Process Address Space

Changwoo Min
Summary of last lectures

• Tools: building, exploring, and debugging Linux kernel
• Core kernel infrastructure
  • syscall, module, kernel data structures
• Process management & scheduling
• Interrupt & interrupt handler
• Kernel synchronization
• Memory management
Today: process address space

- Address space
- Memory descriptor: `mm_struct`
- Virtual Memory Area (VMA)
- VMA manipulation
- Page tables
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 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

![Diagram of memory address translation](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
55fe3db4a000-55fe3ddf0d00 rw-p 00000000 00:00 0  [heap]
7f7522769000-7f75227d9000 rw-p 00000000 00:00 0
7f75227e9000-7f75227f9000 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
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

Virtual Memory Area represented by a vm_area_struct object

Address space defined by a mm_struct object
Memory descriptor: **mm_struct**

- Address space in Linux kernel: **struct mm_struct**

```c
/* 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 start_stack;         /* start address of stack */
    /* ... */
```
Memory descriptor: `mm_struct`

```c
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
Memory descriptor: **mm_struct**

- **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 in an ascending order
  - 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**
 Allocating a memory descriptor

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

```c
/* 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()` 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
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()`

```c
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);
}
void __mmdrop(struct mm_struct *mm) {
    /* ... */
    free_mm(mm);
}
```
Kernel threads do not have a user-space address space

- \texttt{mm} field of a kernel thread \texttt{task_struct} is \texttt{NULL}
The `mm_struct` and kernel threads

User space (ring 3)
- vim
- gcc
- firefox
- ...

Kernel space (ring 0)
- Operating system kernel
- System call interface

Hardware
- CPU
- memory
- disk
- ...

- Process: virtual memory
- Exception-based control transfer: syscall, sysret
- Protocol: calling convention
- Preemptive scheduling
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
Review: Segmentation in x86

Figure 3-1. Segmentation and Paging
Review: Privilege levels of a segment

- CPL (current privilege level)
  - the privilege level of currently executing program
  - bits 0 and 1 in the %cs register

- RPL (requested privilege level)
  - an override privilege level that is assigned to a segment selector
  - a segment selector is a part (16-bit) of segment registers (e.g., ds, fs), which is an index of a segment descriptor and RPL

- DPL (descriptor privilege level)
  - the privilege level of a segment
Review: How isolation is enforced in x86?

- Access is granted if \( RPL \geq CPL \) and \( DPL \geq CPL \)
Review: How to switch b/w rings (ring 0 ↔ ring 3)?

- Controlled transfer: system call
  - `int`, `sysenter` or `syscall` instruction set CPL to 0; change to KERNEL_CS and KERNEL_DS segments
  - set CPL to 3 before going back to user space; change to USER_CS and USER_DS segments
**mm_struct** vs. **task_struct**

- **Thread**
  - vim
    - task_struct
      - stack
    - mm
      - page table
  - gcc
    - task_struct
      - stack
    - mm
      - page table

- **Process**
  - multi-threaded web server
    - task_struct
      - stack
      - ...
    - mm
      - page table

**Virtual address space**
Virtual Memory Area (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)
```
Virtual Memory Area (VMA)

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

```c
/* 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; /* anonymous 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) */
    *vm_private_data; /* private data */
} /* ... */
```
Virtual Memory Area (VMA)

- The VMA exists over \([\text{vm start}, \text{vm end})\) in the corresponding address space → size in bytes: \(\text{vm_end} - \text{vm_start}\)
- Address space is pointed by the \(\text{vm_mm}\) field (of type \(\text{mm_struct}\))
- Each VMA is unique to the associated \(\text{mm_struct}\)
  - Two processes mapping the same file will have two different \(\text{mm_struct}\) objects, and two different \(\text{vm_area_struct}\) objects
  - Two threads sharing a \(\text{mm_struct}\) object also share the \(\text{vm_area_struct}\) objects
# VMA flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effect on the VMA and Its Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_READ</td>
<td>Pages can be read from.</td>
</tr>
<tr>
<td>VM_WRITE</td>
<td>Pages can be written to.</td>
</tr>
<tr>
<td>VM_EXEC</td>
<td>Pages can be executed.</td>
</tr>
<tr>
<td>VM_SHARED</td>
<td>Pages are shared.</td>
</tr>
<tr>
<td>VM_MAYREAD</td>
<td>The VM_READ flag can be set.</td>
</tr>
<tr>
<td>VM_MAYWRITE</td>
<td>The VM_WRITE flag can be set.</td>
</tr>
<tr>
<td>VM_MAYEXEC</td>
<td>The VM_EXEC flag can be set.</td>
</tr>
<tr>
<td>VM_MAYSHARE</td>
<td>The VM_SHARE flag can be set.</td>
</tr>
</tbody>
</table>
# VMA flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effect on the VMA and Its Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_GROWSDOWN</td>
<td>The area can grow downward.</td>
</tr>
<tr>
<td>VM_GROWSUP</td>
<td>The area can grow upward.</td>
</tr>
<tr>
<td>VM_SHM</td>
<td>The area is used for shared memory.</td>
</tr>
<tr>
<td>VM_DENYWRITE</td>
<td>The area maps an unwritable file.</td>
</tr>
<tr>
<td>VM_EXECUTABLE</td>
<td>The area maps an executable file.</td>
</tr>
<tr>
<td>VM_LOCKED</td>
<td>The pages in this area are locked.</td>
</tr>
<tr>
<td>VM_IO</td>
<td>The area maps a device’s I/O space.</td>
</tr>
<tr>
<td>VM_SEQ_READ</td>
<td>The pages seem to be accessed sequentially.</td>
</tr>
</tbody>
</table>
# VMA flags

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

- 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 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

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

        rb_node = rb_next(rb_node);
    }

    return NULL;
}
VMA manipulation: `find_vma()`

