

浙江大学

本科实验报告

课程名称：操作系统

姓 名：应承峻

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：软件工程

学 号：3170103456

指导教师：夏莹杰

2019 年 10 月 5 日

浙江大学实验报告

课程名称： 操作系统 实验类型： 综合

实验项目名称： Linux 内核重建

学生姓名： 应承峻 专业： 软件工程 学号： 3170103456

电子邮件地址： 3170103456@zju.edu.cn 手机： 17326084929

实验地点： 玉泉曹光彪西 503 实验日期： 2019 年 10 月 5 日

一、实验目的和要求

学习重新编译 Linux 内核，理解、掌握 Linux 内核和发行版本的区别。

二、实验内容

在 Linux 操作系统环境下重新编译内核。实验主要内容：

- 查找并且下载一份内核源代码
- 配置内核
- 编译内核和模块
- 配置启动文件

三、主要仪器设备

笔记本电脑 1 台，相关配置如下：

处理器 英特尔 Core i7-8750H @ 2.20GHz 六核
内 存 16 GB （三星 DDR4 2667MHz）
主硬盘 PeM280240GP4C15B （240 GB/固态硬盘）
显 卡 Nvidia GeForce GTX 1060 （6 GB）
操作系统环境： Windows 10 64 位（DirectX 12）
Linux 版本： ubuntu-19.04

四、操作方法和实验步骤

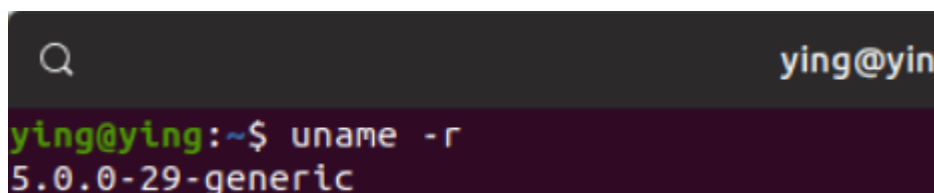
Linux 是当今流行的操作系统之一。由于其源码的开放性，现代操作系统设计的思想和技术能够不断运用于它的新版本中。因此，读懂并修改 Linux 内核源代码无疑是学习操作系统设计技术的有效方法。

1. 查找并且下载一份内核源代码

Linux 受 GNU 通用公共许可证（GPL）保护，其内核源代码是完全开放的。现在很多 Linux 的网站都提供内核代码的下载。推荐你使用 Linux 的官方网站：<http://www.kernel.org>，考虑到网络下载速度，你也可以从<http://mirrors.aliyun.com/linux-kernel/>就近下载。

保险起见，首先确认你的 Linux 运行环境究竟采用哪个版本的内核源代码，第一步首先想办法获取合适版本的 Linux 内核代码。通过命令

`uname -r` 观察到我使用的 linux 运行环境时采用 5.0.0-29-generic 的源代码



```
ying@ying:~$ uname -r
5.0.0-29-generic
```

考虑编译更高版本的内核，我们从网站上下载如下文件到工作目录中：

[linux-5.3.tar.xz](#)

[patch-5.3.xz](#)



2. 部署内核源代码

此过程比较机械、枯燥，因而容易出错。请严格按下述步骤来操作。

- 首先，需要执行 `apt-get update` 来更新软件源。
- 然后，安装需要的环境：(耗时间，下载、安装 `kernel-package` 用时 5 小时)

```
#apt-get install kernel-package libncurses5-dev libssl-dev
```

```
ying@ying: ~  
正在处理用于 fontconfig (2.13.1-2ubuntu2) 的触发器 ...  
正在处理用于 desktop-file-utils (0.23-4ubuntu1) 的触发器 ...  
正在处理用于 mime-support (3.60ubuntu1) 的触发器 ...  
正在处理用于 gnome-menus (3.32.0-1ubuntu1) 的触发器 ...  
正在处理用于 libc-bin (2.29-0ubuntu2) 的触发器 ...  
正在处理用于 man-db (2.8.5-2) 的触发器 ...  
正在处理用于 sgml-base (1.29) 的触发器 ...  
正在设置 docbook-xsl (1.79.1+dfsg-2) ...  
正在设置 sgml-data (2.0.11) ...  
正在处理用于 sgml-base (1.29) 的触发器 ...  
正在设置 docbook-xml (4.5-8) ...  
正在处理用于 sgml-base (1.29) 的触发器 ...  
正在设置 xmlto (0.0.28-2.1) ...  
正在设置 kernel-package (13.018+nmu1) ...  
正在设置 docbook-dsssl (1.79-9.1) ...  
正在设置 dblatex (0.3.10-2) ...  
正在处理用于 sgml-base (1.29) 的触发器 ...  
正在设置 docbook-utils (0.6.14-3.3) ...  
正在处理用于 tex-common (6.11) 的触发器 ...  
Running updmap-sys. This may take some time... done.  
Running mktexlsr /var/lib/texmf ... done.  
Building format(s) --all.  
This may take some time... done.
```

- 在工作目录下解开压缩内核：

```
#xz -d linux-5.3.tar.xz
```

解压后得到 `linux-4.6.tar`。执行：

```
#tar xvf linux-5.3.tar
```

解压得到目录 `linux-5.3`

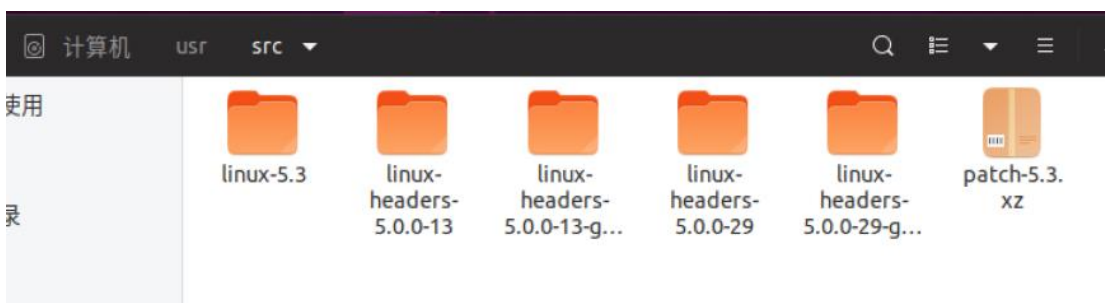


- d) 把内核目录 `linux-5.3` 和补丁 `patch-5.3.xz` 都复制到 `/usr/src`，然后进入 `/usr/src`

```
#cp linux-5.3 /usr/src -rf
```

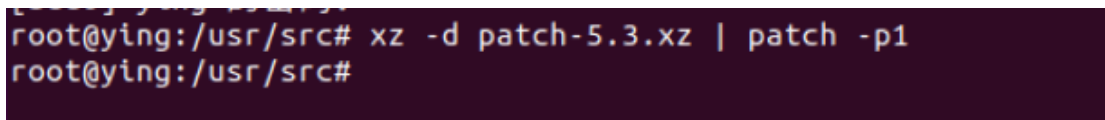
```
#cp patch-5.3.xz /usr/src
```

```
#cd /usr/src
```



