Swap -2- (Swapin & Swapout)

2019-10-29 (http://jake.dothome.co.kr/swap-2/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Swapin & Swapout

If a user process is swapped and accesses an empty unmapped virtual address, it must be recovered by loading the swap page in the swap area. At this time, the process of loading the swapped page in the swap area through the swap cache is called swap-in, and the process of saving it is called swap-out.

Swap readahead

If it is swapped during the swap-in process and approaches the empty area, a fault error occurs. At this time, it is possible to know the swap entry stored in the page table with the faulted virtual address and use it as a key to load the swap page in the relevant swap area, and to preload the surrounding pages to increase processing performance. This method is called readahead, and the readahead used in the swap process uses two methods as follows.

- VMA-based Swap ReadAhead
- cluster 기반 swap readahead (for SSD)

Pages that have been read together with the readahead function are flagged PG_reclaim when they are in the swap cache (to save on the use of the flag, readahead is used for PG_readahead purposes, not reclaim).

VMA-based swap readahead

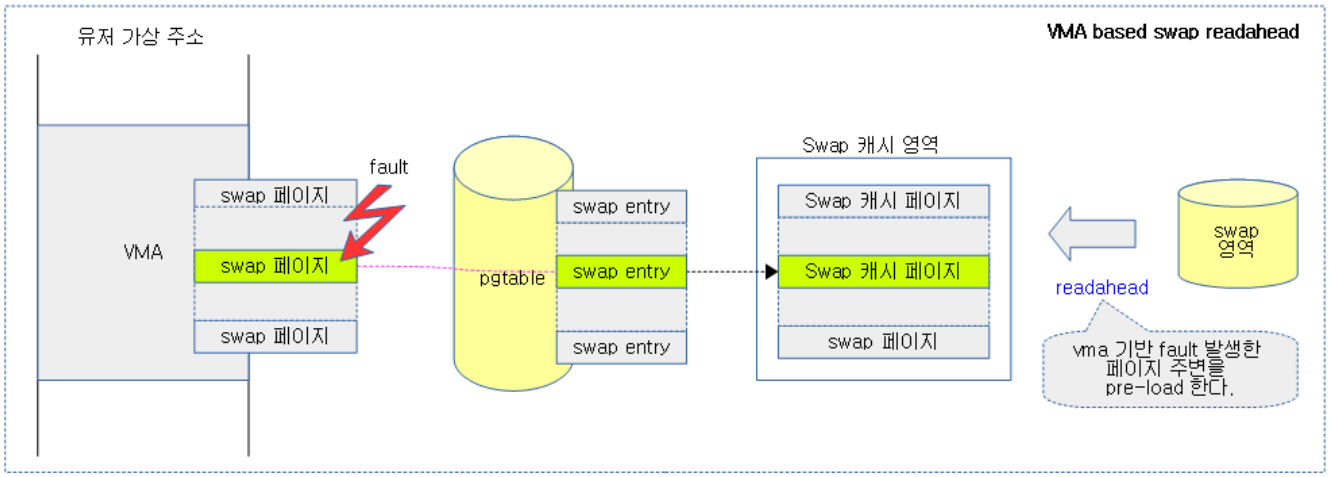
If a user process encounters a fault when it accesses a swapped page, it can look for the page in the swap cache, and if it does not find it in the swap cache, it can read a little more around the faulted page in the VMA area from the swap area, and when it fails, the page that has already been read can be found in the swap cache area, so that it can be quickly converted and mapped to an anon page.

- To use this method, you must have the vma_ra_enabled attribute set in the kernel configuration (default=true) and it can only be used on SSD-type block devices. In the swap area using HDDs, VMA-based swap readahead was no longer allowed to be used due to performance degradation.
- Consultation:
  - mm, swap: don't use VMA based swap readahead if HDD is used as swap (https://github.com/torvalds/linux/commit/81a0298bdfab0203d360df7c9bf690d1d457f999

#diff-58353ba1b37b7ca43f95c0787f5f39bd) (v4.14-rc1)

- mm, swap: VMA based swap readahead (https://lwn.net/Articles/716296/) (2017) | LWN.net
- mm, swap: VMA based swap readahead (https://lwn.net/Articles/730073/) (2017, patch) | LWN.net

The following figure shows the process by which when a swapped page is loaded in the swap area, some of the surrounding pages in the VMA of the fault page are loaded into the swap cache in advance.

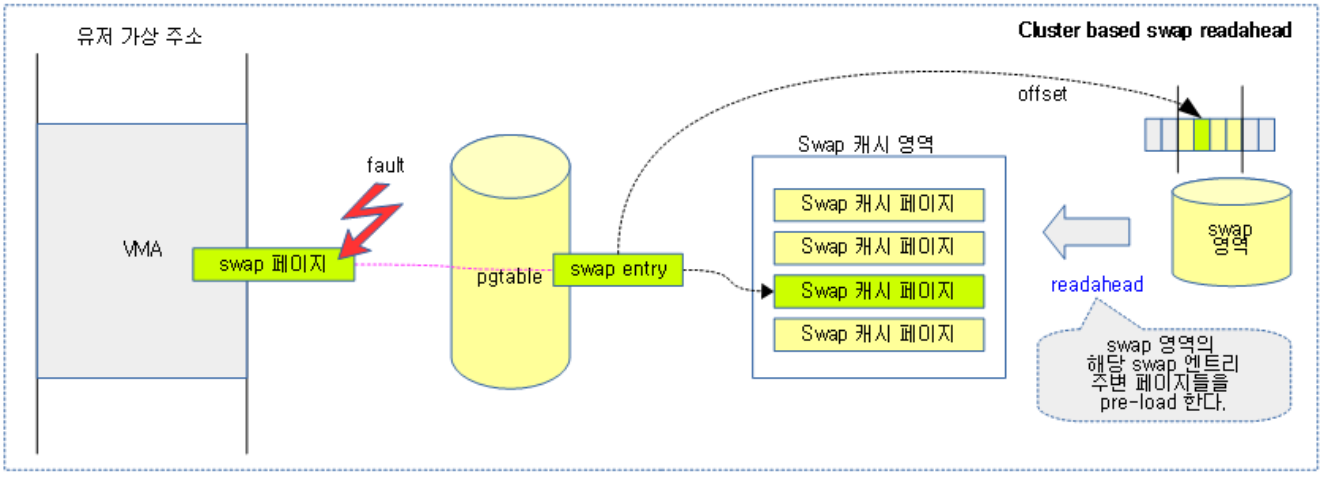


(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-1a.png)

Cluster-based swap readahead

Unlike the VMA base, it reads more of the pages around the swap area, not around the faulted swap page.

The following figure shows the process by which when a swapped page is loaded in the swap area, some of the surrounding pages in the swap area are pre-loaded into the swap cache.

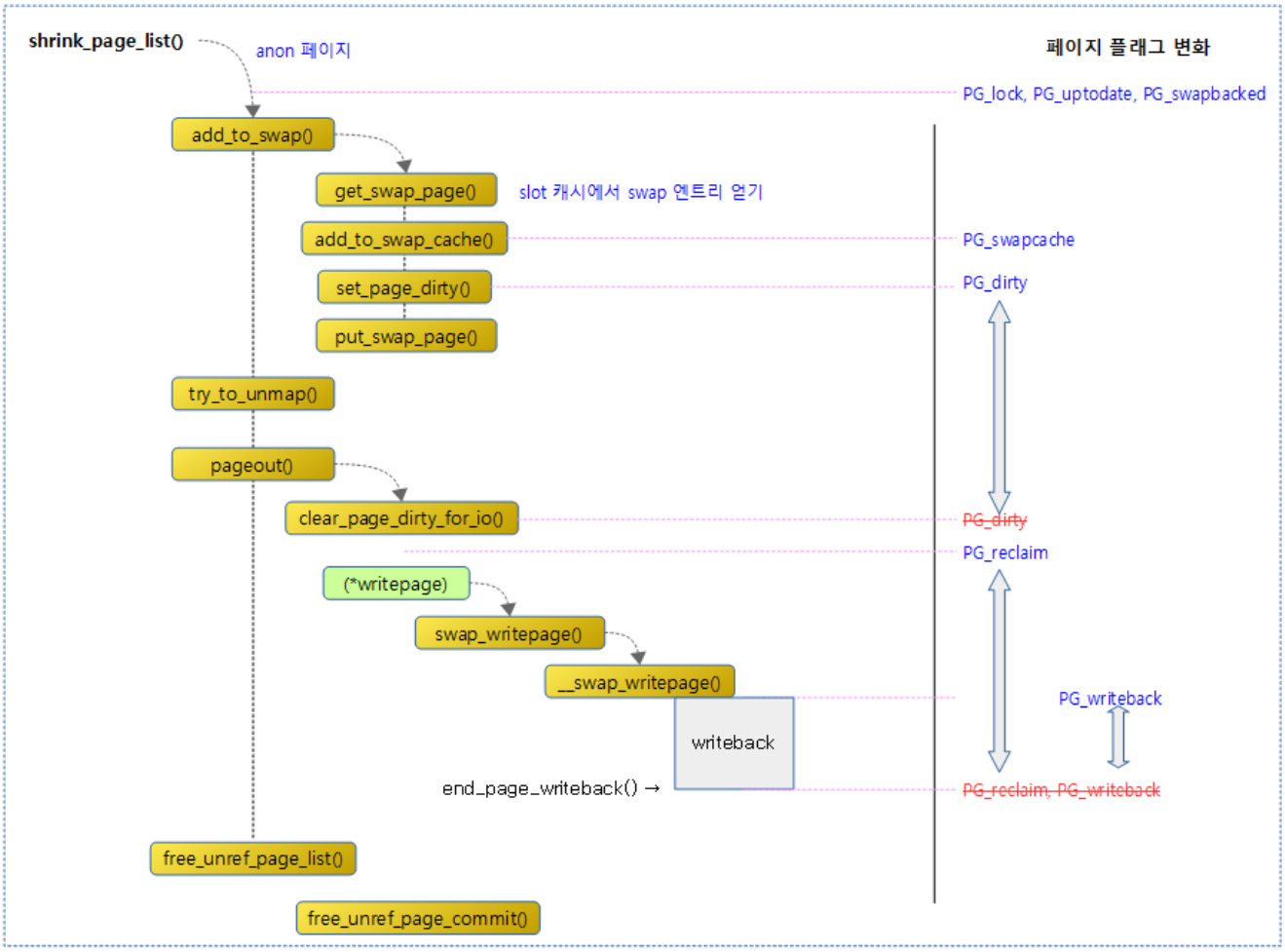


(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-19.png)

