Zoned Allocator -1- (Physics Page Assignment - Fastpath)

2016-06-24 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Structure of the Buddy System

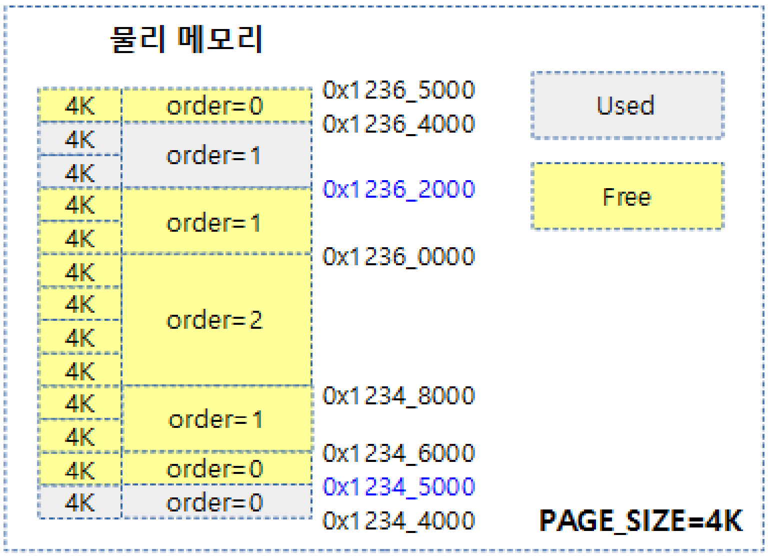
The buddy page allocator, called the buddy system, performs page-by-page memory allocation and release. The buddy system manages pages that can be assigned consecutively in multipliers of two. Pages are divided into order slots from 2^2 = 0 page to 1^(MAX_ORDER – 2).

Page Assignment Order

In the buddy system used by the page allocator, the page assignment is requested using the term order. This means that requests can only be made to the power of two. For example, if you are requesting a page corresponding to order 2, you are requesting 3^2 = 3 pages.

The following figure shows the order pages that the Buddy system can request to be assigned.

- There are 0 free pages in a row between 1234x5000_0 ~ 1236x2000_9, but the pages that can be assigned in the buddy system that are managed by aligning them by order are as follows.
  - 0 order: 1 page
  - 1 order: 2 page
  - 2 order: 1 page



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/buddy-3.png)

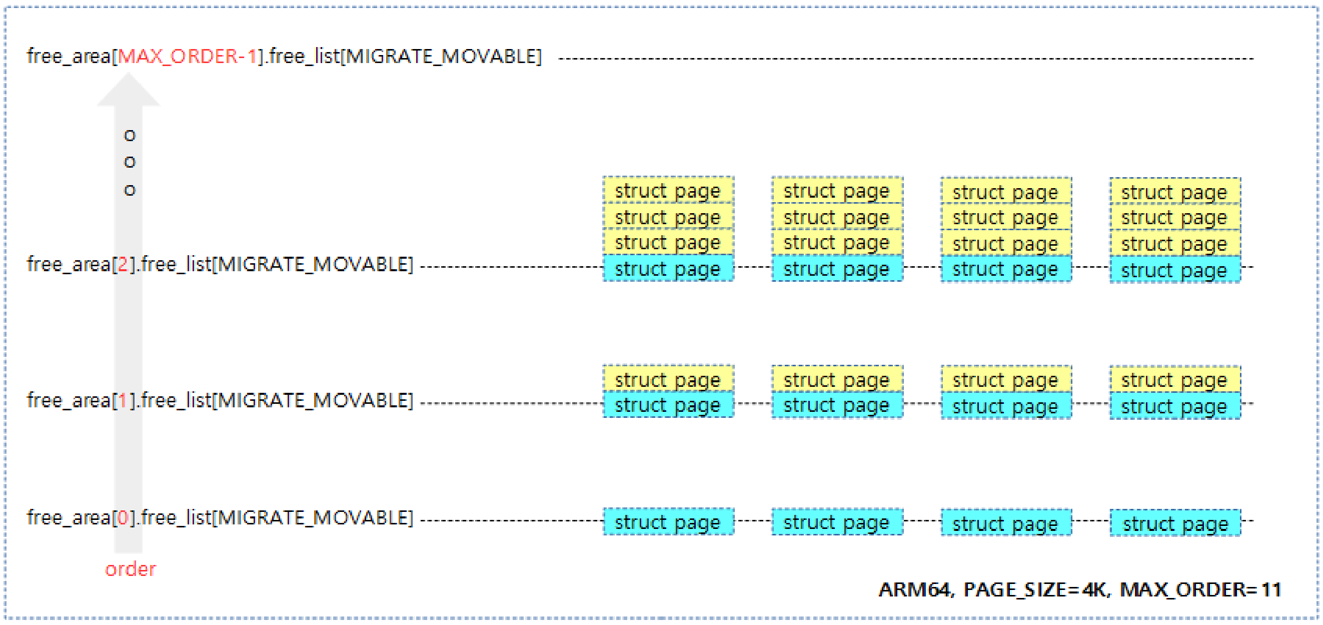
MAX_ORDER

The maximum number of pages that can be assigned at a time in the Buddy system is 2^(MAX_ORDER-1) pages.

- For example, if PAGE_SIZE=4K and MAX_ORDER=11, the maximum allocable page at a time is 1024 pages, or 4M bytes.
  - 2^0, 2^1, 2^2, ... 2^10 pages
  - 4K, 8K, 16K, ..., 4M pages

The following figure shows that the order slot that manages the pages manages free pages from 0 up to MAX_ORDER-1.

- Among the page structures for each free memory, the page structure corresponding to head is the representative page and is linked to the list.


free_area[MAX_ORDER-1].free_list[MIGRATE_MOVABLE]
free_area[2].free_list[MIGRATE_MOVABLE]
free_area[1].free_list[MIGRATE_MOVABLE]
free_area[0].free_list[MIGRATE_MOVABLE]
struct page
order
ARM64, PAGE_SIZE=4K, MAX_ORDER=11

(http://jake.dothome.co.kr/wp-content/uploads/2016/06/buddy-4.png)

The default setting for the ARM32 and ARM64 kernels is 11, and the MAX_ORDER is defined as <>. You can also use the CONFIG_FORCE_MAX_ZONEORDER kernel options to change the size. Each order slot also has the following structure to ensure that it is not fragmented.

- In order to keep pages with the same mobility attributes as close together as possible, each order slot is managed separately by migration type. By dividing and managing in this way, it is possible to increase the efficiency of the page recall and memory compaction process. You can also create a ZONE_MOVABLE area that consists only of MIGRATE_MOVABLE types.
- In the free_list containing each page, the free pages form a pair (buddy), and when two pairs (buddies) are combined, they can be merged into a larger order, and if necessary, they can be divided into one smaller order. Now, we no longer use a bitmap named map to manage buddies, but only a list named free_list and page information.

- The free_list has hot properties in the leading direction and cold properties in the aft direction. The hot and cold attributes are managed in correspondence with the position of the head and tail of the list, respectively. The first pages are likely to be reassigned and used. The pages placed in the back are the ones where the orders are consolidated and are more likely to move up to the top order. This helps to prevent fragmentation of free pages and also increases performance by increasing the persistence of the cache.

As the management techniques of the buddy system continue to evolve, the complexity is increasing, but it is developing to increase the efficiency (non-fragmentation) of the buddy system as much as possible. Figure 4-41 shows the core portion of the buddy memory allocator.

Page Blocks and Mobility (Migrate) Attributes

The memory is divided into page blocks, and the mobility property is expressed in a bitmap using 4 bits for each page block. Using the first 3 bits, each block of memory is separated by a migration type to manage the mobility properties. A page block manages 2^pageblock_order pages and records the mobility attributes used by those pages as representative mobility attributes in the memory block. The remaining 1 bit can be skipped by the compaction function.

Migrate Type

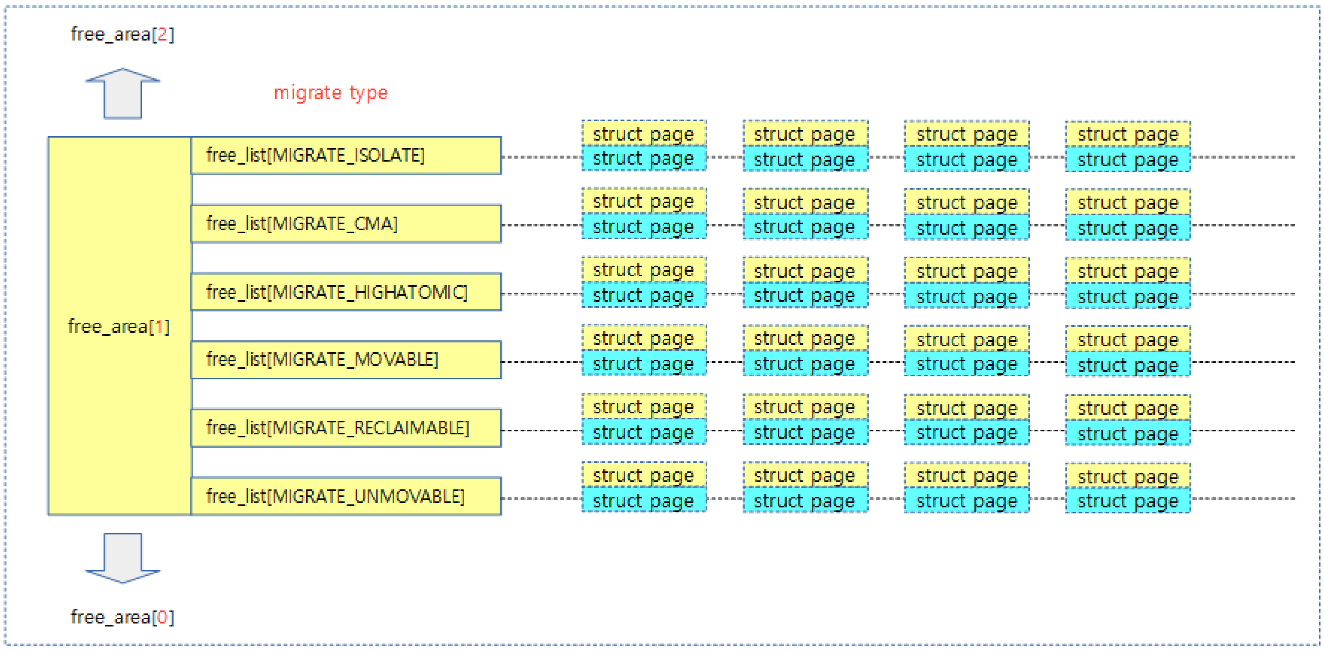
It is also called the page type. Each order page has a migrate type to represent its mobility properties, and if possible, it is necessary to group pages with the same attributes together to prevent continuous memory fragmentation. It is designed for the buddy system with the aim of maintaining as large and contiguous free memory as possible. The free_list of the buddy system are managed by the following migrate types: However, in a system with very little memory (several ~ tens of M) that does not hold more than 1 page block per migrate type, all pages are configured as unmovable.

For reference, Buddy's PCP cache is managed using only the following three types, which are the most common among the following types.

- MIGRATE_UNMOVABLE
  - It is a type that cannot be moved or retrieved from memory.
  - use
    - This type is used for pages allocated by the kernel, slabs, I/O buffers, kernel stacks, page tables, etc.
- MIGRATE_MOVABLE
  - If a large continuous memory is required, it is used to move the currently used page to prevent fragmentation as much as possible.
  - use
    - It is used for user (file, anon) memory allocation.
- MIGRATE_RECLAIMABLE

  - It is a type that cannot be moved, but it is used when it is possible to retrieve memory in case of lack of memory, and it is not a type that is used frequently.
  - use
    - If it is a slap cache created with a special __GFP_RECLAIMABLE flag, it is used as this type
- MIGRATE_HIGHATOMIC
  - In order to reduce the probability of atomic allocation requests failing for high-order page allocation, the kernel prepares this type of page type in advance by 1 block. This type of memory can be expanded to the range of 1% of the maximum memory.
  - use
    - If a kernel thread or interrupt handler that uses an RT scheduler requests a memory allocation using a GFP_ATOMIC when allocating memory, it should be handled without reclaiming the page, which can cause a slip. However, if a request for high-order page allocation is processed without retrieving the pages, there may be no high-order pages even if there is enough memory, resulting in OOM (Out Of Memory). In this case, you can allocate a page of type MIGRATE_HIGHATOMIC that has been reserved in advance to get you through the crisis.
  - In kernel 4.4-rc1, the MIGRATE_RESERVE type was removed and instead a MIGRATE_HIGHATOMIC was added to support high-order atomic allocation.
- MIGRATE_CMA
  - This is a page type that is managed separately by the CMA memory allocator. If a CMA region is configured on the buddy system, this region can also be assigned movable pages. If this area is not enough in the CMA request, the movable page will be moved to another area. Once allocated as CMA pages, the allocation and management of each page is performed separately by the CMA memory allocator.
  - use
    - It is used when the kernel allocates memory for DMA purposes, etc.
- MIGRATE_ISOLATE
  - The kernel temporarily changes a range of movable pages to this type in order to migrate them elsewhere. And we never use the buddy system for pages of this type.
  - use
    - It is used to move movable pages out of the CMA area to secure free pages in a situation where the CMA area is running out of memory.
    - Used to move in-use movable pages in a given area to another location for memory hot-removal.

