OPERATING SYSTEMS LAB

LAB # 7

System Calls

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()

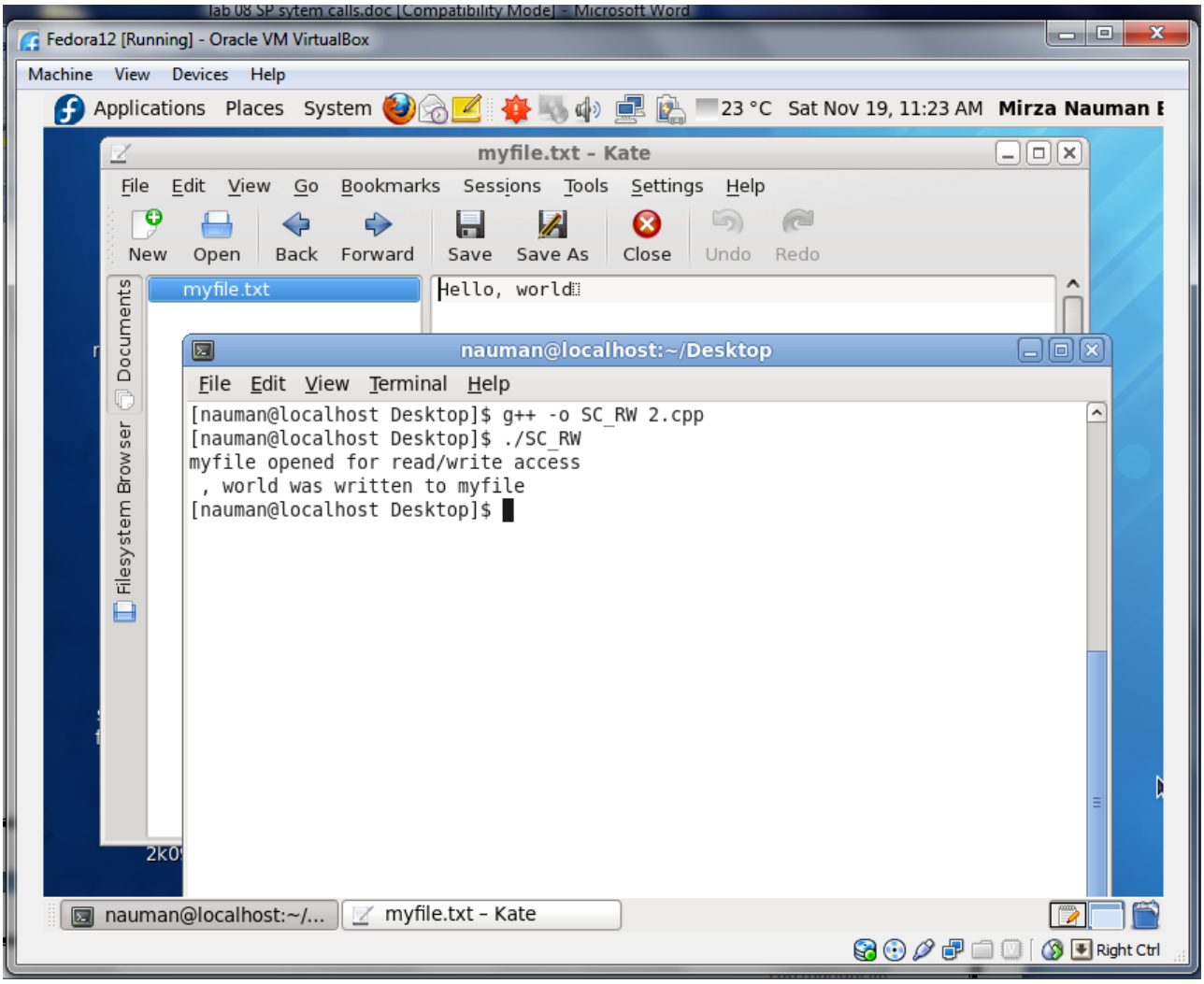
To close a channel, use the close() system call. The prototype for the close() system call is:

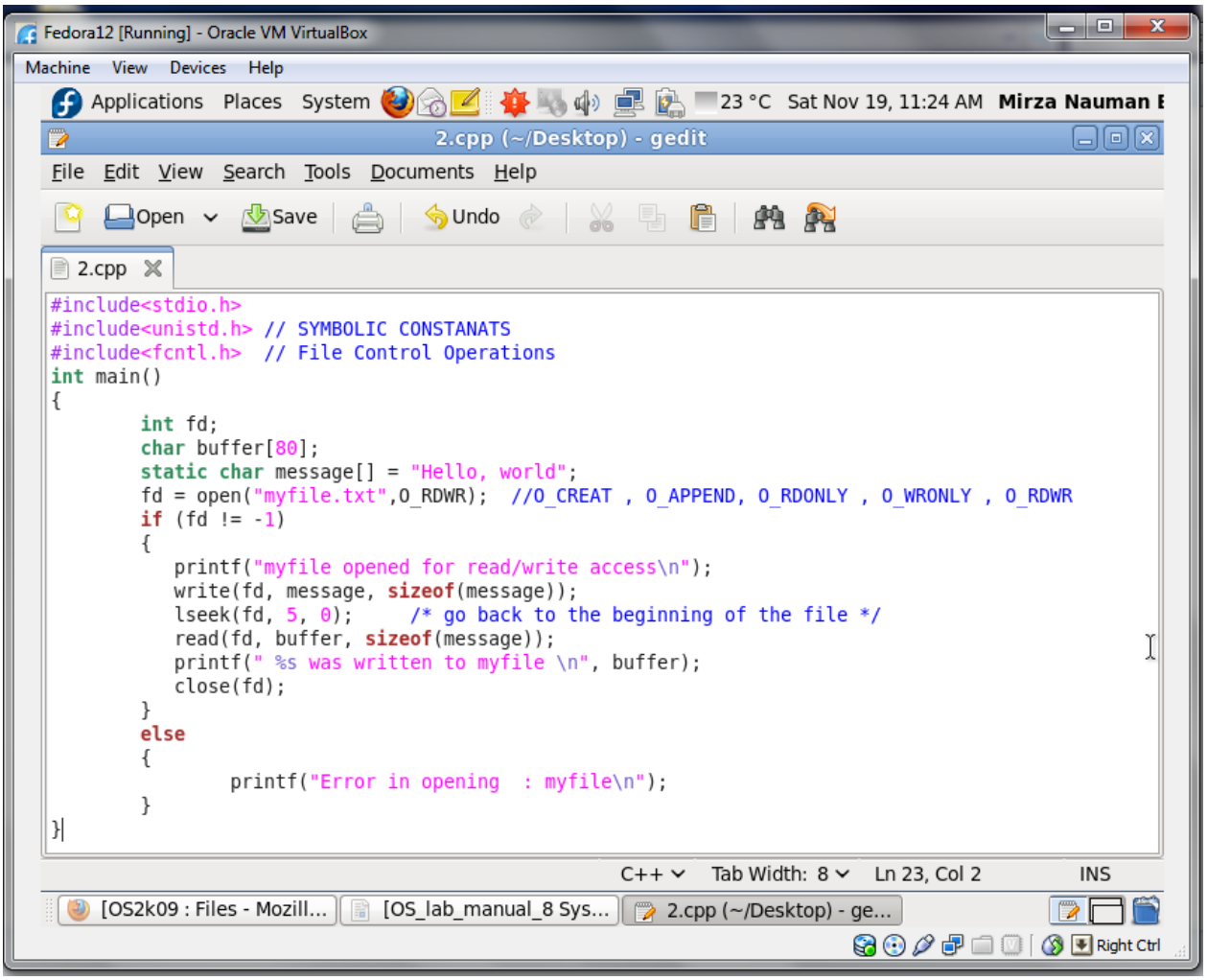
int close(file_descriptor)

