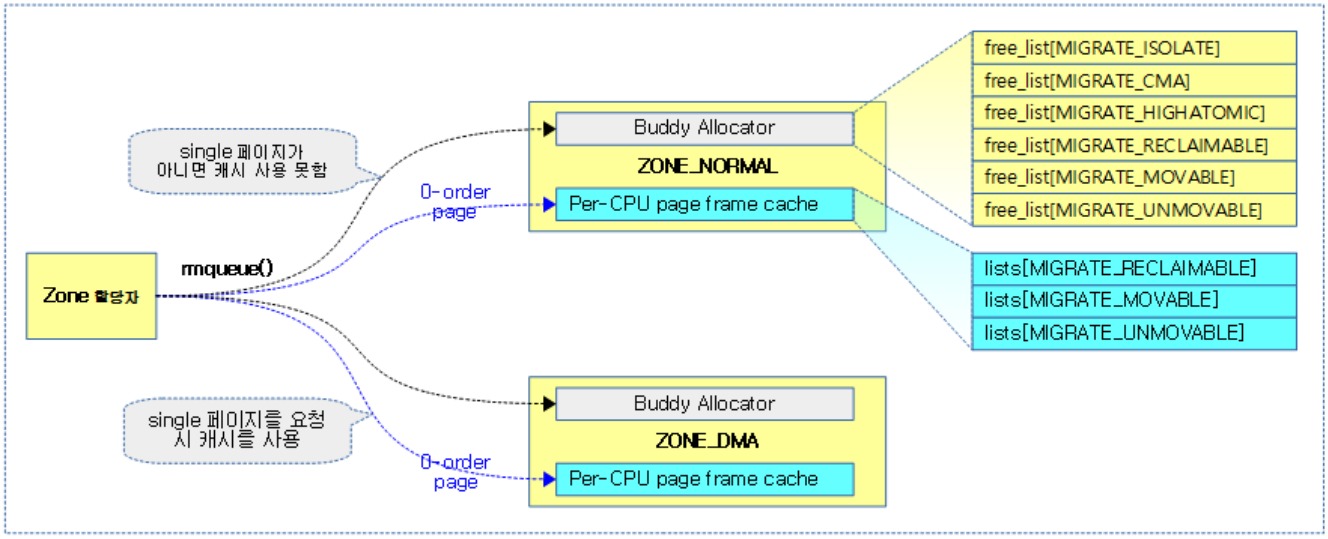
Zoned Allocator -5- (Per-CPU Page Frame Cache)

2016-04-21 (http://jake.dothome.co.kr/per-cpu-page-frame-cache/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Per-CPU Page Frame Cache



(http://jake.dothome.co.kr/wp-content/uploads/2016/04/per-cpu-page-frame-cache-1d.png)

- In the kernel, memory allocation is usually a request for a single page (0-order page) frame rather than a large page.
- In order to increase the allocation processing performance only in the case of a single page request, a per-CPU page frame cache is prepared in each zone, multiple pages are prepared in advance, and the cached page is delivered directly to the requester without using buddy for the requested single page.
  - When using Buddy, it is necessary to lock the zone, but it was implemented as lock-less using Per-CPU to improve performance.
- In the previous kernel, each zone used to use a hot cache and a cold cache, but the purpose has been consolidated into one, and the front of the prepared cache page is used for hot requests, and the back page is used for cold requests.
- Since kernel 2.6.25-rc1, each zone is managed as a cache array with a number of 3 migratetypes.
- If the kernel monitor detects that the cache page falls below the low watermark, it allocates a batch of pages to the cache in advance.
- If there are no cached pages in a single page request, batch of pages are allocated to the cache and then one of them is delivered to the requester.
- If the cached page is higher than high when a single page is released, it returns the number of batches back to the buddy system.

Allocate/unallocate/unassign order 0 pages in PCP

Order 0 Page Assignment

rmqueue_pcplist()

mm/page_alloc.c

```
01 /* Lock and remove page from the per-cpu list */
02 static struct page *rmqueue_pcplist(struct zone *preferred_zone,
03                         struct zone *zone, unsigned int order,
04                         gfp_t gfp_flags, int migratetype,
05                         unsigned int alloc_flags)
06 {
07         struct per_cpu_pages *pcp;
08         struct list_head *list;
09         struct page *page;
10         unsigned long flags;
11
12         local_irq_save(flags);
13         pcp = &this_cpu_ptr(zone->pageset)->pcp;
14         list = &pcp->lists[migratetype];
15         page = __rmqueue_pcplist(zone,  migratetype, alloc_flags, pcp, l
   ist);
16         if (page) {
17                 __count_zid_vm_events(PGALLOC, page_zonenum(page), 1 <<
   order);
18                 zone_statistics(preferred_zone, zone);
19         }
20         local_irq_restore(flags);
21         return page;
22 }
```

@migratetype assigns page 0 of order <> from pcp and returns a page descriptor.

