Swap -3- (allocate/unallocate swap zones)

2019-10-29 (http://jake.dothome.co.kr/swap-3/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Allocate/release swap entries

When a swap file or swap block device designated as swapon is designated as a swap area, a one-byte array called swap_map[] is used to manage the allocation of swap entries.

- The offset index used for swap_map[offset] refers to the offset page of the swap area.
  - swap_map[0] refers to page 0 in the swap area.
- Values used in swap_map[]
  - 0
    - Values used in the free state
  - 1~SWAP_MAP_MAX(0x3e)
    - The reference counter of the swap entry is stored as the value used in the in-use state.
    - If it is exceeded, a SWAP_MAP_CONTINUED (0x80) flag is added and a separate swap_map[] page is created to manage it.
  - SWAP_MAP_BAD (0x3f)
    - Values used as bad pages
  - SWAP_HAS_CACHE (0x40)
    - cache page (additional flags)
  - SWAP_MAP_SHMEM (0xbf)
- Swap entries are loaded into the per-cpu swap slot cache in between.

Swap Entries Allocation

get_swap_pages()

mm/swapfile.c

```
01 int get_swap_pages(int n_goal, swp_entry_t swp_entries[], int entry_siz
   e)
02 {
03         unsigned long size = swap_entry_size(entry_size);
04         struct swap_info_struct *si, *next;
05         long avail_pgs;
06         int n_ret = 0;
07         int node;
08
09         /* Only single cluster request supported */
10         WARN_ON_ONCE(n_goal > 1 && size == SWAPFILE_CLUSTER);
11
```

```
12          avail_pgs = atomic_long_read(&nr_swap_pages) / size;
13          if (avail_pgs <= 0)
14                  goto noswap;
15
16          if (n_goal > SWAP_BATCH)
17                  n_goal = SWAP_BATCH;
18
19          if (n_goal > avail_pgs)
20                  n_goal = avail_pgs;
21
22          atomic_long_sub(n_goal * size, &nr_swap_pages);
23
24          spin_lock(&swap_avail_lock);
25
26  start_over:
27          node = numa_node_id();
28          plist_for_each_entry_safe(si, next, &swap_avail_heads[node], ava
    il_lists[node]) {
29                  /* requeue si to after same-priority siblings */
30                  plist_requeue(&si->avail_lists[node], &swap_avail_heads
    [node]);
31                  spin_unlock(&swap_avail_lock);
32                  spin_lock(&si->lock);
33                  if (!si->highest_bit || !(si->flags & SWP_WRITEOK)) {
34                          spin_lock(&swap_avail_lock);
35                          if (plist_node_empty(&si->avail_lists[node])) {
36                                  spin_unlock(&si->lock);
37                                  goto nextsi;
38                          }
39                          WARN(!si->highest_bit,
40                               "swap_info %d in list but !highest_bit\n",
41                               si->type);
42                          WARN(!(si->flags & SWP_WRITEOK),
43                               "swap_info %d in list but !SWP_WRITEOK\n",
44                               si->type);
45                          __del_from_avail_list(si);
46                          spin_unlock(&si->lock);
47                          goto nextsi;
48                  }
49                  if (size == SWAPFILE_CLUSTER) {
50                          if (!(si->flags & SWP_FS))
51                                  n_ret = swap_alloc_cluster(si, swp_entri
    es);
52                  } else
53                          n_ret = scan_swap_map_slots(si, SWAP_HAS_CACHE,
54                                                      n_goal, swp_entrie
    s);
55                  spin_unlock(&si->lock);
56                  if (n_ret || size == SWAPFILE_CLUSTER)
57                          goto check_out;
58                  pr_debug("scan_swap_map of si %d failed to find offset
    \n",
59                          si->type);
60
61                  spin_lock(&swap_avail_lock);
62  nextsi:
63                  /*
64                   * if we got here, it's likely that si was almost full b
    efore,
65                   * and since scan_swap_map() can drop the si->lock, mult
    iple
66                   * callers probably all tried to get a page from the sam
    e si
67                   * and it filled up before we could get one; or, the si
    filled
68                   * up between us dropping swap_avail_lock and taking si-
    >lock.
```

```
69                      * Since we dropped the swap_avail_lock, the swap_avail_
    head
70                      * list may have been modified; so if next is still in t
    he
71                      * swap_avail_head list then try it, otherwise start ove
    r
72                      * if we have not gotten any slots.
73                      */
74                     if (plist_node_empty(&next->avail_lists[node]))
75                             goto start_over;
76             }
77
78             spin_unlock(&swap_avail_lock);
79
80     check_out:
81             if (n_ret < n_goal)
82                     atomic_long_add((long)(n_goal - n_ret) * size,
83                                     &nr_swap_pages);
84     noswap:
85             return n_ret;
86     }
```

Prepare swap entries and return their number. (THP swap entries will also be returned as 1)

- In line 3 of code, if the kernel supports THP swap, assign the @entry_size to size. If not supported, size is always 1.
  - If THP swap is supported, use HPAGE_PMD_NR as the entry size.
    - For example, if you are using a 4K page, the pmd size is 2M and HPAGE_PMD_NR=512.
- In line 10 of the code, only one can be requested per @n_goal in the cluster method.
- Assign the number of remaining swap pages from line 12~14 divided by size to avail_pgs, and if the number is less than 0, move it to the noswap label.
- In code lines 16~20, limit the @n_goal to not exceed SWAP_BATCH 64 or avail_pgs.
- On line 22 of code, subtract @n_goal * size from the remaining swap page.
- In code lines 26~28, start_over: Label. Travers each swap area registered in the global swap_avail_heads[node] priority list.
  - Swap Zone: swap_info_struct Nodes
- In line 30, add the si->avail_lists[node] nodes back to the end of the global swap_avail_heads[node] priority list.
- On lines 33~48 of code, if SI->highest_bit is not set or is a writable swap area, it will display a warning message and remove the SI->avail_lists[node] nodes from the swap_avail_heads[node[priority list and move them to the nextSI: label.
- In line 49~51 of the code, when requesting a THP swap, if the swap area does not use the file system, prepare swap entries for the THP cluster and assign them to the swp_entries.
- In line 52~54 of the code, if it is not a THP swap request, prepare as many swap entries as @n_goal and assign them to the swp_entries.
- In line 56~57 of the code, if the swap entry is ready, or if it is a THP cluster request, go to the check_out: label.
- In line 58~59 of code, if the swap entry is not ready, it prints a debug message "scan_swap_map of si %d failed to find offset\n".
- In code lines 62~76, the nextsi: label is. Continue to the next swap area.

- In code lines 80~83, check_out: Label. Processing is complete. If there are swap entries below the target (@n_goal), the remaining swap pages are updated by adding the difference.
- In code lines 84~85, noswap: is labeled. Returns the number of swap entries that are ready.

Scan swap_map to allocate 1 swap entry

scan_swap_map()

mm/swapfile.c

```
01 static unsigned long scan_swap_map(struct swap_info_struct *si,
02                                    unsigned char usage)
03 {
04         swp_entry_t entry;
05         int n_ret;
06
07         n_ret = scan_swap_map_slots(si, usage, 1, &entry);
08
09         if (n_ret)
10                 return swp_offset(entry);
11         else
12                 return 0;
13
14 }
```

Scan 1 free swap entry in the swap area and return the offset value. Returns 0 on failure.

Scan swap_map and allocate swap entries

scan_swap_map_slots()

mm/swapfile.c -1/3-

```
01 static int scan_swap_map_slots(struct swap_info_struct *si,
02                                 unsigned char usage, int nr,
03                                 swp_entry_t slots[])
04 {
05         struct swap_cluster_info *ci;
06         unsigned long offset;
07         unsigned long scan_base;
08         unsigned long last_in_cluster = 0;
09         int latency_ration = LATENCY_LIMIT;
10         int n_ret = 0;
11
12         if (nr > SWAP_BATCH)
13                 nr = SWAP_BATCH;
14
15         /*
16          * We try to cluster swap pages by allocating them sequentially
17          * in swap.  Once we've allocated SWAPFILE_CLUSTER pages this
18          * way, however, we resort to first-free allocation, starting
19          * a new cluster.  This prevents us from scattering swap pages
20          * all over the entire swap partition, so that we reduce
21          * overall disk seek times between swap pages.  -- sct
22          * But we do now try to find an empty cluster.  -Andrea
23          * And we let swap pages go all over an SSD partition.  Hugh
24          */
25
26         si->flags += SWP_SCANNING;
27         scan_base = offset = si->cluster_next;
```

```
28
29          /* SSD algorithm */
30          if (si->cluster_info) {
31                  if (scan_swap_map_try_ssd_cluster(si, &offset, &scan_bas
    e))
32                          goto checks;
33                  else
34                          goto scan;
35          }
36
37          if (unlikely(!si->cluster_nr--)) {
38                  if (si->pages - si->inuse_pages < SWAPFILE_CLUSTER) {
39                          si->cluster_nr = SWAPFILE_CLUSTER - 1;
40                          goto checks;
41                  }
42
43                  spin_unlock(&si->lock);
44
45                  /*
46                   * If seek is expensive, start searching for new cluster
    from
47                   * start of partition, to minimize the span of allocated
    swap.
48                   * If seek is cheap, that is the SWP_SOLIDSTATE si->clus
    ter_info
49                   * case, just handled by scan_swap_map_try_ssd_cluster()
    above.
50                   */
51                  scan_base = offset = si->lowest_bit;
52                  last_in_cluster = offset + SWAPFILE_CLUSTER - 1;
53
54                  /* Locate the first empty (unaligned) cluster */
55                  for (; last_in_cluster <= si->highest_bit; offset++) {
56                          if (si->swap_map[offset])
57                                  last_in_cluster = offset + SWAPFILE_CLUS
    TER;
58                          else if (offset == last_in_cluster) {
59                                  spin_lock(&si->lock);
60                                  offset -= SWAPFILE_CLUSTER - 1;
61                                  si->cluster_next = offset;
62                                  si->cluster_nr = SWAPFILE_CLUSTER - 1;
63                                  goto checks;
64                          }
65                          if (unlikely(--latency_ration < 0)) {
66                                  cond_resched();
67                                  latency_ration = LATENCY_LIMIT;
68                          }
69                  }
70
71                  offset = scan_base;
72                  spin_lock(&si->lock);
73                  si->cluster_nr = SWAPFILE_CLUSTER - 1;
74          }
```

It scans the number of free swap entries in the swap area @nr and stores them in a @slots[] array. Returns the number of free swap entries that have been scanned at this time.

