

openEuler 操作系统

内核实验手册

鲲鹏云 ECS 版

v1.0



中国科学院软件研究所
Institute of Software, Chinese Academy of Sciences

华为技术有限公司

中科院软件所



目录

前言.....	1
1 实验一 鲲鹏云 ECS 的构建及内核编译	3
1.1 实验介绍	3
1.1.1 任务描述	3
1.2 实验目的	3
1.3 构建云实验环境	3
1.3.1 创建 VPC.....	3
1.3.2 购买 ECS	6
1.3.3 通过 ssh 登录系统.....	8
1.4 实验任务	9
1.4.1 openEuler 内核编译与安装.....	9
1.4.2 Hello, world!	12
1.5 云环境资源清理	13
1.5.1 ECS 关机	13
1.5.2 删除 ECS	14
2 实验二 内存管理.....	17
2.1 实验介绍	17
2.1.1 相关知识	17
2.1.2 任务描述	18
2.2 实验目的	18
2.3 实验任务	19
2.3.1 使用 kmalloc 分配 1KB, 8KB 的内存, 打印指针地址.....	19
2.3.2 使用 vmalloc 分配 8KB、1MB、64MB 的内存, 打印指针地址.....	19
2.3.3 实验结果分析	20
2.4 思考题	21
3 实验三 进程管理.....	23
3.1 实验介绍	23

3.1.1 任务描述	23
3.2 实验目的	23
3.3 实验任务	23
3.3.1 创建内核进程	23
3.3.2 打印输出当前系统 CPU 负载情况	26
3.3.3 打印输出当前处于运行状态的进程的 PID 和名字	28
3.3.4 使用 cgroup 实现限制 CPU 核数	29
4 实验四 中断和异常管理	35
4.1 实验介绍	35
4.1.1 相关概念	35
4.1.2 任务描述	36
4.2 实验目的	37
4.3 实验任务	37
4.3.1 使用 tasklet 实现打印 helloworld	37
4.3.2 用工作队列实现周期打印 helloworld	38
4.3.3 编写一个信号捕获程序，捕获终端按键信号	41
5 实验五 内核时间管理	47
5.1 实验介绍	47
5.1.1 任务描述	47
5.2 实验目的	47
5.3 实验任务	47
5.3.1 调用内核时钟接口打印当前时间	47
5.3.2 编写 timer，在特定时刻打印 hello,world	49
5.3.3 调用内核时钟接口，监控累加计算代码的运行时间	52
6 实验六 设备管理	54
6.1 实验介绍	54
6.1.1 相关知识	54
6.1.2 任务描述	55
6.2 实验目的	56
6.3 实验任务	56
6.3.1 编写内核模块测试硬盘的写速率	56

6.3.2 编写内核模块测试硬盘的读速率	57
7 实验七 文件系统.....	58
7.1 实验介绍	58
7.1.1 任务描述	58
7.2 实验目的	58
7.3 实验任务	58
7.3.1 为 Ext4 文件系统添加扩展属性	58
7.3.2 注册一个自定义的文件系统类型	63
7.3.3 在/proc 下创建目录	65
7.3.4 使用 sysfs 文件系统传递内核模块参数	67
8 实验八 网络管理.....	71
8.1 实验介绍	71
8.1.1 任务描述	71
8.2 实验目的	71
8.3 实验任务	71
8.3.1 编写基于 socket 的 UDP 发送接收程序	71
8.3.2 使用 tshark 抓包	76
8.3.3 使用 setsockopt 发送记录路由选项	77

前言

鲲鹏云是“华为云鲲鹏云”的简称，本实验手册是基于鲲鹏云弹性云服务器（ECS）的 openEuler 操作系统内核编程实验手册，包括了内核编译、内存管理、进程管理、中断异常管理、内核时间管理、设备管理、文件管理以及网络管理等内核相关实验。

本实验手册不仅包含实验步骤，还在每章简要介绍了实验相关的原理和背景知识，以便读者能更好地理解操作系统内核的原理并进行实验。

一、实验网络环境介绍

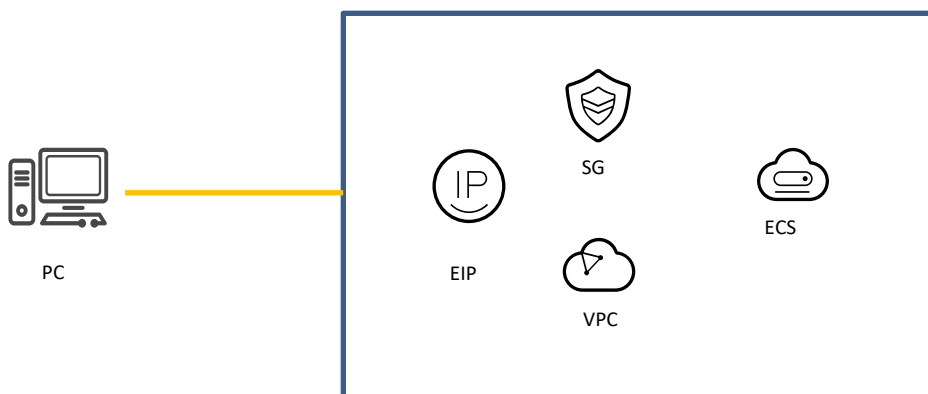


图1-1 openEuler 操作系统实验的云环境

如上图所示，本实验的云环境主要是一台基于鲲鹏架构的 ECS，附加部件有弹性公网 IP（EIP）、虚拟私有云（VPC）和安全组（SG）。学生端的 PC 机使用 ssh 终端登录 ECS。

二、实验设备介绍

关键配置如下表所示：

表1-1 关键配置

云资源	规格
ECS 鲲鹏计算	4vCPUs 8GB 40GB
EIP 带宽	按流量计费

软件方面，本实验需要一台终端电脑与弹性云服务器(ECS)链接以输入操作命令或/和传输文件。对于 Windows 10 / macOS / Linux，我们可以用命令行工具 ssh 和 scp 完成这个过程。

如果在有些 Windows 系统下不能运行 ssh 工具，也可以使用 Putty 和 WinSCP 工具软件。其中 Putty 工具的推荐下载地址：

<https://hcia.obs.cn-north-4.myhuaweicloud.com/v1.5/putty.exe>

WinSCP 的推荐下载地址：

<https://winscp.net/eng/index.php>

下文若无特殊说明，均以命令行工具 ssh 和 scp 为例进行讲解。

1 实验一 鲲鹏云 ECS 的构建及内核编译

1.1 实验介绍

本实验通过构建鲲鹏云 ECS、编译安装 openEuler 操作系统新内核以及简单的内核模块编程任务操作带领大家了解操作系统以及内核编程。

1.1.1 任务描述

- 构建鲲鹏云 ECS
- 编译安装 openEuler 操作系统新内核
- 简单的内核模块编程实验，在内核模块中打印 “Hello, world!”

1.2 实验目的

- 学习掌握如何安装构建 ECS
- 学习掌握如何编译操作系统内核
- 了解内核模块编程。

1.3 构建云实验环境

1.3.1 创建 VPC

步骤 1 在浏览器地址栏输入华为云控制台网址 console.huaweicloud.com 并按回车键，这时页面将跳转至登录页。



账号登录

账号名/邮箱

密码

手机号登录 ☒ 记住登录名

登录

免费注册 | 忘记密码 | IAM用户登录 | HUAWEI ID登录


使用其他账号登录

步骤 2 按要求输入账号密码，进行登录。

注意：在此之前您需要在华为云主页注册华为云账号。

步骤 3 登录成功后会自动进入控制台页面，这时将区域选在 “华北-北京四”。



步骤 4 将鼠标悬停于左侧导航栏  图标处展开服务列表，然后在服务列表中点击“虚拟私有云 VPC”项。



步骤 5 点击“虚拟私有云”控制台页面右上角的“创建虚拟私有云”按钮。



步骤 6 在创建虚拟私有云的页面中按照下表内容配置虚拟私有云参数。

参数	配置
区域	华北-北京四
名称	vpc-test
网段	192.168.1.0/24
企业项目	default
默认子网可用区	可用区1
默认子网名称	subnet-test
子网网段	如192.168.1.0/24

步骤 7 配置完成后，点击“立即创建”，创建完成后会自动回到 VPC 控制台。

步骤 8 点击 VPC 控制台左侧导航栏的“访问控制”→“安全组”，进入安全组控制台。



步骤 9 点击右上角的“创建安全组”。



步骤 10 在弹出的对话框中按“通用 Web 服务器”配置安全组参数，然后点击“确定”。

创建安全组

*

名称

sg-test

*

模板

通用Web服务器

描述

通用Web服务器，默认放通22、3389、80、443端口和ICMP协议。适用于需要远程登录、公网ping及用于网站服务的云服务器场景。


0/255

查看模板规则

确定

取消

1.3.2 购买 ECS

步骤 1 将鼠标悬停于左侧导航栏图标处展开服务列表。然后在服务列表中点击“弹性云服务器 ECS”项。



步骤 2 点击弹性云服务器 ECS 控制台页面右上角的“购买弹性云服务器 ECS”按钮进入购买页面。



步骤 3 按照下表内容配置弹性云服务器 ECS 的参数。

参数	配置
计费模式	按需计费
区域	华北-北京四
可用区	可用区1
CPU架构	鲲鹏计算
规格	鲲鹏通用计算增强型 kc1.xlarge.2 4vCPUs 8GB
镜像	公共镜像 openEuler openEuler 20.03 64bit with ARM(40GB)
系统盘	通用型SSD 40GB

注意：这里“区域”的配置是和 VPC 的区域配置保持一致的。

步骤 4 配置完成后点击“下一步：网络配置”，进入网络配置，按下表配置网络参数。

参数	配置
网络	vpc-test subnet-test 自动分配IP地址
安全组	sg-test
弹性公网IP	现在购买
线路	全动态BGP
公网带宽	按流量计费
带宽大小	5Mbit/s

步骤 5 配置完成后，点击“下一步：高级配置”，按下表配置 ECS 高级配置参数。

参数	配置
云服务器名称	openEuler（输入符合规则名称）
登录凭证	密码
密码	请输入8位以上包含大小写字母、数字和特殊字符的密码，如


	openEuler@123
确认密码	请再次输入密码
云备份	暂不购买
云服务器组	不配置
高级选项	不勾选

步骤 6 配置完成后点击右下角“下一步：确认配置”。勾选同意协议，然后点击：立即购买。

步骤 7 在提交任务成功后，点击“返回云服务器列表”，返回 ECS 控制台。

1.3.3 通过 ssh 登录系统

步骤 1 在 ECS 控制台查看 ECS 弹性公网 IP 地址。

<input type="checkbox"/>	名称/ID	监控	可用区	状态	规格/镜像	IP地址	计费...
<input type="checkbox"/>	openEuler 5e4b28a...		可用区1	 运行中	4vCPUs 8GB kc1.xlarge.2 openEuler 20.03 64bit with ARM	121.36... 192.168...	按需计费 2020/11/15 ...

步骤 2 在客户端机器操作系统里的 Console 控制台或 Terminal 终端里运行 ssh 命令：

```
$ ssh root@121.36.45.64
```

(注意：此处的 IP 地址 121.36.45.64 即是刚刚购买的弹性公网 IP 地址。)

在客户端（本地 PC）第一次登录时会有安全性验证的提示：

```
The authenticity of host '119.8.238.181 (119.8.238.181)' can't be established.
ECDSA key fingerprint is SHA256:RVxC1cSuMmqLtWdMw4n6f/VPsfWLKT/zDMT2q4qWxc0.
Are you sure you want to continue connecting (yes/no)? yes
```

在这里输入 yes 并按回车键继续：

```
Warning: Permanently added '119.8.238.181' (ECDSA) to the list of known hosts.
```

```
Authorized users only. All activities may be monitored and reported.
root@119.8.238.181's password:
```

输入密码(注意这里不会有任何回显)并回车，登录后的界面如下所示：

```
Welcome to Huawei Cloud Service
```

Last login: Mon May 18 15:35:37 2020

Welcome to Huawei Cloud Service

Last login: Mon May 18 15:35:37 2020

Welcome to 4.19.90-2003.4.0.0036.oe1.aarch64

System information as of time: Sun Nov 15 14:41:58 CST 2020

System load: 0.15
Processes: 131
Memory used: 5.0%
Swap used: 0.0%
Usage On: 9%
IP address: 192.168.1.5
Users online: 1

[root@openeuler ~]#

步骤 3 修改主机名

ECS 创建时被命名为 “openEuler” ,所以系统默认 hostname 为 “openeuler” , 为了和本实验手册另外两个版本保持行文上的一致, 我们可以将主机名改为 “openEuler” 或 “localhost” (一般在虚拟机中, 主机名被默认为 localhost, 而 ECS 也是虚拟机。本文的上下文环境中可能同时用到这三种名称, 请鉴别) :

```
[root@openeuler ~]# vi /etc/hostname
```

```
[root@openeuler ~]# cat /etc/hostname  
openEuler
```

```
[root@openeuler ~]# reboot
```

修改完成后重启系统并重新登录。

1.4 实验任务

1.4.1 openEuler 内核编译与安装

步骤 1 安装工具, 构建开发环境:

```
[root@openEuler ~]# yum group install -y "Development Tools"  
[root@openEuler ~]# yum install -y bc
```

```
[root@openEuler ~]# yum install -y openssl-devel
```

步骤 2 备份 boot 目录以防后续步骤更新内核失败

```
[root@openEuler ~]# tar czvf boot.origin.tgz /boot/
```

保存当前内核版本信息

```
[root@openEuler ~]# uname -r > uname_r.log
```

步骤 3 获取内核源代码并解压

```
[root@openEuler ~]# wget https://gitee.com/openeuler/kernel/repository/archive/kernel-4.19.zip
```

```
[root@openEuler ~]# unzip kernel-4.19.zip
```

步骤 4 编译内核

```
[root@openEuler ~]# cd kernel
```

```
[root@openEuler kernel]# make openeuler_defconfig
```

在这里，我们按源代码文件 kernel/arch/arm64/configs/openeuler_defconfig 的配置配置内核。

```
[root@openEuler kernel]# make help | grep Image
```

```
* Image.gz      - Compressed kernel image (arch/arm64/boot/Image.gz)
Image           - Uncompressed kernel image (arch/arm64/boot/Image)
```

这一步查看了可编译的 Image。

```
[root@openEuler kernel]# make -j4 Image modules dtbs
```

这一步是编译内核的 Image、modules 和 dtbs。

步骤 5 安装内核

```
[root@openEuler kernel]# make modules_install
```

```
.....
```

```
INSTALL sound/soundcore.ko
```

```
DEPMOD 4.19.154
```

```
[root@openEuler kernel]# make install
```

```
/bin/sh ./arch/arm64/boot/install.sh 4.19.154 \
```

```
arch/arm64/boot/Image System.map "/boot"
```



```
dracut-install: Failed to find module 'xen-blkfront'
```

```
dracut: FAILED: /usr/lib/dracut/dracut-install -D /var/tmp/dracut.tlIdPu/initramfs --kernel-dir
```

```
/lib/modules/4.19.154/ -m virtio_gpu xen-blkfront xen-netfront virtio_blk virtio_scsi virtio_net virtio_pci virtio_ring virtio
```

注意：在最后一步“make install”时出现的错误在这里可以忽略。

步骤 6 以 VNC 登录 ECS

<input type="checkbox"/>	名称/ID	监控	可用区 	状态 	规格/镜像
<input type="checkbox"/>	openEuler 5e4b28a...				openEuler 5e4b28a8-e74c-4e62-8df5-f911704b7df1 20.03 64bit with ARM

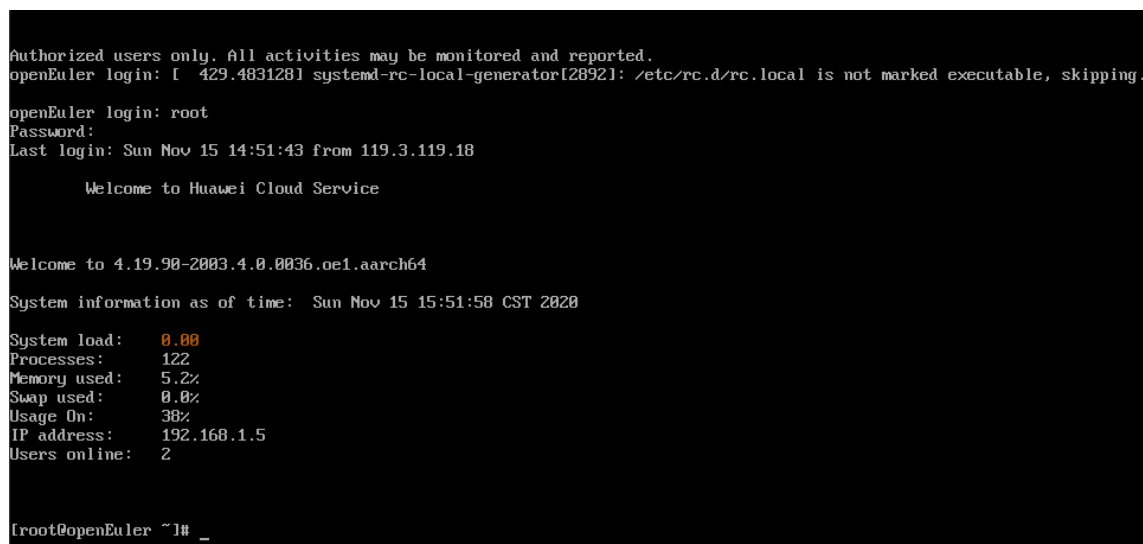
在控制台“弹性云服务器 ECS”的页面中点击刚刚创建的虚拟机“openEuler”的名字超链接，在新打开的页面中点击“远程登录”按钮：



