Zoned Allocator -11- (Direct Reclaim)

2016-07-16 (http://jake.dothome.co.kr/zonned-allocator-reclaim/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Zoned Allocator -11- (Direct Reclaim)

Reclaim Judgment

should_continue_reclaim()

mm/vmscan.c

```
1  /*
2   * Reclaim/compaction is used for high-order allocation requests. It rec
   laims
3   * order-0 pages before compacting the zone. should_continue_reclaim() r
   eturns
4   * true if more pages should be reclaimed such that when the page alloca
   tor
5   * calls try_to_compact_zone() that it will have enough free pages to su
   cceed.
6   * It will give up earlier than that if there is difficulty reclaiming p
   ages.
7   */
```

```
01 static inline bool should_continue_reclaim(struct pglist_data *pgdat,
02                                             unsigned long nr_reclaimed,
03                                             unsigned long nr_scanned,
04                                             struct scan_control *sc)
05 {
06         unsigned long pages_for_compaction;
07         unsigned long inactive_lru_pages;
08         int z;
09
10         /* If not in reclaim/compaction mode, stop */
11         if (!in_reclaim_compaction(sc))
12                 return false;
13
14         /* Consider stopping depending on scan and reclaim activity */
15         if (sc->gfp_mask & __GFP_RETRY_MAYFAIL) {
16                 /*
17                  * For __GFP_RETRY_MAYFAIL allocations, stop reclaiming
   if the
18                  * full LRU list has been scanned and we are still faili
   ng
19                  * to reclaim pages. This full LRU scan is potentially
20                  * expensive but a __GFP_RETRY_MAYFAIL caller really wan
   ts to succeed
21                  */
22                 if (!nr_reclaimed && !nr_scanned)
23                         return false;
24         } else {
25                 /*
26                  * For non-__GFP_RETRY_MAYFAIL allocations which can pre
   sumably
```

```
27                 * fail without consequence, stop if we failed to reclai
   m
28                 * any pages from the last SWAP_CLUSTER_MAX number of
29                 * pages that were scanned. This will return to the
30                 * caller faster at the risk reclaim/compaction and
31                 * the resulting allocation attempt fails
32                 */
33                if (!nr_reclaimed)
34                        return false;
35        }
36
37        /*
38         * If we have not reclaimed enough pages for compaction and the
39         * inactive lists are large enough, continue reclaiming
40         */
41        pages_for_compaction = compact_gap(sc->order);
42        inactive_lru_pages = node_page_state(pgdat, NR_INACTIVE_FILE);
43        if (get_nr_swap_pages() > 0)
44                inactive_lru_pages += node_page_state(pgdat, NR_INACTIVE
   _ANON);
45        if (sc->nr_reclaimed < pages_for_compaction &&
46                        inactive_lru_pages > pages_for_compaction)
47                return true;
48
49        /* If compaction would go ahead or the allocation would succeed,
   stop */
50        for (z = 0; z <= sc->reclaim_idx; z++) {
51                struct zone *zone = &pgdat->node_zones[z];
52                if (!managed_zone(zone))
53                        continue;
54
55                switch (compaction_suitable(zone, sc->order, 0, sc->recl
   aim_idx)) {
56                case COMPACT_SUCCESS:
57                case COMPACT_CONTINUE:
58                        return false;
59                default:
60                        /* check next zone */
61                        ;
62                }
63        }
64        return true;
65 }
```

Returns true if the high order page request is processed and reclaim/compliance should continue.

- In line 11~12 of the code, if it is not in reclaim/compaction mode, the processing will be stopped.
- If the __GFP_RETRY_MAYFAIL flag is used in lines 15~35 of the code, it returns a reclaimed page and false if there is no scanned page. If the reclaimed flag is not used, it returns false if there are no pages.
- In lines 41~47 of code, if the reclaimed page is smaller than twice the order page, which is small for compaction, but the number of inactive LRU pages is greater than twice the order page, it returns true.
- Traversing zones reclaim_idx in lines 50~64 of code and returns false if the action has already succeeded or should continue.
- In line 64 of code, if there is no success of the compaction in all traversed zones, it returns true to indicate that the compaction should continue.

in_reclaim_compaction()

mm/vmscan.c

```
01 /* Use reclaim/compaction for costly allocs or under memory pressure */
02 static bool in_reclaim_compaction(struct scan_control *sc)
03 {
04         if (IS_ENABLED(CONFIG_COMPACTION) && sc->order &&
05                         (sc->order > PAGE_ALLOC_COSTLY_ORDER ||
06                          sc->priority < DEF_PRIORITY - 2))
07                 return true;
08
09         return false;
10 }
```

Returns true if in reclaim/compaction mode.