- e) 打内核补丁

```
#xz -d patch-5.3.xz | patch -p1
```



执行后没有任何提示说明执行正确。

可见，Linux 内核源代码已经在 `/usr/src/linux-5.3` 下面了。作者习惯了通过路径 `/usr/src/linux` 去访问它。这只要建一个符号链接：

```
# ln -s /usr/src/linux-5.3/ linux
```



3. 配置内核

在你进行这项工作之前，不妨先看一看 `/usr/src/linux` 目录下内核源代码自带

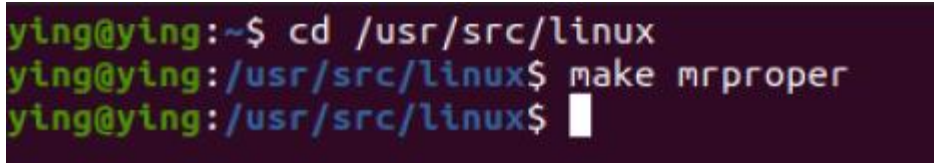
的 README 文件。在这份文件中，对怎样进行内核的解压，配置，安装都进行了详细的讲解。不过，其介绍的步骤不完全符合我们的版本，所以还是以本书为准。

在编译内核前，一般来说都需要对内核进行相应的配置。配置是精确控制新内核功能的机会。配置过程也控制哪些需编译到内核的二进制映像中(在启动时被载入)，哪些是需要时才装入的内核模块（module）。

```
# cd /usr/src/linux
```

第一次编译的话，有必要将内核源代码树置于一种完整和一致的状态。因此，我们推荐执行命令 `make mrproper`。它将清除目录下所有配置文件和先前生成核心时产生的.o 文件：

```
#make mrproper
```



```
ying@ying:~$ cd /usr/src/linux
ying@ying:/usr/src/linux$ make mrproper
ying@ying:/usr/src/linux$
```

用 `uname -r` 命令查看当前的内核版本号，得知为 5.0.0-29-generic。把 /usr/src/linux-headers-5.0.0-29-generic 里面的.config 文件复制到 linux-4.6 文件夹中：

```
# cp ../linux-headers-5.0.0-29-generic/.config .
```

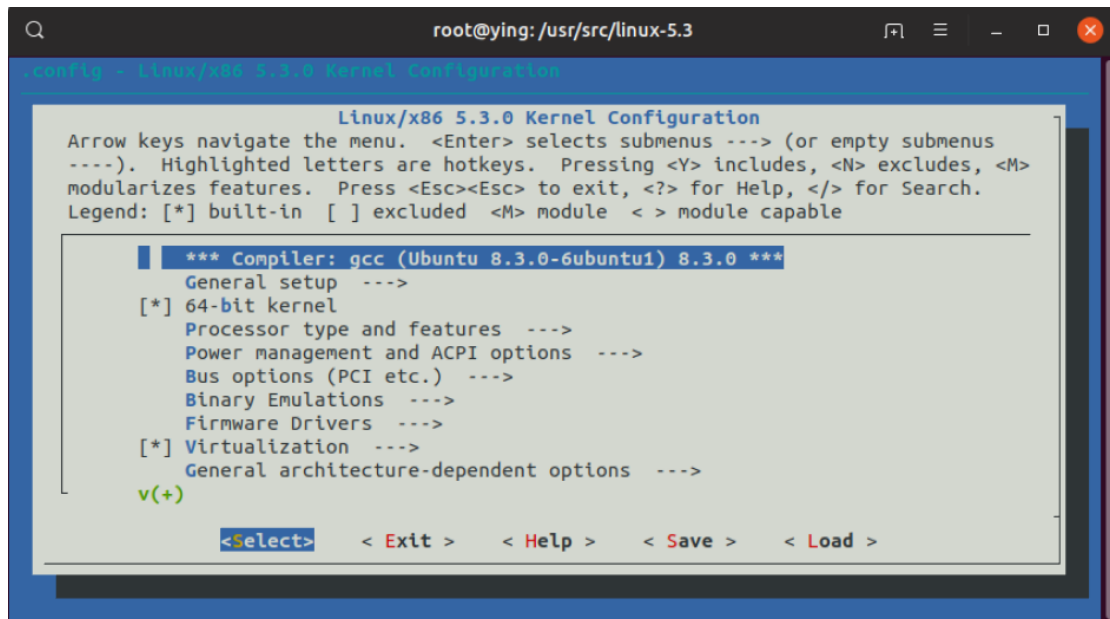


```
root@ying: /usr/src/linux-headers-5.0.0-29-generic
ying@ying:/usr/src/linux-headers-5.0.0-29-generic$ ls -al | grep .config
-rw-r--r-- 1 root root 224293 9月 12 17:19 .config
-rw-r--r-- 1 root root 224417 9月 12 17:19 .config.old
lrwxrwxrwx 1 root root 33 9月 12 17:19 Kconfig -> ../linux-headers-5.0.0-29/Kconfig
ying@ying:/usr/src/linux-headers-5.0.0-29-generic$ cp .config ../linux-5.3
cp: 无法创建普通文件 '../linux-5.3/.config': 权限不够
ying@ying:/usr/src/linux-headers-5.0.0-29-generic$ sudo su
[sudo] ying 的密码:
root@ying:/usr/src/linux-headers-5.0.0-29-generic# cp .config ../linux-5.3
root@ying:/usr/src/linux-headers-5.0.0-29-generic#
```

然后，执行 `make menuconfig`，依次选择 load→OK→Save→OK→EXIT→EXIT，得到更新的.config 文件。

如果需要做一些个性化配置，可以再次进入：

```
# make menuconfig
```



进行配置时，大部分选项可以使用其缺省值，只有小部分需要根据用户不同的需要选择。例如，如果硬盘分区采用 ext2 文件系统（或 ext3 文件系统），则配置项应支持 ext2 文件系统（ext3 文件系统）。又例如，系统如果配有 SCSI 总线及设备，需要在配置中选择 SCSI 卡的支持。又例如，后续实验需要在文件上格式化一个文件系统并且安装到某目录，这需要 device drivers 之下的 block device 之下的 loopback device support 特征。

对每一个配置选项，用户有三种选择，它们分别代表的含义如下：

- “<*>”或“[*]” — 将该功能编译进内核
- “[]” — 不将该功能编译进内核
- “[M]” — 将该功能编译成可以在需要时动态插入到内核中的模块

将与核心其它部分关系较远且不经常使用的部分功能代码编译成为可加载模块，有利于减小内核的长度，减小内核消耗的内存，简化该功能相应的环境改变时对内核的影响。许多功能都可以这样处理，例如像上面提到的对 SCSI 卡的支持，等等。

