CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 16

Memory Management

March 27, 2025

Huaicheng Li

https://people.cs.vt.edu/huaicheng/lkp-sp25/

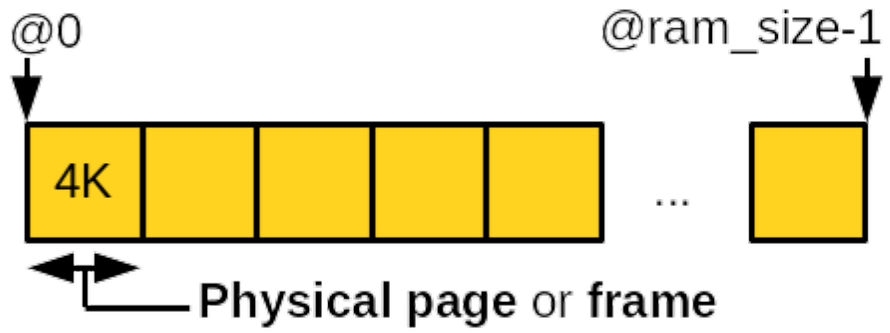
Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

Previously: Kernel Synchronization

- Pages and zones
- Page allocation
- kmalloc, vmalloc (recap)
- Slab allocator
- Stack, high memory, per-CPU data structures

Pages

- Memory is divided into physical pages or frames
- The page is the basic management unit in the kernel
- Page size is machine-dependent
  - Determined by the memory management unit (MMU)
  - 4KB in general, some are 2MB and 1GB: getconf PAGESIZE



Pages

- Each physical page is represented by "struct page"
  - Recently, converted to "folio" as a group of pages, but basic concept is the same
- Defined in "include/linux/mm_types.h"

```
struct page {
    unsigned long flags;     /* page status (permission, dirty, etc.) */
    unsigned counters;       /* usage count */
    struct address_space *mapping;
                             /* address space mapping */
    pgoff_t index;           /* offset within the mapping */
    struct list_head lru;    /* LRU list buffer cache */
    void *virtual;           /* kernel virtual address when kmapped */
}
```

Pages

- The kernel uses "struct page" to keep track of the owner of the page
  - User-space process, kernel statically/dynamically allocated data, page cache, etc.
- There is one "struct page" object per physical memory page
  - sizeof(struct page): 64bytes
  - Assuming 8GB of DRAM and 4K-sized pages: 128MB reserved for "struct page" objects (~1.5%)

Zones

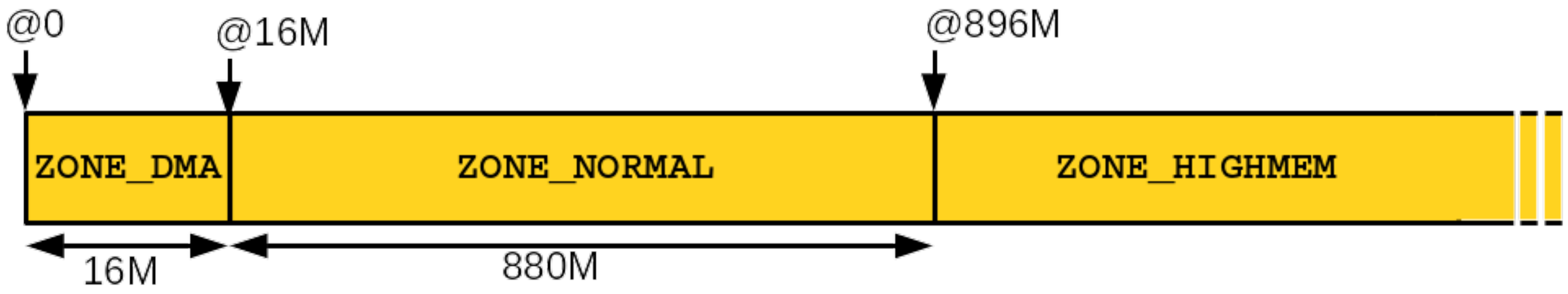
- Certain contexts require certain physical pages due to hardware limitations
  - Some devices can only access the lowest 16MB of physical memory
  - High memory should be mapped before being accessed
- Physical memory is partitioned into zones having the same constraints
  - Zone layout is architecture- and machine-dependent
- Page allocator considers the constraints while allocating pages

Zones

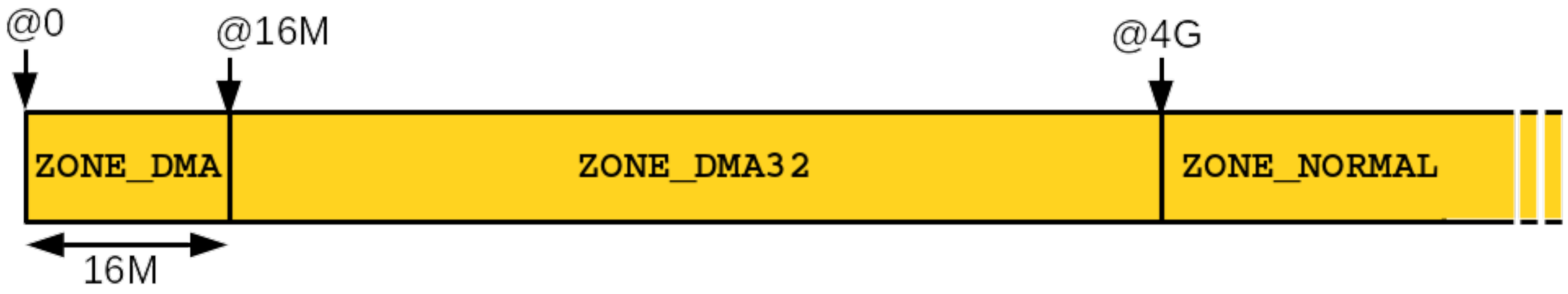
Name	Description
`ZONE_DMA`	Pages can be used for DMA
`ZONE_DMA32`	Pages for 32-bit DMA devices
`ZONE_NORMAL`	Pages always mapped to the address space
`ZONE_HIGHMEM`	Pages should be mapped prior to access