- Limit the maximum number of scans to be scanned at a time in code lines 12~13 to SWAP_BATCH (64).
- In line 26 of code, add a flag to the swap area that says scanning is SWP_SCANNING until scanning is complete.
- In line 27 of code, the starting point of the scan is from the SI->cluster_next page.
- This is the case when the SSD cluster method is used in code lines 30~35.

2) Scan empty cluster (256 free swap pages) (non-SSD)

- This is the case with the non-SSD cluster approach in line 37 of code.This is done if the cluster number is already 1, which is decremented by 0.
- In code lines 38~41, if there are fewer free swap pages left in the swap area than SWAPFILE_CLUSTER(256), change the cluster number to 255 and move to the checks label.
- In code lines 43~55, find the first empty cluster with the lock in the swap area. In other words, if 256 consecutive free pages are found from lowest_bit ~ highest_bit, it will go to the checks: label.
  - Since it is non-SSD, the start of the 256 free swap pages does not have to be sorted by cluster.
- In line 56~57 of code, if the swap page corresponding to offset is already swapped and in use, increment it to the in-progress offset page += 256 to proceed with the page in the next cluster.
- If you get to the last page of the traversing cluster on lines 58~64, revert the offset back to the very first page of that cluster, specify the itinerant offset in the cluster_next, change the cluster number to 255 and go to the checks label.
- On lines 65~68 of the code, perform a preemption point every LATENCY_LIMIT (256) pages in the traversal.
- In line 71~73 of the code, the first empty cluster is not found. Offset back to the starting point, and change the cluster number to 255.

mm/swapfile.c -2/3-

```
01 checks:
02         if (si->cluster_info) {
03                 while (scan_swap_map_ssd_cluster_conflict(si, offset)) {
04                 /* take a break if we already got some slots */
05                         if (n_ret)
06                                 goto done;
07                         if (!scan_swap_map_try_ssd_cluster(si, &offset,
08                                                            &scan_base))
09                                 goto scan;
10                 }
11         }
12         if (!(si->flags & SWP_WRITEOK))
13                 goto no_page;
14         if (!si->highest_bit)
15                 goto no_page;
16         if (offset > si->highest_bit)
17                 scan_base = offset = si->lowest_bit;
18
19         ci = lock_cluster(si, offset);
20         /* reuse swap entry of cache-only swap if not busy. */
21         if (vm_swap_full() && si->swap_map[offset] == SWAP_HAS_CACHE) {
22                 int swap_was_freed;
23                 unlock_cluster(ci);
24                 spin_unlock(&si->lock);
25                 swap_was_freed = __try_to_reclaim_swap(si, offset, TTRS_
   ANYWAY);
26                 spin_lock(&si->lock);
27                 /* entry was freed successfully, try to use this again
   */
28                 if (swap_was_freed)
29                         goto checks;
30                 goto scan; /* check next one */
31         }
```

```
32
33          if (si->swap_map[offset]) {
34                  unlock_cluster(ci);
35                  if (!n_ret)
36                          goto scan;
37                  else
38                          goto done;
39          }
40          si->swap_map[offset] = usage;
41          inc_cluster_info_page(si, si->cluster_info, offset);
42          unlock_cluster(ci);
43
44          swap_range_alloc(si, offset, 1);
45          si->cluster_next = offset + 1;
46          slots[n_ret++] = swp_entry(si->type, offset);
47
48          /* got enough slots or reach max slots? */
49          if ((n_ret == nr) || (offset >= si->highest_bit))
50                  goto done;
51
52          /* search for next available slot */
53
54          /* time to take a break? */
55          if (unlikely(--latency_ration < 0)) {
56                  if (n_ret)
57                          goto done;
58                  spin_unlock(&si->lock);
59                  cond_resched();
60                  spin_lock(&si->lock);
61                  latency_ration = LATENCY_LIMIT;
62          }
63
64          /* try to get more slots in cluster */
65          if (si->cluster_info) {
66                  if (scan_swap_map_try_ssd_cluster(si, &offset, &scan_bas
    e))
67                          goto checks;
68                  else
69                          goto done;
70          }
71          /* non-ssd case */
72          ++offset;
73
74          /* non-ssd case, still more slots in cluster? */
75          if (si->cluster_nr && !si->swap_map[offset]) {
76                  --si->cluster_nr;
77                  goto checks;
78          }
79
80  done:
81          si->flags -= SWP_SCANNING;
82          return n_ret;
```

case 1) Current page (si->cluster_next) Assignment check (non-SSD)

- On line 1 of the code, the checks: label is.
- This is the case when the SSD cluster method is used in code lines 2~11.
- In lines 12~13 of code, if there is no SWP_WRITEOK flag in the swap area, it will go to the no_page label.
- In code lines 14~15, even if the largest page number (highest_bit) is not set in the swap area, it will still go to the no_page label.
- In line 16~17 of the code, if the offset page exceeds the largest page number, change the scan_base and offset back to the lowest page number, lowest_bit.
- On line 19 of code, lock and get the cluster that corresponds to offset.

- In lines 21~31 of code, if the entire swap page is more than half of the available free swap area and is set to a SWAP_HAS_CACHE value in offset, the swap cache is removed. If the removal is successful, go to the checks label, if not, go to the scan label for the next.
- In line 33~39 of the code, the swap_map[] for the offset page is already in use. If there is a n_ret value, it will go to the done label, and if not, it will go to the scan label for the next one.
- In line 40~42 of the code, the swap_map[] for the offset page is empty. Since it is in a free swap state, substitute the value of the reference counter @usage for the allocation and mark the cluster as in use. Then unlock the cluster.
- On line 44 of code, update the swap area (lowest_bit, highest_bit) by 1 additional page. If all pages in the swap area have been allocated, the swap area is removed from the swap_avail_heads.
- In code lines 45~46, do the current offset page + 1 for the next cluster. In the output argument @slots[], store the swap entry corresponding to the offset.
- If you have filled the number of slots requested in line 49~50 of the code, or if you have progressed to the end of the area, move to the done label.
- This is a way to reduce interrupt latency in code lines 55~62. In order not to lock on the swap area for a long time, the lock is released for a while every LATENCY_LIMIT (256) page and then re-acquired.
- This is the case when the SSD cluster method is used in code lines 65~70.
- This is the behavior of code lines 72~78 when it is not the SSD cluster method. Increase the offset for the next page, and decrease the cluster number if more slots are needed, and go to the checks label.
- In code lines 80~82, the done: label is. Remove the SWP_SCANNING flag set during the scanning day from the swap area and return the number of swap entries prepared in the slot.

mm/swapfile.c -3/3-

```
01 scan:
02         spin_unlock(&si->lock);
03         while (++offset <= si->highest_bit) {
04                 if (!si->swap_map[offset]) {
05                         spin_lock(&si->lock);
06                         goto checks;
07                 }
08                 if (vm_swap_full() && si->swap_map[offset] == SWAP_HAS_C
   ACHE) {
09                         spin_lock(&si->lock);
10                         goto checks;
11                 }
12                 if (unlikely(--latency_ration < 0)) {
13                         cond_resched();
14                         latency_ration = LATENCY_LIMIT;
15                 }
16         }
17         offset = si->lowest_bit;
18         while (offset < scan_base) {
19                 if (!si->swap_map[offset]) {
20                         spin_lock(&si->lock);
21                         goto checks;
22                 }
23                 if (vm_swap_full() && si->swap_map[offset] == SWAP_HAS_C
   ACHE) {
24                         spin_lock(&si->lock);
```

```
25                      goto checks;
26              }
27              if (unlikely(--latency_ration < 0)) {
28                      cond_resched();
29                      latency_ration = LATENCY_LIMIT;
30              }
31              offset++;
32      }
33      spin_lock(&si->lock);
34
35 no_page:
36      si->flags -= SWP_SCANNING;
37      return n_ret;
38 }
```

3) Scan to the end of current page ~ area (non-SSD)

- In code lines 1~16, the scan: label is. It iterates to the last free assignable page of the swap area, increments the offset, and moves to the checks label only in the following two cases: In addition, a preemption point is performed every LATENCY_LIMIT (256) pages during the circuit.
  - swap_map[offset] is assignable as free.
  - If the swap area is more than 50% full, and the swap area corresponding to the offset is free, and only the swap cache exists.

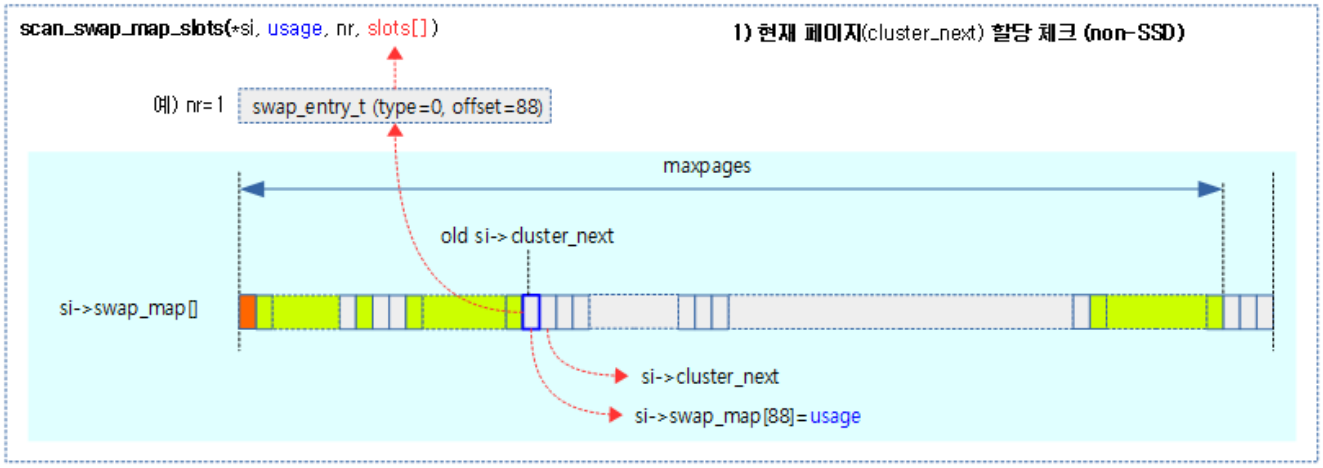
4) Scan to Start Page ~ Current Page (non-SSD)

- In lines 17~32 of the code, offset is traversed from the first page of the free assignable area to the scan_base of the swap area, and only in the following two cases go to the check label. In addition, a preemption point is performed every LATENCY_LIMIT (256) pages during the circuit.
  - swap_map[offset] is assignable as free.
  - If the swap area is more than 50% full, and the swap area corresponding to the offset is free, and only the swap cache exists.

