操作系统

实验报告

|  |  |
| --- | --- |
| 学生姓名 | 罗健刚 |
| 学 号 | 21312030 |
| 专业班级 | 网络空间安全1班 |
| 学 院 | 网络空间安全学院 |
| 完成时间 | 2023/10/13 |

目录

一、实验目的 ..3

二、实验内容 3

2.1 任务一…………………………………………………………………3

2.2 任务二…………………………………………………………………3

2.3 任务二…………………………………………………………………3

2.4 任务二…………………………………………………………………3

三、实验过程及结果…………………………………………………………………4

3.1 任务一…………………………………………………………………4

3.2 任务二…………………………………………………………………5

3.3 任务三…………………………………………………………………7

3.4 任务四…………………………………………………………………9

四、实验总结…………………………………………………………………………12

1. **实验目的**

(1)了解Linux系统的引导和启动的过程；

(2)了解Linux系统的文件管理架构；

(3)掌握Linux服务启动流程与方法；

(4)掌握在linux环境下进行C语言编程的基本方法；

(5)理解和掌握Linux下获取系统运行信息的方法

1. **实验内容**
2. 任务一：编制一个 Shell 程序，并让该程序在用户登录时自动执行，显示信息；
3. 任务二：编写一个C程序实现 daemon 进程，该进程每隔 10秒执行ps 命令并将当前时间和命令的输出写至文件 ps.log 尾部；
4. 任务三：分别利用文件的系统调用 read、write 和文件的库函数 fread、fwrite 实现文件复制功能，比较在每次读取一字节和 1024 字节时两个程序的执行效率，并分析原因；
5. 任务四：编写一个C程序或Shell程序，通过读取/proc文件系统获取系统的各种信息，如主机名、系统启动时间、运行时间、进程信息、CPU使用率、内存使用率等。实时监测（即周期性刷新显示）CPU和内存的使用情2况，以及它们的利用率
6. **实验过程与结果  
   1 .** **任务一：编制一个 Shell 程序，并让该程序在用户登录时自动执行**  
   （1）编写的Shell程序代码（auto.sh）如下所示：

#! /bin/bash

echo "Welcome! Have a nice day!"

echo System Name:

uname

echo Kernel Vision:

cat /proc/version

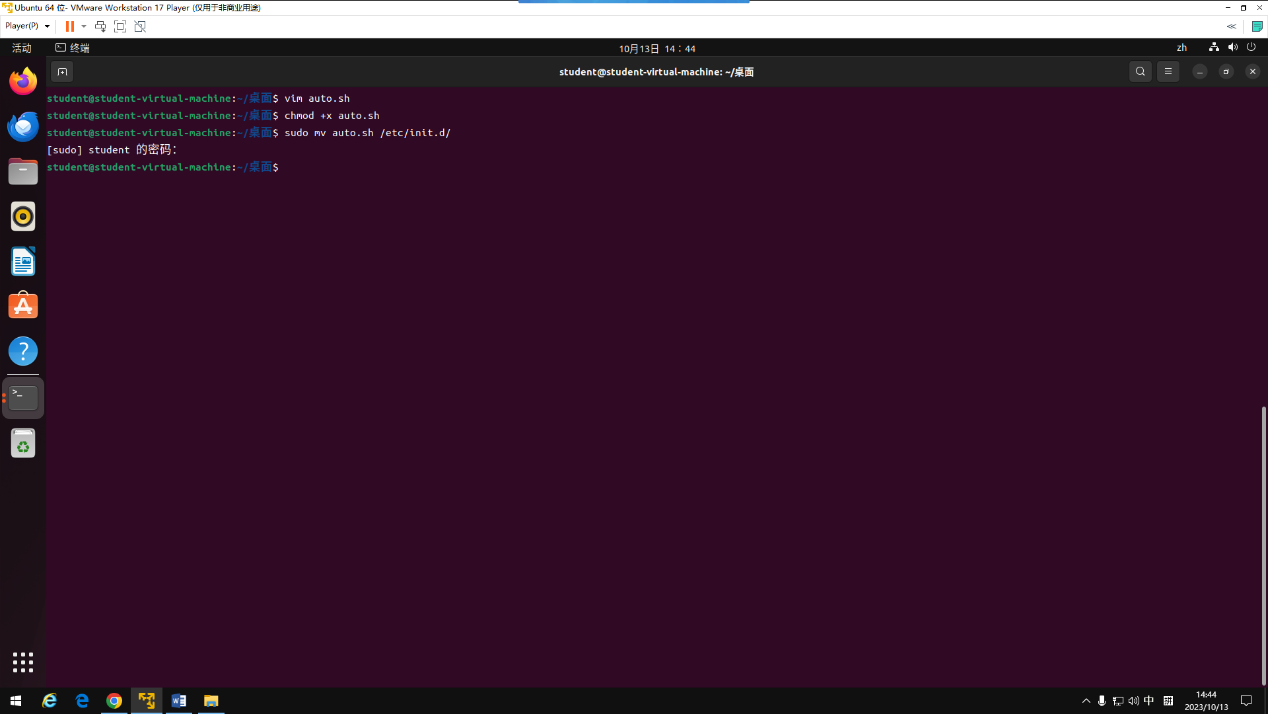
echo Current Directory:

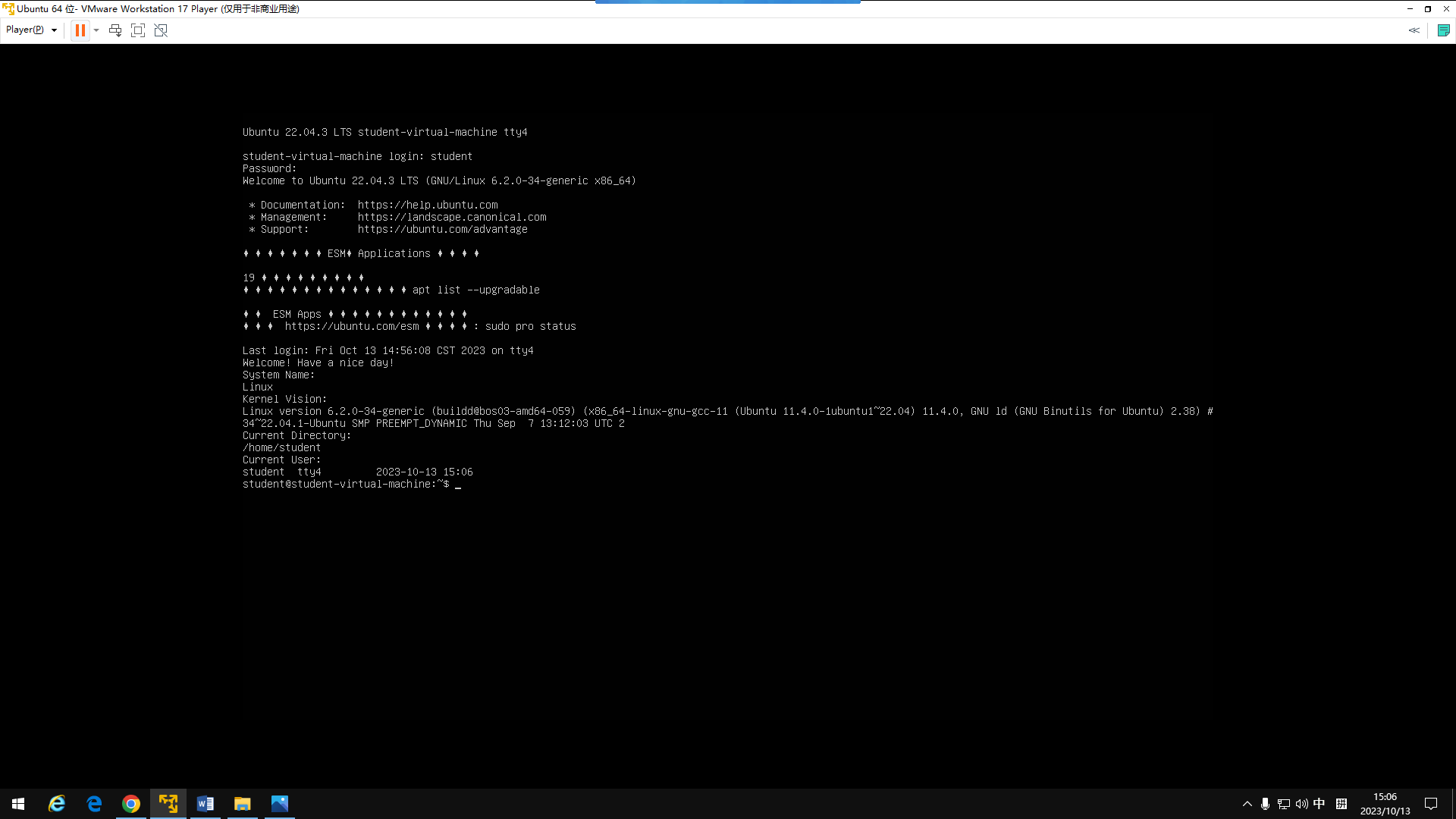
pwd

echo Current User:

who

（2）使用chmod指令更改文件的权限，并且将这个编写好的脚本文件移动到/etc/rc.d/inti.d 之下。移动之后利用chkconfig在chkconfig工具服务列表中增加服务：checkfig -add auto



（3）切换为命令行界面进行登录测试，发现脚本程序成功自启动，结果如下：  


**2. 任务二：编写一个C程序实现守护进程**

（1）为了实现守护进程，我们需要把父进程结束，然后再对子进程进行操作，代码实现如下：

#include <syscall.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int function(){

