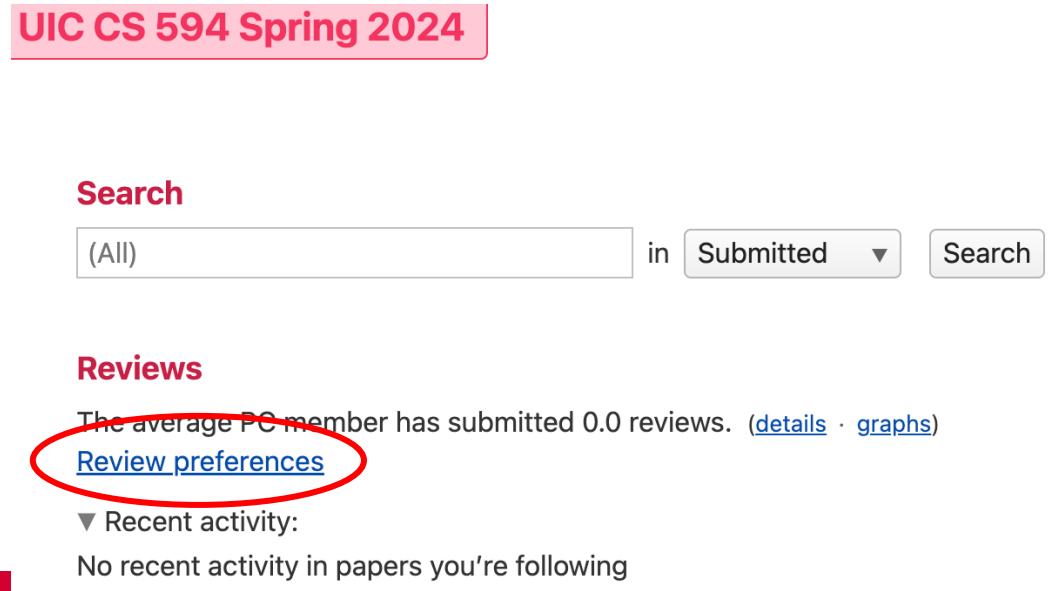
Seminar session

Start after the spring break.

- everyone lead the paper discussion for at least one paper

Update with your paper preferences: https://uic-lkp24sp.hotcrp.com/



Seminar session

Review the abstract and type a score from -20 to 20

- -20 dislike; 20 favorite
- The hotcrp system will assign you papers to review (expect to review 2~3 papers, and lead 1~2 in the discussion)

ID ▾	Title	Preference
#1	Mosaic Pages: Big TLB Reach with Small Pages	20
#2	CARAT KOP: Towards Protecting the Core HPC Kernel from Linux Kernel Modules	0
#3	TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers	0
#4	zpoline: a system call hook mechanism based on binary rewriting	0
#5	Luci: Loader-based Dynamic Software Updates for Off-the-shelf Shared Objects	0
#6	On-demand Container Loading in AWS Lambda	0

Midterm scopes

Kernel code exploration, kernel debugging, etc.

