpu data of a chunk added via the this_cpu_ptr() function.

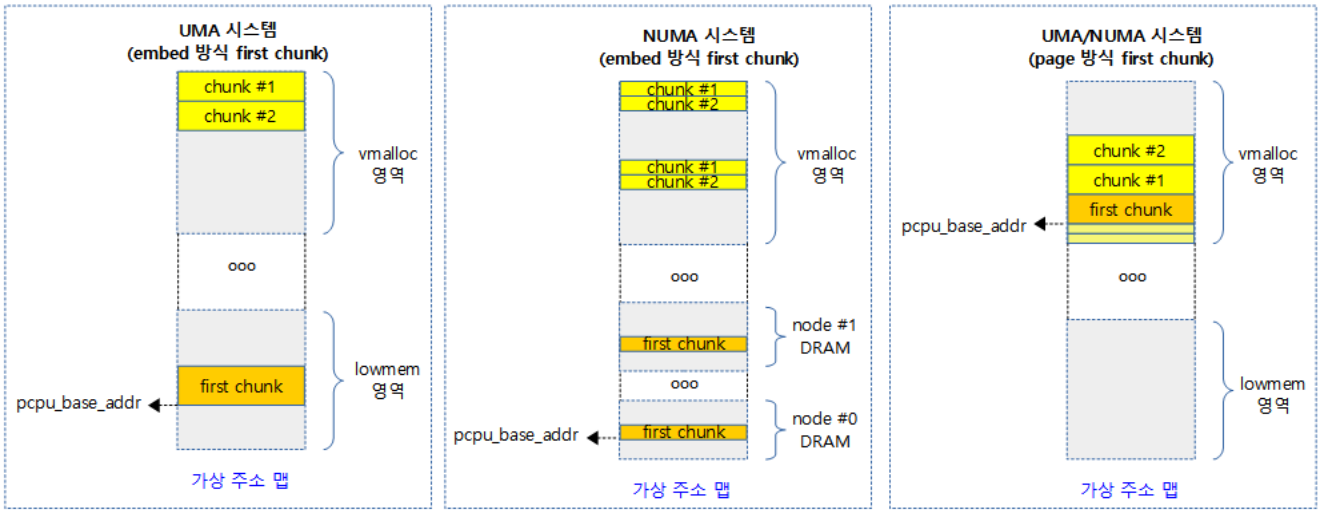


(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-22.png)

pcpu_base_addr

As shown in the following figure, the pcpu_base_addr contains the base address of the first chunk.

- Depending on the architecture, the vmalloc space may be located below the lowmem area. Note that ARM32 has a vmalloc space above lowmem, and ARM64 has a vmalloc space below lowmem.



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

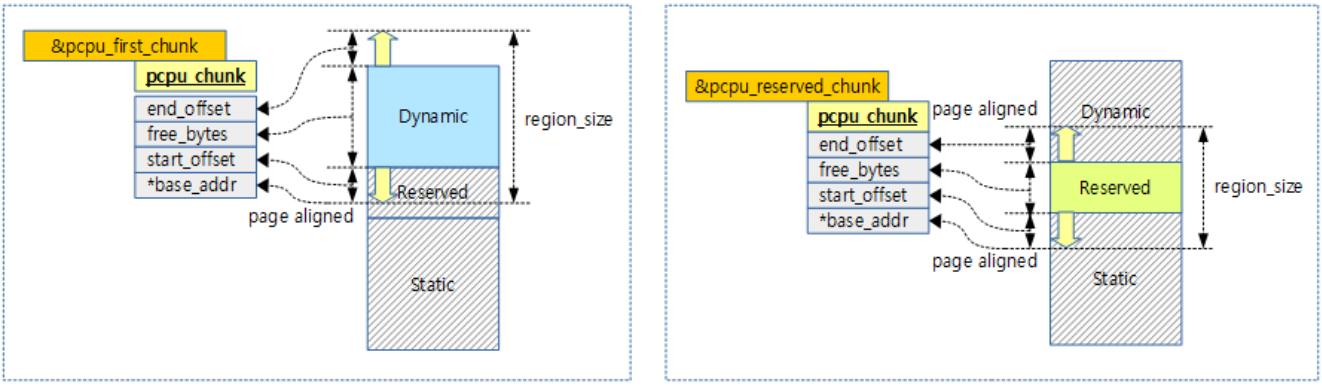
Managing First Chunk's Map Entries

Mapping Management in First chunk

These chunks are specially managed, playing an important role in determining the placement of each group and unit. For one allocation area of the first chunk, the management chunk map can consist of one or two. pcpu_first_chunk manages the dynamic space in the same way as other chunks, and is used in addition to pcpu_slot[ ]. pcpu_reserved_chunk is not added to the pcpu_slot[ ], but is used by the module management mechanism.

If you're using modules in the kernel, pcpu_reserved_chunk have global variables manage the reserved areamap entries for the first chunk. Each time a kernel module is loaded or unloaded, the per-cpu data of the DEFINE_PER_CPU( ) macro used by the kernel module is added to the reserved area.

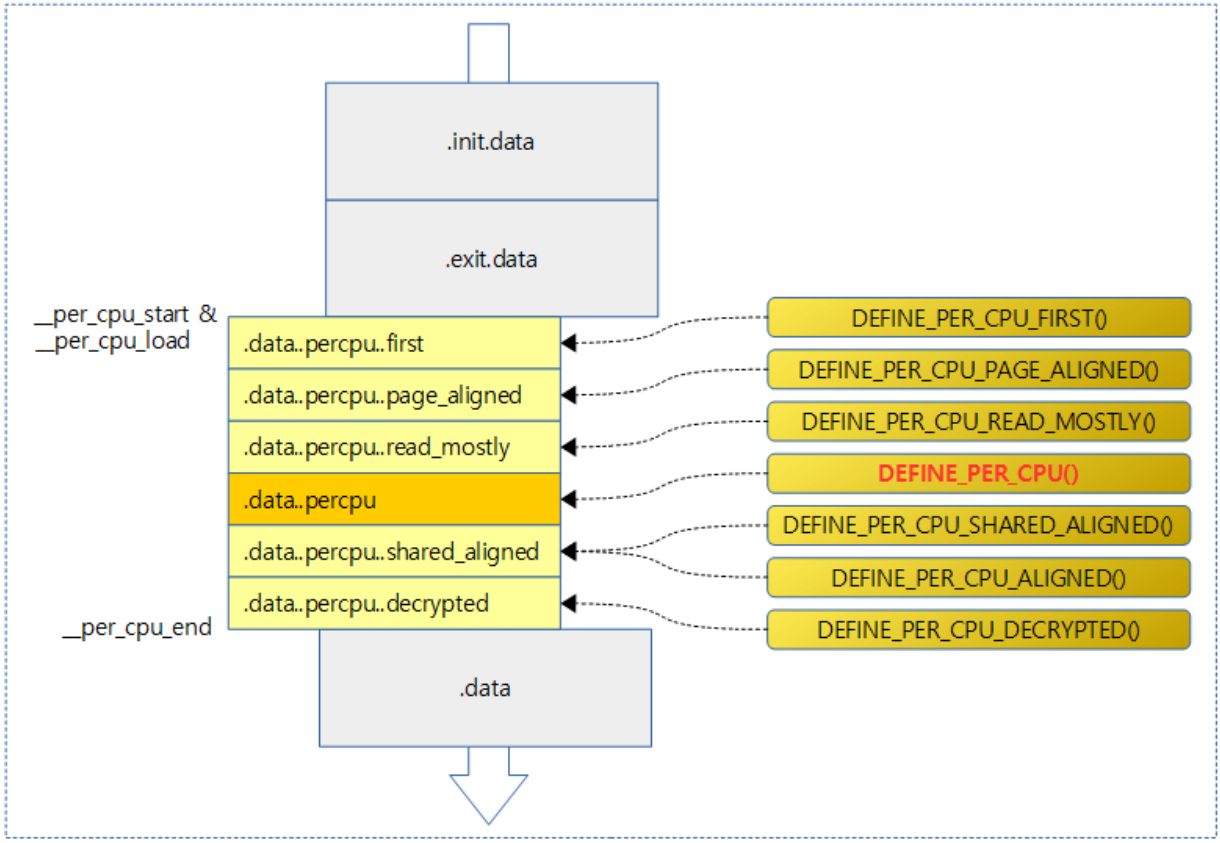
The following figure shows how the management structure of the first chunk is managed in either one or two parts, depending on whether the kernel module is used or not.