    //该函数的功能是执行ps 命令并将当前时间和命令的输出写至文件 ps.log 尾部

    char\* info = (char\*)malloc(1024);

    int i = 0 ;

    char file\_name[] = "ps.log";

    FILE\* log = fopen(file\_name,"w");

    for(i=0 ;i<4 ;i++){

        FILE\* fp ;

        fp = popen("ps","r");

        fread(info,1,1024,fp);

        fwrite(info,1,1024,log);

        time\_t this\_time;

        time(&this\_time);

        struct tm \*local\_time = localtime(&this\_time);

        fprintf(log,"\nTime:%d-%d-%d\n",local\_time->tm\_hour,local\_time->tm\_min,local\_time->tm\_sec);

        sleep(10);                  //10秒的间隔

        fflush(log);

        fflush(fp);

        printf("ps command\n");

    }

    return 0;

}

int main(int argc,char\* argv[]){

    int pid = -1;

    pid = fork();

    if(pid == 0){

        printf("Parents Process Exits!\n");

        exit(0);                                    //父进程退出

    }

    else if(pid > 0){

        printf("This is a daemod process!\n");

        pid = setsid();                             //创建一个新的会话

        function();                                 //调用function函数

    }

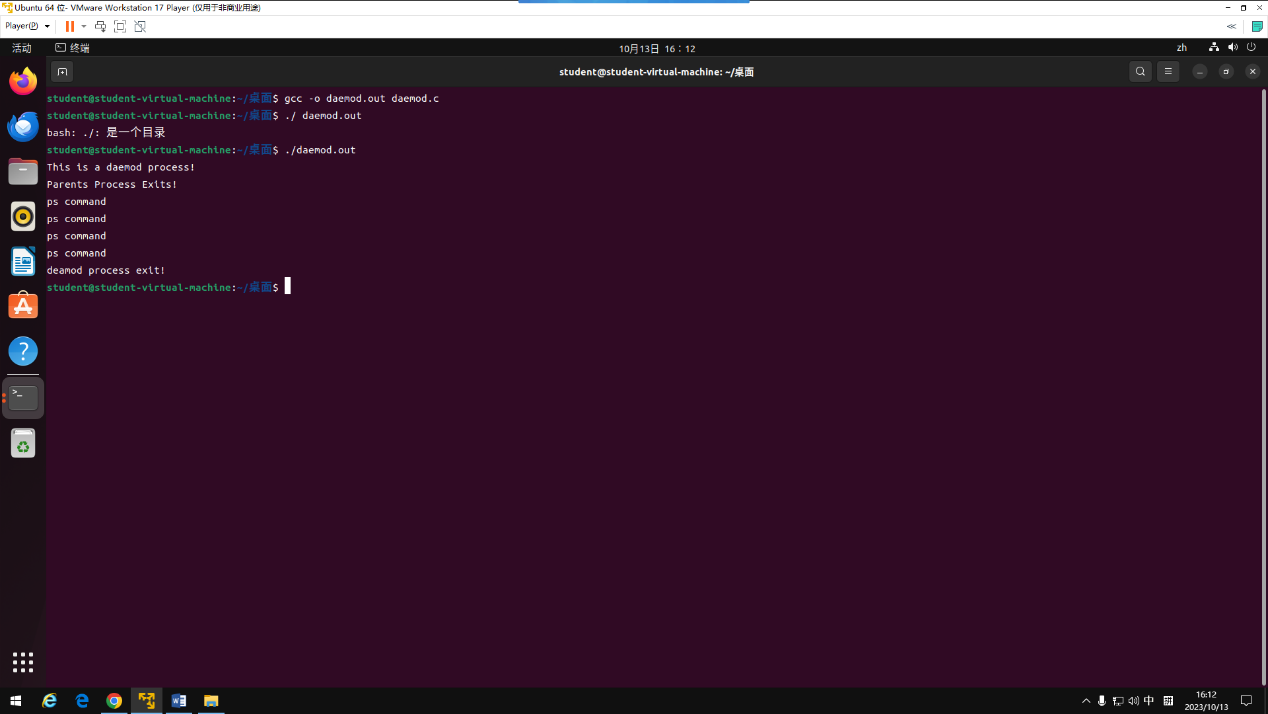
    printf("deamod process exit!\n");

    return 0;

