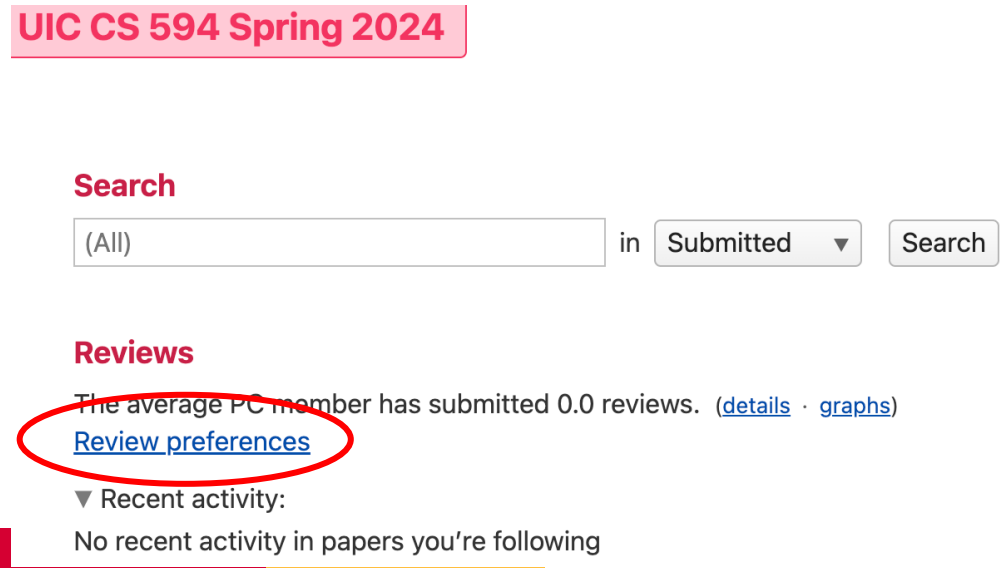
Seminar session

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/

- Deadline for review preference selection: March 8th 5:00 PM



Hw9: virtualization and device drivers

An optional homework

- Up to 25 points (apply to your homework credit if you do not reach the total score in all your homework)

QEMU device emulation and Linux device drivers

Recap: Linux device drivers and I/O virt

Device drivers

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

Write a Linux device driver

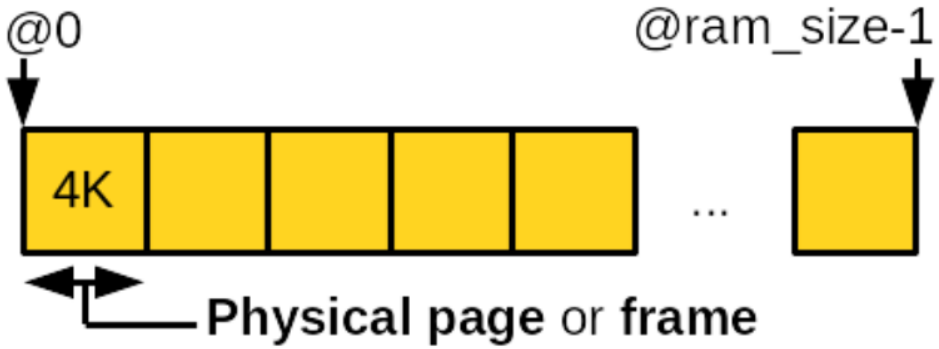
I/O virtualization

- Front-end/Back-end driver model
- Emulation (QEMU)
- Hardware I/O virtualization support (SR-IOV)

OS-level lightweight VM (a.k.a, containers)

Recap: Pages

Memory is divided into **physical pages** or **frames**



The **page** is the basic management unit in the kernel

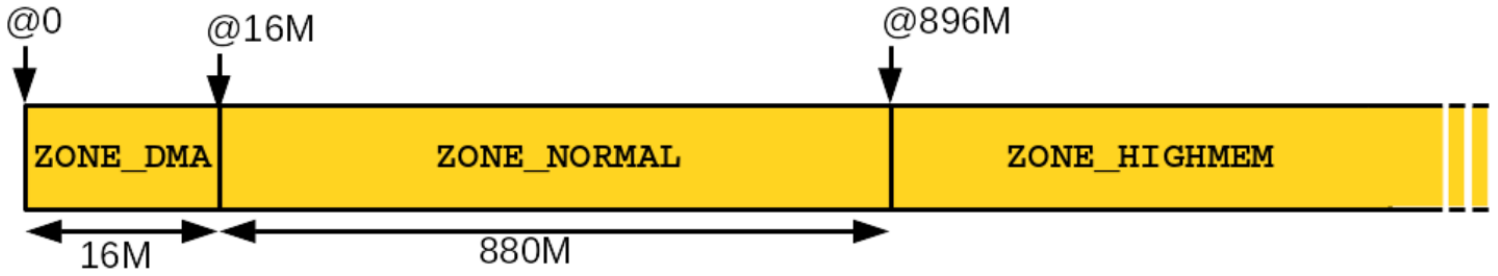
Page size is machine-dependent

- Determined by the the memory management unit (MMU) support
- **4 KB** in general, some are 2 MB and 1 GB: `getconf PAGESIZE`

