Zoned Allocator -7- (Direct Compact)

2016-07-01 (http://jake.dothome.co.kr/zonned-allocator-compaction/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

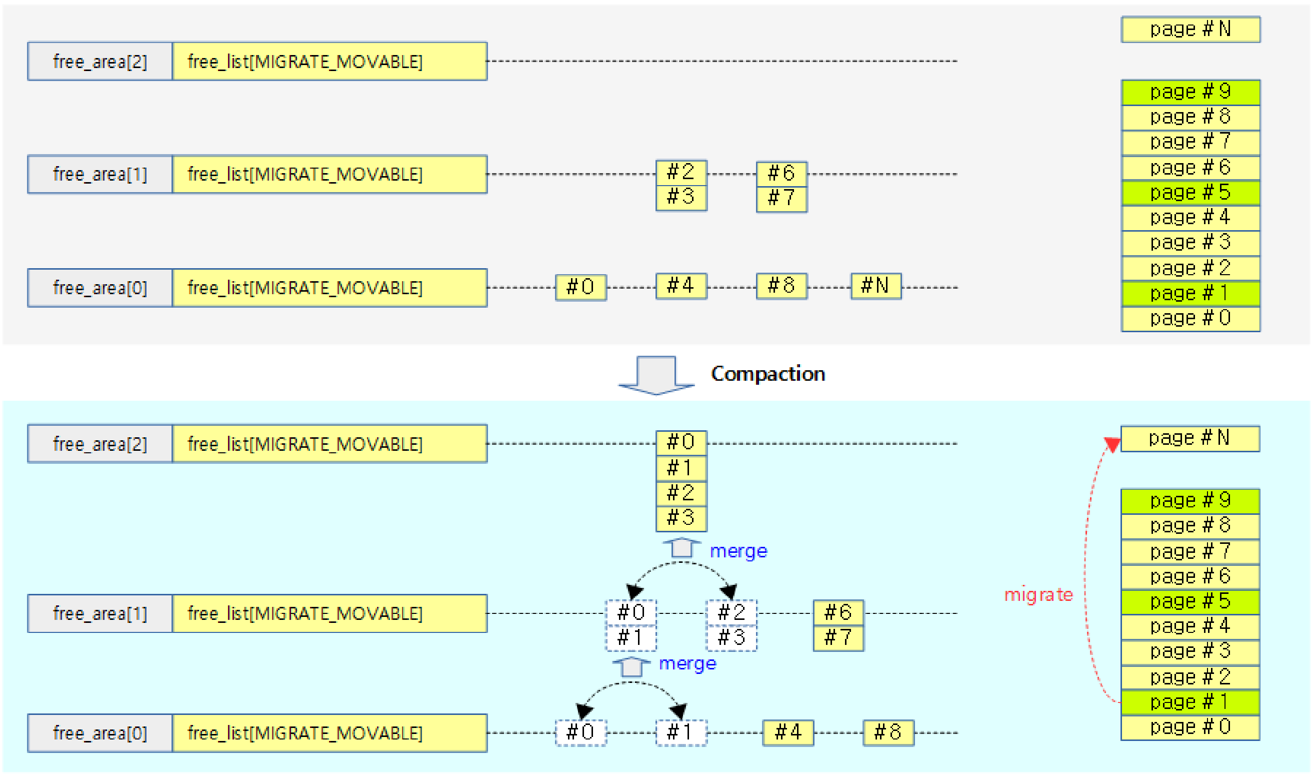
Zoned Allocator -7- (Direct Compact)

Compaction

There may be more free pages for the requested order page, but those pages may be fragmented and unable to respond to high order page allocation requests. In this case, the method of moving the movable page in use to another place and obtaining a continuous free page is called compaction.

- movable pages correspond to memory or files allocated in the user's area. Unmovable pages assigned by the kernel cannot be compactioned.

The following illustration shows how the compaction is performed on the order 2 page allocation request.

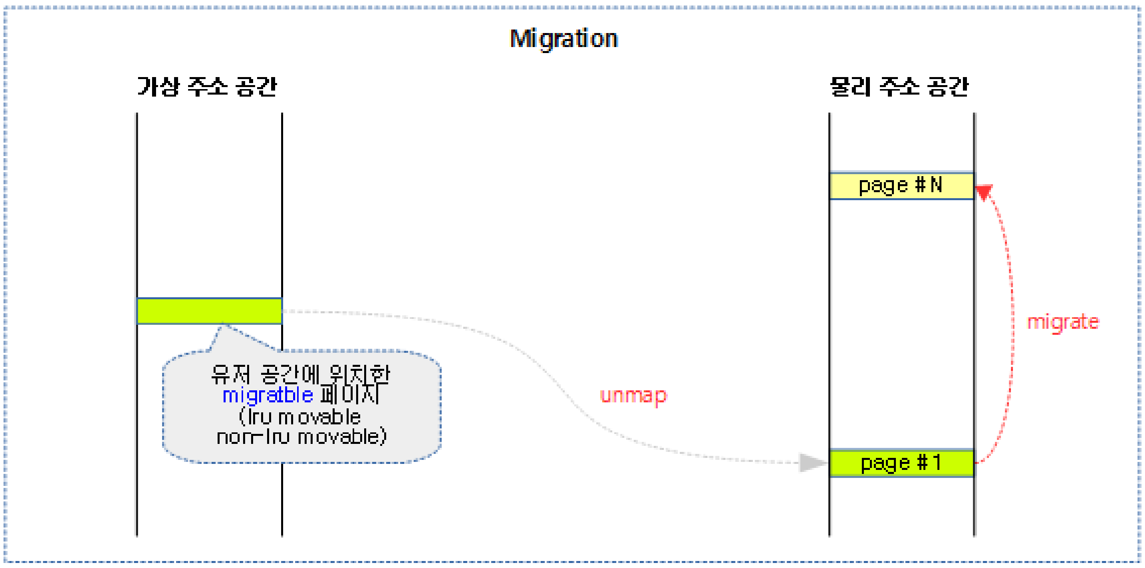


(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction-1.png)

migration

The following illustration shows how to migrate(copy) a migratble page used in userspace to a different physical address.

- In some cases, the CPU does not copy the page during migration, but instead performs the copy by DMA to lower the CPU cost.
- Migrate doesn't map to the virtual address space immediately after it's complete. If the migrated page is accessed in the future, a new mapping is made with only the physical address changed, with no change in the virtual address.


Migration
가상 주소 공간
물리 주소 공간
page #N
page #1
migrate
unmap
유저 공간에 위치한 migratble 페이지 (lru movable non-lru movable)

(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction-2a.png)

migratable page

This is a page that allows you to move an in-use physics page to another physics page. The following types are possible:

- LRU Movable Pages
  - A movable page managed by LRU is a page that can be migrated directly by the kernel page manager.
- Non-LRU Movable Pages
  - Pages that are not managed by LRU are basically not migrated. However, pages in drivers that implement migration are migratory and are called non-lru movable pages.
    - e.g. zsram, balloon memory driver

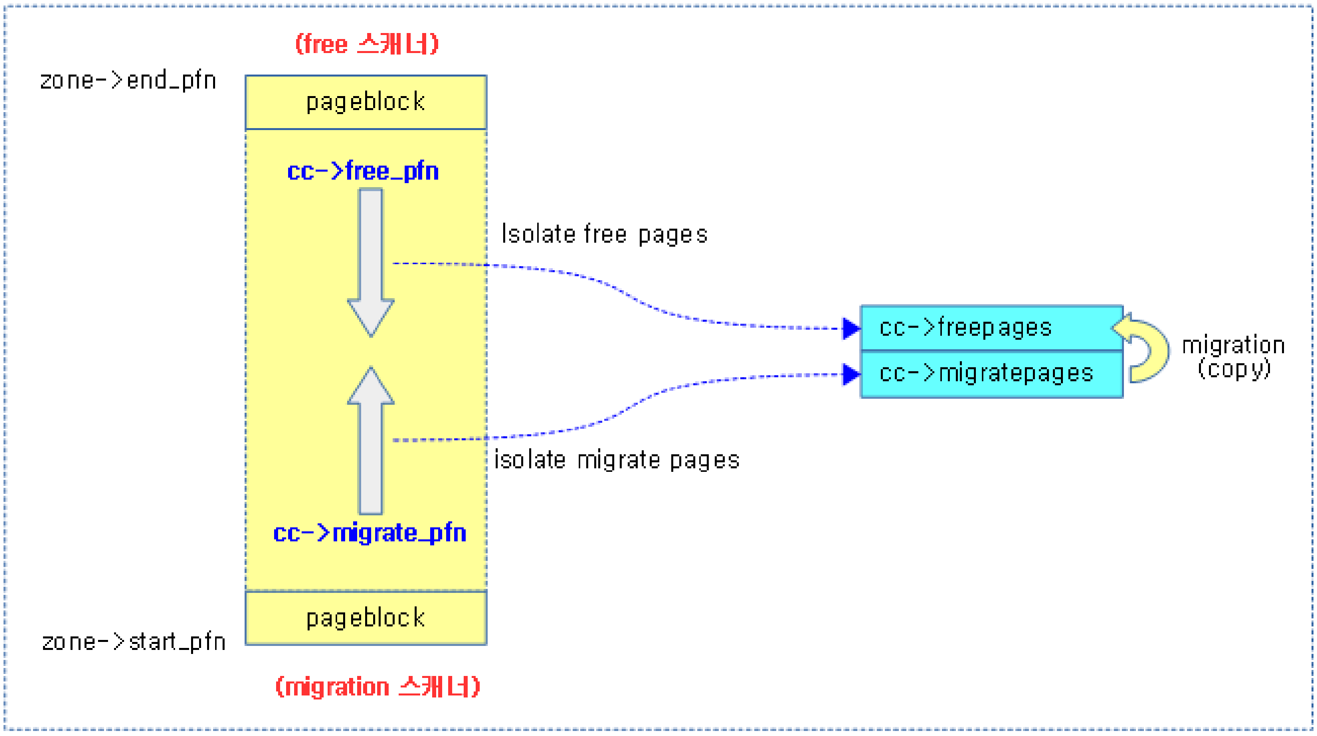
Free Scanner & Migrate Scanner

Once the compaction is in progress, the following two scanners within the zone begin scanning each page block:

- Free Scanner
  - Starting from the top page block, look for the free page in the downward direction.
- Migrate Scanner

  - From the bottom page block, find the migratable page in use in the upward direction.

You can see the process of migrating the migratable pages found by the migrate scanner to the free pages found by the free scanner, as shown in the following figure.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction-3.png)

compact_priority

include/linux/compaction.h

```
/*
 * Determines how hard direct compaction should try to succeed.
 * Lower value means higher priority, analogically to reclaim priority.
 */
```

```
enum compact_priority {
        COMPACT_PRIO_SYNC_FULL,
        MIN_COMPACT_PRIORITY = COMPACT_PRIO_SYNC_FULL,
        COMPACT_PRIO_SYNC_LIGHT,
        MIN_COMPACT_COSTLY_PRIORITY = COMPACT_PRIO_SYNC_LIGHT,
        DEF_COMPACT_PRIORITY = COMPACT_PRIO_SYNC_LIGHT,
        COMPACT_PRIO_ASYNC,
        INIT_COMPACT_PRIORITY = COMPACT_PRIO_ASYNC
};
```

Trying compactin is a three-step priority for success.

- COMPACT_PRIO_SYNC_FULL(0) & MIN_COMPACT_PRIORITY
  - With the highest priority, compaction and migration are in full sync.
- COMPACT_PRIO_SYNC_LIGHT(1) & MIN_COMPACT_COSTLY_PRIORITY & DEF_COMPACT_PRIORITY
  - By default and intermediate priority, compaction works with sync, but migration works with async.
- COMPACT_PRIO_ASYNC(2) & INIT_COMPACT_PRIORITY

- With the initial and lowest priority, compaction and migration work with async.

migrate_mode

include/linux/migrate_mode.h

```
/*
 * MIGRATE_ASYNC means never block
 * MIGRATE_SYNC_LIGHT in the current implementation means to allow block
ing
 *      on most operations but not ->writepage as the potential stall ti
me
 *      is too significant
 * MIGRATE_SYNC will block when migrating pages
 */
```

```
enum migrate_mode {
        MIGRATE_ASYNC,
        MIGRATE_SYNC_LIGHT,
        MIGRATE_SYNC,
        MIGRATE_SYNC_NO_COPY
};
```

This is the mode used to migrate pages.

