Memory Management

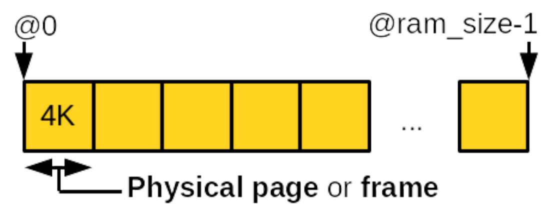
Dongyoon Lee

Today: Memory Management

- Pages and zones
- Page allocation
- kmalloc, vmalloc
- 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 the memory management unit (MMU) support
  - **4 KB** in general, some are 2 MB and 1 GB: `getconf PAGESIZE`

Pages

- Each **physical page** is represented by `struct page`
- 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;         /* virtual address */
}
```

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)`: 64 bytes
  - Assuming 8GB of RAM and 4K-sized pages: 128MB reserved for `struct page` objects (~1.5%)

Zones

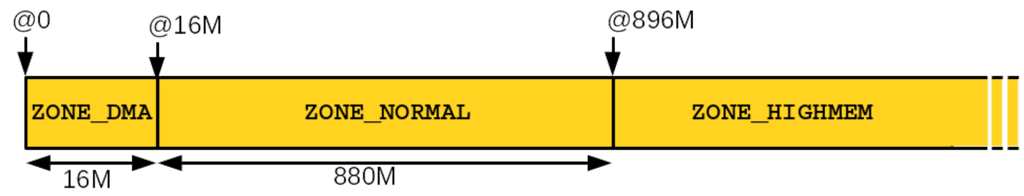
- Certain contexts require certain physical pages due to hardware limitations
