## **实验三**

# 一、shell编程

在图形化桌面出现之前，和Unix系统交互的唯一方式就是通过Shell提供的文本命令行界面。Shell是个很有用的脚本工具，它多半运行在比编译型语言还高的层次，它能够轻易处理文件与目录之类的对象。 Shell脚本最常用于系统管理工作，它可以把很多命令串在一起，放进一个独立的程序或者脚本中，这样只要执行这个脚本或者程序，便能完成工作。

Linux Shell的种类很多，目前流行的Shell包括ash、bash、ksh、csh、等， 我们重点关注bash Shell，bash Shell是一个增强的bourne Shell，这个是个标准的Unix的Shell，也是Linux上默认的Shell。不同的系统默认的shell是不同的，比如FreeBSD是以csh为其默认的Shell。在Linux系统中，默认的Shell是Bash。

本部分的主要内容：

1. 常见shell命令
2. shell编程基础与实践

**实验要求：**

（1）查阅相关资料，使用常见的shell命令，包括但不限于：

目录操作命令 —— pwd、cd、ls、mkdir

文件操作命令 —— touch、file、cp、rm、mv、find

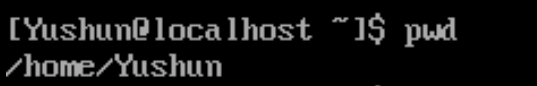
归档及压缩命令 —— tar

输入输出与文件管理 —— echo、read、cat、grep、|（管道）、输入输出重定向

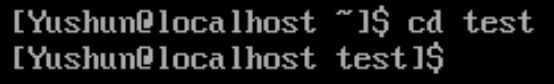
请给出你对这些命令的理解以及使用过程(附截图)

**目录操作命令**

1. pwd：显示当前所在的工作目录

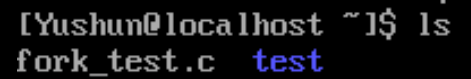


1. cd：用于切换目录

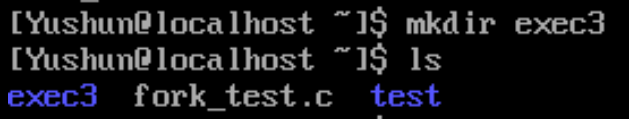


其他常用cd指令，cd -：返回上一次所在目录，cd..：进入上级目录，cd ~：切换到当前用户的家目录，cd ~username：切换到其他用户的家目录

1. ls：用来打印当前目录清单或者打印出指定目录下的文件及文件清单。ls命令在打印文件清单时，通过添加后续选项例如“ls -l -a”还可以查看隐藏文件、文件权限、目录信息等等。



