<Slides download>
http://www.pf.is.s.u-tokyo.ac.jp/class.html

Advanced Operating Systems

#3

Shinpei Kato
Associate Professor

Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo

Course Plan

- Multi-core Resource Management
- Many-core Resource Management
- GPU Resource Management
- Virtual Machines
- Distributed File Systems
- High-performance Networking
- Memory Management
- Network on a Chip
- Embedded Real-time OS
- Device Drivers
- Linux Kernel

Schedule

1. 2018.9.28 Introduction + Linux Kernel (Kato)
2. 2018.10.5 Linux Kernel (Chishiro)
3. 2018.10.12 Linux Kernel (Kato)
4. 2018.10.19 Linux Kernel (Kato)
5. 2018.10.26 Linux Kernel (Kato)
6. 2018.11.2 Advanced Research (Chishiro)
7. 2018.11.9 Advanced Research (Chishiro)
8. 2018.11.16 (No Class)
9. 2018.11.23 (Holiday)
10. 2018.11.30 Advanced Research (Kato)
11. 2018.12.7 Advanced Research (Kato)
12. 2018.12.14 Advanced Research (Chishiro)
13. 2018.12.21 Linux Kernel
14. 2019.1.11 Linux Kernel
15. 2019.1.18 (No Class)
16. 2019.1.25 (No Class)

Memory Management

Abstracting Memory Resources

/* The case for Linux */

Acknowledgement:

Prof. Pierre Olivier, ECE 4984, Linux Kernel Programming, Virginia Tech

Outline

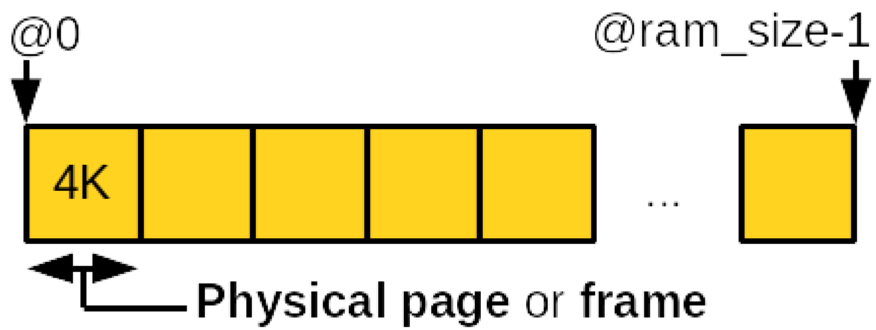
Outline

Pages and zones

Pages

- Memory allocation in the kernel is different from user space
  - How to handle memory allocation failures?
  - In some situations the kernel cannot sleep
  - Etc.
- The **page** is the basic unit for memory management by the MMU, and thus the kernel
  - Page size is machine dependent
    - Typical values for x86 are **4K**, 2M, and 1G
    - Get the page size on your machine:

```
Getconf PAGESIZE
```



Pages and zones

Pages: `struct page`

- Each **physical page** is represented by a **`struct page`**
  - Most of the pages are used for (1) kernel/userspace memory (*anonymous mapping*) or (2) file mapping
- Simplified version:

```
struct page {
  unsigned long flags;
  unsigned counters;
  atomic_t _mapcount;
  unsigned long private;
  struct address_space *mapping;
  pgoff_t index;
  struct list_head lru;
  void *virtual;
}
```

- `flags`: page status (permission, dirty, etc.)
- `counters`: usage count
- `private`: private mapping
- `mapping`: file mapping
- `index`: offset within mapping
- `virtual`: virtual address

- More info on `struct page`: [1]

Pages and zones

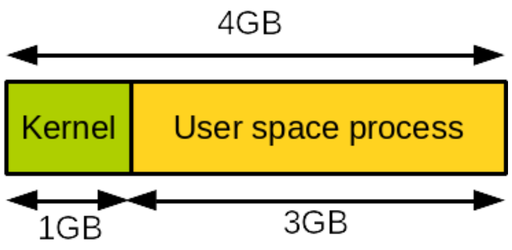
Pages (2)

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

Pages and zones

Zones

- Because of hardware limitations, **only certain physical pages can be used in certain contexts**
  - Example: on x86 some DMA-enabled devices can only access the lowest 16M part of physical memory
  - On x86 32 the kernel address space area sometimes cannot map all physical RAM (1/3 model)

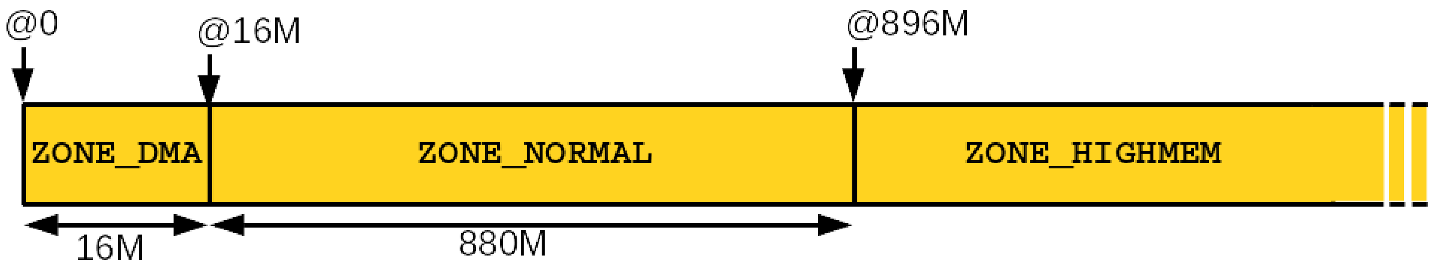


- Physical memory is divided into **zones**:
  - ZONE_DMA: pages with which DMA can be used
  - ZONE_DMA32: memory for other DMA limited devices
  - ZONE_NORMAL: page always mapped in the address space
  - ZONE_HIGHMEM: pages only mapped temporary
- Zones layout is completely architecture dependent

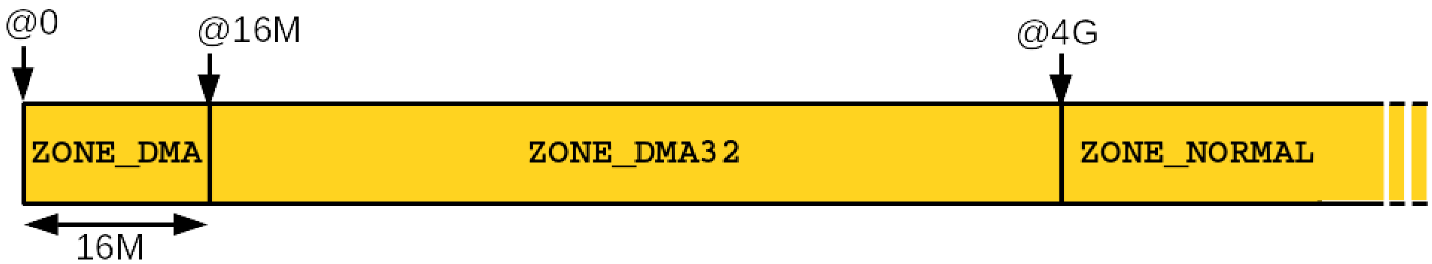
Pages and zones

Zones (2)

