第13章 Linux驱动程序设计

了解概念

什么是驱动程序

1. 操作系统通过各种驱动程序来**驾驭硬件设备**

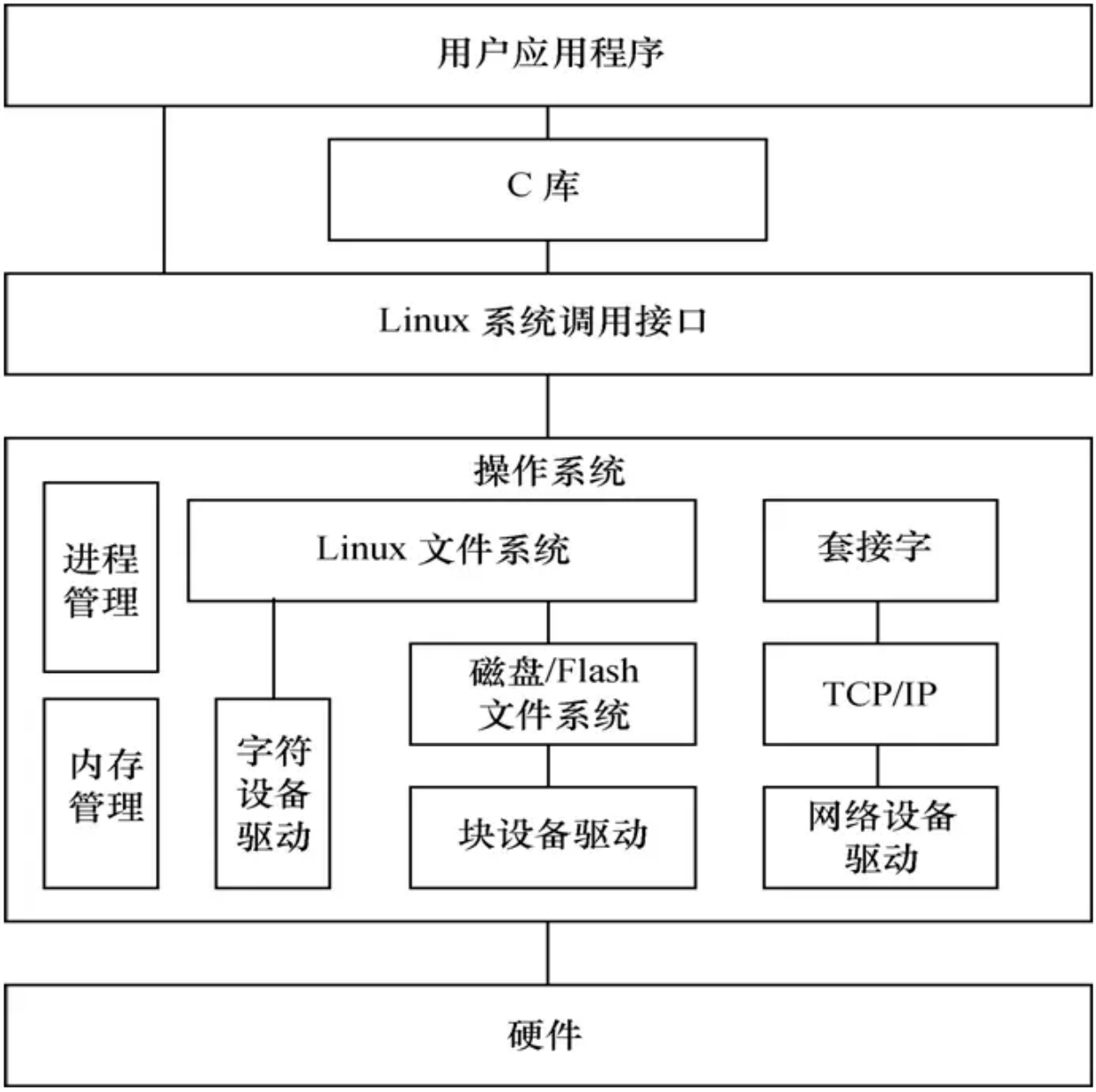
2. 设备驱动程序是内核的一部分，硬件驱动程序是操作系统最基本的组成部分

3. Linux 内核中采用可加载的模块化设计（LKMs， Loadable Kernel Modules），一般情况下编译的 Linux 内核是支持可插入式模块的，也就是将最基本的核心代码编译在内核中，其他的代码可以编译到内核中，或者编译为内核的模块文件（在需要时动态加载）。

4. 常见的驱动程序是作为内核模块动态加载的， 比如声卡驱动和网卡驱动等，而 Linux 最基础的驱动， 如 CPU、PCI 总线、 TCP/IP 协议、 APM（高级电源管理）、 VFS 等驱动程序则直接编译在内核文件中。

