西北工業大學

NORTHWESTERN POLYTECHNICAL UNIVERSITY

Lab I Report

Linux Shell Commands、C programming

Student ID: 2018380039

Student Name: Dikshya Kafle

Report Subject: Computer OS Experiment - Laboratory I

Teacher Name: Tianhai Zhao

COMPUTER OPERATING SYSTEM

TABLE OF CONTENTS

1. Objectives

- To familiar with Linux Operating system.
- To practice the usage of Linux shell commands.
- To practice with the creation of a Makefile.
- To get reacquainted with C programming.
- To practice your code reading skills.

2. Equipment

- VirtualBox with Ubuntu Linux 20.04.

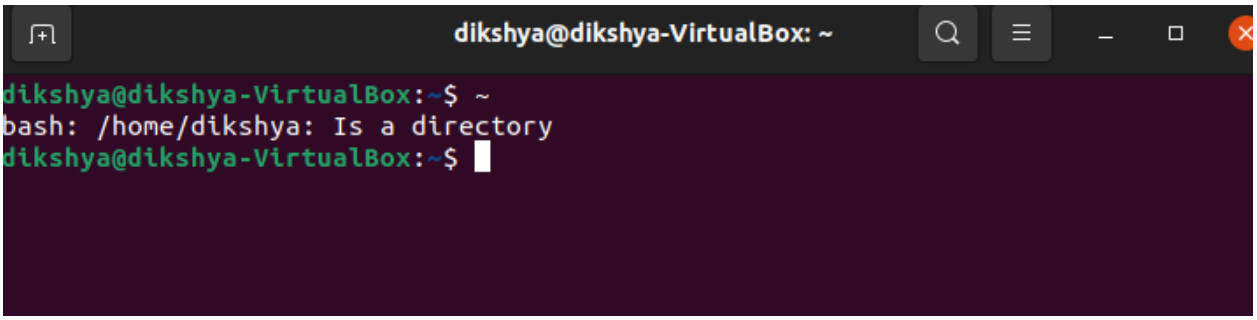
3. Methodlogy

1. Practise and answer the questions

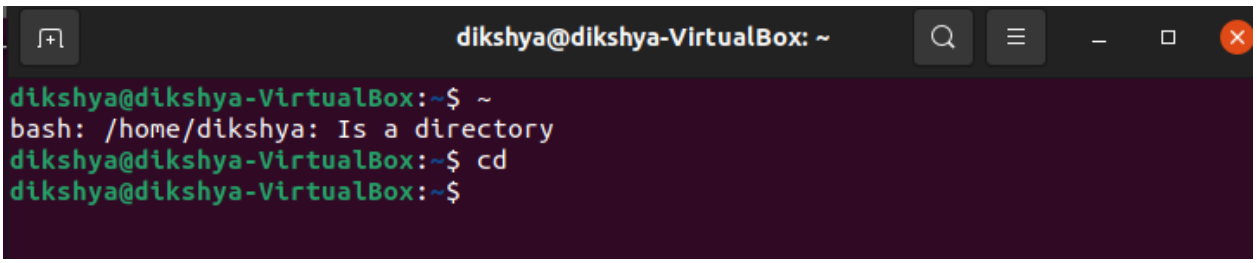
i. How do you find your home directory quickly? And change the directory to your home.

Answer:

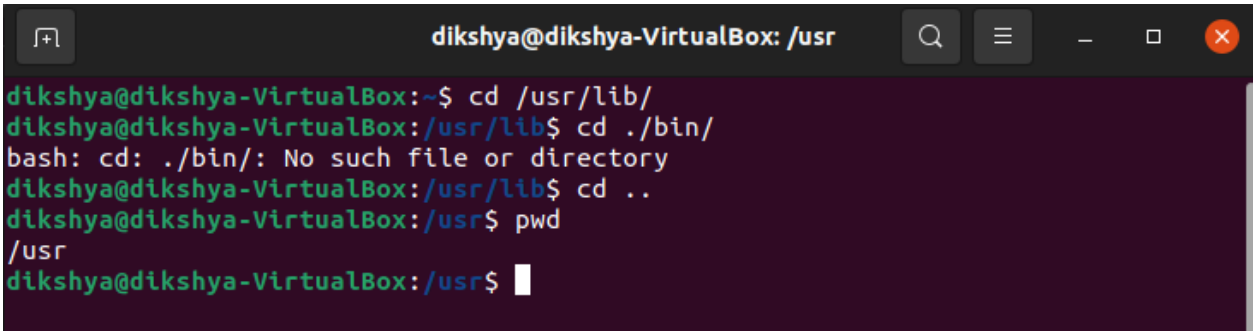
- In linux, the tilde (~) symbol stands for the home directory. In my case my user name is dikshya, so the tilde (~) stands for /home/*dikshya*



- To directly change from the directory to the home directory we simply use the "cd" command.



ii. Type these commands "cd /usr/lib/", "cd ./bin/", "cd ..", and "pwd". Give the current directory which you are located. What are the meanings of . and .. ?

