

中国矿业大学计算机学院



2021-2022(2)本科生 Linux 操作系统课程报告

专业班级 信息安全 19-1 班 学生姓名 江一川 学 号 08193041

序号	报告内容	基础理论掌握程度	综合知识应用能力	报告内容	报告格式	完成状况	工作量	学习、工作态度	抄袭现象	其它	综合成绩
1	Linux 命令	熟练 <input type="checkbox"/> 较熟练 <input type="checkbox"/> 一般 <input type="checkbox"/> 不熟练 <input type="checkbox"/>	强 <input type="checkbox"/> 较强 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	完整 <input type="checkbox"/> 较完整 <input type="checkbox"/> 一般 <input type="checkbox"/> 不完整 <input type="checkbox"/>	规范 <input type="checkbox"/> 较规范 <input type="checkbox"/> 一般 <input type="checkbox"/> 不规范 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	饱满 <input type="checkbox"/> 适中 <input type="checkbox"/> 一般 <input type="checkbox"/> 欠缺 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	学号: 姓名:		
2	shell 编程	熟练 <input type="checkbox"/> 较熟练 <input type="checkbox"/> 一般 <input type="checkbox"/> 不熟练 <input type="checkbox"/>	强 <input type="checkbox"/> 较强 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	完整 <input type="checkbox"/> 较完整 <input type="checkbox"/> 一般 <input type="checkbox"/> 不完整 <input type="checkbox"/>	规范 <input type="checkbox"/> 较规范 <input type="checkbox"/> 一般 <input type="checkbox"/> 不规范 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	饱满 <input type="checkbox"/> 适中 <input type="checkbox"/> 一般 <input type="checkbox"/> 欠缺 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	学号: 姓名:		
3	进程控制编程	熟练 <input type="checkbox"/> 较熟练 <input type="checkbox"/> 一般 <input type="checkbox"/> 不熟练 <input type="checkbox"/>	强 <input type="checkbox"/> 较强 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	完整 <input type="checkbox"/> 较完整 <input type="checkbox"/> 一般 <input type="checkbox"/> 不完整 <input type="checkbox"/>	规范 <input type="checkbox"/> 较规范 <input type="checkbox"/> 一般 <input type="checkbox"/> 不规范 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	饱满 <input type="checkbox"/> 适中 <input type="checkbox"/> 一般 <input type="checkbox"/> 欠缺 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	学号: 姓名:		
4	Linux 网络编程	熟练 <input type="checkbox"/> 较熟练 <input type="checkbox"/> 一般 <input type="checkbox"/> 不熟练 <input type="checkbox"/>	强 <input type="checkbox"/> 较强 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	完整 <input type="checkbox"/> 较完整 <input type="checkbox"/> 一般 <input type="checkbox"/> 不完整 <input type="checkbox"/>	规范 <input type="checkbox"/> 较规范 <input type="checkbox"/> 一般 <input type="checkbox"/> 不规范 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	饱满 <input type="checkbox"/> 适中 <input type="checkbox"/> 一般 <input type="checkbox"/> 欠缺 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	学号: 姓名:		
5	编程附加题	熟练 <input type="checkbox"/> 较熟练 <input type="checkbox"/> 一般 <input type="checkbox"/> 不熟练 <input type="checkbox"/>	强 <input type="checkbox"/> 较强 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	完整 <input type="checkbox"/> 较完整 <input type="checkbox"/> 一般 <input type="checkbox"/> 不完整 <input type="checkbox"/>	规范 <input type="checkbox"/> 较规范 <input type="checkbox"/> 一般 <input type="checkbox"/> 不规范 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	饱满 <input type="checkbox"/> 适中 <input type="checkbox"/> 一般 <input type="checkbox"/> 欠缺 <input type="checkbox"/>	好 <input type="checkbox"/> 较好 <input type="checkbox"/> 一般 <input type="checkbox"/> 差 <input type="checkbox"/>	学号: 姓名:		

任课教师：杨东平

年 月 日



中文摘要

Linux 是一种免费使用和自由传播的类 UNIX 操作系统，是一个基于 POSIX 的多用户、多任务、支持多线程和多 CPU 的操作系统。对于一个信息安全专业的学生来说，熟悉并学会使用 linux 操作系统非常重要。此报告主要从 linux 命令、shell 编程、进程控制编程、网络编程四个方面对 linux 系统进行简单的介绍。

Abstract

Linux is a unix-like operating system that is free to use and freely distributed. It is a posix-based multi-user, multi-task, multi-threading and multi-CPU operating system. For a student majoring in information security, it is very important to be familiar with and learn to use the Linux operating system. This report mainly introduces the Linux system from four aspects: Linux command, shell programming, process control programming and network programming.



目 录

中文摘要	I
Abstract	I
1 Linux 命令	1
1.1 ls 命令	1
1.2 whereis 命令	6
1.3 cd 命令	8
1.4 df 命令	10
1.5 find 命令	13
1.6 du 命令	19
1.7 grep 命令	21
1.8 管道符综合使用	25
1.9 体会	26
1.10 参考文献	27
2 shell 编程	28
2.1 最大公因数及最小公倍数求解算法	28
2.2 选择排序	30
2.3 进制转换	32
2.4 维基利亚密码	35
2.5 体会	38
2.6 参考文献	38
3 进程控制编程	39
3.1 fork 系统调用	39
3.2 exec 系统调用	40
3.3 exit 系统调用	41
3.4 wait 系统调用	42
3.5 代码实例	43
3.6 运行截图	47
3.7 体会	49
3.8 参考文献	49
4 Linux 网络编程	50
4.1 socket 编程常见的系统调用	50
4.2 实现方法	55
4.3 源代码	55
4.4 运行截图	63
4.5 体会	66
4.6 参考文献	66
5 编程附加题	67
5.1 程序介绍	67
5.2 源代码	67
5.3 运行截图	74
5.4 体会	74
5.5 参考文献	74



1 Linux 命令

要求：(1) 尽可能的多测试 LINUX 平台的各种命令。

(2) 学习使用管道符组合使用命令。

(3) 每条命令必须阐明该命令的作用、参数、使用方法并将命令运行结果截图附上。

说明：如果撰写规范不符合《计算机学院考查类课程报告撰写规范》要求的，整体上酌情扣除 1-10 分。

1.1 ls 命令

1.1.1 命令功能

显示指定工作目录下的内容(所含之文件及子目录)，如果不指定文件或目录则显示当前目录的内容。

1.1.2 命令格式

`ls [-alrtAFR] [文件或目录]`

1.1.3 命令参数及作用

参数	作用
-a	显示所有文件及目录 (ls 默认将“.”开头的文件名或目录名视为隐藏档，不会列出)
-l	除文件名外，还列出文件类型、权限、拥有者、文件大小等信息
-r	将文件以相反次序显示(默认按英文字母次序)
-t	将文件依建立时间之先后次序列出
-A	同 -a，但不列出“.”(当前目录)及“..”(父目录)
-F	在文件名后加文件类别标志，如可执行档则加“*”，目录则加“/”
-R	递归显示，列出子目录及其下的文件
-b	将文件名中的不可见字符，使用“\”开头的八进制转换字符表示
-B	不要显示以“~”结尾的文件
-c	按文件修改时间排序，可以使用“-l”选项显示创建时间
-C	使用列的顺序输出文件列表
-d	仅显示目录名，而不显示目录下的内容列表
-G	不显示文件的用户组
-h	自动将文件大小使用方便阅读的方式表示
-H	单位转换使用 1000，而不是 1024
-i	显示文件索引节点号 (inode)，一个索引节点代表一个文件
-k	以 K 为单位显示文件大小
-L	列出符号链接指向的条目，而不是列出符号链接
-n	以用户识别码和群组识别码替代其名称
-N	输出原始名称，对特殊字符不做特别处理
-o	使用没有组信息的长列表格式
-Q	给名称加上双引号
-s	显示文件和目录的大小，以区块为单位
-S	根据文件大小排序
-t	使用修改时间排序
-u	按最后访问时间排序；使用“-l”显示最后访问时间
-x	按行的顺序显示，而不是按列的顺序



1.1.4 命令使用

(1) 显示当前目录下所有非隐藏文件及目录

命令: ls

```
(kali㉿kali)-[~/桌面]
└─$ ls
aaa.sh  client  computer.c  serve  test1.sh  thread
a.c     client.txt  cumt  serve.c  test2.sh  thread.c
cash.c  computer  cumt.txt  serve.txt  test.sh
```

(2) 显示当前目录下所有文件及目录

命令: ls -a

```
(kali㉿kali)-[~/桌面]
└─$ ls -a
.      a.c      computer  serve    test2.sh  thread
..     cash.c  computer.c  serve.c  test.sh   thread.c
aaa.sh client  cumt      serve.txt .test.sh.swo
.aaa.swp client.txt cumt.txt  test1.sh .test.sh.swp
```

(3) 将当前目录下所有非隐藏文件及目录按照字母次序的相反次序显示

命令: ls -r

```
(kali㉿kali)-[~/桌面]
└─$ ls -r
thread.c  test2.sh  serve.c  cumt  client.txt  a.c
thread    test1.sh  serve    computer.c  client  aaa.sh
test.sh   serve.txt  cumt.txt  computer  cash.c
```

(4) 显示当前目录下所有非隐藏文件及目录的详细信息

命令: ls -l

```
(kali㉿kali)-[~/桌面]
└─$ ls -l
总用量 140
-rwxrwxrwx 1 kali kali 336 4月 5 22:19 aaa.sh
-rw-r--r-- 1 kali kali 402 4月 8 11:44 a.c
-rw-r--r-- 1 kali kali 139 4月 9 21:55 cash.c
-rwxr-xr-x 1 kali kali 17000 4月 8 13:13 client
-rwxrwxrwx 1 kali kali 800 4月 8 15:33 client.txt
-rwxr-xr-x 1 kali kali 17048 4月 8 16:19 computer
-rwxrwxrwx 1 kali kali 5625 4月 8 16:18 computer.c
-rw-r--r-- 1 kali kali 72 4月 5 00:09 cumt
-rw-r--r-- 1 kali kali 789 4月 8 10:36 cumt.txt
-rwxr-xr-x 1 kali kali 17136 4月 8 16:19 serve
-rw-r--r-- 1 kali kali 5520 4月 8 16:19 serve.c
-rwxrwxrwx 1 kali kali 800 4月 8 15:34 serve.txt
-rwxr-xr-x 1 kali kali 1324 4月 6 09:26 test1.sh
-rwxrw-rw- 1 kali kali 835 4月 6 08:56 test2.sh
-rwxr-xr-x 1 kali kali 1038 4月 5 21:40 test.sh
-rwxr-xr-x 1 kali kali 16568 4月 9 21:57 thread
-rw-r--r-- 1 kali kali 2758 4月 9 21:57 thread.c
```



(5) 将当前目录下所有非隐藏文件及目录按照创建时间先后排列显示

命令: `ls -t`

```
(kali㉿kali)-[~/桌面]
$ ls -t
thread  computer  computer.c  client  test1.sh  test.sh
thread.c  serve  serve.txt  a.c  test2.sh  cumt
cash.c  serve.c  client.txt  cumt.txt  aaa.sh
```

(6) 显示当前目录下所有文件及目录（当前目录，父目录除外）

命令: `ls -A`

```
(kali㉿kali)-[~/桌面]
$ ls -A
aaa.sh  cash.c  computer  cumt.txt  serve.txt  test.sh  thread
.aaa.swp  client  computer.c  serve  test1.sh  .test.sh.swo  thread.c
a.c  client.txt  cumt  serve.c  test2.sh  .test.sh.swp
```

(7) 显示当前目录下所有非隐藏文件及目录并在其后添加文件类别标志

命令: `ls -F`

```
(kali㉿kali)-[~/桌面]
$ ls -F
aaa.sh*  client*  computer.c*  serve*  test1.sh*  thread*
a.c  client.txt*  cumt  serve.c  test2.sh*  thread.c
cash.c  computer*  cumt.txt  serve.txt*  test.sh*
```

(8) 递归显示当前目录下所有子目录及其下的文件

命令: `ls -R`

```
(kali㉿kali)-[~]
$ ls -R
.:
公共 模板 视频 图片 文档 下载 音乐 桌面 test1 test2 test3.elf

./公共:

./模板:

./视频:

./图片:

./文档:

./下载:

./音乐:

./桌面:
aaa.sh  client  computer.c  serve  test1.sh  thread
a.c  client.txt  cumt  serve.c  test2.sh  thread.c
cash.c  computer  cumt.txt  serve.txt  test.sh

./test3.elf:
```




(9) 显示当前目录下所有非隐藏文件及目录，并给所有文件名加上双引号

命令: `ls -lQ`

```
(kali㉿kali)-[~/桌面]
$ ls -lQ
总用量 140
-rwxrwxrwx 1 kali kali 336 4月 5 22:19 "aaa.sh"
-rw-r--r-- 1 kali kali 402 4月 8 11:44 "a.c"
-rw-r--r-- 1 kali kali 139 4月 9 21:55 "cash.c"
-rwxr-xr-x 1 kali kali 17000 4月 8 13:13 "client"
-rwxrwxrwx 1 kali kali 800 4月 8 15:33 "client.txt"
-rwxr-xr-x 1 kali kali 17048 4月 8 16:19 "computer"
-rwxrwxrwx 1 kali kali 5625 4月 8 16:18 "computer.c"
-rw-r--r-- 1 kali kali 72 4月 5 00:09 "cumt"
-rw-r--r-- 1 kali kali 789 4月 8 10:36 "cumt.txt"
-rwxr-xr-x 1 kali kali 17136 4月 8 16:19 "serve"
-rw-r--r-- 1 kali kali 5520 4月 8 16:19 "serve.c"
-rwxrwxrwx 1 kali kali 800 4月 8 15:34 "serve.txt"
-rwxr-xr-x 1 kali kali 1324 4月 6 09:26 "test1.sh"
-rwxrw-rw- 1 kali kali 835 4月 6 08:56 "test2.sh"
-rwxr-xr-x 1 kali kali 1038 4月 5 21:40 "test.sh"
-rwxr-xr-x 1 kali kali 16568 4月 9 21:57 "thread"
-rw-r--r-- 1 kali kali 2758 4月 9 21:57 "thread.c"
```

(10) 显示当前目录下所有文件和目录，并以区块为单位打印出它们的大小。

命令: `ls -as`

```
(kali㉿kali)-[~/桌面]
$ ls -as
总用量 184
4 .          4 cash.c      4 cumt        4 test1.sh    20 thread
4 ..         20 client      4 cumt.txt    4 test2.sh    4 thread.c
4 aaa.sh     4 client.txt  20 serve      4 test.sh
12 .aaa.swp  20 computer   8 serve.c    12 .test.sh.swo
4 a.c        8 computer.c  4 serve.txt  12 .test.sh.swp
```

(11) 显示当前目录下所有非隐藏文件和目录，并打印出它们的文件索引节点号

命令: `ls -li`

```
(kali㉿kali)-[~/桌面]
$ ls -li
407691 aaa.sh      407698 computer  407727 serve.c   407616 thread
407715 a.c        407720 computer.c 407726 serve.txt  407730 thread.c
407718 cash.c    407640 cumt      407638 test1.sh
406824 client      407704 cumt.txt 407648 test2.sh
407701 client.txt  407722 serve    407627 test.sh
```

(12) 按行显示出当前目录下所有非隐藏文件及目录

命令: `ls -x`

```
(kali㉿kali)-[~/桌面]
$ ls -x
aaa.sh  a.c  cash.c  client  client.txt  computer  computer.c  cumt
cumt.txt  serve  serve.c  serve.txt  test1.sh  test2.sh  test.sh  thread
thread.c
```



(13) 显示当前目录下所有非隐藏文件及目录，并且每一行仅显示一个文件名

命令: `ls -l`

```
(kali㉿kali)-[~/桌面]
└─$ ls -l
aaa.sh
a.c
cash.c
client
client.txt
computer
computer.c
cumt
cumt.txt
serve
serve.c
serve.txt
test1.sh
test2.sh
test.sh
thread
thread.c
```