Zones

- x86_32 zones layout



- x86_64 zones layout

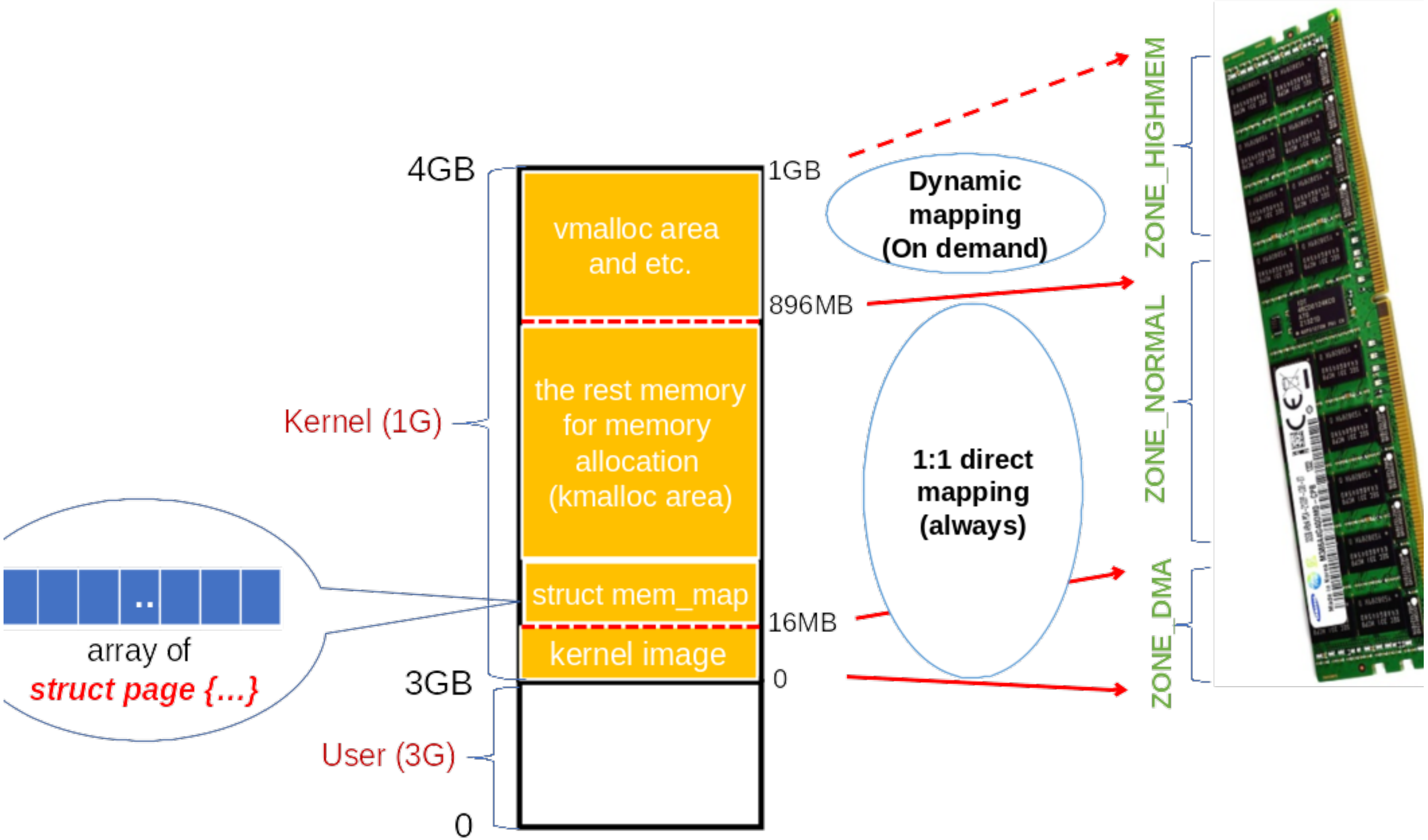


Zones

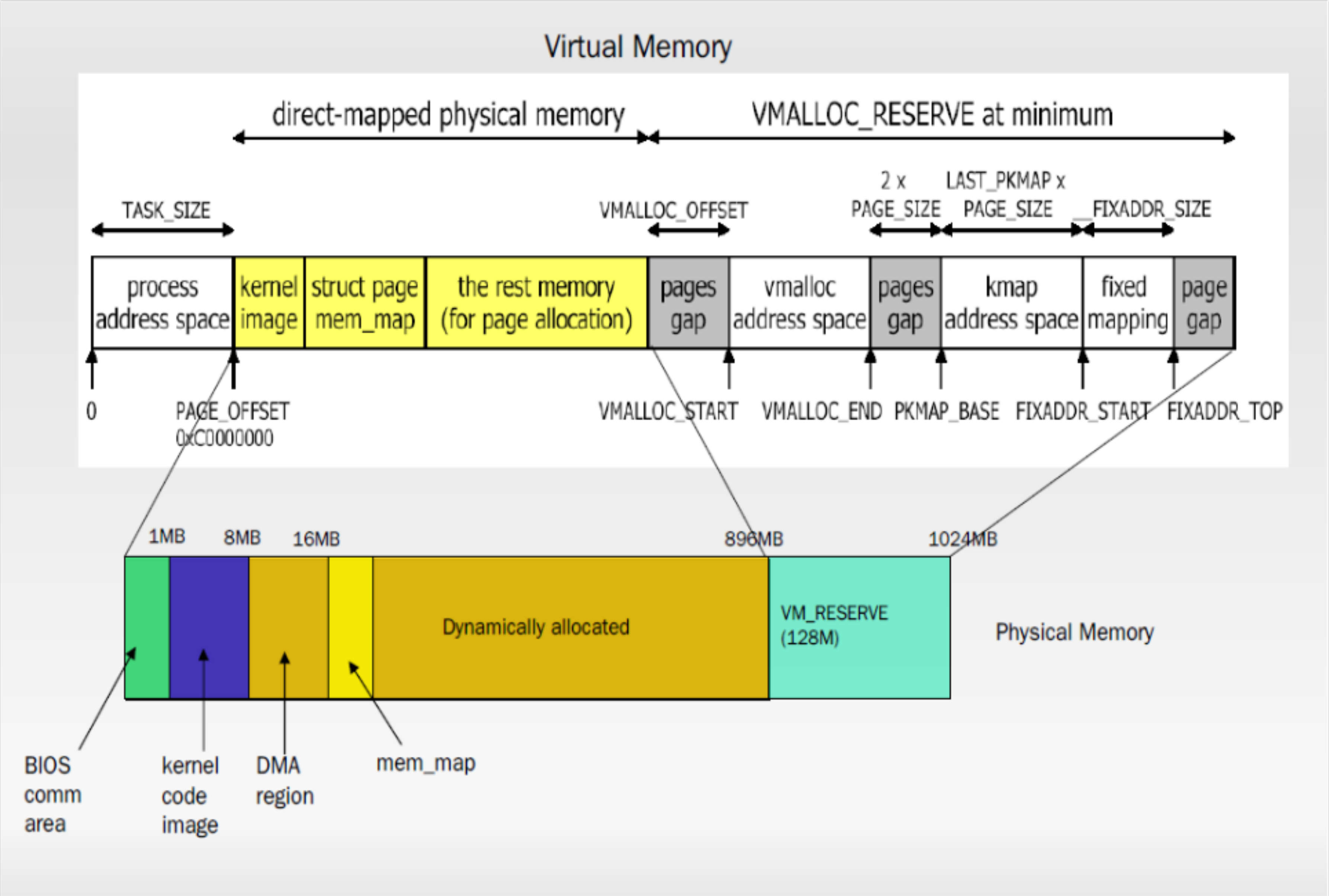
- Each zone is managed with “struct zone” data structure defined in “include/linux/mmzone.h”

```
struct zone {
    const char *name;                   /* Name of this zone */
    unsigned long zone_start_pfn;       /* starting page frame number of the zone */
    unsigned long watermark[NR_WMARK];
                        /* minimum, low, and high watermarks
                         * for per-zone memory allocation */
    spinlock_t lock;   /* protects against concurrent accesses */
    struct free_area free_area[MAX_ORDER];
                        /* list of free pages of different sizes */
};

struct free_area {
    struct list_head    free_list[MIGRATE_TYPES];
    unsigned long       nr_free;
};
```

