



**西北工业大学**  
NORTHWESTERN POLYTECHNICAL UNIVERSITY

## **Lab | Report**

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

**Student ID** : 2019380141  
**Student Name** : ABID ALI  
**Experiment Name** : Process  
**Email** : abiduu354@gmail.com

**Data: 2021/10/20**

## Objective:

- 1) Process create and data sharing between parent and child.
- 2) The execution order of parent and child process.
- 3) Create the specified num of child processes.
- 4) Process termination.
- 5) Zombie process.
- 6) Process create a child process and load a new program

## Equipment:

VirtualBox with Ubuntu Linux

## Methodology:

### 3.1 Experiment 1: process creation

#### Question:

- 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:** “No”.The reason si declaring values outside the main function is known as global declaration and the variables are called global variables.The global variables are `int i = 10;` `double x =3.14159.`We need to change the values of variable x ,y and i in parent process which is inside the main function the we consider the values we input inside the parent process will be local variable. We can see, when we run the teacher’s code the parent and child process has exactly same values except their pid.We can see that,when we put values inside the parent code then the values in child process will not be affected.

```

child sees: j= 2, y= 0.123450
abid@ubuntu:~/1$ make all
gcc -o fork-ex fork-ex.c
abid@ubuntu:~/1$ make exec
./fork-ex
parent process -- pid= 7088
parent sees: i= 6, x= 6.800000
parent sees: j= 2, y= 4.400000
abid@ubuntu:~/1$ child process -- pid= 7089
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450

```

We can clearly see that, parent took the local variable values but the child took the global values. It happened because those values are not inside the block of child.

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

**Answer:**

```

abid@ubuntu:~/1.1$ make exec
./fork-ex
parent process -- pid= 3533
parent sees: i= 10, x= 3.141590
parent sees: j= 2, y= 0.123450
abid@ubuntu:~/1.1$ child process -- pid= 3534
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
abid@ubuntu:~/1.1$

```

The above picture where we didn't change the value. We used the value given to us by the teacher.

```

abid@ubuntu:~/1$ make exec
./fork-ex
parent process -- pid= 3216
parent sees: i= 10, x= 6.800000
parent sees: j= 2, y= 0.123450
child process -- pid= 3217
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
abid@ubuntu:~/1$

```

The above picture where we change the value of 'x' in parent code to 6.8(local variable) .The variable in child process is unchanged.

```
abid@ubuntu:~/1$ make exec
./fork-ex
parent process -- pid= 3686
parent sees: i= 10, x= 6.800000
parent sees: j= 2, y= 4.400000
child process -- pid= 3687
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
abid@ubuntu:~/1$
```

The above picture where we change the value of 'x' to 6.8 , 'y' to 4.4 and 'i' to 4 .The variable in child process is unchanged.Because,those variable are not in the block child code.So,they show the global variable.

```
abid@ubuntu:~/1$ make exec
./fork-ex
parent process -- pid= 3892
parent sees: i= 4, x= 6.800000
parent sees: j= 2, y= 4.400000
abid@ubuntu:~/1$ child process -- pid= 3893
child sees: i= 4, x= 3.141590
child sees: j= 2, y= 0.123450

