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

```
zone->free_area[3]  2^3 * PAGE_SIZE
zone->free_area[2]  2^2 * PAGE_SIZE
zone->free_area[1]  split
zone->free_area[0]  page
```
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
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.
• Design of the slab layer (cont’d)
  – A linked list of caches
• 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
• An example of using the slab allocator

```c
#include <linux/init.h>

__init void __init_fork_init(unsigned long mempages)
{
    if (defined CONFIG_ARCH_TASK_STRUCT_ALLOCATOR)
        if (defined ARCH_MIN_TASKALIGN)
            define ARCH_MIN_TASKALIGN L1_CACHE_BYTES

    /* create a slab on which task_structs can be allocated */
    task_struct_cache =
        kmem_cache_create("task_struct", sizeof(struct task_struct),
                         ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_NOTRACK, NULL);

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

#include <linux/init.h>

#define 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 *t)
{
    kmem_cache_free(task_struct_cachep, t);
    "kernel/fork.c" 1941 lines --6--
}
```
Slab Layer

- Checking slab
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
#define alloc_percpu(type)\n   (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>

---

Example code

```
SWE3015:
  Operating System
  Project
```