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);
      }
}
```


Fedora12 [Running] - Oracle VM VirtualBox
Machine View Devices Help
Applications Places System
23 °C Sat Nov 19, 11:23 AM Mirza Nauman
myfile.txt - Kate
File Edit View Go Bookmarks Sessions Tools Settings Help
New Open Back Forward Save Save As Close Undo Redo
Documents
Filesystem Browser
myfile.txt
Hello, world
nauman@localhost:~/Desktop
File Edit View Terminal Help
[nauman@localhost Desktop]$ g++ -o SC_RW 2.cpp
[nauman@localhost Desktop]$ ./SC_RW
myfile opened for read/write access
, world was written to myfile
[nauman@localhost Desktop]$
nauman@localhost:~/...
myfile.txt - Kate
Right Ctrl



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.

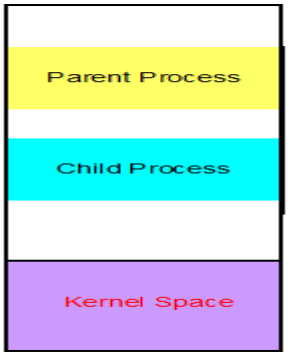
The return code for fork is zero for the child process and the process identifier of child is returned to the parent process.

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

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

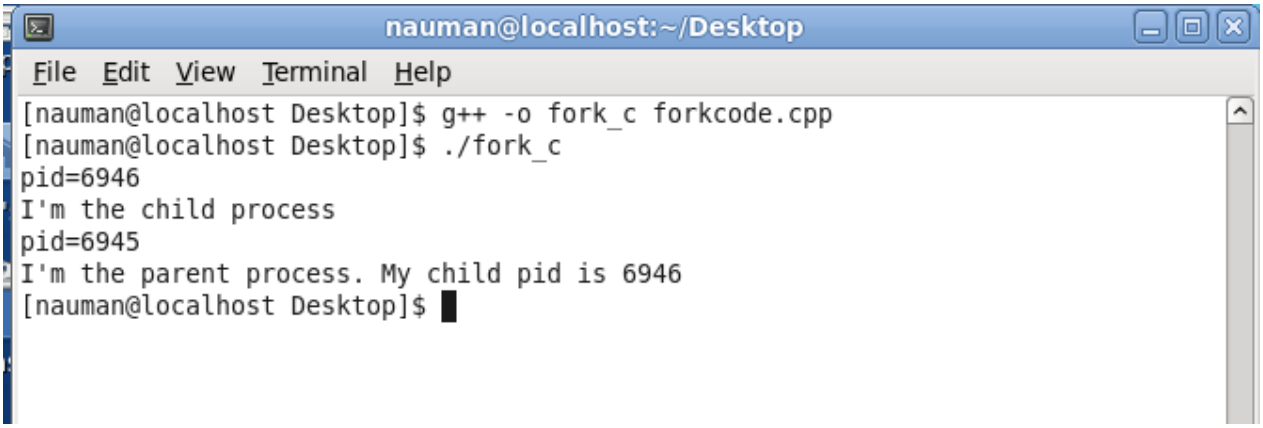
fork()—Sample Code

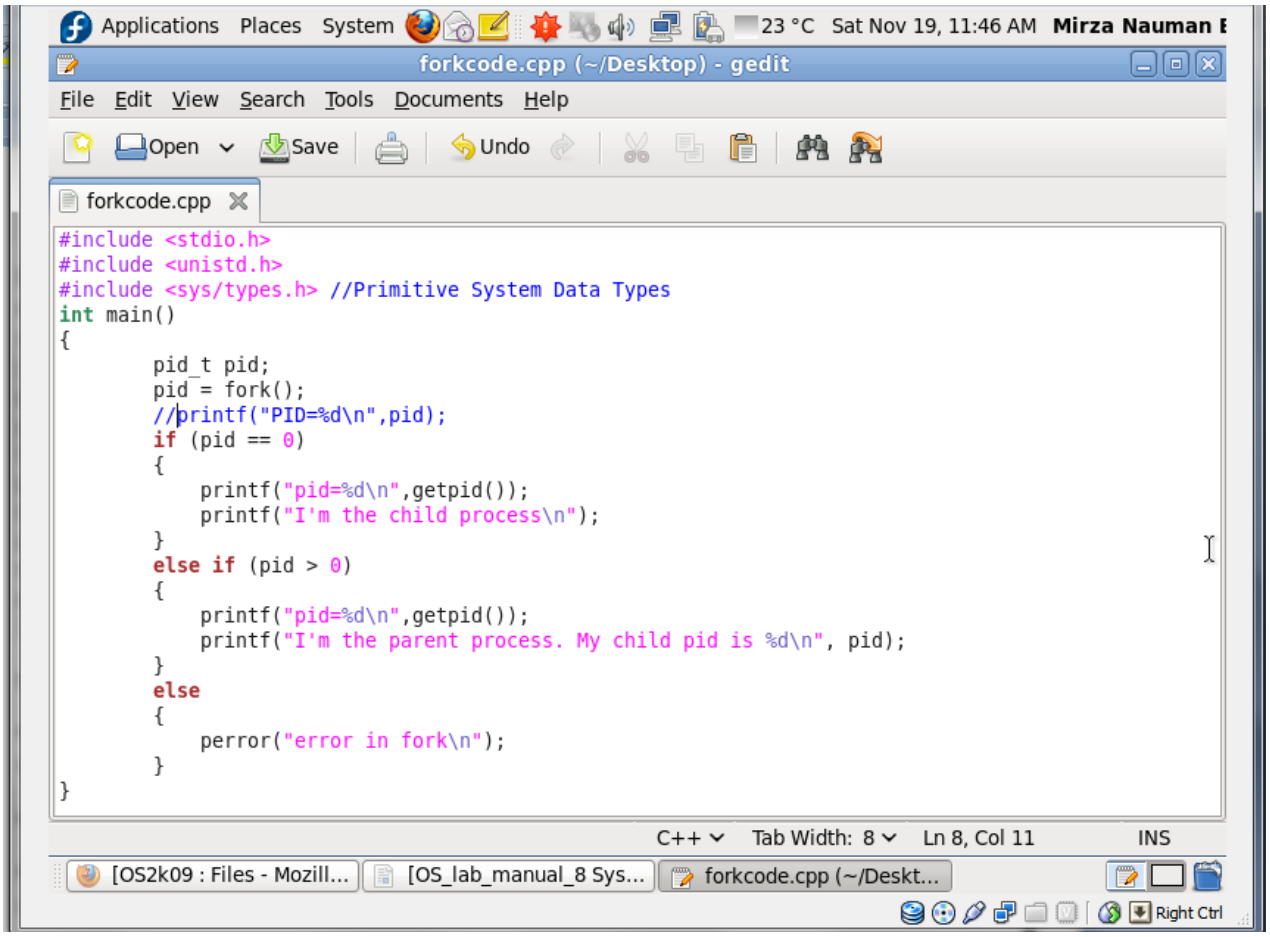
```
main()
{
    int pid;

