

OPERATING SYSTEMS LAB

LAB # 7 System Calls

Lab # 7

OPERATING SYSTEMS

5th Semester-SE

UET Taxila

System Calls in C++

A system call is just what its name implies—a request for the operating system to do something on behalf of the user's program

- open()
- read()
- write()
- close()
- wait()
- fork()
- exec()

open()

Next is the open() system call. open() lets you open a file for reading, writing, or reading and writing.

```
int open(file_name, mode)
```

where file_name is a pointer to the character string that names the file and mode defines the file's access permissions if the file is being created.

read() and write()

The read() system call does all input and the write() system call does all output. When used together, they provide all the tools necessary to do input and output.

Both read() and write() take three arguments. Their prototypes are:

```
int read(file_descriptor, buffer_pointer, transfer_size)
```

```
int file_descriptor;
```

```
char *buffer_pointer;
```

```
unsigned transfer_size;
```

```
int write(file_descriptor, buffer_pointer, transfer_size)
```

```
int file_descriptor;
```

```
char *buffer_pointer;
```

```
unsigned transfer_size;
```

where file_descriptor identifies the I/O channel, buffer_pointer points to the area in memory where the data is stored for a read() or where the data is taken for a write(), and transfer_size defines the maximum

close()

To close a channel, use the close() system call. The prototype for the close() system call is:

```
int close(file_descriptor)
```

```
int file_descriptor;
```

Implementation of open(), read(), write() and close() functions

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

```
#include<fcntl.h>
```

```
int main()
```

```
{
```

```
    int fd;
```

```
    char buffer[80];
```

```
    static char message[] = "Hello, world";
```

```
    fd = open("myfile",O_RDWR);
```

```
    if (fd != -1)
```

```
    {
```

```
        printf("myfile opened for read/write access\n");
```

```
        write(fd, message, sizeof(message));
```

```
        lseek(fd, 0, 0); /* go back to the beginning of the file */
```

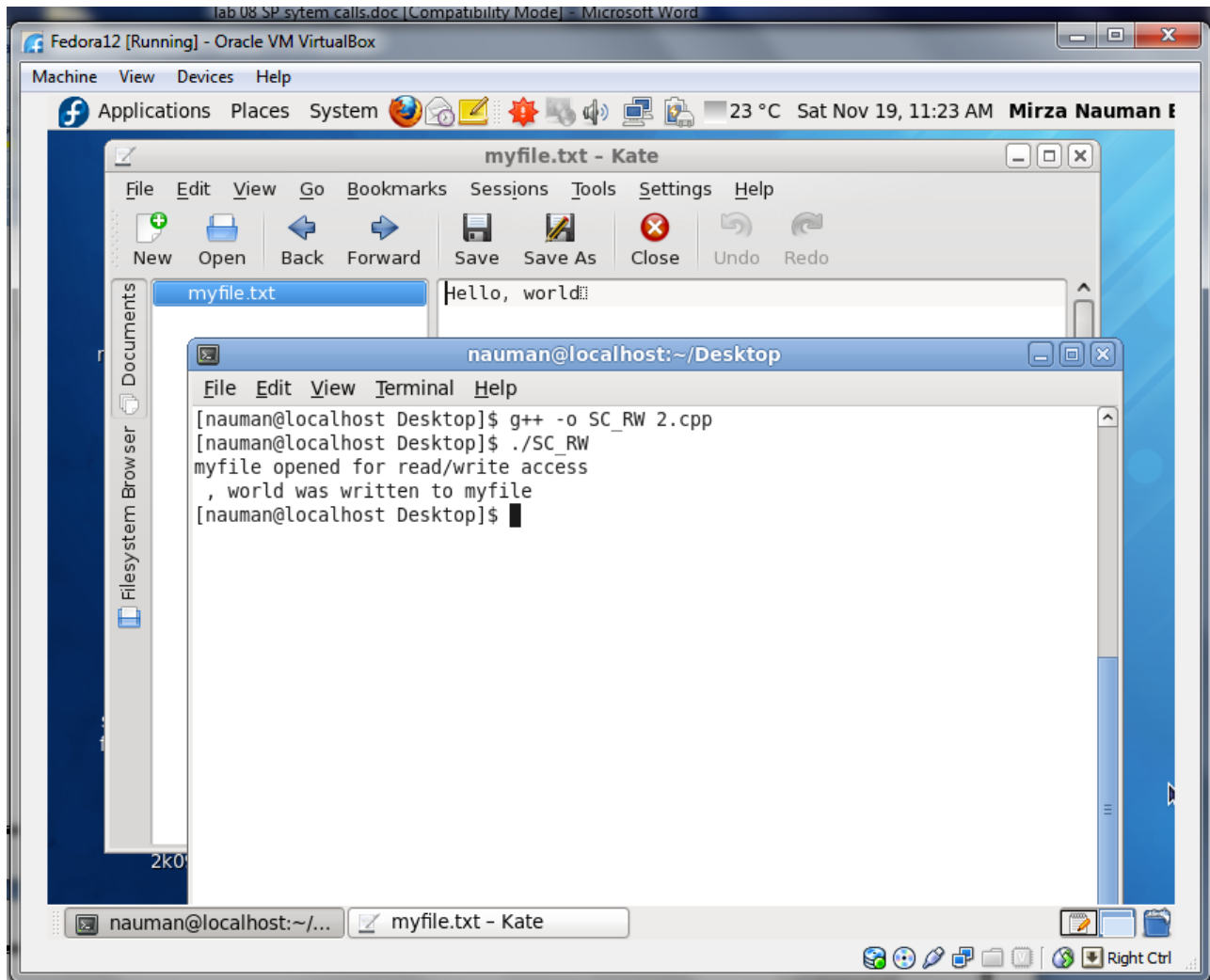
```
        read(fd, buffer, sizeof(message))
```