然后以控制台提供的 VNC 方式登录：



与以 ssh 登录一样，以 root 身份登录：



大部分的时间，我们仅将此作为一个监视器使用。

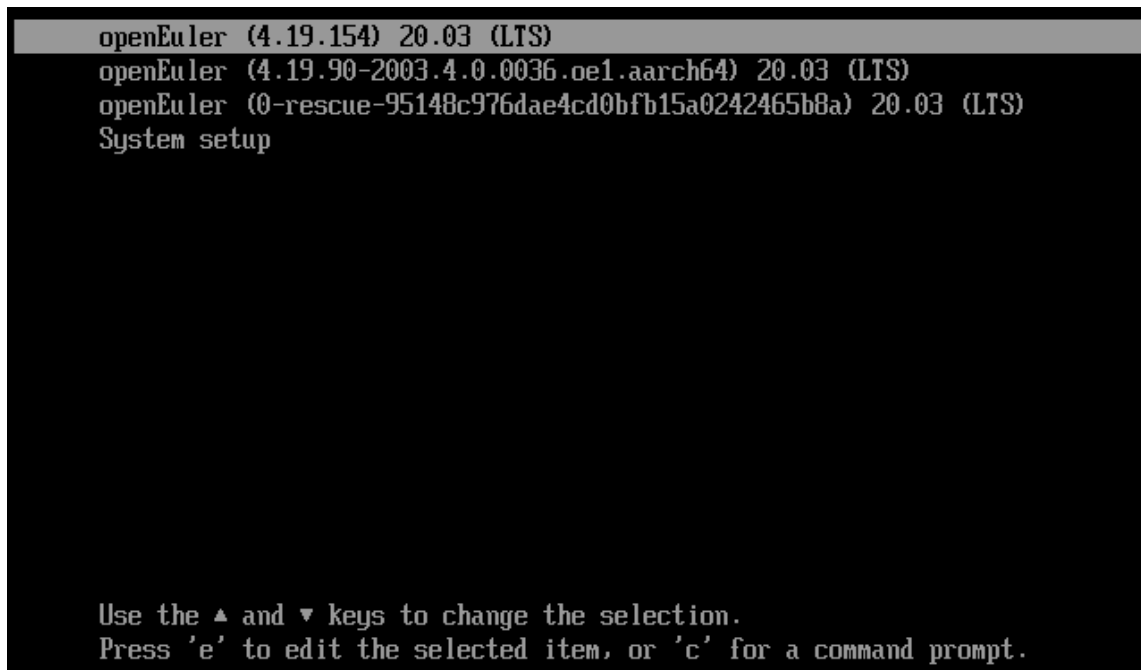
步骤 7 重启系统

在 ssh 终端重启操作系统：


```
[root@openEuler kernel]# reboot
```

步骤 8 登录并验证

在 VNC 窗口中选择以新编译出来的内核启动系统：



在这里新编译出来的内核版本为 4.19.154。您的子版本号可能与此不一样。

步骤 9 登录系统并查看版本

请以 VNC 和 ssh 终端登录系统，并在其中之一查看内核版本：

```
[root@openEuler ~]# uname -r
4.19.154
```

可以看出内核版本已更新。

1.4.2 Hello, world!

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/1/task3 目录下。

```
[root@openEuler ~]# cd tasks_k/1/task3
[root@openEuler task1]# ls
helloworld.c Makefile
```



实验中的源文件可以参考以上压缩包中内容（您可以用 scp 命令将其上传到 ECS）。

步骤 2 编译源文件

```
[root@openEuler task3]# make
make -C /root/kernel M=/root/tasks_k/1/task3 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/1/task3/helloworld.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/1/task3/helloworld.mod.o
LD [M] /root/tasks_k/1/task3/helloworld.ko
make[1]: Leaving directory '/root/kernel'
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task3]# insmod helloworld.ko
[root@openEuler task3]# lsmod | grep helloworld
helloworld                262144  0
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task3]# rmmod helloworld
[root@openEuler task3]# dmesg | tail -n5
[ 708.247970] helloworld: loading out-of-tree module taints kernel.
[ 708.248513] helloworld: module verification failed: signature and/or required key missing - tainting kernel
[ 708.249859] hello_init
[ 708.250043] Hello, world!
[ 747.518247] hello_exit
```

您在 VNC 窗口中也会看到同样的结果。

(在这里，请忽略掉最开始安装模块时出现的两行错误提示信息。)

步骤 5 在虚拟机和 VNC 窗口中退出登录

```
[root@openEuler ~]# exit
```

注意：一般情况下不要 shutdown 虚拟机，若 shutdown 了虚拟机，需要联系实验管理员重启该虚拟机。

步骤 6 关闭 VNC 客户端页面

1.5 云环境资源清理

1.5.1 ECS 关机

当所实验完成后，应该对 ECS 进行关机以节约经费（ECS 关机后仍有少量扣费）。

步骤 1 回到 ECS 控制台，勾选 openEuler 虚拟机，进行关机。



在弹出的对话框中点击“是”按钮：

关机



确定要对以下云服务器进行关机操作吗？

- 1、按需/竞价计费（竞价计费模式）实例关机后，基础资源（vCPU、内存、镜像）不再计费，绑定的云硬盘（包括系统盘、数据盘）、弹性公网IP、带宽等资源按各自产品的计费方法（“包年/包月”或“按需计费”）进行收费。云服务器再次开机时，可能由于基础资源不足无法正常开机。请耐心等待，稍后重试。
- 2、包含本地盘（如磁盘增强型、GPU加速型等）和包含FPGA卡的按需/竞价计费实例，以及竞价计费的共享模式实例，关机后仍然计费。如需停止计费，请删除实例。

名称	状态	备注
openEuler	运行中	--

☐ 强制关机

强制关机会导致云服务器中未保存的数据丢失，请谨慎操作。

是

否

点击“是”按钮即可进行关机。

1.5.2 删除 ECS

可以等到所有的内核实验完成后再删除 ECS，否则每次都得重新编译内核。这里给出删除 ECS 的方法。

步骤 1 待关机完成后点击“更多” → “删除”



在弹出的对话框中勾选“释放云服务器绑定的弹性公网 IP 地址”和“删除云服务器挂载的数据盘”，然后点击“是”，删除 ECS。



步骤 2 您可以在控制台点击“更多 | 资源 | 我的资源”菜单项，检查资源是否全部删除



注意：(1) 虚拟私有云 VPC 和安全组可以不删除，以留下次使用。(2) 若在除“华北-北京四”之外区域购买了 ECS 和 EIP，请切换到那个区域查看。

2 实验二 内存管理

2.1 实验介绍

本实验通过在内核态分配内存的任务操作，让学生们了解并掌握操作系统中内存管理的布局，内核态内存分配的实现，以及内核模块的加载、卸载。

2.1.1 相关知识

一、kmalloc()和vmalloc()分配的是内核态的内存分配函数。

kmalloc()

功能：在设备驱动程序或者内核模块中动态分配内存。

函数原型：static __always_inline void *kmalloc(size_t size, gfp_t flags)

头文件：#include <linux/slab.h>

参数说明：

size：要分配内存的大小，以字节为单位。

flags：要分配的内存类型。如：GFP_USER（代表用户分配内存）、GFP_KERNEL（分配内核内存）、GFP_ATOMIC 等（更多请参考 linux/gfp.h）

返回值：分配成功时，返回分配的虚拟地址；分配失败时，返回 NULL。

特点：

分配的内存存在物理上是连续的（这对于要进行 DMA 的设备十分重要），用于小内存分配。

最多只能分配 32*PAGESIZE 大小的内存。

最小处理 32 字节或者 64 字节的内存块。

分配速度较快，内核中主要的内存分配方法。

使用完之后，用 kfree() 释放内存：void kfree(const void *);

vmalloc()

功能：在设备驱动程序或者内核模块中动态分配内存。

函数原型：void *vmalloc(unsigned long size)

头文件：#include <linux/vmalloc.h>

参数说明：

size：要分配内存的大小，以字节为单位。

返回值：分配成功时，返回分配的虚拟地址；分配失败时，返回 NULL。

特点：

分配的内存：虚拟地址连续，物理地址不连续。

最小处理 4KB 的内存块。

分配速度较慢，一般用于大块内存的分配。

使用完之后，用 `vfree()` 释放内存：`void vfree(const void *addr)`

二、内存布局

不同的体系架构，内存布局各不相同，在内核源码的 `Document` 目录下，有部分架构关于内核布局的详细描述，如：

arm64: `Documentation/arm64/memory.txt`

三、内核模块编程

源码编写——.c 源文件

Makefile 文件编写

编译模块——make

模块加载进内核——insmod

查看加载的内容——dmesg

查看内核模块——lsmod

卸载内核模块——rmmod

2.1.2 任务描述

- 使用 `kmalloc` 分配 1KB、8KB 的内存，打印指针地址；
- 使用 `vmalloc` 分配 8KB、1MB、64MB 的内存，打印指针地址；
- 查看已分配的内存，根据机器是 32 位或 64 位的情况，分析地址落在的区域。

2.2 实验目的

- 掌握正确编写满足功能的源文件，正确编译。
- 掌握正常加载、卸载内核模块；且内核模块功能满足任务所述。
- 了解操作系统的内存管理。

2.3 实验任务

2.3.1 使用 kmalloc 分配 1KB，8KB 的内存，打印指针地址

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/2/task1 目录下。

```
[root@openEuler ~]# cd tasks_k/2/task1
[root@openEuler task1]# ls
kmalloc.c  Makefile
```

步骤 2 编译源文件

```
[root@openEuler task1]# make
make -C /root/kernel M=/root/tasks_k/2/task1 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/2/task1/kmalloc.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/2/task1/kmalloc.mod.o
LD [M] /root/tasks_k/2/task1/kmalloc.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task1]# ls
kmalloc.c  kmalloc.ko  kmalloc.mod.c  kmalloc.mod.o  kmalloc.o  Makefile  modules.order  Module.symvers
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task1]# insmod kmalloc.ko
[root@openEuler task1]# dmesg | tail -n3
[12892.541517] Start kmalloc!
[12892.541688] kmallocmem1 addr = ffff80017407b400
[12892.541920] kmallocmem2 addr = ffff80016a44a000
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task1]# rmmod kmalloc
[root@openEuler task1]# dmesg | tail -n4
[12892.541517] Start kmalloc!
[12892.541688] kmallocmem1 addr = ffff80017407b400
[12892.541920] kmallocmem2 addr = ffff80016a44a000
[12994.315305] Exit kmalloc!
[root@openEuler task1]#
```

2.3.2 使用 vmalloc 分配 8KB、1MB、64MB 的内存，打印指针地址

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/2/task1 目录下。


```
[root@openEuler ~]# cd tasks_k/2/task2
[root@openEuler task2]# ls
vmalloc.c  Makefile
```

步骤 2 编译源文件

```
[root@openEuler task2]# make
make -C /root/kernel M=/root/tasks_k/2/task2 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/2/task2/vmalloc.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/2/task2/vmalloc.mod.o
LD [M] /root/tasks_k/2/task2/vmalloc.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task2]# ls
Makefile  modules.order  Module.symvers  vmalloc.c  vmalloc.ko  vmalloc.mod.c  vmalloc.mod.o  vmalloc.o
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task2]# insmod vmalloc.ko
[root@openEuler task2]# dmesg | tail -n4
[20608.083498] Start vmalloc!
[20608.083718] vmallocmem1 addr = ffff000009dc0000
[20608.083953] vmallocmem2 addr = ffff000022480000
[20608.084370] vmallocmem3 addr = ffff0000242c0000
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task2]# rmmod vmalloc
[root@openEuler task2]# dmesg | tail -n5
[20608.083498] Start vmalloc!
[20608.083718] vmallocmem1 addr = ffff000009dc0000
[20608.083953] vmallocmem2 addr = ffff000022480000
[20608.084370] vmallocmem3 addr = ffff0000242c0000
[20759.537586] Exit vmalloc!
```

2.3.3 实验结果分析

通过 `uname -a` 或 `file /sbin/init` 或 `arch` 命令查看当前是 ARM64 位的机器。使用“`getconf PAGE_SIZE`”查看系统的页表大小；或者通过查看内核配置选项，内核配置文件是：`arch/arm64/configs/openeuler_defconfig`，其中与内存管理相关的配置内容是（可通过 `vi` 打开配置文件，关键词搜索定位）：

```
CONFIG_ARM64=y
CONFIG_64BIT=y
CONFIG_PGTABLE_LEVELS=3
CONFIG_ARM64_PAGE_SHIFT=16
CONFIG_ARM64_64K_PAGES=y
CONFIG_ARM64_VA_BITS_48=y
```

也就是说，目前的配置是：

虚拟地址位数为 48 位（CONFIG_ARM64_VA_BITS=48、CONFIG_ARM64_VA_BITS=48）；

页表的大小是 64K（CONFIG_ARM64_PAGE_SHIFT=16、CONFIG_ARM64_64K_PAGES=y）；

页表转化是 3 级（CONFIG_PGTABLE_LEVELS=3）。

因此查看内核源码的文档 Documentation/arm64/memory.txt 可见，对应的内存布局如下：

AArch64 Linux memory layout with 64KB pages + 3 levels:			
Start	End	Size	Use
0000000000000000	0000ffffffffffffff	256TB	user
ffff000000000000	ffffffffffffffffffff	256TB	kernel

由运行结果可知，kmalloc 和 vmalloc 分配的内存地址，都位于内核空间。

2.4 思考题

一、什么是内存泄漏、内存溢出、内存越界？

内存泄漏（memory leak）：程序中已动态分配的内存未释放或无法释放，就产生了内存泄露。

内存溢出（out of memory）：程序在申请内存时，没有足够的内存空间供其使用。

内存越界：是指程序向系统申请一块内存后，使用时超出申请范围。

二、分析程序

1、分析：下面这个程序是否会产生内存泄露、内存溢出或内存越界？

```
#include <stdlib.h>
void function_which_allocates(void)
{
    float *a = malloc(sizeof(float) * 45);    // 分配包含 45 个浮点数的数组
    /* 使用数组 a 的代码 */
}
int main(void)
{
    function_which_allocates();
    /* 指针 a 已经不存在了，所以包含 45 个浮点数的数组就不能被释放了，但这个数组还存在于内存中，因此
    就造成了内存泄露*/
}
```

分析：内存泄漏，已动态分配的内存未释放。

2、分析：下面这个程序是否会产生内存泄露、内存溢出或内存越界？

```
void func(char * input)
```

```
{
char buffer[16];
strcpy(buffer, input);
}
void main()
{
char longstring[256];
int i;
for( i = 0; i < 255; i++)
    longstring [i] = 'B';
func(longstring);
}
```

分析：内存越界：上述代码中，strcpy()直接将 input 中的内容 copy 到 buffer 中。而 main 函数中，传入 input 的长度是 256，大于 buffer 的长度，造成 buffer 越界，使程序运行出错。

3 实验三 进程管理

3.1 实验介绍

本实验通过在内核态创建进程，读取系统 CPU 负载，打印系统当前运行进程 PID 以及使用 cgroup 限制 CPU 核数等任务操作，让学生们了解并掌握操作系统中的进程管理。

3.1.1 任务描述

- 编写内核模块，创建一个内核线程；并在模块退出时杀死该线程。
- 编写一个内核模块，实现读取系统一分钟内的 CPU 负载。
- 编写一个内核模块，打印当前系统处于运行状态的进程的 PID 和名字。
- 使用 cgroup 实现限制 CPU 核数。

3.2 实验目的

- 掌握正确编写满足功能的源文件，正确编译。
- 掌握正常加载、卸载内核模块；且内核模块功能满足任务所述。
- 了解操作系统的进程管理。

3.3 实验任务

3.3.1 创建内核进程

3.3.1.1 相关知识

一、内核线程介绍

内核经常需要在后台执行一些操作，这种任务就可以通过内核线程（kernel thread）完成，内核线程是指独立运行在内核空间的标准进程。内核线程和普通的进程间的区别在于：内核线程没有独立的地址空间，mm 指针被设置为 NULL；它只在内核空间运行，从来不切换到用户空间去；并且和普通进程一样，可以被调度，也可以被抢占。

内核线程只能由其它的内核线程创建，Linux 内核通过给出的函数接口与系统中的初始内核线程 kthreadd 交互，由 kthreadd 衍生出其它的内核线程。

二、相关接口函数

1、kthread_create():

函数返回一个 task_struct 指针，指向代表新建内核线程的 task_struct 结构体。注意：使用该函数创建的内核线程处于不可运行状态，需要将 kthread_create 返回的 task_struct 传递给 wake_up_process 函数，通过此函数唤醒新建的内核线程。

2、kthread_run()

头文件：<linux/kthread.h>

函数原型：struct task_struct *kthread_run(int (*threadfn)(void *data), void *data, const char *namefmt, ...);

功能：创建并启动一个线程。

参数：int (*threadfn)(void *data)----->线程函数，指定该线程要完成的任务。这个函数会一直运行直到接收到终止信号。

void *data----->线程函数的参数。

const char *namefmt----->线程名字。

3、线程函数

用户在线程函数中指定要让该线程完成的任务。该函数会一直运行，直到接收到结束信号。因此函数中需要有判断是否收到信号的语句。

```
static int func(void *data){
    while(!kthread_should_stop()){
        . . . . . 一些工作
        msleep(2000);
    }
    return 0;
}
```

注意在线程函数中需要在每一轮迭代之后休眠一定时间，让出 CPU 给其他的任务，否则创建的这个线程会一直占用 CPU，使得其他任务均瘫痪。更严重的是，使线程终止的命令也无法执行，导致这种状态一直持续下去。