4. 编译内核和模块

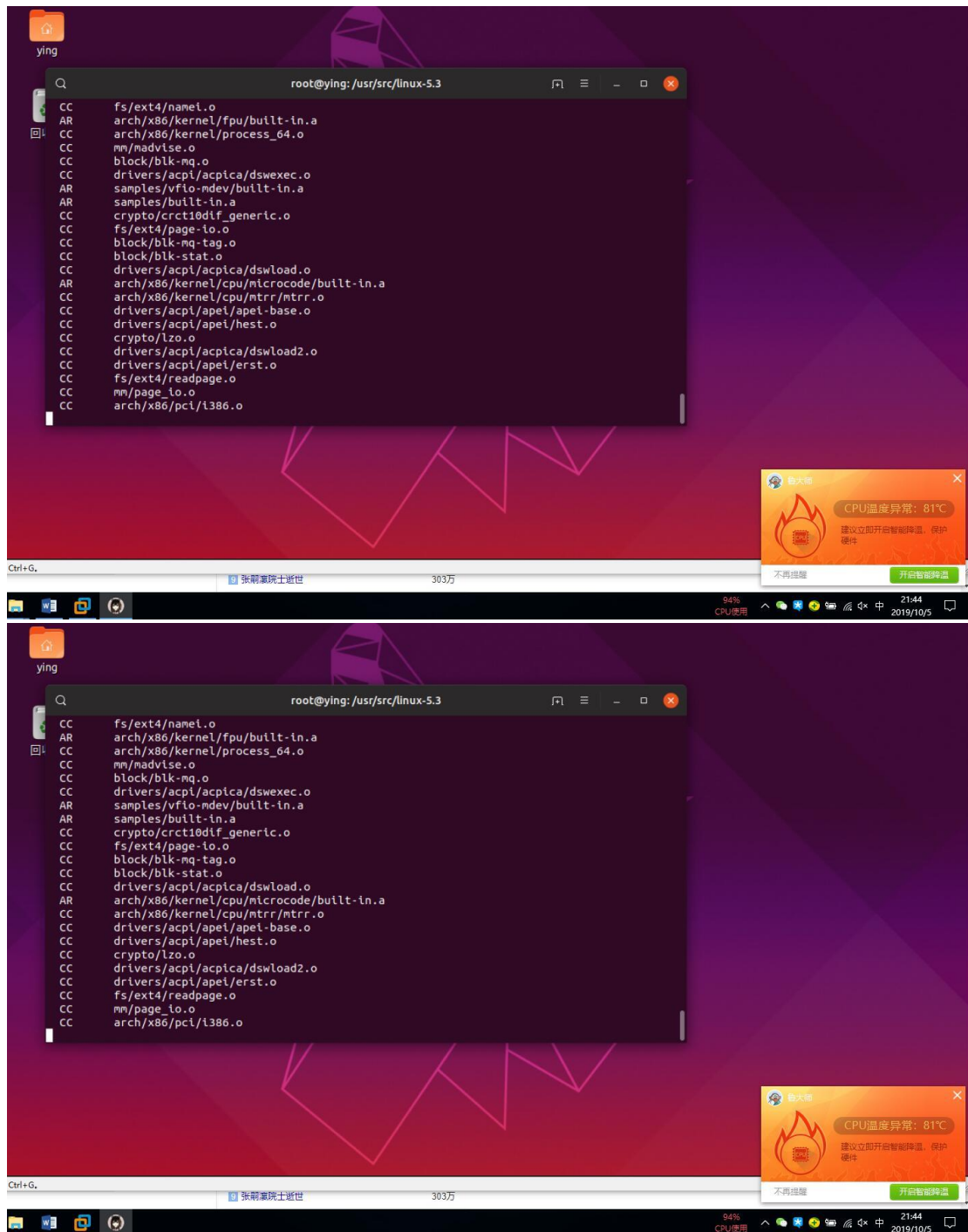
编译内核，就用：

```
#make bzImage -jN
```

这里，N 是主机 CPU 的核数 * 2。比如主机是单核 CPU，那么，应该执行

```
#make bzImage -j2
```

6 核采用 make bzImage -j12



耗时 3 分钟

执行到这一步，可能会出现错误：

fatal error: openssl/openssl.h: No such file or directory

这是因为没有安装 openssl 的，需要先安装 openssl：

#apt-get install libssl-dev


```
root@ying: /usr/src/linux-5.3
root@ying:/usr/src/linux-5.3# apt-get install libssl-dev
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会同时安装下列软件：
  libssl1.1
建议安装：
  libssl-doc
下列【新】软件包将被安装：
  libssl-dev
下列软件包将被升级：
  libssl1.1
升级了 1 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 138 个软件包未被升级。
需要下载 2,876 kB 的归档。
解压缩后会消耗 7,876 kB 的额外空间。
您希望继续执行吗？ [Y/n] y
获取:1 http://cn.archive.ubuntu.com/ubuntu disco-updates/main amd64 libssl1.1 amd64 1.1.1b-1u
buntu2.4 [1,305 kB]
4% [1 libssl1.1 132 kB/1,305 kB 10%]
```

安装对应依赖后再运行。

编译内核需要较长的时间，具体与机器的硬件条件及内核的配置等因素有关。完成后产生的内核文件 bzImage 的位置在/usr/src/linux/arch/x86/boot 目录下，当然这里假设用户的 CPU 是 Intel x86 型的，并且你将内核源代码放在/usr/src/linux 目录下。

如果选择了可加载模块，编译完内核后，要对选择的模块进行编译。用下面的命令编译模块并安装到标准的模块目录中（N 是主机 CPU 的核数 * 2）：

#make modules -jN

```
root@ying: /usr/src/linux-5.3
LD [M] sound/soc/xtensa/snd-soc-xtfpga-i2s.ko
LD [M] sound/soc/zte/zx-tdm.ko
LD [M] sound/soundcore.ko
LD [M] sound/synth/emux/snd-emux-synth.ko
LD [M] sound/synth/snd-util-mem.ko
LD [M] sound/usb/6fire/snd-usb-6fire.ko
LD [M] sound/usb/bcd2000/snd-bcd2000.ko
LD [M] sound/usb/caiaq/snd-usb-caiaq.ko
LD [M] sound/usb/hiface/snd-usb-hiface.ko
LD [M] sound/usb/line6/snd-usb-line6.ko
LD [M] sound/usb/line6/snd-usb-pod.ko
LD [M] sound/usb/line6/snd-usb-podhd.ko
LD [M] sound/usb/line6/snd-usb-toneport.ko
LD [M] sound/usb/line6/snd-usb-variix.ko
LD [M] sound/usb/misc/snd-ua101.ko
LD [M] sound/usb/snd-usb-audio.ko
LD [M] sound/usb/snd-usbmidi-lib.ko
LD [M] sound/usb/usx2y/snd-usb-us122l.ko
LD [M] sound/usb/usx2y/snd-usb-usx2y.ko
LD [M] sound/x86/snd-hdmi-lpe-audio.ko
LD [M] sound/xen/snd_xen_front.ko
LD [M] virt/lib/irqbypass.ko
root@ying:/usr/src/linux-5.3#
root@ying:/usr/src/linux-5.3#
```

同样，编译内核模块也需要较长的时间，具体与机器的硬件条件及内核的配置等因素有关（作者采用 VMWare 虚拟机，需要约 110 分钟）。

由于 6 核速度较快，实际花费时间 16 分钟

5. 安装内核

安装内核的过程就快得很多了。先安装模块：

```
#make modules_install
```

```
root@ying: /usr/src/linux-5.3
INSTALL sound/usb/line6/snd-usb-toneport.ko
INSTALL sound/usb/line6/snd-usb-variax.ko
INSTALL sound/usb/misc/snd-ua101.ko
INSTALL sound/usb/snd-usb-audio.ko
INSTALL sound/usb/snd-usbmidi-lib.ko
INSTALL sound/usb/usx2y/snd-usb-us122l.ko
INSTALL sound/usb/usx2y/snd-usb-usx2y.ko
INSTALL sound/x86/snd-hdmi-lpe-audio.ko
INSTALL sound/xen/snd_xen_front.ko
INSTALL virt/lib/irqbypass.ko
DEPMOD 5.3.0
```

