Zoned Allocator -10- (LRU & pagevecs)

2016-04-26 (http://jake.dothome.co.kr/lru-lists-pagevecs/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Memory Reclaiming

When memory is insufficient, there is a process that periodically cancels pages, and pages are recalled and reused, and there are various memory replacement policies. Among them, Linux uses the LRU algorithm.

The Buddy system, which is the core of the kernel's page management, manages free pages for six migratetypes for each order and multiple orders. However, if you run out of memory when allocating and using them, you can reclaim them for specific allocation pages, which you will learn about in the following.

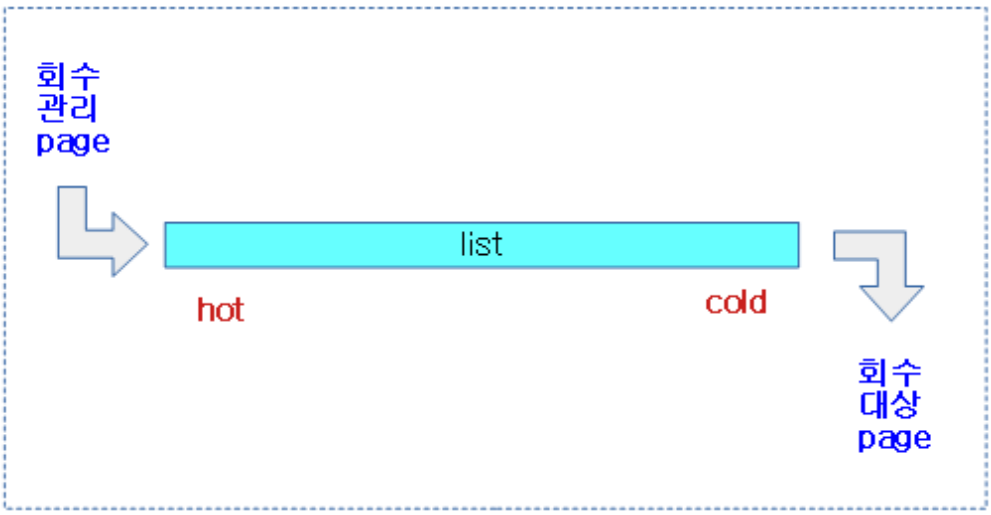
- file (aka page cache)
  - Since the pages of the file that are read by the user and are in memory are copies, and the originals are already stored in the backing storage system (disk, etc.), the pages in the memory can be freed and retrieved immediately. After that, if necessary, it will be loaded and used again.
  - If a page loaded into memory has been modified (dirty), it is written to disk and then retrieved.
  - LRU list.
- ANON Pages
  - Since the memory requested by the user with malloc() or stack memory is the original memory, it can be temporarily stored in the swap backing storage system (disk, etc.) and then released from memory to retrieve it.
  - LRU list.
- Reclaimable Slab Caches
  - Kernel-allocated memories are immovable, making them impossible to compaction and reclaim, but slap caches created using the GFP_RECLAIMABLE option are designed to be retrievable even if they are used as kernel memory.
  - It is not managed through the LRU list.

LRU (Least Recently Used)

Pages used for recall management are managed through the LRU list. Recognize them.

- It is a method of recalling pages that are used with a minimum frequency.

- In the actual implementation, the minimum frequency processing is done by adding the pages to be recalled to the head of the list, and the pages to be recalled are handled in the tail of the list.
- file page and anon page.



(http://jake.dothome.co.kr/wp-content/uploads/2016/04/lru-3.png)

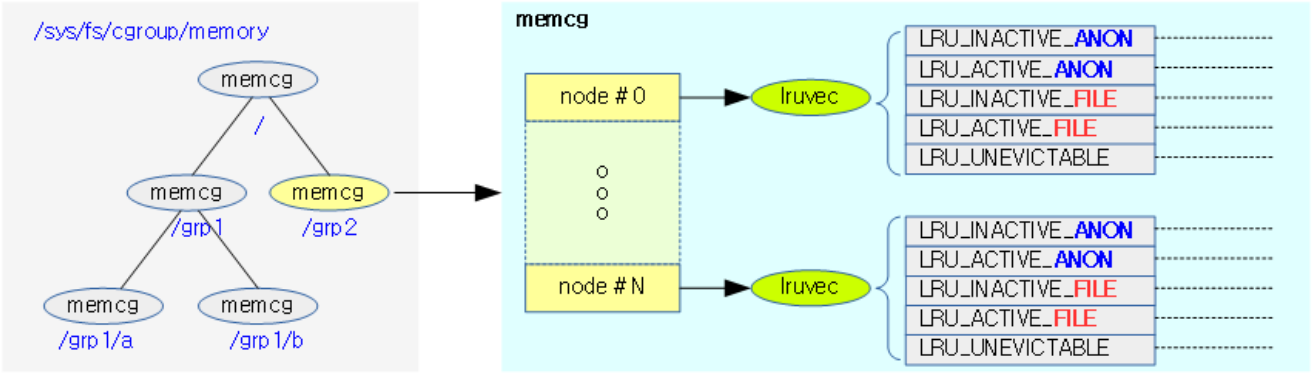
- The recall is for the user-allocated memory, the anon page and the page cache.
  - However, it excludes pages from the page cache that are classified as unevictable: ramfs, shm, and mlocks.
- The pages managed by LRU are movable pages that can be migrated via compaction.
- As of kernel v2.6.28-rc1, the list of only 2 LRUs (active_list and inactive_list) was managed for each zone, but it was divided into anon and file, and the unevictable list of pages to be excluded from recall has been expanded to a total of 5.
  - Existing
    - zone->active_list
    - zone->inactive_list
  - New: LRU list expanded to 5
    - zone->lruvec.lists[LRU_INACTIVE_ANON]
    - zone->lruvec.lists[LRU_ACTIVE_ANON]
    - zone->lruvec.lists[LRU_INACTIVE_FILE]
    - zone->lruvec.lists[LRU_ACTIVE_FILE]
    - zone->lruvec.lists[LRU_UNEVICTABLE]
  - Consultation:
    - vmscan: Use an indexed array for LRU variables (https://github.com/torvalds/linux/commit/b69408e88bd86b98feb7b9a38fd865e1ddb29827)
    - vmscan: split LRU lists into anon & file sets (https://github.com/torvalds/linux/commit/4f98a2fee8acdb4ac84545df98cccecfd130f8db)
- As of kernel v4.8-rc1, it is managed by node rather than zone.

- See: mm, vmscan: move LRU lists to node (https://github.com/torvalds/linux/commit/599d0c954f91d0689c9bb421b5bc04ea02437a41) (2016, v4.8-rc1)

memcg/node lruvecs

As shown in the following figure, the memory controller using cgroup is called memcg, and each memcg manages lruvec for each node.

- The lruvec structure contains the above 5 LRU lists.
- lruvec is managed by each memcg (Cgroup's memory controller) and nodes.
- In other words, a user page is managed by one of many lruvec.



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

LRU List Types

LRU lists are interactive lists, with hot at the top and cold at the end of the page.

- ANON
  - anonymous user memory is mapped to a VM.
  - If there is not enough memory, it will be moved to the swap area and the finished pages will be retrieved.
    - Currently, the Linux kernel has a swap size set to default 0 for performance reasons.
    - Recently, Torvalds has been interested in using SSD-type disks to use Swap again.
      - See: Reconsidering swapping (http://lwn.net/Articles/690079/) | LWN.net
- FILE
  - This is the page that is used to map the file to the VM, and it is the page that is loaded from the regular file.
  - When memory is insufficient, clean pages are simply retrieved, and dirty pages are retrieved after writing to file(backing store).
- ACTIVE
  - The first assigned pages are added to the hot list of inactives.
  - Periodically, the ratio of active and inactive is compared, and pages that are not constantly referenced are moved to the inactive list, and referenced pages are moved back to the top of the active list.
- INACTIVE

    - When the retrieval mechanism is activated, it attempts to retrieve from the cold side of the inactive list.
        - ANON: Swap area.
        - FILE: clean page is immediately reclaimable. If it's a dirty page, use writeback and have it async to the original file, then rotate the page to the front of the inactive list.
            - rotate to suspend processing immediately, and then reclaim it later when it's your turn to write back when it's complete.
- UNEVICTABLE
    - It is used in the following cases as a single page that cannot be used by the memory reclamation mechanism.
        - ramfs
        - SHM_LOCK Shared Memory Regions
        - VM_LOCKED VMAs
    - In the following three cases, migration through isolation is allowed.
        - Managing Memory Fragmentation
        - Workload Management
        - Memory hotplug
    - It doesn't use the LRU pagevec mechanism, which uses per-cpu.

Moving Between LRU Lists

- Anon Page
    - A fault occurs and the newly assigned page enters the head of the inactive list.
        - With the addition of Workingset Detection in kernel v5.9-rc1, the first faulted anon page is added to the head of the inactive list, just like the file page.
            - Note: mm/vmscan: protect the workingset on anonymous LRU (https://github.com/torvalds/linux/commit/b518154e59aab3ad0780a169c5cc84bd4ee4357e) (2020, v5.9-rc1)
    - If there is evidence that the memory has been accessed more than twice by applicaton or kernel for pages scanned from the tail of the inactive list, it is moved to the head of the active list.
        - This is called activating or promoting the page.
    - Swap the scanned page from the tail of the incactive list and revert it to buddy.
    - Pages that are pushed out of the Tail of the Active List to match the Active/Inactive ratio are moved to the head of the Inactive List.
        - This is called the deactivate or demotion of the page.
- file page
    - A fault occurs and the newly assigned page enters the head of the inactive list.
    - The deactivate/demotion or activate/promotion process is identical to the anon page, except for the following:
        - In the case of executable files, even if they are accessed only once, they are promoted.
    - Pages scanned from the tail of the incactive list are written to the backing-storage where the original was located before the page that was dirty before reclaiming. Pages that have

been written back are reverted to buddy.

The following figure shows the LRU lists used to reclaim pages.

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

LRU Related Page Flags

The following page flags are used for LRU management:

- PG_lru
  - It is used while it is managed in the LRU list. If the page is in the pagevec list, which is the LRU cache, the PG_lru is clear.
- PG_active
  - It is set while it is managed in the active list, and it is cleared when it is moved to the inactive list.
- PG_swapbacked
  - When an ANON page is first created, this flag is set to indicate that the page is swappable, regardless of whether swap backing storage is set or not.
  - The madvise() API's MADV_FREE option allows you to change the page to a lazy free state when disabled, at which point this flag is cleared to temporarily create a clean anon page.
- PG_referenced
  - It is used for the purpose of checking if a page has been referenced more than twice recently, activating it, and moving it to the active list, and it is used in conjunction with the AF (Access Flag) of the PTE (Page Table Entry).
  - Continue with more detail in the topics below.
- PG_writeback
  - It is set only while the anon and file pages are being written to backing storage.
- PG_dirty
  - This flag is set when the file page is loaded into memory by open() and then read() and then the memory of the page is changed by write().
- PG_reclaim

    - This flag is set on the page that is subject to recall, and this flag is cleared before the retrieved free page is returned to the buddy system.
- PG_workingset
    - It is set when a page managed in the active list is moved to inactive, and it is used to hide the presence of a page refault so that the performance is not degraded through repeated disk IO due to frequent faults. When a page is stored in backing storage, the state of this flag should also be remembered for some time, as the page is stored and managed in a shadow entry at a time when the cache used to read and write to backing storage (page cache or swap cache) is not in use.
    - e.g. when a file is loaded, it is stored in the page cache, and this information is managed using the xarray data structure. If the page cache is emptied due to insufficient memory, the xarray will also erase information about the page cache, but will then use it to record shadow information. This shadow information contains relevant information about the eviction page (such as whether or not it is working).

