Experiment 3:

Subject: CSL403 Operating System Lab

NAME: GINI CHACKO

ROLL: 8942

CLASS: SE COMPS B

Aim: To study Process and File Management System Calls

Objectives: To gain practical experience with designing and implementing concepts of operatingsystems such as system calls, CPU scheduling, process management, memory management,file systems and deadlock handling using C language in Linux environment.

Problem Statement:
(1.) Process related System Calls.
a) Create a child process in Linux using the fork system call. From the childprocess obtain the process ID of both child and parent by using getpid and getppid system call.
b) Explore wait and waitpid before termination of process.
c) Zombie and Orphan Process

(2)File related system calls
Program to copy contents of one file (source) to another file (destination). Finally displaying contents of destination file.

1. Explain fork(), getpid(), getppid(),wait() and waitpid() with syntax.
Answer: **a.] fork() -** Fork system call is used for creating a new process, which is called **child process**, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is 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(). **Syntax -** pid_t fork(void); **b.] getpid() -** returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames. **Syntax -** pid_t getpid(void);

c.] getppid() - returns the process ID of the parent of the calling process. If the calling process was created by the fork() function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for **init** process.

Syntax - pid_t getppid(void);

d.] wait() - 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.

Syntax - pid_t wait(int *stat_loc);

e.] waitpid() - The **waitpid**() system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behaviour is modifiable via the *options* argument.

Syntax - pid_t waitpid(pid_t pid, int *status, int options);

Program and Output Section:

1.] Process related System Calls.

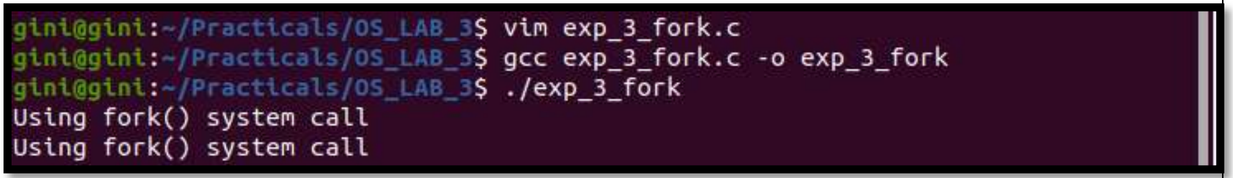
a) Create a child process in Linux using the fork system call. From the childprocess obtain the process ID of both child and parent by using getpid and getppid system call.

- **Create a child process in Linux using the fork system call**

CODE:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
   fork();
   printf("Using fork() system call\n");
   return 0;
}
```

OUTPUT:



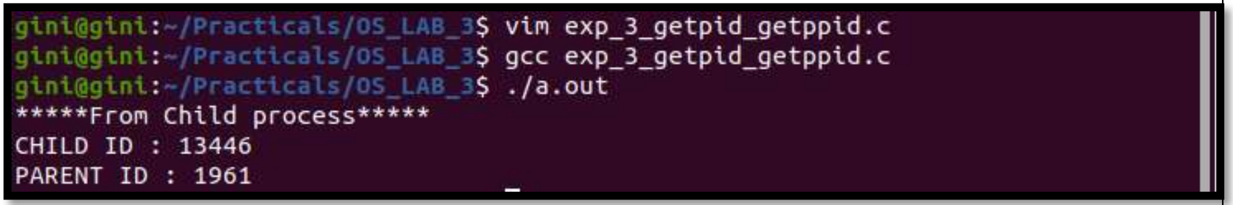
```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_fork.c
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_fork.c -o exp_3_fork
gini@gini:~/Practicals/OS_LAB_3$ ./exp_3_fork
Using fork() system call
Using fork() system call
```

- **From the childprocess obtain the process ID of both child and parent by using getpid and getppid system call.**

CODE:

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
        int ret;
        ret = fork();
        if(ret == 0)
        {
                printf("*****From Child process*****\n");
                printf("CHILD ID : %d\n",getpid());
                printf("PARENT ID : %d\n",getppid());
        }
}
```

