编写一个shell脚本，显示输出九九乘法表。

#!/bin/bash

# Linux course

# ex2 - display a nine by nine multiply table

# 1427405017 - Jiayun Shen - 2017/3/24

echo 'Linux ex2 - 9 by 9 multiply table'

for((i=1;i<=9;i++))

do

echo -en "$i:\t"

for((j=1;j<=i;j++))

do

let "result=$i\*$j"

echo -en "$i\*$j=$result\t"

done

echo ""

done

2. 创建一个文件，其中包含了一个使用date和who 命令的shell脚本，每条命

令写在一个行。使得文件可执行，然后运行这个脚本。写出完成这项工作的

所有步骤。

3. 把echo“Hello，world”命令的输出赋值给myname 变量并打印出它的值。

写出完成这项工作的所有命令。

4. 把 myname 变量的值复制到另一个变量anyname中，使anyname变量变为只

读，对myname 和anyname 两个变量使用unset 命令。这将有什么结果？

5. 编写一个shell脚本，它显示出所有的命令行参数。把它们都左移两位，并再

次显示所有的命令行参数。

6. 编写一个shell脚本，它带一个命令行参数，这个参数是一个文件。如果这个

文件是一个普通文件，则打印文件所有者的名字和最后的修改日期。如果程

序带有多个参数，则输出出错信息。

#!/bin/bash

# Linux course

# ex3 - Shell Scripts

# 1427405017 - Jiayun Shen - 2017/4/7

# 1) date and who

date

who

# 2) myname

myname=`echo "Hello, world"`

echo 'myname=' $myname

# 3) anyname

anyname=$myname

readonly anyname

unset myname

unset anyname

echo 'unset myname=' $myname

echo 'unset readonly anyname=' $anyname

#!/bin/bash

# Linux course

# ex3 - Shell Scripts

# 1427405017 - Jiayun Shen - 2017/4/7

echo $\*

shift 2

echo $\*

#!/bin/bash

# Linux course

# ex3 - Shell Scripts

# 1427405017 - Jiayun Shen - 2017/4/7

if test $# -eq 1

then

if test -f $1

then

echo 'Result shown below:'

row\_entries=`ls -lh $1`

col\_count=`ls -lh $1 | wc -w`

# echo ${row\_str[0]}

i=0

for col\_entry in $row\_entries

do

let "i++"

# user(owner)

if test $i -eq 3

then

echo 'User: ' $col\_entry

fi

# revise time(latest)

if test $i -eq 6

then

echo -en 'Time: '

fi

if test $i -ge 6 && test $i -lt $col\_count

then

echo -en $col\_entry ' '

fi

done

echo ''

else echo 'ERROR: Not a normal file!'

fi

else echo 'ERROR: support only one param(file)!'

fi

1. 针对以下文本内容的操作，请写出相关的awk命令

Beth 4.00 0

Dan 3.75 0

Kathy 4.00 10

Mark 5.00 20

Mary 5.50 22

Susie 4.25 18

（1） 当第3列大于0时，输出文本中对应行的第1列和第2列与第3列之积

（2） 统计第3列大于15的行数，记为X，并以下列格式输出 “X employees worked more than 15 hours.”

（3） 使用特殊变量NR、NF，统计输出整个文本的行数、单词数和字符数

2.针对下列文本内容，使用sed命令完成以下操作

4000001476

4000000041

4000002754

4000000944

4000002549

（1）将文本内容中每行中的400替换为800显示，即输出如下：

8000001476

8000000041

8000002754

8000000944

8000002549

（2）使用正则表达式，将文本内容中每行的前三个数字，以小括号括起来显示，即输出如下：

(400)0001476

(400)0000041

(400)0002754

(400)0000944

(400)0002549

（3）使用正则表达式，将文本内容中每行的中间三个数字前后增加“-”符号，即输出如下：

400-000-1476

400-000-0041

400-000-2754

400-000-0944

400-000-2549

（4）下列命令的输出结果是什么？

echo “123 abc” | sed –r ‘s/[0-9]+/& &/’

echo abcd123 | sed ‘s/\([a-z]\*\).\*/\1/’

