Zoned Allocator -9- (Direct Compact-Migration)

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

<kernel v5.0>

Migration

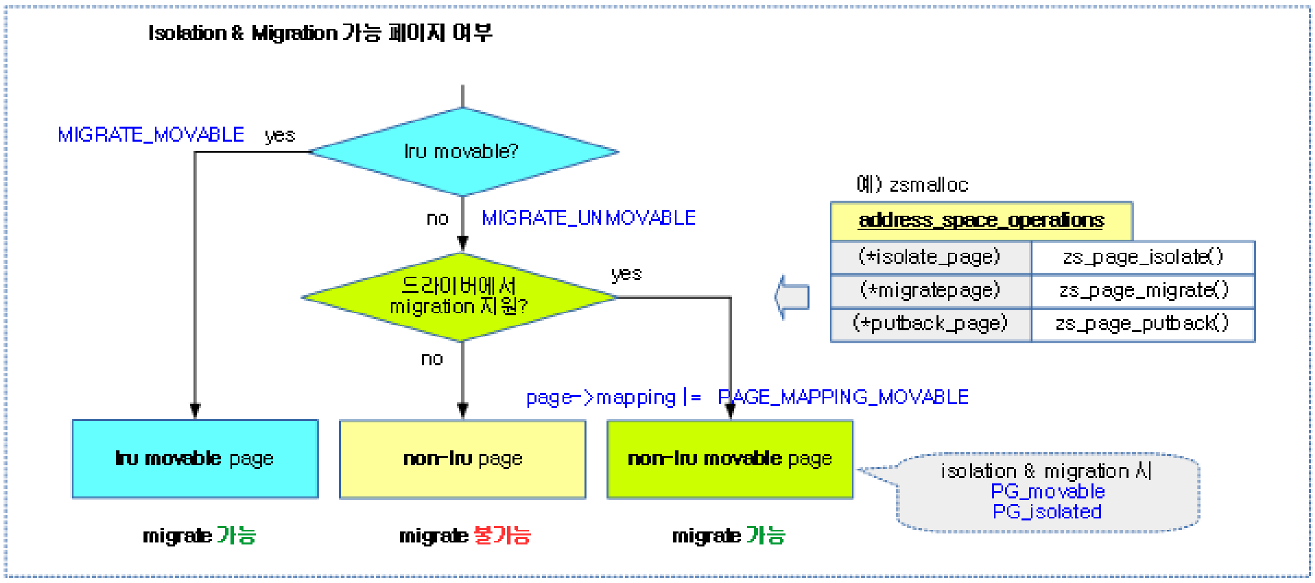
Support for migration of non-movable pages

Traditionally, the kernel has only supported migration for movable LRU pages. Recently, a large number of non-movable pages have been used in embedded systems such as WbOS, android, etc. Due to the widespread use of these non-movable pages, there have been reports of problems with high order allocation. Therefore, although some efforts have been made to eliminate these problems (e.g. improvements to compression algorithms, 0 order allocation in slub fallback, reserved memory, vmalloc, ...), it has not worked in the long run when there are still a lot of non-movable pages in use.

In order to support non-movable pages to be movable, the following patch has been added to the drivers (zram, GPU memory, ...) (*isolate_page), (*migratepage). Drivers that support these hook functions can be migrated through these drivers, even if the kernel classifies them as non-movable.

- 참고: mm: migrate: support non-lru movable page migration (https://github.com/torvalds/linux/commit/bda807d4445414e8e77da704f116bb0880fe0c76)

The following figure shows whether isolation and migration are supported by page type.


Isolation & Migration 가능 페이지 여부
MIGRATE_MOVABLE
yes
lru movable?
no
MIGRATE_UNMOVABLE
드라이버에서 migration 지원?
yes
no
예) zsmalloc
address_space_operations
(*isolate_page) zs_page_isolate()
(*migratepage) zs_page_migrate()
(*putback_page) zs_page_putback()
page->mapping |= PAGE_MAPPING_MOVABLE
lru movable page
non-lru page
non-lru movable page
isolation & migration 시 PG_movable PG_isolated
migrate 가능
migrate 불가능
migrate 가능

(http://jake.dothome.co.kr/wp-content/uploads/2016/07/compaction-7.png)

migrate_pages()

mm/migrate.c

```
/*
 * migrate_pages - migrate the pages specified in a list, to the free pages
 *                   supplied as the target for the page migration
 *
 * @from:                 The list of pages to be migrated.
 * @get_new_page:         The function used to allocate free pages to be used
 *                        as the target of the page migration.
 * @put_new_page:         The function used to free target pages if migration
 *                        fails, or NULL if no special handling is necessary.
 * @private:              Private data to be passed on to get_new_page()
 * @mode:                 The migration mode that specifies the constraints for
 *                        page migration, if any.
 * @reason:               The reason for page migration.
 *
 * The function returns after 10 attempts or if no pages are movable any more
 * because the list has become empty or no retryable pages exist any more.
 * The caller should call putback_movable_pages() to return pages to the LRU
 * or free list only if ret != 0.
 *
 * Returns the number of pages that were not migrated, or an error code.
 */
```

```
int migrate_pages(struct list_head *from, new_page_t get_new_page,
                free_page_t put_new_page, unsigned long private,
                enum migrate_mode mode, int reason)
{
        int retry = 1;
        int nr_failed = 0;
        int nr_succeeded = 0;
        int pass = 0;
        struct page *page;
        struct page *page2;
        int swapwrite = current->flags & PF_SWAPWRITE;
        int rc;

        if (!swapwrite)
                current->flags |= PF_SWAPWRITE;

        for(pass = 0; pass < 10 && retry; pass++) {
                retry = 0;

                list_for_each_entry_safe(page, page2, from, lru) {
retry:
                        cond_resched();

                        if (PageHuge(page))
                                rc = unmap_and_move_huge_page(get_new_page,
                                                put_new_page, private, page,
                                                pass > 2, mode, reason);
                        else
```

```
                                rc = unmap_and_move(get_new_page, put_new_page,
                                                private, page, pass > 2, mode,
                                                reason);

                        switch(rc) {
                        case -ENOMEM:
                                /*
                                 * THP migration might be unsupported or the
                                 * allocation could've failed so we should
                                 * retry on the same page with the THP split
                                 * to base pages.
                                 *
                                 * Head page is retried immediately and tail
                                 * pages are added to the tail of the list so
                                 * we encounter them after the rest of the list
                                 * is processed.
                                 */
                                if (PageTransHuge(page) && !PageHuge(page)) {
                                        lock_page(page);
                                        rc = split_huge_page_to_list(page, from);
                                        unlock_page(page);
                                        if (!rc) {
                                                list_safe_reset_next(page, page2, lru);
                                                goto retry;
                                        }
                                }
                                nr_failed++;
                                goto out;
                        case -EAGAIN:
                                retry++;
                                break;
                        case MIGRATEPAGE_SUCCESS:
                                nr_succeeded++;
                                break;
                        default:
                                /*
                                 * Permanent failure (-EBUSY, -ENOSYS, etc.):
                                 * unlike -EAGAIN case, the failed page is
                                 * removed from migration page list and not
                                 * retried in the next outer loop.
                                 */
                                nr_failed++;
                                break;
                        }
                }
        }
        nr_failed += retry;
        rc = nr_failed;
out:
        if (nr_succeeded)
                count_vm_events(PGMIGRATE_SUCCESS, nr_succeeded);
        if (nr_failed)
                count_vm_events(PGMIGRATE_FAIL, nr_failed);
        trace_mm_migrate_pages(nr_succeeded, nr_failed, mode, reason);
```

```
83
84          if (!swapwrite)
85                  current->flags &= ~PF_SWAPWRITE;
86
87          return rc;
88  }
```

