实验 4　程序设计

课程名称：　LINUX 应用技术基础　实验类型：　操作

实验项目名称：　程序设计

学生姓名：　王冠颖　专业：　计科　学号：　3120103841

电子邮件地址：　543581485@qq.com

实验日期：2014 年　6　月 19　日

实验目的：

1. 学习 Bourne shell 的 shell 脚本的基本概念
2. 学习编写 Bourne shell 脚本的一些基本原则
3. 通过写简短的脚本，学会编写 Bourne shell 脚本程序的方法
4. 学习如何使用 LINUX 的 C 语言工具完成代码编辑，编译，运行程序
5. 学习掌握 make 工具，Makefile 文件的 make 规则
6. 学习使用系统调用编写程序

实验内容：

1. 编写一个 shell 脚本程序，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。

源代码：

```
#!/bin/bash
if [ $# -ne 1 ]; then
  echo "Error: one parameter allowed"  #the number of parameter
  exit 0
elif [ ! -f $1 ]; then
     echo "not a common file"   #judge the file type
else
   echo "the user name :"
   ls -l $1 | awk '{print $3}'  #use ls and select 3th and 6th paramemters
   echo "the update time :"
   ls -l $1 | awk '{print $6}'
fi
```

运行结果：

```
[wgy@localhost ~]$ vi test
[wgy@localhost ~]$ chmod u+x test
[wgy@localhost ~]$ ./test
Error: one parameter allowed
[wgy@localhost ~]$ ./test d3
not a common file
[wgy@localhost ~]$ ./test link
the user name :
wgy
the update time :
05-13
[wgy@localhost ~]$
```

2. 编写 shell 程序，统计指定目录下的普通文件、子目录及可执行文件的数目，目录的路径名字由参数传入。

```
#!/bin/bash
   echo "the common file : $(ls -al $1 | grep '^-' | wc -l)"
   echo "the dictionary file : $(ls -al $1 | grep '^d' | wc -l)"
   echo "the executable file : $(ls -aF $1 | grep '*$' | wc -l)"
```

```
[wgy@localhost ~]$ ./test ./
the common file : 39
the dictionary file : 24
the executable file : 2
[wgy@localhost ~]$ ./test d3
the common file : 0
the dictionary file : 2
the executable file : 0
[wgy@localhost ~]$ ./test temp
the common file : 2
the dictionary file : 5
the executable file : 0
[wgy@localhost ~]$
```

普通文件通过开头标志“-”和“d”来区分。可执行文件用 ls –F 区分，后缀为“*”即是可执行。

3. 用 bash 编写程序，该程序从键盘输入的 100 个整数，计算其总和、最大值和最小值并输出，从小到大排序并输出。

代码：

```
#!/bin/bash
   count=0
   sum=0
   while [ $count -lt 100 ]
      do
            read num
            echo $num >> thetemp    # output to a temporary file
            count=`expr $count + 1`
            sum=`expr $sum + $num`   #calculate the sum
      done
      #use command "sort" to sort the file
      #command "head" to put the first number in the sorted sequence
      echo "the min number:   $(sort -n thetemp | head -n 1)"
      echo "the max number:   $(sort -rn thetemp | head -n 1)"
      echo "the sum: $sum"
      echo "the sort sequence:"
      echo "$(sort -n thetemp)"
      >thetemp   #clear the content in file "thetemp" for next run
```

运行结果（注：为了方便，测试的时候改成输入 5 个数）：

```
[wgy@localhost ~]$ ./test
5
4
3
1
2
the min number:  1
the max number:  5
the sum: 15
the sort sequence:
1
2
3
4
5
[wgy@localhost ~]$
```

4. 本实验目的观察使用带-f选项的 tail 命令及学习如何使用 gcc 编译器，并观察进程运行。自己去查阅资料获取下面源程序中的函数（或系统调用）的作用。。创建一个文件名为 test.c 的 c 语言文件，内容如下：

```
#include  <stdio.h>
main()
{
      int  i;
      i = 0;
      sleep(10);
      while  (i < 5) {
            system("date");
            sleep(5);
            i++;
      }
```

```
        while  (1) {
            system("date");
            sleep(10);
        }
    }
```

在 shell 提示符下，依次运行下列三个命令：

```
gcc –o generate  test.c
./generate  >> dataFile  &
tail –f dataFile
```