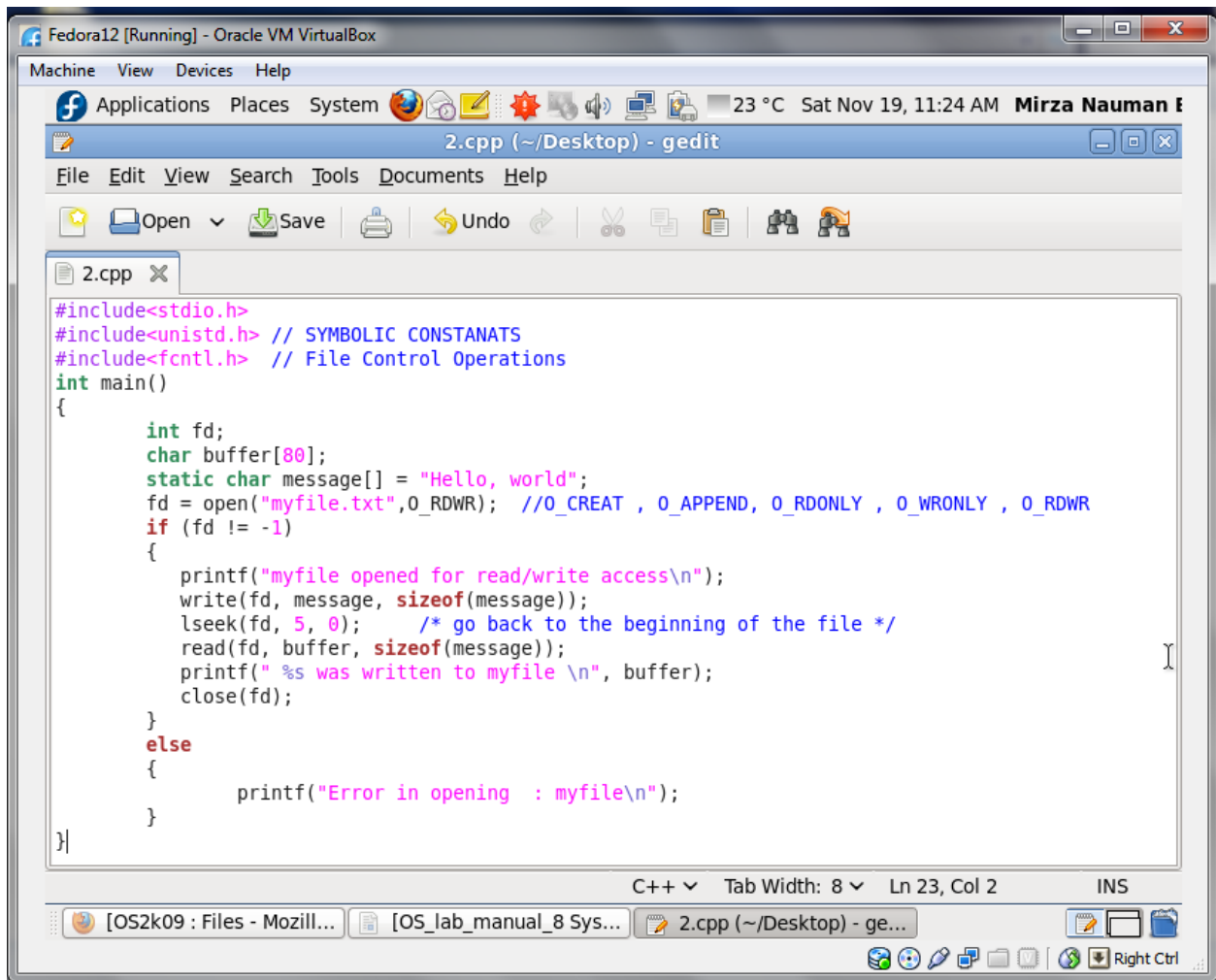
```
        printf(" %s was written to myfile \n", buffer);
```

```
        close (fd);
```

```
    }
```

```
}
```





```
#include<stdio.h>
#include<unistd.h> // SYMBOLIC CONSTANATS
#include<fcntl.h> // File Control Operations
int main()
{
    int fd;
    char buffer[80];
    static char message[] = "Hello, world";
    fd = open("myfile.txt",O_RDWR); //O_CREAT , O_APPEND, O_RDONLY , O_WRONLY , O_RDWR
    if (fd != -1)
    {
        printf("myfile opened for read/write access\n");
        write(fd, message, sizeof(message));
        lseek(fd, 5, 0); /* go back to the beginning of the file */
        read(fd, buffer, sizeof(message));
        printf(" %s was written to myfile \n", buffer);
        close(fd);
    }
    else
    {
        printf("Error in opening : myfile\n");
    }
}
```