It takes up to 10 attempts to unmap the page isolated by the migrate scanner and then migrate to the free page isolated by the free scanner.

- In line 11~15 of the code, if the current task does not support swap write, add a flag to support swap write only during migration.
- In code lines 17~18, limit the maximum number of repetitions to 10.
- In line 20 of code, the @from passed as an argument loops around the pages of the list.
- In lines 24~31 of the code, unmap and move the huge page or the normal page. On the 10th out of 4 attempts, the force value is set to 1, forcing the writeback pages in full sync mode to wait for the writeback to finish.
- In line 33~56 of the code, if the migration result is out of memory, the processing is aborted. If it is a TransHuge page, split the page and retry.
- This is the case when you need to retry the migration on lines 57~59 of the code.
- This is the case when the migration result succeeds in line 60~62 of the code.
- This is the case when the migration result failed in code lines 63~72.
- This is the out label that you reach on line 77 of code to complete after trying and completing 10 times, or to complete processing due to insufficient memory.
- In lines 84~85 of the code, return the swap writing set to the current task.

Migration on the General page

unmap_and_move()

mm/migrate.c -1/2-

```
1  /*
2   * Obtain the lock on page, remove all ptes and migrate the page
3   * to the newly allocated page in newpage.
4   */
```

```
01  static ICE_noinline int unmap_and_move(new_page_t get_new_page,
02                                     free_page_t put_new_page,
03                                     unsigned long private, struct page *p
    age,
04                                     int force, enum migrate_mode mode,
05                                     enum migrate_reason reason)
06  {
07          int rc = MIGRATEPAGE_SUCCESS;
08          struct page *newpage;
09
10          if (!thp_migration_supported() && PageTransHuge(page))
11                  return -ENOMEM;
12
13          newpage = get_new_page(page, private);
14          if (!newpage)
15                  return -ENOMEM;
16
17          if (page_count(page) == 1) {
```

```
18                 /* page was freed from under us. So we are done. */
19                 ClearPageActive(page);
20                 ClearPageUnevictable(page);
21                 if (unlikely(__PageMovable(page))) {
22                         lock_page(page);
23                         if (!PageMovable(page))
24                                 __ClearPageIsolated(page);
25                         unlock_page(page);
26                 }
27                 if (put_new_page)
28                         put_new_page(newpage, private);
29                 else
30                         put_page(newpage);
31                 goto out;
32         }
33
34         rc = __unmap_and_move(page, newpage, force, mode);
35         if (rc == MIGRATEPAGE_SUCCESS)
36                 set_page_owner_migrate_reason(newpage, reason);
```

The migrate scanner unmaps the isolated page, and then the free scanner moves it to the isolated free page.

- In code lines 10~11, it returns -ENOMEM for thp when thp(Transparent Huge Page) migration is not supported.
- In lines 13~15 of the code, get the free page via the @get_new_page function.
  - In compaction, use the compaction_alloc() function on the @get_new_page to retrieve the leading free page from the list maintained by the free scanner.
- In line 17~32 of code, if the page is already free, clear the active and unevictable flags of the page.
- In code lines 21~26, if the page is non-LRU movable with a low probability but the driver does not have an implementation of the (*isolate_page) hook function, clear the isolated flag.
- If a @put_new_page function is given in lines 27~31 of the code, it will use this function to return the free page, and if it is not specified, the page will be returned to the buddy system.
  - In compaction, use the compaction_free() function on the @put_new_page to return it to a list managed by the free scanner.
- After unmapping in lines 34~36 of the code, migrate the page to newpage. If the migration is successful, note the reason in the page owner for debugging.

mm/migrate.c -2/2-

```
01 out:
02         if (rc != -EAGAIN) {
03                 /*
04                  * A page that has been migrated has all references
05                  * removed and will be freed. A page that has not been
06                  * migrated will have kepts its references and be
07                  * restored.
08                  */
09                 list_del(&page->lru);
10
11                 /*
12                  * Compaction can migrate also non-LRU pages which are
13                  * not accounted to NR_ISOLATED_*. They can be recognize
d
14                  * as __PageMovable
15                  */
16                 if (likely(!__PageMovable(page)))
17                         mod_node_page_state(page_pgdat(page), NR_ISOLATED_ANON +
18                                         page_is_file_cache(page), -hpage_nr_pages(page));
19         }
20 
21         /*
22          * If migration is successful, releases reference grabbed during
23          * isolation. Otherwise, restore the page to right list unless
24          * we want to retry.
25          */
26         if (rc == MIGRATEPAGE_SUCCESS) {
27                 put_page(page);
28                 if (reason == MR_MEMORY_FAILURE) {
29                         /*
30                          * Set PG_HWPoison on just freed page
31                          * intentionally. Although it's rather weird,
32                          * it's how HWPoison flag works at the moment.
33                          */
34                         if (set_hwpoison_free_buddy_page(page))
35                                 num_poisoned_pages_inc();
36                 }
37         } else {
38                 if (rc != -EAGAIN) {
39                         if (likely(!__PageMovable(page))) {
40                                 putback_lru_page(page);
41                                 goto put_new;
42                         }
43 
44                         lock_page(page);
45                         if (PageMovable(page))
46                                 putback_movable_page(page);
47                         else
48                                 __ClearPageIsolated(page);
49                         unlock_page(page);
50                         put_page(page);
51                 }
52 put_new:
53                 if (put_new_page)
54                         put_new_page(newpage, private);
55                 else
56                         put_page(newpage);
57         }
58 
59         return rc;
60 }
```

- In code lines 1~19, the out: label is. If the migration result is not -EAGAIN, detach the page from the LRU list. And with a high degree of probability, if the page is not non-LRU movable, it will decrement the nr_isolated_anon or nr_isolated_file counter.
- If migrate succeeds in code lines 26~36, it decrements the reference counter on the page and returns it to the buddy system.
- In lines 37~42 of code, if the migrate result is -EAGAIN, it will be returned to the LRU list if it is not a movable mapped page.
- In line 44~50 of the code, the migrate is handled if it fails. For non-LRU movable free pages, use the (*putback_page) hook function in the file system to return them to their original position, or to the LRU list where they were originally.
- In code lines 52~56, put_new: Label. Given a @put_new_page, the free page is returned to the function via that function. If not, it will be returned to the buddy system.