- PCP, which is a buddy system cache implemented with Per-CPU in line 12~15 of code, allocates the order 0 page of the @migratetype from PCP with only local interrupts disabled without using a cost-intensive lock.
- In lines 16~19 of code, increment the PGALLOC counter by the number of pages.
- Returns the page assigned in line 21 of code.

__rmqueue_pcplist()

mm/page_alloc.c

```
01 /* Remove page from the per-cpu list, caller must protect the list */
02 static struct page *__rmqueue_pcplist(struct zone *zone, int migratetyp
   e,
03                         unsigned int alloc_flags,
04                         struct per_cpu_pages *pcp,
05                         struct list_head *list)
06 {
07         struct page *page;
08
09         do {
10                 if (list_empty(list)) {
11                         pcp->count += rmqueue_bulk(zone, 0,
12                                         pcp->batch, list,
13                                         migratetype, alloc_flags);
14                         if (unlikely(list_empty(list)))
```

```
15                                      return NULL;
16                              }
17
18                              page = list_first_entry(list, struct page, lru);
19                              list_del(&page->lru);
20                              pcp->count--;
21                      } while (check_new_pcp(page));
22
23                      return page;
24              }
```

@migratetype allocates order 0 pages from pcp and returns a page descriptor.

- If the @list of pcp is empty in line 9~16, migrate the number of pcp->batches from the buddy system.
- In code lines 18~20, get the first entry from the @list of PCP.
- Check that there are no problems with the entry to be assigned at line 21 of the code.
- Returns page 23 of order to be assigned in line 0 of code.

Order Recall 0 Pages

free_unref_page()

mm/page_alloc.c

```
1  /*
2   * Free a 0-order page
3   */
```

```
01  void free_unref_page(struct page *page)
02  {
03          unsigned long flags;
04          unsigned long pfn = page_to_pfn(page);
05
06          if (!free_unref_page_prepare(page, pfn))
07                  return;
08
09          local_irq_save(flags);
10          free_unref_page_commit(page, pfn);
11          local_irq_restore(flags);
12  }
```

Order 0 page back to pcp.

- Prepare the page to be free in line 6~7 of the code. If the page status is judged to be bad, the function exits.
- In line 9~11 of code, the order 0 page is returned to PCP with the local irq disabled.

free_unref_page_prepare()

mm/page_alloc.c

```
01  static bool free_unref_page_prepare(struct page *page, unsigned long pf
    n)
02  {
03          int migratetype;
04
05          if (!free_pcp_prepare(page))
```

```
06                  return false;
07
08          migratetype = get_pfnblock_migratetype(page, pfn);
09          set_pcppage_migratetype(page, migratetype);
10          return true;
11  }
```

Prepare a page to free. (normal=true, bad=false)

- Check the status of the page to be free in line 5~6 of the code, and if it is judged bad, it will return a false result.
- In lines 8~10 of the code, store the migrate type of the page block to which the page belongs to the page and return true.

free_unref_page_commit()

mm/page_alloc.c

```
01  static void free_unref_page_commit(struct page *page, unsigned long pfn)
02  {
03          struct zone *zone = page_zone(page);
04          struct per_cpu_pages *pcp;
05          int migratetype;
06
07          migratetype = get_pcppage_migratetype(page);
08          __count_vm_event(PGFREE);
09
10          /*
11           * We only track unmovable, reclaimable and movable on pcp list
    s.
12           * Free ISOLATE pages back to the allocator because they are bei
    ng
13           * offlined but treat HIGHATOMIC as movable pages so we can get
    those
14           * areas back if necessary. Otherwise, we may have to free
15           * excessively into the page allocator
16           */
17          if (migratetype >= MIGRATE_PCPTYPES) {
18                  if (unlikely(is_migrate_isolate(migratetype))) {
19                          free_one_page(zone, page, pfn, 0, migratetype);
20                          return;
21                  }
22                  migratetype = MIGRATE_MOVABLE;
23          }
24
25          pcp = &this_cpu_ptr(zone->pageset)->pcp;
26          list_add(&page->lru, &pcp->lists[migratetype]);
27          pcp->count++;
28          if (pcp->count >= pcp->high) {
29                  unsigned long batch = READ_ONCE(pcp->batch);
30                  free_pcppages_bulk(zone, batch, pcp);
31          }
32  }
```

Reclaim the 0-order page to be free to pcp.