- MIGRATE_ASYNC
  - It operates in asynchronous migration mode and is not blocked.
  - It is used in the async compaction operation.
- MIGRATE_SYNC_LIGHT
  - Most of them run in synchronous mode, except for writepage.
  - Used by kcompactd
  - It is used in the sync compaction operation.
- MIGRATE_SYNC
  - It works in synchronous migration mode and is blocked.
- MIGRATE_SYNC_NO_COPY
  - It works in synchronous migration mode and is blocked, but instead of copying the cpu for the migration page, it uses DMA to copy it.

Compaction Operating Modes

There are three ways to compaction:

- direct-compaction
  - When requesting a free page allocation of an order, when it is difficult to allocate the order due to insufficient memory, it is a method of calling it directly from inside the page allocation API when the compaction is performed.
- manual-compaction
  - Regardless of the order, the manual request is made through the following command:
    - “echo 1 > /proc/sys/vm/compact_memory”
- kcompactd

- It automatically wakes up when you run out of memory and performs compaction in the background.

Manual Compaction

Check the status of each order page as follows.

```
# cat /proc/pagetypeinfo
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4
5      6      7      8      9     10
Node    0, zone      DMA, type    Unmovable    485    196  50044     12      5
0      0      1      1      1      0
Node    0, zone      DMA, type      Movable     22     69     66     51     46
30     21     11      8      1    386
Node    0, zone      DMA, type  Reclaimable     50     25     11      0      1
0      1      1      1      1      0
Node    0, zone      DMA, type   HighAtomic      0      0      0      0      0
0      0      0      0      0      0
Node    0, zone      DMA, type          CMA   1284    886    567    319    149
81     46     31     11     10     61
Node    0, zone      DMA, type      Isolate      0      0      0      0      0
0      0      0      0      0      0

Number of blocks type     Unmovable      Movable  Reclaimable   HighAtomic
CMA      Isolate
Node 0, zone      DMA          403          417            6            0
124            0
```

To compaction a movable page, let's use the following command to compaction the manual.

```
echo 1 > /proc/sys/vm/compact_memory
```

You can see the result of a part of the movable page being compacted as follows: However, you can see that the unmovable pages used by the kernel do not compaction.

```
# cat /proc/pagetypeinfo
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4
5      6      7      8      9     10
Node    0, zone      DMA, type    Unmovable    489    196  50044     11      5
0      0      1      1      1      0
Node    0, zone      DMA, type      Movable     22     43     36     32     27
24     18     14      9      1    386
Node    0, zone      DMA, type  Reclaimable     69     26     11      0      1
0      1      1      1      1      0
Node    0, zone      DMA, type   HighAtomic      0      0      0      0      0
0      0      0      0      0      0
Node    0, zone      DMA, type          CMA   1189    814    521    292    134
75     42     30     12      9     63
Node    0, zone      DMA, type      Isolate      0      0      0      0      0
0      0      0      0      0      0

Number of blocks type     Unmovable      Movable  Reclaimable   HighAtomic
CMA      Isolate
Node 0, zone      DMA          403          417            6            0
124            0
```

kcomactd

It automatically wakes up when memory is low and performs a compact in the background, and was introduced in kernel v4.6-rc1.

- consultation
  - mm, compaction: introduce kcompactd (https://github.com/torvalds/linux/commit/698b1b30642f1ff0ea10ef1de9745ab633031377#diff-a187bd6389cbd112c7e026c401962224)
  - mm, kswapd: replace kswapd compaction with waking up kcompactd (https://github.com/torvalds/linux/commit/accf62422b3a67fce8ce086aa81c8300ddbf42be#diff-a187bd6389cbd112c7e026c401962224)

Balancing Judgment

Check whether the requested 2^order page with a high watermark or higher can be assigned.

pgdat_balanced()

mm/vmscan.c

```
/*
```

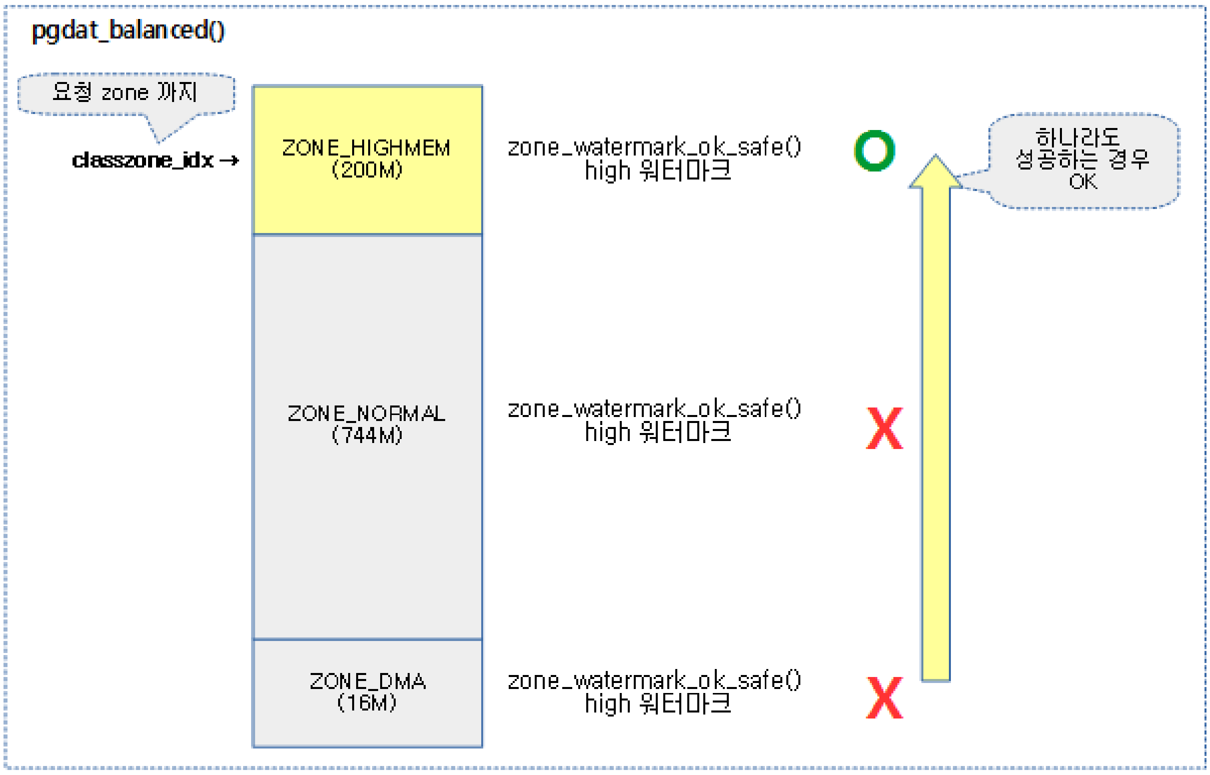
```
2   * Returns true if there is an eligible zone balanced for the request order
3   * and classzone_idx
4   */
```

```
01 static bool pgdat_balanced(pg_data_t *pgdat, int order, int classzone_idx)
02 {
03         int i;
04         unsigned long mark = -1;
05         struct zone *zone;
06 
07         /*
08          * Check watermarks bottom-up as lower zones are more likely to
09          * meet watermarks.
10          */
11         for (i = 0; i <= classzone_idx; i++) {
12                 zone = pgdat->node_zones + i;
13 
14                 if (!managed_zone(zone))
15                         continue;
16 
17                 mark = high_wmark_pages(zone);
18                 if (zone_watermark_ok_safe(zone, order, mark, classzone_idx))
19                         return true;
20         }
21 
22         /*
23          * If a node has no populated zone within classzone_idx, it does not
24          * need balancing by definition. This can happen if a zone-restricted
25          * allocation tries to wake a remote kswapd.
26          */
27         if (mark == -1)
28                 return true;
29 
30         return false;
31 }
```

Returns the presence or absence of balance for zones less than @classzone_idx of the node. Determine whether the free page exceeds the high watermark to determine whether it is balanced.

- In line 11~20 of code, it traverses from zone 0 to zone @classzone_idx, and returns a tuue if it is possible to obtain an order page with a high watermark or higher.
- In line 27~28 of the code, if there are no managed pages, it returns true because no balancing operation is required.

Returns true if the requested node is balanced to the classzone_idx zone, as shown in the following figure.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/pgdat_balanced-1a.png)

Compaction Performance Conditions

Verify that compaction persists

compaction_suitable()

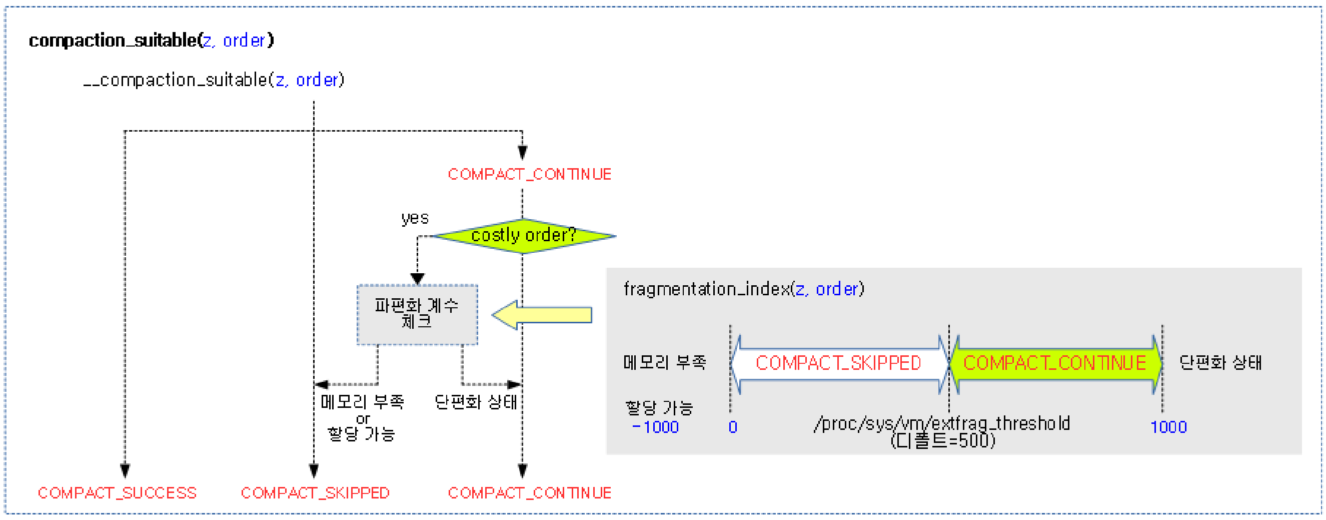
mm/compaction.c

```
enum compact_result compaction_suitable(struct zone *zone, int order,
                                        unsigned alloc_flags,
                                        int classzone_idx)
{
        enum compact_result ret;
        int fragindex;

        ret = __compaction_suitable(zone, order, alloc_flags, classzone_idx,
                                    zone_page_state(zone, NR_FREE_PAGES));
        /*
         * fragmentation index determines if allocation failures are due to
         * low memory or external fragmentation
         *
         * index of -1000 would imply allocations might succeed depending on
         * watermarks, but we already failed the high-order watermark check
         * index towards 0 implies failure is due to lack of memory
         * index towards 1000 implies failure is due to fragmentation
         *
         * Only compact if a failure would be due to fragmentation. Also
         * ignore fragindex for non-costly orders where the alternative to
         * a successful reclaim/compaction is OOM. Fragindex and the
22           * vm.extfrag_threshold sysctl is meant as a heuristic to prevent
23           * excessive compaction for costly orders, but it should not be at the
24           * expense of system stability.
25           */
26          if (ret == COMPACT_CONTINUE && (order > PAGE_ALLOC_COSTLY_ORDER)) {
27                  fragindex = fragmentation_index(zone, order);
28                  if (fragindex >= 0 && fragindex <= sysctl_extfrag_threshold)
29                          ret = COMPACT_NOT_SUITABLE_ZONE;
30          }
31
32          trace_mm_compaction_suitable(zone, order, ret);
33          if (ret == COMPACT_NOT_SUITABLE_ZONE)
34                  ret = COMPACT_SKIPPED;
35
36          return ret;
37  }
```

Returns a compaction suitable result for the assignment of 2^order pages in the request zone.

- From lines 8~9 of the code, we get the result of whether the compaction is persisted or not.
- In line 26~34 of the code, if the costly order page request is a continuation result, check the fragmentation factor value and change the COMPACT_SKIPPED to the return value if it determines that it is difficult to compact.
  - If the fragmentation coefficient is in the range [0, sysctl_extfrag_threshold], then the purpose is not to compaction.
  - sysctl_extfrag_threshold
    - The default value is 500.
    - You can use the "proc/sys/vm/extfrag_threshold" file to change the value.