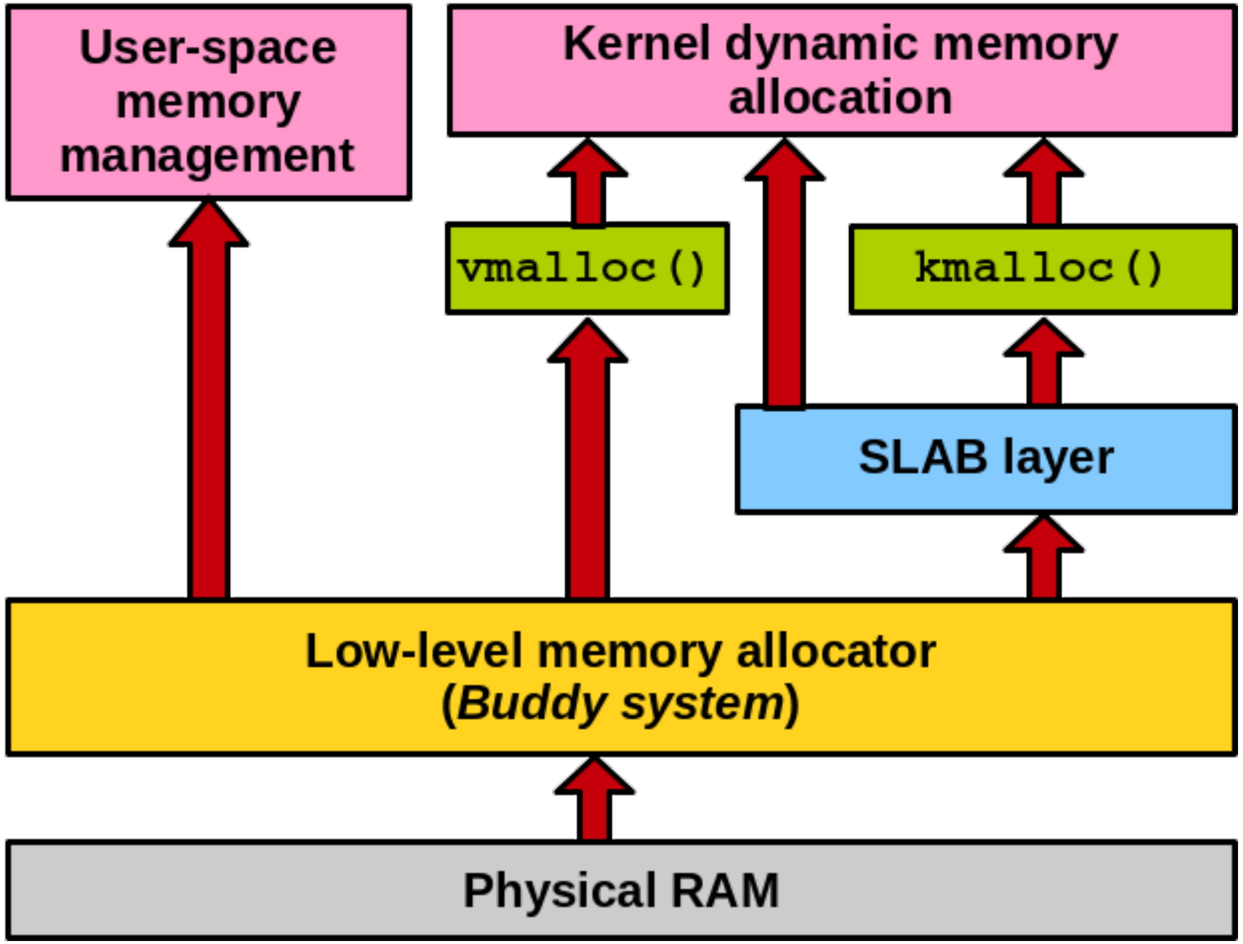
Memory Layout (x86_32)



Memory Layout (x86_32)