4、kthread_stop():

头文件：<linux/kthread.h>

函数原型：int kthread_stop(struct task_struct *k);

功能：在模块卸载时，发送信号给 k 指向的线程，使之退出。

线程一旦启动起来之后，会一直运行，除非该线程主动调用 `do_exit` 函数，或者其他的进程调用 `kthread_stop` 函数，结束线程的运行。当然，如果线程函数永远不返回，并且不检查信号，它将永远不会停止。因此线程函数信号检查语句以及返回值非常重要。

注意，在调用 `kthread_stop` 函数时，线程不能已经结束运行，否则，`kthread_stop` 函数会一直等待。

5、`kthread_should_stop()`:

头文件: `<linux/kthread.h>`

函数原型: `bool kthread_should_stop(void);`

功能: 该函数位于内核线程函数体内，与 `kthread_stop` 配合使用，用于接收 `kthread_stop` 传递的结束线程信号，如果内核线程中未用此函数，则 `kthread_stop` 使其结束。

3.3.1.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 `tasks_k/3/task1` 目录下。

```
[root@openEuler ~]# cd tasks_k/3/task1
[root@openEuler task1]# ls
kthread.c  Makefile
```

步骤 2 编译源文件

```
[root@openEuler task1]# make
make -C /root/kernel M=/root/tasks_k/3/task1 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/3/task1/kthread.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/3/task1/kthread.mod.o
LD [M] /root/tasks_k/3/task1/kthread.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task1]# ls
kthread.c  kthread.ko  kthread.mod.c  kthread.mod.o  kthread.o  Makefile  modules.order  Module.symvers
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task1]# insmod kthread.ko
[root@openEuler task1]# dmesg | tail -n5
[23146.801386] Create kernel thread!
[23146.801978] New kthread is running.
[23148.811025] New kthread is running.
[23150.826971] New kthread is running.
[23152.842938] New kthread is running.
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task1]# rmmod kthread
[root@openEuler task1]# dmesg | tail -n5
[23318.152447] New kthread is running.
[23320.168408] New kthread is running.
[23322.184379] New kthread is running.
[23324.200342] New kthread is running.
[23324.484171] Kill new kthread.
```

3.3.2 打印输出当前系统 CPU 负载情况

3.3.2.1 相关知识

一、proc 文件系统

1、proc 文件简介

proc 文件系统是 Linux 中的特殊文件系统，提供给用户一个可以了解内核内部工作过程的可读窗口，在运行时访问内核内部数据结构、改变内核设置的机制。

proc 文件系统能够保存系统当前工作的特殊数据，但并不存在于任何物理设备中，对其进行读写时，才根据系统中的相关信息即时生成。所有 proc 文件挂载在 /proc 目录下。

/proc 的文件可以用于访问有关内核的状态、计算机的属性、正在运行的进程的状态等信息。大部分 /proc 中的文件和目录提供系统物理环境最新的信息。它们实际上并不存在磁盘上，也不占用任何空间。（用 ls -l 可以显示它们的大小）当查看这些文件时，实际上是在访问存在内存中的信息，这些信息用于访问系统。

尽管 /proc 中的文件是虚拟的，但它们仍可以使用任何文件编辑器或像 'more'、'less' 或 'cat' 这样的程序来查看。当编辑程序试图打开一个虚拟文件时，这个文件就通过内核中的信息被凭空地创建了。

2、proc 文件组成

(1) 有用内核信息

proc 文件系统可以被用于收集有用的关于系统和运行中的内核的信息。下面是一些重要的文件：

/proc/cpuinfo	----- CPU 的信息（型号，家族，缓存大小等）
/proc/meminfo	----- 物理内存、交换空间等的信息
/proc/loadavg	----- 查看系统 1 分钟、5 分钟、15 分钟的平均负载情况
/proc/mounts	----- 已加载的文件系统的列表
/proc/devices	----- 可用设备的列表
/proc/filesystems	----- 被支持的文件系统
/proc/modules	----- 已加载的模块

/proc/version ----- 内核版本

/proc/cmdline ----- 系统启动时输入的内核命令行参数

proc 中的文件远不止上面列出的这么多。想要进一步了解，可以对/proc 的每一个文件都‘more’一下。

(2) 进程相关信息

/proc 文件系统可以用于获取运行中的进程的信息。在/proc 中有一些编号的子目录。每个编号的目录对应一个进程 id(PID)。这样，每一个运行中的进程/proc 中都有一个用它的 PID 命名的目录。这些子目录中包含可以提供有关进程的状态和环境的重要细节信息的文件。

(3) 通过 proc 文件与内核交互

上面讨论的大部分/proc 的文件是只读的。而实际上/proc 文件系统通过/proc 中可读写的文件提供了对内核的交互机制。写这些文件可以改变内核的状态，因而要慎重改动这些文件。

/proc/sys ----- 目录存放所有可读写的文件的目录，可以被用于改变内核行为。

/proc/sys/kernel ----- 这个目录包含通用内核行为的信息。

/proc/sys/kernel/{domainname, hostname} 存放着机器/网络的域名和主机名。

二、内核中读写文件数据的方法

有时候需要在 Linux kernel 中读写文件数据，如调试程序的时候，或者内核与用户空间交换数据的时候。在 kernel 中操作文件没有标准库可用，需要利用 kernel 的一些函数，这些函数主要有：

filp_open() 在 kernel 中打开指定文件。

filp_close() kernel 中文件的读操作。

kernel_read() kernel 中文件的写操作。

kernel_write() 关闭指定文件。

这些函数在 <linux/fs.h> 头文件中声明。具体读写接口说明如下：

3.3.2.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/3/task2 目录下。

```
[root@openEuler ~]# cd tasks_k/3/task2
[root@openEuler task2]# ls
cpu_loadavg.c  Makefile
```

步骤 2 编译源文件

```
[root@openEuler task2]# make
make -C /root/kernel M=/root/tasks_k/3/task2 modules
make[1]: Entering directory '/root/kernel'
```



```
CC [M] /root/tasks_k/3/task2/cpu_loadavg.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/3/task2/cpu_loadavg.mod.o
LD [M] /root/tasks_k/3/task2/cpu_loadavg.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task2]# ls
cpu_loadavg.c  cpu_loadavg.ko  cpu_loadavg.mod.c  cpu_loadavg.mod.o  cpu_loadavg.o  Makefile
modules.order  Module.symvers
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task2]# insmod cpu_loadavg.ko
[root@openEuler task2]# dmesg | tail -n2
[27644.911012] Start cpu_loadavg!
[27644.911209] The cpu loadavg in one minute is: 0.01
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task2]# rmmod cpu_loadavg
[root@openEuler task2]# dmesg | tail -n3
[27644.911012] Start cpu_loadavg!
[27644.911209] The cpu loadavg in one minute is: 0.01
[27686.382949] Exit cpu_loadavg!
```

3.3.3 打印输出当前处于运行状态的进程的 PID 和名字

3.3.3.1 相关知识

当前进程在在 /proc 文件系统也有保存，只不过需要遍历所有进程文件夹，从 stat 文件中读取状态，来判定是否为当前运行进程。而内核中可用进程遍历函数来遍历所有进程，且进程描述符 task_struct 结构里边有 state 状态，state 为 0 的进程就是当前进程。

1、进程描述符 task_struct

系统中存放进程的管理和控制信息的数据结构称为进程控制块 PCB（Process Control Block），是进程管理和控制的最重要的数据结构。

每一个进程均有一个 PCB，在创建进程时，建立 PCB，伴随进程运行的全过程，直到进程撤消而撤消。

在 Linux 中，每一个进程都有一个进程描述符 task_struct，也就是 PCB；task_struct 结构体是 Linux 内核的一种数据结构，它会被装载到 RAM 里并包含每个进程所需的所有信息。是对进程控制的唯一手段也是最有效的手段。

task_struct 定义在 <linux/ sched.h> 头文件中。

2、for_each_process

`for_each_process` 是一个宏，定义在`<linux/sched/signal.h>`文件中，提供了依次访问整个任务队列的能力。

3.3.3.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 `tasks_k/3/task3` 目录下。

```
[root@openEuler ~]# cd tasks_k/3/task3
[root@openEuler task3]# ls
Makefile  process_info.c
```

步骤 2 编译源文件

```
[root@openEuler task3]# make
make -C /root/kernel M=/root/tasks_k/3/task3 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/3/task3/process_info.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/3/task3/process_info.mod.o
LD [M] /root/tasks_k/3/task3/process_info.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task3]# ls
Makefile  modules.order  Module.symvers  process_info.c  process_info.ko  process_info.mod.c
process_info.mod.o  process_info.o
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task3]# insmod process_info.ko
[root@openEuler task3]# dmesg | tail -n3
[27874.701269] 1)name:insmod 2)pid:14142 3)state:0
[27874.701507] 1)name:systemd-udevd 2)pid:14143 3)state:0
[27874.701774] 1)name:(spawn) 2)pid:14144 3)state:0
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task3]# rmmod process_info
[root@openEuler task3]# dmesg | tail -n4
[27874.701269] 1)name:insmod 2)pid:14142 3)state:0
[27874.701507] 1)name:systemd-udevd 2)pid:14143 3)state:0
[27874.701774] 1)name:(spawn) 2)pid:14144 3)state:0
[27894.806272] Exit process_info!
```

3.3.4 使用 cgroup 实现限制 CPU 核数

3.3.4.1 相关知识

一、cgroup 介绍

cgroup (Control Groups) 是将任意进程进行分组化管理的 Linux 内核功能，提供将进程进行分组化管理的功能和接口的基础结构。I/O 或内存的分配控制等具体的资源管理功能是通过这个功能来实现的，这些具体的资源管理功能称为 cgroup 子系统或控制器。

cgroup 的机制是：它以分组的形式对进程使用系统资源的行为进行管理和控制。也就是说，用户通过 cgroup 对所有进程进行分组，再对该分组整体进行资源的分配和控制。

cgroup 中的每个分组称为进程组，它包含多个进程。最初情况下，系统内的所有进程形成一个进程组（根进程组），根据系统对资源的需求，这个根进程组将被进一步细分为子进程组，子进程组内的进程是根进程组内进程的子集。而这些子进程组很有可能继续被进一步细分，最终，系统内所有的进程组形成一颗具有层次等级（hierarchy）关系的进程组树。

如果某个进程组内的进程创建了子进程，那么该子进程默认与父进程处于同一进程组中。也就是说，cgroup 对该进程组的资源控制同样作用于子进程。比如，我们限制进程的 CPU 使用为 20%，我们就可以建一个 cpu 占用为 20% 的 cgroup，然后将进程添加到这个 cgroup 中。当然，一个 cgroup 可以有多个进程。

cgroup 提供了一个 cgroup 虚拟文件系统，作为进行分组管理和各子系统设置的用户接口。要使用 cgroup，必须挂载 cgroup 文件系统。这时通过挂载选项指定使用哪个子系统。

cgroup 为每种可以控制的资源定义了一个子系统。典型的子系统介绍如下：

- 1) cpu 子系统：该子系统为每个进程组设置一个使用 CPU 的权重值，以此来管理进程对 CPU 的访问，限制进程的 CPU 使用率。
- 2) cpuset 子系统：对于多核 CPU，该子系统可以设置进程组只能在指定的核上运行，并且还可以设置进程组在指定的内存节点上申请内存。如果要使用 cpuset 控制器，需要同时配置 cpuset.cpus 和 cpuset.mems 两个文件（参数）。cpuset.mems 用来设置内存节点的；cpuset.cpus 用来限制进程可以使用的 CPU 核心；这两个参数中 CPU 核心、内存节点都用 id 表示，之间用 “,” 分隔，比如 0,1,2；也可以用 “-” 表示范围，如 0-3；两者可以结合起来用。如 “0-2,6,7”。在添加进程前，cpuset.cpus、cpuset.mems 必须同时设置，而且必须是兼容的，否则会出错。
- 3) cpuacct 子系统：该子系统只用于生成当前进程组内的进程对 CPU 的使用报告。
- 4) memory 子系统：该子系统提供了以页面为单位对内存的访问，比如对进程组设置内存使用上限等，同时可以生成内存资源报告。
- 5) blkio 子系统：该子系统用于限制每个块设备的输入输出（比如物理设备（磁盘，固态硬盘，USB 等等）。首先，与 CPU 子系统类似，该系统通过为每个进程组设置权重来控制块设备对其的 I/O 时间；其次，该子系统也可以限制进程组的 I/O 带宽以及 IOPS。
- 6) devices 子系统：通过该子系统可以限制进程组对设备的访问，即该允许或禁止进程组对某设备的访问。
- 7) freezer 子系统：该子系统可以使得进程组中的所有进程挂起或恢复。

8) net_cls 子系统：该子系统使用等级识别符标记网络数据包，可允许 Linux 流量控制程序识别从具体 cgroup 中生成的数据包，提供对网络带宽的访问限制，比如对发送带宽和接收带宽进程限制。

9) ns 子系统：名称空间子系统，可以使不同 cgroups 下面的进程使用不同的 namespace。

针对运行中的内核而言，可以使用的 cgroup 子系统由 /proc/cgroup 来确认。

```
[root@openEuler ~]# cat /proc/cgroups
#subsys_name hierarchy num_cgroups enabled
cpuset 12 1 1
cpu 6 1 1
cpuacct 6 1 1
blkio 13 1 1
memory 11 77 1
devices 9 35 1
freezer 7 1 1
net_cls 5 1 1
perf_event 3 1 1
net_prio 5 1 1
hugetlb 10 1 1
pids 4 41 1
rdma 2 1 1
files 8 1 1
[root@openEuler ~]#
```

cgroups 最初的目标是为资源管理提供的一个统一的框架，既整合现有的 cpuset 等子系统，也为未来开发新的子系统提供接口。现在的 cgroups 适用于多种应用场景，从单个进程的资源控制，到实现操作系统层次的虚拟化。

cgroups 提供了以下功能：

限制进程组可以使用的资源数量。比如：memory 子系统可以为进程组设定一个 memory 使用上限，一旦进程组使用的内存达到限额再申请内存，就会出发 OOM。

进程组的优先级控制。比如：可以使用 cpu 子系统为某个进程组分配特定 cpu share。

记录进程组使用的资源数量。比如：可以使用 cpuacct 子系统记录某个进程组使用的 cpu 时间。

进程组隔离。比如：使用 ns 子系统可以使不同的进程组使用不同的 namespace，以达到隔离的目的，不同的进程组有各自的进程、网络、文件系统挂载空间。

进程组控制。比如：使用 freezer 子系统可以将进程组挂起和恢复。

二、tmpfs 文件系统

tmpfs 即临时文件系统，是一种基于内存的文件系统，也称之为虚拟内存文档系统。它不同于传统的用块设备形式来实现的 ramdisk，也不同于针对物理内存的 ramfs。tmpfs 能够使用物理内存，也能够使用交换分区。

在 Linux 内核中，虚拟内存资源由物理内存（RAM）和交换分区 swap 组成，这些资源是由内核中的虚拟内存子系统来负责分配和管理。tmpfs 就是和虚拟内存子系统来"打交道"的，

他向虚拟内存子系统请求页来存储文档，他同 Linux 的其他请求页的部分相同，不知道分配给自己的页是在内存中还是在交换分区中。tmpfs 同 ramfs 相同，其大小也不是固定的，而是随着所需要的空间而动态的增减。

所有在 tmpfs 上储存的资料在理论上都是临时存放的，也就是说，档案不会建立在硬盘上面。一旦重新开机，所有在 tmpfs 里面的资料都会消失不见。理论上，内存使用量会随着 tmpfs 的使用而时有增长或消减。tmpfs 将所有内容放入内核内部高速缓存中，并进行扩展和收缩以容纳其中包含的文件，并且能够将不需要的页面交换出来以交换空间。它具有最大大小限制，可以通过 “mount -o remount” 即时调整。

由于 tmpfs 完全存在于页面缓存和交换中，因此所有 tmpfs 页面将在 /proc/meminfo 中显示为 “Shmem”，在 free 命令后中显示为 “Shared”。请注意，这些计数器还包括共享内存 (shmem)。获取计数的最可靠方法是使用 df 和 du。

```
[root@openEuler ~]# cat /proc/meminfo | grep 'Shmem'
Shmem:                13312 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
[root@openEuler ~]# free
              total        used        free      shared  buff/cache   available
Mem:      6975488       321408       6089088        13312       564992       6297728
Swap:              0              0              0
```

```
[root@openEuler ~]# df -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs        3.1G   0    3.1G   0% /dev
tmpfs           3.4G   0    3.4G   0% /dev/shm
tmpfs           3.4G  13M   3.4G   1% /run
tmpfs           3.4G   0    3.4G   0% /sys/fs/cgroup
/dev/vda2       39G   14G   23G   38% /
tmpfs           3.4G   64K   3.4G   1% /tmp
/dev/vda1      1022M   5.8M 1017M   1% /boot/efi
tmpfs           682M   0    682M   0% /run/user/0
```

使用 tmpfs，首先您编译内核时得选择 “虚拟内存文档系统支持 (Virtual memory filesystem support)” 或设置 CONFIG_TMPFS=y，然后就能够加载 tmpfs 文档系统了。