(14) 显示当前目录下所有非隐藏文件及目录，并将文件大小转换成方便阅读的形式

命令: `ls -lh`

```
(kali㉿kali)-[~/桌面]
└─$ ls -lh
总用量 140K
-rwxrwxrwx 1 kali kali 336 4月 5 22:19 aaa.sh
-rw-r--r-- 1 kali kali 402 4月 8 11:44 a.c
-rw-r--r-- 1 kali kali 139 4月 9 21:55 cash.c
-rwxr-xr-x 1 kali kali 17K 4月 8 13:13 client
-rwxrwxrwx 1 kali kali 800 4月 8 15:33 client.txt
-rwxr-xr-x 1 kali kali 17K 4月 8 16:19 computer
-rwxrwxrwx 1 kali kali 5.5K 4月 8 16:18 computer.c
-rw-r--r-- 1 kali kali 72 4月 5 00:09 cumt
-rw-r--r-- 1 kali kali 789 4月 8 10:36 cumt.txt
-rwxr-xr-x 1 kali kali 17K 4月 8 16:19 serve
-rw-r--r-- 1 kali kali 5.4K 4月 8 16:19 serve.c
-rwxrwxrwx 1 kali kali 800 4月 8 15:34 serve.txt
-rwxr-xr-x 1 kali kali 1.3K 4月 6 09:26 test1.sh
-rwxrw-rw- 1 kali kali 835 4月 6 08:56 test2.sh
-rwxr-xr-x 1 kali kali 1.1K 4月 5 21:40 test.sh
-rwxr-xr-x 1 kali kali 17K 4月 9 21:57 thread
-rw-r--r-- 1 kali kali 2.7K 4月 9 21:57 thread.c
```




1.2 whereis 命令

1.2.1 命令功能

搜索命令（二进制文件）及帮助文档所在路径。

1.2.2 命令格式

whereis [选项] [命令名]

1.2.3 命令参数及作用

参数	作用
-b	只查找二进制文件
-B	只在指定的目录下查找二进制文件
-m	只查找帮助手册
-s	只查找源代码
-f	不显示文件名前的路径名称
-u	查找不包含指定类型的文件
-l	输出有效查找路径
-M	只在指定的目录下查找帮助手册
-S	只在指定的目录下查找源代码

1.2.4 命令使用

(1) 查看指令 whereis 的位置

命令: whereis whereis

```
(root@kali)-[~]
# whereis whereis
whereis: /usr/bin/whereis /usr/share/man/man1/whereis.1.gz
```

(2) 查看指令 whereis 的二进制文件的位置

命令: whereis -b whereis

```
(root@kali)-[~]
# whereis -b whereis
whereis: /usr/bin/whereis
```

(3) 查看指令 whereis 的帮助文档的位置

命令: whereis -m whereis

```
(root@kali)-[~]
# whereis -m whereis
whereis: /usr/share/man/man1/whereis.1.gz
```

(4) 查看指令 whereis 的源代码的位置

命令: whereis -s whereis

```
(root@kali)-[~]
# whereis -s whereis
whereis:
```



(5) 显示指令 whereis 有效查找路径

命令: whereis -l whereis

```
(root@kali)-[~]
# whereis -l whereis
bin: /usr/bin
bin: /usr/sbin
bin: /usr/lib/x86_64-linux-gnu
bin: /usr/lib
bin: /usr/lib32
bin: /usr/lib64
bin: /etc
bin: /usr/games
bin: /usr/local/bin
bin: /usr/local/sbin
bin: /usr/local/etc
bin: /usr/local/lib
bin: /usr/local/games
bin: /usr/include
bin: /usr/local
bin: /usr/libexec
bin: /usr/share
man: /usr/share/man/es
man: /usr/share/man/ru
man: /usr/share/man/man4
man: /usr/share/man/pt_BR
man: /usr/share/man/man1
man: /usr/share/man/zh_TW
man: /usr/share/man/man7
man: /usr/share/man/man5
man: /usr/share/man/id
man: /usr/share/man/man6
man: /usr/share/man/nl
man: /usr/share/man/sr
man: /usr/share/man/de
man: /usr/share/man/man8
man: /usr/share/man/fr.UTF-8
man: /usr/share/man/uk
man: /usr/share/man/cs
man: /usr/share/man/zh
man: /usr/share/man/man9
man: /usr/share/man/man3
man: /usr/share/man/hu
man: /usr/share/man/ko
man: /usr/share/man/pt
man: /usr/share/man/da
man: /usr/share/man/hr
man: /usr/share/man/sk
man: /usr/share/man/sl
man: /usr/share/man/fr.ISO8859-1
man: /usr/share/man/it
man: /usr/share/man/ro
man: /usr/share/man/pl
man: /usr/share/man/ja
man: /usr/share/man/fi
man: /usr/share/man/sv
man: /usr/share/man/tr
man: /usr/share/man/zh_CN
man: /usr/share/man/fr
man: /usr/share/man/man2
man: /usr/share/info
whereis: /usr/bin/whereis /usr/share/man/man1/whereis.1.gz
```



1.3 cd 命令

1.3.1 命令功能

用于切换当前工作目录。

1.3.2 命令格式

cd [目标目录]

1.3.3 命令参数及作用

参数	作用
cd ~	进入当前用户的 home 目录
cd -	进入上次目录
cd ..	进入上一级目录
cd .	进入当前目录
cd /	进入到根目录

1.3.4 命令使用

(1) 跳转到目录/usr/bin/

命令: cd /usr/bin

```
(root@kali)~[~]
# cd /usr/bin
(root@kali)~[/usr/bin]
#
```

(2) 跳转到当前用户的 home 目录

命令: cd ~

```
(root@kali)~[/usr/bin]
# cd ~
(root@kali)~[~]
#
```

(3) 跳转到上次访问的目录

命令: cd -

```
(root@kali)~[~]
# cd -
/usr/bin
(root@kali)~[/usr/bin]
#
```

(4) 跳转到当前目录的上一级目录

cd ..

```
(root@kali)~[/usr/bin]
# cd ..
(root@kali)~[/usr]
#
```



(5) 进入当前目录

命令: `cd .`

```
(root👤kali)-[/usr]
# cd .

(root👤kali)-[/usr]
#
```

(6) 跳转到根目录

命令: `cd /`

```
(root👤kali)-[/usr]
# cd /

(root👤kali)-[/]
#
```




1.4 df 命令

1.4.1 命令功能

用于显示目前在 Linux 系统上的文件系统磁盘使用情况统计。

1.4.2 命令格式

df [选项] [类型]

1.4.3 命令参数及作用

参数	作用
-l	仅显示本地磁盘(默认)
-a	显示所有文件系统的磁盘使用情况
-h	以 1024 进制计算最合适的单位显示磁盘容量
-H	以 1000 进制计算最合适的单位显示磁盘容量
-T	显示磁盘分区类型
-t	显示指定类型文件系统的磁盘分区
-x	不显示指定类型文件系统的磁盘分区

1.4.4 命令使用

(1) 显示本地文件系统的磁盘使用情况

命令: df -l

```
(root@kali)~# df -l
文件系统      1K-块    已用    可用  已用% 挂载点
udev          962424         0  962424    0% /dev
tmpfs         199840     1104   198736    1% /run
/dev/sda1     19480400 11036752  7428764   60% /
tmpfs         999192         0   999192    0% /dev/shm
tmpfs          5120         0    5120    0% /run/lock
tmpfs         199836         64   199772    1% /run/user/1000
```

(2) 显示本地文件系统的磁盘使用情况并以 1024 进制计算磁盘容量

命令: df -h

```
(root@kali)~# df -h
文件系统      容量  已用  可用  已用% 挂载点
udev          940M         0   940M    0% /dev
tmpfs         196M   1.1M   195M    1% /run
/dev/sda1     19G   11G   7.1G   60% /
tmpfs         976M         0   976M    0% /dev/shm
tmpfs         5.0M         0   5.0M    0% /run/lock
tmpfs         196M   64K   196M    1% /run/user/1000
```

(3) 显示本地文件系统的磁盘使用情况并以 1000 进制计算磁盘容量

命令: df -H



```
(root@kali)~# df -H
文件系统      容量  已用  可用  已用% 挂载点
udev           986M    0  986M    0% /dev
tmpfs          205M  1.2M  204M    1% /run
/dev/sda1       20G   12G   7.7G   60% /
tmpfs          1.1G    0  1.1G    0% /dev/shm
tmpfs          5.3M    0  5.3M    0% /run/lock
tmpfs          205M   66k  205M    1% /run/user/1000
```

(4) 显示所有文件系统的磁盘使用情况

命令: df -a

```
(root@kali)~# df -a
文件系统      1K-块  已用  可用  已用% 挂载点
sysfs         0      0      0    - /sys
proc          0      0      0    - /proc
udev        962424    0  962424    0% /dev
devpts        0      0      0    - /dev/pts
tmpfs        199840   1104  198736    1% /run
/dev/sda1    19480400 11036800 7428716   60% /
securityfs   0      0      0    - /sys/kernel/security
tmpfs        999192    0  999192    0% /dev/shm
tmpfs        5120     0   5120    0% /run/lock
cgroup2       0      0      0    - /sys/fs/cgroup
pstore        0      0      0    - /sys/fs/pstore
none          0      0      0    - /sys/fs/bpf
systemd-1     -      -      -    - /proc/sys/fs/binfmt_misc
mqueue        0      0      0    - /dev/mqueue
hugetlbfs     0      0      0    - /dev/hugepages
debugfs       0      0      0    - /sys/kernel/debug
tracefs       0      0      0    - /sys/kernel/tracing
sunrpc        0      0      0    - /run/rpc_pipefs
configfs      0      0      0    - /sys/kernel/config
fusectl       0      0      0    - /sys/fs/fuse/connections
none          0      0      0    - /run/credentials/systemd-sysusers.service
vmware-vmblock 0      0      0    - /run/vmblock-fuse
binfmt_misc   0      0      0    - /proc/sys/fs/binfmt_misc
tmpfs        199836    64  199772    1% /run/user/1000
gvfsd-fuse    0      0      0    - /run/user/1000/gvfs
```

(5) 显示本地文件系统的磁盘使用情况以及磁盘分区的类型

命令: df -T

```
(root@kali)~# df -T
文件系统      类型      1K-块  已用  可用  已用% 挂载点
udev          devtmpfs  962424    0  962424    0% /dev
tmpfs         tmpfs     199840   1104  198736    1% /run
/dev/sda1     ext4      19480400 11036824 7428692   60% /
tmpfs         tmpfs     999192    0  999192    0% /dev/shm
tmpfs         tmpfs     5120     0   5120    0% /run/lock
tmpfs         tmpfs     199836    64  199772    1% /run/user/1000
```

(6) 显示本地文件系统中磁盘分区类型为 tmpfs 的磁盘使用情况

命令: df -t tmpfs



```
(root@kali)~# df -t tmpfs
```

文件系统	1K-块	已用	可用	已用%	挂载点
tmpfs	199840	1104	198736	1%	/run
tmpfs	999192	0	999192	0%	/dev/shm
tmpfs	5120	0	5120	0%	/run/lock
tmpfs	199836	64	199772	1%	/run/user/1000

(7) 显示本地文件系统中除磁盘分区类型为 tmpfs 以外的磁盘使用情况
命令: df -x tmpfs

```
(root@kali)~# df -x tmpfs
```

文件系统	1K-块	已用	可用	已用%	挂载点
udev	962424	0	962424	0%	/dev
/dev/sda1	19480400	11036852	7428664	60%	/



1.5 find 命令

1.5.1 命令功能

在指定目录下查找文件（从指定目录开始搜索）。

1.5.2 命令格式

find [搜索路径] [选项] [文件名]

1.5.3 命令参数

参数	作用
-amin<分钟>	查找在指定时间曾被存取过的文件或目录，单位以分钟计算
-anewer<参考文件或目录>	查找其存取时间较指定文件或目录的存取时间更接近现在的文件或目录
-atime<24 小时数>	查找在指定时间曾被存取过的文件或目录，单位以 24 小时计算
-cmin<分钟>	查找在指定时间之时被更改过的文件或目录
-cnewer<参考文件或目录>	查找其更改时间较指定文件或目录的更改时间更接近现在的文件或目录
-ctime<24 小时数>	查找在指定时间之时被更改的文件或目录，单位以 24 小时计算
-daystart	从本日开始计算时间
-depth	从指定目录下最深层的子目录开始查找
-expty	寻找文件大小为 0 Byte 的文件，或目录下没有任何子目录或文件的空目录
-exec<执行指令>	假设 find 指令的回传值为 True，就执行该指令
-false	将 find 指令的回传值皆设为 False
-fls<列表文件>	此参数的效果和指定“-ls”参数类似，但会把结果保存为指定的列表文件
-follow	排除符号连接
-fprint<列表文件>	此参数的效果和指定“-print”参数类似，但会把结果保存成指定的列表文件
-fprint0<列表文件>	此参数的效果和指定“-print0”参数类似，但会把结果保存成指定的列表文件
-fprintf<列表文件><输出格式>	此参数的效果和指定“-printf”参数类似，但会把结果保存成指定的列表文件
-fstype<文件系统类型>	只寻找该文件系统类型下的文件或目录
-gid<群组识别码>	查找符合指定之群组识别码的文件或目录
-group<群组名称>	查找符合指定之群组名称的文件或目录
-ilname<范本样式>	此参数的效果和指定“-lname”参数类似，但忽略字符大小写的差别
-iname<范本样式>	此参数的效果和指定“-name”参数类似，但忽略字符大小写的差别
-inum<inode 编号>	查找符合指定的 inode 编号的文件或目录
-ipath<范本样式>	此参数的效果和指定“-path”参数类似，但忽略字符大小写的差别
-iregex<范本样式>	此参数的效果和指定“-regexe”参数类似，但忽略字符大小写的差别
-links<连接数目>	查找符合指定的硬连接数目的文件或目录
-lname<范本样式>	指定字符串作为寻找符号连接的范本样式



-ls	假设 find 指令的回传值为 True，就将文件或目录名称列出到标准输出
-maxdepth<目录层级>	设置最大目录层级
-mindepth<目录层级>	设置最小目录层级
-mmin<分钟>	查找在指定时间曾被更改过的文件或目录，单位以分钟计算
-mount	此参数的效果和指定“-xdev”相同
-mtime<24 小时数>	查找在指定时间曾被更改过的文件或目录，单位以 24 小时计算
-name<范本样式>	指定字符串作为寻找文件或目录的范本样式
-newer<参考文件或目录>	查找其更改时间较指定文件或目录的更改时间更接近现在的文件或目录
-nogroup	找出不属于本地主机群组识别码的文件或目录
-noleaf	不去考虑目录至少需拥有两个硬连接存在
-nouser	找出不属于本地主机用户识别码的文件或目录
-ok<执行指令>	此参数的效果和指定“-exec”类似，但在执行指令之前会先询问用户，若回答“y”或“Y”，则放弃执行命令
-path<范本样式>	指定字符串作为寻找目录的范本样式
-perm<权限数值>	查找符合指定的权限数值的文件或目录
-print	假设 find 指令的回传值为 True，就将文件或目录名称列出到标准输出。格式为每列一个名称，每个名称前皆有“./”字符串
-print0	假设 find 指令的回传值为 True，就将文件或目录名称列出到标准输出。格式为全部的名称皆在同一行
-printf<输出格式>	假设 find 指令的回传值为 True，就将文件或目录名称列出到标准输出。格式可以自行指定
-prune	不寻找字符串作为寻找文件或目录的范本样式
-regex<范本样式>	指定字符串作为寻找文件或目录的范本样式
-size<文件大小>	查找符合指定的文件大小的文件
-true	将 find 指令的回传值皆设为 True
-typ<文件类型>	只寻找符合指定的文件类型的文件
-uid<用户识别码>	查找符合指定的用户识别码的文件或目录
-used<日数>	查找文件或目录被更改之后在指定时间曾被存取过的文件或目录，单位以日计算
-user<拥有者名称>	查找符合指定的拥有者名称的文件或目录
-xdev	将范围局限在先行的文件系统中
-xtype<文件类型>	此参数的效果和指定“-type”参数类似，差别在于它针对符号连接检查



