Page Allocation

SWE3015

Jeaho Hwang

Page

- Unit of memory mapping
  - 4KB size by default
  - For every page there is ***struct page.***
    - 36B / 4KB = 0.88% overhead

```
struct page {
        unsigned long flags;
        struct address_space *mapping;
        pgoff_t index;
        atomic_t _mapcount;
        atomic_t _count;
        struct list_head lru;
        unsigned long private;
        void *virtual;
}
                    <include/linux/mm_types.h>
```

Page

- Flag field
  - Stores the status of the page
    - Dirty, locked, etc..
  - Enumeration type defined in include/linux/page-flags.h
- Count field
  - Reference count for the page
  - Accessed with wrapper function
    - Page_count()

```
static inline int page_count(struct page *page)
{
        return atomic_read(&compound_head(page)-
>_count);
}
```
<include/linux/mm.h>

```
enum pageflags {
        PG_locked,
        PG_error,
        PG_referenced,
        PG_uptodate,
        PG_dirty,
        PG_lru,
        ...
}
```
<include/linux/page-flags.h>

Page

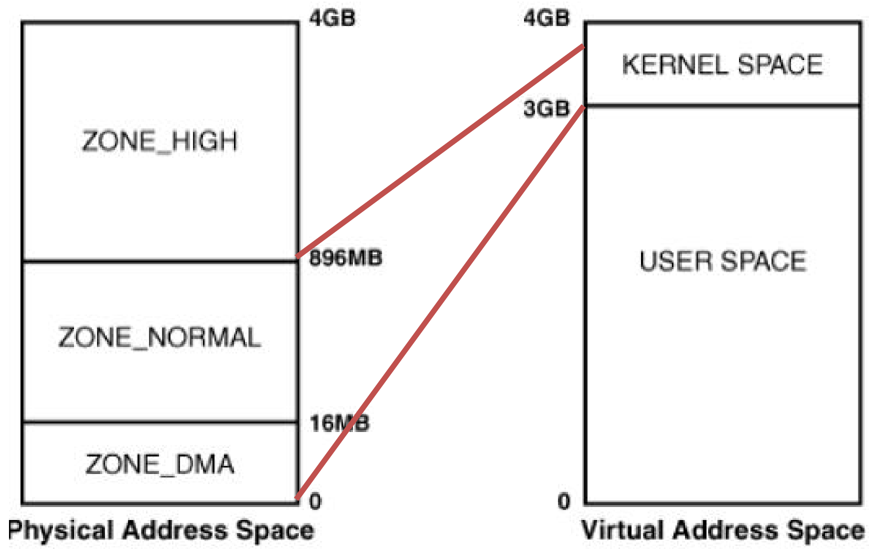
- Virtual field
  - Page’s virtual address
- Lru field
  - Managing page list
  - Use in page replacement
- Others…
  - mapping

Zones

- To group pages of similar properties
- Types
  - ZONE_DMA, ZONE_DMA32
    - Can be used for Direct Memory Access
  - ZONE_NORMAL
    - Regularly mapped pages
  - ZONE_HIGHMEM
    - Over 896MB for i386
  - ZONE_MOVABLE

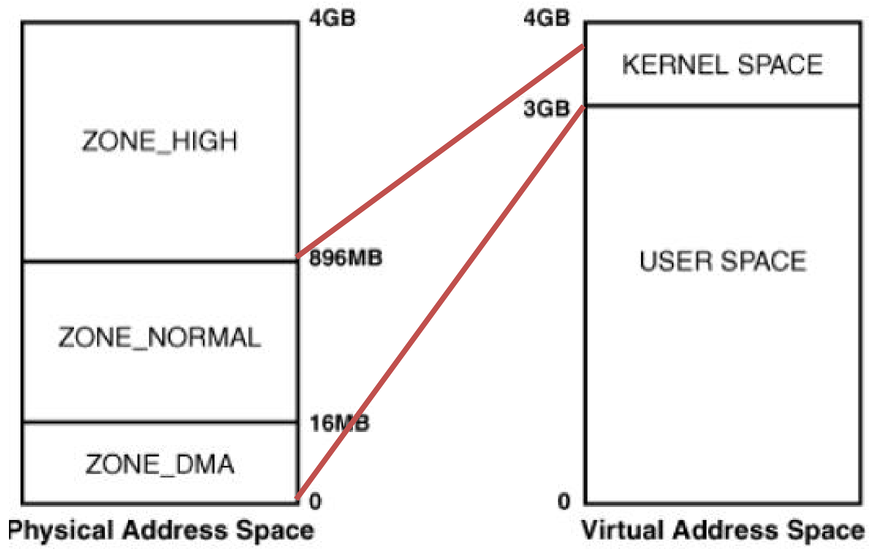
```
Enum zone_type
{
        ZONE_DMA,
        ZONE_DMA32,
        ZONE_NORMAL,
        ZONE_HIGHMEM,
        ZONE_MOVABLE,
        __MAX_NR_ZONES
}
                <include/linux/mmzone.h>
```

