Page Allocation

SWE3015

Jeaho Hwang
• Unit of memory mapping
  – 4KB size by default
  – For every page there is `struct page`.
  – 36B / 4KB = 0.88% overhead

```c
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
```
• 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()

```c
enum pageflags {
  PG_locked,
  PG_error,
  PG_referenced,
  PG_uptodate,
  PG_dirty,
  PG_lru,
  ...
};
```

```c
static inline int page_count(struct page *page)
{
  return atomic_read(&compound_head(page)->_count);
}
```

```
#include/linux/mm.h
```

```
#include/linux/page-flags.h
```
- 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

```c
Enum zone_type {
    ZONE_DMA,
    ZONE_DMA32,
    ZONE_NORMAL,
    ZONE_HIGHMEM,
    ZONE_MOVABLE,
    __MAX_NR_ZONES
}
```

```c
#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 by 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
  
  - 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, unsigned int order)`
  - Passing address for argument
- `free_page(unsigned long addr)`
• `void *kmalloc(size_t size, gfp_t flags)`
  – Similar operation to `malloc()` in user space
  – To allocate physically continuous memory
  – Can allocate up to 4MB continuous memory

```c
#define MAX_ORDER 11
#include/linux/mmzone.h
#define PAGE_SHIFT 12
#include/`arch/x86/include/asm/page_types.h`
#include/linux/mmzone.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)
```
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

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

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__GFP_DMA</td>
<td>Allocates only from ZONE_DMA</td>
</tr>
<tr>
<td>__GFP_DMA32</td>
<td>Allocates only from ZONE_DMA32</td>
</tr>
<tr>
<td>__GFP_HIGHMEM</td>
<td>Allocates from ZONE_HIGHMEM or ZONE_NORMAL</td>
</tr>
</tbody>
</table>
## GFP Flags: types

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFP_ATOMIC</td>
<td>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.</td>
</tr>
<tr>
<td>GFP_NOWAIT</td>
<td>Like GFP_ATOMIC, except that the call will not fallback on emergency memory pools. This increases the likelihood of the memory allocation failing.</td>
</tr>
<tr>
<td>GFP_NOIO</td>
<td>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.</td>
</tr>
<tr>
<td>GFP_NOFS</td>
<td>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.</td>
</tr>
<tr>
<td>GFP_KERNEL</td>
<td>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.</td>
</tr>
<tr>
<td>GFP_USER</td>
<td>This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.</td>
</tr>
<tr>
<td>GFP_HIGHUSER</td>
<td>This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes.</td>
</tr>
<tr>
<td>GFP_DMA</td>
<td>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.</td>
</tr>
</tbody>
</table>
GFP Flags: types

- Composition of type flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFP_ATOMIC</td>
<td>__GFP_HIGH</td>
</tr>
<tr>
<td>GFP_NOIO</td>
<td>__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_NOFS</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_KERNEL</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_USER</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_DMA</td>
<td>__GFP_DMA</td>
</tr>
</tbody>
</table>
### GFP Flags

- **Which flag to use when?**

<table>
<thead>
<tr>
<th>Situation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process context, can sleep</td>
<td>Use GFP_KERNEL.</td>
</tr>
<tr>
<td>Process context, cannot sleep</td>
<td>Use GFP_ATOMIC, or perform your allocations with GFP_KERNEL at an earlier or later point when you can sleep.</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Use GFP_ATOMIC.</td>
</tr>
<tr>
<td>Softirq</td>
<td>Use GFP_ATOMIC.</td>
</tr>
<tr>
<td>Tasklet</td>
<td>Use GFP_ATOMIC.</td>
</tr>
<tr>
<td>Need DMA-able memory, can sleep</td>
<td>Use (GFP_DMA</td>
</tr>
<tr>
<td>Need DMA-able memory, cannot sleep</td>
<td>Use (GFP_DMA</td>
</tr>
</tbody>
</table>
• void kfree(const void *ptr)
  – DO NOT kfree memory **not previously allocated by kmalloc**
  • It may cause bugs
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
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.
• 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 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`
An example of using the slab allocator

```c
void __init fork_init(unsigned long mempages)
{
    #ifdef CONFIG_ARCH_TASK_STRUCT_ALLOCATOR
    #ifdef 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%--
}
```

```c
#ifdef 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%--
• 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))\n  <include/linux/percpu.h>

#define get_cpu_var(var) (*({\n  preempt_disable(); \# \&__get_cpu_var(var); }))

#define put_cpu_var(var) do { \#(void)&(var); \#preempt_enable(); \#} while (0)

Void *percpu_ptr;\nUnsigned long *foo;

percpu_ptr = alloc_percpu(unsigned long);\nIf(!percpu_ptr)\n  /* error handling code */

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

Put_cpu_var(percpu_ptr); <Example code>