1.5.4 命令使用

(1) 显示当前目录下所有文件及文件夹

命令: `find .`

```
(kali㉿kali)-[~/桌面]
$ find .
.
./computer.c
./serve.txt
./cash.c
./a.c
./test.sh.swo
./client
./client.txt
./aaa.sh
./test2.sh
./cumt.txt
./thread.c
./test1.sh
./serve.c
./thread
./aaa.swp
./cumt
./serve
./computer
./test.sh.swp
./test.sh
```

(2) 查找当前目录下以.txt 结尾的文件

命令: `find -name "*.txt"`

```
(kali㉿kali)-[~/桌面]
$ find -name "*.txt"
./serve.txt
./client.txt
./cumt.txt
```

(3) 查找当前目录下以.txt 或.sh 结尾的文件

命令: `find -name "*.txt" -o -name "*.sh"`

```
(kali㉿kali)-[~/桌面]
$ find -name "*.txt" -o -name "*.sh"
./serve.txt
./client.txt
./aaa.sh
./test2.sh
./cumt.txt
./test1.sh
./test.sh
```

(4) 查找当前目录下名称为 cumt 的文件

命令: `find -name cumt`

```
(kali㉿kali)-[~/桌面]
$ find -name cumt
./cumt
```



(5) 查找当前目录下的普通文件

命令: `find -type f`

```
(kali㉿kali)-[~/桌面]
$ find -type f
./computer.c
./serve.txt
./cash.c
./a.c
./test.sh.swo
./client
./client.txt
./aaa.sh
./test2.sh
./cumt.txt
./thread.c
./test1.sh
./serve.c
./thread
./aaa.swp
./cumt
./serve
./computer
./test.sh.swp
./test.sh
```

(6) 查找当前目录及其子目录下权限为 777 的文件

命令: `find -perm 777`

```
(kali㉿kali)-[~/桌面]
$ find -perm 777
./computer.c
./serve.txt
./client.txt
./aaa.sh
```

(7) 查找当前目录及其子目录下过去 3 天内被访问过的文件

命令: `find -atime 3`

```
(root㉿kali)-[/]
# find -atime 3
./usr/lib/jvm/java-11-openjdk-amd64/man/man1/jjs.1.gz
./usr/lib/jvm/java-11-openjdk-amd64/man/man1/unpack200.1.gz
./usr/lib/jvm/java-11-openjdk-amd64/man/man1/rmiregistry.1.gz
./usr/lib/jvm/java-11-openjdk-amd64/man/man1/java.1.gz
./usr/lib/jvm/java-11-openjdk-amd64/man/man1/rmid.1.gz
./usr/lib/jvm/java-11-openjdk-amd64/man/man1/pack200.1.gz
./usr/lib/jvm/java-11-openjdk-amd64/man/man1/keytool.1.gz
./usr/lib/python3/dist-packages/__pycache__/brotli.cpython-39.pyc
./usr/lib/python3/dist-packages/__pycache__/six.cpython-39.pyc
./usr/lib/python3/dist-packages/__pycache__/socks.cpython-39.pyc
```



(8) 查找当前目录及其子目录下大小为 1b 的文件

命令: `find -size 1b`

```
(kali㉿kali)-[~/桌面]
$ find -size 1b
./cash.c
./a.c
./aaa.sh
./cumt
```

(9) 查找/etc/目录下, 大于 40KB 且小于 80KB 的文件

命令: `find /etc -size +40k -a -size -80k`

```
(root㉿kali)-[/]
$ find /etc -size +40k -a -size -80k
/etc/ld.so.cache
/etc/mime.types
/etc/inetd.conf
/etc/guymager/guymager.cfg
/etc/php/7.4/apache2/php.ini
/etc/php/7.4/cli/php.ini
/etc/java-11-openjdk/security/java.security
```

(10) 查找出当前目录下权限不是 644 的 sh 文件

```
(kali㉿kali)-[~/桌面]
$ find . -type f -name "*.sh" ! -perm 644
./aaa.sh
./test2.sh
./test1.sh
./test.sh
```

(11) 查找当前目录某个用户组拥有的所有文件

命令: `find -type f -group kali`

```
(kali㉿kali)-[~/桌面]
$ find -type f -group kali
./computer.c
./serve.txt
./cash.c
./a.c
./test.sh.swo
./client
./client.txt
./aaa.sh
./test2.sh
./cumt.txt
./thread.c
./test1.sh
./serve.c
./thread
./aaa.swp
./cumt
./serve
./computer
./test.sh.swp
./test.sh
```




(12) 删除当前目录下所有.txt 文件

命令: `find -type f -name "*.txt" -delete`

```
(kali㉿kali)-[~/桌面/text]
$ ls
one.txt  three.sh  two.txt

(kali㉿kali)-[~/桌面/text]
$ find -type f -name "*.txt" -delete

(kali㉿kali)-[~/桌面/text]
$ ls
three.sh

(kali㉿kali)-[~/桌面/text]
$
```

(13) 将当前目录下所有属于 root 的文件的所有权改为 kali

命令: `find -type f -user root -exec chown kali {} \;`

```
(root㉿kali)-[/home/kali/桌面/text]
# ls -l
总用量 0
-rw-r--r-- 1 root root 0  4月 12 21:07 one.txt
-rw-r--r-- 1 kali kali 0  4月 12 20:59 three.sh
-rw-r--r-- 1 root root 0  4月 12 21:07 two.sh

(root㉿kali)-[/home/kali/桌面/text]
# find -type f -user root -exec chown kali {} \;

(root㉿kali)-[/home/kali/桌面/text]
# ls -l
总用量 0
-rw-r--r-- 1 kali root 0  4月 12 21:07 one.txt
-rw-r--r-- 1 kali kali 0  4月 12 20:59 three.sh
-rw-r--r-- 1 kali root 0  4月 12 21:07 two.sh
```



1.6 du 命令

1.6.1 命令功能

统计磁盘上的文件大小，并显示出来。

1.6.2 命令格式

du [选项] [文件]

1.6.3 命令参数及作用

参数	作用
-b	以 byte 为单位统计文件
-k	以 kb 为单位统计文件
-m	以 mb 为单位统计文件
-h	按照 1024 进制以最合适的单位统计文件
-H	按照 1000 进制以最合适的单位统计文件
-s	指定统计目标

1.6.4 命令使用

(1) 显示当前目录下所有文件的大小

命令：du

```
(root@kali)-[/tmp]
# du
4  ./ICE-unix
4  ./runtime-root
4  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-ModemManager.service-jcHNQF/tmp
8  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-ModemManager.service-jcHNQF
4  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-colord.service-YtXNZD/tmp
8  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-colord.service-YtXNZD
4  ./VMwareDnD
4  ./font-unix
4  ./X11-unix
4  ./XIM-unix
4  ./ssh-9ADptm1nHMEo
4  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-haveged.service-0nJLnV/tmp
8  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-haveged.service-0nJLnV
4  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-systemd-logind.service-8PSUUE/tmp
8  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-systemd-logind.service-8PSUUE
4  ./vmware-root_387-1815350242
4  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-upower.service-skhykh/tmp
8  ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-upower.service-skhykh
84  .
```

(2) 显示文件 bus-FUDVNWERaF 的大小

命令：du -s bus-FUDVNWERaF

```
(root@kali)-[/tmp]
# du -s dbus-FUDVNWERaF
0  dbus-FUDVNWERaF
```

(3) 以 kb 为单位显示当前目录下所有文件的大小

命令：du -k



```
(root@kali)-[/tmp]
# du -k
4      ./ICE-unix
4      ./runtime-root
4      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-ModemManager.service-jcHNQF/tmp
8      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-ModemManager.service-jcHNQF
4      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-colord.service-YtXNZD/tmp
8      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-colord.service-YtXNZD
4      ./VMwareDnD
4      ./font-unix
4      ./X11-unix
4      ./XIM-unix
4      ./ssh-9ADptm1nHMEo
4      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-haveged.service-0nJLnV/tmp
8      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-haveged.service-0nJLnV
4      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-systemd-logind.service-8PSUUE/tmp
8      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-systemd-logind.service-8PSUUE
4      ./vmware-root_387-1815350242
4      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-upower.service-skhykh/tmp
8      ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-upower.service-skhykh
84      .
```

(4) 按照 1024 进制以最合适的单位统计文件大小并显示出来
命令: du -h

```
(root@kali)-[/tmp]
# du -h
4.0K    ./ICE-unix
4.0K    ./runtime-root
4.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-ModemManager.service-jcHNQF/tmp
8.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-ModemManager.service-jcHNQF
4.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-colord.service-YtXNZD/tmp
8.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-colord.service-YtXNZD
4.0K    ./VMwareDnD
4.0K    ./font-unix
4.0K    ./X11-unix
4.0K    ./XIM-unix
4.0K    ./ssh-9ADptm1nHMEo
4.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-haveged.service-0nJLnV/tmp
8.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-haveged.service-0nJLnV
4.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-systemd-logind.service-8PSUUE/tmp
8.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-systemd-logind.service-8PSUUE
4.0K    ./vmware-root_387-1815350242
4.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-upower.service-skhykh/tmp
8.0K    ./systemd-private-3a3ae5f8e23d48bb86f35e774e91dbd8-upower.service-skhykh
84K     .
```



1.7 grep 命令

1.7.1 命令功能

搜索文件中含有指定字符串的行，并显示出来。

1.7.2 命令格式

grep [选项] 字符串 文件名

1.7.3 命令参数及作用

参数	作用
-a	不要忽略二进制的文件
-A	除了显示符合范本样式的那一行之外，还显示该行之后的内容
-b	在显示符合样式的那一行之前，标示出该行第一个字符的编号
-B	除了显示符合样式的那一行之外，还显示该行之前的内容
-c	计算符合样式的列数
-C	除了显示符合样式的那一行之外，还显示该行之前的内容
-d	当指定要查找的是目录时，必须使用这项参数，否则 grep 指令将回报信息并停止动作
-e	指定字符串做为查找文件内容的样式
-E	将样式为延伸的正则表达式来使用
-f	指定规则文件，其内容含有一个或多个规则样式，让 grep 查找符合规则条件的文件内容，格式为每行一个规则样式
-F	将样式视为固定字符串的列表
-G	将样式视为普通的表示法来使用
-h	在显示符合样式的那一行之前，不标示该行所属的文件名称
-H	在显示符合样式的那一行之前，表示该行所属的文件名称
-i	忽略字符大小写的差别
-l	列出文件内容符合指定的样式的文件名称
-L	列出文件内容不符合指定的样式的文件名称
-n	在显示符合样式的那一行之前，标示出该行的列数编号
-o	只显示匹配 PATTERN 部分
-q	不显示任何信息
-r	此参数的效果和指定"-d recurse"参数相同
-s	不显示错误信息
-v	显示不包含匹配文本的所有行
-w	只显示全字符合的列
-x	只显示全列符合的列



1.7.4 命令使用

(1) 查询 serve.c 文件中的关键字 perror 并显示包含该关键字的行的内容

命令: `grep perror serve.c`

```
(kali㉿kali)-[~/桌面]
$ grep perror serve.c
    perror("服务器 socket 创建失败!");
    perror("服务器绑定失败!");
    perror("服务器监听失败!");
    perror("服务器连接客户端失败!");
        perror("信息发送失败!");
            perror("文件传送失败!");
```

(2) 查询 serve.c 文件中的关键字 perror 并显示包含该关键字的行的内容及行号

命令: `grep -n perror serve.c`

```
(kali㉿kali)-[~/桌面]
$ grep -n perror serve.c
22:     perror("服务器 socket 创建失败!");
33:     perror("服务器绑定失败!");
39:     perror("服务器监听失败!");
46:     perror("服务器连接客户端失败!");
64:         perror("信息发送失败!");
82:             perror("文件传送失败!");
```

(3) 统计 serve.c 文件中包含关键字 perror 的行数并显示出来

命令: `grep -c perror serve.c`

```
(kali㉿kali)-[~/桌面]
$ grep -c perror serve.c
6
```

(4) 显示 serve.c 文件中包含关键字 perror 的行及下一行的内容

命令: `grep -A 1 perror serve.c`

```
(kali㉿kali)-[~/桌面]
$ grep -A 1 perror serve.c
    perror("服务器 socket 创建失败!");
    exit(1);
—
    perror("服务器绑定失败!");
    exit(1);
—
    perror("服务器监听失败!");
    exit(1);
—
    perror("服务器连接客户端失败!");
    exit(1);
—
        perror("信息发送失败!");
    }
—
            perror("文件传送失败!");
            send(clientfd,"error",20,0);
```



(5) 显示 serve.c 文件中包含关键字 perror 的行及上一行的内容

命令: `grep -B 1 perror serve.c`

```
(kali㉿kali)-[~/桌面]
└─$ grep -B 1 perror serve.c
{
    perror("服务器 socket 创建失败!");
--
{
    perror("服务器绑定失败!");
--
{
    perror("服务器监听失败!");
--
{
    perror("服务器连接客户端失败!");
--
        {
            perror("信息发送失败!");
--
                usleep(3000000);
                perror("文件传送失败!");
```

(6) 显示 serve.c 文件中包含关键字 perror 的行及上一行和下一行的内容

命令: `grep -C 1 perror serve.c`

```
(kali㉿kali)-[~/桌面]
└─$ grep -C 1 perror serve.c
{
    perror("服务器 socket 创建失败!");
    exit(1);
--
{
    perror("服务器绑定失败!");
    exit(1);
--
{
    perror("服务器监听失败!");
    exit(1);
--
{
    perror("服务器连接客户端失败!");
    exit(1);
--
        {
            perror("信息发送失败!");
--
        }
--
        usleep(3000000);
        perror("文件传送失败!");
        send(clientfd,"error",20,0);
```

(7) 查询 serve.c, computer.c, cumt.txt 三个文件中哪几个文件包含关键字 perror

命令: `grep -l perror serve.c computer.c cumt.txt`

```
(kali㉿kali)-[~/桌面]
└─$ grep -l perror serve.c computer.c cumt.txt
serve.c
computer.c
```



(8) 连续在 serve.c 文件中匹配多个关键词 (include 和 printf)

命令: `grep -e include -e printf serve.c`

```
(kali@kali)-[~/桌面]
$ grep -e include -e printf serve.c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <sys/prctl.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>
#include <sys/time.h>
#include <ctype.h>
#include <sys/stat.h>
#include <fcntl.h>
printf("服务器启动中.....\n");
printf("服务器已正常启动!\n");
printf("用户%s已成功登录服务器, 可以开始交互!\n", username);
printf("请输入完整的文件名:");
printf("正在向用户%s传送文件.....\n", username);
printf("文件传送成功!\n");
printf("%s: %s\n", username, my_receive); //将接收到的信息打印出来
printf("用户%s已退出服务器\n", username);
printf("正在接收用户%s传送的文件.....\n", username);
printf("文件接收成功!\n");
printf("文件接收失败!\n");
```

(9) 在文件 serve.c 文件中忽略大小写匹配关键字 perror

命令: `grep -i perror serve.c`

```
(kali@kali)-[~/桌面]
$ grep -i perror serve.c
PERROR("服务器 socket 创建失败!");
perror("服务器绑定失败!");
perror("服务器监听失败!");
PERROR("服务器连接客户端失败!");
perror("信息发送失败!");
perror("文件传送失败!");
```

(10) 显示 serve.c 文件中仅含有关键字 perror 的行

命令: `grep -x perror serve.c`

```
(kali@kali)-[~/桌面]
$ grep -x perror serve.c
```

(11) 利用正则表达式, 输出匹配到的部分

命令: `echo cumt 2019 | grep -E -o "[a-z]+"`

```
(kali@kali)-[~/桌面]
$ echo cumt 2019 | grep -E -o "[a-z]+"
cumt
```

(12) 查询带有关键字 pycharm 的进程并显示出来

命令: `ps -ef | grep pycharm`

```
(root@kali)-[/tmp]
$ ps -ef | grep pycharm
root      5090      4812    0 23:43 pts/1    00:00:00 grep --color=auto pycharm
```



1.8 管道符综合使用

1.8.1 管道符简介

“|”是Linux管道命令操作符，简称管道符，使用此符号可以将多个命令连接起来。其中第一条命令的结果会作为第二条命令的参数，第二条命令的结果会作为第三条命令的参数，以此类推。