5. 有时也把内核模块叫做驱动程序，只不过驱动的内容不一定是硬件罢了，比如 ext3 文件系统的驱动。因此，加载驱动就是加载内核模块。

设备驱动程序与整个软硬件系统的关系



设备分类

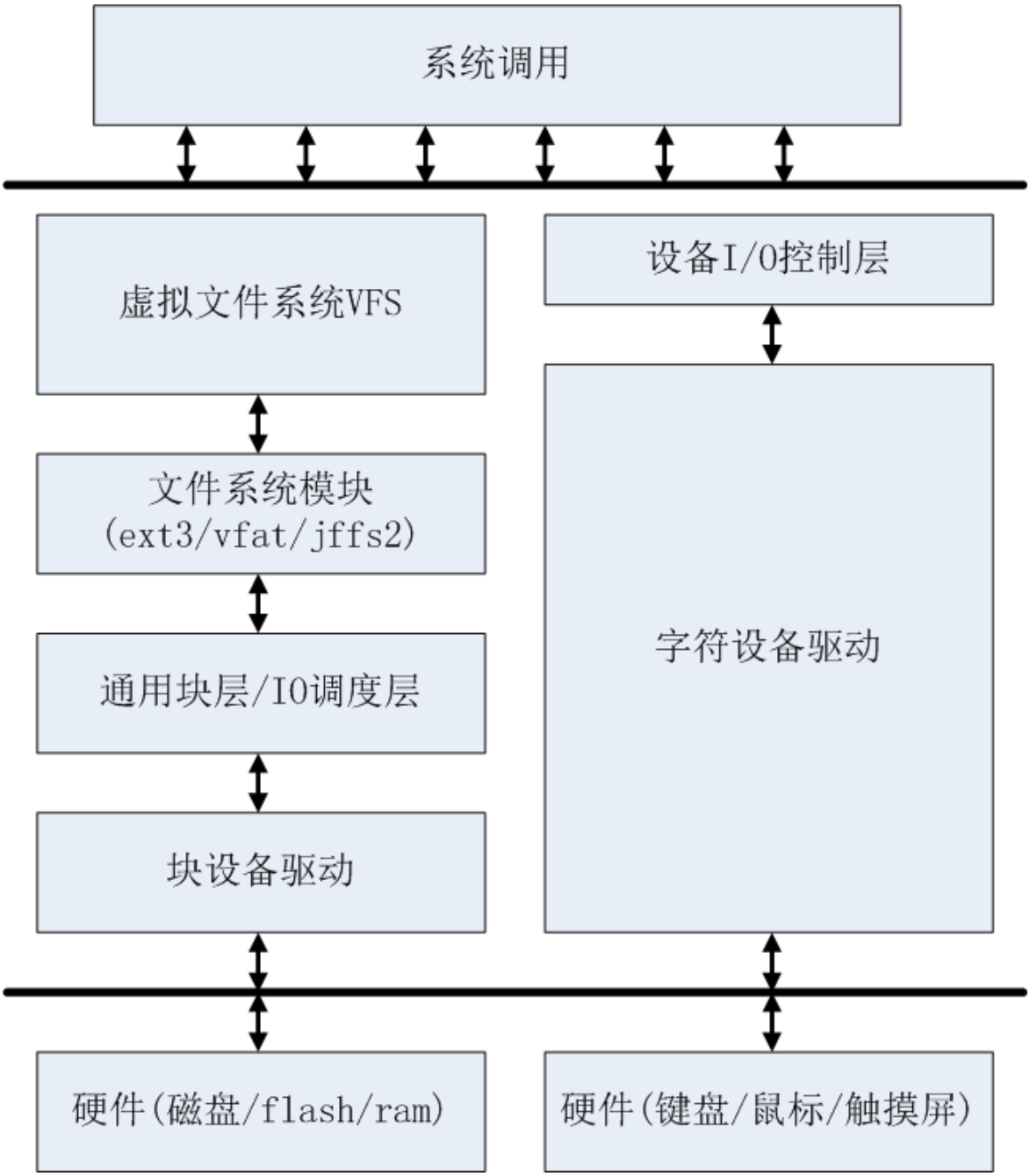
> Linux将所有设备都当作文件进行处理，这类特殊文件就是设备文件，通常在 `/dev` 下面对应一个**以文件形式存在的逻辑设备节点。**

- 字符设备
  - 像普通文件或字节流一样，**以字节为单位顺序读写**的设备，如控制台（shell）。
  - 通过设备文件节点访问
  - 普通文件可以被随机访问（可以前后移动访问指针），而大多数字符设备只能提供顺序访问，因为对它们的访问不会被系统所缓存。但也有例外。
- 块设备
  - 需要**以块为单位随机读写**的设备，如硬盘。
  - 通过设备文件节点来访问，它不仅可以提供随机访问，而且可以容纳文件系统（例如硬盘、闪存等）。
- 网络设备
  - 通过网络能够与其他主机进行数据通信的设备，如网卡等

> 除网络设备外，字符设备与块设备都被映射到Linux文件系统的文件和目录

驱动程序分类

- 字符设备驱动程序
- 块设备驱动程序
- 网络设备驱动程序

> **字符设备驱动与块设备驱动的区别**
>
> 
>

设备文件与设备号

设备文件

- 为了体现“一切都是文件”的设计思想，linux将每个已安装的设备都表示为一个设备文件。
- 设备文件通常位于/dev子目录。
- 对于字符设备，应用程序可以利用open、close、read、write等**系统调用访问**其设备文件，这些I/O操作都被**直接传递给该设备文件所对应的设备**。