再安装内核：

```
#make install
```

```
root@ying:/usr/src/linux-5.3# make install
sh ./arch/x86/boot/install.sh 5.3.0 arch/x86/boot/bzImage \
    System.map "/boot"
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 5.3.0 /boot/vmlinuz-5.3.0
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 5.3.0 /boot/vmlinuz-5.3.0
update-initramfs: Generating /boot/initrd.img-5.3.0
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 5.3.0 /boot/vmlinuz-5.3.0
run-parts: executing /etc/kernel/postinst.d/update-notifier 5.3.0 /boot/vmlinuz-5.3.0
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 5.3.0 /boot/vmlinuz-5.3.0
Sourcing file `/etc/default/grub'
Sourcing file `/etc/default/grub.d/init-select.cfg'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-5.3.0
Found initrd image: /boot/initrd.img-5.3.0
Found linux image: /boot/vmlinuz-5.0.0-29-generic
Found initrd image: /boot/initrd.img-5.0.0-29-generic
Found linux image: /boot/vmlinuz-5.0.0-13-generic
Found initrd image: /boot/initrd.img-5.0.0-13-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

6. 应用 grub 配置启动文件

grub 就是管理 ubuntu 系统启动的一个程序。想运行刚刚编译好的内核，就要修改对应的 grub。

```
#mkinitramfs 5.3.0 -o /boot/initrd.img-5.3.0
```

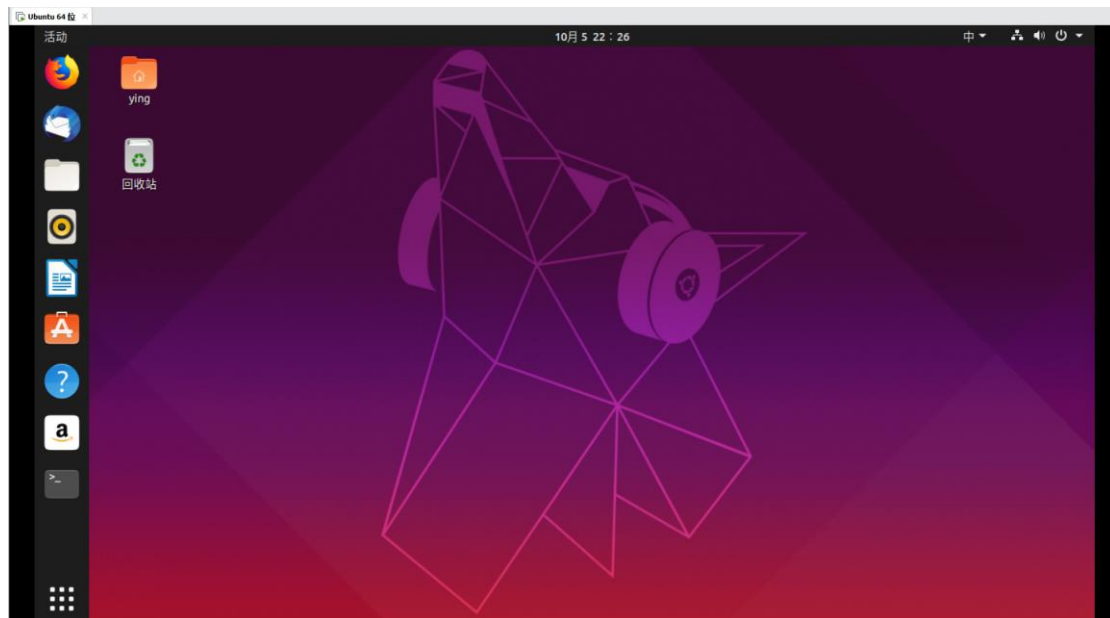
```
#update-grub2
```

```
root@ying:/usr/src/linux-5.3# mkinitramfs 5.3.0 -o /boot/initrd.img-5.3.0
root@ying:/usr/src/linux-5.3#
```

update-grub2 命令会帮我们自动修改 grub。

```
root@ying:/usr/src/linux-5.3# update-grub2
Sourcing file `/etc/default/grub'
Sourcing file `/etc/default/grub.d/init-select.cfg'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-5.3.0
Found initrd image: /boot/initrd.img-5.3.0
Found linux image: /boot/vmlinuz-5.0.0-29-generic
Found initrd image: /boot/initrd.img-5.0.0-29-generic
Found linux image: /boot/vmlinuz-5.0.0-13-generic
Found initrd image: /boot/initrd.img-5.0.0-13-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

好了，我们已经编译了内核 bzImage (vmlinuz-4.6.0)，放到了指定位置/boot；initrd.img-5.3.0、System.map-5.3.0 等辅助资源，同样放到了/boot；我们也配置了 grub。现在，请你重启主机系统，期待编译过的 Linux 操作系统内核正常运行！



```
ying@ying:~$ uname -r
5.3.0
ying@ying:~$
```

做到这一步不容易。初次接触 Linux 的新手，恐怕没这么顺利。

最后，完成整个实验后，使用命令：`#make clean` 清除编译时的临时文件

```
root@ying: /usr/src/linux-5.3
root@ying:/usr/src/linux-5.3# make clean
CLEAN .
CLEAN arch/x86/entry/vdso
CLEAN arch/x86/kernel/cpu
CLEAN arch/x86/kernel
```

五、讨论和心得

通过本次实验，我学习了如何编译以及安装 Linux 内核，也对内核有了一些了解，在实验过程中主要遇到了如下问题：

1. 在安装内核环境时，需要给出 sudo 权限，否则将会无法安装。

```
ying@ying:~$ apt-get install kernel-package libncurses5-dev
E: 无法打开锁文件 /var/lib/dpkg/lock-frontent - open (13: 权限不够)
E: 无法获取 dpkg 前端锁 (/var/lib/dpkg/lock-frontent), 请查看您是否正以 root 用户运行?
```

2. 在执行 make menuconfig 命令时，报了如下错误，原因是 flex 没有安装，因此需要执行 apt-get install flex 命令。对于此后出现的一系列错误都采取同样的方式进行。

```
root@ying:/usr/src/linux-5.3# make menuconfig
HOSTCC scripts/basic/fixdep
UPD scripts/kconfig/mconf-cfg
HOSTCC scripts/kconfig/mconf.o
HOSTCC scripts/kconfig/lxdialog/checklist.o
HOSTCC scripts/kconfig/lxdialog/inputbox.o
HOSTCC scripts/kconfig/lxdialog/menubox.o
HOSTCC scripts/kconfig/lxdialog/textbox.o
HOSTCC scripts/kconfig/lxdialog/util.o
HOSTCC scripts/kconfig/lxdialog/yesno.o
HOSTCC scripts/kconfig/confdata.o
HOSTCC scripts/kconfig/expr.o
LEX scripts/kconfig/lexer.lex.c
/bin/sh: 1: flex: not found
make[1]: *** [scripts/Makefile.lib:196: scripts/kconfig/lexer.lex.c] 错误 127
make: *** [Makefile:562: menuconfig] 错误 2
root@ying:/usr/src/linux-5.3#
```