Isolation, system calls and Linux kernel data structures

Process management and process scheduling

Interrupt handling: top half, bottom half

Kernel synchronization

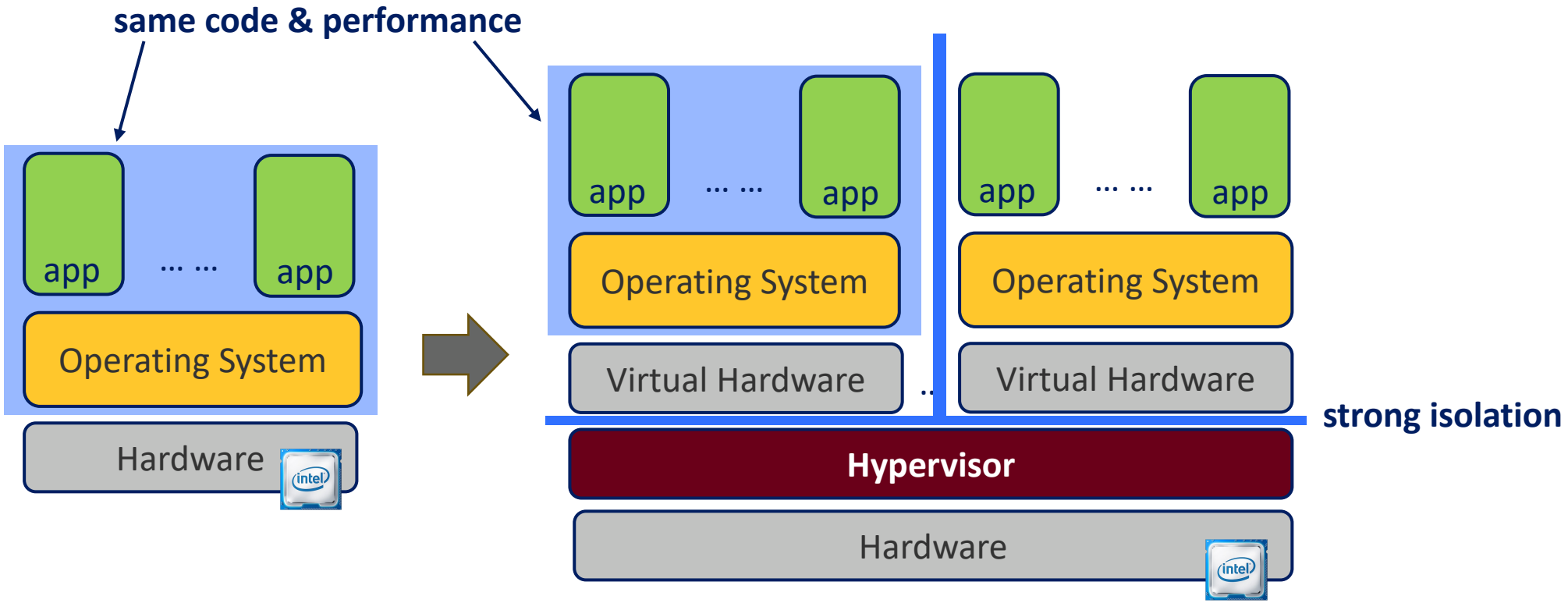
Timer and time management

Virtualization

Memory, address space

VFS, Filesystem and block I/O

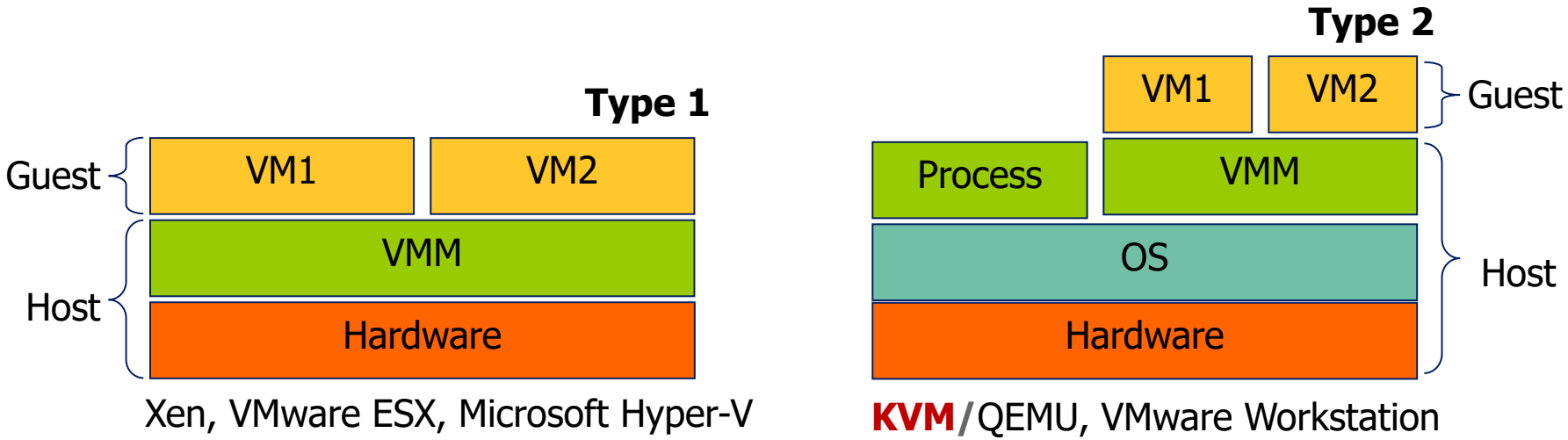
Recap: Operating System Virtualization



Recap: Type 1 v.s. Type 2 Hypervisors

Type-1: hypervisor runs directly on hardware

Type-2: hypervisor runs on a host OS



- Virtual Machine Monitor (VMM) = Hypervisor = Host OS
- Virtual Machine (VM) = Guest OS

Recap: virtualization implementations

System/Machine simulation

- QEMU w/o KVM, Gem5 (architecture simulator)

Hypervisor

- with hardware virtualization support
  - Intel VT-x, AMD SVM – CPU/Memory virtualization
  - vCPU (VMCS), Nested/Extended Page Tables (EPT/NPT)
- without hardware virtualization support (< ~2007)
  - para-virtualization (require Guest OS modification)

OS-level virtualization

- containers (e.g., docker)

Recap: virtualize CPU w/ VMX

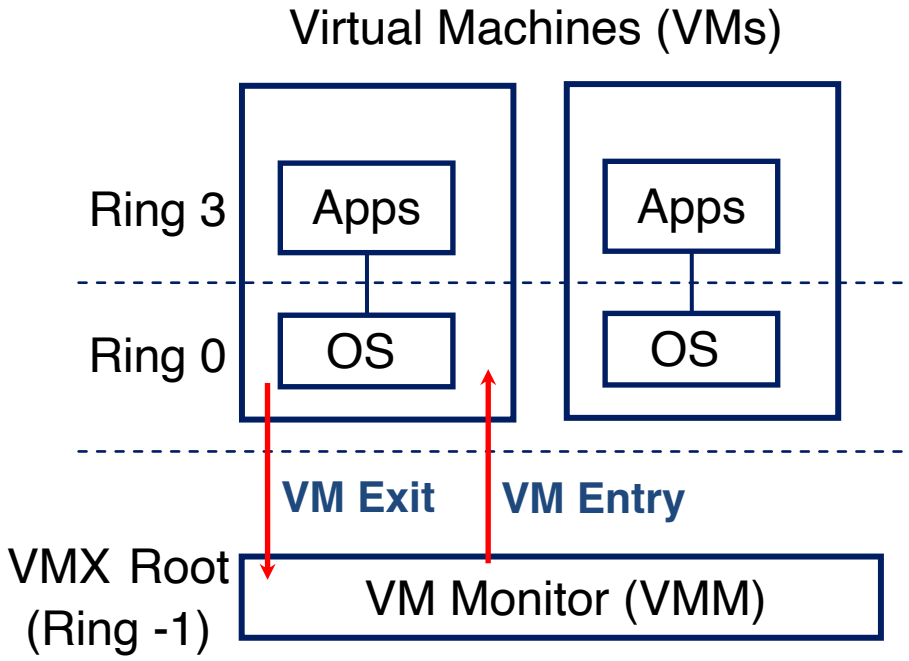
VMX (Virtual Machine Extension) supports virtualizing CPU and memory

Two new VT-x operating modes

- VMX non-root: guest OSes (less-privileged)
- VMX root: for VMM (privileged)

Two new transitions

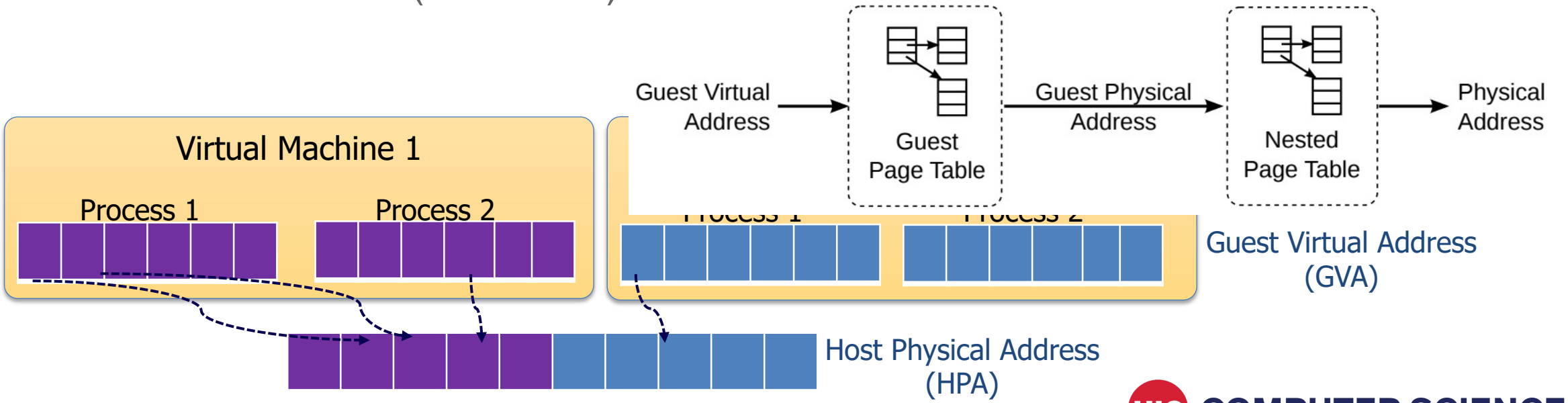
- VM entry to non-root operation
- VM exit to root operation



Recap: virtualize memory

Enforce the Guest OS address translation to reuse physical memory

- Old time (before 2007): Shadow Page Table (SPT)
- vmx/svm: Nested (Extended) Page Table



I/O devices and device drivers

I/O devices

- e.g., printers, disk drives, and network interfaces

Device drivers

- A software component that facilitates communication between the operating system and a hardware device.