创建设备文件与删除设备文件

```
$ mknod filename c major minor      // 创建设备文件
$ rmnod filename                    // 删除设备文件,注意删除设备文件并不会影响驱动模块
```

设备号（设备的标志）

每个**设备文件**中都存储了该设备的"主设备号"和"次设备号"。
Linux内核利用**主设备号**查找该设备所**对应的设备驱动程序**，
利用**次设备号**标识**具体的设备。**
一个设备文件（即设备节点）可以通过`mknod`命令来创建，其中指定了主设备号和次设备号。

- 主设备号（高字节）：表明设备的类型，与一个确定的驱动程序对应
- 次设备号（低字节）：标明不同的属性（标志着某个具体的物理设备）

```
$ ls –l /dev
crw-rw---- 1 root uucp 4, 64 08-30 22:58 ttyS0 /* 主设备号 4，次设备号 64 */
```

管态（内核态、核心态）与目态（用户态、算态）

访管指令

从用户态到内核态

访管指令：（访问管态的指令）

- 访管指令是用户程序在需要操作系统服务时，安排的一条特殊指令。
- 当处理器执行到访管指令时，会产生一个中断事件（自愿中断），暂停用户程序的执行，并转由操作系统来处理该中断事件。
- 操作系统会分析访管指令中的参数，并调用相应的系统调用子程序来为用户服务。
- 服务完成后，操作系统会将处理器的状态从管态（核心态）切换回目态（用户态），并返回到用户程序的断点处继续执行。

与系统调用的关系

系统调用命令中总是包含一条访管指令。
当用户程序想要操作系统提供服务时，会在用户程序中使用系统调用命令。

在执行系统调用命令的过程中，当执行到访管指令时，就会触发访管中断，使得处理器从用户态切换到核心态（管态），以执行特权指令完成操作系统提供的功能。
当中断处理程序结束后，处理器又会从核心态返回用户态，继续执行用户程序。

特权指令

只能在内核态执行的指令

模块（内核模块）

认识

为了使系统功能能够更灵活的扩充，Linux支持内核的动态扩展，即在**系统运行时给内核增加新的功能**（即模块module）。
模块（module）是一段可以被动态链接的目标代码（**.ko**），它可由**insmod命令**动态的装载并链接到正在运行的内核。链接后，它就成了内核的一部分，直到用**rmmod命令**解除链接并卸载。
Linux驱动程序就是一种特殊的内核模块。

内核模块（驱动程序）与应用程序的区别

- 内核模块工作在内核空间（supervisor space），而应用程序工作在用户空间（user space）
- 内核模块是一个由多个回调函数组成的“被动”代码集合体，采用了“事件驱动模型”；而应用程序总是从头至尾的执行单个任务。
- 内核模块不能调用C标准函数库，只能调用linux内核导出的内核函数。
- 内核模块在编程时必须考虑可重入性（reentrant）
- 内核模块可使用的栈很小。（内核空间小）

内核模块的加载与卸载

- 使用insmod命令动态加载模块：

```
insmod ./hello.ko
```

- 使用rmmod命令卸载模块：

```
rmmod hello
```

查看内核已加载模块

使用lsmod命令查看内核中已加载的内核模块的信息

```
lsmod
```

通过查看/proc/modules文件也可查看内核中已加载的内核模块的信息。

```
cat /proc/modules
```

通过查看/sys/module目录也可查看内核中已加载的内核模块的信息。

```
cat /sys/modules
```

查看内核输出信息

```
dmesg
```

Linux内核模块编程

内核模块的程序结构

模块加载函数（必须）

```
static int __init initialization_function(void)
{
/* 初始化代码 */
}
module_init(initialization_function);
```

1. `static int __init initialization_function(void)`：
    - `static`：表示该函数**只在当前源文件中可见**，不会被其他文件链接。

- `int`：表示函数返回一个整数，通常用于表示初始化是否成功（0表示成功，非0表示失败）。
- `__init`：这是一个特殊的宏，用于**标记初始化函数**。它告诉编译器这个函数只在模块加载时执行，并且它的代码段在模块加载后可以被丢弃（从而节省内存）。
- `initialization_function`：这是初始化函数的名称。
- `(void)`：表示这个函数没有参数。
- 在函数体内，通常会看到用于初始化硬件、数据结构或执行其他设置任务的代码。

2. `module_init(initialization_function);`：
   - `module_init`：这是一个宏，用于**注册初始化函数**。当模块被加载到内核时，它会调用此函数。
   - `initialization_function`：这是要注册的初始化函数的名称。

模块卸载函数（必须）

```
static void __exit cleanup_function(void)
{
/* 释放代码 */
}
module_exit(cleanup_function); //通常来说，模块卸载函数
```