```

Fork Function is used in below pictures.

```
abid@ubuntu:~/1$ make all
gcc -o fork-ex fork-ex.c
abid@ubuntu:~/1$ make exec
./fork-ex
parent process -- pid= 4030
parent sees: i= 10, x= 3.141590
parent sees: j= 2, y= 0.123450
abid@ubuntu:~/1$ child process -- pid= 4031
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450

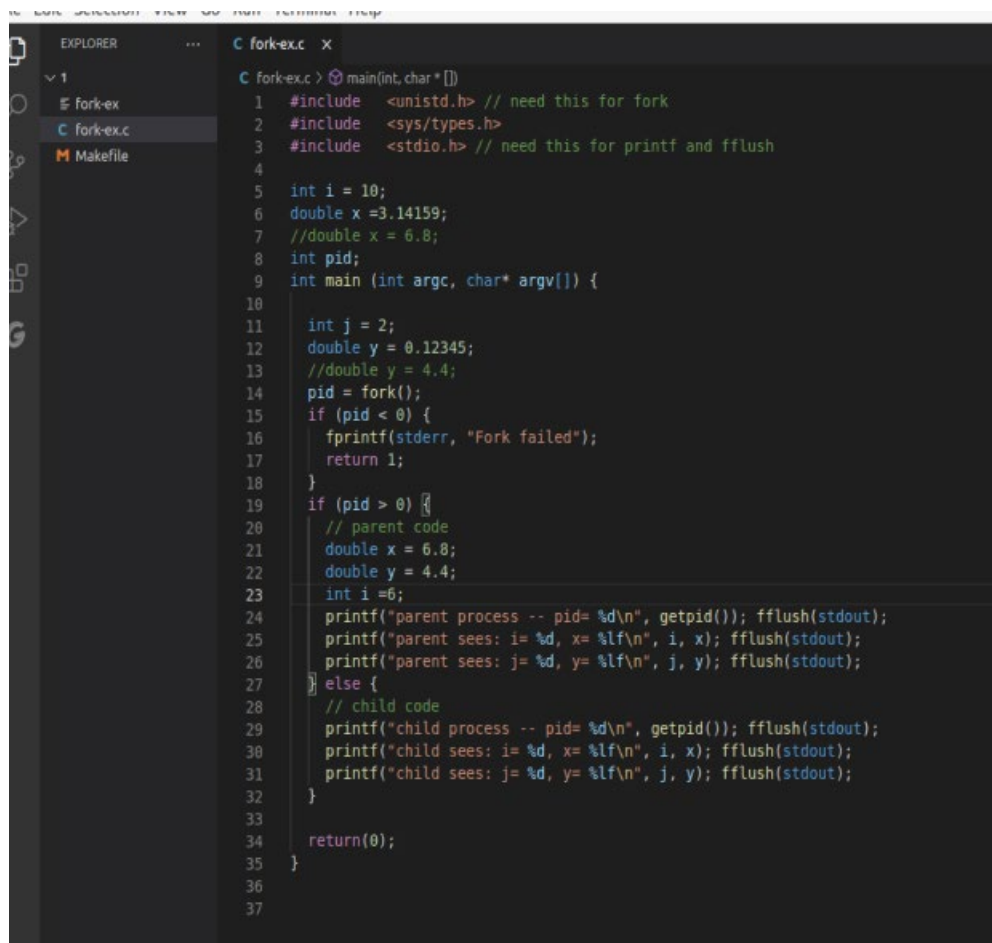
```

```

abid@ubuntu:~/1$ make all
gcc -o fork-ex fork-ex.c
abid@ubuntu:~/1$ make exec
./fork-ex
parent process -- pid= 4397
parent sees: i= 4, x= 6.800000
parent sees: j= 2, y= 4.400000
child process -- pid= 4398
child sees: i= 4, x= 6.800000
child sees: j= 2, y= 4.400000
abid@ubuntu:~/1$

```

We got the result as per our expectation.

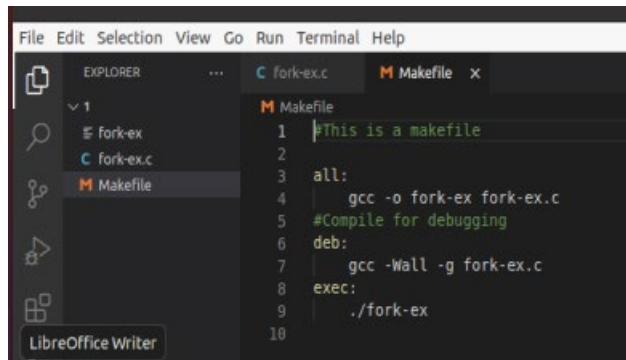


```

1  #include <unistd.h> // need this for fork
2  #include <sys/types.h>
3  #include <stdio.h> // need this for printf and fflush
4
5  int i = 10;
6  double x = 3.14159;
7  //double x = 6.8;
8  int pid;
9  int main (int argc, char* argv[]) {
10
11     int j = 2;
12     double y = 0.12345;
13     //double y = 4.4;
14     pid = fork();
15     if (pid < 0) {
16         fprintf(stderr, "Fork failed");
17         return 1;
18     }
19     if (pid > 0) {
20         // parent code
21         double x = 6.8;
22         double y = 4.4;
23         int i = 6;
24         printf("parent process -- pid= %d\n", getpid()); fflush(stdout);
25         printf("parent sees: i= %d, x= %lf\n", i, x); fflush(stdout);
26         printf("parent sees: j= %d, y= %lf\n", j, y); fflush(stdout);
27     } else {
28         // child code
29         printf("child process -- pid= %d\n", getpid()); fflush(stdout);
30         printf("child sees: i= %d, x= %lf\n", i, x); fflush(stdout);
31         printf("child sees: j= %d, y= %lf\n", j, y); fflush(stdout);
32     }
33
34     return(0);
35 }
36
37

