Linux Kernel Programming

Process Address Space

Pierre Olivier

Systems Software Research Group @ Virginia Tech

March 30, 2017

Virginia Tech

Outline

Virginia Tech

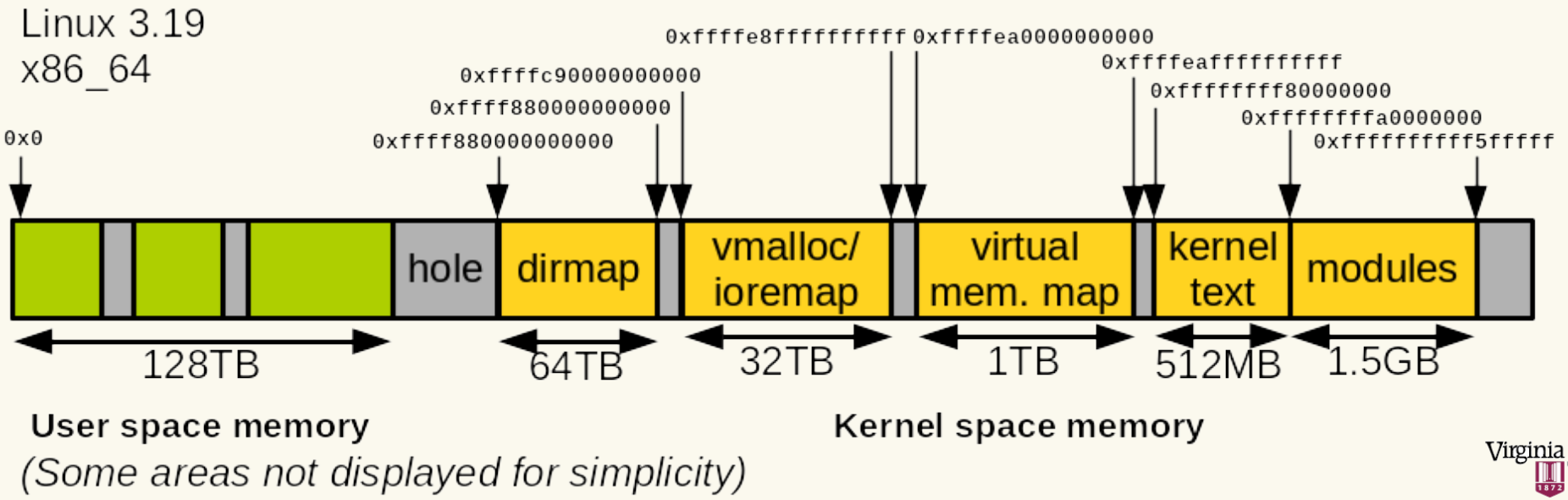
Outline

Virginia Tech

Address space and memory descriptor

Address space

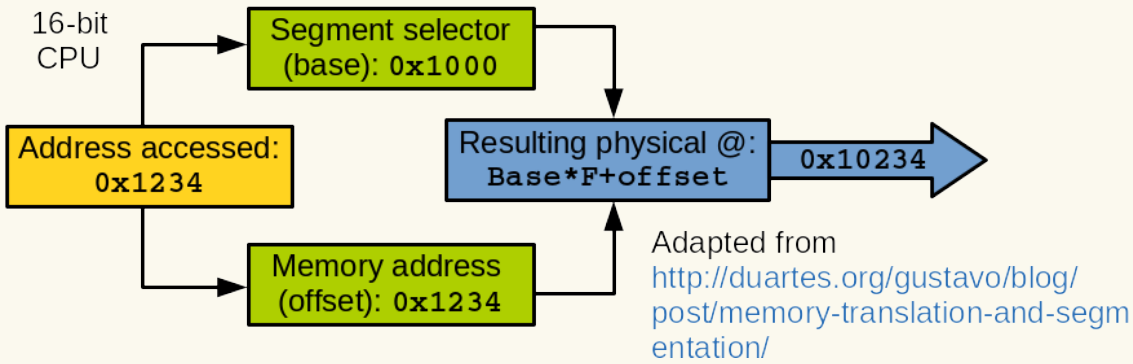
- **The memory that a process can access is called its address space**
  - Illusion that the process can access 100% of the system memory
  - With virtual memory, can be much larger than the actual amount of physical memory
- Defined by the process page table set up by the kernel