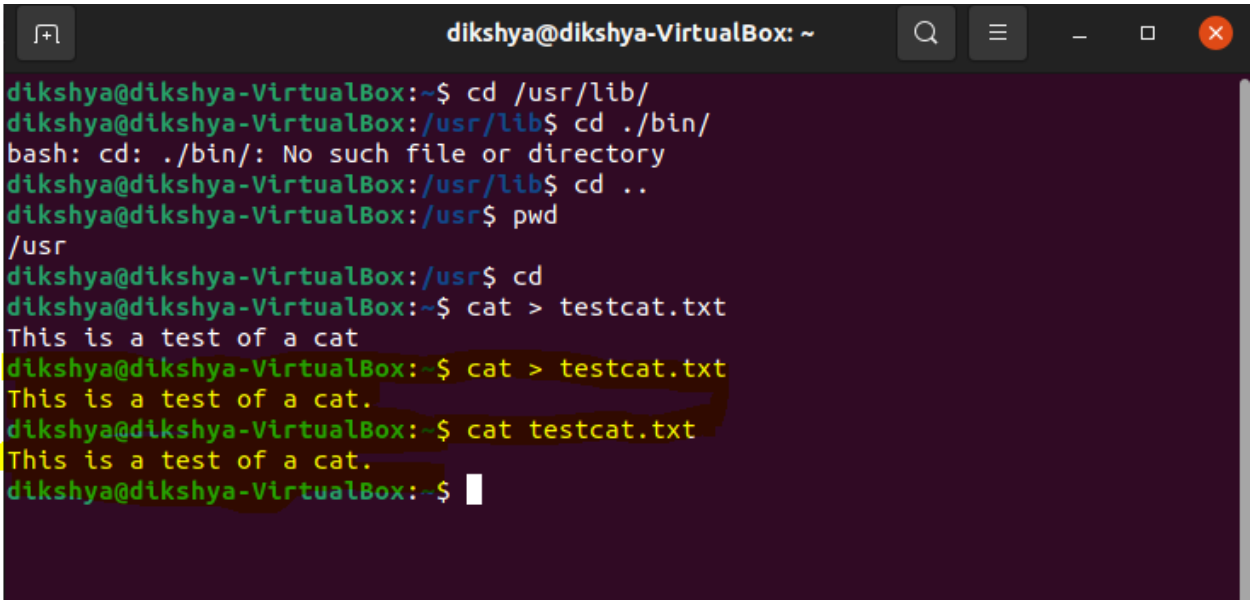


The current directory is "/usr". The single dot "." means the current directory which actually is a hard link to its containing directory. The double dot ".." means one step back i.e. the parent directory, which is the only directory where that directory is referenced from.

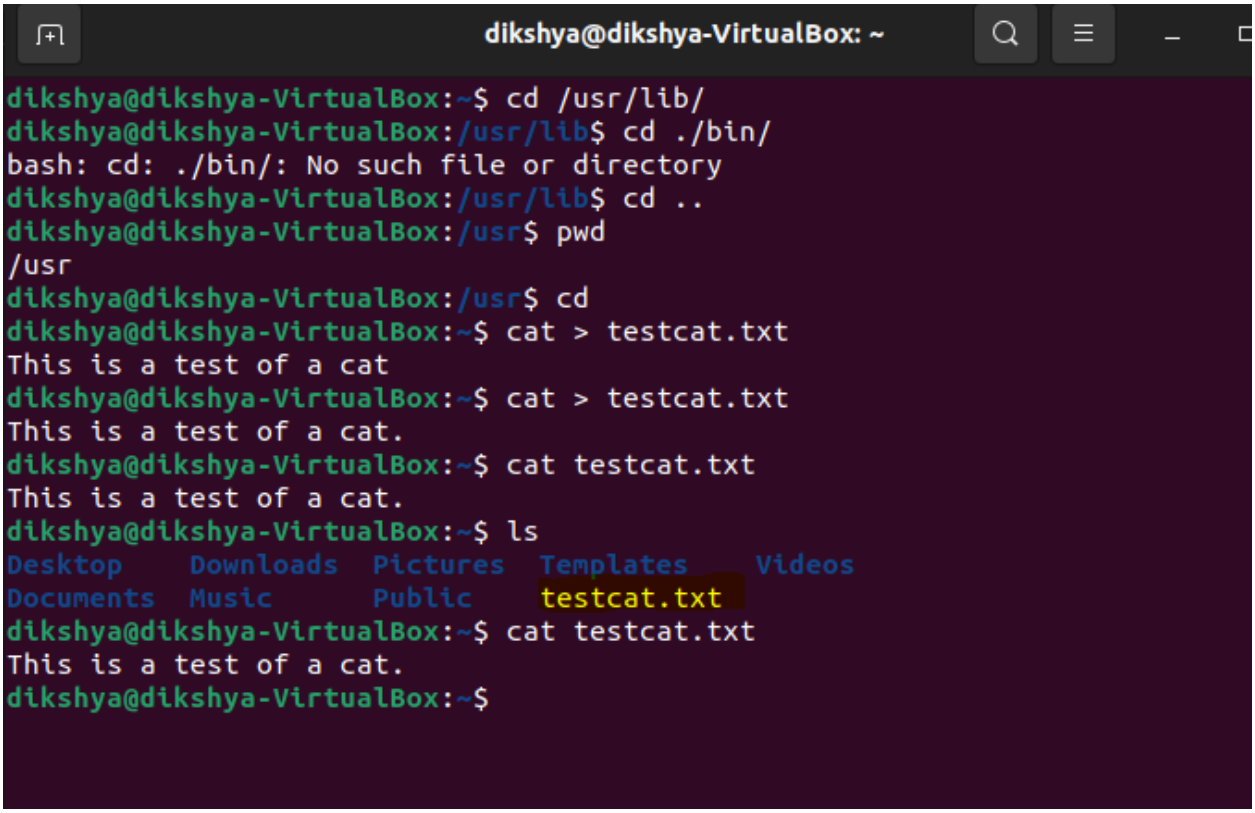
iii. Type "cd" to go back to your home directory. Type "cat > testcat.txt" in the command line. After pressing return, type the following line of text "This is a test of cat.", and then press crtl-d. Type "cat testcat.txt" again. What do you see?