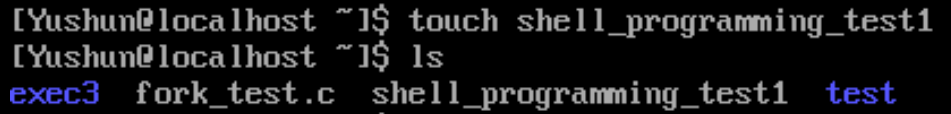
1. mkdir：用来创建目录



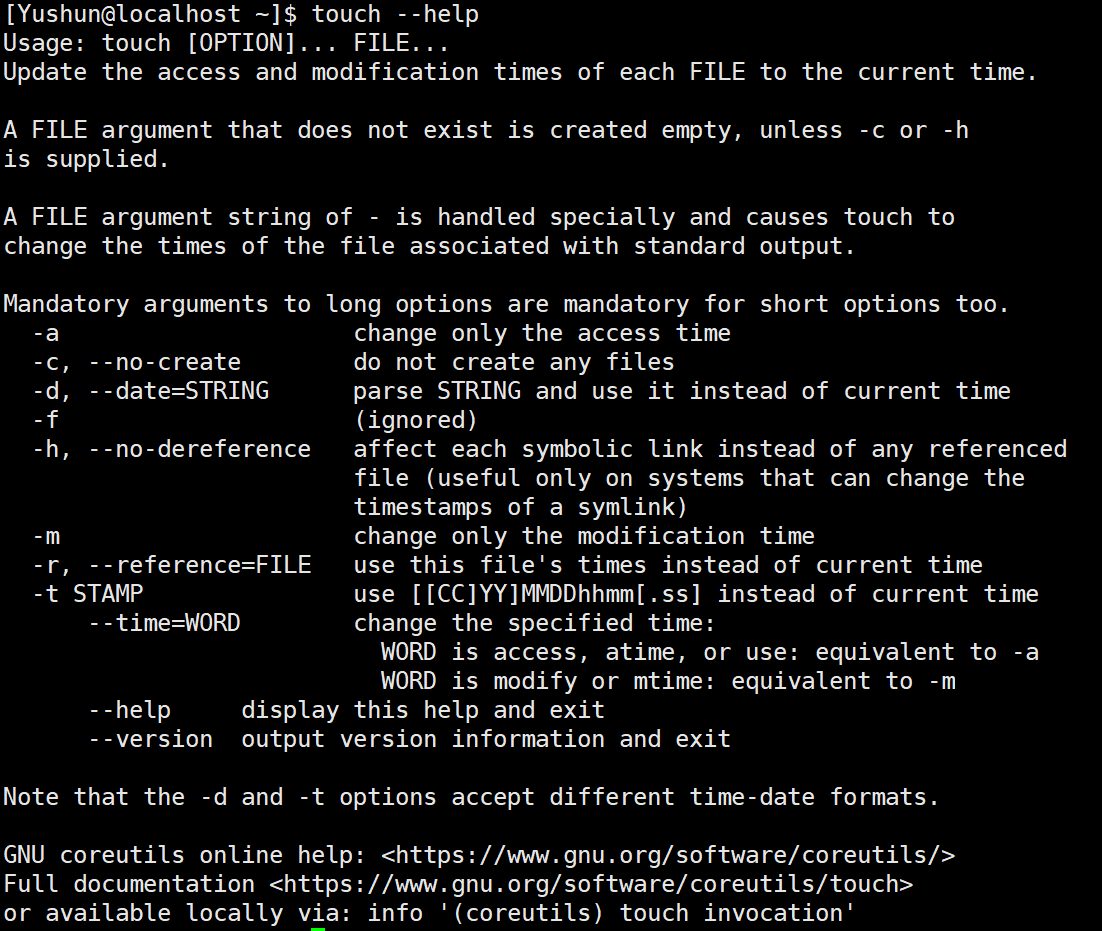
如图，这里成功创建了exec3目录。

**文件操作命令**

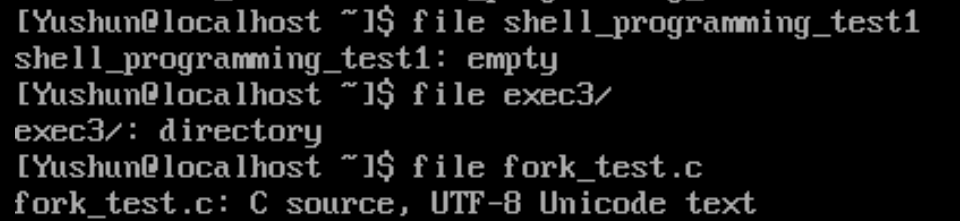
1. touch：主要用来修改文件时间戳，或者新建一个不存在的文件



touch命令使用方法一览：



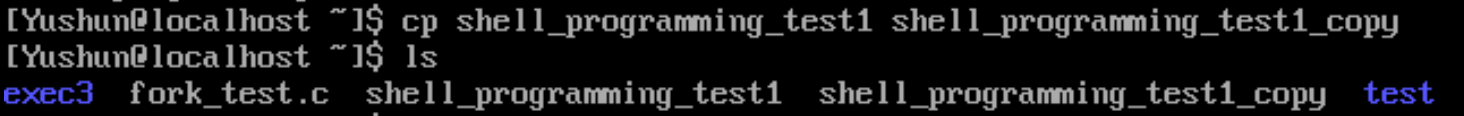
1. file：识别出文件的类型和编码格式



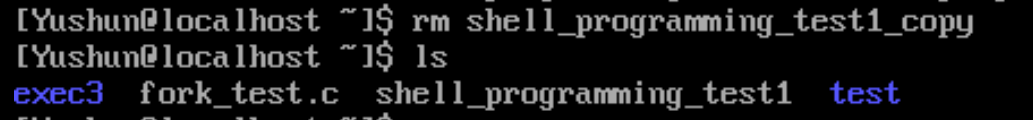
file命令使用方法一览：



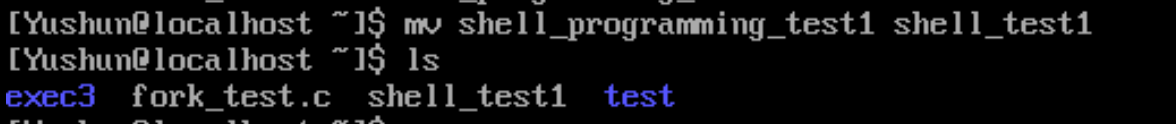
1. cp：主要用于复制文件或目录



1. rm：删除文件或目录



1. mv:用来为文件或目录改名、或将文件或目录移入其它位置



mv使用方法一览：



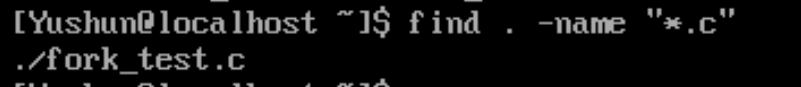
1. find命令：用来在指定目录下查找文件

下面通过几个例子来演示find命令：

将当前目录及其子目录下所有文件后缀为 .c 的文件列出来:

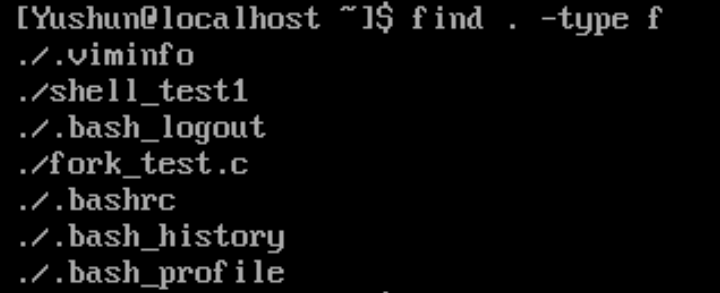
将当前目录及其子目录下所有文件后缀为 .c 的文件列出来:

$ find . -name "\*.c"