The following illustration shows a list of free pages for each of the six migrate types for each order slot.


free_area[2]
migrate type
free_area[1]
free_list[MIGRATE_ISOLATE]
free_list[MIGRATE_CMA]
free_list[MIGRATE_HIGHATOMIC]
free_list[MIGRATE_MOVABLE]
free_list[MIGRATE_RECLAIMABLE]
free_list[MIGRATE_UNMOVABLE]
struct page
free_area[0]

(http://jake.dothome.co.kr/wp-content/uploads/2016/06/buddy-5.png)

page_blockorder

A pageblock_order defined by a macro expresses the number of pages that make up a page block in multipliers.

The ARM64 kernel supports huge pages, and they are used on a huge page-by-page basis. If you don't use huge pages, use them according to the maximum page size used by the buddy system.

- e.g. 4K pages and huge pages pageblock_order=9
- e.g. 4K pages and huge pages without use pageblock_order=10

include/linux/pageblock-flags.h

```
#define pageblock_order         HUGETLB_PAGE_ORDER
```

The ARM32 kernel is used to match the maximum page size used by the buddy system.

- e.g. 4K page pageblock_order=10

include/linux/pageblock-flags.h

```
#define pageblock_order (MAX_ORDER-1)
```

The following ARM64 system shows that the pageblock_order is set to 9, and shows the number of free pages being managed in the buddy for each node, zone, order, and migrate type. It also shows the number of page blocks for each node, zone, and migrate type.

```
$ cat /proc/pagetypeinfo
Page block order: 9
Pages per block:  512

Free pages count per migrate type at order       0      1      2      3      4
5      6      7      8      9     10
Node    0, zone    DMA32, type    Unmovable      0      0    168    322    108
6      2      0      0      1      0
Node    0, zone    DMA32, type      Movable      0      0     31      9      3
4      0      0      0      0    412
Node    0, zone    DMA32, type  Reclaimable      0      0      1      0      0
1      1      1      1      0      0
Node    0, zone    DMA32, type   HighAtomic      0      0      0      0      0
0      0      0      0      0      0
Node    0, zone    DMA32, type          CMA      0      0      0      0      0
0      1      1      1      1      7
Node    0, zone    DMA32, type      Isolate      0      0      0      0      0
0      0      0      0      0      0

Number of blocks type     Unmovable      Movable  Reclaimable   HighAtomic
CMA      Isolate
Node 0, zone    DMA32          232         1250           38            0
16            0

Number of mixed blocks    Unmovable      Movable  Reclaimable   HighAtomic
CMA      Isolate
Node 0, zone    DMA32            0            1            1            0
0           0
```

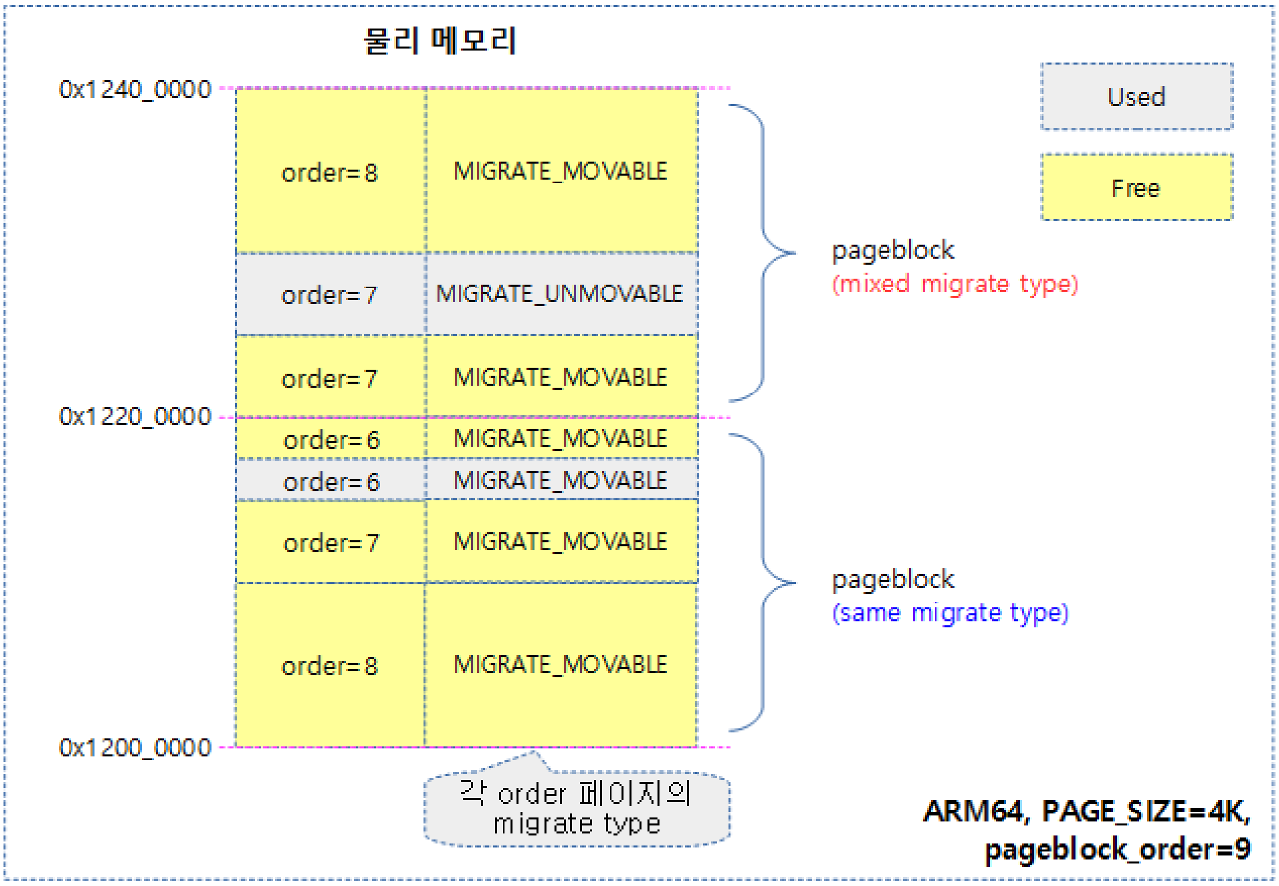
The following ARM32 system shows that the pageblock_order is set to 10.

```
$ cat /proc/pagetypeinfo
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4
5      6      7      8      9     10
Node    0, zone   Normal, type    Unmovable     10     13      9      6      2
2      0      1      0      1      0
Node    0, zone   Normal, type  Reclaimable      1      2      2      3      0
0      1      0      1      0      0
Node    0, zone   Normal, type      Movable      6      3      1      1    447    1
09      4      0      0      1    147
Node    0, zone   Normal, type      Reserve      0      0      0      0      0
0      0      0      0      0      2
Node    0, zone   Normal, type          CMA      1      1      1      2      1
2      1      0      1      1      0
Node    0, zone   Normal, type      Isolate      0      0      0      0      0
0      0      0      0      0      0

Number of blocks type     Unmovable  Reclaimable      Movable      Reserve
CMA      Isolate
Node 0, zone   Normal            4            5          223            2
2            0
```

- In the case of RPI2, you can see that MAX_ORDER=11 and pageblock_order is also set to MAX_ORDER-1.
    - The MAX_ORDER value varies depending on the kernel version. Previously, 9, 10, etc. were used.

The following figure shows that the order pages contained in each pageblock have a single migrate type, and that they are mixed without a mix.

- The kernel tries to maintain one migrate type in the same block whenever possible, but migrate types can be mixed in situations such as running out of memory.
- If you manage the migrate type so that it doesn't mix, it will be more likely that it won't be fragmented.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/buddy-6b.png)

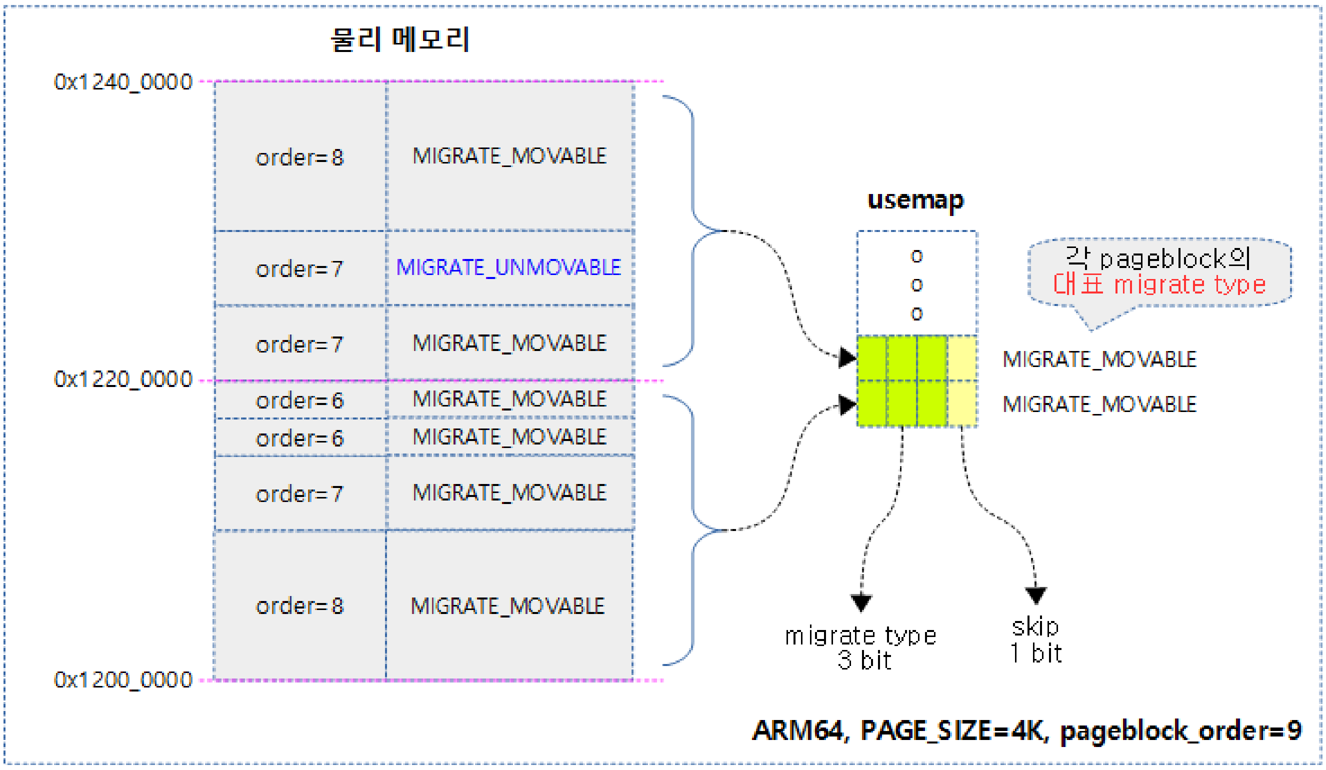
Representative Migrate Type by Pageblock

In the same way that order pages are managed by a migrate type, pageblocks are managed by their own migrate type. If there are a mix of order pages in a pageblock that use multiple migrate types, the most commonly used migrate type is specified as the representative migrate type of the block. Add 3 bits of type migrate and 1 bit skip bit to use in the compaction, and store them in the usemap.

What is the usemap for each bit?

- Representative migrate type (3bits)
  - In order to avoid fragmentation, we try to allocate from a pageblock that has the same representative migrate type as the requested migrate type when allocating the page.
- skip bits (1bits)
  - This is the bit for skipping the compaction when performing a compaction in case of out-of-memory.

The following figure shows the representative migrate type for each pageblock represented in the usemap.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/buddy-7a.png)

PF_MEMALLOC

This is a flag bit used by special kernel threads related to memory reclaim, or threads that need to temporarily allocate pages when reclaiming pages. If the memory is low below the watermark standard, the page retrieval system will be activated. At this time, in the process of retrieving a page, it may be necessary to allocate a page for a while in several special page recall requests, including swaps using the network. For example, when a page is insufficient and a network needs to be swapped, SKB (socket buffer) required for network processing should be allocated. There is a problem that when a memory allocation request is made, the page reclamation system for insufficient memory is activated repeatedly, and so on. Therefore, to prevent such recursive problems from occurring, use the PF_MEMALLOC flag to identify a "temporary memory allocation request to address an out-of-memory situation" when page allocation needs to be done for a special purpose. Using this flag allows you to perform the following actions:

- It also allocates memory below the watermark standard.
- Don't re-request any kind of page recall to avoid repeating it.