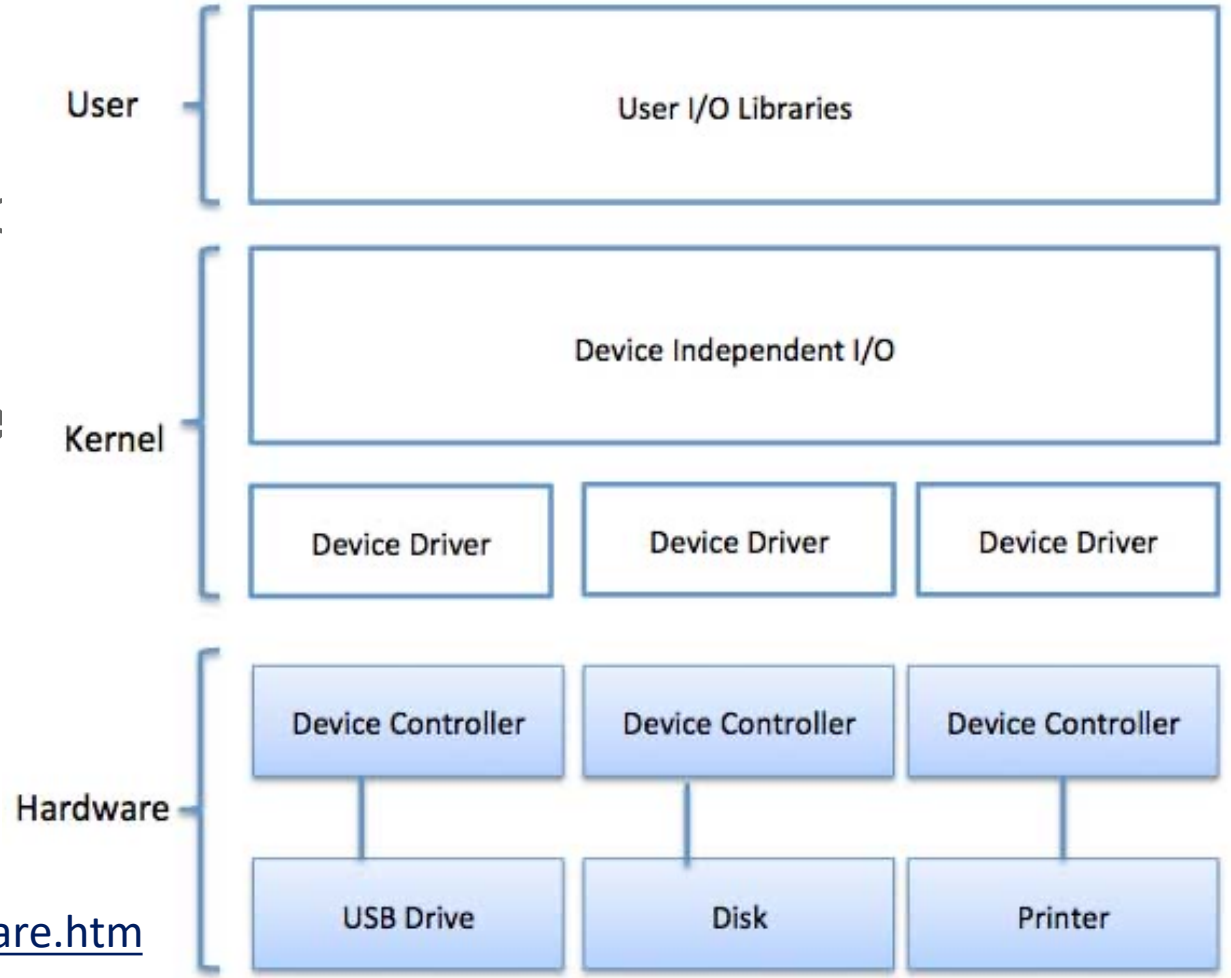
I/O devices and device drivers

I/O devices

- e.g., printers, disk drives, and netwo

Device drivers

- A software component that facilitate operating system and a hardware d



https://www.tutorialspoint.com/operating_system/os_io_software.htm

Write a Linux device driver

Similar to a kernel module: register a device (char or block)

```
static int __init char_device_init(void) {
    major_number = register_chrdev(0, DEVICE_NAME, &fops);

    if (major_number < 0) {
        printk(KERN_ALERT "Failed to register a major number\n");
        return major_number;
    }

    printk(KERN_INFO "Registered correctly with major number %d\n", major_number);
    return 0;
}
```

Q: what is a character device and what is a block device?

Write a Linux device driver

Define the device number and operations

```
static int major_number;
static char message[BUF_SIZE] = "Hello from the kernel!";
static short size_of_message;

static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

device is a file!

Write a Linux device driver

Implement the device read/write, open/close functions

```
static ssize_t device_read(struct file *file, char *buffer, size_t length, loff_t *offs
    size_t bytes_to_copy;

    if (length > size_of_message - *offset)
        bytes_to_copy = size_of_message - *offset;
    else
        bytes_to_copy = length;

    if (copy_to_user(buffer, message + *offset, bytes_to_copy) != 0) {
        return -EFAULT;
    }

    *offset += bytes_to_copy;