- x86_32 zones layout:



- x86_64 zones layout:



- Some memory allocation requests must be served in specific zones
- While the kernel tries to avoid it, general allocations requests can be served from any zone if needed

Pages and zones

Zones (3)

- Each zone is represented by a `struct zone` object
  - Defined in `include/linux/mmzone.h`
  - Simplified version with notable fields:

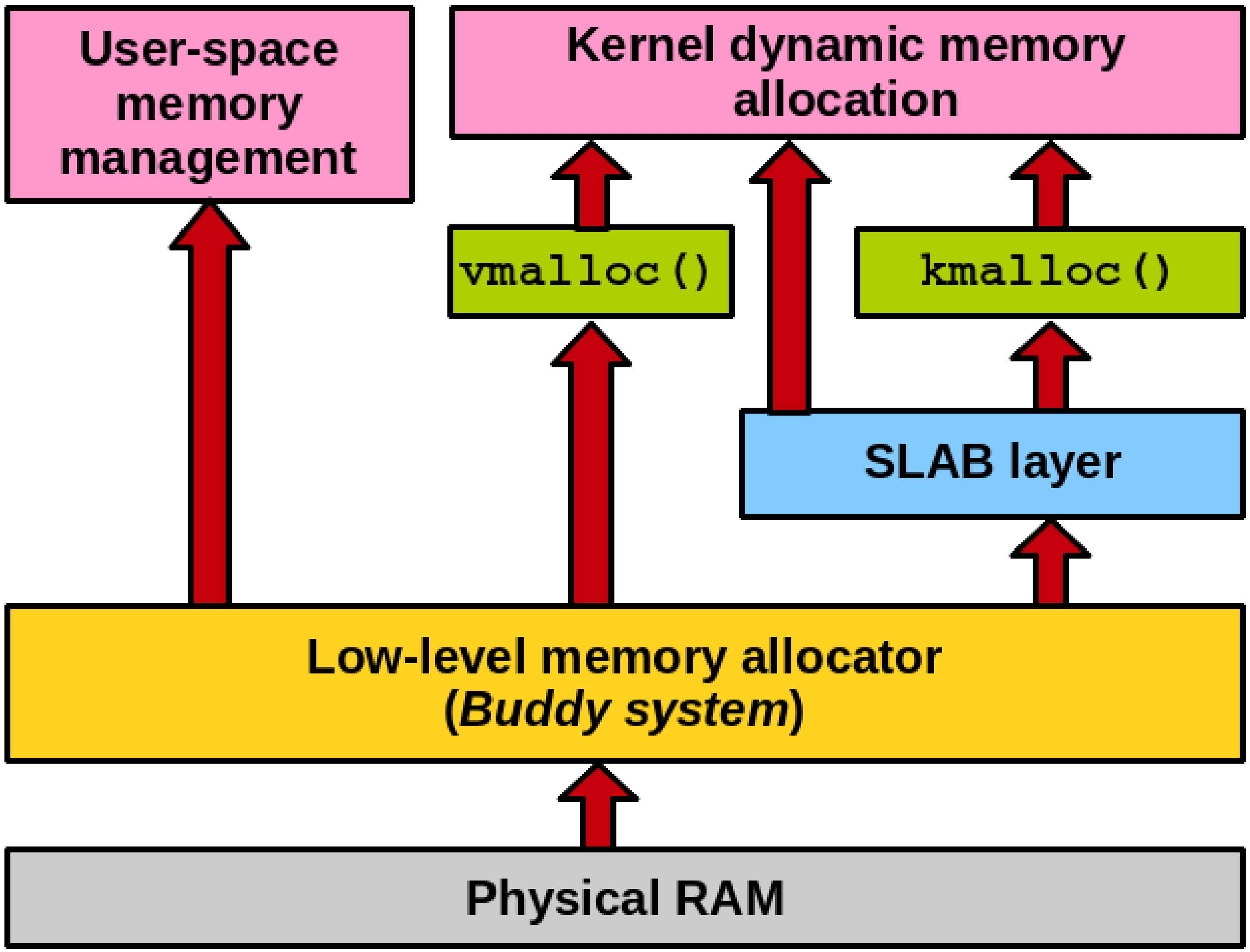
```
struct zone {
  unsigned long       watermark[NR_WMARK];
  const char          *name;
  spinlock_t          lock;
  struct free_area    free_area[MAX_ORDER];
  /* ... */
}
```

- `watermark` minimum, low and high watermarks used for per-area memory allocation
- `lock`: protects against concurrent access
- `free_area`: list of free pages to serve memory allocation requests