This is a bit of an old article, but there is a gem article that explains page flags well, so please refer to the following article, and focus on the changes to page flags that are not mentioned in this article.

- See: [Linux] Lifetime of memory with pageflags (http://studyfoss.egloos.com/5512112) | F/OSS Study
- Note: Some flags, such as PG_buddy, have been removed from the _mapcount of the page and moved to the newly added page_type.

Page flag status changes

Notice the flag changes when the following pages are reclaimed:

- Anon Page
    - When a fault occurs, the newly assigned anon page is set with the PG_swapbacked flag set, and it starts from the inactive list.
        - Note: A page without PG_swapbacked on an ANON page is called a Clean Anon page, which means a page with a lazy free status.
    - There are cases where the promotion is moved from the inactive list back to the inactive list, or promoted to the active list, and the PG_active flag is set in the case of promotion.
    - When swapping scanned pages in the tail of the incactive list, the PG_writeback and PG_reclaim flags are set.
    - The swap is started via the add_to_swap() function, and when the swap is complete, the PG_writeback is cleared, and then the PG_reclaim flag is cleared to return to the buddy system.
    - When demotion from an active list to an inactive list, the PG_workingset flag is set, and the PG_active flag is cleared.
    - In the case of a refaulted activate page, set the PG_active flag, keep the PG_workingset flag at the time of eviction, and start from the activate list.
- file page

- In the event of a fault, the newly assigned file page starts from the inactive list.
- There are cases where the promotion is moved from the inactive list back to the inactive list, or promoted to the active list, and the PG_active flag is set in the case of promotion.
- If a scan is selected for retrieval and a PG_reclaim flag is set.
- The recall starts with the pageout() function, and the page that has been changed in memory by the user must be written to the backing storage where the original was before the PG_dirty is already set, and the PG_writeback is set during this period.
- Once the writeback is complete, the PG_writeback and PG_dirty flags will be cleared and the clean file page will be in a state of clean, after which the PG_reclaim flag will also be cleared to return to the buddy system.
- When demotion from an active list to an inactive list, the PG_workingset flag is set, and the PG_active flag is cleared.
- In the case of a refaulted activate page, set the PG_active flag, keep the PG_workingset flag at the time of eviction, and start from the activate list.

Page References

By checking whether a page has been referenced more than twice by applicaton and the kernel, it determines that it is an active page and moves it to the head of the active list.

- If the page was in the active list, rotate it to move it to the head of the list.
- If the page was in the inactive list, it is promoted to move it to the head of the active list.

1. AF (Access Flag) of Page Table Entry (PTE)

In order to check whether a page has been referenced for the first time, the kernel receives and handles the fault event of the HW architecture. Fault events are caused by a variety of causes, but the items related to page references can be summarized into three items:

- PTE fault
  - Occurs when an application attempts to read an unmapped virtual address space when accessing the page.
    - do_anonymous_page, do_fault, do_swap_page, do_numa_page
- Permission fault
  - Occurs when an application attempts to write to a virtual address space that has been mapped to read-only.
    - do_wp_page
- Access Flag fault
  - Occurs when the PTE's AF bit is not set and you try to access a page.
  - Now that the page has been accessed, the kernel sets the AF bit of the PTE directly in a SW manner. Example: ARMv8.0 or earlier
    - pte_mkyoung – ptep_set_access_flags
  - Recent architectures allow the HW to set the AF bit of the PTE entry directly. Example: ARMv8.1 or later

2. PG_reference

Since the first referenced and a newly loaded or created page has been accessed for the first time, the kernel (or automatically if the HW supports it) sets the AF bit of the PTE.

- Depending on whether this page is a file page or an anon page, the file page starts at the head of the inactive list and the anon page starts at the head of the active list, respectively.
- In the list you started, the page gradually slides to the tail of the list over time, and at the end it becomes the target of scanning for reclaim, so that when you page_check_reference the reference of this page, it knows whether the AF bit of the mapped PTE has been set, and it also examines the PG_reference flag. At this point, the AF bit of the PTE should be cleared for the next reference survey.

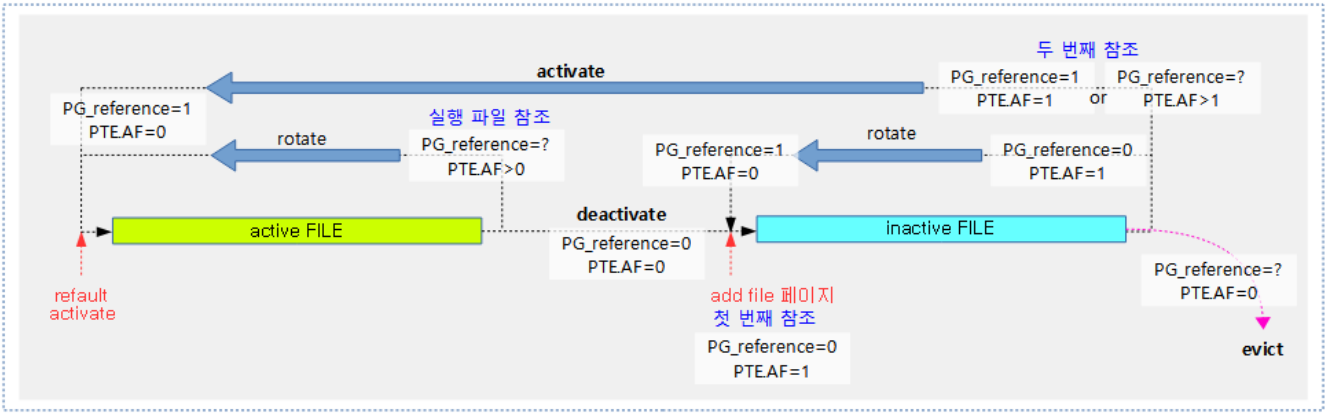
The following describes the processing of pages scanned from the tail of the inactive list.

- In the following case, we set a PG_reference flag for activation, and then move it directly to the head of the active list, which is called promote or activate.
    - If you check two or more references, with or without the PG_reference flag,
    - If 1 executable file is referenced with or without the setting of the PG_reference flag.
    - PG_reference flag is set, and 1 reference is checked.
- If the PG_reference flag is not set, and only one reference is identified, set the PG_reference flag and **rotate** it from the original list to the head of the keep list.
- If no reference is found, the PG_reclaim is set for the evict on this page, with or without the PG_reference flag. Subsequent writebacks on dirty pages on file pages and swaps on anon pages are omitted because they are unrelated to setting the PG_reference flag.

The following describes the processing of pages scanned from the tail of the active list.

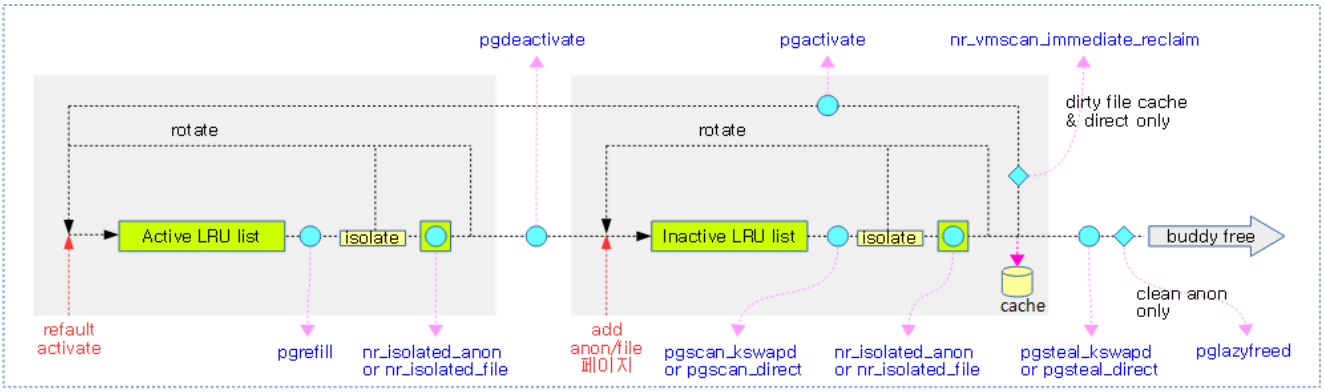
- If more than one executable is referenced, with or without the PG_reference flag, it is **rotated** from the active list to the head of the keep list.
- If it is not a reference condition above, it is judged to be an inactive page, clears the PG_reference flag, and moves it to the head of the inactive list, which is called demote or deactivate.

The following figure shows the change in the flags for references on the file page.



(http://jake.dothome.co.kr/wp-content/uploads/2016/04/lru-4d.png)

The following figure shows the VM counter values involved when a page is reclaimed via LRU.



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

Anon Page

The path to the Anon page is as follows:

- This is a page that the user application has requested to be allocated anonymous to the kernel due to an increase in heap or stack memory.
- When modification of an open shared file occurs, it is a page that is copied from the fault handler using the Copy On Write (COW) function.
- Pages shared by the KSM (Kernel Same Memory) function are also anon pages.

The anon page indicates with a PG_swapbacked flag whether the swap area is available.

- normal anon page
  - This is an anon page with the PG_swapbacked flag set to an anon page with a swap area.
- Clean Anon Page
  - This is an anon page that does not have a swap area and does not have the PG_swapbacked flag set.
  - MADV_FREE page is lazy-free.
    - consultation
      - mm: support madvise(MADV_FREE) (https://lwn.net/Articles/590693/) (2014) | LWN.net

- Volatile ranges and MADV_FREE (https://lwn.net/Articles/590991/) (2014) | LWN.net

pagevecs

Pageveces is an LRU cache. In the page retrieval mechanism, a certain portion of the page in the LRU list is placed in isolation. However, in places where batch processing is not possible, if locks are acquired and processed one by one at the time of request, the performance is reduced by lock contention, so a separate LRU cache is implemented and used. There are five pagevecs implemented with per-cpu, each capable of managing 5 pages.

- lru_add_pvec
- lru_rotate_pvecs
- lru_deactivate_file_pvecs
- lru_lazyfree_pvecs
  - See: mm: move MADV_FREE pages into LRU_INACTIVE_FILE list (https://github.com/torvalds/linux/commit/f7ad2a6cb9f7c4040004bedee84a70a9b985583e) (2017, v4.12-rc1)
- activate_page_pvecs

The following figure shows the calling relationship of a function that uses the LRU cache PageVecs.

- Each time a function is called, it is added to the LRU cache, pagevecs, but if it exceeds the processing limit of 14, it is added directly to the LRU.
- To retrieve a page from the LRU cache Pagevecs to the LRU list, call the lru_add_drain_cpu() function.