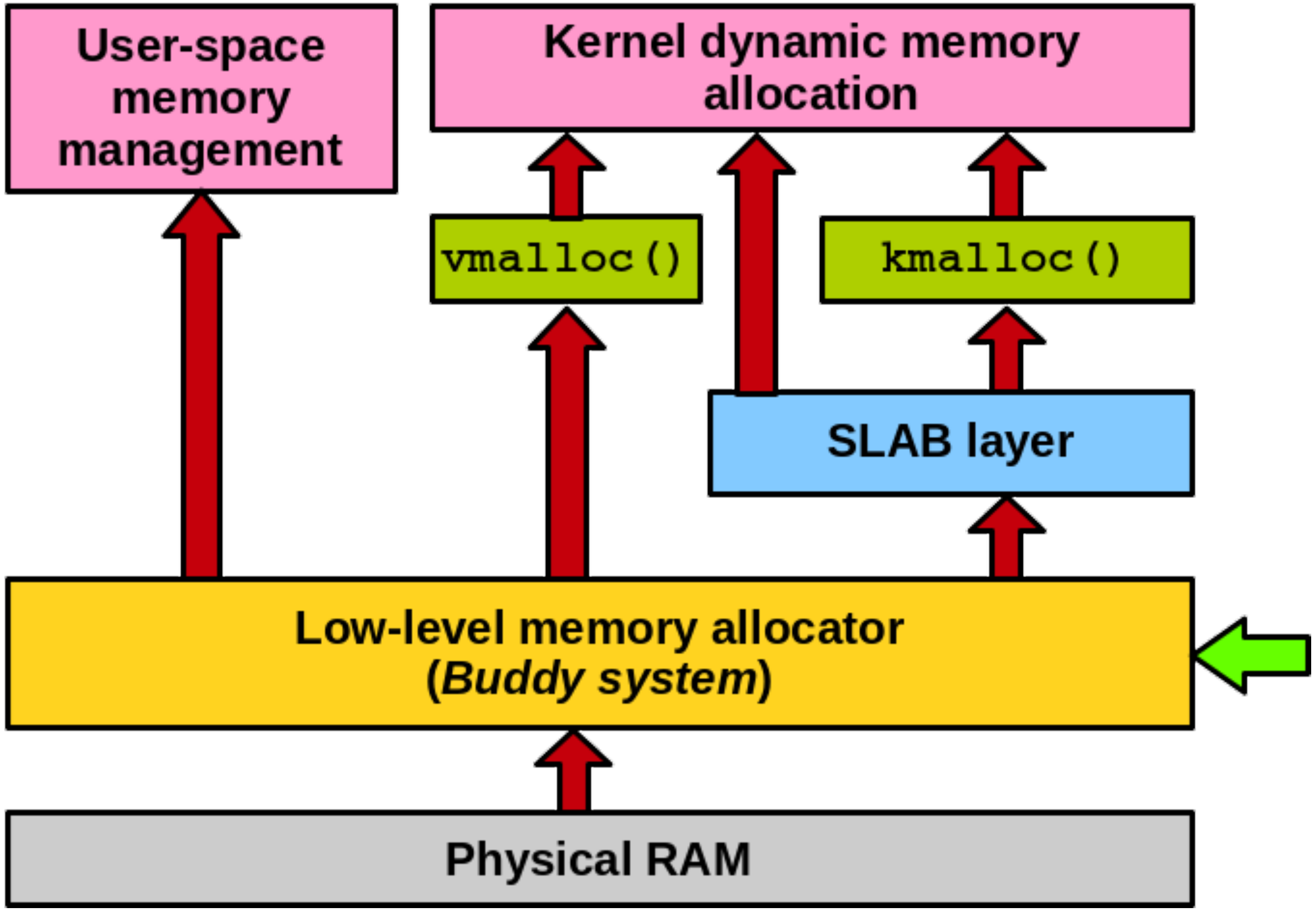


Hierarchy of Memory Allocators



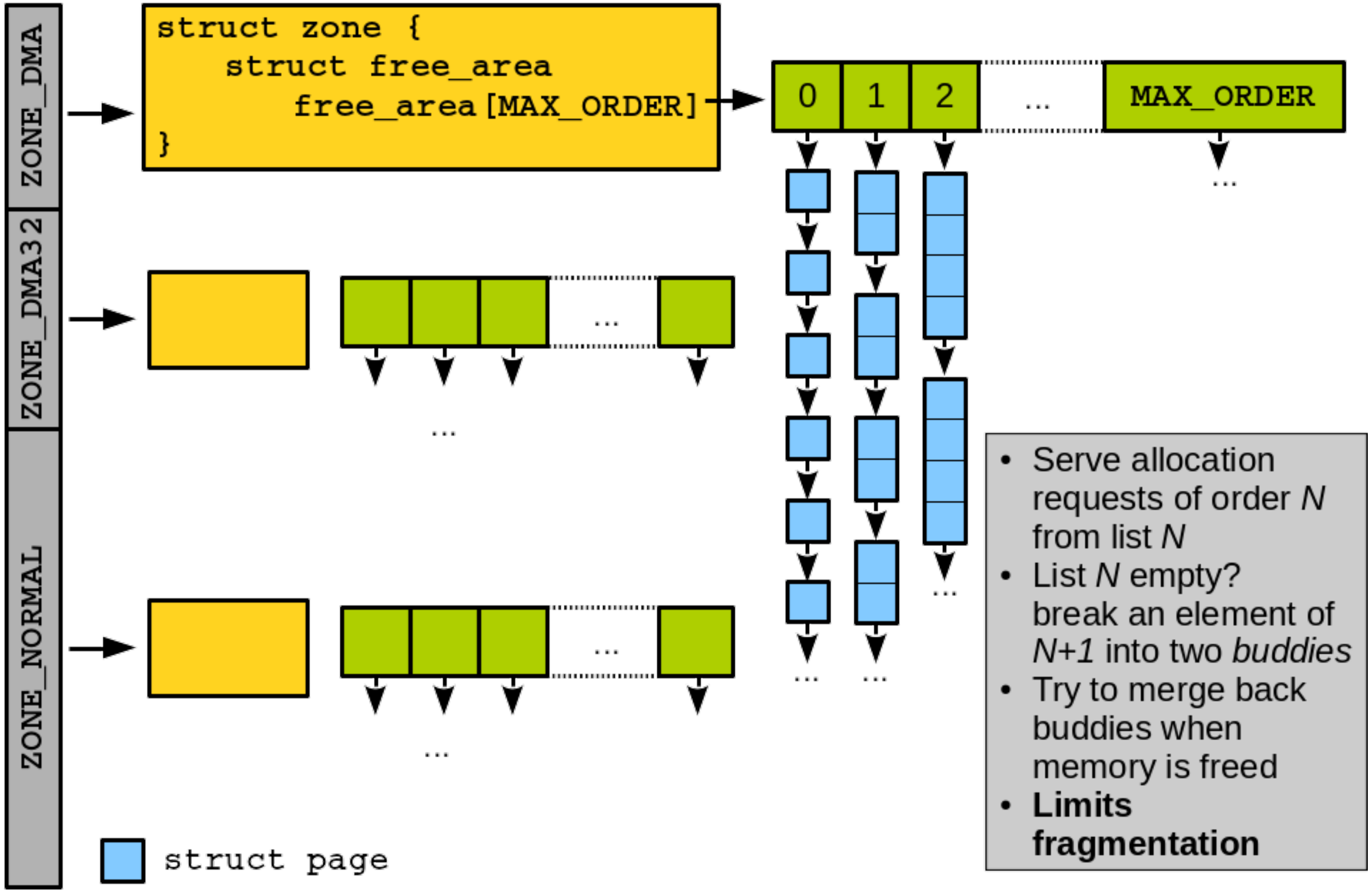
Low-level Memory Allocator

- Buddy system
- Low-level mechanisms to allocate memory at the page granularity
- Interfaces in "include/linux/gfp.h"



Buddy System

- Prevent memory from being fragmented



- Serve allocation requests of order *N* from list *N*
- List *N* empty? break an element of *N+1* into two *buddies*
- Try to merge back buddies when memory is freed
- **Limits fragmentation**

Status of Buddy System

```
$> cat /proc/buddyinfo
Node 0, zone      DMA      1      0      0      1      2      1      1      0
        1      1      2
Node 0, zone    DMA32      9      7      8      9      7     11      8      7
        8      9    525
Node 0, zone   Normal  18184   5454   2414   2628   1562    727    254    721
        999    451   4352
```

Page Allocation / De-allocation

```c
/**
 * Allocate 2^{order} *physically* contiguous pages
 * Return the address of the first allocated `struct page`
 */
struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);
struct page *alloc_page(gfp_t gfp_mask);
```

```c
/**
 * Deallocate 2^{order} *physically* contiguous pages
 * Be careful to put the correct order otherwise corrupt the memory
 */
void __free_pages(struct page *page, unsigned int order);
void __free_page(struct page *page);
```

Page Access

```c
/**
 * Obtain the virtual address to the page frame
 */
void *page_address(struct page *page);


/**
 * Allocate and get the virtual address directly
 */
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);
unsigned long __get_free_page(gfp_t gfp_mask);


/**
 * Free pages using their addresses
 */
void free_pages(unsigned long addr, unsigned int order);
void free_page(unsigned long addr);
```

Allocate Zeroed Page

- By default, the page data is not cleared
- May leak information through the page allocation
- to prevent information leakage, allocate a zero-out page for user-space request
  - unsigned long get_zeroed_page(gfp_t gfp_mask);

gfp_t: get free page flags

- Specify options for memory allocations