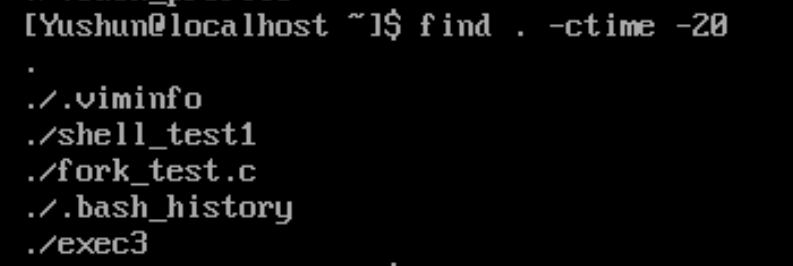
将目前目录其其下子目录中所有一般文件列出

$ find . -type f

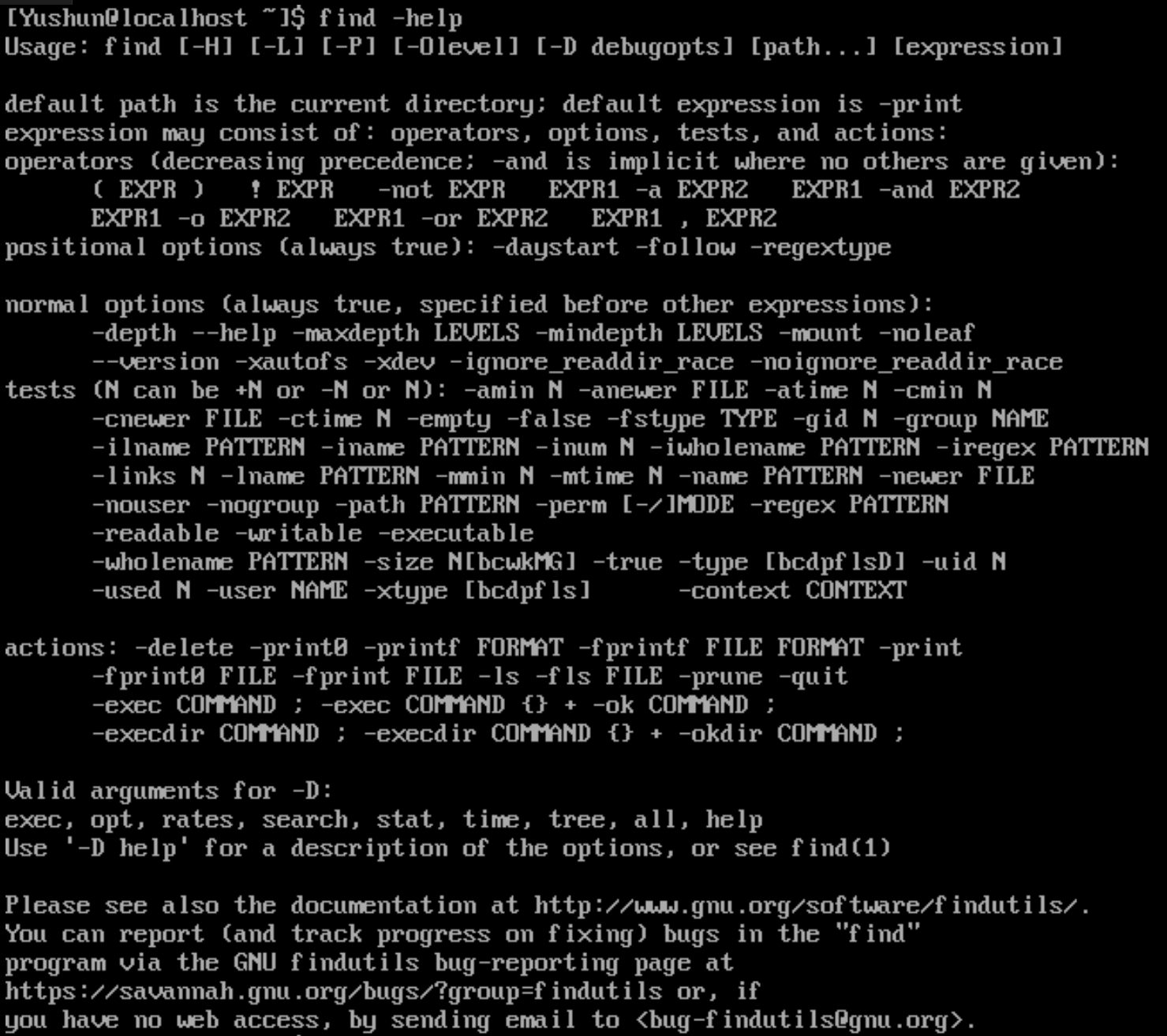


将当前目录及其子目录下所有最近 20 天内更新过的文件列出:

$ find . -ctime -20



find使用方法一览：

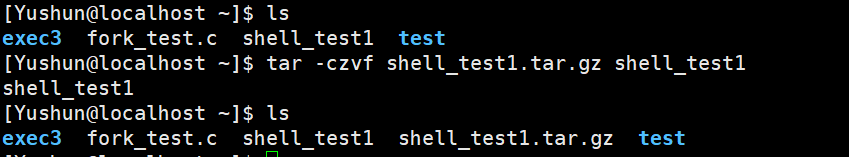


**归档及压缩命令**

tar命令：命令用于备份文件。tar 是用来建立，还原备份文件的工具程序，它可以加入，解开备份文件内的文件。

压缩文件，非打包：

$ tar -czvf shell\_test1.tar.gz shell\_test1（压缩shell\_test1文件为shell\_test.tar.gz）



