Zoned Allocator -8- (Direct Compact-Isolation)

2016-07-05 (http://jake.dothome.co.kr/zonned-allocator-isolation/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

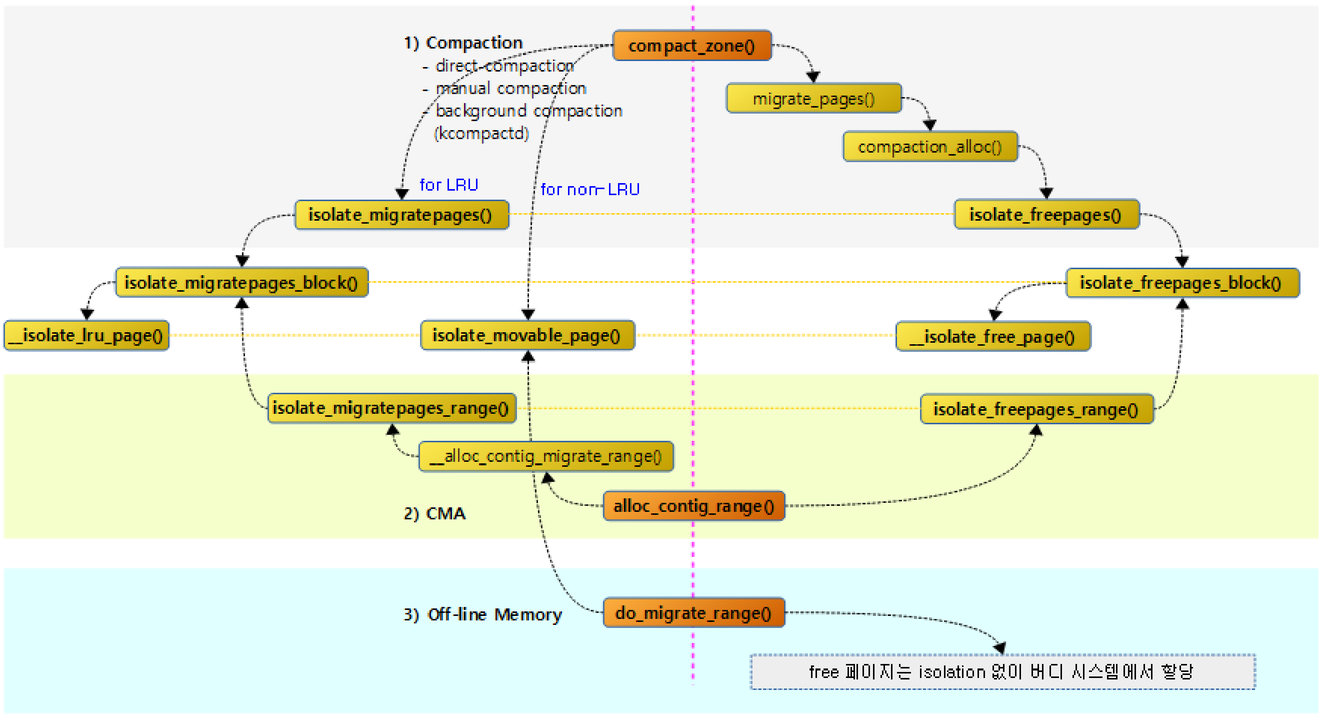
<kernel v5.0>

Zoned Allocator -8- (Direct Compact-Isolation)

Isolation can be used for the following purposes:

- Compaction
  - Migrate migrating migratable pages to the upper part of the zone to make up for the missing high order pages.
  - Direct-compaction, manual-compaction, kcompactd
- Off-line Memory
  - Migrate all in-use movable pages located in the memory area to be offline to another area.
- CMA
  - In order to secure a continuous physical space within the scope requested by the CMA area, the movable pages that are temporarily occupying (?) in the CMA area are migrated.

The following diagram shows the main function call process for isolation in two parts. On the left is the process of isolating the page to be moved, and on the right is the process of isolating or securing the free page.

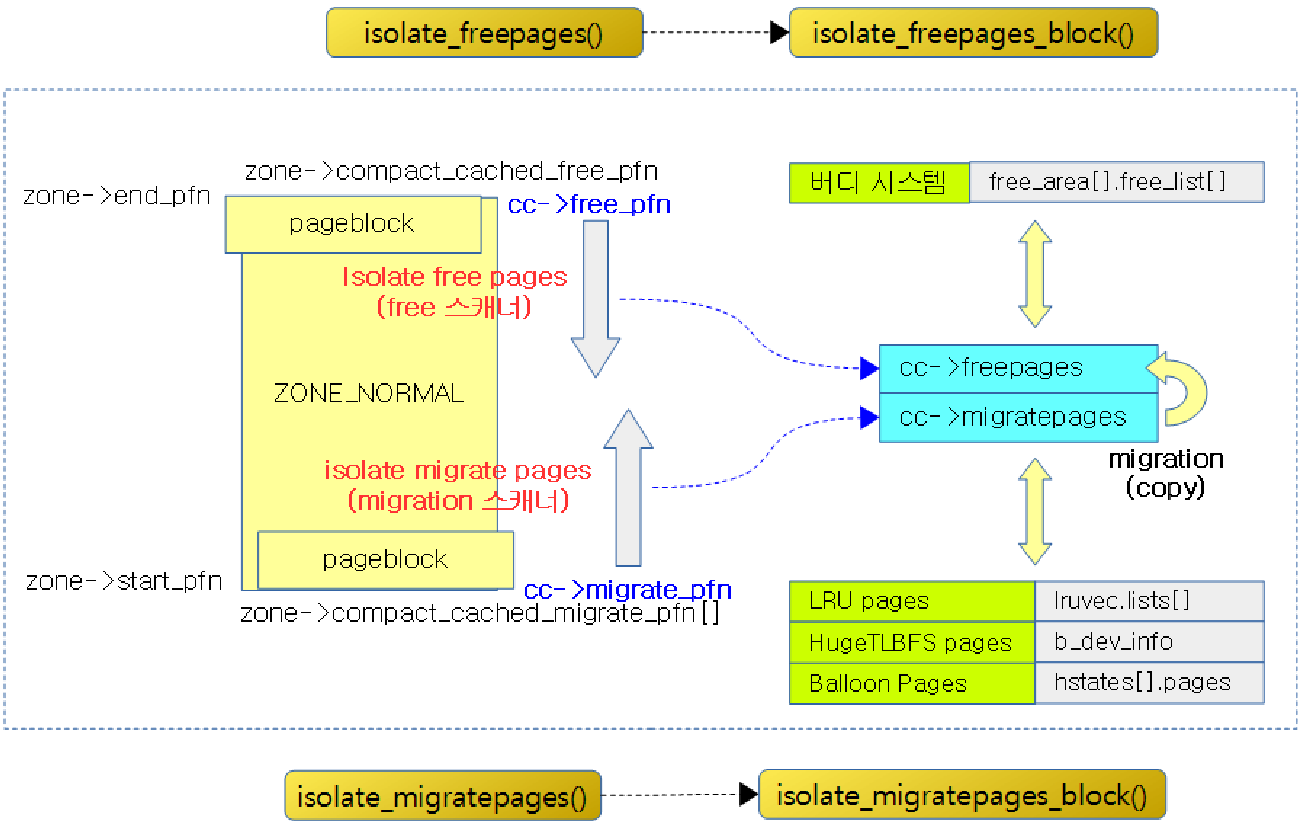


(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction-6b.png)

Two scanners for Compaction

The following diagram shows the direction in which the two migration scanners and the free scanner work when performing a compaction, isolating the pages to be used for their respective purposes.

- The migration scanner scans page blocks upwards to get migratable pages.
  - isolate_migratepages_block() isolates the migratable page from a single selected page block and adds it to the cc->migratepages list.
- The free scanner scans the page blocks in a downward direction to get the free pages.
  - isolate_freepages_block() isolates a free page of type request from a single selected page block and adds it to the cc->freepages list.



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

Migrate Scanner

Isolate migratepages

isolate_migratepages()

mm/compaction.c

```
/*
 * Isolate all pages that can be migrated from the first suitable block,
 * starting at the block pointed to by the migrate scanner pfn within
 * compact_control.
 */
```

```
static isolate_migrate_t isolate_migratepages(struct zone *zone,
                                        struct compact_control *cc)
{
        unsigned long block_start_pfn;
        unsigned long block_end_pfn;
        unsigned long low_pfn;
        struct page *page;
        const isolate_mode_t isolate_mode =
                (sysctl_compact_unevictable_allowed ? ISOLATE_UNEVICTABLE : 0) |
                (cc->mode != MIGRATE_SYNC ? ISOLATE_ASYNC_MIGRATE : 0);

        /*
         * Start at where we last stopped, or beginning of the zone as
         * initialized by compact_zone()
         */
        low_pfn = cc->migrate_pfn;
        block_start_pfn = pageblock_start_pfn(low_pfn);
        if (block_start_pfn < zone->zone_start_pfn)
                block_start_pfn = zone->zone_start_pfn;

        /* Only scan within a pageblock boundary */
        block_end_pfn = pageblock_end_pfn(low_pfn);

        /*
         * Iterate over whole pageblocks until we find the first suitable.
         * Do not cross the free scanner.
         */
        for (; block_end_pfn <= cc->free_pfn;
                        low_pfn = block_end_pfn,
                        block_start_pfn = block_end_pfn,
                        block_end_pfn += pageblock_nr_pages) {

                /*
                 * This can potentially iterate a massively long zone with
                 * many pageblocks unsuitable, so periodically check if we
                 * need to schedule, or even abort async compaction.
                 */
                if (!(low_pfn % (SWAP_CLUSTER_MAX * pageblock_nr_pages))
                                                && compact_should_abort(cc))
                        break;

                page = pageblock_pfn_to_page(block_start_pfn, block_end_pfn,
                                                                        zone);
                if (!page)
                        continue;

                /* If isolation recently failed, do not retry */
                if (!isolation_suitable(cc, page))
                        continue;

                /*
                 * For async compaction, also only scan in MOVABLE blocks.
                 * Async compaction is optimistic to see if the minimum amount
                 * of work satisfies the allocation.
                 */
                if (!suitable_migration_source(cc, page))
                        continue;

                /* Perform the isolation */
60                 low_pfn = isolate_migratepages_block(cc, low_pfn,
61                                                 block_end_pfn, isolate_mode);
62 
63                 if (!low_pfn || cc->contended)
64                         return ISOLATE_ABORT;
65 
66                 /*
67                  * Either we isolated something and proceed with migration. Or
68                  * we failed and compact_zone should decide if we should
69                  * continue or not.
70                  */
71                 break;
72         }
73 
74         /* Record where migration scanner will be restarted. */
75         cc->migrate_pfn = low_pfn;
76 
77         return cc->nr_migratepages ? ISOLATE_SUCCESS : ISOLATE_NONE;
78 }
```

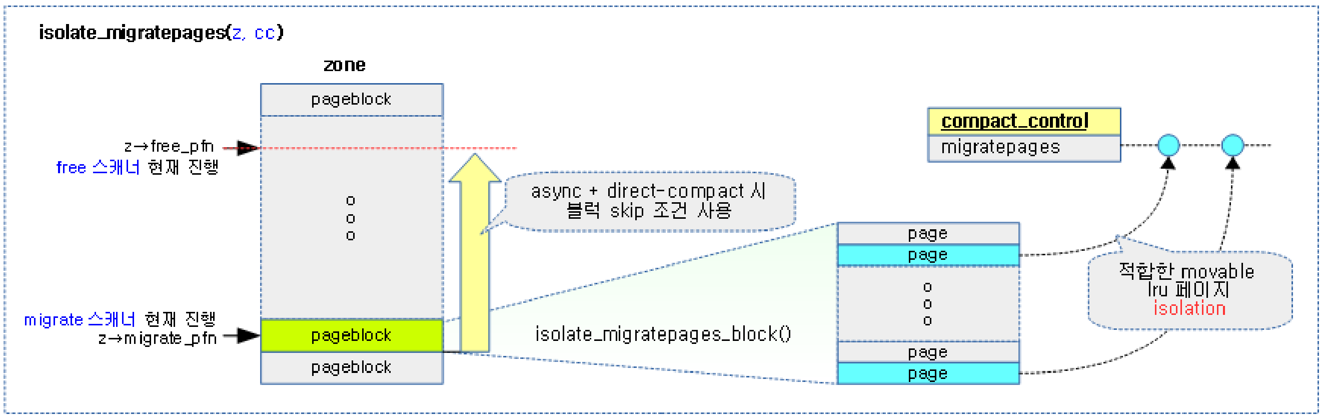
Isolate migratable pages in the requested zone.

