Memory Allocation :Day 1

SWE3015

Sung-hun Kim

This slide is based on Jaeho Hwang's lecture slide

Review

- Virtual memory
  - All processes have their own isolated virtual memory
    - Processes can view virtual address space only
  - All memory space are sliced into pages
  - Page table stores virtual-to-physical address mapping

- Demand paging
  - OSes provide lazy page allocation/load for effective memory management
  - Page fault handler handles several *faults*
    - Major fault
    - Minor fault

Page

- Unit of memory mapping
  - 4KB size by default (can be extended 2 MiB, 1 GiB according t o hardware specification)
  - For every page there is ***struct page.***
    - 36B / 4KiB = 0.88% overhead
    - For 4 GiB memory system -> 20 MiB of page structures

```
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)->_c
ount);
}
```

<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(least recently used) field
  - Managing page list
  - Use in page replacement
- Others…
  - mapping
    - Used for managing address space of device file
    - Or mapped to anon_vma

Kernel Functions for Page Structure

- get_page(struct page *page)
  - Increment page's _count
- put_page(struct page *page)
  - Decrement page's _count
- lock_page(struct page *page)
  - Lock page by using page->flag and wait_queue
- unlock_page(struct page *page)
  - Unlock page by using page->flag and wait_queue
  - Wake up waiting processes

Zones

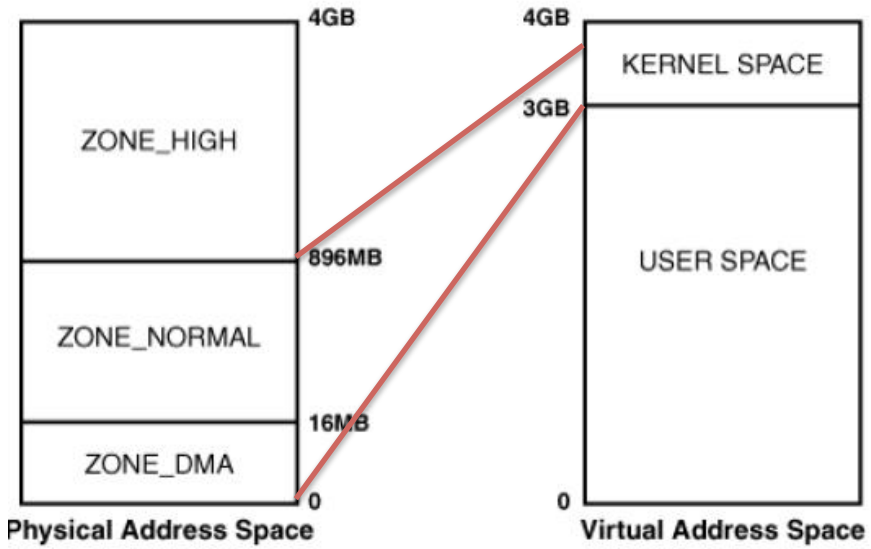
- To group pages of similar properties
- Types
  - ZONE_DMA, ZONE_DMA32
    - Can be used for Direct Memory Access
  - ZONE_NORMAL
    - Regularly mapped pages
  - ZONE_HIGHMEM
    - Over 896MB for i386
  - ZONE_MOVABLE

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

<include/linux/mmzone.h>

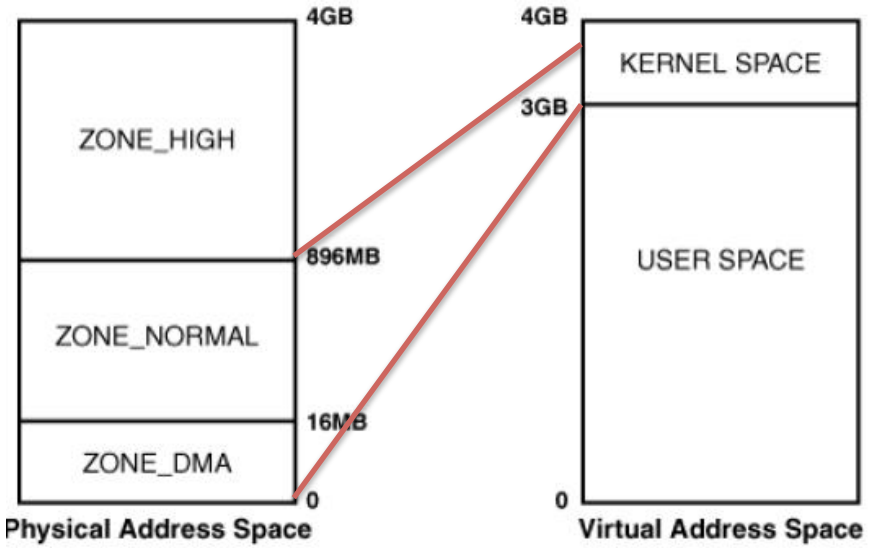
Basic structure - Zone(cont'd)

