**Laboratory Manual for**

**CSE2005- OPERATING SYSTEM LAB**

**Bachelor of Technology in**

**Computer Science and Engineering**

at

**VIT University**

Chennai – 600 127, Tamil Nadu, India



**School of Computing Science and Engineering**

***Table of Content***

|  |  |  |
| --- | --- | --- |
| Sl.No | Content | Page No |
| 1 | **Objectives and outcomes** | **2** |
| 2 | **List of Experiments** | **4** |
| 3 | **Introduction** | **5** |
| 4 | **Description** | **5** |
| 5 | **Exercise 1** | **6** |
| 6 | **Exercise 2** | **9** |
| 7 | **Exercise 3** | **13** |
| 8 | **Exercise 4** | **16** |
| 9 | **Exercise 5** | **18** |
| 10 | **Exercise 6** | **21** |
| 11 | **Exercise 7** | **25** |
| 12 | **Exercise 8** | **29** |

Objective :

On completion of the course, the students will be able to

1. Use Kernel level programming to design and implement OS components.
2. Analyse OS components using different tools.

Expected Outcome:

This course meets the following student outcomes:

1) An ability to analyze a problem, identify and define the computing requirements appropriate to its solution.

2) An ability to design, implement and evaluate a system / computer based system process, component or program to meet desired needs

3) An ability to identify, formulate and solve engineering problems.

4) An ability to use current techniques, skills and tools necessary for computing and engineering practice.

**LIST OF EXPERIMENTS**

1. Write a boot loader - to load a particular OS say TinyOS/ KolibriOS image - code  
to access from BIOS to loading the OS - involves little assembly code – may use  
QEMU/virtual machines for emulation of hardware.

2. Recompile kernel with you own program for 'cat'. Your 'cat' should read and display  
contents of file on screen, check for errors, do it for multiple files while taking input  
via command line.

3. Create and execute a system call in user/privileged mode in ARM/X86 processor.  
Make it part of the kernel.

4. Write a program to put a process to sleep and then wake it up and then kill it when  
completed.

5. Compare the task creation times. Execute a process and kernel thread, determine  
the time taken to create and run the threads.

6. Compare the overhead of a system call with a procedure call. What is the cost of a  
minimal system call?

7. Determine the latency of individual integer access times in main memory, L1  
Cache and L2 Cache. Plot the results in log of memory accessed vs average latency

8. Determine the file read time for sequential and random access based of varying  
sizes of the files. Take care not to read from cached data - used the raw device  
interface. Draw a graph log/log plot of size of file vs average per-block time.

###### 1. Introduction

The XV6 which is a learning OS may be used as a base template for adding OS modules and improving its functionality to create a working OS by the end of the course. Alternate  
base operating systems could be NachOS/PintOS/Xinu/MTX operating system.  
Emulators to use are Bochs/QEMU.

###### 2. Description

###### System with Linux/ Ubundu is used. VMware is used for virtualization and kernel modifications are tested on installed Ubundu OS.

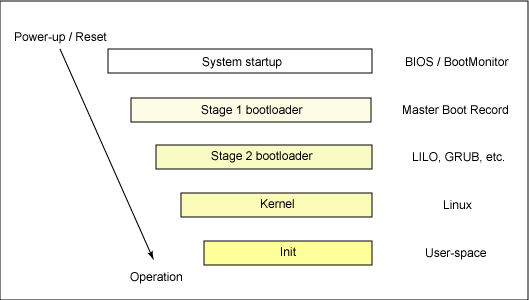
###### EX NO : 1 Write a boot loader - to load a particular OS. OS image - code to access from BIOS to loading the OS

**Aim :**

To Study about boot loader fucntion

Reference:

# Boot Sequence



## 

## 

## POST

When a computer is switched on or reset, it runs through a series of diagnostics called POST - **P**ower-**O**n **S**elf-**T**est. This sequence culminates in locating a bootable device, such as a floppy, cdrom or a harddisk in the order that the firmware is configured to.

## The Bootsector

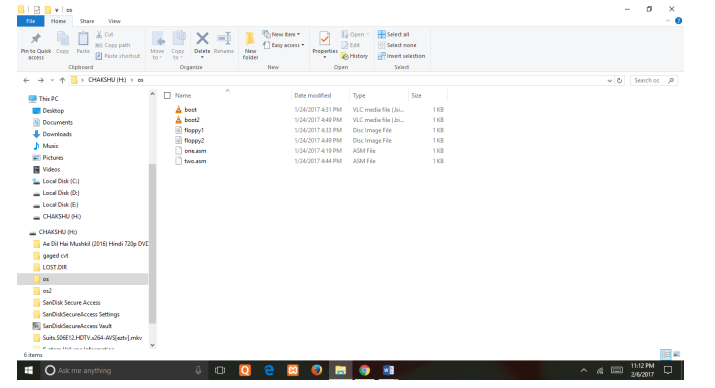
The first 512 bytes of a disk are known as the **bootsector** or **Master Boot Record**. The boot sector is an area of the disk reserved for booting purposes. If the bootsector of a disk contains a valid boot sector (the last word of the sector must contain the signature 0xAA55), then the disk is treated by the BIOS as bootable.

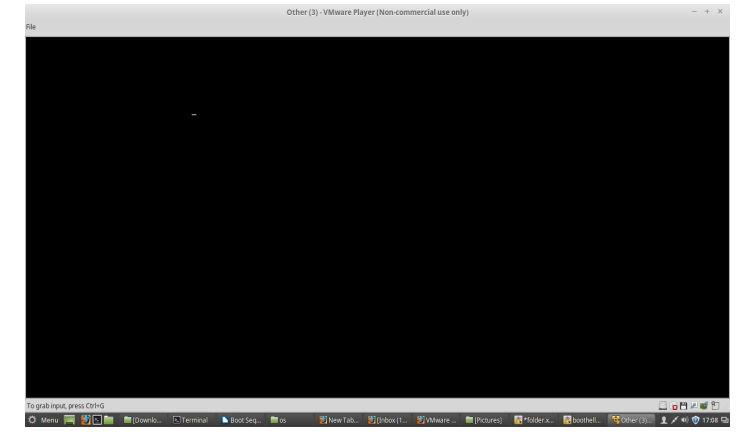
To boot an operating system, the BIOS runtime searches for devices that are both active and bootable in the order of preference defined by the complementary metal oxide semiconductor (CMOS) settings. A boot device can be a floppy disk, a CD-ROM, a partition on a hard disk, a device on the network, or even a USB flash memory stick.

