Zonned Allocator -3- (Buddy Page Allocation)

2016-05-13 (http://jake.dothome.co.kr/buddy-alloc/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Zonned Allocator -3- (Buddy Page Allocation)

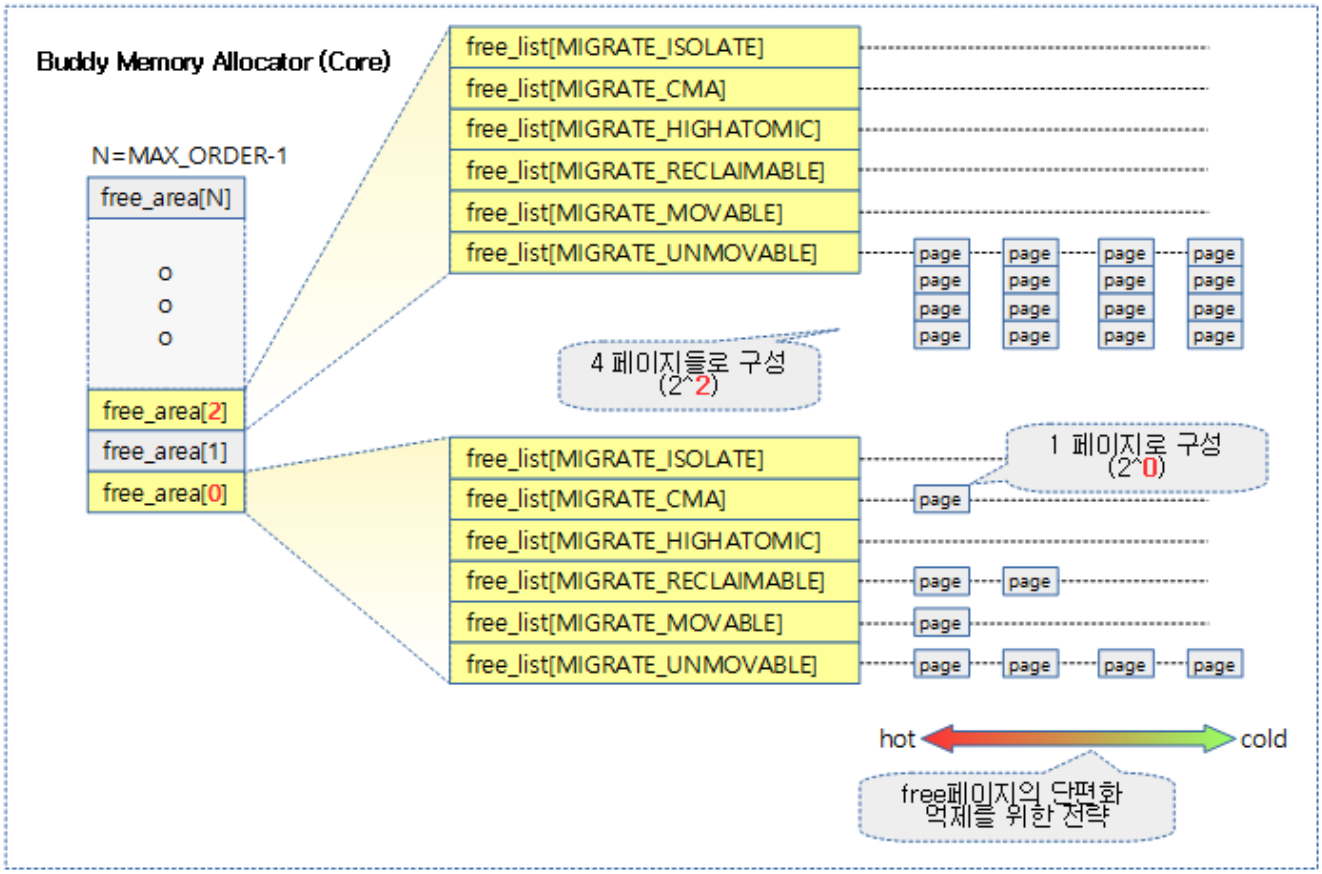
Buddy Memory Allocator is a system that manages the allocation of physical memory by dividing it into pages, and supports the allocation/cancellation of contiguous physical pages. In addition, fragmentation-related algorithms are used to ensure that the largest contiguous pages possible.

- The free memory page is divided into two order page sizes.
  - From 2^0=1 page to 2^(MAX_ORDER-1)=1024 pages, it is divided into 11 slots.
    - MAX_ORDER=11
      - In kernel 2.4.x I used 10 as defalut.
      - CONFIG_FORCE_MAX_ZONEORDER can be resized using kernel options.
- The following structure is in place to ensure that each order slot is also not fragmented.
  - In order to keep pages with the same mobility attribute as close together as possible, each order slot is managed by migratetype.
    - By dividing and managing in this way, the process of page retrieval and memory compaction can be more efficient.
    - In the NUMA system, you can also create ZONE_MOVABLE regions specifically to help more MIGRATE_MOVABLE types.
  - In the free_list that contains each page, the free pages form pairs (buddies), and when two pairs (buddies) are combined, they can be merged into a larger order, and if necessary, they can be divided into one smaller order.
    - Now, we no longer use a bitmap named map to manage buddies, but only a list named free_list and page information.
  - The free_list has hot properties in the leading direction and cold properties in the aft direction.
    - The hot and cold attributes are managed in correspondence with the position of the head and tail of the list, respectively.
      - hot: The pages placed earlier in the list search are the pages that are likely to be reassigned and used.
      - cold: The last page in the list search is the one that is likely to be consolidated and progressively move up to the top order. This allows you to suppress the fragmentation of free pages as much as possible.
- If the migrate type is CMA, both CMA pages and movable pages can be used in this area.
  - Once CMA pages are assigned, they are managed separately by CMA Memory Allocator.

- As the management techniques of the Buddy system continue to be upgraded, the complexity is increasing, but the efficiency of the Buddy system (avoiding fragmentation) is increasing as much as possible

The following figure shows the core portion of the Buddy memory allocator.

- As of kernel v4.4-rc1, the following changes have been made:
  - MIGRATE_RECLAIMABLE position and the position of the MIGRATE_MOVABLE have changed.
  - The MIGRATE_RESERVE was gone and replaced by a MIGRATE_HIGHATOMIC.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/buddy-1d.png)

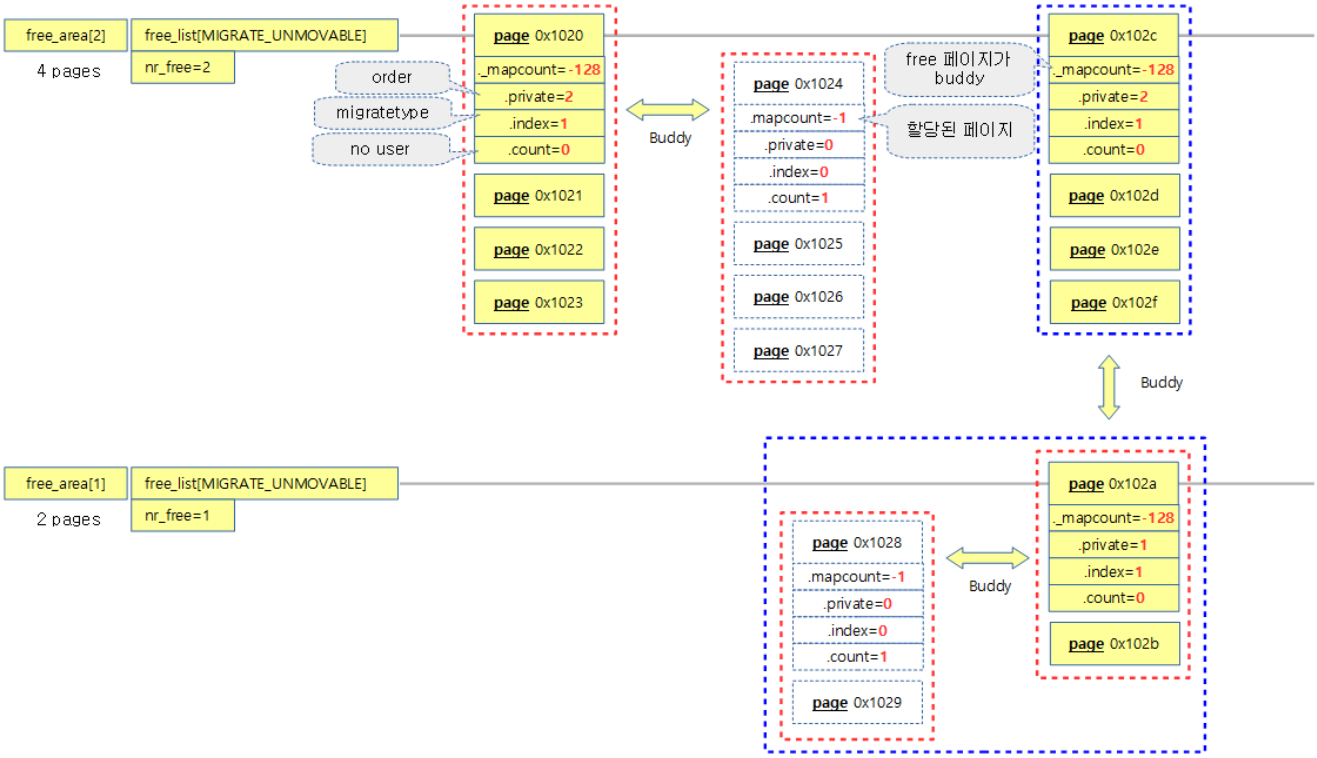
Page properties related to the buddy system

- _mapcount
  - Buddy System's free page page =-128 (PAGE_BUDDY_MAPCOUNT_VALUE)
  - Pages assigned and pulled out of the buddy system=-1
- private
  - The order value used when managed by the buddy system
  - When a page is assigned and exits the buddy system, it is reset to zero.
- index
  - MigrateType Distinct in Buddy System
- count
  - Pages that became free without user=0

- If allocated and used, increment

The following figure shows that 12 pages are registered as free pages in the buddy system, and the matching pages are assigned and used.

