Physical Memory Management in Linux

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University
1. Physical Memory Organization
2. Memory Allocation with Buddy System
3. Memory Allocation with Slab System
4. Q & A
Memory Hierarchy

- **L0**: CPU registers hold words retrieved from L1 cache
- **L1 cache (SRAM)**: L1 cache holds cache lines retrieved from L2 cache
- **L2 cache (SRAM)**: L2 cache holds cache lines retrieved from main memory
- **Main memory (DRAM)**: Main memory holds disk blocks retrieved from local disks
- **Local secondary storage (local disks)**: Local disks hold files retrieved from disks on remote network servers
- **Remote secondary storage (tapes, distributed file systems, Web servers)**

- Smaller, faster, costlier per byte
- Larger, slower, cheaper per byte
Memory Organization

- Two ways of memory organization
  - Uniform Memory Access (UMA)
  - Non-Uniform Memory Access (NUMA)
Zones

Because of hardware limitations, the kernel divides physical memory into different zones

- Some hardware devices can perform DMA (direct memory access) to only certain memory addresses
- Some architectures cannot physically address larger amounts of memory than they can virtually address

<table>
<thead>
<tr>
<th>Zone</th>
<th>Description</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZONE_DMA</td>
<td>DMA-able pages</td>
<td>0-16M</td>
</tr>
<tr>
<td>ZONE_NORMAL</td>
<td>Normally addressable</td>
<td>16-896M</td>
</tr>
<tr>
<td>ZONE_HIGHMEM</td>
<td>Dynamically mapped pages (not used for other architectures)</td>
<td>&gt;896M</td>
</tr>
</tbody>
</table>
Zone Descriptor

- Each zone is a group of pages
- Described by struct zone in `<linux/mmzone.h>`

```c
#include/linux/mmzone.h
struct zone {
    unsigned long watermark[NR_WMARK];
    unsigned long lowmem_reserve[MAX_NR_ZONES];

    int node;
    unsigned long min_unmapped_pages;
    unsigned long min_slab_pages;

    struct per_cpu_pageset __percpu *pageset;
    spinlock_t lock;
    seqlock_t span_seqlock;
    struct free_area free_area[MAX_ORDER];
    ...
    atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS];
    struct page *zone_mem_map;
    struct pglist_data *zone_pgdat;
    unsigned long zone_start_pfn;
    unsigned long spanned_pages;
    unsigned long present_pages;
    const char *name;
} cacheline_internodealigned_in_smp;
```

- Watermarks used when pages are swapped out
- Array to implement per-CPU hot-cold page lists
- Array of data structures used to implement the buddy system
- Statistical information about the zone (NR_ACTIVE, NR_INACTIVE...)
- The index of the first page frame of the zone
Node Descriptor

- Described by pg_data_t in `<linux/mmzone.h>`

```c
#include/linux/mmzone.h
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;

    struct page *node_mem_map;
    struct page_cgroup *node_page_cgroup;
    struct bootmem_data *bdata;

    spinlock_t node_size_lock;
    unsigned long node_start_pfn;
    unsigned long node_present_pages;
    unsigned long node_spanned_pages;

    int node_id;
    wait_queue_head_t kswapd_wait;
    struct task_struct *kswapd;
    int kswapd_max_order;
    enum zone_type classzone_idx;
} pg_data_t;
```
Page Descriptor

- The kernel treats physical pages as the basic unit of memory management
- Every physical page is managed with a struct page structure in `<linux/mm_types.h>`

```c
#include <linux/mm_types.h>
struct page {
    unsigned long flags;
    atomic_t _count;
    union {
        atomic_t _mapcount;
        struct {
            u16 inuse;
            u16 objects;
        }
    }
    union {
        struct {
            struct address_space *mapping;
        }
        void *virtual;
    }
    ...}
```
Node, Zone, and Page