    return bytes_to_copy;
}
```

Write a Linux device driver

Create a device file in /dev before use

- `sudo mknod /dev/my_char_device c <major_number> 0`

You may also need to set the permission to allow user-space applications to access the device.

- `sudo chmod 666 /dev/my_char_device`

User-space applications can use standard I/O to access the device

- `echo "Hello" > /dev/my_char_device`
- `cat /dev/my_char_device`

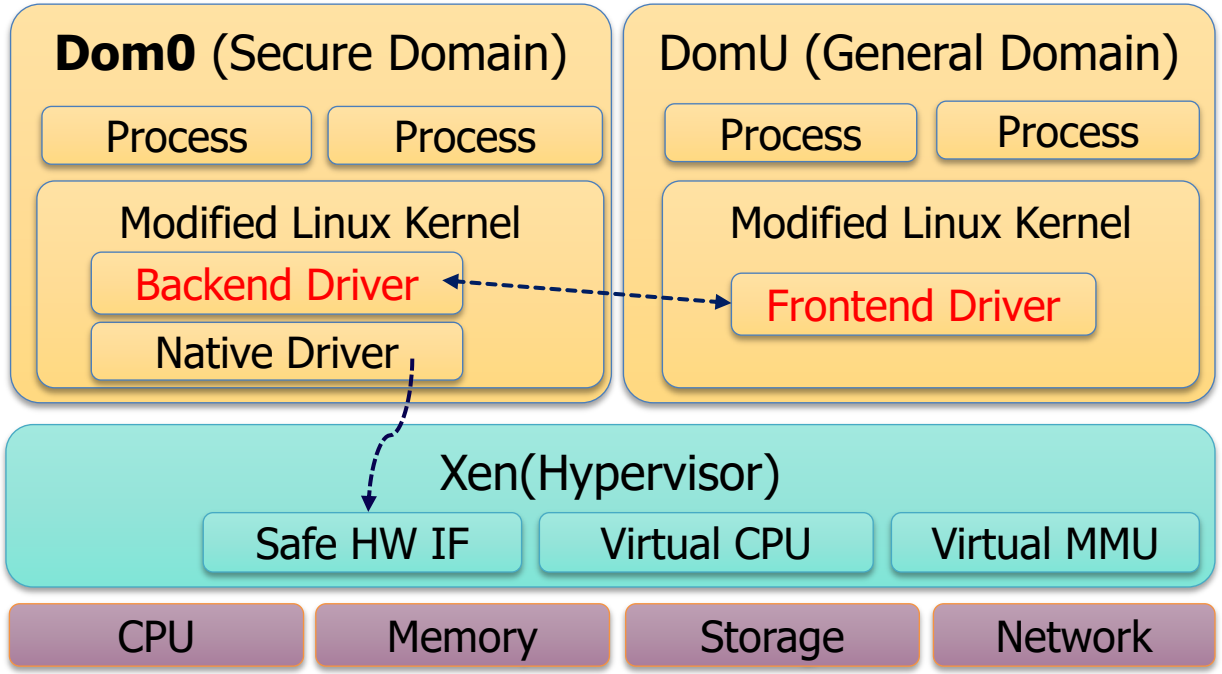
Quiz: which is (are) wrong

1. Character devices provide a stream of characters to or from a user program. They operate in a byte-stream mode, meaning data is transferred as a stream of bytes.
2. Block devices provide a fixed-size block of data, usually 512 bytes or a multiple thereof, to or from a user program.
3. Block devices are capable of random accesses
4. Random access devices like `/dev/null` or `/dev/zero` are block devices
5. USB drives are block devices
6. Serial ports and terminals are character devices

I/O Virtualization

Front-end/Back-end Driver Model

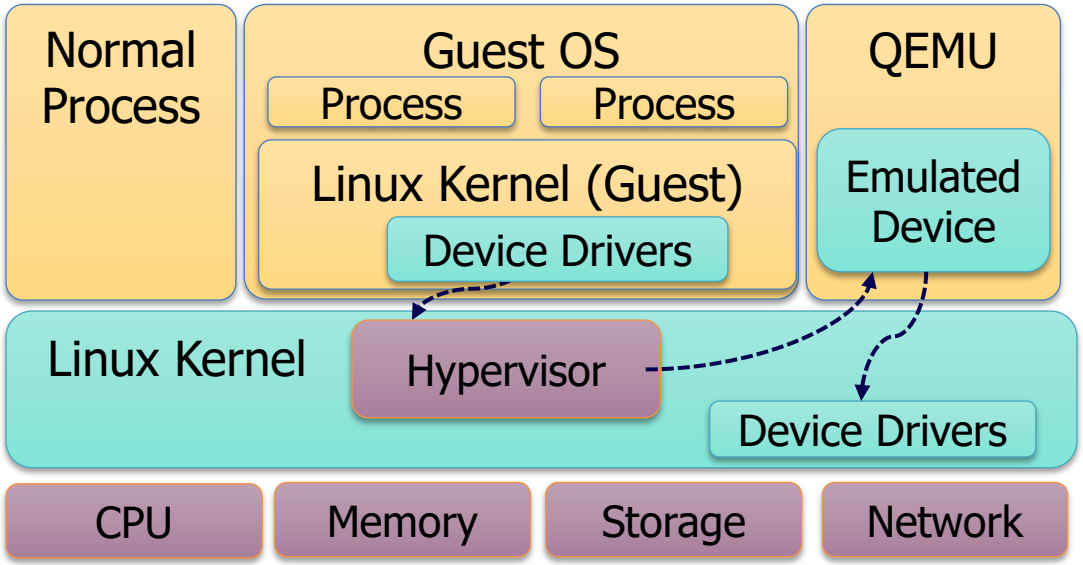
- Guest OS uses para-virtualized front-end drivers to send requests to backend drivers.
- Back-end drivers on secure domain receive the requests, performs actual IO using the native driver.



I/O Virtualization

Emulation