- Increment the PGFREE counter on line 8 of code.
- In code lines 17~23, the isolate type is recalled to the buddy system, and the remaining CMA and highatomic types that are not handled by PCP are changed to movable types.
- In line 25~27 of code, add it to the pcp of type migrate.

- In line 28~31 of the code, the number of entries in the PCP list is PCP->high or higher. In order to manage only a certain amount of PCP, the number of PCP->batches is transferred to the buddy system.

get_pcppage_migratetype()

include/linux/mm.h

```
1  /*
2   * A cached value of the page's pageblock's migratetype, used when the p
   age is
3   * put on a pcplist. Used to avoid the pageblock migratetype lookup when
4   * freeing from pcplists in most cases, at the cost of possibly becoming
   stale.
5   * Also the migratetype set in the page does not necessarily match the p
   cplist
6   * index, e.g. page might have MIGRATE_CMA set but be on a pcplist with
   any
7   * other index - this ensures that it will be put on the correct CMA fre
   elist.
8   */
```

```
1  static inline int get_pcppage_migratetype(struct page *page)
2  {
3          return page->index;
4  }
```

Finds the migratetype stored in page->index.

PCP <-> Buddy System Bulk Allocation/Retrieval

PCP <-Buddy System Bulk Allocation

rmqueue_bulk()

mm/page_alloc.c

```
1  /*
2   * Obtain a specified number of elements from the buddy allocator, all u
   nder
3   * a single hold of the lock, for efficiency.  Add them to the supplied
   list.
4   * Returns the number of new pages which were placed at *list.
5   */
```

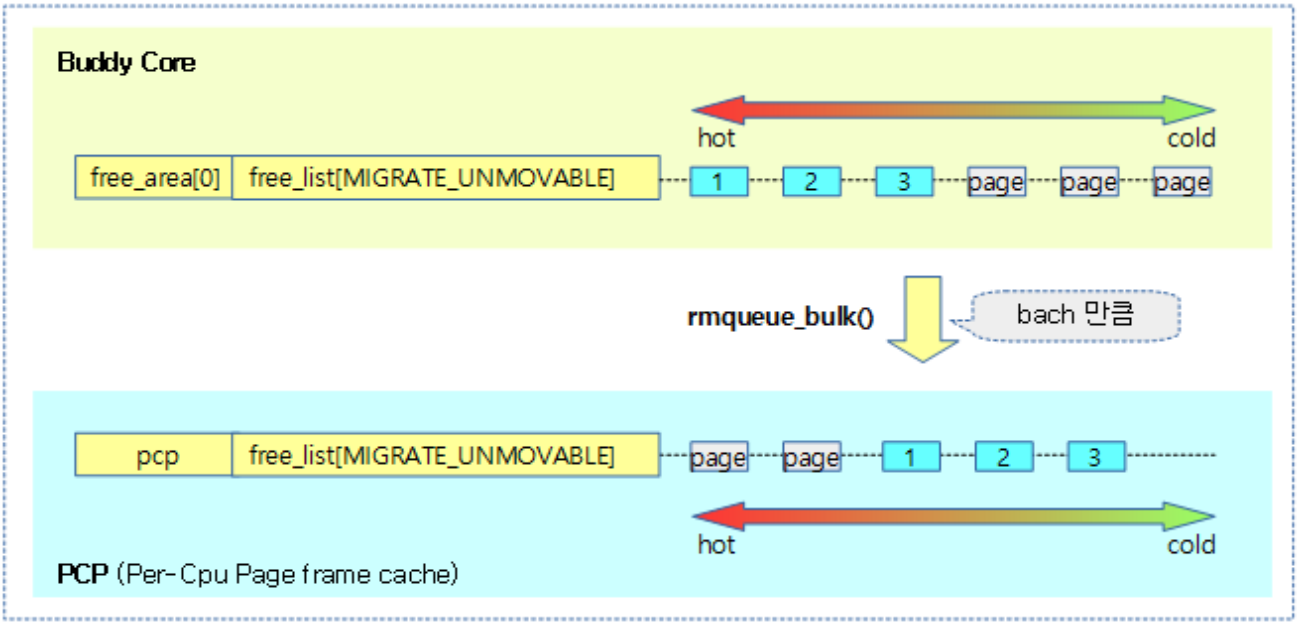
```
01 static int rmqueue_bulk(struct zone *zone, unsigned int order,
02                         unsigned long count, struct list_head *list,
03                         int migratetype, unsigned int alloc_flags)
04 {
05         int i, alloced = 0;
06
07         spin_lock(&zone->lock);
08         for (i = 0; i < count; ++i) {
09                 struct page *page = __rmqueue(zone, order, migratetype,
10                                                                 alloc_fl
   ags);
11                 if (unlikely(page == NULL))
12                         break;
13
```

```
14                  if (unlikely(check_pcp_refill(page)))
15                          continue;
16
17                  /*
18                   * Split buddy pages returned by expand() are received h
    ere in
19                   * physical page order. The page is added to the tail of
20                   * caller's list. From the callers perspective, the link
    ed list
21                   * is ordered by page number under some conditions. This
    is
22                   * useful for IO devices that can forward direction from
    the
23                   * head, thus also in the physical page order. This is u
    seful
24                   * for IO devices that can merge IO requests if the phys
    ical
25                   * pages are ordered properly.
26                   */
27                  list_add_tail(&page->lru, list);
28                  alloced++;
29                  if (is_migrate_cma(get_pcppage_migratetype(page)))
30                          __mod_zone_page_state(zone, NR_FREE_CMA_PAGES,
31                                                -(1 << order));
32          }
33
34          /*
35           * i pages were removed from the buddy list even if some leak du
    e
36           * to check_pcp_refill failing so adjust NR_FREE_PAGES based
37           * on i. Do not confuse with 'alloced' which is the number of
38           * pages added to the pcp list.
39           */
40          __mod_zone_page_state(zone, NR_FREE_PAGES, -(i << order));
41          spin_unlock(&zone->lock);
42          return alloced;
43  }
```

