CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 17

Memory Management (II)

April 1, 2025

Huaicheng Li

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

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

Previously:

- Pages and zones
- Page allocation
- kmalloc, vmalloc (recap)

➔ Today

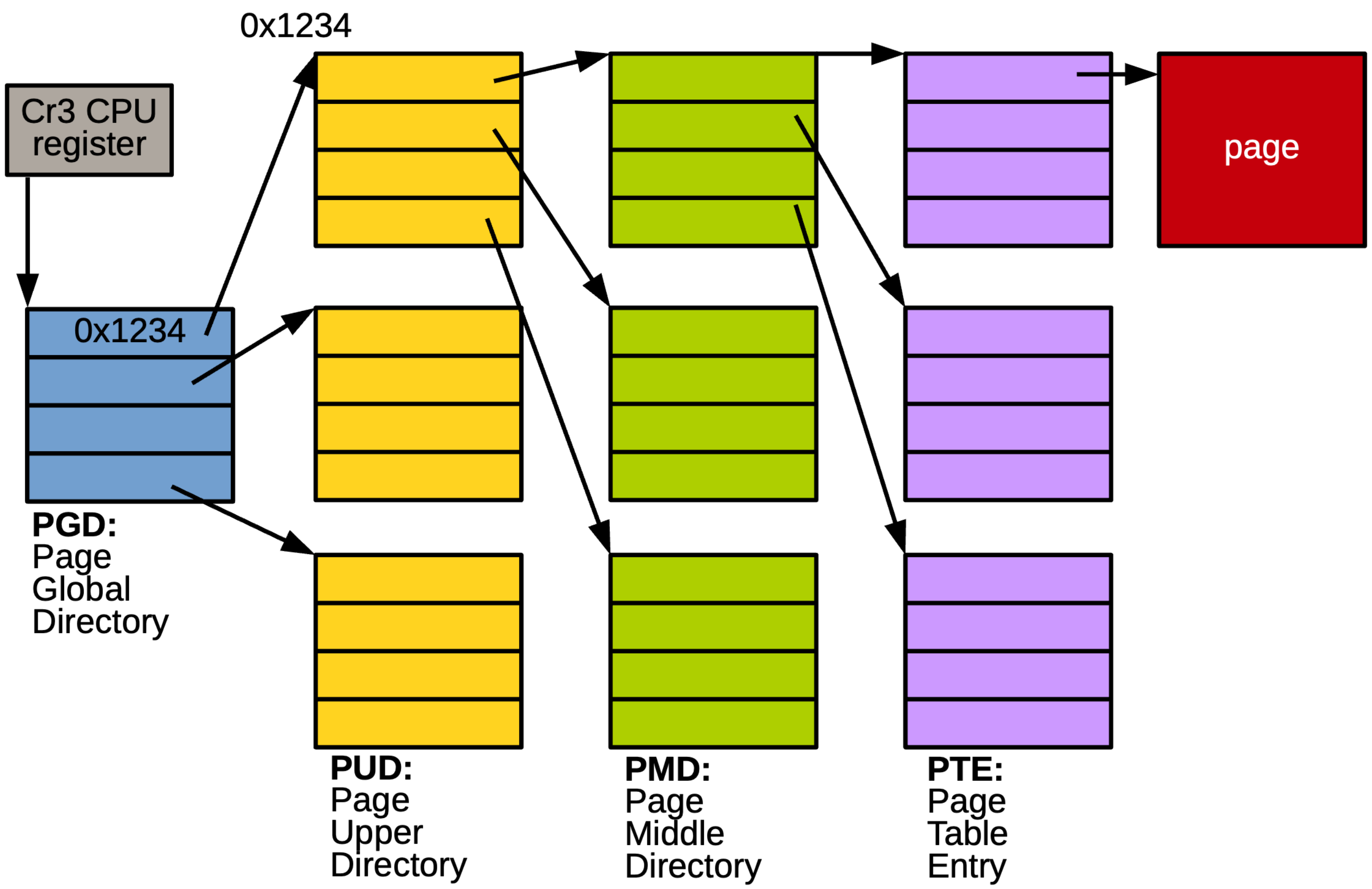
- Slab allocator
- Stack, high memory, per-CPU data structures
- Page tables
- Address space
- Memory descriptor: mm_struct
- Virtual Memory Area (VMA)
- VMA manipulation