Answer:

- The command "cat > testcat.txt" creates a testcat.txt file (in the current directory). After this text file is created we need to type the text we want to include in the text file, once we are done we press CRTL+D to save the file. Then, using the command "cat testcat.txt" will display the text file content.



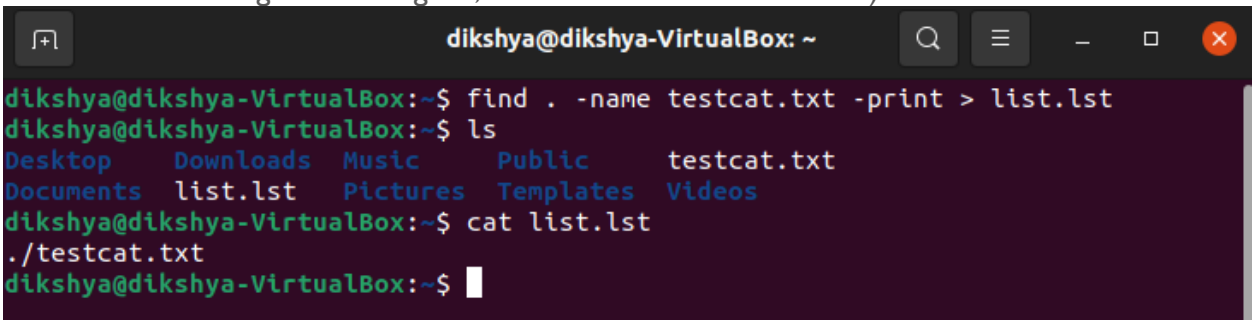
If we check the home directory, we can see the text file is there!



iv. Type "find . -name testcat.txt –print > list.lst" in the command line. You will find a file "list.lst" in your current directory. Use cat commands to show its contents. What is the result? What is the meaning of > list.lst?

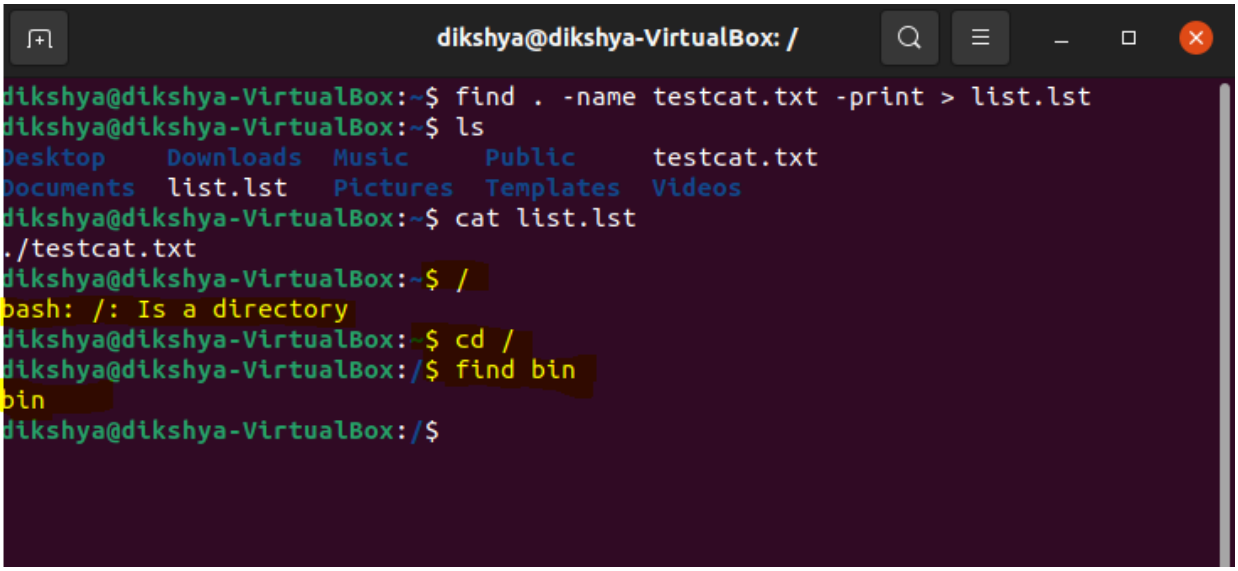
Answer:

- The command "find . -name testcat.txt -print" is used to find the file testcat.txt in the current directory and the -print option is meant to print the path name of the file found. The part "> list.lst" means redirect the output to a newly created file called list.lst (text file containing a list of data). If the file already exists, replace/override it. (If the file existed already and we only want to append the result instead of overriding the existing file, we could use >> instead of >)



v. Go to "/" directory and type the command to find the bin directory.