Outline

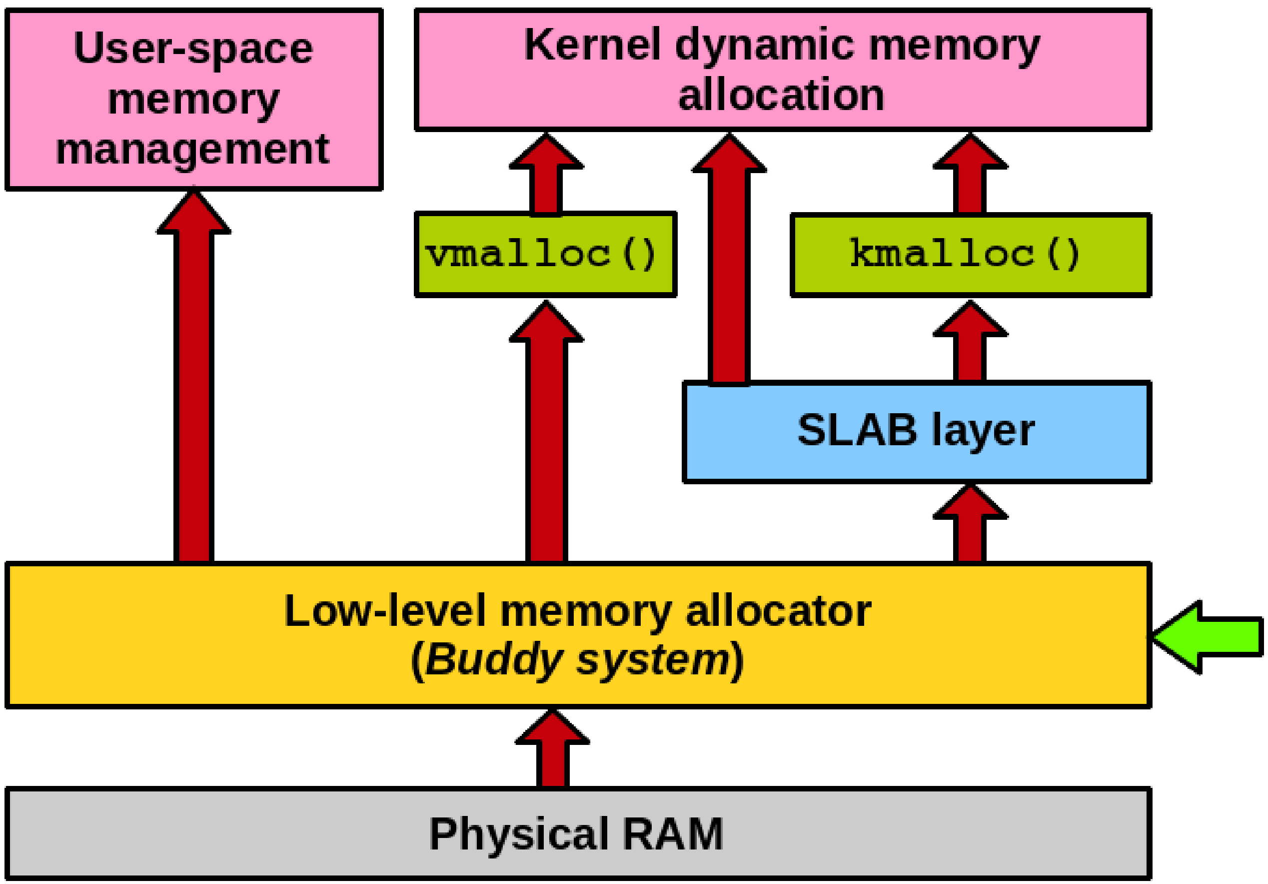
Low-level memory allocator

Memory allocation: general overview



Low-level memory allocator

Memory allocation: general overview



Low-level memory allocator

Getting pages

- Low-level mechanisms allocating memory with page-sized granularity
- Interface in `include/linux/gfp.h` #gfp: Get Free Page
  - Core function:

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order);
```

    - Allocates 2^{order} contiguous pages (1 << *order*)
    - Returns the **address of the first allocated `struct page`**
    - To actually use the allocated memory, need to convert to virtual address:

```
void * page_address(struct page *page);
```

  - To allocate and get the virtual address directly:

```
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);
```

Low-level memory allocator

Getting pages (2)

- To get a single page:

```
struct page * alloc_page(gfp_t gfp_mask);
unsigned long __get_free_page(gfp_t gfp_mask);
```

- To get a page filled with zeros:

```
unsigned long get_zeroed_page(gfp_t gfp_mask);
```

  - Needed for pages given to user space
    - A page containing user space data (process A) that is freed can be later given to another process (process B)
    - We don't want process B to read information from process A

Low-level memory allocator

Freeing pages

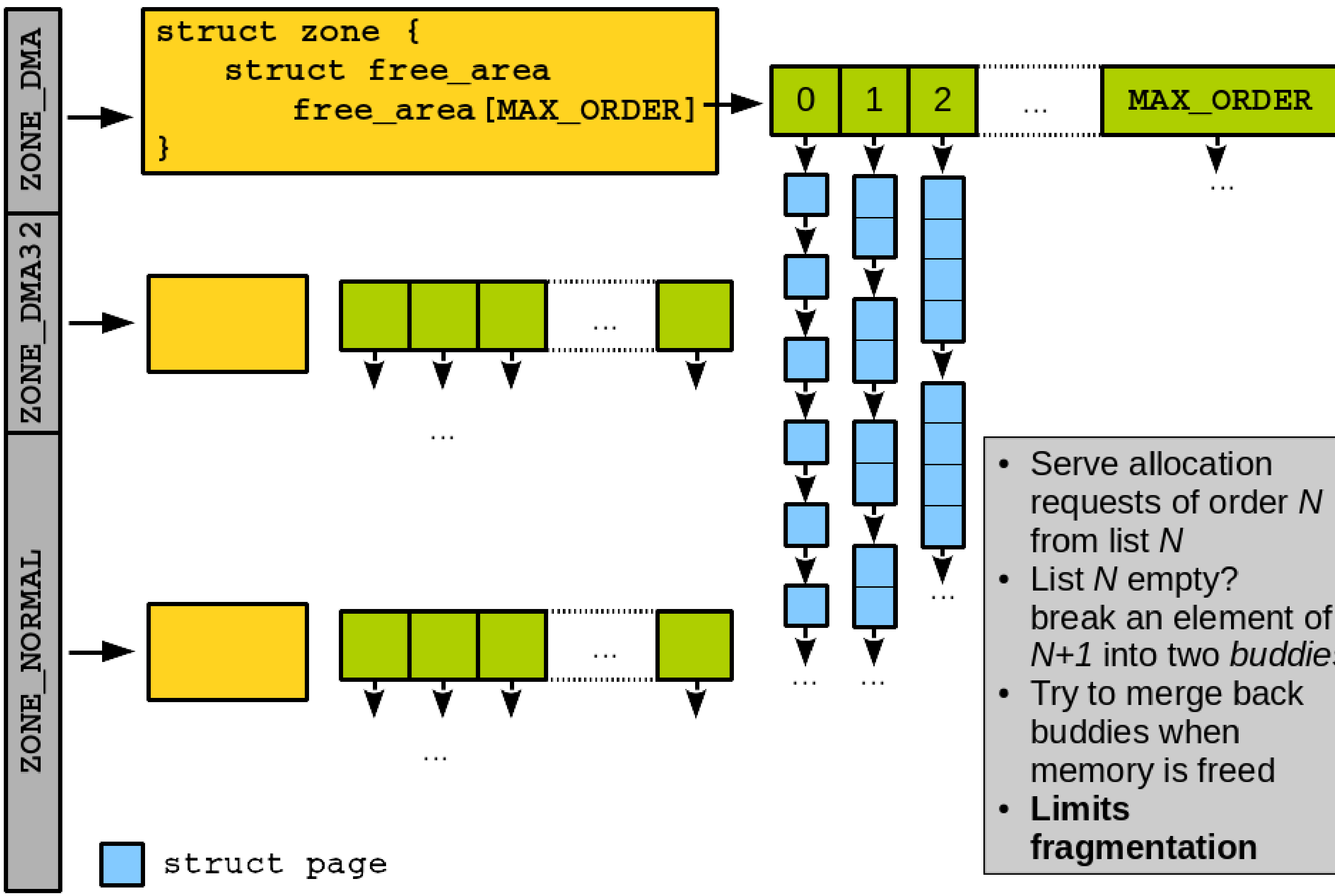
- Freeing pages - low level API:

```
void __free_pages(struct page *page, unsigned int order);
void free_pages(unsigned long addr, unsigned int order);
void __free_page(struct page *page);
void free_page(unsigned long addr);
```

  - Free only the pages you allocate!
    - Otherwise: corruption

Low-level memory allocator

Buddy system



- 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**

Low-level memory allocator

Usage 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;
  }

  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);
```

Outline

1 Pages and zones

2 Low-level memory allocator

3 kmalloc() and vmalloc()

4 Slab layer

5 Stack, high memory and per-cpu allocation

kmalloc() and vmalloc()

kmalloc() usage and kfree()

- kmalloc() allocates byte-sized chunks of memory
- Allocated memory is **physically contiguous**
- Usage similar to userspace malloc()
  - Returns a pointer to the first allocated byte on success
  - Returns NULL on error
- Declared in includes/linux/slab.h:

```
void * kmalloc(size_t size, gfp_t flags);
```

- Usage example:

```
struct my_struct *ptr;

ptr = kmalloc(sizeof(struct my_struct), GFP_KERNEL);
if(!ptr) {
   /* handle error */
}
```

kmalloc() and vmalloc()

Kmalloc flags

- gfp_t type (unsigned int in include/linux/types.h)
  - *get free page* [2]
- Specify options for the allocated memory
  1. **Action modifiers**
     - How should the memory be allocated? (for example, can the kernel sleep during allocation?)
  2. **Zone modifiers**
     - From which zone should the allocated memory come
  3. **Type flags**
     - Combination of action and zone modifiers
     - Generally preferred compared to direct use of action/zone

kmalloc() and vmalloc()

Kmalloc flags: action modifiers

- **Action modifiers:**

Flag	**Description**
_GFP_WAIT	Allocator can 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	Allocator repeats the allocation if it fails, the allocation can potentially fail
_GFP_NOFAIL	Allocator indefinitely repeats the allocation, which cannot fail

kmalloc() and vmalloc()

Kmalloc flags: action modifiers (2)

Flag	Description
_GFP_NORETRY	Allocator does not retry if the allocation fails
_GFP_NOMEMALLOC	Allocation does not fall back on reserves
_GFP_HARDWALL	Allocator enforces ”hardwall” cpuset boundaries
_GFP_RECLAIMABLE	Allocator marks the pages reclaimable
_GFP_COMP	Allocator adds compound page metadata (used by hugetlb code)

- Several action modifiers can be specified together:

```
ptr = kmalloc(size, __ GFP_WAIT | __ GFP_FS | _ GFP_IO);
```

  - Generally, **type modifiers are used instead**

kmalloc() and vmalloc()

Kmalloc flags: zone modifiers

- **Zone modifiers:**

Flag	Description
__GFP_DMA	Allocates **only** from ZONE_DMA
__GFP_DMA32	Allocates **only** from ZONE_DMA32
__GFP_HIGHMEM	Allocated from ZONE_HIGHMEM or ZONE_NORMAL

- __GFP_HIGHMEM: OK to use high memory, normal works too
- No flag specified?
  - Kernel allocates from ZONE_NORMAL or ZONE_DMA with a strong preference for ZONE_NORMAL
- Cannot specify __GFP_HIGHMEM to kmalloc() or __get_free_pages()
  - These function return a virtual address
  - Addresses in ZONE_HIGHMEM might not be mapped yet

kmalloc() and vmalloc()

Kmalloc flags: type flags

- **GFP_ATOMIC**: high priority allocation that cannot sleep
  - Use in interrupt context, while holding locks, etc.
  - Modifier flags: (_GFP_HIGH)
- **GFP_NOWAIT**: same as GFP_ATOMIC but does not fall back on emergency memory pools
  - More likely to fail
  - Modifier flags: (0)
- **GFP_NOIO**: can block but does not initiate disk IO
  - Used in block layer code to avoid recursion
  - Modifier flags: (_GFP_WAIT)
- **GFP_NOFS**: can block and perform disk IO, but does not initiate filesystem operations
  - Used in filesystem code
  - Modifier flags: (_GFP WAIT | _GFP_IO)

kmalloc() and vmalloc()

Kmalloc flags: type flags (2)

- **GFP_KERNEL**: default choice, can sleep and perform IO
  - Modifier flags: (_GFP_WAIT | _GFP_IO | _GFP_FS)
- **GFP_USER**: normal allocation, might block, for user-space memory
  - Modifier flags: (_GFP_WAIT | _GFP_IO | _GFP_FS)
- **GFP_HIGHUSER**: allocation for user-space memory, from ZONE_HIGHMEM
  - Modifier flags: (_GFP_WAIT | _GFP_IO | _GFP_FS | _GFP_HIGHMEM)
- **GFP_DMA**: served from ZONE_DMA
  - Modifier flags: (_GFP_DMA)

kmalloc() and vmalloc()

Kmalloc flags: which flag to use?

Context	Solution
Process context, can sleep	GFP_KERNEL
Process context, cannot sleep	GFP_ATOMIC or allocate at a different time
Interrupt handler	GFP_ATOMIC
Softirq	GFP_ATOMIC
Tasklet	GFP_ATOMIC
Need to handle DMA, can sleep	(GFP_DMA | GFP_KERNEL)
Need DMA, cannot sleep	(GFP_DMA | GFP_ATOMIC)

- Other types and modifier are declared and documented in include/linux/gfp.h

kmalloc() and vmalloc()

kfree

- Memory allocated with kmalloc() needs to be freed with kfree()
- include/linux/slab.h:

```
void kfree(const void *ptr);
```

- Example:

```
struct my_struct ptr;

ptr = kmalloc(sizeof(struct my_struct));
if(!ptr) {
  /* handle error */
}

/* work with ptr */

kfree(ptr);
```

- **Do not free memory that has already been freed**
  - Leads to corruption

kmalloc() and vmalloc()

vmalloc()

- **vmalloc() allocates *virtually* contiguous pages that are not guarantee to map to *physically* contiguous ones**
  - Page table is modified → no problems from the programmer usability standpoint
- Buffers related to communication with the hardware generally need to be physically contiguous
  - But most of the rest do not, for example user-space buffers
- 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

kmalloc() and vmalloc()

vmalloc() (2)

