

# 重庆邮电大学

课程设计（操作系统）

实验指导书

编制单位：计算机专业实验中心

编制时间：2022 年 6 月

# 目 录

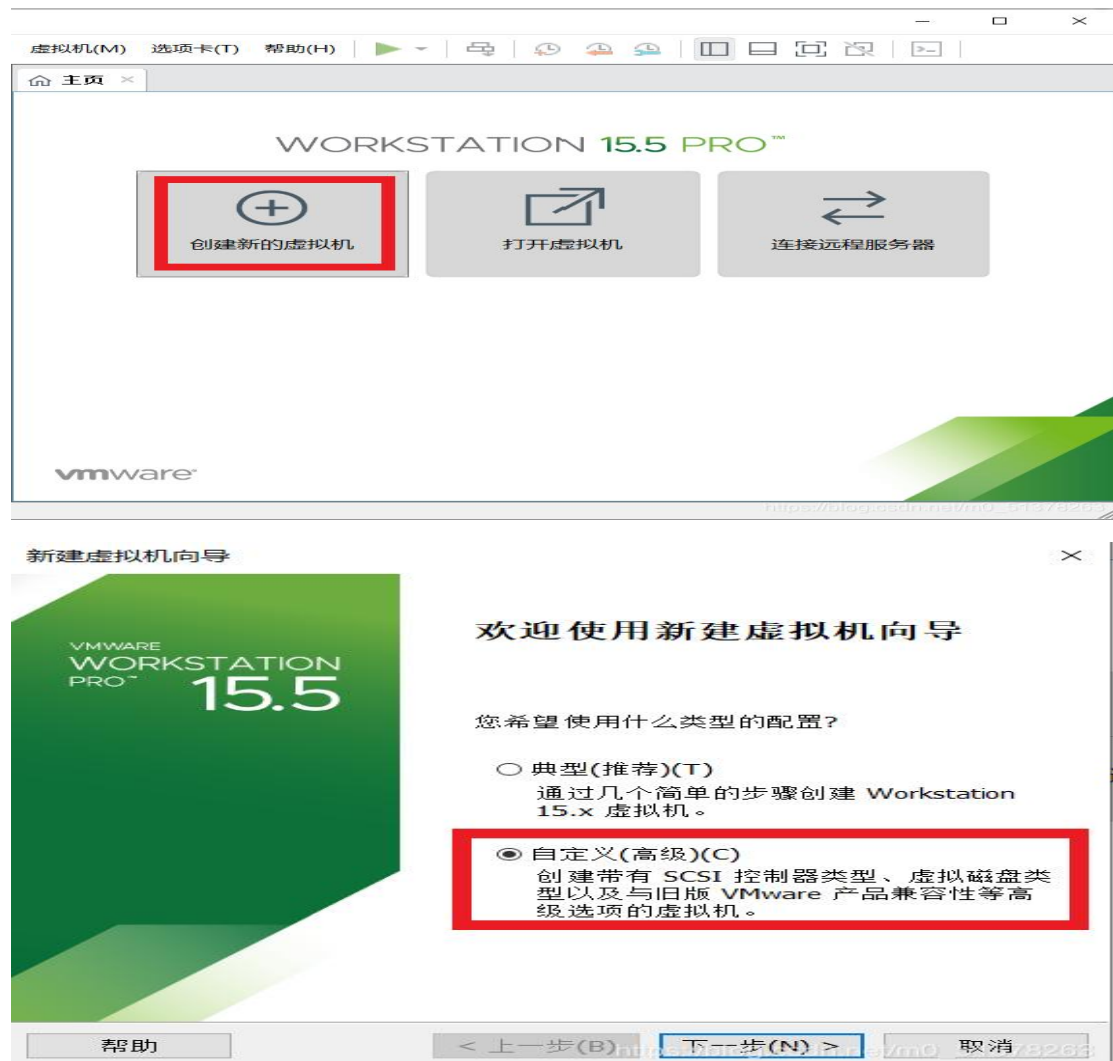
第一部分 Linux 系统安装 .....	3
第二部分 内核实验 .....	10
实验一 内核模块编程入门 .....	10
实验二 内存管理 .....	16
实验三 进程管理 .....	20
实验四 中断和异常管理 .....	27
实验五 设备管理 .....	41
实验六 文件系统 .....	46
实验七 网络管理 .....	57
第三部分 综合实验大作业 .....	66

# 第一部分 Linux 系统安装

随着 Centos 的逐步转移，Centos8 在 2021 年 12 月 31 号将停止维护，Centos7 也将于 2024 年逐步停止维护，因此本实验我们使用华为服务器操作系统 EulerOS，开源后命名为 openEuler，他基于 Linux 稳定系统内核，支持鲲鹏处理器和容器虚拟化技术，特性包括系统高可靠、高安全以及高保障。openEuler 拥有三级智能调度，可以将多进程并发时延缩短 60%，而且还可以智能自动有规划，可将 Web 服务器性能提升 137%。openEuler 是一个 Linux 的发行版，所有开发者、合作伙伴、开源爱好者共同参与，围绕客户的场景进行创新，有更多新的想法产生，创建多样性计算场景最佳操作系统。

## 一、安装 VMware Workstation 虚拟机

主要步骤如以下图所示：



## 选择客户机操作系统

此虚拟机中将安装哪种操作系统？

客户机操作系统

☐ Microsoft Windows(W)

☒ Linux(L)

☐ VMware ESX(X)

☐ 其他(O)

版本(V)

其他 Linux 4.x 64 位

帮助 < 上一步(B) 下一步(N) > 取消

## 处理器配置

为此虚拟机指定处理器数量。

处理器

处理器数量(P): 1

每个处理器的内核数量(C): 2

处理器内核总数: 2

帮助 < 上一步(B) 下一步(N) > 取消

## 此虚拟机的内存

您要为此虚拟机使用多少内存？

指定分配给此虚拟机的内存量。内存大小必须为 4 MB 的信数。

此虚拟机的内存(M): 4096 MB

128 GB -

64 GB -

32 GB -

16 GB -

8 GB -

4 GB -

2 GB -

1 GB -

512 MB -

256 MB -

128 MB -

64 MB -

32 MB -

16 MB -

8 MB -

4 MB -

最大推荐内存: 6.0 GB

推荐内存: 768 MB

客户机操作系统最低推荐内存: 32 MB

帮助 < 上一步(B) 下一步(N) > 取消

## 指定磁盘容量

磁盘大小为多少?

最大磁盘大小 (GB)(S): 

针对 其他 Linux 4.x 64 位 的建议大小: 8 GB

☐ 立即分配所有磁盘空间(A)。

分配所有容量可以提高性能，但要求所有物理磁盘空间立即可用。如果不立即分配所有空间，虚拟磁盘的空间最初很小，会随着您向其中添加数据而不断变大。

☐ 将虚拟磁盘存储为单个文件(O)☒ 将虚拟磁盘拆分成多个文件(M)

拆分磁盘后，可以更轻松地在计算机之间移动虚拟机，但可能会降低大容量磁盘的性能。

帮助

&lt; 上一步(B)

下一步(N) &gt;

取消

openEuler 官网([https://repo.openeuler.org/openEuler-20.03-LTS-SP1/ISO/x86\\_64/](https://repo.openeuler.org/openEuler-20.03-LTS-SP1/ISO/x86_64/))

或

学校 <ftp://s@202.202.43.106> 下载 openEuler-20.03-LTS-SP1-x86\_64-dvd.iso 映像文件。

硬件 选项

设备	摘要
内存	4 GB
处理器	2
硬盘 (SCSI)	60 GB
CD/DVD (IDE)	正在使用文件 E:\虚拟机包\o...
网络适配器	NAT
USB 控制器	存在
声卡	自动检测
打印机	存在
显示器	自动检测

添加(A)...

移除(R)

设备状态

☐ 已连接(C)☒ 启动时连接(O)

连接

☐ 使用物理驱动器(P):

自动检测

☒ 使用 ISO 映像文件(M):

E:\虚拟机包\openEuler-20.03-l

浏览(B)...

高级(V)...