OUTPUT:



```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_getpid_getppid.c
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_getpid_getppid.c
gini@gini:~/Practicals/OS_LAB_3$ ./a.out
*****From Child process*****
CHILD ID : 13446
PARENT ID : 1961
```

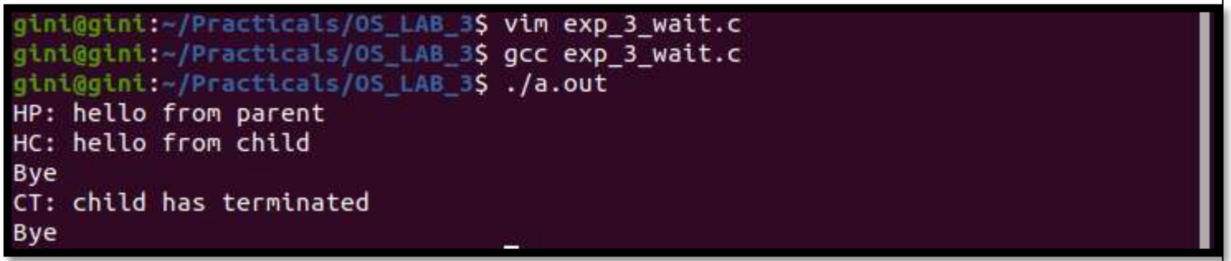
b) Explore wait and waitpid before termination of process.

- **To demonstrate working of wait()**

CODE:

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
   if (fork()== 0)
      printf("HC: hello from child\n");
   else
   {
      printf("HP: hello from parent\n");
      wait(NULL);
      printf("CT: child has terminated\n");
   }

   printf("Bye\n");
   return 0;
}
```

OUTPUT:



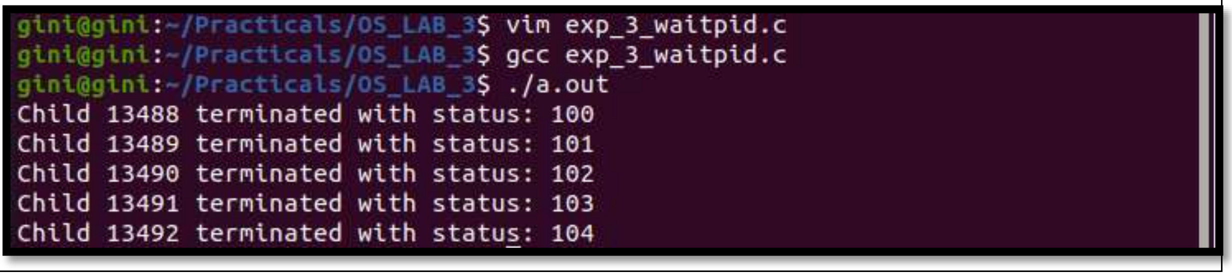
```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_wait.c
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_wait.c
gini@gini:~/Practicals/OS_LAB_3$ ./a.out
HP: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye
```

- **To demonstrate working of waitpid()**

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

void waitexample()
{
   int i, stat;
   pid_t pid[5];
   for (i=0; i<5; i++)
   {
      if ((pid[i] = fork()) == 0)
      {
         sleep(1);
         exit(100 + i);
      }
   }
   for (i=0; i<5; i++)
   {
      pid_t cpid = waitpid(pid[i], &stat, 0);
      if (WIFEXITED(stat))
         printf("Child %d terminated with status: %d\n",
                cpid, WEXITSTATUS(stat));
   }
}
int main()
{
   waitexample();
   return 0;
}
```