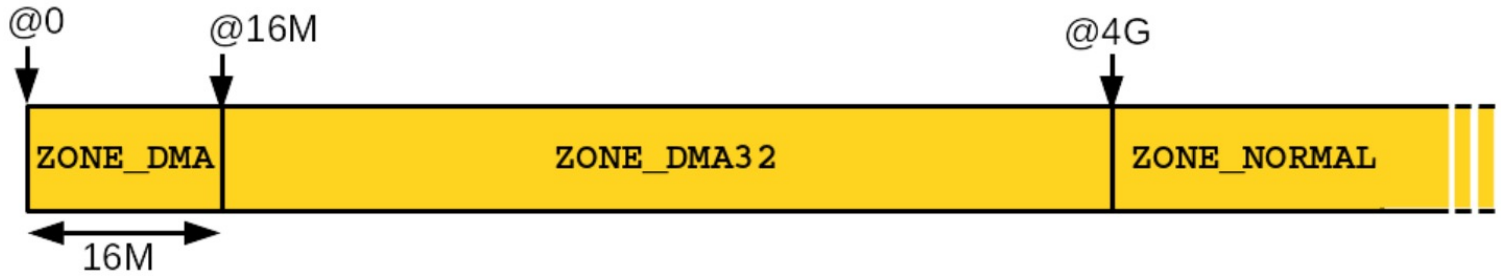
Recap: Zones

Physical memory is partitioned into zones having the same constraints

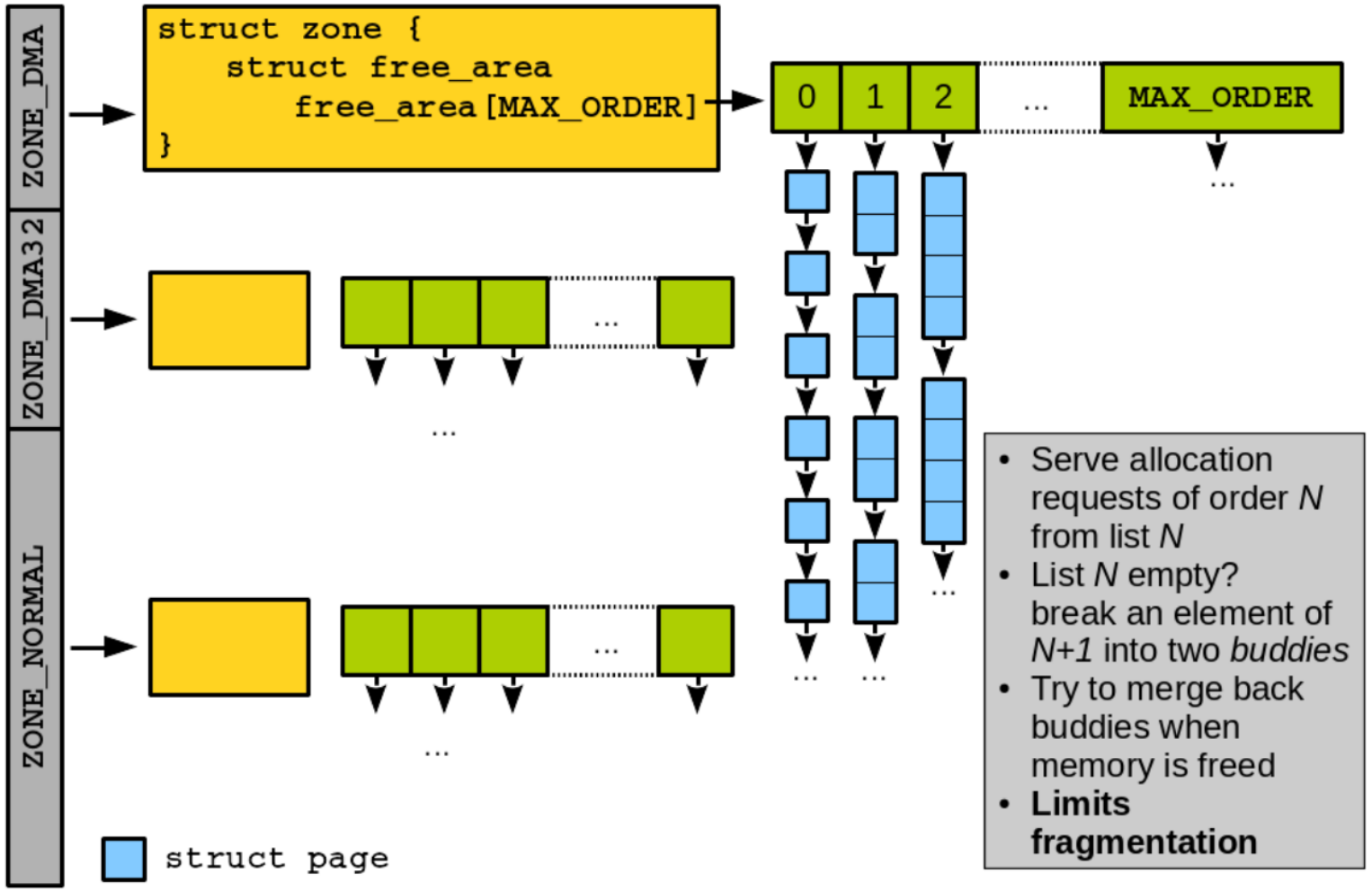
- x86_32 zones layout



- x86_64 zones layout



Recap: 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 pages from being fragmented