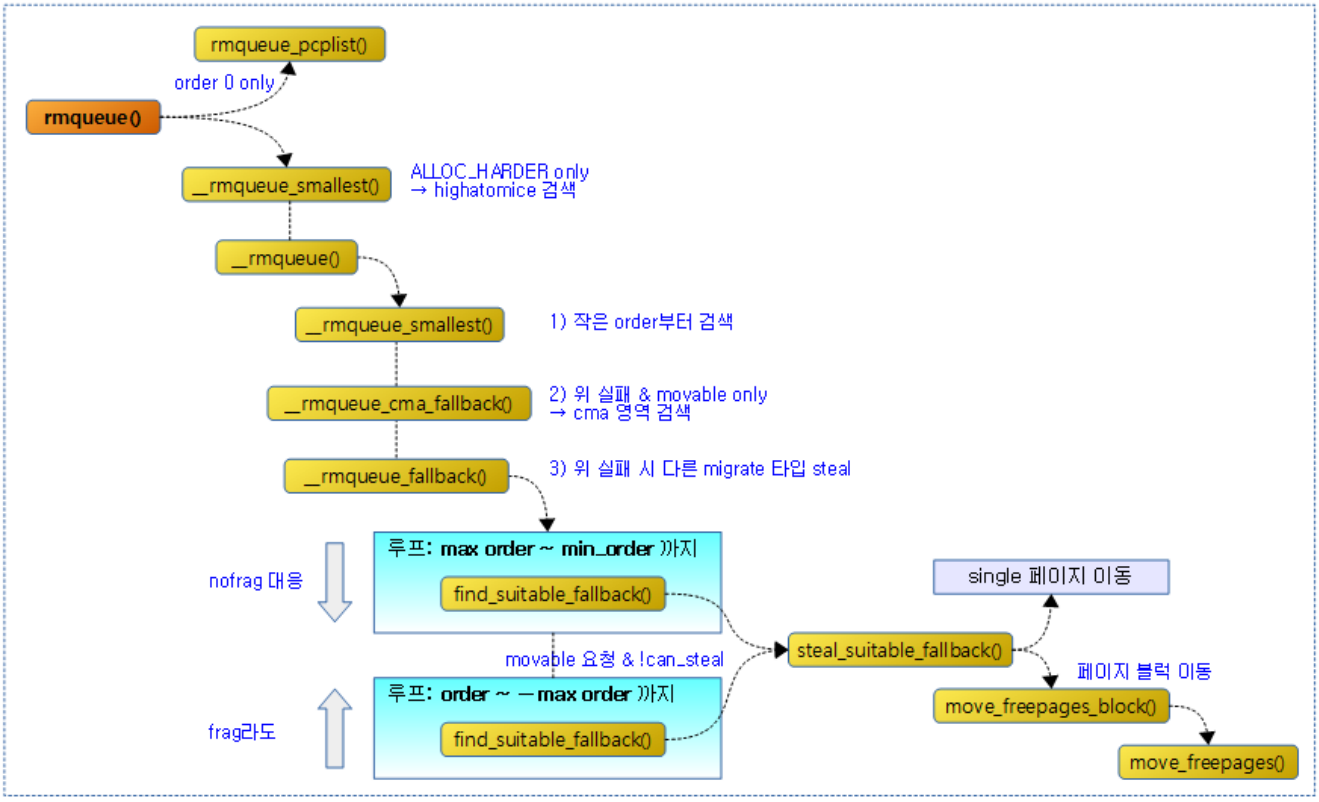
- Two free pages of 4 consecutive pages are listed in free_area[2]
- Two free pages of 2 consecutive pages are listed in free_area[1]



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/buddy-2.png)

Buddy Page Assignment

The following figure shows the process of calling after the rmqueue() function.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/rmqueue-3.png)

rmqueue()

mm/page_alloc.c

```
1  /*
2   * Allocate a page from the given zone. Use pcplists for order-0 allocat
   ions.
3   */
```

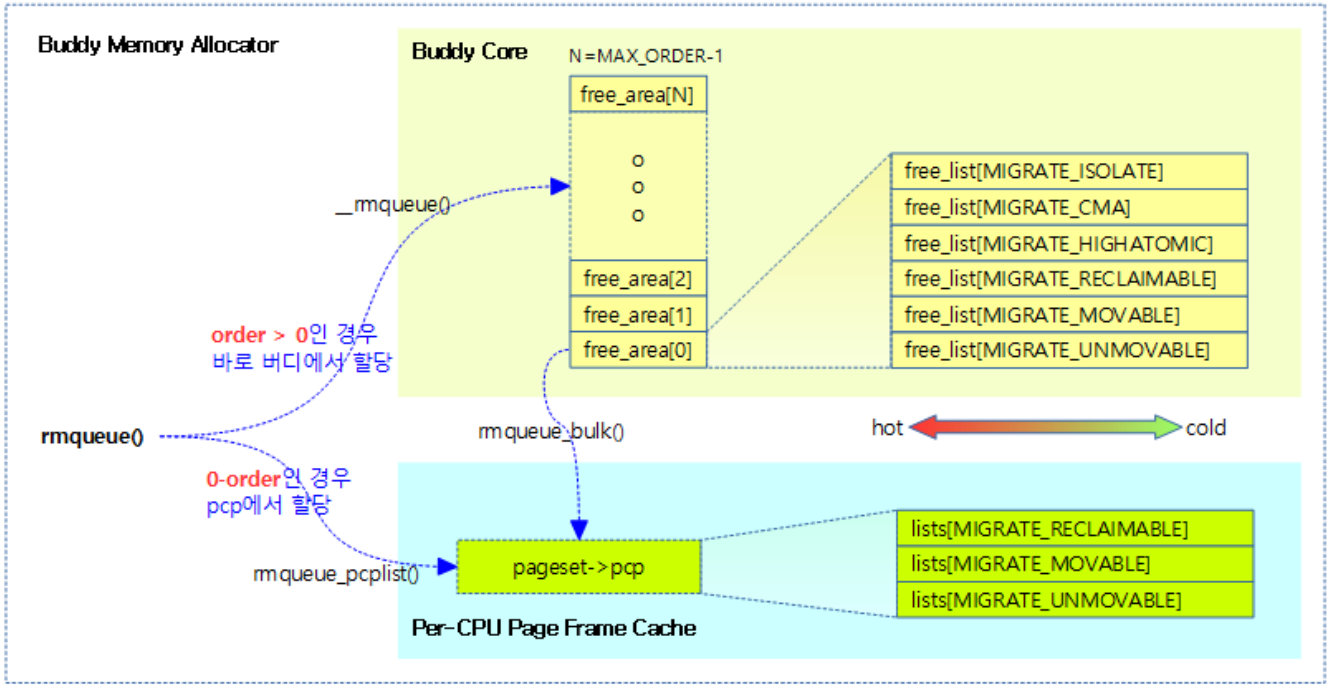
```
01 static inline
02 struct page *rmqueue(struct zone *preferred_zone,
03                        struct zone *zone, unsigned int order,
04                        gfp_t gfp_flags, unsigned int alloc_flags,
05                        int migratetype)
06 {
07         unsigned long flags;
08         struct page *page;
09
10         if (likely(order == 0)) {
11                 page = rmqueue_pcplist(preferred_zone, zone, order,
12                                        gfp_flags, migratetype, alloc_flags);
13                 goto out;
14         }
15
16         /*
17          * We most definitely don't want callers attempting to
18          * allocate greater than order-1 page units with __GFP_NOFAIL.
19          */
20         WARN_ON_ONCE((gfp_flags & __GFP_NOFAIL) && (order > 1));
21         spin_lock_irqsave(&zone->lock, flags);
22
23         do {
24                 page = NULL;
25                 if (alloc_flags & ALLOC_HARDER) {
26                         page = __rmqueue_smallest(zone, order, MIGRATE_H
   IGHATOMIC);
27                         if (page)
```

```
28                                      trace_mm_page_alloc_zone_locked(page, or
der, migratetype);
29                      }
30                      if (!page)
31                              page = __rmqueue(zone, order, migratetype, alloc
_flags);
32              } while (page && check_new_pages(page, order));
33              spin_unlock(&zone->lock);
34              if (!page)
35                      goto failed;
36              __mod_zone_freepage_state(zone, -(1 << order),
37                                        get_pcppage_migratetype(page));
38
39              __count_zid_vm_events(PGALLOC, page_zonenum(page), 1 << order);
40              zone_statistics(preferred_zone, zone);
41              local_irq_restore(flags);
42
43      out:
44              /* Separate test+clear to avoid unnecessary atomics */
45              if (test_bit(ZONE_BOOSTED_WATERMARK, &zone->flags)) {
46                      clear_bit(ZONE_BOOSTED_WATERMARK, &zone->flags);
47                      wakeup_kswapd(zone, 0, 0, zone_idx(zone));
48              }
49
50              VM_BUG_ON_PAGE(page && bad_range(zone, page), page);
51              return page;
52
53      failed:
54              local_irq_restore(flags);
55              return NULL;
56      }
```

Assign @order pages of @migratype through the buddy system in the request zone. Returns the assigned page descriptor on success, and null if it fails.