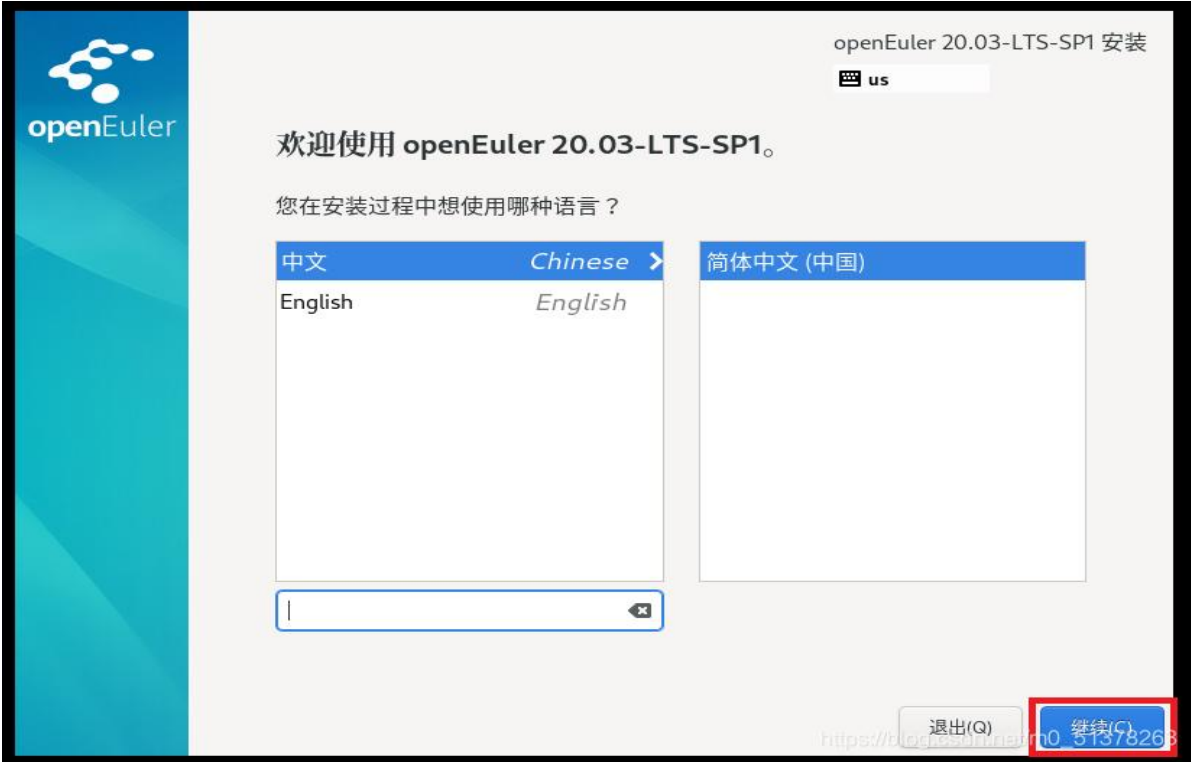
确定

取消

帮助

## 二、安装 openEuler 操作系统

主要步骤如以下图所示：



选最小安装



设 root 帐号密码，创建普通用户并设密码



安装完 openEuler 20.03 LTS，重启，登陆界面如下：

```
Authorized users only. All activities may be monitored and reported.
localhost login: root
Password:
Last failed login: Mon Mar 30 00:35:38 CST 2020 on tty1
There was 1 failed login attempt since the last successful login.

Authorized users only. All activities may be monitored and reported.

Welcome to 4.19.90-2003.4.0.0036.oe1.x86_64

System information as of time: Mon Mar 30 00:35:49 CST 2020

System load:      1.56
Processes:        150
Memory used:      6.7%
Swap used:        0.0%
Usage On:         9%
IP address:       192.168.233.229
Users online:     1

[root@localhost ~]#
```

修改 yum 源：

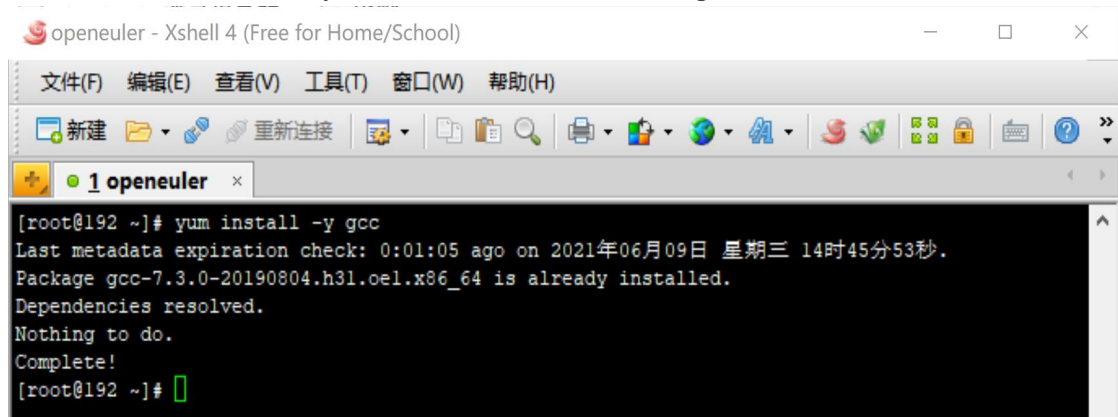


修改/etc/yum.repos.d/openEuler\_x86\_64.repo 文件，增加如下内容

```
[root@localhost ~]# vi /etc/yum.repos.d/openEuler_x86_64.repo
```

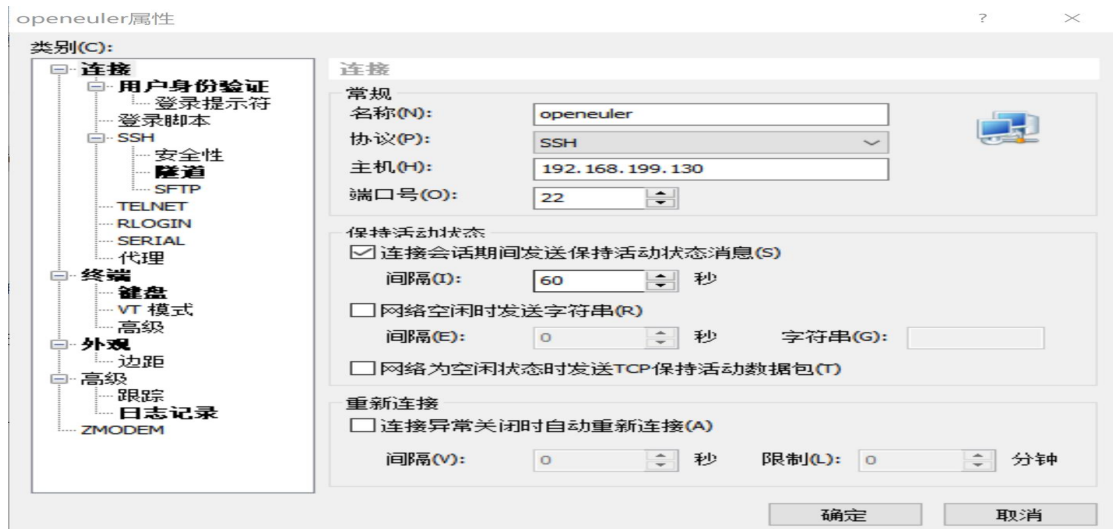
```
[base]
name=openeuler-$releaseve
baseurl=https://repo.openeuler.org/openEuler-20.03-LTS/everything/x86_64/
gpgcheck=1
gpgkey=https://repo.openeuler.org/openEuler-20.03-LTS/everything/x86_64/RPM-GPG-KEY-openeuler
```