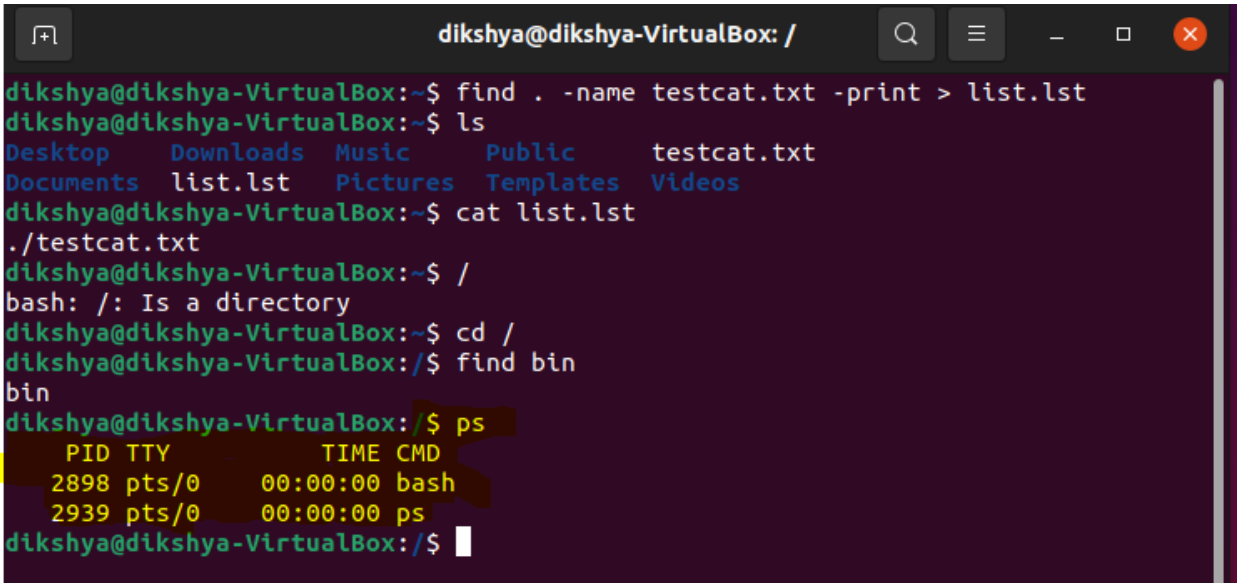
Answer:



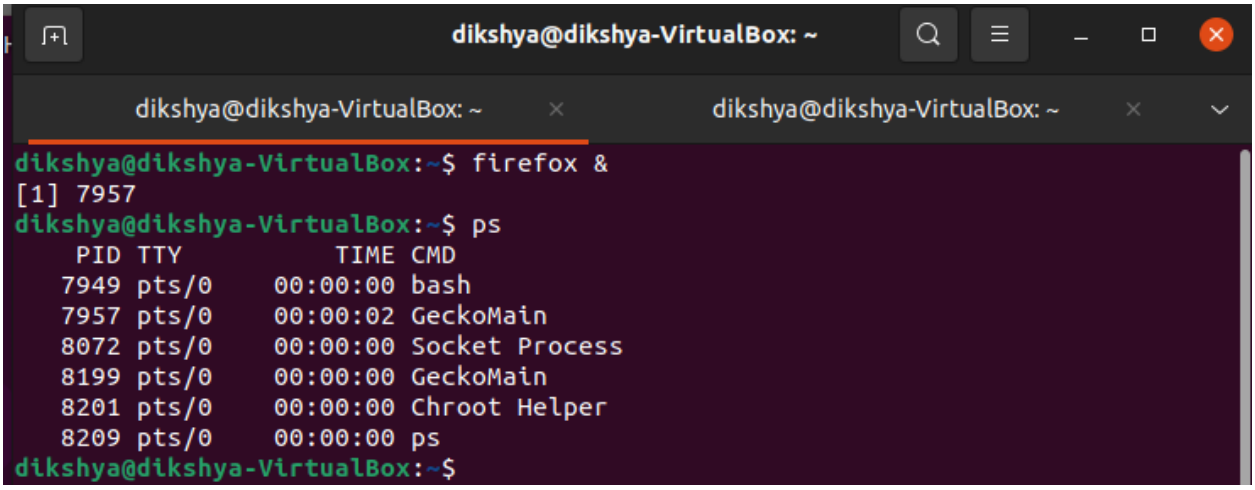
vi. Open two terminals on Linux. In each terminal, type the command "ps". Give the process number (PID) of the bash process in both terminals. Why they are different?

Answer:

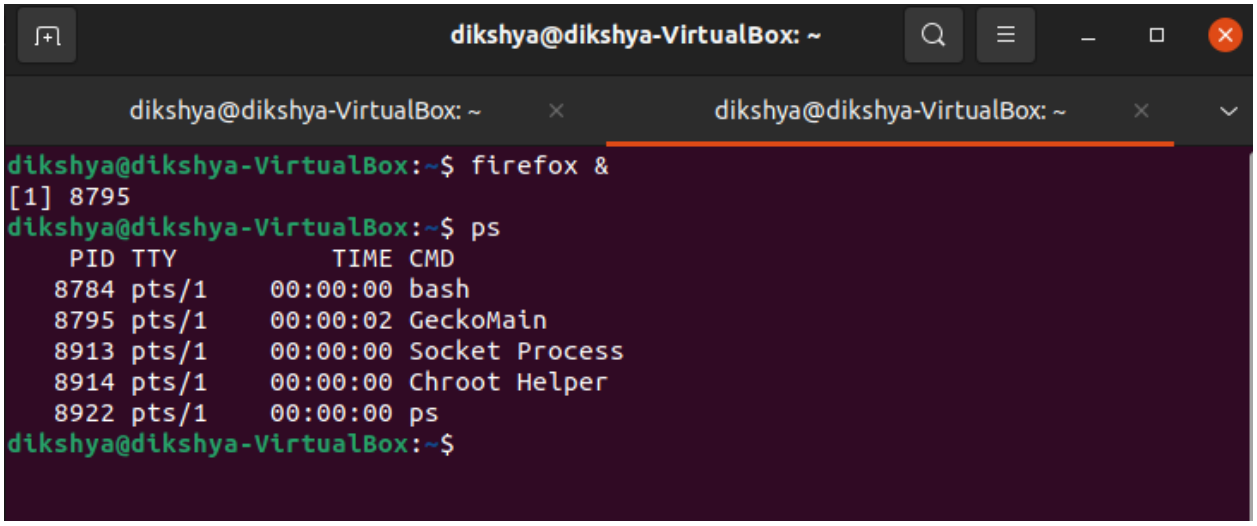
- Each bash process is a different process eventhough the program is the same. They are both two different instances of the same program. There for they are two separate process and this is why they have different process numbers (PID)