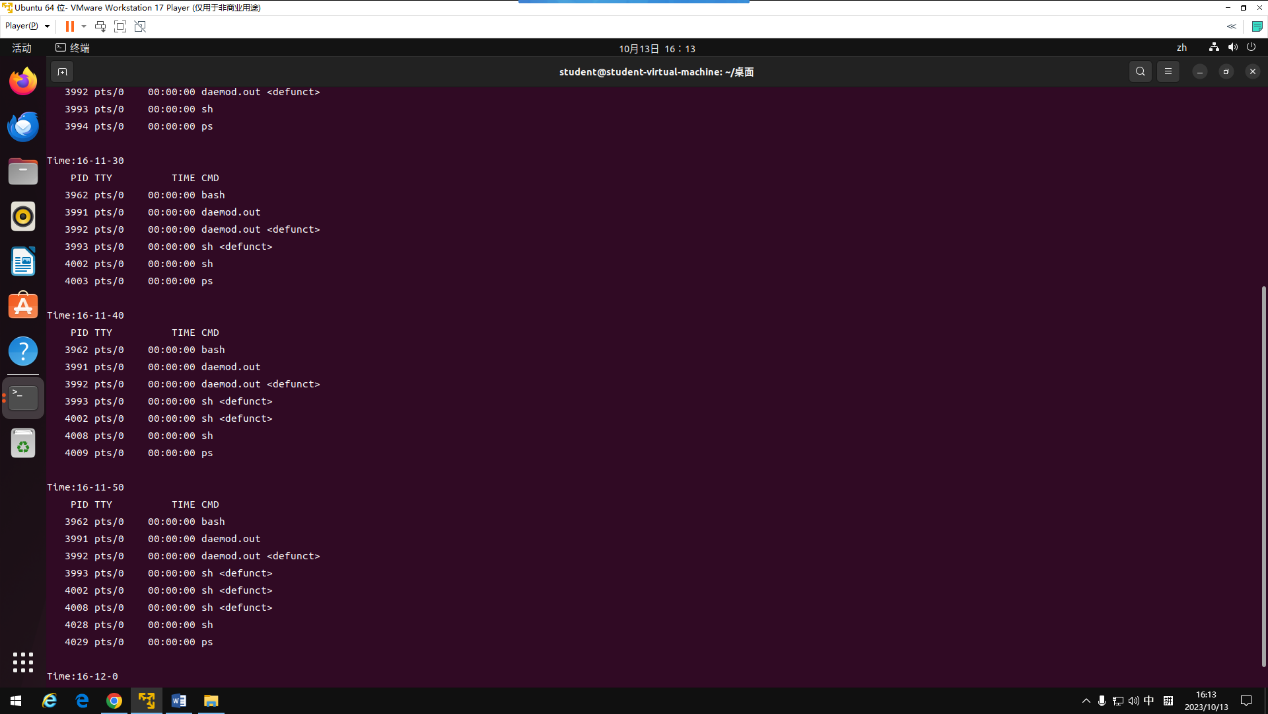
}

（2）使用gcc编译器对源文件进行编译，得到daemod.out可执行文件

（3）运行可执行文件，得到以下结果



（4）打开ps.log文件，内容如下：



图中时间的格式分别为（时-分-秒），可以看出每次写入文件的时间恰好相隔十秒钟

**3.任务三：分别利用文件的系统调用和文件的库函数实现文件复制功能，并且比较效率**

（1）我们对所提及的四种不同情况进行编号：

|  |  |  |
| --- | --- | --- |
| 编号 | 方法 | 单次读取大小（Byte） |
| Condition0 | 库函数 | 1 |
| Condition1 | 库函数 | 1024 |
| Condition2 | 系统调用 | 1 |
| Condition3 | 系统调用 | 1024 |

（2）在同一个程序中依次进行四种情况，通过C语言time.h自带的精确计时功能，记录下四种情况各自需要的时间，代码如下：

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <time.h>

#include <stdlib.h>

int main(int argc,char\* argv[]){

    FILE\* test\_file = fopen("test.txt","r");//源文件

    FILE\* target\_file = fopen("target.txt","r");//复制的目标文件

    struct timespec start\_t,end\_t;

    double total\_t[4];

    int num = 0 ;

    //condition1

    printf("conditon1\n");

    char\* buf = (char\*)malloc(4096\*20);

    clock\_gettime(CLOCK\_MONOTONIC,&start\_t);

    fread(buf,1,4096,test\_file);

    fwrite(buf,1,4096,target\_file);

    clock\_gettime(CLOCK\_MONOTONIC,&end\_t);

    total\_t[num++]=(end\_t.tv\_sec-start\_t.tv\_sec)+(end\_t.tv\_nsec-start\_t.tv\_nsec)/1e9;

    //fflush(test\_file);

    //fflush(target\_file);

    //condition2

    printf("conditon2\n");

    free(buf);

    buf = (char\*)malloc(4096);

    clock\_gettime(CLOCK\_MONOTONIC,&start\_t);

    fread(buf,1024,4096,test\_file);

    fwrite(buf,1024,4096,target\_file);

    clock\_gettime(CLOCK\_MONOTONIC,&end\_t);

    total\_t[num++]=(end\_t.tv\_sec-start\_t.tv\_sec)+(end\_t.tv\_nsec-start\_t.tv\_nsec)/1e9;