fork()

When the fork system call is executed, a new process is created which consists of a copy of the address space of the parent.

The return code for fork is zero for the child process and the process identifier of child is returned to the parent process.

On success, both processes continue execution at the instruction after the fork call.

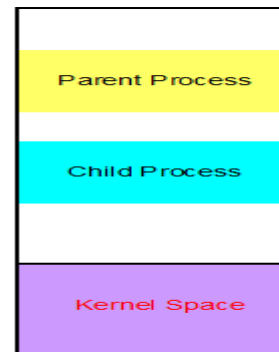
On failure, -1 is returned to the parent process.

fork()—Sample Code

```

main()
{
    int pid;
    ...
    pid = fork();
    if (pid == 0) {
        /* Code for child */
        ...
    }
    else {
        /* Code for parent */
        ...
    }
}

```

**Implementing fork system call using C program**

```

#include <sys/types.h>
main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
        printf("\n I'm the child process");
    else if (pid > 0)
        printf("\n I'm the parent process. My child pid is %d", pid);
    else
        perror("error in fork");
}

```

```

nauman@localhost:~/Desktop
File Edit View Terminal Help
[nauman@localhost Desktop]$ g++ -o fork_c forkcode.cpp
[nauman@localhost Desktop]$ ./fork_c
pid=6946
I'm the child process
pid=6945
I'm the parent process. My child pid is 6946
[nauman@localhost Desktop]$