- Behavior of a particular device is emulated as a software module.
- Guest OS uses the **native device driver** for the device.
- VMM intercepts all the access from guest OS to the device.
- The intercepted accesses are sent to the emulated device.
- The Emulated device do the actual IO operations.



I/O Virtualization

Single-Root IO Virtualization (SR-IOV)

- Hardware support (e.g., NIC)

A SR-IOV-enabled device can present several instances of itself

- Each assigned to a different VM
- The hardware multiplexes itself

A device has at least one Physical Function controlled by the hypervisor

- The instances of itself visible to VMs are called Virtual Functions

I/O Virtualization

Single-Root IO Virtualization

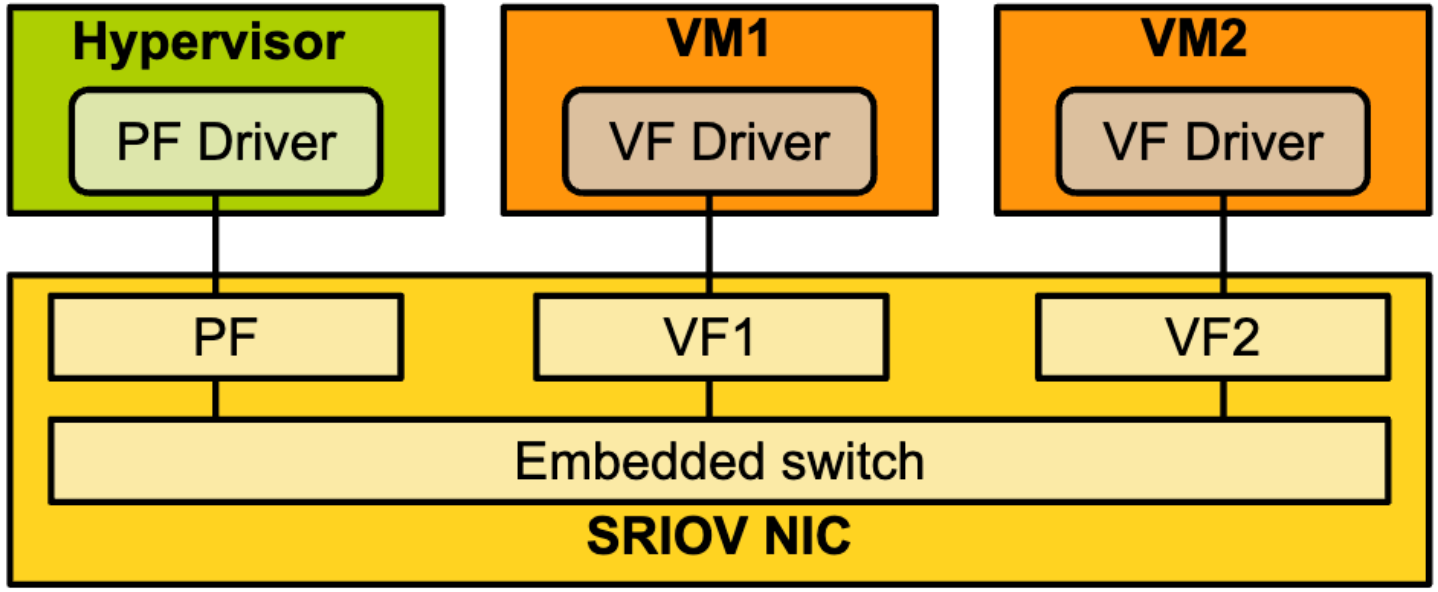
- Hardware support (e.g., N

A SR-IOV-enabled device ca

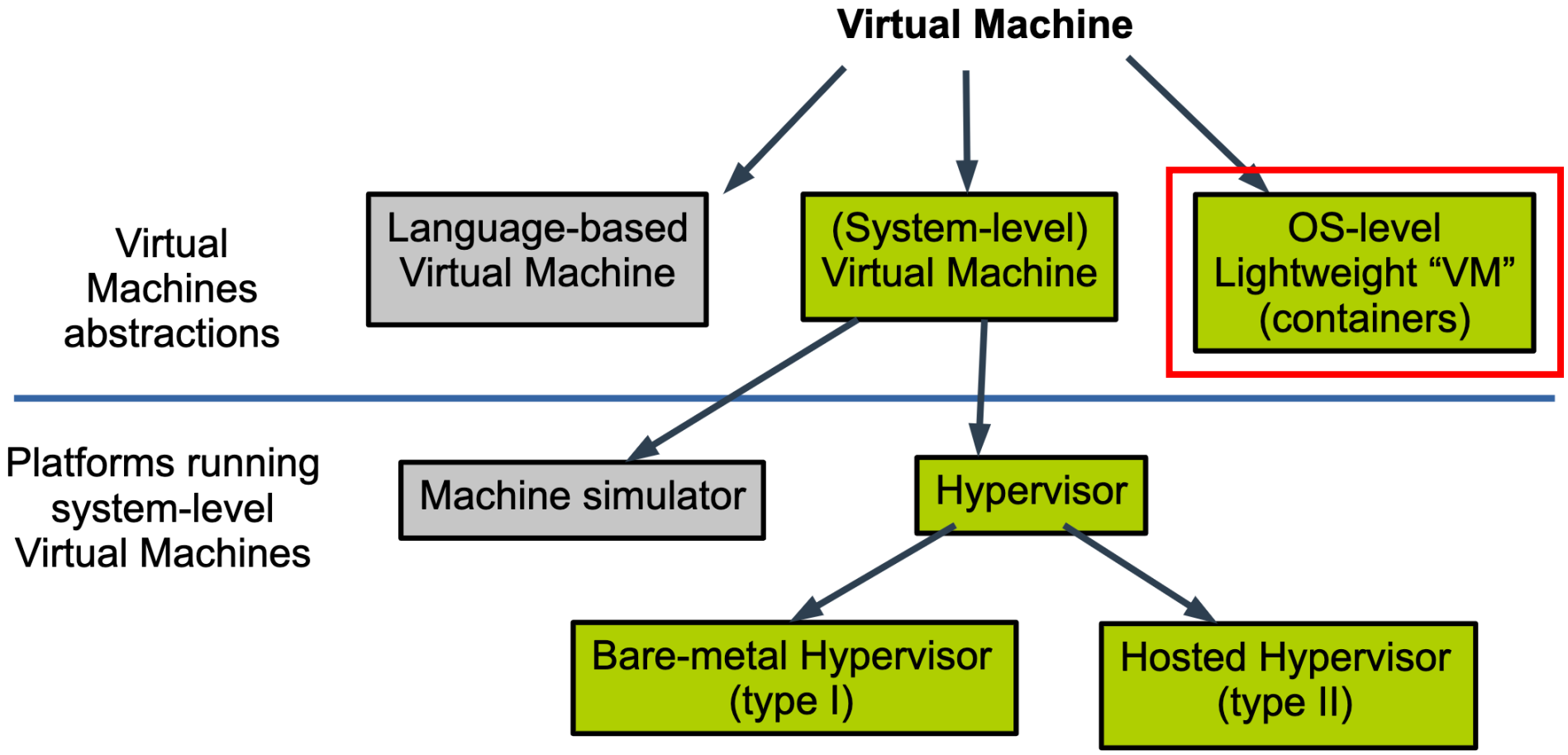
- Each assigned to a differe
- The hardware multiplexes

A device has at least one Ph

- The instances of itself visible to VMs are called Virtual Functions



High-level categories of virtual machines



Ref: Pierre Olivier's slides

OS-level Lightweight VMs

E.g., containers

- Isolation of native applications through OS mechanisms (chroot, cgroups)
  - Also packs the dependencies (and libraries) into the container
  - An illusion of running applications in an OS
  - Lightweight: fast boot than a hypervisor-based VM
