/dev/mem

/dev/kmem

/dev/null

/dev/zero

/dev/port

/dev/full

/dev/random

/dev/urandom

/dev/kmsg

/dev/mem --- physical memory, read / write

/dev/kmem --- the kernel virtual memory rather than physical memory is accessed

/dev/full --- Writes to the /dev/full device will fail with an ENOSPC error.

This can be used to test how a program handles disk-full errors.

Reads from the /dev/full device will return \0 characters.

Seeks on /dev/full will always succeed.

/dev/port --- port is similar to mem, but the I/O ports are accessed.

The difference between /dev/random and /dev/urandom

When the entropy pool is empty, reads from /dev/random will block until

additional environmental noise is gathered.

A read from the /dev/urandom device will not block waiting for more entropy.

---

in drivers/char/mem.c

```
static const struct memdev {
        const char *name;
        umode_t mode;
        const struct file_operations *fops;
        struct backing_dev_info *dev_info;
} devlist[] = {
         [1] = { "mem", 0, &mem_fops, &directly_mappable_cdev_bdi },
#ifdef CONFIG_DEVKMEM
         [2] = { "kmem", 0, &kmem_fops, &directly_mappable_cdev_bdi },
#endif
         [3] = { "null", 0666, &null_fops, NULL },
#ifdef CONFIG_DEVPORT
         [4] = { "port", 0, &port_fops, NULL },
#endif
         [5] = { "zero", 0666, &zero_fops, &zero_bdi },
         [7] = { "full", 0666, &full_fops, NULL },
         [8] = { "random", 0666, &random_fops, NULL },
         [9] = { "urandom", 0666, &urandom_fops, NULL },
#ifdef CONFIG_PRINTK
        [11] = { "kmsg", 0644, &kmsg_fops, NULL },
#endif
};

......

static int __init chr_dev_init(void)
{
        int minor;
        int err;

        err = bdi_init(&zero_bdi);
        if (err)
                return err;

        if (register_chrdev(MEM_MAJOR, "mem", &memory_fops))
                printk("unable to get major %d for memory devs\n", MEM_MAJOR);

        mem_class = class_create(THIS_MODULE, "mem");
        if (IS_ERR(mem_class))
                return PTR_ERR(mem_class);

        mem_class->devnode = mem_devnode;
        for (minor = 1; minor < ARRAY_SIZE(devlist); minor++) {
                if (!devlist[minor].name)
                        continue;

                /*
                 * Create /dev/port?
                 */
                if ((minor == DEVPORT_MINOR) && !arch_has_dev_port())
                        continue;

                device_create(mem_class, NULL, MKDEV(MEM_MAJOR, minor),
```

```
                                NULL, devlist[minor].name);
        }

        return tty_init();
}
```

/dev/mem

"/dev/mem" 对应的file_operations is

```
static const struct file_operations mem_fops = {
        .llseek         = memory_lseek,
        .read           = read_mem,
        .write          = write_mem,
        .mmap           = mmap_mem,
        .open           = open_mem,
        .get_unmapped_area = get_unmapped_area_mem,
};
```

```c
/*
 * This funcion reads the *physical* memory. The f_pos points directly to the
 * memory location.
 */
static ssize_t read_mem(struct file *file, char __user *buf,
                        size_t count, loff_t *ppos)
{
        phys_addr_t p = *ppos;
        ssize_t read, sz;
        char *ptr;

        if (p != *ppos)
                return 0;

        if (!valid_phys_addr_range(p, count))       ①
                return -EFAULT;
        read = 0;
#ifdef __ARCH_HAS_NO_PAGE_ZERO_MAPPED
        /* we don't have page 0 mapped on sparc and m68k.. */
        if (p < PAGE_SIZE) {
                sz = size_inside_page(p, count);
                if (sz > 0) {
                        if (clear_user(buf, sz))
                                return -EFAULT;
                        buf += sz;
                        p += sz;
                        count -= sz;
                        read += sz;
                }
        }
#endif

        while (count > 0) {
                unsigned long remaining;

                sz = size_inside_page(p, count);

                if (!range_is_allowed(p >> PAGE_SHIFT, count))
                        return -EPERM;

                /*
                 * On ia64 if a page has been mapped somewhere as uncached, then
                 * it must also be accessed uncached by the kernel or data
                 * corruption may occur.
                 */
                ptr = xlate_dev_mem_ptr(p);     ②
                if (!ptr)
                        return -EFAULT;

                remaining = copy_to_user(buf, ptr, sz);     ③
                unxlate_dev_mem_ptr(p, ptr);
                if (remaining)
                        return -EFAULT;
```

```
                buf += sz;
                p += sz;
                count -= sz;
                read += sz;
        }

        *ppos += read;
        return read;
}
```

①

check 读的区域是否在physical memory之内

```
static inline int valid_phys_addr_range(phys_addr_t addr, size_t count)
{
        return addr + count <= __pa(high_memory);
}
```

high_memory是memory的最高边界的virtual address

②

in arch/arm/include/asm/io.h

```
/*
 * Convert a physical pointer to a virtual kernel pointer for /dev/mem
 * access
 */
#define xlate_dev_mem_ptr(p)    __va(p)
```

p是/dev/mem file offset ==> physical address

转变成virtual address

③

ptr is virtual address

/dev/kmem

"/dev/kmem" 对应的file_operations is

```
static const struct file_operations kmem_fops = {
        .llseek         = memory_lseek,
        .read           = read_kmem,
        .write          = write_kmem,
        .mmap           = mmap_kmem,
        .open           = open_kmem,
        .get_unmapped_area = get_unmapped_area_mem,
};
```