- Determine the isolate mode from code lines 8~10.
  - ISOLATE_UNMAPPED(0x2)
    - Only unmapped pages support isolation, and mapped files (file cache, swap cache, etc.) are not processed.
  - ISOLATE_ASYNC_MIGRATE(0x4)
    - Briefly perform isolation and migration. This is done only for page blocks with a migrate type of more than 50% of the blocks, and if 1 order isolation is possible, it will stop without proceeding any further. In addition, it also stops preempt requests.
  - ISOLATE_UNEVICTABLE(0x8)
    - Even unevictable pages have isolation performed.
- In code lines 16~22, find the pfn of the beginning and end of the block to be scanned with the pfn pointing to the location where the migrate scanner wants to scan upwards.
  - e.g. migrate_pfn=0x250, pageblock_order=9
    - block_start_pfn=0x200, block_end_pfn=0x400
- Traverse the Migrate scanner block by block from code lines 28~31 to the location of the free scanner that scans downward.
- Since there is a large scope to be processed in line 38~40 of the code, periodically check for interruptions.
  - Check Frequency
    - SWAP_CLUSTER_MAX (32) page blocks
  - Interrupting Elements
    - Async migration is being processed, but there is a preemption request from a higher priority task.
- In lines 42~45 of the code, get the first page of the page block in the scope of the request zone. If the page leaves the request zone, it returns null and skips.
- In lines 48~49 of code, if you don't want to isolate the block in that zone, skip it.
  - When direct-compact is run in async mode, it returns false to skip the block if the skip bit that is set if isolation has recently failed on that pageblock.

- In line 56~57 of code, when operating in asynchronous direct-compact mode, if the block migrate type is not the same as the requested migrate type, skip it. However, if it is a synchronous (migrate_sync*) request, or if it is a manual and kcompactd request, it will always return true to make an unconditional attempt to isolate the block.
- In code lines 60~61, only the migratable pages are isolated from the pages in that block.
- If isolation fails in line 63~64 or has to be aborted halfway through, it will exit with ISOLATE_ABORT result.
- If isolation completes normally on line 71 of code, it exits the loop.
- Remembers the current position of the migrate scanner in process from code lines 75~77 and returns an isolation result.

The following figure shows how the Migrate scanner uses the isolate_migratepages() function to isolate movable LRU pages against page blocks with conformity conditions.

- lru page isolation (X) -> migratable page suitable movable



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

Isolate migratepages block

The following functions are called and used for two purposes:

- Compaction
  - The isolate_migratepages() function can be called with the addition of the following isolate mode-related flags:
    - If you are not in sync compaction mode, a ISOLATE_ASYNC_MIGRATE is added.
    - If sysctl_compact_unevictable_allowed (default=1) is set, a ISOLATE_UNEVICTABLE mode is added.
- CMA
  - isolate_migratepages_range() function is called in ISOLATE_UNEVICTABLE mode.

isolate_migratepages_block()

mm/compaction.c -1/4-

```
/**
 * isolate_migratepages_block() - isolate all migrate-able pages within
 *                               a single pageblock
 * @cc:          Compaction control structure.
 * @low_pfn:     The first PFN to isolate
 * @end_pfn:     The one-past-the-last PFN to isolate, within same pagebl
ock
 * @isolate_mode: Isolation mode to be used.
 *
 * Isolate all pages that can be migrated from the range specified by
 * [low_pfn, end_pfn). The range is expected to be within same pagebloc
k.
 * Returns zero if there is a fatal signal pending, otherwise PFN of the
 * first page that was not scanned (which may be both less, equal to or
more
 * than end_pfn).
 *
 * The pages are isolated on cc->migratepages list (not required to be e
mpty),
 * and cc->nr_migratepages is updated accordingly. The cc->migrate_pfn f
ield
 * is neither read nor updated.
 */
```

```
isolate_migratepages_block(struct compact_control *cc, unsigned long low
_pfn,
                        unsigned long end_pfn, isolate_mode_t isolate_mo
de)
{
        struct zone *zone = cc->zone;
        unsigned long nr_scanned = 0, nr_isolated = 0;
        struct lruvec *lruvec;
        unsigned long flags = 0;
        bool locked = false;
        struct page *page = NULL, *valid_page = NULL;
        unsigned long start_pfn = low_pfn;
        bool skip_on_failure = false;
        unsigned long next_skip_pfn = 0;

        /*
         * Ensure that there are not too many pages isolated from the LR
U
         * list by either parallel reclaimers or compaction. If there ar
e,
         * delay for some time until fewer pages are isolated
         */
        while (unlikely(too_many_isolated(zone))) {
                /* async migration should just abort */
                if (cc->mode == MIGRATE_ASYNC)
                        return 0;

                congestion_wait(BLK_RW_ASYNC, HZ/10);

                if (fatal_signal_pending(current))
                        return 0;
        }

        if (compact_should_abort(cc))
                return 0;

        if (cc->direct_compaction && (cc->mode == MIGRATE_ASYNC)) {
                skip_on_failure = true;
                next_skip_pfn = block_end_pfn(low_pfn, cc->order);
        }

        /* Time to isolate some pages for migration */
        for (; low_pfn < end_pfn; low_pfn++) {

```

```
41                 if (skip_on_failure && low_pfn >= next_skip_pfn) {
42                         /*
43                          * We have isolated all migration candidates in the
44                          * previous order-aligned block, and did not skip it due
45                          * to failure. We should migrate the pages now and
46                          * hopefully succeed compaction.
47                          */
48                         if (nr_isolated)
49                                 break;
50 
51                         /*
52                          * We failed to isolate in the previous order-aligned
53                          * block. Set the new boundary to the end of the
54                          * current block. Note we can't simply increase
55                          * next_skip_pfn by 1 << order, as low_pfn might have
56                          * been incremented by a higher number due to skipping
57                          * a compound or a high-order buddy page in the
58                          * previous loop iteration.
59                          */
60                         next_skip_pfn = block_end_pfn(low_pfn, cc->order);
61                 }
```

Isolate migratable pages in a one-page block of the requested migratetype.

- In line 5 of the code, initialize the number of scanned pages and the number of isolated pages to zero.
- In line 19~28 of the code, there is a small probability that only if there are too many isolated pages, it will loop around and delay processing by 100ms. However, in the following situations, it aborts processing and returns 0.
  - During asynchronous migration, excessive processing is abandoned.
  - If the processing of the SIGKILL signal is delayed, the function will be aborted.
  - 참고: Avoid the use of congestion_wait under zone pressure (https://lwn.net/Articles/377709/)
- In line 30~31 of the code, if there is a preemption request from a higher priority task during asynchronous processing, the function processing is aborted.
- In line 33~36 of code, if you are processing async migration using direct-compaction, set the skip_on_failure to true.
  - skip_on_failure
    - In order to quickly assign the requested order, the page is searched by order rather than a one-page block, and if there is any isolation, it is terminated.
- Travers the pages in the range to be processed one by one in line 39~61 of the code, but if the skip_on_failure is set and the processing by order unit is completed, specify the end pfn of the next order. If there is any isolated page in the block, it exits the loop.

mm/compaction.c -2/4-

```
01 .               /*
```

```
                 * Periodically drop the lock (if held) regardless of its
                 * contention, to give chance to IRQs. Abort async compaction
                 * if contended.
                 */
                if (!(low_pfn % SWAP_CLUSTER_MAX)
                    && compact_unlock_should_abort(zone_lru_lock(zone), flags,
                                                                &locked, cc))
                        break;

                if (!pfn_valid_within(low_pfn))
                        goto isolate_fail;
                nr_scanned++;

                page = pfn_to_page(low_pfn);

                if (!valid_page)
                        valid_page = page;

                /*
                 * Skip if free. We read page order here without zone lock
                 * which is generally unsafe, but the race window is small and
                 * the worst thing that can happen is that we skip some
                 * potential isolation targets.
                 */
                if (PageBuddy(page)) {
                        unsigned long freepage_order = page_order_unsafe(page);

                        /*
                         * Without lock, we cannot be sure that what we got is
                         * a valid page order. Consider only values in the
                         * valid order range to prevent low_pfn overflow.
                         */
                        if (freepage_order > 0 && freepage_order < MAX_ORDER)
                                low_pfn += (1UL << freepage_order) - 1;
                        continue;
                }

                /*
                 * Regardless of being on LRU, compound pages such as THP and
                 * hugetlbfs are not to be compacted. We can potentially save
                 * a lot of iterations if we skip them at once. The check is
                 * racy, but we can consider only valid values and the only
                 * danger is skipping too much.
                 */
                if (PageCompound(page)) {
                        const unsigned int order = compound_order(page);

                        if (likely(order < MAX_ORDER))
                                low_pfn += (1UL << order) - 1;
                        goto isolate_fail;
                }

                /*
```

```
55                  * Check may be lockless but that's ok as we recheck later.
56                  * It's possible to migrate LRU and non-lru movable pages.
57                  * Skip any other type of page
58                  */
59                 if (!PageLRU(page)) {
60                         /*
61                          * __PageMovable can return false positive so we need
62                          * to verify it under page_lock.
63                          */
64                         if (unlikely(__PageMovable(page)) &&
65                                         !PageIsolated(page)) {
66                                 if (locked) {
67                                         spin_unlock_irqrestore(zone_lru_lock(zone),
68                                                                         flags);
69                                         locked = false;
70                                 }
71 
72                                 if (!isolate_movable_page(page, isolate_mode))
73                                         goto isolate_success;
74                         }
75 
76                         goto isolate_fail;
77                 }
```