保存退出，可以正常使用 yum 安装软件了，如下例安装 gcc：



### 三 SSH 远程登录

Xshell,Xftp 安装与使用（略，参考学院 ftp 上相应教程）。



### 四 Linux 常用命令使用

请在 Linux 系统上练习如下命令，以达到熟练掌握。



man 阅读联机帮助

mkdir 创建目录

cd 切换目录

touch 创建空文件

echo 创建带有内容的文件

cat 查看文件内容

cp 拷贝

mv 移动或者重命名

rm 删除文件

find 在文件系统中搜索某文件

wc 统计文本中的行数，字数，字符数

grep 在文本文件中查找某个字符串

rm dir 删除空目录（dir 是指一个目录名）

ln 创建链接文件

more/less 分页显示文本文件内容

head/tail 显示文件头，尾内容

stat 显示指定文件详细信息，比 ls 更详细

who 显示在线登录用户

whoami 显示当前操作用户

hostname 显示主机名

uname 显示系统信息

top 查看进程信息

ps 显示瞬间进入状态

du 查看目录大小

df 查看磁盘大小

ifconfig 查看网络情况

ping 测试网络连接

kill -9 进程号 强制终止进程

## 第二部分 内核实验

### 实验一 内核模块编程入门

#### 1.相关知识

##### 一、内核模块

Linux 提供了一种动态加载内核的机制，这种机制称为模块(Module)。

内核模块是具有独立功能的程序。

它可以被单独编译，但是不能单独运行，它的运行必须被链接到内核作为内核的一部分在内核空间中运行。

模块具有一下特点：

- 1) 模块本身不被编译入内核映像，从而控制了内核的大小；
- 2) 模块一旦被加载，它就和内核中的其它部分完全一样。

##### 二、内核模块的基本结构

内核模块一般具有如下形式的基本结构：

```
#include<linux/module.h>           //包含了对模块的结构定义以及模块的版本控制
MODULE_LICENSE("GPL");              //声明 GPL 版权
static __init module_init(void){    //加载模块
    . . . . .
}

static __exit module_exit(void){     //卸载模块
    . . . . .
}
module_init(module_init);
module_exit(module_exit);
```

说明：

1、任何模块程序的编写都需要包含 linux/module.h 这个头文件，该文件包含了对模块的结构定义以及模块的版本控制。

2、函数 module\_init ()和函数 module\_exit ()是模块编程中最基本的也是必须的两个函数。module\_init ()向内核注册模块所提供的新功能；module\_exit ()负责注销所有由模块注册

的功能。在 2.3.13 版本的 linux 内核以前，内核模块的初始化函数与卸载函数必须分别使用 `init_module` 与 `cleanup_module` 作为函数名；

从 2.3.13 版本的 linux 内核以后，可以自定义初始化函数与卸载函数的函数名，但同时必须使用 `module_init()` 与 `module_exit()` 指定初始化函数与卸载函数。

3、内核模块使用 `printk` 函数来输出字符串，而不是 `printf`。

这是由于基于 linux 内核的操作系统分为内核空间与用户空间，`printf` 是由标准 C 库函数所定义的输出函数，而标准 C 库函数运行于用户空间，同时又由于内核空间与用户空间不能直接通信，因此运行于内核空间的内核模块不能使用标准 C 库函数，也就不能使用 `printf` 函数，所以 linux 内核需要自己的输出函数 `printk`。

如：`printk("Hello World!\n");` 或 `printk(KERN_INFO "Hello World!\n");`

字符串 `KERN_INFO` 表示消息的优先级，`printk()` 的一个特点就是它对于不同优先级的消息进行不同的处理。

因为内核模块使用的 `printk` 输出是直接输出到内核缓存，可以使用 `dmesg` 查看相关信息。

### 三、Makefile 文件（注意 M 是大写，不是 makefile）：

一个工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中。`makefile` 定义了一系列的规则，来指定哪些文件需要先编译、哪些文件需要后编译、哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 `makefile` 就像一个 Shell 脚本一样，其中也可以执行操作系统的命令。

编写简单的 Makefile 文件，以上述的示例程序为例：

```
ifneq ($(KERNELRELEASE),)
    obj-m := main.o
else
    KERNELDIR ?= /root/raspberrypi-kernel //此处路径为内核源码路径，根据实际情况修改；该内核源码必须要经过编译，否则会报错。
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

说明：

1、`obj-m` 表示把文件 `main.o` 作为内核模块进行编译，不要编译到内核中，编译同时还会生成一个 `"main.ko"` 模块目标文件。

2、`KERNELDIR` 是内核源码路径，路径不正确会报错；且该内核源码必须要经过编译或预编译（进入内核源码根目录，依次执行 `make openeuler-raspi_defconfig && make prepare && make scripts`），不然也会报错。

3、`make -C` 指明了内核源码的所在目录。

4、执行过程解析：

- 1、`ifneq` (\$ (变量名) , 变量值)

`ifneq`是比较两个参数是否不相同，第二个参数空就是NULL，也就是不同，则执行else后的命令。

- 2、先执行 `make -C $(KDIR) M=$(PWD)`

为什么先直线这句呢？这就是makefile的工作原则，定义的变量（宏）只有在使用的时候才会扩展，即：`KDIR`、`PWD`会在使用的时候在去寻找定义。

- 3、make时只执行all：，make clean时执行clean：。

其实make和make all都可以执行all：的命令，另外也可以不加这个all：，这就相当于C语音中的label；实际上无论这个label时什么都要去执行第一句 `make -C $(KDIR) M=$(PWD)`，其他才会去匹配label，这就是个规则。

### 3、KERNELRELEASE的作用

进入正题，说下KERNELRELEASE作用及makefile的执行：

- 1、`ifneq` (\$(KERNELRELEASE),)会使makefile执行两次，为什么？

- 2、第一次读取makefile时，因为 `ifneq` 的存在使得先执行了 `make -C $(KDIR) M=$(PWD)`，关键就在这里了，注释里我也有说明，  
`-C $(KDIR)`：指明跳转到内核源码目录下读取那里的Makefile  
`M=$(PWD)`：表明然后返回到当前目录继续读入、执行当前的Makefile

- 3、从内核Makefile返回时，`KERNELRELEASE`已被定义，`kbuild`也被启动去解析`kbuild`语法的语句，make将继续读取else之前的内容，指明模块源码中各文件的依赖关系，以及要生成的目标模块名，即：**设置了obj-m，最后通过内核的makefile构造模块。**

### 四、内核源码准备

自行下载内核源码，解压、编译。请自行查阅相关资料。

### 五、编译 make 指令

在编写好 Makefile 之后，输入 `make` 指令即可进行编译。

此时用 `ls` 命令可以查看当前目录下生成的文件，其中 `main.ko` 文件就是模块目标文件。

### 六、加载模块

(1) 加载: `sudo insmod main.ko`      #将 main.ko 模块加载进内核

(2) 查看加载内容

`sudo dmesg | tail -n 2`    可以看到打印的信息:

`sudo lsmod | grep main`    可以查看所有名字中包含 main 的内核模块:

## 七、卸载模块

(1) 卸载: `sudo rmmmod main`    #卸载 main 模块。

(2) 查看卸载内容:    `sudo dmesg | tail -n 1`

## 2.任务描述

1. 编写内核模块，功能是打印“hello,world!”字符串。
2. 编写对应 Makefile 文件，并使用 make 编译上述内核模块。
3. 手动加载内核模块，查看加载内容。
4. 手动卸载上述内核模块。

## 3. 参考答案

### 一、参考答案源码

(1) helloworld.c

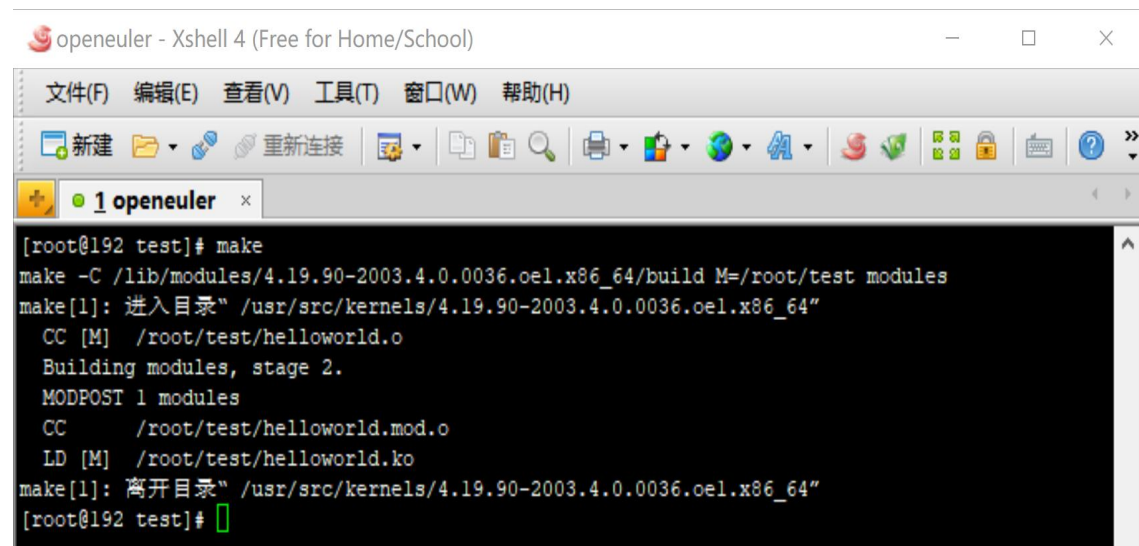
```
#include<linux/module.h>
MODULE_LICENSE("GPL");
int __init hello_init(void)
{
    printk("hello init\n");
    printk("hello,world!\n");
    return 0;
}
void __exit hello_exit(void)
{
    printk("hello exit\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