- In code lines 35~37, no_page: Label. Remove the SWP_SCANNING flag set during the scanning day from the swap area and return the number of swap entries prepared in the slot.

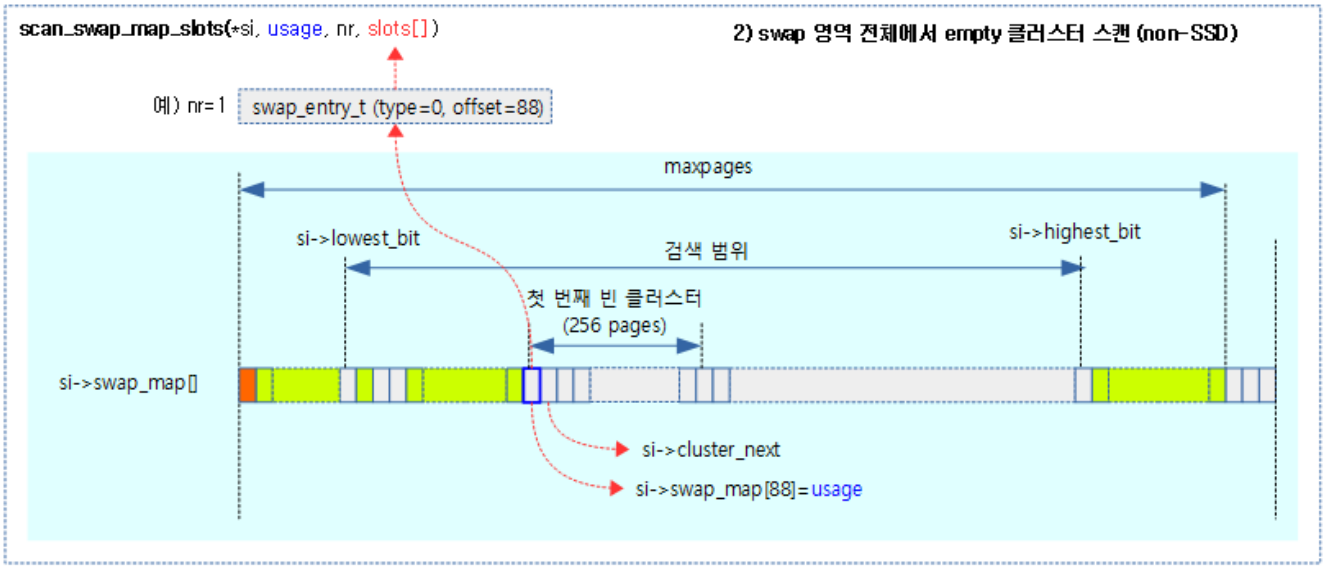
The following four figures are in order to find one free swap entry.

First, check if the si->cluster_next is assignable to the page at the current location.



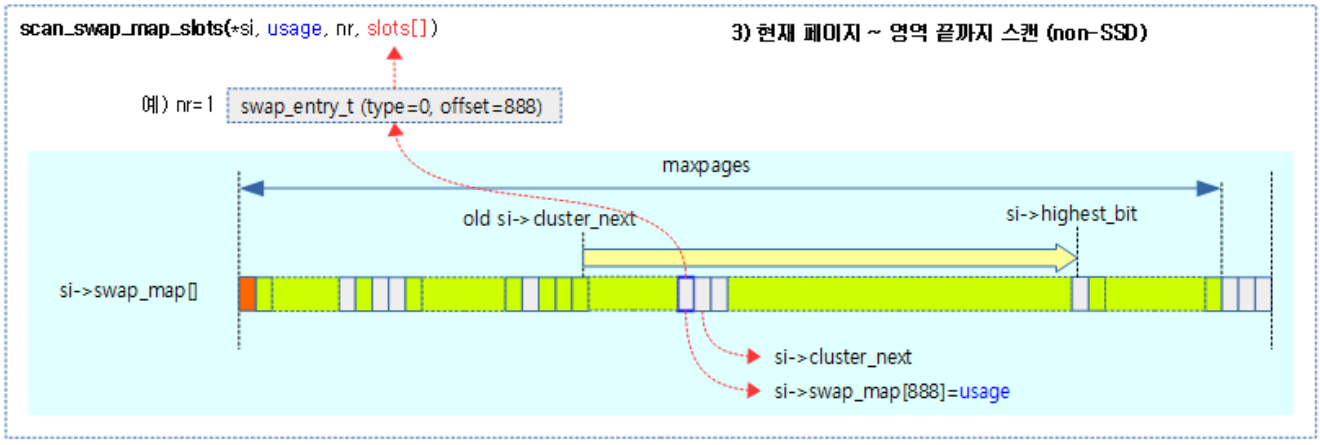
(http://jake.dothome.co.kr/wp-content/uploads/2019/10/scan_swap_map_slots-1.png)

Second, if it is not possible to assign it to the page in the current location, then check if there are 256 empty swap pages throughout the swap area.



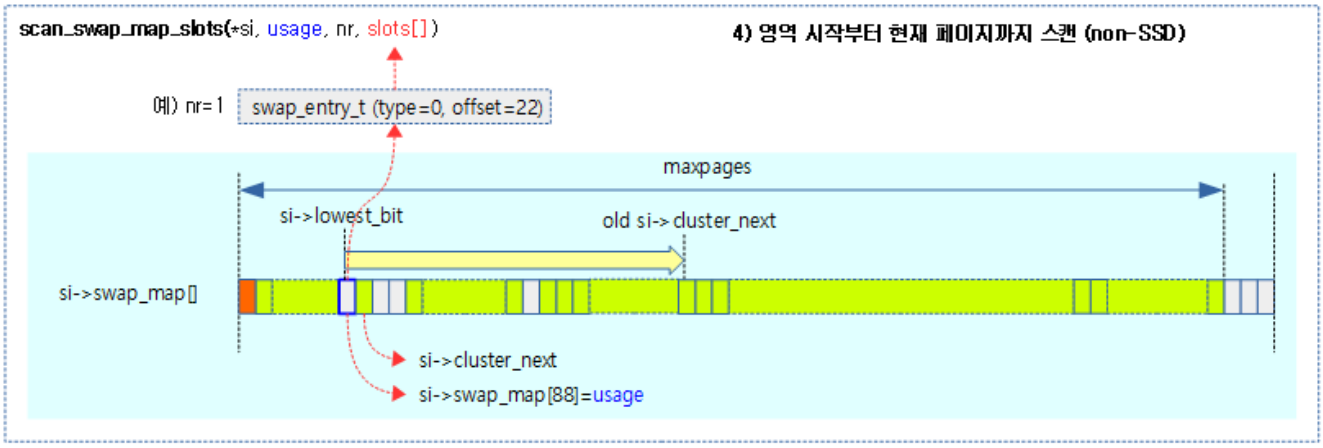
(http://jake.dothome.co.kr/wp-content/uploads/2019/10/scan_swap_map_slots-2.png)

Third, if there are no empty clusters, scan from the current position to the end of the swap zone.



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

Fourth, and finally, scan the swap area from the beginning to the current position.



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

Return swap entries to the swap area

swapcache_free_entries()

mm/swapfile.c

```
01 void swapcache_free_entries(swp_entry_t *entries, int n)
02 {
03         struct swap_info_struct *p, *prev;
04         int i;
05
06         if (n <= 0)
07                 return;
08
09         prev = NULL;
10         p = NULL;
11
12         /*
13          * Sort swap entries by swap device, so each lock is only taken
    once.
14          * nr_swapfiles isn't absolutely correct, but the overhead of so
    rt() is
15          * so low that it isn't necessary to optimize further.
16          */
17         if (nr_swapfiles > 1)
18                 sort(entries, n, sizeof(entries[0]), swp_entry_cmp, NUL
    L);
19         for (i = 0; i < n; ++i) {
20                 p = swap_info_get_cont(entries[i], prev);
21                 if (p)
22                         swap_entry_free(p, entries[i]);
23                 prev = p;
24         }
25         if (p)
26                 spin_unlock(&p->lock);
27 }
```

Change the @n swap entry @entries to free.

- In line 6~7 of the code, if there are less than 0 @n, the function exits.
- If the swap area of the swap entries requested in code lines 17~18 is mixed, it is necessary to hold the lock to access it, but to avoid this frequency, we must first sort the swap entries.

- From code lines 19~24, go through @n and make swap entries free. If the swap area changes, unlock it and lock the new swap area again.

swap_info_get_cont()

mm/swapfile.c

```
01 static struct swap_info_struct *swap_info_get_cont(swp_entry_t entry,
02                                                    struct swap_info_struct *q)
03 {
04         struct swap_info_struct *p;
05
06         p = _swap_info_get(entry);
07
08         if (p != q) {
09                 if (q != NULL)
10                         spin_unlock(&q->lock);
11                 if (p != NULL)
12                         spin_lock(&p->lock);
13         }
14         return p;
15 }
```

When the swap zone changes, the existing swap zone @q is unlocked and the new swap zone P lock is obtained.

Swap Entry Deallocation

swap_entry_free()

mm/swapfile.c

```
01 static void swap_entry_free(struct swap_info_struct *p, swp_entry_t entr
   y)
02 {
03         struct swap_cluster_info *ci;
04         unsigned long offset = swp_offset(entry);
05         unsigned char count;
06
07         ci = lock_cluster(p, offset);
08         count = p->swap_map[offset];
09         VM_BUG_ON(count != SWAP_HAS_CACHE);
10         p->swap_map[offset] = 0;
11         dec_cluster_info_page(p, p->cluster_info, offset);
12         unlock_cluster(ci);
13
14         mem_cgroup_uncharge_swap(entry, 1);
15         swap_range_free(p, offset, 1);
16 }
```

Unassign swap entries.

