**Practical-1**

**AIM:-** **Implement copy command using Open, Create, Read, Write, Access**

**and Close system call.**

**Theory:**

Basically there are total 5 types of I/O system calls:

1. **Create:** Used to Create a new empty file.

Syntax in C language:

int creat(char \*filename, mode\_t mode)

**Parameter :**

* + **filename :** name of the file which you want to create
  + **mode :** indicates permissions of new file.

**Returns :**

* + return first unused file descriptor (generally 3 when first creat use in process beacuse 0, 1, 2 fd are reserved)
  + return -1 when error

**How it work in OS**

* + Create new empty file on disk
  + Create file table entry
  + Set first unused file descriptor to point to file table entry
  + Return file descriptor used, -1 upon failure

1. **open**: Used to Open the file for reading, writing or both.

**How it works in OS**

* Find existing file on disk
* Create file table entry
* Set first unused file descriptor to point to file table entry
* Return file descriptor used, -1 upon failure

int open (const char\* Path, int flags [, int mode ]);

1. **read:** From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

Syntax in C language

Size\_t read (int fd, void\* buf, size\_t cnt);

**Parameters**

* **fd:** file descripter
* **buf:** buffer to read data from
* **cnt:** length of buffer

**Returns: How many bytes were actually read**

* return Number of bytes read on success
* return 0 on reaching end of file
* return -1 on error
* return -1 on signal interrupt
* **buf** needs to point to a valid memory location with length not smaller than the specified size because of overflow.
* **fd** should be a valid file descriptor returned from open() to perform read operation because if fd is NULL then read should generate error.
* **cnt** is the requested number of bytes read, while the return value is the actual number of bytes read. Also, some times read system call should read less bytes than cnt.

1. **write:** Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT\_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

#include <fcntl.h>

size\_t write (int fd, void\* buf, size\_t cnt);

**Parameters**

* **fd:** file descripter
* **buf:** buffer to write data to
* **cnt:** length of buffer

**Returns: How many bytes were actually written**

* return Number of bytes written on success
* return 0 on reaching end of file
* return -1 on error
* return -1 on signal interrupt

Code:

#include<stdio.h>

#include<string.h>

#include<unistd.h>

#include<fcntl.h>

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

{

int i, fd[2],sz;

char c[100] ;

fd[0] = open(argv[1], O\_RDONLY, 0);

fd[1] = open(argv[2], O\_WRONLY|O\_CREAT , 0);

while((i = read(fd[0],c,100)>0))

{

write(fd[1], c, i);

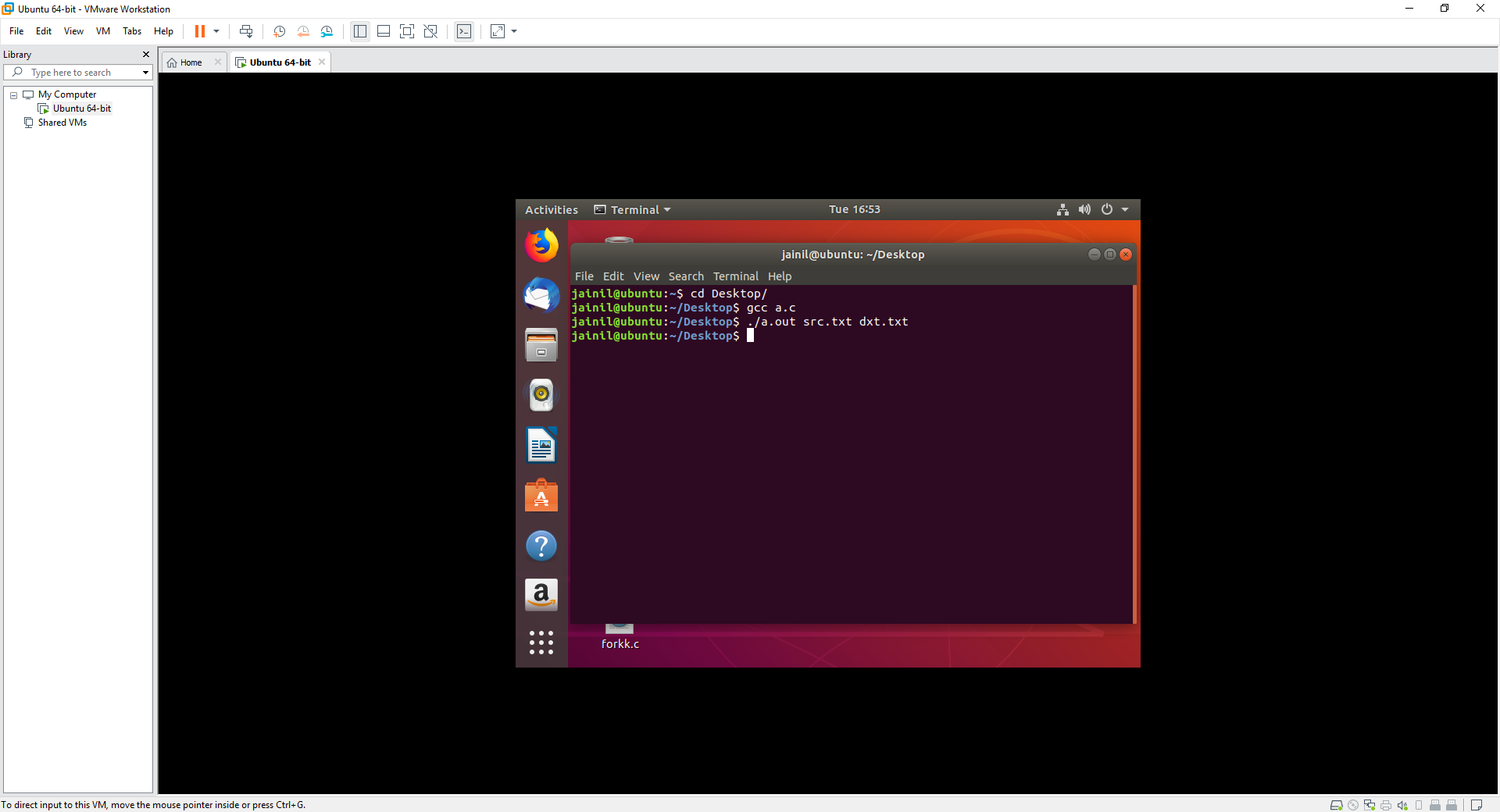
}

close(fd[0]);

close(fd[1]);

}

Output:



**Conclusion :-**

We studied the system call in linux. Open(),close(),read(),write() are few system calls. O\_RDONLY, O\_WRONLY, O\_CREAT etc are argument passed to these system calls. We have copied the file to destination file as well as created destination file if not exists.

**Practical-2**

**AIM:-** **Write a program that creates a file with a hole in it using lseek().**

**Theory:**

write it in another file

From a given file (e.g. input.txt) read the alternate nth byte and write it on another file with the help of “lseek”.

lseek (C System Call): lseek is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.  
Function Definition:

off\_t lseek(int fildes, off\_t offset, int whence);

**Field Description**  
int fildes : The file descriptor of the pointer that is going to be moved  
off\_t offset : The offset of the pointer (measured in bytes).  
int whence : The method in which the offset is to be interpreted  
(rela, absolute, etc.). Legal value r this variable are provided at the end.  
return value : Returns the offset of the pointer (in bytes) from the  
beginning of the file. If the return value is -1,  
then there was an error moving the pointer.

**Code:**

#include <errno.h>

#include <unistd.h>

#include <sys/stat.h>

#include <string.h>

#include<fcntl.h>

#include<stdio.h>

int main(void)

{

int fd;

char name[20] = "jainil patel";

fd = open( "book.txt", O\_RDWR | O\_CREAT , S\_IWRITE | S\_IREAD );

lseek(fd, 100, SEEK\_SET);

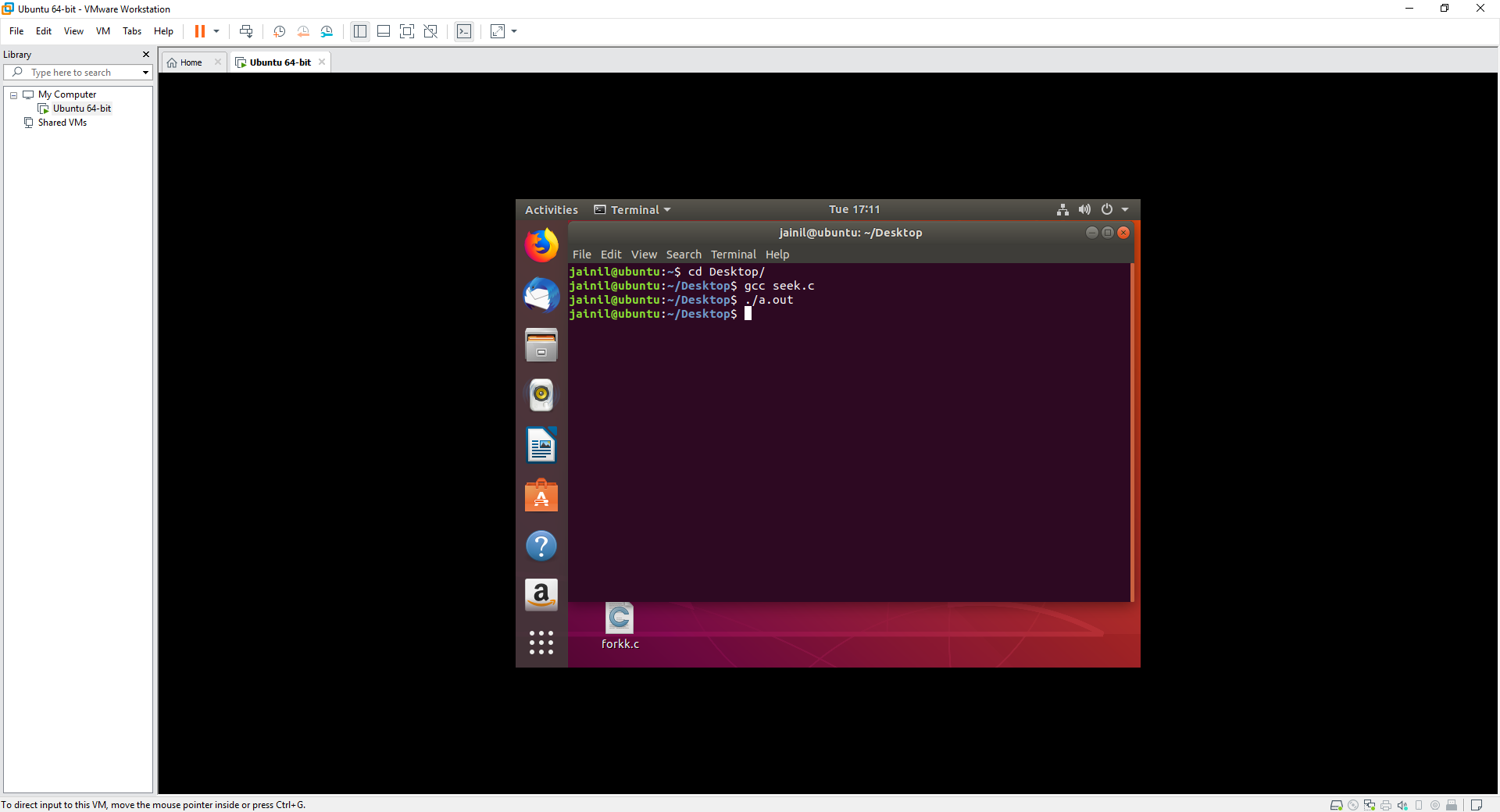
write(fd, name, sizeof(char)\*strlen(name));

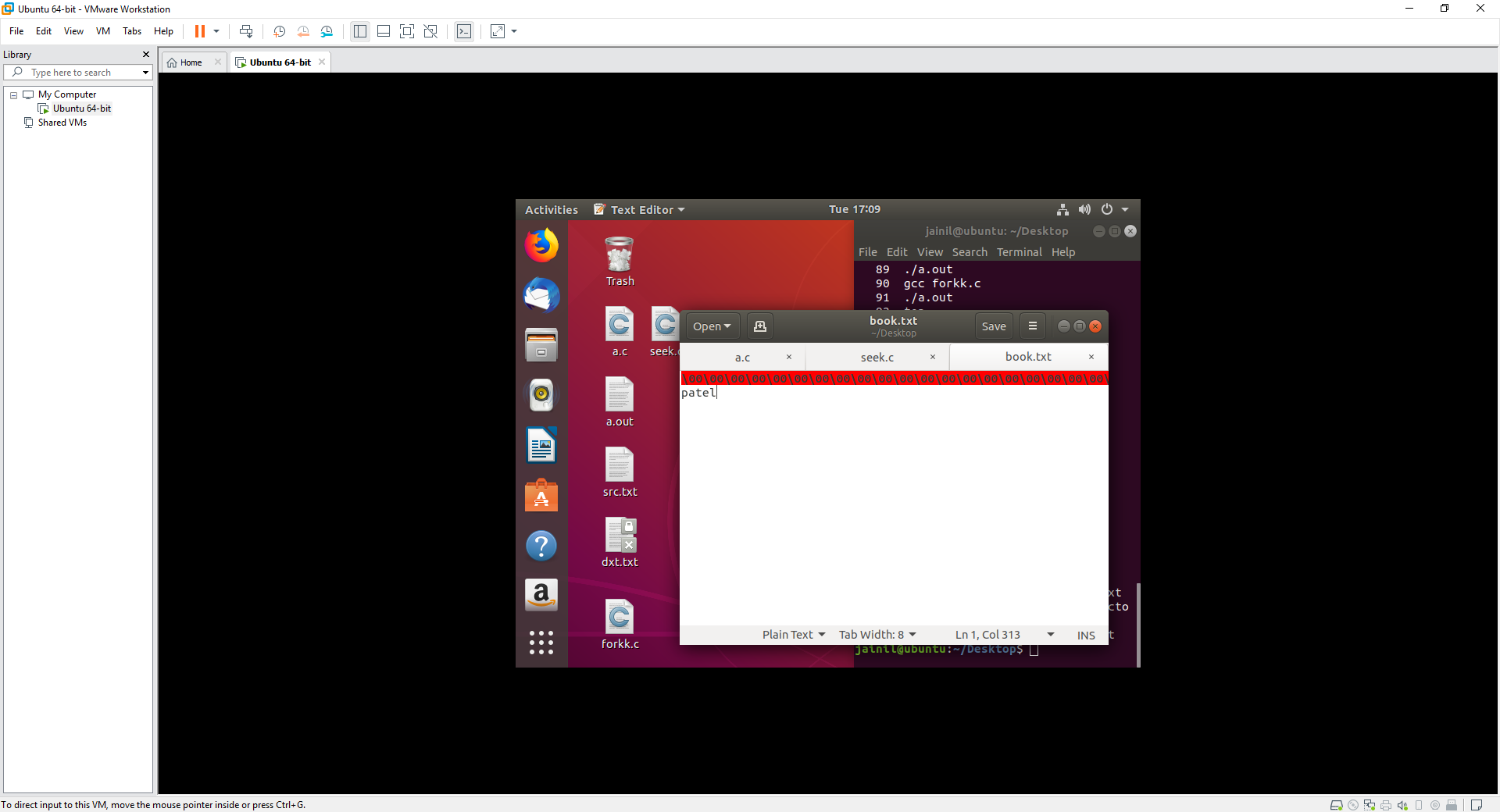
close(fd);

return 0;

}

**Output:**





**Conclusion :-**

We studied lseek() system call to seek in files and learned to create holes in the file.holes are represented by /0 or null in linux file system.

