

**Lab 2 Report**

**Report Subject: OS Experiment - Lab 2**

**Student ID: 2018380039**

**Student Name: Dikshya Kafle**

**Submission Date：2021/11/18**

**Computer Operating System Experiment**

**Computer Operating System**

**Experiment Laboratory 2**

**Process**

1. **Objective:**

Learn to work with Linux system calls related to process creation and control.

Including: 

* Process create and data sharing between parent and child 
* The execution order of parent and child process 
* Create the specified num of child processes 
* Process termination 
* Zombie process 
* Process create a child process and load a new program

1. **Equipment:**

* VirtualBox with Ubuntu Linux 20.04

1. **Experiments:**

**Experiment 1: Process Creation**

1. If you change the values of variable x ,y and i in parent process, do the variable in the child process will affected? Please give the reason.

Answer:

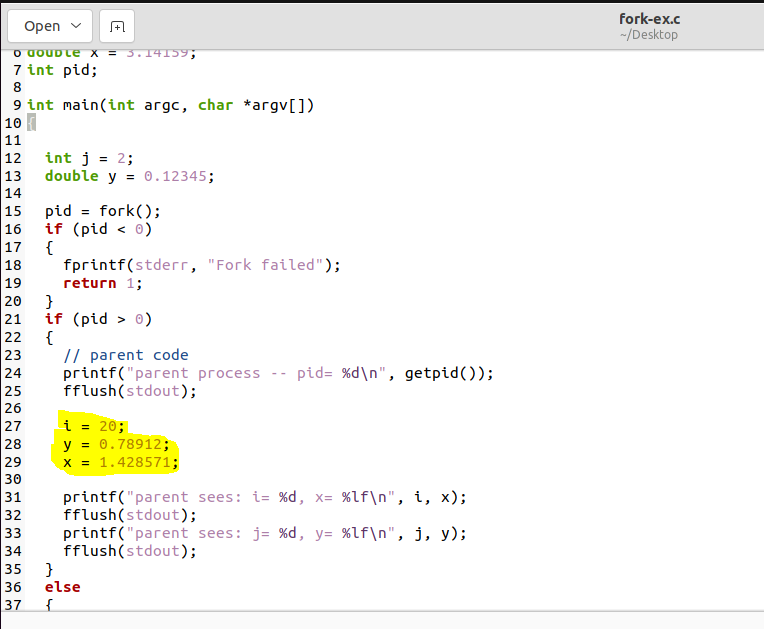
* If we change the values of variable x, y and i in the parent process the variables in the child will not be affected. This is because both the child and the parent process have different copies of the data and this data is not shared between them.

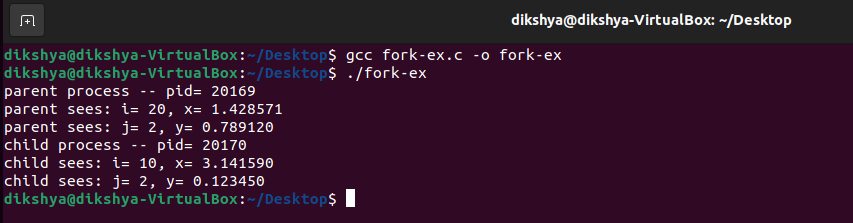
1. Please modify the fork-ex.c, and create a Makefile that builds all the programs you created. Test your expectation.

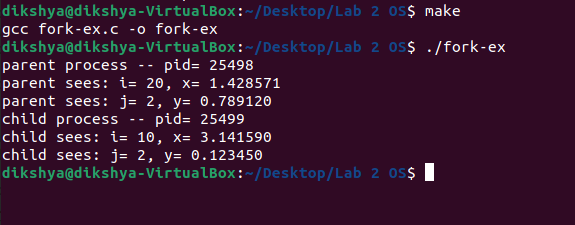
Answer:

* We modify the fork-ex.c to change the variables x,y and j on the parent process and test our last answer. The results are as expected and changing the variables in the parent process does not affect the variables in the child process.









**I have changed the variables in the parent process and run.**

**As we can see that changes of the variables in the parent process didn’t affect child process.**

**Experiment 2: the execution order of parent and child process**

1. The global variable num is declared before the call to fork() as shown in this program. After the call to fork(), when a new process is spawned, does there exist only one instance of num

in the memory space of the parent process shared by the two processes or do there exist two instances: one in the memory space of the parent and one in the memory space of the child?