Swap Related Page Flags

- PG_swapbacked
  - It is whether or not you have a swap zone.
    - With this flag, this is a normal anon page that can be swapped.
    - Without this flag, the clean anon page would not be able to swap.
- PG_swapcached
  - swap and exist in the swap cache area. When a user accesses the swapped virtual address page, the fault handler uses a fault handler to find the swap cache before the swap area.
    - Whether it is written to the swap area cannot be determined by this flag, but by the page_swapped() function.
  - When a swap-in is in progress, it also reads the surrounding pages and loads them into the swap cache area to improve performance when reading from the swap area.
  - When a swap-out is in progress, this swap cache is stored in the swap area.
- PG_writeback
  - It is set during a write (sync or async) to the swap area.
  - It is set in the swap_writepage() function, which is the swap writeback hook function in pageout(), and is cleared when the writeback is completed.
- PG_reclaim (2 uses)
  - PageReclaim() in case of swap-out.
    - It is set during writing to the swap area for reclaim, and can be reclaimed when this flag is removed.
    - pageout() just before the writeback, and cleared when the writeback is complete.
  - PageReadahead() in case of swap-in.
    - It is set to the swap cache page that has been read in advance with readahead.
- PG_dirty
  - swap area, and pageout() is called when this flag is displayed.
  - add_to_swap_cache() function, and cleared just before the writeback in pageout().
- PG_workingset
  - This flag indicates that the page is in the works.
    - When a file page is refaulted in the inactive LRU list, it compares the fault interval to the memory size, and uses the workingset flag to check if the fault interval is smaller than the memory size and use it to detect thrashing.
    - It is a solution to prevent frequently used pages from being fualtized multiple times and degrading performance due to cache replacement.
    - 참고: mm: workingset: tell cache transitions from workingset thrashing (https://github.com/torvalds/linux/commit/1899ad18c6072d689896badafb81267b0a1092a4#diff-d6f69008183f37b689f73cd6f4d5440c) (2018, v4.20-rc1)
  - When the file page is first accessed, it starts at the top of the inactive LRU list. If it detects two accesses, it will be promoted to Active LRU. However, the ANON page starts at the top of the Active LRU list when it is first accessed, so it is set to WorkingSet from the beginning.

The following figure shows the change in the swap-related page flags.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-4.png)

Initialize Swap

swap_init_sysfs()

mm/swap_state.c

```
01 static int __init swap_init_sysfs(void)
02 {
03         int err;
04         struct kobject *swap_kobj;
05
06         swap_kobj = kobject_create_and_add("swap", mm_kobj);
07         if (!swap_kobj) {
08                 pr_err("failed to create swap kobject\n");
09                 return -ENOMEM;
10         }
11         err = sysfs_create_group(swap_kobj, &swap_attr_group);
12         if (err) {
13                 pr_err("failed to register swap group\n");
14                 goto delete_obj;
15         }
16         return 0;
17
18 delete_obj:
19         kobject_put(swap_kobj);
20         return err;
21 }
22 subsys_initcall(swap_init_sysfs);
```

Create the /sys/kernel/mm/swap directory for the swap system and create the associated property (vma_ra_enabled) file.

vma_ra_enabled Properties

mm/swap_state.c

```
01 static ssize_t vma_ra_enabled_show(struct kobject *kobj,
02                                     struct kobj_attribute *attr, char *
   buf)
03 {
04         return sprintf(buf, "%s\n", enable_vma_readahead ? "true" : "fal
   se");
05 }
06 static ssize_t vma_ra_enabled_store(struct kobject *kobj,
07                                      struct kobj_attribute *attr,
08                                      const char *buf, size_t count)
09 {
10         if (!strncmp(buf, "true", 4) || !strncmp(buf, "1", 1))
11                 enable_vma_readahead = true;
12         else if (!strncmp(buf, "false", 5) || !strncmp(buf, "0", 1))
13                 enable_vma_readahead = false;
14         else
15                 return -EINVAL;
16
17         return count;
18 }
19 static struct kobj_attribute vma_ra_enabled_attr =
20         __ATTR(vma_ra_enabled, 0644, vma_ra_enabled_show,
21                vma_ra_enabled_store);
22
23 static struct attribute *swap_attrs[] = {
24         &vma_ra_enabled_attr.attr,
25         NULL,
26 };
27
28 static struct attribute_group swap_attr_group = {
29         .attrs = swap_attrs,
30 };
```

This is an attribute file that enables the VMA-based readahead function.

- The default value of the "/sys/kernel/mm/swap/vma_ra_enabled" attribute is true.
  - swap_vma_readahead - > renamed to vma_ra_enabled attributes.
- The swap_use_vma_readahead() function tells us if this property is set.

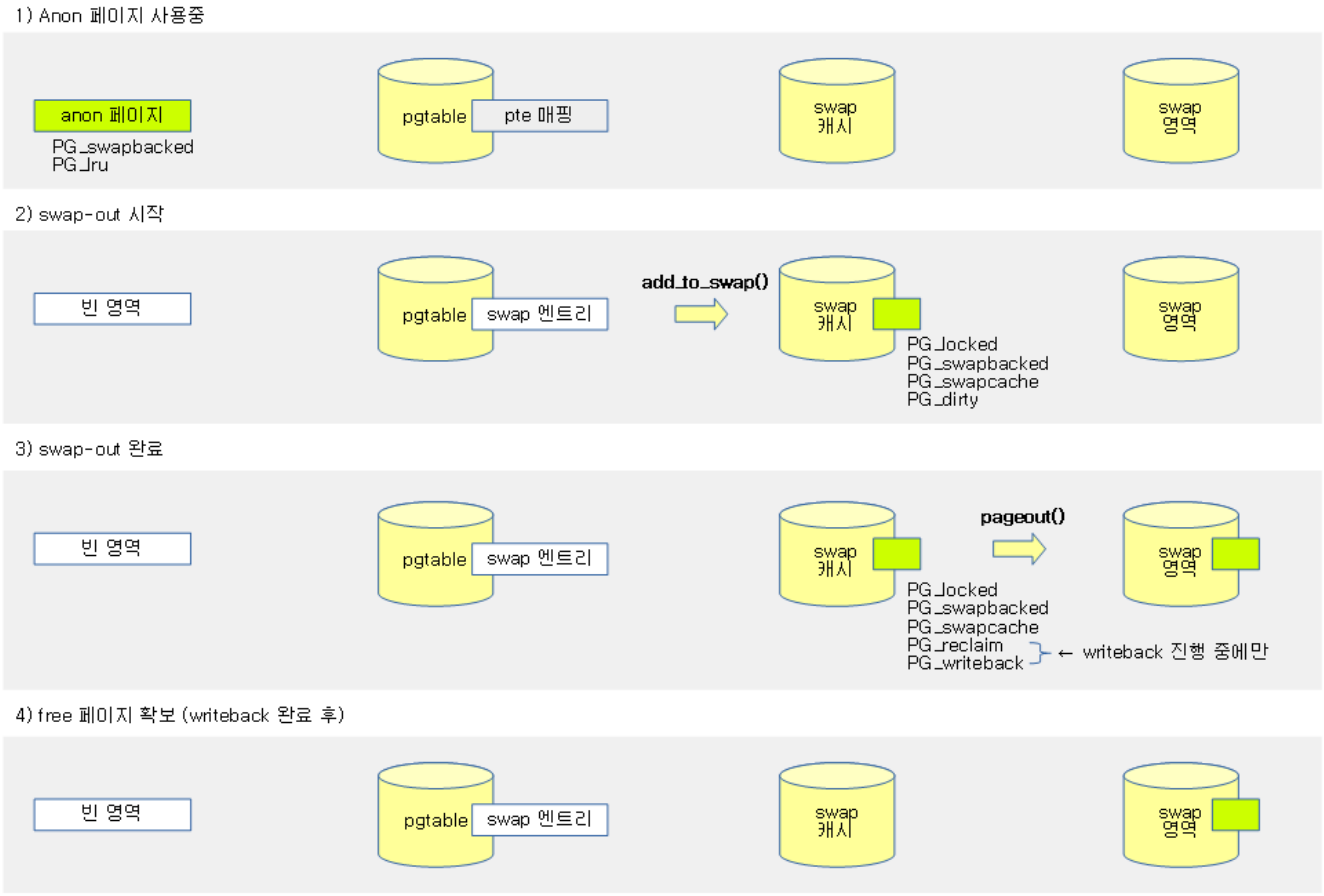
Swap-out

After a normal Anon page is written to the swap area, the order in which the page becomes free is as follows:

- normal anon page → swapcache → unmap → write out → free page
  - add_to_swap()
  - try_to_unmap()
  - pageout()

- free_unref_page_commit()

The following figure shows the swap-out process.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-2.png)