  - Action modifier
    - How the memory should be allocated
  - Zone modifier
    - From which zone the memory should be allocated
  - Type flags
    - Combination of action and zone modifiers
    - Generally preferred compared to the direct use of action/zone
  - Defined in "include/linux/gfp.h"

gfp_t: Action Modifiers

Flag	Description
`__GFP_WAIT`	Allocator may sleep
`__GFP_HIGH`	Allocator can access emergency pools
`__GFP_IO`	Allocator can start disk IO
`__GFP_FS`	Allocator can start filesystem IO
`__GFP_NOWARN`	Allocator does not print failure warnings
`__GFP_REPEAT`	Repeat the allocation if it fails
`__GFP_NOFAIL`	The allocation is guaranteed
`__GFP_NORETRY`	No retry on allocation failure

gfp_t: Action Modifiers

- Some action modifiers can be used together

```
struct page *p = alloc_page(__GFP_WAIT | __GFP_FS | __GFP_IO);
```

gfp_t: Zone Modifier

- If not specified, allocated from ZONE_NORMAL or ZONE_DMA (preference to ZONE_NORMAL)

Flag	Description
`__GFP_DMA`	Allocate only from `ZONE_DMA`
`__GFP_DMA32`	Allocate only from `ZONE_DMA32`
`__GFP_HIGHMEM`	Allocate from `ZONE_HIGHMEM` or `ZONE_NORMAL`

gfp_t: Type Flags

- GFP_ATOMIC: Allocate without sleeping
  - __GFP_HIGH
- GFP_NOWAIT: Same to GFP_ATOMIC but does not fall back to the emergency pools

gfp_t: Type Flags

- GFP_NOIO: Can block but does not initiate disk I/O
  - Used in block layer code to avoid recursion
  - __GFP_WAIT
- GFP_NOFS: Can block and perform disk I/O, but does not initiate filesystem operations
  - Used in filesystem code
  - __GFP_WAIT | __GFP_IO
- GFP_KERNEL: Default. can sleep and perform I/O
  - __GFP_WAIT | __GFP_IO | __GFP_FS
- GFP_USER: Normal allocation for user-space memory
- GFP_HIGHUSER: Normal allocation for user-space memory
  - GFP_USER | __GFP_HIGHMEM
- GFP_DMA: Allocate from ZONE_DMA

gfp_t: Cheat Sheet

Context	Solution
Process context, can sleep	`GFP_KERNEL`
Process context, cannot sleep	`GFP_ATOMIC`
Interrupt handler	`GFP_ATOMIC`
Softirq, tasklet	`GFP_ATOMIC`
DMA-able, can sleep	`GFP_DMA | GFP_KERNEL`
DMA-able, cannot sleep	`GFP_DMA | GFP_ATOMIC`