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 sets up the page table and the translation is made automatically by the hardware (MMU) according to the page table 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

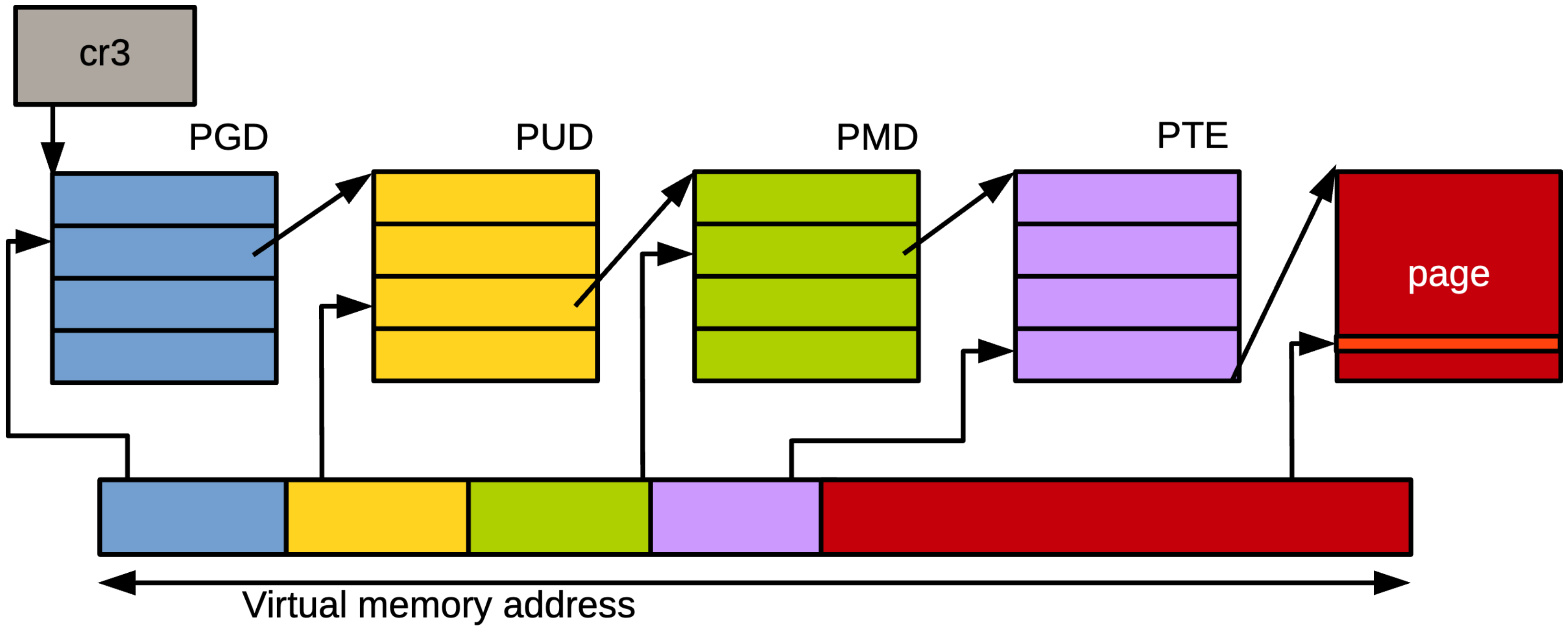
Page Tables

```
/* linux/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 */
    /* ... */
};
```



Page Tables

- Address translation is performed by the hardware (MMU)