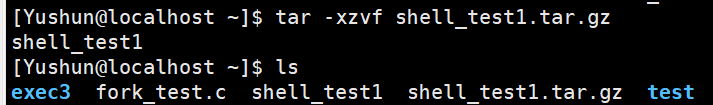
列出压缩文件内容

$ tar -tzvf shell\_test1.tar.gz



解压文件操作：

$ tar -xzvf a.tar.gz

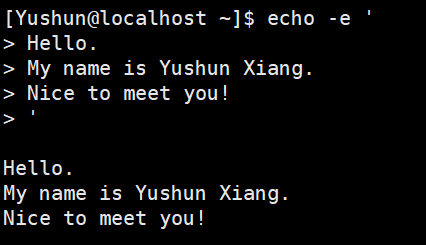


**输入输出与文件管理**

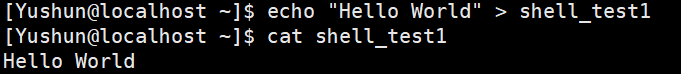
1. echo：用于字符串的输出，命令格式。

例如普通的字符串显示：



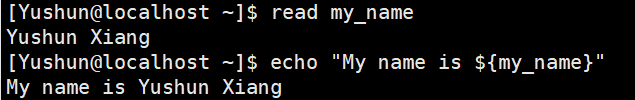


也可输入到文件中去：

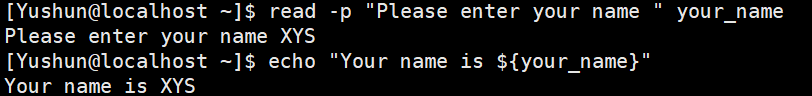


1. read：用于从标准输入读取数值。

read 内部命令被用来从标准输入读取单行数据。这个命令可以用来读取键盘输入，当使用重定向的时候，可以读取文件中的一行数据。



带-p参数可以显示提示信息：



1. cat：显示文件内容



1. grep：用于查找文件里符合条件的字符串



1. 管道命令：管道是一种通信机制，通常用于进程间的通信（也可通过socket进行网络通信），它表现出来的形式将前面每一个进程的输出（stdout）直接作为下一个进程的输入（stdin）



通过管道将ls -al的输出作为 下一个命令less的输入，方便浏览。

（2）查阅shell编程相关资料，编写shell脚本并执行。请提交脚本代码以及代码解释，并附实验过程截图。

1）实现从键盘读取两个数，并比较两个数大小，并打印结果。

提示：echo，read，if

脚本代码如下：

*# 实现从键盘读取两个数，并比较两个数大小，并打印结果*

read -p "Please enter two numbers: " firstNum secondNum

if [ ${firstNum} -gt ${secondNum} ]

then

    echo "${firstNum} is larger than ${secondNum}."

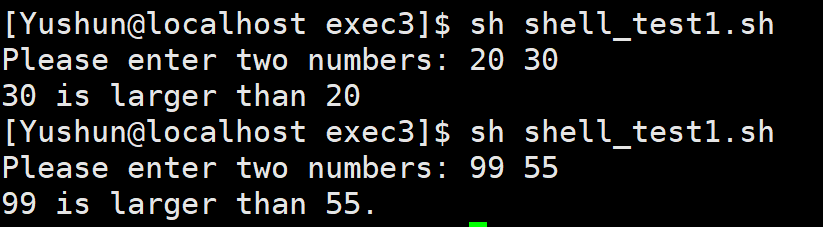
else

    echo "${secondNum} is larger than ${firstNum}"

fi

exit 0

运行截图：



2）实现读取文件的每一行

提示：cat，while，read

*# 实现读取文件的每一行*

read -p "Please enter the file name you want to read: " fileName

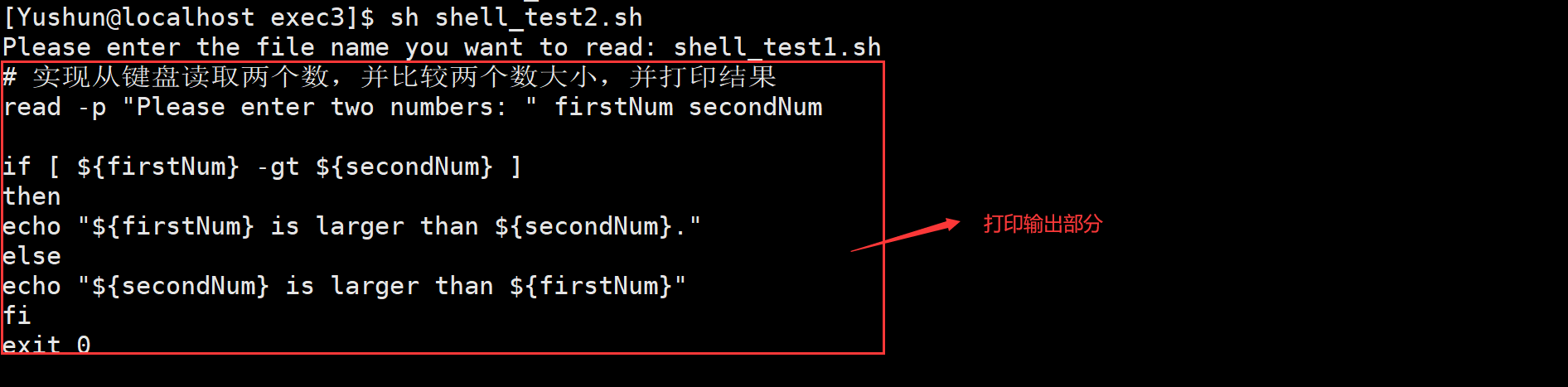
cat ${fileName} | while read line

do

    echo ${line}

done

下面进行测试：



3）把当前目录（包含子目录）下所有后缀为“.txt”的文件后缀变更为“.h”

提示：find，for循环

*# 把当前目录（包含子目录）下所有后缀为“.txt”的文件后缀变更为“.h”*

oldsuffix="txt"

newsuffix="h"

dir=$(eval pwd)

for file in $(find $dir | grep .${oldsuffix})

do

    name=$(find ${file} | cut -d. -f1)

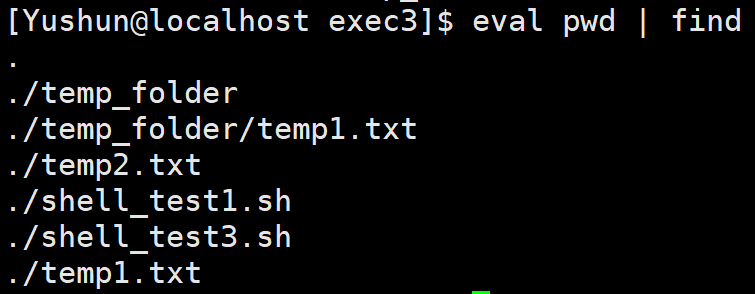
    mv $file ${name}.${newsuffix}

done

echo "All files in the current directory (including subdirectories) with the suffix .txt have been changed to .h"

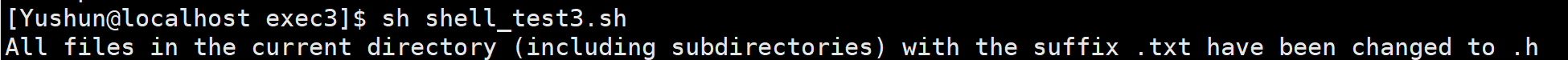
下面进行测试：

先创建一堆.txt以及子目录下的.txt为后缀名的文件

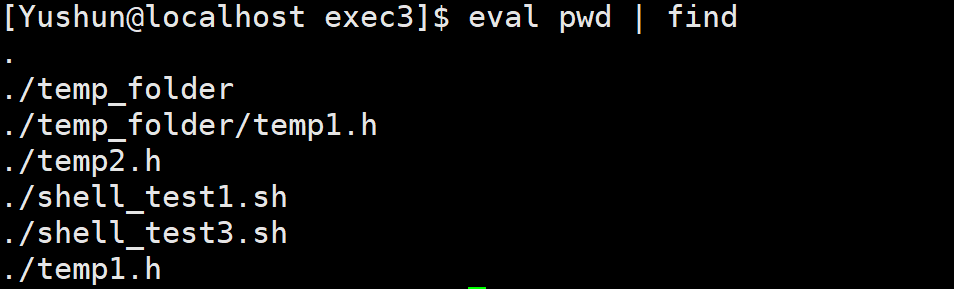


文件结构如上图所示。

运行截图：



我们再查看文件结构



可以看到，所有以.txt为后缀的文件都改为了.h的后缀。

1. 编写make文件实现编译c或c++代码。

提示：make, gcc

下面我将举个写makefile文件的例子

Person.h文件

#include <iostream>

#include <string>

**class** Person

{

**private:**

    std::string name;

**int** age;

**public:**

    Person(std::string, **int**);

**void** printInfo();

};

Person.cpp文件

#include "Person.h"

Person::Person(std::string name, **int** age)

{

    this->name = name;

    this->age = age;

}

**void** Person::printInfo()

{

    std::cout << "name: " << this->name << ", age: " << this->age << std::endl;

}

main.cpp文件

#include <iostream>

#include "Person.h"

**int** main()

{

    Person \*person = new Person("Yushun", 20);

    person->printInfo();

    return 0;

}

接下来试makefile文件

mainProcess: Person.o main.o

    g++ -o mainProcess Person.o main.o

Person.o: Person.cpp Person.h

    g++ -c Person.cpp

main.o: main.cpp

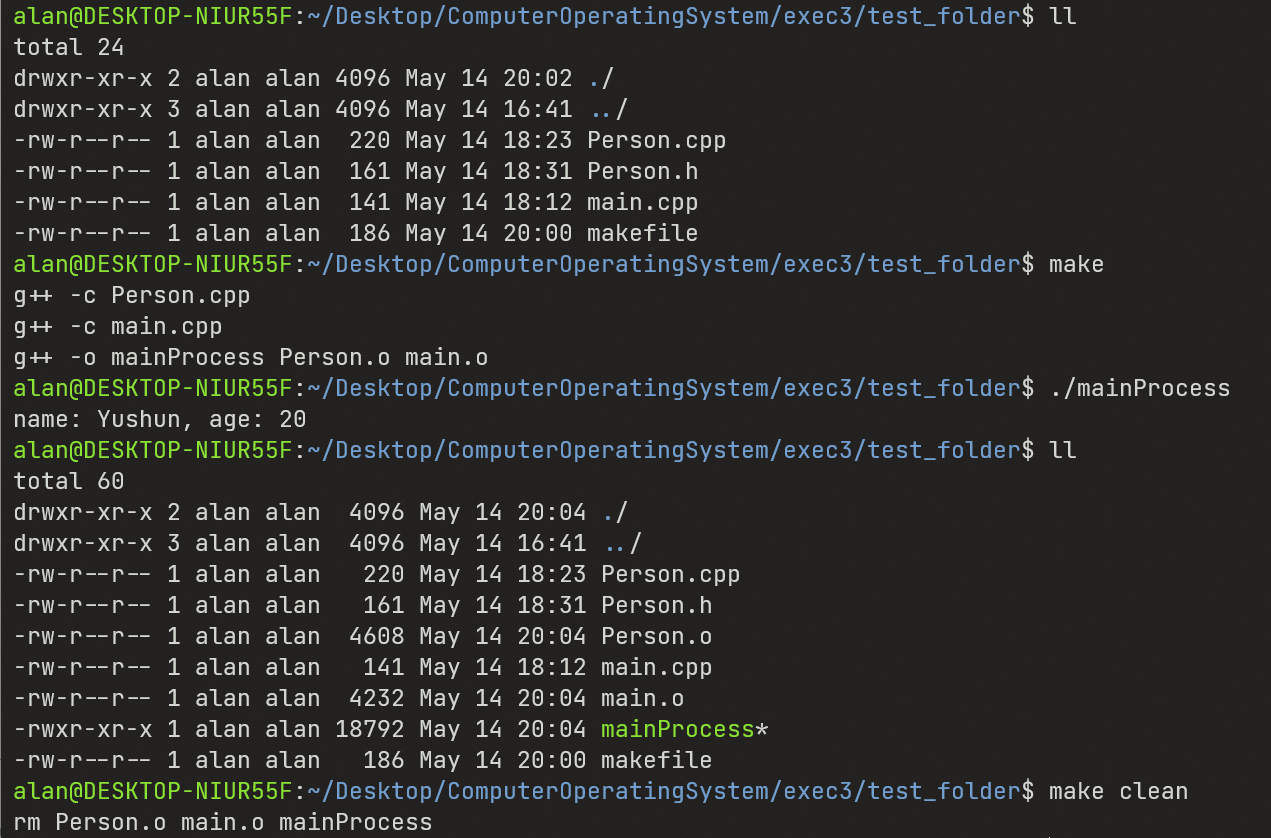
    g++ -c main.cpp

clean:

    rm Person.o main.o mainProcess

下面来演示操作过程：

当我们编辑好Person.cpp, Person.h, main.cpp以及makefile文件后，我们用make指令来编译链接得到目的程序mainProcess，此后我们可以运行./mainProcess来得到结果，最后我们可以用make clean指令来清楚编译中间生成文件。



# 二、进程创建

进程是[操作系统](http://baike.baidu.com/view/880.htm" \t "http://baike.baidu.com/view/_blank)结构的基础；是一次程序的执行；是一个程序及其数据在处理机上顺序执行时所发生的活动。进程是60年代初首先由[麻省理工学院](http://baike.baidu.com/view/1935.htm" \t "http://baike.baidu.com/view/_blank)的[MULTICS系统](http://baike.baidu.com/view/8939006.htm" \t "http://baike.baidu.com/view/_blank)和IBM公司的[CTSS](http://baike.baidu.com/view/3394929.htm" \t "http://baike.baidu.com/view/_blank)/360系统引入的。进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。它不只是程序的[代码](http://baike.baidu.com/view/41.htm" \t "http://baike.baidu.com/view/_blank)，还包括当前的活动，通过[程序计数器](http://baike.baidu.com/view/178145.htm" \t "http://baike.baidu.com/view/_blank)的值和处理[寄存器](http://baike.baidu.com/view/6159.htm" \t "http://baike.baidu.com/view/_blank)的内容来表示。

* 定义

狭义定义：进程就是一段程序的执行过程。

广义定义：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是[操作系统](http://baike.baidu.com/subview/880/4940471.htm" \t "http://baike.baidu.com/view/_blank)动态执行的[基本单元](http://baike.baidu.com/view/693012.htm" \t "http://baike.baidu.com/view/_blank)，在传统的[操作系统](http://baike.baidu.com/view/880.htm" \t "http://baike.baidu.com/view/_blank)中，进程既是基本的[分配单元](http://baike.baidu.com/view/954991.htm" \t "http://baike.baidu.com/view/_blank)，也是基本的执行单元。

进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括[文本](http://baike.baidu.com/view/300107.htm" \t "http://baike.baidu.com/view/_blank)区域（text region）、数据区域（data region）和[堆栈](http://baike.baidu.com/view/93201.htm" \t "http://baike.baidu.com/view/_blank)（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有[处理](http://baike.baidu.com/view/989420.htm" \t "http://baike.baidu.com/view/_blank)器赋予程序生命时，它才能成为一个活动的实体，我们称其为[进程](http://baike.baidu.com/view/19746.htm" \t "http://baike.baidu.com/view/_blank)。

进程是操作系统中最基本、重要的概念。是多道程序系统出现后，为了刻画系统内部出现的动态情况，描述系统内部各道程序的活动规律引进的一个概念,所有多道程序设计操作系统都建立在进程的基础上。

进程和线程的一个简单解释

<http://www.ruanyifeng.com/blog/2013/04/processes_and_threads.html>

下面的链接详细介绍了Linux中fork()函数

<http://blog.csdn.net/jason314/article/details/5640969>

gcc编译器的常用命令

<https://www.jb51.net/article/180825.htm>

关于VI/VIM使用的链接

<https://www.runoob.com/linux/linux-vim.html>

**要求:**

1. 学习进程的基础知识，了解Linux系统中进程创建的基本原理。
2. 熟悉进程的创建、控制、执行和终止等系统调用函数。
3. 安装vim编辑器,使用vim编制一段程序，使用系统调用fork()创建两个子进程，在此程序运行时，系统中就有一个父进程和两个子进程在活动。让每一个进程在屏幕上显示一个字符：父进程显示字符a，两个子进程分别显示字符b和子符c。试观察、记录并分析屏幕上进程调度和并发执行的情况。若在程序中使用系统调用nice()来改变各进程的优先级，会出现什么现象？

4）提交源程序清单，并附加流程图与注释。

并回答以下问题：

1. 系统调用fork()是怎样创建进程的？

2）当首次调用新创建的子进程时，其入口在哪里？

3）分析进程调度和进程并发执行的关系？

1.使用系统调用fork()创建两个子进程

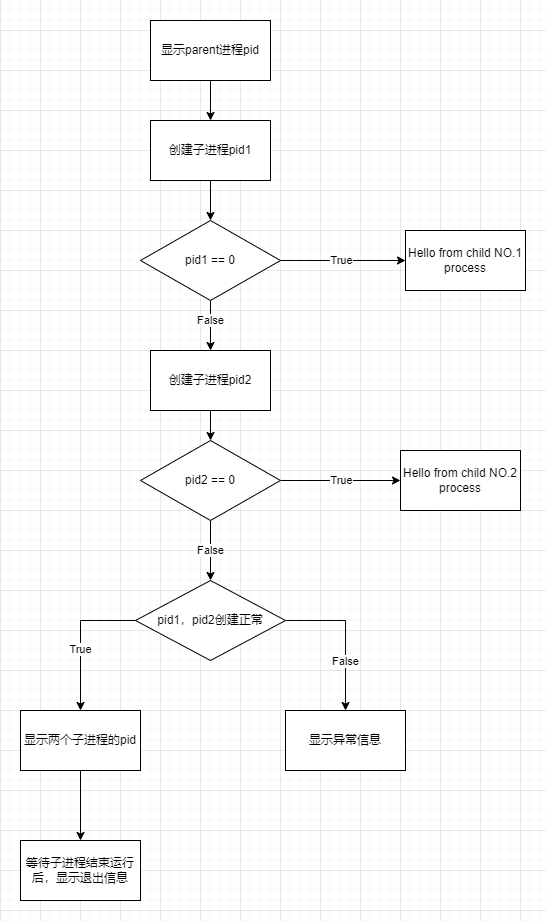
2.各个子进程显示和输出一些提示信息和自己的进程标识符。

3.父进程显示自己的进程ID和一些提示信息，然后调用waitpid()等待多个子进程结束，并在子进程结束

4.考虑到a，b，c不是很直观，我在程序中用了更直观的表示方法

注意：在父进程中调用fork()之后会产生两种结果：一种为分裂子进程失败，另一种就是分裂子进程成功。**如果fork()失败，则返回-1,；否则会出现父进程和子进程两个进程，在子进程中fork()返回0，在父进程中fork()返回子进程的ID。**