- **vmalloc() usage:**
- Similar to user-space malloc()
  - include/linux/vmalloc.h:

```
void *vmalloc(unsigned long size);
void vfree(const void *addr);
```

- Example:

```
struct my_struct *ptr;

ptr = vmalloc(sizeof(struct my_struct));
if(!ptr) {
  /* handle error */
}

/* work with ptr */

vfree(ptr);
```

kmalloc() and vmalloc()

vmalloc(): kmalloc() allocated size limitation

```
#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");
```

kmalloc() and vmalloc()

vmalloc(): kmalloc() allocated size limitation (2)

```
pierre@bulbi: ~/Desktop/VM
root@debian:~# insmod kmalloc_test.ko
[   12.949562] kmalloc_test: loading out-of-tree module taints kernel.
[   12.950338] [KMALLOC_TEST]: Entering module.
[   12.950746] ------------[ cut here ]------------
[   12.951171] WARNING: CPU: 1 PID: 2071 at mm/page_alloc.c:3541 __alloc_pages_s
lowpath+0x9de/0xb10
[   12.951894] Modules linked in: kmalloc_test(O+)
[   12.952320] CPU: 1 PID: 2071 Comm: insmod Tainted: G           O    4.10.4 #5
[   12.952908] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Ubunt
u-1.8.2-1ubuntu2 04/01/2014
[   12.953315] Call Trace:
[   12.953315]  dump_stack+0x4d/0x66
[   12.953315]  __warn+0xc6/0xe0
[   12.953315]  warn_slowpath_null+0x18/0x20
[   12.953315]  __alloc_pages_slowpath+0x9de/0xb10
[   12.953315]  ? get_page_from_freelist+0x514/0xa80
[   12.953315]  ? serial8250_console_putchar+0x22/0x30
[   12.953315]  ? wait_for_xmitr+0x90/0x90
[   12.953315]  __alloc_pages_nodemask+0x183/0x1f0
[   12.953315]  alloc_pages_current+0x9e/0x150
[   12.953315]  kmalloc_order_trace+0x29/0xe0
[   12.953315]  __kmalloc+0x18c/0x1a0
[   12.953315]  ? __free_pages+0x13/0x20
[   12.953315]  my_mod_init+0x23/0x49 [kmalloc_test]
[   12.959278]  ? 0xffffffffa0002000
[   12.959278]  do_one_initcall+0x3e/0x160
[   12.959278]  ? kmem_cache_alloc_trace+0x33/0x150
[   12.959278]  do_init_module+0x5a/0x1c9
[   12.959278]  load_module+0x1dd4/0x23f0
[   12.959278]  ? __symbol_put+0x40/0x40
[   12.959278]  ? kernel_read_file+0x19e/0x1c0
[   12.959278]  ? kernel_read_file_from_fd+0x44/0x70
[   12.959278]  SYSC_finit_module+0xba/0xc0
[   12.959278]  SyS_finit_module+0x9/0x10
[   12.959278]  entry_SYSCALL_64_fastpath+0x13/0x94
[   12.959278] RIP: 0033:0x7f7ef56495b9
[   12.959278] RSP: 002b:00007fff22a92f78 EFLAGS: 00000206 ORIG_RAX: 00000000000
00139
[   12.959278] RAX: ffffffffffffffda RBX: 00007f7ef590a620 RCX: 00007f7ef56495b9
[   12.959278] RDX: 0000000000000000 RSI: 000055a2bd49b3d9 RDI: 0000000000000003
[   12.959278] RBP: 0000000000001021 R08: 0000000000000000 R09: 00007f7ef590cf20
[   12.959278] R10: 0000000000000003 R11: 0000000000000206 R12: 000055a2bf0091c0
[   12.959278] R13: 000055a2bf0091b0 R14: 0000000000001018 R15: 00007f7ef590a678
[   12.971803] ---[ end trace 3bed3649938d2598 ]---
[   12.972456] [KMALLOC_TEST]: could not allocate 8388608 bytes
root@debian:~#
```

- Max size: 8MB

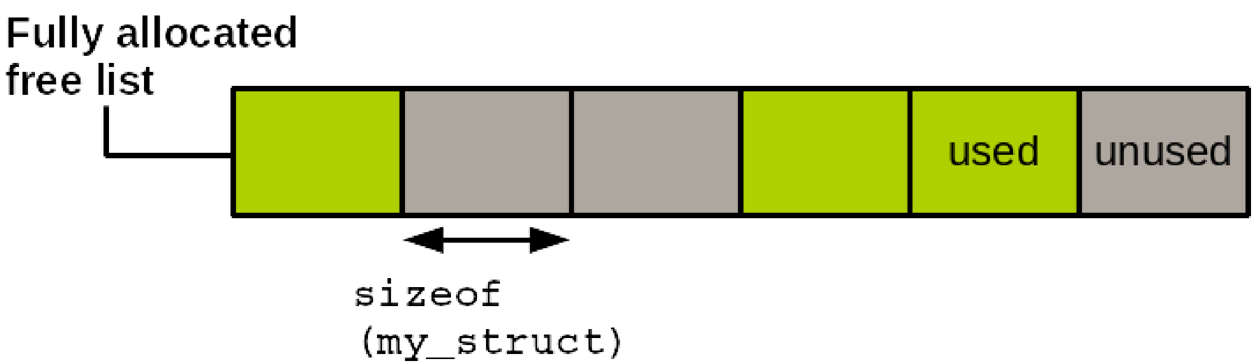
Outline

1. Pages and zones
2. Low-level memory allocator
3. kmalloc() and vmalloc()
4. Slab layer
5. Stack, high memory and per-cpu allocation

Slab layer

Presentation

- Allocating/freeing data structures is done very often in the kernel
- **Free lists:**
  - Block of pre-allocated memory for a given type of data structure



  - **Allocation cache**
    - “Allocating” from the free list is just picking a free slot
    - “Freeing” from the free list is setting back the slot state to free
  - Faster than frequent {k|v}malloc() and {k|v}free()
- Issue with ad-hoc free lists: no global control

Slab layer

Presentation (2)

- **Slab layer/slab allocator**: Linux's generic allocation caching interface
- Basic principles:
  - Frequently used data structures are allocated and freed often → needs to be cached
  - Frequent allocation/free generates memory fragmentation
    - Caching allocation/free operations in large chunks of contiguous memory reduces fragmentation
  - An allocator aware of data structure size, page size, and total cache size is more efficient
  - Part of the cache can be made per-cpu to operate without a lock
  - Allocator should be NUMA-aware and implement cache-coloring