  - Some devices can only access the lowest 16 MB of physical memory
  - High memory should be mapped before being accessed
- Physical memory is partitioned into **zones** having the same constraints
  - Zone layout is architecture-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

Zones

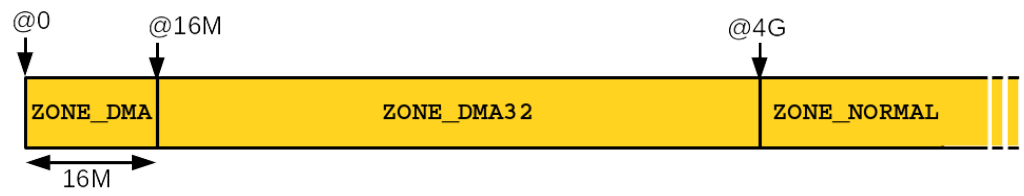
Name	Description
`ZONE_DMA`	Pages can be used for DMA
`ZONE_DMA32`	Pages for 32-bit DMA devices
`ZONE_NORMAL`	Pages always mapped to the address space
`ZONE_HIGHMEM`	Pages should be mapped prior to access

Zones

- x86_32 zones layout



- x86_64 zones layout

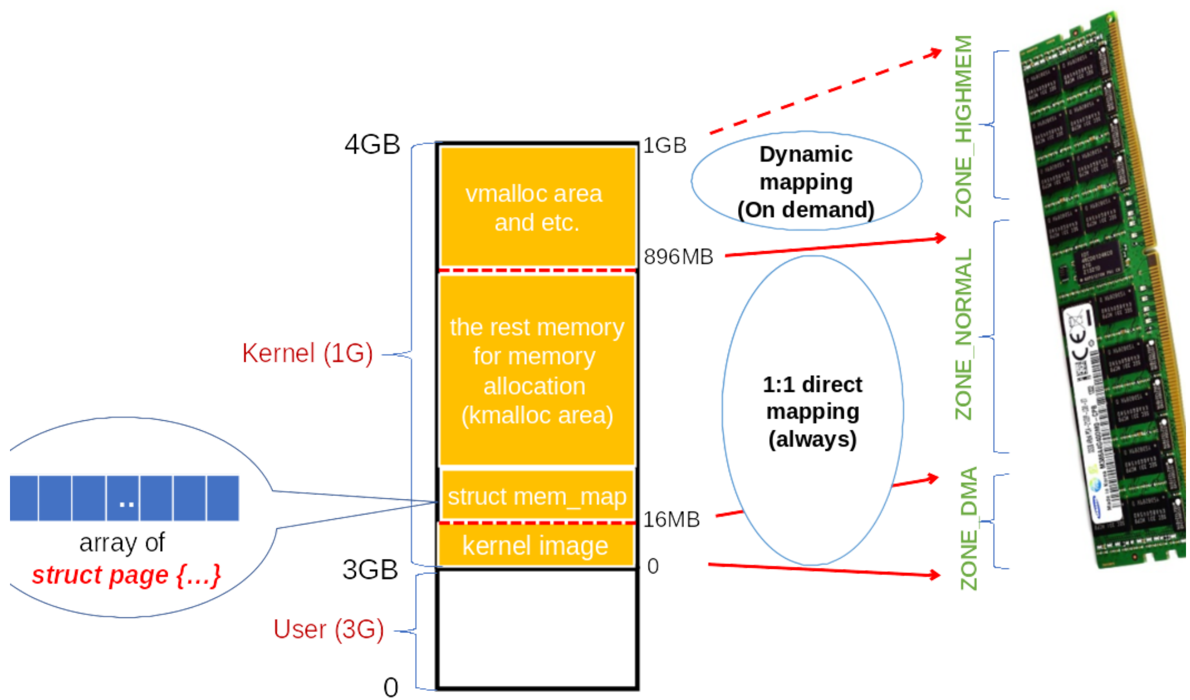


Zones

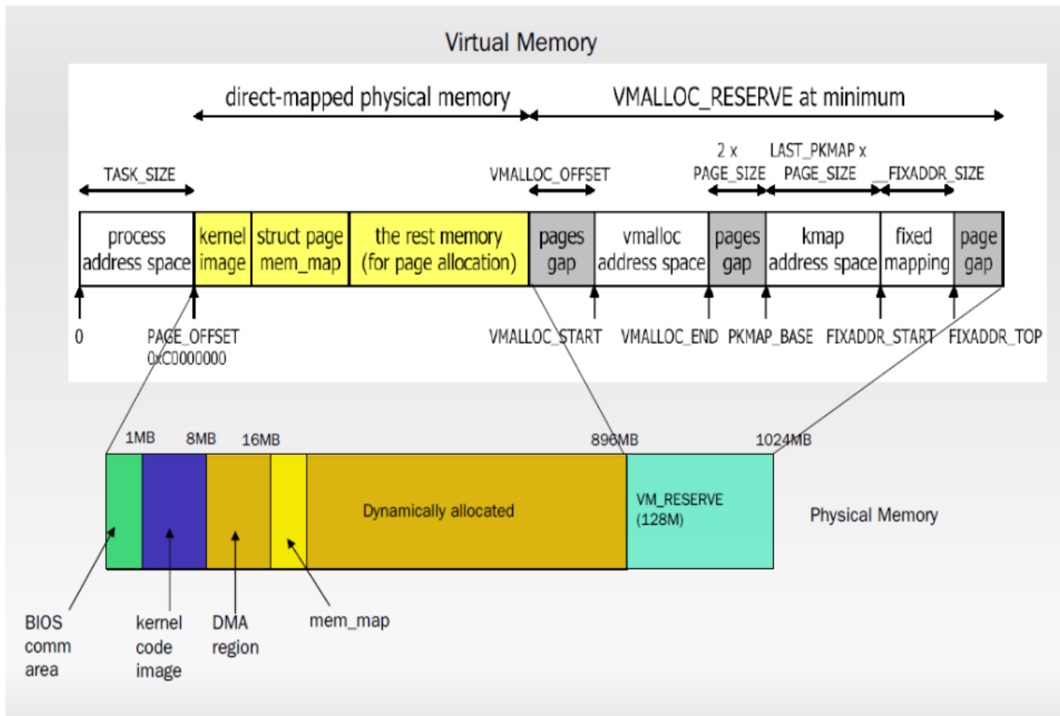
- Each zone is managed with `struct zone` data structure defined in `include/linux/mmzone.h`