```

Fig:Picture of code



**Fig:Makefile of the code**

When we change the value of global variable x,y,and i at the top. We see the value in child is also changed.The value of variable is similar to parent.

### **3.2 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?

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

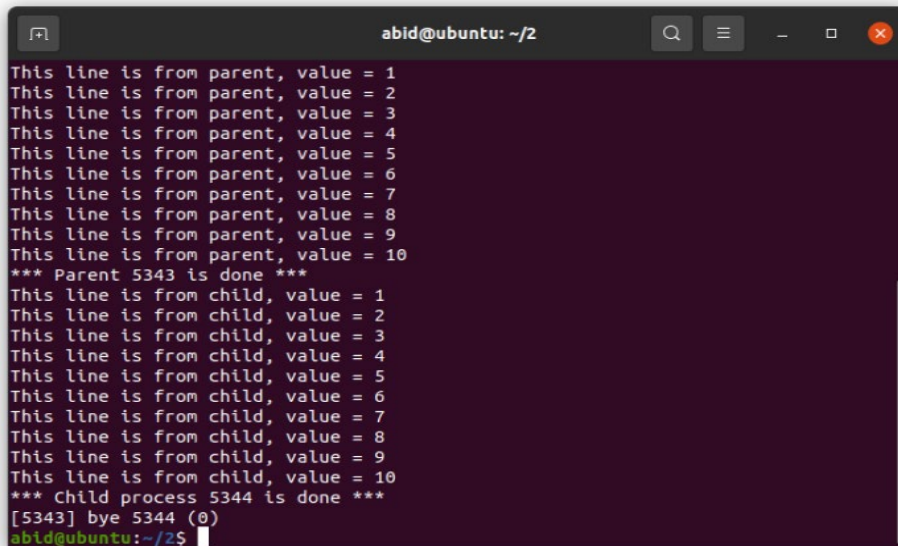
```

int num = 0;
void ChildProcess(void); /* child process prototype
*/
void ParentProcess(void); /* parent process prototype */
void main(void)
{
    pid_t
    pid;
    pid = fork();
    if(pid == -1)
    {
        printf("something went wrong in fork");
        exit(-1);
    }
    else if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}
void ChildProcess(void)
{
    int i;
    int mypid;
    mypid = getpid();
    for (i = 1; i <= num; i++)

        printf("This line is from child, value = %d\n", i);
    fflush(stdout);
    printf("*** Child process %d is done ***\n",mypid);
    exit(0);
}
void ParentProcess(void)
{
    int
    i,status;
    int got_pid,mypid;
    mypid = getpid();
    for (i = 1; i <= num; i++)
        printf("This line is from parent, value = %d\n", i);
    fflush(stdout);
    printf("*** Parent %d is done ***\n",mypid);
    got_pid = wait(&status);
    printf("[%d] bye %d (%d)\n", mypid, got_pid, status);
}

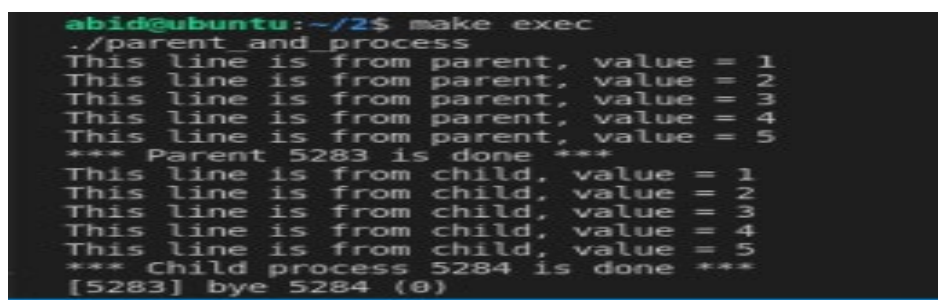
```

2.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.