- In code lines 10~14, there is a high probability that when requesting a 0-order page allocation, it will be allocated from pcp (Per CPU Page Frame Cache), which acts as a 0-order only buddy cache.
- If a non-23-order page is requested in line 32~0 of the code, the allocation is performed by the buddy system. If the ALLOC_HARDER flag is used, it will first try to allocate it from the highatomic type list.
  - The ALLOC_HARDER flag is set by the __GFP_HIGH flag used in the gfp_mask when using a GFP_ATOMIC flag.
    - #define GFP_ATOMIC (__GFP_HIGH|__GFP_ATOMIC|__GFP_KSWAPD_RECLAIM)
  - The check_new_pages() function checks the integrity of the assigned page when debugging is enabled. (hwpoison, etc.)
- Decrease the free page counter by the number of pages allocated in line 36~40 of the code, and increase the PGALLOC counter. Then, it increases or decreases the hit and miss counters for whether or not it was allocated in the first requested zone.
- In lines 43~51 of the code, the page allocation succeeds or fails and exits with the out: label. Before leaving, inspect the zone for a boost watermark and if it's on, clear the flag and wake up kswapd.
  - 참고: mm, page_alloc: do not wake kswapd with zone lock held (https://github.com/torvalds/linux/commit/73444bc4d8f92e46a20cb6bd3342fc2ea75c6787)

The following figure shows how page allocation is handled by the buddy system, including PCP, via the rmqueue() function.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/rmqueue-1a.png)

__rmqueue()

mm/page_alloc.c

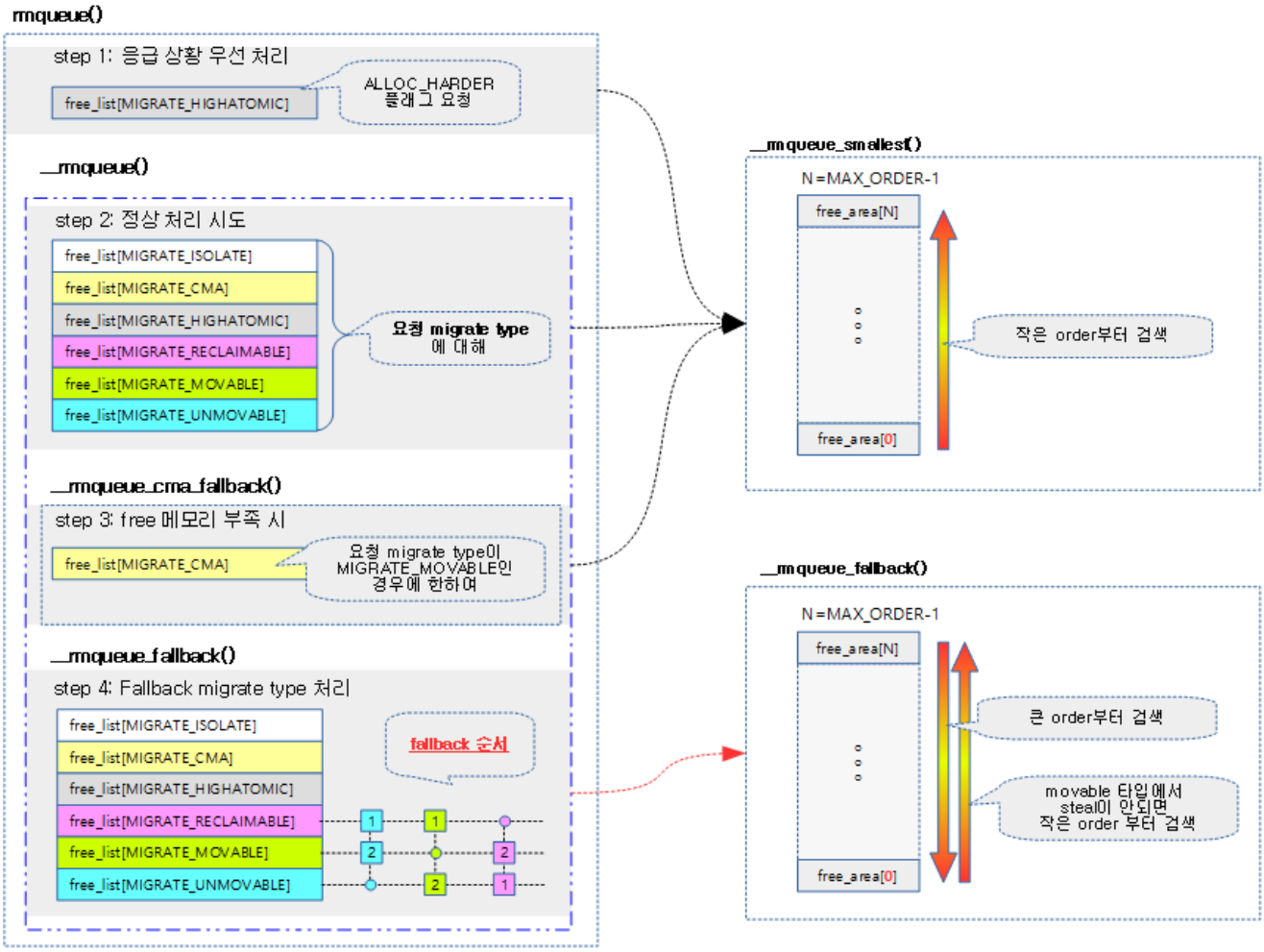
```
1  /*
2   * Do the hard work of removing an element from the buddy allocator.
3   * Call me with the zone->lock already held.
4   */
```

```
01 static __always_inline struct page *
02 __rmqueue(struct zone *zone, unsigned int order, int migratetype,
03                                                 unsigned int alloc_flag
   s)
04 {
05         struct page *page;
06
07 retry:
08         page = __rmqueue_smallest(zone, order, migratetype);
09         if (unlikely(!page)) {
10                 if (migratetype == MIGRATE_MOVABLE)
11                         page = __rmqueue_cma_fallback(zone, order);
12
13                 if (!page && __rmqueue_fallback(zone, order, migratetyp
   e,
14                                                                 alloc_fl
   ags))
15                         goto retry;
16         }
17
18         trace_mm_page_alloc_zone_locked(page, order, migratetype);
19         return page;
20 }
```

Assign @order pages of @migratype through the buddy system in the request zone. Returns the assigned page descriptor on success, and null if it fails.

- In line 8 of the code, search for and assign the order from the first requested order to the maximum order, i.e., starting with the small order.
- If page allocation fails in code lines 9~11, if it is a movable type request, search in the cma space first.
- If page allocation still fails in lines 13~15 of the code, search for the fallback types and migrate them to the desired migrate type. If the migration is successful, retry the allocation.

The following figure shows that when a free page is allocated with the requested migrate type and order from the buddy system, if the allocation fails, the search is continued using the migrate type fallback list.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/rmqueue-2.png)

__rmqueue_smallest()

mm/page_alloc.c

```
1  /*
2   * Go through the free lists for the given migratetype and remove
3   * the smallest available page from the freelists
4   */
```

```
01 static __always_inline
```

```
02 struct page *__rmqueue_smallest(struct zone *zone, unsigned int order,
03                                                 int migratetype)
04 {
05         unsigned int current_order;
06         struct free_area *area;
07         struct page *page;
08
09         /* Find a page of the appropriate size in the preferred list */
10         for (current_order = order; current_order < MAX_ORDER; ++current
   _order) {
11                 area = &(zone->free_area[current_order]);
12                 page = list_first_entry_or_null(&area->free_list[migrate
   type],
13                                                         struct page, lr
   u);
14                 if (!page)
15                         continue;
16                 list_del(&page->lru);
17                 rmv_page_order(page);
18                 area->nr_free--;
19                 expand(zone, page, order, current_order, area, migratety
   pe);
20                 set_pcppage_migratetype(page, migratetype);
21                 return page;
22         }
23
24         return NULL;
25 }
```

Returns the free page found by searching for the requested @order to the top order, i.e., from the small order to the free_list with the requested @migratetype.

- From line 10~15 of the code, go through the free_list of the request @order to the top order@migratetype and find the free page.
- In line 16 of code, remove the hot page found in the free_list.
- In line 17 of code, reset the order information of the page to be assigned to 0 and remove the buddy identification flag.
- Code Rangney Decreases the nr_free of the corresponding order slot in 18.
- In line 19 of the code, if the request order doesn't have a free page, take a large order and expand it.
    - Unless you're using a guard page, you can see how it is added by decomposing half of the requested order under a large order.
    - e.g. order=3, current_order=6
        - Trim order=6 and add one to order 5, order 4, and order 3, and return the remaining order 3 pages.
- Record the migrate type on line 20 of the code.