Answer:

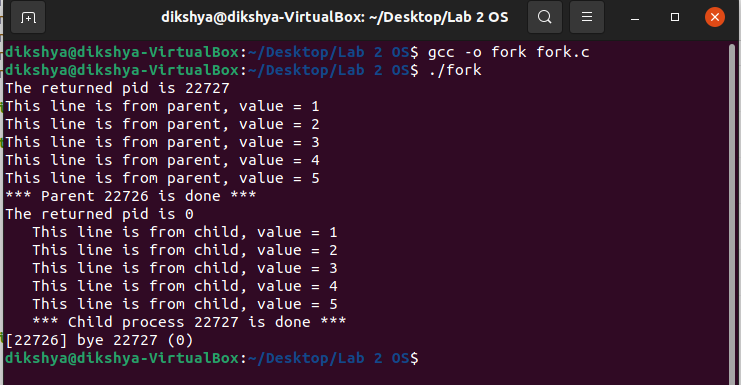
* There exist two instances of num, one in the memory space of the parent and one in the memory space of the the child. This is because the for will create a new copy of the parent process with a new pid that has a separate address space. So each process will have its own copy of the data instead of this data being shared between them.

1. Can you infer the order of execution of these lines? Please try to decrease the num, If the value of num is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

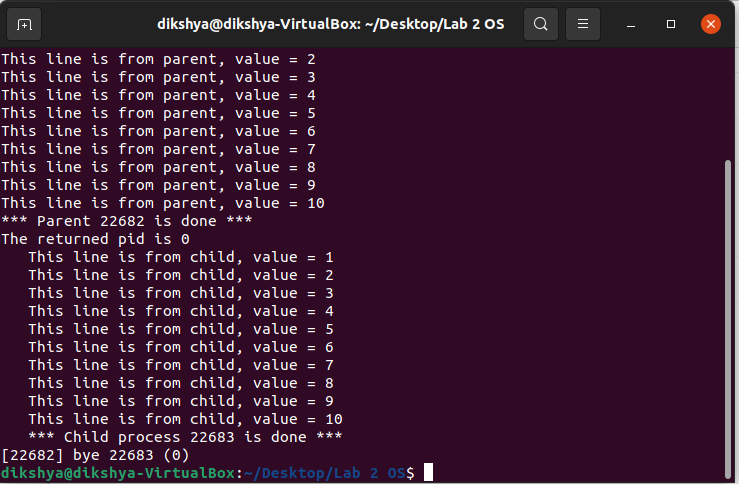
Answer:

* In this code we can’t infer the order of execution of the lines. It is true that the parent parent using wait () but this wait is not called right after the fork and the parent will still execute the for loop concurrently with the child process. However, the parent process will not return until the child process has returned. To illustrate this I increased the value of num from 5 to 40 (5 and 10 was too that each process could finish in one time quantum) and tested the results. As, expected that at one point the parent and the child process were executing concurrently.