The following two flags have been added to be used when called to allocate temporary memory in low memory situations, such as PF_MEMALLOC flags.

PF_MEMALLOC_NOIO

- It also allocates memory below the watermark standard.
- Prevent page recalls with IO requests from being repeated. In other words, page recalls that only work in memory, not IO requests, can be executed.

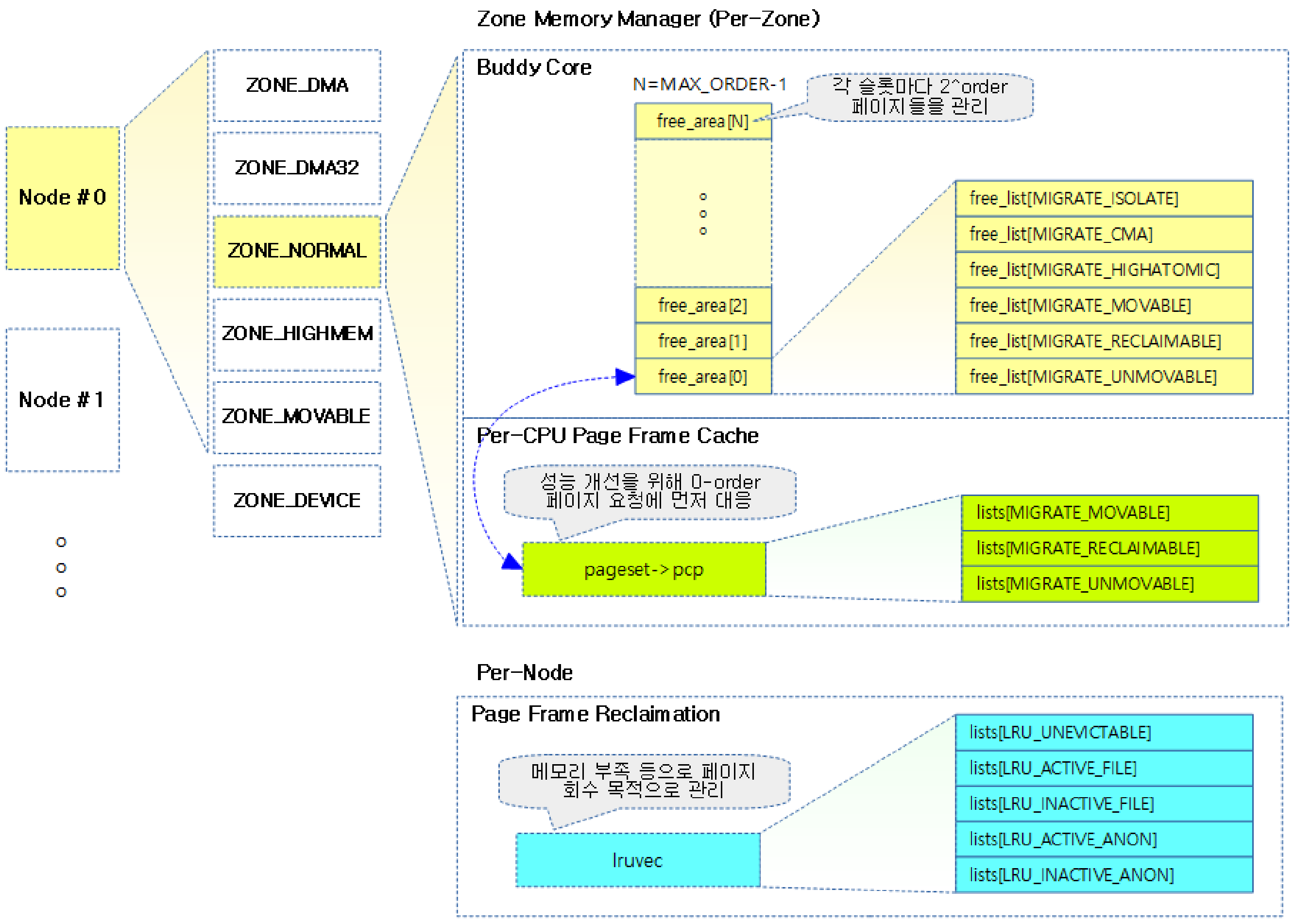
PF_MEMALLOC_NOFS

- It also allocates memory below the watermark standard.
- Prevents repeated recalls of pages using the file system. In other words, page recalls that use memory or other types of IO rather than the file system can be triggered.
- Note: mm: introduce memalloc_nofs_{save,restore} API (https://github.com/torvalds/linux/commit/7dea19f9ee636cb244109a4dba426bbb3e5304b7)

Page allocator structure

The following illustration shows the main items that make up the page allocator.

- By Node
  - Nodes and zonelists according to NUMA memory policies
  - Page Recall Mechanism
- Zone Stars
  - Buddy Core (Heart)
  - Buddy Cache (PCP)



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

Page Assignment Sequence

Page allocators are largely assigned through the following routines:

- The first step is to determine the nodes or nodes that are targeted through the NUMA memory policy.
- Memory Control Group is assigned within control.
- Assignments are made through the buddy system.
  - If it is a 1-page (0-order) allocation request, it is allocated using PCP, the buddy cache system.
  - In the event of a lack of memory, it may be accompanied by memory reclamation, depending on whether there is an interrupt or the flag option requested.

GFP Mask (gfp_mask)

These are the flags used when requesting page assignments.

- See: GFP Flags (http://jake.dothome.co.kr/gfp-flag) | Qc

Assignment Flags (alloc_flags)

These are assignment flags that are used separately from gfp_mask in the page allocation function, and are used for internal purposes.

- ALLOC_WMARK_MIN
  - This is a criterion to limit the allocation if the remaining memory falls below the min watermark.
  - User memory allocation requests (GFP_USER) and general kernel memory allocation requests (GFP_KERNEL) limit allocations below this threshold.
  - However, if a request is made for an allocation of GFP_ATOMIC, about half of this standard is allowed to be used for emergency purposes.
- ALLOC_WMARK_LOW
  - If the remaining memory falls below the low watermark, it is used as a basis to activate page recall systems such as kcompactd and kswapd.
- ALLOC_WMARK_HIGH
  - When the remaining memory is above the high watermark, it is used as a criterion to slip and stop the operation of page retrieval systems such as kcompactd and kswapd.
- ALLOC_NO_WATERMARKS
  - It allows you to override the watermark criteria and assign them.
  - Tasks that use PF_MEMALLOC flags (kswapd, kcompactd, ... When page recall threads) require memory allocation, these flags are used because they need to be able to override the watermark criterion and allocate them.
- ALLOC_HARDER
  - This flag is used in the following situations:
    - When using GFP_ATOMIC flags to request memory allocation

        - Requesting an allocation from a kernel thread that uses the RT scheduler
    - This flag performs the following actions:
        - In case of insufficient memory, the remaining min will be allocated 25% more than the watermark standard.
            - When using GFP_ATOMIC, 50% from the following ALLOC_HIGH and an additional 25% will be applied.
        - In case the high order page allocation fails, use the MIGRATE_HIGHATOMIC freelist to prevent the high order allocation from failing.
- ALLOC_HIGH
    - When using the GFP_ATOMIC flag, it is used in conjunction with the ALLOC_HARDER, and in the event of a low memory, the remaining min is allocated 50% more than the watermark threshold.
- ALLOC_CPUSET
    - Limit the memory requested by the task using the cgroup's cpuset subsystem.
    - For memory requested by the interrupt context, it ignores cpuset and allocates it.
- ALLOC_CMA
    - When requesting an allocation for a movable page, it will attempt to allocate without using the cma region if possible, but if you use this flag, it will also attempt to allocate using the cma area if you run out of memory.
- ALLOC_NOFRAGMENT
    - When allocating a page, it attempts to allocate within a page block consisting only of the requested migratetype.
    - However, if you run out of memory, you will inevitably ignore this flag request and allocate a fragment.
    - When it is necessary to allocate kernel memory using the normal zone, such as GFP_KERNEL or GFP_ATOMIC, etc., if DMA (or DMA32) is configured under the normal zone in the node, use these flags to avoid assigning and configuring multiple pages of migratetype within a one-page block as much as possible.
- ALLOC_KSWAPD
    - When requesting memory allocation for GFP_KERNEL, GFP_USER, GFP_HIGHUSER, etc., except for GFP_ATOMIC, a __GFP_RECLAIM flag (direct + kswapd) is added, and this flag is used by checking the __GFP_RECLAIM_KSWAPD. When memory is insufficient, it immediately triggers kcompactd and kswapd threads.

ALLOC_FAIR

The fair zone policy with the ALLOC_FAIR flag was removed in kernel v4.8-rc1 as it became unnecessary while using the memory policy of the NUMA system.

- Note: mm, page_alloc: remove fair zone allocation policy (https://github.com/torvalds/linux/commit/e6cbd7f2efb433d717af72aa8510a9db6f7a7e05)

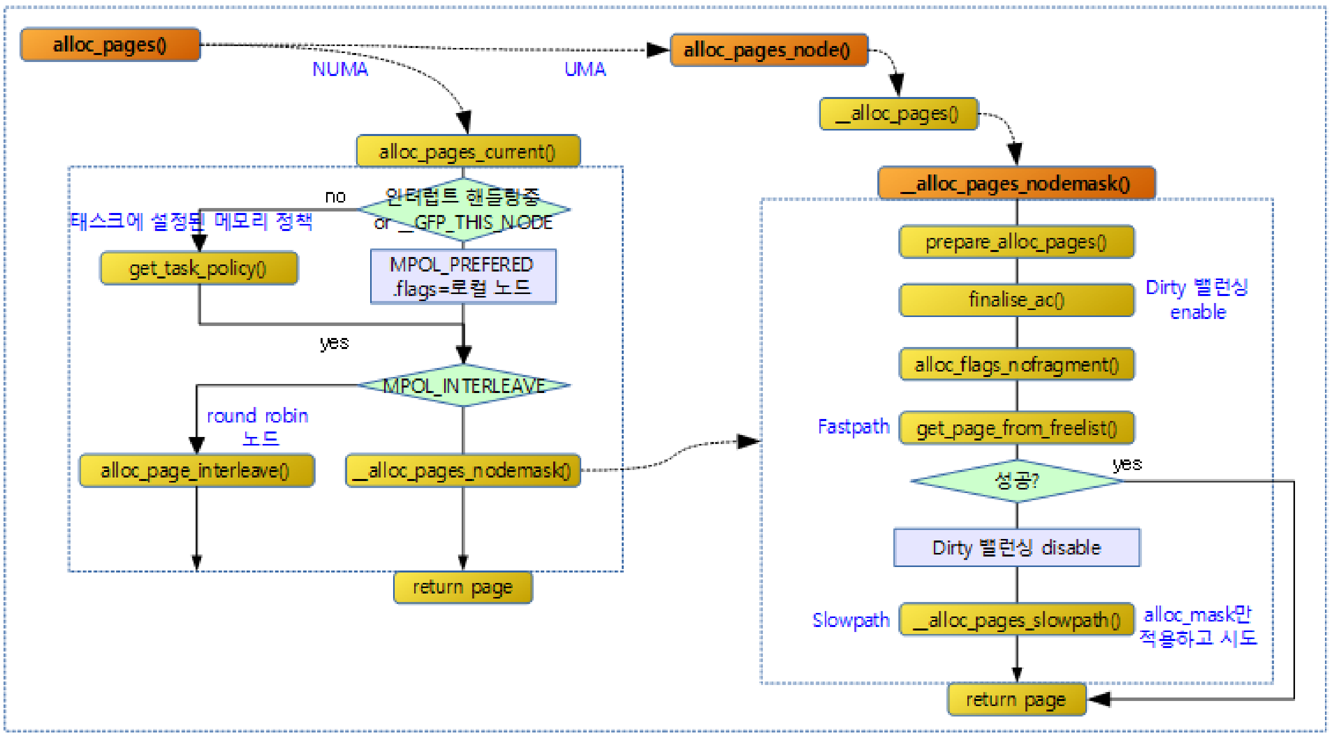
NUMA Memory Policy

- Note: NUMA -3- (Memory policy) (http://jake.dothome.co.kr/numa-3-memory-policy/) | Qc

---

Physics Page Assignment (Alloc)

The following figure shows the flow of page assignment.