/dev/kmem对应的文件是kernel看到的address space。大致如下

vmalloc : 0xf0000000 - 0xff000000 （240 MB）

lowmem ：0xc0000000 - 0xef800000 （760 MB）

pkmap ：0xbfe00000 - 0xc0000000 （ 2 MB）

modules : 0xbf000000 - 0xbfe00000 （ 14 MB）

.text : 0xc0008000 - 0xc05a6d14 (5756 kB)

.init : 0xc05a7000 - 0xc05da000 (204 kB)

.data : 0xc05da000 - 0xc06098c0 (191 kB)

.bss : 0xc06098c0 - 0xc0679c74 (449 kB)

即kernel看到的space是从0xc000,0000(virtual address，对应physical address 0)开始，到high_memory，是对应到physical memory的kernel space，在此上还有vmalloc space，这并不象lowmem那样直接对应到physical memory,它的physical memory也来自与0xc000,0000到high_memory。

```
/*
 * This function reads the *virtual* memory as seen by the kernel.
 */
static ssize_t read_kmem(struct file *file, char __user *buf,
                         size_t count, loff_t *ppos)
{
        unsigned long p = *ppos;                ①
        ssize_t low_count, read, sz;
        char *kbuf; /* k-addr because vread() takes vmlist_lock rwlock */
        int err = 0;

        read = 0;
        if (p < (unsigned long) high_memory) {      ②
                low_count = count;
                if (count > (unsigned long)high_memory - p)
                        low_count = (unsigned long)high_memory - p;

#ifdef __ARCH_HAS_NO_PAGE_ZERO_MAPPED
                /* we don't have page 0 mapped on sparc and m68k.. */
                if (p < PAGE_SIZE && low_count > 0) {
                        sz = size_inside_page(p, low_count);
                        if (clear_user(buf, sz))
                                return -EFAULT;
                        buf += sz;
                        p += sz;
                        read += sz;
                        low_count -= sz;
                        count -= sz;
                }
#endif
                while (low_count > 0) {
                        sz = size_inside_page(p, low_count);

                        /*
                         * On ia64 if a page has been mapped somewhere as
                         * uncached, then it must also be accessed uncached
                         * by the kernel or data corruption may occur
                         */
                        kbuf = xlate_dev_kmem_ptr((char *)p);       ③

                        if (copy_to_user(buf, kbuf, sz))            ④
                                return -EFAULT;
                        buf += sz;
                        p += sz;
                        read += sz;
                        low_count -= sz;
                        count -= sz;
                }
        }

        if (count > 0) {                                    ⑤
                kbuf = (char *)__get_free_page(GFP_KERNEL);     ⑥
                if (!kbuf)
```

```
54.                         return -ENOMEM;
55.                 while (count > 0) {
56.                         sz = size_inside_page(p, count);
57.                         if (!is_vmalloc_or_module_addr((void *)p)) {    ⑦
58.                                 err = -ENXIO;
59.                                 break;
60.                         }
61.                         sz = vread(kbuf, (char *)p, sz);            ⑧
62.                         if (!sz)
63.                                 break;
64.                         if (copy_to_user(buf, kbuf, sz)) {          ⑨
65.                                 err = -EFAULT;
66.                                 break;
67.                         }
68.                         count -= sz;
69.                         buf += sz;
70.                         read += sz;
71.                         p += sz;
72.                 }
73.                 free_page((unsigned long)kbuf);
74.         }
75.         *ppos = p;
76.         return read ? read : err;
77. }
```

①

p是/dev/kmem file的offset，它的size就可以超过physical memory size，可以达vmalloc space的边界

②

表示要读取的文件区域的上边界是小于high_memory的，即落在physical memory之内

③

in arch/arm/include/asm/io.h

#define xlate_dev_kmem_ptr(p) p

由于这里p即是kernel address，所以无需转换

④

copy小于high_memory的data

⑤

count > 0

表示读取区域的下边界大于high_memory，即超出了0xc000,0000 + physical memory，也就是

vmalloc区域

⑥

读取vmalloc区域，需要allocate one page 过度一下 (Why???)

⑦

check p的合法性，超过vmalloc区域也是非法的

⑧

读取到过度page中

⑨

复制到用户空间

/dev/null

"/dev/null" 对应的file_operations is

```
static const struct file_operations null_fops = {
        .llseek         = null_lseek,
        .read           = read_null,
        .write          = write_null,
        .aio_read       = aio_read_null,
        .aio_write      = aio_write_null,
        .splice_write   = splice_write_null,
};


static ssize_t read_null(struct file *file, char __user *buf,
                         size_t count, loff_t *ppos)
{
        return 0;
}

static ssize_t write_null(struct file *file, const char __user *buf,
                          size_t count, loff_t *ppos)
{
        return count;
}
```

/dev/full

"/dev/full" 对应的file_operations is

```
static const struct file_operations full_fops = {
        .llseek         = full_lseek,
        .read           = new_sync_read,
        .read_iter      = read_iter_zero,
        .write          = write_full,
};

static ssize_t write_full(struct file *file, const char __user *buf,
                          size_t count, loff_t *ppos)
{
        return -ENOSPC;
}
```

/dev/port

"/dev/port" 对应的file_operations is

```
static const struct file_operations port_fops = {
        .llseek         = memory_lseek,
        .read           = read_port,
        .write          = write_port,
        .open           = open_port,
};
```

???