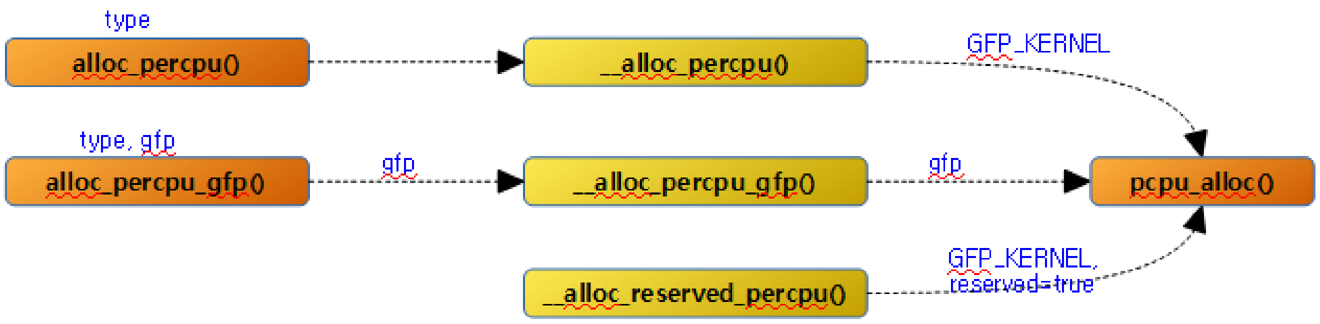
per-cpu -3-(dynamic allocation)

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

<kernel v5.0>

per-cpu -3-(dynamic allocation)



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

alloc_percpu()

include/linux/percpu.h

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

Allocate per-CPU memory of type request.

__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);
```

Allocate per-CPU memory as request @size and @align values.

alloc_percpu_gfp()

include/linux/percpu.h

```
#define alloc_percpu_gfp(type, gfp)                                    \
        (typeof(type) __percpu *)__alloc_percpu_gfp(sizeof(type),      \
                                                __alignof__(type), gfp)
```

Use request types and gfp flags to allocate per-cpu memory.

__alloc_percpu_gfp()

mm/percpu.c

```
/**
 * __alloc_percpu_gfp - allocate dynamic percpu area
 * @size: size of area to allocate in bytes
 * @align: alignment of area (max PAGE_SIZE)
 * @gfp: allocation flags
 *
 * Allocate zero-filled percpu area of @size bytes aligned at @align.  If
 * @gfp doesn't contain %GFP_KERNEL, the allocation doesn't block and can
 * be called from any context but is a lot more likely to fail. If @gfp
 * has __GFP_NOWARN then no warning will be triggered on invalid or failed
 * allocation requests.
 *
 * RETURNS:
 * Percpu pointer to the allocated area on success, NULL on failure.
 */
```