Basic structure - Zone(cont'd)



- ZONE_DMA
  - ISA can DMA only in the first 16MB area
    - For those devices
- ZONE_DMA_32
  - For devices only supports 32bit address space in x86_64
    - So it provides the first 4GB area.

Basic structure - Zone(cont'd)



- ZONE_NORMAL
  - Address which can be accessed directly by kernel
    - Except the reservation, 0 ~ 896MB area in i386
    - 0 ~ 64GB in x86_64
- ZONE_HIGHMEM
  - Address which cannot be accessed directly be kernel
    - Needs special operation to map in kernel space
    - 896MB ~ 4GB in i386
    - **No HIGHMEM for x86_64** since kernel can cover all memory directly

Getting Pages

- Allocating memory with page-size granularity

<linux/gfp.h>

  - alloc_pages(gfp_t gfp_mask, unsigned int order)
    - 2^order core allocation function
  - void *page_address(const struct page *page)
    - Get address of a page structure
  - __get_free_pages
    - Getting pages be alloc_pages() and return using page_address()
  - get_zeroed_page(gfp_t gfp_mask)
    - Getting an empty page
  - #define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
  - #define __get_free_page(gfp_mask)
    __get_free_pages(gfp_mask, 0)

Freeing Pages

- __free_pages(struct page *page, unsigned int order)
  - Passing page structure for argument
- free_pages(unsigned long addr, unsgiend int order)
  - Passing address for argument
- free_page(unsigned long addr)

kmalloc

- void *kmalloc(size_t size, gfp_t flags)
  - Similar operation to malloc() in user space
  - To allocate physically continuous memory
  - Can allocate upto 4MB continuous memory

```
#define MAX_ORDER 11
                              <include/linux/mmzone.h>
```

```
#define PAGE_SHIFT 12
                     <arch/x86/include/asm/page_types.h>
```

```
#define KMALLOC_SHIFT_HIGH ((MAX_ORDER + PAGE_SHIFT -1 <=
25?₩
              (MAX_ORDER + PAGE_SHIFT -1) : 25)
#define KMALLOC_MAX_SIZE (1UL << KMALLOC_SHIFT_HIGH)
                              <include/linux/mmzone.h>
```

kmalloc

- Flags
  - Represented by the gfp_t type (__get_free_pages)
  - Three categories
    - Action modifiers
      - **How** the kernel supposed to allocate
    - Zone modifiers
      - **Where** to allocate
    - Types
      - Combination of action & zone modifiers to stand a certain **type**

GFP Flags: Action Modifiers

Flag	Description
__GFP_WAIT	The allocator can sleep.
__GFP_HIGH	The allocator can access emergency pools.
__GFP_IO	The allocator can start disk I/O.
__GFP_FS	The allocator can start filesystem I/O.
__GFP_COLD	The allocator should use cache cold pages.
__GFP_NOWARN	The allocator does not print failure warnings.
__GFP_REPEAT	The allocator repeats the allocation if it fails, but the allocation can potentially fail.
__GFP_NOFAIL	The allocator indefinitely repeats the allocation. The allocation cannot fail.
__GFP_NORETRY	The allocator never retries if the allocation fails.
__GFP_NOMEMALLOC	The allocator does not fall back on reserves.
__GFP_HARDWALL	The allocator enforces “hardwall” cpuset boundaries.
__GFP_RECLAIMABLE	The allocator marks the pages reclaimable.
__GFP_COMP	The allocator adds compound page metadata (used internally by the hugetlb code)

GFP Flags: Zone Modifiers

- Flags can be combined
  - ptr = kmalloc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);

Flag	Description
__GFP_DMA	Allocates only from ZONE_DMA
__GFP_DMA32	Allocates only from ZONE_DMA32
__GFP_HIGHMEM	Allocates from ZONE_HIGHMEM or ZONE_NORMAL