The following figure shows the return of the result of whether to continue the compaction, and the process of additionally checking the fragmentation factor to see if it is okay to continue when the costly order request is a continuation result.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction_suitable-1c.png)

compact_result

include/linux/compaction.h

```
/* Return values for compact_zone() and try_to_compact_pages() */
/* When adding new states, please adjust include/trace/events/compaction.h */
enum compact_result {
        /* For more detailed tracepoint output - internal to compaction */
        COMPACT_NOT_SUITABLE_ZONE,
        /*
         * compaction didn't start as it was not possible or direct reclaim
         * was more suitable
         */
        COMPACT_SKIPPED,
        /* compaction didn't start as it was deferred due to past failures */
        COMPACT_DEFERRED,

        /* compaction not active last round */
        COMPACT_INACTIVE = COMPACT_DEFERRED,

        /* For more detailed tracepoint output - internal to compaction */
        COMPACT_NO_SUITABLE_PAGE,
        /* compaction should continue to another pageblock */
        COMPACT_CONTINUE,

        /*
         * The full zone was compacted scanned but wasn't successfull to compact
         * suitable pages.
         */
        COMPACT_COMPLETE,
        /*
         * direct compaction has scanned part of the zone but wasn't successfull
         * to compact suitable pages.
         */
        COMPACT_PARTIAL_SKIPPED,

        /* compaction terminated prematurely due to lock contentions */
        COMPACT_CONTENDED,

        /*
         * direct compaction terminated after concluding that the allocation
         * should now succeed
         */
        COMPACT_SUCCESS,
};
```

The result of the check before the compaction attempt or the result after the compaction is performed.

- COMPACT_NOT_SUITABLE_ZONE
  - trace is used for debug output or for internal use.
- COMPACT_SKIPPED
  - Skip the compaction because it can't be done, or if direct-reclaim is more appropriate.
- COMPACT_DEFERRED & COMPACT_INACTIVE
  - Since the previous compaction failed, this time we skip the compaction to suspend it.
- COMPACT_NO_SUITABLE_PAGE
  - trace is used for debug output or for internal use.
- COMPACT_CONTINUE

  - The compaction should be carried on on the other page blocks.
  - In the case of manual compaction, proceed until all blocks in the relevant area have been completed.
- COMPACT_COMPLETE
  - Compaction has completed for all zones, but no pages are available for assignment through compaction.
- COMPACT_PARTIAL_SKIPPED
  - Direct compliance has been performed on some of the zones, but we have not yet succeeded in securing assignable pages.
- COMPACT_CONTENDED
  - Compaction terminated prematurely due to lock contention.
- COMPACT_SUCCESS
  - After assignable pages, Direct Disable was terminated.

__compaction_suitable()

mm/compaction.c

```
/*
 * compaction_suitable: Is this suitable to run compaction on this zone now?
 * Returns
 *   COMPACT_SKIPPED  - If there are too few free pages for compaction
 *   COMPACT_PARTIAL  - If the allocation would succeed without compaction
 *   COMPACT_CONTINUE - If compaction should run now
 */
```

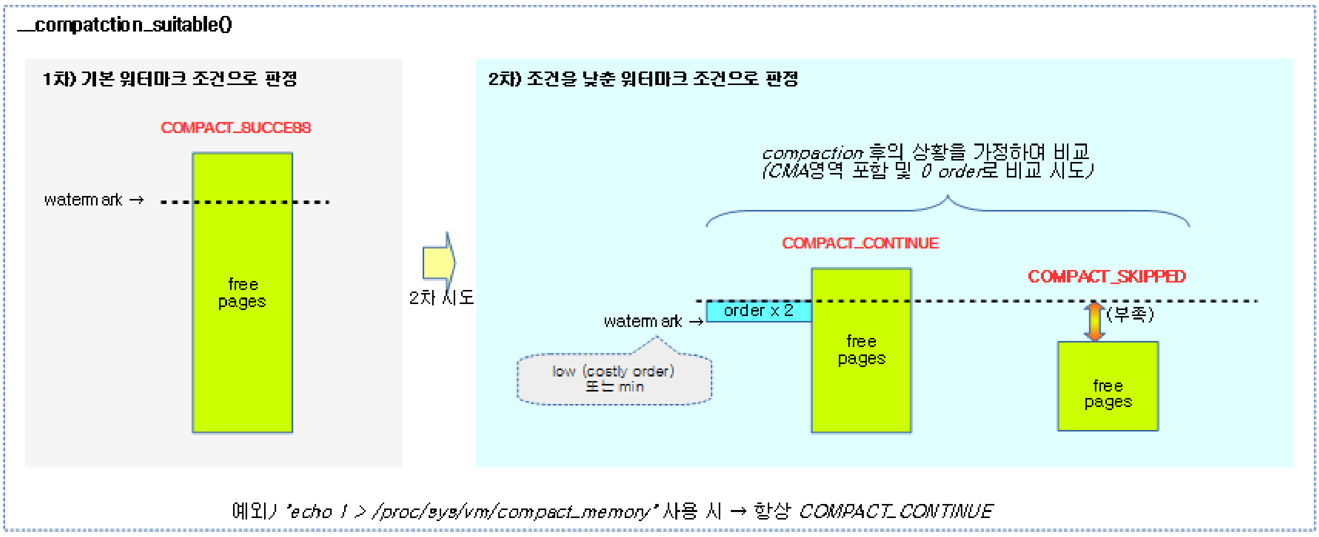
```
static enum compact_result __compaction_suitable(struct zone *zone, int order,
                                        unsigned int alloc_flags,
                                        int classzone_idx,
                                        unsigned long wmark_target)
{
        unsigned long watermark;

        if (is_via_compact_memory(order))
                return COMPACT_CONTINUE;

        watermark = wmark_pages(zone, alloc_flags & ALLOC_WMARK_MASK);
        /*
         * If watermarks for high-order allocation are already met, there
         * should be no need for compaction at all.
         */
        if (zone_watermark_ok(zone, order, watermark, classzone_idx,
                                                        alloc_flags))
                return COMPACT_SUCCESS;

        /*
         * Watermarks for order-0 must be met for compaction to be able to
         * isolate free pages for migration targets. This means that the
         * watermark and alloc_flags have to match, or be more pessimistic than
         * the check in __isolate_free_page(). We don't use the direct
         * compactor's alloc_flags, as they are not relevant for freepage
26          * isolation. We however do use the direct compactor's classzone_idx to
27          * skip over zones where lowmem reserves would prevent allocation even
28          * if compaction succeeds.
29          * For costly orders, we require low watermark instead of min for
30          * compaction to proceed to increase its chances.
31          * ALLOC_CMA is used, as pages in CMA pageblocks are considered
32          * suitable migration targets
33          */
34         watermark = (order > PAGE_ALLOC_COSTLY_ORDER) ?
35                                 low_wmark_pages(zone) : min_wmark_pages(zone);
36         watermark += compact_gap(order);
37         if (!__zone_watermark_ok(zone, 0, watermark, classzone_idx,
38                                                 ALLOC_CMA, wmark_target))
39                 return COMPACT_SKIPPED;
40 
41         return COMPACT_CONTINUE;
42 }
```

Returns the result whether the request is in progress or not using zone and order.

- If the user requests a compaction in line 8~9, it returns a COMPACT_CONTINUE to proceed unconditionally.
    - You can request a compaction with "echo 1 > /proc/sys/vm/compact_memory".
- In line 11 of the code, we notice the watermark of the request zone.
- In line 16~18 of the code, if a free page with a watermark or higher is obtained in the first instance, a COMPACT_SUCCESS is returned to a situation where compaction is no longer needed.
- Comparing the situation assuming that the second compaction is completed in line 34~41 of the code, as a result, it returns COMPACT_SKIPPED if it determines that there is still insufficient memory, and returns COMPACT_CONTINUE if there is a possibility of securing the page.
    - For costly high order requests, use the low watermark criterion, and for low order requests, use the min watermark criterion.
    - During the short period of compaction, the pages are copied and allocated, so that the number of requested order pages is doubled plus the watermark value, lowered to the 0 order page baseline, including the cma area, to determine whether it can be allocated when compared.

The following illustration shows the process of determining if it is okay to continue performing compaction.

- In the secondary condition, it is assumed after the compaction situation and compared with the watermark that changed the criterion to 2 order.


__compatction_suitable()
1차) 기본 워터마크 조건으로 판정
COMPACT_SUCCESS
watermark →
free pages
2차 시도
2차) 조건을 낮춘 워터마크 조건으로 판정
compaction 후의 상황을 가정하여 비교
(CMA영역 포함 및 0 order로 비교 시도)
COMPACT_CONTINUE
COMPACT_SKIPPED
order x 2
watermark →
low (costly order) 또는 min
free pages
(부족)
free pages
예외) 'echo 1 > /proc/sys/vm/compact_memory' 사용 시 → 항상 COMPACT_CONTINUE

(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction_suitable-1.png)

Calculating the Fragmentation Coefficient

fragmentation_index()

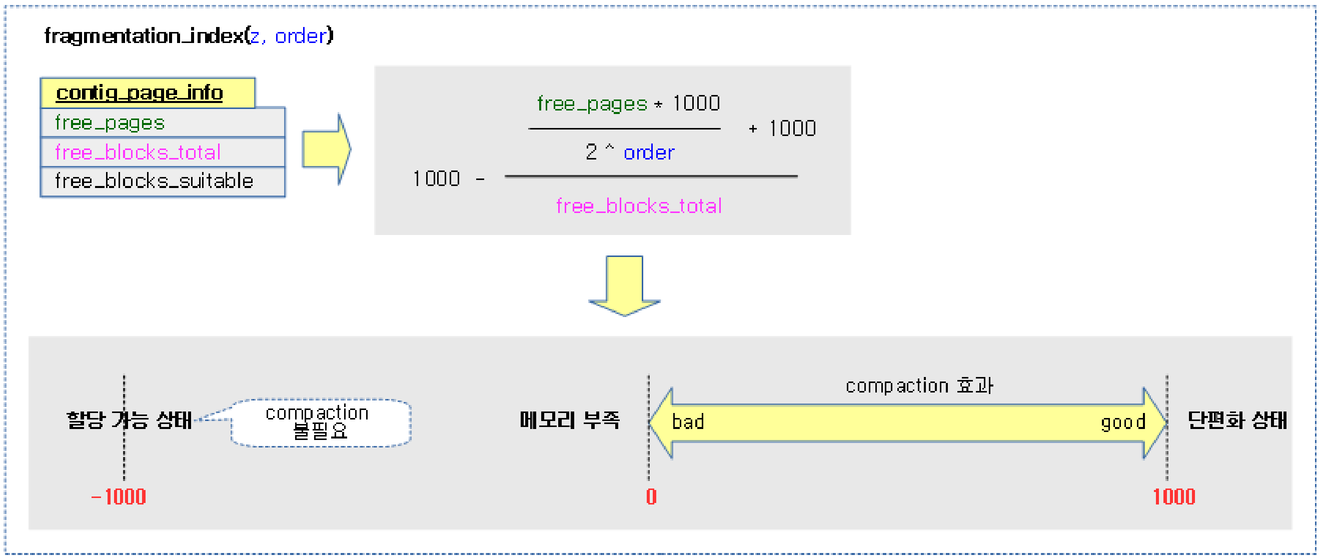
mm/vmstat.c

```
1  /* Same as __fragmentation index but allocs contig_page_info on stack */
2  int fragmentation_index(struct zone *zone, unsigned int order)
3  {
4          struct contig_page_info info;
5
6          fill_contig_page_info(zone, order, &info);
7          return __fragmentation_index(order, &info);
8  }
```

We know the fragmentation factor for the request zone and order to determine whether we should compaction. If the fragmentation factor value returns -1000, there is no need for compaction because there are pages to assign. In other cases, the value is within the range of 0 ~ 1000, and if it is less than sysctl_extfrag_threshold, it is not intended to compact.

- In line 6 of the code, the buddy system for the specified zone contains information about the total free block, the total free page, and the number of free blocks that can be assigned to the order page.
- In line 7 of the code, we use the information contig_page the request order to calculate the fragmentation factor.