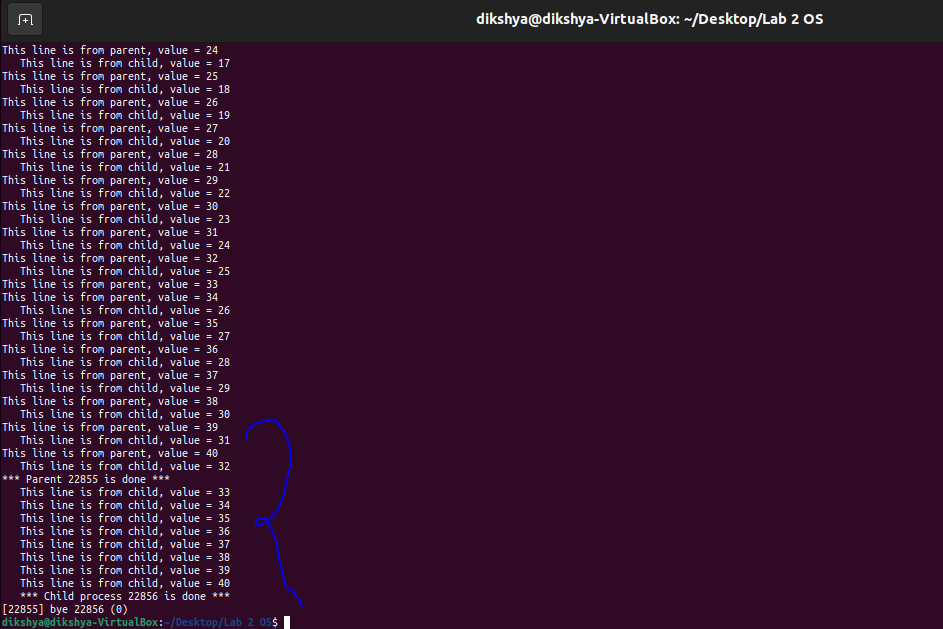
When num=5



When num=10



When num=40



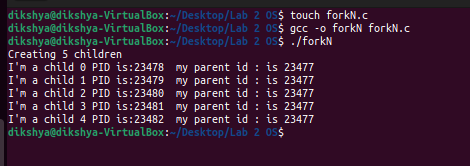
**Experiment 3: Create the specified num of child processes**

1. Please observe the pid and can you tell the policy about pid allocation? Can you determine the parent process and child process from the pid?

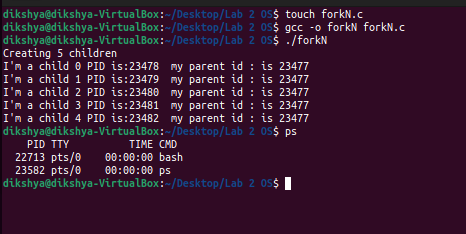
Answer:

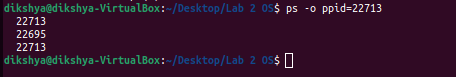
* We can observe that the policy about pid allocation Under Unix implies that process IDs are allocated on a sequential basis, beginning at 0. However, we cannot determine the parent and child process from the pid. For example if we run pgrep -P 22713 (where 22713 is the bash pid) we get: 22695 (not 22712)(see last screen-shot)





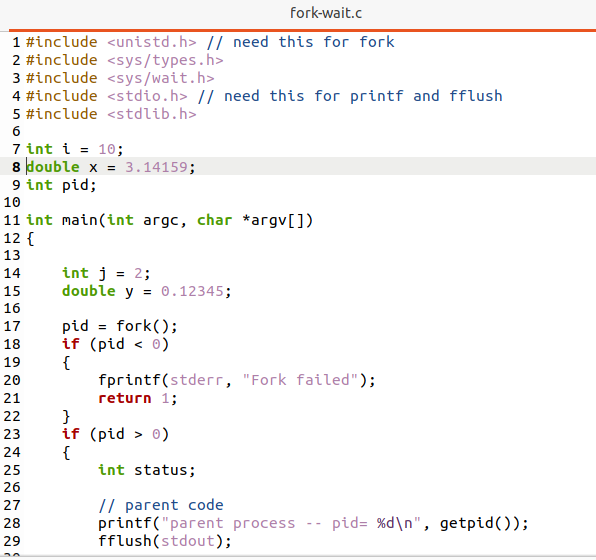
* Can you determine the parent process and child process from the pid? --> No - example:

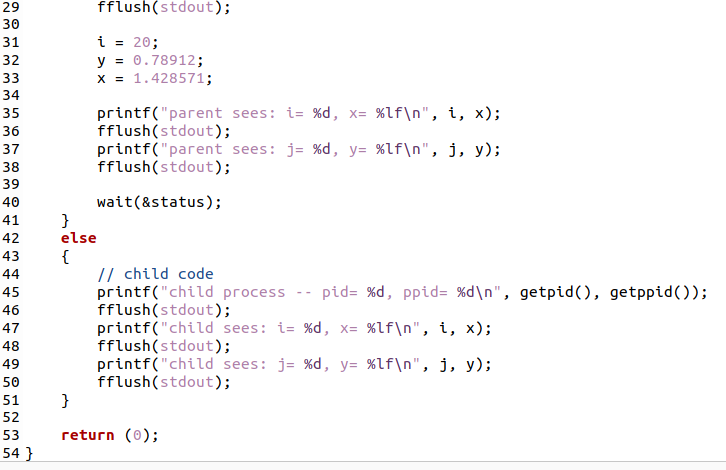
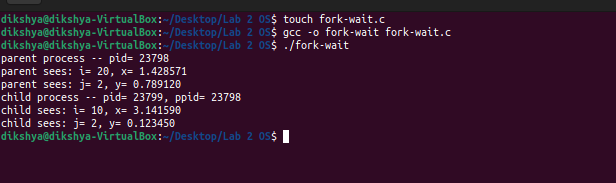