__unmap_and_move()

mm/migrate.c -1/3-

```
01 static int __unmap_and_move(struct page *page, struct page *newpage,
02                                 int force, enum migrate_mode mode)
03 {
04         int rc = -EAGAIN;
05         int page_was_mapped = 0;
06         struct anon_vma *anon_vma = NULL;
07         bool is_lru = !__PageMovable(page);
08 
09         if (!trylock_page(page)) {
10                 if (!force || mode == MIGRATE_ASYNC)
11                         goto out;
12 
13                 /*
14                  * It's not safe for direct compaction to call lock_page.
15                  * For example, during page readahead pages are added locked
16                  * to the LRU. Later, when the IO completes the pages are
17                  * marked uptodate and unlocked. However, the queueing
18                  * could be merging multiple pages for one bio (e.g.
19                  * mpage_readpages). If an allocation happens for the
20                  * second or third page, the process can end up locking
21                  * the same page twice and deadlocking. Rather than
22                  * trying to be clever about what pages can be locked,
23                  * avoid the use of lock_page for direct compaction
24                  * altogether.
25                  */
26                 if (current->flags & PF_MEMALLOC)
27                         goto out;
28 
29                 lock_page(page);
30         }
31 
32         if (PageWriteback(page)) {
33                 /*
34                  * Only in the case of a full synchronous migration is it
35                  * necessary to wait for PageWriteback. In the async case,
36                  * the retry loop is too short and in the sync-light case,
37                  * the overhead of stalling is too much
38                  */
39                 switch (mode) {
40                 case MIGRATE_SYNC:
41                 case MIGRATE_SYNC_NO_COPY:
42                         break;
43                 default:
44                         rc = -EBUSY;
45                         goto out_unlock;
46                 }
47                 if (!force)
48                         goto out_unlock;
49                 wait_on_page_writeback(page);
50         }
```

unmap page and migrate to newpage.

- In line 7 of the code, LRU puts the movable page in the is_lru or not.
    - true=lru movablepage
    - false=non-lru movable pages

- In line 9~30 of the code, we get the lock on the page from . If the lock acquisition attempt fails, if the @force is 0 or if the game is in async migrate mode, the function terminates with a -EAGAIN error. And if it's called in the pfmemalloc situation, migration doesn't have to be redundant.
- In line 32~50 of the code, if page is writing back to the filesystem, it returns a sync or sync_no_copy error -EAGAIN. However, if the @force is forced, wait for the writeback to complete. For all other async and sync_light modes, return an -EBUSY error.

mm/migrate.c -2/3-

```
01 .        /*
02           * By try_to_unmap(), page->mapcount goes down to 0 here. In this case,
03           * we cannot notice that anon_vma is freed while we migrates a page.
04           * This get_anon_vma() delays freeing anon_vma pointer until the end
05           * of migration. File cache pages are no problem because of page_lock()
06           * File Caches may use write_page() or lock_page() in migration, then,
07           * just care Anon page here.
08           *
09           * Only page_get_anon_vma() understands the subtleties of
10           * getting a hold on an anon_vma from outside one of its mms.
11           * But if we cannot get anon_vma, then we won't need it anyway,
12           * because that implies that the anon page is no longer mapped
13           * (and cannot be remapped so long as we hold the page lock).
14           */
15          if (PageAnon(page) && !PageKsm(page))
16                  anon_vma = page_get_anon_vma(page);
17
18          /*
19           * Block others from accessing the new page when we get around to
20           * establishing additional references. We are usually the only one
21           * holding a reference to newpage at this point. We used to have a BUG
22           * here if trylock_page(newpage) fails, but would like to allow for
23           * cases where there might be a race with the previous use of newpage.
24           * This is much like races on refcount of oldpage: just don't BUG().
25           */
26          if (unlikely(!trylock_page(newpage)))
27                  goto out_unlock;
28
29          if (unlikely(!is_lru)) {
30                  rc = move_to_new_page(newpage, page, mode);
31                  goto out_unlock_both;
32          }
33
34          /*
35           * Corner case handling:
36           * 1. When a new swap-cache page is read into, it is added to the LRU
37           * and treated as swapcache but it has no rmap yet.
38           * Calling try_to_unmap() against a page->mapping==NULL page will
39           * trigger a BUG.  So handle it here.
40           * 2. An orphaned page (see truncate_complete_page) might have
41           * fs-private metadata. The page can be picked up due to memory
```

```
42           * offlining.  Everywhere else except page reclaim, the page is
43           * invisible to the vm, so the page can not be migrated.  So try to
44           * free the metadata, so the page can be freed.
45           */
46          if (!page->mapping) {
47                  VM_BUG_ON_PAGE(PageAnon(page), page);
48                  if (page_has_private(page)) {
49                          try_to_free_buffers(page);
50                          goto out_unlock_both;
51                  }
52          } else if (page_mapped(page)) {
53                  /* Establish migration ptes */
54                  VM_BUG_ON_PAGE(PageAnon(page) && !PageKsm(page) && !anon_vma,
55                                  page);
56                  try_to_unmap(page,
57                          TTU_MIGRATION|TTU_IGNORE_MLOCK|TTU_IGNORE_ACCESS);
58                  page_was_mapped = 1;
59          }
60
61          if (!page_mapped(page))
62                  rc = move_to_new_page(newpage, page, mode);
63
64          if (page_was_mapped)
65                  remove_migration_ptes(page,
66                          rc == MIGRATEPAGE_SUCCESS ? newpage : page, false);
```

- In line 15~16 of the code, if it is an anon page without KSM (Kernel Shared Memory), get the anon_vma.
- If the code line 29~32 is a non-LRU movable page with a low probability, we move the page to the new page and go straight to the out_unlock_both: label, as we don't need to do the mapping removal routine.
- In lines 46~59 of code, the following is the handling of an uncommon corner case. If it's an anon page and uses a separate buffer (e.g. ksm), remove the free buffer and exit the function. Otherwise, have them unmap from all page tables mapped to this page.
- If the page is not mapped to the page table in line 61~62 of the code, move the page to the new page and go directly to the out_unlock_both label without having to perform the unmapping routine.
- If the page was mapped in code lines 64~66, move all the mappings linked to the existing page to the new page.