OUTPUT:



```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_waitpid.c
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_waitpid.c
gini@gini:~/Practicals/OS_LAB_3$ ./a.out
Child 13488 terminated with status: 100
Child 13489 terminated with status: 101
Child 13490 terminated with status: 102
Child 13491 terminated with status: 103
Child 13492 terminated with status: 104
```

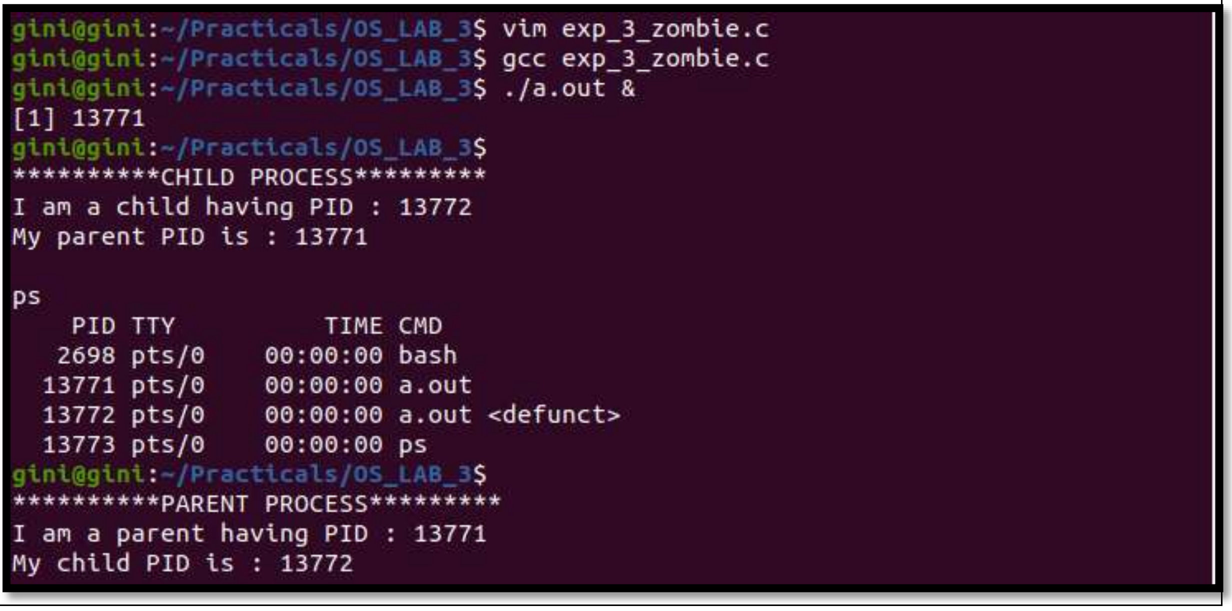
c) Zombie and Orphan Process

- **To demonstrate working of Zombie Process**

CODE:

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
        // fork() creates child process identical to parent
        pid_t p;
        p = fork();
        // child process
        if (p == 0){
           printf("\n**********CHILD PROCESS*********\n");
           printf("I am a child having PID : %d\n", getpid());
           printf("My parent PID is : %d\n\n", getppid());
        }
        // parent process
        else
        {
           sleep(3);
           printf("\n**********PARENT PROCESS*********");
                 printf("\nI am a parent having PID : %d\n", getpid());
           printf("My child PID is : %d\n", p);
        }
        return 0;
}
```

OUTPUT:



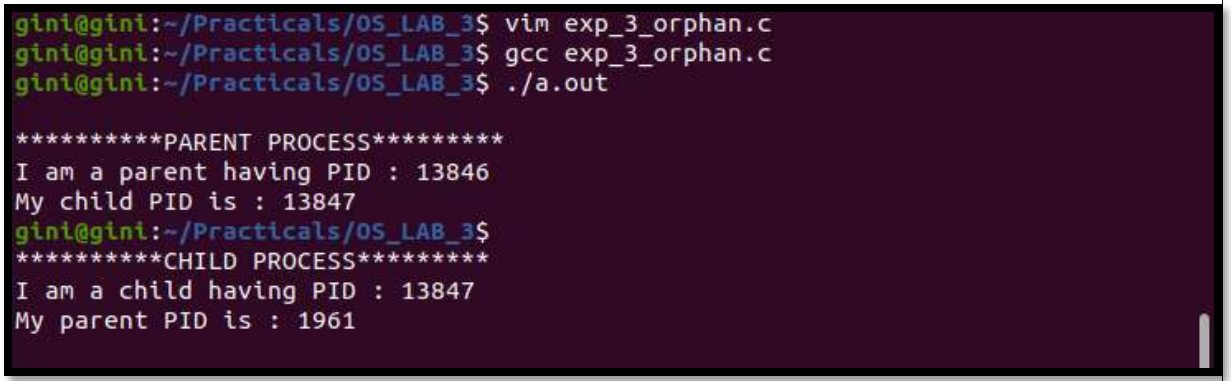
```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_zombie.c
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_zombie.c
gini@gini:~/Practicals/OS_LAB_3$ ./a.out &
[1] 13771
gini@gini:~/Practicals/OS_LAB_3$
**********CHILD PROCESS*********
I am a child having PID : 13772
My parent PID is : 13771

ps
    PID TTY          TIME CMD
   2698 pts/0    00:00:00 bash
  13771 pts/0    00:00:00 a.out
  13772 pts/0    00:00:00 a.out <defunct>
  13773 pts/0    00:00:00 ps
gini@gini:~/Practicals/OS_LAB_3$
**********PARENT PROCESS*********
I am a parent having PID : 13771
My child PID is : 13772
```

- **To demonstrate working of Orphan Process**

CODE:

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
        // fork() creates child process identical to parent
        pid_t p;
        p = fork();
        // child process
        if (p == 0){
           sleep(3);
           printf("\n**********CHILD PROCESS*********\n");
           printf("I am a child having PID : %d\n", getpid());
           printf("My parent PID is : %d\n\n", getppid());
        }


        // parent process
        else
        {
           printf("\n**********PARENT PROCESS*********");
                 printf("\nI am a parent having PID : %d\n", getpid());
           printf("My child PID is : %d\n", p);
        }

        return 0;

}
```

OUTPUT:



```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_orphan.c
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_orphan.c
gini@gini:~/Practicals/OS_LAB_3$ ./a.out

**********PARENT PROCESS*********
I am a parent having PID : 13846
My child PID is : 13847
gini@gini:~/Practicals/OS_LAB_3$
**********CHILD PROCESS*********
I am a child having PID : 13847
My parent PID is : 1961
```

2. Explain creat(), open(), close(), read() and write() with syntax.

Answer:

a.] Create(): Used to Create a new empty file.

Syntax: 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

b.] open(): Used to Open the file for reading, writing or both.

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

Parameters

- **Path :** path to file which you want to use
- use absolute path begin with "/", when you are not work in same directory of file.
- Use relative path which is only file name with extension, when you are work in same directory of file.
- **flags :** How you like to use
- **O_RDONLY**: read only, **O_WRONLY**: write only, **O_RDWR**: read and write, **O_CREAT**: create file if it doesn't exist, **O_EXCL**: prevent creation if it already exists.

c.] close(): Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

Syntax : int close(int fd);

Parameter

- **fd :**file descriptor

Return

- **0** on success.
- **-1** on error.

d.] read(): 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 : 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

e.] 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.

