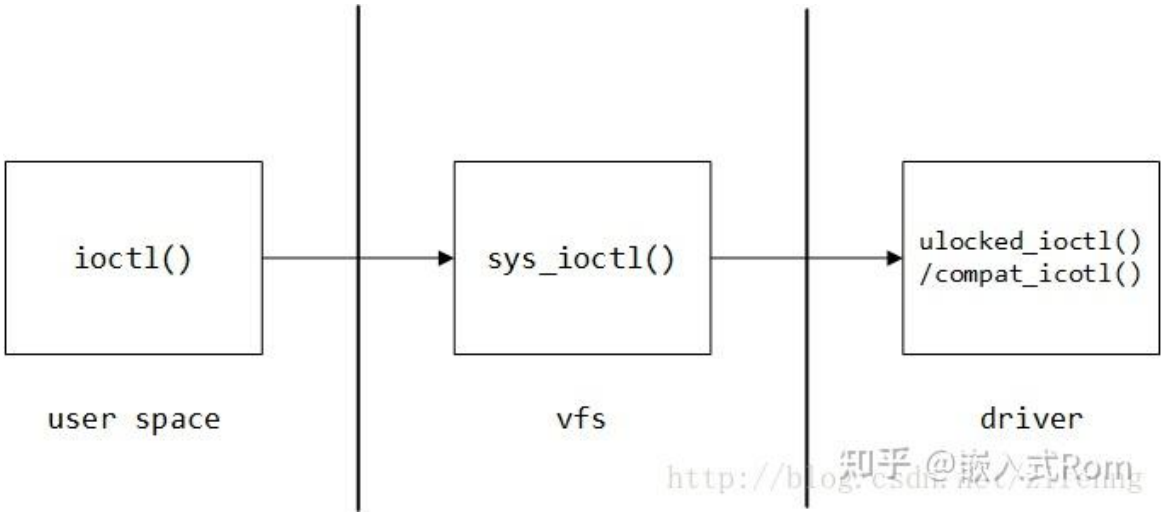
ioctl函数——从用户空间到设备驱动

由 enlai创建, 最后修改于三月 14, 2024

一个字符设备驱动通常会实现常规的打开、关闭、读、写等功能，但在一些细分的情境下，如果需要扩展新的功能，通常以增设ioctl()命令的方式实现，其作用类似于“拾遗补漏”。在文件I/O中，**ioctl**扮演着重要角色，本文将以驱动开发为侧重点，从用户空间到内核空间纵向分析ioctl函数。



`ioctl` 是一个在Unix、Linux和其他POSIX兼容的操作系统中的系统调用，它提供了一种管理I/O设备的通用方法。`ioctl` 代表 "input/output control"。它的原型定义在 `<sys/ioctl.h>` 头文件中，如下所示：

```
int ioctl(int fd, unsigned long request, ...);
```

- **`fd`：文件描述符，代表需要进行操作的设备或文件。**
- **`request`：一个设备特定的请求码，它指示 `ioctl` 要执行的操作。**
- **`...`：这是一个可变参数列表，其参数通常是指向数据结构的指针，这些数据结构包含了执行请求所需的参数或者是用于返回信息的缓冲区。**

`ioctl` 函数可以用于执行以下类型的操作：

1. **设备控制：直接控制硬件设备的操作，例如设置串口的波特率、启用或禁用硬件特性等。**
2. **配置系统或设备参数：设置系统或设备的配置参数，例如网络接口的配置。**
3. **查询系统或设备状态：获取系统或设备的状态信息，例如检查打印机是否有纸。**
4. **文件操作：对于不适合用标准输入输出系统调用的文件操作，如非阻塞I/O或异步I/O。**

在上下文中的UVC设备例子中，`ioctl` 被用来查询视频设备的能力：

```
int ret = ioctl(fd, VIDIOC_QUERYCAP, &cap);
```