**Practical-3**

**AIM:-** **You need to create user defined function to create processes and to join**

**those processes using fork, wait system call.**

**Write a program to demonstrate file sharing among child and parent.**

**Theory:**

**fork() in C**

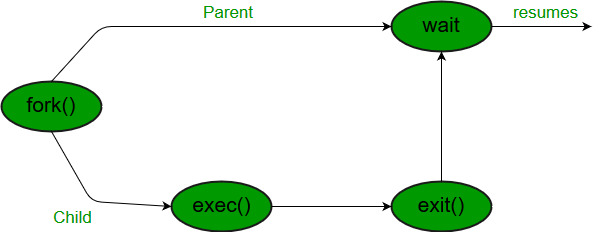
Fork system call use for creates a new process, which is called child process, which runs concurrently with process (which process called system call fork) and this process is called parent process. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

**Wait System Call in C**

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction.  
Child process may terminate due to any of these:

* It calls exit();
* It returns (an int) from main
* It receives a signal (from the OS or another process) whose default action is to terminate.



**Code:**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <errno.h>

#include <sys/wait.h>

pid\_t a()

{

int i=0;

for(i=0;i<5;i++){

pid\_t a=fork();

if(a!=0){

printf("hello from pid %d \n",a);

printf("pid %d is returning\n",a);

return a;

}

}

printf("main call is returning ");

return 0;

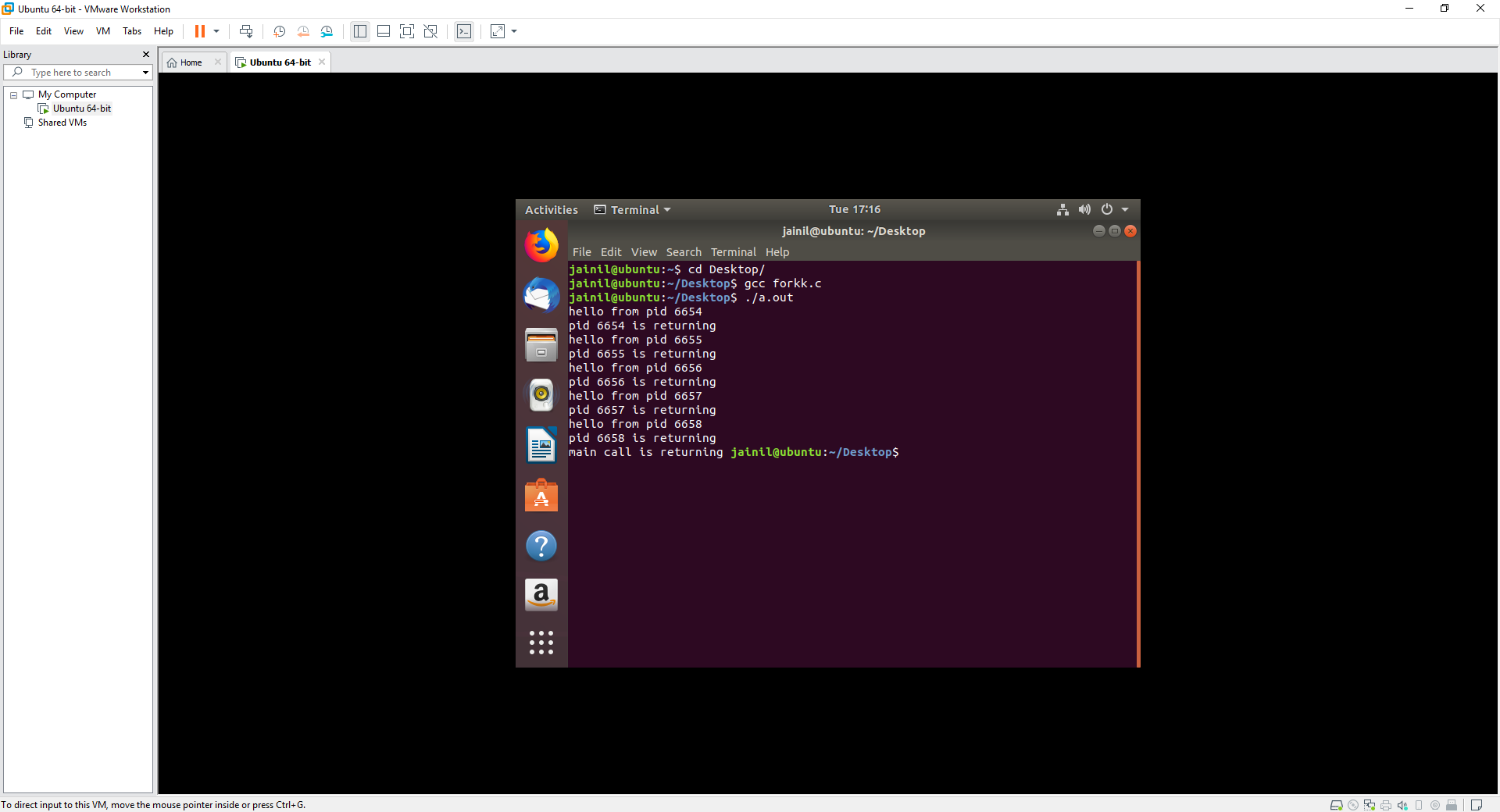
}

int main()

{a(); wait(NULL);

return 0;}

**Output:**



**Conclusion :-**

We seen fork system call to make child processes having same code of parent process and used wait system call for synchronizing the child process and parent process. Fork system call use for creates a new process, which is called child process, which runs concurrently with process (which process called system call fork) and this process is called parent process. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.  
Zero: Returned to the newly created child process.  
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

**Practical-4**

**AIM:-** **Write a program to implement Zombie process and Orphan process**

**Theory:**

What Is a Zombie Process?

A zombie process is a process that has completed but still has an entry in the process table. The process table in

an Operating System records process information such as ID, parent, status, etc. A child process is that which is

created by a higher order process (its parent). Each process might create many children, but each child has only

one parent. If a process doesn’t have a parent, that usually means that this process was created by the kernel.

When a child process is terminated, the kernel keeps some information about it in the process table (including

its exit status). The parent needs to read the exit status of the child before it removes the child’s entry from the

table. A child process must always become a zombie until its status is collected by its parent.

Orphan Process

An orphan process is a computer process whose parent process has finished or terminated, though it remains

running itself.

In a Unix-like operating system any orphaned process will be immediately adopted by the special init system

process. This operation is called re-parenting and occurs automatically. Even though technically the process has

the init process as its parent, it is still called an orphan process since the process that originally created it no

longer exists.

A process can be orphaned unintentionally, such as when the parent process terminates or crashes. The process

group mechanism in most Unix-like operation systems can be used to help protect against accidental orphaning,

where in coordination with the user’s shell will try to terminate all the child processes with the SIGHUP process

signal, rather than letting them continue to run as orphans.

A process may also be intentionally orphaned so that it becomes detached from the user’s session and left

running in the background; usually to allow a long-running job to complete without further user attention, or to

start an indefinitely running service. Under Unix, the latter kinds of processes are typically called daemon

processes. The Unix nohup command is one means to accomplish this.

**Code:**

ZOMBIE PROCESS

// A C program to demonstrate Zombie Process.

// Child becomes Zombie as parent is sleeping

// when child process exits.

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

pid\_t child\_pid = fork();

// Parent process

if (child\_pid > 0)

sleep(50);

// Child process

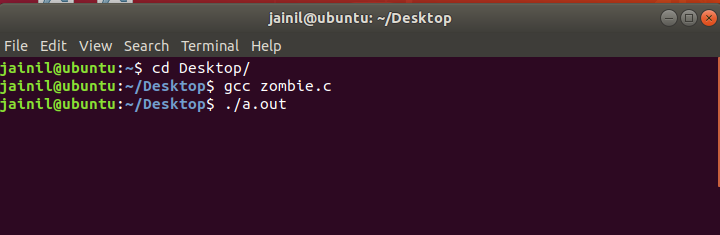
else

exit(0);

return 0;

}

**Output:**



**Code: orphan**

#include<stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

// Create a child process

int pid = fork();

if (pid > 0)

printf("in parent process");

// Note that pid is 0 in child process

// and negative if fork() fails

else if (pid == 0)

{

sleep(30);

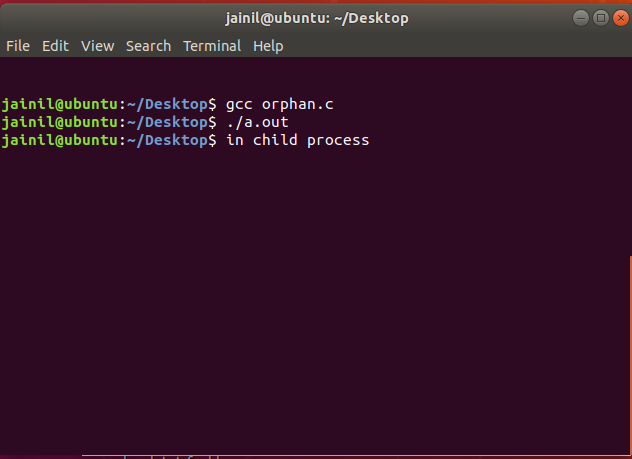
printf("in child process");

}

return 0;

}

**Output:**



**Practical-5**

**AIM:-** **:** **Implement below system calls: (a) grep (b) ls (c) cp (d) head**

**Theory:**

**GREP command:**

The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out).

Syntax: grep [options] pattern [files]

Options Description:

-c : This prints only a count of the lines that match a pattern

-h : Display the matched lines, but do not display the filenames.

-i : Ignores, case for matching

-l : Displays list of a filenames only.

-n : Display the matched lines and their line numbers.

-v : This prints out all the lines that do not matches the pattern

-e exp : Specifies expression with this option. Can use multiple times.

-f file : Takes patterns from file, one per line.

-E : Treats pattern as an extended regular expression (ERE)

-w : Match whole word

-o : Print only the matched parts of a matching line,

with each such part on a separate output line

**Code:**

#include<stdio.h>

#include<string.h>

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

{

FILE \*fp;

char line[100];

// initialsing the file pointer to read

fp = fopen(argv[2],"r");

while(fscanf(fp , "%[^\n]\n" , line)!=EOF)

{

if(strstr(line , argv[1]) !=NULL)

{

// print that line

printf("%s\n" , line);

}

else

{

continue;

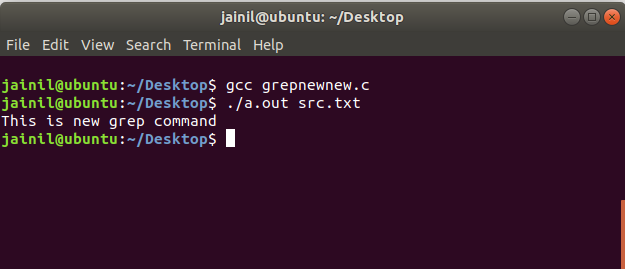
}

}

fclose(fp);

}

**Output:**



**LS Command:**

The ls command is a command-line utility for listing the contents of a directory or directories given to it via standard input. It writes results to standard output. The ls command supports showing a variety of information about files, sorting on a range of options and recursive listing.

How to show the contents of a directory:

To show the contents of a directory pass the directory name to the ls command. This will list the contents of the directory in alphabetical order. If your terminal supports colours you may see that file and directory listings are a different colour.

Syntax: ls [option ...] [file]...

**Code:**

#include<stdio.h>

#include<dirent.h>

#include<sys/types.h>

#include<fcntl.h>

int main(int argc,char \*\*argv)

{

struct dirent \*\*namelist;

int i,n;

if (argc == 2)

{

n = scandir(argv[1],&namelist, 0 ,alphasort);

if( n<0 )

perror("scandir");

else

{

for(i = 0;i < n; i++)

{

printf("%s \n", namelist[i]->d\_name);

free(namelist[i]);

}

}

}

else

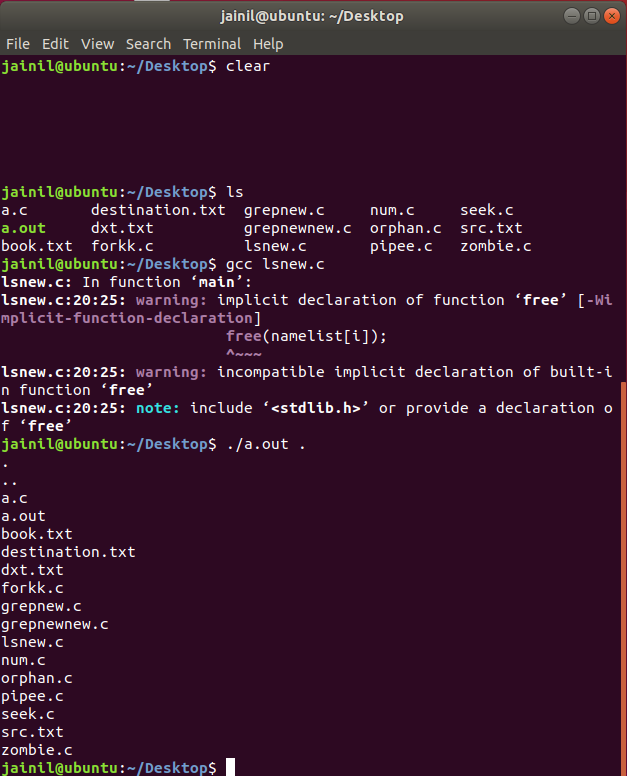
{

printf("enter directory name ");

}

}

**Output:**



**CP command:**

**cp** stands for **copy**. This command is used to copy files or group of files or directory. It creates an exact image of a file on a disk with different file name. cp command require at least two filenames in its arguments.

Syntax:

cp [OPTION] Source Destination

cp [OPTION] Source Directory

cp [OPTION] Source-1 Source-2 Source-3 Source-n Directory

First and second syntax is used to copy Source file to Destination file or Directory.

Third syntax is used to copy multiple Sources(files) to Directory.