Commonly, Linux is booted from a hard disk, where the Master Boot Record (MBR) contains the primary boot loader. The MBR is a 512-byte sector, located in the first sector on the disk (sector 1 of cylinder 0, head 0). After the MBR is loaded into RAM, the BIOS yields control to it.

<http://www.ibm.com/developerworks/library/l-linuxboot/index.html#resources>

Sample Output :





EX NO : 2 Recompile kernel with you own program.

Aim:

To install kernel of our choice in our virtual machine. To do the following in the environment of Linux Ubuntu 14.04 with the help of virtual machine player that contains it.  
• By adding a system call directly to the kernel and compiling it  
• By adding a system call to the module and then in turn adding the module to the kernel to do the same.

Reference:

1) Installing a linux kernel  
2) Implementing system call in the kernel  
3) Implementing it by creating modules and adding it to modularized kernel.

**1.Implement a system call in Linux kernel 4.9.6**

**Procedure:**

**sudo apt-get update**

sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils libssl-dev bc

**uname -r**

**download latest linux kernel from kernel.org**

tar xf linux-4.9.6.tar.xz

cd linux-4.9.6

**creating directory /info**

**mkdir info**

**cd info/**

**creating file processInfo.h**

asmlinkage longsys\_listProcessInfo(void);

**create file listProcessInfo.c**

#include<linux/kernel.h>

#include<linux/init.h>

#include<linux/sched.h>

#include<linux/syscalls.h>

#include "processInfo.h"

asmlinkage long sys\_listProcessInfo(void) {

struct task\_struct \*proces;

for\_each\_process(proces) {

printk(

"Process: %s\n \

PID\_Number: %ld\n \

Process State: %ld\n \

Priority: %ld\n \

RT\_Priority: %ld\n \

Static Priority: %ld\n \

Normal Priority: %ld\n", \

proces->comm, \

(long)task\_pid\_nr(proces), \

(long)proces->state, \

(long)proces->prio, \

(long)proces->rt\_priority, \

(long)proces->static\_prio, \

(long)proces->normal\_prio \

);

if(proces->parent)

printk(

"Parent process: %s, \

PID\_Number: %ld", \

proces->parent->comm, \

(long)task\_pid\_nr(proces->parent) \

);

printk("\n\n");

}

return 0;

}

**create file Makefile**

obj-y:=listProcessInfo.o

#### Modifying necessary kernel files to integrate our system call:

Add the new ‘info’ directory to the kernel’s Makefile.

For this, open the kernel's Makefile (found in the linux-4.7.1 directory) and look for the following line:

**core -y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/**

(highlighted in the screenshot attached below) And, change it to include **info/.**

It should read:

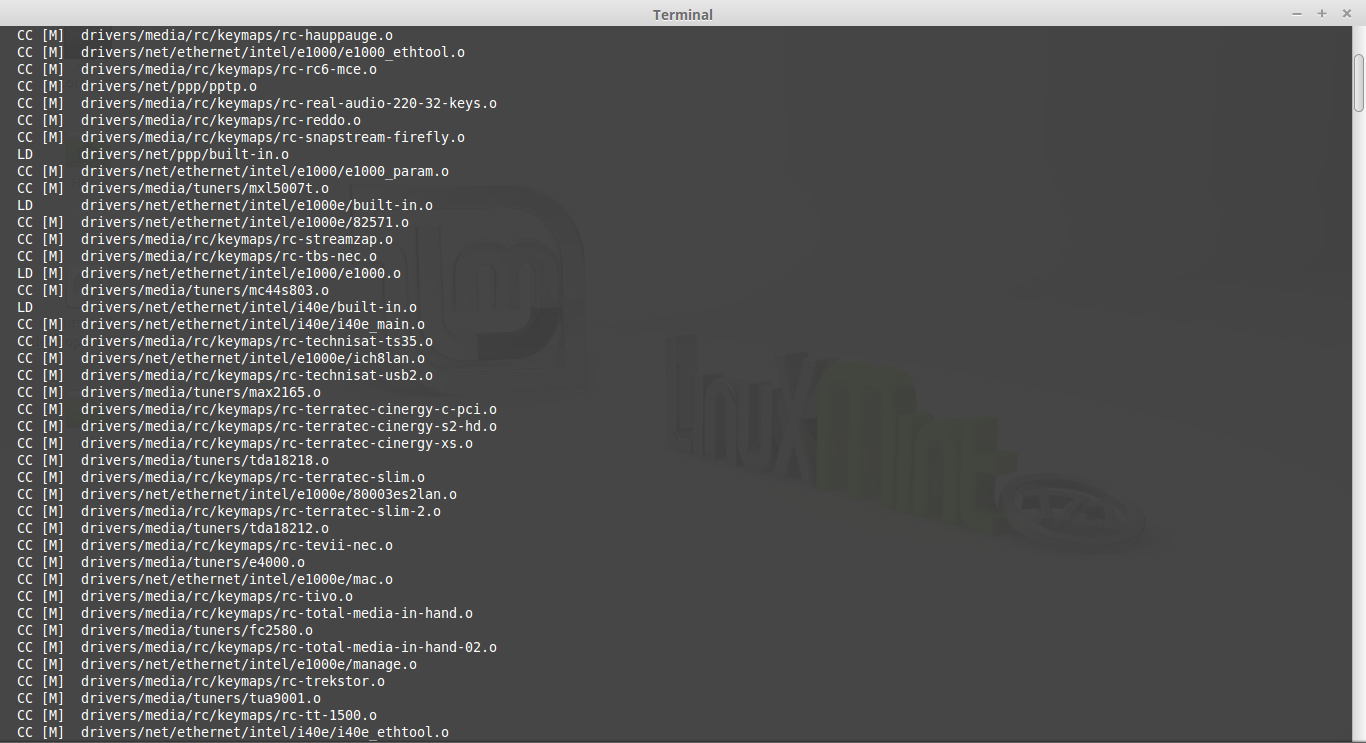
**core -y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ info/**

**Making changes in syscall.h**

Add the following line to the end of the file (before the #endif) as shown:  
**asmlinkage long sys\_hello(void)**

**Recompile and reboot**

**sudo make -j 4 && sudo make modules\_install -j 4 && sudo make install -j 4**

**screenshot 1.png**

**Testing the system call**

**create test.c**

#include <stdio.h>

#include <linux/kernel.h>

#include <sys/syscall.h>

#include <unistd.h>

int main()

{

printf("Invoking 'listProcessInfo' system call");

long int ret\_status = syscall(323); // 323 is the syscall number

if(ret\_status == 0)

printf("System call 'listProcessInfo' executed correctly. Use dmesg to check processInfo\n");

else

printf("System call 'listProcessInfo' did not execute as expected\n");

return 0;