这里的参数解释如下：

- **`fd` 是通过调用 `open` 函数获得的UVC设备的文件描述符。**
- **`VIDIOC_QUERYCAP` 是请求码，用于查询设备的功能。它告诉 `ioctl`，我们想要获取 `v4l2_capability` 结构体中定义的设备能力信息。**
- **`&cap` 是一个指向 `v4l2_capability` 结构体的指针，该结构体将被填充设备的能力信息。**

`ioctl` 的返回值通常是 `-1` 表示失败，此时可以通过 `errno` 获取错误信息。如果成功，返回值通常是0或者是其他非负值，具体的含义取决于请求码。

用户空间的ioctl()

```
#include <sys/ioctl.h>
int ioctl(int fd, int cmd, ...) ;
```

参数	描述
fd	文件描述符
cmd	交互协议、设备驱动将根据cmd执行相应操作
。。。	可变参数arg，依赖cmd指令和类型

2024/3/28 17:21

失败则返回-1并设置全局变量errorno值，如下:

```
EBADF d is not a valid descriptor.
EFAULT argp references an inaccessible memory area.
EINVAL Request or argp is not valid.
ENOTTY d is not associated with a character special device.
ENOTTY The specified request does not apply to the kind of object that the descriptor d references.
```

因此，在用户空间使用ioctl时，可以做如下的出错判断以及处理:

```
int ret;
   ret = ioctl(fd, MYCMD);
   if (ret == -1) {
       printf("ioctl: %s\n", strerror(errno));
   }
```

tips: 在实际应用中，ioctl出错时的errorno大部分是ENOTTY(error not a typewriter)，顾名思义，即第一个参数fd指向的不是一个字符设备，不支持ioctl操作，这时候应该检查前面的open函数是否出错或者设备路径是否正确。

驱动中的ioctl()

1 long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);

2long (*compat_ioctl) (struct file *, unsigned int, unsigned long);

在新版内核中，unlocked_ioctl()与compat_ioctl()取代了ioctl()。unlocked_ioctl()，顾名思义，应该在无大内核锁（BKL）的情况下调用；compat_ioctl()，compat全称compatible（兼容的），主要目的是为64位系统提供32位ioctl的兼容方法，也是在无大内核锁的情况下调用。

tips：在字符设备驱动开发中，一般情况下只要实现unlocked_ioctl()即可，因为在vfs层的代码是直接调用unlocked_ioctl()。

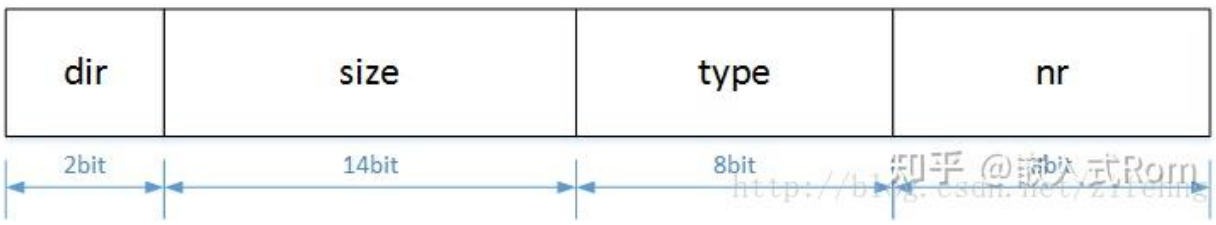
```
static long vfs_ioctl(struct file *filp, unsigned int cmd,
              unsigned long arg)
{
    int error = -ENOTTY;

    if (!filp->f_op || !filp->f_op->unlocked_ioctl)
        goto out;

    error = filp->f_op->unlocked_ioctl(filp, cmd, arg);
    if (error == -ENOIOCTLCMD) {
        error = -ENOTTY;
    }
 out:
    return error;
}
```

ioctl命令，用户与驱动之间的协议

前文提到ioctl方法第二个参数cmd为用户与驱动的“协议”，理论上可以为任意int型数据，可以为0、1、2、3……，但是为了确保该“协议”的唯一性，ioctl命令应该使用更科学严谨的方法赋值，在linux中，提供了一种ioctl命令的统一格式，将32位int型数据划分为四个位段，如下图所示:



在内核中，提供了宏接口以生成上述格式的ioctl命令:

```
// include/uapi/asm-generic/ioctl.h
```

```
((type) << _IOC_TYPESHIFT) | \
((nr)   << _IOC_NRSHIFT) | \
((size) << _IOC_SIZESHIFT))
```

1. **dir**(direction)，ioctl命令访问模式（数据传输方向），占据2bit，可以为_IOC_NONE、_IOC_READ、_IOC_WRITE、_IOC_READ | _IOC_WRITE，分别指示了四种访问模式：无数据、读数据、写数据、读写数据；
2. **type**(device type)，设备类型，占据8bit，在一些文献中翻译为“幻数”或者“魔数”，可以为任意char型字符，例如‘a’、‘b’、‘c’等等，其主要作用是使ioctl命令有唯一的设备标识。**tips：**Documentions/ioctl-number.txt记录了在内核中已经使用的“魔数”字符，为避免冲突，在自定义ioctl命令之前应该先查阅该文档。

3.**nr**(number)，命令编号/序数，占据8bit，可以为任意unsigned char型数据，取值范围0~255，如果定义了多个ioctl命令，通常从0开始编号递增；

4.**size**，涉及到ioctl第三个参数arg，占据13bit或者14bit（体系相关，arm架构一般为14位），指定了arg的数据类型及长度，如果在驱动的ioctl实现中不检查，通常可以忽略该参数。

通常而言，为了方便会使用宏_IOC()衍生的接口来直接定义ioctl命令：

```
// include/uapi/asm-generic/ioctl.h

/* used to create numbers */
#define _IO(type,nr)        _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,size)  _IOC(_IOC_READ,(type),(nr),(_IOC_TYPECHECK(size)))
#define _IOW(type,nr,size)  _IOC(_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
#define _IOWR(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
```

_IO	定义不带参数的ioctl命令
_IOW	定义带写参数的ioctl命令(copy_from_user)
_IOR	定义带读参数的ioctl命令(copy_to_user)
_IOWR	定义带读写参数的ioctl命令

同时，内核还提供了反向解析ioctl命令的宏接口：

```
// include/uapi/asm-generic/ioctl.h

/* used to decode ioctl numbers */
#define _IOC_DIR(nr)        (((nr) >> _IOC_DIRSHIFT) & _IOC_DIRMASK)
#define _IOC_TYPE(nr)       (((nr) >> _IOC_TYPESHIFT) & _IOC_TYPEMASK)
#define _IOC_NR(nr)     (((nr) >> _IOC_NRSHIFT) & _IOC_NRMASK)
#define _IOC_SIZE(nr)       (((nr) >> _IOC_SIZESHIFT) & _IOC_SIZEMASK)
```

ioctl-test，实例分析

本例假设一个带寄存器的设备，设计了一个ioctl接口实现设备初始化、读写寄存器等功能。在本例中，为了携带更多的数据，ioctl的第三个可变参数为指针类型，指向自定义的结构体struct msg。

1、ioctl-test.h，用户空间和内核空间共用的头文件，包含ioctl命令及相关宏定义，可以理解为一份“协议”文件，代码如下：

```
// ioctl-test.h

#ifndef __IOCTL_TEST_H__
#define __IOCTL_TEST_H__

#include <linux/ioctl.h>    // 内核空间
// #include <sys/ioctl.h>   // 用户空间

/* 定义设备类型 */
#define IOC_MAGIC  'c'
```

```
_IO(IOC_MAGIC, 0)

/* 读寄存器 */
#define IOCGREG    _IOW(IOC_MAGIC, 1, int)

/* 写寄存器 */
#define IOCWREG    _IOR(IOC_MAGIC, 2, int)

#define IOC_MAXNR  3

struct msg {
    int addr;
    unsigned int data;
};

#endif
```