```
void __percpu *__alloc_percpu_gfp(size_t size, size_t align, gfp_t gfp)
{
        return pcpu_alloc(size, align, false, gfp);
}
EXPORT_SYMBOL_GPL(__alloc_percpu_gfp);
```

Allocate per-CPU memory with request size, align, and gfp flag values.

__alloc_reserved_percpu()

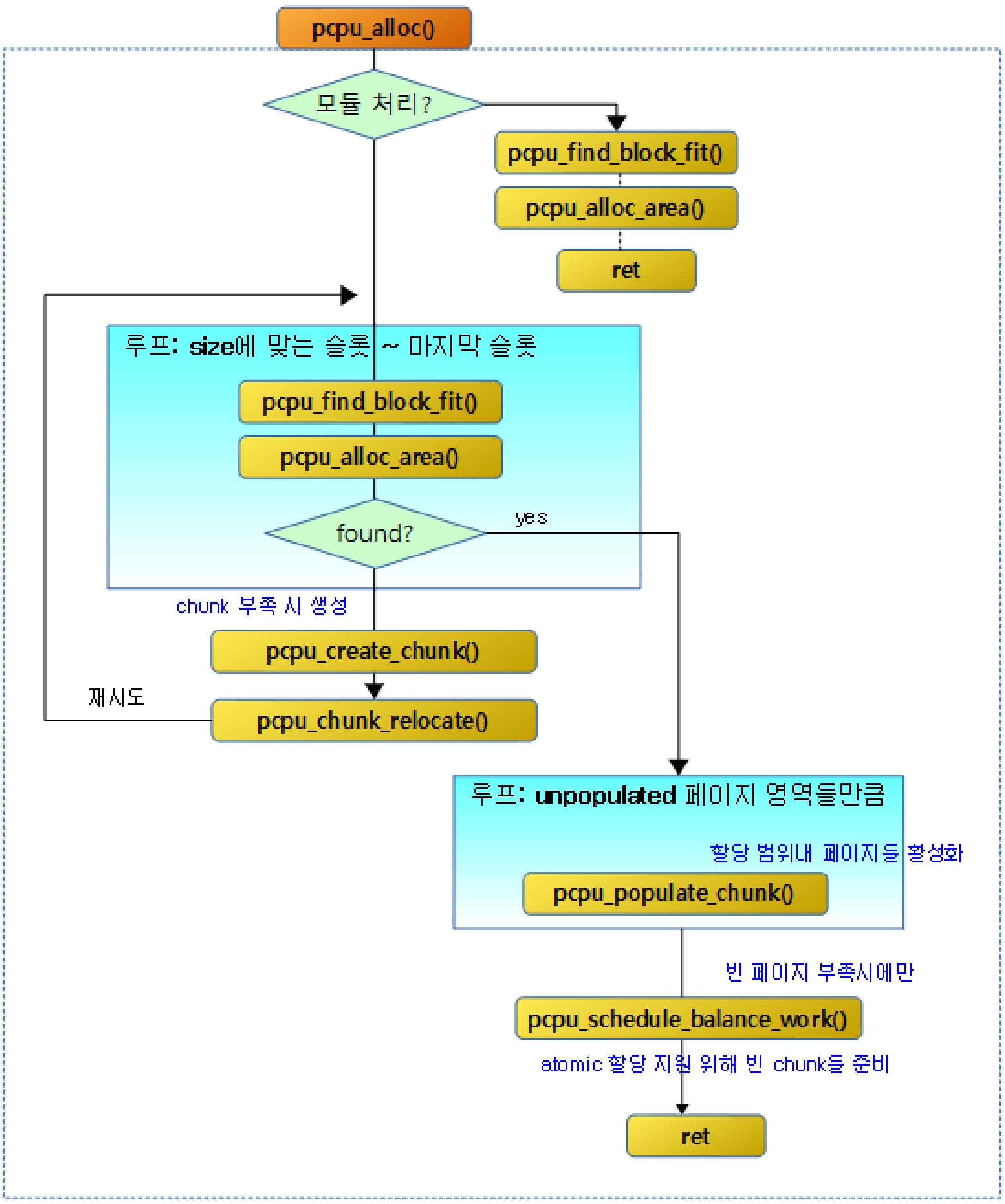
mm/percpu.c

```
/**
 * __alloc_reserved_percpu - allocate reserved percpu area
 * @size: size of area to allocate in bytes
 * @align: alignment of area (max PAGE_SIZE)
 *
 * Allocate zero-filled percpu area of @size bytes aligned at @align
 * from reserved percpu area if arch has set it up; otherwise,
 * allocation is served from the same dynamic area.  Might sleep.
 * Might trigger writeouts.
 *
 * CONTEXT:
 * Does GFP_KERNEL allocation.
 *
 * RETURNS:
 * Percpu pointer to the allocated area on success, NULL on failure.
 */
```

```
void __percpu *__alloc_reserved_percpu(size_t size, size_t align)
{
        return pcpu_alloc(size, align, true, GFP_KERNEL);
}
```

The static per-CPU data declaration areas used by modules at compile time are not ready-to-use data spaces. They are called at runtime when the module is loaded, and allocate them within the reserved area of the first chunk.

PCPU Dynamic Allocation Main



(http://jake.dothome.co.kr/wp-content/uploads/2016/11/pcpu_alloc-1a.png)

pcpu_alloc()

mm/percpu.c -1/3-

```
/**
 * pcpu_alloc - the percpu allocator
 * @size: size of area to allocate in bytes
 * @align: alignment of area (max PAGE_SIZE)
 * @reserved: allocate from the reserved chunk if available
 * @gfp: allocation flags
 *
 * Allocate percpu area of @size bytes aligned at @align.  If @gfp doesn't
 * contain %GFP_KERNEL, the allocation is atomic. If @gfp has __GFP_NOWARN
 * then no warning will be triggered on invalid or failed allocation
 * requests.
 *
 * RETURNS:
 * Percpu pointer to the allocated area on success, NULL on failure.
 */
```

```
static void __percpu *pcpu_alloc(size_t size, size_t align, bool reserved,
                                 gfp_t gfp)
{
        /* whitelisted flags that can be passed to the backing allocators */
        gfp_t pcpu_gfp = gfp & (GFP_KERNEL | __GFP_NORETRY | __GFP_NOWARN);
        bool is_atomic = (gfp & GFP_KERNEL) != GFP_KERNEL;
        bool do_warn = !(gfp & __GFP_NOWARN);
        static int warn_limit = 10;
        struct pcpu_chunk *chunk;
        const char *err;
        int slot, off, cpu, ret;
        unsigned long flags;
        void __percpu *ptr;
        size_t bits, bit_align;