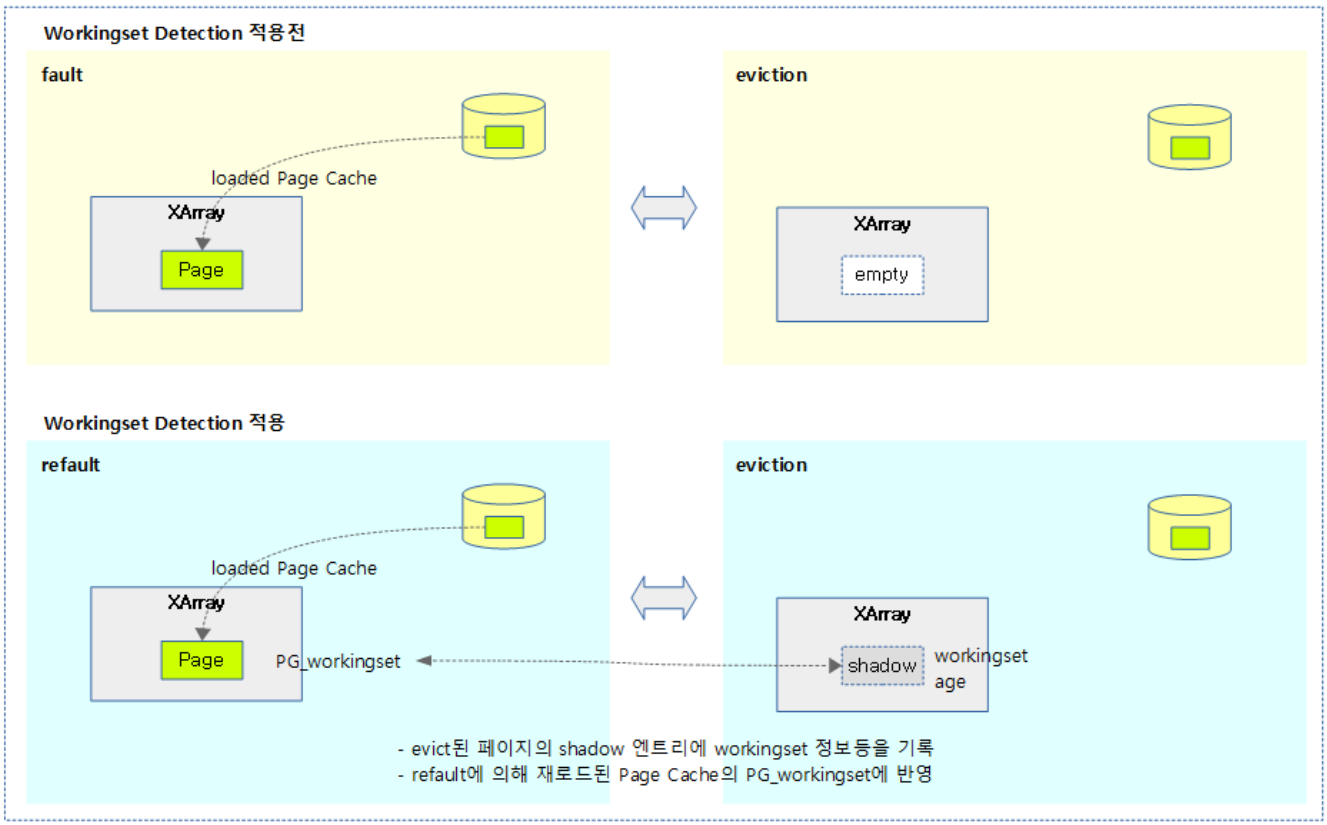
(http://jake.dothome.co.kr/wp-content/uploads/2016/04/pagevecs-1.png)

Workingset Detection

In order to prevent disk IO costs from increasing due to repeated refaults due to repeated recalls of pages, an algorithm related to Workingset Detection was applied.

- It records information in the PG_workingset flag and shadow entries in the swap cache and page cache, and calculates anon/file cost for each lruvec.
- Consultation:
  - mm: balance LRU lists based on relative thrashing (https://github.com/torvalds/linux/commit/314b57fb0460001a090b35ff8be987f2c868ad3c) (2020, v5.8-rc1)
  - mm: workingset: tell cache transitions from workingset thrashing (https://github.com/torvalds/linux/commit/1899ad18c6072d689896badafb81267b0a1092a4) (2018, v4.20-rc1)
  - mm: workingset: eviction buckets for bigmem/lowbit machines (https://github.com/torvalds/linux/commit/612e44939c3c77245ac80843c0c7876c8cf97282) (2016, v4.6-rc1)
  - mm: thrash detection-based file cache sizing (https://github.com/torvalds/linux/commit/a528910e12ec7ee203095eb1711468a66b9b60b0) (2014, v3.15-rc1)

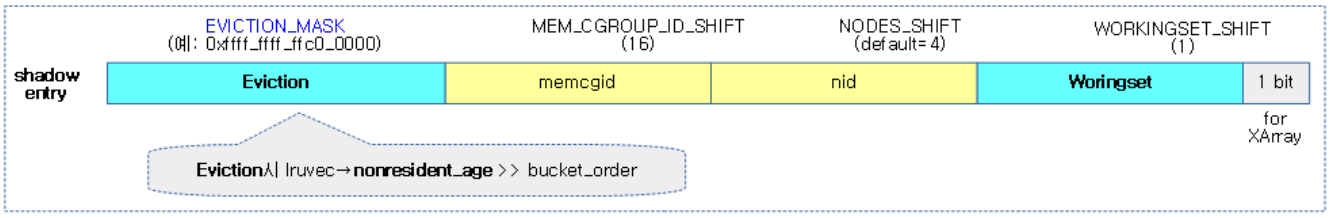
The following illustration shows XAraay's shadow entries that manage the cache to record some of the information on the page when it is evicted, such as the PG_workingset and age of the page, so that when it is later refaulted and reloaded, it can be updated to show that it was already workingset.



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

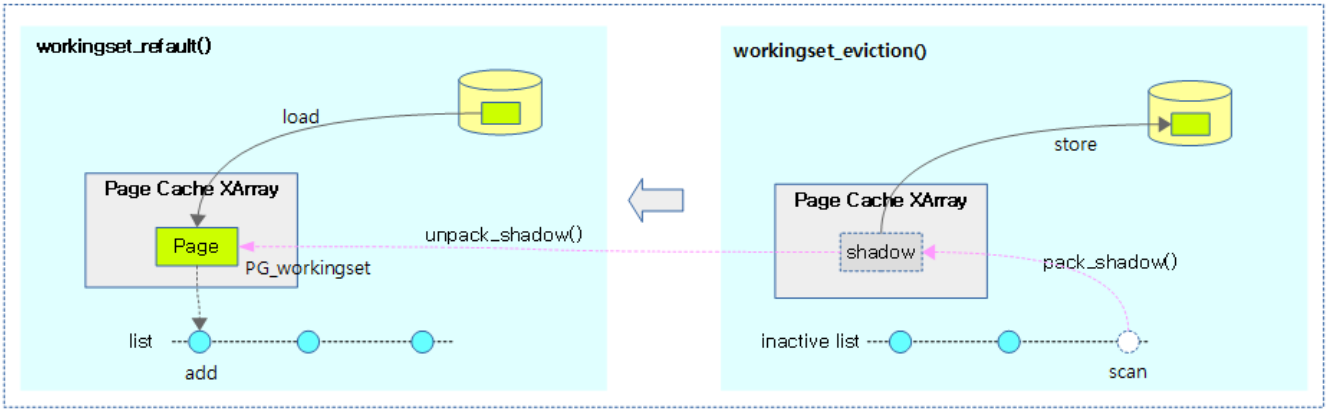
The following illustration shows the values stored in the shadow entry of the Xarray.

- →nonresident_age If the system memory is large or the system is 32-bit, there may not be enough extra bits to store it, so shift it to the right by bucket_order and shift it to the left by bucket_order when ejecting and using it in case of refailure.
- As much as the values are shifted, the lower bits are trimmed to a cleared state, resulting in a rough value, and the comparison is inevitably rough.



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

The following diagram shows the process of creating (packing) or unpacking (refaulting) the values stored in the Xarray's shadow entries.



(http://jake.dothome.co.kr/wp-content/uploads/2016/04/workingset-3.png)

Workingset Detection for file page

Determine whether a Refualt page is activated or not

Here are the factors to determine whether or not you should activate the refault page:

- NR_inactive
  - The number of pages of this LRU type managed by the Inactive LRU list.
- NR_active
  - The number of pages of this LRU type managed by the Active LRU list.
- lruvec->nonresident_age
  - The cumulative number of pages that a page has activated and eviction increases. It is similar to timestamp.
  - There are two paths to activation values.
    - If the page is promoted from the inactive list,
    - The refault page goes straight to the active list.
- PG_workingset
  - A flag that tells you whether the page is workingset or not

Determine whether a Refualt File page is activated or not

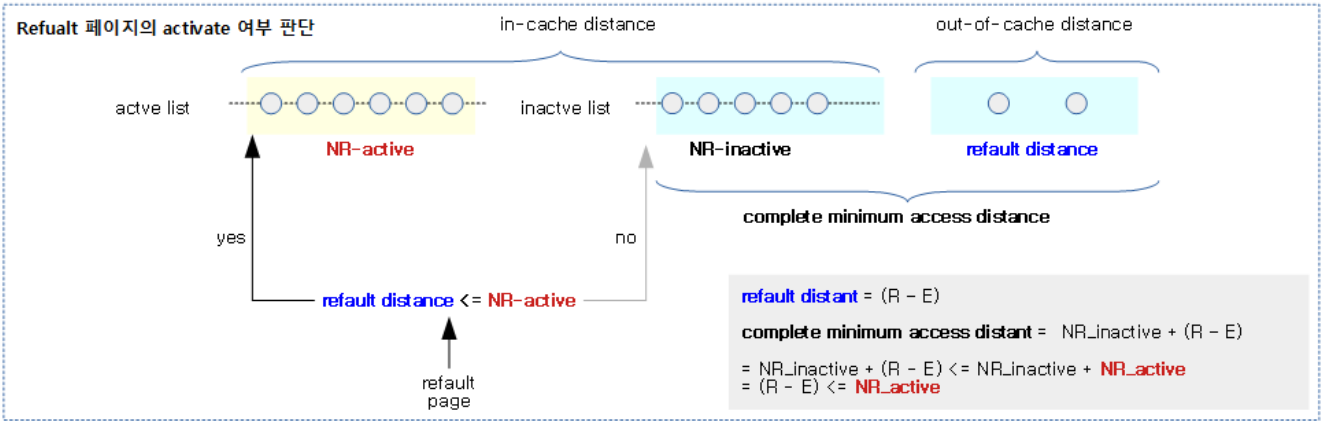
Using the above element, use the following formula (when the Workingset Detection feature was first introduced, it only supported the file cache):

- R
  - LRU->nonresident_age value at the moment of refault
- E
  - LRU->nonresident_age value at the moment of eviction
- refault distant
  - = (R – E)
  - The interval between when it has been refaulted since it was evictioned
- complete minimum access distant
  - = NR_inactive + (R – E)
- Determine if you want to activate
  - = refault distant + NR_inactive <= NR_active + NR_inactive

- = refault distant <= NR_active

The following image shows the process by which the refault file page is added to the active list.

- If the page is refaulted in a short period of time (refault distance <= NR_active) and entered, activate it.



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

Workingset Detection for anon/file page

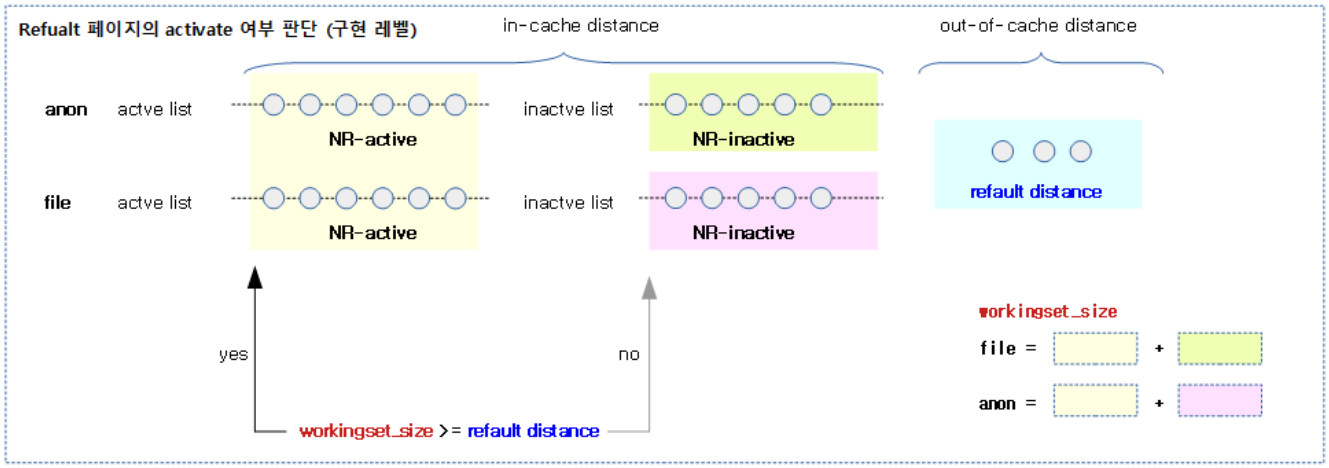
The ANON page can also be refaulted by calculating the refault distance and workingset_size (including a new anon instead of NR_active_file) and comparing them to determine whether to activate or not.