在Linux内核编程中，`module_exit`宏是用来定义模块卸载时的清理函数的。当模块被卸载时，`cleanup_function`会被调用，用于释放模块在内存中分配的资源。

通常来说，模块卸载函数要完成**与模块加载函数相反的功能**

- 若模块加载函数注册XXX，则模块卸载函数应该注销XXX。
- 若模块加载函数动态申请了内存，则模块卸载函数应释放该内存。
- 若模块加载函数申请了硬件资源（中断、DMA通道、I/O端口和I/O内存等)的占用，则模块卸载函数应释放这些硬件资源。
- 若模块加载函数开启了硬件,则卸载函数中一般要关闭硬件。

模块许可证声明（必须）

```
module_license("Dual BSD/GPL");
```

Linux模块许可证主要分为以下几类：

- GPL（GNU通用公共许可证）：GPL是GNU项目创立者Richard Stallman制定的许可证，要求任何使用、修改和分发GPL许可证下的软件的人都必须遵循相同的许可证条款。采用GPL许可证的软件的衍生产品也必须采用GPL许可证。
- LGPL（GNU库许可证）：LGPL是GNU项目的一部分，主要用于保护Linux内核模块等库文件。与GPL不同，LGPL允许商业公司将LGPL库与闭源软件混合发布，使得闭源软件也能享受到LGPL库的自由特性。
- BSD许可证：BSD许可证允许商业公司将开源软件与闭源软件混合发布，但要求闭源软件的源代码中必须包含原开源软件的致谢、许可证声明和免责声明。
- MIT许可证：MIT许可证与BSD许可证类似，但更加简单，仅要求在开源软件的致谢和许可证声明中包含相关信息。

模块参数（可选）

```
module_param(参数名,参数类型,参数读/写权限)  //为模块定义一个参数

static char *str_param = "Linux Module Program";
static int num_param = 4000;
```

```
module_param(str_param, charp, S_IRUGO);
module_param(num_param, int, S_IRUGO);
```

在装载内核模块时，用户可以向模块传递参数，形式为：

```
insmod（或modprobe）模块名 参数名=参数值
```

如果不传递，参数将使用模块内定义的默认值。

参数类型可以是byte、short、ushort、int、uint、long、ulong、charp（字符指针）、bool 或invbool（布尔的反）
在模块被编译时会将module_param中声明的类型与变量定义的类型进行比较，判断是否一致。

模块导出符号（可选）

```
EXPORT_SYMBOL(符号名);
EXPORT_SYMBOL_GPL(符号名);
```

导出符号可以用

```
$ cat /proc/kallsyms | grep "total_symbols"
```

检查

模块作者等信息声明（可选）

```
MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);
```

模块的使用计数

在Linux内核中，模块的使用计数管理是非常重要的，因为它**决定了模块是否可以被卸载。**
当一个模块被加载时，它的使用计数是1。
当其他内核代码或用户空间程序使用这个模块时，它会增加模块的使用计数。
如果使用计数**减少到0**，内核可以**安全地卸载模块。**

在Linux 2.4内核中，模块的使用计数是通过宏来管理的，如 `MOD_INC_USE_COUNT` 和 `MOD_DEC_USE_COUNT` 。这些宏是递增和递减模块使用计数的函数。

在Linux 2.6内核中，引入了 `try_module_get` 和 `module_put` 函数来**更安全地管理**模块的使用计数。这些函数考虑了SMP（对称多处理器）和PREEMPT（可抢占式内核）机制的影响，确保了模块的使用计数管理是线程安全的。

`try_module_get` 函数用于尝试增加模块的使用计数。如果模块已经被加载并且可以被使用，它会成功增加计数并返回非零值。如果模块正在被卸载或者尚未被加载，它会返回0，表示调用失败。

`module_put` 函数用于减少模块的使用计数。当不再需要模块时，应该调用这个函数来减少计数。

```
// 尝试增加模块的使用计数
if (try_module_get(module)) {
    // 模块可以被使用
    // 执行需要模块的操作
```

```
} else {
    // 模块不能被使用
    // 处理错误情况
}

// 减少模块的使用计数
module_put(module);
```

使用这些函数而不是宏的好处在于，它们提供了更清晰的错误处理方式，并且是线程安全的。在编写内核代码时，应该优先使用这些函数来管理模块的使用计数。

Linux内核模块编译

makefile文件第一行：

```
obj-m := hello.o
```

`obj-m := hello.o` 这行告诉Makefile，要编译的模块目标文件是 `hello.o` 。在Linux内核模块编译过程中， `obj-m` 变量用于指定要编译的模块目标文件。这里的 `hello.o` 将会被编译成 `hello.ko` （即 `hello` 模块）

使用如下命令编译HelloWorld模块，如下所示：

```
$ make -C /usr/src/linux-2.6.15.5/ M=/driver_study/ modules
```

这个命令是告诉 `make` 工具
进入Linux内核源代码目录（由 `-C` 参数指定），编译到某个内核的目录下（目标内核源码目录）,然后在该目录下执行Makefile。
`M=/driver_study/` 参数告诉 `make` ，模块源代码位于 `/driver_study/` 目录下。
最后 `modules` 指定要编译的是内核模块。