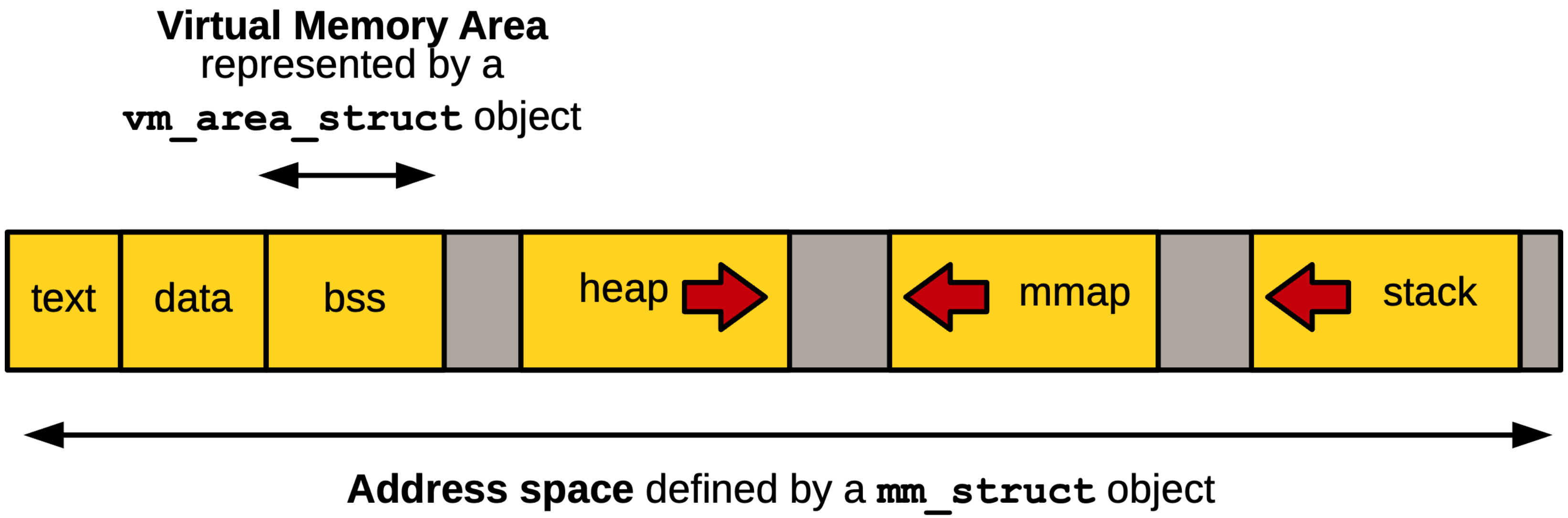


Virtual Address Map in Linux

```
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 - fffffffeffffffff (=64 GB)   EFI region mapping space
... unused hole ...
ffffffff80000000 - ffffffff9fffffff (=512 MB) kernel text mapping, from phys 0
ffffffffa0000000 - fffffffffeffffff (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
```

