

Memory mapped register access

J.-M Friedt, November 29, 2025

As all kernel module not accessing hardware, the module we will develop is portable on all Linux systems, whether running on the PC or the embedded target. However, we will be manipulating memory, leading to a risk of memory corruption and kernel panic, freezing the system. It is strongly advised to test the module under development on the embedded board to avoid the lengthy reboot of the PC.

We have learned how to create a character device with a pseudo-file located in `/dev` linking through its major number with a kernel module. We have seen that the character device makes a peripheral appear like a file, following the philosophy of Unix.

1. What additional system call had to be introduced that led to a departure from the initial Unix philosophy that all peripherals are handled as a file? Why was this additional system call created? Justify.

We have seen that `read()` and `write()` system calls require copying information from kernel space to user space, otherwise the memory management unit will detect memory leak between both memory spaces and generate an error. However, by manipulating virtual memory page flags, it is possible to map two different virtual addresses, one in kernel memory and one in user memory, towards the same physical location, hence sharing the same resource, whether the content of RAM or some physical register for accessing hardware.

We hence wish to add one more feature to this pseudo-file, namely the ability to map kernel space memory to userspace. We have seen that the manipulation of the MMU is possible through the `mmap()` system call from userspace to the `/dev/mem` MMU driver. We here wish to implement the `mmap()` handling in our kernel module.

2. in which file of the kernel source tree can you find the prototype of the `mmap` system call handling function? You might get some inspiration from the other system calls we have already implemented (e.g. `release`). What is the prototype of the kernel function handling the `mmap()` system call in the kernel module?
3. create the necessary file in `/dev` so that a userspace program can communicate with the kernel module, and update the kernel module found at https://github.com/jmfriedt/memory_mapped_linux with the necessary system call handling function to handle `mmap()`. How (command) did you fetch the GIT repository in order to keep track of changes and version future updates?
4. write a userspace C program for accessing the kernel module `mmap` method, and demonstrate that indeed the system call is correctly requested. What is the prototype of the userspace `mmap()` system call? Demonstrate that the `mmap()` handling function in the kernel module is indeed called (e.g. with a `printk()` display). How do you read the output messages of the kernel?
5. Handling the virtual memory page requests is not trivial and requires registering a new set of handling functions, similar to the `file_operations` of a character device. The name of this new structure is `vm_operations_struct`: what are the fields in this structure? In which file of the kernel source tree can you find the information?
6. [1] teaches that for a simple driver, `vm_operations_struct` is not needed since *“a simple driver such as `simple` need not do any extra processing in particular. So we have created `open` and `close` methods, which print a message to the system log informing the world that they have been called. Not particularly useful, but it does allow us to show how these methods can be provided, and see when they are invoked.”*. We hence only allocate a buffer in the `open()` system call

```
static int open(struct inode *inode, struct file *filp)
{char *c;
  c=(char *)get_zeroed_page(GFP_KERNEL);
  sprintf(c,"Hello World\n");
  filp->private_data=c;
  ...
}
```

which is remapped to userspace [2] within `mmap()` system call according to

```
char *c;
c=filp->private_data;
unsigned long pfn = virt_to_phys((void *)c)>>PAGE_SHIFT;
int ret=remap_pfn_range(vma, vma->vm_start, pfn, len, vma->vm_page_prot);
}
```

which requires constants defined in `#include <linux/mm.h>`. To make sure we release the allocated memory, we can also add in the system call associated to the `close()` system call:

```
c = filp->private_data;
free_page((unsigned long)c);
filp->private_data = NULL;
```

After adding these functions to allow kernel physical address space to be shared with a userspace program without generating an MMU error, demonstrate that reading the content of the memory pointed by the address returned by the `mmap()` system call in userspace allows to read the content generated in kernel space. What is the returned message?

References

- [1] *Chapter 15: Memory Mapping and DMA* in J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers* (2005) at <https://static.lwn.net/images/pdf/LDD3/ch15.pdf>
- [2] *Memory mapping* at https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html

Solution

1. the `ioctl()` system call was added to configure peripherals and broke the “all is file” philosophy since it would not be applicable to a file.
2. searching recursively in the kernel source tree:

```
cd /home/jmfriedt/buildroot-2024.05.2_redpit/output/build/linux-xilinx-v2024.1/include/  
grep -r file_operations *
```

```
replies linux/fs.h:struct file_operations { which includes int (*mmap) (struct file *,  
struct vm_area_struct *); on line 1865
```

3. we create the character device

```
mknod /dev/whatever c 91 0
```

and we add to the system call to the file operations list:

```
static const struct file_operations fops = { .mmap = mmap, ...
```

with

```
static int mmap(struct file *filp, struct vm_area_struct *vma)  
{printk(KERN_INFO "mmap\n");}
```

4. the program

```
int main() {  
    int k,fd = open("/dev/jmf", O_RDWR);  
    int* addr=mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);  
}
```

compiled as a userspace program with `arm-linux-gcc mymmap_userspace.c` after making sure the Buildroot compiler found in `output/host/usr/bin` is in the path, leading to a message displayed with `dmesg` proving the `mmap()` system called was requested.

5. as before, recursive search in the kernel source tree

```
cd /home/jmfriedt/buildroot-2024.05.2_redpit/output/build/linux-xilinx-v2024.1/include/  
grep -r vm_operations_struct *
```

```
returns linux/mm.h:struct vm_operations_struct { where we find open, close, may_split,  
mremap, mprotect, fault, huge_fault, map_pages, pagesize, page_mkwrite, pfn_mkwrite, access,  
and name
```

6. executing in userspace

```
#include <fcntl.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <stdint.h>  
#include <sys/mman.h> // mmap  
#include <unistd.h> // sysconf
```

```
#define BUFFER_SIZE 1024
```

```

int main(int argc, char **argv)
{int fd;
  long page_size;
  char *addr;
  char buf[BUFFER_SIZE];
  uintptr_t paddr;
  if (argc < 2) {printf("Usage: %s <mmap_file>\n", argv[0]);return EXIT_FAILURE;}
  page_size = sysconf(_SC_PAGE_SIZE);
  fd = open(argv[1], O_RDWR | O_SYNC); if (fd < 0) perror("open");
  addr=mmap(NULL, page_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if (addr==MAP_FAILED) perror("mmap");
  printf("%s\n",addr); // print the content of memory
  if (munmap(addr, page_size)) perror("munmap");
  close(fd);
  return EXIT_SUCCESS;
}

```

returns Hello World as was defined in the kernel module.

Additional useful resources are found at <https://stackoverflow.com/questions/10760479/how-to-mmap-a-linux-kernel-buffer-to-user-space> and <https://stackoverflow.com/questions/8788289/how-remap-pfn-range-remaps-kernel-memory-to-user-space>