如果当前处于模块所在的目录，以下命令与上述命令同等：

```
$ make –C /usr/src/linux-2.6.15.5 M=$(pwd) modules
```

这个命令与第一个命令作用相同，但是它使用了环境变量 `$(pwd)` 来动态获取当前工作目录，这样就不需要明确指定模块源代码的路径，适用于**已经在**模块**源代码目录下工作**的情况。

设备驱动程序的特点

- 内核代码：设备驱动程序是**内核的一部分**，如果驱动程序出错，则可能导致**系统崩溃**。
- 内核接口：设备驱动程序必须为内核或者其子系统提供一个标准接口。比如，一个终端驱动程序必须为内核提供一个文件I/O接口；一个SCSI设备驱动程序应该为SCSI子系统提供一个SCSI设备接口，同时SCSI子系统也必须为内核提供文件的I/O接口及缓冲区。
- 内核机制和服务：设备驱动程序使用一些标准的内核服务，如内存分配等。
- 可装载：大多数的Linux操作系统设备驱动程序都可以在需要时装载进内核，在不需要时从内核中卸载。
- 可设置：Linux操作系统设备驱动程序可以集成为内核的一部分，并可以根据需要把其中的某一部分集成到内核中，这只需要在系统编译时进行相应的设置即可。
- 动态性：在系统启动且各个设备驱动程序初始化后，驱动程序将维护其控制的设备。如果该设备驱动程序控制的设备不存在也不影响系统的运行，那么此时的设备驱动程序只是多占用了一点系统内存罢了。

驱动程序与硬件设备通信

根据CPU体系结构的不同，对外设端口的编址方式有两种：

- 独立编址：分I/O地址空间和内存地址空间，CPU有专门的I/O指令。（x86处理器等）
- 统一编址：只有一个内存地址空间，物理内存和外设寄存器都映射于其中。（ARM，PowerPC等）

这里介绍两种通信方式：

1. 外设寄存器位于I/O地址空间时，通常被称为**I/O端口**
2. 外设寄存器位于内存地址空间时，通常被称为**I/O内存**

I/O端口通信

I/O端口地址分配

为了避免与其他外设在访问I/O地址空间时发生冲突，驱动程序最好向操作系统声明自己要占用的I/O地址区间。这些函数是Linux内核特有的函数，它们不与标准C库中的函数相混淆。

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long start, unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
int check_region(unsigned long start, unsigned long len);
```

- `request_region(unsigned long start, unsigned long len, char *name)`： 这个函数用于请求对指定范围的I/O端口的访问。如果指定范围内的端口没有被其他设备占用，它将返回一个指向`struct resource`的指针，该结构包含了端口的详细信息。如果指定范围内的端口已经被占用，它将返回`NULL`。
- `release_region(unsigned long start, unsigned long len)`： 这个函数用于释放之前通过`request_region`请求的I/O端口范围。如果指定范围内的端口之前被请求过，它将释放这些端口。
- `check_region(unsigned long start, unsigned long len)`： 这个函数用于检查指定范围内的I/O端口是否被其他设备占用。如果指定范围内的端口被占用，它将返回非零值；如果未被占用，它将返回零。

读写I/O端口

对于通过I/O端口进行通信的硬件，驱动程序会使用`inb`、`outb`、`inw`、`outw`、`inl`、`outl`等汇编语言宏来读写端口。这些函数是系统调用，而不是标准C库函数，它们存在于Linux内核中，用于与硬件进行低级通信。

```
inb(unsigned port)                  //从指定的端口读取一个字节（8位）
outb(unsigned char byte, unsigned port)  //将指定的字节写入指定的端口
inw(unsigned port)                  //从指定的端口读取一个字（16位）
outw(unsigned short word, unsigned port) //将指定的字写入指定的端口
inl(unsigned port)                  //从指定的端口读取一个长字（32位）
outl(unsigned longword, unsigned port)   //将指定的长字写入指定的端口
```

I/O内存通信

I/O内存分配

对于直接访问内存区域的硬件，驱动程序会使用`ioremap`和`iounmap`等函数来映射和取消映射I/O内存。

在Linux内核编程中，`request_mem_region`、`release_mem_region`、`check_mem_region`、`ioremap`和`iounmap`这些函数用于管理对物理内存区域的访问。这些函数通常用于与通过内存映射I/O（MMIO）进行通信的硬件（如PCI设备）进行通信。

- `request_mem_region(unsigned long start, unsigned long len, char *name)`： 这个函数用于请求对指定范围的物理内存区域的访问。如果指定范围内的内存没有被其他设备占用，它将返回一个指向`struct resource`的指针，该结构包含了内存区域的详细信息。如果指定范围内的内存已经被占用，它将返回`NULL`。
- `release_mem_region(unsigned long start, unsigned long len)`： 这个函数用于释放之前通过`request_mem_region`请求的物理内存区域。如果指定范围内的内存之前被请求过，它将释放这些内存。
- `check_mem_region(unsigned long start, unsigned long len)`： 这个函数用于检查指定范围内的物理内存是否被其他设备占用。如果指定范围内的内存被占用，它将返回非零值；如果未被占用，它将返回零。

- `ioremap(unsigned long phys_addr, unsigned long size)`： 这个函数用于将指定的物理内存地址映射到内核虚拟地址空间。返回值是一个指向映射内存区域的指针。这个函数通常用于将物理内存地址映射到内核可以访问的虚拟地址空间。
- `ioremap_nocache(unsigned long phys_addr, unsigned long size)`： 这个函数与`ioremap`类似，但它不会对映射的内存区域进行缓存。这通常用于需要直接访问物理内存的情况。
- `iounmap(void *virt_addr)`： 这个函数用于取消之前通过`ioremap`或`ioremap_nocache`映射的内存区域。它接受一个指向映射内存区域的指针作为参数，并取消映射。

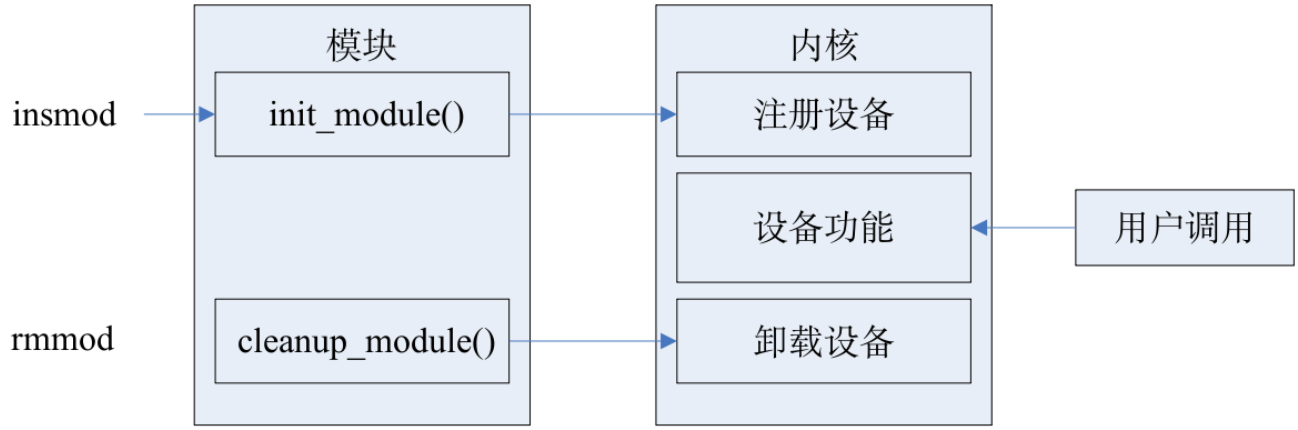
这些函数是内核空间的一部分，用于管理对物理内存区域的访问。在编写内核模块时，需要使用这些函数来确保对物理内存区域的访问不会与其他设备冲突。

读写I/O内存

在Linux内核编程中，`ioread8`、`ioread16`、`ioread32`、`iowrite8`、`iowrite16`、`iowrite32`这些函数用于通过内存映射I/O（MMIO）进行读写操作。这些函数是内核空间的一部分，用于与硬件进行低级通信。

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

字符设备驱动编程



重要数据结构

file_operations

定义

```
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *filp, char *buff, size_t count, loff_t *offp);
    ssize_t (*write) (struct file *filp, const char *buff, size_t count, loff_t *offp);
    int (*readdir) (struct file *, void *, filldir_t);
```

```
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *);
    int (*fasync) (int, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
    int (*lock) (struct file *, int, struct file_lock *);
};
```

用途：定义一组文件操作的集合

结构体里的函数指针定义了各种操作（如读取、写入、寻址等）的方法。当为设备驱动程序编写内核代码时，需要提供这些函数的实现，以定义如何与用户空间交互。

使用方法

```
#include <linux/fs.h>

struct file_operations my_driver_fops = {
    .open = my_driver_open,
    .release = my_driver_release,
    .write = my_driver_write,
    .read = my_driver_read,
    // ... 其他文件操作函数
};

int my_driver_open(struct inode *inode, struct file *file)
{
    // 打开设备的实现
    return 0;
}

int my_driver_release(struct inode *inode, struct file *file)
{
    // 关闭设备的实现
    return 0;
}

ssize_t my_driver_write(struct file *file, const char *buffer, size_t length, loff_t *offset)
{
    // 写入设备的实现
    return length;
}

ssize_t my_driver_read(struct file *file, char *buffer, size_t length, loff_t *offset)
{
    // 从设备读取的实现
    return length;
}

// ... 其他文件操作函数的实现
```

struct file

定义

`struct file`是Linux内核中用于描述打开的文件的数据结构。当一个文件被打开时，内核会创建一个`file`结构来存储与该文件相关的信息。