Address space and memory descriptor

Address space (2)

- Each process is given a *flat* 32/64-bits address space
  - *Flat* as opposed to segmented



- A **memory address** is an index within the address spaces:
  - Identify a specific byte
  - Example: `0x8fffa12dd24123fd`

Address space and memory descriptor

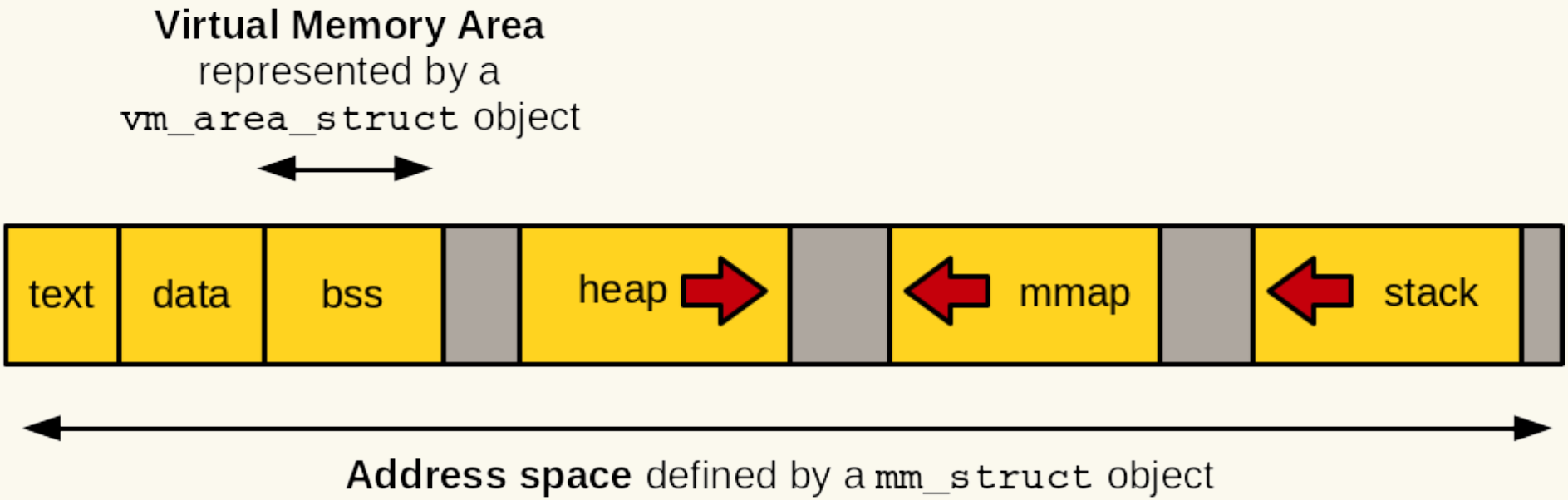
Address space (3)

- Interval of addresses that the process has the right to access: **virtual memory areas** (**VMAs**)
  - VMAs can be dynamically added or removed to the process address space
  - VMAs have associated permissions: read, write, execute
    - When a process try to access an address outside of valid VMAs, or access a VMA with wrong permissions: **segmentation fault**
- **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)

Virginia Tech

Address space and memory descriptor

Address space (4)