- In line 7 of the code, we get the lock of the cluster to which the offset page belongs.
- In line 7~10 of the code, find out the value of swap_map[] for the offset page by count, and then clear it to 0 to change it to free.
- In line 11 of the code, increment the usage counter by 1 for the cluster to which the offset page belongs.
- Unlock the cluster at line 12 of the code.

- In line 14 of code, memcg reports the recall of swap entries.
- In line 15 of the code, swap adjusts the assignable range due to the deallocating of the entry.

Adjust available zones after allocating/unallocating swap zones

swap_range_alloc()

mm/swapfile.c

```
01 static void swap_range_alloc(struct swap_info_struct *si, unsigned long
   offset,
02                                unsigned int nr_entries)
03 {
04         unsigned int end = offset + nr_entries - 1;
05
06         if (offset == si->lowest_bit)
07                 si->lowest_bit += nr_entries;
08         if (end == si->highest_bit)
09                 si->highest_bit -= nr_entries;
10         si->inuse_pages += nr_entries;
11         if (si->inuse_pages == si->pages) {
12                 si->lowest_bit = si->max;
13                 si->highest_bit = 0;
14                 del_from_avail_list(si);
15         }
16 }
```

Update the allotable range through allocation by @nr_entries from the @offset of the swap area.

- When assigning the range offset ~ nr_entries -6 in code lines 9~1, update the minimum or maximum pages as needed.
- In line 10 of code, add the SI->inuse_pages to the number of pages you allocated.
- If the swap area is used up in lines 11~15 of the code, remove it from the swap_avail_heads list.

swap_range_free()

mm/swapfile.c

```
01 static void swap_range_free(struct swap_info_struct *si, unsigned long o
   ffset,
02                               unsigned int nr_entries)
03 {
04         unsigned long end = offset + nr_entries - 1;
05         void (*swap_slot_free_notify)(struct block_device *, unsigned lo
   ng);
06
07         if (offset < si->lowest_bit)
08                 si->lowest_bit = offset;
09         if (end > si->highest_bit) {
10                 bool was_full = !si->highest_bit;
11
12                 si->highest_bit = end;
13                 if (was_full && (si->flags & SWP_WRITEOK))
14                         add_to_avail_list(si);
15         }
16         atomic_long_add(nr_entries, &nr_swap_pages);
17         si->inuse_pages -= nr_entries;
18         if (si->flags & SWP_BLKDEV)
19                 swap_slot_free_notify =
20                         si->bdev->bd_disk->fops->swap_slot_free_notify;
```

```
21          else
22                  swap_slot_free_notify = NULL;
23          while (offset <= end) {
24                  frontswap_invalidate_page(si->type, offset);
25                  if (swap_slot_free_notify)
26                          swap_slot_free_notify(si->bdev, offset);
27                  offset++;
28          }
29  }
```

From the @offset of the swap area, the allotable range is updated by @nr_entries through deallocation.

- When deallocating the range offset ~ nr_entries -4 in line 12~1 of the code, the minimum or maximum pages are updated as needed.
- Since there is a new available area in line 13~14 of the code, add this swap area to the &swap_avail_heads list.
- In line 17 of code, decrement SI->inuse_pages by the number of deallocated pages.
- In line 18~28 of code, invalidate the page in frontswap for the range offset ~ nr_entries -1. And if the swap area is using a block device, call the hook function (*swap_slot_free_notify) to notify that the swap area is free.

swap_avail_heads List

swap_avail_heads[] lists are operated by nodes, and the available swap areas are registered.

swapfile_init()

mm/swapfile.c”

```
01  static int __init swapfile_init(void)
02  {
03          int nid;
04
05          swap_avail_heads = kmalloc_array(nr_node_ids, sizeof(struct plis
    t_head),
06                                           GFP_KERNEL);
07          if (!swap_avail_heads) {
08                  pr_emerg("Not enough memory for swap heads, swap is disa
    bled\n");
09                  return -ENOMEM;
10          }
11
12          for_each_node(nid)
13                  plist_head_init(&swap_avail_heads[nid]);
14
15          return 0;
16  }
17  subsys_initcall(swapfile_init);
```

swap_avail_heads[] list is initialized after allocating the number of nodes.

- This list registers the swap area to be used in the swapon.

add_to_avail_list()

mm/swapfile.c

```
01 static void add_to_avail_list(struct swap_info_struct *p)
02 {
03         int nid;
04
05         spin_lock(&swap_avail_lock);
06         for_each_node(nid) {
07                 WARN_ON(!plist_node_empty(&p->avail_lists[nid]));
08                 plist_add(&p->avail_lists[nid], &swap_avail_heads[nid]);
09         }
10         spin_unlock(&swap_avail_lock);
11 }
```

Add the requested swap area to all nodes swap_avail_heads[].

del_from_avail_list()

mm/swapfile.c

```
1 static void del_from_avail_list(struct swap_info_struct *p)
2 {
3         spin_lock(&swap_avail_lock);
4         __del_from_avail_list(p);
5         spin_unlock(&swap_avail_lock);
6 }
```

Remove the requested swap area from the swap_avail_heads[] list.

__del_from_avail_list()

mm/swapfile.c

```
1 static void __del_from_avail_list(struct swap_info_struct *p)
2 {
3         int nid;
4
5         for_each_node(nid)
6                 plist_del(&p->avail_lists[nid], &swap_avail_heads[nid]);
7 }
```

Remove the requested swap area from the swap_avail_heads[] list of all nodes.

Swap page allocation using clusters

This is possible when using a block device with SSD and persistent memory as a swap area. If there is a shortage of memory, multiple CPUs will try to swap at the same time, and each CPU is designed to run on a cluster-by-cluster basis. In addition, in order to minimize the performance degradation caused by false cache line sharing when each CPU accesses the swap_cluster_info structure corresponding to its own CPU, it is possible to solve this problem by dropping each cluster to at least the cache line size rather than simply registering each cluster in the list of free clusters in order.

Structure

swap_cluster_info Structure

include/linux/swap.h

```
/*
 * We use this to track usage of a cluster. A cluster is a block of swap disk
 * space with SWAPFILE_CLUSTER pages long and naturally aligns in disk. All
 * free clusters are organized into a list. We fetch an entry from the list to
 * get a free cluster.
 *
 * The data field stores next cluster if the cluster is free or cluster usage
 * counter otherwise. The flags field determines if a cluster is free. This is
 * protected by swap_info_struct.lock.
 */
```

```
struct swap_cluster_info {
        spinlock_t lock;        /*
                                 * Protect swap_cluster_info fields
                                 * and swap_info_struct->swap_map
                                 * elements correspond to the swap
                                 * cluster
                                 */
        unsigned int data:24;
        unsigned int flags:8;
};
```

It is a data structure used to point to one cluster state and then to a free cluster number.

- lock
  - Cluster Lock
- data:24
  - Free Cluster Status
    - Next free cluster number
- flags:8
  - CLUSTER_FLAG_FREE (1)
    - Free Clusters
  - CLUSTER_FLAG_NEXT_NULL (2)
    - Last free cluster
  - CLUSTER_FLAG_HUGE (4)
    - Clusters for THP

percpu_cluster Struct

include/linux/swap.h

```
/*
 * We assign a cluster to each CPU, so each CPU can allocate swap entry from
 * its own cluster and swapout sequentially. The purpose is to optimize swapout
 * throughput.
 */
```

```
struct percpu_cluster {
        struct swap_cluster_info index; /* Current cluster index */
        unsigned int next; /* Likely next allocation offset */
};
```

It is a cluster data structure used for per-CPU.

- index
  - It currently contains clusters.
- next
  - There is a high probability that the next assignment page contains an offset.

swap_cluster_list Struct

include/linux/swap.h

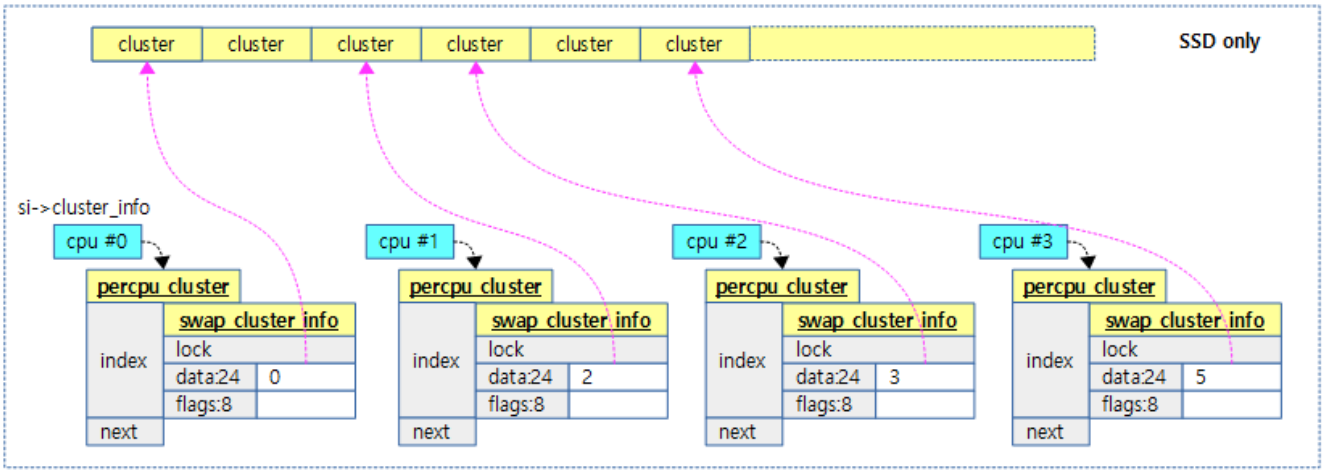
```
struct swap_cluster_list {
        struct swap_cluster_info head;
        struct swap_cluster_info tail;
};
```

It is used for the following list of swap clusters.