详细的介绍，可参见内核源码中的官方文档：Documentation/filesystems/tmpfs.txt。

可挂载 tmpfs 格式的 cgroup 文件夹进行 cgroup 的相关操作。

三、相关命令

mount 命令：加载指定的文件系统。

echo 命令：显示文字。

taskset 命令：依据线程 PID (TID) 查询或设置线程的 CPU 亲和性 (即与哪个 CPU 核心绑定)。

cgexec 命令：在指定的 cgroup 中运行任务。

3.3.4.2 实验步骤

步骤 1 安装 libcgroup: dnf install libcgroup -y

libcgroup 包含 cgroup 用户空间工具套件（如 lscgroup, lssubsys 等）以及静态或者动态库，以供其他程序调用，并且包含 debug 套件。

步骤 2 挂载 tmpfs 格式的 cgroup 文件夹

在 root 权限下执行以下命令：

```
# mkdir /cgroup
# mount -t tmpfs tmpfs /cgroup
# cd /cgroup
```

挂载 tmpfs 文件类型：tmpfs 是直接建立在 VM 之上的，用一个简单的 mount 命令就可以创建 tmpfs 文件系统了。速度快，可以动态分配文件系统大小。

步骤 3 挂载 cpuset 管理子系统

挂载某一个 cgroups 子系统到挂载点之后，就可以通过在挂载点下面建立文件夹或者使用 cgcreate 命令的方法创建 cgroups 层级结构中的节点/控制组；对应的删除则使用 rmdir 删除文件夹，或使用 cgdelete 命令删除。

```
# mkdir cpuset
# mount -t cgroup -o cpuset cpuset /cgroup/cpuset    #挂载 cpuset 子系统
# cd cpuset
# mkdir mycpuset    #创建一个控制组，删除用 rmdir 命令
# cd mycpuset
```

步骤 4 设置 cpu 核数

```
# echo 0 > cpuset.mems    #设置 0 号内存结点。mems 默认为空，因此需要填入值。
# echo 0-2 > cpuset.cpus    #这里的 0-2 指的是使用 cpu 的 0、1、2 三个核。实现了只是用这三个核。
# cat cpuset.mems
0
# cat cpuset.cpus
0-2
```

步骤 5 简单的死循环 C 源文件 while_long.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
```

```
while (1){
    printf("Over");
    exit(0);
}
```

步骤 6 测试验证

(1) 打开一个终端，执行以下命令：

```
# gcc while_long.c -o while_long          # 编译上述 C 源文件
# ls
while_long  while_long.c
# cgexec -g cpuset:mycpuset ./while_long  # 指定在 cpuset 子系统的 mycpuset 控制组中运行
```

(2) 不要关闭上述终端，另打开一个终端，执行以下命令：

```
# top          #查看程序 while_long 的 PID，假设为 19518。输入 q 退出当前查看状态
```

执行如下：

```
top - 17:37:18 up 1:35, 2 users, load average: 1.00, 0.80, 0.41
Tasks: 122 total, 2 running, 120 sleeping, 0 stopped, 0 zombie
%Cpu(s): 25.0 us, 0.0 sy, 0.0 ni, 75.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 6812.0 total, 5852.4 free, 332.1 used, 627.4 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used. 6130.5 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
10127	root	20	0	2368	896	448	R	100.0	0.0	7:38.50	while_long
10201	root	20	0	218432	6144	3392	R	0.3	0.1	0:00.03	top
1	root	20	0	174144	16768	8832	S	0.0	0.2	0:02.23	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd

```
# taskset -p 10127          #显示的如果是 7（111），则测试限制 cpu 核数成功。
pid 10127's current affinity mask: 7
# taskset -pc 10127
pid 10127's current affinity list: 0-2
```


4 实验四 中断和异常管理

4.1 实验介绍

本实验通过在使用软中断延迟机制 tasklet 实现打印 helloworld，用工作队列实现周期打印 helloworld 以及在用户态下捕获终端按键信号等任务操作，让学生们了解并掌握操作系统中的中断和异常管理机制。

4.1.1 相关概念

1、中断

在计算机科学中，中断（Interrupt）是指处理器接收到来自硬件或软件的信号，提示发生了某个事件，应该被注意，这种情况就称为中断。中断通常分为同步中断（synchronous）和异步中断（asynchronous）。

同步中断：是当指令执行时，由 CPU 控制单元产生的，只有在一条指令终止执行后 CPU 才会发生中断。

异步中断：是由其他硬件设备依照 CPU 时钟信号随机产生的。

Intel 微处理器手册中，把同步和异步中断分别称为异常（exception）和中断（interrupt）。

中断是由间隔定时器和 I/O 设备产生的，而异常是由程序的错误产生的，或是由内核必须处理的异常条件产生的。

2、中断的作用

每个中断或者异常都对应着它的中断或者异常处理程序，中断或异常处理程序不是一个进程，而是一个内核控制路径，代表中断发生时正在运行的进程执行。中断处理是内核执行的最敏感的任务之一，必须满足以下约束：

- 1) 内核相应中断后的操作分两部分：关键的紧急的部分，内核立即执行；其余的推迟随后执行。
- 2) 中断程序必须使内核控制路径能以嵌套的方式执行，当最后一个内核控制路径终止时，内核必须能恢复被中断进程的执行，或者如果中断信号已经导致了重新调度，内核能切换到另外的进程。

3) 尽管中断可以嵌套，但在临界区中，中断必须禁止。但内核必须尽可能限制这样的临界区，大部分时间应该以开中断的方式运行。

3、中断请求 (Interrupt Request, IRQ)

IRQ (Interrupt Request) 的作用就是在我们所用的电脑中，执行硬件中断请求的动作，用来停止其相关硬件的工作状态。比如我们要打印一份文件，在打印结束时就需要由系统对打印机提出相应的中断请求，来以此结束这个打印的操作。IRQ 具有以下特点：

- 1) 硬件设备控制器通过 IRQ 线向 CPU 发出中断，可以通过禁用某条 IRQ 线来屏蔽中断。
- 2) 被禁止的中断不会丢失，激活 IRQ 后，中断还会被发到 CPU
- 3) 激活/禁止 IRQ 线 != 可屏蔽中断的全局屏蔽/非屏蔽

4、中断的上半部和下半部

中断服务程序一般都是在中断请求关闭的条件下执行的，以避免嵌套而使中断控制复杂化。但是，中断是一个随机事件，它随时会到来，如果关中断的时间太长，CPU 就不能及时响应其他的中断请求，从而造成中断的丢失。因此，Linux 内核的目标就是尽可能快的处理完中断请求，尽其所能把更多的处理向后推迟。例如，假设一个数据块已经达到了网线，当中断控制器接受到这个中断请求信号时，Linux 内核只是简单地标志数据到来了，然后让处理器恢复到它以前运行的状态，其余的处理稍后再进行（如把数据移入一个缓冲区，接受数据的进程就可以在缓冲区找到数据）。因此，内核把中断处理分为两部分：上半部 (tophalf) 和下半部 (bottomhalf)，上半部（就是中断服务程序）内核立即执行，而下半部（就是一些内核函数）留着稍后处理，

首先，一个快速的“上半部”来处理硬件发出的请求，它必须在一个新的中断产生之前终止。通常，除了在设备和一些内存缓冲区（如果你的设备用到了 DMA，就不止这些）之间移动或传送数据，确定硬件是否处于健全的状态之外，这一部分做的工作很少。

下半部运行时是允许中断请求的，而上半部运行时是关中断的，这是二者之间的主要区别。但是，内核到底什么时候执行下半部，以何种方式组织下半部？这就是我们要讨论的下半部实现机制，这种机制在内核的演变过程中不断得到改进，在以前的内核中，这个机制叫做 bottomhalf（简称 bh），在 2.4 以后的版本中有了新的发展和改进，改进的目标使下半部可以在多处理机上并行执行，并有助于驱动程序的开发者进行驱动程序的开发。

4.1.2 任务描述

- 编写内核模块，使用 tasklet 实现打印 helloworld。
- 编写一个内核模块程序，用工作队列实现周期打印 helloworld。
- 在用户态编写一个信号捕获程序，捕获终端按键信号（包括 ctrl+c、ctrl+z、ctrl+\）。

4.2 实验目的

- 正确编写满足功能的源文件，正确编译。
- 正常加载、卸载内核模块；且内核模块功能满足任务所述。
- 了解操作系统的中断与异常管理。

4.3 实验任务

4.3.1 使用 tasklet 实现打印 helloworld

4.3.1.1 相关知识

tasklet

tasklet 是 Linux 中断处理机制中的软中断延迟机制。引入 tasklet，最主要的是考虑支持 SMP（多处理，Symmetrical Multi-Processing），提高 SMP 多个 CPU 的利用率；不同的 tasklet 可以在不同的 cpu 上运行。tasklet 可以理解为 softirq（软中断）的派生，所以它的调度时机和软中断一样。对于内核中需要延迟执行的多数任务都可以用 tasklet 来完成，由于同类 tasklet 本身已经进行了同步保护，所以使用 tasklet 比软中断要简单的多，tasklet 不需要考虑 SMP 下的并行问题，而又比 workqueues 有着更好的性能。tasklet 通常作为中断下半部来使用，它在性能和易用性之间有着很好的平衡。

1、定义

tasklet 由 tasklet_struct 结构体来表示，定义在头文件<linux/interrupt.h>中。在使用 tasklet 前，必须首先创建一个 tasklet_struct 类型的变量。tasklet 的结构体中包含处理函数的函数指针 func，它指向的是这样的一个函数：

```
void tasklet_handler(unsigned long data);
```

如同上半部分的中断处理程序一样，这个函数需要我们自己来实现。

2、tasklet 常用接口

创建好之后，我们还要通过如下的方法对 tasklet 进行初始化与调度：

```
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);  
/* 初始化 tasklet，func 指向要执行的函数，data 为传递给函数 func 的参数 */  
tasklet_schedule(&my_tasklet)           /*调度执行指定的 tasklet*/  
void tasklet_kill(struct tasklet_struct *t) /*移除指定 tasklet*/  
void tasklet_disable(struct tasklet_struct *t) /*禁用指定 tasklet*/  
void tasklet_enable(struct tasklet_struct *t) /*启用先前被禁用的 tasklet*/
```

4.3.1.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/4/task1 目录下。

```
[root@openEuler ~]# cd tasks_k/4/task1
[root@openEuler task1]# ls
Makefile  tasklet_intertupt.c
```

步骤 2 编译源文件

```
[root@openEuler task1]# make
make -C /root/kernel M=/root/tasks_k/4/task1 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/4/task1/tasklet_intertupt.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/4/task1/tasklet_intertupt.mod.o
LD [M] /root/tasks_k/4/task1/tasklet_intertupt.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task1]# ls
Makefile      Module.symvers  tasklet_intertupt.ko  tasklet_intertupt.mod.o
modules.order tasklet_intertupt.c tasklet_intertupt.mod.c tasklet_intertupt.o
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task1]# insmod tasklet_intertupt.ko
[root@openEuler task1]# dmesg | tail -n2
[86929.981168] Start tasklet module...
[86929.981488] Hello World! tasklet is working...
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task1]# rmmod tasklet_intertupt
[root@openEuler task1]# dmesg | tail -n3
[86929.981168] Start tasklet module...
[86929.981488] Hello World! tasklet is working...
[86973.915367] Exit tasklet module...
[root@openEuler task1]#
```

4.3.2 用工作队列实现周期打印 helloworld

4.3.2.1 相关知识

一、工作队列

我们把推后执行的任务叫做工作（work），描述它的数据结构为 work_struct；这些工作以队列结构组织成工作队列（workqueue），其数据结构为 workqueue_struct，而工作线程就

是负责执行工作队列中的工作。系统默认的工作者线程为 events，自己也可以创建自己的工作线程。

工作队列是实现延迟的新机制，从 2.5 版本 Linux 内核开始提供该功能。工作队列可以把工作延迟，交由一个内核线程去执行，也就是说，这个下半部分可以在进程上下文中执行。这样，通过工作队列执行的代码能占尽进程上下文的所有优势，且工作队列实现了内核线程的封装，不易出错。最重要的就是工作队列允许被重新调度甚至是睡眠。

那么，什么情况下使用工作队列，什么情况下使用 tasklet：如果推后执行的任务需要睡眠，那么就选择工作队列；如果推后执行的任务不需要睡眠，那么就选择 tasklet。如果需要用一个可以重新调度的实体来执行你的下半部处理，也应该使用工作队列。它是唯一能在进程上下文运行的下半部实现的机制，也只有它才可以睡眠。如果推后执行的任务需要延时指定的时间再触发，那么使用工作队列，因其可以利用 timer 延时；如果推后执行的任务需要在一个 tick 之内处理，则只有软中断或 tasklet，因其可以抢占普通进程和内核线程；如果推后执行的任务对延迟的时间没有任何要求，则使用工作队列，此时通常为无关紧要的任务。

工作队列允许内核代码来请求在将来某个时间调用一个函数；用来处理不是很紧急事件的回调方式处理方法。

二、工作队列的数据结构与编程接口 API

1、表示工作的数据结构（定义在内核源码：include/linux/workqueue.h）

(1) 正常的工作用 <linux/workqueue.h> 中定义的 work_struct 结构表示。这些结构被连接成链表。当一个工作者线程被唤醒时，它会执行它的链表上的所有工作。工作被执行完毕，它就将相应的 work_struct 对象从链表上移去。当链表上不再有对象的时候，它就会继续休眠。

(2) 延迟的工作用 delayed_work 数据结构，可直接使用 delay_work 将任务推迟执行。

2、工作队列中待执行的函数（定义在内核源码：include/linux/workqueue.h）

work_struct 结构中包含工作队列待执行的函数定义 work_func_t func；该工作队列待执行的函数原型是：typedef void (*work_func_t)(struct work_struct *work)

这个函数会由一个工作者线程执行，因此，函数会运行在进程上下文中。默认情况下，允许响应中断，并且不持有任何锁。如果需要，函数可以睡眠。需要注意的是，尽管该函数运行在进程上下文中，但它不能访问用户空间，因为内核线程在用户空间没有相关的内存映射。通常在系统调用发生时，内核会代表用户空间的进程运行，此时它才能访问用户空间，也只有在此时它才会映射用户空间的内存。

三、工作队列的使用

1、工作队列的创建

要使用工作队列，需要先创建工作项，有以下两种方式：

(1) 静态创建

```
DECLARE_WORK(n, f);
```

```
#定义正常执行的工作项
```

```
DECLARE_DELAYED_WORK(n, f);      #定义延后执行的工作项
```

其中，n 表示工作项的名字，f 表示工作项执行的函数。

这样就会静态地创建一个名为 n，待执行函数为 f 的 work_struct 结构。

(2) 动态创建、运行时创建：

通常在内核模块函数中执行以下函数：

```
INIT_WORK(_work, _func);          #初始化正常执行的工作项
```

```
INIT_DELAYED_WORK(_work, _func);  #初始化延后执行的工作项
```

其中，_work 表示 work_struct 的任务对象；_func 表示工作项执行的函数。

这会动态地初始化一个由 work 指向的工作。

2、工作项与工作队列的调度运行

(1) 工作项的调度运行

工作成功创建后，我们可以调度它了。想要把给定工作的待处理函数提交给缺省的 events 工作线程，只需调用 schedule_work(&work); work 马上就会被调度，一旦其所在的处理器上的工作者线程被唤醒，它就会被执行。有时候并不希望工作马上就被执行，而是希望它经过一段延迟以后再执行。在这种情况下，可以调度它在指定的时间执行：

```
schedule_delayed_work(&work,delay);
```

这时，&work 指向的 work_struct 直到 delay 指定的时钟节拍用完以后才会执行。

(2) 工作队列的调度运行

对于工作队列的调度，则使用以下两个函数：

```
bool queue_work(struct workqueue_struct *wq, struct work_struct *work)
```

#调度执行一个指定 workqueue 中的任务。

```
bool queue_delayed_work(struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)
```

#延迟调度执行一个指定 workqueue 中的任务，功能与 queue_work 类似，输入参数多了一个 delay。

3、工作队列的释放

```
void flush_workqueue(struct workqueue_struct *wq);  #刷新工作队列，等待指定列队中的任务全部执行完毕。
```

```
void destroy_workqueue(struct workqueue_struct *wq);  #释放工作队列所占的资源
```

4.3.2.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/4/task2 目录下。

```
[root@openEuler ~]# cd tasks_k/4/task2
[root@openEuler task2]# ls
Makefile  workqueue_test.c
```

步骤 2 编译源文件。

```
[root@openEuler task2]# make
make -C /root/kernel M=/root/tasks_k/4/task2 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/4/task2/workqueue_test.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/4/task2/workqueue_test.mod.o
LD [M] /root/tasks_k/4/task2/workqueue_test.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task2]# ls
Makefile      Module.symvers  workqueue_test.ko  workqueue_test.mod.o
modules.order  workqueue_test.c  workqueue_test.mod.c  workqueue_test.o
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task2]# insmod workqueue_test.ko
[root@openEuler task2]# dmesg | tail -n5
[87442.393468] Start workqueue_test module.
[87447.525385] Hello World!
[87462.629144] Hello World!
[87477.732936] Hello World!
[87492.836697] Hello World!
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task2]# rmmod workqueue_test
[root@openEuler task2]# dmesg | tail -n6
[87442.393468] Start workqueue_test module.
[87447.525385] Hello World!
[87462.629144] Hello World!
[87477.732936] Hello World!
[87492.836697] Hello World!
[87526.712833] Exit workqueue_test module.
[root@openEuler task2]#
```

由代码逻辑知：工作队列延时 5*HZ（5 秒）开始执行，模块加载后 5 秒才打印 HelloWorld!；而后每次执行工作队列中间休眠 15 秒。

4.3.3 编写一个信号捕获程序，捕获终端按键信号

4.3.3.1 相关知识

一、Linux 信号处理机制

1、基本概念

Linux 提供的信号机制是一种进程间异步的通信机制，每个进程在运行时，都要通过信号机制来检查是否有信号到达，若有，便中断正在执行的程序，转向与该信号相对应的处理程