```
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 */
}
```

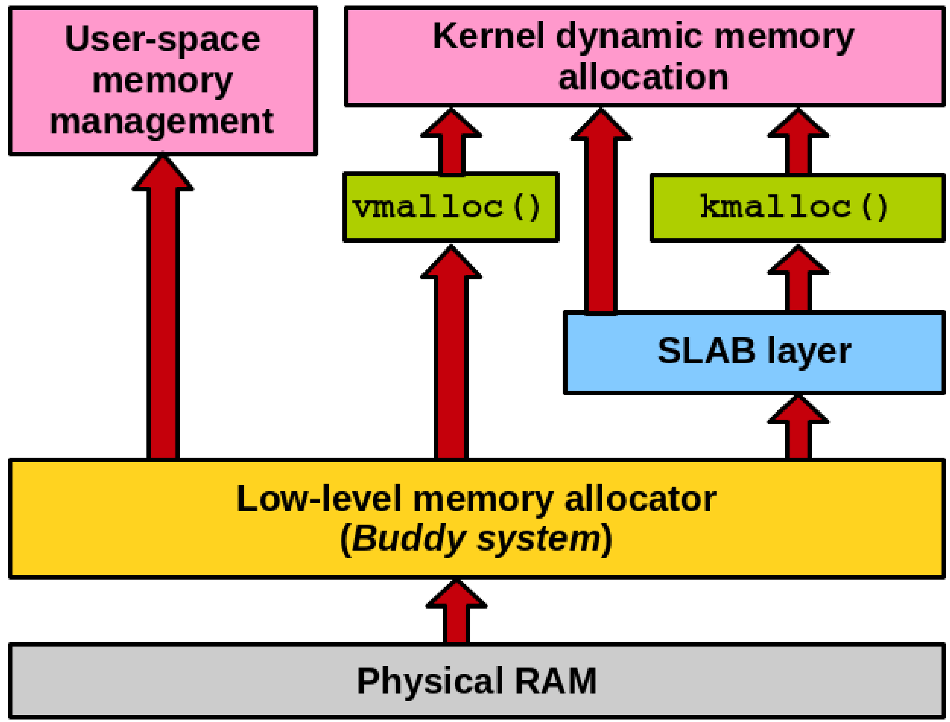
Memory layout (x86_32)



Memory layout (x86_32)



Hierarchy of memory allocators

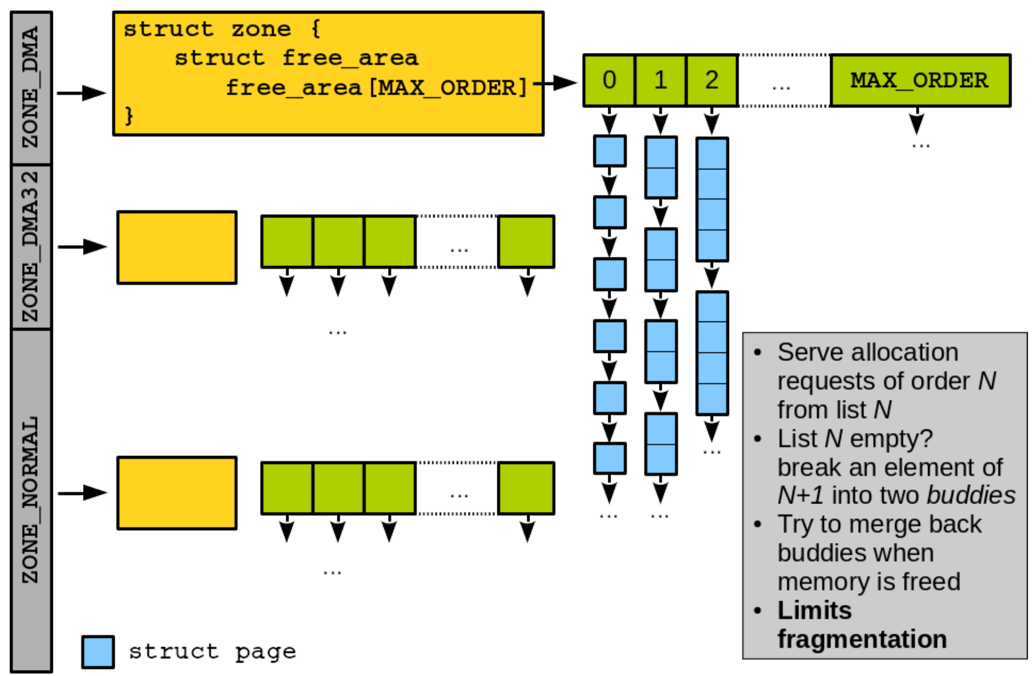


Low-level memory allocator (Buddy system)

- Low-level mechanisms to allocate memory at the *page* granularity
- Interfaces in `include/linux/gfp.h`

Buddy system

- Prevent memory from being fragmented



Status of Buddy System

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

Page allocation / deallocation