Low-level Memory Allocation Example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/gfp.h>

#define PRINT_PREF              "[LOWLEVEL]: "
#define PAGES_ORDER_REQUESTED   3
#define INTS_IN_PAGE            (PAGE_SIZE/sizeof(int))

unsigned long virt_addr;

static int __init my_mod_init(void)
{
    int *int_array;
    int i;

    printk(PRINT_PREF "Entering module.\n");

    virt_addr = __get_free_pages(GFP_KERNEL, PAGES_ORDER_REQUESTED);
    if(!virt_addr) {
        printk(PRINT_PREF "Error in allocation\n");
        return -1;
    }

    int_array = (int *)virt_addr;
    for(i=0; i<INTS_IN_PAGE; i++)
        int_array[i] = i;

    for(i=0; i<INTS_IN_PAGE; i++)
        printk(PRINT_PREF "array[%d] = %d\n", i, int_array[i]);

    return 0;
}

static void __exit my_mod_exit(void)
{
    free_pages(virt_addr, PAGES_ORDER_REQUESTED);
    printk(PRINT_PREF "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

High Memory

- On x86_32, physical memory above 896MB is not permanently mapped within the kernel address space
  - due to limited size of the addr space and the 1/3GB kernel/user-space memory split
- Before using them, pages from highmem should be mapped to the addr space

High Memory


```c
/**
 * Permanent mappings
 * - Maps the `page` and return the address to the `page`
 * - May sleep
 * - Has a limited number of slots
 */
void *kmap(struct page *page);
void  kunmap(struct page *page);

/**
 * Temporary mappings
 * - Use a per-CPU pre-reserved mapping slots
 * - Disable kernel preemption
 * - Should not sleep while holding the mapping
 */
void *kmap_atomic(struct page *page);
void  kunmap_atomic(void *addr);
```

High Memory Example

```
struct page *my_page;
void *my_addr;

my_page = alloc_page(GFP_HIGHUSER);
my_addr = kmap(my_page);

memcpy(my_addr, buffer, sizeof(buffer));

kunmap(my_page);
__free_page(my_page);
```

kmalloc() / kfree()

- void *kmalloc(size_t size, gfp_t flags)
  - Allocates byte-sized chunks of memory
  - Similar to the user-space malloc()
    - » Returns a pointer to the first allocated byte on success
    - » Returns NULL on err
  - Allocated memory is physically contiguous
- void kfree(const void *ptr)
  - Free the memory allocated with kmalloc()

Example

```c
struct my_struct *p;

p = kmalloc(sizeof(*p), GFP_KERNEL);
if (!p) {
    /* Handle error */
} else {
    /* Do something */
    kfree(p);
}
```

vmalloc()

- void *vmalloc(unsigned long size)
  - allocates virtually contiguous chunk of memory
    - » May not be physically contiguous
    - » Cannot be used for I/O buffers requiring physically contiguous memory
  - Used for allocating a large virtually contiguous memory
  - May sleep so cannot be called from interrupt context
- void vfree(const void *addr)

vmalloc()

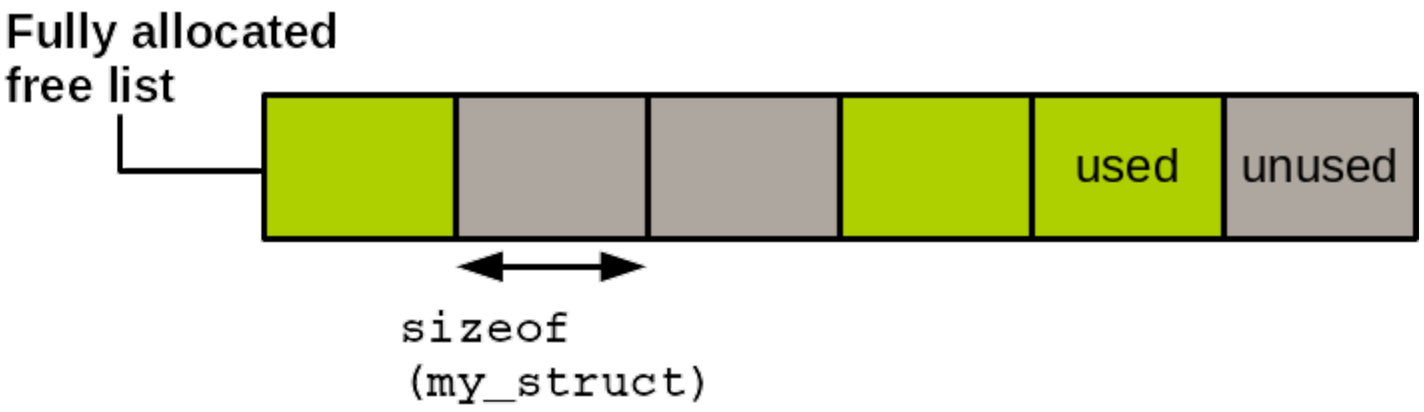
- However, most of the kernel uses kmalloc() for performance reasons
  - Pages allocated with kmalloc() are directly mapped
  - Less overhead in virtual-to-physical mapping setup and translation
- vmalloc() is still needed to allocate large protions of memory
- Declared in include/linux/vmalloc.h

vmalloc() vs. kmalloc()

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>

#define PRINT_PREF "[KMALLOC_TEST]: "

static int __init my_mod_init(void)
{
    unsigned long i;
    void *ptr;

    printk(PRINT_PREF "Entering module.\n");

    for(i=1;;i*=2) {
        ptr = kmalloc(i, GFP_KERNEL);
        if(!ptr) {
            printk(PRINT_PREF "could not allocate %lu bytes\n", i);
            break;
        }
        kfree(ptr);
    }

    return 0;
}

static void __exit my_mod_exit(void)
{
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Slab Allocator

- Allocating/freeing data structures is done very often in the kernel
- Q: how to make memory allocation faster?
- Caching using a free lislt
  - Block of pre-allocated memory for a given type of data structure
  - Allocate from the free list = pick an element in the free list
  - Deallocate an element = add an element to the free list