(http://jake.dothome.co.kr/wp-content/uploads/2016/02/setup_per_cpu_areas-11d-1.png)

Per-CPU Area Storage Section

_per_cpu_load is the virtual address at which the per-CPU area is loaded into memory. To improve performance, various macros are available to store per-CPU data in different areas, as shown in the following figure.



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

Per-CPU Section Location

The following vmlinux.lds.S linker script will show that the per-cpu section is located under the .init.data section.

for ARM32

arch/arm/kernel/vmlinux.lds.S – ARM32

```
.init.data : {
        INIT_DATA
        INIT_SETUP(16)
        INIT_CALLS
        CON_INITCALL
        INIT_RAM_FS
        *(.init.rodata.* .init.bss)     /* from the EFI stub */
}
.exit.data : {
        ARM_EXIT_KEEP(EXIT_DATA)
}

PERCPU_SECTION(L1_CACHE_BYTES)
```

for ARM64

arch/arm64/kernel/vmlinux.lds.S – ARM64

```
        INIT_DATA_SECTION(16)

        .exit.data : {
                ARM_EXIT_KEEP(EXIT_DATA)
        }
```

```
6
7  #ifdef CONFIG_SMP
8          PERCPU_SECTION(L1_CACHE_BYTES)
9  #endif
```

PERCPU_SECTION()

include/asm-generic/vmlinux.lds.h

```
01 /**
02  * PERCPU_SECTION - define output section for percpu area, simple versio
   n
03  * @cacheline: cacheline size
04  *
05  * Align to PAGE_SIZE and outputs output section for percpu area.  This
06  * macro doesn't manipulate @vaddr or @phdr and __per_cpu_load and
07  * __per_cpu_start will be identical.
08  *
09  * This macro is equivalent to ALIGN(PAGE_SIZE); PERCPU_VADDR(@cachelin
   e,,)
10  * except that __per_cpu_load is defined as a relative symbol against
11  * .data..percpu which is required for relocatable x86_32 configuration.
12  */
13 #define PERCPU_SECTION(cacheline)
    \
14         . = ALIGN(PAGE_SIZE);
    \
15         .data..percpu   : AT(ADDR(.data..percpu) - LOAD_OFFSET) {
    \
16                 __per_cpu_load) = .;                    \
17                 PERCPU_INPUT(cacheline)
    \
18         }
```

Define an output section for the per-CPU area.

- LOAD_OFFSET
  - The value defined in asm-generic is 0 and is used unchanged in the ARM architecture.

PERCPU_INPUT()

include/asm-generic/vmlinux.lds.h

```
01 /**
02  * PERCPU_INPUT - the percpu input sections
03  * @cacheline: cacheline size
04  *
05  * The core percpu section names and core symbols which do not rely
06  * directly upon load addresses.
07  *
08  * @cacheline is used to align subsections to avoid false cacheline
09  * sharing between subsections for different purposes.
10  */
11 #define PERCPU_INPUT(cacheline)
    \
12         __per_cpu_start = .;
    \
13         *(.data..percpu..first)
    \
14         . = ALIGN(PAGE_SIZE);
    \
15         *(.data..percpu..page_aligned)
    \
```

```
        . = ALIGN(cacheline);
\
        *(.data..percpu..read_mostly)
\
        . = ALIGN(cacheline);
\
        *(.data..percpu)
\
        *(.data..percpu..shared_aligned)
\
        PERCPU_DECRYPTED_SECTION
\
        __per_cpu_end = .;
```

Define the input section for the per-cpu area.

---

per-cpu data allocation API

Static per-cpu data allocation

Learn which APIs to declare to allocate and use per-cpu data at compile time.

DEFINE_PER_CPU()

include/linux/percpu-defs.h

```
#define DEFINE_PER_CPU(type, name)                              \
        DEFINE_PER_CPU_SECTION(type, name, "")
```

per_cpu declare the data to be used. The per-CPU data declared in the kernel core uses the static area of the first chunk.

- You can also use arrays for types.
  - e.g. DEFINE_PER_CPU(int[3], my_percpu_array);

DEFINE_PER_CPU_SECTION()

include/linux/percpu-defs.h

```
#define DEFINE_PER_CPU_SECTION(type, name, sec)
\
        __PCPU_ATTRS(sec) __typeof__(type) name
```

__PCPU_ATTRS()

include/linux/percpu-defs.h

```
/*
 * Base implementations of per-CPU variable declarations and definition
s, where
 * the section in which the variable is to be placed is provided by the
 * 'sec' argument.  This may be used to affect the parameters governing
the
 * variable's storage.
```

```
 *
 * NOTE!  The sections for the DECLARE and for the DEFINE must match, le
st
 * linkage errors occur due the compiler generating the wrong code to ac
cess
 * that section.
 */
#define __PCPU_ATTRS(sec)
\
        __percpu __attribute__((section(PER_CPU_BASE_SECTION sec)))
\
        PER_CPU_ATTRIBUTES
```

Specifies the section location where the per-cpu data will be stored.

```
#ifndef PER_CPU_BASE_SECTION
#ifdef CONFIG_SMP
#define PER_CPU_BASE_SECTION ".data..percpu"
#else
#define PER_CPU_BASE_SECTION ".data"
#endif
#endif
```

This is the section that is stored in the per-cpu data declaration.