- Since there is a large scope to be processed in line 6~9 of the code, we try to increase the irq latency by periodically unlocking it in this routine. Also, if there is an interruption element, it aborts processing and exits the loop.
  - Check Frequency
    - SWAP_CLUSTER_MAX (32) pages per block
  - Interrupting Elements
    - The SIGKILL signal is delayed.
    - If there is a preemption request from a high-priority task during an asynchronous migration.
- Update the first valid page from code lines 11~18, and increase the scan counter. If the page is not valid, go to the isolate_fail label.
- In code lines 26~37, the migrate scanner migrates only movable pages that are in use. Therefore, if it is a free page managed by the buddy system, it will be skipped.
- In lines 46~52 of code, the compound(slab, hugetlbfs, thp) page also has no compaction effect, so it goes to the isolate_fail label.
- If the LRU page is not user-assigned in code lines 59~77, go to the isolate_fail label. However, if it is a non-LRU movable page that has not already been isolated, it will be isolated, and if it is successful, it will be moved to the isolate_success label.
  - GPU, ZSRAM (z3fold, zsmalloc), and balloon drivers support migration of non-LRU-movable pages.

mm/compaction.c -3/4-

```
01                 /*
```

```
                 * Migration will fail if an anonymous page is pinned in memory,
                 * so avoid taking lru_lock and isolating it unnecessarily in an
                 * admittedly racy check.
                 */
                if (!page_mapping(page) &&
                    page_count(page) > page_mapcount(page))
                        goto isolate_fail;

                /*
                 * Only allow to migrate anonymous pages in GFP_NOFS context
                 * because those do not depend on fs locks.
                 */
                if (!(cc->gfp_mask & __GFP_FS) && page_mapping(page))
                        goto isolate_fail;

                /* If we already hold the lock, we can skip some rechecking */
                if (!locked) {
                        locked = compact_trylock_irqsave(zone_lru_lock(zone),
                                                                &flags, cc);
                        if (!locked)
                                break;

                        /* Recheck PageLRU and PageCompound under lock */
                        if (!PageLRU(page))
                                goto isolate_fail;

                        /*
                         * Page become compound since the non-locked check,
                         * and it's on LRU. It can only be a THP so the order
                         * is safe to read and it's 0 for tail pages.
                         */
                        if (unlikely(PageCompound(page))) {
                                low_pfn += (1UL << compound_order(page)) - 1;
                                goto isolate_fail;
                        }
                }

                lruvec = mem_cgroup_page_lruvec(page, zone->zone_pgdat);

                /* Try isolate the page */
                if (__isolate_lru_page(page, isolate_mode) != 0)
                        goto isolate_fail;

                VM_BUG_ON_PAGE(PageCompound(page), page);

                /* Successfully isolated */
                del_page_from_lru_list(page, lruvec, page_lru(page));
                inc_node_page_state(page,
                                NR_ISOLATED_ANON + page_is_file_cache(page));

isolate_success:
                list_add(&page->lru, &cc->migratepages);
                cc->nr_migratepages++;
                nr_isolated++;

                /*
```

```
58                 * Record where we could have freed pages by migration and not
59                 * yet flushed them to buddy allocator.
60                 * - this is the lowest page that was isolated and likely be
61                 * then freed by migration.
62                 */
63                if (!cc->last_migrated_pfn)
64                        cc->last_migrated_pfn = low_pfn;
65
66                /* Avoid isolating too much */
67                if (cc->nr_migratepages == COMPACT_CLUSTER_MAX) {
68                        ++low_pfn;
69                        break;
70                }
71
72                continue;
```

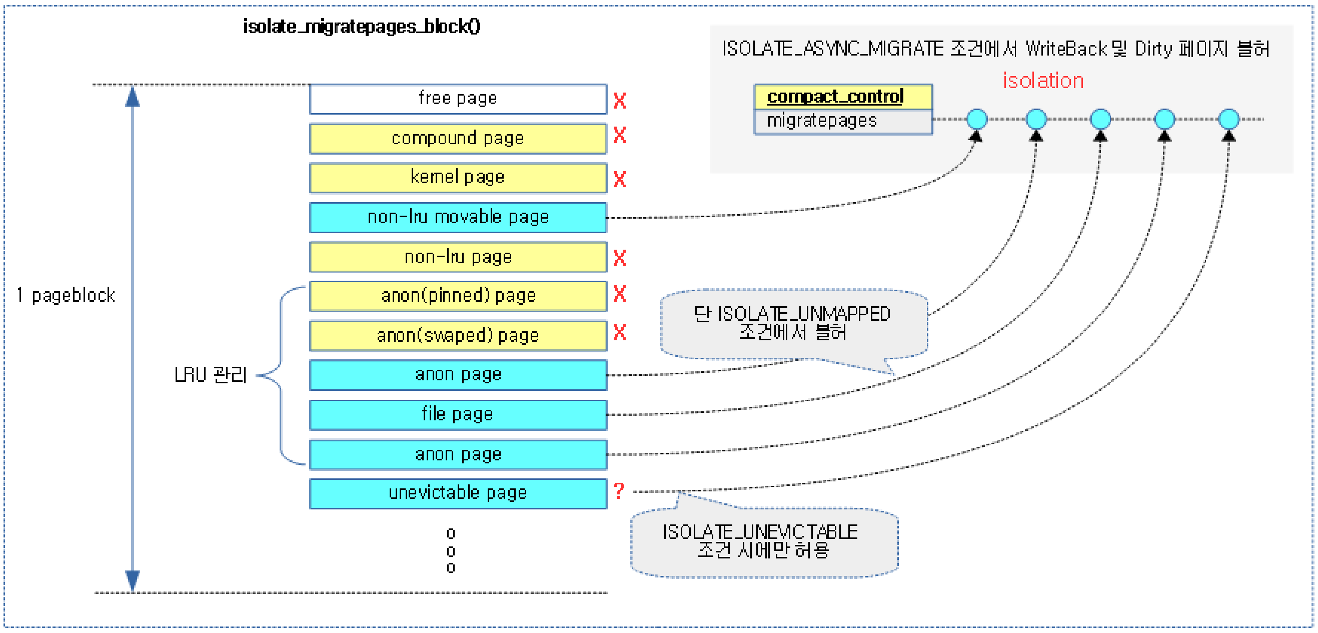
- In line 6~8 of code, move the anonymous page to the isolate_fail if it is in use in a different kernel context.
- If lines 14~15 of the code are mapped to the file system, but fs is not available using GFP_NOFS, go to the isolate_fail label.
- In code lines 18~37, the lock was released from above at regular intervals. In this case, you will need to reacquire it and check the page conditions again. If it's not an LRU page, or if it's a Compound page, go to the isolate_fail label.
- In line 39 of code, select either the node's LRU or the node's LRU from MEMCG for the LRU list.
- In lines 42~50 of code, detach the page from the LRU list and increment the NR_ISOLATED_ANON or NR_ISOLATE_FILE counters.
- In code lines 52~55, isolation is considered successful and can be moved: isolate_success: label. Add the separated pages to the migratepages list and increment the associated stats.
- Update code lines 63~64 if the last migrated page is never specified.
- In line 67~70 of the code, if the migrate page reaches the appropriate amount (32), the processing is stopped.
- Continue on line 72 of code and loop around to process the next page.

mm/compaction.c -4/4-

```
isolate_fail:
                if (!skip_on_failure)
                        continue;

                /*
                 * We have isolated some pages, but then failed. Release them
                 * instead of migrating, as we cannot form the cc->order buddy
                 * page anyway.
                 */
                if (nr_isolated) {
                        if (locked) {
                                spin_unlock_irqrestore(zone_lru_lock(zone), flags);
                                locked = false;
                        }
                        putback_movable_pages(&cc->migratepages);
                        cc->nr_migratepages = 0;
                        cc->last_migrated_pfn = 0;
18                          nr_isolated = 0;
19                  }
20 
21                  if (low_pfn < next_skip_pfn) {
22                          low_pfn = next_skip_pfn - 1;
23                          /*
24                           * The check near the loop beginning would have updated
25                           * next_skip_pfn too, but this is a bit simpler.
26                           */
27                          next_skip_pfn += 1UL << cc->order;
28                  }
29          }
30 
31          /*
32           * The PageBuddy() check could have potentially brought us outside
33           * the range to be scanned.
34           */
35          if (unlikely(low_pfn > end_pfn))
36                  low_pfn = end_pfn;
37 
38          if (locked)
39                  spin_unlock_irqrestore(zone_lru_lock(zone), flags);
40 
41          /*
42           * Update the pageblock-skip information and cached scanner pfn,
43           * if the whole pageblock was scanned without isolating any page.
44           */
45          if (low_pfn == end_pfn)
46                  update_pageblock_skip(cc, valid_page, nr_isolated, true);
47 
48          trace_mm_compaction_isolate_migratepages(start_pfn, low_pfn,
49                                                  nr_scanned, nr_isolated);
50 
51          cc->total_migrate_scanned += nr_scanned;
52          if (nr_isolated)
53                  count_compact_events(COMPACTISOLATED, nr_isolated);
54 
55          return low_pfn;
56  }
```

- In code lines 1~3, isolate_faile: is a label. If isolation fails, if the skip_on_failure is false, continue processing the next page.
- Revert the pages that were isolated in code lines 10~19 and invalidate them.
  - If skip_on_failure is true, if the order unit pages fail in isolation, try the next order unit page.
- In lines 21~28 of the code, change the low_pfn to the next order unit page.
- In lines 35~36 of the code, the low_pfn is restricted from going beyond the end of the processing range.
- If the lock is applied in line 38~39 of the code, unlock it.
- If the end of the page block is processed from line 45~46 of the code and there are no isolated pages processed, set the migrate skip bit of the page block corresponding to valid_page to 1 so that it will be skipped in the next scan. Then, set the start pfn of the migrate scanner (both async and sync) to that page.
- Update the scan counter and isolation counter on lines 51~53 of code.

The following image shows the isolate_migratepages_block() function to isolate a migratable page.



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

too_many_isolated()

mm/compaction.c

```
/* Similar to reclaim, but different enough that they don't share logic */
static bool too_many_isolated(struct zone *zone)
{
        unsigned long active, inactive, isolated;

        inactive = zone_page_state(zone, NR_INACTIVE_FILE) +
                                        zone_page_state(zone, NR_INACTIVE_ANON);
        active = zone_page_state(zone, NR_ACTIVE_FILE) +
                                        zone_page_state(zone, NR_ACTIVE_ANON);
        isolated = zone_page_state(zone, NR_ISOLATED_FILE) +
                                        zone_page_state(zone, NR_ISOLATED_ANON);

        return isolated > (inactive + active) / 2;
}
```

isolated returns true if the number of pages is too high.