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

alloc_pages()

include/linux/gfp.h

```
#ifdef CONFIG_NUMA
static inline struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
{
        return alloc_pages_current(gfp_mask, order);
}
#else
#define alloc_pages(gfp_mask, order) \
                alloc_pages_node(numa_node_id(), gfp_mask, order)
#endif
```

It is assigned a series of 2^order pages via the buddy system. If you are using the NUMA system, a node is selected via the alloc_pages_current( ) function to reflect the NUMA memory policy, and then the page is assigned by calling the alloc_pages_node( ) function inside that function.

alloc_pages_current()

mm/mempolicy.c

```
01 /**
02  *      alloc_pages_current - Allocate pages.
03  *
04  *      @gfp:
05  *              %GFP_USER   user allocation,
06  *              %GFP_KERNEL kernel allocation,
07  *              %GFP_HIGHMEM highmem allocation,
08  *              %GFP_FS     don't call back into a file system.
09  *              %GFP_ATOMIC don't sleep.
10  *      @order: Power of two of allocation size in pages. 0 is a single page.
11  *
12  *      Allocate a page from the kernel page pool.  When not in
13  *      interrupt context and apply the current process NUMA policy.
14  *      Returns NULL when no page can be allocated.
15  */
```

```
01 struct page *alloc_pages_current(gfp_t gfp, unsigned order)
02 {
03         struct mempolicy *pol = &default_policy;
04         struct page *page;
05 
06         if (!in_interrupt() && !(gfp & __GFP_THISNODE))
07                 pol = get_task_policy(current);
08 
09         /*
10          * No reference counting needed for current->mempolicy
11          * nor system default_policy
12          */
13         if (pol->mode == MPOL_INTERLEAVE)
14                 page = alloc_page_interleave(gfp, order, interleave_nodes(pol));
15         else
16                 page = __alloc_pages_nodemask(gfp, order,
17                                 policy_node(gfp, pol, numa_node_id()),
18                                 policy_nodemask(gfp, pol));
19 
20         return page;
21 }
22 EXPORT_SYMBOL(alloc_pages_current);
```

In the NUMA system, nodes are selected according to the memory policy and are allocated consecutive 2^order pages through the buddy system.

- If interrupts are being processed on code lines 3~7 or if the request is to allocate only on the current node, select the default memory policy. Otherwise, select the memory policy given to the current task. If a task does not have a memory policy set, it uses the memory policy that takes precedence over the node if a node is specified. If no node is specified, select the default memory policy. The default memory policy is allocated using local nodes.
  - If you used the __GFP_THISNODE flag to restrict to local nodes, use the -> default memory policy
  - Use the -> default (local node preferred) memory policy during interrupts
  - The policy specified in the task
    - If the task has a policy specified, the > memory policy specified in the task
    - If a specified node exists, use the priority memory policy specified on the -> node
    - If no node is specified, the -> default (local node preferred) memory policy

- If you use the interleaved memory policy in lines 13~14 of the code, have the pages allocated to each node in rotation.
- If you use any other policy in line 15~16 of the code, the order page will be assigned within the requested node limit.

alloc_pages_node()

include/linux/gfp.h

```
static inline struct page *alloc_pages_node(int nid, gfp_t gfp_mask,
                                            unsigned int order)
{
        /* Unknown node is current node */
        if (nid < 0)
                nid = numa_node_id();

        return __alloc_pages(gfp_mask, order, node_zonelist(nid, gfp_mask));
}
```

It is assigned a series of 2^order pages on the specified node. If an unknown node is specified, it is assigned from the current node.

__alloc_pages()

include/linux/gfp.h

```
static inline struct page *
__alloc_pages(gfp_t gfp_mask, unsigned int order,
                struct zonelist *zonelist)
{
        return __alloc_pages_nodemask(gfp_mask, order, zonelist, NULL);
}
```

In the zonelist, which contains the priority of nodes and zones, 2^order pages are allocated contiguous physical memory.

Heart of the Buddy Assigned

Assigning Pages in Specified Nodes

__alloc_pages_nodemask()

mm/page_alloc.c

```
/*
 * This is the 'heart' of the zoned buddy allocator.
 */
```

```
struct page *
__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order, int preferred_nid,
                                                        nodemask_t *nodemask)
```

```
{
        struct page *page;
        unsigned int alloc_flags = ALLOC_WMARK_LOW;
        gfp_t alloc_mask; /* The gfp_t that was actually used for allocation */
        struct alloc_context ac = { };

        /*
         * There are several places where we assume that the order value is sane
         * so bail out early if the request is out of bound.
         */
        if (unlikely(order >= MAX_ORDER)) {
                WARN_ON_ONCE(!(gfp_mask & __GFP_NOWARN));
                return NULL;
        }

        gfp_mask &= gfp_allowed_mask;
        alloc_mask = gfp_mask;
        if (!prepare_alloc_pages(gfp_mask, order, preferred_nid, nodemask, &ac, &alloc_mask, &alloc__
flags))
                return NULL;

        finalise_ac(gfp_mask, &ac);

        /*
         * Forbid the first pass from falling back to types that fragment
         * memory until all local zones are considered.
         */
        alloc_flags |= alloc_flags_nofragment(ac.preferred_zoneref->zone, gfp_mask);

        /* First allocation attempt */
        page = get_page_from_freelist(alloc_mask, order, alloc_flags, &ac);
        if (likely(page))
                goto out;

        /*
         * Apply scoped allocation constraints. This is mainly about GFP_NOFS
         * resp. GFP_NOIO which has to be inherited for all allocation requests
         * from a particular context which has been marked by
         * memalloc_no{fs,io}_{save,restore}.
         */
        alloc_mask = current_gfp_context(gfp_mask);
        ac.spread_dirty_pages = false;

        /*
         * Restore the original nodemask if it was potentially replaced with
         * &cpuset_current_mems_allowed to optimize the fast-path attempt.
         */
        if (unlikely(ac.nodemask != nodemask))
                ac.nodemask = nodemask;

        page = __alloc_pages_slowpath(alloc_mask, order, &ac);

out:
        if (memcg_kmem_enabled() && (gfp_mask & __GFP_ACCOUNT) && page &&
            unlikely(memcg_kmem_charge(page, gfp_mask, order) != 0)) {
                __free_pages(page, order);
                page = NULL;
61         }
62 
63         trace_mm_page_alloc(page, order, alloc_mask, ac.migratetype);
64 
65         return page;
66 }
67 EXPORT_SYMBOL(__alloc_pages_nodemask);
```

Refer to the specified node mask, zonelist, and flags settings to select nodes and zones, and then allocate physical memory as many as 2^order pages consecutively.

- In line 6 of the code, specify that the initial value of the assignment flag is to use the low watermark.
- In code lines 14~17, the @order value cannot be more than MAX_ORDER. In this case, it returns null.
  - The maximum number of pages that can request contiguous memory in the buddy system at a time is 2^(MAX_ORDER-1).
- In line 19 of code, while the kernel bootup process is being processed, no driver or file system is ready for IO processing for page allocation, and therefore no memory reclamation system is running. Therefore, in order to prevent this feature from being used when such requests occur, we remove the __GFP_RECLAIM, __GFP_IO, and __GFP_FS bits from the gfp flag.
  - During boot, GFP_BOOT_MASK is assigned to the global variable gfp_allowed_mask.
    - GFP_BOOT_MASK __GFP_RECLAIM | __GFP_IO | It contains beats with the __GFP_FS removed.
- In line 20~23 of the code, prepare the page assignment by adding the required assignment context and the required assignment flags before attempting to assign the page.
- Finally, on line 25 of code, prepare an additional member of the alloc context.
- In line 31 of code, add an alloc flag such as nofragment to the zone and gfp mask request.
- In line 34~36 of the code, try to allocate the fast-path page for the first time.
- If fast-path allocation fails in code lines 44~54, disable dirty zone balancing and try slow-path allocation.
- In code lines 56~61, the out: label is. Returns the allocated page, but abandons the allocation if it goes beyond the limit of the memory control group.

Preparing the Assignment Context

prepare_alloc_pages()

mm/page_alloc.c

```
01 static inline bool prepare_alloc_pages(gfp_t gfp_mask, unsigned int order,
02                 int preferred_nid, nodemask_t *nodemask,
03                 struct alloc_context *ac, gfp_t *alloc_mask,
04                 unsigned int *alloc_flags)
05 {
06         ac->high_zoneidx = gfp_zone(gfp_mask);
07         ac->zonelist = node_zonelist(preferred_nid, gfp_mask);
08         ac->nodemask = nodemask;
09         ac->migratetype = gfpflags_to_migratetype(gfp_mask);
10 
11         if (cpusets_enabled()) {
12                 *alloc_mask |= __GFP_HARDWALL;
13                  if (!ac->nodemask)
14                          ac->nodemask = &cpuset_current_mems_allowed;
15                  else
16                          *alloc_flags |= ALLOC_CPUSET;
17          }
18
19          fs_reclaim_acquire(gfp_mask);
20          fs_reclaim_release(gfp_mask);
21
22          might_sleep_if(gfp_mask & __GFP_DIRECT_RECLAIM);
23
24          if (should_fail_alloc_page(gfp_mask, order))
25                  return false;
26
27          if (IS_ENABLED(CONFIG_CMA) && ac->migratetype == MIGRATE_MOVABLE)
28                  *alloc_flags |= ALLOC_CMA;
29
30          return true;
31  }
```

Before attempting to allocate a page, we gather the values to pass to each subfunction and prepare it in a ac_context structure. Then, add flags to the input/output factor @alloc_flags if necessary. Returns false only if the allocation fails by force for debugging purposes.

- In line 6 of code, we know the zone index that corresponds to @gfp_mask.
- In line 7 of code, select one of the two zonelists based on the @flags value from the @nid node and return it.
    - See: build_all_zonelists() (http://jake.dothome.co.kr/build_all_zonelists) | Qc
- In line 8~9 of the code, get the node mask and the migration type to be assigned from the gfp flag.
- If you are using the control group's cpuset in lines 11~17, add a hardwall flag to the alloc_mask so that the requested task is assigned with any restrictions specified in the cgroup's current cpuset directory settings. This restriction will be excluded from future GFP_ATOMIC such allocations. In addition, depending on whether the node mask is specified in the allocation function, it is divided as follows.
    - Nodes specified in the nodemask requested by the hardwall + allocation function
    - If you don't have the node mask requested by the hardwall + allocation function, the nodes specified by the task
- If line 22 of the code allows a direct recall, perform a preempt point.
    - Direct recall is allowed for general user and kernel memory allocation requests.
- In line 24~25 of the code, you can create a failure situation for debugging purposes.
- In line 27~28 of code, the use of the cma area is allowed if the page is movable.

finalise_ac()

mm/page_alloc.c

```
01  /* Determine whether to spread dirty pages and what the first usable zone */
02  static inline void finalise_ac(gfp_t gfp_mask, struct alloc_context *ac)
03  {
04          /* Dirty zone balancing only done in the fast path */
05          ac->spread_dirty_pages = (gfp_mask & __GFP_WRITE);
06
07          /*
08           * The preferred zone is used for statistics but crucially it is
09           * also used as the starting point for the zonelist iterator. It
10           * may get reset for allocations that ignore memory policies.
11           */
12          ac->preferred_zoneref = first_zones_zonelist(ac->zonelist,
13                                          ac->high_zoneidx, ac->nodemask);
14  }
```

Finish by preparing additional members of the alloc context.

- In line 5 of the code, if there is a __GFP_WRITE request in the gfp flag, only the fastpath page assignment uses dirty zone balancing
- In lines 12~13 of the code, the first available zone of the current zonelist is stored in the preferred_zoneref. This value will be used later in the statistics. Also, if there is no first zone, the page cannot be allocated, so it goes to the out label and exits the function. If ac.nodemask is NULL because it is not specified, use the nodemask specified by cpuset for the current task.

alloc_flags_nofragment()

mm/page_alloc.c

```
/*
 * The restriction on ZONE_DMA32 as being a suitable zone to use to avoid
 * fragmentation is subtle. If the preferred zone was HIGHMEM then
 * premature use of a lower zone may cause lowmem pressure problems that
 * are worse than fragmentation. If the next zone is ZONE_DMA then it is
 * probably too small. It only makes sense to spread allocations to avoid
 * fragmentation between the Normal and DMA32 zones.
 */
```

```
static inline unsigned int
alloc_flags_nofragment(struct zone *zone, gfp_t gfp_mask)
{
        unsigned int alloc_flags = 0;

        if (gfp_mask & __GFP_KSWAPD_RECLAIM)
                alloc_flags |= ALLOC_KSWAPD;

#ifdef CONFIG_ZONE_DMA32
        if (zone_idx(zone) != ZONE_NORMAL)
                goto out;

        /*
         * If ZONE_DMA32 exists, assume it is the one after ZONE_NORMAL and
         * the pointer is within zone->zone_pgdat->node_zones[]. Also assume
         * on UMA that if Normal is populated then so is DMA32.
         */
        BUILD_BUG_ON(ZONE_NORMAL - ZONE_DMA32 != 1);
        if (nr_online_nodes > 1 && !populated_zone(--zone))
                goto out;

out:
#endif /* CONFIG_ZONE_DMA32 */
        return alloc_flags;
}
```

In response to the zone and gfp mask requests, add an alloc flag such as nofragment.