        /*
         * There is now a minimum allocation size of PCPU_MIN_ALLOC_SIZE,
         * therefore alignment must be a minimum of that many bytes.
         * An allocation may have internal fragmentation from rounding up
         * of up to PCPU_MIN_ALLOC_SIZE - 1 bytes.
         */
        if (unlikely(align < PCPU_MIN_ALLOC_SIZE))
                align = PCPU_MIN_ALLOC_SIZE;

        size = ALIGN(size, PCPU_MIN_ALLOC_SIZE);
        bits = size >> PCPU_MIN_ALLOC_SHIFT;
        bit_align = align >> PCPU_MIN_ALLOC_SHIFT;

        if (unlikely(!size || size > PCPU_MIN_UNIT_SIZE || align > PAGE_SIZE ||
                     !is_power_of_2(align))) {
                WARN(do_warn, "illegal size (%zu) or align (%zu) for percpu allocation\n",
                     size, align);
                return NULL;
        }

        if (!is_atomic) {
                /*
                 * pcpu_balance_workfn() allocates memory under this mutex,
                 * and it may wait for memory reclaim. Allow current task
                 * to become OOM victim, in case of memory pressure.
                 */
42                 if (gfp & __GFP_NOFAIL)
43                         mutex_lock(&pcpu_alloc_mutex);
44                 else if (mutex_lock_killable(&pcpu_alloc_mutex))
45                         return NULL;
46         }
47 
48         spin_lock_irqsave(&pcpu_lock, flags);
49 
50         /* serve reserved allocations from the reserved chunk if availab
   le */
51         if (reserved && pcpu_reserved_chunk) {
52                 chunk = pcpu_reserved_chunk;
53 
54                 off = pcpu_find_block_fit(chunk, bits, bit_align, is_ato
   mic);
55                 if (off < 0) {
56                         err = "alloc from reserved chunk failed";
57                         goto fail_unlock;
58                 }
59 
60                 off = pcpu_alloc_area(chunk, bits, bit_align, off);
61                 if (off >= 0)
62                         goto area_found;
63 
64                 err = "alloc from reserved chunk failed";
65                 goto fail_unlock;
66         }
```

Dynamically allocates per-CPU memory with request size and align values. If you're calling from a module, call Reserved RESERVED to make an allocation in the per-cpu area. If there is not enough space to allocate, a new chunk is added, but if it is an atomic request, it fails without scaling it.

- In line 6 of the code, determine whether the atomic is requested. If you don't use the GFP_KERNEL option, you've received an atomic request. If you call it using a alloc_percpu( ) function, etc., you will always use the GFP_KERNEL option, so you don't use an atomic condition.
  - Currently, if you use the alloc_percpu_gfp( ) function in the kernel, you can change the gfp option, but the actual applied code has not yet used the gfp option other than the GFP_KERNEL option. This is a function that has been prepared in advance to use the atomic condition in the future.
  - If you run this function as an atomic condition, you are restricting the allocation of per-CPU data only on populated pages. If there is not enough chunk, chunks can be created and unpopuated pages can be used through the population process.
- In lines 22~23 of code, limit the minimum sorting unit of per-cpu allocation to a minimum of 4 bytes.
- In lines 25~27 of code, the allocation size is sorted by the per-cpu minimum sorting unit. Then, find the required number of bits in the calculated size and sort units.
  - 예) size=32, align=4
    - bits=8, bit_align=1
- On lines 29~34 of code, if the size is 0 or exceeds the unit size, if align exceeds the page unit, or if it does not use a power of 2, it will print a warning message and return null.
- In line 36~46 of the code, if it is not an atomic allocation request, it is possible to give up halfway through and return null without acquiring a lock for per-cpu allocation in an OOM situation.
- In line 48 of code, interrupt is prevented while preparing the assignment.

- In lines 51~66 of code, the static per-CPU allocation used for the module is managed using the reserved area of the first chunk. Check to see if the space is allocated in this chunk, and then try to allocate it. If it finds a suitable space, it goes to area_found: label, and if it doesn't find a suitable space, it prints the reason for the allocation failure and goes to the faile_unlock: label to exit the function.

mm/percpu.c -2/3-