- On SMP machines, .data.. percpu section.
- On UP machines, per-cpu data is stored in the .data section because it is no different from normal data.

Declaring a static per-cpu data extern

To reference static per-cpu data from other files, use the following API:

include/linux/percpu-defs.h”

```
#define DECLARE_PER_CPU(type, name)
\
        DECLARE_PER_CPU_SECTION(type, name, "")
```

include/linux/percpu-defs.h”

```
#define DECLARE_PER_CPU_SECTION(type, name, sec)
\
        extern __PCPU_ATTRS(sec) __typeof__(type) name
```

Create/Disable Dynamic

Learn about the API to dynamically allocate and deallocate per-CPU data at runtime.

- alloc_percpu()
- free_percpu()

alloc_percpu()

include/linux/percpu.h

```
#define alloc_percpu(type)
\
        (typeof(type) __percpu *)__alloc_percpu(sizeof(type),
\
                                                __alignof__(type))
```

Allocates a per-CPU data area equal to the specified type size. At first, it uses the dynamic of the first chunk to allocate it, and if the dynamic area of the first chunk is insufficient, it creates and uses additional chunks. The added chunk is entirely a dynamic area.

__alloc_percpu()

mm/percpu.c

```
/**
 * __alloc_percpu - allocate dynamic percpu area
 * @size: size of area to allocate in bytes
 * @align: alignment of area (max PAGE_SIZE)
 *
 * Equivalent to __alloc_percpu_gfp(size, align, %GFP_KERNEL).
 */
```

```
void __percpu *__alloc_percpu(size_t size, size_t align)
{
        return pcpu_alloc(size, align, false, GFP_KERNEL);
}
EXPORT_SYMBOL_GPL(__alloc_percpu);
```

@align to the requested @size and allocate a per-cpu data area.

Using Per-CPU Data in Modules

The way a module uses per-CPU is handled slightly differently in kernel internals.

- Dynamic Assignment
  - The method of allocation and management in the existing dynamic area is the same.
- static assignment
  - It is not assigned to the static area of the first chunk, but dynamically manages the allocation to the reserve area.

---

per-cpu data access API

It is designed with preemption in mind, and since per-cpu data does not point to the address of the actual per-cpu data, it must be accessed through the appropriate API without directly accessing and using it.

1) L-value operation

Use the get_cpu_var() function to increment the per-cpu data, as shown in the following example. After use, you must call the put_cpu_var() function.

- get_cpu_var(sockets_in_use)++;
- put_cpu_var(sockets_in_use);

get_cpu_var()

include/linux/percpu-defs.h

```
1 /*
2  * Must be an lvalue. Since @var must be a simple identifier,
3  * we force a syntax error here if it isn't.
4  */
5 #define get_cpu_var(var)
6 (*({
  \
7         preempt_disable();
  \
8         this_cpu_ptr(&var);
  \
9 }))
```

- In SMP machines, when you want to get a value, you preempt_disable it.
- The argument must be l-value.

put_cpu_var()

include/linux/percpu-defs.h

```
1 /*
2  * The weird & is necessary because sparse considers (void)(var) to be
3  * a direct dereference of percpu variable (var).
4  */
5 #define put_cpu_var(var)
  \
6 do {
  \
7         (void)&(var);
  \
8         preempt_enable();
  \
9 } while (0)
```

- Write the value on the SMP machine and then preempt_enable().

2) Pointer Manipulation

If you need to access it with a pointer, as shown in the following example, you first get the current CPU ID, then use the per_cpu_ptr() function to get the pointer and then use the pointer to perform the desired user action. When the per-cpu data is done, put_cpu() must be used.

```
int cpu;

cpu = get_cpu(  )
ptr = per_cpu_ptr(per_cpu_var, cpu);
/* work with ptr */
put_cpu(  );
```

per_cpu_ptr()

include/linux/percpu-defs.h

```
#define per_cpu_ptr(ptr, cpu)
\
({
\
        __verify_pcpu_ptr(ptr);
\
        SHIFT_PERCPU_PTR((ptr), per_cpu_offset((cpu)));
\
})
```

When using the Sparse static code analysis tool, check if there is any problem with the ptr (pointer) through __verify_pcpu_ptr(). It then returns an address that combines the ptr with the offset of the CPU.

per_cpu_offset()

include/asm-generic/percpu.h

```
/*
 * per_cpu_offset() is the offset that has to be added to a
 * percpu variable to get to the instance for a certain processor.
 *
 * Most arches use the __per_cpu_offset array for those offsets but
 * some arches have their own ways of determining the offset (x86_64, s3
90).
 */
#ifndef __per_cpu_offset
extern unsigned long __per_cpu_offset[NR_CPUS];

#define per_cpu_offset(x) (__per_cpu_offset[x])
#endif
```

Returns the per-cpu offset value for the requested CPU @x number.

- __per_cpu_offset[]
  - To access per-CPU data, you need to add a specified offset value for each CPU number, which is stored in an array of global variables.

this_cpu_ptr()

include/linux/percpu-defs.h

```
#ifdef CONFIG_DEBUG_PREEMPT
#define this_cpu_ptr(ptr)
\
({
\
```

```
4         __verify_pcpu_ptr(ptr);
\
5         SHIFT_PERCPU_PTR(ptr, my_cpu_offset);
\
6 })
7 #else
8 #define this_cpu_ptr(ptr) raw_cpu_ptr(ptr)
9 #endif
```

Returns a pointer with the per-cpu offset value applied to the current CPU.

- Call the raw_cpu_ptr() macro action to return the PTR plus the current CPU's offset.

__verify_pcpu_ptr()

include/linux/percpu-defs.h

```
01 /*
02  * __verify_pcpu_ptr() verifies @ptr is a percpu pointer without evaluat
   ing
03  * @ptr and is invoked once before a percpu area is accessed by all
04  * accessors and operations.  This is performed in the generic part of
05  * percpu and arch overrides don't need to worry about it; however, if a
   n
06  * arch wants to implement an arch-specific percpu accessor or operatio
   n,
07  * it may use __verify_pcpu_ptr() to verify the parameters.
08  *
09  * + 0 is required in order to convert the pointer type from a
10  * potential array type to a pointer to a single item of the array.
11  */
12 #define __verify_pcpu_ptr(ptr)
   \
13 do {
   \
14         const void __percpu *__vpp_verify = (typeof((ptr) + 0))NULL;
   \
15         (void)__vpp_verify;
   \
16 } while (0)
```

This macro is for the Sparse static code analysis tool and checks if the ptr (pointer) is used as a pointer to the per-cpu address space.

SHIFT_PERCPU_PTR()

include/linux/percpu-defs.h

```
1 /*
2  * Add an offset to a pointer but keep the pointer as-is.  Use RELOC_HID
   E()
3  * to prevent the compiler from making incorrect assumptions about the
4  * pointer value.  The weird cast keeps both GCC and sparse happy.
5  */
6 #define SHIFT_PERCPU_PTR(__p, __offset)
   \
7         RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset))
```

It simply adds a __offset to the __p and is force-cast to the kernel address space and returned.