The following figure shows the process of calculating the value of the fragmentation coefficient.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/fragmentation_index-1a.png)

fill_contig_page_info()

mm/vmstat.c

```
/*
 * Calculate the number of free pages in a zone, how many contiguous
 * pages are free and how many are large enough to satisfy an allocation of
 * the target size. Note that this function makes no attempt to estimate
 * how many suitable free blocks there *might* be if MOVABLE pages were
 * migrated. Calculating that is possible, but expensive and can be
 * figured out from userspace
 */
```

```
static void fill_contig_page_info(struct zone *zone,
                                unsigned int suitable_order,
                                struct contig_page_info *info)
{
        unsigned int order;

        info->free_pages = 0;
        info->free_blocks_total = 0;
        info->free_blocks_suitable = 0;

        for (order = 0; order < MAX_ORDER; order++) {
                unsigned long blocks;

                /* Count number of free blocks */
                blocks = zone->free_area[order].nr_free;
                info->free_blocks_total += blocks;

                /* Count free base pages */
                info->free_pages += blocks << order;

                /* Count the suitable free blocks */
                if (order >= suitable_order)
                        info->free_blocks_suitable += blocks <<
                                                (order - suitable_order);
        }
}
```

Returns information about the total free block, the total free page, and the number of allocable free blocks in the suitable_order in the buddy system of the specified zone as a structure contig_page_info info.

- In code lines 11~16, we traverse the list of each order in the buddy system managed by the zone and add up the total number of free blocks.
- Sum the number of free pages on line 19 of code.
- In line 22~24 of the code, add up the number of free blocks above the requested order.

contig_page_info Structure

mm/vmstat.c

```
#ifdef CONFIG_COMPACTION
struct contig_page_info {
        unsigned long free_pages;
        unsigned long free_blocks_total;
        unsigned long free_blocks_suitable;
};
#endif
```

This information is used to calculate the fragmentation factor for the requested order.

- free_pages
  - The number of all free pages managed by the Buddy system.
  - e.g. order 3 if there are 2 pages
    - 16 (2^3 * 2) pages
- free_blocks_total
  - The number of all free blocks (main pages) managed in the buddy system
  - e.g. order 3 if there are 2 pages
    - 2
- free_blocks_suitable
  - Number of free blocks (main page) that satisfy the requested order

__fragmentation_index()

mm/vmstat.c

```
/*
 * A fragmentation index only makes sense if an allocation of a requested
 * size would fail. If that is true, the fragmentation index indicates
 * whether external fragmentation or a lack of memory was the problem.
 * The value can be used to determine if page reclaim or compaction
 * should be used
 */
```

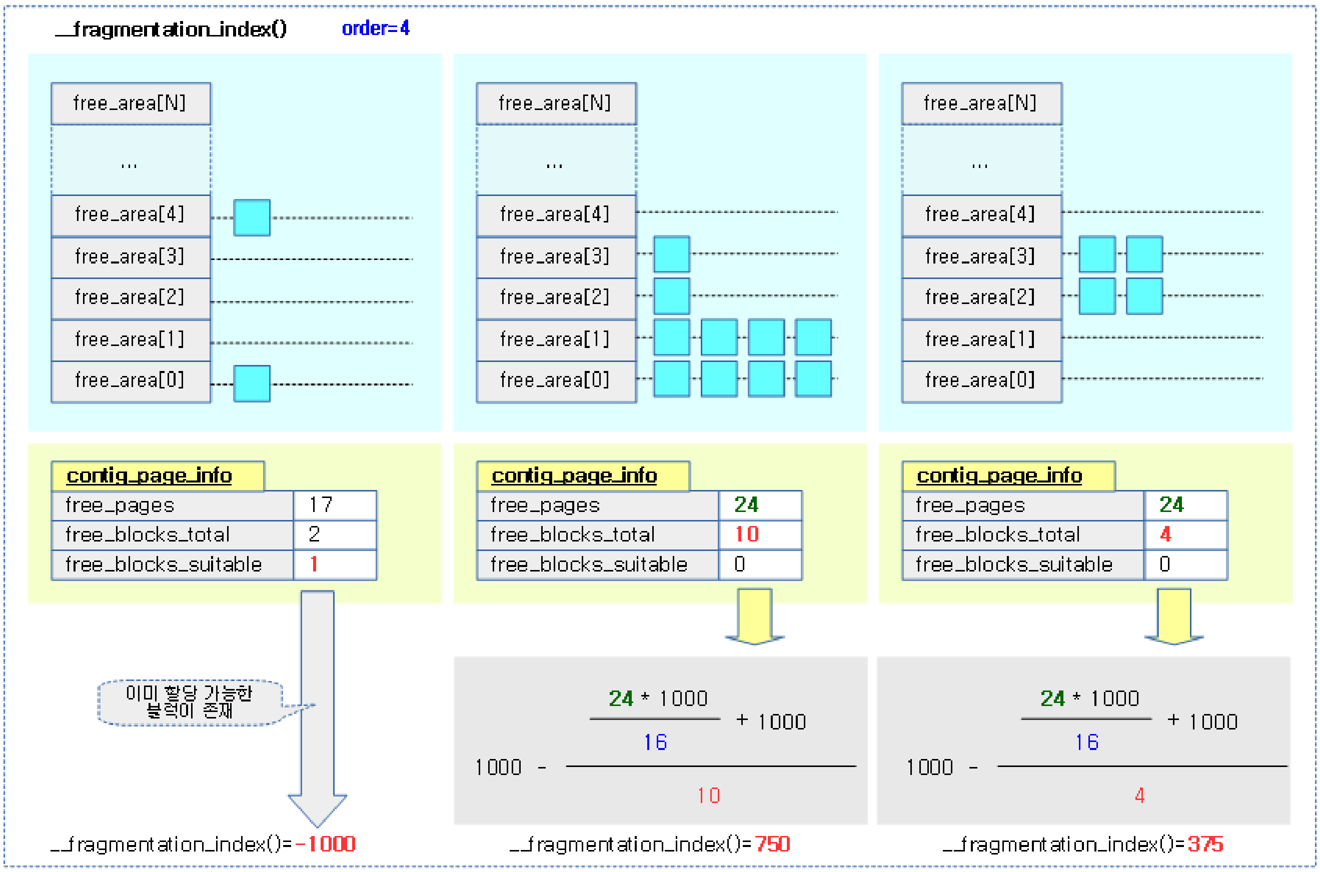
```
static int __fragmentation_index(unsigned int order, struct contig_page_info *info)
{
        unsigned long requested = 1UL << order;

        if (WARN_ON_ONCE(order >= MAX_ORDER))
                return 0;

08          if (!info->free_blocks_total)
09                  return 0;
10
11          /* Fragmentation index only makes sense when a request would fail */
12          if (info->free_blocks_suitable)
13                  return -1000;
14
15          /*
16           * Index is between 0 and 1 so return within 3 decimal places
17           *
18           * 0 => allocation would fail due to lack of memory
19           * 1 => allocation would fail due to fragmentation
20           */
21          return 1000 - div_u64( (1000+(div_u64(info->free_pages * 1000ULL, requested))), info->free_blocks_total);
22  }
```

Returns the fragmentation factor using the request order, the free page, and the free block information.

- Values close to zero (low fragmentation coefficient)
  - is in a situation where the allocation will fail due to insufficient memory.
  - Subsequently, the compaction is also likely to fail the allocation
- Values close to 1000 (high fragmentation coefficient)
  - It is a situation in which allocation will fail due to fragmentation.
  - When you compaction, the allocation is more likely to succeed
- -1000 (assignable status)
  - A request order block exists and can be allocated.
  - No compaction required

- In line 5~6 of the code, if the page requests an order that exceeds the maximum buddy order, it returns 0.
- In line 8~9 of the code, if the total number of free blocks is 0, it returns 0 because it cannot compaction.
- In lines 12~13 of code, if there is a free block that can handle the request order page, it returns -1000 as there is no need for compaction.
- In code line 21 to 1000 – (full free page x 1000 / required page count + 1000) / total free block count
  - The closer it is to zero, the better it is not to allow compaction due to insufficient memory.
  - The closer you get to 1000, the more compaction you need to be on a fragmented page.

The following figure shows the process of calculating the value of the fragmentation coefficient.

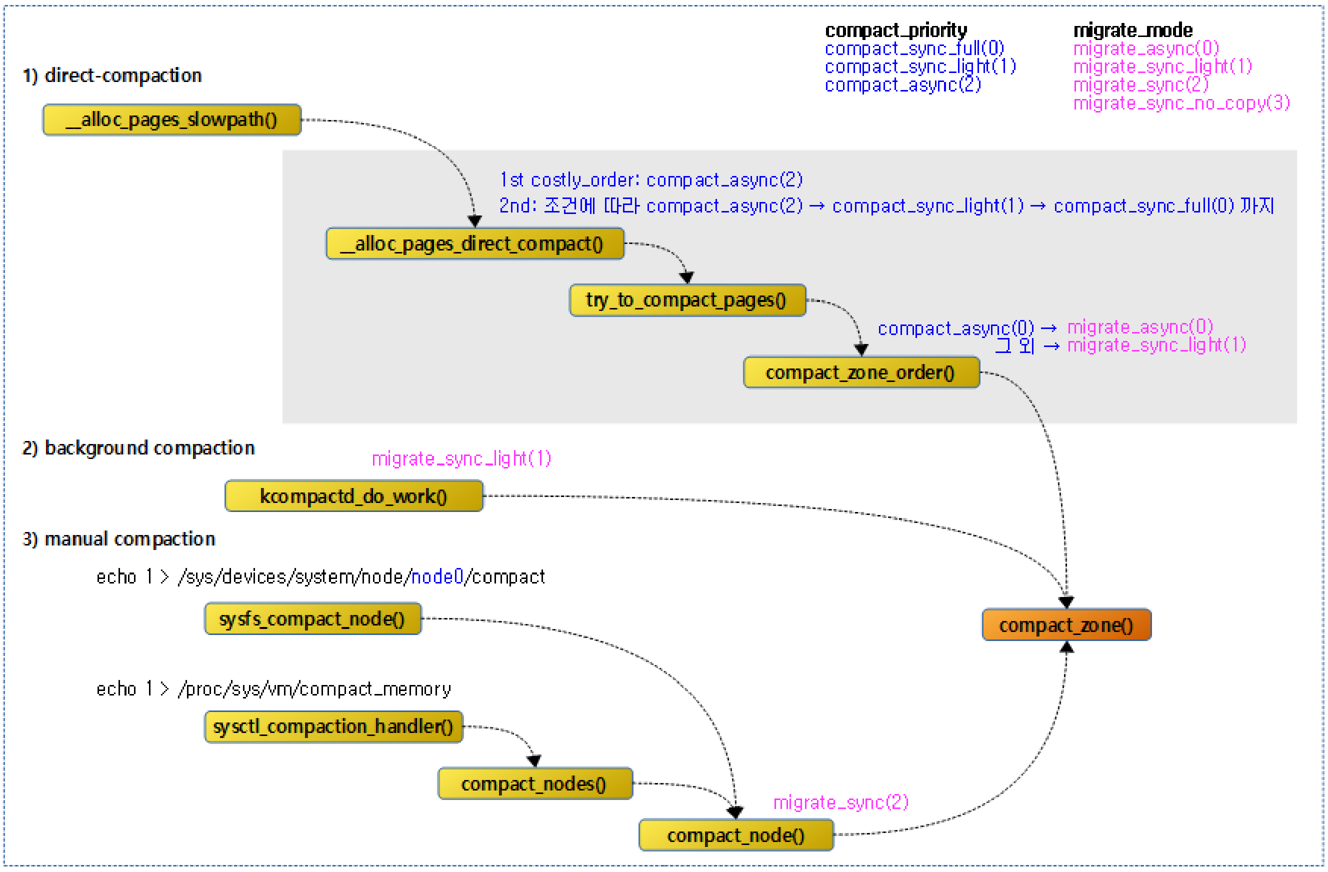


(http://jake.dothome.co.kr/wp-content/uploads/2016/07/fragmentation_index-1b.png)