- Anon Page Formulas:
  - = refault distant + NR_inactive_anon <= NR_active_anon + NR_inactive_anon + NR_inactive_file + NR_inactive_file
  - = refault distant <= NR_active_anon + NR_inactive_file + NR_inactive_file
  - = refault distant <= workingset_size
    - workingset_size = NR_active_anon + NR_inactive_file + NR_inactive_file
- file page formula:
  - = refault distant + NR_inactive_file <= NR_active_anon + NR_inactive_anon + NR_inactive_file + NR_inactive_file
  - = refault distant <= NR_active_anon + NR_inactive_anon + NR_active_file
  - = refault distant <= workingset_size
    - workingset_size = NR_active_anon + NR_inactive_anon + NR_active_file
- However, if there is no swap space, the number associated with Anon is not included.
- Consultation:
  - mm/swap: implement workingset detection for anonymous LRU (https://github.com/torvalds/linux/commit/aae466b0052e1888edd1d7f473d4310d6493619 6) (2020. v5.9-rc1)
  - mm/workingset: prepare the workingset detection infrastructure for anon LRU (https://github.com/torvalds/linux/commit/170b04b7ae49634df103810dad67b22cf8a99aa 6) (2020, v5.9-rc1)
  - mm: workingset: age nonresident information alongside anonymous pages (https://github.com/torvalds/linux/commit/31d8fcac00fcf4007f3921edc69ab4dcb3abcd4d)

(2020, v5.8-rc3)

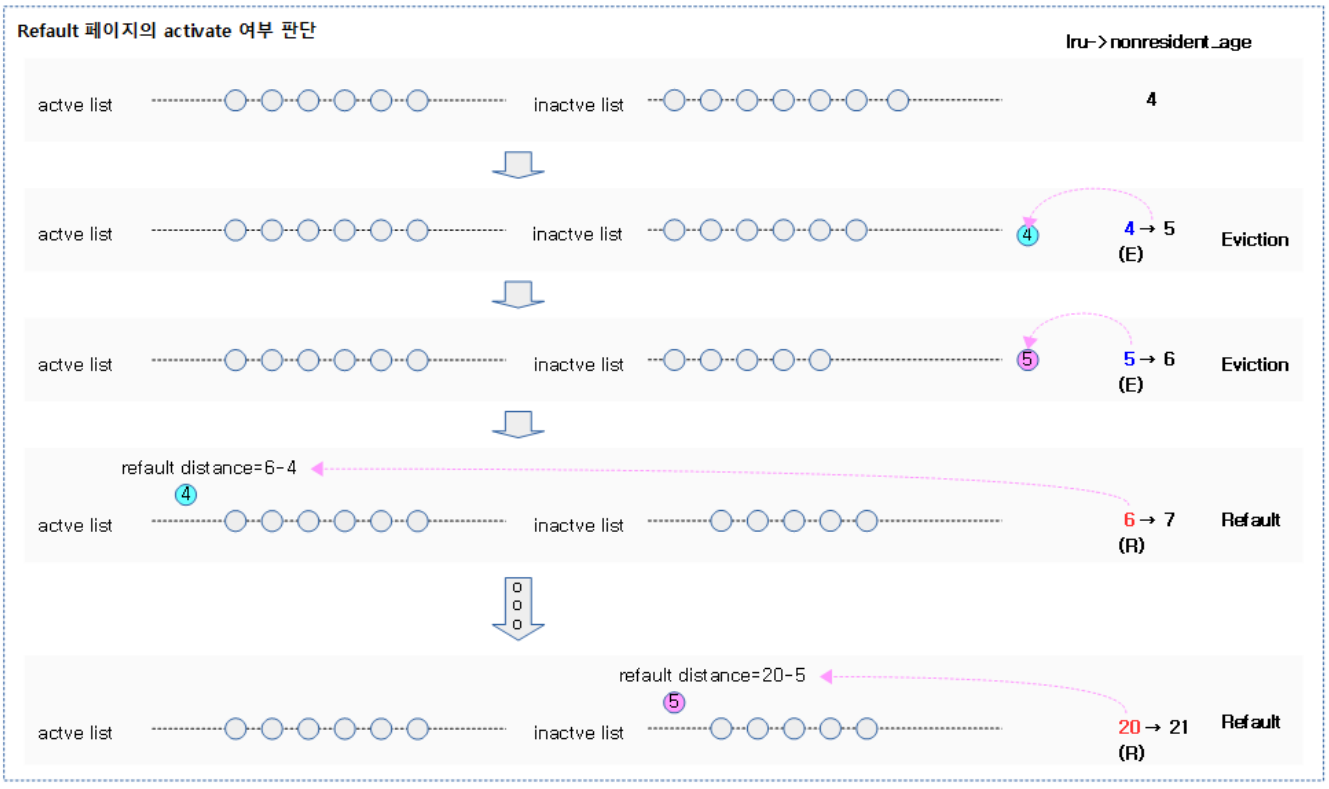
- mm: workingset: let cache workingset challenge anon (https://github.com/torvalds/linux/commit/34e58cac6d8f2a76b609b3510ff0c4468a220e61) (2020, v5.8-rc1)

The following figure shows the process of activating a refault page when supporting the new Workingset Detection for file/anon pages.



(http://jake.dothome.co.kr/wp-content/uploads/2016/04/refault-3.png)

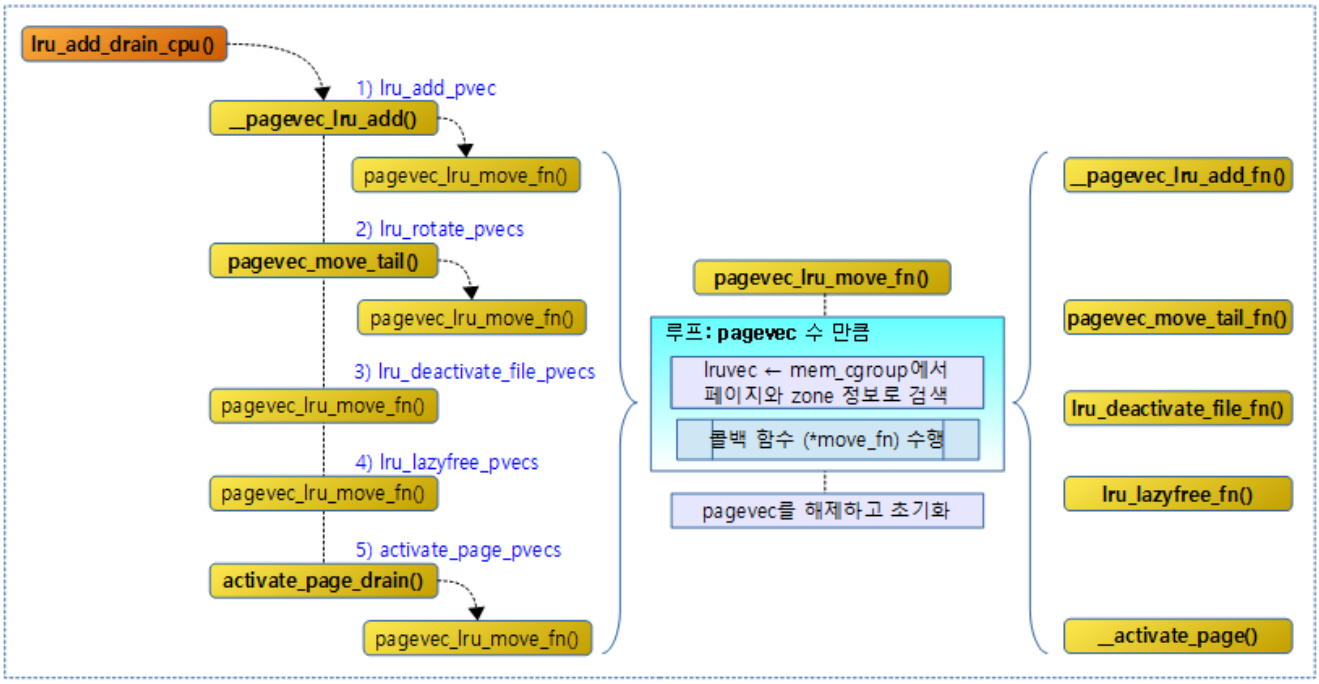
The following figure shows the process of determining whether or not to activate the refault distance value of the refault page according to the short time and long time.



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

Drain from per-cpu LRU cache (pagevec)

The following figure shows the invocation relationship of the lru_add_drain_cpu() function.



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

lru_add_drain_cpu()

mm/swap.c

```
1  /*
2   * Drain pages out of the cpu's pagevecs.
3   * Either "cpu" is the current CPU, and preemption has already been
4   * disabled; or "cpu" is being hot-unplugged, and is already dead.
5   */
```

```
01 void lru_add_drain_cpu(int cpu)
02 {
03         struct pagevec *pvec = &per_cpu(lru_add_pvec, cpu);
04
05         if (pagevec_count(pvec))
06                 __pagevec_lru_add(pvec);
07
08         pvec = &per_cpu(lru_rotate_pvecs, cpu);
09         if (pagevec_count(pvec)) {
10                 unsigned long flags;
11
12                 /* No harm done if a racing interrupt already did this
   */
13                 local_irq_save(flags);
14                 pagevec_move_tail(pvec);
15                 local_irq_restore(flags);
16         }
17
18         pvec = &per_cpu(lru_deactivate_file_pvecs, cpu);
19         if (pagevec_count(pvec))
20                 pagevec_lru_move_fn(pvec, lru_deactivate_file_fn, NULL);
21
22         pvec = &per_cpu(lru_lazyfree_pvecs, cpu);
23         if (pagevec_count(pvec))
```

```
24                 pagevec_lru_move_fn(pvec, lru_lazyfree_fn, NULL);
25
26         activate_page_drain(cpu);
27 }
```

Reclaim the five per-CPU caches used by the page allocator's reclamation mechanism lruvec used by the specified @cpu and move them to lruvec in the zone (or memcg's zone)

- Pages registered in the @cpu cache lru_add_pvec specified in lines 3~6 of the code are migrated to lruvec in the page zone and emptied.
- Move the page registered to the @cpu cache lru_rotate_pvecs specified in lines 8~16 to the last position in lruvec in the page zone and empty it.
- Pages registered in the @cpu cache lru_deactivate_file_pvecs specified in lines 18~20 are migrated to lruvec in the page zone and emptied.
- Pages registered in the @cpu cache lru_lazyfree_pvecs specified in lines 22~24 of the code are moved to lruvec in the page zone and emptied.
- Pages registered in the @cpu cache activate_page_pvecs specified in line 26 are migrated to lruvec in the zone of that page and emptied them.