- file(active+inactive) true if more than half of the pages

compact_should_abort()

mm/compaction.c

```
/*
 * Aside from avoiding lock contention, compaction also periodically checks
 * need_resched() and either schedules in sync compaction or aborts async
 * compaction. This is similar to what compact_unlock_should_abort() does, but
 * is used where no lock is concerned.
 *
 * Returns false when no scheduling was needed, or sync compaction scheduled.
 * Returns true when async compaction should abort.
 */
```

```
static inline bool compact_should_abort(struct compact_control *cc)
{
        /* async compaction aborts if contended */
        if (need_resched()) {
                if (cc->mode == MIGRATE_ASYNC) {
                        cc->contended = true;
                        return true;
                }

                cond_resched();
        }

        return false;
}
```

If there is a lisse-scheduled request from a high-priority task while the asynchronous migration is being processed, true is returned.

compact_unlock_should_abort()

mm/compaction.c

```
/*
 * Compaction requires the taking of some coarse locks that are potentially
 * very heavily contended. The lock should be periodically unlocked to avoid
 * having disabled IRQs for a long time, even when there is nobody waiting on
 * the lock. It might also be that allowing the IRQs will result in
 * need_resched() becoming true. If scheduling is needed, async compaction
 * aborts. Sync compaction schedules.
 * Either compaction type will also abort if a fatal signal is pending.
 * In either case if the lock was locked, it is dropped and not regained.
 *
 * Returns true if compaction should abort due to fatal signal pending, or
 *              async compaction due to need_resched()
 * Returns false when compaction can continue (sync compaction might have
 *              scheduled)
 */
```

```
static bool compact_unlock_should_abort(spinlock_t *lock,
                unsigned long flags, bool *locked, struct compact_control *cc)
{
        if (*locked) {
                spin_unlock_irqrestore(lock, flags);
                *locked = false;
        }

        if (fatal_signal_pending(current)) {
                cc->contended = true;
                return true;
        }

        if (need_resched()) {
                if (cc->mode == MIGRATE_ASYNC) {
                        cc->contended = true;
                        return true;
                }
                cond_resched();
        }

        return false;
}
```

If locked, unlock, and if there is an aborting element, set true to cc->contended and return true.

- Interrupting Factors
  - The SIGKILL signal is delayed.
  - During an asynchronous migration, there is a request for a lease schedule from a high-priority task.

compact_trylock_irqsave()

mm/compaction.c

```
/*
 * Compaction requires the taking of some coarse locks that are potentially
 * very heavily contended. For async compaction, back out if the lock cannot
 * be taken immediately. For sync compaction, spin on the lock if needed.
 *
 * Returns true if the lock is held
 * Returns false if the lock is not held and compaction should abort
 */
```

```
static bool compact_trylock_irqsave(spinlock_t *lock, unsigned long *flags,
                                                struct compact_control *cc)
{
        if (cc->mode == MIGRATE_ASYNC) {
                if (!spin_trylock_irqsave(lock, *flags)) {
                        cc->contended = true;
                        return false;
                }
        } else {
                spin_lock_irqsave(lock, *flags);
        }

        return true;
}
```

During asynchronous migration processing, if the spin lock used for the compaction competes with another CPU and the acquisition attempt fails at once, CC->contended is assigned true (compaction lock congestion) and returns false.

- In the case of synchronous migration, the lock is obtained unconditionally.

mem_cgroup_page_lruvec()

mm/memcontrol.c

```
/**
 * mem_cgroup_page_lruvec - return lruvec for isolating/putting an LRU page
 * @page: the page
 * @zone: zone of the page
 *
 * This function is only safe when following the LRU page isolation
 * and putback protocol: the LRU lock must be held, and the page must
 * either be PageLRU() or the caller must have isolated/allocated it.
 */
```

```
struct lruvec *mem_cgroup_page_lruvec(struct page *page, struct pglist_data *pgdat)
{
        struct mem_cgroup_per_node *mz;
        struct mem_cgroup *memcg;
        struct lruvec *lruvec;

        if (mem_cgroup_disabled()) {
                lruvec = &pgdat->lruvec;
                goto out;
        }

        memcg = page->mem_cgroup;
        /*
         * Swapcache readahead pages are added to the LRU - and
         * possibly migrated - before they are charged.
         */
        if (!memcg)
                memcg = root_mem_cgroup;

        mz = mem_cgroup_page_nodeinfo(memcg, page);
        lruvec = &mz->lruvec;
out:
        /*
         * Since a node can be onlined after the mem_cgroup was created,
         * we have to be prepared to initialize lruvec->zone here;
         * and if offlined then reonlined, we need to reinitialize it.
         */
        if (unlikely(lruvec->pgdat != pgdat))
                lruvec->pgdat = pgdat;
        return lruvec;
}
```

If memcg is enabled, it returns the lruvec of the node of memcg recorded on that page, and if memcg is disabled, it returns the lruvec of that node.

Isolate LRU Page

__isolate_lru_page()

mm/vmscan.c

```
/*
 * Attempt to remove the specified page from its LRU.  Only take this page
 * if it is of the appropriate PageActive status.  Pages which are being
 * freed elsewhere are also ignored.
 *
 * page:        page to consider
 * mode:        one of the LRU isolation modes defined above
 *
 * returns 0 on success, -ve errno on failure.
 */
```

```
int __isolate_lru_page(struct page *page, isolate_mode_t mode)
{
        int ret = -EINVAL;

        /* Only take pages on the LRU. */
        if (!PageLRU(page))
                return ret;

        /* Compaction should not handle unevictable pages but CMA can do
so */
        if (PageUnevictable(page) && !(mode & ISOLATE_UNEVICTABLE))
                return ret;

        ret = -EBUSY;

        /*
         * To minimise LRU disruption, the caller can indicate that it o
nly
         * wants to isolate pages it will be able to operate on without
         * blocking - clean pages for the most part.
         *
         * ISOLATE_ASYNC_MIGRATE is used to indicate that it only wants
to pages
         * that it is possible to migrate without blocking
         */
        if (mode & ISOLATE_ASYNC_MIGRATE) {
                /* All the caller can do on PageWriteback is block */
                if (PageWriteback(page))
                        return ret;

                if (PageDirty(page)) {
                        struct address_space *mapping;
                        bool migrate_dirty;

                        /*
                         * Only pages without mappings or that have a
                         * ->migratepage callback are possible to migrat
e
                         * without blocking. However, we can be racing w
ith
                         * truncation so it's necessary to lock the page
                         * to stabilise the mapping as truncation holds
                         * the page lock until after the page is removed
                         * from the page cache.
                         */
                        if (!trylock_page(page))
                                return ret;

                        mapping = page_mapping(page);
                        migrate_dirty = !mapping || mapping->a_ops->migr
atepage;
                        unlock_page(page);
                        if (!migrate_dirty)
                                return ret;
                }
        }

        if ((mode & ISOLATE_UNMAPPED) && page_mapped(page))
                return ret;

        if (likely(get_page_unless_zero(page))) {
```

```
56                      /*
57                       * Be careful not to clear PageLRU until after we're
58                       * sure the page is not being freed elsewhere -- the
59                       * page release code relies on it.
60                       */
61                      ClearPageLRU(page);
62                      ret = 0;
63              }
64
65              return ret;
66      }
```

Attempt to remove LRU from the page. Returns 0 on success, -EINVAL or -EBUSY if it fails.

- In line 6~7 of the code, if it is not an LRU page, it will stop processing.
- In line 10~11 of the code, if the page is unevictable and the isolation mode does not support ISOLATE_UNEVICTABLE, the processing will be stopped.
- In line 23~26 of the code, if the page is operating in ISOLATE_ASYNC_MIGRATE mode, the page that is being WriteBack will be aborted with -EBUSY.
- If the migratepage handler function is not registered for a page set to Dirty in code lines 28~49, it will stop processing.
- When operating in ISOLATE_UNMAPPED mode on lines 52~53 of code, the mapping pages stop processing.
- In line 55~65 of the code, if the page is in use with a high probability, it clears the LRU flag and exits the function normally.

Isolate movable page

The objects that use the following functions are used for the following purposes:

- Compaction
  - Migration of non-lru movable pages in direct-compliance
- Off-line Memory
  - Migration of non-LRU movable pages to off-line memory

isolate_movable_page()

mm/migrate.c

```
int isolate_movable_page(struct page *page, isolate_mode_t mode)
{
        struct address_space *mapping;

        /*
         * Avoid burning cycles with pages that are yet under __free_pages(),
         * or just got freed under us.
         *
         * In case we 'win' a race for a movable page being freed under us and
         * raise its refcount preventing __free_pages() from doing its job
         * the put_page() at the end of this block will take care of
         * release this page, thus avoiding a nasty leakage.
         */
14         if (unlikely(!get_page_unless_zero(page)))
15                 goto out;
16 
17         /*
18          * Check PageMovable before holding a PG_lock because page's owner
19          * assumes anybody doesn't touch PG_lock of newly allocated page
20          * so unconditionally grapping the lock ruins page's owner side.
21          */
22         if (unlikely(!__PageMovable(page)))
23                 goto out_putpage;
24         /*
25          * As movable pages are not isolated from LRU lists, concurrent
26          * compaction threads can race against page migration functions
27          * as well as race against the releasing a page.
28          *
29          * In order to avoid having an already isolated movable page
30          * being (wrongly) re-isolated while it is under migration,
31          * or to avoid attempting to isolate pages being released,
32          * lets be sure we have the page lock
33          * before proceeding with the movable page isolation steps.
34          */
35         if (unlikely(!trylock_page(page)))
36                 goto out_putpage;
37 
38         if (!PageMovable(page) || PageIsolated(page))
39                 goto out_no_isolated;
40 
41         mapping = page_mapping(page);
42         VM_BUG_ON_PAGE(!mapping, page);
43 
44         if (!mapping->a_ops->isolate_page(page, mode))
45                 goto out_no_isolated;
46 
47         /* Driver shouldn't use PG_isolated bit of page->flags */
48         WARN_ON_ONCE(PageIsolated(page));
49         __SetPageIsolated(page);
50         unlock_page(page);
51 
52         return 0;
53 
54 out_no_isolated:
55         unlock_page(page);
56 out_putpage:
57         put_page(page);
58 out:
59         return -EBUSY;
60 }
```

isolates non-lru movable pages.

- In line 14~15 of the code, if the page has just become free, the function exits with the -EBUSY error.
- In line 22~23 of the code, if the page is not non-lru movable, the function exits with a -EBUSY error.
- In line 35~36 of the code, we get the page lock and check once again if the page is non-lru movable and if it is already isolate. If the page is not non-lru movable or has already been isolate, the function exits with a -EBUSY error.
- On lines 38~42 of code, if the page is not non-LRU movable or isolate, go to the out_no_isolated label.
  - If you call a function registered in the (*isolate_page) hook of a driver with a non-LRU movable applied (e.g. z3fold, zsmalloc, etc.) and it returns false, the function exits with an -

EBUSY error.

- Set the isolate flag in lines 46~49 of the code, and exit the function with a success (0) result.