    //fflush(test\_file);

    //fflush(target\_file);

    fclose(test\_file);

    fclose(target\_file);

    int test = open("test.txt",0);

    int target = open("target.txt",1);

    //condition3

    printf("conditon3\n");

    free(buf);

    buf = (char\*)malloc(4096);

    clock\_gettime(CLOCK\_MONOTONIC,&start\_t);

    read(test,buf,1);

    write(target,buf,1);

    clock\_gettime(CLOCK\_MONOTONIC,&end\_t);

    total\_t[num++]=(end\_t.tv\_sec-start\_t.tv\_sec)+(end\_t.tv\_nsec-start\_t.tv\_nsec)/1e9;

    //conditon4

    printf("conditon4\n");

    free(buf);

    buf = (char\*)malloc(4096);

    clock\_gettime(CLOCK\_MONOTONIC,&start\_t);

    read(test,buf,1024);

    write(target,buf,1024);

    clock\_gettime(CLOCK\_MONOTONIC,&end\_t);

    total\_t[num++]=(end\_t.tv\_sec-start\_t.tv\_sec)+(end\_t.tv\_nsec-start\_t.tv\_nsec)/1e9;

    close(test);

    close(target);

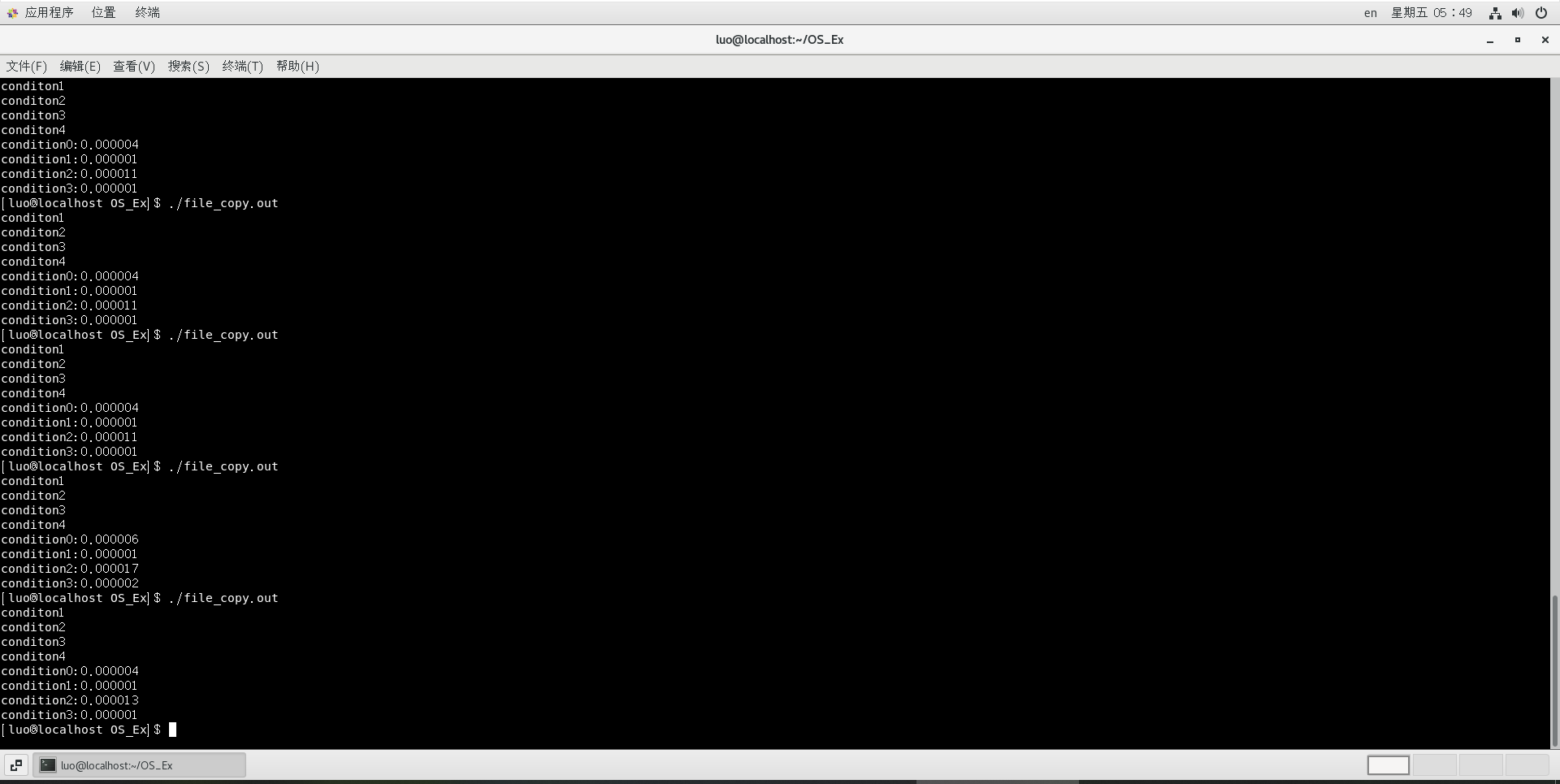
    for( ; num>0;num--){

        printf("condition%d:%lf\n",4-num,total\_t[4-num]);