**Code**

#include <stdio.h>

#include <stdlib.h>

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

{

char ch, \*source\_file, \*target\_file;

FILE \*source, \*target;

source\_file=argv[1];

source = fopen(source\_file, "r");

if (source == NULL)

{

printf("Press any key to exit...\n");

exit(EXIT\_FAILURE);

}

target\_file=argv[2];

target = fopen(target\_file, "w");

if (target == NULL)

{

fclose(source);

printf("Press any key to exit...\n");

exit(EXIT\_FAILURE);

}

while ((ch = fgetc(source)) != EOF)

fputc(ch, target);

printf("File copied successfully.\n");

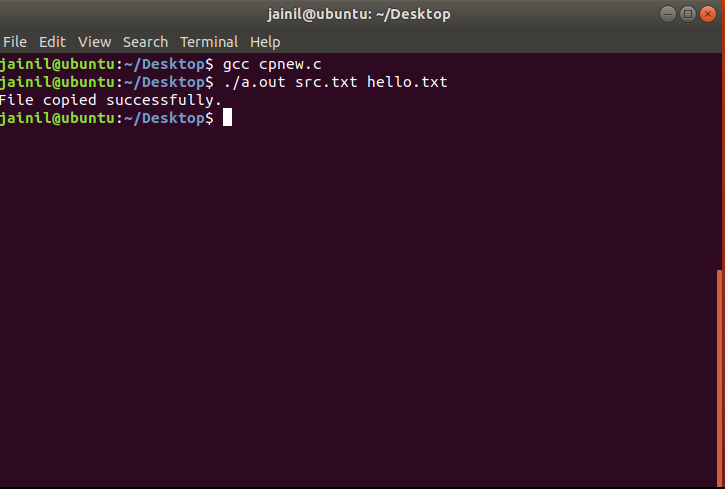
fclose(source);

fclose(target);

return 0;

}

**Output:**

****

**HEAD command:**

It is the complementary of [Tail](https://www.geeksforgeeks.org/tail-command-linux-examples/) command. The head command, as the name implies, print the top N number of data of the given input. By default it prints the first 10 lines of the specified files. If more than one file name is provided then data from each file is precedes by its file name.

Syntax: head [OPTION]... [FILE]...

**Code:**

#include <stdio.h>

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

{

FILE \* fp;

char \* line = NULL;

int len = 0;

int cnt = 0;

if( argc < 3)

{

printf("Insufficient Arguments!!!\n");

printf("Please use \"program-name file-name N\" format.\n");

return -1;

}

fp = fopen(argv[1],"r");

if( fp == NULL )

{

printf("\n%s file can not be opened !!!\n",argv[1]);

return 1;

}

while (getline(&line, &len, fp) != -1)

{

cnt++;

if ( cnt > atoi(argv[2]) )

break;

printf("%s",line);

fflush(stdout);

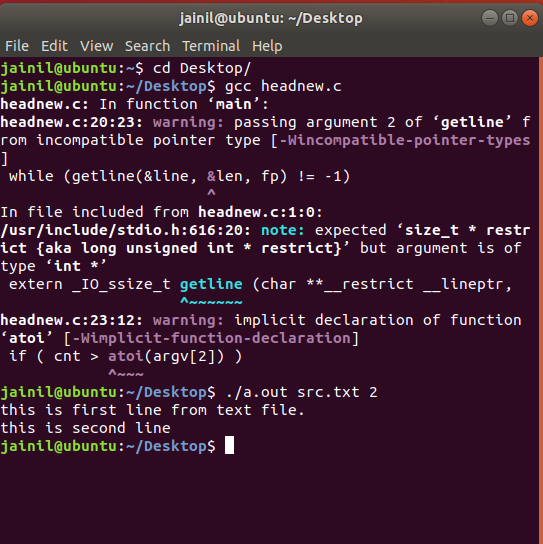
}

fclose(fp);

return 0;

}

**Output:**



**Conclusion:**

In this practical we learn how to implement ls, head, grep and cp command and saw how system executes this different commands.

**Practical-6**

**AIM:-** **:** **Write a program to perform input /output redirection from/to file using**

**dup().**

**Hardware Requirement:** A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory:**

**What is a dup() system call**

dup() system call in unix systems copies a file descriptor into the first free slot in the private file descriptor table and then returns the new file descriptor to the user. It works for all the file types. The syntax is :

newfd = dup(fd);

Here fd is the file descriptor being duped and newfd is returned to the user.

There are basically three different data structures that helps in manipulation of file system. These are - the inode table, private user file descriptor table and the global file table. Before moving forward to the description of dup() command, I urge you to please follow this article on Internal Data Structure for file handling in Unix kernel.

dup() system call doesnt create a separate entry in the global file table like the open() system call, instead it just increments the count field of the entry pointed to by the given input file descriptor in the global file table. Consider an example where fd 0, 1 and 2 are by default engaged to the standard input/output and error. Then if the user opens a file "/var/file1" (fd - 3), then he opens file "/var/file2" (fd - 4) and again he opened "/var/file1" (fd - 5). And now, if he does a dup(3), kernel would follow the pointer from the user file descriptive table for the fd entry '3', and increments the count value in the global file table. Then, it searches for the next avaialable free entry in file descriptor table and returns that value to the user (6 in this case).

**Input / output redirection using dup() system call**

dup() system call finds use in implementing input/output redirection or piping the output on unix shell. Suppose, we wish to redirect the output of 'ls' command to a file, we use the following command on shell to do our job:

root> ls /var/\* > tempfile

File descriptor 1 is bound to the standard output stream. The 'ls /var/\*' command is supposed to output the data on this output stream i.e. 1. But, using '>' operator we are able to redirect this output to file 'tempfile'. What happens when the process that is executing the shell here is that it parses the command and when it finds '>' operator, it will first find the file descriptor of the rhs operand - 'tempfile' OR create the new fd if file doesnt exist already. Once, it finds this fd, it will close the stdout file descriptor and call a dup() on the given fd for this 'tempfile'.

Thats it, from this step onwards, the output will be redirected to the file 'tempfile'. We can also do an additional step of closing the file descriptor to preserve the number of descriptors.

/\*redirection of I/O\*/

{

fd = creat('tempfile', flags); close(stdout); //stdout => 1 dup(fd);

close(fd);

/\* stdout is now redirected \*/

}

The same logic is applied when we apply "pipe" operations on the shell. Thus, although dup() is not an elegant command but yet it is a powerful building block for several higher level commands.

**Code:**

#include <stdio.h>

#include <stdlib.h>   
#include <unistd.h>

#include <errno.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <string.h>

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

{

int i,j;

char buf[512];

int fd1 = open("read.txt",O\_RDONLY); int fd2 = open("write.txt",O\_WRONLY);

int r = dup(fd1); int w = dup(fd2);

close(fd1);

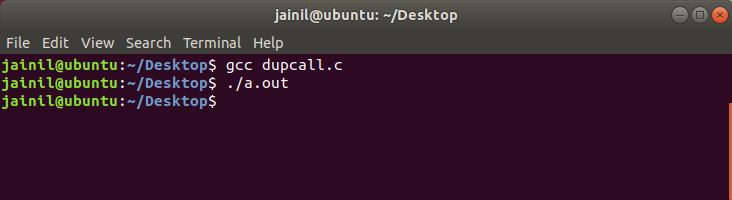
close(fd2);

read(r, buf, sizeof(buf)); write(w, buf, sizeof(buf));

return 0;

}

**Output:**

****

**Conclusion:**

From this practical we have learned dup system call copies a file descriptor into the first free slot in the private file descriptor table and then returns the new file descriptor to the user. We have implemented this logic in C program.

**Practical-7**

**AIM:-** **:** **Write a program to perform addition of 1 to 100. Inter Process**

**Communication using Shared Memory and Pipe. You need to use**

**shmget and shmat system call.**

**Hardware Requirement:** A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory:**

**IPC through Pipe**

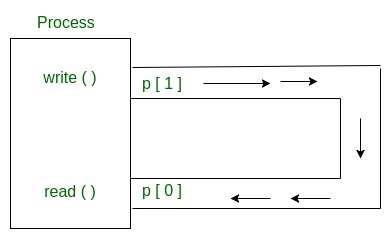
Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes(inter-process communication).

Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a “virtual file”.

The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this “virtual file” or pipe and another related process can read from it.

If a process tries to read before something is written to the pipe, the process is suspended until something is written.

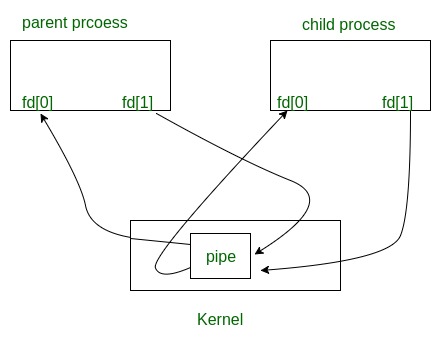
The pipe system call finds the first two available positions in the process’s open file table and allocates them for the read and write ends of the pipe.



Pipes behave FIFO(First in First out), Pipe behave like a queue data structure. Size of read and write don’t have to match here. We can write 512 bytes at a time but we can read only 1 byte at a time in a pipe.

**Parent and child sharing a pipe:**

When we use fork in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.



Code:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#define P1\_READ 0

#define P2\_WRITE 1

#define P2\_READ 2

#define P1\_WRITE 3

#define NUM\_PIPES 2

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

{

int fd[2\*NUM\_PIPES];

int val = 0, len, i;

pid\_t pid;

for (i=0; i<NUM\_PIPES; ++i)

{

if (pipe(fd+(i\*2)) < 0)

{

perror("Failed to allocate pipes");

exit(EXIT\_FAILURE);

}

}

if ((pid = fork()) < 0)

{

perror("Failed to fork process");

return EXIT\_FAILURE;

}

if (pid == 0)

{

close(fd[P1\_READ]);

close(fd[P1\_WRITE]);

pid = getpid();

len = read(fd[P2\_READ], &val, sizeof(val));

if (len < 0)

{

perror("Child: Failed to read data from pipe");

exit(EXIT\_FAILURE);

}

else if (len == 0)

{

fprintf(stderr, "Child: Read EOF from pipe");

}

else

{

printf("Child(%d): Received %d\n", pid, val);

val \*= 2;

printf("Child(%d): Sending %d back\n", pid, val);

if (write(fd[P2\_WRITE], &val, sizeof(val)) < 0)

{

perror("Child: Failed to write response value");

exit(EXIT\_FAILURE);

}

}

close(fd[P2\_READ]);

close(fd[P2\_WRITE]);

return EXIT\_SUCCESS;

}

close(fd[P2\_READ]);

close(fd[P2\_WRITE]);

pid = getpid();

val = 42;

printf("Parent(%d): Sending %d to child\n", pid, val);

if (write(fd[P1\_WRITE], &val, sizeof(val)) != sizeof(val))

{

perror("Parent: Failed to send value to child ");

exit(EXIT\_FAILURE);

}

len = read(fd[P1\_READ], &val, sizeof(val));

if (len < 0)

{

perror("Parent: failed to read value from pipe");

exit(EXIT\_FAILURE);

}

else if (len == 0)

{

fprintf(stderr, "Parent(%d): Read EOF from pipe", pid);

}

else

{

printf("Parent(%d): Received %d\n", pid, val);

}

close(fd[P1\_READ]);

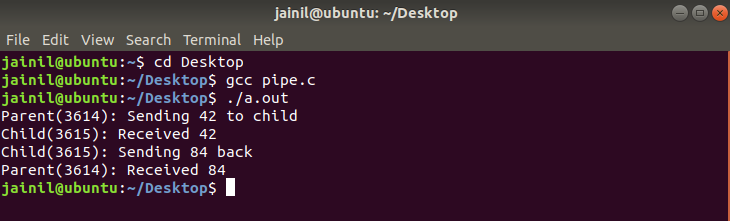
close(fd[P1\_WRITE]);

wait(NULL);

return EXIT\_SUCCESS;

}

Output:



**IPC through shared memory**

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by anther process.

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

* Server reads from the input file.
* The server writes this data in a message using either a pipe, fifo or message queue.
* The client reads the data from the IPC channel,again requiring the data to be copied from kernel’s IPC buffer to the client’s buffer.
* Finally the data is copied from the client’s buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED ARE:

**ftok():** is use to generate a unique key.

**shmget():** int shmget(key\_t,size\_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.

**shmat():** Before you can use a shared memory segment, you have to attach yourself

to it using shmat(). void \*shmat(int shmid ,void \*shmaddr ,int shmflg);

shmid is shared memory id. shmaddr specifies specific address to use but we should set

it to zero and OS will automatically choose the address.

**shmdt():** When you’re done with the shared memory segment, your program should

detach itself from it using shmdt(). int shmdt(void \*shmaddr);

**shmctl():** when you detach from shared memory,it is not destroyed. So, to destroy

shmctl() is used. shmctl(int shmid,IPC\_RMID,NULL);

Code:

#include <stdio.h>

#include <sys/types.h>

#include <sys/shm.h>

#include <unistd.h>

int main(int argc, char \*\*argv)

{

pid\_t child;

int shmid;

int\* shmptr;

shmid = shmget((key\_t) 1234, 3 \* sizeof(int), 0666 | IPC\_CREAT);

if (shmid == -1)

{

perror("shmget");

return -1;

}

shmptr = (int\*) shmat(shmid, NULL, 0);

if (shmptr == (void\*) -1)

{

perror("shmat");

return -1;

}

shmptr[2] = 0;

child = fork();

if (child == -1)

{

perror("fork");

return -1;

}

if (child > 0)

{

waitpid(child, NULL, 0);

int s = shmptr[0], e = shmptr[1], r = shmptr[2];

printf("Sum of numbers from %d to %d is %d.\n", s, e, r);

}

else if (child == 0)

{

int i;

scanf("%d %d", &shmptr[0], &shmptr[1]);

for (i = shmptr[0]; i <= shmptr[1]; i++)

shmptr[2] += i;

return 0;

}

if (shmdt(shmptr) == -1)

{

perror("shmdt");

return -1;

}

if (shmctl(shmid, IPC\_RMID, 0) == -1)

{

perror("shm remove");

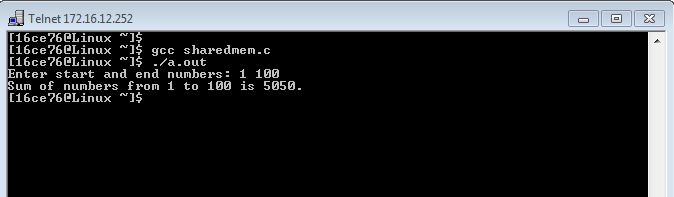
return -1;

}

return 0;

}

Output:



**Practical-8**

**AIM:-** **:** **Implement below file System calls:**

**iget, iput, bmap, namei, ialloc**

**Hardware Requirement:** A Pentium-class processor with minimum 1 GB of RAM

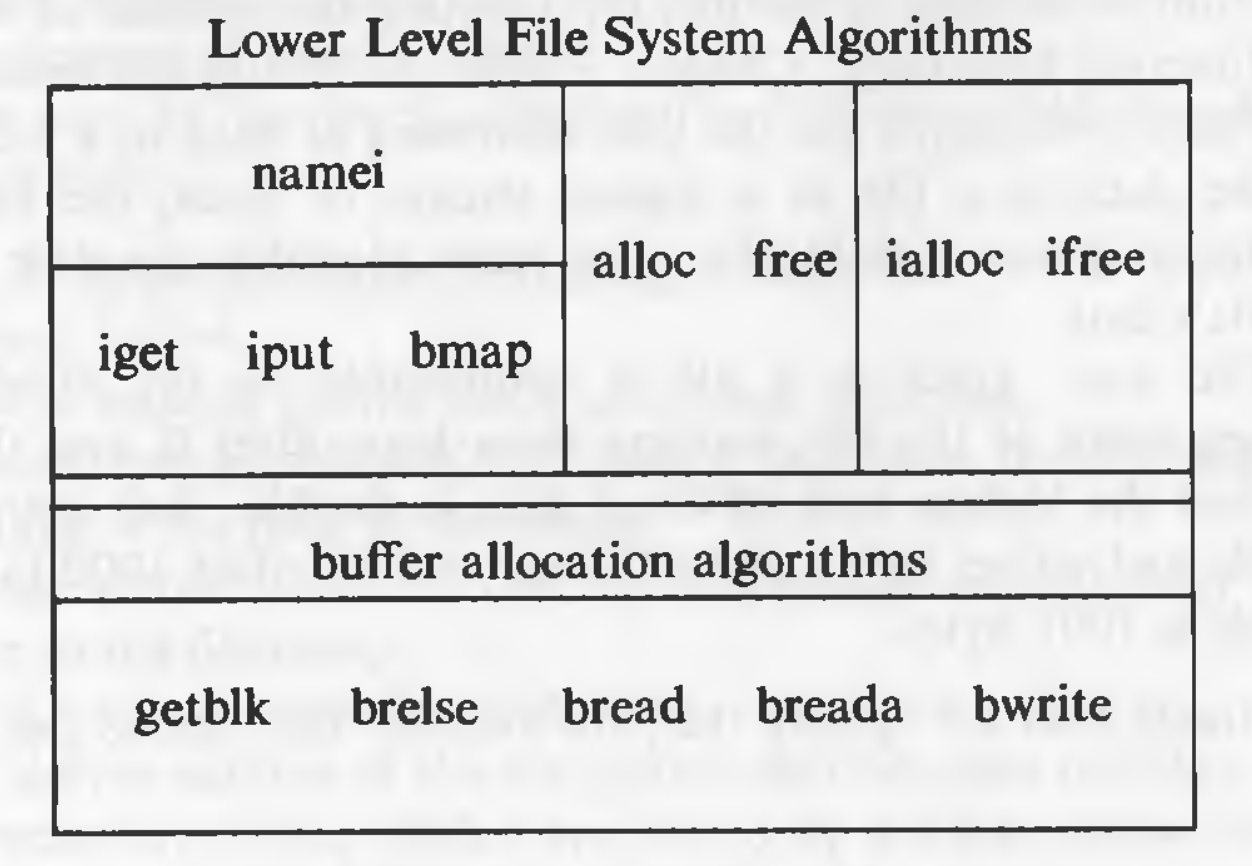
**Software Requirement:** Unix supported Operating System

Theory:

# Internal Representation of Files

Every file a UNIX system has a unique inode. Processes interact with files using well defined system calls. The users specify a file with a character string which is the file's path and then the system get the inode which is mapped to the file which corresponds to the path.

The algorithms described below are above the layer of buffer cache. Diagrammatically, it can be shown like this:

[](https://github.com/suvratapte/Maurice-Bach-Notes/blob/master/Diagrams/Screen_Shot_2017-06-08_at_9.29.01_PM.png)

## Inodes

Inodes exist in a static form on the disk. The kernel reads them into in-core inodes and modifies them.

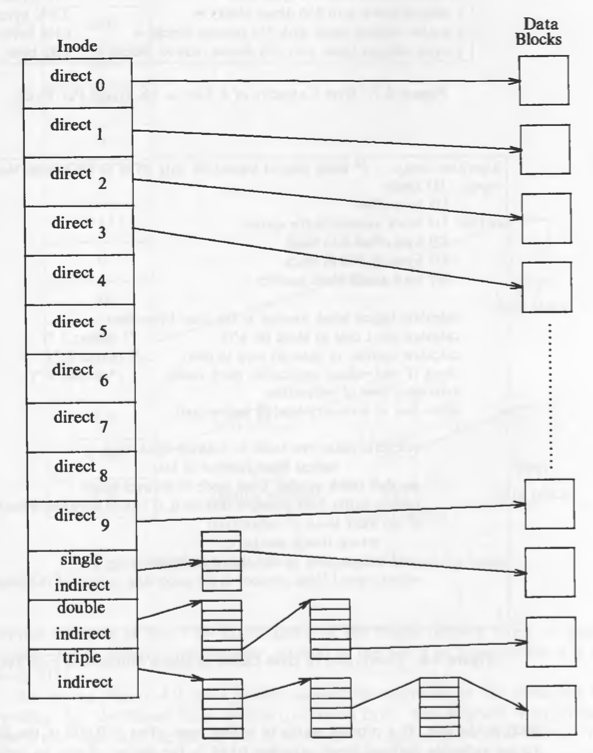
Disk inodes consists of the following fields:

* Owner information: ownership is divided into a user and a group of users. Root user has access to all the files.
* File type: it states whether a file is a normal file, a directory, a block or character special file, or a device file.
* File access permissions: there are 3 types of access permissions: owner, group and others. There are separate permissions for reading, writing and executing. Since execute permission is not applicable to a directory, execute permission for a directory gives the right to search inside the directory.
* Access times: the times at which the file was last accessed and last modified, and the time at which the inodes was last modified
* Number of links: number of places from which the file is being referred.
* Array of disk blocks: even if the users get a logically sequential representation of data in files, the actual data is scattered across the disk. This array keeps the addresses of the disk blocks on which the data is scattered.
* File size: the addressing of the file begins from location 0 from relative to the starting location and the size of the file is the maximum offset of the file + 1. For example, if a user creates a file and writes a byte at offset 999, the size of the file is 1000.

## Structure of a Regular File

In UNIX, the data in files is not stored sequentially on disk. If it was to be stored sequentially, the file size would not be flexible without large fragmentation. In case of sequential storage, the inode would only need to store the starting address and size. Instead, the inode stores the disk block numbers on which the data is present. But for such strategy, if a file had data across 1000 blocks, the inode would need to store the numbers of 1000 blocks and the size of the inode would differ according to the size of the file.

To be able to have constant size and yet allow large files, indirect addressing is used. The inodes have array of size 13 which for storing the block numbers, although, the number of elements in array is independent of the storage strategy. The first 10 members of the array are "direct addresses", meaning that they store the block numbers of actual data. The 11th member is "single indirect", it stores the block number of the block which has "direct addresses". The 12th member is "double indirect", it stores block number of a "single indirect" block. And the 13th member is "triple indirect", it stores block number of a "double indirect" block. This strategy can be extended to "quadruple" or "quintuple" indirect addressing.

[](https://github.com/suvratapte/Maurice-Bach-Notes/blob/master/Diagrams/Screen_Shot_2017-06-09_at_4.07.02_PM.png)

If a logical block on the file system holds 1K bytes and that a block number is addressable by a 32 bit integer, then a block can hold up to 256 block numbers. The maximum file size with 13 member data array is:

10 direct blocks with 1K bytes each = 10K bytes

1 indirect block with 256 direct blocks = 256K bytes

1 double indirect block with 256 indirect blocks = 64M bytes

1 triple indirect block with 256 double indirect blocks = 16G bytes

Code:

#include<stdio.h>

void main()

{

long int offset;

printf("Enter byteoffset : ");

scanf("%d",&offset);

int b\_offset;

int single=-1,doubles=-1,triples=-1;

//Value -1 indicates that there is no redirect of that type

long int block=offset/1024;

if(block < 266)

{

single=block;

b\_offset=offset%1024;

goto SKP;

}

else if(block > 266 && block < 65802)

{

single=block/256;

doubles=block%256-10;

b\_offset=offset%1024;

goto SKP;

}

else if(block > 65802 && block<10000000)

{

single=block/(256\*256);

long int temp=block%(256\*256);

doubles=temp/256;

triples=temp%256;

b\_offset=offset%1024;

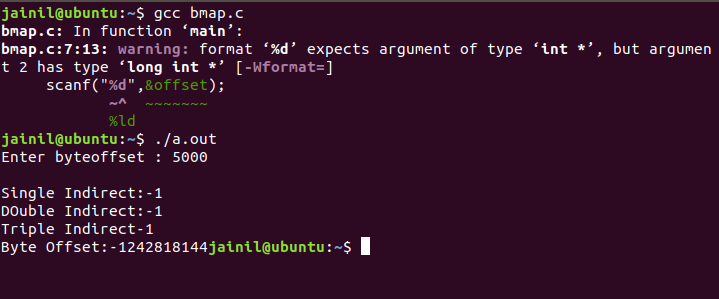
}

SKP:

printf("\nSingle Indirect:%d \nDOuble Indirect:%d \nTriple Indirect%d \nByte Offset:%d",single,doubles,triples,b\_offset);

}

Output:



**Practical-9**

**AIM:-** **Write a program executing on a CPU. Give an example scenario that can cause the process to undergo:  
(a) A voluntary context switch.  
(b) An involuntary context switch**

**Hardware Requirement:** A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

The kernel switches among threads in an effort to share the CPU effectively; this activity is called context switching. When a thread executes for the duration of its time slice or when it blocks because it requires a resource that is currently unavailable, the kernel finds another thread to run and context switches to it. The system can also interrupt the currently executing thread to run a thread triggered by an asynchronous event, such as a device interrupt. Although both scenarios involve switching the execution context of the CPU, switching between threads occurs synchronously with respect to the currently executing thread, whereas servicing interrupts occurs asynchronously with respect to the current thread. In addition, interprocess context switches are classified as voluntary or involuntary.

1. **A voluntary context switch occurs when a thread blocks because it requires a resource that is unavailable.**
2. **An involuntary context switch takes place when a thread executes for the duration of its time slice or when the system identifies a higher-priority thread to run.**

Each type of context switching is done through a different interface. Voluntary context switching is initiated with a call to the sleep() routine, whereas an involuntary context switch is forced by direct invocation of the low-level context-switching mechanism embodied in the mi\_switch() and setrunnable() routines. Asynchronous event handling is triggered by the underlying hardware and is effectively transparent to the system.

**SOLUTION:**

1. **Voluntary Context Switch**

#include<stdio.h>

void main(){

int n;

printf("Enter the input:\n");

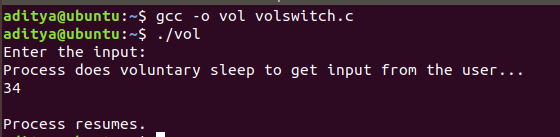
printf("Process does voluntary sleep to get input from the user...\n");

scanf("%d",&n);

printf("\nProcess resumes.\n");

}

**OUTPUT:**

****

1. **Involuntary Context Switch**

#include<stdio.h>

void sl(){

printf("\nProcess is sleeping (Involuntary switch)...\n");

sleep(10);

}

void main()

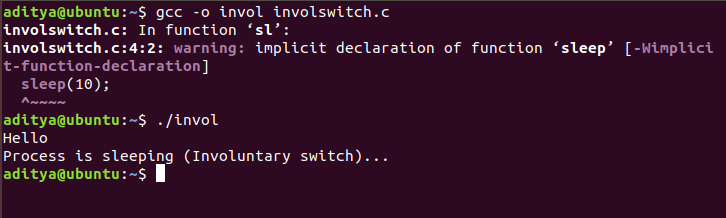
{

printf("Hello");

sl();

}

**OUTPUT:**



**CONCLUSION:**

Thus, we learnt about context switching and its different types and studied an example scenario of how those two types can be caused.

**Practical-10**

**AIM:-** **: Demonstrate the use of signal() function. Write a program to demonstrate the handling of the signals: SIGINT,SIGALRM & SIGQUIT.**

**Hardware Requirement:** A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**1. signal() function**

Code :

#include <stdlib.h>

#include <stdio.h>

#include <signal.h>

void hello(int signum)

{

printf("Hello World!\n");

}

int main()

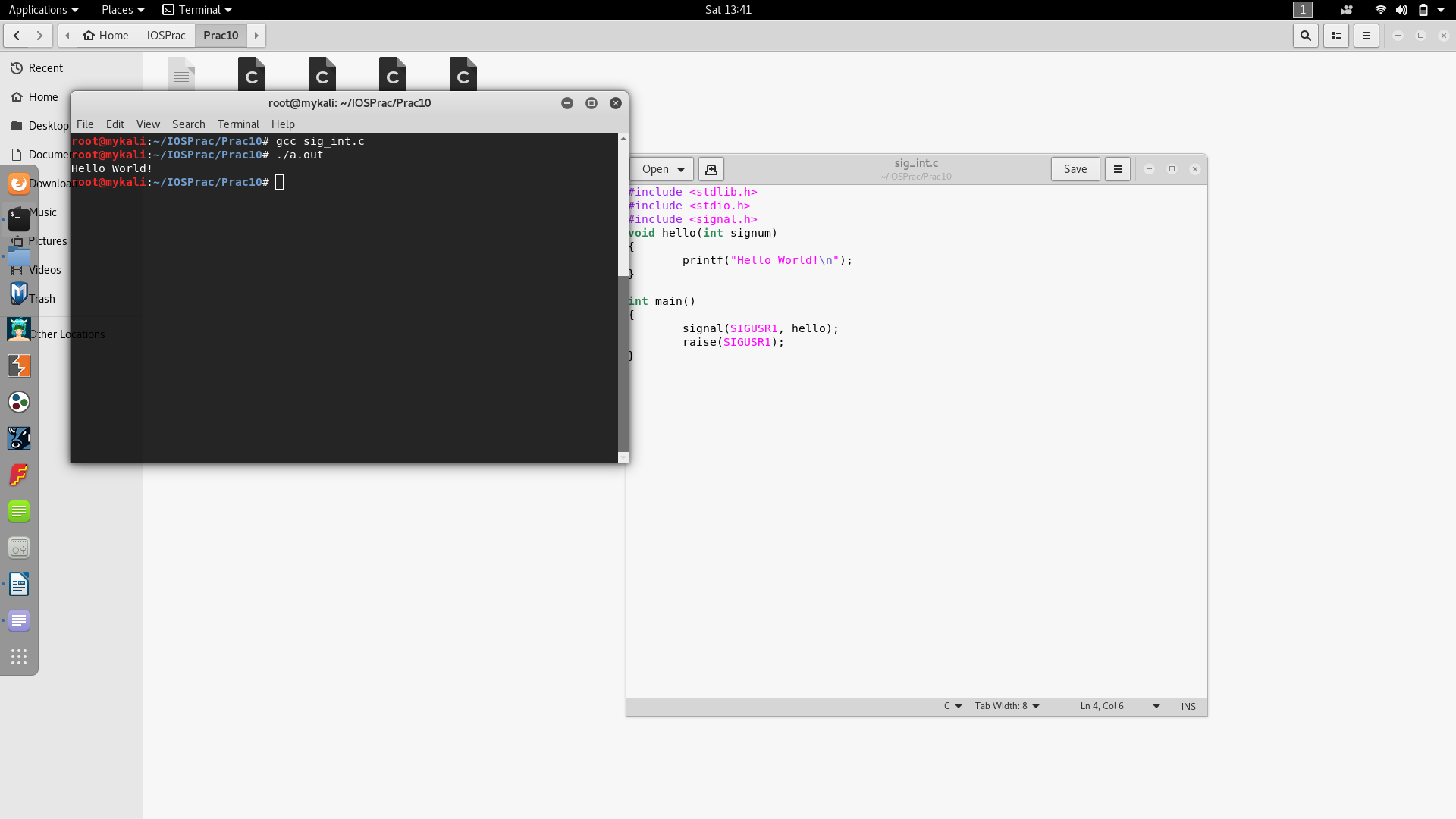
{

signal(SIGUSR1, hello);

raise(SIGUSR1);