```
struct file {
    mode_t f_mode;/*标识文件是否可读或可写，FMODE_READ或FMODE_WRITE*/
    dev_t f_rdev; /* 用于/dev/tty */
    off_t f_pos; /* 当前文件位移 */
    unsigned short f_flags; /* 文件标志，如O_RDONLY、O_NONBLOCK和O_SYNC */
    unsigned short f_count; /* 打开的文件数目 */
    unsigned short f_reada;
    struct inode *f_inode; /*指向inode的结构指针 */
    struct file_operations *f_op;/* 文件索引指针 */
};
```

- `f_mode`： 文件的模式，指示文件是否可读或可写。
- `f_rdev`： 用于特殊文件（如设备文件）的设备类型。
- `f_pos`： 文件当前位置。
- `f_flags`： 打开文件时的标志，如O_RDONLY、O_WRONLY、O_RDWR等。
- `f_count`： 打开文件的数量。
- `f_inode`： 指向`struct inode`的指针，表示与该文件关联的inode。
- `f_op`： 指向`struct file_operations`的指针，定义了文件的行为。

设备号注册与注销

设备号实现

设备号数据类型`dev_t`：在2.6内核中为32位，高12位为主设备号，低20位为次设备号

获取设备号

```
MAJOR(dev_t dev);             /*获得主设备号*/
MINOR(dev_t dev);             /*获得次设备号*/
MKDEV(int major, int minor);  /*将给定的主设备号（`major`）和次设备号（`minor`）组合成一个`dev_t`类型的设备号
*/
```

设备号注册（字符设备）

register_chrdev_region()（设备号范围已知）

```
int register_chrdev_region(
    dev_t from,
    unsigned count,
    const char *name
);
```

参数说明：

- `dev_t from`： 表示要注册的设备号范围的**起始设备号**。
- `unsigned count`： 表示**要注册的设备号的数量**。
- `const char *name`： 表示**设备名称的字符串**，这个名称通常用于在`/proc/devices`文件中显示，也可能在其他内核和用户空间交互的上下文中使用。

返回值：

- 如果成功，函数返回 0。
- 如果失败（例如，指定的设备号范围已经被占用），函数返回一个负的错误码。

alloc_chrdev_region()（自动选择一个未被使用的设备号范围）

```
int alloc_chrdev_region(
    dev_t *dev,
    unsigned int baseminor,
    unsigned int count, c
    const char *name
);
```

参数说明

- `dev_t *dev`：这是一个指向 `dev_t` 类型变量的指针，函数执行成功后会将分配到的设备号保存在这个变量中。
- `unsigned int baseminor`：指定**从哪个次设备号开始**分配设备号。通常设置为 0，表示从第一个次设备号开始分配。
- `unsigned int count`：指定**要分配的设备号的数量**。这通常对应于驱动程序支持的设备实例的数量。
- `const char *name`：设备的**名称**，用于在 `/proc/devices` 文件中显示。

返回值

- 如果成功，函数返回 0。
- 如果失败（例如，由于内存不足或无法找到可用的设备号范围），函数返回一个负的错误码。

设备号注销

unregister_chrdev_region()：用于注销字符设备号

```
void unregister_chrdev_region(
    dev_t from,
    unsigned int count
);
```

参数说明

- `dev_t from`：表示要释放的设备号范围的**起始设备号**。
- `unsigned int count`：表示**要释放的设备号的数量**。

返回值
此函数没有返回值，即其返回类型为 `void`。如果操作成功，它会静默地释放设备号；如果失败（例如，指定的设备号范围不存在或已被其他部分占用），它可能会产生内核错误或警告。

设备注册与注销

struct cdev（结构体）

`struct cdev` 是一个用于描述字符设备的数据结构。它包含了字符设备的元数据，如设备号、文件操作表指针等。当注册一个字符设备时，内核会创建一个 `struct cdev` 结构来描述该设备。

cdev_alloc()（动态申请一个 `cdev` 结构体的内存）

```
struct cdev *cdev_alloc(void);
```

返回值：

- 如果成功，返回一个新的`struct cdev`指针。
- 如果内存分配失败，返回`NULL`。

cdev_init()（初始化`cdev`结构体）

```
void cdev_init(
    struct cdev *p,
    const struct file_operations *fops
);
```

功能：初始化一个已经分配好的`cdev`结构体，并建立它与`file_operations`结构体之间的连接。
函数参数：

- `p`： 指向要初始化的`struct cdev`实例的指针。
- `fops`： 指向`struct file_operations`的指针，定义了设备的文件操作行为。

返回值：

- 无返回值。

cdev_add()（将设备与设备号关联起来）

将一个已初始化的`struct cdev`实例添加到内核的设备列表中，为指定的设备号创建一个设备文件，并设置设备文件的权限和操作。

```
int cdev_add(
    struct cdev *p,
    dev_t dev,
    unsigned count
);
```

函数参数：

- `p`： 指向要添加的`struct cdev`实例的指针。
- `dev`： 设备号，包括主设备号和次设备号。
- `count`： 设备实例的**数量**。

函数返回值：

- 如果成功，返回0。
- 如果设备号已经存在或者设备添加失败，返回负数错误码。

cdev_del()（注销一个字符设备）

用于从内核的设备列表中删除一个已注册的`struct cdev`实例并释放内核资源。