序，以完成对该事件的处理；处理结束后再返回到原来的断点继续执行。实质上，信号机制是对中断机制的一种模拟，在实现上是一种软中断。

2、信号的产生

信号的生成来自内核，让内核生成信号的请求来自 3 个地方：

1) 用户：用户能够通过终端按键产生信号，例如；

ctrl+c ----> 2) SIGINT (终止、中断)

ctrl+\ ----> 3) SIGQUIT (退出)

ctrl+z ----> 20) SIGTSTP (暂时、停止)

或者是终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号

2) 内核：当进程执行出错时，内核会给进程发送一个信号，例如非法段存取(内存访问违规)、浮点数溢出等；

3) 进程：一个进程可以通过系统调用 kill 给另一个进程发送信号，一个进程可以通过信号和另外一个进程进行通信。

当信号发送到某个进程中时，操作系统会中断该进程的正常流程，并进入相应的信号处理函数执行操作，完成后再回到中断的地方继续执行。需要说明的是，信号只是用于通知进程发生了某个事件，除了信号本身的信息之外，并不具备传递用户数据的功能。

3、信号的响应动作/处理

每个信号都有自己的响应动作，当接收到信号时，进程会根据信号的响应动作执行相应的操作，信号的响应动作有以下几种：

1) 中止进程 Term

2) 忽略信号 Ign

3) 中止进程并保存内存信息 Core

4) 停止进程 Stop

5) 继续运行进程 Cont

用户可以通过 signal 或 sigaction 函数修改信号的响应动作（也就是常说的“注册信号”）。另外，在多线程中，各线程的信号响应动作都是相同的，不能对某个线程设置独立的响应动作。

4、信号类型

Linux 支持的信号类型可以参考下面给出的列表。

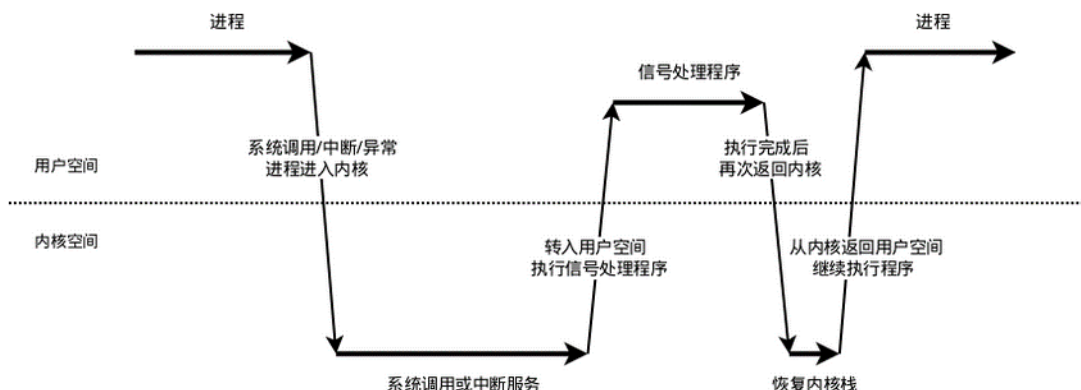
信号	值	动作	说明
SIGHUP	1	Term	终端控制进程结束(终端连接断开)
SIGINT	2	Term	用户发送INTR字符(Ctrl+C)触发
SIGQUIT	3	Core	用户发送QUIT字符(Ctrl+Q)触发
SIGILL	4	Core	非法指令(程序错误、试图执行数据段、栈溢出等)
SIGABRT	6	Core	调用abort函数触发
SIGFPE	8	Core	算术运行错误(浮点运算错误、除数为零等)
SIGKILL	9	Term	无条件结束程序(不能被捕获、阻塞或忽略)
SIGSEGV	11	Core	无效内存引用(试图访问不属于自己的内存空间、对只读内存空间进行写操作)
SIGPIPE	13	Term	消息管道损坏(FIFO/Socket通信时,管道未打开而进行写操作)
SIGALRM	14	Term	时钟定时信号
SIGTERM	15	Term	结束程序(可以被捕获、阻塞或忽略)
SIGUSR1	30,10,16	Term	用户保留
SIGUSR2	31,12,17	Term	用户保留
SIGCHLD	20,17,18	Ign	子进程结束(由父进程接收)
SIGCONT	19,18,25	Cont	继续执行已经停止的进程(不能被阻塞)
SIGSTOP	17,19,23	Stop	停止进程(不能被捕获、阻塞或忽略)
SIGTSTP	18,20,24	Stop	停止进程(可以被捕获、阻塞或忽略)
SIGTTIN	21,21,26	Stop	后台程序从终端中读取数据时触发
SIGTTOU	22,22,27	Stop	后台程序向终端中写数据时触发

也可使用 `kill -l` 查看当前系统的信号编号列表, 其中 1~31 为常规信号, 34~64 为实时信号。

```
[root@openEuler ~]# kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD    18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF    28) SIGWINCH   29) SIGIO      30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

5、信号机制

前面提到过, 信号是异步的, 这就涉及信号何时接收、何时处理的问题。我们知道, 函数运行在用户态, 当遇到系统调用、中断或是异常的情况时, 程序会进入内核态。信号涉及到了这两种状态之间的转换, 过程可以先看一下下面的示意图:



(1) 信号的接收

接收信号的任务是由内核代理的，当内核接收到信号后，会将其放到对应进程的信号队列中，同时向进程发送一个中断，使其陷入内核态。注意，此时信号还只是在队列中，对进程来说暂时是不知道有信号到来的。

(2) 信号的检测

进程陷入内核态后，有两种场景会对信号进行检测：

- 1) 进程从内核态返回到用户态前进行信号检测；
- 2) 进程在内核态中，从睡眠状态被唤醒的时候进行信号检测；

当发现新信号时，便会进入下一步，信号的处理。

(3) 信号的处理

信号处理函数是运行在用户态的，调用处理函数前，内核会将当前内核栈的内容备份拷贝到用户栈上，并且修改指令寄存器（eip）将其指向信号处理函数。

接下来进程返回到用户态中，执行相应的信号处理函数。

信号处理函数执行完成后，还需要返回内核态，检查是否还有其它信号未处理。如果所有信号都处理完成，就会将内核栈恢复（从用户栈的备份拷贝回来），同时恢复指令寄存器（eip）将其指向中断前的运行位置，最后回到用户态继续执行进程。

至此，一个完整的信号处理流程便结束了，如果同时有多个信号到达，上面的处理流程会在第 2 步和第 3 步骤间重复进行。

二、信号处理函数 signal()

函数原型：void (*signal (int signum ,void (*handler)(int))) (int) ;

功 能：设置捕捉某一信号后，对应的处理函数。

头文件：#include <signal.h>

参数说明：

signum：指定的信号的编号（或捕捉的信号），可以使用头文件中规定的宏；

handle：函数指针，是信号到来时需要运行的处理函数，参数是 signal() 的第一个参数 signum。

对于第二个参数，可以设置为 SIG_IGN，表示忽略第一个参数的信号；可以设置为 SIG_DFL，表示采用默认的方式处理信号；也可以指定一个函数地址，自己实现处理方式。

返回值：运行成功，返回原信号处理函数的指针；失败则返回 SIG_ERR。

三、信号与中断的异同点

1、信号与中断的相似点

- (1) 采用了相同的异步通信方式；
- (2) 当检测出有信号或中断请求是，都暂停正在执行的程序，转而去执行相应的处理程序；
- (3) 都在处理完毕后返回到原来的断点；
- (4) 对信号或中断都可进行评比。

2、信号与中断的区别

- (1) 中断有优先级，而信号没有优先级，所有信号都是平等的；
- (2) 信号处理程序是在用户态下运行的；而中断处理程序是在内核态下运行的；
- (3) 中断响应是及时的，而信号响应通常有较大的时间延迟。

4.3.3.2 实验步骤

步骤 1 正确编写满足功能的源文件。在这里我们的示例源文件存放在 tasks_k/4/task3 目录下。

```
[root@openEuler ~]# cd tasks_k/4/task3
[root@openEuler task2]# ls
catch_signal.c
```

步骤 2 编译源文件。

```
[root@openEuler task3]# gcc catch_signal.c -o catch_signal
[root@openEuler task3]# ls
catch_signal  catch_signal.c
```

步骤 3 执行并验证。

```
[root@openEuler task3]# ./catch_signal
Current process ID is 21128
^C
Get a signal:SIGINT. You pressed ctrl+c.
[root@openEuler task3]# ./catch_signal
Current process ID is 21141
^\\
Get a signal:SIGQUIT. You pressed ctrl+\\.
[root@openEuler task3]# ./catch_signal
Current process ID is 21154
```



^Z

Get a signal:SIGHUP. You pressed ctrl+z.

[root@openEuler task3]#

5 实验五 内核时间管理

5.1 实验介绍

本实验通过调用内核时钟接口打印系统当前时间，实现一个 timer 延时打印以及监控实现累加计算所花时间等任务操作，让学生们了解并掌握操作系统中的内核时间管理。

5.1.1 任务描述

- 编写内核模块，调用内核时钟接口，打印出系统当前时间。格式示例：2020-03-09 11:54:31；
- 编写内核模块程序，实现一个 timer，该定时器延时 10 秒后打印“hello,world”。
- 调用内核时钟接口，编写内核模块，监控实现累加计算 $sum=1+2+3+...+100000$ 所花时间。

5.2 实验目的

- 正确编写满足功能的源文件，正确编译。
- 正常加载、卸载内核模块；且内核模块功能满足任务所述。
- 了解操作系统的内核时间管理。

5.3 实验任务

5.3.1 调用内核时钟接口打印当前时间

5.3.1.1 相关知识

一、内核时钟相关定义

1. timeval 结构体

头文件：<linux/time.h>

```
struct timeval {
```

```

__kernel_time_t      tv_sec;          /* seconds */
__kernel_suseconds_t tv_usec;        /* microseconds */
};

```

其中 tv_sec 是自 1970 年 1 月 1 日 00:00:00 起到现在的秒数。而 tv_usec 是当前秒数已经经过的微秒数。

2. do_gettimeofday()

头文件: <linux/time.h>

函数原型: void do_gettimeofday(struct timeval *tv);

功能: 返回自 1970-01-01 00:00:00 到现在的秒数, 及当前秒经过的毫秒数, 保存在 tv 指向的 timeval 结构体中。

3. rtc_time 结构体

头文件<linux/rtc.h>

```

struct rtc_time {
    int tm_sec;          // 表「秒」数, 在[0,61]之间, 多出来的两秒是用来处理跳秒问题用的。
    int tm_min;          // 表「分」数, 在[0,59]之间。
    int tm_hour;         // 表「时」数, 在[0,23]之间。
    int tm_mday;         // 表「本月第几日」, 在[1,31]之间。
    int tm_mon;          // 表「本年第几月」, 在[0,11]之间。
    int tm_year;         // 要加 1900 表示那一年。
    int tm_wday;         // 表「本周第几日」, 在[0,6]之间。
    int tm_yday;         // 表「本年第几日」, 在[0,365]之间, 闰年有 366 日。
    int tm_isdst;        // 表是否为「日光节约时间」。
};

```

年份加上 1900, 月份加上 1, 小时加上 8。

注意: 可以自己先思考一下这样的加上 x 操作如果导致溢出, 应该怎么处理。例如, 得到的 UTC 小时数为 22, 加上 8 之后 30 明显溢出 (超过 24), 并且由于小时数溢出到了第二天, 因此天数也应该加 1。

4. rtc_time_to_tm()

头文件: <linux/rtc.h>

函数原型: void rtc_time_to_tm(unsigned long time, struct rtc_time *tm);

功能: 将 time 存储的秒数转换为年月日时分秒等信息保存在 rtc_time 结构体中。

参数: time 为秒数, 可以是 do_gettimeofday()函数获取的秒数。tm 是 rtc_time 结构体指针, 结构体中存放了年月日时分秒等信息。

5.3.1.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/5/task1 目录下。

```
[root@openEuler ~]# cd tasks_k/5/task1
[root@openEuler task1]# ls
current_time.c  Makefile
```

步骤 2 编译源文件。

```
[root@openEuler task1]# make
make -C /root/kernel M=/root/tasks_k/5/task1 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/5/task1/current_time.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/5/task1/current_time.mod.o
LD [M] /root/tasks_k/5/task1/current_time.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task1]# ls
current_time.c  current_time.ko  current_time.mod.c  current_time.mod.o  current_time.o  Makefile
modules.order  Module.symvers
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task1]# insmod current_time.ko
[root@openEuler task1]# dmesg | tail -n2
[89336.822428] Start current_time module...
[89336.822668] Current time: 2020-11-12 10:45:21
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task1]# rmmod current_time
[root@openEuler task1]# dmesg | tail -n3
[89336.822428] Start current_time module...
[89336.822668] Current time: 2020-11-12 10:45:21
[89363.701104] Exit current_time module...
```

5.3.2 编写 timer，在特定时刻打印 hello,world

5.3.2.1 相关知识

一、内核定时器

时钟中断对于管理操作系统尤为重要，大量内核函数的生命周期都离不开流逝的时间的控制。正如我们所看到的，时钟中断能处理许多内核任务，所以它对内核来说极为重要。

定时器（有时也称为动态定时器或内核定时器）是管理内核流逝的时间的基础。内核经常需要推迟执行某些代码，我们需要的是一种工具，能够使工作在指定时间点上执行——不长不短，正好在希望的时间点上。内核定时器正是解决这个问题的理想工具。

定时器的使用很简单。你只需要执行一些初始化工作，设置一个超时时间，指定超时发生后执行的函数，然后激活定时器就可以了。指定的函数在定时器到期时自动执行。注意定时器并不周期运行，它在超时后就自行撤销，这也正是这种定时器被称为动态定时器的一个原因：动态定时器不断地创建和撤销，而且它的运行次数也不受限制。

二、定时器的使用

1、定时器结构

定时器由结构 `timer_list` 表示，定义在文件 `<linux/timer.h>` 中。

```
struct timer_list
{
    struct hlist_node    entry;           /* 定时器链表的入口 */
    unsigned long        expires;        /* 以 jiffies 为单位的定时值（32 位） */
    void                (*function)(struct timer_list *); /* 定时器处理函数 */
    u32                  flags;          /* 定时器标志位 */
#ifdef CONFIG_LOCKDEP    /* 内核配置项——死锁检测模块，默认未配置 */
    struct lockdep_map    lockdep_map;   /* 数据结构定义在 <linux/lockdep.h>中 */
#endif
    KABI_RESERVE(1)      /* kernel abi 的保留值 */
    KABI_RESERVE(2)
    KABI_RESERVE(3)
    KABI_RESERVE(4)
}
```

注意：

- (1) 内核定时器用于控制某个函数（定时器处理函数）在未来的某个特定时间执行。
- (2) `CONFIG_LOCKDEP` 的设置依赖于 `CONFIG_DEBUG_KERNEL`，而 `CONFIG_DEBUG_KERNEL` 设置为 Y 时，表示正在开发驱动程序或尝试调试和确定内核问题，因此通常情况下，`CONFIG_LOCKDEP` 是未配置的状态。
- (3) `expires` 的值是 32 位的，因为内核定时器并不适用于长的未来时间点。

2、创建定时器

创建定时器时需要先定义定时器结构：一个 `timer_list` 结构体的实例对应一个定时器。也就是说，创建定时器就是实例化定时器数据结构。

3、初始化定时器

初始化 `struct timer_list` 数据结构，只需正确地设置 `expires` 与 处理函数 `function`。其他参数无需设置。

```
timer.expires = jiffies + delay;          /* 定时器超时的节拍数 */
timer.function = my_function;            /* 定时器超时时调用的函数 */
```

(1) 定时的数值 delay 如何设置：

因为内核定时器是基于 jiffies，所以设置的时间要在 jiffies 之上加上我们的定时值；例如：定时 2 秒，则 delay=2*HZ，HZ 是每秒中 jiffies 这个计数器会计多少值。

(2) 若当前 jiffies 计数≥timer.expires，timer.function 指向的处理函数就会开始执行。

(3) 定时器处理函数必须符合下面的函数原型：

```
void my_function(struct timer_list *timer);
```

4、激活定时器

函数：add_timer(struct timer_list *timer)

功能：将定时器注册到内核中（将定时器连接到内核专门的链表中），使之生效。

```
add_timer(&timer);
```

在注册之后，定时器就开始计时，在到达时间 expires 时，执行初始化时指定的处理函数。当激活定时器后，它只会执行一次处理函数，然后将定时器从内核中移除。

5.3.2.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/5/task2 目录下。

```
[root@openEuler ~]# cd tasks_k/5/task2
[root@openEuler task2]# ls
Makefile  timer_example.c
```

步骤 2 编译源文件。

```
[root@openEuler task2]# make
make -C /root/kernel M=/root/tasks_k/5/task2 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/5/task2/timer_example.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/5/task2/timer_example.mod.o
LD [M] /root/tasks_k/5/task2/timer_example.ko
make[1]: Leaving directory '/root/kernel'
[root@openEuler task2]# ls
Makefile  modules.order  Module.symvers  timer_example.c  timer_example.ko  timer_example.mod.c
timer_example.mod.o  timer_example.o
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task2]# insmod timer_example.ko
[root@openEuler task2]# dmesg -T | tail -n2
[Thu Nov 12 10:51:02 2020] Start timer_example module...
```