六、附录

参考资料：李善平《边干边学：Linux 内核指导》

浙江大学实验报告

课程名称： 操作系统 实验类型： 综合

实验项目名称： Linux 内核模块

学生姓名： 应承峻 专业： 软件工程 学号： 3170103456

电子邮件地址： 3170103456@zju.edu.cn 手机： 17326084929

实验地点： 玉泉曹光彪西 503 实验日期： 2019 年 10 月 5 日

一、实验目的和要求

内核模块是 Linux 操作系统中一个比较独特的机制。通过这一章学习，希望能够理解 Linux 提出内核模块这个机制的意义；理解并掌握 Linux 实现内核模块机制的基本技术路线；运用 Linux 提供的工具和命令，掌握操作内核模块的方法。

二、实验内容

针对三个层次的要求，本章安排了 3 个实验。

第一个实验，编写一个很简单的内核模块。虽然简单，但它已经具备了内核模块的基本要素。与此同时，初步阅读编制内核模块所需要的 Makefile。

第二个实验，将多个源文件，合并到一个内核模块中。上述实验过程中，将会遇到 Linux 为此开发的内核模块操作工具 lsmod、insmod、rmmod 等。

第三个实验，编写一个 Linux 的内核模块，其功能是遍历操作系统所有进程。该内核模块输出系统中每个进程的：名字、进程 pid、进程的状态、父进程的名字等；以及统计系统中进程个数，包括统计系统中 TASK_RUNNING、TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE、TASK_ZOMBIE、TASK_STOPPED 等（还有其他状态）状态进程的个数。同时还需要编写一个用

户态下执行的程序，格式化输出（显示）内核模块输出的内容。

三、主要仪器设备

笔记本电脑 1 台，相关配置如下：

处理器 英特尔 Core i7-8750H @ 2.20GHz 六核

内存 16 GB （三星 DDR4 2667MHz）

主硬盘 PeM280240GP4C15B （240 GB/固态硬盘）

显卡 Nvidia GeForce GTX 1060 （6 GB）

操作系统环境：Windows 10 64 位（DirectX 12）

Linux 版本： ubuntu-19.04

四、操作方法和实验步骤

1. 编写一个简单的内核模块

看了这些理论概念，你是不是有点不耐烦了：“我什么时候才能开始在机子上实现一个模块啊？” 好吧，在进一步介绍模块的实现机制以前，我们先试着写一个非常简单的模块程序，它可以在 2.6.15 的版本上实现，对于低于 2.4 的内核版本可能还需要做一些调整，这儿就不具体讲了。

helloworld.c

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void)
{
    printk(KERN_INFO "Hello World!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye!\n");
}
```

```
}  
MODULE_LICENSE("GPL");
```

说明:

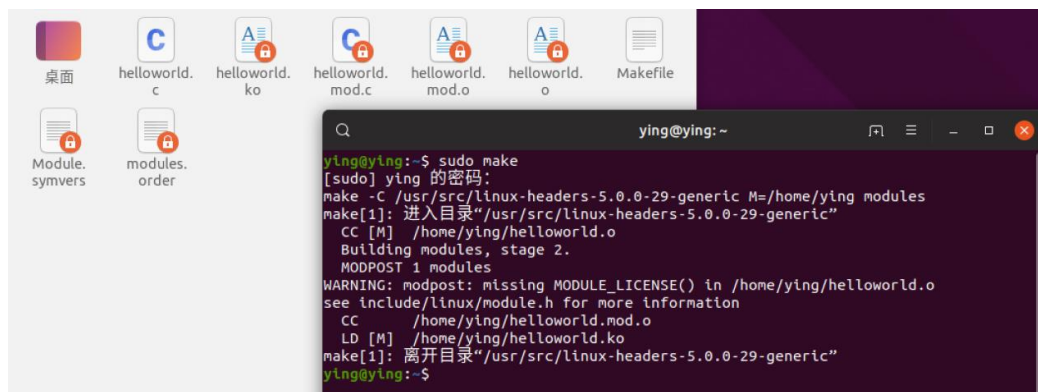
1. 任何模块程序的编写都需要包含 `linux/module.h` 这个头文件, 这个文件包含了对模块的结构定义以及模块的版本控制。文件里的主要数据结构我们会在后面详细介绍。
2. 函数 `init_module()` 和函数 `cleanup_module()` 是模块编程中最基本的也是必须的两个函数。`init_module()` 向内核注册模块所提供的新功能; `cleanup_module()` 负责注销所有由模块注册的功能。
3. 注意我们在这儿使用的是 `printk()` 函数(不要习惯性地写成 `printf`), `printk()` 函数是由 Linux 内核定义的, 功能与 `printf` 相似。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。

接下来, 我们就要编译和加载这个模块了。在前面的章节里我们已经学习了如何使用 `gcc`, 现在还要注意的一点就是: 确定你现在是超级用户。因为只有超级用户才能加载和卸载模块。在编译内核模块前, 先准备一个 `Makefile` 文件:

```
TARGET = helloworld  
  
KDIR = /usr/src/linux  
  
PWD = $(shell pwd)  
  
obj-m += $(TARGET).o  
  
default:  
  
    make -C $(KDIR) M=$(PWD) modules
```

然后简单输入命令 `make`:

`#make`



```
ying@ying: ~  
ying@ying:~$ sudo make  
[sudo] ying 的密码:  
make -C /usr/src/linux-headers-5.0.0-29-generic M=/home/ying modules  
make[1]: 进入目录"/usr/src/linux-headers-5.0.0-29-generic"  
CC [M] /home/ying/helloworld.o  
Building modules, stage 2.  
MODPOST 1 modules  
WARNING: modpost: missing MODULE_LICENSE() in /home/ying/helloworld.o  
see include/linux/module.h for more information  
CC /home/ying/helloworld.mod.o  
LD [M] /home/ying/helloworld.ko  
make[1]: 离开目录"/usr/src/linux-headers-5.0.0-29-generic"  
ying@ying:~$
```

结果, 我们得到文件 “`helloworld.ko`”。然后执行内核模块的装入命令:

```
#insmod helloworld.ko
```

Hello World!

这个时候，生成了字符串“Hello World!”，它是在 `init_module()` 中定义的。由此说明，`helloworld` 模块已经加载到内核中了。我们可以使用 `lsmod` 命令查看。`lsmod` 命令的作用是告诉我们所有在内核中运行的模块的信息，包括模块的名称，占用空间的大小，使用计数以及当前状态和依赖性。