}

**Output :**



**2. SIGINT**

**Code:**

#include <stdlib.h>

#include <stdio.h>

#include <signal.h>

void hello(int signum)

{

printf("Hello World!\n");

}

int main()

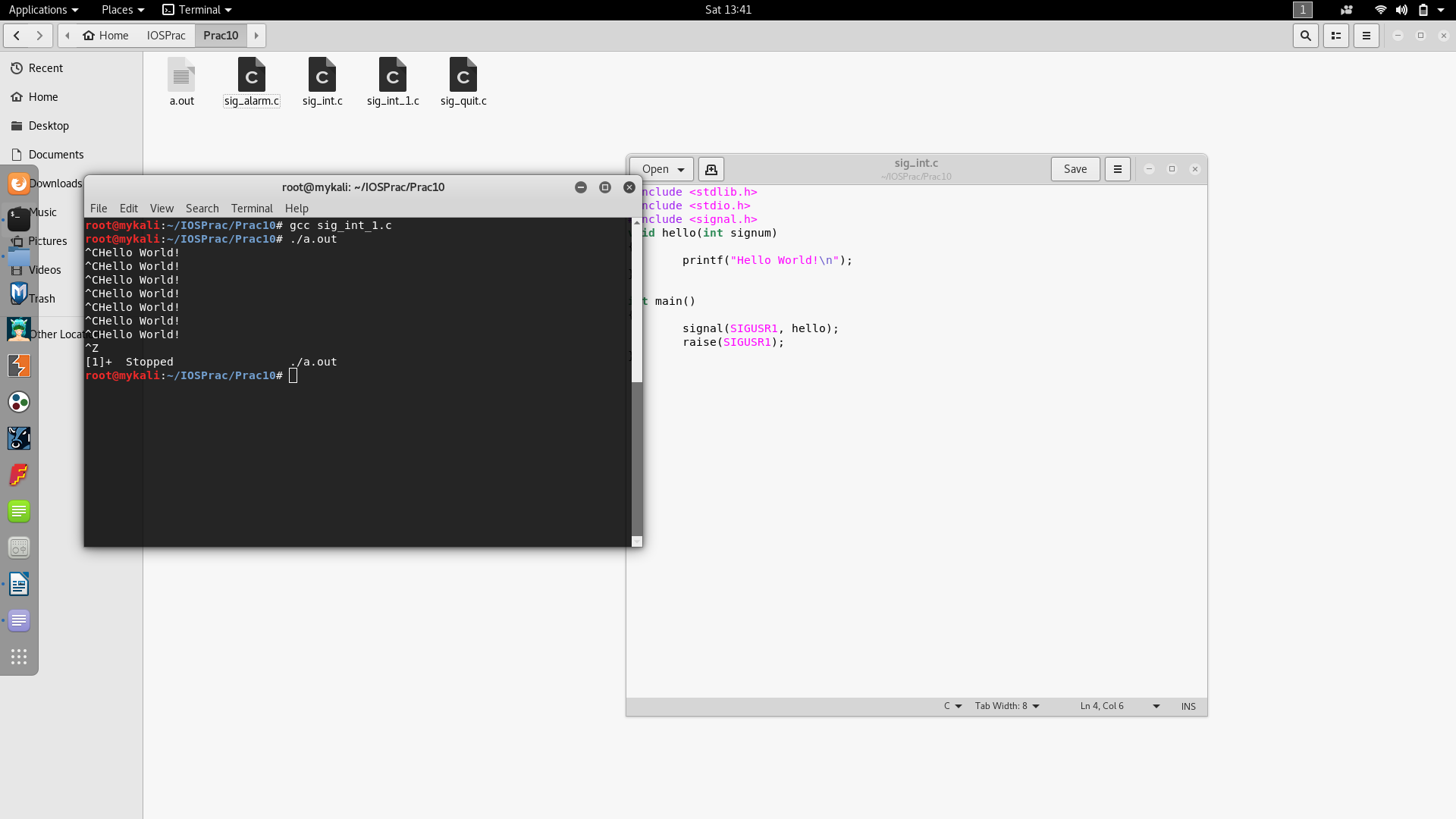
{

signal(SIGINT, hello);

while(1);

}

**Output:**



**3. SIGALRM**

**Code :**

#include <signal.h>

#include <stdio.h>

#include <stdbool.h>

#include <unistd.h>

volatile sig\_atomic\_t print\_flag = false;

void handle\_alarm( int sig ) {

print\_flag = true;

}

int main() {

signal( SIGALRM, handle\_alarm );

alarm( 1 );

for (;;) {

if ( print\_flag ) {

printf( "Hello\n" );

print\_flag = false;

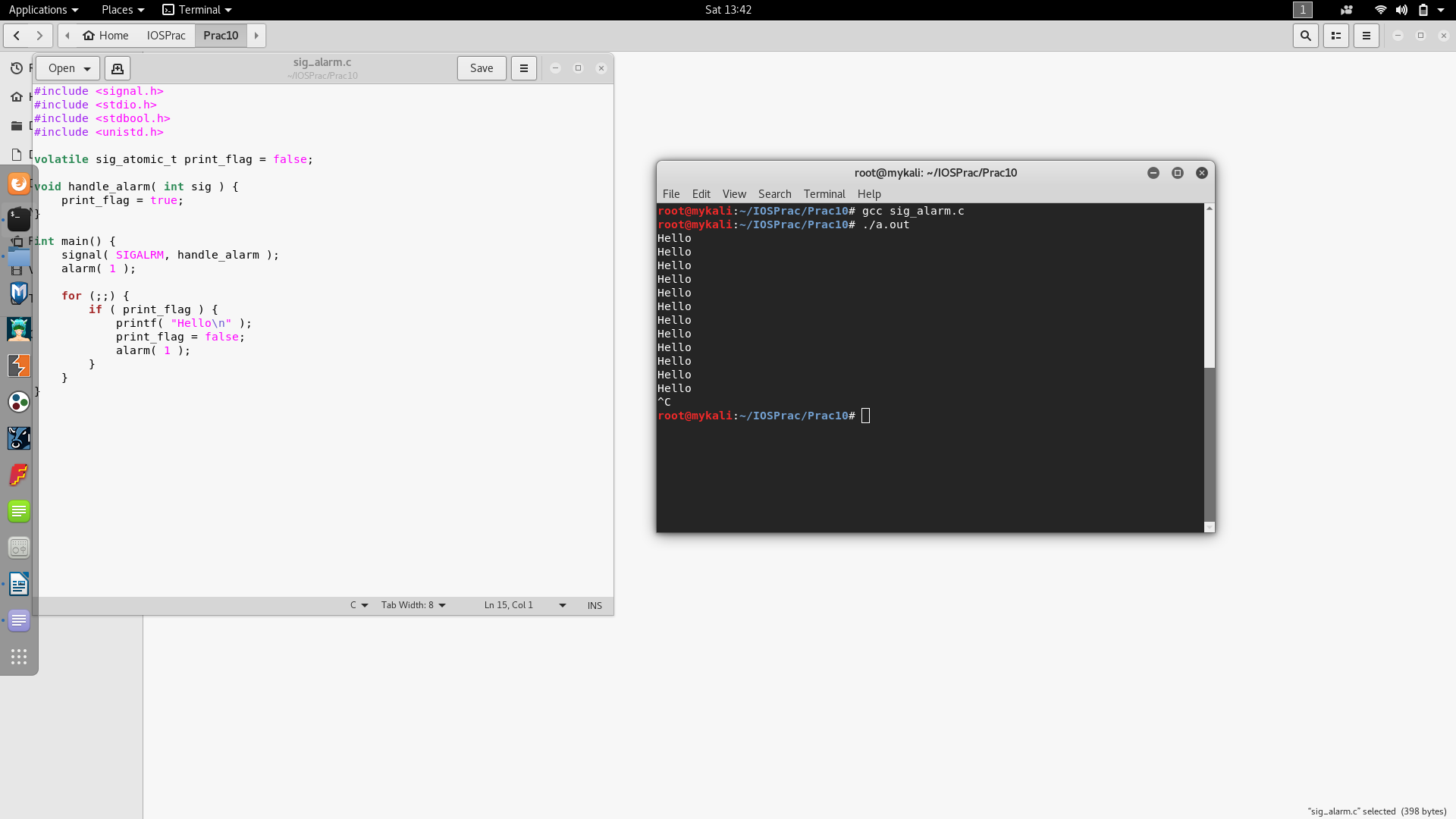
alarm( 1 );

}

}

}

**Output :**



**4. SIGQUIT**

Code :

#include<stdio.h>

#include<signal.h>

#include<unistd.h>

void sig\_handler(int signo)

{

if (signo == SIGINT)

printf("Received SIGINT\n");

else if (signo == SIGQUIT)

printf("Received SIGQUIT\n");

}

int main(void)

{

if (signal(SIGINT, sig\_handler) == SIG\_ERR)

printf("Error in SIGINT\n");

if (signal(SIGQUIT, sig\_handler) == SIG\_ERR)

printf("Error in SIGQUIT\n");

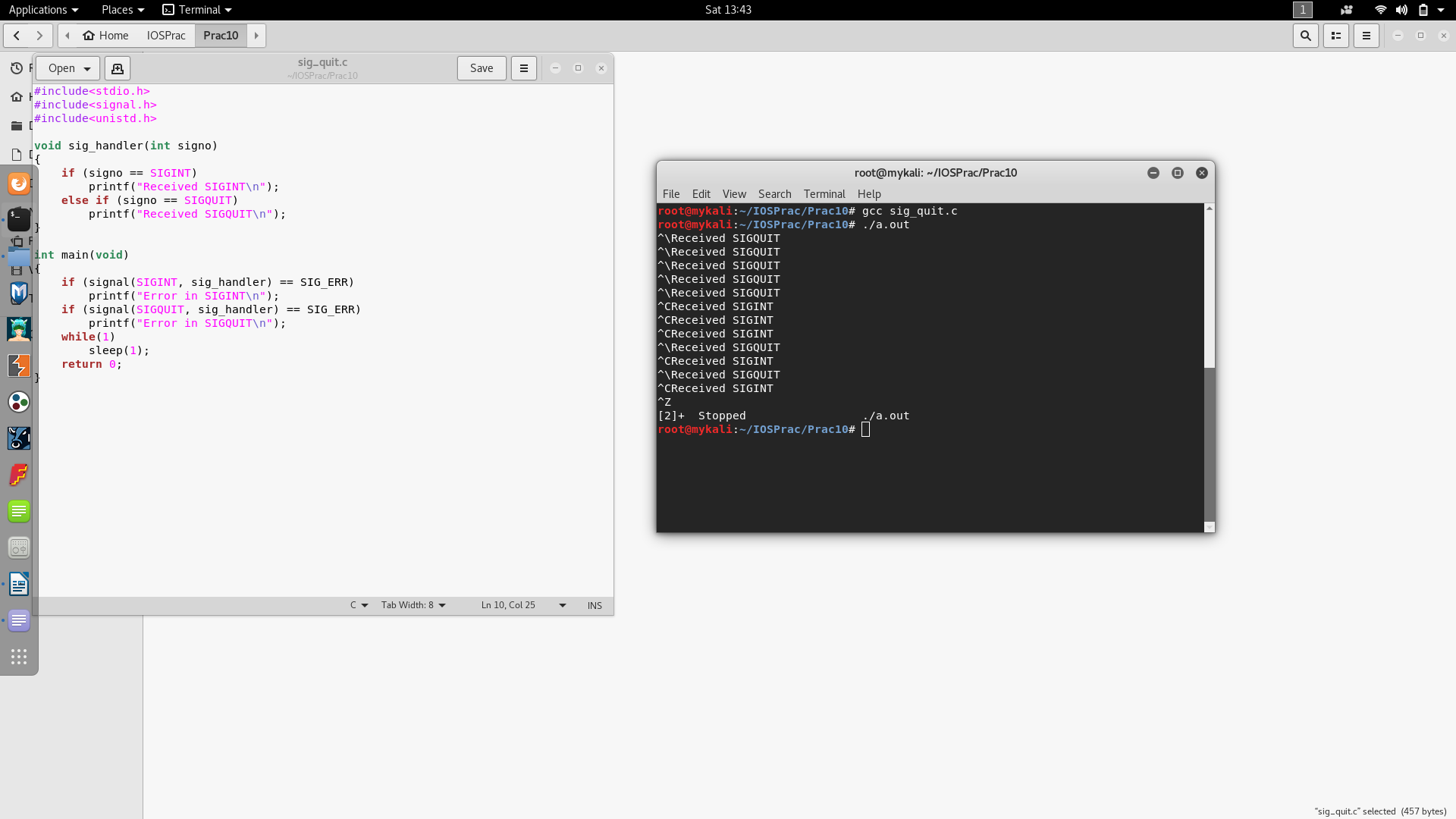
while(1)

sleep(1);

return 0;

}

**Output :**



**Conclusion:** In this practical we have studied about signals. What is signal, how to call it in program, default signal handler and user defined signal handler, about different signals like SIGINT, SIGALRM, SIGQUIT and what they do, etc.

**Practical-11**

**AIM:** Implement an algorithm to create the child processes, also find the real time, processor time, user space time kernel space time for each process after creating. Also display the real time, user space time kernel space time of the parent process.

**SOFTWARE REQUIRED:** Ubuntu terminal

**HARDWARE REQUIRED:** PC/Laptop

**KNOWLEDGE REQUIRED**: Writing code in C/C++, Basics Of UNIX

**THEORY/LOGIC:**

This program has the amount of processor time used by instructions and the system for the parent and child processes.

Gets processor times of interest to a process.

* **struct tms \*buffer**

Points to a memory location where times() can store a structure of information describing processor time used by the current process and other related processes.

times() returns information in a tms structure, which has the following elements:

* **clock\_t tms\_utime**

Amount of processor time used by instructions in the calling process.

Under z/OS® UNIX, this does not include processor time spent running in the kernel. It does include any processor time accumulated for the address space before it became a z/OS UNIX process.

* **clock\_t tms\_stime**

Amount of processor time used by the system.

Under z/OS UNIX, this value represents kernel busy time running on behalf of the calling process. It does not include processor time performing other MVS™ system functions on behalf of the process.