RELOC_HIDE()

include/linux/compiler-gcc.h

```
/*
 * This macro obfuscates arithmetic on a variable address so that gcc
 * shouldn't recognize the original var, and make assumptions about it.
 *
 * This is needed because the C standard makes it undefined to do
 * pointer arithmetic on "objects" outside their boundaries and the
 * gcc optimizers assume this is the case. In particular they
 * assume such arithmetic does not wrap.
 *
 * A miscompilation has been observed because of this on PPC.
 * To work around it we hide the relationship of the pointer and the object
 * using this macro.
 *
 * Versions of the ppc64 compiler before 4.1 had a bug where use of
 * RELOC_HIDE could trash r30. The bug can be worked around by changing
 * the inline assembly constraint from =g to =r, in this particular
 * case either is valid.
 */
#define RELOC_HIDE(ptr, off)                                    \
  ({ unsigned long __ptr;                                       \
    __asm__ ("" : "=r"(__ptr) : "0"(ptr));              \
    (typeof(ptr)) (__ptr + (off)); })
```

- Returns the address of the __ptr plus off.
- A literal translation of the above comment is that this macro obscures the arithmetic operation of a variable, so that gcc cannot recognize the original value and therefore cannot make assumptions about it, in other words, it takes only the address of the variable, regardless of the type of the variable, adds an offset to it, and casts it to the type of the original variable.
- Compilers prior to GCC 4.1 had a bug with the PPC64 architecture, which was resolved by applying this walkaround.

raw_cpu_ptr()

include/linux/percpu-defs.h

```
#define raw_cpu_ptr(ptr)
\
({
\
        __verify_pcpu_ptr(ptr);
\
        arch_raw_cpu_ptr(ptr);
\
})
```

Returns the pointer address of the @ptr plus the per-cpu offset for the current CPU.

- For Sparse static code analysis, check the ptr (pointer) via __verify_pcpu_ptr(). Then call the arch_raw_cpu_ptre() macro action to return the PTR plus the current CPU's Per-CPU offset.

arch_raw_cpu_ptr()

include/asm-generic/percpu.h”

```
/*
 * Arch may define arch_raw_cpu_ptr() to provide more efficient address
 * translations for raw_cpu_ptr().
 */
#ifndef arch_raw_cpu_ptr
#define arch_raw_cpu_ptr(ptr) SHIFT_PERCPU_PTR(ptr, __my_cpu_offset)
#endif
```

Returns the pointer address of the @ptr plus the per-cpu offset for the current CPU.

__my_cpu_offset() – ARM32

arch/arm/include/asm/percpu.h

```
#define __my_cpu_offset __my_cpu_offset()

static inline unsigned long __my_cpu_offset(void)
{
        unsigned long off;

        /*
         * Read TPIDRPRW.
         * We want to allow caching the value, so avoid using volatile and
         * instead use a fake stack read to hazard against barrier().
         */
        asm("mrc p15, 0, %0, c13, c0, 4" : "=r" (off)
                : "Q" (*(const unsigned long *)current_stack_pointer));

        return off;
}
```

Read the value of the offset corresponding to the cpu stored in the TPIDRPRW register.

- This value is used to access the per-cpu data in addition to the per-cpu ptr address.
- In the ARM architecture, the TPIDRPRW register, which is not used among the CP15 registers, is used for CP<> offset archiving per CPU, which increases per-cpu computational performance.

__my_cpu_offset() – ARM64

arch/arm64/include/asm/percpu.h

```
#define __my_cpu_offset __my_cpu_offset()

static inline unsigned long __my_cpu_offset(void)
{
        unsigned long off;

        /*
         * We want to allow caching the value, so avoid using volatile and
         * instead use a fake stack read to hazard against barrier().
         */
        asm(ALTERNATIVE("mrs %0, tpidr_el1",
                        "mrs %0, tpidr_el2",
                        ARM64_HAS_VIRT_HOST_EXTN)
                : "=r" (off) :
                "Q" (*(const unsigned long *)current_stack_pointer));

        return off;
}
```

Reads the per-cpu offset value corresponding to the CPU stored in the TPIDR register.

Other key functions

__percpu Attribute Macros

include/linux/compiler_types.h

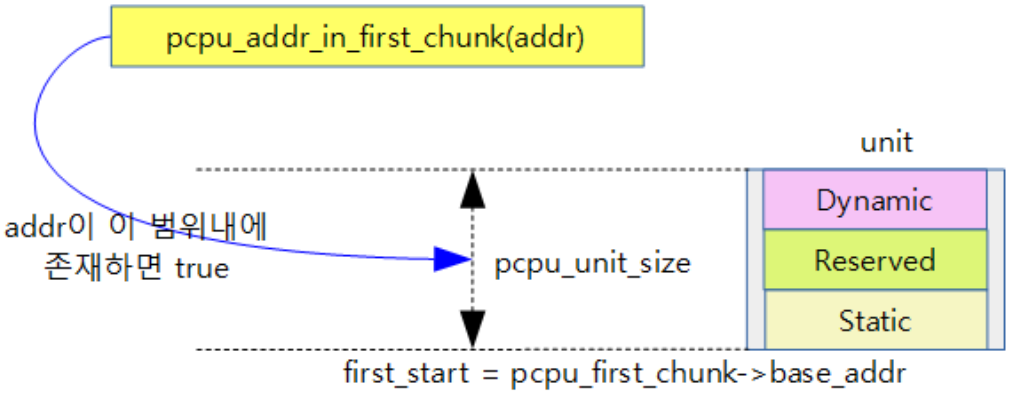
```
1 # define __percpu        __attribute__((noderef, address_space(3)))
```

Sparse is an attribute specified by the static code analysis tool.

- address_space(3):
  - Specify the per-CPU address_space to be used.
- noderef
  - Stipulate that dereferencing pointers cannot be used.
    - You can't use a generic pointer variable, *ptr, etc., but you can use the &val form that directly provides the address of the variable.

pcpu_addr_in_first_chunk()

mm/percpu.c – Removed as of kernel v4.14

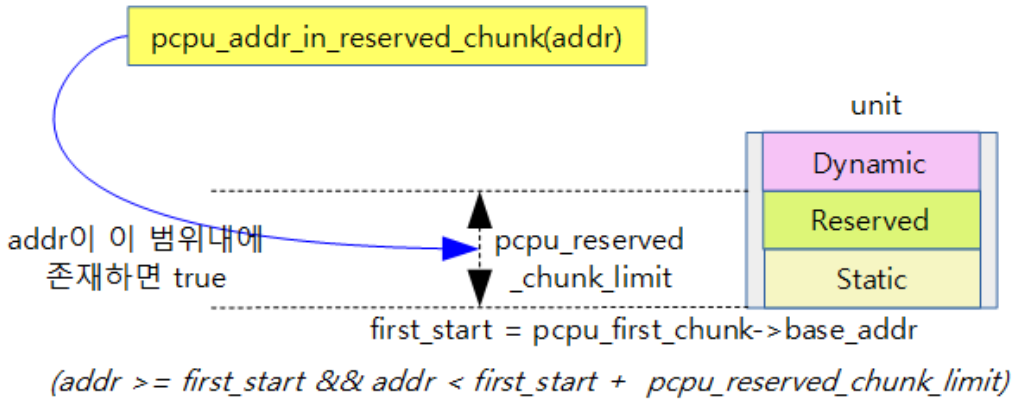


(http://jake.dothome.co.kr/wp-content/uploads/2016/02/percpu-9.png)