A terminal window titled 'abid@ubuntu: ~/2' showing the output of a program. The output consists of two distinct groups of lines. The first group, from the parent process (PID 5343), contains 10 lines: 'This line is from parent, value = 1' through 'value = 10', followed by '\*\*\* Parent 5343 is done \*\*\*'. The second group, from the child process (PID 5344), also contains 10 lines: 'This line is from child, value = 1' through 'value = 10', followed by '\*\*\* Child process 5344 is done \*\*\*'. The terminal ends with '[5343] bye 5344 (0)' and the prompt 'abid@ubuntu:~/2\$'.

```
abid@ubuntu: ~/2
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
This line is from parent, value = 4
This line is from parent, value = 5
This line is from parent, value = 6
This line is from parent, value = 7
This line is from parent, value = 8
This line is from parent, value = 9
This line is from parent, value = 10
*** Parent 5343 is done ***
This line is from child, value = 1
This line is from child, value = 2
This line is from child, value = 3
This line is from child, value = 4
This line is from child, value = 5
This line is from child, value = 6
This line is from child, value = 7
This line is from child, value = 8
This line is from child, value = 9
This line is from child, value = 10
*** Child process 5344 is done ***
[5343] bye 5344 (0)
abid@ubuntu:~/2$
```

When the value of num is 10 then we get the above picture.

A terminal window showing the output of a program with num=5. The output consists of two groups of lines. The first group, from the parent process (PID 5283), contains 5 lines: 'This line is from parent, value = 1' through 'value = 5', followed by '\*\*\* Parent 5283 is done \*\*\*'. The second group, from the child process (PID 5284), also contains 5 lines: 'This line is from child, value = 1' through 'value = 5', followed by '\*\*\* Child process 5284 is done \*\*\*'. The terminal ends with '[5283] bye 5284 (0)'.

```
abid@ubuntu:~/2$ make exec
./parent_and_process
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
This line is from parent, value = 4
This line is from parent, value = 5
*** Parent 5283 is done ***
This line is from child, value = 1
This line is from child, value = 2
This line is from child, value = 3
This line is from child, value = 4
This line is from child, value = 5
*** Child process 5284 is done ***
[5283] bye 5284 (0)
```

When the value of num is 5 then we get the above picture.



```

abid@ubuntu:~/2$ make exec
./parent_and_process
This line is from parent, value = 1
This line is from parent, value = 2
*** Parent 5627 is done ***
This line is from child, value = 1
This line is from child, value = 2
*** Child process 5628 is done ***
[5627] bye 5628 (0)
abid@ubuntu:~/2$

```

When the value of num is 2 then we get the above picture.

```

abid@ubuntu:~/2$ make exec
./parent_and_process
This line is from parent, value = 1
*** Parent 5745 is done ***
This line is from child, value = 1
*** Child process 5746 is done ***
[5745] bye 5746 (0)
abid@ubuntu:~/2$

```

When the value of num is 1 then we get the above picture.

```

42 | get_pid = wait(&status);
abid@ubuntu:~/2$ make exec
./parent_and_process
*** Parent 5995 is done ***
*** Child process 5996 is done ***
[5995] bye 5996 (0)
abid@ubuntu:~/2$

```

When the value of num is 0 then we get the above picture.

### 3.3 Experiment 3: Create the specified num of child processes

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?

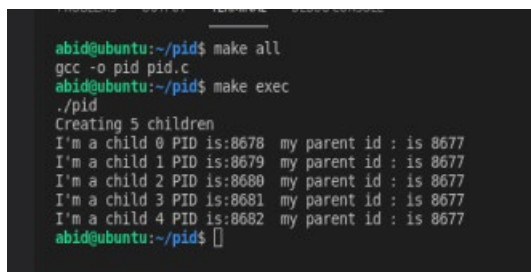
## Answer:

**fork()** creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process and the parent process run in separate memory spaces. At the time of **fork()** both memory spaces have the same content. Memory writes, file mappings ([mmap\(2\)](#)), and unmappings ([munmap\(2\)](#)) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