    }

    return 0;

}

（3）通过gcc编译器编译之后，再执行目标文件，得到如下的运行结果：（为了消除无关因素的影响，重复进行了十次实验）  


得到的结果如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状况 | Condition1 | Condition2 | Condition3 | Condition4 |
| AVG TIME（相对比例） | 4.8 | 1 | 15.1 | 1.6 |

系统调用慢于库函数：使用系统调用会影响系统的性能。与函数调用相比，系统调用的开销要大些，因为在执行系统调用时，Linux必须从运行用户代码切换到执行内核代码，然后再返回用户代码；而库函数可以自动分配缓存,速度相比于系统调用更快；

一次性读取1KB字节流的复制速度快于同等条件下1B的情况：每次读取的字节数量小，其访问内存的次数更多，造成更大的数据存取开销；

**4. 任务四：读取/proc文件系统获取系统的各种信息并实时监测**

（1）/proc文件系统中的部分文件存放的信息如下：

1. /proc/cpuinfo：包含有关系统CPU的信息，如CPU型号、厂商、速度等。
2. /proc/meminfo：提供有关系统内存的信息，包括总内存量、可用内存量、缓存和交换空间等。
3. /proc/loadavg：显示系统的平均负载情况，包括最近1分钟、5分钟和15分钟的平均值。
4. /proc/version：显示当前运行的内核版本信息。
5. /proc/net/dev：提供有关网络设备的统计信息，包括接收和发送的数据包数量、错误数量等。
6. /proc/mounts：列出当前已挂载的文件系统和其挂载点。
7. /proc/PID/status：PID是一个数字，代表一个正在运行的进程的标识符。通过替换PID，可以访问每个运行进程的状态信息，如进程的状态、内存使用情况、打开文件的数量等。
8. /proc/sys/kernel/hostname：显示系统的主机名

（2）根据上述信息，编写代码如下：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <fcntl.h>

#include <syscall.h>

#define CPU\_INFO "/proc/loadavg"

#define MEMORY\_INFO "/proc/meminfo"

#define SYS\_INFO "/proc/stat"

#define UPTIME\_INFO "/proc/uptime"

void display\_sys\_name(){

    printf("\n-------------------------------\n");

    printf("System Name:\nLinux\n");

    printf("\n-------------------------------\n");

}

void display\_uptime(){

    printf("System Up Time:\n");

    printf("Uptime Idletime\n");

    char\* str = (char\*)malloc(128);

    FILE\* fp = fopen(UPTIME\_INFO,"r");

    fgets(str, 128 , fp);

    printf("%s",str);

    free(str);

    fclose(fp);

    printf("\n-------------------------------\n");

}

void display\_cpu(){

    printf("Load Average:\n");

    printf("1min 3min 15min task-number lastest-task\n");

    char\* str = (char\*)malloc(128);

    FILE\* fp = fopen(CPU\_INFO,"r");

    fgets(str, 128 , fp);

    printf("%s",str);

    free(str);

    fclose(fp);

    printf("\n-------------------------------\n");

}

void display\_mem(){

    printf("Memory Info:\n");

    int i = 0 ;

    FILE\* fp = fopen(MEMORY\_INFO,"r");

    for( i = 0 ; i < 7 ; i++){

        char\* str = (char\*)malloc(128);

        fgets(str, 128 , fp);

        printf("%s\n",str);

        free(str);

    }

    printf("\n-------------------------------\n");

    fclose(fp);

}

void display\_process(){

    printf("Process Info:\n");

    FILE\* fp ;

    fp = popen("ps -a","r");

    int i = 0;

    char\* str = (char\*)malloc(128);

    while( fgets(str, 128 , fp)!=NULL){

        printf("%s",str);

    }

    fclose(fp);

    printf("\n-------------------------------\n");

}

int main(int argc,int\* argv[]){

    int i = 0 ;

    for(;i < 1000; i ++){

        display\_sys\_name();

        display\_uptime();

        display\_cpu();

        display\_mem();

        display\_process();