- No attempt is made to virtualize the hardware
- Cannot run a different OS type (e.g., run windows on a Linux host)

Writing a container in a few lines of Go code:
https://github.com/lizrice/containers-from-scratch

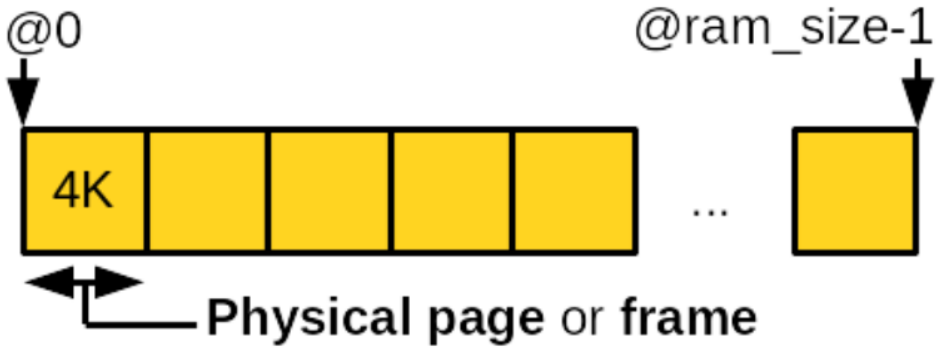
Memory Management

Xiaoguang Wang

UIC COMPUTER SCIENCE

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;          /* 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)`: 64 bytes
- Assuming 8GB of RAM and 4K-sized pages: How many bytes of memory for `struct page` objects ?

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

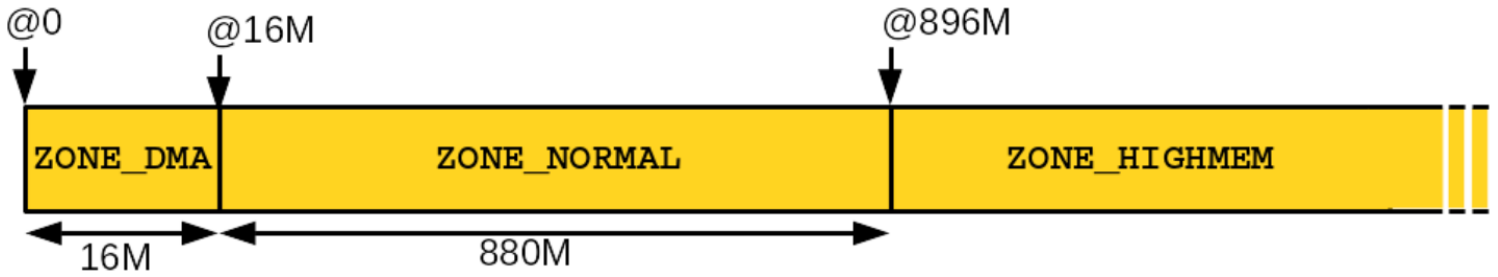
Zones

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

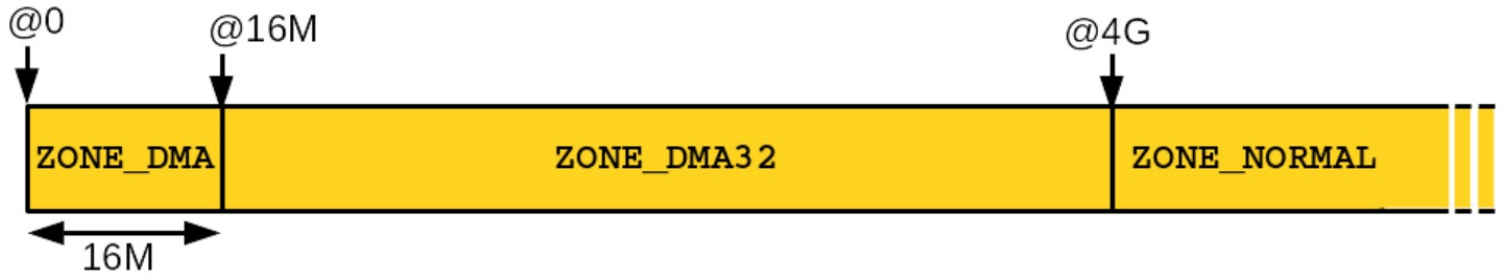
32-bit Linux only

Zones

x86_32 zones layout



x86_64 zones layout



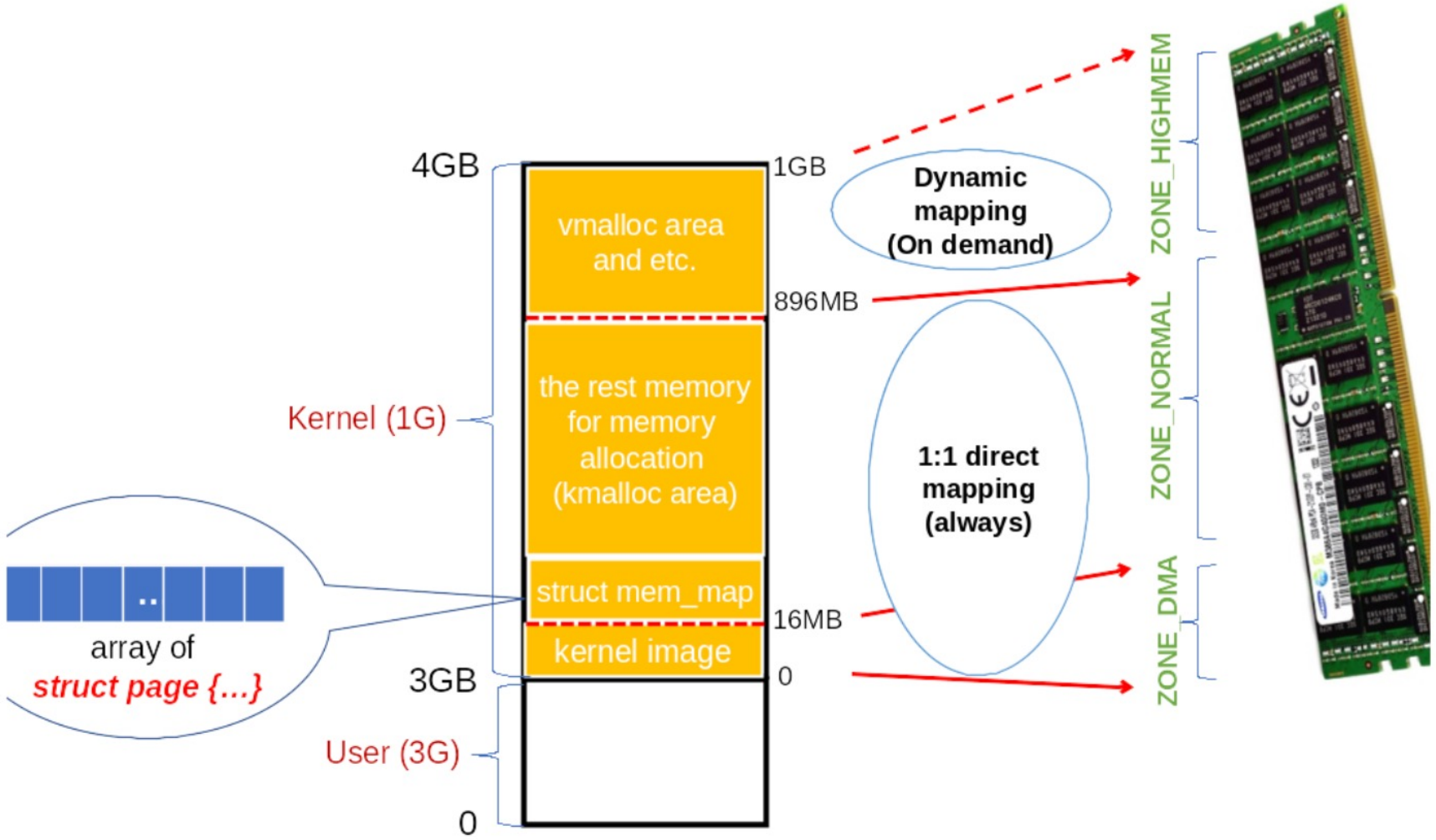
Zones

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

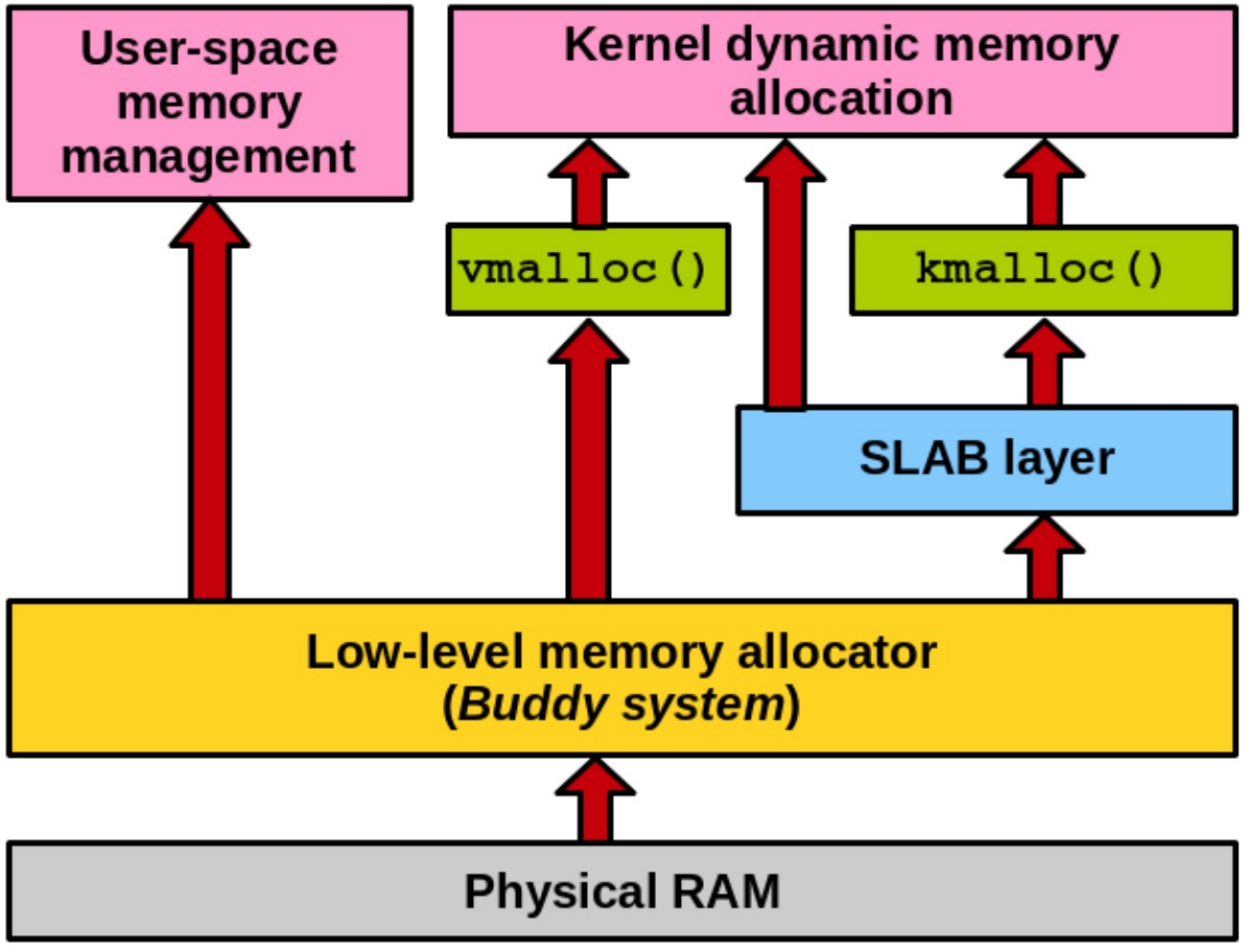
```
struct zone {
    const char *name;                   /* Name of this zone */
    unsigned long zone_start_pfn;       /* starting page frame number of the zone */
    unsigned long watermark[NR_WMARK];
                        /* minimum, low, and high watermarks
                         * for per-zone 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)