vii. Start the Fire-Fox web browser by typing "firefox &" at the command line and type "ps" in the terminal. Can you see the process of Fire-Fox web browser? Find the parent process and child process.



Each terminal has its own bash process hence the two terminals have different PID.



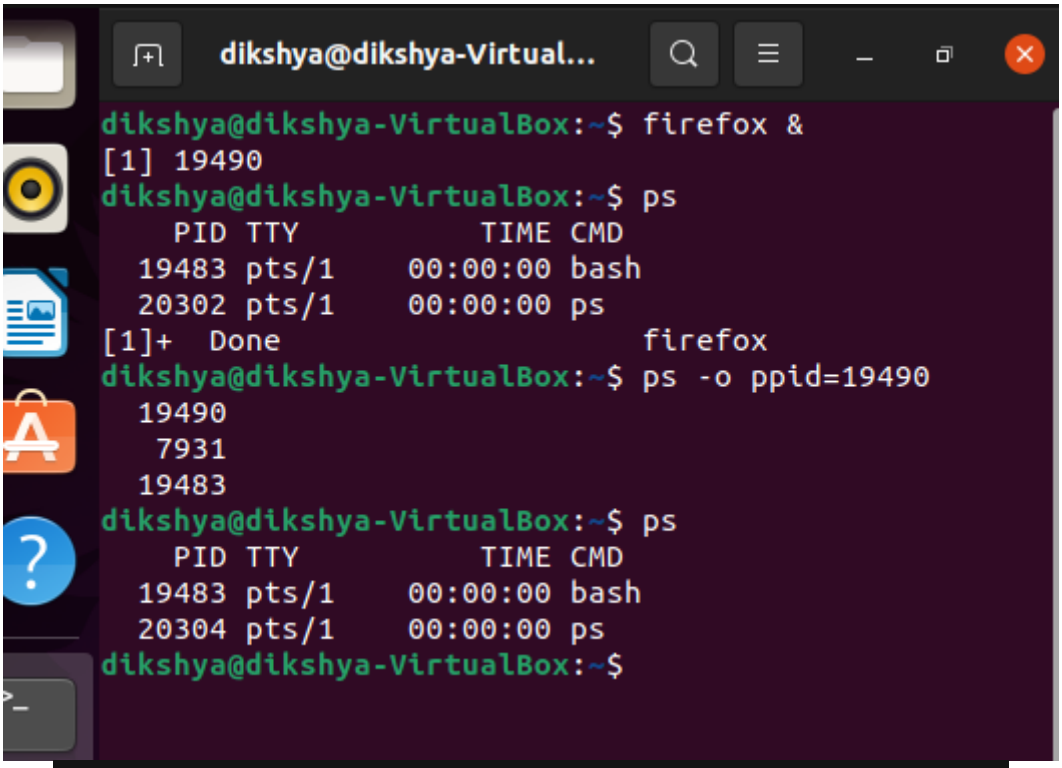
Yes I can see the process of firefox, the parent process is the second one while the child process is web content.

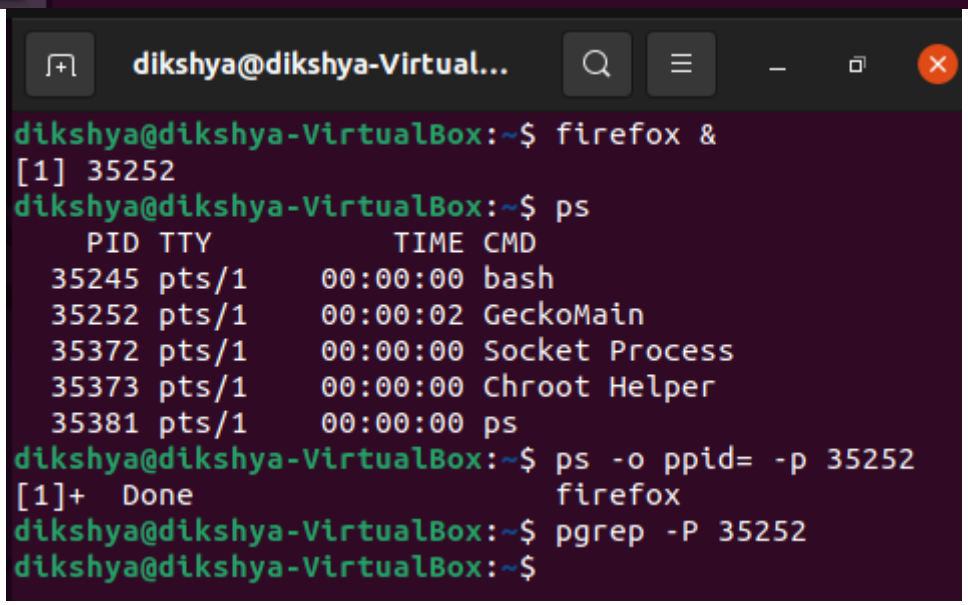
Answer:

- To run firefox: we use the command "firefox &". The command return firefox process id (in this case it is 19490) and we can also see it by using the "ps" command the checking the PID corresponding to the process name under the CMD column.