* **clock\_t tms\_cutime**

The sum of tms\_utime and tms\_cutime values for all waited-for child processes which have terminated.

* **clock\_t tms\_cstime**

The sum of tms\_stime and tms\_cstime values for all terminated child processes of the calling process.

clock\_t is an integral type determined in the time.h header file. It measures times in terms of clock ticks. The number of clock ticks in a second (for your installation) can be found in sysconf(\_SC\_CLK\_TCK).

Times for a terminated child can be determined once wait() or waitpid() have reported the child's termination.

Pthreads can not be separately clocked by the times() function because they do not run in a separate process like forked children do.

**SOLUTION:**

#define \_POSIX\_SOURCE

#include <sys/times.h>

#include <time.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdio.h>

#include <unistd.h>

main() {

int status;

longi, j;

structtms t;

clock\_t dub;

inttics\_per\_second;

tics\_per\_second = sysconf(\_SC\_CLK\_TCK);

if (fork() == 0) {

for (i=0, j=0; i<1000000; i++)

j += i;

exit(0);

}

if (wait(&status) == -1)

perror("wait() error");

else if (!WIFEXITED(status))

puts("Child did not exit successfully");

else if ((dub = times(&t)) == -1)

perror("times() error");

else {

printf("process was dubbed %f seconds ago.\n\n",

((double) dub)/tics\_per\_second);

printf(" utimestime\n");

printf("parent: %f %f\n",

((double) t.tms\_utime)/tics\_per\_second,

((double) t.tms\_stime)/tics\_per\_second);

printf("child: %f %f\n",

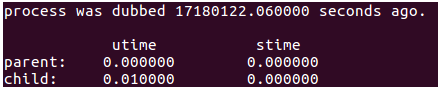
((double) t.tms\_cutime)/tics\_per\_second,

((double) t.tms\_cstime)/tics\_per\_second);

}

}

**OUTPUT:**

****

**CONCLUSION:**

Thus, we learnt about the amount of processor time used by instructions used for parent and child processes.

**Practical-12**

**Aim: Add a new system call to your Kernel of ubuntu**

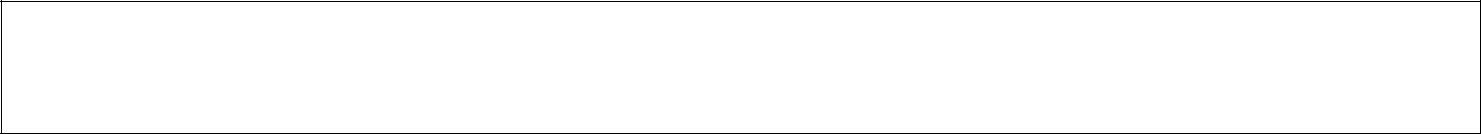
**Theory:**

All of you know about the open source Linux based operating system, Ubuntu. The advantage of these open source operating systems is that you can make modifications to the operating system such as adding your own system calls.

**Steps to add new system call to your kernel of Ubuntu:**

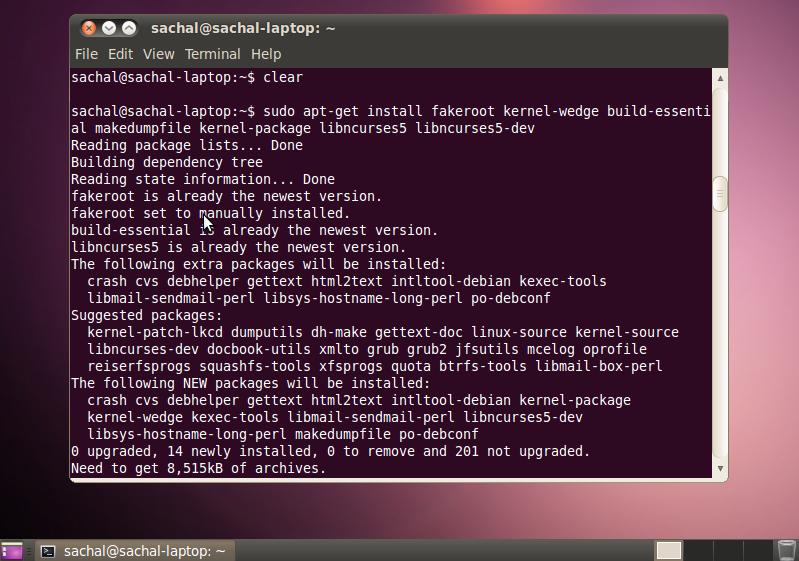
**Step1: Download the kernel’s source code**

We‟ll be using Ubuntu 16.04 (32-bit) but these steps would work on later versions as well. Now as we‟ll be modifying the kernel, first of all we need to download the necessary packages required for modifying the kernel. Fire up the terminal and type in the following command:



sudo apt-get install fakeroot kernel-wedge build-essential makedumpfile kernel-package libncurses5

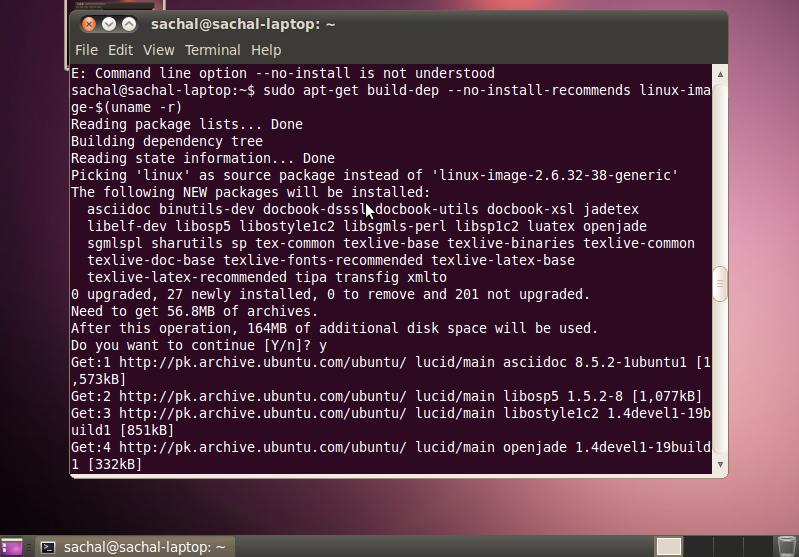
libncurses5-dev



Now enter the following command to download packages related to your specific Ubuntu version:



sudo apt-get build-dep –no-install-recommends linux-image-$(uname -r)



Now you have to make a directory in which you want to place the kernel‟s source code. Simply type in mkdir~/src. Then type cd ~/src in order to navigate to the directory. Now type in the following command to download the source code to the directory:

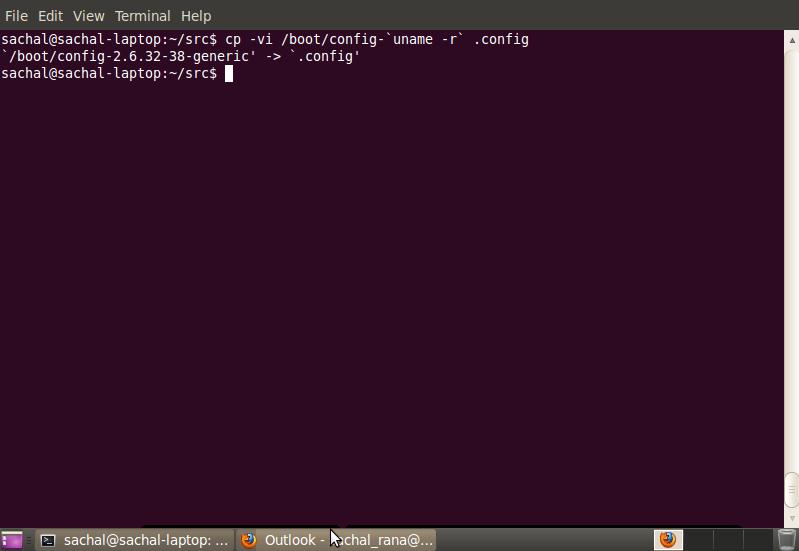


apt-get source linux-image-$(uname -r)

Now you will find several files and folders inside the source code folder. Navigate to the directory named linux-(insert version here). In my case, I got the 2.6.32 kernel. So I typed in cd linux-2.6.32 Now we will make a copy of the configuration file in order to be on the safe side because we don‟t want to mess up the original configuration file. Type in the following command to do so:



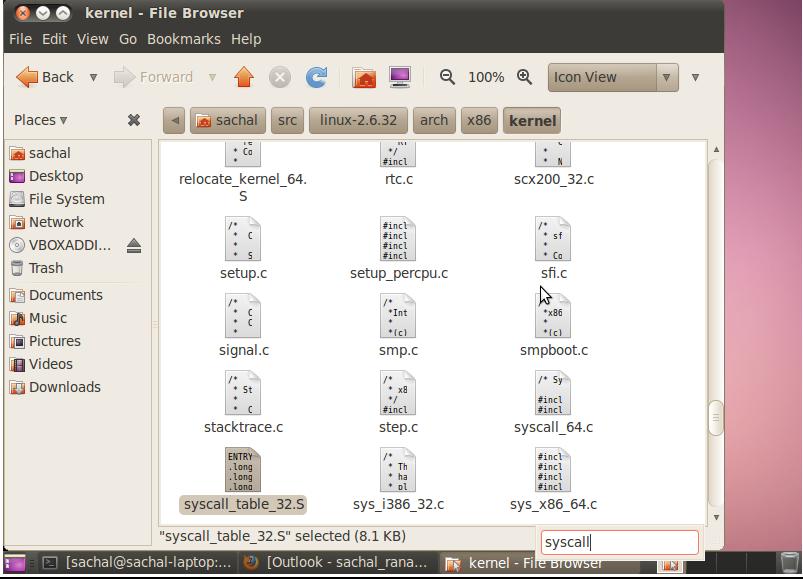
cp -vi /boot/config-`uname -r` .config



Now if you want to make changes to the configuration file, simply type in make menuconfig. If you do not wish to, simply skip this step. Make the changes and simply save and exit.

**Step2: Adding “Hello World” system call**

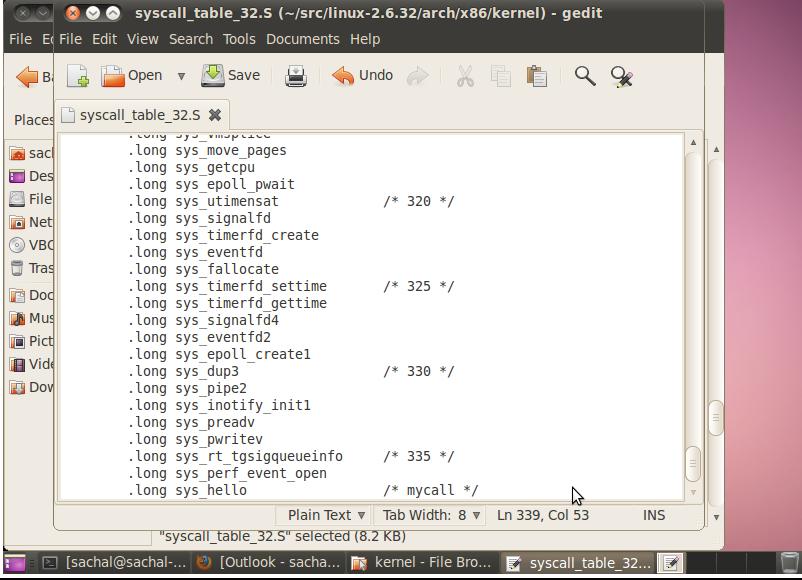
We‟ll be adding our own system call which will print “Hello World” in the log file. Open the folder where you downloaded the sourcecode. Navigate to *arch/x86/kernel.* Open the file called *syscall\_table\_32.S.*



Now at the end of the file add in the following line. You can name your system call according to your own liking.



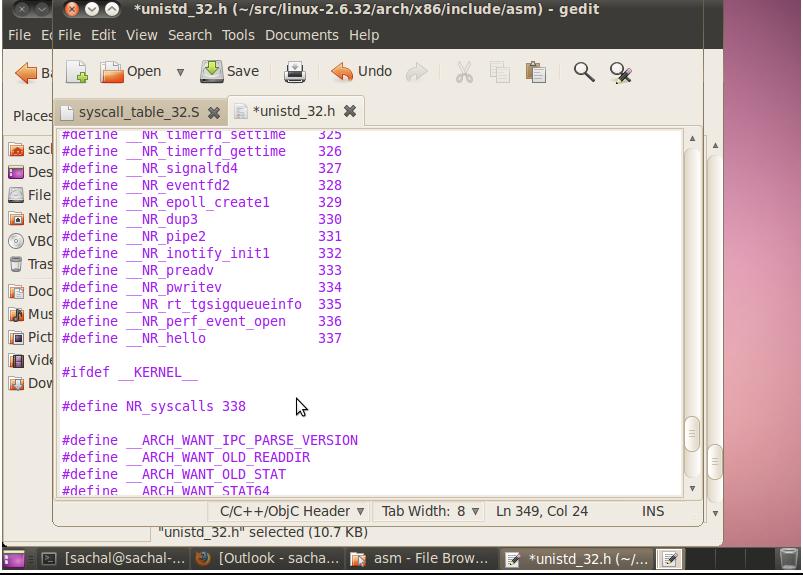
*.long sys\_hello*



Now open the file in the path *arch/x86/include/asm/unistd\_32.h*. You will notice that a #define is defined for each system call. At the end of the huge macro definition, add a definition for our new system call. Assign the number according to the trend.



*#define \_\_NR\_hello 337*

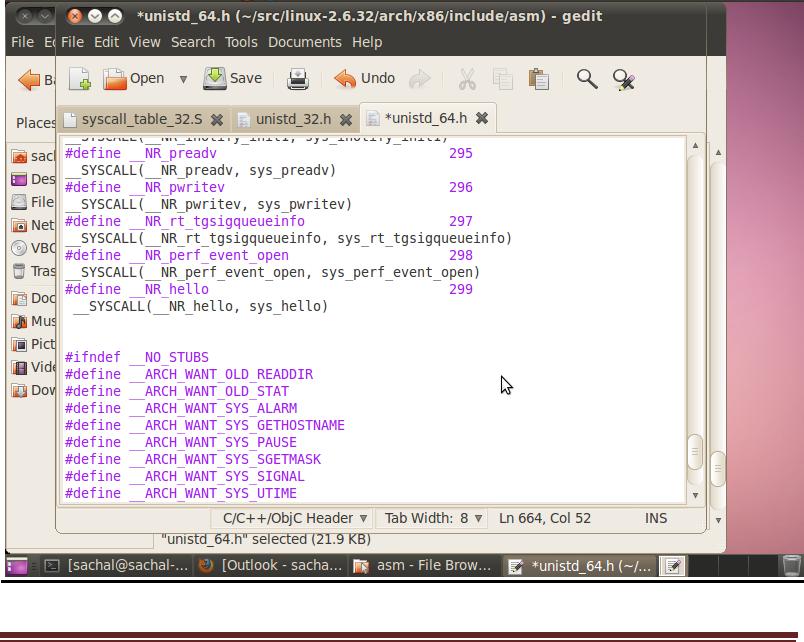


Now accordingly increment the macro of total syscalls. In our case we now had 338. See the screenshot above for reference.