Performing Compaction

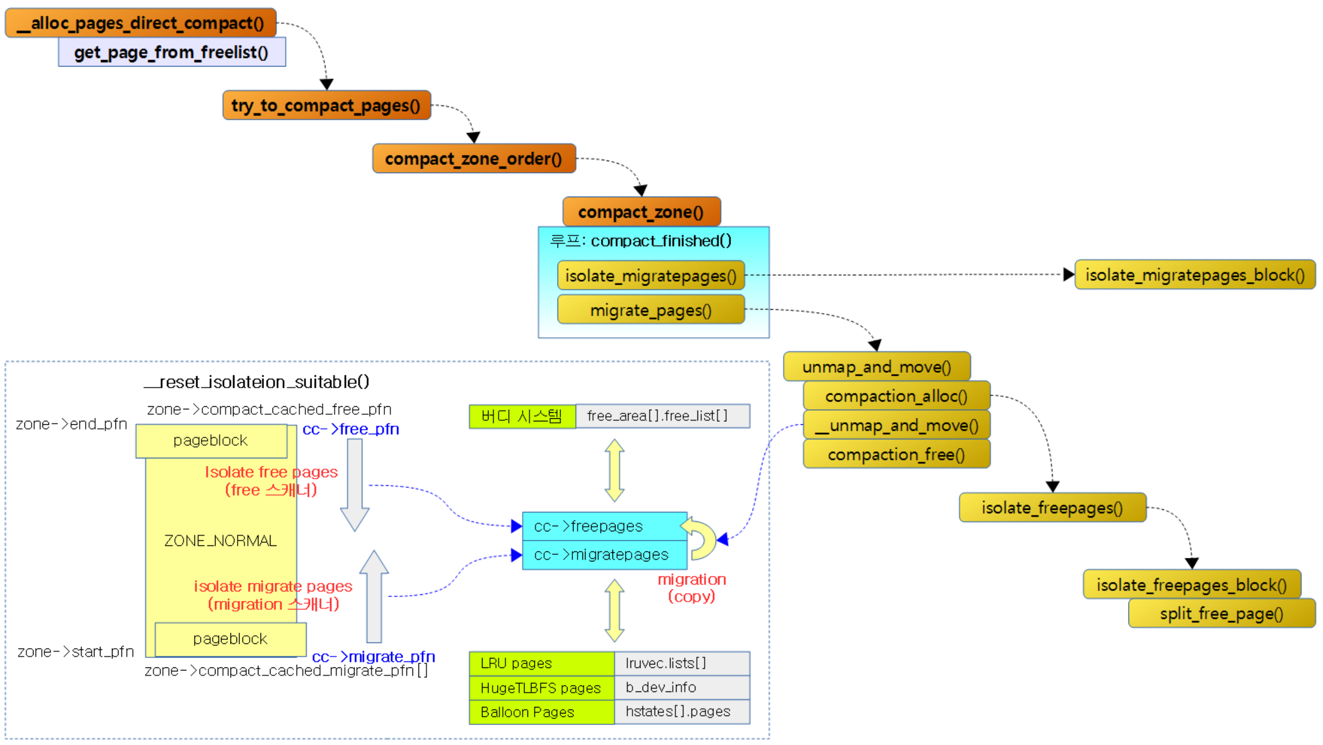
The following figure shows the different paths in which a compaction is performed.

- Check the compact priority and migrate mode as well.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction-4.png)

The following figure shows the flow of a function when direct compliance is performed.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/alloc_pages_direct_compact-2.png)

Page allocation using direct-compliance

__alloc_pages_direct_compact()

mm/page_alloc.c

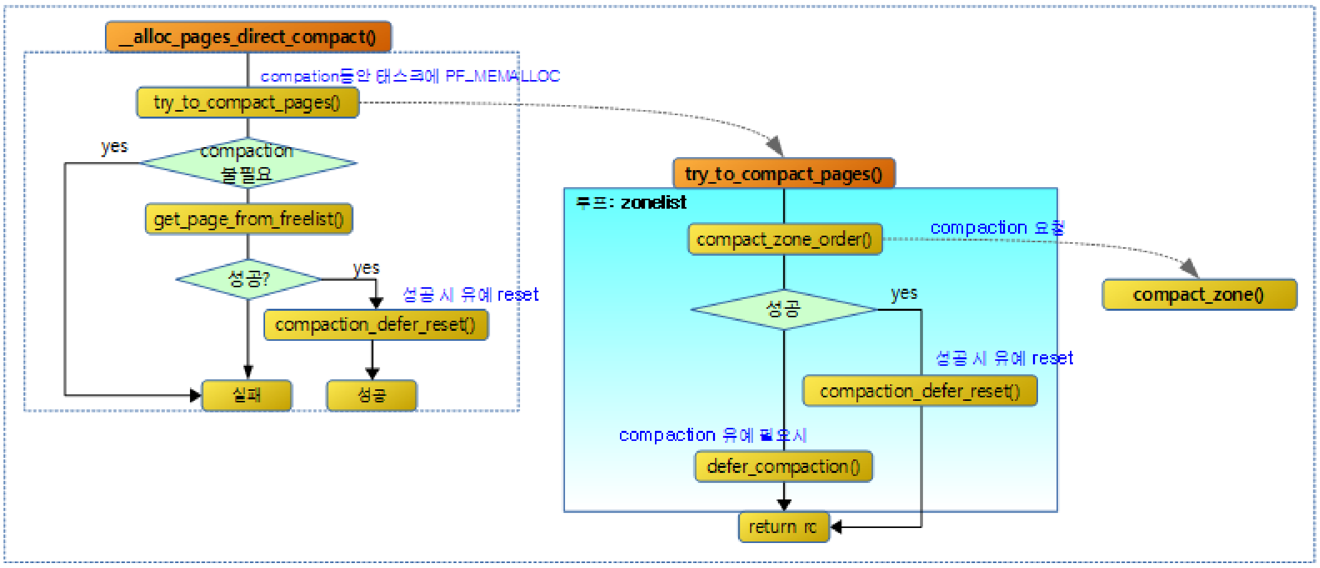
```
01 /* Try memory compaction for high-order allocations before reclaim */
02 static struct page *
03 __alloc_pages_direct_compact(gfp_t gfp_mask, unsigned int order,
04                 unsigned int alloc_flags, const struct alloc_context *ac,
05                 enum compact_priority prio, enum compact_result *compact_result)
06 {
07         struct page *page;
08         unsigned long pflags;
09         unsigned int noreclaim_flag;
10 
11         if (!order)
12                 return NULL;
13 
14         psi_memstall_enter(&pflags);
15         noreclaim_flag = memalloc_noreclaim_save();
16 
17         *compact_result = try_to_compact_pages(gfp_mask, order, alloc_flags, ac,
18                                                                         prio);
19 
20         memalloc_noreclaim_restore(noreclaim_flag);
21         psi_memstall_leave(&pflags);
22 
23         if (*compact_result <= COMPACT_INACTIVE)
24                 return NULL;
25 
26         /*
27          * At least in one zone compaction wasn't deferred or skipped, so let's
28          * count a compaction stall
29          */
30         count_vm_event(COMPACTSTALL);
31 
32         page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
33 
34         if (page) {
35                 struct zone *zone = page_zone(page);
36 
37                 zone->compact_blockskip_flush = false;
38                 compaction_defer_reset(zone, order, true);
39                 count_vm_event(COMPACTSUCCESS);
40                 return page;
41         }
42 
43         /*
44          * It's bad if compaction run occurs and fails. The most likely reason
45          * is that pages exist, but not enough to satisfy watermarks.
46          */
47         count_vm_event(COMPACTFAIL);
48 
49         cond_resched();
50 
51         return NULL;
52 }
```

direct-compaction, and then attempt to allocate the page.

- In code lines 11~12, if order is 0, it cannot be resolved by compaction, so it is not processed.
- This is the point at line 14 of code where we start calculating the psi of the current task due to insufficient memory.

  - PSI was introduced in kernel v2018.4-rc20 in 1.
  - consultation
    - PSI – Pressure Stall Information (https://www.kernel.org/doc/Documentation/accounting/psi.txt) | kernel.org
    - psi: pressure stall information for CPU, memory, and IO (https://github.com/torvalds/linux/commit/eb414681d5a07d28d2ff90dc05f69ec6b232ebd2#diff-afd23506360c175bb0ef2e681113a7ba)
    - psi: cgroup support (https://github.com/torvalds/linux/commit/2ce7135adc9ad081aa3c49744144376ac74fea60#diff-afd23506360c175bb0ef2e681113a7ba)
    - psi: introduce psi monitor (https://github.com/torvalds/linux/commit/0e94682b73bfa6c44c98af7a26771c9c08c055d5#diff-afd23506360c175bb0ef2e681113a7ba)
    - Getting Started with PSI (https://facebookmicrosites.github.io/psi/docs/overview) | PSI
- In line 15 of the code, we need to allocate twice as much memory as the requested order to perform the direct-compaction, but since we are currently out of memory, we use the pfmemalloc flag on the current task to set the memory allocation without watermark restrictions.
- In lines 17~18 of the code, try direct-compliance for the request order page and get the compact progress as a result.
- This is the point at which the PSI output ends at line 20 of code.
- In line 21 of code, reinstate the use of the pfmemalloc flag in the current task.
- If the result of performing a compaction in line 23~24 is inactive or lower, it returns null because it is no longer possible to secure the page.
- Increment the COMPACTSTALL counter on line 30 of code.
- Attempt to secure the page from code lines 32~41. If the page is secured, increment the COMPACTSUCCESS counter, assign false to the compact_blockskip_flush of the zone, reset the tracking counters for compaction, and return the page.
- In line 47~51 of the code, if the page allocation fails, increment the COMPACTFAIL stat and sleep and exit the function if the lisscheduling is necessary.

The following figure shows the progression flow of direct compaction by function.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/alloc_pages_direct_compact-1a.png)

try_to_compact_pages()

mm/compaction.c

```
/**
 * try_to_compact_pages - Direct compact to satisfy a high-order allocat
ion
 * @gfp_mask: The GFP mask of the current allocation
 * @order: The order of the current allocation
 * @alloc_flags: The allocation flags of the current allocation
 * @ac: The context of current allocation
 * @prio: Determines how hard direct compaction should try to succeed
 *
 * This is the main entry point for direct page compaction.
 */
```

```
enum compact_result try_to_compact_pages(gfp_t gfp_mask, unsigned int or
der,
                unsigned int alloc_flags, const struct alloc_context *a
c,
                enum compact_priority prio)
{
        int may_perform_io = gfp_mask & __GFP_IO;
        struct zoneref *z;
        struct zone *zone;
        enum compact_result rc = COMPACT_SKIPPED;

        /*
         * Check if the GFP flags allow compaction - GFP_NOIO is really
         * tricky context because the migration might require IO
         */
        if (!may_perform_io)
                return COMPACT_SKIPPED;

        trace_mm_compaction_try_to_compact_pages(order, gfp_mask, prio);

        /* Compact each zone in the list */
        for_each_zone_zonelist_nodemask(zone, z, ac->zonelist, ac->high_
zoneidx,
                                                                ac->node
mask) {
                enum compact_result status;

                if (prio > MIN_COMPACT_PRIORITY
25                                          && compaction_deferred(zone, order)) {
26                         rc = max_t(enum compact_result, COMPACT_DEFERRED, rc);
27                         continue;
28                 }
29 
30                 status = compact_zone_order(zone, order, gfp_mask, prio,
31                                         alloc_flags, ac_classzone_idx(ac));
32                 rc = max(status, rc);
33 
34                 /* The allocation should succeed, stop compacting */
35                 if (status == COMPACT_SUCCESS) {
36                         /*
37                          * We think the allocation will succeed in this zone,
38                          * but it is not certain, hence the false. The caller
39                          * will repeat this with true if allocation indeed
40                          * succeeds in this zone.
41                          */
42                         compaction_defer_reset(zone, order, false);
43 
44                         break;
45                 }
46 
47                 if (prio != COMPACT_PRIO_ASYNC && (status == COMPACT_COMPLETE ||
48                                         status == COMPACT_PARTIAL_SKIPPED))
49                         /*
50                          * We think that allocation won't succeed in this zone
51                          * so we defer compaction there. If it ends up
52                          * succeeding after all, it will be reset.
53                          */
54                         defer_compaction(zone, order);
55 
56                 /*
57                  * We might have stopped compacting due to need_resched() in
58                  * async compaction, or due to a fatal signal detected. In that
59                  * case do not try further zones
60                  */
61                 if ((prio == COMPACT_PRIO_ASYNC && need_resched())
62                                         || fatal_signal_pending(current))
63                         break;
64         }
65 
66         return rc;
67 }
```

Attempts a compact for the request order and returns the compact progress.

- In the process of compaction on lines 14~15 of code, migration causes io. Therefore, if there is no IO grant request, the compaction cannot proceed, so it returns a COMPACT_SKIPPED.
- If the code line 20~28 is traversed for zones that are high_zoneidx or less than the nodemask specified in the zonelist, and if the compaction priority is not the highest level, it is highly likely that the previous compaction failed in that zone, and the compaction will not succeed even if it is performed immediately, so in this attempt it will be skipped to defer it.

- If the comaction result is a success in the traversing zone on code lines 30~45, reset the suspend flag and return the result.
- In line 47~54 of the code, if the compaction is operating in a non-asynchronous mode, and the result of the compaction is complete or partial skipped, the traversing zone is suspended.
- If a compaction is in progress asynchronously on lines 61~63 of code, the function will exit with the current result if there is a preemption request from another task or if a fatal signal is entered into the current task.