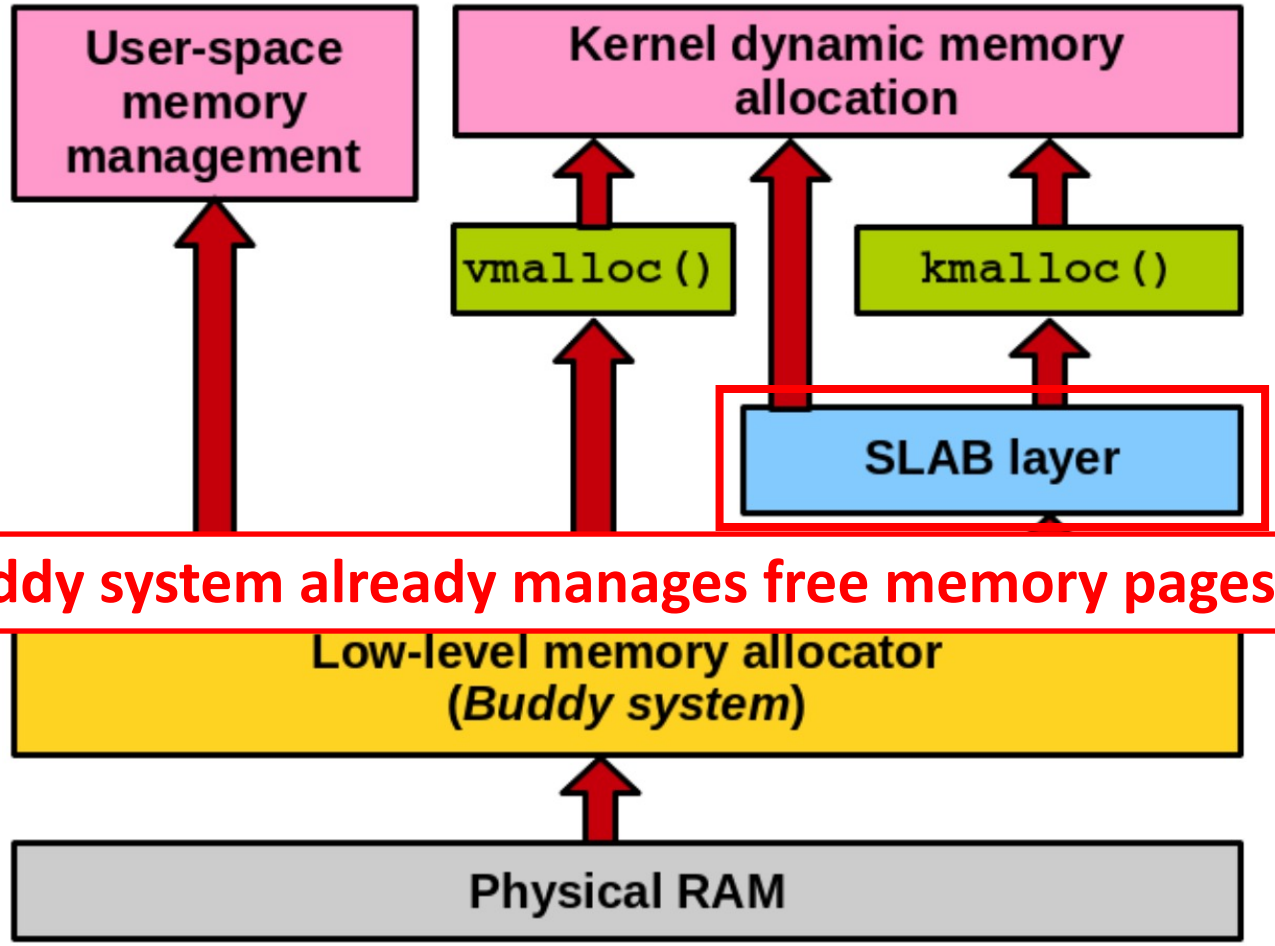
Recap: Page allocation / deallocation

```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);
```

Zeroed pages: `get_zeroed_page(gfp_t gfp_mask);`

`gfp_t`: get free page flags

Recap: Hierarchy of memory allocators



Q: Since the buddy system already manages free memory pages well, why do we need slab?

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 data structure memory allocation faster?

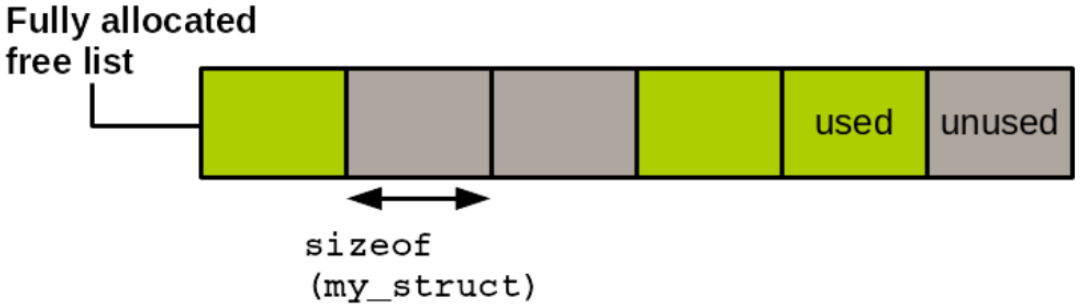
Slab allocator

Allocating/freeing data structures frequently happens in the kernel

Q: how to make data structure 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