The Buddy system takes @count free pages from the @order slot and moves them to the @list. It then returns the actual number moved.

- Loop around the number of counts in lines 8~15 to get @order pages from the buddy system.
- Add the page from lines 27~28 to the @list.
- In code lines 29~31, if it is a cma page, decrement the NR_FREE_CMA_PAGES counter by the number of pages.
- At line 40 of code, decrement the NR_FREE_PAGES by the number of pages moved in the loop.
- Returns the number shifted in line 42 of code.

The following figure shows the bulk movement of free pages in the buddy system to PCP in batches.



(http://jake.dothome.co.kr/wp-content/uploads/2016/04/rmqueue_bulk-1b.png)

PCP -> Buddy System Bulk Recovery

free_pcppages_bulk()

```
01 /*
02  * Frees a number of pages from the PCP lists
03  * Assumes all pages on list are in same zone, and of same order.
04  * count is the number of pages to free.
05  *
06  * If the zone was previously in an "all pages pinned" state then look t
    o
07  * see if this freeing clears that state.
08  *
09  * And clear the zone's pages_scanned counter, to hold off the "all page
    s are
10  * pinned" detection logic.
11  */
```

```
01 static void free_pcppages_bulk(struct zone *zone, int count,
02                                         struct per_cpu_pages *pcp)
03 {
04         int migratetype = 0;
05         int batch_free = 0;
06         int prefetch_nr = 0;
07         bool isolated_pageblocks;
08         struct page *page, *tmp;
09         LIST_HEAD(head);
10
11         while (count) {
12                 struct list_head *list;
13
14                 /*
15                  * Remove pages from lists in a round-robin fashion. A
16                  * batch_free count is maintained that is incremented wh
    en an
17                  * empty list is encountered.  This is so more pages are
    freed
18                  * off fuller lists instead of spinning excessively arou
    nd empty
19                  * lists
20                  */
21                 do {
```

