



暨南大学

本科生课程论文

课程名称 Linux 高级编程

学 院	<u>智能科学与工程学院</u>
学 系	<u></u>
专 业	<u>人工智能</u>
姓 名	<u>赵俊文</u>
学 号	<u>2022104002</u>
指导教师	<u>李军</u>

二〇二五年 六月十五日

目录

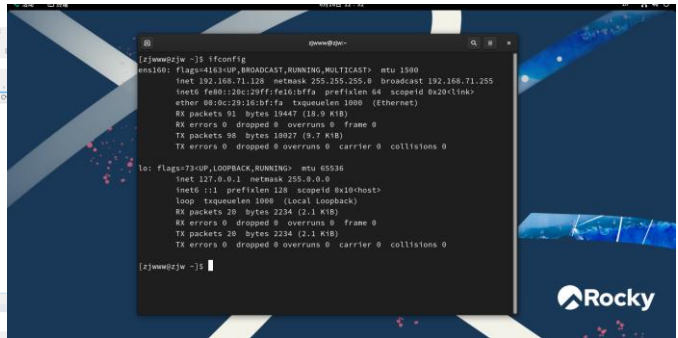
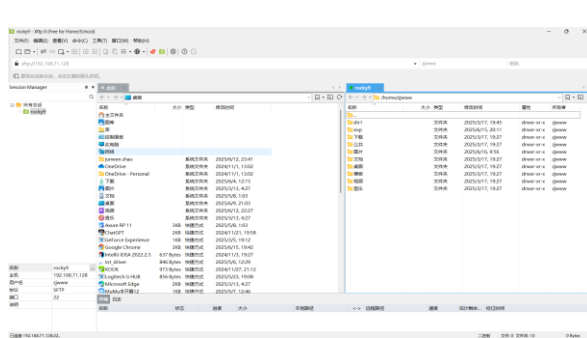
1.	LINUX 安装 (ROCKY9.5)	1
2.	常用命令	2
3.	常用开发工具	3
4.	用户与用户组管理	5
5.	分区与文件系统	7
6.	SHELL 编程-程序结构、变量、运算、IF 语句	9
7.	SHELL 编程-分支、循环	11
8.	底层文件 IO	13
9.	LINUX 进程管理	15
10.	LINUX 信号	17
11.	进程通信-管道	18
12.	进程通信-消息队列	19
13.	进程、线程对比	21
14.	信号量实现生产者-消费者模型	23
15.	并行下载	24

1. Linux 安装 (Rocky9.5)

在 VMware 中新建虚拟机，选中提前准备好的 Rocky9.5 光盘映像文件，跟着安装向导安装即可。安装完成进入 Linux 桌面：



在 Windows 上安装 Xftp 并传输文件，使用 ifconfig 查找虚拟机 IP 地址。连接 Windows 主机和 Linux 虚拟机。可以直接拖动文件进行传输。



2. 常用命令

文件操作命令

使用 `mkdir` 命令创建两层目录结构 (`dir1/dir2`), 使用 `touch` 命令在 `dir2` 中创建文件 (`file1.txt`)

复制文件: 使用 `cp` 命令将 `file1.txt` 复制到 `dir1`

删除文件: 使用 `rm` 命令删除 `dir1/file1.txt`

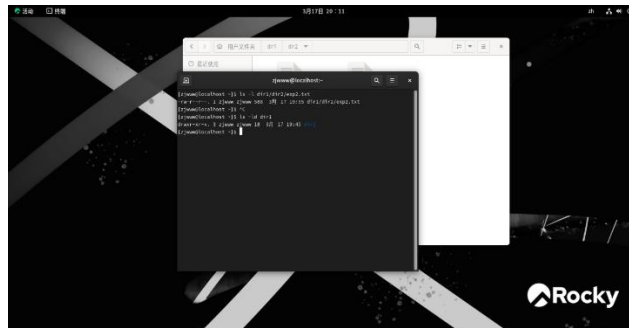
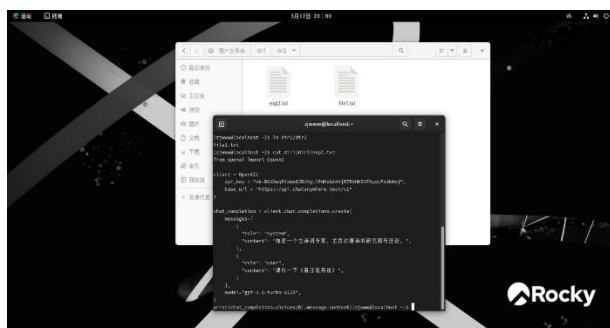
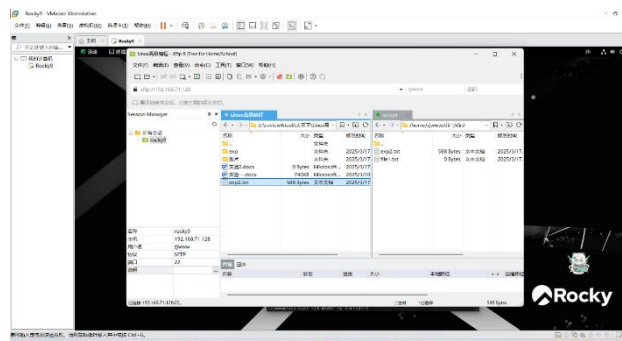
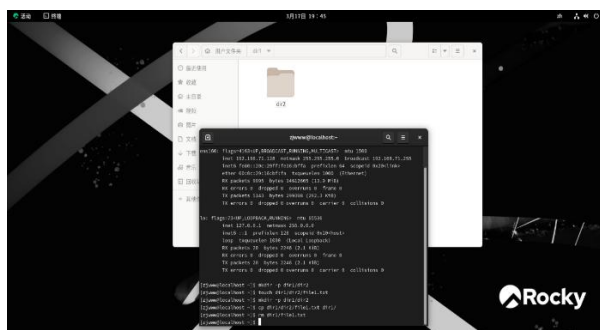
文件查看命令

将 Windows 的 `exp2.txt` 从 Windows 拖到 Linux 的 `dir1/dir2` 目录

在终端中使用 `cat` 或 `less` 查看文件内容

使用 `ls -ld` 查看目录权限, 可以看到该目录为所有者有读、写和执行权限

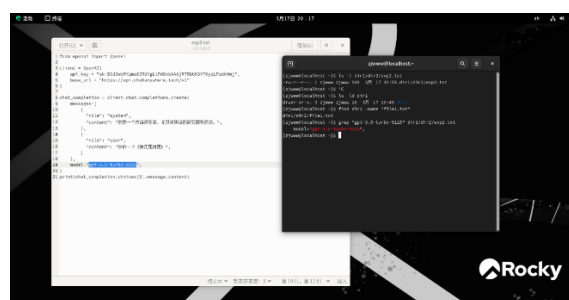
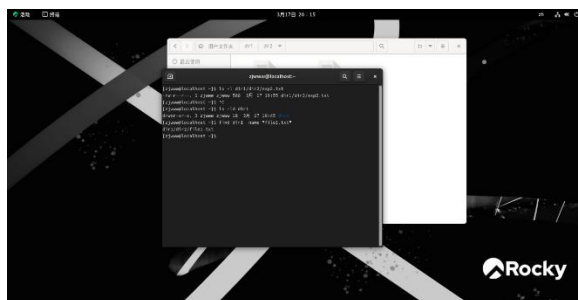
使用 `ls -l` 查看文件权限, 可以看到该文件为所有者可读写



使用 `find` 和 `grep`

Find 命令: 借助搜索关键字 (文件名、文件大小、文件所有者等) 查找文件或目录。

Grep 命令: 在文件中搜索与字符串匹配的行并输出。



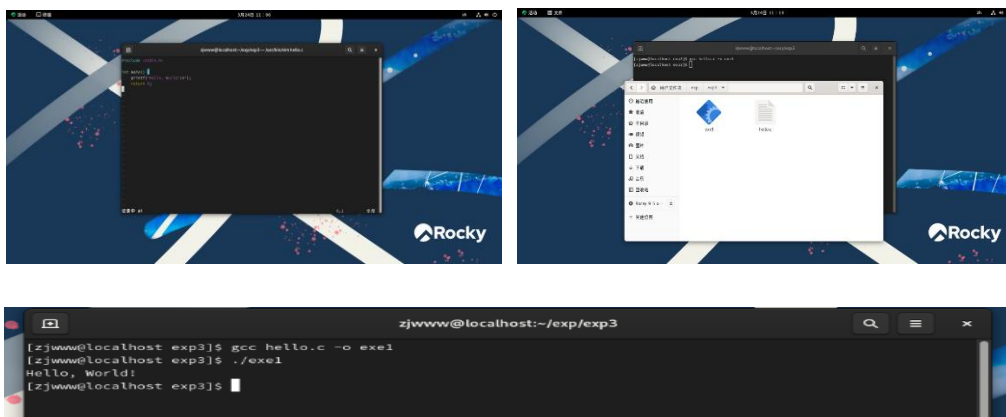
3. 常用开发工具

编写单个文件，单文件编译

打开终端，使用 vi 编辑器创建一个 C 语言源文件，命名为 `hello.c`，并编写一个简单的程序。进入插入模式使用：wq 保存退出。

保存并退出 vi 编辑器，使用 gcc 编译 `hello.c` 并指定输出文件名为 `exe1`。

运行生成的可执行文件 `exe1`



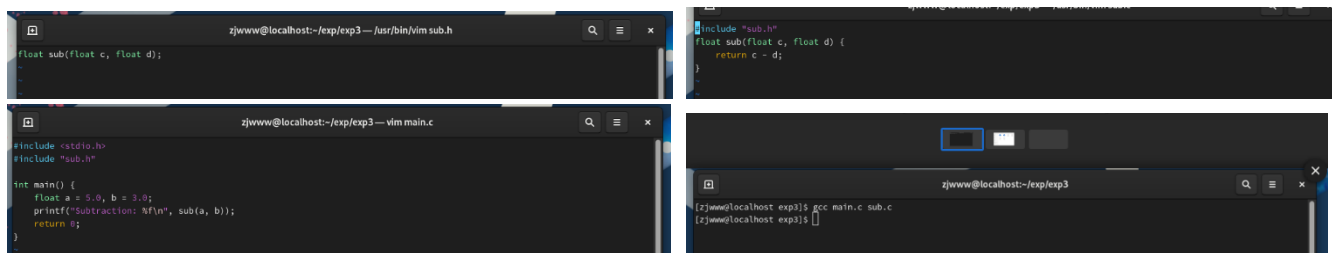
编写多个文件，完成编译

创建 `sub.h` 头文件，并在其中声明函数 `sub`，之后保存并退出

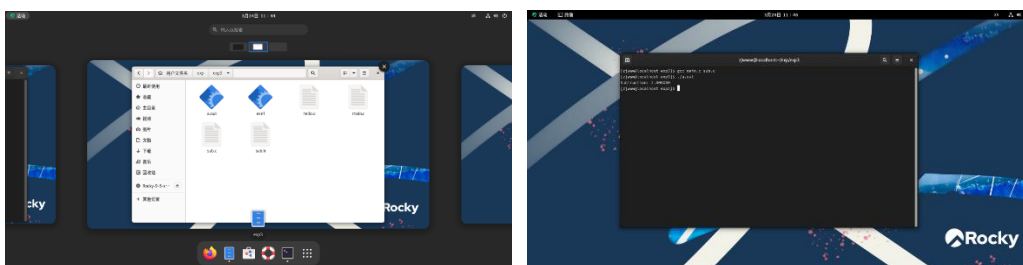
创建 `sub.c`，定义减法函数，之后保存并退出

创建 `main.c` 文件，定义主函数文件，调用 `add` 函数。之后保存并退出

使用 gcc 编译所有源文件，并生成默认的可执行文件（默认命名为 `a.out`）



执行该文件，输出 $5.0 - 3.0 = 2.0$ 的结果



gdb 调试

编译教材 2-39 程序，并加上 -g 选项，以便生成调试信息。生成名为 `debug` 的执行文件

把断点设置在第 16 行，进行调试

```

[jzwww@localhost exp3] gcc main.c sub.c
[jzwww@localhost exp3] ./a.out
Subtraction: 2.000000
[jzwww@localhost exp3] vi demo.c
[jzwww@localhost exp3] gcc -g demo.c -o debug
[jzwww@localhost exp3]

[jzwww@localhost ~] gdb ./demo.debug
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs>
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./demo.debug...
(gdb) break 16
Breakpoint 1 at 0x401174: file demo.c, line 16.
(gdb) run
Starting program: /home/jzwww/exp3/debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
arr[0]=11
arr[1]=10
arr[2]=1
arr[3]=15
arr[4]=1
Breakpoint 1, select_sort (arr=0x7fffffffe030, len=5) at demo.c:16
16      k=j;
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.34-125.el9_5.3.x86_64
(gdb)