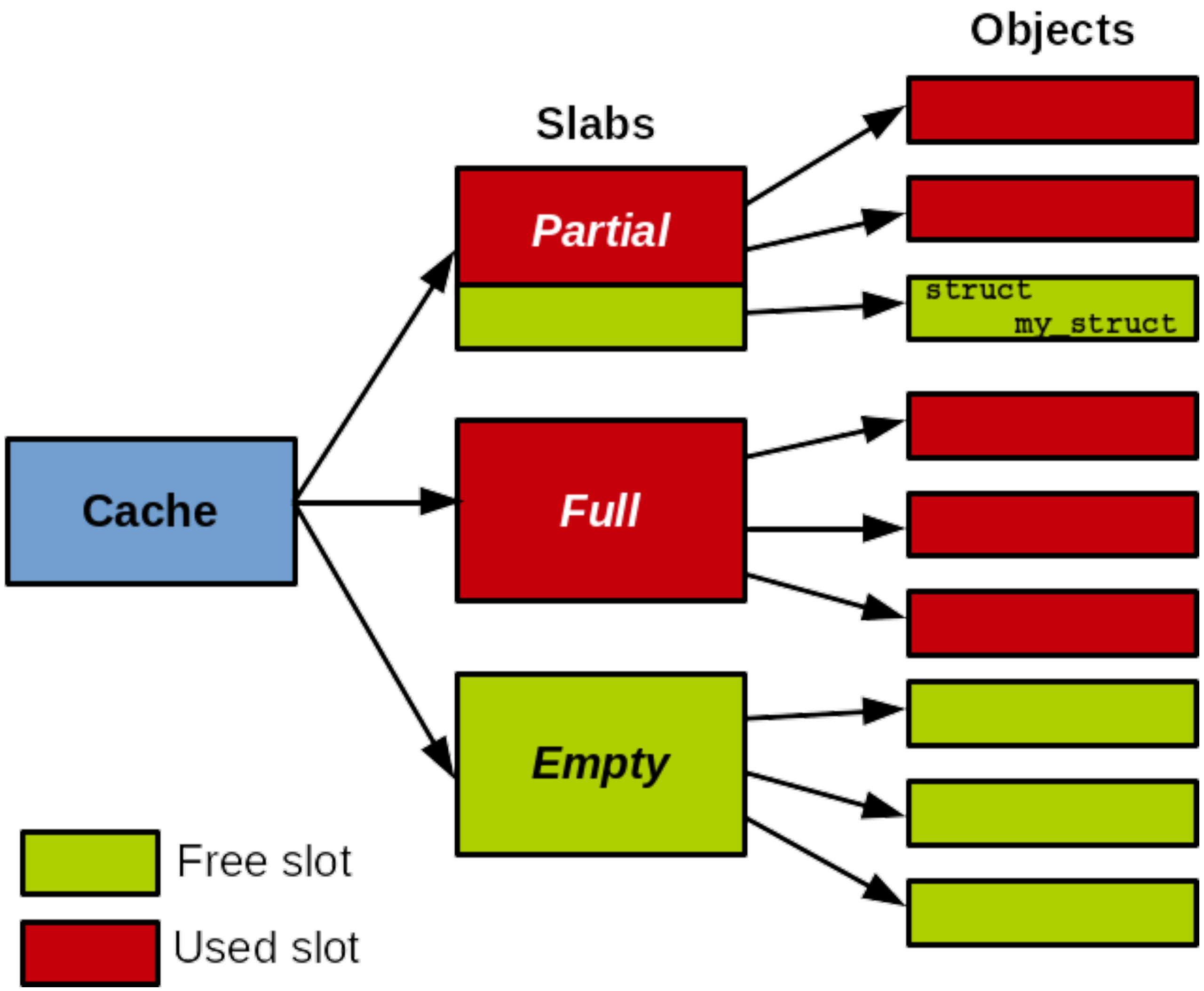
Slab Allocator

- Issue with ad-hoc free lists: no global control
  - When and how to free free lists?
- Slab allocator
  - Generic allocation caching interface
  - Cache objects of a data structure type
    - » e.g., an object cache for "struct task_struct"
  - Consider the data structure size, page size, NUMA, and cache coloring …

Slab Allocator

- A cache has one or more slabs
  - One or several physically contiguous pages
- Slabs contain objects
- A slab may be empty, partially full, or full
- allocate objects from the partially full slabs to prevent memory fragmentation

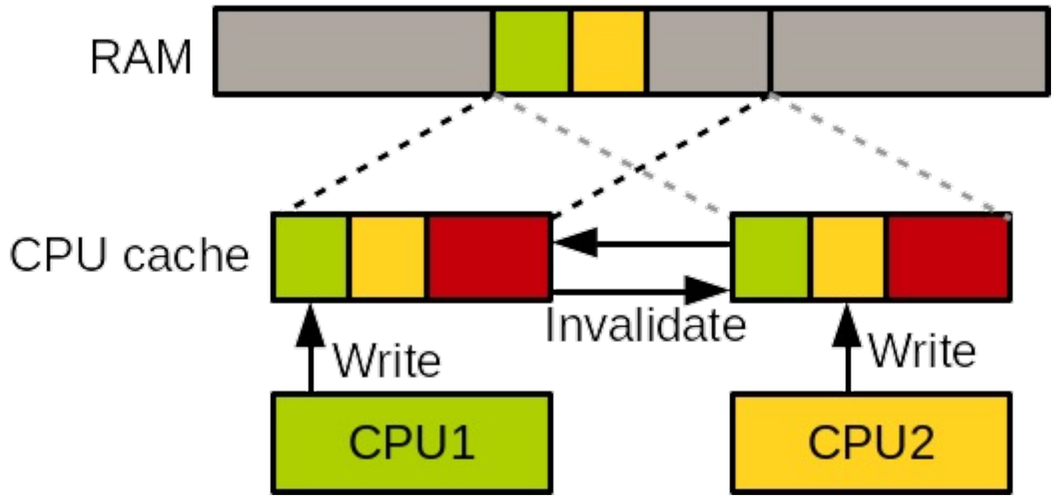
Slab Allocator



```c
/**
 * Create a cache for a data structure type
 */
struct kmem_cache *kmem_cache_create(
    const char *name,     /* Name of the cache */
    size_t size,          /* Size of objects */
    size_t align,         /* Offset of the first element
                             within pages */
    unsigned long flags, /* Options */
    void (*ctor)(void *) /* Constructor */
);


/**
 * Destroy the cache
 * - Should be only called when all slabs in the cache are empty
 * - Should not access the cache during the destruction
 */
void kmem_cache_destroy(struct kmem_cache *cachep);
```