1.8.2 使用实例

(1) 查看 serve.c 文件是否存在于当前目录下

命令：ls -l | grep serve.c

```
(kali㉿kali)-[~/桌面]
$ ls -l | grep serve.c
-rw-r--r-- 1 kali kali 5520 4月 12 11:28 serve.c
```

(2) 统计/etc 目录下的文件总数

命令：ls /etc | wc -l

```
(kali㉿kali)-[~/桌面]
$ ls /etc | wc -l
259
```

(3) 匹配 serve.c 文件中的关键词 perror

命令：cat serve.c | grep "perror"

```
(kali㉿kali)-[~/桌面]
$ cat serve.c | grep "perror"
perror("服务器 socket 创建失败！");
perror("服务器绑定失败！");
perror("服务器监听失败！");
perror("服务器连接客户端失败！");
                perror("信息发送失败！");
                        perror("文件传送失败！");
```

(4) 直接删除带有保护的文件

命令：echo y | rm hello

```
(kali㉿kali)-[~/桌面]
$ rm hello
rm: 是否删除有写保护的普通空文件 'hello'?

(kali㉿kali)-[~/桌面]
$ echo y | rm hello
```

(5) 统计当前目录的子目录数

命令：ls -l | cut -c 1 | grep "d" | wc -l

```
(kali㉿kali)-[/]
$ ls -l | cut -c 1 | grep "d" | wc -l
17
```

(6) 查看当前目录下倒数第二个文件的详细信息

命令：ls -l | tail -n 2 | head -n 1



```
(kali㉿kali)-[~/桌面]
$ ls -l
总用量 152
-rwxrwxrwx 1 kali kali 336 4月 5 22:19 aaa.sh
-rw-r--r-- 1 kali kali 402 4月 8 11:44 a.c
-rw-r--r-- 1 kali kali 139 4月 9 21:55 cash.c
-rwxr-xr-x 1 kali kali 17096 4月 14 14:03 client
-rwxrwxrwx 1 kali kali 800 4月 14 14:22 client.txt
-rwxr-xr-x 1 kali kali 17048 4月 8 16:19 computer
-rwxrwxrwx 1 kali kali 5933 4月 14 14:02 computer.c
-rw-r--r-- 1 kali kali 72 4月 5 00:09 cumt
-rw-r--r-- 1 kali kali 10 4月 14 13:38 cumt.txt
-rw-r--r-- 1 root root 0 4月 12 21:06 id.txt
-rw-r--r-- 1 kali kali 6 4月 13 21:15 kali.txt
-rwxr-xr-x 1 kali kali 17192 4月 14 14:03 serve
-rw-r--r-- 1 kali kali 5823 4月 14 14:03 serve.c
-rwxrwxrwx 1 kali kali 800 4月 14 14:24 serve.txt
-rwxr-xr-x 1 kali kali 1324 4月 6 09:26 test1.sh
-rwxr--r-- 1 kali kali 835 4月 6 08:56 test2.sh
-rwxr-xr-x 1 kali kali 1038 4月 5 21:40 test.sh
drwxr-xr-x 2 kali kali 4096 4月 12 21:07 text
-rwxr-xr-x 1 kali kali 16568 4月 9 21:57 thread
-rw-r--r-- 1 kali kali 2758 4月 9 21:57 thread.c
-rwxrwxrwx 1 kali kali 2425 4月 14 10:26 v.sh

(kali㉿kali)-[~/桌面]
$ ls -l | tail -n 2 | head -n 1
-rw-r--r-- 1 kali kali 2758 4月 9 21:57 thread.c
```

(7) 将 a.txt 和 b.txt 两个文件的内容连接起来并输出

命令: cat a.txt | paste -d: b.txt -

```
(kali㉿kali)-[~/桌面]
$ cat a.txt
cumt

(kali㉿kali)-[~/桌面]
$ cat b.txt
2019

(kali㉿kali)-[~/桌面]
$ cat a.txt | paste -d: b.txt -
2019:cumt
```

1.9 体会

在本次实验中,我测试并总结了包括 grep, ls, find 命令等多个命令的功能及使用方法,使我对 linux 命令系统有了一个基本的认识,也使我对 linux 系统的操作能力有了一定程度上的提升。值得注意的是,由于 linux 系统下的命令和 windows 系统下的命令大致相似却又有细微的区别,因此在实验初期总是会出现习惯性的输入错误,但随着实验的进行,我逐渐的习惯了 linux 系统命令的书写方式。

在本次实验中,我还多次尝试使用管道符来完成组合命令的使用,这使得我对 linux 系统下管道符的使用有了一定的了解,同时让我清醒的认识到,系统命令的使用灵活多变,可以通过许多辅助的功能符号进行各种组合,从而实现某个复杂的功能。



1.10 参考文献

- [1] 菜鸟教程. Linux 命令大全. <https://www.runoob.com/linux/linux-command-manual.html>.
- [2] 鸟哥. 鸟哥的 Linux 私房菜: 基础学习篇[M]. 人民邮电出版社, 2010.
- [3] Robert Love. LINUX 系统编程. 东南大学出版社, 2009.



2 shell 编程

- 要求：
- (1) 知道如何执行 shell 程序。
 - (2) 在 shell 脚本中要体现条件控制（如 if 结构和条件分支）且不少于 4 个。
 - (3) 在 shell 脚本中要体现循环（for, while 和 until 循环），循环嵌套不少于 1 层。
 - (4) 掌握 shell 程序的调试。
 - (5) shell 程序要阐明功能和作用，程序中有必要的注释，并将运行过程和结果截图附上。
 - (6) 程序的有效语句不少于 20 行(符合编程规范)。

说明：如果撰写规范不符合《计算机学院考查类课程报告撰写规范》要求的，整体上酌情扣除 1-10 分。

2.1 最大公因数及最小公倍数求解算法

2.1.1 功能和作用

此程序主要利用扩展欧几里得算法实现对任意两个正整数最大公因数及最小公倍数的求解。

2.1.2 源代码

```
#!/bin/bash

read -p "请输入第一个数字: " num1 #获取用户输入的两个数字
read -p "请输入第二个数字: " num2
ans1=$num1 #对用户输入的数字备份
ans2=$num2

temp=1
until [ $temp -eq 0 ] #只要余数不为零则继续循环:num1=k*num2+temp
do
    if [ $num1 -lt $num2 ] #保证永远是大数模小数
    then
        tmp=$num1
        num1=$num2
        num2=$tmp
    fi
    temp=`expr $num1 % $num2` #求当前余数
    if [ $temp -ne 0 ] #余数不为零则需要进行下一轮运算.此处做预处理
    then
        num1=$num2
        num2=$temp
    fi
done
max=$num2 #余数为零时跳出循环，最后一个非零余数即为最大公因数
sum=`expr $ans1 \* $ans2` #利用公式通过最大公因数求解最小公倍数
```



```
min=`expr $sum / $max`  
echo "$ans1 和$ans2 的最大公因数是: $max"  
echo "$ans1 和$ans2 的最小公倍数是: $min"
```

2.1.3 运行截图

(1) 求解 36 和 16 的最大公因数及最小公倍数

```
(kali㉿kali)-[~/桌面]  
$ ./test2.sh  
请输入第一个数字: 36  
请输入第二个数字: 16  
36和16的最大公因数是: 4  
36和16的最小公倍数是: 144
```

(2) 求解 37 和 159 的最大公因数及最小公倍数

```
(kali㉿kali)-[~/桌面]  
$ ./test2.sh  
请输入第一个数字: 37  
请输入第二个数字: 159  
37和159的最大公因数是: 1  
37和159的最小公倍数是: 5883
```

(3) 求解 3658 和 8965 的最大公因数及最小公倍数

```
(kali㉿kali)-[~/桌面]  
$ ./test2.sh  
请输入第一个数字: 3658  
请输入第二个数字: 8965  
3658和8965的最大公因数是: 1  
3658和8965的最小公倍数是: 32793970
```



2.2 选择排序

2.2.1 功能和作用

此程序依托选择排序算法对用户输入的多个数字进行排序，并将排序结果打印出来。用户可以通过输入参数选择降序排列还是升序排列。

2.2.2 源代码

```
#!/bin/bash
read -p "请输入待排序的数字: " -a num #接收待排序数字
read -p "请选择排序的类型（输入 1 代表升序排序，输入 2 代表降序排序）: " option #接收用户指定排序类型的参数
count=0
len=${#num[@]} #获取待排序数字个数
if [ $option -eq 1 ] #升序排序
then
    for((i=0;i<$len;i++)) #进行 len 轮循环，选出 len 个当前最小数字，顺次写入新的数组
    do
        min=999999999
        for((j=0;j<$len;j++)) #找出当前最小元素并保存其值和下标
        do
            if [ ${num[$j]} -lt $min ]
            then
                min=${num[$j]}
                count=$j
            fi
        done
        num[$count]=999999999 #将数组中当前最小数字赋值 999999999
        ans[$i]=$min #将本轮选出的最小数字添加进数组
    done
    echo "sorting result: ${ans[*]}" #打印出排序结果
elif [ $option -eq 2 ]
then
    for((i=0;i<$len;i++)) #进行 len 轮循环，选出 len 个当前最大数字，顺次写入新的数组
    do
        max=-999999999
        for((j=0;j<$len;j++)) #找出当前最大元素并保存其值和下标
```



```
do
    if [ ${num[$j]} -gt $max ]
    then
        max=${num[$j]}
        count=$j
    fi
done
num[$count]=-999999999 #将数组中当前最大数字赋值-999999999
ans[$i]=$max #将本轮选出的最大数字添加进数组
done
echo "排序结果: ${ans[*]}" #打印出排序结果
fi
```

2.2.3 运行截图

(1) 对数组[5, 8, 65, 85, 35, 568, 210, 65, 84, 12, 33, 96]进行升序排序

```
(kali㉿kali)-[~/桌面]
$ ./test1.sh
请输入待排序的数字: 5 8 65 85 35 568 210 65 84 12 33 96
请选择排序的类型 (输入1代表升序排序, 输入2代表降序排序): 1
sorting result: 5 8 12 33 35 65 65 84 85 96 210 568
```

(2) 对数组[5, 8, 65, 85, 35, 568, 210, 65, 84, 12, 33, 96]进行降序排序

```
(kali㉿kali)-[~/桌面]
$ ./test1.sh
请输入待排序的数字: 5 8 65 85 35 568 210 65 84 12 33 96
请选择排序的类型 (输入1代表升序排序, 输入2代表降序排序): 2
排序结果: 568 210 96 85 84 65 65 35 33 12 8 5
```




2.3 进制转换

2.3.1 功能及作用

此程序可以将用户输入的二进制，八进制以及十六进制的任意正整数转换成十进制数字，并将转换结果打印出来。

2.3.2 源代码

```
#!/bin/bash
read -p "请输入需要转换的数字: " num #接收用户输入的需要转换的数字
read -p "请输入该数字当前的进制: " radix #接收该数字当前的进制
len=${#num} #获取该数字的位数
count=0
ans=0
if [ $radix -eq 2 -o $radix -eq 8 ] #如果是二进制数或八进制数则进入该分支
then
    for((i=0;i<$len;i++)) #利用公式 $num = a_0 \times r^0 + a_1 \times r^1 + \dots + a_n \times r^n$ 对每一位进行运算
    do
        temp=`expr $num % 10` #取出数字的第 i 位(最低位为第 0 位)
        num=`expr $num / 10`
        for((j=0;j<i;j++)) #利用公式 $a_i \times r^i$ 进行计算
        do
            temp=`expr $temp \* $radix`
        done
        ans=`expr $temp + $ans` #将各位计算结果相加，得到最终结果
    done
    echo "转换成十进制的结果是: $ans" #将转换结果打印出来
elif [ $radix==16 ] #如果是 16 进制则进入该分支
then
    for((i=$len;i>=1;i--)) #利用公式 $num = a_0 \times 16^0 + a_1 \times 16^1 + \dots + a_n \times 16^n$ 对每一位进行运算
    do
        temp=${num: `expr $i - 1` : 1} #取出数字的第 i 位
        if [ "$temp" = "a" -o "$temp" = "A" ] #如果第 i 位是字母则进行相应转换
        then
            temp=10
        elif [ "$temp" = "b" -o "$temp" = "B" ]
        then
```



```

temp=11
elif [ "$temp" = "c" -o "$temp" = "C" ]
then
temp=12
elif [ "$temp" = "d" -o "$temp" = "D" ]
then
temp=13
elif [ "$temp" = "e" -o "$temp" = "E" ]
then
temp=14
elif [ "$temp" = "f" -o "$temp" = "F" ]
then
temp=15
fi
for((j=0;j<$count;j++)) #利用公式 $a_i \times 16^{count}$ 进行计算
do
temp=`expr $temp \* 16`
done
ans=`expr $ans + $temp` #将各位计算结果相加，得到最终结果
((count++))
done
echo "转换成十进制的结果是: $ans" #将转换结果打印出来
else #如果不是二进制，八进制，十六进制则返回错误
echo "输入的进制不正确！"
fi

```

2.3.3 运行截图

(1) 将二进制数 1011011101010 转换成十进制

```

(kali㉿kali)-[~/桌面]
$ ./test.sh
请输入需要转换的数字: 1011011101010
请输入该数字当前的进制: 2
转换成十进制的结果是: 5866

```



(2) 将八进制数 56325466325 转换成十进制

```
(kali㉿kali)-[~/桌面]  
$ ./test.sh  
请输入需要转换的数字：56325466325  
请输入该数字当前的进制：8  
转换成十进制的结果是：6230011093
```

(3) 将十六进制数 165abced56 转换成十进制

```
(kali㉿kali)-[~/桌面]  
$ ./test.sh  
请输入需要转换的数字：165abced56  
请输入该数字当前的进制：16  
转换成十进制的结果是：96011611478
```



2.4 维基利亚密码

2.4.1 功能及作用

本程序可以根据用户的选择，利用维基利亚密码算法对用户输入的字符串进行加密解密，其中加密时要求用户输入待加密的明文以及用于加密的密钥，解密时要求用户输入待解密的密文以及用于解密的密钥。

2.4.2 源代码

```
#!/bin/bash

read -p "请选择您需要的功能（输入 1 代表加密，输入 2 代表解密）：" option #接收用户选择的功
能

if [ $option -eq 1 ] #加密
then
    read -p "请输入需要加密的明文：" m #接收需要加密的字符串
    read -p "请输入密钥：" k #接收用于加密的密钥
    len_m=${#m} #获取明文及密钥长度
    len_k=${#k}
    for((i=0;i<$len_m;i++)) #将明文字符转换成 ascii 码
    do
        m_temp[i]=`printf "%d" \${m:$i:1}`
    done
    for((i=0;i<$len_k;i++)) #将密钥中的小写字母转换成大写字母并将所有密钥字母转换成 ascii
    do
        k_temp[i]=`printf "%d" \${k:$i:1}`
        if [ ${k_temp[i]} -ge 97 -a ${k_temp[i]} -le 122 ]
        then
            k_temp[i]=`expr $(( ${k_temp[i]} - 32 ))`
        fi
    done
    count=0
    printf "加密后的结果是："
    for((i=0;i<$len_m;i++)) #开始加密
    do
        count=`expr $(( $count % $len_k ))`
        if [ ${m_temp[i]} -ge 97 -a ${m_temp[i]} -le 122 ] #对小写字母进行加密
        then
```



```

m_temp[i]=`expr $(( ${m_temp[i]}+${k_temp[count]}-65 ))`
if [ ${m_temp[i]} -gt 122 ]
then
    m_temp[i]=`expr $(( ${m_temp[i]}-26 ))`
elif [ ${m_temp[i]} -lt 97 ]
then
    m_temp[i]=`expr $(( ${m_temp[i]}+26 ))`
fi
elif [ ${m_temp[i]} -ge 65 -a ${m_temp[i]} -le 90 ] #对大写字母进行加密
then
    m_temp[i]=`expr $(( ${m_temp[i]}+${k_temp[count]}-65 ))`
    if [ ${m_temp[i]} -gt 90 ]
    then
        m_temp[i]=`expr $(( ${m_temp[i]}-26 ))`
    elif [ ${m_temp[i]} -lt 65 ]
    then
        m_temp[i]=`expr $(( ${m_temp[i]}+26 ))`
    fi
fi
t=`printf "%x" ${m_temp[i]}` #将加密结果打印出来
printf "\\x$t"
count=`expr $((count+1))`
done
elif [ $option -eq 2 ] #解密
then
    read -p "请输入需要加密的明文: " c #接收需要解密的字符串
    read -p "请输入密钥: " k #接收用于解密的密钥
    len_c=${#c} #获取密文及密钥的长度
    len_k=${#k}
    for((i=0;i<$len_c;i++)) #将密文字符转换成 ascii 码
    do
        c_temp[i]=`printf "%d" \${c:$i:1}`
    done
    for((i=0;i<$len_k;i++)) #将密钥中的小写字母转换成大写字母并将所有密钥字母转换成 ascii
    do