Expand large order pages (decomposition)

expand()

```
01 /*
02  * The order of subdivision here is critical for the IO subsystem.
03  * Please do not alter this order without good reasons and regression
04  * testing. Specifically, as large blocks of memory are subdivided,
05  * the order in which smaller blocks are delivered depends on the order
06  * they're subdivided in this function. This is the primary factor
```

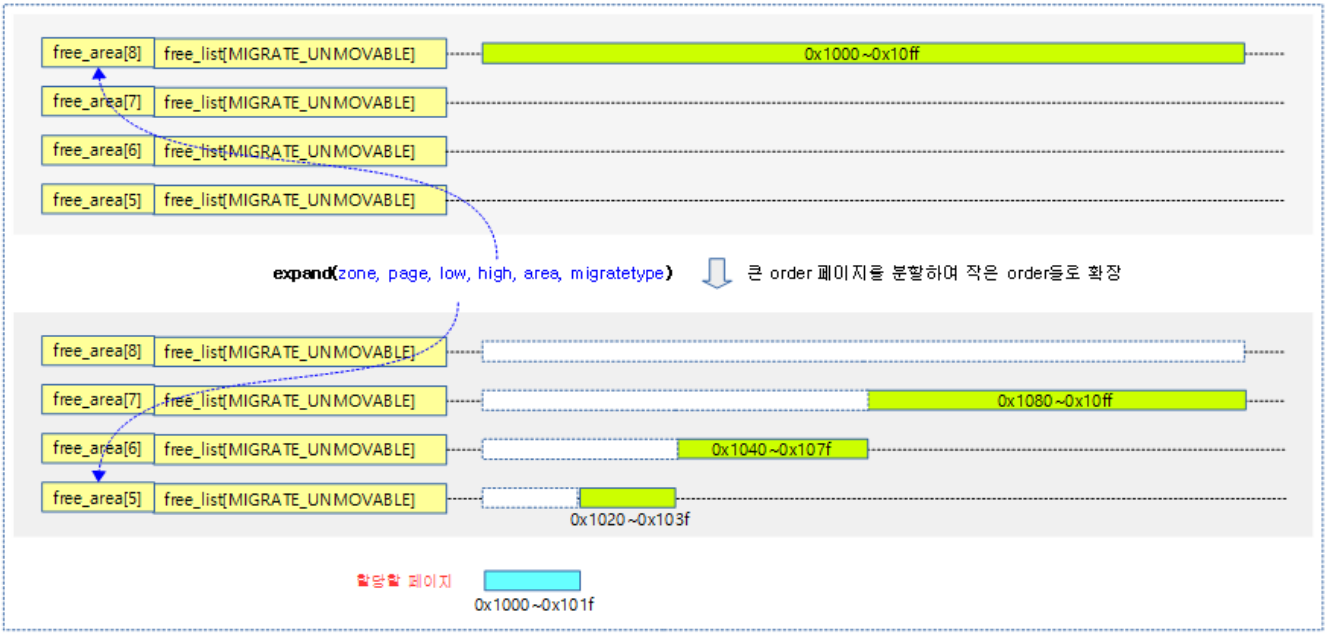
```
 * influencing the order in which pages are delivered to the IO
 * subsystem according to empirical testing, and this is also justified
 * by considering the behavior of a buddy system containing a single
 * large block of memory acted on by a series of small allocations.
 * This behavior is a critical factor in sglist merging's success.
 *
 * -- nyc
 */
```

```
01 static inline void expand(struct zone *zone, struct page *page,
02         int low, int high, struct free_area *area,
03         int migratetype)
04 {
05         unsigned long size = 1 << high;
06
07         while (high > low) {
08                 area--;
09                 high--;
10                 size >>= 1;
11                 VM_BUG_ON_PAGE(bad_range(zone, &page[size]), &page[siz
   e]);
12
13                 if (IS_ENABLED(CONFIG_DEBUG_PAGEALLOC) &&
14                         debug_guardpage_enabled() &&
15                         high < debug_guardpage_minorder()) {
16                         /*
17                          * Mark as guard pages (or page), that will allo
   w to
18                          * merge back to allocator when buddy will be fr
   eed.
19                          * Corresponding page table entries will not be
   touched,
20                          * pages will stay not present in virtual addres
   s space
21                          */
22                         set_page_guard(zone, &page[size], high, migratet
   ype);
23                         continue;
24                 }
25                 list_add(&page[size].lru, &area->free_list[migratetyp
   e]);
26                 area->nr_free++;
27                 set_page_order(&page[size], high);
28         }
29 }
```

If a page is assigned from a @high order that is larger than the requested @low order, expand (decompose) and register a free page from high-1 to low.

- In line 5 of the code, we calculate the number of @high order pages in advance.
- In code lines 7~10, if the @high order is larger than the @low order, the area and high are decremented, and the size is halved.
- If you use the guard page for debug in line 13~24 of the code, skip less than half of the max order (0~5 order).
- In line 25, add page[size] to area->free_list[migratetype].
- On line 26 of code, increase the number of free entries in the area.
- Store the order value on the page you added in line 27 of the code.

The following figure expands the order 8 page and adds the remaining pages to the free_list corresponding to order 5 ~ 7 in addition to the order 5 pages that were requested to be assigned.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/expand-1b.png)

Fallback when migrate type is insufficient

Using CMA Zones

__rmqueue_cma_fallback()

mm/page_alloc.c

```
1 static __always_inline struct page *__rmqueue_cma_fallback(struct zone *
  zone,
2                                         unsigned int order)
3 {
4         return __rmqueue_smallest(zone, order, MIGRATE_CMA);
5 }
```

If a search fails on that migrate type when requesting a movable type, it will also try in the cma area.

Using the Fallback Migrate Type

__rmqueue_fallback()

mm/page_alloc.c

```
01 /*
02  * Try finding a free buddy page on the fallback list and put it on the
  free
03  * list of requested migratetype, possibly along with other pages from t
  he same
04  * block, depending on fragmentation avoidance heuristics. Returns true
  if
05  * fallback was found so that __rmqueue_smallest() can grab it.
06  *
07  * The use of signed ints for order and current_order is a deliberate
08  * deviation from the rest of this file, to make the for loop
09  * condition simpler.
10  */
```

```
01 static __always_inline bool
02 __rmqueue_fallback(struct zone *zone, int order, int start_migratetype,
03                                                 unsigned int alloc_flag
   s)
04 {
05         struct free_area *area;
06         int current_order;
07         int min_order = order;
08         struct page *page;
09         int fallback_mt;
10         bool can_steal;
11
12         /*
13          * Do not steal pages from freelists belonging to other pagebloc
   ks
14          * i.e. orders < pageblock_order. If there are no local zones fr
   ee,
15          * the zonelists will be reiterated without ALLOC_NOFRAGMENT.
16          */
17         if (alloc_flags & ALLOC_NOFRAGMENT)
18                 min_order = pageblock_order;
19
20         /*
21          * Find the largest available free page in the other list. This
   roughly
22          * approximates finding the pageblock with the most free pages,
   which
23          * would be too costly to do exactly.
24          */
25         for (current_order = MAX_ORDER - 1; current_order >= min_order;
26                                 --current_order) {
27                 area = &(zone->free_area[current_order]);
28                 fallback_mt = find_suitable_fallback(area, current_orde
   r,
29                                 start_migratetype, false, &can_steal);
30                 if (fallback_mt == -1)
31                         continue;
32
33                 /*
34                  * We cannot steal all free pages from the pageblock and
   the
35                  * requested migratetype is movable. In that case it's b
   etter to
36                  * steal and split the smallest available page instead o
   f the
37                  * largest available page, because even if the next mova
   ble
38                  * allocation falls back into a different pageblock than
   this
39                  * one, it won't cause permanent fragmentation.
40                  */
41                 if (!can_steal && start_migratetype == MIGRATE_MOVABLE
42                                         && current_order > order)
43                         goto find_smallest;
44
45                 goto do_steal;
46         }
47
48         return false;
49
50 find_smallest:
51         for (current_order = order; current_order < MAX_ORDER;
52                                                         current_order++)
   {
53                 area = &(zone->free_area[current_order]);
54                 fallback_mt = find_suitable_fallback(area, current_orde
   r,
55                                 start_migratetype, false, &can_steal);
```