```c
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;
```
VMA manipulation

/* 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 VMAs
  - Or a merge of the create area with an adjacent one when they have the same permissions

```c
/*
 * 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);
```
Creating an address interval

- `prot` specifies access permissions for the memory pages

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effect on the new interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROT_READ</td>
<td>Corresponds to VM_READ</td>
</tr>
<tr>
<td>PROT_WRITE</td>
<td>Corresponds to VM_WRITE</td>
</tr>
<tr>
<td>PROT_EXEC</td>
<td>Corresponds to VM_EXEC</td>
</tr>
<tr>
<td>PROT_NONE</td>
<td>Cannot access page</td>
</tr>
</tbody>
</table>
Creating an address interval

- **flags** specifies the rest of the VMA options

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effect on the new interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP_SHARED</td>
<td>The mapping can be shared.</td>
</tr>
<tr>
<td>MAP_PRIVATE</td>
<td>The mapping cannot be shared.</td>
</tr>
<tr>
<td>MAP_FIXED</td>
<td>The new interval must start at the given address addr.</td>
</tr>
<tr>
<td>MAP_ANONYMOUS</td>
<td>The mapping is not file-backed, but is anonymous.</td>
</tr>
<tr>
<td>MAP_GROWSDOWN</td>
<td>Corresponds to <code>VM_GROWSDOWN</code>.</td>
</tr>
</tbody>
</table>
Creating an address interval

- **flags** specifies the rest of the VMA options

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effect on the new interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP_DENYWRITE</td>
<td>Corresponds to <a href="#">VM_DENYWRITE</a></td>
</tr>
<tr>
<td>MAP_EXECUTABLE</td>
<td>Corresponds to <a href="#">VM_EXECUTABLE</a></td>
</tr>
<tr>
<td>MAP_LOCKED</td>
<td>Corresponds to <a href="#">VM_LOCKED</a></td>
</tr>
<tr>
<td>MAP_NORESERVE</td>
<td>No need to reserve space for the mapping.</td>
</tr>
<tr>
<td>MAP_POPULATE</td>
<td>Populate (prefault) page tables.</td>
</tr>
<tr>
<td>MAP_NONBLOCK</td>
<td>Do not block on I/O.</td>
</tr>
</tbody>
</table>
Creating 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()`

```c
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()

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

- Exported to user-space through munmap()

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

/* 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 - 000007fffffffffff (=47 bits) user space, different per mm
hole caused by [47:63] sign extension
ffff800000000000 - fffff87fffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7fffffffffff (=64 TB)  **direct mapping of all phys. memory**
ffffc800000000000 - ffffc8fffffffffff (=40 bits) hole
ffffc900000000000 - ffffef8fffffffffff (=45 bits) vmalloc/ioemap space
ffffe900000000000 - ffffe9fffffffffff (=40 bits) hole
ffffea000000000000 - ffffeafffffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec000000000000 - ffffffff8fffffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...

vaddr_end for KASLR
fffffe000000000000 - ffffe7fffffffffff (=39 bits) cpu_entry_area mapping
fffffe800000000000 - ffffffffefffffffff (=39 bits) LDT remap for PTI
ffffffff000000000000 - ffffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
fffffeef000000000000 - ffffffffefffffffff (=64 GB)  EFI region mapping space
... unused hole ...
ffffffff800000000000 - ffffffff9fffffffff (=512 MB) kernel text mapping, from phys 0
fffffffffa0000000000 - ffffffffeffffffffff (=1520 MB) module mapping space
[fixmap start] - ffffffff65fffff kernel-internal fixmap range
ffffffff600000000000 - ffffffff6000000 (=4 kB) legacy vsyscall ABI
fffffffff000000000000 - ffffffff7fffffffff (=2 MB) unused hole
Further readings

- Introduction to Memory Management in Linux
- 20 years of Linux virtual memory
- Linux Kernel Virtual Memory Map
- Kernel page-table isolation
- Addressing Meltdown and Spectre in the kernel
- Meltdown and Spectre
- Meltdown Attack Lab
Further readings

• 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
Next action

• Reading assignment
  
  • [EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs,](#) USENIX ATC 2018
  
  • Due: before next class (Nov 1st)
Next class

- Virtual File System