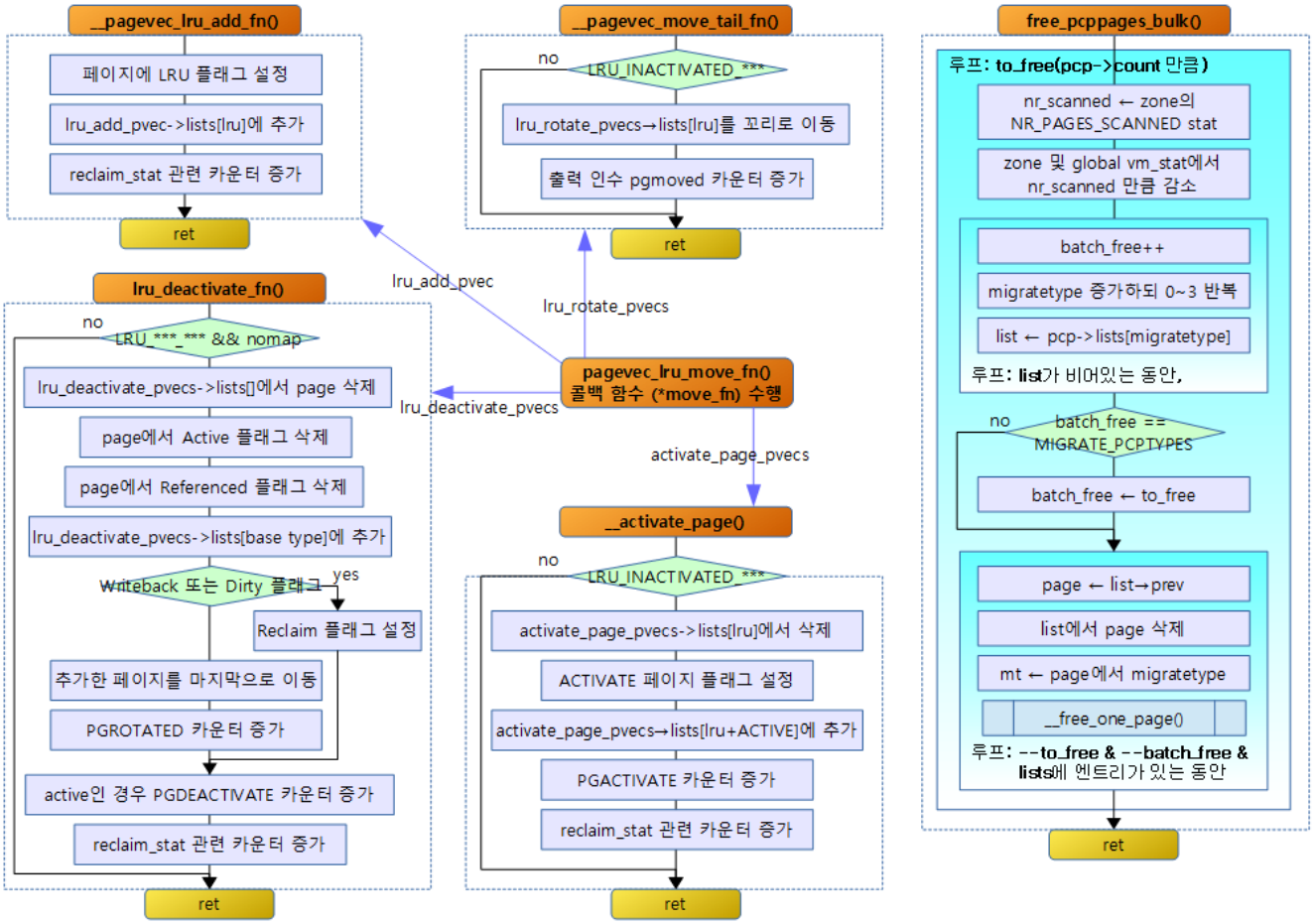
__pagevec_lru_add()

mm/swap.c

```
1 /*
2  * Add the passed pages to the LRU, then drop the caller's refcount
3  * on them.  Reinitialises the caller's pagevec.
4  */
```

```
1 void __pagevec_lru_add(struct pagevec *pvec)
2 {
3         pagevec_lru_move_fn(pvec, __pagevec_lru_add_fn, NULL);
4 }
5 EXPORT_SYMBOL(__pagevec_lru_add);
```

Move the pages registered in the CPU cache pagevec to the page zone (or zone in the memory cgroup)->lruvec, empty the pagevec, and initialize it.



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

pagevec_move_tail()

mm/swap.c

```
1  /*
2   * pagevec_move_tail() must be called with IRQ disabled.
3   * Otherwise this may cause nasty races.
4   */
```

```
1  static void pagevec_move_tail(struct pagevec *pvec)
2  {
3          int pgmoved = 0;
4
5          pagevec_lru_move_fn(pvec, pagevec_move_tail_fn, &pgmoved);
6          __count_vm_events(PGROTATED, pgmoved);
7  }
```

Add the pages registered to Pagevec to the back of the list by type of LRU in the Memory Control Group of the page, empty Pagevec, and initialize it. Add the number of pages you've added to the vm_events relevant pgmoved entry.

activate_page_drain()

mm/swap.c

```
1  static void activate_page_drain(int cpu)
2  {
3          struct pagevec *pvec = &per_cpu(activate_page_pvecs, cpu);
4
```

```
5           if (pagevec_count(pvec))
6                   pagevec_lru_move_fn(pvec, __activate_page, NULL);
7   }
```

Delete the pages registered in the CPU cache list called activate_page_pvecs from the list of LRU's type in the memory control group, add LRU's type + active to the hot, set the active flag, increment the vm_events-related PGACTIVATE entry, and increment the reclaim-related statistics. Then empty pagevec and initialize it.

---

5 pagevec migration functions

1) Common Migration Functions

pagevec_lru_move_fn()

mm/swap.c

```
01  static void pagevec_lru_move_fn(struct pagevec *pvec,
02          void (*move_fn)(struct page *page, struct lruvec *lruvec, void *
    arg),
03          void *arg)
04  {
05          int i;
06          struct pglist_data *pgdat = NULL;
07          struct lruvec *lruvec;
08          unsigned long flags = 0;
09
10          for (i = 0; i < pagevec_count(pvec); i++) {
11                  struct page *page = pvec->pages[i];
12                  struct pglist_data *pagepgdat = page_pgdat(page);
13
14                  if (pagepgdat != pgdat) {
15                          if (pgdat)
16                                  spin_unlock_irqrestore(&pgdat->lru_lock,
    flags);
17                          pgdat = pagepgdat;
18                          spin_lock_irqsave(&pgdat->lru_lock, flags);
19                  }
20
21                  lruvec = mem_cgroup_page_lruvec(page, pgdat);
22                  (*move_fn)(page, lruvec, arg);
23          }
24          if (pgdat)
25                  spin_unlock_irqrestore(&pgdat->lru_lock, flags);
26          release_pages(pvec->pages, pvec->nr);
27          pagevec_reinit(pvec);
28  }
```

Move the page registered in Pagevec to Lruvec in the memory control group of the page, empty the pagevec, and initialize it.

- In line 10~19 of the code, it traverses the number registered in the pagevec list, and every time the node changes, it unlocks the spin lock and reacquires it. Suppresses the acquisition of locks for a long time.
- On lines 21~22 of code, move the page to the lruvec list in memcg to which it belongs. If there is no memcg, use the lruvec list of the nodes.
    - move_fn calls the function specified in the argument.

    - e.g. __pagevec_lru_add_fn()
        - Add a page from pagevec to lruvec.
- At line 25 of the code, terminate the pages in pagevec.
- Reinitialize pagevec at line 26 of code.

2) 5 migration functions

__pagevec_lru_add_fn()

mm/swap.c

```
01 static void __pagevec_lru_add_fn(struct page *page, struct lruvec *lruvec,
02                                   void *arg)
03 {
04         enum lru_list lru;
05         int was_unevictable = TestClearPageUnevictable(page);
06
07         VM_BUG_ON_PAGE(PageLRU(page), page);
08
09         SetPageLRU(page);
10         /*
11          * Page becomes evictable in two ways:
12          * 1) Within LRU lock [munlock_vma_pages() and __munlock_pagevec()].
13          * 2) Before acquiring LRU lock to put the page to correct LRU and then
14          *   a) do PageLRU check with lock [check_move_unevictable_pages]
15          *   b) do PageLRU check before lock [clear_page_mlock]
16          *
17          * (1) & (2a) are ok as LRU lock will serialize them. For (2b), we need
18          * following strict ordering:
19          *
20          * #0: __pagevec_lru_add_fn             #1: clear_page_mlock
21          *
22          * SetPageLRU()                         TestClearPageMlocked()
23          * smp_mb() // explicit ordering        // above provides strict
24          *                                      // ordering
25          * PageMlocked()                        PageLRU()
26          *
27          *
28          * if '#1' does not observe setting of PG_lru by '#0' and fails
29          * isolation, the explicit barrier will make sure that page_evictable
30          * check will put the page in correct LRU. Without smp_mb(), SetPageLRU
31          * can be reordered after PageMlocked check and can make '#1' to fail
32          * the isolation of the page whose Mlocked bit is cleared (#0 is also
33          * looking at the same page) and the evictable page will be stranded
34          * in an unevictable LRU.
35          */
36         smp_mb();
37
38         if (page_evictable(page)) {
39                 lru = page_lru(page);
40                 update_page_reclaim_stat(lruvec, page_is_file_cache(page),
41                                          PageActive(page));
```

```
42                  if (was_unevictable)
43                          count_vm_event(UNEVICTABLE_PGRESCUED);
44          } else {
45                  lru = LRU_UNEVICTABLE;
46                  ClearPageActive(page);
47                  SetPageUnevictable(page);
48                  if (!was_unevictable)
49                          count_vm_event(UNEVICTABLE_PGCULLED);
50          }
51
52          add_page_to_lru_list(page, lruvec, lru);
53          trace_mm_lru_insertion(page, lru);
54  }
```

Add a page to the list of the appropriate types (inactive_anon, active_anon, inactive_file, active_file, unevictable) for the specified @lruvec. The LRU flag bit is set on the page to indicate that it belongs to the LRU list.

- In line 5 of the code, check to see if the page was in the unevictable list and clear the flag.
- In line 9 of the code, indicate that the page belongs to the LRU list.
- See the comments above for an explanation of the case where the order of memory access needs to be clarified in line 36 of code.
- On lines 38~43 of code, if the page is in a reclaimable state, select the LRU list and increment the reclaim-related scanned[] and rocated[] entries. If it was previously unevictable, increment the UNEVICTABLE_PGRESCUED counter.
- In line 44~50 of the code, if the page is not in a reclaimable state, select the list of unevectable LRUs, clear the active flag, and set the unevictable flag. If it was previously evictable, increment the UNEVICTABLE_PGCULLED counter.
- In line 52 of code, add a page to lruvec.

pagevec_move_tail_fn()

mm/swap.c

```
01  static void pagevec_move_tail_fn(struct page *page, struct lruvec *lruve
    c,
02                                    void *arg)
03  {
04          int *pgmoved = arg;
05
06          if (PageLRU(page) && !PageUnevictable(page)) {
07                  del_page_from_lru_list(page, lruvec, page_lru(page));
08                  ClearPageActive(page);
09                  add_page_to_lru_list_tail(page, lruvec, page_lru(page));
10                  (*pgmoved)++;
11          }
12  }
```

If the page is of type LRU rather than unevictable, add the page to the back (cold) of the list. and remove the active flag.

- If the page is set to the LRU flag in line 6 and is not in the unevitable flag state, the page is removed from the existing LRU list.
- Remove the active flag from the page in lines 7~8 and add the page to the back of LRU's type-specific list.

- Increment the counter passed in as the last argument in line 9.

lru_deactivate_file_fn()

mm/swap.c

```
01 /*
02  * If the page can not be invalidated, it is moved to the
03  * inactive list to speed up its reclaim.  It is moved to the
04  * head of the list, rather than the tail, to give the flusher
05  * threads some time to write it out, as this is much more
06  * effective than the single-page writeout from reclaim.
07  *
08  * If the page isn't page_mapped and dirty/writeback, the page
09  * could reclaim asap using PG_reclaim.
10  *
11  * 1. active, mapped page -> none
12  * 2. active, dirty/writeback page -> inactive, head, PG_reclaim
13  * 3. inactive, mapped page -> none
14  * 4. inactive, dirty/writeback page -> inactive, head, PG_reclaim
15  * 5. inactive, clean -> inactive, tail
16  * 6. Others -> none
17  *
18  * In 4, why it moves inactive's head, the VM expects the page would
19  * be write it out by flusher threads as this is much more effective
20  * than the single-page writeout from reclaim.
21  */
```

```
01 static void lru_deactivate_file_fn(struct page *page, struct lruvec *lru
   vec,
02                                    void *arg)
03 {
04         int lru, file;
05         bool active;
06
07         if (!PageLRU(page))
08                 return;
09
10         if (PageUnevictable(page))
11                 return;
12
13         /* Some processes are using the page */
14         if (page_mapped(page))
15                 return;
16
17         active = PageActive(page);
18         file = page_is_file_cache(page);
19         lru = page_lru_base_type(page);
20
21         del_page_from_lru_list(page, lruvec, lru + active);
22         ClearPageActive(page);
23         ClearPageReferenced(page);
24         add_page_to_lru_list(page, lruvec, lru);
25
26         if (PageWriteback(page) || PageDirty(page)) {
27                 /*
28                  * PG_reclaim could be raced with end_page_writeback
29                  * It can make readahead confusing.  But race window
30                  * is _really_ small and  it's non-critical problem.
31                  */
32                 SetPageReclaim(page);
33         } else {
34                 /*
35                  * The page's writeback ends up during pagevec
36                  * We moves tha page into tail of inactive.
```

```
37                  */
38                 list_move_tail(&page->lru, &lruvec->lists[lru]);
39                 __count_vm_event(PGROTATED);
40         }
41
42         if (active)
43                 __count_vm_event(PGDEACTIVATE);
44         update_page_reclaim_stat(lruvec, file, 0);
45 }
```

If the page is of type LRU but is not unevictable and is not a mapped file, delete the page from the list of LRU types and add the page to the beginning of LRU's base type. The page flag removes the active and referenced flags. If the page has a history attribute, set the reclaim flag and move it to the back of the list if it doesn't.

- In line 7~8 of the code, if the page does not have the LRU flag set, it exits without proceeding any further.
- If the Unevitable flag is set on the page in line 10~11 of the code, it will exit without proceeding any further.
- In line 14~15 of the code, if the page is already mapped and in use by the process, it exits without proceeding any further.
- In line 17 of the code, we find out whether the page has an active flag status.
- In line 18 of the code, we find out whether the page is cached from the file.
- Retrieve the LRU base type from the page at line 19 of code.
    - Returns a LRU_INACTIVE_FILE or LRU_INACTIVE_ANON type.
- In line 21 of the code, find the page in the LRU list in the LRU + Active array and delete it.
- In lines 22~24 of the code, delete the Active flag and the Referencewd flag from the page, and then add the page to the LRU list in the LRU base type array.
- In lines 26~32 of the code, if the page is set to Writeback or Dirty, set the Reclaim flag.
- In line 33~40 of the code, if not, add a page to the end of the LRU list in the LRU type array. Then increment the PGROTATED counter.
        - If added to the afterword, it indicates that it is used most frequently as a cold page.
- In line 42~43 of code, increment the vm_event of the PGDEACTIVATE entry if it is active.
- On line 44 of code, increment the reclaim-related scanned[] and rocated[] entries

lru_lazyfree_fn()

mm/swap.c

```
01 static void lru_lazyfree_fn(struct page *page, struct lruvec *lruvec,
02                              void *arg)
03 {
04         if (PageLRU(page) && PageAnon(page) && PageSwapBacked(page) &&
05             !PageSwapCache(page) && !PageUnevictable(page)) {
06                 bool active = PageActive(page);
07
08                 del_page_from_lru_list(page, lruvec,
09                                        LRU_INACTIVE_ANON + active);
10                 ClearPageActive(page);
11                 ClearPageReferenced(page);
12                 /*
13                  * lazyfree pages are clean anonymous pages. They have
14                  * SwapBacked flag cleared to distinguish normal anonymo
us
```

```
15                  * pages
16                  */
17                 ClearPageSwapBacked(page);
18                 add_page_to_lru_list(page, lruvec, LRU_INACTIVE_FILE);
19
20                 __count_vm_events(PGLAZYFREE, hpage_nr_pages(page));
21                 count_memcg_page_event(page, PGLAZYFREE);
22                 update_page_reclaim_stat(lruvec, 1, 0);
23         }
24 }
```

Replace a normal anon page with a swap area with a clean anon page that does not have a swap area and add it to the hot list of inactive file LRU.

- In line 4~9 of the code, if it is a normal anon page with a swap area and is not in a swap cached state, remove it from the lruvec list.
- On lines 10~18 of the code, clear the Active, Referenced, and SwapBacked flags on the page and add them to the LRU list.
- On line 20 of code, increment the PGLAZYFREE vm counter by the number of pages.
- In line 21 of the code, increment the PGLAZYFREE counter in memcg.
- On line 22 of code, increment the reclaim-related scanned[] and rocated[] entries

__activate_page()

mm/swap.c

```
01 static void __activate_page(struct page *page, struct lruvec *lruvec,
02                             void *arg)
03 {
04         if (PageLRU(page) && !PageActive(page) && !PageUnevictable(pag
   e)) {
05                 int file = page_is_file_cache(page);
06                 int lru = page_lru_base_type(page);
07
08                 del_page_from_lru_list(page, lruvec, lru);
09                 SetPageActive(page);
10                 lru += LRU_ACTIVE;
11                 add_page_to_lru_list(page, lruvec, lru);
12                 trace_mm_lru_activate(page);
13
14                 __count_vm_event(PGACTIVATE);
15                 update_page_reclaim_stat(lruvec, file, 1);
16         }
17 }
```

Remove the page from lruvec->lists[basic type], set the active flag, and add it to the hot of lruvec->lists[lru+active].

- If LRU is set on the page in line 4~8 and it is inactive and does not have an unevictable flag set, it will be removed from the list of LRU of that LRU type.
- In line 9~11 of the code, set the page to active, and add the page to the beginning of the list of active LRU of that type (file or anon).
- In line 14 of the code, increment the counter of the vm_event's PGACTIVATE entry.
- On line 15 of code, increment the reclaim-related scanned[] and rocated[] entries

guitar

page_evictable()

mm/vmscan.c

```
/*
 * page_evictable - test whether a page is evictable
 * @page: the page to test
 *
 * Test whether page is evictable--i.e., should be placed on active/inac
tive
 * lists vs unevictable list.
 *
 * Reasons page might not be evictable:
 * (1) page's mapping marked unevictable
 * (2) page is part of an mlocked VMA
 *
 */
```

```
int page_evictable(struct page *page)
{
        int ret;

        /* Prevent address_space of inode and swap cache from being free
d */
        rcu_read_lock();
        ret = !mapping_unevictable(page_mapping(page)) && !PageMlocked(p
age);
        rcu_read_unlock();
        return ret;
}
```

Returns whether the page is in an evictable state.

- If the page is already mapped or is not in the mlock state, it is in the evicatable state.

page_is_file_cache()

include/linux/mm_inline.h

```
/**
 * page_is_file_cache - should the page be on a file LRU or anon LRU?
 * @page: the page to test
 *
 * Returns 1 if @page is page cache page backed by a regular filesystem,
 * or 0 if @page is anonymous, tmpfs or otherwise ram or swap backed.
 * Used by functions that manipulate the LRU lists, to sort a page
 * onto the right LRU list.
 *
 * We would like to get this info without a page flag, but the state
 * needs to survive until the page is last deleted from the LRU, which
 * could be as far down as __page_cache_release.
 */
```

```
static inline int page_is_file_cache(struct page *page)
{
        return !PageSwapBacked(page);
}
```

Returns whether the page is in File LRU or ANON LRU.

- 1: file belongs to LRU.

    - File cache page or clean anon page that does not have swap area
- 0: Belongs to Anon LRU.
    - normal anon page or tmpfs with swap zone

page_lru()

include/linux/mm_inline.h

```
1  /**
2   * page_lru - which LRU list should a page be on?
3   * @page: the page to test
4   *
5   * Returns the LRU list a page should be on, as an index
6   * into the array of LRU lists.
7   */
```

```
01 static __always_inline enum lru_list page_lru(struct page *page)
02 {
03         enum lru_list lru;
04
05         if (PageUnevictable(page))
06                 lru = LRU_UNEVICTABLE;
07         else {
08                 lru = page_lru_base_type(page);
09                 if (PageActive(page))
10                         lru += LRU_ACTIVE;
11         }
12         return lru;
13 }
```

Get the LRU (5 states) values for the page.

- If the page has an unevictable flag in lines 5~6, return LRU_UNEVICTABLE(4).
- In line 7~8 of the code, we get LRU_INACTIVE_FILE(2) if the page is of a cached file, and LRU_INACTIVE_ANON(0) if it is not.
- On lines 9~10 of the code, clear if the page is active, and add LRU_ACTIVE(1) to LRU.
    - LRU_INACTIVE_FILE(2) -> LRU_ACTIVE_FILE(3)
    - LRU_INACTIVE_ANON(0) -> LRU_ACTIVE_ANON(1)

add_page_to_lru_list()

include/linux/mm_inline.h

```
1 static __always_inline void add_page_to_lru_list(struct page *page,
2                                 struct lruvec *lruvec, enum lru_list lr
  u)
3 {
4         update_lru_size(lruvec, lru, page_zonenum(page), hpage_nr_pages
  (page));
5         list_add(&page->lru, &lruvec->lists[lru]);
6 }
```

Add the page to the LRU list.

- Update LRU-related stats on line 4 of code.
    - If the page is huge, know the number of small pages. If not, it is 1.
        - huge -2 pages > 512MB

- In line 5 of the code, add the page to the list by type in LRU at the front. Adding to the top means a hot page that is used frequently.

update_lru_size()

include/linux/mm_inline.h

```
1 static __always_inline void update_lru_size(struct lruvec *lruvec,
2                                 enum lru_list lru, enum zone_type zid,
3                                 int nr_pages)
4 {
5         __update_lru_size(lruvec, lru, zid, nr_pages);
6 #ifdef CONFIG_MEMCG
7         mem_cgroup_update_lru_size(lruvec, lru, zid, nr_pages);
8 #endif
9 }
```

- In line 5 of code, add the number of pages to the VM counter that corresponds to the LRU type of the node's and zone's pages.
- In line 7 of code, add the number of pages to the lru_size[lru] of the memory cgroup.

__update_lru_size()

include/linux/mm_inline.h

```
01 static __always_inline void __update_lru_size(struct lruvec *lruvec,
02                                 enum lru_list lru, enum zone_type zid,
03                                 int nr_pages)
04 {
05         struct pglist_data *pgdat = lruvec_pgdat(lruvec);
06
07         __mod_node_page_state(pgdat, NR_LRU_BASE + lru, nr_pages);
08         __mod_zone_page_state(&pgdat->node_zones[zid],
09                                 NR_ZONE_LRU_BASE + lru, nr_pages);
10 }
```

Add the number of pages to the VM counter that corresponds to the LRU type of the node's and zone's pages.