- 第一个命令生成一个 c 语言的可执行文件，文件名为 generate；
- 第二个命令是每隔 5 秒和 10 秒把 date 命令的输出追加到 dataFile 文件中，这个命令为后台执行，注意后台执行的命令尾部加上&字符；
- 最后一个命令 tail - f dataFile，显示 dataFile 文件的当前内容和新追加的数据：

在输入 tail -f 命令 1 分钟左右后，按<Ctrl-C>终止 tail 程序。用 kill -9 *pid* 命令终止 generate 后台进程的执行。

最后用 tail dataFile 命令显示文件追加的内容。给出这些过程的你的会话。

注：pid 是执行 generate 程序的进程号；使用 generate >> dataFile &命令后，屏幕打印后台进程作业号和进程号，其中第一个字段方括号内的数字为作业号，第二个数字为进程号；也可以用 kill -9 %job 终止 generate 后台进程，job 为作业号。

```
[wgy@localhost ~]$ touch test.c
[wgy@localhost ~]$ vi test.c
[wgy@localhost ~]$ gcc -o generate test.c
[wgy@localhost ~]$ ./generate >> dataFile&
[1] 4917
[wgy@localhost ~]$ tail -f dataFile
2014年 06月 20日 星期五 21:12:02 CST
2014年 06月 20日 星期五 21:12:07 CST
2014年 06月 20日 星期五 21:12:12 CST
2014年 06月 20日 星期五 21:12:17 CST
2014年 06月 20日 星期五 21:12:22 CST
2014年 06月 20日 星期五 21:12:27 CST
2014年 06月 20日 星期五 21:12:37 CST
2014年 06月 20日 星期五 21:12:47 CST
2014年 06月 20日 星期五 21:12:57 CST
2014年 06月 20日 星期五 21:13:07 CST

[wgy@localhost ~]$ kill -9 %1
```

```
[wgy@localhost ~]$ tail dataFile
2014年 06月 20日 星期五 21:12:22 CST
2014年 06月 20日 星期五 21:12:27 CST
2014年 06月 20日 星期五 21:12:37 CST
2014年 06月 20日 星期五 21:12:47 CST
2014年 06月 20日 星期五 21:12:57 CST
2014年 06月 20日 星期五 21:13:07 CST
2014年 06月 20日 星期五 21:13:17 CST
2014年 06月 20日 星期五 21:13:28 CST
2014年 06月 20日 星期五 21:13:38 CST
2014年 06月 20日 星期五 21:13:48 CST
[wgy@localhost ~]$
```

5. 用 C 语言写一个程序名字为 **prog，该程序运行过程中共有 4 个进程，prog** 程序（父进程）创建 2 个子进程 p1 和 p2，p1 子进程再创建一个子进程 p3。**4 个进程还需要完成如下工作：**

1) 父进程打印字符串“I am main process”；
2) 使用任何一种通信机制实现 p1 进程和 p2 进程之间的通信，可以使用的通信机制如：管道、消息队列、共享内存等。P1 进程发送“Child process p1 is sending a message!”信息给 p2 进程，P2 进程发送“Child process p2 is sending a message!” 信息给 p1 进程，p1 和 p2 两个子进程接收信息后分别打印这两个字符串。
3) 子进程 p3 打印字符串“I am child process p3”，然后使用 exec（族）系统调用打印当前目录下文件和子目录下的详细信息。
4) 每个进程的要打印自己的 pid。

