Name: Khushveen Kaur Umra
Section: 2
University ID: 947005108

Lab 2 Report

Summary:

For this specific lab, we ran multiple experiments to learn about how processes are created an executed. All of the experiments in this lab, allows the user to understand the relationship of the parents and the child processes. These lab experiments also allows us to understand how the kernel scheduler functions in linux.

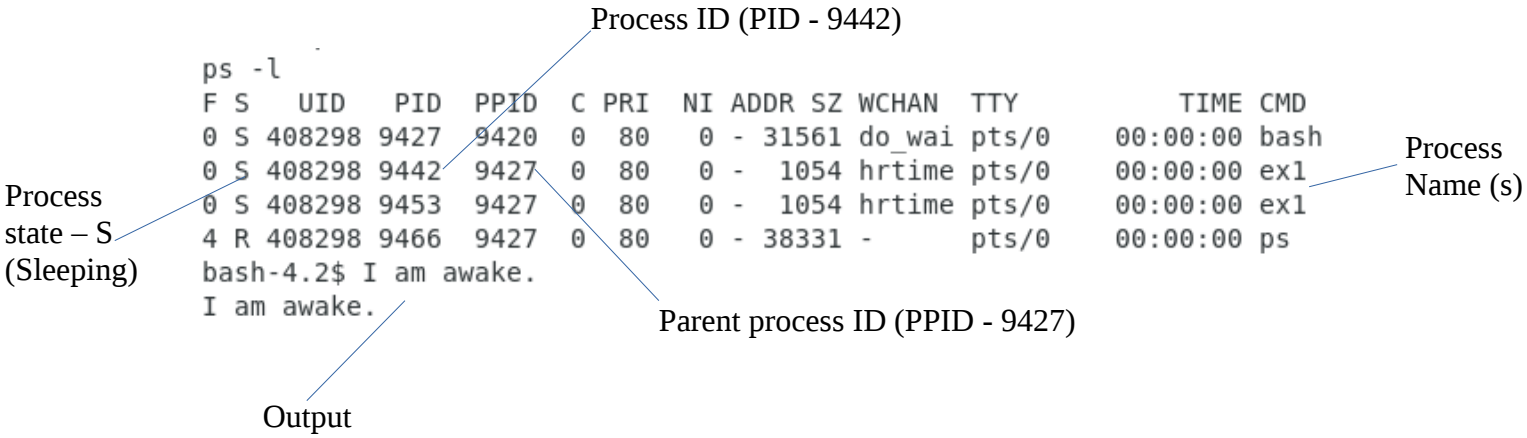
We also learned how to understand and evaluate the status of a unix process based on the letter status. The following Unix calls were used in this lab: getpid, getppid, sleep, fork, exec, wait, and kills.

Lab Questions:

3.1:

6pts In the report, include the relevant lines from “ps -l” for your programs, and point out the following items:

- output
- process name
- process state (decode the letter!)
- process ID (PID)
- parent process ID (PPID)



2pts Repeat this experiment and observe what changes and doesn't change.

```
./ex1 &
[3] 9590
[1]   Done                    ./ex1
[2]   Done                    ./ex1
bash-4.2$ Process ID is: 9590
Parent process ID is: 9427
./ex1 &
[4] 9595
bash-4.2$ Process ID is: 9595
Parent process ID is: 9427
ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 408298 9427  9420  0  80   0 - 31561 do_wai pts/0    00:00:00 bash
0 S 408298 9590  9427  0  80   0 -  1054 hrtime pts/0    00:00:00 ex1
0 S 408298 9595  9427  0  80   0 -  1054 hrtime pts/0    00:00:00 ex1
4 R 408298 9598  9427  0  80   0 - 38331 -      pts/0    00:00:00 ps
bash-4.2$ I am awake.
I am awake.
```

Once you repeat this experiment, the PID of the init process does not change.
That is, the UID, PID, and the PPID of the bash process (init) does not change, even after repeating the experiment. Also, the PPID of the program (ex1), remains the same throughout the initial and the repeated experiment.