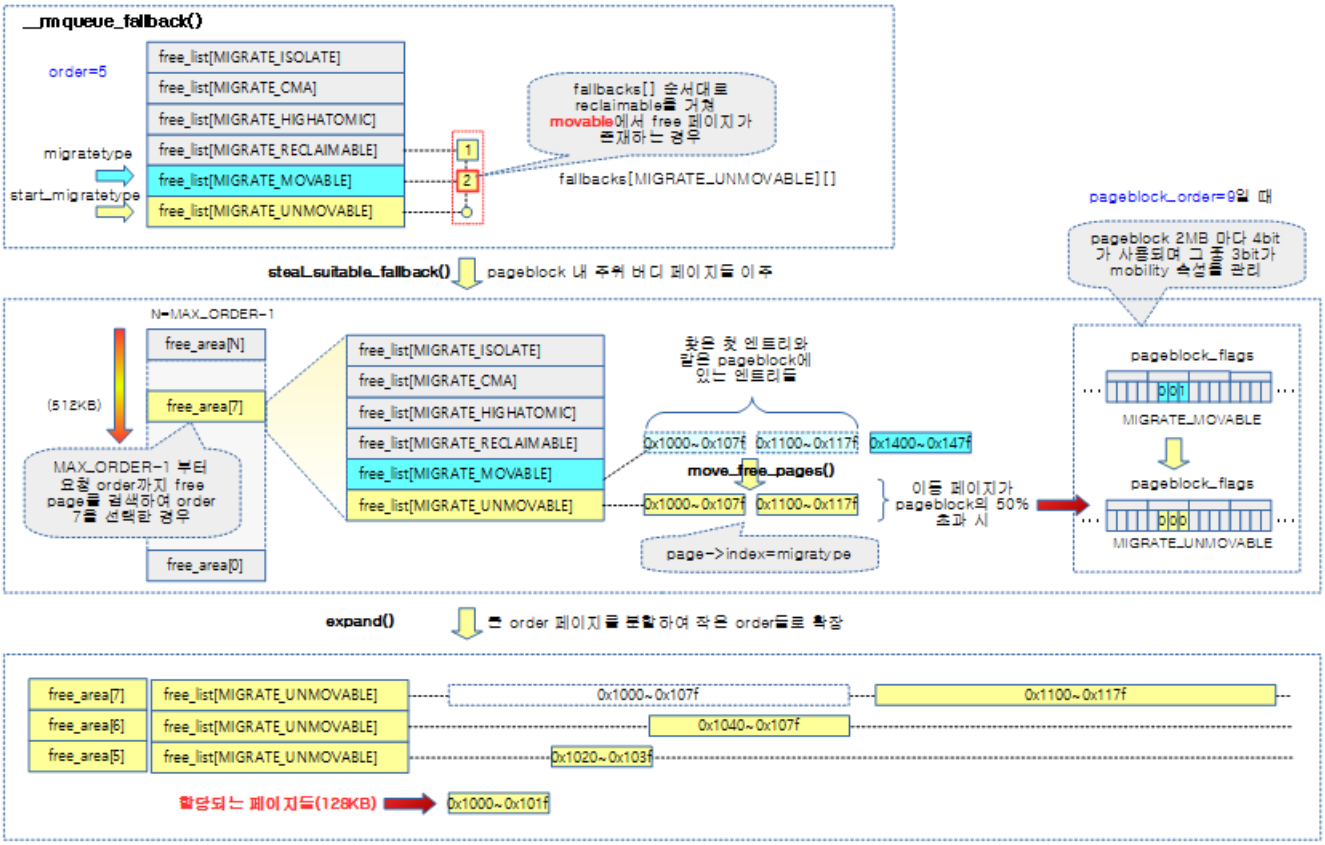
```
56                  if (fallback_mt != -1)
57                          break;
58          }
59
60          /*
61           * This should not happen - we already found a suitable fallback
62           * when looking for the largest page.
63           */
64          VM_BUG_ON(current_order == MAX_ORDER);
65
66  do_steal:
67          page = list_first_entry(&area->free_list[fallback_mt],
68                                                  struct page, lr
    u);
69
70          steal_suitable_fallback(zone, page, alloc_flags, start_migratety
    pe,
71                                                          can_stea
    l);
72
73          trace_mm_page_alloc_extfrag(page, order, current_order,
74                  start_migratetype, fallback_mt);
75
76          return true;
77
78  }
```

Called when page allocation fails on the requested migrate type, the next migrate type is taken away by the order of the migrate type fallback. (steal). When we take other migrate types, we focus on the largest order. This large page can be retrieved so that the next time a request is made for the same migrate type, it will be immediately responded.

- If there is a ALLOC_NOFRAGMENT request in line 17~18 of the code, assign the page block order to the min_order value to prevent the migrate type from being mixed in the page block.
- In code lines 25~31, it traverses the buddy list backwards from the largest order to min_order and determines whether there is a free page in the order of the fallback migrate type.
- If the type requested in lines 41~45 is movable, and the output result of can_steal is false, it will be moved to the find_smallest label. If not, go to do_steal Labels.
- If the page is not found even by the fallback migrate type until the loop is completed at line 48, it returns false.
- In code lines 50~58, find_smallest: Label. Since the requested migrate type is movable, even if the migrate type is shuffled in the page block, it will escape the loop if there is a free page in the fallback migrate type, starting from the requested order to the largest order, i.e., starting with the small order order.
- In code lines 66~76, do_steal: Label. The found Fallback Migrate type free page is retrieved from the requested Migrate type free_list. It then returns true.

The figure below shows that if you want to get a 5-order page of the unmovable type, but you can't get a free page from the free_list of the unmovable type, you can search for the free page of the free_list in the order of the Fallback Migrate type and steal it.

- Search for the free_list of the Reclaimable Migrate type, which is the first Fallback Migrate type for Unmovable, and then the free_list of the second Movable Migrate type.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/rmqueue_fallback-1b.png)

Find the right fallback migrate type

find_suitable_fallback()

mm/page_alloc.c

```
1  /*
2   * Check whether there is a suitable fallback freepage with requested or
   der.
3   * If only_stealable is true, this function returns fallback_mt only if
4   * we can steal other freepages all together. This would help to reduce
5   * fragmentation due to mixed migratetype pages in one pageblock.
6   */
```

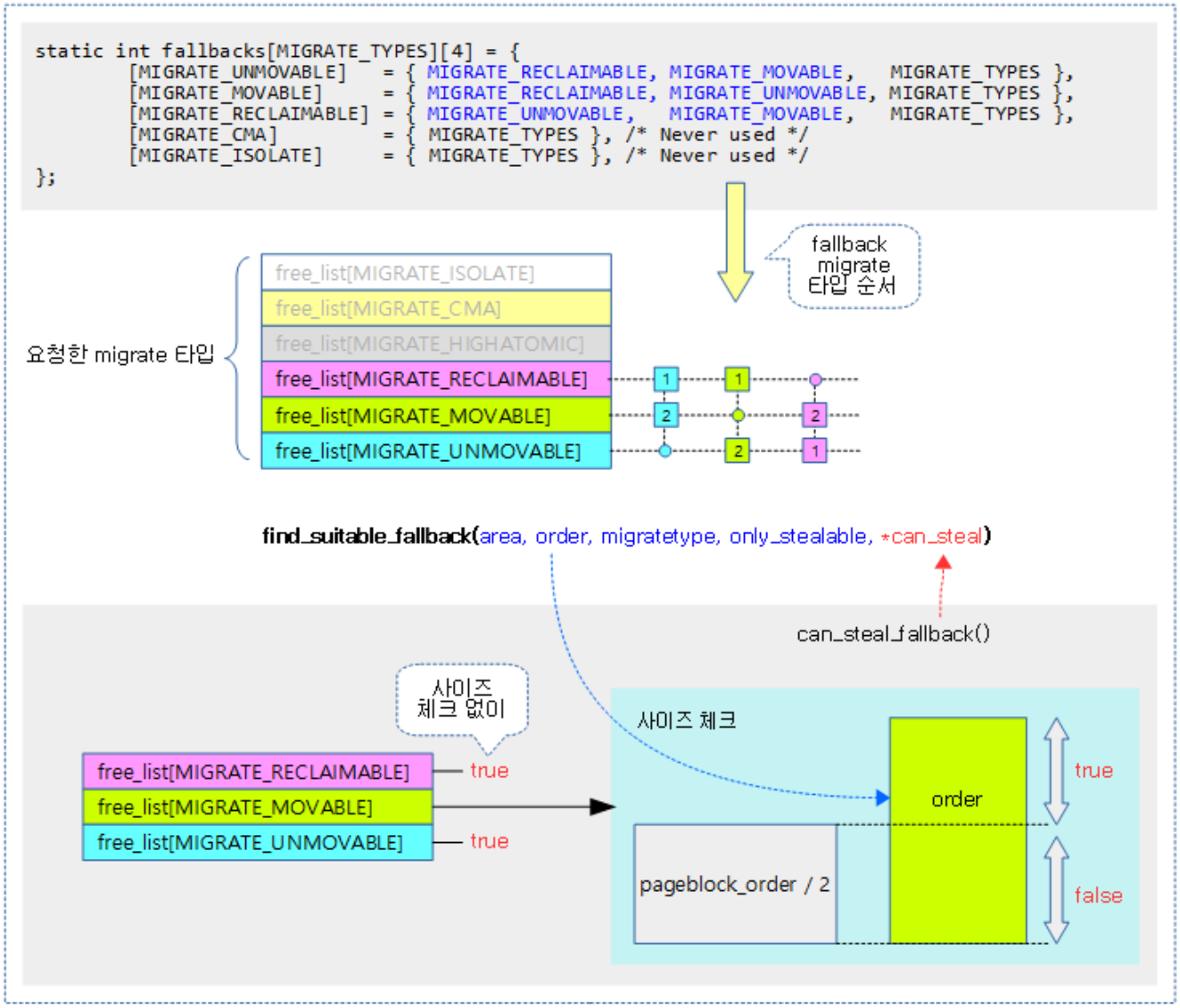
```
01 int find_suitable_fallback(struct free_area *area, unsigned int order,
02                       int migratetype, bool only_stealable, bool *can_
   steal)
03 {
04         int i;
05         int fallback_mt;
06
07         if (area->nr_free == 0)
08                 return -1;
09
10         *can_steal = false;
11         for (i = 0;; i++) {
12                 fallback_mt = fallbacks[migratetype][i];
13                 if (fallback_mt == MIGRATE_TYPES)
14                         break;
15
16                 if (list_empty(&area->free_list[fallback_mt]))
17                         continue;
18
19                 if (can_steal_fallback(order, migratetype))
20                         *can_steal = true;
```

```
21
22                  if (!only_stealable)
23                          return fallback_mt;
24
25                  if (*can_steal)
26                          return fallback_mt;
27          }
28
29          return -1;
30  }
```

free_list the order page uses the fallback migrate type, it knows which fallback migrate type it can steal. The fallback migrate type also returns -1 if it is no longer found. If you use true for @only_stealable, you must steal the entire 1-page block. If the output argument @can_steal is true, it tells it that all the other free pages in the page block it belongs to are stealable, and if it is false, it tells the page that it is stealable.