    ...
    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");
}
```





wait()

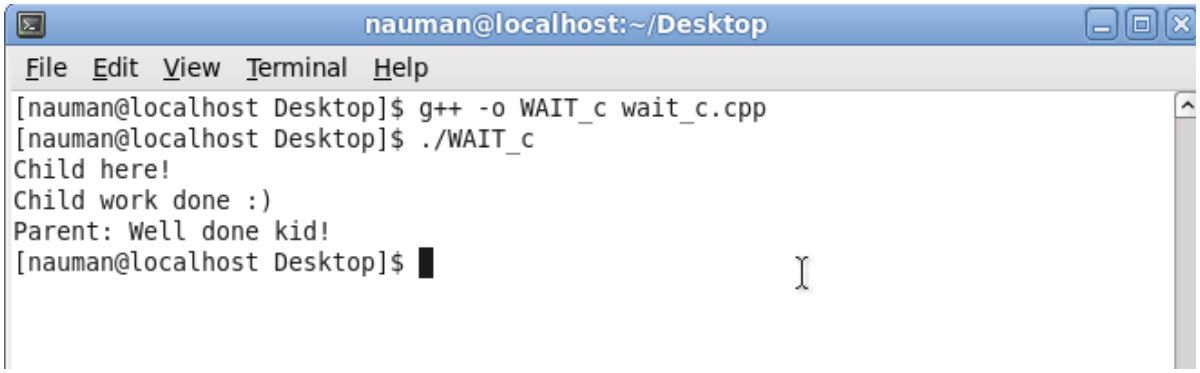
- The wait system call suspends the calling process until one of its immediate children terminates.
- If the call is successful, the process ID of the terminating child is returned.
- **Zombie process**—a process that has terminated but whose exit status has not yet been received by its parent process.
    - 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)

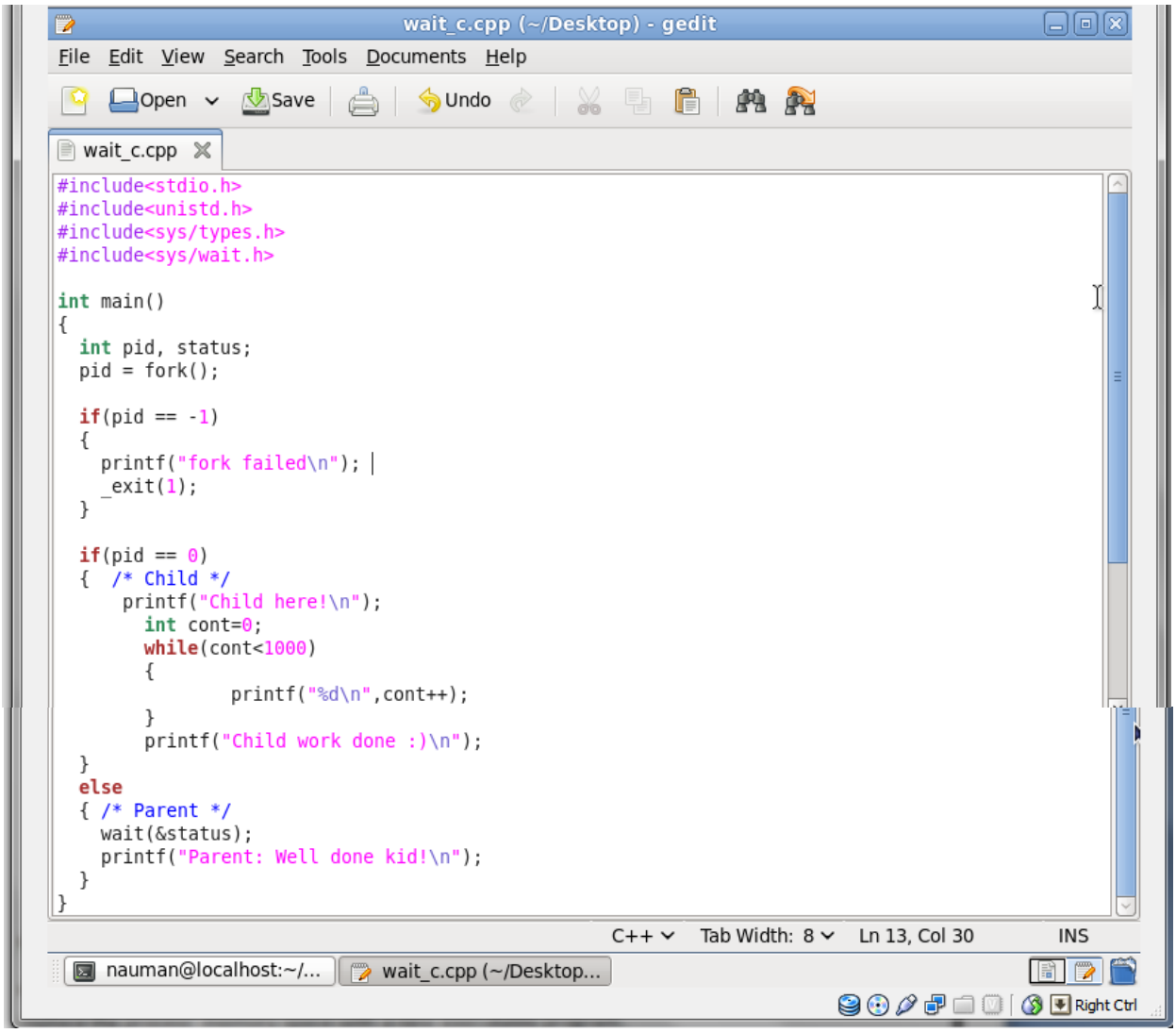
pid_t wait(int *_status_);

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");
       }
      }
```