```
[Thu Nov 12 10:51:12 2020] hello,world!
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task2]# rmmod timer_example
[root@openEuler task2]# dmesg -T | tail -n3
[Thu Nov 12 10:51:02 2020] Start timer_example module...
[Thu Nov 12 10:51:12 2020] hello,world!
[Thu Nov 12 10:51:35 2020] Exit timer_example module...
```

代码中设置定时器超时时间为 10*HZ（10 秒），对应模块加载后 10 秒才打印 hello，world!。

5.3.3 调用内核时钟接口，监控累加计算代码的运行时间

5.3.3.1 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/5/task3 目录下。

```
[root@openEuler ~]# cd tasks_k/5/task3
[root@openEuler task3]# ls
Makefile  sum_time.c
```

步骤 2 编译源文件。

```
[root@openEuler task3]# make
make -C /root/kernel M=/root/tasks_k/5/task3 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/5/task3/sum_time.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/5/task3/sum_time.mod.o
LD [M] /root/tasks_k/5/task3/sum_time.ko
make[1]: Leaving directory '/root/kernel'
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task3]# insmod sum_time.ko
[root@openEuler task3]# dmesg | tail -n5
[90056.765027] Start sum_time module...
[90056.778831] The start time is: 1605149841 s 906211 us
[90056.779095] The sum of 1 to 100000 is: 5000050000
[90056.779334] The end time is: 1605149841 s 906715 us
[90056.779587] The cost time of sum from 1 to 100000 is: 504 us
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler task3]# rmmod sum_time
[root@openEuler task3]# dmesg | tail -n6
```

```
[90056.765027] Start sum_time module...  
[90056.778831] The start time is: 1605149841 s 906211 us  
[90056.779095] The sum of 1 to 100000 is: 5000050000  
[90056.779334] The end time is: 1605149841 s 906715 us  
[90056.779587] The cost time of sum from 1 to 100000 is: 504 us  
[90073.886442] Exit sum_time module...
```

运行结果可以看出，从 1 到 100000 的累加和所花时间是 504 us。

6 实验六 设备管理

6.1 实验介绍

本实验通过编写内核模块测试硬盘的读写速率，加载、卸载模块并查看模块打印信息的任务操作，让学生们了解并掌握操作系统中的设备管理。

6.1.1 相关知识

一、内核文件读写介绍

有时候需要在 Linux kernel 中读写文件数据，如调试程序的时候，或者内核与用户空间交换数据的时候。在 kernel 中操作文件没有标准库可用，需要利用 kernel 的一些函数，这些函数主要有：

```
filp_open()
filp_close()
kernel_read()
kernel_write()
```

这些函数在 `<linux/fs.h>` 头文件中声明。

二、内核文件读写接口

1、打开文件

函数原型：struct file* filp_open(const char* filename, int open_mode, int mode);

功能：在 kernel 中打开指定文件。

返回值：该函数返回 struct file* 结构指针，供后续函数操作使用；该返回值用 IS_ERR() 来检验其有效性。

参数说明：

filename：表明要打开或创建文件的名称（包括路径部分）。

注意：在内核中打开文件时需要注意打开的时机，很容易出现需要打开文件的驱动很早就加载并打开文件，但需要打开的文件所在设备还没有挂载到文件系统中，而导致打开失败。

open_mode：文件的打开方式，其取值与标准库中的 open 相应参数类似，包括：O_RDONLY（只读打开）、O_WRONLY（只写打开）、O_RDWR（读写打开）、O_CREAT（文件不存在则创建）等。

mode: 创建文件时使用, 设置创建文件的读写权限 (如 644) , 其它情况可以设为 0。

2、读文件

函数原型: `ssize_t kernel_read(struct file *file, void *buf, size_t count, loff_t *pos);`

功能: kernel 中文件的读操作。

参数说明:

file: 进行读取信息的目标文件, 即 `file_open()` 函数的返回值。

buf: 对应放置信息的缓冲区。

count: 要读取的信息长度。

pos: 表示用户在当前文件中进行读取操作的位置/偏移量, 即读的位置相对于文件开头的偏移。在读取信息后, 这个指针一般都会移动, 移动的值是要读取信息的长度值。

3、写文件

函数原型: `ssize_t kernel_write(struct file *file, const void *buf, size_t count, loff_t *pos);`

功能: kernel 中文件的写操作。

参数说明:

file: 进行信息写入的目标文件, 即 `file_open()` 函数的返回值。

buf: 要写入文件的信息缓冲区。

count: 要写入信息的长度。

pos: 表示用户在当前文件中进行写入操作的位置/偏移量。即写的位置相对于文件开头的偏移。

4、关闭文件

函数原型: `int filp_close(struct file*filp, fl_owner_t id);`

功能: 关闭指定文件。

参数说明:

filp: 待关闭的目标文件的文件指针。

id: 一般传递 NULL 值, 也可用 `current->files` 作为实参。

6.1.2 任务描述

- 编写内核模块测试硬盘的读、写速率, 加载、卸载模块并查看模块打印信息。

6.2 实验目的

- 正确编写满足功能的源文件，正确编译。
- 正常加载、卸载内核模块；且内核模块功能满足任务所述。
- 了解操作系统的设备管理。

6.3 实验任务

6.3.1 编写内核模块测试硬盘的写速率

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/6/write_disk 目录下。

```
[root@openEuler ~]# cd tasks_k/6/write_disk/
[root@openEuler write_disk]# ls
Makefile  write_to_disk.c
```

步骤 2 编译源文件。

```
[root@openEuler write_disk]# make
make -C /root/kernel M=/root/tasks_k/6/write_disk modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/6/write_disk/write_to_disk.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/6/write_disk/write_to_disk.mod.o
LD [M] /root/tasks_k/6/write_disk/write_to_disk.ko
make[1]: Leaving directory '/root/kernel'
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler write_disk]# insmod write_to_disk.ko
[root@openEuler write_disk]# dmesg | tail -n3
[104186.675791] Start write_to_disk module...
[104187.685460] Writing to file costs 1009173 us
[104187.685732] Writing speed is 531 M/s
[root@openEuler read_disk]# ll /home/tmp_file
-rw-----. 1 root root 512M Nov 12 14:52 /home/tmp_file
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler write_disk]# rmmod write_to_disk
[root@openEuler write_disk]# dmesg | tail -n4
[104186.675791] Start write_to_disk module...
[104187.685460] Writing to file costs 1009173 us
```

```
[104187.685732] Writing speed is 531 M/s
[104237.777141] Exit write_to_disk module...
```

6.3.2 编写内核模块测试硬盘的读速率

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/6/read_disk 目录下。

```
[root@openEuler ~]# cd tasks_k/6/read_disk/
[root@openEuler read_disk]# ls
Makefile  read_from_disk.c
```

步骤 2 编译源文件。

```
[root@openEuler read_disk]# make
make -C /root/kernel M=/root/tasks_k/6/read_disk modules
make[1]: Entering directory '/root/kernel'
  CC [M]  /root/tasks_k/6/read_disk/read_from_disk.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/tasks_k/6/read_disk/read_from_disk.mod.o
  LD [M]  /root/tasks_k/6/read_disk/read_from_disk.ko
make[1]: Leaving directory '/root/kernel'
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler read_disk]# insmod read_from_disk.ko
[root@openEuler read_disk]# dmesg | tail -n3
[104496.506192] Start read_from_disk module...
[104496.841149] Read file costs 267124 us
[104496.841381] Reading speed is 2009 M/s
```

步骤 4 卸载内核模块，并查看结果。

```
[root@openEuler read_disk]# rmmod read_from_disk
[root@openEuler read_disk]# dmesg | tail -n4
[104496.506192] Start read_from_disk module...
[104496.841149] Read file costs 267124 us
[104496.841381] Reading speed is 2009 M/s
[104601.931372] Exit read_from_disk module...
[root@openEuler read_disk]#
```

7 实验七 文件系统

7.1 实验介绍

本实验通过为 Ext4 文件系统添加扩展属性，注册自定义文件系统类型，以及使用内核模块操作文件系统等任务，让学生们了解并掌握操作系统中的文件系统。

7.1.1 任务描述

- 熟悉文件系统扩展属性 EA，为 Ext4 文件系统添加扩展属性；
- 使用文件系统注册/注销函数，注册一个自定义文件系统类型；
- 编写一个模块，在加载模块时，在 /proc 目录下创建一个名称为 myproc 的目录；
- 编写一个模块，该模块有三个参数：一个为字符串型，两个为整型。两个整型中，一个在 /sys 下不可见。加载模块后，使用 echo 向模块传递参数值来改变指定参数的值。

7.2 实验目的

- 正确编写满足功能的源文件，正确编译。
- 正常加载、卸载内核模块；且内核模块功能满足任务所述。
- 了解操作系统的内核文件系统管理。

7.3 实验任务

7.3.1 为 Ext4 文件系统添加扩展属性

7.3.1.1 相关知识

1、基本概念

文件扩展属性（xattr-Extended attributes）提供了一种机制，用来将 key-value 键值对永久地关联到文件；让现有的文件系统得以支持在原始设计中未提供的功能。扩展属性是目前流行的 POSIX 文件系统具有的一项特殊的功能，可以给文件、文件夹添加额外的 Key-value 的键值

对，键和值都是字符串并且有一定长度的限制——定义于 include/uapi/linux/limits.h 文件中。在保存 xattr 时，key 的长度不能超过 255byte，value 不能超过 64k，总的配对数不能超过 64k。

xattr 扩展函数，仅仅作用于“支持扩展属性的文件系统”，并在挂载文件系统时，需要开启 xattr。因为扩展属性需要底层文件系统的支持，在使用扩展属性时，需要查看文件系统说明文章，看此文件系统是否支持扩展属性，以及对扩展属性命名空间等相关的支持。支持扩展属性常见的文件系统有：ext2、ext3、reiserfs、jfs 和 xfs，这些文件系统对于扩展属性的支持都是可选项。

2、扩展属性名称空间

扩展属性名称的格式是 namespace.attribute，名称空间 namespace 是用来定义不同的扩展属性的类。目前有 security，system，trusted，user 四种扩展属性类。

(1) 扩展的安全属性——security

安全属性名称空间被内核用于安全模块，例如 SELinux。对安全属性的读和写权限依赖于策略的设定。这策略是由安全模块载入的。如果没有载入安全模块，所有的进程都对安全属性有读权限，写权限只有那些有 CAP_SYS_ADMIN（允许执行系统管理任务，如加载或卸载文件系统、设置磁盘配额等）的进程才有。

(2) 扩展的系统属性——system

扩展的系统属性被内核用来存储系统对象，比如说 ACL。对系统属性的读和写权限依赖于策略的设定。

(3) 受信任的扩展属性——trusted

受信任的扩展属性只对那些有 CAP_SYS_ADMIN 的进程可见和可获得。这个类中的属性被用来在用户空间中保存一些普通进程无法得到的信息。

(4) 扩展的用户属性——user

扩展的用户属性被分配给文件和目录用来存储任意的附加信息，比如 mime type、字符集或是文件的编码。用户属性的权限由文件权限位来定义。对于普通文件和目录，文件权限位定义文件内容的访问，对于设备文件来说，它们定义对设备的访问。扩展的用户属性只被用于普通的文件和目录，对用户属性的访问被限定于属主和那些对目录有 sticky 位设置的用户。

3、文件系统特殊要求

对于 ext 文件系统，为了使用扩展用户属性，要求文件系统挂载时有 user_xattr 选项。

在 ext 文件系统中，每一个扩展属性必须占用一个单独的文件系统块，块大小取决于创建文件系统时的设置。

4、文件扩展属性的设置命令

(1) setfattr：设置文件系统对象的扩展属性（无文件系统限制）

语法: setfattr [-h] -n name [-v value] pathname...

setfattr [-h] -x name pathname...

setfattr [-h] --restore=file

-n name: 指定属性名称

-v value: 设置属性值。可以使用三种方法对 value 进行编码: 如果给定的字符串用双引号引起来, 则将其中的字符串视为文本。在这种情况下, 其中的反斜线和双引号需转义。任何控制字符都可以编码为“反斜杠后跟三个数字”作为八进制的 ASCII 码。如果给定的字符串以 0x 开头, 则表示一个十六进制数。如果给定的字符串以 0s 开头, 则需要 base64 编码。

-x name: 删除属性

-h: 不要遵循符号链接。如果路径名是符号链接, 则不会遵循它, 而是将其本身修改为 inode。

--version: 打印 setfattr 的版本并退出。

--help: 打印帮助以解释命令行选项。

(2) getfattr: 获取文件系统对象的扩展属性

-n name: 显示指定名称的属性。

-d: 显示所有属性的值。

-e en: 在提取属性之后进行编码, en 的值为 text、hex 和 base64。

en="text"时, 编码为文本的字符串的值用双引号 (") 引起来;

en="hex"时, 编码为十六进制的字符串以 0x 为前缀;

en="base64"时, 编码为 base64 的字符串以 0s 为前缀。

-m pattern: 正则表达式匹配的属性显示。默认匹配为 '^user\\.', 即匹配 user 名称空间的属性, 可以指定为 '.' 或 '.' 匹配所有属性。

-R: 递归显示。

7.3.1.2 实验步骤

步骤 1 环境准备: 为了使用扩展属性, 需要安装 libattr:

```
dnf install -y libattr
```

步骤 2 查看当前文件系统类型

```
df -Th
```

其中, -T 用于显示文件系统类型; -h 表示以 1024 的幂为单位显示文件系统大小。

```
[root@openEuler ~]# df -Th
Filesystem      Type      Size  Used Avail Use% Mounted on
devtmpfs        devtmpfs  3.1G   0    3.1G   0% /dev
tmpfs           tmpfs     3.4G   0    3.4G   0% /dev/shm
tmpfs           tmpfs     3.4G  14M   3.4G   1% /run
tmpfs           tmpfs     3.4G   0    3.4G   0% /sys/fs/cgroup
/dev/vda2       ext4      39G   14G   23G   38% /
tmpfs           tmpfs     3.4G   64K   3.4G   1% /tmp
/dev/vda1       vfat     1022M   5.8M 1017M   1% /boot/efi
tmpfs           tmpfs     682M   0    682M   0% /run/user/0
tmpfs           tmpfs     3.4G   0    3.4G   0% /cgroup
```

可使用 `df --help` 查看 `df` 的具体参数及用法。

步骤 3 检查当前文件系统是否支持文件扩展属性

在 `ext3` 和 `ext4` 文件系统上，可通过命令 `tune2fs -l` 来检查是否支持扩展属性。

(1) 用 `fdisk -l` 查看硬盘及分区信息

```
[root@openEuler ~]# fdisk -l
Disk /dev/vda: 40 GiB, 42949672960 bytes, 83886080 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: E4391D6A-9819-45CA-BDE4-52285BCE52A7

Device      Start      End  Sectors  Size Type
/dev/vda1    2048    2099199   2097152    1G EFI System
/dev/vda2   2099200  83884031  81784832   39G Linux filesystem
```

(2) 用 `tune2fs -l` 显示设备的详细信息

`tune2fs` 命令允许系统管理员在 `ext2`、`ext3` 或 `ext4` 文件系统上调整各种可调的文件系统参数。这些选项的当前值可以使用 `-l` 选项显示。

```
# tune2fs -l /dev/vda2 | grep user_xattr
Default mount options: user_xattr acl
```

通过检查 `/dev/vda2` 文件系统参数中的默认挂载选项“Default mount options”是否有对应内容；来确定分区设备的文件系统是否支持扩展属性——上图所示说明，该文件系统支持扩展用户属性 `user_xattr`、与 `ACL` 权限。

步骤 4 创建文件 `file.txt`，用 `setfattr` 设置文件系统对象的扩展属性

```
[root@openEuler task1]# touch file.txt
[root@openEuler task1]# setfattr -n user.name -v xattr_test file.txt
[root@openEuler task1]# setfattr -n user.city -v "Beijing" file.txt
[root@openEuler task1]# getfattr -d -m . file.txt
# file: file.txt
security.selinux="unconfined_u:object_r:admin_home_t:s0"
user.city="Beijing"
user.name="xattr_test"
```

对于不含转义字符 \ 的纯文本属性值，有无双引号限定效果一样。

步骤 5 设置八进制数属性值“\012”，最终以八进制数的 base64 编码存储。

```
[root@openEuler task1]# setfattr -n user.age -v "\012" file.txt
[root@openEuler task1]# getfattr -d -m . file.txt
# file: file.txt
security.selinux="unconfined_u:object_r:admin_home_t:s0"
user.age=0sCg==
user.city="Beijing"
user.name="xattr_test"
```

对于包含转义字符 \ 的文本属性值，无双引号则不对转义符 \ 进行转义；有双引号则对其进行转义。

步骤 6 设置十六进制数属性值，所设置的数的位数必须为偶数，即 0x 或 0X 后的数字必须为偶数位，否则出错。若设置成功，最终以十六进制数的 base64 编码存储。

```
[root@openEuler task1]# setfattr -n user.hex -v 0x0123 file.txt
[root@openEuler task1]# getfattr -d -m . file.txt
# file: file.txt
security.selinux="unconfined_u:object_r:admin_home_t:s0"
user.age=0sCg==
user.city="Beijing"
user.hex=0sASM=
user.name="xattr_test"
```

步骤 7 设置 base64 编码属性值，所设置的编码必须符合 base64 编码，即 0s 后的编码字符串必须符合 base64 编码，否则出错。若设置成功，最终以 base64 编码对应的文本信息存储。

```
[root@openEuler task1]# setfattr -n user.base64 -v 0sSGVsbG8gV29ybGQh file.txt
[root@openEuler task1]# getfattr -d -m . file.txt
# file: file.txt
security.selinux="unconfined_u:object_r:admin_home_t:s0"
user.age=0sCg==
user.base64="Hello World!"
user.city="Beijing"
user.hex=0sASM=
user.name="xattr_test"
```

tips：可在 <https://base64.us/> 中，将需要设置的属性值进行 base64 编码后，再使用 setfattr 命令设置，注意设置时需在 base64 编码前加 0s 前缀。

步骤 8 用 getfattr 编码设置。

保持原编码设置：