- Returns whether the given address is in the first chunk range.

pcpu_addr_in_reserved_chunk()

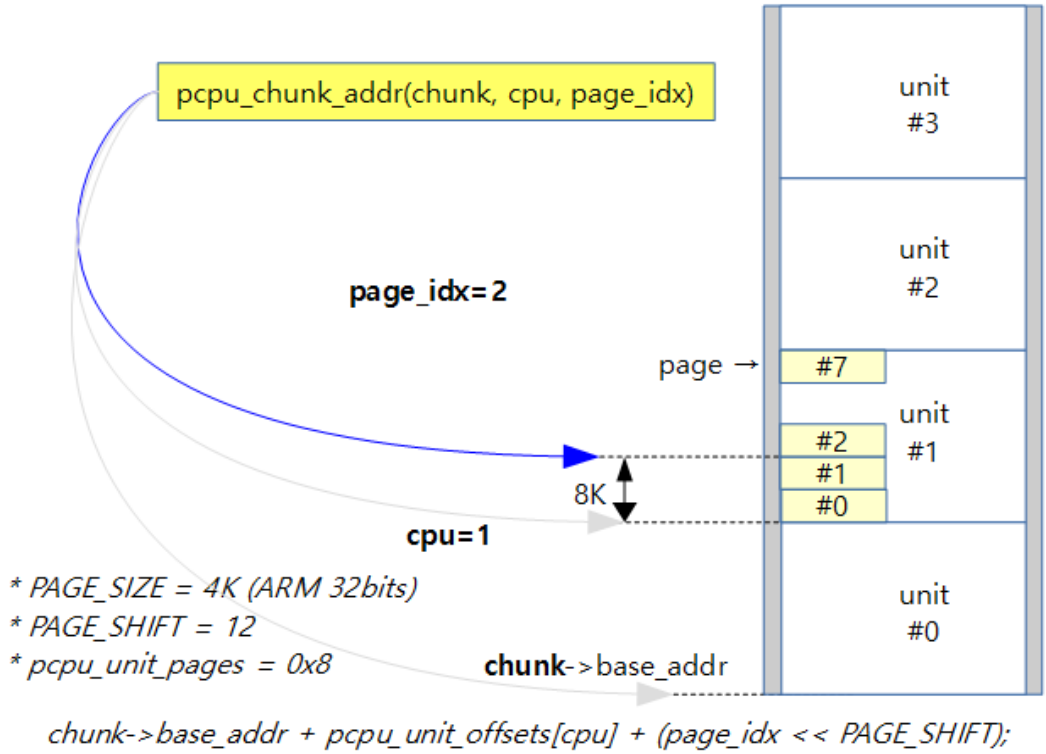
mm/percpu.c – Removed as of kernel v4.14



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

- Returns whether a given address is included in the Static and Reserved zones.
- If no reserved chunk is given, i.e. there is no reserved zone, the pcpu_reserved_chunk_limit value is 0.

pcpu_chunk_addr()



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

- The given chunk number, CPU number, and page number are used to get the start address of the page.

Example of using per-cpu data used to count slub objects

include/linux/slub_def.h

```
1 /*
2  * Slab cache management.
```

```
 */
struct kmem_cache {
        struct kmem_cache_cpu __percpu *cpu_slab;
```

```
struct kmem_cache_cpu {
        void **freelist;        /* Pointer to next available object */
        unsigned long tid;      /* Globally unique transaction id */
        struct page *page;      /* The slab from which we are allocating
*/
        struct page *partial;   /* Partially allocated frozen slabs */
#ifdef CONFIG_SLUB_STATS
        unsigned stat[NR_SLUB_STAT_ITEMS];
#endif
};
```

```
static inline void flush_slab(struct kmem_cache *s, struct kmem_cache_cp
u *c)
{
        stat(s, CPUSLAB_FLUSH);
        deactivate_slab(s, c->page, c->freelist);

        c->tid = next_tid(c->tid);
        c->page = NULL;
        c->freelist = NULL;
}
```

```
static inline void stat(const struct kmem_cache *s, enum stat_item si)
{
#ifdef CONFIG_SLUB_STATS
        /*
         * The rmw is racy on a preemptible kernel but this is acceptabl
e, so
         * avoid this_cpu_add()'s irq-disable overhead.
         */
        raw_cpu_inc(s->cpu_slab->stat[si]);
#endif
}
```

include/linux/slub_def.h

```
enum stat_item {
        ALLOC_FASTPATH,         /* Allocation from cpu slab */
        ALLOC_SLOWPATH,         /* Allocation by getting a new cpu slab
*/
        FREE_FASTPATH,          /* Free to cpu slab */
        FREE_SLOWPATH,          /* Freeing not to cpu slab */
        FREE_FROZEN,            /* Freeing to frozen slab */
        FREE_ADD_PARTIAL,       /* Freeing moves slab to partial list */
        FREE_REMOVE_PARTIAL,    /* Freeing removes last object */
        ALLOC_FROM_PARTIAL,     /* Cpu slab acquired from node partial l
ist */
        ALLOC_SLAB,             /* Cpu slab acquired from page allocator
*/
        ALLOC_REFILL,           /* Refill cpu slab from slab freelist */
        ALLOC_NODE_MISMATCH,    /* Switching cpu slab */
        FREE_SLAB,              /* Slab freed to the page allocator */
        CPUSLAB_FLUSH,          /* Abandoning of the cpu slab */
        (..생략..)
```

include/linux/percpu-defs.h

```
#define raw_cpu_inc(pcp)                raw_cpu_add(pcp, 1)
```

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

```
01 #define __pcpu_size_call(stem, variable, ...)
   \
02 do {
   \
03         __verify_pcpu_ptr(&(variable));
   \
04         switch(sizeof(variable)) {
   \
05                 case 1: stem##1(variable, __VA_ARGS__);break;
   \
06                 case 2: stem##2(variable, __VA_ARGS__);break;
   \
07                 case 4: stem##4(variable, __VA_ARGS__);break;
   \
08                 case 8: stem##8(variable, __VA_ARGS__);break;
   \
09                 default:
   \
10                         __bad_size_call_parameter();break;
   \
11         }
   \
12 } while (0)
```

include/asm-generic/percpu.h

```
1 #ifndef raw_cpu_add_4
2 #define raw_cpu_add_4(pcp, val)         raw_cpu_generic_to_op(pcp, val,
  +=)
3 #endif
```