wait_c.cpp (~/Desktop) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
wait_c.cpp
#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");
}
}
C++ Tab Width: 8 Ln 13, Col 30 INS
nauman@localhost:~/...
wait_c.cpp (~/Desktop...
Right Ctrl

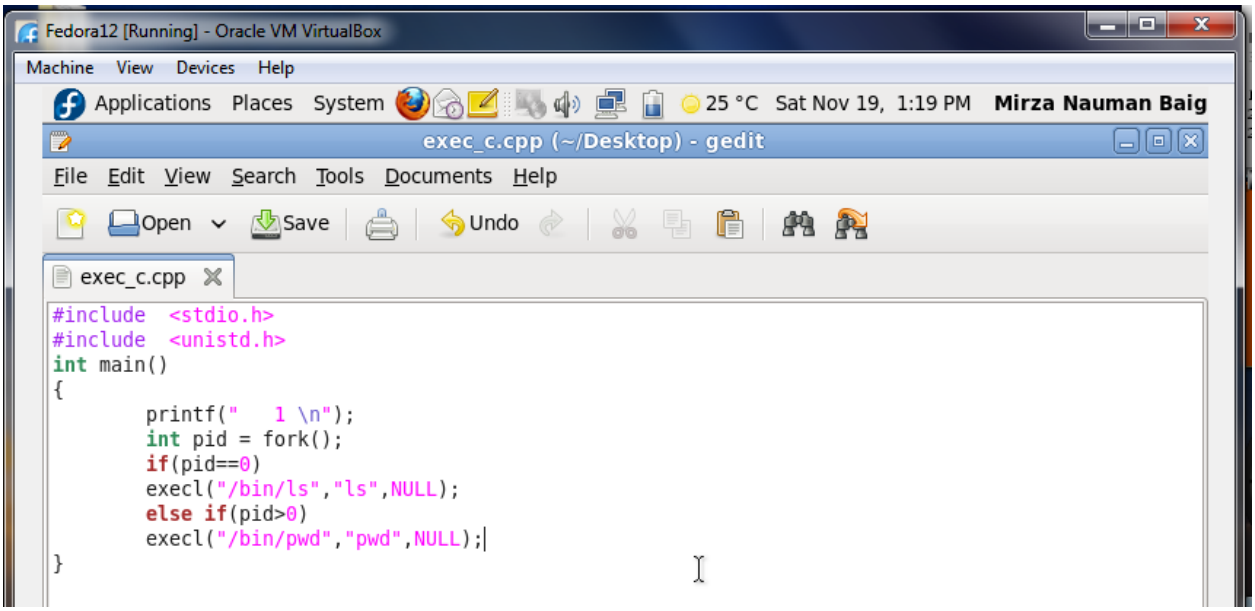
exec()