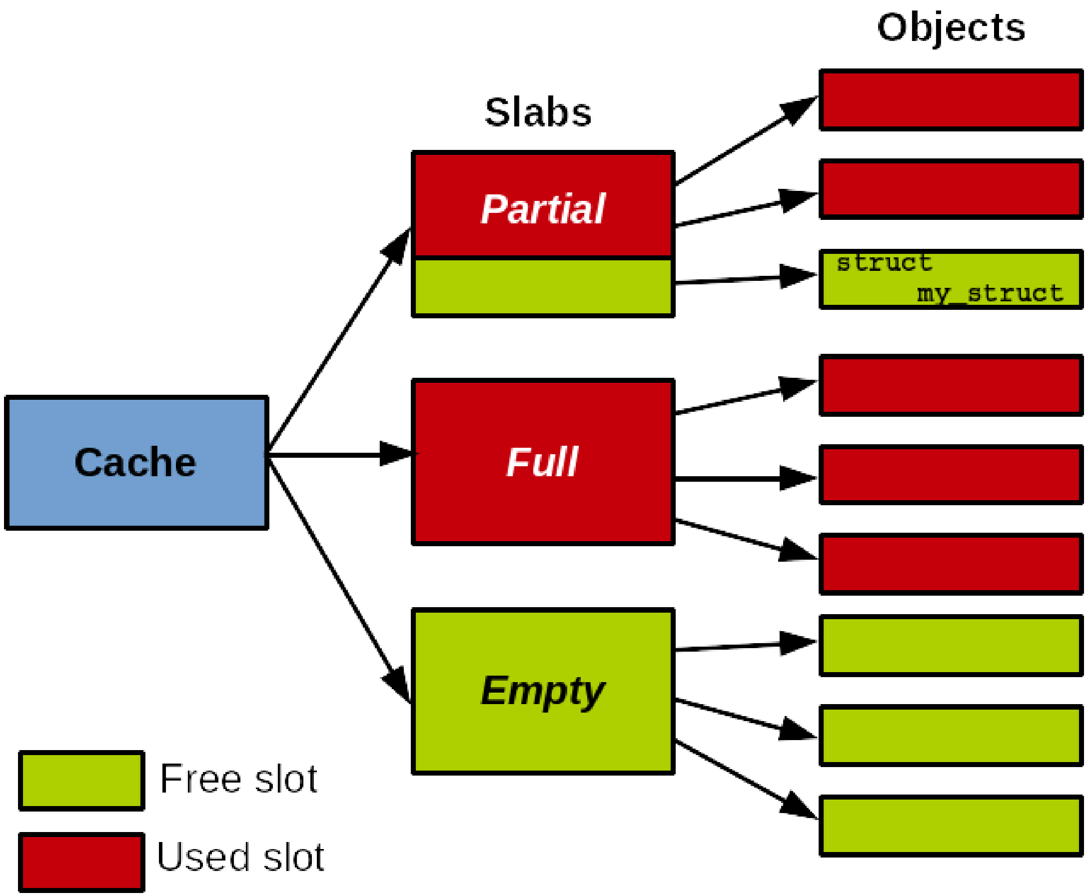
Slab layer

Presentation (3)

- The slab layer introduces defines the concept of **caches**
  - One cache per type of data structure supported
    - Example: one cache for `struct task_struct`, one cache for `struct inode`, etc.
- Each cache contains one or several **slabs**
  - One or several physically contiguous pages
- Slabs contain **objects**
  - The actual data structure slots
- A slab can be in one of three states:
  - Full: all slots used
  - Partial: some, not all slots used
  - Empty: all slots free
- Allocation requests are served from partial slabs if present, or empty slabs → **reduces fragmentation**
  - A new empty slab is actually allocated in case the cache is full

Slab layer

Presentation (4)



Slab layer

Usage

- A new cache is created using:

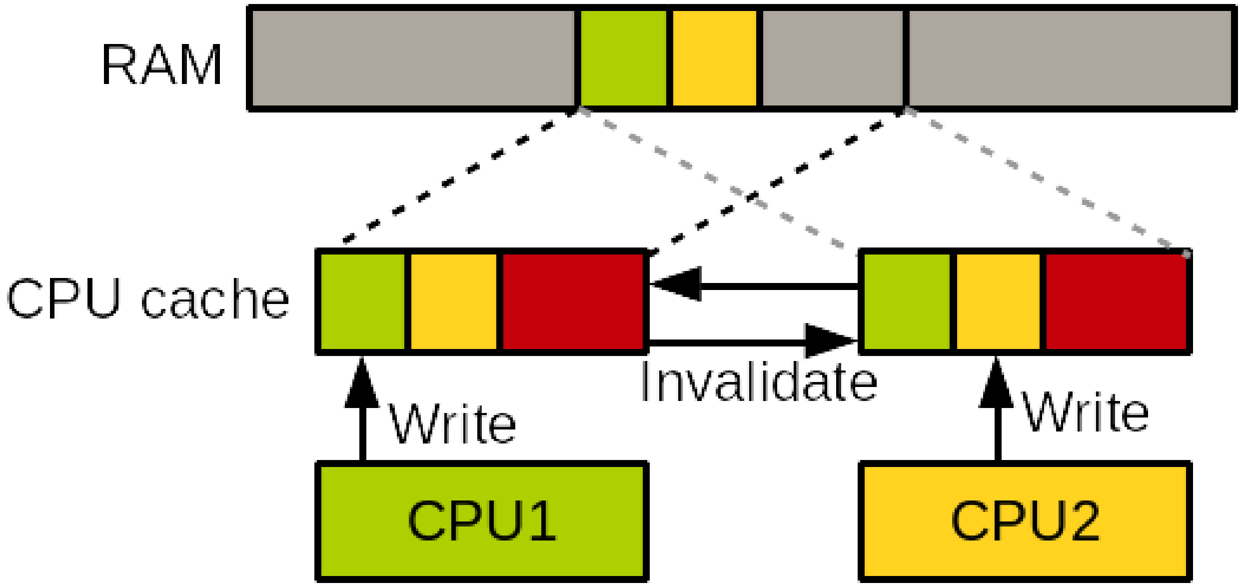
```
struct kmem_cache *kmem_cache_create(const char *name,
                                     size_t size,
                                     size_t align,
                                     unsigned long flags,
                                     void (*ctor)(void *));
```

- `name`: cache name
- `size`: data structure size
- `align`: offset of the first element within the page (can put 0)
- `ctor`: constructor called when a new data structure is allocated
  - Rarely used, can put NULL
- `flags`: settings controlling the cache behavior