Address space and memory descriptor

Memory descriptor

- The kernel represent a process address space through a **`struct mm_struct`** object, the **memory descriptor**
  - Defined in `include/linux/mm_types.h`
  - Interesting fields:

```
1  struct mm_struct {
2    struct vm_area_struct *mmap;           /* list of VMAs */
3    struct rb_root        mm_rb;           /* rbtree of VMAs */
4    pgd_t                 *pgd;            /* page global directory */
5    atomic_t              mm_users;        /* address space users */
6    atomic_t              *mm_count;       /* primary usage counters */
7    int                   map_count;       /* number of VMAs */
8    struct rw_semaphore   mmap_sem;        /* VMA semaphore */
9    spinlock_t            page_table_lock; /* page table lock */
10   struct list_head      mmlist;          /* list of all mm_struct */
11   unsigned long         start_code;      /* start address of code */
12   unsigned long         end_code;        /* end address of code */
13   unsigned long         start_data;      /* start address of data */
14   unsigned long         end_data;        /* end address of data */
15 /* ... */
```

Address space and memory descriptor

Memory descriptor (2)

```
16    /* ... */
17    unsigned long         start_brk;  /* start address of heap */
18    unsigned long         end_brk;    /* end address of heap */
19    unsigned long         start_stack; /* start address of stack */
20    unsigned long         arg_start;   /* start of arguments */
21    unsigned long         arg_end;     /* end of arguments */
22    unsigned long         env_start;   /* start of environment */
23    unsigned long         total_vm;    /* total pages mapped */
24    unsigned long         locked_vm;   /* number of locked pages */
25    unsigned long         flags;       /* architecture specific data */
26    spinlock_t            ioctx_lock;  /* Asynchronous I/O list lock */
27    /* ... */
28 };
```

- `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

Virginia Tech

Address space and memory descriptor

Memory descriptor (3)

- `mmap` and `mm_rb` are respectively a linked list and a tree containing all the VMAs in this address space
  - List used to iterate over all the VMAs
    - Links all VMAs sorted by ascending virtual addresses
  - Tree used to find a specific VMA
- All `mm_struct` are linked together in a doubly linked list
  - Through the `mmlist` field if the `mm_struct`

Virginia Tech

Address space and memory descriptor

Memory descriptor allocation

- **A task memory descriptor is located in the `mm` field of the corresponding `task_struct`**
  - Current task memory descriptor: `current->mm`
  - During `fork()`, `copy_mm()` is making 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
    - in `copy_mm()`:

```
1  /* ... */
2  struct mm_struct *oldmm;
3  oldmm = current->mm;
4  /* ... */
5  if (clone_flags & CLONE_VM) {
6    atomic_inc(&oldmm->mm_users);
7    mm = oldmm;
8    goto good_mm;
9  }
10 /* ... */
11 good_mm:
12   /* ... */
13 return 0;
```

Address space and memory descriptor

Memory descriptor destruction

- When a process exits, `do_exit()` is called
  - It calls `exit_mm()`
    - Performs some housekeeping/statistics updates
    - Calls `mmput()`

```
1 void mmput(struct mm_struct *mm)
2 {
3   might_sleep();
4
5   if (atomic_dec_and_test(&mm->mm_users))
6     __mmput(mm);
7 }
```

    - `mmput()` decrements the `users` field and calls `__mmput()` if it reaches 0
    - `__mmput()` calls `mmdrop()`, that decrements the `count` field, and calls `__mm_drop()` if it reaches 0
    - `__mmdrop()` calls `free_mm()` which return the memory for the `mm_struct()` to the slab cache (i.e. free)

Address space and memory descriptor

Memory descriptor and kernel threads

- Kernel threads do not have a user-space address space
  - `mm` field of a kernel thread `task_struct` is `NULL`
- However they 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`
  - OK because **kernel part is the same in all address spaces**