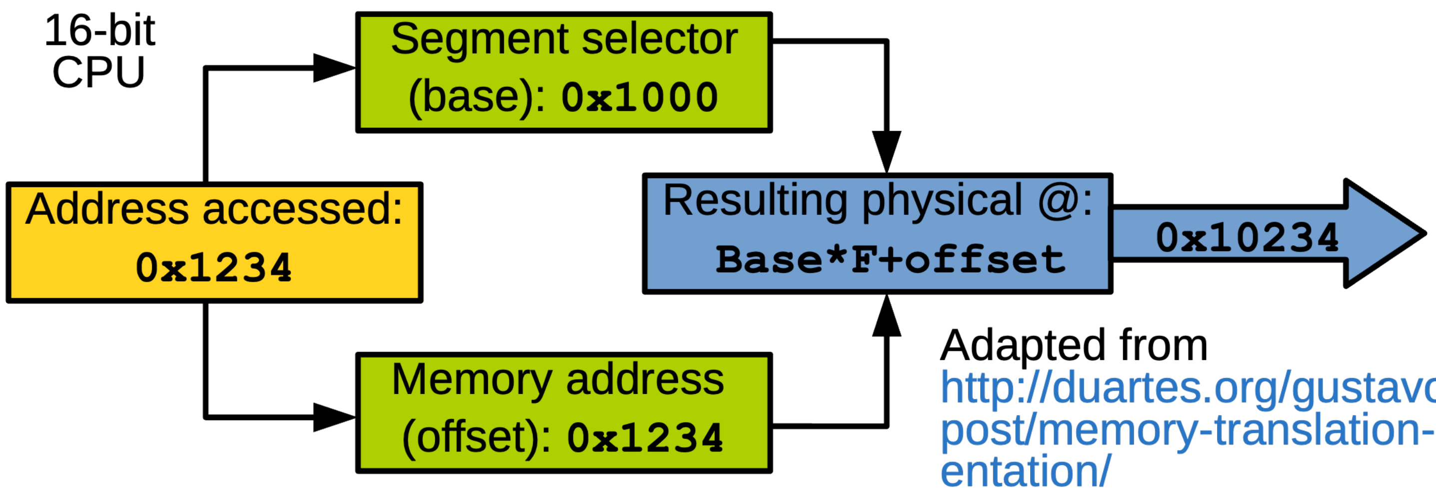
Address Space

- The memory that a process can access
  - Illusion that the process can access 100% of the system memory
  - With virtual memory, can be much larger than the actual amount of physical memory
- Defined by the process page table setup by the kernel



Address Space

- A memory address is an index within the address space
  - Identify a specific byte
- Each process is given a flat 32/64-bits address space
  - Not segmented



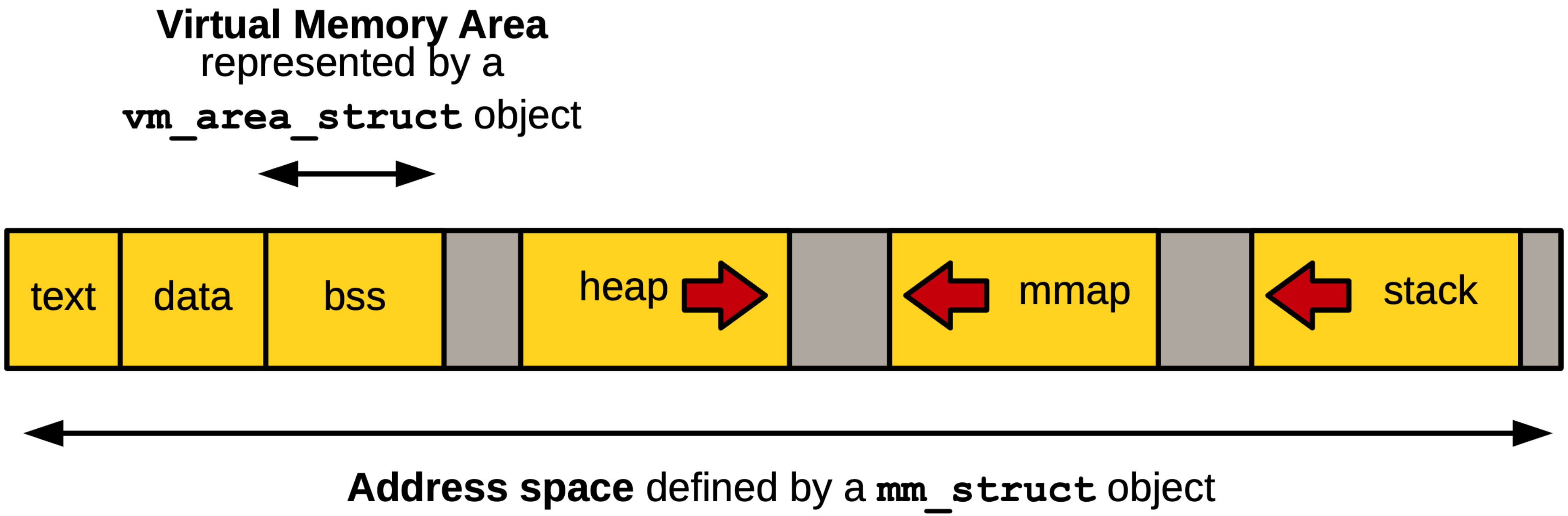
Adapted from
http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation/

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

```
$ cat /proc/1/maps        # or sudo pmap 1
55fe3bf02000-55fe3bff9000 r-xp 00000000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3bffa000-55fe3c021000 r--p 000f7000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3c021000-55fe3c022000 rw-p 0011e000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3db4a000-55fe3ddfd000 rw-p 00000000 00:00 0        [heap]
7f7522769000-7f7522fd9000 rw-p 00000000 00:00 0
7f7523150000-7f7523265000 r-xp 00000000 fd:00 1979800  /usr/lib64/libm-2.25.so
```