Although, the only change that can be seen is the PID's of the executed program (ex1) after repeating the experiment. (Screenshot attached for reference)

2pts Find out the name of the process that started your programs. What is it, and what does it do?

The PID of the process that started the program initially is "9442", and the PID of the process that started the program during the second experiment is "9590". The names for both the processes were "ex1", as seen under the CMD column.

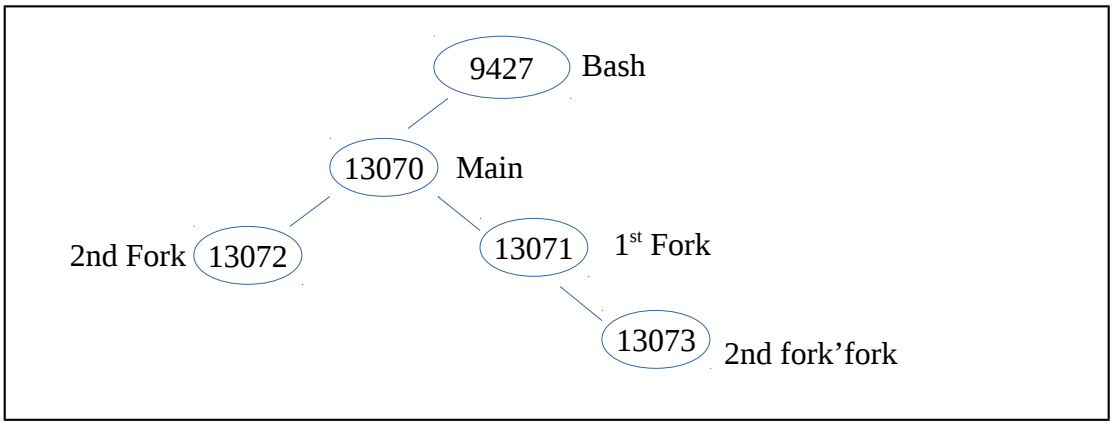
The PID / process is running on the bash kernel.

3.2:

1pt Include the output from the program

```
[2] 13070
bash-4.2$ Process 13070's parent process ID is 9427
Process 13072's parent process ID is 13070
Process 13071's parent process ID is 13070
Process 13073's parent process ID is 13071
```

4pts Draw the process tree (label processes with PIDs)



3pts Explain how the tree was built in terms of the program code

In the tree, we can see that the total number of processes are 4. This is because we call fork() only twice. The first fork command creates the first child for the "Main" parent process, which has a PID of " 13071". The second fork command creates two children, one for the "Main" parent, with a PID of "13072", and another one for the first fork command, with a PID of "13073". This is how the tree was built in terms of the program code.

```
int main() {
fork();
fork();
usleep(1);
printf("Process %d's parent process ID is %d\n", getpid(), getppid());
sleep(2);
return 0;
}
```

2pts Explain what happens when the sleep statement is removed. You should see processes reporting a parent PID of 1. Redirecting output to a file may interfere with this, and you may need to run the program multiple times to see it happen.

When the processes reported a parent PID of 1, it means that the parent process has already completed running and exited by the time the third child checked for the parent's PID. This caused the child to be re-parented under init, due to which the processes reporteed a parent PID of 1. This in total is what happens when the sleep statement is removed.

```
bash-4.2$ ./ex2 &
[1] 15604
bash-4.2$ Process 15604's parent process ID is 15339
Process 15606's parent process ID is 1
Process 15605's parent process ID is 1
Process 15607's parent process ID is 15605
```

3.3:

2pts Include the (completed) program and its output

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

int main () {
int ret;
ret = fork();

printf("The ret value is: %d\n", ret);

if(ret == 0)
{
        /* this is the child process */
printf("The child process ID is %d\n", getpid());
printf("The child's parent process ID is %d\n", getppid());
}
else {
        /*this is the paremt process */
printf("The parent process ID is %d\n", getpid());
printf("The parent's parent process ID is %d\n", getppid());
}
sleep(2);
return 0;
}
```