```

程序在第 16 行 ($k=j$) 使用 `next` 命令逐步执行代码，观察循环次数，直到出现异常

```

[jzwww@localhost ~] gdb ./demo.debug
Breakpoint 1, select_sort (arr=0x7fffffffe030, len=5) at demo.c:16
16      k=j;
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.34-125.el9_5.3.x86_64
(gdb) next
17      for (j = 1; j < len; j++)
(gdb) next
18          if (arr[k] < arr[j])
(gdb) next
19              swap(arr[k], arr[j]);
Program received signal SIGSEGV, Segmentation fault.
16      k=j;
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.34-125.el9_5.3.x86_64
Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb)

```

可以看到在 $i=0$ 也就是第一次循环时就发生异常。

删除之前的断点，设置条件断点 (`break 16 if i==0`) 以便让在出现异常之前就停下来，然后再次运行。

```

[jzwww@localhost ~] gdb ./demo.debug
(gdb) break 16 if i==0
Breakpoint 5 at 0x401174: file demo.c, line 16.
(gdb)

[jzwww@localhost ~] gdb ./demo.debug
(gdb) break 16 if i==0
Breakpoint 5 at 0x401174: file demo.c, line 16.
(gdb) run
Starting program: /home/jzwww/exp3/debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
arr[0]=11
arr[1]=10
arr[2]=1
arr[3]=15
arr[4]=1
Breakpoint 5, select_sort (arr=0x7fffffffe030, len=5) at demo.c:16
16      k=j;
(gdb) Quit
(gdb)

```

分析程序：根据调试结果可以分析出这段代码在选择排序函数的 $i = 0$ （第一次循环）时就会报错，因为 $k = j$ ；这一行中 j 尚未被初始化。将 $k=j$ ；改为 $k=i$ ；，代码将正确运行，并且能够对数组进行选择排序。

4. 用户与用户组管理

创建用户、用户组

进入 root，使用 'useradd' 添加三个用户，并使用 tail 命令查看，使用 'groupadd' 创建用户组 group1

```
[zjwww@localhost ~]$ sudo useradd user1

我们信任您已经从系统管理员那里了解了日常注意事项。
总结起来不外乎这三点：

#1) 尊重别人的隐私。
#2) 输入前要先考虑(后果和风险)。
#3) 权力越大，责任越大。

[sudo] zjwww 的密码:
[zjwww@localhost ~]$ sudo useradd user2
[zjwww@localhost ~]$ sudo useradd user3
[zjwww@localhost ~]$ sudo useradd user3
[sudo] zjwww 的密码:
[zjwww@localhost ~]$
```

```
我们信任您已经从系统管理员那里了解了日常注意事项。
总结起来不外乎这三点：

#1) 尊重别人的隐私。
#2) 输入前要先考虑(后果和风险)。
#3) 权力越大，责任越大。

[sudo] zjwww 的密码:
[zjwww@localhost ~]$ sudo useradd user2
[zjwww@localhost ~]$ sudo useradd user3
[zjwww@localhost ~]$ sudo groupadd group1
[sudo] zjwww 的密码:
[zjwww@localhost ~]$
```

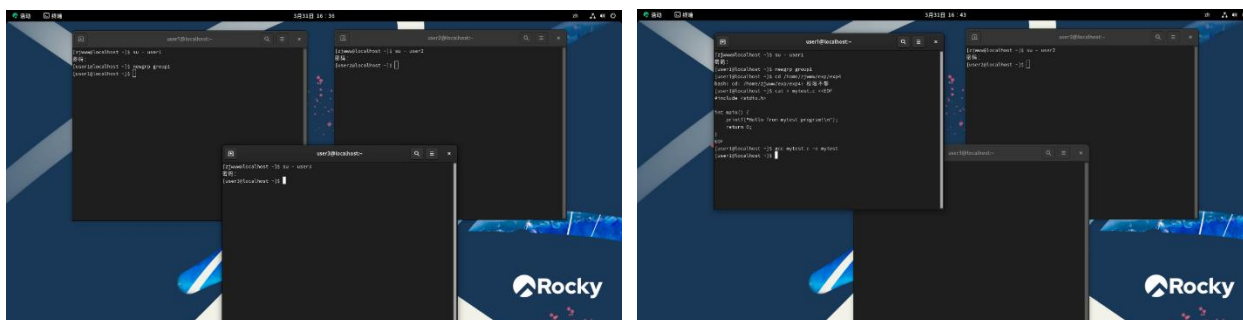
将 user1 和 user2 添加到 group1 组，使用 id 查看用户组信息

```
zjwww@localhost~
[zjwww@localhost ~]$ sudo usermod -aG group1 user1
[zjwww@localhost ~]$ sudo usermod -aG group1 user2
[zjwww@localhost ~]$
```

修改当前组并编写 C 程序

修改当前组为 group1

编写、编译 C 程序为可执行文件 mytest



修改文件权限并移动文件

修改组权限为---x--x---，之后查看文件详细属性

```
EOF
[user1@localhost ~]$ gcc mytest.c -o mytest
[user1@localhost ~]$ chmod 110 mytest
[user1@localhost ~]$
```

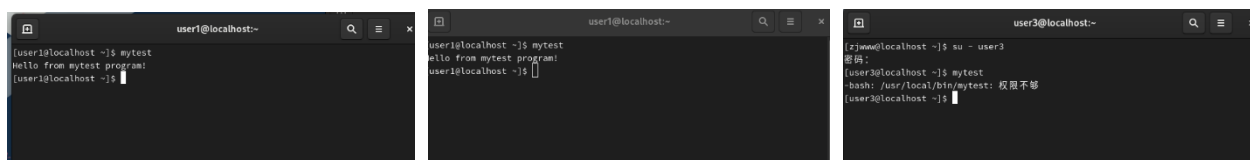
```
EOF
[user1@localhost ~]$ gcc mytest.c -o mytest
[user1@localhost ~]$ chmod 110 mytest
[user1@localhost ~]$ ls -l mytest
---x--x---. 1 user1 group1 17448  3月 31 16:43 mytest
[user1@localhost ~]$
```

移动文件到/usr/local/bin 中

```
[user1@localhost ~]$ sudo mv mytest /usr/local/bin/
[sudo] user1 的密码:
[user1@localhost ~]$ mytest
Hello from mytest program!
```

运行 mytest 程序

分别在在 user1, user2,user3 终端中直接运行



可以看到，User1 可以直接运行，User2 也可以直接运行，User3 不能运行，提示权限不够

a) 为什么 user1 可以直接运行？

- /usr/local/bin 目录在系统的 PATH 环境变量中
- 当输入命令时，系统会在 PATH 指定的目录中查找可执行文件
- 不需要 ./ 前缀是因为当前目录(.)通常不在 PATH 中

b) 为什么 user2 可以直接运行

- user2 在 group1 组中
- 文件有组执行权限(x)

c) 为什么 user3 不可以直接运行（权限被拒绝）？

- user3 不在 group 组中
- 其他用户没有任何权限

5. 分区与文件系统

启动 linux 后, 用 `lsblk` 命令查看机器上的磁盘 (块设备), 截图。然后关闭 linux (命令为 `shutdown -h now`)

1. 在安装了 linux 的虚拟机上, 再添加一块虚拟硬盘。重新启动 linux 之后, 再用 `lsblk` 命令查看磁盘, 观察机器上新增的硬盘, 观察它的名称

```
[zjwww@localhost exp7]$ lsblk
NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
sr0          8:0    0  20G  0 disk 
nvme0n1     259:0    0   60G  0 disk 
nvme0n1p1  259:1    0  600M  0 part /boot/efi
nvme0n1p2  259:2    0    1G  0 part /boot
nvme0n1p3  259:3    0  58.4G  0 part 
  └─rl-root 253:0    0  36.6G  0 lvm /
    └─rl-swap 253:1    0   3.9G  0 lvm [SWAP]
      └─rl-home 253:2    0  17.9G  0 lvm /home

[zjwww@localhost exp7]$
```

```
[zjwww@localhost exp7]$ lsblk
NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
sda          8:0    0  20G  0 disk 
sr0          11:0    1  10.7G  0 rom  /run/media/zjwww/Rocky-9-5-x86_64-dvd
nvme0n1     259:0    0   60G  0 disk 
nvme0n1p1  259:1    0  600M  0 part /boot/efi
nvme0n1p2  259:2    0    1G  0 part /boot
nvme0n1p3  259:3    0  58.4G  0 part 
  └─rl-root 253:0    0  36.6G  0 lvm /
    └─rl-swap 253:1    0   3.9G  0 lvm [SWAP]
      └─rl-home 253:2    0  17.9G  0 lvm /home

[zjwww@localhost exp7]$
```

添加硬盘后发现新增 `sdb` 硬盘, 大小为 20G。

用 `gdisk` 命令把新添加的硬盘分成 2 个区, 观察它们的名称是什么?

```
[zjwww@localhost exp7]$ sudo fdisk /dev/sdb
欢迎使用 fdisk (util-linux 2.37.4).
更改将停留在内存中, 直到您决定将更改写入磁盘。
使用写入命令前请三思。

命令(输入 m 获取帮助): d
已选择分区 2
分区 2 已删除。

命令(输入 m 获取帮助): n
分区号 (1-128, 默认 1): 1
第一个扇区 (34-41943006, 默认 2048):
最后一个扇区, +/-sectors 或 +size{K,M,G,T,P} (2048-41943006, 默认 41943006): +10G
创建了一个新分区 1, 类型为“Linux filesystem”, 大小为 10 GiB。

命令(输入 m 获取帮助): n
分区号 (2-128, 默认 2): 2
第一个扇区 (20973568-41943006, 默认 20973568):
最后一个扇区, +/-sectors 或 +size{K,M,G,T,P} (20973568-41943006, 默认 41943006):
创建了一个新分区 2, 类型为“Linux filesystem”, 大小为 10 GiB。

命令(输入 m 获取帮助): w
分区表已调整。
将调用 ioctl() 来重新读写分区表。
正在同步磁盘。

[zjwww@localhost exp7]$
```

```
[zjwww@localhost exp7]$ lsblk | grep sda
sda          8:0    0  20G  0 disk 
├─sda1       8:1    0  10G  0 part 
└─sda2       8:2    0  10G  0 part 

[zjwww@localhost exp7]$
```