- 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)


Virtual Memory Area
represented by a
vm_area_struct object
text
data
bss
heap
mmap
stack
Address space defined by a mm_struct object

Memory Descriptor: mm_struct

```
/* linux/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         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 addr space
- mm_count: reference count
  - +1 if num_users > 0
  - +1 if the kernel is using the address space
  - When mm_count reaches 0, the mm_struct can be freed
- mmap and mm_rb are respectively a linked list and a tree containing all the VMAs in the addr space
  - List is used to iterate over all the VMAs in an ascending order
  - Tree is used to find a specific VMA
- All mm_struct are linked together in a doubly linked list
  - Through the mmlist field in the mm_struct

Allocating a Memory Descriptor

- A task memory descriptor is located in the ”mm” field of the “task_struct”

```
/* linux/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;
    /* ... */
};
```

Allocating 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() → alloates an "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

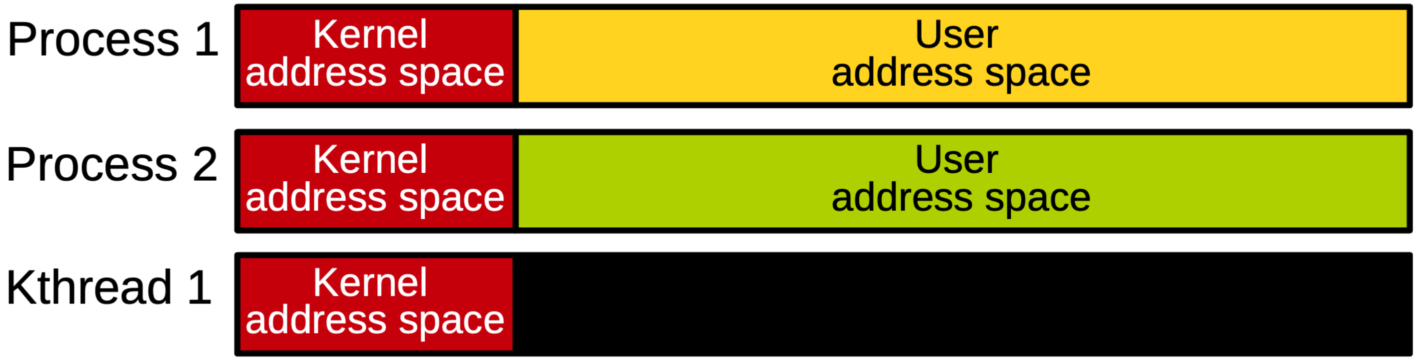
Destroying a Memory Descriptor

- 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)))
```

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 kernel threads still need to access the kernel address space
  - When a kernel thread is scheduled, the kernel notices 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 to the borrowed mm_struct
  - It is okay b/c the kernel address space is the same in all tasks

VMA

- Each line corresponds to one VMA

```
$ cat /proc/1/maps  # or sudo pmap 1
55fe3bf02000-55fe3bff9000 r-xp 00000000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3bffa000-55fe3c021000 r--p 000f7000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3c021000-55fe3c022000 rw-p 0011e000 fd:00 1975429  /usr/lib/systemd/systemd
55fe3db4a000-55fe3ddfd000 rw-p 00000000 00:00 0        [heap]
7f7522769000-7f7522fd9000 rw-p 00000000 00:00 0
7f7523150000-7f7523265000 r-xp 00000000 fd:00 1979800  /usr/lib64/libm-2.25.so
7f7523265000-7f7523464000 ---p 00115000 fd:00 1979800  /usr/lib64/libm-2.25.so
7f7523464000-7f7523465000 r--p 00114000 fd:00 1979800  /usr/lib64/libm-2.25.so
7f7523465000-7f7523466000 rw-p 00115000 fd:00 1979800  /usr/lib64/libm-2.25.so

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

- Each VMA is represented by an object of type "vm_area_struct"