Node Descriptors

Zone Descriptors

Page Descriptors
Memory Allocation with Buddy System
Memory Allocation Structure

- User Code
- Linux Memory Allocation Code
- Slab Allocator
- Buddy System
- NUMA
- Zone
- Physical Memory
- Page

allocate caches
allocate pages
Buddy System

Used for low level page allocation
- Memory is partitioned into “buddy blocks”
- Allocates a group of contiguous page frames
- Allocates memory from one of a set of disjoint zones
- Efficiently reduces external fragmentation

Invented in 1963, Harry Markowitz

Harry Markowitz, American economist who won Nobel Prize in Economic Sciences
Buddy Algorithm

Allocation
- Allocate “2<sup>n</sup> pages”
- When a X bytes of memory request comes, allocate 2<sup>n</sup> contiguous physical pages that is larger than X
- If the block of pages found is larger than requested, it is “broken down” until it can find a block of the requested size

Deallocation
- “Recombine pages” into large blocks of free pages whenever it can
- Whenever a block of pages is freed, the adjacent or buddy block of the same size is checked to see if it is free
struct free_area
{
    struct list_head free_list;
    unsigned long *map;
};
Buddy Algorithm

order  free_area_t
zone->free_area

0
1
2
3
4
5
6
7
8
9
MAX_ORDER

Requesting Process

2 x 2² block
2 x 2³ block
1 x 2⁴ block
Allocating Pages

- **alloc_pages()** in `<linux/gfp.h>`
  - Allocates $2^\text{order}$ contiguous physical pages and returns a pointer to the first page’s page structure
  - The logical address can be obtained with the function `page_address()`
    - This returns a pointer to the logical address where the given physical page currently resides

- **__get_free_pages()**
  - Works the same as `alloc_pages()`
  - The difference is that it directly returns the logical address of the first requested page

- **alloc_page()** and **__get_free_page()**
  - Used for getting only one page

\(\text{(gfp: get free pages)}\)
Other Allocation Functions

- **get_zeroed_page()**
  - Returns a page filled with zeroes

- **kmalloc()**
  - Similar to the user-space malloc()
  - Allocates memory in byte-sized chunks
  - For most kernel allocations, kmalloc() is the preferred choice
Memory Allocation Hierarchy

APIs for getting pages

```c
#include <mm/page_alloc.c>
struct page *
__alloc_pages_nodemask(...) {
    ...
    page = get_page_from_freelist(
        gfp_mask | GFP_HARDWALL,
        nodemask, order, zonelist, high_zoneidx,
        ALLOC_WMARK_LOW | ALLOC_CPUSSET,
        preferred_zone, migratetype);
    ...
}
```
Freeing Pages

- **Five functions**
  - void __free_pages(struct page *page, unsigned int order)
  - void free_pages(unsigned long addr, unsigned int order)
  - void free_page(unsigned long addr)
  - void kfree(const void *ptr)
  - void vfree(const void *addr)

- **Notes**
  - We must be careful to free only pages we allocated
  - Otherwise, serious corruption may happen or the kernel may hang itself (the kernel trusts itself)

```c
page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
    /* insufficient memory: you must handle this error! */
    return -ENOMEM;
}
free_pages(page, 3);
```
Memory Reclamation Hierarchy

```c
static inline void __free_one_page
    (struct page *page, struct zone *zone,
     unsigned int order, int migratetype) {
    ...
    zone->free_area[order].nr_free--;  
    combined_idx = buddy_idx & page_idx;  
    page = page + (combined_idx - page_idx);  
    page_idx = combined_idx;  
    order++;
    }  
    If possible, merge buddies into larger one
    ...
    list_add(&page->lru, &zone->
    free_area[order].free_list[migratetype]);  
    zone->free_area[order].nr_free++;  
    }  
    Put a free buddy into the free_area list
```
# Buddy System APIs