```

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h> //Primitive System Data Types
int main()
{
    pid_t pid;
    pid = fork();
    //printf("PID=%d\n",pid);
    if (pid == 0)
    {
        printf("pid=%d\n",getpid());
        printf("I'm the child process\n");
    }
    else if (pid > 0)
    {
        printf("pid=%d\n",getpid());
        printf("I'm the parent process. My child pid is %d\n", pid);
    }
    else
    {
        perror("error in fork\n");
    }
}

```

wait()

- The wait system call suspends the calling process until one of its immediate children terminates.
- If the call is successful, the process ID of the terminating child is returned.
- **Zombie process**—a process that has terminated but whose exit status has not yet been received by its parent process.
 - the process will remain in the operating system's process table as a *zombie process*, indicating that it is not to be scheduled for further execution
 - But that it cannot be completely removed (and its process ID cannot be reused)

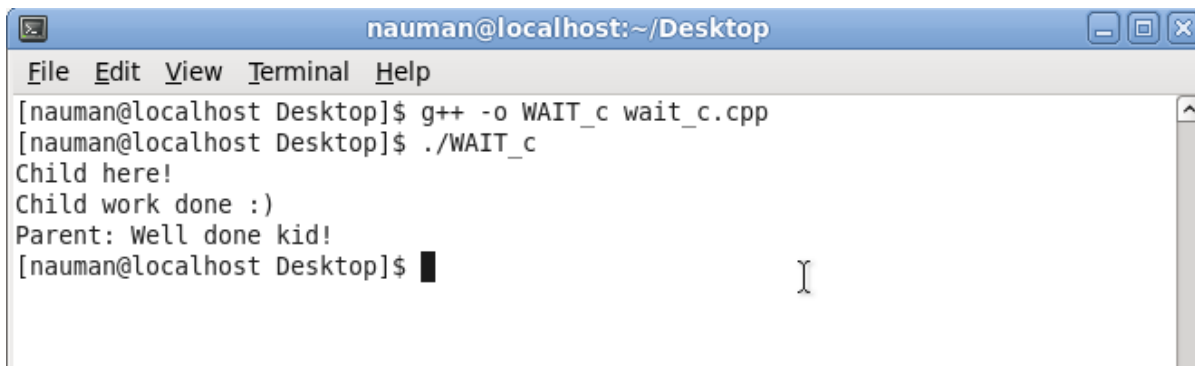
pid_t wait(int *status);

Where status is an integer value where the UNIX system stores the value returned by child process

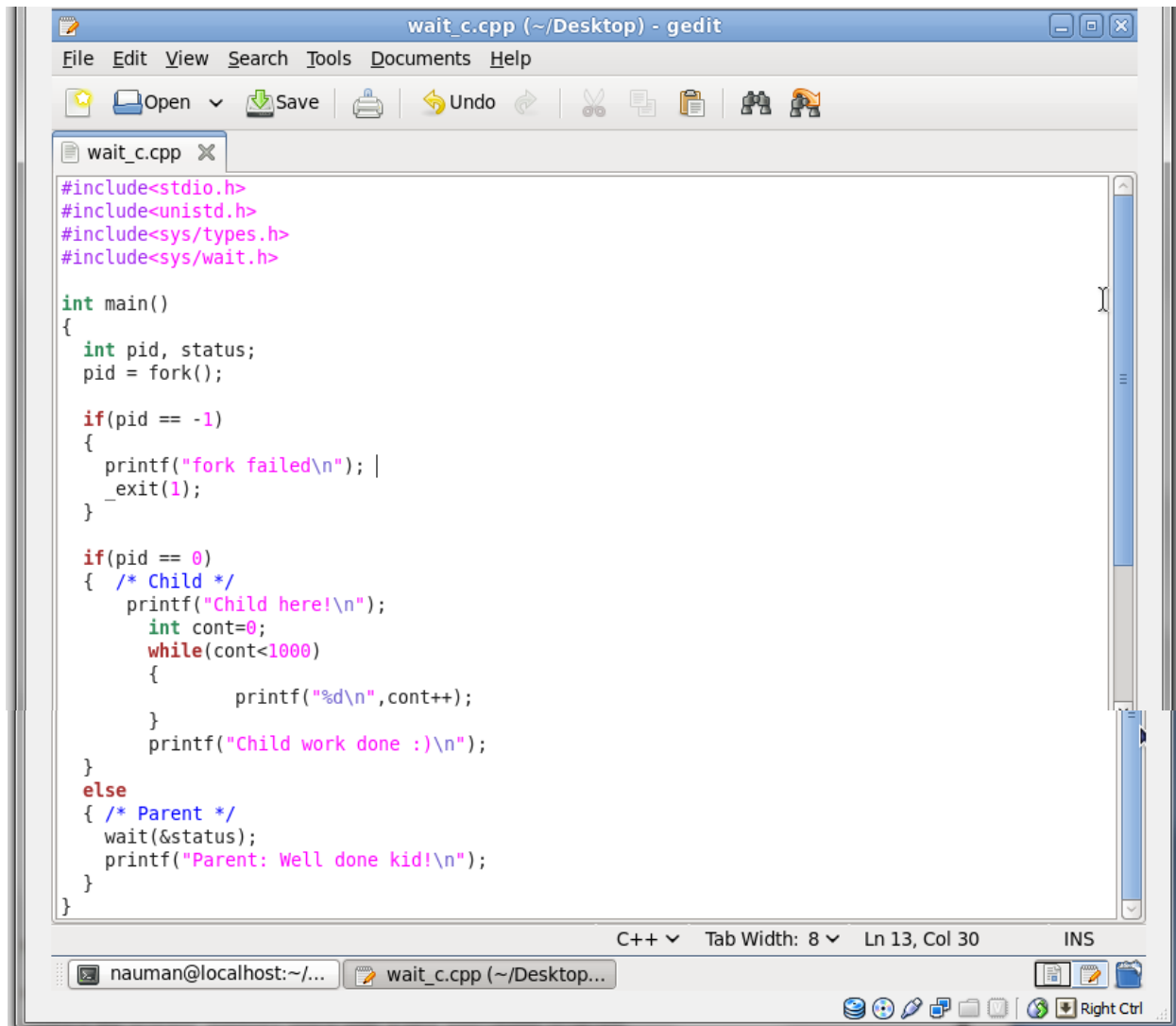
Implementing wait system call using C program