```
[root@openEuler task1]# getfattr -d -m . file.txt
# file: file.txt
```

```
security.selinux="unconfined_u:object_r:admin_home_t:s0"
user.age=0sCg==
user.base64="Hello World!"
user.city="Beijing"
user.hex=0sASM=
user.name="xattr_test"
```

对属性设置 text 编码时，结果如下：

```
[root@openEuler task1]# getfattr -d -e text file.txt
# file: file.txt
user.age="\012"
user.base64="Hello World!"
user.city="Beijing"
user.hex="#"
user.name="xattr_test"
```

对属性设置 hex 编码：

```
[root@openEuler task1]# getfattr -d -e hex file.txt
# file: file.txt
user.age=0x0a
user.base64=0x48656c6c6f20576f726c6421
user.city=0x4265696a696e67
user.hex=0x0123
user.name=0x78617474725f74657374
```

user.age 的属性值由八进制变成了十六进制；user.hex 的属性值还原成了最初设置的原值；user.name、user.city、user.base64 的属性值都转化为对应的十六进制值。

对属性设置 base64 编码：

```
[root@openEuler task1]# getfattr -d -e base64 file.txt
# file: file.txt
user.age=0sCg==
user.base64=0sSGVsbG8gV29ybGQh
user.city=0sQmVpamluZw==
user.hex=0sASM=
user.name=0seGF0dHJfdGVzdA==
```

user.age 与 user.hex 的属性值都保留最初的 base64 编码存储；user.name、user.city、user.base64 的属性值都转化为对应的 base64 编码值。

7.3.2 注册一个自定义的文件系统类型

7.3.2.1 相关知识

通常，用户在为自己的系统编译内核时可以把 Linux 配置为能够识别所有需要的文件系统。但是，文件系统的源代码实际上要么包含在内核映像中，要么作为一个模块被动态装入。VFS 必须对目前已在内核中的所有文件系统类型进行跟踪，这是通过进行文件系统类型注册来实现的。

注册函数 register_filesystem/注销函数 unregister_filesystem:

两个函数分别用于文件系统类型的注册和注销，都只有一个参数，即代表文件系统类型的一个指向 file_system_type 对象的指针。

1、头文件: <linux/fs.h>

2、函数原型: 两个函数的原型分别为:

```
int register_filesystem(struct file_system_type *);
int unregister_filesystem(struct file_system_type *);
```

3、返回值: 函数执行成功时，返回 0；否则，返回一个错误值。

4、参数说明:

每个注册的文件系统都用一个类型为 file_system_type 的对象来表示。

文件系统对 file_system_type 结构体的填充:

1) ext4fs 文件系统对这个结构体的填充 (fs/ext4/super.c) :

```
static struct file_system_type ext4_fs_type = {
    .owner          = THIS_MODULE,
    .name           = "ext4",
    .mount          = ext4_mount,
    .kill_sb        = kill_block_super,
    .fs_flags       = FS_REQUIRES_DEV,
};
MODULE_ALIAS_FS("ext4");
```

2) ramfs 文件系统对这个结构体的填充 (fs/ramfs/inode.c) :

```
static struct file_system_type ramfs_fs_type = {
    .name           = "ramfs",
    .mount          = ramfs_mount,
    .kill_sb        = ramfs_kill_sb,
    .fs_flags       = FS_USERNS_MOUNT,
};
```

可见，一般文件系统的实现，实现这几个字段即可；其它的由内核利用或者填充。

7.3.2.2 实验步骤

步骤 1 查看系统中已经注册的文件系统类型

```
cat /proc/filesystems
```

步骤 2 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/7/task2 目录下。

```
[root@openEuler ~]# cd tasks_k/7/task2
[root@openEuler task2]# ls
Makefile  register_newfs.c
```

步骤 3 编译源文件。

```
[root@openEuler task2]# make
```

```
make -C /root/kernel M=/root/tasks_k/7/task2 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/7/task2/register_newfs.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/7/task2/register_newfs.mod.o
LD [M] /root/tasks_k/7/task2/register_newfs.ko
make[1]: Leaving directory '/root/kernel'
```

步骤 4 对比加载内核模块前后的文件系统结果。

```
[root@openEuler task2]# cat /proc/filesystems | grep myfs
[root@openEuler task2]# insmod register_newfs.ko
[root@openEuler task2]# cat /proc/filesystems | grep myfs
nodev myfs
```

步骤 5 卸载内核模块，并查看结果。

```
[root@openEuler task2]# rmmod register_newfs
[root@openEuler task2]# cat /proc/filesystems | grep myfs
[root@openEuler task2]# dmesg | tail -n2
[114719.268797] Start register_newfs module...
[114726.442879] Exit register_newfs module...
```

由实验结果可见：

当未加载内核模块时，当前系统中无自定义的文件系统“myfs”；当加载内核模块时，当前系统中可打印出自定义的文件系统“myfs”；当卸载内核模块时，当前系统中无自定义的文件系统“myfs”。

7.3.3 在/proc 下创建目录

7.3.3.1 相关知识

一、proc 文件系统

Linux 系统上的/proc 目录是一种文件系统，即 proc 文件系统。与其它常见的文件系统不同的是，/proc 是一种伪文件系统（也即虚拟文件系统），存储的是当前内核运行状态的一系列特殊文件，用户可以通过这些文件查看有关系统硬件及当前正在运行进程的信息，甚至可以通过更改其中某些文件来改变内核的运行状态。

基于/proc 文件系统如上所述的特殊性，其内的文件也常被称作虚拟文件，并具有一些独特的特点。例如，其中有些文件虽然使用查看命令查看时会返回大量信息，但文件本身的大小却会显示为 0 字节。此外，这些特殊文件中大多数文件的时间及日期属性通常为当前系统时间和日期，这跟它们随时会被刷新（存储于 RAM 中）有关。

为了查看及使用上的方便，这些文件通常会按照相关性进行分类存储于不同的目录甚至子目录中，如/proc/scsi 目录中存储的就是当前系统上所有 SCSI 设备的相关信息，/proc/N 中存储

的则是系统当前正在运行的进程的相关信息，其中 N 为正在运行的进程 PID（在某进程结束后其相关目录则会消失）。

大多数虚拟文件可以使用文件查看命令如 cat、more 或者 less 进行查看，有些文件信息表述的内容可以一目了然，但也有文件的信息却不怎么具有可读性。不过，这些可读性较差的文件在使用一些命令如 apm、free、lspci 或 top 查看时却可以有着不错的表现。

二、/proc 目录中的目录操作函数

1、proc_mkdir()

功能：用于在/proc 目录中添加一个目录。

头文件：linux/proc_fs.h

函数原型为：static inline struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent)

参数说明：

name 建立的目录的名称

parent 父目录指针。如果为 NULL，则在 /proc 目录下建立。

返回值：成功时返回创建的目录的指针；失败时返回 NULL。

2、proc_dir_entry 结构体

头文件：定义于 fs/proc/internal.h，使用时添加 linux/proc_fs.h 即可。

结构体对 proc 文件系统目录项的对象做了定义。分为三个部分：

proc 目录项的属性，例如：low_ino 此目录项对应的索引节点的值；mode 对应的索引节点的类型和权限模式；size 对应的索引节点的大小；name 目录项的名称；

proc 目录项的方法，例如：proc_iops 索引节点的操作的集合；proc_fops 文件操作的集合；

linux 内核使用的属性，例如 in_use, pde_unload_lock 等等。

3、proc_remove()

功能：用于移除 /proc 目录下的一个目录及其子目录

函数原型为：static inline void proc_remove(struct proc_dir_entry *de)

参数为要删除的目录的指针（即 proc_mkdir()函数的返回值）。

7.3.3.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/7/task3 目录下。

```
[root@openEuler ~]# cd tasks_k/7/task3
[root@openEuler task3]# ls
```



```
Makefile  proc_mkdir.c
```

步骤 2 编译源文件.

```
[root@openEuler task3]# make
make -C /root/kernel M=/root/tasks_k/7/task3 modules
make[1]: Entering directory '/root/kernel'
CC [M]  /root/tasks_k/7/task3/proc_mkdir.o
Building modules, stage 2.
MODPOST 1 modules
CC      /root/tasks_k/7/task3/proc_mkdir.mod.o
LD [M]  /root/tasks_k/7/task3/proc_mkdir.ko
make[1]: Leaving directory '/root/kernel'
```

步骤 3 对比加载内核模块前后的文件系统结果.

```
[root@openEuler task3]# find /proc/ -name myproc
[root@openEuler task3]# insmod proc_mkdir.ko
[root@openEuler task3]# find /proc/ -name myproc
/proc/myproc
```

步骤 4 卸载内核模块，并查看结果.

```
[root@openEuler task3]# rmmod proc_mkdir
[root@openEuler task3]# find /proc/ -name myproc
[root@openEuler task3]# dmesg | tail -n2
[115355.971491] Start proc_mkdir module...
[115370.638388] Exit proc_mkdir module...
```

由实验结果可见：

当未加载内核模块时，/proc 下无 myproc 目录；当加载内核模块后，/proc 下可查找到 myproc 目录。当卸载内核模块后，/proc 下无 myproc 目录。

7.3.4 使用 sysfs 文件系统传递内核模块参数

7.3.4.1 相关知识

一、sysfs 文件系统

1、基本概念

内核子系统或设备驱动可以直接编译到内核，也可以编译成模块。如果编译到内核，可以通过内核启动参数来向它们传递参数；如果编译成模块，则可以通过命令行在插入模块时传递参数，或者在运行时，通过 sysfs 来设置或读取模块数据。

sysfs 是一个基于内存的虚拟文件系统，可以看成与 proc、devfs 和 devpty 同类别的文件系统；它的作用是将内核信息以文件的方式提供给用户程序使用。sysfs 文件系统要求总是被挂载在/sys 挂载点上，这个文件系统不仅可以把设备（devices）和驱动程序（drivers）的信息从内核输出到用户空间，也可以用来对设备和驱动程序做设置。

sysfs 提供一种机制，使得可以显式地描述内核对象、对象属性及对象间关系。sysfs 有两组接口，一组针对内核，用于将设备映射到文件系统中；另一组针对用户程序，用于读取或操作这些设备。下表描述了内核中的 sysfs 要素及其在用户空间的表现：

sysfs在内核中的组成要素	在用户空间的显示
内核对象（kobject）	目录
对象属性（attribute）	文件
对象关系（relationship）	链接（Symbolic Link）

2、sysfs 与 sysctl 区别

sysctl：是内核的一些控制参数，其目的是方便用户对内核的行为进行控制；

sysfs：仅仅是把内核的 kobject 对象的层次关系与属性开放给用户查看，因此 sysfs 的绝大部分是只读的，模块作为一个 kobject 也被出口到 sysfs，模块参数则是作为模块属性出口的，内核实现者为模块的使用提供了更灵活的方式，允许用户设置模块参数在 sysfs 的可见性并允许用户在编写模块时设置这些参数在 sysfs 下的访问权限，然后用户就可以通过 sysfs 来查看和设置模块参数，从而使得用户能在模块运行时控制模块行为。

二、module_param 宏

对于模块而言，声明为 static 的变量都可以通过命令行来设置，但要想在 sysfs 下可见，必须通过宏 module_param 来显式声明。

module_param 宏有三个参数：

第一个为参数名，即已经定义的变量名；

第二个参数则为变量类型，可用的类型有 byte, short, ushort, int, uint, long, ulong, charp 和 bool 或 invbool，分别对应于 c 语言类型 char, short, unsigned short, int, unsigned int, long, unsigned long, char *和 int，用户也可以自定义类型 xxx（如果用户自己定义了 param_get_XXX, param_set_XXX 和 param_check_XXX）。

第三个参数用于指定访问权限，如果为 0，该参数将不出现在 sysfs 文件系统中，允许的访问权限为 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH 和 S_IWOTH 的组合，它们分别对应于用户读，用户写，用户组读，用户组写，其他用户读和其他用户写，因此用文件的访问权限设置是一致的。

module_param 宏使用示例：

```
module_param(a, int, 0);
MODULE_PARM_DESC(a, "An invisible int under sysfs");
module_param(b, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(b, "An visible int under sysfs");
module_param(c, charp, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

MODULE_PARM_DESC(c, "An visible string under sysfs");注：MODULE_PARM_DESC()是对模块的参数进行描述，使用 modinfo 查看模块信息时可见。

三、使用示例

当模块加载后，可通过/sys 目录与模块进行交互，通常的做法是使用 echo 向内核传递参数。例如：

```
echo -n 'example' > /sys/module/module_exam/parameters/c
```

说明：用 echo 传递字符串时，使用 -n 选项；指定传递参数时，在 /sys/module/ 目录下找到自定义的模块名（如此处示例的 module_exam），模块名目录下的 parameters 目录，即为该模块的参数列表，选定需要传递的参数名即可（如此处示例的参数 c）；必须使用 su 切换到 root 模式下，使用 sudo 命令会出现权限受限的问题。

7.3.4.2 参考答案

步骤 1 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件。在这里我们的示例源文件存放在 tasks_k/7/task4 目录下。

```
[root@openEuler ~]# cd tasks_k/7/task4
[root@openEuler task4]# ls
Makefile  sysfs_exam.c
```

步骤 2 编译源文件。

```
[root@openEuler task4]# make
make -C /root/kernel M=/root/tasks_k/7/task4 modules
make[1]: Entering directory '/root/kernel'
CC [M] /root/tasks_k/7/task4/sysfs_exam.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/tasks_k/7/task4/sysfs_exam.mod.o
LD [M] /root/tasks_k/7/task4/sysfs_exam.ko
make[1]: Leaving directory '/root/kernel'
```

步骤 3 加载编译完成的内核模块，并查看加载结果。

```
[root@openEuler task4]# find /sys/module/ -name 'sysfs_exam'
[root@openEuler task4]# insmod sysfs_exam.ko
[root@openEuler task4]# find /sys/module/ -name 'sysfs_exam'
/sys/module/sysfs_exam
[root@openEuler task4]# modinfo sysfs_exam.ko
filename:      /root/tasks_k/7/task4/sysfs_exam.ko
license:       GPL
srcversion:    3088958EF3BE7DAB164D005
depends:
name:          sysfs_exam
vermagic:      4.19.154 SMP mod_unload modversions aarch64
parm:          a:An invisible int under sysfs (int)
```

```

parm:          b:An visible int under sysfs (int)
parm:          c:An visible string under sysfs (charp)
[root@openEuler task4]# dmesg | tail -n4
[168419.281573] Start sysfs_exam module...
[168419.281576] a = 0
[168419.295572] b = 0
[168419.295676] c = 'Hello, World'
[root@openEuler task4]# ls -al /sys/module/sysfs_exam/parameters/
total 0
drwxr-xr-x. 2 root root    0 Nov 13 08:43 .
drwxr-xr-x. 6 root root    0 Nov 13 08:43 ..
-rw-r--r--. 1 root root 65536 Nov 13 08:44 b
-rw-r--r--. 1 root root 65536 Nov 13 08:44 c
[root@openEuler task4]# cat /sys/module/sysfs_exam/parameters/c
Hello, World
[root@openEuler task4]# echo -n 'Happy new year!' > /sys/module/sysfs_exam//parameters/c
[root@openEuler task4]# cat /sys/module/sysfs_exam/parameters/c
Happy new year!

```

步骤 4 卸载内核模块，并查看结果。

```

[root@openEuler task4]# rmmod sysfs_exam
[root@openEuler task4]# dmesg | tail -n4
[168613.260984] Exit sysfs_exam module...
[168613.260986] a = 0
[168613.261299] b = 0
[168613.261403] c = 'Happy new year!'