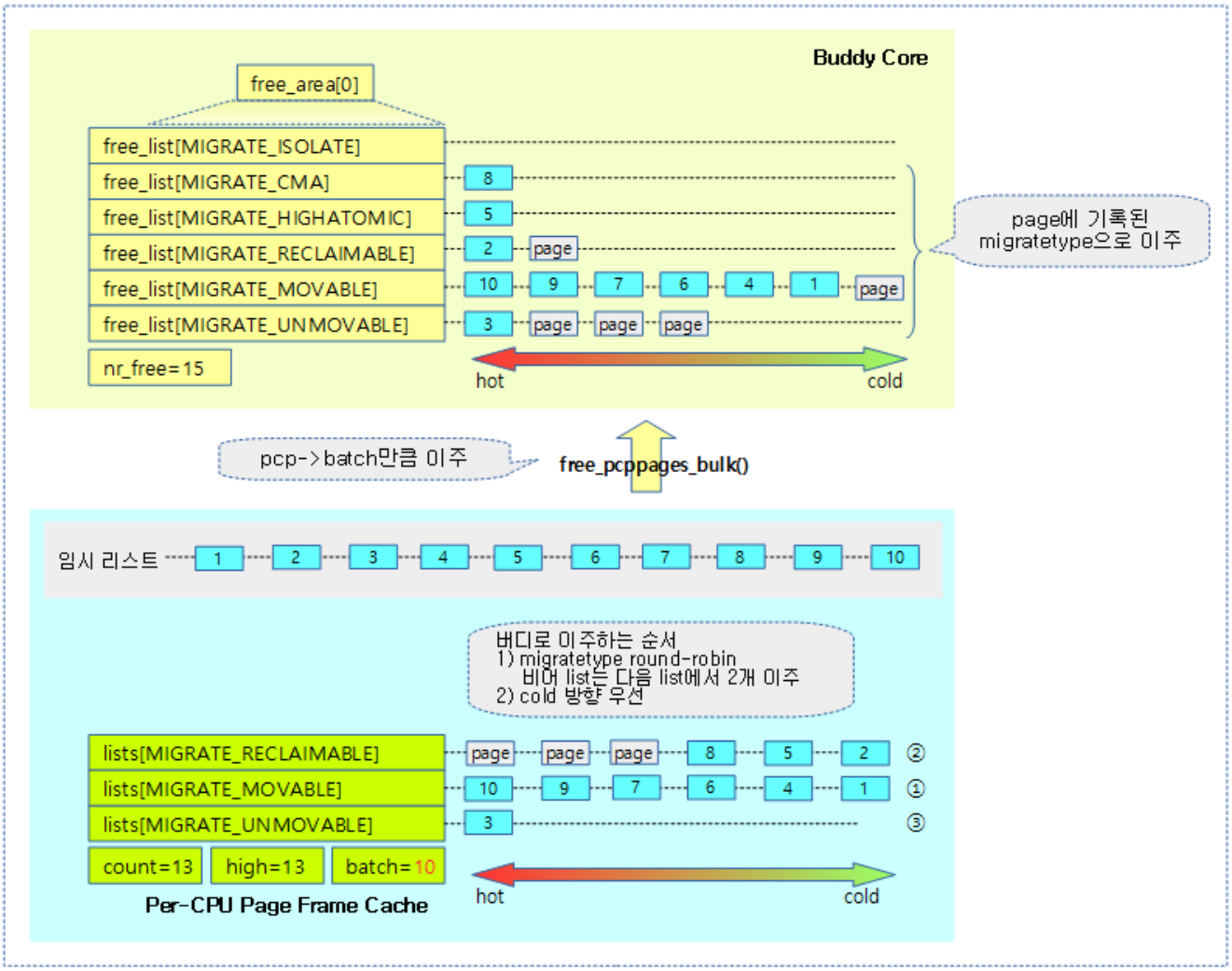
```
22                              batch_free++;
23                              if (++migratetype == MIGRATE_PCPTYPES)
24                                      migratetype = 0;
25                              list = &pcp->lists[migratetype];
26                      } while (list_empty(list));
27
28                      /* This is the only non-empty list. Free them all. */
29                      if (batch_free == MIGRATE_PCPTYPES)
30                              batch_free = count;
31
32                      do {
33                              page = list_last_entry(list, struct page, lru);
34                              /* must delete to avoid corrupting pcp list */
35                              list_del(&page->lru);
36                              pcp->count--;
37
38                              if (bulkfree_pcp_prepare(page))
39                                      continue;
40
41                              list_add_tail(&page->lru, &head);
42
43                              /*
44                               * We are going to put the page back to the glob
   al
45                               * pool, prefetch its buddy to speed up later ac
   cess
46                               * under zone->lock. It is believed the overhead
   of
47                               * an additional test and calculating buddy_pfn
   here
48                               * can be offset by reduced memory latency late
   r. To
49                               * avoid excessive prefetching due to large coun
   t, only
50                               * prefetch buddy for the first pcp->batch nr of
   pages.
51                               */
52                              if (prefetch_nr++ < pcp->batch)
53                                      prefetch_buddy(page);
54                      } while (--count && --batch_free && !list_empty(list));
55              }
56
57              spin_lock(&zone->lock);
58              isolated_pageblocks = has_isolate_pageblock(zone);
59
60              /*
61               * Use safe version since after __free_one_page(),
62               * page->lru.next will not point to original list.
63               */
64              list_for_each_entry_safe(page, tmp, &head, lru) {
65                      int mt = get_pcppage_migratetype(page);
66                      /* MIGRATE_ISOLATE page should not go to pcplists */
67                      VM_BUG_ON_PAGE(is_migrate_isolate(mt), page);
68                      /* Pageblock could have been isolated meanwhile */
69                      if (unlikely(isolated_pageblocks))
70                              mt = get_pageblock_migratetype(page);
71
72                      __free_one_page(page, page_to_pfn(page), zone, 0, mt);
73                      trace_mm_page_pcpu_drain(page, 0, mt);
74              }
75              spin_unlock(&zone->lock);
76      }
```

The PCP of the request zone is collected by the buddy system as much as it @count.