Zone compaction for order

compact_zone_order()

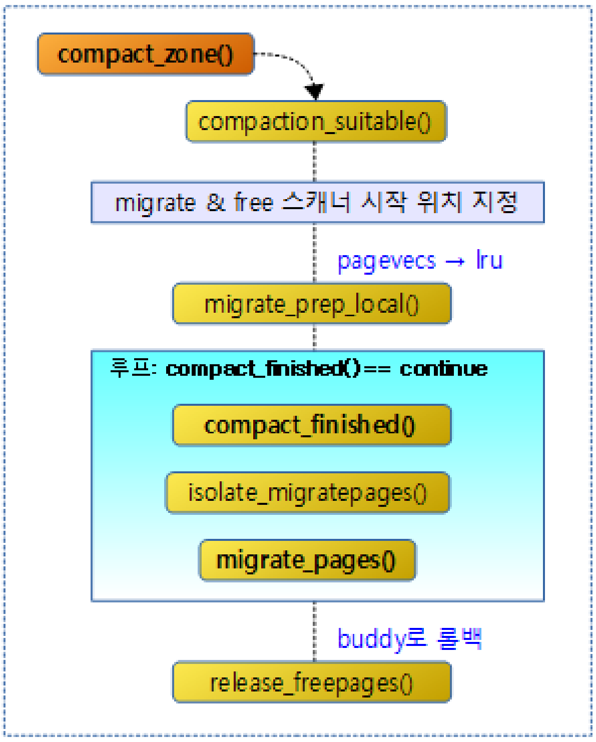
mm/compaction.c

```
static enum compact_result compact_zone_order(struct zone *zone, int order,
                gfp_t gfp_mask, enum compact_priority prio,
                unsigned int alloc_flags, int classzone_idx)
{
        enum compact_result ret;
        struct compact_control cc = {
                .nr_freepages = 0,
                .nr_migratepages = 0,
                .total_migrate_scanned = 0,
                .total_free_scanned = 0,
                .order = order,
                .gfp_mask = gfp_mask,
                .zone = zone,
                .mode = (prio == COMPACT_PRIO_ASYNC) ?
                                        MIGRATE_ASYNC : MIGRATE_SYNC_LIGHT,
                .alloc_flags = alloc_flags,
                .classzone_idx = classzone_idx,
                .direct_compaction = true,
                .whole_zone = (prio == MIN_COMPACT_PRIORITY),
                .ignore_skip_hint = (prio == MIN_COMPACT_PRIORITY),
                .ignore_block_suitable = (prio == MIN_COMPACT_PRIORITY)
        };
        INIT_LIST_HEAD(&cc.freepages);
        INIT_LIST_HEAD(&cc.migratepages);

        ret = compact_zone(zone, &cc);

        VM_BUG_ON(!list_empty(&cc.freepages));
        VM_BUG_ON(!list_empty(&cc.migratepages));

        return ret;
}
```

After preparing the compact_control_cc struct, it performs compact with the requested zone, order, and migrate modes and returns the result.

John compaction

The following figure shows the processing process of the compact_zone() function.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compact_zone-1.png)

compact_zone()

mm/compaction.c -1/3-

```
static enum compact_result compact_zone(struct zone *zone, struct compact_control *cc)
{
        enum compact_result ret;
        unsigned long start_pfn = zone->zone_start_pfn;
        unsigned long end_pfn = zone_end_pfn(zone);
        const bool sync = cc->mode != MIGRATE_ASYNC;

        cc->migratetype = gfpflags_to_migratetype(cc->gfp_mask);
        ret = compaction_suitable(zone, cc->order, cc->alloc_flags,
                                                        cc->classzone_idx);
        /* Compaction is likely to fail */
        if (ret == COMPACT_SUCCESS || ret == COMPACT_SKIPPED)
                return ret;

        /* huh, compaction_suitable is returning something unexpected */
        VM_BUG_ON(ret != COMPACT_CONTINUE);

        /*
         * Clear pageblock skip if there were failures recently and compaction
         * is about to be retried after being deferred.
         */
        if (compaction_restarting(zone, cc->order))
                __reset_isolation_suitable(zone);

        /*
         * Setup to move all movable pages to the end of the zone. Used cached
         * information on where the scanners should start (unless we explicitly
28          * want to compact the whole zone), but check that it is initialised
29          * by ensuring the values are within zone boundaries.
30          */
31         if (cc->whole_zone) {
32                 cc->migrate_pfn = start_pfn;
33                 cc->free_pfn = pageblock_start_pfn(end_pfn - 1);
34         } else {
35                 cc->migrate_pfn = zone->compact_cached_migrate_pfn[sync];
36                 cc->free_pfn = zone->compact_cached_free_pfn;
37                 if (cc->free_pfn < start_pfn || cc->free_pfn >= end_pfn) {
38                         cc->free_pfn = pageblock_start_pfn(end_pfn - 1);
39                         zone->compact_cached_free_pfn = cc->free_pfn;
40                 }
41                 if (cc->migrate_pfn < start_pfn || cc->migrate_pfn >= end_pfn) {
42                         cc->migrate_pfn = start_pfn;
43                         zone->compact_cached_migrate_pfn[0] = cc->migrate_pfn;
44                         zone->compact_cached_migrate_pfn[1] = cc->migrate_pfn;
45                 }
46 
47                 if (cc->migrate_pfn == start_pfn)
48                         cc->whole_zone = true;
49         }
50 
51         cc->last_migrated_pfn = 0;
52 
53         trace_mm_compaction_begin(start_pfn, cc->migrate_pfn,
54                                 cc->free_pfn, end_pfn, sync);
55 
56         migrate_prep_local();
```

Attempts a compact for the request order and returns the compact progress.

- In code lines 4~5, the compaction is from the beginning pfn of the zone to the end pfn.
- In line 6 of the code comes the migraton to find out whether it sinks or not.
- In line 8 of code, use the gfp flag to get the migrate type.
  - unmovable(0), movable(1), reclaimable(2)
- In line 9~13 of the code, we know the result of whether to proceed with the compaction, and if there is already a page to assign or if it is a skipped result, we exit the function without compaction.
- In lines 22~23 of code, if the maximum number of compaction suspensions (63) is reached, clear all PB_migrate_skip bits in the zone's usemap(pageblock_flags) to start the compaction from scratch.
- At the beginning of the code line 31~33, the start of the migrate scanner is set to the start of the zone pfn, and the start of the free scanner is set to the end pfn of the zone.
- If you need to run back-to-back to the last compactin on lines 34~36, the migrate scanner and the free scanner should continue at the last pfn position processed.
- In line 37~40 of the code, if the PFN of the free scanner is out of the zone range, it will be moved back to the page corresponding to the end block of the zone.
- In lines 41~45 of the code, if the pfn of the migrate scanner is out of range of the zone, it is moved back to the page corresponding to the start block of the zone.
  - There are two caches that remember the migrate pfn location: async(0) and sync(1).

- In code lines 47~48, if the pfn of the migrate scanner is in the starting position, whole_zone is true.
- Reset the last Migrated pfn to 51 on line 0 of code.
- In line 52 of code, the local cpu does what it would do before starting Migrate.
  - Drain pages from the LRU cache, Pagevec, into LRU.

mm/compaction.c -2/3-

```
01 .        while ((ret = compact_finished(zone, cc)) == COMPACT_CONTINUE) {
02                  int err;
03
04                  switch (isolate_migratepages(zone, cc)) {
05                  case ISOLATE_ABORT:
06                          ret = COMPACT_CONTENDED;
07                          putback_movable_pages(&cc->migratepages);
08                          cc->nr_migratepages = 0;
09                          goto out;
10                  case ISOLATE_NONE:
11                          /*
12                           * We haven't isolated and migrated anything, but
13                           * there might still be unflushed migrations from
14                           * previous cc->order aligned block.
15                           */
16                          goto check_drain;
17                  case ISOLATE_SUCCESS:
18                          ;
19                  }
20
21                  err = migrate_pages(&cc->migratepages, compaction_alloc,
22                                  compaction_free, (unsigned long)cc, cc->mode,
23                                  MR_COMPACTION);
24
25                  trace_mm_compaction_migratepages(cc->nr_migratepages, err,
26                                                          &cc->migratepages);
27
28                  /* All pages were either migrated or will be released */
29                  cc->nr_migratepages = 0;
30                  if (err) {
31                          putback_movable_pages(&cc->migratepages);
32                          /*
33                           * migrate_pages() may return -ENOMEM when scanners meet
34                           * and we want compact_finished() to detect it
35                           */
36                          if (err == -ENOMEM && !compact_scanners_met(cc)) {
37                                  ret = COMPACT_CONTENDED;
38                                  goto out;
39                          }
40                          /*
41                           * We failed to migrate at least one page in the current
42                           * order-aligned block, so skip the rest of it.
43                           */
44                          if (cc->direct_compaction &&
45                                                  (cc->mode == MIGRATE_ASYNC)) {
46                                  cc->migrate_pfn = block_end_pfn(
47                                                      cc->migrate_pfn - 1, cc->order);
48                                      /* Draining pcplists is useless in this case */
49                                      cc->last_migrated_pfn = 0;
50
51                              }
52                      }
```

- Perform compact on line 1 of code and loop while the result is COMPACT_CONTINUE.
- If the result of isorating the page in line 4~9 is ISOLATE_ABORT, change the compaction result to COMPACT_CONTENDED, reset the migrate pages, clear the number of migrate pages to 0, and exit the function with the out label.
- If the isolation result in lines 10~16 is ISOLATE_NONE, then no page is isolated, and in this case, it goes to the check_drain label to drain the CPU cache and continues the loop.
- If the result is ISOLATE_SUCCESS on lines 17~19 of code, continue with the next routine for migration.
- In lines 21~23 of code, migrate the page that the migrate scanner points to to the page that the free scanner points to.
- This is the case when migration fails on code lines 30~31. Put the pages you want to migrate back to their original location.
- In line 36~39 of the code, if the scanning is not complete and there is not enough memory, the function exits by putting the COMPACT_CONTENDED as a compaction result and moving it to the out label.
- If you request direct-compliance with async on lines 44~51 of code, it will skip the migrate block that is being processed.

mm/compaction.c -3/3-

```
01 check_drain:
02                 /*
03                  * Has the migration scanner moved away from the previous
04                  * cc->order aligned block where we migrated from? If yes,
05                  * flush the pages that were freed, so that they can merge and
06                  * compact_finished() can detect immediately if allocation
07                  * would succeed.
08                  */
09                 if (cc->order > 0 && cc->last_migrated_pfn) {
10                         int cpu;
11                         unsigned long current_block_start =
12                                 block_start_pfn(cc->migrate_pfn, cc->order);
13
14                         if (cc->last_migrated_pfn < current_block_start) {
15                                 cpu = get_cpu();
16                                 lru_add_drain_cpu(cpu);
17                                 drain_local_pages(zone);
18                                 put_cpu();
19                                 /* No more flushing until we migrate again */
20                                 cc->last_migrated_pfn = 0;
21                         }
22                      }
23
24              }
25
26      out:
27              /*
28               * Release free pages and update where the free scanner should restart,
29               * so we don't leave any returned pages behind in the next attempt.
30               */
31              if (cc->nr_freepages > 0) {
32                      unsigned long free_pfn = release_freepages(&cc->freepages);
33
34                      cc->nr_freepages = 0;
35                      VM_BUG_ON(free_pfn == 0);
36                      /* The cached pfn is always the first in a pageblock */
37                      free_pfn = pageblock_start_pfn(free_pfn);
38                      /*
39                       * Only go back, not forward. The cached pfn might have been
40                       * already reset to zone end in compact_finished()
41                       */
42                      if (free_pfn > zone->compact_cached_free_pfn)
43                              zone->compact_cached_free_pfn = free_pfn;
44              }
45
46              count_compact_events(COMPACTMIGRATE_SCANNED, cc->total_migrate_scanned);
47              count_compact_events(COMPACTFREE_SCANNED, cc->total_free_scanned);
48
49              trace_mm_compaction_end(start_pfn, cc->migrate_pfn,
50                                      cc->free_pfn, end_pfn, sync, ret);
51
52              return ret;
53      }
```

- check_drain in line 1 of code: The label determines whether to empty the LRU cache, the PageveC.
- In line 9~22 of code, if the request order is not 0 and the last migrate pfn exists under the current running block, emptying the LRU cache Pagevec will increase the likelihood of merging and the compact_finished() will be able to detect the success of the allocation immediately.
- On code lines 26~44, the out: label is. Revert the free pages for the free scanner and remember their location in the cache.
- Update the COMPACTMIGRATE_SCANNED and COMPACTFREE_SCANNED counters on lines 46~47 of the code.