```
bash-4.2$ The ret value is: 14216
The parent process ID is 14215
The parent's parent process ID is 13021
The ret value is: 0
The child process ID is 14216
The child's parent process ID is 14215
```

4pts Speculate why it might be useful to have fork return different values to the parent and child. What advantage does returning the child's PID have?

It is useful to have fork return different values to the parent and the child because it helps the user to identify which value is for the parent, and which value is for the child. The value of PID and PPID is useful for keeping track of the child and the parent, which fork() runs the same code in different threads.

3.4:

2pts Include small (but relevant) sections of the output

4pts Make some observations about time slicing. Can you find any output that appears to have been cut off? Are there any missing parts? What's going on (mention the kernel scheduler)?

Time slicing is the effect of a short interval of time allotted to each program in a multitasking system, which is typically in milliseconds.

In linux, time slicing is decided by the kernal scheduler which handles the CPU allocation for executing purposes and aims to maximize overall CPU utilization.

3.5:

6pts Explain the major difference between this experiment and experiment 4. Be sure to look up what wait does (*man 2 wait*).

The wait(NULL) command in the code blocks the parent process until any of its children has finished. If the child terminates before parent process reaches wait(NULL), then the child process turns to a Z (zombie) process until the parent waits and is released from memory. Hence, the wait command places the parent process on hold until its child is finished running. This resolves the issue of time slicing which is seen in the 4th experiment. The output will be the child loop finishing 500000 iterations and only then will the parent process resume.

```
Child: 499996
Child: 499997
Child: 499998
Child: 499999
Child ends
Parent starts
Parent: 0
Parent: 1
Parent: 2
Parent: 3
Parent: 4
Parent: 5
Parent: 6
Parent: 7
Parent: 8
Parent: 9
```

3.6:

2pts The program appears to have an infinite loop. Why does it stop?

In this specific program, each iteration of the loop takes 1/100 of a second, and the parent is asleep of 10 seconds. So that means that the child program will only be able to execute for about 1000 iterations before the process ends. The program appears to have an infinite loop only in the child process. The parent process terminates the child, which eventually stops the loop.

4pts From the definitions of *sleep* and *usleep*, what do you expect the child's count

to be just before it ends?

In this specific program, each iteration of the loop takes 1/100 of a second, and the parent is asleep of 10 seconds. So that means that the child program will only be able to execute for about 1000 iterations before the process ends. Although, the child's count just before it ends is about 828.

```
child at count 822
child at count 823
child at count 824
child at count 825
child at count 826
child at count 827
child at count 828
Child has been killed. Waiting for it...
Done.
```

2pts Why doesn't the child reach this count before it terminates?

The child process ends before 1000 iterations, because the program is spending the time on other commands and tasks, such as printing the output, and increment the value of "i" every loop. These multiple tasks take up the time till 10 seconds before the child could finish 1000 iterations.

3.7:

8pts Read the man page for *execl* and relatives (*man 3 exec*). Under what conditions is the *printf* statement executed? Why isn't it always executed?
(consider what would happen if you changed the program to execute something other than */bin/ls*.)

According to the manual, the exec() can be defined as a family of functions that replaces the current process image with a new process image. The exec system call is used to execute a file which is residing in an active process. The "execl" command is technically equivalent to the "ls" command which is used to display the files in the current working directory. The same way, it executed a Shell command without creating a new process, and instead it replaces the currently open Shell operation. In this code, since it created a new process, replacing the current one, it terminated the current program, due to what the printf is never executed.
Although, the printf statement will be only be executed with exec() throws an error.

3.8:

2pts What is the range of values returned from the child process?

The range of values returned from the child process is 7-255

2pts What is the range of values sent by child process and captured by the parent process?

2pts When do you think the return value would be useful?
Hint: look at the commands *true* and *false*.