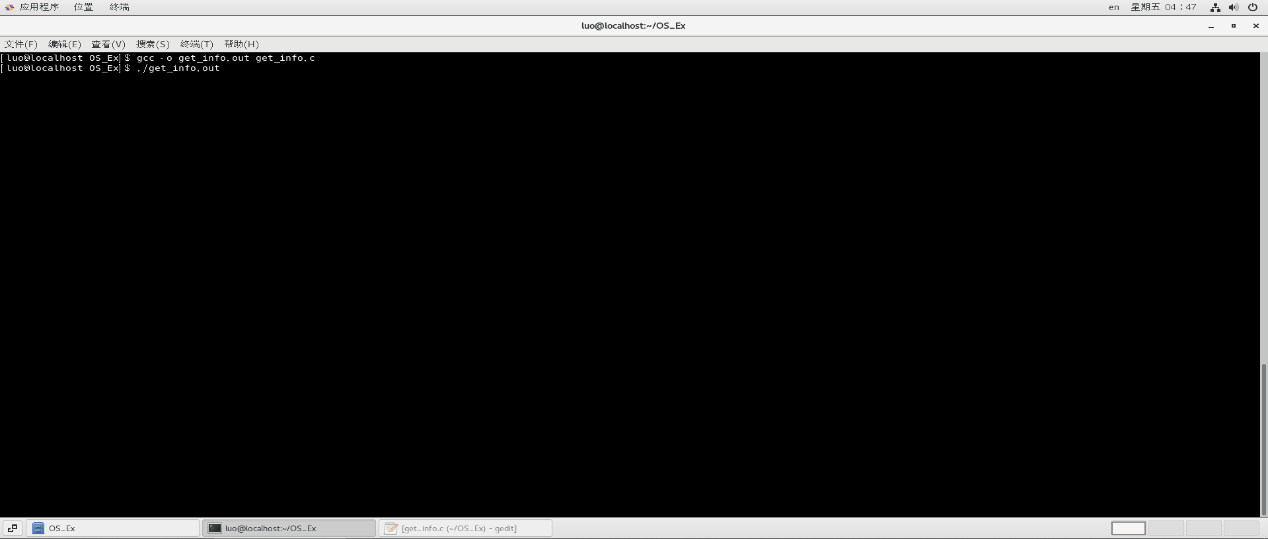
        sleep(2);           //两秒更新一次

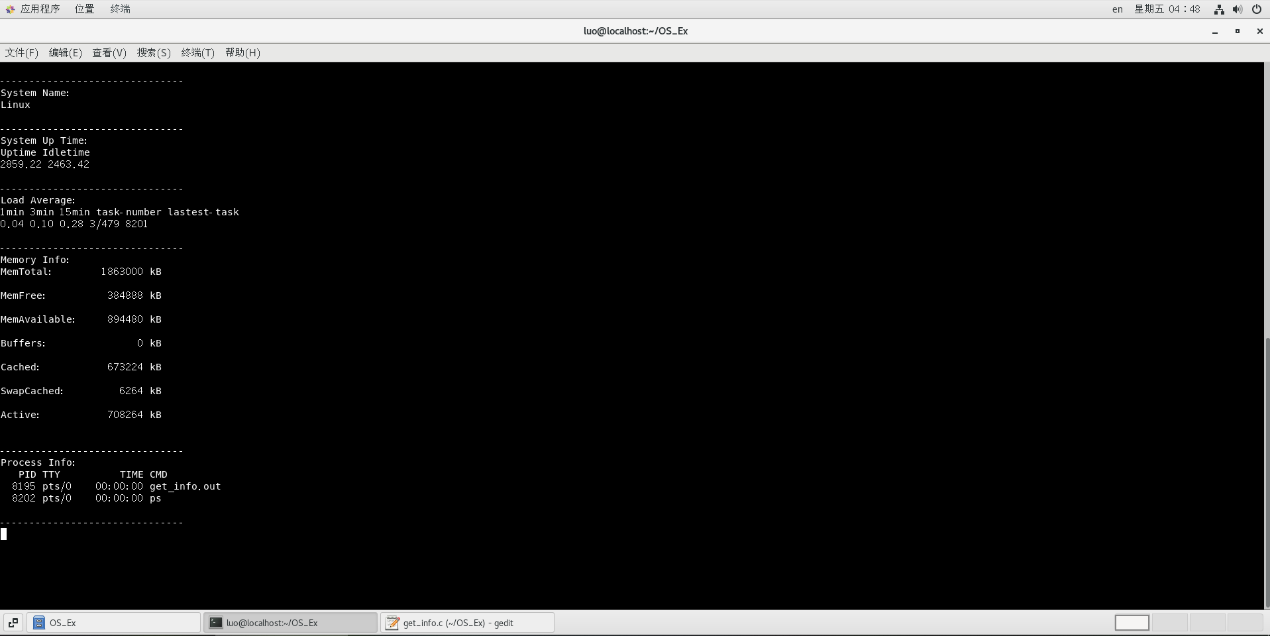
        system("clear");