}

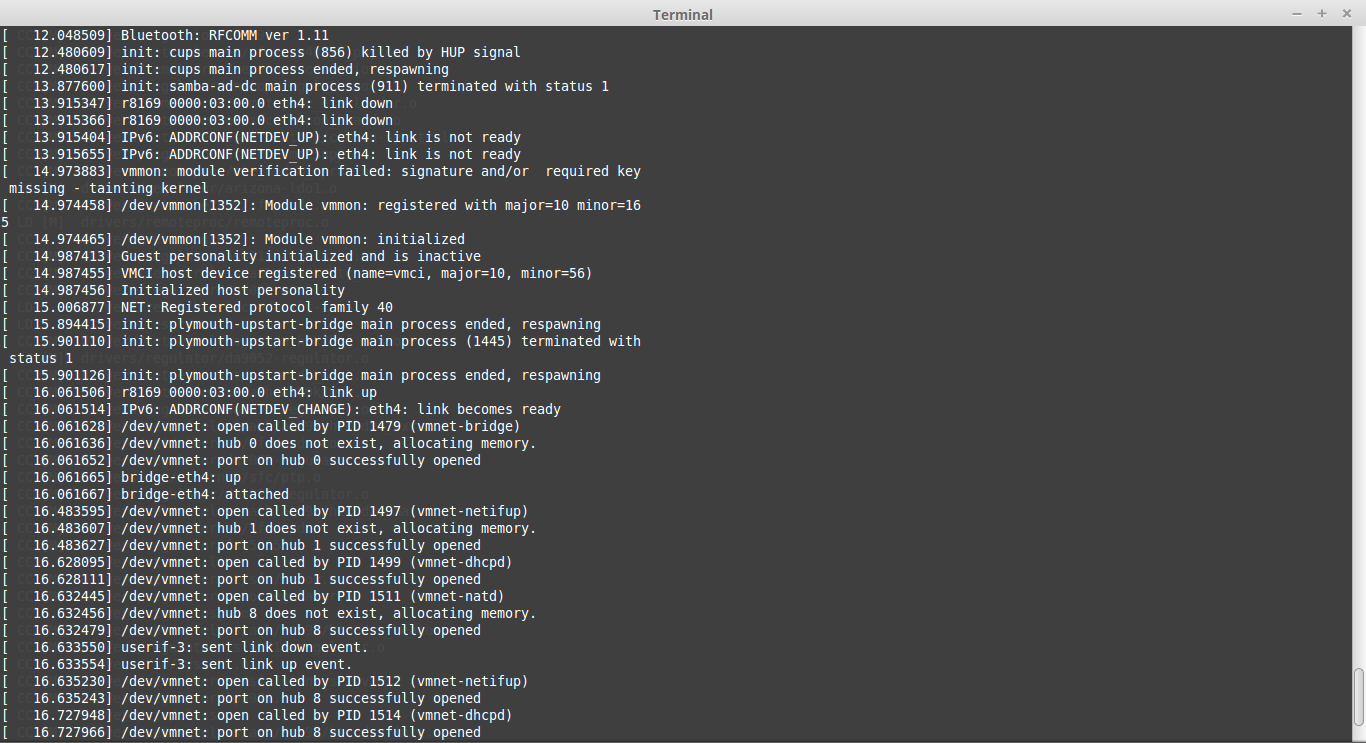
**Compile and run using commands**

**gcc test.c -o sagar**

**./sagar**

**Check the kernel log to which we print the process info use following command**

**dmesg**



Ex: No 3: Write own CAT function

**Introduction And Overview**

This would show how the bootloader and kernals are compiled and can be simulated in VM Ware.

The following codes are compiled and exicuted in vmware.

**Topics**

* Boot Loader
* Kernal
* nasm
* BIOS
* intrupts
* Identifying processes
* File descriptors
* fork
* execv
* waitpid/exit
* Other system calls
* Scheduling
* Synchronization issues

**Plan of Action**

The codes are compiled and the image is exicuted in VM Ware and the output

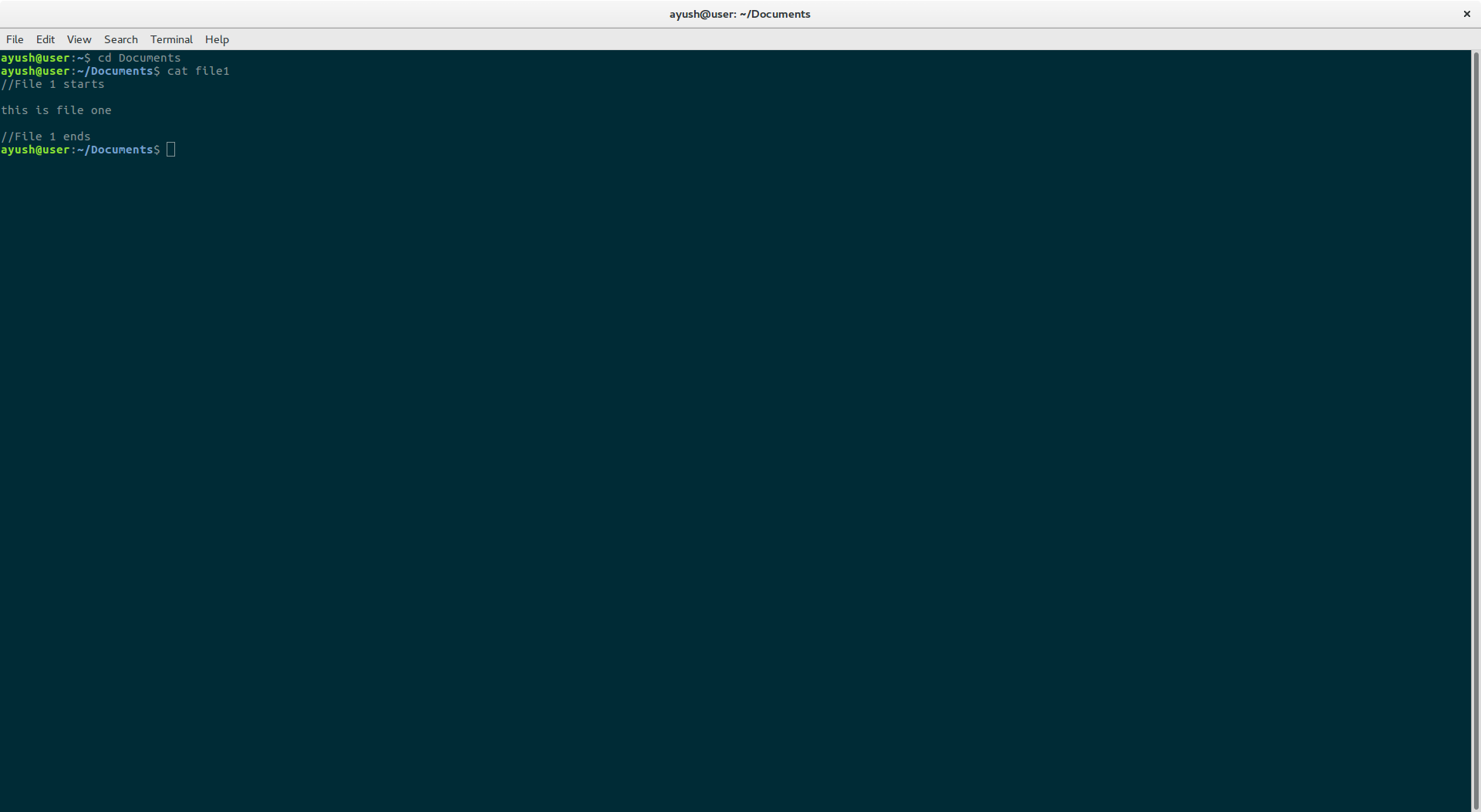
is provided below each Bootloder examples.