```
#include <stdio.h>
void main()
{
    int pid, status;
    pid = fork();

    if(pid == -1)
    {
        printf("fork failed\n");
        exit(1);
    }
    if(pid == 0)
    { /* Child */
        printf("Child here!\n");
    }
    else
    { /* Parent */
        wait(&status);
        printf("Well done kid!\n");
    }
}
```



```
nauman@localhost:~/Desktop
File Edit View Terminal Help
[nauman@localhost Desktop]$ g++ -o WAIT_c wait_c.cpp
[nauman@localhost Desktop]$ ./WAIT_c
Child here!
Child work done :)
Parent: Well done kid!
[nauman@localhost Desktop]$
```



The image shows a screenshot of a gedit editor window titled "wait_c.cpp (~/Desktop) - gedit". The window contains C++ code for a program that demonstrates the use of fork() and wait(). The code is as follows:

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

int main()
{
    int pid, status;
    pid = fork();

    if(pid == -1)
    {
        printf("fork failed\n"); |
        _exit(1);
    }

    if(pid == 0)
    { /* Child */
        printf("Child here!\n");
        int cont=0;
        while(cont<1000)
        {
            printf("%d\n",cont++);
        }
        printf("Child work done :)\n");
    }
    else
    { /* Parent */
        wait(&status);
        printf("Parent: Well done kid!\n");
    }
}
```

The editor interface includes a menu bar (File, Edit, View, Search, Tools, Documents, Help), a toolbar with icons for Open, Save, Undo, and other functions, and a status bar at the bottom showing "C++", "Tab Width: 8", "Ln 13, Col 30", and "INS". The terminal bar at the bottom shows the user "nauman@localhost" and the file path "~/Desktop...".

exec()

- Typically the exec system call is used after a fork system call by one of the two processes to replace the process' memory space with a new executable program.
- The new process image is constructed from an ordinary, executable file.
- There can be no return from a successful exec because the calling process image is overlaid by the new process image

Implementing exec system using C program

execl Takes the path name of an executable program (binary file) as its first argument. The rest of the arguments are a list of command line arguments to the new program (argv[]). The list is terminated with a null pointer:

execl("a.out", "a.out",NULL)

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf(" 1 \n");
    int pid = fork();
    if(pid==0)
        execl("/bin/ls", "ls", NULL);
    else if(pid>0)
        execl("/bin/pwd", "pwd", NULL);
}
```