```
01 restart:
02         /* search through normal chunks */
03         for (slot = pcpu_size_to_slot(size); slot < pcpu_nr_slots; slot++) {
04                 list_for_each_entry(chunk, &pcpu_slot[slot], list) {
05                         off = pcpu_find_block_fit(chunk, bits, bit_align,
06                                                           is_atomic);
07                         if (off < 0)
08                                 continue;
09 
10                         off = pcpu_alloc_area(chunk, bits, bit_align, off);
11                         if (off >= 0)
12                                 goto area_found;
13 
14                 }
15         }
16 
17         spin_unlock_irqrestore(&pcpu_lock, flags);
18 
19         /*
20          * No space left.  Create a new chunk.  We don't want multiple
21          * tasks to create chunks simultaneously.  Serialize and create iff
22          * there's still no empty chunk after grabbing the mutex.
23          */
24         if (is_atomic) {
25                 err = "atomic alloc failed, no space left";
26                 goto fail;
27         }
28 
29         if (list_empty(&pcpu_slot[pcpu_nr_slots - 1])) {
30                 chunk = pcpu_create_chunk(pcpu_gfp);
31                 if (!chunk) {
32                         err = "failed to allocate new chunk";
33                         goto fail;
34                 }
35 
36                 spin_lock_irqsave(&pcpu_lock, flags);
37                 pcpu_chunk_relocate(chunk, -1);
38         } else {
39                 spin_lock_irqsave(&pcpu_lock, flags);
40         }
41 
42         goto restart;
```

The restart: label performs processing on dynamic per-CPU allocation.

- In line 3 of code, we traverse from the slot corresponding to the size to be assigned to find the appropriate chunk first, from the top slot.
    - pcpu_size_to_slot() returns a slot number that corresponds to the size range.
        - e.g. size of 44K, return slot 13.

- In lines 4~8 of the code, it traverses the list of chunks in the slot, and if there is no free space larger than the size to be allocated, skip it.
- If you have been assigned the size requested for assignment in line 10~12 of the code area_found; Go to Labels.
- If atomic is requested in line 24~27 of the code, it goes to the fail: label without retrying.
- In code lines 29~42, the top-level slot should always have an empty chunk. If it doesn't exist, it creates an empty chunk, places it in the top slot, and goes to the restart: label to retry.

mm/percpu.c -3/3-

```
01 area_found:
02         pcpu_stats_area_alloc(chunk, size);
03         spin_unlock_irqrestore(&pcpu_lock, flags);
04 
05         /* populate if not all pages are already there */
06         if (!is_atomic) {
07                 int page_start, page_end, rs, re;
08 
09                 page_start = PFN_DOWN(off);
10                 page_end = PFN_UP(off + size);
11 
12                 pcpu_for_each_unpop_region(chunk->populated, rs, re,
13                                            page_start, page_end) {
14                         WARN_ON(chunk->immutable);
15 
16                         ret = pcpu_populate_chunk(chunk, rs, re, pcpu_gfp);
17 
18                         spin_lock_irqsave(&pcpu_lock, flags);
19                         if (ret) {
20                                 pcpu_free_area(chunk, off);
21                                 err = "failed to populate";
22                                 goto fail_unlock;
23                         }
24                         pcpu_chunk_populated(chunk, rs, re, true);
25                         spin_unlock_irqrestore(&pcpu_lock, flags);
26                 }
27 
28                 mutex_unlock(&pcpu_alloc_mutex);
29         }
30 
31         if (pcpu_nr_empty_pop_pages < PCPU_EMPTY_POP_PAGES_LOW)
32                 pcpu_schedule_balance_work();
33 
34         /* clear the areas and return address relative to base address */
35         for_each_possible_cpu(cpu)
36                 memset((void *)pcpu_chunk_addr(chunk, cpu, 0) + off, 0, size);
37 
38         ptr = __addr_to_pcpu_ptr(chunk->base_addr + off);
39         kmemleak_alloc_percpu(ptr, size, gfp);
40 
41         trace_percpu_alloc_percpu(reserved, is_atomic, size, align,
42                         chunk->base_addr, off, ptr);
43 
44         return ptr;
45 
46 fail_unlock:
47         spin_unlock_irqrestore(&pcpu_lock, flags);
48 fail:
```

```
49          trace_percpu_alloc_percpu_fail(reserved, is_atomic, size, align);
50  
51          if (!is_atomic && do_warn && warn_limit) {
52                  pr_warn("allocation failed, size=%zu align=%zu atomic=%d, %s\n",
53                          size, align, is_atomic, err);
54                  dump_stack();
55                  if (!--warn_limit)
56                          pr_info("limit reached, disable warning\n");
57          }
58          if (is_atomic) {
59                  /* see the flag handling in pcpu_blance_workfn() */
60                  pcpu_atomic_alloc_failed = true;
61                  pcpu_schedule_balance_work();
62          } else {
63                  mutex_unlock(&pcpu_alloc_mutex);
64          }
65          return NULL;
66  }
```

area_found: The label has a routine for subsequent processing if the per-cpu allocation succeeds, and the fail: label returns null after performing an error output for the cause if the allocation fails.

- In line 2 of code, increment and update the stats for the per-cpu allocation.
- If you are not requesting atomic processing from code lines 6~29, you don't need to use the scheduler, so it will be activated immediately. Locate the start (rs) and end (re) addresses of the un-populated area of the chunk from the start pfn to the end pfn of the assigned page, and populate the specified area of the chunk. The actual page assigned is mapped to the specified vmalloc area. The page numbers are all based on the first unit of the chunk, not the PFN per page. Then, the pcpu_chunk_populated() function records the information in the chunk that the area has populated in a bitmapped fashion.
- If the number of empty populated pages in line 31~32 is less than 2, call pcpu_balance_workfn( ) through the workqueue in the background to populate and get populated free pages from PCPU_EMPTY_POP_PAGES_LOW(2)~HIGH(4) so that atomic assignment can be made to one empty chunk. Obtain a pre-populate page for the automation operation.
- Clean the area of the assigned size in code lines 35~36 to zero.
- Convert to the address given in line 38 to a per-cpu pointer address and return it. The per-cpu pointer address does not refer to the address of the actual data corresponding to unit 0, but to that address minus delta. In actual use, this value is added to the TPIDRPRW value stored by the CPU.
    - TPIDRPRW holds the unit offset + delta value that corresponds to the cpu at the time of the first chunk's initial configuration, and this value will not change in the future.
    - The delta value is the hypothetical address of the base offset of the lowest node (group) calculated when the first chunk was first set minus the start address of the per-cpu section.
    - In the case of the actual static variable, it has an address created at compile time in the per-cpu section, and in dynamic allocation, it refers to a value that has been pre-decremented to account for its delta value. Therefore, under no circumstances should the per-cpu pointer value be accessed. It was considered to allow the use of the same API functions such as static per-cpu data and dynamically allocated per-cpu data regardless of the this_cpu_ptr( ).

- In lines 39~44 of the code, we register the object to watch for memory leaks, and since it succeeded, we exit the function.
- If the assignment fails on lines 46~57 of the code, enter this routine. If an atomic assignment request is not received, a warning is printed.
- If you receive an atomic assignment request on code lines 58~65, call the pcpu_schedule_balance_work( ) routine to receive the schedule assignment separately from the workqueue and prepare the allocation of the populated free page.

---

chunk my need to search for free space

pcpu_find_block_fit()

mm/percpu.c