代码：

```c
#include<stdlib.h>
#include<string.h>
#include<sys/wait.h>
int main(void)
{
    pid_t tmp_p1,tmp_p2,tmp_p3;
    int pipe1[2],pipe2[2];
    int wpid,status;
    char buf_r1[100],buf_r2[100]; //the read interface
    char buf_w1[100] = "child process p1 is sending a message!";
    char buf_w2[100] = "child process p2 is sending a message!";
         //the write message
    char *arg[]={"ls","-l",NULL};

    memset(buf_r1,0,sizeof(buf_r1));
    memset(buf_r2,0,sizeof(buf_r2));//initialize

    if(pipe(pipe1) < 0) return -1;
    if(pipe(pipe2) < 0) return -1;//create two pipes

    tmp_p1 = fork(); //create process p1 from main process
```

```
    if(tmp_p1 < 0)
    {
       printf("fail to create sub process\n");
       exit(0);
    }
    else if(tmp_p1 == 0)
    {
          tmp_p3 = fork(); //create process p3 from p1
          if(tmp_p3 == 0) //process p3
          {
              printf("I'm the child process p3\n");
              printf("the pid pf process p3 is %d\n",getpid());
              execv("/bin/ls",arg); //ls -l
          }
          else if(tmp_p3 > 0) //process p1
          {
              printf("the pid pf process p1 is %d\n",getpid());
              close(pipe1[1]);
              close(pipe2[0]);
              write(pipe2[1],buf_w2,strlen(buf_w2));
              if(read(pipe1[0],buf_r1,100)>0)
              printf("%s\n",buf_r1);

              close(pipe1[0]);
              close(pipe2[1]);
              wpid = wait(&status); //wait p3
          }
    }
    else
    {
          tmp_p2 = fork(); //create process p2 from main process
          if(tmp_p2 > 0) //main process
          {
              printf("I'm main process\n");
              printf("the pid pf main process is %d\n",getpid());
              wpid = wait(&status);//wait p2
          }
          else if(tmp_p2 == 0) //process p2
          {
                printf("the pid pf process p2 is %d\n",getpid());
                close(pipe2[1]);
                close(pipe1[0]);
                write(pipe1[1],buf_w1,strlen(buf_w1));
                if(read(pipe2[0],buf_r2,100)>0)
                    printf("%s\n",buf_r2);
                close(pipe2[0]);

                close(pipe1[1]);
          }
          wpid = wait(&status); //wait p1
    }

    return 0;
}
```

运行结果：

```
[wgy@localhost ~]$ gcc ./temp/prog.c -o prog
[wgy@localhost ~]$ ./prog
I'm the child process p3
the pid pf process p3 is 4870
the pid pf process pl is 4869
the pid pf process p2 is 4871
child process p2 is sending a message!
I'm main process
the pid pf main process is 4868
child process pl is sending a message!
总计 644
-r--r--r-- 1 wgy wgy      0 05-13 19:27 -
-rw-rw-r-- 1 wgy wgy     24 05-30 15:18 1
-rw-rw-r-- 1 wgy wgy     24 05-30 15:18 2
drwxrwxr-x 2 wgy wgy   4096 05-29 20:04 dl
drwxrwxr-x 2 wgy wgy   4096 05-29 20:51 d2
drwxrwxr-x 2 wgy wgy   4096 05-29 20:04 d3
-rw-rw-r-- 1 wgy wgy    602 06-20 21:13 dataFile
dr--r--r-- 2 wgy wgy   4096 03-25 19:38 Desktop
-rw-rw-r-- 1 wgy wgy      0 05-30 14:59 error.log
-rw-rw-r-- 1 wgy wgy      0 05-30 14:50 errpr.log
-rw-rw-r-- 1 wgy wgy     80 05-30 15:06 fdata
-r--r--r-- 1 wgy wgy      0 04-19 21:11 file
-r--r--r-- 1 wgy wgy     16 04-23 18:11 firscrip
-rw------- 1 wgy wgy    849 06-20 21:02 tmp
-rw------- 1 wgy wgy    637 06-20 20:57 tmp~
dr--r--r-- 2 wgy wgy   4096 04-19 21:14 use
-r--r--r-- 1 wgy wgy      0 03-25 19:40 新文件~
[wgy@localhost ~]$
```

(完成本题的有关资料请参考教材第 6、7 章)

6. Fibonacci 序列为 0 ,1,1,2,3,5 ,8,…，通常，这可表达为:

$fib_0=0$

$fib_1 = 1$

$fib_n = fib_{n-1} + fib_{n-2}$

编写一个多线程程序来生成Fibonacci 序列。程序应该这样工作:用户运行程序时在命令行输入要产生Fibonacci序列的个数，然后程序创建一个新的线程来产生Fibonacci序列，把这个序列放到线程共享的数据中(数组可能是一种最方便的数据结构)。当线程执行完成后，父线程将输出由子线程产生的序列。由于在子线程结束前，父线程不能开始输出Fibonacci 序列，因此父线程必须等待子线程的结束。

（完成本题的有关知识请参考教材第8章）

代码：

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

void* fibonacci(void* ptr)
{
      int* fib = (int*) ptr;
      int max,a,b,i;
      max = fib[0];
      fib[0] = a = 0; fib[1] = b = 1;
      for(i = 2; i < max; i++)
      {
            fib[i] = a + b; //store the sum of former two operates
            a = b;
            b = fib[i];  //replace the new operates
      }
}

int main(void)
{
    pthread_t p;
    int* fib;

    int num,i;
    void* re;

    scanf("%d",&num);
    fib = (int*) malloc (num * sizeof(int));
   //allocate memory for the fib[] array
    fib[0] = num;

    if(pthread_create(&p,0,&fibonacci, fib) == 0) //create a thread
    {
         pthread_join(p,&re); //wait
         for(i = 0;i < num; i++)
             printf("%d ",fib[i]);
    }
   else printf("Error: fail to create\n");
    printf("\n");
    free(fib);
    return 0;
}
```

运行结果：