- Except for a 0 order request, it returns true if the following two conditions are met:
  - It is being repeated by increasing the priority more than once. (The lower the number, the higher the priority)
  - Costly order request. (from order 4)

Perform a direct-reclaim

__alloc_pages_direct_reclaim()

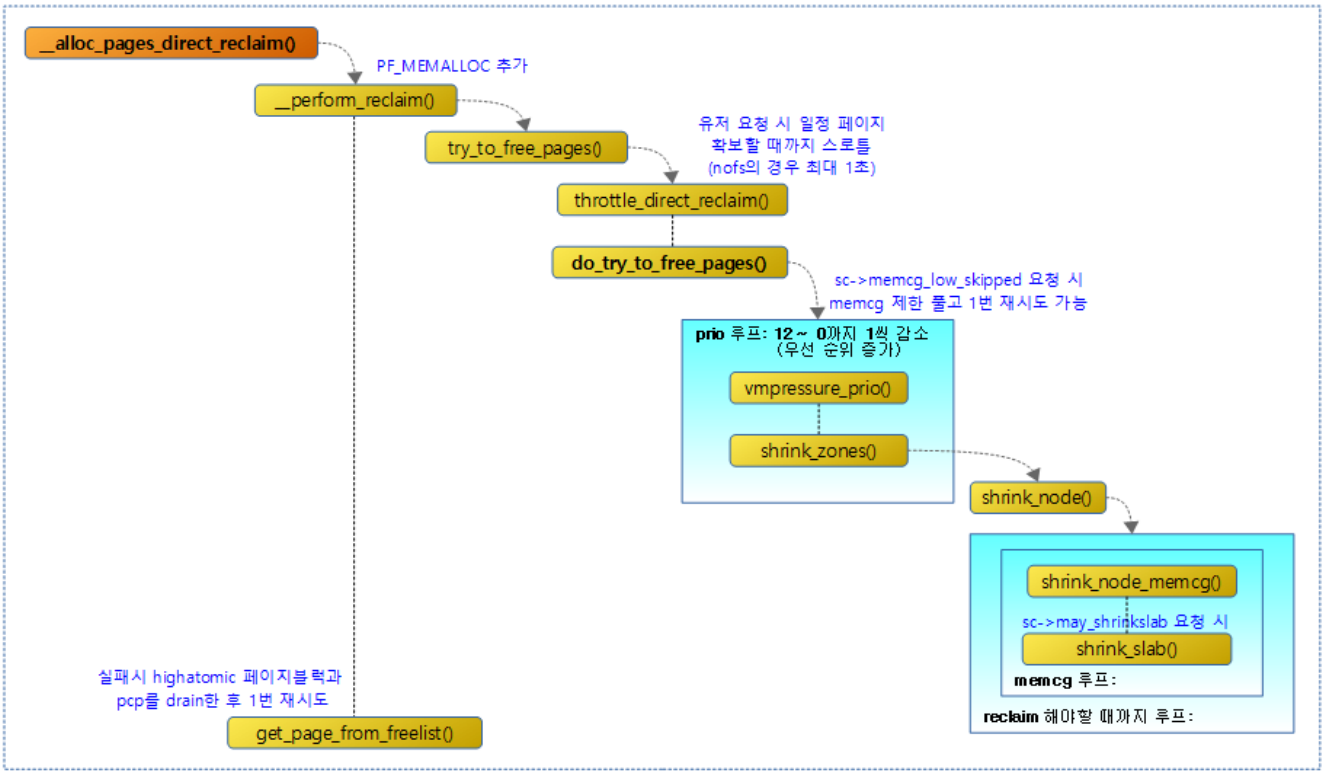
mm/page_alloc.c

```
01 /* The really slow allocator path where we enter direct reclaim */
02 static inline struct page *
03 __alloc_pages_direct_reclaim(gfp_t gfp_mask, unsigned int order,
04                 int alloc_flags, const struct alloc_context *ac,
05                 unsigned long *did_some_progress)
06 {
07         struct page *page = NULL;
08         bool drained = false;
09
10         *did_some_progress = __perform_reclaim(gfp_mask, order, ac);
11         if (unlikely(!(*did_some_progress)))
12                 return NULL;
13
14 retry:
15         page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
16
17         /*
18          * If an allocation failed after direct reclaim, it could be bec
    ause
19          * pages are pinned on the per-cpu lists. Drain them and try aga
    in
20          */
21         if (!page && !drained) {
22                 unreserve_highatomic_pageblock(ac, false);
23                 drain_all_pages(NULL);
24                 drained = true;
25                 goto retry;
26         }
27
28         return page;
29 }
```

After retrieving the page, try to assign the page. If it fails the first time, it will empty the PCP cache to free the buddy system and try again.