```
/**
 * pcpu_find_block_fit - finds the block index to start searching
 * @chunk: chunk of interest
 * @alloc_bits: size of request in allocation units
 * @align: alignment of area (max PAGE_SIZE bytes)
 * @pop_only: use populated regions only
 *
 * Given a chunk and an allocation spec, find the offset to begin searching
 * for a free region.  This iterates over the bitmap metadata blocks to
 * find an offset that will be guaranteed to fit the requirements.  It is
 * not quite first fit as if the allocation does not fit in the contig hint
 * of a block or chunk, it is skipped.  This errs on the side of caution
 * to prevent excess iteration.  Poor alignment can cause the allocator to
 * skip over blocks and chunks that have valid free areas.
 *
 * RETURNS:
 * The offset in the bitmap to begin searching.
 * -1 if no offset is found.
 */
```

```
static int pcpu_find_block_fit(struct pcpu_chunk *chunk, int alloc_bits,
                               size_t align, bool pop_only)
{
        int bit_off, bits, next_off;

        /*
         * Check to see if the allocation can fit in the chunk's contig hint.
         * This is an optimization to prevent scanning by assuming if it
         * cannot fit in the global hint, there is memory pressure and creating
         * a new chunk would happen soon.
         */
        bit_off = ALIGN(chunk->contig_bits_start, align) -
                  chunk->contig_bits_start;
        if (bit_off + alloc_bits > chunk->contig_bits)
                return -1;

        bit_off = chunk->first_bit;
        bits = 0;
        pcpu_for_each_fit_region(chunk, alloc_bits, align, bit_off, bits) {
```

```
20                  if (!pop_only || pcpu_is_populated(chunk, bit_off, bits,
21                                                     &next_off))
22                          break;
23
24                  bit_off = next_off;
25                  bits = 0;
26          }
27
28          if (bit_off == pcpu_chunk_map_bits(chunk))
29                  return -1;
30
31          return bit_off;
32  }
```

Find a suitable empty space in the @chunk and return a bit_off value based on chunk. If the @pop_only is given as 1, it will only search on populate pages.

- In lines 12~15 of the code, if we look at the maximum contiguous free space and don't have space to allocate @alloc_bits in @align sort units, return -1.
- Starting from the bit_off position in code lines 17~19, it traverses in bits and finds the bit_off by finding an empty space where the align condition is satisfied.
- If the @pop_only is 20 in line 22~0 of the code, the first found position is confirmed. Otherwise, if the @pop_only is set to 1, check whether the space from @bit_off to bits is populate. If it is not populate, the next_off returns a value that points to the start bit of the next populate page.
- In codelines 24~26, we have taken the starting bit of the next populate page into the next_off, so we put it in the bit_off and continue the loop.
- On code lines 28~31, it returns -1 if the allocation fails, and bit_off if it succeeds.

pcpu_for_each_fit_region()

mm/percpu.c

```
#define pcpu_for_each_fit_region(chunk, alloc_bits, align, bit_off, bits)       \
        for (pcpu_next_fit_region((chunk), (alloc_bits), (align), &(bit_off), \
                                  &(bits));                                     \
             (bit_off) < pcpu_chunk_map_bits((chunk));                         \
             (bit_off) += (bits),                                               \
             pcpu_next_fit_region((chunk), (alloc_bits), (align), &(bit_off), \
                                  &(bits)))
```

Starting with the @bit_off of the I/O factor in the @chunk, it finds the area where free space can be secured @alloc_bits in @align units, and returns the location of the bit offset in the I/O factor @bit_off and the size of the free area in the output factor @bits. Note that each bit shows the allocation status in 4-byte increments.

pcpu_next_fit_region()

mm/percpu.c