Add to Swap Area

add_to_swap()

mm/swap_state.c

```
1  /**
2   * add_to_swap - allocate swap space for a page
3   * @page: page we want to move to swap
4   *
5   * Allocate swap space for the page and add the page to the
6   * swap cache.  Caller needs to hold the page lock.
7   */
```

```
01 int add_to_swap(struct page *page)
02 {
03         swp_entry_t entry;
04         int err;
05
06         VM_BUG_ON_PAGE(!PageLocked(page), page);
07         VM_BUG_ON_PAGE(!PageUptodate(page), page);
08
09         entry = get_swap_page(page);
10         if (!entry.val)
11                 return 0;
12
13         /*
```

```
14          * XArray node allocations from PF_MEMALLOC contexts could
15          * completely exhaust the page allocator. __GFP_NOMEMALLOC
16          * stops emergency reserves from being allocated.
17          *
18          * TODO: this could cause a theoretical memory reclaim
19          * deadlock in the swap out path.
20          */
21         /*
22          * Add it to the swap cache.
23          */
24         err = add_to_swap_cache(page, entry,
25                         __GFP_HIGH|__GFP_NOMEMALLOC|__GFP_NOWARN);
26         if (err)
27                 /*
28                  * add_to_swap_cache() doesn't return -EEXIST, so we can safely
29                  * clear SWAP_HAS_CACHE flag.
30                  */
31                 goto fail;
32         /*
33          * Normally the page will be dirtied in unmap because its pte should be
34          * dirty. A special case is MADV_FREE page. The page'e pte could have
35          * dirty bit cleared but the page's SwapBacked bit is still set because
36          * clearing the dirty bit and SwapBacked bit has no lock protected. For
37          * such page, unmap will not set dirty bit for it, so page reclaim will
38          * not write the page out. This can cause data corruption when the page
39          * is swap in later. Always setting the dirty bit for the page solves
40          * the problem.
41          */
42         set_page_dirty(page);
43
44         return 1;
45
46 fail:
47         put_swap_page(page, entry);
48         return 0;
49 }
```

After assigning a swap entry, use this value as the key to store the anon page in the swap cache and swap area. Returns 1 on success

- Before swap-out in line 6~7 of code, the PG_lock and PG_uptodate must be set.
- In line 9~11 of the code, get the swap entry to be used for the anon page to be swapped.
- In lines 24~31 of code, add the anon page to be swapped with the swap entry key to the swap cache.
- In lines 42~44 of the code, set the page to dirty. It then returns 1 because it succeeded.
    - For pages mapped to address_space, the driver is used to set the dirty setting, and the page is also set to PG_dirty flag.
    - The dirty page is stored in the swap area by the pageout() function is called during the reclaim process, and the PG_dirty flag is cleared after completion.
- In line 46~48 of the code, the fail: label is. Returns 0 because it failed.

Add to Swap Cache

add_to_swap_cache()

mm/swap_state.c

```
1  /*
2   * add_to_swap_cache resembles add_to_page_cache_locked on swapper_spac
   e,
3   * but sets SwapCache flag and private instead of mapping and index.
4   */
```

```
01 int add_to_swap_cache(struct page *page, swp_entry_t entry, gfp_t gfp)
02 {
03         struct address_space *address_space = swap_address_space(entry);
04         pgoff_t idx = swp_offset(entry);
05         XA_STATE_ORDER(xas, &address_space->i_pages, idx, compound_order
   (page));
06         unsigned long i, nr = 1UL << compound_order(page);
07
08         VM_BUG_ON_PAGE(!PageLocked(page), page);
09         VM_BUG_ON_PAGE(PageSwapCache(page), page);
10         VM_BUG_ON_PAGE(!PageSwapBacked(page), page);
11
12         page_ref_add(page, nr);
13         SetPageSwapCache(page);
14
15         do {
16                 xas_lock_irq(&xas);
17                 xas_create_range(&xas);
18                 if (xas_error(&xas))
19                         goto unlock;
20                 for (i = 0; i < nr; i++) {
21                         VM_BUG_ON_PAGE(xas.xa_index != idx + i, page);
22                         set_page_private(page + i, entry.val + i);
23                         xas_store(&xas, page + i);
24                         xas_next(&xas);
25                 }
26                 address_space->nrpages += nr;
27                 __mod_node_page_state(page_pgdat(page), NR_FILE_PAGES, n
   r);
28                 ADD_CACHE_INFO(add_total, nr);
29 unlock:
30                 xas_unlock_irq(&xas);
31         } while (xas_nomem(&xas, gfp));
32
33         if (!xas_error(&xas))
34                 return 0;
35
36         ClearPageSwapCache(page);
37         page_ref_sub(page, nr);
38         return xas_error(&xas);
39 }
```

Add the anon page to the swap cache with the swap entry information key. Returns 0 on success.

- In line 3 of code, use the swap entry to get the address_space pointer for swap.
- In line 4 of the code, read only the offset part as a swap entry and assign it to IDX.
- In line 5 of code, declare the xarray operation state.
  - The xarray to work with is &address_space->i_pages, and specifies the initial index (idx) and the order of the entries.
  - The XArray data structure (https://lwn.net/Articles/745073/) (2018) | LWN.net
- In line 6 of code, assign nr to the number of compound pages.

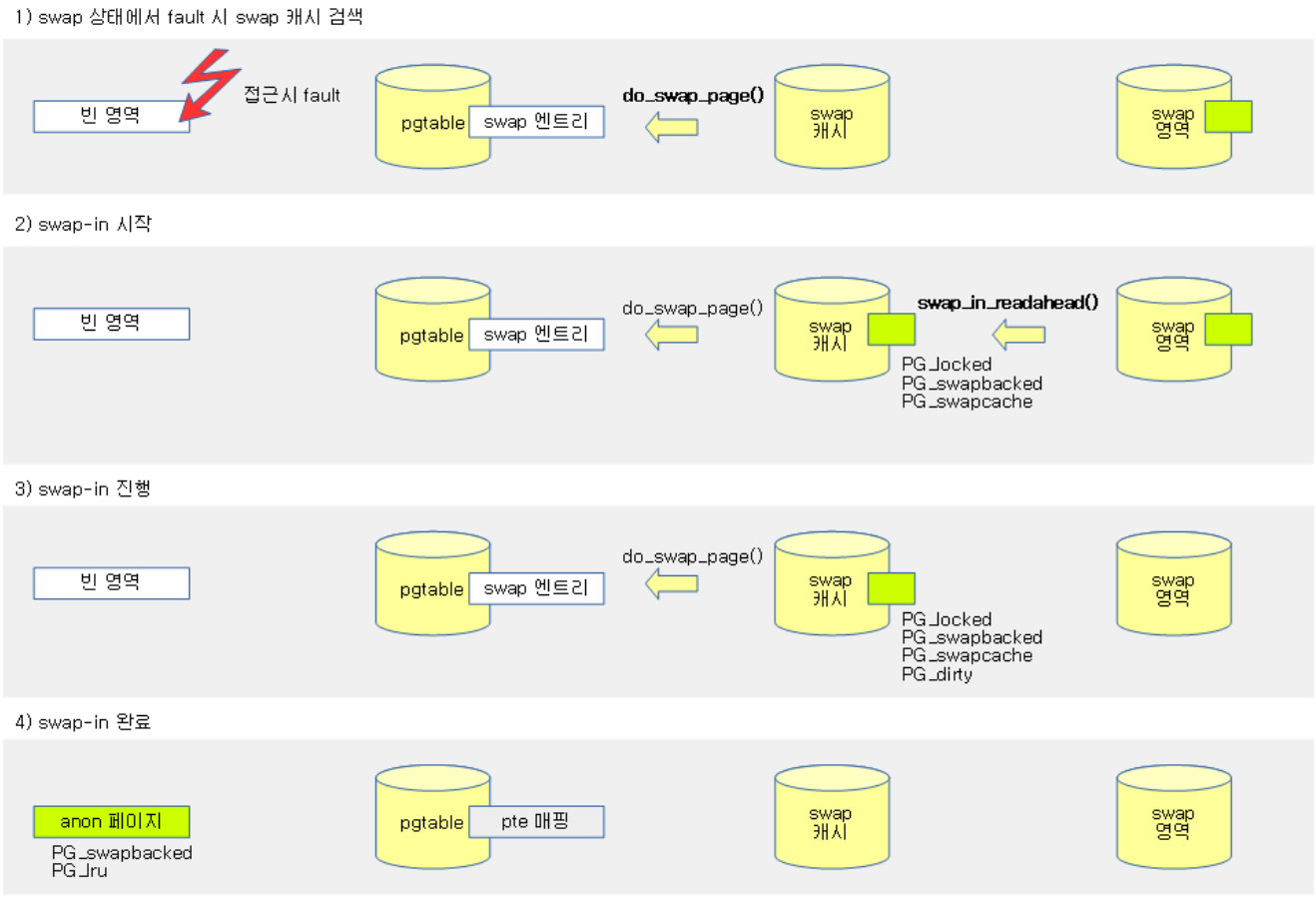
    - For a normal page, 1 is substituted, but for THP, the number of pages that are composed of a compound is substituted.
- In line 8~10 of code, the page to be added to the swap cache area managed by xarray must have a PG_locked, PG_swapbacked flag setting, and no PG_swapcache flag.
- In lines 12~13 of the code, increment the reference counter of the page to be added to the swap cache area by nr, and set the PG_swapcache flag.
    - If THP swap is supported, increment the reference counter on the head page by nr.
- In line 15~19 of the code, prepare an xarray with a range of idx to nr pages.
- Traversing the number of pages in line 20~25 of the code, storing swap entries in p->private, and storing the pages in xarray.
- In lines 26~27 of code, increment the following counters by the number of NR pages:
    - Total number of pages managed by address_space
    - NR_FILE_PAGES Counter
    - swap_cache_info->add_total Counter
- In code lines 29~31, it is labeled unlock: If xa_node allocation fails, reassign and repeat.
- Returns 33 if the assignment was successful on lines 34~0 of the code.
- If the allocation fails on line 36~38 of the code, clear the PG_swapcache, lower the reference counter by nr again, and return an error code.

Swap-in

The swapped pages will be restored in the following order.

- New Page Allocation → Change to swapcache → Map read → in swap area (file/partition)

The following figure shows the swap-in process.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-3.png)

swapin_readahead()

mm/swap_state.c

```
01 /**
02  * swapin_readahead - swap in pages in hope we need them soon
03  * @entry: swap entry of this memory
04  * @gfp_mask: memory allocation flags
05  * @vmf: fault information
06  *
07  * Returns the struct page for entry and addr, after queueing swapin.
08  *
09  * It's a main entry function for swap readahead. By the configuration,
10  * it will read ahead blocks by cluster-based(ie, physical disk based)
11  * or vma-based(ie, virtual address based on faulty address) readahead.
12  */
```

```
1 struct page *swapin_readahead(swp_entry_t entry, gfp_t gfp_mask,
2                                struct vm_fault *vmf)
3 {
4         return swap_use_vma_readahead() ?
5                         swap_vma_readahead(entry, gfp_mask, vmf) :
6                         swap_cluster_readahead(entry, gfp_mask, vmf);
7 }
```

Swap on the swap entry to load the page.

- If you use the "/sys/kernel/mm/swap/vma_ra_enabled" attribute, use VMA-based readahead, otherwise use the cluster-based readahead method used by SSDs, etc.

Swap-in with VMA-based readhead

It is a method of swap-in for faulted swap pages within VMA equal to the number of readahead pages calculated before and after the faulted virtual address. (SSD only)

- The pages to be readaheaded cannot exceed the VMA boundary or the scope of a single PTE table.

swap_vma_readahead()

mm/swap_state.c

```
01 static struct page *swap_vma_readahead(swp_entry_t fentry, gfp_t gfp_mas
   k,
02                                         struct vm_fault *vmf)
03 {
04         struct blk_plug plug;
05         struct vm_area_struct *vma = vmf->vma;
06         struct page *page;
07         pte_t *pte, pentry;
08         swp_entry_t entry;
09         unsigned int i;
10         bool page_allocated;
11         struct vma_swap_readahead ra_info = {0,};
12
13         swap_ra_info(vmf, &ra_info);
14         if (ra_info.win == 1)
15                 goto skip;
16
17         blk_start_plug(&plug);
18         for (i = 0, pte = ra_info.ptes; i < ra_info.nr_pte;
19              i++, pte++) {
20                 pentry = *pte;
21                 if (pte_none(pentry))
22                         continue;
23                 if (pte_present(pentry))
24                         continue;
25                 entry = pte_to_swp_entry(pentry);
26                 if (unlikely(non_swap_entry(entry)))
27                         continue;
28                 page = __read_swap_cache_async(entry, gfp_mask, vma,
29                                                vmf->address, &page_alloc
   ated);
30                 if (!page)
31                         continue;
32                 if (page_allocated) {
33                         swap_readpage(page, false);
34                         if (i != ra_info.offset) {
35                                 SetPageReadahead(page);
36                                 count_vm_event(SWAP_RA);
37                         }
38                 }
39                 put_page(page);
40         }
41         blk_finish_plug(&plug);
42         lru_add_drain();
43 skip:
44         return read_swap_cache_async(fentry, gfp_mask, vma, vmf->addres
   s,
45                                      ra_info.win == 1);
46 }
```

Perform VMA-based swap readahead on swap entries.

- In lines 11~15 of the code, configure the readahead information for the swap. If the page to swap in is at least 1, it will go directly to the skip label.
- In line 17 of the code, initialize the blk_plug and ask the block device to hold submit until the blk_finish_plug() ends.
- Travers the number of PTE entries in code lines 18~27 and skip unless the swap entry information is recorded in the PTE.
- On lines 28~31 of the code, find the page in the swap cache area.
- If the newly allocated page is from code lines 32~38, the page is read by making an asynchronous bio request from the swap area. If the assigned page is not the requested offset page, it sets the PG_reclaim flag (readahead on swap-in) and increments the SWAP_RA counter.
- In line 41 of the code, we use a function paired with the blk_start_plug() function to enable the block device to execute submit from now on.
- In line 42 of code, we revert the per-cpu LRU caches back to LRU.
- On code lines 43~45, the skip: label is. Once again, find the page in the swap cache area. When processing only one page (win=1), the swap cache reads the page in sync mode.

Calculate the number of pages for readahed at swapin

swapin_nr_pages()

mm/swap_state.c

```
01 static unsigned long swapin_nr_pages(unsigned long offset)
02 {
03         static unsigned long prev_offset;
04         unsigned int hits, pages, max_pages;
05         static atomic_t last_readahead_pages;
06
07         max_pages = 1 << READ_ONCE(page_cluster);
08         if (max_pages <= 1)
09                 return 1;
10
11         hits = atomic_xchg(&swapin_readahead_hits, 0);
12         pages = __swapin_nr_pages(prev_offset, offset, hits, max_pages,
13                                   atomic_read(&last_readahead_pages));
14         if (!hits)
15                 prev_offset = offset;
16         atomic_set(&last_readahead_pages, pages);
17
18         return pages;
19 }
```

Calculate the number of pages to readahead on swapin for @offset pages.

- In line 3~5 of the code, the offset value used to calculate the latest readahead is stored in prev_offset, and the number of most recently calculated readahead pages is stored in last_readahead_pages.
- In lines 7~9 of the code, specify a value of 1 << page_cluster as the maximum page limit. If the value is 1, it returns the smallest number, 1, without the need for additional calculations.
- In line 11 of the code, we get the number of pages to hit readahead on swap-in.

- In lines 12~13 of the code, use the values of Recent Offset (prev_offset), @offset, Readahead Hit Pages (HITS), Maximum Page Limit (@max_pages), and Last Calculated ReadAhead Page Count (last_readahead_pages) to calculate the appropriate number of ReadAhead pages.
- In lines 14~15 of the code, if the number of pages in the readahead hit is 0, remember the offset in the prev_offset.
- Remembers the number of readahead pages calculated from lines 16~18 of the code in last_readahead_pages and returns them.

__swapin_nr_pages()

mm/swap_state.c

```
01 static unsigned int __swapin_nr_pages(unsigned long prev_offset,
02                                        unsigned long offset,
03                                        int hits,
04                                        int max_pages,
05                                        int prev_win)
06 {
07         unsigned int pages, last_ra;
08
09         /*
10          * This heuristic has been found to work well on both sequential
    and
11          * random loads, swapping to hard disk or to SSD: please don't a
    sk
12          * what the "+ 2" means, it just happens to work well, that's al
    l.
13          */
14         pages = hits + 2;
15         if (pages == 2) {
16                 /*
17                  * We can have no readahead hits to judge by: but must n
    ot get
18                  * stuck here forever, so check for an adjacent offset i
    nstead
19                  * (and don't even bother to check whether swap type is
    same).
20                  */
21                 if (offset != prev_offset + 1 && offset != prev_offset -
    1)
22                         pages = 1;
23         } else {
24                 unsigned int roundup = 4;
25                 while (roundup < pages)
26                         roundup <<= 1;
27                 pages = roundup;
28         }
29
30         if (pages > max_pages)
31                 pages = max_pages;
32
33         /* Don't shrink readahead too fast */
34         last_ra = prev_win / 2;
35         if (pages < last_ra)
36                 pages = last_ra;
37
38         return pages;
39 }
```

Returns the appropriate number of readahead pages using the most recent offset (@prev_offset), @offset, readahead hitpage (@hits), maximum page limit (@max_pages), and the recently determined readahead page count (@prev_win).

- If there is no recent readahead hit page in code lines 14~22, specify 2 pages. However, if the offset is less than +- 1 from the most recent offset, it will be designated as 1 page.
- In code lines 23~28, if a recent readahead hit page exists, it should be determined by one of the values (4, 4, 8, ...) starting from 16 and increasing by a factor of two, which must be significantly smaller than (hit page + 2).
  - e.g. hits=10
    - pages = 16
- Limit the number of pages calculated from lines 30~31 of the code so that it does not exceed the maximum number of pages.
- In order to prevent the number of pages produced from code lines 34~36 from shrinking rapidly, limit the number of pages that have been readaheaded to less than half of the most recent readaheads.
- Returns the number of readahead pages calculated on line 38 of the code.
  - Consultation:
    - swap: add a simple detector for inappropriate swapin readahead (https://github.com/torvalds/linux/commit/579f82901f6f41256642936d7e632f3979ad76d4) (v3.14-rc2)
    - mm, swap: VMA based swap readahead (https://github.com/torvalds/linux/commit/ec560175c0b6fce86994bdf036754d48122c5c87) (v4.14-rc1)

Swap Cache Fetching Pages

read_swap_cache_async()

mm/swap_state.c

```
1  /*
2   * Locate a page of swap in physical memory, reserving swap cache space
3   * and reading the disk if it is not already cached.
4   * A failure return means that either the page allocation failed or that
5   * the swap entry is no longer in use.
6   */
```

```
01 struct page *read_swap_cache_async(swp_entry_t entry, gfp_t gfp_mask,
02                 struct vm_area_struct *vma, unsigned long addr, bool do_
   poll)
03 {
04         bool page_was_allocated;
05         struct page *retpage = __read_swap_cache_async(entry, gfp_mask,
06                         vma, addr, &page_was_allocated);
07
08         if (page_was_allocated)
09                 swap_readpage(retpage, do_poll);
10
11         return retpage;
12 }
```

In the swap cache area, read the page corresponding to the swap entry. If it can't be found in the swap cache, it will allocate a swap cache to register it and ask the swap area to read the block device asynchronously. If the @do_poll is true, it waits for it to be read from the swap cache. (sync)

- In line 5~6 of the code, retrieve the page from the swap cache area. If it can't be found in the swap cache, prepare a new swap cache page that will be needed to read from the swap area. In this case, the page_was_allocated value contains true.
- In line 8~9 of the code, if a new swap cache page is allocated, make a bio request to read it from the swap area.
- Returns the page read from line 11 of code.

__read_swap_cache_async()

mm/swap_state.c

```
01 struct page *__read_swap_cache_async(swp_entry_t entry, gfp_t gfp_mask,
02                         struct vm_area_struct *vma, unsigned long addr,
03                         bool *new_page_allocated)
04 {
05         struct page *found_page, *new_page = NULL;
06         struct address_space *swapper_space = swap_address_space(entry);
07         int err;
08         *new_page_allocated = false;
09 
10         do {
11                 /*
12                  * First check the swap cache.  Since this is normally
13                  * called after lookup_swap_cache() failed, re-calling
14                  * that would confuse statistics.
15                  */
16                 found_page = find_get_page(swapper_space, swp_offset(ent
   ry));
17                 if (found_page)
18                         break;
19 
20                 /*
21                  * Just skip read ahead for unused swap slot.
22                  * During swap_off when swap_slot_cache is disabled,
23                  * we have to handle the race between putting
24                  * swap entry in swap cache and marking swap slot
25                  * as SWAP_HAS_CACHE.  That's done in later part of code
   or
26                  * else swap_off will be aborted if we return NULL.
27                  */
28                 if (!__swp_swapcount(entry) && swap_slot_cache_enabled)
29                         break;
30 
31                 /*
32                  * Get a new page to read into from swap.
33                  */
34                 if (!new_page) {
35                         new_page = alloc_page_vma(gfp_mask, vma, addr);
36                         if (!new_page)
37                                 break;          /* Out of memory */
38                 }
39 
40                 /*
41                  * Swap entry may have been freed since our caller obser
   ved it.
42                  */
43                 err = swapcache_prepare(entry);
44                 if (err == -EEXIST) {
```

```
45                      /*
46                       * We might race against get_swap_page() and stu
mble
47                       * across a SWAP_HAS_CACHE swap_map entry whose
page
48                       * has not been brought into the swapcache yet.
49                       */
50                      cond_resched();
51                      continue;
52              } else if (err)         /* swp entry is obsolete ? */
53                      break;
54
55              /* May fail (-ENOMEM) if XArray node allocation failed.
*/
56              __SetPageLocked(new_page);
57              __SetPageSwapBacked(new_page);
58              err = add_to_swap_cache(new_page, entry, gfp_mask & GFP_
KERNEL);
59              if (likely(!err)) {
60                      /* Initiate read into locked page */
61                      SetPageWorkingset(new_page);
62                      lru_cache_add_anon(new_page);
63                      *new_page_allocated = true;
64                      return new_page;
65              }
66              __ClearPageLocked(new_page);
67              /*
68               * add_to_swap_cache() doesn't return -EEXIST, so we can
safely
69               * clear SWAP_HAS_CACHE flag.
70               */
71              put_swap_page(new_page, entry);
72      } while (err != -ENOMEM);
73
74      if (new_page)
75              put_page(new_page);
76      return found_page;
77  }
```

Retrieve the page from the swap cache area. If it can't be found in the swap cache, it allocates a new swap cache to read from the swap area, prepares it, and returns it. If you have prepared a new swap cache, store true in the output argument @new_page_allocated.

- In lines 16~18 of code, in the swap cache area, use the offset of the swap entry to retrieve the swap cache page.
- In code lines 28~29, it is swaped off, leaving the loop to return a null page if the swap entry is no longer valid.
    - 참고: mm/swap: skip readahead only when swap slot cache is enabled (https://github.com/torvalds/linux/commit/ba81f83842549871cbd7226fc11530dc464500bb#diff-ab76e5bd92ca2482619b9e9b2954f392) (v4.11-rc1)
- In lines 34~38 of code, allocate a new swap cache page to store the page data to be read from the swap area.
- In line 43~65 of the code, first set the PG_locked and PG_swapbacked flags of the new swap cache page and add them to the swap cache. When the addition is complete, it returns a new swap cache page.
- In line 66~72 of the code, if add to swap cache fails, decrement the reference counter on the new swap cache page and repeat again unless there is not enough memory.
- In line 75 of the code, swap returns the page found in the cache.

Reads from the swap area and stores them in the swap cache

swap_readpage()

mm/page_io.c

```
01 int swap_readpage(struct page *page, bool synchronous)
02 {
03         struct bio *bio;
04         int ret = 0;
05         struct swap_info_struct *sis = page_swap_info(page);
06         blk_qc_t qc;
07         struct gendisk *disk;
08 
09         VM_BUG_ON_PAGE(!PageSwapCache(page) && !synchronous, page);
10         VM_BUG_ON_PAGE(!PageLocked(page), page);
11         VM_BUG_ON_PAGE(PageUptodate(page), page);
12         if (frontswap_load(page) == 0) {
13                 SetPageUptodate(page);
14                 unlock_page(page);
15                 goto out;
16         }
17 
18         if (sis->flags & SWP_FS) {
19                 struct file *swap_file = sis->swap_file;
20                 struct address_space *mapping = swap_file->f_mapping;
21 
22                 ret = mapping->a_ops->readpage(swap_file, page);
23                 if (!ret)
24                         count_vm_event(PSWPIN);
25                 return ret;
26         }
27 
28         ret = bdev_read_page(sis->bdev, swap_page_sector(page), page);
29         if (!ret) {
30                 if (trylock_page(page)) {
31                         swap_slot_free_notify(page);
32                         unlock_page(page);
33                 }
34 
35                 count_vm_event(PSWPIN);
36                 return 0;
37         }
38 
39         ret = 0;
40         bio = get_swap_bio(GFP_KERNEL, page, end_swap_bio_read);
41         if (bio == NULL) {
42                 unlock_page(page);
43                 ret = -ENOMEM;
44                 goto out;
45         }
46         disk = bio->bi_disk;
47         /*
48          * Keep this task valid during swap readpage because the oom kil
   ler may
49          * attempt to access it in the page fault retry time check.
50          */
51         get_task_struct(current);
52         bio->bi_private = current;
53         bio_set_op_attrs(bio, REQ_OP_READ, 0);
54         if (synchronous)
55                 bio->bi_opf |= REQ_HIPRI;
56         count_vm_event(PSWPIN);
57         bio_get(bio);
58         qc = submit_bio(bio);
```

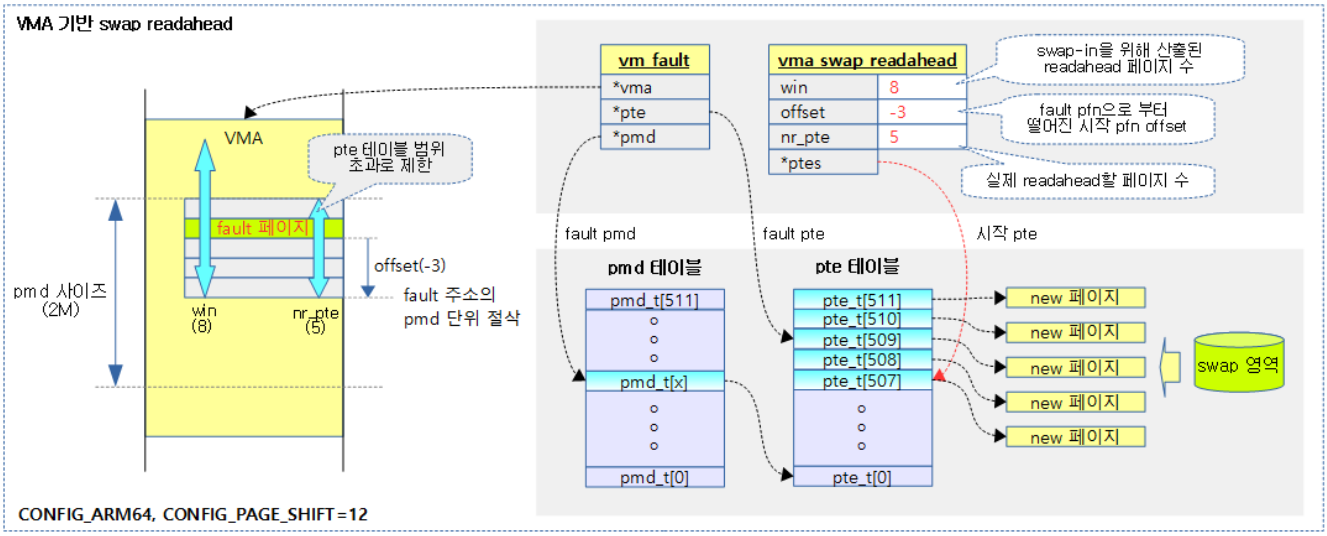
```
59          while (synchronous) {
60                  set_current_state(TASK_UNINTERRUPTIBLE);
61                  if (!READ_ONCE(bio->bi_private))
62                          break;
63
64                  if (!blk_poll(disk->queue, qc, true))
65                          io_schedule();
66          }
67          __set_current_state(TASK_RUNNING);
68          bio_put(bio);
69
70  out:
71          return ret;
72  }
```

Request BIO to store the data read from the swap area in the swap cache page. If the @synchronous is 1 at the time of the request, the motivation request will be made. The result returns 0 if successful.

- In line 5 of the code, we get the information from the swap entry information stored in the private member of the swap cache page, which is swap_info_struct information.
- In line 9~11 of the code, the PG_swapcache and PG_locked flags must be set on the page, and the PG_uptodate must be cleared.
- If the front swap is supported in line 12~16 of the code, set the PG_uptodate after the front swap load and return success.
- In line 18~26 of code, if the swap area is used through the filesystem, the page is read via the driver's (*readpage) hook function, increments the PSWPIN counter, and returns the result.
- In line 28~37 of the code, if the swap area is used by the block device, the page is read by the block device, increments the PSWPIN, and returns the result.
- In lines 40~56 of code, increment the PSWPIN counter that has prepared a request to read the swap area via bio.
- In line 57~68 of the code, make a request via bio. Then, if the @synchronous is set, wait for it to complete.
- In code lines 70~71, OUT: returns the result directly from the label.

Configuring VMA-based Swap readahead information

The following figure illustrates the VMA-based swap readahead process.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-5.png)

vma_swap_readahead Struct

include/linux/swap.h

```
01 struct vma_swap_readahead {
02         unsigned short win;
03         unsigned short offset;
04         unsigned short nr_pte;
05 #ifdef CONFIG_64BIT
06         pte_t *ptes;
07 #else
08         pte_t ptes[SWAP_RA_PTE_CACHE_SIZE];
09 #endif
10 };
```

Within the PMD size range, i.e. within a single PTE page table, the PTE

- win
  - Calculated number of pages to readahead on swapin
- offset
  - Fault pfn to readahead based on pfn offset
- nr_pte
  - Number of PTE entries to readahead in swapin (may differ from Win values by VMA and PMD unit boundaries)
- *Ptas
  - The PTE address corresponding to the fault page PTE + offset

swap_ra_info()

mm/swap_state.c

```
01 static void swap_ra_info(struct vm_fault *vmf,
02                           struct vma_swap_readahead *ra_info)
03 {
04         struct vm_area_struct *vma = vmf->vma;
05         unsigned long ra_val;
06         swp_entry_t entry;
07         unsigned long faddr, pfn, fpfn;
08         unsigned long start, end;
```

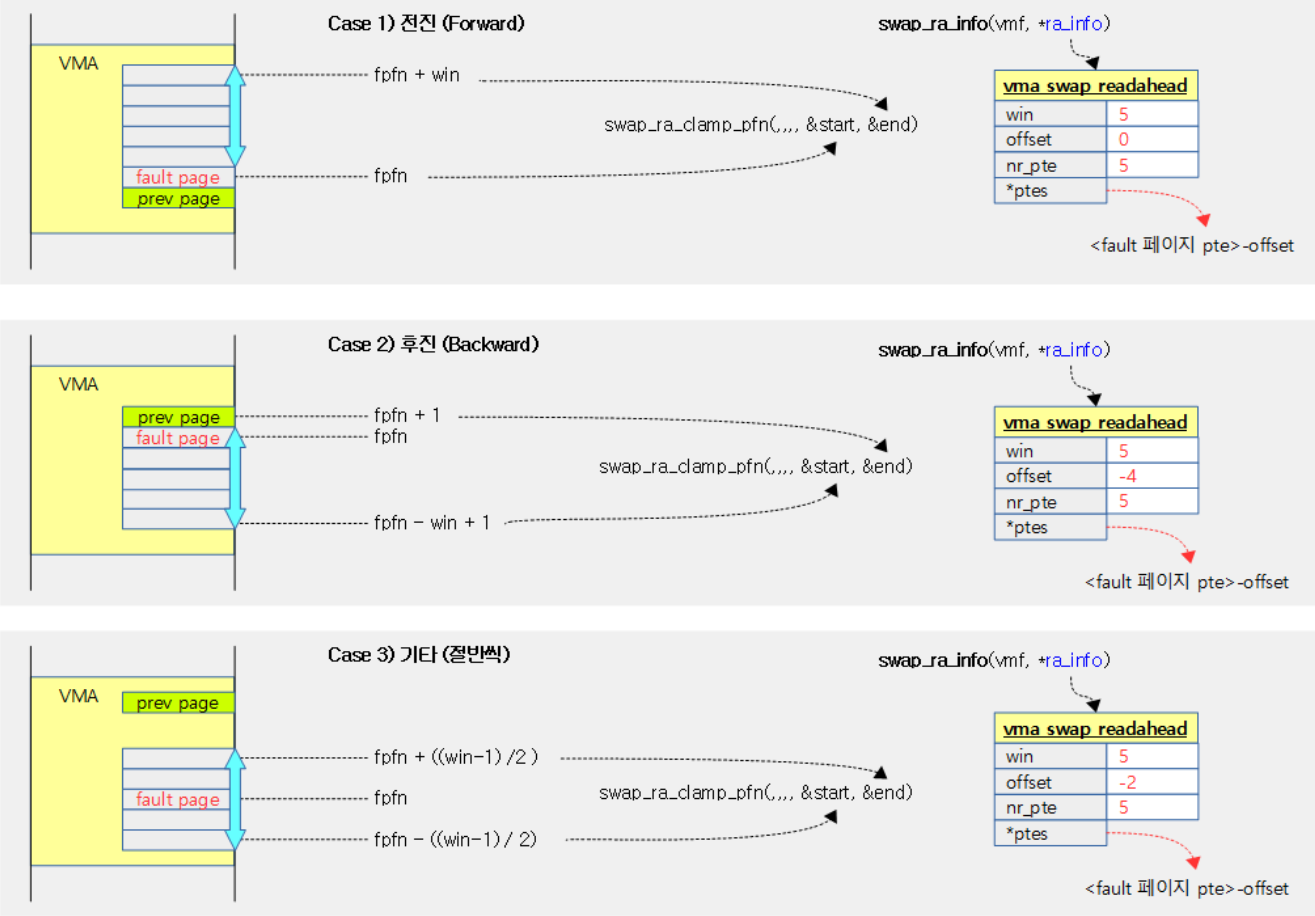
```
09          pte_t *pte, *orig_pte;
10          unsigned int max_win, hits, prev_win, win, left;
11  #ifndef CONFIG_64BIT
12          pte_t *tpte;
13  #endif
14
15          max_win = 1 << min_t(unsigned int, READ_ONCE(page_cluster),
16                               SWAP_RA_ORDER_CEILING);
17          if (max_win == 1) {
18                  ra_info->win = 1;
19                  return;
20          }
21
22          faddr = vmf->address;
23          orig_pte = pte = pte_offset_map(vmf->pmd, faddr);
24          entry = pte_to_swp_entry(*pte);
25          if ((unlikely(non_swap_entry(entry)))) {
26                  pte_unmap(orig_pte);
27                  return;
28          }
29
30          fpfn = PFN_DOWN(faddr);
31          ra_val = GET_SWAP_RA_VAL(vma);
32          pfn = PFN_DOWN(SWAP_RA_ADDR(ra_val));
33          prev_win = SWAP_RA_WIN(ra_val);
34          hits = SWAP_RA_HITS(ra_val);
35          ra_info->win = win = __swapin_nr_pages(pfn, fpfn, hits,
36                                                 max_win, prev_win);
37          atomic_long_set(&vma->swap_readahead_info,
38                          SWAP_RA_VAL(faddr, win, 0));
39
40          if (win == 1) {
41                  pte_unmap(orig_pte);
42                  return;
43          }
44
45          /* Copy the PTEs because the page table may be unmapped */
46          if (fpfn == pfn + 1)
47                  swap_ra_clamp_pfn(vma, faddr, fpfn, fpfn + win, &start,
    &end);
48          else if (pfn == fpfn + 1)
49                  swap_ra_clamp_pfn(vma, faddr, fpfn - win + 1, fpfn + 1,
50                                    &start, &end);
51          else {
52                  left = (win - 1) / 2;
53                  swap_ra_clamp_pfn(vma, faddr, fpfn - left, fpfn + win -
    left,
54                                    &start, &end);
55          }
56          ra_info->nr_pte = end - start;
57          ra_info->offset = fpfn - start;
58          pte -= ra_info->offset;
59  #ifdef CONFIG_64BIT
60          ra_info->ptes = pte;
61  #else
62          tpte = ra_info->ptes;
63          for (pfn = start; pfn != end; pfn++)
64                  *tpte++ = *pte++;
65  #endif
66          pte_unmap(orig_pte);
67  }
```

Configure the readahead information for the swap.

- In line 4 of code, we use the VMA of the VMF passed by the fault handler.

- Find the maximum number of readahead pages from lines 15~20 of the code and assign them to the max_win. When this value is 1, we don't need to compute anything more, so we substitute 1 for ra_info->win and exit the function.
    - 1 << page_cluster and SWAP_RA_ORDER_CEILING, whichever is less.
        - The initial value of page_cluster is 2~3 (16 for memory less than 2M, 3 for others), and is adjusted via the value "/proc/sys/vm/page-cluster".
        - SWAP_RA_ORDER_CEILING values are 64 on 5-bit systems and 32 on 3-bit systems.
        - In the arm64 default settings, the max_win value is 2^3=8.
- In lines 22~28 of the code, get the orig_pte value from the page table with the fault address, and use this value to get the swap entry. If it's not a swap entry, it unmaps the pte and exits the function.
- On line 30 of the code, find the pfn value that corresponds to the fault address.
- In lines 31~34 of the code, we get the ra_val values specified in the VMA, and from these values, we know the pfn, prev_win and hits values.
    - ra_val value contains three values.
        - PAGE_SHIFT Puts the pfn value in the bits that exceed the number of bits.
        - PAGE_SHIFT put the win value in the top half of the number of bits.
        - PAGE_SHIFT puts the hits value in the bottom half of the number of bits.
    - e.g. ra_val=0b111_101010_100001
        - SWAP_RA_ADDR(ra_val)=0b111_000000_000000
            - PFN_DOWN() is 0b111
        - SWAP_RA_WIN(ra_val)=0b101010
        - SWAP_RA_HITS(ra_val)=0b100001
    - If not specified, it starts with prev_win=0, hists=4.
- In lines 35~36 of the code, use the values pfn, fault pfn, hits, max_win, and prev_win to calculate the number of pages to readahead.
- In lines 37~38 of code, create and store ra_val values using faddr, win, hits=0 values for the vma->swap_readahead_info value.
- In line 40~43 of the code, if the win value is 1, we don't need to modify the number of PTEs and PTES, so we unmap the existing PTE mapping and exit the function.
- In code lines 46~55, calculate the start and end pfn with the following three conditions.
    - If a fault occurs after the last page used
    - If a fault occurs in the migration of the last page used
    - In other cases
- In line 56~65 of the code, substitute the PTE information into the output factor @ra_info.
- In line 66 of the code, the original pte is unmapped.

The following figure shows the process of calculating the number of readahead pages to be read from the swap cache based on the location of the last page and the fault page.



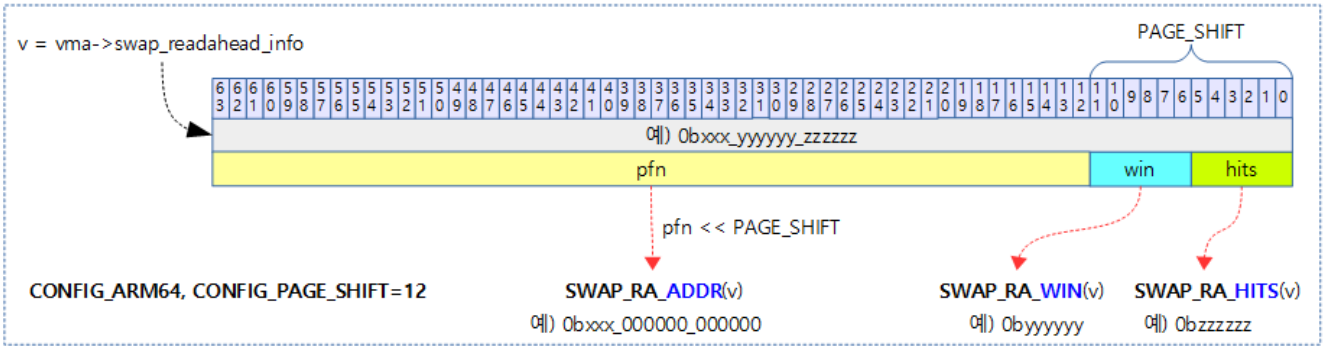
(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap_ra_info-1.png)

ra_val value-related macros

mm/swap_state.c

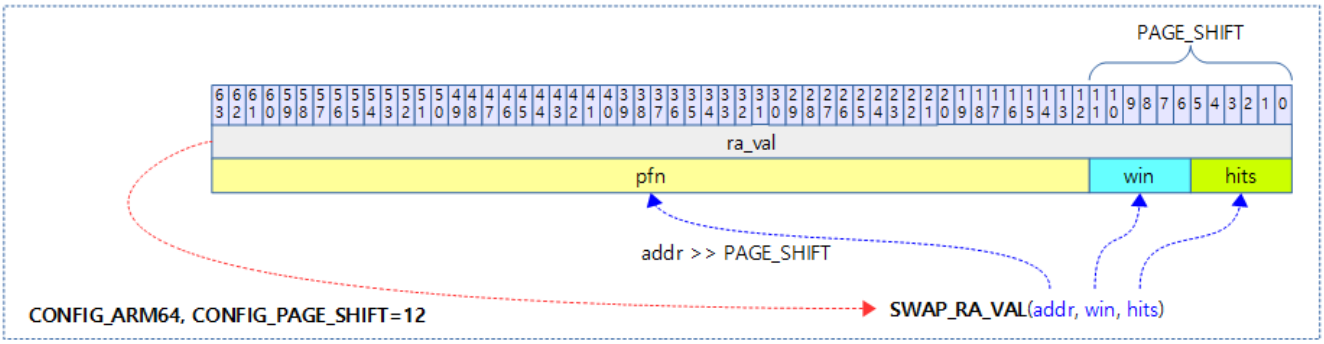
```
01 #define SWAP_RA_WIN_SHIFT       (PAGE_SHIFT / 2)
02 #define SWAP_RA_HITS_MASK       ((1UL << SWAP_RA_WIN_SHIFT) - 1)
03 #define SWAP_RA_HITS_MAX        SWAP_RA_HITS_MASK
04 #define SWAP_RA_WIN_MASK        (~PAGE_MASK & ~SWAP_RA_HITS_MASK)
05
06 #define SWAP_RA_HITS(v)         ((v) & SWAP_RA_HITS_MASK)
07 #define SWAP_RA_WIN(v)          (((v) & SWAP_RA_WIN_MASK) >> SWAP_RA_WIN
   _SHIFT)
08 #define SWAP_RA_ADDR(v)         ((v) & PAGE_MASK)
09
10 #define SWAP_RA_VAL(addr, win, hits)                            \
11         (((addr) & PAGE_MASK) |                                 \
12          (((win) << SWAP_RA_WIN_SHIFT) & SWAP_RA_WIN_MASK) |    \
13          ((hits) & SWAP_RA_HITS_MASK))
```

The following figure shows three macros that take the addr, win, and hits values from the swap_ra values.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/SWAP_RA_ADDR-1.png)

The following illustration shows the process of using the SWAP_RA_VAL() macro to create swap_ra values with the addr, win, and hits arguments.



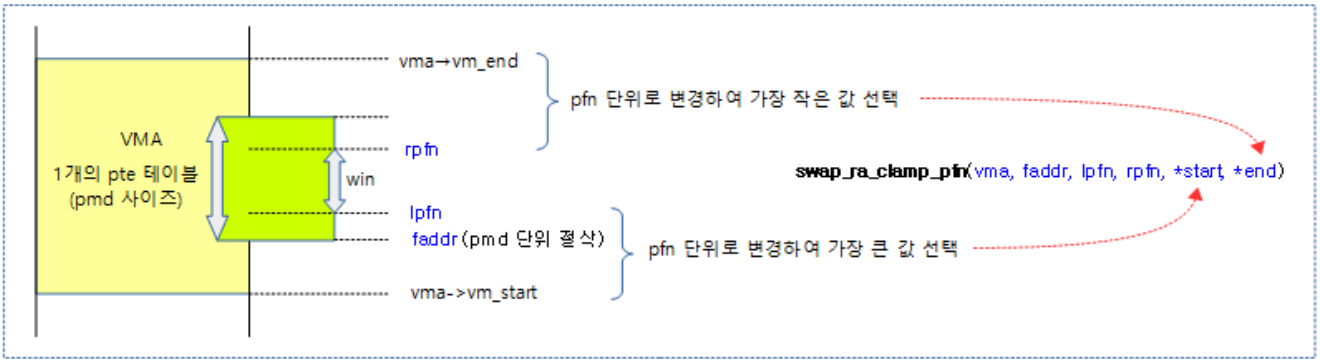
(http://jake.dothome.co.kr/wp-content/uploads/2019/10/SWAP_RA_VAL-1.png)

swap_ra_clamp_pfn()

mm/swap_state.c

```
01 static inline void swap_ra_clamp_pfn(struct vm_area_struct *vma,
02                                       unsigned long faddr,
03                                       unsigned long lpfn,
04                                       unsigned long rpfn,
05                                       unsigned long *start,
06                                       unsigned long *end)
07 {
08         *start = max3(lpfn, PFN_DOWN(vma->vm_start),
09                       PFN_DOWN(faddr & PMD_MASK));
10         *end = min3(rpfn, PFN_DOWN(vma->vm_end),
11                     PFN_DOWN((faddr & PMD_MASK) + PMD_SIZE));
12 }
```

The following diagram illustrates the process of determining the start and end addresses for the PTE entries to be imported from a single PTE page table.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap_ra_clamp_pfn-1a.png)

Swap-in with cluster-based readhead

swap_cluster_readahead()

mm/swap_state.c

```
01 /**
02  * swap_cluster_readahead - swap in pages in hope we need them soon
03  * @entry: swap entry of this memory
04  * @gfp_mask: memory allocation flags
05  * @vmf: fault information
06  *
07  * Returns the struct page for entry and addr, after queueing swapin.
08  *
09  * Primitive swap readahead code. We simply read an aligned block of
10  * (1 << page_cluster) entries in the swap area. This method is chosen
11  * because it doesn't cost us any seek time.  We also make sure to queue
12  * the 'original' request together with the readahead ones...
13  *
14  * This has been extended to use the NUMA policies from the mm triggerin
    g
15  * the readahead.
16  *
17  * Caller must hold down_read on the vma->vm_mm if vmf->vma is not NULL.
18  */
```

```
01 struct page *swap_cluster_readahead(swp_entry_t entry, gfp_t gfp_mask,
02                                     struct vm_fault *vmf)
03 {
04         struct page *page;
05         unsigned long entry_offset = swp_offset(entry);
06         unsigned long offset = entry_offset;
07         unsigned long start_offset, end_offset;
08         unsigned long mask;
09         struct swap_info_struct *si = swp_swap_info(entry);
10         struct blk_plug plug;
11         bool do_poll = true, page_allocated;
12         struct vm_area_struct *vma = vmf->vma;
13         unsigned long addr = vmf->address;
14
15         mask = swapin_nr_pages(offset) - 1;
16         if (!mask)
17                 goto skip;
18
19         do_poll = false;
20         /* Read a page_cluster sized and aligned cluster around offset.
    */
21         start_offset = offset & ~mask;
22         end_offset = offset | mask;
```

```
23          if (!start_offset)      /* First page is swap header. */
24                  start_offset++;
25          if (end_offset >= si->max)
26                  end_offset = si->max - 1;
27
28          blk_start_plug(&plug);
29          for (offset = start_offset; offset <= end_offset ; offset++) {
30                  /* Ok, do the async read-ahead now */
31                  page = __read_swap_cache_async(
32                          swp_entry(swp_type(entry), offset),
33                          gfp_mask, vma, addr, &page_allocated);
34                  if (!page)
35                          continue;
36                  if (page_allocated) {
37                          swap_readpage(page, false);
38                          if (offset != entry_offset) {
39                                  SetPageReadahead(page);
40                                  count_vm_event(SWAP_RA);
41                          }
42                  }
43                  put_page(page);
44          }
45          blk_finish_plug(&plug);
46
47          lru_add_drain();        /* Push any new pages onto the LRU now
    */
48  skip:
49          return read_swap_cache_async(entry, gfp_mask, vma, addr, do_pol
    l);
50  }
```

Perform cluster-based swap readahead on swap entries.

- In line 9 of the code, the swap entry is used to retrieve the swap information.
- In line 12 of code, we use the VMA of the VMF passed by the fault handler.
- In line 15~22 of the code, the relative offset pfn for the fault pnf is the start pfn, and use this value to get the start and end offset aligned in units of the number of pages to swap.
  - 예) offset=0x3, mask=0xf
    - start_offset=0x0
    - end_offset=0xf
  - 예) offset=0xffff_ffff_ffff_fffd(-3), mask=0xf
    - start_offset=0xffff_ffff_ffff_fff0(-16)
    - end_offset=0xffff_ffff_ffff_ffff(-1)
- In line 23~26 of code, if the starting offset is 0, it is a swap header, so increment it to use the next, and restrict the end offset to be less than the maximum.
- In line 28 of the code, initialize the blk_plug and ask the block device to hold submit until the blk_finish_plug() ends.
- Traversing the number of pte entries in code lines 29~35 and using swap entries in the swap cache area to retrieve the page.
- If the newly allocated page is from code lines 36~42, the page is read by making an asynchronous bio request from the swap area. If the assigned page is not the requested offset page, it sets the PG_reclaim flag (readahead on swap-in) and increments the SWAP_RA counter.
- In line 45 of the code, we use a function paired with the blk_start_plug() function to enable the block device to execute submit from now on.
- In line 47 of code, we revert the per-cpu LRU caches back to LRU.

- In code lines 48~49, the skip: label is. Once again, the swap cache area brings up the page in async mode.

Find the Swap Cash Page

If the fault handler records a swap entry in the fault entry value, it will call the do_swap_page() function to find the swap cache area and map it to the anon page.

lookup_swap_cache()

mm/swap_state.c

```
1  /*
2   * Lookup a swap entry in the swap cache. A found page will be returned
3   * unlocked and with its refcount incremented - we rely on the kernel
4   * lock getting page table operations atomic even if we drop the page
5   * lock before returning.
6   */
```

```
01 struct page *lookup_swap_cache(swp_entry_t entry, struct vm_area_struct
   *vma,
02                                 unsigned long addr)
03 {
04         struct page *page;
05
06         page = find_get_page(swap_address_space(entry), swp_offset(entr
   y));
07
08         INC_CACHE_INFO(find_total);
09         if (page) {
10                 bool vma_ra = swap_use_vma_readahead();
11                 bool readahead;
12
13                 INC_CACHE_INFO(find_success);
14                 /*
15                  * At the moment, we don't support PG_readahead for anon
   THP
16                  * so let's bail out rather than confusing the readahead
   stat.
17                  */
18                 if (unlikely(PageTransCompound(page)))
19                         return page;
20
21                 readahead = TestClearPageReadahead(page);
22                 if (vma && vma_ra) {
23                         unsigned long ra_val;
24                         int win, hits;
25
26                         ra_val = GET_SWAP_RA_VAL(vma);
27                         win = SWAP_RA_WIN(ra_val);
28                         hits = SWAP_RA_HITS(ra_val);
29                         if (readahead)
30                                 hits = min_t(int, hits + 1, SWAP_RA_HITS
   _MAX);
31                         atomic_long_set(&vma->swap_readahead_info,
32                                         SWAP_RA_VAL(addr, win, hits));
33                 }
34
35                 if (readahead) {
36                         count_vm_event(SWAP_RA_HIT);
```

```
37                          if (!vma || !vma_ra)
38                                  atomic_inc(&swapin_readahead_hits);
39                  }
40          }
41
42          return page;
43  }
```

Find the swap cash page with the swap entry value.

- In line 6 of the code, the page is retrieved with the address_space mapped to the swap entry and the offset value of the swap entry.
- In line 8 of code, increment the find_total counter of the swap cache stat.
- If you find a cached page in line 9~10 of the code, assign VMA-based readahead to the vam_ra whether VMA-based readahead is enabled or not.
- In line 13 of code, increment the find_success counter of the swap cache stat.
- In line 18~19 of code, there is a low probability that if it is a thp or hugetlbfs page, it will not support readahead for anon THP and will just return that page.
- In line 21 of the code, assign the value of the page's PG_reclaim (which acts as a readahed flag when swap-in) flag to readahead and clear it.
- If you are using VMA-based readahead in line 22~33 of code, update the ra_val value stored in VMA. If the page had a readahead flag, the value of hits in the ra_val is incremented.
- Increment the SWAP_RA_HIT counter if the page had the readahead flag in lines 35~39 of code. If VMA is not specified, or if VMA-based readahead is not used, increment the swapin_readahead_hits counter.

---

swap operations

address_space_operations Struct

mm/swap_state.c

```
1  /*
2   * swapper_space is a fiction, retained to simplify the path through
3   * vmscan's shrink_page_list.
4   */
```

```
1  static const struct address_space_operations swap_aops = {
2          .writepage      = swap_writepage,
3          .set_page_dirty = swap_set_page_dirty,
4  #ifdef CONFIG_MIGRATION
5          .migratepage    = migrate_page,
6  #endif
7  };
```

Write to the swap zone

Save the anon page to the swap cache, set it dirty, exit and unmap it. You can then write the swap cache via pageout() to the swap area.

swap_writepage()

mm/page_io.c

```
1  /*
2   * We may have stale swap cache pages in memory: notice
3   * them here and get rid of the unnecessary final write.
4   */
```

```
01 int swap_writepage(struct page *page, struct writeback_control *wbc)
02 {
03         int ret = 0;
04
05         if (try_to_free_swap(page)) {
06                 unlock_page(page);
07                 goto out;
08         }
09         if (frontswap_store(page) == 0) {
10                 set_page_writeback(page);
11                 unlock_page(page);
12                 end_page_writeback(page);
13                 goto out;
14         }
15         ret = __swap_writepage(page, wbc, end_swap_bio_write);
16 out:
17         return ret;
18 }
```

Write the dirty swap cache to the swap area.

- In line 5~8 of the code, remove this page from the swap cache and move it to the out: label unless:
  - It has already been removed from the swap cache. (without PG_swapcache flag)
  - The writeback is complete. (PG_writeback state)
  - It has already been saved in the swap area. (with bits set in swap_map[])
- This is the case when frontswap is supported in code lines 9~14.
- In line 15~17 of code, the page is requested to be written to the swap area and the result is returned. When sync/async recording is complete, call the end_swap_bio_write() function to change the page flag to indicate that writeback is complete.

swap_set_page_dirty()

mm/page_io.c

```
01 int swap_set_page_dirty(struct page *page)
02 {
03         struct swap_info_struct *sis = page_swap_info(page);
04
05         if (sis->flags & SWP_FS) {
06                 struct address_space *mapping = sis->swap_file->f_mappin
   g;
07
08                 VM_BUG_ON_PAGE(!PageSwapCache(page), page);
09                 return mapping->a_ops->set_page_dirty(page);
10         } else {
11                 return __set_page_dirty_no_writeback(page);
12         }
13 }
```

Dirty marks on swap pages. Returns 1 if it is newly changed to dirty.

- If the SWP_FS is set in line 5~9 of the code, use the set_page_dirty() function provided by the driver to mark it dirty.
  - SWP_FS is used by Sunrpc, NFS, XFS, BTRFS, etc.
- If you are using any other normal swap area in line 10~12 of the code, set it to dirty on the swap cache page.

__set_page_dirty_no_writeback()

mm/page-writeback.c

```
/*
 * For address_spaces which do not use buffers nor write back.
 */
```

```
int __set_page_dirty_no_writeback(struct page *page)
{
        if (!PageDirty(page))
                return !TestSetPageDirty(page);
        return 0;
}
```

Set the dirty flag on the page. Returns 1 if newly set dirty.

consultation

- Swap -1- (Basic, initialization) (http://jake.dothome.co.kr/swap-1) | Qc
- Swap -2- (Swapin & Swapout) (http://jake.dothome.co.kr/swap-2) | Question C – Current Article
- Swap -3- (allocate/unallocate swap area) (http://jake.dothome.co.kr/swap-3) | Qc
- Swap -4- (Swap Entry) (http://jake.dothome.co.kr/swap-entry) | Qc

---

LEAVE A COMMENT

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

Comments

name *

email *

Website

WRITE A COMMENT

‹ Swap -1- (Basic, initialization) (http://jake.dothome.co.kr/swap-1/)

Swap -3- (allocate/unallocate swap zones) › (http://jake.dothome.co.kr/swap-3/)

Munc Blog (2015 ~ 2024)