- 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)

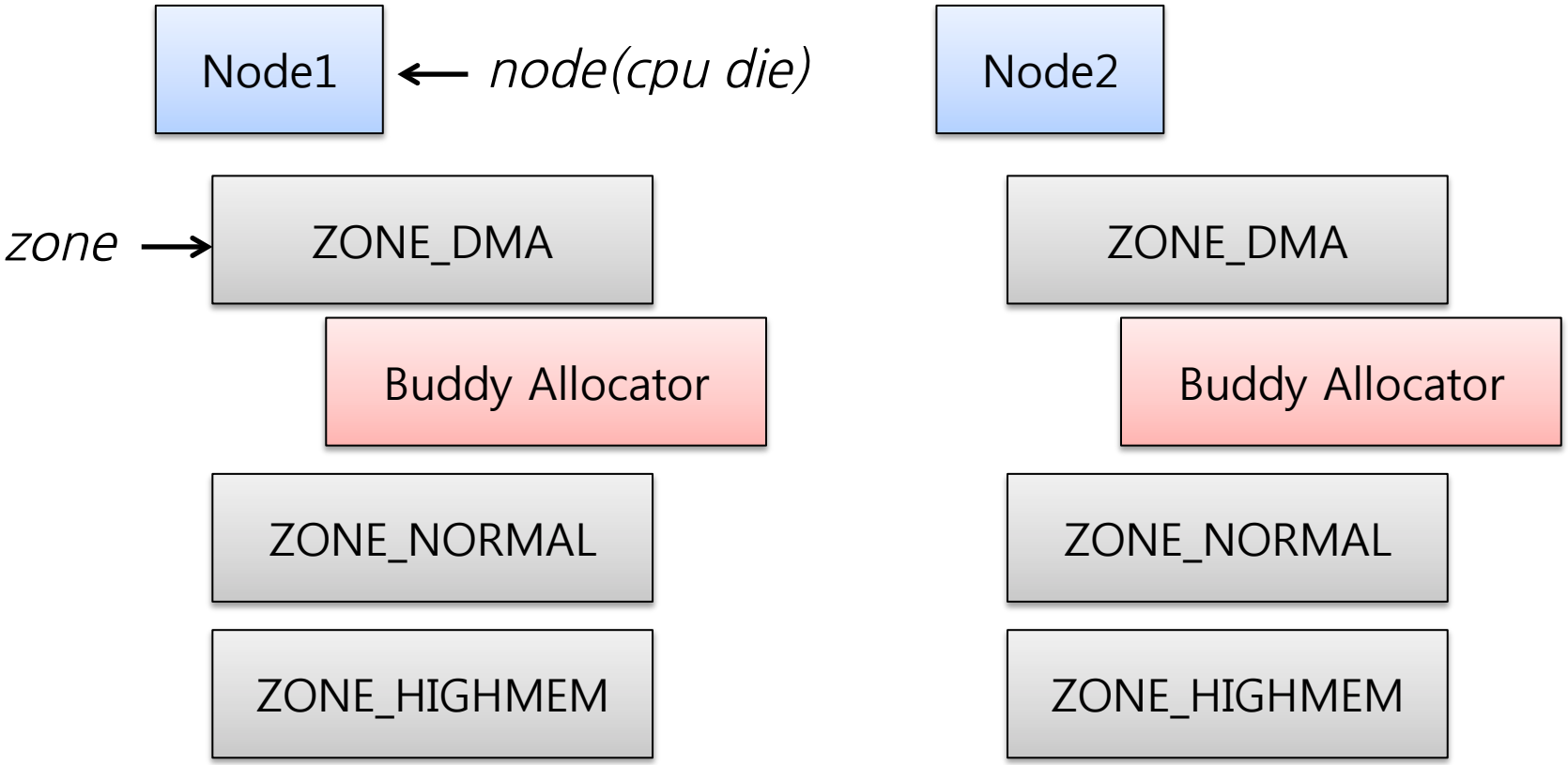
Freeing Pages

- __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)

Page Allocation Steps

alloc_pages(gfp, order)

__alloc_pages_nodemask(gfp, order, zone, node)



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 <= 2
5?₩
                (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