```

由实验结果可见：

(1) 当未加载内核模块时，当前系统中无自定义的内核模块“sysfs_exam”；

当加载内核模块后，当前系统可见自定义的文件系统“sysfs_exam”；

(2) 使用 modinfo 查看内核模块信息，可见 a、b、c 三个参数的描述：

```

parm:          a:An invisible int under sysfs (int)
parm:          b:An visible int under sysfs (int)
parm:          c:An visible string under sysfs (charp)

```

即参数 a 在/sys 中不可见，b、c 在/sys 中可见，ls -al /sys/module/sysfs_exam/parameters/的输出列表可验证；

(3) 可使用 echo 向模块传递参数值来改变指定参数的值。

8 实验八 网络管理

8.1 实验介绍

本实验通过编写 C 源码程序实现客户端与服务端的简单通信，使用 tshark 抓取该通信数据包以及查看使用 setsockopt 发送的带 IP 记录路由选项的数据包中是否包含了记录路由选项等任务操作，让学生们了解并掌握操作系统中的网络管理。

8.1.1 任务描述

- 编写 C 源码，基于 socket 的 UDP 发送接收程序，实现客户端与服务端的简单通信。客户端从命令行输入中读取要发送的内容，服务端接收后实时显示。
- 基于任务 1 的服务端与客户端程序运行时，使用 tshark 抓取该通信数据包。
- 基于任务 1 的客户端与服务端，使用 setsockopt 发送一个带 IP 记录路由选项的数据包；使用 tshark 查看发送的数据包中是否包含了记录路由选项。

8.2 实验目的

- 正确编写满足功能的源文件，正确编译。
- 正常加载、卸载内核模块；且内核模块功能满足任务所述。
- 了解操作系统的网络管理。

8.3 实验任务

8.3.1 编写基于 socket 的 UDP 发送接收程序

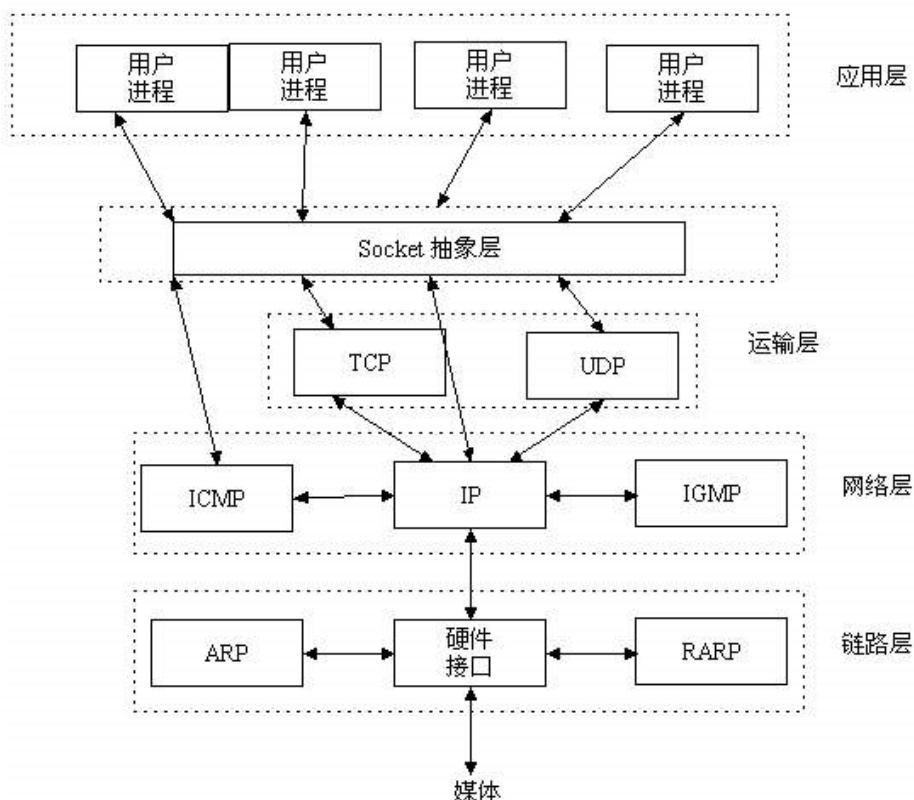
8.3.1.1 相关知识

一、Socket 是做什么的？

socket 起源于 Unix，而 Unix/Linux 基本哲学之一就是“一切皆文件”，都可以用“打开 open→读写 write/read→关闭 close”模式来操作。Socket 就是该模式的一个实现，socket 即是一种特殊的文件，一些 socket 函数就是对其进行的操作（读/写 IO、打开、关闭）。

简言之，socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，socket 其实就是一个门面模式，它把复杂的 TCP/IP 协议族隐藏在 socket 接口后面，对用户来说，一组简单的接口就是全部，让 socket 去组织数据，以符合指定的协议。

TCP/IP 协议族包括运输层、网络层、链路层，而 socket 所在位置如图，Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层。

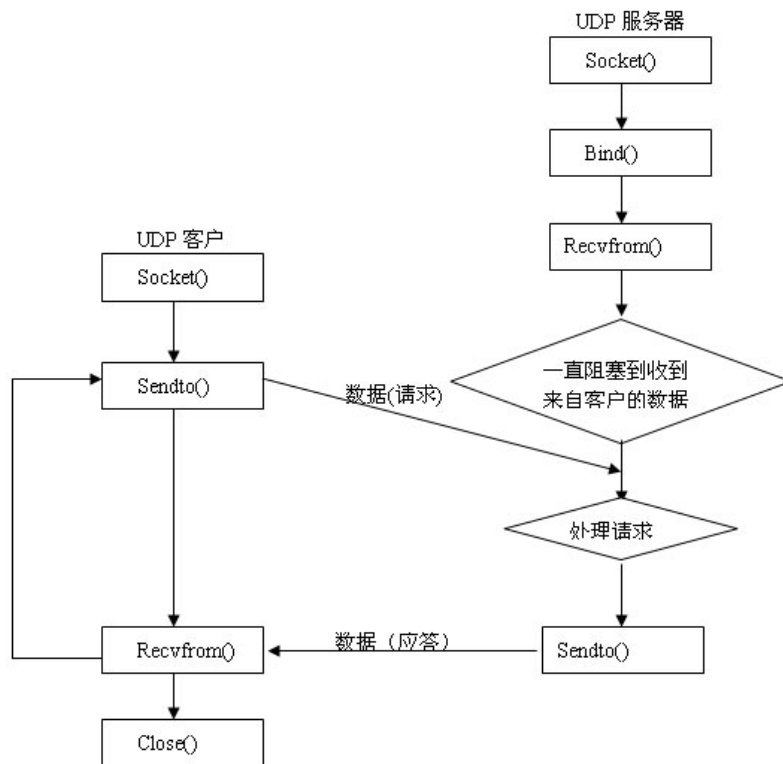


二、UDP

在 TCP/IP 模型中，UDP 为网络层以上和应用层以下提供了一个简单的接口。UDP 只提供数据的不可靠传递，它一旦把应用程序发给网络层的数据发送出去，就不保留数据备份（所以 UDP 有时候也被认为是不可靠的数据报协议）。UDP 在 IP 数据报的头部仅仅加入了复用和数据校验（字段）。

UDP 首部字段由 4 个部分组成，其中两个是可选的。各 16bit 的来源端口和目的端口用来标记发送和接受的应用进程。因为 UDP 不需要应答，所以来源端口是可选的，如果来源端口不用，那么置为零。在目的端口后面是长度固定的以字节为单位的长度域，用来指定 UDP 数据报包括数据部分的长度，长度最小值为 8byte。首部剩下的 16bit 是用来对首部和数据部分一起做校验和（Checksum）的，这部分是可选的，但在实际应用中一般都使用这一功能。

由于缺乏可靠性且属于非连接导向协定，UDP 应用一般必须允许一定量的丢包、出错和复制粘贴。但有些应用，比如 TFTP，如果需要则必须在应用层增加根本的可靠机制。但是绝大多数 UDP 应用都不需要可靠机制，甚至可能因为引入可靠机制而降低性能。流媒体（串流技术）、即时多媒体游戏和 IP 电话（VoIP）一定就是典型的 UDP 应用。如果某个应用需要很高的可靠性，那么可以用传输控制协议（TCP 协议）来代替 UDP。



三、socket 编程 API

1、int socket(int domain, int type, int protocol);

(1) 功能：根据指定的地址族、数据类型和协议来分配一个 socket 的描述字及其所用的资源。

(2) 头文件（C 库）：

```
#include <sys/types.h>
```

```
#include<sys/socket.h>
```

(3) 参数说明：

domain：地址族，常用的有：AF_INET、AF_INET6、AF_LOCAL、AF_ROUTE，其中 AF_INET 代表使用 ipv4 地址。

type：socket 类型，常用的 socket 类型有：SOCK_STREAM、SOCK_DGRAM、SOCK_RAW、SOCK_PACKET、SOCK_SEQPACKET 等。SOCK_STREAM----提供有序的、可靠的、双向的和基于连接的字节流，使用带外数据传送机制，为 Internet 地址族使用 TCP。SOCK_DGRAM----支持无连

接的、不可靠的和使用固定大小（通常很小）缓冲区的数据报服务，为 Internet 地址族使用 UDP。

protocol：协议。常用的协议有：IPPROTO_IP、IPPROTO_TCP、IPPROTO_UDP、IPPROTO_SCTP、IPPROTO_TIPC 等。

(4) 返回值：调用成功则返回新创建的套接字描述符；失败就返回 INVALID_SOCKET。

2、int sendto(int sockfd, const void* msg, int len, int flags, const struct sockaddr *to, int tolen);

(1) 功能：进行无连接的 UDP 通讯使用，使用时，数据会在没有建立任何网络连接的网络上传输。

(2) 头文件（C 库）：

```
#include <sys/types.h>
```

```
#include<sys/socket.h>
```

(3) 参数说明：

sockfd：指与远程程序连接的套接字，即 socket()函数的返回值。

msg：是一个指针，指向发送的信息的地址。

len：指发送信息的长度。

flags：通常是 0。

to：一个指向 struct sockaddr 结构的指针，里面包含了远程主机和端口数据。

tolen：指出了 struct sockaddr 的大小，通常用 sizeof(struct sockaddr)。

(4) 返回值：正常时返回真正发送的数据的大小；错误时：返回-1。

3、int bind(int sockfd, struct sockaddr* addr, socklen_t addrlen)

(1) 功能：将指定地址与指定套接口绑定。

(2) 头文件（C 库）：

```
#include <sys/types.h>
```

```
#include<sys/socket.h>
```

(3) 参数说明：

sockfd：指定地址与哪个套接字绑定，即 socket()函数调用返回的套接字。调用 bind 的函数之后，该套接字与一个相应的地址关联，发送到这个地址的数据可以通过这个套接字来读取与使用。

addr：指定地址。这是一个地址结构，并且是一个已经经过填写的有效的地址结构。调用 bind 之后这个地址与参数 sockfd 指定的套接字关联，从而实现上面所说的效果。

addrlen：地址的长度。正如大多数 socket 接口一样，内核不关心地址结构，当它复制或传递地址给驱动的时候，它依据这个值来确定需要复制多少数据。这已经成为 socket 接口中最常见的参数之一了。

bind 函数并不是总是需要调用的，只有用户进程想与一个具体的地址或端口相关联的时候才需要调用这个函数。如果用户进程没有这个需要，那么程序可以依赖内核的自动的选址机制来完成自动地址选择，而不需要调用 bind 的函数。

(4) 返回值：0 —成功，-1 —失败。

4、int recvfrom(int s,void *buf,int len,unsigned int flags ,struct sockaddr *from ,int *fromlen);

(1) 功能：接收远程主机经指定的 socket 传来的数据，并把数据存到由参数 buf 指向的内存空间。

(2) 头文件（C 库）：

```
#include<sys/types.h>
```

```
#include<sys/socket.h>
```

(3) 参数说明：

s：表示正在监听的端口的套接字，即函数 socket()的返回值；

buff：表示接收数据缓冲区，接收到的数据将放在这个指针所指向的内存空间中；

len：表示接收数据缓冲区大小/可接收数据的最大长度，系统根据这个值来确保接收缓冲区的安全，防止溢出；

flags：一般设 0；

from：是一个 struct sockaddr 类型的变量，该变量保存发送方的 IP 地址及端口号；

fromlen：表示 sockaddr 的结构长度，可以使用 sizeof(struct sockaddr_in)来获得。

(4) 返回值：成功则返回接收到的字符数；失败则返回-1，错误原因存于 errno 中。

错误代码：

EBADF	参数 s 非合法的 socket 处理代码
EFAULT	参数中有一指针指向无法存取的内存空间。
ENOTSOCK	参数 s 为一文件描述词，非 socket。
EINTR	被信号所中断。
EAGAIN	此动作会令进程阻断，但参数 s 的 socket 为不可阻断。
ENOBUFS	系统的缓冲内存不足
ENOMEM	核心内存不足
EINVAL	传给系统调用的参数不正确。

8.3.1.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括服务端源文件和客户端源文件。在这里我们的示例源文件存放在 tasks_k/8/task1 目录下。


```
[root@openEuler ~]# cd tasks_k/8/task1
[root@openEuler task1]# ls
client.c  server.c
```

步骤 2 编译源文件。

```
[root@openEuler task1]# gcc server.c -o server
[root@openEuler task1]# gcc client.c -o client
[root@openEuler task1]# ls
client  client.c  server  server.c
```

步骤 3 开启两个终端，一个运行客户端，一个运行服务端；client 中输入发送的消息回车后，server 端即能收到。

Client 端

```
[root@openEuler task1]# ./client
```

Server 端

```
[root@openEuler task1]# ./server
```

Client 端

```
Please enter the content to be sent:
Hello World!
Please enter the content to be sent:
Happy New Year!
Please enter the content to be sent:
```

Server 端

```
The message received is: Hello World!
The message received is: Happy New Year!
```

8.3.2 使用 tshark 抓包

8.3.2.1 相关知识

tshark 是 wireshark 中提供的 Linux 命令行工具，tshark 不仅有抓包的功能，还带了解析各种协议的能力。由于 wireshark 的运行需要 GUI（图形用户界面，Graphical User Interface）的环境，但是 openEuler 系统的缺省安装是不含 GUI 的，此时就可以使用 tshark 进行抓包。

8.3.2.2 实验步骤

步骤 1 安装 wireshark。

```
dnf install -y wireshark
```

步骤 2 用 ifconfig 查看网卡信息。

```
[root@openEuler ~]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 192.168.1.5  netmask 255.255.255.0  broadcast 192.168.1.255
```

```

inet6 fe80::f816:3eff:fee5:4ee prefixlen 64 scopeid 0x20<link>
ether fa:16:3e:e5:04:ee txqueuelen 1000 (Ethernet)
RX packets 3725 bytes 421060 (411.1 KiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 2911 bytes 581945 (568.3 KiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

```

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

由于任务 1 的 client 与 server 均运行于本机，因此指定的网络接口应该是回环地址接口 lo。

步骤 3 开启三个终端，一个运行客户端，一个运行服务端，一个运行 tshark；client 中输入发送的消息回车后，server 端即能收到。

Client 端

```

[root@openEuler task1]# ./client
Please enter the content to be sent:
Hello World!
Please enter the content to be sent:
Happy New Year!
Please enter the content to be sent:

```

Server 端

```

[root@openEuler task1]# ./server
The message received is: Hello World!
The message received is: Happy New Year!

```

tshark 端，直接把抓包结果输出到命令行

```

[root@openEuler ~]# tshark -i lo -n -f 'udp port 40000'
Running as user "root" and group "root". This could be dangerous.
Capturing on 'Loopback'
  1 0.000000000    127.0.0.1 → 127.0.0.1      UDP 55 46360 → 40000 Len=13
  2 7.174615157    127.0.0.1 → 127.0.0.1      UDP 58 46360 → 40000 Len=16

```

8.3.3 使用 setsockopt 发送记录路由选项

8.3.3.1 相关知识

getsockopt/setsockopt 系统调用

1、功能描述：

获取或者设置与某个套接字关联的选项。选项可能存在于多层协议中，它们总会出现在最上面的套接字层。当操作套接字选项时，选项位于的层和选项的名称必须给出。为了操作套接字层的选项，应该将层的值指定为 SOL_SOCKET。为了操作其它层的选项，控制选项的合适协议号必须给出。例如，为了表示一个选项由 IP 协议解析，层应该设定为协议号 IPPROTO_IP。

2、用法：

```
//头文件
#include <sys/types.h>
#include <sys/socket.h>

//函数原型
int getsockopt(int sock, int level, int optname, void *optval, socklen_t *optlen);
int setsockopt(int sock, int level, int optname, const void *optval, socklen_t optlen);
```

3、参数说明：

sock (套接字)：将要被设置或者获取选项的套接字。

level (级别)：指定选项代码的类型/选项所在的协议层。支持：

SOL_SOCKET：通用套接字选项

IPPROTO_IP：IP 套接字选项

IPPROTO_TCP：TCP 套接字选项

optname (选项名)：需要访问的选项名。

optval(选项值)：是一个指向变量的指针，对于 getsockopt()，指向返回选项值的缓冲；对于 setsockopt()，指向包含新选项值的缓冲。类型：整型；支持套接口结构或其他结构类型，如：linger{}，timeval{ }。

optlen(选项长度)：optval 的大小。对于 getsockopt()，作为入口参数时，选项值的最大长度；作为出口参数时，选项值的实际长度。对于 setsockopt()，现选项的长度。

4、返回值说明：成功执行时，返回 0。失败返回-1，errno 被设为以下的某个值：

EBADF：sock 不是有效的文件描述词

EFAULT：optval 指向的内存并非有效的进程空间

EINVAL：在调用 setsockopt()时，optlen 无效

ENOPROTOOPT：指定的协议层不能识别选项

ENOTSOCK：sock 描述的不是套接字

8.3.3.2 实验步骤

步骤 1 正确编写满足功能的源文件，包括服务端源文件和客户端源文件（服务端同任务 1，客户端添加了记录路由代码）。在这里我们的示例源文件存放在 tasks_k/8/task3 目录下。

```
[root@openEuler ~]# cd tasks_k/8/task3
```

```
[root@openEuler task3]# ls
client.c  server.c  setsockopt.xml
```

步骤 2 编译源文件。

```
[root@openEuler task3]# gcc server.c -o server
[root@openEuler task3]# gcc client.c -o client
[root@openEuler task3]# ls
client  client.c  server  server.c
```

步骤 3 开启三个终端，一个运行客户端，一个运行服务端，一个运行 tshark。

Client 端

```
[root@openEuler task3]# ./client
Please enter the content to be sent:
Hello!
Please enter the content to be sent:
How are you?
Please enter the content to be sent:
Nice to meet you!
```

Server 端

```
[root@openEuler task3]# ./server
The message received is: Hello!
The message received is: How are you?
The message received is: Nice to meet you!
```

tshark 端，直接把抓包结果输出到命令行

```
[root@openEuler task3]# tshark -i lo -n -f 'udp port 40000' -T pdml > ./setsockopt.xml
Running as user "root" and group "root". This could be dangerous.
Capturing on 'Loopback'
3
```

请在重定向的 setsockopt.xml 文件中查看相应路由内容（client 与 server 有 3 次通信，则 setsockopt.xml 文件中会有 3 段记录路由的内容）。