```
/**
 * 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);


/**
 * 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

```
/**
 * 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 allocation
  - 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 modifiers

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`

- If not specified, allocated from `ZONE_NORMAL` or `ZONE_DMA` (high preference to `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 IO
    - Used in block layer code to avoid recursion
    - `__GFP_WAIT`
- `GFP_NOFS` : Can block and perform disk IO, but does not initiate filesystem operations
    - Used in filesystem code
    - `__GFP_WAIT | __GFP_IO`

`gfp_t` : type flags

- `GFP_KERNEL` : Default. Can sleep and perform IO
  - `__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

```
#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;
    }
```

Low-level memory allocation example

```
    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 896 MB is not permanently mapped within the kernel address space
  - Due to the limited size of the address space and the 1/3 GB kernel/user-space memory split
- Before use, pages from highmem should be mapped to the address space

High memory

```
/**
 * 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 error
  - Allocated memory is *physically contiguous*
- `void kfree(const void *ptr)`
  - Free the memory allocated with `kmalloc()`

kmalloc() / kfree()

- Example

```
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 chunk
  - May sleep so cannot be called from interrupt context
- Free using `vfree()`
  - `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 portions of memory
- Declared in include/linux/vmalloc.h

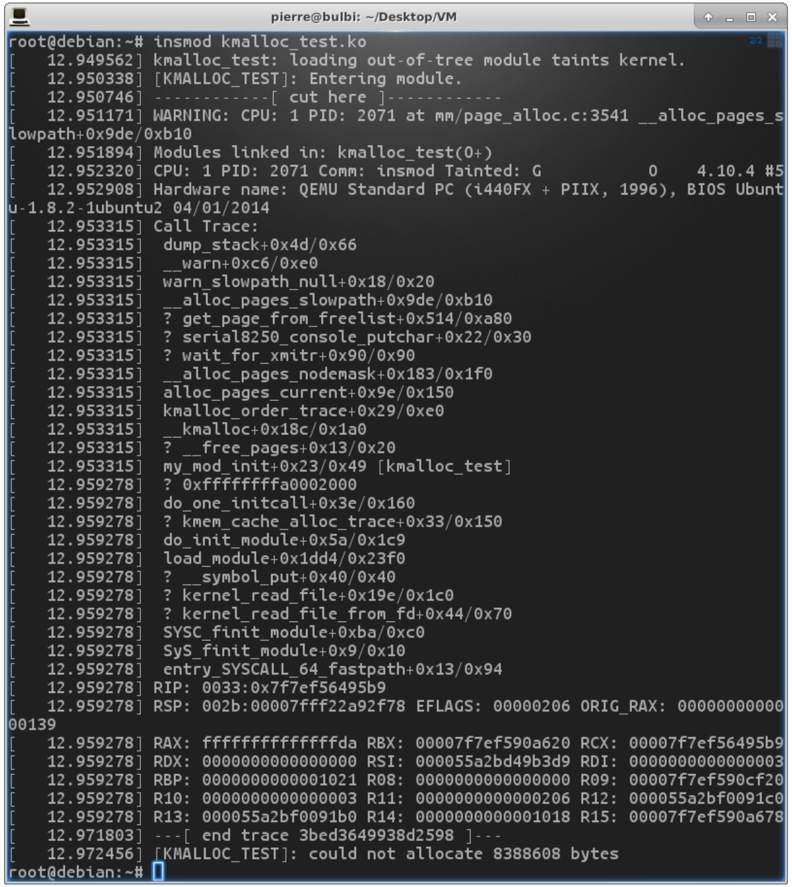
vmalloc() vs. kmalloc()

```
#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);
    }
```