Slab Allocator

- SLAB_HW_CACHEALIGN
  - Align objects to the cache line to prevent false sharing
  - Increase memory footprint



Slab Allocator

- SLAB_POISON
  - Initially fill slabs with a known value(0xa5a5a5a5) to detect accesses to uninitialized memory
- SLAB_RED_ZONE
  - Put extra padding around objects to detect overflows
- SLAB_PANIC
  - Panic if allocation fails
- SLAB_CACHE_DMA
  - Allocate from DMA-enabled memory

Slab Allocator

```
/**
 * Allocate an object from the cache
 */
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);

/**
 * Free an object allocated from a cache
 */
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

Example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#define PRINT_PREF  "[SLAB_TEST] "
struct my_struct {
    int int_param;
    long long_param;
};

static int __init my_mod_init(void)
{
    int ret = 0;
    struct my_struct *ptr1, *ptr2;
    struct kmem_cache *my_cache;

    printk(PRINT_PREF "Entering module.\n");

    my_cache = kmem_cache_create("lkp-cache", sizeof(struct my_struct),
        0, 0, NULL);
    if(!my_cache)
        return -1;

    ptr1 = kmem_cache_alloc(my_cache, GFP_KERNEL);
    if(!ptr1){
        ret = -ENOMEM;
        goto destroy_cache;
    }

    ptr2 = kmem_cache_alloc(my_cache, GFP_KERNEL);
    if(!ptr2){
        ret = -ENOMEM;
        goto freeptr1;
    }

    ptr1->int_param = 42;
    ptr1->long_param = 42;
    ptr2->int_param = 43;
    ptr2->long_param = 43;

    printk(PRINT_PREF "ptr1 = {%d, %ld} ; ptr2 = {%d, %ld}\n", ptr1->int_param,
        ptr1->long_param, ptr2->int_param, ptr2->long_param);

    kmem_cache_free(my_cache, ptr2);

freeptr1:
    kmem_cache_free(my_cache, ptr1);

destroy_cache:
    kmem_cache_destroy(my_cache);

    return ret;
}

static void __exit my_mod_exit(void)
{
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Slab Allocator Status

```
$> 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>
nf_conntrack          575     675     320    25      2 : tunables     0     0     0 :
        slabdata      27      27        0
rpc_inode_cache        46      46     704    46      8 : tunables     0     0     0 :
        slabdata       1       1        0
fat_inode_cache       133     176     744    44      8 : tunables     0     0     0 :
        slabdata       4       4        0
fat_cache               0       0      40   102      1 : tunables     0     0     0 :
        slabdata       0       0        0
squashfs_inode_cache    368     368     704    46      8 : tunables     0     0     0 :
        slabdata       8       8        0
kvm_async_pf            0       0     136    30      1 : tunables     0     0     0 :
        slabdata       0       0        0
kvm_vcpu                0       0   15104     2      8 : tunables     0     0     0 :
        slabdata       0       0        0
kvm_mmu_page_header       0       0     168    24      1 : tunables     0     0     0 :
        slabdata       0       0        0
x86_emulator            0       0    2672    12      8 : tunables     0     0     0 :
        slabdata       0       0        0
```

Slab Allocator Variants

- SLOB (Simple List of Blocks)
  - Used in early Linux version
  - Low memory footprint, suitable for embedded systems
- SLAB
  - from 1999
  - Cache-friendly
- SLUB
  - in 2008
  - Improved scalability over SLAB on many cores

Per-CPU Data Structure

- Allow each core to have their own values
  - No locking required
  - Reduce cache thrashing
- Implemented through arrays in which each index corresponds to a CPU

```
unsigned long my_percpu[NR_CPUS];
int cpu;
cpu = get_cpu();    /* get_cpu() disables kernel preemption
                     * otherwise `cpu` might become incorrect
                     * across context switches */
my_percpu[cpu]++;
put_cpu();          /* put_cpu() enables kernel preemption */
```

Per-CPU API

- Defined in `include/linux/percpu.h`

```
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(name, type);

get_cpu_var(name);   /* Start accessing the per-CPU variable */
put_cpu_var(name);   /* Done accessing the per-CPU variable */

/* Access per-CPU data through pointers */
get_cpu_ptr(name);
put_cpu_ptr(name);

per_cpu(name, cpu); /* Access other CPU's data */
```

- Example

```
DEFINE_PER_CPU(int, my_var);

int cpu;
for (cpu = 0; cpu < NR_CPUS; cpu++)
    per_cpu(my_var, cpu) = 0;

printk("%d\n", get_cpu_var(my_var)++);
put_cpu_var(my_var);

int *my_var_ptr;
my_var_ptr = get_cpu_ptr(my_var);
put_cpu_ptr(my_var_ptr);
```

Stack

- Each process has
  - A userspace stack for execution
  - A kernel stack for in-kernel execution
- Userspace stack is large and grows dynamically
- Kernel-stack is small and has a fixed-size → 2 pages
- Interrupt stack is for interrupt handlers → 1 page per CPU
- Reduce kernel stack usage to a minimum
  - Local variables and function parameters

Takeaway

- Need physically contig memory
  - kmalloc(), alloc_page()
- Virtually contig
  - vmalloc()
- Frequently creating/destroying large amount of the same data structures
  - Use SLAB allocator
- Need to allocate from high memory
  - Use alloc_page() then kmap()/kmap_atomic()

Further Readings

- Virtual Memory: What is Virtual Memory?
- 20 years of Linux virtual memory
- complete virtual memory map x86_64 architecture