Slab layer

Usage: flags

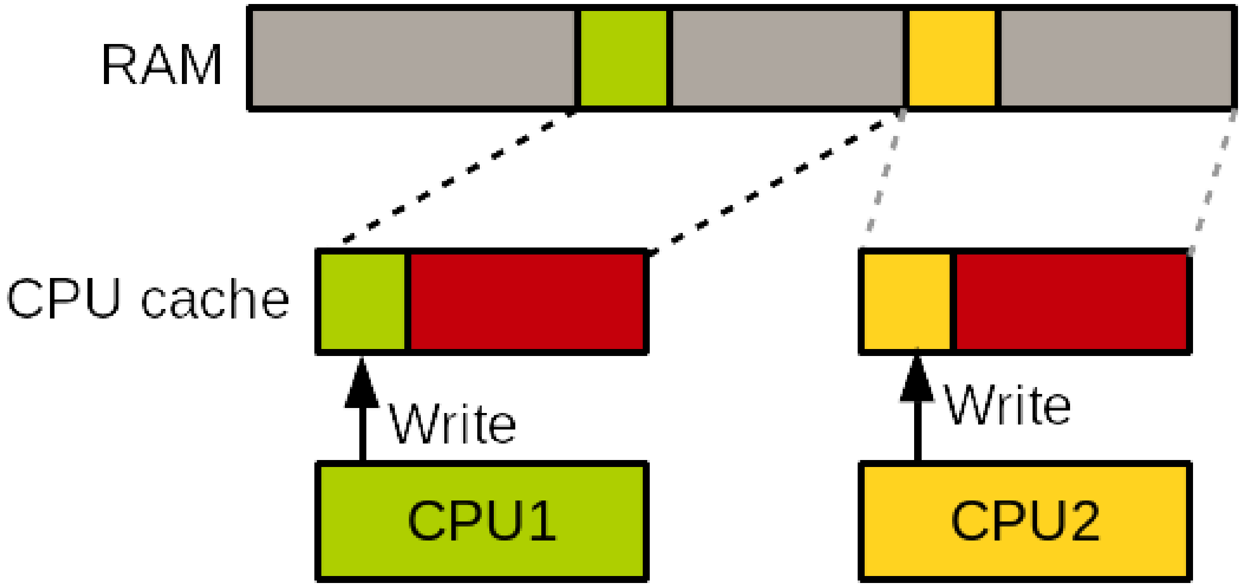
- SLAB_HWCACHEALIGN
  - Each object in a slab is aligned to a cache line
    - Prevent *false sharing*:



Slab layer

Usage: flags

- SLAB_HW_CACHEALIGN
  - Each object in a slab is aligned to a cache line
    - Prevent *false sharing*:



- Increased memory footprint

Slab layer

Usage: flags (2)

- SLAB_POISON
  - Fill the slab with know values (a5a5a5a5) to detect accesses to uninitialized memory
- SLAB_RED_ZONE
  - Extra padding around objects to detect buffer overflows
- SLAB_PANIC
  - Slab layer panics if the allocation fails
- SLAB_CACHE_DMA
  - Allocation made from DMA-enabled memory

Slab layer

Usage: allocating from the cache, cache destruction

- **Allocating from the cache:**

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

- **To free an object:**

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

- **To destroy a cache:**

```
int kmem_cache_destroy(struct kmem_cache *cachep);
```

- kmem_cache_destroy() should only be called when:
  - All slabs in the cache are empty
  - Nobody is accessing the cache during the execution of kmem_cache_destroy()
    - Implement synchronization

Slab layer

Usage 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("pierre-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;
```

Slab layer

Usage example (2)

```
  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 layer

Slab allocator variants and additional info

- **Slab allocator variants**:
  - **SLOB** (Simple List Of Blocks):
    - Used in early Linux version (from 1991)
    - Low memory footprint → used in embedded systems
  - **SLAB**:
    - Integrated in 1999
    - Cache-friendly
  - **SLUB**:
    - Integrated in 2008
    - Scalability improvements (amongst other things) over SLAB on many-cores
- More info: [5, 3, 4]

Outline

Stack, high memory and per-cpu allocation

Allocating on the stack

- Each process has:
  - A user-space stack for execution in user space
  - A kernel stack for execution in the kernel
- User-space stack is large and grows dynamically
- **Kernel stack is small and has a fixed-size**
  - Generally 8KB on 32-bit architectures and 16KB on 64-bit
- What about interrupt handlers?
  - Historically using the kernel stack of the interrupted process
  - Now using a per-cpu stack (1 single page) dedicated to interrupt handlers
  - Same thing for softirqs
- **Reduce kernel stack usage to a minimum**
  - Local variables & function parameters

Stack, high memory and per-cpu allocation

High memory allocation

- On x86_32, physical memory above 896MB is not permanently mapped within the kernel address space
  - Because of the limited size of the address space and the 1G/3G kernel/user-space physical memory split
- **Before usage, pages from highmem must be mapped after allocation**
  - Recall that allocation is done through `alloc_pages()` with the `GFP_HIGHMEM` flag
- **Permanent mapping (`include/linux/highmem.h`):**

```
void *kmap(struct page *page);
```

  - Works on low and high memory
  - Maps (update the page table) and return the given
  - **May sleep**, use only in process context
  - Number of permanent mappings is limited, unmap when done:

```
void kunmap(struct page *page);
```

Stack, high memory and per-cpu allocation

High memory allocation: usage example

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

#define PRINT_PREF      "[HIGHMEM]: "
#define INTS_IN_PAGE  (PAGE_SIZE/sizeof(
      int))

static int __init my_mod_init(void)
{
  struct page *my_page;
  void *my_ptr;
  int i, *int_array;

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

  my_page = alloc_page(GFP_HIGHUSER);

  if(!my_page)
    return -1;

  my_ptr = kmap(my_page);
  int_array = (int *)my_ptr;

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

  kunmap(my_page);
  __free_pages(my_page, 0);

  return 0;
}

static void __exit my_mod_exit(void)
{
  printk(PRINT_PREF "Exiting module.¥n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
```

Stack, high memory and per-cpu allocation

High memory allocation: temporary mappings

- **Temporary mappings**
  - Also called *atomic mappings* as they can be used from interrupt context
  - Uses a per-cpu pre-reserved slot

```
void *kmap_atomic(struct page *page);
```

  - Disables kernel preemption
  - Unmap with:

```
void kunmap_atomic(void *addr);
```

- **Do not sleep while holding a temporary mapping**
- After `kunmap_atomic()`, the next temporary mapping will overwrite the slot

Stack, high memory and per-cpu allocation

High memory allocation: temporary mappings usage example

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

#define PRINT_PREF      "[HIGHMEM_ATOMIC]: "
#define INTS_IN_PAGE  (PAGE_SIZE/sizeof(
      int))

static int __init my_mod_init(void)
{
  struct page *my_page;
  void *my_ptr;
  int i, *int_array;

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

  my_page = alloc_page(GFP_HIGHUSER);
  if(!my_page)
    return -1;

  my_ptr = kmap_atomic(my_page);
  int_array = (int *)my_ptr;
  for(i=0; i<INTS_IN_PAGE; i++) {
    int_array[i] = i;
    printk(PRINT_PREF "array[%d] = %d¥n", i,
      int_array[i]);
  }

  kunmap_atomic(my_ptr);
  __free_pages(my_page, 0);

  return 0;
}

static void __exit my_mod_exit(void)
{
  printk(PRINT_PREF "Exiting module.¥n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
```