mm/migrate.c -3/3-

```
01  out_unlock_both:
02          unlock_page(newpage);
03  out_unlock:
04          /* Drop an anon_vma reference if we took one */
05          if (anon_vma)
06                  put_anon_vma(anon_vma);
07          unlock_page(page);
08  out:
09          /*
10           * If migration is successful, decrease refcount of the newpage
11           * which will not free the page because new page owner increased
12           * refcounter. As well, if it is LRU page, add the page to LRU
```

```
13          * list in here. Use the old state of the isolated source page to
14          * determine if we migrated a LRU page. newpage was already unlocked
15          * and possibly modified by its owner - don't rely on the page
16          * state.
17          */
18         if (rc == MIGRATEPAGE_SUCCESS) {
19                 if (unlikely(!is_lru))
20                         put_page(newpage);
21                 else
22                         putback_lru_page(newpage);
23         }
24 
25         return rc;
26 }
```

- In code lines 1~2, out_unlock_both: The label releases the lock for the new page first.
- In code lines 3~7, out_unlock: The label releases the lock on the existing page. If it is an anon page, the reference counter is decremented because the use of the anon_vma is complete.
- If the migration is successful on lines 8~23 of the code, it completes the use of the newpage that has been changed to busy. If the page is LRU movable, revert to the LRU list.

Check if a page is non-lru movable

PageMovable()

mm/compaction.c

```
01 int PageMovable(struct page *page)
02 {
03         struct address_space *mapping;
04 
05         VM_BUG_ON_PAGE(!PageLocked(page), page);
06         if (!__PageMovable(page))
07                 return 0;
08 
09         mapping = page_mapping(page);
10         if (mapping && mapping->a_ops && mapping->a_ops->isolate_page)
11                 return 1;
12 
13         return 0;
14 }
15 EXPORT_SYMBOL(PageMovable);
```

Returns whether the page is movable or not.

- In line 6~7 of the code, it returns 0 if the page is not non-LRU movable.
- Returns 9 if the driver of the mapped page in line 13~1 supports the (*isolate_page) hook function, otherwise returns 0.

__PageMovable()

```
static __always_inline int __PageMovable(struct page *page)
{
        return ((unsigned long)page->mapping & PAGE_MAPPING_FLAGS) ==
                                PAGE_MAPPING_MOVABLE;
}
```

Returns whether the page is movable or not.

Wait for writeback to complete

wait_on_page_writeback()

include/linux/pagemap.h

```
/*
 * Wait for a page to complete writeback
 */
static inline void wait_on_page_writeback(struct page *page)
{
        if (PageWriteback(page))
                wait_on_page_bit(page, PG_writeback);
}
```

page waits for the writeback to complete.

Go to a new page

move_to_new_page()

mm/migrate.c

```
/*
 * Move a page to a newly allocated page
 * The page is locked and all ptes have been successfully removed.
 *
 * The new page will have replaced the old page if this function
 * is successful.
 *
 * Return value:
 *   < 0 - error code
 *  MIGRATEPAGE_SUCCESS - success
 */
```

```
static int move_to_new_page(struct page *newpage, struct page *page,
                                enum migrate_mode mode)
{
        struct address_space *mapping;
        int rc = -EAGAIN;
        bool is_lru = !__PageMovable(page);

        VM_BUG_ON_PAGE(!PageLocked(page), page);
        VM_BUG_ON_PAGE(!PageLocked(newpage), newpage);

        mapping = page_mapping(page);

        if (likely(is_lru)) {
                if (!mapping)
                        rc = migrate_page(mapping, newpage, page, mode);
                else if (mapping->a_ops->migratepage)
                        /*
                         * Most pages have a mapping and most filesystem
s
                         * provide a migratepage callback. Anonymous pag
es
                         * are part of swap space which also has its own
21                           * migratepage callback. This is the most common path
22                           * for page migration.
23                           */
24                          rc = mapping->a_ops->migratepage(mapping, newpage,
25                                                          page, mode);
26                  else
27                          rc = fallback_migrate_page(mapping, newpage,
28                                                          page, mode);
29          } else {
30                  /*
31                   * In case of non-lru page, it could be released after
32                   * isolation step. In that case, we shouldn't try migration.
33                   */
34                  VM_BUG_ON_PAGE(!PageIsolated(page), page);
35                  if (!PageMovable(page)) {
36                          rc = MIGRATEPAGE_SUCCESS;
37                          __ClearPageIsolated(page);
38                          goto out;
39                  }
40
41                  rc = mapping->a_ops->migratepage(mapping, newpage,
42                                                  page, mode);
43                  WARN_ON_ONCE(rc == MIGRATEPAGE_SUCCESS &&
44                          !PageIsolated(page));
45          }
46
47          /*
48           * When successful, old pagecache page->mapping must be cleared before
49           * page is freed; but stats require that PageAnon be left as PageAnon.
50           */
51          if (rc == MIGRATEPAGE_SUCCESS) {
52                  if (__PageMovable(page)) {
53                          VM_BUG_ON_PAGE(!PageIsolated(page), page);
54
55                          /*
56                           * We clear PG_movable under page_lock so any compactor
57                           * cannot try to migrate this page.
58                           */
59                          __ClearPageIsolated(page);
60                  }
61
62                  /*
63                   * Anonymous and movable page->mapping will be cleard by
64                   * free_pages_prepare so don't reset it here for keeping
65                   * the type to work PageAnon, for example.
66                   */
67                  if (!PageMappingFlags(page))
68                          page->mapping = NULL;
69          }
70  out:
71          return rc;
72  }
```

Migrate the page to the newly assigned page.

- In line 6 of the code, we find out whether the movable page is a managed page in the LRU list.
- In line 11~28 of code, perform the migration of the LRU movable page. There are three types of processing:
  - A) Migration on anon pages

    - B) Migration using (*migratepage) on the swap cache and file cache pages
    - C) Fallback migration used when the swap cache and file cache page (*migratepage) are absent.
- Perform the migration of the non-LRU movable page on lines 29~45 of the code. There is one type of treatment.
    - D) Perform non-lru page migration. However, if the (*migratepages) hook is not implemented in the file system, remove the isolated flag and return success.
- If the migration succeeds in code lines 51~69, the existing page is now a free page. If it was a non-LRU page, remove the isolated flag. And if it was mapped, remove the mapping.

A) lru movable – migration from anon pages

migrate_page()

mm/migrate.c

```
/*
 * Common logic to directly migrate a single LRU page suitable for
 * pages that do not use PagePrivate/PagePrivate2.
 *
 * Pages are locked upon entry and exit.
 */
```

```
01 int migrate_page(struct address_space *mapping,
02                 struct page *newpage, struct page *page,
03                 enum migrate_mode mode)
04 {
05         int rc;
06
07         BUG_ON(PageWriteback(page));    /* Writeback must be complete */
08
09         rc = migrate_page_move_mapping(mapping, newpage, page, mode, 0);
10
11         if (rc != MIGRATEPAGE_SUCCESS)
12                 return rc;
13
14         if (mode != MIGRATE_SYNC_NO_COPY)
15                 migrate_page_copy(newpage, page);
16         else
17                 migrate_page_states(newpage, page);
18         return MIGRATEPAGE_SUCCESS;
19 }
20 EXPORT_SYMBOL(migrate_page);
```

LRU migrates and copies the mapping of movable pages to newly assigned pages.