```
/**
```

```
 * pcpu_next_fit_region - finds fit areas for a given allocation request
 * @chunk: chunk of interest
 * @alloc_bits: size of allocation
 * @align: alignment of area (max PAGE_SIZE)
 * @bit_off: chunk offset
 * @bits: size of free area
 *
 * Finds the next free region that is viable for use with a given size and
 * alignment.  This only returns if there is a valid area to be used for this
 * allocation.  block->first_free is returned if the allocation request fits
 * within the block to see if the request can be fulfilled prior to the contig
 * hint.
 */
```

```
static void pcpu_next_fit_region(struct pcpu_chunk *chunk, int alloc_bits,
                                 int align, int *bit_off, int *bits)
{
        int i = pcpu_off_to_block_index(*bit_off);
        int block_off = pcpu_off_to_block_off(*bit_off);
        struct pcpu_block_md *block;

        *bits = 0;
        for (block = chunk->md_blocks + i; i < pcpu_chunk_nr_blocks(chunk);
             block++, i++) {
                /* handles contig area across blocks */
                if (*bits) {
                        *bits += block->left_free;
                        if (*bits >= alloc_bits)
                                return;
                        if (block->left_free == PCPU_BITMAP_BLOCK_BITS)
                                continue;
                }

                /* check block->contig_hint */
                *bits = ALIGN(block->contig_hint_start, align) -
                        block->contig_hint_start;
                /*
                 * This uses the block offset to determine if this has been
                 * checked in the prior iteration.
                 */
                if (block->contig_hint &&
                    block->contig_hint_start >= block_off &&
                    block->contig_hint >= *bits + alloc_bits) {
                        *bits += alloc_bits + block->contig_hint_start -
                                 block->first_free;
                        *bit_off = pcpu_block_off_to_off(i, block->first_free);
                        return;
                }
                /* reset to satisfy the second predicate above */
                block_off = 0;

                *bit_off = ALIGN(PCPU_BITMAP_BLOCK_BITS - block->right_free,
                                 align);
                *bits = PCPU_BITMAP_BLOCK_BITS - *bit_off;
                *bit_off = pcpu_block_off_to_off(i, *bit_off);
                if (*bits >= alloc_bits)
                        return;
        }

46              /* no valid offsets were found - fail condition */
47              *bit_off = pcpu_chunk_map_bits(chunk);
48      }
```

Starting with the @bit_off of the input/output factors within the @chunk, we look for areas where free space can be secured as much as @alloc_bits in @align units. It then returns the starting bit offset position of the free region found in the I/O factor @bit_off and the size of the free region in the output factor @bits. Note that each bit shows the allocation status in 4-byte increments.

- In code lines 9~18, it traverses the number of pcpu blocks (pages) in chunks, and if it is called for the first time and the @bits is 0, it will update the @bits if it is allocated in the first free space and exit the function, otherwise it will skip if it is in the middle of the free space.
- If it can be assigned in the free space from code lines 21~34, its position and size are calculated and returned.
- If it can be assigned in the rightmost free space in code lines 36~43, its position and size are calculated and returned.
- In line 47 of code, if it doesn't find any more, it puts the last value in the @bit_off to end the loop outside the function.

Mark the assigned area on a bitmap

pcpu_alloc_area()

mm/percpu.c

```
01      /**
02       * pcpu_alloc_area - allocates an area from a pcpu_chunk
03       * @chunk: chunk of interest
04       * @alloc_bits: size of request in allocation units
05       * @align: alignment of area (max PAGE_SIZE)
06       * @start: bit_off to start searching
07       *
08       * This function takes in a @start offset to begin searching to fit an
09       * allocation of @alloc_bits with alignment @align.  It needs to scan
10       * the allocation map because if it fits within the block's contig hint,
11       * @start will be block->first_free. This is an attempt to fill the
12       * allocation prior to breaking the contig hint.  The allocation and
13       * boundary maps are updated accordingly if it confirms a valid
14       * free area.
15       *
16       * RETURNS:
17       * Allocated addr offset in @chunk on success.
18       * -1 if no matching area is found.
19       */
```