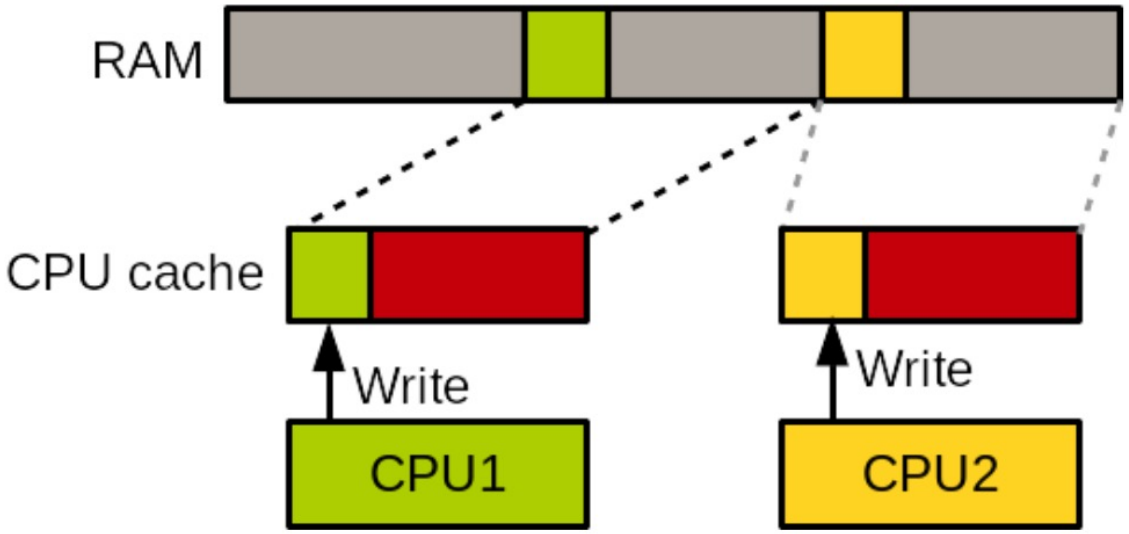
```
/**
 * Create a cache for a data structure type
 */
struct kmem_cache *kmem_cache_create(
    const char *name,     /* Name of the cache */
    size_t size,          /* Size of objects */
    size_t align,         /* Offset of the first element
                             within pages */
    unsigned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

`SLAB_HW_CACHEALIGN`

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



Slab allocator flags

`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 APIs

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

Stacks

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

Further readings

LKD3: Chapter 12 Memory Management

Virtual Memory: 3 What is Virtual Memory?

20 years of Linux virtual memory

Complete virtual memory map x86_64 architecture

Process Address Space

Xiaoguang Wang

Outline

Page tables

Address space

Memory descriptor: `mm_struct`

Virtual Memory Area (VMA): `vm_area_struct`

VMA manipulation

Page tables

Linux enables paging early in the boot process

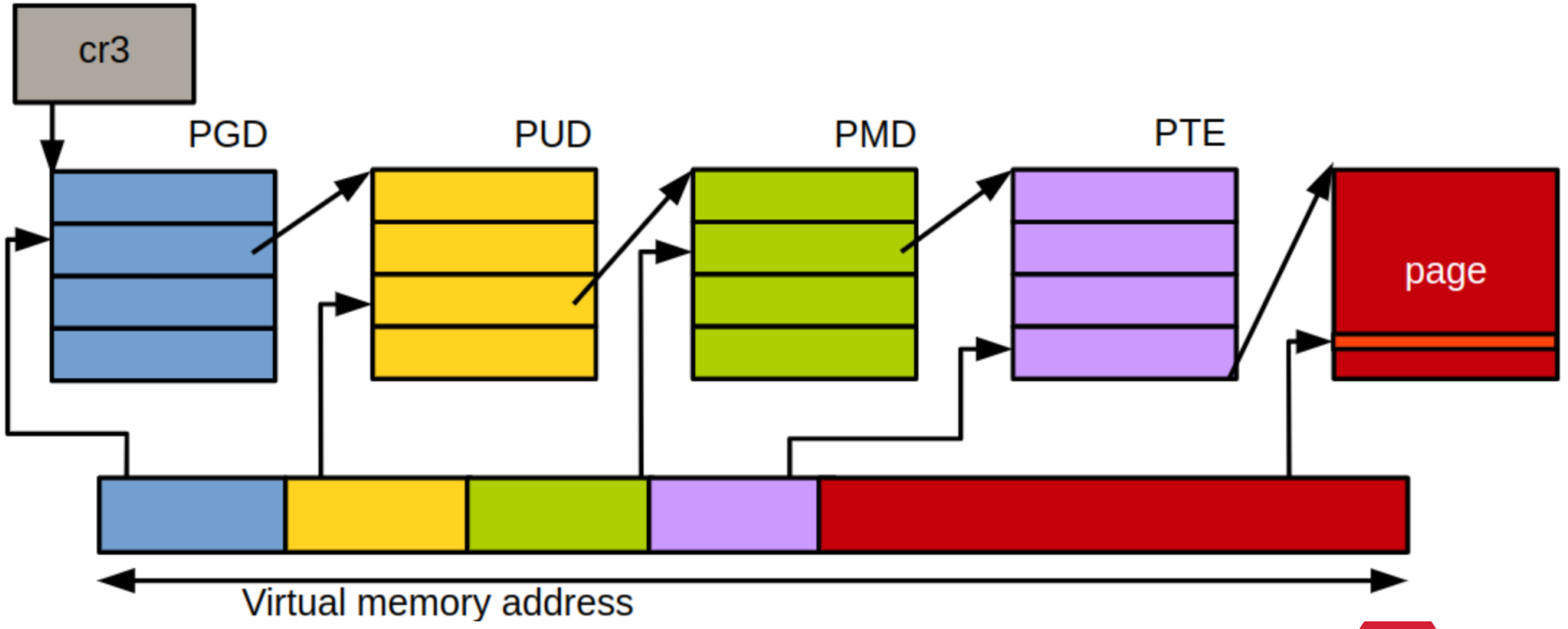
- All memory accesses made by the CPU are virtual and translated to physical addresses through the page tables
- Linux set the page tables and the translation is made automatically by the hardware (MMU) according to the page tables content

The address space is defined by VMAs and is sparsely populated

- One address space per process → one page table per process
- Lots of “empty” areas

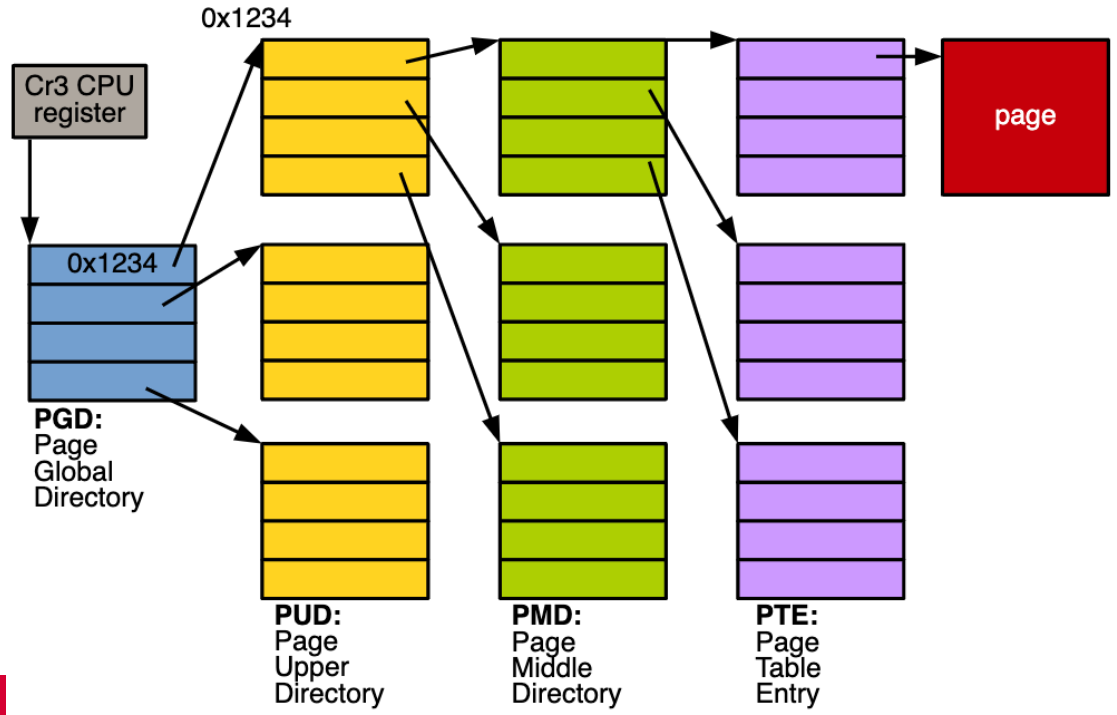
Page tables

Address translation is performed by the hardware memory management unit (**MMU**)



Page tables

```
/* include/linux/mm_types.h */
struct mm_struct {
    struct vm_area_struct *mmap;          /* list of VMAs */
    struct rb_root        mm_rb;          /* rbtree of VMAs */
    pgd_t                 *pgd;           /* page global directory */
    /* ... */
};
```



Virtual address map in 64-bit Linux

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [47:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB)   **direct mapping of all phys. memory**
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffbffffffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...
                    vaddr_end for KASLR
fffffe0000000000 - fffffe7fffffffff (=39 bits) cpu_entry_area mapping
fffffe8000000000 - fffffeffffffffff (=39 bits) LDT remap for PTI
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffef00000000 - ffffffffffffffff (=64 GB)   EFI region mapping space
... unused hole ...
ffffffff80000000 - ffffffff9fffffff (=512 MB) kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffffefffff (1520 MB) module mapping space
[fixmap start]   - ffffffffff5fffff           kernel-internal fixmap range
ffffffffff600000 - ffffffffff600fff (=4 kB)   legacy vsyscall ABI
ffffffffffe00000 - ffffffffffffffff (=2 MB)   unused hole
```

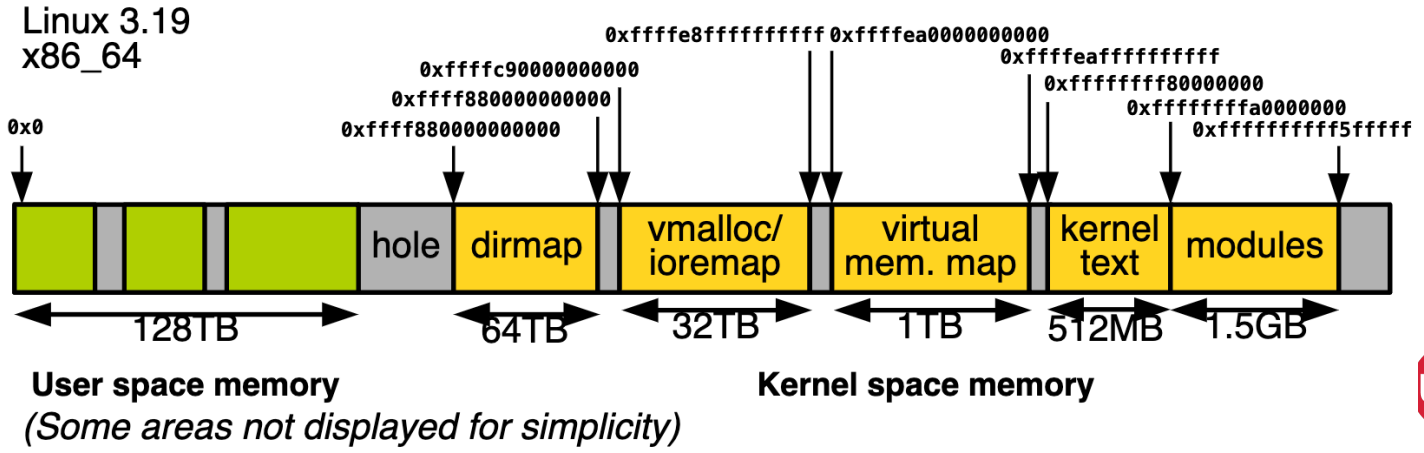
https://www.kernel.org/doc/html/v5.8/x86/x86_64/mm.html

Address space

The memory that a process can access

- Illusion that the process can access 100% of the system memory
- Virtual memory can be much larger than the actual amount of physical memory

Defined by the process page table set up by the kernel



Address space

A memory address is an index within the address spaces:

- Identify a specific byte

Each process is given a flat 32/64-bits address space

- Not segmented

Address space

Virtual Memory Areas (VMA)

- Interval of addresses that the process has the right to access
- Can be dynamically added or removed to the process address space
- Associated permissions: read, write, execute
- *Illegal access → segmentation fault*

```
└─$ sudo cat /proc/1/maps     # or sudo pmap 1
55ff919f7000-55ff91a2d000 r--p 00000000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91a2d000-55ff91b0d000 r-xp 00036000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91b0d000-55ff91b6c000 r--p 00116000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91b6c000-55ff91bbb000 r--p 00174000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91bbb000-55ff91bbc000 rw-p 001c3000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91bbc000-55ff91bbd000 rw-p 00000000 00:00 0
55ff93061000-55ff93383000 rw-p 00000000 00:00 0                          [heap]
```

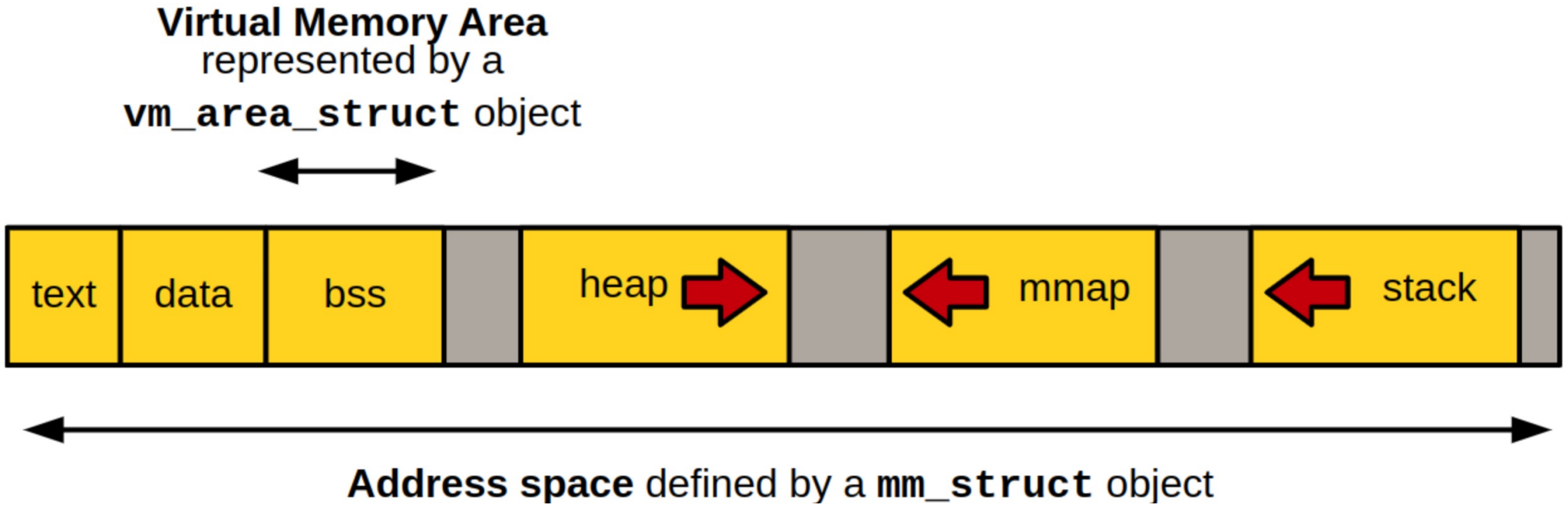
CIENCE

Address space

VMAs can contain:

- Mapping of the executable file code (.text section)
- Mapping of the executable file initialized variables (.data section)
- Mapping of the zero page for uninitialized variables (.bss section)
- Mapping of the zero page for the user-space stack
- Text, data, bss for each shared library used
- Memory-mapped files, shared memory segment, anonymous mappings (used by malloc)

Address space



Quiz: which is (are) wrong

1. MMU is a hardware component typically integrated into the CPU or found as a separate chip closely connected to the CPU.
2. MMU is responsible for managing the translation of virtual addresses used by software into physical addresses in the system's RAM.
3. MMUs can implement either segmentation or paging or both.
4. MMU caches the recently translated virtual-to-physical address mappings using translation lookaside buffer (TLB).
5. The VMA information is directly loaded into the MMU.
6. Each entry in `/proc/<pid>/maps` corresponds to a VMA in Linux kernel.
7. The Linux kernel `mm_struct` represents a VMA entry in a process.

Memory descriptor: `mm_struct`

Address space in Linux kernel: `struct mm_struct`

```
/* include/linux/mm_types.h */
struct mm_struct {
    struct vm_area_struct *mmap;            /* list of VMAs */
    struct rb_root        mm_rb;            /* rbtree of VMAs */
    pgd_t                 *pgd;             /* page global directory */
    atomic_t              mm_users;         /* address space users */
    atomic_t              mm_count;         /* primary usage counters */
    int                   map_count;        /* number of VMAs */
    struct rw_semaphore   mmap_sem;         /* VMA semaphore */
    spinlock_t            page_table_lock;  /* page table lock */
    struct list_head      mmlist;           /* list of all mm_struct */
    unsigned long         start_code;       /* start address of code */
    unsigned long         end_code;         /* end address of code */
    unsigned long         start_data;       /* start address of data */
    unsigned long         end_data;         /* end address of data */
    unsigned long         start_brk;   /* start address of heap */
    unsigned long         end_brk;     /* end address of heap */
    unsigned long         start_stack; /* start address of stack */
    /* ... */
```

Memory descriptor: `mm_struct`

```
        unsigned long           arg_start;    /* start of arguments */
        unsigned long           arg_end;      /* end of arguments */
        unsigned long           env_start;    /* start of environment */
        unsigned long           total_vm;     /* total pages mapped */
        unsigned long           locked_vm;    /* number of locked pages */
        unsigned long           flags;        /* architecture specific data */
        spinlock_t              ioctx_lock;   /* Asynchronous I/O list lock */
        /* ... */
    };
```

`mm_users`: number of processes (threads) using the address space

`mm_count`: reference count:

- +1 if `mm_users` > 0
- +1 if the kernel is using the address space
- When `mm_count` reaches 0, the `mm_struct` can be freed

Allocate a memory descriptor

A task memory descriptor is in the `mm` field of the corresponding `task_struct`

```
/* include/linux/sched.h */
struct task_struct {
    struct thread_info          thread_info;
    /* ... */
    const struct sched_class    *sched_class;
    struct sched_entity         se;
    struct sched_rt_entity      rt;
    /* ... */
    struct mm_struct            *mm;
    struct mm_struct            *active_mm;
    /* ... */
};
```

Allocate a memory descriptor

Current task memory descriptor: `current->mm`

During `fork()`, `copy_mm()` makes a copy of the parent memory descriptor for the child

- `copy_mm()` calls `dup_mm()` which calls `allocate_mm()` → allocates a `mm` struct object from a slab cache

Two threads sharing the same address space have the `mm` field of their `task_struct` pointing to the same `mm_struct` object

- Threads are created using the `CLONE_VM` flag passed to `clone()` → `allocate_mm()` is not called

Destroying a memory descriptor

When a process exits, `do_exit()` is called, and it calls `exit_mm()`

- Performs some housekeeping/statistics updates and calls `mmput()`

```
void mmput(struct mm_struct *mm) {
    might_sleep();
    if (atomic_dec_and_test(&mm->mm_users))
        __mmput(mm);
}
static inline void __mmput(struct mm_struct *mm) {
    /* ... */
    mmdrop(mm);
}
static inline void mmdrop(struct mm_struct *mm) {
    if (unlikely(atomic_dec_and_test(&mm->mm_count)))
        __mmdrop(mm);
}
```

The `mm_struct` and kernel threads

Kernel threads do not have a user-space address space

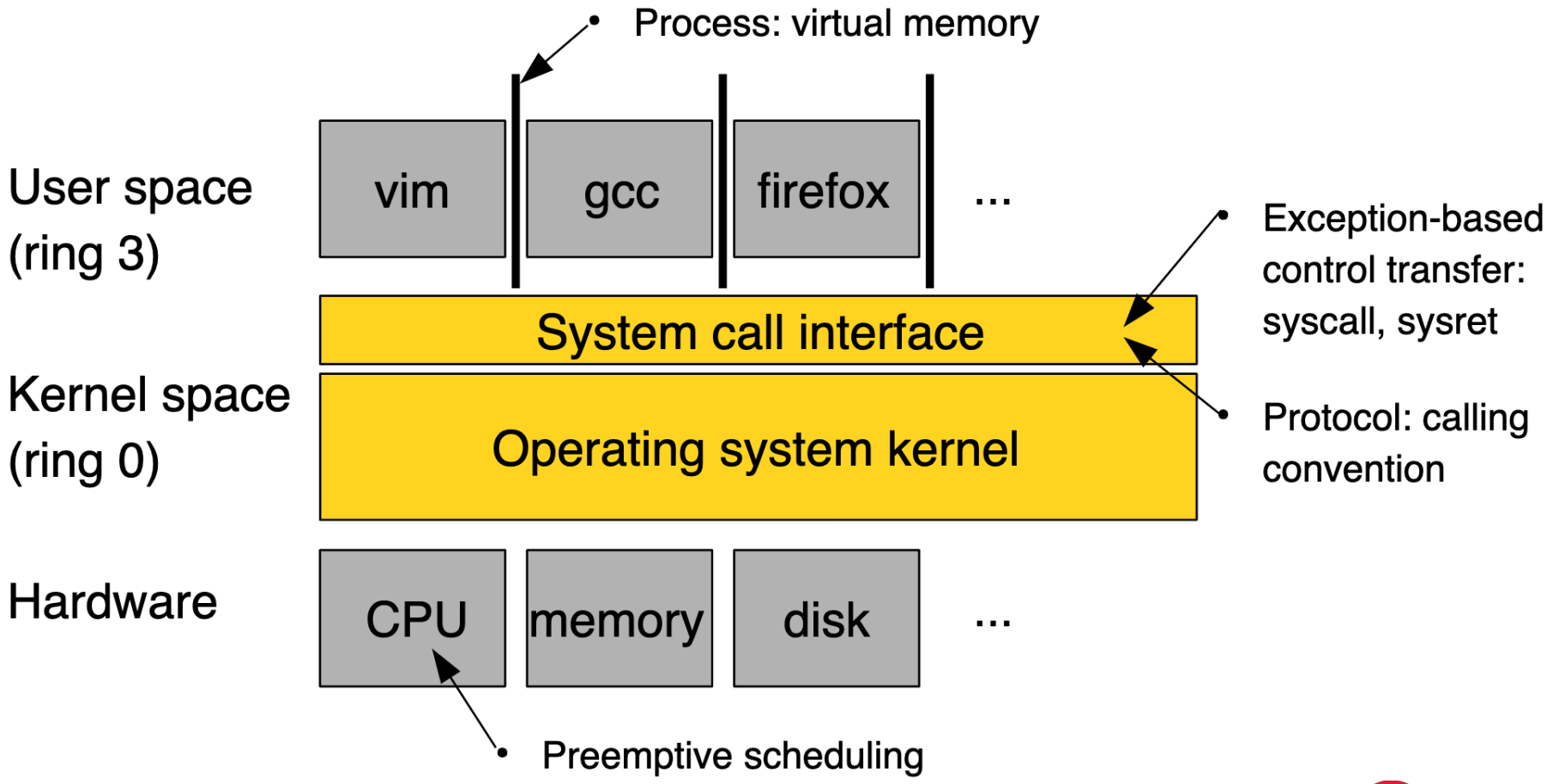
- `mm` field of a kernel thread `task_struct` is NULL

The `mm_struct` and kernel threads

However, kernel threads still need to access the kernel address space

- When a kernel thread is scheduled, the kernel notice its `mm` is `NULL`, so it keeps the previous address space loaded (page tables)
- Kernel makes the `active_mm` field of the kernel thread to point on the borrowed `mm_struct`
- It is okay because the kernel address space is the same in all tasks

The `mm_struct` and kernel threads



Virtual Memory Area (VMA)

Each line corresponds to one VMA

```
└─$ sudo cat /proc/1/maps     # or sudo pmap 1
55ff919f7000-55ff91a2d000 r--p 00000000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91a2d000-55ff91b0d000 r-xp 00036000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91b0d000-55ff91b6c000 r--p 00116000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91b6c000-55ff91bbb000 r--p 00174000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91bbb000-55ff91bbc000 rw-p 001c3000 103:02 8658138                   /usr/lib/systemd/systemd
55ff91bbc000-55ff91bbd000 rw-p 00000000 00:00 0
55ff93061000-55ff93383000 rw-p 00000000 00:00 0                          [heap]
```

```
# r = read
# w = write
# x = execute
# s = shared
# p = private (copy on write)
```

Virtual Memory Area (VMA)

Each VMA is represented by an object of type `vm_area_struct`

```
/* include/linux/mm_types.h */
struct vm_area_struct {
    struct                      mm_struct *vm_mm; /* associated address space */
    unsigned long               vm_start;         /* VMA start, inclusive */
    unsigned long               vm_end;           /* VMA end, exclusive */
    struct vm_area_struct       *vm_next;         /* list of VMAs */
    struct vm_area_struct       *vm_prev;         /* list of VMAs */
    pgprot_t                    vm_page_prot;     /* access permissions */
    unsigned long               vm_flags;         /* flags */
    struct rb_node              vm_rb;            /* VMA node in the tree */
    struct list_head            anon_vma_chain;   /* list of anonymous mappings */
    struct anon_vma             *anon_vma;        /* anonmous vma object */
    struct vm_operation_struct  *vm_ops;          /* operations */
    unsigned long               vm_pgoff;         /* offset within file */
    struct file                 *vm_file;         /* mapped file (can be NULL) */
    void                        *vm_private_data; /* private data */
    /* ... */
}
```

Virtual Memory Area (VMA)

The VMA exists over `[vm_start, vm_end)` in the corresponding address space → size in bytes: `vm_end - vm_start`

Each VMA is unique to the associated `mm_struct`

- Two processes mapping the same file will have two different `mm_struct` objects, and two different `vm_area_struct` objects
- Two threads sharing a `mm_struct` object also share the `vm_area_struct` objects

VMA flags

Flag	Effect on the VMA and Its Pages
`VM_READ`	Pages can be read from.
`VM_WRITE`	Pages can be written to.
`VM_EXEC`	Pages can be executed.
`VM_SHARED`	Pages are shared.
`VM_MAYREAD`	The `VM_READ` flag can be set.
`VM_MAYWRITE`	The `VM_WRITE` flag can be set.
`VM_MAYEXEC`	The `VM_EXEC` flag can be set.
`VM_MAYSHARE`	The `VM_SHARE` flag can be set.

VMA flags

Combining `VM_READ`, `VM_WRITE` and `VM_EXEC` gives the permissions for the entire area, for example:

- Executable code is `VM_READ` and `VM_EXEC`
- Stack is `VM_READ` and `VM_WRITE`

`VM_SEQ_READ` and `VM_RAND_READ` are set through the `madvise()` system call

- Instructs the file pre-fetching algorithm read-ahead to increase or decrease its pre-fetch window

VMA flags

`VM_HUGETLB` indicates that the area uses pages larger than the regular size

- 2M and 1G on x86
- Larger page size → less TLB miss → faster memory access

VMA operations

vm_ops in vm_area_struct is a struct of function pointers to operate on a specific VMA

```
/* include/linux/mm.h */
struct vm_operations_struct {
    /* called when the area is added to an address space */
    void (*open)(struct vm_area_struct * area);

    /* called when the area is removed from an address space */
    void (*close)(struct vm_area_struct * area);

    /* invoked by the page fault handler when a page that is
     * not present in physical memory is accessed*/
    int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);

    /* invoked by the page fault handler when a previously read-only
     * page is made writable */
    int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
    /* ... */
}
```

Create an address interval

`do_mmap()` is used to create a new linear address interval

- Can result in the creation of a new VMAs
- Or a merge of the create area with an adjacent one when they have the same permissions

```
/*
 * The caller must hold down_write(&current->mm->mmap_sem).
 */
unsigned long do_mmap(struct file *file, unsigned long addr,
            unsigned long len, unsigned long prot,
            unsigned long flags, vm_flags_t vm_flags,
            unsigned long pgoff, unsigned long *populate,
            struct list_head *uf);
```

Create an address interval

`prot` specifies access permissions for the memory pages

Flag	Effect on the new interval
`PROT_READ`	Corresponds to `VM_READ`
`PROT_WRITE`	Corresponds to `VM_WRITE`
`PROT_EXEC`	Corresponds to `VM_EXEC`
`PROT_NONE`	Cannot access page

Create an address interval

`flags` specifies the rest of the VMA options

Flag	Effect on the new interval
`MAP_SHARED`	The mapping can be shared.
`MAP_PRIVATE`	The mapping cannot be shared.
`MAP_FIXED`	The new interval must start at the given address addr.
`MAP_ANONYMOUS`	The mapping is not file-backed, but is anonymous.
`MAP_GROWSDOWN`	Corresponds to `VM_GROWSDOWN`.

Create an address interval

On error `do_mmap()` returns a negative value

On success

- The kernel tries to merge the new interval with an adjacent one having same permissions
- Otherwise, create a new VMA
- Returns a pointer to the start of the mapped memory area

`do_mmap()` is exported to user-space through `mmap2()`

```
void *mmap2(void *addr, size_t length, int prot,
            int flags, int fd, off_t pgoffset);
```

Remove an address interval

Removing an address interval is done through `do_munmap()`

```
/* linux/include/linux/mm.h */
int do_munmap(struct mm_struct *, unsigned long, size_t);
```

Exported to user-space through `munmap()`

```
int munmap(void *addr, size_t len);
```

Further readings

LKD3: Chapter 15 The Process Address Space

Supporting bigger and heterogeneous memory efficiently

- AutoNUMA, Transparent Hugepage Support, Five-level page tables
- Heterogeneous memory management

Optimization for virtualization

- Kernel same-page merging (KSM)
- MMU notifier