- In line 7~8 of the code, if there are no free entries in the order, it returns -1.
- Travers the Fallback Migrate type of the Migrate type requested in code lines 11~17 in the order of the Migrate type.
- In line 19~26 of the code, it determines whether or not all 1 page blocks should be stolen.
  - If it is possible to steal an entire block of a page, substitute true for the output argument *can_steal and return the corresponding migrate type that is traversing.
  - If the only_stealable is false, it will no longer traverse and return the corresponding migrate type, regardless of whether it is *can_stean or not.
- On line 29 of code, returns -1 if there is no stealable migrate type to complete the traversal.

The following figure shows the process of knowing which migrate type to use with fallback.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/find_suitable_fallback-1.png)

can_steal_fallback()

mm/page_alloc.c

```
01 /*
02  * When we are falling back to another migratetype during allocation, tr
   y to
03  * steal extra free pages from the same pageblocks to satisfy further
04  * allocations, instead of polluting multiple pageblocks.
05  *
06  * If we are stealing a relatively large buddy page, it is likely there
   will
07  * be more free pages in the pageblock, so try to steal them all. For
08  * reclaimable and unmovable allocations, we steal regardless of page si
   ze,
09  * as fragmentation caused by those allocations polluting movable pagebl
   ocks
10  * is worse than movable allocations stealing from unmovable and reclaim
   able
11  * pageblocks.
12  */
```

```
01 static bool can_steal_fallback(unsigned int order, int start_mt)
02 {
03         /*
04          * Leaving this order check is intended, although there is
```

```
         * relaxed order check in next check. The reason is that
         * we can actually steal whole pageblock if this condition met,
         * but, below check doesn't guarantee it and that is just heuristic
         * so could be changed anytime.
         */
        if (order >= pageblock_order)
                return true;

        if (order >= pageblock_order / 2 ||
                start_mt == MIGRATE_RECLAIMABLE ||
                start_mt == MIGRATE_UNMOVABLE ||
                page_group_by_mobility_disabled)
                return true;

        return false;
}
```

Determine whether or not a one-page block should be stolen. If it is not possible to assign from the freelist of the desired migratetype, use fallback migratetype. If you use this fallback allocation, you can safely perform additional allocation in the future by stealing all the remaining free pages in the same page block of the page to be stolen, thus preventing the contamination of the pageblocks (many page blocks spread out in multiple places as unmovable). In this way, the decision on whether to steal all the free pages in the one-page block is as follows.

- Somewhat large order request
  - When using the huge page used by x86, arm, arm64 systems, etc., the pageblock_order is usually 9, so half of it is 4.5, which is a 4.<> or more decimal point.
- If the direction of the reclaimable or unmovable type is the dest, this type of contamination must be prevented.
- There is too little main memory to manage the freelist by dividing it by migratetype.

Stealing from the Fallback Migrate type

steal_suitable_fallback()

mm/page_alloc.c

```
/*
 * This function implements actual steal behaviour. If order is large enough,
 * we can steal whole pageblock. If not, we first move freepages in this
 * pageblock to our migratetype and determine how many already-allocated pages
 * are there in the pageblock with a compatible migratetype. If at least half
 * of pages are free or compatible, we can change migratetype of the pageblock
 * itself, so pages freed in the future will be put on the correct free list.
 */
```

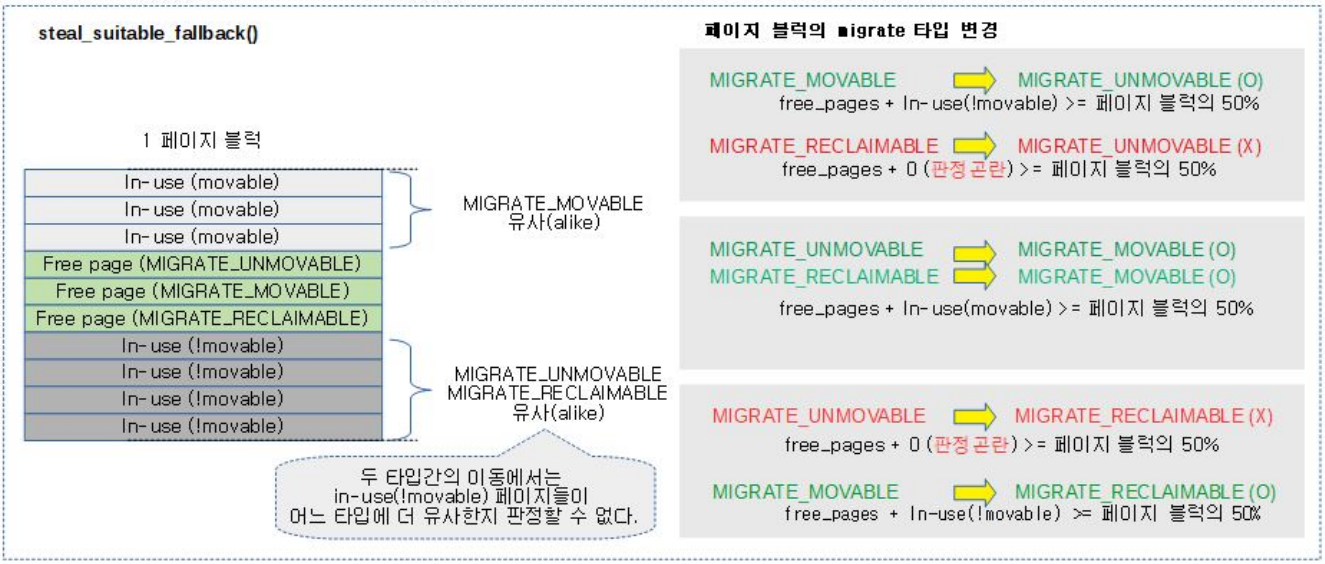
```
static void steal_suitable_fallback(struct zone *zone, struct page *page,
                unsigned int alloc_flags, int start_type, bool whole_block)
{
        unsigned int current_order = page_order(page);
        struct free_area *area;
```

```
06          int free_pages, movable_pages, alike_pages;
07          int old_block_type;
08
09          old_block_type = get_pageblock_migratetype(page);
10
11          /*
12           * This can happen due to races and we want to prevent broken
13           * highatomic accounting.
14           */
15          if (is_migrate_highatomic(old_block_type))
16                  goto single_page;
17
18          /* Take ownership for orders >= pageblock_order */
19          if (current_order >= pageblock_order) {
20                  change_pageblock_range(page, current_order, start_type);
21                  goto single_page;
22          }
23
24          /*
25           * Boost watermarks to increase reclaim pressure to reduce the
26           * likelihood of future fallbacks. Wake kswapd now as the node
27           * may be balanced overall and kswapd will not wake naturally.
28           */
29          boost_watermark(zone);
30          if (alloc_flags & ALLOC_KSWAPD)
31                  set_bit(ZONE_BOOSTED_WATERMARK, &zone->flags);
32
33          /* We are not allowed to try stealing from the whole block */
34          if (!whole_block)
35                  goto single_page;
36
37          free_pages = move_freepages_block(zone, page, start_type,
38                                                  &movable_pages);
39          /*
40           * Determine how many pages are compatible with our allocation.
41           * For movable allocation, it's the number of movable pages whic
    h
42           * we just obtained. For other types it's a bit more tricky.
43           */
44          if (start_type == MIGRATE_MOVABLE) {
45                  alike_pages = movable_pages;
46          } else {
47                  /*
48                   * If we are falling back a RECLAIMABLE or UNMOVABLE all
    ocation
49                   * to MOVABLE pageblock, consider all non-movable pages
    as
50                   * compatible. If it's UNMOVABLE falling back to RECLAIM
    ABLE or
51                   * vice versa, be conservative since we can't distinguis
    h the
52                   * exact migratetype of non-movable pages.
53                   */
54                  if (old_block_type == MIGRATE_MOVABLE)
55                          alike_pages = pageblock_nr_pages
56                                                  - (free_pages + movable_
    pages);
57                  else
58                          alike_pages = 0;
59          }
60
61          /* moving whole block can fail due to zone boundary conditions
    */
62          if (!free_pages)
63                  goto single_page;
64
65          /*
```

```
66            * If a sufficient number of pages in the block are either free
    or of
67            * comparable migratability as our allocation, claim the whole b
    lock.
68            */
69           if (free_pages + alike_pages >= (1 << (pageblock_order-1)) ||
70                           page_group_by_mobility_disabled)
71                   set_pageblock_migratetype(page, start_type);
72
73           return;
74
75   single_page:
76           area = &zone->free_area[current_order];
77           list_move(&page->lru, &area->free_list[start_type]);
78   }
```

- In line 9 of code, we know the type of migrate before the change in the page block to which the current page belongs.
- In code lines 15~16, if it is of highatomic type, go to the single_page label.
- If the order request is larger than the page block in line 19~22 of the code, the migrate type will not be mixed in the page blocks. Therefore, go to the single_page label.
- In line 29~31 of the code, if a fallback order smaller than the page block is executed, the watermark boost with the watermark_boost_factor percentage (default=15000, 150%) is applied, and if there is a ALLOC_KSWAPD request, the boost watermark flag of the zone is set.
  - 참고: mm: reclaim small amounts of memory when an external fragmentation event occurs (https://github.com/torvalds/linux/commit/1c30844d2dfe272d58c8fc000960b835d13aa2ac)
- If there is no @whole_block request in line 34~35 of the code, it will go to the single_page label to handle only that page.
- In lines 37~38 of code, move the migrate type of the buddy system to @start_type with all the free pages in the page block containing that page.
- In code lines 44~59, calculate the compatible migrate type pages for each request migrate type as follows and assign them to the alike_pages.
  - Pages in use are recognized as similar pages (alike_pages) that are compatible with each type.
  - However, the pages in use cannot be distinguished by exactly three migratetypes, only movable and !movable.
  - Therefore, migrations between MIGRATE_UNMOVABLE and MIGRATE_RECLAIMABLE cannot be carried out on the alike page.
- In line 62~63 of the code, if there is no free page, go to the single_page label.
- In line 69~71 of the code, if more than half of the page block is free with compatible pages, change the migrate type of the page block.
- In code lines 75~77, single_page: Label. Move only that page to the @start_type of the free_list.