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct page *</td>
<td>alloc_pages</td>
<td>gfp_t gfp_mask</td>
<td>Allocate 2\textsuperscript{order} pages and return a pointer to the first page structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned int order</td>
<td></td>
</tr>
<tr>
<td>struct page *</td>
<td>alloc_page</td>
<td>gfp_t gfp_mask</td>
<td>Allocate a page and return pointer to its page structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td>__get_free_pages</td>
<td>gfp_t gfp_mask</td>
<td>Allocate 2\textsuperscript{order} pages and return first page’s virtual address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned int order</td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td>__get_free_page</td>
<td>gfp_t gfp_mask</td>
<td>Allocate a free page and return its virtual address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>struct page *</td>
<td>get_dma_pages</td>
<td>gfp_t gfp_mask</td>
<td>Allocate 2\textsuperscript{order} pages from zone DMA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned int order</td>
<td></td>
</tr>
<tr>
<td>struct page *</td>
<td>get_zeroed_page</td>
<td>gfp_t gfp_mask</td>
<td>Allocate a page filled with zeros</td>
</tr>
<tr>
<td>void</td>
<td>free_pages</td>
<td>void * addr</td>
<td>Free 2\textsuperscript{order} allocated pages starting at addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned int order</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>free_page</td>
<td>void * addr</td>
<td>Free an allocated page</td>
</tr>
<tr>
<td>void</td>
<td>__free_pages</td>
<td>struct page *</td>
<td>Free 2\textsuperscript{order} allocated pages starting from given page structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned int order</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>__free_page</td>
<td>struct page *</td>
<td>Free an allocated page</td>
</tr>
</tbody>
</table>
MemoryAllocation with Slab System
Slab Layer

- Slab layer is used to facilitate frequent allocations and deallocations of kernel data structures
  - It maintains a free list for each type of data structure, which contains a block of available, already allocated, data structures

- Free lists serve as “caches for kernel objects”
  - When code requires a new instance of a data structure, it grab one of the structures off the free list
  - Later, when the data structure is no longer needed, it is returned to the free list
Object, Slab, and Cache

- **Object**: an instance of kernel data structure
  - e.g.) task_struct, struct_inode, ...
- **Slab**: group of objects of the same type
- **Cache**: group of slabs
**task_struct Allocation**

- **Generic Kernel Code**
  - Requesting memory allocation for data structure `task_struct A` (3264B)

- **Slab Allocator**

- **Cache for `task_struct`**
  - 3264KB
    - `task_struct`
    - `task_struct`
    - `task_struct`
    - ...
    - `task_struct`
Cache Descriptor

```
/mm/slab.c
struct kmem_cache_s
{
    struct array_cache *array[NR_CPUS];
    struct kmem_list3 lists;
    unsigned int objsize;
    unsigned int flags;
    unsigned int gfporder;
    unsigned int gfpflags;
    void (*ctor)(void *, kmem_cache_t *, unsigned long);
    void (*dtor)(void *, kmem_cache_t *, unsigned long);
    const char *name;
    struct list_head next;
};
```

- Per-CPU slab cache
- Parameters passed to the Buddy system
- Constructor
- De-constructor
Slab Descriptor

/mm/slab.c

struct slab {
    struct list_head list;
    unsigned long colouroff;
    void * s_mem;
    unsigned int inuse;
    kmem_bufctl_t free;
};

Diagram:
- Pointer of first object in the slab
- # of allocated objects
- Index of first free object

Free object index
Relationship
Slab States and Object Allocation

- Each slab is in one of three states
  - Full: slab has no free objects
  - Partial: slab has some allocated objects and some free objects
  - Empty: slab has no allocated objects

- When kernel code requests a new object,
  - The request is satisfied from a partial slab, if one exists
  - Otherwise, the request is satisfied from an empty slab
  - If there exists no empty, one is created
Slab Algorithm

[Diagram showing the Slab Algorithm with states: completely full slabs, partly empty slabs, completely empty slabs, kmem_cache_t, slabs, firstnotfull, and connections between them].
Creating a Cache