## (2) Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := helloworld.o
else
    KERNELDIR = /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

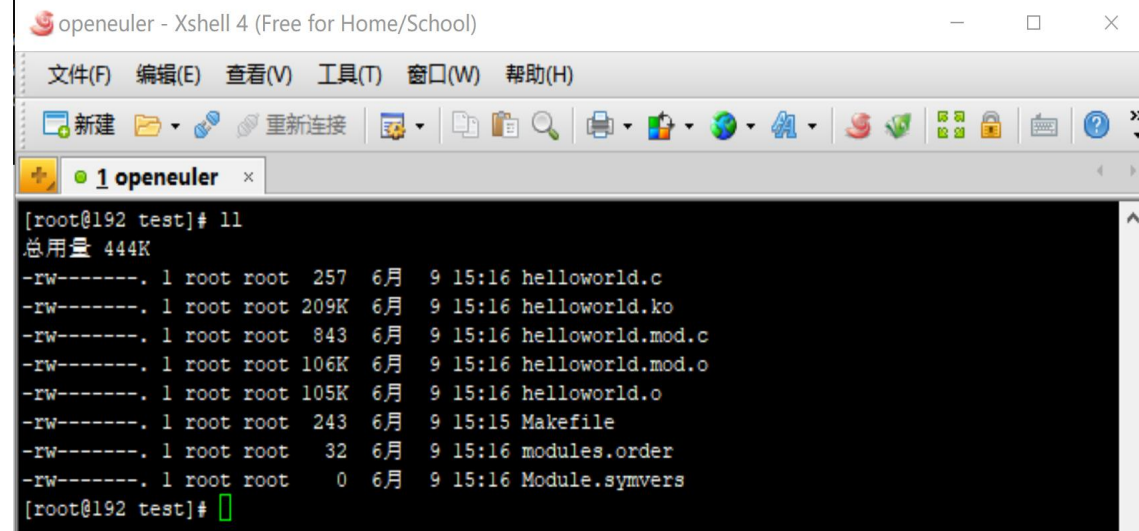
## 二、内核模块编译、加载与卸载

### (1) 执行 make 编译源码：



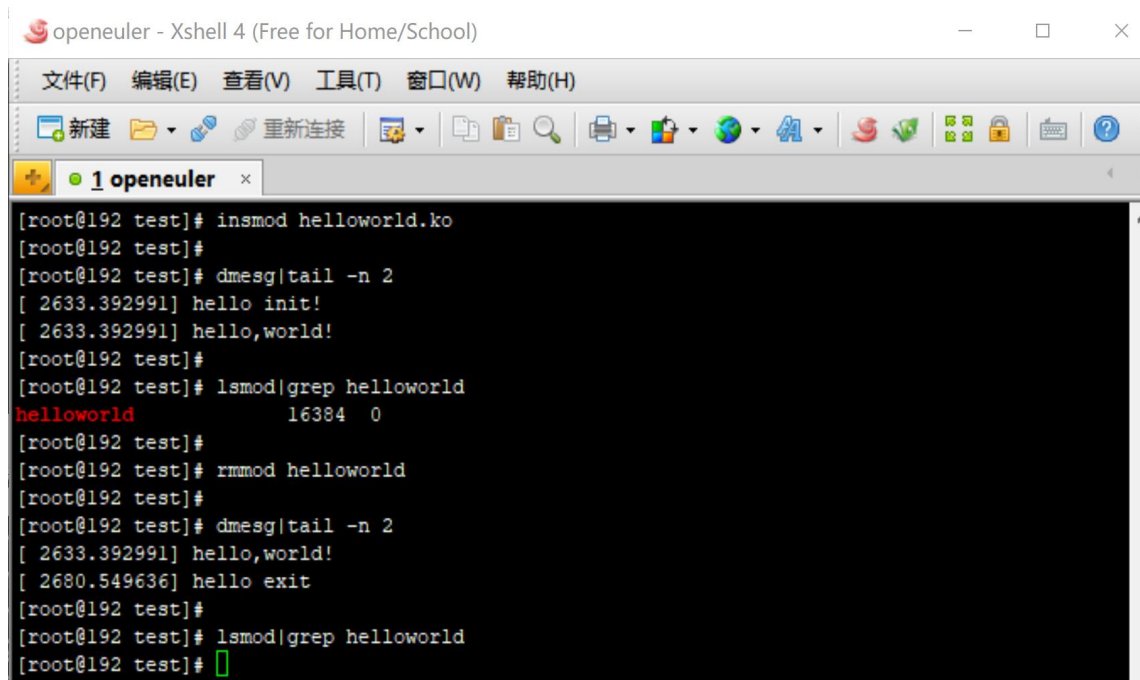
```
openeuler - Xshell 4 (Free for Home/School)
文件(F) 编辑(E) 查看(V) 工具(T) 窗口(W) 帮助(H)
新建 重新连接
1 openeuler x
[root@192 test]# make
make -C /lib/modules/4.19.90-2003.4.0.0036.oel.x86_64/build M=/root/test modules
make[1]: 进入目录 "/usr/src/kernels/4.19.90-2003.4.0.0036.oel.x86_64"
CC [M] /root/test/helloworld.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/test/helloworld.mod.o
LD [M] /root/test/helloworld.ko
make[1]: 离开目录 "/usr/src/kernels/4.19.90-2003.4.0.0036.oel.x86_64"
[root@192 test]#
```

### 编译完成后的文件列表：



```
openeuler - Xshell 4 (Free for Home/School)
文件(F) 编辑(E) 查看(V) 工具(T) 窗口(W) 帮助(H)
新建 重新连接
1 openeuler x
[root@192 test]# ll
总用量 444K
-rw-rw-r--. 1 root root 257 6月 9 15:16 helloworld.c
-rw-rw-r--. 1 root root 209K 6月 9 15:16 helloworld.ko
-rw-rw-r--. 1 root root 843 6月 9 15:16 helloworld.mod.c
-rw-rw-r--. 1 root root 106K 6月 9 15:16 helloworld.mod.o
-rw-rw-r--. 1 root root 105K 6月 9 15:16 helloworld.o
-rw-rw-r--. 1 root root 243 6月 9 15:15 Makefile
-rw-rw-r--. 1 root root 32 6月 9 15:16 modules.order
-rw-rw-r--. 1 root root 0 6月 9 15:16 Module.symvers
[root@192 test]#
```

(2) 进行模块加载、查看、卸载



The screenshot shows an Xshell terminal window titled "openeuler - Xshell 4 (Free for Home/School)". The terminal session is performed as root on a system named 192. The user 'test' executes the following commands and receives the following output:

```
[root@192 test]# insmod helloworld.ko
[root@192 test]#
[root@192 test]# dmesg|tail -n 2
[ 2633.392991] hello init!
[ 2633.392991] hello,world!
[root@192 test]#
[root@192 test]# lsmod|grep helloworld
helloworld                16384  0
[root@192 test]#
[root@192 test]# rmmod helloworld
[root@192 test]#
[root@192 test]# dmesg|tail -n 2
[ 2633.392991] hello,world!
[ 2680.549636] hello exit
[root@192 test]#
[root@192 test]# lsmod|grep helloworld
[root@192 test]#
```

The terminal output demonstrates the successful loading of the `helloworld.ko` module, its presence in the loaded modules list, and its subsequent unloading, with corresponding kernel messages for initialization and exit.