- In line 7 of code, add the number of pages to the VM counter that corresponds to the LRU type of the node's pages.
- In lines 8~9 of the code, add the number of pages to the VM counter that corresponds to the LRU type of John's page.

mem_cgroup_update_lru_size()

mm/memcontrol.c

```
01 /**
02  * mem_cgroup_update_lru_size - account for adding or removing an lru pa
   ge
03  * @lruvec: mem_cgroup per zone lru vector
04  * @lru: index of lru list the page is sitting on
05  * @zid: zone id of the accounted pages
06  * @nr_pages: positive when adding or negative when removing
07  *
```

```
 * This function must be called under lru_lock, just before a page is ad
ded
 * to or just after a page is removed from an lru list (that ordering be
ing
 * so as to allow it to check that lru_size 0 is consistent with list_em
pty).
 */
```

```
void mem_cgroup_update_lru_size(struct lruvec *lruvec, enum lru_list lr
u,
                                int zid, int nr_pages)
{
        struct mem_cgroup_per_node *mz;
        unsigned long *lru_size;
        long size;

        if (mem_cgroup_disabled())
                return;

        mz = container_of(lruvec, struct mem_cgroup_per_node, lruvec);
        lru_size = &mz->lru_zone_size[zid][lru];

        if (nr_pages < 0)
                *lru_size += nr_pages;

        size = *lru_size;
        if (WARN_ONCE(size < 0,
                "%s(%p, %d, %d): lru_size %ld\n",
                __func__, lruvec, lru, nr_pages, size)) {
                VM_BUG_ON(1);
                *lru_size = 0;
        }

        if (nr_pages > 0)
                *lru_size += nr_pages;
}
```

Add the number of pages to the node-specific lru_size [LRU] of the memory cgroup.

update_page_reclaim_stat()

mm/swap.c

```
static void update_page_reclaim_stat(struct lruvec *lruvec,
                                     int file, int rotated)
{
        struct zone_reclaim_stat *reclaim_stat = &lruvec->reclaim_stat;

        reclaim_stat->recent_scanned[file]++;
        if (rotated)
                reclaim_stat->recent_rotated[file]++;
}
```

Increment the scanned[] and rocated[] entries related to reclaim. Each of the two entries uses two arrays, each of which looks like this:

- [0]: anon LRU stat
- [1]: file LRU stat

Return to LRU List

putback_movable_pages()

mm/migrate.c

```
01 /*
02  * Put previously isolated pages back onto the appropriate lists
03  * from where they were once taken off for compaction/migration.
04  *
05  * This function shall be used whenever the isolated pageset has been
06  * built from lru, balloon, hugetlbfs page. See isolate_migratepages_ran
    ge()
07  * and isolate_huge_page().
08  */
09 void putback_movable_pages(struct list_head *l)
10 {
11         struct page *page;
12         struct page *page2;
13
14         list_for_each_entry_safe(page, page2, l, lru) {
15                 if (unlikely(PageHuge(page))) {
16                         putback_active_hugepage(page);
17                         continue;
18                 }
19                 list_del(&page->lru);
20                 dec_zone_page_state(page, NR_ISOLATED_ANON +
21                                     page_is_file_cache(page));
22                 if (unlikely(isolated_balloon_page(page)))
23                         balloon_page_putback(page);
24                 else
25                         putback_lru_page(page);
26         }
27 }
```

Returns previously isolated pages back to their original location.

- list_for_each_entry_safe(page, page2, l, lru) {
  - Loop around the pages on the list.
- if (unlikely(PageHuge(page))) { putback_active_hugepage(page); continue; }
  - If it's a huge page with a small probability, move it to the back of hstate[].hugepage_activelist and skip it.
    - Huge pages are managed by hstate[].
- dec_zone_page_state(page, NR_ISOLATED_ANON + page_is_file_cache(page));
  - Depending on the type of page, it will reduce the NR_ISOLATE_ANON or NR_ISOLATED_FILE stat.
- if (unlikely(isolated_balloon_page(page))) balloon_page_putback(page);
  - If it's a balloon page, there is a small chance that it will be returned to the balloon_dev_info's pages list.
    - The balloon page is managed by the balloon device.
- else putback_lru_page(page);
  - Revert the page to lurvec.lists[].

putback_lru_page()

mm/vmscan.c

```
/**
 * putback_lru_page - put previously isolated page onto appropriate LRU
list
 * @page: page to be put back to appropriate lru list
 *
 * Add previously isolated @page to appropriate LRU list.
 * Page may still be unevictable for other reasons.
 *
 * lru_lock must not be held, interrupts must be enabled.
 */
void putback_lru_page(struct page *page)
{
        bool is_unevictable;
        int was_unevictable = PageUnevictable(page);

        VM_BUG_ON_PAGE(PageLRU(page), page);

redo:
        ClearPageUnevictable(page);

        if (page_evictable(page)) {
                /*
                 * For evictable pages, we can use the cache.
                 * In event of a race, worst case is we end up with an
                 * unevictable page on [in]active list.
                 * We know how to handle that.
                 */
                is_unevictable = false;
                lru_cache_add(page);
        } else {
                /*
                 * Put unevictable pages directly on zone's unevictable
                 * list.
                 */
                is_unevictable = true;
                add_page_to_unevictable_list(page);
                /*
                 * When racing with an mlock or AS_UNEVICTABLE clearing
                 * (page is unlocked) make sure that if the other thread
                 * does not observe our setting of PG_lru and fails
                 * isolation/check_move_unevictable_pages,
                 * we see PG_mlocked/AS_UNEVICTABLE cleared below and mo
ve
                 * the page back to the evictable list.
                 *
                 * The other side is TestClearPageMlocked() or shmem_loc
k().
                 */
                smp_mb();
        }

        /*
         * page's status can change while we move it among lru. If an ev
ictable
         * page is on unevictable list, it never be freed. To avoid tha
t,
         * check after we added it to the list, again.
         */
        if (is_unevictable && page_evictable(page)) {
                if (!isolate_lru_page(page)) {
                        put_page(page);
                        goto redo;
                }
                /* This means someone else dropped this page from LRU
                 * So, it will be freed or putback to LRU again. There i
s
                 * nothing to do here.
                 */
```

```
63          }
64
65          if (was_unevictable && !is_unevictable)
66                  count_vm_event(UNEVICTABLE_PGRESCUED);
67          else if (!was_unevictable && is_unevictable)
68                  count_vm_event(UNEVICTABLE_PGCULLED);
69
70          put_page(page);         /* drop ref from isolate */
71  }
```

Revert the isolated page back to lruvec.

- int was_unevictable = PageUnevictable(page);
  - Determines whether the page is unevictable.
- ClearPageUnevictable(page);
  - Clear the PG_unevictable flag on the page.
- if (page_evictable(page)) { is_unevictable = false; lru_cache_add(page);
  - If the page mapping state is evictable, it will put false in the is_unevictable and register the page in the lru_add_pvec cache.
- } else { is_unevictable = true; add_page_to_unevictable_list(page); smp_mb(); }
  - Add the page to lruvec.list[LRU_UNEVICTABLE].
- if (is_unevictable && page_evictable(page)) { if (!isolate_lru_page(page)) { put_page(page); goto redo; } }
  - If a page added to lruvec.list[LRU_UNEVICTABLE] is changed to an evictable state, it will never be free. To avoid this, repeat this page once again to isolate and check it.
- if (was_unevictable && !is_unevictable) count_vm_event(UNEVICTABLE_PGRESCUED);
  - If it was unevictable and is not now unevictable, it increments UNEVICTABLE_PGRESCUED stat.
- else if (!was_unevictable && is_unevictable) count_vm_event(UNEVICTABLE_PGCULLED);
  - If it wasn't unevictable and is now unevictable, it UNEVICTABLE_PG increments the CULLED stat.
- put_page(page);
  - Clear the LRU bit flag on the page, remove it from the LRU list, and free the page to the buddy system in the hot direction.

Huge Page & Huge TLB

- It can only be used on architectures that support Huge TLB.
  - It is available for x86, ia64, arm with LPAE, sparc64, s390, and more.
  - Note: hugetlbpage.txt (http://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt) | kernel.org
- If you are using Huge TLB, you can load a large page as a single TLB entry, which reduces the overhead on mapping and maintains fast access performance.
- When using Huge TLB, page blocks can be operated on a HugeTLB basis rather than a MAX_ORDER-1 page unit to improve the performance of TLB H/W.
- The global hstate[] is composed of arrays that can be used to construct multiple TLB entries of different sizes.

    - See: Hugetlb: multiple hstates for multiple page sizes (https://github.com/torvalds/linux/commit/e5ff215941d59f8ae6bf58f6428dc5c26745a612)
- Kernel parameters are used to reserve a specified amount of space.
    - e.g. "default_hugepagesz=1G hugepagesz=1G"
- Changing settings at runtime
    - "/proc/sys/vm/nr_hugepages" and "/sys/devices/system/node/node_id/hugepages/hugepages" on NUMA systems.
- When you open and create shared memory, you can use the SHM_HUGETLB option to use huge tlb.
    - e.g. shmid = shmget(2, LENGTH, SHM_HUGETLB | IPC_CREAT | SHM_R | SHM_W)) < 0)

HugeTLBFS

- It behaves like a file system, so you can mount it and use it.
    - e.g. mount -t hugetlbfs -o uid=<value>,gid=<value>,mode=<value>,size=<value>,nr_inodes=<value> none /mnt/huge
- Files created in the mounted directory (/mnt/huge) are mapped using huge tlb.

putback_active_hugepage()

mm/hugetlb.c

```
1  void putback_active_hugepage(struct page *page)
2  {
3          VM_BUG_ON_PAGE(!PageHead(page), page);
4          spin_lock(&hugetlb_lock);
5          list_move_tail(&page->lru, &(page_hstate(page))->hugepage_active
   list);
6          spin_unlock(&hugetlb_lock);
7          put_page(page);
8  }
```

Puts the isolated page back at the end of the global hstate[] hugepage_activelist.

- Increases the reference counter when isolation is decremented.

Managing Balloon Pages

- Linux provides balloon device drivers for virtual machines such as KVM and XEN.
- Balloon memory compaction is supported to prevent memory fragmentation.

balloon_page_putback()

mm/balloon_compaction.c

```
01 /* putback_lru_page() counterpart for a ballooned page */
02 void balloon_page_putback(struct page *page)
03 {
04         /*
```

```
05          * 'lock_page()' stabilizes the page and prevents races against
06          * concurrent isolation threads attempting to re-isolate it.
07          */
08         lock_page(page);
09
10         if (__is_movable_balloon_page(page)) {
11                 __putback_balloon_page(page);
12                 /* drop the extra ref count taken for page isolation */
13                 put_page(page);
14         } else {
15                 WARN_ON(1);
16                 dump_page(page, "not movable balloon page");
17         }
18         unlock_page(page);
19 }
```

If the isolated page is a ballon page, it is returned to the Ballon device's pages list where the page was recorded.

- Increases the reference counter when isolation is decremented.

__is_movable_balloon_page()

include/linux/balloon_compaction.h

```
1  /*
2   * __is_movable_balloon_page - helper to perform @page PageBalloon tests
3   */
4  static inline bool __is_movable_balloon_page(struct page *page)
5  {
6          return PageBalloon(page);
7  }
```

Ballon returns whether the page is or not.