To find the parent process of the firefox process: use *"ps -o ppid= -p 19490"* . The parent process pid is "19483" which is the bash in this case (verify in the result of the ps command)

To find the child processes of the firefox process: use *"pgrep -P 19490"*. The child processes are "19490", "7931" and "19483". These, respectively, correspond to the process names "Privileged Cont", "WebExtensions" and "Web Content".

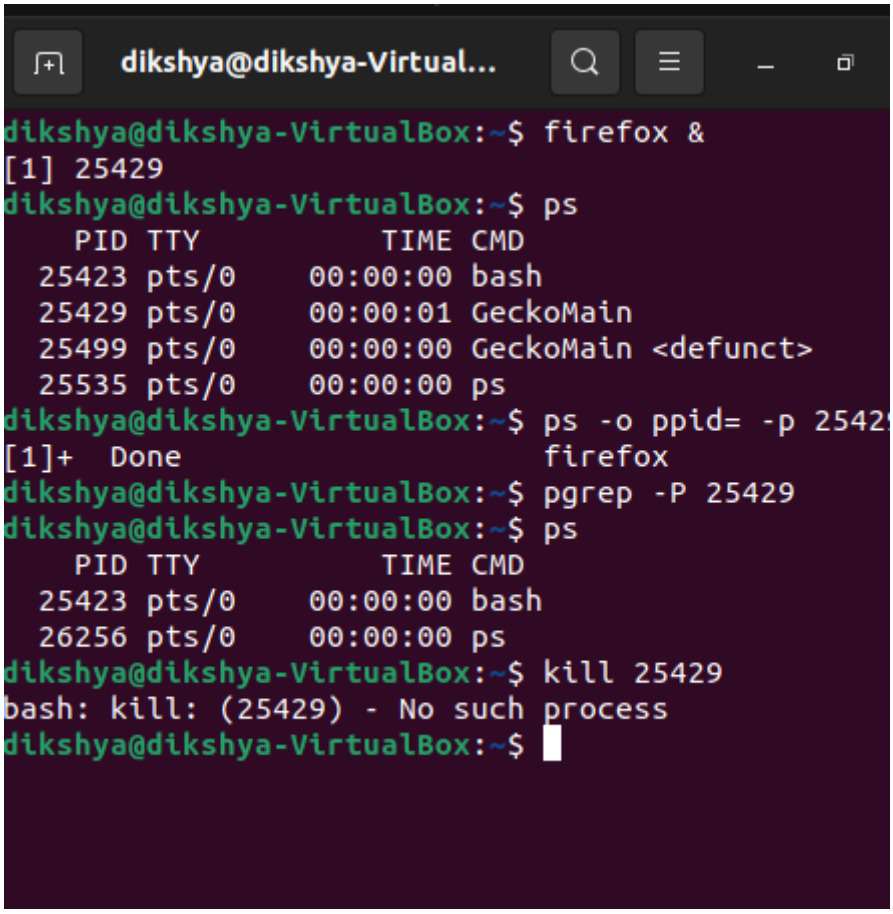


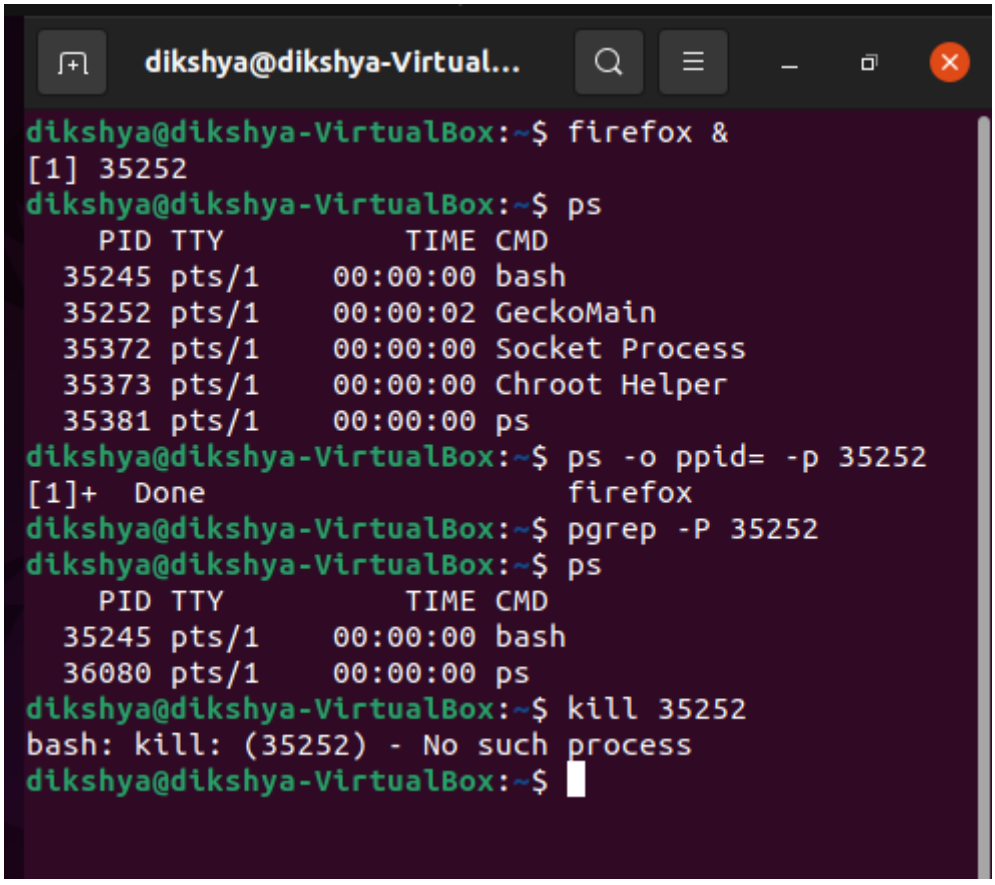


viii. Use the command "kill" to kill the processes of Fire-Fox. Give the command you have used. What did you see after killing their processes?

Answer:

- To kill the firefox process I used the command "kill 35252". This has resulted in channel error and I had to click CRTL+C to stop it.





2. **Now you are ready to write a program that reveals its own executing structure. The first.c provides a rather complete skeleton. You will need to modify it to get the**

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

/* A statically allocated variable */
int var1;

int my_func(int i);

/* A statically allocated, pre-initialized variable */
volatile int stuff = 7;

int main(int argc, char *argv[]) {
   /* A stack allocated variable */
   volatile int i = 0;

   /* Dynamically allocate some stuff */
   volatile char *buf1 = malloc(10);
   /* ... and some more stuff */
   volatile char *buf2 = malloc(10);

   my_func (3);
   return 0;
}
```

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3       /* A statically allocated variable */
4       int var1;
5       int my_func(int i){
6           var1=i;
7          return 0;
8       }
9       /* A statically allocated, pre-initialized variable */
10      volatile int stuff = 7;
(gdb) list
11      int main(int argc, char *argv[])
12      {                                       /* A stack allocated variable */
13          volatile int i = 0;                 /* Dynamically allocate some stuff */
14          volatile char *buf1 = (char*)malloc(10); /* ... and some more stuff */
15          volatile char *buf2 = (char*)malloc(10);
16          my_func(3);
17          return 0;
18      }
```

after running the gdb first we list the programme