#!/bin/bash

# Linux - ex5

# 沈家赟 1427405017

echo '================> ex5 - 1 <==============='

echo -e '\n1 - (1)\n'

cat emp.data | awk '$3>0{print $1, $2\*$3}'

echo -e '\n---------------------------------------'

echo -e '\n1 - (2)\n'

cat emp.data | awk '$3>15{emp=emp+1}

END{print emp, "employees worked more than 15 hours."}'

echo -e '\n---------------------------------------'

echo -e '\n1 - (3)\n'

cat emp.data | awk '{words=words+NF

chars=chars+length($0)}

END{print NR, "lines\t", words, "words\t", chars, "characters"}'

echo -e '\n---------------------------------------'

echo '======================================='

echo '================> ex5 - 2 <==============='

echo -e '\n2 - (1)\n'

cat telephone.txt | sed 's/^400/800/'

echo -e '\n---------------------------------------'

echo -e '\n2 - (2)\n'

cat telephone.txt | sed 's/^[0-9]\{3\}/(&)/'

echo -e '\n---------------------------------------'

echo -e '\n2 - (3)\n'

cat telephone.txt | sed 's/^\([0-9]\{3\}\)\([0-9]\{3\}\)/\1-\2-/'

echo -e '\n---------------------------------------'

echo -e '\n2 - (4)\n'

# echo '123abc' | sed -r 's/[0-9]+/& &/'

echo '123abc' | sed 's/[0-9]+/& &/' # -r则报错

echo abcd123 | sed 's/\([a-z]\*\).\*/\1/'

echo -e '\n---------------------------------------'

echo '======================================='

1、 合法的IP地址

pattern\_IP = re.compile(r"^((25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9]{1}[0-9]{1}|[0-9])\.){3}(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9]{1}[0-9]{1}|[0-9])$")

实现IP地址匹配，且滤除01.1.1.1这种不符合格式的IP。

经测试，可以分辨合法与不合法的IP地址串。

2、 合法的QQ号

QQ号是5~12位的数字串，且不以0开头。

pattern\_QQ = re.compile(r'^[1-9]{1}[0-9]{4,11}$')

3、 带区号的电话号码

带区号的电话号码，我国的电话号码区号3~4位，可以0开头，如北京010，苏州0512；电话号码则8位。

分隔符按书写习惯可以区号部分带小括号，也有减号间隔，或包含空格等。

pattern\_Tel = re.compile(r'^([0-9]{3,4}(-| | - )[0-9]{8})|(\([0-9]{3,4}\)[0-9]{8})$')

**1**．**Linux**中的**IPC**机制简介

进程间通信（Interprocess Communication，IPC）实现了进程之间同步和交换数据的功能。本实验要求完成的是一个或几个用户态的进程，依靠内核提供的进程间通信的机制，完成几个用户进程之间的通信。通常，在Linux中允许以下几种进程间通信的机制：

管道和命名管道（*FIFO*）

最适合在进程之间实现生产者/消费者的交互。有些进程往管道中写入数据，而另外一些进程则从管道中读出数据。

信号量

这就是我们在原理课程中讲述的内核信号量的用户态版本。

消息

允许进程异步的交换消息（小块数据）。可以认为消息是传递附加信息的信号。

共享内存区

当进程之间在高效的共享大量数据的时候，这是一种最适合实现的交互方式。

另外，Linux也允许相同主机上的进程之间，利用套接字（socket）进行通信。但套接字引入的最初目的是为了应用程序和网络接口之间实现数据通信，因此在这里就不再介绍。2

**2**．**FIFO**

POSIX引入了一个名为mkfifo()的系统调用专门用来创建FIFO，mkfifo()的函数原型为：

int mkfifo(const char \* pathname,mode\_t mode);

mkfifo()会根据参数pathname建立特殊的FIFO文件，该文件必须不存在，而参数mode为该文件的权限。FIFO一旦被创建，就可以使用普通的open()、read()、write()和close()系统调用进行访问。

mkfifo()函数若成功，则返回0，否则返回-1，错误原因存于errno中。

下面是一段示例代码。

//判断FIFO是否存在，如果不存在则建立

if (access(FIFO\_NAME, F\_OK) == -1) {

res = mkfifo(FIFO\_NAME, 0777);

if (res != 0) {

fprintf(stderr, "Could not create fifo %s\n",FIFO\_NAME);

exit(1);

}

}

**3**．**SystemV IPC**

System V IPC包含一组系统调用，上文提及的利用信号量、消息和共享内存区进行进程间通信，都属于其中。IPC数据结构是在进程请求IPC资源（信号量、消息队列或共享内存区）时动态创建的。每个IPC资源都是持久的，除非被进程显式的释放，否则永远驻留在内存中。由于一个进程可能需要相同类型的多个IPC资源，因此每个新资源都使用一个32位的IPC关键字来标识。这里重点介绍消息队列通信。