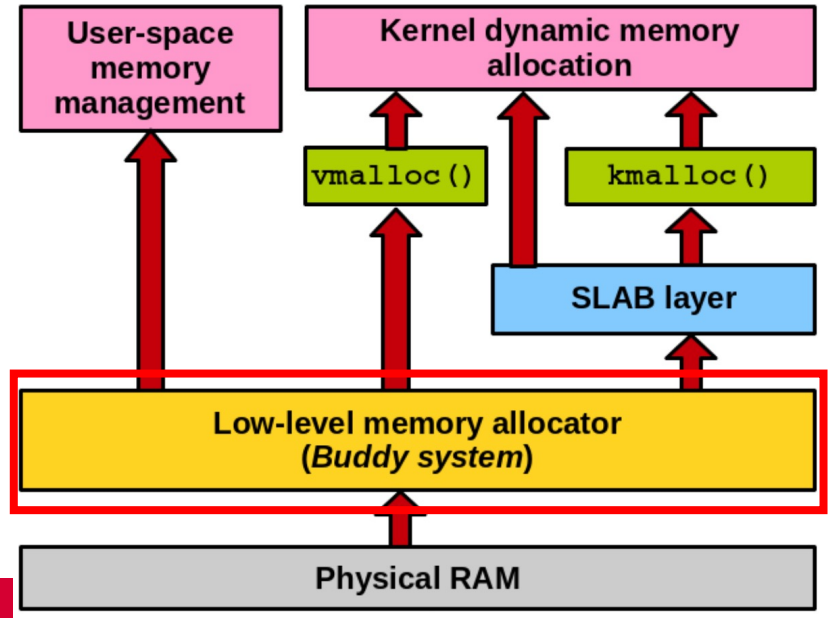
Hierarchy of memory allocators



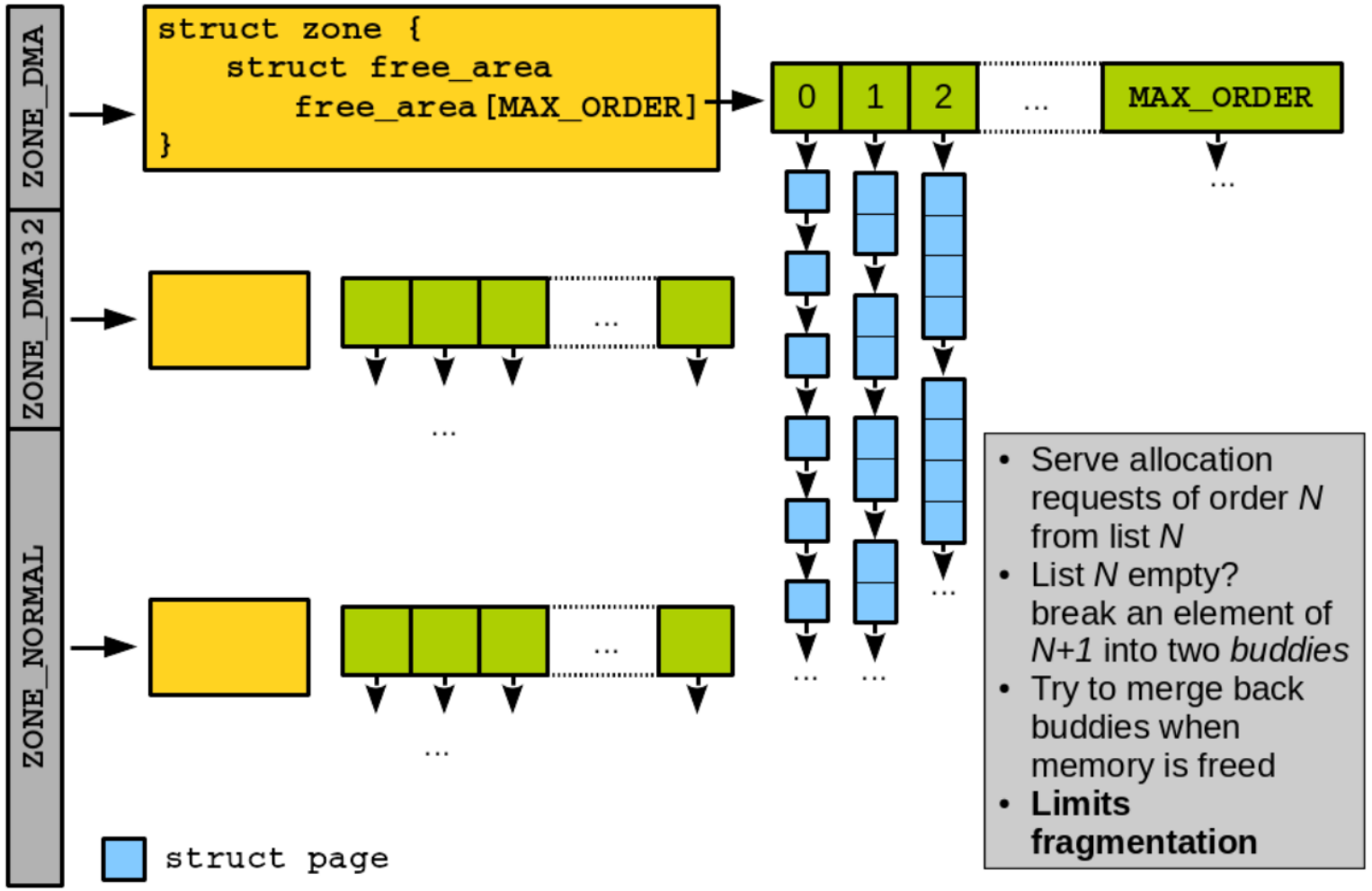
Low-level memory allocator

Buddy system

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



Buddy system



- 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**

- Prevent memory from being fragmented

Status of Buddy System

```
└─$ cat /proc/buddyinfo
Node 0, zone      DMA      2      2      1      1      3      2      1      2      1      2      0
Node 0, zone    DMA32   5571   2308   1501    443    141     65     58     47     38     18      4
```

```
└─$ cat /proc/buddyinfo
Node 0, zone      DMA      0      0      0      0      0      0      0      0      1      1      2
Node 0, zone    DMA32     10     11      6      6      8      6      8      7      6      7    541
Node 0, zone   Normal   2268  15369  28723  27177  29171  16561   9876   6337   3430     44   1458
```

Page allocation / deallocation

```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);

/**
 * 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_KERNEL`: Default. Can sleep and perform IO

- `__GFP_WAIT | __GFP_IO | __GFP_FS`

gfp_t: type flags

GFP_NOIO: Can block but does not initiate disk IO

- Used in block layer code to avoid recursion

GFP_NOFS: Can block and perform disk IO, but does not initiate filesystem operations

- Used in filesystem code

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;

    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");
}
```

High memory (x86_32)

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

`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() v.s. kmalloc()