```
#include <unistd.h>
```

```
void main()
{
printf(" 1 \n");
execl("/bin/lis", "lis", NULL);
}
```

Example 2

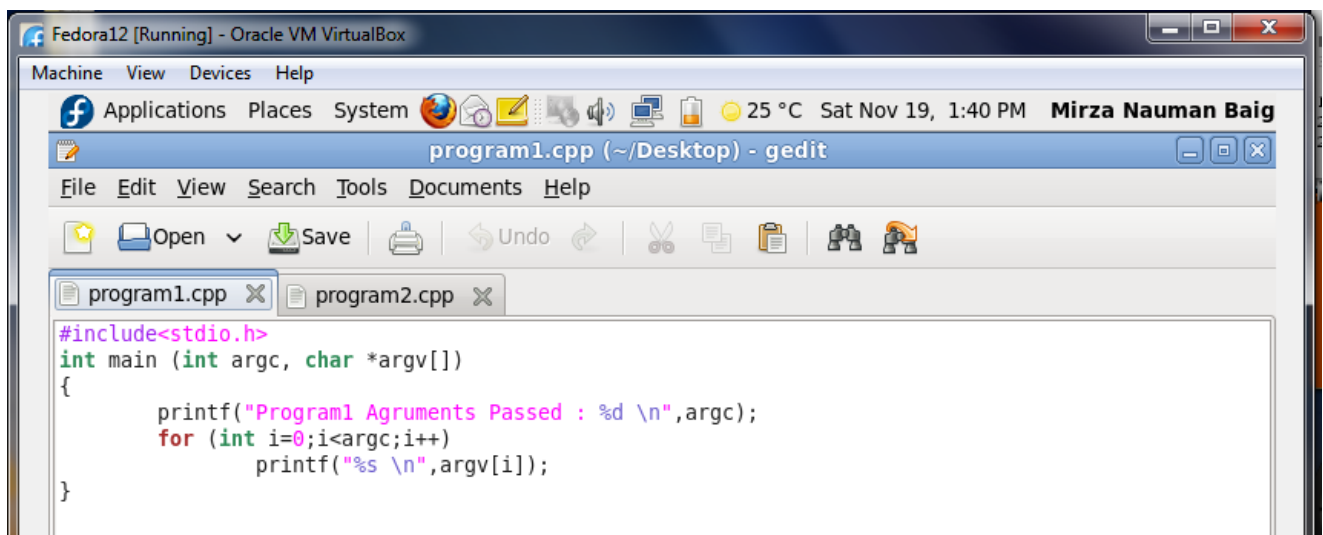
Step 1

Create a file with name prog2 ,compile it and run it.

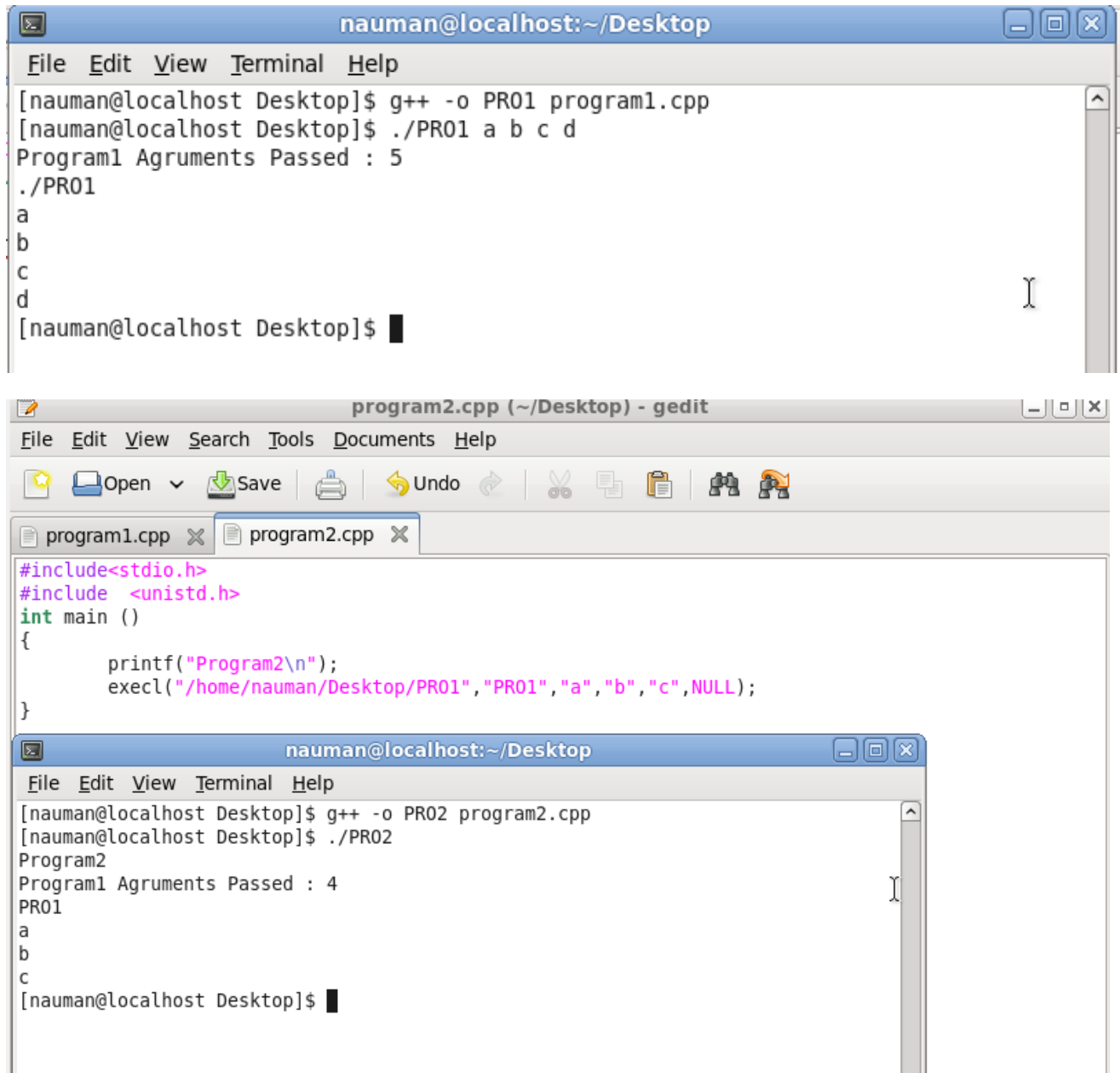
```
#include <stdio.h>
void main()
{
printf("2");
}
chmod 755 prog2.c
gcc -o prog2 prog2.c
```

Step 2