```
01      static int pcpu_alloc_area(struct pcpu_chunk *chunk, int alloc_bits,
02                                 size_t align, int start)
03      {
04              size_t align_mask = (align) ? (align - 1) : 0;
05              int bit_off, end, oslot;
06
07              lockdep_assert_held(&pcpu_lock);
08
09              oslot = pcpu_chunk_slot(chunk);
10
11              /*
12               * Search to find a fit.
13          */
14         end = start + alloc_bits + PCPU_BITMAP_BLOCK_BITS;
15         bit_off = bitmap_find_next_zero_area(chunk->alloc_map, end, start,
16                                              alloc_bits, align_mask);
17         if (bit_off >= end)
18                 return -1;
19 
20         /* update alloc map */
21         bitmap_set(chunk->alloc_map, bit_off, alloc_bits);
22 
23         /* update boundary map */
24         set_bit(bit_off, chunk->bound_map);
25         bitmap_clear(chunk->bound_map, bit_off + 1, alloc_bits - 1);
26         set_bit(bit_off + alloc_bits, chunk->bound_map);
27 
28         chunk->free_bytes -= alloc_bits * PCPU_MIN_ALLOC_SIZE;
29 
30         /* update first free bit */
31         if (bit_off == chunk->first_bit)
32                 chunk->first_bit = find_next_zero_bit(
33                                         chunk->alloc_map,
34                                         pcpu_chunk_map_bits(chunk),
35                                         bit_off + alloc_bits);
36 
37         pcpu_block_update_hint_alloc(chunk, bit_off, alloc_bits);
38 
39         pcpu_chunk_relocate(chunk, oslot);
40 
41         return bit_off * PCPU_MIN_ALLOC_SIZE;
42 }
```

Updates bitmaps, block metadata, and other related information about the per-CPU allocation area.

- In line 9 of the code, the current slot number is determined because the allocation area update can cause the slot to be moved.
- In lines 14~18 of the code, find the free area of the @alloc_bits as the @align sort unit in the range. Returns -1 if no eligible free zone exists.
- In line 21 of the code, fill all assignment maps in the assignment range with 1.
- In line 24~26 of the code, clear all the boundary maps in the assigned range, and set only the start and end +1 bits to 1.
- Update the remaining free bytes at line 28 of code.
- In code lines 31~35, if the allocated area is the first free space, the first free space bit position is updated.
- In line 37 of code, update the per-cpu block metadata in the chunk.
- The code is updated on line 39 if a move of the slot is necessary.
- This is the case when the assignment was successful on line 41 of the code. Returns the bit offset of the allocated area.

Activate assignment page scopes

pcpu_populate_chunk()

mm/percpu-vm.c

```
01 /**
```

```
 * pcpu_populate_chunk - populate and map an area of a pcpu_chunk
 * @chunk: chunk of interest
 * @page_start: the start page
 * @page_end: the end page
 * For each cpu, populate and map pages [@page_start,@page_end) into
 *
 * For each cpu, populate and map pages [@page_start,@page_end) into
 * @chunk.
 *
 * CONTEXT:
 * pcpu_alloc_mutex, does GFP_KERNEL allocation.
 */
```

```
01 static int pcpu_populate_chunk(struct pcpu_chunk *chunk,
02                                int page_start, int page_end, gfp_t gfp)
03 {
04         struct page **pages;
05 
06         pages = pcpu_get_pages();
07         if (!pages)
08                 return -ENOMEM;
09 
10         if (pcpu_alloc_pages(chunk, pages, page_start, page_end, gfp))
11                 return -ENOMEM;
12 
13         if (pcpu_map_pages(chunk, pages, page_start, page_end)) {
14                 pcpu_free_pages(chunk, pages, page_start, page_end);
15                 return -ENOMEM;
16         }
17         pcpu_post_map_flush(chunk, page_start, page_end);
18 
19         return 0;
20 }
```

Enable (population) for the scope of the chunk's request page.

- In line 6~8 of the code, the required page descriptor is allocated. On allocation failure, return to -ENOMEM.
- In code lines 10~11, the required page range is allocated by the number of CPUs. On allocation failure, return to -ENOMEM.
- Map the area pages assigned in lines 13~16 to the vmalloc space.
- When the mapping is done in line 17 of the code, flush the TLB cache.

pcpu_get_pages()

mm/percpu-vm.c

```
/**
 * pcpu_get_pages - get temp pages array
 * @chunk: chunk of interest
 *
 * Returns pointer to array of pointers to struct page which can be inde
xed
 * with pcpu_page_idx().  Note that there is only one array and accesses
 * should be serialized by pcpu_alloc_mutex.
 *
 * RETURNS:
 * Pointer to temp pages array on success.
 */
```

```
static struct page **pcpu_get_pages(void)
{
        static struct page **pages;
        size_t pages_size = pcpu_nr_units * pcpu_unit_pages * sizeof(pages[0]);

        lockdep_assert_held(&pcpu_alloc_mutex);

        if (!pages)
                pages = pcpu_mem_zalloc(pages_size, GFP_KERNEL);
        return pages;
}
```

For chunk allocation, the entire per-CPU unit receives a memory allocation equal to the page descriptor size required.

pcpu_map_pages()

mm/percpu-vm.c