- If a __GFP_KSWAPD_RECLAIM is requested with the gfp flag in line 6~7 of the code, add a ALLOC_KSWAPD in the alloc flag so that kswapd can be woken up in case of low memory.
- In line 9~23 of code, if you are using both DMA32 zone and normal zone, add the ALLOC_NOFRAGMENT flag when you request the allocation of the normal zone. (Only 5.0 code is buggy)
    - Due to a bug that does not add the ALLOC_NOFRAGMENT flag, it has been patched in kernel v5.1-rc7.
        - Note: mm/page_alloc.c: fix never set ALLOC_NOFRAGMENT flag (https://github.com/torvalds/linux/commit/8118b82eb756e271929697e8ada5f637dc443af1#diff-d1793bdc2ce9e21814061bb882d0c3ac)

ALLOC_NOFRAGMENT

- If there is not enough memory when allocating the requested migratetype, it will steal it from another type (fallback migratetype), but it will only steal more than 1 page block to limit the mixing of other migratetypes in the page block. This excludes fragment elements between migratetypes.

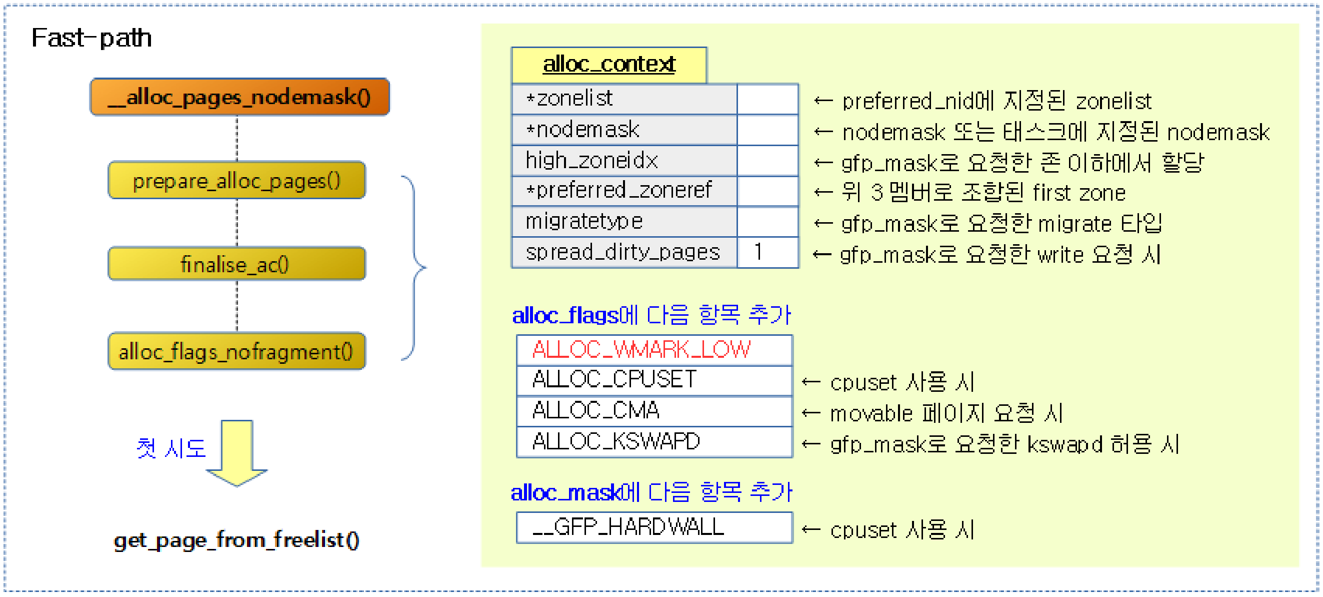
Fastpath page assignment

The following function is used to allocate a page by calling both fastpath and slowpath, and when it is called from fastpath, it is called by adding a __GFP_HARDWALL to the argument gfp_mask.

- GFP_KERNEL
    - (__GFP_RECLAIM | __GFP_IO | __GFP_FS)
    - __GFP_RECLAIM has two flags, ___GFP_DIRECT_RECLAIM and ___GFP_KSWAPD_RECLAIM.
- GFP_USER
    - (__GFP_RECLAIM | __GFP_IO | __GFP_FS | __GFP_HARDWALL)
    - Set the __GFP_HARDWALL when requesting an allocation of pages in user space so that they are not allowed to be allocated outside of the memory node that the current task's cpuset allows.

The following figure shows the separation between the Fastpath routines used in page allocation and the functions used in the Slowpath routines.

- However, if the get_page_from_freelist() function is called from the __alloc_pages_slowpath() function, it is part of the Slowpath.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/alloc_pages-2b.png)

get_page_from_freelist()

mm/page_alloc.c -1/2-

```
/*
 * get_page_from_freelist goes through the zonelist trying to allocate
 * a page.
 */
```

```
static struct page *
get_page_from_freelist(gfp_t gfp_mask, unsigned int order, int alloc_flags,
                                                const struct alloc_context *ac)
{
        struct zoneref *z;
        struct zone *zone;
        struct pglist_data *last_pgdat_dirty_limit = NULL;
        bool no_fallback;

retry:
        /*
         * Scan zonelist, looking for a zone with enough free.
         * See also __cpuset_node_allowed() comment in kernel/cpuset.c.
         */
        no_fallback = alloc_flags & ALLOC_NOFRAGMENT;
        z = ac->preferred_zoneref;
        for_next_zone_zonelist_nodemask(zone, z, ac->zonelist, ac->high_zoneidx,
                                                                ac->nodemask) {
                struct page *page;
                unsigned long mark;

                if (cpusets_enabled() &&
                        (alloc_flags & ALLOC_CPUSET) &&
                        !__cpuset_zone_allowed(zone, gfp_mask))
                                continue;
                /*
                 * When allocating a page cache page for writing, we
                 * want to get it from a node that is within its dirty
                 * limit, such that no single node holds more than its
                 * proportional share of globally allowed dirty pages.
                 * The dirty limits take into account the node's
```

```
32                   * lowmem reserves and high watermark so that kswapd
33                   * should be able to balance it without having to
34                   * write pages from its LRU list.
35                   *
36                   * XXX: For now, allow allocations to potentially
37                   * exceed the per-node dirty limit in the slowpath
38                   * (spread_dirty_pages unset) before going into reclaim,
39                   * which is important when on a NUMA setup the allowed
40                   * nodes are together not big enough to reach the
41                   * global limit.  The proper fix for these situations
42                   * will require awareness of nodes in the
43                   * dirty-throttling and the flusher threads.
44                   */
45                  if (ac->spread_dirty_pages) {
46                          if (last_pgdat_dirty_limit == zone->zone_pgdat)
47                                  continue;
48
49                          if (!node_dirty_ok(zone->zone_pgdat)) {
50                                  last_pgdat_dirty_limit = zone->zone_pgdat;
51                                  continue;
52                          }
53                  }
54
55                  if (no_fallback && nr_online_nodes > 1 &&
56                      zone != ac->preferred_zoneref->zone) {
57                          int local_nid;
58
59                          /*
60                           * If moving to a remote node, retry but allow
61                           * fragmenting fallbacks. Locality is more important
62                           * than fragmentation avoidance.
63                           */
64                          local_nid = zone_to_nid(ac->preferred_zoneref->zone);
65                          if (zone_to_nid(zone) != local_nid) {
66                                  alloc_flags &= ~ALLOC_NOFRAGMENT;
67                                  goto retry;
68                          }
69                  }
```

Assign the requested 2^@order pages based on the allocation context information. If the assignment succeeds, it returns the page, and if it fails, it returns null.

- In line 15 of the code, the assignment flag specifies whether to no_fallback the ALLOC_NOFRAGMENT assignment flag if it is present so that it is not fragmented within the page block on the first allocation attempt.
  - Within a page block, several mobility types can be mixed.
  - In order to avoid fragmentation, if there is enough memory, it is better to induce one type of mobility in each page block.
- Traverse the nodes requested in lines 16~18 of the code and the zones of [Preferred Zone, high_zoneidx] in the zonelist in order.
- In lines 22~25 of the code, skip to exclude the current task if it does not support zones supported by the control group's cpuset.
- In line 45~53 of the code, the dirty limit is limited for each node. Unless the dirty limit of all nodes has been exceeded, the nodes that have exceeded the dirty limit will be skipped. When assigning

a slowpath, the spread_dirty_pages value is called false and is not subject to the dirty limit restriction.

- When allocating a page in line 55~69 of code, it is more important to allocate it from the local node than to request a nofragment from the remote node. If you have a nofragment request on a system with two or more nodes, but you need to allocate it on another node, remove the nofragment request and move it to the retry label so that it can be allocated on the local node.

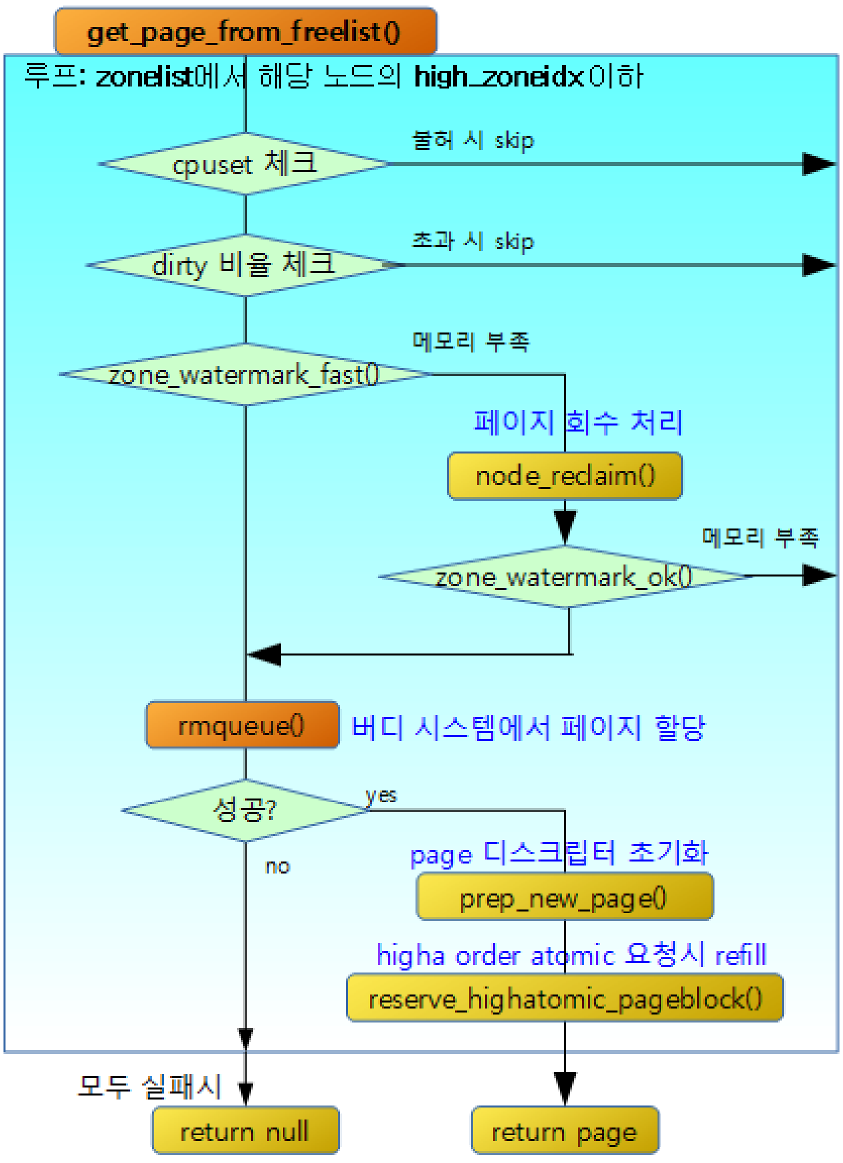
mm/page_alloc.c -2/2-

```
                mark = wmark_pages(zone, alloc_flags & ALLOC_WMARK_MASK);
                if (!zone_watermark_fast(zone, order, mark,
                                       ac_classzone_idx(ac), alloc_flags)) {
                        int ret;

#ifdef CONFIG_DEFERRED_STRUCT_PAGE_INIT
                        /*
                         * Watermark failed for this zone, but see if we can
                         * grow this zone if it contains deferred pages.
                         */
                        if (static_branch_unlikely(&deferred_pages)) {
                                if (_deferred_grow_zone(zone, order))
                                        goto try_this_zone;
                        }
#endif
                        /* Checked here to keep the fast path fast */
                        BUILD_BUG_ON(ALLOC_NO_WATERMARKS < NR_WMARK);
                        if (alloc_flags & ALLOC_NO_WATERMARKS)
                                goto try_this_zone;

                        if (node_reclaim_mode == 0 ||
                            !zone_allows_reclaim(ac->preferred_zoneref->zone, zone))
                                continue;

                        ret = node_reclaim(zone->zone_pgdat, gfp_mask, order);
                        switch (ret) {
                        case NODE_RECLAIM_NOSCAN:
                                /* did not scan */
                                continue;
                        case NODE_RECLAIM_FULL:
                                /* scanned but unreclaimable */
                                continue;
                        default:
                                /* did we reclaim enough */
                                if (zone_watermark_ok(zone, order, mark,
                                                ac_classzone_idx(ac), alloc_flags))
                                        goto try_this_zone;

                                continue;
                        }
                }

try_this_zone:
                page = rmqueue(ac->preferred_zoneref->zone, zone, order,
                                gfp_mask, alloc_flags, ac->migratetype);
                if (page) {
                        prep_new_page(page, order, gfp_mask, alloc_flags);
48
49                         /*
50                          * If this is a high-order atomic allocation then check
51                          * if the pageblock should be reserved for the future
52                          */
53                         if (unlikely(order && (alloc_flags & ALLOC_HARDER)))
54                                 reserve_highatomic_pageblock(page, zone, order);
55
56                         return page;
57                 } else {
58 #ifdef CONFIG_DEFERRED_STRUCT_PAGE_INIT
59                         /* Try again if zone has deferred pages */
60                         if (static_branch_unlikely(&deferred_pages)) {
61                                 if (_deferred_grow_zone(zone, order))
62                                         goto try_this_zone;
63                         }
64 #endif
65                 }
66         }
67
68         /*
69          * It's possible on a UMA machine to get through all zones that are
70          * fragmented. If avoiding fragmentation, reset and try again.
71          */
72         if (no_fallback) {
73                 alloc_flags &= ~ALLOC_NOFRAGMENT;
74                 goto retry;
75         }
76
77         return NULL;
78 }
```