- 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 <unistd.h>

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

Example 2

Step 1

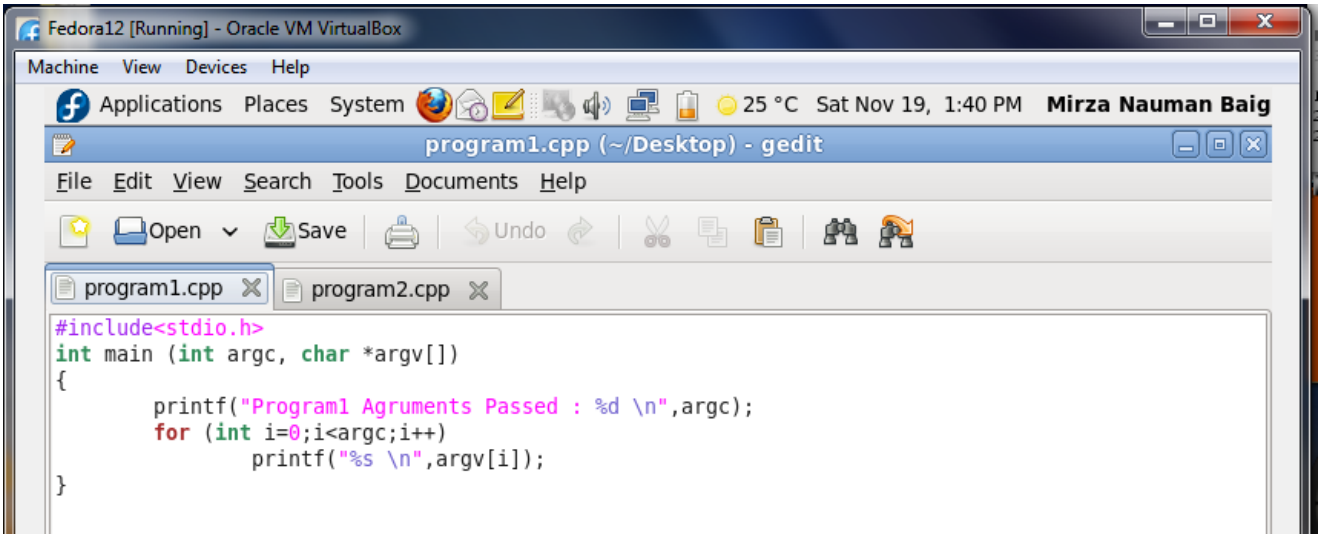
Create a file with name prog2 ,compile it and run it.

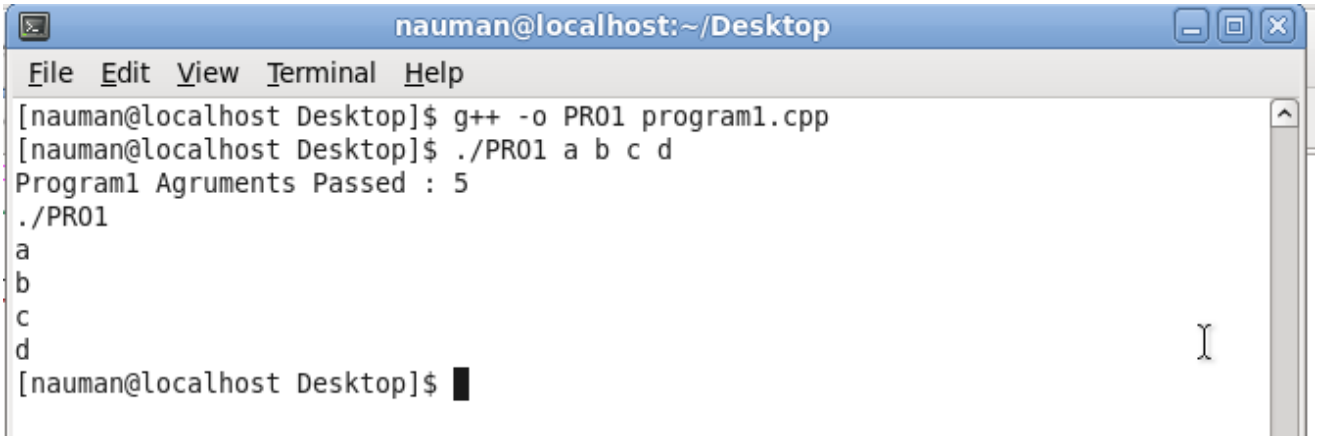
```
#include <stdio.h>
void main()
{
printf("2");
}
```

