Memory Allocation: Day 2

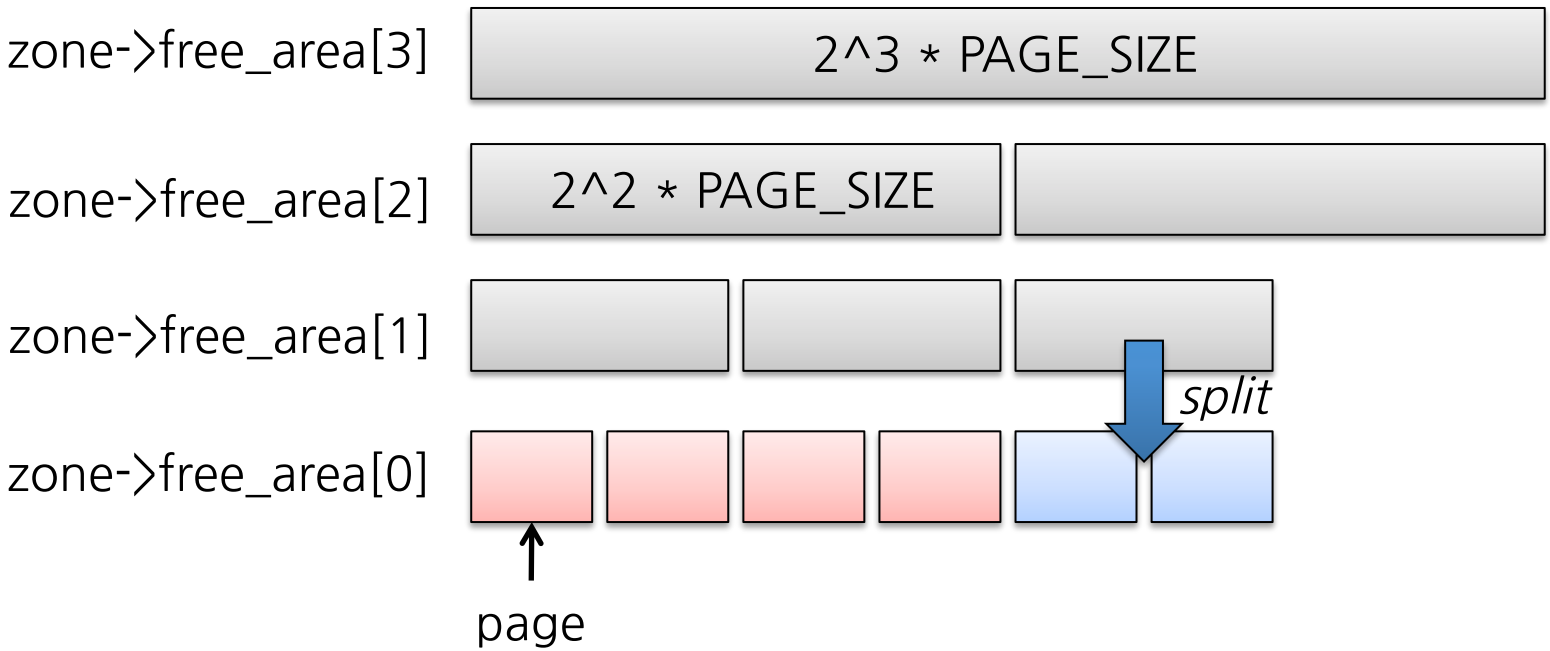
SWE3015

Sung-hun Kim

This slide is based on Jaeho Hwang's lecture slide

Buddy Allocator

- Linux memory allocator
  - Treat memory as a collection of pages aligned on squares of two pages boundaries
    - From 2^0 to 2^10 (MAX_ORDER == 11)
    - If low-order pages exhausted, higher-order page will be splitted



Buddy Allocator

- Buddy allocator functions
  - struct page *__rmqueue(struct zone *zone,
    unsigned int order);
    - Get free pages from buddy allocator
  - void __free_one_page(struct page *page, struct zone *zone,
    unsigned int order);
    - Free and put pages to buddy allocator's free page list
  - void page_is_buddy(struct page *page, unsigned int order);
    - Check whether a page is buddy and free or not

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

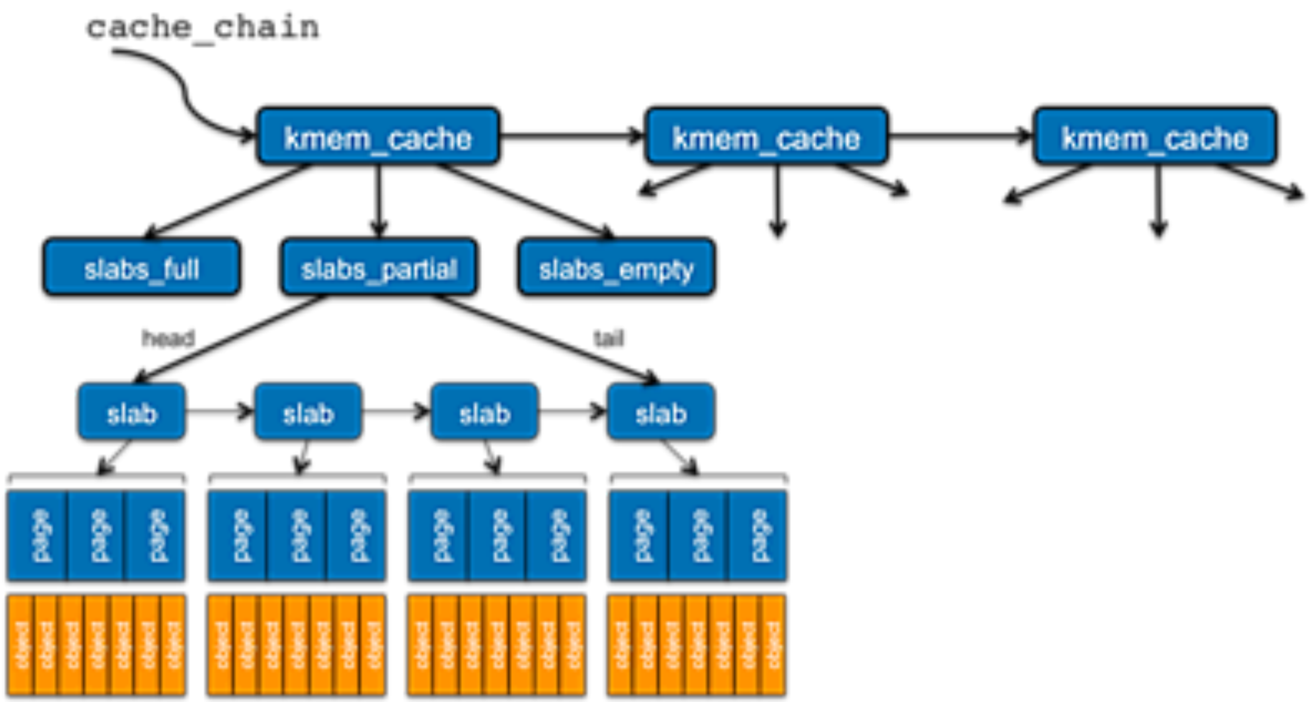
- **The slab layer is a generic data structure-caching layer**
  - task_struct, inode, mm_struct, etc.
  - Slab layer works on buddy allocator

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> <batchcount> <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
```

성균관대학교
SUNGKYUNKWAN UNIVERSITY

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;
unsigned 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>
```