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

per-cpu-related structs

pcpu_chunk Structure

mm/percpu-internal.h

```
01 struct pcpu_chunk {
02 #ifdef CONFIG_PERCPU_STATS
03         int                     nr_alloc;       /* # of allocations */
04         size_t                  max_alloc_size; /* largest allocation si
   ze */
05 #endif
06
07         struct list_head        list;           /* linked to pcpu_slot l
   ists */
08         int                     free_bytes;     /* free bytes in the chu
   nk */
09         int                     contig_bits;    /* max contiguous size h
   int */
```

```
10          int                 contig_bits_start; /* contig_bits starti
    ng
11                                                 offset */
12          void                *base_addr;     /* base address of this
    chunk */
13
14          unsigned long       *alloc_map;     /* allocation map */
15          unsigned long       *bound_map;     /* boundary map */
16          struct pcpu_block_md    *md_blocks;     /* metadata blocks */
17
18          void                *data;          /* chunk data */
19          int                 first_bit;      /* no free below this */
20          bool                immutable;      /* no [de]population all
    owed */
21          int                 start_offset;   /* the overlap with the
    previous
22                                                 region to have a page
    aligned
23                                                 base_addr */
24          int                 end_offset;     /* additional area requi
    red to
25                                                 have the region end p
    age
26                                                 aligned */
27
28          int                 nr_pages;       /* # of pages served by
    this chunk */
29          int                 nr_populated;   /* # of populated pages
    */
30          int                 nr_empty_pop_pages; /* # of empty popula
    ted pages */
31          unsigned long       populated[];    /* populated bitmap */
32  };
```

- nr_alloc
  - Number of allocations requested in this chunk
- max_alloc_size
  - Each time an allocation request is made to this chunk, the maximum allocation size of the requested allocation size is updated.
- list
  - This is a list of pcpu_slot maps used to manage chunks.
  - All chunks are linked to a list of pcpu_slot maps and used to manage dynamic spaces, sorted by the size of the free space in each chunk, with the allocation starting with the chunk with the fewest free space.
- free_bytes
  - It is the sum of the free space in the chunk. A unit is a byte.
  - Each allocation and release changes the free_size and moves from the chunk slot to the other appropriate slot according to the free_size.
- contig_bits (formerly called contig_hints, and the size was in bytes.)
  - It is the largest contiguous free space size within a chunk. (4 bytes per bit)
- contig_bits_start
  - This is the offset bit from which the contig_bits begins.
- *alloc_map
  - A bitmap that expresses an assignment, with each bit representing whether or not it is allocated in units of 4 bytes.
- *bound_map

    - A bitmap that represents a boundary, with each bit managed in 4-byte increments, representing the starting bit of the allocated data as 1.
    - The last bit of the bound_map must also be represented by 1, so it is always 1 bit larger than the size of the alloc_map.
- *md_blocks
    - It refers to metadata that is managed in per-CPU blocks.
    - Within the chunk, this meter data is used to quickly scan each per-CPU block for allocation.
- data
    - The address of the page structure pointer array is stored to which the chunk is assigned.
- immutable
    - Indicates whether the chunk in question can be changed.
- base_addr
    - It is the lowest virtual starting address in the area where the chunk is assigned, and uses an address that is aligned on a page-by-page basis.
    - Since it is aligned on a page-by-page basis, the number of offset addresses that are aligned is contained in the start_offset below.
        - Example: If you are using 4K page units, the start address is 0x1_2340, and the size is 0x1000.
            - base_addr=0x1_2000, start_offset=0x340, end_offset=0xcb0
- start_offset
    - This is the gap between the start address of the area to be managed by the chunk and the start address sorted down on a page-by-page basis.
- end_offset
    - This is the gap between the end address of the area to be managed by the chunk and the end address sorted up on a page-by-page basis.
- nr_pages
    - A chunk is the number of pages that can be used.
- nr_populated
    - Within the chunk, the actual physics pages have been allocated and mapped, and the number of pages currently in use.
    - If you allocated a new chunk, this value is 0 because the actual page is not mapped.
- nr_empty_pop_pages
    - The number of pages within the chunk where the actual physical pages have been allocated and mapped, but are currently unused.
    - If you allocated a new chunk, this value is 0 because the actual page is not mapped.
- populated[]
    - This arrangement is for managing the areas within the chunk where page assignments are made and which are not.
    - The first chunk uses memory that the actual page has been allocated and mapped to allocated, so all pages are set to 1.
    - The added chunk will have all pages set to 0 until the actual area is allocated and used.
        - The vmalloc space is managed as in use, but no physical pages are allocated and no maps are made.

Members used in the legacy Areamap method

- contig_hints (X)
  - It is the largest contiguous space size within a chunk.
  - In the case of the first chunk, the contig_hint is the same size as the free_size.
  - When an allocation is required, sizes larger than this value are not possible in this chunk.
- map_used (X)
  - Indicates the number of items in the map[] array within the chunk.
  - The entry of the number +1 contains the last address offset+1 (in-use flag).
    - For example, if map_used=4, use 4+1 entries.
- map_alloc (X)
  - The maximum number of items in the map array allocated within a chunk. (The initial default value starts at 128 and expands if insufficient.)
- map[] (X)
  - The map[] array is a mutable array that expands dynamically at runtime.
    - As the map array size expands, the number of map_alloc increases accordingly.
    - While the Early Memory Allocator is running, PERCPU_DYNAMIC_EARLY_SLOTS (128) arrays are initially declared in the map array.
    - After the Slub Memory Allocator is running, the map can grow larger if it needs more.
  - Distinguish between map array values
    - How to distinguish old array values (legacy method)
      - A positive number is the size of the empty space within a unit.
      - A negative number is the size of the space being used.
    - The new array value separation method was introduced in kernel v2014.3-rc3 in March 15.
      - Note: percpu: store offsets instead of lengths in ->map[] (https://github.com/torvalds/linux/commit/723ad1d90b5663ab623bb3bfba3e4ee7101795d7)
      - Each byte represents a size, and a value aligned with sizeof(int) is used. However, it is used for the last other purpose of the size, so it must be cut and used.
        - The last bit separates the allocation status:
          - 1=in-use status
          - 0=free status
      - e.g. 4 bytes free, 8 in use, 4 in use, 4 free, 12 in use, 100 free, total unit size=132, map_used=3
        - Old method: map[]= {4, -8, -4, 4, -12, 100, 0, }
        - new method: map[]= {0, 5, 13, 16, 21, 32, 133, 0, }
          - <0,0>, <4,1>, <12,1>, <16,0>, <20,1>, <32,0>, <132,1> and <offset, in-use flag> and 1 byte of offset is used as the in-use flag in actual storage.
          - 0 bytes free from 3~4, 4 in use from 11~8, 12 in use from 15~4, 16 in use from 19~4, 20 in use from 31~32, 32 free from 131~100

pcpu_block_md Struct

mm/percpu-internal.h

```
/*
 * pcpu_block_md is the metadata block struct.
 * Each chunk's bitmap is split into a number of full blocks.
 * All units are in terms of bits.
 */
```

```
struct pcpu_block_md {
        int                     contig_hint;    /* contig hint for block */
        int                     contig_hint_start; /* block relative starting
                                                      position of the contig hint */
        int                     left_free;      /* size of free space along
                                                   the left side of the block */
        int                     right_free;     /* size of free space along
                                                   the right side of the block */
        int                     first_free;     /* block position of first free */
};
```

This is the metadata that is managed by the bitmap in the chunk in per-CPU blocks (the bitmap representation of 1 page in 4-byte increments).