__putback_balloon_page()

mm/balloon_compaction.c

```
01 static inline void __putback_balloon_page(struct page *page)
02 {
03         struct balloon_dev_info *b_dev_info = balloon_page_device(page);
04         unsigned long flags;
05
06         spin_lock_irqsave(&b_dev_info->pages_lock, flags);
07         SetPagePrivate(page);
08         list_add(&page->lru, &b_dev_info->pages);
09         b_dev_info->isolated_pages--;
10         spin_unlock_irqrestore(&b_dev_info->pages_lock, flags);
11 }
```

Set the PG_private flag on the page and revert the ballon page recorded to the device's Pages list.

balloon_page_device()

include/linux/balloon_compaction.h

```
1  /*
```

```
 * balloon_page_device - get the b_dev_info descriptor for the balloon device
 *                       that enqueues the given page.
 */
static inline struct balloon_dev_info *balloon_page_device(struct page *page)
{
        return (struct balloon_dev_info *)page_private(page);
}
```

Recognize the ballon page device.

Structure

pagevec structure

```
struct pagevec {
        unsigned long nr;
        boool percpu_pvec_drained;
        struct page *pages[PAGEVEC_SIZE];
};
```

- nr
  - The number of pages managed by PageVec
- percpu_pvec_drained
  - Drain or not
- *pages[]
  - These are pages managed by pagevec. (up to 15)

lruvec struct

include/linux/mmzone.h

```
struct lruvec {
        struct list_head                lists[NR_LRU_LISTS];
        struct zone_reclaim_stat        reclaim_stat;
        /* Evictions & activations on the inactive file list */
        atomic_long_t                   inactive_age;
        /* Refaults at the time of last reclaim cycle */
        unsigned long                   refaults;
#ifdef CONFIG_MEMCG
        struct pglist_data *pgdat;
#endif
};
```

- lists[]
  - Here is a list of 5 lruvec.
- reclaim_stat
  - Reclaim-related stat
- inactive_age
- refaults
- *pgdat
  - Node.
  - When using the Memory Control cGroup, LRUVEC is managed on a per-node basis.

zone_reclaim_stat Struct

include/linux/mmzone.h

```
struct zone_reclaim_stat {
        /*
         * The pageout code in vmscan.c keeps track of how many of the
         * mem/swap backed and file backed pages are referenced.
         * The higher the rotated/scanned ratio, the more valuable
         * that cache is.
         *
         * The anon LRU stats live in [0], file LRU stats in [1]
         */
        unsigned long           recent_rotated[2];
        unsigned long           recent_scanned[2];
};
```

lru_list

include/linux/mmzone.h

```
/*
 * We do arithmetic on the LRU lists in various places in the code,
 * so it is important to keep the active lists LRU_ACTIVE higher in
 * the array than the corresponding inactive lists, and to keep
 * the *_FILE lists LRU_FILE higher than the corresponding _ANON lists.
 *
 * This has to be kept in sync with the statistics in zone_stat_item
 * above and the descriptions in vmstat_text in mm/vmstat.c
 */
#define LRU_BASE 0
#define LRU_ACTIVE 1
#define LRU_FILE 2

enum lru_list {
        LRU_INACTIVE_ANON = LRU_BASE,
        LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
        LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
        LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
        LRU_UNEVICTABLE,
        NR_LRU_LISTS
};
```

Global Pagevec Cache

mm/swap.c

```
static DEFINE_PER_CPU(struct pagevec, lru_add_pvec);
static DEFINE_PER_CPU(struct pagevec, lru_rotate_pvecs);
static DEFINE_PER_CPU(struct pagevec, lru_deactivate_file_pvecs);
static DEFINE_PER_CPU(struct pagevec, lru_lazyfree_pvecs);
#ifdef CONFIG_SMP
static DEFINE_PER_CPU(struct pagevec, activate_page_pvecs);
#endif
```

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) | Qc
- Zoned Allocator -6- (Watermark) (http://jake.dothome.co.kr/zonned-allocator-watermark) | Qc
- Zoned Allocator -7- (Direct Compact) (http://jake.dothome.co.kr/zonned-allocator-compaction) | Qc
- Zoned Allocator -8- (Direct Compact-Isolation) (http://jake.dothome.co.kr/zonned-allocator-isolation) | Qc
- Zoned Allocator -9- (Direct Compact-Migration) (http://jake.dothome.co.kr/zonned-allocator-migration) | Qc
- Zoned Allocator -10- (LRU & pagevec) (http://jake.dothome.co.kr/lru-lists-pagevecs) | Sentence C – Current post
- Zoned Allocator -11- (Direct Reclaim) (http://jake.dothome.co.kr/zonned-allocator-reclaim) | Qc
- Zoned Allocator -12- (Direct Reclaim-Shrink-1) (http://jake.dothome.co.kr/zonned-allocator-shrink-1) | Qc
- Zoned Allocator -13- (Direct Reclaim-Shrink-2) (http://jake.dothome.co.kr/zonned-allocator-shrink-2) | Qc
- Zoned Allocator -14- (Kswapd) (http://jake.dothome.co.kr/zonned-allocator-kswapd) | Qc

- Lifetime of memory with pageflags (http://egloos.zum.com/studyfoss/v/5512112) | F/OSS
- Linux Memory Allocation | Comumbia Edu. – PDF Download (https://www.cs.columbia.edu/~smb/classes/s06-4118/l19.pdf)
- PageReplacementDesign (https://linux-mm.org/PageReplacementDesign) | linux-mm.org
- UNEVICTABLE LRU INFRASTRUCTURE (http://www.kernel.org/doc/Documentation/vm/unevictable-lru.txt) | kernel.org
- Overview of Memory Reclaim in the Current Upstream Kernel (2021) | SUSE – Download PDF (https://lpc.events/event/11/contributions/896/attachments/793/1493/slides-r2.pdf)

---

10 thoughts to “Zoned Allocator -10- (LRU & pagevecs)”

PARAN LEE (HTTPS://PARANLEE.GITHUB.IO/)
2021-03-06 17:03 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-304845)

I appreciate it.

I'm always looking at it when I'm studying!

Balloon page management

Linux is a virtual machine like KVM and GEN

-> GEN XEN

Is this a typo?

RESPONSE (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=304845#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-03-08 06:44 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-304866)

I appreciate it. GEN -> XEN Typo fixed. ^^

RESPONSE (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=304866#RESPOND)

DDTUX
2021-09-02 18:47 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-305979)

Hi,
I'm always looking at it well when looking for kernel related stuff.
While studying about the page reclamation process, I got curious and wrote about it.
In the process of retrieving the page, I didn't do a good job of researching the priority between the Anonymous page and the File-backed page, so I posted this on the bulletin board.
I only know that the value of vm.swapiness has an effect, but if you know the details, can you explain it or explain the relevant code?

In my experimental testing, if my application was holding memory with malloc(), it would not swap and the amount of pagecache (buff/cache) would no longer increase.

I appreciate it.

RESPONSE (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=305979#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-09-03 13:27 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-305983)

Hello?

메모리 회수는 커널에서 복잡하고 방대한 루틴입니다.
아래 제 글들이 관련된 글이며, 그 중 12번 항목이 vm.swapiness 쪽 루틴을 다루고 있습니다.
http://jake.dothome.co.kr/lru-lists-pagevecs (http://jake.dothome.co.kr/lru-lists-pagevecs)

Zoned Allocator -6- (Watermark) | 문c
Zoned Allocator -7- (Direct Compact) | 문c
Zoned Allocator -8- (Direct Compact-Isolation) | 문c

Zoned Allocator -9- (Direct Compact-Migration) | 문c
Zoned Allocator -10- (LRU & pagevec) | 문c
Zoned Allocator -11- (Direct Reclaim) | 문c
Zoned Allocator -12- (Direct Reclaim-Shrink-1) | 문c – 현재 글
Zoned Allocator -13- (Direct Reclaim-Shrink-2) | 문c
Zoned Allocator -14- (Kswapd) | 문c

참고로 메모리가 부족한 상황에서 swap이 일어나잖아요?
이 때 swap 디바이스가 지정되어 있어야 하고, swapness도 설정이 되어 있어야 합니다.
또한 malloc()으로 할당 받은 메모리를 반복하여 액세스하면 swap 하지 않으므로,
한 번 액세스한 이후에 방치해두어야 합니다. ^^

감사합니다.

응답 (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=305983#RESPOND)

양원혁 (HTTPS://WWW.BHRAL.COM/)
2022-07-16 20:41 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-306851)

안녕하세요, 글을 통해 항상 많은 것을 배우고 갑니다.

pvec 파트를 읽다가 참고 링크(mm: move MADV_FREE pages into LRU_INACTIVE_FILE list)가 다른 커밋 url로 연결되는 것 같아 댓글을 남깁니다.

감사합니다.

응답 (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=306851#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2022-07-16 22:18 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-306852)

양원혁님, 안녕하세요?

재빠르게 본문의 링크를 수정하였습니다.

감사합니다. 즐거운 주말되시기 바랍니다. ^^

응답 (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=306852#RESPOND)

DORU
2023-09-15 06:45 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-308357)

안녕하세요, 항상 좋은 글 감사합니다.
pagevec과 lruvec관련 궁금한 점이 있어서 댓글을 남깁니다.

pagevec에 있던 페이지들이 lruvec으로 drain이 되고 난 후에
다시 그 페이지들에 대한 접근이 발생하면,
그때부터는 lruvec에 spin lock을 건 후 해당 페이지들에 대한 상태를 업데이트 해주는 게 맞나요?
lruvec에 있던 페이지들이 다시 pagevec으로 캐시 되는 일은 없는지 궁금합니다.

감사합니다.

응답 (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=308357#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2023-09-18 09:43 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-308368)

아시는 바와 같이 페이지에 접근이 발생한 경우 active라는 플래그를 설정하면되는데 이미 설정되어 있는 경우 무시합니다.
그런데 아직 설정되어 있지 않은 경우 LRU에 있든, lru_add pagevec에 있든 active 페이지로 플래그를 설정합니다.
질문과 같이 LRU로 drain되었으면 spin-lock걸고 LRU에 접근해서 activate를 곧장 찍는 것이 아니라,
다시 active LRU 리스트의 선두로 옮기는 과정을 거치는데, 이 과정에서 local-lock을 걸고 lru_add pagevec을 통해서 옮깁니다.
결국 lruvec에 있던 페이지들에 접근이 발생되는 경우 pagevec으로 캐시되는 것입니다.

감사합니다.

응답 (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=308368#RESPOND)

DORU
2023-09-15 06:49 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-308358)

또다른 질문이 생겨 댓글을 추가로 남깁니다.

pagevec은 per-core별로 가지고 있는데
어떤 페이지가 1번 core의 pagevec에 들어있는 상황에서
2번 core에서 해당 페이지에 접근이 발생이 되면 어떻게 되나요?

2번 core의 pagevec에도 해당 페이지가 들어가게 되는지,
아니면 1번 core의 pagevec이 lruvec으로 drain이 되어야하는지 궁금합니다.

응답 (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=308358#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2023-09-18 09:46 (http://jake.dothome.co.kr/lru-lists-pagevecs/#comment-308369)

1번 코어에서 해당 페이지에 접근하여 pagevec에 이미 추가하였으므로,
2번 코어의 경우 PageActive가 이미 설정되어 있는 상태이므로 이 페이지에 다시 접근하여 또 active 플래그를 설정할 필요가 없게 됩니다.

감사합니다.

응답 (/LRU-LISTS-PAGEVECS/?REPLYTOCOM=308369#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

‹ Zoned Allocator -5- (Per-CPU Page Frame Cache) (http://jake.dothome.co.kr/per-cpu-page-frame-cache/)

page_alloc_init() › (http://jake.dothome.co.kr/page_alloc_init/)

문c 블로그 (2015 ~ 2023)