release_freepages()

mm/compaction.c

```
01      static unsigned long release_freepages(struct list_head *freelist)
02      {
03              struct page *page, *next;
04              unsigned long high_pfn = 0;
05
06              list_for_each_entry_safe(page, next, freelist, lru) {
07                      unsigned long pfn = page_to_pfn(page);
08                      list_del(&page->lru);
```

```
                __free_page(page);
                if (pfn > high_pfn)
                        high_pfn = pfn;
        }

        return high_pfn;
}
```

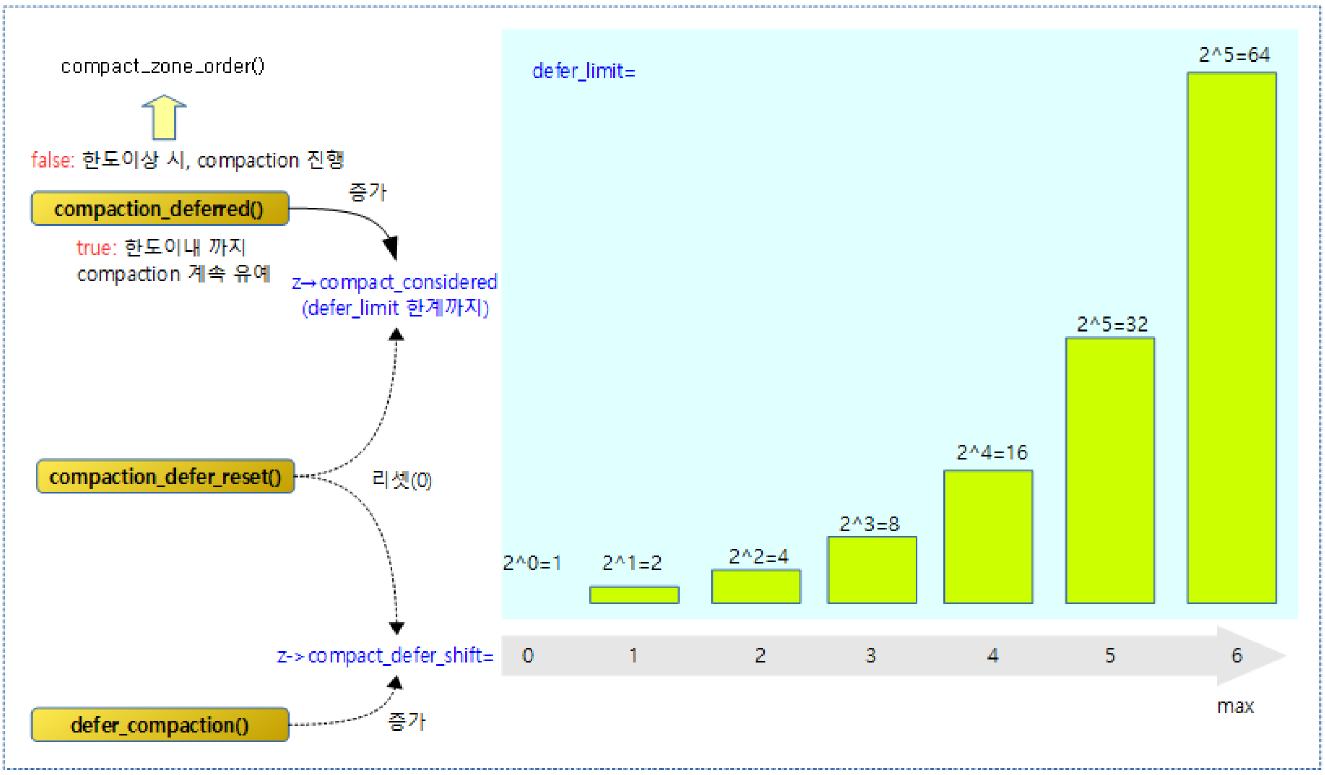
Remove all pages from the freelist, and return the largest pfn value.

Compaction reprieve

It knows whether to suspend compaction for the zone, for the purpose of increasing the suspension counter and suspending the compaction until it reaches its limit. The grace counter (compact_considered) can be increased at each compact_defer_shift level, up to a maximum of 64.

The following figure shows the use of three functions to increment or reset for the suspend counter and the suspend shift counter.

- If the result of the compaction_deferred() function is true, it will suspend the compaction immediately.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction_deferred-1a.png)

compaction_deferred()

mm/compaction.c

```
/* Returns true if compaction should be skipped this time */
bool compaction_deferred(struct zone *zone, int order)
```

```
03 {
04         unsigned long defer_limit = 1UL << zone->compact_defer_shift;
05 
06         if (order < zone->compact_order_failed)
07                 return false;
08 
09         /* Avoid possible overflow */
10         if (++zone->compact_considered > defer_limit)
11                 zone->compact_considered = defer_limit;
12 
13         if (zone->compact_considered >= defer_limit)
14                 return false;
15 
16         trace_mm_compaction_deferred(zone, order);
17 
18         return true;
19 }
```

At this time, the compaction is deferred and returns whether it should be skipped. (true=compaction suspended, false=compaction suspended)

- In line 6~7 of the code, if the order request is smaller than the failed order value used in the last compaction, it will return false because it will have to try the compaction again.
- Increase the grace counter (compact_considered) in the code line 10~18 zone. Below the maximum grace limit (1 << compact_defer_shift), it returns true to suspend compactin. If the maximum grace limit is exceeded, it will return false to attempt compaction.

defer_compaction()

mm/compaction.c

```
/*
 * Compaction is deferred when compaction fails to result in a page
 * allocation success. 1 << compact_defer_limit compactions are skipped up
 * to a limit of 1 << COMPACT_MAX_DEFER_SHIFT
 */
void defer_compaction(struct zone *zone, int order)
{
        zone->compact_considered = 0;
        zone->compact_defer_shift++;

        if (order < zone->compact_order_failed)
                zone->compact_order_failed = order;

        if (zone->compact_defer_shift > COMPACT_MAX_DEFER_SHIFT)
                zone->compact_defer_shift = COMPACT_MAX_DEFER_SHIFT;

        trace_mm_compaction_defer_compaction(zone, order);
}
```

Each time a compaction is completed without allocating the order page in the requested zone, the grace counter is reset to 0, and the grace limit counter is increased to 1, 2, 4, 8, 16, 32, and 64.

compaction_defer_reset()

mm/compaction.c

```
/*
 * Update defer tracking counters after successful compaction of given order,
 * which means an allocation either succeeded (alloc_success == true) or is
 * expected to succeed.
 */
```

```
01 void compaction_defer_reset(struct zone *zone, int order,
02                 bool alloc_success)
03 {
04         if (alloc_success) {
05                 zone->compact_considered = 0;
06                 zone->compact_defer_shift = 0;
07         }
08         if (order >= zone->compact_order_failed)
09                 zone->compact_order_failed = order + 1;
10
11         trace_mm_compaction_defer_reset(zone, order);
12 }
```

This function is called when a compaction is performed in the requested zone and the order page is expected to succeed. If it is called on the actual successful page allocation, it resets the grace counter and the grace limit counter to zero.

- If the assignment of the order page succeeds after performing compaction in the zone requested in code lines 4~7, the suspend counter and suspend limit counter are reset to 0.
- In line 8~9 of the code, set the failed order value to the requested order + 1 value.

compaction_restarting()

mm/compaction.c

```
/* Returns true if restarting compaction after many failures */
bool compaction_restarting(struct zone *zone, int order)
{
        if (order < zone->compact_order_failed)
                return false;

        return zone->compact_defer_shift == COMPACT_MAX_DEFER_SHIFT &&
                zone->compact_considered >= 1UL << zone->compact_defer_shift;
}
```

Returns true if the maximum number of repears (64) has been reached.

- Returns false if the request order is less than compact_order_failed
- Returns true if the compact_defer_shift is last (6) and the compact_considered value is 64 or greater.

Compact Shutdown Check

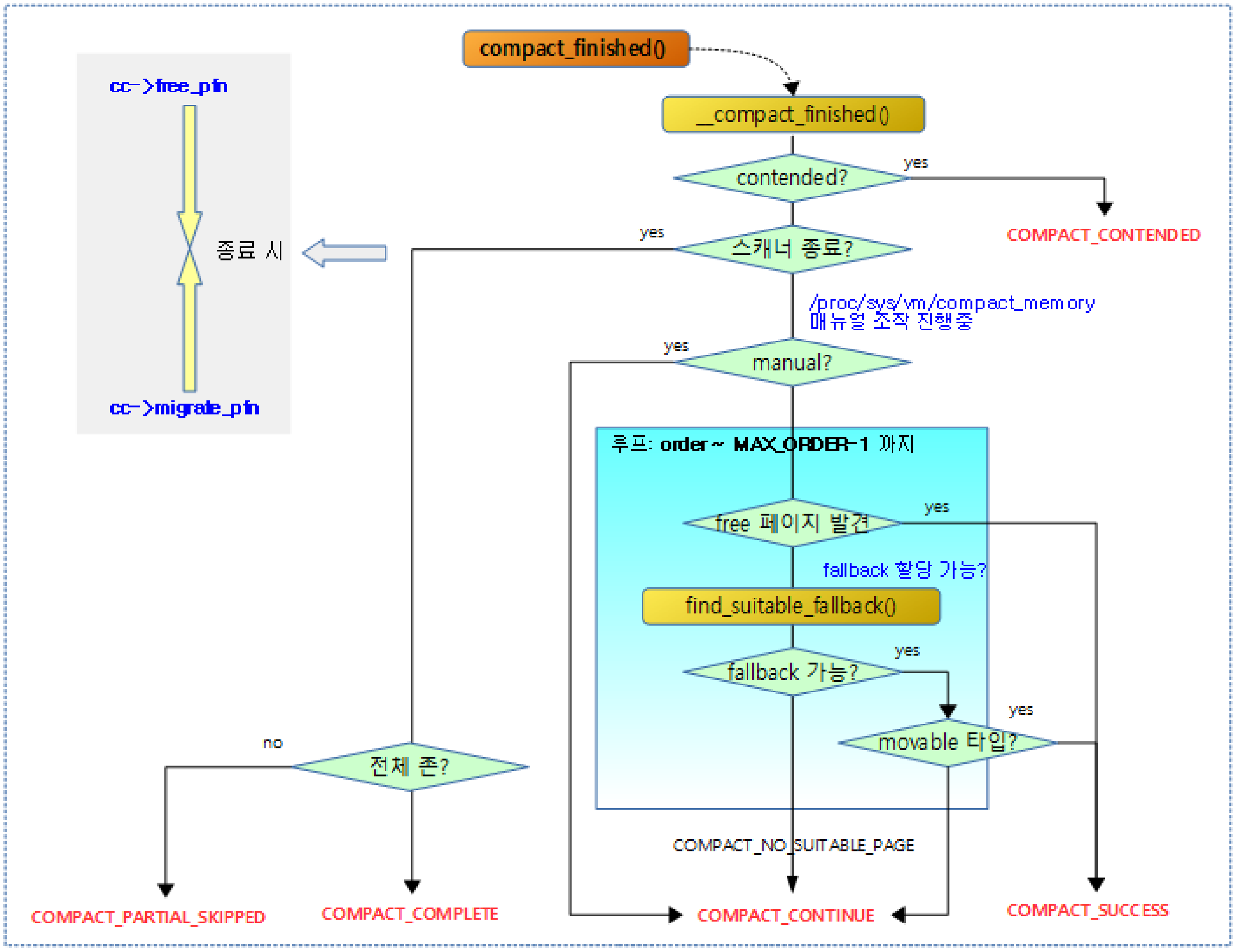
compact_finished()

mm/compaction.c

```
static enum compact_result compact_finished(struct zone *zone,
                        struct compact_control *cc)
{
        int ret;

        ret = __compact_finished(zone, cc);
        trace_mm_compaction_finished(zone, cc->order, ret);
        if (ret == COMPACT_NO_SUITABLE_PAGE)
                ret = COMPACT_CONTINUE;

        return ret;
}
```

Returns the progress to determine whether the compact is complete.