Isolatin Eligibility

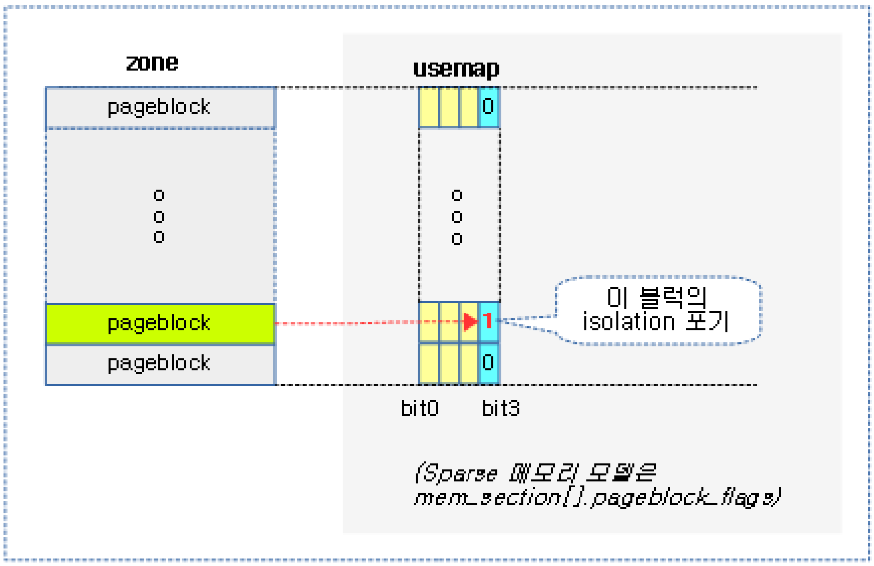
Isolation Eligibility

Isolation for blocks?

Direct-Compaction, with the exception of sync-full, skips the isolation of that page block when it encounters a skip bit on the page block being scanned for fast compaction.

- If isolation has recently failed in the page block, skip the block.

The following illustration shows the use of the usemap's skip bit for page blocks.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction-5.png)

Isolation Force

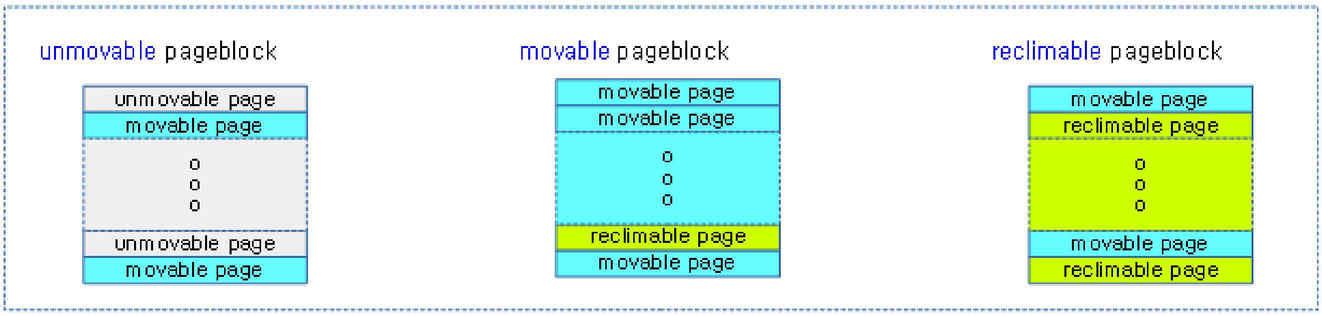
The following performance conditions force an attempt to isolate all blocks by setting a ignore_skip_hint internally, regardless of whether the skip bit is set or not.

- When using direct-compact using compact mode except for sync-full, which has the highest priority
- When using manual-compact
- With kcompactd

Mobility characteristics of Pageblock

As shown in the following figure, the type of the page block has a representative mobility characteristic with more than half (50%) of the page (migratetype) type in the block.

- When using direct-compaction, which should be fast, the compaction is performed only on movable page blocks.



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

isolation_suitable()

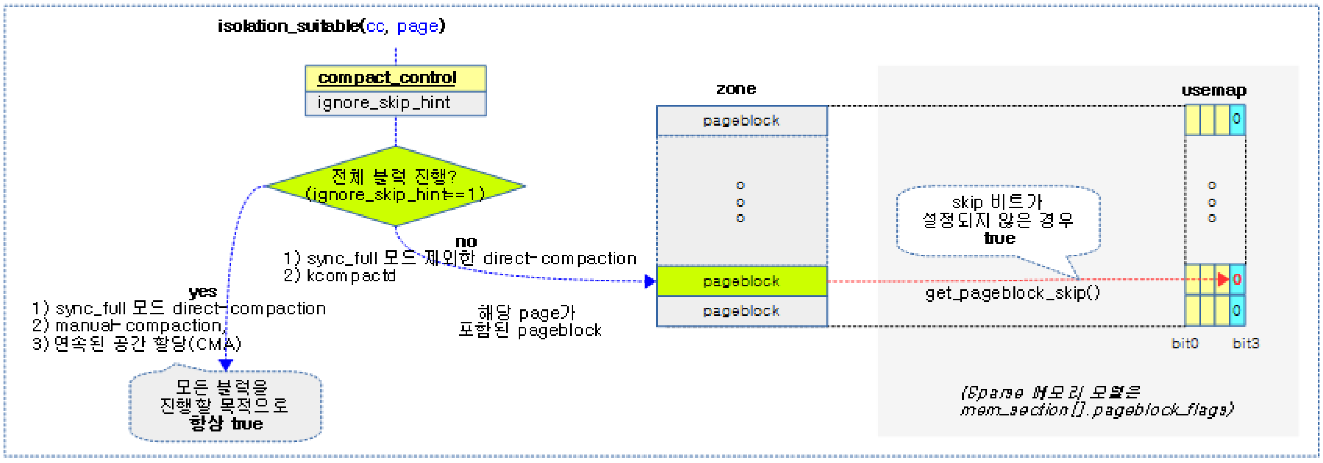
mm/compaction.c

```
1  /* Returns true if the pageblock should be scanned for pages to isolate.
   */
2  static inline bool isolation_suitable(struct compact_control *cc,
3                                          struct page *page)
4  {
5          if (cc->ignore_skip_hint)
6                  return true;
7
8          return !get_pageblock_skip(page);
9  }
```

Check whether it is okay to try isolation from the block.

- If the ignore_skip_hint is set in line 5~6 of the code, it returns true so that the page block can proceed with isolation regardless of whether it is skipped or not.
- Line 8 of the code returns false to skip if isolation has failed in that page block recently.
  - usemap stores the skip bit of each page block.

The following figure shows the processing of the isolation_suitable() function, which returns the isolation of the block.



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

Suitability as a migration source

suitable_migration_source()

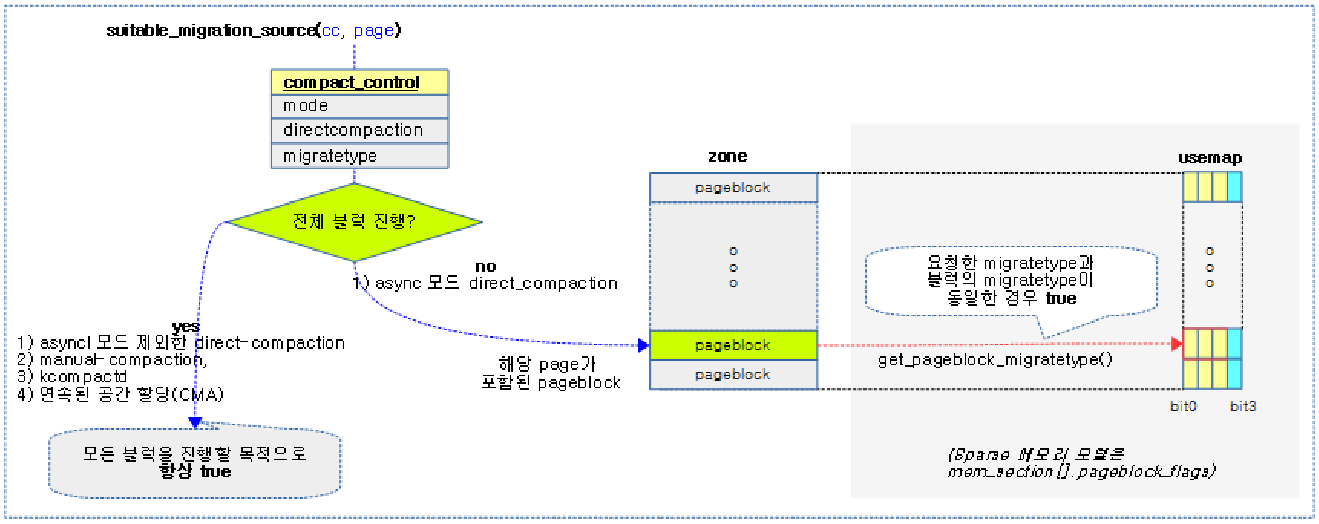
mm/compaction.c

```
01 static bool suitable_migration_source(struct compact_control *cc,
02                                                         struct page *page)
03 {
04         int block_mt;
05 
06         if ((cc->mode != MIGRATE_ASYNC) || !cc->direct_compaction)
07                 return true;
08 
09         block_mt = get_pageblock_migratetype(page);
10 
11         if (cc->migratetype == MIGRATE_MOVABLE)
12                 return is_migrate_movable(block_mt);
13         else
14                 return block_mt == cc->migratetype;
15 }
```

Returns whether a page contained in a page block is eligible for the migration target. In order to make direct-compaction in asynchronous mode lighter and faster, it returns true only if the block migrate type is the same as the requested migrate type. Otherwise, it always returns true.

- In line 6~7 of code, it returns true to unconditionally isolate the block and handle the migrated pages in the following 2 cases.
  - Non-migrate_async requests
  - Requesting manual compaction or kcompactd that are not of type direct_compaction
- When requesting a movable assignment in line 9~12 of the code, it returns whether the movable and CMA block types are present.
  - Unmovable, reclimable, highatomic, and iolate types cannot be subject to migration.
- In line 13~14 of the code, if the movable allocation request is not the same, it returns whether the block type corresponding to the requested type is the same.