- contig_hint
  - It contains the number of bits for the maximum free area within the block.
    - e.g. contig_hint=16 means that there is a maximum free area of 16 x 4 bytes = 64 bytes in this block.
- conting_hint_start
  - The bit position where the maximum free area starts within the block
- left_free
  - If the beginning of this block is the free area, it contains the number of bits in the free area.
- right_free
  - If the end of this block is in the free region, it contains the number of bits in the free region.
- first_free
  - It contains an offset bit that points to the first free area of this block.
  - Note: Unlike left_free and right_free, it contains positions, not counts.

pcpu_group_info Structure

include/linux/percpu.h

```
struct pcpu_group_info {
        int                     nr_units;       /* aligned # of units */
        unsigned long           base_offset;    /* base address offset */
        unsigned int            *cpu_map;       /* unit->cpu map, empty
                                                 * entries contain NR_CPUS */
};
```

The nodes in the NUMA system are managed as groups, and each group manages the CPUs that belong to that node. The UMA system is a single node, so it uses one group.

- nr_units
  - The number of units used by that group.
- base_offset
  - The base_offset address used by the group
- *cpu_map[]
  - Mapping the unit->cpu used by that group

pcpu_alloc_info Struct

include/linux/percpu.h

```
01 struct pcpu_alloc_info {
02         size_t                  static_size;
03         size_t                  reserved_size;
04         size_t                  dyn_size;
05         size_t                  unit_size;
06         size_t                  atom_size;
07         size_t                  alloc_size;
08         size_t                  __ai_size;      /* internal, don't use
   */
09         int                     nr_groups;      /* 0 if grouping unneces
   sary */
10         struct pcpu_group_info  groups[];
11 };
```

This struct is stored and used as a local variable within the function during the initial initialization of the per-cpu data in the setup_per_cpu_areas() function. Since the information disappears after initialization, you need to use global variables. (e.g. when doing unit->cpu mapping, etc...)

- static_size
  - Static area size within 1 unit
  - RPI2: Yes) 0x3ec0
- reserved_size
  - The reserved area size within one unit can be used when CONFIG_MODULE kernel options are used.
  - rpi2: 8K.
- dyn_size
  - The size of the dyn_size area within one unit varies from one architecture to another.
  - ARM also has a slightly larger value than before, so it basically asks for 32K for 20bit and 64K for 28bit, but if you align 4K with static_size + reserved_size + dyn_size, there will be extra space, which is added to the dyn_size.
  - rpi2: 0x5140 (initial request value 32x0 to 5000K)
- atom_size
  - Minimum Assigned Size
  - ARM 4K
- alloc_size
  - Size to assign

    - rpi2: 0xb000 x 4 = 0x3_c000
- nr_groups
    - Number of groups (nodes)
    - rpi2: 1

Key Global Variables

- pcpu_unit_pages
    - Number of pages in 1 unit
        - RPI2: e.g. 0xb (1 unit page=11 pages)
- pcpu_unit_size
    - 1 Unit Size
        - rpi2: e.g. 0xb000 (1 unit size=44K)
- pcpu_nr_units
    - Total number of units
        - RPI2: 4 pcs
- pcpu_atom_size
    - As the minimum page size to use for allocation, ARM and ARM64 use PAGE_SIZE.
    - x86-64-bit systems and a few architectures use huge pages such as 2M pages.
- pcpu_nr_slots
    - Number of slots for management
    - Change it to the slot number that corresponds to the unit size and add 2
        - rpi2: If the unit size is 44K, the slot number is 13, so adding 2 is pcpu_nr_slots=15
- pcpu_chunk_struct_size
    - The value set when the first chunk is created is the chunk structure size + the number of bytes of the populated bitmap (determined by the unit size).
- pcpu_low_unit_cpu
    - The CPU number where base addr is located at the bottom
    - rpi2: 0
- pcpu_high_unit_cpu
    - The CPU number where the base_addr is located at the top
    - rpi2: 3
- pcpu_base_addr
    - The virtual start address to which the first chunk is assigned
- pcpu_unit_map[]
    - CPU -> Unit Mapping
    - rpi2: { 0, 1, 2, 3 }
- pcpu_unit_offsets[]
    - An array that stores the start offset of each unit
        - rpi2: { 0, 0xb000, 0x1_6000, 0x2_1000 }
- pcpu_nr_groups
    - Number of groups (number of NUMA nodes)
        - rpi2: 1

- pcpu_group_offsets[]
  - An array that stores the starting offset of each group.
    - rpi2: 0
- pcpu_group_sizes[]
  - An array that stores the size of each group.
    - rpi2: { 0x2_c000 }
- pcpu_first_chunk
  - pcpu_chunk struct pointer to manage dynamic area maps in the First Chunk
- pcpu_reserved_chunk
  - pcpu_chunk struct pointer to manage reserved area maps in the First Chunk
  - null when not using a module
- pcpu_reserved_chunk_limit
  - If there is a reserved zone, the static size plus the reserved_size
- pcpu_slot[]
  - pcpu_nr_slots controls the array size, and each array contains a list_head that points to a chunk.
  - The last slot is for empty chunks.
  - This arrangement of slots manages chunks that use dynamic areas.
  - rpi2: static_size=0x3ec0, reserved_size=0x2000, dyn_size=0x5140 example)
    - The free_size of the first chunk = 0x5140, and the slot is slot 12.
    - The reserved chunk has free_size=0x5ec0, but the reserved chunk is allocated when the module is loaded, so it is not managed by the pcpu_slot[] that manages the dynamic region.
    - Thus, when the per-CPU data is initialized for the first time, pcpu_slot[12] is configured with one first chunk as a list, and a spare empty chunk is added to the last slot.
- pcpu_nr_empty_pop_pages
  - Indicates the number of empty populated pages in the added chunks that manage only the dynamic area.
  - In this case, the reserved chunk, which manages the reserve area of the first chunk, of course, does not have an effect.
- pcpu_async_enabled
  - If you allow async for free in the per-cpu area, the pcpu_balance_workfn() function will be scheduled and late.
  - It is enabled when the kernel is initialized and initcall is called.
- pcpu_atomic_alloc_failed
  - If atomic allocation fails, it will be changed to true.