- \* The child has its own unique process ID, and this PID does not match the ID of any existing process group ([setpgid\(2\)](#)) or session.
- \* The child's parent process ID is the same as the parent's process ID.
- \* The child does not inherit its parent's memory locks([mlock\(2\)](#), [mlockall\(2\)](#)).
- \* The child does not inherit process-associated record locks from its parent ([fcntl\(2\)](#)). (On the other hand, it does inherit [fcntl\(2\)](#) open file description locks and [flock\(2\)](#) locks from its parent.)



```
abid@ubuntu:~/pid$ make all
gcc -o pid pid.c
abid@ubuntu:~/pid$ make exec
./pid
Creating 5 children
I'm a child 0 PID is:8678 my parent id : is 8677
I'm a child 1 PID is:8679 my parent id : is 8677
I'm a child 2 PID is:8680 my parent id : is 8677
I'm a child 3 PID is:8681 my parent id : is 8677
I'm a child 4 PID is:8682 my parent id : is 8677
abid@ubuntu:~/pid$
```

Yes, We can determine the parent process and child process from the pid.

### 3.4 Experiment 4: Process termination

```
1 //include <unistd.h> // need this for fork
2 #include <sys/types.h>
3 #include <stdio.h> // need this for printf and fflush
4 #include<sys/wait.h>
5
6 int i = 10;
7 double x =3.14159;
8 int pid;
9
10 int main (int argc, char* argv[]) {
11
12     int j = 2;
13     double y = 0.12345;
14
15     pid = fork();
16     int status;
17     if (pid < 0) {
18         fprintf(stderr, "Fork failed");
19         return 1;
20     }
21     if (pid > 0) {
22         // parent code
23         printf("parent process -- pid= %d\n", getpid()); fflush(stdout);
24         printf("parent sees: i= %d, x= %lf\n", i, x); fflush(stdout);
25         printf("parent sees: j= %d, y= %lf\n", j, y); fflush(stdout);
26         printf("Parent process ID: %d\n",getppid());
27         wait(&status);
28         printf("I waited for the child process\n");
29     } else {
30         // child code
31         printf("child process -- pid= %d\n", getpid()); fflush(stdout);
32         printf("child sees: i= %d, x= %lf\n", i, x); fflush(stdout);
33         printf("child sees: j= %d, y= %lf\n", j, y); fflush(stdout);
34         printf("Child process ID: %d\n",getpid());
35         sleep(10);
36     }
37 }
38
39 return(0);
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
fork-wait.c:36:5: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
   36 |     sleep(10);
      |     ~~~~~
abid@ubuntu:~/4$ make exec
./fork-wait
parent process -- pid= 13115
parent sees: i= 10, x= 3.141590
parent sees: j= 2, y= 0.123450
Parent process ID: 13114
child process -- pid= 13116
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
Child process ID: 13116
I waited for the child process
abid@ubuntu:~/4$
```

```
I waited for the child process
abid@ubuntu:~/4$ make exec
./fork-wait
parent process -- pid= 13240
parent sees: i= 10, x= 3.141590
parent sees: j= 2, y= 0.123450
Parent process ID: 13239
child process -- pid= 13241
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
Child process ID: 13241
```

In the code ,I intentionally put the child process to sleep for 10 seconds.We can see that the parent process executed all the print function until the wait system call.Then,child process started to execute .After completing all the print function then child sleep for 10 seconds.After completing

10 second,we can see that last print function is executed.In this way ,we can guarantee the parent process will execute at the end .