```
[wgy@localhost ~]$ gcc test.c -o generate -lpthread
[wgy@localhost ~]$ ./generate
4
0 1 1 2
[wgy@localhost ~]$ ./generate
20
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
[wgy@localhost ~]$
```

下面题目为选做题，计算机学院学生或计算机学院的准学生为必做题，其他同学可以不做。

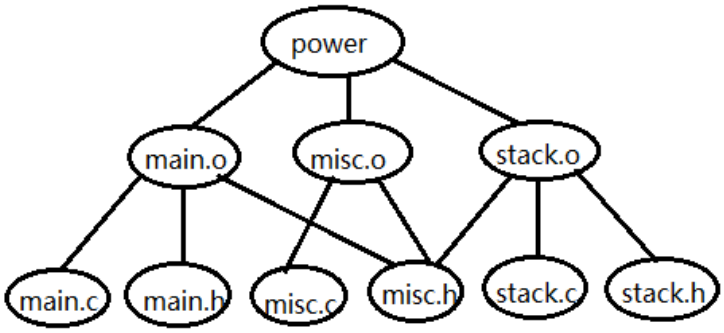
7. 在 linux 系统下的软件开发中，经常要使用 make 工具，要掌握 make 的规则。makfie 文件中的每一行是描述文件间依赖关系的 make 规则。本实验是关于 makefile 内容的，您不需要在计算机上进行编程运行，只要书面回答下面这些问题。

对于下面的 makefile:

```
CC = gcc
OPTIONS = -O3 -o
OBJECTS = main.o stack.o misc.o
SOURCES = main.c stack.c misc.c
HEADERS = main.h stack.h misc.h
power: main.c $(OPJECTS)
    $(CC) $(OPTIONS) power $(OBJECTS) -lm
main.o: main.c main.h misc.h
stack.o: stack.c stack.h misc.h
misc.o: misc.c misc.h
```

回答下列问题

a. 所有宏定义的名字
   CC、OPTIONS、OBJECTS、SOURCES、HEADERS。
b. 所有目标文件的名字
   power、main.o、stack.o、misc.o
c. 每个目标的依赖文件
   power: 依赖于 main.o stack.o misc.o
   main.o: 依赖于 main.c main.h misc.h
   stack.o: 依赖于 stack.c stack.h misc.h
   misc.o: 依赖于 misc.c misc.h
d. 生成每个目标文件所需执行的命令
   power: gcc -O3 -o power main.o stack.o misc.o -lm
   main.o: gcc -c main.c
   stack.o: gcc -c stack.c
   misc.o: gcc -c misc.c
e. 画出 makefile 对应的依赖关系树。
   自下而上依赖：



f. 生成 main.o stack.o 和 misc.o 时会执行哪些命令，为什么？

```
main.o: gcc  -c main.c
stack.o: gcc  -c stack.c
misc.o: gcc  -c misc.c
```

用-c 命令将 c 文件编译成 obj 文件。

8. 用编辑器创建 main.c, compute.c, input.c, compute.h, input.h 和 main.h 文件。下面是它们的内容。注意 compute.h 和 input.h 文件仅包含了 compute 和 input 函数的声明但没有定义。定义部分是在 compute.c 和 input.c 文件中。main.c 包含的是两条显示给用户的提示信息。

```
$ cat compute.h
/* compute 函数的声明原形 */
double compute(double, double);
$ cat input.h
/* input 函数的声明原形 */
double input(char *);
$ cat main.h
/* 声明用户提示 */
#define PROMPT1 "请输入 x 的值："
#define PROMPT2 "请输入 y 的值："
$ cat compute.c
#include <math.h>
#include <stdio.h>
#include "compute.h"
double compute(double x, double y)
{
   return (pow ((double)x, (double)y));
}
$ cat input.c
#include <stdio.h>
#include"input.h"
double input(char *s)
{
```