**(1)** 系统调用**msgget()**

为了创建一个新的消息队列，或者访问一个现有的队列，可以使用系统调用msgget()，其函数原型为：

int msgget(ket\_t key, int msgflg);

msgget()的第一个参数是IPC关键字的值。这个关键字的值将被拿来与内核中其他消息队列的现有值相比较。比较之后，打开或访问操作依赖于第二个参数的内容。

·IPC\_CREAT——如果内核中不存在该队列，则创建它。

·IPC\_EXCL——与IPC\_CREAT一起使用时，若队列早已存在则将出错。

如果只使用了IPC\_CREAT，msgget()或者返回新创建消息队列的标识符，或者返回现有的具有同一个关键字值的队列的标识符。如果同时使用了IPC\_CREAT和IPC\_EXCL，那么将可能会有两个结果。或者创建一个新的队列，或者如果该队列存在，则调用将出错，并返回-1。IPC\_EXCL本身是没有什么用处的，但在与IPC\_CREAT组合使用时，它可以用于保证没有一个现存的队列为3

了访问而被打开。

有个可选的八进制许可模式，它是与掩码进行OR操作以后得到的。这是因为从功能上讲，每个IPC对象的访问权限与Linux文件系统的文件许可权是相似的。

下面是两段示例代码，说明如何创建一个新的消息队列，以及如何使用一个已有的消息队列。

//建立消息队列，只有相同用户可以使用

queue\_id = msgget(QUEUE\_ID, IPC\_CREAT | IPC\_EXCL | 0600);

if (queue\_id == -1) {

perror("main: msgget");

exit(1);

}

//取得消息队列ID

queue\_id = msgget(QUEUE\_ID, 0);

if (queue\_id == -1) {

perror("main: msgget");

exit(1);

}

**(2)** 消息缓冲区

这个特殊的数据结构可以认为是消息数据的模板。虽然定义这种类型的数据结构是程序员的职责，但是读者绝对有必要知道实际上存在msgbuf类型的结构。在linux/msg.h中，它的定义如下：

/\* message buffer for msgsnd and msgrcv calls \*/

struct msgbuf {

long mtype;

char mtext[1];

};

msgbuf结构中有两个成员：

·mtype——它是消息类型，以正数表示。

·mtext——它就是消息数据。

不要被消息数据元素（mtext）的名称所误导，应用程序的编程人员是可以重新定义msgbuf这个结构的。对于给定消息的最大的大小，存在一个内部的限制。在Linux中，它在linux/msg.h中定义。

**(3)** 系统调用**msgsnd()**

一旦获得了消息队列的标识符，就可以开始在该队列上执行相关操作了。为了向队列传递消息，可以使用msgsnd()系统调用：

int msgsnd(int msqid, struct msgbuf \* msgp, size\_t msgsz, int msgflg);

第一个参数是消息队列的标识符，它是前面调用msgget()获得的返回值。4

第二个参数是一个指针，指向我们重新定义和载入的消息缓冲区。msgsz则包含了消息的大小，它是以字节为单位的，其中不包括消息类型的长度。

msgflg可以设置为0（忽略），也可以设置为IPC\_NOWAIT。如果消息队列已满，则消息将不会被写入到队列中，控制权将被还给调用进程。如果没有指定IPC\_NOWAIT，则调用进程将被阻塞，直到可以写消息为止。

下面是一段示例程序：

//申请消息缓冲区

msg =(struct msgbuf\*)malloc(sizeof(struct msgbuf)+MAX\_MSG\_SIZE);

//发送数据

msg->mtype = 1;

sprintf(msg->mtext, "hello world");

rc = msgsnd(queue\_id, msg, strlen(msg->mtext)+1, 0);

if (rc == -1) {

perror("main: msgsnd");

exit(1);

}

**(4)** 系统调用**msgrcv()**

一旦消息队列中有消息了，进程就可以从队列中获得消息，可以使用系统调用msgrcv()来完成这个功能：

ssize\_t msgrcv(int msqid, struct msgbuf \* msgq, size\_t msgsz, long msgtype, int msgflg);

显然，第一个参数是用来指定在消息获取过程中所使用的队列的。第二个参数代表消息缓冲区变量的地址，获取的消息存放在这里。第三个参数代表消息缓冲区结构的大小，不包括mtype成员的长度，可以使用下面的公式来计算大小：

msgsz = sizeof(struct mymsgbuf) –sizeof(long);

第四个参数指定要从队列中获取的消息的类型。内核将查找队列中具有匹配类型的最老的消息，并把它的一个拷贝返回到由msgp变量指定的地址中。如果把IPC\_NOWAIT作为一个参数传递给msgflg，而队列中没有任何消息。则该次调用会向调用进程返回ENOMSG。否则，调用进程将阻塞，直到满足msgrcv()参数的消息到达队列为止。

下面是一段示例程序：

//申请消息缓冲区

msg = (struct msgbuf\*)malloc(sizeof(struct msgbuf)+MAX\_MSG\_SIZE);

//读取消息队列中的数据

rc = msgrcv(queue\_id, msg, MAX\_MSG\_SIZE+1, msg\_type, 0);

if (rc == -1) {

perror("main: msgrcv");

exit(1);

}