# 实验二 内存管理

## 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。

### 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 `Documentation` 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`

## 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 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。

## 3 实验任务

### 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 `kmalloc.c` 源文件和 `Makefile` 文件。参考代码如下。

```

#include <linux/module.h>
#include <linux/slab.h>

MODULE_LICENSE("GPL");

unsigned char *kmallocmem1;
unsigned char *kmallocmem2;

static int __init mem_module_init(void)
{
    printk("Start kmallocl!\n");
    kmallocmem1 = (unsigned char*)kmallocl(1024, GFP_KERNEL);
    if (kmallocmem1 != NULL){
        printk(KERN_ALERT "kmallocmem1  addr  =  %lx\n", (unsigned
long)kmallocmem1);
    }else{
        printk("Failed to allocate kmallocmem1!\n");
    }
    kmallocmem2 = (unsigned char *)kmallocl(8192, GFP_KERNEL);
    if (kmallocmem2 != NULL){
        printk(KERN_ALERT "kmallocmem2  addr  =  %lx\n", (unsigned
long)kmallocmem2);
    }else{
        printk("Failed to allocate kmallocmem2!\n");
    }
    return 0;
}

static void __exit mem_module_exit(void)
{
    kfree(kmallocmem1);
    kfree(kmallocmem2);
    printk("Exit kmallocl!\n");
}

module_init(mem_module_init);
module_exit(mem_module_exit);

```

```

[root@openEuler ~]# cd tasks_k/2/task1
[root@openEuler task1]# ls
kmallocl.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/kmallocl.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /root/tasks_k/2/task1/kmallocl.mod.o
  LD [M]  /root/tasks_k/2/task1/kmallocl.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]#
```

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

正确编写满足功能的源文件，包括 vmalloc.c 源文件和 Makefile 文件。请同学们参照 3.1 自行编写完成，完成测试。

## 3.3 实验结果分析

查阅相关资料分析 kmalloc 和 vmalloc 分配的内存地址是否都位于内核空间？

# 实验三 进程管理

## 1 实验介绍

本实验通过在内核态创建进程，打印系统当前运行进程 PID，让学生们了解并掌握操作系统中的进程管理。

### 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，打印当前系统处于运行状态的进程的 PID 和名字。

## 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 实验任务

### 3.1 创建内核进程

#### 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。

### 3.1.2 实验步骤

步骤 5 正确编写满足功能的源文件 , 包括 kthread.c 源文件和 Makefile 文件。参考源代码如下 :

```
#include <linux/kthread.h>
#include <linux/module.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");

#define BUF_SIZE 20

static struct task_struct *myThread = NULL;

static int print(void *data)
{
    while(!kthread_should_stop()){
        printk("New kthread is running.");
        msleep(2000);
    }
    return 0;
}

static int __init kthread_init(void)
{
    printk("Create kernel thread!\n");
    myThread = kthread_run( ? );//请同学们自行补充代码。
    return 0;
}

static void __exit kthread_exit(void)
{
    printk("Kill new kthread.\n");
    if(myThread)
```



```
        kthread_stop(myThread);
    }

    module_init(kthread_init);
    module_exit(kthread_exit);
}
```

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

## 步骤 6 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 7 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 8 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.
```

## 3.2 打印输出当前处于运行状态的进程的 PID 和名字

### 3.2.1 相关知识

当前进程在 `/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。

### 3.2.2 实验步骤

**步骤 9** 正确编写满足功能的源文件，包括 `process_info.c` 源文件和 `Makefile` 文件。

参考源代码如下。

```
#include <linux/module.h>
#include <linux/sched/signal.h>
#include <linux/sched.h>

MODULE_LICENSE("GPL");

struct task_struct *p;

static int __init process_info_init(void)
{
    printk("Start process_info!\n");
    for_each_process(p){
```

```

        if( ? ) //请同学们自行补充代码。
            printk("1)name:%s 2)pid:%d 3)state:%ld\n", p->comm, p->pid,
p->state);
    }
    return 0;
}

static void __exit process_info_exit(void)
{
    printk("Exit process_info!\n");
}

module_init(process_info_init);
module_exit(process_info_exit);

```

```

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

```

#### 步骤 10 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 11 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 12 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!
```

# 实验四 中断和异常管理

## 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。

### 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进行驱动程序的开发。

## 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+\)。

## 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。

## 3 实验任务

### 3.1 使用 tasklet 实现打印 helloworld

#### 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*/
```

#### 3.1.2 实验步骤

步骤 13 正确编写满足功能的源文件，包括 tasklet\_intertupt.c 源文件和 Makefile 文件。参考源代码如下：

```
#include <linux/module.h>  
#include <linux/interrupt.h>  
MODULE_LICENSE("GPL");  
  
static struct tasklet_struct my_tasklet;
```



```

static void tasklet_handler(unsigned long data)
{
    printk("Hello World! tasklet is working...\n");
}

static int __init mytasklet_init(void)
{
    printk("Start tasklet module...\n");

    tasklet_init( ? );//请同学们自行补充代码。

    tasklet_schedule(&my_tasklet);

    return 0;
}

static void __exit mytasklet_exit(void)
{
    tasklet_kill(&my_tasklet);

    printk("Exit tasklet module...\n");
}

module_init(mytasklet_init);
module_exit(mytasklet_exit);

```

```

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

```

#### 步骤 14 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 15 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 16 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]#
```

## 3.2 用工作队列实现周期打印 helloworld

### 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，一旦其所在的处理单元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
```

## 3.2.2 实验步骤

步骤 17 正确编写满足功能的源文件，包括 workqueue\_test.c 源文件和 Makefile 文件。参考源代码如下：

```
#include <linux/module.h>
#include <linux/workqueue.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");

static struct workqueue_struct *queue = NULL;
static struct delayed_work mywork;
static int i = 0;

//work handle
void work_handle(struct work_struct *work)
{
    printk(KERN_ALERT "Hello World!\n");
}

static int __init timewq_init(void)
{
    printk(KERN_ALERT "Start workqueue_test module.");
    queue = create_singlethread_workqueue("workqueue_test");
    if(queue == NULL){
        printk(KERN_ALERT "Failed to create workqueue_test!\n");
    }
}
```

```

        return -1;
    }

    INIT_DELAYED_WORK( ? );//请同学们自行补充代码。

    for(i <= 3; i++){
        queue_delayed_work(queue, &mywork, 5 * HZ);
        ssleep(15);
    }
    return 0;
}

static void __exit timewq_exit(void)
{
    flush_workqueue(queue);
    destroy_workqueue(queue);
    printk(KERN_ALERT "Exit workqueue_test module.");
}

module_init(timewq_init);
module_exit(timewq_exit);

```

```

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

```

**步骤 18 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 19 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 20 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。

## 3.3 编写一个信号捕获程序，捕获终端按键信号

### 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+/)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

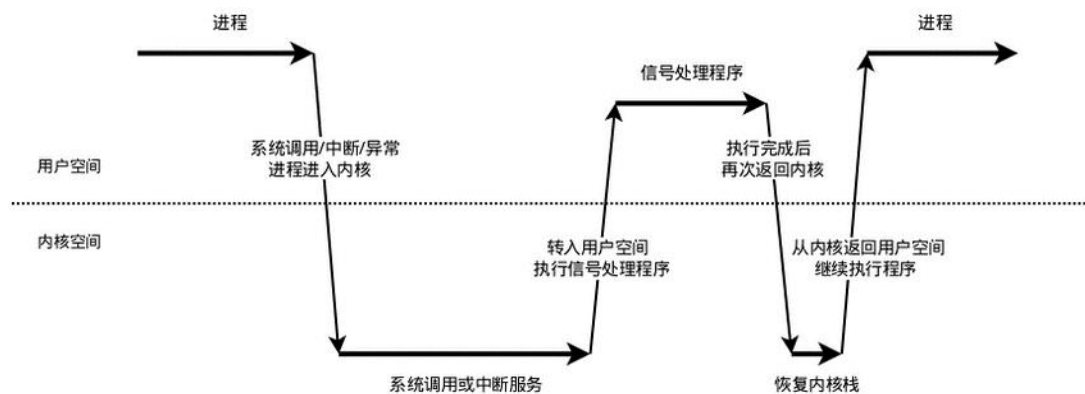


也可使用 `kill -l` 查看当前系统的信号编号列表，其中 1~31 为常规信号，34~64 为实时信号。

```
[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。

## 3.2.2 实验步骤

步骤 21 正确编写满足功能的源文件 catch\_signal.c。参考源代码如下：

```

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void signal_handler(int sig)
{
    switch(sig){
        case SIGINT:
            printf("\nGet a signal:SIGINT. You pressed ctrl+c.\n");
            break;
        case SIGQUIT:
            printf("\nGet a signal:SIGQUIT. You pressed ctrl+\\.\n");
            break;
        case ? : //请同学们自行补充代码。
            printf("\nGet a signal: ? ? ? . You pressed ctrl+z.\n");
            break;
    }
    exit(0);
}

int main()
{
    printf("Current process ID is %d\n", getpid());
    signal(SIGINT, signal_handler);
    signal(SIGQUIT, signal_handler);
    signal( ? );//请同学们自行补充代码。
    for(;;);
}