```




```

k_temp[i]=`printf "%d" \${k:i:1}`
if [ ${k_temp[i]} -ge 97 -a ${k_temp[i]} -le 122 ]
then
    k_temp[i]=`expr $(( ${k_temp[i]} - 32 ))`
fi
done
count=0
printf "解密后的结果是: "
for((i=0;i<$len_c;i++)) #开始解密
do
    count=`expr $(( $count%$len_k ))`
    if [ ${c_temp[i]} -ge 97 -a ${c_temp[i]} -le 122 ] #对小写字母进行解密
    then
        c_temp[i]=`expr $(( ${c_temp[i]} - ${k_temp[count]} + 65 ))`
        if [ ${c_temp[i]} -gt 122 ]
        then
            c_temp[i]=`expr $(( ${c_temp[i]} - 26 ))`
        elif [ ${c_temp[i]} -lt 97 ]
        then
            c_temp[i]=`expr $(( ${c_temp[i]} + 26 ))`
        fi
    elif [ ${c_temp[i]} -ge 65 -a ${c_temp[i]} -le 90 ] #对大写字母进行解密
    then
        c_temp[i]=`expr $(( ${c_temp[i]} - ${k_temp[count]} + 65 ))`
        if [ ${c_temp[i]} -gt 90 ]
        then
            c_temp[i]=`expr $(( ${c_temp[i]} - 26 ))`
        elif [ ${c_temp[i]} -lt 65 ]
        then
            c_temp[i]=`expr $(( ${c_temp[i]} + 26 ))`
        fi
    fi
    t=`printf "%x" ${c_temp[i]}` #将解密结果打印出来
    printf "\\x$t"
    count=`expr $(( $count + 1 ))`

```



done

fi

2.4.3 运行截图

(1) 加密

```
(kali㉿kali)-[~/桌面]
$ ./v.sh
请选择您需要的功能（输入1代表加密，输入2代表解密）：1
请输入需要加密的明文：cumtCUMT
请输入密钥：cumt
加密后的结果是：eoymEOYM
```

(2) 解密

```
(kali㉿kali)-[~/桌面]
$ ./v.sh
请选择您需要的功能（输入1代表加密，输入2代表解密）：2
请输入需要加密的明文：eoymEOYM
请输入密钥：cumt
解密后的结果是：cumtCUMT
```

2.5 体会

在本次实验中，我共编写了四个 shell 脚本，分别是最大最小公因数求解，选择排序，进制转换以及维基利亚密码。在编写的初期，我遇到了各种各样的问题，其中大部分问题都是由于对 shell 编程的语法规则不熟悉造成的。随着实验的进行，我逐渐熟悉这种编程方式，代码的书写速度也逐渐的变快，但是没有实时错误提醒的集成化编程软件确实很不习惯。

通过这四个 shell 脚本的编写，我对 shell 程序编写语言有了一定的了解，也对 shell 程序的调试有了一定的认识。在编写过程中让我感受最深的是，这种编程语言和高级语言既有相同之处，却又各有各的特色，其中最突出的就是 shell 编程更加贴近操作系统，可以直接嵌入 linux 命令。

2.6 参考文献

- [1] 菜鸟教程. Linux Shell 教程. <https://www.runoob.com/linux/linux-shell.html>.
- [2] 鸟哥. 鸟哥的 Linux 私房菜: 基础学习篇[M]. 人民邮电出版社, 2010.
- [3] Robert Love. LINUX 系统编程. 东南大学出版社, 2009.



3 进程控制编程

要求: (1) 掌握进程的创建 fork 系统调用的原理。

(2) 掌握 exec 系统调用的原理。

(3) 掌握 exit 系统调用的原理。

(4) 掌握 wait 系统调用的原理。

(5) 程序代码必须包含以上四个子函数功能, 每个 5 分, 根据阐述的详细和完整程度给出相应的分值。

说明: 如果撰写规范不符合《计算机学院考查类课程报告撰写规范》要求的, 整体上酌情扣除 1-10 分。

3.1 fork 系统调用

3.1.1 原型

```
pid_t fork(void)
```

3.1.2 头文件

```
unistd.h
```

3.1.3 返回值

fork 被调用一次能够返回两次且可能有三种不同的返回值:

- (1) 在父进程中, fork 返回新创建子进程的进程 ID (通常为父进程 PID+1)。
- (2) 在子进程中, fork 返回 0。
- (3) 如果出现错误, fork 返回一个负值。

3.1.4 功能

- (1) 创建一个与原来进程几乎完全相同的进程, 即两个进程可以做完全相同的事, 但如果初始参数或者传入的变量不同, 两个进程也可以做不同的事。
- (2) 一个进程调用 fork 函数后, 系统先给新的进程分配资源, 如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的进程中, 只有少数值与原来的进程的值不同。相当于克隆了一个自己。

3.1.5 底层原理

- (1) Linux 通过 clone() 系统调用实现 fork()。这个调用通过一系列的参数标志来指明父、子进程需要共享的资源。fork()、vfork()、__clone() 库函数都根据各自需要的参数标志去调用 clone(), 然后由 clone() 去调用 do_fork()。
- (2) do_fork() 完成创建中的大部分工作, 其被定义在 kernel/fork.c 文件中, 该函数调用 copy_process() 函数, 让进程开始运行。
- (3) copy_process() 函数工作过程如下:
 - ① 调用 dup_task_struct() 为新进程创建一个内核栈、thread_info 结构和 task_struct, 这些值与当前进程的值相同。此时, 子进程和父进程的描述符完全相同。
 - ② 检查并确保新建这个子进程后, 当前用户所拥有的进程数目没有超出给它分配的资源限制。
 - ③ 子进程着手使自己与父进程区别开来: 进程描述符内的许多成员都要被清零或设为初始化值 (不是继承而来的进程描述符成员, 主要是统计信息)。task_struct 中的大多数数据依然未被修改。
 - ④ 子进程状态被设置为 TASK_UNINTERRUPTIBLE, 以保证它不会投入运行。



- ⑤ copy_process()调用 copy_flags()以更新 task_struct 的 flags 成员。表明进程是否拥有超级用户权限的 PF_SUPERPRIV 标志被清 0。表明进程还没有调用 exec()函数的 PF_FORKNOEXEC 标志被设置。
- ⑥ 调用 alloc_pid()为新进程分配一个有效的 PID。
- ⑦ 根据传递给 clone()的参数标志, copy_process()拷贝或共享打开的文件、文件系统信息、信号处理函数、进程地址空间和命名空间等。在一般情况下, 这些资源会被给定进程的所有线程共享; 否则, 这些资源对每个进程是不同的, 因此被拷贝到这里。
- ⑧ copy_process()做扫尾工作并返回一个指向子进程的指针。
- (4) 再回到 do_fork()函数, 如果 copy_process()函数成功返回, 新创建的子进程被唤醒并让其投入运行。内核会选择子进程首先执行。因为一般子进程都会马上调用 exec()函数, 这样可以避免写时拷贝的额外开销, 如果父进程首先执行的话, 有可能会开始向地址空间写入。

3.1.6 fork 出错的原因

- (1) 当前的进程数已经达到了系统规定的上限, 这时 errno 的值被设置为 EAGAIN。
- (2) 系统内存不足, 这时 errno 的值被设置为 ENOMEM。

3.2 exec 系统调用

3.2.1 原型

```
int execl(const char *path, const char *arg, ...)
int execv(const char *path, char *const argv[])
int execl(const char *path, const char *arg, ..., char *const envp[])
int execve(const char *path, char *const argv[], char *const envp[])
int execlp(const char *file, const char *arg, ...)
int execvp(const char *file, char *const argv[])
```

3.2.2 头文件

```
unistd.h
```

3.2.3 返回值

- (1) 成功: 函数不返回值。
- (2) 出错: 返回-1, 失败原因记录在 error 中。

3.2.4 六个函数的区别

- (1) 查找方式不同
 - ① 前 4 个函数的查找方式都是要求有完整的文件目录路径。
 - ② 后 2 个函数 (以 p 结尾的两个函数) 可以只给出文件名, 系统会自动从环境变量 “\$PATH” 所指出的路径中进行查找。
- (2) 参数传递方式不同
 - ① 函数名的第 5 位字母为 “l” (list) 的表示逐个列举的方式。
 - ② 函数名的第 5 位字母为 “v” (vector) 的表示将所有参数整体构造成指针数组传递, 然后将该数组的首地址当做参数传给它, 数组中的最后一个指针要求是 NULL。
- (3) 环境变量不同
 - ① 以 “e” (environment) 结尾的两个函数 execl、execve 可以在 envp[] 中指定当前进程所使用的环境变量替换掉该进程继承的环境变量。
 - ② 其它函数把调用进程的环境传递给新进程。



3.2.5 功能

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容。换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何 Linux 下可执行的脚本文件。在 exec 函数执行完毕后，原调用进程的内容除了进程号外，全部被新程序的内容替换。

3.2.6 底层原理

内核中实际执行 execv()或 execve()系统调用的程序是 do_execve(), 这个函数先打开目标映像文件，并从目标文件的头部(从第一个字节开始)读入若干(128)字节，然后调用另一个函数 search_binary_handler()。内核所支持的每种可执行程序都有 struct linux_binfmt 数据结构，该结构会通过向内核登记挂入一个队列。而 search_binary_handler()则会扫描这个队列，让各种可执行程序的处理程序前来认领和处理。如果类型匹配，则调用 load_binary()函数指针所指向的处理函数来处理目标映像文件。

3.3 exit 系统调用

3.3.1 原型

```
void _exit(int status)
void exit(int status)
```

3.3.2 头文件

```
unistd.h
stdlib.h
```

3.3.3 功能

终止发出调用的进程，status 是返回给父进程的状态值，父进程可通过 wait 系统调用获得。

3.3.4 exit()和_exit()的区别和联系

- (1) _exit()的作用最简单：直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构。
- (2) exit()在终止进程之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，即“清理 I/O 缓冲”。
- (3) 两者最终都要将控制权交给内核。
- (4) 因此，要想保证数据的完整性，就一定要使用 exit()。

3.3.5 linux 中进程的两种退出方式

(1) 正常退出

- ① main()函数执行完成或在 main()函数中执行 return。
- ② 调用 exit()函数。
- ③ 调用 _exit()函数。

(2) 异常退出

- ① 调用 abort()函数。
- ② 进程收到某个信号，而该信号使程序终止。

(3) 注意

- ① exit()终止进程时，将使终止的进程进入僵死状态，释放它占有的资源，撤除进程上下文，但仍保留 proc 结构。



② 子进程还未终止，但父进程已终止时，将交由 init 进程处理。

3.4 wait 系统调用

3.4.1 原型

```
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options)
```

3.4.2 头文件

```
sys/types.h
sys/wait.h
```

3.4.3 参数

- (1) status: 返回子进程退出时的状态。
- (2) pid:
 - ① pid>0 时: 等待进程号为 pid 的子进程结束。
 - ② pid=0 时: 等待组 ID 等于调用进程组 ID 的子进程结束。
 - ③ pid=-1 时: 等待任一子进程结束，等价于调用 wait()。
 - ④ pid<-1 时: 等待组 ID 等于 PID 的绝对值的任一子进程结束。
- (3) options:
 - ① WNOHANG: 若 pid 指定的子进程没有结束，则 waitpid()不阻塞而立即返回，返回值为 0。
 - ② WUNTRACED: 为了实现某种操作，由 pid 指定的任一进程已被暂停，且其状态自暂停以来还未报告过，则返回其状态。
 - ③ 0: 同 wait(), 阻塞父进程，等待子进程退出。

3.4.4 返回值

- (1) 成功: 则返回成功结束运行的子进程的进程号 PID。
- (2) 出错: 返回 -1。

3.4.5 功能

wait(), waitpid()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 wait(), waitpid()时子进程已经结束，则 wait(), waitpid()会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一起返回。如果不在意结束状态值，则参数 status 可以设成 NULL。

3.4.6 wait()和 waitpid()的区别

- (1) wait()函数等待所有子进程的僵死状态。
- (2) waitpid()函数等待 PID 与参与 pid 相关的子进程的僵死状态。

3.4.9 底层原理

进程一旦调用了 wait(), 就立即阻塞自己，由 wait()自动分析当前进程是否有某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait()就会收集这个子进程的信息，并把它彻底销毁后返回。如果没有找到这样一个子进程，wait()就会一直阻塞在这里，直到有一个出现为止。

3.4.8 检查子进程的返回状态码 status

- (1) WIFEXITED(status): 进程中通过调用 _exit()或 exit()正常退出，该宏值为非 0。
- (2) WIFSIGNALED(status): 子进程因得到的信号没有被捕捉导致退出，该宏值为非 0。



- (3) WIFSTOPPED(status): 子进程没有终止但停止了, 并可重新执行时, 该宏值为 0。这种情况仅出现在 waitpid()调用中使用了 WUNTRACED 选项。
- (4) WEXITSTATUS(status): 如果 WIFEXITED(status)返回非 0, 该宏返回由子进程调用_exit(status)或 exit(status)时设置的调用参数 status 值。
- (5) WTERMSIG(status): 如果 WIFSIGNALED(status)返回非 0, 该宏返回导致子进程退出的信号的值。
- (6) WSTOPSIG(status): 如果 WIFSTOPPED(status)返回非 0, 该宏返回导致子进程停止的信号的值。

3.5 代码实例

(1) 主体程序 thread.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/wait.h>
int main()
{
    printf("-----fork 功能测试-----\n");
    pid_t pid_A=fork();
    if(pid_A<0)
    {
        perror("子进程 A 创建失败! ");
    }
    else if(pid_A==0)
    {
        printf("这是子进程 A, 进程识别号是: %d\n",getpid());
        exit(0);
    }
    else
    {
        printf("这是父进程, 进程识别号是: %d\n",getpid());
        pid_t pid_B=fork();
        if(pid_B<0)
        {
            perror("子进程 B 创建失败! ");
        }
    }
}
```



```
else if(pid_B==0)
{
    printf("这是子进程 B，进程识别号是： %d\n",getpid());
    exit(0);
}
}

usleep(1000000);
printf("-----exec 功能测试-----\n");
pid_t pid_C=fork();
if(pid_C<0)
{
    perror("子进程 C 创建失败！");
}
else if(pid_C==0)
{
    printf("这是子进程 C，进程识别号是： %d\n",getpid());
    printf("即将使用 execl()函数将子进程 C 替换成命令 ls -l\n");
    execl("/bin/ls","ls","-l",NULL);
    printf("脚本没有成功执行！ \n");
}
usleep(1000000);
pid_t pid_Q=fork();
if(pid_Q<0)
{
    perror("子进程 Q 创建失败！");
}
else if(pid_Q==0)
{
    printf("这是子进程 Q，进程识别号是： %d\n",getpid());
    printf("即将使用 execv()函数将子进程 Q 替换成可执行文件 exchange\n");
    execv("./exchange", NULL);
    printf("脚本没有成功执行！ \n");
}
usleep(1000000);
printf("-----exit 功能测试-----\n");
```



```
pid_t pid_D=fork();
if(pid_D<0)
{
    perror("子进程 D 创建失败！");
}
else if(pid_D==0)
{
    printf("这是子进程 D，进程识别号是： %d\n",getpid());
    printf("此进程使用 exit()函数退出\n");
    exit(0);
}
else if(pid_D>0)
{
    pid_t pid_E=fork();
    if(pid_E<0)
    {
        perror("子进程 E 创建失败！");
    }
    else if(pid_E==0)
    {
        printf("这是子进程 E，进程识别号是： %d\n",getpid());
        printf("此进程使用 _exit()函数退出\n");
        _exit(0);
    }
}
usleep(1000000);
printf("-----wait 功能测试-----\n");
pid_t pid_F=fork();
pid_t ret;
int status=0;
if(pid_F<0)
{
    perror("子进程 F 创建失败！");
}
else if(pid_F>0)
```



```
{  
    printf("这里是父进程, 进程识别号是: %d\n",getpid());  
    printf("即将使用 wait()阻塞父进程, 直至子进程 F 退出\n");  
    ret=waitpid(pid_F,&status,0);  
    if(ret<0)  
    {  
        perror("阻塞失败! \n");  
    }  
}  
else if(pid_F==0)  
{  
    printf("这里是子进程 F, 进程识别号是: %d\n",getpid());  
    usleep(1000000);  
    printf("这里是子进程 F, 即将使用 exit(144)结束本进程\n");  
    exit(144);  
}  
printf("这里是父进程, 子进程 F 已经结束, 退出状态是: %d\n",WEXITSTATUS(status));  
}
```

(2) 替换程序 exchange.c

```
#include <stdio.h>  
  
int main()  
{  
    printf("这里是子进程 Q 的替换文件\n");  
    printf("如果看到这两句话说明替换成功\n");  
    return 0;  
}
```



3.6 运行截图

3.6.1 完整运行截图

```
(kali㉿kali)-[~/桌面]
$ ./thread
-----fork功能测试-----
这是父进程，进程识别号是：2745
这是子进程B，进程识别号是：2747
这是子进程A，进程识别号是：2746
-----exec功能测试-----
这是子进程C，进程识别号是：2748
即将使用execl()函数将子进程C替换成命令ls -l
总用量 188
-rwxrwxrwx 1 kali kali 336 4月 5 22:19 aaa.sh
-rw-r--r-- 1 kali kali 402 4月 8 11:44 a.c
-rw-r--r-- 1 kali kali 4 4月 14 20:13 a.txt
-rw-r--r-- 1 kali kali 4 4月 14 20:13 b.txt
-rw-r--r-- 1 kali kali 139 4月 9 21:55 cash.c
-rwxr-xr-x 1 kali kali 17096 4月 14 14:03 client
-rwxrwxrwx 1 kali kali 800 4月 14 14:22 client.txt
-rwxr-xr-x 1 kali kali 17048 4月 8 16:19 computer
-rwxrwxrwx 1 kali kali 5933 4月 14 14:02 computer.c
-rw-r--r-- 1 kali kali 72 4月 5 00:09 cumt
-rw-r--r-- 1 kali kali 10 4月 14 13:38 cumt.txt
-rwxr-xr-x 1 kali kali 16136 4月 18 12:50 exchange
-rw-r--r-- 1 kali kali 160 4月 18 12:50 exchange.c
-rw-r--r-- 1 root root 0 4月 12 21:06 id.txt
-rw-r--r-- 1 kali kali 6 4月 13 21:15 kali.txt
-rw-r--r-- 1 kali kali 183 4月 17 20:51 linux_64.c
-rwxr-xr-x 1 kali kali 17192 4月 14 14:03 serve
-rw-r--r-- 1 kali kali 5823 4月 14 14:03 serve.c
-rwxrwxrwx 1 kali kali 800 4月 14 14:24 serve.txt
-rwxr-xr-x 1 kali kali 1324 4月 6 09:26 test1.sh
-rwxr--r-- 1 kali kali 835 4月 6 08:56 test2.sh
-rwxr-xr-x 1 kali kali 1038 4月 5 21:40 test.sh
drwxr-xr-x 2 kali kali 4096 4月 12 21:07 text
-rwxr-xr-x 1 kali kali 16624 4月 18 12:49 thread
-rw-r--r-- 1 kali kali 3239 4月 18 12:49 thread.c
-rwxr--r-- 1 kali kali 240 4月 18 09:11 vir.cpp
-rwxrwxrwx 1 kali kali 2425 4月 14 10:26 v.sh
这是子进程Q，进程识别号是：2749
即将使用execv()函数将子进程Q替换成可执行文件exchange
这里是子进程Q的替换文件
如果看到这两句话说明替换成功
-----exit功能测试-----
这是子进程E，进程识别号是：2751
此进程使用_exit()函数退出
这是子进程D，进程识别号是：2750
此进程使用exit()函数退出
-----wait功能测试-----
这里是父进程，进程识别号是：2745
即将使用wait()阻塞父进程，直至子进程F退出
这里是子进程F，进程识别号是：2752
这里是子进程F，即将使用exit(144)结束本进程
这里是父进程，子进程F已经结束，退出状态是：144
```




3.6.2 fork 部分分析

首先令父进程打印自己的进程识别号，接着在父进程中利用 fork()函数创建两个子进程 A 和 B，并在子进程中打印出自己的进程识别号。最后终止子进程 A 和 B。

```

fork功能测试
这是父进程，进程识别号是：2745
这是子进程B，进程识别号是：2747
这是子进程A，进程识别号是：2746
  
```

3.6.3 exec 部分分析

首先在父进程中创建两个子进程 C 和 Q，接着让子进程 C 和 Q 打印出自己的进程识别号，并且让子进程 C 执行 execl()函数，让子进程 Q 执行 execv()函数。根据运行结果可以看到，子进程 C 的内容已经被 ls 命令完全替换，子进程 Q 的内容已经被脚本 exchange 完全替换。

```

exec功能测试
这是子进程C，进程识别号是：2748
即将使用execl()函数将子进程C替换成命令ls -l
总用量 188
-rwxrwxrwx 1 kali kali 336 4月 5 22:19 aaa.sh
-rw-r--r-- 1 kali kali 402 4月 8 11:44 a.c
-rw-r--r-- 1 kali kali 4 4月 14 20:13 a.txt
-rw-r--r-- 1 kali kali 4 4月 14 20:13 b.txt
-rw-r--r-- 1 kali kali 139 4月 9 21:55 cash.c
-rwxr-xr-x 1 kali kali 17096 4月 14 14:03 client
-rwxrwxrwx 1 kali kali 800 4月 14 14:22 client.txt
-rwxr-xr-x 1 kali kali 17048 4月 8 16:19 computer
-rwxrwxrwx 1 kali kali 5933 4月 14 14:02 computer.c
-rw-r--r-- 1 kali kali 72 4月 5 00:09 cumt
-rw-r--r-- 1 kali kali 10 4月 14 13:38 cumt.txt
-rwxr-xr-x 1 kali kali 16136 4月 18 12:50 exchange
-rw-r--r-- 1 kali kali 160 4月 18 12:50 exchange.c
-rw-r--r-- 1 root root 0 4月 12 21:06 id.txt
-rw-r--r-- 1 kali kali 6 4月 13 21:15 kali.txt
-rw-r--r-- 1 kali kali 183 4月 17 20:51 linux_64.c
-rwxr-xr-x 1 kali kali 17192 4月 14 14:03 serve
-rw-r--r-- 1 kali kali 5823 4月 14 14:03 serve.c
-rwxrwxrwx 1 kali kali 800 4月 14 14:24 serve.txt
-rwxr-xr-x 1 kali kali 1324 4月 6 09:26 test1.sh
-rwxr--r-- 1 kali kali 835 4月 6 08:56 test2.sh
-rwxr-xr-x 1 kali kali 1038 4月 5 21:40 test.sh
drwxr-xr-x 2 kali kali 4096 4月 12 21:07 text
-rwxr-xr-x 1 kali kali 16624 4月 18 12:49 thread
-rw-r--r-- 1 kali kali 3239 4月 18 12:49 thread.c
-rwxr--r-- 1 kali kali 240 4月 18 09:11 vir.cpp
-rwxrwxrwx 1 kali kali 2425 4月 14 10:26 v.sh
这是子进程Q，进程识别号是：2749
即将使用execv()函数将子进程Q替换成可执行文件exchange
这里是子进程Q的替换文件
如果看到这两句话说明替换成功
  
```

3.6.4 exit 部分分析

首先在父进程中创建两个子进程 D 和 E，接着令两个子进程打印出自己的进程识别号，打印完毕后，两个子进程分别使用 exit()和 _exit()函数退出。



```
—————exit功能测试—————  
这是子进程E, 进程识别号是: 2751  
此进程使用 _exit()函数退出  
这是子进程D, 进程识别号是: 2750  
此进程使用 exit()函数退出
```

3.6.5 wait 部分分析

首先在父进程中创建子进程 F, 接着利用 wait()函数阻塞父进程, 并当子进程 F 结束时, 在父进程中打印出子进程 F 的退出时状态。

```
—————wait功能测试—————  
这里是父进程, 进程识别号是: 2745  
即将使用 wait()阻塞父进程, 直至子进程F退出  
这里是子进程F, 进程识别号是: 2752  
这里是子进程F, 即将使用 exit(144)结束本进程  
这里是父进程, 子进程F已经结束, 退出状态是: 144
```

3.7 体会

在本次实验中, 我完整的总结了进程调度时使用的四个系统调用 fork, wait, exit, exec 的功能, 使用方法, 以及实现原理。这让我对 linux 系统下的进程调度有了一个整体的认识。同时, 我还编写了一个简单的测试程序去测试各个系统调用的功能, 其中让我印象最深的就是通过 fork 系统调用创建出来的子进程和父进程完全一样, 这让我在编程初期并不习惯, 出现了很多错误。

通过本次实验, 我对 linux 操作系统有了更加深刻的认识, 同时也清醒的认识到一个系统所包括的东西庞杂无序, 想要对其有一个比较系统的了解, 需要付出大量的时间和精力。

3.8 参考文献

- [1] Robert Love. LINUX 系统编程. 东南大学出版社, 2009.
- [2] 鸟哥. 鸟哥的 Linux 私房菜: 基础学习篇[M]. 人民邮电出版社, 2010.



4 Linux 网络编程

要求: (1) 掌握 linux 下 socket 编程相关的各种系统调用: socket、bind、connect、listen、accept、read、recvfrom、write、sendto、close。

(2) 独立完成一个 linux 下的网络通信程序, 要求包括客户端和服务端两部分程序, 互相能够通信, 传递消息, 传送文件, 即时聊天等等, 要有必要的注释, 并将运行过程和结果截图附上, 若无运行截图, 扣 10 分。

(3) 根据所完成的网络通信程序的难度与功能复杂程度给出分值。

说明: 如果撰写规范不符合《计算机学院考查类课程报告撰写规范》要求的, 整体上酌情扣除 1-10 分。

4.1 socket 编程常见的系统调用

4.1.1 socket

(1) 功能

socket 系统调用常用于创建一个 socket 描述符。

(2) 原型

```
int socket(int domain, int type, int protocol)
```

(3) 头文件

```
sys/types.h
sys/socket.h
```

(4) 参数

① domain: 程序采用的通讯协族

- a. AF_UNIX: 只用于单一的 Unix 系统进程间通信
- b. AF_INET: 用于 Internet 通信

② type: 采用的通讯协议

- a. SOCK_STREAM: 使用 TCP 协议
- b. SOCK_DGRAM: 使用 UDP 协议

③ protocol: 由于指定了 type, 此值一般为 0

(5) 返回值

成功时返回 socket 描述符, 失败时返回-1, 可用 error 查看出错的详细情况。

4.1.2 bind

(1) 功能

bind 系统调用常用于 server 程序, 绑定被侦听的端口。

(2) 原型

```
int bind(int sockfd, struct sockaddr* servaddr, int addrlen)
```

(3) 头文件

```
sys/types.h
sys/socket.h
```

(4) 参数

① sockfd: 由 socket 调用返回的文件描述符

② servaddr: 出于兼容性, 一般使用 sockaddr_in 结构

③ addrlen: servaddr 结构的长度

(5) 返回值

① 成功: 0



- ② 失败: -1, 相应的设定全局变量 error, 最常见的错误是该端口已经被其它程序绑定

4.1.3 connect

(1) 功能

connect 系统调用常用于 Client 程序, 连接到某个 Server。

(2) 原型

```
int connect(int sockfd, struct sockaddr* servaddr, int addrlen)
```

(3) 头文件

```
sys/types.h  
sys/socket.h
```

(4) 参数

- ① sockfd: socket 返回的文件描述符
- ② servaddr: 被连接的服务器端地址和端口信息, 出于兼容性, 一般使用 sockaddr_in 结构
- ③ addrlen: servaddr 的长度

(5) 返回值

- ① 成功: 0
- ② 失败: -1, 相应地设定全局变量 errno

4.1.4 listen

(1) 功能

listen 系统调用常用于 server 程序, 侦听 bind 绑定的套接字。

(2) 原型

```
int listen(int sockfd, int backlog)
```

(3) 头文件

```
sys/types.h  
sys/socket.h
```

(4) 参数

- ① sockfd: 被 bind 的文件描述符 (socket()建立的)
- ② backlog: 设置 Server 端请求队列的最大长度

(5) 返回值

- ① 成功: 0
- ② 失败: -1

4.1.5 accept

(1) 功能

Server 用它响应连接请求, 建立与 Client 连接

(2) 原型

```
int accept(int sockfd, struct sockaddr* addr, int *addrlen)
```

(3) 头文件

```
sys/types.h  
sys/socket.h
```

(4) 参数

- ① sockfd: listen 后的文件描述符 (socket()建立的)
- ② addr: 返回 Client 的 IP、端口等信息, 确切格式由套接字的地址类别 (如 TCP 或 UDP) 决定。若 addr 为 NULL, 则 addrlen 应置为 NULL



- ③ `addrlen`: 返回真实的 `addr` 所指向结构的大小, 只要传递指针就可以, 但必须先初始化为 `addr` 所指向结构的大小

(5) 返回值

- ① 成功: Server 用于与 Client 进行数据传输的文件描述符
② 失败: -1, 相应地设定全局变量 `errno`

4.1.6 `recv`

(1) 功能

`recv` 系统调用常用于 TCP 协议中接收信息。

(2) 原型

```
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags)
```

(3) 头文件

```
sys/types.h
sys/socket.h
```

(4) 参数

- ① `sockfd`: 接收端套接字描述符
② `buf`: 指向容纳接收信息的缓冲区的指针
③ `nbytes`: `buf` 缓冲区的大小
④ `flags`: 接收标志, 一般置为 0 或:

flags	说明
<code>MSG_DONTWAIT</code>	仅本操作非阻塞
<code>MSG_OOB</code>	发送或接收带外数据
<code>MSG_PEEK</code>	窥看外来消息
<code>MSG_WAITALL</code>	等待所有数据

(5) 返回值

- ① 成功: 实际接收的字节数
② 失败: -1, 相应地设定全局变量 `errno`
③ 为 0: 表示对端已经关闭

4.1.7 `recvfrom`

(1) 功能

`Recvfrom` 系统调用常用于 UDP 协议中接收信息。

(2) 原型

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)
```

(3) 头文件

```
sys/types.h
sys/socket.h
```

(4) 参数

- ① `sockfd`: socket 描述符
② `buf`: 指向容纳接收 UDP 数据报的缓冲区的指针
③ `len`: `buf` 缓冲区的大小
④ `flags`: 接收标志, 一般为 0
⑤ `from`: 指明数据的来源
⑥ `fromlen`: 传入函数之前初始化为 `from` 的大小, 返回之后存放 `from` 实际大小



(5) 返回值

- ① 成功：实际接收的字节数
- ② 失败：-1，错误原因存于 `errno` 中
- ③ 为 0：表示对端已经关闭

4.1.8 send

(1) 功能

send 系统调用常用于 TCP 协议中发送信息。

(2) 原型

```
ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags)
```

(3) 头文件

```
sys/types.h
sys/socket.h
```

(4) 参数

- ① sockfd：指定发送端套接字描述符
- ② buf：存放要发送数据的缓冲区
- ③ nbytes：实际要发送的数据的字节数
- ④ flags：一般设置为 0 或：

flags	说明
MSG_DONTRoute	绕过路由表查找
MSG_DONTWAIT	仅本操作非阻塞
MSG_OOB	发送或接收带外数据

(5) 返回值

- ① 成功：已发送的字节数
- ② 失败：-1，相应地设定全局变量 `errno`

4.1.9 sendto

(1) 功能

sendto 系统调用常用于 UDP 协议中发送信息。

(2) 原型

```
int sendto(int sockfd, const void *msg, int len, unsigned int flag, const struct sockaddr *to,
int tolen)
```

(3) 头文件

```
sys/types.h
sys/socket.h
```

(4) 参数

- ① sock：将要从其发送数据的套接字
- ② buf：指向将要发送数据的缓冲区
- ③ len：数据缓冲区的长度
- ④ flags：一般是 0
- ⑤ to：指明数据的目的地
- ⑥ tolen：to 内存区的长度

(5) 返回值

- ① 成功：实际传送出去的字符数
- ② 失败：-1，错误原因存于 `errno` 中



4.1.10 close

(1) 功能

close 系统调用常用于 TCP，关闭特定的 socket 连接

(2) 原型

```
int close(int sockfd)
```

(3) 头文件

```
sys/types.h
sys/socket.h
```

(4) 参数

① sockfd: 要关闭的 socket 描述符

(5) 返回值

① 成功: 0

② 失败: -1, 并置错误码 errno:

EBADF	sockfd 不是一个有效描述符
EINTR	close 函数被信号中断
EIO	IO 错误

4.1.11 read

(1) 功能

read 系统调用负责从 socket 描述符对应的 socket 中读取内容。

(2) 原型

```
ssize_t read(int fd, void *buf, size_t count)
```

(3) 头文件

```
unistd.h
```

(4) 参数

① fd: 文件描述符

② buf: 指向内存块的指针, 存放从文件中读出的字节

③ count: 写入到 buf 中的字节数

(5) 返回值

① 成功: 返回实际所读取的字节数, 如果已经读取到文件的结尾则返回 0

② 失败: 返回小于 0 的数

4.1.12 write

(1) 功能

write 系统调用常用于向发送缓冲区写数据。

(2) 原型

```
ssize_t write(int fd, const void *buf, size_t count)
```

(3) 头文件

```
unistd.h
```

(4) 参数

① fd: 文件描述符

② buf: 指向内存块的指针, 从该内存块读出数据写入文件

③ count: 写入到文件中的字节数

(5) 返回值



- ① 成功：返回写入的字节数
- ② 失败：返回-1 并设置 error

4.2 实现方法

4.2.1 服务器端

- (1) 首先创建服务器端 socket 以及 sockaddr_in 结构体。
- (2) 接着利用 bind 函数绑定要监听的端口，利用 listen 函数进行监听，如果监听到客户端的连接请求则利用 accept 函数和客户端连接。
- (3) 成功连接后利用 fork 函数建立一个子进程，和父进程并行执行。其中子进程负责向客户端发送信息及文件，父进程负责接收来自客户端的进程和文件。
- (4) 在父进程中，如果接收到"file"指令则进入文件接收分支：创建新文件，并将接收到的内容写入新文件中。如果接收到"quit"指令则代表客户端下线，则服务器端也停止运行。除上述两种情况，接收到的所有信息均作为实时通信内容，父进程在接收后会将其打印在命令框中。
- (5) 在子进程中，如果发送内容为"file"，则进入文件发送分支：按照输入的文件地址读取文件，并将读取到的内容发送给客户端。如果发送内容为"quit"，则服务器下线，停止运行。除此之外发送的所有内容均作为实时通信内容。

4.2.2 客户端

- (1) 首先设定一个用户名，在和服务器交互时使用。
- (2) 接着创建客户端 socket 以及 sockaddr_in 结构体。
- (3) 接着利用 connect 函数请求和服务器连接。
- (4) 成功连接后将用户名发送给服务器，并利用 fork 函数建立一个子进程，和父进程并行执行。其中父进程负责向客户端发送信息及文件，子进程负责接收来自客户端的进程和文件。
- (5) 在父进程中，如果发送内容为"file"，则进入文件发送分支：按照输入的文件地址读取文件，并将读取到的内容发送给服务器端。如果发送内容为"quit"，则客户端下线，停止运行。除此之外发送的所有内容均作为实时通信内容。
- (6) 在子进程中，如果接收到"file"指令则进入文件接收分支：创建新文件，并将接收到的内容写入新文件。如果接收到"quit"指令则代表服务器下线，则客户端也停止运行。除此之外，接收到的所有信息均作为实时通信内容，子进程在接收后会将其打印在命令框中。

4.3 源代码

4.3.1 服务器端

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <sys/prctl.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>
#include <sys/time.h>
```



```
#include <ctype.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    printf("服务器启动中.....\n");
    int servefd=socket(AF_INET,SOCK_STREAM,0); //创建服务器端套接字
    if(servefd==-1)
    {
        perror("服务器 socket 创建失败! ");
        exit(1);
    }
    struct sockaddr_in server_addr; //创建 sockaddr_in 结构体
    server_addr.sin_family=AF_INET;
    server_addr.sin_port=htons(8888);
    server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
    bzero(&(server_addr.sin_zero),8);
    int my_bind=bind(servefd,(struct sockaddr*)&server_addr,sizeof(server_addr)); //绑定被侦听端口
    if(my_bind==-1)
    {
        perror("服务器绑定失败! ");
        exit(1);
    }
    int my_listen=listen(servefd,10); //监听 bind 绑定的套接字
    if(my_listen==-1)
    {
        perror("服务器监听失败! ");
        exit(1);
    }
    printf("服务器已正常启动!\n");
    int clientfd=accept(servefd,NULL,NULL); //响应客户端请求, 建立连接
    if(clientfd==-1)
    {
        perror("服务器连接客户端失败! ");
        exit(1);
    }
    char username[100];
    recv(clientfd,username,100,0); //接收客户端传来的用户名
    printf("用户%s 已成功登录服务器, 可以开始交互\n\n",username);
    pid_t my_fork=fork(); //创建一个子进程
    while(1)
    {
        if(my_fork==0) //子进程用来发送信息及文件
        {
```



```

char my_sent[100];
scanf("%s",my_sent); //输入要发送的信息
if(strlen(my_sent)!=0)
{
    int my_send=send(clientfd,my_sent,100,0); //发送信息
    if(my_send==-1)
    {
        perror("信息发送失败! ");
    }
    if(my_sent[0]=='q'&&my_sent[1]=='u'&&my_sent[2]=='i'&&my_sent[3]=='t') //如果发送 quit
    则服务器下线
    {
        printf("服务器已下线, 通信终止\n");
        close(servefd);
        kill(getppid(),SIGTERM);
        exit(0);
    }
    if(my_sent[0]=='f'&&my_sent[1]=='i'&&my_sent[2]=='l'&&my_sent[3]=='e') //遇到 file 指令
    则传送文件
    {
        char filename[100];
        int my_eof=1;
        printf("请输入完整的文件名: ");
        scanf("%s",filename);
        printf("正在向用户%s 传送文件.....\n",username);
        int fd=open(filename,O_RDWR); //建立文件流
        int error=0;
        while(my_eof) //传送文件
        {
            my_eof=read(fd,my_sent,100);
            my_send=send(clientfd,my_sent,my_eof,0);
            if(my_send==-1)
            {
                usleep(3000000);
                perror("文件传送失败! ");
                send(clientfd,"error",20,0);
                error=1;
                break;
            }
        }
        if(error==0)
        {
            usleep(3000000);
            send(clientfd,"exit",20,0);
            printf("文件传送成功! \n");
        }
    }
}

```



```

    }

    }

}

else if(my_fork>0) //父进程用于接收信息及文件
{
    char my_receive[100];
    int my_recv=recv(clientfd,my_receive,100,0); //接收信息
    if(strlen(my_receive)!=0)
    {
        printf("%s: %s\n",username,my_receive); //将接收到的信息打印出来
        if(my_receive[0]=='q'&&my_receive[1]=='u'&&my_receive[2]=='i'&&my_receive[3]=='t') //
遇到quit 则客户端下线
        {
            printf("用户%s 已退出服务器, 通信终止\n",username);
            close(servefd);
            kill(my_fork,SIGTERM);
            exit(0);
        }
        if(my_receive[0]=='f'&&my_receive[1]=='i'&&my_receive[2]=='l'&&my_receive[3]=='e') //
遇到file 则接收文件
        {
            printf("正在接收用户%s 传送的文件.....\n",username);
            int fd=open("/home/kali/桌面/client.txt",O_RDWR|O_CREAT); //将接收的文件保存在
client.txt 中
            while(1)
            {
                my_recv=recv(clientfd,my_receive,100,0); //接收文件内容
                if(strlen(my_receive)!=0)
                {
                    if(my_receive[0]=='e'&&my_receive[1]=='x'&&my_receive[2]=='i'&&my_receive[
3]=='t') //遇到exit 的接收完毕
                    {
                        printf("文件接收成功! \n");
                        close(fd); //关闭文件流
                        break;
                    }
                    if(my_receive[0]=='e'&&my_receive[1]=='r'&&my_receive[2]=='r'&&my_receive[
3]=='o'&&my_receive[4]=='r') //遇到error 则接收失败
                    {
                        printf("文件接收失败! \n");
                        close(fd); //关闭文件流
                        break;
                    }
                }
            }
        }
    }
}

```




```
else write(fd,my_receive,strlen(my_receive)); //将接收到的内容写入文件
    }
    }
    }
    }
    }
}
```

4.3.2 客户端

```
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <ctype.h>
#include <sys/wait.h>
#include <sys/prctl.h>
#include <unistd.h>
#include <sys/time.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char username[100];
    while(1) //设定登录服务器后的用户名，用户名不能为空
    {
        printf("请设定用户名: ");
        scanf("%s",username);
        if(strlen(username)==0)
        {
            perror("用户名不能为空! ");
        }
        else
        {
            break;
        }
    }
}
```



```

    }
}
int clientfd=socket(AF_INET,SOCK_STREAM,0); //创建客户端套接字
if(clientfd==-1)
{
    perror("客户端 socket 创建失败! ");
    exit(1);
}
struct sockaddr_in server_addr; //创建 sockaddr_in 结构体
server_addr.sin_family=AF_INET;
server_addr.sin_port=htons(8888);
server_addr.sin_addr.s_addr=inet_addr("127.0.0.1");
bzero(&(server_addr.sin_zero),8);
printf("正在登陆服务器.....\n");
int my_connect=connect(clientfd,(struct sockaddr*)&server_addr,sizeof(struct sockaddr_in)); //向服务器
请求连接
if(my_connect==-1)
{
    perror("连接服务器失败! ");
    exit(1);
}
printf("服务器登陆成功,可以开始交互\n\n");
send(clientfd,username,100,0); //将设定的用户名发送给服务器
pid_t my_fork=fork(); //创建一个子进程
while(1)
{
    if(my_fork>0) //父进程用于发送消息及文件
    {
        char my_sent[100];
        scanf("%s",my_sent); //输入要发送的信息
        if(strlen(my_sent)!=0)
        {
            int my_send=send(clientfd,my_sent,100,0); //将信息发送给服务器
            if(my_send==-1)
            {
                perror("信息发送失败! ");
            }
            if(my_sent[0]=='q'&&my_sent[1]=='u'&&my_sent[2]=='i'&&my_sent[3]=='t') //如果发送 quit
则客户端下线
            {
                printf("您已成功退出服务器,通信终止");
                close(clientfd);
                kill(my_fork,SIGTERM);
                exit(0);
            }
        }
    }
}

```



if(my_sent[0]=='f'&&my_sent[1]=='i'&&my_sent[2]=='l'&&my_sent[3]=='e') //如果发送 file
则客户端向服务器发送文件

```
{
    char filename[100];
    int my_eof=1;
    printf("请输入完整的文件名: "); //输入要发送文件的名称
    scanf("%s",filename);
    printf("正在向服务器传送文件.....\n");
    int fd=open(filename,O_RDWR); //创建文件流
    int error=0;
    while(my_eof) //发送文件
    {
        my_eof=read(fd,my_sent,100);
        my_send=send(clientfd,my_sent,my_eof,0);
        if(my_send==-1)
        {
            usleep(3000000);
            send(clientfd,"error",20,0);
            perror("文件传送失败! ");
            error=1;
            break;
        }
    }
    if(error==0)
    {
        usleep(3000000);
        send(clientfd,"exit",20,0);
        printf("文件传送成功! \n");
    }
}

}

else if(my_fork==0) //子进程用于接收信息及文件
{
    char my_receive[100];
    int my_rcv=recv(clientfd,my_receive,100,0); //接收服务器发送的信息
    if(strlen(my_receive)!=0)
    {
        if(my_receive[0]=='q'&&my_receive[1]=='u'&&my_receive[2]=='i'&&my_receive[3]=='t') //
        遇到 quit 则服务器下线
        {
            printf("服务器已下线, 通信终止\n");
            close(clientfd);
            kill(getppid(),SIGTERM);
        }
    }
}
```

```
3]== 'o' && my_receive[4] == 'r') //如果遇到 error 则接收失败
```



4.4 运行截图

4.4.1 实时通信

(1) 服务器端

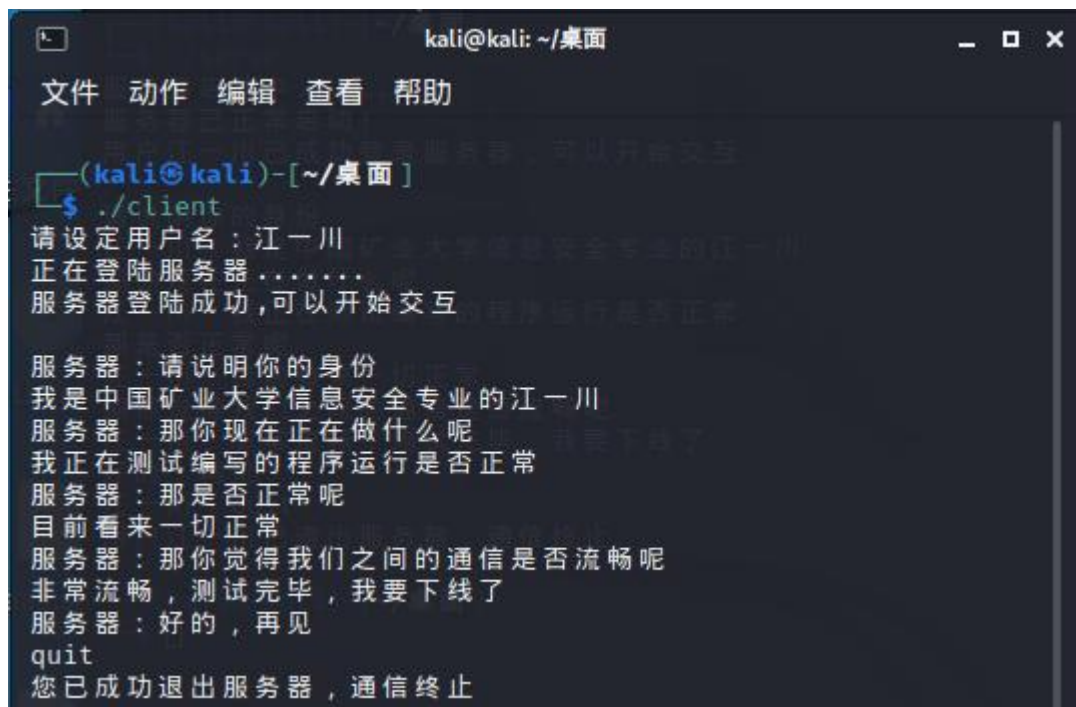


```
kali@kali: ~/桌面
文件 动作 编辑 查看 帮助

(kali@kali)-[~/桌面]
$ ./serve
服务器启动中 .....
服务器已正常启动！
用户江一川已成功登录服务器，可以开始交互

请说明你的身份
江一川：我是中国矿业大学信息安全专业的江一川
那你现在正在做什么呢
江一川：我正在测试编写的程序运行是否正常
那是否正常呢
江一川：目前看来一切正常
那你觉得我们之间的通信是否流畅呢
江一川：非常流畅，测试完毕，我要下线了
好的，再见
江一川：quit
用户江一川已退出服务器，通信终止
```

(2) 客户端



```
kali@kali: ~/桌面
文件 动作 编辑 查看 帮助

(kali@kali)-[~/桌面]
$ ./client
请设定用户名：江一川
正在登陆服务器 .....
服务器登陆成功，可以开始交互

服务器：请说明你的身份
我是中国矿业大学信息安全专业的江一川
服务器：那你现在正在做什么呢
我正在测试编写的程序运行是否正常
服务器：那是否正常呢
目前看来一切正常
服务器：那你觉得我们之间的通信是否流畅呢
非常流畅，测试完毕，我要下线了
服务器：好的，再见
quit
您已成功退出服务器，通信终止
```




4.4.2 文件传输

(1) 服务器端

```
kali@kali: ~/桌面
文件 动作 编辑 查看 帮助
正在接收用户江一川传送的文件.....
(kali@kali)-[~/桌面]
$ ./serve
服务器启动中.....
服务器已正常启动！
用户江一川已成功登录服务器，可以开始交互

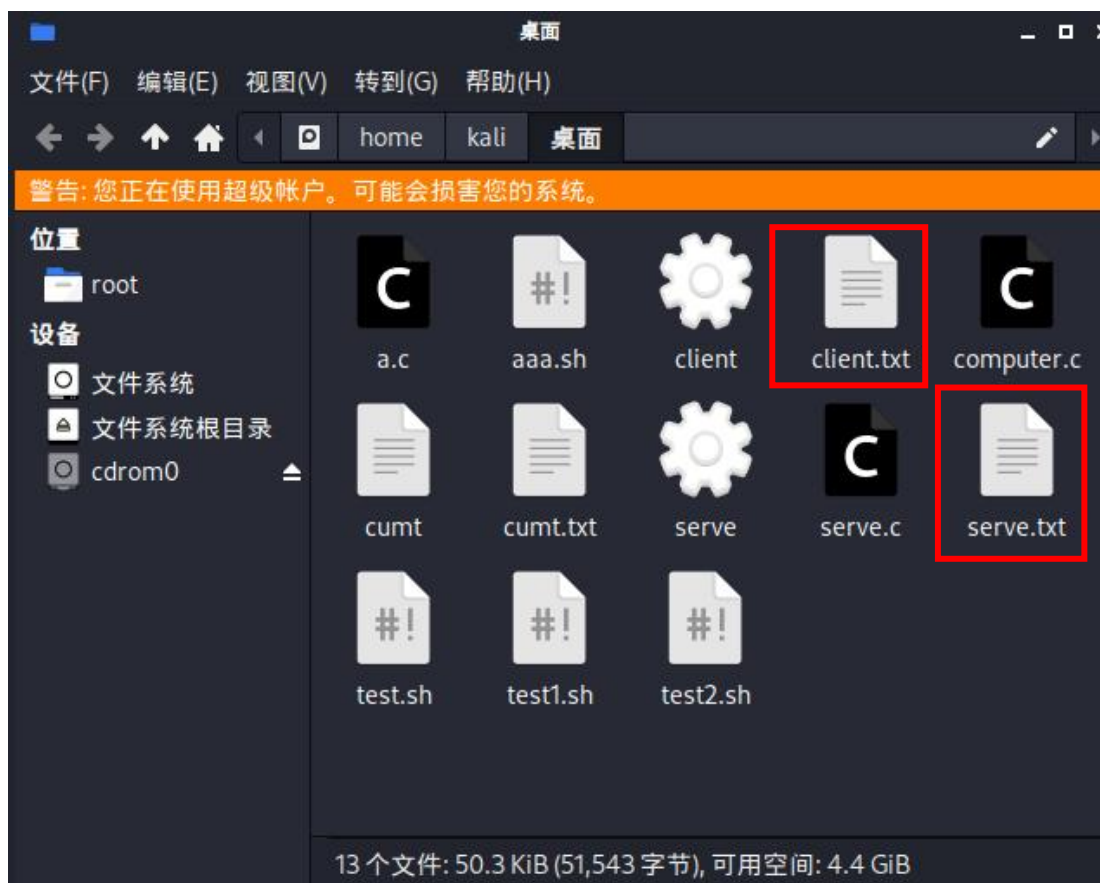
江一川：我可以向你传输文件么
可以
江一川：file
正在接收用户江一川传送的文件.....
文件接收成功！
我可以向你传输文件么
江一川：可以
file
请输入完整的文件名：/home/kali/桌面/cumt.txt
正在向用户江一川传送文件.....
文件传送成功！
quit
服务器已下线，通信终止
```

(2) 客户端

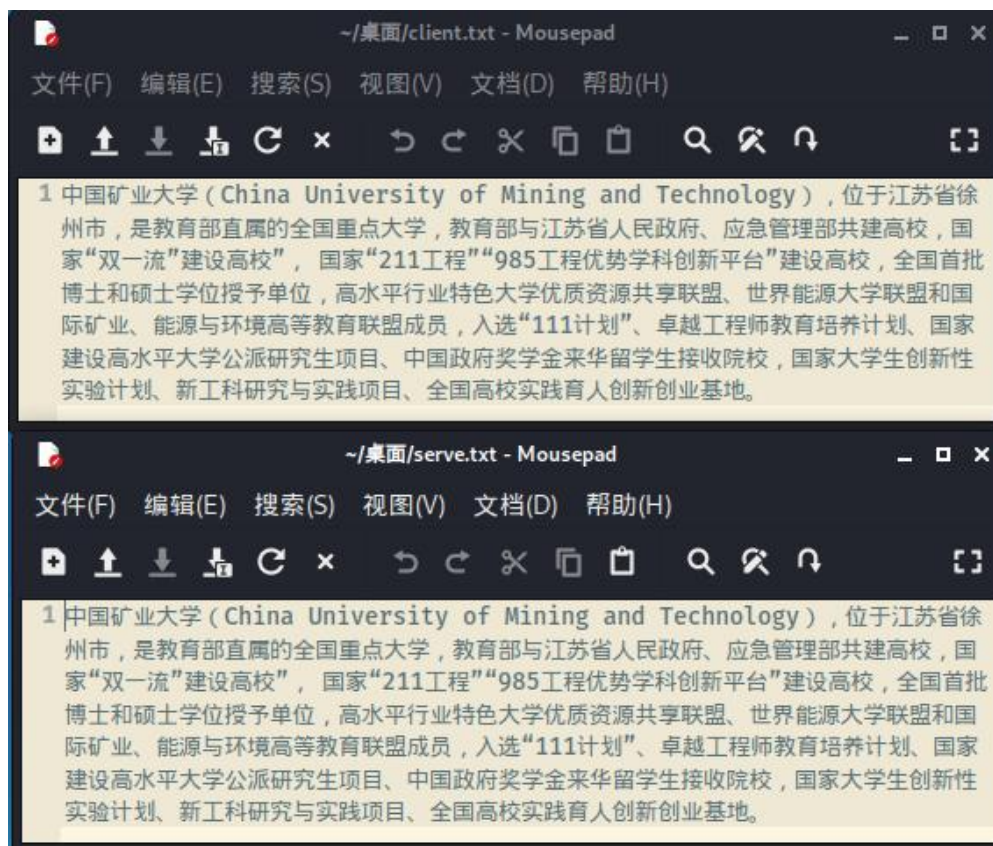
```
kali@kali: ~/桌面
文件 动作 编辑 查看 帮助
(kali@kali)-[~/桌面]
$ ./client
请设定用户名：江一川
正在登陆服务器.....
服务器登陆成功,可以开始交互

我可以向你传输文件么
服务器：可以
file
请输入完整的文件名：/home/kali/桌面/cumt.txt
正在向服务器传送文件.....
文件传送成功！
服务器：我可以向你传输文件么
可以
正在接收服务器传送的文件.....
文件接收成功！
服务器已下线，通信终止
```

(3) 创建的文件



(4) 文件内容





4.5 体会

在本次实验中，我总结了网络通信时会使用到的各种函数的功能，使用方法以及实现原理，并依托此编写了一个基于客户端-服务器模式的网络通信程序。在编写的过程中，遇到了诸多的问题，其中大部分问题都是由于帮助手册等资料对函数的描述不够细节，而在使用的时候却需要落到细节处造成的。

通过本次实验，我对网络通信的基本模式有了一定的了解，同时也对用于网络通信的各个功能函数有了一定的了解，虽然可能说认识的并不完整，但我相信随着后期的继续学习，我会对此有更加深刻完整的认识。

4.6 参考文献

- [1] Robert Love. LINUX 系统编程. 东南大学出版社, 2009.
- [2] 鸟哥. 鸟哥的 Linux 私房菜: 基础学习篇[M]. 人民邮电出版社, 2010.



5 编程附加题

要求：(1) 此题为附加题目，可选择有创意的题目实现。

(2) 根据题目的新颖性，正确性酌情扣分。

说明：如果撰写规范不符合《计算机学院考查类课程报告撰写规范》要求的，整体上酌情扣除 1-10 分。

5.1 程序介绍

本程序是利用 linux 操作系统下的 shell 实现的一个 arkanoid 游戏。该游戏的规则是玩家需要利用发射出去的弹球去撞击砖块，当砖块全部击碎则成功通关，否则通过失败。玩家可以利用游戏提供的底板反弹下落的弹球，让其继续弹出去撞击砖块，该底板的控制方式是按下 a 键左移，按下 d 键右移，如果弹球落地则玩家减少一条生命(每个玩家共有三条生命)。

5.2 源代码

```
#!/bin/bash
stty -echo

OLD_IFS="$IFS"
IFS=""

delay=0.2

tput civis
clear

screen_width=$(tput cols)
screen_height=$((tput lines - 1))
width=$((40 % (screen_width - 2)))
height=$((14 % (screen_height - 2)))

half_width=$((width/2))
half_henght=$((height/2))

top=$((screen_height - height) / 2)
left=$((screen_width - width) / 2)
bottom=$((top + height))
right=$((left + width))

state='stop'

plateX=$((left + half_width))
plateW=5
plateS=""

ballY=$((bottom - 1))
ballX=$((left + half_width + plateW / 2))
```



```
ballDY=-1
ballDX=-1
ballColors=(52 88 124 160 196)
currentColor=0
ballChar="+"
state="stop"
lives=3

bricks=()

function putBrik {
    index=1$1\0$2
    briks[$index]=1
    tput cup $1 $2
    echo '='
}

function drawBorder {
    line=""
    tput setaf 241
    for (( column = 0; column <= width; column++ ))
    do
        line+="_ "
    done
    tput cup $((top - 1)) $left
    echo $line
    tput cup $bottom $left
    echo $line

    for (( row = 0; row <= height; row++ ))
    do
        tput cup $((top + row)) $((left - 1))
        echo "|"
        tput cup $((top + row)) $((right + 1))
        echo "|"
    done

    tput cup $((bottom + 2)) $left
    echo "Press 'a' or 'd' to start playing"
}

function clearBall {
    tput cup $ballY $ballX
    echo " "
}
```



```
function drawBall {
    tput setaf ${ballColors[$currentColor]}
    tput cup $ballY $ballX
    echo $ballChar

    currentColor=$((currentColor + 1))
    if [ $currentColor -gt 4 ]
    then
        currentColor=0
    fi
}

function resetBall {
    clearBall
    ballY=$((bottom - 1))
    ballX=$((plateX + plateW / 2))
    ballDY=-1
    ballDY=-1
    drawPlate
    drawBall
}

function drawBricks {
    for (( row = $(top + 2); row <= $(top + 7); row++ ))
    do
        for (( column = $(left + 3); column <= $(right - 3); column++ ))
        do
            if [ $(($column % 3)) == 0 ]
            then
                tput setaf 83
                putBrik $row $column
            fi
        done
    done

    for (( column = $(left + 3); column <= $(right - 3); column++ ))
    do
        if [ $(($column % 2)) == 0 ]
        then
            tput setaf 84
            putBrik $(top + 3) $column
        fi
        if [ $(($column % 3)) == 0 ]
        then
```




```

        tput setaf 85
        putBrik $((top + 6)) $column
    fi
done

for (( column = $(left + 2); column <= $(right - 2); column++ ))
do
    if [ $(column % 4) == 0 ]
    then
        tput setaf 86
        putBrik $((top + 8)) $column
    fi
    if [ $(column % 2) == 0 ]
    then
        tput setaf 86
        tput setaf 87
        putBrik $((top + 4)) $column
    fi
done
}

function move {
    (sleep $delay && kill -ALRM $$) &

    if [ $state != 'playing' ]
    then
        return
    fi

    clearBall
    ballY=$((ballY + ballDY))
    ballX=$((ballX + ballDX))

    if [ $ballX -gt $right ] || [ $ballX -lt $left ]
    then
        ballDX=$((-ballDX))
        ballX=$((ballX + ballDX + ballDX))
    fi

    if [ $ballY -lt $top ]
    then
        ballDY=$((-ballDY))
        ballY=$((ballY + ballDY + ballDY))
    fi
}

```



```

if [ $ballY -gt $((bottom - 1)) ]
then
    if [ $ballX -le $((plateX + plateW)) ] && [ $ballX -ge $plateX ]
    then
        ballX=$((plateX + plateW / 2))
        ballDY=$((-ballDY))
        ballY=$((ballY + ballDY + ballDY))
        drawPlate
    else
        resetBall
        drawBorder
        state='stop'
        lifes=$((lifes - 1))
    fi
fi

index=1$ballY\0$ballX
if [ ${briks[$index]} ] && [ ${briks[$index]} == 1 ]
then
    tput cup $ballY $ballX
    drawBall
    clearBall
    ballDY=$((-ballDY))
    ballY=$((ballY))
    briks[$index]=0
fi

drawBall
tput setaf 7
tput cup $((bottom + 2)) $left
if [ $lifes -gt 0 ]
then
    echo "Lifes: " $lifes "
else
    echo "Game over
    exitGame
fi
}

function calcPlateS {
    plateS="+"
    for (( i = 0; i < $((plateW - 2)); i++ ))
    do
        plateS+="-"
    done
}

```



```
    plateS+=" "
}

function drawPlate {
    tput setaf 2
    tput cup $((bottom - 1)) $((plateX - 1))
    echo $plateS
    if [ $plateX == $left ]
    then
        tput setaf 241
        tput cup $((bottom - 1)) $((plateX - 1))
        echo "|"
    fi
}

function exitGame {
    echo "Goodbye!"
    trap exit ALRM
    tput cnorm
    IFS="$OLD_IFS"
    tput cvvis
    stty echo
    exit 0
}

function startGame {
    if [ $lives -ge 0 ]
    then
        state='playing'
    else
        state='gameOver'
    fi
}

trap move ALRM

calcPlateS
drawBricks
drawBorder
drawBall
drawPlate
resetBall
move

while :
```



```
do
  read -s -n 1 key
  case "$key" in
    a)
      if [ $state == 'stop' ]
      then
        ballDX=-1
        startGame
      else
        if [ $plateX -gt $left ]
        then
          plateX=$((plateX - 1))
          drawPlate
          drawBall
        fi
      fi
      ;;
    d)
      if [ $state == 'stop' ]
      then
        ballDX=1
        startGame444
      else
        if [ $plateX -lt $((right - $plateW)) ]
        then
          plateX=$((plateX + 1))
          drawPlate
          drawBall
        fi
      fi
      ;;
    q)
      exitGame
      ;;
  esac
done
```

5.3 运行截图

(1) 运行前



(2) 运行中



5.4 体会

在本次实验中，我利用 shell 编程实现了一个 arkanoid 游戏，在编写过程中我使用了很多之前并未学习过的 linux 命令，也多次使用了之前并未使用过的 shell 编程下的函数定义与调用。通过本次实验，我对 shell 编程有了更加深刻的理解，对 linux 系统下的命令也有了更加全面的认识。

5.5 参考文献

- [1] Robert Love. LINUX 系统编程. 东南大学出版社, 2009.
- [2] 鸟哥. 鸟哥的 Linux 私房菜: 基础学习篇[M]. 人民邮电出版社, 2010.
- [3] 菜鸟教程. Linux Shell 教程. <https://www.runoob.com/linux/linux-shell.html>.