**Experiment 4: Process termination**

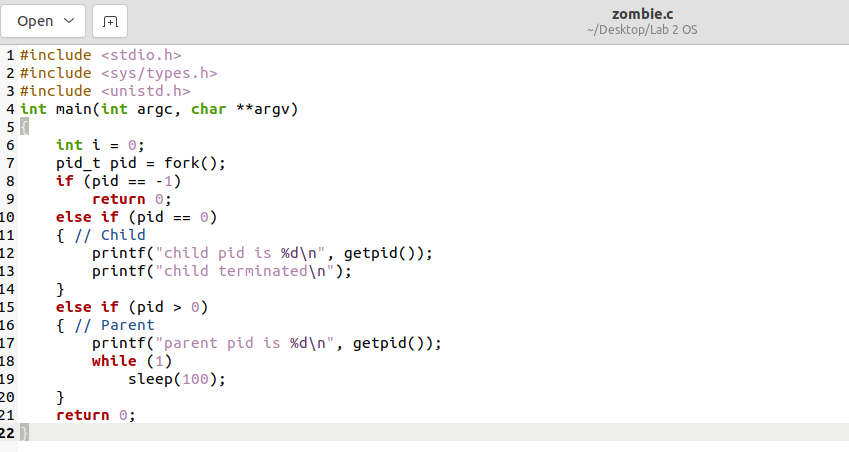
Copy fork\_ex.c to file fork-wait.c and modify it so that you can guarantee that the parent process will always terminate after the child process has terminated. Your solution cannot rely on the termination condition of the for loops or on the use of sleep. The right way to handle this is using a syscall such as wait or waitpid – read their man pages before jumping into this task. One more thing: Modify the child process so that it makes calls to getpid(2) and getppid(2) and prints out the values returned by these calls



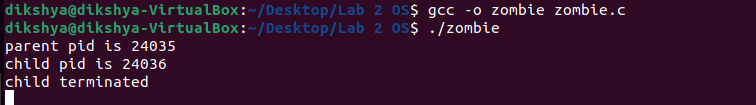
**Experiment 5: Zombie process**

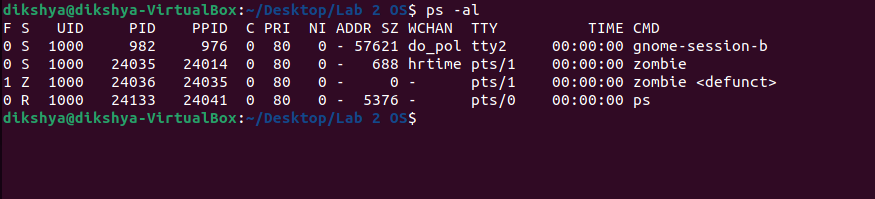
* To make a zombie process, I made the parent process sleep before executing the wait() and made the child process exit. I execute the ./zombie command on a shell window and executed “ps aux” in the an other window while the parent process is sleeping and I found that child process has a “zombie” state.



The zombie process created through this code will run for 100 seconds. We can increase the time duration by specifying a time (in seconds) in the sleep() function.

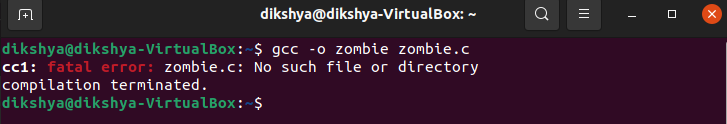
Now I run the zombie program:





The **ps** commands will now also show this defunct process, we open a new terminal and use the command to check the defunct process(**zombie process**):

1. Can you use kill command to kill the zombie process? If not, how can you reap the zombie process?



A Zombie process can’t be killed because it is already dead. One thing we can try to reap it is to notify its parent process explicitly so that it can retry to read the child (dead) process’s status and eventually clean it from the process table. This can be done by sending a **SIGCHLD** signal to the parent process.

*$ kill -s SIGCHLD <Parent PID>*

We have seen that to get the parent pid we can simply run: *$ ps -o ppid= <Child PID>*

However, this might not always work. In our case the parent process is sleeping, so the only way to reap the child zombie process is by killing its parent. After the parent dies, the zombie will be inherited by pid 1, which will wait on it and clear its entry in the process table (This solution always works):

$ kill <Parent PID>

The init process regularly performs the necessary cleanup of zombies, so to kill them, you just have to kill the process that created them

**Experiment 6: create a child process and load a new program**

I first code a Scan() function, which performs a lexical analysis to the user input. The contents of the command entered by the user are loaded into the args array as specified. Then the parent process forks a new child process that run the execvp() function. If the user added “&” the parent process will wait for the child process to exit, otherwise they will run concurrently.