- In line 1~3 of the code, the number of free pages remaining is roughly estimated for quick output, and the number of low, low, and min watermarks requested by the allocation flag is below the standard of insufficient memory.
- In code lines 11~14, the initialization of some memory is suspended during bootup on high-memory systems such as x86 systems. If this is the case and the pages in the current zone are not initialized, then it is not really out of memory, so it will try to allocate them in this zone.
- Even if the watermark criteria are not set in line 18~19 of the code, try to allocate in this zone.
- In lines 21~23 of code, the "/sys/vm/zone_reclaim_mode" setting is 0 (default) or the allocation is skipped in this zone unless it is a local or nearby remote node.
- If you didn't scan for page retrieval for a node on line 25~32 of code, or if you have already done a full scan and it is no longer working, skip this zone.
- If there are some or successful page reclamations for nodes in line 33~40 of the code, we will once again accurately estimate them and compare them with the number of free pages remaining and one of the three high, low, and min watermark values requested by the allocation flag, and if the threshold is exceeded and not out of memory, we will attempt to allocate in this zone. Otherwise, if the memory is still low, skip this zone.
- In line 43~45 of the code, try_this_zone: In the label, we try to assign the order page through the actual buddy system.
- If memory is normally allocated in code lines 46~56, it will prepare for a new page structure and return that page.

- If the initialization is deferred on code lines 57~65, try to allocate pages in this zone again.
- If the first allocation attempt fails in code lines 72~75, remove the nofragment attribute and try again so that the allocation can be made even if the migrate type is different in the block.
- If the second allocation attempt at line 77 also fails, it returns null.

The following figure shows the process processed by the get_page_from_freelist() function.



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

zone_reclaim_mode

- In a kernel that supports NUMA systems, set it to the "proc/sys/fs/zone_reclaim_mode" file value
  - RECLAIM_OFF(0)
    - If the watermark standard is lower, skip to the next zone without retrieving the current zone.
  - RECLAIM_ZONE(1)
    - inactive LRU list.
  - RECLAIM_WRITE(2)
    - The modified file page is recalled through a writeout process.
  - RECLAIM_UNMAP(3)

- The mapped file page is reclaimed through the unmap process.

zone_allows_reclaim()

mm/page_alloc.c

```
static bool zone_allows_reclaim(struct zone *local_zone, struct zone *zone)
{
        return node_distance(zone_to_nid(local_zone), zone_to_nid(zone)) <
                                RECLAIM_DISTANCE;
}
```

Returns true if the local_zone and the request zone are within RECLAIM_DISTANCE (30) of each other.

- Page recall is only possible in the nearest remote zone.

Initialize after assigning a new page

prep_new_page()

page_alloc.c

```
01 static void prep_new_page(struct page *page, unsigned int order, gfp_t gfp_flags,
02                                                         unsigned int alloc_flags)
03 {
04         int i;
05 
06         post_alloc_hook(page, order, gfp_flags);
07 
08         if (!free_pages_prezeroed() && (gfp_flags & __GFP_ZERO))
09                 for (i = 0; i < (1 << order); i++)
10                         clear_highpage(page + i);
11 
12         if (order && (gfp_flags & __GFP_COMP))
13                 prep_compound_page(page, order);
14 
15         /*
16          * page is set pfmemalloc when ALLOC_NO_WATERMARKS was necessary to
17          * allocate the page. The expectation is that the caller is taking
18          * steps that will free more memory. The caller should avoid the page
19          * being used for !PFMEMALLOC purposes.
20          */
21         if (alloc_flags & ALLOC_NO_WATERMARKS)
22                 set_page_pfmemalloc(page);
23         else
24                 clear_page_pfmemalloc(page);
25 }
```

Initializes all page descriptors corresponding to the allocated 2^order page size memory.

- Initialize the first page descriptor that corresponds to the 6^order page size memory allocated in line 2.

- If zero initialization request is received in code lines 8~10, all allocated memory will be reset to zero.
    - Memories in the highmem area of a 32-bit system are temporarily mapped, initialized to zero, and then re-mapped.
- In line 12~13 of the code, initialize the page descriptors to compound pages if they are compound pages.
    - All tail pages point to the head page.
- If the assignment is requested on the basis of no watermark in line 21~24 of the code, the index member of the page descriptor should be assigned with -1 to indicate that it has been assigned in the pfmemalloc state.

post_alloc_hook()

mm/page_alloc.c

```
01 inline void post_alloc_hook(struct page *page, unsigned int order,
02                                 gfp_t gfp_flags)
03 {
04         set_page_private(page, 0);
05         set_page_refcounted(page);
06 
07         arch_alloc_page(page, order);
08         kernel_map_pages(page, 1 << order, 1);
09         kernel_poison_pages(page, 1 << order, 1);
10         kasan_alloc_pages(page, order);
11         set_page_owner(page, order, gfp_flags);
12 }
```

Initializes the first page descriptor that corresponds to the allocated 2^order page size memory.

- In line 4 of code, initialize the private member of the page descriptor to 0.
- In line 5 of code, initialize the reference counter to 1.
- In line 7 of code, call the page assignment hook that corresponds to the architecture.
    - ARM and ARM64 don't have a corresponding calling function.
- Line 8 of the code calls page allocation for debugging.
- Perform debugging with poison on line 9 of code. Check the allocated memory for pre-marked poisons.
- In line 10 of the code, KASAN is called for debugging.
- In line 11 of the code, we call it to track the page owner for debugging.

CPUSET Related

cpuset_zone_allowed()

include/linux/cpuset.h

```
static inline int cpuset_zone_allowed(struct zone *z, gfp_t gfp_mask)
{
        if (cpusets_enabled())
                return __cpuset_zone_allowed(z, gfp_mask);
        return true;
}
```

Returns true if the node in the request zone is a node that is currently supported by the CPU. Otherwise, if the interrupt was called, if the __GFP_THISNODE flag is set, if the current task already allows it, if the task has a TIF_MEMDIE flag or PF_EXITING flag set, then returns true, and if __GFP_HARDWALL is set, it returns false to prevent a chance from giving a chance to a node other than the memory node allowed by the current task's cpuset.

cpuset_node_allowed()

include/linux/cpuset.h

```
static inline int cpuset_node_allowed(int node, gfp_t gfp_mask)
{
        return __cpuset_node_allowed(zone_to_nid(z), gfp_mask);
}
```

Call the function below

__cpuset_node_allowed()

kernel/cpuset.c

```
/**
 * cpuset_node_allowed - Can we allocate on a memory node?
 * @node: is this an allowed node?
 * @gfp_mask: memory allocation flags
 *
 * If we're in interrupt, yes, we can always allocate.  If @node is set in
 * current's mems_allowed, yes.  If it's not a __GFP_HARDWALL request and this
 * node is set in the nearest hardwalled cpuset ancestor to current's cpuset,
 * yes.  If current has access to memory reserves as an oom victim, yes.
 * Otherwise, no.
 *
 * GFP_USER allocations are marked with the __GFP_HARDWALL bit,
 * and do not allow allocations outside the current tasks cpuset
 * unless the task has been OOM killed.
 * GFP_KERNEL allocations are not so marked, so can escape to the
 * nearest enclosing hardwalled ancestor cpuset.
 *
 * Scanning up parent cpusets requires callback_lock.  The
 * __alloc_pages() routine only calls here with __GFP_HARDWALL bit
 * _not_ set if it's a GFP_KERNEL allocation, and all nodes in the
 * current tasks mems_allowed came up empty on the first pass over
 * the zonelist.  So only GFP_KERNEL allocations, if all nodes in the
 * cpuset are short of memory, might require taking the callback_lock.
 *
 * The first call here from mm/page_alloc:get_page_from_freelist()
 * has __GFP_HARDWALL set in gfp_mask, enforcing hardwall cpusets,
 * so no allocation on a node outside the cpuset is allowed (unless
 * in interrupt, of course).
 *
 * The second pass through get_page_from_freelist() doesn't even call
 * here for GFP_ATOMIC calls.  For those calls, the __alloc_pages()
 * variable 'wait' is not set, and the bit ALLOC_CPUSET is not set
 * in alloc_flags.  That logic and the checks below have the combined
 * affect that:
 *      in_interrupt - any node ok (current task context irrelevant)
```

```
36   *      GFP_ATOMIC    - any node ok
37   *      tsk_is_oom_victim    - any node ok
38   *      GFP_KERNEL    - any node in enclosing hardwalled cpuset ok
39   *      GFP_USER      - only nodes in current tasks mems allowed ok.
40   */
```

```
01  bool __cpuset_node_allowed(int node, gfp_t gfp_mask)
02  {
03          struct cpuset *cs;              /* current cpuset ancestors */
04          int allowed;                    /* is allocation in zone z allowed? */
05          unsigned long flags;
06  
07          if (in_interrupt())
08                  return true;
09          if (node_isset(node, current->mems_allowed))
10                  return true;
11          /*
12           * Allow tasks that have access to memory reserves because they have
13           * been OOM killed to get memory anywhere.
14           */
15          if (unlikely(tsk_is_oom_victim(current)))
16                  return true;
17          if (gfp_mask & __GFP_HARDWALL)  /* If hardwall request, stop here */
18                  return false;
19  
20          if (current->flags & PF_EXITING) /* Let dying task have memory */
21                  return true;
22  
23          /* Not hardwall and node outside mems_allowed: scan up cpusets */
24          spin_lock_irqsave(&callback_lock, flags);
25  
26          rcu_read_lock();
27          cs = nearest_hardwall_ancestor(task_cs(current));
28          allowed = node_isset(node, cs->mems_allowed);
29          rcu_read_unlock();
30  
31          spin_unlock_irqrestore(&callback_lock, flags);
32          return allowed;
33  }
```

Returns true if the requesting node is a node currently supported by the CPU. Otherwise, if the interrupt was called, if the __GFP_THISNODE flag is set, if the current task already allows it, if the task has a TIF_MEMDIE flag or PF_EXITING flag set, then returns true, and if __GFP_HARDWALL is set, it returns false to prevent a chance from giving a chance to a node other than the memory node allowed by the current task's cpuset.

- Returns true if called from an interrupt handler on lines 7~8 of code.
- On lines 9~10 of the code, it returns true if the node is allowed by the current task.
- On lines 15~16 of code, if there is a small probability that the current task is being terminated due to insufficient memory, it will return true.
- On lines 17~18 of the code, it returns false if it is a hardwall request.
- On lines 20~21 of code, return true if the current task is terminating.
- There is no hardwall request in code lines 24~32, and the node is not allowed by the current task. In this case, it finds out which hardwall parent cpuset is closest to the current task's cpuset and returns whether the cpuset has allowed memory nodes.