- Migrate the page in line 9~12 of the code.
- In line 14~17 of the code, MIGRATE_SYNC_NO_COPY mode only moves the page descriptor information, but in other modes, it also adds and does the page frame copy from the CPU.

B) Migration using (*migratepage) on the swap cache and file cache pages

Support in the following routines:

- mm/shmem.c – migrate_page()
- mm/swap_state.c – migrate_page()
- fs/block_dev.c – buffer_migrate_page_norefs()
- fs/ubifs/file.c – ubifs_migrate_page()
- fs/ext2/inode.c – buffer_migrate_page()
- fs/btrfs/disk-io.c – btree_migratepage()
- fs/f2fs/checkpoint.c – f2fs_migrate_page()
- fs/xfs/xfs_aops.c – iomap_migrate_page()
- fs/hugetlbfs/inode.c – hugetlbfs_migrate_page()
- fs/nfs/file.c – nfs_migrate_page()
- …

C) fallback migrate

fallback_migrate_page()

mm/migrate.c

```
/*
 * Default handling if a filesystem does not provide a migration function.
 */
```

```
01 static int fallback_migrate_page(struct address_space *mapping,
02         struct page *newpage, struct page *page, enum migrate_mode mode)
03 {
04         if (PageDirty(page)) {
05                 /* Only writeback pages in full synchronous migration */
06                 switch (mode) {
07                 case MIGRATE_SYNC:
08                 case MIGRATE_SYNC_NO_COPY:
09                         break;
10                 default:
11                         return -EBUSY;
12                 }
13                 return writeout(mapping, page);
14         }
15 
16         /*
17          * Buffers may be managed in a filesystem specific way.
18          * We must have no buffers or drop them.
19          */
20         if (page_has_private(page) &&
21             !try_to_release_page(page, GFP_KERNEL))
22                 return -EAGAIN;
23 
24         return migrate_page(mapping, newpage, page, mode);
25 }
```

When the file system does not support the migration function, default is called to migrate the page.

- If it is a ditty page in line 4~14 of the code and it is operated in MIGRATE_SYNC or MIGRATE_SYNC_NO_COPY mode, the page is recorded and returned after clearing the dirty state. FOR ALL OTHER MODES, IT RETURNS -EBUSY.
- In line 20~22 of the code, if it is a private page, remove the metadata for the page generated by the file system, etc.

- On line 24 of code, migrate page to a new page.

D) non-lru page migration

Support in the following routines:

- mm/zsmalloc.c – zs_page_migrate()
- mm/balloon_compaction.c – balloon_page_migrate()
- drivers/virtio/virtio_balloon.c – virtballoon_migratepage()

Moving Mapping

migrate_page_move_mapping()

This function is called and used in the following routines:

- mm/migrate.c – migrate_page()
- mm/migrate.c – __buffer_migrate_page()
- fs/iomap.c – iomap_migrate_page()
- fs/ubifs/file.c – ubifs_migrate_page()
- fs/f2fs/data.c – f2fs_migrate_page()
- fs/aio.c – aio_migratepage()

mm/migrate.c -1/2-

```
1  /*
2   * Replace the page in the mapping.
3   *
4   * The number of remaining references must be:
5   * 1 for anonymous pages without a mapping
6   * 2 for pages with a mapping
7   * 3 for pages with a mapping and PagePrivate/PagePrivate2 set.
8   */
```

```
01 int migrate_page_move_mapping(struct address_space *mapping,
02                 struct page *newpage, struct page *page, enum migrate_mode mode,
03                 int extra_count)
04 {
05         XA_STATE(xas, &mapping->i_pages, page_index(page));
06         struct zone *oldzone, *newzone;
07         int dirty;
08         int expected_count = expected_page_refs(page) + extra_count;
09
10         if (!mapping) {
11                 /* Anonymous page without mapping */
12                 if (page_count(page) != expected_count)
13                         return -EAGAIN;
14
15                 /* No turning back from here */
16                 newpage->index = page->index;
17                 newpage->mapping = page->mapping;
18                 if (PageSwapBacked(page))
19                         __SetPageSwapBacked(newpage);
20
21                 return MIGRATEPAGE_SUCCESS;
22         }
23 
24         oldzone = page_zone(page);
25         newzone = page_zone(newpage);
26 
27         xas_lock_irq(&xas);
28         if (page_count(page) != expected_count || xas_load(&xas) != page) {
29                 xas_unlock_irq(&xas);
30                 return -EAGAIN;
31         }
32 
33         if (!page_ref_freeze(page, expected_count)) {
34                 xas_unlock_irq(&xas);
35                 return -EAGAIN;
36         }
37 
38         /*
39          * Now we know that no one else is looking at the page:
40          * no turning back from here.
41          */
42         newpage->index = page->index;
43         newpage->mapping = page->mapping;
44         page_ref_add(newpage, hpage_nr_pages(page)); /* add cache reference */
45         if (PageSwapBacked(page)) {
46                 __SetPageSwapBacked(newpage);
47                 if (PageSwapCache(page)) {
48                         SetPageSwapCache(newpage);
49                         set_page_private(newpage, page_private(page));
50                 }
51         } else {
52                 VM_BUG_ON_PAGE(PageSwapCache(page), page);
53         }
```

Migrate the mapping of the page to newpage.

- In the case of an anon page that is not mapped in code lines 10~22, move the index and mapping information to the new page. And if the SwapBacked flag is set, it will be removed. If the reference counter is different from expected_count, it returns -EAGAIN, otherwise it returns success.
- On line 27 of code, acquire the XAS array lock.
  - 참고: The XArray data structure (https://lwn.net/Articles/745073/)
- In lines 28~31 of the code, if the reference counter is not expected_count, it returns -EAGAIN.
- In lines 33~36 of the code, reset the reference counter to 0. IF THE PRE-RESET VALUE IS NOT expected_count, RETURNS -EAGAIN.
- In lines 42~44 of the code, move the index, mapping, and reference counters to the new page of the mapped page.
- Move the SwapBacked flag from lines 45~53 of code. It also moves the SwapCache flag and moves the values stored in the private data.