- Traverse @count number of lines of code 11.

- In lines 21~30 of the code, set the migratetype to traverse the list of pcps of 3 migrate types. However, the empty PCP list should be skipped.
    - At the beginning, proceed with movable(1), reclaimable(2), and unmovable(0) migrate types.
    - Load balance by the number of batch_free, and when the list becomes empty, the batch_free is further increased to prevent too many spins.
        - There are no empty lists, and when you work on three lists, you can rotate one by one.
        - One list is empty, and two are processed when working on the remaining two.
        - If two lists are empty, and only the last list remains, @count is substituted to process them all at once.
- In line 32~41 of code, take the tail direction entry from the pcp list of the specified migratetype and add it to the head direction of the temporary list.
- Lines 52~53 of code from pcp->batch prefetch the buddy page for the page. This is done so that the buddy system can process it with a little faster performance.
- In line 54 of the code, iterate through only batch_free of a list of pcps. However, it is completed when it is empty, or when the @count is 0 and all of them have been processed.
- Traversing the temporary list in code lines 64~74 and using the migrate type to which the page belongs, it retrieves it to the list using the corresponding migrate type of the buddy system. However, if there is an isolate type page in the zone when retrieving, use the migrate type of the page block to which the page belongs.

The figure below shows the process and sequence in which PCPs overflow and migrate to buddy by a batch.



(http://jake.dothome.co.kr/wp-content/uploads/2016/04/free_pcppages_bulk-2c.png)

- When a page is migrated to the free_list[0] slot, if a buddy page exists in free_list[0], the buddy page is removed and added by merging it into the next order, free_list[1]. Similarly, if a buddy page is found in free_list[1], it will be merged in the next order, until no buddy page is found.

The following illustration shows the order of pages that are transferred from PCP to Buddy.

(http://jake.dothome.co.kr/wp-content/uploads/2016/04/free_pcppages_bulk-1b.png)

You can check the counter information for PageSets by Zone as follows.

```
01 pi@pi /proc $ cat zoneinfo
02 Node 0, zone   Normal
03   pages free     190861
04         min      2048
```

```
        low      2560
        high     3072
        scanned  0
        spanned  241664
        present  241664
        managed  233403
    nr_free_pages 190861
(...생략...)
    nr_free_cma  935
        protection: (0, 0)
  pagesets
    cpu: 0
              count: 50
              high:  186
              batch: 31
  vm stats threshold: 24
    cpu: 1
              count: 106
              high:  186
              batch: 31
  vm stats threshold: 24
    cpu: 2
              count: 153
              high:  186
              batch: 31
  vm stats threshold: 24
    cpu: 3
              count: 156
              high:  186
              batch: 31
  vm stats threshold: 24
  all_unreclaimable: 0
  start_pfn:         0
  inactive_ratio:    1
```

PCP(Per-Cpu Page frame cache) Drain

drain_all_pages()

```
/*
 * Spill all the per-cpu pages from all CPUs back into the buddy allocator.
 *
 * When zone parameter is non-NULL, spill just the single zone's pages.
 *
 * Note that this can be extremely slow as the draining happens in a workqueue.
 */
```

```
void drain_all_pages(struct zone *zone)
{
        int cpu;

        /*
         * Allocate in the BSS so we wont require allocation in
         * direct reclaim path for CONFIG_CPUMASK_OFFSTACK=y
         */
        static cpumask_t cpus_with_pcps;

        /*
         * Make sure nobody triggers this path before mm_percpu_wq is fully
         * initialized.
         */
```

```
15          if (WARN_ONCE(!mm_percpu_wq))
16                  return;
17
18          /*
19           * Do not drain if one is already in progress unless it's specif
    ic to
20           * a zone. Such callers are primarily CMA and memory hotplug and
    need
21           * the drain to be complete when the call returns.
22           */
23          if (unlikely(!mutex_trylock(&pcpu_drain_mutex))) {
24                  if (!zone)
25                          return;
26                  mutex_lock(&pcpu_drain_mutex);
27          }
28
29          /*
30           * We don't care about racing with CPU hotplug event
31           * as offline notification will cause the notified
32           * cpu to drain that CPU pcps and on_each_cpu_mask
33           * disables preemption as part of its processing
34           */
35          for_each_online_cpu(cpu) {
36                  struct per_cpu_pageset *pcp;
37                  struct zone *z;
38                  bool has_pcps = false;
39
40                  if (zone) {
41                          pcp = per_cpu_ptr(zone->pageset, cpu);
42                          if (pcp->pcp.count)
43                                  has_pcps = true;
44                  } else {
45                          for_each_populated_zone(z) {
46                                  pcp = per_cpu_ptr(z->pageset, cpu);
47                                  if (pcp->pcp.count) {
48                                          has_pcps = true;
49                                          break;
50                                  }
51                          }
52                  }
53
54                  if (has_pcps)
55                          cpumask_set_cpu(cpu, &cpus_with_pcps);
56                  else
57                          cpumask_clear_cpu(cpu, &cpus_with_pcps);
58          }
59
60          for_each_cpu(cpu, &cpus_with_pcps) {
61                  struct pcpu_drain *drain = per_cpu_ptr(&pcpu_drain, cp
    u);
62
63                  drain->zone = zone;
64                  INIT_WORK(&drain->work, drain_local_pages_wq);
65                  queue_work_on(cpu, mm_percpu_wq, &drain->work);
66          }
67          for_each_cpu(cpu, &cpus_with_pcps)
68                  flush_work(&per_cpu_ptr(&pcpu_drain, cpu)->work);
69
70          mutex_unlock(&pcpu_drain_mutex);
71  }
```

Move the Per-CPU Page Frame Cache on all online CPUs in the specified zone to the buddy memory allocator. If no zone is specified, this is done for all populated zones.

drain_local_pages()

mm/page_alloc.c

```
/*
 * Spill all of this CPU's per-cpu pages back into the buddy allocator.
 *
 * The CPU has to be pinned. When zone parameter is non-NULL, spill just
 * the single zone's pages.
 */
```

```
void drain_local_pages(struct zone *zone)
{
        int cpu = smp_processor_id();

        if (zone)
                drain_pages_zone(cpu, zone);
        else
                drain_pages(cpu);
}
```

drain_pages()

mm/page_alloc.c

```
/*
 * Drain pcplists of all zones on the indicated processor.
 *
 * The processor must either be the current processor and the
 * thread pinned to the current processor or a processor that
 * is not online.
 */
```

```
static void drain_pages(unsigned int cpu)
{
        struct zone *zone;

        for_each_populated_zone(zone) {
                drain_pages_zone(cpu, zone);
        }
}
```

Empty the Per-CPU Page Fram Cache for all enabled zones.

drain_pages_zone()

mm/page_alloc.c

```
/*
 * Drain pcplists of the indicated processor and zone.
 *
 * The processor must either be the current processor and the
 * thread pinned to the current processor or a processor that
 * is not online.
 */
```

```
static void drain_pages_zone(unsigned int cpu, struct zone *zone)
{
        unsigned long flags;
        struct per_cpu_pageset *pset;
        struct per_cpu_pages *pcp;

        local_irq_save(flags);
        pset = per_cpu_ptr(zone->pageset, cpu);
```

```
09
10          pcp = &pset->pcp;
11          if (pcp->count)
12                  free_pcppages_bulk(zone, pcp->count, pcp);
13          local_irq_restore(flags);
14  }
```

All pages registered in the Per-CPU Page Fram Cache for the request zone are migrated to the buddy system.

consultation

- Zoned Allocator -1- (Physics Page Assignment - Fastpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath) | Qc
- Zoned Allocator -2- (Physics Page Assignment - Slowpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-slowpath) | Qc
- Zoned Allocator -3- (Buddy Page Allocation) (http://jake.dothome.co.kr/buddy-alloc) | Qc
- Zoned Allocator -4- (Buddy Page Terminated) (http://jake.dothome.co.kr/buddy-free/) | Qc
- Zoned Allocator -5- (Per-CPU Page Frame Cache) (http://jake.dothome.co.kr/per-cpu-page-frame-cache) | 문c – 현재 글
- Zoned Allocator -6- (Watermark) (http://jake.dothome.co.kr/zonned-allocator-watermark) | 문c
- Zoned Allocator -7- (Direct Compact) (http://jake.dothome.co.kr/zonned-allocator-compaction) | 문c
- Zoned Allocator -8- (Direct Compact-Isolation) (http://jake.dothome.co.kr/zonned-allocator-isolation) | 문c
- Zoned Allocator -9- (Direct Compact-Migration) (http://jake.dothome.co.kr/zonned-allocator-migration) | 문c
- Zoned Allocator -10- (LRU & pagevec) (http://jake.dothome.co.kr/lru-lists-pagevecs) | 문c
- Zoned Allocator -11- (Direct Reclaim) (http://jake.dothome.co.kr/zonned-allocator-reclaim) | 문c
- Zoned Allocator -12- (Direct Reclaim-Shrink-1) (http://jake.dothome.co.kr/zonned-allocator-shrink-1) | 문c
- Zoned Allocator -13- (Direct Reclaim-Shrink-2) (http://jake.dothome.co.kr/zonned-allocator-shrink-2) | 문c
- Zoned Allocator -14- (Kswapd) (http://jake.dothome.co.kr/zonned-allocator-kswapd) | 문c

- setup_per_cpu_pageset() (http://jake.dothome.co.kr/setup_per_cpu_pageset()) | Qc
- page_alloc_init() (http://jake.dothome.co.kr/page_alloc_init) | Qc
- CPU Bitmap (API) (http://jake.dothome.co.kr/cpu-api) | Qc

4 thoughts to “Zoned Allocator -5- (Per-CPU Page Frame

Cache)”

TARO
2021-05-20 22:06 (http://jake.dothome.co.kr/per-cpu-page-frame-cache/#comment-305337)

Hello

In
the __rmqueue_pcplist function, page = list_first_entry(list, struct page, lru) to get page.

If you look at the list_first_entry function, you get the page from the first entry in the list using the offset of LRU,
which is the concatenation list inside the struct page.

If the list is created by the setup_per_cpu_pageset() function,
is this list the same as the LRU of the struct page?

RESPONSE (/PER-CPU-PAGE-FRAME-CACHE/?REPLYTOCOM=305337#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-05-22 14:16 (http://jake.dothome.co.kr/per-cpu-page-frame-cache/#comment-305338)

Hello? Mr. Taro.

The last question sentence "Is this list the same as the LRU on the struct page?" Except for that,
you solved it correctly. The last sentence of the question makes it difficult for me to grasp the exact intent of the question.

The list is the lists[] of the per_cpu_pages struct member, and the pages are hung from this list.
list = &pcp->lists[migratetype];

If you haven't figured it out yet, I'd love to hear you clarify the last question a little more closely.

RESPONSE (/PER-CPU-PAGE-FRAME-CACHE/?REPLYTOCOM=305338#RESPOND)

TARO
2021-05-23 16:34 (http://jake.dothome.co.kr/per-cpu-page-frame-cache/#comment-305342)

What
I don't really understand is the part in your answer that says 'pages are stuck in lists[]'.

When a page hangs on lists[], is the address of the struct list_head lru inside the page hanging?

page = list_first_entry(list, struct page, lru) 를 통해 page를 가져올 때
list_first_entry(list, struct page, lru)
list_entry(list->next, struct page, lru)
container_of(list->next, struct page, lru) 이고,

In the end, it becomes (struct page *)(list->next – offsetof(struct page, lru)) and we get the address of the page by subtracting the offset of the LRU inside the struct page.

그러려면, lists[]에 저장되는 것이 struct page 내부의 lru의 주소여야 한다고 생각하는데
제가 생각하는 것이 맞을까요?

응답 (/PER-CPU-PAGE-FRAME-CACHE/?REPLYTOCOM=305342#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2021-05-24 14:11 (http://jake.dothome.co.kr/per-cpu-page-frame-cache/#comment-305349)

네. 맞습니다.

list에 두 개의 페이지가 연결되어 있는 경우 다음과 같이 연결됩니다.
(참고로 lru는 double linked list로, list_head 구조체이며 *next와 *prev 멤버가 사용됩니다)

list < ----> A page->lru < ----> B page->lru

위를 더 자세히 표현하면 다음과 같습니다.

list->next —> A page->lru.next —> B page->lru.next —> 맨 앞 list->next
list->prev < --- A page->lru.prev < --- B page->lru.prev < --- 맨 앞 list->prev

감사합니다.

응답 (/PER-CPU-PAGE-FRAME-CACHE/?REPLYTOCOM=305349#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

Control Group for Memory (http://jake.dothome.co.kr/mem_cgroup/)

Zoned Allocator -10- (LRU & pagevecs) (http://jake.dothome.co.kr/lru-lists-pagevecs/)

문c 블로그 (2015 ~ 2023)