用 `mkfs.xfs` 命令, 把上面的 2 个分区格式化为 XFS 文件系统 `lsblk` 命令可以列出所有可用的块设备的信息, 包括它们的文件系统类型, `-f` 选项告诉 `lsblk` 显示文件系统信息。输出如下

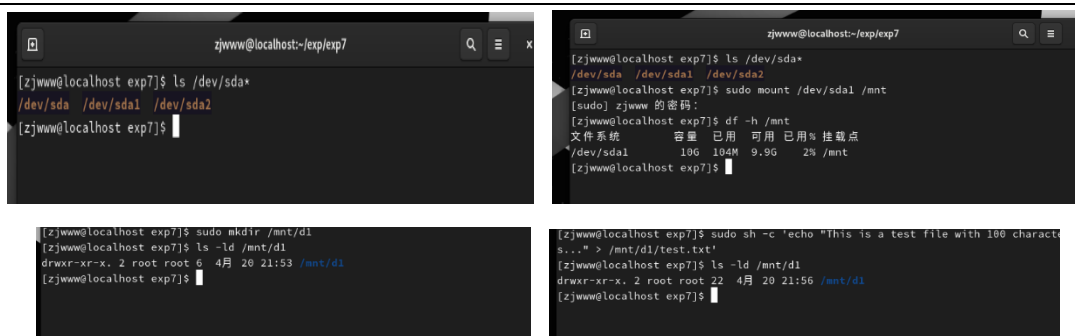
```
[zjwww@localhost exp7]$ sudo mkfs.xfs /dev/sda1
meta-data=/dev/sda1            isize=512    agcount=4, agsize=655360 blks
                        =       sectsz=512    attr=2, projid32bit=1
                        =       crc=1        finobt=1, sparse=1, rmapbt=0
                        =       reflink=1    bigtime=1 inobtcount=1 nrext64=0
data      =                   bsize=4096    blocks=2621440, imaxpct=25
                        =                   sunit=0    swidth=0 blks
naming    =version 2          ascii-ci=0, ftype=1
log       =internal log      bsize=4096    blocks=16384, version=2
                        =       sectsz=512    sunit=0 blks, lazy-count=1
realtime  =none              extsz=4096    blocks=0, rtextents=0

[zjwww@localhost exp7]$ sudo mkfs.xfs /dev/sda2
meta-data=/dev/sda2            isize=512    agcount=4, agsize=655295 blks
                        =       sectsz=512    attr=2, projid32bit=1
                        =       crc=1        finobt=1, sparse=1, rmapbt=0
                        =       reflink=1    bigtime=1 inobtcount=1 nrext64=0
data      =                   bsize=4096    blocks=2621179, imaxpct=25
                        =                   sunit=0    swidth=0 blks
naming    =version 2          ascii-ci=0, ftype=1
log       =internal log      bsize=4096    blocks=16384, version=2
                        =       sectsz=512    sunit=0 blks, lazy-count=1
realtime  =none              extsz=4096    blocks=0, rtextents=0

[zjwww@localhost exp7]$ lsblk -f
NAME FSTYPE FSVER LABEL UUID                                 FSAVAIL FSUSE% MOUNTPOINTS
├─sda
│   └─xfs
│       └─sda1
│           └─xfs
│               └─sda2
│                   └─xfs
│                       └─sr0 iso9660 Jolie Rocky-9-5-x86_64-dvd 0 100% /run/media/zjwww/Rocky-9-5-x86_64-dvd
├─nvme0n1
│   └─nvme0n1p1 vfat FAT32 16D5-484A 591.8M 1% /boot/efi
│   └─nvme0n1p2 xfs 3ff529e8-7fa5-4f02-a9c5-e4aa321d5ef9 628.7M 35% /boot
│   └─nvme0n1p3 LVM2_m LVM2 1yp0Ar-R9bq-ee3C-b3aP-d9g6-IVks-sBL0zG
│       └─rl-root xfs 52a7b8aa-d7ca-49fa-a2a6-c5409bba285e 31G 15% /
│       └─rl-swap swap 1 be38759f-d391-40ec-aede-b4c2b799aebb [SWAP]
│       └─rl-home xfs c4e06e99-f353-4850-9c8a-6214dc61abdb 17.6G 1% /home

[zjwww@localhost exp7]$
```

查看目录 `/dev` 下面是否有上述两个分区, 然后把其中一个分区挂载到 `/mnt` 目录下。在 `/mnt` 目录下, 建立一个目录 `d1`, 用 `ls` 命令查看其大小; 然后在 `d1` 里建立一个 100 字符左右的文件, 用 `ls` 命令再次查看 `d1` 的大小;

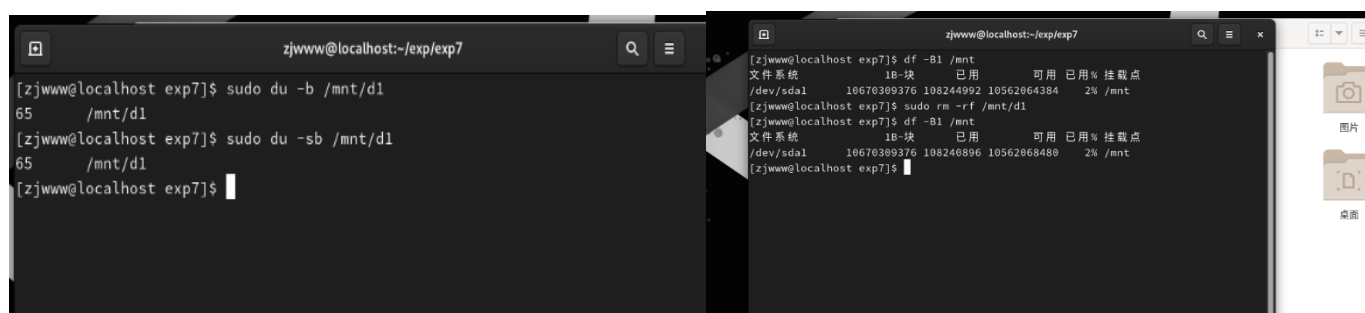


```
[zjwww@localhost exp7]$ ls /dev/sda*  
/dev/sda /dev/sda1 /dev/sda2  
[zjwww@localhost exp7]$  
[zjwww@localhost exp7]$ sudo mkdir /mnt/d1  
[zjwww@localhost exp7]$ ls -ld /mnt/d1  
drwxr-xr-x. 2 root root 6 4月 20 21:53 /mnt/d1  
[zjwww@localhost exp7]$  
[zjwww@localhost exp7]$ sudo sh -c 'echo "This is a test file with 100 characters..." > /mnt/d1/test.txt'  
[zjwww@localhost exp7]$ ls -ld /mnt/d1  
drwxr-xr-x. 2 root root 22 4月 20 21:56 /mnt/d1  
[zjwww@localhost exp7]$  
[zjwww@localhost exp7]$ df -h /mnt  
文件系统 容量 已用 可用 已用% 挂载点  
/dev/sda1 19G 104M 9.9G 2% /mnt  
[zjwww@localhost exp7]$
```

对比发现 d1 目录大小无显著变化，这是因为当使用 `ls -l` 命令时，显示的通常是文件的“大小”字段，它表示的是文件的内容所占用的字节数。但是，对于目录而言，这个“大小”字段实际上显示的是目录本身元数据的大小，而不是目录内所有文件内容的总大小。

在 d1 目录中创建一个约 100 字符的文件。由于字符文件的实际大小通常比字符数要大，所以文件实际占用的空间可能远大于 100 字节。但是，即使文件的大小超过 100 字节，d1 目录本身的大小可能仍然不会有显著变化，因为目录的大小主要与其内部文件和子目录的引用有关，而不是这些文件内容的大小。

用 `du` (加 `-b` 表示以字节为单位显示) 显示 d1 目录的大小，这个命令将会递归地计算 /mnt/d1 目录下所有文件和子目录的大小，并以字节为单位显示。输出将是一个数字，表示 d1 目录及其所有内容的总大小。用 `df` 命令 (加 `-B1` 选项表示以 1byte 为单位) 显示 d1 所在分区已使用空间；然后删除 d1，再令查看该分区的已使用空间，前后对比说明新建的文件确实在你新添加的硬盘上。



```
[zjwww@localhost exp7]$ sudo du -b /mnt/d1  
65 /mnt/d1  
[zjwww@localhost exp7]$ sudo du -sb /mnt/d1  
65 /mnt/d1  
[zjwww@localhost exp7]$  
[zjwww@localhost exp7]$ df -B1 /mnt  
文件系统 1B-块 已用 可用 已用% 挂载点  
/dev/sda1 10670309376 108244992 10562064384 2% /mnt  
[zjwww@localhost exp7]$ sudo rm -rf /mnt/d1  
[zjwww@localhost exp7]$ df -B1 /mnt  
文件系统 1B-块 已用 可用 已用% 挂载点  
/dev/sda1 10670309376 108246896 10562064480 2% /mnt  
[zjwww@localhost exp7]$
```

可以看到删除文件后，可用值减少（约 100 字节），证明新建的文件确实在新添加的硬盘上

6. Shell 编程-程序结构、变量、运算、if 语句

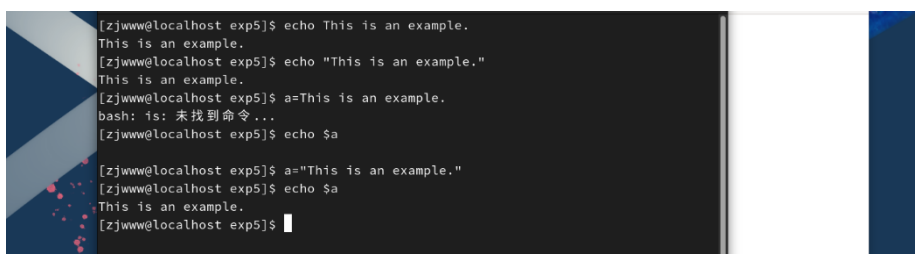
echo 同时输出多变量输出 echo This is an example 和 echo "This is an example."



```
[zjwww@localhost exp5]$ echo This is an example.
This is an example.
[zjwww@localhost exp5]$ echo "This is an example."
This is an example.
[zjwww@localhost exp5]$
```

在大多数情况下，输出效果是相同的。区别在于，第一种形式参数之间可以有多个空格，但输出时会被压缩为一个空格。第二种形式引号内的内容会原样输出，包括空格

变量赋值比较



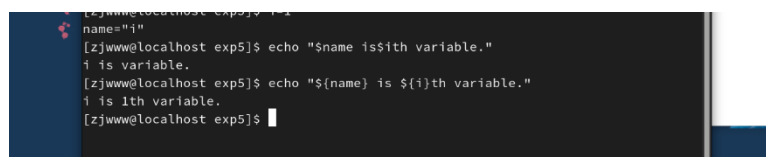
```
[zjwww@localhost exp5]$ echo This is an example.
This is an example.
[zjwww@localhost exp5]$ echo "This is an example."
This is an example.
[zjwww@localhost exp5]$ a=This is an example.
bash: is: 未找到命令...
[zjwww@localhost exp5]$ echo $a

[zjwww@localhost exp5]$ a="This is an example."
[zjwww@localhost exp5]$ echo $a
This is an example.
[zjwww@localhost exp5]$
```