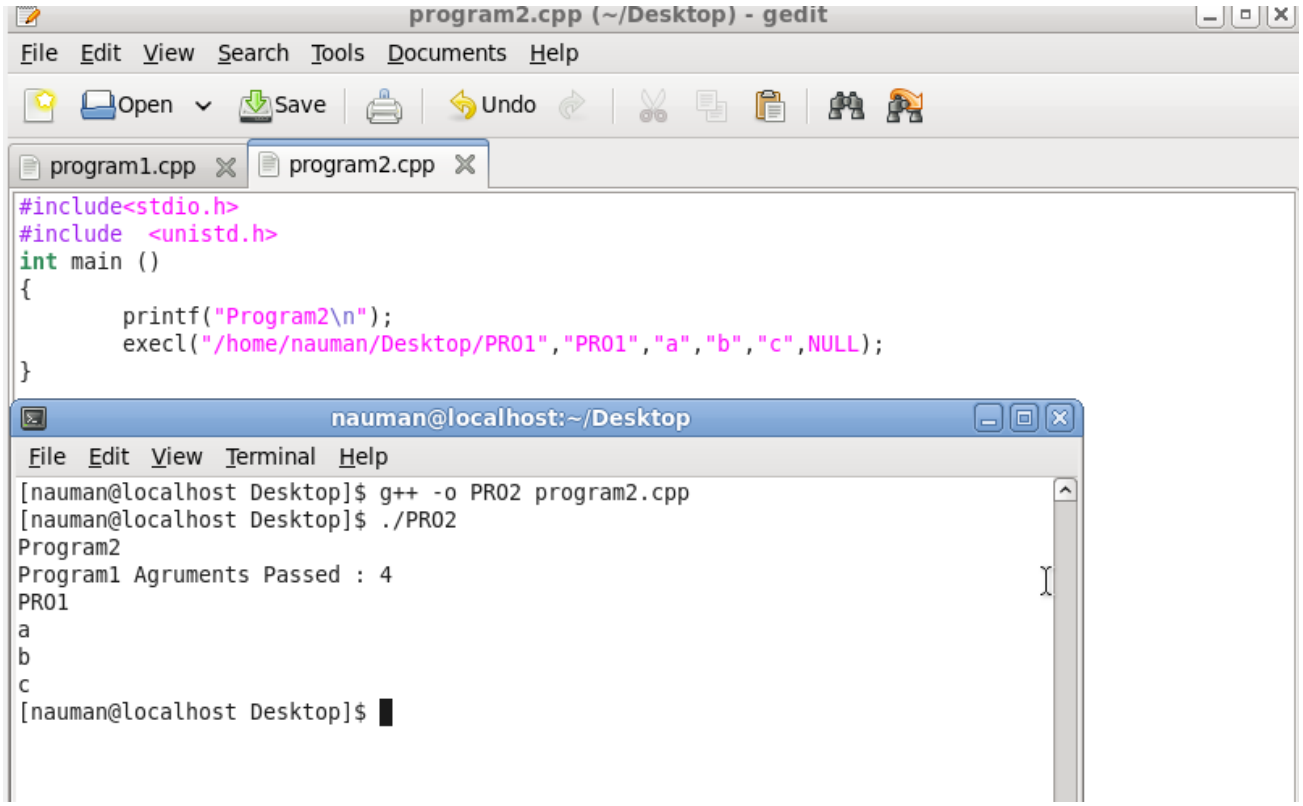
```
chmod 755 prog2.c
gcc –o prog2 prog2.c
```

Step 2

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







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