Using the list command
And

```
18      }
(gdb) b 13
Breakpoint 1 at 0x670: file test.c, line 13.
(gdb) r
Starting program: /home/sobit/a.out
```

Set the breakpoint in the line 13 to example the program

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe0d8) at test.c:13
13          volatile int i = 0;              /* Dynamically allocate some s
(gdb) disassemble
Dump of assembler code for function main(int, char**):
   0x0000555555554661 <+0>:     push   %rbp
   0x0000555555554662 <+1>:     mov    %rsp,%rbp
   0x0000555555554665 <+4>:     sub    $0x30,%rsp
   0x0000555555554669 <+8>:     mov    %edi,-0x24(%rbp)
   0x000055555555466c <+11>:    mov    %rsi,-0x30(%rbp)
=> 0x0000555555554670 <+15>:    movl   $0x0,-0x14(%rbp)
   0x0000555555554677 <+22>:    mov    $0xa,%edi
   0x000055555555467c <+27>:    callq  0x555555554520 <malloc@plt>
   0x0000555555554681 <+32>:    mov    %rax,-0x10(%rbp)
   0x0000555555554685 <+36>:    mov    $0xa,%edi
   0x000055555555468a <+41>:    callq  0x555555554520 <malloc@plt>
   0x000055555555468f <+46>:    mov    %rax,-0x8(%rbp)
   0x0000555555554693 <+50>:    mov    $0x3,%edi
   0x0000555555554698 <+55>:    callq  0x55555555464a <my_func(int)>
   0x000055555555469d <+60>:    mov    $0x0,%eax
   0x00005555555546a2 <+65>:    leaveq
   0x00005555555546a3 <+66>:    retq
End of assembler dump.
```

So we show the stack content

```
(gdb) x/32x $rsp
0x7fffffffdfc0: 0xffffe0d8      0x00007fff      0x00000000      0x00000001
0x7fffffffdfd0: 0x555546b0      0x00005555      0x55554540      0x00005555
0x7fffffffdfe0: 0xffffe0d0      0x00007fff      0x00000000      0x00000000
0x7fffffffdff0: 0x555546b0      0x00005555      0xf7a03bf7      0x00007fff
0x7fffffffe000: 0x00000001      0x00000000      0xffffe0d8      0x00007fff
0x7fffffffe010: 0x00008000      0x00000001      0x55554661      0x00005555
0x7fffffffe020: 0x00000000      0x00000000      0x77894d06      0xec019f6d
```

Than increment the instruction by using n command
After that we again print the stack value

```
(gdb) print argv
$1 = (char **) 0x7fffffffe0d8
(gdb) prtin argv[0]
Undefined command: "prtin".  Try "help".
(gdb) print argv[0]
$2 = 0x7fffffffe40d "/home/sobit/a.out"
(gdb)
```