Syntax : 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

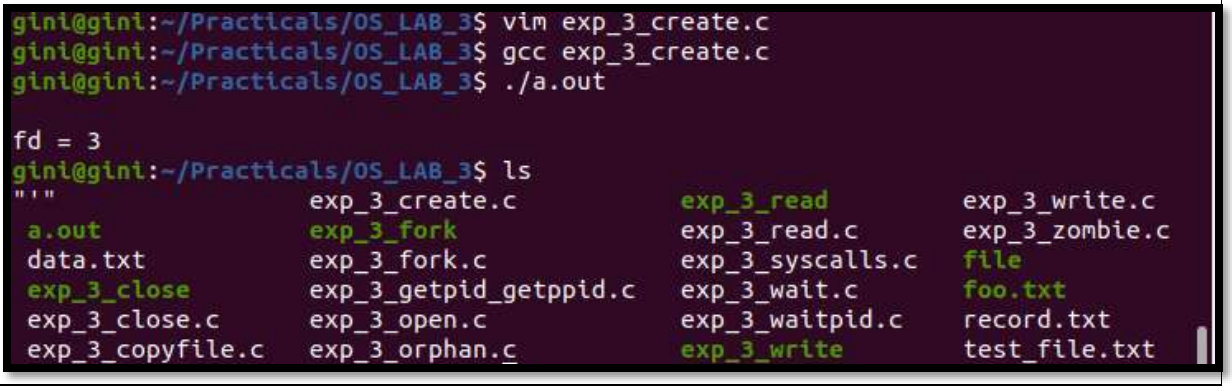
Program and Output Section:

- **To illustrate create()**

CODE:

```
#include<stdio.h>
#include <fcntl.h>
#include<errno.h>
extern int errno;

int main()
{
   int fd = creat("test_file.txt", O_RDONLY);
   printf("\nfd = %d\n", fd);

   if (fd ==-1)
   {
      // print which type of error have in a code
      printf("Error Number %d\n", errno);

      // print program detail "Success or failure"
      perror("Program");
   }
   return 0;
}
```

OUTPUT:

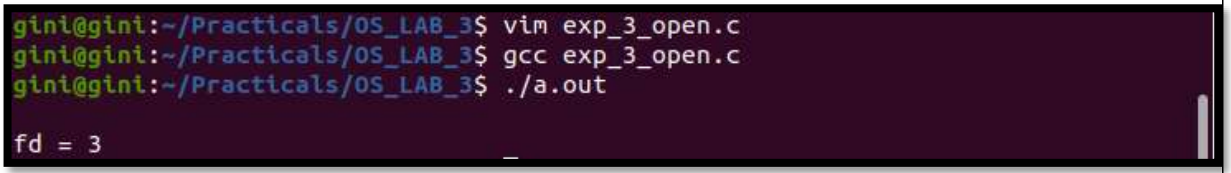


```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_create.c
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_create.c
gini@gini:~/Practicals/OS_LAB_3$ ./a.out

fd = 3
gini@gini:~/Practicals/OS_LAB_3$ ls
"'"                   exp_3_create.c          exp_3_read          exp_3_write.c
 a.out                exp_3_fork              exp_3_read.c        exp_3_zombie.c
 data.txt             exp_3_fork.c            exp_3_syscalls.c    file
 exp_3_close          exp_3_getpid_getppid.c  exp_3_wait.c        foo.txt
 exp_3_close.c        exp_3_open.c            exp_3_waitpid.c     record.txt
 exp_3_copyfile.c     exp_3_orphan.c          exp_3_write         test_file.txt
```

- **To illustrate open()**

CODE:
```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main()
{
    // if file does not have in directory
    // then file foo.txt is created.
    int fd = open("foo.txt", O_RDONLY | O_CREAT);

    printf("\nfd = %d\n", fd);

    if (fd ==-1)
    {
        // print which type of error have in a code
        printf("Error Number % d\n", errno);

        // print program detail "Success or failure"
        perror("Program");
    }
    return 0;
}
```

OUTPUT:

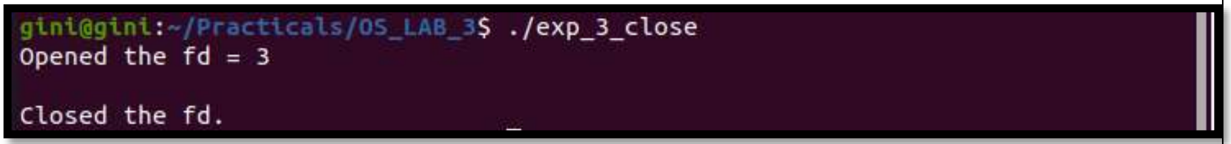


```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_open.c
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_open.c
gini@gini:~/Practicals/OS_LAB_3$ ./a.out

fd = 3
```

- **To illustrate close()**

CODE:
```
#include<stdio.h>
#include <fcntl.h>
int main()
{
    int fd1 = open("foo.txt", O_RDONLY);
    if (fd1 < 0)
    {
        perror("c1");
        exit(1);
    }
    printf("opened the fd = %d\n", fd1);

    // Using close system Call
    if (close(fd1) < 0)
    {
        perror("c1");
        exit(1);
    }
    printf("closed the fd.\n");
}
```

OUTPUT:



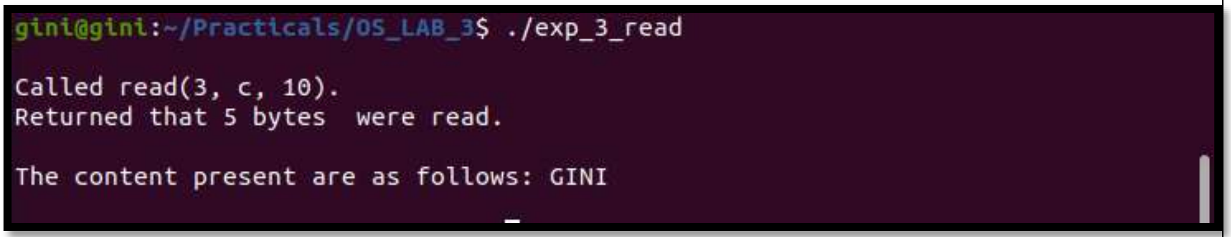
```
gini@gini:~/Practicals/OS_LAB_3$ ./exp_3_close
Opened the fd = 3

Closed the fd.
```

- **To illustrate read()**

CODE:

```
#include<stdio.h>
#include <fcntl.h>
int main()
{
  int fd, sz;
  char *c = (char *) calloc(100, sizeof(char));

  fd = open("foo.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = read(fd, c, 10);
  printf("\nCalled read(%d, c, 10).\nReturned that"
         " %d bytes  were read.\n", fd, sz);
  c[sz] = '\0';
  printf("The content present are as follows: %s\n", c);
}
```

OUTPUT:



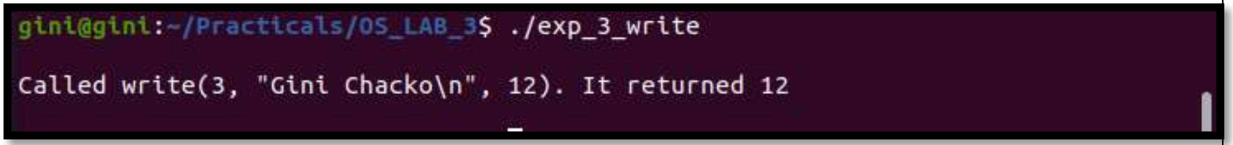
```
gini@gini:~/Practicals/OS_LAB_3$ ./exp_3_read

Called read(3, c, 10).
Returned that 5 bytes  were read.

The content present are as follows: GINI
```

- **To illustrate write()**

CODE:

```
#include<stdio.h>
#include <fcntl.h>
main()
{
  int sz;

  int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd < 0)
  {
     perror("r1");
     exit(1);
  }

  sz = write(fd, "Gini Chacko\n", strlen("Gini Chacko\n"));

  printf("\nCalled write(%d, \"Gini Chacko\\n\", %d)." " It returned %d\n", fd, strlen("Gini
Chacko\n\n"), sz);

  close(fd);
}
```