GFP Flags: types

Flag	Description
GFP_ATOMIC	The allocation is high priority and must not sleep. This is the flag to use in interrupt handlers, in bottom halves, while holding a spin-lock, and in other situations where you cannot sleep.
GFP_NOWAIT	Like GFP_ATOMIC, except that the call will not fallback on emergency memory pools. This increases the liklihood of the memory allocation failing.
GFP_NOIO	This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when you cannot cause more disk I/O, which might lead to some unpleasant recursion.
GFP_NOFS	This allocation can block and can initiate disk I/O, if it must, but it will not initiate a filesystem operation. This is the flag to use in filesystem code when you cannot start another filesystem operation.
GFP_KERNEL	This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to do to obtain the memory requested by the caller. This flag should be your default choice.
GFP_USER	This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.
GFP_HIGHUSER	This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes.
GFP_DMA	This is an allocation from ZONE_DMA. Device drivers that need DMA-able memory use this flag, usually in combination with one of the preceding flags.

GFP Flags: types

- Composition of type flags

Flag	Value
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_WAIT
GFP_NOFS	(__GFP_WAIT | __GFP_IO)
GFP_KERNEL	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_USER	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_DMA	__GFP_DMA

GFP Flags

- Which flag to use when?

Situation	Solution
Process context, can sleep	Use GFP_KERNEL.
Process context, cannot sleep	Use GFP_ATOMIC, or perform your allocations with GFP_KERNEL at an earlier or later point when you can sleep.
Interrupt handler	Use GFP_ATOMIC.
Softirq	Use GFP_ATOMIC.
Tasklet	Use GFP_ATOMIC.
Need DMA-able memory, can sleep	Use (GFP_DMA | GFP_KERNEL).
Need DMA-able memory, cannot sleep	Use (GFP_DMA | GFP_ATOMIC), or perform your allocation at an earlier point when you can sleep.

kfree

- void kfree(const void *ptr)
  - DO NOT kfree memory **not previously allocated by kmalloc**
    - It may cause bugs

vmalloc/vfree

- void *vmalloc(unsigned long size)
  – Allocating more than 4MB
  – Logically continuous, but physically not
  – Large overhead
    - Changing kernel page table to allocate non-continuous area
    - Use kmalloc as possible
    - Kernel modules are loaded into memory via vmalloc
- void vfree(const void *addr)
- Declared in mm/vmalloc.c

Slab Layer

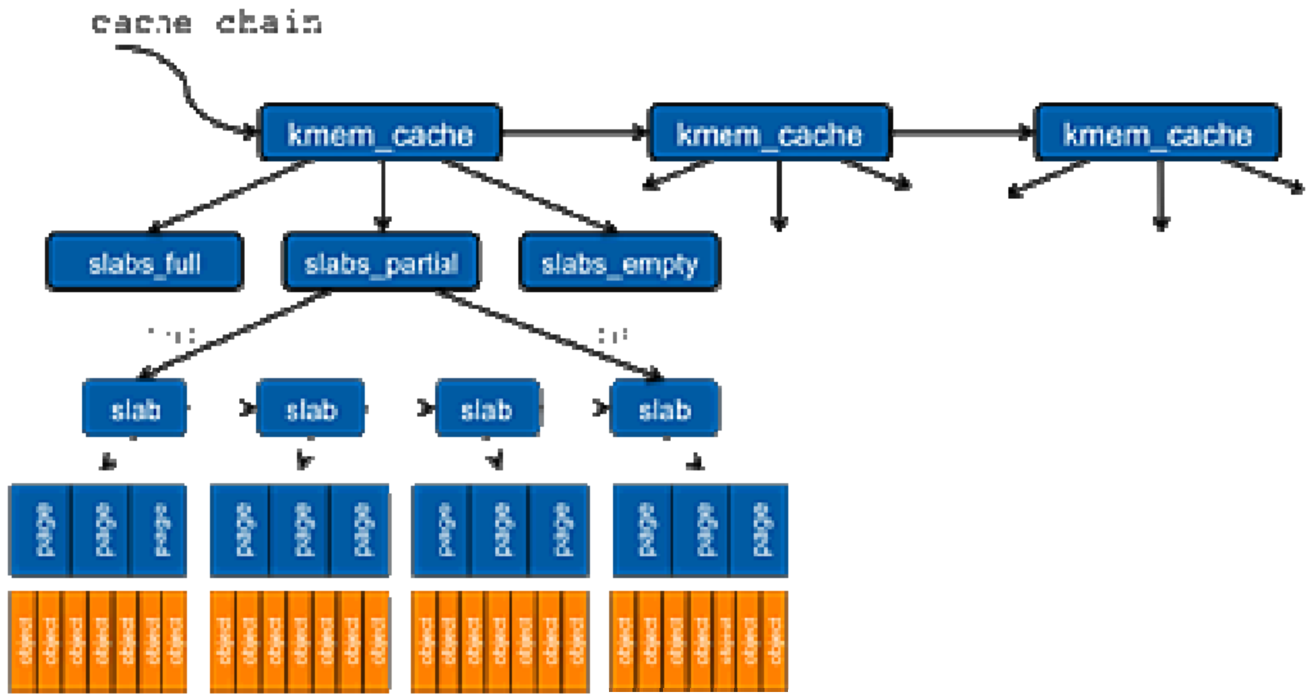
- Why slab layer is required?
  - A lot of data structures are frequently allocated/freed.
    - By naïve allocation, slowdown/fragmentation caused
  - To solve it, *free list* is maintained for each structure.
    - A block of available, already allocated data structures
  - There exists no central control by kernel for free lists.
    - E.g. shrink the list size if available memory size is low
- **<u>The slab layer is a generic data structure-caching layer</u>**
  - task_struct, inode, mm_struct, etc.