**以下是流程图：**



为了分析执行fork前后程序的运行情况，我编写了如下代码：

#include <unistd.h>

#include <stdarg.h>

#include <time.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

**int** main(**void**)

{

    pid\_t pid1, pid2;

**int** i;

    printf("Hello from parent process,pid is %d.\n", getpid());

    pid1 = fork();

    if (pid1 == 0)

    {

        sleep(1);

        for (i = 0; i < 3; i++)

        {

            printf("Hello from child NO.1 process %d. %d times\n", getpid(), i + 1);

            sleep(1);

        }

        return 0;

    }

    pid2 = fork();

    if (pid2 == 0)

    {

        sleep(1);

        for (i = 0; i < 3; i++)

        {

            printf("Hello from child NO.2  process %d. %dtimes\n", getpid(), i + 1);

            sleep(1);

        }

        return 0;

    }

    else if (pid1 != -1 && pid2 != -1)

    {

        printf("parentforked one child pross--%d.\n", pid1);

        printf("parentforked one child pross--%d.\n", pid2);

        printf("parentis waiting for child exit .\n");

        waitpid(pid1, NULL, 0);

        printf("childNO.1 process had exited .\n");

        waitpid(pid2, NULL, 0);

        printf("childNO.2 process had exited .\n");

        printf("parentprocess had exited .\n");

    }

    else

    {

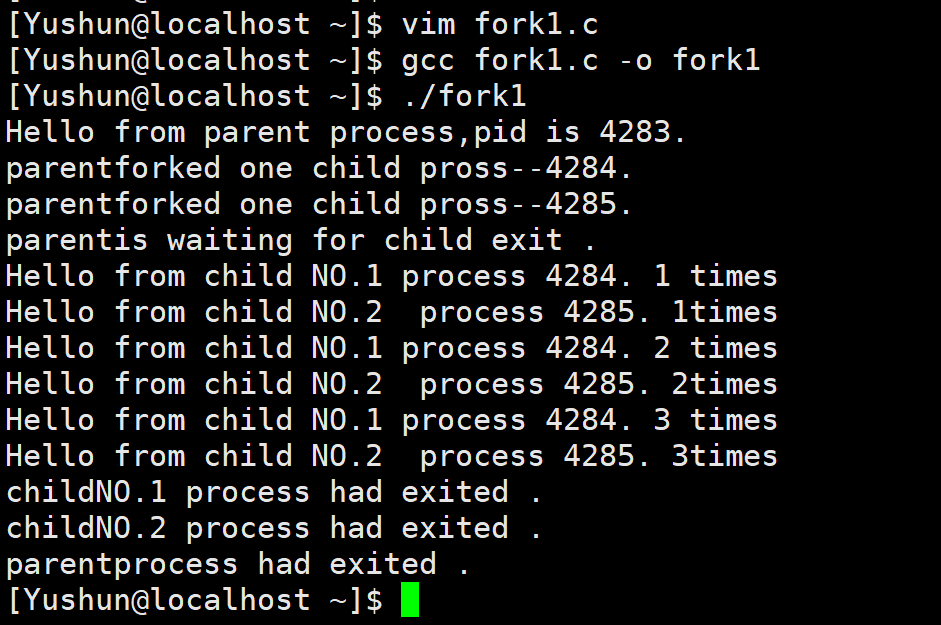
        printf("every thing was done withouterror.\n");

    }

    return 0;

}

以下是编译后执行的运行结果：



分析：

在执行fork函数之前，对于我自己这个进程本身而言，我能看到我的绝对pid是4283.

我定义了两个变量pid1和pid2来承载fork函数的返回值，通过查阅我知道，对于子进程来说，fork的返回值是0，对于父进程来说，fork的返回值是子进程的pid，即4284和4285，

也就是说，我们可以通过判断fork的值来确定一段代码是应该又父进程还是子进程执行，如果fork==0，说明当前代码块在子进程内，如果为-1，代表子进程创建失败，如果为其它，则说明当前代码块又父进程执行。

可以看到，两个子进程是交替执行的。

**使用nice()函数改变进程优先级：**

代码如下：

#include <unistd.h>

#include <stdarg.h>

#include <time.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

**int** main(**void**)

{

    pid\_t pid1, pid2;

**int** i;

    printf("Hello from parent process,pid is %d.\n", getpid());

    pid1 = fork();

    if (pid1 == 0)

    {

        sleep(1);

        for (i = 0; i < 3; i++)

        {

            if (nice(10) == 10)

            {

                printf("A Set Nice Successfully\n");

            }

            printf("Hello from child NO.1 process %d. %d times\n", getpid(), i + 1);

            sleep(1);

        }

        return 0;

    }

    pid2 = fork();

    if (pid2 == 0)

    {

        sleep(1);

        for (i = 0; i < 3; i++)

        {

            if (nice(2) == 2)

            {

                printf("B Set Nice Successfully\n");

            }

            printf("Hello from child NO.2  process %d. %dtimes\n", getpid(), i + 1);

            sleep(1);

        }

        return 0;