```

```

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

```

**步骤 22 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 23 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+\\.
```

# 实验五 设备管理

## 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,让学生们了解并掌握操作系统中的设备管理。

### 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。

## 1.2 任务描述

编写内核模块测试硬盘的读、写速率。

## 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。

## 3 实验任务

### 3.1 编写内核模块测试硬盘的写速率

步骤 24 正确编写满足功能的源文件 ,包括 write\_to\_disk.c 源文件和 Makefile 文件。

源代码如下 :

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

#define buf_size 1024
#define write_times 524288

MODULE_LICENSE("GPL");

struct timeval tv;

static int __init write_disk_init(void)
{
    struct file *fp_write;
    char buf[buf_size];
    int i;
    int write_start_time;
    int write_start_time_u;
    int write_end_time;
    int write_end_time_u;
    int write_time;
    loff_t pos;
    printk("Start write_to_disk module...\n");
    for(i = 0; i < buf_size; i++)
    {
        buf[i] = i + '0';
    }
    fp_write = filp_open("/home/tmp_file", O_RDWR | O_CREAT, 0644);
    if (IS_ERR(fp_write)) {
```

```

        printk("Failed to open file...\n");
        return -1;
    }
    pos = 0;
    do_gettimeofday(&tv);
    write_start_time = (int)tv.tv_sec;
    write_start_time_u = (int)tv.tv_usec;
    for(i = 0; i < write_times; i++) {
        kernel_write(fp_write, buf, buf_size, &pos);
    }
    do_gettimeofday(&tv);
    write_end_time = (int)tv.tv_sec;
    write_end_time_u = (int)tv.tv_usec;
    filp_close(fp_write, NULL);
    write_time = (write_end_time - write_start_time) * 1000000 +
(write_end_time_u - write_start_time_u);
    printk(KERN_ALERT "Writing to file costs %d us\n", write_time);
    printk("Writing speed is %d M/s\n", buf_size * write_times / write_time);
    return 0;
}

static void __exit write_disk_exit(void)
{
    printk("Exit write_to_disk module...\n");
}

module_init(write_disk_init);
module_exit(write_disk_exit);

```

```

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

```

**步骤 25** 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 26 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 27 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...
```

## 3.2 编写内核模块测试硬盘的读速率

正确编写满足功能的源文件，包括 `read_from_disk.c` 源文件和 `Makefile` 文件。请同学们参照 3.1 自行完成，并进行测试。

# 实验六 文件系统

## 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。

### 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。

## 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。

## 3 实验任务

### 3.1 为 Ext4 文件系统添加扩展属性

#### 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，为了能使用扩展用户属性，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。

### 3.1.2 实验步骤

步骤 28 环境准备：为了使用扩展属性，需要安装 libattr：

```
dnf install -y libattr
```

步骤 29 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   0    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 30 检查当前文件系统是否支持文件扩展属性

在 ext3 和 ext4 文件系统中，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 31 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 32 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 33** 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 34** 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 35** 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。

## 3.2 注册一个自定义的文件系统类型

### 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。

### 3.2.2 实验步骤

步骤 36 查看系统中已经注册的文件系统类型

```
cat /proc/filesystems
```

步骤 37 正确编写满足功能的源文件,包括 `register_newfs.c` 源文件和 `Makefile` 文件。

源代码如下：

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

MODULE_LICENSE("GPL");

static struct file_system_type myfs_type = {
    .name  = "myfs",
    .owner = THIS_MODULE,
};

MODULE_ALIAS_FS("myfs");

static int __init register_newfs_init(void)
{
    printk("Start register_newfs module...");
}
```



```

        return register_filesystem( ? );//请同学们自行补充代码。
    }

static void __exit register_newfs_exit(void)
{
    printk("Exit register_newfs module...");
    unregister_filesystem(&myfs_type);
}

module_init(register_newfs_init);
module_exit(register_newfs_exit);

```

```

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

```

步骤 38 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 39 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 40 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”。

## 3.3 在/proc 下创建目录

### 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）。

## 3.3.2 实验步骤

步骤 41 正确编写满足功能的源文件，包括 proc\_mkdir.c 源文件和 Makefile 文件。

源代码如下：

```
#include <linux/module.h>
#include <linux/proc_fs.h>

MODULE_LICENSE("GPL");

static struct proc_dir_entry *myproc_dir;

static int __init myproc_init(void)
{
    int ret = 0;
    printk("Start proc_mkdir module...");
    myproc_dir = proc_mkdir("myproc",NULL);
    if(myproc_dir == NULL)
        return -ENOMEM;
    return ret;
}

static void __exit myproc_exit(void)
{

```

```

        printk("Exit proc_mkdir module...");

        proc_remove(?);//请同学们自行补充代码。
    }

module_init(myproc_init);
module_exit(myproc_exit);

```

```

[root@openEuler ~]# cd tasks_k/7/task3
[root@openEuler task3]# ls
Makefile  proc_mkdir.c

```

#### 步骤 42 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 43 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 44 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。

# 实验七 网络管理

## 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，让学生们了解并掌握操作系统中的网络管理。

### 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。

## 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。

## 3 实验任务

### 3.1 编写基于 socket 的 UDP 发送接收程序

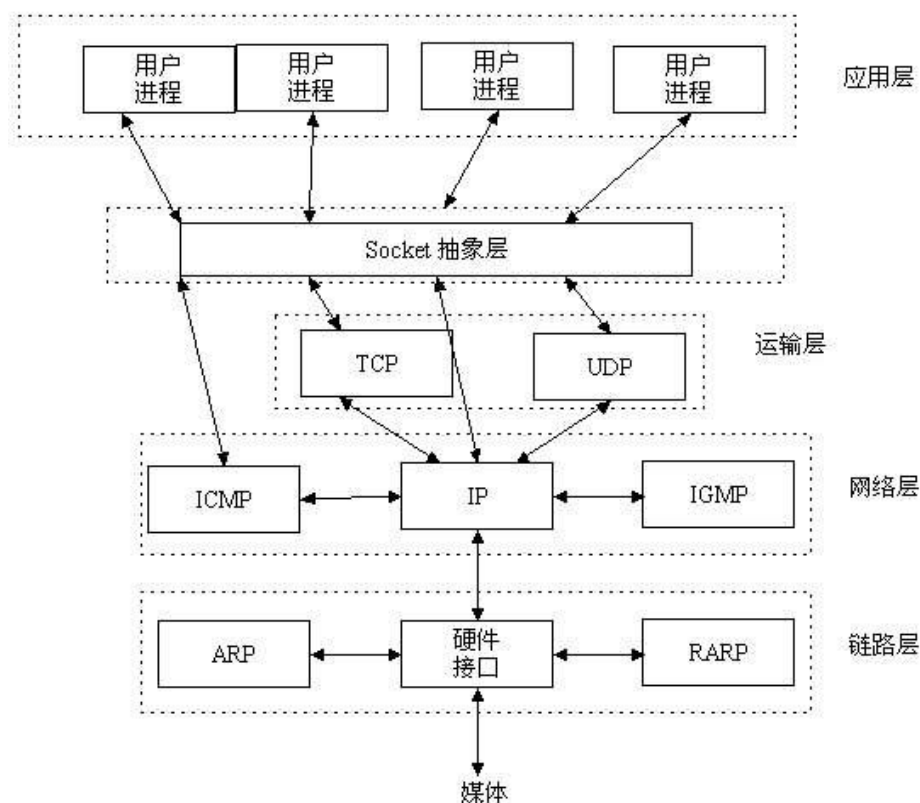
#### 3.1.1 相关知识

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。

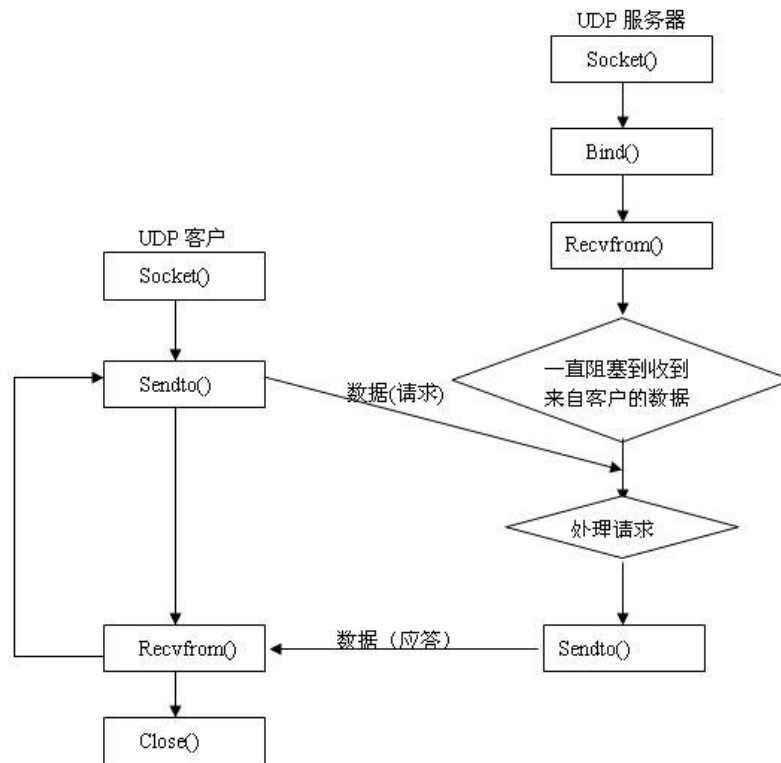


## 二、UDP

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路上传输。

( 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。

### 3.1.2 实验步骤

步骤 45 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。服务端源代码如下 ( server.c ) :

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
```

```

#define      PORT      40000
#define      BUF_SIZE  1024

int main(void)
{
    int sock_fd;
    int len;
    char buffer[BUF_SIZE];
    struct sockaddr_in server_addr, client_addr;
    if(-1 == (sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) )
    {
        printf("Failed to create a socket!\n");
        return 0;
    }
    //server information
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if(-1 == bind(sock_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)))
    {
        printf("Failed to bind the socket!\n");
        return 0;
    }
    len = sizeof(client_addr);

    //rec and print
    while(1)
    {
        bzero(buffer, BUF_SIZE);
        if(-1 != (recvfrom(sock_fd, buffer, BUF_SIZE, 0, (struct
sockaddr*)&client_addr, &len)) )
        {
            printf("The message received is: %s", buffer);

```

```
    }  
    }  
    return 0;  
}
```

客户端源代码如下 ( client.c ) :

```
#include <stdio.h>  
#include <string.h>  
#include <sys/socket.h>  
#include <arpa/inet.h>  
#include <unistd.h>  
  
#define PORT    40000  
#define BUF_SIZE    1024  
  
int main(void)  
{  
    int sock_fd;  
    char buffer[BUF_SIZE];  
    int size;  
    int len;  
    int ret;  
    struct sockaddr_in server_addr;  
    if(-1 == (sock_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP))){  
        printf("Failed to create a socket!\n");  
        return 0;  
    }  
  
    //server infomation  
    memset(&server_addr, 0, sizeof(server_addr));  
    server_addr.sin_family = AF_INET;  
    server_addr.sin_port = htons(PORT);  
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  
    bzero(buffer, BUF_SIZE);
```

```

len = sizeof(server_addr);

//read from stdin and send to server
while(1){
    printf("Please enter the content to be sent:\n");
    size = read(0, buffer, BUF_SIZE);
    if(size){
        sendto(sock_fd, buffer, size, 0, (struct sockaddr*)&server_addr, len);
        bzero(buffer, BUF_SIZE);
    }
}
close(sock_fd);
return 0;
}

```

```

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

```

步骤 46 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 47 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!

### 3.1.2 功能扩展

扩展实现一个点对点的聊天程序。

# 第三部分 综合实验大作业

## 1.作业导读

虚拟化 ( virtualization ) 是操作系统永恒的主题，本作业旨在让学生利用 openEuler 轻量级虚拟化容器引擎 iSulad 创建容器并在容器中运行一个 Web 服务器。

完成本作业需要三个阶段：(1) 在 openEuler 上安装、配置 iSulad 并用它来下载镜像、启动容器；(2) 准备 Web Server 源代码；(3) 在容器中编译并运行这个 Web Server。

## 2 功能要求

使一个 Web Server 在容器中编译并运行，可以通过浏览器访问这个网站。

### 2.1 作业要求

1. **技术要求** :openEuler 操作系统基本操作、iSulad 容器的构建、配置和操作、对 TCP/IP 协议栈的理解。

2. **特别注意事项**：(1) 该 Web Server 须运行于 iSulad 容器中；(2) 该 Web Server 需提供静态网页和动态网页访问功能；(3) 实现该 Web Server 的程序设计语言不限，例如，可以使用 C/C++、Rust、Go、Python 等任何一种语言来构建，作为学习，亦可参照或使用即成案例。

### 2.2 作业任务

任务 1：安装、配置 iSulad、下载镜像并运行容器

该任务包括 3 个方面的工作：

- 1.在 openEuler 操作系统上安装、配置 iSulad 容器引擎；
- 2.下载一个操作系统系统镜像，配置并运行其容器。

输出：操作步骤。

任务 2：准备 Web Server 源代码

本任务包括以下内容：

1. 编写/准备 Web Server 源代码；
2. 写出该 Web Server 的流程图。

输出：Web Server 源代码和流程图。

说明：可以使用 Web Server 的既有案例。

任务 3：在容器中编译并运行 Web Server

本任务包括以下内容：

1. 在上述步骤中创建的容器中编译该 Web Server 的源代码（包含一个简单的网站）；
2. 在容器中运行该 Web Server；
3. 通过浏览器访问该网站的静态页面和动态页面。

输出：操作步骤和成功访问静态页面和动态页面的截图。

## 3. 实验步骤

### 3.1 安装 isulad

```
[root@openeuler ~]# yum install -y iSulad
```

### 3.2 启动并查看版本

步骤 1 用 systemctl start 命令启动

```
[root@openeuler ~]# systemctl start isulad
```

步骤 2 查看状态

```
[root@openeuler ~]# systemctl status isulad
```

按‘q’或‘Q’键退出信息显示。

步骤 3 查看版本

```
[root@openeuler ~]# isula version
```

步骤 4 查看帮助

```
[root@openeuler ~]# isula --help
```

### 3.3 安装 JSON 格式数据处理工具

```
[root@openeuler ~]# yum install -y jq
```

### 3.4 修改 isulad 的配置文件

先备份，后修改。

```
[root@openeuler iSula]# cp /etc/isulad/daemon.json /etc/isulad/daemon.json.origin
[root@openeuler iSula]# vi /etc/isulad/daemon.json    #在此处修改
```

在上述文件中，把"registry-mirrors"的值设为"hub.oepkgs.net"。（hub.oepkgs.net 为 openEuler 社区与中科院软件所共建的、开源免费的容器镜像仓库）。

### 3.5 检查配置文件的合法性和重启 isulad 使配置生效

```
[root@openeuler iSula]# cat /etc/isulad/daemon.json | jq
如果配置文件的内容能正确显示出来，即表示其格式合法。
```

重启 isulad:

```
[root@openeuler iSula]# systemctl restart isulad
```

### 3.6 运行容器 busybox

```
[root@openeuler iSula]# isula run busybox echo "hello world"
Unable to find image 'busybox' locally
Image "busybox" pulling
Image "219ee5171f8006d1462fa76c12b9b01ab672dbc8b283f186841bf2c3ca8e3c93" pulled
hello world
```

由于这是第一次运行，所以会拉取 busybox 的镜像，然后会运行它的一个实例，并且调用 echo 命令打印 hello world 字符串。（按：由于 oepkgs 目前只有 x86 的 busybox 镜像，在鲲鹏平台上运行这一步会提示错误，请忽略这一步。）

查看其镜像：

```
[root@openeuler iSula]# isula images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
busybox	latest	219ee5171f80	2020-12-04 06:19:53	1.385 MB