Modified Code:

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <stdlib.h>

#include <sys/types.h>

#include <string.h>

#define MAX\_LINE 80 /\* 80 chars per line, per command, should be enough. \*/

#define MAX\_COMMANDS 5 /\* size of history \*/

char history[MAX\_COMMANDS][MAX\_LINE];

char display\_history [MAX\_COMMANDS][MAX\_LINE];

int command\_count = 0;

/\*\*

\* Add the most recent command to the history.

\*/

void addtohistory(char inputBuffer[]) {

int i = 0;

// add the command to history

strcpy(history[command\_count % MAX\_COMMANDS], inputBuffer);

// add the display-style command to history

while (inputBuffer[i] != '\n' && inputBuffer[i] != '\0') {

display\_history[command\_count % MAX\_COMMANDS][i] = inputBuffer[i];

i++;

}

display\_history[command\_count % MAX\_COMMANDS][i] = '\0';

++command\_count;

return;

}

int setup(char inputBuffer[], char \*args[],int \*background)

{

int length, /\* # of characters in the command line \*/

i, /\* loop index for accessing inputBuffer array \*/

start, /\* index where beginning of next command parameter is \*/

ct, /\* index of where to place the next parameter into args[] \*/

command\_number; /\* index of requested command number \*/

ct = 0;

/\* read what the user enters on the command line \*/

do {

printf("os>");

fflush(stdout);

length = read(STDIN\_FILENO,inputBuffer,MAX\_LINE);

}

while (inputBuffer[0] == '\n'); /\* swallow newline characters \*/

start = -1;

if (length == 0)

exit(0); /\* ^d was entered, end of user command stream \*/

if ( (length < 0) && (errno != EINTR) ) {

perror("error reading the command");

exit(-1); /\* terminate with error code of -1 \*/

}

/\*\*

\* Check if they are using history

\*/

if (inputBuffer[0] == '!') {

if (command\_count == 0) {

printf("No history\n");

return 1;

}

else if (inputBuffer[1] == '!') {

// restore the previous command

strcpy(inputBuffer,history[(command\_count - 1) % MAX\_COMMANDS]);

length = strlen(inputBuffer) + 1;

}

else if (isdigit(inputBuffer[1])) { /\* retrieve the nth command \*/

command\_number = atoi(&inputBuffer[1]);

strcpy(inputBuffer,history[command\_number]);

length = strlen(inputBuffer) + 1;

}

}

/\*\*

\* Add the command to the history

\*/

addtohistory(inputBuffer);

/\*\*

\* Parse the contents of inputBuffer

\*/

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

/\* examine every character in the inputBuffer \*/

switch (inputBuffer[i]){

case ' ':

case '\t' : /\* argument separators \*/

if(start != -1){

args[ct] = &inputBuffer[start]; /\* set up pointer \*/

ct++;

}

inputBuffer[i] = '\0'; /\* add a null char; make a C string \*/

start = -1;

break;

case '\n': /\* should be the final char examined \*/

if (start != -1){

args[ct] = &inputBuffer[start];

ct++;

}

inputBuffer[i] = '\0';

args[ct] = NULL; /\* no more arguments to this command \*/

break;

default : /\* some other character \*/

if (start == -1)

start = i;

if (inputBuffer[i] == '&') {

\*background = 1;

inputBuffer[i-1] = '\0';

}

} /\* end of switch \*/

} /\* end of for \*/

/\*\*

\* If we get &, don't enter it in the args array

\*/

if (\*background)

args[--ct] = NULL;

args[ct] = NULL; /\* just in case the input line was > 80 \*/

return 1;

} /\* end of setup routine \*/

int main(void)

{

char inputBuffer[MAX\_LINE]; /\* buffer to hold the command entered \*/

int background; /\* equals 1 if a command is followed by '&' \*/

char \*args[MAX\_LINE/2 + 1]; /\* command line (of 80) has max of 40 arguments \*/

pid\_t child; /\* process id of the child process \*/

int status; /\* result from execvp system call\*/

int shouldrun = 1;

int i, upper;

while (shouldrun){ /\* Program terminates normally inside setup \*/

background = 0;

shouldrun = setup(inputBuffer,args,&background); /\* get next command \*/

if (strncmp(inputBuffer, "exit", 4) == 0)

return 0;

else if (strncmp(inputBuffer,"history", 7) == 0) {

if (command\_count < MAX\_COMMANDS)

upper = command\_count;

else

upper = MAX\_COMMANDS;

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

printf("%d \t %s\n", i, display\_history[i]);

}

continue;

}

if (shouldrun) {

child = fork(); /\* creates a duplicate process! \*/

switch (child) {

case -1:

perror("could not fork the process");

break;

case 0: /\* this is the child process \*/

status = execvp(args[0],args);

if (status != 0){

perror("error in execvp");

exit(-2); /\* terminate this process with error code -2 \*/

}

break;

default : /\* this is the parent \*/

if (background == 0) /\* handle parent,wait for child \*/

while (child != wait(NULL))

;