Creat cat program using c

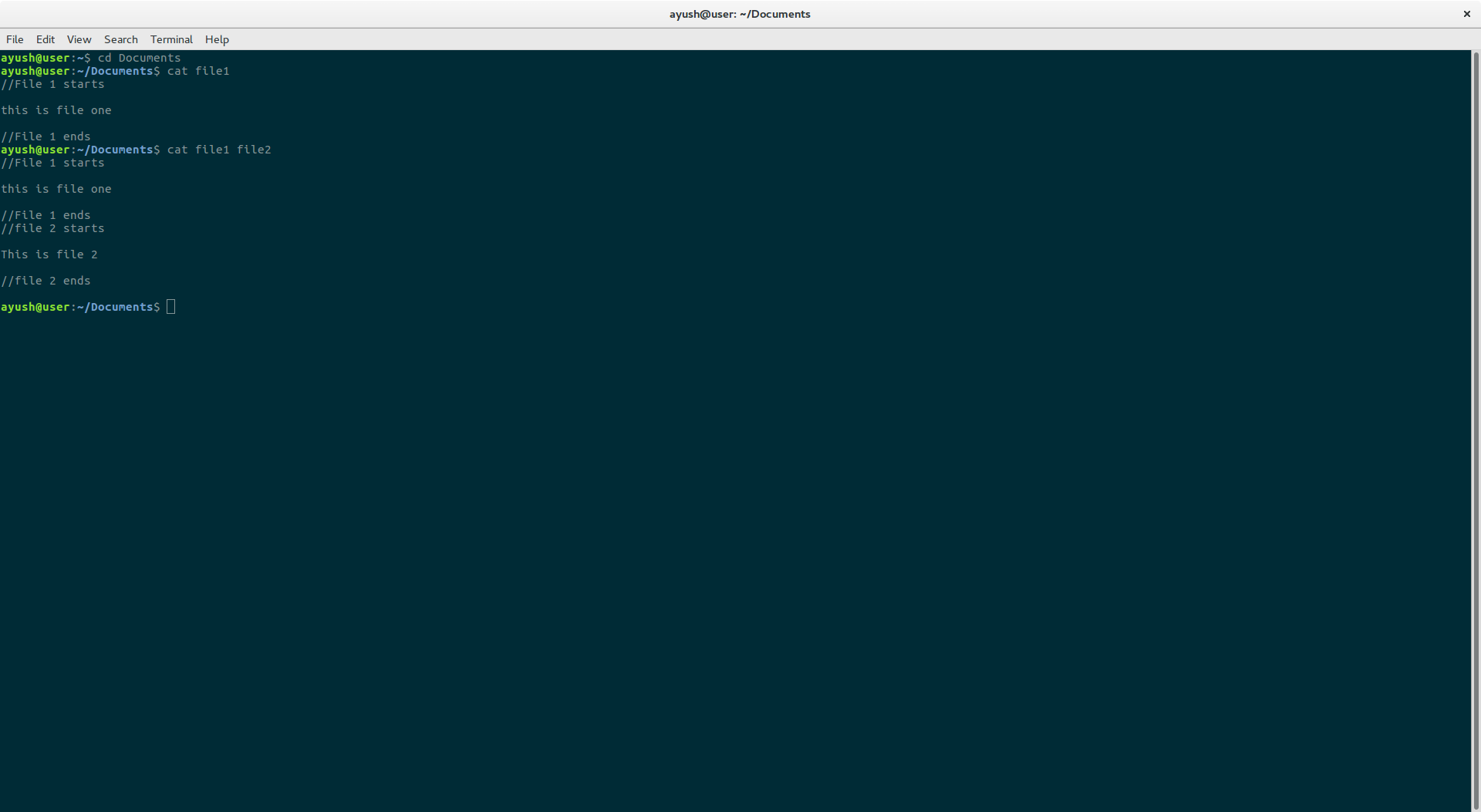
use system call to use cat in c

**Finishing up**

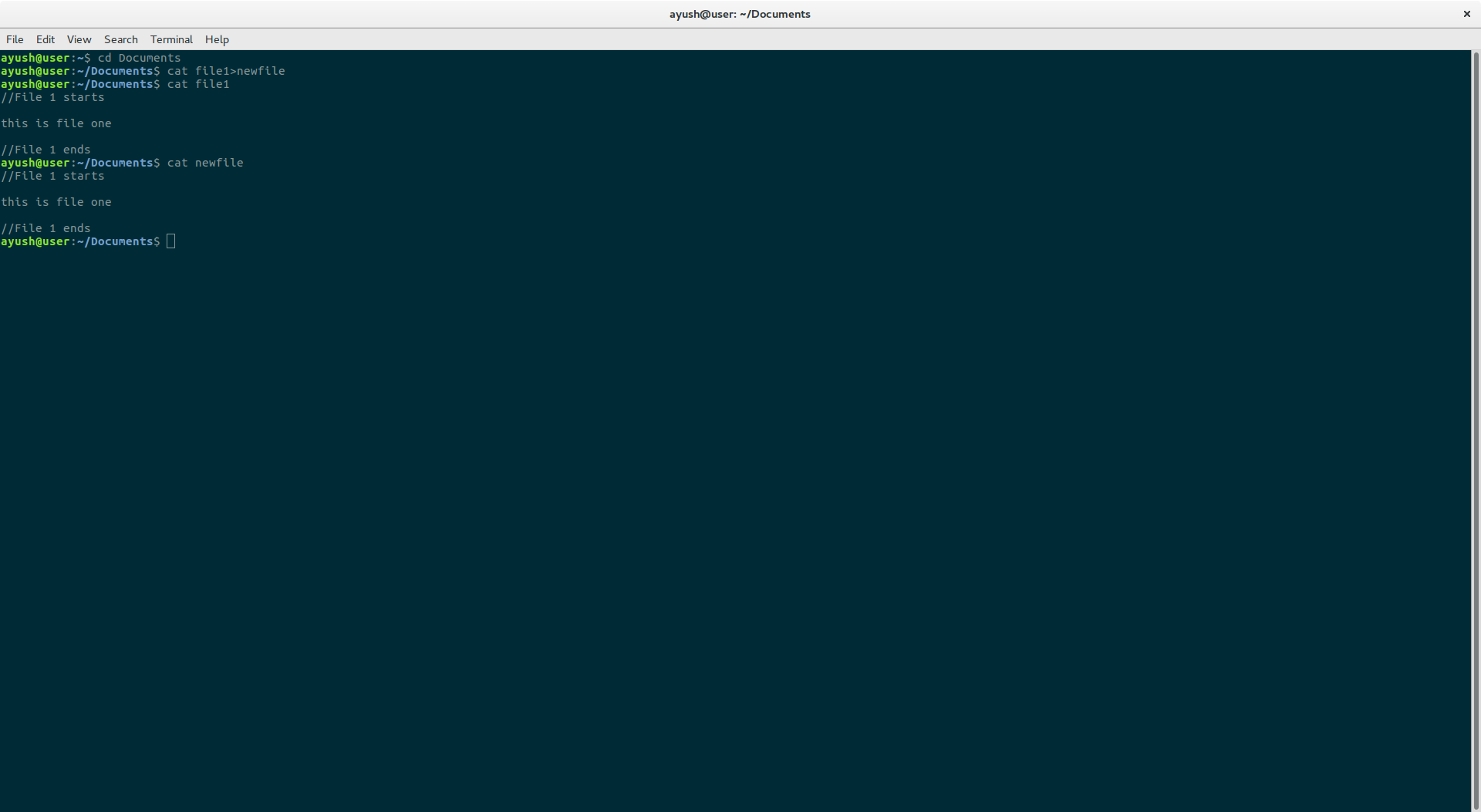
Learnt the function of cat and make it using c

$ cat file1

$ cat file1 file2

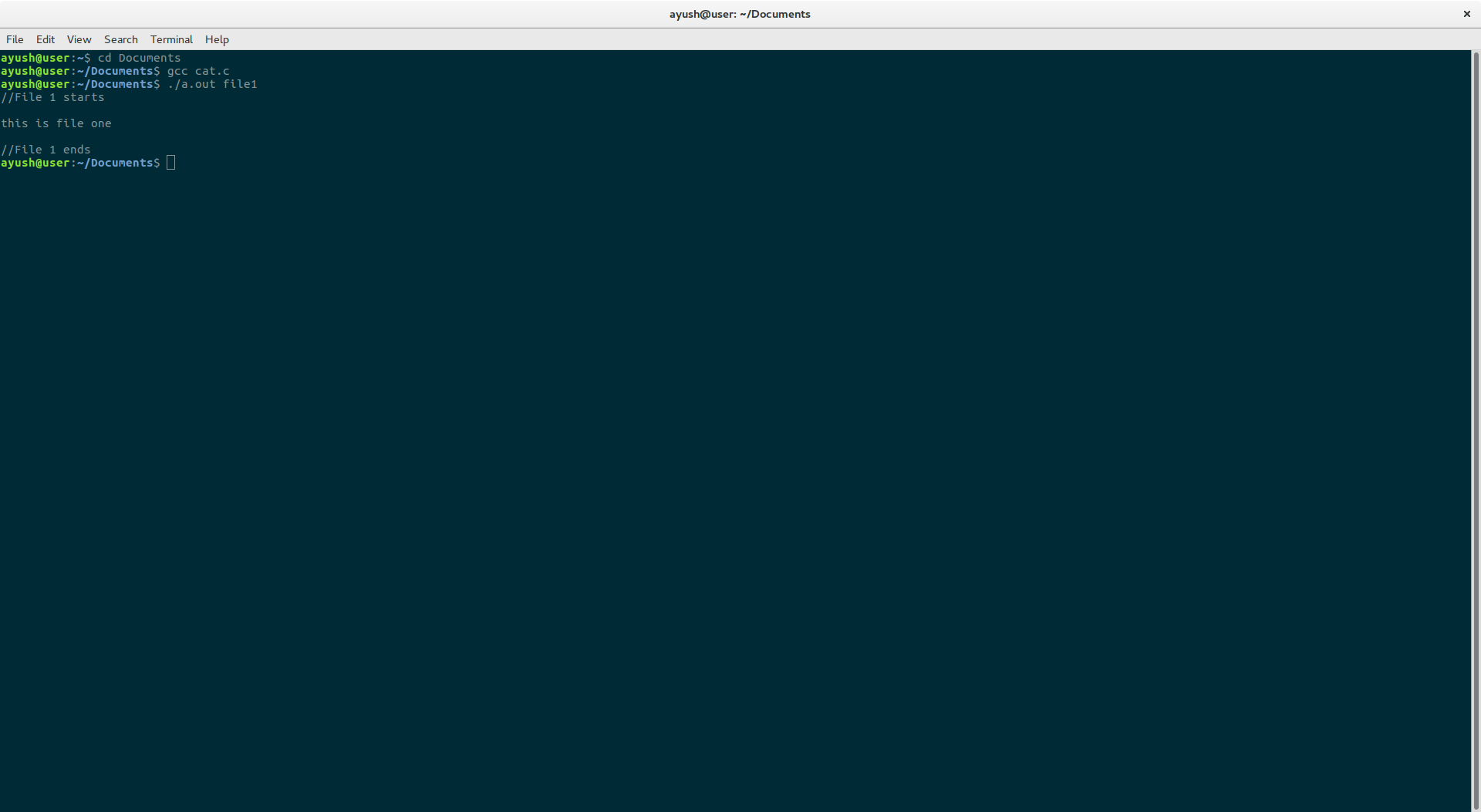


$ cat file1>newfile



Creating our own cat using system calls in c

$ ./a.out file1



Ex: No 4: Write a program to put a process to sleep and then wake it up and then kill it when  
completed

Aim:

In this lab exercise, to generate processes and work on interrupts and signals between different processes. The following interrupts are going to be worked on:  
• Ignore the signal  
• Default action will be taken  
• Run a user defined function

Procedure:

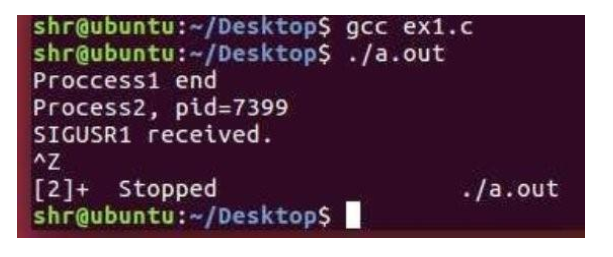
* Modify sample 3 such that signal can be sent from child process to parent  
  process.
* Signal number for sleep and wake signal. First process will send signals to the  
  second process and the second has to respond correspondingly.