Virginia Tech

Outline

Virtual Memory Area

vm_area_srtuct

- Each VMA is represented by an object of type **vm_area_struct**
  - Defined in `include/linux/mm_types.h`
  - Interesting fields:

```
1  struct vm_area_struct {
2    struct i                    mm_struct *vm_mm; /* associated address space (mm_struct) */
3    unsigned long               vm_start;         /* VMA start, inclusive */
4    unsigned long               vm_end;           /* VMA end, exclusive */
5    struct vm_area_struct       *vm_next;         /* list of VMAs */
6    struct vm_area_struct       *vm_prev;         /* list of VMAs */
7    pgprot_t                    vm_page_prot;     /* access permissions */
8    unsigned long               vm_flags;         /* flags */
9    struct rb_node              vm_rb;            /* VMA node in the tree */
10   struct list_head            anon_vma_chain;   /* list of anonymous mappings */
11   struct anon_vma             *anon_vma;        /* anonmous vma object */
12   struct vm_operation_struct  *vm_ops;          /* operations */
13   unsigned long               vm_pgoff;         /* offset within file */
14   struct file                 *vm_file;         /* mapped file (can be NULL) */
15   void                        *vm_private_data; /* private data */
16   /* ... */
17 }
```

Virtual Memory Area

vm_area_srtuct (2)

- The VMA exists over [vm_start, vm_end[in the corresponding address space
  - Address space is pointed by the vm_mm field (of type mm_struct)
- 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

Virtual Memory Area

Flags

- **Flags** specify properties and information for all the pages contained in the VMA

- `VM_READ`: pages can be read from
- `VM_WRITE`: pages can be written to
- `VM_EXEC`: code inside pages can be executed
- `VM_SHARED`: pages are shared between multiple processes (if unset the mapping is *private*)
- `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_SHARED` flag can be set
- `VM_GROWSDOWN`: area can grow downwards
- `VM_GROWSUP`: area can grow upwards
- `VM_SHM`: area can be used for shared memory
- `VM_DENYWRITE`: area maps an unwritable file
- `VM_EXECUTABLE`: area maps an executable file

Virtual Memory Area

Flags (2)

- `VM_LOCKED`: the area pages are locked (will not be swapped-out)
- `VM_IO`: the area maps a device IO space
- `VM_SEQ_READ`: pages in the area seem to be accessed sequentially
- `VM_RAND_READ`: pages seem to be accessed randomly
- `VM_DONTCOPY`: area will not be copied upon `fork()`
- `VM_DONTEXPAND`: area cannot grow through `mremap()`
- `VM_ACCOUNT`: area is an accounted VM object
- `VM_HUGETLB`: area uses `hugetlb` pages

Virginia Tech

Virtual Memory Area

Flags (3)

- Combining `VM_READ`, `VM_WRITE` and `VM_EXEC` gives the permissions 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()` system call
  - Instructs the file pre-fetching algorithm *read-ahead* to increase or decrease its agressivity
- `VM_HUGETLB` indicates that the area uses pages larger than the regular size
  - 2M and 1G on x86
  - Smaller page table $\rightarrow$ good for the TLB
  - Less levels of page tables $\rightarrow$ faster address translation

Virtual Memory Area

VMA operations

- `vm_ops` in `vm_area_struct` points to a **`vm_operations_struct`** object
  - Contains function pointers to operate on a specific VMAs
  - Defined in `include/linux/mm.h`

```
1 struct vm_operations_struct {
2   void (*open)(struct vm_area_struct * area);
3   void (*close)(struct vm_area_struct * area);
4   int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
5   int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
6   /* ... */
7 }
```

Virtual Memory Area

VMA operations (2)

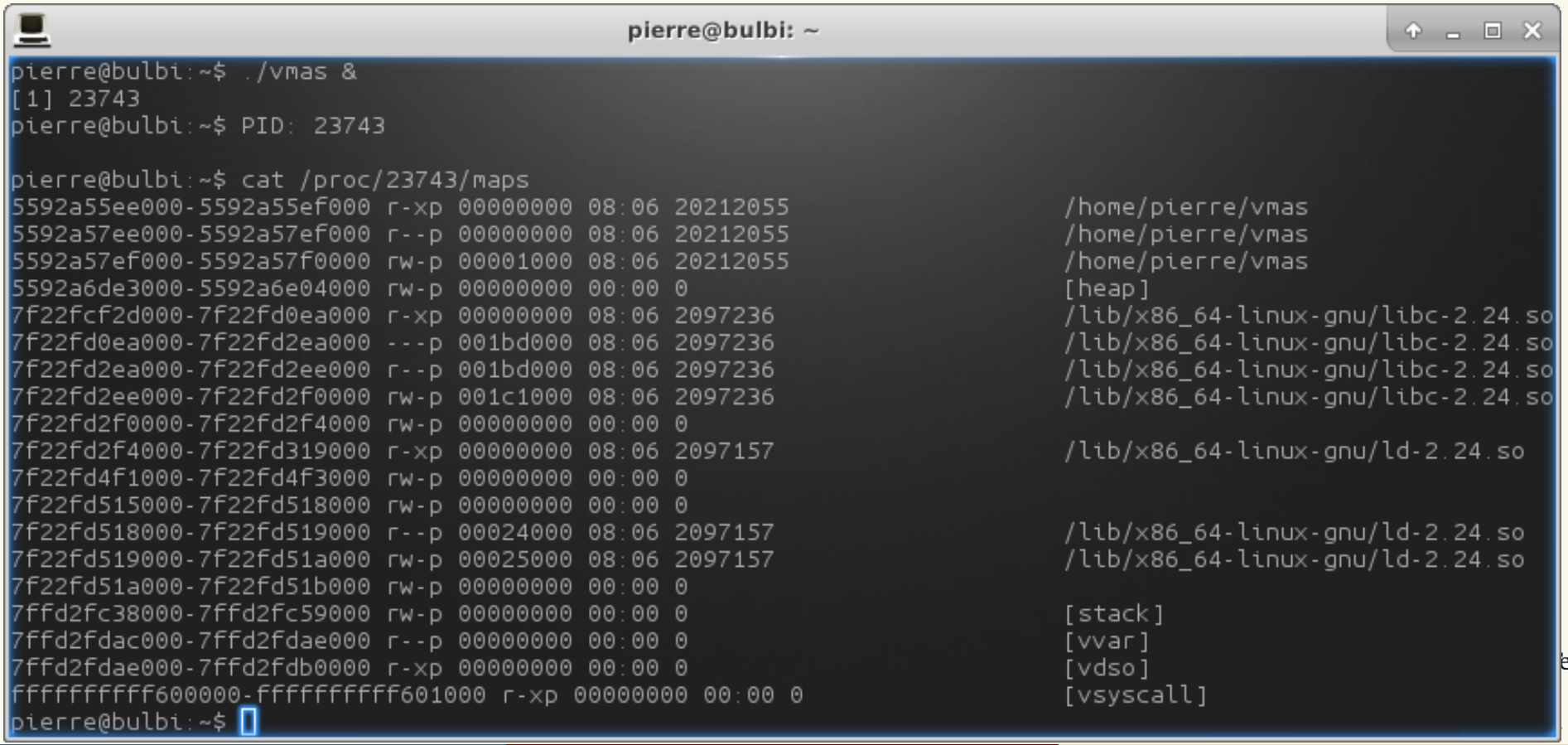
- Function pointers in `vm_operations_struct`:
  - `open()`: called when the area is added to an address space
  - `close()`: called when the area is removed from an address space
  - `fault()`: invoked by the page fault handler when a page that is not present in physical memory is accessed
  - `page_mkwrite()`: invoked by the page fault handler when a previously read-only page is made writable
  - Description of all operations in `include/linux/mm.h`

Virginia Tech

Virtual Memory Area

VMAs in real life

- From userspace, one can observe the VMAs map for a given process:
  - `cat /proc/<pid>/maps`



Virtual Memory Area

VMAs in real life (2)

- `/proc/<pid>/maps` columns description:
  1. Address range
  2. Permissions
  3. Start offset of file mapping
  4. Device containing the mapped file
  5. Mapped file inode number
  6. Mapped file pathname
- Can also use the command `pmap <pid>`

Outline

VMA manipulation

Finding a VMA

- **`find_vma()`**: used to find a VMA in which a specific memory address resides
  - Prototype in `include/linux/mm.h`:

```
1 struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr);
```

  - Defined in `mm/mmap.c`:

```
1  struct vm_area_struct *find_vma(struct
       mm_struct *mm, unsigned long addr)
2  {
3    struct rb_node *rb_node;
4    struct vm_area_struct *vma;
5
6    /* Check the cache first. */
7    vma = vmacache_find(mm, addr);
8    if (likely(vma))
9      return vma;
10
11   rb_node = mm->mm_rb.rb_node;
12
13   while (rb_node) {
14     struct vm_area_struct *tmp;
15
16     tmp = rb_entry(rb_node, struct
         vm_area_struct, vm_rb);
17
18     if (tmp->vm_end > addr) {
19       vma = tmp;
20       if (tmp->vm_start <= addr)
21         break;
22       rb_node = rb_node->rb_left;
23     } else
24       rb_node = rb_node->rb_right;
25   }
26
27   if (vma)
28     vmacache_update(addr, vma);
29   return vma;
30 }
```

VMA manipulation

Finding a VMA (2)

- `find_vma_prev()`: returns in addition the last VMA before a given address
  - `include/linux/mm.h`:

```
1 struct vm_area_struct *find_vma_prev(struct mm_struct *mm, unsigned long addr,
      struct vm_area_struct **pprev);
```

- `find_vma_intersection()`: returns the first VMA overlapping a given address range
  - `include/linux/mm.h`:

```
1 static inline struct vm_area_struct * find_vma_intersection(struct mm_struct *
      mm, unsigned long start_addr, unsigned long end_addr)
2 {
3   struct vm_area_struct * vma = find_vma(mm,start_addr);
4
5   if (vma && end_addr <= vma->vm_start)
6     vma = NULL;
7   return vma;
8 }
```

VMA manipulation

Creating 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
  - `include/linux/mm.h`:

```
1 extern unsigned long do_mmap(struct file *file, unsigned long addr,
2   unsigned long len, unsigned long prot, unsigned long flags,
3   vm_flags_t vm_flags, unsigned long pgoff, unsigned long *populate);
```

- Caller must hold `mm->mmap_sem` (RW semaphore)
- Maps the file `file` in the address space at address `addr` for length `len`. Mapping starts at offset `pgoff` in the file
- `prot` specifies access permissions for the memory pages:
  - `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`

Virginia Tech

VMA manipulation

Creating an address interval (2)

- `flags` specifies the rest of the VMA options:

- `MAP_SHARED`: mapping can be shared
- `MAP_PRIVATE`: mapping cannot be shared
- `MAP_FIXED`: created interval *must* start at `addr`
- `MAP_ANONYMOUS`: mapping is not file-backed
- `MAP_GROWSDOWN`: corresponds to `VM_GROWSDOWN`
- `MAP_DENYWRITE`: corresponds to `VM_DENYWRITE`
- `MAP_EXECUTABLE`: corresponds to `VM_EXECUTABLE`
- `MAP_LOCKED`: corresponds to `VM_LOCKED`
- `MAP_NORESERVE`: no space reserved for the mapping
- `MAP_POPULATE`: populate (default) page tables
- `MAP_NONBLOCK`: do not block on IO

Virginia Tech

VMA manipulation

Creating an address interval (3)

- 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()`

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

VMA manipulation

Removing an address interval

- Removing an address interval is done through `do_munmap()`
  - `include/linux/mm.h`:

```
1 int do_munmap(struct mm_struct *, unsigned long, size_t);
```

  - 0 returned on success
- Exported to user-space through `munmap()`:

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

Outline

Page tables

Presentation

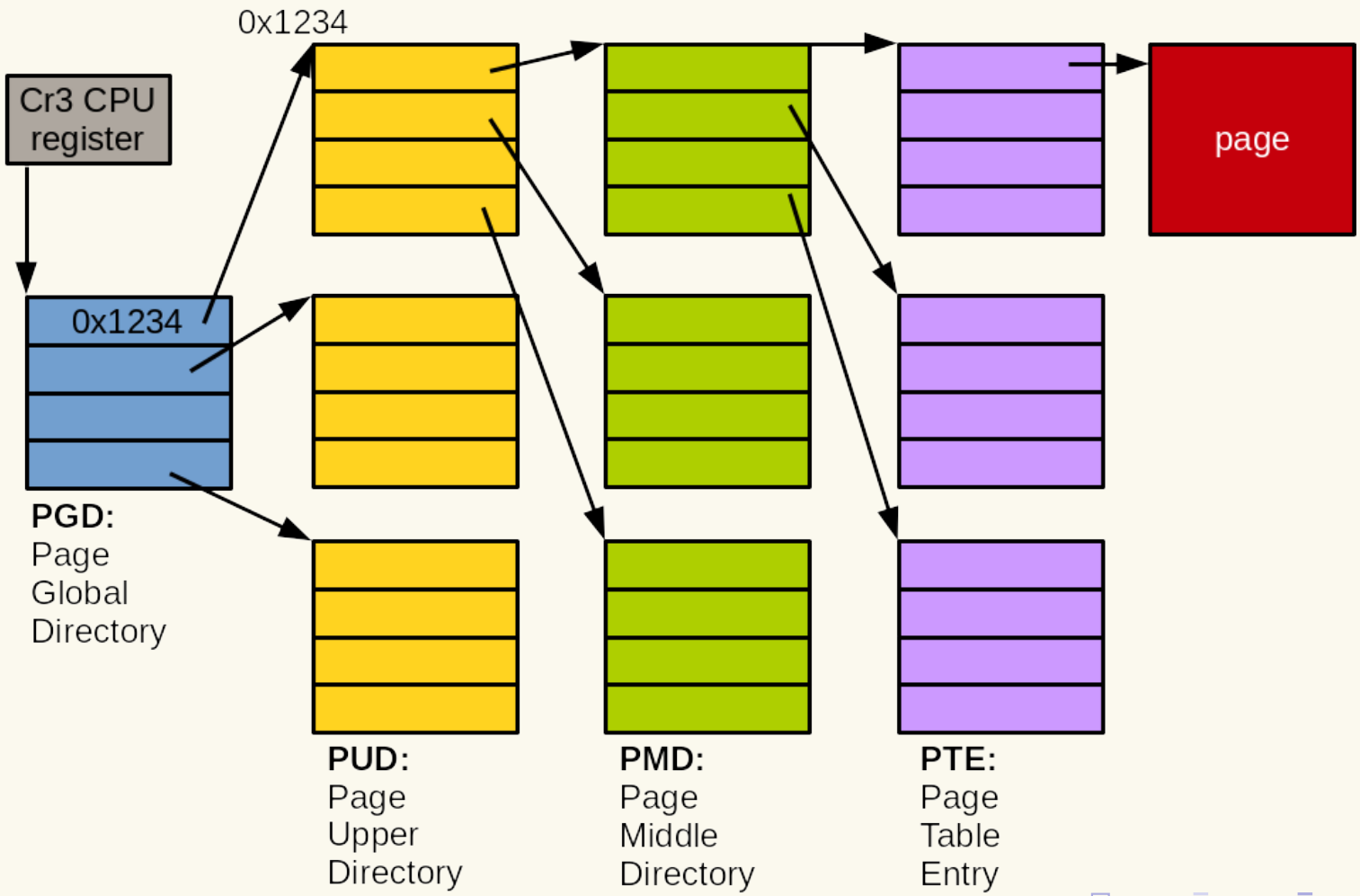
- Linux enables **paging** early in the boot process
  - **All memory accesses made by the CPU are virtual and translated to physical addresses through the *page tables***
  - Linux set the page tables and the translation is made automatically by the hardware (MMU) according to the page tables content
- The address space is defined by VMAs and is sparsely populated
  - One address space per process $\rightarrow$ one page table per process
  - Lots of ”empty” areas
  - **Defining the page table as a single static array is a huge waste of space**
    - A hierarchical tree structure is used

Virginia Tech

Page tables

Page table setup (2)

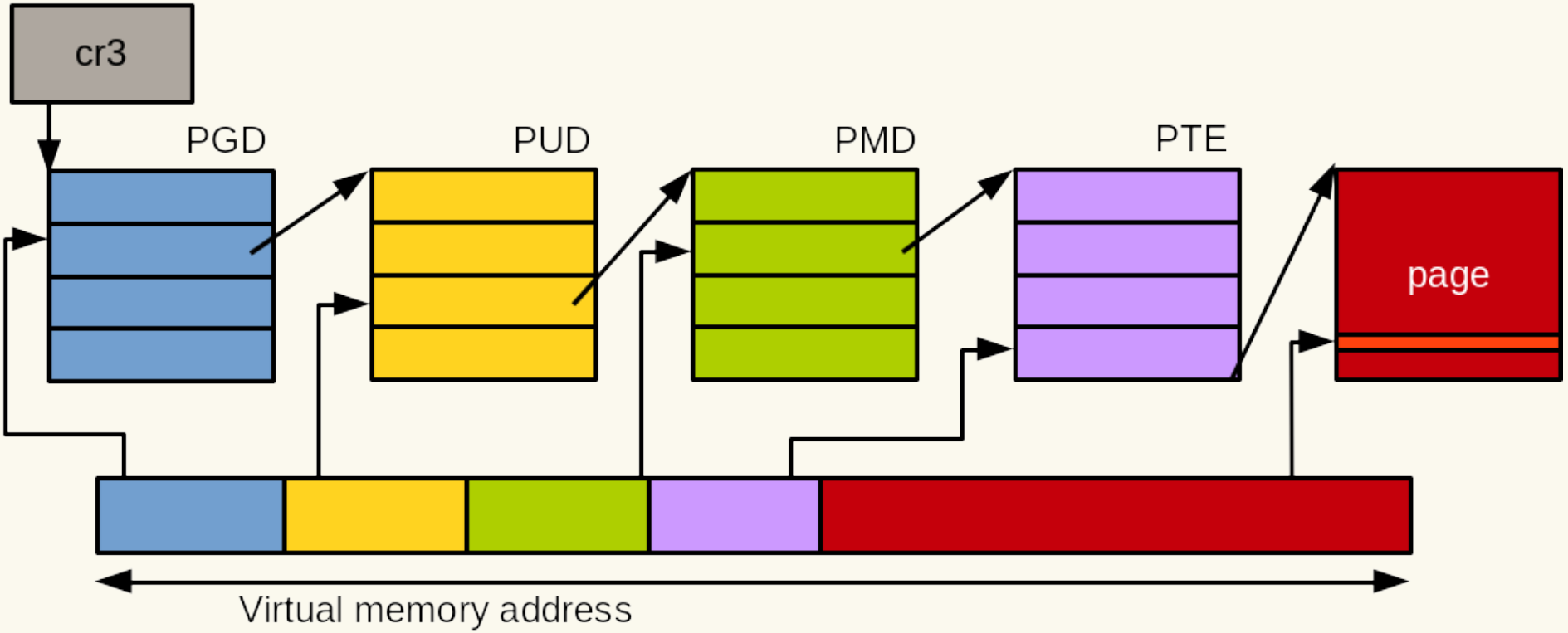
- **Setting up the page table is performed by the kernel**



Page tables

Address translation

- **Address translation is performed by the hardware**



- More info on page tables and memory management: [2, 1]

Virginia Tech

Bibliography I

[1] Four-level page tables.
`https://lwn.net/Articles/106177/`.
Accessed: 2017-03-25.

[2] GORMAN, M.
Understanding the Linux virtual memory manager.
Prentice Hall Upper Saddle River, 2004.
Accessed: 2017-03-25.

Virginia Tech