Slab Layer

- Design of the slab layer
  - *Cache*: a storage for a specific type of object
    - One cache per object type
    - semaphores, file objects, process descriptors, etc.
    - kmalloc() is built on the slab layer
  - *Slab*: a contiguous piece of memory, often several page size.
    - A cache is stored in 1 or more slabs.
    - Each slab contains some number of equal-sized *objects*.
      - No fragmentation
    - Three states
      - Full: all objects in the slab are in use.
      - Empty: all objects in the slab are free, so reclaimable by the kernel.
      - Partial: the slab contains both free and in-use objects.

Slab Layer

- Design of the slab layer (cont'd)
  - A linked list of caches



Slab Layer

- Slab operations
  - kmem_cache_create()
    - Creating a new cache
    - Typically used when the kernel initializes or a kernel module is loaded
  - kmem_cache_destroy()
    - Destroying a cache
  - void * kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
    - getting a free object pointer from cachep
    - If no free object, it obtains new pages via kmem_getpages().
  - void kmem_cache_free(struct kmem_cache *cachep, void *objp)
    - Freeing objp in cachep

Slab Layer

- An example of using the slab allocator

hahaman5@ubuntu: ~/linux-3.8.0

```
void __init fork_init(unsigned long mempages)
{
#ifndef CONFIG_ARCH_TASK_STRUCT_ALLOCATOR
#ifndef ARCH_MIN_TASKALIGN
#define ARCH_MIN_TASKALIGN      L1_CACHE_BYTES
#endif
        /* create a slab on which task_structs can be allocated */
        task_struct_cachep =
                kmem_cache_create("task_struct", sizeof(struct task_struct),
                        ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_NOTRACK, NULL);
#endif

        /* do the arch specific task caches init */
"kernel/fork.c" 1941 lines --13%--
```

hahaman5@ubuntu: ~/linux-3.8.0

```
#ifndef CONFIG_ARCH_TASK_STRUCT_ALLOCATOR
static struct kmem_cache *task_struct_cachep;

static inline struct task_struct *alloc_task_struct_node(int node)
{
        return kmem_cache_alloc_node(task_struct_cachep, GFP_KERNEL, node);
}

static inline void free_task_struct(struct task_struct *tsk)
{
        kmem_cache_free(task_struct_cachep, tsk);
"kernel/fork.c" 1941 lines --6%--
```