Sample Code:

To get the signal handler function registered to the kernel, the signal handler function  
pointer is passed as second argument to the ‘signal’ function. The prototype of the  
signal function is:  
void (\*signal(int signo, void (\*func )(int)))(int); C Code:  
#include <stdio.h>  
#include <signal.h> #include<stdlib.h> void sighup(); /\*  
routines child will call upon sigtrap \*/ void sigint(); void  
sigquit(); main()  
{ int pid;  
/\* get child process \*/ if  
((pid = fork()) < 0)  
{  
perror("fork"); exit(1)

} if (pid == 0) { /\* child \*/ signal(SIGHUP,sighup); /\*  
set function calls \*/ signal(SIGINT,sigint);  
signal(SIGQUIT, sigquit); for(;;); /\* loop for ever \*/  
}  
else /\* parent \*/ { /\* pid hold  
id of child \*/  
printf("\nPARENT: sending  
SIGHUP\n\n");  
kill(pid,SIGHUP); sleep(3); /\*  
pause for 3 secs \*/  
printf("\nPARENT: sending  
SIGINT\n\n"); kill(pid,SIGINT);  
sleep(3); /\* pause for 3 secs  
\*/ printf("\nPARENT: sending  
SIGQUIT\n\n");  
kill(pid,SIGQUIT); sleep(3);  
}  
} void sighup()  
{  
signal(SIGHUP,sighup); /\* reset signal \*/ printf("CHILD: I have  
received a SIGHUP\n");  
} void sigint()  
{  
signal(SIGINT,sigint); /\* reset signal \*/ printf("CHILD: I have  
received a SIGINT\n");

Output:



Ex:No:5 Compare the task creation times. Execute a process and kernel thread, determine  
the time taken to create and run the threads.

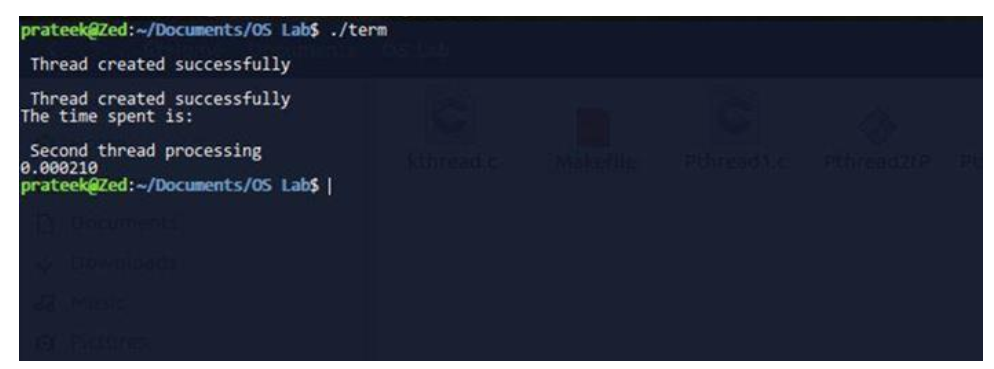
Aim:

* create kernel threads and user threads,
* link them with the kernel and then
* find the latency time between the user thread and the kernel thread.

Sample Code:

**USER THREAD**#include <stdio.h>  
#include <string.h>  
#include <sys/types.h>  
#include <time.h>  
#include <unistd.h>  
#define MAX 2000000  
#define BUF\_SIZE 100  
int main(void) {  
clock\_t begin = clock();  
pid\_t processId;  
int  
i;  
char buf[BUF\_SIZE];  
fork();  
processId = getpid();  
for (i = 1; i <= MAX; i++) {  
}  
clock\_t end = clock();  
double time\_spent = ((double)(end - begin) /  
CLOCKS\_PER\_SEC); printf("The time spent is: ");  
printf("%f\n", time\_spent); return 0;  
}  
**PROCESS THREAD**#include <stdio.h>  
#include <string.h>  
#include <pthread.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <time.h>  
pthread\_t tid[2];  
void\*  
doSomeThing(void  
\*arg){ unsigned long i  
= 0; pthread\_t id =  
pthread\_self();  
if(pthread\_equal(id,tid[0])) {  
printf("\n First thread  
processing\n");  
}  
else{  
printf("\n Second thread processing\n");  
}  
for(i=0; i< 100;i++){  
}  
return NULL;  
}  
int main(void){  
clock\_t begin =  
clock(); int i = 0;  
int err; while(i <  
2){  
err = pthread\_create(&(tid[i]), NULL,  
&doSomeThing, NULL); if (err != 0)  
printf("\ncan't create thread :[%s]",  
strerror(err)); else  
printf("\n Thread created  
successfully\n"); i++;  
}  
clock\_t end = clock(); printf("The time spent is: \n");  
double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;  
printf("%f\n",time\_spent);  
return 0;  
}  
**KERNEL THREAD**MakeFile  
obj-m += hello.o  
all  
:  
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  
clean:  
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

Output:

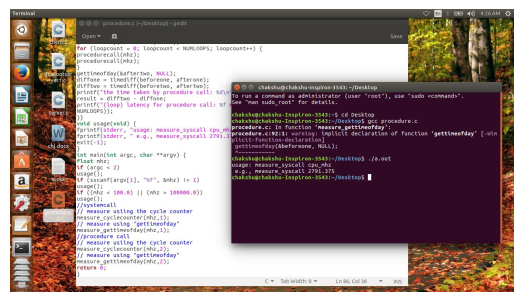


Ex:No:6 Compare the overhead of a system call with a procedure call. Analyse the cost of a  
minimal system call

Aim:

This experiment shows the latency difference between execution of a system call and a  
procedure call.  
There are two methods to find the latency of call (System or Procedure) :-  
➢ first using the access\_counter function which uses the system counter to get time  
difference and  
➢ secondly using get\_time\_of\_day function to get time difference  
Sample Code  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/uio.h>  
#include <unistd.h>  
#include <stdlib.h>  
#include <sys/syscall.h>  
// only works on pentium+ x86  
// access the pentium cycle counter  
// this routine lifted from somewhere on the Web...  
void access\_counter(unsigned int \*hi, unsigned int \*lo) {  
asm("rdtsc; movl %%edx,%0; movl %%eax,%1" /\* Read cycle counter \*/  
: "=r" (\*hi), "=r" (\*lo) /\* and move results to \*/  
: /\* No input \*/ /\* the two outputs \*/  
: "%edx", "%eax");  
}  
// here's the system call we'll use  
void procedurecall(float mhz)  
{  
int i,j,k;  
int count=0;  
for(i=0;i<25;i++)  
count++;  
}  
#define DO\_SYSCALL syscall(SYS\_getpid)  
// calculate difference (in microseconds) between two struct timevals  
// assumes difference is less than 2^32 seconds, and unsigned int is 32 bits  
unsigned int timediff(struct timeval before, struct timeval after) {  
unsigned int diff;  
diff = after.tv\_sec - before.tv\_sec;  
diff \*= 1000000;  
diff += (after.tv\_usec - before.tv\_usec);  
return diff;  
}  
// measure the system call using the cycle counter. measures the  
// difference in time between doing two system calls and doing  
// one system call, to try to factor out any measurement overhead  
void measure\_cyclecounter(float mhz,int calltype) {  
unsigned int high\_s, low\_s, high\_e, low\_e;  
size\_t nbytes;  
float latency\_with\_read, latency\_no\_read;  
if(calltype==1)  
{  
// warm up all the caches by exercising the functions  
access\_counter(&high\_s, &low\_s);  
// read(5, buf, 4);  
DO\_SYSCALL;  
access\_counter(&high\_e, &low\_e);  
// now do it for real  
access\_counter(&high\_s, &low\_s);  
DO\_SYSCALL;  
access\_counter(&high\_e, &low\_e);  
latency\_with\_read = ((float) (low\_e - low\_s) / mhz);  
access\_counter(&high\_s, &low\_s);  
access\_counter(&high\_e, &low\_e);  
latency\_no\_read = ((float) (low\_e - low\_s) / mhz);  
// print out the results  
printf("(cyclecounter) latency of systemcall: %f microseconds\n", latency\_with\_read -  
latency\_no\_read);  
}  
else  
{  
// warm up all the caches by exercising the functions  
access\_counter(&high\_s, &low\_s);  
// read(5, buf, 4);  
procedurecall(mhz);  
access\_counter(&high\_e, &low\_e);  
// now do it for real  
access\_counter(&high\_s, &low\_s);  
procedurecall(mhz);  
access\_counter(&high\_e, &low\_e);  
latency\_with\_read = ((float) (low\_e - low\_s) / mhz);  
access\_counter(&high\_s, &low\_s);  
access\_counter(&high\_e, &low\_e);  
latency\_no\_read = ((float) (low\_e - low\_s) / mhz);  
// print out the results  
printf("(cyclecounter) latency of procedurecall: %f microseconds\n", latency\_with\_read  
- latency\_no\_read);  
} }  
// measure the system call using the cycle counter. measures the  
// difference in time between doing two\*NLOOPS system calls and doing  
// one\*NLOOPS system calls, to try to factor out any measurement overhead  
#define NUMLOOPS 10000  
void measure\_gettimeofday(float mhz,int calltype) {  
struct timeval beforeone, afterone;  
struct timeval beforetwo, aftertwo;  
int loopcount;  
unsigned int diffone, difftwo, result;  
if(calltype==1)  
{  
// warm up all caches  
gettimeofday(&beforeone, NULL);  
gettimeofday(&beforetwo, NULL);  
for (loopcount = 0; loopcount < NUMLOOPS; loopcount++) {  
DO\_SYSCALL;  
}  
gettimeofday(&afterone, NULL);  
gettimeofday(&aftertwo, NULL);  
// measure loop of one syscall  
gettimeofday(&beforeone, NULL);  
for (loopcount = 0; loopcount < NUMLOOPS; loopcount++) {  
DO\_SYSCALL;  
}  
gettimeofday(&afterone, NULL);  
// measure loop of two syscalls  
gettimeofday(&beforetwo, NULL);  
for (loopcount = 0; loopcount < NUMLOOPS; loopcount++) {  
DO\_SYSCALL;  
DO\_SYSCALL;  
}  
gettimeofday(&aftertwo, NULL);  
diffone = timediff(beforeone, afterone);  
difftwo = timediff(beforetwo, aftertwo);  
printf("the time taken by systemcall: %d\n",diffone);  
result = difftwo - diffone;  
printf("(loop) latency for systemcall: %f microseconds\n", ((float) result) / ((float)  
NUMLOOPS));  
}  
else  
{  
// warm up all caches  
gettimeofday(&beforeone, NULL);  
gettimeofday(&beforetwo, NULL);  
for (loopcount = 0; loopcount < NUMLOOPS; loopcount++) {  
procedurecall(mhz);  
}  
gettimeofday(&afterone, NULL);  
gettimeofday(&aftertwo, NULL);  
// measure loop of one syscall  
gettimeofday(&beforeone, NULL);  
for (loopcount = 0; loopcount < NUMLOOPS; loopcount++) {  
procedurecall(mhz);  
}  
gettimeofday(&afterone, NULL);

Output:



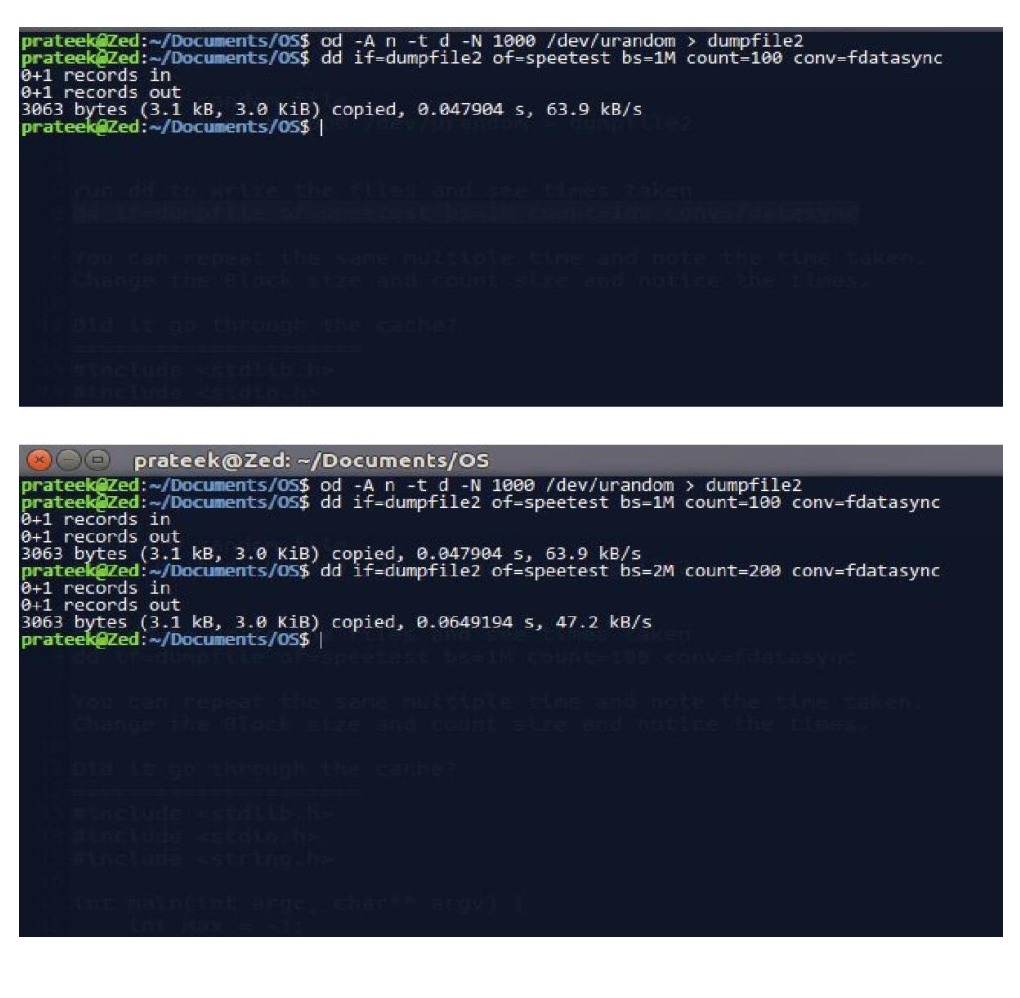
Ex:No:7 Determine the file read time for sequential and random access based of varying  
sizes of the files. Take care not to read from cached data - used the raw device  
interface. Draw a graph log/log plot of size of file vs average per-block time.

**INTRODUCTION**It is about the sequential and random file access in an operating system, and to simulate  
a sequential memory access and a random memory access (via the malloc() function in  
C++).  
❖ **We will deal with :-**create a random file  
od -A n -t d -N 1000 /dev/urandom >  
dumpfile2 run dd to write the files and see  
times taken  
dd if=dumpfile of=speetest bs=1M count=100 conv=fdatasync  
We can repeat the same multiple times and note the time taken.  
Change the Block size and count size and notice the time.  
Check if it went through the cache.  
**SOUCE CODE**#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
int main(int argc, char\*\* argv)  
{  
int max = -1; int mb  
= 0; char\* buffer;  
if(argc > 1) max =  
atoi(argv[1]);  
while((buffer=malloc(1024\*1024)) != NULL && mb != max)  
{  
memset(buffer, 0, 1024\*1024);  
mb++;  
printf("Allocated %d MB\n", mb);  
}  
return 0;  
}  
$ sudo swapoff -a  
$ free -m  
Find the effect of disk cache on swapping.  
$ free -m  
./a.out 500  
$ free -m  
Clearing the disk cache.  
$ free -m  
$ echo 3 | sudo tee /proc/sys/vm/drop\_caches  
$ free -m  
Effects of disk cache on load times  
$ cat hello.py print "Hello  
World! Love, Python"  
$ echo 3 | sudo tee /proc/sys/vm/drop\_caches  
$ time python hello.py  
$ echo 3 | sudo tee /proc/sys/vm/drop\_caches  
$ free -m  
$ dd if=/dev/zero of=bigfile bs=1M count=200  
$ ls -lh bigfile  
$ free -m  
$ time cat bigfile > /dev/null  
$ echo 3 | sudo tee /proc/sys/vm/drop\_caches  
$ time cat bigfile > /dev/null

**Sample Source Code (For Sequential File Access):**

#include <iostream>  
#include <fstream>  
#include <time.h>  
#include <stdlib.h>  
#include <exception>  
using namespace std;  
void readseq(){ //  
sequential read  
ifstream fin;  
fin.open("dumpfil  
e"); char temp;  
while(  
fin.get(temp))  
cout << temp;  
cout << endl;  
fin.close();  
}  
void randomseek(){  
// random seek using rand.  
/\* an array is created that stores all the used positions every  
newly generated random number is checked against the array.  
If number already exists in array, then generate new random  
number. If number is found that is not in array, then push it into  
the array and read a character \*/  
srand((double)(clock())); // seed rand with current  
time ifstream fin; fin.open("dumpfile"); fin.seekg(0,  
ios::end); int size = fin.tellg(); fin.clear(); int \*a;  
cout<<size<<endl ;

try{ a = new  
int[size];  
}  
catch(bad\_alloc& ba){  
cerr << "Bad alloc caught: " << ba.what() << endl;  
}  
char temp; bool used = false, found =  
false; int pos = 0, n = 0; while(n < size-  
1){ found = false; do{ used = false;  
pos = rand()%(size-1); for (int i = 0; i <  
n; ++i){ if(a[i] == pos){ used = true;  
break;  
}  
}  
if( !used ){  
found = true;  
a[n] = pos;  
n++;  
}  
}  
while( !found ) fin.seekg(a[n-  
1], ios::beg);



Ex: No: 8 Determine the latency of individual integer access times in main memory, L1  
Cache and L2 Cache. Plot the results in log of memory accessed vs average latency

Procedure:

1. Download Lmbench from http://www.bitmover.com/lmbench/get\_lmbench.html.  
2. Extract the .taz.gz file. ($ tar -xzvf lmbench3.tar.gz)  
3. Open lmbench3 folder. ($ cd lmbench3)  
4. Run Makefile ($ make)  
you may get error. To overcome the errors,  
5. Create a folder SCCS ($ mkdir SCCS)  
6. Update the access date or modification date of the file. ($ touch SCCS/s.ChangeSet)  
To assign particular CPU cores to a program or process, you can use taskset, a command  
line tool for retrieving or setting a process' CPU affinity on Linux.  
To measure the latency,  
7. Open lmbench3/bin/your-current-os- (cd bin/x86\_64-linux-gnu)  
8. Type taskset command at prompt. ($ taskset -c 0/1 ./lat\_mem\_rd 256 100)

Output:

