Zoned Allocator -2- (Physics Page Assignment - Slowpath)

2016-06-29 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-slowpath/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Slowpath

If fastpath allocation fails in the zonelist according to the NUMA memory policy, proceed with the slowpath step. If the nofail option is used, it repeats until the allocation succeeds. During the Slowpath phase, if there are not enough free pages, the following retrieval actions are performed according to the request option:

- direct-compaction
  - When allocating pages, if the requested order cannot be allocated due to insufficient pages, the direct compaction action is performed to secure the pages and then allocate them.
- direct-reclaim
  - When allocating a page, if it is not possible to allocate the pages due to insufficient pages in the requested order, the direct reclaim operation is performed to secure the pages and then allocate them.
  - If you perform a reclaim action, you can use the
- OOM killing
  - When allocating pages, there are not enough pages for the requested order, and the specific task is finally terminated through OOM killing, so it is assigned to the pages that have been secured.
- kswapd
  - It operates a page reclaim mechanism in the background to record dirty file caches, empty clean file caches, and move pages to the swap system to secure free pages.
- kcompactd
  - Perform a compaction operation in the background to move fragmented movable pages and merge free pages to get larger order free pages.

OOM(Out of Memory) Killing

If you're running out of memory and both compaction or reclaim page retrieval fails and you can't proceed any further, you need to kill one of the following specific tasks:

- 0 rank if the current task is terminating
- Assigned task 1 rank to be processed first in OOM state
- Among the tasks, one task that is above the certain calculation standard ranks 2

The following shows the results of forcing OOM killing.

```
$ echo f > /proc/sysrq-trigger
$ dmesg
[460767.036092] sysrq: SysRq : Manual OOM execution
[460767.037248] kworker/0:0 invoked oom-killer: gfp_mask=0x24000c0, order=-1, oom_s
core_adj=0
[460767.038016] kworker/0:0 cpuset=/ mems_allowed=0
[460767.038468] CPU: 0 PID: 8063 Comm: kworker/0:0 Tainted: G        W       4.4.10
3-g94108fb3583f-dirty #4
[460767.039307] Hardware name: ROCK960 - 96boards based on Rockchip RK3399 (DT)
[460767.039948] Workqueue: events moom_callback
[460767.040348] Call trace:
[460767.040603] [<ffffff800808806c>] dump_backtrace+0x0/0x21c
[460767.041104] [<ffffff80080882ac>] show_stack+0x24/0x30
[460767.041583] [<ffffff80083b56f4>] dump_stack+0x94/0xbc
[460767.042064] [<ffffff80081bdd4c>] dump_header.isra.5+0x50/0x15c
[460767.042603] [<ffffff800817f240>] oom_kill_process+0x94/0x3dc
[460767.043128] [<ffffff800817f7fc>] out_of_memory+0x1e4/0x2ac
[460767.043639] [<ffffff800845d9ac>] moom_callback+0x48/0x70
[460767.044128] [<ffffff80080cd264>] process_one_work+0x220/0x378
[460767.044663] [<ffffff80080ce124>] worker_thread+0x2e0/0x3a0
[460767.045176] [<ffffff80080d3004>] kthread+0xe0/0xe8
[460767.045627] [<ffffff80080826c0>] ret_from_fork+0x10/0x50
[460767.046235] Mem-Info:
[460767.046484] active_anon:33752 inactive_anon:6597 isolated_anon:0
                 active_file:535284 inactive_file:190256 isolated_file:0
                 unevictable:0 dirty:31 writeback:0 unstable:0
                 slab_reclaimable:48749 slab_unreclaimable:5217
                 mapped:25870 shmem:6685 pagetables:894 bounce:0
                 free:145659 free_pcp:686 free_cma:0
[460767.049564] DMA free:582636kB min:7900kB low:9872kB high:11848kB active_anon:13
5008kB inactive_anon:26388kB active_file:2141136kB inactive_file:761024kB unevictab
le:0kB isolated(anon):0kB isolated(file):0kB present:4061184kB managed:3903784kB ml
ocked:0kB dirty:124kB writeback:0kB mapped:103480kB shmem:26740kB slab_reclaimable:
194996kB slab_unreclaimable:20868kB kernel_stack:4352kB pagetables:3576kB unstable:
0kB bounce:0kB free_pcp:2744kB local_pcp:620kB free_cma:0kB writeback_tmp:0kB pages
_scanned:0 all_unreclaimable? no
[460767.053616] lowmem_reserve[]: 0 0 0
[460767.054043] DMA: 997*4kB (UME) 613*8kB (UME) 559*16kB (UME) 482*32kB (UM) 356*6
4kB (UM) 160*128kB (UME) 77*256kB (UME) 40*512kB (UM) 29*1024kB (ME) 21*2048kB (M)
96*4096kB (M) = 582636kB
[460767.055848] 732229 total pagecache pages
[460767.056242] 0 pages in swap cache
[460767.056559] Swap cache stats: add 0, delete 0, find 0/0
[460767.057065] Free swap  = 1048572kB
[460767.057390] Total swap = 1048572kB
[460767.057714] 1015296 pages RAM
[460767.058027] 0 pages HighMem/MovableOnly
[460767.058388] 39350 pages reserved
[460767.058689] [ pid ]   uid  tgid total_vm      rss nr_ptes nr_pmds swapents oom_
score_adj name
[460767.059547] [  195]     0   195     7917     1596       9       3        0
0 systemd-journal
[460767.060436] [  229]     0   229     3281      822       9       4        0
-1000 systemd-udevd
[460767.061301] [  257]   102   257     1952      979       8       4        0
0 systemd-network
```