- SI->free_clusters List
- SI->discard_clusters List

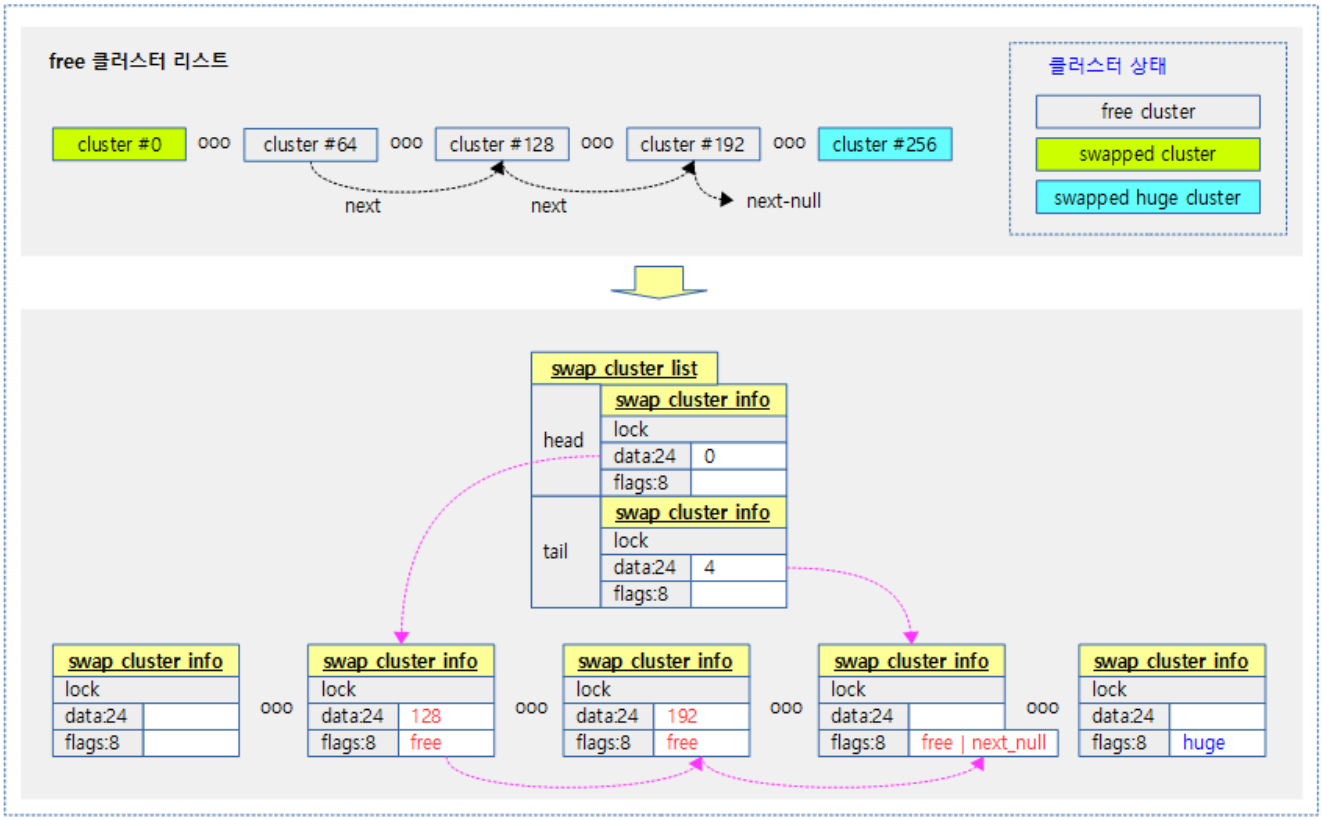
The following figure shows how each CPU uses one cluster.

- If not specified, use null for the SI->cluster value of that CPU.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-17.png)

The following figure shows a list of free swap clusters with free swap clusters connected.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-16.png)

SWAP_CLUSTER_COLS

mm/swapfile.c

```
1 #define SWAP_CLUSTER_COLS
  \
2         max_t(unsigned int, SWAP_CLUSTER_INFO_COLS, SWAP_CLUSTER_SPACE_C
  OLS)
```

When the list of free clusters is first constructed, it calculates the interval at which the swap_info_cluster structure should be skipped to reduce the false cache line share phenomenon.

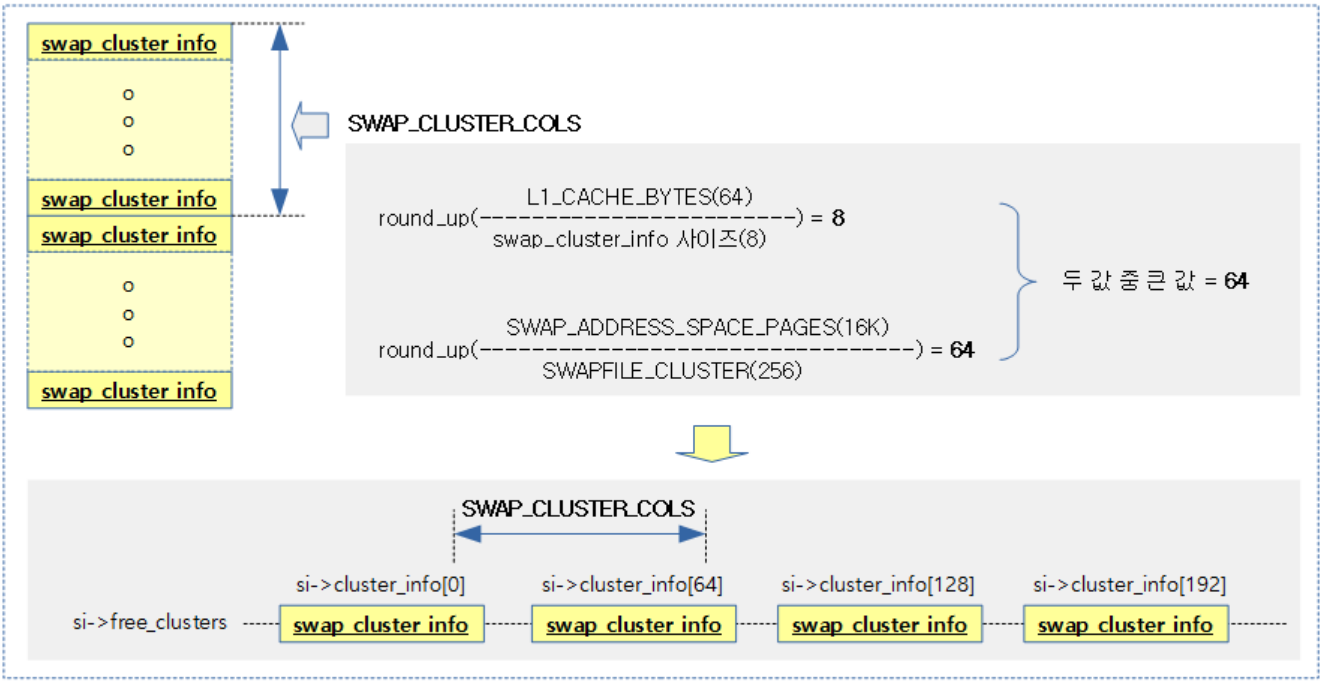
mm/swapfile.c

```
1 #define SWAP_CLUSTER_INFO_COLS
  \
2         DIV_ROUND_UP(L1_CACHE_BYTES, sizeof(struct swap_cluster_info))
3 #define SWAP_CLUSTER_SPACE_COLS
  \
4         DIV_ROUND_UP(SWAP_ADDRESS_SPACE_PAGES, SWAPFILE_CLUSTER)
```

The following shows how swap_cluster_info structs are registered at SWAP_CLUSTER_COLS numbered intervals when they are configured in the free_clusters list.

- Register all clusters in the following order:
- 0, 64, 128, 192, ...
- 1, 65, 129, 193, ...
- 2, 66, 130, 194, ...

- ...



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

Assign an entire cluster (non-SSD can be used)

swap_alloc_cluster()

mm/swapfile.c

```
01 static int swap_alloc_cluster(struct swap_info_struct *si, swp_entry_t *
   slot)
02 {
03         unsigned long idx;
04         struct swap_cluster_info *ci;
05         unsigned long offset, i;
06         unsigned char *map;
07
08         /*
09          * Should not even be attempting cluster allocations when huge
10          * page swap is disabled.  Warn and fail the allocation.
11          */
12         if (!IS_ENABLED(CONFIG_THP_SWAP)) {
13                 VM_WARN_ON_ONCE(1);
14                 return 0;
15         }
16
17         if (cluster_list_empty(&si->free_clusters))
18                 return 0;
19
20         idx = cluster_list_first(&si->free_clusters);
21         offset = idx * SWAPFILE_CLUSTER;
22         ci = lock_cluster(si, offset);
23         alloc_cluster(si, idx);
24         cluster_set_count_flag(ci, SWAPFILE_CLUSTER, CLUSTER_FLAG_HUGE);
25
26         map = si->swap_map + offset;
27         for (i = 0; i < SWAPFILE_CLUSTER; i++)
28                 map[i] = SWAP_HAS_CACHE;
29         unlock_cluster(ci);
30         swap_range_alloc(si, offset, SWAPFILE_CLUSTER);
31         *slot = swp_entry(si->type, offset);
```

```
32
33          return 1;
34  }
```

Store the swap entries to allocate the entire Free Swap cluster (page 256) to be used for THP Swap in the output argument @slot. Returns 1 if successful.

- In code lines 12~15, a kernel that does not support thp swap returns 0 as a failure.
- In code lines 17~18, if there are no free clusters left in the free cluster list, it returns 0 as a failure.
- In lines 20~24 of the code, change the cluster corresponding to the first cluster number (IDX) in the list of free clusters to a huge allocation status.
- In lines 26~28 of the code, change the assignment status for the pages in the cluster to SWAP_HAS_CACHE.
- In line 30 of the code, the cluster is replaced with an allocation state, which is updated when the lowest_bit and highest_bit range change.
- In line 31~33 of code, store the swap entry in the output argument @slot, and return success 1.

scan_swap_map_ssd_cluster_conflict()

mm/swapfile.c

```
1  /*
2   * It's possible scan_swap_map() uses a free cluster in the middle of fr
   ee
3   * cluster list. Avoiding such abuse to avoid list corruption.
4   */
```

```
01  static bool
02  scan_swap_map_ssd_cluster_conflict(struct swap_info_struct *si,
03          unsigned long offset)
04  {
05          struct percpu_cluster *percpu_cluster;
06          bool conflict;
07
08          offset /= SWAPFILE_CLUSTER;
09          conflict = !cluster_list_empty(&si->free_clusters) &&
10                  offset != cluster_list_first(&si->free_clusters) &&
11                  cluster_is_free(&si->cluster_info[offset]);
12
13          if (!conflict)
14                  return false;
15
16          percpu_cluster = this_cpu_ptr(si->percpu_cluster);
17          cluster_set_null(&percpu_cluster->index);
18          return true;
19  }
```

Verify that the requested @offset page is the first free cluster in the free cluster. If not, it is a conflict situation and returns true.