Now open the file at path *arch/x86/include/asm/unistd\_64.h* and add the following line after the last syscall.



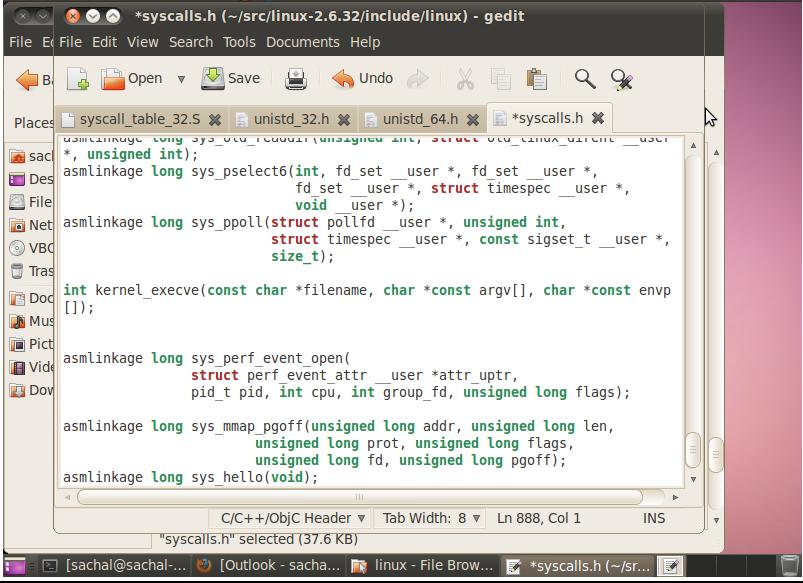
*#define \_\_NR\_hello 299 \_\_SYSCALL(\_\_NR\_hello,sys\_hello)*



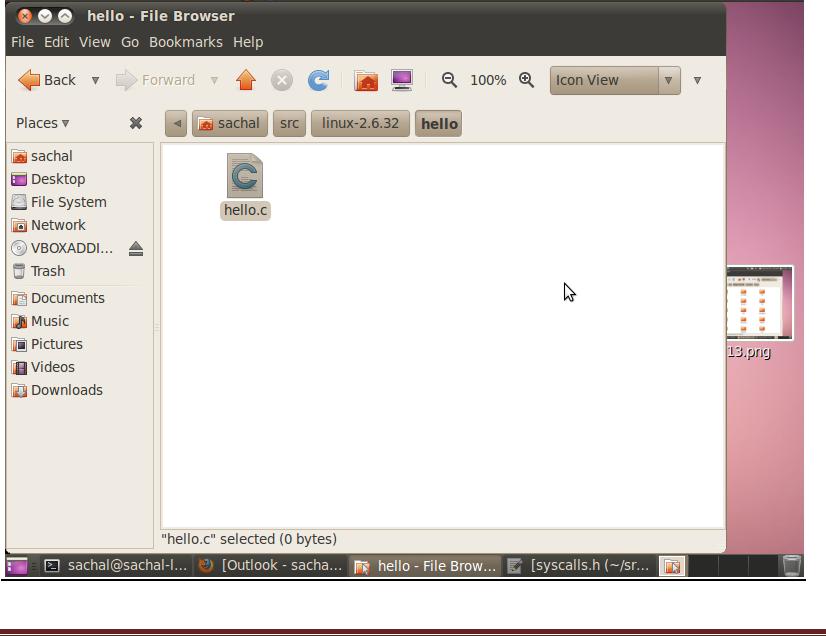
Afterwards, open the file at path *arch/x86/include/asm/syscalls.h* and add the prototype of hello world system call before #endif.



*asmlinkage long sys\_hello(void);*



Open the root directory of kernel sources. Make a directory named “hello” and in that directory make a file and name it “hello.c”.

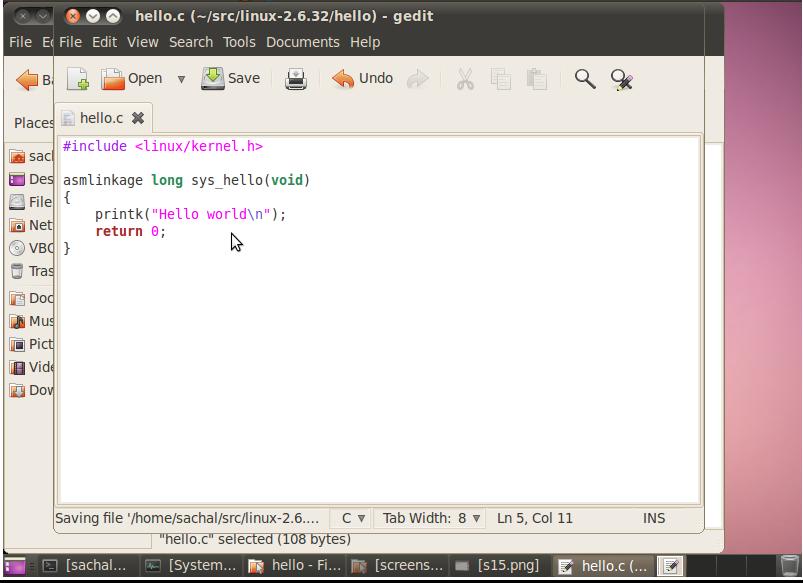


Inside that file write this piece of code:

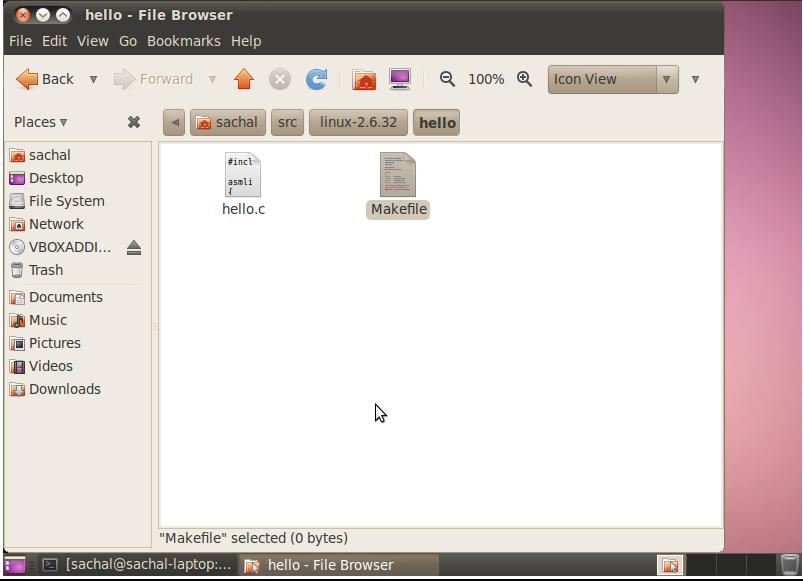
*#include <linux/kernel.h> asmlinkage long sys\_hello(void)*

*{*

*printk(“Hello world\n”); return 0;*

*}*

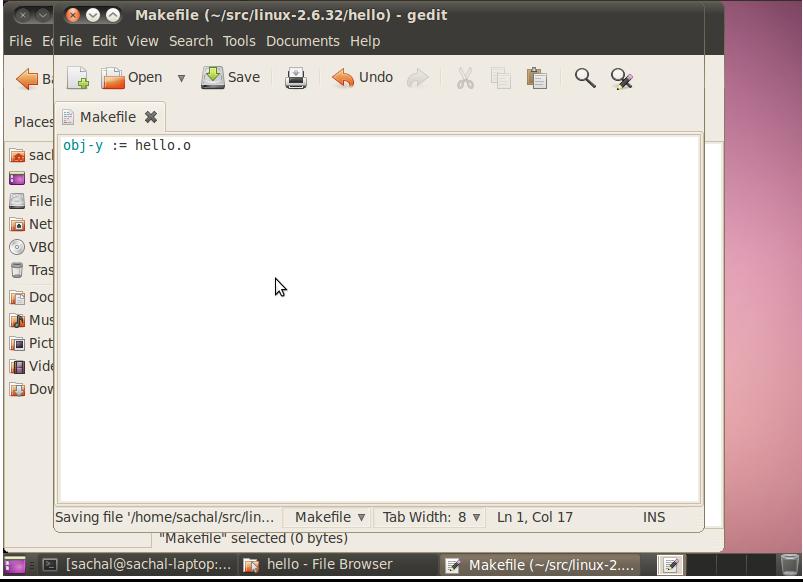
Now make a file named “Makefile” without any extension in the hello directory.



Now write the following line in it:



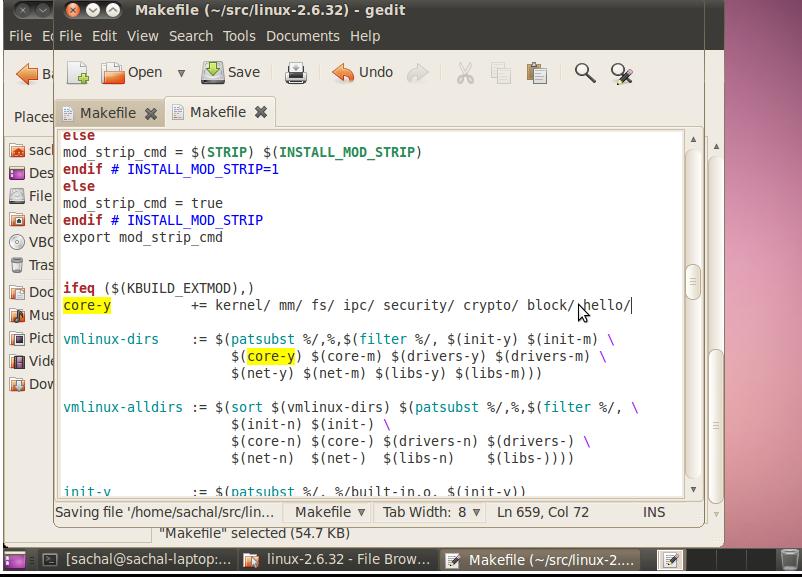
*obj-y := hello.o*



Open the Makefile present in the root directory and modify the line “*core-y += kernel/ mm/ fs/ ipc/* *security/ crypto/ block/”* to



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



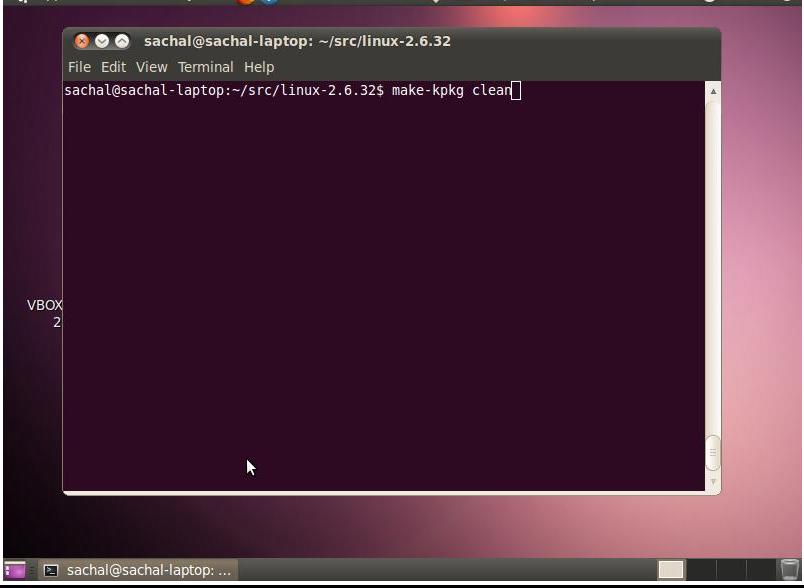
We have successfully embedded our own system call. Now we just have to recompile the kernel in order to make our system call a part of the operating system.

Compiling The Kernel

Now open the terminal again and navigate to the source code folder. Run the following command:



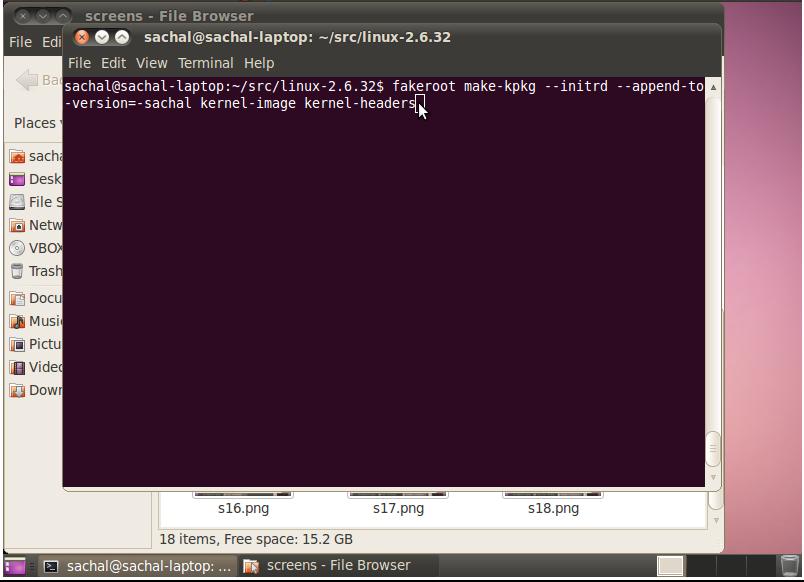
*make-kpkg clean*



Now in order to speed up your kernel compilation, type *EXPORT CONCCURENCY\_LEVEL =* *3.* The number 3 signifies that I have 2 cores. If you have, for example, a quad core machine, enter5. Basically its number of cores+1.

Then run this command:*fakeroot make-kpkg* *–initrd* *–append-to-version=-sachal kernel-image* *kernel-headers*

NOTE: You can substitute Sachal with anything.