```
/* linux/include/linux/mm_types.h */

struct vm_area_struct {
    struct                      mm_struct *vm_mm; /* associated address space
         (mm_struct) */
    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 */
```

VMA

- The VMA exists over [vm start, vm end) in the corresponding address space → size in bytes: (vm_end – vm_start)
- Address spaxe is pointed by the vm_mm field (of type mm_struct)
- 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 an 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.

Flag	Effect on the VMA and Its Pages
`VM_GROWSDOWN`	The area can grow downward.
`VM_GROWSUP`	The area can grow upward.
`VM_SHM`	The area is used for shared memory.
`VM_DENYWRITE`	The area maps an unwritable file.
`VM_EXECUTABLE`	The area maps an executable file.
`VM_LOCKED`	The pages in this area are locked.
`VM_IO`	The area maps a device's I/O space.
`VM_SEQ_READ`	The pages seem to be accessed sequentially.

Flag	Effect on the VMA and Its Pages
`VM_RAND_READ`	The pages seem to be accessed randomly.
`VM_DONTCOPY`	This area must not be copied on fork().
`VM_DONTEXPAND`	This area cannot grow via mremap().
`VM_RESERVED`	This area must not be swapped out.
`VM_ACCOUNT`	This area is an accounted VM object.
`VM_HUGETLB`	This area uses hugetlb pages.
`VM_NONLINEAR`	This area is a nonlinear mapping.

VMA Flags

- Combining VM_READ, VM_WRITE, and VM_EXEC gives the permission for the entire area, for example
  - Object 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() syscall
  - Instructs the file pre-fetching algorithm read-ahread to increase or decrease its prefetch window
- VM_HUGETLB indicates that the area uses pages larger than the regular size
  - 2MB and 1GB 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

```
/* linux/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);
    /* ... */
}
```

VMA Manipulation: find_vma()

```c
/* linux/mm/mmap.c */

/* Look up the first VMA which satisfies  addr < vm_end,  NULL if none. */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
    struct rb_node *rb_node;
    struct vm_area_struct *vma;

    /* Check the cache first. */
    vma = vmacache_find(mm, addr);
    if (likely(vma))
        return vma;

    rb_node = mm->mm_rb.rb_node;

    while (rb_node) {
        struct vm_area_struct *tmp;

        tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);

        if (tmp->vm_end > addr) {
            vma = tmp;
            if (tmp->vm_start <- addr)
                break;
            rb_node = rb_node->rb_left;
        } else
            rb_node = rb_node->rb_right;
    }

    if (vma)
        vmacache_update(addr, vma);
    return vma;
}
```

```c
/* linux/include/linux/mm.h */

/* Look up the first VMA which satisfies  addr < vm_end,  NULL if none. */
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr);

/*
 * Same as find_vma, but also return a pointer to the previous VMA in *pprev.
 */
struct vm_area_struct *
find_vma_prev(struct mm_struct *mm, unsigned long addr,
              struct vm_area_struct **pprev);

/* Look up the first VMA which intersects the interval start_addr..end_addr-1,
   NULL if none.  Assume start_addr < end_addr. */
struct vm_area_struct * find_vma_intersection(struct mm_struct * mm,
              unsigned long start_addr, unsigned long end_addr);
```

Creating an Address Interval

- do_mmap() is used to create a new linear address interval
  - Can result in the creation of a new VMA
  - Or a merge of the created 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);
```

- `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

- “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`.

Flag	Effect on the new interval
`MAP_DENYWRITE`	Corresponds to `VM_DENYWRITE`.
`MAP_EXECUTABLE`	Corresponds to `VM_EXECUTABLE`.
`MAP_LOCKED`	Corresponds to `VM_LOCKED`.
`MAP_NORESERVE`	No need to reserve space for the mapping.
`MAP_POPULATE`	Populate (prefault) page tables.
`MAP_NONBLOCK`	Do not block on I/O.

- On error do_mmap() returns a negative value
- On success
  - The kernel tries to merge the new interval with an adjacent one having the 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);
```

Removing an Address Interval

- Removing an address interval is done through do_munmap()
- Exported to user-space through munmap()

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

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

Further Readings

- Introduction to Memory Management in Linux
- 20 years of Linux virtual memory
- Linux Kernel virtual Memory Map
- Kernel page-table isolation
- Heterogeneous memory support
  - AutoNUMA, Transparent Hugepage Support, Five-level page tables
- Optimizations for virtualization
  - Kernel same-page merging (KSM)
  - MMU notifier