- In lines 8~11 of code, scan_swap_map() can use the free cluster in the middle of the list of available clusters. This is a way to avoid list corruption. If the requested @offset page is not the first of the free cluster, it is a conflict situation.
- If there is no conflict situation in line 13~14 of the code, it will return false normally.

- In line 16~18 of the code, if there is a conflict situation, set the percpu_cluster of the current CPU to null and return true.

SSD cluster unit swap map allocation

scan_swap_map_try_ssd_cluster()

mm/swapfile.c

```
1  /*
2   * Try to get a swap entry from current cpu's swap entry pool (a cluste
   r). This
3   * might involve allocating a new cluster for current CPU too.
4   */
```

```
01 static bool scan_swap_map_try_ssd_cluster(struct swap_info_struct *si,
02         unsigned long *offset, unsigned long *scan_base)
03 {
04         struct percpu_cluster *cluster;
05         struct swap_cluster_info *ci;
06         bool found_free;
07         unsigned long tmp, max;
08
09 new_cluster:
10         cluster = this_cpu_ptr(si->percpu_cluster);
11         if (cluster_is_null(&cluster->index)) {
12                 if (!cluster_list_empty(&si->free_clusters)) {
13                         cluster->index = si->free_clusters.head;
14                         cluster->next = cluster_next(&cluster->index) *
15                                         SWAPFILE_CLUSTER;
16                 } else if (!cluster_list_empty(&si->discard_clusters)) {
17                         /*
18                          * we don't have free cluster but have some clus
   ters in
19                          * discarding, do discard now and reclaim them
20                          */
21                         swap_do_scheduled_discard(si);
22                         *scan_base = *offset = si->cluster_next;
23                         goto new_cluster;
24                 } else
25                         return false;
26         }
27
28         found_free = false;
29
30         /*
31          * Other CPUs can use our cluster if they can't find a free clus
   ter,
32          * check if there is still free entry in the cluster
33          */
34         tmp = cluster->next;
35         max = min_t(unsigned long, si->max,
36                     (cluster_next(&cluster->index) + 1) * SWAPFILE_CLUST
   ER);
37         if (tmp >= max) {
38                 cluster_set_null(&cluster->index);
39                 goto new_cluster;
40         }
41         ci = lock_cluster(si, tmp);
42         while (tmp < max) {
43                 if (!si->swap_map[tmp]) {
44                         found_free = true;
45                         break;
46                 }
```

```
47                      tmp++;
48              }
49              unlock_cluster(ci);
50              if (!found_free) {
51                      cluster_set_null(&cluster->index);
52                      goto new_cluster;
53              }
54              cluster->next = tmp + 1;
55              *offset = tmp;
56              *scan_base = tmp;
57              return found_free;
58      }
```

It finds the free swap entry in the cluster specified in the current CPU and stores it in the output argument @offset and @scan_base. If it fails, it returns false.

- In line 9 of the code, when a new cluster needs to be relocated, it new_cluster be moved: label.
- If there is no cluster specified in the current CPU in code lines 10~26, do one of the following:
  - Prepare from the list of free clusters.
  - If there are no free clusters registered in the free cluster list, the clusters in the discard cluster list are discard to the block device and moved to the free cluster. Then go back to the new_cluster: label and try again.
  - If there are no clusters registered in the free cluster list and the discard cluster list, no clusters are available, and false is returned.
- In line 34~40 of the code, assign the next page to be found in the cluster to tmp, and if it leaves the last page of the next cluster, the current cpu will assign null to this cluster and give up using it. Then I go back to the new_cluster label to find a new cluster. Since the cluster is not locked and may have already been used by other CPUs in a competitive situation, it is abandoned for this cluster.
- In line 41~53 of the code, it traverses the tmp ~ max page while acquiring the cluster lock to which the tmp page belongs, and finds if there is a free swap page in swap_map[], and if it is found, substitute true to the found_free and exit the loop. If you don't find any free swap pages in this cluster, give up on this cluster and go back to the new_cluster: label to find a new one.
- Store the pages found in lines 54~57 in the @offset and @scan_base and return the success result of true.

Increased cluster usage

inc_cluster_info_page()

mm/swapfile.c

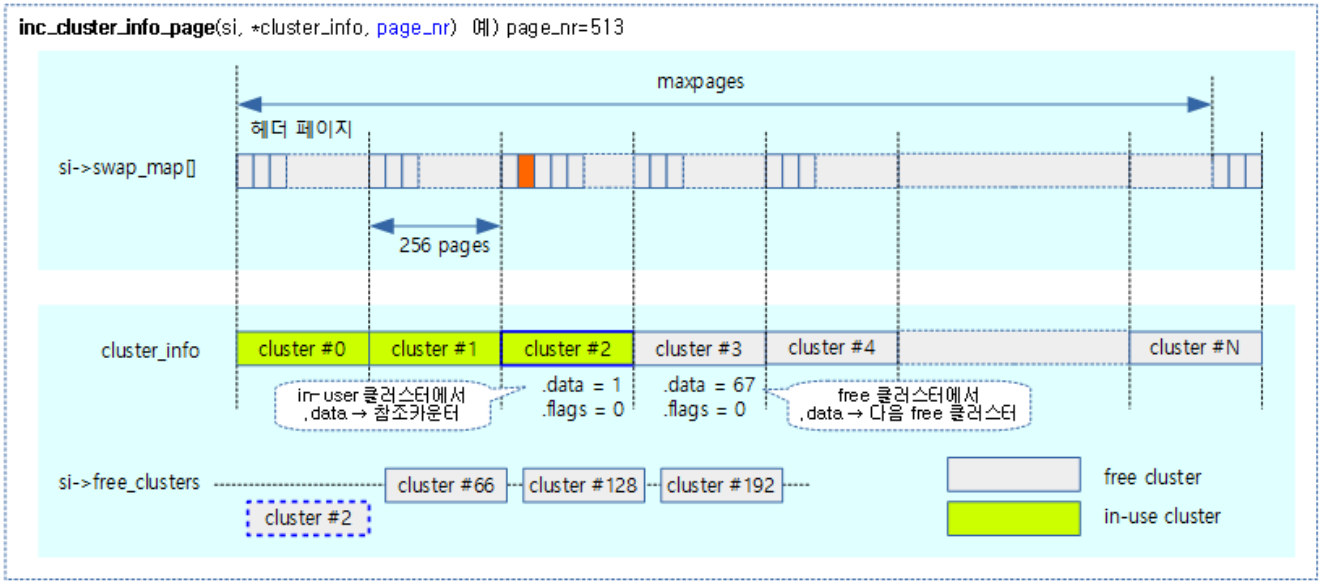
```
1       /*
2        * The cluster corresponding to page_nr will be used. The cluster will b
        e
3        * removed from free cluster list and its usage counter will be increase
        d.
4        */
```

```
01      static void inc_cluster_info_page(struct swap_info_struct *p,
02              struct swap_cluster_info *cluster_info, unsigned long page_nr)
03      {
04              unsigned long idx = page_nr / SWAPFILE_CLUSTER;
05
```

```
06          if (!cluster_info)
07                  return;
08          if (cluster_is_free(&cluster_info[idx]))
09                  alloc_cluster(p, idx);
10
11          VM_BUG_ON(cluster_count(&cluster_info[idx]) >= SWAPFILE_CLUSTE
    R);
12          cluster_set_count(&cluster_info[idx],
13                  cluster_count(&cluster_info[idx]) + 1);
14  }
```

Increment the usage counters of the cluster corresponding to @page_nr. If this is the first time it is used, change the free cluster to the allocation state.

- In line 6~7 of the code, if you don't use the SSD cluster method, you exit the function without doing anything.
- In line 8~9 of the code, if the @idx cluster is in a free state, remove it from the free cluster and change it to the allocation state.
- In line 11 of the code, if the usage counter for the @idx cluster is above 256, it is a bug.
- In lines 12~13 of the code, increment the usage counter of the @idx cluster by 1.



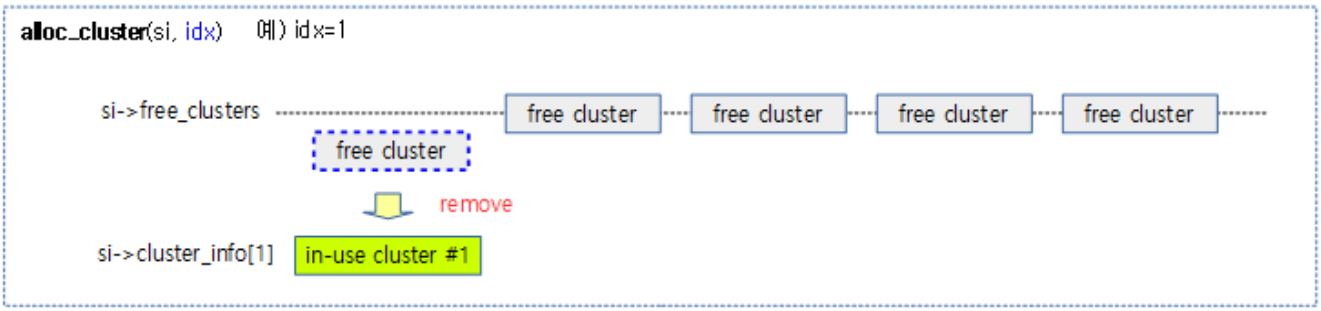
(http://jake.dothome.co.kr/wp-content/uploads/2019/10/inc_cluster_info_page-1.png)

alloc_cluster()

mm/swapfile.c

```
1   static void alloc_cluster(struct swap_info_struct *si, unsigned long id
    x)
2   {
3           struct swap_cluster_info *ci = si->cluster_info;
4
5           VM_BUG_ON(cluster_list_first(&si->free_clusters) != idx);
6           cluster_list_del_first(&si->free_clusters, ci);
7           cluster_set_count_flag(ci + idx, 0, 0);
8   }
```

Remove the @idx cluster from the free cluster, and set the usage counter to 0.