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() v.s. kmalloc()

```c
#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");
```

vmalloc() v.s. kmalloc()

```
[Feb29 03:49] [KMALLOC_TEST]: Entering module.
[  +0.004927] ------------[ cut here ]------------
[  +0.000002] WARNING: CPU: 2 PID: 3005 at mm/page_alloc.c:5396 __alloc_pages+0x2b0/0x3
[  +0.000013] Modules linked in: kmalloc(OE+) ppdev binfmt_misc nls_iso8859_1 parport_p
[  +0.000019] CPU: 2 PID: 3005 Comm: insmod Tainted: G           OE     5.15.0-97-gener
[  +0.000002] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel-1.16.3-0-
[  +0.000002] RIP: 0010:__alloc_pages+0x2b0/0x330
[  +0.000002] Code: 48 8b 04 25 c0 fb 01 00 48 05 a0 0d 00 00 41 be 01 00 00 00 48 89 4
[  +0.000001] RSP: 0018:ffffc1dd411f3b80 EFLAGS: 00010246
```

```
[  +0.000000] Call Trace:
[  +0.000003]  <TASK>
[  +0.000008]  ? show_trace_log_lvl+0x1d6/0x2ea
[  +0.000009]  ? show_trace_log_lvl+0x1d6/0x2ea
[  +0.000002]  ? alloc_pages+0x9e/0x1e0
[  +0.000002]  ? show_regs.part.0+0x23/0x29
[  +0.000001]  ? show_regs.cold+0x8/0xd
[  +0.000002]  ? __alloc_pages+0x2b0/0x330
[  +0.000002]  ? __warn+0x8c/0x100
[  +0.000005]  ? __alloc_pages+0x2b0/0x330
[  +0.000004]  ? report_bug+0xa4/0xd0
[  +0.000006]  ? handle_bug+0x39/0x90
[  +0.000006]  ? exc_invalid_op+0x19/0x70
[  +0.000001]  ? asm_exc_invalid_op+0x1b/0x20
[  +0.000002]  ? __alloc_pages+0x2b0/0x330
[  +0.000002]  alloc_pages+0x9e/0x1e0
[  +0.000001]  kmalloc_order+0x2f/0xd0
[  +0.000003]  kmalloc_order_trace+0x1d/0x90
[  +0.000001]  __kmalloc+0x2b1/0x330
[  +0.000002]  my_mod_init+0x2a/0x1000 [kmalloc]

[  +0.000002] ---[ end trace 24e36de1ed8200e9 ]---
[  +0.000001] [KMALLOC_TEST]: could not allocate 8388608 bytes
[  +6.368453] Exiting module.
```

Outline

Zones

Buddy systems

Page allocation

`kmalloc()` / `vmalloc()`

Slab allocator (kernel object cache)

Per-CPU data

Slab allocator

Allocating/freeing data structures frequently happens in the kernel

Q: how to make memory allocation faster?

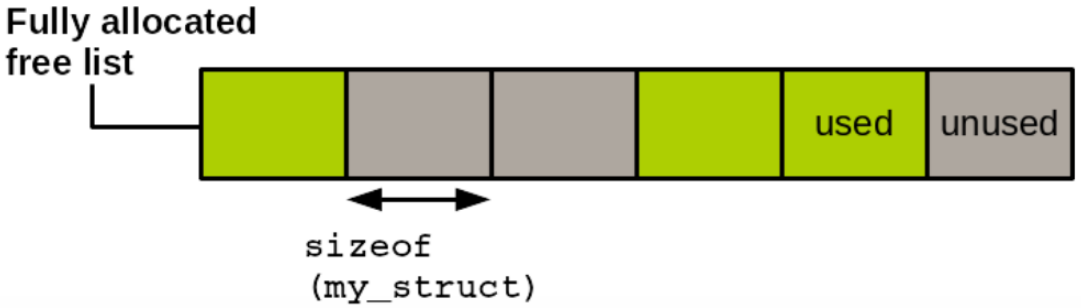
Slab allocator

Allocating/freeing data structures frequently happens in the kernel

Q: how to make memory allocation faster?

Caching objects using a free list

- Store 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

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

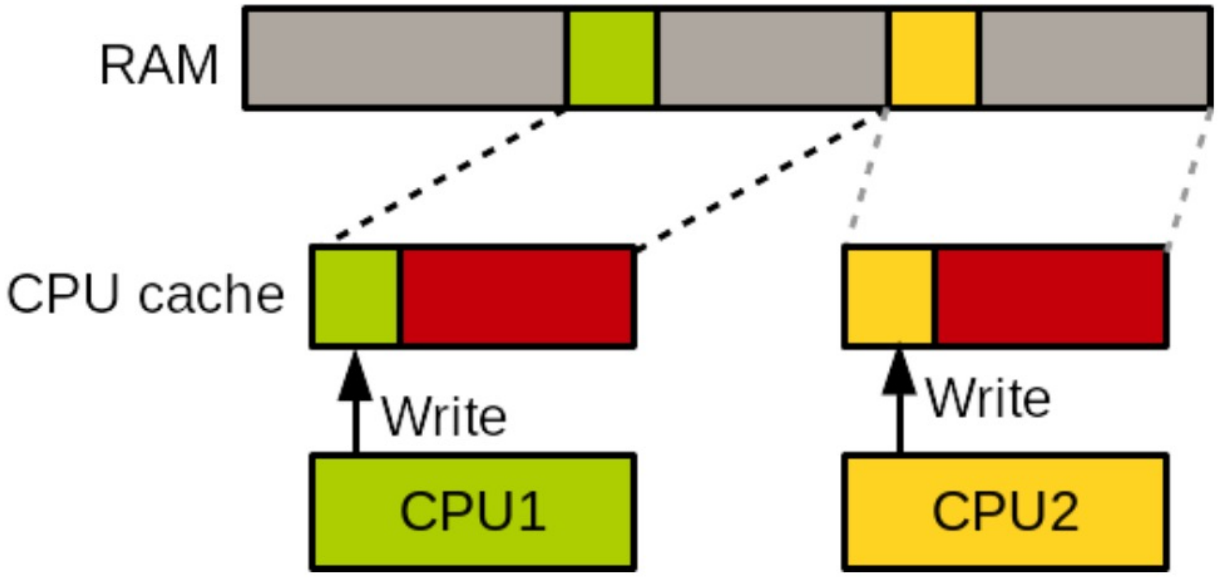
```
/**
 * Create a cache for a data structure type
 */
struct kmem_cache *kmem_cache_create(
    const char *name,     /* Name of the cache */
    size_t size,          /* Size of objects */
    size_t align,         /* Offset of the first element
                             within pages */
    unsigned 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

```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;
```

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 <limit> <batchcount>
kvm_async_pf           0      0    136   30    1 : tunables    0    0    0 : slabdata      0      0      0
kvm_vcpu              24     24   9152    3    8 : tunables    0    0    0 : slabdata      8      8      0
kvm_mmu_page_header    946    946    184   22    1 : tunables    0    0    0 : slabdata     43     43      0
x86_emulator           96     96   2656   12    8 : tunables    0    0    0 : slabdata      8      8      0
i915_dependency       256    256    128   32    1 : tunables    0    0    0 : slabdata      8      8      0
execute_cb             0      0     64   64    1 : tunables    0    0    0 : slabdata      0      0      0
i915_request         750    851    704   23    4 : tunables    0    0    0 : slabdata     37     37      0
drm_i915_gem_object    717   1288   1152   28    8 : tunables    0    0    0 : slabdata     46     46      0
ext4_groupinfo_4k   1914   1914    184   22    1 : tunables    0    0    0 : slabdata     87     87      0
scsi_sense_cache     320    320    128   32    1 : tunables    0    0    0 : slabdata     10     10      0
fsverity_info          0      0    272   30    2 : tunables    0    0    0 : slabdata      0      0      0
fscrypt_info           0      0    136   30    1 : tunables    0    0    0 : slabdata      0      0      0
MPTCPv6                0      0   2112   15    8 : tunables    0    0    0 : slabdata      0      0      0
ip6-frags             22     22    184   22    1 : tunables    0    0    0 : slabdata      1      1      0
PINGv6                 0      0   1216   26    8 : tunables    0    0    0 : slabdata      0      0      0
RAWv6                182    182   1216   26    8 : tunables    0    0    0 : slabdata      7      7      0
UDPv6                192    192   1344   24    8 : tunables    0    0    0 : slabdata      8      8      0
```

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);
```

Further readings

Virtual Memory: 3 What is Virtual Memory?

20 years of Linux virtual memory

Complete virtual memory map x86_64 architecture