The following figure shows how the steal_suitable_fallback() function works by determining whether to steal a page block or just the page.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/steal_suitable_fallback-1a.jpg)

change_pageblock_range()

mm/page_alloc.c

```
01 static void change_pageblock_range(struct page *pageblock_page,
02                                     int start_order, int migratetyp
   e)
03 {
04         int nr_pageblocks = 1 << (start_order - pageblock_order);
05
06         while (nr_pageblocks--) {
07                 set_pageblock_migratetype(pageblock_page, migratetype);
08                 pageblock_page += pageblock_nr_pages;
09         }
10 }
```

Set a migratetype for each page block equal to the number of all page blocks within the requested order.

- In line 4 of code, calculate the number of pageblocks that can fit in a start_order.
- In line 6~9 of the code, traverse the number of page blocks and set the migrate type of the page block.

Move the migrate type of all free pages in the page block

move_freepages_block()

mm/page_alloc.c

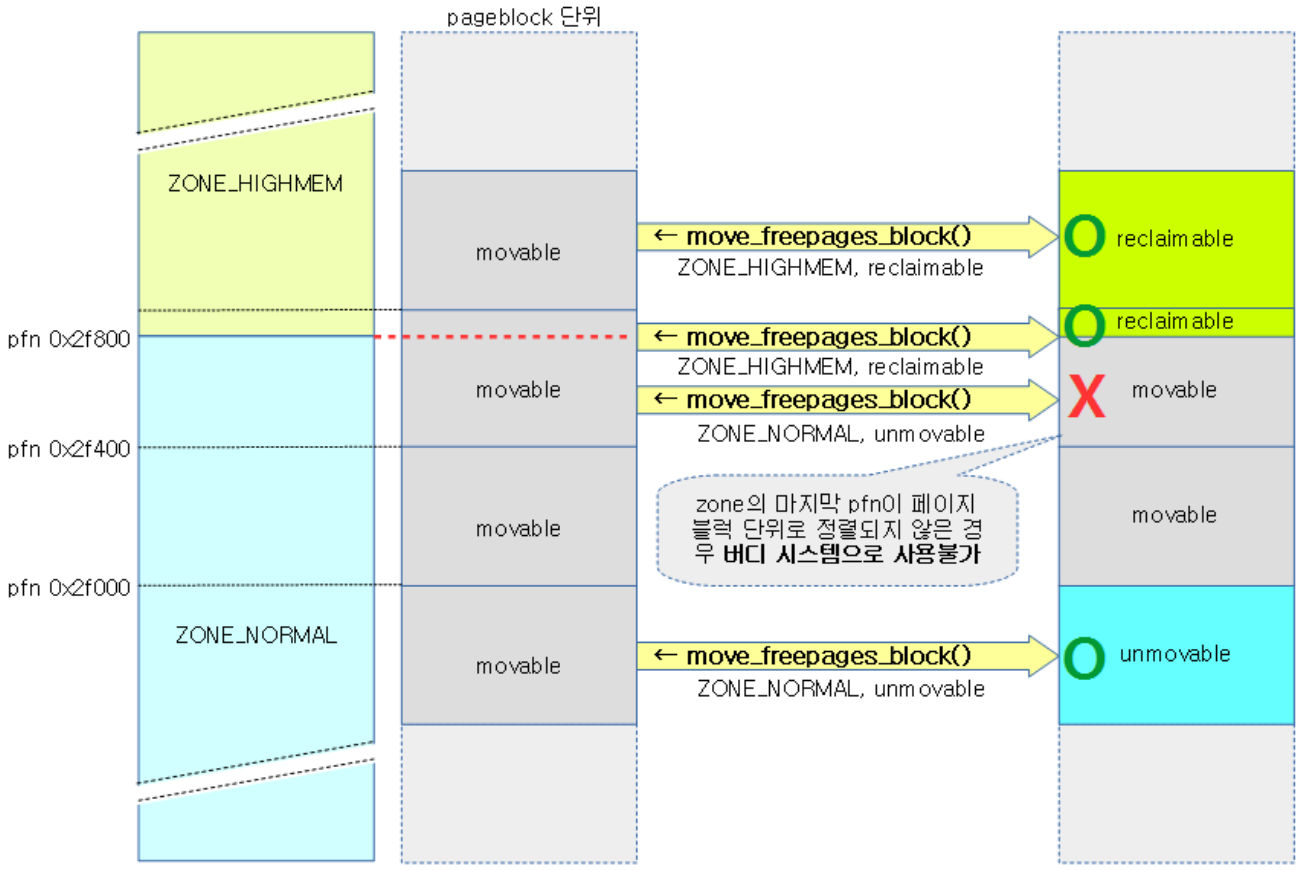
```
01 int move_freepages_block(struct zone *zone, struct page *page,
02                                 int migratetype)
03 {
04         unsigned long start_pfn, end_pfn;
05         struct page *start_page, *end_page;
06
07         start_pfn = page_to_pfn(page);
08         start_pfn = start_pfn & ~(pageblock_nr_pages-1);
09         start_page = pfn_to_page(start_pfn);
10         end_page = start_page + pageblock_nr_pages - 1;
11         end_pfn = start_pfn + pageblock_nr_pages - 1;
```

```
12
13          /* Do not cross zone boundaries */
14          if (!zone_spans_pfn(zone, start_pfn))
15                  start_page = page;
16          if (!zone_spans_pfn(zone, end_pfn))
17                  return 0;
18
19          return move_freepages(zone, start_page, end_page, migratetype);
20  }
```

Change all free pages in the pageblock with the requested pages in the specified zone to the request migrate type, move them to the request migrate type in the buddy system, and return the number of pages moved. However, if the end of the page block is partial, it cannot be moved due to zone boundaries.

- Even if the start page of the request zone is not sorted by pageblock, it can be moved from the start address of the zone (partial)
- If the end page of the request zone is not aligned by pageblock, that pageblock cannot be used.

The figure below shows changing the migrate type of the free pages in the page block of the specified zone to the requested type.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/move_freepages_block-1.png)

zone_spans_pfn()

include/linux/mmzone.h

```
1  static inline bool zone_spans_pfn(const struct zone *zone, unsigned long
   pfn)
2  {
```

```
        return zone->zone_start_pfn <= pfn && pfn < zone_end_pfn(zone);
}
```

Returns whether the pfn is in the zone of the zone.

move_freepages()

mm/page_alloc.c

```
/*
 * Move the free pages in a range to the free lists of the requested type.
 * Note that start_page and end_pages are not aligned on a pageblock
 * boundary. If alignment is required, use move_freepages_block()
 */
```