The following figure shows the processing of the suitable_migration_source() function, which returns whether the block is appropriate as a migration source.


suitable_migration_source(cc, page)
compact_control
mode
directcompaction
migratetype
전체 블럭 진행?
no
1) async 모드 direct_compaction
yes
1) async 모드 제외한 direct-compaction
2) manual-compaction,
3) kcompactd
4) 연속된 공간 할당(CMA)
모든 블럭을 진행할 목적으로 항상 true
해당 page가 포함된 pageblock
zone
pageblock
pageblock
pageblock
usemap
요청한 migratetype과 블럭의 migratetype이 동일한 경우 true
get_pageblock_migratetype()
bit0
bit3
(Sparse 메모리 모델은 mem_section[].pageblock_flags)

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

Suitability as a migration target

suitable_migration_target()

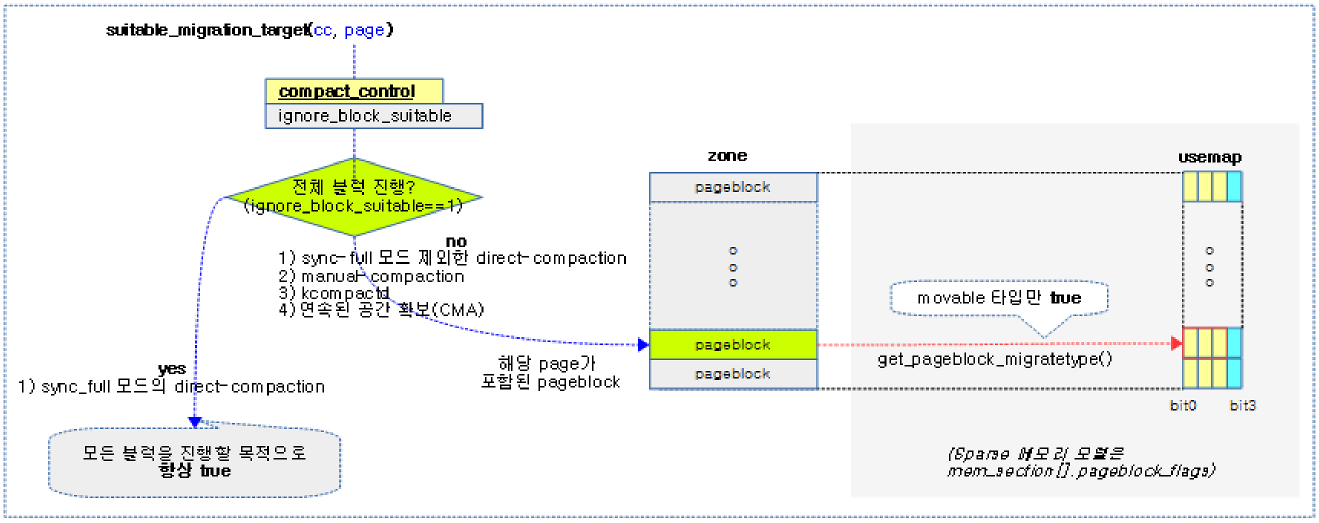
mm/compaction.c

```
01 /* Returns true if the page is within a block suitable for migration to */
02 static bool suitable_migration_target(struct compact_control *cc,
03                                                         struct page *page)
04 {
05         /* If the page is a large free page, then disallow migration */
06         if (PageBuddy(page)) {
07                 /*
08                  * We are checking page_order without zone->lock taken. But
09                  * the only small danger is that we skip a potentially suitable
10                  * pageblock, so it's not worth to check order for valid range.
11                  */
12                 if (page_order_unsafe(page) >= pageblock_order)
13                         return false;
14         }
15 
16         if (cc->ignore_block_suitable)
17                 return true;
18 
19         /* If the block is MIGRATE_MOVABLE or MIGRATE_CMA, allow migration */
20         if (is_migrate_movable(get_pageblock_migratetype(page)))
21                 return true;
22 
23         /* Otherwise skip the block */
24         return false;
25 }
```

Returns whether a page contained in a page block is eligible for the migration target. However, direct-compaction in sync-full mode always returns true.

- In code lines 6~14, a free page on a buddy system that is larger than a page block returns false.
- In lines 16~17 of the code, return true to unconditionally target regardless of the type of page block type.
  - The ignore_block_suitable is only set when direct-component in sync-full mode is enabled.
- In lines 20~24 of code, it returns true only if the page block is of type movable, and false otherwise.

The following figure shows the processing of the suitable_migration_target() function, which returns whether the block is appropriate as a migration target.



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

Specifying a Page Block Skip

update_pageblock_skip()

mm/compaction.c

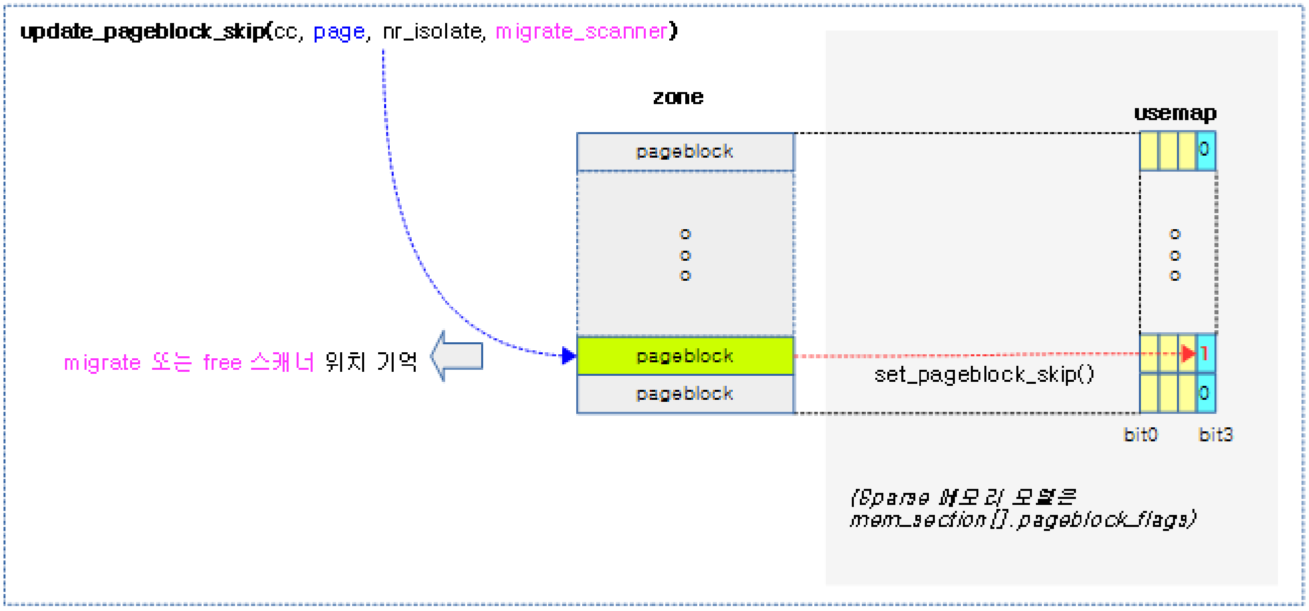
```
/*
 * If no pages were isolated then mark this pageblock to be skipped in t
he
 * future. The information is later cleared by __reset_isolation_suitabl
e().
 */
```

```
static void update_pageblock_skip(struct compact_control *cc,
                        struct page *page, unsigned long nr_isolated,
                        bool migrate_scanner)
{
        struct zone *zone = cc->zone;
        unsigned long pfn;

        if (cc->no_set_skip_hint)
                return;

        if (!page)
                return;

        if (nr_isolated)
                return;

        set_pageblock_skip(page);

        pfn = page_to_pfn(page);

        /* Update where async and sync compaction should restart */
        if (migrate_scanner) {
                if (pfn > zone->compact_cached_migrate_pfn[0])
                        zone->compact_cached_migrate_pfn[0] = pfn;
                if (cc->mode != MIGRATE_ASYNC &&
                    pfn > zone->compact_cached_migrate_pfn[1])
                        zone->compact_cached_migrate_pfn[1] = pfn;
        } else {
                if (pfn < zone->compact_cached_free_pfn)
                        zone->compact_cached_free_pfn = pfn;
        }
}
```

If there are no isolated pages processed, set the migrate skip bit in the page block and set the start pfn of the migrate or free scanner to that page.

- In line 8~15 of the code, the following 3 cases abort to avoid setting the skip bit of the corresponding page block.
  - When no_set_skip_hint is set
    - It is set when isolation is in progress to secure the CMA area.
  - If you don't have a page
  - If you don't have an isolated page
- In line 17 of code, set the skip bit for that page block.
- If a request comes from the migrate scanner routine on lines 22~27 of code, update the pfn to start the migrate scan.
- If a request comes from the free scanner routine on code lines 28~31, update the free scan start pfn.

The following figure shows the handling of the update_pageblock_skip() function, which sets the skip bit to prohibit isolation of the block.


update_pageblock_skip(cc, page, nr_isolate, migrate_scanner)
zone
usemap
pageblock
pageblock
pageblock
0
1
0
set_pageblock_skip()
migrate 또는 free 스캐너 위치 기억
bit0
bit3
(Sparse 메모리 모델은 mem_section[].pageblock_flags)

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

Reset Zone Skip Bit

__reset_isolation_suitable()

mm/compaction.c

```
/*
 * This function is called to clear all cached information on pageblocks that
 * should be skipped for page isolation when the migrate and free page scanner
 * meet.
 */
```

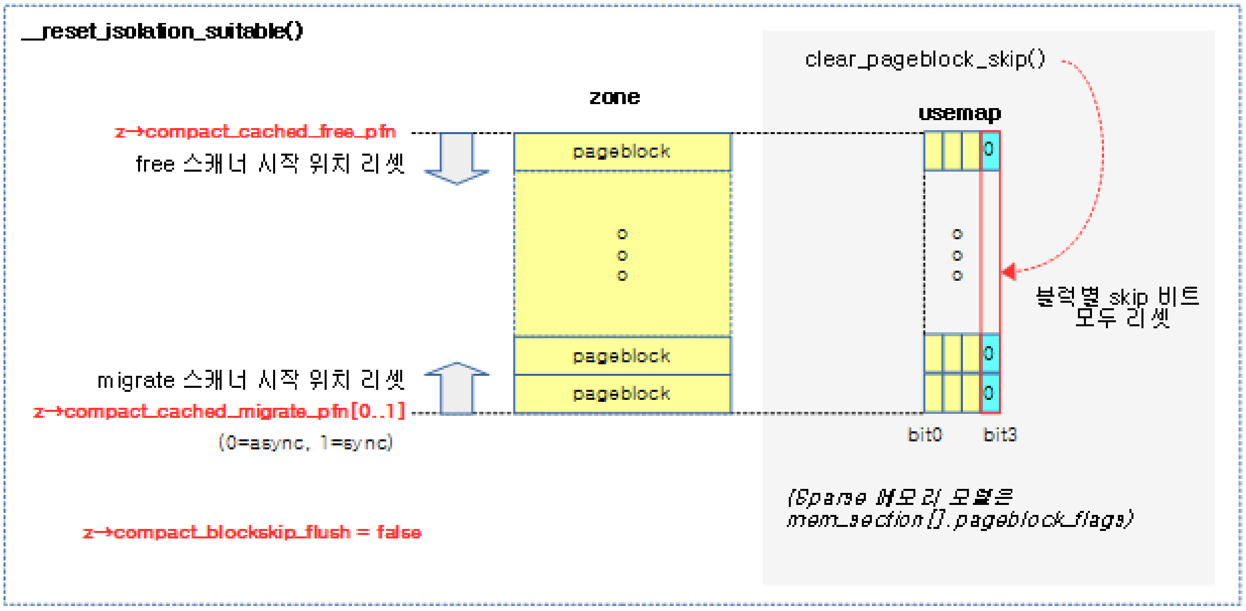
```
static void __reset_isolation_suitable(struct zone *zone)
```

```
{
        unsigned long start_pfn = zone->zone_start_pfn;
        unsigned long end_pfn = zone_end_pfn(zone);
        unsigned long pfn;

        zone->compact_blockskip_flush = false;

        /* Walk the zone and mark every pageblock as suitable for isolat
ion */
        for (pfn = start_pfn; pfn < end_pfn; pfn += pageblock_nr_pages)
{
                struct page *page;

                cond_resched();

                page = pfn_to_online_page(pfn);
                if (!page)
                        continue;
                if (zone != page_zone(page))
                        continue;
                if (pageblock_skip_persistent(page))
                        continue;

                clear_pageblock_skip(page);
        }

        reset_cached_positions(zone);
}
```

Reset the isolate and free scan start address of the zone, and clear all skip bits in the page block. Excludes the skip bit for compound pages that use more than a page block order.

- The migration target area is the start pfn and the end pfn of the zone.
  - Assign the starting pfn of the zone to compact_cached_migrate_pfn[0..1] as the pfn that the isolate scanner will start.
  - Assign the end pfn of the zone to the compact_cached_free_pfn with the pfn that the free scanner will start.
- pageblock->flags
  - If not the Sparse memory model, zone->pageblock_flags specifies usemap
  - If it's a Sparse memory model, the usemap specified by mem_section[].pageblock_flags

The following figure shows the process of the __reset_isolation_suitable() function, which clears the skip bit to allow isolation of all blocks in the zone, and resets the starting position of the scanners.


__reset_isolation_suitable()
clear_pageblock_skip()
zone
usemap
z->compact_cached_free_pfn
free 스캐너 시작 위치 리셋
pageblock
pageblock
pageblock
0
0
0
블럭별 skip 비트 모두 리셋
migrate 스캐너 시작 위치 리셋
z->compact_cached_migrate_pfn[0..1]
(0=async, 1=sync)
bit0
bit3
(Sparse 메모리 모델은 mem_section[].pageblock_flags)
z->compact_blockskip_flush = false

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

Isolation Freepages

The following functions are used for compaction when the high order page is insufficient, where direct-compact, manual-compact, and kcompactd functions are used.

isolate_freepages()

mm/compaction.c

```
/*
 * Based on information in the current compact_control, find blocks
 * suitable for isolating free pages from and then isolate them.
 */
```

```
static void isolate_freepages(struct compact_control *cc)
{
        struct zone *zone = cc->zone;
        struct page *page;
        unsigned long block_start_pfn;  /* start of current pageblock */
        unsigned long isolate_start_pfn; /* exact pfn we start at */
        unsigned long block_end_pfn;    /* end of current pageblock */
        unsigned long low_pfn;         /* lowest pfn scanner is able to scan */
        struct list_head *freelist = &cc->freepages;

        /*
         * Initialise the free scanner. The starting point is where we last
         * successfully isolated from, zone-cached value, or the end of the
         * zone when isolating for the first time. For looping we also need
         * this pfn aligned down to the pageblock boundary, because we do
         * block_start_pfn -= pageblock_nr_pages in the for loop.
         * For ending point, take care when isolating in last pageblock of a
         * a zone which ends in the middle of a pageblock.
         * The low boundary is the end of the pageblock the migration scanner
         * is using.
         */
        isolate_start_pfn = cc->free_pfn;
        block_start_pfn = pageblock_start_pfn(cc->free_pfn);
        block_end_pfn = min(block_start_pfn + pageblock_nr_pages,
                                                zone_end_pfn(zone));
        low_pfn = pageblock_end_pfn(cc->migrate_pfn);

        /*
         * Isolate free pages until enough are available to migrate the
         * pages on cc->migratepages. We stop searching if the migrate
         * and free page scanners meet or enough free pages are isolated.
         */
        for (; block_start_pfn >= low_pfn;
                                block_end_pfn = block_start_pfn,
                                block_start_pfn -= pageblock_nr_pages,
                                isolate_start_pfn = block_start_pfn) {
                /*
                 * This can iterate a massively long zone without finding any
                 * suitable migration targets, so periodically check if we need
                 * to schedule, or even abort async compaction.
                 */
                if (!(block_start_pfn % (SWAP_CLUSTER_MAX * pageblock_nr_pages))
                                                && compact_should_abort(cc))
                        break;

                page = pageblock_pfn_to_page(block_start_pfn, block_end_pfn,
                                                                        zone);
                if (!page)
                        continue;

                /* Check the block is suitable for migration */
                if (!suitable_migration_target(cc, page))
                        continue;

                /* If isolation recently failed, do not retry */
                if (!isolation_suitable(cc, page))
                        continue;

                /* Found a block suitable for isolating free pages from. */
                isolate_freepages_block(cc, &isolate_start_pfn, block_end_pfn,
                                        freelist, false);

                /*
                 * If we isolated enough freepages, or aborted due to lock
                 * contention, terminate.
                 */
                if ((cc->nr_freepages >= cc->nr_migratepages)
                                                        || cc->contended) {
                        if (isolate_start_pfn >= block_end_pfn) {
                                /*
                                 * Restart at previous pageblock if more
                                 * freepages can be isolated next time.
                                 */
                                isolate_start_pfn =
                                        block_start_pfn - pageblock_nr_pages;
```

```
76                              }
77                              break;
78                      } else if (isolate_start_pfn < block_end_pfn) {
79                              /*
80                               * If isolation failed early, do not continue
81                               * needlessly.
82                               */
83                              break;
84                      }
85              }
86
87              /* __isolate_free_page() does not map the pages */
88              map_pages(freelist);
89
90              /*
91               * Record where the free scanner will restart next time. Either we
92               * broke from the loop and set isolate_start_pfn based on the last
93               * call to isolate_freepages_block(), or we met the migration scanner
94               * and the loop terminated due to isolate_start_pfn < low_pfn
95               */
96              cc->free_pfn = isolate_start_pfn;
97      }
```

- Start with cc->free_pfn which was discontinued for the start pfn of the free scanner in code lines 22~25. Then specify the start and end pfn of the block.
  - The free scanner scans the block by moving it down the zone. When isolating free pages within a page block, proceed in the upward direction.
- In order to prevent the block that the migrate scanner is working on at line 26, substitute the end pfn of the page block where the migrate scanner is active in the low_pfn.
- In lines 33~36 of code, the free scanner tverses in the downward direction where the migrate scanner is located, decreasing by page blocks.
- In line 42~44 of code, if you isolate the zones all at once with a free scanner, the area is too large and it is checked for compaction interrupts once in the middle.
  - Check Frequency
    - SWAP_CLUSTER_MAX (32) pages are checked in blocks.
  - Interrupting Factors
    - Asynchronous compaction processing is in progress and there is a preemption request from a higher priority task.
- In lines 46~49 of the code, get the first page of the page block. If the pfn scope is not in the requested zone, skip it.
  - If the last page block of the zone is partial, it should also be skipped
- In lines 52~53 of code, if the page block is not suitable as a target for migration, skip it.
- In line 56~57 of code, if the page block is not suitable for isolation, skip it.
  - Returns false to skip if isolation has recently been canceled from the pageblock.
- In lines 60~61 of the code, isolate the page block and move it to the cc->freepages list.
- In line 67~77 of the code, if there are more free pages secured by the free scanner than migratable pages obtained by the migrate scanner, or if there is a lock contention situation, the loop will be aborted.
- If a single block is not scanned in code lines 78~84, the loop is aborted.

- For a list of free pages isolated from code line 88, remove each page from the LRU list, and decompose it into a 0 order.
- In line 96 of code, note the location of the free scanner for the next scan.

Isolate freepages block

The following function is called to be used for two purposes:

- Compaction
  - isolate_freepages() function
    - It is used for direct-compact, manual-compact, and kcompactd for compaction to obtain high order pages.
- CMA
  - isolate_freepages_range() function.
    - It is used to clear the requested range of the CMA area.

isolate_freepages_block()

mm/compaction.c -1/2-

```
1  /*
2   * Isolate free pages onto a private freelist. If @strict is true, will abort
3   * returning 0 on any invalid PFNs or non-free pages inside of the pageblock
4   * (even though it may still end up isolating some pages).
5   */
```

```
01 static unsigned long isolate_freepages_block(struct compact_control *cc,
02                                 unsigned long *start_pfn,
03                                 unsigned long end_pfn,
04                                 struct list_head *freelist,
05                                 bool strict)
06 {
07         int nr_scanned = 0, total_isolated = 0;
08         struct page *cursor, *valid_page = NULL;
09         unsigned long flags = 0;
10         bool locked = false;
11         unsigned long blockpfn = *start_pfn;
12         unsigned int order;
13 
14         cursor = pfn_to_page(blockpfn);
15 
16         /* Isolate free pages. */
17         for (; blockpfn < end_pfn; blockpfn++, cursor++) {
18                 int isolated;
19                 struct page *page = cursor;
20 
21                 /*
22                  * Periodically drop the lock (if held) regardless of its
23                  * contention, to give chance to IRQs. Abort if fatal signal
24                  * pending or async compaction detects need_resched()
25                  */
26                 if (!(blockpfn % SWAP_CLUSTER_MAX)
                    && compact_unlock_should_abort(&cc->zone->lock, flags,
                                                                &locked, cc))
                        break;

                nr_scanned++;
                if (!pfn_valid_within(blockpfn))
                        goto isolate_fail;

                if (!valid_page)
                        valid_page = page;

                /*
                 * For compound pages such as THP and hugetlbfs, we can save
                 * potentially a lot of iterations if we skip them at once.
                 * The check is racy, but we can consider only valid values
                 * and the only danger is skipping too much.
                 */
                if (PageCompound(page)) {
                        const unsigned int order = compound_order(page);

                        if (likely(order < MAX_ORDER)) {
                                blockpfn += (1UL << order) - 1;
                                cursor += (1UL << order) - 1;
                        }
                        goto isolate_fail;
                }

                if (!PageBuddy(page))
                        goto isolate_fail;

                /*
                 * If we already hold the lock, we can skip some rechecking.
                 * Note that if we hold the lock now, checked_pageblock was
                 * already set in some previous iteration (or strict is true),
                 * so it is correct to skip the suitable migration target
                 * recheck as well.
                 */
                if (!locked) {
                        /*
                         * The zone lock must be held to isolate freepages.
                         * Unfortunately this is a very coarse lock and can be
                         * heavily contended if there are parallel allocations
                         * or parallel compactions. For async compaction do not
                         * spin on the lock and we acquire the lock as late as
                         * possible.
                         */
                        locked = compact_trylock_irqsave(&cc->zone->lock,
                                                                &flags, cc);
                        if (!locked)
                                break;

                        /* Recheck this is a buddy page under lock */
```

```
79                              if (!PageBuddy(page))
80                                      goto isolate_fail;
81                      }
```

- In code lines 14~19, traverse the page from the beginning pfn to the end pfn within the block.
- In line 26~29 of code, I have a large scope to process while scanning a block, so I periodically unlock it if it is locked, and if there is an interrupting element, I set ture to cc->contended and exit the loop.
    - Check Frequency
        - SWAP_CLUSTER_MAX (32) pages per block
    - Interrupting Elements
        - The SIGKILL signal is delayed.
        - If there is a preemption request from a higher priority task during an asynchronous migration.
- Increment the scan counter at line 31 of code.
- On lines 32~36 of the code, if it is not a valid page, go to the isolate_fail label. If it is the first valid page, remember it.
- On lines 44~52 of the code, go to the isolate_fail label if it is a compound page.
- In code lines 54~55, go to the isolate_fail label if the page is not free managed by the buddy system.
- In code lines 64~81, if there is no lock, get a lock, check whether the buddy system is managing the free page again, and if not, move to the isolate_fail label.

mm/compaction.c -2/2-

```
01                      /* Found a free page, will break it into order-0 pages */
02                      order = page_order(page);
03                      isolated = __isolate_free_page(page, order);
04                      if (!isolated)
05                              break;
06                      set_page_private(page, order);
07
08                      total_isolated += isolated;
09                      cc->nr_freepages += isolated;
10                      list_add_tail(&page->lru, freelist);
11
12                      if (!strict && cc->nr_migratepages <= cc->nr_freepages) {
13                              blockpfn += isolated;
14                              break;
15                      }
16                      /* Advance to the end of split page */
17                      blockpfn += isolated - 1;
18                      cursor += isolated - 1;
19                      continue;
20
21 isolate_fail:
22                      if (strict)
23                              break;
24                      else
25                              continue;
26
27              }
28
29              if (locked)
```

```
30                         spin_unlock_irqrestore(&cc->zone->lock, flags);
31 
32                 /*
33                  * There is a tiny chance that we have read bogus compound_order(),
34                  * so be careful to not go outside of the pageblock.
35                  */
36                 if (unlikely(blockpfn > end_pfn))
37                         blockpfn = end_pfn;
38 
39                 trace_mm_compaction_isolate_freepages(*start_pfn, blockpfn,
40                                                 nr_scanned, total_isolated);
41 
42                 /* Record how far we have got within the block */
43                 *start_pfn = blockpfn;
44 
45                 /*
46                  * If strict isolation is requested by CMA then check that all the
47                  * pages requested were isolated. If there were any failures, 0 is
48                  * returned and CMA will fail.
49                  */
50                 if (strict && blockpfn < end_pfn)
51                         total_isolated = 0;
52 
53                 /* Update the pageblock-skip if the whole pageblock was scanned */
54                 if (blockpfn == end_pfn)
55                         update_pageblock_skip(cc, valid_page, total_isolated, false);
56 
57                 cc->total_free_scanned += nr_scanned;
58                 if (total_isolated)
59                         count_compact_events(COMPACTISOLATED, total_isolated);
60                 return total_isolated;
61 }
```

- In code lines 2~6, separate the free page from the free list of the buddy system and record the order.
- In line 8~9 of the code, increment the isolate counter and the secured free page counter.
- Add the page you separated from line 10 to the list of free scanners.
- If the @strict is true in line 12~15 of the code, it is requested to obtain contiguous pages in the CMA area, and if any of them fail, the processing will be aborted. If the @strict is zero and the free scanner has more pages than the migration scanner has obtained, the process will still be stopped.
- On lines 17~19 of the code, let the loop continue for the next page.
- In code lines 21~25, isolate_fail: Label. If the @strict is 1, it will exit the loop immediately if it fails, otherwise it will continue the loop.
- Unlock the lock obtained in code lines 29~30.
- Limit the ongoing pfn in line 36~37 to not exceed the ending pfn.
- Update the @start_pfn value received as an input/output argument in line 43 with the current pfn value.
- In code lines 50~51, if the @strict is true, it is requested to obtain contiguous pages in the CMA area, in which case the total_isolated value is changed to 0 to return the failure value to 0.
- I went all the way to the end of line 54~55 of the code, but it failed, so I marked this page block with a skip mark.

- At line 57 of code, add the number of pages scanned and the number of pages isolated, and return the number of isolated pages.

Isolate Free Page

__isolate_free_page()

mm/page_alloc.c

```
int __isolate_free_page(struct page *page, unsigned int order)
{
        unsigned long watermark;
        struct zone *zone;
        int mt;

        BUG_ON(!PageBuddy(page));

        zone = page_zone(page);
        mt = get_pageblock_migratetype(page);

        if (!is_migrate_isolate(mt)) {
                /*
                 * Obey watermarks as if the page was being allocated. We can
                 * emulate a high-order watermark check with a raised order-0
                 * watermark, because we already know our high-order page
                 * exists.
                 */
                watermark = min_wmark_pages(zone) + (1UL << order);
                if (!zone_watermark_ok(zone, 0, watermark, 0, ALLOC_CMA))
                        return 0;

                __mod_zone_freepage_state(zone, -(1UL << order), mt);
        }

        /* Remove page from free list */
        list_del(&page->lru);
        zone->free_area[order].nr_free--;
        rmv_page_order(page);

        /*
         * Set the pageblock if the isolated page is at least half of a
         * pageblock
         */
        if (order >= pageblock_order - 1) {
                struct page *endpage = page + (1 << order) - 1;
                for (; page < endpage; page += pageblock_nr_pages) {
                        int mt = get_pageblock_migratetype(page);
                        if (!is_migrate_isolate(mt) && !is_migrate_cma(mt)
                            && !is_migrate_highatomic(mt))
                                set_pageblock_migratetype(page,
                                                          MIGRATE_MOVABLE);
                }
        }


        return 1UL << order;
}
```

If the order page is a free page, it will be decomposed and returned as order-0 free page. If this is normal, the number of pages corresponding to 2^order is returned.

- In line 9~10 of the code, we know the zone and migrate types that correspond to the page.
- In line 12~24 of the code, if the migrate type is impossible to isolate, it will request order 0 and return 0 as a failure if it does not pass the watermark boundary (OK) even with the value of low watermark + order number of pages. If it can pass, reduce the number of free pages by the number of order pages.
- In line 27~29 of the code, remove the buddy flag and clear the private value of 0 where the order information is recorded in order to separate the page from the buddy system and remove the buddy information from the page.
- When freeing the order page in line 35~44, if the page block is more than 50%, the type of the page block can be changed to movable if the existing type of the page block is unmovable and reclaimable. However, the isolate, CMA, and highatomic types cannot be changed.
    - Since there can be multiple page blocks within an order page, it traverses page block by page to the end of the order page, and if the page block is not all of the three types: isolate, cma, and highatomic, it is changed to movable.

split_page()

mm/page_alloc.c

```
/*
 * split_page takes a non-compound higher-order page, and splits it into
 * n (1<<order) sub-pages: page[0..n]
 * Each sub-page must be freed individually.
 *
 * Note: this is probably too low level an operation for use in drivers.
 * Please consult with lkml before using this in your driver.
 */
```

```
void split_page(struct page *page, unsigned int order)
{
        int i;

        VM_BUG_ON_PAGE(PageCompound(page), page);
        VM_BUG_ON_PAGE(!page_count(page), page);

        for (i = 1; i < (1 << order); i++)
                set_page_refcounted(page + i);
        split_page_owner(page, order);
}
EXPORT_SYMBOL_GPL(split_page);
```

Set the reference counter to 1 on all split pages. Then, move all the owners of the existing pages.

Get new page hook function

compaction_alloc()

mm/compaction.c

```
/*
```

```
 * This is a migrate-callback that "allocates" freepages by taking pages
 * from the isolated freelists in the block we are migrating to.
 */
```

```
static struct page *compaction_alloc(struct page *migratepage,
                                        unsigned long data,
                                        int **result)
{
        struct compact_control *cc = (struct compact_control *)data;
        struct page *freepage;

        /*
         * Isolate free pages if necessary, and if we are not aborting due to
         * contention.
         */
        if (list_empty(&cc->freepages)) {
                if (!cc->contended)
                        isolate_freepages(cc);

                if (list_empty(&cc->freepages))
                        return NULL;
        }

        freepage = list_entry(cc->freepages.next, struct page, lru);
        list_del(&freepage->lru);
        cc->nr_freepages--;

        return freepage;
}
```

Returns the leading free page from the cc->freepages list isolated from the free scanner. If there are no free pages to return, run the free scanner. If it fails, it returns null.

Put new page hook function

compaction_free()

mm/compaction.c

```
/*
 * This is a migrate-callback that "frees" freepages back to the isolated
 * freelist.  All pages on the freelist are from the same zone, so there is no
 * special handling needed for NUMA.
 */
```

```
static void compaction_free(struct page *page, unsigned long data)
{
        struct compact_control *cc = (struct compact_control *)data;

        list_add(&page->lru, &cc->freepages);
        cc->nr_freepages++;
}
```

Add the page back to the cc->freepages list.

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) | 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) | Sentence C – Current post
- 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

4 thoughts to "Zoned Allocator -8- (Direct Compact-Isolation)"

KWON YONGBEOM
2020-12-26 23:22 (http://jake.dothome.co.kr/zonned-allocator-isolation/#comment-297399)

Report what appears to be a typo.
In the __isolate_free_page() code description, there are some parts
such as "from the corresponding list of the buddy system on code lines 27~29" and the code line does not match the description.
I appreciate it

RESPONSE (/ZONNED-ALLOCATOR-ISOLATION/?REPLYTOCOM=297399#RESPOND)

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

2020-12-27 19:35 (http://jake.dothome.co.kr/zonned-allocator-isolation/#comment-297552)

Thanks for looking for the typo.

The blog code was in version 4.0 and the description was in version 5.4.
The code
has been changed to 5.4 code, and the explanations in lines 35~44 have been slightly enhanced.

I would be grateful if you could continue with the kernel analysis and find any oddities.

I appreciate it.

RESPONSE (/ZONNED-ALLOCATOR-ISOLATION/?REPLYTOCOM=297552#RESPOND)

KWON YONGBEOM
2020-12-28 17:01 (http://jake.dothome.co.kr/zonned-allocator-isolation/#comment-297753)

yes ^^ and I think the kernel versions are v4.x, v5.0, v5.4 and so on.
I wondered if the __reset_isolation_suitable() function had also been changed (as of v5.1) and not updated, but
it says kernel v5.0 at the top of the page.

RESPONSE (/ZONNED-ALLOCATOR-ISOLATION/?REPLYTOCOM=297753#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-12-28 17:22 (http://jake.dothome.co.kr/zonned-allocator-isolation/#comment-297757)

Yes, that's right. I wrote in 4.0 in the early days of my blog, and
I've been switching to 5.x since last year.
Most of the memory was used in the 5.0 source code, and then the scheduler was upgraded to the LTS version of 5.4.

Sometimes, however, I find a source that I haven't changed. ^^;
The __reset_isolation_suitable() function has also been modified.

I appreciate it.

응답 (/ZONNED-ALLOCATOR-ISOLATION/?REPLYTOCOM=297757#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

Zoned Allocator -7- (Direct Compact) (http://jake.dothome.co.kr/zonned-allocator-compaction/)

Zoned Allocator -9- (Direct Compact-Migration) (http://jake.dothome.co.kr/zonned-allocator-migration/)