}

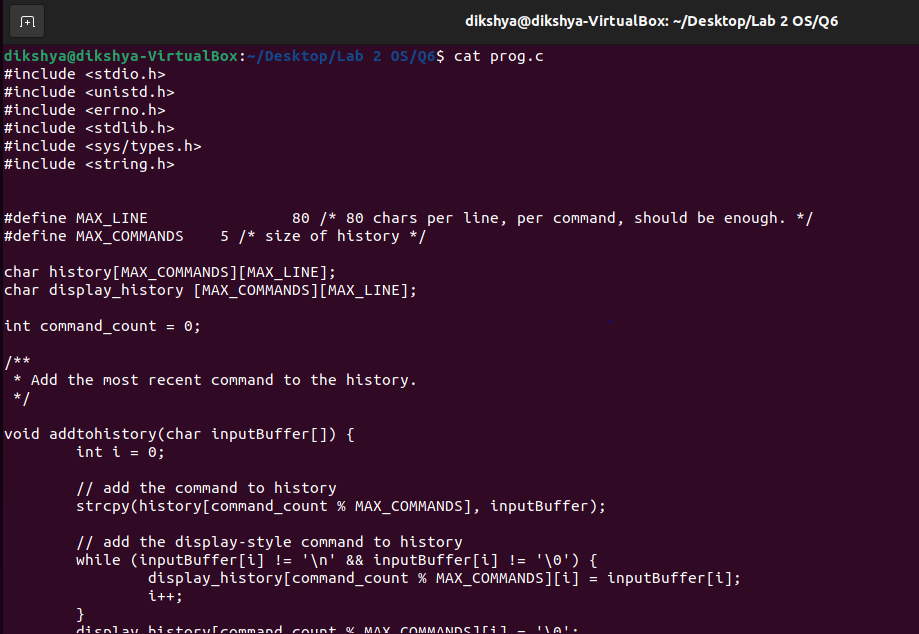
}

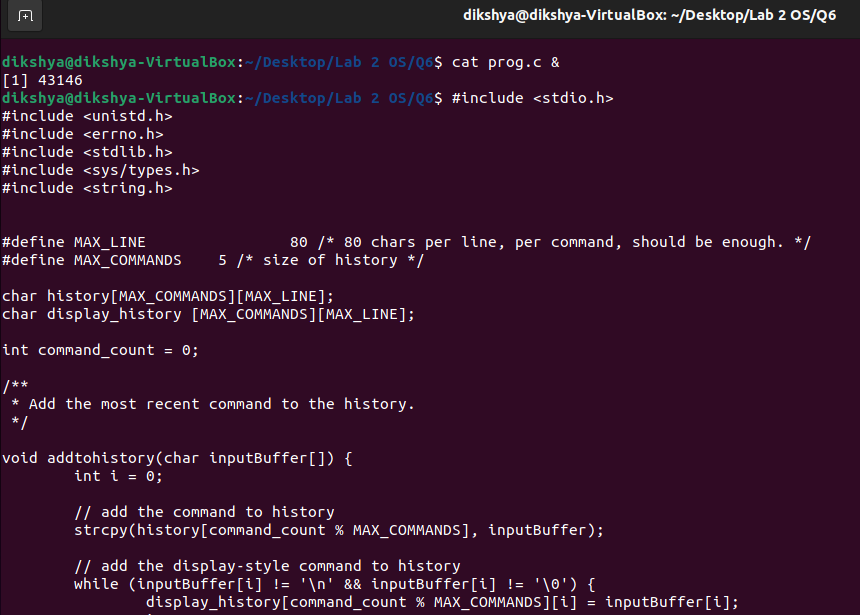
}

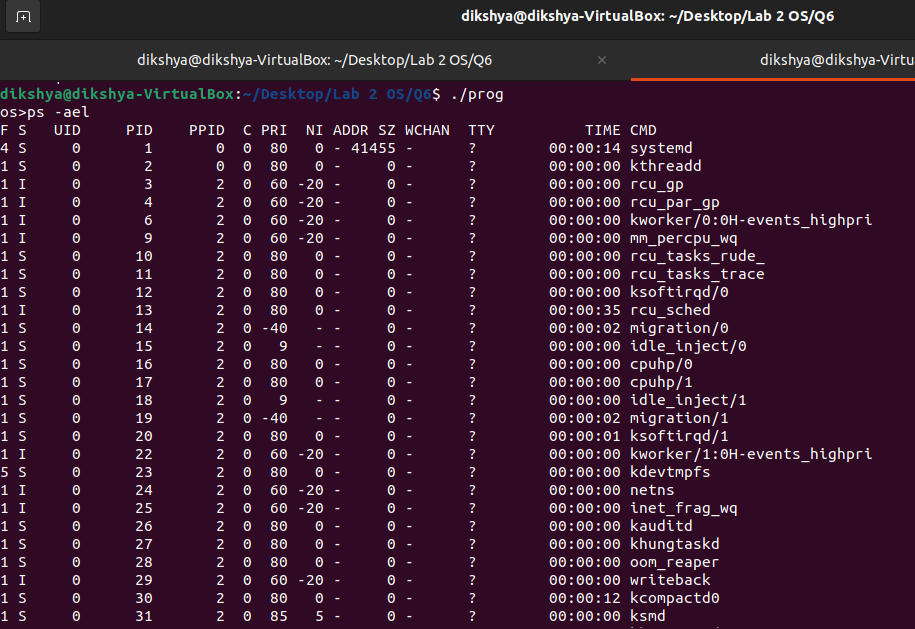
return 0;

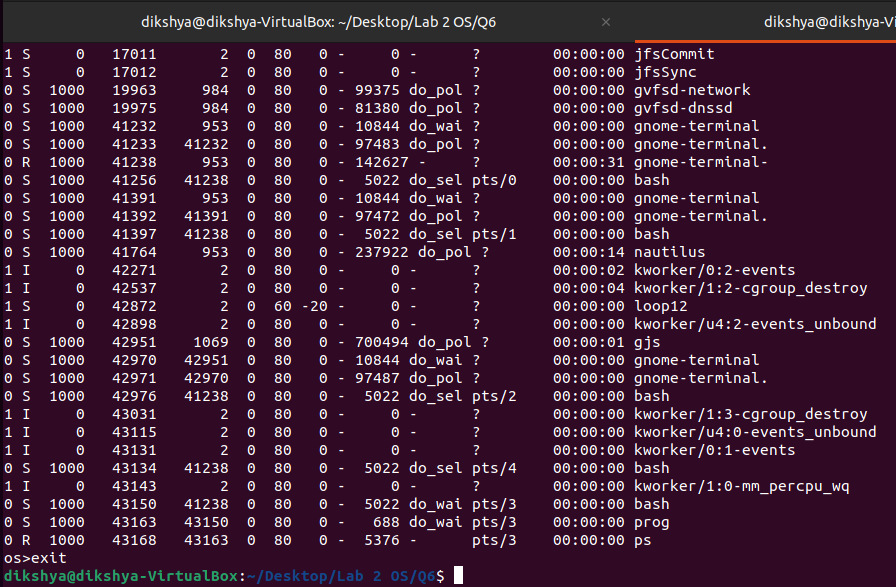
}

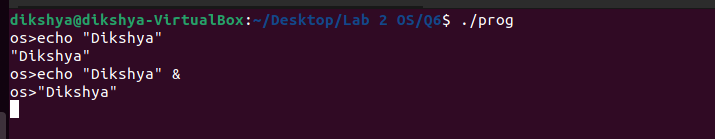
Procedure & Results:











Results: Notice how “OS>”may be printed by the parent process before the child return if “&” is not added to the command (child runs still runs in the background). However if it is added the parent won’t as for the command until the child process terminates.