```
root# lsmod
```

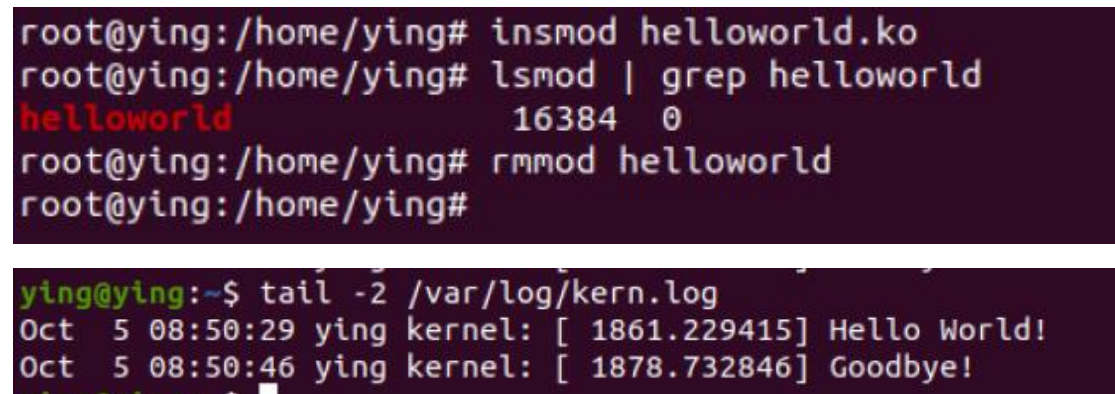
Module	Size	Used by
helloworld	464	0 (unused)
...		

最后，我们要卸载这个模块。

```
# rmmod helloworld
```

Goodbye!

编译成功后，也用 `insmod` 命令成功地将内核载入。但控制台并没有任何输出。查阅资料得，它的输出是在内核日志中，而不是在控制台中。所以需要在另一个窗口中使用命令 `tail -2 /var/log/kern.log` 查看日志文件。



The image contains two terminal screenshots. The top screenshot shows a root user at a prompt 'root@ying:/home/ying#' performing three commands: 'insmod helloworld.ko', 'lsmod | grep helloworld', and 'rmmod helloworld'. The output of the second command shows 'helloworld' with a size of 16384 and 0 uses. The bottom screenshot shows a user 'ying' at a prompt 'ying@ying:~\$' running 'tail -2 /var/log/kern.log'. The output shows two log entries: 'Oct 5 08:50:29 ying kernel: [1861.229415] Hello World!' and 'Oct 5 08:50:46 ying kernel: [1878.732846] Goodbye!'.

```
root@ying:/home/ying# insmod helloworld.ko
root@ying:/home/ying# lsmod | grep helloworld
helloworld                16384  0
root@ying:/home/ying# rmmod helloworld
root@ying:/home/ying#

ying@ying:~$ tail -2 /var/log/kern.log
Oct 5 08:50:29 ying kernel: [ 1861.229415] Hello World!
Oct 5 08:50:46 ying kernel: [ 1878.732846] Goodbye!
```

2. 多文件内核模块的 make 文件

在所有的源文件中，只有一个文件增加一行`#define __NO_VERSION__`。这是因为 `module.h` 一般包括全局变量 `kernel_version` 的定义，该全局变量包含模块编译的内核版本信息。如果你需要 `version.h`，你需要自己把它包含进去，因为定义了 `__NO_VERSION__` 后 `module.h` 就不会包含 `version.h`。

下面给出多文件的内核模块的范例。

```
Start.c


---


/* start.c
 *
 * "Hello, world" -内核模块版本
 * 这个文件仅包括启动模块例程
 */

/* 必要的头文件 */

/* 内核模块中的标准 */
#include <linux/kernel.h> /*我们在做内核的工作 */
#include <linux/module.h>

/*初始化模块 */
int init_module()
{
    printk("Hello, world!\n");

    /* 如果我们返回一个非零值，那就意味着
    * init_module 初始化失败并且内核模块
    * 不能加载 */
    return 0;
}



---



stop.c


---


/* stop.c
 *
 * "Hello, world" -内核模块版本
 * 这个文件仅包括关闭模块例程
 */

/*必要的头文件 */

/*内核模块中的标准 */
#include <linux/kernel.h> /*我们在做内核的工作 */

#define __NO_VERSION__
```

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

#include <linux/version.h> /* 不被 module.h 包括, 因为__NO_VERSION__ */

/* Cleanup - 撤消 init_module 所做的任何事情*/
void cleanup_module()
{
    printk("Bye!\n");
}
/*结束*/
```

这一次，helloworld 内核模块包含了两个源文件，“start.c”和“stop.c”。再来看看对于多文件内核模块，该怎么写 Makefile 文件

```
Makefile
TARGET = helloworld
KDIR = /usr/src/linux
PWD = $(shell pwd)
obj-m += $(TARGET).o
$(TARGET)-y := start.o stop.o
default:
    make -C $(KDIR) M=$(PWD) modules
```

相比前面，只增加一行：

```
$(TARGET)-y := start.o stop.o
```

因此我们创建一个 helloworld.c 文件，里面内容如下：

```
#include <start.c>
```

```
#include <stop.c>
```

然后使用保存，使用 make 命令，得到如下内容：



使用 insmod helloworld.ko 装载模块，并使用 lsmod 命令查看是否装载成功，在日志中查看输出：

```

root@ying:/home/ying/task2# insmod helloworld.ko
root@ying:/home/ying/task2# lsmod | grep helloworld
helloworld                16384  0
root@ying:/home/ying/task2# tail -5 /var/log/kern.log
Oct  6 11:29:35 ying kernel: [ 286.518821] Disabling lock debugging due to kernel taint
Oct  6 11:29:35 ying kernel: [ 286.518851] helloworld: module verification failed: signature and/or required key missing - tainting kernel
Oct  6 11:29:35 ying kernel: [ 286.520885] Hello World!
Oct  6 11:30:22 ying kernel: [ 333.327609] Goodbye!
Oct  6 13:23:40 ying kernel: [ 7130.907065] Hello World!

```

使用 `rmmod helloworld` 删除模块，在日志中查看输出：

```

root@ying:/home/ying/task2# rmmod helloworld.ko
root@ying:/home/ying/task2# lsmod | grep helloworld
root@ying:/home/ying/task2# tail -5 /var/log/kern.log
Oct  6 11:29:35 ying kernel: [ 286.518851] helloworld: module verification failed: signature and/or required key missing - tainting kernel
Oct  6 11:29:35 ying kernel: [ 286.520885] Hello World!
Oct  6 11:30:22 ying kernel: [ 333.327609] Goodbye!
Oct  6 13:23:40 ying kernel: [ 7130.907065] Hello World!
Oct  6 13:25:47 ying kernel: [ 7258.671595] Bye!
root@ying:/home/ying/task2#

```

3. 内核模块进程访问

第3题中，每个进程的进程名字、pid、进程状态、父进程的指针等在 `task_struct` 结构的字段中。在内核中使用 `printk` 函数打印有关变量的值。遍历进程可以使用 `next_task` 宏，`init_task` 进程为0号进程。

`task_struct` 结构参阅“边干边学—Linux 内核指导”教材 11.2 节；遍历进程方法可以参阅“边干边学—Linux 内核指导”教材 11.6 节。用户态下的程序是从 `/var/log/kern.log (ubuntu)` 文件中读出内核模块输出的内容。

