

Memory Management

User space memory management in Linux

by Aidan Brumsickle

The Basics

- Each process has an associated address space; usually there is one per process, though processes can elect to share their memory (this is how threads work in Linux: multiple processes with the same address space).
- Each address space is split into *memory areas*, which are intervals of memory addresses with associated permissions, ie, read-only, write, executable.
- This helps protect against dangerous bugs and malicious code by limiting where the process can write and what it can execute (ie, prevent self-modifying code)
- But how does Linux do it?

The Memory Descriptor

- struct mm_struct, found in include/linux/mm_types.h
- link to it in task_struct
- set up in fork()
- kernel threads do not have a mm_struct, since they don't have a process address space.
- Let's take a look.

Virtual Memory Areas

- `vm_area_struct`, defined in `include/linux/mm_types.h`
- it's a linked list
- keeps track of permissions
- VMA flags... some interesting ones:

`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_SHM`

Area is used for shared memory

`VM_IO`

Area maps to a device's I/O space

`VM_DONTCOPY`

Area mustn't be copied on `fork()`

`VM_SEQ_READ`

Pages seem to be accessed sequentially

`VM_RAND_READ`

Pages seem to be accessed randomly.

- Let's look...

An Example of Memory Areas

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, World!\n");
    return 0;
}
```

There's the stack, the text section, the data section, the bss, as well as sections for the dynamically linked C library and the dynamic linker.

Memory Area Functions

The kernel needs to be able to manipulate and find information about different memory areas, such as their locations, and whether a given address in an address space maps to a memory area.

Useful functions:

- `find_vma()`
- `find_vma_prev()`
- `find_vma_intersection()`

Let's look at the code a bit.

do_mmap()

do_mmap() is used by the kernel to create a new linear address interval:

```
unsigned long do_map(struct file *file,  
unsigned long addr,  
                unsigned long len,  
unsigned long prot,  
                unsigned long flag,  
unsigned long offset)
```

That's a lot of parameters... what's the file for?

- you can optionally map a file starting at offset: file-backed mapping vs anonymous mapping
- prot = protection flags (read, write, execute)
- begin looking for a free interval at addr (optional)
- flags correspond to VMA flags
- actually, it's a wrapper function for do_mmap_pgoff()...

do_munmap()

quite simply, there's a corresponding function and system call to remove an address interval.

Page Tables in Linux

- 3 levels: PGD, pgd_t, PMD, pmd_t, PTE, pte_t
- quite architecture dependent... so defined in `<asm/page.h>`
- let's look at x86 implementation
- our good friend the translation lookaside buffer!

My change: logging maps and unmaps

I wanted to do a log of how many maps were created and destroyed.

I edited mm/mmap.c, adding 2 global variables and modifying do_mmap_pgoff() and do_munmap().

I don't have the unmap function decrement what the map function increments, since it isn't really a LIFO structure at all with respect to time.

...the code, the code!

...and let's see if it works...