- A new cache is created via `kmem_cach_create()`
  - Returns a pointer to the created cache
  - Otherwise, it returns NULL
  - This function must not be called from interrupt context because it can sleep

```c
<mm/slab.c>
struct kmem_cache * kmem_cache_create( const char *name,
size_t size,
size_t align,
unsigned long flags,
void (*ctor)(void*)
);
```

- A string which is used in `/proc/slabinfo` to identify this cache
- The size of objects to be created in this cache
- The required alignment for the objects (offset of the first object within slab)
- SLAB flags
- A constructor for the objects
Destroying a Cache

- `kmem_cache_destroy()` destroys the given cache
  - It is generally invoked from module shutdown code
  - It must not be called from interrupt context because it may sleep

- The caller of this function must ensure two conditions are true prior to invoking this function:
  - All slabs in the cache are empty
  - No one accesses the cache during (and obviously after) a call to `kmem_cache_destroy()`
Destroying a Cache

```c
#include "mm/slab.c"

void kmem_cache_destroy(struct kmem_cache *cachep) { 
  ...
  mutex_lock(&cache_chain_mutex); 
  /* the chain is never empty, cache_cache is never destroyed */
  list_del(&cachep->next);
  if (__cache_shrink(cachep)) { 
    slab_error(cachep, "Can't free all objects");
    list_add(&cachep->next, &cache_chain);
    mutex_unlock(&cache_chain_mutex);
    put_online_cpus();
    return;
  }
  ...
  __kmem_cache_destroy(cachep);
  mutex_unlock(&cache_chain_mutex);
  ...
}
```

No one accesses the cache during a call to `kmem_cache_destroy`

All slabs in the cache should be empty
Creating a Slab

- The slab allocator creates new slabs by interfacing with the low-level kernel page allocator via `__get_free_pages()`.

```c
#include <mm/slab.c>
static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags, int nodeid)
{
    struct page *page;
    // Specific cache that needs more pages

    ... flags |= cachep->gfpflags;
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        flags |= GFP_RECLAIMABLE;

    page = alloc_pages_exact_node(nodeid, flags | GFP_NOTRACK, cachep->gfporder);
    if (!page)
        return NULL;

    ... }
```
Allocating and Freeing an Object

- **kmem_cache_alloc**
  - Object is obtained from the cache

  ```c
  <mm/slab.c>
  void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
  ```

  - If no free objects are in any slabs, the slab must obtain new pages via `kmem_getpages()`

- **kmem_cache_free**
  - Free an object and return it to its originating slab

  ```c
  <mm/slab.c>
  void kmem_cache_free(struct kmem_cache *cachep, void *objp)
  ```

  - Marks the object objp in cachep as free
Alternatives to Slab

- Slab works well for many possible workloads
  - However, fails to provide optimal performance

- To cope with such situations
  - Two alternative slab allocators are designed during development of kernel 2.6
  - Slub (from 2.6.16) and slob (from 2.6.22) allocators
SLUB (for large systems)

- Became the default allocator in 2.6.23
  - SLUB has no metadata (slab descriptor) at the beginning of each slab like SLAB, but instead it has added its metadata variables in the page structure to track the allocator’s data on the physical pages
  - A page’s freelist pointer is used to point to the first free object of the slab
    - Object allocation from this per-CPU slab (fast path)
    - If full, fallback to per-node partial lists (slow path)
SLUB (for large systems)

- freelist
- tid
- cpu slab
- partial slabs

free objects from slab 1

- slab 1
- slab 2
- slab 3

kmem_cache_cpu

- free object
- slab
SLOB (for small systems)

- Simple Linked-lists Of Blocks (Slob)
  - The design is closer to traditional userland memory allocators rather than the slab allocators SLAB and SLUB
  - SLOB places all objects/structures on pages arranged in three linked lists, for small, medium and large allocations
  - Small are the allocations of size less than SLOB_BREAK1 (256 bytes), medium those less than SLOB_BREAK2 (1024 bytes), and large are all the other allocations
  - Uses a simple first fit algorithm

```
struct slob_page

slob
```

```
size

offset
```

free unit

used unit