- It retrieves pages from lines 10~12 of the code, and returns null if there is a small probability that no pages have been retrieved.
- In lines 14~15 of the code, try to assign the order page from the retry: label.
- If page allocation fails in line 21~26 of the code, and it is the first failure, unblock the highatomic page, empty the pcp cache to free the page in the buddy system, and try again.

The following illustration shows the process of reclaiming a page via direct reclaim.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/alloc_pages_direct_reclaim-1a.png)

__perform_reclaim()

mm/page_alloc.c

```
01 /* Perform direct synchronous page reclaim */
02 static int
03 __perform_reclaim(gfp_t gfp_mask, unsigned int order,
04                                         const struct alloc_context *ac)
05 {
06         struct reclaim_state reclaim_state;
07         int progress;
08         unsigned int noreclaim_flag;
09         unsigned long pflags;
10
11         cond_resched();
12
13         /* We now go into synchronous reclaim */
14         cpuset_memory_pressure_bump();
15         psi_memstall_enter(&pflags);
16         fs_reclaim_acquire(gfp_mask);
17         noreclaim_flag = memalloc_noreclaim_save();
18         reclaim_state.reclaimed_slab = 0;
19         current->reclaim_state = &reclaim_state;
20
21         progress = try_to_free_pages(ac->zonelist, order, gfp_mask,
```

```
22                                                              ac->node
    mask);
23
24          current->reclaim_state = NULL;
25          memalloc_noreclaim_restore(noreclaim_flag);
26          fs_reclaim_release(gfp_mask);
27          psi_memstall_leave(&pflags);
28
29          cond_resched();
30
31          return progress;
32  }
```

Reclaim the page. The value returned is the number of pages reclaimed.

- If global cpuset_memory_pressure_enabled is set in line 14 of code, update the frequency meter of the current task cpuset.
  - Use the memory_pressure_enabled file in the root cpuset set to 1.
- Line 15 of the code tells psi that memory compression has begun.
- At line 17 of the code, a page allocation is needed for the purpose of retrieving the page. In order to prevent the recall routine from being called recursively, a PF_MEMALLOC is set on the flag of the current task for a while while reclaiming, so that it can be assigned after removing the watermark criteria.
- In code lines 18~19, reset the reclaimed_slab counter to 0 and assign it to the current task.
- Recall pages from lines 21~22 of the code and find out how many pages were retrieved.
- Release the reclaim_state you specified for the task in line 24.
- In line 25 of code, remove the PF_MEMALLOC that was set for a while while reclaiming the flag of the current task.
- Line 27 of code tells psi that memory compression is complete.

Scan Control

The routines for using the scan controls are as follows:

- reclaim_clean_pages_from_list()
- try_to_free_pages()
- mem_cgroup_shrink_node()
- try_to_free_mem_cgroup_pages()
- balance_pgdat()
- shrink_all_memory()
- __node_reclaim()

Attempt to get a free page by reclaiming a page

try_to_free_pages()

mm/vmscan.c

```
01  unsigned long try_to_free_pages(struct zonelist *zonelist, int order,
02                                   gfp_t gfp_mask, nodemask_t *nodemask)
03  {
```

```
04          unsigned long nr_reclaimed;
05          struct scan_control sc = {
06                  .nr_to_reclaim = SWAP_CLUSTER_MAX,
07                  .gfp_mask = current_gfp_context(gfp_mask),
08                  .reclaim_idx = gfp_zone(gfp_mask),
09                  .order = order,
10                  .nodemask = nodemask,
11                  .priority = DEF_PRIORITY,
12                  .may_writepage = !laptop_mode,
13                  .may_unmap = 1,
14                  .may_swap = 1,
15                  .may_shrinkslab = 1,
16          };
17
18          /*
19           * scan_control uses s8 fields for order, priority, and reclaim_
    idx.
20           * Confirm they are large enough for max values.
21           */
22          BUILD_BUG_ON(MAX_ORDER > S8_MAX);
23          BUILD_BUG_ON(DEF_PRIORITY > S8_MAX);
24          BUILD_BUG_ON(MAX_NR_ZONES > S8_MAX);
25
26          /*
27           * Do not enter reclaim if fatal signal was delivered while thro
    ttled.
28           * 1 is returned so that the page allocator does not OOM kill at
    this
29           * point.
30           */
31          if (throttle_direct_reclaim(sc.gfp_mask, zonelist, nodemask))
32                  return 1;
33
34          trace_mm_vmscan_direct_reclaim_begin(order,
35                                  sc.may_writepage,
36                                  sc.gfp_mask,
37                                  sc.reclaim_idx);
38
39          nr_reclaimed = do_try_to_free_pages(zonelist, &sc);
40
41          trace_mm_vmscan_direct_reclaim_end(nr_reclaimed);
42
43          return nr_reclaimed;
44  }
```

Attempts to reclaim pages and returns the number of pages recovered. At the user's request, the task may be throttled until the free page is below the normal zone and has at least half of the min watermark threshold.

- On lines 5~16 of code, prepare a scan_control struct for page retrieval.
- If a fatal signal is received during throttling for direct-reclaim on code lines 31~32, it will immediately exit the routine. It returns only 1, which prevents the OOM kill routine from being performed.
- Attempt to reclaim the page at line 39 of code.

Throttling on request by users

throttle_direct_reclaim()

mm/vmscan.c

```
/*
 * Throttle direct reclaimers if backing storage is backed by the network
 * and the PFMEMALLOC reserve for the preferred node is getting dangerously
 * depleted. kswapd will continue to make progress and wake the processes
 * when the low watermark is reached.
 *
 * Returns true if a fatal signal was delivered during throttling. If this
 * happens, the page allocator should not consider triggering the OOM killer.
 */
```

```
static bool throttle_direct_reclaim(gfp_t gfp_mask, struct zonelist *zonelist,
                                    nodemask_t *nodemask)
{
        struct zoneref *z;
        struct zone *zone;
        pg_data_t *pgdat = NULL;

        /*
         * Kernel threads should not be throttled as they may be indirectly
         * responsible for cleaning pages necessary for reclaim to make forward
         * progress. kjournald for example may enter direct reclaim while
         * committing a transaction where throttling it could forcing other
         * processes to block on log_wait_commit().
         */
        if (current->flags & PF_KTHREAD)
                goto out;

        /*
         * If a fatal signal is pending, this process should not throttle.
         * It should return quickly so it can exit and free its memory
         */
        if (fatal_signal_pending(current))
                goto out;

        /*
         * Check if the pfmemalloc reserves are ok by finding the first node
         * with a usable ZONE_NORMAL or lower zone. The expectation is that
         * GFP_KERNEL will be required for allocating network buffers when
         * swapping over the network so ZONE_HIGHMEM is unusable.
         *
         * Throttling is based on the first usable node and throttled processes
         * wait on a queue until kswapd makes progress and wakes them. There
         * is an affinity then between processes waking up and where reclaim
         * progress has been made assuming the process wakes on the same node.
         * More importantly, processes running on remote nodes will not compete
         * for remote pfmemalloc reserves and processes on different nodes
         * should make reasonable progress.
         */
```

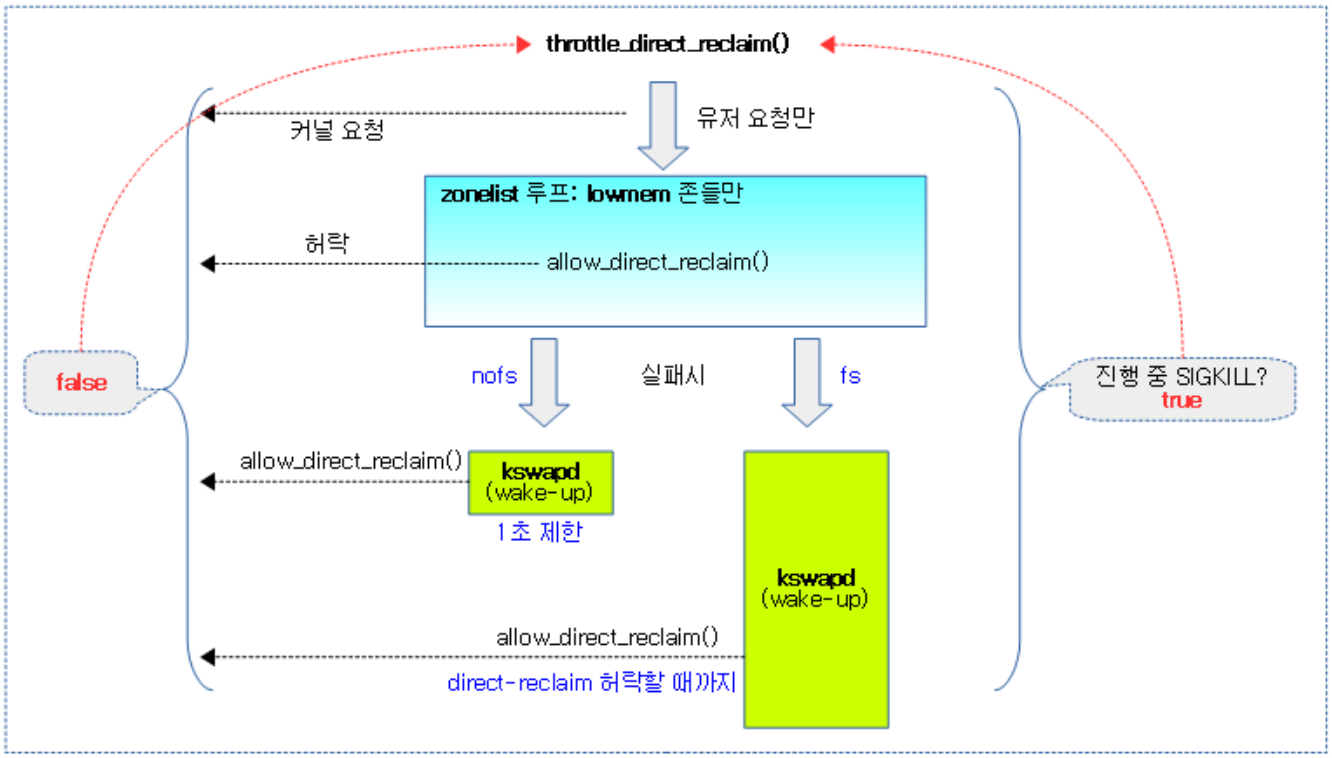
```
39          for_each_zone_zonelist_nodemask(zone, z, zonelist,
40                                          gfp_zone(gfp_mask), nodemask) {
41                  if (zone_idx(zone) > ZONE_NORMAL)
42                          continue;
43
44                  /* Throttle based on the first usable node */
45                  pgdat = zone->zone_pgdat;
46                  if (allow_direct_reclaim(pgdat))
47                          goto out;
48                  break;
49          }
50
51          /* If no zone was usable by the allocation flags then do not thr
    ottle */
52          if (!pgdat)
53                  goto out;
54
55          /* Account for the throttling */
56          count_vm_event(PGSCAN_DIRECT_THROTTLE);
57
58          /*
59           * If the caller cannot enter the filesystem, it's possible that
    it
60           * is due to the caller holding an FS lock or performing a journ
    al
61           * transaction in the case of a filesystem like ext[3|4]. In thi
    s case,
62           * it is not safe to block on pfmemalloc_wait as kswapd could be
63           * blocked waiting on the same lock. Instead, throttle for up to
    a
64           * second before continuing.
65           */
66          if (!(gfp_mask & __GFP_FS)) {
67                  wait_event_interruptible_timeout(pgdat->pfmemalloc_wait,
68                          allow_direct_reclaim(pgdat), HZ);
69
70                  goto check_pending;
71          }
72
73          /* Throttle until kswapd wakes the process */
74          wait_event_killable(zone->zone_pgdat->pfmemalloc_wait,
75                  allow_direct_reclaim(pgdat));
76
77  check_pending:
78          if (fatal_signal_pending(current))
79                  return true;
80
81  out:
82          return false;
83  }
```

Stream as many direct-reclaim requests as needed on user tasks. For non-file-system (NOFS) direct-reclaim requests, throttling is limited to 1 second. Returns whether the sigkill signal has been received during throttling.

- In line 15~16 of the code, if the request is made by the kernel thread, it aborts processing and returns false to avoid throttling.
- In the case of a task that is being processed with a SIGKILL signal on lines 22~23 of code, it will also stop processing and return false.
- If the direct-reclaim of the node's lowmem zones requested in code lines 39~49 is above the allowed standard, throttling is waived.
- If there are no nodes available in code lines 52~53, the processing is abandoned.
- This is where throttling starts at code line 56. PGSCAN_DIRECT_THROTTLE increases stat.

- In line 66~71 of the code, if the request is direct-reclaim that does not use the file system, it will be throttled for up to 1 second until direct-reclaim is allowed, and then moved to the check_pending label.
- In the case of direct-reclaim using the filesystem in line 74~75 of the code, it wakes up kswapd to free page and sleeps until direct-reclaim is allowed.
- In line 77~82 of code, if the current task requests a SIGKILL signal, it returns true, otherwise it returns false.

The following figure shows the process of throttling to use direct-reclaim based on whether the file system is used in a user request for direct-reclaim.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/throttle_direct_reclaim-1.png)

Direct-reclaim permission?

allow_direct_reclaim()

mm/vmscan.c

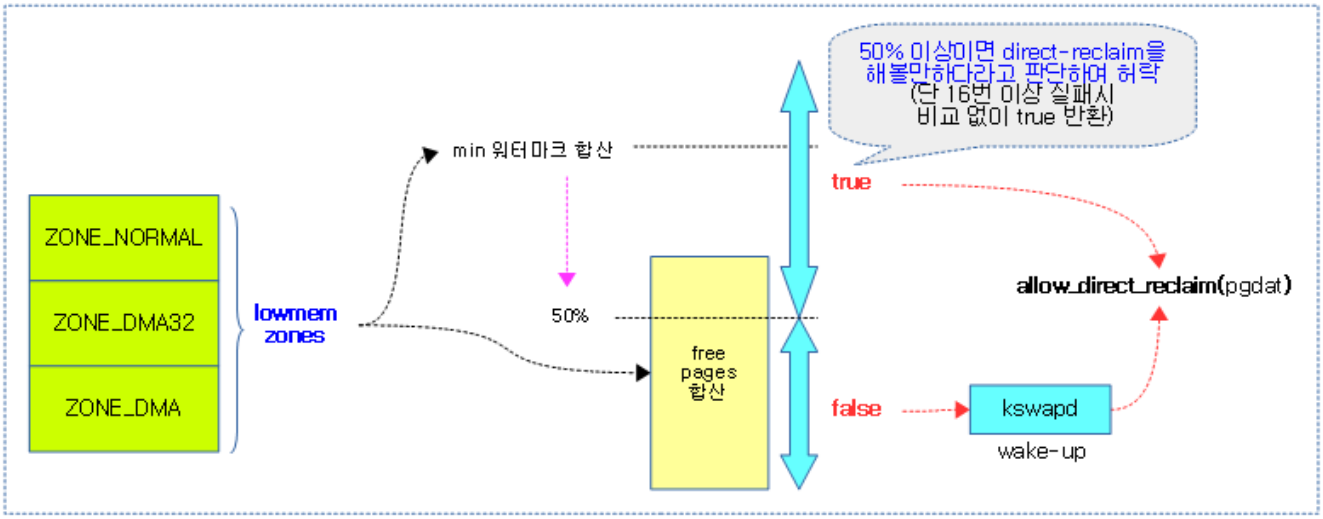
```
01 static bool allow_direct_reclaim(pg_data_t *pgdat)
02 {
03         struct zone *zone;
04         unsigned long pfmemalloc_reserve = 0;
05         unsigned long free_pages = 0;
06         int i;
07         bool wmark_ok;
08
09         if (pgdat->kswapd_failures >= MAX_RECLAIM_RETRIES)
10                 return true;
11
12         for (i = 0; i <= ZONE_NORMAL; i++) {
13                 zone = &pgdat->node_zones[i];
14                 if (!managed_zone(zone))
15                         continue;
```

```
16
17                  if (!zone_reclaimable_pages(zone))
18                          continue;
19
20                  pfmemalloc_reserve += min_wmark_pages(zone);
21                  free_pages += zone_page_state(zone, NR_FREE_PAGES);
22          }
23
24          /* If there are no reserves (unexpected config) then do not thro
    ttle */
25          if (!pfmemalloc_reserve)
26                  return true;
27
28          wmark_ok = free_pages > pfmemalloc_reserve / 2;
29
30          /* kswapd must be awake if processes are being throttled */
31          if (!wmark_ok && waitqueue_active(&pgdat->kswapd_wait)) {
32                  pgdat->kswapd_classzone_idx = min(pgdat->kswapd_classzon
    e_idx,
33                                                  (enum zone_type)ZONE_NOR
    MAL);
34                  wake_up_interruptible(&pgdat->kswapd_wait);
35          }
36
37          return wmark_ok;
38  }
```

Returns whether the requesting node allows direct-reclaim. If the free page of the lowmem zones is below the min watermark of 50%, sleep the current task, wake kswapd, and return false. If it's more than that, it thinks it's okay to try direct-reclaim, and returns true.

- In line 9~10 of the code, when the number of reclaim failures is more than MAX_RECLAIM_RETRIES (16), it immediately returns true to prevent throttling.
- In line 12~22 of the code, find the sum of the pfmemalloc_reserve value of the min watermark of the lowmem zones and the sum of the number of free pages.
- If the pfmemalloc_reserve value is 25 in line 26~0 of the code, it will immediately return true to prevent throttling.
- In line 28~35 of code, if the sum of free pages is less than 50% of the sum of the min watermark of the lowmem zones, the current task will be sleeped, kswapd will be woken up, and it will return false.

The following illustration illustrates the process of determining whether a direct-reclaim is granted.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/allow_direct_reclaim-1a.png)

do_try_to_free_pages()

mm/vmscan.c

```
/*
 * This is the main entry point to direct page reclaim.
 *
 * If a full scan of the inactive list fails to free enough memory then we
 * are "out of memory" and something needs to be killed.
 *
 * If the caller is !__GFP_FS then the probability of a failure is reasonably
 * high - the zone may be full of dirty or under-writeback pages, which this
 * caller can't do much about.  We kick the writeback threads and take explicit
 * naps in the hope that some of these pages can be written.  But if the
 * allocating task holds filesystem locks which prevent writeout this might not
 * work, and the allocation attempt will fail.
 *
 * returns:     0, if no pages reclaimed
 *              else, the number of pages reclaimed
 */
```

```
static unsigned long do_try_to_free_pages(struct zonelist *zonelist,
                                          struct scan_control *sc)
{
        int initial_priority = sc->priority;
        pg_data_t *last_pgdat;
        struct zoneref *z;
        struct zone *zone;
retry:
        delayacct_freepages_start();

        if (global_reclaim(sc))
                __count_zid_vm_events(ALLOCSTALL, sc->reclaim_idx, 1);

        do {
                vmpressure_prio(sc->gfp_mask, sc->target_mem_cgroup,
                                sc->priority);
                sc->nr_scanned = 0;
                shrink_zones(zonelist, sc);

                if (sc->nr_reclaimed >= sc->nr_to_reclaim)
```

```
21                      break;
22
23              if (sc->compaction_ready)
24                      break;
25
26              /*
27               * If we're getting trouble reclaiming, start doing
28               * writepage even in laptop mode.
29               */
30              if (sc->priority < DEF_PRIORITY - 2)
31                      sc->may_writepage = 1;
32      } while (--sc->priority >= 0);
33
34      last_pgdat = NULL;
35      for_each_zone_zonelist_nodemask(zone, z, zonelist, sc->reclaim_i
    dx,
36                                      sc->nodemask) {
37              if (zone->zone_pgdat == last_pgdat)
38                      continue;
39              last_pgdat = zone->zone_pgdat;
40              snapshot_refaults(sc->target_mem_cgroup, zone->zone_pgda
    t);
41              set_memcg_congestion(last_pgdat, sc->target_mem_cgroup,
    false);
42      }
43
44      delayacct_freepages_end();
45
46      if (sc->nr_reclaimed)
47              return sc->nr_reclaimed;
48
49      /* Aborted reclaim to try compaction? don't OOM, then */
50      if (sc->compaction_ready)
51              return 1;
52
53      /* Untapped cgroup reserves?  Don't OOM, retry. */
54      if (sc->memcg_low_skipped) {
55              sc->priority = initial_priority;
56              sc->memcg_low_reclaim = 1;
57              sc->memcg_low_skipped = 0;
58              goto retry;
59      }
60
61      return 0;
62  }
```

Attempt to secure a free page by reclaiming the page via a direct-reclaim request.

- In code lines 8~9, the retry: label is. Start by measuring the time it takes to retrieve a page.
  - See: delayacct_init() (http://jake.dothome.co.kr/delayacct_init) | Qc
- If you need to use global reclaim on code lines 11~12, increment the ALLOCSTALL stat.
- In line 14~16 of the code, the vmpressure information is updated when the priority is increased and the scan depth is deepened.
  - See: vmpressure (http://jake.dothome.co.kr/vmpressure) | Qc
- Reset the number of scans in line 17~18 of the code, retrieve the page, and know the number of recalls.
- In code lines 20~21, if the number of recalls is greater than the number of cases to be retrieved, it is taken out of the loop for processing.
- If the compaction is prepared in line 23~24 of code, it will get out of the loop for processing.
- If you want to increase the priority by 30 levels in line 31~2 of the code, set the writepage function.

- At line 32 of code, the priority is increased to the highest (the higher it is to 0), and the loop goes around.
- Traversing the zonelist on lines 34~42 of the code, updating the refaults of the node or memcg LRU for the node, and resetting the congested of the memcg node to false.
- At line 44 of the code, measure the time it takes to recall a page.
- If lines 46~47 of the code have been retrieved, return the value.
- Returns 50 if compaction is prepared on lines 51~1 of the code.
- If sc->memcg_low_skipped is set in code lines 54~59, the priority will be changed back to the original request priority and retry only for the first retry.

global_reclaim()

mm/vmscan.c

```
01 #ifdef CONFIG_MEMCG
02 static bool global_reclaim(struct scan_control *sc)
03 {
04         return !sc->target_mem_cgroup;
05 }
06 #else
07 static bool global_reclaim(struct scan_control *sc)
08 {
09         return true;
10 }
11 #endif
```

If you use the Memory Control Group using the CONFIG_MEMCG kernel option, it will return false if the target_mem_cgroup of the scan_control is determined. If not, it returns true for a global reclaim. This is always true if you don't use CONFIG_MEMCG.