The following diagram shows the progress of the compact to determine whether it is complete.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compact_finished-1b.png)

__compact_finished()

mm/compaction.c -1/2-

```
static enum compact_result __compact_finished(struct zone *zone,
                                              struct compact_control *cc)
{
        unsigned int order;
        const int migratetype = cc->migratetype;

        if (cc->contended || fatal_signal_pending(current))
                return COMPACT_CONTENDED;

```

```
10          /* Compaction run completes if the migrate and free scanner meet */
11          if (compact_scanners_met(cc)) {
12                  /* Let the next compaction start anew. */
13                  reset_cached_positions(zone);
14
15                  /*
16                   * Mark that the PG_migrate_skip information should be cleared
17                   * by kswapd when it goes to sleep. kcompactd does not set the
18                   * flag itself as the decision to be clear should be directly
19                   * based on an allocation request.
20                   */
21                  if (cc->direct_compaction)
22                          zone->compact_blockskip_flush = true;
23
24                  if (cc->whole_zone)
25                          return COMPACT_COMPLETE;
26                  else
27                          return COMPACT_PARTIAL_SKIPPED;
28          }
29
30          if (is_via_compact_memory(cc->order))
31                  return COMPACT_CONTINUE;
32
33          if (cc->finishing_block) {
34                  /*
35                   * We have finished the pageblock, but better check again that
36                   * we really succeeded.
37                   */
38                  if (IS_ALIGNED(cc->migrate_pfn, pageblock_nr_pages))
39                          cc->finishing_block = false;
40                  else
41                          return COMPACT_CONTINUE;
42          }
```

- In line 7~8 of the code, if the current task has a preemption request or fatal signal to be urgently processed during compaction, it returns a COMPACT_CONTENDED.
- If the two free scanners and the migrate scanner meet in code lines 11~28, the compaction is complete. Reset the start of the scan for the next scan. Returns a COMPACT_COMPLETE if the entire zone has been scanned, or a COMPACT_PARTIAL_SKIPPED if only a few zones have been scanned.
- In line 30~31 of the code, if the user intervenes and performs a compaction, it returns a COMPACT_CONTINUE to force the entire block.
  - "echo 1 > /proc/sys/vm/compact_memory"
- This is the case when a page block is completed on lines 33~42 of the code. Double-check to see if the migrate scanner has indeed finished a block of pages, and if so, return a COMPACT_CONTINUE to continue.
  - Note: mm, compaction: finish whole pageblock to reduce fragmentation (https://github.com/torvalds/linux/commit/baf6a9a1db5a40ebfa5d3e761428d3deb2cc3a3b#diff-a187bd6389cbd112c7e026c401962224)

mm/compaction.c -2/2-

```
01          /* Direct compactor: Is a suitable page free? */
```

```
02          for (order = cc->order; order < MAX_ORDER; order++) {
03                  struct free_area *area = &zone->free_area[order];
04                  bool can_steal;
05
06                  /* Job done if page is free of the right migratetype */
07                  if (!list_empty(&area->free_list[migratetype]))
08                          return COMPACT_SUCCESS;
09
10  #ifdef CONFIG_CMA
11                  /* MIGRATE_MOVABLE can fallback on MIGRATE_CMA */
12                  if (migratetype == MIGRATE_MOVABLE &&
13                          !list_empty(&area->free_list[MIGRATE_CMA]))
14                          return COMPACT_SUCCESS;
15  #endif
16                  /*
17                   * Job done if allocation would steal freepages from
18                   * other migratetype buddy lists.
19                   */
20                  if (find_suitable_fallback(area, order, migratetype,
21                                                  true, &can_steal) != -1) {
22
23                          /* movable pages are OK in any pageblock */
24                          if (migratetype == MIGRATE_MOVABLE)
25                                  return COMPACT_SUCCESS;
26
27                          /*
28                           * We are stealing for a non-movable allocation. Make
29                           * sure we finish compacting the current pageblock
30                           * first so it is as free as possible and we won't
31                           * have to steal another one soon. This only applies
32                           * to sync compaction, as async compaction operates
33                           * on pageblocks of the same migratetype.
34                           */
35                          if (cc->mode == MIGRATE_ASYNC ||
36                                          IS_ALIGNED(cc->migrate_pfn,
37                                                          pageblock_nr_pages)) {
38                                  return COMPACT_SUCCESS;
39                          }
40
41                          cc->finishing_block = true;
42                          return COMPACT_CONTINUE;
43                  }
44          }
45
46          return COMPACT_NO_SUITABLE_PAGE;
47  }
```

Make sure that the required free pages can be secured.

- In line 2~8 of the code, it traverses from the @order to the last order, and returns a COMPACT_SUCCESS if a free page is found in the free list of the order.
- In the case of a movable page request in line 12~14 of the code, if a free page is found in the CMA type list, it returns a COMPACT_SUCCESS.
- In line 20~25 of the code, if there is a free page to be imported from another type, it returns COMPACT_SUCCESS if it is of movable type.
- On lines 35~39 of code, if the compaction is performing an aync, or if the migrate scanner has completed a one-page block, it returns a COMPACT_SUCCESS.

- Returns COMPACT_CONTINUE in lines 41~42 of code. Then, change the finishing_block to true so that the next compaction will investigate whether it will continue to compact until the page block is complete without interrupting.
- At line 46 of code, page acquisition failed, so returns COMPACT_NO_SUITABLE_PAGE.

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) | Qc
- 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) | Qc
- Zoned Allocator -6- (Watermark) (http://jake.dothome.co.kr/zonned-allocator-watermark) | Qc
- Zoned Allocator -7- (Direct Compact) (http://jake.dothome.co.kr/zonned-allocator-compaction) | Sentence C – Current post
- Zoned Allocator -8- (Direct Compact-Isolation) (http://jake.dothome.co.kr/zonned-allocator-isolation) | Qc
- Zoned Allocator -9- (Direct Compact-Migration) (http://jake.dothome.co.kr/zonned-allocator-migration) | Qc
- Zoned Allocator -10- (LRU & pagevec) (http://jake.dothome.co.kr/lru-lists-pagevecs) | Qc
- Zoned Allocator -11- (Direct Reclaim) (http://jake.dothome.co.kr/zonned-allocator-reclaim) | Qc
- Zoned Allocator -12- (Direct Reclaim-Shrink-1) (http://jake.dothome.co.kr/zonned-allocator-shrink-1) | Qc
- Zoned Allocator -13- (Direct Reclaim-Shrink-2) (http://jake.dothome.co.kr/zonned-allocator-shrink-2) | Qc
- Zoned Allocator -14- (Kswapd) (http://jake.dothome.co.kr/zonned-allocator-kswapd) | Qc

- Memory compaction (https://lwn.net/Articles/368869/) (2010) | LWN.net
- Linux: Memory fragmentation and compaction (https://www.uninformativ.de/blog/postings/2017-12-23/0/POSTING-en.html) | uninformativ.de

10 thoughts to “Zoned Allocator -7- (Direct Compact)”

PETER

2020-12-20 09:34 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-296175)

Hello

There is a typo in the calculation in the __fragmentation_index() example figure above (fragmentation_index-1a-1).

Second example from left 1000 – (1500+1000)/10 = 1000-250 =750 Third example

1000 – (1500+1000
)/4 = 1000-625 =375
.

In other words, the second 750 is closer to 375 than the third 1000, so the allocation is more likely to succeed than the third.

I appreciate it.

RESPONSE (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=296175#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-12-21 06:57 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-296365)

Hello? THANKS TO PETER FOR TELLING ME, I CONFIRMED THAT THERE WAS AN ERROR.
The illustration has been re-corrected, and the erroneous interpretation has been removed.
Thanks for finding the error. Merry Christmas ^^

RESPONSE (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=296365#RESPOND)

KWON YONGBEOM
2020-12-23 18:13 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-296878)

Hello
"The closer it is to zero, the better it is not to allow compaction due to insufficient memory."
I don't really understand this either. If we talk about the lack of memory free_pages, the free_pages in the
expression will be close to zero.
Thank you as always.

RESPONSE (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=296878#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-12-24 08:37 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-297030)

Hello?

This is the point where the desired order page cannot be assigned, so you need to determine whether it is compaction.
At the extreme, you have a 50M free page, but you may not be able to allocate 1M memory, so you may need to do compaction.
In this case, the focus is not on whether there is a shortage of memory, but on how much is scattered.
So, even if they are in the same low memory state, if they are scattered, I think it is useful to collect them and make a compaction.

First, the number of free pages is always greater than or equal to the number of free blocks, and the larger the difference, the greater the fragmentation factor, and

it is assumed that the scattered memory can be collected and allocated through compaction.

The fragmentation coefficient is as follows:
The closer to 0 it is -> it is really impossible to allocate due to lack of memory, the closer it is to
1000 -- the more impossible it is to allocate because of > fragmentation.

I appreciate it.

RESPONSE (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=297030#RESPOND)

KWON YONGBEOM
2020-12-24 16:55 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-297088)

I'm stuck with the word 'out of memory' and I still don't quite understand it. I judge the meaning of this word based on the number of free_pages, and
I think the meaning of what you said is a little different. If you want to allocate 2M (order 9) blocks, but there are
two blocks of 1M (order 8), you can theoretically allocate them by compaction. In this case, the fragmentation coefficient will be zero.
What you said really means of lack of memory keeps me confused.
I appreciate it.

RESPONSE (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=297088#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-12-24 22:11 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-297140)

It seems that the confusion comes from changing the number of free pages.
If you're running out of memory, the number of free pages is already small, and you need to use this as a benchmark to see how many free pages are scattered in free blocks.

For example, if you have 1024 free_pages(4M) and the allocation is 512(2M) pages, and you get the fragmentation factor by the number of free blocks that fail and are scattered to compaction,
you get the following:
1000 – (1024 * 1000 / 512 + 1000) / 3 = 0
1000 – (1024 * 1000 / 512 + 1000) / 4 = 250
1000 – (1024 * 1000 / 512 + 1000) / 5 = 400
1000 – (1024 * 1000 / 512 + 1000) / 6 = 500
1000 – (1024 * 1000 / 512 + 1000) / 7 = 571
1000 – (1024 * 1000 / 512 + 1000) / 10 = 700
1000 – (1024 * 1000 / 512 + 1000) / 30 = 900
1000 – (1024 * 1000 / 512 + 1000) / 300 = 970

Based on the specific amount of free pages that are lacking, only the free blocks need to be changed in order to compare the relative fragmentation coefficients.
(Note that compaction does not change the free page, only the number of blocks.)

If the fragmentation factor is large, try compaction, and if that doesn't work, get a free page through a separate memory reclaim.

Merry Christmas!

RESPONSE (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=297140#RESPOND)

KWON YONGBEOM
2020-12-24 23:31 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-297153)

Oh, I guess I'm lacking in understanding. In the example above, the free_pages with a fragmentation factor of 0 or 970 is calculated based on the time when there is the same amount of memory left, so
why are you saying that only the one with 4 is out of memory?
I think you categorized the one that doesn't have a compact because the fragmentation coefficient is low memory, but I think I bothered you too much on the eve.
Have a good year-end and be healthy~

RESPONSE (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=297153#RESPOND)

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

2020-12-25 19:43 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-297272)

The __fragmentation_index() function has the following annotation:

```
 /*
* Index is between 0 and 1 so return within 3 decimal places
*
* 0 => allocation would fail due to lack of memory
* 1 => allocation would fail due to fragmentation
*/
```

위의 주석 설명문은 다음과 같이 단순하게 설명하였습니다.
0 => 메모리 부족으로 할당이 실패할 수 있다.
1 => 단편화로 인해 할당이 실패할 수 있다.

다만 위의 주석 설명문을 조금 더 풀어 설명드리면,
0 => 메모리가 부족한 상태이고, 단편화 해결과 관련 없이 요청한 페이지의 확보가 실패할 수 있다.
1 => 메모리가 부족한 상태이고, 단편화로 인해 요청한 페이지의 확보가 실패할 수 있다. 즉 단편화로 인해 흩어진 메모리를 모으는데 성공하면 요청한 페이지의 확보가 성공할 수 있다.

용범님도 연말 잘 보내시길 바랍니다.

응답 (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=297272#RESPOND)

권용범
2020-12-26 17:22 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-297335)

소스 내에서도 0에 가까운 쪽을 lack of memory 로 표현하였군요.
감사합니다.

응답 (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=297335#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2020-12-26 22:06 (http://jake.dothome.co.kr/zonned-allocator-compaction/#comment-297384)

네. 맞습니다.
감사합니다. ^^

응답 (/ZONNED-ALLOCATOR-COMPACTION/?REPLYTOCOM=297384#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

Zoned Allocator -6- (Watermark) (http://jake.dothome.co.kr/zonned-allocator-watermark/)

Zoned Allocator -8- (Direct Compact-Isolation) (http://jake.dothome.co.kr/zonned-allocator-isolation/)