consultation

- Per-cpu -1- (Basic) | Question C – Current Article
- Per-cpu -2-(initialize) (http://jake.dothome.co.kr/setup_per_cpu_areas) | Qc
- 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) | Qc

- Per-CPU Variables (http://www.makelinux.net/ldd3/chp-8-sect-5)
- robust per_cpu allocation for modules (https://lwn.net/Articles/180101/) | LWN.net
- Per-CPU Memory Management (1) (http://egloos.zum.com/studyfoss/v/5375570) | F/OSS
- Per-CPU Memory Management (2) (http://studyfoss.egloos.com/5377666) | F/OSS
- Documentation/preempt-locking.txt (https://www.kernel.org/doc/Documentation/preempt-locking.txt) | kernel.org
- Better per-CPU variables (https://lwn.net/Articles/258238/) | LWN.net
- What every programmer should know about memory, Part 1 (http://lwn.net/Articles/250967/) | LWN.net
- ARM: implement optimized percpu variable access (http://lwn.net/Articles/527927/) | LWN.net

9 thoughts to “Per-cpu -1- (Basic)”

IPARAN (HTTPS://PARANLEE.GITHUB.IO/)
2021-04-13 11:48 (http://jake.dothome.co.kr/per-cpu/#comment-305107)

Hello. Thank you for always keeping an eye on me~

I was watching it again today during the review.

CPU -> Unit Mapping

Per-CPU data uses NR_CPUS arrays, which
can waste memory on <>.
This is because a system that is configured to support thousands of CPUs (NR_CPUS) sometimes has only a few CPUs that are capable of running at actual boot.

I don't know if I'm misunderstanding the part I <> in, but I think the meaning is different from the line below.

"If the number of CPUs that the system will use after booting is less than NR_CPUS"

Is this what I understand correct?

RESPONSE (/PER-CPU/?REPLYTOCOM=305107#RESPOND)

IPARAN (HTTPS://PARANLEE.GITHUB.IO/)
2021-04-13 11:50 (http://jake.dothome.co.kr/per-cpu/#comment-305108)

[[If you are using a number greater than the number of CPUs that actually exist]]

[[...]] I don't know if I'm misunderstanding the part I put in, or if it's different in meaning from the bottom line.

"If the number of CPUs that the system will use after booting is less than NR_CPUS"

RESPONSE (/PER-CPU/?REPLYTOCOM=305108#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-04-13 13:40 (http://jake.dothome.co.kr/per-cpu/#comment-305109)

Hello?

To explain it a little bit, they mean the same thing:

Since
per-CPU data uses an array of NR_CPUS numbers, 1) a number greater than the number of CPUs that actually exist > if the number of NR_CPUS is greater than the number of CPUs that actually exist
2) A number that is greater than the number of CPUs that actually exist > the NR_CPUS used by the built kernel is a large number, but the booted system has fewer
CPUs 3) A number that is greater than the number of CPUs that actually exist > The number of CPUs that the system will use after booting is less than NR_CPUS (Iparan)

Note that CONFIG_NR_CPUS kernel options have changed a few times in the case of arm64 defaults, and have been increased from 5 to 1 since kernel v64.256. Since then, it hasn't changed until the latest v5.12.

I appreciate it.

RESPONSE (/PER-CPU/?REPLYTOCOM=305109#RESPOND)

이파란 (HTTPS://PARANLEE.GITHUB.IO/)
2021-04-13 18:00 (http://jake.dothome.co.kr/per-cpu/#comment-305110)

질문으로 더 깊은 지식을 얻어가네요! 항상 감사드립니다~

응답 (/PER-CPU/?REPLYTOCOM=305110#RESPOND)

이파란 (HTTPS://PARANLEE.GITHUB.IO/)
2021-04-13 18:13 (http://jake.dothome.co.kr/per-cpu/#comment-305111)

댓글에 추가적으로 말씀해주신 내용에서

CONFIG_NR_CPUS 관련 호기심이 생겨서 질문하나 추가로 드려도 될까요?

제가 만약에 CPU 코어가 4000개 짜리 프로세서가 있으면,

직접 수정해서 NR_CPUS 값을 4000개로 만들어주어야 per-cpu 데이터를 효과적으로 쓸 수 있나요?

응답 (/PER-CPU/?REPLYTOCOM=305111#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2021-04-15 08:42 (http://jake.dothome.co.kr/per-cpu/#comment-305126)

CONFIG_NR_CPUS 값은 빌드할 커널이 지원하는 최대 수의 cpu를 의미합니다.
디폴트 값이 4096일 때 4000개의 cpu를 가진 경우 수정할 필요 없습니다.
만일 8000개의 cpu를 가진 경우 8000개 이상으로 수정해야 시스템이 동작할 수 있습니다.

물론 시스템의 cpu 수와 동일한 CONFIG_NR_CPUS 값을 사용하면 메모리 낭비가 없어 좋긴 하지만,
하나의 커널 이미지로 여러 시스템에서 돌릴 수 있게 하기 위해 조금 큰 수를 사용합니다.

감사합니다.

응답 (/PER-CPU/?REPLYTOCOM=305126#RESPOND)

이파란 (HTTPS://PARANLEE.GITHUB.IO/)
2021-04-15 09:11 (http://jake.dothome.co.kr/per-cpu/#comment-305128)

네 초보적인 질문인데도 잘 설명해주셔서 감사합니다!
좋은 하루되세요~

응답 (/PER-CPU/?REPLYTOCOM=305128#RESPOND)

이파란 (HTTPS://PARANLEE.GITHUB.IO/)
2021-04-14 17:30 (http://jake.dothome.co.kr/per-cpu/#comment-305123)

항상 감사드립니다.

다음 그림은 가장 낮은 주소의 unit으로 부터 각각의 unit의 시작 offset이 지정되는 것을 보여준다.
pcpu_unit_offset[] 배열의 초기화는 pcpu_setup_first_chunk() 함수를 참고한다.

percpu-3a.png 그림에서

예) node #0 -> 2 cpu
[[2 cpu]] -> [[4 cpu]] 인가요?

응답 (/PER-CPU/?REPLYTOCOM=305123#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2021-04-15 08:46 (http://jake.dothome.co.kr/per-cpu/#comment-305127)

지적해주신 것 처럼 첫 번째 노드의 cpu 수는 2가 아니라 4입니다. 오타입니다. ^^;
node #0 -> 4 cpus로 수정합니다.

감사합니다.

응답 (/PER-CPU/?REPLYTOCOM=305127#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

‹ kernel/head.S – __enable_mmu: (http://jake.dothome.co.kr/__enable_mmu/)

Per-cpu -2- (초기화) › (http://jake.dothome.co.kr/setup_per_cpu_areas/)

문c 블로그 (2015 ~ 2024)