Slab Layer

- Checking slab

```
hahaman5@ubuntu: ~/linux-3.8.0
hahaman5@ubuntu:~/linux-3.8.0$ sudo cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcoun
t> <sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
ext2_inode_cache      21     21    752   21    4 : tunables    0    0    0 : slabdata      1      1      0
UDPLITEv6              0      0   1088   15    4 : tunables    0    0    0 : slabdata      0      0      0
UDPv6                 15     15   1088   15    4 : tunables    0    0    0 : slabdata      1      1      0
tw_sock_TCPv6          0      0    256   16    1 : tunables    0    0    0 : slabdata      0      0      0
TCPv6                  8      8   1984    8    4 : tunables    0    0    0 : slabdata      1      1      0
zcache_objnode         0      0    536   15    2 : tunables    0    0    0 : slabdata      0      0      0
kcopyd_job             0      0   3240   10    8 : tunables    0    0    0 : slabdata      0      0      0
dm_uevent              0      0   2608   12    8 : tunables    0    0    0 : slabdata      0      0      0
dm_rq_clone_bio_info       0      0    144   28    1 : tunables    0    0    0 : slabdata      0      0      0
dm_rq_target_io        0      0    416   19    2 : tunables    0    0    0 : slabdata      0      0      0
bsg_cmd                0      0    312   13    1 : tunables    0    0    0 : slabdata      0      0      0
mqueue_inode_cache      9      9    896    9    2 : tunables    0    0    0 : slabdata      1      1      0
fuse_request           0      0    608   13    2 : tunables    0    0    0 : slabdata      0      0      0
fuse_inode             0      0    704   11    2 : tunables    0    0    0 : slabdata      0      0      0
ecryptfs_key_record_cache      0      0    576   14    2 : tunables    0    0    0 : slabdata      0      0
   0
ecryptfs_inode_cache       0      0    960    8    2 : tunables    0    0    0 : slabdata      0      0      0
fat_inode_cache        0      0    688   23    4 : tunables    0    0    0 : slabdata      0      0      0
fat_cache              0      0     40  102    1 : tunables    0    0    0 : slabdata      0      0      0
hugetlbfs_inode_cache     14     14    576   14    2 : tunables    0    0    0 : slabdata      1      1      0
journal_handle       170    170     24  170    1 : tunables    0    0    0 : slabdata      1      1      0
journal_head         936    936    112   36    1 : tunables    0    0    0 : slabdata     26     26      0
ext4_inode_cache   31994  31994    912   17    4 : tunables    0    0    0 : slabdata   1882   1882      0
ext4_free_data        64     64     64   64    1 : tunables    0    0    0 : slabdata      1      1      0
ext4_allocation_context     60     60    136   30    1 : tunables    0    0    0 : slabdata      2      2
 0
ext4_io_end           70     70   1128   14    4 : tunables    0    0    0 : slabdata      5      5      0
ext4_io_page         770   2304     16  256    1 : tunables    0    0    0 : slabdata      9      9      0
extent_status        128    128     32  128    1 : tunables    0    0    0 : slabdata      1      1      0
ext3_inode_cache       0      0    776   10    2 : tunables    0    0    0 : slabdata      0      0      0
ext3_xattr             0      0     88   46    1 : tunables    0    0    0 : slabdata      0      0      0
```

Kernel stack

- Every active thread has a kernel stack
  - Statically allocated 2 contiguous pages
    - Which stores task_struct (SEE Lecture 1)
    - Kernel thread uses only the kernel stack.
  - Used when syscall or interrupt
    - Interrupt handler uses the interrupted process.
  - 4k kernel stack option is available
    - To reduce kernel memory space
    - Interrupt stack per CPU is provided for interrupt handlers
      - Some legacy handlers overflow 4k stack.

High memory mapping

- Kernel can directly access to 1G space
  - Accessing to other part needs mapping
    - Permanent mapping
      - kmap/unmap
    - Temporary mapping
      - kmap(kunmap)_atomic
      - Must not sleep between map and unmap
- Use 896MB ~ 1GB space to mapping

Percpu allocation

- Maintaining a counter per CPU
  - No need to use global lock
  - Reducing cache invalidation
- Pros
  - Reduced locking requirement
  - Reduced cache invalidation
- Cons
  - Disabling kernel preemption
  - Can’t sleep in using percpu data

Percpu allocation

```
#define alloc_percpu(type)₩
        (typeof(type) __percpu *)__alloc_percpu(sizeof(type), __alignof__(type))
                                                    <include/linux/percpu.h>
```

```
#define get_cpu_var(var) (*({
        ₩
        preempt_disable();          ₩
        &__get_cpu_var(var); }))

#define put_cpu_var(var) do {       ₩
        (void)&(var);               ₩
        preempt_enable();           ₩
} while (0)
                <include/linux/percpu.h>
```

```
Void *percpu_ptr;
Uinsigned long *foo;

Percpu_ptr = alloc_percpu(unsigned long);
If(!percpu_ptr)
        /* error handling code */

Foo = get_cpu_var(percpu_ptr);
/* manipulate foo .. */

Put_cpu_var(percpu_ptr);
                        <Example code>
```