Memory Pressure (per-cpuset reclaims)

cpuset_memory_pressure_bump()

include/linux/cpuset.h

```
1 #define cpuset_memory_pressure_bump()                           \
2         do {                                                    \
3                 if (cpuset_memory_pressure_enabled)             \
4                         __cpuset_memory_pressure_bump();        \
5         } while (0)
```

Update the frequency meter of the current task cpuset.

__cpuset_memory_pressure_bump()

kernel/cpuset.c

```
01 /**
02  * cpuset_memory_pressure_bump - keep stats of per-cpuset reclaims.
03  *
04  * Keep a running average of the rate of synchronous (direct)
05  * page reclaim efforts initiated by tasks in each cpuset.
```

```
 *
 * This represents the rate at which some task in the cpuset
 * ran low on memory on all nodes it was allowed to use, and
 * had to enter the kernels page reclaim code in an effort to
 * create more free memory by tossing clean pages or swapping
 * or writing dirty pages.
 *
 * Display to user space in the per-cpuset read-only file
 * "memory_pressure".  Value displayed is an integer
 * representing the recent rate of entry into the synchronous
 * (direct) page reclaim by any task attached to the cpuset.
 **/
```

```
void __cpuset_memory_pressure_bump(void)
{
        rcu_read_lock();
        fmeter_markevent(&task_cs(current)->fmeter);
        rcu_read_unlock();
}
```

Update the frequency meter of the current task cpuset.

fmeter_markevent()

kernel/cpuset.c

```
/* Process any previous ticks, then bump cnt by one (times scale). */
static void fmeter_markevent(struct fmeter *fmp)
{
        spin_lock(&fmp->lock);
        fmeter_update(fmp);
        fmp->cnt = min(FM_MAXCNT, fmp->cnt + FM_SCALE);
        spin_unlock(&fmp->lock);
}
```

Update the requested frequency meter and assign 1,000 to the number of events for the next calculation, but not more than 1,000,000.

fmeter_update()

kernel/cpuset.c

```
/* Internal meter update - process cnt events and update value */
static void fmeter_update(struct fmeter *fmp)
{
        time_t now = get_seconds();
        time_t ticks = now - fmp->time;

        if (ticks == 0)
                return;

        ticks = min(FM_MAXTICKS, ticks);
        while (ticks-- > 0)
                fmp->val = (FM_COEF * fmp->val) / FM_SCALE;
        fmp->time = now;

        fmp->val += ((FM_SCALE - FM_COEF) * fmp->cnt) / FM_SCALE;
        fmp->cnt = 0;
}
```

Calculate the val value with the requested frequency meter and reset the number of events to zero.

- In line 4~8 of the code, we find the number of seconds that have elapsed from the second recorded in the fmeter.
- In code lines 10~12, ticks are limited to a maximum of 99, and fmp->val *= 93.3% is repeated for ticks.
- On line 13 of the code, update it to the current second for the next calculation.
- In line 15~16 of code, add fmp->val to fmp->cnt x 6.7% and reset the number of events to zero.

fmeter struct

kernel/cgroup/cpuset.c

```
1 struct fmeter {
2         int cnt;                /* unprocessed events count */
3         int val;                /* most recent output value */
4         time_t time;            /* clock (secs) when val computed */
5         spinlock_t lock;        /* guards read or write of above */
6 };
```

- cnt
  - Number of unhandled events
- val
  - Calculated value from the most recent fmeter update
- time
  - clock(secs) when val value is calculated.

consultation

- Zoned Allocator -1- (Physics Page Assignment - Fastpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath) | Qc
- Zoned Allocator -2- (Physics Page Assignment - Slowpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-slowpath) | Qc
- Zoned Allocator -3- (Buddy Page Allocation) (http://jake.dothome.co.kr/buddy-alloc) | Qc
- Zoned Allocator -4- (Buddy Page Terminated) (http://jake.dothome.co.kr/buddy-free/) | Qc
- Zoned Allocator -5- (Per-CPU Page Frame Cache) (http://jake.dothome.co.kr/per-cpu-page-frame-cache) | 문c
- 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) | Sentence C – Current post

- 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

- Overview of Memory Reclaim in the Current Upstream Kernel (2021) | SUSE – 다운로드 pdf (https://lpc.events/event/11/contributions/896/attachments/793/1493/slides-r2.pdf)
- Optimizing Linux Memory Management for Low-latency / High-throughput Databases (https://engineering.linkedin.com/performance/optimizing-linux-memory-management-low-latency-high-throughput-databases)

LEAVE A COMMENT

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

Comments

name *

email *

Website

WRITE A COMMENT

‹ Zoned Allocator -13- (Direct Reclaim-Shrink-2) (http://jake.dothome.co.kr/zonned-allocator-shrink-2/)

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

Munc Blog (2015 ~ 2023)