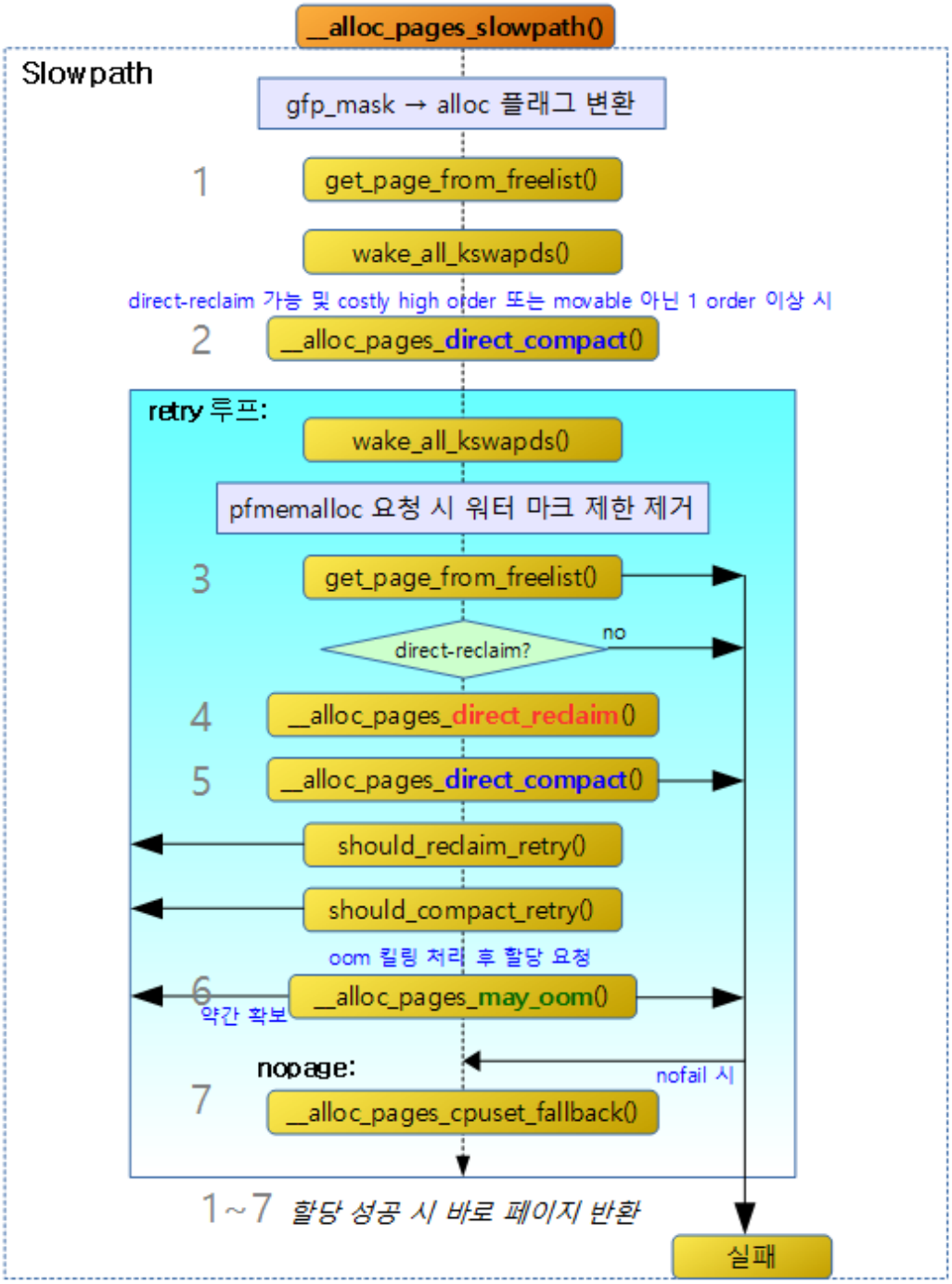
```
[460767.062193] [  382]   101   382    20508      936       10        5        0
0 systemd-timesyn
[460767.063081] [  416]     0   416    61035     1826       18        3        0
0 upowerd
[460767.063883] [  419]   106   419     1676      980        7        3        0
-900 dbus-daemon
[460767.064739] [  434]     0   434     2481     1120        7        4        0
0 wpa_supplicant
[460767.065614] [  437]     0   437     1900     1091        9        4        0
0 systemd-logind
[460767.066533] [  441]     0   441    79590     2400       20        4        0
0 udisksd
[460767.067365] [  450]     0   450      446      231        5        4        0
0 acpid
[460767.068170] [  454]     0   454    54453      725       11        3        0
0 rsyslogd
[460767.068973] [  459]     0   459    88404     4283       28        4        0
0 NetworkManager
[460767.069849] [  478]     0   478    58975     2351       18        4        0
0 polkitd
[460767.070677] [  568]   103   568     2102     1141        8        4        0
0 systemd-resolve
[460767.071565] [  585]     0   585    58528     1963       17        4        0
0 lightdm
[460767.072396] [  590]     0   590      623      363        6        4        0
0 agetty
[460767.073213] [  592]     0   592     1841      780        8        4        0
0 login
[460767.073991] [  603]     0   603   274336    17176       90        5        0
0 Xorg
[460767.074794] [  630]   110   630      441       92        5        4        0
0 uml_switch
[460767.075645] [  658]     0   658     2357     1385        8        3        0
0 systemd
[460767.076472] [  668]     0   668     2994      454       11        4        0
0 (sd-pam)
[460767.077298] [  673]     0   673     1675     1097        7        3        0
0 bash
[460767.078102] [  772]     0   772    40489     2052       14        4        0
0 lightdm
[460767.078904] [  780]  1000   780     2476     1328        9        4        0
0 systemd
[460767.079731] [  785]  1000   785     2994      454       11        4        0
0 (sd-pam)
[460767.080559] [  788]  1000   788    61601     3212       23        3        0
0 lxsession
[460767.081398] [  808]  1000   808     1801      382        7        4        0
0 dbus-launch
[460767.082257] [  809]  1000   809     1596      681        7        4        0
0 dbus-daemon
[460767.083134] [  830]  1000   830      984       79        6        4        0
0 ssh-agent
[460767.083948] [  838]  1000   838    58269     1523       14        3        0
0 gvfsd
[460767.084752] [  848]  1000   848    13921     3757       19        3        0
0 openbox
```

```
[460767.085579] [  853]  1000   853   196606     6848       44        4         0
0 lxpanel
[460767.086479] [  854]  1000   854    98755     7408       36        3         0
0 pcmanfm
[460767.087311] [  859]  1000   859      984       79        7        4         0
0 ssh-agent
[460767.088153] [  863]  1000   863    92314    13900       46        5         0
0 blueman-applet
[460767.089028] [  868]  1000   868   112028    14267       67        5         0
0 nm-applet
[460767.089842] [  869]  1000   869    43274     2728       20        3         0
0 xfce4-power-man
[460767.090729] [  876]  1000   876     2359     1082        9        4         0
0 xfconfd
[460767.091555] [  885]  1000   885   123932     2468       19        3         0
0 pulseaudio
[460767.092403] [  898]  1000   898    39537     1649       13        3         0
0 menu-cached
[460767.093268] [  905]     0   905     1810      963        7        4         0
0 bluetoothd
[460767.094139] [  915]  1000   915    67422     2887       21        5         0
0 gvfs-udisks2-vo
[460767.095036] [  923]  1000   923    77517     2078       19        5         0
0 gvfsd-trash
[460767.095889] [  933]  1000   933     9809     1559       12        3         0
0 obexd
[460767.096706] [ 5483]     0  5483     3049     1593       11        3         0
0 sshd
[460767.097537] [ 5498]     0  5498     2968     1498       10        3         0
0 sshd
[460767.098351] [ 5511]     0  5511      573      403        5        3         0
0 sftp-server
[460767.099273] [ 5518]     0  5518     1691     1156        7        3         0
0 bash
[460767.100048] [ 5735]     0  5735     3048     1579       11        4         0
0 sshd
[460767.100810] [ 5743]     0  5743     2968     1511       10        4         0
0 sshd
[460767.101580] [ 5766]     0  5766      573      427        6        3         0
0 sftp-server
[460767.102396] [ 5773]     0  5773     1698     1173        7        3         0
0 bash
[460767.103166] [ 5994]     0  5994     3048     1565        9        4         0
0 sshd
[460767.103928] [ 5999]     0  5999     2968     1529       10        4         0
0 sshd
[460767.104697] [ 6021]     0  6021      573      409        5        4         0
0 sftp-server
[460767.105515] [ 6028]     0  6028     1699     1172        7        3         0
0 bash
[460767.106289] [ 7742]     0  7742     2968     1514       10        3         0
0 sshd
[460767.107059] [ 7758]     0  7758     1697     1221        7        3         0
0 bash
[460767.107821] [ 7849]     0  7849     2968     1506       10        4         0
0 sshd
```

```
[460767.108591] [ 7863]     0  7863     1699     1201        7        3        0
0 bash
[460767.109360] Out of memory: Kill process 603 (Xorg) score 13 or sacrifice child
[460767.110302] Killed process 603 (Xorg) total-vm:1097344kB, anon-rss:15892kB, fil
e-rss:52812kB
```

__alloc_pages_slowpath()

The following illustration shows the slow-path page allocation process.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/alloc_pages_slowpath-1.png)

mm/page_alloc.c -1/5-

```
01 static inline struct page *
02 __alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order,
```

```
03                                                      struct alloc_context *a
    c)
04  {
05          bool can_direct_reclaim = gfp_mask & __GFP_DIRECT_RECLAIM;
06          const bool costly_order = order > PAGE_ALLOC_COSTLY_ORDER;
07          struct page *page = NULL;
08          unsigned int alloc_flags;
09          unsigned long did_some_progress;
10          enum compact_priority compact_priority;
11          enum compact_result compact_result;
12          int compaction_retries;
13          int no_progress_loops;
14          unsigned int cpuset_mems_cookie;
15          int reserve_flags;
16
17          /*
18           * We also sanity check to catch abuse of atomic reserves being
    used by
19           * callers that are not in atomic context.
20           */
21          if (WARN_ON_ONCE((gfp_mask & (__GFP_ATOMIC|__GFP_DIRECT_RECLAI
    M)) ==
22                                       (__GFP_ATOMIC|__GFP_DIRECT_RECLAIM)))
23                  gfp_mask &= ~__GFP_ATOMIC;
24
25  retry_cpuset:
26          compaction_retries = 0;
27          no_progress_loops = 0;
28          compact_priority = DEF_COMPACT_PRIORITY;
29          cpuset_mems_cookie = read_mems_allowed_begin();
30
31          /*
32           * The fast path uses conservative alloc_flags to succeed only u
    ntil
33           * kswapd needs to be woken up, and to avoid the cost of setting
    up
34           * alloc_flags precisely. So we do that now.
35           */
36          alloc_flags = gfp_to_alloc_flags(gfp_mask);
37
38          /*
39           * We need to recalculate the starting point for the zonelist it
    erator
40           * because we might have used different nodemask in the fast pat
    h, or
41           * there was a cpuset modification and we are retrying - otherwi
    se we
42           * could end up iterating over non-eligible zones endlessly.
43           */
44          ac->preferred_zoneref = first_zones_zonelist(ac->zonelist,
45                                          ac->high_zoneidx, ac->nodemask);
46          if (!ac->preferred_zoneref->zone)
47                  goto nopage;
48
49          if (alloc_flags & ALLOC_KSWAPD)
50                  wake_all_kswapds(order, gfp_mask, ac);
51
52          /*
53           * The adjusted alloc_flags might result in immediate success, s
    o try
54           * that first
55           */
56          page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
57          if (page)
58                  goto got_pg;
```

- In line 5 of the code, if a free page is below the threshold during page allocation, a direct-reclaim is required and whether it is allowed. Requests to allow direct-reclaim include:
  - GFP_KERNEL, GFP_KERNEL_ACCOUNT, GFP_NOIO, GFP_NOFS used to allocate kernel memory
  - GFP_USER used to allocate user memory
- If order 6 or higher is on line 3 of the code, it is judged to be a high order and is called a costly order.
- In line 21~23 of code, if you unreasonably make an atomic and direct-reclaim request to the gfp mask at the same time, remove it from the gfp mask to ignore the atomic request.
- In line 25~29 of code, prepare the compact as the default priority.
- In line 36 of the code, obtain the assignment flag as a gfp mask.
- In lines 44~47 of code, reset the node mask and zonelist and select the first zone again. If there are no zones available in the target node mask, go to the nopage: label.
- If kswapd reclaim is allowed in line 49~50 of the code, the nodes for the available zone related to the zonelist will wake up kswapd if the nodes have too little or too much free memory.
- Perform the first slow-path allocation attempt using the adjusted assignment flag in lines 56~58 of code.

mm/page_alloc.c -2/5-

```
01  .       /*
02           * For costly allocations, try direct compaction first, as it's
    likely
03           * that we have enough base pages and don't need to reclaim. For
    non-
04           * movable high-order allocations, do that as well, as compactio
    n will
05           * try prevent permanent fragmentation by migrating from blocks
    of the
06           * same migratetype.
07           * Don't try this for allocations that are allowed to ignore
08           * watermarks, as the ALLOC_NO_WATERMARKS attempt didn't yet hap
    pen.
09           */
10          if (can_direct_reclaim &&
11                          (costly_order ||
12                             (order > 0 && ac->migratetype != MIGRATE_MOVA
    BLE))
13                          && !gfp_pfmemalloc_allowed(gfp_mask)) {
14                  page = __alloc_pages_direct_compact(gfp_mask, order,
15                                                      alloc_flags, ac,
16                                                      INIT_COMPACT_PRIORITY,
17                                                      &compact_result);
18                  if (page)
19                          goto got_pg;
20
21                  /*
22                   * Checks for costly allocations with __GFP_NORETRY, whi
    ch
23                   * includes THP page fault allocations
24                   */
25                  if (costly_order && (gfp_mask & __GFP_NORETRY)) {
26                          /*
27                           * If compaction is deferred for high-order allo
    cations,
28                           * it is because sync compaction recently faile
    d. If
```

```
29                          * this is the case and the caller requested a T
HP
30                          * allocation, we do not want to heavily disrupt
the
31                          * system, so we fail the allocation instead of
entering
32                          * direct reclaim.
33                          */
34                         if (compact_result == COMPACT_DEFERRED)
35                                 goto nopage;
36
37                         /*
38                          * Looks like reclaim/compaction is worth tryin
g, but
39                          * sync compaction could be very expensive, so k
eep
40                          * using async compaction.
41                          */
42                         compact_priority = INIT_COMPACT_PRIORITY;
43                 }
44         }
```

- In line 10~19 of the code, if the following 3 conditions are satisfied at the same time, the first direct-component is performed and the page is allocated.
  - Allocation requests with direct-reclaim allowed
  - Costly High Order allocation requests, or allocation requests of more than 1 order that are not movable
  - Normal allocation requests that should not use pfmemalloc, which is used when making temporary allocation requests to reclaim pages.
- In line 25~43 of code, the first direct-compaction also failed to allocate the page. If the noretry option is used for costly order, use async compaction to try again. However, if the compact result is in a suspended state, it will be moved to the nopage label.

mm/page_alloc.c -3/5-

```
01 retry:
02         /* Ensure kswapd doesn't accidentally go to sleep as long as we
loop */
03         if (alloc_flags & ALLOC_KSWAPD)
04                 wake_all_kswapds(order, gfp_mask, ac);
05
06         reserve_flags = __gfp_pfmemalloc_flags(gfp_mask);
07         if (reserve_flags)
08                 alloc_flags = reserve_flags;
09
10         /*
11          * Reset the nodemask and zonelist iterators if memory policies
can be
12          * ignored. These allocations are high priority and system rathe
r than
13          * user oriented.
14          */
15         if (!(alloc_flags & ALLOC_CPUSET) || reserve_flags) {
16                 ac->nodemask = NULL;
17                 ac->preferred_zoneref = first_zones_zonelist(ac->zonelis
t,
18                                         ac->high_zoneidx, ac->nodemask);
19         }
20
21         /* Attempt with potentially adjusted zonelist and alloc_flags */
```

```
22          page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
23          if (page)
24                  goto got_pg;
25
26          /* Caller is not willing to reclaim, we can't balance anything
    */
27          if (!can_direct_reclaim)
28                  goto nopage;
29
30          /* Avoid recursion of direct reclaim */
31          if (current->flags & PF_MEMALLOC)
32                  goto nopage;
33
34          /* Try direct reclaim and then allocating */
35          page = __alloc_pages_direct_reclaim(gfp_mask, order, alloc_flag
    s, ac,
36                                                  &did_some_progre
    ss);
37          if (page)
38                  goto got_pg;
39
40          /* Try direct compaction and then allocating */
41          page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flag
    s, ac,
42                                              compact_priority, &compact_resul
    t);
43          if (page)
44                  goto got_pg;
45
46          /* Do not loop if specifically requested */
47          if (gfp_mask & __GFP_NORETRY)
48                  goto nopage;
49
50          /*
51           * Do not retry costly high order allocations unless they are
52           * __GFP_RETRY_MAYFAIL
53           */
54          if (costly_order && !(gfp_mask & __GFP_RETRY_MAYFAIL))
55                  goto nopage;
56
57          if (should_reclaim_retry(gfp_mask, order, ac, alloc_flags,
58                                   did_some_progress > 0, &no_progress_loo
    ps))
59                  goto retry;
```

- This is the retry: label for retrying page allocation on lines 1~4 of the code. If it is requested to be allocated in a state that can wake kswapd, it looks at the memory state and breaks kswapd.
- In line 6~8 of the code, if you need to use pfmemalloc, which is used when temporarily requesting allocation for page recall, set the allocation flag to ALLOC_NO_WATERMARKS to disable the watermark criterion.
- In line 15~24 of the code, if one of the following two conditions is present, try to allocate the page after switching to the first node and zone, ignoring the order of the nodes and zones that are traversing according to the memory policies.
    - Kernel allocation requests, not page allocation requests using cpuset
    - Allocation requests that should not use watermark criteria due to pfmemalloc
- In line 27~28 of the code, if the allocation request is not allowed to be direct-reclaimed, such as the atomic allocation request, it will no longer be able to reclaim the page, so move it to the nopage: label.
- In line 31~32 of code, if you need to use pfmemalloc, which is used when temporarily requesting an allocation to reclaim a page, move to the nopage: label without reclaiming the page so that a

recursive operation is not performed.

- Try direct-recalim on lines 35~38 of code.
- Try the second direct-component on lines 41~44 of code.
- If there is a noretry request on lines 47~48 of the code, go to the nopage: label.
- If the allocation request does not use the __GFP_RETRY_MAYFAIL flag in code lines 54~55, it will not retry for costly order, but will go to the nopage: label.
- If you need to retry the reclaim on lines 57~59 of the code, go to the retry: label and retry the assignment.

mm/page_alloc.c -4/5-

```
01  .       /*
02           * It doesn't make any sense to retry for the compaction if the
    order-0
03           * reclaim is not able to make any progress because the current
04           * implementation of the compaction depends on the sufficient am
    ount
05           * of free memory (see __compaction_suitable)
06           */
07          if (did_some_progress > 0 &&
08                          should_compact_retry(ac, order, alloc_flags,
09                                  compact_result, &compact_priority,
10                                  &compaction_retries))
11                  goto retry;
12
13
14          /* Deal with possible cpuset update races before we start OOM ki
    lling */
15          if (check_retry_cpuset(cpuset_mems_cookie, ac))
16                  goto retry_cpuset;
17
18          /* Reclaim has failed us, start killing things */
19          page = __alloc_pages_may_oom(gfp_mask, order, ac, &did_some_prog
    ress);
20          if (page)
21                  goto got_pg;
22
23          /* Avoid allocations with no watermarks from looping endlessly
    */
24          if (tsk_is_oom_victim(current) &&
25              (alloc_flags == ALLOC_OOM ||
26               (gfp_mask & __GFP_NOMEMALLOC)))
27                  goto nopage;
28
29          /* Retry as long as the OOM killer is making progress */
30          if (did_some_progress) {
31                  no_progress_loops = 0;
32                  goto retry;
33          }
```

- If you need to retry the compaction on lines 7~11 of the code, go to the retry: label and retry.
- If there is a change in the cpuset in line 15~16 of the code and the race situation is detected, go to the retry_cpuset: label and retry.
- In line 19~21 of the code, try the assignment again with the page obtained through OOM killing.
- In line 24~27 of the code, if the current task is in a state where a particular task is being killed due to OOM, it will be moved to the nopage: label.

- On lines 30~33 of code, if there is a possibility that the page will be reclaimed via OOM killing, go to the retry: label and retry.

mm/page_alloc.c -5/5-

```
01 nopage:
02         /* Deal with possible cpuset update races before we fail */
03         if (check_retry_cpuset(cpuset_mems_cookie, ac))
04                 goto retry_cpuset;
05
06         /*
07          * Make sure that __GFP_NOFAIL request doesn't leak out and make
    sure
08          * we always retry
09          */
10         if (gfp_mask & __GFP_NOFAIL) {
11                 /*
12                  * All existing users of the __GFP_NOFAIL are blockable,
    so warn
13                  * of any new users that actually require GFP_NOWAIT
14                  */
15                 if (WARN_ON_ONCE(!can_direct_reclaim))
16                         goto fail;
17
18                 /*
19                  * PF_MEMALLOC request from this context is rather bizar
    re
20                  * because we cannot reclaim anything and only can loop
    waiting
21                  * for somebody to do a work for us
22                  */
23                 WARN_ON_ONCE(current->flags & PF_MEMALLOC);
24
25                 /*
26                  * non failing costly orders are a hard requirement whic
    h we
27                  * are not prepared for much so let's warn about these u
    sers
28                  * so that we can identify them and convert them to some
    thing
29                  * else.
30                  */
31                 WARN_ON_ONCE(order > PAGE_ALLOC_COSTLY_ORDER);
32
33                 /*
34                  * Help non-failing allocations by giving them access to
    memory
35                  * reserves but do not use ALLOC_NO_WATERMARKS because t
    his
36                  * could deplete whole memory reserves which would just
    make
37                  * the situation worse
38                  */
39                 page = __alloc_pages_cpuset_fallback(gfp_mask, order, AL
    LOC_HARDER, ac);
40                 if (page)
41                         goto got_pg;
42
43                 cond_resched();
44                 goto retry;
45         }
46 fail:
47         warn_alloc(gfp_mask, ac->nodemask,
48                         "page allocation failure: order:%u", order);
49 got_pg:
```

```
50          return page;
51  }
```

- In line 1~4 of the code, the page was not allocated and was on the verge of giving up. If there is a change in the cpuset and a race situation is detected, go to the retry_cpuset: label and retry.
- If you used the nofail option on lines 10~45 of the code, it will retry until the page is allocated. However, if direct-reclaim is not allowed, it will fail.

GFP Mask-> Assignment Flag Conversion

gfp_to_alloc_flags()

mm/page_alloc.c

```
01  static inline unsigned int
02  gfp_to_alloc_flags(gfp_t gfp_mask)
03  {
04          unsigned int alloc_flags = ALLOC_WMARK_MIN | ALLOC_CPUSET;
05
06          /* __GFP_HIGH is assumed to be the same as ALLOC_HIGH to save a
    branch. */
07          BUILD_BUG_ON(__GFP_HIGH != (__force gfp_t) ALLOC_HIGH);
08
09          /*
10           * The caller may dip into page reserves a bit more if the calle
    r
11           * cannot run direct reclaim, or if the caller has realtime sche
    duling
12           * policy or is asking for __GFP_HIGH memory.  GFP_ATOMIC reques
    ts will
13           * set both ALLOC_HARDER (__GFP_ATOMIC) and ALLOC_HIGH (__GFP_HI
    GH).
14           */
15          alloc_flags |= (__force int) (gfp_mask & __GFP_HIGH);
16
17          if (gfp_mask & __GFP_ATOMIC) {
18                  /*
19                   * Not worth trying to allocate harder for __GFP_NOMEMAL
    LOC even
20                   * if it can't schedule.
21                   */
22                  if (!(gfp_mask & __GFP_NOMEMALLOC))
23                          alloc_flags |= ALLOC_HARDER;
24                  /*
25                   * Ignore cpuset mems for GFP_ATOMIC rather than fail, s
    ee the
26                   * comment for __cpuset_node_allowed().
27                   */
28                  alloc_flags &= ~ALLOC_CPUSET;
29          } else if (unlikely(rt_task(current)) && !in_interrupt())
30                  alloc_flags |= ALLOC_HARDER;
31
32          if (gfp_mask & __GFP_KSWAPD_RECLAIM)
33                  alloc_flags |= ALLOC_KSWAPD;
34
35  #ifdef CONFIG_CMA
36          if (gfpflags_to_migratetype(gfp_mask) == MIGRATE_MOVABLE)
37                  alloc_flags |= ALLOC_CMA;
38  #endif
39          return alloc_flags;
40  }
```

Configure the assignment flag with the @gfp_mask value and return it. The returned allocation flags and conditions are as follows:

- ALLOC_WMARK_MIN(0)
  - Default
- ALLOC_NO_WATERMARKS
  - pfmemalloc request:
- ALLOC_CPUSET
  - If it's not an atomic request
- ALLOC_HIGH
  - If you have a high request
- ALLOC_HARDER
  - RT task request
  - If there is an atomic request but no nomemalloc
- ALLOC_CMA
  - movable page type for assignment requests
- ALLOC_KSWAPD
  - If you have a swapd_reclaim request

- In line 4 of the code, assign the min watermark to the allocation flag and use cpuset.
- In line 15 of the code, add whether to request high to the assignment flag.
- In line 17~28 of the code, remove cpuset from the allocation flag if it is an atomic request. It also adds a harder flag unless it's a nomemalloc request.
- In line 29~30 of code, add the harder flag even if requested by the rt task.
- In line 32~33 of code, add the kswpd flag if there is a swapd_reclaim request.
- In line 36~37 of code, add a cma flag if there is a removable request.

gfp flags -> migrate type conversion

gfpflags_to_migratetype()

include/linux/gfp.h

```
01 static inline int gfpflags_to_migratetype(const gfp_t gfp_flags)
02 {
03         VM_WARN_ON((gfp_flags & GFP_MOVABLE_MASK) == GFP_MOVABLE_MASK);
04         BUILD_BUG_ON((1UL << GFP_MOVABLE_SHIFT) != ___GFP_MOVABLE);
05         BUILD_BUG_ON((___GFP_MOVABLE >> GFP_MOVABLE_SHIFT) != MIGRATE_MO
   VABLE);
06
07         if (unlikely(page_group_by_mobility_disabled))
08                 return MIGRATE_UNMOVABLE;
09
10         /* Group based on mobility */
11         return (gfp_flags & GFP_MOVABLE_MASK) >> GFP_MOVABLE_SHIFT;
12 }
```

Convert the migrate type that corresponds to the GFP flag to one of the following:

- MIGRATE_UNMOVABLE

- MIGRATE_RECLAIMABLE
- MIGRATE_MOVABLE

Wake up all kswapd

wake_all_kswapds()

mm/page_alloc.c

```
01 static void wake_all_kswapds(unsigned int order, gfp_t gfp_mask,
02                              const struct alloc_context *ac)
03 {
04         struct zoneref *z;
05         struct zone *zone;
06         pg_data_t *last_pgdat = NULL;
07         enum zone_type high_zoneidx = ac->high_zoneidx;
08
09         for_each_zone_zonelist_nodemask(zone, z, ac->zonelist, high_zone
   idx,
10                                         ac->nodemask) {
11                 if (last_pgdat != zone->zone_pgdat)
12                         wakeup_kswapd(zone, gfp_mask, order, high_zoneid
   x);
13                 last_pgdat = zone->zone_pgdat;
14         }
15 }
```

It traverses zones that are less than high_zoneidx in the zonelist of high_zoneidx or less, which are nodes set in the node mask, and wakes up all of those nodes' kswapd.

Reclaim Retry Required Check

should_reclaim_retry()

mm/page_alloc.c

```
01 /*
02  * Checks whether it makes sense to retry the reclaim to make a forward
   progress
03  * for the given allocation request.
04  *
05  * We give up when we either have tried MAX_RECLAIM_RETRIES in a row
06  * without success, or when we couldn't even meet the watermark if we
07  * reclaimed all remaining pages on the LRU lists.
08  *
09  * Returns true if a retry is viable or false to enter the oom path.
10  */
```

```
01 static inline bool
02 should_reclaim_retry(gfp_t gfp_mask, unsigned order,
03                      struct alloc_context *ac, int alloc_flags,
04                      bool did_some_progress, int *no_progress_loops)
05 {
06         struct zone *zone;
07         struct zoneref *z;
08         bool ret = false;
09
10         /*
11          * Costly allocations might have made a progress but this does
   n't mean
```

```
12          * their order will become available due to high fragmentation s
    o
13          * always increment the no progress counter for them
14          */
15         if (did_some_progress && order <= PAGE_ALLOC_COSTLY_ORDER)
16                 *no_progress_loops = 0;
17         else
18                 (*no_progress_loops)++;
19
20         /*
21          * Make sure we converge to OOM if we cannot make any progress
22          * several times in the row.
23          */
24         if (*no_progress_loops > MAX_RECLAIM_RETRIES) {
25                 /* Before OOM, exhaust highatomic_reserve */
26                 return unreserve_highatomic_pageblock(ac, true);
27         }
28
29         /*
30          * Keep reclaiming pages while there is a chance this will lead
31          * somewhere.  If none of the target zones can satisfy our alloc
    ation
32          * request even if all reclaimable pages are considered then we
    are
33          * screwed and have to go OOM.
34          */
35         for_each_zone_zonelist_nodemask(zone, z, ac->zonelist, ac->high_
    zoneidx,
36                                         ac->nodemask) {
37                 unsigned long available;
38                 unsigned long reclaimable;
39                 unsigned long min_wmark = min_wmark_pages(zone);
40                 bool wmark;
41
42                 available = reclaimable = zone_reclaimable_pages(zone);
43                 available += zone_page_state_snapshot(zone, NR_FREE_PAGE
    S);
44
45                 /*
46                  * Would the allocation succeed if we reclaimed all
47                  * reclaimable pages?
48                  */
49                 wmark = __zone_watermark_ok(zone, order, min_wmark,
50                                 ac_classzone_idx(ac), alloc_flags, avail
    able);
51                 trace_reclaim_retry_zone(z, order, reclaimable,
52                                 available, min_wmark, *no_progress_loop
    s, wmark);
53                 if (wmark) {
54                         /*
55                          * If we didn't make any progress and have a lot
    of
56                          * dirty + writeback pages then we should wait f
    or
57                          * an IO to complete to slow down the reclaim an
    d
58                          * prevent from pre mature OOM
59                          */
60                         if (!did_some_progress) {
61                                 unsigned long write_pending;
62
63                                 write_pending = zone_page_state_snapshot
    (zone,
64                                                         NR_ZONE_WRITE_PE
    NDING);
65
66                                 if (2 * write_pending > reclaimable) {
```

```
67                                      congestion_wait(BLK_RW_ASYNC, H
  Z/10);
68                                      return true;
69                                  }
70                          }
71
72                          ret = true;
73                          goto out;
74                  }
75          }
76
77  out:
78          /*
79           * Memory allocation/reclaim might be called from a WQ context a
  nd the
80           * current implementation of the WQ concurrency control doesn't
81           * recognize that a particular WQ is congested if the worker thr
  ead is
82           * looping without ever sleeping. Therefore we have to do a shor
  t sleep
83           * here rather than calling cond_resched().
84           */
85          if (current->flags & PF_WQ_WORKER)
86                  schedule_timeout_uninterruptible(1);
87          else
88                  cond_resched();
89          return ret;
90  }
```

Check the reclaimed pages and the remaining free pages to see if you want to continue the reclaim attempt. (true=continuous, false=stop) can be repeated within MAX_RECLAIM_RETRIES (16) times in the event of a costly high order exceeding request, and on the last attempt, all the free pages of the highatomic type that are reserved for high order processing by the atomic request should be changed to the requested page type.

- In line 15~27 of code, if there was a page recall from the direct-reclaim made before this function call, and the request exceeds the high order, increment the output argument no_progress_loops by 1 to save the number of times the direct-reclaim was retried. If this value exceeds the maximum number of reclaim attempts, it is on the verge of OOM, so the atomic request retrieves all free pages of the highatomic type that were reserved for high order processing and converts them to the requested page type. If none of the pages were retrieved during the previous direct-reclaim process, or if the request is less than a costly high order, the retry counter is set to 0.
- In line 35~43 of code, traverse the zones below high_zoneidx that target the node mask in the zonelist and calculate the number of free pages plus the maximum number of pages that can be recalled in that zone.
- In code lines 49~74, if the number of availble pages exceeds the min watermark threshold, go to the out: label to return true so that the reclaim can be retried. If the number of reclaim pages is 0 and more than 50% of the pages that can be reclaimed are in a write delay state, it will wait 0.1 seconds before returning true because it is unlikely that the page will be reclaimed even if it is retried immediately.
- In lines 77~89 of code, the out: label determines whether to slip or not before exiting the function. If a worker thread requests a page allocation, it will loop without slipping in the congestion state, so it will rest for at least 1 tick and give way to execution to another task. In

other cases, if there is a premption request, it will slip, otherwise it will not slip and exit the function immediately.

Check compaction retry required

should_compact_retry()

mm/page_alloc.c

```
01 static inline bool
02 should_compact_retry(struct alloc_context *ac, int order, int alloc_flag
   s,
03                     enum compact_result compact_result,
04                     enum compact_priority *compact_priority,
05                     int *compaction_retries)
06 {
07         int max_retries = MAX_COMPACT_RETRIES;
08         int min_priority;
09         bool ret = false;
10         int retries = *compaction_retries;
11         enum compact_priority priority = *compact_priority;
12
13         if (!order)
14                 return false;
15
16         if (compaction_made_progress(compact_result))
17                 (*compaction_retries)++;
18
19         /*
20          * compaction considers all the zone as desperately out of memor
   y
21          * so it doesn't really make much sense to retry except when the
22          * failure could be caused by insufficient priority
23          */
24         if (compaction_failed(compact_result))
25                 goto check_priority;
26
27         /*
28          * make sure the compaction wasn't deferred or didn't bail out e
   arly
29          * due to locks contention before we declare that we should give
   up.
30          * But do not retry if the given zonelist is not suitable for
31          * compaction.
32          */
33         if (compaction_withdrawn(compact_result)) {
34                 ret = compaction_zonelist_suitable(ac, order, alloc_flag
   s);
35                 goto out;
36         }
37
38         /*
39          * !costly requests are much more important than __GFP_RETRY_MAY
   FAIL
40          * costly ones because they are de facto nofail and invoke OOM
41          * killer to move on while costly can fail and users are ready
42          * to cope with that. 1/4 retries is rather arbitrary but we
43          * would need much more detailed feedback from compaction to
44          * make a better decision.
45          */
46         if (order > PAGE_ALLOC_COSTLY_ORDER)
47                 max_retries /= 4;
48         if (*compaction_retries <= max_retries) {
49                 ret = true;
```

```
50                  goto out;
51          }
52
53          /*
54           * Make sure there are attempts at the highest priority if we ex
    hausted
55           * all retries or failed at the lower priorities.
56           */
57  check_priority:
58          min_priority = (order > PAGE_ALLOC_COSTLY_ORDER) ?
59                          MIN_COMPACT_COSTLY_PRIORITY : MIN_COMPACT_PRIORI
    TY;
60
61          if (*compact_priority > min_priority) {
62                  (*compact_priority)--;
63                  *compaction_retries = 0;
64                  ret = true;
65          }
66  out:
67          trace_compact_retry(order, priority, compact_result, retries, ma
    x_retries, ret);
68          return ret;
69  }
```

Returns whether a compaction needs to be retried. (true=retries required, false=no retries required)

- On lines 13~14 of the code, a 0 order assignment request returns false as there is no need for a compaction attempt.
- In line 16~17 of code, increment the compaction_retries counter if there are pages that were migrated in the last compaction process.
- On lines 24~25 of the code, go to the check_priority: label when the last compaction process is fully completed.
- In line 33~36 of the code, the last compaction process is not completed for some reason. Returns whether it is appropriate to try compaction again.
- Returns true on lines 46~51 to iterate up to the maximum number of compaction retries. However, if the allocation request exceeds the costly high order, the 16 retries will be reduced by 1/4 to 4 retryes.
- In code lines 57~65, check_priority: Labels. We want to give more opportunities to retry cases with high compact priority. Therefore, if the compact priority exceeds the minimum priority, it degrades the compact priority by 1, resets the number of retries to 0, and returns true.

Page Assignment with OOM Killing

__alloc_pages_may_oom()

mm/page_alloc.c

```
01  static inline struct page *
02  __alloc_pages_may_oom(gfp_t gfp_mask, unsigned int order,
03          const struct alloc_context *ac, unsigned long *did_some_progres
    s)
04  {
05          struct oom_control oc = {
06                  .zonelist = ac->zonelist,
07                  .nodemask = ac->nodemask,
08                  .memcg = NULL,
09                  .gfp_mask = gfp_mask,
10                  .order = order,
```

```
        };
        struct page *page;

        *did_some_progress = 0;

        /*
         * Acquire the oom lock.  If that fails, somebody else is
         * making progress for us.
         */
        if (!mutex_trylock(&oom_lock)) {
                *did_some_progress = 1;
                schedule_timeout_uninterruptible(1);
                return NULL;
        }

        /*
         * Go through the zonelist yet one more time, keep very high watermark
         * here, this is only to catch a parallel oom killing, we must fail if
         * we're still under heavy pressure. But make sure that this reclaim
         * attempt shall not depend on __GFP_DIRECT_RECLAIM && !__GFP_NORETRY
         * allocation which will never fail due to oom_lock already held.
         */
        page = get_page_from_freelist((gfp_mask | __GFP_HARDWALL) &
                                      ~__GFP_DIRECT_RECLAIM, order,
                                      ALLOC_WMARK_HIGH|ALLOC_CPUSET, ac);
        if (page)
                goto out;

        /* Coredumps can quickly deplete all memory reserves */
        if (current->flags & PF_DUMPCORE)
                goto out;
        /* The OOM killer will not help higher order allocs */
        if (order > PAGE_ALLOC_COSTLY_ORDER)
                goto out;
        /*
         * We have already exhausted all our reclaim opportunities without any
         * success so it is time to admit defeat. We will skip the OOM killer
         * because it is very likely that the caller has a more reasonable
         * fallback than shooting a random task.
         */
        if (gfp_mask & __GFP_RETRY_MAYFAIL)
                goto out;
        /* The OOM killer does not needlessly kill tasks for lowmem */
        if (ac->high_zoneidx < ZONE_NORMAL)
                goto out;
        if (pm_suspended_storage())
                goto out;
        /*
         * XXX: GFP_NOFS allocations should rather fail than rely on
         * other request to make a forward progress.
         * We are in an unfortunate situation where out_of_memory cannot
         * do much for this context but let's try it to at least get
         * access to memory reserved if the current task is killed (see
         * out_of_memory). Once filesystems are ready to handle allocation
         * failures more gracefully we should just bail out here.
         */

        /* The OOM killer may not free memory on a specific node */
```

```
69          if (gfp_mask & __GFP_THISNODE)
70                  goto out;
71
72          /* Exhausted what can be done so it's blame time */
73          if (out_of_memory(&oc) || WARN_ON_ONCE(gfp_mask & __GFP_NOFAIL))
    {
74                  *did_some_progress = 1;
75
76                  /*
77                   * Help non-failing allocations by giving them access to
    memory
78                   * reserves
79                   */
80                  if (gfp_mask & __GFP_NOFAIL)
81                          page = __alloc_pages_cpuset_fallback(gfp_mask, o
    rder,
82                                               ALLOC_NO_WATERMARKS, ac);
83          }
84  out:
85          mutex_unlock(&oom_lock);
86          return page;
87  }
```

Attempt to secure the page through OOM killing. If it can be secured through this, a 1 will be printed on the did_some_progress.

- If OOM lock acquisition fails on code lines 20~24, schedule it for 1 tick and give it over execution to another task.
- In line 33~37 of the code, add hardwall and cpuset, and try again to allocate the page based on the high watermark without direct-reclaim.
  - This is because OOM killing is likely to cause memory pressure to be released.
- In line 40~41 of the code, if the current task is already in the core dump, go to the out: label.
- If the request exceeds the costly high order in line 43~44 of the code, it will not be overcome by the OOM killer, so it will be moved to the out label.
- In line 51~52 of code, if the request is __GFP_RETRY_MAYFAIL, it has already exhausted many reclaim opportunities, and it is likely to have a reasonable fallback, so it goes to the out:label to skip OOM killing.
  - migrate_pages() -> new_page() is used to request an assignment when __GFP_RETRY_MAYFILE is used.
- In code lines 54~55, if the allocation is requested in the DMA32 or lower zone, go to the out: label to skip OOM killing.
- If io and fs fail to use in code lines 56~57, go to the out: label to skip OOM killing.
- In line 69~70 of the code, if the request is to allocate only on the local node, go to the out: label to skip OOM killing.
- If OOM killing was performed on lines 73~83 of the code and the page was recalled, or if the nofail option was used, substitute 1 for the output argument did_some_progress. If the nofail option is enabled, try to allocate it via cpuset fallback.
- In code lines 84~86, the out: label unlocks the oom lock and returns the page.

cpuset fallback after OOM killing

__alloc_pages_cpuset_fallback()

mm/page_alloc.c

```
01 static inline struct page *
02 __alloc_pages_cpuset_fallback(gfp_t gfp_mask, unsigned int order,
03                               unsigned int alloc_flags,
04                               const struct alloc_context *ac)
05 {
06         struct page *page;
07
08         page = get_page_from_freelist(gfp_mask, order,
09                         alloc_flags|ALLOC_CPUSET, ac);
10         /*
11          * fallback to ignore cpuset restriction if our nodes
12          * are depleted
13          */
14         if (!page)
15                 page = get_page_from_freelist(gfp_mask, order,
16                                 alloc_flags, ac);
17
18         return page;
19 }
```

After applying cpuset to the alloc flag, try to allocate the page first, and if the allocation fails, try to allocate again, except for cpuset.

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) | Sentence C – Current post
- Zoned Allocator -3- (Buddy Page Allocation)) (http://jake.dothome.co.kr/buddy-alloc) | Qc
- Zoned Allocator -4- (Buddy Page Terminated) (http://jake.dothome.co.kr/buddy-free/) | Qc
- Zoned Allocator -5- (Per-CPU Page Frame Cache) (http://jake.dothome.co.kr/per-cpu-page-frame-cache) | 문c
- Zoned Allocator -6- (Watermark) (http://jake.dothome.co.kr/zonned-allocator-watermark) | 문c
- Zoned Allocator -7- (Direct Compact) (http://jake.dothome.co.kr/zonned-allocator-compaction) | 문c
- Zoned Allocator -8- (Direct Compact-Isolation) (http://jake.dothome.co.kr/zonned-allocator-isolation) | 문c
- Zoned Allocator -9- (Direct Compact-Migration) (http://jake.dothome.co.kr/zonned-allocator-migration) | 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

LEAVE A COMMENT

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

Comments

name *

email *

Website

WRITE A COMMENT

‹ Zoned Allocator -1- (Physics Page Assignment - Fastpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/)

Zoned Allocator -6- (Watermark) › (http://jake.dothome.co.kr/zonned-allocator-watermark/)

Munc Blog (2015 ~ 2023)