```
(gdb) n
14          volatile char *buf1 = (char*)malloc(10); /* ... and some more stuff */
(gdb) x/32x $rsp
0x7fffffffdfc0: 0xffffe0d8      0x00007fff      0x00000000      0x00000001
0x7fffffffdfd0: 0x555546b0      0x00005555      0x55554540      0x00000000
0x7fffffffdfe0: 0xffffe0d0      0x00007fff      0x00000000      0x00000000
0x7fffffffdff0: 0x555546b0      0x00005555      0xf7a03bf7      0x00007fff
0x7fffffffe000: 0x00000001      0x00000000      0xffffe0d8      0x00007fff
0x7fffffffe010: 0x00008000      0x00000001      0x55554661      0x00005555
0x7fffffffe020: 0x00000000      0x00000000      0x77894d06      0xec019f6d
0x7fffffffe030: 0x55554540      0x00005555      0xffffe0d0      0x00007fff
(gdb)
```

So we can see there that the value of i is store in the stack

```
(gdb) info locals
i = 0
buf1 = 0x7fffffffe0d0 "\001"
buf2 = 0x0
(gdb)
```

```
(gdb) p main
$5 = {int (int, char **)} 0x555555554661 <main(int, char**)>
(gdb)
```

```
20          return 0;
(gdb) info stack
#0  main (argc=1, argv=0x7fffffffe0d8) at test.c:20
(gdb) n
21      }
(gdb) info stack
#0  main (argc=1, argv=0x7fffffffe0d8) at test.c:21
(gdb) p rsp
No symbol "rsp" in current context.
(gdb) info registers rsp
rsp            0x7fffffffdfc0   0x7fffffffdfc0
(gdb)
```

So the main function is in the stack frame 0 which is the first stack frame which is created by the os when programmer is firstly run. The code for the main function reside in the code segment but when the program is called the first stack frame is given to the main function by the OS.

Try info registers. Which registers are holding aspects of the program that you recognize?

```
(gdb) info registers
rax            0x555555554661   93824992233057
rbx            0x0      0
rcx            0x5555555546b0   93824992233136
rdx            0x7fffffffe0e8   140737488347368
rsi            0x7fffffffe0d8   140737488347352
rdi            0x1      1
rbp            0x7fffffffdff0   0x7fffffffdff0
rsp            0x7fffffffdfc0   0x7fffffffdfc0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x0      0
r11            0x0      0
r12            0x555555554540   93824992232768
r13            0x7fffffffe0d0   140737488347344
r14            0x0      0
r15            0x0      0
rip            0x555555554670   0x555555554670 <main(int, char**)+15>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

We can see instruction pointer is pointing to the main function with reference to the code segment and rsp is the stack pointer

Then I replace the value of i=3 and run program which we can see in the stack in second row last column

```
(gdb) x/20x $rsp
0x7fffffffdfc0: 0xffffe0d8      0x00007fff      0x00000000      0x00000001
0x7fffffffdfd0: 0x555546b0      0x00005555      0x55554540      0x00005555
0x7fffffffdfe0: 0xffffe0d0      0x00007fff      0x00000000      0x00000000
0x7fffffffdff0: 0x555546b0      0x00005555      0xf7a03bf7      0x00007fff
0x7ffffffe000: 0x00000001       0x00000000      0xffffe0d8      0x00007fff
(gdb) n
14          char *buf1 = ( char*)malloc(10);
(gdb) x/20x $rsp
0x7fffffffdfc0: 0xffffe0d8      0x00007fff      0x00000000      0x00000001
0x7fffffffdfd0: 0x555546b0      0x00005555      0x55554540      0x00000003
0x7fffffffdfe0: 0xffffe0d0      0x00007fff      0x00000000      0x00000000
0x7fffffffdff0: 0x555546b0      0x00005555      0xf7a03bf7      0x00007fff
0x7ffffffe000: 0x00000001       0x00000000      0xffffe0d8      0x00007fff
(gdb)
```

```
19          my_func(3);
(gdb) si
0x0000555555554698      19          my_func(3);
(gdb) si
my_func (i=0) at test.c:5
5       int my_func(int i){
(gdb)
```

Then we inter in to the my_func lets check the registers

```
(gdb) info stack
#0  my_func (i=9) at test.c:7
#1  0x00005555555546a4 in main (argc=1, argv=0x7fffffffe0d8) at test.c:20
(gdb)
```

argv and argc are how command line arguments are passed to main() in C. argc will be the number of strings pointed to by argv. This will (in practice) be 1 plus the number of arguments, as virtually all implementations will prepend the name of the program to the array. argv[0] is the name of the program.The address of main is "0x7fffffffe0d8". The gdb info stack prints a backtrace of stack frames and prints a verbose description of the selected stack frame.

3. Write a shell script to capture Process ID by the process name and kill it if it exists.

The shell script will ask to input a process name, then it tries to capture the process id. If the process is running it kills it otherwise it just outputs that the process is not running and does not perform any action.

Open

p_killer.sh
~/Desktop

```
1 #!/bin/bash
2
3 echo "------------Process KIller----------"
4 echo -n "process name > "
5 read pname
6 process_id=$(pidof $pname)
7 if [ 0 == $? ]
8 then
9     echo "Process is running"
10    echo "killing process"
11    kill $process_id
12    echo "Process killed!"
13 echo
14    echo "Process is not running"
15 fi
```

dikshya@dikshya-VirtualBox: ~/Desktop

```
dikshya@dikshya-VirtualBox:~/Desktop$ firefox&
[1] 43841
dikshya@dikshya-VirtualBox:~/Desktop$ bash p_killer.sh
------------Process KIller----------
process name >
[1]+  Done                    firefox
dikshya@dikshya-VirtualBox:~/Desktop$
```