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

Reduced cluster usage

dec_cluster_info_page()

mm/swapfile.c

```
1  /*
2   * The cluster corresponding to page_nr decreases one usage. If the usag
   e
3   * counter becomes 0, which means no page in the cluster is in using, we
   can
4   * optionally discard the cluster and add it to free cluster list.
5   */
```

```
01 static void dec_cluster_info_page(struct swap_info_struct *p,
02         struct swap_cluster_info *cluster_info, unsigned long page_nr)
03 {
04         unsigned long idx = page_nr / SWAPFILE_CLUSTER;
05
06         if (!cluster_info)
07                 return;
08
09         VM_BUG_ON(cluster_count(&cluster_info[idx]) == 0);
10         cluster_set_count(&cluster_info[idx],
11                 cluster_count(&cluster_info[idx]) - 1);
12
13         if (cluster_count(&cluster_info[idx]) == 0)
14                 free_cluster(p, idx);
15 }
```

Decreases the usage counters of the clusters that correspond to the @page_nr. When the usage counter reaches 0, change it to free status and add it to the list of free clusters.

- In line 6~7 of the code, if you don't use the SSD cluster method, you exit the function without doing anything.
- In line 9 of the code, if the @idx cluster's usage counter is 0, it's a bug.
- In code lines 10~11, the usage counter of the @idx cluster is reduced by 1.
- In line 13~14 of the code, when the usage counter of the @idx cluster reaches 0, change it to free status and add it to the last of the list of free clusters.

free_cluster()

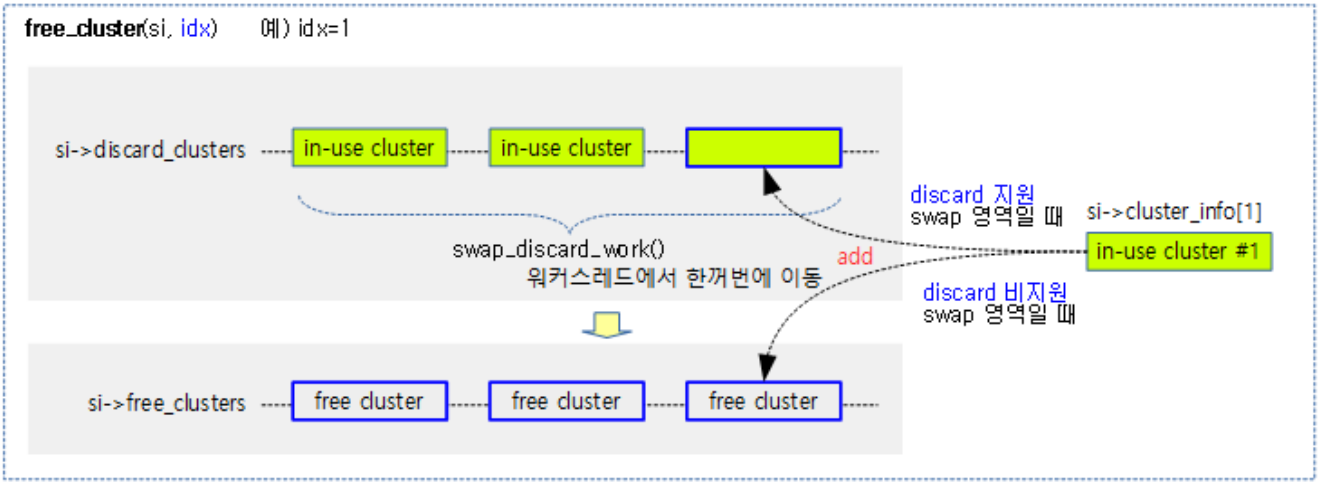
mm/swapfile.c

```
01 static void free_cluster(struct swap_info_struct *si, unsigned long idx)
02 {
```

```
03          struct swap_cluster_info *ci = si->cluster_info + idx;
04
05          VM_BUG_ON(cluster_count(ci) != 0);
06          /*
07           * If the swap is discardable, prepare discard the cluster
08           * instead of free it immediately. The cluster will be freed
09           * after discard.
10           */
11          if ((si->flags & (SWP_WRITEOK | SWP_PAGE_DISCARD)) ==
12              (SWP_WRITEOK | SWP_PAGE_DISCARD)) {
13                  swap_cluster_schedule_discard(si, idx);
14                  return;
15          }
16
17          __free_cluster(si, idx);
18  }
```

Change the @idx cluster to free and add it to the list of free clusters.

- In line 11~15 of the code, if the discard is a allowed recordable swap area, change the @idx cluster to the free state after discard via the worker thread and add it to the end of the free cluster list.
- Otherwise, in line 17 of the code, change the @idx cluster directly to free and add it to the end of the free cluster list.



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

Discard Policy

If your SSD device is mounted using the discard option or supports trim operations, you can use the swap discard feature by adding a discard option when specifying a swap area using swapon. In this case, it can also improve the performance of some SSD devices. Swapon's discard option specifies two of the following, or if you don't specify it, it uses both policies.

- –discard=once
  - Perform a discard operation only once for the entire swap area.
- –discard=pages
  - Discard asynchronously before reusing the available swap page.

__free_cluster()

mm/swapfile.c

```
1  static void __free_cluster(struct swap_info_struct *si, unsigned long id
   x)
2  {
3          struct swap_cluster_info *ci = si->cluster_info;
4
5          cluster_set_flag(ci + idx, CLUSTER_FLAG_FREE);
6          cluster_list_add_tail(&si->free_clusters, ci, idx);
7  }
```

Change the @idx cluster to free and add it to the list of free clusters.

Scheduled Discard Cluster

swap_cluster_schedule_discard()

mm/swapfile.c

```
01 /* Add a cluster to discard list and schedule it to do discard */
02 static void swap_cluster_schedule_discard(struct swap_info_struct *si,
03                 unsigned int idx)
04 {
05         /*
06          * If scan_swap_map() can't find a free cluster, it will check
07          * si->swap_map directly. To make sure the discarding cluster is
   n't
08          * taken by scan_swap_map(), mark the swap entries bad (occupie
   d). It
09          * will be cleared after discard
10          */
11         memset(si->swap_map + idx * SWAPFILE_CLUSTER,
12                         SWAP_MAP_BAD, SWAPFILE_CLUSTER);
13
14         cluster_list_add_tail(&si->discard_clusters, si->cluster_info, i
   dx);
15
16         schedule_work(&si->discard_work);
17 }
```

Run the schedule work to add the @idx cluster to the free cluster after requesting a discard.

- In lines 11~12 of the code, mark the swap_map[] corresponding to all pages of the @idx cluster as bad pages.
- In line 14 of code, register this cluster with the discard cluster.
- At line 16 of code, we run the schedule work and call the swap_discard_work() function. In this function, we request a discard from the block device, change it to a free state, and add it to the list of free clusters.

swap_discard_work()

mm/swapfile.c

```
01 static void swap_discard_work(struct work_struct *work)
02 {
03         struct swap_info_struct *si;
04
```

```
05          si = container_of(work, struct swap_info_struct, discard_work);
06
07          spin_lock(&si->lock);
08          swap_do_scheduled_discard(si);
09          spin_unlock(&si->lock);
10  }
```

The clusters in the discard cluster list are moved to the free cluster after the discard request.

swap_do_scheduled_discard()

mm/swapfile.c

```
1  /*
2   * Doing discard actually. After a cluster discard is finished, the clus
    ter
3   * will be added to free cluster list. caller should hold si->lock.
4   */
```

```
01  static void swap_do_scheduled_discard(struct swap_info_struct *si)
02  {
03          struct swap_cluster_info *info, *ci;
04          unsigned int idx;
05
06          info = si->cluster_info;
07
08          while (!cluster_list_empty(&si->discard_clusters)) {
09                  idx = cluster_list_del_first(&si->discard_clusters, inf
    o);
10                  spin_unlock(&si->lock);
11
12                  discard_swap_cluster(si, idx * SWAPFILE_CLUSTER,
13                                       SWAPFILE_CLUSTER);
14
15                  spin_lock(&si->lock);
16                  ci = lock_cluster(si, idx * SWAPFILE_CLUSTER);
17                  __free_cluster(si, idx);
18                  memset(si->swap_map + idx * SWAPFILE_CLUSTER,
19                                       0, SWAPFILE_CLUSTER);
20                  unlock_cluster(ci);
21          }
22  }
```

The clusters in the discard cluster list are moved to the free cluster after the discard request.

- In line 8~9 of the code, traverse all the clusters in the discard cluster list and remove them from the discard cluster list.
- In lines 12~13 of the code, the block device is asked to discard all the pages contained in each cluster.
- In line 15~20 of the code, change the traversing cluster with the cluster lock to free and add it to the last of the free cluster list.

discard_swap_cluster()

mm/swapfile.c

```
1  /*
2   * swap allocation tell device that a cluster of swap can now be discard
    ed,
3   * to allow the swap device to optimize its wear-levelling.
```

```
 */

static void discard_swap_cluster(struct swap_info_struct *si,
                                 pgoff_t start_page, pgoff_t nr_pages)
{
        struct swap_extent *se = si->curr_swap_extent;
        int found_extent = 0;

        while (nr_pages) {
                if (se->start_page <= start_page &&
                    start_page < se->start_page + se->nr_pages) {
                        pgoff_t offset = start_page - se->start_page;
                        sector_t start_block = se->start_block + offset;
                        sector_t nr_blocks = se->nr_pages - offset;

                        if (nr_blocks > nr_pages)
                                nr_blocks = nr_pages;
                        start_page += nr_blocks;
                        nr_pages -= nr_blocks;

                        if (!found_extent++)
                                si->curr_swap_extent = se;

                        start_block <<= PAGE_SHIFT - 9;
                        nr_blocks <<= PAGE_SHIFT - 9;
                        if (blkdev_issue_discard(si->bdev, start_block,
                                    nr_blocks, GFP_NOIO, 0))
                                break;
                }

                se = list_next_entry(se, list);
        }
}
```

Request a discard from @start_page to @nr_pages SSD-style swap devices that support discard.

Per-cpu Swap Slot Cache

Assign swap entries via the swap_map[] of the swap area. Each time, a lock in the swap area is required. To improve its performance, the per-CPU swap slot cache is used at the front end for swap entry allocation, allowing for quick allocation/unlocking of swap zones.

- 참고: mm/swap: add cache for swap slots allocation (https://github.com/torvalds/linux/commit/67afa38e012e9581b9b42f2a41dfc56b1280794d#diff-ab76e5bd92ca2482619b9e9b2954f392) (v4.11-rc1)

swap_slots_cache Struct

include/linux/swap_slots.h

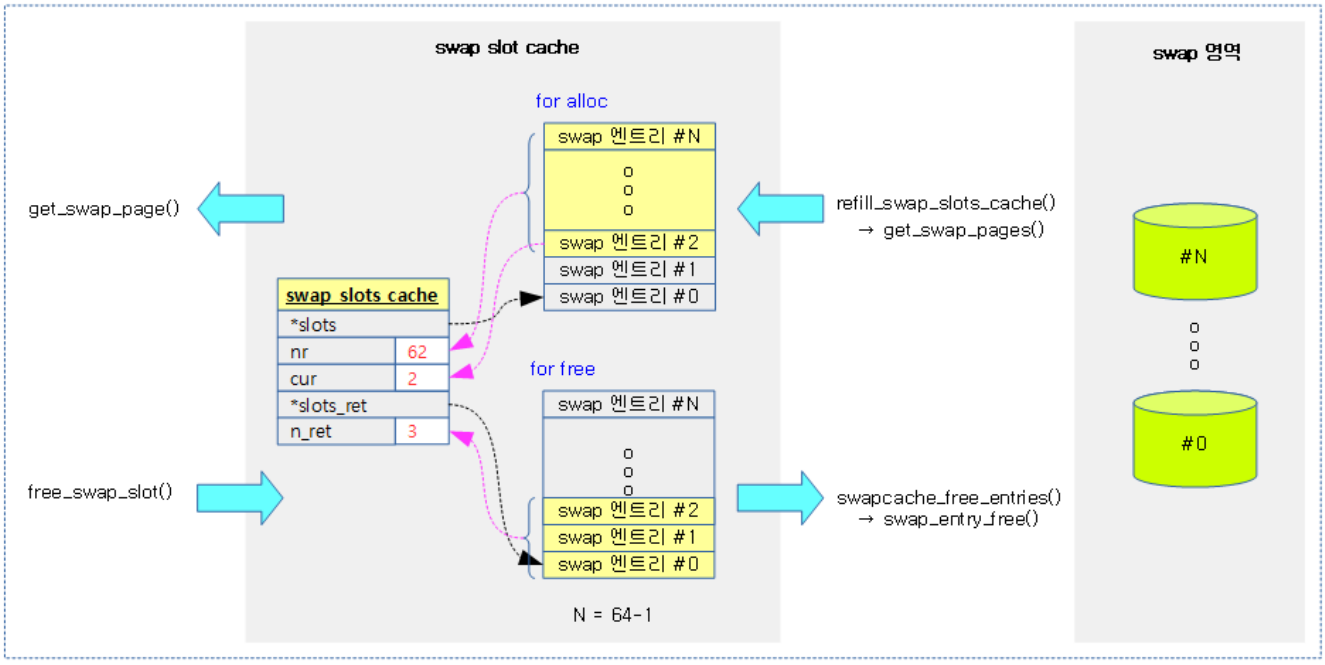
```
struct swap_slots_cache {
        bool            lock_initialized;
        struct mutex    alloc_lock; /* protects slots, nr, cur */
        swp_entry_t     *slots;
        int             nr;
        int             cur;
        spinlock_t      free_lock;  /* protects slots_ret, n_ret */
        swp_entry_t     *slots_ret;
```

```
09          int             n_ret;
10  };
```

One swap slot cache has an array of swap entries for allocation and reclaim.

- lock_initialized
  - Indicates whether the lock has been initialized.
- alloc_lock
  - swap slot cache lock for swap entry allocation
- *slots
  - Swap Entry Array for Swap Entry Allocation
- No
  - Number of swap entries for swap entry allocation
- cur
  - Current swap entry number for swap entry allocation
- free_lock
  - SWAP Slot Cash Lock for Swap Entry Retrieval
- *slots_ret
  - Swap entry array for swap entry retrieval
- n_ret
  - Number of swap entries for retrieval of swap entries

The following figure shows the use of the two swap entry arrays in the swap slot cache.



(http://jake.dothome.co.kr/wp-content/uploads/2019/10/swap-15a.png)

Swap Entry Allocation in Swap Slot Cache

get_swap_page()

mm/swap_slots.c

```
01 swp_entry_t get_swap_page(struct page *page)
02 {
03         swp_entry_t entry, *pentry;
04         struct swap_slots_cache *cache;
05
06         entry.val = 0;
07
08         if (PageTransHuge(page)) {
09                 if (IS_ENABLED(CONFIG_THP_SWAP))
10                         get_swap_pages(1, &entry, HPAGE_PMD_NR);
11                 goto out;
12         }
13
14         /*
15          * Preemption is allowed here, because we may sleep
16          * in refill_swap_slots_cache().  But it is safe, because
17          * accesses to the per-CPU data structure are protected by the
18          * mutex cache->alloc_lock.
19          *
20          * The alloc path here does not touch cache->slots_ret
21          * so cache->free_lock is not taken.
22          */
23         cache = raw_cpu_ptr(&swp_slots);
24
25         if (likely(check_cache_active() && cache->slots)) {
26                 mutex_lock(&cache->alloc_lock);
27                 if (cache->slots) {
28 repeat:
29                         if (cache->nr) {
30                                 pentry = &cache->slots[cache->cur++];
31                                 entry = *pentry;
32                                 pentry->val = 0;
33                                 cache->nr--;
34                         } else {
35                                 if (refill_swap_slots_cache(cache))
36                                         goto repeat;
37                         }
38                 }
39                 mutex_unlock(&cache->alloc_lock);
40                 if (entry.val)
41                         goto out;
42         }
43
44         get_swap_pages(1, &entry, 1);
45 out:
46         if (mem_cgroup_try_charge_swap(page, entry)) {
47                 put_swap_page(page, entry);
48                 entry.val = 0;
49         }
50         return entry;
51 }
```

Retrieve the swap entry for the request page via the swap slot cache. Returns null on failure.

- In the case of THP in line 8~12 of the code, the swap entry is secured in HPAGE_PMD_NR units only when THP swap is supported.
  - Currently, x86_64 architecture supports THP Swap.
- This is the case when it is not THP on code lines 23~42. If the swap slot cache is enabled, swap entries are obtained through the swap slot cache. If the swap slot cash is insufficient, it will be refilled.
- In line 44 of the code, if the swap slot cache is not enabled, the swap entry is obtained directly.
- In code lines 45~49, the out: label. If the memory + swap capacity exceeds the memory limit via memcg, the swap entry will be dropped.

    - memory.memsw.limit_in_bytes
- Returns a swap entry at line 50 of code.

Swap Slot Cache Top Up

refill_swap_slots_cache()

mm/swap_slots.c

```
01 /* called with swap slot cache's alloc lock held */
02 static int refill_swap_slots_cache(struct swap_slots_cache *cache)
03 {
04         if (!use_swap_slot_cache || cache->nr)
05                 return 0;
06
07         cache->cur = 0;
08         if (swap_slot_cache_active)
09                 cache->nr = get_swap_pages(SWAP_SLOTS_CACHE_SIZE,
10                                            cache->slots, 1);
11
12         return cache->nr;
13 }
```

It refills the slot cache with swap entries retrieved from the swap area and returns the number.

- In line 4~5 of the code, if the swap slot cache is not used or if swap entries already exist in the cache, it returns 0.
- If slot cache is enabled in code lines 7~10, search the swap_map and get SWAP_SLOTS_CACHE_SIZE (64) swap entries.
- Returns the number of swap entries taken from line 12 of code.

Reclaim swap entries with swap slot cache

free_swap_slot()

mm/swap_slots.c

```
01 int free_swap_slot(swp_entry_t entry)
02 {
03         struct swap_slots_cache *cache;
04
05         cache = raw_cpu_ptr(&swp_slots);
06         if (likely(use_swap_slot_cache && cache->slots_ret)) {
07                 spin_lock_irq(&cache->free_lock);
08                 /* Swap slots cache may be deactivated before acquiring
   lock */
09                 if (!use_swap_slot_cache || !cache->slots_ret) {
10                         spin_unlock_irq(&cache->free_lock);
11                         goto direct_free;
12                 }
13                 if (cache->n_ret >= SWAP_SLOTS_CACHE_SIZE) {
14                         /*
15                          * Return slots to global pool.
16                          * The current swap_map value is SWAP_HAS_CACHE.
17                          * Set it to 0 to indicate it is available for
18                          * allocation in global pool
19                          */
```

```
20                          swapcache_free_entries(cache->slots_ret, cache->
n_ret);
21                          cache->n_ret = 0;
22                  }
23                  cache->slots_ret[cache->n_ret++] = entry;
24                  spin_unlock_irq(&cache->free_lock);
25          } else {
26  direct_free:
27                  swapcache_free_entries(&entry, 1);
28          }
29
30          return 0;
31  }
```

Retrieval one swap entry via the swap slot cache. (Called from the put_swap_page() function)

- In line 5 of the code comes to know the swap slot cache for the current CPU.
- This is the case when the swap slot cache is available in code lines 6~12. With the swap slot cache locked, check once again if the swap slot cache is available. If it is not available, it will be unlocked, moved to the direct_free label, and returned directly to the swap area instead of returning to the slot cache.
- In line 13~22 of the code, if the maximum number of slots in the cache is exceeded, the swap entries in the existing swap slot cache are retrieved to each global swap area at once.
- On code lines 23~24, retrieve one swap entry from the swap slot cache.
- In code lines 25~28, direct_free: Labels. Reclaim one swap entry directly into the global swap area, not the swap slot cache.

consultation

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

Compound page › (http://jake.dothome.co.kr/compound/)

Munc Blog (2015 ~ 2024)