遍历进程方法

```
for(p = &init_task; (p = next_task(p)) != &init_task; ) {
```

```
// process task
```

```
}
```

创建内核文件 `process.c`，代码如下：

```

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

```

```
int show_processes(void);
```

```
int show_processes(void) {
```

```
    int numOfRunning = 0;
```

```
    int numOfInterruptible = 0;
```

```

int numOfUnInterruptible = 0;
int numOfStopped = 0;
int numOfTraced = 0;
int numOfZombie = 0;
int numOfExitDead = 0;
int numOfDead = 0;
int numOfDefault = 0;
int numOfWaking = 0;
int numOfWakeKill = 0;
int numOfTotal = 0;
int state;
int exit_state;
int index;
char *status[]={ "DEFAULT", "TASK_RUNNING", "TASK_INTERRUPTIBLE",
"TASK_UNINTERRUPTIBLE", "TASK_UNNING", "TASK_TRACED", "EXIT_ZOMBIE",
"EXIT_DEAD", "TASK_DEAD", "TASK_WAKEKILL", "TASK_WAKING"};

struct task_struct *p;

printk(KERN_INFO "[PROCESSES INFO]:\n");
printk(KERN_INFO "Pid\t\tNAME\t\tState\t\tParentName\n");
//for(p=&init_task;(p=next_task(p))!=&init_task;) {
for_each_process(p) {
    /* State: -1 unrunnable, 0 runnable, >0 stopped */
    /* Parent: use real_parent instead of (current) parent */
    index = 0;
    numOfTotal++;

    state = p->state;
    exit_state = p->exit_state;
    if (exit_state!=0) {
        switch(exit_state) {
            case EXIT_ZOMBIE: numOfZombie++; index = 6; break; /*
EXIT_ZOMBIE 16 */
            case EXIT_DEAD: numOfExitDead++; index = 7; break; /*
EXIT_DEAD 32 */
            default: numOfDefault++; break;
        }
    } else {
        switch(state) {
            case TASK_RUNNING: numOfRunning++; index = 1; break; /*
TASK_RUNNING 0 */
            case TASK_INTERRUPTIBLE: numOfInterruptible++; index = 2;
break; /* TASK_INTERRUPTIBLE 1 */

```

```

        case TASK_UNINTERRUPTIBLE: numOfUnInterruptible++; index =
3; break; /* TASK_UNINTERRUPTIBLE 2 */
        case TASK_STOPPED: numOfStopped++; index = 4; break; /*
TASK_UNNING 4 */
        case TASK_TRACED: numOfTraced++; index = 5; break; /*
TASK_TRACED 8 */
        case TASK_DEAD: numOfDead++; index = 8; break; /* TASK_DEAD
64 */
        case TASK_WAKEKILL: numOfWakeKill++; index = 9; break; /*
TASK_WAKEKILL 128 */
        case TASK_WAKING: numOfWaking++; index = 10; break; /*
TASK_WAKING 256 */
        default: numOfDefault++; break;
    }
}
printk(KERN_INFO "%d\t\t%s\t\t%s\t\t%s\n", p->pid, p->comm,
status[index], p->real_parent->comm);
}
printk(KERN_INFO "numOfTotal: %d\n",numOfTotal);
printk(KERN_INFO "numOfRunning: %d\n",numOfRunning);
printk(KERN_INFO "numOfInterruptible: %d\n",numOfInterruptible);
printk(KERN_INFO "numOfUnInterruptible: %d\n",numOfUnInterruptible);
printk(KERN_INFO "numOfStopped: %d\n",numOfStopped);
printk(KERN_INFO "numOfTraced: %d\n",numOfTraced);
printk(KERN_INFO "numOfZombie: %d\n",numOfZombie);
printk(KERN_INFO "numOfExitDead: %d\n",numOfExitDead);
printk(KERN_INFO "numOfDead: %d\n",numOfDead);
printk(KERN_INFO "numOfDefault: %d\n",numOfDefault);
printk(KERN_INFO "numOfWakeKill: %d\n",numOfWakeKill);
printk(KERN_INFO "numOfWaking: %d\n",numOfWaking);
return 0;
}

int init_module(void) {
    return show_processes();
}

void cleanup_module(void) {
    printk(KERN_INFO "Process Printer Cleaned!\n");
}

```

然后创建用户程序（脚本）user:

```
insmod process.ko
```

```
dmesg
```

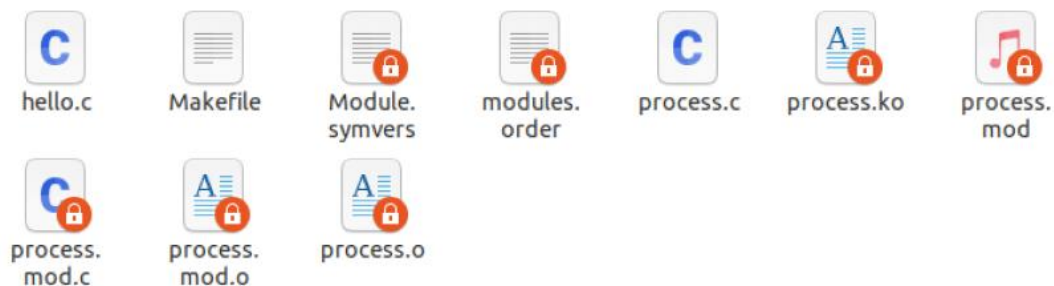
```
cat < /var/log/kern.log > klogs
```

```
rmmod process
```

使用 make 进行编译：

```
root@ying:/home/ying/task3# make
make -C /usr/src/linux M=/home/ying/task3 modules
make[1]: 进入目录“/usr/src/linux-5.3”
  CC [M] /home/ying/task3/process.o
Building modules, stage 2.
MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/ying/task3/process.o
see include/linux/module.h for more information
  CC /home/ying/task3/process.mod.o
  LD [M] /home/ying/task3/process.ko
make[1]: 离开目录“/usr/src/linux-5.3”
```