可以看到两种赋值方式的输出不同，第一种赋值方式会报错

- ✓ 赋值时如果不加引号，空格会被解释为命令分隔符
- ✓ 引用变量时最好加双引号，可以保留字符串中的空格和特殊字符
- ✓ 对于包含空格的字符串，赋值时必须使用引号

变量引用



```
[zjwww@localhost exp5]$ name=i
[zjwww@localhost exp5]$ echo "$name is$ith variable."
1 is variable.
[zjwww@localhost exp5]$ echo "$(name) is ${i}th variable."
1 is 1th variable.
[zjwww@localhost exp5]$
```

第一个语句 echo "\$name is\$ith variable."有问题，因为\$ith 会被解释为一个名为"ith"的变量，而不是"i"变量后跟"th";第二个语句正确使用了\${}来明确变量边界，\${name}明确表示引用 name 变量\${i}明确表示引用 i 变量后面的"th"不会被误认为是变量名的一部分

调用参数

编写一个脚本，假设它的文件名为 t2, 执行时给它几个参数，假设为 3, (1) 在 t2 中添加 echo "The price is \$1.", echo 'The price is \$1.' 添加 echo \$*, echo \$@。

```

zjwww@localhost:~/exp/exp5
[zjwww@localhost exp5]$ vim t2
[zjwww@localhost exp5]$ chmod +x t2
[zjwww@localhost exp5]$ ./t2 a b c
./t2 has 3 parameters.
[zjwww@localhost exp5]$ echo "The price is $1."
The price is .
[zjwww@localhost exp5]$ echo 'The price is $1.'
The price is $1.
[zjwww@localhost exp5]$

```

发现输出没有区别

算术运算：let 方式和()方式

```

zjwww@localhost:~/exp/exp5
zjwww@localhost exp5$ vim calc.sh
zjwww@localhost exp5$ vim calc.sh
zjwww@localhost exp5$ let a=(b-3)*2
cho "let result: $a"
et result: -6
zjwww@localhost exp5$

```

```

www@localhost exp5$ vim calc.sh
www@localhost exp5$ vim calc.sh
www@localhost exp5$ let a=(b-3)*2
o "let result: $a"
result: -6
www@localhost exp5$

```

用\$(comand) 执行命令并返回该命令的输出。没有生成目录时的结果和创建 fi 文件后的结果：

```

#!/bin/bash
if [ ! -d "$1" ]; then
    echo "File not exist!"
    exit 1
fi

filename=$1
extension=${filename##*.}

if [ "$filename" = "$extension" ]; then # 无扩展名
    # 获取月份 (使用stat更可靠)
    month=$(stat -c %Y "$filename" | awk '{print $2}')
    # 或名是ls查询: month=$(ls -l "$filename" | awk '{print $6}')
    mv -v "$filename" "${filename}.${month}"
else
    mv -v "$filename" "${filename}.${extension}"
fi

```

```

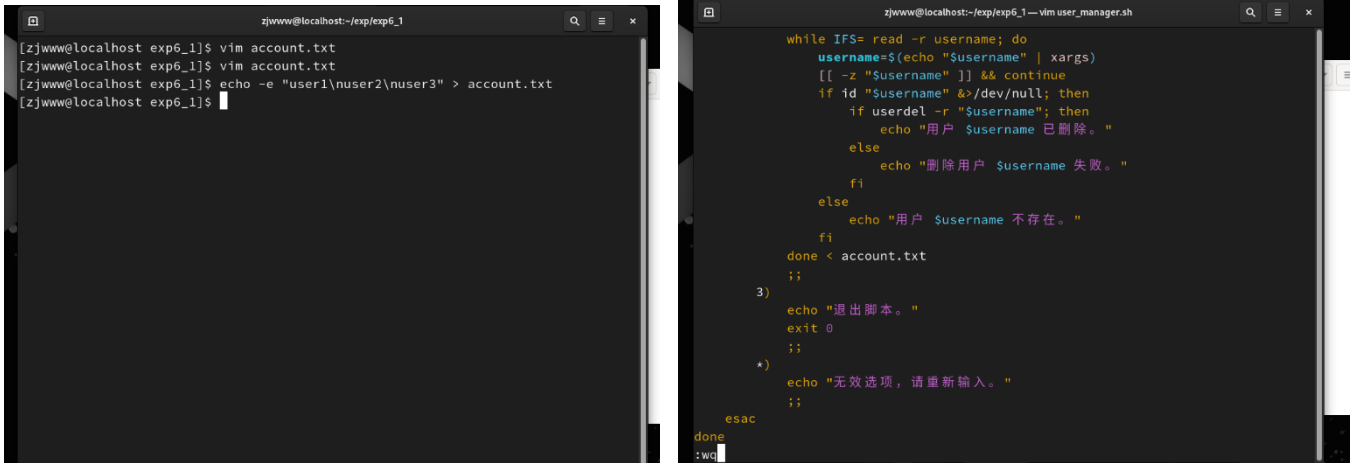
zjwww@localhost exp5$ touch testfile
zjwww@localhost exp5$ ./exp5.sh testfile
bash: ./exp5.sh: 没有那个文件或目录
zjwww@localhost exp5$ ./exp5.sh testfile
已重命名 'testfile' -> 'testfile.16114720.546893145'
zjwww@localhost exp5$ touch file.txt
zjwww@localhost exp5$ touch file.txt
zjwww@localhost exp5$ ./exp5.sh testfile
file not exist!
zjwww@localhost exp5$ ./exp5.sh file.txt
已重命名 'file.txt' -> 'file'
zjwww@localhost exp5$

```

7. Shell 编程-分支、循环

7.1 批量添加/删除用户

部分脚本代码

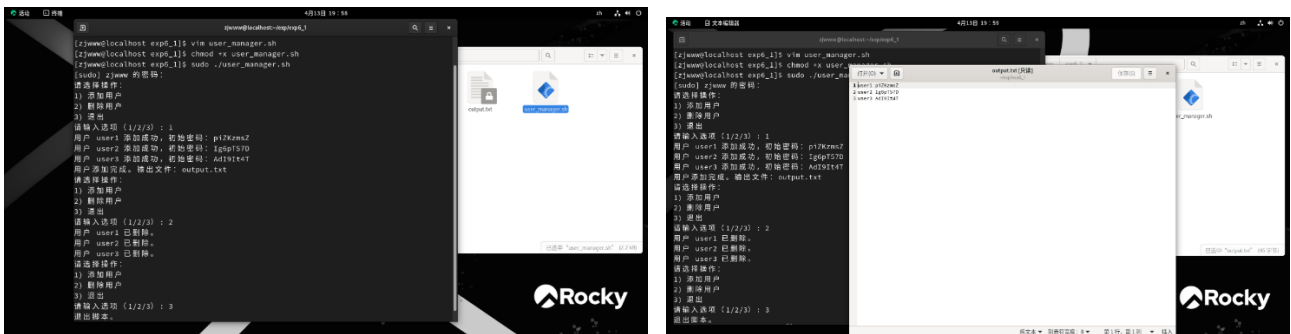


```

[jzwww@localhost exp6_1]$ vim account.txt
[jzwww@localhost exp6_1]$ vim account.txt
[jzwww@localhost exp6_1]$ echo -e "user1\nuser2\nuser3" > account.txt
[jzwww@localhost exp6_1]$

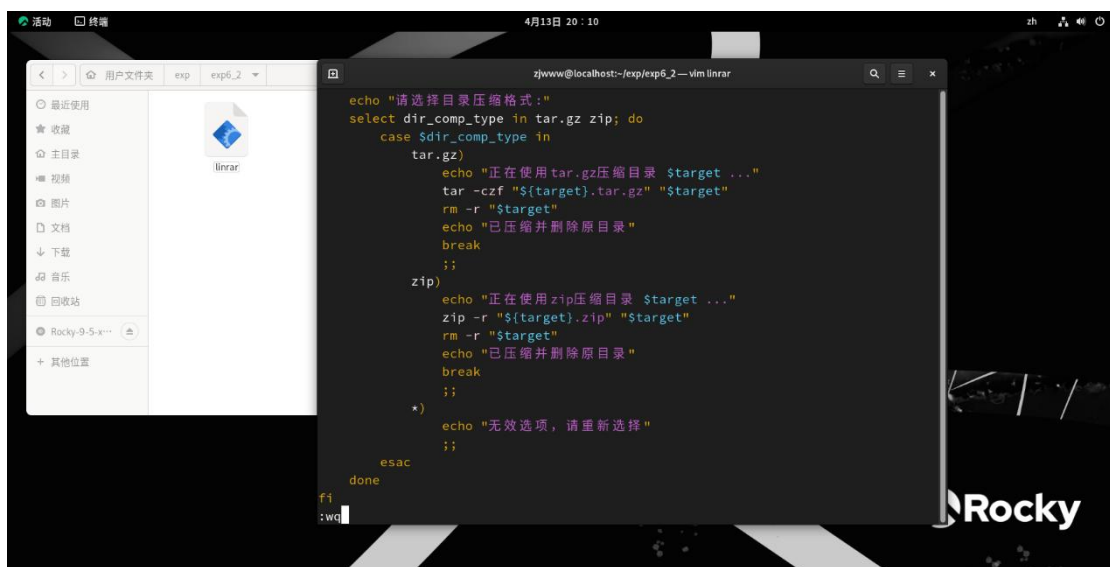
jzwww@localhost:~/exp/exp6_1 — vim user_manager.sh
while IFS= read -r username; do
    username=$(echo "$username" | xargs)
    [[ -z "$username" ]] && continue
    if id "$username" &>/dev/null; then
        if userdel -r "$username"; then
            echo "用户 $username 已删除。"
        else
            echo "删除用户 $username 失败。"
        fi
    else
        echo "用户 $username 不存在。"
    fi
done < account.txt
;;
3)
echo "退出脚本。"
exit 0
;;
*)
echo "无效选项，请重新输入。"
;;
esac
done
:wq
  
```

运行结果

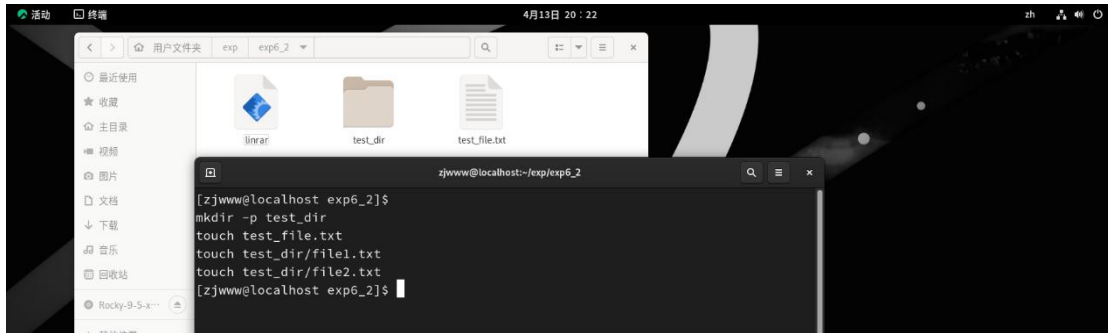


7.2 压缩解压脚本 linrar

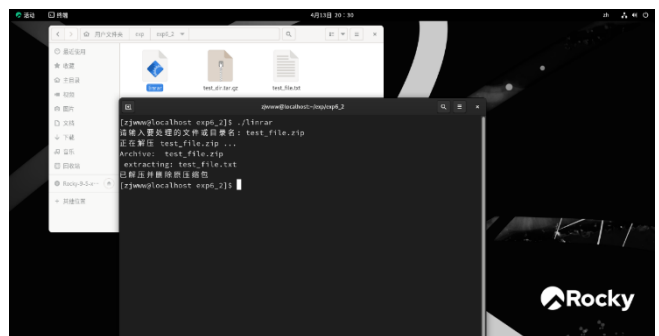
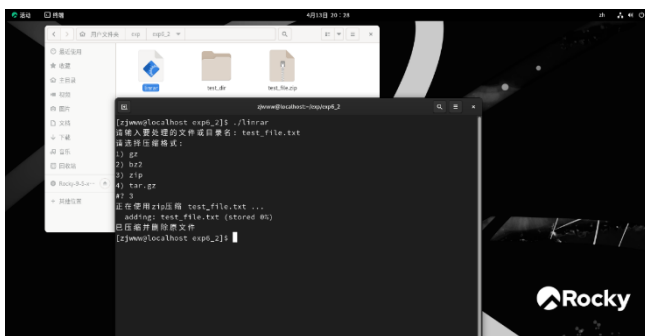
编写合适的脚本代码