`vmalloc()` vs. `kmalloc()`

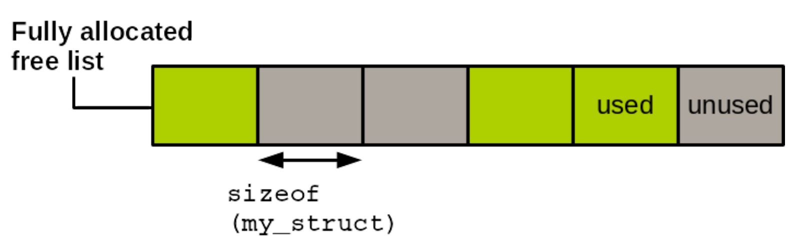
```
    return 0;
}

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");
```

`vmalloc()` vs. `kmalloc()`



Slab allocator

- Allocating/freeing data structures is done very often in the kernel
- **Q: how to make memory allocation faster? → caching using a free list**
- **Free lists:**
  - 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



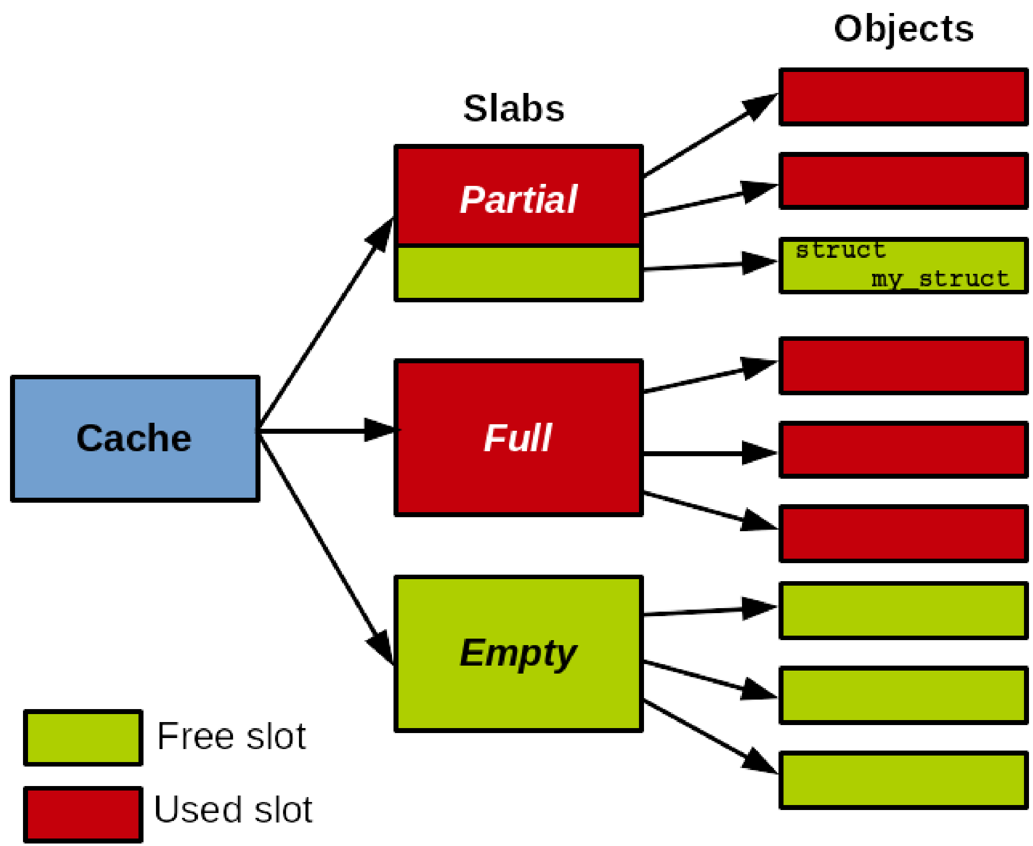
Slab allocator

- 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

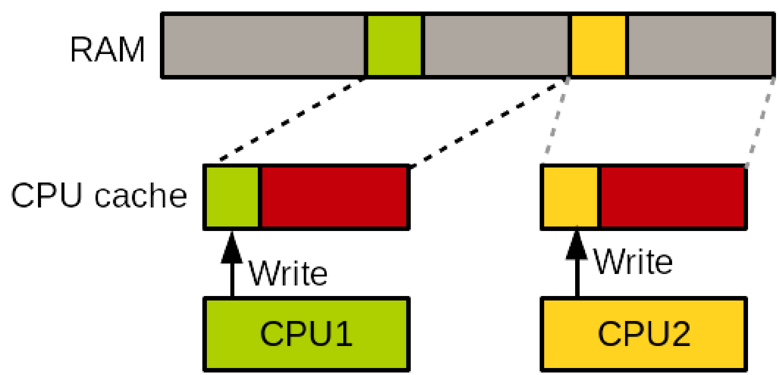


Slab allocator

```
/**
 * 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);
```

Slab allocator example

```
#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;
```

Slab allocator example

```
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);
```

Slab allocator example

```
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");
```

Status of Slab allocator

```
$> sudo cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <l
nf_conntrack           575    675    320   25    2 : tunables    0    0    0 : slabdata     27
rpc_inode_cache         46     46    704   46    8 : tunables    0    0    0 : slabdata      1
fat_inode_cache        133    176    744   44    8 : tunables    0    0    0 : slabdata      4
fat_cache                0      0     40  102    1 : tunables    0    0    0 : slabdata      0
squashfs_inode_cache    368    368    704   46    8 : tunables    0    0    0 : slabdata
kvm_async_pf             0      0    136   30    1 : tunables    0    0    0 : slabdata      0
kvm_vcpu                 0      0  15104    2    8 : tunables    0    0    0 : slabdata      0
kvm_mmu_page_header       0      0    168   24    1 : tunables    0    0    0 : slabdata      0
x86_emulator             0      0   2672   12    8 : tunables    0    0    0 : slabdata      0
x86_fpu                  0      0   4160    7    8 : tunables    0    0    0 : slabdata      0
ext4_groupinfo_4k     3724   3724    144   28    1 : tunables    0    0    0 : slabdata    133