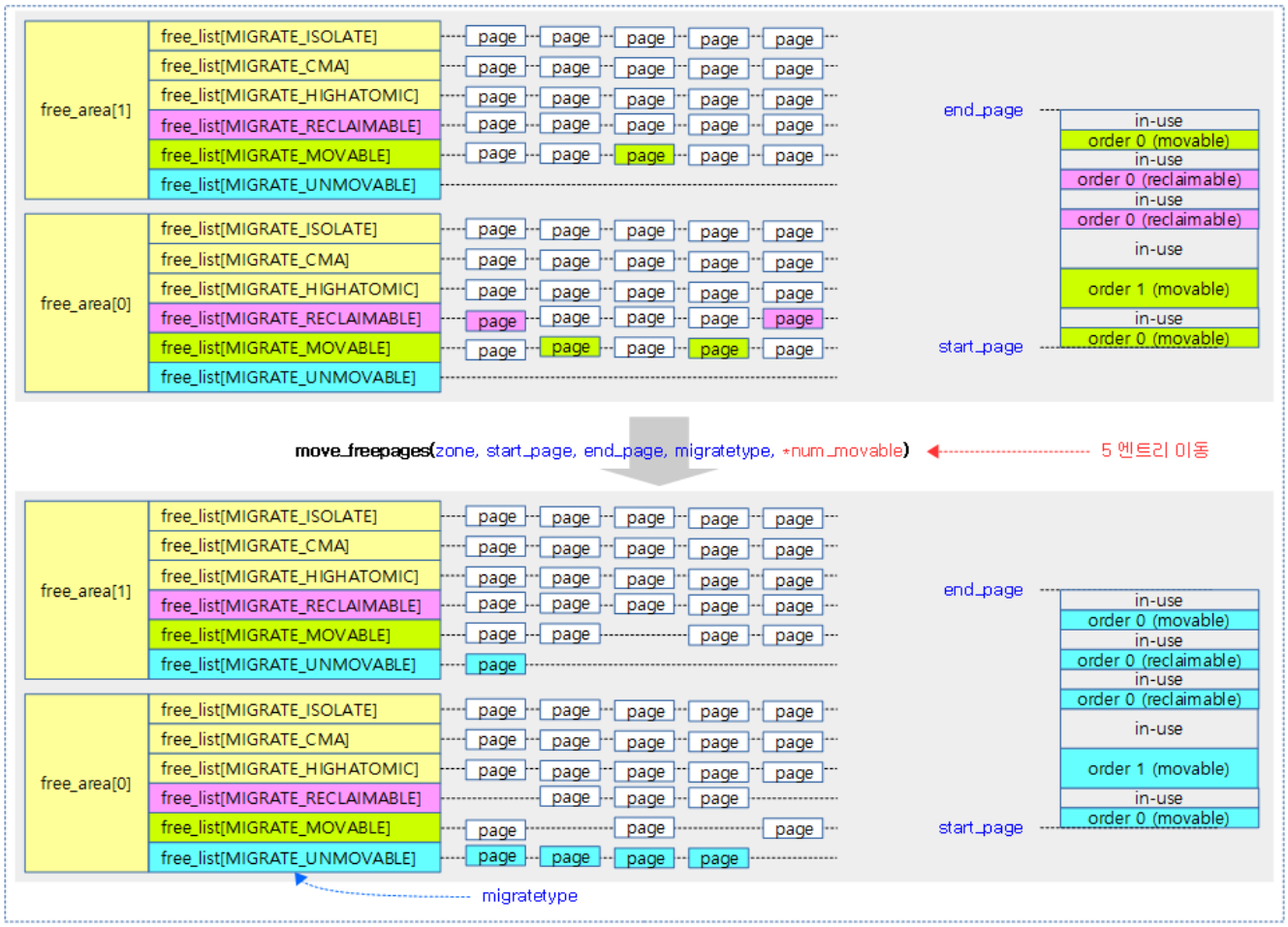
```
static int move_freepages(struct zone *zone,
                          struct page *start_page, struct page *end_page,
                          int migratetype, int *num_movable)
{
        struct page *page;
        unsigned int order;
        int pages_moved = 0;

#ifndef CONFIG_HOLES_IN_ZONE
        /*
         * page_zone is not safe to call in this context when
         * CONFIG_HOLES_IN_ZONE is set. This bug check is probably redundant
         * anyway as we check zone boundaries in move_freepages_block().
         * Remove at a later date when no bug reports exist related to
         * grouping pages by mobility
         */
        VM_BUG_ON(pfn_valid(page_to_pfn(start_page)) &&
                  pfn_valid(page_to_pfn(end_page)) &&
                  page_zone(start_page) != page_zone(end_page));
#endif
        for (page = start_page; page <= end_page;) {
                if (!pfn_valid_within(page_to_pfn(page))) {
                        page++;
                        continue;
                }

                /* Make sure we are not inadvertently changing nodes */
                VM_BUG_ON_PAGE(page_to_nid(page) != zone_to_nid(zone), page);

                if (!PageBuddy(page)) {
                        /*
                         * We assume that pages that could be isolated for
                         * migration are movable. But we don't actually try
                         * isolating, as that would be expensive.
                         */
                        if (num_movable &&
                                        (PageLRU(page) || __PageMovable(page)))
                                (*num_movable)++;

                        page++;
                        continue;
                }

                order = page_order(page);
```

```
45                  list_move(&page->lru,
46                            &zone->free_area[order].free_list[migratetyp
    e]);
47                  page += 1 << order;
48                  pages_moved += 1 << order;
49          }
50
51          return pages_moved;
52  }
```

For all free pages from the start page to the end page of the request zone, the migrate type is moved, and the number of pages moved is returned.

- In lines 21~25 of the code, traverse from the start page to the end page, and skip if the page is in the hole area.
- If the page is free from the buddy system in line 30~42 and is not managed, skip it.
  - If the page you are using is included in LRU Movable or non-LRU Movable, increment the output factor *num_movable.
- In code lines 44~48, move the specified migratetype of the free_list corresponding to the current order.
- Returns the number of pages moved from line 51 of the code.

The following figure shows how the pages in the request scope are found and migrated to the specified migratetype of the buddy system.



(http://jake.dothome.co.kr/wp-content/uploads/2016/05/move_freepages-1a.png)

consultation

- Zoned Allocator -1- (Physics Page Assignment - Fastpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath) | Qc
- Zoned Allocator -2- (Physics Page Assignment - Slowpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-slowpath) | Qc
- Zoned Allocator -3- (Buddy Page Allocation) (http://jake.dothome.co.kr/buddy-alloc) | Sentence C – Current post
- Zoned Allocator -4- (Buddy Page Terminated) (http://jake.dothome.co.kr/buddy-free/) | Qc
- Zoned Allocator -5- (Per-CPU Page Frame Cache) (http://jake.dothome.co.kr/per-cpu-page-frame-cache) | 문c
- Zoned Allocator -6- (Watermark) (http://jake.dothome.co.kr/zonned-allocator-watermark) | 문c
- Zoned Allocator -7- (Direct Compact) (http://jake.dothome.co.kr/zonned-allocator-compaction) | 문c
- Zoned Allocator -8- (Direct Compact-Isolation) (http://jake.dothome.co.kr/zonned-allocator-isolation) | 문c
- Zoned Allocator -9- (Direct Compact-Migration) (http://jake.dothome.co.kr/zonned-allocator-migration) | 문c
- Zoned Allocator -10- (LRU & pagevec) (http://jake.dothome.co.kr/lru-lists-pagevecs) | 문c
- Zoned Allocator -11- (Direct Reclaim) (http://jake.dothome.co.kr/zonned-allocator-reclaim) | 문c
- Zoned Allocator -12- (Direct Reclaim-Shrink-1) (http://jake.dothome.co.kr/zonned-allocator-shrink-1) | 문c
- Zoned Allocator -13- (Direct Reclaim-Shrink-2) (http://jake.dothome.co.kr/zonned-allocator-shrink-2) | 문c
- Zoned Allocator -14- (Kswapd) (http://jake.dothome.co.kr/zonned-allocator-kswapd) | 문c

---

7 thoughts to "Zonned Allocator -3- (Buddy Page Allocation)"

HOYOUNG LEE
2019-03-14 11:49 (http://jake.dothome.co.kr/buddy-alloc/#comment-204472)

Great looking at your blog.
While watching the slab allocator and buddy memory allocator, I have a question, so I leave a comment like this.
I wonder if the slab allocator also has a cleaning function inside the page (in the case of a partial list, when the objects are distributed in several partial slabs and not used efficiently, instead of using all the objects inside each slab and not going to the full slab, there is a cleaning or compaction function for the problem of having a lot of partsials).
I appreciate it!! I'm always looking at your posts

RESPONSE (/BUDDY-ALLOC/?REPLYTOCOM=204472#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2019-03-14 12:45 (http://jake.dothome.co.kr/buddy-alloc/#comment-204478)

Hello?

When you create a KMEM cache, the internal management is always working with a minimum slub ~ maximum slub range so that there are not too many partials. (/sys/kernel/slab, < cache name > directory to determine the status of each cache.)

For example, if a partial range is specified as 10 ~ 30 parts, but the current partial slub is 30, and if it is increased by 1 more, it will be 31 partsial.
If there are less than 10, it will be refilled to make it 10, and if there are more than 30, the logic will only free the slub if there is a slub in the slub where all the objects are empty.

Apparently, the operation is different from the compaction and migration of the buddy system.

I appreciate it. Moon Young-il.

RESPONSE (/BUDDY-ALLOC/?REPLYTOCOM=204478#RESPOND)

HOYOUNG LEE
2019-03-14 17:22 (http://jake.dothome.co.kr/buddy-alloc/#comment-204518)

Oh thank you so much 😵 it's solved!!

RESPONSE (/BUDDY-ALLOC/?REPLYTOCOM=204518#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2019-03-15 16:32 (http://jake.dothome.co.kr/buddy-alloc/#comment-204738)

It's up to you to solve it. Keep up the good luck. ^^

RESPONSE (/BUDDY-ALLOC/?REPLYTOCOM=204738#RESPOND)

정광민
2019-06-08 15:42 (http://jake.dothome.co.kr/buddy-alloc/#comment-214257)

안녕하세요. 블로그 보면 항상 이미지로 정리가 매우 잘 되어 있는데요, 어떤 툴 사용하시는지 알 수 있을까요?
다이어그램을 그리는 노하우가 있으신지요? 항상 처음부터 그리는지 아니면 템플릿같은걸 사용하시는지도 궁금합니다.

응답 (/BUDDY-ALLOC/?REPLYTOCOM=214257#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2019-06-08 17:45 (http://jake.dothome.co.kr/buddy-alloc/#comment-214262)

안녕하세요? 제가 사용하는 툴은 MS Office의 파워포인트와 유사한 LibreOffice의 Impress 입니다.
여러번 그리다 보니 숙달된 경우이고요. 조금 더 빠르게 그리기 위해,
보통 기존에 그렸던 도형들을 옮겨서 사용합니다. 그러면 약간 시간을 줄여줍니다.
감사합니다.

응답 (/BUDDY-ALLOC/?REPLYTOCOM=214262#RESPOND)

정광민
2019-06-09 19:57 (http://jake.dothome.co.kr/buddy-alloc/#comment-214291)

답변 감사합니다!

응답 (/BUDDY-ALLOC/?REPLYTOCOM=214291#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

‹ Zoned Allocator -4- (Buddy 페이지 해지) (http://jake.dothome.co.kr/buddy-free/)

Per-cpu -4- (atomic operations) › (http://jake.dothome.co.kr/per-cpu-atomic/)

문c 블로그 (2015 ~ 2023)