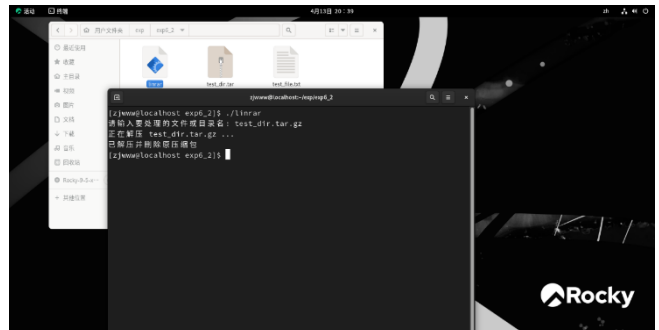
创建测试文件



压缩/解压文件及结果

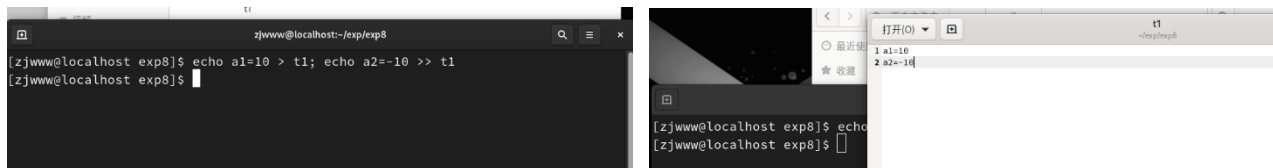


压缩/解压目录



8. 底层文件 IO

在命令行运行 `echo a1=10 >t1;echo a2=-10 >> t1`. 请你说明此命令的含义



这个命令的含义是创建一个文件 `t1`, 然后将字符串 `"a1=10"` 写入文件 `t1` 中, 并覆盖文件原有内容; 再将字符串 `"a2=-10"` 追加到文件 `t1` 的末尾。

用 `vi` 打开 `t1`, 然后在命令模式下, 输入命令 `!xxd`, 让 `vi` 以 16 进制显示该文件内容。(`!xxd -r` 可返回正常显示)



请你说明 `6131..3130 0a` 这些 16 进制数分别表示什么含义。可以借助附件中的 ASCII 码表。该文件占 13 字节的空间, 为什么?

a) 16 进制解释:

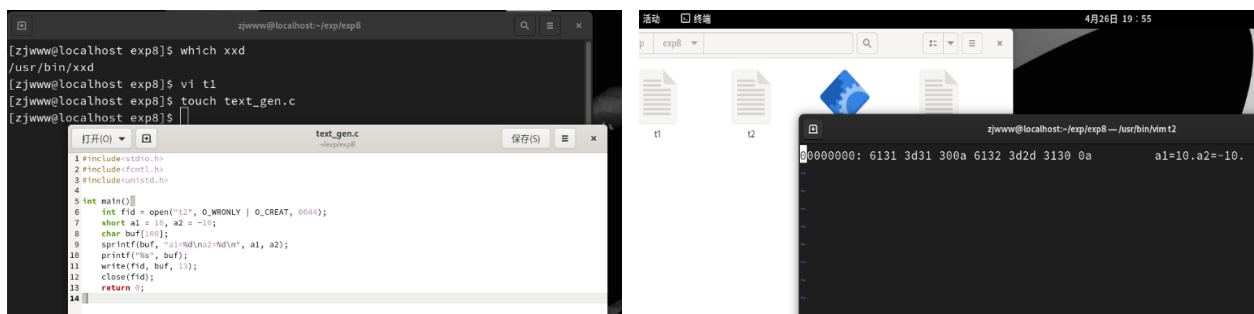
- `61 = 'a'; 31 = '1'; 3d = '='; 31 = '1'`
- `30 = '0'; 0a = 换行符'\n'; 61 = 'a'; 32 = '2'`
- `3d = '='; 2d = '-'; 31 = '1'; 30 = '0'`

b) 文件大小是 13 字节, 因为:

- 第一行 `a1=10\n` 占 6 字节 (5 字符 + 1 换行符)。
- 第二行 `a2=-10\n` 占 7 字节 (6 字符 + 1 换行符)。

两行合计: $6 + 7 = 13$ 字节。

生成文本文件。用 `touch` 命令建立新的空文件 `t2`, 然后编写下面的程序, 它编译运行后, 在 16 进制下查看, `t2` 与 `t1` 的内容一样吗?



可以看到与 `t1` 的 16 进制查看完全相同

生成二进制文件。用 `touch` 命令建立新的空文件 `t3`,然后编写下面的程序，它编译运行后，在 16 进制下查看，`t3` 与 `t2` 的内容一样吗?请你将 `t3` 的 16 进制形式截图，并在图上标出 10，-10 这 2 个数字的补码范围。

The top screenshot shows a terminal window where a file `bin_gen.c` is created using `touch bin_gen.c`. The file content is shown in a separate window:

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4
5 int main() {
6     int fd = open("t3", O_WRONLY | O_CREAT, 0644);
7     short a1 = 10, a2 = -10;
8     write(fd, &a1, sizeof(a1));
9     write(fd, &a2, sizeof(a2));
10    write(fd, "\n", 1);
11    write(fd, &a2, sizeof(a2));
12    write(fd, &a1, sizeof(a1));
13    write(fd, "\n", 1);
14    close(fd);
15    return 0;
16 }
```

The bottom screenshot shows the hex dump of the file `t3` in vim:

```
00000000: 6131 3d0a 003a 6132 3df6 ffd0 a1=...a2=...
```

Red and blue boxes highlight the hex values `3d0a` and `ffd0` respectively.

可以看到在 16 进制下查看 `t3` 和 `t2` 的内容不一样，其中红色区域为 10 的补码，蓝色区域为 -10 的补码

按数据类型读取文件。将 `t3` 中的“`a2=`”的内容读取到变量 `s2`, -10 读取到变量 `n2` 中。将 `n2` 加 5 后，用下列语句打印输出。 `printf("%s%d\n", s2, n2);`

The left screenshot shows the source code of `read_bin.c`:

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4 #include <string.h>
5 int main() {
6     // 打开文件
7     int fd = open("t3", O_RDONLY);
8     if (fd == -1) {
9         perror("Failed to open file");
10        return 1;
11    }
12    // 计算"a2="的偏移量
13    // "a1=" (3) + short (2) + "\n" (1) = 6
14    off_t offset = 6;
15
16    // 定位到"a2="的位置
17    if (lseek(fd, offset, SEEK_SET) == -1) {
18        perror("Failed to seek");
19        close(fd);
20        return 1;
21    }
22    // 读取"a2="
23    char s2[3]; // 3字节+'\0'
24    if (read(fd, s2, 3) != 3) {
25        perror("Failed to read 'a2='");
26        close(fd);
27        return 1;
28    }
29    s2[3] = '\0'; // 添加字符串结束符
30    // 读取-10 (short类型)
31    short n2;
32    if (read(fd, &n2, sizeof(short)) != sizeof(short)) {
33        perror("Failed to read number");
34        close(fd);
35        return 1;
36    }
37    close(fd);
38    // 打印结果
39    printf("s2:\n", s2, n2);
40
41    return 0;
42 }
```

The right screenshot shows the execution of the program:

```
[zjwww@localhost exp8]$ touch read_bin.c
[zjwww@localhost exp8]$ gcc read_bin.c -o read_bin
[zjwww@localhost exp8]$ ./read_bin
a2=-10
[zjwww@localhost exp8]$
```


9. Linux 进程管理

9.1 进程同步

编写一个程序，main 函数里开启一个子进程。在 main 函数里输出 “world.”，子进程睡眠一秒后输出 “hello,”

```

zjwww@localhost:~/exp/exp9 — /usr/bin/vim sync1.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    pid_t pid = fork();

    if (pid == 0) {
        // 子进程
        sleep(1);
        printf("hello,");
    } else {
        // 父进程
        printf("world.");
    }

    return 0;
}

zjwww@localhost:~/exp/exp9
[zjwww@localhost exp9]$ vi sync1.c
[zjwww@localhost exp9]$ gcc sync1.c -o sync1
[zjwww@localhost exp9]$ ./sync1
world.[zjwww@localhost exp9]$ hello,

```

可以看见输出的顺序是 world hello，原因是在大多数操作系统中，进程的执行是并发的，这意味着两个进程的执行顺序是由操作系统的调度器决定的，而不是由程序的编写顺序决定的。

在父进程中使用 wait 函数后，等待子进程完成后，父进程再输出。观察屏幕输出。可以看到输出了正确的 hello world

```

zjwww@localhost:~/exp/exp9 — /usr/bin/vim sync2.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid = fork();

    if (pid == 0) {
        // 子进程
        sleep(1);
        printf("hello,");
    } else {
        // 父进程
        wait(NULL); // 等待子进程结束
        printf("world.");
    }

    return 0;
}

zjwww@localhost:~/exp/exp9
[zjwww@localhost exp9]$ vi sync2.c
[zjwww@localhost exp9]$ gcc sync2.c -o sync2
[zjwww@localhost exp9]$ ./sync2
hello,world.[zjwww@localhost exp9]$

```

- 这次会先看到 “hello,”，然后看到 “world.”
- 由于父进程等待子进程完成，输出顺序是确定的 “hello,world.”

9.2 exec 函数族

设你编写的源程序名为 exec.c，父子线程分别用 execl，execvp 来分别启动命令 grep，在 exec.c 中分别查找 “execl”，“execvp” 所在的行。输出结果中间夹杂着命令提示符没关系。

```
zjwww@localhost:~/exp/exp9 — /usr/bin/vim exec.c
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid = fork();

    if (pid == 0) {
        // 子进程 - 使用execl
        printf("Child process using execl:\n");
        execl("/bin/grep", "grep", "execl", "exec.c", NULL);
        perror("execl failed");
    } else {
        // 父进程 - 使用execvp
        wait(NULL); // 等待子进程完成

        printf("\nParent process using execvp:\n");
        char *args[] = {"grep", "execvp", "exec.c", NULL};
        execvp("grep", args);
        perror("execvp failed");
    }

    return 0;
}
```

```
zjwww@localhost:~/exp/exp9
[zjwww@localhost exp9]$ vi exec.c
[zjwww@localhost exp9]$ gcc exec.c -o exec
[zjwww@localhost exp9]$ ./exec
Child process using execl:
// 子进程 - 使用execl
printf("Child process using execl:\n");
execl("/bin/grep", "grep", "execl", "exec.c", NULL);
perror("execl failed");

Parent process using execvp:
// 父进程 - 使用execvp
printf("\nParent process using execvp:\n");
char *args[] = {"grep", "execvp", "exec.c", NULL};
execvp("grep", args);
perror("execvp failed");
[zjwww@localhost exp9]$
```

- 子进程会使用 execl 执行 grep 命令，查找 exec.c 文件中包含“execl”的行
- 父进程会使用 execvp 执行 grep 命令，查找 exec.c 文件中包含“execvp”的行

10. Linux 信号

定时器应用：用 `setitimer` 函数设置一个周期性的定时器, 每当定时时间到时, 按序输出 26 个字母之一。

通过 `sigaction` 函数设置 `SIGALRM` 信号的处理函数, 当定时器触发时, 信号处理函数仅对静态变量 `count` 进行计数累加, 避免在信号处理函数中使用不安全函数。

然后, 使用 `setitimer(insert_element\0\)_mer` 函数配置一个周期性定时器, 初始延迟 1 秒, 之后每隔 1 秒触发一次 `SIGALRM` 信号。在主循环中, 通过 `pause(insert_element\1\>()` 函数等待信号到来, 每次信号触发后, 根据 `count` 的值计算并输出对应的字母 (A-Z 循环), 同时调用 `fflush(stdout)` 确保输出立即刷新, 避免缓冲机制影响显示效果。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/time.h>

static int count = 0; // 记录触发次数

// SIGALRM 信号处理函数
void alarm_handler(int sig) {
    count++; // 每次信号触发时增加计数
}

int main() {
    struct sigaction sa;
    struct itimerval timer;

    // 设置信号处理函数
    sa.sa_handler = alarm_handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGALRM, &sa, NULL) == -1) {
        perror("sigaction");
    }

    // 设置定时器: 初始延迟 1 秒, 之后每隔 1 秒触发一次
    timer.it_value.tv_sec = 1;
    timer.it_value.tv_usec = 0;
    timer.it_interval.tv_sec = 1;
    timer.it_interval.tv_usec = 0;

    if (setitimer(ITIMER_REAL, &timer, NULL) == -1) {
        perror("setitimer");
        exit(1);
    }

    // 主循环
    while (1) {
        pause(); // 等待信号
        // 输出当前字母 (A-Z 循环)
        printf("%c", 'A' + (count % 26));
        fflush(stdout); // 立即刷新输出缓冲区
    }

    return 0;
}
```

运行结果

```
zjwww@localhost:~/exp/exp10 — /alphabet_timer
zjwww@localhost exp10$ vi alphabet_timer.c
zjwww@localhost exp10$ gcc alphabet_timer.c -o alphabet_timer
zjwww@localhost exp10$ ./alphabet_timer
CDEFG
```

```
zjwww@localhost:~/exp/exp10 — /alphabet_timer
zjwww@localhost exp10$ vi alphabet_timer.c
zjwww@localhost exp10$ gcc alphabet_timer.c -o alphabet_timer
zjwww@localhost exp10$ ./alphabet_timer
BCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMN
```

11. 进程通信-管道

父子进程通过匿名管道通信，父子程序实现匿名管道通信

1. 子进程每 sleep2 秒, 随机产生一个 1-9 的整数, 写入管道后, 则给父进程发信号 SIGUSR1
2. 父进程捕获 SIGUSR1 信号, 在信号处理函数中对收到的数累加到全局变量 sum
3. 在父进程的代码块中, 每 sleep1 秒, 打印输出 sum 的值。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h> // 添加 wait 函数的头文件
#include <signal.h>
#include <time.h>

int sum = 0; // 全局变量, 用于累加
int pipefd[2]; // 将管道文件描述符设为全局变量

// 信号处理函数
void sigusr1_handler(int signo) {
    int num;
    read(pipefd[0], &num, sizeof(int)); // 从管道读取数据
    sum += num;
}

int main() {
    pid_t pid;

    // 创建匿名管道
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // 设置 SIGUSR1 信号处理函数
    signal(SIGUSR1, sigusr1_handler);

    // 创建子进程
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
}
```

```
if (pid == 0) { // 子进程
    close(pipefd[0]); // 关闭读端

    srand(time(NULL)); // 初始化随机数种子

    while (1) {
        int num = rand() % 9 + 1; // 生成 1~9 的随机数
        write(pipefd[1], &num, sizeof(int)); // 写入管道
        kill(getppid(), SIGUSR1); // 向父进程发送信号
        sleep(2); // 等待 2 秒
    }

    close(pipefd[1]); // 关闭写端
    exit(EXIT_SUCCESS);
} else { // 父进程
    close(pipefd[1]); // 关闭写端

    while (1) {
        printf("Current sum: %d\n", sum);
        sleep(1); // 等待 1 秒
    }

    close(pipefd[0]); // 关闭读端
    wait(NULL); // 等待子进程结束
    exit(EXIT_SUCCESS);
}

return 0;
}
```

```
zjwww@localhost:~/exp/exp11 - ./pipe_signal
zjwww@localhost exp11$ gcc pipe_signal.c -o pipe_signal
zjwww@localhost exp11$ ./pipe_signal
Current sum: 0
Current sum: 8
Current sum: 8
Current sum: 9
Current sum: 9
Current sum: 11
```

```
Current sum: 9
Current sum: 9
Current sum: 11
Current sum: 11
Current sum: 16
Current sum: 16
Current sum: 18
Current sum: 18
Current sum: 24
Current sum: 24
Current sum: 30
Current sum: 30
Current sum: 39
Current sum: 39
Current sum: 44
Current sum: 44
Current sum: 53
Current sum: 53
Current sum: 59
Current sum: 59
Current sum: 60
Current sum: 60
Current sum: 63
```

子进程任务: 每隔 2 秒随机生成 1-9 的整数, 将整数写入管道。写入后向父进程发送 SIGUSR1 信号。

父进程任务: 捕获 SIGUSR1 信号, 在处理函数中将接收到的整数累加到全局变量 sum。每隔 1 秒打印一次 sum 的当前值。

12. 进程通信-消息队列

编写程序，通过命令行参数 A、B、C 分别运行得到 3 个进程 A、B、C。

A 向 B 发送 [3-5] 的随机整数 a，B 完成 1-a 的累加，将结果发回 A。

A 向 C 发送 [3-5] 的随机整数 a，C 完成 1-a 的累乘，将结果发回 A。

A、B、C 仅使用一个消息队列协作通信，重复执行上述任务 3 次，适当加入 sleep 控制执行速度。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <time.h>

#define MSG_SIZE 128

typedef struct {
    long mtype;
    int data;
    char text[MSG_SIZE];
} Message;

int msgid;

void process_A() {
    srand(time(NULL) + getpid());
    int count = 0;

    while(count < 3) {
        // 生成随机数
        int a = rand() % 3 + 3; // [3,5]

        // 发送给 B
        Message msg;
        msg.mtype = 1; // B 的类型
        msg.data = a;
        sprintf(msg.text, "A->B: 计算 1-%d 的和", a);
        msgsnd(msgid, &msg, sizeof(Message)-sizeof(long), 0);
        printf("%s\n", msg.text);

        // 等待 B 的回复
        msgrcv(msgid, &msg, sizeof(Message)-sizeof(long), 2, 0);
        printf("B 回复 A: 1-%d 的和是 %d\n", a, msg.data);

        // 发送给 C
        msg.mtype = 3; // C 的类型
        msg.data = a;
        sprintf(msg.text, "A->C: 计算 1-%d 的积", a);
        msgsnd(msgid, &msg, sizeof(Message)-sizeof(long), 0);
        printf("%s\n", msg.text);

        // 等待 C 的回复
        msgrcv(msgid, &msg, sizeof(Message)-sizeof(long), 4, 0);
        printf("C 回复 A: 1-%d 的积是 %d\n", a, msg.data);

        printf("==== 第 %d 次任务完成 =====\n", count+1);
        sleep(1); // 避免太快
        count++;
    }

    // 发送结束信号
    Message end_msg;
    end_msg.mtype = 1; // 发给 B
    end_msg.data = -1;
    sprintf(end_msg.text, "结束");
    msgsnd(msgid, &end_msg, sizeof(Message)-sizeof(long), 0);

    end_msg.mtype = 3; // 发给 C
    msgsnd(msgid, &end_msg, sizeof(Message)-sizeof(long), 0);
}

void process_B() {
    Message msg;

    while(1) {
        msgrcv(msgid, &msg, sizeof(Message)-sizeof(long), 1, 0);
        if (msg.data == -1) break; // 结束信号

        printf("%s\n", msg.text);
        int sum = 0;
        for(int i=1; i<=msg.data; i++) {
            sum += i;
        }

        msg.mtype = 2; // 回复给 A
        msg.data = sum;
        sprintf(msg.text, "B->A: 1-%d 的和是 %d", msg.data, sum);
        msgsnd(msgid, &msg, sizeof(Message)-sizeof(long), 0);
        sleep(1);
    }
}

void process_C() {
    Message msg;

    while(1) {
        msgrcv(msgid, &msg, sizeof(Message)-sizeof(long), 3, 0);
        if (msg.data == -1) break; // 结束信号

        printf("%s\n", msg.text);
        int product = 1;
        for(int i=1; i<=msg.data; i++) {
            product *= i;
        }

        msg.mtype = 4; // 回复给 A
        msg.data = product;
        sprintf(msg.text, "C->A: 1-%d 的积是 %d", msg.data, product);
        msgsnd(msgid, &msg, sizeof(Message)-sizeof(long), 0);
        sleep(1);
    }
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <A|B|C>\n", argv[0]);
        exit(1);
    }

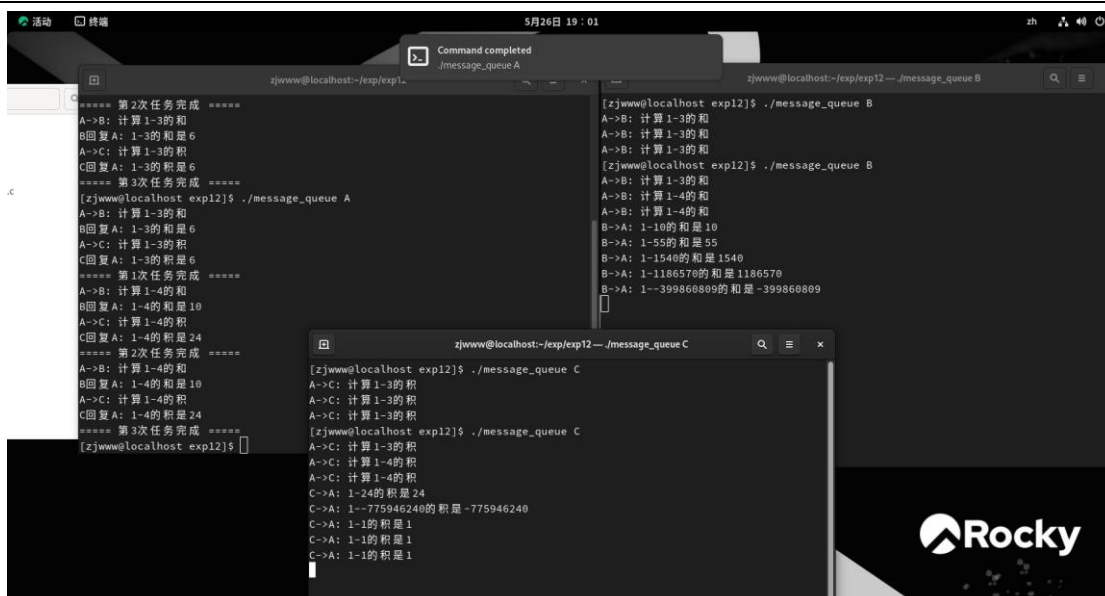
    // 创建消息队列
    key_t key = ftok("message.queue.c", 'a');
    msgid = msgget(key, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    if (strcmp(argv[1], "A") == 0) {
        process_A();
    } else if (strcmp(argv[1], "B") == 0) {
        process_B();
    } else if (strcmp(argv[1], "C") == 0) {
        process_C();
    } else {
        printf("Invalid argument. Use A, B or C.\n");
        exit(1);
    }
}
```

程序基于消息队列实现进程 A、B、C 的通信，进程 A 通过命令行参数启动后，会循环 3 次生成 3 到 5 的随机数，先向 B 发送计算 1 到该数累加和的消息，待 B 回复结果后，再向 C 发送计算 1 到该数累乘积的消息，每次任务间隔 1 秒。进程 B 和 C 启动后持续监听消息队列，分别处理来自 A 的求和与求积任务，计算完成后向 A 返回结果，直至接收到 A 发送的 data 为 -1 的结束消息才终止运行。

程序通过定义包含消息类型、数据和文本内容的结构体来规范消息格式，利用不同的 mtype 值区分发送给 B 和 C 的消息以及对应的回复消息，主函数根据命令行参数确定运行的进程角色，通过 ftok 和 msgget 创建共享消息队列，确保三个进程能基于同一队列进行通信，最后由进程在结束时删除消息队列以释放资源。

编译后，3. 运行三个终端窗口，分别启动 A、B、C 进程



```
5月26日 19:01
Command completed
./message_queue A

===== 第2次任务完成 =====
A->B: 计算 1-3 的和
B回复 A: 1-3 的和是 6
A->C: 计算 1-3 的积
C回复 A: 1-3 的积是 6
===== 第3次任务完成 =====
[zjwww@localhost exp12]$ ./message_queue A
A->B: 计算 1-3 的和
B回复 A: 1-3 的和是 6
A->C: 计算 1-3 的积
C回复 A: 1-3 的积是 6
===== 第1次任务完成 =====
A->B: 计算 1-4 的和
B回复 A: 1-4 的和是 10
A->C: 计算 1-4 的积
C回复 A: 1-4 的积是 24
===== 第2次任务完成 =====
A->B: 计算 1-4 的和
B回复 A: 1-4 的和是 10
A->C: 计算 1-4 的积
C回复 A: 1-4 的积是 24
===== 第3次任务完成 =====
[zjwww@localhost exp12]$

[zjwww@localhost exp12]$ ./message_queue B
A->B: 计算 1-3 的和
A->B: 计算 1-3 的和
A->B: 计算 1-3 的和
[zjwww@localhost exp12]$ ./message_queue B
A->B: 计算 1-3 的和
A->B: 计算 1-4 的和
A->B: 计算 1-4 的和
B->A: 1-10 的和是 10
B->A: 1-55 的和是 55
B->A: 1-1540 的和是 1540
B->A: 1-1186570 的和是 1186570
B->A: 1--399860809 的和是 -399860809

[zjwww@localhost exp12]$ ./message_queue C
A->C: 计算 1-3 的积
A->C: 计算 1-3 的积
A->C: 计算 1-3 的积
[zjwww@localhost exp12]$ ./message_queue C
A->C: 计算 1-3 的积
A->C: 计算 1-4 的积
A->C: 计算 1-4 的积
C->A: 1-24 的积是 24
C->A: 1--775946240 的积是 -775946240
C->A: 1-1 的积是 1
C->A: 1-1 的积是 1
C->A: 1-1 的积是 1
```

13. 进程、线程对比

编写程序来搜索大的素数，使用多线程：

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

int is_prime(long long n) {
    if (n <= 1) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0 || n % 3 == 0) return 0;
    for (long long i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return 0;
    }
    return 1;
}

typedef struct {
    long long start;
    long long result;
} thread_data;

void* find_prime(void* arg) {
    thread_data* data = (thread_data*)arg;
    for (long long i = data->start; ; i++) {
        if (is_prime(i)) {
            data->result = i;
            pthread_exit(NULL);
        }
    }
}

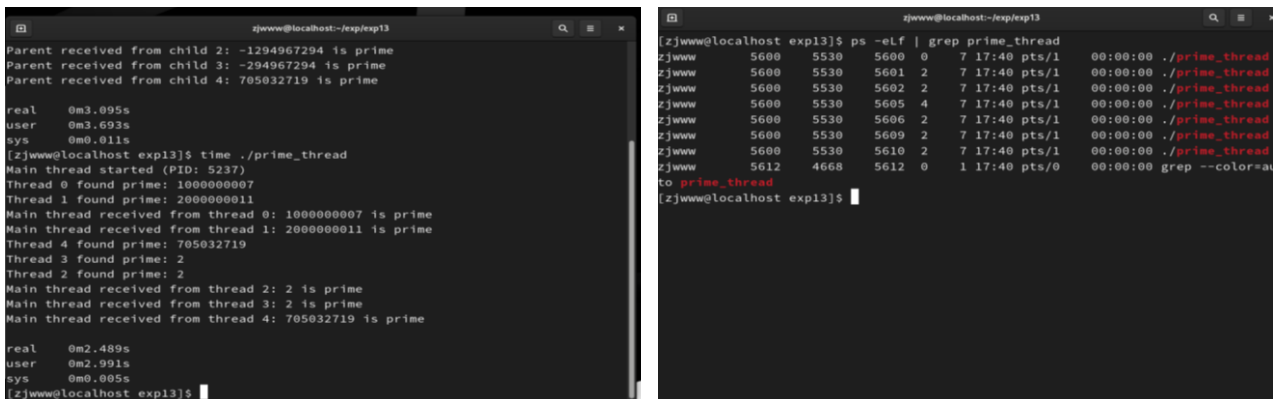
int main() {
    const int num_threads = 5;
    pthread_t threads[num_threads];
    thread_data data[num_threads];

    // 创建线程
    for (int i = 0; i < num_threads; i++) {
        data[i].start = (i + 1) * 100000000LL;
        if (pthread_create(&threads[i], NULL, find_prime, &data[i]) != 0) {
            perror("pthread_create");
            exit(EXIT_FAILURE);
        }
    }

    // 等待线程完成
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
        printf("Thread %d found prime: %lld\n", i+1, data[i].result);
    }

    return 0;
}
```

查看线程空间大小使用命令：ps -a|grep test，测试运行时。结果如下：



```
Parent received from child 2: -1294967294 is prime
Parent received from child 3: -294967294 is prime
Parent received from child 4: 705032719 is prime

real    0m3.095s
user    0m3.693s
sys     0m0.011s
[zjwww@localhost exp13]$ time ./prime_thread
Main thread started (PID: 5237)
Thread 0 found prime: 1000000007
Thread 1 found prime: 2000000011
Main thread received from thread 0: 1000000007 is prime
Main thread received from thread 1: 2000000011 is prime
Thread 4 found prime: 705032719
Thread 3 found prime: 2
Thread 2 found prime: 2
Main thread received from thread 2: 2 is prime
Main thread received from thread 3: 2 is prime
Main thread received from thread 4: 705032719 is prime

real    0m2.489s
user    0m2.991s
sys     0m0.005s
[zjwww@localhost exp13]$
```

```
[zjwww@localhost exp13]$ ps -elf | grep prime_thread
zjwww    5600    5530    5600    0   7 17:40 pts/1    00:00:00 ./prime_thread
zjwww    5600    5530    5601    2   7 17:40 pts/1    00:00:00 ./prime_thread
zjwww    5600    5530    5602    2   7 17:40 pts/1    00:00:00 ./prime_thread
zjwww    5600    5530    5605    4   7 17:40 pts/1    00:00:00 ./prime_thread
zjwww    5600    5530    5606    2   7 17:40 pts/1    00:00:00 ./prime_thread
zjwww    5600    5530    5609    2   7 17:40 pts/1    00:00:00 ./prime_thread
zjwww    5600    5530    5610    2   7 17:40 pts/1    00:00:00 ./prime_thread
zjwww    5612    4668    5612    0   1 17:40 pts/0    00:00:00 grep --color=
to prime_thread
[zjwww@localhost exp13]$
```

使用进程完成任务：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int is_prime(long long n) {
    if (n <= 1) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0 || n % 3 == 0) return 0;
    for (long long i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return 0;
    }
    return 1;
}

void find_prime(long long start) {
    for (long long i = start; ; i++) {
        if (is_prime(i)) {
            printf("%lld\n", i);
            exit(0);
        }
    }
}

int main() {
    const int num_processes = 5;
    pid_t pids[num_processes];
    int pipes[num_processes][2];

    // 创建管道
    for (int i = 0; i < num_processes; i++) {
        if (pipe(pipes[i]) == -1) {
            perror("pipe");
            exit(EXIT_FAILURE);
        }
    }

    // 创建子进程
    for (int i = 0; i < num_processes; i++) {
        pids[i] = fork();
        if (pids[i] == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }

        if (pids[i] == 0) { // 子进程
            close(pipes[i][0]); // 关闭读端
            dup2(pipes[i][1], STDOUT_FILENO); // 重定向输出到管道
            find_prime((i + 1) * 100000000LL);
            exit(EXIT_SUCCESS);
        } else { // 父进程
            close(pipes[i][1]); // 关闭写端
        }
    }

    // 父进程读取结果
    for (int i = 0; i < num_processes; i++) {
        char buffer[20];
        read(pipes[i][0], buffer, sizeof(buffer));
        printf("Process %d found prime: %s", i+1, buffer);
        close(pipes[i][0]);
    }

    // 等待所有子进程结束
    for (int i = 0; i < num_processes; i++) {
        waitpid(pids[i], NULL, 0);
    }

    return 0;
}
```

```

[zjwww@localhost exp13]$ time ./prime_process
Parent process started (PID: 5223)
Child 0 found prime: 1000000007
Child 1 found prime: 2000000011
Parent received from child 0: 1000000007 is prime
Parent received from child 1: 2000000011 is prime
Child 4 found prime: 705032719
Child 3 found prime: 2
Child 2 found prime: 2
Parent received from child 2: -1294967294 is prime
Parent received from child 3: -294967294 is prime
Parent received from child 4: 705032719 is prime