### 3.5 Experiment 5: Zombie process

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

```
abid@ubuntu:~/zombie$ make exec
./zombie
parent process -- pid= 13226
parent sees: i= 10, x= 3.141590
parent sees: j= 2, y= 0.123450
Parent process ID: 13225
child process -- pid= 13227
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
Child process ID: 13227
I am child,I executed before parents
I am a Zombie now
I didn't waited for the child process
abid@ubuntu:~/zombie$
```

```
abid@ubuntu:~/zombie$ ps -al
F S  UID      PID      PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000     1444      1442  0  80   0 - 98286 ep_pol tty2      00:01:46 Xorg
0 S  1000     1490      1442  0  80   0 - 49826 poll_s tty2      00:00:00 gnome-sess
0 S  1000     13225     12355  0  80   0 - 4271 do_wai pts/0      00:00:00 make
0 S  1000     13226     13225  0  80   0 - 624 hrtim pts/0      00:00:00 zombie
1 Z  1000     13227     13226  0  80   0 -    - pts/0      00:00:00
0 R  1000     13279     13273  0  80   0 - 5013 - pts/2      00:00:00 ps
abid@ubuntu:~/zombie$
```

```
top - 14:55:39 up 5:46, 1 user, load average: 0.11, 0.08, 0.02
Tasks: 313 total, 1 running, 311 sleeping, 0 stopped, 1 zombie
%Cpu(s): 0.8 us, 0.3 sy, 0.0 ni, 98.8 id, 0.0 wa, 0.0 ht, 0.0 st, 0.0 sr
MiB Mem : 3901.4 total, 1312.9 free, 1634.4 used, 954.1 buff/cache
MiB Swap: 2048.0 total, 2045.6 free, 2.4 used, 1978.7 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 1606 abid      20   0 4251228 302640 96152 S   2.0   7.6   3:10.15 gnome-s+
 1444 abid      20   0 376852 107416 48296 S   0.7   2.7   1:51.31 Xorg
    20 root       20   0      0      0      0 S   0.3   0.0   0:00.46 ksoftir+
   728 root       20   0 321872   7608  6232 S   0.3   0.2   0:24.45 vmtoolsd
   991 rtkit      21   1 152940   3020  2784 S   0.3   0.1   0:00.32 rtkit-d+
  1800 abid      20   0 300680  35264 23392 S   0.3   0.9   0:26.25 vmtoolsd
 12270 abid      20   0   36.5g 141720 57592 S   0.3   3.5   0:06.84 code
 13387 root       20   0      0      0      0 I   0.3   0.0   0:00.10 kworker+
    1 root       20   0 167592  11464  8372 S   0.0   0.3   0:03.11 systemd
    2 root       20   0      0      0      0 S   0.0   0.0   0:00.03 kthreadd
    3 root       0 -20      0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20      0      0      0 I   0.0   0.0   0:00.00 rcu_par+
    6 root       0 -20      0      0      0 I   0.0   0.0   0:00.00 kworker+
    9 root       0 -20      0      0      0 I   0.0   0.0   0:00.00 mm_perc+
   10 root       20   0      0      0      0 S   0.0   0.0   0:00.00 rcu_tas+
```

**Answer:** The term zombie process derives from the common definition of zombie an undead person. In the term's metaphor, the child process has "died" but has not yet been "reaped". Also, unlike normal processes, the kill command has no effect on a zombie process.

Zombie processes are already dead, so they cannot be killed, they can only be reaped, which has to be done by their parent process via `wait*()`. This is usually called the **child reaper** idiom, in the signal handler for **SIGCHLD**

The **top** command displays a detailed view of the processes running on your system along with the memory and CPU resources they are using. It also gives you information about any zombie processes running on your system. Open the Terminal by pressing **Ctrl+Alt+T** and then type **top**. I got the following output after running this command.

```
ps axo stat,ppid,pid,comm | grep -w defunct
```

This command will give you the state, parentID, the process ID, the program that is running the zombie process(a dummy program by the name 'zombie' on my system). The defunct flag tells you that this is a dead, zombie process.

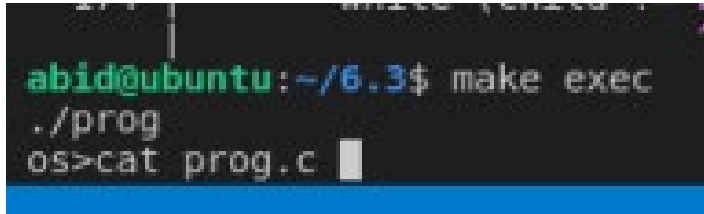
We can use GUI to delete the that process by clicking kill.

We can also use command line to kill the Zombie process

```
kill -s SIGCHLD PID
```

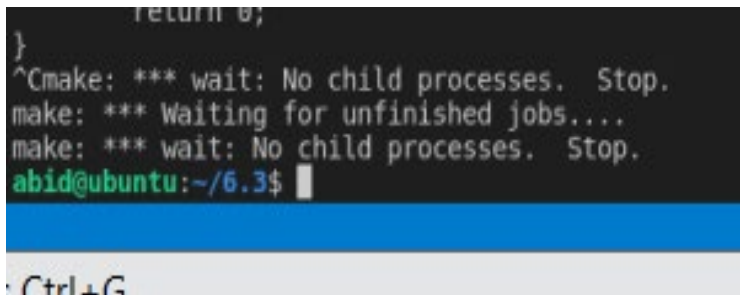
Alternate way is to restart the system ,it removes those zombies.

### 3.6 Experiment 6: create a child process and load a new program



```
abid@ubuntu:~/6.3$ make exec
./prog
os>cat prog.c
```

We are typing cat prog.c in prompt screen of our shell. The lines inside the file are printed.



```
return 0;
}
^Cmake: *** wait: No child processes. Stop.
make: *** Waiting for unfinished jobs....
make: *** wait: No child processes. Stop.
abid@ubuntu:~/6.3$
```

We can see that, if we suddenly close the running program, it shows "Waiting for unfinished job" and also "wait for the child process".

```

int main(void)
{
    char buf[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){
        background = 0; /* Program terminates normally inside setup */

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

        if (strcmp(buf, "exit", 4) == 0)
            return 0;
        else if (strcmp(buf,"NULL", 7) == 0) {
            if (command < MAX_LINE)
                upper = command;
            else
                upper = MAX_LINE;

            for (i = 0; i < upper; i++) {
                printf("%d \t %d\n", i, display[i]);
            }

            continue;
        }

        if (shouldrun) {
            child = fork(); /* creates a duplicate process! */
            switch (child) {
                case -1:
                    perror("could not fork the process");
                    break; /* perror is a library routine that displays a system
error message, according to the value of the system
variable "errno" which will be set during a function
(like fork) that was unable to successfully
complete its task. */
                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;
    }
}

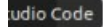
```

When we type the command `cat prog.c` ,we get the lines of the files inside

```
abid@ubuntu:~/6.3$ make exec
./prog
os>ps -ael
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
4 S 0 1 0 0 80 0 - 41898 - ? 00:00:03 systemd
1 S 0 2 0 0 80 0 - 0 - ? 00:00:00 kthreadd
1 I 0 3 2 0 60 -20 - 0 - ? 00:00:00 rcu_gp
1 I 0 4 2 0 60 -20 - 0 - ? 00:00:00 rcu_par_gp
1 I 0 6 2 0 60 -20 - 0 - ? 00:00:00 kworker/0:0H-events_highpri
1 I 0 9 2 0 60 -20 - 0 - ? 00:00:00 mm_percpu_wq
1 S 0 10 2 0 80 0 - 0 - ? 00:00:00 rcu_tasks_rude
1 S 0 11 2 0 80 0 - 0 - ? 00:00:00 rcu_tasks_trace
1 S 0 12 2 0 80 0 - 0 - ? 00:00:00 ksoftirqd/0
1 I 0 13 2 0 80 0 - 0 - ? 00:00:12 rcu_sched
1 S 0 14 2 0 -40 - - 0 - ? 00:00:00 migration/0
1 S 0 15 2 0 9 - - 0 - ? 00:00:00 idle_inject/0
1 S 0 16 2 0 80 0 - 0 - ? 00:00:00 cpuhp/0
1 S 0 17 2 0 80 0 - 0 - ? 00:00:00 cpuhp/1
1 S 0 18 2 0 9 - - 0 - ? 00:00:00 idle_inject/1
1 S 0 19 2 0 -40 - - 0 - ? 00:00:00 migration/1
1 S 0 20 2 0 80 0 - 0 - ? 00:00:00 ksoftirqd/1
1 I 0 22 2 0 60 -20 - 0 - ? 00:00:00 kworker/1:0H-events_highpri
5 S 0 23 2 0 80 0 - 0 - ? 00:00:00 kdevtmpfs
1 I 0 24 2 0 60 -20 - 0 - ? 00:00:00 netns
1 I 0 25 2 0 60 -20 - 0 - ? 00:00:00 inet_frag_wq
1 S 0 26 2 0 80 0 - 0 - ? 00:00:00 kauditd
1 S 0 28 2 0 80 0 - 0 - ? 00:00:00 khungtaskd
1 S 0 29 2 0 80 0 - 0 - ? 00:00:00 oom_reaper
1 I 0 30 2 0 60 -20 - 0 - ? 00:00:00 writeback
1 S 0 31 2 0 80 0 - 0 - ? 00:00:00 kcompactd0
1 S 0 32 2 0 85 5 - 0 - ? 00:00:00 ksm
1 S 0 33 2 0 99 19 - 0 - ? 00:00:00 khugepaged
1 I 0 80 2 0 60 -20 - 0 - ? 00:00:00 kintegrityd
1 I 0 81 2 0 60 -20 - 0 - ? 00:00:00 kblockd
1 I 0 82 2 0 60 -20 - 0 - ? 00:00:00 blkcg_punt_bio
1 I 0 83 2 0 60 -20 - 0 - ? 00:00:00 tpm_dev_wq
1 I 0 84 2 0 60 -20 - 0 - ? 00:00:00 ata_sff
1 I 0 85 2 0 60 -20 - 0 - ? 00:00:00 md
1 I 0 86 2 0 60 -20 - 0 - ? 00:00:00 edac-poller
1 I 0 87 2 0 60 -20 - 0 - ? 00:00:00 devfreq_wq
1 S 0 88 2 0 9 - - 0 - ? 00:00:00 watchdogd
1 I 0 90 2 0 60 -20 - 0 - ? 00:00:00 kworker/0:1H-kblockd
1 S 0 92 2 0 80 0 - 0 - ? 00:00:00 kswapd0
1 S 0 93 2 0 80 0 - 0 - ? 00:00:00 ecryptfs-kthrea
1 I 0 95 2 0 60 -20 - 0 - ? 00:00:00 kthrotld
1 S 0 96 2 0 9 - - 0 - ? 00:00:00 irq/24-pciehp
1 S 0 97 2 0 9 - - 0 - ? 00:00:00 irq/25-pciehp
1 S 0 98 2 0 9 - - 0 - ? 00:00:00 irq/26-pciehp
1 S 0 99 2 0 9 - - 0 - ? 00:00:00 irq/27-pciehp
1 S 0 100 2 0 9 - - 0 - ? 00:00:00 irq/28-pciehp
1 S 0 101 2 0 9 - - 0 - ? 00:00:00 irq/29-pciehp
1 S 0 102 2 0 9 - - 0 - ? 00:00:00 irq/30-pciehp
1 S 0 103 2 0 9 - - 0 - ? 00:00:00 irq/31-pciehp
1 S 0 104 2 0 9 - - 0 - ? 00:00:00 irq/32-pciehp
1 S 0 105 2 0 9 - - 0 - ? 00:00:00 irq/33-pciehp
1 S 0 106 2 0 9 - - 0 - ? 00:00:00 irq/34-pciehp
1 S 0 107 2 0 9 - - 0 - ? 00:00:00 irq/35-pciehp
1 S 0 108 2 0 9 - - 0 - ? 00:00:00 irq/36-pciehp
1 S 0 109 2 0 9 - - 0 - ? 00:00:00 irq/37-pciehp
1 S 0 110 2 0 9 - - 0 - ? 00:00:00 irq/38-pciehp
1 S 0 111 2 0 9 - - 0 - ? 00:00:00 irq/39-pciehp
1 S 0 112 2 0 9 - - 0 - ? 00:00:00 irq/40-pciehp
```

When we type the command `cat ps -ael`, we get the lines of the files inside





We can see that, this shell created by me exactly work like the Linux terminal. It can open files ,read files, open app, modify files and etc.

**Problems:**

I was confused with the Linux OS and how to use terminal and what to type. The linux keyword was unfamiliar with me.

**Solution:**

To solve those problems I looked for information in internet. In order to understand some questions and procedure I also asked the teacher to help me understand them. And provided instructions helped to solve some of my errors during the experiment.

**Conclusion:**

At the beginning, I was unfamiliar with Linux operation system and terminal. Gradually, reading lot of article and reading teachers ppt then I solved those problem one by one. In this experiment ,small helps and suggestions from the teacher was very helpful that saved my time.I enjoyed the practical and learned lot of interesting things.

**Attachments:**

- 1) ABID ALI\_2019380141\_OS(Lab 2).docx
- 2) ABID ALI\_2019380141\_OS(Lab 2).pdf
- 3)Code\_ABID ALI\_2019380141(Lab-2).zip