以上输出表明这一阶段的安装成功了，下面继续验证其他的一些命令。

**附：isulad 常用命令**

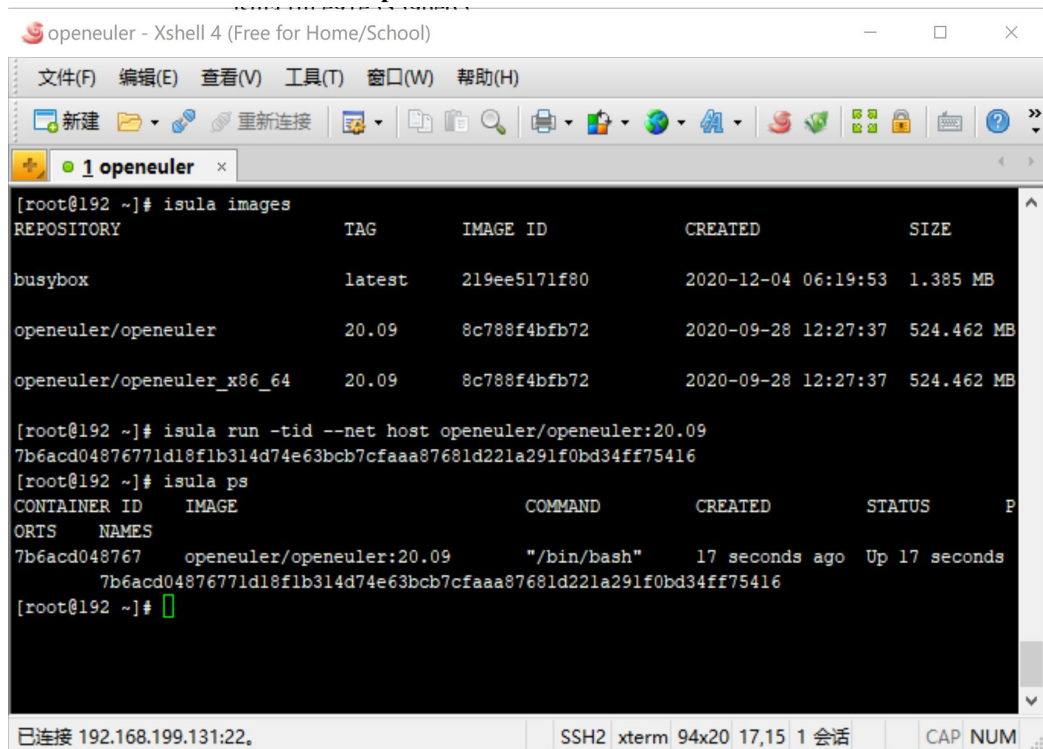
1. 查看镜像：  
isula images
2. 创建容器：  
例:isula create -it openeuler/openeuler:20.09
3. 查看容器  
isula ps 运行状态的容器；



- isula ps -a 查看所有创建的容器，包括未运行的。
- 3.启动容器 isula start e91e5359be65，其中 e91e5359be65 为容器 ID。
- 4.直接运行容器
- ```
isula run openeuler/openeuler:20.09
```
- 5.交互式运行容器
- ```
isula run -it openeuler/openeuler:20.09
```
- 6.暂停/恢复一个容器
- ```
isula pause e91e5359be65
isula unpause e91e5359be65
```
- 7.先停止，再删除一个容器
- ```
isula stop e91e5359be65
isula rm e91e5359be65
```
- 8.强制删除运行中的容器
- ```
isula rm -f 6c1d81467d33
```
- 删除所有容器：
- ```
isula rm -f $(isula ps -a)
```
- 9.先将关联到镜像的容器销毁
- ```
isula rm -f bb85ce20525d 3139ce089c56
```
- 10.然后删除镜像
- ```
isula rmi openeuler/openeuler:20.09
```

## 3.7 下载镜像并运行容器

### 3.7.1 使用宿主机网络创建并运行 openeuler 容器



```
openeuler - Xshell 4 (Free for Home/School)
文件(F) 编辑(E) 查看(V) 工具(T) 窗口(W) 帮助(H)
新建 重新连接
1 openeuler x
[root@192 ~]# isula images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
busybox              latest       219ee5171f80     2020-12-04 06:19:53  1.385 MB
openeuler/openeuler 20.09       8c788f4bfb72     2020-09-28 12:27:37  524.462 MB
openeuler/openeuler_x86_64 20.09       8c788f4bfb72     2020-09-28 12:27:37  524.462 MB

[root@192 ~]# isula run -tid --net host openeuler/openeuler:20.09
7b6acd04876771d18f1b314d74e63bcb7cfaaa87681d221a291f0bd34ff75416
[root@192 ~]# isula ps
CONTAINER ID   IMAGE                                COMMAND              CREATED          STATUS          P
ORTS          NAMES
7b6acd048767  openeuler/openeuler:20.09          "/bin/bash"         17 seconds ago  Up 17 seconds
7b6acd04876771d18f1b314d74e63bcb7cfaaa87681d221a291f0bd34ff75416
[root@192 ~]#
```

已连接 192.168.199.131:22. SSH2 xterm 94x20 17,15 1 会话 CAP NUM

选项“--net host”表示在容器里使用宿主机网络。

### 3.7.2 交互式运行这个容器



The screenshot shows an Xshell 4 terminal window titled 'openeuler - Xshell 4 (Free for Home/School)'. The terminal output is as follows:

```
[root@192 ~]# clear

[root@192 ~]# isula ps
CONTAINER ID      IMAGE                                     COMMAND                  CREATED           STATUS           P
ORTS      NAMES
7b6acd048767      openeuler/openeuler:20.09              "/bin/bash"            4 minutes ago    Up 4 minutes
7b6acd04876771d18f1b314d74e63bcb7cf8aa87681d221a291f0bd34ff75416
[root@192 ~]# isula exec -it 7b6acd048767

Welcome to 4.19.90-2003.4.0.0036.oel.x86_64

System information as of time:  Wed 09 Jun 2021 07:16:19 PM CST

System load:      0.04
Processes:        6
Memory used:      18.3%
Swap used:        0.0%
Usage On:         26%
IP address:       192.168.199.131
```

The status bar at the bottom indicates a connection to 192.168.199.131:22 via SSH2, with a terminal size of 94x20 and 20,29 columns.

### 3.7.3 安装 make ,gcc

```
yum install -y make
yum install -y gcc
yum install -y git
```

## 3.8 下载 Web Server 源代码

Web Server 源代码可以采用《深入理解计算机系统》(作者 Randal E. Bryant; David R. O'Hallaron)一书中的示例代码，当前下载地址是：

[git clone https://github.com/marcustedesco/webserver.git](https://github.com/marcustedesco/webserver.git)

从最初发布以来，该源代码只做了少许更新，即便如此，为了简单地展示显示网页的效果，本作业对其中的 `cgi-bin/adder.c` 略做了改动，准确地说是将其还原到了最初的版本。

可在容器内下载，也可在宿主机下载

在宿主机下载后要拷贝到容器中，命令为：

例： `isula cp /home/tinyServer.tar 70fe2e12b713:/home/`

上述命令表示把文件 `tinyServer.tar` 从宿主机的 `/home` 目录拷贝到了容器里的 `/home` 目录。

## 3.9 在容器中编译并运行 Web Server

- 1.进入到 `webserver` 所在目录。
- 2.编译 `make all`
- 3.运行 `webserver`

```
[root@openeuler webserver]# ./runServer.sh
MAKING
(cd cgi-bin; make)
make[1]: Entering directory '/home/tiny/webserver/cgi-bin'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/tiny/webserver/cgi-bin'
RUNNING SERVER
Port: 8084
LISTENFD: 3
.....
```

灰色字体表示命令在容器中执行。可以看到，一个 tiny/webserver 的目录被解压出来，对源代码进行编译后我们通过一个脚本运行这个 Web Server，她在容器 IP 地址(实际就是宿主机 IP 地址)的 TCP 8084 端口监听(所以宿主机的安全策略应开放这个端口)。

开放某个端口 ( 如 8080 ) 命令：

```
[root@192 ~]# iptables -I INPUT -p tcp --dport 8080 -j ACCEPT
```

注：上述命令在宿主机中执行，不在容器中。

## 3.10 在浏览器中访问网页

在浏览器的地址栏里分别输入如下两个 URL 访问这个 Web Server 并观察效果：

静态网页：<http://119.8.238.181:8084/>

动态网页：<http://119.8.238.181:8084/cgi-bin/adder?1&2>

注意要把上述 URL 中的 IP 地址改成当前宿主机的 IP 地址，端口改成 webserver 的配置端口(默认如 13051)。

( 完 )

后续：本实验指导书是由华为 openEuler 操作系统实验指导书改编而成，改编时间仓促，水平有限，难免有错误的地方，请老师和同学们指正。