生成文件如下：



执行./user 这一 bash 脚本后，可以看到控制台输出了日志：

```
root@ying:/home/ying/task3
Oct 7 12:54:39 ying kernel: [ 3281.178562] 2863 kworker/0:0 DEFAULT kthreadd
Oct 7 12:54:39 ying kernel: [ 3281.178563] 2897 kworker/5:2 DEFAULT kthreadd
Oct 7 12:54:39 ying kernel: [ 3281.178563] 2909 kworker/u256:2 DEFAULT kthreadd
Oct 7 12:54:39 ying kernel: [ 3281.178564] 2910 kworker/10:2 DEFAULT kthreadd
Oct 7 12:54:39 ying kernel: [ 3281.178564] 2933 bash TASK_INTERRUPTIBLE gnome-terminal-
Oct 7 12:54:39 ying kernel: [ 3281.178565] 2939 sudo TASK_INTERRUPTIBLE bash
Oct 7 12:54:39 ying kernel: [ 3281.178566] 2940 su TASK_INTERRUPTIBLE sudo
Oct 7 12:54:39 ying kernel: [ 3281.178566] 2941 bash TASK_INTERRUPTIBLE su
Oct 7 12:54:39 ying kernel: [ 3281.178567] 3343 bash TASK_INTERRUPTIBLE bash
Oct 7 12:54:39 ying kernel: [ 3281.178567] 3344 insmod TASK_RUNNING bash
Oct 7 12:54:39 ying kernel: [ 3281.178567] numOfTotal: 364
Oct 7 12:54:39 ying kernel: [ 3281.178568] numOfRunning: 2
Oct 7 12:54:39 ying kernel: [ 3281.178568] numOfInterruptible: 253
Oct 7 12:54:39 ying kernel: [ 3281.178582] numOfUninterruptible: 0
Oct 7 12:54:39 ying kernel: [ 3281.178582] numOfStopped: 0
Oct 7 12:54:39 ying kernel: [ 3281.178582] numOfTraced: 0
Oct 7 12:54:39 ying kernel: [ 3281.178583] numOfZombie: 0
Oct 7 12:54:39 ying kernel: [ 3281.178583] numOfExitDead: 0
Oct 7 12:54:39 ying kernel: [ 3281.178583] numOfDead: 0
Oct 7 12:54:39 ying kernel: [ 3281.178583] numOfDefault: 109
Oct 7 12:54:39 ying kernel: [ 3281.178583] numOfWakeKill: 0
Oct 7 12:54:39 ying kernel: [ 3281.178584] numOfWaking: 0
Oct 7 12:54:39 ying kernel: [ 3281.181766] Process Printer Cleaned!
root@ying:/home/ying/task3#
```

与此同时将日志重定向到了 klogs 文件中：

```
打开(O)  文件  klogs [只读]  保存(S)  菜单  窗口  退出
~/task3
Oct 7 12:54:39 yling kernel: [ 3281.178543] 2285 bash TASK_INTERRUPTIBLE gnome-terminal-
Oct 7 12:54:39 yling kernel: [ 3281.178543] 2298 fwupd TASK_INTERRUPTIBLE systemd
Oct 7 12:54:39 yling kernel: [ 3281.178544] 2305 boltd TASK_INTERRUPTIBLE systemd
Oct 7 12:54:39 yling kernel: [ 3281.178544] 2448 sudo TASK_INTERRUPTIBLE bash
Oct 7 12:54:39 yling kernel: [ 3281.178545] 2504 su TASK_INTERRUPTIBLE sudo
Oct 7 12:54:39 yling kernel: [ 3281.178559] 2505 bash TASK_INTERRUPTIBLE su
Oct 7 12:54:39 yling kernel: [ 3281.178560] 2720 gvfsd-metadata TASK_INTERRUPTIBLE systemd
Oct 7 12:54:39 yling kernel: [ 3281.178560] 2725 gedlt TASK_INTERRUPTIBLE systemd
Oct 7 12:54:39 yling kernel: [ 3281.178561] 2784 kworker/9:0 DEFAULT kthreadd
Oct 7 12:54:39 yling kernel: [ 3281.178561] 2803 kworker/10:0 DEFAULT kthreadd
Oct 7 12:54:39 yling kernel: [ 3281.178562] 2819 kworker/u256:0 DEFAULT kthreadd
Oct 7 12:54:39 yling kernel: [ 3281.178562] 2863 kworker/0:0 DEFAULT kthreadd
Oct 7 12:54:39 yling kernel: [ 3281.178563] 2897 kworker/5:2 DEFAULT kthreadd
Oct 7 12:54:39 yling kernel: [ 3281.178563] 2909 kworker/u256:2 DEFAULT kthreadd
Oct 7 12:54:39 yling kernel: [ 3281.178564] 2910 kworker/10:2 DEFAULT kthreadd
Oct 7 12:54:39 yling kernel: [ 3281.178564] 2933 bash TASK_INTERRUPTIBLE gnome-terminal-
Oct 7 12:54:39 yling kernel: [ 3281.178565] 2939 sudo TASK_INTERRUPTIBLE bash
Oct 7 12:54:39 yling kernel: [ 3281.178566] 2940 su TASK_INTERRUPTIBLE sudo
Oct 7 12:54:39 yling kernel: [ 3281.178566] 2941 bash TASK_INTERRUPTIBLE su
Oct 7 12:54:39 yling kernel: [ 3281.178567] 3343 bash TASK_INTERRUPTIBLE bash
Oct 7 12:54:39 yling kernel: [ 3281.178567] 3344 lnsmod TASK_RUNNING bash
Oct 7 12:54:39 yling kernel: [ 3281.178567] numOfTotal: 364
Oct 7 12:54:39 yling kernel: [ 3281.178568] numOfRunning: 2
Oct 7 12:54:39 yling kernel: [ 3281.178568] numOfInterruptible: 253
Oct 7 12:54:39 yling kernel: [ 3281.178582] numOfUninterruptible: 0
Oct 7 12:54:39 yling kernel: [ 3281.178582] numOfStopped: 0
Oct 7 12:54:39 yling kernel: [ 3281.178582] numOfTraced: 0
Oct 7 12:54:39 yling kernel: [ 3281.178583] numOfZombie: 0
Oct 7 12:54:39 yling kernel: [ 3281.178583] numOfExitDead: 0
Oct 7 12:54:39 yling kernel: [ 3281.178583] numOfDead: 0
Oct 7 12:54:39 yling kernel: [ 3281.178583] numOfDefault: 109
Oct 7 12:54:39 yling kernel: [ 3281.178583] numOfWakeupkill: 0
Oct 7 12:54:39 yling kernel: [ 3281.178584] numOfWaking: 0
Oct 7 12:54:39 yling kernel: [ 3281.181766] Process Printer Cleaned!
纯文本 制表符宽度: 8 第 1 行, 第 1 列 插入
```

五、讨论和心得

通过本次实验，我了解了 `task_struct` 这一结构体中一些常见的属性，能够通过它来实现遍历进程的操作。在实验过程中遇到了如下问题：

1. 内核模块的装载需要 `root` 权限，因此需要先使用 `sudo su` 切换到超级管理员权限。
2. 在编写内核模块声明函数时，使用 `int show_processes()` 声明原型在编译时会报错说不符合函数原型，因此需要使用 `int show_processes(void)` 来声明这个函数没有参数。
3. 在使用循环遍历时，编译出现了报错问题，去检查了 `linux/sched.h` 发现该文件下没有定义 `init_task` 指针，查阅资料后，选择加入头文件

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

然后使用 `for_each_process(p)` 代替遍历


```

/home/ying/task3/process.c:21:10: error: 'init_task' undeclared (first use in this function); did you mean 'init_stack'?
   for (p=&init_task;(p=next_task(p))!=&init_task;) {
       ^~~~~~
/home/ying/task3/process.c:21:10: note: each undeclared identifier is reported only once for each function it appears in
/home/ying/task3/process.c:21:23: error: implicit declaration of function 'next_task'; did you mean 'next_arg'? [-Werror=implicit-function-declaration]
   for (p=&init_task;(p=next_task(p))!=&init_task;) {
                       ^~~~~~
/home/ying/task3/process.c:21:22: warning: assignment to 'struct task_struct *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
   for (p=&init_task;(p=next_task(p))!=&init_task;) {

```

最终通过编译。

六、附录

参考资料：李善平《边干边学：Linux 内核指导》