There are three ways to use the __GFP_HARDWALL flag:

- FastPath Page Allocation Request
- Requesting a Page Assignment from a User Task
- Requesting a slab page assignment

When the kernel requests memory allocation, it does not use __GFP_HARDWALL flags, except in special cases, such as when allocating for slab pages.

- If you don't use the __GFP_HARDWALL flag, use cgroup.
- In the cgroup's cpuset subsystem, the group that contains the current task is used to find and use the nearest parent group with hardwall or exclusive settings in the parent direction.
- By changing the values of cpuset.mem_exclusive and cpuset.mem_hardwall in the cgroup's CPUSet subsystem to 1 each, the hardwall and exclusive of the group are set.
- The geometry of all subsystems in the cgroup is managed in a tree structure, and the children inherit the values of the parents without setting any specific values. If you use the hardwall feature, you create a wall between your own group and your parent group

nearest_hardwall_ancestor()

kernel/cpuset.c

```
/*
 * nearest_hardwall_ancestor() - Returns the nearest mem_exclusive or
 * mem_hardwall ancestor to the specified cpuset.  Call holding
 * callback_lock.  If no ancestor is mem_exclusive or mem_hardwall
 * (an unusual configuration), then returns the root cpuset.
 */
static struct cpuset *nearest_hardwall_ancestor(struct cpuset *cs)
{
        while (!(is_mem_exclusive(cs) || is_mem_hardwall(cs)) && parent_cs(cs))
                cs = parent_cs(cs);
        return cs;
}
```

Returns cpuset if cpuset is using memory beta or if the parent cpuset is hardwall. If the conditions are not satisfied, the parent CPUSET will continue to be searched until it is satisfied.

read_mems_allowed_begin()

include/linux/cpuset.h

```
/*
 * read_mems_allowed_begin is required when making decisions involving
 * mems_allowed such as during page allocation. mems_allowed can be updated in
 * parallel and depending on the new value an operation can fail potentially
 * causing process failure. A retry loop with read_mems_allowed_begin and
 * read_mems_allowed_retry prevents these artificial failures.
 */
```

```
static inline unsigned int read_mems_allowed_begin(void)
{
        if (!static_branch_unlikely(&cpusets_pre_enable_key))
                return 0;
        return read_seqcount_begin(&current->mems_allowed_seq);
}
```

Knows the mems_allowed_seq sequence lock value of the current task.

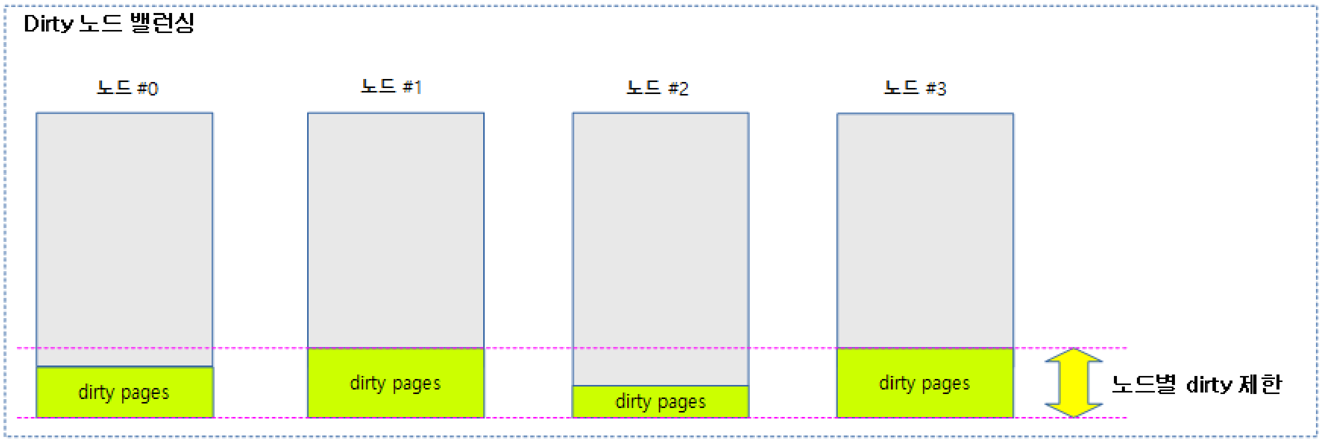
- If the cpuset setting for the current task is changed (the setting in the /sys/fs/cpuset directory), the current->mems_allowed_seq value is changed.

Balancing Dirty Nodes

Dirty (a state in which files are written to a file but are written to a lazy state that resides in the file cache memory) limit, and if the dirty limit is exceeded for the node that requested the file history, it is assigned to another node to distribute the dirty files.

- If the allocation fails because the dirty limit is exceeded on all nodes, simply loosen the dirty limit and try again to allocate.

The following figure shows how the dirty page allocation requests are operated without exceeding the 20% dirty limit.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/zone_dirty_limit-2a.png)

node_dirty_ok()

mm/page-writeback.c

```
/**
 * node_dirty_ok - tells whether a node is within its dirty limits
 * @pgdat: the node to check
 *
 * Returns %true when the dirty pages in @pgdat are within the node's
 * dirty limit, %false if the limit is exceeded.
 */
```

```
bool node_dirty_ok(struct pglist_data *pgdat)
```

```
{
        unsigned long limit = node_dirty_limit(pgdat);
        unsigned long nr_pages = 0;

        nr_pages += node_page_state(pgdat, NR_FILE_DIRTY);
        nr_pages += node_page_state(pgdat, NR_UNSTABLE_NFS);
        nr_pages += node_page_state(pgdat, NR_WRITEBACK);

        return nr_pages <= limit;
}
```

Returns whether the requested node is below the dirty limit. 1=Dirty below limit, 0=Dirty above limit

- Returns true if the node's NR_FILE_DRITY + NR_UNSTABLE_NFS + NR_WRITEBACK pages are below the node's dirty(write buffer) limit.
- To check the numbers for the zone counters, see

You can check the counter information for each node using the "cat /proc/zoneinfo" command as follows:

- Reason for zoneinfo file and not nodeinfo file: Previously, it was not a node-specific information, but a zone-specific information.

```
$ cat /proc/zoneinfo
Node 0, zone    DMA32
  per-node stats
      nr_inactive_anon 2162
      nr_active_anon 4186
      nr_inactive_file 8415
      nr_active_file 5303
      nr_unevictable 0
      nr_slab_reclaimable 3490
      nr_slab_unreclaimable 6347
      nr_isolated_anon 0
      nr_isolated_file 0
      workingset_nodes 0
      workingset_refault 0
      workingset_activate 0
      workingset_restore 0
      workingset_nodereclaim 0
      nr_anon_pages 4141
      nr_mapped    6894
      nr_file_pages 15924
      nr_dirty     91                      <-----
      nr_writeback 0                       <-----
      nr_writeback_temp 0
      nr_shmem     2207
      nr_shmem_hugepages 0
      nr_shmem_pmdmapped 0
      nr_anon_transparent_hugepages 0
      nr_unstable  0                       <-----
      nr_vmscan_write 0
      nr_vmscan_immediate_reclaim 0
      nr_dirtied   330
      nr_written   239
      nr_kernel_misc_reclaimable 0
  pages free     634180
        min      5632
        low      7040
        high     8448
        spanned  786432
        present  786432
        managed  765785
        protection: (0, 0, 0)
      nr_free_pages 634180
      nr_zone_inactive_anon 2162
      nr_zone_active_anon 4186
      nr_zone_inactive_file 8415
      nr_zone_active_file 5303
      nr_zone_unevictable 0
      nr_zone_write_pending 91
      nr_mlock     0
      nr_page_table_pages 233
      nr_kernel_stack 1472
      nr_bounce    0
      nr_free_cma  8128
      numa_hit     97890
      numa_miss    0
      numa_foreign 0
      numa_interleave 6202
      numa_local   97890
      numa_other   0
  pagesets
    cpu: 0
              count: 320
              high:  378
              batch: 63
  vm stats threshold: 24
    cpu: 1
              count: 275
              high:  378
              batch: 63
  vm stats threshold: 24
  node_unreclaimable:  0
  start_pfn:           262144
Node 0, zone   Normal
  pages free     0
        min      0
        low      0
        high     0
        spanned  0
        present  0
        managed  0
        protection: (0, 0, 0)
Node 0, zone  Movable
  pages free     0
        min      0
        low      0
        high     0
        spanned  0
        present  0
        managed  0
        protection: (0, 0, 0)
```

node_dirty_limit()

mm/page-writeback.c

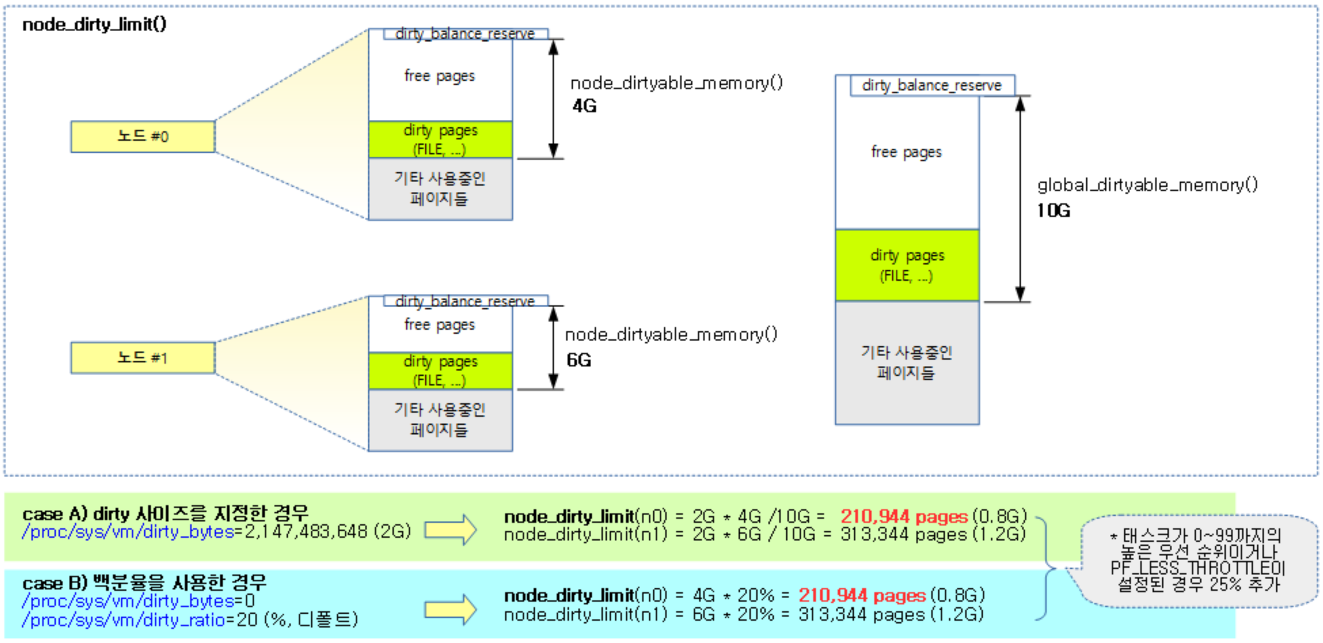
```
/**
 * node_dirty_limit - maximum number of dirty pages allowed in a node
 * @pgdat: the node
 *
 * Returns the maximum number of dirty pages allowed in a node, based
 * on the node's dirtyable memory.
 */
```

```
static unsigned long node_dirty_limit(struct pglist_data *pgdat)
{
        unsigned long node_memory = node_dirtyable_memory(pgdat);
        struct task_struct *tsk = current;
        unsigned long dirty;

        if (vm_dirty_bytes)
                dirty = DIV_ROUND_UP(vm_dirty_bytes, PAGE_SIZE) *
                        node_memory / global_dirtyable_memory();
        else
                dirty = vm_dirty_ratio * node_memory / 100;

        if (tsk->flags & PF_LESS_THROTTLE || rt_task(tsk))
                dirty += dirty / 4;

        return dirty;
}
```

Returns the number of pages limited by a certain percentage of the number of pages allowed on the node. If the task has a PF_LESS_THROTTLE or a task with a higher priority than the user task, add 25%.

- If vm_dirty_bytes is set, assign the percentage of dirty pages used by the node.
- If no vm_dirty_bytes is set, vm_dirty_ratio assigns as a percentage.
- Task Priority
  - rt_task of 0~139 have a high priority of 100 or less.
  - 100~139 has the priority of user tasks.
  - The lower number has the highest priority.