```
    float x;
    printf("%s", s);
    scanf("%f", &x);
    return (x);
}
$ cat main.c
#include <stdio.h>
#include "main.h"
#include "compute.h"
#include "input.h"

main()
{
    double x, y;
    printf("本程序从标准输入获取 x 和 y 的值并显示 x 的 y 次方.\n");
    x = input(PROMPT1);
    y = input(PROMPT2);
    printf("x 的 y 次方是:%6.3f\n",compute(x,y));
}
$
```

为了得到可执行文件 power，我们必须首先从三个源文件编译得到目标文件，并把它们连接在一起。下面的命令将完成这一任务。注意，在生成可执行代码时不要忘了连接上数学库。

```
$ gcc -c main.c input.c compute.c
$ gcc main.o input.o compute.o -o power -lm
$
```

相应的 Makefile 文件是:

```
$ cat Makefile
power: main.o input.o compute.o
    gcc main.o input.o compute.o -o power -lm

main.o: main.c main.h input.h compute.h
    gcc -c main.c

input.o: input.c input.h
    gcc -c input.c

compute.o: compute.c compute.h
    gcc -c compute.c
$
```

（1）、创建上述三个源文件和相应头文件，用 gcc 编译器，生成 power 可执行文件，并运行 power 程序。给出完成上述工作的步骤和程序运行结果。

```
[wgy@localhost temp]$ cat > compute.h
double compute(double,double);

[wgy@localhost temp]$ cat > input.h
double input(char*);
[wgy@localhost temp]$ cat > main.h
#define PROMPT1 "please input x:"
#define PROMPT2 "please input y:"

[wgy@localhost temp]$ cat > compute.c
#include <math.h>
#include <stdio.h>
#include "compute.h"
double compute(double x,double y)
{
   return (pow ((double)x, (double)y));
}
[wgy@localhost temp]$ cat > input.c
#include <stdio.h>
#include "input.h"
double input (char *s)
{
    float x;
printf("%s",s);
scanf("%f",&x);
return(x);
}
[wgy@localhost temp]$ touch main.c
[wgy@localhost temp]$ vi main.c
[wgy@localhost temp]$
```

```
#include<stdio.h>
#include "main.h"
#include "compute.h"
#include "input.h"

main()
{
    double x,y;
    printf("this program read x and y from standard input and print x^y.\n");
    x = input(PROMPT1);
   y = input(PROMPT2);
   printf("the result is : %6.3f\n",compute(x,y));
}
~
~
~
```

编译运行：

```
[wgy@localhost temp]$ gcc -c main.c input.c compute.c
[wgy@localhost temp]$ gcc main.o input.o compute.o -o power -lm
[wgy@localhost temp]$ ./power
this program read x and y from standard input and print x^y.
please input x:2
please input y:4
the result is : 16.000
[wgy@localhost temp]$ ./power
this program read x and y from standard input and print x^y.
please input x:8.1
please input y:3.4
the result is : 1227.013
[wgy@localhost temp]$
```

（2）、创建 Makefile 文件，使用 make 命令，生成 power 可执行文件，并运行 power 程序。给出完成上述工作的步骤和程序运行结果。

```
[wgy@localhost temp]$ touch Makefile
[wgy@localhost temp]$ vi Makefile
```

```
power: main.o input.o compute.o
        gcc main.o input.o compute.o -o power -lm

main.o: main.c main.h input.h compute.h
        gcc -c main.c
input.o: input.c input.h
        gcc -c input.c
compute.o: compute.c compute.h
        gcc -c compute.c
```