After you are done with compiling, change your current directory to the one where your kernel was previously.Then run the following two commands:



*sudo dpkg -i linux-image-2.6.32.60+drm33.26-sachal\_2.6.32.60+drm33.26-sachal -*



*10.00.Custom\_i386.deb*

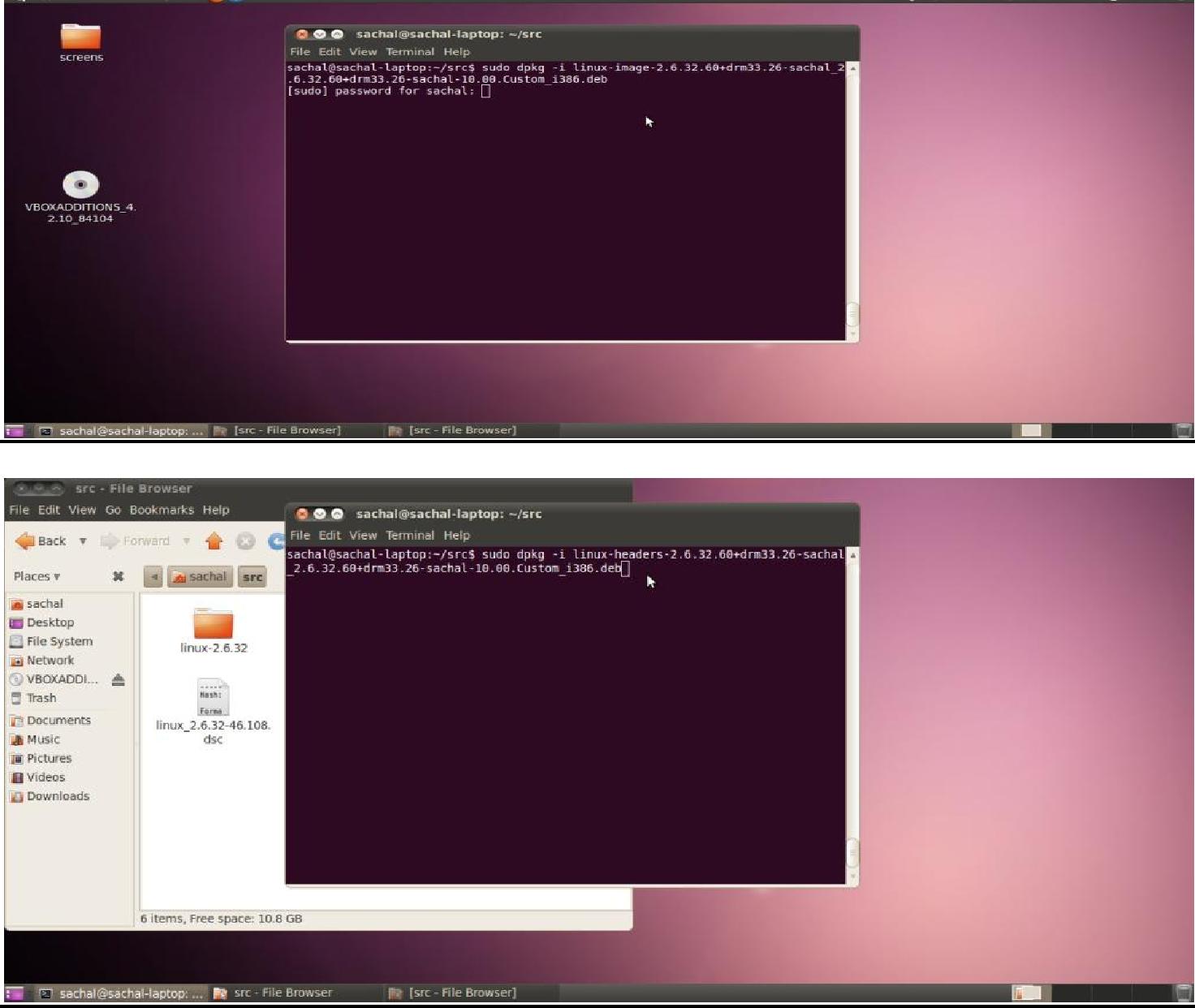


*sudo dpkg -i linux-headers-2.6.32.60+drm33.26-sachal\_2.6.32.60+drm33.26-sachal-*

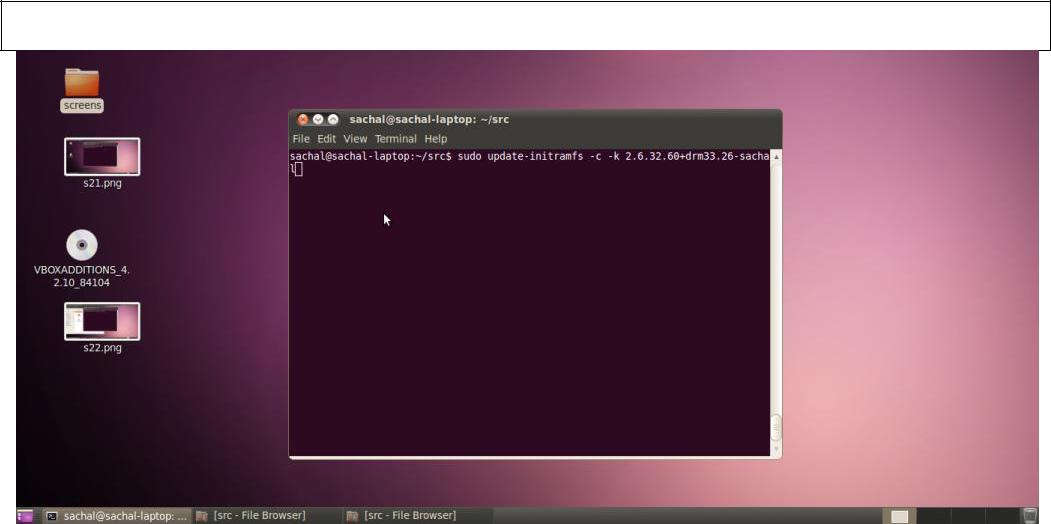


*10.00.Custom\_i386.deb*

NOTE: The text after linux-image is according to the files which are made after compiling the kernel. Change it accordingly.



Now run the following command:

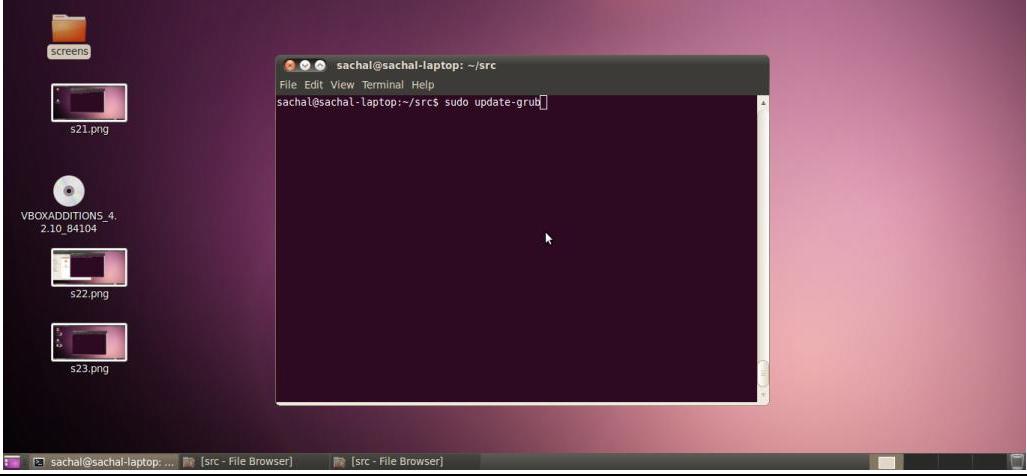


*sudo update-initramfs -c -k 2.6.32.60+drm33.26-sachal*

Now to update your GRUB menu with your own kernel, run the following command:



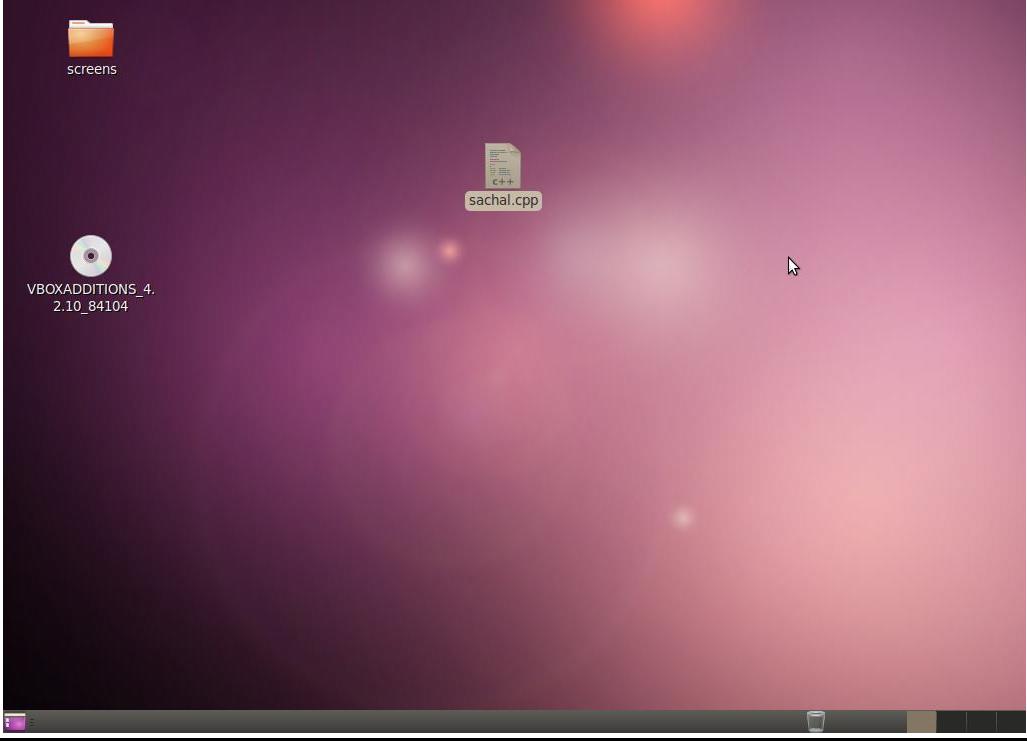
*sudo update-grub*



Now simply reboot your system and choose your own kernel version from the GRUB menu.

Testing “Hello World” System Call

Make a file on the desktop and name it with a .cpp extension.



Open the file and write in the following piece of code:

*#include <stdio.h> #include <linux/kernel.h> #include <sys/syscall.h> #include <unistd.h> #define \_\_NR\_hello 337 long hello\_syscall(void)*

*{*

*return syscall(\_\_NR\_hello);*

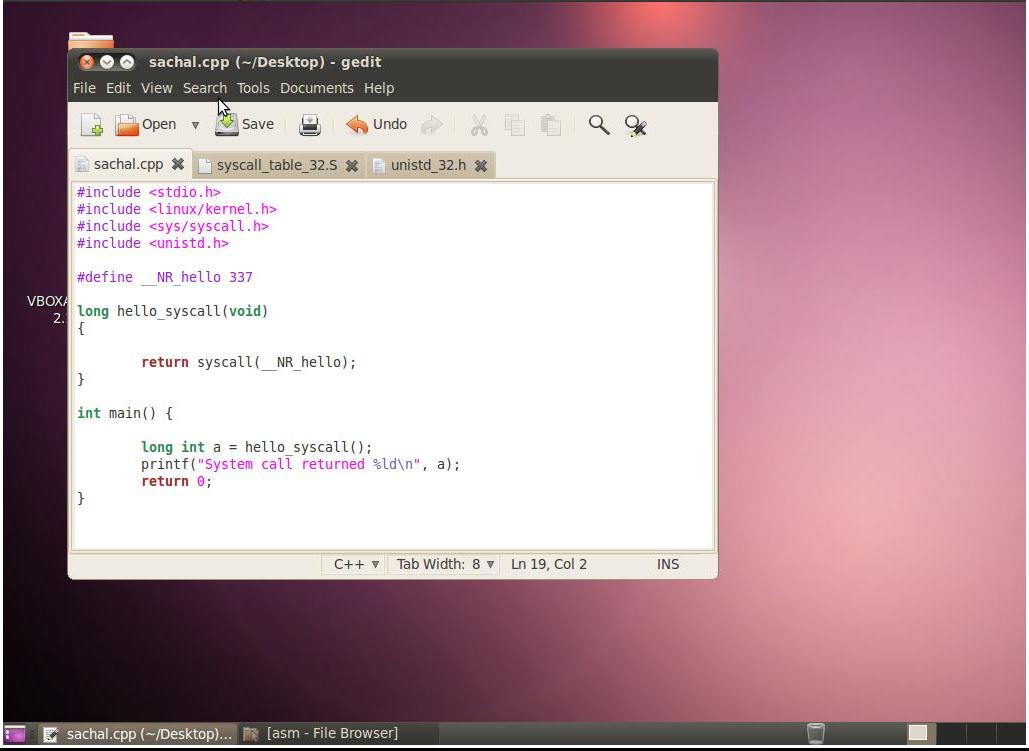
*}*

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

*{*

*long int a = hello\_syscall(); printf(“System call returned %ld\n”, a); return 0;*

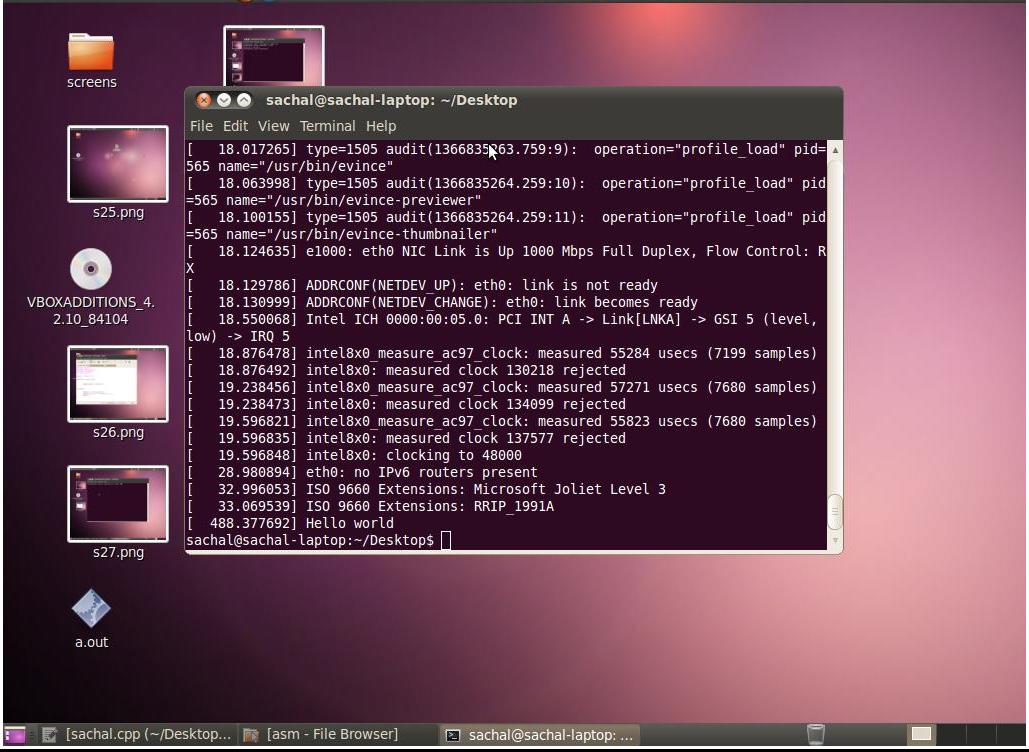
*}*



Now fire up the terminal and run your code. If the system call returns 0, it means that the call was successful.



Then type *dmesg* in the terminal to open up the log and see the *Hello World* statement printed in the log.



Congratulations! You have successfully added your own system call.