OUTPUT:



```
gini@gini:~/Practicals/OS_LAB_3$ ./exp_3_write

Called write(3, "Gini Chacko\n", 12). It returned 12
```

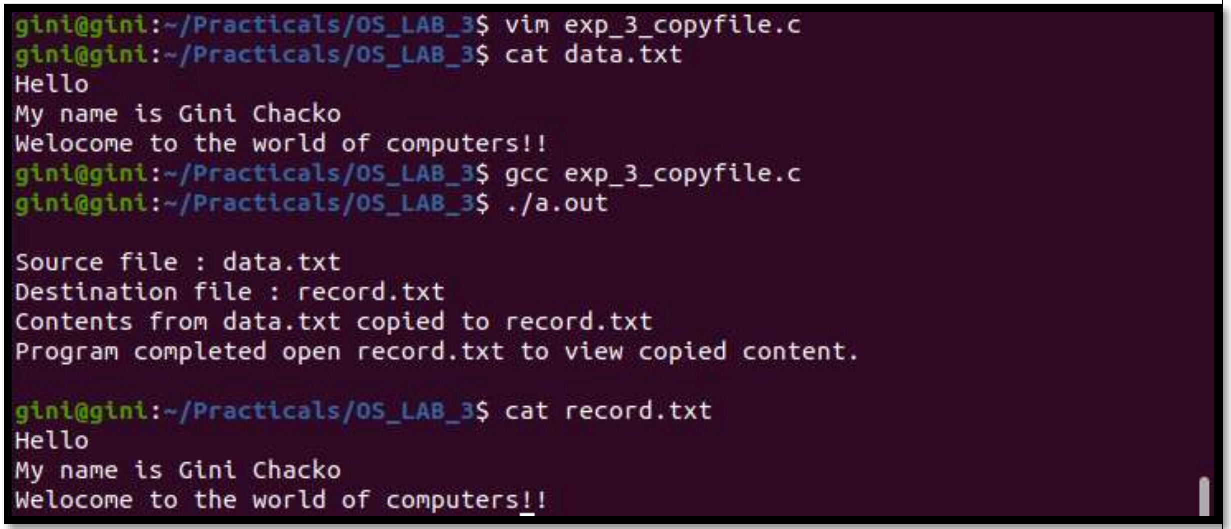
(2)File related system calls

Program to copy contents of one file (source) to another file (destination). Finally displaying contents of destination file.

CODE:

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

int main()
{
 FILE *source, *destination;
 char ch;
 source = fopen("data.txt","r");

 if(source==NULL)
 {
  printf("File error!");
  exit(1);
 }

 destination = fopen("record.txt","w");
 if(source==NULL)
 {
  printf("File error!");
  exit(1);
 }

 do
 {
  ch = fgetc(source);
  fputc(ch, destination);
 }while(ch!=EOF);

 fclose(source);
 fclose(destination);
 printf("\nSource file : data.txt\n");
 printf("Destination file : record.txt\n");
 printf("Contents from data.txt copied to record.txt\n");
 printf("Program completed open record.txt to view copied content.\n\n");
 return 0;
}
```

OUTPUT:



```
gini@gini:~/Practicals/OS_LAB_3$ vim exp_3_copyfile.c
gini@gini:~/Practicals/OS_LAB_3$ cat data.txt
Hello
My name is Gini Chacko
Welocome to the world of computers!!
gini@gini:~/Practicals/OS_LAB_3$ gcc exp_3_copyfile.c
gini@gini:~/Practicals/OS_LAB_3$ ./a.out

Source file : data.txt
Destination file : record.txt
Contents from data.txt copied to record.txt
Program completed open record.txt to view copied content.

gini@gini:~/Practicals/OS_LAB_3$ cat record.txt
Hello
My name is Gini Chacko
Welocome to the world of computers!!
```