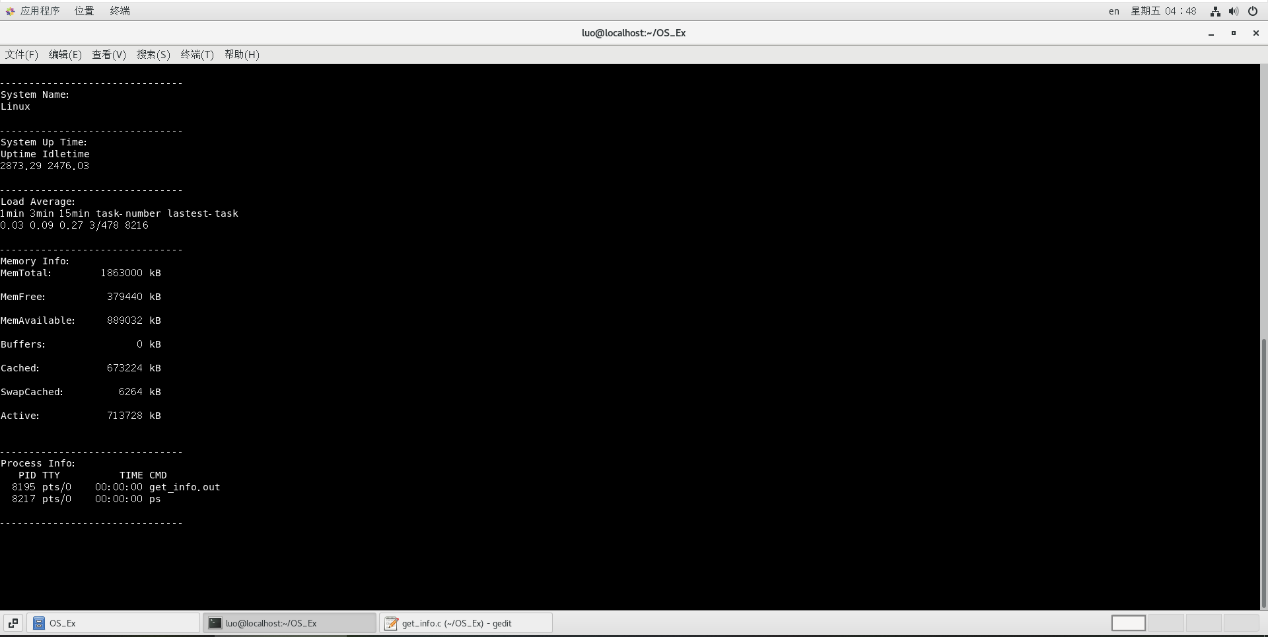
    }

}

（3）编译并运行文件，运行结果如下：







1. **实验总结**
2. 在Linux系统中，有多种方式可以让脚本在启动时自动执行：
3. 使用init.d或rc.d脚本：在某些Linux发行版中，可以将脚本放置在/etc/init.d/或/etc/rc.d/目录中，并使用适当的权限设置。然后，可以使用update-rc.d或chkconfig等工具将脚本添加到启动过程中；
4. 使用systemd服务：Systemd是现代Linux系统中广泛使用的初始化系统。可以创建一个Systemd服务单元文件（通常位于/etc/systemd/system/目录），定义脚本的启动方式和依赖关系。然后，使用systemctl命令来启用和管理该服务；
5. 使用用户的登录脚本：对于特定用户，可以将脚本添加到他们的登录文件中。例如，对于Bash shell，可以将脚本添加到~/.bashrc或~/.bash\_profile文件中，以在用户登录时自动执行；

2.创建守护进程的一般步骤：

（1）创建子进程，终止父进程。由于守护进程是脱离控制终端的，因此首先创建子进程，终止父进程，使得程序在shell终端里造成一个已经运行完毕的假象；

（2）在子进程中创建新会话。setsid函数用于创建一个新的会话，并担任该会话组的组长。调用setsid可以让进程摆脱原会话的控制、让进程摆脱原进程组的控制和让进程摆脱原控制终端的控制；

（3）改变工作目录。（可选）由于在进程运行过程中，当前目录所在的文件系统不能卸载，可以把当前工作目录换成其他的路径；

（4）重设文件创建掩码（可选）。由于使用fork函数新建的子进程继承了父进程的文件创建掩码，把文件创建掩码设置为0，可以大大增强该守护进程的灵活性；

（5）关闭文件描述符（可选）。用fork新建的子进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读或写，但它们一样消耗系统资源，可能导致所在的文件系统无法卸载；

3.下面给出两个关于文件拷贝不同的猜想：

（1）与cache的关系：1024字节移动效率大于1字节的原因除了读取内存次数更少之外，也和cache的大小有关，当一次内存访问恰好和cache块的大小一致或者接近时，存取速度大大提升；

（2）与虚拟机的关系：由于本次实验所用系统为虚拟机的系统，在一定程度上无法反映硬件在其中所占的影响，并且虚拟机在进行操作时受虚拟机管理程序的影响较大，本次实验并不能反映出操作系统的真实性能；