```
/**
 * pcpu_map_pages - map pages into a pcpu_chunk
 * @chunk: chunk of interest
 * @pages: pages array containing pages to be mapped
 * @page_start: page index of the first page to map
 * @page_end: page index of the last page to map + 1
 *
 * For each cpu, map pages [@page_start,@page_end) into @chunk.  The
 * caller is responsible for calling pcpu_post_map_flush() after all
 * mappings are complete.
 *
 * This function is responsible for setting up whatever is necessary for
 * reverse lookup (addr -> chunk).
 */
```

```
static int pcpu_map_pages(struct pcpu_chunk *chunk,
                          struct page **pages, int page_start, int page_end)
{
        unsigned int cpu, tcpu;
        int i, err;

        for_each_possible_cpu(cpu) {
                err = __pcpu_map_pages(pcpu_chunk_addr(chunk, cpu, page_start),
                                       &pages[pcpu_page_idx(cpu, page_start)],
                                       page_end - page_start);
                if (err < 0)
                        goto err;

                for (i = page_start; i < page_end; i++)
                        pcpu_set_page_chunk(pages[pcpu_page_idx(cpu, i)],
                                            chunk);
        }
        return 0;
err:
        for_each_possible_cpu(tcpu) {
                if (tcpu == cpu)
                        break;
                __pcpu_unmap_pages(pcpu_chunk_addr(chunk, tcpu, page_start),
                                   page_end - page_start);
        }
        pcpu_post_unmap_tlb_flush(chunk, page_start, page_end);
        return err;
}
```

Map the assigned area pages to the vmalloc space

- In lines 7~12 of code, loop around the number of possible CPUs and map the per-cpu chunk to the vmalloc space.
- In lines 14~16 of the code, set each page (page->index) to point to a pcpu_chunk.

__pcpu_map_pages()

mm/percpu-vm.c

```
static int __pcpu_map_pages(unsigned long addr, struct page **pages,
                            int nr_pages)
{
        return map_kernel_range_noflush(addr, nr_pages << PAGE_SHIFT,
                                        PAGE_KERNEL, pages);
}
```

Map the assigned zone pages to the request vmalloc virtual address space

map_kernel_range_noflush()

mm/vmalloc.c

```
/**
 * map_kernel_range_noflush - map kernel VM area with the specified page
s
 * @addr: start of the VM area to map
 * @size: size of the VM area to map
 * @prot: page protection flags to use
 * @pages: pages to map
 *
 * Map PFN_UP(@size) pages at @addr.  The VM area @addr and @size
 * specify should have been allocated using get_vm_area() and its
 * friends.
 *
 * NOTE:
 * This function does NOT do any cache flushing.  The caller is
 * responsible for calling flush_cache_vmap() on to-be-mapped areas
 * before calling this function.
 *
 * RETURNS:
 * The number of pages mapped on success, -errno on failure.
 */
```

```
int map_kernel_range_noflush(unsigned long addr, unsigned long size,
                             pgprot_t prot, struct page **pages)
{
        return vmap_page_range_noflush(addr, addr + size, prot, pages);
}
```

Map the assigned zone pages to the requested vmalloc virtual address space

Is the scope active?

pcpu_is_populated()

mm/percpu.c

```
/**
 * pcpu_is_populated - determines if the region is populated
 * @chunk: chunk of interest
 * @bit_off: chunk offset
 * @bits: size of area
 * @next_off: return value for the next offset to start searching
 *
 * For atomic allocations, check if the backing pages are populated.
 *
 * RETURNS:
 * Bool if the backing pages are populated.
 * next_index is to skip over unpopulated blocks in pcpu_find_block_fit.
 */
```

```
static bool pcpu_is_populated(struct pcpu_chunk *chunk, int bit_off, int bits,
                              int *next_off)
{
        int page_start, page_end, rs, re;

        page_start = PFN_DOWN(bit_off * PCPU_MIN_ALLOC_SIZE);
        page_end = PFN_UP((bit_off + bits) * PCPU_MIN_ALLOC_SIZE);

        rs = page_start;
        pcpu_next_unpop(chunk->populated, &rs, &re, page_end);
        if (rs >= page_end)
                return true;

        *next_off = re * PAGE_SIZE / PCPU_MIN_ALLOC_SIZE;
        return false;
}
```

Returns whether the space from @bit_off to @bits in the @chunk is populate. The output argument @next_off contains the offset location from which the next search will begin.

pcpu_next_unpop()

mm/percpu.c

```
static void pcpu_next_unpop(unsigned long *bitmap, int *rs, int *re, int end)
{
        *rs = find_next_zero_bit(bitmap, end, *rs);
        *re = find_next_bit(bitmap, end, *rs + 1);
}
```

Calculate the start @rs and end @re of unactivated per-CPU pages up to @end.

consultation

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

< kmalloc vs vmalloc (http://jake.dothome.co.kr/kmalloc-vs-vmalloc/)

Freeze (hibernation/suspend) > (http://jake.dothome.co.kr/freeze/)