```
[wgy@localhost temp]$ make power
make:  “power”是最新的。
[wgy@localhost temp]$ ./power
this program read x and y from standard input and print x^y.
please input x:8.1
please input y:3.4
the result is : 1227.013
[wgy@localhost temp]$
```

：

9. 用 C 语言写一个名字为 myls 程序，实现类似 Linux 的 ls 命令，其中 myls 命令必须实现-a、-l、-i 等选项的功能。要求 myls 程序使用系统调用函数编写，**不能使用 exec 系统调用或 system()函数等调用 ls 命令来实现**。命令 man ls 可以得到更多 ls 选项的含义。

(完成本题的有关知识请参考教材第 5 章)

代码：

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>
#include <dirent.h>
void getMode(int mode, char* str);//change mode to a string
void list(char *file, struct stat buf); //list the detail information
char* getUname(uid_t uid); //change uid_t to a string user name
char* getGname(gid_t gid); //change gid_t to a string group name
void run(char* dir);
char glo[80];
//a global string to store the return information of getUname and getGname
int main(int argc, char** argv)
{
   run(argv[1]);
   return 0;
}
```

```c
void run(char* dir)
{
   DIR *dirp;
   struct dirent *direntp;
   struct stat buf;
   char tmp_str[30];
   if((dirp = opendir(dir))== NULL) //if it is not a directionary
   {
      if(stat(dir,&buf)==-1) perror(dir);
      else list(dir,buf); //list as a file
   }
   else
   {
      direntp = readdir(dirp);
      if(direntp == NULL) return;
      while(1)  //read and list each file
      {
            strcpy(tmp_str,dir);
            strcat(tmp_str,"/");
            strcat(tmp_str,direntp->d_name);
            //get the whole name of the file
```

```
            if(stat(tmp_str,&buf)==-1) perror(tmp_str);
            else list(tmp_str,buf);
            free(direntp);
            direntp = readdir(dirp);
            if(direntp == NULL) break;
        }

    }
    closedir(dirp);
}


void getMode(int mode, char* str)
{
    strcpy(str,"----------"); //the common and initial case

    if(S_ISDIR(mode)) str[0] = 'd';
    if(S_ISCHR(mode)) str[0] = 'c';
    if(S_ISBLK(mode)) str[0] = 'b';

    if(mode & S_IRUSR) str[1] = 'r';
    if(mode & S_IWUSR) str[2] = 'w';
    if(mode & S IXUSR) str[3] = 'x';


    if(mode & S_IRGRP) str[4] = 'r';
    if(mode & S_IWGRP) str[5] = 'w';
    if(mode & S_IXGRP) str[6] = 'x';

    if(mode & S_IROTH) str[7] = 'r';
    if(mode & S_IWOTH) str[8] = 'w';
    if(mode & S_IXOTH) str[9] = 'x';
    //exchange mode to string

    str[10] = 0; //end string
}

void list(char *file, struct stat buf)
{
    char str[11];
    getMode(buf.st_mode,str);

    printf("%d ",buf.st_ino); //the inode
    printf("%s ",str);    //the mode
    printf("%d ",(int)(buf.st_nlink)); //the nuber of link
    printf("%s ",getUname(buf.st_uid)); //the uname
```

```
    printf("%s ",getGname(buf.st_gid)); //the gname
    printf("%ld ",(long)buf.st_size); //size
    printf("%.12s ",4+ctime(&buf.st_mtime));
    //time with month,date, hour and minute
    printf("%s\n",file);//filename
}

char* getUname(uid_t uid)
{
    struct passwd* p;
    char uid_str[10];

    p = getpwuid(uid);
    if(p == NULL)
    {
        sprintf(uid_str,"%d",uid); // output the number directly
        strcpy(glo,uid_str);
    }
    else
        strcpy(glo,(*p).pw_name); //get the name
    return glo;
}

char* getGname(gid_t gid) //very familiar with getUname
{
    struct group* p;
    char gid_str[10];

    p = getgrgid(gid);
    if(p == NULL)
    {
        sprintf(gid_str,"%d",gid);
        strcpy(glo,gid_str);
    }
    else
        strcpy(glo,(*p).gr_name);
    return glo;
}
```

运行结果：

显示类似于 ls –lai 的功能。

```
[wgy@localhost temp]$ gcc myls.c -o myls
[wgy@localhost temp]$ ./myls dl/fl
3430738 -rw-rw-r-- 1 wgy wgy 0 May 29 20:06 dl/fl
[wgy@localhost temp]$ ./myls dl/smallFile
3430744 -rw-rw-r-- 2 wgy wgy 1042 May 29 20:46 dl/smallFile
[wgy@localhost temp]$ ./myls d2
3430638 drwxrwxr-x 5 wgy wgy 4096 Jun 24 19:05 d2/..

3430736 drwxrwxr-x 2 wgy wgy 4096 May 30 13:55 d2/.
3430862 -rw-rw-r-- 1 wgy wgy   18 Jun 24 19:38 d2/tmpl
3430858 -rw-rw-r-- 1 wgy wgy    0 Jun 24 19:38 d2/tmp2
```

心得体会

这次实验让我更了解了 vi 编辑器、gcc 编译器、make 等编程工具的使用，也复习了 shell 文本过滤器的基本语句。附加题最后一题更是学习了 stat、dirent 结构体，这一块的函数和结构确实十分强大。

不过，因为对 linux 下的 debug 太不熟悉，只好用 printf 大法来调试，花了不少时间。下次写程序一定要把 gcc 的 debug 功能弄清楚。