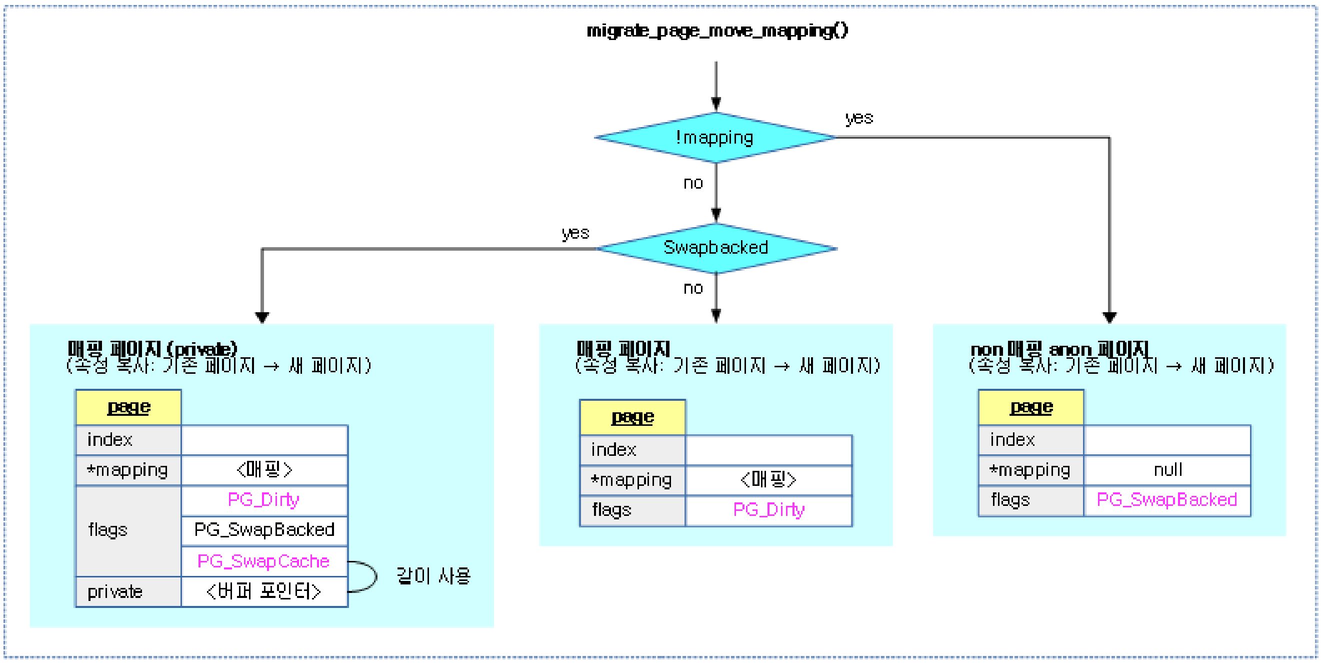
mm/migrate.c -2/2-

```
01         /* Move dirty while page refs frozen and newpage not yet exposed */
02         dirty = PageDirty(page);
03         if (dirty) {
04                 ClearPageDirty(page);
05                 SetPageDirty(newpage);
06         }
07 
08         xas_store(&xas, newpage);
09         if (PageTransHuge(page)) {
10                 int i;
11 
12                 for (i = 1; i < HPAGE_PMD_NR; i++) {
13                         xas_next(&xas);
14                         xas_store(&xas, newpage + i);
15                 }
16         }
17 
18         /*
19          * Drop cache reference from old page by unfreezing
20          * to one less reference.
21          * We know this isn't the last reference.
22          */
23         page_ref_unfreeze(page, expected_count - hpage_nr_pages(page));
24 
25         xas_unlock(&xas);
26         /* Leave irq disabled to prevent preemption while updating stats */
27 
28         /*
29          * If moved to a different zone then also account
30          * the page for that zone. Other VM counters will be
31          * taken care of when we establish references to the
32          * new page and drop references to the old page.
33          *
34          * Note that anonymous pages are accounted for
35          * via NR_FILE_PAGES and NR_ANON_MAPPED if they
36          * are mapped to swap space.
37          */
38         if (newzone != oldzone) {
39                 __dec_node_state(oldzone->zone_pgdat, NR_FILE_PAGES);
40                 __inc_node_state(newzone->zone_pgdat, NR_FILE_PAGES);
41                 if (PageSwapBacked(page) && !PageSwapCache(page)) {
42                         __dec_node_state(oldzone->zone_pgdat, NR_SHMEM);
43                         __inc_node_state(newzone->zone_pgdat, NR_SHMEM);
44                 }
45                 if (dirty && mapping_cap_account_dirty(mapping)) {
46                         __dec_node_state(oldzone->zone_pgdat, NR_FILE_DIRTY);
47                         __dec_zone_state(oldzone, NR_ZONE_WRITE_PENDING);
48                         __inc_node_state(newzone->zone_pgdat, NR_FILE_DIRTY);
49                         __inc_zone_state(newzone, NR_ZONE_WRITE_PENDING);
50                 }
51         }
52         local_irq_enable();
53 
54         return MIGRATEPAGE_SUCCESS;
55 }
56 EXPORT_SYMBOL(migrate_page_move_mapping);
```

- If there is a dirty flag in lines 2~6 of the code, move it to a new page and remove the existing page.
- In line 8 of code, save the new page to xas xarray. Previously, we used a radix tree, but we changed it to an xarray data structure.
- In line 9~16 of code, if thp, each page belonging to it is stored in xas xarray.
- In line 23 of code, specify the number of page reference counters.

- At line 25 of code, release the xas array lock.
- If the code line 38~51 zone is changed, the value of the relevant counters will be increased or decreased.

The following illustration shows copying the relevant properties from the old page to the new page, depending on the page type.



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

Page migration via Rmap walk

remove_migration_ptes()

mm/migrate.c

```
1  /*
2   * Get rid of all migration entries and replace them by
3   * references to the indicated page.
4   */
```

```
01 static void remove_migration_ptes(struct page *old, struct page *new)
02 {
03         struct rmap_walk_control rwc = {
04                 .rmap_one = remove_migration_pte,
05                 .arg = old,
06         };
07 
08         if (locked)
09                 rmap_walk_locked(new, &rwc);
10         else
11                 rmap_walk(new, &rwc);
12 }
```

Move all mappings linked to the @old page to the @new page.

- In lines 3~6 of the code, prepare to use rmap walk to find all VMAs that refer to existing pages.

   - Rmap -2- (TTU & Rmap Walk) (http://jake.dothome.co.kr/rmap-2) | 문c
- In lines 8~11 of the code, remove the existing mapping linked to the relevant VMA via rmap walk and move the mapping to a new page.

remove_migration_pte()

mm/migrate.c

```c
/*
 * Restore a potential migration pte to a working pte entry
 */
```

```c
static bool remove_migration_pte(struct page *page, struct vm_area_struct *vma,
                                 unsigned long addr, void *old)
{
        struct page_vma_mapped_walk pvmw = {
                .page = old,
                .vma = vma,
                .address = addr,
                .flags = PVMW_SYNC | PVMW_MIGRATION,
        };
        struct page *new;
        pte_t pte;
        swp_entry_t entry;

        VM_BUG_ON_PAGE(PageTail(page), page);
        while (page_vma_mapped_walk(&pvmw)) {
                if (PageKsm(page))
                        new = page;
                else
                        new = page - pvmw.page->index +
                                linear_page_index(vma, pvmw.address);

#ifdef CONFIG_ARCH_ENABLE_THP_MIGRATION
                /* PMD-mapped THP migration entry */
                if (!pvmw.pte) {
                        VM_BUG_ON_PAGE(PageHuge(page) || !PageTransCompound(page), page);
                        remove_migration_pmd(&pvmw, new);
                        continue;
                }
#endif

                get_page(new);
                pte = pte_mkold(mk_pte(new, READ_ONCE(vma->vm_page_prot)));
                if (pte_swp_soft_dirty(*pvmw.pte))
                        pte = pte_mksoft_dirty(pte);

                /*
                 * Recheck VMA as permissions can change since migration started
                 */
                entry = pte_to_swp_entry(*pvmw.pte);
                if (is_write_migration_entry(entry))
                        pte = maybe_mkwrite(pte, vma);

                if (unlikely(is_zone_device_page(new))) {
                        if (is_device_private_page(new)) {
                                entry = make_device_private_entry(new, pte_write(pte));
                                pte = swp_entry_to_pte(entry);
                        } else if (is_device_public_page(new)) {
```

```
48                                 pte = pte_mkdevmap(pte);
49                                 flush_dcache_page(new);
50                         }
51                 } else
52                         flush_dcache_page(new);
53 
54 #ifdef CONFIG_HUGETLB_PAGE
55                 if (PageHuge(new)) {
56                         pte = pte_mkhuge(pte);
57                         pte = arch_make_huge_pte(pte, vma, new, 0);
58                         set_huge_pte_at(vma->vm_mm, pvmw.address, pvmw.pte, pte);
59                         if (PageAnon(new))
60                                 hugepage_add_anon_rmap(new, vma, pvmw.address);
61                         else
62                                 page_dup_rmap(new, true);
63                 } else
64 #endif
65                 {
66                         set_pte_at(vma->vm_mm, pvmw.address, pvmw.pte, pte);
67 
68                         if (PageAnon(new))
69                                 page_add_anon_rmap(new, vma, pvmw.address, false);
70                         else
71                                 page_add_file_rmap(new, false);
72                 }
73                 if (vma->vm_flags & VM_LOCKED && !PageTransCompound(new))
74                         mlock_vma_page(new);
75 
76                 if (PageTransHuge(page) && PageMlocked(page))
77                         clear_page_mlock(page);
78 
79                 /* No need to invalidate - it was non-present before */
80                 update_mmu_cache(vma, pvmw.address, pvmw.pte);
81         }
82 
83         return true;
84 }
```

Move all mappings linked to the @old page to the @new page.

- Repeat until all the mappings for the old page on lines 4~15 of the code have been removed.
  - Rmap -3- (PVMW) (http://jake.dothome.co.kr/rmap-3) | Qc
- Retrieve the new page to be migrated from lines 16~20 of the code.
- If the THP entry is mapped to PMD in code lines 24~28, perform the migration to PMD entry and then continue.
- Increment the reference counter from code lines 31~34 to use the new page. Then prepare a PTE entry for this page. The soft dirty status of the existing PTE entry will also be transferred.
- In line 39~41 of the code, if the swap entry uses the VMA of the write attribute in a swap entry that can be migrated, the write attribute is added to the pte attribute.
  - If permissions have changed since the migration started, check the VMA again with the write attribute and add them if they have changed.
- In lines 43~52 of the code, flush the data cache for the new page.
  - If you are a zone device that operates a private page, it will fetch the PTE entry for the new page.

    - If it is a zone device that is not a private page, it fetches the PTE entry for the new page and additionally flush the data cache for the new page.
- In line 55~62 of the code, if the new page is a huge page, add the regular mapping, and also add it to rmap.
- In line 63~72 of the code, add a regular mapping if it is not a huge page, and also add it to rmap.
- In line 73~74 of code, the VMA with the VM_LOCKED flag should set the mlock flag on the new page only if it is not a thp and hugetlbfs page.
- In code lines 76~77, if the mlocked flag is set on the new thp, clear it.
    - Note that thp can't set mlock.
- In line 80 of code, perform a flush on an architecture that requires a cache flush.
    - It doesn't do anything on ARMv6 or higher, but flush routines for cache coherent on ARMv6 and below.

Copy page frames

migrate_page_copy()

mm/migrate.c

```
01 void migrate_page_copy(struct page *newpage, struct page *page)
02 {
03         if (PageHuge(page) || PageTransHuge(page))
04                 copy_huge_page(newpage, page);
05         else
06                 copy_highpage(newpage, page);
07
08         migrate_page_states(newpage, page);
09 }
10 EXPORT_SYMBOL(migrate_page_copy);
```

Copy the existing page frame to the new page frame. It also copies the relevant page descriptors.

- In line 3~4 of the code, perform the copy routine for huge pages or THP.
- In line 5~6 of the code, perform the copy routine for a normal page that is not otherwise.
- Copy the page descriptor from line 8 as well.

copy_huge_page()

mm/migrate.c

```
01 static void copy_huge_page(struct page *dst, struct page *src)
02 {
03         int i;
04         int nr_pages;
05
06         if (PageHuge(src)) {
07                 /* hugetlbfs page */
08                 struct hstate *h = page_hstate(src);
09                 nr_pages = pages_per_huge_page(h);
10
11                 if (unlikely(nr_pages > MAX_ORDER_NR_PAGES)) {
12                         __copy_gigantic_page(dst, src, nr_pages);
13                         return;
14                 }
15         } else {
```

```
16                      /* thp page */
17                      BUG_ON(!PageTransHuge(src));
18                      nr_pages = hpage_nr_pages(src);
19              }
20
21              for (i = 0; i < nr_pages; i++) {
22                      cond_resched();
23                      copy_highpage(dst + i, src + i);
24              }
25      }
```

@src Copy the huge page frame into the @dst.

- In line 6~14 of code, if the hugetlbfs page is larger than the maximum number of order pages, simply incrementing the page struct pointer does not guarantee that it will behave correctly. Therefore, it is handled in a separate function to accurately handle the pointer of the page descriptor.
- In line 15~19 of the code, we find out the number of pages for thp.
- Travers the number of pages in line 21~24 of the code and copy the page frame of @src+i to @dst+i.