```
void cdev_del(struct cdev *p);
```

函数参数：

- `p`： 指向要删除的`struct cdev`实例的指针。

返回值：

- 无返回值。

file_operations中部分函数的实现

打开设备open

在file_operations结构体中定义：

```
int (*open) (struct inode *, struct file *);
```

通常情况下在open函数接口中要完成如下工作：

- 如果未初始化，则进行初始化。
- 识别次设备号，如果必要，更新f_op指针。
- 分配并填写被置于filp→private_data的数据结构。
- 检查设备特定的错误（诸如设备未就绪或类似的硬件问题）

释放设备release

```
int (*release) (struct inode *, struct file *);
```

释放设备时要完成的工作如下：

- 释放打开设备时系统所分配的内存空间（包括filp→private_data指向的内存空间）。
- 在最后一次关闭设备（使用close()系统调用）时，才会真正释放设备（执行release()函数）。即在打开计数等于0时的close()系统调用才会真正进行设备的释放操作。

读设备read

```
ssize_t (*read) (struct file *filp, char *buff, size_t count, loff_t *offp);
```

写设备write

```
ssize_t (*write) (struct file *filp, const char *buff, size_t count, loff_t *offp);
```

非读写操作机制ioctl

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

大部分设备除了读写操作，还需要硬件配置和控制（例如，设置串口设备的波特率）等很多其他操作。
在字符设备驱动中ioctl函数接口给用户提供对设备的非读写操作机制。

辅助以上函数指针实现的内核函数

copy_to_user()（read函数指针的实现）

`copy_to_user()` 函数用于将数据从内核空间复制到用户空间。由于内核不能直接访问用户空间的内存（出于安全和稳定性的原因），因此需要使用这个函数来确保数据的安全复制。

函数原型

```
unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);
```

- `to`：指向用户空间的目标地址的指针。
- `from`：指向内核空间源数据的指针。
- `n`：要复制的字节数。

返回值：成功复制的字节数（通常应为`n`，除非发生错误）。如果返回值小于`n`，则表示复制过程中发生了错误。

copy_from_user()（write函数指针的实现）

`copy_from_user`用于将数据从用户空间复制到内核空间。

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);
```

- `to`：指向内核空间目标地址的指针。
- `from`：指向用户空间源数据的指针。
- `n`：要复制的字节数。

返回值：成功复制的字节数（通常应为`n`，除非发生错误）。如果返回值小于`n`，则表示复制过程中发生了错误。

获取内存

```
void *kmalloc(size_t size, gfp_t flags);
```

函数参数：

- `size`：要分配的内存大小，以字节为单位。
- `flags`：指定内存分配的属性，如是否允许睡眠、是否需要内存锁定等。

函数返回值：

- 如果成功，返回指向新分配内存的指针。
- 如果内存分配失败，返回`NULL`。

`kmalloc`函数会返回一个指向新分配的内存块的指针，该内存块的大小至少为`size`字节。
这个内存块位于**内核内存**中，因此不需要通过页表来访问。

`flags`参数是一个`gfp_t`类型的枚举值，它用于指定内存分配的属性。常见的`gfp_t`标志包括：

- `GFP_KERNEL`：默认标志，允许在低优先级上下文中睡眠。
- `GFP_ATOMIC`：不允许睡眠，适合于中断上下文。
- `GFP_DMA`：适合于DMA传输，可能会导致性能下降。
- `GFP_IO`：允许在IO上下文中睡眠。

释放内存

```
void kfree(const void *ptr);
```

函数参数：

- `ptr`：指向之前通过内核内存分配器分配的内存块的指针。

函数返回值：

- 无返回值。

`kfree`函数接受一个指针作为参数，该指针**指向内核内存中的一个块**。函数会释放这个内存块，并将其回收到内核内存池中，以便其他内核代码可以重新使用。

打印信息

```
void printk(const char *fmt, ...);
```

函数参数：

- `fmt`：格式化字符串，与`printf`函数的格式化字符串类似。
- `...`：可变数量的参数，用于填充`fmt`字符串中的格式化项。

函数返回值：

- 无返回值。

`printk`函数接受一个格式化字符串和一系列可选参数。它将这些参数插入到格式化字符串中，并将其打印到控制台或其他日志系统。
查看需要通过`dmesg`命令（重点）

其他

设备与文件

- 设备文件都在`/dev`目录下
- 通过对文件操作可以实现对设备的访问

主设备与次设备

- 主设备用来标识与设备相关的驱动（one-major-one-drive）**一个主设备一个驱动**
  - 多个同类设备（主设备号相同，次设备号不同）使用同一个驱动程序
- 次设备号标识具体设备

设备号

- 查看：/proc/devices显示了已分配哪些设备号、设备名
- 可以向系统申请指定的主设备号
- 可以向系统动态申请主设备号