Stack, high memory and per-cpu allocation

Per-CPU allocation

- **Per-cpu data: data that is unique to each CPU (i.e. each core)**
  - No locking required
  - Implemented through arrays in which each index corresponds to a CPU:

```c
unsigned long my_percpu[NR_CPUS]; /* NR_CPUS contains the number of cores */
```

```c
int cpu;

cpu = get_cpu();      /* get current CPU, disable kernel preemption */
my_percpu[cpu]++;     /* access the data */
put_cpu();            /* re-enable kernel preemption */
```

- Disabling kernel preemption (`get_cpu()`/`put_cpu()`) while accessing per-cpu data is necessary:
  - Preemption then reschedule on another core → `cpu` not valid anymore
  - Another task preempting the current one might access the per-cpu data → race condition

Stack, high memory and per-cpu allocation

Per-CPU allocation: the *percpu* API

- Linux provides an API to manipulate per-cpu data: *percpu*
  - In `include/linux/percpu.h`
- **Compile-time per-cpu data structure usage**:
  - Creation:

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

  - To refer to a per-cpu data structure declared elsewhere:

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

  - Data manipulation:

```
get_cpu_var(name)++;  /* increment 'name' on this CPU */
put_cpu_var(name);    /* Done, disable kernel preemption */
```

  - Access another CPU data:

```
per_cpu(name, cpu)++; /* increment name on the given CPU */
```

    - **Need to use locking!**

Stack, high memory and per-cpu allocation

Per-CPU allocation (2)

- **Per-cpu data at runtime**:
  - Allocation:

```
struct my_struct  *my_var = alloc_percpu(struct my_struct);
if(!my_var) {
  /* allocation  error */
}
```

  - Manipulation:

```
get_cpu_var(my_var)++;
put_cpu_var(my_var);
```

  - Deallocation:

```
free_percpu(my_var);
```

- Benefits of per-cpu data:
  - Removes/minimizes the need for locking
  - Reduces cache thrashing
    - Processor access local data so there is less cache coherency overhead (invalidation) in multicore systems

Stack, high memory and per-cpu allocation

Per-CPU allocation: usage example (static)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/percpu.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/smp.h>

#define PRINT_PREF "[PERCPU] "
struct task_struct *thread1, *thread2, *thread3;
DEFINE_PER_CPU(int, my_var);

static int thread_function(void *data)
{
    while(!kthread_should_stop()) {
      int cpu;
      get_cpu_var(my_var)++;
      cpu = smp_processor_id();
        printk("cpu[%d] = %d¥n", cpu, get_cpu_var(my_var));
      put_cpu_var(my_var);
      msleep(500);
   }
}
```

```
static int __init my_mod_init(void)
{
  int cpu;

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

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

  wmb();

  thread1 = kthread_run(thread_function, NULL, "percpu-thread1");
  thread2 = kthread_run(thread_function, NULL, "percpu-thread2");
  thread3 = kthread_run(thread_function, NULL, "percpu-thread3");

  return 0;
}
```

Stack, high memory and per-cpu allocation

Per-CPU allocation: usage example (static) (2)

```
static void __exit my_mod_exit(void)
{
  kthread_stop(thread1);
  kthread_stop(thread2);
  kthread_stop(thread3);
  printk(KERN_INFO "Exiting module.¥n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Stack, high memory and per-cpu allocation

Per-CPU allocation: usage example (dynamic)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/percpu.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/smp.h>

#define PRINT_PREF "[PERCPU] "
struct task_struct *thread1, *thread2, * thread3;
void *my_var2;

static int thread_function(void *data)
{
    while(!kthread_should_stop()) {
        int *local_ptr, cpu;
        local_ptr = get_cpu_ptr(my_var2);
        cpu = smp_processor_id();
        (*local_ptr)++;
        printk("cpu[%d] = %d¥n", cpu, *local_ptr);
        put_cpu_ptr(my_var2);
        msleep(500);
    }
    do_exit(0);
}
```

```
static int __init my_mod_init(void)
{
    int *local_ptr;
    int cpu;
    printk(PRINT_PREF "Entering module.¥n");

    my_var2 = alloc_percpu(int);
    if(!my_var2)
        return -1;

    for(cpu=0; cpu<NR_CPUS; cpu++) {
        local_ptr = per_cpu_ptr(my_var2, cpu);
        *local_ptr = 0;
        put_cpu();
    }

    wmb();

    thread1 = kthread_run(thread_function, NULL, "percpu-thread1");
    thread2 = kthread_run(thread_function, NULL, "percpu-thread2");
    thread3 =
    return 0;
}
```

Stack, high memory and per-cpu allocation

Per-CPU allocation: usage example (dynamic) (2)

```
static void __exit my_mod_exit(void)
{
    kthread_stop(thread1);
    kthread_stop(thread2);
    kthread_stop(thread3);

    free_percpu(my_var2);

    printk(KERN_INFO "Exiting module.¥n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Stack, high memory and per-cpu allocation

Choosing the right allocation method

- Need physically contiguous memory?
  - kmalloc() or low-level allocator, with flags:
    - GFP_KERNEL if sleeping is allowed
    - GFP_ATOMIC otherwise
- Need large amount of memory, not physically contiguous:
  - vmalloc()
- Frequently creating/destroying large amount of the same data structure:
  - Use the slab layer
- Need to allocate from high memory?
  - Use alloc_page() then kmap() or kmap_atomic()

Bibliography I

1 Cramming more into struct page.
https://lwn.net/Articles/565097/.
Accessed: 2017-03-12.

2 Gfp_kernel or slab_kernel?
https://lwn.net/Articles/23042/.
Accessed: 2017-03-18.

3 The slub allocator.
https://lwn.net/Articles/229984/.
Accessed: 2017-03-18.

4 BONWICK, J., ET AL.
The slab allocator: An object-caching kernel memory allocator.
In *USENIX summer* (1994), vol. 16, Boston, MA, USA.

5 LAMETER, C.
Slab allocators in the linux kernel: Slab, slob, slub.
LinuxCon/Dsseldorf 2014,
http://events.linuxfoundation.org/sites/events/files/slides/slaballocators.pdf.
Accessed: 2017-03-18.