__copy_gigantic_page()

mm/migrate.c

```
/*
 * Gigantic pages are so large that we do not guarantee that page++ pointer
 * arithmetic will work across the entire page.  We need something more
 * specialized.
 */
```

```
static void __copy_gigantic_page(struct page *dst, struct page *src,
                                int nr_pages)
{
        int i;
        struct page *dst_base = dst;
        struct page *src_base = src;

        for (i = 0; i < nr_pages; ) {
                cond_resched();
                copy_highpage(dst, src);

                i++;
                dst = mem_map_next(dst, dst_base, i);
                src = mem_map_next(src, src_base, i);
        }
}
```

Copy @src page frames larger than the maximum number of order pages to @dst.

mem_map_next()

mm/internal.h

```
/*
 * Iterator over all subpages within the maximally aligned gigantic
 * page 'base'.  Handle any discontiguity in the mem_map.
 */
```

```
01 static inline struct page *mem_map_next(struct page *iter,
02                                                 struct page *base, int offset)
03 {
04         if (unlikely((offset & (MAX_ORDER_NR_PAGES - 1)) == 0)) {
05                 unsigned long pfn = page_to_pfn(base) + offset;
06                 if (!pfn_valid(pfn))
07                         return NULL;
08                 return pfn_to_page(pfn);
09         }
10         return iter + 1;
11 }
```

Retrieves the next page of the @iter page and returns it. In the case of pages exceeding the maximum number of order pages, there is a possibility that the wrong address may be obtained by exceeding the boundaries of the mem_map managed by section, so the address of the page descriptor is recalculated correctly each time.

- In line 4~9 of the code, if the offset is ordered by the maximum number of order pages (default: 1024 = 4M (4K pages)), incrementing the page descriptor pointer may exceed the boundary of mem_map, so get the pfn value first, and then use the pfn_to_page() function to get the page descriptor again.
- Returns the next page descriptor on line 10 of the code.

copy_highpage()

include/linux/highmem.h

```
01 static inline void copy_highpage(struct page *to, struct page *from)
02 {
03         char *vfrom, *vto;
04 
05         vfrom = kmap_atomic(from);
06         vto = kmap_atomic(to);
07         copy_page(vto, vfrom);
08         kunmap_atomic(vto);
09         kunmap_atomic(vfrom);
10 }
```

@from Copy the page frame to the @to.

- In line 5~6 of the code, if it is a highmem page on a 32-bit system, use fixmap to map it temporarily.
- In line 7 of code, copy the page frame in the fastest way that the architecture supports.
- Unmap the page temporarily mapped to fixmap in line 8~9 of the code.

Copying Page Descriptor Information

migrate_page_states()

mm/migrate.c

```
/*
 * Copy the page to its new location
 */
```

```
01 void migrate_page_states(struct page *newpage, struct page *page)
02 {
03         int cpupid;
04 
05         if (PageError(page))
06                 SetPageError(newpage);
07         if (PageReferenced(page))
08                 SetPageReferenced(newpage);
09         if (PageUptodate(page))
10                 SetPageUptodate(newpage);
11         if (TestClearPageActive(page)) {
12                 VM_BUG_ON_PAGE(PageUnevictable(page), page);
13                 SetPageActive(newpage);
14         } else if (TestClearPageUnevictable(page))
15                 SetPageUnevictable(newpage);
16         if (PageWorkingset(page))
17                 SetPageWorkingset(newpage);
18         if (PageChecked(page))
19                 SetPageChecked(newpage);
20         if (PageMappedToDisk(page))
21                 SetPageMappedToDisk(newpage);
22 
23         /* Move dirty on pages not done by migrate_page_move_mapping() */
24         if (PageDirty(page))
25                 SetPageDirty(newpage);
26 
27         if (page_is_young(page))
28                 set_page_young(newpage);
29         if (page_is_idle(page))
30                 set_page_idle(newpage);
31 
32         /*
33          * Copy NUMA information to the new page, to prevent over-eager
34          * future migrations of this same page.
35          */
36         cpupid = page_cpupid_xchg_last(page, -1);
37         page_cpupid_xchg_last(newpage, cpupid);
38 
39         ksm_migrate_page(newpage, page);
40         /*
41          * Please do not reorder this without considering how mm/ksm.c's
42          * get_ksm_page() depends upon ksm_migrate_page() and PageSwapCache().
43          */
44         if (PageSwapCache(page))
45                 ClearPageSwapCache(page);
46         ClearPagePrivate(page);
47         set_page_private(page, 0);
48 
49         /*
50          * If any waiters have accumulated on the new page then
51          * wake them up.
52          */
53         if (PageWriteback(newpage))
54                 end_page_writeback(newpage);
55 
56         copy_page_owner(page, newpage);
57 
58         mem_cgroup_migrate(page, newpage);
59 }
60 EXPORT_SYMBOL(migrate_page_states);
```

Move the contents of an existing page descriptor to a descriptor on a new page.

- Move the PG_error flag from lines 5~6 of the code.
- Move the PG_referenced flag from lines 7~8 of code.

- Move the PG_uptodate flag from lines 9~10 of code.
- Move the PG_active flag or PG_unevictable flag from lines 11~15 of the code, and remove it from the existing page.
- Move the PG_workingset flag from lines 16~17 of code.
- Move the PG_checked flag from lines 18~19 of code.
- Move the PG_mappedtodist flag from lines 20~21 of code.
- Move the PG_dirty flag from lines 24~25 of the code.
- Move the Young flag from lines 27~28 of code.
    - On a 32-bit system, it exists in a page_ext structure, not a page structure.
- Move the idle flag from lines 29~30 of code.
    - On a 32-bit system, it exists in a page_ext structure, not a page structure.
- Move the cpupid information from lines 36~37 and assign -1 to the existing page.
- In line 39 of the code, if it is a KSM page and the target to be copied is mapped to a statble node, remove the stable node mapping information.
    - KSM (Kernel Same Page Merging) is a feature that allows an application to scan pages with the same content in the address space and merge them into a single page.
    - 참고: How to use the Kernel Samepage Merging feature (http://www.kernel.org/doc/Documentation/vm/ksm.txt) | kernel.org
- Remove the SwapCache information from the existing page in lines 44~45 of the code.
- Remove the private flag from the existing page in lines 46~47 and assign 0 to p->private.
- If the Writeback flag is set on a new page in line 53~54 of the code, rotate it with the lru's tail, wake up the relevant task, and recall it immediately.
- Replace page owner information on line 56.
- Moving the memcg information from line 58 of the code.

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) | 문c
- Zoned Allocator -9- (Direct Compact-Migration) (http://jake.dothome.co.kr/zonned-allocator-migration) | Sentence C – Current post
- 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

LEAVE A COMMENT

Your email will not be published. Required fields are marked with *

Comments

name *

email *

Website

WRITE A COMMENT

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

Zoned Allocator -12- (Direct Reclaim-Shrink-1) (http://jake.dothome.co.kr/zonned-allocator-shrink-1/)