2、ioctl-test-driver.c，字符设备驱动，实现了unlocked_ioctl接口，根据上层用户的cmd执行对应的操作（初始化设备、读寄存器、写寄存器）。在接收上层cmd之前应该对其进行充分的检查，流程及具体代码实现如下：

```
// ioctl-test-driver.c
......

static const struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = test_open,
    .release = test_close,
    .read = test_read,
    .write = etst_write,
    .unlocked_ioctl = test_ioctl,
};

......

static long test_ioctl(struct file *file, unsigned int cmd, \
                       unsigned long arg)
{
    //printk("[%s]\n", __func__);

    int ret;
    struct msg my_msg;

    /* 检查设备类型 */
    if (_IOC_TYPE(cmd) != IOC_MAGIC) {
        pr_err("[%s] command type [%c] error!\n", \
            __func__, _IOC_TYPE(cmd));
        return -ENOTTY;
    }

    /* 检查序数 */
    if (_IOC_NR(cmd) > IOC_MAXNR) {
        pr_err("[%s] command numer [%d] exceeded!\n",
            __func__, _IOC_NR(cmd));
        return -ENOTTY;
    }

    /* 检查访问模式 */
    if (_IOC_DIR(cmd) & _IOC_READ)
        ret= !access_ok(VERIFY_WRITE, (void __user *)arg, \
                _IOC_SIZE(cmd));
```

```
    IR(cmd) & _IOC_WRITE)
    ss_ok(VERIFY_READ, (void __user *)arg, \
                _IOC_SIZE(cmd));
    if (ret)
        return -EFAULT;

    switch(cmd) {
    /* 初始化设备 */
    case IOCINIT:
        init();
        break;

    /* 读寄存器 */
    case IOCGREG:
        ret = copy_from_user(&msg, \
            (struct msg __user *)arg, sizeof(my_msg));
        if (ret)
            return -EFAULT;
        msg->data = read_reg(msg->addr);
        ret = copy_to_user((struct msg __user *)arg, \
                &msg, sizeof(my_msg));
        if (ret)
            return -EFAULT;
        break;

    /* 写寄存器 */
    case IOCWREG:
        ret = copy_from_user(&msg, \
            (struct msg __user *)arg, sizeof(my_msg));
        if (ret)
            return -EFAULT;
        write_reg(msg->addr, msg->data);
        break;

    default:
        return -ENOTTY;
    }

    return 0;
}
```

ioctl-test.c，运行在用户空间的测试程序：

```
// ioctl-test.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "ioctl-test.h"

int main(int argc, char **argv)
{

    int fd;
    int ret;
```

```
    struct ioctl_msg my_msg;

    fd = open("/dev/ioctl-test", O_RDWR);
    if (fd < 0) {
        perror("open");
        exit(-2);
    }

    /* 初始化设备 */
    ret = ioctl(fd, IOCINIT);
    if (ret) {
        perror("ioctl init:");
        exit(-3);
    }

    /* 往寄存器0x01写入数据0xef */
    memset(&my_msg, 0, sizeof(my_msg));
    my_msg.addr = 0x01;
    my_msg.data = 0xef;
    ret = ioctl(fd, IOCWREG, &my_msg);
    if (ret) {
        perror("ioctl read:");
        exit(-4);
    }

    /* 读寄存器0x01 */
    memset(&my_msg, 0, sizeof(my_msg));
    my_msg.addr = 0x01;
    ret = ioctl(fd, IOCGREG, &my_msg);
    if (ret) {
        perror("ioctl write");
        exit(-5);
    }
    printf("read: %#x\n", my_msg.data);

    return 0;
}
```

无标签