i915_dependency        512    512    128   32    1 : tunables    0    0    0 : slabdata     16
execute_cb               0      0    128   32    1 : tunables    0    0    0 : slabdata      0
i915_request          1964   1988    576   28    4 : tunables    0    0    0 : slabdata     71
intel_context          630    630    384   42    4 : tunables    0    0    0 : slabdata     15
fuse_inode             156    156    832   39    8 : tunables    0    0    0 : slabdata      4
btrfs_delayed_node       0      0    312   26    2 : tunables    0    0    0 : slabdata      0
btrfs_ordered_extent      0      0    416   39    4 : tunables    0    0    0 : slabdata
btrfs_free_space_bitmap      0      0  12288    2    8 : tunables    0    0    0 : slabdata
btrfs_inode              0      0   1184   27    8 : tunables    0    0    0 : slabdata      0
fsverity_info            0      0    256   32    2 : tunables    0    0    0 : slabdata      0
```

Slab allocator variants

- **SLOB (Simple List Of Blocks)**
  - Used in early Linux version (from 1991)
  - Low memory footprint, suitable for embedded systems
- **SLAB**
  - Integrated in 1999
  - Cache-friendly
- **SLUB**
  - Integrated 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 */
```

Per-CPU data structure

- 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 user-space stack for execution
  - A kernel stack for in-kernel execution
- **User-space stack** is large and grows dynamically
- **Kernel-stack** is small and has a fixed-size → two pages (= 8 KB)
- **Interrupt stack** is for interrupt handlers → one page for each CPU
- Reduce kernel stack usage to a minimum
  - Local variables and function parameters

Take-away

- Need physically contiguous memory
  - `kmalloc()` or `alloc_page()` series
- Virtually contiguous chunk
  - `vmalloc()`
- Frequently creating/destroying large amount of the same data structures
  - Use the slab allocator
- Need to allocate from high memory
  - Use `alloc_page()` then `kmap()/kmap_atomic()`

Further readings

- [Virtual Memory: 3 What is Virtual Memory?]
- [Complete virtual memory map x86_64 architecture]

Next Lecture

- Process Address Space