real    0m3.095s
user    0m3.693s
sys     0m0.011s
[zjwww@localhost exp13]$

```

```

[zjwww@localhost exp13]$ ps au | grep prime_process
zjwww    4971  0.0  0.0  2492  896 pts/0    T   17:18   0:00  ./prime_proce
ss
zjwww    4972  1.1  0.0  2492    0 pts/0    T   17:18   0:07  ./prime_proce
ss
zjwww    4973  1.1  0.0  2492    0 pts/0    T   17:18   0:07  ./prime_proce
ss
zjwww    4974  1.2  0.0  2492    0 pts/0    T   17:18   0:08  ./prime_proce
ss
zjwww    4975  1.1  0.0  2492    0 pts/0    T   17:18   0:07  ./prime_proce
ss
zjwww    4976  1.3  0.0  2492    0 pts/0    T   17:18   0:08  ./prime_proce
ss
zjwww    4977  1.1  0.0  2492    0 pts/0    T   17:18   0:07  ./prime_proce
ss
zjwww    4978  1.1  0.0  2492    0 pts/0    T   17:18   0:07  ./prime_proce
ss
zjwww    4979  1.3  0.0  2492    0 pts/0    T   17:18   0:08  ./prime_proce
ss
zjwww    4980  1.1  0.0  2492    0 pts/0    T   17:18   0:07  ./prime_proce
ss
zjwww    4981  1.2  0.0  2492    0 pts/0    T   17:18   0:07  ./prime_proce
ss
zjwww    5069  0.0  0.0  2492  896 pts/0    T   17:23   0:00  ./prime_proce
ss

```

进程、线程对比：

特性	进程	线程
地址空间	每个进程拥有独立的地址空间	同一进程内线程共享地址空间
内存分配	需要独立的内存空间	共享进程的内存空间
创建开销	较大，涉及内存空间分配	较小，共享内存空间
销毁开销	较大，需要操作系统进行大量工作	较小，配置线程特定资源
运行时间	创建和销毁进程开销较大	创建和销毁线程开销较小

14. 信号量实现生产者-消费者模型

问题：某工厂有两个生产车间和一个装配车间，两个生产车间分别生产 A、B 两种零件，装配车间负责组装零件 A、B。两个生产车间每生产一个零件后都要分别将这两个零件送到装配车间的货架 F1(容量 4)、F 2(容量 6)上，装配车间每次在组装零件时都要从货架 F1、F 2 上分别取下零件 A 和零件 B。分析题目，写出实现题目描述问题需要定义的信号量并说明每个信号量的功能和初值。

在这个场景中，可以定义以下信号量来实现生产者-消费者模型：

1. sem_F1：初值为 4，用于表示货架 F1 上的零件数量，生产车间每生产一个零件 A 都会将其放入 F1 上，装配车间每次组装（消费）一个零件 A 时会从 F1 上取走一个零件 A。
2. sem_F2：初值为 6，用于表示货架 F2 上的零件数量，生产车间每生产一个零件 B 都会将其放入 F2 上，装配车间每次组装（消费）一个零件 B 时会从 F2 上取走一个零件 B。
3. mutex：初值为 1，用于实现对货架 F1 和 F2 的互斥访问，确保一次只有一个车间可以访问货架。
4. empty：初值为 10，用于表示可以生产的零件数量。每个生产车间每生产一个零件时都会将 empty 减一，装配车间每次组装一个零件时会将 empty 加一，确保生产和消费的平衡。

通过正确地操作这些信号量，就可以实现生产者-消费者模型，保证每个车间的正常运行。

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int is_prime(long long n) {
    if (n <= 1) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0 || n % 3 == 0) return 0;
    for (long long i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return 0;
    }
    return 1;
}

void find_prime(long long start) {
    for (long long i = start; i++; ) {
        if (is_prime(i)) {
            printf("%lld\n", i);
            exit(0);
        }
    }
}

int main() {
    const int num_processes = 5;
    pid_t pids[num_processes];
    int pipes[num_processes][2];

    // 创建管道
    for (int i = 0; i < num_processes; i++) {
        if (pipe(pipes[i]) == -1) {
            perror("pipe");
            exit(EXIT_FAILURE);
        }
    }

    #include <stdio.h>
    #include <stdlib.h>
    #include <pthread.h>
    #include <semaphore.h>
    #include <unistd.h>

    // 信号量定义
    sem_t empty1, full1; // F1 货架 (容量 4)
    sem_t empty2, full2; // F2 货架 (容量 6)

// 生产者 A: 生产零件 A
void* producer_A(void* arg) {
    while (1) {
        // 模拟生产零件 A 的时间
        sleep(rand() % 2 + 1);

        // 申请 F1 空槽位
        sem_wait(&empty1);
        // 申请互斥锁
        sem_wait(&mutex);

        shelf1++;
        printf("生产者 A 生产零件 A, F1 现有 %d 个零件\n", shelf1);

        // 释放互斥锁
        sem_post(&mutex);
        // 唤醒消费者 (F1 有零件)
        sem_post(&full1);
    }
    return NULL;
}

// 生产者 B: 生产零件 B
void* producer_B(void* arg) {
    while (1) {
        // 模拟生产零件 B 的时间
        sleep(rand() % 2 + 1);

        // 申请 F2 空槽位
        sem_wait(&empty2);
        // 申请互斥锁
        sem_wait(&mutex);

        shelf2++;
        printf("生产者 B 生产零件 B, F2 现有 %d 个零件\n", shelf2);

        // 释放互斥锁
        sem_post(&mutex);
        // 唤醒消费者 (F2 有零件)
        sem_post(&full2);
    }
    return NULL;
}

// 消费者互斥锁
// 申请互斥锁
sem_wait(&mutex);

shelf1--;
shelf2--;
printf("消费者 取出 A+B 零件组装, F1 剩余 %d 个, F2 剩余 %d 个\n",
      shelf1, shelf2);

// 释放互斥锁
sem_post(&mutex);
// 唤醒生产者 (F1/F2 有空槽位)
sem_post(&empty1);
sem_post(&empty2);

// 模拟组装时间
sleep(rand() % 3 + 1);
return NULL;
}

int main() {
    // 初始化信号量
    sem_init(&empty1, 0, 4); // F1 空槽位初始值 4
    sem_init(&full1, 0, 0); // F1 零件数量初始值 0
    sem_init(&empty2, 0, 6); // F2 空槽位初始值 6
    sem_init(&full2, 0, 0); // F2 零件数量初始值 0
    sem_init(&mutex, 0, 1); // 互斥锁初始值 1

    pthread_t tid_A, tid_B, tid_consumer;

    // 创建生产者线程
    pthread_create(&tid_A, NULL, producer_A, NULL);
    pthread_create(&tid_B, NULL, producer_B, NULL);

    // 创建消费者线程
    pthread_create(&tid_consumer, NULL, consumer, NULL);

    // 等待线程结束 (按 Ctrl+C 终止程序)
    pthread_join(tid_A, NULL);
    pthread_join(tid_B, NULL);
    pthread_join(tid_consumer, NULL);

    // 销毁信号量
    sem_destroy(&empty1);
}

```

```

zjwww@192-exp14 ~ % ./semaphore
生产者 B 生产零件 B, F2 现有 1 个零件
生产者 A 生产零件 A, F1 现有 1 个零件
消费者 取出 A+B 零件组装, F1 剩余 0 个, F2 剩余 0 个
生产者 A 生产零件 A, F1 现有 1 个零件
生产者 B 生产零件 B, F2 现有 1 个零件
消费者 取出 A+B 零件组装, F1 剩余 0 个, F2 剩余 0 个
生产者 A 生产零件 A, F1 现有 1 个零件
生产者 B 生产零件 B, F2 现有 1 个零件
消费者 取出 A+B 零件组装, F1 剩余 0 个, F2 剩余 0 个
生产者 A 生产零件 A, F1 现有 2 个零件
生产者 B 生产零件 B, F2 现有 1 个零件
消费者 取出 A+B 零件组装, F1 剩余 1 个, F2 剩余 0 个
生产者 B 生产零件 B, F2 现有 2 个零件
生产者 A 生产零件 A, F1 现有 2 个零件
生产者 B 生产零件 B, F2 现有 2 个零件
生产者 A 生产零件 A, F1 现有 3 个零件
生产者 B 生产零件 B, F2 现有 3 个零件
消费者 取出 A+B 零件组装, F1 剩余 2 个, F2 剩余 2 个
生产者 B 生产零件 B, F2 现有 3 个零件
生产者 A 生产零件 A, F1 现有 3 个零件
生产者 B 生产零件 B, F2 现有 4 个零件
生产者 A 生产零件 A, F1 现有 4 个零件

```

```

zjwww@192-exp14 ~ % ./semaphore
生产者 B 生产零件 B, F2 现有 1 个零件
生产者 A 生产零件 A, F1 现有 1 个零件
消费者 取出 A+B 零件组装, F1 剩余 0 个, F2 剩余 0 个
生产者 A 生产零件 A, F1 现有 1 个零件
生产者 B 生产零件 B, F2 现有 1 个零件
消费者 取出 A+B 零件组装, F1 剩余 0 个, F2 剩余 0 个
生产者 A 生产零件 A, F1 现有 1 个零件
生产者 B 生产零件 B, F2 现有 1 个零件
消费者 取出 A+B 零件组装, F1 剩余 0 个, F2 剩余 0 个
生产者 A 生产零件 A, F1 现有 2 个零件
生产者 B 生产零件 B, F2 现有 1 个零件
消费者 取出 A+B 零件组装, F1 剩余 1 个, F2 剩余 0 个
生产者 B 生产零件 B, F2 现有 2 个零件
生产者 A 生产零件 A, F1 现有 2 个零件
生产者 B 生产零件 B, F2 现有 2 个零件
生产者 A 生产零件 A, F1 现有 3 个零件
生产者 B 生产零件 B, F2 现有 3 个零件
消费者 取出 A+B 零件组装, F1 剩余 2 个, F2 剩余 2 个
生产者 B 生产零件 B, F2 现有 3 个零件
生产者 A 生产零件 A, F1 现有 3 个零件
生产者 B 生产零件 B, F2 现有 4 个零件
生产者 A 生产零件 A, F1 现有 4 个零件

```

15. 并行下载

编写一个 2 个客户端程序，从服务器下载一个文件。一个是单进程(或线程)，另一个是 4 个子进程(或线程)，分别下载一个文件的不同分段。提交代码，记录它们下载速度的差异。

服务器上的文件内容为“One World One Dream”，共 19 个字符。客户端向服务器发送 2 个字节的內容，第一个字节 为文件的偏移量，第 2 个字节为下载的长度。比如发送的是 char D=(4, 5), 表明希望下载的片段是“World”。服务器模拟网络延迟，每隔 1 秒才能发送一个字节。

编译文件并运行 ser.c 服务器

```
[zjwww@192 exp15]$ gcc -o server ser.c -lpthread
ser.c: 在函数 'mysend' 中:
ser.c:21:25: 警告: 隐式声明函数 'sleep' [-Wimplicit-function-declaration]
   21 |         sleep(1);
       |         ^~~~~~
ser.c: 在函数 'do_work' 中:
ser.c:51:9: 警告: implicit declaration of function 'close'; did you mean 'pclose'
      '[-Wimplicit-function-declaration]
   51 |         close(connfd);
       |         ^~~~~~
       |         pclose
[zjwww@192 exp15]$ ./server
Accepting connections ...
```

编写、编译和保持服务器运行时运行单进程客户端代码 single_thread_time.c 和多线程客户端（4 线程）代码 multi_thread_time.c 以及客户端响应 和 服务器端响应（由于代码过长，不在此展示）

```
[zjwww@exp15-115 ~]$ ./server
Accepting connections ...
10th character is sent.
20th character is sent.
30th character is sent.
40th character is sent.
50th character is sent.
60th character is sent.
70th character is sent.
80th character is sent.
90th character is sent.
100th character is sent.
110th character is sent.
120th character is sent.
130th character is sent.
140th character is sent.
150th character is sent.
160th character is sent.
170th character is sent.
180th character is sent.
190th character is sent.
```

```
[zjwww@exp15-115 ~]$ ./multi_thread_time
Accepting connections ...
10th character is sent.
20th character is sent.
30th character is sent.
40th character is sent.
50th character is sent.
60th character is sent.
70th character is sent.
80th character is sent.
90th character is sent.
100th character is sent.
110th character is sent.
120th character is sent.
130th character is sent.
140th character is sent.
150th character is sent.
160th character is sent.
170th character is sent.
180th character is sent.
190th character is sent.
```

速度对比

多线程（4 线程）	单线程
18.022s	6.007s

多线程下载可以显著提高下载速度，特别是在服务器有延迟限制的情况下。在这个实验中，4 线程下载比单线程快了约 3 倍。