    }

    else if (pid1 != -1 && pid2 != -1)

    {

        printf("parentforked one child pross--%d.\n", pid1);

        printf("parentforked one child pross--%d.\n", pid2);

        printf("parentis waiting for child exit .\n");

        waitpid(pid1, NULL, 0);

        printf("childNO.1 process had exited .\n");

        waitpid(pid2, NULL, 0);

        printf("childNO.2 process had exited .\n");

        printf("parentprocess had exited .\n");

    }

    else

    {

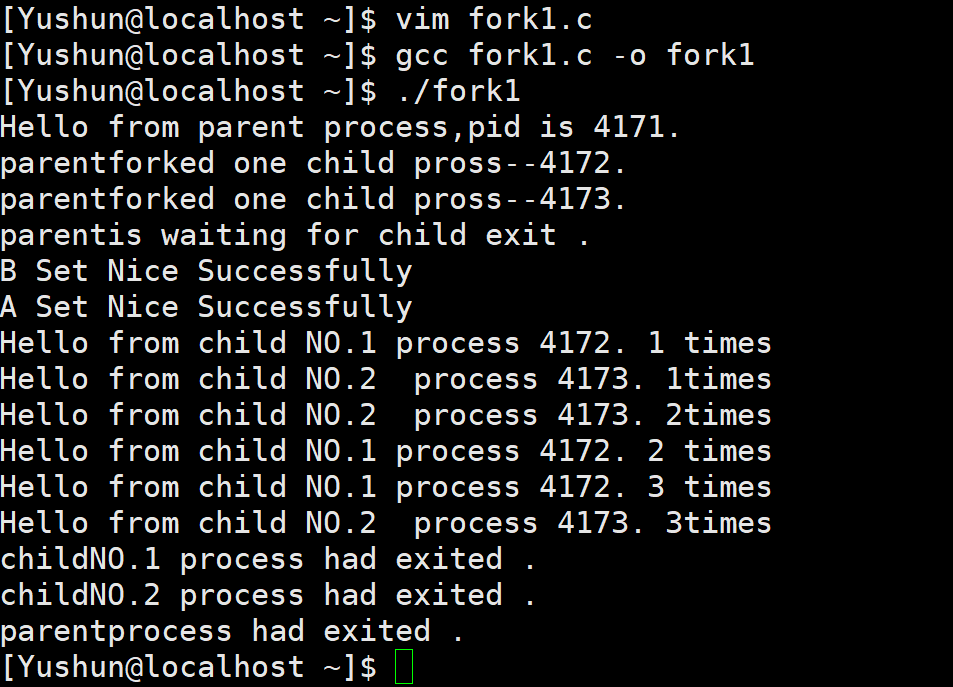
        printf("every thing was done withouterror.\n");

    }

    return 0;

}

以下是编译后执行的运行结果：



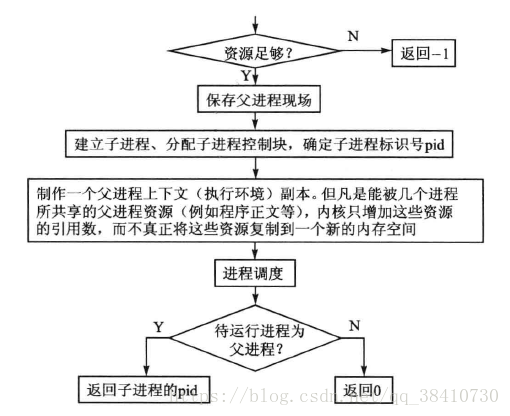
如图分析可得，一个进程的友善值越低，优先级就越高，时间片也越大，nice()会提高一个进程的友善值,而且会返回刚才所更新的值。

在修改优先级后，进程按照设置顺序执行，多次执行输出固定结果。

1）系统调用fork()是怎样创建进程的？

两个进程具有各自的数据区和用户堆栈，在函数fork()生成子进程时，将父进程数据区和用户堆栈的内容分别复制给了子进程。同时，接下来的内容，父进程和子进程都是对自己的数据区和堆栈中的内容进行修改运算了。

系统调用fork()工作流程示意图如下：



也就是说，在fork()函数之前需要确认内核中有足够的资源来完成。如果资源满足要求，则内核slab分配器在相应的缓冲区中构造子进程的进程控制块，并将父进程控制块中的全部成员都复制到子进程的控制块，然后再把子进程控制块必须的私有数据改成子进程的数据。

当fork()返回到用户空间之前，向子进程的内核栈中压入返回值0，而向父进程内核堆栈压入子进程的pid。最后进行一次进程调度，决定是运行子进程还是父进程。

最后说明：父子进程的调度的顺序是由调度器决定的，与进程的创建顺序无关。

1. 当首次调用新创建的子进程时，其入口在哪里？

系统为所有用户进程维护了一个task table，在这里面存放着指向每个进程的进程控制块（PCB）的指针。在某次时钟中断中，轮到一个进程被唤醒（新建进程也得乖乖进入任务队列排队），CPU读取该进程PCB结构中那个指向TTS的指针。各进程的TTS构成一个表，表的段描述符存放于GDT（全局段描述符表）或者LDT（局部段描述符表）。然后CPU读取TTS中的各项数据并且根据它的各项值来设置寄存器，包括CS（代码段选择子）和IP（指令偏移地址），而这就是进程的入口。接下来，CPU开始执行进程的指令。

1. 分析进程调度和进程并发执行的关系？

进程的运行不仅仅需要CPU，还需要很多其他资源，如内存，显卡，GPS，磁盘等等，统称为程序的执行环境，也就是程序上下文。

　在这里就出现了并发的概念，调度器切换CPU给不同进程使用的速度非常快，于是在使用者看来程序是在同时运行，这就是并发，而实际上CPU在同一时刻只在运行一个进程。