```
#include <stdio.h>
#include <unistd.h>
void main()
{
printf("1 \n");
execl("/home/user/prog2", "prog2", NULL);
}
```



```
Fedora12 [Running] - Oracle VM VirtualBox
Machine View Devices Help
Applications Places System 25 °C Sat Nov 19, 1:40 PM Mirza Nauman Baig
program1.cpp (~/Desktop) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
program1.cpp program2.cpp
#include<stdio.h>
int main (int argc, char *argv[])
{
    printf("Program1 Arguments Passed : %d \n",argc);
    for (int i=0;i<argc;i++)
        printf("%s \n",argv[i]);
}
```



The image shows a Linux desktop environment with two windows. The top window is a terminal titled "nauman@localhost:~/Desktop". It shows the following commands and output:

```
[nauman@localhost Desktop]$ g++ -o PR01 program1.cpp
[nauman@localhost Desktop]$ ./PR01 a b c d
Program1 Agruments Passed : 5
./PR01
a
b
c
d
[nauman@localhost Desktop]$
```

The middle window is a gedit editor titled "program2.cpp (~/.Desktop) - gedit". It shows the following code:

```
#include<stdio.h>
#include <unistd.h>
int main ()
{
    printf("Program2\n");
    execl("/home/nauman/Desktop/PR01", "PR01", "a", "b", "c", NULL);
}
```

The bottom window is another terminal titled "nauman@localhost:~/Desktop". It shows the following commands and output:

```
[nauman@localhost Desktop]$ g++ -o PR02 program2.cpp
[nauman@localhost Desktop]$ ./PR02
Program2
Program1 Agruments Passed : 4
PR01
a
b
c
[nauman@localhost Desktop]$
```

For more help: <http://linux.die.net/man/2/syscalls>