The figure below shows how the node_dirty_limit() value changes when using dirty_bytes or dirty_ratio.


node_dirty_limit()
노드 #0
노드 #1
dirty_balance_reserve
free pages
dirty pages (FILE, ...)
기타 사용중인 페이지들
node_dirtyable_memory() 4G
node_dirtyable_memory() 6G
global_dirtyable_memory() 10G
case A) dirty 사이즈를 지정한 경우
/proc/sys/vm/dirty_bytes=2,147,483,648 (2G)
node_dirty_limit(n0) = 2G * 4G /10G = 210,944 pages (0.8G)
node_dirty_limit(n1) = 2G * 6G / 10G = 313,344 pages (1.2G)
case B) 백분율을 사용한 경우
/proc/sys/vm/dirty_bytes=0
/proc/sys/vm/dirty_ratio=20 (%, 디폴트)
node_dirty_limit(n0) = 4G * 20% = 210,944 pages (0.8G)
node_dirty_limit(n1) = 6G * 20% = 313,344 pages (1.2G)
* 태스크가 0~99까지의 높은 우선 순위이거나 PF_LESS_THROTTLE이 설정된 경우 25% 추가

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

node_dirtyable_memory()

mm/page-writeback.c

```
/*
 * In a memory zone, there is a certain amount of pages we consider
 * available for the page cache, which is essentially the number of
 * free and reclaimable pages, minus some zone reserves to protect
 * lowmem and the ability to uphold the zone's watermarks without
 * requiring writeback.
 *
 * This number of dirtyable pages is the base value of which the
 * user-configurable dirty ratio is the effective number of pages that
 * are allowed to be actually dirtied.  Per individual zone, or
 * globally by using the sum of dirtyable pages over all zones.
 *
 * Because the user is allowed to specify the dirty limit globally as
 * absolute number of bytes, calculating the per-zone dirty limit can
 * require translating the configured limit into a percentage of
 * global dirtyable memory first.
 */

/**
 * node_dirtyable_memory - number of dirtyable pages in a node
 * @pgdat: the node
 *
 * Returns the node's number of pages potentially available for dirty
 * page cache.  This is the base value for the per-node dirty limits.
 */
```

```
01 static unsigned long node_dirtyable_memory(struct pglist_data *pgdat)
02 {
03         unsigned long nr_pages = 0;
04         int z;
05 
06         for (z = 0; z < MAX_NR_ZONES; z++) {
07                 struct zone *zone = pgdat->node_zones + z;
08 
09                 if (!populated_zone(zone))
10                         continue;
11 
12                 nr_pages += zone_page_state(zone, NR_FREE_PAGES);
13         }
14 
15         /*
16          * Pages reserved for the kernel should not be considered
17          * dirtyable, to prevent a situation where reclaim has to
18          * clean pages in order to balance the zones.
19          */
20         nr_pages -= min(nr_pages, pgdat->totalreserve_pages);
21 
22         nr_pages += node_page_state(pgdat, NR_INACTIVE_FILE);
23         nr_pages += node_page_state(pgdat, NR_ACTIVE_FILE);
24 
25         return nr_pages;
26 }
```

Returns the number of dirty possible pages for that node. (node's free page + used file cache page – totalreserve_pages)

- In code lines 6~13, we produce a free page of all the populate zones in the node.
- Exclude totalreserve_pages from the page calculated on line 20 of the code.
- In lines 22~23 of code, add the node's all (inactive+active) file cache page.

global_dirtyable_memory()

mm/page-writeback.c

```
/**
 * global_dirtyable_memory - number of globally dirtyable pages
 *
 * Returns the global number of pages potentially available for dirty
 * page cache.  This is the base value for the global dirty limits.
 */
```

```
static unsigned long global_dirtyable_memory(void)
{
        unsigned long x;

        x = global_zone_page_state(NR_FREE_PAGES);
        /*
```

```
         * Pages reserved for the kernel should not be considered
         * dirtyable, to prevent a situation where reclaim has to
         * clean pages in order to balance the zones.
         */
        x -= min(x, totalreserve_pages);

        x += global_node_page_state(NR_INACTIVE_FILE);
        x += global_node_page_state(NR_ACTIVE_FILE);

        if (!vm_highmem_is_dirtyable)
                x -= highmem_dirtyable_memory(x);

        return x + 1;   /* Ensure that we never return 0 */
}
```

Returns the number of pages that can be dirty in the system. (system's free page + file cache page – totalreserve_pages)

- Free page + file page being used as cache – returns dirty_balance_reserve value.
- If you have prevented highmem from being used as a dirty page, exclude the dirty page part of highmem.

highmem_dirtyable_memory()

mm/page-writeback.c

```
static unsigned long highmem_dirtyable_memory(unsigned long total)
{
#ifdef CONFIG_HIGHMEM
        int node;
        unsigned long x = 0;
        int i;

        for_each_node_state(node, N_HIGH_MEMORY) {
                for (i = ZONE_NORMAL + 1; i < MAX_NR_ZONES; i++) {
                        struct zone *z;
                        unsigned long nr_pages;

                        if (!is_highmem_idx(i))
                                continue;

                        z = &NODE_DATA(node)->node_zones[i];
                        if (!populated_zone(z))
                                continue;

                        nr_pages = zone_page_state(z, NR_FREE_PAGES);
                        /* watch for underflows */
                        nr_pages -= min(nr_pages, high_wmark_pages(z));
                        nr_pages += zone_page_state(z, NR_ZONE_INACTIVE_FILE);
                        nr_pages += zone_page_state(z, NR_ZONE_ACTIVE_FILE);
                        x += nr_pages;
                }
        }

        /*
         * Unreclaimable memory (kernel memory or anonymous memory
         * without swap) can bring down the dirtyable pages below
         * the zone's dirty balance reserve and the above calculation
         * will underflow.  However we still want to add in nodes
         * which are below threshold (negative values) to get a more
         * accurate calculation but make sure that the total never
         * underflows.
```

```
	 */
	if ((long)x < 0)
		x = 0;

	/*
	 * Make sure that the number of highmem pages is never larger
	 * than the number of the total dirtyable memory. This can only
	 * occur in very strange VM situations but we want to make sure
	 * that this does not occur.
	 */
	return min(x, total);
#else
	return 0;
#endif
}
```

Dirty pages about high memory get a possible number of (Don't use 64-bit systems.)

Structure

alloc_context Structure

mm/internal.h

```
/*
 * Structure for holding the mostly immutable allocation parameters pass
ed
 * between functions involved in allocations, including the alloc_pages*
 * family of functions.
 *
 * nodemask, migratetype and high_zoneidx are initialized only once in
 * __alloc_pages_nodemask() and then never change.
 *
 * zonelist, preferred_zone and classzone_idx are set first in
 * __alloc_pages_nodemask() for the fast path, and might be later change
d
 * in __alloc_pages_slowpath(). All other functions pass the whole struc
ure
 * by a const pointer.
 */
```

```
struct alloc_context {
	struct zonelist *zonelist;
	nodemask_t *nodemask;
	struct zone *preferred_zone;
	int migratetype;
	enum zone_type high_zoneidx;
	bool spread_dirty_pages;
};
```

It is a structure used in the alloc_pages* family of functions for the purpose of passing various parameters.

- zonelist
  - Zonelist used for page assignment
- nodemask
  - It is restricted to assignment only from the nodes specified in the zonelist.
  - If not specified, all nodes are targeted.
- preferred_zone

    - FastPath specifies the zone to be assigned first.
    - SlowPath specifies the first available zone in the ZoneList.
- migratetype
    - Migrate type type to assign
- high_zoneidx
    - It is limited to the zones in the zonelist that can only be assigned below the specified high zone.
- spread_dirty_pages
    - It is used as a dirty zone balancing or not.
        - Set to 1 on fastpath assignment request
        - Set to 0 on slowpath assignment request

consultation

- Zoned Allocator -1- (Physics Page Assignment - Fastpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath) | Sentence C – Current post
- 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) | Qc
- 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

- Tunable watermark (https://lwn.net/Articles/422291/) | LWN.net
- Memory Reallocations and Kernel Parameters (https://brunch.co.kr/@alden/14) | Jinwoo Kang
- Memory compaction (https://lwn.net/Articles/368869/) | LWN.net
- Memory Compaction (http://woodz.tistory.com/42) | Daeseok's Blog

- mm, compaction: introduce kcompactd (https://github.com/torvalds/linux/commit/698b1b30642f1ff0ea10ef1de9745ab633031377)
- Page migration (http://www.kernel.org/doc/Documentation/vm/page_migration) | www.kernel.org
- Understanding the Linux Kernel Chapter 16 Swapping – How to Release Memory | Choi Sungja of Hannam University – PPT Download (https://www.google.co.kr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&sqi=2&ved=0ahUKEwjc0NzHqb7NAhVDopQKHQr6BAkQFggoMAE&url=http%3A%2F%2Fnetwk.hannam.ac.kr%2Fdata%2Flinux_kernel%2FChap%252016.PPT&usg=AFQjCNFacbagZDFkysciNjE8IkAqb969zA&sig2=s_URw8j8S7qpa9aoqnIB2w&cad=rja)
- Controlling Memory Fragmentation and Higher Order Allocation Failure: Analysis, Observations and Results | Pintu Kumar – pdf download (http://elinux.org/images/a/a8/Controlling_Linux_Memory_Fragmentation_and_Higher_Order_Allocation_Failure-_Analysis,_Observations_and_Results.pdf)
- ZONE Bitmap (API) (http://jake.dothome.co.kr/zone-api) | Qc

7 thoughts to "Zoned Allocator -1- (Physics Page Assignment - Fastpath)"

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2022-04-02 13:22 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/#comment-306481)

Hello?

I've corrected the typo in what you said.
2pageblock_order -> 2^pageblock_order

I appreciate it. ^^

RESPONSE (/ZONNED-ALLOCATOR-ALLOC-PAGES-FASTPATH/?REPLYTOCOM=306481#RESPOND)

MINHO KIM
2022-09-29 15:54 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/#comment-307091)

Hi there,
thanks for the great write-up. It was very helpful.

While studying, I have a question, so I write an article.
First of all, my understanding is that if the dirty limit of the local node is exceeded during the fast path allocation process, it will try to allocate to the remote node first, but if this is correct,
then I wonder why the dirty page cache is even allocated to the remote node.

I'm a kernel newbie, so please forgive me if the question is awkward..!

I appreciate it.

RESPONSE (/ZONNED-ALLOCATOR-ALLOC-PAGES-FASTPATH/?REPLYTOCOM=307091#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2022-09-30 19:55 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/#comment-307094)

Hello? Minho Kim,

get_page_from_freelist() code description body has been modified by zone -> node.

When this feature was first developed, it was titled "MM: Try to Distribute Dirty Pages Fairly Across Zones" and was developed to enable zone balancing for
dirty pages. However, in a certain version of the kernel, a
new zone memory reclamation system was changed to a per-node system, and this code was also affected, controlling
the dirty limit on a per-node basis, which I couldn't explain to it.

Luckily, Kim Minho was very aware of it and asked me questions node by node. ^^;

The reason for using this feature is that if there are too many dirty pages
on one node when the memory runs out, the reclamation system will take a long time to reclaim the memory.
In other words, dirty pages are very slow because they can only be retrieved after they have been written to disk.

그래서 메모리가 부족하지기 전까지는 쓰기 가능한 메모리를 할당 요청할 때엔,
커널은 노드별로 dirty 페이지들을 분산하도록 노드 별 dirty limit을 초과하면 다른 노드를 선택하도록 합니다.

감사합니다.

응답 (/ZONNED-ALLOCATOR-ALLOC-PAGES-FASTPATH/?REPLYTOCOM=307094#RESPOND)

김민호
2022-10-01 16:58 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/#comment-307100)

아하 그렇군요, 이해하였습니다.
상세한 답변 감사드립니다.

응답 (/ZONNED-ALLOCATOR-ALLOC-PAGES-FASTPATH/?REPLYTOCOM=307100#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)

2022-10-04 08:54 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/#comment-307107)

민호님도 즐거운 하루되세요. 감사합니다.

응답 (/ZONNED-ALLOCATOR-ALLOC-PAGES-FASTPATH/?REPLYTOCOM=307107#RESPOND)

지나가는 과객.

2023-09-30 17:38 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/#comment-308407)

안녕하세요. 문영일님.

0x1234_5000 ~ 0x1236_2000 사이에 연속된 9개의 fee 페이지가 존재하지만,
=>0x1234_5000 ~ 0x1236_2000 사이에 연속된 9개의 free 페이지가 존재하지만,(fee에서 r이 빠진것 같습니다)

감사합니다.

응답 (/ZONNED-ALLOCATOR-ALLOC-PAGES-FASTPATH/?REPLYTOCOM=308407#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)

2023-10-04 07:59 (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-fastpath/#comment-308417)

오타 수정하였습니다. 감사합니다. ^^

응답 (/ZONNED-ALLOCATOR-ALLOC-PAGES-FASTPATH/?REPLYTOCOM=308417#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

pgtable_init() (http://jake.dothome.co.kr/pgtable_init/)

Zoned Allocator -2- (물리 페이지 할당-Slowpath) (http://jake.dothome.co.kr/zonned-allocator-alloc-pages-slowpath/)